开发人员指南

Amazon API Gateway

Amazon API Gateway: 开发人员指南

Table of Contents

什么是 Amazon API Gateway？

Amazon API Gateway 是一项AWS服务，用于创建、发布、维护、监控和保护任意规模的 REST、HTTP 和 WebSocket API。API 开发人员可以创建能够访问 AWS 或其他 Web 服务以及存储在 AWS 云中的数据的 API。作为 API Gateway API 开发人员，您可以创建 API 以在您自己的客户端应用程序中使用。或者，您可以将您的 API 提供给第三方应用程序开发人员。有关更多信息，请参阅 the section called "谁使用 API Gateway？"。

API Gateway 创建符合下列条件的 RESTful API：

- 基于 HTTP 的。
- 启用无状态客户端-服务器通信。
- 实施标准 HTTP 方法例，如 GET、POST、PUT、PATCH 和 DELETE。

有关 API Gateway REST API 和 HTTP API 的更多信息，请参阅 the section called "在 REST API 和 HTTP API 之间选择 "、使用 HTTP API、the section called "使用 API Gateway 创建 REST API" 和 the section called "开发"。

API Gateway 创建以下 WebSocket API：

- 遵守 WebSocket 协议，从而支持客户端和服务器之间的有状态的全双工通信。
- 基于消息内容路由传入的消息。

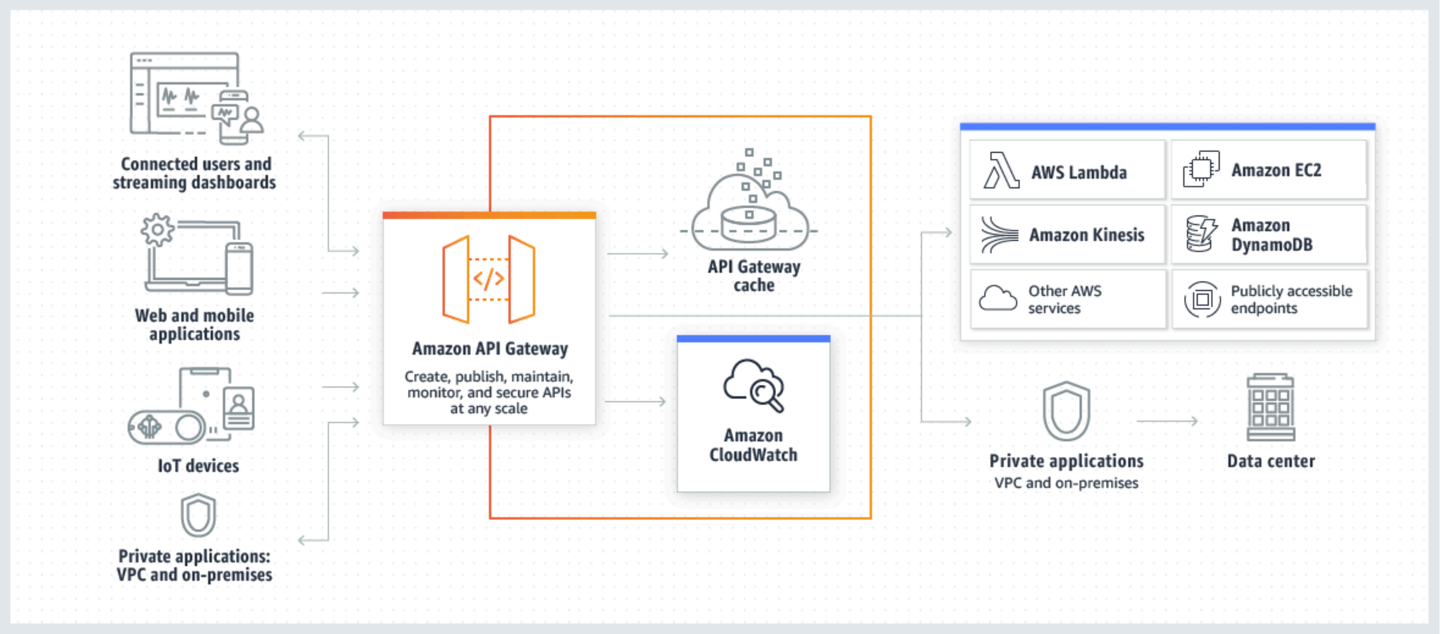
有关 API Gateway WebSocket API 的更多信息，请参阅 the section called "使用 API Gateway 创建 WebSocket API"和 the section called "关于 WebSocket API"。

主题

- 在 REST API 和 HTTP API 之间选择
- 开始使用 REST API 控制台

API Gateway 的架构

下图显示 API Gateway 架构。



此图表说明您在 Amazon API Gateway 中构建的 API 如何为您或开发人员客户提供用于构建AWS无服务器应用程序的集成且一致的开发人员体验。API Gateway 将处理涉及接受和处理多达几十万个并发 API 调用的所有任务。这些任务包括流量管理、授权和访问控制、监控以及 API 版本管理。

API Gateway 充当应用程序从后端服务访问数据、业务逻辑或功能的“前门”，例如，在 Amazon Elastic Compute Cloud (Amazon EC2) 上运行的负载、在 AWS Lambda 上运行的代码、任何 Web 应用程序或实时通信应用程序。

API Gateway 的功能

Amazon API Gateway 提供如下功能：

- 支持有状态 (WebSocket) 和无状态（HTTP 和 REST）API。
- 强大且灵活的身份验证机制，如 AWS Identity and Access Management 策略、Lambda 授权方函数和 Amazon Cognito 用户池。
- 用以安全地推出更改的金丝雀版本部署。

- CloudTrail 记录和监控 API 使用情况和 API 更改。
- CloudWatch 访问日志记录和执行日志记录，包括设置警报的能力。有关更多信息，请参阅the section called “CloudWatch 指标” 和 the section called “指标”。
- 能够使用 AWS CloudFormation 模板以支持创建 API 有关更多信息，请参阅 Amazon API Gateway 资源类型参考和 Amazon API Gateway V2 资源类型参考。
- 支持自定义域名。
- 与 AWS WAF 集成，以保护您的 API 免遭常见 Web 漏洞的攻击。
- 与 AWS X-Ray 集成，以了解和分类性能延迟。

有关 API Gateway 功能版本的完整列表，请参阅文档历史记录。

API Gateway 使用案例

主题

- 使用 API Gateway 创建 REST API
- 使用 API Gateway 创建 HTTP API
- 使用 API Gateway 创建 WebSocket API
- 谁使用 API Gateway？

使用 API Gateway 创建 REST API

API Gateway REST API 由资源和方法组成。资源是一种逻辑实体，应用程序可以通过资源路径来访问资源。方法与您的 API 用户提交的 REST API 请求以及返回给该用户的相应响应对应。

例如，`/incomes` 可以是代表应用程序用户收入的资源路径。一个资源可以包含一个或多个由适当的 HTTP 动词 (如 GET、POST、PUT、PATCH 和 DELETE) 定义的操作。资源路径和操作的组合构成 API 的方法。例如，`POST /incomes` 方法可以添加调用方获得的收入，`GET /expenses` 方法可以查询报告的调用方支出。

应用程序不需要知道在后端所请求数据的存储位置和提取位置。在 API Gateway REST API 中，前端由方法请求 和方法响应 封装。API 通过集成请求和集成响应与后端连接。

例如，使用 DynamoDB 作为后端，API 开发人员会设置集成请求以便将传入方法请求转发到所选的后端。该设置包括适当 DynamoDB 操作的规范、所需的 IAM 角色和策略以及所需的输入数据转换。后端将结果作为集成响应返回到 API Gateway。

要将与指定 HTTP 状态代码的适当方法响应对应的集成响应路由到客户端，您可以配置集成响应，将所需的响应参数从集成映射到方法。然后，您可以根据需要将后端的输出数据格式转换为前端的输出数据格式。API Gateway 让您能够为负载定义一个架构或模型，以方便设置正文映射模板。

API Gateway 提供 REST API 管理功能，如下所示：

- 支持使用 API Gateway 对 OpenAPI 的扩展生成开发工具包和创建 API 文档
- HTTP 请求的限制

使用 API Gateway 创建 HTTP API

使用 HTTP API，您可以创建比 REST API 具有更低延迟和更低成本的 RESTful API。

您可以使用 HTTP API 将请求发送到 AWS Lambda 函数或任何可公开路由的 HTTP 终端节点。

例如，您可以创建一个与后端上的 Lambda 函数集成的 HTTP API。当客户端调用您的 API 时，API Gateway 将请求发送到 Lambda 函数并将该函数的响应返回给客户端。

HTTP API 支持 OpenID Connect 和 OAuth 2.0 授权。它们内置了对跨域资源共享 (CORS) 和自动部署的支持。

要了解更多信息，请参阅“the section called “在 REST API 和 HTTP API 之间选择 ””。

使用 API Gateway 创建 WebSocket API

在 WebSocket API 中，客户端和服务器都可以随时相互发送消息。后端服务器可以轻松地将数据推送到连接的用户和设备，从而无需实施复杂的轮询机制。

例如，您可以使用 API Gateway WebSocket API 和 AWS Lambda 来构建无服务器应用程序，以便在聊天室中向个人用户或用户组发送消息以及从个人用户或用户组接收消息。或者，您可以根据消息内容调用后端服务，例如 AWS Lambda、Amazon Kinesis 或 HTTP 终端节点。

您可以使用 API Gateway WebSocket API 来构建安全的实时通信应用程序，而无需预置或管理任何服务器来管理连接或大规模数据交换。目标使用案例包括实时应用程序，如下所示：

- 聊天应用程序
- 实时控制面板，如股票代码
- 实时提醒和通知

API Gateway 提供 WebSocket API 管理功能，如下所示：

- 连接和消息的监控和限制
- 使用 AWS X-Ray 在消息经由 API 传递到后端服务时跟踪消息
- 易于与 HTTP/HTTPS 终端节点集成

谁使用 API Gateway？

使用 API Gateway 的开发人员有两种：API 开发人员和应用程序开发人员。

API 开发人员创建和部署 API，以便启用 API Gateway 中所需的功能。API 开发人员必须是拥有 API 的 AWS 账户中的用户。

应用程序开发人员通过在 API Gateway 中调用由 API 开发人员创建的 WebSocket 或 REST API 来构建一个正常运行的应用程序以调用AWS服务。

应用程序开发人员是 API 开发人员的客户。应用程序开发人员不需要拥有 AWS 账户，前提是 API 不需要 IAM 权限或通过 Amazon Cognito 用户池联合身份 所支持的第三方联合身份提供商来支持用户授权。此类身份提供商包括 Amazon、Amazon Cognito 用户池、Facebook 和 Google。

创建和管理 API Gateway API

API 开发人员使用名为 `apigateway` 的用于 API 管理的 API Gateway 服务组件来创建、配置和部署 API。

作为 API 开发人员，您可以按照开始使用 API Gateway中所述使用 API Gateway 控制台或者通过调用 API 参考 来创建和管理 API。这一 API 有若干种调用方式。这包括使用 AWS Command Line Interface (AWS CLI)，或使用 AWS 开发工具包。此外，您还可以使用 AWS CloudFormation 模板或（如果是 REST API 和 HTTP API）使用基于 OpenAPI 的 API Gateway 扩展 来支持创建 API。

有关提供 API Gateway 的区域以及相关控制服务终端节点的列表，请参阅 Amazon API Gateway 终端节点和配额。

调用 API Gateway API

应用程序开发人员使用名为 `execute-api` 的用于 API 执行的 API Gateway 服务组件来调用在 API Gateway 中创建或部署的 API。底层编程实体由创建的 API 公开。此类 API 有若干种调用方式。要了解更多信息，请参阅在 Amazon API Gateway 中调用 REST API和调用 WebSocket API。

访问 API Gateway

您可以通过以下方式访问 Amazon API Gateway：

- AWS Management Console – AWS Management Console提供用于创建和管理 API 的 Web 界面。完成先决条件中的步骤后，您可以访问 API Gateway 控制台：https://console.aws.amazon.com/apigateway。
- AWS开发工具包 – 如果您使用AWS为其提供开发工具包的编程语言，可以使用开发工具包访问 API Gateway。开发工具包可简化身份验证、与您的开发环境轻松集成，并有助于访问 API Gateway 命令。有关更多信息，请参阅用于 Amazon Web Services 的工具。
- API Gateway V1 和 V2 API – 如果您使用的是开发工具包不适用的编程语言，请参阅 Amazon API Gateway 版本 1 API 参考和 Amazon API Gateway 版本 2 API 参考。
- AWS Command Line Interface – 有关更多信息，请参阅 AWS Command Line Interface 用户指南中的开始设置 AWS Command Line Interface。
- AWS Tools for Windows PowerShell – 有关更多信息，请参阅 AWS Tools for Windows PowerShell 用户指南中的设置 AWS Tools for Windows PowerShell。

AWS 无服务器基础设施的一部分

API Gateway 与 AWS Lambda 共同构成AWS无服务器基础设施中面向应用程序的部分。要了解有关无服务器入门的更多信息，请参阅无服务器开发人员指南。

对于调用公开可用的AWS服务的应用程序，您可以使用 Lambda 与所需的服务进行交互，或在 API Gateway 中通过 API 方法公开 Lambda 函数。AWS Lambda 在高度可用的计算基础设施上运行您的代码。它会进行必要的计算资源执行和管理工作。为了启用无服务器应用程序，API Gateway 支持与 AWS Lambda 以及 HTTP 终端节点之间的简化代理集成。

了解如何开始使用 Amazon API Gateway

有关 Amazon API Gateway 的简介，请参阅以下内容：

- 开始使用，提供了创建 HTTP API 的演练步骤。
- 无服务器 land，提供了教学视频。
- 快乐的小 API 短视频，这是一系列简短的教学视频。

Amazon API Gateway 概念

API Gateway

API Gateway 是一项AWS服务，支持以下操作：

- 创建、部署和管理 RESTful 应用程序编程接口 (API) 以便公开后端 HTTP 终端节点、AWS Lambda 函数或其他 AWS 服务。
- 创建、部署和管理 WebSocket API 以公开 AWS Lambda 函数或其他AWS服务。
- 通过前端 HTTP 和 WebSocket 终端节点调用公开的 API 方法。

API Gateway REST API

与后端 HTTP 终端节点、Lambda 函数或其他AWS服务集成的 HTTP 资源与方法的集合。您可以在一个或多个阶段部署中此集合。通常情况下，根据应用程序逻辑将 API 资源组织成资源树形式。每个 API 资源均可公开一个或多个 API 方法，这些方法具有受 API Gateway 支持的唯一 HTTP 命令动词。有关更多信息，请参阅 the section called “在 REST API 和 HTTP API 之间选择 ”。

API Gateway HTTP API

与后端 HTTP 终端节点或 Lambda 函数集成的路由和方法的集合。您可以在一个或多个阶段部署中此集合。每个路由均可公开一个或多个 API 方法，这些方法具有受 API Gateway 支持的唯一 HTTP 命令动词。有关更多信息，请参阅 the section called “在 REST API 和 HTTP API 之间选择 ”。

API Gateway WebSocket API

与后端 HTTP 终端节点、Lambda 函数或其他AWS服务集成的 WebSocket 路由和路由键的集合。您可以在一个或多个阶段部署中此集合。API 方法通过可以与注册的自定义域名关联的前端 WebSocket 连接进行调用。

API 部署

API Gateway API 的时间点快照。要使客户端可用，必须将部署与一个或多个 API 阶段关联。

API 开发人员

拥有 API Gateway 部署的AWS账户（例如，也支持编程访问的服务提供商。）

API 终端节点

API Gateway 中部署到特定区域的 API 的主机名。主机名的格式是 `{api-id}.execute-api.{region}.amazonaws.com`。支持以下类型的 API 终端节点：

- 边缘优化的 API 终端节点
- 私有 API 终端节点
- 区域 API 终端节点

API 密钥

API Gateway 用于识别使用 REST 或 WebSocket API 的应用程序开发人员的字母数字字符串。API Gateway 可以代表您生成 API 密钥，也可以从 CSV 文件中导入 API 密钥。您可以将 API 密钥与 Lambda 授权方或使用计划一起使用来控制对 API 的访问。

请参阅 API 终端节点。

API 所有者

请参阅 API 开发人员。

API 阶段

对您的 API 生命周期状态（例如，“dev”、“prod”、“beta”、“v2”）的逻辑引用。API 阶段由 API ID 和阶段名称标识。

应用程序开发人员

应用程序创建者，可能拥有也可能不拥有 AWS 账户并与您（API 开发人员）部署的 API 交互。应用程序开发人员是您的客户。应用程序开发人员通常由 API 密钥标识。

回调 URL

当新客户端通过 WebSocket 连接进行连接时，您可以调用 API Gateway 中的集成来存储客户端的回调 URL。然后，您可以使用该回调 URL 从后端系统向客户端发送消息。

开发人员门户

一个能让您的客户注册、发现和订阅 API 产品（API Gateway 使用计划），管理其 API 密钥以及查看 API 使用指标的应用程序。

边缘优化的 API 终端节点

API Gateway API 的默认主机名，它部署到指定区域，并使用 CloudFront 分配以方便客户端跨AWS区域进行常规访问。API 请求将路由至最近的 CloudFront 接入点 (POP)，通常可改进不同地理位置客户端的连接时间。

请参阅 API 终端节点。

集成请求

API Gateway 中的 WebSocket API 路由或 REST API 方法的内部接口，在其中您会将路由请求的正文或方法请求的参数和正文映射到后端所需的格式。

集成响应

API Gateway 中的 WebSocket API 路由或 REST API 方法的内部接口，在其中您会将从后端接收到的状态代码、标头和负载映射到返回到客户端应用程序的响应格式。

映射模板

使用 Velocity 模板语言 (VTL) 表示的脚本，此脚本用于将请求正文从前端数据格式转换为后端数据格式，或将响应正文从后端数据格式转换为前端数据格式。映射模板可以在集成请求中或在集成响应中指定。它们可以将运行时提供的数据引用为上下文和阶段变量。

映射可以像身份转换一样简单，它通过集成将标头或正文从客户端按原样传递到请求的后端。对于响应也是如此，在其中负载从后端传递到客户端。

方法请求

API Gateway 中 API 方法的公共接口，定义了应用程序开发人员在通过 API 访问后端时必须在请求中发送的参数和正文。

方法响应

REST API 的公共接口，定义应用程序开发人员期望在 API 的响应中收到的状态代码、标头和正文模型。

模拟集成

在模拟集成中，API 响应直接从 API Gateway 生成，而无需集成后端。作为 API 开发人员，您可以决定 API Gateway 响应模拟集成的方式。为此，您可以配置方法的集成请求和集成响应，以将响应与给定的状态代码相关联。

模型

指定请求或响应负载的数据结构的数据架构。生成 API 的强类型的开发工具包时需要使用模型。它还用于验证负载。模型可以方便地用于生成示例映射模板，以便开始创建生产映射模板。虽然模型很有用，但不是创建映射模板所必需的。

私有 API

请参阅私有 API 终端节点。

私有 API 终端节点

一个通过接口 VPC 终端节点公开的 API 终端节点，能让客户端安全地访问 VPC 内的私有 API 资源。私有 API 与公有 Internet 隔离，只能使用已授予访问权限的 API Gateway 的 VPC 终端节点访问它们。

私有集成

一种 API Gateway 集成类型，供客户端通过私有 REST API 终端节点访问客户 VPC 中的资源，而不向公有 Internet 公开资源。

代理集成

简化的 API Gateway 集成配置。您可以将代理集成设置为 HTTP 代理集成或 Lambda 代理集成。

对于 HTTP 代理集成，API Gateway 在前端与 HTTP 后端之间传递整个请求和响应。对于 Lambda 代理集成，API Gateway 将整个请求作为输入发送到后端 Lambda 函数。API Gateway 随后将 Lambda 函数输出转换为前端 HTTP 响应。

在 REST API 中，代理集成最常与代理资源一起使用，以“贪婪”路径变量（例如 `{proxy+}`）与“捕获所有”ANY 方法相结合的方式表示。

快速创建

您可以使用快速创建来简化 HTTP API 的创建。使用“快速创建”可以创建具有 Lambda 或 HTTP 集成、默认“捕获全部”路由和默认阶段（配置为自动部署更改）的 API。有关更多信息，请参阅 the section called “使用AWS CLI 创建 HTTP API”。

区域 API 终端节点

部署到指定区域并旨在服务于同一 AWS 区域中的客户端（例如 EC2 实例）的 API 的主机名。API 请求直接以区域特定的 API Gateway API 为目标而不经过任何 CloudFront 分配。对于区域内请求，区域性终端节点会绕过到 CloudFront 分配的不必要往返行程。

此外，您可以在区域性终端节点上应用基于延迟的路由，从而使用相同的区域性 API 终端节点配置将 API 部署到多个区域，为每个已部署的 API 设置相同的自定义域名，以及在 Route 53 中配置基于延迟的 DNS 记录以将客户端请求路由到具有最低延迟的区域。

请参阅 API 终端节点。

路由

API Gateway 中的 WebSocket 路由用于根据消息内容将传入消息定向到特定集成，例如 AWS Lambda 函数。在定义 WebSocket API 时，指定路由键和集成后端。路由键是消息正文中的一种属性。当路由键在传入消息中匹配时，将会调用集成后端。

还可以为不匹配的路由键设置默认路由，或者设置默认路由来指定一个将消息按原样传递给执行路由和处理请求的后端组件的代理模型。

路由请求

API Gateway 中 WebSocket API 方法的公共接口，定义了应用程序开发人员在通过 API 访问后端时必须在请求中发送的正文。

路由响应

WebSocket API 的公共接口，定义应用程序开发人员期望从 API Gateway 收到的状态代码、标头和正文模型。

使用计划

使用计划可以提供能够访问一个或多个部署的 REST 或 WebSocket API 的选定 API 客户端。您可以通过使用计划来配置限制和配额限制，这些限制会应用到单独的客户端 API 密钥。

WebSocket 连接

API Gateway 在客户端和 API Gateway 本身之间维护持久连接。API Gateway 和后端集成（如 Lambda 函数）之间没有持久连接。基于从客户端收到的消息内容，根据需要来调用后端服务。

在 REST API 和 HTTP API 之间选择

REST API 和 HTTP API 都是 RESTful API 产品。REST API 支持的功能比 HTTP API 多，而 HTTP API 在设计时功能就极少，因此能够以更低的价格提供。如果您需要如 API 密钥、每客户端节流、请求验证、AWS WAF 集成或私有 API 端点等功能，请选择 REST API。如果您不需要 REST API 中包含的功能，请选择 HTTP API。

以下各节汇总了 REST API 和 HTTP API 中可用的核心功能。

端点类型

端点类型是指 API Gateway 为 API 创建的端点。有关更多信息，请参阅 the section called “API Gateway 端点类型”。

端点类型	REST API	HTTP API
边缘优化	✓	
区域性	✓	✓
私有	✓	

安全性

API Gateway 提供了多种方法来保护您的 API 免受某些威胁危害，例如恶意行为者或流量高峰。要了解更多信息，请参阅the section called “保护”和the section called “保护”。

安全功能	REST API	HTTP API
双向 TLS 身份验证	✓	✓
用于后端身份验证的证书	✓	
AWS WAF	✓	

授权

API Gateway 支持多种用于控制和管理对 API 的访问的机制：有关更多信息，请参阅the section called “访问控制” 和the section called “访问控制”。

授权选项	REST API	HTTP API
IAM	✓	✓
资源策略	✓	
Amazon Cognito	✓	✓ [1]
使用 AWS Lambda 函数的自定义授权	✓	✓
JSON Web 令牌 (JWT) [2]		✓

[1] 您可以将 Amazon Cognito 与 JWT 授权方结合使用。

[2] 您可以使用 Lambda 授权方以验证适用于 REST API 的 JWT。

API 管理

如果您需要 API 管理功能（例如 API 密钥和每客户端费率限制），请选择 REST API。有关更多信息，请参阅 the section called “分发”、the section called “自定义域名” 和 the section called “自定义域名”。

功能	REST API	HTTP API
自定义域	✓	✓
API 密钥	✓	
每客户端费率限制	✓	
每客户端使用量节流	✓	

开发

在开发 API Gateway API 时，您可以决定 API 的许多特征。这些特征取决于 API 的使用案例。有关更多信息，请参阅the section called “开发”和the section called “开发”。

功能	REST API	HTTP API
CORS 配置	✓	✓
测试调用	✓	
缓存	✓	
用户控制的部署	✓	✓
自动部署		✓
自定义网关响应	✓	
Canary 版本部署	✓	
请求验证	✓	

功能	REST API	HTTP API
请求参数转换	✓	✓
请求正文转换	✓	

监控

API Gateway 支持多种选项来记录 API 请求和监控 API。有关更多信息，请参阅the section called “显示器”和the section called “显示器”。

功能	REST API	HTTP API
Amazon CloudWatch 指标	✓	✓
访问 CloudWatch Logs 的日志	✓	✓
访问 Amazon Data Firehose 的日志	✓	
执行日志	✓	
AWS X-Ray 跟踪	✓	

集成

集成将 API Gateway API 连接到后端资源。有关更多信息，请参阅the section called “集成”和the section called “集成”。

功能	REST API	HTTP API
公有 HTTP 端点	✓	✓
AWS 服务	✓	✓
AWS Lambda 函数	✓	✓
与网络负载均衡器的私有集成	✓	✓

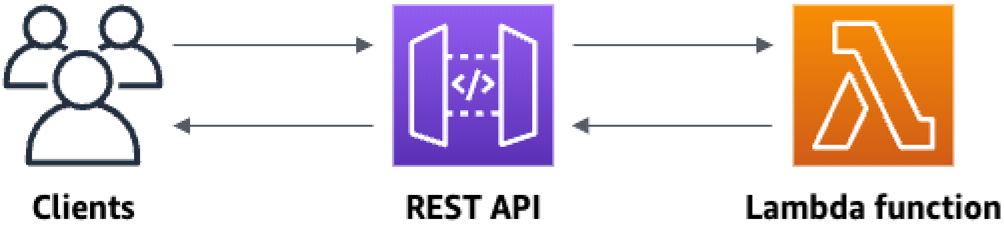
功能	REST API	HTTP API
与应用程序负载均衡器的私有集成		✓
与 AWS Cloud Map 的私有集成		✓
模拟集成	✓	

开始使用 REST API 控制台

在本入门练习中，您将使用 API Gateway REST API 控制台创建无服务器 REST API。无服务器 API 让您可以专注于应用程序，而不是花时间预置和管理服务器。本次练习应在 20 分钟内完成，并且可以使用AWS 免费套餐完成。

首先，您将使用 Lambda 控制台创建 Lambda 函数。接下来，使用 API Gateway REST API 控制台创建 REST API 。然后，您将创建一个 API 方法，并使用 Lambda 代理集成将该方法与 Lambda 函数集成。最后，您将部署并调用您的 API。

当您调用 REST API 时，API Gateway 会将请求路由到您的 Lambda 函数。Lambda 运行函数并将响应返回 API Gateway。然后 API Gateway 会向您返回该响应。



要完成本练习，您需要一个 AWS 账户以及一位具有控制台访问权限的 AWS Identity and Access Management (IAM) 用户。有关更多信息，请参阅 开始使用 API Gateway 的先决条件。

主题

- 步骤 1：创建 Lambda 函数
- 步骤 2：创建 REST API
- 步骤 3：创建 Lambda 代理集成
- 步骤 4：部署您的 API

- 步骤 5：调用您的 API
- （可选）第 6 步：清除

步骤 1：创建 Lambda 函数

将 Lambda 函数用作您的 API 后端。只有在需要时 Lambda 才运行您的代码，并且能自动扩展，从每天几个请求扩展到每秒数千个请求。

在本练习中，您使用 Lambda 控制台中的默认 Node.js 函数。

创建 Lambda 函数

1. 通过以下网址登录 Lambda 控制台：https://console.aws.amazon.com/lambda。
2. 选择创建函数。
3. 在基本信息 下，对于函数名称，输入 **`my-function`**。
4. 选择创建函数。

默认的 Lambda 函数代码应类似于以下内容：

```
export const handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('The API Gateway REST API console is great!'),
    };
    return response;
};
```

对于本练习，您可以修改 Lambda 函数，前提是函数的响应与 API Gateway 所要求的格式保持一致。

将默认响应正文 (`Hello from Lambda!`) 替换为 `The API Gateway REST API console is great!`。当您调用示例函数时，它会向客户端返回 `200` 响应以及更新的响应。

步骤 2：创建 REST API

接下来，您将使用根资源 (/) 创建一个 REST API。

创建 REST API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 请执行以下操作之一：
    - 要创建第一个 API，对于 REST API，请选择构建。
    - 如果您之前已经创建了 API，请选择创建 API，然后为 REST API 选择构建。
3. 对于 API 名称，请输入 **`my-rest-api`**。
4. （可选）对于描述，输入描述。
5. 将 API 端点类型设置保留为区域。
6. 选择创建 API。

步骤 3：创建 Lambda 代理集成

接下来，您将在根资源 (/) 上为您的 REST API 创建 API 方法，并使用代理集成将方法与您的 Lambda 函数集成。在 Lambda 代理集成中，API Gateway 将来自客户端的传入请求直接传递给 Lambda 函数。

创建 Lambda 代理集成

1. 选择 / 资源，然后选择创建方法。
2. 对于方法类型中，选择 ANY。
3. 对于集成类型，选择 Lambda。
4. 打开 Lambda 代理集成。
5. 对于 Lambda 函数，输入 **`my-function`**，然后选择您的 Lambda 函数。
6. 选择创建方法。

步骤 4：部署您的 API

接下来，您将创建 API 部署并将其与阶段关联。

部署 API

1. 选择部署 API。
2. 对于阶段，选择新建阶段。

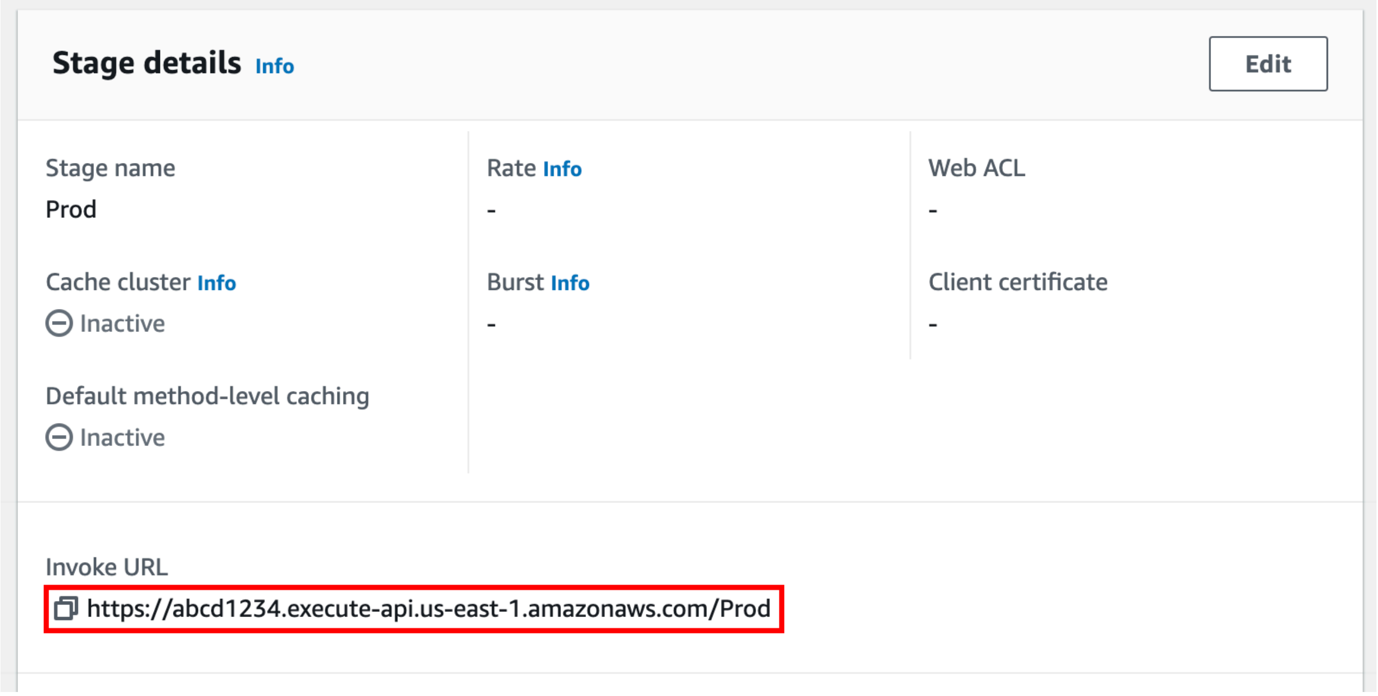
3. 对于阶段名称，输入 **`Prod`**。
4. （可选）对于描述，输入描述。
5. 选择部署。

现在，客户端可以调用您的 API。要在部署 API 之前对其进行测试，您可以选择 ANY 方法，导航到测试选项卡，然后选择测试。

步骤 5：调用您的 API

调用 API

1. 从主导航窗格中选择阶段。
2. 在阶段详细信息下，选择复制图标以复制您 API 的调用 URL。



Stage details Info

Stage name: Prod

Rate Info: -

Web ACL: -

Cache cluster Info: Inactive

Burst Info: -

Client certificate: -

Default method-level caching: Inactive

Invoke URL: https://abcd1234.execute-api.us-east-1.amazonaws.com/Prod

3. 在 Web 浏览器中输入调用 URL。

   完整的 URL 应类似于 `https://abcd123.execute-api.us-east-2.amazonaws.com/Prod`。

   您的浏览器向 API 发送 `GET` 请求。

4. 验证 API 的响应。您应该会在浏览器中看到文本 `"The API Gateway REST API console is great!"`。

（可选）第 6 步：清除

为避免对您的 AWS 账户产生不必要的成本，请删除您在本练习中创建的资源。以下步骤将删除您的 REST API、Lambda 函数和相关资源。

删除 REST API

1. 在资源窗格中，选择 API 操作，然后选择删除 API。
2. 在删除 API 对话框中，输入确认，然后选择删除。

删除 Lambda 函数

1. 通过以下网址登录 Lambda 控制台：https://console.aws.amazon.com/lambda。
2. 在函数页面上，选择您的函数。依次选择操作、删除。
3. 在删除 1 函数对话框中，输入 **`delete`**，然后选择删除。

删除 Lambda 函数的日志组

1. 在 Amazon CloudWatch 控制台中，打开日志组页面。
2. 在日志组页面上，选择函数的日志组 (`/aws/lambda/my-function`)。然后，对于操作，选择删除日志组。
3. 在 Delete log group(s)（删除日志组）对话框中，选择 Delete（删除）。

删除 Lambda 函数的执行角色

1. 打开 IAM 控制台的角色页面。
2. （可选）在角色页面的搜索框中，输入 **`my-function`**。
3. 选择函数的角色（例如，`my-function-`*`31exxmpl`*），然后选择删除。
4. 在删除 **`my-function-`*`31exxmpl`*** ? 对话框中，输入角色的名称，然后选择删除。

Tip

您可以使用 AWS CloudFormation 或 AWS Serverless Application Model (AWS SAM) 自动创建和清理 AWS 资源。有关示例 AWS CloudFormation 模板，请参阅 awsdocs GitHub 存储库中的 API Gateway 示例模板。

开始使用 API Gateway 的先决条件

首次使用 Amazon API Gateway 前，请完成以下任务。

注册 AWS 账户

如果您还没有 AWS 账户，请完成以下步骤来创建一个。

注册 AWS 账户

1. 打开 https://portal.aws.amazon.com/billing/signup。
2. 按照屏幕上的说明进行操作。

   在注册时，将接到一通电话，要求使用电话键盘输入一个验证码。

   当您注册 AWS 账户时，系统将会创建一个 AWS 账户根用户。根用户有权访问该账户中的所有 AWS 服务 和资源。作为安全最佳实践，请为用户分配管理访问权限，并且只使用根用户来执行需要根用户访问权限的任务。

注册过程完成后，AWS 会向您发送一封确认电子邮件。在任何时候，您都可以通过转至 https://aws.amazon.com/ 并选择我的账户来查看当前的账户活动并管理您的账户。

创建具有管理访问权限的用户

注册 AWS 账户 后，请保护好您的 AWS 账户根用户，启用 AWS IAM Identity Center，并创建一个管理用户，以避免使用根用户执行日常任务。

保护您的 AWS 账户根用户

1. 选择根用户并输入您的 AWS 账户电子邮件地址，以账户拥有者身份登录 AWS Management Console。在下一页上，输入您的密码。

   要获取使用根用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的以根用户身份登录。
2. 为您的根用户启用多重身份验证 (MFA)。

   有关说明，请参阅《IAM 用户指南》中的为 AWS 账户根用户启用虚拟 MFA 设备（控制台）。

创建具有管理访问权限的用户

1. 启用 IAM Identity Center

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的启用 AWS IAM Identity Center。
2. 在 IAM Identity Center 中，为用户授予管理访问权限。

   有关如何使用 IAM Identity Center 目录 作为身份源的教程，请参阅《AWS IAM Identity Center 用户指南》中的使用默认的 IAM Identity Center 目录 配置用户访问权限。

以具有管理访问权限的用户身份登录

- 要使用您的 IAM Identity Center 用户身份登录，请使用您在创建 IAM Identity Center 用户时发送到您的电子邮件地址的登录网址。

  要获取使用 IAM Identity Center 用户登录方面的帮助，请参阅《AWS 登录 用户指南》中的 登录 AWS 访问门户。

将访问权限分配给其他用户

1. 在 IAM Identity Center 中，创建一个权限集，该权限集遵循应用最低权限的最佳做法。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的创建权限集。
2. 将用户分配到一个组，然后为该组分配单点登录访问权限。

   有关说明，请参阅《AWS IAM Identity Center 用户指南》中的添加组。

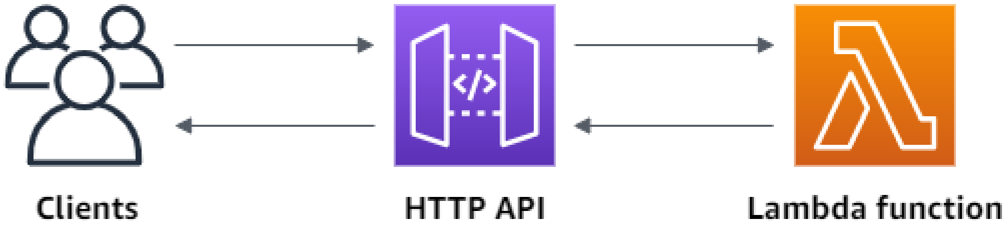
开始使用 API Gateway

在此入门练习中，您将创建一个无服务器 API。无服务器 API 让您可以专注于应用程序，而不是花时间预置和管理服务器。本次练习可在 20 分钟内完成，并且可以使用AWS免费套餐完成。

然后，使用 AWS Lambda 控制台创建 Lambda 函数。接下来，使用 API Gateway 控制台创建 HTTP API。然后，调用您的 API。

> ⓘ Note
>
> 本练习使用 HTTP API。API Gateway 还支持包含更多功能的 REST API。有关使用 REST API 的教程，请参阅the section called “开始使用 REST API 控制台”。
>
> 有关 HTTP API 与 REST API 之间差异的更多信息，请参阅the section called “在 REST API 和 HTTP API 之间选择 ”。

当您调用 HTTP API 时，API Gateway 会将请求路由到您的 Lambda 函数。Lambda 运行 Lambda 函数并将响应返回 API Gateway。然后 API Gateway 会向您返回响应。



要完成本练习，您需要一个AWS账户以及一位具有控制台访问权限的 AWS Identity and Access Management 用户。有关更多信息，请参阅 先决条件。

主题

- 步骤 1：创建 Lambda 函数
- 步骤 2：创建 HTTP API
- 步骤 3：测试您的 API
- （可选）步骤 4：清理
- 后续步骤

步骤 1：创建 Lambda 函数

将 Lambda 函数用作您的 API 后端。只有在需要时 Lambda 才运行您的代码，并且能自动扩展，从每天几个请求扩展到每秒数千个请求。

在本示例中，您可以使用 Lambda 控制台中的默认 Node.js 函数。

创建 Lambda 函数

1. 通过以下网址登录 Lambda 控制台：https://console.aws.amazon.com/lambda。
2. 选择创建函数。
3. 对于 Function name（函数名称），请输入 **`my-function`**。
4. 选择创建函数。

示例函数返回 `200` 响应到客户端，以及文本 `Hello from Lambda!`。

只要函数的响应与 API Gateway 所要求的格式保持一致，您就可以修改您的 Lambda 函数。

默认的 Lambda 函数代码应类似于以下内容：

```
export const handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

步骤 2：创建 HTTP API

接下来，创建一个 HTTP API。API Gateway 还支持 REST API 和 WebSocket API，但是 HTTP API 是本练习的最佳选择。REST API 比 HTTP API 支持更多功能，但我们在本练习中不需要这些功能。HTTP API 在设计时功能就极少，因此能够以更低的价格提供。WebSocket API 与客户端保持持久连接，以进行全双工通信，但这不是本示例所必需的。

HTTP API 为您的 Lambda 函数提供了 HTTP 终端节点。API Gateway 将请求路由到您的 Lambda 函数，然后将该函数的响应返回给客户端。

要创建 HTTP API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 请执行下列操作之一：
   - 要创建第一个 API，对于 HTTP API，请选择构建。
   - 如果您之前已经创建了 API，请选择创建 API，然后为 HTTP API 选择构建。
3. 对于集成，选择添加集成。
4. 选择 Lambda。
5. 对于 Lambda function（Lambda 函数），输入 **`my-function`**。
6. 对于 API Name (API 名称)，请输入 **`my-http-api`**。
7. 选择 Next (下一步)。
8. 查看 API Gateway 为您创建的路径，然后选择下一步。
9. 查看 API Gateway 为您创建的阶段，然后选择下一步。
10. 选择创建。

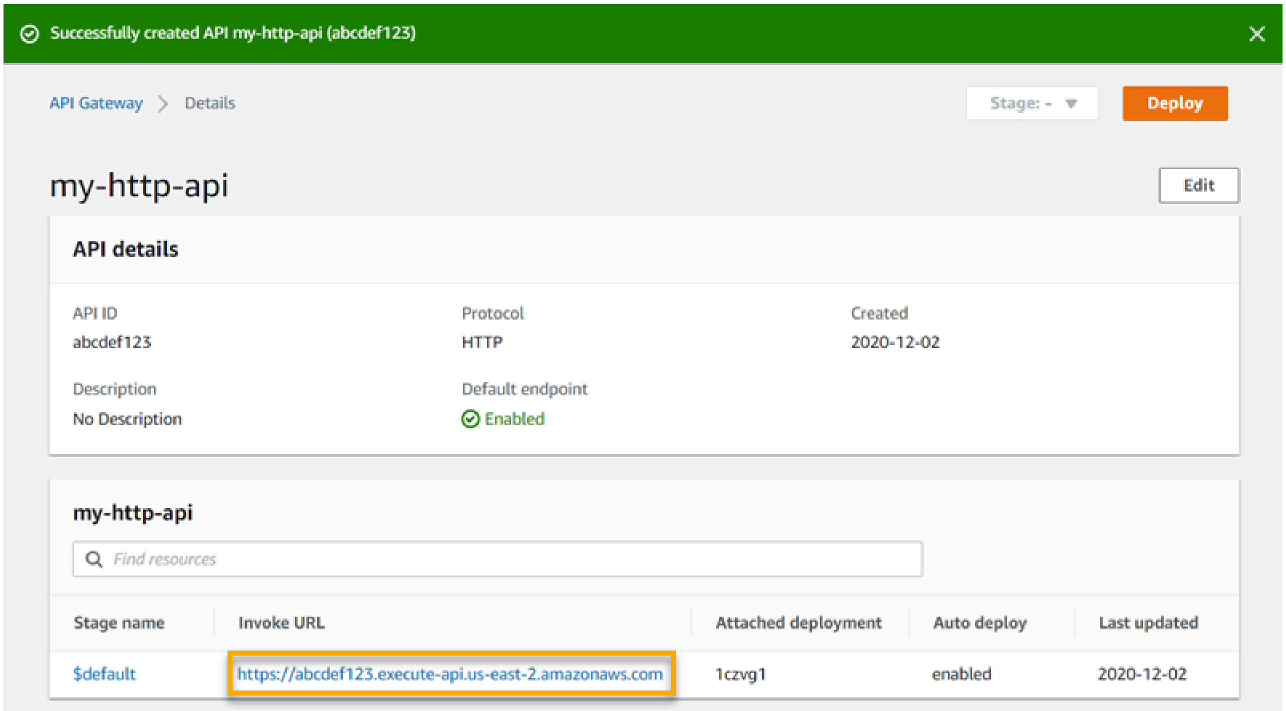
现在，您已经创建了一个集成了 Lambda 的 HTTP API，可以随时接收来自客户端的请求。

步骤 3：测试您的 API

接下来，测试您的 API 以确保它正常工作。为简单起见，请使用 Web 浏览器调用 API。

要测试您的 API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 请记下您的 API 的调用 URL。



Successfully created API my-http-api (abcdef123)

my-http-api

API details

API ID: abcdef123

Protocol: HTTP

Created: 2020-12-02

Description: No Description

Default endpoint: Enabled

my-http-api

Stage name	Invoke URL	Attached deployment	Auto deploy	Last updated
$default	https://abcdef123.execute-api.us-east-2.amazonaws.com	1czvg1	enabled	2020-12-02

4. 复制 API 的调用 URL，然后在 Web 浏览器中输入它。将 Lambda 函数的名称附加到调用您的 Lambda 函数的调用 URL 中。默认情况下，API Gateway 控制台创建与您的 Lambda 函数名称相同的路由，`my-function`。

   完整的 URL 应类似于 `https://`*`abcdef123`*`.execute-api.`*`us-east-2`*`.amazonaws.com/`*`my-function`*。

   您的浏览器向 API 发送 `GET` 请求。

5. 验证 API 的响应。您应该会在浏览器中看到文本 `"Hello from Lambda!"`。

（可选）步骤 4：清理

为避免不必要的成本，请删除作为本入门练习的一部分而创建的资源。以下步骤将删除 HTTP API、Lambda 函数和相关资源。

要删除 HTTP API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在 API 页面上，选择一个 API。选择操作，然后选择删除。
3. 选择删除。

要删除 Lambda 函数

1. 通过以下网址登录 Lambda 控制台：https://console.aws.amazon.com/lambda。
2. 在函数页面上，选择一个函数。选择操作，然后选择删除。
3. 选择删除。

要删除 Lambda 函数的日志组

1. 在 Amazon CloudWatch 控制台中，打开日志组页面。
2. 在日志组页面上，选择函数的日志组 (`/aws/lambda/my-function`)。选择操作，然后选择删除日志组。
3. 选择删除。

要删除 Lambda 函数的执行角色

1. 打开AWS Identity and Access Management控制台中的角色页面。
2. 选择函数的角色，例如 `my-function-`*`31exxmpl`*。
3. 选择删除角色。
4. 选择 Yes, delete (是，删除)。

您可以使用 AWS CloudFormation 或 AWS SAM 自动创建和清理AWS资源。如需示例 AWS CloudFormation 模板，请参阅示例 AWS CloudFormation 模板。

后续步骤

在本示例中，您使用AWS Management Console创建了一个简单的 HTTP API。该 HTTP API 调用 Lambda 函数并向客户端返回响应。

以下是继续使用 API Gateway 的后续步骤。

- 配置其他类型的 API 集成，包括：
  - HTTP 终端节点
  - VPC 中的私有资源，例如 Amazon ECS 服务
  - AWS 服务，例如 Amazon Simple Queue Service、AWS Step Functions 和 Kinesis Data Streams

- 控制对 API 的访问
- 为您的 API 启用日志记录
- 为 API 配置限制
- 为您的 API 配置自定义域

要从社区获取有关 Amazon API Gateway 的帮助，请参阅 API Gateway 开发论坛。当您进入此论坛时，AWS可能会要求您登录。

要直接从AWS获得 API Gateway 方面的帮助，请参阅AWS支持页面中的支持选项。

另请参阅我们的常见问题 (FAQ)，或者直接与我们联系。

Amazon API Gateway 教程和研讨会

以下教程和研讨会将提供有助于您了解 API Gateway 的实践练习。

REST API 教程

- 选择 AWS Lambda 集成教程
- 教程：通过导入示例创建 REST API
- 选择 HTTP 集成教程
- 教程：使用 API Gateway 私有集成构建 REST API
- 教程：使用AWS集成构建 API Gateway REST API
- 教程：通过两种 AWS 服务集成和一种 Lambda 非代理集成创建计算器 REST API
- 教程：在 API Gateway 中创建 REST API 作为 Amazon S3 代理
- 教程：在 API Gateway 中创建 REST API 作为 Amazon Kinesis 代理
- 教程：使用 AWS SDK 或 AWS CLI 设置边缘优化的 API
- 教程：构建私有 REST API

HTTP API 教程

- 教程：使用 Lambda 和 DynamoDB 构建 CRUD API
- 教程：构建具有私有集成到 Amazon ECS 服务的 HTTP API

WebSocket API 教程

- 教程：使用 WebSocket API、Lambda 和 DynamoDB 构建无服务器聊天应用程序

研讨会

- 构建无服务器 Web 应用程序
- 适用于无服务器应用程序的 CI/CD
- 无服务器安全研讨会
- 无服务器身份管理、身份验证和授权
- Amazon API Gateway 研讨会

Amazon API Gateway REST API 教程

以下教程将提供有助于您了解 API Gateway REST API 的实践练习。

主题

- 选择 AWS Lambda 集成教程
- 教程：通过导入示例创建 REST API
- 选择 HTTP 集成教程
- 教程：使用 API Gateway 私有集成构建 REST API
- 教程：使用AWS集成构建 API Gateway REST API
- 教程：通过两种 AWS 服务集成和一种 Lambda 非代理集成创建计算器 REST API
- 教程：在 API Gateway 中创建 REST API 作为 Amazon S3 代理
- 教程：在 API Gateway 中创建 REST API 作为 Amazon Kinesis 代理
- 教程：使用 AWS SDK 或 AWS CLI 设置边缘优化的 API
- 教程：构建私有 REST API

选择 AWS Lambda 集成教程

要使用 Lambda 集成构建 API，您可以使用 Lambda 代理集成或 Lambda 非代理集成。

使用 Lambda 代理集成，Lambda 函数的输入可以表示为请求标头、路径变量、查询字符串参数、正文和 API 配置数据的任意组合。您只需选择 Lambda 函数。API Gateway 将为您配置集成请求和集成响应。设置完成后，API 方法会发生变化，而不会修改现有设置。这是可能的，因为后端 Lambda 函数会解析传入的请求数据，并向客户端发出响应。

在 Lambda 非代理集成中，您必须确保将 Lambda 函数的输入作为集成请求负载进行提供。您必须将客户端作为请求参数提供的任何输入数据映射到正确的集成请求正文。您可能还需要将客户端提供的请求正文转换为 Lambda 函数可识别的格式。

在 Lambda 代理或 Lambda 非代理集成中，您可以在一个账户中创建 API，而在另一个账户中使用 Lambda 函数。

主题

- 教程：使用 Lambda 代理集成构建 Hello World REST API
- 教程：使用 Lambda 非代理集成构建 API Gateway REST API
- 教程：使用跨账户 Lambda 代理集成构建 API Gateway REST API

教程：使用 Lambda 代理集成构建 Hello World REST API

Lambda 代理集成 是一种轻量型、灵活的 API Gateway API 集成类型，可让您能够使用 Lambda 函数集成 API 方法（或整个 API）。Lambda 函数可以用 Lambda 支持的任何语言编写。由于这是代理集成，因此您可以随时更改 Lambda 函数实现，而无需重新部署您的 API。

在本教程中，您将执行以下操作：

- 创建“Hello, World!” 要作为 API 的后端的 Lambda 函数。
- 创建并测试“Hello, World!” API 与 Lambda 代理集成。

主题

创建“Hello, World!” Lambda 函数

创建“Hello, World!” Lambda 控制台中的 Lambda 函数

1. 通过以下网址登录 Lambda 控制台：https://console.aws.amazon.com/lambda。
2. 在 AWS 导航栏上，选择 AWS 区域。

> **Note**
>
> 请注意您创建 Lambda 函数时所在的区域。在创建 API 时，会需要它。

3. 在导航窗格中，选择函数。
4. 选择创建函数。
5. 选择从头开始创作。
6. 在基本信息中，执行以下操作：
   a. 在函数名称中，输入 **`GetStartedLambdaProxyIntegration`**。
   b. 在运行时中，选择受支持的最新 Node.js 或 Python 运行时。
   c. 在权限下，展开更改默认执行角色。在执行角色下拉列表中，选择从 AWS 策略模板创建新角色。

   d. 在角色名称中，输入 **`GetStartedLambdaBasicExecutionRole`**。

   e. 将策略模板字段留空。

   f. 选择创建函数。

7. 在内联代码编辑器的函数代码下，复制/粘贴以下代码：

   Node.js

```
export const handler = function(event, context, callback) {
    console.log('Received event:', JSON.stringify(event, null, 2));
    var res ={
        "statusCode": 200,
        "headers": {
            "Content-Type": "*/*"
        }
    };
    var greeter = 'World';
    if (event.greeter && event.greeter!=="") {
        greeter =  event.greeter;
    } else if (event.body && event.body !== "") {
        var body = JSON.parse(event.body);
        if (body.greeter && body.greeter !== "") {
            greeter = body.greeter;
        }
    } else if (event.queryStringParameters && event.queryStringParameters.greeter && event.queryStringParameters.greeter !== "") {
        greeter = event.queryStringParameters.greeter;
    } else if (event.multiValueHeaders && event.multiValueHeaders.greeter && event.multiValueHeaders.greeter != "") {
        greeter = event.multiValueHeaders.greeter.join(" and ");
    } else if (event.headers && event.headers.greeter && event.headers.greeter != "") {
        greeter = event.headers.greeter;
    }

    res.body = "Hello, " + greeter + "!";
    callback(null, res);
};
```

Python

```
import json


def lambda_handler(event, context):
    print(event)

    greeter = 'World'

    try:
        if (event['queryStringParameters']) and (event['queryStringParameters']
['greeter']) and (
                event['queryStringParameters']['greeter'] is not None):
            greeter = event['queryStringParameters']['greeter']
    except KeyError:
        print('No greeter')

    try:
        if (event['multiValueHeaders']) and (event['multiValueHeaders']
['greeter']) and (
                event['multiValueHeaders']['greeter'] is not None):
            greeter = " and ".join(event['multiValueHeaders']['greeter'])
    except KeyError:
        print('No greeter')

    try:
        if (event['headers']) and (event['headers']['greeter']) and (
                event['headers']['greeter'] is not None):
            greeter = event['headers']['greeter']
    except KeyError:
        print('No greeter')

    if (event['body']) and (event['body'] is not None):
        body = json.loads(event['body'])
        try:
            if (body['greeter']) and (body['greeter'] is not None):
                greeter = body['greeter']
        except KeyError:
            print('No greeter')

    res = {
        "statusCode": 200,
```

```
        "headers": {
            "Content-Type": "*/*"
        },
        "body": "Hello, " + greeter + "!"
    }

    return res
```

8. 选择部署。

创建“Hello, World!” API

现在为您的“Hello, World!”创建 API。使用 API Gateway 控制台的 Lambda 函数。

创建“Hello, World!” API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 如果您是第一次使用 API Gateway，您会看到一个介绍服务特征的页面。在 REST API 下，选择生成。当创建示例 API 弹出框出现时，选择确定。

   如果这不是您首次使用 API Gateway，请选择创建 API。在 REST API 下，选择生成。
3. 对于 API 名称，请输入 **`LambdaProxyAPI`**。
4. （可选）对于描述，输入描述。
5. 将 API 端点类型设置保留为区域。
6. 选择创建 API。

创建 API 后，您将创建一个资源。通常情况下，根据应用程序逻辑将 API 资源组织成资源树形式。在本示例中，您将创建一个 /helloworld 资源。

创建资源

1. 选择 / 资源，然后选择创建资源。
2. 将代理资源保持为关闭状态。
3. 将资源路径保持为 `/`。
4. 对于资源名称，输入 **`helloworld`**。
5. 将 CORS（跨源资源共享）保持为关闭状态。

6. 选择创建资源。

在代理集成中，整个请求将通过表示任何 HTTP 方法的“捕获全部”ANY 方法按原样发送到后端 Lambda 函数。实际的 HTTP 方法由客户端在运行时指定。ANY 方法可让您对所有支持的 HTTP 方法使用单个 API 方法设置：DELETE、GET、HEAD、OPTIONS、PATCH、POST 和 PUT。

创建 **ANY** 方法

1. 选择 /helloworld 资源，然后选择创建方法。
2. 对于方法类型，选择 ANY。
3. 对于集成类型，选择 Lambda 函数。
4. 打开 Lambda 代理集成。
5. 对于 Lambda 函数，选择您创建 Lambda 函数的 AWS 区域，然后输入函数名称。
6. 要使用默认超时值 29 秒，请保持默认超时处于开启状态。要设置自定义超时，请选择默认超时，然后输入一个介于 50 到 29000 毫秒之间的超时值。
7. 选择创建方法。

部署并测试 API

部署 API

1. 选择部署 API。
2. 对于阶段，选择新建阶段。
3. 对于阶段名称，输入 **`test`**。
4. （可选）对于描述，输入描述。
5. 选择部署。
6. 在阶段详细信息下，选择复制图标以复制您 API 的调用 URL。

使用浏览器和 cURL 来通过 Lambda 代理集成测试 API

您可以使用浏览器或 cURL 来测试您的 API。

要仅使用查询字符串参数测试 GET 请求，您可以在浏览器地址栏中输入 API 的 `helloworld` 资源的 URL。

要创建 API 的 `helloworld` 资源的 URL，请将资源 `helloworld` 和查询字符串参数 `?greeter=John` 附加到您的调用 URL。URL 应类似以下内容。

```
https://r275xc9bmd.execute-api.us-east-1.amazonaws.com/test/helloworld?greeter=John
```

对于其他方法，您必须使用更高级的 REST API 测试实用程序，如 [POSTMAN](https://www.postman.com/) 或 [cURL](https://curl.se/)。本教程使用的是 cURL。以下 cURL 命令示例假定您的计算机上已安装 cURL。

使用 cURL 测试已部署的 API：

1. 打开终端窗口。
2. 复制以下 cURL 命令并粘贴到终端窗口中，将调用 URL 替换为在上一步中复制的调用 URL，并在该 URL 的末尾添加 **/helloworld**。

   > **Note**
   >
   > 如果您在 Windows 上运行命令，请改用以下语法：
   >
   > ```
   > curl -v -X POST "https://r275xc9bmd.execute-api.us-east-1.amazonaws.com/
   > test/helloworld" -H "content-type: application/json" -d "{ \"greeter\":
   >  \"John\" }"
   > ```

   a. 使用查询字符串参数 `?greeter=John` 调用 API：

   ```
   curl -X GET 'https://r275xc9bmd.execute-api.us-east-1.amazonaws.com/test/
   helloworld?greeter=John'
   ```

   b. 使用标头参数 `greeter:John` 调用 API：

   ```
   curl -X GET https://r275xc9bmd.execute-api.us-east-1.amazonaws.com/test/
   helloworld \
     -H 'content-type: application/json' \
     -H 'greeter: John'
   ```

   c. 使用正文 `{"greeter":"John"}` 调用 API：

   ```
   curl -X POST https://r275xc9bmd.execute-api.us-east-1.amazonaws.com/test/
   helloworld \
     -H 'content-type: application/json' \
   ```

```
  -d '{ "greeter": "John" }'
```

在所有情况下，输出为具有以下响应正文的 200 响应：

```
Hello, John!
```

教程：使用 Lambda 非代理集成构建 API Gateway REST API

在本演练中，我们使用 API Gateway 控制台构建一个 API，该 API 允许客户端通过 Lambda 非代理集成（又称为“自定义集成”）来调用 Lambda 函数。有关 AWS Lambda 和 Lambda 函数的更多信息，请参阅 AWS Lambda 开发人员指南。

为了便于学习，我们选择了一个所需 API 设置最少的简单 Lambda 函数，逐步指导您完成使用 Lambda 自定义集成构建 API Gateway API 的步骤。必要时，我们会介绍一些逻辑。有关 Lambda 自定义集成的更详细示例，请参阅 教程：通过两种 AWS 服务集成和一种 Lambda 非代理集成创建计算器 REST API。

在创建 API 之前，请通过在 AWS Lambda 中创建 Lambda 函数来设置 Lambda 后端，如下所述。

主题

为 Lambda 非代理集成创建 Lambda 函数

> **Note**
>
> 创建 Lambda 函数可能会导致您的AWS账户产生费用。

在此步骤中，您将为 Lambda 自定义集成创建一个如“Hello, World!”的 Lambda 函数。在本演练中，该函数称为 `GetStartedLambdaIntegration`。

该 `GetStartedLambdaIntegration` Lambda 函数的实现如下所示：

Node.js

```
'use strict';
var days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
 'Saturday'];
var times = ['morning', 'afternoon', 'evening', 'night', 'day'];

console.log('Loading function');

export const handler = function(event, context, callback) {
  // Parse the input for the name, city, time and day property values
  let name = event.name === undefined ? 'you' : event.name;
  let city = event.city === undefined ? 'World' : event.city;
  let time = times.indexOf(event.time)<0 ? 'day' : event.time;
  let day = days.indexOf(event.day)<0 ? null : event.day;

  // Generate a greeting
  let greeting = 'Good ' + time + ', ' + name + ' of ' + city + '. ';
  if (day) greeting += 'Happy ' + day + '!';

  // Log the greeting to CloudWatch
  console.log('Hello: ', greeting);

  // Return a greeting to the caller
  callback(null, {
      "greeting": greeting
  });
};
```

Python

```
import json

days = {
    'Sunday',
    'Monday',
    'Tuesday',
    'Wednesday',
    'Thursday',
    'Friday',
    'Saturday'}
```

```
times = {'morning', 'afternoon', 'evening', 'night', 'day'}


def lambda_handler(event, context):
    print(event)
    # parse the input for the name, city, time, and day property values
    try:
        if event['name']:
            name = event['name']
    except KeyError:
        name = 'you'
    try:
        if event['city']:
            city = event['city']
    except KeyError:
        city = 'World'
    try:
        if event['time'] in times:
            time = event['time']
        else:
            time = 'day'
    except KeyError:
        time = 'day'
    try:
        if event['day'] in days:
            day = event['day']
        else:
            day = ''
    except KeyError:
        day = ''
    # Generate a greeting
    greeting = 'Good ' + time + ', ' + name + ' of ' + \
        city + '.' + ['', ' Happy ' + day + '!'][day != '']
    # Log the greeting to CloudWatch
    print(greeting)

    # Return a greeting to the caller
    return {"greeting": greeting}
```

对于 Lambda 自定义集成，API Gateway 会将来自客户端的 Lambda 函数的输入作为集成请求正文来传递。Lambda 函数处理程序的 `event` 对象是输入。

我们的 Lambda 函数很简单。它会解析输入 `event` 对象的 `name`、`city`、`time` 和 `day` 属性。然后，它会以 `{"message":greeting}` 的 JSON 对象的形式向调用方返回问候语。该消息采用 `"Good [morning|afternoon|day], [name|you] in [city|World]. Happy day!"` 模式。假设 Lambda 函数的输入属于以下 JSON 对象：

```
{
  "city": "...",
  "time": "...",
  "day": "...",
  "name" : "..."
}
```

有关更多信息，请参阅 AWS Lambda 开发人员指南。

此外，该函数通过调用 `console.log(...)` 将其执行记录到 Amazon CloudWatch。这有助于在调试函数时跟踪调用。要允许 `GetStartedLambdaIntegration` 函数记录调用，请用用于 Lambda 函数的适当策略来设置 IAM 角色，以创建 CloudWatch 流并向流中添加日志条目。Lambda 控制台将指导您创建所需的 IAM 角色和策略。

如果您设置 API 时不使用 API Gateway 控制台（例如，当从 OpenAPI 导入 API 时），您必须显式创建（如有必要）并设置调用角色和策略以便 API Gateway 调用 Lambda 函数。有关如何为 API Gateway API 设置 Lambda 调用和执行角色的更多信息，请参阅使用 IAM 权限控制对 API 的访问。

与用于 Lambda 代理集成的 Lambda 函数 `GetStartedLambdaProxyIntegration` 相比，用于 Lambda 自定义集成的 `GetStartedLambdaIntegration` Lambda 函数仅从 API Gateway API 集成请求正文中获取输入。该函数可以返回任何 JSON 对象、字符串、数字、布尔值甚至是二进制 blob 形式的输出。相比而言，用于 Lambda 代理集成的 Lambda 函数可从任何请求数据获取输入，但必须返回特定 JSON 对象形式的输出。用于 Lambda 自定义集成的 `GetStartedLambdaIntegration` 函数可以使用 API 请求参数作为输入，前提是 API Gateway 在将客户端请求转发至后端之前，将所需的 API 请求参数映射至集成请求正文。要实现这一点，API 开发人员必须在创建 API 时创建一个映射模板并在 API 方法中对其进行配置。

现在，创建 `GetStartedLambdaIntegration` Lambda 函数。

为 Lambda 自定义集成创建 **`GetStartedLambdaIntegration`** Lambda 函数

1. 打开 AWS Lambda 控制台，地址：https://console.aws.amazon.com/lambda/。
2. 请执行下列操作之一：
    - 如果显示欢迎页面，请选择立即开始使用，然后选择创建函数。

- 如果显示 Lambda > 函数列表页面，请选择创建函数。

3. 选择从头开始创作。
4. 在从头开始创作窗格中，执行以下操作：

   a. 在名称中，输入 **GetStartedLambdaIntegration** 作为 Lambda 函数名称。

   b. 在运行时中，选择受支持的最新 Node.js 或 Python 运行时。

   c. 在权限下，展开更改默认执行角色。在执行角色下拉列表中，选择从 AWS 策略模板创建新角色。

   d. 对于角色名称，输入您角色的名称（例如 **GetStartedLambdaIntegrationRole**）。

   e. 对于策略模板，选择简单微服务权限。

   f. 选择创建函数。
5. 在配置函数窗格中的函数代码下，执行以下操作：

   a. 复制本节开头部分列出的 Lambda 函数代码并将其粘贴到内联代码编辑器中。

   b. 对本部分中的所有其他字段保留默认选择。

   c. 选择部署。
6. 要测试新创建的函数，请选择测试选项卡。

   a. 对于事件名称，输入 **HelloWorldTest**。

   b. 对于事件 JSON，请将默认代码替换为以下代码。

   ```
   {
     "name": "Jonny",
     "city": "Seattle",
     "time": "morning",
     "day": "Wednesday"
   }
   ```

   c. 选择测试以调用该函数。将显示执行结果: 成功部分。展开详细信息，您会看到以下输出。

   ```
   {
       "greeting": "Good morning, Jonny of Seattle. Happy Wednesday!"
   }
   ```

   还会将输出写入 CloudWatch Logs 中。

作为同步练习，您可以使用 IAM 控制台查看 IAM 角色 (`GetStartedLambdaIntegrationRole`)，此角色是在创建 Lambda 函数的过程中创建的。此 IAM 角色附加了两个内联策略。一个策略规定 Lambda 执行的最基本权限。它允许在创建 Lambda 函数的区域中为您账户的任何 CloudWatch 资源调用 CloudWatch `CreateLogGroup`。此策略还允许创建 CloudWatch 流和为 `GetStartedLambdaIntegration` Lambda 函数记录事件。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "logs:CreateLogGroup",
            "Resource": "arn:aws:logs:region:account-id:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": [
                "arn:aws:logs:region:account-id:log-group:/aws/lambda/
GetStartedLambdaIntegration:*"
            ]
        }
    ]
}
```

另一个策略文档适用于调用此示例中未使用的其他 AWS 服务。您可以暂时跳过此策略。

与 IAM 角色关联的是可信实体 `lambda.amazonaws.com`。下面是信任关系：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
```

```
  ]
}
```

此信任关系与内联策略的组合，使得 Lambda 函数能够调用 `console.log()` 函数来将事件记录到 CloudWatch Logs。

使用 Lambda 非代理集成创建 API

创建并测试 Lambda 函数 (`GetStartedLambdaIntegration`) 后，您便已准备就绪，可以通过 API Gateway API 来公开该函数。为了便于说明，我们用通用 HTTP 方法公开 Lambda 函数。我们使用请求正文、URL 路径变量、查询字符串和标头来接收来自客户端的所需输入数据。我们为 API 启用 API Gateway 请求验证程序，以确保正确定义并指定所有必需的数据。我们为 API Gateway 配置映射模板，以根据后端 Lambda 函数的要求将客户端提供的请求数据转换为有效格式。

使用 Lambda 非代理集成创建 API

1. 通过以下网址登录到 API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 如果您是第一次使用 API Gateway，您会看到一个介绍服务特征的页面。在 REST API 下，选择生成。当创建示例 API 弹出框出现时，选择确定。

   如果这不是您首次使用 API Gateway，请选择创建 API。在 REST API 下，选择生成。
3. 对于 API 名称，请输入 **`LambdaNonProxyAPI`**。
4. （可选）对于描述，输入描述。
5. 将 API 端点类型设置保留为区域。
6. 选择创建 API。

创建 API 后，您将创建一个 /{city} 资源。这是带有从客户端获取输入的路径变量的资源示例。稍后，您将使用映射模板将此路径变量映射到 Lambda 函数输入。

创建资源

1. 选择创建资源。
2. 将代理资源保持为关闭状态。
3. 将资源路径保持为 `/`。
4. 对于资源名称，输入 **`{city}`**。
5. 将 CORS（跨源资源共享）保持为关闭状态。
6. 选择创建资源。

创建 /{city} 资源后，您将创建一个 ANY 方法。ANY HTTP 动词是客户端在运行时提交的有效 HTTP 方法的占位符。此示例显示，ANY 方法可用于 Lambda 自定义集成和 Lambda 代理集成。

创建 **`ANY`** 方法

1. 选择 /{city} 资源，然后选择创建方法。
2. 对于方法类型，选择 ANY。
3. 对于集成类型，选择 Lambda 函数。
4. 保持 Lambda 代理集成处于关闭状态。
5. 对于 Lambda 函数，选择您创建 Lambda 函数的 AWS 区域，然后输入函数名称。
6. 选择方法请求设置。

   现在，您可以为 URL 路径变量、查询字符串参数和标头开启请求验证程序，来确保定义了所有必需的数据。在本示例中，您将创建一个 `time` 查询字符串参数和一个 `day` 标头。
7. 对于请求验证程序，选择验证查询字符串参数和标头。
8. 选择 URL 查询字符串参数并执行以下操作：
   a. 选择添加查询字符串。
   b. 在名称中，输入 **`time`**。
   c. 打开必需。
   d. 将缓存保持为关闭状态。
9. 选择 HTTP 请求标头并执行以下操作：
   a. 选择添加标头。
   b. 在名称中，输入 **`day`**。
   c. 打开必需。
   d. 将缓存保持为关闭状态。
10. 选择创建方法。

打开请求验证程序后，您可以根据后端 Lambda 函数的要求，通过添加正文映射模板来配置 ANY 方法的集成请求，以将传入的请求转换为 JSON 负载。

配置集成请求

1. 在集成请求选项卡的集成请求设置下，选择编辑。

2. 对于请求正文传递，选择当未定义模板时（推荐）。
3. 选择映射模板。
4. 选择添加映射模板。
5. 对于内容类型，输入 **`application/json`**。
6. 对于模板正文，输入以下代码：

```
#set($inputRoot = $input.path('$'))
{
  "city": "$input.params('city')",
  "time": "$input.params('time')",
  "day":  "$input.params('day')",
  "name": "$inputRoot.callerName"
}
```

7. 选择保存。

测试 API 方法的调用

API Gateway 控制台提供了测试工具，以供您在部署 API 之前测试 API 的调用。您使用控制台的测试特征通过提交以下请求来测试 API：

```
POST /Seattle?time=morning
day:Wednesday

{
    "callerName": "John"
}
```

在此测试请求中，您可以将 ANY 设置为 `POST`、将 `{city}` 设置为 `Seattle`、将 `Wednesday` 分配为 `day` 标头值，将 `"John"` 分配为 `callerName` 值。

测试 **ANY** 方法

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
2. 对于方法类型，选择 `POST`。
3. 对于路径，在城市下输入 **`Seattle`**。
4. 对于查询字符串，输入 **`time=morning`**。
5. 对于标头，输入 **`day:Wednesday`**。

6. 对于请求正文，输入 **`{ "callerName": "John" }`**。
7. 选择测试。

验证返回的响应负载是否如下所示：

```
{
  "greeting": "Good morning, John of Seattle. Happy Wednesday!"
}
```

您还可以查看日志，以确定 API Gateway 如何处理请求和响应。

```
Execution log for request test-request
Thu Aug 31 01:07:25 UTC 2017 : Starting execution for request: test-invoke-request
Thu Aug 31 01:07:25 UTC 2017 : HTTP Method: POST, Resource Path: /Seattle
Thu Aug 31 01:07:25 UTC 2017 : Method request path: {city=Seattle}
Thu Aug 31 01:07:25 UTC 2017 : Method request query string: {time=morning}
Thu Aug 31 01:07:25 UTC 2017 : Method request headers: {day=Wednesday}
Thu Aug 31 01:07:25 UTC 2017 : Method request body before transformations:
 { "callerName": "John" }
Thu Aug 31 01:07:25 UTC 2017 : Request validation succeeded for content type
 application/json
Thu Aug 31 01:07:25 UTC 2017 : Endpoint request URI: https://
lambda.us-west-2.amazonaws.com/2015-03-31/functions/arn:aws:lambda:us-
west-2:123456789012:function:GetStartedLambdaIntegration/invocations
Thu Aug 31 01:07:25 UTC 2017 : Endpoint request headers: {x-amzn-lambda-integration-
tag=test-request,
 Authorization=********************************************************************************
 X-Amz-Date=20170831T010725Z, x-amzn-apigateway-api-id=beags1mnid, X-Amz-
Source-Arn=arn:aws:execute-api:us-west-2:123456789012:beags1mnid/null/POST/
{city}, Accept=application/json, User-Agent=AmazonAPIGateway_beags1mnid,
 X-Amz-Security-Token=FQoDYXdzELL//////////wEaDMHGzEdEOT/VvGhabiK3AzgKrJw
+3zLqJZG4PhOq12K6W21+QotY2rrZyOzqhLoiuRg3CAYNQ2eqgL5D54+63ey9bIdtwHGoyBdq8ecWxJK/
YUnT2Rau0L9HCG5p7FC05h3IvwlFfvcidQNXeYvsKJTLXI05/
yEnY3ttIAnpNYLOezD9Es8rBfyruHfJfOqextKlsC8DymCcqlGkig8qLKcZ0hWJWVwiPJiFgL7laabXs+
+ZhCa4hdZo4iqlG729DE4gaV1mJVdoAagIUwLMo+y4NxFDu0r7I0/
EO5nYcCrppGVVBYiGk7H4T6sXuhTkbNNqVmXtV3ch5bOlh7 [TRUNCATED]
Thu Aug 31 01:07:25 UTC 2017 : Endpoint request body after transformations: {
  "city": "Seattle",
  "time": "morning",
  "day": "Wednesday",
  "name" : "John"
}
```

```
Thu Aug 31 01:07:25 UTC 2017 : Sending request to https://lambda.us-
west-2.amazonaws.com/2015-03-31/functions/arn:aws:lambda:us-
west-2:123456789012:function:GetStartedLambdaIntegration/invocations
Thu Aug 31 01:07:25 UTC 2017 : Received response. Integration latency: 328 ms
Thu Aug 31 01:07:25 UTC 2017 : Endpoint response body before transformations:
 {"greeting":"Good morning, John of Seattle. Happy Wednesday!"}
Thu Aug 31 01:07:25 UTC 2017 : Endpoint response headers: {x-amzn-Remapped-Content-
Length=0, x-amzn-RequestId=c0475a28-8de8-11e7-8d3f-4183da788f0f, Connection=keep-
alive, Content-Length=62, Date=Thu, 31 Aug 2017 01:07:25 GMT, X-Amzn-Trace-
Id=root=1-59a7614d-373151b01b0713127e646635;sampled=0, Content-Type=application/json}
Thu Aug 31 01:07:25 UTC 2017 : Method response body after transformations:
 {"greeting":"Good morning, John of Seattle. Happy Wednesday!"}
Thu Aug 31 01:07:25 UTC 2017 : Method response headers: {X-Amzn-Trace-
Id=sampled=0;root=1-59a7614d-373151b01b0713127e646635, Content-Type=application/json}
Thu Aug 31 01:07:25 UTC 2017 : Successfully completed execution
Thu Aug 31 01:07:25 UTC 2017 : Method completed with status: 200
```

日志在映射之前显示传入请求并在映射之后显示集成请求。当测试失败时，日志对于评估原始输入是否正确或映射模板工作是否正常很有用。

部署 API

测试调用是一种模拟，会受到一些限制。例如，它会绕过 API 中应用的任何授权机制。要实时测试 API 执行，您必须先部署 API。要部署 API，您需创建一个阶段，以创建当时的 API 快照。阶段名称还定义在 API 的默认主机名后面的基本路径。API 的根资源附加在阶段名称之后。当您修改 API 时，必须将其重新部署到新阶段或现有阶段，然后更改才会生效。

将 API 部署到某个阶段

1. 选择部署 API。
2. 对于阶段，选择新建阶段。
3. 对于阶段名称，输入 **test**。

> **Note**
>
> 输入必须是 UTF-8 编码（即未本地化）的文本。

4. （可选）对于描述，输入描述。
5. 选择部署。

在阶段详细信息下，选择复制图标以复制您 API 的调用 URL。API 的基本 URL 的一般模式是 `https://`*`api-id`*`.`*`region`*`.amazonaws.com/`*`stageName`*。例如，在 `beags1mnid` 区域中创建并部署到 `us-west-2` 阶段的 API (`test`) 的基本 URL 是 `https://beags1mnid.execute-api.us-west-2.amazonaws.com/test`。

在部署阶段测试 API

可通过若干种方法来测试已部署的 API。对于仅使用 URL 路径变量或查询字符串参数的 GET 请求，您可以在浏览器中输入 API 资源 URL。对于其他方法，您必须使用更高级的 REST API 测试实用程序，如 POSTMAN 或 cURL。

使用 cURL 测试 API

1. 在连接到 Internet 的本地计算机上打开终端窗口。
2. 测试 `POST /Seattle?time=evening`：

   复制以下 cURL 命令并将其粘贴到终端窗口中。

   ```
   curl -v -X POST \
     'https://beags1mnid.execute-api.us-west-2.amazonaws.com/test/Seattle?
   time=evening' \
     -H 'content-type: application/json' \
     -H 'day: Thursday' \
     -H 'x-amz-docs-region: us-west-2' \
     -d '{
    "callerName": "John"
   }'
   ```

   您应获得一个包含以下负载的成功响应：

   ```
   {"greeting":"Good evening, John of Seattle. Happy Thursday!"}
   ```

   如果您在此方法请求中将 `POST` 更改为 `PUT`，则会获得相同的响应。

清除

如果您不再需要您为本演练创建的 Lambda 函数，现在可以将其删除。您也可以删除附带的 IAM 资源。

⚠ Warning

> 如果您计划完成本系列中的其他演练，请不要删除 Lambda 执行角色或 Lambda 调用角色。如果您删除您的 API 所依赖的某个 Lambda 函数，这些 API 将不再有效。Lambda 函数删除操作无法撤消。如果您要再次使用 Lambda 函数，必须重新创建该函数。
>
> 如果您删除 Lambda 函数所依赖的 IAM 资源，Lambda 函数将不再有效，依赖于此函数的 API 也不再有效。IAM 资源删除操作无法撤消。如果您要再次使用 IAM 资源，必须重新创建该资源。

删除 Lambda 函数

1. 登录到 AWS Management Console，然后通过以下网址打开 AWS Lambda 控制台：https://console.aws.amazon.com/lambda/。
2. 从函数列表中选择 GetStartedLambdaIntegration，再选择操作，然后选择删除函数。当系统提示时，再次选择删除。

删除相关联的 IAM 资源

1. 通过以下网址打开 IAM 控制台：https://console.aws.amazon.com/iam/。
2. 从详细信息中，选择角色。
3. 从角色列表中选择 GetStartedLambdaIntegrationRole，再选择角色操作，然后选择删除角色。按照控制台中的步骤删除该角色。

教程：使用跨账户 Lambda 代理集成构建 API Gateway REST API

您现在可以使用其他 AWS Lambda 账户中的 AWS 函数作为 API 集成后端。每个账户都可以位于 Amazon API Gateway 可用的任何区域中。这样便可轻松地跨多个 API 集中管理和共享 Lambda 后端函数。

在本节中，我们将介绍如何使用 Amazon API Gateway 控制台配置跨账户 Lambda 代理集成。

为 API Gateway 跨账户 Lambda 集成创建 API

创建 API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。

2. 如果您是第一次使用 API Gateway，您会看到一个介绍服务特征的页面。在 REST API 下，选择生成。当创建示例 API 弹出框出现时，选择确定。

   如果这不是您首次使用 API Gateway，请选择创建 API。在 REST API 下，选择生成。
3. 对于 API 名称，请输入 **`CrossAccountLambdaAPI`**。
4. （可选）对于描述，输入描述。
5. 将 API 端点类型设置保留为区域。
6. 选择创建 API。

在另一个账户中创建 Lambda 集成函数

现在，您将在与创建示例 API 不同的账户中创建 Lambda 函数。

在另一个账户中创建 Lambda 函数

1. 从与创建 API Gateway API 不同的账户登录 Lambda 控制台。
2. 选择创建函数。
3. 选择从头开始创作。
4. 在从头开始创作下，执行以下操作：
   a. 对于函数名称，输入一个名称。
   b. 从运行时下拉列表中，选择受支持的 Node.js 运行时。
   c. 在权限下，展开选择或创建执行角色。您可以创建角色或选择现有角色。
   d. 选择创建函数以继续。
5. 向下滚动到函数代码窗格。
6. 输入来自 the section called “教程：使用 Lambda 代理集成的 Hello World API” 的 Node.js 函数实现。
7. 选择 Deploy (部署)。
8. 记下函数的完整 ARN（位于 Lambda 函数窗格的右上角）。当您创建跨账户 Lambda 集成时，将需要此信息。

配置跨账户 Lambda 集成

一旦您在其他账户中拥有 Lambda 集成函数，就可以使用 API Gateway 控制台将其添加到第一个账户中的 API。

> **Note**
>
> 如果要配置跨区域、跨账户授权方，则添加到目标函数的 `sourceArn` 应使用此函数的区域，而不是 API 的区域。

创建 API 后，您将创建一个资源。通常情况下，根据应用程序逻辑将 API 资源组织成资源树形式。在本示例中，您将创建一个 /helloworld 资源。

创建资源

1. 选择 / 资源，然后选择创建资源。
2. 将代理资源保持为关闭状态。
3. 将资源路径保持为 `/`。
4. 对于资源名称，输入 **`helloworld`**。
5. 将 CORS（跨源资源共享）保持为关闭状态。
6. 选择创建资源。

创建资源后，您将创建一个 `GET` 方法。您将此 `GET` 方法与另一个账户中的 Lambda 函数集成。

创建 **`GET`** 方法

1. 选择 /helloworld 资源，然后选择创建方法。
2. 对于方法类型，选择 GET。
3. 对于集成类型，选择 Lambda 函数。
4. 打开 Lambda 代理集成。
5. 对于 Lambda 函数，输入步骤 1 中您的 Lambda 函数的完整 ARN。

   在 Lambda 控制台中，您可以在控制台窗口的右上角找到函数的 ARN。
6. 当您输入 ARN 时，将出现一个 `aws lambda add-permission` 命令字符串。此策略将向您的第一个账户授予对第二个账户的 Lambda 函数的访问权限。将 `aws lambda add-permission` 命令字符串复制粘贴到为您的第二个账户配置的 AWS CLI 窗口中。
7. 选择创建方法。

您可以在 Lambda 控制台中查看您的函数更新后的策略。

（可选）查看更新后的策略

1. 登录到 AWS Management Console，然后通过以下网址打开 AWS Lambda 控制台：https://console.aws.amazon.com/lambda/。
2. 选择您的 Lambda 函数。
3. 选择权限。

   您应该看到具有 `Allow` 子句的 `Condition` 策略，其中 `AWS:SourceArn` 是您 API 的 `GET` 方法的 ARN。

教程：通过导入示例创建 REST API

您可以使用 Amazon API Gateway 控制台，借助 PetStore 网站的 HTTP 集成来创建并测试简单的 REST API。API 定义预配置为 OpenAPI 2.0 文件。在将 API 定义加载到 API Gateway 中后，您可以使用 API Gateway 控制台来检查 API 的基本结构或直接部署并测试 API。

PetStore 示例 API 支持客户端使用以下方法来访问 HTTP 后端网站 `http://petstore-demo-endpoint.execute-api.com/petstore/pets`。

> Note
>
> 本教程以 HTTP 端点为例。在创建自己的 API 时，建议您使用 HTTPS 端点进行 HTTP 集成。

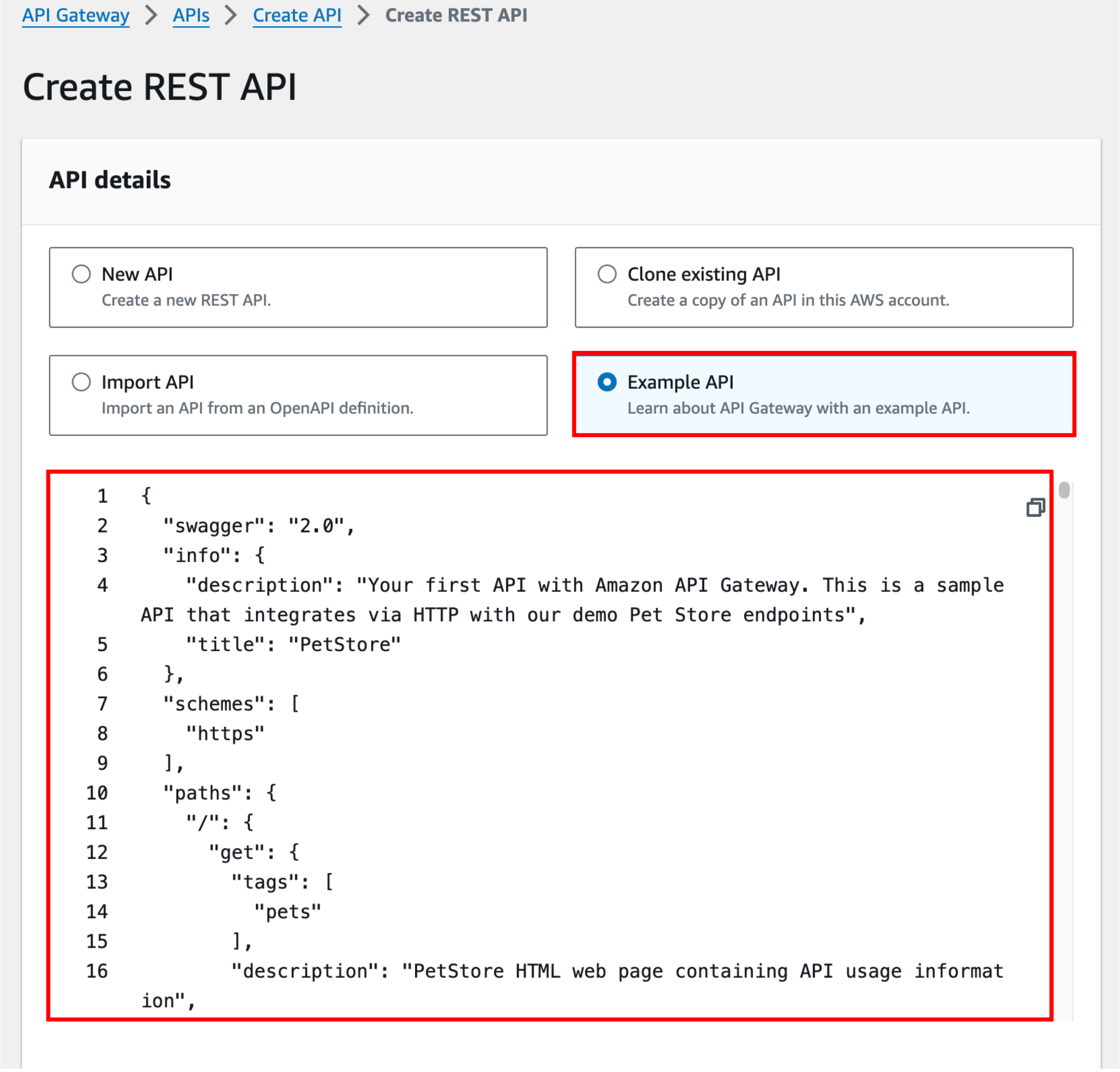
- `GET /`：用于对未与任何后端终端节点集成的 API 根资源进行读取访问。API Gateway 会使用 PetStore 网站的概述进行响应。这是 `MOCK` 集成类型的示例。
- `GET /pets`：用于对已与名称相同的后端 `/pets` 资源集成的 API `/pets` 资源进行读取访问。后端会返回 PetStore 中的可用宠物的页面。这是 `HTTP` 集成类型的示例。集成终端节点的 URL 为 `http://petstore-demo-endpoint.execute-api.com/petstore/pets`。
- `POST /pets`：用于对已与后端 `/pets` 资源集成的 API `/petstore/pets` 资源进行写入访问。收到正确请求后，后端会将指定的宠物添加到 PetStore 中并将结果返回给调用方。该集成也是 `HTTP` 集成。
- `GET /pets/{petId}`：用于对指定为传入请求 URL 的路径变量的 `petId` 值标识的宠物进行读取访问。此方法也具有 `HTTP` 集成类型。后端会返回在 PetStore 中找到的指定宠物。后端 HTTP 终端节点的 URL 是 `http://petstore-demo-endpoint.execute-api.com/petstore/pets/n`，其中 n 是一个用作所查询宠物的标识符的整数。

API 支持通过 `OPTIONS` 集成类型的 `MOCK` 方法进行 CORS 访问。API Gateway 会返回支持 CORS 访问所需的标头。

以下过程将指导您完成在 API Gateway 控制台中根据示例创建一个 API 并进行测试的步骤。

导入、构建并测试示例 API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 请执行以下操作之一：
    - 要创建第一个 API，对于 REST API，请选择构建。
    - 如果您之前已经创建了 API，请选择创建 API，然后为 REST API 选择构建。
3. 在创建 REST API 下，选择示例 API，然后选择创建 API 以创建示例 API。



API Gateway > APIs > Create API > Create REST API

Create REST API

API details

○ New API
Create a new REST API.

○ Clone existing API
Create a copy of an API in this AWS account.

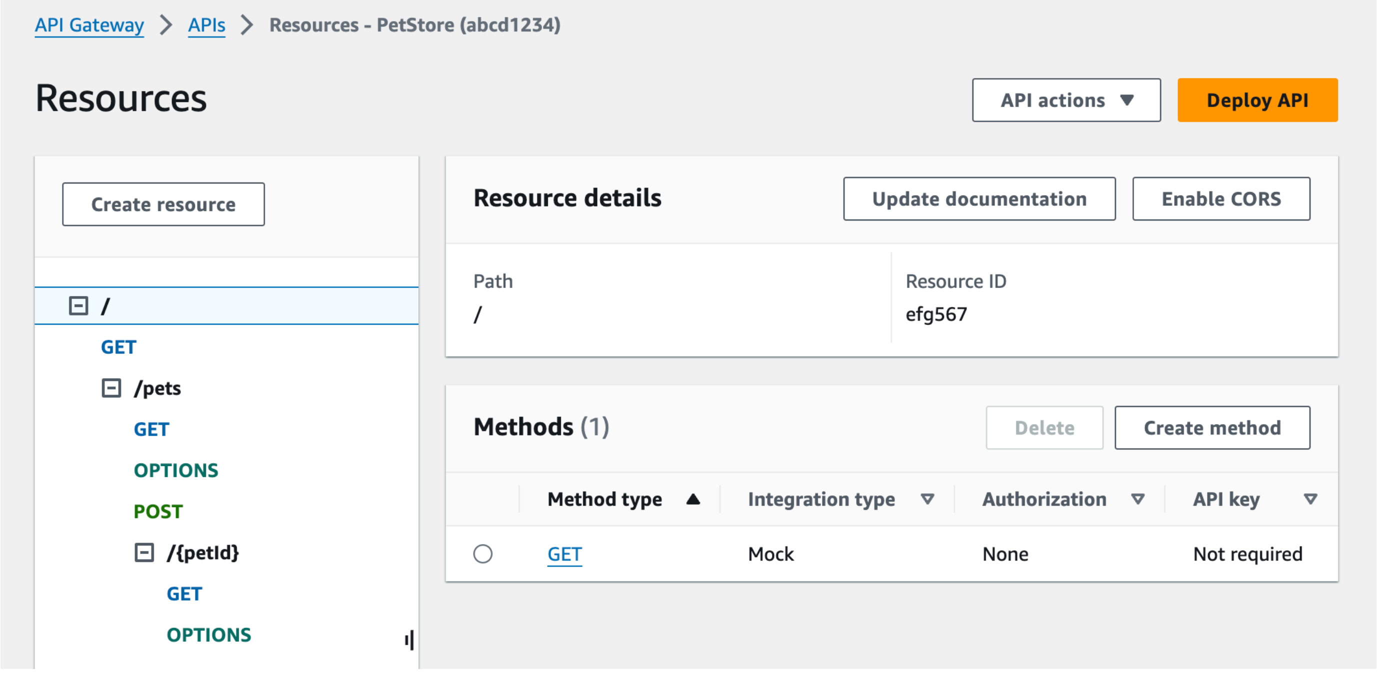
○ Import API
Import an API from an OpenAPI definition.

● Example API
Learn about API Gateway with an example API.

```
{
  "swagger": "2.0",
  "info": {
    "description": "Your first API with Amazon API Gateway. This is a sample API that integrates via HTTP with our demo Pet Store endpoints",
    "title": "PetStore"
  },
  "schemes": [
    "https"
  ],
  "paths": {
    "/": {
      "get": {
        "tags": [
          "pets"
        ],
        "description": "PetStore HTML web page containing API usage information",
```

在选择创建 API 之前，您可以向下滚动 OpenAPI 定义以了解此示例 API 的详细信息。

4. 在主导航窗格中，选择资源。新创建的 API 如下所示：



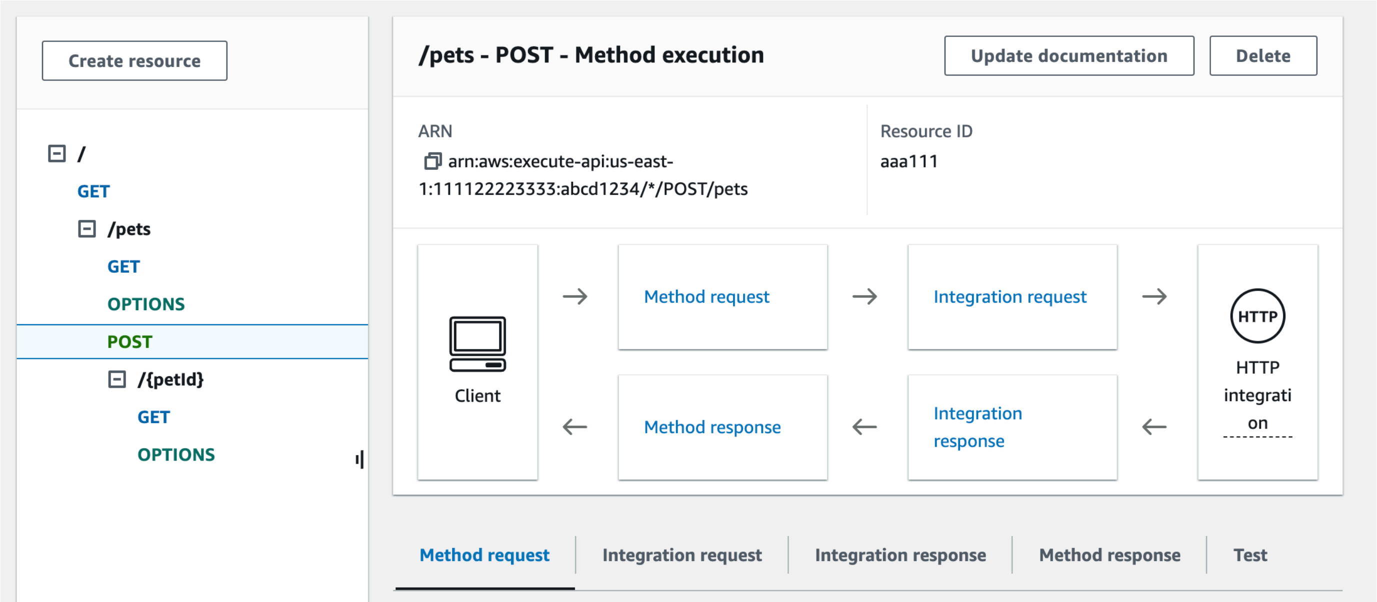
Path
/

Resource ID
efg567

	Method type ▲	Integration type ▽	Authorization ▽	API key ▽
○	GET	Mock	None	Not required

Resources (资源) 窗格将所创建 API 的结构显示为节点树。在每个资源上定义的 API 方法均位于节点树的边缘。当选中某个资源时，其所有方法都会在右侧的方法表中列出。与每种方法一起显示的是方法类型、集成类型、授权类型和 API 密钥要求。

5. 要查看某一方法的详细信息以修改其设置或测试方法调用，请从方法列表或资源树中选择相应的方法名称。在这里，我们选择 `POST /pets` 方法作为说明示例：



ARN
arn:aws:execute-api:us-east-1:111122223333:abcd1234/*/POST/pets

Resource ID
aaa111

生成的方法执行窗格显示所选（`POST /pets`）方法的结构和行为的逻辑视图。

方法请求和方法响应表示 API 与前端的接口，集成请求和集成响应表示 API 与后端的接口。

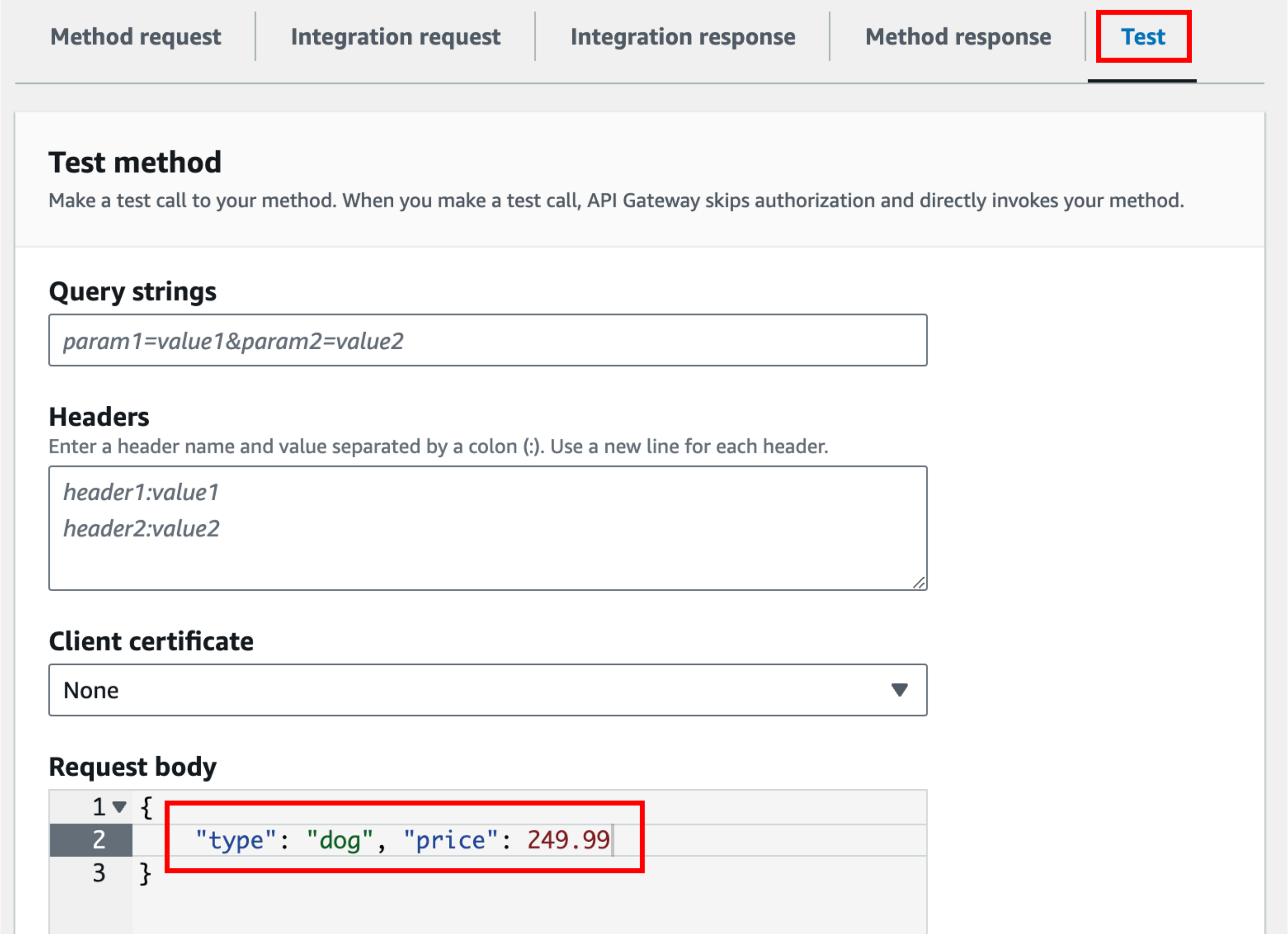
客户端可以使用 API 通过方法请求访问后端特征。如有必要，API Gateway 会先将客户端请求转换为集成请求中后端可接受的形式，然后再将该传入请求转发至后端。转换后的请求被称为集成请求。同样，后端在集成响应中向 API Gateway 返回响应。API Gateway 随后将其路由至方法响应，然后再将其发送到客户端。API Gateway 也会在必要时将后端响应数据映射为客户端所需的形式。

对于 API 资源上的 `POST` 方法，如果该方法请求负载的格式与集成请求负载的格式相同，则无需修改该方法请求负载即可将其传递到集成请求。

`GET /` 方法请求使用 `MOCK` 集成类型，且不与任何真实后端终端节点相关联。相应的集成响应设置为返回静态 HTML 页面。调用该方法时，API Gateway 只需接受该请求，并立即通过方法响应将配置的集成响应返回给客户端。您可以使用模拟集成来测试 API，无需后端终端节点。您也可以将其用于处理本地响应，即从正文映射模板生成的响应。

作为 API 开发人员，您可以通过配置方法请求和方法响应来控制 API 的前端交互行为，通过设置集成请求和集成响应来控制 API 的后端交互行为。这些涉及到方法和它的相应集成之间的数据映射。目前，我们的侧重点是测试 API 以提供端到端的用户体验。

6. 选择测试选项卡。您可能需要选择右箭头按钮以显示该选项卡。
7. 例如，要测试 `POST /pets` 方法，请在请求正文中输入以下 **`{"type": "dog","price": 249.99}`** 负载，然后选择测试。



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Query strings

param1=value1&param2=value2

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1
header2:value2

Client certificate

None

Request body

```
{
    "type": "dog", "price": 249.99
}
```

该输入可指定我们要添加到 PetStore 网站上的宠物列表中的宠物的属性。

8. 结果显示如下：

/pets - POST method test results

Request

/pets

Latency

9

Status

200

Response body

```
{
  "pet": {
    "type": "dog",
    "price": 249.99
  },
  "message": "success"
}
```

Response headers

```
{
  "Access-Control-Allow-Origin": "*",
  "Content-Type": "application/json",
  "X-Amzn-Trace-Id": "Root=1-65df8d2b-782cd3c572391cf4a85295f5"
}
```

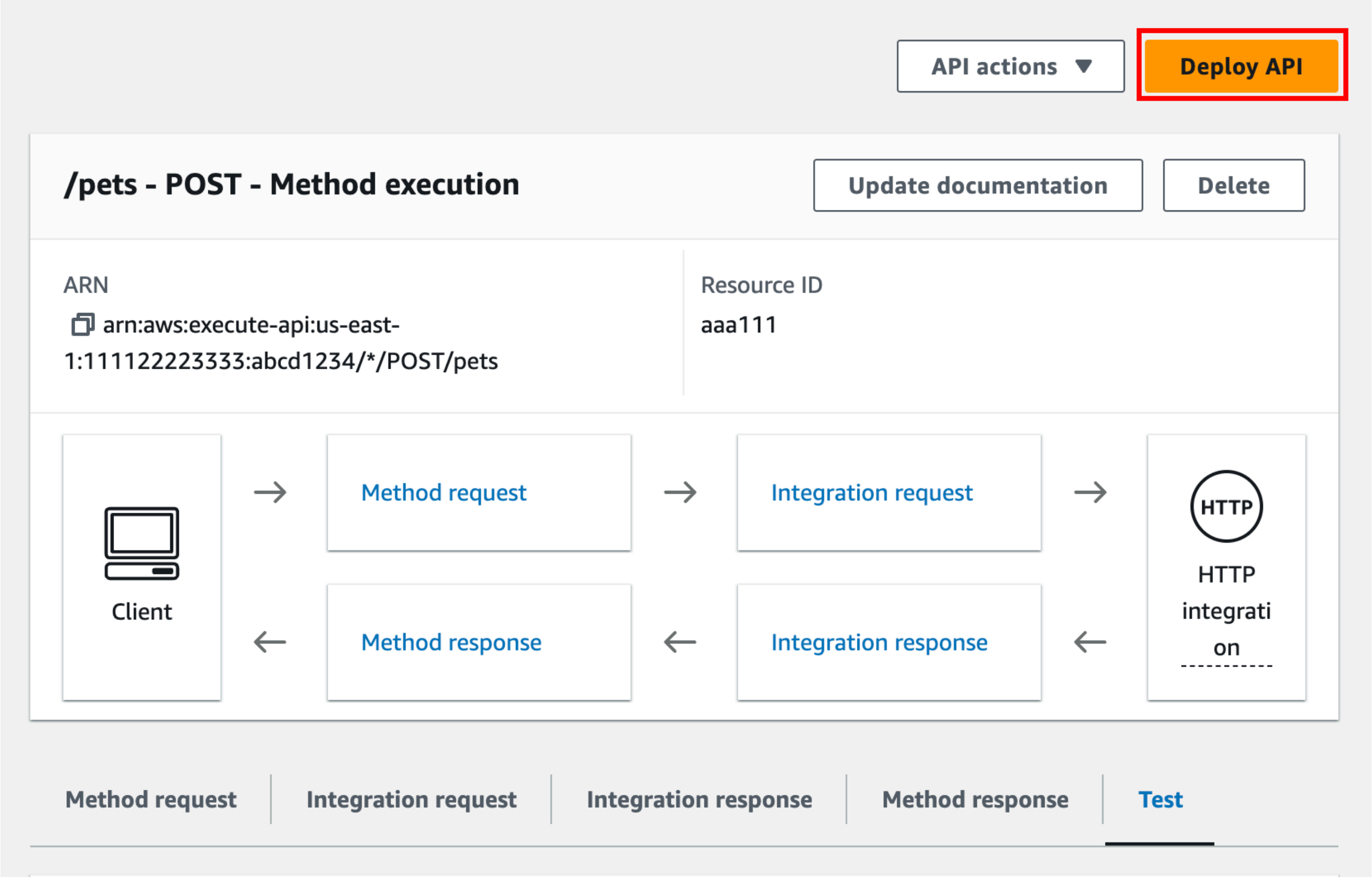
Log

```
Execution log for request 30f01060-307f-4447-803c-61679ea4c5d6
Wed Feb 28 19:44:43 UTC 2024 : Starting execution for request: 30f01060-
307f-4447-803c-61679ea4c5d6
```

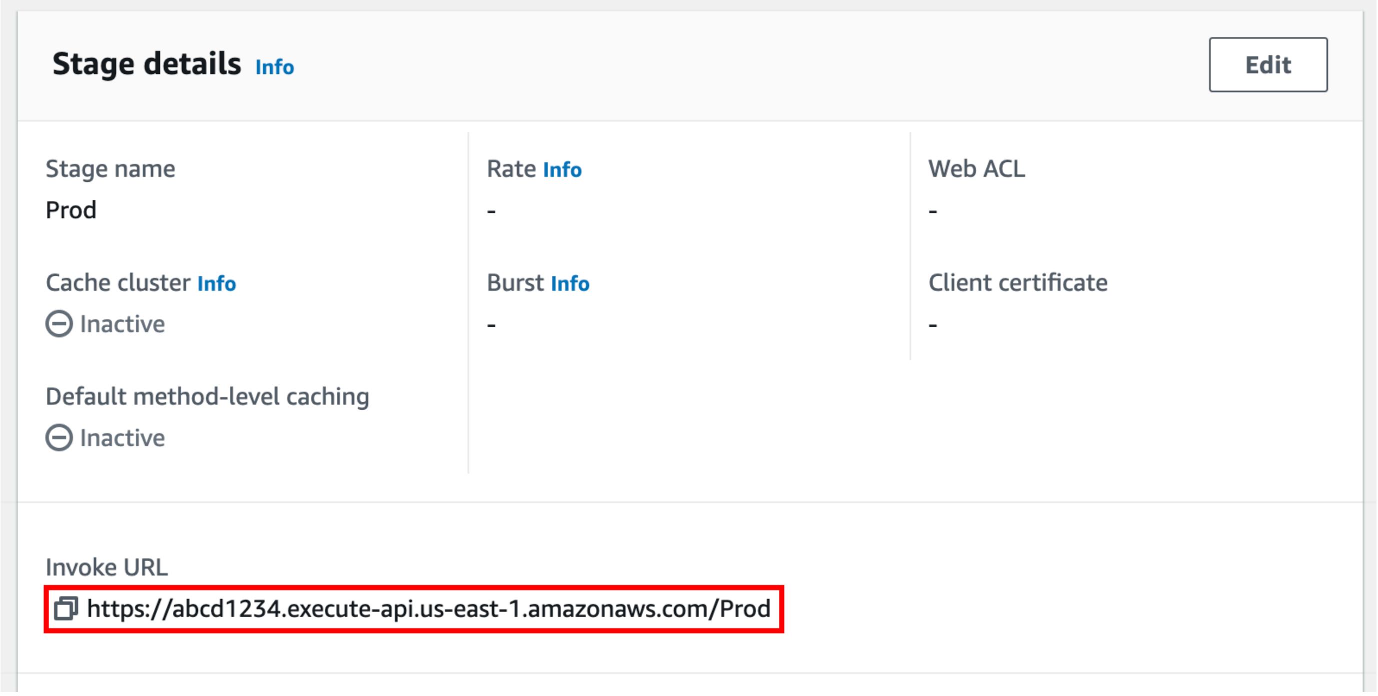
该输出的日志条目显示了从方法请求到集成请求以及从集成响应到方法响应的状态更改。这可用于对导致请求失败的任何映射错误进行故障排除。此示例中没有应用任何映射：方法请求负载通过集成请求传递到后端；同样地，后端响应通过集成响应传递到方法响应。

要使用客户端而非 API Gateway 的 test-invoke-request 特征来测试 API，您必须先将 API 部署至一个阶段。

9. 要部署示例 API，请选择部署 API。



10. 对于阶段，选择新建阶段，然后输入 **`test`**。
11. （可选）对于描述，输入描述。
12. 选择部署。
13. 在生成的阶段窗格中的阶段详细信息下，调用 URL 将显示用于调用 API 的 GET / 方法请求的 URL。



14. 选择复制图标以复制 API 的调用 URL，然后在 Web 浏览器中输入您 API 的调用 URL。成功的响应会返回由集成响应中的映射模板生成的结果。
15. 在 Stages（阶段）导航窗格中，展开 test（测试）阶段，选择 `/pets/{petId}` 上的 GET，然后复制 `https://`*`api-id`*`.execute-api.`*`region`*`.amazonaws.com/test/pets/{petId}` 的 Invoke URL（调用 URL）值。`{petId}` 代表路径变量。

    将 Invoke URL (调用 URL) 值（在上一步中获取）粘贴到浏览器的地址栏中，将 `{petId}` 替换为 1（举例来说），然后按 Enter 键提交请求。系统应返回一个包含以下 JSON 负载的 200 OK 响应：

    ```
    {
      "id": 1,
      "type": "dog",
      "price": 249.99
    }
    ```

    按上图所示来调用 API 方法是可行的，因为其 Authorization (授权) 类型设置为 `NONE`。如果使用了 `AWS_IAM` 授权，那么您需要使用签名版本 4 (SigV4) 协议对请求进行签名。有关此类请求的示例，请参阅the section called "教程：使用 HTTP 非代理集成构建 API"。

选择 HTTP 集成教程

要使用 HTTP 集成构建 API，您可以使用 HTTP 代理集成或 HTTP 自定义集成。

在 HTTP 代理集成中，您只需根据后端要求设置 HTTP 方法和 HTTP 端点 URI。我们建议您尽可能使用 HTTP 代理集成，以利用简化版的 API 设置过程。

如果您需要为后端转换客户端请求数据或为客户端转换后端响应数据，则您可能要使用 HTTP 自定义集成。

主题

- 教程：使用 HTTP 代理集成构建 REST API
- 教程：使用 HTTP 非代理集成构建 REST API

教程：使用 HTTP 代理集成构建 REST API

HTTP 代理集成是一个简单、强大的多功能机制，用于构建 API，它允许 Web 应用程序通过简单设置单个 API 方法即可访问集成的 HTTP 终端节点的多个资源或特征，例如整个网站。在 HTTP 代理集成中，API Gateway 会将客户端提交的方法请求传递至后端。传递的请求数据包括请求标头、查询字符串参数、URL 路径变量和负载。后端 HTTP 终端节点或 Web 服务器会对传入请求数据进行解析，以确定要返回的响应。在设置 API 方法后，HTTP 代理集成使得客户端和后端可以直接交互，而不受 API Gateway 的任何干预。不受支持的字符等已知问题除外，the section called “重要提示” 中列出了此类问题。

借助无所不包的代理资源 `{proxy+}` 和用于 HTTP 方法的“捕获全部”ANY 动词，您可以使用 HTTP 代理集成来创建单个 API 方法的 API。该方法会公开网站的一整套可公开访问的 HTTP 资源和操作。当后端 Web 服务器打开更多资源以供公开访问时，客户端可以通过相同的 API 设置来使用这些新资源。为了实现此功能，网站开发人员必须向客户端开发人员讲清楚什么是新资源以及适用于每个新资源的操作是什么。

作为快速介绍，以下教程演示了 HTTP 代理集成。在本教程中，我们将使用 API Gateway 控制台创建一个 API，以通过通用代理资源 `{proxy+}` 与 PetStore 网站集成，还将创建 ANY 的 HTTP 方法占位符。

主题

- 通过 API Gateway 控制台使用 HTTP 代理集成创建 API
- 使用 HTTP 代理集成测试 API

通过 API Gateway 控制台使用 HTTP 代理集成创建 API

以下过程将指导您完成在 API Gateway 控制台中使用代理资源创建并测试用于 HTTP 后端的 API 的步骤。HTTP 后端是来自 `PetStore` 的 `http://petstore-demo-endpoint.execute-api.com/petstore/pets` 网站 (教程：使用 HTTP 非代理集成构建 REST API)，其中的屏幕截图用作直观辅助手段来阐释 API Gateway UI 元素。如果您是首次使用 API Gateway 控制台来创建 API，可能需要先按照该部分中的说明操作。

创建 API

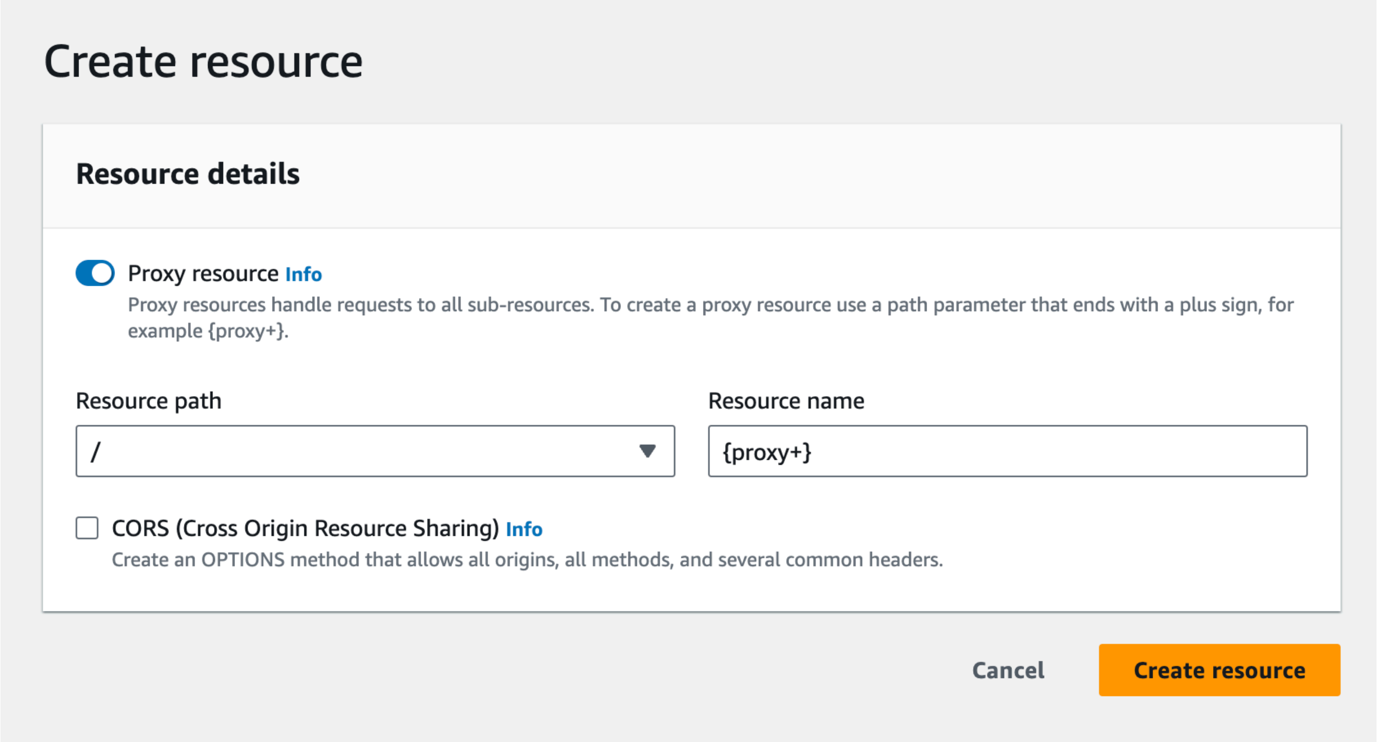
1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 如果您是第一次使用 API Gateway，您会看到一个介绍服务特征的页面。在 REST API 下，选择生成。当创建示例 API 弹出框出现时，选择确定。

   如果这不是您首次使用 API Gateway，请选择创建 API。在 REST API 下，选择生成。
3. 对于 API 名称，请输入 **`HTTPProxyAPI`**。
4. （可选）对于描述，输入描述。
5. 将 API 端点类型设置保留为区域。
6. 选择创建 API。

在此步骤中，您将创建一个 `{proxy+}` 代理资源路径。这是 `http://petstore-demo-endpoint.execute-api.com/` 下任意后端端点的占位符。例如，它可以是 `petstore`、`petstore/pets` 和 `petstore/pets/{petId}`。API Gateway 可在您创建 `{proxy+}` 资源时创建 `ANY` 方法，并在运行时用作任意支持的 HTTP 动词的占位符。

创建 /{proxy+} 资源

1. 选择 API。
2. 在主导航窗格中，选择资源。
3. 选择创建资源。
4. 开启代理资源。
5. 将资源路径保持为 `/`。
6. 对于资源名称，输入 **`{proxy+}`**。
7. 将 CORS（跨源资源共享）保持为关闭状态。
8. 选择创建资源。



Create resource

Resource details

Proxy resource Info

Proxy resources handle requests to all sub-resources. To create a proxy resource use a path parameter that ends with a plus sign, for example {proxy+}.

Resource path: /

Resource name: {proxy+}

☐ CORS (Cross Origin Resource Sharing) Info

Create an OPTIONS method that allows all origins, all methods, and several common headers.

在此步骤中，您将使用代理集成将 ANY 方法与后端 HTTP 端点集成。在代理集成中，API Gateway 会将客户端提交的方法请求传递至后端，而不进行任何干预。

创建 **ANY** 方法

1. 选择 /{proxy+} 资源。
2. 选择 ANY 方法。
3. 在警告符号下，选择编辑集成。您无法部署具有方法但没有集成的 API。
4. 对于集成类型，选择 HTTP。
5. 打开 HTTP 代理集成。
6. 对于 HTTP 方法，选择 ANY。
7. 对于端点 URL，输入 **`http://petstore-demo-endpoint.execute-api.com/{proxy}`**。
8. 选择保存。

使用 HTTP 代理集成测试 API

特定客户端请求是否成功取决于以下因素：

- 后端是否提供了相应的后端终端节点，如果已提供，是否授予了所需的访问权限。

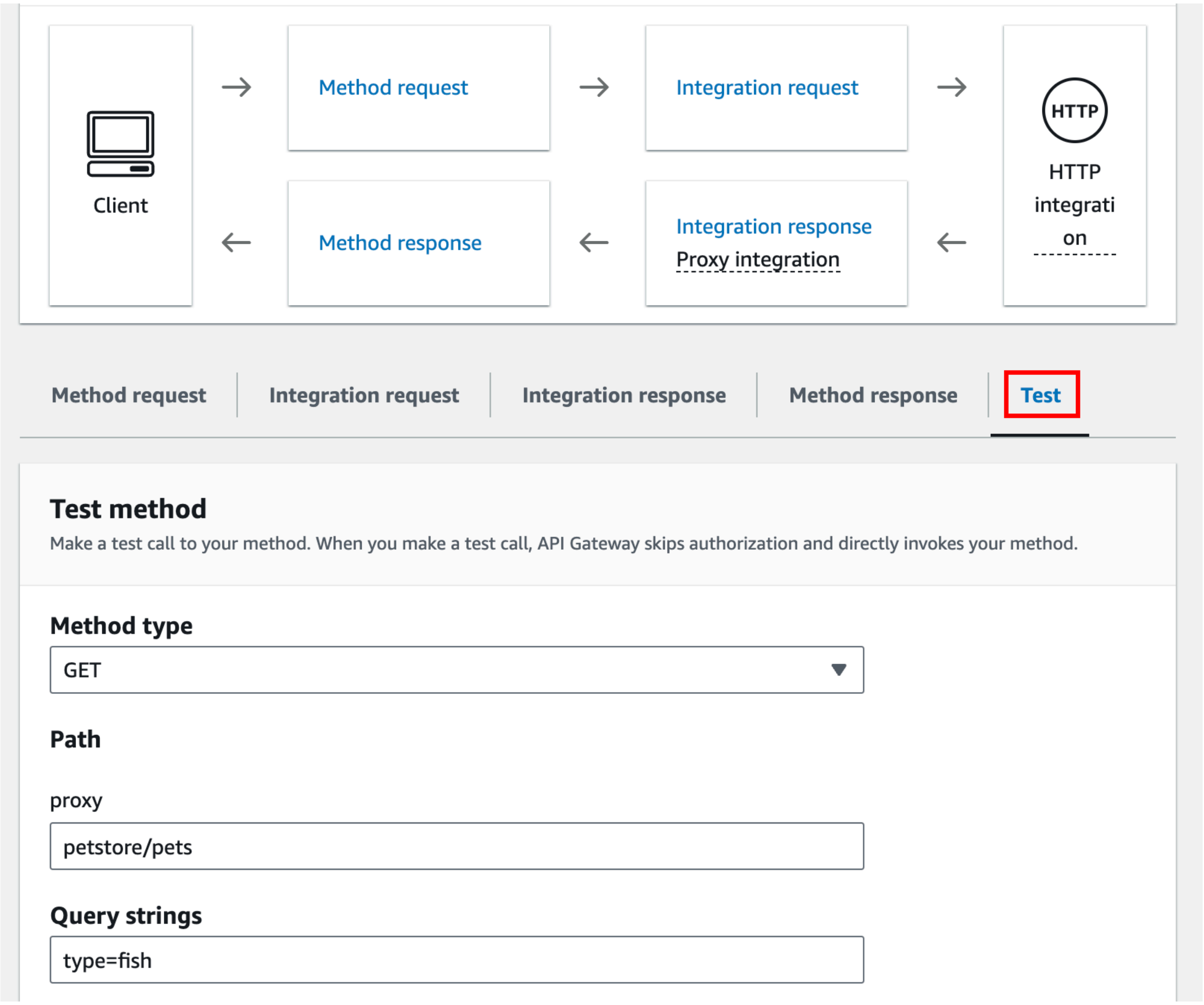
- 客户端是否提供了正确输入。

例如，此处使用的 PetStore API 不会公开 `/petstore` 资源。因此，您会收到 `404 Resource Not Found` 响应，其中包含错误消息 `Cannot GET /petstore`。

此外，客户端必须能够处理后端的输出格式，以便正确解析结果。API Gateway 不会通过调解来促进客户端与后端之间的交互。

通过代理资源使用 HTTP 代理集成测试与 PetStore 网站集成的 API

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
2. 对于方法类型，选择 `GET`。
3. 对于路径，在代理下输入 **`petstore/pets`**。
4. 对于查询字符串，输入 **`type=fish`**。
5. 选择测试。



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Method type

GET

Path

proxy

petstore/pets

Query strings

type=fish

由于后端网站支持 `GET /petstore/pets?type=fish` 请求，它将返回类似于以下内容的成功响应：

```
[
  {
    "id": 1,
    "type": "fish",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "fish",
    "price": 124.99
```

```
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
```

如果尝试调用 `GET /petstore`，您将会收到 `404` 响应，并显示一条错误消息：`Cannot GET /petstore`。这是因为后端不支持指定的操作。如果调用 `GET /petstore/pets/1`，您会收到包含以下负载的 `200 OK` 响应，因为 PetStore 网站支持该请求。

```
{
  "id": 1,
  "type": "dog",
  "price": 249.99
}
```

您也可以使用浏览器来测试您的 API。部署您的 API 并将其关联到阶段以创建 API 的调用 URL。

部署 API

1. 选择部署 API。
2. 对于阶段，选择新建阶段。
3. 对于阶段名称，输入 **`test`**。
4. （可选）对于描述，输入描述。
5. 选择部署。

现在，客户端可以调用您的 API。

调用 API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 在主导航窗格中，选择阶段。
4. 在阶段详细信息下，选择复制图标以复制您 API 的调用 URL。

在 Web 浏览器中输入 API 的调用 URL。

完整的 URL 应类似于 `https://abcdef123.execute-api.us-east-2.amazonaws.com/test/petstore/pets?type=fish`。

您的浏览器向 API 发送 `GET` 请求。

5. 结果应与您在 API Gateway 控制台中使用测试功能时返回的结果相同。

教程：使用 HTTP 非代理集成构建 REST API

在本教程中，您将使用 Amazon API Gateway 控制台从头开始创建 API。您可以将控制台看作一个 API 设计室，并将其用于确定 API 的特征、试验其行为、构建 API 并分阶段部署您的 API。

主题

- 使用 HTTP 自定义集成创建 API
- （可选）映射请求参数

使用 HTTP 自定义集成创建 API

本部分详细介绍了创建资源、在资源上公开方法、配置方法来实现所需的 API 行为以及测试和部署 API 的步骤。

在此步骤中，您将创建空 API。在以下步骤中，您将创建资源和方法，以使用非代理 HTTP 集成将 API 连接到 `http://petstore-demo-endpoint.execute-api.com/petstore/pets` 端点。

创建 API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 如果您是第一次使用 API Gateway，您会看到一个介绍服务特征的页面。在 REST API 下，选择生成。当创建示例 API 弹出框出现时，选择确定。

   如果这不是您首次使用 API Gateway，请选择创建 API。在 REST API 下，选择生成。
3. 对于 API 名称，请输入 **`HTTPNonProxyAPI`**。
4. （可选）对于描述，输入描述。
5. 将 API 端点类型设置保留为区域。
6. 选择创建 API。

资源树显示了不带任何方法的根资源 (`/`)。在本练习中，我们将创建一个与 PetStore 网站 (http://petstore-demo-endpoint.execute-api.com/petstore/pets) 进行 HTTP 自定义集成的 API。出于说明目的，我们将创建一个 `/pets` 资源作为根的子项，并在该资源上公开一个 GET 方法，以便客户端从 PetStore 网站上检索可用宠物项目列表。

创建 /pets 资源

1. 选择 / 资源，然后选择创建资源。
2. 将代理资源保持为关闭状态。
3. 将资源路径保持为 `/`。
4. 对于资源名称，输入 **`pets`**。
5. 将 CORS（跨源资源共享）保持为关闭状态。
6. 选择创建资源。

在此步骤中，您将在 /pets 资源上创建 `GET` 方法。该 `GET` 方法已与 `http://petstore-demo-endpoint.execute-api.com/petstore/pets` 网站集成。适用于 API 方法的其他选项包括以下这些：

- POST，主要用于创建子资源。
- PUT，主要用于更新现有资源（也可用于创建子资源，但我们不建议这样做）。
- DELETE，用于删除资源。
- PATCH，用于更新资源。
- HEAD，主要用在测试场景中。它与 GET 相同，但不能返回资源表示。
- OPTIONS，可供调用方用于获取目标服务的可用通信选项相关信息。

对于集成请求的 HTTP 方法，您必须选择一种受后端支持的方法。对于 `HTTP` 或 `Mock integration`，方法请求和集成请求可以使用相同的 HTTP 动词。对于其他集成类型，方法请求与集成请求可能会使用不同的 HTTP 动词。例如，要调用 Lambda 函数，集成请求必须使用 `POST` 调用该函数，而方法请求可能会根据 Lambda 函数的逻辑使用任何 HTTP 动词来进行调用。

在 /pets 资源上创建 **`GET`** 方法

1. 选择 /pets 资源。
2. 选择创建方法。
3. 对于方法类型，选择 GET。

4. 对于集成类型，选择 HTTP 集成。
5. 将 HTTP 代理集成保持为关闭状态。
6. 对于 HTTP 方法，选择 GET。
7. 对于端点 URL，输入 **`http://petstore-demo-endpoint.execute-api.com/petstore/pets`**。

   PetStore 网站可让您在给定页面上按宠物类型（例如“Dog”或“Cat”）检索 `Pet` 项目的列表。
8. 对于内容处理，选择传递。
9. 选择 URL 查询字符串参数。

   PetStore 网站使用 `type` 和 `page` 查询字符串参数来接受输入。您可以将查询字符串参数添加到方法请求中，并将其映射到集成请求的相应查询字符串参数中。
10. 要添加查询字符串参数，请执行以下操作：

    a. 选择添加查询字符串。

    b. 对于名称，输入 **`type`**。

    c. 保持必填和缓存为已关闭状态。

    重复上述步骤，再创建一个命名为 **`page`** 查询字符串。
11. 选择创建方法。

客户端现在可以在提交请求时提供一个宠物类型和页码作为查询字符串参数。这些输入参数必须映射到集成的查询字符串参数中，以便将输入值转发到后端的 PetStore 网站。

将输入参数映射到集成请求

1. 在集成请求选项卡的集成请求设置下，选择编辑。
2. 选择 URL 查询字符串参数，然后执行以下操作：

   a. 选择添加查询字符串参数。

   b. 对于名称，请输入 **`type`**。

   c. 对于映射自，输入 **`method.request.querystring.type`**。

   d. 将缓存保持为关闭状态。

   e. 选择添加查询字符串参数。

   f. 对于名称，请输入 **`page`**。

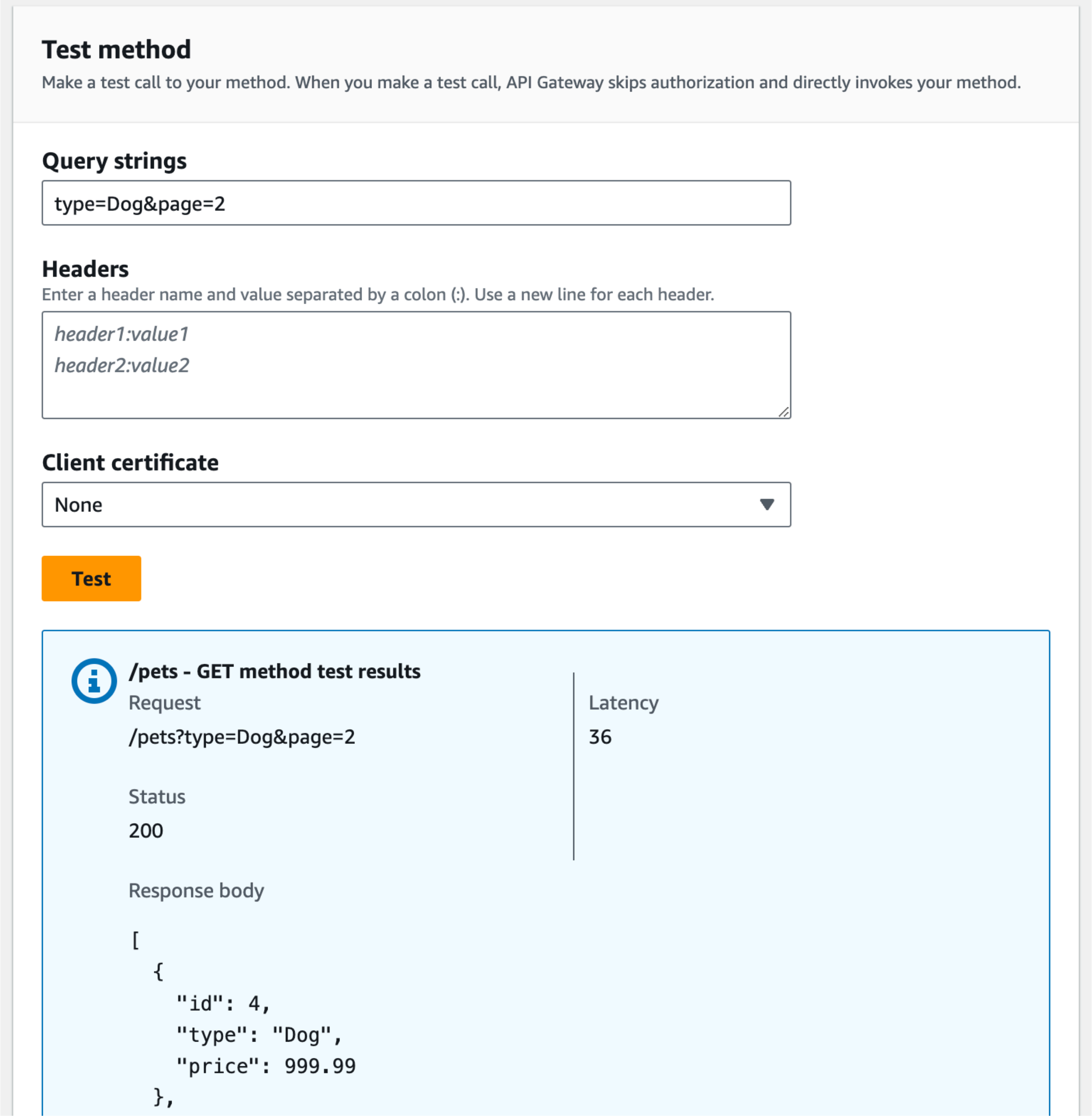
   g. 对于映射自，输入 **`method.request.querystring.page`**。
   h. 将缓存保持为关闭状态。

3. 选择保存。

测试 API

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
2. 对于查询字符串，输入 **`type=Dog&page=2`**。
3. 选择测试。

   结果类似于以下内容：



Query strings

type=Dog&page=2

Request

/pets?type=Dog&page=2

Latency

36

Status

200

Response body

```
[
  {
    "id": 4,
    "type": "Dog",
    "price": 999.99
  },
```

测试成功后，我们可以部署 API 以使其公开可用。

4. 选择部署 API。
5. 对于阶段，选择新建阶段。
6. 对于阶段名称，输入 **`Prod`**。
7. （可选）对于描述，输入描述。

8. 选择部署。
9. （可选）在阶段详细信息下，对于调用 URL，您可以选择复制图标以复制您 API 的调用 URL。您可以将此值与 Postman 和 cURL 等工具结合使用来测试您的 API。

如果您使用开发工具包创建客户端，您可以调用开发工具包公开的方法来对请求签名。有关具体实施方式，请参阅您选择的AWS开发工具包。

> **Note**
>
> 如果您的 API 发生更改，您必须重新部署 API 以便让新特征或更新后的特征生效，然后才能再次调用请求 URL。

（可选）映射请求参数

API Gateway API 的映射请求参数

本教程演示了如何在 API 的方法请求 URL 上创建 `{petId}` 路径参数，以指定一个项目 ID，将其映射到集成请求 URL 中的 `{id}` 路径参数，并将请求发送至 HTTP 端点。

> **Note**
>
> 如果未正确输入字母的大小写，例如应该输入大写字母时输入了小写字母，则可能会在稍后的演练中导致错误。

步骤 1：创建资源

在此步骤中，您将使用路径参数 {petId} 创建资源。

创建 {petId} 资源

1. 选择 /pets 资源，然后选择创建资源。
2. 将代理资源保持为关闭状态。
3. 对于资源路径，选择 /pets/。
4. 对于资源名称，输入 **`{petId}`**。

   在 `petId` 两边使用大括号 (`{ }`)，以便显示为 /pets/{petId}。
5. 将 CORS（跨源资源共享）保持为关闭状态。

6. 选择创建资源。

步骤 2：创建和测试方法

在此步骤中，您将使用 `{petId}` 路径参数创建 GET 方法。

设置 GET 方法

1. 选择 /{petId} 资源，然后选择创建方法。
2. 对于方法类型，选择 GET。
3. 对于集成类型，选择 HTTP 集成。
4. 将 HTTP 代理集成保持为关闭状态。
5. 对于 HTTP 方法，选择 GET。
6. 对于端点 URL，输入 **`http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}`**。
7. 对于内容处理，选择传递。
8. 将默认超时保持为开启状态。
9. 选择创建方法。

现在，将 `{petId}` 路径参数映射到 HTTP 端点中的 `{id}` 路径参数。

映射 **`{petId}`** 路径参数

1. 在集成请求选项卡的集成请求设置下，选择编辑。
2. 选择 URL 路径参数。
3. API Gateway 将为名为 petId 的集成请求创建路径参数。这不适用于您的后端。HTTP 端点使用 `{id}` 作为路径参数。将 petId 重命名为 **`id`**。

   这可将方法请求路径参数 `petId` 映射到集成请求路径参数 `id`。
4. 选择保存。

现在测试该方法。

测试 方法

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。

2. 在 petId 的路径下，输入 **4**。
3. 选择 Test (测试)。

   如果成功，响应正文将显示以下内容：

   ```
   {
     "id": 4,
     "type": "bird",
     "price": 999.99
   }
   ```

步骤 3：部署 API

在此步骤中，您将部署 API，以便在 API Gateway 控制台外部对其进行调用。

部署 API

1. 选择部署 API。
2. 对于阶段，选择 Prod。
3. （可选）对于描述，输入描述。
4. 选择 Deploy (部署)。

步骤 4：测试 API

在此步骤中，您将转到 API Gateway 控制台外部，并使用您的 API 访问 HTTP 端点。

1. 在主导航窗格中，选择阶段。
2. 在阶段详细信息下，选择复制图标以复制您 API 的调用 URL。

   它应该如下所示：

   ```
   https://my-api-id.execute-api.region-id.amazonaws.com/prod
   ```
3. 将此 URL 输入到新浏览器标签页的地址框中并在提交请求前将 `/pets/4` 附加到该 URL。
4. 浏览器将返回以下内容：

   ```
   {
     "id": 4,
   ```

```
  "type": "bird",
  "price": 999.99
}
```

后续步骤

您可以通过开启请求验证、转换数据或创建自定义网关响应来进一步自定义 API。

要探索更多自定义 API 的方法，请参阅以下教程：

- 有关请求验证的更多信息，请参阅在 API Gateway 中设置基本请求验证。
- 有关如何转换请求和响应负载的信息，请参阅在 API Gateway 中设置数据转换。
- 有关如何创建自定义网关响应的信息，请参阅使用 API Gateway 控制台为 REST API 设置网关响应。

教程：使用 API Gateway 私有集成构建 REST API

您可以使用私有集成创建 API Gateway API 向客户提供对您 Amazon Virtual Private Cloud (Amazon VPC) 中 HTTP/HTTPS 资源的访问。此类 VPC 资源是位于 VPC 中网络负载均衡器之后的 EC2 实例上的 HTTP/HTTPS 终端节点。网络负载均衡器封装 VPC 资源并将传入请求路由到目标资源。

客户端调用 API 时，API Gateway 通过预置的 VPC 链接连接到网络负载均衡器。VPC 链接由 VpcLink 的 API Gateway 资源封装。它负责将 API 方法请求转发到 VPC 资源，并将后端响应返回到调用方。对于 API 开发人员，`VpcLink` 的功能等同于集成终端节点。

要使用私有集成创建 API，您必须创建新的 `VpcLink`，或者选择已连接到针对所需 VPC 资源的网络负载均衡器的现有链接。您必须具有合适的权限以创建和管理 `VpcLink`。然后，您设置 API 方法并将其集成到 `VpcLink`，方法是将 `HTTP` 或 `HTTP_PROXY` 设置为集成类型，将 `VPC_LINK` 设置为集成连接类型，并在集成 `VpcLink` 上设置 `connectionId` 标识符。

> Note
>
> 网络负载均衡器和 API 必须归同一个AWS账户所有。

为快速开始创建 API 以访问 VPC 资源，我们将引导您完成必需步骤，通过 API Gateway 控制台使用私有集成来构建 API。开始创建 API 之前，请执行以下操作：

1. 创建 VPC 资源，在相同区域中您的账户下创建或选择网络负载均衡器，然后添加 EC2 实例，该实例托管了作为网络负载均衡器目标的资源。有关更多信息，请参阅 为 API Gateway 私有集成设置网络负载均衡器。
2. 授予权限，为私有集成创建 VPC 链接。有关更多信息，请参阅 授予创建 VPC 链接的权限。

创建您的 VPC 资源并使用在此目标组中配置的 VPC 资源创建网络负载均衡器之后，按照以下说明创建 API，并在私有集成中通过 `VpcLink` 将其与 VPC 资源集成。

使用私有集成创建 API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 如果您是第一次使用 API Gateway，您会看到一个介绍服务特征的页面。在 REST API 下，选择生成。当创建示例 API 弹出框出现时，选择确定。

   如果这不是您首次使用 API Gateway，请选择创建 API。在 REST API 下，选择生成。
3. 创建边缘优化或区域性 REST API。
4. 选择您的 API。
5. 选择创建方法，然后执行以下操作：
   a. 对于方法类型，选择 `GET`。
   b. 对于集成类型，选择 VPC 链接。
   c. 打开 VPC 代理集成。
   d. 对于 HTTP 方法，选择 `GET`。
   e. 对于 VPC 链接，选择 [使用阶段变量]，然后在下面的文本框中输入 **`${stageVariables.vpcLinkId}`**。

      在将 API 部署到阶段之后，您定义 `vpcLinkId` 阶段变量，并将其值设置为 `VpcLink` 的 ID。
   f. 对于端点 URL，输入 URL，例如 `http://myApi.example.com`。

      此处，主机名（例如 `myApi.example.com`）用于设置集成请求的标头 `Host`。
   g. 选择创建方法。

      通过代理集成，已准备好部署 API。否则，您需要继续设置适当的方法响应和集成响应。
6. 选择部署 API，然后执行以下操作：

   a. 对于阶段，选择新建阶段。
   b. 对于阶段名称，输入阶段名称。
   c. （可选）对于描述，输入描述。
   d. 选择部署。
7. 在阶段详细信息部分下，记下生成的调用 URL。您需要它来调用 API。执行此操作前，您必须设置 `vpcLinkId` 阶段变量。
8. 在阶段窗格中，选择阶段变量选项卡，然后执行以下操作：

   a. 选择管理变量，然后选择添加阶段变量。
   b. 对于名称，请输入 **`vpcLinkId`**。
   c. 对于值，输入 `VPC_LINK` 的 ID，例如 *`gix6s7`*。
   d. 选择保存。

   使用阶段变量，您可以通过更改阶段变量值，轻松地切换到 API 的不同 VPC 链接。

教程：使用AWS集成构建 API Gateway REST API

教程：使用 Lambda 代理集成构建 Hello World REST API 和 选择 AWS Lambda 集成教程主题介绍如何创建 API Gateway API 以公开集成的 Lambda 函数。此外，您还可以创建 API Gateway API 来公开其他AWS服务，例如 Amazon SNS、Amazon S3、Amazon Kinesis，甚至是 AWS Lambda。AWS 集成使其成为可能。Lambda 集成或 Lambda 代理集成是一种特殊情况，其中 Lambda 函数调用通过 API Gateway API 公开。

所有 AWS 服务都支持通过专用 API 来公开其特征。但是，应用程序协议或编程接口可能因服务而异。具有 AWS 集成的 API Gateway API 的优点是，可为您的客户端提供用于访问不同AWS服务的一致的应用程序协议。

在本演练中，我们创建了一个 API 来公开 Amazon SNS。有关将 API 与其他 AWS 服务集成的更多示例，请参阅 Amazon API Gateway 教程和研讨会。

与 Lambda 代理集成不同，没有用于其他AWS服务的相应代理集成。因此，API 方法与单个 AWS 操作集成。为了获得更大的灵活性，可以按照类似于代理集成的方式来设置 Lambda 代理集成。然后，Lambda 函数会为其他AWS操作解析和处理请求。

当终端节点超时的时候，API Gateway 不会重新尝试。API 调用方必须实施重试逻辑来处理终端节点超时。

本演练以 选择 AWS Lambda 集成教程 中的说明和概念为基础。如果您尚未完成该演练，我们建议您先完成它。

主题

先决条件

在开始本演练之前，请执行以下操作：

1. 完成 开始使用 API Gateway 的先决条件 中的步骤。
2. 创建名为 `MyDemoAPI` 的新 API。有关更多信息，请参阅 教程：使用 HTTP 非代理集成构建 REST API。
3. 至少将 API 部署到名为 `test` 的阶段一次。有关更多信息，请参阅中的部署 API选择 AWS Lambda 集成教程。
4. 完成中的剩余步骤选择 AWS Lambda 集成教程
5. 在 Amazon Simple Notification Service (Amazon SNS) 中至少创建一个主题。您将使用已部署的 API 获取 Amazon SNS 中与您的AWS账户相关联的主题列表。要了解如何在 Amazon SNS 中创建主题，请参阅创建主题。(您不需要复制步骤 5 中提到的主题 ARN。)

步骤 1：创建 AWS 服务代理执行角色

要允许 API 调用 Amazon SNS 操作，您必须已将适当的 IAM policy 附加到 IAM 角色。

创建 AWS 服务代理执行角色

1. 登录 AWS Management Console，然后使用以下网址打开 IAM 控制台：https://console.aws.amazon.com/iam/。

2. 选择角色。
3. 选择 Create role(创建角色)。
4. 在选择受信任实体的类型下选择 AWS 服务，然后选择 API Gateway 并选择允许 API Gateway 将日志推送到 CloudWatch Logs。
5. 选择下一步，然后再次选择下一步。
6. 对于角色名称，输入 **`APIGatewaySNSProxyPolicy`**，然后选择创建角色。
7. 在 Roles 列表中，选择您刚创建的角色。您可能需要滚动或使用搜索栏来查找角色。
8. 对于所选角色，选择添加权限选项卡。
9. 从下拉列表中选择附加策略。
10. 在搜索栏中，输入 **`AmazonSNSReadOnlyAccess`** 然后选择添加权限。

    > ⓘ Note
    >
    > 为简单起见，本教程使用托管策略。作为最佳实践，您应创建自己的 IAM 策略以授予所需的最低权限。

11. 记下新创建的角色 ARN，稍后将使用它。

步骤 2：创建资源

在此步骤中，您将创建一个资源，使 AWS 服务代理能够与 AWS 服务进行交互。

创建资源

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 选择根资源 /（由一个正斜杠 (/) 表示），然后选择创建资源。
4. 将代理资源保持为关闭状态。
5. 将资源路径保持为 `/`。
6. 对于资源名称，输入 **`mydemoawsproxy`**。
7. 将 CORS（跨源资源共享）保持为关闭状态。
8. 选择创建资源。

步骤 3：创建 GET 方法

在此步骤中，您将创建一个 GET 方法，使 AWS 服务代理能够与 AWS 服务进行交互。

创建 **GET** 方法

1. 选择 /mydemoawsproxy 资源，然后选择创建方法。
2. 对于方法类型，选择 GET。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您创建 Amazon SNS 主题的 AWS 区域。
5. 对于 AWS 服务，选择 Amazon SNS。
6. 将 AWS 子域保留为空白。
7. 对于 HTTP 方法，选择 GET。
8. 对于操作类型，选择使用操作名称。
9. 对于操作名称，输入 **`ListTopics`**。
10. 对于执行角色，输入 **`APIGatewaySNSProxyPolicy`** 的角色 ARN。
11. 选择创建方法。

步骤 4：指定方法设置并测试方法

现在，您可以测试 GET 方法来验证它是否已经过正确设置，可以列出您的 Amazon SNS 主题。

测试 **GET** 方法

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
2. 选择测试。

   结果显示与以下内容类似的响应：

   ```
   {
     "ListTopicsResponse": {
       "ListTopicsResult": {
         "NextToken": null,
         "Topics": [
           {
             "TopicArn": "arn:aws:sns:us-east-1:80398EXAMPLE:MySNSTopic-1"
           },
           {
   ```

```
          "TopicArn": "arn:aws:sns:us-east-1:80398EXAMPLE:MySNSTopic-2"
        },
        ...
        {
          "TopicArn": "arn:aws:sns:us-east-1:80398EXAMPLE:MySNSTopic-N"
        }
      ]
    },
    "ResponseMetadata": {
      "RequestId": "abc1de23-45fa-6789-b0c1-d2e345fa6b78"
    }
  }
}
```

步骤 5：部署 API

在此步骤中，您将部署 API，以便在 API Gateway 控制台外部对其进行调用。

部署 API

1. 选择部署 API。
2. 对于阶段，选择新建阶段。
3. 对于阶段名称，输入 **`test`**。
4. （可选）对于描述，输入描述。
5. 选择 Deploy (部署)。

步骤 6：测试 API

在本步骤中，您将转到 API Gateway 控制台外部，并使用您的AWS服务代理与 Amazon SNS 服务进行交互。

1. 在主导航窗格中，选择阶段。
2. 在阶段详细信息下，选择复制图标以复制您 API 的调用 URL。

   它应如下所示：

   ```
   https://my-api-id.execute-api.region-id.amazonaws.com/test
   ```
3. 将 URL 输入到新浏览器标签页的地址框中。

4. 附加 `/mydemoawsproxy`，使 URL 如下所示：

   ```
   https://my-api-id.execute-api.region-id.amazonaws.com/test/mydemoawsproxy
   ```

   浏览到该 URL。此时应显示以下信息：

   ```
   {"ListTopicsResponse":{"ListTopicsResult":{"NextToken": null,"Topics":
   [{"TopicArn": "arn:aws:sns:us-east-1:80398EXAMPLE:MySNSTopic-1"},{"TopicArn":
    "arn:aws:sns:us-east-1:80398EXAMPLE:MySNSTopic-2"},...{"TopicArn":
    "arn:aws:sns:us-east-1:80398EXAMPLE:MySNSTopic-N}]},"ResponseMetadata":
   {"RequestId":"abc1de23-45fa-6789-b0c1-d2e345fa6b78}}}
   ```

步骤 7：清除

您可以删除AWS服务代理运行所需的 IAM 资源。

> ⚠ Warning
>
> 如果您删除AWS服务代理所依赖的 IAM 资源，那么该AWS服务代理和任何依赖它的 API 将无法正常运行。IAM 资源删除操作无法撤消。如果要再次使用 IAM 资源，您必须重新创建它。

删除相关联的 IAM 资源

1. 通过以下网址打开 IAM 控制台：https://console.aws.amazon.com/iam/。
2. 在详细信息区域中，选择角色。
3. 选择 APIGatewayAWSProxyExecRole，然后依次选择角色操作、删除角色。在系统提示时，选择 Yes, Delete。
4. 在详细信息区域中，选择策略。
5. 选择 APIGatewayAWSProxyExecPolicy，然后选择策略操作和删除。系统提示时，选择删除。

本演练到此结束。有关创建 API 作为AWS服务代理的更深入讨论，请参阅教程：在 API Gateway 中创建 REST API 作为 Amazon S3 代理、教程：通过两种 AWS 服务集成和一种 Lambda 非代理集成创建计算器 REST API或教程：在 API Gateway 中创建 REST API 作为 Amazon Kinesis 代理。

教程：通过两种 AWS 服务集成和一种 Lambda 非代理集成创建计算器 REST API

非代理集成教程入门完全使用 `Lambda Function` 集成。`Lambda Function` 集成是 `AWS Service` 集成类型的特殊情况，该类型可为您执行大量集成设置，如自动添加所需的基于资源的权限以调用 Lambda 函数。在这里，三个集成中的两个集成使用的是 `AWS Service` 集成。在此集成类型中，您将具有更多控制，但您将需要手动执行任务，如创建和指定包含相应权限的 IAM 角色。

在本教程中，您将创建一个 `Calc` Lambda 函数，该函数可实施基本算术运算，同时接受 JSON 格式的输入和输出。然后，您将采用以下方式创建一个 REST API 并将其与 Lambda 函数集成：

1. 通过在 `GET` 资源上公开 `/calc` 方法以调用 Lambda 函数，同时提供输入作为查询字符串参数。（`AWS Service` 集成）
2. 通过在 `POST` 资源上公开 `/calc` 方法以调用 Lambda 函数，同时在方法请求负载中提供输入。（`AWS Service` 集成）
3. 通过在嵌套的 `GET` 资源上公开 `/calc/{operand1}/{operand2}/{operator}` 以调用 Lambda 函数，同时提供输入作为路径参数。（`Lambda Function` 集成）

除了尝试使用本教程外，您可能还希望研究 `Calc` API 的 OpenAPI 定义文件，您可通过按照 the section called “OpenAPI” 中的说明操作来将其导入 API Gateway 中。

主题

- 创建一个可代入的 IAM 角色
- 创建 Calc Lambda 函数
- 测试 Calc Lambda 函数
- 创建 Calc API
- 集成 1：创建使用查询参数的 GET 方法以调用 Lambda 函数
- 集成 2：创建使用 JSON 负载的 POST 方法以调用 Lambda 函数
- 集成 3：创建使用路径参数的 GET 方法以调用 Lambda 函数
- 与 Lambda 函数集成的示例 API 的 OpenAPI 定义

创建一个可代入的 IAM 角色

为了确保您的 API 调用您的 `Calc` Lambda 函数，您将需要具有一个 API Gateway 可代入的 IAM 角色，这是一个具有以下信任关联的 IAM 角色：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

您创建的角色将需要具有 Lambda InvokeFunction 权限。否则，API 调用方将收到 `500 Internal Server Error` 响应。要向该角色提供此权限，请将以下 IAM 策略附加到它：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "*"
        }
    ]
}
```

以下是完成所有操作的方式：

创建 API Gateway 可代入的 IAM 角色

1. 登录 IAM 控制台。
2. 选择角色。
3. 选择创建角色。
4. 在选择受信任实体的类型下，选择 AWS 服务。

5. 在选择将使用此角色的服务下，选择 Lambda。
6. 选择下一步: 权限。
7. 选择创建策略。

   新的创建策略控制台窗口将会打开。在该窗口中，执行以下操作：

   a. 在 JSON 选项卡中，将现有策略替换为以下策略：

   ```
   {
       "Version": "2012-10-17",
       "Statement": [
           {
               "Effect": "Allow",
               "Action": "lambda:InvokeFunction",
               "Resource": "*"
           }
       ]
   }
   ```

   b. 选择查看策略。
   c. 在查看策略下，执行以下操作：
      i. 在名称中，键入一个名称，如 **lambda_execute**。
      ii. 选择创建策略。
8. 在原来的创建角色控制台窗口中，执行以下操作：

   a. 从下拉列表的添加权限策略下，选择您的 **lambda_execute** 策略。

      如果您未在策略列表中看到您的策略，请选择列表顶部的刷新按钮。（请勿刷新浏览器页面！）
   b. 选择下一步: 标签。
   c. 选择下一步: 审核。
   d. 对于角色名称，键入一个名称，如 **lambda_invoke_function_assume_apigw_role**。
   e. 选择创建角色。
9. 从角色列表中，选择您的 **lambda_invoke_function_assume_apigw_role**。
10. 选择信任关系选项卡。

11. 选择编辑信任关系。
12. 使用以下策略替换现有策略：

    ```
    {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Sid": "",
          "Effect": "Allow",
          "Principal": {
            "Service": [
              "lambda.amazonaws.com",
              "apigateway.amazonaws.com"
            ]
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
    ```

13. 选择更新信任策略。
14. 请记下您刚刚创建的角色的角色 ARN。您以后将需要它。

创建 `Calc` Lambda 函数

接下来，您将使用 Lambda 控制台创建 Lambda 函数。

1. 在 Lambda 控制台中，选择创建函数。
2. 选择从头开始创作。
3. 在名称中，输入 **`Calc`**。
4. 在运行时中，选择受支持的最新 Node.js 或 Python 运行时。
5. 选择创建函数。
6. 在首选运行时中复制以下 Lambda 函数，并将其粘贴到 Lambda 控制台的代码编辑器中。

   Node.js

   ```
   export const handler = async function (event, context) {
   ```

```
  console.log("Received event:", JSON.stringify(event));

  if (
    event.a === undefined ||
    event.b === undefined ||
    event.op === undefined
  ) {
    return "400 Invalid Input";
  }

  const res = {};
  res.a = Number(event.a);
  res.b = Number(event.b);
  res.op = event.op;
  if (isNaN(event.a) || isNaN(event.b)) {
    return "400 Invalid Operand";
  }
  switch (event.op) {
    case "+":
    case "add":
      res.c = res.a + res.b;
      break;
    case "-":
    case "sub":
      res.c = res.a - res.b;
      break;
    case "*":
    case "mul":
      res.c = res.a * res.b;
      break;
    case "/":
    case "div":
      if (res.b == 0) {
        return "400 Divide by Zero";
      } else {
        res.c = res.a / res.b;
      }
      break;
    default:
      return "400 Invalid Operator";
  }

  return res;
```

```
};
```

Python

```
import json


def lambda_handler(event, context):
    print(event)

    try:
        (event['a']) and (event['b']) and (event['op'])
    except KeyError:
        return '400 Invalid Input'

    try:
        res = {
            "a": float(
                event['a']), "b": float(
                event['b']), "op": event['op']}
    except ValueError:
        return '400 Invalid Operand'

    if event['op'] == '+':
        res['c'] = res['a'] + res['b']
    elif event['op'] == '-':
        res['c'] = res['a'] - res['b']
    elif event['op'] == '*':
        res['c'] = res['a'] * res['b']
    elif event['op'] == '/':
        if res['b'] == 0:
            return '400 Divide by Zero'
        else:
            res['c'] = res['a'] / res['b']
    else:
        return '400 Invalid Operator'

    return res
```

7. 在“执行角色”下，选择选择现有角色。
8. 输入您之前创建的 **`lambda_invoke_function_assume_apigw_role`** 角色的角色 ARN。
9. 选择部署。

此函数需要来自 a 输入参数的两个操作数 (b 和 op) 以及一个运算符 (event)。该输入是格式如下的 JSON 对象：

```
{
  "a": "Number" | "String",
  "b": "Number" | "String",
  "op": "String"
}
```

此函数会返回计算所得的结果 (c) 和输入。对于无效的输入，该函数将返回空值或“Invalid op”字符串作为结果。输出具有以下 JSON 格式：

```
{
  "a": "Number",
  "b": "Number",
  "op": "String",
  "c": "Number" | "String"
}
```

您应该先在 Lambda 控制台中测试该函数，然后再在下一步中将其与 API 集成。

测试 `Calc` Lambda 函数

以下内容介绍如何在 Lambda 控制台中测试您的 `Calc` 函数：

1. 选择测试选项卡。
2. 对于测试事件名称，请输入 **`calc2plus5`**。
3. 将测试事件定义替换为以下内容：

   ```
   {
     "a": "2",
     "b": "5",
     "op": "+"
   }
   ```

4. 选择保存。
5. 选择测试。

6. 展开执行结果: 成功。您将看到以下内容：

   ```
   {
     "a": 2,
     "b": 5,
     "op": "+",
     "c": 7
   }
   ```

创建 `Calc` API

以下过程介绍如何为您刚刚创建的 `Calc` Lambda 函数创建 API。在后续部分中，您将资源和方法添加到该 API。

创建 API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 如果您是第一次使用 API Gateway，您会看到一个介绍服务特征的页面。在 REST API 下，选择生成。当创建示例 API 弹出框出现时，选择确定。

   如果这不是您首次使用 API Gateway，请选择创建 API。在 REST API 下，选择生成。
3. 对于 API 名称，请输入 **`LambdaCalc`**。
4. （可选）对于描述，输入描述。
5. 将 API 端点类型设置保留为区域。
6. 选择 Create API (创建 API)。

集成 1：创建使用查询参数的 `GET` 方法以调用 Lambda 函数

通过创建将查询字符串参数传递给 Lambda 函数的 `GET` 方法，启用要从浏览器中进行调用的 API。这种方法可能非常有用，特别是对于允许开放访问的 API。

创建 API 后，您将创建一个资源。通常情况下，根据应用程序逻辑将 API 资源组织成资源树形式。在此步骤中，您将创建一个 /calc 资源。

创建 /calc 资源

1. 选择创建资源。

2. 将代理资源保持为关闭状态。
3. 将资源路径保持为 `/`。
4. 对于资源名称，输入 **`calc`**。
5. 将 CORS（跨源资源共享）保持为关闭状态。
6. 选择创建资源。

通过创建将查询字符串参数传递给 Lambda 函数的 `GET` 方法，启用要从浏览器中进行调用的 API。这种方法可能非常有用，特别是对于允许开放访问的 API。

在此方法中，Lambda 要求使用 `POST` 请求调用任何 Lambda 函数。此示例表明，前端方法请求中的 HTTP 方法可以与后端中的集成请求不同。

创建 **`GET`** 方法

1. 选择 /calc 资源，然后选择创建方法。
2. 对于方法类型，选择 GET。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您创建 Lambda 函数的 AWS 区域。
5. 对于 AWS 服务，选择 Lambda。
6. 将 AWS 子域保留为空白。
7. 对于 HTTP 方法，选择 POST。
8. 对于操作类型，选择使用路径覆盖。借助此选项，我们可以指定用来执行 `Calc` 函数的调用操作的 ARN。
9. 对于路径覆盖，输入 **`2015-03-31/functions/arn:aws:lambda:`*`us-east-2`*`:`*`account-id`*`:function:Calc/invocations`**。对于 *`account-id`*，输入您的 AWS 账户 ID。对于 *`us-east-2`*，输入您创建 Lambda 函数的 AWS 区域。
10. 对于执行角色，输入 **`lambda_invoke_function_assume_apigw_role`** 的角色 ARN。
11. 请勿更改凭证缓存和默认超时的设置。
12. 选择方法请求设置。
13. 对于请求验证程序，选择验证查询字符串参数和标头。

    如果客户端没有指定必需的参数，此设置将导致返回错误消息。
14. 选择 URL 查询字符串参数。

现在，为 /calc 资源上的 GET 方法设置查询字符串参数，来代表后端 Lambda 函数接收输入。

要创建查询字符串参数，请执行以下操作：

a. 选择添加查询字符串。
b. 在名称中，输入 **operand1**。
c. 打开必需。
d. 将缓存保持为关闭状态。

重复相同的步骤，创建一个名为 **operand2** 的查询字符串和一个名为 **operator** 的查询字符串。

15. 选择创建方法。

现在，创建映射模板，以将客户端提供的查询字符串转换为 `Calc` 函数需要的集成请求负载。此模板会将方法请求中声明的三个查询字符串参数映射为 JSON 对象的指定属性值，作为后端 Lambda 函数的输入。转换后的 JSON 对象将作为集成请求负载包含在内。

将输入参数映射到集成请求

1. 在集成请求选项卡的集成请求设置下，选择编辑。
2. 对于请求正文传递，选择当未定义模板时（推荐）。
3. 选择映射模板。
4. 选择添加映射模板。
5. 对于内容类型，输入 **application/json**。
6. 对于模板正文，输入以下代码：

```
{
    "a":  "$input.params('operand1')",
    "b":  "$input.params('operand2')",
    "op": "$input.params('operator')"
}
```

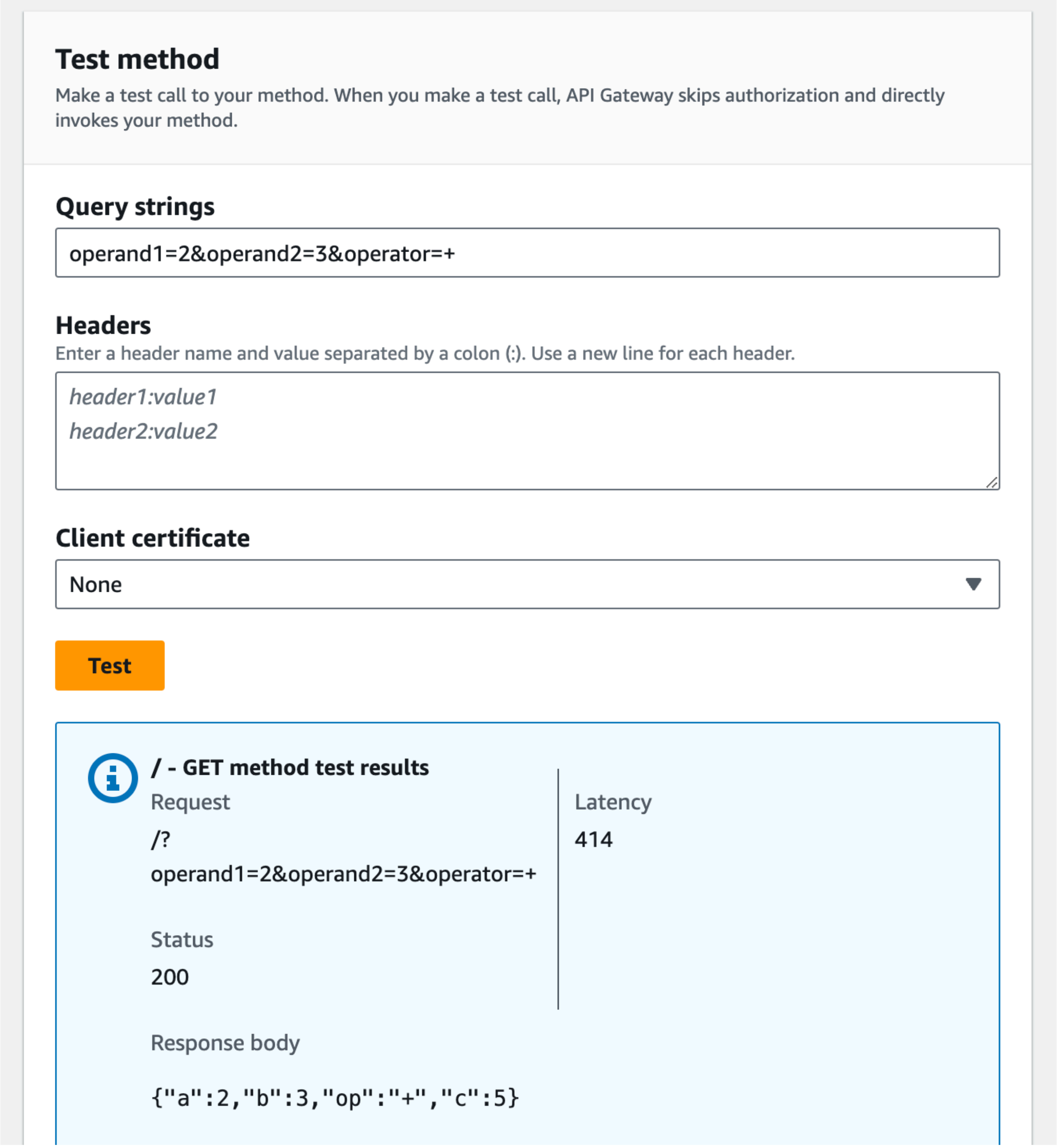
7. 选择保存。

现在，您可以测试 GET 方法来验证它是否已经过正确设置，可以调用 Lambda 函数。

测试 **`GET`** 方法

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
2. 对于查询字符串，输入 **`operand1=2&operand2=3&operator=+`**。
3. 选择 Test (测试)。

   结果应与如下显示类似：



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Query strings

operand1=2&operand2=3&operator=+

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1
header2:value2

Client certificate

None

/ - GET method test results

Request

/?operand1=2&operand2=3&operator=+

Latency

414

Status

200

Response body

```
{"a":2,"b":3,"op":"+","c":5}
```

集成 2：创建使用 JSON 负载的 **POST** 方法以调用 Lambda 函数

通过创建使用 JSON 负载的 POST 方法以调用 Lambda 函数完成此操作，以便客户端必须将必要的输入提供给请求正文中的后端函数。为确保客户端上传正确的输入数据，您将对负载启用请求验证。

创建使用 JSON 负载的 **POST** 方法

1. 选择 /calc 资源，然后选择创建方法。
2. 对于方法类型，选择 POST。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您创建 Lambda 函数的 AWS 区域。
5. 对于 AWS 服务，选择 Lambda。
6. 将 AWS 子域保留为空白。
7. 对于 HTTP 方法，选择 POST。
8. 对于操作类型，选择使用路径覆盖。借助此选项，我们可以指定用来执行 `Calc` 函数的调用操作的 ARN。
9. 对于路径覆盖，输入 **`2015-03-31/functions/arn:aws:lambda:`*`us-east-2`*`:`*`account-id`*`:function:Calc/invocations`**。对于 ***`account-id`***，输入您的 AWS 账户 ID。对于 ***`us-east-2`***，输入您创建 Lambda 函数的 AWS 区域。
10. 对于执行角色，输入 **`lambda_invoke_function_assume_apigw_role`** 的角色 ARN。
11. 请勿更改凭证缓存和默认超时的设置。
12. 选择创建方法。

现在，创建输入模型以描述输入数据结构和验证传入请求正文。

创建输入模型

1. 在主导航窗格中，选择模型。
2. 选择创建模型。
3. 对于名称，请输入 **`input`**。
4. 对于内容类型，输入 **`application/json`**。

   如果未找到匹配的内容类型，则不执行请求验证。要使用同一模型而不考虑内容类型，请输入 **`$default`**。
5. 对于模型架构，输入以下模型：

```
{
    "type":"object",
    "properties":{
        "a":{"type":"number"},
```

```
        "b":{"type":"number"},
        "op":{"type":"string"}
    },
    "title":"input"
}
```

6. 选择创建模型。

现在，创建输出模型。此模型描述了后端计算得出的输出的数据结构。它可用于将集成响应数据映射到不同的模型。本教程依靠传递行为，并不会使用此模型。

创建输出模型

1. 选择创建模型。
2. 对于名称，请输入 **`output`**。
3. 对于内容类型，输入 **`application/json`**。

   如果未找到匹配的内容类型，则不执行请求验证。要使用同一模型而不考虑内容类型，请输入 **`$default`**。
4. 对于模型架构，输入以下模型：

```
{
    "type":"object",
    "properties":{
        "c":{"type":"number"}
    },
    "title":"output"
}
```

5. 选择创建模型。

现在，创建结果模型。此模型描述了返回响应数据的数据结构。它同时引用您的 API 中定义的输入和输出架构。

创建结果模型

1. 选择创建模型。
2. 对于名称，请输入 **`result`**。
3. 对于内容类型，输入 **`application/json`**。

如果未找到匹配的内容类型，则不执行请求验证。要使用同一模型而不考虑内容类型，请输入 **`$default`**。

4. 对于模型架构，使用您的 *`restapi-id`* 输入以下模型。您的 *`restapi-id`* 用括号列在控制台顶部，可通过以下流程找到：`API Gateway > APIs > LambdaCalc (abc123)`.

```
{
    "type":"object",
    "properties":{
        "input":{
            "$ref":"https://apigateway.amazonaws.com/restapis/restapi-id/models/
input"
        },
        "output":{
            "$ref":"https://apigateway.amazonaws.com/restapis/restapi-id/models/
output"
        }
    },
    "title":"result"
}
```

5. 选择创建模型。

现在，配置 POST 方法的方法请求，以便在传入的请求正文上启用请求验证。

在 POST 方法上启用请求验证

1. 在主导航窗格中，选择资源，然后从资源树中选择 `POST` 方法。
2. 在方法请求选项卡上的方法请求设置下，选择编辑。
3. 对于请求验证程序，选择验证正文。
4. 选择请求正文，然后选择添加模型。
5. 对于内容类型，输入 **`application/json`**。

   如果未找到匹配的内容类型，则不执行请求验证。要使用同一模型而不考虑内容类型，请输入 **`$default`**。
6. 对于模型，选择输入。
7. 选择保存。

现在，您可以测试 POST 方法来验证它是否已经过正确设置，可以调用 Lambda 函数。

测试 **POST** 方法

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
2. 对于请求正文，输入以下 JSON 负载。

```
{
    "a": 1,
    "b": 2,
    "op": "+"
}
```

3. 选择测试。

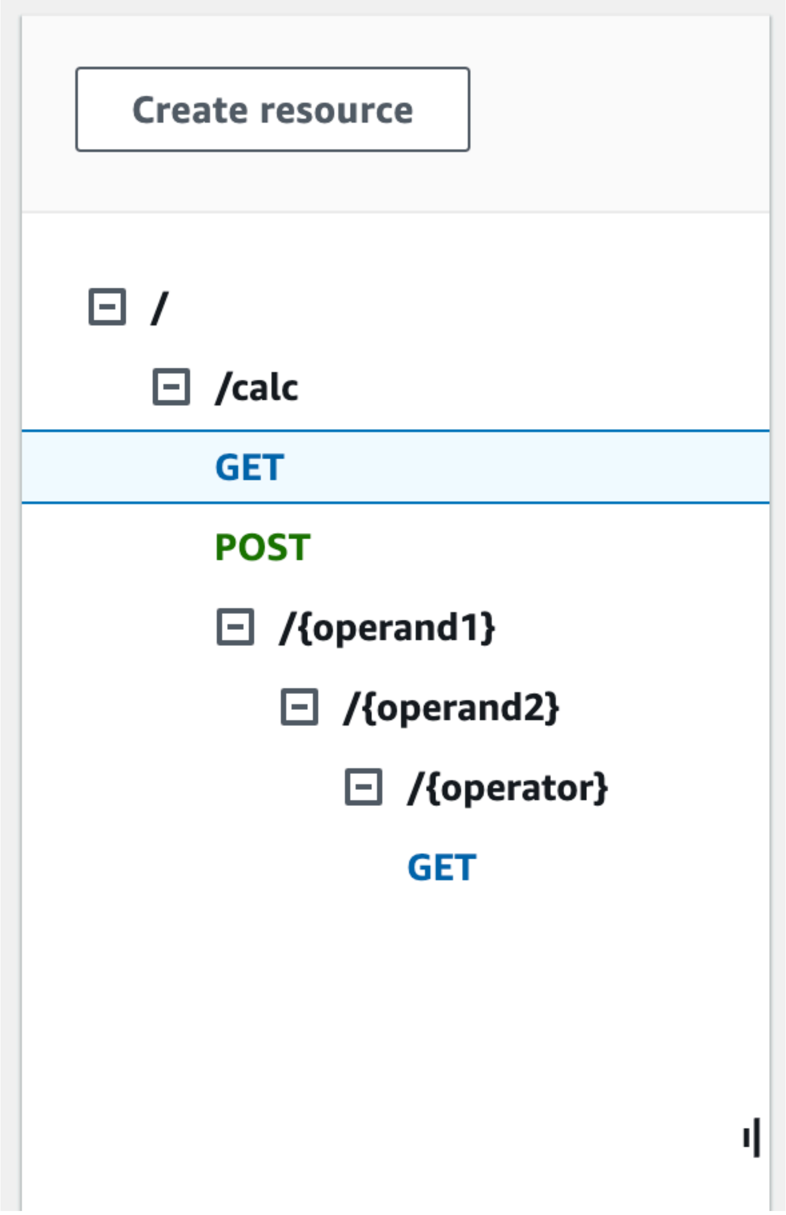
   您应看到以下输出：

```
{
  "a": 1,
  "b": 2,
  "op": "+",
  "c": 3
}
```

集成 3：创建使用路径参数的 **GET** 方法以调用 Lambda 函数

现在，您将在由一系列路径参数指定的资源上创建一个 GET 方法，以调用后端 Lambda 函数。路径参数值指定 Lambda 函数的输入数据。您将使用映射模板，将传入路径参数值映射到必需的集成请求负载。

生成的 API 资源结构将如下所示：



创建 /{operand1}/{operand2}/{operator} 资源

1. 选择创建资源。
2. 对于资源路径，选择 `/calc`。
3. 对于资源名称，输入 **{operand1}**。
4. 将 CORS（跨源资源共享）保持为关闭状态。
5. 选择创建资源。
6. 对于资源路径，选择 `/calc/{operand1}/`。
7. 对于资源名称，输入 **{operand2}**。

8. 将 CORS（跨源资源共享）保持为关闭状态。
9. 选择创建资源。
10. 对于资源路径，选择 `/calc/{operand1}/{operand2}/`。
11. 对于资源名称，输入 **`{operator}`**。
12. 将 CORS（跨源资源共享）保持为关闭状态。
13. 选择创建资源。

这次，您将在 API Gateway 控制台中使用内置的 Lambda 集成来设置方法集成。

设置方法集成

1. 选择 /{operand1}/{operand2}/{operator} 资源，然后选择创建方法。
2. 对于方法类型，选择 GET。
3. 对于集成类型，选择 Lambda。
4. 保持 Lambda 代理集成处于关闭状态。
5. 对于 Lambda 函数，选择您创建 Lambda 函数的 AWS 区域，并输入 **`Calc`**。
6. 将默认超时保持为开启状态。
7. 选择创建方法。

现在，创建映射模板，以将创建 /calc/{operand1}/{operand2}/{operator} 资源时声明的三个 URL 路径参数映射到 JSON 对象中的指定属性值。由于 URL 路径必须经过 URL 编码，必须将除法运算符指定为 `%2F`，而不是 `/`。此模板先将 `%2F` 转换为 `'/'`，然后再将其发送到 Lambda 函数。

创建映射模板

1. 在集成请求选项卡的集成请求设置下，选择编辑。
2. 对于请求正文传递，选择当未定义模板时（推荐）。
3. 选择映射模板。
4. 对于内容类型，输入 **`application/json`**。
5. 对于模板正文，输入以下代码：

```
{
   "a": "$input.params('operand1')",
   "b": "$input.params('operand2')",
```

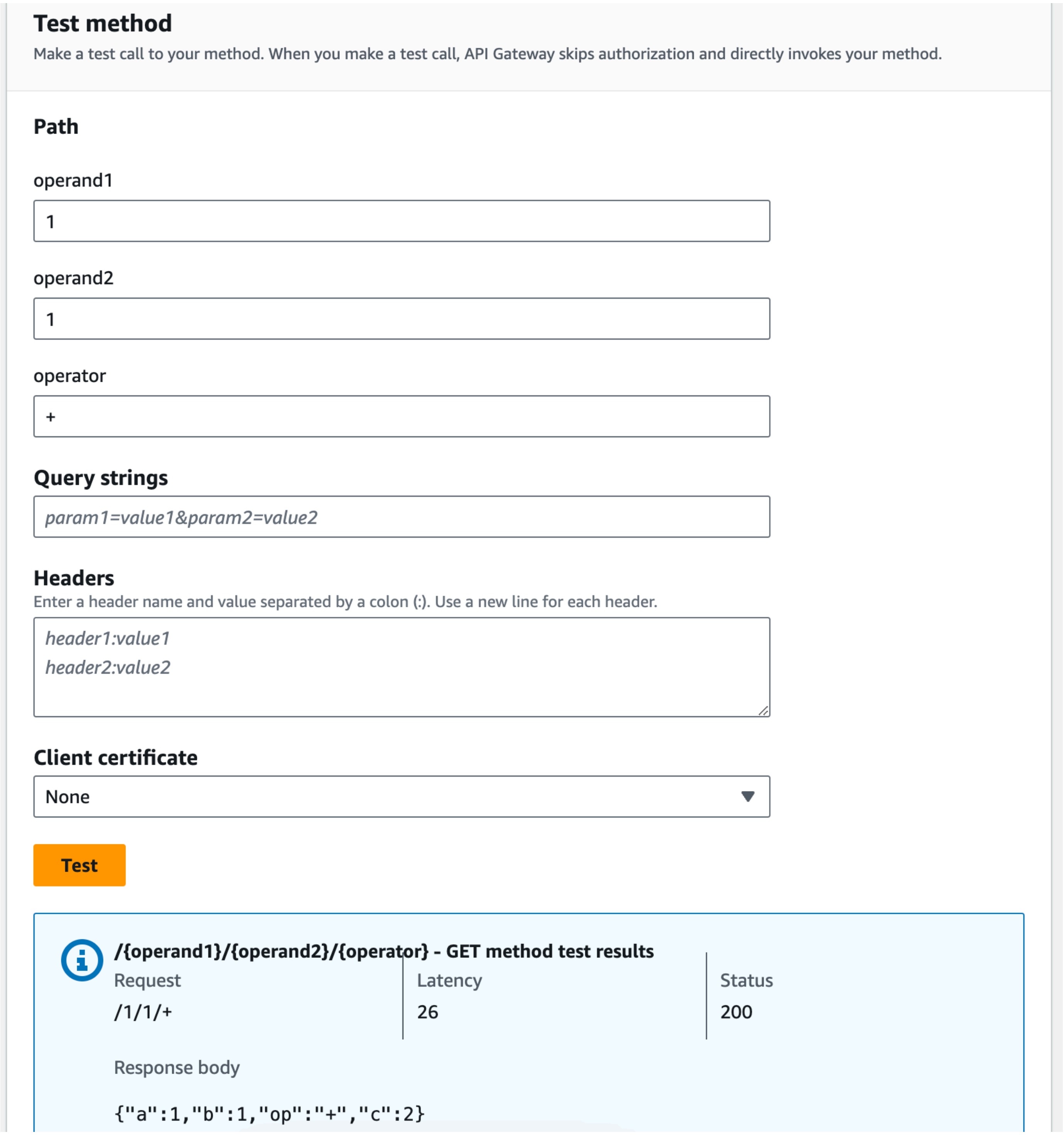
```
    "op":
 #if($input.params('operator')=='%2F')"/"#{else}"$input.params('operator')"#end
}
```

6. 选择保存。

现在，您可以测试 GET 方法来验证它是否已经过正确设置，无需映射即可通过集成响应调用 Lambda 函数并传递原始输出。

测试 **GET** 方法

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
2. 对于路径，请执行以下操作：
    a. 对于 operand1，输入 **1**。
    b. 对于 operand2，输入 **1**。
    c. 对于 operator，输入 **+**。
3. 选择 Test (测试)。
4. 结果应该如下所示：



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Path

operand1: 1

operand2: 1

operator: +

Query strings

param1=value1&param2=value2

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1
header2:value2

Client certificate

None

/{operand1}/{operand2}/{operator} - GET method test results

Request	Latency	Status
/1/1/+	26	200

Response body

```
{"a":1,"b":1,"op":"+","c":2}
```

接下来，根据 `result` 架构对方法响应负载的数据结构进行建模。

默认情况下，系统会为方法响应正文分配一个空白模型。这将在不映射的情况下传递集成响应正文。但是，当您为一种强类型语言（例如 Java 或 Objective-C）生成一个开发工具包时，您的开发工具包用户会收到一个空白对象作为结果。为确保 REST 客户端和开发工具包客户端能收到期望的结果，您必

须使用预定义的架构为响应数据建模。在这里，您将为方法响应正文定义一个模型，并构建一个映射模板，以便将集成响应正文转换为方法响应正文。

创建方法响应

1. 在方法响应选项卡的响应 200 下，选择编辑。
2. 在响应正文下，选择添加模型。
3. 对于内容类型，输入 **application/json**。
4. 对于模型，选择结果。
5. 选择保存。

为方法响应正文设置模型可以确保将响应数据转换为给定开发工具包的 `result` 对象。要确保对集成响应数据进行相应的映射，您将需要一个映射模板。

创建映射模板

1. 在集成响应选项卡上的默认 - 响应下，选择编辑。
2. 选择映射模板。
3. 对于内容类型，输入 **application/json**。
4. 对于模板正文，输入以下代码：

   ```
   #set($inputRoot = $input.path('$'))
   {
     "input" : {
       "a" : $inputRoot.a,
       "b" : $inputRoot.b,
       "op" : "$inputRoot.op"
     },
     "output" : {
       "c" : $inputRoot.c
     }
   }
   ```

5. 选择保存。

测试映射模板

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。

2. 对于路径，请执行以下操作：
    a. 对于 operand1，输入 `1`。
    b. 对于 operand2，输入 `2`。
    c. 对于 operator，输入 `+`。
3. 选择测试。
4. 结果将类似于以下内容：

```
{
  "input": {
    "a": 1,
    "b": 2,
    "op": "+"
  },
  "output": {
    "c": 3
  }
}
```

目前，您只能在 API Gateway 控制台中通过测试功能调用 API。要使 API 可用于客户端，您将需要部署 API。每当您添加、修改或删除资源和方法、更新数据映射或者更新阶段设置时，请始终确保重新部署 API。否则，新功能或更新将不可用于您 API 的客户端。如以下所示：

部署 API

1. 选择部署 API。
2. 对于阶段，选择新建阶段。
3. 对于阶段名称，输入 **`Prod`**。
4. （可选）对于描述，输入描述。
5. 选择部署。
6. （可选）在阶段详细信息下，对于调用 URL，您可以选择复制图标以复制您 API 的调用 URL。您可以将此值与 Postman 和 cURL 等工具结合使用来测试您的 API。

Note

每当您添加、修改或删除资源或方法、更新数据映射或者更新阶段设置时，请务必重新部署 API。否则，新特征或更新将不可用于您的 API 的客户端。

与 Lambda 函数集成的示例 API 的 OpenAPI 定义

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2017-04-20T04:08:08Z",
    "title": "LambdaCalc"
  },
  "host": "uojnr9hd57.execute-api.us-east-1.amazonaws.com",
  "basePath": "/test",
  "schemes": [
    "https"
  ],
  "paths": {
    "/calc": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "operand2",
            "in": "query",
            "required": true,
            "type": "string"
          },
          {
            "name": "operator",
            "in": "query",
            "required": true,
```

```
            "type": "string"
          },
          {
            "name": "operand1",
            "in": "query",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Result"
            },
            "headers": {
              "operand_1": {
                "type": "string"
              },
              "operand_2": {
                "type": "string"
              },
              "operator": {
                "type": "string"
              }
            }
          }
        },
        "x-amazon-apigateway-request-validator": "Validate query string parameters
 and headers",
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.operator": "integration.response.body.op",
                "method.response.header.operand_2": "integration.response.body.b",
                "method.response.header.operand_1": "integration.response.body.a"
              },
              "responseTemplates": {
                "application/json": "#set($res = $input.path('$'))\n{\n    \"result
\": \"$res.a, $res.b, $res.op => $res.c\",\n  \"a\" : \"$res.a\",\n  \"b\" :
 \"$res.b\",\n  \"op\" : \"$res.op\",\n  \"c\" : \"$res.c\"\n}"
```

```
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST",
          "requestTemplates": {
            "application/json": "{\n    \"a\":  \"$input.params('operand1')\",\n
 \"b\":  \"$input.params('operand2')\", \n    \"op\": \"$input.params('operator')\"
  \n}"
          },
          "type": "aws"
        }
      },
      "post": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "in": "body",
            "name": "Input",
            "required": true,
            "schema": {
              "$ref": "#/definitions/Input"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Result"
            }
          }
        },
        "x-amazon-apigateway-request-validator": "Validate body",
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
```

```
            "default": {
              "statusCode": "200",
              "responseTemplates": {
                "application/json": "#set($inputRoot = $input.path('$'))\n{\n  \"a
\" : $inputRoot.a,\n  \"b\" : $inputRoot.b,\n  \"op\" : $inputRoot.op,\n  \"c\" :
 $inputRoot.c\n}"
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "passthroughBehavior": "when_no_templates",
          "httpMethod": "POST",
          "type": "aws"
        }
      }
    },
    "/calc/{operand1}/{operand2}/{operator}": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "operand2",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "operator",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "operand1",
            "in": "path",
            "required": true,
            "type": "string"
          }
```

```
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Result"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseTemplates": {
                "application/json": "#set($inputRoot = $input.path('$'))\n{\n
 \"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
 \"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "passthroughBehavior": "when_no_templates",
          "httpMethod": "POST",
          "requestTemplates": {
            "application/json": "{\n   \"a\": \"$input.params('operand1')\",
\n   \"b\": \"$input.params('operand2')\",\n   \"op\":
 #if($input.params('operator')=='%2F')\"/\"#{else}\"$input.params('operator')\"#end
\n   \n}"
          },
          "contentHandling": "CONVERT_TO_TEXT",
          "type": "aws"
        }
      }
    }
  },
  "definitions": {
    "Input": {
      "type": "object",
      "required": [
        "a",
        "b",
        "op"
```

```
     ],
     "properties": {
       "a": {
         "type": "number"
       },
       "b": {
         "type": "number"
       },
       "op": {
         "type": "string",
         "description": "binary op of ['+', 'add', '-', 'sub', '*', 'mul', '%2F', 
'div']"
       }
     },
     "title": "Input"
   },
   "Output": {
     "type": "object",
     "properties": {
       "c": {
         "type": "number"
       }
     },
     "title": "Output"
   },
   "Result": {
     "type": "object",
     "properties": {
       "input": {
         "$ref": "#/definitions/Input"
       },
       "output": {
         "$ref": "#/definitions/Output"
       }
     },
     "title": "Result"
   }
 },
 "x-amazon-apigateway-request-validators": {
   "Validate body": {
     "validateRequestParameters": false,
     "validateRequestBody": true
   },
   "Validate query string parameters and headers": {
```

```
      "validateRequestParameters": true,
      "validateRequestBody": false
    }
  }
}
```

教程：在 API Gateway 中创建 REST API 作为 Amazon S3 代理

作为展示如何在 API Gateway 中使用 REST API 以代理 Amazon S3 的示例，本部分将介绍如何创建和配置 REST API 以公开以下 Amazon S3 操作：

- 在 API 的根资源上公开 GET 以列出调用方的所有 Amazon S3 存储桶。
- 在 Folder 资源上公开 GET 以查看 Amazon S3 存储桶中所有对象的列表。
- 在 Folder/Item 资源上公开 GET 以从 Amazon S3 存储桶查看或下载对象。

您可能需要导入示例 API 作为 Amazon S3 代理，如作为 Amazon S3 代理的示例 API 的 OpenAPI 定义中所示。此示例包含更多公开的方法。有关如何使用 OpenAPI 定义导入 API 的说明，请参阅 使用 OpenAPI 配置 REST API。

> **Note**
>
> 要将您的 API Gateway API 与 Amazon S3 集成，您必须选择同时提供 API Gateway 和 Amazon S3 服务的区域。有关区域可用性，请参阅 Amazon API Gateway 端点和配额。

主题

为 API 设置 IAM 权限以调用 Amazon S3 操作

要允许 API 调用 Amazon S3 操作，您必须已将适当的 IAM policy 附加到 IAM 角色。

创建 AWS 服务代理执行角色

1. 登录 AWS Management Console，然后使用以下网址打开 IAM 控制台：https://console.aws.amazon.com/iam/。
2. 选择角色。
3. 选择 Create role(创建角色)。
4. 在选择受信任实体的类型下选择 AWS 服务，然后选择 API Gateway 并选择允许 API Gateway 将日志推送到 CloudWatch Logs。
5. 选择下一步，然后再次选择下一步。
6. 对于角色名称，输入 **`APIGatewayS3ProxyPolicy`**，然后选择创建角色。
7. 在 Roles 列表中，选择您刚创建的角色。您可能需要滚动或使用搜索栏来查找角色。
8. 对于所选角色，选择添加权限选项卡。
9. 从下拉列表中选择附加策略。
10. 在搜索栏中，输入 **`AmazonS3FullAccess`** 然后选择添加权限。

    > Note
    >
    > 为简单起见，本教程使用托管策略。作为最佳实践，您应创建自己的 IAM 策略以授予所需的最低权限。

11. 记下新创建的角色 ARN，稍后将使用它。

创建 API 资源来代表 Amazon S3 资源

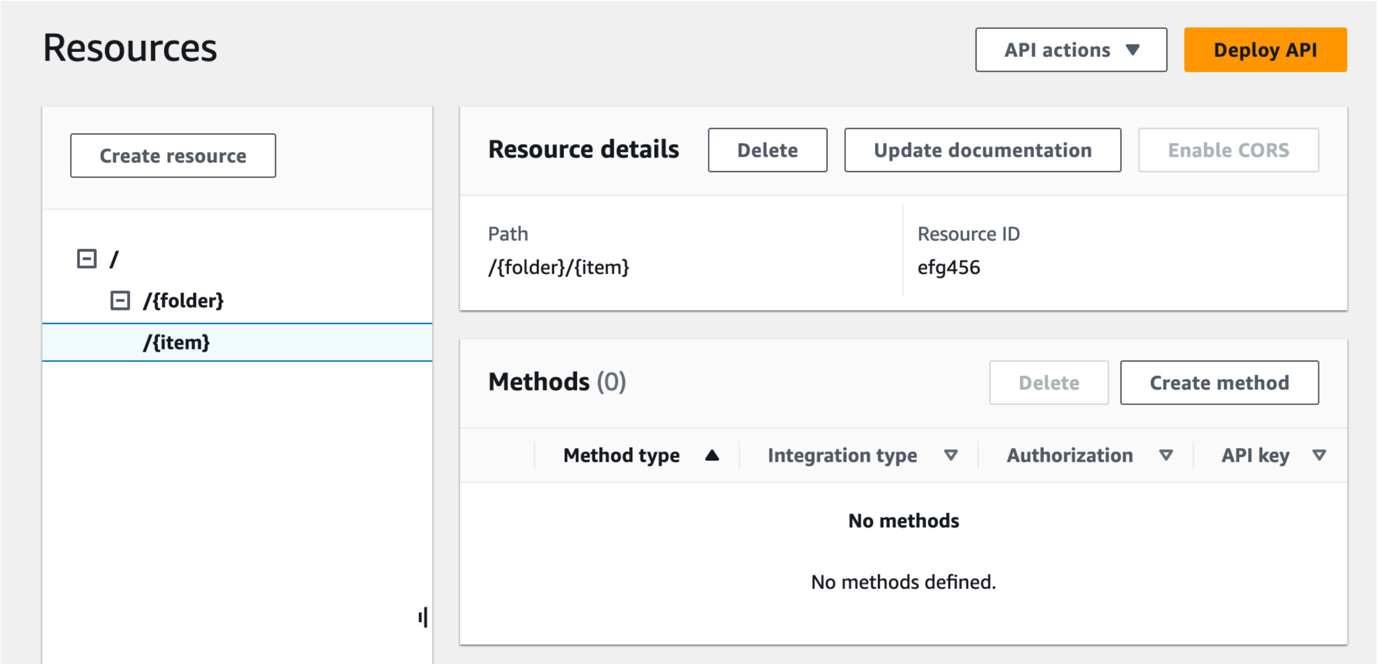
您使用 API 的根（/）资源作为经身份验证的调用方的 Amazon S3 存储桶的容器。您还将创建 `Folder` 和 `Item` 资源来分别代表特定的 Amazon S3 存储桶和 Amazon S3 对象。调用方将按照作为请求 URL 一部分的路径参数形式指定文件夹名称和对象键。

> Note
>
> 在访问其对象键包含 / 或任何其他特殊字符的对象时，字符需要进行 URL 编码。例如，`test/test.txt` 应编码为 `test%2Ftest.txt`。

创建公开 Amazon S3 服务特征的 API 资源

1. 在创建 Amazon S3 桶的同一 AWS 区域，创建名为 MyS3 的 API。此 API 的根资源 (/) 表示 Amazon S3 服务。在此步骤中，您将创建另外两个资源：/{folder} 和 /{item}。
2. 选择 API 的根资源，然后选择创建资源。
3. 将代理资源保持为关闭状态。
4. 对于资源路径，选择 `/`。
5. 对于资源名称，输入 **`{folder}`**。
6. 将 CORS（跨源资源共享）保持为未选中。
7. 选择创建资源。
8. 选择 /{folder} 资源，然后选择创建资源。
9. 使用上述步骤创建 /{folder} 的子资源，名为 **`{item}`**。

   最终的 API 应类似以下内容：



Path
/{folder}/{item}

Resource ID
efg456

No methods

No methods defined.

公开 API 方法以列出调用方的 Amazon S3 存储桶

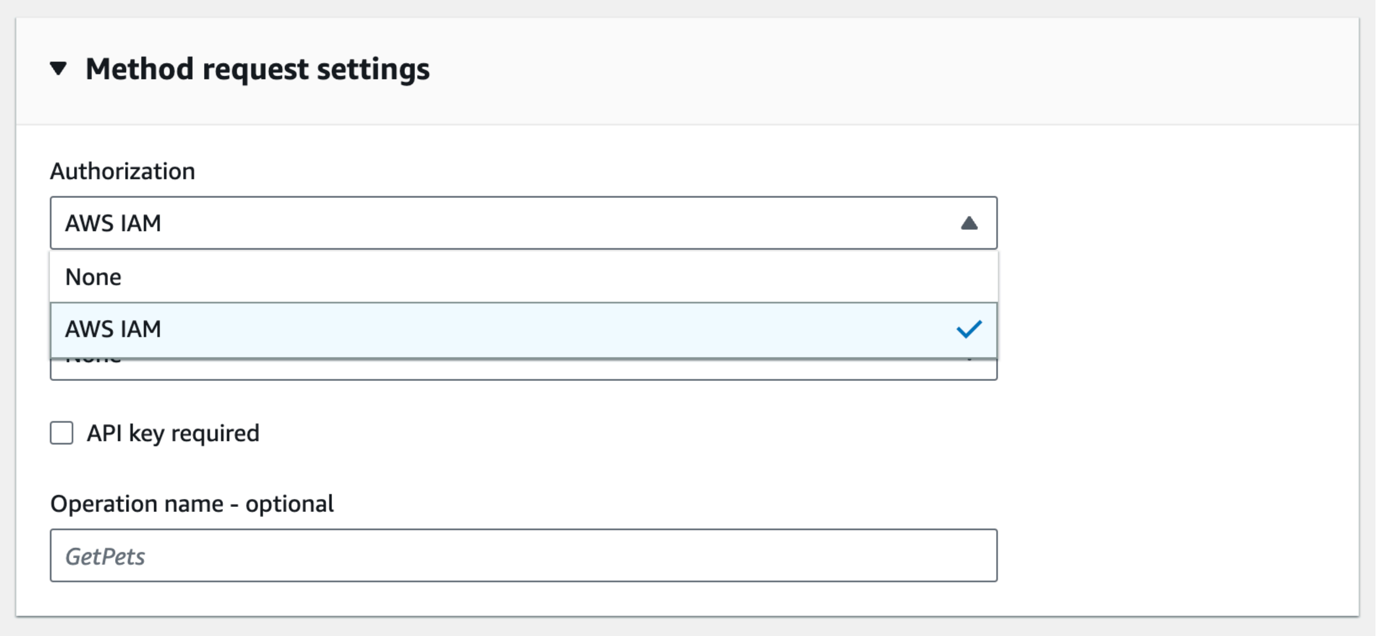
在获取调用方的 Amazon S3 存储桶列表的过程中，涉及针对 Amazon S3 调用 GET 服务操作。在 API 的根资源 (/) 上，创建 GET 方法。按如下所示，配置 GET 方法以与 Amazon S3 集成。

创建和初始化 API 的 **`GET /`** 方法

1. 选择 / 资源，然后选择创建方法。
2. 对于方法类型，选择 GET。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您创建 Amazon S3 桶的 AWS 区域。
5. 对于 AWS 服务，选择 Amazon Simple Storage Service。
6. 将 AWS 子域保留为空白。
7. 对于 HTTP 方法，选择 GET。
8. 对于操作类型，选择使用路径覆盖。

   利用路径覆盖，API Gateway 可将客户端请求作为对应的 Amazon S3 REST API 路径样式请求转发到 Amazon S3，其中 Amazon S3 资源用 `s3-host-name/bucket/key` 模式的资源路径表示。API Gateway 设置 `s3-host-name` 并将客户端指定的 `bucket` 和 `key` 从客户端传递到 Amazon S3。
9. 对于路径覆盖，输入 /。
10. 对于执行角色，输入 **`APIGatewayS3ProxyPolicy`** 的角色 ARN。
11. 选择方法请求设置。

    您可以使用方法请求设置来控制谁可以调用 API 的此方法。
12. 对于授权，从下拉菜单中选择 `AWS_IAM`。



Authorization: AWS IAM

☐ API key required

Operation name - optional: GetPets

13. 选择创建方法。

此设置会将前端 GET `https://your-api-host/stage/` 请求与后端 GET `https://your-s3-host/` 集成。

为使 API 能够正确地向调用方返回成功响应和异常，您可以在方法响应中声明 200、400 和 500 响应。您针对 200 响应使用默认映射，以便将未在此处声明的状态代码的后端响应作为 200 响应返回给调用方。

声明 **GET /** 方法的响应类型

1. 在方法响应选项卡的响应 200 下，选择编辑。
2. 选择添加标头，然后执行以下操作：
    a. 对于标头名称，输入 **Content-Type**。
    b. 选择添加标头。

   重复上述步骤以创建 **Timestamp** 标头和 **Content-Length** 标头。
3. 选择保存。
4. 在方法响应选项卡的方法响应下，选择创建响应。
5. 对于 HTTP 状态代码，输入 400。

   您不为此响应设置任何标头。
6. 选择保存。
7. 重复以下步骤以创建 500 响应。

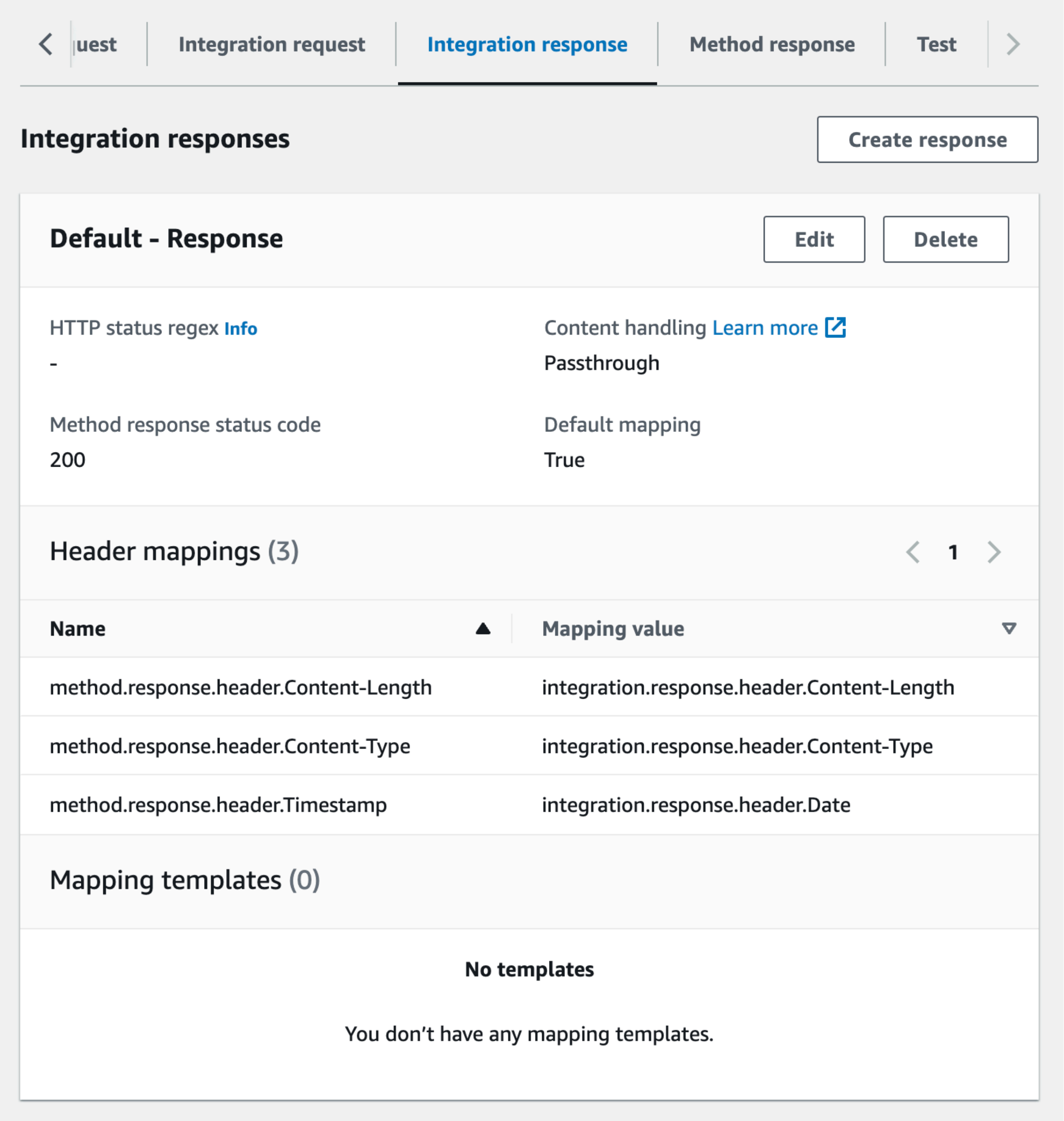
   您不为此响应设置任何标头。

因为来自 Amazon S3 的成功集成响应会返回存储桶列表作为 XML 负载，并且来自 API Gateway 的默认方法响应会返回 JSON 负载，所以您必须将后端 Content-Type 标头参数值映射到对应前端。或者，当响应正文实际上为 XML 字符串时，客户端将接收内容类型的 `application/json`。以下步骤将演示如何对其进行设置。此外，您还希望向客户端展示其它标头参数，如 Date 和 Content-Length。

设置用于 GET / 方法的响应标头映射

1. 在集成响应选项卡上的默认 - 响应下，选择编辑。

2. 对于 Content-Length 标头，输入 **`integration.response.header.Content-Length`** 作为映射值。
3. 对于 Content-Type 标头，输入 **`integration.response.header.Content-Type`** 作为映射值。
4. 对于 Timestamp 标头，输入 **`integration.response.header.Date`** 作为映射值。
5. 选择保存。结果应类似以下内容：



Integration responses

Default - Response

HTTP status regex: -

Content handling: Passthrough

Method response status code: 200

Default mapping: True

Header mappings (3)

Name	Mapping value
method.response.header.Content-Length	integration.response.header.Content-Length
method.response.header.Content-Type	integration.response.header.Content-Type
method.response.header.Timestamp	integration.response.header.Date

Mapping templates (0)

No templates

You don’t have any mapping templates.

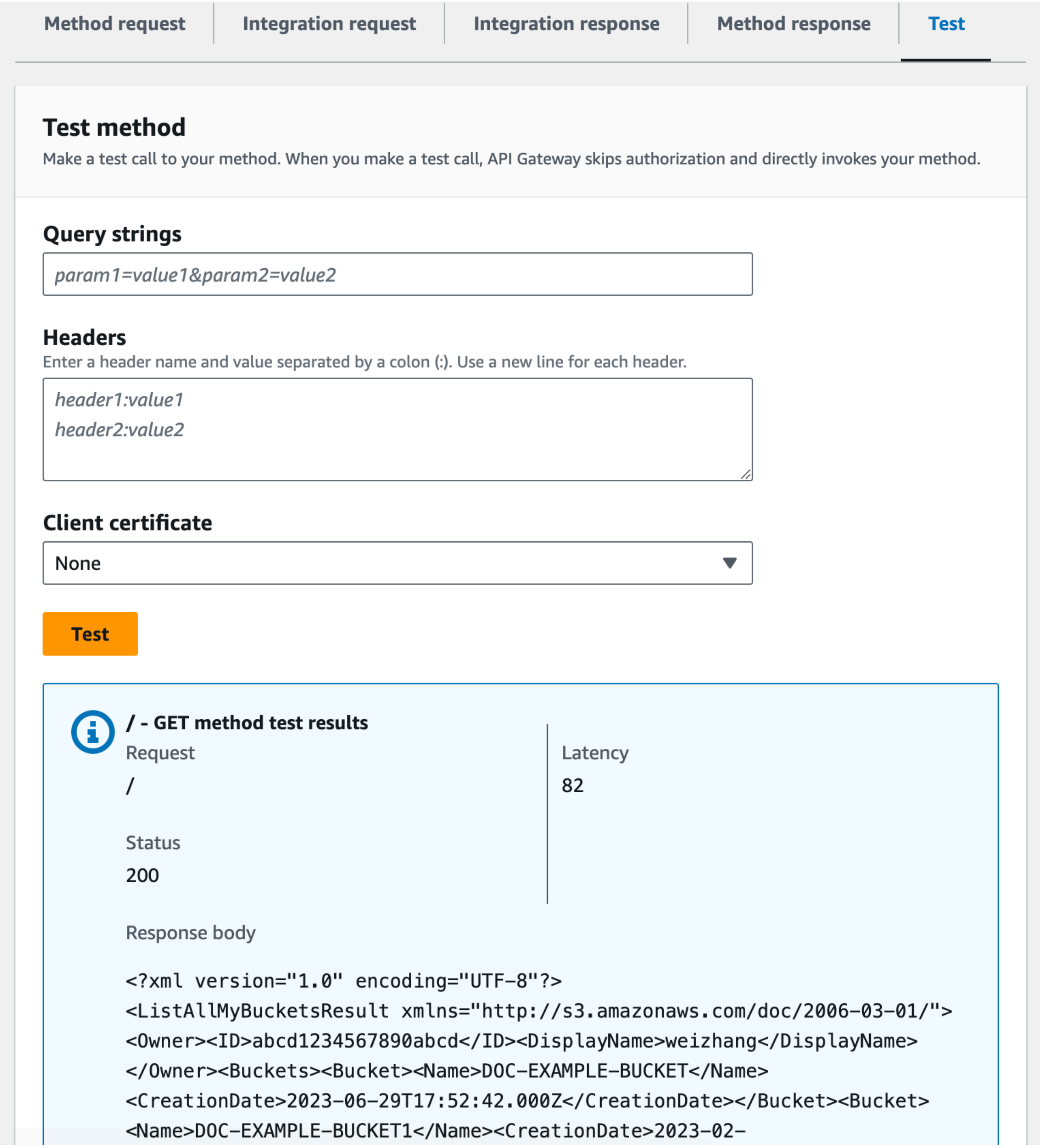
6. 在集成响应选项卡的集成响应下，选择创建响应。
7. 对于 HTTP 状态正则表达式，输入 **`4\d{2}`**。这会将所有 4xx HTTP 响应状态代码映射到方法响应。
8. 对于方法响应状态代码，选择 **`400`**。
9. 选择创建。

10. 重复以下步骤，为 500 方法响应创建集成响应。对于 HTTP 状态正则表达式，输入 **`5\d{2}`**。

作为一个良好做法，您可以测试到目前为止已配置的 API。

测试 **`GET /`** 方法

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
2. 选择测试。结果应类似下图内容：



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Query strings

param1=value1&param2=value2

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1
header2:value2

Client certificate

None

/ - GET method test results

Request
/

Status
200

Latency
82

Response body

```
<?xml version="1.0" encoding="UTF-8"?>
<ListAllMyBucketsResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
<Owner><ID>abcd1234567890abcd</ID><DisplayName>weizhang</DisplayName>
</Owner><Buckets><Bucket><Name>DOC-EXAMPLE-BUCKET</Name>
<CreationDate>2023-06-29T17:52:42.000Z</CreationDate></Bucket><Bucket>
<Name>DOC-EXAMPLE-BUCKET1</Name><CreationDate>2023-02-
```

公开 API 方法以访问 Amazon S3 存储桶

为了使用 Amazon S3 存储桶，您在 /{folder} 资源上公开 GET 方法，来列出桶中的对象。相关说明类似于公开 API 方法以列出调用方的 Amazon S3 存储桶中所述的说明。要了解更多方法，您可以转至作为 Amazon S3 代理的示例 API 的 OpenAPI 定义导入示例 API。

在文件夹资源上公开 GET 方法

1. 选择 /{folder} 资源，然后选择创建方法。
2. 对于方法类型，选择 GET。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您创建 Amazon S3 桶的 AWS 区域。
5. 对于 AWS 服务，选择 Amazon Simple Storage Service。
6. 将 AWS 子域保留为空白。
7. 对于 HTTP 方法，选择 GET。
8. 对于操作类型，选择使用路径覆盖。
9. 对于路径覆盖，输入 **`{bucket}`**。
10. 对于执行角色，输入 **`APIGatewayS3ProxyPolicy`** 的角色 ARN。
11. 选择创建方法。

在 Amazon S3 端点 URL 中设置 `{folder}` 路径参数。您需要将方法请求的 `{folder}` 路径参数映射到集成请求的 `{bucket}` 路径参数。

将 **`{folder}`** 映射到 **`{bucket}`**

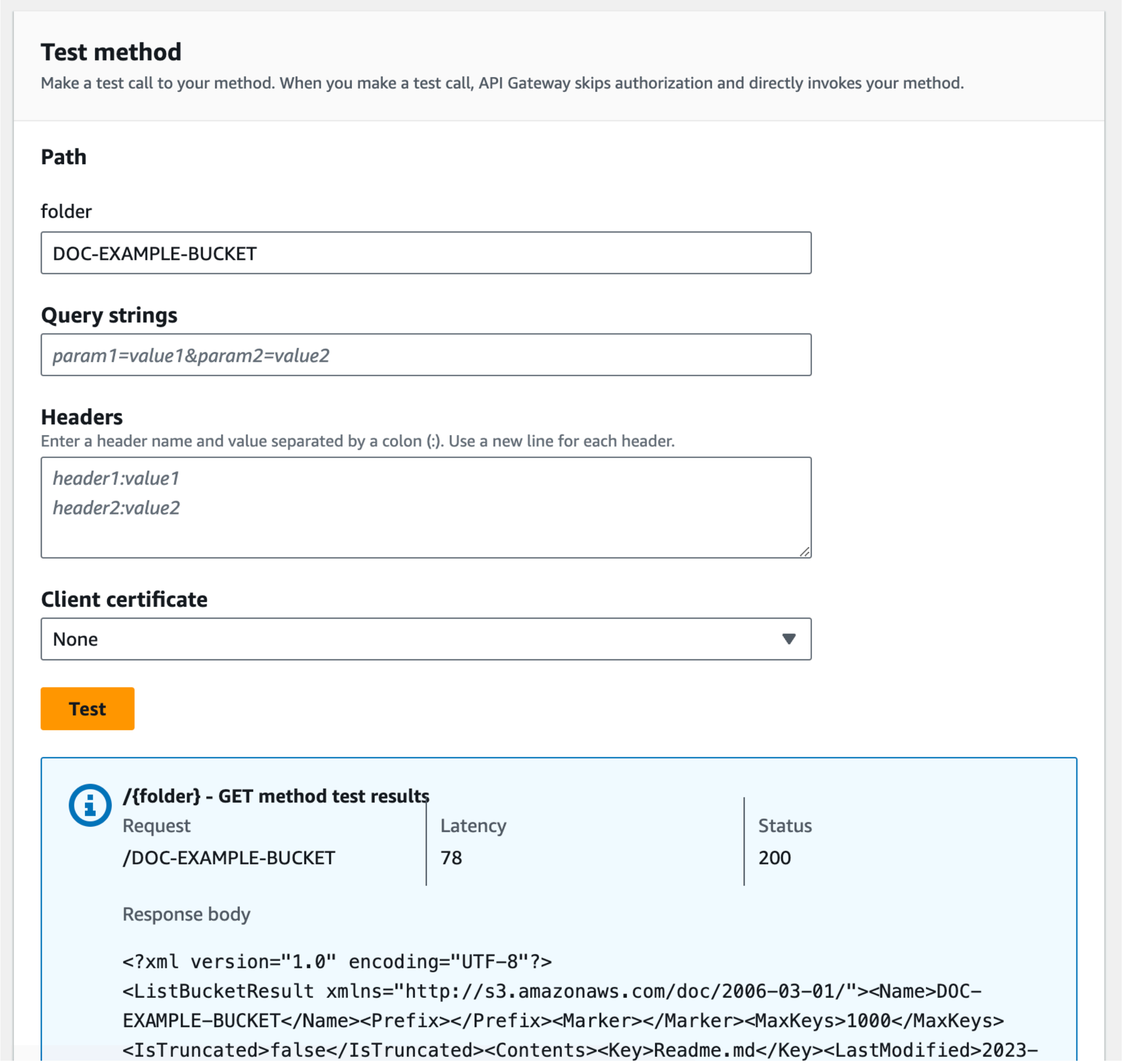
1. 在集成请求选项卡的集成请求设置下，选择编辑。
2. 选择 URL 路径参数，然后选择添加路径参数。
3. 对于名称，请输入 **`bucket`**。
4. 对于映射自，输入 **`method.request.path.folder`**。
5. 选择保存。

现在测试您的 API。

测试 **`/{folder}` `GET`** 方法。

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
2. 在路径下，对于文件夹，输入桶名称。
3. 选择测试。

   测试结果将包含桶中对象的列表。



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Path

folder: DOC-EXAMPLE-BUCKET

Query strings

param1=value1&param2=value2

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1
header2:value2

Client certificate

None

/{folder} - GET method test results

Request	Latency	Status
/DOC-EXAMPLE-BUCKET	78	200

Response body

```
<?xml version="1.0" encoding="UTF-8"?>
<ListBucketResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/"><Name>DOC-
EXAMPLE-BUCKET</Name><Prefix></Prefix><Marker></Marker><MaxKeys>1000</MaxKeys>
<IsTruncated>false</IsTruncated><Contents><Key>Readme.md</Key><LastModified>2023-
```

公开 API 方法以访问存储桶中的 Amazon S3 对象

Amazon S3 支持执行 GET、DELETE、HEAD、OPTIONS、POST 和 PUT 操作以访问和管理给定存储桶中的对象。在本教程中，您将在 `{folder}/{item}` 资源上公开一个 GET 方法，以从桶中获取图像。有关 `{folder}/{item}` 资源的更多应用，请转至作为 Amazon S3 代理的示例 API 的 OpenAPI 定义参阅示例 API。

对项目资源公开 GET 方法

1. 选择 /{item} 资源，然后选择创建方法。
2. 对于方法类型，选择 GET。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您创建 Amazon S3 桶的 AWS 区域。
5. 对于 AWS 服务，选择 Amazon Simple Storage Service。
6. 将 AWS 子域保留为空白。
7. 对于 HTTP 方法，选择 GET。
8. 对于操作类型，选择使用路径覆盖。
9. 对于路径覆盖，输入 {bucket}/{object}。
10. 对于执行角色，输入 **`APIGatewayS3ProxyPolicy`** 的角色 ARN。
11. 选择创建方法。

在 Amazon S3 端点 URL 中设置 `{folder}` 和 `{item}` 路径参数。您需要将方法请求的路径参数映射到集成请求的路径参数。

在此步骤中，您将执行以下操作：

- 将方法请求的 `{folder}` 路径参数映射到集成请求的 `{bucket}` 路径参数。
- 将方法请求的 `{item}` 路径参数映射到集成请求的 `{object}` 路径参数。

将 **`{folder}`** 映射到 **`{bucket}`**，并将 **`{item}`** 映射到 **`{object}`**

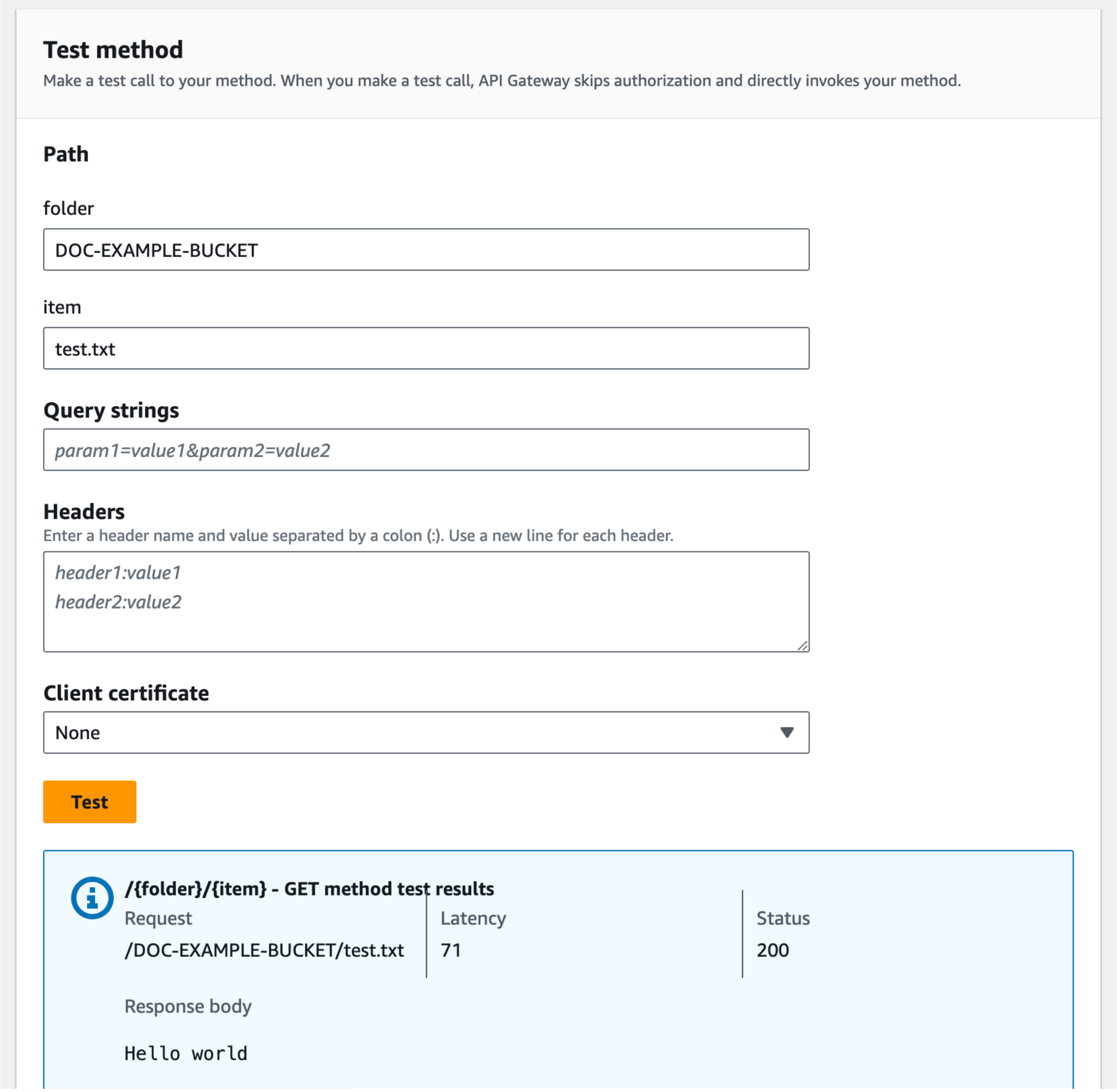
1. 在集成请求选项卡的集成请求设置下，选择编辑。
2. 选择 URL 路径参数。
3. 选择添加路径参数。
4. 对于名称，请输入 **`bucket`**。

5. 对于映射自，输入 **`method.request.path.folder`**。
6. 选择添加路径参数。
7. 对于名称，请输入 **`object`**。
8. 对于映射自，输入 **`method.request.path.item`**。
9. 选择保存。

测试 **`/{folder}/{object} GET`** 方法。

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
2. 在路径下，对于文件夹，输入桶名称。
3. 在路径下，对于项目，输入项目名称。
4. 选择测试。

   响应正文将包含该项目的内容。



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Path

folder

DOC-EXAMPLE-BUCKET

item

test.txt

Query strings

param1=value1&param2=value2

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:value1
header2:value2

Client certificate

None

/{folder}/{item} - GET method test results

Request	Latency	Status
/DOC-EXAMPLE-BUCKET/test.txt	71	200

Response body

```
Hello world
```

该请求正确返回纯文本（“Hello world”）作为给定 Amazon S3 桶 (DOC-EXAMPLE-BUCKET) 中指定文件 (test.txt) 的内容。

要下载或上传二进制文件（在 API Gateway 中被视为 utf-8 编码的 JSON 内容之外的任何项），需要额外的 API 设置。概述如下所示：

从 S3 下载或上传二进制文件

1. 将受影响文件的介质类型注册到 API 的 binaryMediaTypes。您可以在控制台中执行此操作：
    a. 选择 API 的 API 设置。
    b. 在二进制媒体类型下，选择管理媒体类型。
    c. 选择添加二进制媒体类型，然后输入所需的媒体类型，例如 `image/png`。
    d. 选择保存更改以保存设置。
2. 将 `Content-Type`（用于上传）和/或 `Accept`（用于下载）标头添加到方法请求，以要求客户端指定所需的二进制介质类型并将其映射到集成请求。
3. 在集成请求（用于上传）和集成响应（用于下载）中，将内容处理 设置为 `Passthrough`。确保未为受影响的内容类型定义任何映射模板。有关更多信息，请参阅集成传递行为和选择 VTL 映射模板。

负载大小限制为 10 MB。请参阅 用于配置和运行 REST API 的 API Gateway 配额。

确保 Amazon S3 上的文件具有作为文件的元数据添加的正确内容类型。对于可流式传输的介质内容，可能还需将 `Content-Disposition:inline` 添加到元数据。

有关 API Gateway 中的二进制文件支持的更多信息，请参阅API Gateway 中的内容类型转换。

作为 Amazon S3 代理的示例 API 的 OpenAPI 定义

以下 OpenAPI 定义描述可用作 Amazon S3 代理的 API。与您在本教程中创建的 API 相比，此 API 包含更多 Amazon S3 操作。此 OpenAPI 定义中公开了以下方法：

- 在 API 的根资源上公开 GET 以列出调用方的所有 Amazon S3 存储桶。
- 在 Folder 资源上公开 GET 以查看 Amazon S3 存储桶中所有对象的列表。
- 在 Folder 资源上公开 PUT 以将存储桶添加到 Amazon S3。
- 在 Folder 资源上公开 DELETE 以从 Amazon S3 中删除存储桶。
- 在 Folder/Item 资源上公开 GET 以从 Amazon S3 存储桶查看或下载对象。
- 在 Folder/Item 资源上公开 PUT 以将对象上传到 Amazon S3 存储桶。
- 在 Folder/Item 资源上公开 HEAD 以在 Amazon S3 存储桶中获取对象元数据。
- 在 Folder/Item 资源上公开 DELETE 以从 Amazon S3 存储桶中删除对象。

有关如何使用 OpenAPI 定义导入 API 的说明，请参阅 使用 OpenAPI 配置 REST API。

有关如何创建类似 API 的说明，请参阅教程：在 API Gateway 中创建 REST API 作为 Amazon S3 代理。

要了解如何使用支持 AWS IAM 授权的 Postman 调用此 API，请参阅使用 REST API 客户端调用 API。

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-10-13T23:04:43Z",
    "title": "MyS3"
  },
  "host": "9gn28ca086.execute-api.{region}.amazonaws.com",
  "basePath": "/S3",
  "schemes": [
    "https"
  ],
  "paths": {
    "/": {
      "get": {
        "produces": [
          "application/json"
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Length": {
                "type": "string"
              },
              "Timestamp": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
```

```
         "400": {
           "description": "400 response"
         },
         "500": {
           "description": "500 response"
         }
       },
       "security": [
         {
           "sigv4": []
         }
       ],
       "x-amazon-apigateway-integration": {
         "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
         "responses": {
           "4\\d{2}": {
             "statusCode": "400"
           },
           "default": {
             "statusCode": "200",
             "responseParameters": {
               "method.response.header.Content-Type":
"integration.response.header.Content-Type",
               "method.response.header.Content-Length":
"integration.response.header.Content-Length",
               "method.response.header.Timestamp":
"integration.response.header.Date"
             }
           },
           "5\\d{2}": {
             "statusCode": "500"
           }
         },
         "uri": "arn:aws:apigateway:us-west-2:s3:path//",
         "passthroughBehavior": "when_no_match",
         "httpMethod": "GET",
         "type": "aws"
       }
     }
   },
   "/{folder}": {
     "get": {
       "produces": [
         "application/json"
```

```
        ],
        "parameters": [
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Length": {
                "type": "string"
              },
              "Date": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
        },
        "security": [
          {
            "sigv4": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
```

```
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type",
                "method.response.header.Date": "integration.response.header.Date",
                "method.response.header.Content-Length":
 "integration.response.header.content-length"
              }
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
          "requestParameters": {
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "GET",
          "type": "aws"
        }
      },
      "put": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "Content-Type",
            "in": "header",
            "required": false,
            "type": "string"
          },
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
```

```
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Length": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
        },
        "security": [
          {
            "sigv4": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type",
                "method.response.header.Content-Length":
 "integration.response.header.Content-Length"
              }
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
```

```
          "requestParameters": {
            "integration.request.path.bucket": "method.request.path.folder",
            "integration.request.header.Content-Type":
  "method.request.header.Content-Type"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "PUT",
          "type": "aws"
        }
      },
      "delete": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Date": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
```

```
        },
        "security": [
          {
            "sigv4": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type":
  "integration.response.header.Content-Type",
                "method.response.header.Date": "integration.response.header.Date"
              }
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
          "requestParameters": {
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "DELETE",
          "type": "aws"
        }
      }
    },
    "/{folder}/{item}": {
      "get": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "item",
            "in": "path",
            "required": true,
```

```
            "type": "string"
          },
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "content-type": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
        },
        "security": [
          {
            "sigv4": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200",
```

```
              "responseParameters": {
                "method.response.header.content-type":
 "integration.response.header.content-type",
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type"
              }
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
          "requestParameters": {
            "integration.request.path.object": "method.request.path.item",
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "GET",
          "type": "aws"
        }
      },
      "head": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "item",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
```

```
            },
            "headers": {
              "Content-Length": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
        },
        "security": [
          {
            "sigv4": []
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type":
"integration.response.header.Content-Type",
                "method.response.header.Content-Length":
"integration.response.header.Content-Length"
              }
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
          "requestParameters": {
            "integration.request.path.object": "method.request.path.item",
            "integration.request.path.bucket": "method.request.path.folder"
```

```
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "HEAD",
          "type": "aws"
        }
      },
      "put": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "Content-Type",
            "in": "header",
            "required": false,
            "type": "string"
          },
          {
            "name": "item",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Length": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
```

```
           }
         },
         "400": {
           "description": "400 response"
         },
         "500": {
           "description": "500 response"
         }
       },
       "security": [
         {
           "sigv4": []
         }
       ],
       "x-amazon-apigateway-integration": {
         "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
         "responses": {
           "4\\d{2}": {
             "statusCode": "400"
           },
           "default": {
             "statusCode": "200",
             "responseParameters": {
               "method.response.header.Content-Type":
"integration.response.header.Content-Type",
               "method.response.header.Content-Length":
"integration.response.header.Content-Length"
             }
           },
           "5\\d{2}": {
             "statusCode": "500"
           }
         },
         "requestParameters": {
           "integration.request.path.object": "method.request.path.item",
           "integration.request.path.bucket": "method.request.path.folder",
           "integration.request.header.Content-Type":
"method.request.header.Content-Type"
         },
         "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
         "passthroughBehavior": "when_no_match",
         "httpMethod": "PUT",
         "type": "aws"
       }
```

```
      },
      "delete": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "item",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "folder",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Length": {
                "type": "string"
              },
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response"
          },
          "500": {
            "description": "500 response"
          }
        },
        "security": [
          {
            "sigv4": []
```

```
          }
        ],
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400"
            },
            "default": {
              "statusCode": "200"
            },
            "5\\d{2}": {
              "statusCode": "500"
            }
          },
          "requestParameters": {
            "integration.request.path.object": "method.request.path.item",
            "integration.request.path.bucket": "method.request.path.folder"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "DELETE",
          "type": "aws"
        }
      }
    }
  },
  "securityDefinitions": {
    "sigv4": {
      "type": "apiKey",
      "name": "Authorization",
      "in": "header",
      "x-amazon-apigateway-authtype": "awsSigv4"
    }
  },
  "definitions": {
    "Empty": {
      "type": "object",
      "title": "Empty Schema"
    }
  }
}
```

OpenAPI 3.0

```
{
  "openapi" : "3.0.1",
  "info" : {
    "title" : "MyS3",
    "version" : "2016-10-13T23:04:43Z"
  },
  "servers" : [ {
    "url" : "https://9gn28ca086.execute-api.{region}.amazonaws.com/{basePath}",
    "variables" : {
      "basePath" : {
        "default" : "S3"
      }
    }
  } ],
  "paths" : {
    "/{folder}" : {
      "get" : {
        "parameters" : [ {
          "name" : "folder",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Content-Length" : {
                "schema" : {
                  "type" : "string"
                }
              },
```

```
             "Date" : {
               "schema" : {
                 "type" : "string"
               }
             },
             "Content-Type" : {
               "schema" : {
                 "type" : "string"
               }
             }
           },
           "content" : {
             "application/json" : {
               "schema" : {
                 "$ref" : "#/components/schemas/Empty"
               }
             }
           }
         }
       },
       "x-amazon-apigateway-integration" : {
         "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
         "httpMethod" : "GET",
         "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
         "responses" : {
           "4\\d{2}" : {
             "statusCode" : "400"
           },
           "default" : {
             "statusCode" : "200",
             "responseParameters" : {
               "method.response.header.Content-Type" :
"integration.response.header.Content-Type",
               "method.response.header.Date" : "integration.response.header.Date",
               "method.response.header.Content-Length" :
"integration.response.header.content-length"
             }
           },
           "5\\d{2}" : {
             "statusCode" : "500"
           }
         },
         "requestParameters" : {
           "integration.request.path.bucket" : "method.request.path.folder"
```

```
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
      },
      "put" : {
        "parameters" : [ {
          "name" : "Content-Type",
          "in" : "header",
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "folder",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Content-Length" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Content-Type" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
```

```
             "application/json" : {
               "schema" : {
                 "$ref" : "#/components/schemas/Empty"
               }
             }
           }
         }
       },
       "x-amazon-apigateway-integration" : {
         "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
         "httpMethod" : "PUT",
         "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
         "responses" : {
           "4\\d{2}" : {
             "statusCode" : "400"
           },
           "default" : {
             "statusCode" : "200",
             "responseParameters" : {
               "method.response.header.Content-Type" :
"integration.response.header.Content-Type",
               "method.response.header.Content-Length" :
"integration.response.header.Content-Length"
             }
           },
           "5\\d{2}" : {
             "statusCode" : "500"
           }
         },
         "requestParameters" : {
           "integration.request.path.bucket" : "method.request.path.folder",
           "integration.request.header.Content-Type" :
"method.request.header.Content-Type"
         },
         "passthroughBehavior" : "when_no_match",
         "type" : "aws"
       }
     },
     "delete" : {
       "parameters" : [ {
         "name" : "folder",
         "in" : "path",
         "required" : true,
         "schema" : {
```

```
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Date" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Content-Type" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "httpMethod" : "DELETE",
          "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}",
          "responses" : {
            "4\\d{2}" : {
              "statusCode" : "400"
            },
            "default" : {
```

```
              "statusCode" : "200",
              "responseParameters" : {
                "method.response.header.Content-Type" :
"integration.response.header.Content-Type",
                "method.response.header.Date" : "integration.response.header.Date"
              }
            },
            "5\\d{2}" : {
              "statusCode" : "500"
            }
          },
          "requestParameters" : {
            "integration.request.path.bucket" : "method.request.path.folder"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
      }
    },
    "/{folder}/{item}" : {
      "get" : {
        "parameters" : [ {
          "name" : "item",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "folder",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
```

```
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "content-type" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Content-Type" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "httpMethod" : "GET",
          "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "responses" : {
            "4\\d{2}" : {
              "statusCode" : "400"
            },
            "default" : {
              "statusCode" : "200",
              "responseParameters" : {
                "method.response.header.content-type" :
"integration.response.header.content-type",
                "method.response.header.Content-Type" :
"integration.response.header.Content-Type"
              }
            },
            "5\\d{2}" : {
              "statusCode" : "500"
            }
```

```
          },
          "requestParameters" : {
            "integration.request.path.object" : "method.request.path.item",
            "integration.request.path.bucket" : "method.request.path.folder"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
      },
      "put" : {
        "parameters" : [ {
          "name" : "Content-Type",
          "in" : "header",
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "item",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "folder",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Content-Length" : {
```

```
              "schema" : {
                "type" : "string"
              }
            },
            "Content-Type" : {
              "schema" : {
                "type" : "string"
              }
            }
          },
          "content" : {
            "application/json" : {
              "schema" : {
                "$ref" : "#/components/schemas/Empty"
              }
            }
          }
        }
      },
      "x-amazon-apigateway-integration" : {
        "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
        "httpMethod" : "PUT",
        "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
        "responses" : {
          "4\\d{2}" : {
            "statusCode" : "400"
          },
          "default" : {
            "statusCode" : "200",
            "responseParameters" : {
              "method.response.header.Content-Type" :
"integration.response.header.Content-Type",
              "method.response.header.Content-Length" :
"integration.response.header.Content-Length"
            }
          },
          "5\\d{2}" : {
            "statusCode" : "500"
          }
        },
        "requestParameters" : {
          "integration.request.path.object" : "method.request.path.item",
          "integration.request.path.bucket" : "method.request.path.folder",
```

```
            "integration.request.header.Content-Type" :
 "method.request.header.Content-Type"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
      },
      "delete" : {
        "parameters" : [ {
          "name" : "item",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "folder",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Content-Length" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Content-Type" : {
                "schema" : {
                  "type" : "string"
                }
```

```
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "httpMethod" : "DELETE",
          "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "responses" : {
            "4\\d{2}" : {
              "statusCode" : "400"
            },
            "default" : {
              "statusCode" : "200"
            },
            "5\\d{2}" : {
              "statusCode" : "500"
            }
          },
          "requestParameters" : {
            "integration.request.path.object" : "method.request.path.item",
            "integration.request.path.bucket" : "method.request.path.folder"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
      },
      "head" : {
        "parameters" : [ {
          "name" : "item",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "folder",
```

```
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Content-Length" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Content-Type" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "httpMethod" : "HEAD",
          "uri" : "arn:aws:apigateway:us-west-2:s3:path/{bucket}/{object}",
          "responses" : {
            "4\\d{2}" : {
```

```
              "statusCode" : "400"
            },
            "default" : {
              "statusCode" : "200",
              "responseParameters" : {
                "method.response.header.Content-Type" :
 "integration.response.header.Content-Type",
                "method.response.header.Content-Length" :
 "integration.response.header.Content-Length"
              }
            },
            "5\\d{2}" : {
              "statusCode" : "500"
            }
          },
          "requestParameters" : {
            "integration.request.path.object" : "method.request.path.item",
            "integration.request.path.bucket" : "method.request.path.folder"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "aws"
        }
      }
    },
    "/" : {
      "get" : {
        "responses" : {
          "400" : {
            "description" : "400 response",
            "content" : { }
          },
          "500" : {
            "description" : "500 response",
            "content" : { }
          },
          "200" : {
            "description" : "200 response",
            "headers" : {
              "Content-Length" : {
                "schema" : {
                  "type" : "string"
                }
              },
              "Timestamp" : {
```

```
               "schema" : {
                 "type" : "string"
               }
             },
             "Content-Type" : {
               "schema" : {
                 "type" : "string"
               }
             }
           },
           "content" : {
             "application/json" : {
               "schema" : {
                 "$ref" : "#/components/schemas/Empty"
               }
             }
           }
         }
       },
       "x-amazon-apigateway-integration" : {
         "credentials" : "arn:aws:iam::123456789012:role/apigAwsProxyRole",
         "httpMethod" : "GET",
         "uri" : "arn:aws:apigateway:us-west-2:s3:path//",
         "responses" : {
           "4\\d{2}" : {
             "statusCode" : "400"
           },
           "default" : {
             "statusCode" : "200",
             "responseParameters" : {
               "method.response.header.Content-Type" :
"integration.response.header.Content-Type",
               "method.response.header.Content-Length" :
"integration.response.header.Content-Length",
               "method.response.header.Timestamp" :
"integration.response.header.Date"
             }
           },
           "5\\d{2}" : {
             "statusCode" : "500"
           }
         },
         "passthroughBehavior" : "when_no_match",
         "type" : "aws"
```

```
        }
      }
    }
  },
  "components" : {
    "schemas" : {
      "Empty" : {
        "title" : "Empty Schema",
        "type" : "object"
      }
    }
  }
}
```

使用 REST API 客户端调用 API

为了提供端到端教程，我们现在演示如何使用支持AWS IAM 授权的 Postman 调用 API。

使用 Postman 调用我们的 Amazon S3 代理 API

1. 部署或重新部署 API。记下位于阶段编辑器顶部的调用 URL 旁边显示的 API 的基本 URL。
2. 启动 Postman。
3. 选择授权，然后选择 `AWS Signature`。分别在 AccessKey 和 SecretKey 输入字段中输入您的 IAM 用户的访问密钥 ID 和秘密访问密钥。在 AWS 区域文本框中输入您的 API 部署到的 AWS 区域。在服务名称输入字段中键入 `execute-api`。

   您可以在 IAM 管理控制台的 IAM 用户账户的安全凭证选项卡中创建一对密钥。
4. 要在 `apig-demo-5` 区域内将名为 *{region}* 的存储桶添加到您的 Amazon S3 账户，请执行以下操作：

   > Note
   >
   > 请确保保存储桶名称具有全局唯一性。

   a. 从下拉方法列表中选择 PUT 并键入方法 URL (`https://`*api-id*`.execute-api.`*aws-region*`.amazonaws.com/`*stage*`/`*folder-name*)
   b. 将 `Content-Type` 标头值设置为 `application/xml`。在设置内容类型之前，您可能需要先删除任何现有标头。

c. 选择正文菜单项并键入以下 XML 片段作为请求正文：

```
<CreateBucketConfiguration>
  <LocationConstraint>{region}</LocationConstraint>
</CreateBucketConfiguration>
```

d. 选择发送以提交请求。如果成功，您应该会收到一个负载为空的 `200 OK` 响应。

5. 要将文本文件添加到存储桶，请按照上述说明执行操作。如果您在 URL 中为 **`apig-demo-5`** 指定存储桶名称 `{folder}`，为 **`Readme.txt`** 指定文件名 `{item}`，并提供文本字符串 **`Hello, World!`** 作为文件内容（从而使其成为请求负载），则此请求将成为

```
PUT /S3/apig-demo-5/Readme.txt HTTP/1.1
Host: 9gn28ca086.execute-api.{region}.amazonaws.com
Content-Type: application/xml
X-Amz-Date: 20161015T062647Z
Authorization: AWS4-HMAC-SHA256 Credential=access-key-id/20161015/{region}/execute-
api/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=ccadb877bdb0d395ca38cc47e18a0d76bb5eaf17007d11e40bf6fb63d28c705b
Cache-Control: no-cache
Postman-Token: 6135d315-9cc4-8af8-1757-90871d00847e

Hello, World!
```

如果一切正常，您应该会收到一个负载为空的 `200 OK` 响应。

6. 要获取我们刚刚添加到 `Readme.txt` 存储桶的 `apig-demo-5` 文件的内容，请发出类似于以下的 GET 请求：

```
GET /S3/apig-demo-5/Readme.txt HTTP/1.1
Host: 9gn28ca086.execute-api.{region}.amazonaws.com
Content-Type: application/xml
X-Amz-Date: 20161015T063759Z
Authorization: AWS4-HMAC-SHA256 Credential=access-key-id/20161015/{region}/
execute-api/aws4_request, SignedHeaders=content-type;host;x-amz-date,
 Signature=ba09b72b585acf0e578e6ad02555c00e24b420b59025bc7bb8d3f7aed1471339
Cache-Control: no-cache
Postman-Token: d60fcb59-d335-52f7-0025-5bd96928098a
```

如果成功，您应该会收到负载为 `200 OK` 文本字符串的 `Hello, World!` 响应。

7. 要列出 `apig-demo-5` 存储桶中的项目，请提交以下请求：

```
GET /S3/apig-demo-5 HTTP/1.1
Host: 9gn28ca086.execute-api.{region}.amazonaws.com
Content-Type: application/xml
X-Amz-Date: 20161015T064324Z
Authorization: AWS4-HMAC-SHA256 Credential=access-key-id/20161015/{region}/
execute-api/aws4_request, SignedHeaders=content-type;host;x-amz-date,
 Signature=4ac9bd4574a14e01568134fd16814534d9951649d3a22b3b0db9f1f5cd4dd0ac
Cache-Control: no-cache
Postman-Token: 9c43020a-966f-61e1-81af-4c49ad8d1392
```

如果成功，您应该会收到 200 OK 响应且其 XML 负载在指定存储桶中显示单个项目，除非您在提交请求前将更多文件添加到存储桶中。

```
<?xml version="1.0" encoding="UTF-8"?>
<ListBucketResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <Name>apig-demo-5</Name>
    <Prefix></Prefix>
    <Marker></Marker>
    <MaxKeys>1000</MaxKeys>
    <IsTruncated>false</IsTruncated>
    <Contents>
        <Key>Readme.txt</Key>
        <LastModified>2016-10-15T06:26:48.000Z</LastModified>
        <ETag>"65a8e27d8879283831b664bd8b7f0ad4"</ETag>
        <Size>13</Size>
        <Owner>
            <ID>06e4b09e9d...603addd12ee</ID>
            <DisplayName>user-name</DisplayName>
        </Owner>
        <StorageClass>STANDARD</StorageClass>
    </Contents>
</ListBucketResult>
```

Note

要上传或下载映像，您需要将内容处理设置为 CONVERT_TO_BINARY。

教程：在 API Gateway 中创建 REST API 作为 Amazon Kinesis 代理

此页面介绍如何利用 AWS 类型的集成创建和配置 REST API 以访问 Kinesis。

> ⓘ Note
>
> 要将 API Gateway API 与 Kinesis 集成，您必须选择同时提供 API Gateway 和 Kinesis 服务的区域。有关区域可用性，请参阅服务端点和配额。

为了进行说明，我们创建一个示例 API，以使客户端能够执行以下操作：

1. 列出 Kinesis 中用户的可用流
2. 创建、描述或删除指定流
3. 从指定流读取数据记录或将数据记录写入指定流

为了完成上述任务，API 分别在各种资源上使用了多种方法来调用以下内容：

1. Kinesis 中的 `ListStreams` 操作
2. `CreateStream`、`DescribeStream` 或 `DeleteStream` 操作
3. Kinesis 中的 `GetRecords` 或 `PutRecords`（包括 `PutRecord`）操作

具体来说，我们按如下所示构建 API：

- 在 API 的 `/streams` 资源上公开 HTTP GET 方法，并将此方法与 Kinesis 中的 ListStreams 操作集成，以列出调用方账户中的流。
- 在 API 的 `/streams/{stream-name}` 资源上公开 HTTP POST 方法，并将此方法与 Kinesis 中的 CreateStream 操作集成，以在调用方账户中创建指定流。
- 在 API 的 `/streams/{stream-name}` 资源上公开 HTTP GET 方法，并将此方法与 Kinesis 中的 DescribeStream 操作集成，以描述调用方账户中的指定流。
- 在 API 的 `/streams/{stream-name}` 资源上公开 HTTP DELETE 方法，并将此方法与 Kinesis 中的 DeleteStream 操作集成，以删除调用方账户中的流。
- 在 API 的 `/streams/{stream-name}/record` 资源上公开 HTTP PUT 方法，并将此方法与 Kinesis 中的 PutRecord 操作集成。这使客户端能够向指定流添加一个数据记录。
- 在 API 的 `/streams/{stream-name}/records` 资源上公开 HTTP PUT 方法，并将此方法与 Kinesis 中的 PutRecords 操作集成。这使客户端能够向指定流添加一个数据记录列表。

- 在 API 的 `/streams/{stream-name}/records` 资源上公开 HTTP GET 方法，并将此方法与 Kinesis 中的 GetRecords 操作集成。这使客户端能够使用指定的分片迭代器在指定流中列出数据记录。分片迭代器指定了分片位置，可以从该位置开始按顺序读取数据记录。
- 在 API 的 `/streams/{stream-name}/sharditerator` 资源上公开 HTTP GET 方法，并将此方法与 Kinesis 中的 GetShardIterator 操作集成。此辅助标记方法必须提供给 Kinesis 中的 `ListStreams` 操作。

您可以将此处提供的说明应用于其他 Kinesis 操作。有关 Kinesis 操作的完整列表，请参阅 Amazon Kinesis API 参考。

您可以使用 API Gateway 导入 API 将示例 API 导入到 API Gateway 中，而不是使用 API Gateway 控制台创建示例 API。有关如何使用 Import API 的信息，请参阅 使用 OpenAPI 配置 REST API。

为 API 创建 IAM 角色和策略以访问 Kinesis

要允许 API 调用 Kinesis 操作，您必须将适当的 IAM policy 附加到 IAM 角色。

创建 AWS 服务代理执行角色

1. 登录 AWS Management Console，然后使用以下网址打开 IAM 控制台：https://console.aws.amazon.com/iam/。
2. 选择角色。
3. 选择 Create role(创建角色)。
4. 在选择受信任实体的类型下选择 AWS 服务，然后选择 API Gateway 并选择允许 API Gateway 将日志推送到 CloudWatch Logs。
5. 选择下一步，然后再次选择下一步。
6. 对于角色名称，输入 **`APIGatewayKinesisProxyPolicy`**，然后选择创建角色。
7. 在 Roles 列表中，选择您刚创建的角色。您可能需要滚动或使用搜索栏来查找角色。
8. 对于所选角色，选择添加权限选项卡。
9. 从下拉列表中选择附加策略。
10. 在搜索栏中，输入 **`AmazonKinesisFullAccess`** 然后选择添加权限。

> Note
>
> 为简单起见，本教程使用托管策略。作为最佳实践，您应创建自己的 IAM 策略以授予所需的最低权限。

11. 记下新创建的角色 ARN，稍后将使用它。

创建 API 作为 Kinesis 代理

使用以下步骤在 API Gateway 控制台中创建 API。

创建 API 作为 Kinesis 的AWS服务代理

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 如果您是第一次使用 API Gateway，您会看到一个介绍服务特征的页面。在 REST API 下，选择生成。当创建示例 API 弹出框出现时，选择确定。

   如果这不是您首次使用 API Gateway，请选择创建 API。在 REST API 下，选择生成。
3. 选择新 API。
4. 在 API 名称中，输入 **`KinesisProxy`**。对于所有其他字段，保留默认值。
5. （可选）对于描述，输入描述。
6. 选择 Create API (创建 API)。

API 创建完成后，API Gateway 控制台将显示资源页面，该页面仅包含 API 的根 (/) 资源。

列出 Kinesis 中的流

Kinesis 支持使用以下 REST API 调用执行 `ListStreams` 操作：

```
POST /?Action=ListStreams HTTP/1.1
Host: kinesis.<region>.<domain>
Content-Length: <PayloadSizeBytes>
User-Agent: <UserAgentString>
Content-Type: application/x-amz-json-1.1
Authorization: <AuthParams>
X-Amz-Date: <Date>

{
   ...
}
```

在上述 REST API 请求中，已经在 `Action` 查询参数中指定了操作。或者，您可以在 `X-Amz-Target` 标头中指定操作：

```
POST / HTTP/1.1
Host: kinesis.<region>.<domain>
Content-Length: <PayloadSizeBytes>
User-Agent: <UserAgentString>
Content-Type: application/x-amz-json-1.1
Authorization: <AuthParams>
X-Amz-Date: <Date>
X-Amz-Target: Kinesis_20131202.ListStreams
{
   ...
}
```

在本教程中，我们使用查询参数指定操作。

要在 API 中公开 Kinesis 操作，请将 `/streams` 资源添加到 API 的根中。然后，在此资源上设置 `GET` 方法，并将该方法与 Kinesis 的 `ListStreams` 操作集成。

以下过程介绍如何使用 API Gateway 控制台列出 Kinesis 流。

使用 API Gateway 控制台列出 Kinesis 流

1. 选择 / 资源，然后选择创建资源。
2. 对于资源名称，输入 **`streams`**。
3. 将 CORS（跨源资源共享）保持为关闭状态。
4. 选择创建资源。
5. 选择 `/streams` 资源，再选择创建方法，然后执行以下操作：
   a. 对于方法类型，选择 GET。

      > **Note**
      >
      > 客户端调用的方法的 HTTP 动词可能不同于后端所需的集成的 HTTP 动词。我们在此处选择了 `GET`，因为凭直觉判断，列出流是一个 READ 操作。

   b. 对于集成类型，选择 AWS 服务。
   c. 对于 AWS 区域，选择您创建 Kinesis 流的 AWS 区域。
   d. 对于 AWS 服务，选择 Kinesis。
   e. 将 AWS 子域保留为空白。

   f. 对于 HTTP 方法，选择 POST。

   > **Note**
   >
   > 我们在此处选择了 `POST`，因为 Kinesis 要求使用它来调用 `ListStreams` 操作。

   g. 对于操作类型，选择使用操作名称。

   h. 对于操作名称，输入 **`ListStreams`**。

   i. 对于执行角色，输入执行角色的 ARN。

   j. 对于内容处理，保留默认值传递。

   k. 选择创建方法。

6. 在集成请求选项卡的集成请求设置下，选择编辑。
7. 对于请求正文传递，选择当未定义模板时（推荐）。
8. 选择 URL 请求标头参数并执行以下操作：

   a. 选择添加请求标头参数。

   b. 对于名称，请输入 **`Content-Type`**。

   c. 对于映射自，输入 **`'application/x-amz-json-1.1'`**。

   我们使用请求参数映射将 `Content-Type` 标头设置为静态值 `'application/x-amz-json-1.1'`，以告知 Kinesis 该输入是特定版本的 JSON。

9. 选择映射模板，然后选择添加映射模板并执行以下操作：

   a. 对于 Content-Type，输入 **`application/json`**。

   b. 对于模板正文，输入 **`{}`**。

   c. 选择保存。

   [ListStreams](https://docs.aws.amazon.com/kinesis/latest/APIReference/API_ListStreams.html) 请求使用以下 JSON 格式的负载：

```
{
    "ExclusiveStartStreamName": "string",
    "Limit": number
}
```

但这些属性为可选属性，为了使用默认值，我们在此选择了一个空的 JSON 负载。

10. 在 /streams 资源上测试 GET 方法以调用 Kinesis 中的 `ListStreams` 操作：

    选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。

    选择测试，以测试您的方法。

    如果您已经在 Kinesis 中创建了两个分别名为“myStream”和“yourStream”的流，则成功的测试将返回一个包含以下负载的 200 OK 响应：

```
{
     "HasMoreStreams": false,
     "StreamNames": [
         "myStream",
         "yourStream"
     ]
}
```

在 Kinesis 中创建、描述和删除流

在 Kinesis 中创建、描述和删除流分别需要发出以下 Kinesis REST API 请求：

```
POST /?Action=CreateStream HTTP/1.1
Host: kinesis.region.domain
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "ShardCount": number,
    "StreamName": "string"
}
```

```
POST /?Action=DescribeStream HTTP/1.1
Host: kinesis.region.domain
```

```
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "StreamName": "string"
}
```

```
POST /?Action=DeleteStream HTTP/1.1
Host: kinesis.region.domain
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "StreamName":"string"
}
```

我们可以构建 API 来接受必需的输入作为方法请求的 JSON 负载，并将负载传递到集成请求。但是，为了提供更多方法与集成请求之间以及方法与集成响应之间的数据映射示例，我们创建 API 的方式稍有不同。

我们在待指定的 GET 资源上公开 `POST`、`Delete` 和 `Stream` HTTP 方法。我们使用 `{stream-name}` 路径变量作为流传输资源的占位符，并将这些 API 方法分别与 Kinesis 的 `DescribeStream`、`CreateStream` 和 `DeleteStream` 操作集成。我们要求客户端传递其他输入数据作为标头、查询参数或有效负载的方法请求。我们提供了集成数据所需的映射模板来转换请求负载。

创建 {stream-name} 资源

1. 选择 /streams 资源，然后选择创建资源。
2. 将代理资源保持为关闭状态。
3. 对于资源路径，选择 `/streams`。
4. 对于资源名称，输入 **`{stream-name}`**。
5. 将 CORS（跨源资源共享）保持为关闭状态。
6. 选择创建资源。

在流传输资源上配置和测试 GET 方法

1. 选择 /{stream-name} 资源，然后选择创建方法。
2. 对于方法类型，选择 GET。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您创建 Kinesis 流的 AWS 区域。
5. 对于 AWS 服务，选择 Kinesis。
6. 将 AWS 子域保留为空白。
7. 对于 HTTP 方法，选择 POST。
8. 对于操作类型，选择使用操作名称。
9. 对于操作名称，输入 **`DescribeStream`**。
10. 对于执行角色，输入执行角色的 ARN。
11. 对于内容处理，保留默认值传递。
12. 选择创建方法。
13. 在集成请求部分，添加以下 URL 请求标头参数：

    ```
    Content-Type: 'x-amz-json-1.1'
    ```

    该任务将按照相同的过程为 `GET /streams` 方法设置请求参数映射。

14. 添加以下正文映射模板，以将数据从 `GET /streams/{stream-name}` 方法请求映射到 `POST /?Action=DescribeStream` 集成请求：

    ```
    {
        "StreamName": "$input.params('stream-name')"
    }
    ```

    此映射模板使用方法请求的 `DescribeStream` 路径参数值为 Kinesis 的 `stream-name` 操作生成所需的集成请求负载。

15. 要测试调用 Kinesis 中的 `DescribeStream` 操作的 `GET /stream/{stream-name}` 方法，请选择测试选项卡。
16. 对于路径，在 stream-name 下输入一个现有 Kinesis 流的名称。
17. 选择测试。如果测试成功，将返回一个 200 OK 响应，其所含负载与以下内容类似：

    ```
    {
    ```

```
  "StreamDescription": {
    "HasMoreShards": false,
    "RetentionPeriodHours": 24,
    "Shards": [
      {
        "HashKeyRange": {
          "EndingHashKey": "68056473384187692692674921486353642290",
          "StartingHashKey": "0"
        },
        "SequenceNumberRange": {
          "StartingSequenceNumber":
 "49559266461454070523309915164834022007924120923395850242"
        },
        "ShardId": "shardId-000000000000"
      },
      ...
      {
        "HashKeyRange": {
          "EndingHashKey": "340282366920938463463374607431768211455",
          "StartingHashKey": "272225893536750770770699685945414569164"
        },
        "SequenceNumberRange": {
          "StartingSequenceNumber":
 "49559266461543273504104037657400164881014714369419771970"
        },
        "ShardId": "shardId-000000000004"
      }
    ],
    "StreamARN": "arn:aws:kinesis:us-east-1:12345678901:stream/myStream",
    "StreamName": "myStream",
    "StreamStatus": "ACTIVE"
  }
}
```

部署 API 后，您可以根据此 API 方法做出 REST 请求：

```
GET https://your-api-id.execute-api.region.amazonaws.com/stage/streams/myStream
 HTTP/1.1
Host: your-api-id.execute-api.region.amazonaws.com
Content-Type: application/json
Authorization: ...
X-Amz-Date: 20160323T194451Z
```

在流传输资源上配置和测试 POST 方法

1. 选择 /{stream-name} 资源，然后选择创建方法。
2. 对于方法类型，选择 POST。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您创建 Kinesis 流的 AWS 区域。
5. 对于 AWS 服务，选择 Kinesis。
6. 将 AWS 子域保留为空白。
7. 对于 HTTP 方法，选择 POST。
8. 对于操作类型，选择使用操作名称。
9. 对于操作名称，输入 **`CreateStream`**。
10. 对于执行角色，输入执行角色的 ARN。
11. 对于内容处理，保留默认值传递。
12. 选择创建方法。
13. 在集成请求部分，添加以下 URL 请求标头参数：

    ```
    Content-Type: 'x-amz-json-1.1'
    ```

    该任务将按照相同的过程为 `GET /streams` 方法设置请求参数映射。
14. 添加以下正文映射模板，以将数据从 `POST /streams/{stream-name}` 方法请求映射到 `POST /?Action=CreateStream` 集成请求：

    ```
    {
        "ShardCount": #if($input.path('$.ShardCount') == '') 5 #else
     $input.path('$.ShardCount') #end,
        "StreamName": "$input.params('stream-name')"
    }
    ```

    在上述映射模板中，如果客户端未在方法请求负载中指定值，我们会将 `ShardCount` 设为固定值 5。

15. 要测试调用 Kinesis 中的 `CreateStream` 操作的 `POST /stream/{stream-name}` 方法，请选择测试选项卡。
16. 对于路径，在 stream-name 下输入一个新 Kinesis 流的名称。
17. 选择测试。如果测试成功，将返回一个不含数据的 200 OK 响应。

    部署 API 后，您也可以针对流传输资源上的 POST 方法发出 REST API 请求，以调用 Kinesis 中的 `CreateStream` 操作：

    ```
    POST https://your-api-id.execute-api.region.amazonaws.com/stage/streams/yourStream
     HTTP/1.1
    Host: your-api-id.execute-api.region.amazonaws.com
    Content-Type: application/json
    Authorization: ...
    X-Amz-Date: 20160323T194451Z

    {
        "ShardCount": 5
    }
    ```

在流传输资源上配置和测试 DELETE 方法

1. 选择 /{stream-name} 资源，然后选择创建方法。
2. 对于方法类型，选择 DELETE。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您创建 Kinesis 流的 AWS 区域。
5. 对于 AWS 服务，选择 Kinesis。
6. 将 AWS 子域保留为空白。
7. 对于 HTTP 方法，选择 POST。
8. 对于操作类型，选择使用操作名称。
9. 对于操作名称，输入 **`DeleteStream`**。
10. 对于执行角色，输入执行角色的 ARN。
11. 对于内容处理，保留默认值传递。
12. 选择创建方法。
13. 在集成请求部分，添加以下 URL 请求标头参数：

```
Content-Type: 'x-amz-json-1.1'
```

该任务将按照相同的过程为 `GET /streams` 方法设置请求参数映射。

14. 添加以下正文映射模板，以将数据从 `DELETE /streams/{stream-name}` 方法请求映射到 `POST /?Action=DeleteStream` 的相应集成请求：

    ```
    {
        "StreamName": "$input.params('stream-name')"
    }
    ```

    此映射模板将使用客户端提供的 URL 路径名称 `DELETE /streams/{stream-name}` 为 `stream-name` 操作生成所需的输入。

15. 要测试调用 Kinesis 中的 `DeleteStream` 操作的 `DELETE /stream/{stream-name}` 方法，请选择测试选项卡。
16. 对于路径，在 stream-name 下输入一个现有 Kinesis 流的名称。
17. 选择测试。如果测试成功，将返回一个不含数据的 200 OK 响应。

    部署 API 后，您也可以针对流传输资源上的 DELETE 方法发出以下 REST API 请求，以调用 Kinesis 中的 `DeleteStream` 操作：

    ```
    DELETE https://your-api-id.execute-api.region.amazonaws.com/stage/
    streams/yourStream HTTP/1.1
    Host: your-api-id.execute-api.region.amazonaws.com
    Content-Type: application/json
    Authorization: ...
    X-Amz-Date: 20160323T194451Z

    {}
    ```

从 Kinesis 中的流获取记录并向其添加记录

在 Kinesis 中创建流后，您可以将数据记录添加到流中，也可以从流中读取数据。添加数据记录包括调用 Kinesis 中的 PutRecords 或 PutRecord 操作。前者可向流添加多条记录，而后者可向流添加一条记录。

```
POST /?Action=PutRecords HTTP/1.1
Host: kinesis.region.domain
Authorization: AWS4-HMAC-SHA256 Credential=..., ...
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "Records": [
        {
            "Data": blob,
            "ExplicitHashKey": "string",
            "PartitionKey": "string"
        }
    ],
    "StreamName": "string"
}
```

或

```
POST /?Action=PutRecord HTTP/1.1
Host: kinesis.region.domain
Authorization: AWS4-HMAC-SHA256 Credential=..., ...
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "Data": blob,
    "ExplicitHashKey": "string",
    "PartitionKey": "string",
    "SequenceNumberForOrdering": "string",
    "StreamName": "string"
}
```

其中，`StreamName` 用于标识要添加记录的目标流。`StreamName`、`Data` 和 `PartitionKey` 是必需的输入数据。在示例中，我们针对所有可选输入数据使用了默认值，并且不会在方法请求的输入中显式指定它们的值。

读取 Kinesis 中的数据相当于调用 GetRecords 操作：

```
POST /?Action=GetRecords HTTP/1.1
Host: kinesis.region.domain
Authorization: AWS4-HMAC-SHA256 Credential=..., ...
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "ShardIterator": "string",
    "Limit": number
}
```

其中，我们要从中获取记录的源流在必需的 `ShardIterator` 值中进行指定，如以下获取分片迭代器的 Kinesis 操作中所示：

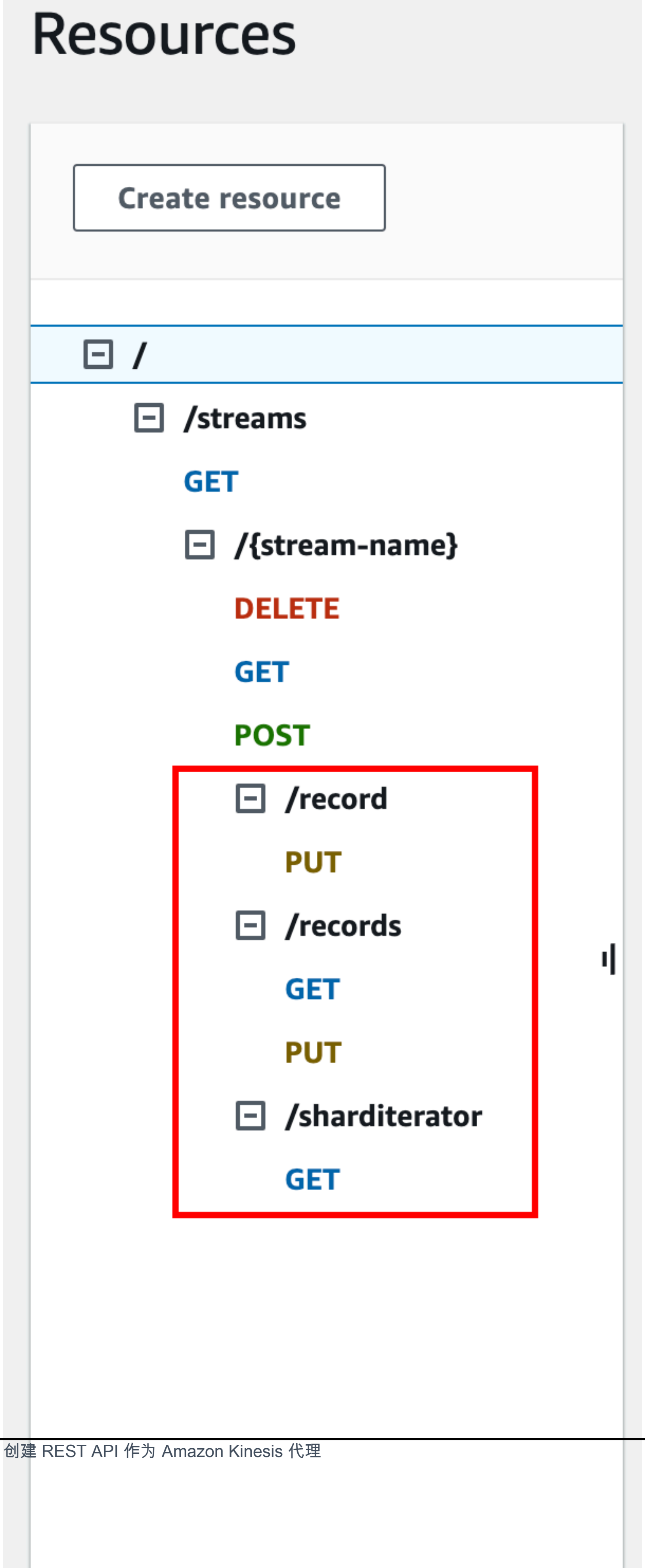
```

POST /?Action=GetShardIterator HTTP/1.1
Host: kinesis.region.domain
Authorization: AWS4-HMAC-SHA256 Credential=..., ...
...
Content-Type: application/x-amz-json-1.1
Content-Length: PayloadSizeBytes

{
    "ShardId": "string",
    "ShardIteratorType": "string",
    "StartingSequenceNumber": "string",
    "StreamName": "string"
}
```

对于 `GetRecords` 和 `PutRecords` 操作，我们在附加到指定流传输资源 (GET) 的 PUT 资源上分别使用了 `/records` 和 `/{stream-name}` 方法。同样，我们在 `PutRecord` 资源上使用了 PUT 操作作为 `/record` 方法。

由于 `GetRecords` 操作将 `ShardIterator` 值 (该值通过调用 `GetShardIterator` 辅助标记操作获得) 作为输入，我们在 GET 资源 (`ShardIterator`) 上使用了 `/sharditerator` 辅助标记方法。

创建 /record、/records 和 /sharditerator 资源

1. 选择 /{stream-name} 资源，然后选择创建资源。
2. 将代理资源保持为关闭状态。
3. 对于资源路径，选择 `/{stream-name}`。
4. 对于资源名称，输入 **`record`**。
5. 将 CORS（跨源资源共享）保持为关闭状态。
6. 选择创建资源。
7. 重复上述步骤，创建 /records 和 /sharditerator 资源。最终 API 应与以下内容类似：



Resources

Create resource

- /
  - /streams
    - GET
    - /{stream-name}
      - DELETE
      - GET
      - POST
      - /record
        - PUT
      - /records
        - GET
        - PUT
      - /sharditerator
        - GET

以下四个过程介绍了如何设置每个方法，如何将数据从方法请求映射到集成请求，以及如何测试方法。

设置并测试 `PUT /streams/{stream-name}/record` 方法以调用 Kinesis 中的 `PutRecord`

1. 选择 /record，然后选择创建方法。
2. 对于方法类型，选择 PUT。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您创建 Kinesis 流的 AWS 区域。
5. 对于 AWS 服务，选择 Kinesis。
6. 将 AWS 子域保留为空白。
7. 对于 HTTP 方法，选择 POST。
8. 对于操作类型，选择使用操作名称。
9. 对于操作名称，输入 **PutRecord**。
10. 对于执行角色，输入执行角色的 ARN。
11. 对于内容处理，保留默认值传递。
12. 选择创建方法。
13. 在集成请求部分，添加以下 URL 请求标头参数：

    ```
    Content-Type: 'x-amz-json-1.1'
    ```

    该任务将按照相同的过程为 `GET /streams` 方法设置请求参数映射。
14. 添加以下正文映射模板，以将数据从 `PUT /streams/{stream-name}/record` 方法请求映射到 `POST /?Action=PutRecord` 的相应集成请求：

    ```
    {
        "StreamName": "$input.params('stream-name')",
        "Data": "$util.base64Encode($input.json('$.Data'))",
        "PartitionKey": "$input.path('$.PartitionKey')"
    }
    ```

    此映射模板假定方法请求负载采用的是以下格式：

    ```
    {
       "Data": "some data",
       "PartitionKey": "some key"
    ```

```
}
```

此数据可通过以下 JSON 架构建模：

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "PutRecord proxy single-record payload",
  "type": "object",
  "properties": {
      "Data": { "type": "string" },
      "PartitionKey": { "type": "string" }
  }
}
```

您可以创建一个模型以包含此架构，并使用该模型来帮助生成映射模板。但您也可以在不使用任何模型的情况下生成映射模板。

15. 要测试 PUT `/streams/{stream-name}/record` 方法，请将 `stream-name` 路径变量设置为现有流的名称，提供所需格式的负载，然后提交方法请求。如果成功，将返回一个包含以下格式负载的 `200 OK` 响应：

```
{
  "SequenceNumber": "49559409944537880850133345460169886593573102115167928386",
  "ShardId": "shardId-000000000004"
}
```

设置并测试 **`PUT /streams/{stream-name}/records`** 方法以调用 Kinesis 中的 **`PutRecords`**

1. 选择 /records 资源，然后选择创建方法。
2. 对于方法类型，选择 PUT。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您创建 Kinesis 流的 AWS 区域。
5. 对于 AWS 服务，选择 Kinesis。
6. 将 AWS 子域保留为空白。
7. 对于 HTTP 方法，选择 POST。
8. 对于操作类型，选择使用操作名称。

9. 对于操作名称，输入 **`PutRecords`**。
10. 对于执行角色，输入执行角色的 ARN。
11. 对于内容处理，保留默认值传递。
12. 选择创建方法。
13. 在集成请求部分，添加以下 URL 请求标头参数：

    ```
    Content-Type: 'x-amz-json-1.1'
    ```

    该任务将按照相同的过程为 `GET /streams` 方法设置请求参数映射。
14. 添加以下映射模板，以将数据从 `PUT /streams/{stream-name}/records` 方法请求映射到 `POST /?Action=PutRecords` 的相应集成请求：

    ```
    {
        "StreamName": "$input.params('stream-name')",
        "Records": [
           #foreach($elem in $input.path('$.records'))
              {
                "Data": "$util.base64Encode($elem.data)",
                "PartitionKey": "$elem.partition-key"
              }#if($foreach.hasNext),#end
            #end
        ]
    }
    ```

    此映射模板假设可以通过以下 JSON 架构为方法请求负载建模：

    ```
    {
      "$schema": "http://json-schema.org/draft-04/schema#",
      "title": "PutRecords proxy payload data",
      "type": "object",
      "properties": {
        "records": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "data": { "type": "string" },
              "partition-key": { "type": "string" }
            }
    ```

```
          }
        }
      }
    }
```

您可以创建一个模型以包含此架构，并使用该模型来帮助生成映射模板。但您也可以在不使用任何模型的情况下生成映射模板。

在本教程中，我们使用了两种稍有不同的负载格式来说明 API 开发人员可以选择向客户端公开或隐藏后端数据格式。一种格式用于 `PUT /streams/{stream-name}/records` 方法 (上文)。另一种格式用于 `PUT /streams/{stream-name}/record` 方法 (上一过程)。在生产环境中，您应该将两种格式保持一致。

15. 要测试 `PUT /streams/{stream-name}/records` 方法，请将 `stream-name` 路径变量设置为现有流，提供以下负载，并提交方法请求。

```

{
    "records": [
        {
            "data": "some data",
            "partition-key": "some key"
        },
        {
            "data": "some other data",
            "partition-key": "some key"
        }
    ]
}

```

如果成功，将返回一个包含类似以下输出的负载的 200 OK 响应：

```
{
  "FailedRecordCount": 0,
  "Records": [
    {
      "SequenceNumber": "49559409944537880850133345460167468741933742152373764162",
      "ShardId": "shardId-000000000004"
    },
    {
```

```
      "SequenceNumber": "49559409944537880850133345460168677667753356781548470338",
      "ShardId": "shardId-000000000004"
    }
  ]
}
```

设置并测试 `GET /streams/{stream-name}/sharditerator` 方法以调用 Kinesis 中的 `GetShardIterator`

`GET /streams/{stream-name}/sharditerator` 方法为辅助标记方法，用于在调用 `GET /streams/{stream-name}/records` 方法之前获得必需的分片迭代器。

1. 选择 /sharditerator 资源，然后选择创建方法。
2. 对于方法类型，选择 GET。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您创建 Kinesis 流的 AWS 区域。
5. 对于 AWS 服务，选择 Kinesis。
6. 将 AWS 子域保留为空白。
7. 对于 HTTP 方法，选择 POST。
8. 对于操作类型，选择使用操作名称。
9. 对于操作名称，输入 **`GetShardIterator`**。
10. 对于执行角色，输入执行角色的 ARN。
11. 对于内容处理，保留默认值传递。
12. 选择 URL 查询字符串参数。

    `GetShardIterator` 操作需要输入 ShardId 值。要传递客户端提供的 `ShardId` 值，我们将向方法请求添加一个 `shard-id` 查询参数，如以下步骤所示。
13. 选择添加查询字符串。
14. 在名称中，输入 **`shard-id`**。
15. 保持必填和缓存为已关闭状态。
16. 选择创建方法。
17. 在集成请求部分中，添加以下映射模板，以从方法请求的 `shard-id` 和 `stream-name` 参数生成 `GetShardIterator` 操作所需的输入（`ShardId` 和 `StreamName`）。此外，映射模板还需将 `ShardIteratorType` 设置为 `TRIM_HORIZON`，并作为默认值。

```
{
    "ShardId": "$input.params('shard-id')",
    "ShardIteratorType": "TRIM_HORIZON",
    "StreamName": "$input.params('stream-name')"
}
```

18. 使用 API Gateway 控制台中的测试选项，输入现有流名称作为 `stream-name` 路径变量值，将 `shard-id` 查询字符串设置为现有 `ShardId` 值（例如 `shard-000000000004`），然后选择测试。

    成功的响应负载与以下输出类似：

    ```
    {
      "ShardIterator": "AAAAAAAAAAFYVN3VlFy..."
    }
    ```

    记下此 `ShardIterator` 值。您需要使用此值来从流中获取记录。

配置并测试 **`GET /streams/{stream-name}/records`** 方法以调用 Kinesis 中的 **`GetRecords`** 操作

1. 选择 /records 资源，然后选择创建方法。
2. 对于方法类型，选择 GET。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您创建 Kinesis 流的 AWS 区域。
5. 对于 AWS 服务，选择 Kinesis。
6. 将 AWS 子域保留为空白。
7. 对于 HTTP 方法，选择 POST。
8. 对于操作类型，选择使用操作名称。
9. 对于操作名称，输入 **`GetRecords`**。
10. 对于执行角色，输入执行角色的 ARN。
11. 对于内容处理，保留默认值传递。
12. 选择 HTTP 请求标头。

`GetRecords` 操作需要输入 `ShardIterator` 值。要传递客户端提供的 `ShardIterator` 值，我们将向方法请求添加一个 `Shard-Iterator` 标头参数。

13. 选择添加标头。
14. 对于名称，请输入 **`Shard-Iterator`**。
15. 保持必填和缓存为已关闭状态。
16. 选择创建方法。
17. 在集成请求部分中，添加以下正文映射模板，以便为 Kinesis 中的 `GetRecords` 操作将 `Shard-Iterator` 标头参数值映射到 JSON 负载的 `ShardIterator` 属性值。

```
{
    "ShardIterator": "$input.params('Shard-Iterator')"
}
```

18. 使用 API Gateway 控制台中的测试选项，输入现有流名称作为 `stream-name` 路径变量值，将 `Shard-Iterator` 标头设置为从 `GET /streams/{stream-name}/sharditerator` 方法的测试运行中获得的 `ShardIterator` 值（上文），然后选择测试。

    成功的响应负载与以下输出类似：

```
{
  "MillisBehindLatest": 0,
  "NextShardIterator": "AAAAAAAAAAF...",
  "Records": [ ... ]
}
```

作为 Kinesis 代理的示例 API 的 OpenAPI 定义

以下是本教程中使用的作为 Kinesis 代理的示例 API 的 OpenAPI 定义。

OpenAPI 3.0

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "KinesisProxy",
    "version": "2016-03-31T18:25:32Z"
  },
  "paths": {
```

```
    "/streams/{stream-name}/sharditerator": {
      "get": {
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          },
          {
            "name": "shard-id",
            "in": "query",
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/GetShardIterator",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
```

```
          "requestTemplates": {
            "application/json": "{\n    \"ShardId\": \"$input.params('shard-
id')\",\n    \"ShardIteratorType\": \"TRIM_HORIZON\",\n    \"StreamName\":
 \"$input.params('stream-name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    },
    "/streams/{stream-name}/records": {
      "get": {
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          },
          {
            "name": "Shard-Iterator",
            "in": "header",
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
```

```
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/GetRecords",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"ShardIterator\": \"$input.params('Shard-
Iterator')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      },
      "put": {
        "parameters": [
          {
            "name": "Content-Type",
            "in": "header",
            "schema": {
              "type": "string"
            }
          },
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/PutRecordsMethodRequestPayload"
              }
            },
            "application/x-amz-json-1.1": {
```

```
              "schema": {
                "$ref": "#/components/schemas/PutRecordsMethodRequestPayload"
              }
            }
          },
          "required": true
        },
        "responses": {
          "200": {
            "description": "200 response",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/PutRecords",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\",\n    \"Records\": [\n          {\n            \"Data\":
 \"$util.base64Encode($elem.data)\",\n            \"PartitionKey\":
 \"$elem.partition-key\"\n          }#if($foreach.hasNext),#end\n    ]\n}",
            "application/x-amz-json-1.1": "{\n  \"StreamName\":
 \"$input.params('stream-name')\",\n  \"records\" : [\n    {\n        \"Data
\" : \"$elem.data\",\n        \"PartitionKey\" : \"$elem.partition-key\"\n
  }#if($foreach.hasNext),#end\n  ]\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
```

```
        }
      }
    },
    "/streams/{stream-name}": {
      "get": {
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/DescribeStream",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestTemplates": {
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      },
```

```
      "post": {
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/CreateStream",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"ShardCount\": 5,\n    \"StreamName\":
 \"$input.params('stream-name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      },
```

```
      "delete": {
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "headers": {
              "Content-Type": {
                "schema": {
                  "type": "string"
                }
              }
            },
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
            }
          },
          "400": {
            "description": "400 response",
            "headers": {
              "Content-Type": {
                "schema": {
                  "type": "string"
                }
              }
            },
            "content": {}
          },
          "500": {
            "description": "500 response",
            "headers": {
              "Content-Type": {
```

```
                "schema": {
                  "type": "string"
                }
              }
            },
            "content": {}
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/DeleteStream",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type"
              }
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type"
              }
            },
            "5\\d{2}": {
              "statusCode": "500",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type"
              }
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
```

```
          "httpMethod": "POST"
        }
      }
    },
    "/streams/{stream-name}/record": {
      "put": {
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/PutRecord",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
```

```
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\",\n    \"Data\": \"$util.base64Encode($input.json('$.Data'))\",\n
 \"PartitionKey\": \"$input.path('$.PartitionKey')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    },
    "/streams": {
      "get": {
        "responses": {
          "200": {
            "description": "200 response",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Empty"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/ListStreams",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    }
```

```
  },
  "components": {
    "schemas": {
      "Empty": {
        "type": "object"
      },
      "PutRecordsMethodRequestPayload": {
        "type": "object",
        "properties": {
          "records": {
            "type": "array",
            "items": {
              "type": "object",
              "properties": {
                "data": {
                  "type": "string"
                },
                "partition-key": {
                  "type": "string"
                }
              }
            }
          }
        }
      }
    }
  }
}
```

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-03-31T18:25:32Z",
    "title": "KinesisProxy"
  },
  "basePath": "/test",
  "schemes": [
    "https"
  ],
  "paths": {
    "/streams": {
```

```
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/ListStreams",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    },
    "/streams/{stream-name}": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
```

```
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/DescribeStream",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestTemplates": {
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      },
      "post": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
```

```
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/CreateStream",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"ShardCount\": 5,\n    \"StreamName\":
 \"$input.params('stream-name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      },
      "delete": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
```

```
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            },
            "headers": {
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "400": {
            "description": "400 response",
            "headers": {
              "Content-Type": {
                "type": "string"
              }
            }
          },
          "500": {
            "description": "500 response",
            "headers": {
              "Content-Type": {
                "type": "string"
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/DeleteStream",
          "responses": {
            "4\\d{2}": {
              "statusCode": "400",
              "responseParameters": {
                "method.response.header.Content-Type": 
 "integration.response.header.Content-Type"
              }
```

```
            },
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type"
              }
            },
            "5\\d{2}": {
              "statusCode": "500",
              "responseParameters": {
                "method.response.header.Content-Type":
 "integration.response.header.Content-Type"
              }
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    },
    "/streams/{stream-name}/record": {
      "put": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "type": "string"
          }
```

```
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/PutRecord",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\",\n    \"Data\": \"$util.base64Encode($input.json('$.Data'))\",\n
 \"PartitionKey\": \"$input.path('$.PartitionKey')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    },
    "/streams/{stream-name}/records": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
```

```
              "required": true,
              "type": "string"
            },
            {
              "name": "Shard-Iterator",
              "in": "header",
              "required": false,
              "type": "string"
            }
          ],
          "responses": {
            "200": {
              "description": "200 response",
              "schema": {
                "$ref": "#/definitions/Empty"
              }
            }
          },
          "x-amazon-apigateway-integration": {
            "type": "aws",
            "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
            "uri": "arn:aws:apigateway:us-east-1:kinesis:action/GetRecords",
            "responses": {
              "default": {
                "statusCode": "200"
              }
            },
            "requestParameters": {
              "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
            },
            "requestTemplates": {
              "application/json": "{\n    \"ShardIterator\": \"$input.params('Shard-
Iterator')\"\n}"
            },
            "passthroughBehavior": "when_no_match",
            "httpMethod": "POST"
          }
        },
        "put": {
          "consumes": [
            "application/json",
            "application/x-amz-json-1.1"
          ],
```

```
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "Content-Type",
            "in": "header",
            "required": false,
            "type": "string"
          },
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "in": "body",
            "name": "PutRecordsMethodRequestPayload",
            "required": true,
            "schema": {
              "$ref": "#/definitions/PutRecordsMethodRequestPayload"
            }
          },
          {
            "in": "body",
            "name": "PutRecordsMethodRequestPayload",
            "required": true,
            "schema": {
              "$ref": "#/definitions/PutRecordsMethodRequestPayload"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
```

```
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/PutRecords",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"StreamName\": \"$input.params('stream-
name')\",\n    \"Records\": [\n          {\n            \"Data\":
 \"$util.base64Encode($elem.data)\",\n            \"PartitionKey\":
 \"$elem.partition-key\"\n          }#if($foreach.hasNext),#end\n    ]\n}",
            "application/x-amz-json-1.1": "{\n  \"StreamName\":
 \"$input.params('stream-name')\",\n  \"records\" : [\n    {\n        \"Data
\" : \"$elem.data\",\n        \"PartitionKey\" : \"$elem.partition-key\"\n
  }#if($foreach.hasNext),#end\n  ]\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    },
    "/streams/{stream-name}/sharditerator": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "stream-name",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "shard-id",
            "in": "query",
            "required": false,
```

```
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "aws",
          "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
          "uri": "arn:aws:apigateway:us-east-1:kinesis:action/GetShardIterator",
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.header.Content-Type": "'application/x-amz-
json-1.1'"
          },
          "requestTemplates": {
            "application/json": "{\n    \"ShardId\": \"$input.params('shard-
id')\",\n    \"ShardIteratorType\": \"TRIM_HORIZON\",\n    \"StreamName\":
 \"$input.params('stream-name')\"\n}"
          },
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST"
        }
      }
    }
  },
  "definitions": {
    "Empty": {
      "type": "object"
    },
    "PutRecordsMethodRequestPayload": {
      "type": "object",
      "properties": {
        "records": {
          "type": "array",
```

```
          "items": {
            "type": "object",
            "properties": {
              "data": {
                "type": "string"
              },
              "partition-key": {
                "type": "string"
              }
            }
          }
        }
      }
    }
  }
}
```

教程：使用 AWS SDK 或 AWS CLI 设置边缘优化的 API

以下教程显示如何创建支持 `GET /pets` 和 `GET /pets/{petId}` 方法的 PetStore API。这些方法与 HTTP 端点集成。您可以按照本教程的说明，使用适用于 JavaScript 的 AWS SDK、适用于 Python 的 SDK（Boto3）或 AWS CLI。您可以使用以下函数或命令来设置 API：

JavaScript v3

- CreateRestApiCommand
- CreateResourceCommand
- PutMethodCommand
- PutMethodResponseCommand
- PutIntegrationCommand
- PutIntegrationResponseCommand
- CreateDeploymentCommand

Python

- create_rest_api
- create_resource
- put_method

- put_method_response
- put_integration
- put_integration_response
- create_deployment

AWS CLI

- create-rest-api
- create-resource
- put-method
- put-method-response
- put-integration
- put-integration-response
- create-deployment

有关适用于 JavaScript 的 AWS SDK v3 的更多信息，请参阅 What's the AWS SDK for JavaScript? 有关适用于 Python 的 SDK（Boto3）的更多信息，请参阅 AWS SDK for Python (Boto3)。有关 AWS CLI 的更多信息，请参阅 What is the AWS CLI?

设置边缘优化的 PetStore API

在本教程中，示例命令使用占位符值作为值 ID，例如 API ID 和资源 ID。在完成本教程时，请将这些值替换为您自己的值。

使用 AWS SDK 设置边缘优化的 PetStore API

1. 使用以下示例来创建 `RestApi` 实体：

   JavaScript v3

```
import {APIGatewayClient, CreateRestApiCommand} from "@aws-sdk/client-api-
gateway";
(async function (){
const apig = new APIGatewayClient({region:"us-east-1"});
const command = new CreateRestApiCommand({
    name: "Simple PetStore (JavaScript v3 SDK)",
    description: "Demo API created using the AWS SDK for JavaScript v3",
```

```
    version: "0.00.001",
    binaryMediaTypes: [
    '*']
});
try {
    const results = await apig.send(command)
    console.log(results)
} catch (err) {
    console.error(Couldn't create API:\n", err)
}
})();
```

成功的调用会在输出中返回 API ID 和 API 的根资源 ID，如下所示：

```
{
  id: 'abc1234',
  name: 'PetStore (JavaScript v3 SDK)',
  description: 'Demo API created using the AWS SDK for node.js',
  createdDate: 2017-09-05T19:32:35.000Z,
  version: '0.00.001',
  rootResourceId: 'efg567'
  binaryMediaTypes: [ '*' ]
}
```

Python

```
import botocore
import boto3
import logging

logger = logging.getLogger()
apig = boto3.client('apigateway')

try:
    result = apig.create_rest_api(
        name='Simple PetStore (Python SDK)',
        description='Demo API created using the AWS SDK for Python',
        version='0.00.001',
        binaryMediaTypes=[
            '*'
        ]
    )
```

```
except botocore.exceptions.ClientError as error:
    logger.exception("Couldn't create REST API %s.", error)
    raise
attribute=["id","name","description","createdDate","version","binaryMediaTypes","apiKeyS
filtered_result ={key:result[key] for key in attribute}
print(filtered_result)
```

成功的调用会在输出中返回 API ID 和 API 的根资源 ID，如下所示：

```
{'id': 'abc1234', 'name': 'Simple PetStore (Python SDK)', 'description':
 'Demo API created using the AWS SDK for Python', 'createdDate':
 datetime.datetime(2024, 4, 3, 14, 31, 39, tzinfo=tzlocal()), 'version':
 '0.00.001', 'binaryMediaTypes': ['*'], 'apiKeySource': 'HEADER',
 'endpointConfiguration': {'types': ['EDGE']}, 'disableExecuteApiEndpoint':
 False, 'rootResourceId': 'efg567'}
```

AWS CLI

```
aws apigateway create-rest-api --name 'Simple PetStore (AWS CLI)' --region us-
west-2
```

此命令的输出如下：

```
{
    "id": "abcd1234",
    "name": "Simple PetStore (AWS CLI)",
    "createdDate": "2022-12-15T08:07:04-08:00",
    "apiKeySource": "HEADER",
    "endpointConfiguration": {
        "types": [
            "EDGE"
        ]
    },
    "disableExecuteApiEndpoint": false,
    "rootResourceId": "efg567"
}
```

您创建的 API 的 API ID 为 `abcd1234`，根资源 ID 为 `efg567`。您可以在 API 的设置中使用这些值。

2. 接下来，在根目录下附加子资源，您可以将 `RootResourceId` 指定为 `parentId` 属性值。使用以下示例为 API 创建 `/pets` 资源：

   JavaScript v3

   ```
   import {APIGatewayClient,  CreateResourceCommand } from "@aws-sdk/client-api-gateway";
   (async function (){
   const apig = new APIGatewayClient({region:"us-east-1"});
   const command = new CreateResourceCommand({
       restApiId: 'abcd1234',
       parentId: 'efg567',
       pathPart: 'pets'
   });
   try {
       const results = await apig.send(command)
       console.log(results)
   } catch (err) {
       console.log("The '/pets' resource setup failed:\n", err)
   }
   })();
   ```

   成功的调用会在输出中返回有关您的资源的信息，如下所示：

   ```
   {
       "path": "/pets",
       "pathPart": "pets",
       "id": "aaa111",
       "parentId": "efg567'"
   }
   ```

   Python

   ```
   import botocore
   import boto3
   import logging

   logger = logging.getLogger()
   apig = boto3.client('apigateway')

   try:
       result = apig.create_resource(
   ```

```
            restApiId='abcd1234',
            parentId='efg567',
            pathPart='pets'
        )
except botocore.exceptions.ClientError as error:
    logger.exception("The '/pets' resource setup failed: %s.", error)
    raise
attribute=["id","parentId", "pathPart", "path",]
filtered_result ={key:result[key] for key in attribute}
print(filtered_result)
```

成功的调用会在输出中返回有关您的资源的信息，如下所示：

```
{'id': 'aaa111', 'parentId': 'efg567', 'pathPart': 'pets', 'path': '/pets'}
```

AWS CLI

```
aws apigateway create-resource --rest-api-id abcd1234 \
  --region us-west-2 \
  --parent-id efg567 \
  --path-part pets
```

此命令的输出如下：

```
{
    "id": "aaa111",
    "parentId": "efg567",
    "pathPart": "pets",
    "path": "/pets"
}
```

您创建的 `/pets` 资源的资源 ID 为 `aaa111`。您可以在 API 的设置中使用此值。

3. 接下来，在 `/pets` 资源下附加子资源。此资源 `/{petId}` 具有 `{petId}` 的路径参数。为了让路径部分成为路径参数，请将其括在一对大括号 `{ }` 内。使用以下示例为 API 创建 `/pets/{petId}` 资源：

JavaScript v3

```
import {APIGatewayClient,  CreateResourceCommand } from "@aws-sdk/client-api-
gateway";
(async function (){
const apig = new APIGatewayClient({region:"us-east-1"});
const command = new CreateResourceCommand({
    restApiId: 'abcd1234',
    parentId: 'aaa111',
    pathPart: '{petId}'
});
try {
    const results = await apig.send(command)
    console.log(results)
} catch (err) {
    console.log("The '/pets/{petId}' resource setup failed:\n", err)
}
})();
```

成功的调用会在输出中返回有关您的资源的信息，如下所示：

```
{
    "path": "/pets/{petId}",
    "pathPart": "{petId}",
    "id": "bbb222",
    "parentId": "aaa111'"
}
```

Python

```
import botocore
import boto3
import logging

logger = logging.getLogger()
apig = boto3.client('apigateway')

try:
    result = apig.create_resource(
        restApiId='abcd1234',
        parentId='aaa111',
```

```
        pathPart='{petId}'
    )
except botocore.exceptions.ClientError as error:
    logger.exception("The '/pets/{petId}' resource setup failed: %s.", error)
    raise
attribute=["id","parentId", "pathPart", "path",]
filtered_result ={key:result[key] for key in attribute}
print(filtered_result)
```

成功的调用会在输出中返回有关您的资源的信息，如下所示：

```
{'id': 'bbb222', 'parentId': 'aaa111', 'pathPart': '{petId}', 'path': '/pets/
{petId}'}
```

AWS CLI

```
aws apigateway create-resource --rest-api-id abcd1234 \
  --region us-west-2 \
  --parent-id aaa111 \
  --path-part '{petId}'
```

如果成功，该命令会返回以下响应：

```
{
    "id": "bbb222",
    "parentId": "aaa111",
    "path": "/pets/{petId}",
    "pathPart": "{petId}"
}
```

您创建的 `/pets/{petId}` 资源的资源 ID 为 `bbb222`。您可以在 API 的设置中使用此值。

4. 在以下两个步骤中，您向资源添加 HTTP 方法。在本教程中，您可以通过将 `authorization-type` 设置为 `NONE`，来将方法设置为具有开放访问权限。要仅允许已验证身份的用户调用此方法，您可以使用 IAM 角色和策略、Lambda 授权方（以前称为自定义授权方）或者 Amazon Cognito 用户池。有关更多信息，请参阅 the section called “访问控制”。

   以下示例在 `/pets` 资源上添加 `GET` HTTP 方法：

JavaScript v3

```
import {APIGatewayClient,  PutMethodCommand } from "@aws-sdk/client-api-
gateway";
(async function (){
const apig = new APIGatewayClient({region:"us-east-1"});
const command = new PutMethodCommand({
    restApiId: 'abcd1234',
    resourceId: 'aaa111',
    httpMethod: 'GET',
    authorizationType: 'NONE'
});
try {
    const results = await apig.send(command)
    console.log(results)
} catch (err) {
    console.log("The 'GET /pets' method setup failed:\n", err)
}
})();
```

成功的调用返回以下输出：

```
{
    "apiKeyRequired": false,
    "httpMethod": "GET",
    "authorizationType": "NONE"
}
```

Python

```
import botocore
import boto3
import logging

logger = logging.getLogger()
apig = boto3.client('apigateway')

try:
    result = apig.put_method(
        restApiId='abcd1234',
        resourceId='aaa111',
```

```
        httpMethod='GET',
        authorizationType='NONE'
    )
except botocore.exceptions.ClientError as error:
    logger.exception("The 'GET /pets' method setup failed: %s", error)
    raise
attribute=["httpMethod","authorizationType","apiKeyRequired"]
filtered_result ={key:result[key] for key in attribute}
print(filtered_result)
```

成功的调用返回以下输出：

```
{'httpMethod': 'GET', 'authorizationType': 'NONE', 'apiKeyRequired': False}
```

AWS CLI

```
aws apigateway put-method --rest-api-id abcd1234 \
  --resource-id aaa111 \
  --http-method GET \
  --authorization-type "NONE" \
  --region us-west-2
```

此命令的成功输出如下：

```
{
    "httpMethod": "GET",
    "authorizationType": "NONE",
    "apiKeyRequired": false
}
```

5. 以下示例在 `/pets/{petId}` 资源上添加 GET HTTP 方法，并设置 `requestParameters` 属性来将客户端提供的 `petId` 值传递给后端：

JavaScript v3

```
import {APIGatewayClient,  PutMethodCommand } from "@aws-sdk/client-api-
gateway";
(async function (){
const apig = new APIGatewayClient({region:"us-east-1"});
const command = new PutMethodCommand({
    restApiId: 'abcd1234',
```

```
    resourceId: 'bbb222',
    httpMethod: 'GET',
    authorizationType: 'NONE'
    requestParameters: {
        "method.request.path.petId" : true
    }
});
try {
    const results = await apig.send(command)
    console.log(results)
} catch (err) {
    console.log("The 'GET /pets/{petId}' method setup failed:\n", err)
}
})();
```

成功的调用返回以下输出：

```
{
    "apiKeyRequired": false,
    "httpMethod": "GET",
    "authorizationType": "NONE",
    "requestParameters": {
       "method.request.path.petId": true
    }
}
```

Python

```
import botocore
import boto3
import logging

logger = logging.getLogger()
apig = boto3.client('apigateway')

try:
    result = apig.put_method(
        restApiId='abcd1234',
        resourceId='bbb222',
        httpMethod='GET',
        authorizationType='NONE',
        requestParameters={
```

```
            "method.request.path.petId": True
        }
    )
except botocore.exceptions.ClientError as error:
    logger.exception("The 'GET /pets/{petId}' method setup failed: %s", error)
    raise
attribute=["httpMethod","authorizationType","apiKeyRequired",
 "requestParameters" ]
filtered_result ={key:result[key] for key in attribute}
print(filtered_result)
```

成功的调用返回以下输出：

```
{'httpMethod': 'GET', 'authorizationType': 'NONE', 'apiKeyRequired': False,
 'requestParameters': {'method.request.path.petId': True}}
```

AWS CLI

```
aws apigateway put-method --rest-api-id abcd1234 \
  --resource-id bbb222 --http-method GET \
  --authorization-type "NONE" \
  --region us-west-2 \
  --request-parameters method.request.path.petId=true
```

此命令的成功输出如下：

```
{
    "httpMethod": "GET",
    "authorizationType": "NONE",
    "apiKeyRequired": false,
    "requestParameters": {
        "method.request.path.petId": true
    }
}
```

6. 使用以下示例为 `GET /pets` 方法添加 200 OK 方法响应：

JavaScript v3

```
import {APIGatewayClient,  PutMethodResponseCommand } from "@aws-sdk/client-api-
gateway";
```

```
(async function (){
const apig = new APIGatewayClient({region:"us-east-1"});
const command = new PutMethodResponseCommand({
    restApiId: 'abcd1234',
    resourceId: 'aaa111',
    httpMethod: 'GET',
    statusCode: '200'
});
try {
    const results = await apig.send(command)
    console.log(results)
} catch (err) {
    console.log("Set up the 200 OK response for the 'GET /pets' method failed:
\n", err)
}
})();
```

成功的调用返回以下输出：

```
{
    "statusCode": "200"
}
```

Python

```
import botocore
import boto3
import logging

logger = logging.getLogger()
apig = boto3.client('apigateway')

try:
    result = apig.put_method_response(
        restApiId='abcd1234',
        resourceId='aaa111',
        httpMethod='GET',
        statusCode='200'
    )
except botocore.exceptions.ClientError as error:
    logger.exception("Set up the 200 OK response for the 'GET /pets' method
 failed %s.", error)
```

```
    raise
attribute=["statusCode"]
filtered_result ={key:result[key] for key in attribute}
logger.info(filtered_result)
```

成功的调用返回以下输出：

```
{'statusCode': '200'}
```

AWS CLI

```
aws apigateway put-method-response --rest-api-id abcd1234 \
  --resource-id aaa111 --http-method GET \
  --status-code 200  --region us-west-2
```

此命令的输出如下：

```
{
    "statusCode": "200"
}
```

7. 使用以下示例为 `GET /pets/{petId}` 方法添加 200 OK 方法响应：

JavaScript v3

```
import {APIGatewayClient,  PutMethodResponseCommand } from "@aws-sdk/client-api-
gateway";
(async function (){
const apig = new APIGatewayClient({region:"us-east-1"});
const command = new PutMethodResponseCommand({
    restApiId: 'abcd1234',
    resourceId: 'bbb222',
    httpMethod: 'GET',
    statusCode: '200'
});
try {
    const results = await apig.send(command)
    console.log(results)
} catch (err) {
    console.log("Set up the 200 OK response for the 'GET /pets/{petId}' method
 failed:\n", err)
```

```
}
})();
```

成功的调用返回以下输出：

```
{
    "statusCode": "200"
}
```

Python

```
import botocore
import boto3
import logging

logger = logging.getLogger()
apig = boto3.client('apigateway')

try:
    result = apig.put_method_response(
        restApiId='abcd1234',
        resourceId='bbb222',
        httpMethod='GET',
        statusCode='200'
    )
except botocore.exceptions.ClientError as error:
    logger.exception("Set up the 200 OK response for the 'GET /pets/{petId}'
 method failed %s.", error)
    raise
attribute=["statusCode"]
filtered_result ={key:result[key] for key in attribute}
logger.info(filtered_result)
```

成功的调用返回以下输出：

```
{'statusCode': '200'}
```

AWS CLI

```
aws apigateway put-method-response --rest-api-id abcd1234 \
  --resource-id bbb222 --http-method GET \
```

```
  --status-code 200  --region us-west-2
```

此命令的输出如下：

```
{
    "statusCode": "200"
}
```

8. 以下示例使用 HTTP 端点为 `GET /pets` 方法配置集成。HTTP 端点为 `http://petstore-demo-endpoint.execute-api.com/petstore/pets`。

   JavaScript v3

```
import {APIGatewayClient,  PutIntegrationCommand } from "@aws-sdk/client-api-
gateway";
(async function (){
const apig = new APIGatewayClient({region:"us-east-1"});
const command = new PutIntegrationCommand({
    restApiId: 'abcd1234',
    resourceId: 'aaa111',
    httpMethod: 'GET',
    type: 'HTTP',
    integrationHttpMethod: 'GET',
    uri: 'http://petstore-demo-endpoint.execute-api.com/petstore/pets'
});
try {
    const results = await apig.send(command)
    console.log(results)
} catch (err) {
    console.log("Set up the integration of the 'GET /pets' method of the API
 failed:\n", err)
}
})();
```

   成功的调用返回以下输出：

```
{
    "httpMethod": "GET",
    "passthroughBehavior": "WHEN_NO_MATCH",
    "cacheKeyParameters": [],
    "type": "HTTP",
    "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
```

```
    "cacheNamespace": "ccc333"
}
```

Python

```
import botocore
import boto3
import logging

logger = logging.getLogger()
apig = boto3.client('apigateway')

try:
    result = apig.put_integration(
        restApiId='abcd1234',
        resourceId='aaa111',
        httpMethod='GET',
        type='HTTP',
        integrationHttpMethod='GET',
        uri='http://petstore-demo-endpoint.execute-api.com/petstore/pets'
    )
except botocore.exceptions.ClientError as error:
    logger.exception("Set up the integration of the 'GET /' method of the API
 failed %s.", error)
    raise
attribute=["httpMethod","passthroughBehavior","cacheKeyParameters", "type",
 "uri", "cacheNamespace"]
filtered_result ={key:result[key] for key in attribute}
print(filtered_result)
```

成功的调用返回以下输出：

```
{'httpMethod': 'GET', 'passthroughBehavior': 'WHEN_NO_MATCH',
 'cacheKeyParameters': [], 'type': 'HTTP', 'uri': 'http://petstore-demo-
endpoint.execute-api.com/petstore/pets', 'cacheNamespace': 'ccc333'}
```

AWS CLI

```
aws apigateway put-integration --rest-api-id abcd1234 \
  --resource-id aaa111 --http-method GET --type HTTP \
  --integration-http-method GET \
  --uri 'http://petstore-demo-endpoint.execute-api.com/petstore/pets' \
```

```
  --region us-west-2
```

此命令的输出如下：

```
{
    "type": "HTTP",
    "httpMethod": "GET",
    "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
    "connectionType": "INTERNET",
    "passthroughBehavior": "WHEN_NO_MATCH",
    "timeoutInMillis": 29000,
    "cacheNamespace": "6sxz2j",
    "cacheKeyParameters": []
}
```

9. 以下示例使用 HTTP 端点为 `GET /pets/{petId}` 方法配置集成。HTTP 端点为 `http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}`。在此步骤中，您将路径参数 `petId` 映射到集成端点路径参数 `id`。

   JavaScript v3

```
import {APIGatewayClient,  PutIntegrationCommand } from "@aws-sdk/client-api-
gateway";
(async function (){
const apig = new APIGatewayClient({region:"us-east-1"});
const command = new PutIntegrationCommand({
    restApiId: 'abcd1234',
    resourceId: 'bbb222',
    httpMethod: 'GET',
    type: 'HTTP',
    integrationHttpMethod: 'GET',
    uri: 'http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}'
    requestParameters: {
        "integration.request.path.id": "method.request.path.petId"
     }
});
try {
    const results = await apig.send(command)
    console.log(results)
} catch (err) {
    console.log("Set up the integration of the 'GET /pets/{petId}' method of the
 API failed:\n", err)
```

```
}
})();
```

成功的调用返回以下输出：

```
{
    "httpMethod": "GET",
    "passthroughBehavior": "WHEN_NO_MATCH",
    "cacheKeyParameters": [],
    "type": "HTTP",
    "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}",
    "cacheNamespace": "ddd444",
    "requestParameters": {
       "integration.request.path.id": "method.request.path.petId"
    }
}
```

Python

```
import botocore
import boto3
import logging

logger = logging.getLogger()
apig = boto3.client('apigateway')

try:
    result = apig.put_integration(
        restApiId='ieps9b05sf',
        resourceId='t8zeb4',
        httpMethod='GET',
        type='HTTP',
        integrationHttpMethod='GET',
        uri='http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}',
        requestParameters={
            "integration.request.path.id": "method.request.path.petId"
        }
    )
except botocore.exceptions.ClientError as error:
    logger.exception("Set up the integration of the 'GET /pets/{petId}' method
 of the API failed %s.", error)
    raise
```

```
attribute=["httpMethod","passthroughBehavior","cacheKeyParameters", "type",
 "uri", "cacheNamespace", "requestParameters"]
filtered_result ={key:result[key] for key in attribute}
print(filtered_result)
```

成功的调用返回以下输出：

```
{'httpMethod': 'GET', 'passthroughBehavior': 'WHEN_NO_MATCH',
 'cacheKeyParameters': [], 'type': 'HTTP', 'uri': 'http://petstore-
demo-endpoint.execute-api.com/petstore/pets/{id}', 'cacheNamespace':
 'ddd444', 'requestParameters': {'integration.request.path.id':
 'method.request.path.petId'}}}
```

AWS CLI

```
aws apigateway put-integration --rest-api-id abcd1234 \
  --resource-id bbb222 --http-method GET --type HTTP \
  --integration-http-method GET \
  --uri 'http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}' \
  --request-parameters
 '{"integration.request.path.id":"method.request.path.petId"}' \
  --region us-west-2
```

此命令的输出如下：

```
{
    "type": "HTTP",
    "httpMethod": "GET",
    "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}",
    "connectionType": "INTERNET",
    "requestParameters": {
        "integration.request.path.id": "method.request.path.petId"
    },
    "passthroughBehavior": "WHEN_NO_MATCH",
    "timeoutInMillis": 29000,
    "cacheNamespace": "rjkmth",
    "cacheKeyParameters": []
}
```

10. 以下示例添加 `GET /pets` 集成的集成响应：

JavaScript v3

```
import {APIGatewayClient,  PutIntegrationResponseCommand } from "@aws-sdk/
client-api-gateway";
(async function (){
const apig = new APIGatewayClient({region:"us-east-1"});
const command = new PutIntegrationResponseCommand({
    restApiId: 'abcd1234',
    resourceId: 'aaa111',
    httpMethod: 'GET',
    statusCode: '200',
    selectionPattern: ''
});
try {
    const results = await apig.send(command)
    console.log(results)
} catch (err) {
    console.log("The 'GET /pets' method integration response setup failed:\n",
 err)
}
})();
```

成功的调用返回以下输出：

```
{
    "selectionPattern": "",
    "statusCode": "200"
}
```

Python

```
import botocore
import boto3
import logging

logger = logging.getLogger()
apig = boto3.client('apigateway')

try:
    result = apig.put_integration_response(
        restApiId='abcd1234',
```

```
            resourceId='aaa111',
            httpMethod='GET',
            statusCode='200',
            selectionPattern='',
        )
    except botocore.exceptions.ClientError as error:
        logger.exception("Set up the integration response of the 'GET /pets' method
     of the API failed: %s", error)
        raise
    attribute=["selectionPattern","statusCode"]
    filtered_result ={key:result[key] for key in attribute}
    print(filtered_result)
```

成功的调用返回以下输出：

```
{'selectionPattern': "", 'statusCode': '200'}
```

AWS CLI

```
aws apigateway put-integration-response --rest-api-id abcd1234 \
  --resource-id aaa111 --http-method GET \
  --status-code 200 --selection-pattern ""  \
  --region us-west-2
```

此命令的输出如下：

```
{
    "statusCode": "200",
    "selectionPattern": ""
}
```

11. 以下示例添加 GET `/pets/{petId}` 集成的集成响应：

JavaScript v3

```
import {APIGatewayClient,  PutIntegrationResponseCommand } from "@aws-sdk/
client-api-gateway";
(async function (){
const apig = new APIGatewayClient({region:"us-east-1"});
const command = new PutIntegrationResponseCommand({
    restApiId: 'abcd1234',
```

```
    resourceId: 'bbb222',
    httpMethod: 'GET',
    statusCode: '200',
    selectionPattern: ''
});
try {
    const results = await apig.send(command)
    console.log(results)
} catch (err) {
    console.log("The 'GET /pets/{petId}' method integration response setup
 failed:\n", err)
}
})();
```

成功的调用返回以下输出：

```
{
    "selectionPattern": "",
    "statusCode": "200"
}
```

Python

```
import botocore
import boto3
import logging

logger = logging.getLogger()
apig = boto3.client('apigateway')

try:
    result = apig.put_integration_response(
        restApiId='abcd1234',
        resourceId='bbb222',
        httpMethod='GET',
        statusCode='200',
        selectionPattern='',
    )
except botocore.exceptions.ClientError as error:
    logger.exception("Set up the integration response of the 'GET /pets/{petId}'
 method of the API failed: %s", error)
    raise
```

```
attribute=["selectionPattern","statusCode"]
filtered_result ={key:result[key] for key in attribute}
print(filtered_result)
```

成功的调用返回以下输出：

```
{'selectionPattern': "", 'statusCode': '200'}
```

AWS CLI

```
aws apigateway put-integration-response --rest-api-id abcd1234 \
  --resource-id bbb222 --http-method GET
  --status-code 200 --selection-pattern ""
  --region us-west-2
```

此命令的输出如下：

```
{
    "statusCode": "200",
    "selectionPattern": ""
}
```

创建集成响应后，API 可以在 PetStore 网站上查询所提供的宠物，并查看指定标识符的单个宠物。在客户能够调用 API 之前，您必须先部署该 API。我们建议您在部署 API 之前先对其进行测试。

12. 以下示例测试 `GET /pets` 方法：

JavaScript v3

```
import {APIGatewayClient,  TestInvokeMethodCommand } from "@aws-sdk/client-api-gateway";
(async function (){
const apig = new APIGatewayClient({region:"us-east-1"});
const command = new TestInvokeMethodCommand({
    restApiId: 'abcd1234',
    resourceId: 'aaa111',
    httpMethod: 'GET',
    pathWithQueryString: '/',
});
```

```
try {
    const results = await apig.send(command)
    console.log(results)
} catch (err) {
    console.log("The test on 'GET /pets' method failed:\n", err)
}
})();
```

Python

```
import botocore
import boto3
import logging

logger = logging.getLogger()
apig = boto3.client('apigateway')

try:
    result = apig.test_invoke_method(
        restApiId='abcd1234',
        resourceId='aaa111',
        httpMethod='GET',
        pathWithQueryString='/',
    )
except botocore.exceptions.ClientError as error:
    logger.exception("Test invoke method on 'GET /pets' failed: %s", error)
    raise
print(result)
```

AWS CLI

```
aws apigateway test-invoke-method --rest-api-id abcd1234 /
  --resource-id aaa111 /
  --http-method GET /
  --path-with-query-string '/'
```

13. 以下示例以 `petId` 为 3 来测试 `GET /pets/{petId}` 方法：

JavaScript v3

```
import {APIGatewayClient,  TestInvokeMethodCommand } from "@aws-sdk/client-api-
gateway";
```

```
(async function (){
const apig = new APIGatewayClient({region:"us-east-1"});
const command = new TestInvokeMethodCommand({
    restApiId: 'abcd1234',
    resourceId: 'bbb222',
    httpMethod: 'GET',
    pathWithQueryString: '/pets/3',
});
try {
    const results = await apig.send(command)
    console.log(results)
} catch (err) {
    console.log("The test on 'GET /pets/{petId}' method failed:\n", err)
}
})();
```

Python

```
import botocore
import boto3
import logging

logger = logging.getLogger()
apig = boto3.client('apigateway')

try:
    result = apig.test_invoke_method(
        restApiId='abcd1234',
        resourceId='bbb222',
        httpMethod='GET',
        pathWithQueryString='/pets/3',
    )
except botocore.exceptions.ClientError as error:
    logger.exception("Test invoke method on 'GET /pets/{petId}' failed: %s",
 error)
    raise
print(result)
```

AWS CLI

```
aws apigateway test-invoke-method --rest-api-id abcd1234 /
  --resource-id bbb222 /
  --http-method GET /
```

```
  --path-with-query-string '/pets/3'
```

成功测试 API 后，您可以将其部署到阶段。

14. 以下示例将 API 部署到名为 `test` 的阶段。将 API 部署到阶段时，API 调用方可以调用 API。

    JavaScript v3

    ```
    import {APIGatewayClient,  CreateDeploymentCommand } from "@aws-sdk/client-api-gateway";
    (async function (){
    const apig = new APIGatewayClient({region:"us-east-1"});
    const command = new CreateDeploymentCommand({
        restApiId: 'abcd1234',
        stageName: 'test',
        stageDescription: 'test deployment'
    });
    try {
        const results = await apig.send(command)
        console.log("Deploying API succeeded\n", results)
    } catch (err) {
        console.log("Deploying API failed:\n", err)
    }
    })();
    ```

    Python

    ```
    import botocore
    import boto3
    import logging

    logger = logging.getLogger()
    apig = boto3.client('apigateway')

    try:
        result = apig.create_deployment(
            restApiId='ieps9b05sf',
            stageName='test',
            stageDescription='my test stage',
        )
    except botocore.exceptions.ClientError as error:
        logger.exception("Error deploying stage  %s.", error)
    ```

```
    raise
print('Deploying API succeeded')
print(result)
```

AWS CLI

```
aws apigateway create-deployment --rest-api-id abcd1234 \
  --region us-west-2 \
  --stage-name test \
  --stage-description 'Test stage' \
  --description 'First deployment'
```

此命令的输出如下：

```
{
    "id": "ab1c1d",
    "description": "First deployment",
    "createdDate": "2022-12-15T08:44:13-08:00"
}
```

现在，客户可以调用您的 API 了。您可以通过在浏览器中输入 `https://abcd1234.execute-api.us-west-2.amazonaws.com/test/pets` URL 并将 `abcd1234` 替换为 API 的标识符，测试此 API。

有关如何使用 AWS SDK 或 AWS CLI 创建或更新 API 的更多示例，请参阅 Actions for API Gateway using AWS SDKs。

自动设置 API

在创建 API 时，您不必按部就班地创建，而是可以通过使用 OpenAPI、AWS CloudFormation 或 Terraform 来自动创建和清理 AWS 资源。

OpenAPI 3.0 定义

您可以将 OpenAPI 定义导入到 API Gateway 中。有关更多信息，请参阅 the section called “OpenAPI”。

```
{
  "openapi" : "3.0.1",
  "info" : {
```

```
    "title" : "Simple PetStore (OpenAPI)",
    "description" : "Demo API created using OpenAPI",
    "version" : "2024-05-24T20:39:34Z"
  },
  "servers" : [ {
    "url" : "{basePath}",
    "variables" : {
      "basePath" : {
        "default" : "Prod"
      }
    }
  } ],
  "paths" : {
    "/pets" : {
      "get" : {
        "responses" : {
          "200" : {
            "description" : "200 response",
            "content" : { }
          }
        },
        "x-amazon-apigateway-integration" : {
          "type" : "http",
          "httpMethod" : "GET",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
          "responses" : {
            "default" : {
              "statusCode" : "200"
            }
          },
          "passthroughBehavior" : "when_no_match",
          "timeoutInMillis" : 29000
        }
      }
    },
    "/pets/{petId}" : {
      "get" : {
        "parameters" : [ {
          "name" : "petId",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
```

```
        } ],
        "responses" : {
          "200" : {
            "description" : "200 response",
            "content" : { }
          }
        },
        "x-amazon-apigateway-integration" : {
          "type" : "http",
          "httpMethod" : "GET",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}",
          "responses" : {
            "default" : {
              "statusCode" : "200"
            }
          },
          "requestParameters" : {
            "integration.request.path.id" : "method.request.path.petId"
          },
          "passthroughBehavior" : "when_no_match",
          "timeoutInMillis" : 29000
        }
      }
    }
  },
  "components" : { }
}
```

AWS CloudFormation 模板

要部署您的 AWS CloudFormation 模板，请参阅在 AWS CloudFormation 控制台上创建堆栈。

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  Api:
    Type: 'AWS::ApiGateway::RestApi'
    Properties:
      Name: Simple PetStore (AWS CloudFormation)
  PetsResource:
    Type: 'AWS::ApiGateway::Resource'
    Properties:
      RestApiId: !Ref Api
      ParentId: !GetAtt Api.RootResourceId
      PathPart: 'pets'
```

```
  PetIdResource:
    Type: 'AWS::ApiGateway::Resource'
    Properties:
      RestApiId: !Ref Api
      ParentId: !Ref PetsResource
      PathPart: '{petId}'
  PetsMethodGet:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref PetsResource
      HttpMethod: GET
      AuthorizationType: NONE
      Integration:
        Type: HTTP
        IntegrationHttpMethod: GET
        Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets/
        IntegrationResponses:
          - StatusCode: '200'
      MethodResponses:
        - StatusCode: '200'
  PetIdMethodGet:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref PetIdResource
      HttpMethod: GET
      AuthorizationType: NONE
      RequestParameters:
        method.request.path.petId: true
      Integration:
        Type: HTTP
        IntegrationHttpMethod: GET
        Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets/{id}
        RequestParameters:
          integration.request.path.id: method.request.path.petId
        IntegrationResponses:
          - StatusCode: '200'
      MethodResponses:
        - StatusCode: '200'
  ApiDeployment:
    Type: 'AWS::ApiGateway::Deployment'
    DependsOn:
      - PetsMethodGet
```

```
    Properties:
      RestApiId: !Ref Api
      StageName: Prod
Outputs:
  ApiRootUrl:
    Description: Root Url of the API
    Value: !Sub 'https://${Api}.execute-api.${AWS::Region}.amazonaws.com/Prod'
```

Terraform 配置

有关 Terraform 的信息，请参阅 Terraform。

```
provider "aws" {
  region = "us-east-1" # Update with your desired region
}
resource "aws_api_gateway_rest_api" "Api" {
  name        = "Simple PetStore (Terraform)"
  description = "Demo API created using Terraform"
}
resource "aws_api_gateway_resource" "petsResource"{
    rest_api_id = aws_api_gateway_rest_api.Api.id
    parent_id = aws_api_gateway_rest_api.Api.root_resource_id
    path_part = "pets"
}
resource "aws_api_gateway_resource" "petIdResource"{
    rest_api_id = aws_api_gateway_rest_api.Api.id
    parent_id = aws_api_gateway_resource.petsResource.id
    path_part = "{petId}"
}
resource "aws_api_gateway_method" "petsMethodGet" {
  rest_api_id   = aws_api_gateway_rest_api.Api.id
  resource_id   = aws_api_gateway_resource.petsResource.id
  http_method   = "GET"
  authorization = "NONE"
}


resource "aws_api_gateway_method_response" "petsMethodResponseGet" {
    rest_api_id = aws_api_gateway_rest_api.Api.id
    resource_id = aws_api_gateway_resource.petsResource.id
    http_method = aws_api_gateway_method.petsMethodGet.http_method
    status_code ="200"
}
```

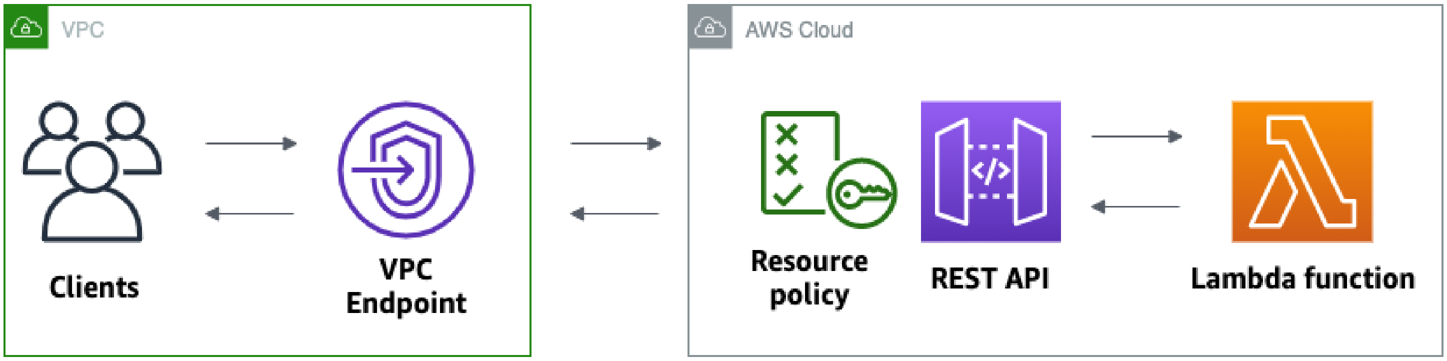
```
resource "aws_api_gateway_integration" "petsIntegration" {
  rest_api_id = aws_api_gateway_rest_api.Api.id
  resource_id = aws_api_gateway_resource.petsResource.id
  http_method = aws_api_gateway_method.petsMethodGet.http_method
  type        = "HTTP"

  uri                     = "http://petstore-demo-endpoint.execute-api.com/petstore/
pets"
  integration_http_method = "GET"
  depends_on              = [aws_api_gateway_method.petsMethodGet]
}

resource "aws_api_gateway_integration_response" "petsIntegrationResponse" {
    rest_api_id = aws_api_gateway_rest_api.Api.id
    resource_id = aws_api_gateway_resource.petsResource.id
    http_method = aws_api_gateway_method.petsMethodGet.http_method
    status_code = aws_api_gateway_method_response.petsMethodResponseGet.status_code
}

resource "aws_api_gateway_method" "petIdMethodGet" {
    rest_api_id   = aws_api_gateway_rest_api.Api.id
    resource_id   = aws_api_gateway_resource.petIdResource.id
    http_method   = "GET"
    authorization = "NONE"
    request_parameters = {"method.request.path.petId" = true}
}

resource "aws_api_gateway_method_response" "petIdMethodResponseGet" {
    rest_api_id = aws_api_gateway_rest_api.Api.id
    resource_id = aws_api_gateway_resource.petIdResource.id
    http_method = aws_api_gateway_method.petIdMethodGet.http_method
    status_code ="200"
}


resource "aws_api_gateway_integration" "petIdIntegration" {
    rest_api_id = aws_api_gateway_rest_api.Api.id
    resource_id = aws_api_gateway_resource.petIdResource.id
    http_method = aws_api_gateway_method.petIdMethodGet.http_method
    type        = "HTTP"
    uri                     = "http://petstore-demo-endpoint.execute-api.com/petstore/
pets/{id}"
    integration_http_method = "GET"
    request_parameters = {"integration.request.path.id" = "method.request.path.petId"}
```

```
    depends_on              = [aws_api_gateway_method.petIdMethodGet]
}

resource "aws_api_gateway_integration_response" "petIdIntegrationResponse" {
    rest_api_id = aws_api_gateway_rest_api.Api.id
    resource_id = aws_api_gateway_resource.petIdResource.id
    http_method = aws_api_gateway_method.petIdMethodGet.http_method
    status_code = aws_api_gateway_method_response.petIdMethodResponseGet.status_code
}


resource "aws_api_gateway_deployment" "Deployment" {
  rest_api_id = aws_api_gateway_rest_api.Api.id
  depends_on  =
 [aws_api_gateway_integration.petsIntegration,aws_api_gateway_integration.petIdIntegration ]
}
resource "aws_api_gateway_stage" "Stage" {
  stage_name    = "Prod"
  rest_api_id   = aws_api_gateway_rest_api.Api.id
  deployment_id = aws_api_gateway_deployment.Deployment.id
}
```

教程：构建私有 REST API

在本教程中，您将创建一个私有 REST API。客户端只能从您的 Amazon VPC 内访问 API。API 与公共互联网隔离，这是一项常见的安全要求。

完成本教程需要大约 30 分钟。首先，您要使用 AWS CloudFormation 模板创建 Amazon VPC、VPC 端点、AWS Lambda 函数，然后启动用于测试 API 的 Amazon EC2 实例。接下来，您可以使用AWS Management Console创建私有 API 并附加仅允许从 VPC 端点访问的资源策略。最后，测试您的 API。



要完成本教程，您需要一个AWS账户以及一位具有控制台访问权限的 AWS Identity and Access Management 用户。有关更多信息，请参阅 先决条件。

在本教程中，您将使用 AWS Management Console。如需创建此 API 和所有相关资源的 AWS CloudFormation 模板，请参阅 template.yaml。

主题

- 步骤 1：创建依赖关系
- 步骤 2：创建私有密钥
- 步骤 3：创建方法和集成
- 步骤 4：附加资源策略
- 步骤 5：部署您的 API
- 步骤 6：验证您的 API 是否不可公开访问
- 步骤 7：连接到 VPC 中的实例并调用 API
- 步骤 8：清除
- 接下来的步骤：通过 实现自动化AWS CloudFormation

步骤 1：创建依赖关系

下载并解压缩此 AWS CloudFormation 模板。您可以使用模板为私有 API 创建所有依赖项，包括 Amazon VPC、VPC 端点和作为 API 后端的 Lambda 函数。您稍后创建私有 API。

创建 AWS CloudFormation 堆栈

1. 打开 AWS CloudFormation 控制台，地址：https://console.aws.amazon.com/cloudformation。
2. 选择创建堆栈，然后选择使用新资源(标准)。
3. 对于指定模板，选择上传模板文件。
4. 选择您下载的模板。
5. 选择下一步。
6. 对于堆栈名称，输入 **`private-api-tutorial`**，然后选择下一步。
7. 对于配置堆栈选项，请选择下一步。
8. 对于功能，请确认 AWS CloudFormation 可以在您的账户中创建 IAM 资源。
9. 选择提交。

AWS CloudFormation 为 API 配置依赖项，这可能需要几分钟的时间。当 AWS CloudFormation 堆栈的状态为 CREATE_COMPLETE 时，请选择输出。记下您的 VPC 端点 ID。在本教程的后续步骤中，您需要使用该信息。

步骤 2：创建私有密钥

您可以创建一个私有 API，以便只允许 VPC 中的客户端可以访问。

要创建私有 API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择创建 API，然后为 REST API 选择构建。
3. 对于 API Name (API 名称)，请输入 **`private-api-tutorial`**。
4. 对于 API 端点类型，选择私有。
5. 对于 VPC 端点 ID，请输入 AWS CloudFormation 堆栈的输出中的 VPC 端点 ID。
6. 选择创建 API。

步骤 3：创建方法和集成

您可以创建 `GET` 方法和 Lambda 集成来处理对 API 的 `GET` 请求。当客户端调用 API 时，API Gateway 会将请求发送到您在步骤 1 中创建的 Lambda 函数，然后向客户端返回响应。

要创建方法和集成

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 选择 / 资源，然后选择创建方法。
4. 对于方法类型，选择 `GET`。
5. 对于集成类型，选择 Lambda 函数。
6. 打开 Lambda 代理集成。通过 Lambda 代理集成，API Gateway 将具有定义结构的事件发送到 Lambda，并将响应从您的 Lambda 函数转换为 HTTP 响应。
7. 对于 Lambda 函数，请选择您在步骤 1 中使用 AWS CloudFormation 模板创建的函数。函数的名称以 **`private-api-tutorial`** 为开头。
8. 选择创建方法。

步骤 4：附加资源策略

您将资源策略附加到 API，该策略允许客户端仅通过 VPC 端点调用您的 API。要进一步限制对 API 的访问，您还可以为 VPC 端点配置 VPC 端点策略，但这对于本教程并不是必需的。

要附加资源策略

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 选择资源策略，然后选择创建策略。
4. 输入以下策略。使用来自您 AWS CloudFormation 堆栈的 输出 中的 VPC 端点 ID 替换 *vpceID*。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "execute-api:/*",
            "Condition": {
                "StringNotEquals": {
                    "aws:sourceVpce": "vpceID"
                }
            }
        },
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "execute-api:/*"
        }
    ]
}
```

5. 选择 Save changes（保存更改）。

步骤 5：部署您的 API

接下来，您部署 API 以使其可供 Amazon VPC 中的客户端使用。

要部署 API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 选择部署 API。
4. 对于阶段，选择新建阶段。
5. 对于阶段名称，输入 **`test`**。
6. （可选）对于描述，输入描述。
7. 选择 Deploy (部署)。

现在您已经准备好测试 API 了。

步骤 6：验证您的 API 是否不可公开访问

用 `curl` 于验证您是否无法从 Amazon VPC 之外调用 API。

要测试您的 API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 在主导航窗格中，选择阶段，然后选择测试阶段。
4. 在阶段详细信息下，选择复制图标以复制您 API 的调用 URL。URL 类似于 `https://abcdef123.execute-api.us-west-2.amazonaws.com/test`。您在步骤 1 中创建的 VPC 端点启用了私有 DNS，因此您可以使用提供的 URL 调用 API。
5. 使用 curl 尝试从 VPC 外部调用 API。

```
curl https://abcdef123.execute-api.us-west-2.amazonaws.com/test
```

Curl 表示您的 API 的端点无法解析。如果您收到其他响应，请返回步骤 2，并确保为 API 的端点类型选择私有。

```
curl: (6) Could not resolve host: abcdef123.execute-api.us-west-2.amazonaws.com/
test
```

接下来，您连接到 VPC 中的 Amazon EC2 实例以调用 API。

步骤 7：连接到 VPC 中的实例并调用 API

接下来，您可以从 Amazon VPC 内测试 API。要访问您的私有 API，您需要连接到 VPC 中的 Amazon EC2 实例，然后使用 curl 调用 API。您可以使用 Systems Manager 会话管理器在浏览器中连接到实例。

要测试您的 API

1. 通过以下网址打开 Amazon EC2 控制台：https://console.aws.amazon.com/ec2/。
2. 选择实例。
3. 选择您在步骤 1 中使用 AWS CloudFormation 模板创建的名为 private-api-tutorial 的实例。
4. 选择连接，然后选择会话管理器。
5. 选择连接可启动与实例的基于浏览器的会话。
6. 在会话管理器会话中，使用 curl 调用 API。您可以调用 API，因为您在 Amazon VPC 中使用的是实例。

   ```
   curl https://abcdef123.execute-api.us-west-2.amazonaws.com/test
   ```

   验证您得到了回复 `Hello from Lambda!`。

Session ID: user-[redacted]

Instance ID: i-[redacted]

```
sh-4.2$ curl https://[redacted].execute-api.us-west-2.amazonaws.com/prod
"Hello from Lambda!"sh-4.2$
```

您成功创建了只能从 Amazon VPC 内访问的 API，然后验证其是否有效。

步骤 8：清除

为避免不必要的成本，请删除作为本教程的一部分而创建的资源。以下步骤将删除您的 REST API 和 AWS CloudFormation 堆栈。

要删除 REST API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在 API 页面上，选择一个 API。选择 API 操作，选择删除 API，然后确认您的选择。

删除 AWS CloudFormation 堆栈

1. 打开 AWS CloudFormation 控制台，地址：https://console.aws.amazon.com/cloudformation。
2. 选择您的 AWS CloudFormation 堆栈。
3. 选择删除，然后确认您的选择。

接下来的步骤：通过 实现自动化AWS CloudFormation

您可以自动创建和清理本教程中涉及的所有AWS资源。如需完整的示例 AWS CloudFormation 模板，请参阅 template.yaml。

Amazon API Gateway HTTP API 教程

以下教程将提供有助于您了解 API Gateway HTTP API 的实践练习。

主题

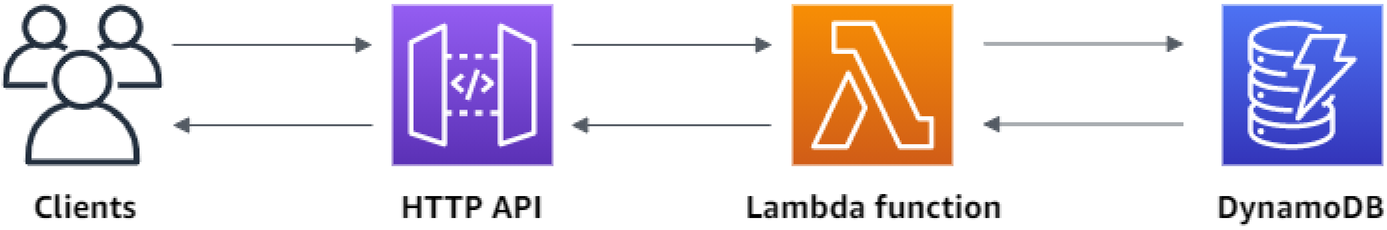
- 教程：使用 Lambda 和 DynamoDB 构建 CRUD API
- 教程：构建具有私有集成到 Amazon ECS 服务的 HTTP API

教程：使用 Lambda 和 DynamoDB 构建 CRUD API

在本教程中，您将创建一个无服务器 API，用于创建、读取、更新和删除 DynamoDB 表中的项目。DynamoDB 是一项完全托管的 NoSQL 数据库服务，提供快速而可预测的性能，能够实现无缝扩展。完成本教程大约需要 30 分钟，您可以在AWS免费套餐中完成。

首先，使用 DynamoDB 控制台创建 DynamoDB 表。然后，使用 AWS Lambda 控制台创建 Lambda 函数。接下来，使用 API Gateway 控制台创建 HTTP API。最后，测试您的 API。

当您调用 HTTP API 时，API Gateway 会将请求路由到您的 Lambda 函数。Lambda 函数与 DynamoDB 进行交互，并将响应返回 API Gateway。然后 API Gateway 会向您返回响应。



要完成本练习，您需要一个AWS账户以及一位具有控制台访问权限的 AWS Identity and Access Management 用户。有关更多信息，请参阅 先决条件。

在本教程中，您将使用 AWS Management Console。如需创建此 API 和所有相关资源的 AWS SAM 模板，请参阅 `template.yaml`。

主题

- 步骤 1：创建 DynamoDB 表
- 步骤 2：创建 Lambda 函数
- 步骤 3：创建 HTTP API
- 步骤 4：创建路由
- 步骤 5：创建集成
- 步骤 6：将集成附加到路由
- 步骤 7：测试您的 API
- 步骤 8：清除
- 下一步：使用 AWS SAM 或 AWS CloudFormation 实现自动化

步骤 1：创建 DynamoDB 表

您可以使用 DynamoDB 表为您的 API 存储数据。

每个项目都有一个唯一的 ID，我们将其用作表的分区键。

创建 DynamoDB 表

1. 从 https://console.aws.amazon.com/dynamodb/ 打开 DynamoDB 控制台。
2. 选择创建表。
3. 对于 表名称，输入 **`http-crud-tutorial-items`**。
4. 对于分区键，输入 **`id`**。
5. 选择创建表。

步骤 2：创建 Lambda 函数

创建 Lambda 函数以用作您的 API 后端。此 Lambda 函数从 DynamoDB 中创建、读取、更新和删除项目。该函数使用 API Gateway 中的事件来决定如何与 DynamoDB 交互。为简单起见，本教程使用单个 Lambda 函数。作为最佳实践，您应该为每个路由创建单独的函数。

创建 Lambda 函数

1. 通过以下网址登录 Lambda 控制台：https://console.aws.amazon.com/lambda。
2. 选择创建函数。
3. 对于函数名称，请输入 **`http-crud-tutorial-function`**。
4. 在运行时中，选择受支持的最新 Node.js 或 Python 运行时。
5. 在权限下，选择更改默认执行角色。
6. 选择从 AWS 策略模板中创建新角色。
7. 对于角色名称，输入 **`http-crud-tutorial-role`**。
8. 对于策略模板，选择 **`Simple microservice permissions`**。此策略授予 Lambda 函数与 DynamoDB 进行交互的权限。

> **Note**
>
> 为简单起见，本教程使用托管策略。作为最佳实践，您应创建自己的 IAM 策略以授予所需的最低权限。

9. 选择创建函数。
10. 在控制台的代码编辑器中打开 Lambda 函数，并将其内容替换为以下代码。选择部署以更新您的函数。

Node.js

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  DynamoDBDocumentClient,
  ScanCommand,
  PutCommand,
  GetCommand,
  DeleteCommand,
} from "@aws-sdk/lib-dynamodb";
```

```
const client = new DynamoDBClient({});

const dynamo = DynamoDBDocumentClient.from(client);

const tableName = "http-crud-tutorial-items";

export const handler = async (event, context) => {
  let body;
  let statusCode = 200;
  const headers = {
    "Content-Type": "application/json",
  };

  try {
    switch (event.routeKey) {
      case "DELETE /items/{id}":
        await dynamo.send(
          new DeleteCommand({
            TableName: tableName,
            Key: {
              id: event.pathParameters.id,
            },
          })
        );
        body = `Deleted item ${event.pathParameters.id}`;
        break;
      case "GET /items/{id}":
        body = await dynamo.send(
          new GetCommand({
            TableName: tableName,
            Key: {
              id: event.pathParameters.id,
            },
          })
        );
        body = body.Item;
        break;
      case "GET /items":
        body = await dynamo.send(
          new ScanCommand({ TableName: tableName })
        );
        body = body.Items;
        break;
      case "PUT /items":
```

```
        let requestJSON = JSON.parse(event.body);
        await dynamo.send(
          new PutCommand({
            TableName: tableName,
            Item: {
              id: requestJSON.id,
              price: requestJSON.price,
              name: requestJSON.name,
            },
          })
        );
        body = `Put item ${requestJSON.id}`;
        break;
      default:
        throw new Error(`Unsupported route: "${event.routeKey}"`);
    }
  } catch (err) {
    statusCode = 400;
    body = err.message;
  } finally {
    body = JSON.stringify(body);
  }

  return {
    statusCode,
    body,
    headers,
  };
};
```

Python

```
import json
import boto3
from decimal import Decimal

client = boto3.client('dynamodb')
dynamodb = boto3.resource("dynamodb")
table = dynamodb.Table('http-crud-tutorial-items')
tableName = 'http-crud-tutorial-items'


def lambda_handler(event, context):
```

```
    print(event)
    body = {}
    statusCode = 200
    headers = {
        "Content-Type": "application/json"
    }

    try:
        if event['routeKey'] == "DELETE /items/{id}":
            table.delete_item(
                Key={'id': event['pathParameters']['id']})
            body = 'Deleted item ' + event['pathParameters']['id']
        elif event['routeKey'] == "GET /items/{id}":
            body = table.get_item(
                Key={'id': event['pathParameters']['id']})
            body = body["Item"]
            responseBody = [
                {'price': float(body['price']), 'id': body['id'], 'name': 
body['name']}]
            body = responseBody
        elif event['routeKey'] == "GET /items":
            body = table.scan()
            body = body["Items"]
            print("ITEMS----")
            print(body)
            responseBody = []
            for items in body:
                responseItems = [
                    {'price': float(items['price']), 'id': items['id'], 'name': 
items['name']}]
                responseBody.append(responseItems)
            body = responseBody
        elif event['routeKey'] == "PUT /items":
            requestJSON = json.loads(event['body'])
            table.put_item(
                Item={
                    'id': requestJSON['id'],
                    'price': Decimal(str(requestJSON['price'])),
                    'name': requestJSON['name']
                })
            body = 'Put item ' + requestJSON['id']
    except KeyError:
        statusCode = 400
        body = 'Unsupported route: ' + event['routeKey']
```

```
    body = json.dumps(body)
    res = {
        "statusCode": statusCode,
        "headers": {
            "Content-Type": "application/json"
        },
        "body": body
    }
    return res
```

步骤 3：创建 HTTP API

HTTP API 为您的 Lambda 函数提供了 HTTP 端点。在此步骤中，您将创建空 API。在以下步骤中，您将配置路由和集成以连接 API 和 Lambda 函数。

要创建 HTTP API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择创建 API，然后为 HTTP API 选择构建。
3. 对于 API 名称，请输入 **`http-crud-tutorial-api`**。
4. 选择下一步。
5. 对于配置路由，选择下一步以跳过路由创建。稍后再创建路由。
6. 查看 API Gateway 为您创建的阶段，然后选择下一步。
7. 选择创建。

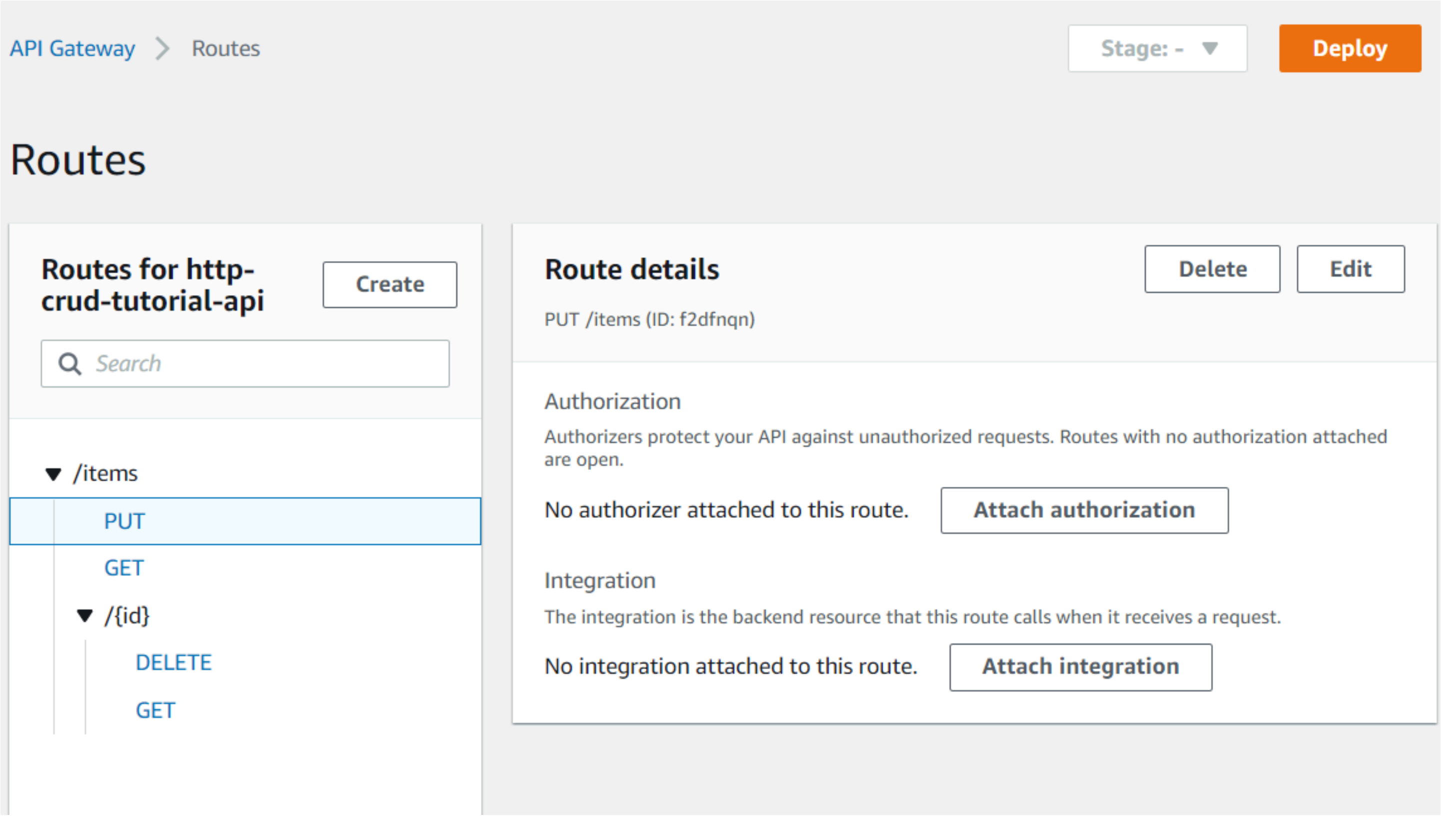
步骤 4：创建路由

路由是将传入的 API 请求发送到后端资源的一种方式。路由包含两部分：HTTP 方法和资源路径，例如 `GET /items`。对于此示例 API，我们创建了四个路由：

- `GET /items/{id}`
- `GET /items`
- `PUT /items`
- `DELETE /items/{id}`

要创建路由

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 选择路由。
4. 选择创建。
5. 对于方法，选择 **GET**。
6. 对于路径，请输入 **`/items/{id}`**。路径结尾的 `{id}` 是 API Gateway 在客户端提出请求时从请求路径中接收的路径参数。
7. 选择创建。
8. 对 `GET /items`、`DELETE /items/{id}` 和 `PUT /items` 重复步骤 4-7。



PUT /items (ID: f2dfnqn)

Authorization

Authorizers protect your API against unauthorized requests. Routes with no authorization attached are open.

No authorizer attached to this route.

Integration

The integration is the backend resource that this route calls when it receives a request.

No integration attached to this route.

步骤 5：创建集成

您可以创建集成以将路由连接到后端资源。对于此示例 API，您可以创建一个用于所有路由的 Lambda 集成。

要创建集成

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 选择集成。
4. 选择管理集成，然后选择创建。
5. 跳过将此集成附加到一个路由。在后续步骤中再完成此操作。
6. 对于集成类型，选择 Lambda 函数。
7. 对于 Lambda 函数，输入 **`http-crud-tutorial-function`**。
8. 选择创建。

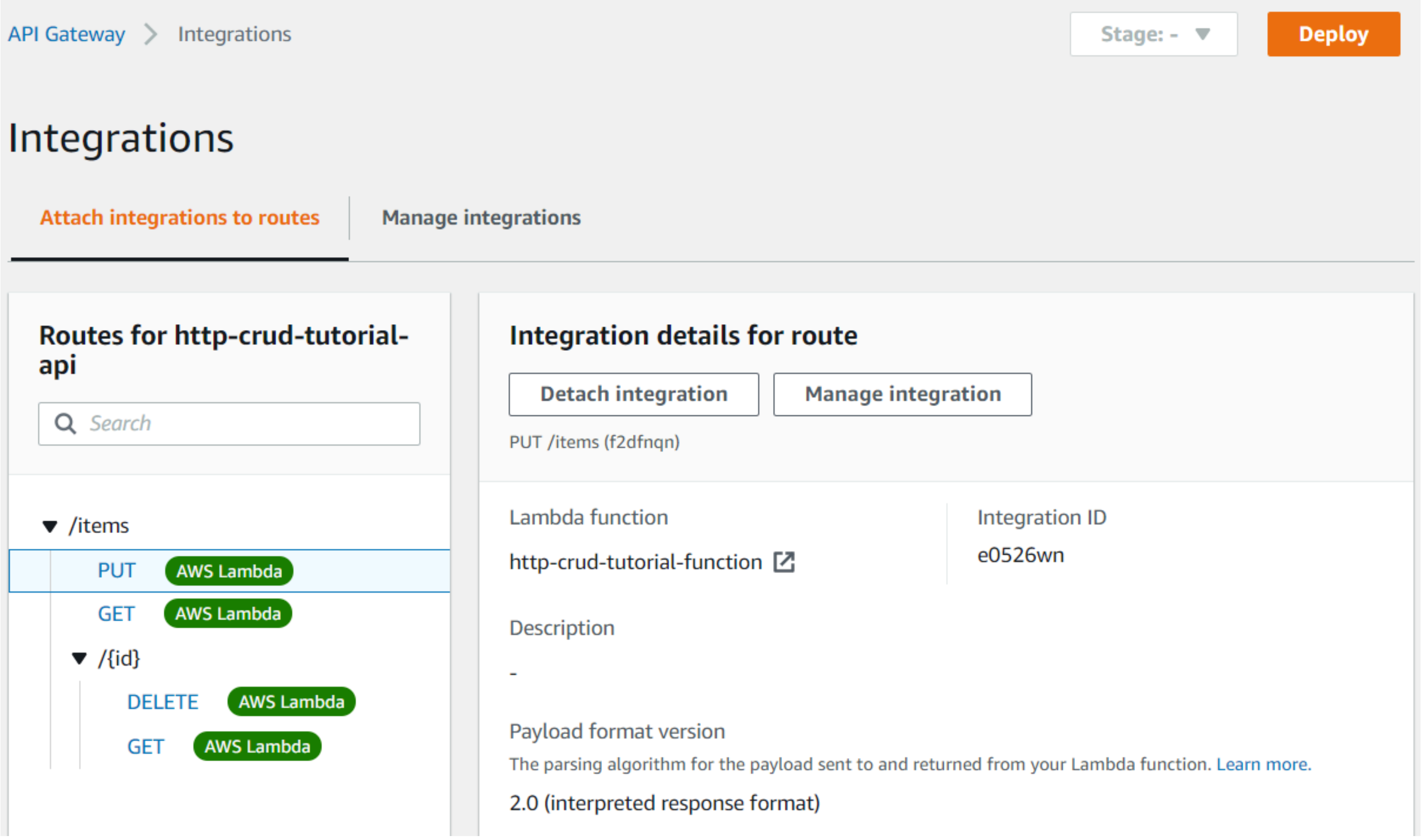
步骤 6：将集成附加到路由

对于此示例 API，您对所有路由都使用相同的 Lambda 集成。将集成附加到所有的 API 路由后，当客户端调用您的任何路由时，您的 Lambda 函数将被调用。

要将集成附加到路由

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 选择集成。
4. 选择路由。
5. 在选择现有集成下，选择 **`http-crud-tutorial-function`**。
6. 选择附加集成。
7. 对所有路由重复步骤 4-6。

所有路由均显示附加了 AWS Lambda 集成。



PUT /items (f2dfnqn)

Lambda function: http-crud-tutorial-function

Integration ID: e0526wn

Description: -

Payload format version: The parsing algorithm for the payload sent to and returned from your Lambda function. Learn more.

2.0 (interpreted response format)

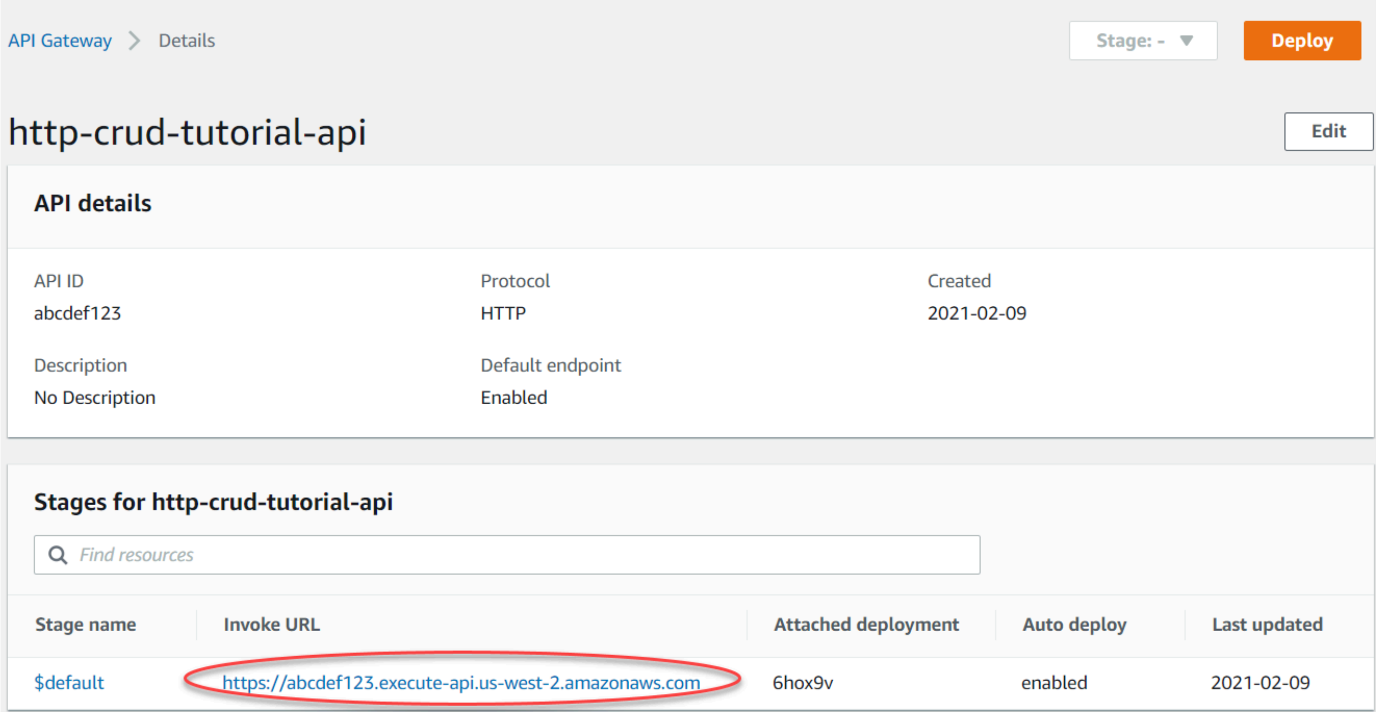
现在您已经有了一个包含路由和集成的 HTTP API，您可以测试 API 了。

步骤 7：测试您的 API

为了确保您的 API 正常工作，您可以使用 curl。

要获取调用 API 的 URL

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 请记下您的 API 的调用 URL。它显示在详细信息页面上的调用 URL 下。



http-crud-tutorial-api

API details

API ID	Protocol	Created
abcdef123	HTTP	2021-02-09

Description	Default endpoint
No Description	Enabled

Stages for http-crud-tutorial-api

Stage name	Invoke URL	Attached deployment	Auto deploy	Last updated
$default	https://abcdef123.execute-api.us-west-2.amazonaws.com	6hox9v	enabled	2021-02-09

4. 请复制您的 API 的调用 URL。

   完整的 URL 类似于以下所示：`https://abcdef123.execute-api.us-west-2.amazonaws.com`。

要创建或更新项目

- 使用以下命令以创建或更新项目。该命令包括带有项目 ID、价格和名称的请求正文。

```
curl -X "PUT" -H "Content-Type: application/json" -d "{\"id\": \"123\",
 \"price\": 12345, \"name\": \"myitem\"}" https://abcdef123.execute-api.us-
west-2.amazonaws.com/items
```

要获取所有项目

- 使用以下命令列出所有项目。

```
curl https://abcdef123.execute-api.us-west-2.amazonaws.com/items
```

要获取一个项目

- 使用以下命令按 ID 获取项目。

  ```
  curl https://abcdef123.execute-api.us-west-2.amazonaws.com/items/123
  ```

如何删除项目

1. 使用以下命令删除项目。

   ```
   curl -X "DELETE" https://abcdef123.execute-api.us-west-2.amazonaws.com/items/123
   ```

2. 获取所有项目以验证项目已删除。

   ```
   curl https://abcdef123.execute-api.us-west-2.amazonaws.com/items
   ```

步骤 8：清除

为避免不必要的成本，请删除作为本入门练习的一部分而创建的资源。以下步骤将删除 HTTP API、Lambda 函数和相关资源。

删除 DynamoDB 表

1. 从 https://console.aws.amazon.com/dynamodb/ 打开 DynamoDB 控制台。
2. 选择您的表。
3. 选择删除表。
4. 确认您的选择，然后选择删除。

要删除 HTTP API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在 API 页面上，选择一个 API。选择操作，然后选择删除。
3. 选择删除。

要删除 Lambda 函数

1. 通过以下网址登录 Lambda 控制台：https://console.aws.amazon.com/lambda。
2. 在函数页面上，选择一个函数。选择操作，然后选择删除。
3. 选择删除。

要删除 Lambda 函数的日志组

1. 在 Amazon CloudWatch 控制台中，打开日志组页面。
2. 在日志组页面上，选择函数的日志组 (`/aws/lambda/http-crud-tutorial-function`)。选择操作，然后选择删除日志组。
3. 选择删除。

要删除 Lambda 函数的执行角色

1. 打开AWS Identity and Access Management控制台中的角色页面。
2. 选择函数的角色，例如 `http-crud-tutorial-role`。
3. 选择删除角色。
4. 选择是，删除。

下一步：使用 AWS SAM 或 AWS CloudFormation 实现自动化

您可以使用 AWS CloudFormation 或 AWS SAM 自动创建和清理AWS资源。如需本教程中所用的 AWS SAM 示例模板，请参阅 `template.yaml`。

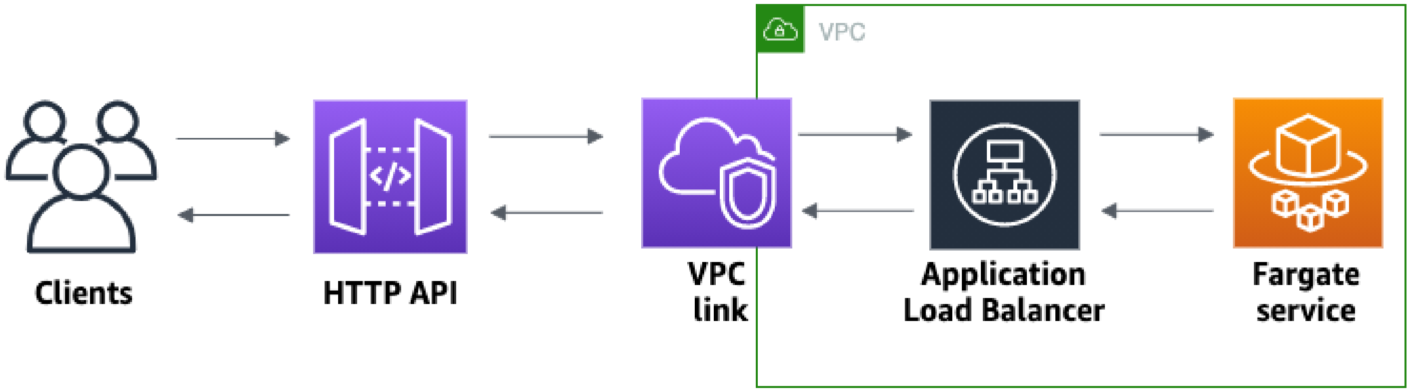
如需示例 AWS CloudFormation 模板，请参阅示例 AWS CloudFormation 模板。

教程：构建具有私有集成到 Amazon ECS 服务的 HTTP API

在本教程中，您将创建一个无服务器 API，该 API 连接到在 Amazon VPC 中运行的 Amazon ECS 服务。Amazon VPC 之外的客户端可以使用 API 访问您的 Amazon ECS 服务。

完成本教程需要大约 1 个小时。首先，您使用 AWS CloudFormation 模板创建 Amazon VPC 和 Amazon ECS 服务。然后，您可以使用 API Gateway 控制台创建 VPC 链接。VPC 链接允许 API 网关访问在您的 Amazon VPC 中运行的 Amazon ECS 服务。接下来，您创建一个 HTTP API，该 API 使用 VPC 链接连接到您的 Amazon ECS 服务。最后，测试您的 API。

当您调用 HTTP API 时，API Gateway 会通过 VPC 链接将请求路由到您的 Amazon ECS 服务，然后从该服务返回响应。



要完成本教程，您需要一个AWS账户以及一位具有控制台访问权限的 AWS Identity and Access Management 用户。有关更多信息，请参阅 先决条件。

在本教程中，您将使用 AWS Management Console。如需创建此 API 和所有相关资源的 AWS CloudFormation 模板，请参阅 template.yaml。

主题

- 步骤 1：创建 Amazon ECS 服务
- 步骤 2：创建 VPC 链接
- 步骤 3：创建 HTTP API
- 步骤 4：创建路由
- 步骤 5：创建集成
- 步骤 6：测试您的 API
- 步骤 7：清除
- 接下来的步骤：通过 实现自动化AWS CloudFormation

步骤 1：创建 Amazon ECS 服务

Amazon ECS 是一项容器管理服务，可让您轻松地在 cluster 上运行、停止和管理 Docker 容器。在本教程中，您将在 Amazon ECS 管理的无服务器基础设施上运行 cluster。

下载并解压此 AWS CloudFormation 模板，这将为服务创建所有依赖项，包括 Amazon VPC。您可以使用模板创建使用 Application Load Balancer 的 Amazon ECS 服务。

创建 AWS CloudFormation 堆栈

1. 打开 AWS CloudFormation 控制台，地址：https://console.aws.amazon.com/cloudformation。
2. 选择创建堆栈，然后选择使用新资源(标准)。
3. 对于指定模板，选择上传模板文件。
4. 选择您下载的模板。
5. 选择下一步。
6. 对于堆栈名称，输入 **`http-api-private-integrations-tutorial`**，然后选择下一步。
7. 对于配置堆栈选项，请选择下一步。
8. 对于功能，请确认 AWS CloudFormation 可以在您的账户中创建 IAM 资源。
9. 选择提交。

AWS CloudFormation 配置 ECS 服务，这可能需要几分钟的时间。当 AWS CloudFormation 堆栈的状态为 CREATE_COMPLETE 时，您就可以继续下一步了。

步骤 2：创建 VPC 链接

VPC 链接允许 API 网关访问 Amazon VPC 中的私有资源。您可以使用 VPC 链接允许客户端通过 HTTP API 访问您的 Amazon ECS 服务。

要创建 VPC 链接

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在主导航窗格上，选择 VPC 链接，然后选择创建。

   您可能需要选择菜单图标才能打开主导航窗格。
3. 对于选择 VPC 链接版本，选择 HTTP API 的 VPC 链接。
4. 对于名称，请输入 **`private-integrations-tutorial`**。
5. 对于 VPC，选择已在步骤 1 中创建的 VPC。名称应该以 PrivateIntegrationsStack 开头。
6. 对于子网，选择 VPC 中的两个私有子网。名字以 `PrivateSubnet` 结尾。
7. 选择创建。

创建 VPC 链接后，API Gateway 会配置弹性网络接口以访问您的 VPC。此过程可能耗时数分钟。同时，您可以创建 API。

步骤 3：创建 HTTP API

HTTP API 为您的 Amazon ECS 服务提供了 HTTP 端点。在此步骤中，您将创建空 API。在步骤 4 和 5 中，您配置路由和集成以连接 API 和 Amazon ECS 服务。

要创建 HTTP API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择创建 API，然后为 HTTP API 选择构建。
3. 对于 API 名称，请输入 **`http-private-integrations-tutorial`**。
4. 选择下一步。
5. 对于配置路由，选择下一步以跳过路由创建。稍后再创建路由。
6. 查看 API Gateway 为您创建的阶段。API Gateway 创建了启用自动部署的 `$default` 阶段，这是本教程的最佳选择。选择下一步。
7. 选择创建。

步骤 4：创建路由

路由是将传入的 API 请求发送到后端资源的一种方式。路由包含两部分：HTTP 方法和资源路径，例如 `GET /items`。对于此示例 API，我们创建了一个路由。

要创建路由

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 选择路由。
4. 选择创建。
5. 对于方法，选择 **`ANY`**。
6. 对于路径，请输入 **`/{proxy+}`**。路径末尾 `{proxy+}` 是一个贪婪型路径变量。API Gateway 将所有请求发送到您的 API 到此路由。
7. 选择创建。

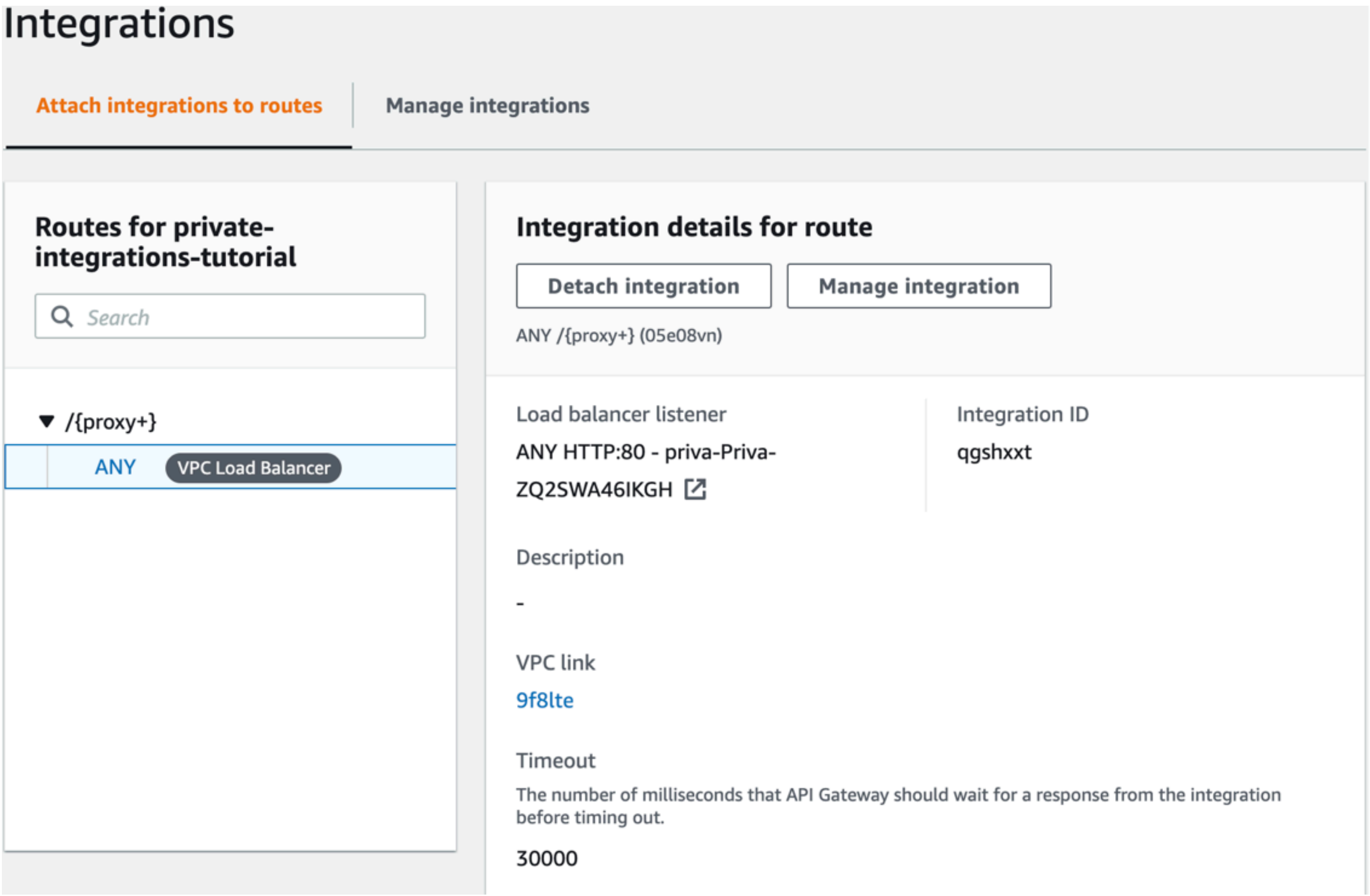
步骤 5：创建集成

您可以创建集成以将路由连接到后端资源。

要创建集成

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 选择集成。
4. 选择管理集成，然后选择创建。
5. 对于将此集成附加到一个路由，请选择您之前创建的 ANY/{proxy+} 路由。
6. 对于集成类型，选择私有资源。
7. 对于集成详情，请选择手动选择。
8. 对于目标服务，请选择 ALB/NLB。
9. 对于负载均衡器)，请选择您在步骤 1 中使用 AWS CloudFormation 模板创建的负载均衡器。其名称应当以 http-Priva 为开头。
10. 对于侦听器，请选择 **`HTTP 80`**。
11. 对于 VPC 链接，请选择您在步骤 2 中创建的 VPC 链接。其名称应当为 `private-integrations-tutorial`。
12. 选择创建。

要验证您的路由和集成设置是否正确，请选择将集成附加到路由。控制台显示您拥有集成到 VPC 负载均衡器的 `ANY /{proxy+}` 路由。



ANY /{proxy+} (05e08vn)

Load balancer listener
ANY HTTP:80 - priva-Priva-ZQ2SWA46IKGH

Integration ID
qgshxxt

Description
-

VPC link
9f8lte

Timeout
The number of milliseconds that API Gateway should wait for a response from the integration before timing out.
30000

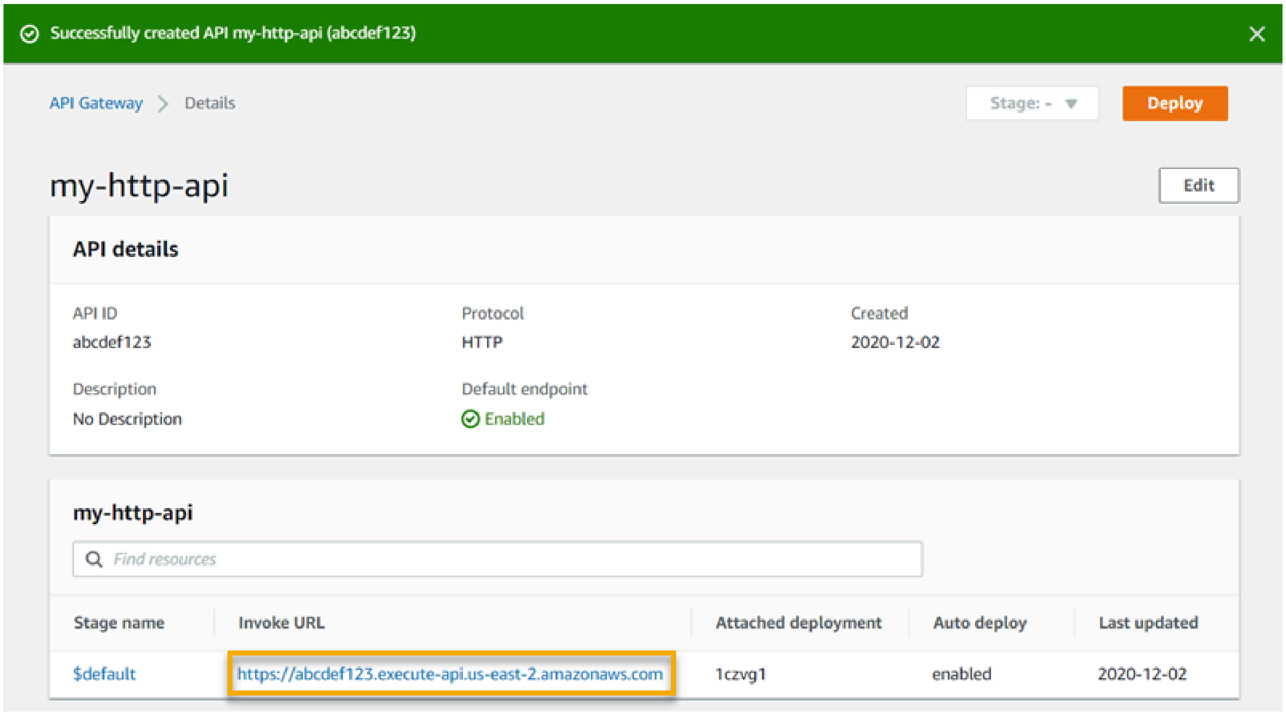
现在您已经准备好测试 API 了。

步骤 6：测试您的 API

接下来，测试您的 API 以确保它正常工作。为简单起见，请使用 Web 浏览器调用 API。

要测试您的 API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 请记下您的 API 的调用 URL。


Successfully created API my-http-api (abcdef123)

my-http-api

API details

API ID
abcdef123

Protocol
HTTP

Created
2020-12-02

Description
No Description

Default endpoint
Enabled

my-http-api

Stage name	Invoke URL	Attached deployment	Auto deploy	Last updated
$default	https://abcdef123.execute-api.us-east-2.amazonaws.com	1czvg1	enabled	2020-12-02

4. 在 Web 浏览器中，转到 API 的调用 URL。

   完整的 URL 应类似于 `https://`*`abcdef123`*`.execute-api.`*`us-east-2`*`.amazonaws.com`。

   您的浏览器向 API 发送 `GET` 请求。

5. 验证 API 的响应是欢迎消息，告诉您应用程序正在 Amazon ECS 上运行。

   如果您看到欢迎消息，则表示您成功创建了在 Amazon VPC 中运行的 Amazon ECS 服务，并使用带有 VPC 链接的 API 网关 HTTP API 访问 Amazon ECS 服务。

步骤 7：清除

为避免不必要的成本，请删除作为本教程的一部分而创建的资源。以下步骤将删除您的 VPC 链接、AWS CloudFormation 堆栈和 HTTP API。

要删除 HTTP API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在 API 页面上，选择一个 API。选择操作，选择删除，然后确认您的选择。

要删除 VPC 链接

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 VPC 链接。
3. 选择您的 VPC 链接，选择删除，然后确认您的选择。

删除 AWS CloudFormation 堆栈

1. 打开 AWS CloudFormation 控制台，地址：https://console.aws.amazon.com/cloudformation。
2. 选择您的 AWS CloudFormation 堆栈。
3. 选择删除，然后确认您的选择。

接下来的步骤：通过 实现自动化AWS CloudFormation

您可以自动创建和清理本教程中涉及的所有AWS资源。如需完整的示例 AWS CloudFormation 模板，请参阅 template.yaml。

Amazon API Gateway WebSocket API 教程

以下教程提供有助于您了解 API Gateway WebSocket API 的实践练习。

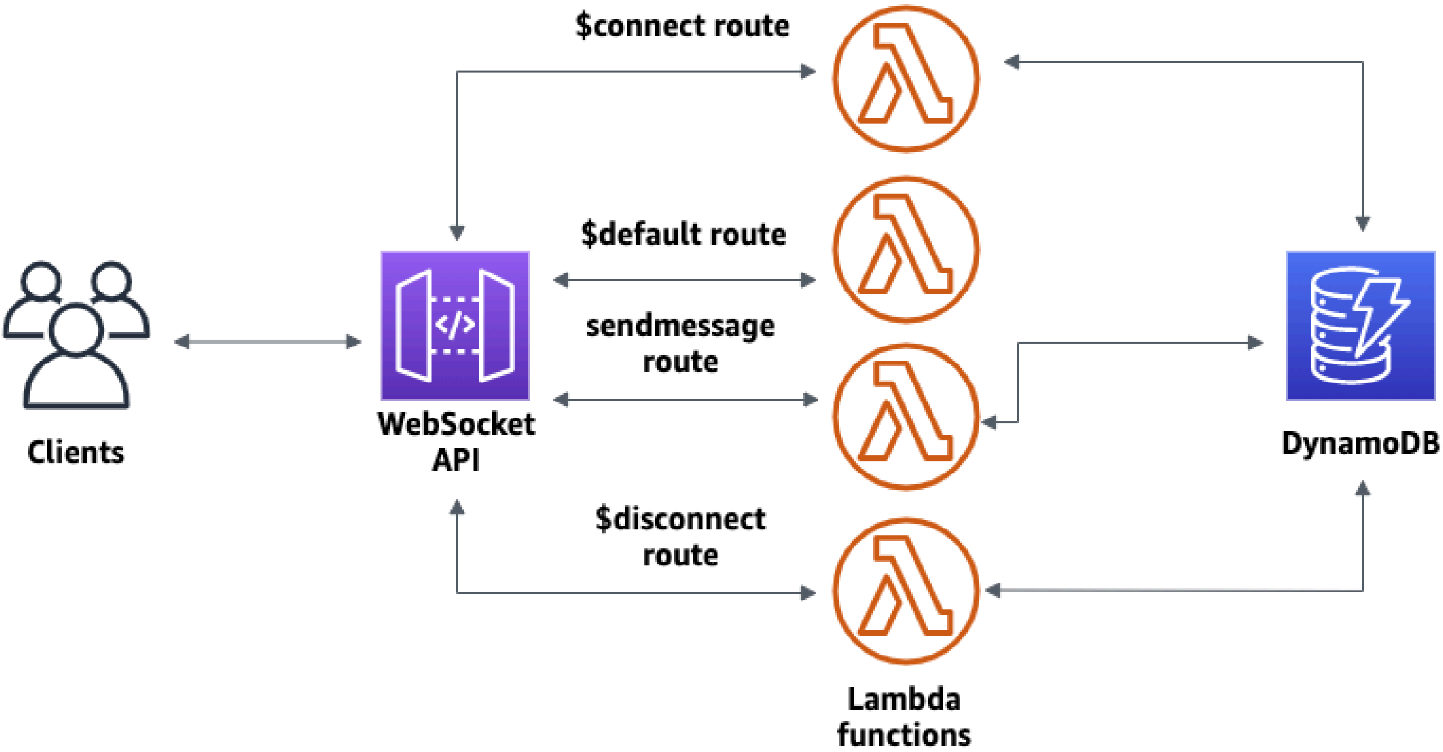
主题

- 教程：使用 WebSocket API、Lambda 和 DynamoDB 构建无服务器聊天应用程序
- 教程：使用三种集成类型构建无服务器应用程序

教程：使用 WebSocket API、Lambda 和 DynamoDB 构建无服务器聊天应用程序

在本教程中，您将使用 WebSocket API 创建无服务器聊天应用程序。借助 WebSocket API，您可以支持客户端之间的双向通信。客户端可以接收消息，而无需轮询更新。

完成本教程需要大约 30 分钟。首先，您将使用 AWS CloudFormation 模板来创建用于处理 API 请求的 Lambda 函数，以及存储客户端 ID 的 DynamoDB 表。然后，您将使用 API Gateway 控制台创建与您的 Lambda 函数集成的 WebSocket API。最后，您将测试 API 以验证消息是否已发送和接收。



要完成本教程，您需要一个AWS账户以及一位具有控制台访问权限的 AWS Identity and Access Management 用户。有关更多信息，请参阅 先决条件。

还需要 `wscat` 以连接到 API。有关更多信息，请参阅 the section called “使用 `wscat` 连接到 WebSocket API 并向其发送消息”。

主题

步骤 1：创建 Lambda 函数和 DynamoDB 表

下载并解压缩适用于 AWS CloudFormation 的应用程序创建模板。您将使用此模板创建 Amazon DynamoDB 表以存储应用程序的客户端 ID。每个连接的客户端都有一个唯一 ID，我们要将其用作表的分区键。此模板还创建 Lambda 函数，这些函数用于更新 DynamoDB 中的客户端连接并处理向已连接的客户端发送消息的事宜。

创建 AWS CloudFormation 堆栈

1. 打开 AWS CloudFormation 控制台，地址：https://console.aws.amazon.com/cloudformation。
2. 选择创建堆栈，然后选择使用新资源(标准)。
3. 对于指定模板，选择上传模板文件。
4. 选择您下载的模板。
5. 选择下一步。
6. 对于堆栈名称，输入 **`websocket-api-chat-app-tutorial`**，然后选择下一步。
7. 对于配置堆栈选项，请选择下一步。
8. 对于功能，请确认 AWS CloudFormation 可以在您的账户中创建 IAM 资源。
9. 选择提交。

AWS CloudFormation 预置在模板中指定的资源。完成资源预置可能需要几分钟时间。当 AWS CloudFormation 堆栈的状态为 CREATE_COMPLETE 时，您就可以继续下一步了。

步骤 2：创建 WebSocket API

您将创建 WebSocket API 来处理客户端连接并将请求路由到您在步骤 1 中创建的 Lambda 函数。

创建 WebSocket API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择创建 API。对于 WebSocket API，选择构建。
3. 对于 API 名称，请输入 **`websocket-chat-app-tutorial`**。
4. 对于路由选择表达式，输入 **`request.body.action`**。路由选择表达式确定当客户端发送消息时 API Gateway 调用的路由。
5. 选择下一步。
6. 对于 Predefined routes（预定义路由），选择 Add $connect（添加 $connect）、Add $disconnect（添加 $disconnect）和 Add $default（添加 $default）。$connect 和 $disconnect 路由是 API Gateway 在客户端连接到 API 或断开与 API 的连接时自动调用的特殊路由。当没有其他路由与请求匹配时，API Gateway 调用 `$default` 路由。
7. 对于 Custom routes（自定义路由），选择 Add custom route（添加自定义路由）。对于 Route key（路由键），请输入 **`sendmessage`**。此自定义路由处理发送到已连接的客户端的消息。

8. 选择下一步。
9. 在 Attach integrations（附加集成）下，对于每个路由和 Integration type（集成类型），选择 Lambda。

   对于 Lambda，请选择您在步骤 1 中使用 AWS CloudFormation 创建的相应 Lambda 函数。每个函数的名称与一个路由匹配。例如，对于 $connect 路由，选择名为 **`websocket-chat-app-tutorial-ConnectHandler`** 的函数。
10. 查看 API Gateway 为您创建的阶段。原定设置情况下，API Gateway 会创建阶段名称 `production`，然后自动将您的 API 部署到该阶段。选择下一步。
11. 选择 Create and deploy（创建和部署）。

步骤 3：测试您的 API

接下来，测试您的 API 以确保它正常工作。使用 `wscat` 命令连接到 API。

获取调用 API 的调用 URL

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 选择阶段，然后选择生产。
4. 记下 API 的 WebSocket URL。该 URL 应类似于 `wss://abcdef123.execute-api.us-east-2.amazonaws.com/production`。

连接到您的 API

1. 使用以下命令连接到您的 API。当您连接到 API 时，API Gateway 会调用 `$connect` 路由。调用此路由时，它会调用用于将连接 ID 存储在 DynamoDB 中的 Lambda 函数。

```
wscat -c wss://abcdef123.execute-api.us-west-2.amazonaws.com/production
```

```
Connected (press CTRL+C to quit)
```

2. 打开一个新终端并使用以下参数再次运行 wscat 命令。

```
wscat -c wss://abcdef123.execute-api.us-west-2.amazonaws.com/production
```

```
Connected (press CTRL+C to quit)
```

这为您提供了两个可以交换消息的已连接客户端。

发送邮件

- API Gateway 根据 API 的路由选择表达式确定要调用的路由。您的 API 的路由选择表达式是 `$request.body.action`。因此，当您发送以下消息时，API Gateway 会调用 `sendmessage` 路由：

  ```
  {"action": "sendmessage", "message": "hello, everyone!"}
  ```

  与调用的路由关联的 Lambda 函数会从 DynamoDB 收集客户端 ID。然后，该函数调用 API Gateway 管理 API 并将消息发送给这些客户端。所有连接的客户端都会收到以下消息：

  ```
  < hello, everyone!
  ```

调用 API 的 $default 路由

- 当客户端发送与您定义的路由不匹配的消息时，API Gateway 会调用您的 API 的默认路由。与 `$default` 路由关联的 Lambda 函数使用 API Gateway 管理 API 发送有关其连接的客户端信息。

  ```
  test
  ```

  ```
  Use the sendmessage route to send a message. Your info:
   {"ConnectedAt":"2022-01-25T18:50:04.673Z","Identity":
  {"SourceIp":"192.0.2.1","UserAgent":null},"LastActiveAt":"2022-01-25T18:50:07.642Z","connec
  ```

从 API 断开连接

- 按 **CTRL+C** 以从 API 断开连接。当客户端与 API 断开连接时，API Gateway 会调用 API 的 `$disconnect` 路由。适用于 API 的 `$disconnect` 路由的 Lambda 集成会从 DynamoDB 中删除连接 ID。

步骤 4：清除

为避免不必要的成本，请删除作为本教程的一部分而创建的资源。以下步骤将删除您的 AWS CloudFormation 堆栈和 WebSocket API。

删除 WebSocket API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在 API 页面上，选择您的 `websocket-chat-app-tutorial` API。选择操作，选择删除，然后确认您的选择。

删除 AWS CloudFormation 堆栈

1. 打开 AWS CloudFormation 控制台，地址：https://console.aws.amazon.com/cloudformation。
2. 选择您的 AWS CloudFormation 堆栈。
3. 选择删除，然后确认您的选择。

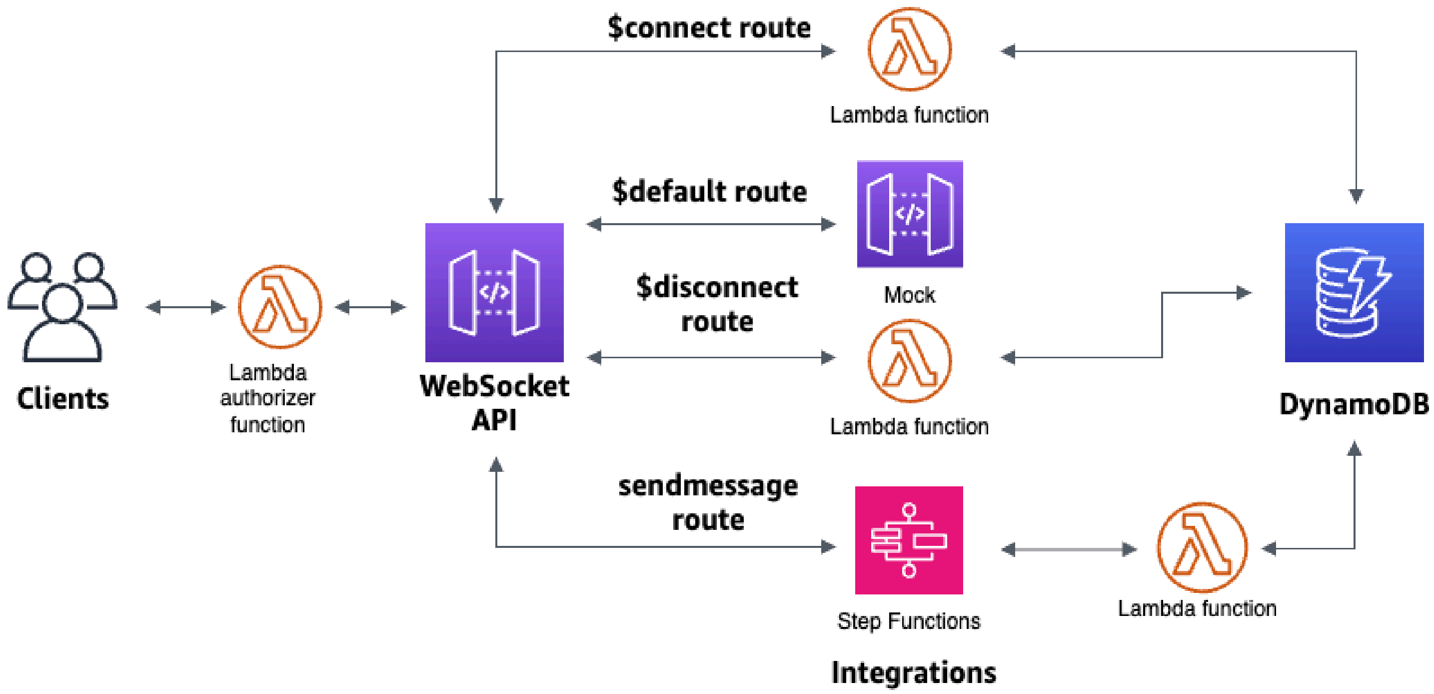
接下来的步骤：通过 实现自动化AWS CloudFormation

您可以自动创建和清理本教程中涉及的所有 AWS 资源。有关创建此 API 和所有相关资源的 AWS CloudFormation 模板，请参阅 chat-app.yaml。

教程：使用三种集成类型构建无服务器应用程序

在本教程中，您使用 WebSocket API 创建一个无服务器广播应用程序。客户端可以接收消息，而无需轮询更新。

本教程介绍如何向连接的客户端广播消息，并包括 Lambda 授权方、模拟集成以及与 Step Functions 的非代理集成的示例。



使用 AWS CloudFormation 模板创建资源后，您将使用 API Gateway 控制台来创建与 AWS 资源集成的 WebSocket API。您需要将 Lambda 授权方附加到您的 API，并创建 AWS 服务与 Step Functions 的集成，才能启动状态机执行。Step Functions 状态机将调用一个 Lambda 函数，来向所有连接的客户端发送消息。

构建 API 后，您将测试到 API 的连接，并验证消息是否已发送和接收。完成本教程需要大约 45 分钟。

主题

先决条件

您需要以下先决条件：

- 拥有控制台访问权限的 AWS 账户和 AWS Identity and Access Management 用户。有关更多信息，请参阅 先决条件。
- 要连接到您的 API 的 `wscat` 有关更多信息，请参阅 the section called “使用 `wscat` 连接到 WebSocket API 并向其发送消息”。

我们建议您在开始本教程之前，完成 WebSocket 聊天应用程序教程。要完成 WebSocket 聊天应用程序教程，请参阅the section called “WebSocket 聊天应用程序”。

步骤 1：创建资源

下载并解压缩适用于 AWS CloudFormation 的应用程序创建模板。您将使用此模板创建以下各项：

- 处理 API 请求并授权访问您的 API 的 Lambda 函数。
- 用于存储客户端 ID 和由 Lambda 授权方返回的主体用户标识的 DynamoDB 表。
- 用于向连接的客户端发送消息的 Step Functions 状态机。

创建 AWS CloudFormation 堆栈

1. 打开 AWS CloudFormation 控制台，地址：https://console.aws.amazon.com/cloudformation。
2. 选择创建堆栈，然后选择使用新资源(标准)。
3. 对于指定模板，选择上传模板文件。
4. 选择您下载的模板。
5. 选择下一步。
6. 对于堆栈名称，输入 **`websocket-step-functions-tutorial`**，然后选择下一步。
7. 对于配置堆栈选项，请选择下一步。
8. 对于功能，请确认 AWS CloudFormation 可以在您的账户中创建 IAM 资源。
9. 选择提交。

AWS CloudFormation 预置在模板中指定的资源。完成资源预置可能需要几分钟时间。选择输出选项卡，来查看您创建的资源及其 ARN。当 AWS CloudFormation 堆栈的状态为 CREATE_COMPLETE 时，您就可以继续下一步了。

步骤 2：创建 WebSocket API

您将创建一个 WebSocket API 来处理客户端连接，并将请求路由到您在步骤 1 中创建的资源。

创建 WebSocket API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择创建 API。对于 WebSocket API，选择构建。
3. 对于 API 名称，请输入 **`websocket-step-functions-tutorial`**。
4. 对于路由选择表达式，输入 **`request.body.action`**。

   路由选择表达式确定当客户端发送消息时 API Gateway 调用的路由。
5. 选择下一步。
6. 对于预定义路由，选择添加 $connect、添加 $disconnect、添加 $default。

   $connect 和 $disconnect 路由是 API Gateway 在客户端连接到 API 或断开与 API 的连接时自动调用的特殊路由。当没有其它路由与请求匹配时，API Gateway 调用 $default 路由。创建 API 后，您将创建一个连接到 Step Functions 的自定义路由。
7. 选择下一步。
8. 对于 $connect 的集成，请执行以下操作：
   a. 对于集成类型，选择 Lambda。
   b. 对于 Lambda 函数，选择您在步骤 1 中使用 AWS CloudFormation 创建的相应 $connect Lambda 函数。Lambda 函数名称应该以 **`websocket-step`** 开头。
9. 对于 $disconnect 的集成，请执行以下操作：
   a. 对于集成类型，选择 Lambda。
   b. 对于 Lambda 函数，选择您在步骤 1 中使用 AWS CloudFormation 创建的相应 $disconnect Lambda 函数。Lambda 函数名称应该以 **`websocket-step`** 开头。
10. 对于 $default 的集成，请选择模拟。

    在模拟集成中，API Gateway 在没有集成后端的情况下管理路由响应。
11. 选择下一步。
12. 查看 API Gateway 为您创建的阶段。默认情况下，API Gateway 会创建名为生产的阶段，然后自动将您的 API 部署到该阶段。选择下一步。

13. 选择创建和部署。

步骤 3：创建 Lambda 授权方

要控制对您的 WebSocket API 的访问权限，您需要创建 Lambda 授权方。AWS CloudFormation 模板为您创建了 Lambda 授权方函数。您可以在 Lambda 控制台中看到 Lambda 函数。名称应以 **`websocket-step-functions-tutorial-AuthorizerHandler`** 开头。除非 `Authorization` 标头是 `Allow`，否则此 Lambda 函数会拒绝对 WebSocket API 的所有调用。Lambda 函数还会将 `$context.authorizer.principalId` 变量传递给您的 API，稍后会在 DynamoDB 表中使用该变量来标识 API 调用方。

在此步骤中，您将配置 $connect 路由来使用 Lambda 授权方。

创建 Lambda 授权方

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在主导航窗格中，选择授权方。
3. 选择创建授权方。
4. 对于授权方名称，输入 **`LambdaAuthorizer`**。
5. 对于授权方 ARN，输入由 AWS CloudFormation 模板创建的授权方的名称。名称应以 **`websocket-step-functions-tutorial-AuthorizerHandler`** 开头。

   > Note
   >
   > 我们建议不要为生产 API 使用此示例授权方。

6. 对于身份来源类型，选择标头。对于键，输入 **`Authorization`**。
7. 选择创建授权方。

创建授权方后，将其附加到您的 API 的 $connect 路由。

将授权方附加到 $connect 路由

1. 在主导航窗格中，选择路由。
2. 选择 $connect 路由。
3. 在路由请求设置部分中，选择编辑。

4. 对于授权，请选择下拉菜单，然后选择您的请求授权方。
5. 选择保存更改。

步骤 4：创建模拟双向集成

接下来，您将为 $default 路由创建双向模拟集成。模拟集成可让您在不使用后端的情况下向客户端发送响应。当您为 $default 路由创建集成时，您可以向客户端展示如何与 API 进行交互。

您可以配置 $default 路由来通知客户端使用 sendmessage 路由。

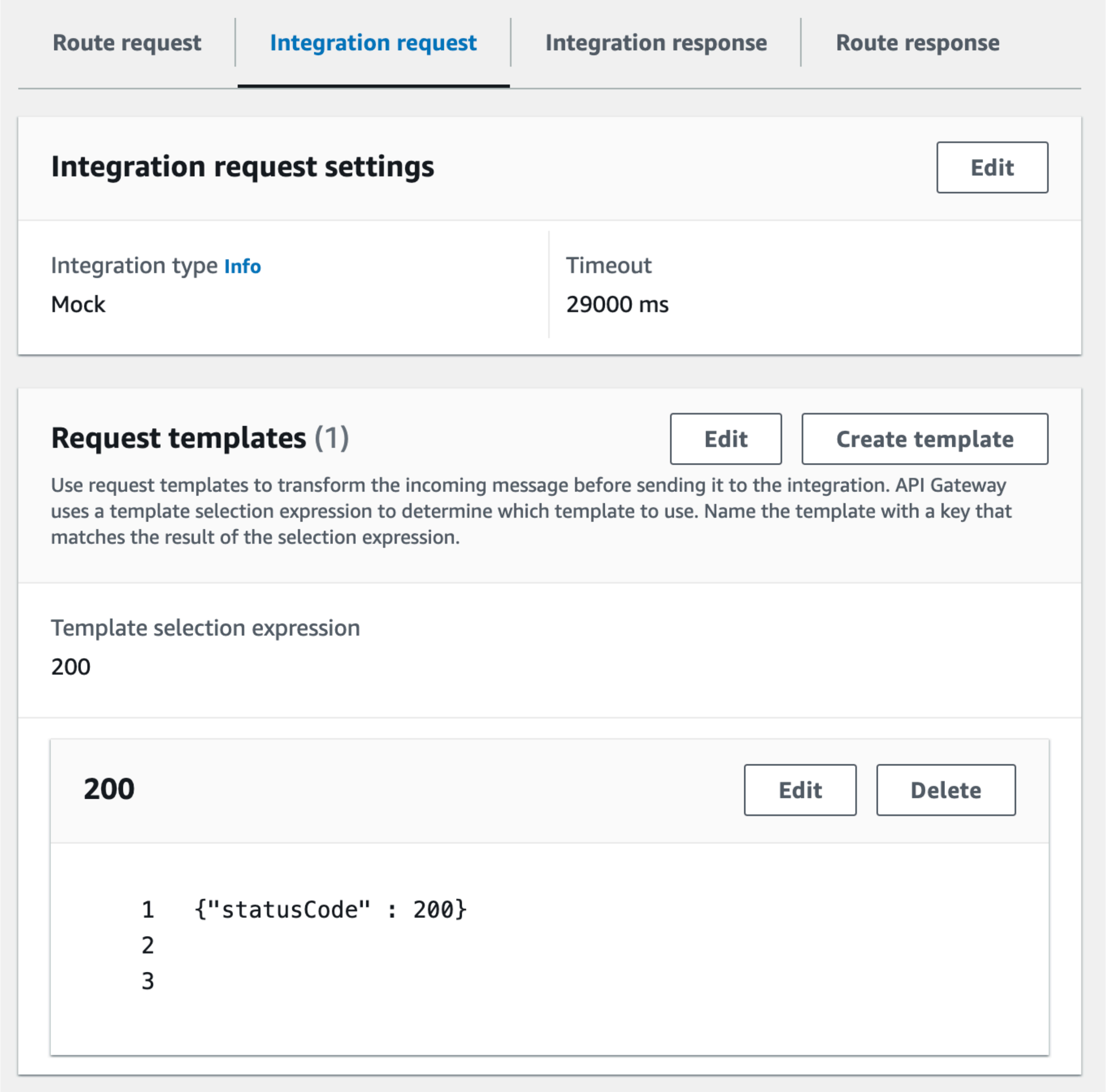
创建模拟请求

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 $default 路由，然后选择集成请求选项卡。
3. 对于请求模板，请选择编辑。
4. 对于模板选择表达式，输入 **`200`**，然后选择编辑。
5. 在集成请求选项卡上，对于请求模板，选择创建模板。
6. 对于模板密钥，输入 **`200`**。
7. 对于生成模板，输入以下映射模板：

   ```
   {"statusCode": 200}
   ```

   选择创建模板。

   结果应该类似以下内容：



Integration request settings

Integration type Info
Mock

Timeout
29000 ms

Request templates (1)

Use request templates to transform the incoming message before sending it to the integration. API Gateway uses a template selection expression to determine which template to use. Name the template with a key that matches the result of the selection expression.

Template selection expression
200

200

```
{"statusCode" : 200}


```

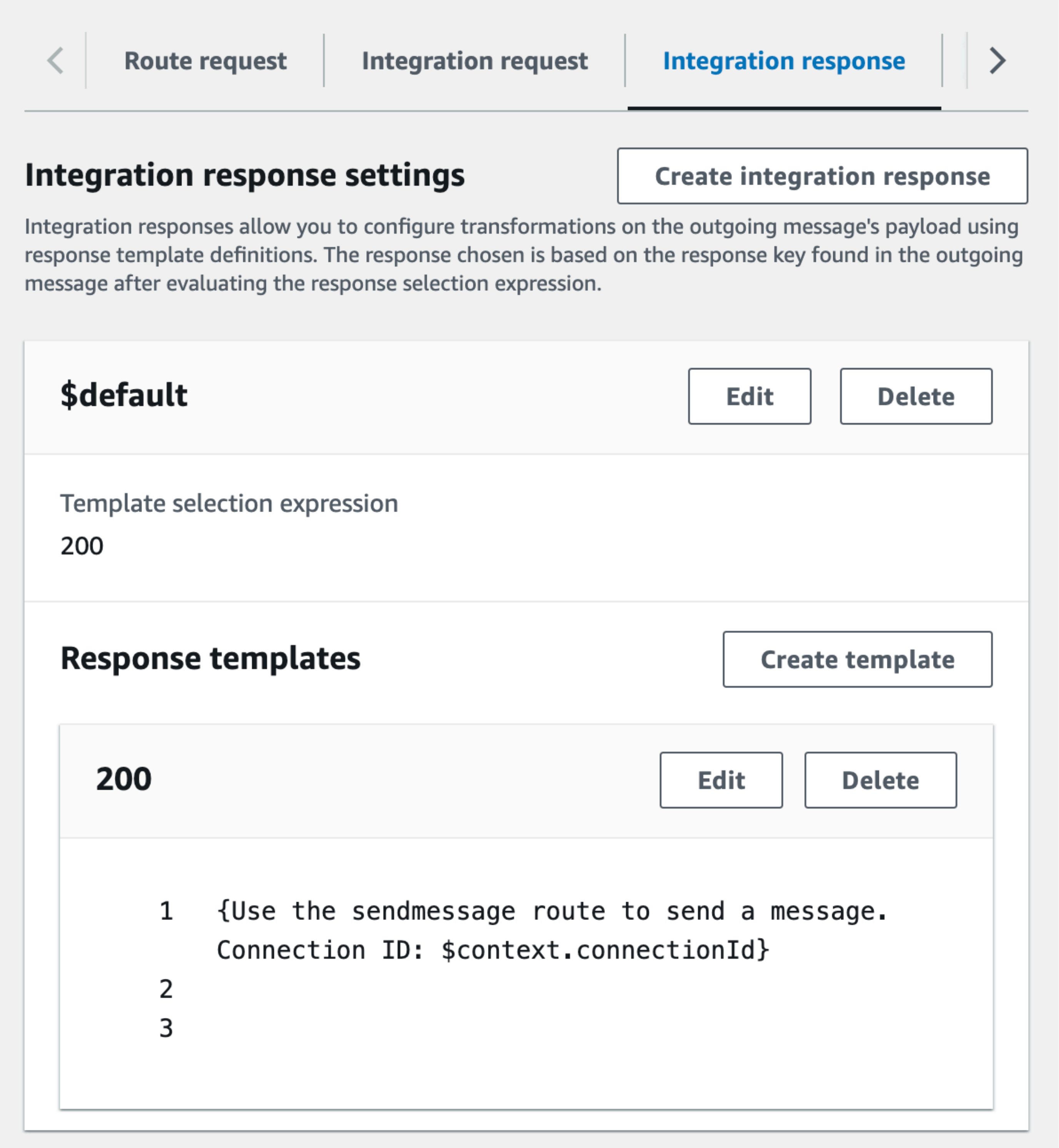
8. 在 $default 路由窗格中，选择启用双向通信。
9. 选择集成响应选项卡，然后选择创建集成响应。
10. 对于响应密钥，输入 **`$default`**。
11. 对于模板选择表达式，输入 **`200`**。
12. 选择创建响应。
13. 在响应模板下，选择创建模板。

14. 对于模板密钥，输入 **200**。
15. 对于响应模板，输入以下映射模板：

    ```
    {"Use the sendmessage route to send a message. Connection ID:
     $context.connectionId"}
    ```

16. 选择创建模板。

    结果应该类似以下内容：



Integration response settings

Integration responses allow you to configure transformations on the outgoing message's payload using response template definitions. The response chosen is based on the response key found in the outgoing message after evaluating the response selection expression.

$default

Template selection expression
200

Response templates

200

```
{Use the sendmessage route to send a message. Connection ID: $context.connectionId}
```

步骤 5：使用 Step Functions 创建非代理集成

接下来，创建 sendmessage 路由。客户端可以调用 sendmessage 路由，来向所有连接的客户端广播消息。sendmessage 路由已将非代理 AWS 服务与 AWS Step Functions 集成。该集成会针对 AWS CloudFormation 模板为您创建的 Step Functions 状态机调用 [StartExecution](#) 命令。

创建非代理集成

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择创建路由。
3. 对于路由键，请输入 **`sendmessage`**。
4. 对于集成类型，选择 AWS 服务。
5. 对于 AWS 区域，输入您部署了 AWS CloudFormation 模板的区域。
6. 对于 AWS 服务，选择 Step Functions。
7. 对于 HTTP 方法，选择 POST。
8. 对于操作名称，输入 **`StartExecution`**。
9. 对于执行角色，输入由 AWS CloudFormation 模板创建的执行角色。名称应为 WebsocketTutorialApiRole。
10. 选择创建路由。

接下来，您创建一个映射模板，来将请求参数发送到 Step Functions 状态机。

创建映射模板

1. 选择 sendmessage 路由，然后选择集成请求选项卡。
2. 在请求模板部分中，选择编辑。
3. 对于模板选择表达式，输入 **`\$default`**。
4. 选择编辑。
5. 在请求模板部分，选择创建模板。
6. 对于模板密钥，输入 **`\$default`**。
7. 对于生成模板，输入以下映射模板：

```
#set($domain = "$context.domainName")
#set($stage = "$context.stage")
```

```
#set($body = $input.json('$'))
#set($getMessage = $util.parseJson($body))
#set($mymessage = $getMessage.message)
{
"input": "{\"domain\": \"$domain\", \"stage\": \"$stage\", \"message\":
 \"$mymessage\"}",
"stateMachineArn": "arn:aws:states:us-east-2:123456789012:stateMachine:WebSocket-
Tutorial-StateMachine"
}
```

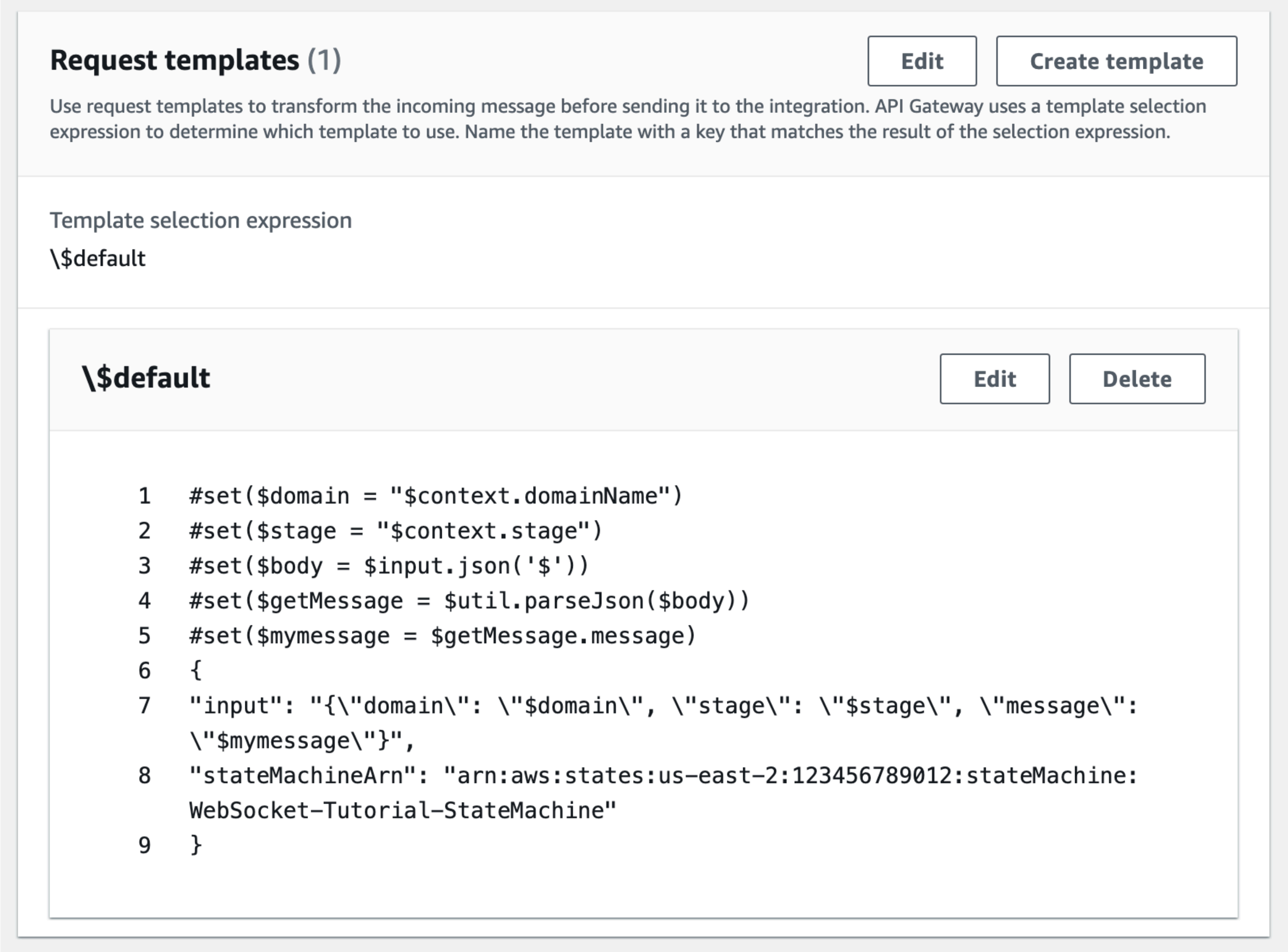
将 *`stateMachineArn`* 替换为由 AWS CloudFormation 创建的状态机的 ARN。

映射模板执行以下操作：

- 使用上下文变量 `domainName` 创建变量 `$domain`。
- 使用上下文变量 `stage` 创建变量 `$stage`。

  `$domain` 和 `$stage` 变量是构建回调 URL 所必需的。
- 接收传入的 `sendmessage` JSON 消息，并提取 `message` 属性。
- 为状态机创建输入。输入是 WebSocket API 的域和阶段以及来自 `sendmessage` 路由的消息。

8. 选择创建模板。



Request templates (1)

Use request templates to transform the incoming message before sending it to the integration. API Gateway uses a template selection expression to determine which template to use. Name the template with a key that matches the result of the selection expression.

Template selection expression

\$default

\$default

```
#set($domain = "$context.domainName")
#set($stage = "$context.stage")
#set($body = $input.json('$'))
#set($getMessage = $util.parseJson($body))
#set($mymessage = $getMessage.message)
{
"input": "{\"domain\": \"$domain\", \"stage\": \"$stage\", \"message\": \"$mymessage\"}",
"stateMachineArn": "arn:aws:states:us-east-2:123456789012:stateMachine:WebSocket-Tutorial-StateMachine"
}
```

您可以在 $connect 或 $disconnect 路由上创建非代理集成，来直接在 DynamoDB 表中添加或移除连接 ID，而无需调用 Lambda 函数。

步骤 6：测试您的 API

接下来，您将部署和测试您的 API 来确保它正常工作。您将使用 `wscat` 命令连接到 API，然后，您将使用斜杠命令发送 ping 帧来检查与 WebSocket API 的连接。

部署 API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在主导航窗格中，选择路由。
3. 选择部署 API。

4. 对于阶段，选择生产。
5. （可选）对于部署描述，输入描述。
6. 选择部署。

部署 API 后，您可以调用它。使用调用 URL 来调用您的 API。

获取您的 API 的调用 URL

1. 选择 API。
2. 选择阶段，然后选择生产。
3. 记下 API 的 WebSocket URL。该 URL 应类似于 `wss://abcdef123.execute-api.us-east-2.amazonaws.com/production`。

现在您已经有了调用 URL，您可以测试与 WebSocket API 的连接了。

测试与您的 API 的连接

1. 使用以下命令连接到您的 API。首先，通过调用 `/ping` 路径来测试连接。

   ```
   wscat -c wss://abcdef123.execute-api.us-east-2.amazonaws.com/production -H
    "Authorization: Allow" --slash -P
   ```

   ```
   Connected (press CTRL+C to quit)
   ```

2. 输入以下命令来 ping 控制帧。您可以从客户端使用控制帧来实现保持活动目的。

   ```
   /ping
   ```

   结果应该类似以下内容：

   ```
   < Received pong (data: "")
   ```

现在您已经测试了连接，您可以测试 API 是否正常工作。在此步骤中，您打开一个新的终端窗口，以便 WebSocket API 可以向所有连接的客户端发送消息。

要测试您的 API

1. 打开一个新终端并使用以下参数再次运行 `wscat` 命令。

   ```
   wscat -c wss://abcdef123.execute-api.us-east-2.amazonaws.com/production -H
    "Authorization: Allow"
   ```

   ```
   Connected (press CTRL+C to quit)
   ```

2. API Gateway 根据 API 的路由请求选择表达式确定要调用的路由。您的 API 的路由选择表达式是 `$request.body.action`。因此，当您发送以下消息时，API Gateway 会调用 `sendmessage` 路由：

   ```
   {"action": "sendmessage", "message": "hello, from Step Functions!"}
   ```

   与路由关联的 Step Functions 状态机使用消息和回调 URL 调用 Lambda 函数。Lambda 函数调用 API Gateway 管理 API，并将消息发送给所有连接的客户端。所有客户端都会收到以下消息：

   ```
   < hello, from Step Functions!
   ```

现在您已经测试了 WebSocket API，可以断开与 API 的连接。

从 API 断开连接

- 按 `CTRL+C` 以从 API 断开连接。

  当客户端与 API 断开连接时，API Gateway 会调用 API 的 $disconnect 路由。适用于 API 的 $disconnect 路由的 Lambda 集成会从 DynamoDB 中删除连接 ID。

步骤 7：清除

为避免不必要的成本，请删除作为本教程的一部分而创建的资源。以下步骤将删除您的 AWS CloudFormation 堆栈和 WebSocket API。

删除 WebSocket API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。

2. 在 API 页面上，选择您的 websocket-api。
3. 选择操作，选择删除，然后确认您的选择。

删除 AWS CloudFormation 堆栈

1. 打开 AWS CloudFormation 控制台，地址：https://console.aws.amazon.com/cloudformation。
2. 选择您的 AWS CloudFormation 堆栈。
3. 选择删除，然后确认您的选择。

后续步骤

您可以自动创建和清理本教程中涉及的所有 AWS 资源。有关本教程中自动执行这些操作的 AWS CloudFormation 模板的示例，请参阅 ws-sfn.zip。

使用 REST API

API Gateway 中的 REST API 是与后端 HTTP 终端节点、Lambda 函数或其他 AWS 服务集成的资源与方法的集合。您可以使用 API Gateway 功能，帮助您处理 API 生命周期从创建到监控生产 API 的各个方面。

API Gateway REST API 使用请求/响应模型，其中客户端将请求发送到服务，服务将同步回应。对于许多依赖于同步通信的不同类型的应用程序，这种模型类型适用。

主题

在 API Gateway 中开发 REST API

在 Amazon API Gateway 中，您可以将 REST API 构建为称为 API Gateway 资源的可编程实体的集合。例如，您可以使用 RestApi 资源表示可以包含资源实体集合的 API。

每个 `Resource` 实体可以具有一个或多个 Method 资源。`Method` 是客户端提交的传入请求，以请求参数和正文表示。它定义客户端用于访问公开 `Resource` 的应用程序编程接口。要将 `Method` 与后端端点（也称为集成端点）进行集成，可创建 Integration 资源。这会将传入的请求转发到指定的集成端点 URI。如有必要，您可以转换请求参数或请求正文来满足后端要求。

对于响应，您可以创建 MethodResponse 资源来代表客户端收到的请求响应，然后可以创建 IntegrationResponse 资源来代表后端返回的请求响应。您可以配置集成响应，以在转换后端响应数据之后将其返回给客户端，或按原样将后端响应传递给客户端。

要帮助您的客户了解 API，您还可以在创建 API 时或之后为 API 提供文档。要实现这一点，请为受支持的 API 实体添加一个 DocumentationPart 资源。

要控制客户端调用 API 的方式，请使用 IAM 权限、Lambda 授权方或 Amazon Cognito 用户池。要计量 API 的使用情况，请设置使用计划以限制 API 请求。您可在创建或更新 API 时启用这些选项。

有关如何创建 API 的介绍，请参阅the section called “教程：使用 Lambda 代理集成的 Hello World API”。要了解有关开发 REST API 时可能使用的 API Gateway 功能的更多信息，请参阅以下主题。这些主题包含概念信息以及您可以使用 API Gateway 控制台、API Gateway REST API、AWS CLI 或其中一个 AWS SDK 执行的过程。

主题

- API Gateway API 端点类型
- API Gateway 中用于 REST API 的方法
- 在 API Gateway 中控制和管理对 REST API 的访问
- 设置 REST API 集成
- 在 API Gateway 中使用请求验证
- 为 REST API 设置数据转换
- API Gateway 中的网关响应
- 为 REST API 资源启用 CORS
- 使用 REST API 的二进制媒体类型
- 在 Amazon API Gateway 中调用 REST API
- 使用 OpenAPI 配置 REST API

API Gateway API 端点类型

API 端点 类型指的是 API 的主机名。API 端点类型可以是边缘优化的、区域的或私有的，具体取决于您的大部分 API 流量的源头位置。

边缘优化的 API 端点

边缘优化的 API 端点通常将请求路由至最近的 CloudFront 入网点（PoP），这在您的客户呈地理分布的情况下可能会有所帮助。这是 API Gateway REST API 的默认端点类型。

边缘优化 API 利用 HTTP 标头的名称（例如，`Cookie`）。

CloudFront 在转发请求到源之前以自然顺序按 Cookie 名称对 HTTP Cookie 进行排序。有关 CloudFront 如何处理 Cookie 的更多信息，请参阅基于 Cookie 缓存内容。

您用于边缘优化 API 的任何自定义域名都适用于所有区域。

区域 API 端点

区域 API 端点 适用于同一区域中的客户端。当在 EC2 实例上运行的客户端调用同一区域中的 API，或 API 用于为具有高需求的少数客户端提供服务时，区域 API 可以降低连接开销。

对于区域 API，您使用的任何自定义域名都特定于部署了 API 的区域。如果您在多个区域中部署区域 API，则它可以在所有区域中具有相同的自定义域名。您可以将自定义域与 Amazon Route 53 一起使用来执行诸如基于延迟的路由之类的任务。有关更多信息，请参阅 the section called “设置区域自定义域名” 和 the section called “创建边缘优化的自定义域名”。

区域 API 端点按原样传递所有标头名称。

> Note
>
> 在 API 客户端在地理上分散的案例中，使用区域 API 端点以及您自己的 Amazon CloudFront 分配可能仍然有意义，这可确保 API Gateway 不会将 API 与服务控制的 CloudFront 分配关联。有关此使用案例的更多信息，请参阅如何使用我自己的 CloudFront 分配设置 API Gateway？

私有 API 端点

私有 API 端点 是一个只能使用接口 VPC 端点从 Amazon Virtual Private Cloud (VPC) 访问的 API 端点，该接口是您在 VPC 中创建的端点网络接口 (ENI)。有关更多信息，请参阅 the section called “私有 REST API”。

私有 API 端点按原样传递所有标头名称。

在 API Gateway 中更改公有或私有 API 端点类型

更改 API 端点类型要求您更新 API 的配置。您可以使用 API Gateway 控制台、AWS CLI 或适用于 API Gateway 的AWS开发工具包更改现有 API 类型。端点类型无法再次进行更改，直到当前更改完成，但您的 API 将可用。

支持以下端点类型更改：

- 从边缘优化到区域或私有
- 从区域到边缘优化或私有
- 从私有到区域

您不能将私有 API 更改为边缘优化的 API。

如果您正在将公有 API 从边缘优化更改为区域（或反之），请注意边缘优化 API 可能与区域 API 具有不同的行为。例如，边缘优化的 API 删除 `Content-MD5` 标头。传递到后端的任意 MD5 哈希值可以表示在请求字符串参数或正文属性中。但是，区域 API 会传递此标头映射，虽然它可能会将标头名称重新映射到某个其它名称。了解区别有助于您决定如何将边缘优化的 API 更新为区域 API，或者从区域 API 更新为边缘优化的 API。

主题

- 使用 API Gateway 控制台更改 API 端点类型
- 使用 AWS CLI 更改 API 端点类型

使用 API Gateway 控制台更改 API 端点类型

要更改 API 的 API 端点类型，请执行以下步骤：

将一个公有端点从区域转换为边缘优化（以及反之）

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 选择 API 设置。
4. 在 API 详细信息部分中，选择编辑。
5. 对于 API 端点类型，选择边缘优化或区域。
6. 选择 Save changes（保存更改）。
7. 重新部署您的 API，以使更改生效。

将私有端点转换为区域端点

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 编辑 API 的资源策略以删除对 VPC 或 VPC 端点的任何提及，以便 API 从您的 VPC 之外以及在您的 VPC 之内进行的调用成功。
4. 选择 API 设置。
5. 在 API 详细信息部分中，选择编辑。

6. 对于 API 端点类型，选择区域。
7. 选择 Save changes（保存更改）。
8. 从 API 中删除资源策略。
9. 重新部署您的 API，以使更改生效。

将区域端点转换为私有端点

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 创建资源策略来授予对您的 VPC 或 VPC 端点的访问权限。有关更多信息，请参阅 ???。
4. 选择 API 设置。
5. 在 API 详细信息部分中，选择编辑。
6. 对于 API 端点类型，选择私有。
7. （可选）对于 VPC 端点 ID，选择您要与私有 API 关联的 VPC 端点 ID。
8. 选择 Save changes（保存更改）。
9. 重新部署您的 API，以使更改生效。

使用 AWS CLI 更改 API 端点类型

要使用 AWS CLI 更新其 API ID 为 *{api-id}* 的边缘优化的 API，请按如下所示调用 update-rest-api：

```
aws apigateway update-rest-api \
    --rest-api-id {api-id} \
    --patch-operations op=replace,path=/endpointConfiguration/types/EDGE,value=REGIONAL
```

成功的响应包含 `200 OK` 状态代码以及与以下类似的负载：

```
{


    "createdDate": "2017-10-16T04:09:31Z",
    "description": "Your first API with Amazon API Gateway. This is a sample API that
 integrates via HTTP with our demo Pet Store endpoints",
    "endpointConfiguration": {
```

```
        "types": "REGIONAL"
    },
    "id": "0gsnjtjck8",
    "name": "PetStore imported as edge-optimized"
}
```

相反，也可以将区域 API 更新为边缘优化的 API，如下所示：

```
aws apigateway update-rest-api \
    --rest-api-id {api-id} \
    --patch-operations op=replace,path=/endpointConfiguration/types/REGIONAL,value=EDGE
```

由于 put-rest-api 用于更新 API 定义，因此它不适用于更新 API 端点类型。

API Gateway 中用于 REST API 的方法

在 API Gateway 中，API 方法包含方法请求和方法响应。您可以设置 API 方法以定义客户端提交访问后端服务的请求时应该或必须执行的操作，并定义返回给客户端时收到的响应。对于输入，您可以为客户端选择方法请求参数或适用的负载，以在运行时提供必需或可选的数据。对于输出，您确定将方法响应状态代码、标头和适用的正文作为将后端响应数据映射到的目标，然后再将其返回到客户端。要帮助客户端开发人员理解 API 的行为以及输入和输出格式，您可以记录 API 并为无效请求提供正确的错误消息。

API 方法请求是 HTTP 请求。要设置方法请求，您需要配置 HTTP 方法（或动词）以及指向 API 资源、标头和适用查询字符串参数的路径。您还可在 HTTP 方法是 `POST`、`PUT` 或 `PATCH` 时配置负载。例如，要使用 PetStore 示例 API 检索宠物，您可以定义 `GET /pets/{petId}` 的 API 方法请求，其中 `{petId}` 是可在运行时获取数字的路径参数。

```
GET /pets/1
Host: apigateway.us-east-1.amazonaws.com
...
```

如果客户端指定了错误的路径，例如 `/pet/1` 或 `/pets/one`，而不是 `/pets/1`，系统就会抛出异常。

API 方法响应是具有指定状态代码的 HTTP 响应。对于非代理集成，您必须设置方法响应以指定映射的必需或可选目标。这些会将集成响应标头或正文转换为关联的方法响应标头或正文。映射可以像身份转换一样简单，它通过集成按原样传递标头或正文。例如，以下 `200` 方法响应显示了成功按原样传递集成响应的示例。

```
200 OK
Content-Type: application/json
...

{
    "id": "1",
    "type": "dog",
    "price": "$249.99"
}
```

原则上，您可以定义对应于来自后端的特定响应的方法响应。通常，这会涉及任何 2XX、4XX 和 5XX 响应。但是，这可能并不可行，因为您通常不可能提前知道后端可能返回的所有响应。实际上，您可以指定一个默认方法响应来处理来自后端的未知响应或未映射的响应。最好将 500 响应指定为默认响应。在任何情况下，您都必须为非代理集成设置至少一个方法响应。否则，API Gateway 将向客户端返回 500 错误响应，即使后端的请求成功也是如此。

要为 API 支持强类型开发工具包 (如 Java 开发工具包)，您应该为方法请求定义输入数据模型，并为方法响应定义输出数据模型。

先决条件

设置 API 方法之前，请验证以下内容：

- 方法在 API Gateway 中必须可用。按照教程：使用 HTTP 非代理集成构建 REST API中的说明进行操作。
- 如果您希望方法与 Lambda 函数进行通信，则必须已在 IAM 中创建 Lambda 调用角色和 Lambda 执行角色。您还必须创建将在 AWS Lambda 中与该方法进行通信的 Lambda 函数。要创建角色和函数，请使用选择 AWS Lambda 集成教程的为 Lambda 非代理集成创建 Lambda 函数中的说明。
- 如果您希望该方法与 HTTP 或 HTTP 代理集成进行通信，则必须已创建且有权访问将与您的方法进行通信的 HTTP 端点 URL。
- 确认 API Gateway 支持 HTTP 和 HTTP 代理端点的证书。有关详细信息，请参阅 API Gateway 支持的 HTTP 和 HTTP 代理集成证书的颁发机构。

> Note
>
> 使用 REST API 控制台创建方法时，您可以同时配置集成请求和方法请求。有关更多信息，请参阅 the section called “ 使用控制台设置集成请求”。

主题

- 在 API Gateway 中设置方法请求
- 在 API Gateway 中设置方法响应
- 使用 API Gateway 控制台设置方法

在 API Gateway 中设置方法请求

设置方法请求涉及在创建 RestApi 资源之后执行以下任务：

1. 创建新的 API 或选择现有的 API 资源实体。
2. 创建特定于新创建或选定的 API `Resource` 上的 HTTP 动词的 API 方法资源。此任务可进一步分为以下子任务：
    - 为方法请求添加 HTTP 方法
    - 配置请求参数
    - 定义请求正文的模型
    - 制定授权方案
    - 启用请求验证

您可以使用以下方法执行这些任务：

- API Gateway 控制台
- AWS CLI 命令 (create-resource 和 put-method)
- AWS 开发工具包函数（例如 Node.js 中的 createResource 和 putMethod ）
- API Gateway REST API（ resource:create 和 method:put ）。

主题

- 设置 API 资源
- 设置 HTTP 方法
- 设置方法请求参数
- 设置方法请求模型
- 设置方法请求授权
- 设置方法请求验证

设置 API 资源

在 API Gateway API 中，您通过 API 资源实体树公开可寻址的资源，树的根资源 (/) 位于层次结构顶部。根资源相对于 API 的基本 URL，其中包含 API 端点和阶段名称。在 API Gateway 控制台中，该基本 URI 称为调用 URI，会在 API 部署后在 API 的阶段编辑器中显示。

API 端点可以是默认主机名或自定义域名。默认主机名采用以下格式：

```
{api-id}.execute-api.{region}.amazonaws.com
```

在此格式中，*{api-id}* 表示 API Gateway 生成的 API 标识符。*{region}* 变量表示您在创建 API 时选择的 AWS 区域（例如 `us-east-1`）。自定义域名是有效 Internet 域的便于用户识别的任何名称。例如，如果您已经注册 Internet 域 `example.com`，任何 `*.example.com` 都是有效的自定义域名。有关更多信息，请参阅创建自定义域名。

对于 PetStore 示例 API，根资源 (/) 用于公开 Pet Store。`/pets` 资源表示 Pet Store 中提供的宠物集合。`/pets/{petId}` 公开指定标识符 (`petId`) 的单个宠物。`{petId}` 的路径参数是请求参数的一部分。

要设置 API 资源，您可以选择现有资源作为父资源，然后在此父资源下创建子资源。您开始时将根资源作为父资源，向此父资源添加资源，向这个子资源添加另一个资源作为新的父资源，依此类推，直到添加到其父标识符。然后将命名的资源添加到父资源。

借助 AWS CLI，您可以调用 `get-resources` 命令，以查找可供使用的 API 资源：

```
aws apigateway get-resources --rest-api-id <apiId> \
                             --region <region>
```

结果是当前可用的 API 资源的列表。对于我们的 PetStore 示例 API，此列表类似于以下内容：

```
{
    "items": [
        {
            "path": "/pets",
            "resourceMethods": {
                "GET": {}
            },
            "id": "6sxz2j",
            "pathPart": "pets",
            "parentId": "svzr2028x8"
```

```
        },
        {
            "path": "/pets/{petId}",
            "resourceMethods": {
                "GET": {}
            },
            "id": "rjkmth",
            "pathPart": "{petId}",
            "parentId": "6sxz2j"
        },
        {
            "path": "/",
            "id": "svzr2028x8"
        }
    ]
}
```

每个项目都列出了资源 (`id`) (根资源除外) 的标识符、其直接父级 (`parentId`) 以及资源名称 (`pathPart`)。根资源的特殊之处在于不具有任何父级。选择某个资源作为父资源之后，调用以下命令添加子资源。

```
aws apigateway create-resource --rest-api-id <apiId> \
                             --region <region> \
                             --parent-id <parentId> \
                             --path-part <resourceName>
```

例如，要在 PetStore 网站上添加要销售的宠物食品，通过将 `food` 设置为 `path-part` 并将 `food` 设置为 `parent-id` 来将 `svzr2028x8` 资源添加到根 (/)。结果类似于以下内容：

```
{
    "path": "/food",
    "pathPart": "food",
    "id": "xdsvhp",
    "parentId": "svzr2028x8"
}
```

使用代理资源来简化 API 设置

随着业务的增长，PetStore 所有者可能决定添加食物、玩具及其他宠物相关的商品进行销售。为此，您可在根资源下添加 `/food`、`/toys` 和其他资源。在每个销售类别下，您可能还需要添加更多资源，例如 `/food/{type}/{item}`、`/toys/{type}/{item}` 等。这很麻烦。如果您决定在资源路径

中添加一个中间层 `{subtype}` 以将路径层次结构更改为 `/food/{type}/{subtype}/{item}`、`/toys/{type}/{subtype}/{item}` 等，这些更改将会中断现有的 API 设置。为了避免这种情况，您可以使用 API Gateway 代理资源一次性公开一组 API 资源。

API Gateway 将代理资源定义为要在提交请求时指定的资源的占位符。代理资源用 `{proxy+}` 的特殊路径参数表示，该路径通常被称为“贪婪”路径参数。`+` 号表示其附加了哪些子资源。`/parent/{proxy+}` 占位符表示与 `/parent/*` 路径模式匹配的任何资源。“贪婪”路径参数名称 `proxy` 可以按照与处理常规路径参数名称相同的方法用其他字符串替换。

使用 AWS CLI，您可以调用以下命令来设置根 (`/{proxy+}`) 下的代理资源：

```
aws apigateway create-resource --rest-api-id <apiId> \
                               --region <region> \
                               --parent-id <rootResourceId> \
                               --path-part {proxy+}
```

结果类似于以下内容：

```
{
    "path": "/{proxy+}",
    "pathPart": "{proxy+}",
    "id": "234jdr",
    "parentId": "svzr2028x8"
}
```

对于 `PetStore` API 示例，您可以使用 `/{proxy+}` 表示 `/pets` 和 `/pets/{petId}`。此代理资源也可以引用任何其他 (现有或要添加的) 资源，例如 `/food/{type}/{item}`、`/toys/{type}/{item}` 等，或者 `/food/{type}/{subtype}/{item}`、`/toys/{type}/{subtype}/{item}` 等。后端开发人员确定资源层次结构，客户端开发人员负责了解此结构。API Gateway 会将客户端提交的内容传递到后端。

API 可以包含多个代理资源。例如，API 中允许包含以下代理资源。

```
/{proxy+}
/parent/{proxy+}
/parent/{child}/{proxy+}
```

如果代理资源具有非代理同级资源，则会从代理资源的表示形式中排除同级资源。在前面的示例中，`/{proxy+}` 是指除 `/parent[/*]` 资源之外的根资源下的任何资源。换言之，针对特定资源的方法请求优先于针对同一资源层次结构级别的通用资源的方法请求。

代理资源不能包含任何子资源。`{proxy+}` 后面的任何 API 资源都是冗余和不确定的资源。API 中不允许包含以下代理资源。

```
/{proxy+}/child
/parent/{proxy+}/{child}
/parent/{child}/{proxy+}/{grandchild+}
```

设置 HTTP 方法

API 方法请求通过 API Gateway 方法资源进行封装。要设置方法请求，您必须先实例化 `Method` 资源，设置至少一个 HTTP 方法和方法的授权类型。

API Gateway 与代理资源密切相关，支持 HTTP 方法 `ANY`。此 `ANY` 方法表示将在运行时提供的任何 HTTP 方法。它允许您将单个 API 方法设置用于 `DELETE`、`GET`、`HEAD`、`OPTIONS`、`PATCH`、`POST` 和 `PUT` 支持的所有 HTTP 方法。

您也可以在非代理资源上设置 `ANY` 方法。通过将 `ANY` 方法与代理资源组合使用，即可获得适用于针对 API 的任何资源所支持的所有 HTTP 方法的单个 API 方法设置。此外，后端可以发展变化，而不会中断现有的 API 设置。

设置 API 方法之前，请考虑谁可以调用此方法。根据您的计划设置授权类型。对于开放式访问，将其设置为 `NONE`。要使用 IAM 权限，请将授权类型设置为 `AWS_IAM`。要使用 Lambda 授权方函数，请将此属性设置为 `CUSTOM`。要使用 Amazon Cognito 用户池，请将授权类型设置为 `COGNITO_USER_POOLS`。

以下 AWS CLI 命令显示了如何针对指定资源 (`6sxz2j`) 为 `ANY` 动词创建使用 IAM 权限来控制其访问权限的方法请求。

```
aws apigateway put-method --rest-api-id vaz7da96z6 \
       --resource-id 6sxz2j \
       --http-method ANY \
       --authorization-type AWS_IAM \
       --region us-west-2
```

要创建使用不同授权类型的 API 方法请求，请参阅the section called “设置方法请求授权”。

设置方法请求参数

方法请求参数是一种客户端提供完成方法请求所需的输入数据或执行上下文的方式。方法参数可以是路径参数、标头或查询字符串参数。在方法请求设置期间，您必须声明必需的请求参数，使其可供客户端使用。对于非代理集成，您可以将这些请求参数转换为与后端要求兼容的形式。

例如，对于 `GET /pets/{petId}` 方法请求，`{petId}` 路径变量为必需的请求参数。您可在调用 AWS CLI 的 `put-method` 命令时声明此路径参数。下文对此进行了说明：

```
aws apigateway put-method --rest-api-id vaz7da96z6 \
       --resource-id rjkmth \
       --http-method GET \
       --authorization-type "NONE" \
       --region us-west-2 \
       --request-parameters method.request.path.petId=true
```

如果参数不是必需的，则可在 `false` 中将其设置为 `request-parameters`。例如，如果 `GET /pets` 方法使用 `type` 的可选查询字符串参数和 `breed` 的可选标头参数，您可以使用以下 CLI 命令声明它们，假设 `/pets` 资源 id 为 `6sxz2j`：

```
aws apigateway put-method --rest-api-id vaz7da96z6 \
       --resource-id 6sxz2j \
       --http-method GET \
       --authorization-type "NONE" \
       --region us-west-2 \
       --request-parameters
 method.request.querystring.type=false,method.request.header.breed=false
```

如果不使用这种缩写形式，您可以使用 JSON 字符串来设置 `request-parameters` 值：

```
'{"method.request.querystring.type":false,"method.request.header.breed":false}'
```

使用此设置，客户端可以按照类型查询宠物：

```
GET /pets?type=dog
```

客户端可以按照以下所示查询贵宾品种的狗：

```
GET /pets?type=dog
breed:poodle
```

有关如何将方法请求参数映射到集成请求参数的信息，请参阅the section called “集成”。

设置方法请求模型

对于可在负载中获取输入数据的 API 方法，您可以使用模型。模型采用 JSON 架构草案 4 表示，描述了请求正文的数据结构。借助模型，客户端可以确定如何构建方法请求负载以作为输入。更重要的

是，API Gateway 使用模型来验证请求、生成开发工具包以及初始化用于在 API Gateway 控制台中设置集成的映射模板。有关如何创建模型的信息，请参阅了解数据模型。

根据内容类型，该方法负载可以具有不同格式。模型根据应用负载的媒体类型来编制索引。API Gateway 使用 `Content-Type` 请求标头来确定内容类型。要设置方法请求模型，请在调用 AWS CLI `put-method` 命令时将 `"<media-type>":"<model-name>"` 格式的键/值对添加到 `requestModels` 映射。

要使用同一模型而不考虑内容类型，请指定 `$default` 作为键。

例如，要在 PetStore 示例 API 的 `POST /pets` 方法请求的 JSON 负载上设置模型，您可以调用以下 AWS CLI 命令：

```
aws apigateway put-method \
       --rest-api-id vaz7da96z6 \
       --resource-id 6sxz2j \
       --http-method POST \
       --authorization-type "NONE" \
       --region us-west-2 \
       --request-models '{"application/json":"petModel"}'
```

其中，`petModel` 是描述宠物的 `name` 资源的 `Model` 属性值。实际架构定义采用 `schema` 资源的 `Model` 属性的 JSON 字符串值表示。

在 Java 或 API 的其他强类型开发工具包中，输入数据将强制转换为派生自架构定义的 `petModel` 类。使用请求模型，生成的开发工具包中的输入数据将强制转换为派生自默认 `Empty` 模型的 `Empty` 类。在这种情况下，客户端无法实例化正确的数据类以提供所需的输入。

设置方法请求授权

要控制哪些人员可以调用 API 方法，您可在方法中配置授权类型。您可以使用此类型制定其中一个支持的授权方，包括 IAM 角色和策略 (`AWS_IAM`)、Amazon Cognito 用户池 (`COGNITO_USER_POOLS`) 或 Lambda 授权方 (`CUSTOM`)。

要使用 IAM 权限为 API 方法授予访问权限，请将 `authorization-type` 输入属性设置为 **`AWS_IAM`**。当您设置此选项时，API Gateway 将根据调用方的凭证验证调用方对请求的签名。如果经验证的用户有权调用此方法，它将接受请求。否则，它将拒绝请求，而调用方会收到未经授权的错误响应。除非调用方有权调用 API 方法，否则对该方法的调用不会成功。以下 IAM policy 向调用方授予权限，以调用在相同 AWS 账户 中创建的任何 API 方法：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "execute-api:Invoke"
            ],
            "Resource": "arn:aws:execute-api:*:*:*"
        }
    ]
}
```

有关更多信息，请参阅 the section called "使用 IAM 权限"。

目前，您只能向 API 所有者的 AWS 账户 内的用户、组和角色授予此策略。来自不同 AWS 账户 的用户只有在被允许代入 API 所有者的 AWS 账户 中的角色并具有调用 `execute-api:Invoke` 操作所需的权限时，才能调用 API 方法。有关跨账户权限的信息，请参阅使用 IAM 角色。

您可以使用 AWS CLI、AWS SDK 或 REST API 客户端（例如 Postman），这将实施签名版本 4（SigV4）签名。

要使用 Lambda 授权方来授予对 API 方法的访问权限，请将 `authorization-type` 输入属性设置为 `CUSTOM`，并将 `authorizer-id` 输入属性设置为已存在的 Lambda 授权方的 `id` 属性值。引用的 Lambda 授权方可以是 `TOKEN` 或 `REQUEST` 类型。有关创建 Lambda 授权方的信息，请参阅 the section called "使用 Lambda 授权方"。

要使用 Amazon Cognito 用户池来授予对 API 方法的访问权限，请将 `authorization-type` 输入属性设置为 `COGNITO_USER_POOLS`，并将 `authorizer-id` 输入属性设置为已创建的 `COGNITO_USER_POOLS` 授权方的 `id` 属性值。有关创建 Amazon Cognito 用户池授权方的信息，请参阅 the section called "使用 Amazon Cognito 用户池作为 REST API 的授权方"。

设置方法请求验证

您可在设置 API 方法请求时启用请求验证。您需要先创建请求验证程序：

```
aws apigateway create-request-validator \
    --rest-api-id 7zw9uyk9kl \
    --name bodyOnlyValidator \
    --validate-request-body  \
    --no-validate-request-parameters
```

此 CLI 命令创建仅限正文的请求验证程序。输出示例如下：

```
{
    "validateRequestParameters": false,
    "validateRequestBody": true,
    "id": "jgpyy6",
    "name": "bodyOnlyValidator"
}
```

使用此请求验证程序，您可在方法请求设置期间启用请求验证：

```
aws apigateway put-method \
    --rest-api-id 7zw9uyk9kl
    --region us-west-2
    --resource-id xdsvhp
    --http-method PUT
    --authorization-type "NONE"
    --request-parameters '{"method.request.querystring.type": false,
 "method.request.querystring.page":false}'
    --request-models '{"application/json":"petModel"}'
    --request-validator-id jgpyy6
```

要包含在请求验证中，必须将请求参数声明为必需参数。如果在请求验证中使用了该页面的查询字符串参数，则必须将前述示例的 `request-parameters` 映射指定为 `'{"method.request.querystring.type": false, "method.request.querystring.page":true}'`。

在 API Gateway 中设置方法响应

API 方法响应将客户端将收到 API 方法请求的输出进行封装。输出数据包括 HTTP 状态代码、部分标头，还可能包含正文。

使用非代理集成，可从相关的集成响应数据映射指定的响应参数和正文，也可以根据映射分配特定静态值。集成响应中已指定这些映射。映射可以是按原样传递集成响应的恒等变换。

使用代理集成，API Gateway 可自动将后端响应传递到方法响应。您无需设置 API 方法响应。但如果使用 Lambda 代理集成，Lambda 函数必须为 API Gateway 返回此输出格式的结果，才能将集成响应成功映射到方法响应。

通过编程，该方法响应设置相当于创建 API Gateway 的 MethodResponse 资源和设置 statusCode、responseParameters 和 responseModels 的属性。

为 API 方法设置状态代码时，您应该选择一个状态代码作为处理任何意料之外的集成响应的默认代码。将 500 设置默认状态代码比较合理，因为这相对于将未映射的响应转换为服务器端错误。出于说明原因，API Gateway 控制台将 200 响应设置为默认响应。但您可以将其重置为 500 响应。

要设置方法响应，您必须已创建方法请求。

设置方法响应状态代码

方法响应的状态代码定义了响应类型。例如，响应 200、400 和 500 分别表示响应成功、客户端错误和服务器端错误。

要设置方法响应状态代码，请将 `statusCode` 属性设置为 HTTP 状态代码。以下 AWS CLI 命令将创建 200 的方法响应。

```
aws apigateway put-method-response \
       --region us-west-2  \
       --rest-api-id vaz7da96z6 \
       --resource-id 6sxz2j \
       --http-method GET \
       --status-code 200
```

设置方法响应参数

方法响应参数定义客户端在哪些标头中收到相关方法请求的响应。响应参数还会根据 API 方法的集成响应中规定的映射指定 API Gateway 将集成响应参数映射到的目标。

要设置方法请求参数，请添加到 `responseParameters` 格式的 `MethodResponse` 键/值对的 `"{parameter-name}":"{boolean}"` 映射。以下 CLI 命令显示了设置 `my-header` 标头的示例：

```
aws apigateway put-method-response \
        --region us-west-2 \
        --rest-api-id vaz7da96z6 \
        --resource-id 6sxz2j \
        --http-method GET \
        --status-code 200  \
        --response-parameters method.response.header.my-header=false
```

设置方法响应模型

方法响应模型定义了方法响应正文的格式。设置响应模型之前，必须先在 API Gateway 中创建模型。为此，可以调用 `create-model` 命令。以下示例显示了如何创建能够描述 `PetStorePet` 方法请求的响应正文的 `GET /pets/{petId}` 模型。

```
aws apigateway create-model \
    --region us-west-2 \
    --rest-api-id vaz7da96z6 \
    --content-type application/json \
    --name PetStorePet \
    --schema '{ \
                  "$schema": "http://json-schema.org/draft-04/schema#", \
                  "title": "PetStorePet", \
                  "type": "object", \
                  "properties": { \
                    "id": { "type": "number" }, \
                    "type": { "type": "string" }, \
                    "price": { "type": "number" } \
                  } \
              }'
```

结果将创建为 API Gateway `Model` 资源。

要设置方法响应模型以定义有效负载格式，请将 "application/json":"PetStorePet" 密钥值对添加至 `MethodResponse` 资源的 `requestModels` 地图。`put-method-response` 的以下 AWS CLI 命令显示了如何执行此操作：

```
aws apigateway put-method-response \
        --region us-west-2 \
        --rest-api-id vaz7da96z6 \
        --resource-id 6sxz2j \
        --http-method GET \
        --status-code 200  \
        --response-parameters method.response.header.my-header=false \
        --response-models '{"application/json":"PetStorePet"}'
```

为 API 生成强类型开发工具包时，必须设置方法响应模型。它可确保输出将在 Java 或 Objective-C 中转换为适当的类。在其他情况下，设置模型是可选操作。

使用 API Gateway 控制台设置方法

使用 REST API 控制台创建方法时，您可以同时配置集成请求和方法请求。默认情况下，API Gateway 会为您的方法创建 `200` 方法响应。

以下说明介绍如何编辑方法请求设置，以及如何为您的方法创建其它方法响应。

主题

- 在 API Gateway 控制台中编辑 API Gateway 方法请求
- 使用 API Gateway 控制台设置 API Gateway 方法响应

在 API Gateway 控制台中编辑 API Gateway 方法请求

这些说明假设您已创建了方法请求。有关如何创建方法的更多信息，请参阅the section called " 使用控制台设置集成请求"。

1. 在资源窗格中，选择方法，然后选择方法请求选项卡。
2. 在方法请求设置部分中，选择编辑。
3. 对于授权，选择一个可用的授权方。
    a. 要为任何用户启用对此方法的开放式访问，请选择无。如果未更改默认设置，则可跳过此步骤。
    b. 要使用 IAM 权限来控制客户端对方法的访问权限，请选择 `AWS_IAM`。如果选择此选项，只允许具有已附加正确 IAM 策略的 IAM 角色的用户调用此方法。

       要创建 IAM 角色，请指定类似于以下格式的访问策略：

       ```
       {
         "Version": "2012-10-17",
         "Statement": [
           {
             "Effect": "Allow",
             "Action": [
               "execute-api:Invoke"
             ],
             "Resource": [
               "resource-statement"
             ]
           }
         ]
       }
       ```

       在此访问策略中，*`resource-statement`* 是方法的 ARN。您可以通过在资源页面上选择方法来找到该方法的 ARN。有关设置 IAM 权限的更多信息，请参阅使用 IAM 权限控制对 API 的访问。

要创建 IAM 角色，您可以调整以下教程中的说明：???。

   c. 要使用 Lambda 授权方，请选择令牌或请求授权方。创建 Lambda 授权方，让此选择显示在下拉菜单中。有关如何创建 Lambda 授权方的信息，请参阅 使用 API Gateway Lambda 授权方。

   d. 要使用 Amazon Cognito 用户池，请在 Cognito 用户池授权方下选择可用的用户池。在 Amazon Cognito 中创建一个用户群体，并在 API Gateway 中创建一个 Amazon Cognito 用户群体授权方，以便在下拉菜单中显示此选择。有关如何创建 Amazon Cognito 用户池授权方的信息，请参阅 使用 Amazon Cognito 用户池作为授权方控制对 REST API 的访问。

4. 要指定请求验证，请从请求验证程序下拉菜单中选择一个值。要关闭请求验证，请选择无。有关各选项的更多信息，请参阅在 API Gateway 中使用请求验证。
5. 选择需要 API 密钥以需要 API 密钥。启用后，将在使用计划中使用 API 密钥以限制客户端流量。
6. （可选）要在 API Gateway 生成的此 API 的 Java SDK 中分配操作名称，请为输入名称输入一个名称。例如，对于 `GET /pets/{petId}` 的方法请求，相对应的默认 Java 开发工具包操作名称为 `GetPetsPetId`。此名称由此方法的 HTTP 动词 (`GET`) 和资源路径变量名称 (`Pets` 和 `PetId`) 构建而成。如果您将操作名称设置为 `getPetById`，开发工具包操作名称将变为 `GetPetById`。
7. 要向方法中添加查询字符串参数，请执行以下操作：

   a. 选择 URL 查询字符串参数，然后选择添加查询字符串。

   b. 对于名称，输入查询字符串参数的名称。

   c. 如果要将新创建的查询字符串参数用于请求验证，请选择必填。有关请求验证的更多信息，请参阅在 API Gateway 中使用请求验证。

   d. 如果要将新创建的查询字符串参数用作缓存密钥的一部分，请选中缓存。有关缓存的更多信息，请参阅 将方法或集成参数用作索引缓存响应的缓存键。

   要移除查询字符串参数，请选择移除。

8. 要向方法中添加标头参数，请执行以下操作：

   a. 选择 HTTP 请求标头，然后选择添加标头。

   b. 对于名称，输入标头的名称。

   c. 如果要将新创建的标头用于请求验证，请选择必填。有关请求验证的更多信息，请参阅在 API Gateway 中使用请求验证。

   d. 如果要将新创建的标头用作缓存密钥的一部分，请选中缓存。有关缓存的更多信息，请参阅 将方法或集成参数用作索引缓存响应的缓存键。

要移除标头，请选择移除。

9. 要声明包含 POST、PUT 或 PATCH HTTP 动词的方法请求的负载格式，请选择请求正文，然后执行以下操作：

   a. 选择添加模型。

   b. 对于内容类型，输入 MIME 类型（例如 `application/json`）。

   c. 对于模型，请从下拉菜单中选择一个模型。该 API 目前可用的模型包括默认 `Empty` 和 `Error` 模型以及您创建和添加到 API 的模型集合的任何模型。有关创建模型的更多信息，请参阅了解数据模型。

   > **Note**
   >
   > 模型适用于向客户端通知预期数据格式的负载。这对生成骨骼映射模板非常有帮助。请务必使用 Java、C #、Objective-C 和 Swift 等语言编程生成 API 的强类型开发工具包。仅在已针对负载启用请求验证时才需要该开发工具包。

10. 选择保存。

使用 API Gateway 控制台设置 API Gateway 方法响应

API 方法可以有一个或多个响应。每个响应都通过 HTTP 状态代码编制索引。默认情况下，API Gateway 控制台将 `200` 响应添加到方法响应。您可以对其进行修改，例如让该方法返回 `201`。还可以添加其他响应，例如添加 `409` 表示拒绝访问，添加 `500` 表示所用的阶段变量未初始化。

要使用 API Gateway 控制台修改、删除或添加对 API 方法的响应，请按照以下说明操作。

1. 在资源窗格中，选择方法，然后选择方法响应选项卡。您可能需要选择右箭头按钮，以显示该选项卡。

2. 在方法响应设置部分，选择创建响应。

3. 对于 HTTP 状态代码，输入 HTTP 状态代码，例如`200`、`400` 或 `500`。

   如果后端返回的响应未定义相应的方法响应，API Gateway 就无法将响应返回给客户端。而是返回 `500 Internal server error` 错误响应。

4. 选择添加标头。

5. 对于标头名称，输入名称。

要将标头从后端返回给客户端，请在方法响应中添加标头。

6. 选择添加模型以定义方法响应正文的格式。

   为内容类型输入响应负载的媒体类型，然后从模型下拉菜单中选择模型。

7. 选择保存。

要修改现有响应，请导航到您的方法响应，然后选择编辑。要更改 HTTP 状态代码，请选择删除并创建新的方法响应。

对于从后端返回的每个响应，您必须将兼容的响应配置为方法响应。但配置方法响应标头和负载模型是可选的，除非您先将后端的结果映射到方法响应，然后再返回给客户端。此外，如果您要为 API 生成强类型开发工具包，方法响应负载模型就非常重要。

在 API Gateway 中控制和管理对 REST API 的访问

API Gateway 支持多种用于控制和管理对 API 的访问的机制：

您可以使用以下机制进行身份验证和授权：

- 资源策略允许您创建基于资源的策略，以允许或拒绝从指定的源 IP 地址或 VPC 终端节点访问您的 API 和方法。有关更多信息，请参阅 the section called “使用 API Gateway 资源策略”。
- 标准的AWS IAM 角色和策略提供可应用于整个 API 集或各个方法的灵活且可靠的访问控制。可以使用 IAM 角色和策略来控制哪些人可以创建和管理您的 API，以及谁可以调用它们。有关更多信息，请参阅 the section called “使用 IAM 权限”。
- IAM 标签 可与 IAM 策略结合使用来控制访问权限。有关更多信息，请参阅 the section called “基于属性的访问控制”。
- 接口 VPC 终端节点的终端节点策略允许您将 IAM 资源策略附加到接口 VPC 终端节点，用于改进私有 API 的安全性。有关更多信息，请参阅 the section called “为私有 API 使用 VPC 端点策略”。
- Lambda 授权方 是 Lambda 函数，它们使用持有者令牌身份验证以及由标头、路径、查询字符串、阶段变量或上下文变量请求参数描述的信息来控制对 REST API 方法的访问。Lambda 授权方用于控制谁可以调用 REST API 方法。有关更多信息，请参阅 the section called “使用 Lambda 授权方”。
- Amazon Cognito 用户池 可让您为 REST API 创建可自定义的身份验证和授权解决方案。Amazon Cognito 用户池用于控制谁可以调用 REST API 方法。有关更多信息，请参阅 the section called “使用 Amazon Cognito 用户池作为 REST API 的授权方”。

您可以使用以下机制执行与访问控制相关的其他任务：

- 跨源资源共享 (CORS) 可让您控制您的 REST API 响应跨域资源请求的方式。有关更多信息，请参阅 the section called “CORS”。
- 客户端 SSL 证书 可用于验证发送到后端系统的 HTTP 请求是否来自 API Gateway。有关更多信息，请参阅 the section called “客户端证书”。
- AWS WAF 可用于保护您的 API Gateway API 免受常见 Web 漏洞攻击。有关更多信息，请参阅 the section called “AWS WAF”。

您可以使用以下机制跟踪和限制您已向授权客户端授予的访问权限：

- 使用计划 可让您向客户提供 API 密钥，然后跟踪和限制每个 API 密钥的 API 阶段和方法的使用。有关更多信息，请参阅 the section called “使用计划”。

使用 API Gateway 资源策略控制对 API 的访问

Amazon API Gateway 资源策略是您附加到 API 的 JSON 策略文档，用于控制指定的主体（通常是 IAM 角色或组）能否调用 API。您可以使用 API Gateway 资源策略来允许 API 安全地被以下对象调用：

- 指定的 AWS 账户中的用户。
- 指定源 IP 地址范围或 CIDR 块
- 指定的 Virtual Private Cloud (VPC) 或 VPC 端点（在任何账户中）。

您可以使用 AWS Management Console、AWS CLI 或 AWS SDK，为 API Gateway 中的任何 API 端点类型附加资源策略。对于 私有 API，您可以将资源策略与 VPC 端点策略一起使用，控制委托人有权访问哪些资源和操作。有关更多信息，请参阅 the section called “为私有 API 使用 VPC 端点策略”。

API Gateway 资源策略与基于 IAM 身份的策略不同。基于 IAM 身份的策略附加到 IAM 用户、组或角色并定义这些身份能够对哪些资源执行哪些操作。API Gateway 资源策略是附加到资源的。您可以同时使用 API Gateway 资源策略和 IAM 策略。有关更多信息，请参阅基于身份的策略和基于资源的策略。

主题

- Amazon API Gateway 的访问策略语言概述
- API Gateway 资源策略如何影响授权工作流程
- API Gateway 资源策略示例

- 创建 API Gateway 资源策略并将其附加到 API
- 可在 API Gateway 资源策略中使用的 AWS 条件键

Amazon API Gateway 的访问策略语言概述

本页介绍 Amazon API Gateway 资源策略中使用的基本元素。

指定资源策略所用的语法与 IAM 策略相同。如需全面了解策略语言，请参阅 IAM 用户指南中的 IAM 策略概述和 AWS Identity and Access Management 策略参考。

有关 AWS 服务如何决定是应允许还是拒绝指定请求的更多信息，请参阅决定是允许还是拒绝请求。

访问策略中的常用元素

就其最基本的意义而言，资源策略包含以下元素：

- 资源 – API 是您能够允许或拒绝权限的 Amazon API Gateway 资源。在策略中，使用 Amazon Resource Name (ARN) 标识资源。您还可以使用缩写语法，当您保存资源策略时，API Gateway 会自动将其扩展为完整的 ARN。要了解更多信息，请参阅 API Gateway 资源策略示例。

  有关完整 `Resource` 元素的格式，请参阅 在 API Gateway 中执行 API 的权限的 Resource 格式。
- 操作 – 对于每个资源，Amazon API Gateway 支持一组操作。您可使用操作关键字标识允许（或拒绝）的资源操作。

  例如，`execute-api:Invoke` 权限将允许在客户端请求时调用 API 的用户权限。

  有关 `Action` 元素的格式，请参阅在 API Gateway 中执行 API 的权限的 Action 格式。
- 效果 – 当用户请求特定操作（可以是 `Allow` 或 `Deny`）时的效果。您也可显式拒绝对资源的访问，这样可确保用户无法访问该资源，即使有其他策略授予了访问权限的情况下也是如此。

> **Note**
>
> “隐式拒绝”与“默认拒绝”相同。
> “隐式拒绝”与“明确拒绝”不同。有关更多信息，请参阅“默认拒绝”与“显式拒绝”的区别。

- 主体 – 允许访问语句中的操作和资源的账户或用户。在资源策略中，主体是接收此权限的用户或账户。

下面的资源策略示例展示了上述常用策略元素。该策略向其源 IP 地址位于地址块 *123.4.5.6/24* 内的任何用户授予对指定 *region* 中的指定 *account-id* 下的 API 的访问权限。如果用户的源 IP 不在该范围内，该策略将拒绝对 API 的所有访问。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "arn:aws:execute-api:region:account-id:*"
        },
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "arn:aws:execute-api:region:account-id:*",
            "Condition": {
                "NotIpAddress": {
                    "aws:SourceIp": "123.4.5.6/24"
                }
            }
        }
    ]
}
```

API Gateway 资源策略如何影响授权工作流程

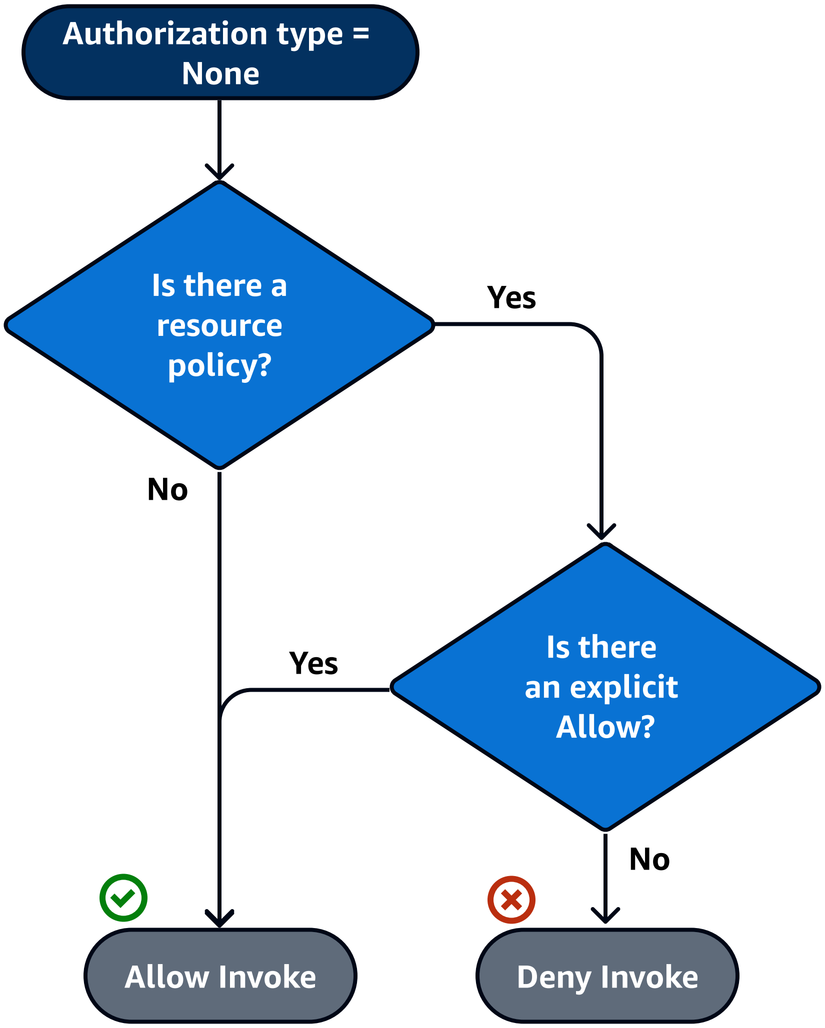
当 API Gateway 评估附加到您的 API 的资源策略时，结果会受到您为 API 定义的身份验证类型的影响，如以下章节的流程图所示。

主题

- 仅限 API Gateway 资源策略
- Lambda 授权方和资源策略
- IAM 身份验证和资源策略
- Amazon Cognito 身份验证和资源策略
- 策略评估结果表

仅限 API Gateway 资源策略

在此工作流中，API Gateway 资源策略附加至 API，但没有为 API 定义任何身份验证类型。对策略的评估将涉及根据调用方的入站标准寻求明确的许可。隐式拒绝或任何明确拒绝都将导致拒绝调用方。



以下是此类资源策略的一个示例。

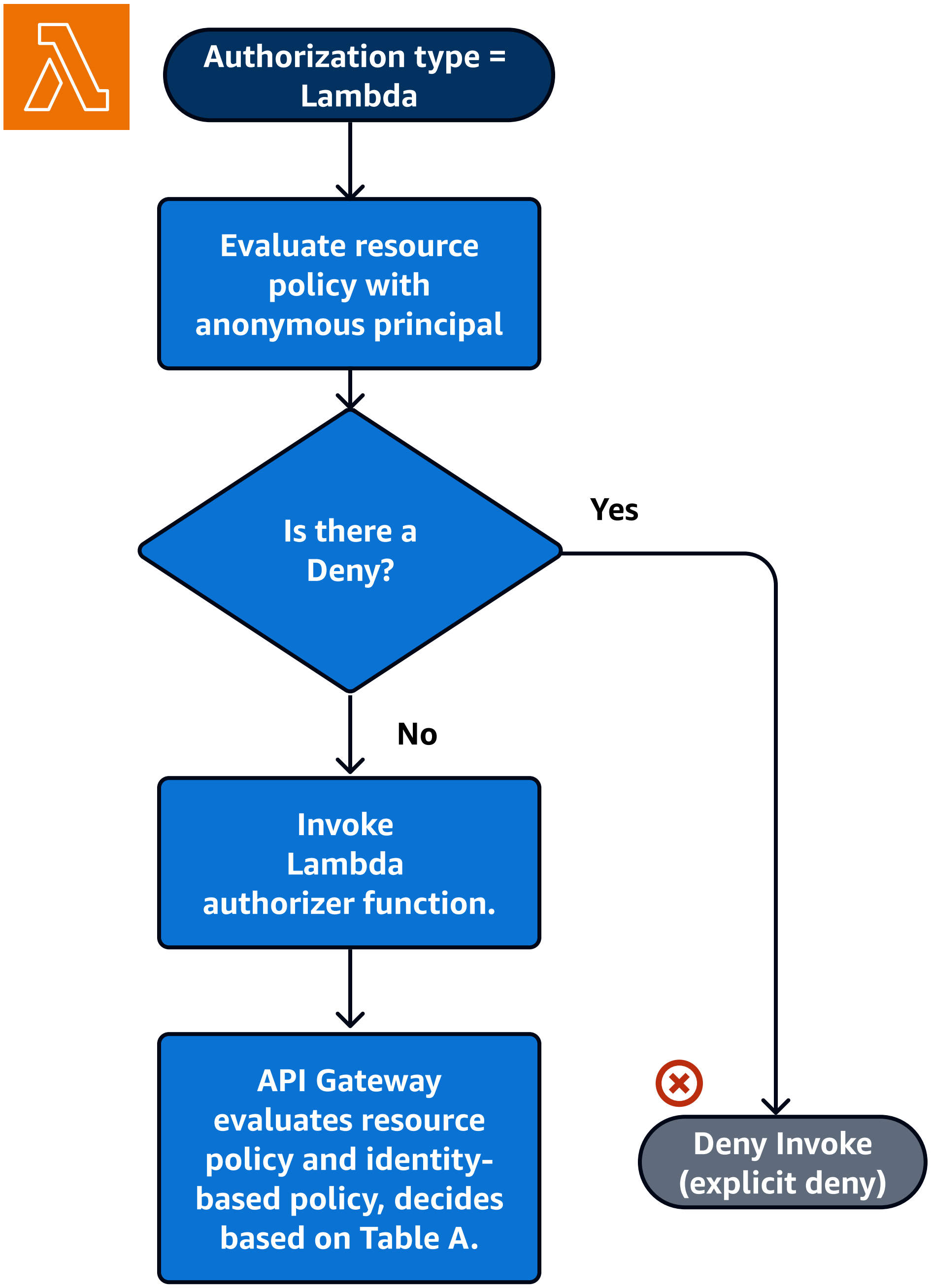
```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "arn:aws:execute-api:region:account-id:api-id/",
            "Condition": {
```

```
                    "IpAddress": {
                        "aws:SourceIp": ["192.0.2.0/24", "198.51.100.0/24" ]
                    }
                }
            }
        ]
    }
```

Lambda 授权方和资源策略

在此工作流程中，除了资源策略外，还为 API 配置了 Lambda 授权方。将在两个阶段中对资源策略进行评估。在调用 Lambda 授权方之前，API Gateway 首先评估策略并检查是否存在任何明确拒绝。一经发现，将立即拒绝调用方访问。否则，会调用 Lambda 授权方，它将返回策略文档，对该文档与资源策略进行评估。结果根据表 A 确定。

以下示例资源策略仅允许从 VPC 端点调用，其 VPC 端点 ID 为 *vpce-1a2b3c4d*。在“预身份验证”评估期间，只有来自示例中所述的 VPC 端点的调用才允许向前推进并评估 Lambda 授权方。阻止所有剩余的调用。



```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
```

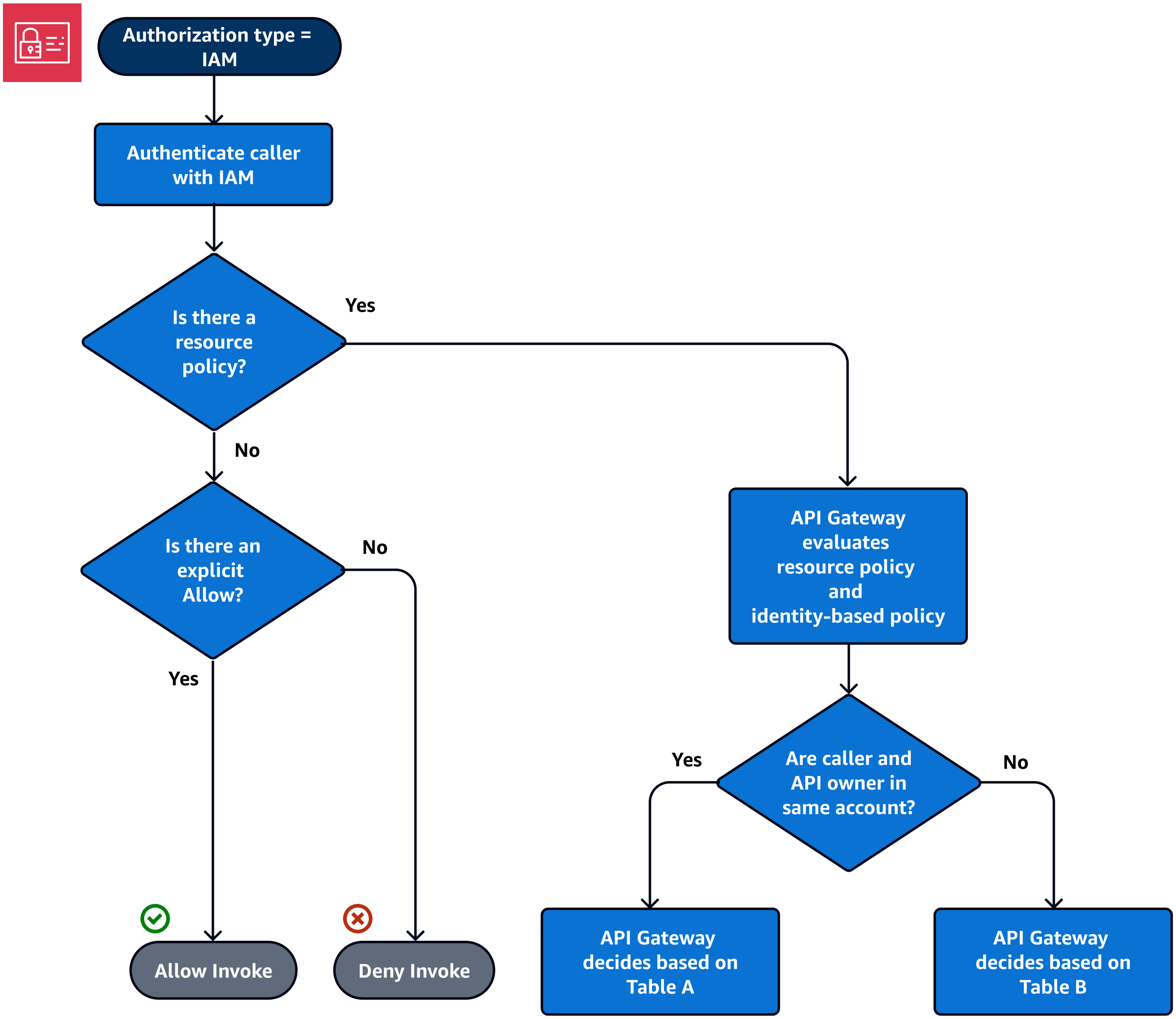
```
                "arn:aws:execute-api:region:account-id:api-id/"
            ],
            "Condition" : {
                "StringNotEquals": {
                    "aws:SourceVpce": "vpce-1a2b3c4d"
                }
            }
        }
    ]
}
```

IAM 身份验证和资源策略

在此工作流中，除资源策略外，还为 API 配置 IAM 身份验证。使用 IAM 服务对用户进行身份验证后，API 将同时评估附加到用户的策略以及资源策略。结果因调用方是在同一 AWS 账户 中还是在与 API 所有者不同的 AWS 账户 中而异。

如果调用方和 API 所有者来自不同的账户，则 IAM policy 和资源策略都明确允许调用方继续操作。有关更多信息，请参阅表 B。

然而，如果调用方和 API 所有者在同一 AWS 账户中，则 IAM 用户策略或资源策略必须明确允许调用方继续操作。有关更多信息，请参阅表 A。



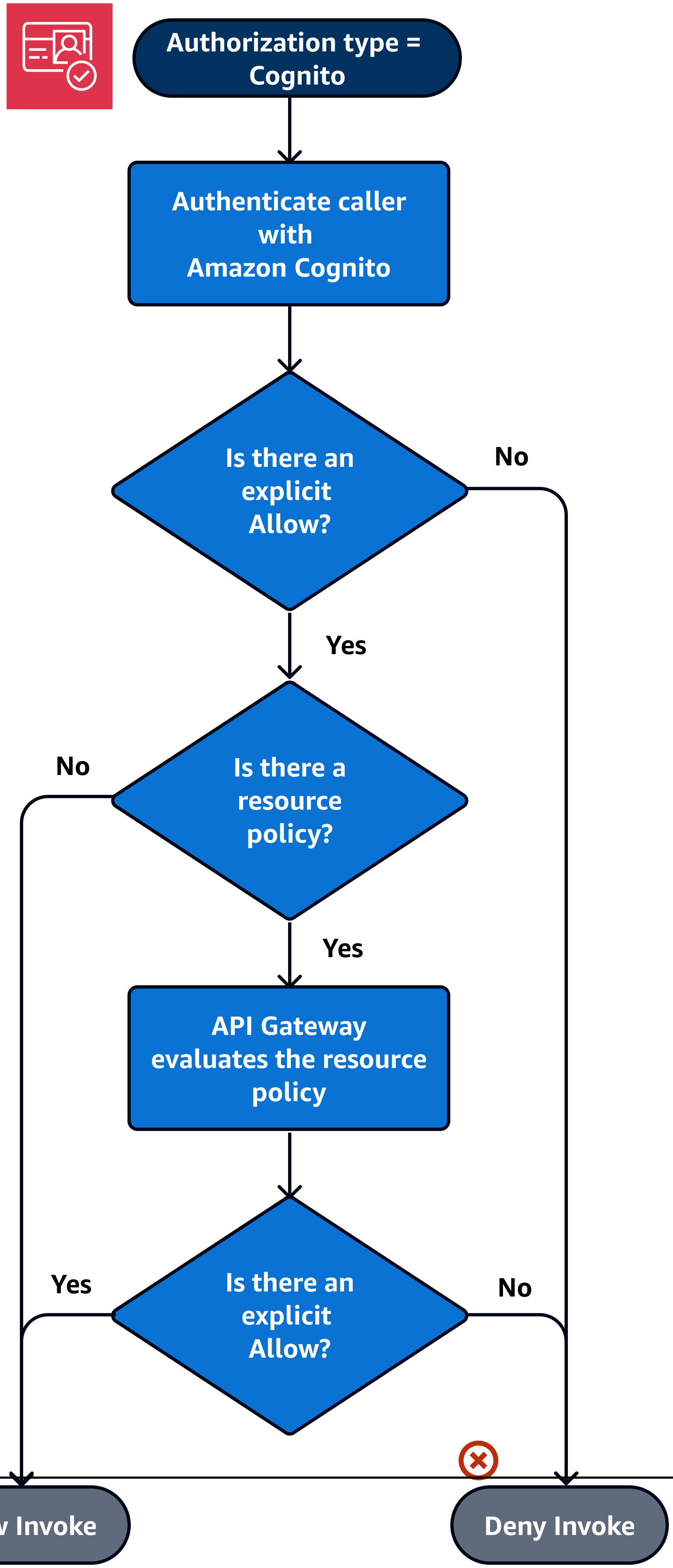
以下是跨账户资源策略的一个示例。假定 IAM policy 包含允许效果，此资源策略将仅允许来自 VPC ID 为 *vpc-2f09a348* 的 VPC 的调用。有关更多信息，请参阅表 B。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
```

```
                "arn:aws:execute-api:region:account-id:api-id/"
            ],
            "Condition" : {
                "StringEquals": {
                    "aws:SourceVpc": "vpc-2f09a348"
                    }
            }
        }
    ]
}
```

Amazon Cognito 身份验证和资源策略

在此工作流程中，除了资源策略之外，还为 API 配置了 Amazon Cognito 用户池。API Gateway 首先尝试通过 Amazon Cognito 对调用方进行身份验证。这通常通过调用方提供的 JWT 令牌执行。如果身份验证成功，则资源策略将被独立评估，且需要显示允许。拒绝或“不允许也不拒绝”将导致拒绝。下面是资源策略的一个示例，可以与 Amazon Cognito 用户池一起使用。


Authorization type = Cognito
Authenticate caller with Amazon Cognito
Is there an explicit Allow?
No
Yes
Is there a resource policy?
No
Yes
API Gateway evaluates the resource policy
Is there an explicit Allow?
Yes
No
Allow Invoke
Deny Invoke

下面是资源策略的一个示例，该策略只允许从指定的源 IP 调用（假定 Amazon Cognito 身份验证令牌包含允许）。有关更多信息，请参阅表 B。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": "arn:aws:execute-api:region:account-id:api-id/",
            "Condition": {
                "IpAddress": {
                    "aws:SourceIp": ["192.0.2.0/24", "198.51.100.0/24" ]
                }
            }
        }
    ]
}
```

策略评估结果表

表 A 列出了对 API Gateway API 的访问由 IAM 策略（或 Lambda 授权方）和 API Gateway 资源策略（两者都位于同一 AWS 账户中）控制时产生的行为。

表 A：账户 A 调用账户 A 拥有的 API

IAM 策略（或 Lambda 授权方）	API Gateway 资源策略	产生的行为
允许	允许	允许
允许	既不允许也不拒绝	允许
允许	拒绝	显式拒绝
既不允许也不拒绝	允许	允许
既不允许也不拒绝	既不允许也不拒绝	隐式拒绝
既不允许也不拒绝	拒绝	显式拒绝

IAM 策略（或 Lambda 授权方）	API Gateway 资源策略	产生的行为
拒绝	允许	显式拒绝
拒绝	既不允许也不拒绝	显式拒绝
拒绝	拒绝	显式拒绝

表 B 列出了对 API Gateway API 的访问由 IAM 策略（或 Amazon Cognito 用户池授权方）和 API Gateway 资源策略（两者位于不同 AWS 账户中）控制时产生的行为。如果其中一个静默（既不允许也不拒绝），则跨账户访问会被拒绝。这是因为跨账户访问要求资源策略和 IAM 策略（或 Amazon Cognito 用户池授权方）均明确授予访问权限。

表 B：账户 B 调用账户 A 所有的 API

IAM 策略（或 Amazon Cognito 用户池授权方）	API Gateway 资源策略	产生的行为
允许	允许	允许
允许	既不允许也不拒绝	隐式拒绝
允许	拒绝	显式拒绝
既不允许也不拒绝	允许	隐式拒绝
既不允许也不拒绝	既不允许也不拒绝	隐式拒绝
既不允许也不拒绝	拒绝	显式拒绝
拒绝	允许	显式拒绝
拒绝	既不允许也不拒绝	显式拒绝
拒绝	拒绝	显式拒绝

API Gateway 资源策略示例

此页面介绍 API Gateway 资源策略的几个典型使用案例。

以下示例策略使用简化语法来指定 API 资源。此简化语法是一种缩写方式，通过这种方式，您可以引用 API 资源，而不指定完整的 Amazon Resource Name (ARN)。当您保存策略时，API Gateway 会将缩写语法转换为完整 ARN。例如，您可以在资源策略中指定资源 `execute-api:/stage-name/GET/pets`。当您保存资源策略时， API Gateway 会将资源转换为 `arn:aws:execute-api:us-east-2:123456789012:aabbccddee/stage-name/GET/pets`。API Gateway 使用当前区域、您的 AWS 账户 ID 以及与资源策略关联的 REST API 的 ID 来构建完整的 ARN。您可以使用 `execute-api:/*` 来表示当前 API 中的所有阶段、方法和路径。有关访问策略语言的更多信息，请参阅 Amazon API Gateway 的访问策略语言概述。

主题

- 示例：允许另一个 AWS 账户中的角色使用 API
- 示例：基于源 IP 地址或范围拒绝 API 流量
- 示例：使用私有 API 时，基于源 IP 地址或范围拒绝 API 流量
- 示例：允许基于源 VPC 或 VPC 端点的私有 API 流量

示例：允许另一个 AWS 账户中的角色使用 API

以下示例资源策略通过签名版本 4（SigV4）协议将一个 AWS 账户中的 API 访问权限授予位于另一个 AWS 账户中的两个角色。具体而言，向由 *account-id-2* 标识的 AWS 账户的开发人员和管理员角色授予了 `execute-api:Invoke` 操作权限，允许他们对您 AWS 账户中的 `pets` 资源（API）执行 `GET` 操作。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::account-id-2:role/developer",
                    "arn:aws:iam::account-id-2:role/Admin"
                ]
            },
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/stage/GET/pets"
            ]
        }
    ]
```

```
}
```

示例：基于源 IP 地址或范围拒绝 API 流量

以下资源策略示例拒绝（阻止）从两个指定源 IP 地址块向 API 传入流量。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ]
        },
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
               "execute-api:/*"
            ],
            "Condition" : {
                "IpAddress": {
                    "aws:SourceIp": ["192.0.2.0/24", "198.51.100.0/24" ]
                }
            }
        }
    ]
}
```

示例：使用私有 API 时，基于源 IP 地址或范围拒绝 API 流量

以下资源策略示例拒绝（阻止）两个指定源 IP 地址块向私有 API 传入流量。使用私有 API 时，`execute-api` VPC 端点重新写入原始源 IP 地址。`aws:VpcSourceIp` 条件根据原始请求方 IP 地址筛选请求。

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ]
        },
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
               "execute-api:/*"
            ],
            "Condition" : {
                "IpAddress": {
                    "aws:VpcSourceIp": ["192.0.2.0/24", "198.51.100.0/24"]
                }
            }
        }
    ]
}
```

示例：允许基于源 VPC 或 VPC 端点的私有 API 流量

以下示例资源策略允许仅从指定的 Virtual Private Cloud (VPC) 或 VPC 端点传入到私有 API 的流量。

此示例资源策略指定源 VPC：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ]
        },
        {
            "Effect": "Deny",
```

```
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ],
            "Condition" : {
                "StringNotEquals": {
                   "aws:SourceVpc": "vpc-1a2b3c4d"
                }
            }
        }
    ]
}
```

此示例资源策略指定源 VPC 端点：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ]
        },
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ],
            "Condition" : {
                "StringNotEquals": {
                    "aws:SourceVpce": "vpce-1a2b3c4d"
                }
            }
        }
    ]
}
```

创建 API Gateway 资源策略并将其附加到 API

要允许用户通过调用 API 执行服务访问您的 API，您必须创建 API Gateway 资源策略，并将该策略附加到 API。当您将策略附加到 API 时，它会将该策略中的权限应用于 API 中的方法。如果您更新资源策略，您将需要部署 API。

主题

- 先决条件
- 将资源策略附加到 API Gateway API
- 排除资源策略故障

先决条件

要更新 API Gateway 资源策略，您将需要 `apigateway:UpdateRestApiPolicy` 权限和 `apigateway:PATCH` 权限。

对于边缘优化的 API 或区域 API，您可以在创建 API 时或者在部署 API 后，将资源策略附加到该 API。对于私有 API，如果没有资源策略，就无法部署 API。有关更多信息，请参阅 the section called “私有 REST API”。

将资源策略附加到 API Gateway API

以下过程说明如何将资源策略附加到 API Gateway API。

AWS Management Console

将资源策略附加到 API Gateway API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 在主导航窗格中，选择资源策略。
4. 选择创建策略。
5. （可选）选择选择模板以生成示例策略。

   在策略示例中，占位符括在双大括号 (`"{{placeholder}}"`) 内。将每个占位符（包括大括号）替换为必要的信息。
6. 如果您没有使用任何一个模板示例，则输入您的资源策略。

7. 选择 Save changes（保存更改）。

如果先前已在 API Gateway 控制台中部署了 API，则需要重新进行部署以使资源策略生效。

AWS CLI

要使用 AWS CLI 创建新的 API 并向其附加资源策略，请按以下所示调用 create-rest-api 命令：

```
aws apigateway create-rest-api \
    --name "api-name" \
    --policy "{\"jsonEscapedPolicyDocument\"}"
```

要使用 AWS CLI 将资源策略附加到现有 API，请按以下所示调用 update-rest-api 命令：

```
aws apigateway update-rest-api \
    --rest-api-id api-id \
    --patch-operations op=replace,path=/
policy,value='"{\"jsonEscapedPolicyDocument\"}"'
```

AWS CloudFormation

您可以使用 AWS CloudFormation 通过资源策略创建 API。以下示例使用示例资源策略the section called “示例：基于源 IP 地址或范围拒绝 API 流量”创建 REST API。

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  Api:
    Type: 'AWS::ApiGateway::RestApi'
    Properties:
      Name: testapi
      Policy:
        Statement:
          - Action: 'execute-api:Invoke'
            Effect: Allow
            Principal: '*'
            Resource: 'execute-api/*'
          - Action: 'execute-api:Invoke'
            Effect: Deny
            Principal: '*'
            Resource: 'execute-api/*'
            Condition:
              IpAddress:
```

```
                'aws:SourceIp': ["192.0.2.0/24", "198.51.100.0/24" ]
        Version: 2012-10-17
  Resource:
    Type: 'AWS::ApiGateway::Resource'
    Properties:
      RestApiId: !Ref Api
      ParentId: !GetAtt Api.RootResourceId
      PathPart: 'helloworld'
  MethodGet:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref Resource
      HttpMethod: GET
      ApiKeyRequired: false
      AuthorizationType: NONE
      Integration:
        Type: MOCK
  ApiDeployment:
    Type: 'AWS::ApiGateway::Deployment'
    DependsOn:
      - MethodGet
    Properties:
      RestApiId: !Ref Api
      StageName: test
```

排除资源策略故障

以下故障排除指南可能有助于解决资源策略问题。

我的 API 返回 {"Message":"User: anonymous is not authorized to perform: execute-api:Invoke on resource: arn:aws:execute-api:us-east-1:********/****/****/"}

在资源策略中，如果您将主体设置为 AWS 主体，如下所示：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            ""Principal": {
                "AWS": [
```

```
                    "arn:aws:iam::account-id:role/developer",
                    "arn:aws:iam::account-id:role/Admin"
                ]
            },
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ]
        },
        ...
        }
```

您必须对 API 中的每个方法使用 `AWS_IAM` 授权，否则 API 会返回之前的错误消息。有关如何为方法开启 `AWS_IAM` 授权的更多说明，请参阅 the section called “方法”。

我的资源策略未更新

如果在创建 API 后更新资源策略，您将需要附加更新的策略，然后部署 API 以传播更改。单独更新或保存策略不会更改 API 的运行时行为。有关部署 API 的更多信息，请参阅the section called “部署 REST API”。

可在 API Gateway 资源策略中使用的 AWS 条件键

下表包含可针对每种授权类型，对于 API Gateway 中的 API 的资源策略中使用的 AWS 条件键。

有关这些AWS条件键的更多信息，请参阅AWS全局条件上下文键。

条件键表

条件键	标准	是否需要 **AuthN**？	授权类型
`aws:CurrentTime`	无	否	All
`aws:EpochTime`	无	否	All
`aws:TokenIssueTime`	键仅在使用临时安全凭证签名的请求中存在。	是	IAM
`aws:MultiFactorAuthPresent`	键仅在使用临时安全凭证签名的请求中存在。	是	IAM

条件键	标准	是否需要 **AuthN**？	授权类型
`aws:MultiFactorAuthAge`	键仅在请求中存在 MFA 时存在。	是	IAM
`aws:PrincipalAccount`	无	是	IAM
`aws:PrincipalArn`	无	是	IAM
`aws:PrincipalOrgID`	仅在委托人是组织成员时，才将此键包含在请求上下文中。	是	IAM
`aws:PrincipalOrgPaths`	仅在委托人是组织成员时，才将此键包含在请求上下文中。	是	IAM
`aws:PrincipalTag`	仅在委托人使用具有附加标签的 IAM 用户时，才将此键包含在请求上下文中。对于委托人，如果使用附加了标签或会话标签的 IAM 角色，则应包括它。	是	IAM
`aws:PrincipalType`	无	是	IAM
`aws:Referer`	键仅在调用方在 HTTP 标头中提供了值时存在。	否	All
`aws:SecureTransport`	无	否	All
`aws:SourceArn`	无	否	All

条件键	标准	是否需要 **AuthN**？	授权类型
`aws:SourceIp`	无	否	All
`aws:SourceVpc`	此键只能用于私有 API。	否	All
`aws:SourceVpce`	此键只能用于私有 API。	否	All
`aws:VpcSourceIp`	此键只能用于私有 API。	否	All
`aws:UserAgent`	键仅在调用方在 HTTP 标头中提供了值时存在。	否	All
`aws:userid`	无	是	IAM
`aws:username`	无	是	IAM

使用 IAM 权限控制对 API 的访问

通过控制对以下两个 API Gateway 组件进程的访问，您可以使用 IAM 权限控制对 Amazon API Gateway API 的访问：

- 要在 API Gateway 中创建、部署和管理 API，您必须授予 API 开发人员相关权限，使其可执行受 API Gateway 的 API 管理组件支持的必要操作。
- 要调用已部署的 API 或者刷新 API 缓存，您必须授予 API 调用方相应权限，使其可执行受 API Gateway 的 API 执行组件支持的必要 IAM 操作。

对此两个进程的访问控制将采用两种不同的许可模型，我们将在下文进行说明。

用于创建和管理 API 的 API Gateway 权限模型

要使 API 开发人员可在 API Gateway 中创建和管理 API，您必须创建 IAM 权限策略，允许特定的 API 开发人员创建、更新、部署、查看或删除必要的 API 实体。将权限策略附加到用户、角色或组。

要提供访问权限，请为您的用户、组或角色添加权限：

- AWS IAM Identity Center 中的用户和群组：

  创建权限集合。按照《AWS IAM Identity Center 用户指南》中创建权限集的说明进行操作。
- 通过身份提供商在 IAM 中托管的用户：

  创建适用于身份联合验证的角色。按照《IAM 用户指南》中为第三方身份提供商创建角色（联合身份验证）的说明进行操作。
- IAM 用户：
  - 创建您的用户可以担任的角色。按照《IAM 用户指南》中为 IAM 用户创建角色的说明进行操作。
  - （不推荐使用）将策略直接附加到用户或将用户添加到用户组。按照《IAM 用户指南》中向用户添加权限（控制台）中的说明进行操作。

有关如何使用此许可模型的更多信息，请参阅 the section called “API Gateway 基于身份的策略”。

用于调用 API 的 API Gateway 权限模型

要使 API 调用方可调用 API 或刷新其缓存，您必须创建相关的 IAM policy，以允许指定的 API 调用方调用已启用用户身份验证的 API 方法。API 开发人员可将该方法的 `authorizationType` 属性设置为 `AWS_IAM`，以要求调用方提交用户的凭证以进行身份验证。然后，将该策略附加到您的用户、角色或组。

在此 IAM 权限策略语句中，IAM `Resource` 元素包含一系列已部署的 API 方法，并使用给定的 HTTP 动词和 API Gateway 资源路径进行识别。IAM `Action` 元素包含必要的 API Gateway API 执行操作。这些操作包括 `execute-api:Invoke` 或 `execute-api:InvalidateCache`，其中 `execute-api` 表示 API Gateway 的底层 API 执行组件。

有关如何使用此许可模型的更多信息，请参阅 针对调用 API 的访问控制。

当 API 在后端集成 AWS 服务（例如 AWS Lambda）时，API Gateway 还必须具有代表 API 调用方访问集成的 AWS 资源（例如，调用 Lambda 函数）的许可。要授予这些权限，请创建适用于 API Gateway 的AWS服务类型的 IAM 角色。当您在 IAM 管理控制台中创建该角色时，生成的角色包含以下 IAM 信任策略，该策略声明 API Gateway 为可信实体，允许其代入该角色：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
```

```
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

如果您通过调用 create-role CLI 命令或相应的开发工具包方法创建 IAM 角色，则必须提供以上信任策略作为 `assume-role-policy-document` 的输入参数。请勿试图直接在 IAM 管理控制台中创建这样的策略，或调用 AWS CLI create-policy 命令或相应的开发工具包方法创建这样的策略。

要让 API Gateway 调用集成 AWS 服务，您还必须为此角色附加适用于调用集成 AWS 服务的 IAM 权限策略。例如，要调用 Lambda 函数，您必须在 IAM 角色中包含以下 IAM 权限策略：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "*"
        }
    ]
}
```

请注意，Lambda 支持结合了信任策略和权限策略这两者的基于资源的标准访问策略。当使用 API Gateway 控制台集成 API 与 Lambda 函数时，系统不会明确要求您设置此 IAM 角色，因为控制台会经您同意，对 Lambda 函数设置基于资源的权限。

> **Note**
>
> 要制定对 AWS 服务的访问控制，您可以使用基于调用方的权限模型，将权限策略直接附加到调用方的用户或组；或者使用基于角色的权限模型，将权限策略附加到 API Gateway 可代入的 IAM 角色。在两种模型中的许可策略可能有所不同。例如，当基于调用方的策略可阻止访问时，基于角色的策略可能会允许访问。您可以利用这一点来要求用户仅通过 API Gateway API 访问 AWS 服务。

针对调用 API 的访问控制

在本节中，您将了解如何编写 IAM 策略语句，以控制谁可以调用 API Gateway 中部署的 API。在这里，您还将找到策略语句参考，包括与 API 执行服务有关的 `Action` 和 `Resource` 字段的格式。您还应该研究the section called “资源策略如何影响授权工作流”中的 IAM 部分。

对于私有 API，您应使用 API Gateway 资源策略和 VPC 终端节点策略的组合。有关更多信息，请参阅以下主题：

- the section called “使用 API Gateway 资源策略”
- the section called “为私有 API 使用 VPC 端点策略”

控制谁可以使用 IAM 策略调用 API Gateway API 方法

要控制谁可以或不可以使用 IAM 权限调用已部署的 API，请创建一个包含必要权限的 IAM 策略文档。此类策略文档的模板如下所示。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Permission",
      "Action": [
        "execute-api:Execution-operation"
      ],
      "Resource": [
        "arn:aws:execute-api:region:account-id:api-id/stage/METHOD_HTTP_VERB/Resource-
path"
      ]
    }
  ]
}
```

此处，*`Permission`* 将被替换为 `Allow` 或 `Deny`，具体取决于您是要授予或撤消相应权限。*`Execution-operation`* 将被替换为受 API 执行服务支持的操作。*`METHOD_HTTP_VERB`* 表示受指定资源支持的 HTTP 动词。*`Resource-path`* 是已部署的 API `Resource` 实例的 URL 路径占位符，该实例可支持上述的 *`METHOD_HTTP_VERB`*。有关更多信息，请参阅 在 API Gateway 中执行 API 的 IAM 策略的语句参考。

Note

要使 IAM 策略生效，您必须已通过将该方法的 `authorizationType` 属性设置为 `AWS_IAM` 对 API 方法启用 IAM 身份验证。否则，这些 API 方法会从外部进行访问。

例如，要授予某个用户查看特定 API 提供的宠物列表的权限，同时拒绝该用户向列表添加宠物的权限，您可以在 IAM 策略中包含以下语句：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:account-id:api-id/*/GET/pets"
      ]
    },
    {
      "Effect": "Deny",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:account-id:api-id/*/POST/pets"
      ]
    }
  ]
}
```

要授予某个用户查看配置为 `GET /pets/{petId}` 的 API 提供的某个特定宠物的权限，您可以在 IAM 策略中包含以下语句：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
```

```
          "execute-api:Invoke"
        ],
        "Resource": [
          "arn:aws:execute-api:us-east-1:account-id:api-id/*/GET/pets/a1b2"
        ]
      }
    ]
}
```

在 API Gateway 中执行 API 的 IAM 策略的语句参考

以下信息将介绍用于执行 API 的访问权限的 IAM 策略语句的 Action 和 Resource 格式。

在 API Gateway 中执行 API 的权限的 Action 格式

API 执行的 `Action` 表达式有以下一般格式：

```
execute-api:action
```

其中 *`action`* 是一项可用的 API 执行操作：

- *，代表以下所有操作。
- Invoke (调用)，用于根据客户端请求调用 API。
- InvalidateCache，用于根据客户端请求使 API 缓存失效。

在 API Gateway 中执行 API 的权限的 Resource 格式

API 执行的 `Resource` 表达式有以下一般格式：

```
arn:aws:execute-api:region:account-id:api-id/stage-name/HTTP-VERB/resource-path-specifier
```

其中：

- *`region`* 为 AWS 区域（例如，适用于所有 **`us-east-1`** 区域的 **`*`** 或 AWS），与该方法已部署的 API 相对应。
- *`account-id`* 是 REST API 所有者的 12 位 AWS 账户 ID。
- *`api-id`* 是 API Gateway 为该方法分配给 API 的标识符。
- *`stage-name`* 是与方法关联的阶段的名称。

- *HTTP-VERB* 是该方法的 HTTP 动词。它可以是以下任一动词：GET、POST、PUT、DELETE、PATCH。
- *resource-path-specifier* 是前往预期方法的路径。

> **Note**
>
> 如果您指定了通配符 (`*`)，则 `Resource` 表达式会将通配符应用于表达式的其余部分。

一些示例资源表达式包括：

- **`arn:aws:execute-api:*:*:*`**，适用于任何 AWS 区域内任何 API 的任何阶段中的任何资源路径。
- **`arn:aws:execute-api:us-east-1:*:*`**，适用于 `us-east-1` 的任何 AWS 区域内任何 API 的任何阶段中的任何资源路径。
- **`arn:aws:execute-api:us-east-1:*:`*`api-id`*`/*`**，适用于AWS区域 us-east-1 内带 *`api-id`* 标识符的 API 中任何阶段的任何资源路径。
- **`arn:aws:execute-api:us-east-1:*:`*`api-id`*`/test/*`**，适用于AWS区域 us-east-1 内带 *`api-id`* 标识符的 API 中 `test` 阶段的资源路径。

要了解更多信息，请参阅 API Gateway Amazon 资源名称 (ARN) 参考。

API 执行权限的 IAM 策略示例

有关许可模型和其他背景信息，请参阅 针对调用 API 的访问控制。

以下策略语句将为带 `a123456789` 标识符的 API 授予在 `test` 阶段沿 `mydemoresource` 路径调用任何 POST 方法的用户权限，并假定已将相应的 API 部署至 AWS 区域 us-east-1：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:*:a123456789/test/POST/mydemoresource/*"
```

```
      ]
    }
  ]
}
```

以下示例策略语句将在已部署相应 API 的任何 `petstorewalkthrough/pets` 区域为带 a123456789 标识符的 API 授予在任何阶段沿 AWS 资源路径调用任意方法的用户许可：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:*:*:a123456789/*/*/petstorewalkthrough/pets"
      ]
    }
  ]
}
```

创建策略并将其附加到用户

要使用户可调用 API 管理服务或 API 执行服务，您必须创建 IAM policy，该策略控制对 API Gateway 实体的访问。

使用 JSON 策略编辑器创建策略

1. 登录AWS Management Console，然后通过以下网址打开 IAM 控制台：https://console.aws.amazon.com/iam/。
2. 在左侧的导航窗格中，选择策略。

   如果这是您首次选择策略，则会显示欢迎访问托管式策略页面。选择开始使用。
3. 在页面的顶部，选择创建策略。
4. 在策略编辑器部分，选择 JSON 选项。
5. 输入以下 JSON 策略文档：

   ```
   {
     "Version": "2012-10-17",
   ```

```
  "Statement" : [
    {
      "Effect" : "Allow",
      "Action" : [
        "action-statement"
      ],
      "Resource" : [
        "resource-statement"
      ]
    },
    {
      "Effect" : "Allow",
      "Action" : [
        "action-statement"
      ],
      "Resource" : [
        "resource-statement"
      ]
    }
  ]
}
```

6. 选择下一步。

> **Note**
>
> 您可以随时在可视化和 JSON 编辑器选项卡之间切换。不过，如果您进行更改或在可视化编辑器中选择下一步，IAM 可能会调整策略结构以针对可视化编辑器进行优化。有关更多信息，请参阅《IAM 用户指南》中的调整策略结构。

7. 在查看并创建页面上，为您要创建的策略输入策略名称和描述（可选）。查看此策略中定义的权限以查看策略授予的权限。
8. 选择创建策略可保存新策略。

在此语句中，根据需要替换 *`action-statement`* 和 *`resource-statement`*，然后添加其他语句以指定您要让用户管理的 API Gateway 实体，以及让用户可以调用的 API 方法，或者两者。原定设置情况下，用户没有权限，除非有明确的对应 `Allow` 语句。

您刚刚创建了一个 IAM 策略。在您附加它之前，它不会有任何效果。

要提供访问权限，请为您的用户、组或角色添加权限：

- AWS IAM Identity Center 中的用户和群组：

  创建权限集合。按照《AWS IAM Identity Center 用户指南》中创建权限集的说明进行操作。
- 通过身份提供商在 IAM 中托管的用户：

  创建适用于身份联合验证的角色。按照《IAM 用户指南》中为第三方身份提供商创建角色（联合身份验证）的说明进行操作。
- IAM 用户：
  - 创建您的用户可以担任的角色。按照《IAM 用户指南》中为 IAM 用户创建角色的说明进行操作。
  - （不推荐使用）将策略直接附加到用户或将用户添加到用户组。按照《IAM 用户指南》中向用户添加权限（控制台）中的说明进行操作。

将 IAM 策略文档附加到 IAM 组

1. 从主导航窗格中选择组。
2. 在所选组下选择权限选项卡。
3. 选择 Attach policy。
4. 选择您之前创建的策略文档，然后选择 Attach policy (挂载策略)。

要让 API Gateway 代表您调用其他AWS服务，请创建 Amazon API Gateway 类型的 IAM 角色。

创建 Amazon API Gateway 类型的角色

1. 从主导航窗格中选择角色。
2. 选择 Create New Role。
3. 键入 Role name (角色名称) 的名称，然后选择 Next Step (下一步)。
4. 在 Select Role Type (选择角色类型) 下的 AWS Service Roles (服务角色) 中，选择 Amazon API Gateway 旁边的 Select (选择)。
5. 在附加策略下，选择一个可用的托管 IAM 权限策略（例如，如果您想让 API Gateway 在 CloudWatch 中记录指标，则选择 AmazonAPIGatewayPushToCloudWatchLog），然后选择下一步。
6. 在 Trusted Entities (可信任的实体) 下，验证 apigateway.amazonaws.com 已作为条目列出，然后选择 Create Role (创建角色)。
7. 在新创建的角色中，选择权限选项卡，然后选择 Attach Policy (挂载策略)。
8. 选择先前创建的自定义 IAM 策略文档，然后选择附加策略。

在 API Gateway 中为私有 API 使用 VPC 端点策略

要提高私有 API 的安全性，您可以创建 VPC 端点策略。VPC 端点策略是一种 IAM 资源策略，您可以将其附加到接口端点。有关更多信息，请参阅使用 VPC 端点控制对服务的访问。

您可能想要创建 VPC 端点策略来执行以下操作：

- 仅允许某些组织或资源访问您的 VPC 端点和调用您的 API。
- 使用单一策略，并避免使用基于会话或基于角色的策略，来控制 API 的流量。
- 从本地迁移到 AWS 时，加强应用程序的安全边界。

VPC 端点策略注意事项

- 根据 `Authorization` 标头值评估调用方的身份。这可能会导致 `403 IncompleteSignatureException` 或 `403 InvalidSignatureException` 错误，具体视您的 `authorizationType` 而定。下表显示每个 `authorizationType` 的 `Authorization` 标头值。

`authorizationType`	已评估 **`Authorization`** 标头？	允许的 **`Authorization`** 标头值
`NONE`，使用默认完全访问策略	否	未传递
`NONE`，使用自定义访问策略	是	必须是有效的 SigV4 值
`IAM`	是	必须是有效的 SigV4 值
`CUSTOM` 或者 `COGNITO_USER_POOLS`	否	未传递

- 如果策略限制了对特定 IAM 主体（例如 `arn:aws:iam::account-id:role/developer`）的访问权限，则必须将 API 方法的 `authorizationType` 设置为 `AWS_IAM` 或 `NONE`。有关如何为方法设置 `authorizationType` 的更多说明，请参阅 the section called “方法”。
- VPC 端点策略可以与 API Gateway 资源策略一起使用。API Gateway 资源策略指定哪些主体可以访问 API。端点策略指定了谁可以访问 VPC，以及可以从 VPC 端点调用哪些 API。您的私有 API 需要资源策略，但您不需要创建自定义 VPC 端点策略。

VPC 端点策略示例

您可以为 Amazon API Gateway 创建 Amazon Virtual Private Cloud 端点策略，您可以在其中指定：

- 可执行操作的委托人。
- 可执行的操作。
- 可用于执行操作的资源。

要将策略附加到 VPC 端点，您需要使用 VPC 控制台。有关更多信息，请参阅使用 VPC 端点控制对服务的访问。

示例 1：授予对两个 API 的访问权限的 VPC 端点策略

以下示例策略只授予通过该策略附加到的 VPC 端点访问两个特定 API。

```
{
    "Statement": [
        {
            "Principal": "*",
            "Action": [
                "execute-api:Invoke"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:execute-api:us-east-1:123412341234:a1b2c3d4e5/*",
                "arn:aws:execute-api:us-east-1:123412341234:aaaaa11111/*"
            ]
        }
    ]
}
```

示例 2：授予对 GET 方法的访问权限的 VPC 端点策略

以下示例策略授予用户通过该策略附加到的 VPC 端点访问特定 API 的 GET 方法。

```
{
    "Statement": [
        {
            "Principal": "*",
            "Action": [
```

```
                "execute-api:Invoke"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:execute-api:us-east-1:123412341234:a1b2c3d4e5/stageName/GET/*"
            ]
        }
    ]
}
```

示例 3：授予用户对特定 API 的特定用户访问权限的 VPC 端点策略

以下示例策略授予特定用户通过该策略附加到的 VPC 端点访问特定 API。

在这种情况下，因为策略限制了对特定 IAM 主体的访问权限，所以您必须将方法的 `authorizationType` 设置为 `AWS_IAM` 或 `NONE`。

```
{
    "Statement": [
        {
            "Principal": {
                "AWS": [
                    "arn:aws:iam::123412341234:user/MyUser"
                ]
            },
            "Action": [
                "execute-api:Invoke"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:execute-api:us-east-1:123412341234:a1b2c3d4e5/*"
            ]
        }
    ]
}
```

使用标签控制对 API Gateway 中的 REST API 的访问

使用 IAM 策略中基于属性的访问控制，可以微调访问 REST API 的权限。

有关更多信息，请参阅 the section called “基于属性的访问控制”。

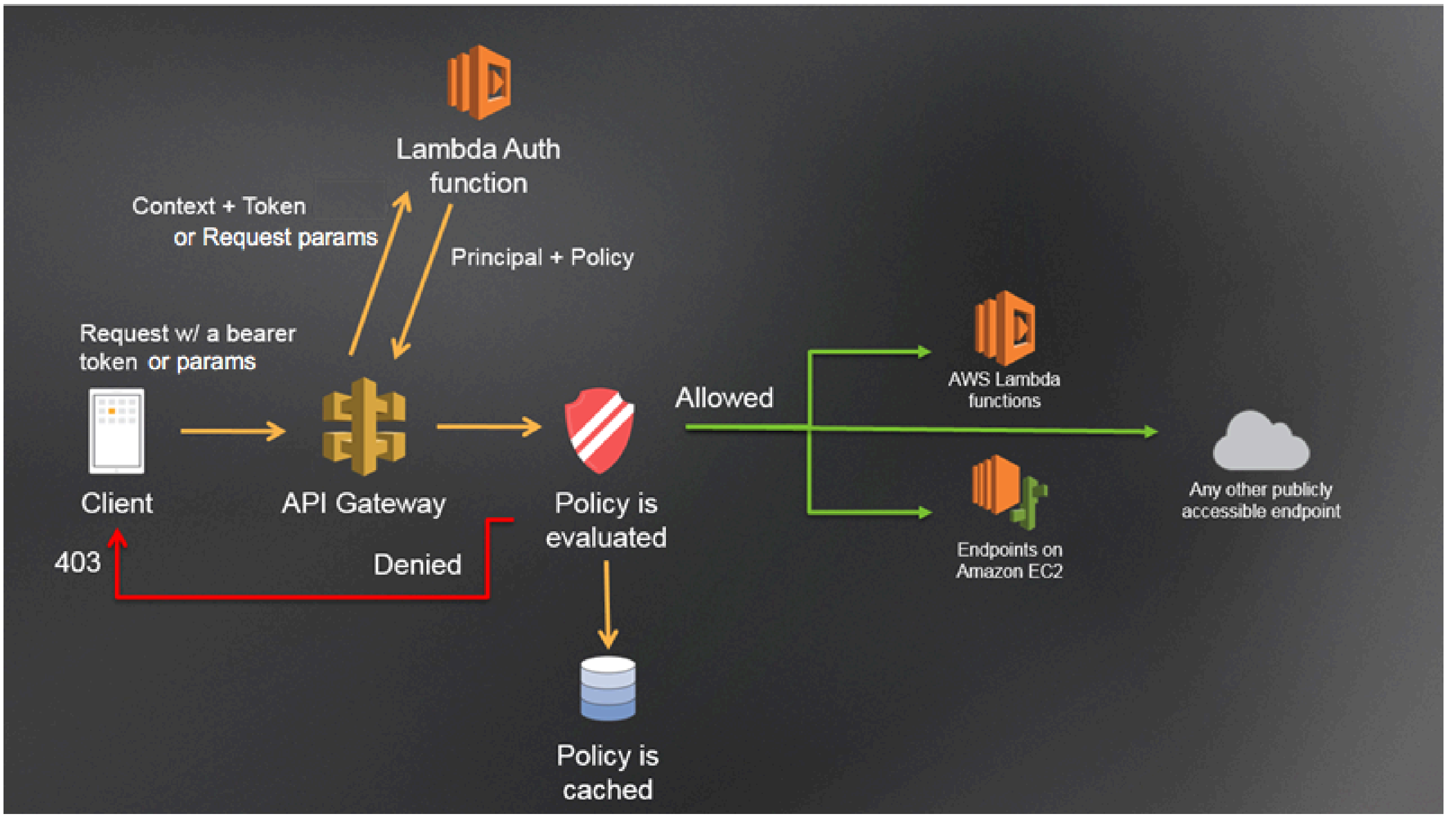
使用 API Gateway Lambda 授权方

使用 Lambda 授权方（以前称为自定义授权方）控制对 API 的访问。当客户端请求 API 方法时，API Gateway 会调用 Lambda 授权方。Lambda 授权方将调用方的身份作为输入，并返回 IAM 策略作为输出。

使用 Lambda 授权方实施自定义授权方案。该方案可以使用请求参数来确定调用方的身份或使用持有者令牌身份验证策略（例如 OAuth 或 SAML）。在 API Gateway REST API 控制台中、使用 AWS CLI 或 AWS SDK 创建 Lambda 授权方。

Lambda 授权方授权工作流

下图展示了 Lambda 授权方的授权工作流。



API Gateway Lambda 授权工作流

1. 客户端对 API Gateway API 调用方法，以传递持有者令牌或请求参数。
2. API Gateway 检查是否为方法请求配置了 Lambda 授权方。如果已配置，API Gateway 将调用 Lambda 函数。
3. Lambda 函数会对调用方进行身份验证。该函数可通过以下方式进行身份验证：

- 调用 OAuth 提供程序来获取 OAuth 访问令牌。
- 调用 SAML 提供程序来获取 SAML 断言。
- 基于请求参数值生成 IAM 策略。
- 从数据库中检索凭证。

4. Lambda 函数返回一个 IAM 策略和一个主体标识符。如果 Lambda 函数不返回该信息，表明调用失败。
5. API Gateway 会对 IAM 策略进行评估。
   - 如果拒绝访问，API Gateway 将返回一个合适的 HTTP 状态代码，如 `403 ACCESS_DENIED`。
   - 如果允许访问，API Gateway 将调用该方法。

     如果启用了授权缓存，API Gateway 将缓存策略，这样就不会再次调用 Lambda 授权方函数。

您可以自定义 `403 ACCESS_DENIED` 或 `401 UNAUTHORIZED` 网关响应。要了解更多信息，请参阅 the section called “网关响应”。

选择 Lambda 授权方类型

Lambda 授权方有两种类型：

基于请求参数的 Lambda 授权方（**`REQUEST`** 授权方）

> `REQUEST` 授权方接收标头、查询字符串参数、`stageVariables` 和 `$context` 变量组合中的调用方身份。您可以使用 `REQUEST` 授权方，根据来自多个身份来源（例如 `$context.path` 和 `$context.httpMethod` 上下文变量）的信息创建精细策略。
>
> 如果您为 `REQUEST` 授权方开启授权缓存，API Gateway 会验证请求中是否存在所有指定的身份来源。如果指定的身份来源缺失、为 null 或为空，API Gateway 将返回 `401 Unauthorized` HTTP 响应，而不调用 Lambda 授权方函数。如果定义了多个身份来源，它们都将用于派生授权方的缓存键并保持相应顺序。您可以使用多个身份来源定义精细缓存键。
>
> 如果您更改缓存键的任意部分并重新部署 API，授权方将丢弃缓存策略文档并生成新的文档。
>
> 如果您为 `REQUEST` 授权方关闭授权缓存，API Gateway 会直接将请求传递给 Lambda 函数。

基于令牌的 Lambda 授权方（**`TOKEN`** 授权方）

> `TOKEN` 授权方接收持有者令牌（例如 JSON Web 令牌（JWT）或 OAuth 令牌）中的调用方身份。
>
> 如果您为 `TOKEN` 授权方开启授权缓存，令牌来源中指定的标头名称将成为缓存键。

此外，您还可以使用令牌验证来输入正则表达式语句。API Gateway 将针对此表达式执行输入令牌的初始验证并在成功验证后调用 Lambda 授权方函数。这有助于减少对您的 API 的调用。

仅 `TOKEN` 授权方支持 `IdentityValidationExpression` 属性。有关更多信息，请参阅 the section called “`x-amazon-apigateway-authorizer`”。

> **Note**
>
> 建议您使用 `REQUEST` 授权方来控制对 API 的访问。使用 `REQUEST` 授权方时，您可以基于多个身份来源控制对 API 的访问，而使用 `TOKEN` 授权方时则基于单一身份来源。此外，您还可以使用 `REQUEST` 授权方的多个身份来源分离缓存键。

`REQUEST` 授权方 Lambda 函数示例

以下代码示例创建一个 Lambda 授权方函数，如果客户端提供的 `HeaderAuth1` 标头、`QueryString1` 查询参数和 `StageVar1` 阶段变量都分别与 `headerValue1`、`queryValue1` 和 `stageValue1` 的指定值相匹配，该函数将允许调用请求。

Node.js

```
    // A simple request-based authorizer example to demonstrate how to use request
     // parameters to allow or deny a request. In this example, a request is
     // authorized if the client-supplied HeaderAuth1 header, QueryString1
     // query parameter, and stage variable of StageVar1 all match
     // specified values of 'headerValue1', 'queryValue1', and 'stageValue1',
     // respectively.

export const handler = function(event, context, callback) {
    console.log('Received event:', JSON.stringify(event, null, 2));

    // Retrieve request parameters from the Lambda function input:
    var headers = event.headers;
    var queryStringParameters = event.queryStringParameters;
    var pathParameters = event.pathParameters;
    var stageVariables = event.stageVariables;

    // Parse the input for the parameter values
    var tmp = event.methodArn.split(':');
    var apiGatewayArnTmp = tmp[5].split('/');
    var awsAccountId = tmp[4];
```

```
    var region = tmp[3];
    var restApiId = apiGatewayArnTmp[0];
    var stage = apiGatewayArnTmp[1];
    var method = apiGatewayArnTmp[2];
    var resource = '/'; // root resource
    if (apiGatewayArnTmp[3]) {
        resource += apiGatewayArnTmp[3];
    }

    // Perform authorization to return the Allow policy for correct parameters and
    // the 'Unauthorized' error, otherwise.
    var authResponse = {};
    var condition = {};
    condition.IpAddress = {};

    if (headers.HeaderAuth1 === "headerValue1"
        && queryStringParameters.QueryString1 === "queryValue1"
        && stageVariables.StageVar1 === "stageValue1") {
        callback(null, generateAllow('me', event.methodArn));
    }  else {
        callback("Unauthorized");
    }
}

// Help function to generate an IAM policy
var generatePolicy = function(principalId, effect, resource) {
    // Required output:
    var authResponse = {};
    authResponse.principalId = principalId;
    if (effect && resource) {
        var policyDocument = {};
        policyDocument.Version = '2012-10-17'; // default version
        policyDocument.Statement = [];
        var statementOne = {};
        statementOne.Action = 'execute-api:Invoke'; // default action
        statementOne.Effect = effect;
        statementOne.Resource = resource;
        policyDocument.Statement[0] = statementOne;
        authResponse.policyDocument = policyDocument;
    }
    // Optional output with custom properties of the String, Number or Boolean type.
    authResponse.context = {
        "stringKey": "stringval",
        "numberKey": 123,
```

```
        "booleanKey": true
    };
    return authResponse;
}

var generateAllow = function(principalId, resource) {
    return generatePolicy(principalId, 'Allow', resource);
}

var generateDeny = function(principalId, resource) {
    return generatePolicy(principalId, 'Deny', resource);
}
```

Python

```
# A simple request-based authorizer example to demonstrate how to use request
# parameters to allow or deny a request. In this example, a request is
# authorized if the client-supplied HeaderAuth1 header, QueryString1
# query parameter, and stage variable of StageVar1 all match
# specified values of 'headerValue1', 'queryValue1', and 'stageValue1',
# respectively.

import json


def lambda_handler(event, context):
    print(event)

    # Retrieve request parameters from the Lambda function input:
    headers = event['headers']
    queryStringParameters = event['queryStringParameters']
    pathParameters = event['pathParameters']
    stageVariables = event['stageVariables']

    # Parse the input for the parameter values
    tmp = event['methodArn'].split(':')
    apiGatewayArnTmp = tmp[5].split('/')
    awsAccountId = tmp[4]
    region = tmp[3]
    restApiId = apiGatewayArnTmp[0]
    stage = apiGatewayArnTmp[1]
    method = apiGatewayArnTmp[2]
    resource = '/'
```

```
    if (apiGatewayArnTmp[3]):
        resource += apiGatewayArnTmp[3]

    # Perform authorization to return the Allow policy for correct parameters
    # and the 'Unauthorized' error, otherwise.

    authResponse = {}
    condition = {}
    condition['IpAddress'] = {}

    if (headers['HeaderAuth1'] == "headerValue1" and
 queryStringParameters['QueryString1'] == "queryValue1" and
 stageVariables['StageVar1'] == "stageValue1"):
        response = generateAllow('me', event['methodArn'])
        print('authorized')
        return json.loads(response)
    else:
        print('unauthorized')
        raise Exception('Unauthorized') # Return a 401 Unauthorized response
        return 'unauthorized'

    # Help function to generate IAM policy


def generatePolicy(principalId, effect, resource):
    authResponse = {}
    authResponse['principalId'] = principalId
    if (effect and resource):
        policyDocument = {}
        policyDocument['Version'] = '2012-10-17'
        policyDocument['Statement'] = []
        statementOne = {}
        statementOne['Action'] = 'execute-api:Invoke'
        statementOne['Effect'] = effect
        statementOne['Resource'] = resource
        policyDocument['Statement'] = [statementOne]
        authResponse['policyDocument'] = policyDocument

    authResponse['context'] = {
        "stringKey": "stringval",
        "numberKey": 123,
        "booleanKey": True
    }
```

```
    authResponse_JSON = json.dumps(authResponse)

    return authResponse_JSON


def generateAllow(principalId, resource):
    return generatePolicy(principalId, 'Allow', resource)


def generateDeny(principalId, resource):
    return generatePolicy(principalId, 'Deny', resource)
```

在此示例中，Lambda 授权方函数将检查输入参数并按如下所示操作：

- 如果所有必需的参数值匹配预期值，该授权方函数将返回 `200 OK` HTTP 响应和 IAM 策略（类似于以下内容）并且方法请求成功：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "execute-api:Invoke",
      "Effect": "Allow",
      "Resource": "arn:aws:execute-api:us-east-1:123456789012:ivdtdhp7b5/
ESTestInvoke-stage/GET/"
    }
  ]
}
```

- 否则，授权方函数将返回 `401 Unauthorized` HTTP 响应并且方法请求将失败。

除了返回 IAM 策略之外，Lambda 授权方函数还必须返回调用方的委托人标识符。它还可以选择返回一个 `context` 对象，其中包含可传入集成后端的其他信息。有关更多信息，请参阅 来自 API Gateway Lambda 授权方的输出。

在生产代码中，您可能需要先对用户进行身份验证，然后才能授予授权。您可以通过调用身份验证提供程序，在 Lambda 函数中添加身份验证逻辑，如该提供程序的文档中的指示。

`TOKEN` 授权方 Lambda 函数示例

以下代码示例创建一个 TOKEN Lambda 授权方函数，如果客户端提供的令牌值为 `allow`，该函数将允许调用方调用方法。如果令牌值为 `deny`，则不允许调用方调用请求。如果令牌值为 `unauthorized` 或空字符串，该授权方函数将返回 `401 UNAUTHORIZED` 响应。

Node.js

```
// A simple token-based authorizer example to demonstrate how to use an
 authorization token
// to allow or deny a request. In this example, the caller named 'user' is allowed
 to invoke
// a request if the client-supplied token value is 'allow'. The caller is not
 allowed to invoke
// the request if the token value is 'deny'. If the token value is 'unauthorized' or
 an empty
// string, the authorizer function returns an HTTP 401 status code. For any other
 token value,
// the authorizer returns an HTTP 500 status code.
// Note that token values are case-sensitive.

export const handler =  function(event, context, callback) {
    var token = event.authorizationToken;
    switch (token) {
        case 'allow':
            callback(null, generatePolicy('user', 'Allow', event.methodArn));
            break;
        case 'deny':
            callback(null, generatePolicy('user', 'Deny', event.methodArn));
            break;
        case 'unauthorized':
            callback("Unauthorized");   // Return a 401 Unauthorized response
            break;
        default:
            callback("Error: Invalid token"); // Return a 500 Invalid token response
    }
};

// Help function to generate an IAM policy
var generatePolicy = function(principalId, effect, resource) {
    var authResponse = {};

    authResponse.principalId = principalId;
```

```
    if (effect && resource) {
        var policyDocument = {};
        policyDocument.Version = '2012-10-17';
        policyDocument.Statement = [];
        var statementOne = {};
        statementOne.Action = 'execute-api:Invoke';
        statementOne.Effect = effect;
        statementOne.Resource = resource;
        policyDocument.Statement[0] = statementOne;
        authResponse.policyDocument = policyDocument;
    }

    // Optional output with custom properties of the String, Number or Boolean type.
    authResponse.context = {
        "stringKey": "stringval",
        "numberKey": 123,
        "booleanKey": true
    };
    return authResponse;
}
```

Python

```
# A simple token-based authorizer example to demonstrate how to use an authorization
 token
# to allow or deny a request. In this example, the caller named 'user' is allowed to
 invoke
# a request if the client-supplied token value is 'allow'. The caller is not allowed
 to invoke
# the request if the token value is 'deny'. If the token value is 'unauthorized' or
 an empty
# string, the authorizer function returns an HTTP 401 status code. For any other
 token value,
# the authorizer returns an HTTP 500 status code.
# Note that token values are case-sensitive.

import json


def lambda_handler(event, context):
    token = event['authorizationToken']
    if token == 'allow':
        print('authorized')
```

```
        response = generatePolicy('user', 'Allow', event['methodArn'])
    elif token == 'deny':
        print('unauthorized')
        response = generatePolicy('user', 'Deny', event['methodArn'])
    elif token == 'unauthorized':
        print('unauthorized')
        raise Exception('Unauthorized')  # Return a 401 Unauthorized response
        return 'unauthorized'
    try:
        return json.loads(response)
    except BaseException:
        print('unauthorized')
        return 'unauthorized'  # Return a 500 error


def generatePolicy(principalId, effect, resource):
    authResponse = {}
    authResponse['principalId'] = principalId
    if (effect and resource):
        policyDocument = {}
        policyDocument['Version'] = '2012-10-17'
        policyDocument['Statement'] = []
        statementOne = {}
        statementOne['Action'] = 'execute-api:Invoke'
        statementOne['Effect'] = effect
        statementOne['Resource'] = resource
        policyDocument['Statement'] = [statementOne]
        authResponse['policyDocument'] = policyDocument
    authResponse['context'] = {
        "stringKey": "stringval",
        "numberKey": 123,
        "booleanKey": True
    }
    authResponse_JSON = json.dumps(authResponse)
    return authResponse_JSON
```

在此示例中，当 API 接收方法请求时，API Gateway 将源令牌传递到 `event.authorizationToken` 属性中的此 Lambda 授权方函数。Lambda 授权方函数将读取令牌并按下所示做出行为：

- 如果令牌值为 `allow`，该授权方函数将返回 `200 OK` HTTP 响应和 IAM 策略（类似于以下内容）并且方法请求成功：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "execute-api:Invoke",
      "Effect": "Allow",
      "Resource": "arn:aws:execute-api:us-east-1:123456789012:ivdtdhp7b5/
ESTestInvoke-stage/GET/"
    }
  ]
}
```

- 如果令牌值为 deny，该授权方函数将返回 `200 OK` HTTP 响应和 Deny IAM 策略（类似于以下内容）并且方法请求失败：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "execute-api:Invoke",
      "Effect": "Deny",
      "Resource": "arn:aws:execute-api:us-east-1:123456789012:ivdtdhp7b5/
ESTestInvoke-stage/GET/"
    }
  ]
}
```

> **Note**
>
> 在测试环境之外，API Gateway 返回 `403 Forbidden` HTTP 响应并且方法请求将失败。

- 如果令牌值为 `unauthorized` 或空字符串，该授权方函数将返回 `401 Unauthorized` HTTP 响应并且方法调用失败。
- 如果令牌为任何其他内容，客户端将收到 `500 Invalid token` 响应并且方法调用失败。

除了返回 IAM 策略之外，Lambda 授权方函数还必须返回调用方的委托人标识符。它还可以选择返回一个 `context` 对象，其中包含可传入集成后端的其他信息。有关更多信息，请参阅 来自 API Gateway Lambda 授权方的输出。

在生产代码中，您可能需要先对用户进行身份验证，然后才能授予授权。您可以通过调用身份验证提供程序，在 Lambda 函数中添加身份验证逻辑，如该提供程序的文档中的指示。

其他 Lambda 授权方函数示例

以下列表显示了 Lambda 授权方函数的其他示例。您可以在创建 API 的相同账户或不同账户中创建 Lambda 函数。

对于前面的 Lambda 函数示例，您可以使用内置 AWSLambdaBasicExecutionRole，因为这些函数不会调用其他 AWS 服务。如果您的 Lambda 函数调用其他 AWS 服务，您需要为该 Lambda 函数分配 IAM 执行角色。要创建该角色，请按照 AWS Lambda 执行角色中的说明操作。

其他 Lambda 授权方函数示例

- 有关应用程序示例，请参阅 GitHub 中的 Open Banking Brazil - Authorization Samples。
- 有关更多示例 Lambda 函数，请参阅 GitHub 上的 aws-apigateway-lambda-authorizer-blueprints。
- 您可以创建 Lambda 授权方，该授权方使用 Amazon Cognito 用户池对用户进行身份验证，并使用 Verified Permissions 根据策略存储对调用方进行授权。有关更多信息，请参阅《Amazon Verified Permissions 用户指南》中的 Create a policy store with a connected API and identity provider。
- Lambda 控制台提供一个 Python 蓝图，您可以通过选择使用蓝图并选择 api-gateway-authorizer-python 蓝图来使用该蓝图。

配置 Lambda 授权方

创建 Lambda 函数后，您可以将 Lambda 函数配置为您的 API 的授权方。然后，您可以将方法配置为调用 Lambda 授权方，以确定调用方是否可以调用该方法。您可以在创建 API 的相同账户或不同账户中创建 Lambda 函数。

您可以使用 API Gateway 控制台中的内置工具或使用 Postman 来测试 Lambda 授权方。有关如何使用 Postman 测试 Lambda 授权方函数的说明，请参阅the section called “使用 Lambda 授权方调用 API”。

配置 Lambda 授权方（控制台）

以下过程介绍如何在 API Gateway REST API 控制台中创建 Lambda 授权方。要详细了解不同类型的 Lambda 授权方，请参阅the section called “选择 Lambda 授权方类型”。

REQUEST authorizer

配置 **`REQUEST`** Lambda 授权方

1. 通过以下网址登录到 API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 API，然后选择授权方。
3. 选择创建授权方。
4. 对于授权方名称，输入授权方名称。
5. 对于授权方类型，选择 Lambda。
6. 对于 Lambda 函数，选择您在其中创建 Lambda 授权方函数的 AWS 区域，然后输入函数名称。
7. 将 Lambda 调用角色留空，以允许 API Gateway REST API 控制台设置基于资源的策略。此策略授予 API Gateway 调用 Lambda 授权方函数的权限。您还可以选择输入 IAM 角色的名称，以允许 API Gateway 调用 Lambda 授权方函数。有关角色示例，请参阅创建一个可代入的 IAM 角色。
8. 对于 Lambda 事件负载，选择请求。
9. 在身份来源类型中，选择一个参数类型。支持的参数类型为 `Header`、`Query string`、`Stage variable` 和 `Context`。要添加更多身份来源，请选择添加参数。
10. 要缓存授权方生成的授权策略，请将授权缓存保持开启状态。在启用策略缓存后，您可以修改 TTL 值。将 TTL 设置为零将禁用策略缓存。

    如果启用缓存，授权方必须返回适用于 API 中所有方法的策略。要强制执行特定于方法的策略，请使用上下文变量 `$context.path` 和 `$context.httpMethod`。
11. 选择创建授权方。

TOKEN authorizer

配置 **`TOKEN`** Lambda 授权方

1. 通过以下网址登录到 API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 API，然后选择授权方。
3. 选择创建授权方。
4. 对于授权方名称，输入授权方名称。
5. 对于授权方类型，选择 Lambda。

6. 对于 Lambda 函数，选择您在其中创建 Lambda 授权方函数的 AWS 区域，然后输入函数名称。
7. 将 Lambda 调用角色留空，以允许 API Gateway REST API 控制台设置基于资源的策略。此策略授予 API Gateway 调用 Lambda 授权方函数的权限。您还可以选择输入 IAM 角色的名称，以允许 API Gateway 调用 Lambda 授权方函数。有关角色示例，请参阅创建一个可代入的 IAM 角色。
8. 对于 Lambda 事件负载，选择令牌。
9. 对于令牌来源，输入包含授权令牌的标头名称。客户端必须包括此标头名称才能将授权令牌发送到 Lambda 授权方。
10. （可选）对于令牌验证，请输入正则表达式语句。API Gateway 将针对此表达式执行对输入令牌的初始验证并在成功验证后调用授权方。
11. 要缓存授权方生成的授权策略，请将授权缓存保持开启状态。在启用策略缓存后，令牌来源中指定的标头名称将成为缓存键。在启用策略缓存后，您可以修改 TTL 值。将 TTL 设置为零将禁用策略缓存。

    如果启用缓存，授权方必须返回适用于 API 中所有方法的策略。要强制执行特定于方法的策略，您可以关闭授权缓存。
12. 选择创建授权方。

创建 Lambda 授权方后，您可以对其进行测试。以下过程介绍如何测试 Lambda 授权方。

REQUEST authorizer

测试 **REQUEST** Lambda 授权方

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择授权方的名称。
3. 在测试授权方下，输入身份来源的值。

   如果您使用的是 the section called “REQUEST 授权方 Lambda 函数示例”，请执行以下操作：

   a. 选择标头并输入 **headerValue1**，然后选择添加参数。
   b. 在身份来源类型下，选择查询字符串并输入 **queryValue1**，然后选择添加参数。
   c. 在身份来源类型下，选择阶段变量并输入 **stageValue1**。

您无法修改测试调用的上下文变量，但可以修改 Lambda 函数的 API Gateway 授权方测试事件模板。然后，您可以使用修改后的上下文变量来测试 Lambda 授权方函数。有关更多信息，请参阅《AWS Lambda 开发人员指南》中的在控制台中测试 Lambda 函数。

4. 选择测试授权方。

TOKEN authorizer

测试 **`TOKEN`** Lambda 授权方

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择授权方的名称。
3. 在测试授权方下，输入令牌的值。

   如果您使用的是 the section called “`TOKEN` 授权方 Lambda 函数示例”，请执行以下操作：

   - 对于 authorizationToken，请输入 **`allow`**。
4. 选择测试授权方。

   如果 Lambda 授权方成功拒绝了测试环境中的请求，测试将返回 `200 OK` HTTP 响应。但在测试环境之外，API Gateway 将返回 `403 Forbidden` HTTP 响应并且方法请求将失败。

配置 Lambda 授权方（AWS CLI）

以下 create-authorizer 命令演示了如何使用 AWS CLI 创建 Lambda 授权方。

REQUEST authorizer

以下示例创建 `REQUEST` 授权方并使用 `Authorizer` 标头和 `accountId` 上下文变量作为身份来源：

```
aws apigateway create-authorizer \
    --rest-api-id 1234123412 \
    --name 'First_Request_Custom_Authorizer' \
    --type REQUEST \
    --authorizer-uri 'arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123412341234:function:customAuthFunction/invocations' \
```

```
    --identity-source 'method.request.header.Authorization,context.accountId' \
    --authorizer-result-ttl-in-seconds 300
```

TOKEN authorizer

以下示例创建 `TOKEN` 授权方并使用 `Authorization` 标头作为身份来源：

```
aws apigateway create-authorizer \
    --rest-api-id 1234123412 \
    --name 'First_Token_Custom_Authorizer' \
    --type TOKEN \
    --authorizer-uri 'arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123412341234:function:customAuthFunction/invocations' \
    --identity-source 'method.request.header.Authorization' \
    --authorizer-result-ttl-in-seconds 300
```

创建 Lambda 授权方后，您可以对其进行测试。以下 test-invoke-authorizer 命令演示了如何测试 Lambda 授权方：

```
aws apigateway test-invoke-authorizer --rest-api-id 1234123412 \
   --authorizer-id efg1234 \
   --headers Authorization='Value'
```

配置方法以使用 Lambda 授权方（控制台）

配置 Lambda 授权方后，必须将其附加到 API 方法。

配置 API 方法以使用 Lambda 授权方

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 选择资源，然后选择一种新方法或选择现有方法。
4. 在方法请求选项卡上的方法请求设置下，选择编辑。
5. 对于授权方，从下拉菜单中选择您刚刚创建的 Lambda 授权方。
6. （可选）如果您希望将授权令牌传递到后端，请选择 HTTP 请求标头。选择添加标头，然后添加授权标头的名称。在名称中，输入标头名称，该名称必须与您在为 API 创建 Lambda 授权方时指定的令牌来源名称相匹配。此步骤不适用于 `REQUEST` 授权方。

7. 选择保存。
8. 选择部署 API，将 API 部署到某个阶段。对于使用阶段变量的 `REQUEST` 授权方，您还必须在阶段页面中定义必需的阶段变量并指定其值。

配置 API 方法以使用 Lambda 授权方（AWS CLI）

配置 Lambda 授权方后，必须将其附加到 API 方法。您可以创建新方法或使用补丁操作将授权方附加到现有方法。

以下 put-method 命令演示了如何创建使用 Lambda 授权方的新方法：

```
aws apigateway put-method --rest-api-id 1234123412 \
  --resource-id a1b2c3 \
  --http-method PUT \
  --authorization-type CUSTOM \
  --authorizer-id efg1234
```

以下 update-method 命令演示了如何更新现有方法以使用 Lambda 授权方：

```
aws apigateway update-method \
    --rest-api-id 1234123412 \
    --resource-id a1b2c3 \
    --http-method PUT \
    --patch-operations op="replace",path="/authorizationType",value="CUSTOM"
 op="replace",path="/authorizerId",value="efg1234"
```

向 Amazon API Gateway Lambda 授权方的输入

`TOKEN` 输入格式

对于 `TOKEN` 类型的 Lambda 授权方（以前称为自定义授权方），您必须在为 API 配置授权方时指定自定义标头作为令牌源。API 客户端必须在传入请求中传递该标头中必需的授权令牌。在收到传入方法请求后，API Gateway 将从自定义标头中提取此令牌。随后，它将此令牌作为 Lambda 函数的 `event` 对象的 `authorizationToken` 属性传递，并将方法 ARN 作为 `methodArn` 属性传递：

```
{
    "type":"TOKEN",
    "authorizationToken":"{caller-supplied-token}",
    "methodArn":"arn:aws:execute-api:{regionId}:{accountId}:{apiId}/{stage}/{httpVerb}/
[{resource}/[{child-resources}]]"
```

```
}
```

在此示例中，`type` 属性指定授权方类型，后者是 `TOKEN` 授权方。*`{caller-supplied-token}`* 源自客户端请求中的授权标头，可以是任何字符串值。`methodArn` 是传入方法请求的 ARN，由 API Gateway 根据 Lambda 授权方配置填充。

`REQUEST` 输入格式

对于 REQUEST 类型的 Lambda 授权方，API Gateway 会将请求参数作为 `event` 对象的一部分传递到授权方 Lambda 函数。请求参数包括标头、路径参数、查询字符串参数、阶段变量以及一些请求上下文变量。API 调用方可以设置路径参数、标头和查询字符串参数。API 开发人员必须在 API 部署期间设置阶段变量，并且 API Gateway 将在运行时提供请求上下文。

> Note
>
> 路径参数可作为请求参数传递到 Lambda 授权方函数，但它们不能作为身份来源。

以下示例显示针对带代理集成的 API 方法 (REQUEST) 的 `GET /request` 授权方的输入：

```
{
  "type": "REQUEST",
  "methodArn": "arn:aws:execute-api:us-east-1:123456789012:abcdef123/test/GET/request",
  "resource": "/request",
  "path": "/request",
  "httpMethod": "GET",
  "headers": {
    "X-AMZ-Date": "20170718T062915Z",
    "Accept": "*/*",
    "HeaderAuth1": "headerValue1",
    "CloudFront-Viewer-Country": "US",
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Is-Mobile-Viewer": "false",
    "User-Agent": "..."
  },
  "queryStringParameters": {
    "QueryString1": "queryValue1"
  },
  "pathParameters": {},
  "stageVariables": {
```

```
    "StageVar1": "stageValue1"
  },
  "requestContext": {
    "path": "/request",
    "accountId": "123456789012",
    "resourceId": "05c7jb",
    "stage": "test",
    "requestId": "...",
    "identity": {
      "apiKey": "...",
      "sourceIp": "...",
      "clientCert": {
        "clientCertPem": "CERT_CONTENT",
        "subjectDN": "www.example.com",
        "issuerDN": "Example issuer",
        "serialNumber": "a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1",
        "validity": {
          "notBefore": "May 28 12:30:02 2019 GMT",
          "notAfter": "Aug  5 09:36:04 2021 GMT"
        }
      }
    },
    "resourcePath": "/request",
    "httpMethod": "GET",
    "apiId": "abcdef123"
  }
}
```

`requestContext` 是键值对的映射，与 $context 变量相对应。其结果与 API 相关。

API Gateway 可能会向映射中添加新键。有关 Lambda 代理集成中的 Lambda 函数输入的更多信息，请参阅 用于代理集成的 Lambda 函数的输入格式。

来自 API Gateway Lambda 授权方的输出

Lambda 授权方函数的输出是类似于目录的对象，其中必须包含委托人标识符 (`principalId`) 以及包含策略语句列表的策略文档 (`policyDocument`)。输出还可以包含一个含键值对的 `context` 映射。如果 API 利用使用计划（`apiKeySource` 设置为 AUTHORIZER），则 Lambda 授权方函数必须返回使用计划的 API 密钥之一作为 `usageIdentifierKey` 属性值。

下面显示了一个此类输出的示例。

```
{
```

```
  "principalId": "yyyyyyyy", // The principal user identification associated with the
 token sent by the client.
  "policyDocument": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action": "execute-api:Invoke",
        "Effect": "Allow|Deny",
        "Resource": "arn:aws:execute-
api:{regionId}:{accountId}:{apiId}/{stage}/{httpVerb}/[{resource}/[{child-resources}]]"
      }
    ]
  },
  "context": {
    "stringKey": "value",
    "numberKey": "1",
    "booleanKey": "true"
  },
  "usageIdentifierKey": "{api-key}"
}
```

在此示例中，策略语句指定是允许还是拒绝 (`Effect`) API Gateway 执行服务调用 (`Action`) 指定的 API 方法 (`Resource`)。您可以使用通配符 (`*`) 指定资源类型（方法）。有关针对 API 调用设置有效策略的信息，请参阅在 API Gateway 中执行 API 的 IAM 策略的语句参考。

对于已启用授权的方法 ARN，例如 `arn:aws:execute-api:{regionId}:{accountId}:{apiId}/{stage}/{httpVerb}/[{resource}/[{child-resources}]]`，最大长度为 1600 字节。路径参数值（其大小在运行时确定）可能导致 ARN 长度超过限制。出现这种情况是时，API 客户端将收到 `414 Request URI too long` 响应。

此外，如授权方的策略声明中所示，资源 ARN 当前长度限制为 512 个字符。因此，您不能在请求 URI 中将 URI 与具有很大长度的 JWT 令牌一起使用。而是可以安全地在请求标头中传递 JWT 令牌。

您可以使用 `$context.authorizer.principalId` 变量访问映射模板中的 `principalId` 值。如果您想要将该值传递到后端，这会非常有用。有关更多信息，请参阅 适用于数据模型、授权方、映射模板和 CloudWatch 访问日志记录的 `$context` 变量。

您可以通过分别调用 `$context.authorizer.stringKey`、`$context.authorizer.numberKey` 或 `$context.authorizer.booleanKey` 来访问映射模板中 `context` 映射的 `stringKey`、`numberKey` 或 `booleanKey` 值（如 `"value"`、`"1"` 或 `"true"`）。返回的值都是字符串化的。请注意，您不能在 `context` 映射中将 JSON 对象或数组设置为任何键的有效值。

您可以使用 `context` 映射将缓存的凭证从授权方返回到后端（使用集成请求映射模板）。这使后端能够提供改善的用户体验，方式是使用缓存的凭证来减少为每个请求访问私有密钥并打开授权令牌的需要。

对于 Lambda 代理集成，API Gateway 将 `context` 对象作为 `event` 输入的一部分从 Lambda 授权方直接传递到后端 Lambda 函数。您可以通过调用 `$event.requestContext.authorizer.key`，在 Lambda 函数中检索 `context` 键值对。

`{api-key}` 表示 API 阶段的使用计划中的 API 键。有关更多信息，请参阅 the section called “使用计划”。

下面显示了来自示例 Lambda 授权方的示例输出。此示例输出包含一个策略语句，用于阻止 (Deny) 在 `dev` 阶段调用 AWS 账户 (`123456789012`) API (`ymy8tbxw7b`) 的 `GET` 方法。

```
{
  "principalId": "user",
  "policyDocument": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action": "execute-api:Invoke",
        "Effect": "Deny",
        "Resource": "arn:aws:execute-api:us-west-2:123456789012:ymy8tbxw7b/dev/GET/"
      }
    ]
  }
}
```

使用 API Gateway Lambda 授权方调用 API

配置 Lambda 授权方（以前称为自定义授权方）并部署 API 后，您应在启用 Lambda 授权方的情况下测试 API。为此，您需要一个 REST 客户端（如 cURL 或 Postman）。对于以下示例，我们使用 Postman。

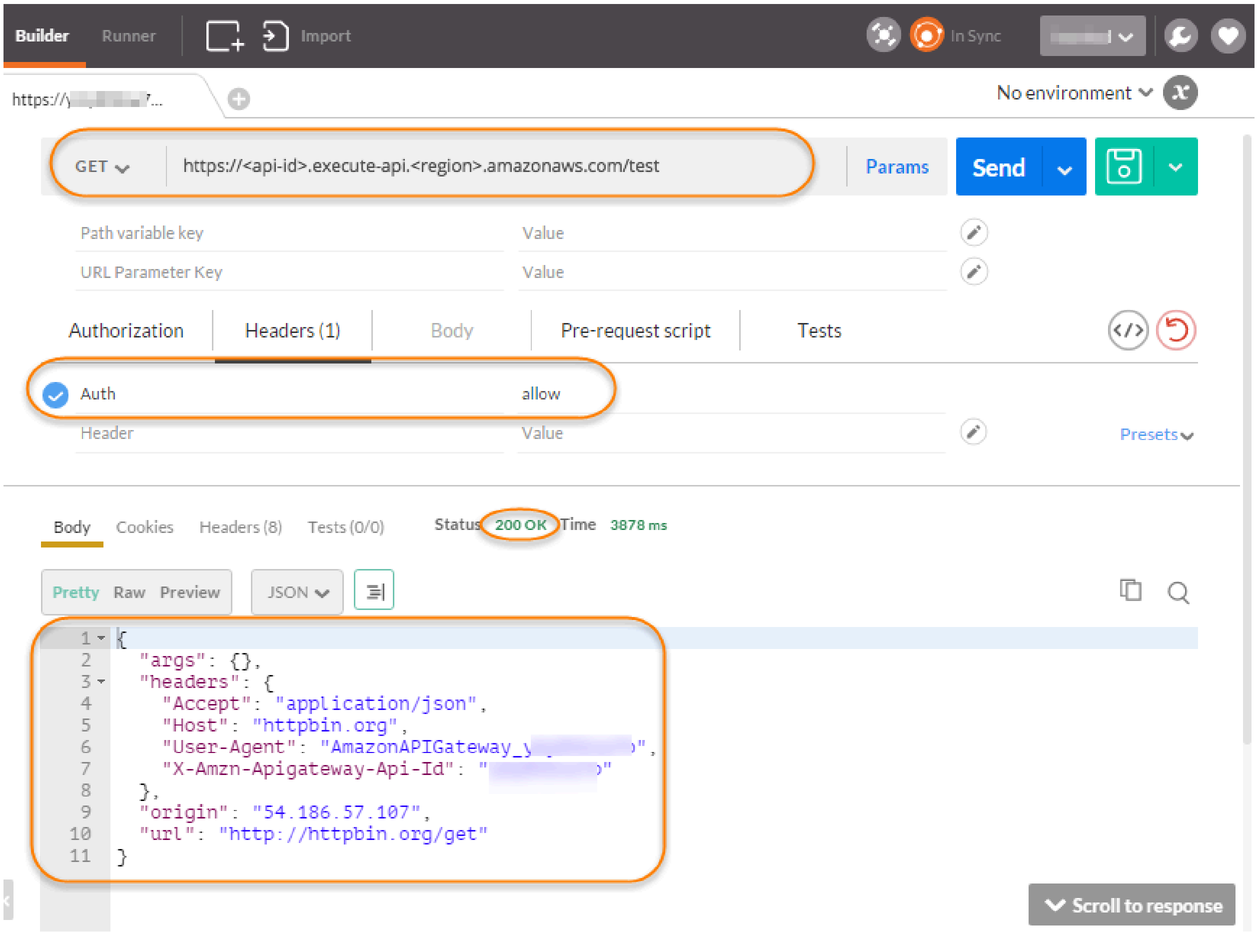
> **Note**
>
> 调用已启用授权方的方法时，如果 `TOKEN` 授权方所需的令牌未设置、为 null 或由指定的令牌验证表达式认定为无效，则 API Gateway 不会将该调用记录到 CloudWatch 中。同样，如果 `REQUEST` 授权方所需的任何身份来源未设置、为 null 或为空，则 API Gateway 不会将该调用记录到 CloudWatch 中。

在以下示例中，我们将说明如何在 Lambda TOKEN 授权方的情况下使用 Postman 调用或测试 API。如果您显式指定必需的路径、标头或查询字符串参数，则可应用此方法以使用 Lambda REQUEST 授权方调用 API。

使用自定义 **TOKEN** 授权方调用 API

1. 打开 Postman，选择 GET 方法，并将 API 的调用 URL 粘贴到相邻的 URL 字段中。

   添加 Lambda 授权令牌标头，并将值设置为 `allow`。选择发送。



GET https://<api-id>.execute-api.<region>.amazonaws.com/test

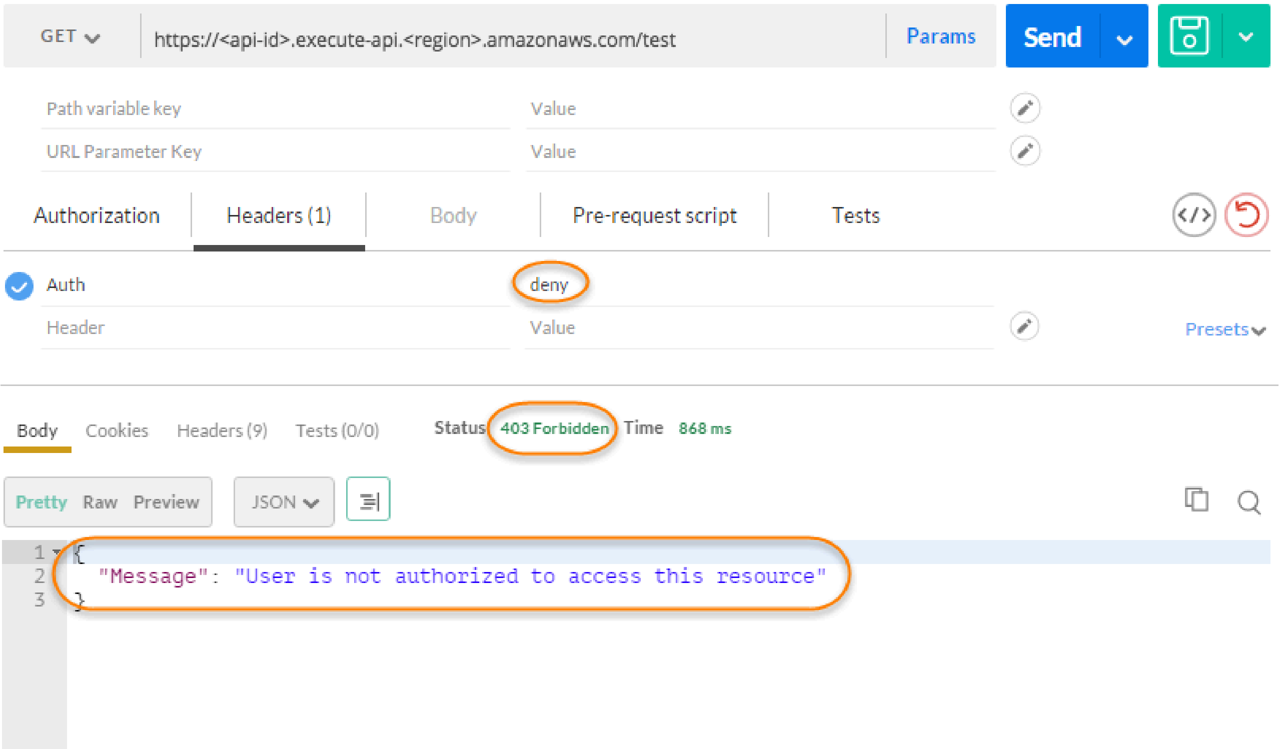
Auth: allow

Status 200 OK Time 3878 ms

```
{
  "args": {},
  "headers": {
    "Accept": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "AmazonAPIGateway_[redacted]",
    "X-Amzn-Apigateway-Api-Id": "[redacted]"
  },
  "origin": "54.186.57.107",
  "url": "http://httpbin.org/get"
}
```

   响应显示，API Gateway Lambda 授权方返回了 200 OK 响应，并成功授权调用访问与此方法集成的 HTTP 端点 (http://httpbin.org/get)。

2. 仍然是在 Postman 中，将 Lambda 授权令牌标头值更改为 `deny`。选择发送。



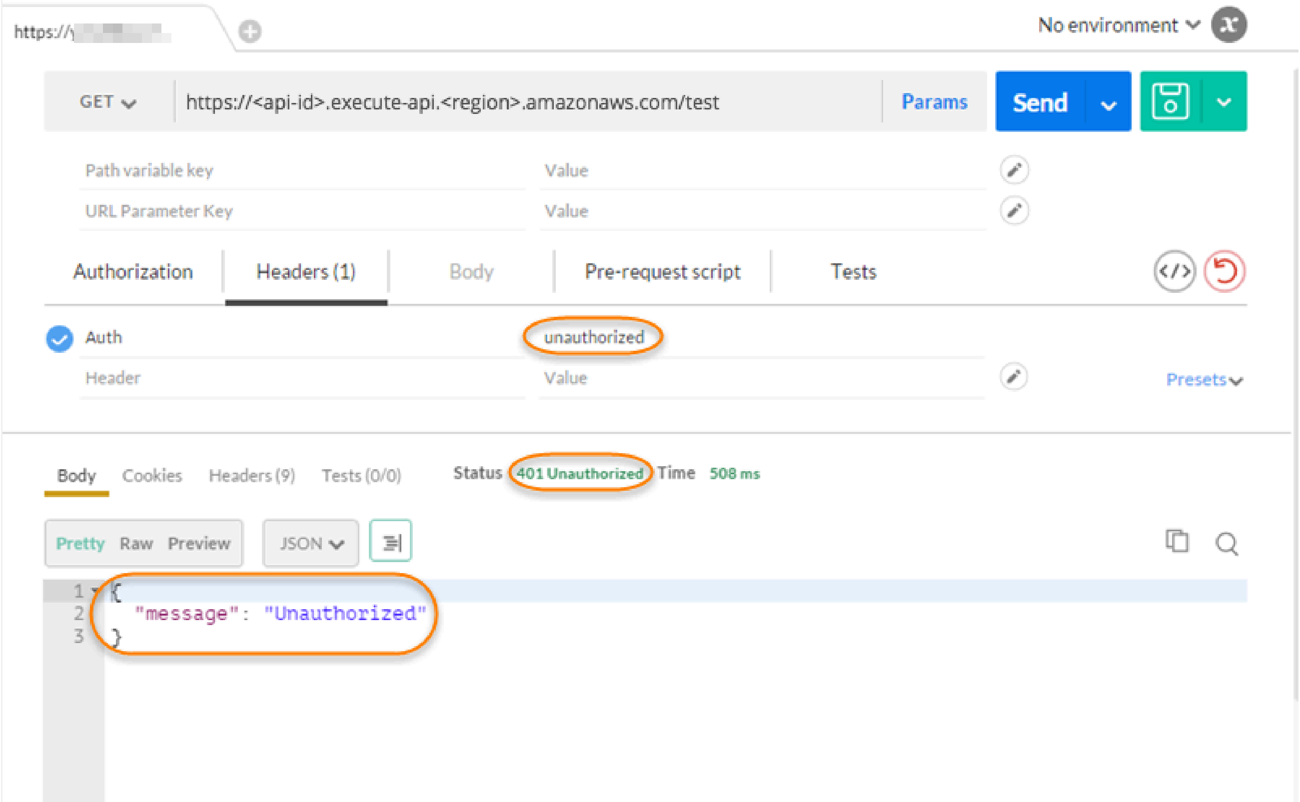
Auth: deny

Status 403 Forbidden Time 868 ms

```
{
  "Message": "User is not authorized to access this resource"
}
```

响应显示，API Gateway Lambda 授权方返回了 403 禁止访问响应，而未授权调用访问 HTTP 端点。

3. 在 Postman 中，将 Lambda 授权令牌标头值更改为 `unauthorized`，并选择发送。



GET https://<api-id>.execute-api.<region>.amazonaws.com/test

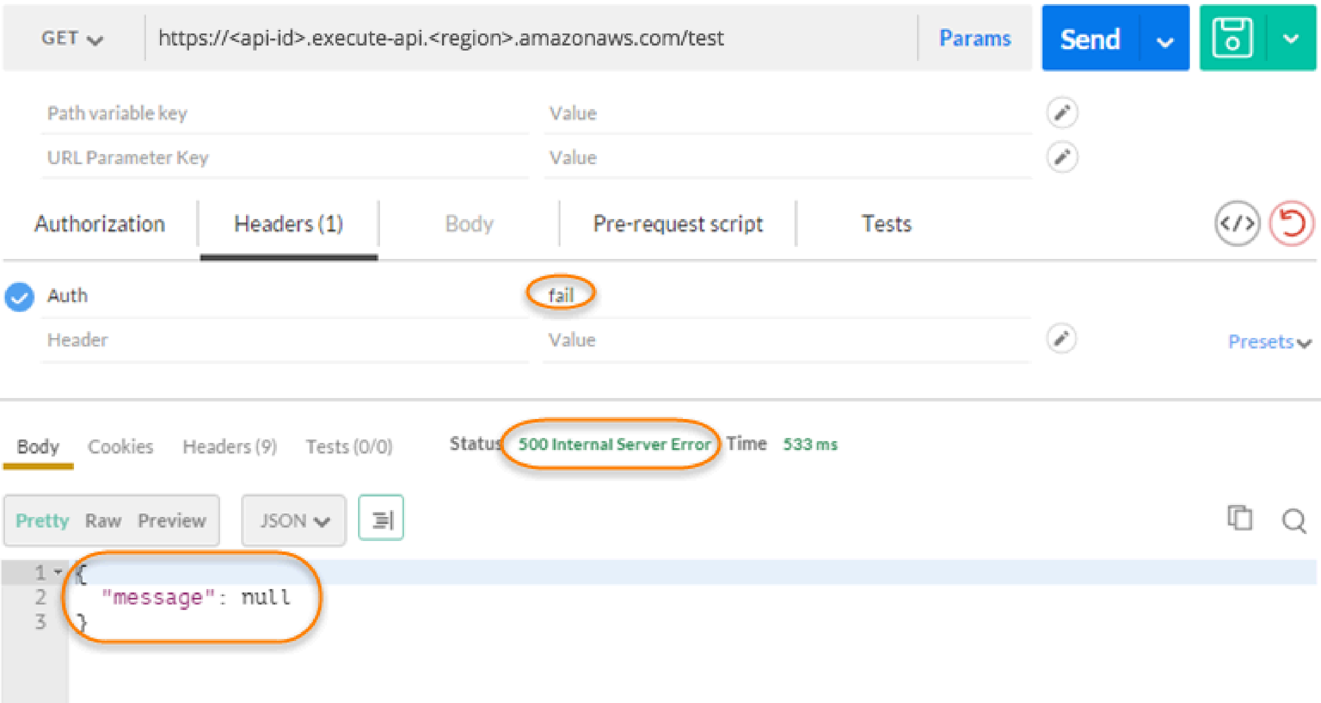
Auth: unauthorized

Status 401 Unauthorized Time 508 ms

```
{
  "message": "Unauthorized"
}
```

响应显示，API Gateway 返回了 401 未授权响应，而未授权调用访问 HTTP 端点。

4. 现在，将 Lambda 授权令牌标头值更改为 `fail`。选择发送。



GET https://<api-id>.execute-api.<region>.amazonaws.com/test

Auth: fail

Status 500 Internal Server Error Time 533 ms

```
{
  "message": null
}
```

响应显示，API Gateway 返回了 500 内部服务器错误响应，而未授权调用访问 HTTP 端点。

配置跨账户 Lambda 授权方

您现在还可以使用其他 AWS 账户中的 AWS Lambda 函数作为 API 授权方函数。每个账户都可以位于 Amazon API Gateway 可用的任何区域中。Lambda 授权方函数可使用持有者令牌身份验证策略，如 OAuth 或 SAML。这样便可轻松地跨多个 API Gateway API 集中管理和共享重要的 Lambda 授权方函数。

在本节中，我们将介绍如何使用 Amazon API Gateway 控制台配置跨账户 Lambda 授权方函数。

这些说明假设您已经在一个 AWS 账户中有 API Gateway API，在另一个账户中有 Lambda 授权方函数。

使用 API Gateway 控制台配置跨账户 Lambda 授权方

在您拥有 API 的账户中登录 Amazon API Gateway 控制台，然后执行以下操作：

1. 选择您的 API，然后在主导航窗格中选择授权方。
2. 选择创建授权方。
3. 对于授权方名称，输入授权方名称。
4. 对于授权方类型，选择 Lambda。
5. 对于 Lambda 函数，输入您第二个账户中的 Lambda 授权方函数的完整 ARN。

   > Note
   >
   > 在 Lambda 控制台中，您可以在控制台窗口的右上角找到函数的 ARN。

6. 此时将显示警告并带有 `aws lambda add-permission` 命令字符串。此策略授予 API Gateway 调用授权方 Lambda 函数的权限。复制命令并保存，以供稍后使用。您可在创建授权方后运行该命令。
7. 将 Lambda 调用角色留空，以允许 API Gateway 控制台设置基于资源的策略。此策略授予 API Gateway 调用授权方 Lambda 函数的权限。您还可选择输入 IAM 角色，以允许 API Gateway 调用授权方 Lambda 函数。有关此类角色的示例，请参阅创建一个可代入的 IAM 角色。
8. 对于 Lambda 事件负载，请选择令牌（对于 `TOKEN` 授权方）或请求（对于 `REQUEST` 授权方）。
9. 根据上一步的选择，执行下列操作之一：

   a. 对于令牌选项，执行下列操作：

- 对于令牌来源，输入包含授权令牌的标头名称。API 客户端必须包括此标头名称才能将授权令牌发送到 Lambda 授权方。
- 或者，对于令牌验证，输入正则表达式语句。API Gateway 将针对此表达式执行对输入令牌的初始验证并在成功验证后调用授权方。这有助于减少对您 API 的调用。
- 要缓存授权方生成的授权策略，请将授权缓存保持开启状态。在启用策略缓存后，您可选择修改 TTL 值。将 TTL 设置为零将禁用策略缓存。在启用策略缓存后，令牌来源中指定的标头名称将成为缓存键。如果在请求中将多个值传递到此标头，则所有值都将成为缓存键，并保留顺序。

> Note
>
> 默认 TTL 值为 300 秒。最大值为 3600 秒；目前不能提高此限制。

b. 对于请求选项，执行以下操作：

- 在身份源类型中，选择一个参数类型。支持的参数类型为 `Header`、`Query string`、`Stage variable` 和 `Context`。要添加更多身份来源，请选择添加参数。
- 要缓存授权方生成的授权策略，请将授权缓存保持开启状态。在启用策略缓存后，您可选择修改 TTL 值。将 TTL 设置为零将禁用策略缓存。

  API Gateway 将使用指定的身份来源作为请求授权方缓存键。在启用缓存后，API Gateway 将仅在成功确认所有指定的身份来源在运行时存在后才调用授权方的 Lambda 函数。如果指定的身份来源缺失、为 null 或为空，则 API Gateway 将返回 `401 Unauthorized` 响应，而不调用授权方 Lambda 函数。

  如果定义了多个身份来源，则它们都将用于派生授权方的缓存键。更改缓存键的任意部分将导致授权方丢弃缓存的策略文档并生成新的文档。如果在请求中传递了具有多个值的标头，则所有值都将成为缓存键的一部分，并保留顺序。
- 在关闭缓存后，无需指定身份来源。

> Note
>
> 要启用缓存，授权方必须返回适用于 API 中所有方法的策略。要强制执行特定于方法的策略，您可以关闭授权缓存。

10. 选择创建授权方。

11. 将您在上一步中复制的 `aws lambda add-permission` 命令字符串粘贴到为第二个账户配置的 AWS CLI 窗口中。使用您的授权方 ID 替换 `AUTHORIZER_ID`。这将向您的第一个账户授予对第二个账户的 Lambda 授权方函数的访问权限。

使用 Amazon Cognito 用户池作为授权方控制对 REST API 的访问

作为使用 IAM 角色和策略或 Lambda 授权方（以前称为自定义授权方）的替代方案，您可以使用 Amazon Cognito 用户池来控制谁可以在 Amazon API Gateway 中访问您的 API。

要将 Amazon Cognito 用户池与您的 API 一起使用，您必须先创建 `COGNITO_USER_POOLS` 类型的授权方，然后配置 API 方法以使用该授权方。部署 API 之后，客户端必须先将用户注册到用户池，获取用户的身份令牌或访问令牌，然后使用令牌之一调用 API 方法，这通常设置为请求的 `Authorization` 标头。只有提供了所需的令牌并且提供的令牌有效时，API 调用才会成功，否则，客户端未获得授权来执行调用，因为客户端没有可用于授权的凭证。

使用身份令牌，基于已登录用户的身份声明来授权 API 调用。使用访问令牌，基于指定访问受保护资源的自定义范围授权 API 调用。有关更多信息，请参阅将令牌与用户池结合使用和资源服务器和自定义范围。

要为 API 创建和配置 Amazon Cognito 用户池，请执行以下任务：

- 使用 Amazon Cognito 控制台、CLI/开发工具包或 API 创建用户池，或者使用由其他 AWS 账户拥有的用户池。
- 使用 API Gateway 控制台、CLI/开发工具包或 API 创建具有选定用户池的 API Gateway Authorizer。
- 使用 API Gateway 控制台、CLI/开发工具包或 API，在所选 API 方法上启用授权方。

要在启用了用户池时调用任意 API 方法，您的客户端执行以下任务：

- 使用 Amazon Cognito CLI/SDK 或 API 将用户注册到所选用户池，并获取身份令牌或访问令牌。要了解有关使用 SDK 的更多信息，请参阅使用 SDK 的 Amazon Cognito 代码示例。
- 使用客户端特定的框架调用已部署的 API Gateway API 并在 `Authorization` 标头中提供合适的令牌。

作为 API 开发人员，您必须向客户端开发人员提供用户池 ID、客户端 ID，并可能需要提供作为用户池的一部分定义的关联客户端密钥。

Note

要允许用户使用 Amazon Cognito 凭证登录，同时还获取用于 IAM 角色权限的临时凭证，请使用 Amazon Cognito 联合身份。对于每个 API 资源端点 HTTP 方法，将授权类型、类别 `Method Execution` 设置为 `AWS_IAM`。

在此部分中，我们介绍如何创建用户池、如何将 API Gateway API 与用户池集成，以及如何调用与用户池集成的 API。

主题

- 为 REST API 获取权限以创建 Amazon Cognito 用户池授权方
- 为 REST API 创建 Amazon Cognito 授权用户池
- 将 REST API 与 Amazon Cognito 用户池集成
- 调用与 Amazon Cognito 用户池集成的 REST API
- 使用 API Gateway 控制台为 REST API 配置跨账户 Amazon Cognito 授权方
- 使用 AWS CloudFormation 为 REST API 创建 Amazon Cognito 授权方

为 REST API 获取权限以创建 Amazon Cognito 用户池授权方

要使用 Amazon Cognito 用户池创建授权方，您必须具有 `Allow` 权限以使用选定 Amazon Cognito 用户池创建或更新授权方。以下 IAM 策略文档显示了此类权限的示例：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "apigateway:POST"
            ],
            "Resource": "arn:aws:apigateway:*::/restapis/*/authorizers",
            "Condition": {
                "ArnLike": {
                    "apigateway:CognitoUserPoolProviderArn": [
                        "arn:aws:cognito-idp:us-east-1:123456789012:userpool/us-
east-1_aD06NQmjO",
                        "arn:aws:cognito-idp:us-east-1:234567890123:userpool/us-
east-1_xJ1MQtPEN"
```

```
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "apigateway:PATCH"
            ],
            "Resource": "arn:aws:apigateway:*::/restapis/*/authorizers/*",
            "Condition": {
                "ArnLike": {
                    "apigateway:CognitoUserPoolProviderArn": [
                        "arn:aws:cognito-idp:us-east-1:123456789012:userpool/us-east-1_aD06NQmjO",
                        "arn:aws:cognito-idp:us-east-1:234567890123:userpool/us-east-1_xJ1MQtPEN"
                    ]
                }
            }
        }
    ]
}
```

确保将策略附加到您所属的 IAM 组或将您分配到的 IAM 角色。

在上一个策略文档中，`apigateway:POST` 操作用于创建新的授权方，`apigateway:PATCH` 操作用于更新现有授权方。您可以通过相应地覆盖 `Resource` 值的前两个通配符 (*)，将策略限制为特定区域或特定 API。

此处使用的 `Condition` 子句用于将 `Allowed` 权限限制为指定的用户池。当提供 `Condition` 子句时，对任何不满足条件的用户池的访问将被拒绝。当权限没有 `Condition` 子句时，将允许访问任何用户池。

您可以通过以下几种方式设置 `Condition` 子句：

- 您可以将 `ArnLike` 或 `ArnEquals` 条件表达式设置为仅允许使用指定的用户池创建或更新 `COGNITO_USER_POOLS` 授权方。
- 您可以将 `ArnNotLike` 或 `ArnNotEquals` 条件表达式设置为允许使用表达式中未指定的任何用户池创建或更新 `COGNITO_USER_POOLS` 授权方。
- 您可以省略 `Condition` 子句，以允许使用任何 AWS 账户和任何区域中的任何用户池创建或更新 `COGNITO_USER_POOLS` 授权方。

有关 Amazon Resource Name (ARN) 条件表达式的更多信息，请参阅 Amazon 资源名称条件运算符。如示例中所示，`apigateway:CognitoUserPoolProviderArn` 是 `COGNITO_USER_POOLS` 用户池的 ARN 的列表，它们可用于或者不能用于 `COGNITO_USER_POOLS` 类型的 API Gateway Authorizer。

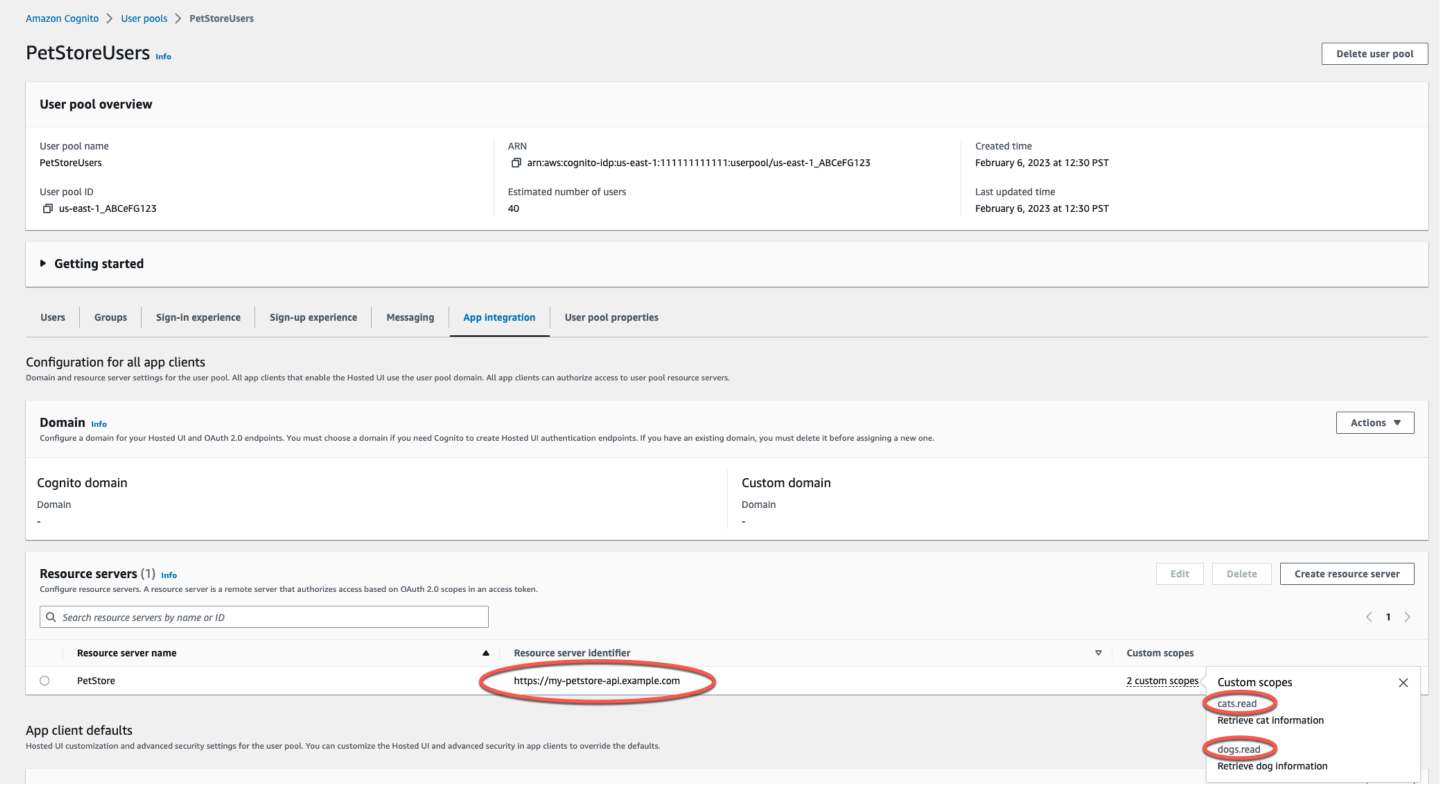
为 REST API 创建 Amazon Cognito 授权用户池

将 API 与用户池集成之前，您必须在 Amazon Cognito 中创建用户池。您的用户群体配置必须遵守全部 Amazon Cognito 的资源配额。所有用户定义的 Amazon Cognito 变量（如组、用户和角色）应仅使用字母数字字符。有关创建用户群体的说明，请参阅 Amazon Cognito 开发人员指南中的教程：创建用户池。

记下用户池 ID、客户端 ID 和任意客户端密钥。客户端必须将这些信息提供给 Amazon Cognito，供用户注册到用户池、登录用户池，并获取要包含在请求中的身份令牌或访问令牌，以便调用配置了用户池的 API 方法。此外，在您将用户池配置为 API Gateway 中的授权方时，您还必须指定用户池名称，如下所述。

如果您使用访问令牌授权 API 方法调用，请确保使用用户池配置应用程序集成，以在指定资源服务器上设置您需要的自定义范围。有关将令牌与 Amazon Cognito 用户池结合使用的更多信息，请参阅将令牌与用户池结合使用。有关资源服务器的更多信息，请参阅为您的用户池定义资源服务器。

记录配置的资源服务器标识符和自定义范围名称。您需要这些内容来构建 OAuth Scopes (OAuth 范围) 的访问范围完整名称，它由 `COGNITO_USER_POOLS` 授权方使用。



User pool name: PetStoreUsers

User pool ID: us-east-1_ABCeFG123

ARN: arn:aws:cognito-idp:us-east-1:111111111111:userpool/us-east-1_ABCeFG123

Estimated number of users: 40

Created time: February 6, 2023 at 12:30 PST

Last updated time: February 6, 2023 at 12:30 PST

Configuration for all app clients

Domain and resource server settings for the user pool. All app clients that enable the Hosted UI use the user pool domain. All app clients can authorize access to user pool resource servers.

Configure a domain for your Hosted UI and OAuth 2.0 endpoints. You must choose a domain if you need Cognito to create Hosted UI authentication endpoints. If you have an existing domain, you must delete it before assigning a new one.

Cognito domain — Domain: -

Custom domain — Domain: -

Configure resource servers. A resource server is a remote server that authorizes access based on OAuth 2.0 scopes in an access token.

	Resource server name	Resource server identifier	Custom scopes
○	PetStore	https://my-petstore-api.example.com	2 custom scopes

Custom scopes

cats.read
Retrieve cat information

dogs.read
Retrieve dog information

App client defaults

Hosted UI customization and advanced security settings for the user pool. You can customize the Hosted UI and advanced security in app clients to override the defaults.

将 REST API 与 Amazon Cognito 用户池集成

创建 Amazon Cognito 用户池后，您必须在 API Gateway 中创建一个使用该用户池的 `COGNITO_USER_POOLS` 授权方。以下过程介绍如何使用 API Gateway 控制台执行此操作。

> **Note**
>
> 您可以使用 [`CreateAuthorizer`] 操作来创建使用多个用户池的 `COGNITO_USER_POOLS` 授权方。一个 `COGNITO_USER_POOLS` 授权方最多可以使用 1000 个用户池。不能提高此限制。

> **Important**
>
> 在执行以下任意步骤之后，您都需要部署或重新部署 API 以便传播更改。有关部署 API 的更多信息，请参阅在 Amazon API Gateway 中部署 REST API。

使用 API Gateway 控制台创建 **`COGNITO_USER_POOLS`** 授权方

1. 在 API Gateway 中创建新的 API 或选择现有 API。
2. 在主导航窗格中，选择**授权方**。
3. 选择**创建授权方**。
4. 要配置新的授权方来使用用户池，请执行以下操作：

   a. 在**授权方名称**中，输入名称。
   b. 对于**授权方类型**，请选择 **Cognito**。
   c. 对于 **Cognito 用户群体**，请选择您创建 Amazon Cognito 的 AWS 区域并选择可用的用户群体。

      您可以使用阶段变量来定义用户池。对用户池使用以下格式：`arn:aws:cognito-idp:`*`us-east-2`*`:`*`111122223333`*`:userpool/${stageVariables.`*`MyUserPool`*`}`。
   d. 对于**令牌来源**，输入 **`Authorization`** 作为标头名称，以在用户成功登录时传递 Amazon Cognito 所返回的身份令牌或访问令牌。
   e. （可选）在**令牌验证**字段中输入一个正则表达式，以在使用 Amazon Cognito 对请求进行授权之前，验证身份令牌的 `aud`（受众）字段。请注意，在使用访问令牌时，由于访问令牌不包含 `aud` 字段，此验证将拒绝请求。
   f. 选择**创建授权方**。

5. 创建了 `COGNITO_USER_POOLS` 授权方之后，您可以通过提供从用户池预置的身份验证令牌来对其进行测试调用。您可以通过调用 Amazon Cognito 身份开发工具包来获取此身份令牌以执行用户登录。您也可以使用 `InitiateAuth` 操作。如果您未配置任何授权范围，API Gateway 会将提供的令牌视为身份令牌。

上述过程创建使用新创建的 Amazon Cognito 用户池的 `COGNITO_USER_POOLS` 授权方。根据在 API 方法上启用授权方的方法，您可以使用从集成用户池预置的身份令牌或访问令牌。

在方法上配置 **`COGNITO_USER_POOLS`** 授权方

1. 选择资源。选择新方法或选择现有方法。如有必要，请创建资源。
2. 在方法请求选项卡上的方法请求设置下，选择编辑。
3. 对于授权方，从下拉菜单中选择您刚刚创建的 Amazon Cognito 用户群体授权方。
4. 要使用身份令牌，请执行以下操作：
    a. 将授权范围保留为空。
    b. 如果需要，在集成请求中，在正文映射模板中添加 `$context.authorizer.claims['`*`property-name`*`']` 或 `$context.authorizer.claims.`*`property-name`* 表达式，从而将指定的身份声明属性从用户群体传递到后端。对于简单的属性名称，例如 `sub` 或 `custom-sub`，两种表示法相同。对于复杂的属性名称，例如 `custom:role`，不能使用点表示法。例如，以下映射表达式会将声明的 `sub` 和 `email` 的标准字段传递到后端：

```
{
 "context" : {
  "sub" : "$context.authorizer.claims.sub",
  "email" : "$context.authorizer.claims.email"
 }
}
```

如果您在配置用户池时声明了自定义声明字段，那么您可以用同样的方式来访问自定义字段。以下示例获取的是声明的自定义 `role` 字段：

```
{
 "context" : {
  "role" : "$context.authorizer.claims.role"
    }
```

```
}
```

如果自定义声明字段被声明为 `custom:role`，可以使用以下示例来获取声明的属性：

```
{
 "context" : {
  "role" : "$context.authorizer.claims['custom:role']"
    }
}
```

5. 要使用访问令牌，请执行以下操作：

    a. 对于授权范围，输入某个范围的一个或多个全名，该范围在创建 Amazon Cognito 用户池时已配置。例如，根据 为 REST API 创建 Amazon Cognito 授权用户池 中给出的示例，其中一个范围是 `https://my-petstore-api.example.com/cats.read`。

        在运行时，如果在此步骤的方法上指定的任何范围与在传入令牌中声明的范围匹配，则方法调用成功。否则，调用失败并出现 `401 Unauthorized` 响应。

    b. 选择保存。

6. 对您选择的其他方法重复这些步骤。

使用 `COGNITO_USER_POOLS` 授权方，如果未指定 OAuth 范围选项，则 API Gateway 将提供的令牌视为身份令牌，并根据来自用户池的身份之一验证所声明的身份。否则，API Gateway 将提供的令牌视为访问令牌，并根据在方法上声明的授权范围，验证在令牌中所声明的访问范围。

除了使用 API Gateway 控制台之外，您还可以指定 OpenAPI 定义文件并将 API 定义导入到 API Gateway，从而在方法上启用 Amazon Cognito 用户池。

用 OpenAPI 定义文件导入 COGNITO_USER_POOLS 授权方

1. 为您的 API 创建（或导出）一个 OpenAPI 定义文件。

2. 作为 OpenAPI 3.0 中的 `securitySchemes` 部分或 Open API 2.0 中的 `securityDefinitions` 部分，指定 `COGNITO_USER_POOLS` 授权方 (`MyUserPool`) JSON 定义，如下所示：

    OpenAPI 3.0

```
  "securitySchemes": {
    "MyUserPool": {
```

```
      "type": "apiKey",
      "name": "Authorization",
      "in": "header",
      "x-amazon-apigateway-authtype": "cognito_user_pools",
      "x-amazon-apigateway-authorizer": {
        "type": "cognito_user_pools",
        "providerARNs": [
          "arn:aws:cognito-idp:{region}:{account_id}:userpool/{user_pool_id}"
        ]
      }
    }
```

OpenAPI 2.0

```
  "securityDefinitions": {
    "MyUserPool": {
      "type": "apiKey",
      "name": "Authorization",
      "in": "header",
      "x-amazon-apigateway-authtype": "cognito_user_pools",
      "x-amazon-apigateway-authorizer": {
        "type": "cognito_user_pools",
        "providerARNs": [
          "arn:aws:cognito-idp:{region}:{account_id}:userpool/{user_pool_id}"
        ]
      }
    }
```

3. 要将身份令牌用于方法授权，请将 `{ "MyUserPool": [] }` 添加到方法的 `security` 定义，如根资源上的以下 GET 方法所示。

```
  "paths": {
    "/": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "text/html"
        ],
        "responses": {
          "200": {
            "description": "200 response",
```

```
          "headers": {
            "Content-Type": {
              "type": "string"
            }
          }
        }
      },
      "security": [
        {
          "MyUserPool": []
        }
      ],
      "x-amazon-apigateway-integration": {
        "type": "mock",
        "responses": {
          "default": {
            "statusCode": "200",
            "responseParameters": {
              "method.response.header.Content-Type": "'text/html'"
            },
          }
        },
        "requestTemplates": {
          "application/json": "{\"statusCode\": 200}"
        },
        "passthroughBehavior": "when_no_match"
      }
    },
    ...
  }
```

4. 要为方法授权使用访问令牌，请将以上安全定义更改为 `{ "MyUserPool": [resource-server/scope, ...] }`：

```
"paths": {
  "/": {
    "get": {
      "consumes": [
        "application/json"
      ],
      "produces": [
        "text/html"
      ],
```

```
        "responses": {
          "200": {
            "description": "200 response",
            "headers": {
              "Content-Type": {
                "type": "string"
              }
            }
          }
        },
        "security": [
          {
            "MyUserPool": ["https://my-petstore-api.example.com/cats.read",
 "http://my.resource.com/file.read"]
          }
        ],
        "x-amazon-apigateway-integration": {
          "type": "mock",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseParameters": {
                "method.response.header.Content-Type": "'text/html'"
              },
            }
          },
          "requestTemplates": {
            "application/json": "{\"statusCode\": 200}"
          },
          "passthroughBehavior": "when_no_match"
        }
      },
      ...
  }
```

5. 如果需要，您可以使用合适的 OpenAPI 定义或扩展来设置其他 API 配置设置。有关更多信息，请参阅 使用基于 OpenAPI 的 API Gateway 扩展。

调用与 Amazon Cognito 用户池集成的 REST API

要调用一个配置了用户池授权方的方法，客户端必须执行以下操作：

- 让用户注册到用户池中。

- 让用户登录用户池。
- 从用户池获取已登录用户的身份或访问令牌。
- 将令牌添加到 `Authorization` 标头（或您创建授权方时指定的其他标头）中。

您可以使用 AWS Amplify 来执行这些任务。有关更多信息，请参阅将 Amazon Cognito 与 Web 和移动应用程序集成。

- 对于 Android，请参阅 Amplify for Android 入门。
- 要使用 iOS，请参阅 Amplify for iOS 入门。
- 要使用 JavaScript，请参阅 Amplify for Javascript 入门。

使用 API Gateway 控制台为 REST API 配置跨账户 Amazon Cognito 授权方

您现在还可以使用来自不同 AWS 账户的 Amazon Cognito 用户池作为 API 授权方。Amazon Cognito 用户池可使用持有者令牌身份验证策略（如 OAuth 或 SAML）。这样就可以轻松地跨多个 API Gateway API 集中管理和共享中央 Amazon Cognito 用户池授权方。

在本节中，我们将介绍如何使用 Amazon API Gateway 控制台配置跨账户 Amazon Cognito 用户池。

这些说明假定您已在一个 AWS 账户中拥有 API Gateway API，在另一个账户中有 Amazon Cognito 用户池。

使用 API Gateway 控制台配置跨账户 Amazon Cognito 授权方

在您拥有 API 的账户中登录 Amazon API Gateway 控制台，然后执行以下操作：

1. 在 API Gateway 中创建新的 API 或选择现有 API。
2. 在主导航窗格中，选择授权方。
3. 选择创建授权方。
4. 要配置新的授权方来使用用户池，请执行以下操作：
    a. 在授权方名称中，输入名称。
    b. 对于授权方类型，请选择 Cognito。
    c. 对于 Cognito 用户群体，输入您第二个账户中的用户群体的完整 ARN。

> **Note**
>
> 在 Amazon Cognito 控制台中，您可以在常规设置窗格的池 ARN 字段找到您的用户池的 ARN。

d. 对于令牌来源，输入 **`Authorization`** 作为标头名称，以在用户成功登录时传递 Amazon Cognito 所返回的身份令牌或访问令牌。

e. （可选）在令牌验证字段中输入一个正则表达式，以在使用 Amazon Cognito 对请求进行授权之前，验证身份令牌的 aud（受众）字段。请注意，在使用访问令牌时，由于访问令牌不包含 aud 字段，此验证将拒绝请求。

f. 选择创建授权方。

使用 AWS CloudFormation 为 REST API 创建 Amazon Cognito 授权方

可以使用 AWS CloudFormation 创建 Amazon Cognito 用户群体和 Amazon Cognito 授权方。示例 AWS CloudFormation 模板执行以下操作：

- 创建 Amazon Cognito 用户群体。客户端必须先将用户登录到用户群体并获取身份或访问令牌。如果您使用访问令牌授权 API 方法调用，请确保使用用户池配置应用程序集成，以在指定资源服务器上设置您需要的自定义范围。
- 使用 `GET` 方法创建 API Gateway API。
- 创建使用 `Authorization` 标头作为令牌来源的 Amazon Cognito 授权方。

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  UserPool:
    Type: AWS::Cognito::UserPool
    Properties:
      AccountRecoverySetting:
        RecoveryMechanisms:
          - Name: verified_phone_number
            Priority: 1
          - Name: verified_email
            Priority: 2
      AdminCreateUserConfig:
        AllowAdminCreateUserOnly: true
      EmailVerificationMessage: The verification code to your new account is {####}
```

```
      EmailVerificationSubject: Verify your new account
      SmsVerificationMessage: The verification code to your new account is {####}
      VerificationMessageTemplate:
        DefaultEmailOption: CONFIRM_WITH_CODE
        EmailMessage: The verification code to your new account is {####}
        EmailSubject: Verify your new account
        SmsMessage: The verification code to your new account is {####}
    UpdateReplacePolicy: Retain
    DeletionPolicy: Retain
  CogAuthorizer:
    Type: AWS::ApiGateway::Authorizer
    Properties:
      Name: CognitoAuthorizer
      RestApiId:
        Ref: Api
      Type: COGNITO_USER_POOLS
      IdentitySource: method.request.header.Authorization
      ProviderARNs:
        - Fn::GetAtt:
            - UserPool
            - Arn
  Api:
    Type: AWS::ApiGateway::RestApi
    Properties:
      Name: MyCogAuthApi
  ApiDeployment:
    Type: AWS::ApiGateway::Deployment
    Properties:
      RestApiId:
        Ref: Api
    DependsOn:
      - CogAuthorizer
      - ApiGET
  ApiDeploymentStageprod:
    Type: AWS::ApiGateway::Stage
    Properties:
      RestApiId:
        Ref: Api
      DeploymentId:
        Ref: ApiDeployment
      StageName: prod
  ApiGET:
    Type: AWS::ApiGateway::Method
    Properties:
```

```
      HttpMethod: GET
      ResourceId:
        Fn::GetAtt:
          - Api
          - RootResourceId
      RestApiId:
        Ref: Api
      AuthorizationType: COGNITO_USER_POOLS
      AuthorizerId:
        Ref: CogAuthorizer
      Integration:
        IntegrationHttpMethod: GET
        Type: HTTP_PROXY
        Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets
Outputs:
  ApiEndpoint:
    Value:
      Fn::Join:
        - ""
        - - https://
          - Ref: Api
          - .execute-api.
          - Ref: AWS::Region
          - "."
          - Ref: AWS::URLSuffix
          - /
          - Ref: ApiDeploymentStageprod
          - /
```

设置 REST API 集成

设置 API 方法之后，您必须将其与后端中的端点集成。后端端点也称为集成端点，它可以是 Lambda 函数、HTTP 网页或 AWS 服务操作。

使用 API 方法，API 集成有一个集成请求和一个集成响应。集成请求封装后端收到的 HTTP 请求。它可能与客户端提交的方法请求不同或相同。集成响应是封装了后端返回的输出的 HTTP 响应。

设置集成请求涉及以下内容：配置如何将客户端提交的方法请求传递到后端；配置如何转换请求数据（如有必要）为集成请求数据；指定要调用的 Lambda 函数，指定将传入请求转发到哪个 HTTP 服务器，或者指定要调用的 AWS 服务操作。

设置集成响应（仅适用于非代理集成）涉及以下内容：配置如何传递后端返回的结果到具有指定状态代码的方法响应，配置如何转换指定的集成响应参数为预配置的方法响应参数，以及配置如何根据指定的正文映射模板将集成响应正文映射到方法响应正文。

集成请求由 API Gateway 的 `Integration` 资源以编程方式封装，集成响应由 `IntegrationResponse` 资源以编程方式封装。

要设置集成请求，您需要创建 `Integration` 资源并用它来配置集成端点 URL。然后，您可以设置 IAM 权限以访问后端，并指定映射以在将传入请求数据传递到后端之前进行转换。要为非代理集成设置集成响应，您需要创建 `IntegrationResponse` 资源并用它设置其目标方法响应。然后，您可以配置如何将后端输出映射到方法响应。

主题

- 在 API Gateway 中设置集成请求
- 在 API Gateway 中设置集成响应
- 在 API Gateway 中设置 Lambda 集成
- 在 API Gateway 中设置 HTTP 集成
- 设置 API Gateway 私有集成
- 在 API Gateway 中设置模拟集成

在 API Gateway 中设置集成请求

要设置集成请求，您可以执行以下必需任务和可选任务：

1. 选择确定如何将方法请求数据传递到后端的集成类型。
2. 对于非模拟集成，请指定 HTTP 方法和目标集成端点的 URI，但 `MOCK` 集成除外。
3. 对于与 Lambda 函数和其他 AWS 服务操作的集成，请设置具有所需权限的 IAM 角色，以便 API Gateway 代表您调用后端。
4. 对于非代理集成，设置必要的参数映射，将预定义方法请求参数映射到合适的集成请求参数。
5. 对于非代理集成，设置必要的正文映射，根据指定的映射模板来映射给定内容类型的传入方法请求正文。
6. 对于非代理集成，指定传入方法请求按原样传递到后端的条件。
7. （可选）指定如何处理二进制负载的类型转换。
8. （可选）声明缓存命名空间名称和缓存密钥参数以启用 API 缓存。

执行这些任务涉及到创建 API Gateway 的集成资源以及设置合适的属性值。您可以使用 API Gateway 控制台、AWS CLI 命令、AWS 开发工具包或 API Gateway REST API 来执行此操作。

主题

- API 集成请求的基本任务
- 选择 API Gateway API 集成类型
- 设置具有代理资源的代理集成
- 使用 API Gateway 控制台设置 API 集成请求

API 集成请求的基本任务

集成请求是 API Gateway 提交到后端的 HTTP 请求，与客户端提交的请求数据一起传递，并在需要时转换数据。HTTP 方法（或动词）和集成请求的 URI 由后端（即，集成端点）决定。它们分别可以与方法请求的 HTTP 方法和 URI 相同或不同。

例如，当 Lambda 函数返回从 Amazon S3 提取的文件时，您可以直观地公开此操作，作为对客户端的 `GET` 方法请求，即使对应的集成请求需要使用 `POST` 请求来调用 Lambda 函数。对于 HTTP 端点，有可能方法请求和对应的集成请求均使用相同的 HTTP 动词。但这不是必需的。您可以集成以下方法请求：

```
GET /{var}?query=value
Host: api.domain.net
```

使用以下集成请求：

```
POST /
Host: service.domain.com
Content-Type: application/json
Content-Length: ...

{
   path: "{var}'s value",
   type: "value"
}
```

作为 API 开发人员，您可以为方法请求使用任意满足需求的 HTTP 动词和 URI。但是，您必须遵守集成端点的要求。当方法请求数据不同于集成请求数据时，您可以通过提供从方法请求数据到集成请求数据的映射来协调差异。

在前面的示例中，映射将 `{var}` 方法请求的路径变量 (query) 和查询参数 (GET) 值转换为集成请求的负载属性 `path` 和 `type` 的值。其他可映射请求数据包含请求标头和正文。使用 API Gateway 控制台设置请求和响应数据映射 中介绍了这些内容。

设置 HTTP 或 HTTP 代理集成请求时，您分配后端 HTTP 端点 URL 作为集成请求 URI 值。例如，在 PetStore API 中，用于获取一页宠物的方法请求具有以下集成请求 URI：

```
http://petstore-demo-endpoint.execute-api.com/petstore/pets
```

在设置 Lambda 或 Lambda 代理集成时，您分配 Amazon Resource Name (ARN) 用于调用 Lambda 函数作为集成请求 URI 值。此 ARN 具有以下格式：

```
arn:aws:apigateway:api-region:lambda:path//2015-03-31/functions/arn:aws:lambda:lambda-region:account-id:function:lambda-function-name/invocations
```

`arn:aws:apigateway:`*`api-region`*`:lambda:path/` 之后的部分，即 `/2015-03-31/functions/arn:aws:lambda:`*`lambda-region`*`:`*`account-id`*`:function:`*`lambda-function-name`*`/invocations`，是 Lambda 调用操作的 REST API URI 路径。如果您使用 API Gateway 控制台来设置 Lambda 集成，API Gateway 会在提示您从区域中选择 *`lambda-function-name`* 之后，创建 ARN 并将其分配给集成 URI。

使用其他 AWS 服务操作设置集成请求时，集成请求 URI 也是 ARN，类似于使用 Lambda `Invoke` 操作的集成。例如，对于具有 Amazon S3 的 GetBucket 操作的集成，集成请求 URI 是以下格式的 ARN：

```
arn:aws:apigateway:api-region:s3:path/{bucket}
```

集成请求 URI 是用于指定操作的路径约定，其中 *`{bucket}`* 是存储桶名称的占位符。此外，AWS 服务操作可以按其名称引用。使用操作名称，Amazon S3 的 `GetBucket` 操作的集成请求 URI 成为以下内容：

```
arn:aws:apigateway:api-region:s3:action/GetBucket
```

使用基于操作的集成请求 URI，存储桶名称 (*`{bucket}`*) 必须在集成请求正文中指定 (`{ Bucket: "`*`{bucket}`*`" }`)，按照 `GetBucket` 操作的输入格式。

对于AWS集成，您还必须配置凭证以允许 API Gateway 调用集成操作。您可以为 API Gateway 创建新的 IAM 角色或选择现有此类角色以调用操作，然后使用其 ARN 指定角色。下面显示了此 ARN 的示例：

```
arn:aws:iam::account-id:role/iam-role-name
```

此 IAM 角色必须包含策略以允许执行操作。它还必须将 API Gateway 声明（在角色的信任关系中）作为可信实体以代入角色。此类权限可以在操作本身上授予。他们被称为基于资源的权限。对于 Lambda 集成，您可以调用 Lambda 的 addPermission 操作来设置基于资源的权限，然后在 API Gateway 集成请求中将 `credentials` 设置为 null。

我们讨论了基本集成设置。高级设置涉及到将方法请求数据映射到集成请求数据。讨论集成响应的基本设置之后，我们将介绍 使用 API Gateway 控制台设置请求和响应数据映射 中的高级主题，其中还会介绍传递负载和处理内容编码。

选择 API Gateway API 集成类型

您可以根据所使用的集成端点的类型以及希望如何与集成端点往返传输数据，来选择 API 集成类型。对于 Lambda 函数，您可以使用 Lambda 代理集成或 Lambda 自定义集成。对于 HTTP 端点，您可以使用 HTTP 代理集成，或者 HTTP 自定义集成。对于 AWS 服务操作，您只能使用非代理类型的 AWS 集成。API Gateway 还支持模拟集成，在这种情况下，API Gateway 用作集成端点来响应方法请求。

Lambda 自定义集成是AWS集成的特殊用例，其中集成端点对应于 Lambda 服务的函数调用操作。

您可以采用编程方式，通过设置 `Integration` 资源中的 `type` 属性来选择集成类型。对于 Lambda 代理集成，该值为 `AWS_PROXY`。对于 Lambda 自定义集成以及所有其他 AWS 集成，这是 `AWS`。对于 HTTP 代理集成和 HTTP 集成，该值分别为 `HTTP_PROXY` 和 `HTTP`。对于模拟集成，`type` 值为 `MOCK`。

Lambda 代理集成支持通过单个 Lambda 函数来简化集成设置。设置很简单，并且可以随后端演变，而无需停用现有设置。出于这些原因，强烈建议使用 Lambda 函数集成。

相比之下，Lambda 自定义集成允许将配置的映射模板用于各种具有类似输入和输出数据格式要求的集成端点。设置会更为复杂，建议用于更高级的应用程序场景。

同样，HTTP 代理集成具有简化的集成设置，可以随后端演变，而无需停用现有设置。HTTP 自定义集成的设置更为复杂，但允许将配置的映射模板重新用于其他集成端点。

下表汇总了支持的端点类型：

- `AWS`：这种类型的集成使得 API 可以公开 AWS 服务操作。在 `AWS` 集成中，您必须同时配置集成请求和集成响应，并设置从方法请求到集成请求以及从集成响应到方法响应的必需数据映射。

- `AWS_PROXY`：这种类型的集成让 API 方法可以通过灵活、多功能和简化的集成设置与 Lambda 函数调用操作集成。这种集成依赖于客户端与集成的 Lambda 函数之间的直接交互。

  借助于这类集成，也称为 Lambda 代理集成，您无需设置集成请求或集成响应。API Gateway 将来自客户端的传入请求作为输入传递给后端 Lambda 函数。集成 Lambda 函数采用此格式的输入并解析来自所有可用源的输入，包括请求标头、URL 路径变量、查询字符串参数以及适用的正文。函数返回的结构遵守此输出格式。

  这是通过 API Gateway 调用 Lambda 函数的首选集成类型，不适用于任何其他 AWS 服务操作，包括函数调用操作之外的 Lambda 操作。
- `HTTP`：这种类型的集成允许 API 在后端中公开 HTTP 端点。借助于 HTTP 集成，也称为 HTTP 自定义集成，您必须同时配置集成请求和集成响应。您必须设置从方法请求到集成请求以及从集成响应到方法响应的必需映射。
- `HTTP_PROXY`：HTTP 代理集成允许客户端使用单个 API 方法上的简化集成设置来访问后端 HTTP 端点。您无需设置集成请求或集成响应。API Gateway 将来自客户端的传入请求传递到 HTTP 端点，并将来自 HTTP 端点的传出响应传递到客户端。
- `MOCK`：这种类型的集成允许 API Gateway 返回响应而不将请求进一步发送给后端。这对于 API 测试非常有用，因为它可用于测试集成设置而不会导致使用后端的收费，并可以实现协作开发 API。

  在协作开发中，团队可以通过使用 `MOCK` 集成，设置其他团队拥有的 API 组件的模拟，从而隔离其开发工作。它还可用于返回与 CORS 相关的标头以确保 API 方法允许 CORS 访问。事实上，API Gateway 控制台集成 `OPTIONS` 方法以通过模拟集成来支持 CORS。网关响应是模拟集成的其他示例。

设置具有代理资源的代理集成

要在 API Gateway API 中设置具有代理资源的代理集成，您需要执行以下任务：

- 创建一个“贪婪”路径变量为 `{proxy+}` 的代理资源。
- 在代理资源上设置 `ANY` 方法。
- 使用 HTTP 或 Lambda 集成类型将资源和方法与后端相集成。

> **Note**
>
> “贪婪”路径变量、`ANY` 方法和代理集成类型通常结合使用，但它们都是独立的功能。您可以在“贪婪”资源上配置特定 HTTP 方法，也可以将非代理集成类型应用于代理资源。

在通过 Lambda 代理集成或 HTTP 代理集成处理方法时，API Gateway 存在某些限制。有关详细信息，请参阅the section called “重要提示”。

> Note
>
> 将代理集成与传递搭配使用时，如果未指定负载的内容类型，则 API Gateway 将返回默认的 `Content-Type:application/json` 标头。

使用以下 HTTP 代理集成或 Lambda 代理集成将代理资源与后端集成时，该代理资源的功能最为强大。

具有代理资源的 HTTP 代理集成

HTTP 代理集成，在 API Gateway REST API 中由 `HTTP_PROXY` 指定，用于将方法请求与后端 HTTP 终端节点集成。使用此集成类型，API Gateway 只需遵守特定限制在前端和后端之间传递整个请求和响应。

> Note
>
> HTTP 代理集成支持多值标头和查询字符串。

将 HTTP 代理集成用于代理资源时，您可以设置 API 来公开采用单个集成设置的 HTTP 后端的部分或整个终端节点层次结构。例如，假设将网站后端组织成多个脱离根节点 (`/site`) 的树节点的多个分支，如 /site/a_0/a_1/.../a_N、/site/b_0/b_1/.../b_M 等。如果将 `ANY` 的代理资源上的 `/api/{proxy+}` 方法与 URL 路径为 `/site/{proxy}` 的后端终端节点集成，则单个集成请求可以支持任何 HTTP 操作（GET、POST 等），在任何 [a_0, a_1, ..., a_N, b_0, b_1, ...b_M, ...] 上都如此。如果您将代理集成应用于特定的 HTTP 方法（如 `GET`），则生成的集成请求将与任何后端节点上的指定（即 `GET`）操作配合发挥作用。

具有代理资源的 Lambda 代理集成

Lambda 代理集成，在 API Gateway REST API 中由 `AWS_PROXY` 指定，用于将方法请求与后端中的 Lambda 函数集成。使用这种集成类型，API Gateway 会应用默认映射模板将整个请求发送到 Lambda 函数，并将 Lambda 函数的输出转换为 HTTP 响应。

同样，您可以将 Lambda 代理集成应用于 `/api/{proxy+}` 的代理资源以设置单个集成，从而让后端 Lambda 函数单独响应 `/api` 下任何 API 资源的更改。

使用 API Gateway 控制台设置 API 集成请求

API 方法设置定义该方法并描述其行为。要设置方法，您必须指定一个包含根 (“/”) 的资源以便在其中公开该方法，并指定 HTTP 方法（`GET`、`POST` 等）以及该方法如何与目标后端集成。方法请求和响应指定与调用应用程序的协定，其中规定 API 可以接收哪些参数以及响应是什么样子。

以下过程介绍如何使用 API Gateway 控制台创建集成请求。

主题

- 设置 Lambda 集成
- 设置 HTTP 集成
- 设置 AWS 服务集成
- 设置模拟集成

设置 Lambda 集成

使用 Lambda 函数集成，来将您的 API 与 Lambda 函数集成。在 API 级别，如果您创建非代理集成，则这是 `AWS` 集成类型；如果您创建代理集成，则这是 `AWS_PROXY` 集成类型。

设置 Lambda 集成

1. 在资源窗格中，选择创建方法。
2. 对于方法类型，选择一种 HTTP 方法。
3. 对于集成类型，选择 Lambda 函数。
4. 要使用 Lambda 代理集成，请开启 Lambda 代理集成。要了解有关 Lambda 代理集成的更多信息，请参阅the section called “ 了解 Lambda 代理集成 ”。
5. 对于 Lambda 函数，输入函数 Lambda 的名称。

   如果您在与您的 API 不同的区域中使用 Lambda 函数，请从下拉菜单中选择该区域并输入 Lambda 函数的名称。如果您使用的是跨账户 Lambda 函数，请输入函数 ARN。
6. 要使用默认超时值 29 秒，请保持默认超时处于开启状态。要设置自定义超时，请选择默认超时，然后输入一个介于 `50` 到 `29000` 毫秒之间的超时值。
7. （可选）您可以使用以下下拉菜单配置方法请求设置。选择方法请求设置并配置您的方法请求。有关更多信息，请参阅the section called “在控制台中编辑方法请求”的步骤 3。

   您也可以在创建方法后配置方法请求设置。

8. 选择创建方法。

设置 HTTP 集成

使用 HTTP 集成将您的 API 与 HTTP 端点集成。在 API 级别，这是 HTTP 集成类型。

设置 HTTP 集成

1. 在资源窗格中，选择创建方法。
2. 对于方法类型，选择一种 HTTP 方法。
3. 对于集成类型，选择 HTTP。
4. 要使用 HTTP 代理集成，请开启 HTTP 代理集成。要了解有关 HTTP 代理集成的更多信息，请参阅the section called “在 API Gateway 中设置 HTTP 代理集成”。
5. 对于 HTTP 方法，选择与 HTTP 后端中的方法最匹配的 HTTP 方法类型。
6. 对于端点 URL，输入您希望此方法使用的 HTTP 后端的 URL。
7. 对于内容处理，请选择内容处理行为。
8. 要使用默认超时值 29 秒，请保持默认超时处于开启状态。要设置自定义超时，请选择默认超时，然后输入一个介于 50 到 29000 毫秒之间的超时值。
9. （可选）您可以使用以下下拉菜单配置方法请求设置。选择方法请求设置并配置您的方法请求。有关更多信息，请参阅the section called “在控制台中编辑方法请求”的步骤 3。

   您也可以在创建方法后配置方法请求设置。
10. 选择创建方法。

设置 AWS 服务集成

使用 AWS 服务集成将您的 API 直接与 AWS 服务集成。在 API 级别，这是 AWS 集成类型。

设置 API Gateway API 执行以下任一操作：

- 新建 Lambda 函数。
- 设置 Lambda 函数的资源权限。
- 执行任何其它 Lambda 服务操作。

您必须选择 AWS 服务。

设置 AWS 服务集成

1. 在资源窗格中，选择创建方法。
2. 对于方法类型，选择一种 HTTP 方法。
3. 对于集成类型，选择 AWS 服务。
4. 对于 AWS 区域，选择您希望此方法用于调用操作的 AWS 区域。
5. 对于 AWS 服务，选择您希望此方法调用的 AWS 服务。
6. 对于 AWS 子域，请输入 AWS 服务使用的子域。通常，您会将此项保留为空。某些 AWS 服务可以支持子域作为主机的一部分。有关可用性和详细信息 (如果有)，请参阅服务文档。
7. 对于 HTTP 方法，选择与操作对应的 HTTP 方法类型。对于 HTTP 方法类型，请参阅您为 AWS 服务选择的 AWS 服务的 API 参考文档。
8. 对于操作类型，选择使用操作名称以使用 API 操作，或选择使用路径覆盖以使用自定义资源路径。有关可用的操作和自定义资源路径，请参阅您为 AWS 服务选择的 AWS 服务的 API 参考文档。
9. 输入操作名称或路径覆盖。
10. 对于执行角色，输入该方法将用于调用操作的 IAM 角色的 ARN。

    要创建 IAM 角色，您可以调整the section called “步骤 1：创建 AWS 服务代理执行角色”中的说明。使用所需数量的操作和资源语句，指定以下格式的访问策略：

    ```
    {
      "Version": "2012-10-17",
      "Statement": [
        {
          "Effect": "Allow",
          "Action": [
            "action-statement"
          ],
          "Resource": [
            "resource-statement"
          ]
        },
        ...
      ]
    }
    ```

    有关操作和资源语句语法，请参阅您为 AWS 服务选择的 AWS 服务的文档。

有关 IAM 角色的信任关系，请指定以下内容，使 API Gateway 代表您的 AWS 账户采取行动：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

11. 要使用默认超时值 29 秒，请保持默认超时处于开启状态。要设置自定义超时，请选择默认超时，然后输入一个介于 `50` 到 `29000` 毫秒之间的超时值。
12. （可选）您可以使用以下下拉菜单配置方法请求设置。选择方法请求设置并配置您的方法请求。有关更多信息，请参阅the section called “在控制台中编辑方法请求”的步骤 3。

    您也可以在创建方法后配置方法请求设置。
13. 选择创建方法。

设置模拟集成

如果您希望 API Gateway 充当您的后端来返回静态响应，则使用模拟集成。在 API 级别，这是 `MOCK` 集成类型。通常，如果您的 API 尚未最终确定，但您想生成 API 响应以便不妨碍相关团队进行测试，则可使用 `MOCK` 集成。对于 `OPTION` 方法，API Gateway 将 `MOCK` 集成设置为默认值以便为应用的 API 资源返回支持 CORS 的标头。

设置模拟集成

1. 在资源窗格中，选择创建方法。
2. 对于方法类型，选择一种 HTTP 方法。
3. 对于集成类型，选择模拟。
4. （可选）您可以使用以下下拉菜单配置方法请求设置。选择方法请求设置并配置您的方法请求。有关更多信息，请参阅the section called “在控制台中编辑方法请求”的步骤 3。

您也可以在创建方法后配置方法请求设置。

5. 选择创建方法。

在 API Gateway 中设置集成响应

对于非代理集成，您必须设置至少一个集成响应，并使其作为默认响应，将后端返回的结果传递到客户端。您可以选择按原样传递结果，或者在具有不同格式时，将集成响应数据转换为方法响应数据。

对于代理集成，API Gateway 会自动将后端输出传递到客户端作为 HTTP 响应。您无需设置集成响应或方法响应。

要设置集成响应，您可以执行以下必需任务和（可选）任务：

1. 指定集成响应数据要映射到的方法响应的 HTTP 状态代码。该项为必填项。
2. 定义正则表达式以选择要由此集成响应表示的后端输出。如果您将此项留空，则响应为拥有捕获任何尚未配置响应的默认响应。
3. 如果需要，声明由键/值对组成的映射，用于将指定的集成响应参数映射到指定的方法响应参数。
4. 如果需要，请添加正文映射模板来将指定集成响应负载转换为指定方法响应负载。
5. 如果需要，指定如何处理二进制负载的类型转换。

集成响应是封装了后端响应的 HTTP 响应。对于 HTTP 端点，后端响应是 HTTP 响应。集成响应状态代码可以采用后端返回的状态代码，集成响应正文是后端返回的负载。对于 Lambda 端点，后端响应是从 Lambda 函数返回的输出。借助 Lambda 集成，Lambda 函数输出作为 `200 OK` 响应返回。负载可以包含 JSON 数据格式的结果，包括 JSON 字符串或 JSON 对象，或者作为 JSON 对象的错误消息。您可以将正则表达式分配到 selectionPattern 属性，以将错误响应映射到适当的 HTTP 错误响应。有关 Lambda 函数错误响应的更多信息，请参阅处理 API Gateway 中的 Lambda 错误。对于 Lambda 代理集成，Lambda 函数必须返回以下格式的输出：

```
{
    statusCode: "...",            // a valid HTTP status code
    headers: {
        custom-header: "..."      // any API-specific custom header
    },
    body: "...",                  // a JSON string.
    isBase64Encoded:  true|false  // for binary support
}
```

不需要将 Lambda 函数响应映射到其正确的 HTTP 响应。

要将结果返回到客户端，请设置集成响应将端点响应按原样传递到对应的方法响应。或者，您可以将端点响应数据映射到方法响应数据。可以映射的响应数据包括响应状态代码、响应标头参数和响应正文。如果没有为返回的状态代码定义方法响应，API Gateway 返回 500 错误。有关更多信息，请参阅 使用映射模板覆盖 API 的请求和响应参数以及状态代码。

在 API Gateway 中设置 Lambda 集成

您可以使用 Lambda 代理集成或 Lambda 非代理（自定义）集成将 API 方法与 Lambda 函数集成。

在 Lambda 代理集成中，所需的设置非常简单。将集成的 HTTP 方法设置为 POST，将集成端点 URI 指向特定 Lambda 函数的 Lambda 函数调用操作的 ARN，并授予 API Gateway 代表您调用 Lambda 函数的权限。

在 Lambda 非代理集成中，除了执行代理集成设置步骤，您还需要指定如何将传入请求数据映射到集成请求以及如何将生成的集成响应数据映射到方法响应。

主题

- 在 API Gateway 中设置 Lambda 代理集成
- 在 API Gateway 中设置 Lambda 自定义集成
- 设置后端 Lambda 函数的异步调用
- 处理 API Gateway 中的 Lambda 错误

在 API Gateway 中设置 Lambda 代理集成

主题

- 了解 API Gateway Lambda 代理集成
- 支持多值标头和查询字符串参数
- 设置具有 Lambda 代理集成的代理资源
- 使用 AWS CLI 设置 Lambda 代理集成
- 用于代理集成的 Lambda 函数的输入格式
- 用于代理集成的 Lambda 函数的输出格式

了解 API Gateway Lambda 代理集成

Amazon API Gateway Lambda 代理集成是通过设置单个 API 方法来构建 API 的简单、强大且灵活的机制。Lambda 代理集成允许客户端调用后端的单个 Lambda 函数。该函数访问其他 AWS 服务的许多资源或功能，包括调用其他 Lambda 函数。

在 Lambda 代理集成中，当客户端提交 API 请求时，API Gateway 将事件对象传递给集成的 Lambda 函数，但不会保留请求参数的顺序。此请求数据包括请求标头、查询字符串参数、URL 路径变量、负载和 API 配置数据。配置数据可以包括当前部署阶段名称、阶段变量、用户身份或授权上下文（如果有）。后端 Lambda 函数会对传入请求数据进行解析，以确定要返回的响应。要使 API Gateway 将 Lambda 输出作为 API 响应传递给客户端，Lambda 函数必须返回此格式的结果。

因为 API Gateway 不过多地在 Lambda 代理集成的客户端和后端 Lambda 函数之间进行干预，所以客户端和集成的 Lambda 函数可以适应彼此的变化而不破坏 API 的现有集成设置。要实现这一点，客户端必须遵循后端 Lambda 函数所制定的应用程序协议。

您可以为任何 API 方法设置 Lambda 代理集成。但是，当为涉及通用代理资源的 API 方法配置 Lambda 代理集成时，该集成更加有效。通用代理资源可由 `{proxy+}` 的特殊模板化路径变量和/或“捕获全部”ANY 方法占位符表示。客户端可以在传入请求中将输入作为请求参数或适用负载传递给后端 Lambda 函数。请求参数包括标头、URL 路径变量、查询字符串参数和适用负载。集成的 Lambda 函数会在处理请求之前验证所有输入源，如果缺少任何必需的输入，则会使用有意义的错误消息向客户端做出响应。

在调用与通用 HTTP 方法 ANY 和通用资源 `{proxy+}` 集成的 API 方法时，客户端会使用特定 HTTP 方法而非 ANY 来提交请求。客户端还指定特定 URL 路径而非 `{proxy+}`，并包含任何所需的标头、查询字符串参数或适用负载。

以下列表总结了使用 Lambda 代理集成的不同 API 方法的运行时行为：

- `ANY /{proxy+}`：客户端必须选择特定 HTTP 方法，必须设置特定资源路径层次结构，并可以设置任何标头、查询字符串参数和适用负载以将数据作为输入传递给集成的 Lambda 函数。
- `ANY /res`：客户端必须选择特定 HTTP 方法，并可以设置任何标头、查询字符串参数和适用负载以将数据作为输入传递给集成的 Lambda 函数。
- `GET|POST|PUT|... /{proxy+}`：客户端可以设置特定资源路径层次结构、任何标头、查询字符串参数和适用负载以将数据作为输入传递给集成的 Lambda 函数。
- `GET|POST|PUT|... /res/{path}/...`：客户端必须选择特定路径分段（针对 `{path}` 变量），并可以设置任何请求标头、查询字符串参数和适用负载以将输入数据传递给集成的 Lambda 函数。

- `GET|POST|PUT|... /res`：客户端可以选择任何请求标头、查询字符串参数和适用负载以将输入数据传递给集成的 Lambda 函数。

代理资源 `{proxy+}` 和自定义资源 `{custom}` 都表示为模板化路径变量。但是 `{proxy+}` 可以指路径层次结构中的任何资源，而 `{custom}` 只能指特定路径分段。例如，一个杂货店可以按品类名称、农产品类别和产品类型整理其上线产品库存。然后，该杂货店的网站可以通过自定义资源的以下模板化路径变量来表示可用产品：`/{department}/{produce-category}/{product-type}`。例如，通过 `/produce/fruit/apple` 来表示苹果，通过 `/produce/vegetables/carrot` 来表示胡萝卜。它还可以使用 `/{proxy+}` 来表示客户在其在线商店中购物时可以搜索的任何品类、任何农产品类别或任何产品类型。例如，`/{proxy+}` 可以指以下任一项：

- `/produce`
- `/produce/fruit`
- `/produce/vegetables/carrot`

要让客户搜索任何可用产品、其农产品类别和关联的商店品类，您可以公开具有只读权限的单个方法 `GET /{proxy+}`。同样地，要允许主管更新 `produce` 品类的库存，您可以设置另一个具有读/写权限的单个方法 `PUT /produce/{proxy+}`。要允许出纳员更新蔬菜的流水式总计，您可以设置一个具有读/写权限的 `POST /produce/vegetables/{proxy+}` 方法。要让商店经理对任何可用产品执行任何可能的操作，在线商店开发人员可以公开具有读/写权限的 `ANY /{proxy+}` 方法。任何情况下，在运行时，客户或员工都必须选择所选品类中给定类型的特定产品、所选品类中的特定农产品类别或特定品类。

有关设置 API Gateway 代理集成的更多信息，请参阅 设置具有代理资源的代理集成。

代理集成要求客户端更详细地了解后端要求。因此，要确保最佳的应用程序性能和用户体验，后端开发人员必须清楚地向客户端开发人员表明后端的要求，并在不符合要求时提供可靠的错误反馈机制。

支持多值标头和查询字符串参数

API Gateway 支持多个具有相同名称的标头和查询字符串参数。多值标头以及单值标头和参数可以组合使用相同的请求和响应。有关更多信息，请参阅用于代理集成的 Lambda 函数的输入格式 和用于代理集成的 Lambda 函数的输出格式。

设置具有 Lambda 代理集成的代理资源

要设置具有 Lambda 代理集成类型的代理资源，请创建一个具有“贪婪”路径参数（例如，`/parent/{proxy+}`）的 API 资源，并将该资源与 `arn:aws:lambda:us-`

`west-2:123456789012:function:SimpleLambda4ProxyResource` 方法上的 Lambda 函数后端（例如，ANY）集成。“贪婪”路径参数必须位于 API 资源路径的末尾。与处理非代理资源一样，您可以通过使用 API Gateway 控制台、导入 OpenAPI 定义文件或直接调用 API Gateway REST API 来设置代理资源。

以下 OpenAPI API 定义文件显示了一个 API 的示例，该 API 具有与名为 `SimpleLambda4ProxyResource` 的 Lambda 函数集成的代理资源。

OpenAPI 3.0

```
{
   "openapi": "3.0.0",
   "info": {
      "version": "2016-09-12T17:50:37Z",
      "title": "ProxyIntegrationWithLambda"
   },
   "paths": {
      "/{proxy+}": {
         "x-amazon-apigateway-any-method": {
            "parameters": [
               {
                  "name": "proxy",
                  "in": "path",
                  "required": true,
                  "schema": {
                     "type": "string"
                  }
               }
            ],
            "responses": {},
            "x-amazon-apigateway-integration": {
               "responses": {
                  "default": {
                     "statusCode": "200"
                  }
               },
               "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:123456789012:function:SimpleLambda4ProxyResource/
invocations",
               "passthroughBehavior": "when_no_match",
               "httpMethod": "POST",
               "cacheNamespace": "roq9wj",
               "cacheKeyParameters": [
```

```
                  "method.request.path.proxy"
               ],
               "type": "aws_proxy"
            }
         }
      }
   },
   "servers": [
      {
         "url": "https://gy415nuibc.execute-api.us-east-1.amazonaws.com/{basePath}",
         "variables": {
            "basePath": {
              "default": "/testStage"
            }
         }
      }
   ]
}
```

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-09-12T17:50:37Z",
    "title": "ProxyIntegrationWithLambda"
  },
  "host": "gy415nuibc.execute-api.us-east-1.amazonaws.com",
  "basePath": "/testStage",
  "schemes": [
    "https"
  ],
  "paths": {
    "/{proxy+}": {
      "x-amazon-apigateway-any-method": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "proxy",
            "in": "path",
            "required": true,
```

```
            "type": "string"
          }
        ],
        "responses": {},
        "x-amazon-apigateway-integration": {
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:SimpleLambda4ProxyResource/
invocations",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST",
          "cacheNamespace": "roq9wj",
          "cacheKeyParameters": [
            "method.request.path.proxy"
          ],
          "type": "aws_proxy"
        }
      }
    }
  }
}
```

在 Lambda 代理集成中，在运行时，API Gateway 将传入请求映射到 Lambda 函数的输入 `event` 参数中。该输入包含请求方法、路径、标头、任何字符串参数、任何负载、关联的上下文以及定义的任何阶段变量。输入格式在用于代理集成的 Lambda 函数的输入格式中说明。为了让 API Gateway 将 Lambda 输出成功映射到 HTTP 响应，Lambda 函数必须按照用于代理集成的 Lambda 函数的输出格式中说明的格式输出结果。

在通过 ANY 方法实现代理资源的 Lambda 代理集成的过程中，单个后端 Lambda 函数充当通过代理资源的所有请求的事件处理程序。例如，要记录流量模式，您可以让移动设备提交一个代理资源的 URL 路径中附带 `/state/city/street/house` 的请求，从而发送包含州、城市、街道和建筑信息的位置信息。然后，后端 Lambda 函数可以分析 URL 路径并将位置元组插入 DynamoDB 表。

使用 AWS CLI 设置 Lambda 代理集成

在本部分中，我们将演示如何使用 AWS CLI 设置 API 与 Lambda 代理集成。

Note

有关使用 API Gateway 控制台配置与 Lambda 代理集成的代理资源的详细说明，请参阅 教程：使用 Lambda 代理集成构建 Hello World REST API。

例如，我们使用以下示例 Lambda 函数作为 API 的后端：

```
export const handler = function(event, context, callback) {
    console.log('Received event:', JSON.stringify(event, null, 2));
    var res ={
        "statusCode": 200,
        "headers": {
            "Content-Type": "*/*"
        }
    };
    var greeter = 'World';
    if (event.greeter && event.greeter!=="") {
        greeter =  event.greeter;
    } else if (event.body && event.body !== "") {
        var body = JSON.parse(event.body);
        if (body.greeter && body.greeter !== "") {
            greeter = body.greeter;
        }
    } else if (event.queryStringParameters && event.queryStringParameters.greeter &&
 event.queryStringParameters.greeter !== "") {
        greeter = event.queryStringParameters.greeter;
    } else if (event.multiValueHeaders && event.multiValueHeaders.greeter &&
 event.multiValueHeaders.greeter != "") {
        greeter = event.multiValueHeaders.greeter.join(" and ");
    } else if (event.headers && event.headers.greeter && event.headers.greeter != "") {
        greeter = event.headers.greeter;
    }

    res.body = "Hello, " + greeter + "!";
    callback(null, res);
};
```

将这一点与 Lambda 自定义集成设置对比，对此 Lambda 函数的输入可能出现在请求参数和正文中。您可以有更多的自主度来允许客户端传递相同的输入数据。在这里，客户端可以将欢迎者的姓名作为查询字符串参数、标题或正文属性传递。该函数还支持 Lambda 自定义集成。API 设置更简单。您完全无需配置方法响应或集成响应。

使用 AWS CLI 设置 Lambda 代理集成

1. 调用 `create-rest-api` 命令以创建 API：

   ```
   aws apigateway create-rest-api --name 'HelloWorld (AWS CLI)' --region us-west-2
   ```

   请注意在响应中生成的 API 的 `id` 值 (`te6si5ach7`)。

   ```
   {
       "name": "HelloWorldProxy (AWS CLI)",
       "id": "te6si5ach7",
       "createdDate": 1508461860
   }
   ```

   您在此部分中需要 API `id`。

2. 调用 `get-resources` 命令以获取根资源 `id`：

   ```
   aws apigateway get-resources --rest-api-id te6si5ach7 --region us-west-2
   ```

   成功的响应如下所示：

   ```
   {
       "items": [
           {
               "path": "/",
               "id": "krznpq9xpg"
           }
       ]
   }
   ```

   记录根资源 `id` 值 (`krznpq9xpg`)。您在下一个步骤中和以后需要用到它。

3. 调用 `create-resource` 以创建 API Gateway 资源 `/greeting`：

   ```
   aws apigateway create-resource --rest-api-id te6si5ach7 \
         --region us-west-2 \
         --parent-id krznpq9xpg \
         --path-part {proxy+}
   ```

   成功响应的形式与下方类似：

```
{
    "path": "/{proxy+}",
    "pathPart": "{proxy+}",
    "id": "2jf6xt",
    "parentId": "krznpq9xpg"
}
```

记录生成的 `{proxy+}` 资源的 `id` 值 (`2jf6xt`)。在下一个步骤中，您需要它在 `/{proxy+}` 资源上创建方法。

4. 调用 `put-method` 以创建 `ANY` 方法的 `ANY /{proxy+}` 请求：

   ```
   aws apigateway put-method --rest-api-id te6si5ach7 \
          --region us-west-2 \
          --resource-id 2jf6xt \
          --http-method ANY \
          --authorization-type "NONE"
   ```

   成功响应的形式与下方类似：

   ```
   {
       "apiKeyRequired": false,
       "httpMethod": "ANY",
       "authorizationType": "NONE"
   }
   ```

   此 API 方法允许客户端从后端的 Lambda 函数接收或发送问候语。

5. 调用 `put-integration` 以使用名为 `ANY /{proxy+}` 的 Lambda 函数设置 `HelloWorld` 方法的集成。在提供 `greeter` 参数时，此函数使用消息 `"Hello, {name}!"` 响应请求，在未设置查询字符串参数时使用 `"Hello, World!"` 响应。

   ```
   aws apigateway put-integration \
           --region us-west-2 \
           --rest-api-id te6si5ach7 \
           --resource-id 2jf6xt \
           --http-method ANY \
           --type AWS_PROXY \
           --integration-http-method POST \
           --uri arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
   arn:aws:lambda:us-west-2:123456789012:function:HelloWorld/invocations \
   ```

```
        --credentials arn:aws:iam::123456789012:role/apigAwsProxyRole
```

⚠ Important

对于 Lambda 集成，根据函数调用的 Lambda 服务操作规范，您必须为集成请求使用 HTTP 方法 POST。apigAwsProxyRole 的 IAM 角色必须具有策略，允许 apigateway 服务调用 Lambda 函数。有关 IAM 权限的更多信息，请参阅the section called "用于调用 API 的 API Gateway 权限模型"。

成功输出的形式与下方类似：

```
{
    "passthroughBehavior": "WHEN_NO_MATCH",
    "cacheKeyParameters": [],
    "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:1234567890:function:HelloWorld/invocations",
    "httpMethod": "POST",
    "cacheNamespace": "vvom7n",
    "credentials": "arn:aws:iam::1234567890:role/apigAwsProxyRole",
    "type": "AWS_PROXY"
}
```

不同于为 `credentials` 提供 IAM 角色，您可以调用 add-permission 命令以添加基于资源的权限。这是 API Gateway 控制台的功能。

6. 调用 `create-deployment` 以部署 API 到 `test` 阶段：

```
aws apigateway create-deployment --rest-api-id te6si5ach7 --stage-name test --
region us-west-2
```

7. 在终端中使用以下 cURL 命令测试 API。

使用查询字符串参数 `?greeter=jane` 调用 API：

```
curl -X GET 'https://te6si5ach7.execute-api.us-west-2.amazonaws.com/test/greeting?
greeter=jane'
```

使用 `greeter:jane` 的标头参数调用 API：

```
curl -X GET https://te6si5ach7.execute-api.us-west-2.amazonaws.com/test/hi \
  -H 'content-type: application/json' \
  -H 'greeter: jane'
```

使用 `{"greeter":"jane"}` 的正文调用 API：

```
curl -X POST https://te6si5ach7.execute-api.us-west-2.amazonaws.com/test/hi \
  -H 'content-type: application/json' \
  -d '{ "greeter": "jane" }'
```

在所有情况下，输出为具有以下响应正文的 200 响应：

```
Hello, jane!
```

用于代理集成的 Lambda 函数的输入格式

使用 Lambda 代理集成，API Gateway 可以将整个客户端请求映射到后端 Lambda 函数的输入 `event` 参数：以下示例显示了 API Gateway 发送到 Lambda 代理集成的事件的结构。

```
{
  "resource": "/my/path",
  "path": "/my/path",
  "httpMethod": "GET",
  "headers": {
    "header1": "value1",
    "header2": "value1,value2"
  },
  "multiValueHeaders": {
    "header1": [
      "value1"
    ],
    "header2": [
      "value1",
      "value2"
    ]
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
```

```
  "multiValueQueryStringParameters": {
    "parameter1": [
      "value1",
      "value2"
    ],
    "parameter2": [
      "value"
    ]
  },
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "id",
    "authorizer": {
      "claims": null,
      "scopes": null
    },
    "domainName": "id.execute-api.us-east-1.amazonaws.com",
    "domainPrefix": "id",
    "extendedRequestId": "request-id",
    "httpMethod": "GET",
    "identity": {
      "accessKey": null,
      "accountId": null,
      "caller": null,
      "cognitoAuthenticationProvider": null,
      "cognitoAuthenticationType": null,
      "cognitoIdentityId": null,
      "cognitoIdentityPoolId": null,
      "principalOrgId": null,
      "sourceIp": "IP",
      "user": null,
      "userAgent": "user-agent",
      "userArn": null,
      "clientCert": {
        "clientCertPem": "CERT_CONTENT",
        "subjectDN": "www.example.com",
        "issuerDN": "Example issuer",
        "serialNumber": "a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1",
        "validity": {
          "notBefore": "May 28 12:30:02 2019 GMT",
          "notAfter": "Aug  5 09:36:04 2021 GMT"
        }
      }
    },
```

```
    "path": "/my/path",
    "protocol": "HTTP/1.1",
    "requestId": "id=",
    "requestTime": "04/Mar/2020:19:15:17 +0000",
    "requestTimeEpoch": 1583349317135,
    "resourceId": null,
    "resourcePath": "/my/path",
    "stage": "$default"
  },
  "pathParameters": null,
  "stageVariables": null,
  "body": "Hello from Lambda!",
  "isBase64Encoded": false
}
```

> **Note**
>
> 在输入中：
>
> - `headers` 密钥只能包含单值标头。
> - `multiValueHeaders` 密钥可以包含多值标头以及单值标头。
> - 如果您指定 `headers` 和 `multiValueHeaders` 的值，API Gateway 会将它们合并为一个列表。如果两者都指定了相同的键/值对，则合并列表中只会出现 `multiValueHeaders` 的值。

在后端 Lambda 函数的输入中，`requestContext` 对象是键/值对的映射。在每对中，键为 $context 变量属性的名称，值为该属性的值。API Gateway 可能会向映射中添加新键。

根据启用的功能，不同 API 的 `requestContext` 映射可能有所不同。例如，在上述示例中，未指定任何授权类型，因此不存在任何 `$context.authorizer.*` 或 `$context.identity.*` 属性。当指定某个授权类型时，这会导致 API Gateway 将授权用户信息传递给 `requestContext.identity` 对象中的集成端点，如下所示：

- 当授权类型为 `AWS_IAM` 时，授权用户信息包括 `$context.identity.*` 属性。
- 当授权类型为 `COGNITO_USER_POOLS`（Amazon Cognito 授权方）时，授权用户信息包括 `$context.identity.cognito*` 和 `$context.authorizer.claims.*`。
- 当授权类型为 `CUSTOM`（Lambda 授权方）时，授权用户信息包括 `$context.authorizer.principalId` 及其他适用的 `$context.authorizer.*` 属性。

用于代理集成的 Lambda 函数的输出格式

在 Lambda 代理集成中，API Gateway 需要后端 Lambda 函数根据以下 JSON 格式返回输出：

```
{
    "isBase64Encoded": true|false,
    "statusCode": httpStatusCode,
    "headers": { "headerName": "headerValue", ... },
    "multiValueHeaders": { "headerName": ["headerValue", "headerValue2", ...], ... },
    "body": "..."
}
```

在输出中：

- 如果不返回任何额外的响应标头，则可以不指定 `headers` 和 `multiValueHeaders` 键。
- `headers` 密钥只能包含单值标头。
- `multiValueHeaders` 密钥可以包含多值标头以及单值标头。您可以使用 `multiValueHeaders` 密钥来指定所有额外的标头，包括任何单值标头。
- 如果您指定 `headers` 和 `multiValueHeaders` 的值，API Gateway 会将它们合并为一个列表。如果两者都指定了相同的键/值对，则合并列表中只会出现 `multiValueHeaders` 的值。

要为 Lambda 代理集成启用 CORS，您必须将 `Access-Control-Allow-Origin:`*`domain-name`* 添加到输出 `headers`。*`domain-name`* 可以为 `*`，表示任意域名。输出 `body` 作为方法响应负载封送到前端。如果 `body` 是二进制 blob，您可以通过将 `isBase64Encoded` 设置为 `true` 并将 `*/*` 配置为二进制媒体类型来将其编码为采用 Base64 编码的字符串。否则，您可以将其设置为 `false` 或不进行指定。

> **Note**
>
> 有关启用二进制文件支持的更多信息，请参阅使用 API Gateway 控制台启用二进制支持。有关示例 Lambda 函数，请参阅 从 Lambda 代理集成返回二进制媒体。

如果函数输出属于其他格式，则 API Gateway 将返回 `502 Bad Gateway` 错误响应。

要在 Node.js 的 Lambda 函数中返回响应，您可以使用如下所示的命令：

- 要返回成功的结果，请调用 `callback(null, {"statusCode": 200, "body": "results"})`。

- 要引发异常，请调用 `callback(new Error('internal server error'))`。
- 对于客户端错误（例如，如果缺少必需参数），可以调用 `callback(null, {"statusCode": 400, "body": "Missing parameters of ..."})` 以返回错误而不引发异常。

在 Node.js 的 Lambda `async` 函数中，等效的语法为：

- 要返回成功的结果，请调用 `return {"statusCode": 200, "body": "results"}`。
- 要引发异常，请调用 `throw new Error("internal server error")`。
- 对于客户端错误（例如，如果缺少必需参数），可以调用 `return {"statusCode": 400, "body": "Missing parameters of ..."}` 以返回错误而不引发异常。

在 API Gateway 中设置 Lambda 自定义集成

为显示如何设置 Lambda 自定义集成，我们创建了一个 API Gateway API 用于公开 `GET /greeting?greeter={name}` 方法来调用 Lambda 函数。为您的 API 使用以下 Lambda 函数示例之一。

使用以下 Lambda 函数示例之一：

Node.js

```
export const handler = function(event, context, callback) {
    var res ={
        "statusCode": 200,
        "headers": {
            "Content-Type": "*/*"
        }
    };
    if (event.greeter==null) {
        callback(new Error('Missing the required greeter parameter.'));
    } else if (event.greeter === "") {
        res.body = "Hello, World";
        callback(null, res);
    } else {
        res.body = "Hello, " + event.greeter +"!";
        callback(null, res);
    }
};
```

Python

```
import json


def lambda_handler(event, context):
    print(event)
    res = {
        "statusCode": 200,
        "headers": {
            "Content-Type": "*/*"
        }
    }

    if event['greeter'] == "":
        res['body'] = "Hello, World"
    elif (event['greeter']):
        res['body'] = "Hello, " + event['greeter'] + "!"
    else:
        raise Exception('Missing the required greeter parameter.')

    return res
```

在 `greeter` 参数值为非空字符串时，该函数使用消息 `"Hello, {name}!"` 进行响应。如果 `greeter` 值是空字符串，则返回消息 `"Hello, World!"`。如果传入请求中未设置 greeter 参数，则返回错误消息 `"Missing the required greeter parameter."`。我们将函数命名为 `HelloWorld`。

您可以在 Lambda 控制台中或使用 AWS CLI 创建它。在此部分中，我们使用以下 ARN 引用此函数：

```
arn:aws:lambda:us-east-1:123456789012:function:HelloWorld
```

在后端设置了 Lambda 函数之后，继续设置 API。

使用 AWS CLI 设置 Lambda 自定义集成

1. 调用 `create-rest-api` 命令以创建 API：

    ```
    aws apigateway create-rest-api --name 'HelloWorld (AWS CLI)' --region us-west-2
    ```

请注意在响应中生成的 API 的 `id` 值 (`te6si5ach7`)。

```
{
    "name": "HelloWorld (AWS CLI)",
    "id": "te6si5ach7",
    "createdDate": 1508461860
}
```

您在此部分中需要 API id。

2. 调用 `get-resources` 命令以获取根资源 `id`：

```
aws apigateway get-resources --rest-api-id te6si5ach7 --region us-west-2
```

成功的响应如下所示：

```
{
    "items": [
        {
            "path": "/",
            "id": "krznpq9xpg"
        }
    ]
}
```

记录根资源 `id` 值 (krznpq9xpg)。您在下一个步骤中和以后需要用到它。

3. 调用 `create-resource` 以创建 API Gateway 资源 `/greeting`：

```
aws apigateway create-resource --rest-api-id te6si5ach7 \
      --region us-west-2 \
      --parent-id krznpq9xpg \
      --path-part greeting
```

成功响应的形式与下方类似：

```
{
    "path": "/greeting",
    "pathPart": "greeting",
    "id": "2jf6xt",
    "parentId": "krznpq9xpg"
```

```
}
```

记录生成的 `greeting` 资源的 `id` 值 (`2jf6xt`)。在下一个步骤中，您需要它在 `/greeting` 资源上创建方法。

4. 调用 `put-method` 以创建 API 方法请求 `GET /greeting?greeter={name}`：

```
aws apigateway put-method --rest-api-id te6si5ach7 \
       --region us-west-2 \
       --resource-id 2jf6xt \
       --http-method GET \
       --authorization-type "NONE" \
       --request-parameters method.request.querystring.greeter=false
```

成功响应的形式与下方类似：

```
{
    "apiKeyRequired": false,
    "httpMethod": "GET",
    "authorizationType": "NONE",
    "requestParameters": {
        "method.request.querystring.greeter": false
    }
}
```

此 API 方法允许客户端从后端的 Lambda 函数接收问候语。此 `greeter` 参数是可选的，因为后端应处理匿名调用方或自识别调用方。

5. 调用 `put-method-response` 将 `200 OK` 响应设置到方法请求 `GET /greeting?greeter={name}`：

```
aws apigateway put-method-response \
        --region us-west-2 \
        --rest-api-id te6si5ach7 \
        --resource-id 2jf6xt \
        --http-method GET \
        --status-code 200
```

6. 调用 `put-integration` 以使用名为 `GET /greeting?greeter={name}` 的 Lambda 函数设置 `HelloWorld` 方法的集成。在提供 `greeter` 参数时，函数使用消息 `"Hello, {name}!"` 响应请求，在未设置查询字符串参数时使用 `"Hello, World!"` 响应。

```
aws apigateway put-integration \
        --region us-west-2 \
        --rest-api-id te6si5ach7 \
        --resource-id 2jf6xt \
        --http-method GET \
        --type AWS \
        --integration-http-method POST \
        --uri arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:HelloWorld/invocations \
        --request-templates '{"application/json":"{\"greeter\":
\"$input.params('greeter')\"}"}' \
        --credentials arn:aws:iam::123456789012:role/apigAwsProxyRole
```

此处提供的映射模板转换为 JSON 负载 `greeter` 属性的 `greeter` 查询字符串参数。这么做是必要的，因为对 Lambda 函数的输入必须在正文中表示。

⚠ Important

对于 Lambda 集成，根据函数调用的 Lambda 服务操作规范，您必须为集成请求使用 HTTP 方法 POST。`uri` 参数是函数调用操作的 ARN。

成功输出类似于以下内容：

```
{
    "passthroughBehavior": "WHEN_NO_MATCH",
    "cacheKeyParameters": [],
    "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:HelloWorld/invocations",
    "httpMethod": "POST",
    "requestTemplates": {
        "application/json": "{\"greeter\":\"$input.params('greeter')\"}"
    },
    "cacheNamespace": "krznpq9xpg",
    "credentials": "arn:aws:iam::123456789012:role/apigAwsProxyRole",
    "type": "AWS"
}
```

`apigAwsProxyRole` 的 IAM 角色必须具有策略，允许 `apigateway` 服务调用 Lambda 函数。不同于为 `credentials` 提供 IAM 角色，您可以调用 add-permission 命令以添加基于资源的权限。这就是 API Gateway 控制台添加这些权限的方式。

7. 调用 `put-integration-response` 以设置集成响应，用于将 Lambda 函数输出传递到客户端作为 `200 OK` 方法响应。

```
aws apigateway put-integration-response \
        --region us-west-2 \
        --rest-api-id te6si5ach7 \
        --resource-id 2jf6xt \
        --http-method GET \
        --status-code 200 \
        --selection-pattern ""
```

   将 selection-pattern 设置为空字符串时，`200 OK` 响应是默认值。

   成功的响应内容应类似如下所示：

```
 {
    "selectionPattern": "",
    "statusCode": "200"
}
```

8. 调用 `create-deployment` 以部署 API 到 `test` 阶段：

```
aws apigateway create-deployment --rest-api-id te6si5ach7 --stage-name test --
region us-west-2
```

9. 在终端中使用以下 cURL 命令测试 API：

```
curl -X GET 'https://te6si5ach7.execute-api.us-west-2.amazonaws.com/test/greeting?
greeter=me' \
  -H 'authorization: AWS4-HMAC-SHA256 Credential={access_key}/20171020/us-
west-2/execute-api/aws4_request, SignedHeaders=content-type;host;x-amz-date,
 Signature=f327...5751'
```

设置后端 Lambda 函数的异步调用

在 Lambda 非代理（自定义）集成中，默认情况下后端 Lambda 函数是同步调用的。这是大多数 REST API 操作的预期行为。但是，某些应用程序通常需要由某个单独的后端组件进行异步执行（如分批操作或长延迟操作）才能运行。在这种情况下，后端 Lambda 函数将进行异步调用，而且前端 REST API 方法不会返回结果。

您可以通过将 `'Event'` 指定为 Lambda 调用类型，为要异步调用的 Lambda 非代理集成配置 Lambda 函数。按如下所示完成此操作：

在 API Gateway 控制台中配置 Lambda 异步调用

要使所有调用均为异步，请执行以下操作：

- 在集成请求中，添加使用静态值 `'Event'` 的 `X-Amz-Invocation-Type` 标头。

要让客户端决定调用为异步还是同步，请执行以下操作：

1. 在方法请求中，添加 `InvocationType` 标头。
2. 在集成请求中，添加使用映射表达式 `method.request.header.InvocationType` 的 `X-Amz-Invocation-Type` 标头。
3. 在 API 请求中，对于异步调用，客户端可以包含 `InvocationType: Event` 标头，对于同步调用则可以包含 `InvocationType: RequestResponse`。

使用 OpenAPI 配置 Lambda 异步调用

要使所有调用均为异步，请执行以下操作：

- 将 `X-Amz-Invocation-Type` 标头添加到 x-amazon-apigateway-integration 部分。

```
"x-amazon-apigateway-integration" : {
          "type" : "aws",
          "httpMethod" : "POST",
          "uri" : "arn:aws:apigateway:us-east-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-2:123456789012:function:my-function/invocations",
          "responses" : {
            "default" : {
              "statusCode" : "200"
            }
          },
```

```
          "requestParameters" : {
            "integration.request.header.X-Amz-Invocation-Type" : "'Event'"
          },
          "passthroughBehavior" : "when_no_match",
          "contentHandling" : "CONVERT_TO_TEXT"
        }
```

要让客户端决定调用为异步还是同步，请执行以下操作：

1. 在任何 OpenAPI 路径项对象上添加以下标头。

```
"parameters" : [ {
"name" : "InvocationType",
"in" : "header",
"schema" : {
  "type" : "string"
}
} ]
```

2. 将 `X-Amz-Invocation-Type` 标头添加到 x-amazon-apigateway-integration 部分。

```
"x-amazon-apigateway-integration" : {
          "type" : "aws",
          "httpMethod" : "POST",
          "uri" : "arn:aws:apigateway:us-east-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-2:123456789012:function:my-function/invocations",
          "responses" : {
            "default" : {
              "statusCode" : "200"
            }
          },
          "requestParameters" : {
            "integration.request.header.X-Amz-Invocation-Type" :
 "method.request.header.InvocationType"
          },
          "passthroughBehavior" : "when_no_match",
          "contentHandling" : "CONVERT_TO_TEXT"
        }
```

3. 在 API 请求中，对于异步调用，客户端可以包含 `InvocationType: Event` 标头，对于同步调用则可以包含 `InvocationType: RequestResponse`。

使用 AWS CloudFormation 配置 Lambda 异步调用

以下 AWS CloudFormation 模板显示如何配置 `AWS::ApiGateway::Method` 来进行异步调用。

要使所有调用均为异步，请执行以下操作：

```
AsyncMethodGet:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref AsyncResource
      HttpMethod: GET
      ApiKeyRequired: false
      AuthorizationType: NONE
      Integration:
        Type: AWS
        RequestParameters:
          integration.request.header.X-Amz-Invocation-Type: "'Event'"
        IntegrationResponses:
            - StatusCode: '200'
        IntegrationHttpMethod: POST
        Uri: !Sub arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/
${myfunction.Arn}$/invocations
      MethodResponses:
        - StatusCode: '200'
```

要让客户端决定调用为异步还是同步，请执行以下操作：

```
AsyncMethodGet:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref AsyncResource
      HttpMethod: GET
      ApiKeyRequired: false
      AuthorizationType: NONE
      RequestParameters:
        method.request.header.InvocationType: false
      Integration:
        Type: AWS
        RequestParameters:
```

```
          integration.request.header.X-Amz-Invocation-Type:
 method.request.header.InvocationType
        IntegrationResponses:
            - StatusCode: '200'
        IntegrationHttpMethod: POST
        Uri: !Sub arn:aws:apigateway:${AWS::Region}:lambda:path/2015-03-31/functions/
${myfunction.Arn}$/invocations
      MethodResponses:
        - StatusCode: '200'
```

在 API 请求中，对于异步调用，客户端可以包含 `InvocationType: Event` 标头，对于同步调用则可以包含 `InvocationType: RequestResponse`。

处理 API Gateway 中的 Lambda 错误

对于 Lambda 自定义集成，您必须将在集成响应中由 Lambda 返回的错误映射到客户端的标准 HTTP 错误响应。否则，Lambda 错误会默认返回为 `200 OK` 响应，并且结果对于您的 API 用户而言并不直观。

Lambda 可以返回两种类型的错误：标准错误和自定义错误。在您的 API 中，您必须以不同方式处理这些错误。

对于 Lambda 代理集成，Lambda 需要返回以下格式的输出：

```
{
  "isBase64Encoded" : "boolean",
  "statusCode": "number",
  "headers": { ... },
  "body": "JSON string"
}
```

在此输出中，`statusCode` 通常为 `4XX` (对于客户端错误) 和 `5XX` (对于服务器错误)。API Gateway 可根据指定的 `statusCode`，通过将 Lambda 错误到 HTTP 错误响应来处理这些错误。要使 API Gateway 将错误类型（例如 `InvalidParameterException`）作为响应的一部分传递给客户端，Lambda 函数必须在 `headers` 属性中包括标头（例如 `"X-Amzn-ErrorType":"InvalidParameterException"`）。

主题

- 处理 API Gateway 中的标准 Lambda 错误
- 处理 API Gateway 中的自定义 Lambda 错误

处理 API Gateway 中的标准 Lambda 错误

标准的 AWS Lambda 错误具有以下格式：

```
{
  "errorMessage": "<replaceable>string</replaceable>",
  "errorType": "<replaceable>string</replaceable>",
  "stackTrace": [
    "<replaceable>string</replaceable>",
    ...
  ]
}
```

其中，`errorMessage` 是错误的字符串表达式。`errorType` 是取决于语言的错误或异常类型。`stackTrace` 是一个字符串表达式列表，显示导致发生错误的堆栈跟踪。

例如，请考虑以下 JavaScript (Node.js) Lambda 函数。

```
export const handler = function(event, context, callback) {
    callback(new Error("Malformed input ..."));
};
```

此函数返回以下标准 Lambda 错误，其中包含 `Malformed input ...` 作为错误消息：

```
{
  "errorMessage": "Malformed input ...",
  "errorType": "Error",
  "stackTrace": [
    "export const handler (/var/task/index.js:3:14)"
  ]
}
```

同样，请考虑以下 Python Lambda 函数，该函数会引发具有相同 `Malformed input ...` 错误消息的 `Exception`。

```
def lambda_handler(event, context):
    raise Exception('Malformed input ...')
```

此函数会返回以下标准 Lambda 错误：

```
{
```

```
  "stackTrace": [
    [
      "/var/task/lambda_function.py",
      3,
      "lambda_handler",
      "raise Exception('Malformed input ...')"
    ]
  ],
  "errorType": "Exception",
  "errorMessage": "Malformed input ..."
}
```

请注意，`errorType` 和 `stackTrace` 属性值因语言而异。标准错误也适用于作为 `Error` 对象的扩展或 `Exception` 类的子类的任何错误对象。

要将标准 Lambda 错误映射到方法响应，您必须首先为给定的 Lambda 错误确定 HTTP 状态代码。然后，针对与给定 HTTP 状态代码相关联的 IntegrationResponse 的 `selectionPattern` 属性设置正则表达式模式。在 API Gateway 控制台的集成响应部分，此 `selectionPattern` 在每个集成响应下方表示为 Lambda 错误正则表达式。

> **Note**
>
> API Gateway 使用 Java 模式的正则表达式来响应映射。有关更多信息，请参阅 Oracle 文档中的模式。

例如，要设置新的 `selectionPattern` 表达式，请使用 AWS CLI 调用以下 put-integration-response 命令：

```
aws apigateway put-integration-response --rest-api-id z0vprf0mdh --resource-id x3o5ih
 --http-method GET --status-code 400 --selection-pattern "Malformed.*" --region us-
west-2
```

确保您还针对 方法响应 设置了相应的错误代码 (`400`)。否则，API Gateway 会在运行时引发一个关于配置无效的错误响应。

> **Note**
>
> 在运行时，API Gateway 将 Lambda 错误的 `errorMessage` 与 `selectionPattern` 属性的正则表达式模式相匹配。当存在匹配时，API Gateway 将返回 Lambda 错误作为相应 HTTP 状

态码的 HTTP 响应。如果没有匹配，API Gateway 将返回错误作为默认响应，或者引发配置无效异常（如果未配置默认响应）。
针对给定响应数量，将 `selectionPattern` 值设置为 `.*` 会将此响应重置为默认响应。这是因为此类选择模式将匹配所有错误消息，包括 null (即，任何未指定的错误消息)。生成的映射会覆盖默认映射。

要使用 `selectionPattern` 更新现有 AWS CLI 值，请调用 update-integration-response 操作，以将 `/selectionPattern` 路径值替换为 `Malformed*` 模式的指定正则表达式。

要使用 API Gateway 控制台设置 `selectionPattern` 表达式，请在设置或更新指定 HTTP 状态代码的集成响应时，在 Lambda 错误正则表达式文本框中输入表达式。

处理 API Gateway 中的自定义 Lambda 错误

AWS Lambda 允许您返回自定义错误对象作为 JSON 字符串，而不是返回上一部分所述的标准错误。该错误可以是任何有效的 JSON 对象。例如，下面的 JavaScript (Node.js) Lambda 函数会返回自定义错误：

```
export const handler = (event, context, callback) => {
    ...
    // Error caught here:
    var myErrorObj = {
        errorType : "InternalServerError",
        httpStatus : 500,
        requestId : context.awsRequestId,
        trace : {
            "function": "abc()",
            "line": 123,
            "file": "abc.js"
        }
    }
    callback(JSON.stringify(myErrorObj));
};
```

您必须先将 `myErrorObj` 对象转换为 JSON 字符串，然后才能调用 `callback` 以退出函数。否则，`myErrorObj` 会返回为 `"[object Object]"` 字符串。当您的 API 方法与上述 Lambda 函数集成时，API Gateway 会接收具有以下负载的集成响应：

```
{
```

```
    "errorMessage": "{\"errorType\":\"InternalServerError\",\"httpStatus\":500,
\"requestId\":\"e5849002-39a0-11e7-a419-5bb5807c9fb2\",\"trace\":{\"function\":
\"abc()\",\"line\":123,\"file\":\"abc.js\"}}"
}
```

与任何集成响应一样，您可以将此错误响应按原样传递至方法响应。或者，您可以使用映射模板将负载转换为另一种格式。例如，针对 500 状态代码的方法响应考虑以下正文映射模板：

```
{
    errorMessage: $input.path('$.errorMessage');
}
```

此模板会将包含自定义错误 JSON 字符串的集成响应正文转换为以下方法响应正文。此方法响应正文包含自定义错误 JSON 对象：

```
{
    "errorMessage" : {
        errorType : "InternalServerError",
        httpStatus : 500,
        requestId : context.awsRequestId,
        trace : {
            "function": "abc()",
            "line": 123,
            "file": "abc.js"
        }
    }
};
```

根据您的 API 要求，您可能需要传递部分或全部自定义错误属性作为方法响应标头参数。为实现这一点，您可以将来自集成响应正文的自定义错误映射应用到方法响应标头。

例如，下面的 OpenAPI 扩展定义了 `errorMessage.errorType`、`errorMessage.httpStatus`、`errorMessage.trace.function` 和 `errorMessage.trace` 属性分别到 `error_type`、`error_status`、`error_trace_function` 和 `error_trace` 标头的映射。

```
"x-amazon-apigateway-integration": {
    "responses": {
        "default": {
          "statusCode": "200",
          "responseParameters": {
```

```
            "method.response.header.error_trace_function":
 "integration.response.body.errorMessage.trace.function",
            "method.response.header.error_status":
 "integration.response.body.errorMessage.httpStatus",
            "method.response.header.error_type":
 "integration.response.body.errorMessage.errorType",
            "method.response.header.error_trace":
 "integration.response.body.errorMessage.trace"
          },
          ...
        }
    }
}
```

在运行时，API Gateway 会在执行标头映射时反序列化 `integration.response.body` 参数。但是，这种反序列化仅适用于 Lambda 自定义错误响应的“正文到标头”映射，并不适用于使用 `$input.body` 的“正文到正文”映射。使用这些 custom-error-body-to-header 映射，客户端会收到以下标头作为方法响应的一部分，前提是方法请求中声明了 `error_status`、`error_trace`、`error_trace_function` 和 `error_type` 标头。

```
"error_status":"500",
"error_trace":"{\"function\":\"abc()\",\"line\":123,\"file\":\"abc.js\"}",
"error_trace_function":"abc()",
"error_type":"InternalServerError"
```

集成响应正文的 `errorMessage.trace` 属性是一个复杂的属性。它会作为 JSON 字符串映射到 `error_trace` 标头。

在 API Gateway 中设置 HTTP 集成

您可以使用 HTTP 代理集成或 HTTP 自定义集成将 API 方法与 HTTP 终端节点进行集成。

API Gateway 支持以下终端节点端口：80、443 和 1024-65535。

借助代理集成，进行设置非常简单。如果不考虑内容编码或缓存，则只需根据后端要求设置 HTTP 方法和 HTTP 终端节点 URI。

借助自定义集成，进行设置更为复杂。除了执行代理集成设置步骤，还需要指定如何将传入请求数据映射到集成请求以及如何将生成的集成响应数据映射到方法响应。

主题

- 在 API Gateway 中设置 HTTP 代理集成

- 在 API Gateway 中设置 HTTP 自定义集成

在 API Gateway 中设置 HTTP 代理集成

要设置具有 HTTP 代理集成类型的代理资源，请创建一个具有“贪婪”路径参数（例如，`/parent/{proxy+}`）的 API 资源，并将该资源与 `https://petstore-demo-endpoint.execute-api.com/petstore/{proxy}` 方法上的 HTTP 后端终端节点（例如，ANY）集成。“贪婪”路径参数必须位于资源路径的末尾。

与处理非代理资源一样，您可以通过 API Gateway 控制台、导入 OpenAPI 定义文件或直接调用 API Gateway REST API 来设置具有 HTTP 代理集成的代理资源。有关使用 API Gateway 控制台配置与 HTTP 集成的代理资源的详细说明，请参阅 教程：使用 HTTP 代理集成构建 REST API。

以下 OpenAPI 定义文件显示了一个 API 的示例，该 API 具有与 PetStore 网站集成的代理资源。

OpenAPI 3.0

```
{
   "openapi": "3.0.0",
   "info": {
      "version": "2016-09-12T23:19:28Z",
      "title": "PetStoreWithProxyResource"
   },
   "paths": {
      "/{proxy+}": {
         "x-amazon-apigateway-any-method": {
            "parameters": [
               {
                  "name": "proxy",
                  "in": "path",
                  "required": true,
                  "schema": {
                     "type": "string"
                  }
               }
            ],
            "responses": {},
            "x-amazon-apigateway-integration": {
               "responses": {
                  "default": {
                     "statusCode": "200"
                  }
```

```
                },
                "requestParameters": {
                   "integration.request.path.proxy": "method.request.path.proxy"
                },
                "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/
{proxy}",
                "passthroughBehavior": "when_no_match",
                "httpMethod": "ANY",
                "cacheNamespace": "rbftud",
                "cacheKeyParameters": [
                   "method.request.path.proxy"
                ],
                "type": "http_proxy"
             }
          }
       }
    },
    "servers": [
       {
          "url": "https://4z9giyi2c1.execute-api.us-east-1.amazonaws.com/{basePath}",
          "variables": {
             "basePath": {
               "default": "/test"
             }
          }
       }
    ]
}
```

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-09-12T23:19:28Z",
    "title": "PetStoreWithProxyResource"
  },
  "host": "4z9giyi2c1.execute-api.us-east-1.amazonaws.com",
  "basePath": "/test",
  "schemes": [
    "https"
  ],
  "paths": {
```

```
    "/{proxy+}": {
      "x-amazon-apigateway-any-method": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "proxy",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {},
        "x-amazon-apigateway-integration": {
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.path.proxy": "method.request.path.proxy"
          },
          "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/{proxy}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "ANY",
          "cacheNamespace": "rbftud",
          "cacheKeyParameters": [
            "method.request.path.proxy"
          ],
          "type": "http_proxy"
        }
      }
    }
  }
}
```

在本示例中，代理资源的 `method.request.path.proxy` 路径参数上声明了一个缓存键。这是您使用 API Gateway 控制台创建 API 时的默认设置。API 的基本路径 (`/test`，与一个阶段对应) 映射到网站的 PetStore 页面 (`/petstore`)。单个集成请求可以使用 API 的“贪婪”路径变量和“捕获所有”的 ANY 方法镜像整个 PetStore 网站。建立镜像的过程如下所示。

- 将 **ANY** 设置为 **GET** 并将 **{proxy+}** 设置为 **pets**

  从前端发起的方法请求：

  ```
  GET https://4z9giyi2c1.execute-api.us-west-2.amazonaws.com/test/pets HTTP/1.1
  ```

  发送到后端的集成请求：

  ```
  GET http://petstore-demo-endpoint.execute-api.com/petstore/pets HTTP/1.1
  ```

  ANY 方法和代理资源的运行时实例都是有效的。此调用将返回一个 `200 OK` 响应以及包含从后端返回的第一批宠物的负载。

- 将 **ANY** 设置为 **GET** 并将 **{proxy+}** 设置为 **pets?type=dog**

  ```
  GET https://4z9giyi2c1.execute-api.us-west-2.amazonaws.com/test/pets?type=dog
   HTTP/1.1
  ```

  发送到后端的集成请求：

  ```
  GET http://petstore-demo-endpoint.execute-api.com/petstore/pets?type=dog HTTP/1.1
  ```

  ANY 方法和代理资源的运行时实例都是有效的。此调用将返回一个 `200 OK` 响应以及包含从后端返回的第一批指定宠物狗的负载。

- 将 **ANY** 设置为 **GET** 并将 **{proxy+}** 设置为 **pets/{petId}**

  从前端发起的方法请求：

  ```
  GET https://4z9giyi2c1.execute-api.us-west-2.amazonaws.com/test/pets/1 HTTP/1.1
  ```

  发送到后端的集成请求：

  ```
  GET http://petstore-demo-endpoint.execute-api.com/petstore/pets/1 HTTP/1.1
  ```

  ANY 方法和代理资源的运行时实例都是有效的。此调用将返回一个 `200 OK` 响应以及包含从后端返回的指定宠物的负载。

- 将 **ANY** 设置为 **POST** 并将 **{proxy+}** 设置为 **pets**

从前端发起的方法请求：

```
POST https://4z9giyi2c1.execute-api.us-west-2.amazonaws.com/test/pets HTTP/1.1
Content-Type: application/json
Content-Length: ...

{
  "type" : "dog",
  "price" : 1001.00
}
```

发送到后端的集成请求：

```
POST http://petstore-demo-endpoint.execute-api.com/petstore/pets HTTP/1.1
Content-Type: application/json
Content-Length: ...

{
  "type" : "dog",
  "price" : 1001.00
}
```

ANY 方法和代理资源的运行时实例都是有效的。此调用将返回一个 `200 OK` 响应以及包含从后端返回的新创建的宠物的负载。

- 将 **ANY** 设置为 **GET** 并将 **{proxy+}** 设置为 **pets/cat**

  从前端发起的方法请求：

```
GET https://4z9giyi2c1.execute-api.us-west-2.amazonaws.com/test/pets/cat
```

  发送到后端的集成请求：

```
GET http://petstore-demo-endpoint.execute-api.com/petstore/pets/cat
```

  代理资源路径的运行时实例与后端终端节点不对应，且生成的请求无效。因此，系统会返回 `400 Bad Request` 响应，并显示以下错误消息。

```
{
  "errors": [
```

```
    {
      "key": "Pet2.type",
      "message": "Missing required field"
    },
    {
      "key": "Pet2.price",
      "message": "Missing required field"
    }
  ]
}
```

- 将 **ANY** 设置为 **GET** 并将 **{proxy+}** 设置为 **null**

  从前端发起的方法请求：

  ```
  GET https://4z9giyi2c1.execute-api.us-west-2.amazonaws.com/test
  ```

  发送到后端的集成请求：

  ```
  GET http://petstore-demo-endpoint.execute-api.com/petstore/pets
  ```

  目标资源是代理资源的父资源，但未在该资源上的 API 中定义 ANY 方法的运行时实例。因此，此 GET 请求将返回一个 `403 Forbidden` 响应，而 API Gateway 返回 `Missing Authentication Token` 错误消息。如果 API 在父资源 (ANY) 上公开 GET 或 / 方法，则此调用将返回一个 `404 Not Found` 响应，同时从后端返回 `Cannot GET /petstore` 消息。

对于任何客户端请求，如果目标终端节点 URL 无效或 HTTP 命令动词有效但不受支持，则后端将返回 `404 Not Found` 响应。对于不受支持的 HTTP 方法，系统会返回 `403 Forbidden` 响应。

在 API Gateway 中设置 HTTP 自定义集成

使用 HTTP 自定义集成，您可以更好地控制在 API 方法和 API 集成之间传递哪些数据以及如何传递数据。您可以使用数据映射执行此操作。

作为方法请求设置的一部分，您需要设置 Method（方法）资源中的 requestParameters 属性。这会声明对于从客户端预配置的方法请求参数，哪些要先映射到集成请求参数或适用的正文属性，然后再分派到后端。然后，作为集成请求设置的一部分，在相应的 Integration（集成）资源上设置 requestParameters 属性，以指定参数到参数的映射。您还要设置 requestTemplates 属性，以指定映射模板，为每个支持的内容类型设置一个映射模板。映射模板将方法请求参数或正文映射到集成请求正文。

同样，作为方法响应设置的一部分，您可以在 MethodResponse 资源上设置 responseParameters 属性。这可声明哪些方法响应参数将分派到客户端，哪些将从已从后端返回的集成响应参数或某些适用的正文属性进行映射。然后，作为集成响应设置的一部分，在相应的 IntegrationResponse 资源上设置 responseParameters 属性，以指定参数到参数的映射。您还要设置 responseTemplates 映射，以指定映射模板，为每个支持的内容类型设置一个映射模板。映射模板将集成响应参数或集成响应正文属性映射到方法响应正文。

有关设置映射模板的更多信息，请参阅为 REST API 设置数据转换。

设置 API Gateway 私有集成

利用 API Gateway 私有集成，可轻松公开位于 Amazon VPC 之内的 HTTP/HTTPS 资源，供 VPC 之外的客户端访问。要将对私有 VPC 资源的访问扩展到 VPC 边界之外，您可以创建具有私有集成的 API。您可以使用 API Gateway 支持的任何授权方法来控制对 API 的访问。

要创建私有集成，您必须首先创建网络负载均衡器。您的网络负载均衡器必须具有可将请求路由到 VPC 中的资源的侦听器。要提高 API 的可用性，请确保网络负载均衡器将流量路由到 AWS 区域 中多个可用性区域中的资源。然后，您创建一个 VPC 链接，来连接 API 和网络负载均衡器。创建 VPC 链接后，您可以创建私有集成，从而通过 VPC 链接和网络负载均衡器将流量从 API 路由到 VPC 中的资源。

> Note
>
> 网络负载均衡器和 API 必须归同一个 AWS 账户所有。

借助 API Gateway 私有集成，您无需掌握详细的私有网络配置或技术特定设备的知识，即可启用对 VPC 中 HTTP/HTTPS 资源的访问。

主题

- 为 API Gateway 私有集成设置网络负载均衡器
- 授予创建 VPC 链接的权限
- 使用 API Gateway 控制台设置具有私有集成的 API Gateway API
- 使用 AWS CLI 设置具有私有集成的 API Gateway API
- 使用 OpenAPI 设置具有私有集成的 API
- 用于私有集成的 API Gateway 账户

为 API Gateway 私有集成设置网络负载均衡器

以下过程概述使用 Amazon EC2 控制台为 API Gateway 私有集成设置网络负载均衡器 (NLB) 的步骤，并提供对各个步骤的详细说明的参考。

对于其中包含有您的资源的每个 VPC，您只需要配置一个 NLB 和一个 VPCLink。NLB 支持每个 NLB 有多个侦听器和目标组。您可以在 NLB 上将每个服务配置成为一个特定的侦听器，然后使用一个 VPCLink 连接到 NLB。在 API Gateway 中创建私有集成时，您随后可以定义每个服务使用已为各个服务分配的特定端口。有关更多信息，请参阅 the section called “教程：使用私有集成构建 API”。

> **Note**
>
> Network Load Balancer 和 API 必须归同一个 AWS 账户所有。

使用 API Gateway 控制台为私有集成创建网络负载均衡器

1. 登录到 AWS Management Console，然后通过以下网址打开 Amazon EC2 控制台：https://console.aws.amazon.com/ec2/。
2. 在 Amazon EC2 实例上设置 Web 服务器。如需示例设置，请参阅在 Amazon Linux 2 上安装 LAMP Web 服务器。
3. 创建网络负载均衡器，将 EC2 实例注册到目标组，然后将目标组添加到网络负载均衡器的侦听器。有关详细信息，请参阅网络负载均衡器入门中的说明。
4. 创建网络负载均衡器之后，执行以下操作：

    a. 记录网络负载均衡器的 ARN。您需要它在 API Gateway 中创建 VPC 链接，用于将 API 与位于网络负载均衡器之后的 VPC 资源集成。

    b. 关闭 PrivateLink 的安全组评估。

    - 要使用控制台关闭对 PrivateLink 流量的安全组评估，可以选择安全选项卡，然后选择编辑。在安全设置下，清除对 PrivateLink 流量强制执行入站规则。
    - 要使用 AWS CLI 关闭对 PrivateLink 流量的安全组评估，请使用以下命令：

```
aws elbv2 set-security-groups --load-balancer-
arn arn:aws:elasticloadbalancing:us-east-2:111122223333:loadbalancer/net/
my-loadbalancer/abc12345 \
  --security-groups sg-123345a --enforce-security-group-inbound-rules-on-
private-link-traffic off
```

> **Note**
>
> 请勿向 API Gateway CIDR 添加任何依赖项，因为这些依赖项必然会未经通知而更改。

授予创建 VPC 链接的权限

对于您或者您账户中的用户，如果要创建和维护 VPC 链接，您或者用户必须有权创建、删除和查看 VPC 端点服务配置，更改 VPC 端点服务权限，以及检查负载均衡器。要授予此类权限，请使用以下步骤。

授予创建、更新和删除 VPC 链接的权限

1. 创建类似于以下的 IAM 策略：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "apigateway:POST",
                "apigateway:GET",
                "apigateway:PATCH",
                "apigateway:DELETE"
            ],
            "Resource": [
                "arn:aws:apigateway:us-east-1::/vpclinks",
                "arn:aws:apigateway:us-east-1::/vpclinks/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "elasticloadbalancing:DescribeLoadBalancers"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateVpcEndpointServiceConfiguration",
```

```
                "ec2:DeleteVpcEndpointServiceConfigurations",
                "ec2:DescribeVpcEndpointServiceConfigurations",
                "ec2:ModifyVpcEndpointServicePermissions"
            ],
            "Resource": "*"
        }
    ]
}
```

2. 创建或选择 IAM 角色并将前述策略附加到角色。
3. 将 IAM 角色分配给您或您账户中创建 VPC 链接的用户。

使用 API Gateway 控制台设置具有私有集成的 API Gateway API

有关使用 API Gateway 控制台设置采用私有集成的 API 的说明，请参阅教程：使用 API Gateway 私有集成构建 REST API。

使用 AWS CLI 设置具有私有集成的 API Gateway API

使用私有集成创建 API 之前，您必须已设置 VPC 资源，已创建网络负载均衡器并已使用 VPC 源作为目标进行配置。如果未满足要求，请按照为 API Gateway 私有集成设置网络负载均衡器安装 VPC 资源，创建网络负载均衡器，设置 VPC 资源作为网络负载均衡器的目标。

> **Note**
>
> Network Load Balancer 和 API 必须归同一个 AWS 账户所有。

为了能够创建和管理 `VpcLink`，您还必须具有适当的权限。有关更多信息，请参阅 授予创建 VPC 链接的权限。

> **Note**
>
> 您只需要具有在 API 中创建 `VpcLink` 的权限。您不需要具有使用 `VpcLink` 的权限。

创建网络负载均衡器之后，记录其 ARN。您需要它来为私有集成创建 VPC 链接。

使用 AWS CLI 设置具有私有集成的 API

1. 创建 `VpcLink`，定位到指定的网络负载均衡器。

```
aws apigateway create-vpc-link \
    --name my-test-vpc-link \
    --target-arns arn:aws:elasticloadbalancing:us-east-2:123456789012:loadbalancer/
net/my-vpclink-test-nlb/1234567890abcdef
```

此命令的输出确认收到请求并显示正在创建的 `VpcLink` 的 `PENDING` 状态。

```
{
    "status": "PENDING",
    "targetArns": [
        "arn:aws:elasticloadbalancing:us-east-2:123456789012:loadbalancer/net/my-
vpclink-test-nlb/1234567890abcdef"
    ],
    "id": "gim7c3",
    "name": "my-test-vpc-link"
}
```

API Gateway 需要 2-4 分钟才能完成创建 `VpcLink`。操作成功完成后，`status` 为 `AVAILABLE`。您可以通过调用以下 CLI 命令验证这一点：

```
aws apigateway get-vpc-link --vpc-link-id gim7c3
```

如果操作失败，您将收到 `FAILED` 状态，以及包含错误消息的 `statusMessage`。例如，如果您尝试使用已经与 VPC 端点关联的网络负载均衡器创建 `VpcLink`，在 `statusMessage` 属性上将获得以下内容：

```
"NLB is already associated with another VPC Endpoint Service"
```

在成功创建 `VpcLink` 之后，您可以创建 API 并通过 `VpcLink` 将其与 VPC 资源集成。

记录新创建 `VpcLink` 的 `id` 值 (之前输出中的 *`gim7c3`*)。设置私有集成时需要该值。

2. 通过创建 API Gateway [`RestApi`] 资源设置 API：

```
aws apigateway create-rest-api --name 'My VPC Link Test'
```

请记录返回结果中 `RestApi` 的 `id` 值。您需要该值才能在 API 上进一步执行操作。

为了方便说明，我们将在根资源 (/) 上创建仅具有 `GET` 方法的 API，并将方法与 `VpcLink` 集成。

3. 设置 GET / 方法。首先获取根资源 (/) 的标识符：

   ```
   aws apigateway get-resources --rest-api-id  abcdef123
   ```

   在输出中，记录 id 路径的 / 值。在本例中，我们假设它为 *skpp60rab7*。

   为 API 方法 GET / 设置方法请求：

   ```
   aws apigateway put-method \
          --rest-api-id  abcdef123 \
          --resource-id skpp60rab7 \
          --http-method GET \
          --authorization-type "NONE"
   ```

   如果您不将代理集成用于 VpcLink，则还必须至少设置一个 200 状态代码的方法响应。我们在这里使用代理集成。

4. 设置 HTTP_PROXY 类型的私有集成，并按以下所示调用 `put-integration` 命令：

   ```
   aws apigateway put-integration \
       --rest-api-id abcdef123 \
       --resource-id skpp60rab7 \
       --uri 'http://my-vpclink-test-nlb-1234567890abcdef.us-east-2.amazonaws.com' \
       --http-method GET \
       --type HTTP_PROXY \
       --integration-http-method GET \
       --connection-type VPC_LINK \
       --connection-id gim7c3
   ```

   对于私有集成，请将 `connection-type` 设置为 VPC_LINK，将 `connection-id` 设置为 `VpcLink` 的标识符或引用 `VpcLink` ID 的阶段变量。`uri` 参数不用于将请求路由到端点，而是用于设置 Host 标头和证书验证。

   该命令将返回以下输出：

   ```
   {
       "passthroughBehavior": "WHEN_NO_MATCH",
       "timeoutInMillis": 29000,
       "connectionId": "gim7c3",
       "uri": "http://my-vpclink-test-nlb-1234567890abcdef.us-east-2.amazonaws.com",
       "connectionType": "VPC_LINK",
   ```

```
    "httpMethod": "GET",
    "cacheNamespace": "skpp60rab7",
    "type": "HTTP_PROXY",
    "cacheKeyParameters": []
}
```

使用阶段变量，在创建集成时设置 `connectionId` 属性：

```
aws apigateway put-integration \
    --rest-api-id abcdef123 \
    --resource-id skpp60rab7 \
    --uri 'http://my-vpclink-test-nlb-1234567890abcdef.us-east-2.amazonaws.com' \
    --http-method GET \
    --type HTTP_PROXY \
    --integration-http-method GET \
    --connection-type VPC_LINK \
    --connection-id "\${stageVariables.vpcLinkId}"
```

请确保使用双引号括起阶段变量表达式 (`${stageVariables.vpcLinkId}`) 并转义 $ 字符。

或者，您可以更新集成以使用阶段变量重置 `connectionId` 值：

```
 aws apigateway update-integration \
    --rest-api-id abcdef123 \
    --resource-id skpp60rab7 \
    --http-method GET \
    --patch-operations '[{"op":"replace","path":"/
connectionId","value":"${stageVariables.vpcLinkId}"}]'
```

请确保使用字符串化的 JSON 列表作为 `patch-operations` 参数值。

您可以使用阶段变量，通过重置 `VpcLinks` 阶段变量值，将您的 API 与其他 VPC 或网络负载均衡器集成。

由于我们使用了私有代理集成，API 现已准备好，可用于部署和测试运行。使用非代理集成，您还必须设置方法响应和集成响应，就像您在使用 HTTP 自定义集成设置 API 时一样。

5. 测试 API 需要部署 API。如果您使用了阶段变量作为 VpcLink ID 的占位符，则这是必需的。要使用阶段变量部署 API，请按以下所示调用 `create-deployment` 命令：

```
aws apigateway create-deployment \
```

```
    --rest-api-id abcdef123 \
    --stage-name test \
    --variables vpcLinkId=gim7c3
```

要使用不同 VpcLink ID 更新阶段变量 (例如 *asf9d7*)，请调用 `update-stage` 命令：

```
aws apigateway update-stage \
    --rest-api-id abcdef123 \
    --stage-name test \
    --patch-operations op=replace,path='/variables/vpcLinkId',value='asf9d7'
```

使用以下命令来调用 API：

```
curl -X GET https://abcdef123.execute-api.us-east-2.amazonaws.com/test
```

此外，您可以在 Web 浏览器中键入 API 的调用 URL 来查看结果。

当您使用 VpcLink ID 文本硬编码 `connection-id` 时，您还可以调用 `test-invoke-method`，在部署 API 之前进行测试。

使用 OpenAPI 设置具有私有集成的 API

您可以通过导入 API 的 OpenAPI 文件，设置具有私有集成的 API。其设置类似于具有 HTTP 集成的 API 的 OpenAPI 定义，但有以下例外：

- 您必须明确将 `connectionType` 设置为 `VPC_LINK`。
- 您必须明确将 `connectionId` 设置为 `VpcLink` 的 ID 或者设置为引用 `VpcLink` 的 ID 的阶段变量。
- 私有集成中的 `uri` 参数指向 VPC 中的 HTTP/HTTPS 端点，但是用于设置集成请求的 `Host` 标头。
- 在 VPC 中，使用具有 HTTPS 端点的私有集成中的 `uri` 参数，根据在 VPC 端点上已安装证书中的名称验证所述域名。

您可以使用阶段变量来引用 `VpcLink` ID。或者，您可以将 ID 值直接分配到 `connectionId`。

以下 JSON 格式的 OpenAPI 文件显示的示例中，API 具有 VPC 链接，通过阶段变量 (`${stageVariables.vpcLinkId}`) 来引用：

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2017-11-17T04:40:23Z",
    "title": "MyApiWithVpcLink"
  },
  "host": "p3wocvip9a.execute-api.us-west-2.amazonaws.com",
  "basePath": "/test",
  "schemes": [
    "https"
  ],
  "paths": {
    "/": {
      "get": {
        "produces": [
          "application/json"
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "responses": {
            "default": {
              "statusCode": "200"
            }
          },
          "uri": "http://my-vpclink-test-nlb-1234567890abcdef.us-
east-2.amazonaws.com",
          "passthroughBehavior": "when_no_match",
          "connectionType": "VPC_LINK",
          "connectionId": "${stageVariables.vpcLinkId}",
          "httpMethod": "GET",
          "type": "http_proxy"
        }
      }
    }
  },
```

```
  "definitions": {
    "Empty": {
      "type": "object",
      "title": "Empty Schema"
    }
  }
}
```

用于私有集成的 API Gateway 账户

在创建 `VpcLink` 时，以下特定于区域的 API Gateway 账户 ID 会作为 `AllowedPrincipals` 自动添加到 VPC 端点服务中。

区域	账户 ID
us-east-1	392220576650
us-east-2	718770453195
us-west-1	968246515281
us-west-2	109351309407
ca-central-1	796887884028
eu-west-1	631144002099
eu-west-2	544388816663
eu-west-3	061510835048
eu-central-1	474240146802
eu-central-2	166639821150
eu-north-1	394634713161
eu-south-1	753362059629
eu-south-2	359345898052

区域	账户 ID
ap-northeast-1	969236854626
ap-northeast-2	020402002396
ap-northeast-3	360671645888
ap-southeast-1	195145609632
ap-southeast-2	798376113853
ap-southeast-3	652364314486
ap-southeast-4	849137399833
ap-south-1	507069717855
ap-south-2	644042651268
ap-east-1	174803364771
sa-east-1	287228555773
me-south-1	855739686837
me-central-1	614065512851

在 API Gateway 中设置模拟集成

Amazon API Gateway 支持 API 方法的模拟集成。借助该特征，API 开发人员可以直接从 API Gateway 生成 API 响应，无需集成后端。作为 API 开发人员，您可以在项目开发结束之前使用此特征，以便不妨碍需要使用 API 开展工作的相关团队。您还可以使用此特征来预配置 API 的登录页面，该网页可提供您的 API 概述和导航。有关此类登录页面的示例，请参阅 教程：通过导入示例创建 REST API中所述示例 API 的根资源上 GET 方法的集成请求和响应。

作为 API 开发人员，您可以决定 API Gateway 响应模拟集成的方式。为此，您可以配置方法的集成请求和集成响应，以将响应与给定的状态代码相关联。对于具有模拟集成以返回 `200` 响应的方法，请配置集成请求正文映射模板以返回下列内容。

```
{"statusCode": 200}
```

配置 200 集成响应以具有以下正文映射模板，例如：

```
{
    "statusCode": 200,
    "message": "Go ahead without me."
}
```

与此类似，举例而言，对于返回 500 错误响应的方法，请设置集成请求正文映射模板以返回下列内容。

```
{"statusCode": 500}
```

例如，使用以下映射模板设置 500 集成响应：

```
{
    "statusCode": 500,
    "message": "The invoked method is not supported on the API resource."
}
```

或者，您可以让模拟集成的方法返回默认集成响应，无需定义集成请求映射模板。默认集成响应是具有未定义 HTTP status regex (HTTP 状态正则表达式) 的响应。请确保设置了合适的传递行为。

> **Note**
>
> 模拟集成并非用于支持大型响应模板。如果您的使用案例需要它们，您应该考虑改为使用 Lambda 集成。

使用集成请求映射模板，您可以注入应用程序逻辑，用来确定基于特定条件返回什么模拟集成响应。例如，您可以在传入请求上使用 `scope` 查询参数来确定返回成功响应还是错误响应：

```
{
  #if( $input.params('scope') == "internal" )
    "statusCode": 200
  #else
    "statusCode": 500
```

```
  #end
}
```

这样，模拟集成的方法让内部调用通过，同时拒绝其他类型的调用并提供错误响应。

本部分介绍如何使用 API Gateway 控制台来启用 API 方法模拟集成。

主题

- 使用 API Gateway 控制台启用模拟集成

使用 API Gateway 控制台启用模拟集成

您必须在 API Gateway 中有一个可用的方法。按照教程：使用 HTTP 非代理集成构建 REST API中的说明进行操作。

1. 选择 API 资源，然后选择创建方法。

   要创建方法，请执行以下操作：

   a. 对于方法类型，选择一种方法。
   b. 对于集成类型，选择模拟。
   c. 选择创建方法。
   d. 在方法请求选项卡上，对于方法请求设置，选择编辑。
   e. 选择 URL 查询字符串参数。选择添加查询字符串，然后为名称中输入 **`scope`**。此查询参数确定调用方是否为内部。
   f. 选择保存。
2. 在方法响应选项卡上，选择创建响应，然后执行以下操作：

   a. 对于 HTTP 状态，输入 **`500`**。
   b. 选择保存。
3. 在集成请求选项卡上，对于集成请求设置，选择编辑。
4. 选择映射模板，然后执行以下操作：

   a. 选择添加映射模板。
   b. 对于内容类型，输入 **`application/json`**。
   c. 对于模板正文，输入以下内容：

```
{
  #if( $input.params('scope') == "internal" )
    "statusCode": 200
  #else
    "statusCode": 500
  #end
}
```

   d. 选择保存。

5. 在集成响应选项卡上，对于默认 - 响应，选择编辑。
6. 选择映射模板，然后执行以下操作：

   a. 对于内容类型，输入 **application/json**。
   b. 对于模板正文，输入以下内容：

```
{
    "statusCode": 200,
    "message": "Go ahead without me"
}
```

   c. 选择保存。

7. 选择创建响应。

   要创建 500 响应，请执行以下操作：

   a. 对于 HTTP 状态正则表达式，输入 **5\d{2}**。
   b. 对于方法响应状态，选择 **500**。
   c. 选择保存。
   d. 对于 5\d{2} - 响应，选择编辑。
   e. 选择映射模板，然后选择添加映射模板。
   f. 对于内容类型，输入 **application/json**。
   g. 对于模板正文，输入以下内容：

```
{
    "statusCode": 500,
    "message": "The invoked method is not supported on the API resource."
}
```

   h. 选择保存。

8. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。要测试模拟集成，请执行以下操作：

   a. 在查询字符串下，输入 `scope=internal`。选择 Test (测试)。测试结果显示：

   ```
   Request: /?scope=internal
   Status: 200
   Latency: 26 ms
   Response Body

   {
     "statusCode": 200,
     "message": "Go ahead without me"
   }

   Response Headers

   {"Content-Type":"application/json"}
   ```

   b. 在 `Query strings` 下输入 `scope=public` 或将其留空。选择 Test (测试)。测试结果显示：

   ```
   Request: /
   Status: 500
   Latency: 16 ms
   Response Body

   {
     "statusCode": 500,
     "message": "The invoked method is not supported on the API resource."
   }

   Response Headers

   {"Content-Type":"application/json"}
   ```

您也可以首先将标头添加到方法响应，然后在集成响应中设置标头映射，从而在模拟集成响应中返回标头。实际上，这是 API Gateway 控制台通过返回 CORS 需要的标头来启用 CORS 支持的方法。

在 API Gateway 中使用请求验证

您可以配置 API Gateway，使其在处理集成请求之前对 API 请求执行基本验证。如果验证失败，API Gateway 会立即取消请求、向调用方返回 400 错误响应，并在 CloudWatch Logs 中发布验证结果。这可以减少对后端进行的不必要调用。更重要的是，它可以让您把精力集中在特定于应用程序的验证工作上。您可以通过验证所需请求参数是否有效并且不为空，或为更复杂的数据验证指定一个模型架构，从而验证请求正文。

主题

- API Gateway 中的基本请求验证概览
- 了解数据模型
- 在 API Gateway 中设置基本请求验证
- 启用基本请求验证的示例 API 的 OpenAPI 定义
- AWS CloudFormation 模板，提供带有基本请求验证的示例 API

API Gateway 中的基本请求验证概览

API Gateway 可以执行基本请求验证，以便您可以专注于后端的应用程序特定验证。进行验证时，API Gateway 验证以下一项或两项条件：

- 传入请求的 URI、查询字符串和标头中带有所需的请求参数且都不为空。
- 适用的请求负载符合相关方法的所配置的 JSON 架构请求。

要开启验证，您需要在请求验证程序中指定验证规则，将验证程序添加到 API 的请求验证程序的映射中，并将验证程序分配给各个 API 方法。

> **Note**
>
> 请求正文验证和集成传递行为是两个独立的主题。当请求负载没有匹配的模型架构时，可以选择传递或阻止原始负载。有关更多信息，请参阅 集成传递行为。

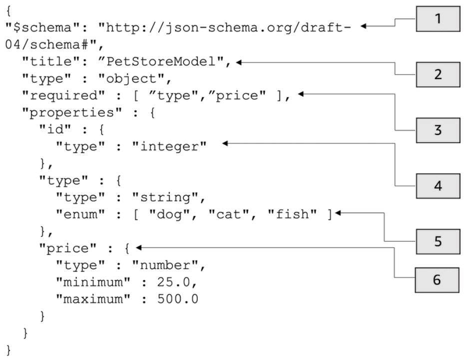
了解数据模型

在 API Gateway 中，模型定义负载的数据结构。在 API Gateway 中，使用 [JSON 架构草案 4](https://json-schema.org/) 定义模型。以下 JSON 对象是 Pet Store 示例中的示例数据。

```
{
    "id": 1,
    "type": "dog",
    "price": 249.99
}
```

数据包含宠物的 `id`、`type` 和 `price`。这些数据的模型允许您：

- 使用基本请求验证。
- 创建用于数据转换的映射模板。
- 生成 SDK 时创建用户定义的数据类型（UDT）。



```
{
"$schema": "http://json-schema.org/draft-
04/schema#",
  "title": ″PetStoreModel",
  "type" : "object",
  "required" : [ ″type",″price" ],
  "properties" : {
    "id" : {
      "type" : "integer"
    },
    "type" : {
      "type" : "string",
      "enum" : [ "dog", "cat", "fish" ]
    },
    "price" : {
      "type" : "number",
      "minimum" : 25.0,
      "maximum" : 500.0
    }
  }
}
```

在这个模型中：

1. `$schema` 对象表示一个有效的 JSON 架构版本标识符。此架构为 JSON 架构草案 v4。
2. `title` 对象是人类可读的模型标识符。此标题是 `PetStoreModel`。
3. `required` 验证关键字要求使用 `type` 和 `price` 进行基本请求验证。
4. 模型的 `properties` 为 `id`、`type` 和 `price`。每个对象都有模型中描述的属性。
5. 对象 `type` 只能具有值 `dog`、`cat` 或 `fish`。
6. 对象 `price` 是一个数字，并受 `minimum` 为 25 和 `maximum` 为 500 所限制。

PetStore 模型

```
1 {
2 "$schema": "http://json-schema.org/draft-04/schema#",
3  "title": "PetStoreModel",
4  "type" : "object",
5  "required" : [ "price", "type" ],
6  "properties" : {
7    "id" : {
8      "type" : "integer"
9    },
10    "type" : {
11      "type" : "string",
12      "enum" : [ "dog", "cat", "fish" ]
13    },
14    "price" : {
15      "type" : "number",
16      "minimum" : 25.0,
17      "maximum" : 500.0
18    }
19  }
20 }
```

在这个模型中：

1. 在第 2 行上，`$schema` 对象表示一个有效的 JSON 架构版本标识符。此架构为 JSON 架构草案 v4。
2. 在第 3 行上，`title` 对象是用户可读的模型标识符。此标题是 `PetStoreModel`。
3. 在第 5 行上，`required` 验证关键字要求使用 `type` 和 `price` 进行基本请求验证。
4. 在第 6 -- 17 行上，模型的 `properties` 为 `id`、`type` 和 `price`。每个对象都有模型中描述的属性。
5. 在第 12 行上，对象 `type` 只能具有值 `dog`、`cat` 或 `fish`。
6. 在第 14 -- 17 行上，对象 `price` 是一个数字，并受 `minimum` 为 25 和 `maximum` 为 500 所限制。

创建更复杂的模型

您可以使用 `$ref` 基元为较长的模型创建可重复使用的定义。例如，可以在描述 `price` 对象的 `definitions` 部分中创建称为 `Price` 的定义。`$ref` 的值是 `Price` 定义。

```
{
```

```
  "$schema" : "http://json-schema.org/draft-04/schema#",
  "title" : "PetStoreModelReUsableRef",
  "required" : ["price", "type" ],
  "type" : "object",
  "properties" : {
    "id" : {
      "type" : "integer"
    },
    "type" : {
      "type" : "string",
      "enum" : [ "dog", "cat", "fish" ]
    },
    "price" : {
        "$ref": "#/definitions/Price"
    }
  },
  "definitions" : {
      "Price": {
        "type" : "number",
        "minimum" : 25.0,
        "maximum" : 500.0
            }
      }
}
```

您也可以引用在外部模型文件中定义的另一个模型架构。将 `$ref` 属性的值设置为模型的位置。在以下示例中，`Price` 模型是在 API a1234 的 `PetStorePrice` 模型中定义的。

```
{
  "$schema" : "http://json-schema.org/draft-04/schema#",
  "title" : "PetStorePrice",
  "type": "number",
  "minimum": 25,
  "maximum": 500
}
```

较长的模型可以引用 `PetStorePrice` 模型。

```
{
  "$schema" : "http://json-schema.org/draft-04/schema#",
  "title" : "PetStoreModelReusableRefAPI",
  "required" : [ "price", "type" ],
  "type" : "object",
```

```
  "properties" : {
    "id" : {
      "type" : "integer"
    },
    "type" : {
      "type" : "string",
      "enum" : [ "dog", "cat", "fish" ]
    },
    "price" : {
        "$ref": "https://apigateway.amazonaws.com/restapis/a1234/models/PetStorePrice"
    }
  }
}
```

使用输出数据模型

如果您转换数据，则可以在集成响应中定义负载模型。生成 SDK 时可以使用负载模型。对于强类型语言（如 Java、Objective-C 或 Swift），对象对应于用户定义的数据类型 (UDT)。如果您在生成 SDK 时向其提供数据模型，API Gateway 将创建 UDT。有关数据转换的更多信息，请参阅了解映射模板。

输出数据

```
{
                [
  {
    "description" : "Item 1 is a dog.",
    "askingPrice" : 249.99
  },
  {
    "description" : "Item 2 is a cat.",
    "askingPrice" : 124.99
  },
  {
    "description" : "Item 3 is a fish.",
    "askingPrice" : 0.99
  }
]
}
```

输出模型

```
{
```

```
"$schema": "http://json-schema.org/
draft-04/schema#",
  "title": ”PetStoreOutputModel",
  "type" : "object",
  "required" : [ "description",
 "askingPrice" ],
  "properties" : {
    "description" : {
      "type" : "string"
    },
    "askingPrice" : {
      "type" : "number",
      "minimum" : 25.0,
      "maximum" : 500.0
    }
  }
}
```

借助此模型，您可以调用 SDK，以便通过读取 `PetStoreOutputModel[i].description` 和 `PetStoreOutputModel[i].askingPrice` 属性来检索 `description` 和 `askingPrice` 属性值。如果未提供模型，API Gateway 将使用空模型创建默认 UDT。

后续步骤

- 本节提供的资源可用于获得有关本主题中介绍的概念的更多知识。

  您可以按照请求验证教程进行操作：

  - 使用 API Gateway 控制台设置请求验证
  - 使用 AWS CLI 设置基本请求验证
  - 使用 OpenAPI 定义设置基本请求验证
- 您可以获得有关数据转换和映射模板了解映射模板的更多信息。
- 您还可查看更复杂的数据模型。请参阅 API Gateway 的示例数据模型和映射模板。

在 API Gateway 中设置基本请求验证

本节介绍如何使用控制台、AWS CLI 和 OpenAPI 定义为 API Gateway 设置请求验证。

主题

- 使用 API Gateway 控制台设置请求验证
- 使用 AWS CLI 设置基本请求验证
- 使用 OpenAPI 定义设置基本请求验证

使用 API Gateway 控制台设置请求验证

您可以使用 API Gateway 控制台从 API 请求的三个验证程序中选择一个来验证请求：

- 验证正文。
- 验证查询字符串参数和标头。
- 验证正文、查询字符串参数和标头。

当您对 API 方法应用以上一种验证程序时，API Gateway 控制台会向 API 的 RequestValidators 映射添加该验证程序。

要按本教程操作，您将使用 AWS CloudFormation 模板来创建不完整的 API Gateway API。此 API 拥有的 `/validator` 资源具有 `GET` 和 `POST` 方法。两种方法都与 `http://petstore-demo-endpoint.execute-api.com/petstore/pets` HTTP 端点集成。您将配置两种请求验证：

- 在 `GET` 方法中，您将为 URL 查询字符串参数配置请求验证。
- 在 `POST` 方法中，您将为请求正文配置请求验证。

这将只允许特定的 API 调用传递给 API。

下载并解压缩适用于 AWS CloudFormation 的应用程序创建模板。您将使用此模板创建不完整的 API。您将在 API Gateway 控制台中完成其余步骤。

创建 AWS CloudFormation 堆栈

1. 打开 AWS CloudFormation 控制台，地址：https://console.aws.amazon.com/cloudformation。
2. 选择创建堆栈，然后选择使用新资源(标准)。
3. 对于指定模板，选择上传模板文件。
4. 选择您下载的模板。
5. 选择下一步。
6. 对于堆栈名称，输入 **`request-validation-tutorial-console`**，然后选择下一步。
7. 对于配置堆栈选项，请选择下一步。

8. 对于功能，请确认 AWS CloudFormation 可以在您的账户中创建 IAM 资源。
9. 选择提交。

AWS CloudFormation 预置在模板中指定的资源。完成资源预置可能需要几分钟时间。当 AWS CloudFormation 堆栈的状态为 CREATE_COMPLETE 时，您就可以继续下一步了。

选择您新创建的 API

1. 选择新创建的 **request-validation-tutorial-console** 堆栈。
2. 选择资源。
3. 在物理 ID 下，选择您的 API。此链接将引导您进入 API Gateway 控制台。

在修改 `GET` 和 `POST` 方法之前，必须创建模型。

创建模型

1. 需要一个模型来对传入请求的正文使用请求验证。要创建模型，请在主导航窗格中选择模型。
2. 选择创建模型。
3. 对于名称，请输入 **PetStoreModel**。
4. 对于内容类型，输入 **application/json**。如果未找到匹配的内容类型，则不执行请求验证。要使用同一模型而不考虑内容类型，请输入 **$default**。
5. 对于描述，输入 **My PetStore Model** 作为模型描述。
6. 对于模型架构，将以下模型粘贴到代码编辑器中，然后选择创建。

```
{
  "type" : "object",
  "required" : [ "name", "price", "type" ],
  "properties" : {
    "id" : {
      "type" : "integer"
    },
    "type" : {
      "type" : "string",
      "enum" : [ "dog", "cat", "fish" ]
    },
    "name" : {
      "type" : "string"
    },
```

```
      "price" : {
        "type" : "number",
        "minimum" : 25.0,
        "maximum" : 500.0
      }
    }
  }
```

有关模型的更多信息，请参阅了解数据模型。

为 **GET** 方法配置请求验证

1. 在主导航窗格中，选择资源，然后选择 GET 方法。
2. 在方法请求选项卡上的方法请求设置下，选择编辑。
3. 对于请求验证程序，选择验证查询字符串参数和标头。
4. 在 URL 查询字符串参数下，执行以下操作：
    a. 选择添加查询字符串。
    b. 在名称中，输入 **petType**。
    c. 打开必需。
    d. 将缓存保持为关闭状态。
5. 选择保存。
6. 在集成请求选项卡的集成请求设置下，选择编辑。
7. 在 URL 查询字符串参数下，执行以下操作：
    a. 选择添加查询字符串。
    b. 在名称中，输入 **petType**。
    c. 对于映射自，输入 **method.request.querystring.petType**。这会将 **petType** 映射到宠物的类型。

       有关数据映射的更多信息，请参阅数据映射教程。
    d. 将缓存保持为关闭状态。
8. 选择保存。

为 **GET** 方法测试请求验证

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
2. 对于查询字符串，输入 **`petType=dog`**，然后选择测试。
3. 方法测试将返回 `200 OK` 并提供狗的列表。

   有关如何转换此输出数据的信息，请参阅数据映射教程。
4. 删除 **`petType=dog`** 并选择测试。
5. 方法测试将返回 `400` 错误并显示以下错误消息：

```
{
  "message": "Missing required request parameters: [petType]"
}
```

为 **POST** 方法配置请求验证

1. 在主导航窗格中，选择资源，然后选择 POST 方法。
2. 在方法请求选项卡上的方法请求设置下，选择编辑。
3. 对于请求验证程序，选择验证正文。
4. 在请求正文下，选择添加模型。
5. 对于内容类型，输入 **`application/json`**，然后对于模型，选择 PetStoreModel。
6. 选择保存。

为 **POST** 方法测试请求验证

1. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
2. 对于请求正文，将以下内容粘贴到代码编辑器中：

```
{
  "id": 2,
  "name": "Bella",
  "type": "dog",
  "price": 400
}
```

   选择测试。

3. 方法测试将返回 `200 OK` 和成功消息。
4. 对于请求正文，将以下内容粘贴到代码编辑器中：

   ```
   {
     "id": 2,
     "name": "Bella",
     "type": "dog",
     "price": 4000
   }
   ```

   选择测试。
5. 方法测试将返回 `400` 错误并显示以下错误消息：

   ```
   {
    "message": "Invalid request body"
   }
   ```

   在测试日志的底部，将返回请求正文无效的原因。在这种情况下，宠物的价格超出了模型中规定的最高价格。

删除 AWS CloudFormation 堆栈

1. 打开 AWS CloudFormation 控制台，地址：https://console.aws.amazon.com/cloudformation。
2. 选择您的 AWS CloudFormation 堆栈。
3. 选择删除，然后确认您的选择。

后续步骤

- 有关如何转换输出数据和执行更多数据映射的信息，请参阅数据映射教程。
- 按照使用 AWS CLI 设置基本请求验证教程操作，使用 AWS CLI 执行类似的步骤。

使用 AWS CLI 设置基本请求验证

您可以使用 AWS CLI 创建验证程序来设置请求验证。要按本教程操作，您将使用 AWS CloudFormation 模板来创建不完整的 API Gateway API。

> **Note**
>
> 这与控制台教程的 AWS CloudFormation 模板不同。

使用预先公开的 `/validator` 资源，您将创建 `GET` 和 `POST` 方法。两种方法都将与 `http://petstore-demo-endpoint.execute-api.com/petstore/pets` HTTP 端点集成。您将配置以下两个请求验证：

- 在 `GET` 方法上，您将创建一个 `params-only` 验证程序来验证 URL 查询字符串参数。
- 在 `POST` 方法上，您将创建一个 `body-only` 验证程序来验证请求正文。

这将只允许特定的 API 调用传递给 API。

创建 AWS CloudFormation 堆栈

下载并解压缩适用于 AWS CloudFormation 的应用程序创建模板。

要完成以下教程，您需要 AWS Command Line Interface（AWS CLI）版本 2。

对于长命令，使用转义字符 (\) 将命令拆分为多行。

> **Note**
>
> 在 Windows 中，操作系统的内置终端不支持您经常使用的某些 Bash CLI 命令（例如 `zip`）。安装 Windows Subsystem for Linux，获取 Ubuntu 和 Bash 与 Windows 集成的版本。本指南中的示例 CLI 命令使用 Linux 格式。如果您使用的是 Windows CLI，则必须重新格式化包含内联 JSON 文档的命令。

1. 输入以下命令创建 AWS CloudFormation 堆栈。

   ```
   aws cloudformation create-stack --stack-name request-validation-tutorial-cli
    --template-body file://request-validation-tutorial-cli.zip --capabilities
    CAPABILITY_NAMED_IAM
   ```

2. AWS CloudFormation 预置在模板中指定的资源。完成资源预置可能需要几分钟时间。使用以下命令查看 AWS CloudFormation 的状态。

```
aws cloudformation describe-stacks --stack-name request-validation-tutorial-cli
```

3. 当 AWS CloudFormation 堆栈的状态为 `StackStatus: "CREATE_COMPLETE"` 时，使用以下命令检索将来步骤的相关输出值。

    ```
    aws cloudformation describe-stacks --stack-name request-validation-tutorial-cli
     --query "Stacks[*].Outputs[*].{OutputKey: OutputKey, OutputValue: OutputValue,
     Description: Description}"
    ```

    输出值包括：

    - ApiId，这是 API 的 ID。对于本教程，API ID 为 `abc123`。
    - ResourceId，这是在其中公开 `GET` 和 `POST` 方法的验证程序资源的 ID。对于本教程，资源 ID 为 `efg456`

创建请求验证程序并导入模型

1. 需要验证程序才能通过 AWS CLI 使用请求验证。使用以下命令创建仅验证请求参数的验证程序。

    ```
    aws apigateway create-request-validator --rest-api-id abc123 \
          --no-validate-request-body \
          --validate-request-parameters \
          --name params-only
    ```

    记下 `params-only` 验证程序的 ID。

2. 使用以下命令创建仅验证请求正文的验证程序。

    ```
    aws apigateway create-request-validator --rest-api-id abc123 \
          --validate-request-body \
          --no-validate-request-parameters \
          --name body-only
    ```

    记下 `body-only` 验证程序的 ID。

3. 需要一个模型来对传入请求的正文使用请求验证。使用以下命令导入模型。

    ```
    aws apigateway create-model --rest-api-id abc123 --name PetStoreModel --description
     'My PetStore Model' --content-type 'application/json' --schema '{"type":
     "object", "required" : [ "name", "price", "type" ], "properties" : { "id" :
    ```

```
{"type" : "integer"},"type" : {"type" : "string", "enum" : [ "dog", "cat",
"fish" ]},"name" : { "type" : "string"},"price" : {"type" : "number","minimum" :
25.0, "maximum" : 500.0}}}}'
```

如果未找到匹配的内容类型，则不执行请求验证。要使用同一模型而不考虑内容类型，请指定 `$default` 作为键。

创建 GET 和 POST 方法

1. 使用以下命令对 `/validate` 资源添加 GET HTTP 方法。此命令创建 GET 方法，添加 `params-only` 验证程序，并根据需要设置查询字符串 `petType`。

   ```
   aws apigateway put-method --rest-api-id abc123 \
          --resource-id efg456 \
          --http-method GET \
          --authorization-type "NONE" \
          --request-validator-id aaa111 \
          --request-parameters "method.request.querystring.petType=true"
   ```

   使用以下命令对 `/validate` 资源添加 POST HTTP 方法。此命令创建 POST 方法，添加 `body-only` 验证程序，并将模型附加到仅限正文的验证程序。

   ```
   aws apigateway put-method --rest-api-id abc123 \
          --resource-id efg456 \
          --http-method POST \
          --authorization-type "NONE" \
          --request-validator-id bbb222 \
          --request-models 'application/json'=PetStoreModel
   ```

2. 使用以下命令设置 `GET /validate` 方法的 `200 OK` 响应。

   ```
   aws apigateway put-method-response --rest-api-id abc123  \
               --resource-id efg456 \
               --http-method GET \
               --status-code 200
   ```

   使用以下命令设置 `POST /validate` 方法的 `200 OK` 响应。

   ```
   aws apigateway put-method-response --rest-api-id abc123  \
               --resource-id efg456 \
   ```

```
            --http-method POST \
            --status-code 200
```

3. 使用以下命令通过指定的 HTTP 端点为 `GET /validation` 方法设置 `Integration`。

   ```
   aws apigateway put-integration --rest-api-id abc123  \
               --resource-id efg456 \
               --http-method GET \
               --type HTTP \
               --integration-http-method GET \
               --request-parameters '{"integration.request.querystring.type" :
    "method.request.querystring.petType"}' \
               --uri 'http://petstore-demo-endpoint.execute-api.com/petstore/pets'
   ```

   使用以下命令通过指定的 HTTP 端点为 `POST /validation` 方法设置 `Integration`。

   ```
   aws apigateway put-integration --rest-api-id abc123  \
                 --resource-id efg456 \
                 --http-method POST \
                 --type HTTP \
                 --integration-http-method GET \
                 --uri 'http://petstore-demo-endpoint.execute-api.com/petstore/pets'
   ```

4. 使用以下命令设置 `GET /validation` 方法的集成响应。

   ```
   aws apigateway put-integration-response --rest-api-id abc123 \
                 --resource-id efg456\
                 --http-method GET \
                 --status-code 200 \
                 --selection-pattern ""
   ```

   使用以下命令设置 `POST /validation` 方法的集成响应。

   ```
   aws apigateway put-integration-response --rest-api-id abc123 \
               --resource-id efg456 \
               --http-method POST \
               --status-code 200 \
               --selection-pattern ""
   ```

测试 API

1. 要测试将对查询字符串执行请求验证的 GET 方法，请使用以下命令：

   ```
   aws apigateway test-invoke-method --rest-api-id abc123 \
               --resource-id efg456 \
               --http-method GET \
               --path-with-query-string '/validate?petType=dog'
   ```

   结果将返回 200 OK 和狗的列表。

2. 使用以下命令在不包含查询字符串 petType 的情况下进行测试

   ```
   aws apigateway test-invoke-method --rest-api-id abc123 \
               --resource-id efg456 \
               --http-method GET
   ```

   结果将返回 400 错误。

3. 要测试将对请求正文执行请求验证的 POST 方法，请使用以下命令：

   ```
    aws apigateway test-invoke-method --rest-api-id abc123 \
               --resource-id efg456 \
               --http-method POST \
               --body '{"id": 1, "name": "bella", "type": "dog", "price" : 400 }'
   ```

   结果将返回 200 OK 和一条成功消息。

4. 使用以下命令通过无效的正文进行测试。

   ```
    aws apigateway test-invoke-method --rest-api-id abc123 \
                 --resource-id efg456 \
                 --http-method POST \
                 --body '{"id": 1, "name": "bella", "type": "dog", "price" : 1000 }'
   ```

   结果将返回 400 错误，因为狗的价格超过了模型定义的最高价格。

删除 AWS CloudFormation 堆栈

- 使用以下命令删除您的 AWS CloudFormation 资源。

```
aws cloudformation delete-stack  --stack-name request-validation-tutorial-cli
```

使用 OpenAPI 定义设置基本请求验证

您可以在 API 级别声明请求验证程序，方法是在 x-amazon-apigateway-request-validators 对象 映射中指定一组 x-amazon-apigateway-request-validators.requestValidator 对象 对象，以选择将对请求的哪个部分进行验证。在示例 OpenAPI 定义中，有两个验证程序：

- `all` 验证程序，它验证正文（使用 `RequestBodyModel` 数据模型）和参数。
- `param-only`，它只验证参数。

要在 API 的所有方法上开启请求验证程序，请在 API 级别指定 OpenAPI 定义的 x-amazon-apigateway-request-validator 属性 属性。在示例 OpenAPI 定义中，`all` 验证器用于所有 API 方法，除非被覆盖。使用模型验证正文时，如果找不到匹配的内容类型，则不执行请求验证。要使用同一模型而不考虑内容类型，请指定 `$default` 作为键。

要针对单独的方法开启请求验证程序，请在方法级别指定 `x-amazon-apigateway-request-validator` 属性。在示例 OpenAPI 定义中，`param-only` 验证程序会覆盖 `GET` 方法上的 `all` 验证程序。

要将 OpenAPI 示例导入 API Gateway，请参阅以下有关将区域 API 导入到 API Gateway 中或将边缘优化的 API 导入 API Gateway的说明。

OpenAPI 3.0

```
{
  "openapi" : "3.0.1",
  "info" : {
    "title" : "ReqValidators Sample",
    "version" : "1.0.0"
  },
  "servers" : [ {
    "url" : "/{basePath}",
    "variables" : {
      "basePath" : {
        "default" : "/v1"
      }
    }
```

```
  } ],
  "paths" : {
    "/validation" : {
      "get" : {
        "parameters" : [ {
          "name" : "q1",
          "in" : "query",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "200" : {
            "description" : "200 response",
            "headers" : {
              "test-method-response-header" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/ArrayOfError"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-request-validator" : "params-only",
        "x-amazon-apigateway-integration" : {
          "httpMethod" : "GET",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
          "responses" : {
            "default" : {
              "statusCode" : "400",
              "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static
value'"
              },
              "responseTemplates" : {
                "application/xml" : "xml 400 response template",
```

```
                "application/json" : "json 400 response template"
              }
            },
            "2\\d{2}" : {
              "statusCode" : "200"
            }
          },
          "requestParameters" : {
            "integration.request.querystring.type" : "method.request.querystring.q1"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "http"
        }
      },
      "post" : {
        "parameters" : [ {
          "name" : "h1",
          "in" : "header",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "requestBody" : {
          "content" : {
            "application/json" : {
              "schema" : {
                "$ref" : "#/components/schemas/RequestBodyModel"
              }
            }
          },
          "required" : true
        },
        "responses" : {
          "200" : {
            "description" : "200 response",
            "headers" : {
              "test-method-response-header" : {
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
```

```
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/ArrayOfError"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-request-validator" : "all",
        "x-amazon-apigateway-integration" : {
          "httpMethod" : "POST",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
          "responses" : {
            "default" : {
              "statusCode" : "400",
              "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static
value'"
              },
              "responseTemplates" : {
                "application/xml" : "xml 400 response template",
                "application/json" : "json 400 response template"
              }
            },
            "2\\d{2}" : {
              "statusCode" : "200"
            }
          },
          "requestParameters" : {
            "integration.request.header.custom_h1" : "method.request.header.h1"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "http"
        }
      }
    }
  },
  "components" : {
    "schemas" : {
      "RequestBodyModel" : {
        "required" : [ "name", "price", "type" ],
        "type" : "object",
        "properties" : {
          "id" : {
```

```
            "type" : "integer"
          },
          "type" : {
            "type" : "string",
            "enum" : [ "dog", "cat", "fish" ]
          },
          "name" : {
            "type" : "string"
          },
          "price" : {
            "maximum" : 500.0,
            "minimum" : 25.0,
            "type" : "number"
          }
        }
      },
      "ArrayOfError" : {
        "type" : "array",
        "items" : {
          "$ref" : "#/components/schemas/Error"
        }
      },
      "Error" : {
        "type" : "object"
      }
    }
  },
  "x-amazon-apigateway-request-validators" : {
    "all" : {
      "validateRequestParameters" : true,
      "validateRequestBody" : true
    },
    "params-only" : {
      "validateRequestParameters" : true,
      "validateRequestBody" : false
    }
  }
}
```

OpenAPI 2.0

```
{
  "swagger" : "2.0",
```

```
  "info" : {
    "version" : "1.0.0",
    "title" : "ReqValidators Sample"
  },
  "basePath" : "/v1",
  "schemes" : [ "https" ],
  "paths" : {
    "/validation" : {
      "get" : {
        "produces" : [ "application/json", "application/xml" ],
        "parameters" : [ {
          "name" : "q1",
          "in" : "query",
          "required" : true,
          "type" : "string"
        } ],
        "responses" : {
          "200" : {
            "description" : "200 response",
            "schema" : {
              "$ref" : "#/definitions/ArrayOfError"
            },
            "headers" : {
              "test-method-response-header" : {
                "type" : "string"
              }
            }
          }
        },
        "x-amazon-apigateway-request-validator" : "params-only",
        "x-amazon-apigateway-integration" : {
          "httpMethod" : "GET",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
          "responses" : {
            "default" : {
              "statusCode" : "400",
              "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static
value'"
              },
              "responseTemplates" : {
                "application/xml" : "xml 400 response template",
                "application/json" : "json 400 response template"
              }
```

```
              },
              "2\\d{2}" : {
                "statusCode" : "200"
              }
            },
            "requestParameters" : {
              "integration.request.querystring.type" : "method.request.querystring.q1"
            },
            "passthroughBehavior" : "when_no_match",
            "type" : "http"
          }
        },
        "post" : {
          "consumes" : [ "application/json" ],
          "produces" : [ "application/json", "application/xml" ],
          "parameters" : [ {
            "name" : "h1",
            "in" : "header",
            "required" : true,
            "type" : "string"
          }, {
            "in" : "body",
            "name" : "RequestBodyModel",
            "required" : true,
            "schema" : {
              "$ref" : "#/definitions/RequestBodyModel"
            }
          } ],
          "responses" : {
            "200" : {
              "description" : "200 response",
              "schema" : {
                "$ref" : "#/definitions/ArrayOfError"
              },
              "headers" : {
                "test-method-response-header" : {
                  "type" : "string"
                }
              }
            }
          },
          "x-amazon-apigateway-request-validator" : "all",
          "x-amazon-apigateway-integration" : {
            "httpMethod" : "POST",
```

```
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
          "responses" : {
            "default" : {
              "statusCode" : "400",
              "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static 
  value'"
              },
              "responseTemplates" : {
                "application/xml" : "xml 400 response template",
                "application/json" : "json 400 response template"
              }
            },
            "2\\d{2}" : {
              "statusCode" : "200"
            }
          },
          "requestParameters" : {
            "integration.request.header.custom_h1" : "method.request.header.h1"
          },
          "passthroughBehavior" : "when_no_match",
          "type" : "http"
        }
      }
    }
  },
  "definitions" : {
    "RequestBodyModel" : {
      "type" : "object",
      "required" : [ "name", "price", "type" ],
      "properties" : {
        "id" : {
          "type" : "integer"
        },
        "type" : {
          "type" : "string",
          "enum" : [ "dog", "cat", "fish" ]
        },
        "name" : {
          "type" : "string"
        },
        "price" : {
          "type" : "number",
          "minimum" : 25.0,
```

```
            "maximum" : 500.0
          }
        }
      },
      "ArrayOfError" : {
        "type" : "array",
        "items" : {
          "$ref" : "#/definitions/Error"
        }
      },
      "Error" : {
        "type" : "object"
      }
    },
    "x-amazon-apigateway-request-validators" : {
      "all" : {
        "validateRequestParameters" : true,
        "validateRequestBody" : true
      },
      "params-only" : {
        "validateRequestParameters" : true,
        "validateRequestBody" : false
      }
    }
  }
```

启用基本请求验证的示例 API 的 OpenAPI 定义

以下 OpenAPI 定义定义了一个启用了请求验证的示例 API。该 API 是 PetStore API 的一部分。其使用 `POST` 方法将宠物添加到 `pets` 集合，并使用 `GET` 方法按指定类型查询宠物。

在 API 级别的 `x-amazon-apigateway-request-validators` 映射中声明了两种请求验证程序。`params-only` 验证程序在 API 上启用，并由 `GET` 方法继承。该验证程序让 API Gateway 可以验证所需的查询参数 (q1) 包含在传入请求内且不为空。`all` 验证程序在 `POST` 方法上启用。该验证程序验证所需的标头参数 (h1) 是否已设置且不为空。它还会验证有效负载格式是否符合指定的 `RequestBodyModel`。如果未找到匹配的内容类型，则不执行请求验证。使用模型验证正文时，如果找不到匹配的内容类型，则不执行请求验证。要使用同一模型而不考虑内容类型，请指定 `$default` 作为键。

这种模型要求输入 JSON 对象包含 `name`、`type` 和 `price` 属性。`name` 属性可以是任何字符串，`type` 必须是一种指定枚举字段 (`["dog", "cat", "fish"]`)，而 `price` 必须介于 25 和 500 之间。`id` 参数不是必需参数。

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "title": "ReqValidators Sample",
    "version": "1.0.0"
  },
  "schemes": [
    "https"
  ],
  "basePath": "/v1",
  "produces": [
    "application/json"
  ],
  "x-amazon-apigateway-request-validators" : {
    "all" : {
      "validateRequestBody" : true,
      "validateRequestParameters" : true
    },
    "params-only" : {
      "validateRequestBody" : false,
      "validateRequestParameters" : true
    }
  },
  "x-amazon-apigateway-request-validator" : "params-only",
  "paths": {
    "/validation": {
      "post": {
        "x-amazon-apigateway-request-validator" : "all",
        "parameters": [
          {
            "in": "header",
            "name": "h1",
            "required": true
          },
          {
            "in": "body",
            "name": "RequestBodyModel",
```

```
            "required": true,
            "schema": {
              "$ref": "#/definitions/RequestBodyModel"
            }
          }
        ],
        "responses": {
          "200": {
            "schema": {
              "type": "array",
              "items": {
                "$ref": "#/definitions/Error"
              }
            },
            "headers" : {
              "test-method-response-header" : {
                "type" : "string"
              }
            }
          }
        },
        "security" : [{
          "api_key" : []
        }],
        "x-amazon-apigateway-auth" : {
          "type" : "none"
        },
        "x-amazon-apigateway-integration" : {
          "type" : "http",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
          "httpMethod" : "POST",
          "requestParameters": {
            "integration.request.header.custom_h1": "method.request.header.h1"
          },
          "responses" : {
            "2\\d{2}" : {
              "statusCode" : "200"
            },
            "default" : {
              "statusCode" : "400",
              "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static
 value'"
              },
```

```
              "responseTemplates" : {
                "application/json" : "json 400 response template",
                "application/xml" : "xml 400 response template"
              }
            }
          }
        }
      },
      "get": {
        "parameters": [
          {
            "name": "q1",
            "in": "query",
            "required": true
          }
        ],
        "responses": {
          "200": {
            "schema": {
              "type": "array",
              "items": {
                "$ref": "#/definitions/Error"
              }
            },
            "headers" : {
              "test-method-response-header" : {
                "type" : "string"
              }
            }
          }
        },
        "security" : [{
          "api_key" : []
        }],
        "x-amazon-apigateway-auth" : {
          "type" : "none"
        },
        "x-amazon-apigateway-integration" : {
          "type" : "http",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
          "httpMethod" : "GET",
          "requestParameters": {
            "integration.request.querystring.type": "method.request.querystring.q1"
          },
```

```
          "responses" : {
            "2\\d{2}" : {
              "statusCode" : "200"
            },
            "default" : {
              "statusCode" : "400",
              "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static
 value'"
              },
              "responseTemplates" : {
                "application/json" : "json 400 response template",
                "application/xml" : "xml 400 response template"
              }
            }
          }
        }
      }
    }
  },
  "definitions": {
    "RequestBodyModel": {
      "type": "object",
      "properties": {
        "id":    { "type": "integer" },
        "type":  { "type": "string", "enum": ["dog", "cat", "fish"] },
        "name":  { "type": "string" },
        "price": { "type": "number", "minimum": 25, "maximum": 500 }
      },
      "required": ["type", "name", "price"]
    },
    "Error": {
      "type": "object",
      "properties": {

      }
    }
  }
}
```

AWS CloudFormation 模板，提供带有基本请求验证的示例 API

以下 AWS CloudFormation 示例模板定义一个启用了请求验证的示例 API。该 API 是 PetStore API 的一部分。其使用 `POST` 方法将宠物添加到 `pets` 集合，并使用 `GET` 方法按指定类型查询宠物。

其中声明了两个请求验证程序：

`GETValidator`

此验证程序已在 `GET` 方法上启用。它允许 API Gateway 验证所需的查询参数 (q1) 是否包含在传入请求内且不为空。

`POSTValidator`

此验证程序已在 `POST` 方法上启用。它允许 API Gateway 在内容类型为 `application/json` 时，验证负载请求格式是否遵循指定的 `RequestBodyModel`，如果未找到匹配的内容类型，则不执行请求验证。要使用同一模型而不考虑内容类型，请指定`$default`。`RequestBodyModel` 包含一个额外的模型 `RequestBodyModelId`，用于定义宠物 ID。

```
AWSTemplateFormatVersion: 2010-09-09
Parameters:
  StageName:
    Type: String
    Default: v1
    Description: Name of API stage.
Resources:
  Api:
    Type: 'AWS::ApiGateway::RestApi'
    Properties:
      Name: ReqValidatorsSample
  RequestBodyModelId:
    Type: 'AWS::ApiGateway::Model'
    Properties:
      RestApiId: !Ref Api
      ContentType: application/json
      Description: Request body model for Pet ID.
      Schema:
        $schema: 'http://json-schema.org/draft-04/schema#'
        title: RequestBodyModelId
        properties:
            id:
              type: integer
```

```
  RequestBodyModel:
    Type: 'AWS::ApiGateway::Model'
    Properties:
      RestApiId: !Ref Api
      ContentType: application/json
      Description: Request body model for Pet type, name, price, and ID.
      Schema:
        $schema: 'http://json-schema.org/draft-04/schema#'
        title: RequestBodyModel
        required:
          - price
          - name
          - type
        type: object
        properties:
            id:
              "$ref": !Sub 
                - 'https://apigateway.amazonaws.com/restapis/${Api}/models/
${RequestBodyModelId}'
                - Api: !Ref Api
                  RequestBodyModelId: !Ref RequestBodyModelId
            price:
              type: number
              minimum: 25
              maximum: 500
            name:
              type: string
            type:
              type: string
              enum:
                - "dog"
                - "cat"
                - "fish"
  GETValidator:
    Type: AWS::ApiGateway::RequestValidator
    Properties:
      Name: params-only
      RestApiId: !Ref Api
      ValidateRequestBody: False
      ValidateRequestParameters: True 
  POSTValidator:
    Type: AWS::ApiGateway::RequestValidator
    Properties:
      Name: body-only
```

```
      RestApiId: !Ref Api
      ValidateRequestBody: True
      ValidateRequestParameters: False
  ValidationResource:
    Type: 'AWS::ApiGateway::Resource'
    Properties:
      RestApiId: !Ref Api
      ParentId: !GetAtt Api.RootResourceId
      PathPart: 'validation'
  ValidationMethodGet:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref ValidationResource
      HttpMethod: GET
      AuthorizationType: NONE
      RequestValidatorId: !Ref GETValidator
      RequestParameters:
        method.request.querystring.q1: true
      Integration:
        Type: HTTP_PROXY
        IntegrationHttpMethod: GET
        Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets/
  ValidationMethodPost:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref ValidationResource
      HttpMethod: POST
      AuthorizationType: NONE
      RequestValidatorId: !Ref POSTValidator
      RequestModels:
        application/json : !Ref RequestBodyModel
      Integration:
        Type: HTTP_PROXY
        IntegrationHttpMethod: POST
        Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets/
  ApiDeployment:
    Type: 'AWS::ApiGateway::Deployment'
    DependsOn:
      - ValidationMethodGet
      - RequestBodyModel
    Properties:
      RestApiId: !Ref Api
```

```
      StageName: !Sub '${StageName}'
Outputs:
  ApiRootUrl:
    Description: Root Url of the API
    Value: !Sub 'https://${Api}.execute-api.${AWS::Region}.amazonaws.com/${StageName}'
```

为 REST API 设置数据转换

在 API Gateway 中，API 的方法请求采用的负载格式可能与集成请求负载的格式不同。同样，后端返回的集成响应负载可能不同于方法响应负载。您可以使用映射模板，跨 API Gateway 映射 URL 路径参数、URL 查询字符串参数、HTTP 标头和请求正文。

映射模板是一个用 Velocity 模板语言 (VTL) 表示的脚本，应用于使用 JSONPath 表达式的负载。

负载可以拥有符合 JSON 架构草案 4 的数据模型。要了解有关模型的更多信息，请参阅了解数据模型。

> **Note**
>
> 您不必定义任何模型即可创建映射模板，但必须定义模型才能让 API Gateway 生成 SDK 或为您的 API 开启请求正文验证。

主题

了解映射模板

在 API Gateway 中，API 的方法请求或响应采用的负载格式可能与集成请求或响应的格式不同。

您可以转换数据，以便：

- 将负载与 API 指定的格式进行匹配。
- 覆盖 API 的请求和响应参数以及状态代码。
- 返回客户端选择的响应标头。
- 在 HTTP 代理或 AWS 服务 代理的方法请求中关联路径参数、查询字符串参数或标头参数。
- 选择要使用与 AWS 服务（例如 Amazon DynamoDB 或 Lambda 函数或 HTTP 端点）的集成发送哪些数据。

您可以使用映射模板来转换数据。映射模板是一个用 Velocity 模板语言（VTL）表示的脚本，应用于使用 JSONPath 的负载。

以下示例显示了 PetStore 数据转换的输入数据、映射模板和输出数据。

输入数据

```
[
  {
    "id": 1,
    "type": "dog",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "cat",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
```

映射模板

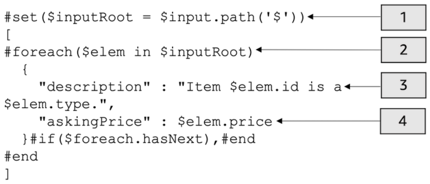
```
#set($inputRoot = $input.path('$'))
[
#foreach($elem in $inputRoot)
  {
    "description" : "Item $elem.id is a $elem.type.",
    "askingPrice" : $elem.price
  }#if($foreach.hasNext),#end

#end
```

```
]
```

输出数据

```
[
  {
    "description" : "Item 1 is a dog.",
    "askingPrice" : 249.99
  },
  {
    "description" : "Item 2 is a cat.",
    "askingPrice" : 124.99
  },
  {
    "description" : "Item 3 is a fish.",
    "askingPrice" : 0.99
  }
]
```

下图显示了此映射模板的详细信息。



```
#set($inputRoot = $input.path('$'))
[
#foreach($elem in $inputRoot)
  {
    "description" : "Item $elem.id is a
$elem.type.",
    "askingPrice" : $elem.price
  }#if($foreach.hasNext),#end
#end
]
```

1. `$inputRoot` 变量表示上一部分的原始 JSON 数据中的根对象。指令以 # 符号开头。
2. `foreach` 循环遍历原始 JSON 数据中的每个对象。
3. 该描述是原始 JSON 数据中宠物的 `id` 和 `type` 拼接的结果。
4. `askingPrice` 是原始 JSON 数据中的 `price`。

PetStore 映射模板

```
#set($inputRoot = $input.path('$'))
[
#foreach($elem in $inputRoot)
 {
   "description" : "Item $elem.id is a $elem.type.",
   "askingPrice" : $elem.price
 }#if($foreach.hasNext),#end
#end
```

```
9 ]
```

在此映射模板中：

1. 在第 1 行上，`$inputRoot` 变量表示上一部分的原始 JSON 数据中的根对象。指令以 # 符号开头。
2. 在第 3 行上，`foreach` 循环遍历原始 JSON 数据中的每个对象。
3. 在第 5 行上，`description` 是原始 JSON 数据中宠物的 `id` 和 `type` 拼接的结果。
4. 在第 6 行上，`askingPrice` 是原始 JSON 数据中的 `price`。

有关 Velocity 模板语言的更多信息，请参阅 Apache Velocity – VTL 参考。有关 JSONPath 的更多信息，请参阅 JSONPath – 适用于 JSON 的 XPath。

该映射模板假定基础数据为 JSON 对象。它不要求为数据定义模型。但是，输出数据的模型允许将前面的数据作为语言特定的对象返回。有关更多信息，请参阅 了解数据模型。

复杂的映射模板

您还可以创建更复杂的映射模板。以下示例显示了引用连接和 100 的截止值，以确定宠物价格是否合适。

输入数据

```
[
  {
    "id": 1,
    "type": "dog",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "cat",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
```

映射模板

```
#set($inputRoot = $input.path('$'))
#set($cheap = 100)
[
#foreach($elem in $inputRoot)
  {
#set($name = "${elem.type}number$elem.id")
    "name" : $name,
    "description" : "Item $elem.id is a $elem.type.",
    #if($elem.price > $cheap )#set ($afford = 'too much!') #{else}#set
 ($afford = $elem.price)#end
"askingPrice" : $afford
  }#if($foreach.hasNext),#end

#end
]
```

输出数据

```
[
  {
    "name" : dognumber1,
    "description" : "Item 1 is a dog.",
    "askingPrice" : too much!
  },
  {
    "name" : catnumber2,
    "description" : "Item 2 is a cat.",
    "askingPrice" : too much!
  },
  {
    "name" : fishnumber3,
    "description" : "Item 3 is a fish.",
    "askingPrice" : 0.99
  }
]
```

您还可查看更复杂的数据模型。请参阅 API Gateway 的示例数据模型和映射模板。

在 API Gateway 中设置数据转换

本节介绍如何设置映射模板，以使用控制台和 AWS CLI 转换集成请求和响应。

主题

- 使用 API Gateway 控制台设置数据转换
- 使用 AWS CLI 设置数据转换
- 已完成的数据转换 AWS CloudFormation 模板
- 后续步骤

使用 API Gateway 控制台设置数据转换

在本教程中，您将使用以下 .zip 文件 data-transformation-tutorial-console.zip 创建不完整的 API 和 DynamoDB 表。这个不完整的 API 拥有的 `/pets` 资源具有 `GET` 和 `POST` 方法。

- `GET` 方法将从 `http://petstore-demo-endpoint.execute-api.com/petstore/pets` HTTP 端点获取数据。输出数据将根据PetStore 映射模板中的映射模板进行转换。
- `POST` 方法将允许用户使用映射模板将宠物信息 `POST` 到 Amazon DynamoDB 表中。

下载并解压缩适用于 AWS CloudFormation 的应用程序创建模板。您将使用此模板创建 DynamoDB 表来发布宠物信息和不完整的 API。您将在 API Gateway 控制台中完成其余步骤。

创建 AWS CloudFormation 堆栈

1. 打开 AWS CloudFormation 控制台，地址：https://console.aws.amazon.com/cloudformation。
2. 选择创建堆栈，然后选择使用新资源(标准)。
3. 对于指定模板，选择上传模板文件。
4. 选择您下载的模板。
5. 选择下一步。
6. 对于堆栈名称，输入 **`data-transformation-tutorial-console`**，然后选择下一步。
7. 对于配置堆栈选项，请选择下一步。
8. 对于功能，请确认 AWS CloudFormation 可以在您的账户中创建 IAM 资源。
9. 选择提交。

AWS CloudFormation 预置在模板中指定的资源。完成资源预置可能需要几分钟时间。当 AWS CloudFormation 堆栈的状态为 CREATE_COMPLETE 时，您就可以继续下一步了。

测试 **GET** 集成响应

1. 在 **data-transformation-tutorial-console** 的 AWS CloudFormation 堆栈的资源选项卡上，选择您的 API 的物理 ID。
2. 在主导航窗格中，选择资源，然后选择 GET 方法。
3. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。

   测试的输出将显示以下内容：

   ```
   [
     {
       "id": 1,
       "type": "dog",
       "price": 249.99
     },
     {
       "id": 2,
       "type": "cat",
       "price": 124.99
     },
     {
       "id": 3,
       "type": "fish",
       "price": 0.99
     }
   ]
   ```

   您将根据PetStore 映射模板中的映射模板转换此输出。

转换 **GET** 集成响应

1. 选择集成响应选项卡。

   目前，没有定义任何映射模板，因此不会转换集成响应。
2. 对于默认 - 响应，选择编辑。
3. 选择映射模板，然后执行以下操作：

   a. 选择添加映射模板。

   b. 对于内容类型，输入 **application/json**。

c. 对于模板正文，输入以下内容：

```
#set($inputRoot = $input.path('$'))
[
#foreach($elem in $inputRoot)
  {
    "description" : "Item $elem.id is a $elem.type.",
    "askingPrice" : $elem.price
  }#if($foreach.hasNext),#end

#end
]
```

选择保存。

测试 **GET** 集成响应

- 选择测试选项卡，然后选择测试。

  测试的输出将显示转换后的响应。

```
[
  {
    "description" : "Item 1 is a dog.",
    "askingPrice" : 249.99
  },
  {
    "description" : "Item 2 is a cat.",
    "askingPrice" : 124.99
  },
  {
    "description" : "Item 3 is a fish.",
    "askingPrice" : 0.99
  }
]
```

转换来自 **POST** 方法的输入数据

1. 选择 POST 方法。

2. 选择集成请求选项卡，然后对于集成请求设置，选择编辑。

   AWS CloudFormation 模板已填充了一些集成请求字段。

   - 集成类型为 AWS 服务。
   - AWS 服务是 DynamoDB。
   - HTTP 方法为 `POST`。
   - 操作是 `PutItem`。
   - 允许 API Gateway 将项目放入 DynamoDB 表的执行角色是 `data-transformation-tutorial-console-APIGatewayRole`。AWS CloudFormation 创建此角色以允许 API Gateway 拥有与 DynamoDB 交互的最低权限。

   尚未指定 DynamoDB 表的名称。您将在以下步骤中指定名称。

3. 对于请求正文传递，选择从不。

   这意味着 API 将拒绝其 Content-Type（内容类型）没有映射模板的数据。

4. 选择映射模板。
5. 内容类型设置为 `application/json`。这意味着 API 将拒绝所有不是 application/json 的内容类型。有关集成传递行为的更多信息，请参阅集成传递行为
6. 在文本编辑器中输入以下代码。

```
{
    "TableName":"data-transformation-tutorial-console-ddb",
    "Item": {
        "id": {
            "N": $input.json("$.id")
        },
        "type": {
            "S": $input.json("$.type")
        },
        "price": {
            "N": $input.json("$.price")
        }
    }
}
```

此模板将表指定为 `data-transformation-tutorial-console-ddb` 并将项目设置为 `id`、`type` 和 `price`。这些项目将来自 POST 方法的正文。您也可以使用数据模型来帮助创建映射模板。有关更多信息，请参阅 在 API Gateway 中使用请求验证。

7. 选择保存以保存映射模板。

从 **POST** 方法添加方法和集成响应

AWS CloudFormation 创建了一个空白方法和集成响应。您将编辑此回复以提供更多信息。有关如何编辑响应的更多信息，请参阅Amazon API Gateway API 请求和响应数据映射参考。

1. 在集成响应选项卡上，对于默认 - 响应，选择编辑。
2. 选择映射模板，然后选择添加映射模板。
3. 对于 Content-Type，输入 **application/json**。
4. 在代码编辑器中，输入以下输出映射模板以发送输出消息：

```
{ "message" : "Your response was recorded at $context.requestTime" }
```

   有关上下文变量的更多信息，请参阅适用于数据模型、授权方、映射模板和 CloudWatch 访问日志记录的 `$context` 变量。

5. 选择保存以保存映射模板。

测试 **POST** 方法

选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。

1. 在请求正文中，输入以下示例。

```
{
          "id": "4",
          "type" : "dog",
          "price": "321"
}
```

2. 选择测试。

   输出应显示您的成功消息。

您可以通过 https://console.aws.amazon.com/dynamodb/ 打开 DynamoDB 控制台，以验证示例项目是否在您的表中。

删除 AWS CloudFormation 堆栈

1. 打开 AWS CloudFormation 控制台，地址：https://console.aws.amazon.com/cloudformation。
2. 选择您的 AWS CloudFormation 堆栈。
3. 选择删除，然后确认您的选择。

使用 AWS CLI 设置数据转换

在本教程中，您将使用以下 .zip 文件 data-transformation-tutorial-cli.zip 创建不完整的 API 和 DynamoDB 表。这个不完整的 API 所具有的 `/pets` 资源包含与 `http://petstore-demo-endpoint.execute-api.com/petstore/pets` HTTP 端点集成的 GET 方法。您将创建 `POST` 方法以连接到 DynamoDB 表，并使用映射模板将数据输入到 DynamoDB 表中。

- 您将根据PetStore 映射模板中的映射模板转换输出数据。
- 您将创建 `POST` 方法，以允许用户使用映射模板将宠物信息 `POST` 到 Amazon DynamoDB 表中。

创建 AWS CloudFormation 堆栈

下载并解压缩适用于 AWS CloudFormation 的应用程序创建模板。

要完成以下教程，您需要 AWS Command Line Interface（AWS CLI）版本 2。

对于长命令，使用转义字符 (\) 将命令拆分为多行。

> **Note**
>
> 在 Windows 中，操作系统的内置终端不支持您经常使用的某些 Bash CLI 命令（例如 `zip`）。安装 Windows Subsystem for Linux，获取 Ubuntu 和 Bash 与 Windows 集成的版本。本指南中的示例 CLI 命令使用 Linux 格式。如果您使用的是 Windows CLI，则必须重新格式化包含内联 JSON 文档的命令。

1. 输入以下命令创建 AWS CloudFormation 堆栈。

```
aws cloudformation create-stack --stack-name data-transformation-tutorial-cli
 --template-body file://data-transformation-tutorial-cli.zip --capabilities
 CAPABILITY_NAMED_IAM
```

2. AWS CloudFormation 预置在模板中指定的资源。完成资源预置可能需要几分钟时间。使用以下命令查看 AWS CloudFormation 的状态。

   ```
   aws cloudformation describe-stacks --stack-name data-transformation-tutorial-cli
   ```

3. 当 AWS CloudFormation 堆栈的状态为 `StackStatus: "CREATE_COMPLETE"` 时，使用以下命令检索将来步骤的相关输出值。

   ```
    aws cloudformation describe-stacks --stack-name data-transformation-tutorial-cli
    --query "Stacks[*].Outputs[*].{OutputKey: OutputKey, OutputValue: OutputValue,
    Description: Description}"
   ```

   输出值包括：

   - ApiRole，这是允许 API Gateway 在 DynamoDB 表中放置项目的角色名称。对于本教程，角色名称为 `data-transformation-tutorial-cli-APIGatewayRole-ABCDEFG`。
   - DDBTableName，这是 DynamoDB 表的名称。对于本教程，表名称为 `data-transformation-tutorial-cli-ddb`
   - ResourceId，这是公开 `GET` 和 `POST` 方法的宠物资源的 ID。对于本教程，资源 ID 为 `efg456`
   - ApiId，这是 API 的 ID。对于本教程，API ID 为 `abc123`。

在数据转换之前测试 GET 方法

- 使用以下命令测试 `GET` 方法。

  ```
  aws apigateway test-invoke-method --rest-api-id abc123 \
            --resource-id efg456 \
            --http-method GET
  ```

  测试的输出将显示以下内容。

  ```
  [
    {
      "id": 1,
  ```

```
    "type": "dog",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "cat",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
```

您将根据PetStore 映射模板中的映射模板转换此输出。

转换 **GET** 集成响应

- 使用以下命令更新 GET 方法的集成响应。将 *rest-api-id* 和 *resource-id* 替换为您的值。

  使用以下命令创建集成响应。

  ```
  aws apigateway put-integration-response --rest-api-id abc123 \
    --resource-id efg456 \
    --http-method GET \
    --status-code 200 \
    --selection-pattern "" \
    --response-templates '{"application/json": "#set($inputRoot = $input.path(\"$
  \"))\n[\n#foreach($elem in $inputRoot)\n {\n  \"description\": \"Item $elem.id is a
   $elem.type\",\n  \"askingPrice\": \"$elem.price\"\n }#if($foreach.hasNext),#end\n
  \n#end\n]"}'
  ```

测试 **GET** 方法

- 使用以下命令测试 GET 方法。

  ```
  aws apigateway test-invoke-method --rest-api-id abc123 \
    --resource-id efg456 \
    --http-method GET \
  ```

测试的输出将显示转换后的响应。

```
[
  {
    "description" : "Item 1 is a dog.",
    "askingPrice" : 249.99
  },
  {
    "description" : "Item 2 is a cat.",
    "askingPrice" : 124.99
  },
  {
    "description" : "Item 3 is a fish.",
    "askingPrice" : 0.99
  }
]
```

创建 **POST** 方法

1. 使用以下命令在您的 `/pets` 资源上创建新的方法。

```
aws apigateway put-method --rest-api-id abc123 \
  --resource-id efg456 \
  --http-method POST \
  --authorization-type "NONE" \
```

此方法允许您将宠物信息发送到您在 AWS CloudFormation 堆栈中创建的 DynamoDB 表。

2. 使用以下命令在 POST 方法上创建 AWS 服务 集成。

```
aws apigateway put-integration --rest-api-id abc123 \
  --resource-id efg456 \
  --http-method POST \
  --type AWS \
  --integration-http-method POST \
  --uri "arn:aws:apigateway:us-east-2:dynamodb:action/PutItem" \
  --credentials arn:aws:iam::111122223333:role/data-transformation-tutorial-cli-
APIGatewayRole-ABCDEFG \
```

```
  --request-templates '{"application/json":"{\"TableName\":\"data-transformation-
tutorial-cli-ddb\",\"Item\":{\"id\":{\"N\":$input.json(\"$.id\")},\"type\":{\"S\":
$input.json(\"$.type\")},\"price\":{\"N\":$input.json(\"$.price\")} }}"}'
```

3. 使用以下命令为成功调用 POST 方法创建方法响应。

```
aws apigateway put-method-response --rest-api-id abc123 \
  --resource-id efg456 \
  --http-method POST \
  --status-code 200
```

4. 使用以下命令为成功调用 POST 方法创建集成响应。

```
aws apigateway put-integration-response --rest-api-id abc123 \
  --resource-id efg456 \
  --http-method POST \
  --status-code 200 \
  --selection-pattern "" \
  --response-templates '{"application/json": "{\"message\": \"Your response was
 recorded at $context.requestTime\"}"}'
```

测试 **POST** 方法

- 使用以下命令测试 POST 方法。

```
aws apigateway test-invoke-method --rest-api-id abc123 \
  --resource-id efg456 \
  --http-method POST \
  --body '{\"id\": \"4\", \"type\": \"dog\", \"price\": \"321\"}'
```

输出将显示成功消息。

删除 AWS CloudFormation 堆栈

- 使用以下命令删除您的 AWS CloudFormation 资源。

```
aws cloudformation delete-stack  --stack-name data-transformation-tutorial-cli
```

已完成的数据转换 AWS CloudFormation 模板

以下示例是一个已完成的 AWS CloudFormation 模板，它创建了一个 API 和一个 DynamoDB 表，该表的 `/pets` 资源具有 `GET` 和 `POST` 方法。

- `GET` 方法将从 `http://petstore-demo-endpoint.execute-api.com/petstore/pets` HTTP 端点获取数据。输出数据将根据PetStore 映射模板中的映射模板进行转换。
- `POST` 方法将允许用户使用映射模板将宠物信息 `POST` 到 DynamoDB 表中。

```
AWSTemplateFormatVersion: 2010-09-09
Description: A completed Amazon API Gateway REST API that uses non-proxy integration to POST to an Amazon DynamoDB table and non-proxy integration to GET transformed pets data.
Parameters:
  StageName:
    Type: String
    Default: v1
    Description: Name of API stage.
Resources:
  DynamoDBTable:
    Type: 'AWS::DynamoDB::Table'
    Properties:
      TableName: !Sub data-transformation-tutorial-complete
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: N
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      ProvisionedThroughput:
        ReadCapacityUnits: 5
        WriteCapacityUnits: 5
  APIGatewayRole:
    Type: 'AWS::IAM::Role'
    Properties:
      AssumeRolePolicyDocument:
        Version: 2012-10-17
        Statement:
          - Action:
              - 'sts:AssumeRole'
            Effect: Allow
            Principal:
```

```
              Service:
                - apigateway.amazonaws.com
      Policies:
        - PolicyName: APIGatewayDynamoDBPolicy
          PolicyDocument:
            Version: 2012-10-17
            Statement:
              - Effect: Allow
                Action:
                  - 'dynamodb:PutItem'
                Resource: !GetAtt DynamoDBTable.Arn
  Api:
    Type: 'AWS::ApiGateway::RestApi'
    Properties:
      Name: data-transformation-complete-api
      ApiKeySourceType: HEADER
  PetsResource:
    Type: 'AWS::ApiGateway::Resource'
    Properties:
      RestApiId: !Ref Api
      ParentId: !GetAtt Api.RootResourceId
      PathPart: 'pets'
  PetsMethodGet:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref PetsResource
      HttpMethod: GET
      ApiKeyRequired: false
      AuthorizationType: NONE
      Integration:
        Type: HTTP
        Credentials: !GetAtt APIGatewayRole.Arn
        IntegrationHttpMethod: GET
        Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets/
        PassthroughBehavior: WHEN_NO_TEMPLATES
        IntegrationResponses:
          - StatusCode: '200'
            ResponseTemplates:
              application/json: "#set($inputRoot = $input.path(\"$
\"))\n[\n#foreach($elem in $inputRoot)\n {\n  \"description\": \"Item $elem.id is a
 $elem.type\",\n  \"askingPrice\": \"$elem.price\"\n }#if($foreach.hasNext),#end\n
\n#end\n]"
      MethodResponses:
```

```
        - StatusCode: '200'
  PetsMethodPost:
    Type: 'AWS::ApiGateway::Method'
    Properties:
      RestApiId: !Ref Api
      ResourceId: !Ref PetsResource
      HttpMethod: POST
      ApiKeyRequired: false
      AuthorizationType: NONE
      Integration:
        Type: AWS
        Credentials: !GetAtt APIGatewayRole.Arn
        IntegrationHttpMethod: POST
        Uri: arn:aws:apigateway:us-west-1:dynamodb:action/PutItem
        PassthroughBehavior: NEVER
        RequestTemplates:
          application/json: "{\"TableName\":\"data-transformation-tutorial-complete
\",\"Item\":{\"id\":{\"N\":$input.json(\"$.id\")},\"type\":{\"S\":$input.json(\"$.type
\")},\"price\":{\"N\":$input.json(\"$.price\")} }}"
        IntegrationResponses:
          - StatusCode: 200
            ResponseTemplates:
              application/json: "{\"message\": \"Your response was recorded at
 $context.requestTime\"}"
      MethodResponses:
        - StatusCode: '200'

  ApiDeployment:
    Type: 'AWS::ApiGateway::Deployment'
    DependsOn:
      - PetsMethodGet
    Properties:
      RestApiId: !Ref Api
      StageName: !Sub '${StageName}'
Outputs:
  ApiId:
    Description: API ID for CLI commands
    Value: !Ref Api
  ResourceId:
    Description: /pets resource ID for CLI commands
    Value: !Ref PetsResource
  ApiRole:
    Description: Role ID to allow API Gateway to put and scan items in DynamoDB table
    Value: !Ref APIGatewayRole
```

```
  DDBTableName:
    Description: DynamoDB table name
    Value: !Ref DynamoDBTable
```

后续步骤

要了解更复杂的映射模板，请参阅以下示例：

- 使用示例相册相册示例查看更复杂的模型和映射模板。
- 有关模型的更多信息，请参阅了解数据模型。
- 有关如何映射不同的响应代码输出的信息，请访问使用 API Gateway 控制台设置请求和响应数据映射。
- 有关如何从 API 的方法请求数据设置数据映射的信息，请访问API Gateway 映射模板和访问日志记录变量引用。

使用映射模板覆盖 API 的请求和响应参数以及状态代码

利用标准 API Gateway 参数和响应代码映射模板，您可以一对一地映射参数，并将一系列集成响应状态代码（由正则表达式匹配）映射到单个响应状态代码。映射模板覆盖使您能够灵活地执行多对一参数映射；在应用标准 API Gateway 映射后覆盖参数；根据正文内容或其他参数值有条件地映射参数；通过编程方式动态地创建新的参数；并覆盖由集成端点返回的状态代码。可覆盖任何类型的请求参数、响应标头或响应状态代码。

以下是用于映射模板覆盖的示例：

- 创建新的标头（或覆盖现有标头）作为两个参数的联接
- 根据正文内容覆盖响应代码以获得成功或失败代码
- 根据一个参数的内容或其他参数的内容有条件地重新映射该参数
- 循环访问 JSON 正文的内容并将密钥值对重新映射到标头或查询字符串

要创建映射模板覆盖，请映射模板中一个或多个以下的 `$context` 变量：

请求正文映射模板	响应正文映射模板
`$context.requestOverride.header.` *`header_name`*	`$context.responseOverride.header.` *`header_name`*

请求正文映射模板	响应正文映射模板
`$context.requestOverride.path.` *`path_name`*	`$context.responseOverride.status`
`$context.requestOverride.querystring.` *`querystring_name`*	

Note

映射模板覆盖不能与代理集成端点（它们缺少数据映射）结合使用。有关集成类型的更多信息，请参阅选择 API Gateway API 集成类型。

Important

覆盖是最终的。对于每个参数，覆盖只能应用一次。多次尝试覆盖同一个参数将导致来自 Amazon API Gateway 的 5XX 响应。如果您必须在整个模板中多次覆盖相同的参数，我们建议创建一个变量并在模板末尾应用覆盖。请注意，仅在解析整个模板后应用模板。请参阅 教程：使用 API Gateway 控制台覆盖 API 的请求参数和标头。

以下教程介绍如何在 API Gateway 控制台中创建和测试映射模板覆盖。这些教程使用 PetStore 示例 API 作为起点。这两个教程都假定您已创建 PetStore 示例 API。

主题

- 教程：使用 API Gateway 控制台覆盖 API 的响应状态代码
- 教程：使用 API Gateway 控制台覆盖 API 的请求参数和标头
- 示例：使用 API Gateway CLI 覆盖 API 的请求参数和标头
- 示例：使用适用于 JavaScript 的开发工具包覆盖 API 的请求参数和标头

教程：使用 API Gateway 控制台覆盖 API 的响应状态代码

要使用 PetStore 示例 API 检索宠物，您使用 `GET /pets/{petId}` 的 API 方法请求，其中 `{petId}` 是可在运行时获取数字的路径参数。

在本教程中，您将通过创建映射模板（此模板在检测到错误条件时，将 `$context.responseOverride.status` 映射到 `400`）来覆盖此 `GET` 方法的响应代码。

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在 API 下，选择 PetStore API，然后选择资源。
3. 在资源树中，选择 `/{petId}` 下的 `GET` 方法。
4. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。
5. 对于 petId，输入 **`-1`**，然后选择测试。

   在结果中，您会发现两件事：

   首先，响应正文指示超出范围错误：

   ```
   {
     "errors": [
       {
         "key": "GetPetRequest.petId",
         "message": "The value is out of range."
       }
     ]
   }
   ```

   其次，日志框下的最后一行的结尾为：`Method completed with status: 200`。
6. 在集成响应选项卡上，对于默认 - 响应，选择编辑。
7. 选择映射模板。
8. 选择添加映射模板。
9. 对于内容类型，输入 **`application/json`**。
10. 对于模板正文，输入以下内容：

    ```
    #set($inputRoot = $input.path('$'))
    $input.json("$")
    #if($inputRoot.toString().contains("error"))
    #set($context.responseOverride.status = 400)
    #end
    ```
11. 选择保存。
12. 选择测试选项卡。

13. 对于 petId，输入 **`-1`**。
14. 在结果中，响应正文表示超出范围错误：

    ```
    {
      "errors": [
        {
          "key": "GetPetRequest.petId",
          "message": "The value is out of range."
        }
      ]
    }
    ```

    不过，日志框下的最后一行现在的结尾为：`Method completed with status: 400`。

教程：使用 API Gateway 控制台覆盖 API 的请求参数和标头

在本教程中，您将通过创建映射模板（此模板将 `$context.requestOverride.header.`*`header_name`* 映射到一个组合其他两个标头的新标头）来覆盖 `GET` 方法的请求标头。

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在 API 下，选择 PetStore API。
3. 在资源树中，选择 `/pet` 下的 `GET` 方法。
4. 在方法请求选项卡上，对于方法请求设置，选择编辑。
5. 选择 HTTP 请求标头，然后选择添加标头。
6. 对于名称，请输入 **`header1`**。
7. 选择添加标头，然后创建第二个标头，名为 **`header2`**。
8. 选择保存。
9. 在集成请求选项卡上，对于集成请求设置，选择编辑。
10. 对于请求正文传递，选择当未定义模板时（推荐）。
11. 选择映射模板，然后执行以下操作：
    a. 选择添加映射模板。
    b. 对于内容类型，输入 **`application/json`**。
    c. 对于模板正文，输入以下内容：

```
#set($header1Override = "foo")
#set($header3Value = "$input.params('header1')$input.params('header2')")
$input.json("$")
#set($context.requestOverride.header.header3 = $header3Value)
#set($context.requestOverride.header.header1 = $header1Override)
#set($context.requestOverride.header.multivalueheader=[$header1Override,
 $header3Value])
```

12. 选择保存。
13. 选择测试选项卡。
14. 在 {pets} 的标头下，复制以下代码：

```
header1:header1Val
header2:header2Val
```

15. 选择 Test (测试)。

    在日志中，您应看到一个包含以下文本的条目：

```
Endpoint request headers: {header3=header1Valheader2Val,
header2=header2Val, header1=foo, x-amzn-apigateway-api-id=<api-id>,
Accept=application/json, multivalueheader=foo,header1Valheader2Val}
```

示例：使用 API Gateway CLI 覆盖 API 的请求参数和标头

以下 CLI 示例说明如何使用 `put-integration` 命令来覆盖响应代码：

```
aws apigateway put-integration --rest-api-id <API_ID> --resource-
id <PATH_TO_RESOURCE_ID> --http-method <METHOD>
            --type <INTEGRATION_TYPE> --request-templates <REQUEST_TEMPLATE_MAP>
```

其中，*`<REQUEST_TEMPLATE_MAP>`* 是从内容类型到要应用的模板字符串的映射。此映射的结构如下所示：

```
Content_type1=template_string,Content_type2=template_string
```

或，在 JSON 语法中：

```
{"content_type1": "template_string"
```

```
...}
```

以下示例说明如何使用 `put-integration-response` 命令来覆盖 API 的响应代码：

```
aws apigateway put-integration-response --rest-api-id <API_ID> --resource-
id <PATH_TO_RESOURCE_ID> --http-method <METHOD>
            --status-code <STATUS_CODE> --response-templates <RESPONSE_TEMPLATE_MAP>
```

其中，*<RESPONSE_TEMPLATE_MAP>* 与上述的*<REQUEST_TEMPLATE_MAP>* 具有相同格式。

示例：使用适用于 JavaScript 的开发工具包覆盖 API 的请求参数和标头

以下示例说明如何使用 `put-integration` 命令来覆盖响应代码：

请求:

```
var params = {
  httpMethod: 'STRING_VALUE', /* required */
  resourceId: 'STRING_VALUE', /* required */
  restApiId: 'STRING_VALUE', /* required */
  type: HTTP | AWS | MOCK | HTTP_PROXY | AWS_PROXY, /* required */
  requestTemplates: {
    '<Content_type>': 'TEMPLATE_STRING',
    /* '<String>': ... */
  },
};
apigateway.putIntegration(params, function(err, data) {
  if (err) console.log(err, err.stack); // an error occurred
  else     console.log(data);           // successful response
});
```

响应：

```
var params = {
  httpMethod: 'STRING_VALUE', /* required */
  resourceId: 'STRING_VALUE', /* required */
  restApiId: 'STRING_VALUE', /* required */
  statusCode: 'STRING_VALUE', /* required */
  responseTemplates: {
    '<Content_type>': 'TEMPLATE_STRING',
    /* '<String>': ... */
```

```
  },
};
apigateway.putIntegrationResponse(params, function(err, data) {
  if (err) console.log(err, err.stack); // an error occurred
  else     console.log(data);           // successful response
});
```

使用 API Gateway 控制台设置请求和响应数据映射

要使用 API Gateway 控制台定义 API 的集成请求/响应，请按照以下说明操作。

> Note
>
> 这些说明假设您已完成 使用 API Gateway 控制台设置 API 集成请求中的步骤。

1. 在资源窗格中，选择您的方法。
2. 在集成请求选项卡的集成请求设置下，选择编辑。
3. 为请求正文传递选择一个选项，以配置如何将未映射内容类型的方法请求正文通过集成请求传递至 Lambda 函数、HTTP 代理或 AWS 服务代理而不进行转换。这里有三个选项：

   - 如果您希望在方法请求内容类型不匹配任何与映射模板关联的内容类型时，将方法请求正文通过集成请求发送到后端而不进行转换（如下一步所定义），则选择当没有模板匹配请求的 Content-Type 标头时。

     > Note
     >
     > 调用 API Gateway API 时，您通过将 `WHEN_NO_MATCH` 设置为集成资源的 `passthroughBehavior` 属性值来选择此选项。

   - 如果您希望在集成请求中未定义映射模板时，将方法请求正文通过集成请求发送到后端而不进行转换，则选择当未定义模板时(推荐)。如果选择此选项时定义了模板，则会以“HTTP 415 Unsupported Media Type”响应拒绝未映射内容类型的方法请求。

     > Note
     >
     > 调用 API Gateway API 时，您通过将 `WHEN_NO_TEMPLATE` 设置为集成资源的 `passthroughBehavior` 属性值来选择此选项。

- 如果您不希望在任一方法请求内容类型与集成请求中定义的映射模板关联的任何内容类型均不匹配时或者集成请求中未定义映射模板时传递方法请求，则选择永不。将以“HTTP 415 Unsupported Media Type”响应拒绝未映射内容类型的方法请求。

> **Note**
>
> 调用 API Gateway API 时，您通过将 `NEVER` 设置为集成资源的 `passthroughBehavior` 属性值来选择此选项。

有关集成传递行为的更多信息，请参阅集成传递行为。

4. 对于 HTTP 代理或 AWS 服务代理，要将集成请求中定义的路径参数、查询字符串参数或标头参数与 HTTP 代理或 AWS 服务代理的方法请求中对应的路径参数、查询字符串参数或标头参数进行关联，请执行以下操作：

   a. 分别选择 URL 路径参数、URL 查询字符串参数或 HTTP 请求标头，然后分别选择添加路径、添加查询字符串或添加标头。

   b. 对于名称，键入 HTTP 代理或 AWS 服务代理中的路径参数、查询字符串参数或标头参数的名称。

   c. 对于映射自，输入路径参数、查询字符串参数或标头参数的映射值。使用以下格式之一：

      - **`method.request.path.`*`parameter-name`*** 用于名为 *`parameter-name`* 的路径参数，如方法请求页面所定义。
      - **`method.request.querystring.`*`parameter-name`*** 用于名为 *`parameter-name`* 的查询字符串参数，如方法请求页面所定义。
      - **`method.request.multivaluequerystring.`*`parameter-name`*** 用于名为 *`parameter-name`* 的多值查询字符串参数，如方法请求页面所定义。
      - **`method.request.header.`*`parameter-name`*** 用于名为 *`parameter-name`* 的标头参数，如方法请求页面所定义。

        或者，您可以将一个文字字符串值（用单引号引起）设置为集成标头。
      - **`method.request.multivalueheader.`*`parameter-name`*** 用于名为 *`parameter-name`* 的多值标头参数，如方法请求页面所定义。

   d. 要添加其他参数，请选择添加按钮。

5. 要添加映射模板，请选择映射模板。

6. 要定义传入请求的映射模板，请选择添加映射模板。对于内容类型，输入一个内容类型（例如 **`application/json`**）。然后，输入映射模板。有关更多信息，请参阅 了解映射模板。
7. 选择保存。
8. 您可以将集成响应从后端映射到返回给调用应用程序的 API 的方法响应。这包括向客户端返回从后端可用的响应标头中选择的响应标头，从而将后端响应负载的数据格式转换为 API 指定的格式。您可以配置方法响应和集成响应来指定此类映射。

   要让该方法基于 Lambda 函数、HTTP 代理或 AWS 服务代理返回的 HTTP 状态代码接收自定义响应数据格式，请执行以下操作：

   a. 选择集成响应。选择默认 - 响应上的编辑，以便为方法中的 200 HTTP 响应代码指定设置，或选择创建响应，以便为方法中的任何其他 HTTP 响应状态代码指定设置。
   b. 对于 Lambda 错误正则表达式（针对 Lambda 函数）或 HTTP 状态正则表达式（针对 HTTP 代理或 AWS 服务代理），输入正则表达式以指定哪些 Lambda 函数错误字符串（针对 Lambda 函数）或 HTTP 响应状态代码（针对 HTTP 代理或 AWS 服务代理）映射到此输出映射。例如，要将所有 2xx HTTP 响应状态代码从 HTTP 代理映射到此输出映射，请为 HTTP 状态正则表达式键入“**`2\d{2}`**”。要对来自 Lambda 函数的包含“无效请求”的错误消息返回 `400 Bad Request` 响应，请输入“**`.*Invalid request.*`**”作为 Lambda 错误正则表达式。另一方面，要对来自 Lambda 的所有未映射错误消息返回 `400 Bad Request`，请在 Lambda 错误正则表达式中输入“**`(\n|.)+`**”。这最后一个正则表达式可用于 API 的默认错误响应。

      Note

      API Gateway 使用 Java 模式的正则表达式来响应映射。有关更多信息，请参阅 Oracle 文档中的模式。
      错误模式与 Lambda 响应中的 `errorMessage` 属性（在 Node.js 中由 `callback(errorMessage)` 填充，或在 Java 中由 `throw new MyException(errorMessage)` 填充）的整个字符串匹配。此外，在应用正则表达式之前，转义字符将不会转义。
      如果您使用“.+”作为选择模式来筛选响应，请注意，它可能与包含换行符 ("\n") 的响应不匹配。

   c. 如果启用，对于方法响应状态，选择您在方法响应页面上定义的 HTTP 响应状态代码。
   d. 对于标头映射，针对您在方法响应页面上为 HTTP 响应状态代码定义的每个标头，指定映射值。对于映射值，请使用以下格式之一：

    - **`integration.response.multivalueheaders.`*`header-name`*** 格式，其中 *`header-name`* 是来自后端的多值响应标头的名称。

      例如，要将后端响应的 `Date` 标头作为 API 方法响应的 `Timestamp` 标头返回，响应标头列将包含一个时间戳条目并且关联的映射值应设置为 integration.response.multivalueheaders.Date。
    - **`integration.response.header.`*`header-name`*** 格式，其中 *`header-name`* 是来自后端的多值响应标头的名称。

      例如，要将后端响应的 `Date` 标头作为 API 方法响应的 `Timestamp` 标头返回，响应标头列将包含一个时间戳条目并且关联的映射值应设置为 integration.response.header.Date。
  e. 选择映射模板，然后选择添加映射模板。在内容类型框中，输入将从 Lambda 函数、HTTP 代理或 AWS 服务代理传递至该方法的数据的内容类型。然后，输入映射模板。有关更多信息，请参阅 了解映射模板。
  f. 选择保存。

API Gateway 的示例数据模型和映射模板

以下部分提供 API Gateway 中可用作您自己的 API 的起点的模型和映射模板的示例。有关数据转换的更多信息，请参阅了解映射模板。有关数据模型的更多信息，请参阅the section called “了解数据模型”。

主题

- 相册示例
- 新闻文章示例

相册示例

以下示例显示了 API Gateway 中的相册 API。我们提供了示例数据转换、其他模型和映射模板。

主题

- 数据转换示例
- 照片数据的输入模型
- 照片数据的输出模型

- 照片数据的输入映射模板

数据转换示例

以下示例说明如何使用 Velocity 模板语言（VTL）映射模板转换有关照片的输入数据。有关 Velocity 模板语言的更多信息，请参阅 Apache Velocity – VTL 参考。

输入数据

```
{
  "photos": {
    "page": 1,
    "pages": "1234",
    "perpage": 100,
    "total": "123398",
    "photo": [
      {
        "id": "12345678901",
        "owner": "23456789@A12",
        "photographer_first_name" : "Saanvi",
        "photographer_last_name" : "Sarkar",
        "secret": "abc123d456",
        "server": "1234",
        "farm": 1,
        "title": "Sample photo 1",
        "ispublic": true,
        "isfriend": false,
        "isfamily": false
      },
      {
        "id": "23456789012",
        "owner": "34567890@B23",
        "photographer_first_name" : "Richard",
        "photographer_last_name" : "Roe",
        "secret": "bcd234e567",
        "server": "2345",
        "farm": 2,
        "title": "Sample photo 2",
        "ispublic": true,
        "isfriend": false,
        "isfamily": false
      }
    ]
  }
```

```
}
```

输出映射模板

```
#set($inputRoot = $input.path('$'))
{
  "photos": [
#foreach($elem in $inputRoot.photos.photo)
    {
      "id": "$elem.id",
      "photographedBy": "$elem.photographer_first_name $elem.photographer_last_name",
      "title": "$elem.title",
      "ispublic": $elem.ispublic,
      "isfriend": $elem.isfriend,
      "isfamily": $elem.isfamily
    }#if($foreach.hasNext),#end

#end
  ]
}
```

输出数据

```
{
  "photos": [
    {
      "id": "12345678901",
      "photographedBy": "Saanvi Sarkar",
      "title": "Sample photo 1",
      "ispublic": true,
      "isfriend": false,
      "isfamily": false
    },
    {
      "id": "23456789012",
      "photographedBy": "Richard Roe",
      "title": "Sample photo 2",
      "ispublic": true,
      "isfriend": false,
      "isfamily": false
    }
  ]
}
```

照片数据的输入模型

您可以为输入数据定义模型。此输入模型要求您上传一张照片，并且对于每页至少指定 10 张照片。您可以使用此输入模型生成 SDK 或为您的 API 开启请求验证。

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "PhotosInputModel",
  "type": "object",
  "properties": {
    "photos": {
      "type": "object",
      "required" : [
      "photo"
      ],
      "properties": {
        "page": { "type": "integer" },
        "pages": { "type": "string" },
        "perpage": { "type": "integer", "minimum" : 10 },
        "total": { "type": "string" },
        "photo": {
          "type": "array",
          "items": {
            "type": "object",
            "properties": {
              "id": { "type": "string" },
              "owner": { "type": "string" },
              "photographer_first_name" : {"type" : "string"},
              "photographer_last_name" : {"type" : "string"},
              "secret": { "type": "string" },
              "server": { "type": "string" },
              "farm": { "type": "integer" },
              "title": { "type": "string" },
              "ispublic": { "type": "boolean" },
              "isfriend": { "type": "boolean" },
              "isfamily": { "type": "boolean" }
            }
          }
        }
      }
    }
  }
}
```

照片数据的输出模型

您可以为输出数据定义模型。您可以将此模型用于方法响应模型，这在为 API 生成强类型 SDK 时是必需的。这会导致输出将在 Java 或 Objective-C 中强制转换为适当的类。

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "PhotosOutputModel",
  "type": "object",
  "properties": {
    "photos": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id": { "type": "string" },
          "photographedBy": { "type": "string" },
          "title": { "type": "string" },
          "ispublic": { "type": "boolean" },
          "isfriend": { "type": "boolean" },
          "isfamily": { "type": "boolean" }
        }
      }
    }
  }
}
```

照片数据的输入映射模板

您可以定义映射模板来修改输入数据。您可以修改输入数据以进行进一步的函数集成或集成响应。

```
#set($inputRoot = $input.path('$'))
{
  "photos": {
    "page": $inputRoot.photos.page,
    "pages": "$inputRoot.photos.pages",
    "perpage": $inputRoot.photos.perpage,
    "total": "$inputRoot.photos.total",
    "photo": [
#foreach($elem in $inputRoot.photos.photo)
      {
        "id": "$elem.id",
        "owner": "$elem.owner",
```

```
        "photographer_first_name" : "$elem.photographer_first_name",
        "photographer_last_name" : "$elem.photographer_last_name",
        "secret": "$elem.secret",
        "server": "$elem.server",
        "farm": $elem.farm,
        "title": "$elem.title",
        "ispublic": $elem.ispublic,
        "isfriend": $elem.isfriend,
        "isfamily": $elem.isfamily
      }#if($foreach.hasNext),#end

#end
    ]
  }
}
```

新闻文章示例

以下示例显示了 API Gateway 中的新闻文章 API。我们提供了示例数据转换、其他模型和映射模板。

主题

- 数据转换示例
- 新闻数据的输入模型
- 新闻数据的输出模型
- 新闻数据的输入映射模板

数据转换示例

以下示例说明如何使用 Velocity 模板语言（VTL）映射模板转换有关新闻文章的输入数据。有关 Velocity 模板语言的更多信息，请参阅 Apache Velocity – VTL 参考。

输入数据

```
{
  "count": 1,
  "items": [
    {
      "last_updated_date": "2015-04-24",
      "expire_date": "2016-04-25",
      "author_first_name": "John",
      "description": "Sample Description",
      "creation_date": "2015-04-20",
```

```
      "title": "Sample Title",
      "allow_comment": true,
      "author": {
        "last_name": "Doe",
        "email": "johndoe@example.com",
        "first_name": "John"
      },
      "body": "Sample Body",
      "publish_date": "2015-04-25",
      "version": "1",
      "author_last_name": "Doe",
      "parent_id": 2345678901,
      "article_url": "http://www.example.com/articles/3456789012"
    }
  ],
  "version": 1
}
```

输出映射模板

```
#set($inputRoot = $input.path('$'))
{
  "count": $inputRoot.count,
  "items": [
#foreach($elem in $inputRoot.items)
    {
      "creation_date": "$elem.creation_date",
      "title": "$elem.title",
      "author": "$elem.author.first_name $elem.author.last_name",
      "body": "$elem.body",
      "publish_date": "$elem.publish_date",
      "article_url": "$elem.article_url"
    }#if($foreach.hasNext),#end

#end
  ],
  "version": $inputRoot.version
}
```

输出数据

```
{
  "count": 1,
  "items": [
    {
      "creation_date": "2015-04-20",
      "title": "Sample Title",
      "author": "John Doe",
      "body": "Sample Body",
      "publish_date": "2015-04-25",
      "article_url": "http://www.example.com/articles/3456789012"
    }
  ],
  "version": 1
}
```

新闻数据的输入模型

您可以为输入数据定义模型。此输入模型要求新闻文章包含 URL、标题和正文。您可以使用此输入模型生成 SDK 或为您的 API 开启请求验证。

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "NewsArticleInputModel",
  "type": "object",
  "properties": {
    "count": { "type": "integer" },
    "items": {
      "type": "array",
      "items": {
        "type": "object",
        "required": [
      "article_url",
      "title",
      "body"
      ],
        "properties": {
          "last_updated_date": { "type": "string" },
          "expire_date": { "type": "string" },
          "author_first_name": { "type": "string" },
          "description": { "type": "string" },
          "creation_date": { "type": "string" },
```

```
          "title": { "type": "string" },
          "allow_comment": { "type": "boolean" },
          "author": {
            "type": "object",
            "properties": {
              "last_name": { "type": "string" },
              "email": { "type": "string" },
              "first_name": { "type": "string" }
            }
          },
          "body": { "type": "string" },
          "publish_date": { "type": "string" },
          "version": { "type": "string" },
          "author_last_name": { "type": "string" },
          "parent_id": { "type": "integer" },
          "article_url": { "type": "string" }
        }
      }
    },
    "version": { "type": "integer" }
  }
}
```

新闻数据的输出模型

您可以为输出数据定义模型。您可以将此模型用于方法响应模型，这在为 API 生成强类型 SDK 时是必需的。这会导致输出将在 Java 或 Objective-C 中强制转换为适当的类。

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "PhotosOutputModel",
  "type": "object",
  "properties": {
    "photos": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id": { "type": "string" },
          "photographedBy": { "type": "string" },
          "title": { "type": "string" },
          "ispublic": { "type": "boolean" },
          "isfriend": { "type": "boolean" },
```

```
          "isfamily": { "type": "boolean" }
        }
      }
    }
  }
}
```

新闻数据的输入映射模板

您可以定义映射模板来修改输入数据。您可以修改输入数据以进行进一步的函数集成或集成响应。

```
#set($inputRoot = $input.path('$'))
{
  "count": $inputRoot.count,
  "items": [
#foreach($elem in $inputRoot.items)
    {
      "last_updated_date": "$elem.last_updated_date",
      "expire_date": "$elem.expire_date",
      "author_first_name": "$elem.author_first_name",
      "description": "$elem.description",
      "creation_date": "$elem.creation_date",
      "title": "$elem.title",
      "allow_comment": "$elem.allow_comment",
      "author": {
        "last_name": "$elem.author.last_name",
        "email": "$elem.author.email",
        "first_name": "$elem.author.first_name"
      },
      "body": "$elem.body",
      "publish_date": "$elem.publish_date",
      "version": "$elem.version",
      "author_last_name": "$elem.author_last_name",
      "parent_id": $elem.parent_id,
      "article_url": "$elem.article_url"
    }#if($foreach.hasNext),#end

#end
  ],
  "version": $inputRoot.version
}
```

Amazon API Gateway API 请求和响应数据映射参考

本节将说明如何设置从 API 的方法请求数据（包括存储在 `context`、`stage` 或 `util` 变量中的其他数据）到相应的集成请求参数，以及从包括其他数据在内的集成响应数据到方法响应参数的数据映射。方法请求数据包括请求参数（路径、查询字符串和标头）和正文。集成响应数据包括响应参数（标头）和正文。有关使用阶段变量的更多信息，请参阅 Amazon API Gateway 阶段变量参考。

主题

- 将方法请求数据映射至集成请求参数
- 将集成响应数据映射到方法响应标头
- 在方法和集成之间映射请求和响应负载
- 集成传递行为

将方法请求数据映射至集成请求参数

可从任何定义的方法请求参数和负载映射采用路径变量、查询字符串或标头形式的集成请求参数。

此处，*PARAM_NAME* 是给定参数类型的方法请求参数的名称。它必须匹配正则表达式 `'^[a-zA-Z0-9._$-]+$]'`。它必须经过定义才能被引用。*JSONPath_EXPRESSION* 是请求或响应正文的 JSON 字段的 JSONPath 表达式。

> **Note**
>
> 此语法中省略了前缀 `"$"`。

集成请求数据映射表达式

映射数据源	映射表达式
方法请求路径	`method.request.path.` *PARAM_NAME*
方法请求查询字符串	`method.request.querystring.` *PARAM_NAME*
多值方法请求查询字符串	`method.request.multivaluequerystring.` *PARAM_NAME*

映射数据源	映射表达式
方法请求标头	`method.request.header.` *`PARAM_NAME`*
多值方法请求标头	`method.request.multivalueheader.` *`PARAM_NAME`*
方法请求正文	`method.request.body`
方法请求正文 (JsonPath)	`method.request.body.` *`JSONPath_EXPRESSION`* `.`
阶段变量	`stageVariables.` *`VARIABLE_NAME`*
上下文变量	`context.`*`VARIABLE_NAME`* ，必须为受支持的上下文变量之一。
静态值	*`'STATIC_VALUE'`* 。*`STATIC_VALUE`* 为字符串文本值，必须括在一对单引号内。

Example 从 OpenAPI 中的方法请求参数映射

下面的示例显示了一个 OpenAPI 代码段：

- 将名为 `methodRequestHeaderParam` 的方法请求标头映射到名为 `integrationPathParam` 的集成请求路径参数
- 将名为 `methodRequestQueryParam` 的多值方法请求查询字符串映射到名为 `integrationQueryParam` 的集成请求查询字符串

```
...
"requestParameters" : {

    "integration.request.path.integrationPathParam" :
 "method.request.header.methodRequestHeaderParam",
    "integration.request.querystring.integrationQueryParam" :
 "method.request.multivaluequerystring.methodRequestQueryParam"

}
```

```
...
```

还可以使用 JSONPath 表达式从 JSON 请求正文中的字段映射集成请求参数。下表显示了适用于方法请求正文及其 JSON 字段的映射表达式。

Example 从 OpenAPI 中的方法请求正文映射

下面的示例显示了一个 OpenAPI 代码段，该代码段 1) 将方法请求正文映射到名为 `body-header` 的集成请求标头，以及 2) 该正文的某个 JSON 字段，由 JSON 表达式（`petstore.pets[0].name`，不带 `$.` 前缀）表示。

```
...
"requestParameters" : {

    "integration.request.header.body-header" : "method.request.body",
    "integration.request.path.pet-name" : "method.request.body.petstore.pets[0].name",

}
...
```

将集成响应数据映射到方法响应标头

可从任意集成响应标头或集成响应正文、`$context` 变量或静态值映射方法响应标头参数。

方法响应标头映射表达式

映射数据源	映射表达式
集成响应标头	`integration.response.header. PARAM_NAME`
集成响应标头	`integration.response.multivalueheader. PARAM_NAME`
集成响应正文	`integration.response.body`

映射数据源	映射表达式
集成响应正文 (JsonPath)	`integration.response.body.` *`JSONPath_EXPRESSION`*
阶段变量	`stageVariables.` *`VARIABLE_NAME`*
上下文变量	`context.`*`VARIABLE_NAME`* ，必须为受支持的上下文变量之一。
静态值	*`'STATIC_VALUE'`* 。*`STATIC_VALUE`* 为字符串文本值，必须括在一对单引号内。

Example 从 OpenAPI 中的集成响应进行数据映射

下面的示例显示了一个 OpenAPI 代码段，该代码段 1) 将集成响应的 JSONPath 字段 `redirect.url` 映射到请求响应的 `location` 标头；并 2) 将集成响应的 `x-app-id` 标头映射到方法响应的 `id` 标头。

```
...
"responseParameters" : {

    "method.response.header.location" : "integration.response.body.redirect.url",
    "method.response.header.id" : "integration.response.header.x-app-id",
    "method.response.header.items" : "integration.response.multivalueheader.item",

}
...
```

在方法和集成之间映射请求和响应负载

API Gateway 使用 Velocity 模板语言 (VTL) 引擎来处理集成请求和集成响应的正文映射模板。映射模板将方法请求负载转换成相应的集成请求负载，并将集成响应正文转换成方法响应正文。

VTL 模板使用 JSONPath 表达式、调用上下文和阶段变量等其他参数，以及实用工具函数来处理 JSON 数据。

如果定义了一个模型来描述负载的数据结构，API Gateway 可以使用该模型为集成请求或集成响应生成骨骼映射模板。您可以使用该骨骼模板来辅助自定义和扩展映射 VTL 脚本。但是，您可以从头开始创建一个映射模板，而不为负载的数据结构定义模型。

选择 VTL 映射模板

API Gateway 使用以下逻辑选择 Velocity 模板语言 (VTL) 的映射模板，将负载从方法请求映射到相应的集成请求，或者将它从集成响应映射到相应的方法响应。

对于请求负载，API Gateway 将请求的 `Content-Type` 标头值作为密钥来选择请求负载的映射模板。对于响应负载，API Gateway 使用传入请求的 `Accept` 标头作为密钥来选择映射模板。

当请求中缺少 `Content-Type` 标头时，API Gateway 会假定其默认值为 `application/json`。对于此类请求，API Gateway 将使用 `application/json` 作为默认密钥来选择映射模板（如果已定义模板的话）。当没有模板与该密钥相匹配时， API Gateway 会在 passthroughBehavior 属性设置为 `WHEN_NO_MATCH` 或 `WHEN_NO_TEMPLATES` 时以非映射形式传递负载。

未在请求中指定 `Accept` 标头时，API Gateway 会假定其默认值为 `application/json`。在这种情况下，API Gateway 会选择 `application/json` 的现有映射模板映射响应负载。如果未对 `application/json` 定义任何模板，API Gateway 会选择第一个现有模板并将其用作默认模板来映射响应负载。类似地，当指定的 `Accept` 标头值不与任何现有的模板密钥匹配时，API Gateway 将使用第一个现有的模板。如果未定义任何模板，则 API Gateway 以非映射形式传递响应负载。

例如，假定 API 有一个为请求负载定义的 `application/json` 模板和一个为响应负载定义的 `application/xml` 模板。如果客户端设置了请求的 `"Content-Type : application/json"` 和 `"Accept : application/xml"` 标头，将使用相应的映射模板处理请求负载和响应负载。如果缺少 `Accept:application/xml` 标头，则将使用 `application/xml` 映射模板来映射响应负载。而要返回未映射的响应负载，则必须为 `application/json` 设置一个空模板。

选择映射模板时，仅从 `Accept` 和 `Content-Type` 标头中使用 MIME 类型。例如，`"Content-Type: application/json; charset=UTF-8"` 的标头将有一个已选择 `application/json` 密钥的请求模板。

集成传递行为

如果没有代理集成，当方法请求携带负载，并且 `Content-Type` 标头不匹配任何指定的映射模板或未定义任何映射模板时，您可以选择将客户端提供的请求负载通过集成请求传递到后端而不进行转换。此过程称为集成传递。

对于代理集成，API Gateway 会将整个请求传递到您的后端，并且您不能修改传递行为。

传入请求的实际传递行为由您在集成请求设置期间为指定映射模板选择的选项以及客户端在传入请求中设置的 Content Type 标头确定。这里有三个选项：

当没有模板与请求的 Content-Type 标头匹配时

> 如果您希望在方法请求内容类型不匹配任何与映射模板关联的内容类型时，将方法请求正文通过集成请求发送到后端而不进行转换，则选择此选项。
>
> 调用 API Gateway API 时，您通过将 `WHEN_NO_MATCH` 设置为集成的 `passthroughBehavior` 属性值来选择此选项。

未定义任何模板时（推荐）

> 如果您希望在集成请求中未定义映射模板时，将方法请求正文通过集成请求发送到后端而不进行转换，则选择此选项。如果选择此选项时定义了模板，则会以“HTTP 415 Unsupported Media Type”响应拒绝未映射内容类型的方法请求。
>
> 调用 API Gateway API 时，您通过将 `WHEN_NO_TEMPLATES` 设置为集成的 `passthroughBehavior` 属性值来选择此选项。

从不

> 如果您不希望在集成请求中未定义映射模板时，将方法请求正文通过集成请求发送到后端而不进行转换，则选择此选项。如果选择此选项时定义了模板，则会以“HTTP 415 Unsupported Media Type”响应拒绝未映射内容类型的方法请求。
>
> 调用 API Gateway API 时，您通过将 `NEVER` 设置为集成的 `passthroughBehavior` 属性值来选择此选项。

以下示例说明了可能的传递行为。

示例 1：`application/json` 内容类型的集成请求中定义了一个映射模板。

Content-Type 标头\选定的传递选项	**`WHEN_NO_MATCH`**	**`WHEN_NO_TEMPLATES`**	**`NEVER`**
无（默认为 `application/json`）	使用模板转换请求负载。	使用模板转换请求负载。	使用模板转换请求负载。

Content-Type 标头\选定的传递选项	**WHEN_NO_MATCH**	**WHEN_NO_TEMPLATES**	**NEVER**
`application/json`	使用模板转换请求负载。	使用模板转换请求负载。	使用模板转换请求负载。
`application/xml`	请求负载未转换，并按原样发送到后端。	请求被拒绝，得到 HTTP 415 `Unsupported Media Type` 响应。	请求被拒绝，得到 HTTP 415 `Unsupported Media Type` 响应。

示例 2：`application/xml` 内容类型的集成请求中定义了一个映射模板。

Content-Type 标头\选定的传递选项	**WHEN_NO_MATCH**	**WHEN_NO_TEMPLATES**	**NEVER**
无（默认为 `application/json`）	请求负载未转换，并按原样发送到后端。	请求被拒绝，得到 HTTP 415 `Unsupported Media Type` 响应。	请求被拒绝，得到 HTTP 415 `Unsupported Media Type` 响应。
`application/json`	请求负载未转换，并按原样发送到后端。	请求被拒绝，得到 HTTP 415 `Unsupported Media Type` 响应。	请求被拒绝，得到 HTTP 415 `Unsupported Media Type` 响应。
`application/xml`	使用模板转换请求负载。	使用模板转换请求负载。	使用模板转换请求负载。

API Gateway 映射模板和访问日志记录变量引用

本部分提供了有关 Amazon API Gateway 定义的用于数据模型、授权方、映射模板和 CloudWatch 访问日志记录的变量和函数的参考信息。有关如何使用这些变量和函数的详细信息，请参阅了解映射模板。有关 Velocity 模板语言 (VTL) 的更多信息，请参阅 VTL 参考。

主题

- 适用于数据模型、授权方、映射模板和 CloudWatch 访问日志记录的 $context 变量
- $context 变量模板示例
- 仅适用于访问日志记录的 $context 变量
- $input 变量
- $input 变量模板示例
- $stageVariables
- $util 变量

> Note
>
> 有关 `$method` 和 `$integration` 变量，请参阅 the section called “请求和响应数据映射参考”。

适用于数据模型、授权方、映射模板和 CloudWatch 访问日志记录的 `$context` 变量

以下 `$context` 变量可用于数据模型、授权方、映射模板和 CloudWatch 访问日志记录。API Gateway 可能会添加其他上下文变量。

有关只能在 CloudWatch 访问日志记录中使用的 `$context` 变量，请参阅the section called “仅适用于访问日志记录的 `$context` 变量”。

参数	说明
`$context.accountId`	API 拥有者的 AWS 账户 ID。
`$context.apiId`	API Gateway 分配给您的 API 的标识符。
`$context.authorizer.claims.`*`property`*	成功对方法调用方进行身份验证后从 Amazon Cognito 用户池返回的声明的属性。有关更多信息，请参阅 the section called “使用 Amazon Cognito 用户池作为 REST API 的授权方”。

参数	说明
	Note 调用 `$context.authorizer.claims` 将返回 null。
`$context.authorizer.principalId`	与由客户端发送的令牌关联并从 API Gateway Lambda 授权方（以前称为自定义授权方）返回的委托人用户标识。有关更多信息，请参阅 [the section called “使用 Lambda 授权方”]()。
`$context.authorizer.` *`property`*	从 API Gateway Lambda 授权方函数返回的 `context` 映射的指定键/值对的字符串化值。例如，如果授权方返回以下 `context` 映射：

```
"context" : {
  "key": "value",
  "numKey": 1,
  "boolKey": true
}
```

参数	说明
	调用 `$context.authorizer.key` 将返回 `"value"` 字符串，调用 `$context.authorizer.numKey` 将返回 `"1"` 字符串，而调用 `$context.authorizer.boolKey` 将返回 `"true"` 字符串。 有关更多信息，请参阅 [the section called “使用 Lambda 授权方”]()。
`$context.awsEndpointRequestId`	AWS 端点的请求 ID。
`$context.deploymentId`	API 部署的 ID。
`$context.domainName`	用于调用 API 的完整域名。这应与传入的 `Host` 标头相同。
`$context.domainPrefix`	`$context.domainName` 的第一个标签。

参数	说明
`$context.error.message`	包含 API Gateway 错误消息的字符串。此变量只能用于 GatewayResponse 正文映射模板中（不由 Velocity 模板语言引擎处理）和访问日志记录中的简单变量替换。有关更多信息，请参阅the section called “指标” 和the section called “设置网关响应以自定义错误响应”。
`$context.error.messageString`	`$context.error.message` 的带引号的值，即 `"$context.error.message"` 。
`$context.error.responseType`	GatewayResponse 的类型 此变量只能用于 GatewayResponse 正文映射模板中（不由 Velocity 模板语言引擎处理）和访问日志记录中的简单变量替换。有关更多信息，请参阅the section called “指标” 和the section called “设置网关响应以自定义错误响应”。
`$context.error.validationErrorString`	包含详细验证错误消息的字符串。
`$context.extendedRequestId`	API Gateway 生成并分配给 API 请求的扩展 ID。扩展请求 ID 包含调试和故障排除的有用信息。
`$context.httpMethod`	所用的 HTTP 方法。有效值包括：`DELETE`、`GET`、`HEAD`、`OPTIONS`、`PATCH`、`POST` 和 `PUT`。
`$context.identity.accountId`	与请求关联的 AWS 账户 ID。
`$context.identity.apiKey`	对于需要 API 密钥的 API 方法，此变量是与该方法请求关联的 API 密钥。对于无需 API 密钥的方法，此变量为 null。有关更多信息，请参阅the section called “使用计划”。

参数	说明
`$context.identity.apiKeyId`	与需要 API 密钥的 API 请求关联的 API 密钥 ID。
`$context.identity.caller`	签发请求的调用方的委托人标识符。对于使用 IAM 授权的资源支持此项。
`$context.identity.cognitoAuthenticationProvider`	发出请求的调用方使用的 Amazon Cognito 身份验证提供商的逗号分隔列表。仅当使用 Amazon Cognito 凭证对请求签名时才可用。 例如，对于 Amazon Cognito 身份池中的身份，`cognito-idp.` *`region`* `.amazonaws.com/` *`user_pool_id`* `,cognito-idp.` *`region`* `.amazonaws.com/` *`user_pool_id`* `:CognitoSignIn:` *`token subject claim`* 有关更多信息，请参阅 Amazon Cognito 开发人员指南 中的使用联合身份。
`$context.identity.cognitoAuthenticationType`	发出请求的调用方的 Amazon Cognito 身份验证类型。仅当使用 Amazon Cognito 凭证对请求签名时才可用。可能的值包括经过身份验证的身份的 `authenticated` 和未经身份验证的身份的 `unauthenticated`。
`$context.identity.cognitoIdentityId`	发出请求的调用方的 Amazon Cognito 身份 ID。仅当使用 Amazon Cognito 凭证对请求签名时才可用。
`$context.identity.cognitoIdentityPoolId`	发出请求的调用方的 Amazon Cognito 身份池 ID。仅当使用 Amazon Cognito 凭证对请求签名时才可用。
`$context.identity.principalOrgId`	AWS 组织 ID。

参数	说明
`$context.identity.sourceIp`	向 API Gateway 端点发出请求的即时 TCP 连接的源 IP 地址。
`$context.identity.clientCert.clientCertPem`	客户端在双向 TLS 身份验证过程中提供的 PEM 编码的客户端证书。当客户端使用已启用双向 TLS 的自定义域名访问 API 时提供。仅在双向 TLS 身份验证失败时才出现在访问日志中。
`$context.identity.clientCert.subjectDN`	客户端提供的证书的主题的可分辨名称。当客户端使用已启用双向 TLS 的自定义域名访问 API 时提供。仅在双向 TLS 身份验证失败时才出现在访问日志中。
`$context.identity.clientCert.issuerDN`	客户端提供的证书的颁发者的可分辨名称。当客户端使用已启用双向 TLS 的自定义域名访问 API 时提供。仅在双向 TLS 身份验证失败时才出现在访问日志中。
`$context.identity.clientCert.serialNumber`	证书的序列号。当客户端使用已启用双向 TLS 的自定义域名访问 API 时提供。仅在双向 TLS 身份验证失败时才出现在访问日志中。
`$context.identity.clientCert.validity.notBefore`	证书无效之前的日期。当客户端使用已启用双向 TLS 的自定义域名访问 API 时提供。仅在双向 TLS 身份验证失败时才出现在访问日志中。
`$context.identity.clientCert.validity.notAfter`	证书无效后的日期。当客户端使用已启用双向 TLS 的自定义域名访问 API 时提供。仅在双向 TLS 身份验证失败时才出现在访问日志中。
`$context.identity.vpcId`	向 API Gateway 端点发出请求的 VPC 的 VPC ID。
`$context.identity.vpceId`	向 API Gateway 端点发出请求的 VPC 端点的 VPC 端点 ID。仅当您具有私有 API 时才会显示。

参数	说明
`$context.identity.user`	将获得资源访问权限授权的用户的委托人标识符。对于使用 IAM 授权的资源支持此项。
`$context.identity.userAgent`	API 调用方的 [`User-Agent`](#) 标头。
`$context.identity.userArn`	身份验证后标识的有效用户的 Amazon Resource Name (ARN)。有关更多信息，请参阅 https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users.html。
`$context.isCanaryRequest`	如果请求定向到金丝雀，将返回 `true`，如果请求没有定向到金丝雀，将返回 `false`。仅当您启用金丝雀时才会显示。
`$context.path`	请求路径。例如，对于 `https://{rest-api-id}.execute-api.{region}.amazonaws.com/{stage}/root/child` 的非代理请求 URL，`$context.path` 值为 `/{stage}/root/child`。
`$context.protocol`	请求的协议，例如，`HTTP/1.1`。 **Note** API Gateway API 可以接受 HTTP/2 请求，但 API Gateway 使用 HTTP/1.1 向后端集成发送请求。因此，即使客户端发送的请求使用 HTTP/2，请求协议也会记录为 HTTP/1.1。
`$context.requestId`	请求的 ID。客户可以覆盖此请求 ID。使用 `$context.extendedRequestId` 用于 API Gateway 生成的唯一请求 ID。

参数	说明
`$context.requestOverride.header.` *`header_name`*	请求标头覆盖。如果定义此参数，则它将包含要使用的标题，而不是集成请求窗格中定义的 HTTP 标头。有关更多信息，请参阅 使用映射模板覆盖 API 的请求和响应参数以及状态代码。
`$context.requestOverride.path.` *`path_name`*	请求路径覆盖。如果定义此参数，则它将包含要使用的请求路径，而不是集成请求窗格中定义的 URL 路径参数。有关更多信息，请参阅 使用映射模板覆盖 API 的请求和响应参数以及状态代码。
`$context.requestOverride.querystring.` *`querystring_name`*	请求查询字符串覆盖。如果定义此参数，则它将包含要使用的请求查询字符串，而不是集成请求窗格中定义的 URL 查询字符串参数。有关更多信息，请参阅 使用映射模板覆盖 API 的请求和响应参数以及状态代码。
`$context.responseOverride.header.` *`header_name`*	响应标头覆盖。如果定义此参数，则它包含要使用的标头，而不是集成响应窗格中定义为默认映射的响应标头。有关更多信息，请参阅 使用映射模板覆盖 API 的请求和响应参数以及状态代码。
`$context.responseOverride.status`	响应状态代码覆盖。如果定义此参数，则它包含要使用的状态代码，而不是集成响应窗格中定义为默认映射的方法响应状态。有关更多信息，请参阅 使用映射模板覆盖 API 的请求和响应参数以及状态代码。
`$context.requestTime`	CLF 格式的请求时间 (`dd/MMM/yyyy:HH:mm:ss +-hhmm`)。
`$context.requestTimeEpoch`	Epoch 格式的请求时间，以毫秒为单位。
`$context.resourceId`	API Gateway 分配给您的资源的标识符。

参数	说明
`$context.resourcePath`	资源路径。例如，对于 `https://{rest-api-id}.execute-api.{region}.amazonaws.com/{stage}/root/child` 的非代理请求 URI，`$context.resourcePath` 值为 `/root/child`。有关更多信息，请参阅 教程：使用 HTTP 非代理集成构建 REST API。
`$context.stage`	API 请求的部署阶段（例如，`Beta` 或 `Prod`）。
`$context.wafResponseCode`	从 AWS WAF 中收到的响应：`WAF_ALLOW` 或 `WAF_BLOCK`。如果阶段未与 Web ACL 关联，则不会设置。有关更多信息，请参阅 the section called “AWS WAF”。
`$context.webaclArn`	Web ACL 的完整 ARN，用于决定是允许还是阻止请求。如果阶段未与 Web ACL 关联，则不会设置。有关更多信息，请参阅 the section called “AWS WAF”。

`$context` 变量模板示例

如果您的 API 方法将结构化数据传递到要求数据采用特定格式的后端，则您可能要在映射模板中使用 `$context` 变量。

以下示例显示了一个映射模板，该模板将传入的 `$context` 变量映射到后端变量，这些变量在集成请求负载中的名称稍有不同：

> **Note**
>
> 其中一个变量是 API 密钥。此示例假设方法需要 API 密钥。

```
{
    "stage" : "$context.stage",
```

```
    "request_id" : "$context.requestId",
    "api_id" : "$context.apiId",
    "resource_path" : "$context.resourcePath",
    "resource_id" : "$context.resourceId",
    "http_method" : "$context.httpMethod",
    "source_ip" : "$context.identity.sourceIp",
    "user-agent" : "$context.identity.userAgent",
    "account_id" : "$context.identity.accountId",
    "api_key" : "$context.identity.apiKey",
    "caller" : "$context.identity.caller",
    "user" : "$context.identity.user",
    "user_arn" : "$context.identity.userArn"
}
```

此映射模板的输出应与以下内容类似：

```
{
  stage: 'prod',
  request_id: 'abcdefg-000-000-0000-abcdefg',
  api_id: 'abcd1234',
  resource_path: '/',
  resource_id: 'efg567',
  http_method: 'GET',
  source_ip: '192.0.2.1',
  user-agent: 'curl/7.84.0',
  account_id: '111122223333',
  api_key: 'MyTestKey',
  caller: 'ABCD-0000-12345',
  user: 'ABCD-0000-12345',
  user_arn: 'arn:aws:sts::111122223333:assumed-role/Admin/carlos-salazar'
}
```

仅适用于访问日志记录的 `$context` 变量

以下 `$context` 变量仅适用于访问日志记录。有关更多信息，请参阅 the section called “CloudWatch 日志”。（有关 WebSocket API，请参阅the section called “指标”。）

参数	说明
`$context.authorize.error`	授权错误消息。
`$context.authorize.latency`	授权延迟时间（以毫秒为单位）。

参数	说明
`$context.authorize.status`	从授权尝试返回的状态代码。
`$context.authorizer.error`	从授权方返回的错误消息。
`$context.authorizer.integrationLatency`	授权方延迟（以毫秒为单位）。
`$context.authorizer.integrationStatus`	从 Lambda 授权方返回的状态代码。
`$context.authorizer.latency`	授权方延迟（以毫秒为单位）。
`$context.authorizer.requestId`	AWS 端点的请求 ID。
`$context.authorizer.status`	从授权方返回的状态代码。
`$context.authenticate.error`	从身份验证尝试返回的错误消息。
`$context.authenticate.latency`	身份验证延迟时间（以毫秒为单位）。
`$context.authenticate.status`	从身份验证尝试返回的状态代码。
`$context.customDomain.basePathMatched`	传入请求所匹配的 API 映射路径。适用于客户端使用自定义域名访问 API 的情况。例如，如果客户端向 `https://api.example.com/v1/orders/1234` 发送请求，且该请求匹配路径为 `v1/orders` 的 API 映射，则值为 `v1/orders`。要了解更多信息，请参阅 the section called “API 映射”。
`$context.endpointType`	API 的端点类型。
`$context.integration.error`	从集成返回的错误消息。
`$context.integration.integrationStatus`	对于 Lambda 代理集成，从 AWS Lambda（而不是从后端 Lambda 函数代码）返回的状态代码。

参数	说明
`$context.integration.latency`	集成延迟（毫秒）。等效于 `$context.integrationLatency` 。
`$context.integration.requestId`	AWS 端点的请求 ID。等效于 `$context.awsEndpointRequestId` 。
`$context.integration.status`	从集成返回的状态代码。对于 Lambda 代理集成，这是 Lambda 函数代码返回的状态代码。
`$context.integrationLatency`	集成延迟（毫秒）。
`$context.integrationStatus`	对于 Lambda 代理集成，此参数表示从 AWS Lambda Lambda（而不是从后端 Lambda 函数代码）返回的状态代码。
`$context.responseLatency`	响应延迟（毫秒）。
`$context.responseLength`	响应负载长度（以字节为单位）。
`$context.status`	方法响应状态。
`$context.waf.error`	从 返回的错误消息AWS WAF
`$context.waf.latency`	AWS WAF 延迟时间（以毫秒为单位）。
`$context.waf.status`	从 返回的状态代码AWS WAF
`$context.xrayTraceId`	X-Ray 跟踪的跟踪 ID。有关更多信息，请参阅 the section called “设置 AWS X-Ray”。

`$input` 变量

`$input` 变量表示将由映射模板处理的方法请求负载和参数。它可以提供以下函数：

变量和函数	说明
`$input.body`	以字符串形式返回原始请求负载。

变量和函数	说明
`$input.json(x)`	此函数计算 JSONPath 表达式并以 JSON 字符串形式返回结果。 例如，`$input.json('$.pets')` 返回一个表示 `pets` 结构的 JSON 字符串。 有关 JSONPath 的更多信息，请参阅 JSONPath 或适用于 Java 的 JSONPath。
`$input.params()`	返回所有请求参数的映射。我们建议您使用 `$util.escapeJavaScript` 对结果进行清理，以避免潜在的注入攻击。要完全控制请求清理，可以使用没有模板的代理集成，并在集成中处理请求清理。
`$input.params(x)`	在给定参数名称字符串 x 的情况下，返回路径中的方法请求参数值、查询字符串或标头值（按照该顺序搜索）。我们建议您使用 `$util.escapeJavaScript` 对参数进行清理，以避免潜在的注入攻击。要完全控制参数清理，可以使用没有模板的代理集成，并在集成中处理请求清理。

变量和函数	说明
`$input.path(x)`	获取一个 JSONPath 表达式字符串 (x) 并返回结果的 JSON 对象表达式。这样，您便可通过 Apache Velocity 模板语言 (VTL) 在本机访问和操作负载的元素。 例如，如果表达式 `$input.path('$.pets')` 返回一个如下所示的对象：

```
[
  {
    "id": 1,
    "type": "dog",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "cat",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
```

`$input.path('$.pets').count()` 将返回 `"3"`。

有关 JSONPath 的更多信息，请参阅 JSONPath 或适用于 Java 的 JSONPath。

`$input` 变量模板示例

以下示例显示了如何在映射模板中使用 `$input` 变量：您可以使用模拟集成或用于将输入事件返回到 API Gateway 的 Lambda 非代理集成来尝试这些示例。

参数映射模板示例

以下示例将所有请求参数（包括 path、querystring 和 header）通过 JSON 负载传递到集成端点：

```
#set($allParams = $input.params())
{
  "params" : {
    #foreach($type in $allParams.keySet())
    #set($params = $allParams.get($type))
    "$type" : {
      #foreach($paramName in $params.keySet())
      "$paramName" : "$util.escapeJavaScript($params.get($paramName))"
      #if($foreach.hasNext),#end
      #end
    }
    #if($foreach.hasNext),#end
    #end
  }
}
```

对于包含以下输入参数的请求：

- 名为 myparam 的路径参数
- 查询字符串参数 querystring1=value1,value2&querystring2=value3
- 标头 "header1" : "value1"、"header2" : "value2"、"header3" : "value3"。

此映射模板的输出应与以下内容类似：

```
{
  "params" : {
        "path" : {
            "path" : "myparam"
                }
    ,        "querystring" : {
            "querystring1" : "value1,value2"
      ,             "querystring2" : "value3"
                }
    ,        "header" : {
            "header3" : "value3"
      ,             "header2" : "value2"
```

```
      ,            "header1" : "value1"
                 }
            }
}
```

JSON 映射模板示例

您可能希望使用 `$input` 变量来获取查询字符串和请求正文（无论是否使用模型）。您可能还希望获取参数和负载或者负载的子部分。以下三个示例显示如何执行此操作。

以下示例使用映射模板来获取负载的子部分。此示例获取输入参数 `name`，然后获取整个 POST 正文：

```
{
    "name" : "$input.params('name')",
    "body" : $input.json('$')
}
```

对于包含查询字符串参数 `name=Bella&type=dog` 和以下正文的请求：

```
{
    "Price" : "249.99",
    "Age": "6"
}
```

此映射模板的输出应与以下内容类似：

```
{
    "name" : "Bella",
    "body" : {"Price":"249.99","Age":"6"}
}
```

如果 JSON 输入包含无法通过 JavaScript 解析的非转义字符，则 API Gateway 可能会返回 400 响应。应用 `$util.escapeJavaScript($input.json('$'))` 来确保正确解析 JSON 输入。

上面的示例在应用了 `$util.escapeJavaScript($input.json('$'))` 之后会如下所示：

```
{
    "name" : "$input.params('name')",
    "body" : $util.escapeJavaScript($input.json('$'))
}
```

在这种情况下，此映射模板的输出应与以下内容类似：

```
{
    "name" : "Bella",
    "body": {\"Price\":\"249.99\",\"Age\":\"6\"}
}
```

JSONPath 表达式示例

以下示例介绍如何将 JSONPath 表达式传递到 `json()` 方法。您也可以通过使用句点 `.` 指定属性，来读取请求正文对象的子部分：

```
{
    "name" : "$input.params('name')",
    "body" : $input.json('$.Age')
}
```

对于包含查询字符串参数 `name=Bella&type=dog` 和以下正文的请求：

```
{
    "Price" : "249.99",
    "Age": "6"
}
```

此映射模板的输出应与以下内容类似：

```
{
    "name" : "Bella",
    "body" : "6"
}
```

如果方法请求负载包含无法通过 JavaScript 解析的非转义字符，则 API Gateway 可能会返回 `400` 响应。应用 `$util.escapeJavaScript()` 来确保正确解析 JSON 输入。

上面的示例在应用了 `$util.escapeJavaScript($input.json('$.Age'))` 之后会如下所示：

```
{
    "name" : "$input.params('name')",
    "body" : "$util.escapeJavaScript($input.json('$.Age'))"
}
```

在这种情况下，此映射模板的输出应与以下内容类似：

```
{
    "name" : "Bella",
    "body": "\"6\""
}
```

请求和响应示例

以下示例对路径为 `/things/{id}` 的资源使用 `$input.params()`、`$input.path()` 和 `$input.json()`：

```
{
    "id" : "$input.params('id')",
    "count" : "$input.path('$.things').size()",
    "things" : $input.json('$.things')"
}
```

对于包含路径参数 123 和以下正文的请求：

```
{
      "things": {
            "1": {},
            "2": {},
            "3": {}
      }
}
```

此映射模板的输出应与以下内容类似：

```
{"id":"123","count":"3","things":{"1":{},"2":{},"3":{}}}
```

如果方法请求负载包含无法通过 JavaScript 解析的非转义字符，则 API Gateway 可能会返回 `400` 响应。应用 `$util.escapeJavaScript()` 来确保正确解析 JSON 输入。

上面的示例在应用了 `$util.escapeJavaScript($input.json('$.things'))` 之后会如下所示：

```
{
     "id" : "$input.params('id')",
     "count" : "$input.path('$.things').size()",
     "things" : "$util.escapeJavaScript($input.json('$.things'))"
}
```

此映射模板的输出应与以下内容类似：

```
{"id":"123","count":"3","things":"{\"1\":{},\"2\":{},\"3\":{}}"}
```

有关更多映射示例，请参阅了解映射模板。

`$stageVariables`

阶段变量可用于参数映射和映射模板，并可用作在方法集成中使用的 ARN 和 URL 中的占位符。有关更多信息，请参阅 the section called “设置阶段变量”。

语法	描述
`$stageVariables.` *`<variable_name>`* 、`$stageVariables['` *`<variable_name>`* `']` 或 `${stageVariables['` *`<variable_name>`* `']}`	*`<variable_name>`* 表示阶段变量名称。

`$util` 变量

`$util` 变量包含映射模板中使用的实用程序函数。

> **Note**
>
> 除非另行指定，否则默认字符集为 UTF-8。

函数	说明
`$util.escapeJavaScript()`	使用 JavaScript 字符串规则对字符串中的字符进行转义。 **Note** 此函数会将任何常规单引号 (') 变成转义单引号 (\')。但是，转义单引号在 JSON 中无效。因此，当此函数的输出

函数	说明
	用于 JSON 属性时，必须将任何转义单引号 (\') 变回常规单引号 (')。如下例所示:

```
"input" : "$util.escapeJavaScript(data).replaceAll("\\'","'")"
```

`$util.parseJson()`	获取“字符串化的”JSON 并返回结果的对象表示形式。您可以使用此函数的结果通过 Apache Velocity 模板语言 (VTL) 在本机访问和操作负载的元素。例如，如果您具有以下负载：

```
{"errorMessage":"{\"key1\":\"var1\",\"key2\":{\"arr\":[1,2,3]}}"}
```

并使用以下映射模板

```
#set ($errorMessageObj = $util.parseJson($input.path('$.errorMessage')))
{
   "errorMessageObjKey2ArrVal" : $errorMessageObj.key2.arr[0]
}
```

您将得到以下输出：

```
{
   "errorMessageObjKey2ArrVal" : 1
}
```

`$util.urlEncode()`	将字符串转换为“application/x-www-form-urlencoded”格式。

函数	说明
`$util.urlDecode()`	对“application/x-www-form-urlencoded”字符串进行解码。
`$util.base64Encode()`	将数据编码为 base64 编码的字符串。
`$util.base64Decode()`	对 base64 编码字符串中的数据进行解码。

API Gateway 中的网关响应

网关响应使用 API Gateway 定义的响应类型进行标识。响应由 HTTP 状态代码 (这是一组由参数映射指定的额外标头) 和由非 VTL 映射模板生成的负载组成。

在 API Gateway REST API 中，GatewayResponse 表示网关响应。在 OpenAPI 中，x-amazon-apigateway-gateway-responses.gatewayResponse 扩展表示 `GatewayResponse` 实例。

要启用网关响应，您需要在 API 级别为受支持的响应类型设置网关响应。每当 API Gateway 返回该类型的响应时，都将使用在网关响应中定义的标头映射和负载映射模板，以将映射结果返回 API 调用方。

下一节我们将介绍如何使用 API Gateway 控制台和 API Gateway REST API 设置网关响应。

设置网关响应以自定义错误响应

如果 API Gateway 无法处理某个传入请求，它会向客户端返回一个错误响应，而不会将请求转发到集成后端。默认情况下，错误响应会包含一个简短的描述性错误消息。例如，如果您尝试对未定义的 API 资源调用操作，您将收到一个显示 `{ "message": "Missing Authentication Token" }` 消息的错误响应。如果首次使用 API Gateway，您可能会发现很难找到真正的问题。

对于某些错误响应，API Gateway 允许 API 开发人员通过自定义返回不同格式的响应。对于 `Missing Authentication Token` 示例，您可以为原始响应负载添加提示标注可能的原因，如下例所示：`{"message":"Missing Authentication Token", "hint":"The HTTP method or resources may not be supported."}`。

当您的 API 在外部交换和 AWS 云之间提供协调时，您可以使用集成请求或集成响应的 VTL 映射模板将负载从一种格式映射到另一种格式。但是，VTL 映射模板仅适用于能够成功响应的有效请求。

对于无效请求，API Gateway 可以完全绕过该集成，返回错误响应。您必须通过自定义，以可支持交换的格式发出错误响应。此时，使用仅支持简单变量替换的非 VTL 映射模板进行自定义。

将 API Gateway 生成的错误响应泛化处理成由 API Gateway 生成的任何响应，我们将它们称为网关响应。以此区分 API Gateway 生成的响应与集成响应。网关响应映射模板能以 `$context` 的形式访问 `$stageVariables` 变量值和 `method.request.`*`param-position`*`.`*`param-name`* 属性值以及方法请求参数。

有关 `$context` 变量的更多信息，请参阅 适用于数据模型、授权方、映射模板和 CloudWatch 访问日志记录的 `$context` 变量。有关 `$stageVariables` 的更多信息，请参阅 `$stageVariables`。有关方法请求参数的更多信息，请参阅 the section called “`$input` 变量”。

主题

- 使用 API Gateway 控制台为 REST API 设置网关响应
- 使用 API Gateway REST API 设置网关响应
- 在 OpenAPI 中设置网关响应自定义
- 网关响应类型

使用 API Gateway 控制台为 REST API 设置网关响应

使用 API Gateway 控制台自定义网关响应

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 在主导航窗格中，选择网关响应。
4. 选择响应类型，然后选择编辑。在本次演练中，我们将以缺少身份验证令牌为例。
5. 您可以更改 API Gateway 生成的状态代码，以返回满足您的 API 要求的不同状态代码。在此示例中，自定义会将状态代码从默认值 (`403`) 更改为 `404`，因为在客户端调用可被视为未找到的不受支持或无效的资源时，会出现此错误消息。
6. 要返回自定义标头，请选择响应标头下的添加响应标头。为方便说明，我们将添加以下自定义标头：

   ```
   Access-Control-Allow-Origin:'a.b.c'
   x-request-id:method.request.header.x-amzn-RequestId
   x-request-path:method.request.path.petId
   x-request-query:method.request.querystring.q
   ```

在前面的标头映射中，将静态域名 (`'a.b.c'`) 映射到 `Allow-Control-Allow-Origin` 标头以允许 CORS 访问 API；将 `x-amzn-RequestId` 的输入请求标头映射到响应中的 `request-id`；将传入请求的 `petId` 路径变量映射到响应中的 `request-path` 标头；以及将原始请求的 `q` 查询参数映射到响应的 `request-query` 标头。

7. 在响应模板下，将 `application/json` 保留为内容类型，然后在模板正文编辑器中输入以下正文映射模板：

```
{
     "message":"$context.error.messageString",
     "type": "$context.error.responseType",
     "statusCode": "'404'",
     "stage": "$context.stage",
     "resourcePath": "$context.resourcePath",
     "stageVariables.a": "$stageVariables.a"
}
```

   此示例显示了如何将 `$context` 和 `$stageVariables` 属性映射到网关响应正文的属性。

8. 选择保存更改。
9. 将 API 部署到新阶段或现有阶段。

通过调用以下 CURL 命令测试您的网关响应，假设相应 API 方法的调用 URL 是 `https://`*`o81lxisefl`*`.execute-api.us-east-1.amazonaws.com/custErr/pets/{petId}`：

```
curl -v -H 'x-amzn-RequestId:123344566' https://o81lxisefl.execute-api.us-
east-1.amazonaws.com/custErr/pets/5/type?q=1
```

因为额外查询字符串参数 `q=1` 与 API 不兼容，所以返回错误，从而触发指定的网关响应。您应收到与以下内容类似的网关响应：

```
> GET /custErr/pets/5?q=1 HTTP/1.1
Host: o81lxisefl.execute-api.us-east-1.amazonaws.com
User-Agent: curl/7.51.0
Accept: */*

HTTP/1.1 404 Not Found
Content-Type: application/json
Content-Length: 334
```

```
Connection: keep-alive
Date: Tue, 02 May 2017 03:15:47 GMT
x-amzn-RequestId: 123344566
Access-Control-Allow-Origin: a.b.c
x-amzn-ErrorType: MissingAuthenticationTokenException
header-1: static
x-request-query: 1
x-request-path: 5
X-Cache: Error from cloudfront
Via: 1.1 441811a054e8d055b893175754efd0c3.cloudfront.net (CloudFront)
X-Amz-Cf-Id: nNDR-fX4csbRoAgtQJ16u0rTDz9FZWT-Mk93KgoxnfzDlTUh3flmzA==

{
     "message":"Missing Authentication Token",
     "type": MISSING_AUTHENTICATION_TOKEN,
     "statusCode": '404',
     "stage": custErr,
     "resourcePath": /pets/{petId},
     "stageVariables.a": a
}
```

前面的示例假定 API 后端为 Pet Store，并且 API 有一个定义的阶段变量 a。

使用 API Gateway REST API 设置网关响应

在使用 API Gateway REST API 自定义网关响应之前，您必须已创建 API 并获得其标识符。要检索 API 标识符，您可以使用 restapi:gateway-responses 链接关系并检查结果。

使用 API Gateway REST API 自定义网关响应

1. 要覆盖整个 GatewayResponse 实例，请调用 gatewayresponse:put 操作。在 URL 路径参数中指定所需的 responseType，并在请求负载中提供 statusCode、responseParameters 和 responseTemplates 映射。
2. 要更新 GatewayResponse 实例的一部分，请调用 gatewayresponse:update 操作。在 URL 路径参数中指定所需的 `responseType`，并在请求中提供所需的单个 `GatewayResponse` 属性，例如，`responseParameters` 或 `responseTemplates` 映射。

在 OpenAPI 中设置网关响应自定义

在 OpenAPI 中，您可以在 API 根级别使用 `x-amazon-apigateway-gateway-responses` 扩展来自定义网关响应。下面的 OpenAPI 定义显示了自定义 `MISSING_AUTHENTICATION_TOKEN` 类型的 GatewayResponse 的示例。

```
  "x-amazon-apigateway-gateway-responses": {
    "MISSING_AUTHENTICATION_TOKEN": {
      "statusCode": 404,
      "responseParameters": {
        "gatewayresponse.header.x-request-path": "method.input.params.petId",
        "gatewayresponse.header.x-request-query": "method.input.params.q",
        "gatewayresponse.header.Access-Control-Allow-Origin": "'a.b.c'",
        "gatewayresponse.header.x-request-header": "method.input.params.Accept"
      },
      "responseTemplates": {
        "application/json": "{\n     \"message\": $context.error.messageString,\n     \"type\":  \"$context.error.responseType\",\n     \"stage\":  \"$context.stage\",\n     \"resourcePath\":  \"$context.resourcePath\",\n     \"stageVariables.a\":  \"$stageVariables.a\",\n     \"statusCode\": \"'404'\"\n}"
      }
    }
```

在此示例中，自定义设置将状态代码从默认 (`403`) 更改为 `404`。此外，它还会为网关响应添加了四个标头参数和一个 `application/json` 媒体类型的正文映射模板。

网关响应类型

API Gateway 公开了以下网关响应以供 API 开发人员进行自定义。

网关响应类型	默认状态代码	说明
`ACCESS_DENIED`	`403`	授权失败的网关响应；例如，被自定义授权方或 Amazon Cognito 授权方拒绝访问时。如果未指定响应类型，则默认该响应为 `DEFAULT_4XX` 类型。
`API_CONFIGURATION_ERROR`	`500`	无效 API 配置的网关响应，包括提交无效的端点地址时，在

网关响应类型	默认状态代码	说明
		设置二进制支持时对二进制数据进行 Base64 解码失败时，或者集成响应映射无法匹配任何模板且未配置默认模板时。如果未指定响应类型，则默认该响应为 `DEFAULT_5XX` 类型。
`AUTHORIZER_CONFIGURATION_ERROR`	`500`	未能连接到自定义或 Amazon Cognito 授权方的网关响应。如果未指定响应类型，则默认该响应为 `DEFAULT_5XX` 类型。
`AUTHORIZER_FAILURE`	`500`	当自定义或 Amazon Cognito 授权方无法对调用方进行身份验证时的网关响应。如果未指定响应类型，则默认该响应为 `DEFAULT_5XX` 类型。
`BAD_REQUEST_PARAMETERS`	`400`	当无法根据已启用的请求验证程序验证请求参数时的网关响应。如果未指定响应类型，则默认该响应为 `DEFAULT_4XX` 类型。
`BAD_REQUEST_BODY`	`400`	当无法根据已启用的请求验证程序验证请求正文时的网关响应。如果未指定响应类型，则默认该响应为 `DEFAULT_4XX` 类型。

网关响应类型	默认状态代码	说明
`DEFAULT_4XX`	Null	状态代码为 4XX 的未指定响应类型的默认网关响应。更改此回退网关响应的状态代码会将所有其他 4XX 响应的状态代码更改为新值。将此状态代码重置为 Null 会使所有其他 4XX 响应的状态代码恢复到原始值。 Note AWS WAF 自定义响应优先于自定义网关响应。
`DEFAULT_5XX`	Null	状态代码为 5XX 的未指定响应类型的默认网关响应。更改此回退网关响应的状态代码会将所有其他 5XX 响应的状态代码更改为新值。将此状态代码重置为 Null 会使所有其他 5XX 响应的状态代码恢复到原始值。
`EXPIRED_TOKEN`	`403`	发生 AWS 身份验证令牌过期错误时的网关响应。如果未指定响应类型，则默认该响应为 `DEFAULT_4XX` 类型。
`INTEGRATION_FAILURE`	`504`	发生集成失败错误时的网关响应。如果未指定响应类型，则默认该响应为 `DEFAULT_5XX` 类型。

网关响应类型	默认状态代码	说明
`INTEGRATION_TIMEOUT`	504	发生集成超时错误时的网关响应。如果未指定响应类型，则默认该响应为 `DEFAULT_5XX` 类型。
`INVALID_API_KEY`	403	为要求 API 密钥的方法提交无效 API 密钥时的网关响应。如果未指定响应类型，则默认该响应为 `DEFAULT_4XX` 类型。
`INVALID_SIGNATURE`	403	发生无效的 AWS 签名错误时的网关响应。如果未指定响应类型，则默认该响应为 `DEFAULT_4XX` 类型。
`MISSING_AUTHENTICATION_TOKEN`	403	发生缺少身份验证令牌错误时的网关响应，包括客户端尝试调用不受支持的 API 方法或资源的情况。如果未指定响应类型，则默认该响应为 `DEFAULT_4XX` 类型。
`QUOTA_EXCEEDED`	429	发生超出使用计划配额错误时的网关响应。如果未指定响应类型，则默认该响应为 `DEFAULT_4XX` 类型。
`REQUEST_TOO_LARGE`	413	发生请求太大错误时的网关响应。如果未指定响应类型，则该响应默认为：`HTTP content length exceeded 10485760 bytes`。

网关响应类型	默认状态代码	说明
`RESOURCE_NOT_FOUND`	404	在 API 请求通过身份验证和授权（不包括 API 密钥身份验证和授权）后，API Gateway 找不到指定资源时的网关响应。如果未指定响应类型，则默认该响应为 `DEFAULT_4XX` 类型。
`THROTTLED`	429	当超出使用计划、方法、阶段或账户的节流限制时的网关响应。如果未指定响应类型，则默认该响应为 `DEFAULT_4XX` 类型。
`UNAUTHORIZED`	401	当自定义或 Amazon Cognito 授权方无法对调用方进行身份验证时的网关响应。
`UNSUPPORTED_MEDIA_TYPE`	415	当启用严格的传递限制后，负载为不受支持的媒体类型时的网关类型。如果未指定响应类型，则默认该响应为 `DEFAULT_4XX` 类型。
`WAF_FILTERED`	403	当请求被 AWS WAF 阻止时的网关响应。如果未指定响应类型，则默认该响应为 `DEFAULT_4XX` 类型。 Note AWS WAF 自定义响应优先于自定义网关响应。

为 REST API 资源启用 CORS

跨源资源共享 (CORS) 是一项浏览器安全特征，该特征限制从在浏览器中运行的脚本启动的跨源 HTTP 请求。有关更多信息，请参阅什么是 CORS？。

确定是否启用 CORS 支持

跨源 HTTP 请求将向以下项发出：

- 一个不同的域（例如，从 `example.com` 到 `amazondomains.com`）
- 一个不同的子域（例如，从 `example.com` 到 `petstore.example.com`）
- 一个不同的端口（例如，从 `example.com` 到 `example.com:10777`）
- 一个不同的协议（例如，从 `https://example.com` 到 `http://example.com`）

如果您无法访问自己的 API 并收到包含 `Cross-Origin Request Blocked` 的错误消息，则可能需要启用 CORS。

跨源 HTTP 请求可分为两种类型：简单 请求和非简单 请求。

为简单请求启用 CORS

如果满足以下所有条件，则 HTTP 请求为简单 请求：

- 其针对仅允许 `GET`、`HEAD` 和 `POST` 请求的 API 资源发出。
- 如果它是一个 `POST` 方法请求，则它必须包含 `Origin` 标头。
- 请求负载内容类型为 `text/plain`、`multipart/form-data` 或 `application/x-www-form-urlencoded`。
- 请求不包含自定义标头。
- 简单请求的 Mozilla CORS 文档中列出的任何其他要求。

对于简单的跨源 `POST` 方法请求，来自资源的响应需要包含标头 `Access-Control-Allow-Origin: '*'` 或 `Access-Control-Allow-Origin:'origin'`。

所有其他跨源 HTTP 请求均为非简单 请求。

为非简单请求启用 CORS

如果 API 的资源收到非简单请求，则必须根据集成类型启用额外的 CORS 支持。

为非代理集成启用 CORS

对于这些集成，CORS 协议要求浏览器在发送实际请求之前向服务器发送一个预检请求，并等待来自服务器的批准（或对于凭证的请求）。您必须配置您的 API 以向预检请求发送适当的响应。

要创建预检响应，请执行以下操作：

1. 使用模拟集成创建 `OPTIONS` 方法。
2. 将以下响应标头添加到 200 方法响应中：
    - `Access-Control-Allow-Headers`
    - `Access-Control-Allow-Methods`
    - `Access-Control-Allow-Origin`
3. 将集成传递行为设置为 `NEVER`。在这种情况下，将拒绝未映射内容类型的方法请求，并返回“HTTP 415 不支持的媒体类型”响应。有关更多信息，请参阅集成传递行为。
4. 输入响应标头的值。要允许所有来源、所有方法和通用标头，请使用以下标头值：
    - `Access-Control-Allow-Headers: 'Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token'`
    - `Access-Control-Allow-Methods: '*'`
    - `Access-Control-Allow-Origin: '*'`

创建预检请求后，对于至少所有 200 响应，必须为所有启用 CORS 的方法返回 `Access-Control-Allow-Origin: '*'` 或 `Access-Control-Allow-Origin:`*`'origin'`* 标头。

使用 AWS Management Console为非代理集成启用 CORS

您可以使用 AWS Management Console来启用 CORS。API Gateway 会创建 `OPTIONS` 方法，并尝试将 `Access-Control-Allow-Origin` 标头添加到现有的方法集成响应中。这并不总是有效，有时您需要手动修改集成响应，以便为至少所有 200 响应的所有启用 CORS 的方法返回 `Access-Control-Allow-Origin` 标头。

为代理集成启用 CORS 支持

对于 Lambda 代理集成或 HTTP 代理集成，您的后端负责返回 `Access-Control-Allow-Origin`、`Access-Control-Allow-Methods` 和 `Access-Control-Allow-Headers` 标头，因为代理集成不返回集成响应。

以下 Lambda 函数示例返回所需的 CORS 标头：

Node.js

```
export const handler = async (event) => {
    const response = {
        statusCode: 200,
        headers: {
            "Access-Control-Allow-Headers" : "Content-Type",
            "Access-Control-Allow-Origin": "https://www.example.com",
            "Access-Control-Allow-Methods": "OPTIONS,POST,GET"
        },
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

Python 3

```
import json

def lambda_handler(event, context):
    return {
        'statusCode': 200,
        'headers': {
            'Access-Control-Allow-Headers': 'Content-Type',
            'Access-Control-Allow-Origin': 'https://www.example.com',
            'Access-Control-Allow-Methods': 'OPTIONS,POST,GET'
        },
        'body': json.dumps('Hello from Lambda!')
    }
```

主题

- 使用 API Gateway 控制台对资源启用 CORS
- 使用 API Gateway 导入 API 在资源上启用 CORS
- 测试 CORS

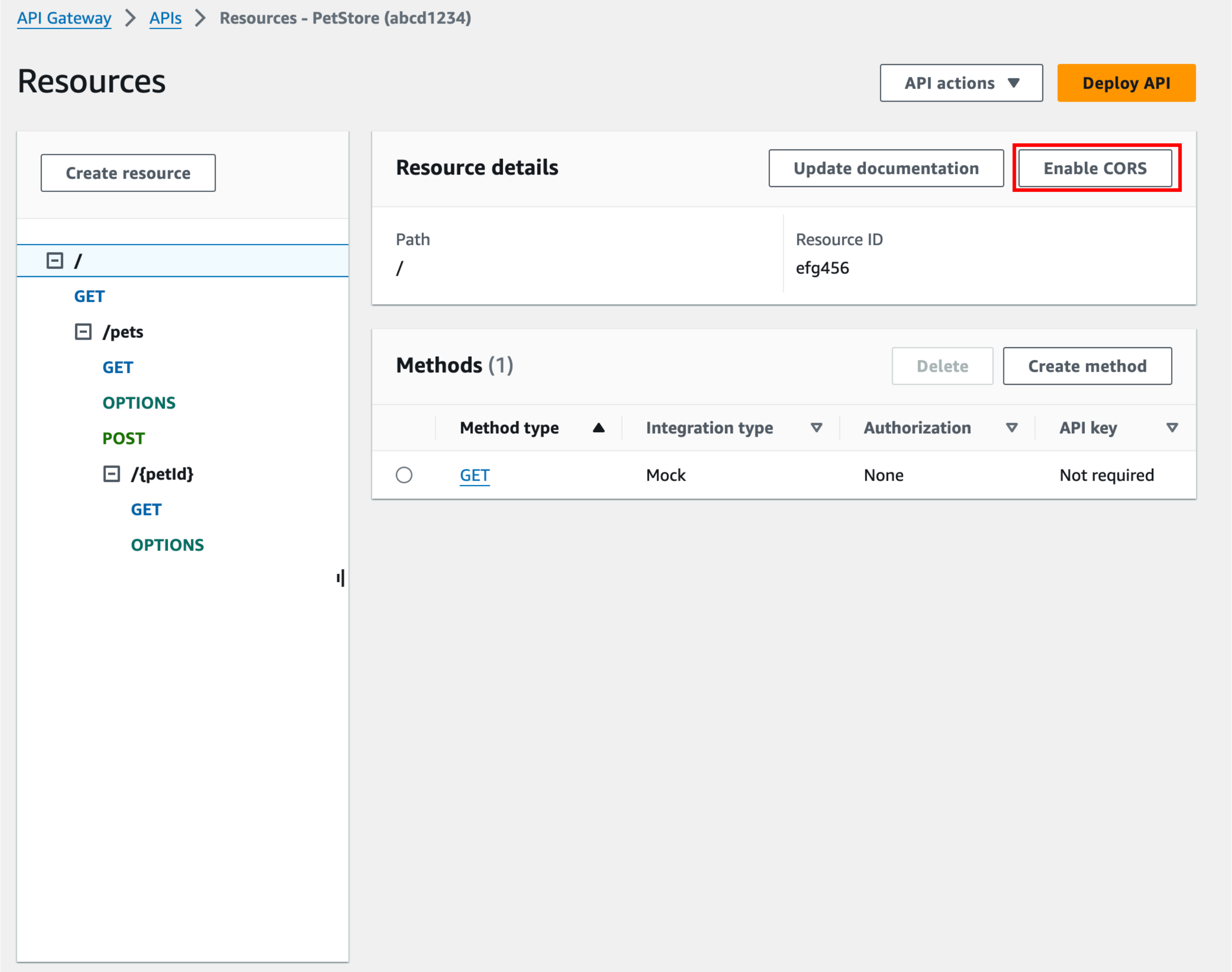
使用 API Gateway 控制台对资源启用 CORS

您可以使用 API Gateway 控制台为已创建的 REST API 资源上的一个或所有方法启用 CORS 支持。启用 COR 支持后，将集成传递行为设置为 `NEVER`。在这种情况下，将拒绝未映射内容类型的方法请求，并返回“HTTP 415 不支持的媒体类型”响应。有关更多信息，请参阅 集成传递行为

> **Important**
>
> 资源可以包含子资源。为某个资源及其方法启用 CORS 支持不会以递归方式为子资源及其方法启用它。

在 REST API 资源上启用 CORS 支持

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 API。
3. 在资源下选择一个资源。
4. 在资源详细信息部分，选择启用 CORS。



Path	Resource ID
/	efg456

	Method type	Integration type	Authorization	API key
○	GET	Mock	None	Not required

5. 在启用 CORS 框中，执行以下操作：

   a. （可选）如果您创建了自定义网关响应并希望为响应启用 CORS 支持，请选择一种网关响应。

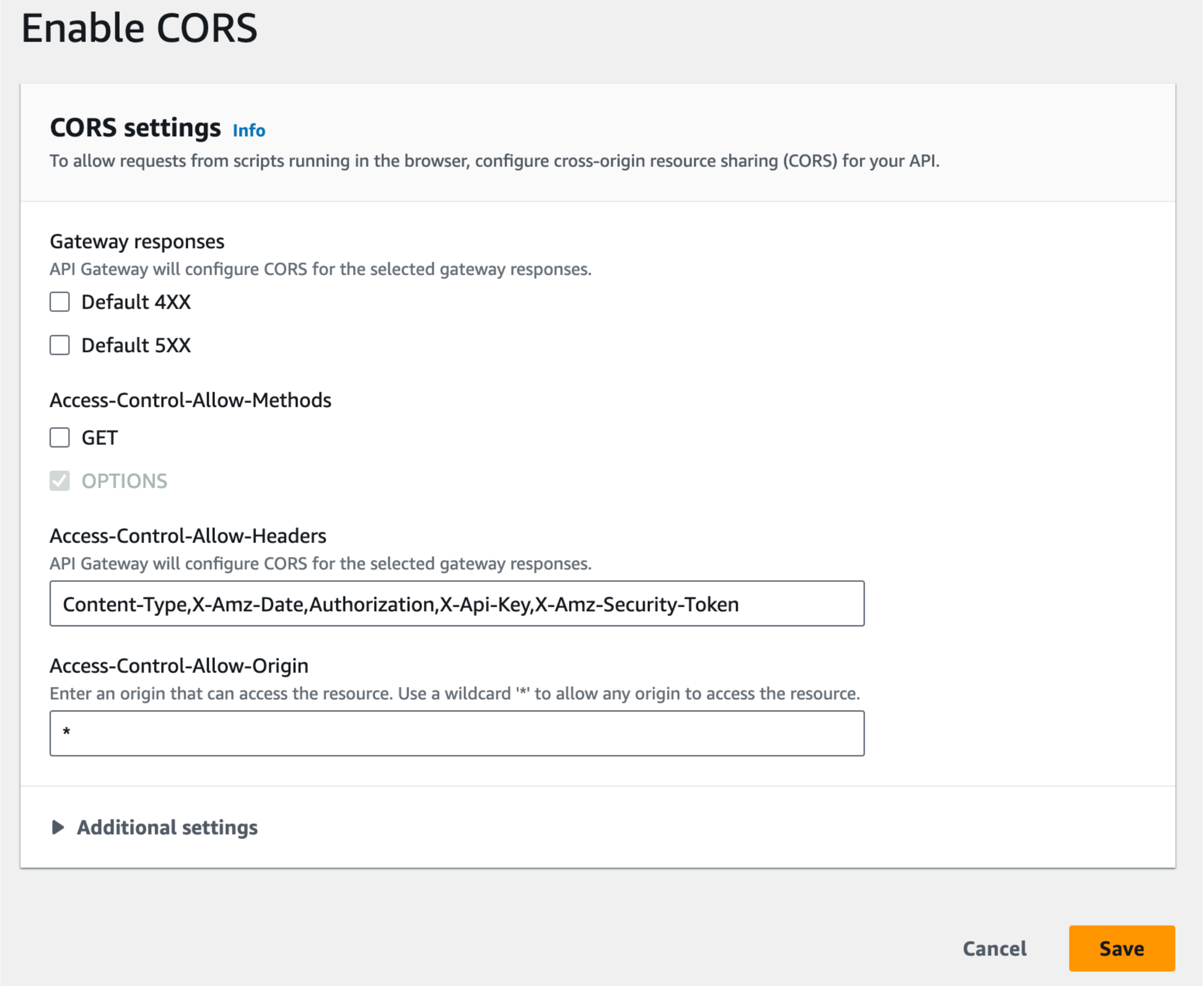
   b. 选择各方法以启用 CORS 支持。`OPTION` 方法必须启用 CORS。

      如果您为某个 ANY 方法启用 CORS 支持，则会为所有方法启用 CORS。

   c. 在 Access-Control-Allow-Headers 输入字段中，输入静态字符串，该字符串是客户端必须在实际资源请求中提交的标头列表，以逗号分隔。使用控制台提供的 `'Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token'` 标头列表，或指定您自己的标头。

   d. 将控制台提供的 `'*'` 的值用作 Access-Control-Allow-Origin 标头值，以允许来自所有源的访问请求，或指定允许访问该资源的源。

e. 选择保存。



Enable CORS

CORS settings Info

To allow requests from scripts running in the browser, configure cross-origin resource sharing (CORS) for your API.

Gateway responses

API Gateway will configure CORS for the selected gateway responses.

☐ Default 4XX

☐ Default 5XX

Access-Control-Allow-Methods

☐ GET

☑ OPTIONS

Access-Control-Allow-Headers

API Gateway will configure CORS for the selected gateway responses.

Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token

Access-Control-Allow-Origin

Enter an origin that can access the resource. Use a wildcard '*' to allow any origin to access the resource.

*

> **Important**
>
> 如果在代理集成中将以上说明应用于 ANY 方法，那么将不会设置任何适用的 CORS 标头。相反，您的后端必须返回适用的 CORS 标头，例如 `Access-Control-Allow-Origin`。

在 GET 方法上启用 CORS 后，如果资源中没有 OPTIONS 方法，则该方法将添加到资源中。OPTIONS 方法的 200 响应会自动配置为返回三个 `Access-Control-Allow-*` 标头，以完成预检握手。此

外，默认情况下，实际 (GET) 方法还会配置为在 200 响应内返回 `Access-Control-Allow-Origin` 标头。对于其他类型的响应，如果您不希望返回 `Cross-origin access` 错误，您将需要手动对其进行配置，以返回带有“*”或特定源的 `Access-Control-Allow-Origin'` 标头。

在您的资源上启用 CORS 支持后，您必须部署或重新部署 API 以使新设置生效。有关更多信息，请参阅 the section called “部署 REST API（控制台）”。

Note

如果按照此过程操作后无法在资源上启用 CORS 支持，我们建议您将您的 CORS 配置与示例 API `/pets` 资源进行比较。要了解如何创建示例 API，请参阅the section called “教程：通过导入示例创建 REST API”。

使用 API Gateway 导入 API 在资源上启用 CORS

如果您使用 API Gateway 导入 API，则可以使用 OpenAPI 文件设置 CORS 支持。您必须先在您的资源中定义可返回所需标头的 `OPTIONS` 方法。

Note

Web 浏览器预计接受 CORS 请求的每个 API 方法中会设置 Access-Control-Allow 标头和 Access-Control-Allow-Origin 标头。此外，某些浏览器首先向同一资源中的 `OPTIONS` 方法发出 HTTP 请求，然后预计收到相同的标头。

`Options` 方法示例

以下示例创建了一个 `OPTIONS` 方法以进行模拟集成。

OpenAPI 3.0

```
/users:
  options:
    summary: CORS support
    description: |
      Enable CORS by returning correct headers
    tags:
    - CORS
    responses:
```

```
      200:
        description: Default response for CORS method
        headers:
          Access-Control-Allow-Origin:
            schema:
              type: "string"
          Access-Control-Allow-Methods:
            schema:
              type: "string"
          Access-Control-Allow-Headers:
            schema:
              type: "string"
        content: {}
    x-amazon-apigateway-integration:
      type: mock
      requestTemplates:
        application/json: "{\"statusCode\": 200}"
      passthroughBehavior: "never"
      responses:
        default:
          statusCode: "200"
          responseParameters:
            method.response.header.Access-Control-Allow-Headers: "'Content-Type,X-Amz-Date,Authorization,X-Api-Key'"
            method.response.header.Access-Control-Allow-Methods: "'*'"
            method.response.header.Access-Control-Allow-Origin: "'*'"
```

OpenAPI 2.0

```
/users:
   options:
      summary: CORS support
      description: |
        Enable CORS by returning correct headers
      consumes:
        - "application/json"
      produces:
        - "application/json"
      tags:
        - CORS
      x-amazon-apigateway-integration:
        type: mock
```

```
          requestTemplates: "{\"statusCode\": 200}"
          passthroughBehavior: "never"
          responses:
            "default":
              statusCode: "200"
              responseParameters:
                method.response.header.Access-Control-Allow-Headers : "'Content-
Type,X-Amz-Date,Authorization,X-Api-Key'"
                method.response.header.Access-Control-Allow-Methods : "'*'"
                method.response.header.Access-Control-Allow-Origin : "'*'"
        responses:
          200:
            description: Default response for CORS method
            headers:
              Access-Control-Allow-Headers:
                type: "string"
              Access-Control-Allow-Methods:
                type: "string"
              Access-Control-Allow-Origin:
                type: "string"
```

在您为资源配置 `OPTIONS` 方法后，可以将所需的标头添加到同一资源中需要接受 CORS 请求的其他方法。

1. 将 Access-Control-Allow-Origin 和标头声明为响应类型。

   OpenAPI 3.0

   ```
   responses:
     200:
       description: Default response for CORS method
       headers:
         Access-Control-Allow-Origin:
           schema:
             type: "string"
         Access-Control-Allow-Methods:
           schema:
             type: "string"
         Access-Control-Allow-Headers:
           schema:
             type: "string"
       content: {}
   ```

OpenAPI 2.0

```
    responses:
        200:
          description: Default response for CORS method
          headers:
            Access-Control-Allow-Headers:
              type: "string"
            Access-Control-Allow-Methods:
              type: "string"
            Access-Control-Allow-Origin:
              type: "string"
```

2. 在 `x-amazon-apigateway-integration` 标签中，为这些标头设置到静态值的映射：

OpenAPI 3.0

```
    responses:
        default:
          statusCode: "200"
          responseParameters:
            method.response.header.Access-Control-Allow-Headers: "'Content-
Type,X-Amz-Date,Authorization,X-Api-Key'"
            method.response.header.Access-Control-Allow-Methods: "'*'"
            method.response.header.Access-Control-Allow-Origin: "'*'"
          responseTemplates:
            application/json: |
              {}
```

OpenAPI 2.0

```
    responses:
          "default":
            statusCode: "200"
            responseParameters:
              method.response.header.Access-Control-Allow-Headers : "'Content-
Type,X-Amz-Date,Authorization,X-Api-Key'"
              method.response.header.Access-Control-Allow-Methods : "'*'"
              method.response.header.Access-Control-Allow-Origin : "'*'"
```

API 示例

以下示例创建一个完整的 API，其中包含一个 `OPTIONS` 方法和一个集成了 `HTTP` 的 `GET` 方法。

OpenAPI 3.0

```
openapi: "3.0.1"
info:
  title: "cors-api"
  description: "cors-api"
  version: "2024-01-16T18:36:01Z"
servers:
- url: "/{basePath}"
  variables:
    basePath:
      default: "/test"
paths:
  /:
    get:
      operationId: "GetPet"
      responses:
        "200":
          description: "200 response"
          headers:
            Access-Control-Allow-Origin:
              schema:
                type: "string"
          content: {}
      x-amazon-apigateway-integration:
        httpMethod: "GET"
        uri: "http://petstore.execute-api.us-east-1.amazonaws.com/petstore/pets"
        responses:
          default:
            statusCode: "200"
            responseParameters:
              method.response.header.Access-Control-Allow-Origin: "'*'"
        passthroughBehavior: "never"
        type: "http"
    options:
      responses:
        "200":
          description: "200 response"
          headers:
            Access-Control-Allow-Origin:
```

```
              schema:
                type: "string"
            Access-Control-Allow-Methods:
              schema:
                type: "string"
            Access-Control-Allow-Headers:
              schema:
                type: "string"
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Empty"
      x-amazon-apigateway-integration:
        responses:
          default:
            statusCode: "200"
            responseParameters:
              method.response.header.Access-Control-Allow-Methods: "'GET,OPTIONS'"
              method.response.header.Access-Control-Allow-Headers: "'Content-Type,X-Amz-Date,Authorization,X-Api-Key'"
              method.response.header.Access-Control-Allow-Origin: "'*'"
        requestTemplates:
          application/json: "{\"statusCode\": 200}"
        passthroughBehavior: "never"
        type: "mock"
components:
  schemas:
    Empty:
      type: "object"
```

OpenAPI 2.0

```
swagger: "2.0"
info:
  description: "cors-api"
  version: "2024-01-16T18:36:01Z"
  title: "cors-api"
basePath: "/test"
schemes:
- "https"
paths:
  /:
    get:
```

```
      operationId: "GetPet"
      produces:
      - "application/json"
      responses:
        "200":
          description: "200 response"
          headers:
            Access-Control-Allow-Origin:
              type: "string"
      x-amazon-apigateway-integration:
        httpMethod: "GET"
        uri: "http://petstore.execute-api.us-east-1.amazonaws.com/petstore/pets"
        responses:
          default:
            statusCode: "200"
            responseParameters:
              method.response.header.Access-Control-Allow-Origin: "'*'"
        passthroughBehavior: "never"
        type: "http"
    options:
      consumes:
      - "application/json"
      produces:
      - "application/json"
      responses:
        "200":
          description: "200 response"
          schema:
            $ref: "#/definitions/Empty"
          headers:
            Access-Control-Allow-Origin:
              type: "string"
            Access-Control-Allow-Methods:
              type: "string"
            Access-Control-Allow-Headers:
              type: "string"
      x-amazon-apigateway-integration:
        responses:
          default:
            statusCode: "200"
            responseParameters:
              method.response.header.Access-Control-Allow-Methods: "'GET,OPTIONS'"
              method.response.header.Access-Control-Allow-Headers: "'Content-Type,X-
Amz-Date,Authorization,X-Api-Key'"
```

```
              method.response.header.Access-Control-Allow-Origin: "'*'"
        requestTemplates:
          application/json: "{\"statusCode\": 200}"
        passthroughBehavior: "never"
        type: "mock"
definitions:
  Empty:
    type: "object"
```

测试 CORS

您可以通过调用 API 并检查响应中的 CORS 标头来测试 API 的 CORS 配置。以下 `curl` 命令将 OPTIONS 请求发送到已部署的 API。

```
curl -v -X OPTIONS https://{restapi_id}.execute-api.{region}.amazonaws.com/{stage_name}
```

```
< HTTP/1.1 200 OK
< Date: Tue, 19 May 2020 00:55:22 GMT
< Content-Type: application/json
< Content-Length: 0
< Connection: keep-alive
< x-amzn-RequestId: a1b2c3d4-5678-90ab-cdef-abc123
< Access-Control-Allow-Origin: *
< Access-Control-Allow-Headers: Content-Type,Authorization,X-Amz-Date,X-Api-Key,X-Amz-
Security-Token
< x-amz-apigw-id: Abcd=
< Access-Control-Allow-Methods: DELETE,GET,HEAD,OPTIONS,PATCH,POST,PUT
```

响应中的 `Access-Control-Allow-Origin`、`Access-Control-Allow-Headers` 和 `Access-Control-Allow-Methods` 标头显示此 API 支持 CORS。有关更多信息，请参阅 为 REST API 资源启用 CORS。

使用 REST API 的二进制媒体类型

在 API Gateway 中，API 请求和响应具有文本或二进制负载。文本负载是 UTF-8 编码的 JSON 字符串。二进制负载是文本负载以外的负载。例如，二进制负载可以是 JPEG 文件、GZip 文件或 XML 文件。支持二进制媒体所需的 API 配置取决于 API 是使用代理集成还是非代理集成。

AWS Lambda 代理集成

要处理 AWS Lambda 代理集成的二进制负载，您必须对函数的响应进行 base64 编码。还必须为 API 配置 binaryMediaTypes。API 的 `binaryMediaTypes` 配置是 API 视为二进制数据的内容类型的列表。示例二进制媒体类型包括 `image/png` 或 `application/octet-stream`。您可以使用通配符 (`*`) 来涵盖多种媒体类型。例如，`*/*` 包括所有内容类型。

有关代码示例，请参阅 the section called “从 Lambda 代理集成返回二进制媒体”。

非代理集成

要处理非代理集成的二进制负载，请将媒体类型添加到 `RestApi` 资源的 binaryMediaTypes 列表中。API 的 `binaryMediaTypes` 配置是 API 视为二进制数据的内容类型的列表。或者，您可以设置 Integration 和 IntegrationResponse 资源上的 contentHandling 属性。`contentHandling` 值可以是 `CONVERT_TO_BINARY`、`CONVERT_TO_TEXT` 或未定义。

根据 `contentHandling` 值，以及响应的 `Content-Type` 标头或传入请求的 `Accept` 标头是否匹配 `binaryMediaTypes` 列表中的某个条目，API Gateway 可以将原始二进制字节编码为 Base64 编码的字符串，将 Base64 编码的字符串解码回其原始字节，或者直接传递正文而不作修改。

您必须按如下方式配置 API，以对 API Gateway 中 API 的二进制负载提供支持：

- 将所需的二进制媒体类型添加到 RestApi 资源的 `binaryMediaTypes` 列表中。如果未定义此属性和 `contentHandling` 属性，会将负载作为 UTF-8 编码的 JSON 字符串进行处理。
- 解决 Integration 资源的 `contentHandling` 属性。
  - 要将请求负载从 base64 编码的字符串转换为其二进制 blob，请将此属性设置为 `CONVERT_TO_BINARY`。
  - 要将请求负载从二进制 blob 转换为 base64 编码的字符串，请将此属性设置为 `CONVERT_TO_TEXT`。
  - 要在不修改的情况下传递负载，请将此属性保留为未定义。要在不修改的情况下传递二进制负载，还必须确保 `Content-Type` 与其中一个 `binaryMediaTypes` 条目匹配，并且已为 API 启用传递行为。
- 设置 IntegrationResponse 资源的 `contentHandling` 属性。`contentHandling` 属性、客户端请求中的 `Accept` 标头以及 API 的 `binaryMediaTypes` 组合决定了 API Gateway 如何处理内容类型转换。有关详细信息，请参阅the section called “API Gateway 中的内容类型转换”。

> **Important**
>
> 如果某个请求在其 `Accept` 标头中包含多个媒体类型，API Gateway 将只接受第一个 `Accept` 媒体类型。如果无法控制 `Accept` 媒体类型的顺序并且二进制内容的媒体类型不是列表中的第一个，则添加 API 的 `binaryMediaTypes` 列表中的第一个 `Accept` 媒体类型。API Gateway 将以二进制形式处理此列表中的所有内容类型。
> 例如，要在浏览器中使用 `<img>` 元素发送 JPEG 文件，该浏览器可能会在请求中发送 `Accept:image/webp,image/*,*/*;q=0.8`。将 `image/webp` 添加到 `binaryMediaTypes` 列表后，终端节点将收到二进制形式的 JPEG 文件。

有关 API Gateway 如何处理文本和二进制负载的详细信息，请参阅 API Gateway 中的内容类型转换。

API Gateway 中的内容类型转换

API 的 `binaryMediaTypes`、客户端请求中的标头和集成 `contentHandling` 属性的组合决定了 API Gateway 对负载编码的方式。

下表显示了 API Gateway 如何转换某个请求的 `Content-Type` 标头的特定配置的请求负载、RestApi 资源的 `binaryMediaTypes` 列表以及 Integration 资源的 `contentHandling` 属性值。

API Gateway 中的 API 请求内容类型转换

方法请求负载	请求 **`Content-Type`** 标头	**`binaryMediaTypes`**	**`contentHandling`**	集成请求负载
文本数据	任意数据类型	未定义	未定义	UTF8 编码的字符串
文本数据	任意数据类型	未定义	`CONVERT_TO_BINARY`	Base64 解码的二进制 blob
文本数据	任意数据类型	未定义	`CONVERT_TO_TEXT`	UTF8 编码的字符串
文本数据	文本数据类型	包含匹配的媒体类型的集合	未定义	文本数据
文本数据	文本数据类型	包含匹配的媒体类型的集合	`CONVERT_TO_BINARY`	Base64 解码的二进制 blob

方法请求负载	请求 `Content-Type` 标头	`binaryMediaTypes`	`contentHandling`	集成请求负载
文本数据	文本数据类型	包含匹配的媒体类型的集合	`CONVERT_TO_TEXT`	文本数据
二进制数据	二进制数据类型	包含匹配的媒体类型的集合	未定义	二进制数据
二进制数据	二进制数据类型	包含匹配的媒体类型的集合	`CONVERT_TO_BINARY`	二进制数据
二进制数据	二进制数据类型	包含匹配的媒体类型的集合	`CONVERT_TO_TEXT`	Base64 编码的字符串

下表显示了 API Gateway 如何转换某个请求的 `Accept` 标头的特定配置的响应负载、RestApi 资源的 `binaryMediaTypes` 列表以及 IntegrationResponse 资源的 `contentHandling` 属性值。

⚠ Important

如果某个请求在其 `Accept` 标头中包含多个媒体类型，API Gateway 将只接受第一个 `Accept` 媒体类型。如果无法控制 `Accept` 媒体类型的顺序并且二进制内容的媒体类型不是列表中的第一个，则添加 API 的 `binaryMediaTypes` 列表中的第一个 `Accept` 媒体类型。API Gateway 将以二进制形式处理此列表中的所有内容类型。

例如，要在浏览器中使用 `<img>` 元素发送 JPEG 文件，该浏览器可能会在请求中发送 `Accept:image/webp,image/*,*/*;q=0.8`。将 `image/webp` 添加到 `binaryMediaTypes` 列表后，终端节点将收到二进制形式的 JPEG 文件。

API Gateway 响应内容类型转换

集成响应负载	请求 `Accept` 标头	`binaryMediaTypes`	`contentHandling`	方法响应负载
文本或二进制数据	文本类型	未定义	未定义	UTF8 编码的字符串

集成响应负载	请求 `Accept` 标头	`binaryMediaTypes`	`contentHandling`	方法响应负载
文本或二进制数据	文本类型	未定义	`CONVERT_TO_BINARY`	Base64 解码的 blob
文本或二进制数据	文本类型	未定义	`CONVERT_TO_TEXT`	UTF8 编码的字符串
文本数据	文本类型	包含匹配的媒体类型的集合	未定义	文本数据
文本数据	文本类型	包含匹配的媒体类型的集合	`CONVERT_TO_BINARY`	Base64 解码的 blob
文本数据	文本类型	包含匹配的媒体类型的集合	`CONVERT_TO_TEXT`	UTF8 编码的字符串
文本数据	二进制类型	包含匹配的媒体类型的集合	未定义	Base64 解码的 blob
文本数据	二进制类型	包含匹配的媒体类型的集合	`CONVERT_TO_BINARY`	Base64 解码的 blob
文本数据	二进制类型	包含匹配的媒体类型的集合	`CONVERT_TO_TEXT`	UTF8 编码的字符串
二进制数据	文本类型	包含匹配的媒体类型的集合	未定义	Base64 编码的字符串
二进制数据	文本类型	包含匹配的媒体类型的集合	`CONVERT_TO_BINARY`	二进制数据
二进制数据	文本类型	包含匹配的媒体类型的集合	`CONVERT_TO_TEXT`	Base64 编码的字符串
二进制数据	二进制类型	包含匹配的媒体类型的集合	未定义	二进制数据

集成响应负载	请求 **`Accept`** 标头	**`binaryMediaTypes`**	**`contentHandling`**	方法响应负载
二进制数据	二进制类型	包含匹配的媒体类型的集合	`CONVERT_TO_BINARY`	二进制数据
二进制数据	二进制类型	包含匹配的媒体类型的集合	`CONVERT_TO_TEXT`	Base64 编码的字符串

将文本负载转换为二进制 blob 时，API Gateway 假定文本数据是 Base64 编码的字符串，并将二进制数据作为 Base64 解码的 blob 输出。如果转换失败，它将返回一个 `500` 响应，指示出现 API 配置错误。尽管必须对 API 启用传递行为，但您不用提供此类转换的映射模板。

将二进制负载转换为文本字符串时，API Gateway 始终对二进制数据应用 Base64 编码。您可以定义此类负载的映射模板，但只能通过 `$input.body` 访问映射模板中的 Base64 编码字符串，如映射模板示例的以下摘录中所示。

```
{
    "data": "$input.body"
}
```

要直接传递二进制负载而不作修改，必须对 API 启用传递行为。

使用 API Gateway 控制台启用二进制支持

本节说明如何使用 API Gateway 控制台启用二进制支持。例如，我们使用一个与 Amazon S3 集成的 API。我们的任务重点是设置受支持的媒体类型并指定如何处理负载。有关如何创建与 Amazon S3 集成的 API 的详细信息，请参阅教程：在 API Gateway 中创建 REST API 作为 Amazon S3 代理。

使用 API Gateway 控制台启用二进制支持

1. 设置 API 的二进制媒体类型：
    a. 创建新 API 或选择现有 API。在本例中，我们将 API 命名为 `FileMan`。
    b. 在主导航面板中所选的 API 之下，选择 API 设置。
    c. 在 API 设置窗格中的二进制媒体类型部分，选择管理媒体类型。
    d. 选择添加二进制媒体类型。

  e. 在文本输入字段中输入所需的媒体类型，例如 **`image/png`**。如果需要，请重复此步骤，添加更多媒体类型。要支持所有二进制媒体类型，请指定 `*/*`。
  f. 选择 Save changes（保存更改）。

2. 设置如何针对 API 方法处理消息负载：

  a. 创建新资源或选择 API 中的现有资源。在本例中，我们使用 `/{folder}/{item}` 资源。
  b. 对该资源创建新方法或选择一个现有方法。例如，我们使用与 Amazon S3 中的 `GET /{folder}/{item}` 操作集成的 `Object GET` 方法。
  c. 对于内容处理，选择一个选项。

Action type

○ Use action name

● Use path override

Path override - *optional*

{bucket}/{object}

Execution role

arn:aws:iam::444455556666:role/s3-ApiGatewayS3ReadOnlyRole

Credential cache

Do not add caller credentials to cache key

Content handling

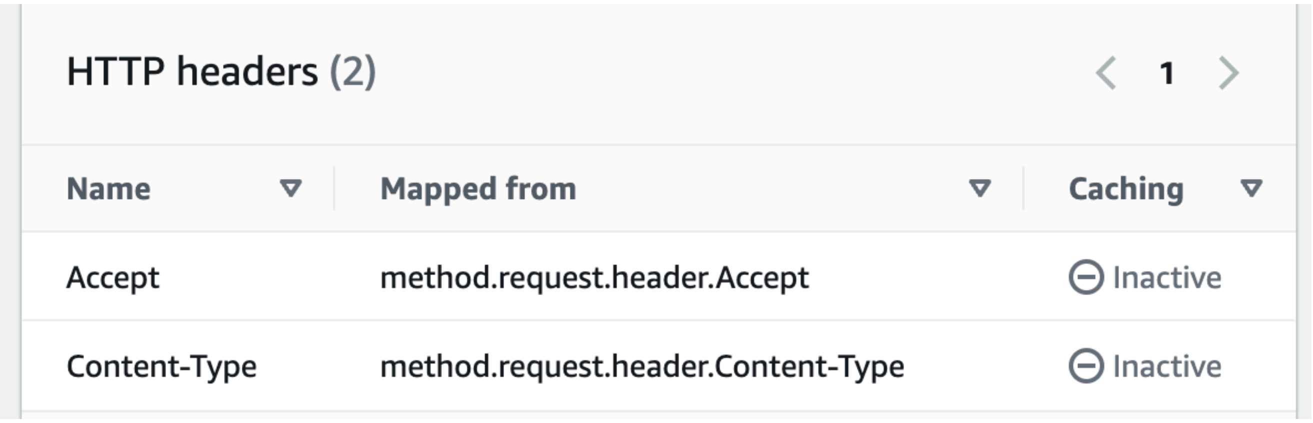
Passthrough

如果不想在客户端和后端接受相同的二进制格式时转换正文，则选择传递。当存在后端要求将二进制请求负载作为 JSON 属性传入等情况时，选择转换为文本以将二进制正文转换为 Base64 编码的字符串。当客户端提交 Base64 编码的字符串且后端需要原始二进制格式，或者当端点返回 Base64 编码的字符串且客户端只接受二进制输出时，选择转换为二进制。

  d. 对于请求正文传递，选择当未定义模板时（推荐），以在请求正文上启用传递行为。

  您也可以选择从不。这意味着 API 将拒绝其内容类型没有映射模板的数据。

  e. 在集成请求中保留传入请求的 `Accept` 标头。如果您已将 `contentHandling` 设置为 `passthrough` 并且希望在运行时覆盖该设置，则应执行此操作。



HTTP headers (2)

Name	Mapped from	Caching
Accept	method.request.header.Accept	⊖ Inactive
Content-Type	method.request.header.Content-Type	⊖ Inactive

f. 转换为文本时，请定义映射模板，以将 Base64 编码的二进制数据转换为所需格式。

   以下是用于转换为文本的映射模板的示例：

```
{
  "operation": "thumbnail",
  "base64Image": "$input.body"
}
```

   此映射模板的格式取决于输入的端点要求。

g. 选择保存。

使用 API Gateway REST API 启用二进制支持

以下任务演示如何使用 API Gateway REST API 调用来启用二进制支持。

主题

- 向 API 添加和更新受支持的二进制媒体类型
- 配置请求负载转换
- 配置响应负载转换
- 将二进制数据转换为文本数据
- 将文本数据转换为二进制负载
- 传递二进制负载

向 API 添加和更新受支持的二进制媒体类型

要允许 API Gateway 支持新的二进制媒体类型，必须将该二进制媒体类型添加到 `RestApi` 资源的 `binaryMediaTypes` 列表中。例如，要让 API Gateway 处理 JPEG 图像，应向 `RestApi` 资源提交 `PATCH` 请求：

```
PATCH /restapis/<restapi_id>

{
  "patchOperations" : [ {
    "op" : "add",
    "path" : "/binaryMediaTypes/image~1jpeg"
  }
 ]
}
```

属于 `image/jpeg` 属性值一部分的 MIME 类型规范 `path` 将转义为 `image~1jpeg`。

要更新受支持的二进制媒体类型，请替换或删除 `binaryMediaTypes` 资源的 `RestApi` 列表中的该媒体类型。例如，要将二进制支持从 JPEG 文件更改为原始字节，请向 `RestApi` 资源提交 `PATCH` 请求，如下所示：

```
PATCH /restapis/<restapi_id>

{
  "patchOperations" : [{
    "op" : "replace",
    "path" : "/binaryMediaTypes/image~1jpeg",
    "value" : "application/octet-stream"
  },
  {
    "op" : "remove",
    "path" : "/binaryMediaTypes/image~1jpeg"
  }]
}
```

配置请求负载转换

如果终端节点需要二进制输入，则将 `contentHandling` 资源的 `Integration` 属性设置为 `CONVERT_TO_BINARY`。为此，请提交 `PATCH` 请求，如下所示：

```
PATCH /restapis/<restapi_id>/resources/<resource_id>/methods/<http_method>/integration
```

```
{
  "patchOperations" : [ {
    "op" : "replace",
    "path" : "/contentHandling",
    "value" : "CONVERT_TO_BINARY"
  }]
}
```

配置响应负载转换

如果客户端接受二进制 blob 形式的结果而不是从终端节点返回的 Base64 编码的负载，则将 `IntegrationResponse` 资源的 `contentHandling` 属性设置为 `CONVERT_TO_BINARY`。为此，请提交 PATCH 请求，如下所示：

```
PATCH /restapis/<restapi_id>/resources/<resource_id>/methods/<http_method>/integration/
responses/<status_code>

{
  "patchOperations" : [ {
    "op" : "replace",
    "path" : "/contentHandling",
    "value" : "CONVERT_TO_BINARY"
  }]
}
```

将二进制数据转换为文本数据

要通过 API Gateway 将二进制数据作为输入的 JSON 属性发送到 AWS Lambda 或 Kinesis，请执行以下操作：

1. 通过将新的二进制媒体类型 `application/octet-stream` 添加到 API 的 `binaryMediaTypes` 列表，启用 API 的二进制负载支持。

   ```
   PATCH /restapis/<restapi_id>

   {
     "patchOperations" : [ {
       "op" : "add",
       "path" : "/binaryMediaTypes/application~1octet-stream"
     }
    ]
   ```

```
}
```

2. 在 `Integration` 资源的 `contentHandling` 属性上设置 `CONVERT_TO_TEXT`，并提供映射模板以将二进制数据的 Base64 编码字符串分配给 JSON 属性。在以下示例中，JSON 属性为 body，并且 `$input.body` 保存 Base64 编码的字符串。

```
PATCH /restapis/<restapi_id>/resources/<resource_id>/methods/<http_method>/
integration

{
  "patchOperations" : [
    {
      "op" : "replace",
      "path" : "/contentHandling",
      "value" : "CONVERT_TO_TEXT"
    },
    {
      "op" : "add",
      "path" : "/requestTemplates/application~1octet-stream",
      "value" : "{\"body\": \"$input.body\"}"
    }
  ]
}
```

将文本数据转换为二进制负载

假设 Lambda 函数返回一个 Base64 编码字符串形式的图像文件。要通过 API Gateway 将此二进制输出传递到客户端，请执行以下操作：

1. 通过添加二进制媒体类型 `binaryMediaTypes` (如果该类型尚不在列表中) 来更新 API 的 `application/octet-stream` 列表。

```
PATCH /restapis/<restapi_id>

{
  "patchOperations" : [ {
    "op" : "add",
    "path" : "/binaryMediaTypes/application~1octet-stream",
  }]
}
```

2. 将 `contentHandling` 资源的 `Integration` 属性设置为 `CONVERT_TO_BINARY`。请勿定义映射模板。如果不定义映射模板，API Gateway 会调用传递模板，以将 Base64 解码的二进制 blob 作为图像文件返回给客户端。

```
PATCH /restapis/<restapi_id>/resources/<resource_id>/methods/<http_method>/
integration/responses/<status_code>

{
  "patchOperations" : [
    {
      "op" : "replace",
      "path" : "/contentHandling",
      "value" : "CONVERT_TO_BINARY"
    }
  ]
}
```

传递二进制负载

要使用 API Gateway 将图像存储在 Amazon S3 存储桶中，请执行以下操作：

1. 通过添加二进制媒体类型 `binaryMediaTypes`（如果该类型尚不在列表中）来更新 API 的 `application/octet-stream` 列表。

```
PATCH /restapis/<restapi_id>

{
  "patchOperations" : [ {
    "op" : "add",
    "path" : "/binaryMediaTypes/application~1octet-stream"
  }
 ]
}
```

2. 在 `contentHandling` 资源的 `Integration` 属性上，设置 `CONVERT_TO_BINARY`。将 `WHEN_NO_MATCH` 设置为 `passthroughBehavior` 属性值，而不定义映射模板。这使 API Gateway 能够调用传递模板。

```
PATCH /restapis/<restapi_id>/resources/<resource_id>/methods/<http_method>/
integration
```

```
{
  "patchOperations" : [
    {
      "op" : "replace",
      "path" : "/contentHandling",
      "value" : "CONVERT_TO_BINARY"
    },
    {
      "op" : "replace",
      "path" : "/passthroughBehaviors",
      "value" : "WHEN_NO_MATCH"
    }
  ]
}
```

导入和导出内容编码

要导入 RestApi 上的 `binaryMediaTypes` 列表，请使用 API 的 OpenAPI 定义文件的以下 API Gateway 扩展。该扩展也用于导出 API 设置。

- x-amazon-apigateway-binary-media-types 属性

要导入和导出 `Integration` 或 `IntegrationResponse` 资源的 `contentHandling` 属性值，请使用 OpenAPI 定义的以下 API Gateway 扩展：

- x-amazon-apigateway-integration 对象
- x-amazon-apigateway-integration.response 对象

从 Lambda 代理集成返回二进制媒体

要从 AWS Lambda 代理集成返回二进制媒体，请对来自 Lambda 函数的响应进行 base64 编码。还必须配置 API 的二进制媒体类型。负载大小限制为 10 MB。

> **Note**
>
> 要使用 Web 浏览器调用具有此示例集成的 API，请将 API 的二进制媒体类型设置为 `*/*`。API Gateway 使用来自客户端的第一个 `Accept` 标头来确定响应是否应返回二进制媒体。要在无法

控制 `Accept` 标头值（如浏览器中的请求）的顺序时返回二进制媒体，请将 API 的二进制媒体类型设置为 `*/*`（对于所有内容类型）。

以下 Lambda 函数示例可以从 Amazon S3 返回二进制图像或将文本返回到客户端。该函数的响应包括一个 `Content-Type` 标头，用于向客户端指示它返回的数据类型。函数有条件地设置其响应中的 `isBase64Encoded` 属性，具体取决于它返回的数据类型。

Node.js

```
import { S3Client, GetObjectCommand } from "@aws-sdk/client-s3"

const client = new S3Client({region: 'us-east-2'});

export const handler = async (event) => {

  var randomint = function(max) {
    return Math.floor(Math.random() * max);
  }
  var number = randomint(2);
  if (number == 1){
    const input = {
      "Bucket" : "bucket-name",
      "Key" : "image.png"
      }
    try {
      const command = new GetObjectCommand(input)
      const response = await client.send(command);
      var str = await response.Body.transformToByteArray();
    } catch (err) {
      console.error(err);
    }
    const base64body = Buffer.from(str).toString('base64');
    return {
      'headers': { "Content-Type": "image/png" },
      'statusCode': 200,
      'body': base64body,
      'isBase64Encoded': true
      }
    } else {
        return {
        'headers': { "Content-Type": "text/html" },
```

```
            'statusCode': 200,
            'body': "<h1>This is text</h1>",
            }
        }
}
```

Python

```
import base64
import boto3
import json
import random

s3 = boto3.client('s3')

def lambda_handler(event, context):
    number = random.randint(0,1)
    if number == 1:
        response = s3.get_object(
            Bucket='bucket-name',
            Key='image.png',
        )
        image = response['Body'].read()
        return {
            'headers': { "Content-Type": "image/png" },
            'statusCode': 200,
            'body': base64.b64encode(image).decode('utf-8'),
            'isBase64Encoded': True
        }
    else:
        return {
            'headers': { "Content-type": "text/html" },
            'statusCode': 200,
            'body': "<h1>This is text</h1>",
        }
```

要了解有关二进制媒体类型的更多信息，请参阅 使用 REST API 的二进制媒体类型。

通过 API Gateway API 访问 Amazon S3 中的二进制文件

以下示例显示了用于访问 Amazon S3 中图像的 OpenAPI 文件如何从 Amazon S3 下载图像以及如何将图像上传到 Amazon S3。

主题

- 用于访问 Amazon S3 中图像的示例 API 的 OpenAPI 文件
- 从 Amazon S3 下载图像
- 将图像上传到 Amazon S3

用于访问 Amazon S3 中图像的示例 API 的 OpenAPI 文件

以下 OpenAPI 文件显示了一个示例 API，阐明了如何从 Amazon S3 下载图像文件以及如何将图像文件上传到 Amazon S3。此 API 公开了下载和上传指定图像文件所用的 `GET /s3?key={file-name}` 和 `PUT /s3?key={file-name}` 方法。`GET` 方法按照所提供的映射模板，在一个 200 OK 响应中返回 Base64 编码字符串形式的图像文件作为 JSON 输出的一部分。PUT 方法将原始二进制 blob 作为输入，并返回一个负载为空的 200 OK 响应。

OpenAPI 3.0

```
{
   "openapi": "3.0.0",
   "info": {
      "version": "2016-10-21T17:26:28Z",
      "title": "ApiName"
   },
   "paths": {
      "/s3": {
         "get": {
            "parameters": [
               {
                  "name": "key",
                  "in": "query",
                  "required": false,
                  "schema": {
                     "type": "string"
                  }
               }
            ],
            "responses": {
               "200": {
                  "description": "200 response",
                  "content": {
                     "application/json": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
```

```
                        }
                     }
                  }
               },
               "500": {
                  "description": "500 response"
               }
            },
            "x-amazon-apigateway-integration": {
               "credentials": "arn:aws:iam::123456789012:role/binarySupportRole",
               "responses": {
                  "default": {
                     "statusCode": "500"
                  },
                  "2\\d{2}": {
                     "statusCode": "200"
                  }
               },
               "requestParameters": {
                  "integration.request.path.key": "method.request.querystring.key"
               },
               "uri": "arn:aws:apigateway:us-west-2:s3:path/{key}",
               "passthroughBehavior": "when_no_match",
               "httpMethod": "GET",
               "type": "aws"
            }
         },
         "put": {
            "parameters": [
               {
                  "name": "key",
                  "in": "query",
                  "required": false,
                  "schema": {
                     "type": "string"
                  }
               }
            ],
            "responses": {
               "200": {
                  "description": "200 response",
                  "content": {
                     "application/json": {
                        "schema": {
```

```
                           "$ref": "#/components/schemas/Empty"
                        }
                     },
                     "application/octet-stream": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
                        }
                     }
                  }
               },
               "500": {
                  "description": "500 response"
               }
            },
            "x-amazon-apigateway-integration": {
               "credentials": "arn:aws:iam::123456789012:role/binarySupportRole",
               "responses": {
                  "default": {
                     "statusCode": "500"
                  },
                  "2\\d{2}": {
                     "statusCode": "200"
                  }
               },
               "requestParameters": {
                  "integration.request.path.key": "method.request.querystring.key"
               },
               "uri": "arn:aws:apigateway:us-west-2:s3:path/{key}",
               "passthroughBehavior": "when_no_match",
               "httpMethod": "PUT",
               "type": "aws",
               "contentHandling": "CONVERT_TO_BINARY"
            }
         }
      }
   },
   "x-amazon-apigateway-binary-media-types": [
      "application/octet-stream",
      "image/jpeg"
   ],
   "servers": [
      {
         "url": "https://abcdefghi.execute-api.us-east-1.amazonaws.com/{basePath}",
         "variables": {
```

```
            "basePath": {
              "default": "/v1"
            }
          }
        }
      ],
      "components": {
        "schemas": {
          "Empty": {
            "type": "object",
            "title": "Empty Schema"
          }
        }
      }
}
```

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-10-21T17:26:28Z",
    "title": "ApiName"
  },
  "host": "abcdefghi.execute-api.us-east-1.amazonaws.com",
  "basePath": "/v1",
  "schemes": [
    "https"
  ],
  "paths": {
    "/s3": {
      "get": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "key",
            "in": "query",
            "required": false,
            "type": "string"
          }
        ],
```

```
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          },
          "500": {
            "description": "500 response"
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/binarySupportRole",
          "responses": {
            "default": {
              "statusCode": "500"
            },
            "2\\d{2}": {
              "statusCode": "200"            }
          },
          "requestParameters": {
            "integration.request.path.key": "method.request.querystring.key"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{key}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "GET",
          "type": "aws"
        }
      },
      "put": {
        "produces": [
          "application/json", "application/octet-stream"
        ],
        "parameters": [
          {
            "name": "key",
            "in": "query",
            "required": false,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
```

```
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          },
          "500": {
            "description": "500 response"
          }
        },
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/binarySupportRole",
          "responses": {
            "default": {
              "statusCode": "500"
            },
            "2\\d{2}": {
              "statusCode": "200"
            }
          },
          "requestParameters": {
            "integration.request.path.key": "method.request.querystring.key"
          },
          "uri": "arn:aws:apigateway:us-west-2:s3:path/{key}",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "PUT",
          "type": "aws",
          "contentHandling" : "CONVERT_TO_BINARY"
        }
      }
    }
  },
  "x-amazon-apigateway-binary-media-types" : ["application/octet-stream", "image/
jpeg"],
  "definitions": {
    "Empty": {
      "type": "object",
      "title": "Empty Schema"
    }
  }
}
```

从 Amazon S3 下载图像

从 Amazon S3 下载二进制 blob 形式的图像文件 (`image.jpg`)：

```
GET /v1/s3?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/octet-stream
```

成功的响应类似于以下示例：

```
200 OK HTTP/1.1

[raw bytes]
```

因为 `Accept` 标头设置为二进制媒体类型 `application/octet-stream`，并且您对 API 启用了二进制支持，所以返回原始字节。

要从 Amazon S3 下载采用 JSON 属性格式的 Base64 编码字符串形式的图像文件 (`image.jpg`)，请向 200 集成响应添加一个响应模板，如以下粗体 OpenAPI 定义块所示：

```
        "x-amazon-apigateway-integration": {
          "credentials": "arn:aws:iam::123456789012:role/binarySupportRole",
          "responses": {
            "default": {
              "statusCode": "500"
            },
            "2\\d{2}": {
              "statusCode": "200",
              "responseTemplates": {
                "application/json": "{\n   \"image\": \"$input.body\"\n}"
              }
            }
          },
```

下载图像文件的请求如下所示：

```
GET /v1/s3?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/json
```

成功响应如下所示：

```
200 OK HTTP/1.1
```

```
{
  "image": "W3JhdyBieXRlc10="
}
```

将图像上传到 Amazon S3

将二进制 blob 形式的图像文件 (`image.jpg`) 上传到 Amazon S3：

```
PUT /v1/s3?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/octet-stream
Accept: application/json

[raw bytes]
```

成功响应如下所示：

```
200 OK HTTP/1.1
```

将 Base64 编码字符串形式的图像文件 (`image.jpg`) 上传到 Amazon S3：

```
PUT /v1/s3?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/json

W3JhdyBieXRlc10=
```

输入负载必须是 Base64 编码的字符串，因为 `Content-Type` 标头值设置为 `application/json`。成功响应如下所示：

```
200 OK HTTP/1.1
```

使用 API Gateway API 访问 Lambda 中的二进制文件

以下 OpenAPI 示例展示了如何通过 API Gateway API 访问 AWS Lambda 中的二进制文件。此 API 公开了用于下载和上传指定图像文件的 `GET /lambda?key={file-name}` 和 `PUT /lambda?key={file-name}` 方法。`GET` 方法按照所提供的映射模板，在一个 200 OK 响应中返回 Base64 编码字符串形式的图像文件作为 JSON 输出的一部分。PUT 方法将原始二进制 blob 作为输入，并返回一个负载为空的 200 OK 响应。

您创建 API 调用的 Lambda 函数，该函数必须返回 `Content-Type` 标头为 `application/json` 的 base64 编码字符串。

主题

- 用于访问 Lambda 中图像的示例 API 的 OpenAPI 文件
- 从 Lambda 下载图像
- 将图像上传到 Lambda

用于访问 Lambda 中图像的示例 API 的 OpenAPI 文件

以下 OpenAPI 文件显示了一个 API 示例，阐明了如何从 Lambda 下载图像文件以及如何将图像文件上传到 Lambda。

OpenAPI 3.0

```
{
   "openapi": "3.0.0",
   "info": {
      "version": "2016-10-21T17:26:28Z",
      "title": "ApiName"
   },
   "paths": {
      "/lambda": {
         "get": {
            "parameters": [
               {
                  "name": "key",
                  "in": "query",
                  "required": false,
                  "schema": {
                     "type": "string"
                  }
               }
            ],
            "responses": {
               "200": {
                  "description": "200 response",
                  "content": {
                     "application/json": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
```

```
                        }
                     }
                  }
               },
               "500": {
                  "description": "500 response"
               }
            },
            "x-amazon-apigateway-integration": {
               "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:123456789012:function:image/invocations",
               "type": "AWS",
               "credentials": "arn:aws:iam::123456789012:role/Lambda",
               "httpMethod": "POST",
               "requestTemplates": {
                  "application/json": "{\n   \"imageKey\":
 \"$input.params('key')\"\n}"
               },
               "responses": {
                  "default": {
                     "statusCode": "500"
                  },
                  "2\\d{2}": {
                     "statusCode": "200",
                     "responseTemplates": {
                        "application/json": "{\n   \"image\": \"$input.body\"\n}"
                     }
                  }
               }
            }
         },
         "put": {
            "parameters": [
               {
                  "name": "key",
                  "in": "query",
                  "required": false,
                  "schema": {
                     "type": "string"
                  }
               }
            ],
            "responses": {
               "200": {
```

```
                  "description": "200 response",
                  "content": {
                     "application/json": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
                        }
                     },
                     "application/octet-stream": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
                        }
                     }
                  }
               },
               "500": {
                  "description": "500 response"
               }
            },
            "x-amazon-apigateway-integration": {
               "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:123456789012:function:image/invocations",
               "type": "AWS",
               "credentials": "arn:aws:iam::123456789012:role/Lambda",
               "httpMethod": "POST",
               "contentHandling": "CONVERT_TO_TEXT",
               "requestTemplates": {
                  "application/json": "{\n   \"imageKey\": \"$input.params('key')\",
 \"image\": \"$input.body\"\n}"
               },
               "responses": {
                  "default": {
                     "statusCode": "500"
                  },
                  "2\\d{2}": {
                     "statusCode": "200"
                  }
               }
            }
         }
      }
   },
   "x-amazon-apigateway-binary-media-types": [
      "application/octet-stream",
      "image/jpeg"
```

```
   ],
   "servers": [
      {
         "url": "https://abcdefghi.execute-api.us-east-1.amazonaws.com/{basePath}",
         "variables": {
            "basePath": {
              "default": "/v1"
            }
         }
      }
   ],
   "components": {
      "schemas": {
         "Empty": {
            "type": "object",
            "title": "Empty Schema"
         }
      }
   }
}
```

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-10-21T17:26:28Z",
    "title": "ApiName"
  },
  "host": "abcdefghi.execute-api.us-east-1.amazonaws.com",
  "basePath": "/v1",
  "schemes": [
    "https"
  ],
  "paths": {
    "/lambda": {
      "get": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "key",
```

```
            "in": "query",
            "required": false,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          },
          "500": {
            "description": "500 response"
          }
        },
        "x-amazon-apigateway-integration": {
          "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:image/invocations",
          "type": "AWS",
          "credentials": "arn:aws:iam::123456789012:role/Lambda",
          "httpMethod": "POST",
          "requestTemplates": {
            "application/json": "{\n   \"imageKey\": \"$input.params('key')\"\n}"
          },
          "responses": {
            "default": {
              "statusCode": "500"
            },
            "2\\d{2}": {
              "statusCode": "200",
              "responseTemplates": {
                "application/json": "{\n   \"image\": \"$input.body\"\n}"
              }
            }
          }
        }
      },
      "put": {
        "produces": [
          "application/json", "application/octet-stream"
        ],
        "parameters": [
          {
```

```
            "name": "key",
            "in": "query",
            "required": false,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
          },
          "500": {
            "description": "500 response"
          }
        },
        "x-amazon-apigateway-integration": {
          "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:image/invocations",
          "type": "AWS",
          "credentials": "arn:aws:iam::123456789012:role/Lambda",
          "httpMethod": "POST",
          "contentHandling" : "CONVERT_TO_TEXT",
          "requestTemplates": {
            "application/json": "{\n   \"imageKey\": \"$input.params('key')\",
 \"image\": \"$input.body\"\n}"
          },
          "responses": {
            "default": {
              "statusCode": "500"
            },
            "2\\d{2}": {
              "statusCode": "200"
            }
          }
        }
      }
    }
  },
  "x-amazon-apigateway-binary-media-types" : ["application/octet-stream", "image/
jpeg"],
  "definitions": {
    "Empty": {
```

```
        "type": "object",
        "title": "Empty Schema"
      }
    }
  }
```

从 Lambda 下载图像

从 Lambda 下载二进制 blob 形式的图像文件 (`image.jpg`)：

```
GET /v1/lambda?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/octet-stream
```

成功响应如下所示：

```
200 OK HTTP/1.1

[raw bytes]
```

从 Lambda 下载 Base64 编码字符串（格式化为 JSON 属性）形式的图像文件 (`image.jpg`)：

```
GET /v1/lambda?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/json
```

成功响应如下所示：

```
200 OK HTTP/1.1

{
  "image": "W3JhdyBieXRlc10="
}
```

将图像上传到 Lambda

将二进制 blob 形式的图像文件 (`image.jpg`) 上传到 Lambda：

```
PUT /v1/lambda?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/octet-stream
Accept: application/json

[raw bytes]
```

成功响应如下所示：

```
200 OK
```

将 Base64 编码字符串形式的图像文件 (`image.jpg`) 上传到 Lambda：

```
PUT /v1/lambda?key=image.jpg HTTP/1.1
Host: abcdefghi.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
Accept: application/json

W3JhdyBieXRlc10=
```

成功响应如下所示：

```
200 OK
```

在 Amazon API Gateway 中调用 REST API

为调用已部署的 API，客户端需要向 API Gateway 组件服务的 URL 提交请求以完成 API 执行（称为 `execute-api`）。

REST API 的基本 URL 采用以下格式：

```
https://restapi_id.execute-api.region.amazonaws.com/stage_name/
```

其中，*`restapi_id`* 是 API 标识符，*`region`* 是 AWS 区域，*`stage_name`* 是 API 部署的阶段名称。

Important

在可以调用 API 之前，必须将其部署在 API Gateway 中。有关部署 API 的说明，请参阅在 Amazon API Gateway 中部署 REST API。

主题

获取 API 的调用 URL

您可以使用控制台、AWS CLI 或导出的 OpenAPI 定义来获取 API 的调用 URL。

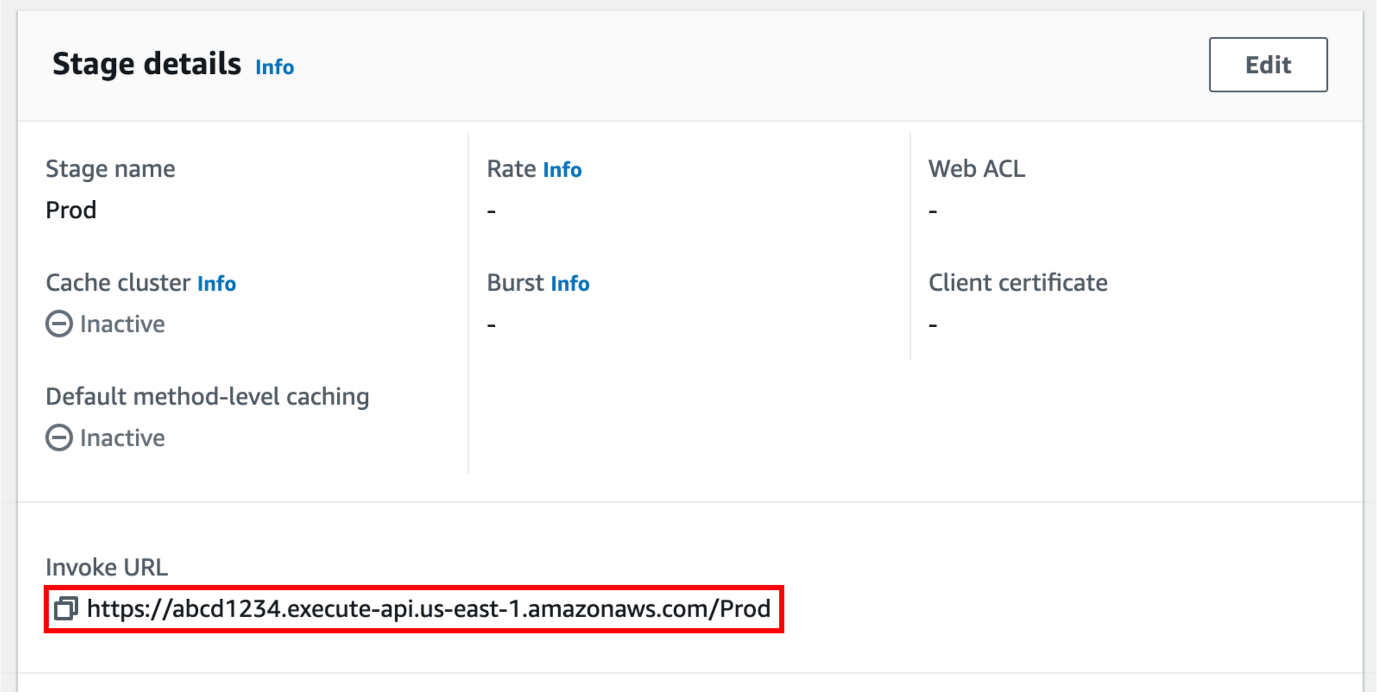
使用控制台获取 API 的调用 URL

以下过程介绍了如何在 REST API 控制台中获取 API 的调用 URL。

使用 REST API 控制台获取 API 的调用 URL

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择已部署的 API。
3. 从主导航窗格中选择阶段。
4. 在阶段详细信息下，选择复制图标以复制您 API 的调用 URL。

   此 URL 用于您的 API 的根资源。



Stage details Info

Stage name
Prod

Cache cluster Info
⊖ Inactive

Default method-level caching
⊖ Inactive

Rate Info
-

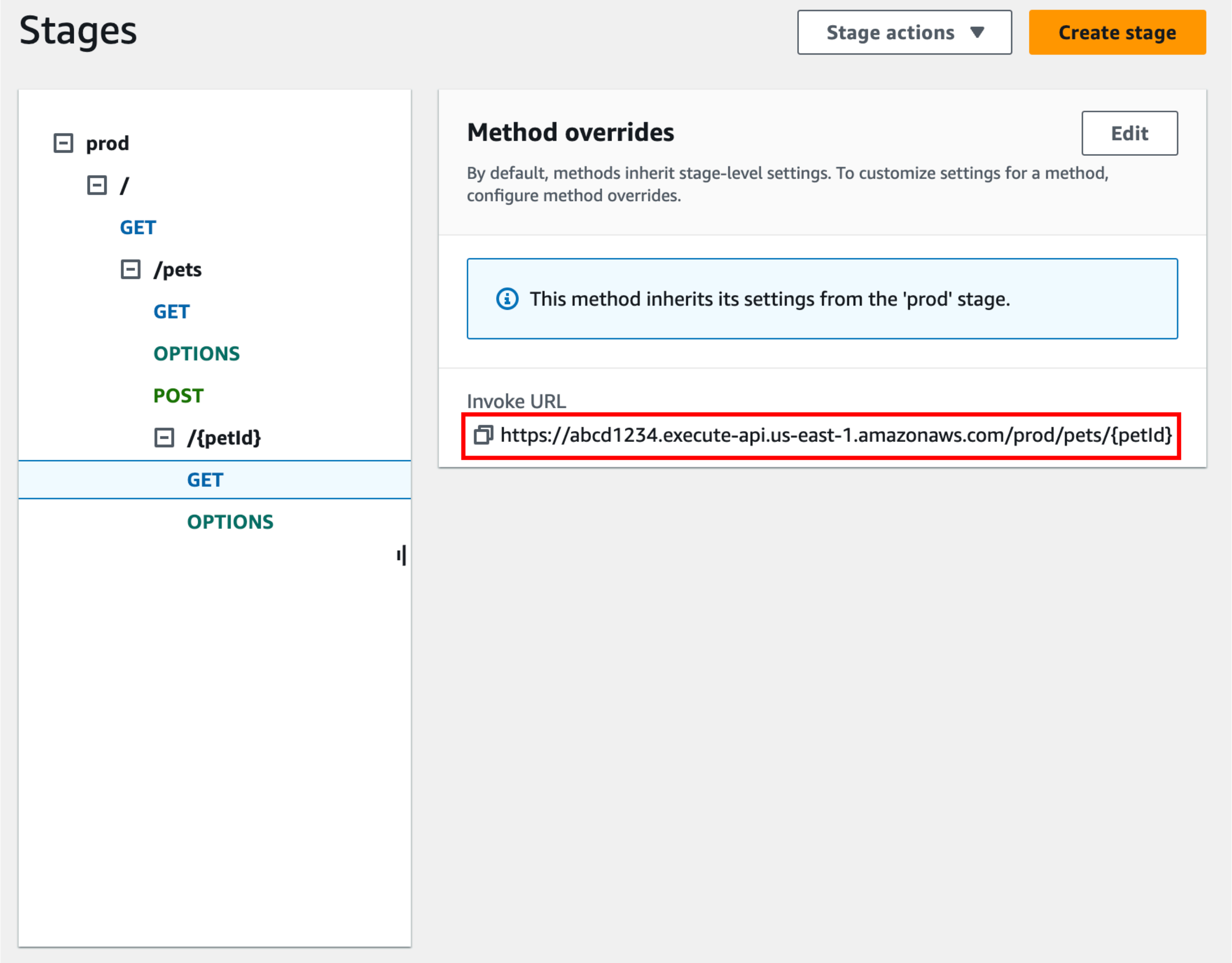
Burst Info
-

Web ACL
-

Client certificate
-

Invoke URL
https://abcd1234.execute-api.us-east-1.amazonaws.com/Prod

5. 要获取 API 中其它资源的 API 的调用 URL，请在二级导航窗格下展开此阶段，然后选择一个方法。
6. 选择复制图标来复制 API 的资源级调用 URL。



By default, methods inherit stage-level settings. To customize settings for a method, configure method overrides.

This method inherits its settings from the 'prod' stage.

Invoke URL

https://abcd1234.execute-api.us-east-1.amazonaws.com/prod/pets/{petId}

使用 AWS CLI 获取 API 的调用 URL

以下过程介绍了如何使用 AWS CLI 获取 API 的调用 URL。

使用 AWS CLI 获取 API 的调用 URL

1. 使用以下命令来获取 `rest-api-id`。此命令返回您的区域中的所有 `rest-api-id` 值。有关更多信息，请参阅 get-rest-apis。

   ```
   aws apigateway get-rest-apis
   ```

2. 将示例 `rest-api-id` 替换为您的 `rest-api-id`，将示例 *`{stage-name}`* 替换为您的 *`{stage-name}`*，将 *`{region}`* 替换为您的区域。

```
https://{restapi_id}.execute-api.{region}.amazonaws.com/{stage_name}/
```

使用 API 的已导出 OpenAPI 定义文件获取 API 的调用 URL

此外，您也可以构造根 URL，方法是组合 API 的已导出 OpenAPI 定义文件的 `host` 和 `basePath` 字段。有关如何导出 API 的说明，请参阅the section called “导出 REST API”。

调用 API

您可以使用浏览器、curl 或其它应用程序（例如 Postman）调用已部署的 API。

此外，您可以使用 API Gateway 控制台测试 API 调用。测试使用 API Gateway 的 `TestInvoke` 特征，该特征允许在部署 API 之前对 API 进行测试。有关更多信息，请参阅 the section called “使用控制台测试 REST API 方法”。

> Note
>
> 调用 URL 中的查询字符串参数值不得包含 `%%`。

使用 Web 浏览器调用 API

如果 API 允许匿名访问，您可以使用任何 Web 浏览器来调用任何 `GET` 方法。在浏览器的地址栏中输入完整的调用 URL。

对于其它方法或任何要求身份验证的调用，您必须指定负载或签署请求。您可以在 HTML 页面背后的脚本中，或者在使用 AWS 开发工具包之一的客户端应用程序中处理这些调用。

使用 curl 调用 API

您可以在终端中使用像 curl 这样的工具来调用 API。以下示例 curl 命令在 API 的 `prod` 阶段的 `getUsers` 资源上调用 GET 方法。

Linux or Macintosh

```
curl -X GET 'https://b123abcde4.execute-api.us-west-2.amazonaws.com/prod/getUsers'
```

Windows

```
curl -X GET "https://b123abcde4.execute-api.us-west-2.amazonaws.com/prod/getUsers"
```

使用 API Gateway 控制台测试 REST API 方法

使用 API Gateway 控制台测试 REST API 方法。

主题

- 先决条件
- 使用 API Gateway 控制台测试方法

先决条件

- 您必须指定要测试的方法的设置。按照API Gateway 中用于 REST API 的方法中的说明进行操作。

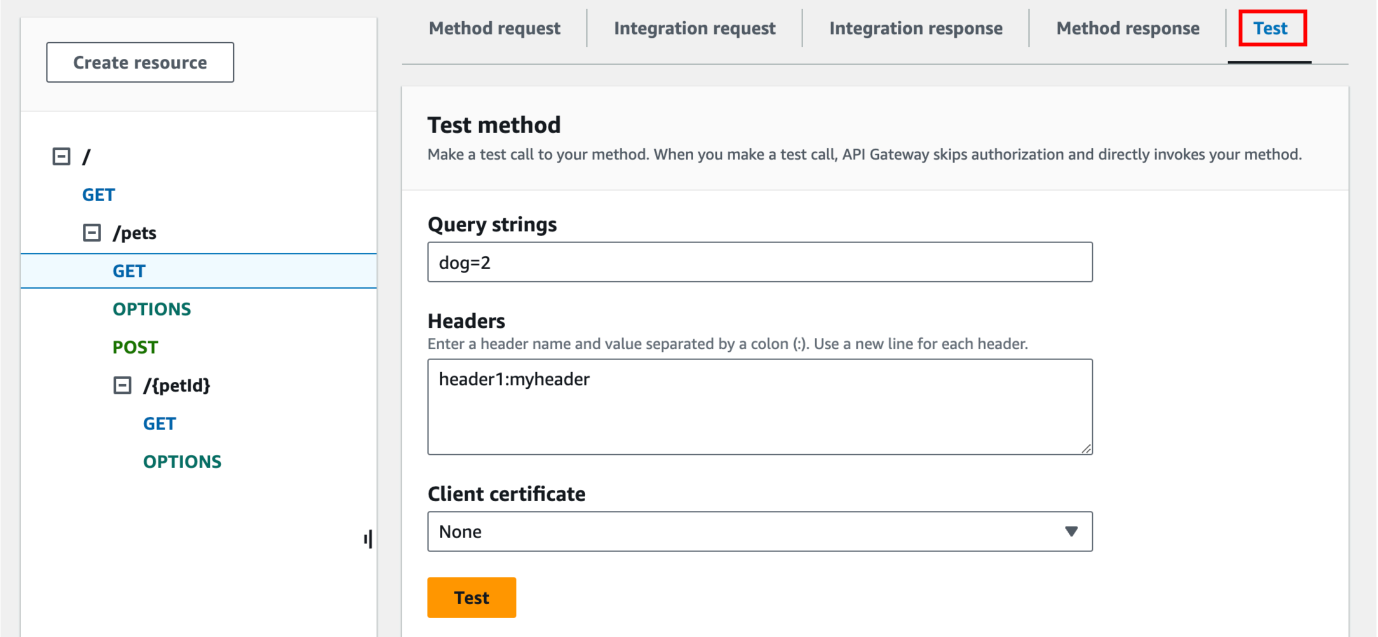
使用 API Gateway 控制台测试方法

Important

使用 API Gateway 控制台测试方法可能会导致对资源进行无法撤销的更改。使用 API Gateway 控制台测试方法与在 API Gateway 控制台之外调用方法相同。例如，如果您使用 API Gateway 控制台调用用于删除 API 资源的方法，并且方法调用成功，那么将删除 API 资源。

测试方法

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 在资源窗格中，选择要测试的方法。
4. 选择测试选项卡。您可能需要选择右箭头按钮，以显示该选项卡。



Test method

Make a test call to your method. When you make a test call, API Gateway skips authorization and directly invokes your method.

Query strings

dog=2

Headers

Enter a header name and value separated by a colon (:). Use a new line for each header.

header1:myheader

Client certificate

None

在任何显示的框中输入值（例如查询字符串、标头和请求正文。控制台会以默认 application/json 形式将这些值包括在方法请求中。

对于您可能需要指定的其它选项，请联系 API 所有者。

5. 选择测试。此时将显示以下信息：

- 请求是为方法调用的资源路径。
- 状态是响应的 HTTP 状态代码。
- 延迟（毫秒）是收到调用方请求和返回响应之间的时间。
- 响应正文是 HTTP 响应正文。
- 响应标头是 HTTP 响应标头。

> **Tip**
>
> 根据映射的不同，HTTP 状态代码、响应正文和响应标头可能不同于从 Lambda 函数、HTTP 代理或 AWS 服务代理发送的内容。

- 日志 是模拟的 Amazon CloudWatch Logs 条目，如果在 API Gateway 控制台之外调用此方法，则会写入这些条目。

> **Note**
>
> 尽管 CloudWatch Logs 条目是模拟的，但方法调用的结果是真实的。

除了使用 API Gateway 控制台之外，您还可以使用 AWS CLI 或适用于 API Gateway 的 AWS 开发工具包来测试调用方法。要使用 AWS CLI 执行此操作，请参阅 test-invoke-method。

使用由 API Gateway 为 REST API 生成的 Java 开发工具包

在此部分中，我们将使用简单结算器 API 作为示例，概述使用由 API Gateway 为 REST API 生成的 Java 开发工具包的步骤。您必须先完成在 API Gateway 中为 REST API 生成开发工具包中的步骤，然后才能继续操作。

安装和使用 API Gateway 生成的 Java 开发工具包

1. 提取您之前下载的 API Gateway 生成的 .zip 文件中的内容。
2. 下载并安装 Apache Maven（必须为版本 3.5 或更高版本）。
3. 下载并安装 JDK 8。
4. 设置 `JAVA_HOME` 环境变量。
5. 转到 pom.xml 文件所在的解压缩的开发工具包文件夹。默认情况下，此文件夹为 `generated-code`。运行 mvn install 命令，将编译的构件文件安装到您的本地 Maven 存储库中。这将创建包含编译的开发工具包库的 `target` 文件夹。
6. 在空目录中输入以下命令来创建客户端项目存根，以使用已安装的开发工具包库来调用 API。

```
mvn -B archetype:generate \
    -DarchetypeGroupdId=org.apache.maven.archetypes \
    -DgroupId=examples.aws.apig.simpleCalc.sdk.app \
    -DartifactId=SimpleCalc-sdkClient
```

> **Note**
>
> 为提高可读性，在上述命令中添加了分隔符 \。整个命令应在一行中且不带分隔符。

此命令会创建一个应用程序存根。应用程序存根在项目的根目录（上述命令中的 *`SimpleCalc-sdkClient`*）下包含一个 `pom.xml` 文件和一个 `src` 文件夹。最初，有两个源文件：`src/`

`main/java/{package-path}/App.java` 和 `src/test/java/{package-path}/AppTest.java`。在本示例中，`{package-path}` 为 `examples/aws/apig/simpleCalc/sdk/app`。此程序包路径源自 `DarchetypeGroupdId` 值。您可以使用 `App.java` 文件作为您的客户端应用程序模板，并且可以根据需要在同一文件夹中添加其他文件。您可以使用 `AppTest.java` 文件作为应用程序的单元测试模板，并且可以根据需要将其他测试代码文件添加到同一个测试文件夹中。

7. 在生成的 `pom.xml` 文件中将程序包依赖关系更新为以下内容，根据需要替换项目的 `groupId`、`artifactId`、`version` 和 `name` 属性：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/
POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>examples.aws.apig.simpleCalc.sdk.app</groupId>
  <artifactId>SimpleCalc-sdkClient</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>SimpleCalc-sdkClient</name>
  <url>http://maven.apache.org</url>

   <dependencies>
      <dependency>
          <groupId>com.amazonaws</groupId>
          <artifactId>aws-java-sdk-core</artifactId>
          <version>1.11.94</version>
      </dependency>
      <dependency>
          <groupId>my-apig-api-examples</groupId>
          <artifactId>simple-calc-sdk</artifactId>
          <version>1.0.0</version>
      </dependency>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>

    <dependency>
        <groupId>commons-io</groupId>
        <artifactId>commons-io</artifactId>
```

```
        <version>2.5</version>
      </dependency>
    </dependencies>

    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.5.1</version>
          <configuration>
            <source>1.8</source>
            <target>1.8</target>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </project>
```

Note

当 `aws-java-sdk-core` 从属构件的较新版本与以上指定的版本 (`1.11.94`) 不兼容时，您必须将 `<version>` 标记更新到新版本。

8. 接下来，我们将介绍如何通过调用开发工具包的 `getABOp(GetABOpRequest req)`、`getApiRoot(GetApiRootRequest req)` 和 `postApiRoot(PostApiRootRequest req)` 方法，来使用开发工具包调用 API。这些方法分别与 `GET /{a}/{b}/{op}`、`GET /?a={x}&b={y}&op={operator}` 和 `POST /` 方法对应，具有 `{"a": x, "b": y, "op": "operator"}` API 请求的负载。

   按如下所示更新 `App.java` 文件：

```
package examples.aws.apig.simpleCalc.sdk.app;

import java.io.IOException;

import com.amazonaws.opensdk.config.ConnectionConfiguration;
import com.amazonaws.opensdk.config.TimeoutConfiguration;

import examples.aws.apig.simpleCalc.sdk.*;
import examples.aws.apig.simpleCalc.sdk.model.*;
```

```
import examples.aws.apig.simpleCalc.sdk.SimpleCalcSdk.*;

public class App
{
    SimpleCalcSdk sdkClient;

    public App() {
        initSdk();
    }

    // The configuration settings are for illustration purposes and may not be a
 recommended best practice.
    private void initSdk() {
        sdkClient = SimpleCalcSdk.builder()
              .connectionConfiguration(
                  new ConnectionConfiguration()
                        .maxConnections(100)
                        .connectionMaxIdleMillis(1000))
              .timeoutConfiguration(
                  new TimeoutConfiguration()
                        .httpRequestTimeout(3000)
                        .totalExecutionTimeout(10000)
                        .socketTimeout(2000))
        .build();

    }
    // Calling shutdown is not necessary unless you want to exert explicit control
 of this resource.
    public void shutdown() {
        sdkClient.shutdown();
    }

    // GetABOpResult getABOp(GetABOpRequest getABOpRequest)
    public Output getResultWithPathParameters(String x, String y, String operator)
 {
     operator = operator.equals("+") ? "add" : operator;
     operator = operator.equals("/") ? "div" : operator;

        GetABOpResult abopResult = sdkClient.getABOp(new
 GetABOpRequest().a(x).b(y).op(operator));
        return abopResult.getResult().getOutput();
    }

    public Output getResultWithQueryParameters(String a, String b, String op) {
```

```
        GetApiRootResult rootResult = sdkClient.getApiRoot(new
 GetApiRootRequest().a(a).b(b).op(op));
        return rootResult.getResult().getOutput();
    }

    public Output getResultByPostInputBody(Double x, Double y, String o) {
     PostApiRootResult postResult = sdkClient.postApiRoot(
      new PostApiRootRequest().input(new Input().a(x).b(y).op(o)));
     return postResult.getResult().getOutput();
    }

    public static void main( String[] args )
    {
        System.out.println( "Simple calc" );
        // to begin
        App calc = new App();

        // call the SimpleCalc API
        Output res = calc.getResultWithPathParameters("1", "2", "-");
        System.out.printf("GET /1/2/-: %s\n", res.getC());

        // Use the type query parameter
        res = calc.getResultWithQueryParameters("1", "2", "+");
        System.out.printf("GET /?a=1&b=2&op=+: %s\n", res.getC());

        // Call POST with an Input body.
        res = calc.getResultByPostInputBody(1.0, 2.0, "*");
        System.out.printf("PUT /\n\n{\"a\":1, \"b\":2,\"op\":\"*\"}\n %s\n",
 res.getC());


    }
}
```

在上述示例中，用于实例化开发工具包客户端的配置设置用于说明用途，并不一定是建议的最佳实践。此外，调用 `sdkClient.shutdown()` 是可选的，尤其是在您需要精确控制何时释放资源的情况下。

我们已介绍使用 Java 开发工具包调用 API 的基本模式。您可以扩展说明以调用其他 API 方法。

使用由 API Gateway 为 REST API 生成的 Android 开发工具包

在本部分中，我们介绍由 API Gateway 为 REST API 生成的 Android 开发工具包的使用步骤。您必须先完成在 API Gateway 中为 REST API 生成开发工具包中的步骤，然后才能继续进行操作。

> **Note**
>
> 生成的开发工具包与 Android 4.4 及更早版本不兼容。有关更多信息，请参阅 the section called “重要提示”。

安装和使用由 API Gateway 生成的 Android 开发工具包

1. 提取您之前下载的 API Gateway 生成的 .zip 文件中的内容。
2. 下载并安装 Apache Maven（最好是版本 3.x）。
3. 下载并安装 JDK 8。
4. 设置 `JAVA_HOME` 环境变量。
5. 运行 mvn install 命令，将编译的构件文件安装到您的本地 Maven 存储库中。这将创建包含编译的开发工具包库的 `target` 文件夹。
6. 将 `target` 文件夹中的开发工具包文件（其名称源自您在生成开发工具包时指定的 Artifact Id 和 Artifact Version，如 `simple-calcsdk-1.0.0.jar`），以及 `target/lib` 文件夹中的所有文件夹库复制到项目的 `lib` 文件夹中。

   如果您使用的是 Android Studio，请在客户端应用程序模块下创建一个 `libs` 文件夹，并将所需的 .jar 文件复制到此文件夹中。验证该模块的 Gradle 文件中的依赖项部分是否包含以下内容。

   ```
       compile fileTree(include: ['*.jar'], dir: 'libs')
       compile fileTree(include: ['*.jar'], dir: 'app/libs')
   ```

   确保没有声明重复的 .jar 文件。
7. 使用 `ApiClientFactory` 类初始化 API Gateway 生成的开发工具包。例如：

   ```
   ApiClientFactory factory = new ApiClientFactory();

   // Create an instance of your SDK. Here, 'SimpleCalcClient.java' is the compiled
    java class for the SDK generated by API Gateway.
   final SimpleCalcClient client = factory.build(SimpleCalcClient.class);
   ```

```
// Invoke a method:
//   For the 'GET /?a=1&b=2&op=+' method exposed by the API, you can invoke it by
 calling the following SDK method:

Result output = client.rootGet("1", "2", "+");

//     where the Result class of the SDK corresponds to the Result model of the
 API.
//

//   For the 'GET /{a}/{b}/{op}'  method exposed by the API, you can call the
 following SDK method to invoke the request,

Result output = client.aBOpGet(a, b, c);

//     where a, b, c can be "1", "2", "add", respectively.

//   For the following API method:
//        POST /
//        host: ...
//        Content-Type: application/json
//
//        { "a": 1, "b": 2, "op": "+" }
// you can call invoke it by calling the rootPost method of the SDK as follows:
Input body = new Input();
input.a=1;
input.b=2;
input.op="+";
Result output = client.rootPost(body);

//      where the Input class of the SDK corresponds to the Input model of the API.

// Parse the result:
//     If the 'Result' object is { "a": 1, "b": 2, "op": "add", "c":3"}, you
 retrieve the result 'c') as

String result=output.c;
```

8. 要使用 Amazon Cognito 凭证提供程序授权调用您的 API，请通过 API Gateway 生成的开发工具包使用 `ApiClientFactory` 类传递一组 AWS 凭证，如下例所示。

```
// Use CognitoCachingCredentialsProvider to provide AWS credentials
// for the ApiClientFactory
AWSCredentialsProvider credentialsProvider = new CognitoCachingCredentialsProvider(
        context,          // activity context
        "identityPoolId", // Cognito identity pool id
        Regions.US_EAST_1 // region of Cognito identity pool
);

ApiClientFactory factory = new ApiClientFactory()
  .credentialsProvider(credentialsProvider);
```

9. 要使用 API Gateway 生成的开发工具包设置 API 键，请使用与以下示例类似的代码。

```
ApiClientFactory factory = new ApiClientFactory()
  .apiKey("YOUR_API_KEY");
```

使用由 API Gateway 为 REST API 生成的 JavaScript 开发工具包

Note

这些说明假设您已完成在 API Gateway 中为 REST API 生成开发工具包中的说明。

Important

如果您的 API 仅定义了 ANY 方法，则生成的开发工具包将不会包含 `apigClient.js` 文件，您将需要自己定义 ANY 方法。

要安装，请启动并调用由 API Gateway 为 REST API 生成的 JavaScript 开发工具包

1. 提取您之前下载的 API Gateway 生成的 .zip 文件中的内容。
2. 针对 API Gateway 生成的开发工具包将调用的所有方法启用跨源资源共享 (CORS)。有关说明，请参阅 为 REST API 资源启用 CORS。

3. 在您的网页中，添加对以下脚本的引用。

```
<script type="text/javascript" src="lib/axios/dist/axios.standalone.js"></script>
<script type="text/javascript" src="lib/CryptoJS/rollups/hmac-sha256.js"></script>
<script type="text/javascript" src="lib/CryptoJS/rollups/sha256.js"></script>
<script type="text/javascript" src="lib/CryptoJS/components/hmac.js"></script>
<script type="text/javascript" src="lib/CryptoJS/components/enc-base64.js"></
script>
<script type="text/javascript" src="lib/url-template/url-template.js"></script>
<script type="text/javascript" src="lib/apiGatewayCore/sigV4Client.js"></script>
<script type="text/javascript" src="lib/apiGatewayCore/apiGatewayClient.js"></
script>
<script type="text/javascript" src="lib/apiGatewayCore/simpleHttpClient.js"></
script>
<script type="text/javascript" src="lib/apiGatewayCore/utils.js"></script>
<script type="text/javascript" src="apigClient.js"></script>
```

4. 在代码中，使用类似于以下内容的代码初始化 API Gateway 生成的开发工具包。

```
var apigClient = apigClientFactory.newClient();
```

要使用 AWS 凭证初始化 API Gateway 生成的开发工具包，请使用类似于以下内容的代码。如果您使用 AWS 凭证，系统将签署针对 API 的所有请求。

```
var apigClient = apigClientFactory.newClient({
  accessKey: 'ACCESS_KEY',
  secretKey: 'SECRET_KEY',
});
```

要将 API 密钥用于 API Gateway 生成的开发工具包，您可以使用类似于以下内容的代码，将 API 密钥作为参数传递给 `Factory` 对象。如果您使用 API 密钥，它将被指定为 `x-api-key` 标头的一部分，并且系统将签署针对 API 的所有请求。这意味着您必须为每个请求设置相应的 CORS Accept 标头。

```
var apigClient = apigClientFactory.newClient({
  apiKey: 'API_KEY'
});
```

5. 使用类似于以下内容的代码调用 API Gateway 中的 API 方法。每次调用都会返回成功和失败回调的承诺。

```
var params = {
  // This is where any modeled request parameters should be added.
  // The key is the parameter name, as it is defined in the API in API Gateway.
  param0: '',
  param1: ''
};

var body = {
  // This is where you define the body of the request,
};

var additionalParams = {
  // If there are any unmodeled query parameters or headers that must be
  //   sent with the request, add them here.
  headers: {
    param0: '',
    param1: ''
  },
  queryParams: {
    param0: '',
    param1: ''
  }
};

apigClient.methodName(params, body, additionalParams)
    .then(function(result){
      // Add success callback code here.
    }).catch( function(result){
      // Add error callback code here.
    });
```

此处，*`methodName`* 由方法请求的资源路径和 HTTP 动词构成。对于 SimpleCalc API，API 方法的开发工具包方法为

```
1. GET /?a=...&b=...&op=...
2. POST /

   { "a": ..., "b": ..., "op": ...}
3. GET /{a}/{b}/{op}
```

相应的开发工具包方法如下所示：

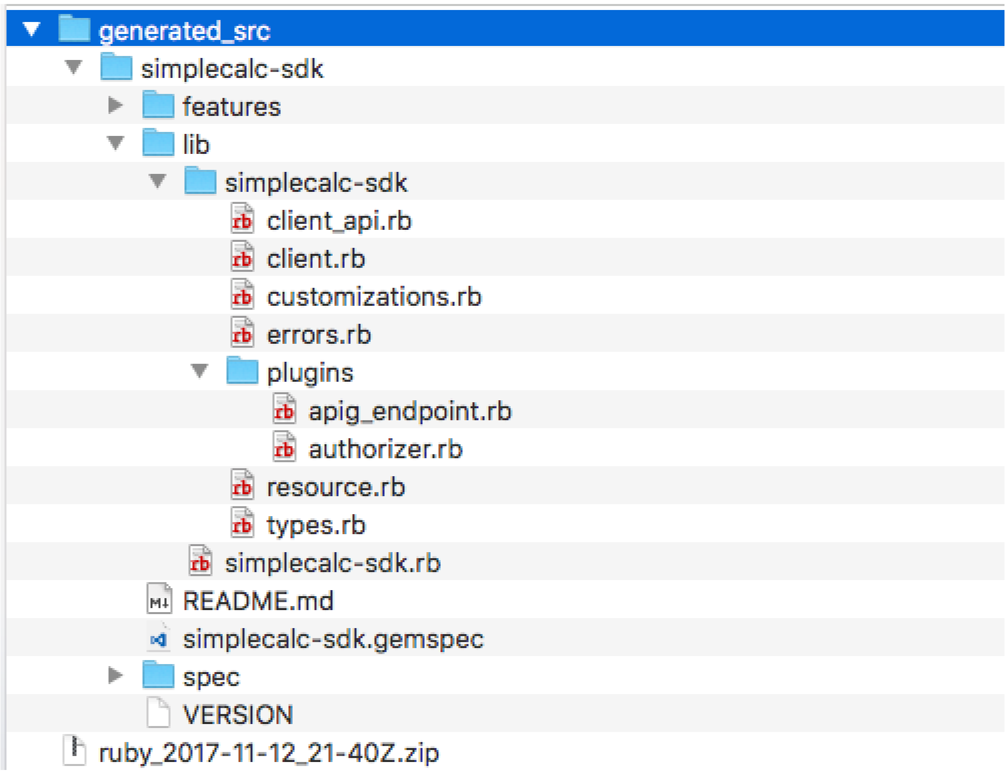
```
1. rootGet(params);       // where params={"a": ..., "b": ..., "op": ...} is
 resolved to the query parameters
2. rootPost(null, body); // where body={"a": ..., "b": ..., "op": ...}
3. aBOpGet(params);       // where params={"a": ..., "b": ..., "op": ...} is
 resolved to the path parameters
```

使用由 API Gateway 为 REST API 生成的 Ruby 开发工具包

> **Note**
>
> 这些说明假设您已完成在 API Gateway 中为 REST API 生成开发工具包中的说明。

要开始安装，请实例化并调用由 API Gateway 为 REST API 生成的 Ruby 开发工具包

1. 解压缩下载的 Ruby 开发工具包文件。生成的开发工具包源如下所示。



```
generated_src
    simplecalc-sdk
        features
        lib
            simplecalc-sdk
                client_api.rb
                client.rb
                customizations.rb
                errors.rb
                plugins
                    apig_endpoint.rb
                    authorizer.rb
                resource.rb
                types.rb
            simplecalc-sdk.rb
        README.md
        simplecalc-sdk.gemspec
        spec
        VERSION
    ruby_2017-11-12_21-40Z.zip
```

2. 从生成的开发工具包源构建 Ruby Gem，在终端窗口中使用以下 shell 命令：

```
# change to /simplecalc-sdk directory
cd simplecalc-sdk

# build the generated gem
gem build simplecalc-sdk.gemspec
```

在此之后，simplecalc-sdk-1.0.0.gem 变为可用状态。

3. 安装 gem：

```
gem install simplecalc-sdk-1.0.0.gem
```

4. 创建客户端应用程序。在应用程序中实例化和初始化 Ruby 开发工具包客户端：

```
require 'simplecalc-sdk'
client = SimpleCalc::Client.new(
    http_wire_trace: true,
    retry_limit: 5,
    http_read_timeout: 50
)
```

如果 API 已经配置了 `AWS_IAM` 类型的授权，您可以在初始化期间提供 `accessKey` 和 `secretKey`，包括调用方的 AWS 凭证。

```
require 'pet-sdk'
client = Pet::Client.new(
    http_wire_trace: true,
    retry_limit: 5,
    http_read_timeout: 50,
    access_key_id: 'ACCESS_KEY',
    secret_access_key: 'SECRET_KEY'
)
```

5. 在应用程序中通过开发工具包进行 API 调用。

> **Tip**
>
> 如果您不熟悉开发工具包方法调用约定，可以查看生成的开发工具包 `client.rb` 文件夹中的 `lib` 文件。该文件夹包含支持的各个 API 方法调用的文档。

搜索支持的操作：

```
# to show supported operations:
puts client.operation_names
```

这将生成以下显示内容，分别对应于 `GET /?a={.}&b={.}&op={.}`、`GET /{a}/{b}/{op}` 的 API 方法，以及 `POST /`，加上 `{a:"…", b:"…", op:"…"}` 格式的负载：

```
[:get_api_root, :get_ab_op, :post_api_root]
```

要调用 `GET /?a=1&b=2&op=+` API 方法，请调用以下 Ruby 开发工具包方法：

```
resp = client.get_api_root({a:"1", b:"2", op:"+"})
```

要使用 `POST /` 的负载调用 `{a: "1", b: "2", "op": "+"}` API 方法，请调用以下 Ruby 开发工具包方法：

```
resp = client.post_api_root(input: {a:"1", b:"2", op:"+"})
```

要调用 `GET /1/2/+` API 方法，请调用以下 Ruby 开发工具包方法：

```
resp = client.get_ab_op({a:"1", b:"2", op:"+"})
```

成功的开发工具包方法调用返回以下响应：

```
resp : {
    result: {
        input: {
            a: 1,
            b: 2,
            op: "+"
        },
        output: {
            c: 3
        }
    }
}
```

在 Objective-C 或 Swift 中使用由 API Gateway 为 REST API 生成的 iOS 开发工具包

在本教程中，我们将介绍如何在 Objective-C 或 Swift 应用程序中使用由 API Gateway 为 REST API 生成的 iOS 开发工具包来调用底层 API。我们将使用 SimpleCalc API 作为示例来说明以下主题：

- 如何将所需的 AWS 移动开发工具包组件安装到您的 Xcode 项目中
- 如何在调用 API 的方法前创建 API 客户端对象
- 如何通过 API 客户端对象上相应的 SDK 方法调用 API 方法
- 如何使用 SDK 的相应模型类来准备方法输入并分析其结果

主题

使用生成的 iOS 开发工具包 (Objective-C) 来调用 API

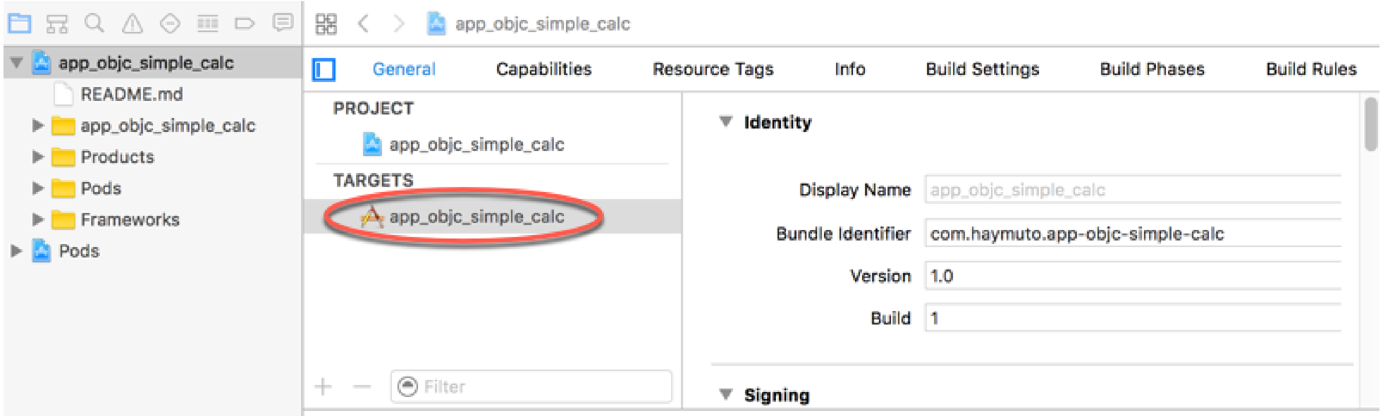
在开始以下过程之前，您必须完成在 API Gateway 中为 REST API 生成开发工具包中适用于 Objective-C 版 iOS 的步骤，并下载已生成的软件开发工具包的 .zip 文件。

在 Objective-C 项目中安装 AWS 移动开发工具包和 API Gateway 生成的 iOS 开发工具包

以下过程将介绍如何安装开发工具包。

安装并使用 API Gateway 生成的 Objective-C 版 iOS 开发工具包

1. 提取您之前下载的 API Gateway 生成的 .zip 文件中的内容。使用 SimpleCalc API，您可能需要将解压的软件开发工具包文件夹重命名为类似 **`sdk_objc_simple_calc`** 的名字。此开发工具包文件夹中有一个 `README.md` 文件和一个 `Podfile` 文件。`README.md` 文件包含介绍如何安装和使用开发工具包的说明。本教程将提供有关这些说明的详细信息。安装过程会利用 CocoaPods 导入必需的 API Gateway 库和其他依赖的AWS移动开发工具包组件。您必须更新 `Podfile`，才能将开发工具包导入应用程序的 Xcode 项目中。解档开发工具包文件夹还包含一个 `generated-src` 文件夹，其中包含 API 已生成开发工具包的源代码。
2. 启动 Xcode，并创建一个新的 iOS Objective-C 项目。请记下该项目的目标。您需要在 `Podfile` 中对其进行设置。



Display Name app_objc_simple_calc
Bundle Identifier com.haymuto.app-objc-simple-calc
Version 1.0
Build 1

3. 要使用 CocoaPods 将 AWS Mobile SDK for iOS 导入 Xcode 项目中，请执行以下操作：

   a. 通过在终端窗口运行以下命令来安装 CocoaPods：

   ```
   sudo gem install cocoapods
   pod setup
   ```

   b. 将 `Podfile` 文件从提取的开发工具包文件夹复制到包含 Xcode 项目文件的同一目录。将以下代码块：

   ```
   target '<YourXcodeTarget>' do
       pod 'AWSAPIGateway', '~> 2.4.7'
   end
   ```

   替换为您的项目的目标名称：

   ```
   target 'app_objc_simple_calc' do
       pod 'AWSAPIGateway', '~> 2.4.7'
   end
   ```

   如果 Xcode 项目已经包含名为 `Podfile` 的文件，请向其添加以下代码行：

   ```
   pod 'AWSAPIGateway', '~> 2.4.7'
   ```

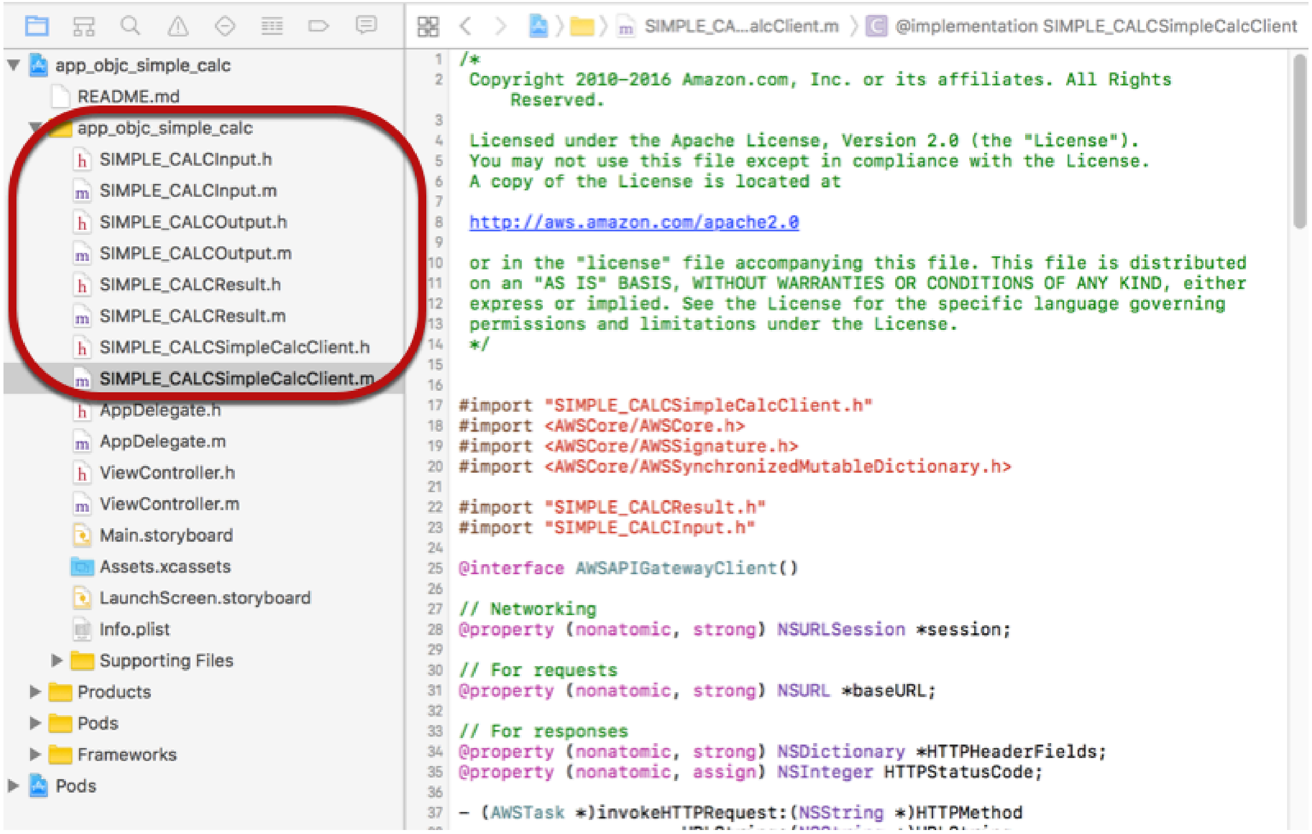
   c. 打开终端窗口，并运行以下命令：

   ```
   pod install
   ```

   这将安装 API Gateway 组件和其他依赖的 AWS 移动开发工具包组件。

d. 关闭 Xcode 项目，然后打开 `.xcworkspace` 文件，以重新启动 Xcode。

e. 将已提取开发工具包的 `.h` 目录中的所有 `.m` 和 `generated-src` 文件添加到 Xcode 项目中。



```
/*
 Copyright 2010-2016 Amazon.com, Inc. or its affiliates. All Rights
     Reserved.

 Licensed under the Apache License, Version 2.0 (the "License").
 You may not use this file except in compliance with the License.
 A copy of the License is located at

 http://aws.amazon.com/apache2.0

 or in the "license" file accompanying this file. This file is distributed
 on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 express or implied. See the License for the specific language governing
 permissions and limitations under the License.
 */


#import "SIMPLE_CALCSimpleCalcClient.h"
#import <AWSCore/AWSCore.h>
#import <AWSCore/AWSSignature.h>
#import <AWSCore/AWSSynchronizedMutableDictionary.h>

#import "SIMPLE_CALCResult.h"
#import "SIMPLE_CALCInput.h"

@interface AWSAPIGatewayClient()

// Networking
@property (nonatomic, strong) NSURLSession *session;

// For requests
@property (nonatomic, strong) NSURL *baseURL;

// For responses
@property (nonatomic, strong) NSDictionary *HTTPHeaderFields;
@property (nonatomic, assign) NSInteger HTTPStatusCode;

- (AWSTask *)invokeHTTPRequest:(NSString *)HTTPMethod
```

要通过显式下载 AWS Mobile SDK for iOS 移动开发工具包或使用 AWSCarthage 的方式将 Objective-C 导入您的项目中，请遵循 README.md 文件中的说明。确保只使用以下选项之一来导入 AWS 移动开发工具包。

在 Objective-C 项目中使用 API Gateway 生成的 iOS 开发工具包调用 API 方法

当您为此 SimpleCalc API 生成具有 `SIMPLE_CALC` 前缀的开发工具包，并将两个模型用于方法的输入 (`Input`) 和输出 (`Result`) 时，在开发工具包中，由此产生的 API 客户端类将变成 `SIMPLE_CALCSimpleCalcClient`，而且相应数据类分别是 `SIMPLE_CALCInput` 和 `SIMPLE_CALCResult`。API 请求和响应映射到开发工具包方法，如下所示：

- 以下 API 请求

```
GET /?a=...&b=...&op=...
```

将变为以下开发工具包方法

```
(AWSTask *)rootGet:(NSString *)op a:(NSString *)a b:(NSString *)b
```

`AWSTask.result` 属性的类型是 `SIMPLE_CALCResult`，前提是 `Result` 模型已添加到该方法响应中。否则，属性的类型将是 `NSDictionary`。

- 以下 API 请求

```
POST /

{
   "a": "Number",
   "b": "Number",
   "op": "String"
}
```

将变为以下开发工具包方法

```
(AWSTask *)rootPost:(SIMPLE_CALCInput *)body
```

- 以下 API 请求

```
GET /{a}/{b}/{op}
```

将变为以下开发工具包方法

```
(AWSTask *)aBOpGet:(NSString *)a b:(NSString *)b op:(NSString *)op
```

以下过程将介绍如何在 Objective-C 应用程序源代码中调用 API 方法；例如，作为 `viewDidLoad` 文件中 `ViewController.m` 委派的一部分进行调用。

通过 API Gateway 生成的 iOS 开发工具包调用 API

1. 导入 API 客户端类标头文件，使 API 客户端类可在该应用程序内调用：

```
#import "SIMPLE_CALCSimpleCalc.h"
```

`#import` 语句还将为两个模型类导入 `SIMPLE_CALCInput.h` 和 `SIMPLE_CALCResult.h`。

2. 实例化 API 客户端类：

```
SIMPLE_CALCSimpleCalcClient *apiInstance = [SIMPLE_CALCSimpleCalcClient
 defaultClient];
```

要将 Amazon Cognito 和该 API 结合使用，请在默认的 `AWSServiceManager` 对象上设置 `defaultServiceConfiguration` 属性（如下所示），然后再调用 `defaultClient` 方法，以创建 API 客户端对象（如上一示例所示）：

```
AWSCognitoCredentialsProvider *creds = [[AWSCognitoCredentialsProvider alloc]
 initWithRegionType:AWSRegionUSEast1 identityPoolId:your_cognito_pool_id];
AWSServiceConfiguration *configuration = [[AWSServiceConfiguration alloc]
 initWithRegion:AWSRegionUSEast1 credentialsProvider:creds];
AWSServiceManager.defaultServiceManager.defaultServiceConfiguration =
 configuration;
```

3. 调用 `GET /?a=1&b=2&op=+` 方法以执行 `1+2`：

```
[[apiInstance rootGet: @"+" a:@"1" b:@"2"] continueWithBlock:^id _Nullable(AWSTask
 * _Nonnull task) {
    _textField1.text = [self handleApiResponse:task];
    return nil;
}];
```

其中，帮助程序函数 `handleApiResponse:task` 会将结果格式设定为要在文本字段 (`_textField1`) 中显示的字符串。

```
- (NSString *)handleApiResponse:(AWSTask *)task {
    if (task.error != nil) {
        return [NSString stringWithFormat: @"Error: %@", task.error.description];
    } else if (task.result != nil && [task.result isKindOfClass:[SIMPLE_CALCResult
 class]]) {
        return [NSString stringWithFormat:@"%@ %@ %@ = %@\n",task.result.input.a,
 task.result.input.op, task.result.input.b, task.result.output.c];
    }
    return nil;
```

```
}
```

最终显示为 `1 + 2 = 3`。

4. 使用负载调用 `POST /` 以执行 1-2：

    ```
    SIMPLE_CALCInput *input = [[SIMPLE_CALCInput alloc] init];
        input.a = [NSNumber numberWithInt:1];
        input.b = [NSNumber numberWithInt:2];
        input.op = @"-";
        [[apiInstance rootPost:input] continueWithBlock:^id _Nullable(AWSTask *
     _Nonnull task) {
            _textField2.text = [self handleApiResponse:task];
            return nil;
        }];
    ```

    最终显示为 `1 - 2 = -1`。

5. 调用 `GET /{a}/{b}/{op}` 以执行 1/2：

    ```
    [[apiInstance aBOpGet:@"1" b:@"2" op:@"div"] continueWithBlock:^id
     _Nullable(AWSTask * _Nonnull task) {
        _textField3.text = [self handleApiResponse:task];
        return nil;
    }];
    ```

    最终显示为 `1 div 2 = 0.5`。在这里，`div` 用于代替 `/`，因为后端的简单 Lambda 函数不会处理 URL 编码的路径变量。

使用生成的 iOS 开发工具包 (Swift) 来调用 API

在开始以下过程之前，您必须完成 在 API Gateway 中为 REST API 生成开发工具包 中适用于 Swift 版 iOS 的步骤，并下载已生成的开发工具包的 .zip 文件。

主题

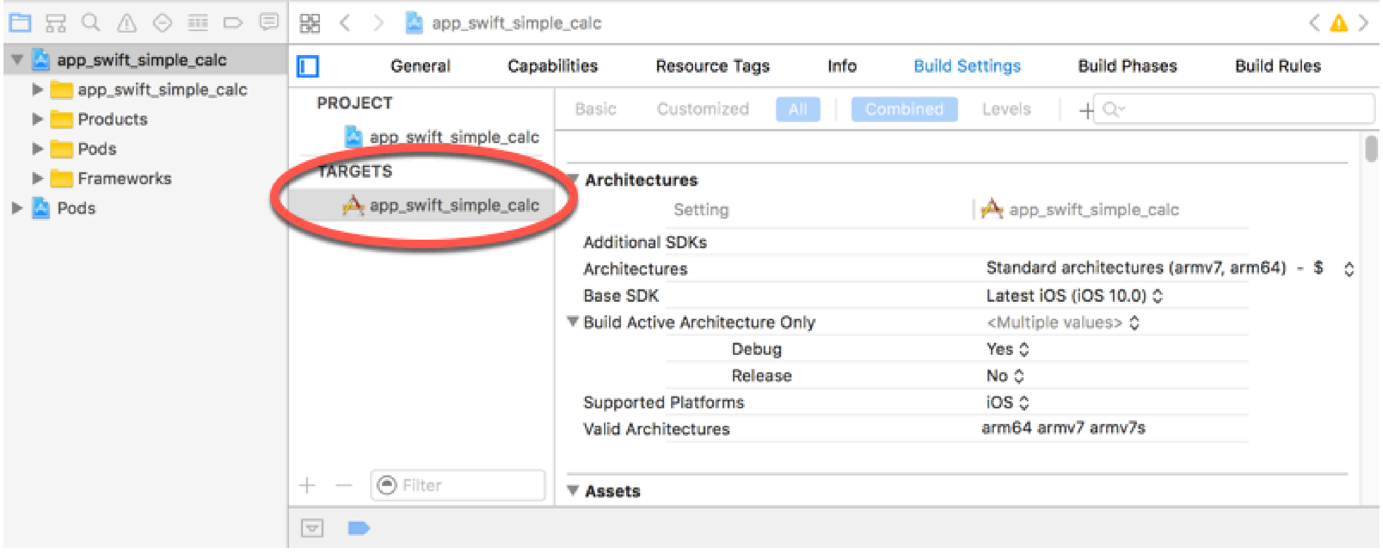
- 在 Swift 项目中安装 AWS 移动开发工具包和 API Gateway 生成的开发工具包
- 在 Swift 项目中通过 API Gateway 生成的 iOS 开发工具包调用 API 方法

在 Swift 项目中安装 AWS 移动开发工具包和 API Gateway 生成的开发工具包

以下过程将介绍如何安装开发工具包。

安装并使用 API Gateway 生成的 Swift 版 iOS 开发工具包

1. 提取您之前下载的 API Gateway 生成的 .zip 文件中的内容。使用 SimpleCalc API，您可能需要将解压的软件开发工具包文件夹重命名为类似 **`sdk_swift_simple_calc`** 的名字。此开发工具包文件夹中有一个 `README.md` 文件和一个 `Podfile` 文件。`README.md` 文件包含介绍如何安装和使用开发工具包的说明。本教程将提供有关这些说明的详细信息。安装过程会利用 CocoaPods 导入必需的 AWS 移动开发工具包组件。您必须更新 `Podfile`，才能将开发工具包导入 Swift 应用程序的 Xcode 项目中。解档开发工具包文件夹还包含一个 `generated-src` 文件夹，其中包含 API 已生成开发工具包的源代码。
2. 启动 Xcode，并创建一个新的 iOS Swift 项目。请记下该项目的目标。您需要在 `Podfile` 中对其进行设置。



Setting	app_swift_simple_calc
Additional SDKs	
Architectures	Standard architectures (armv7, arm64) - $
Base SDK	Latest iOS (iOS 10.0)
Build Active Architecture Only	<Multiple values>
Debug	Yes
Release	No
Supported Platforms	iOS
Valid Architectures	arm64 armv7 armv7s

3. 要使用 CocoaPods 将必需的 AWS 移动开发工具包组件导入 Xcode 项目中，请执行以下操作：

   a. 如果 CocoaPods 还未安装，请在终端窗口中运行以下命令进行安装：

   ```
   sudo gem install cocoapods
   pod setup
   ```

   b. 将 `Podfile` 文件从提取的开发工具包文件夹复制到包含 Xcode 项目文件的同一目录。将以下代码块：

   ```
   target '<YourXcodeTarget>' do
   ```

```
    pod 'AWSAPIGateway', '~> 2.4.7'
end
```

替换为您的项目的目标名称，如下所示：

```
target 'app_swift_simple_calc' do
    pod 'AWSAPIGateway', '~> 2.4.7'
end
```

如果 Xcode 项目已经包含带有正确目标的 `Podfile`，您只需将以下代码行添加到 do `...` end 循环：

```
pod 'AWSAPIGateway', '~> 2.4.7'
```

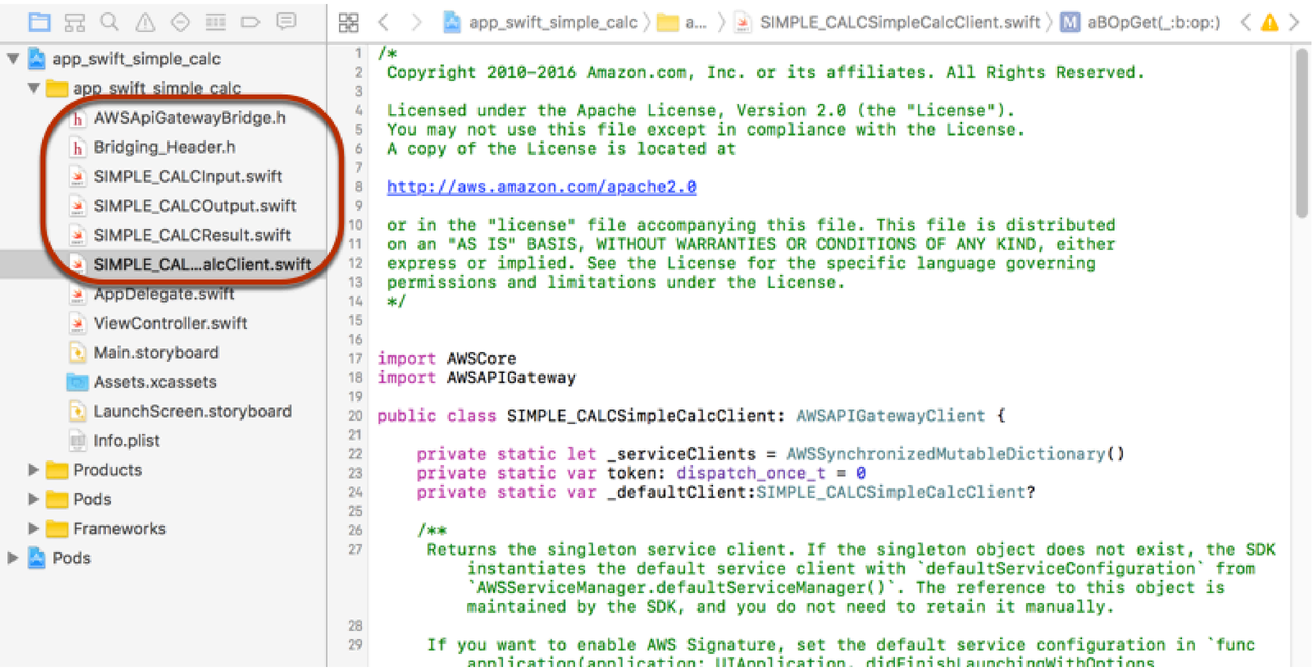
c. 打开终端窗口，并在应用程序目录中运行以下命令：

```
pod install
```

这会将 API Gateway 组件及任何依赖的 AWS 移动开发工具包组件安装到应用程序的项目中。

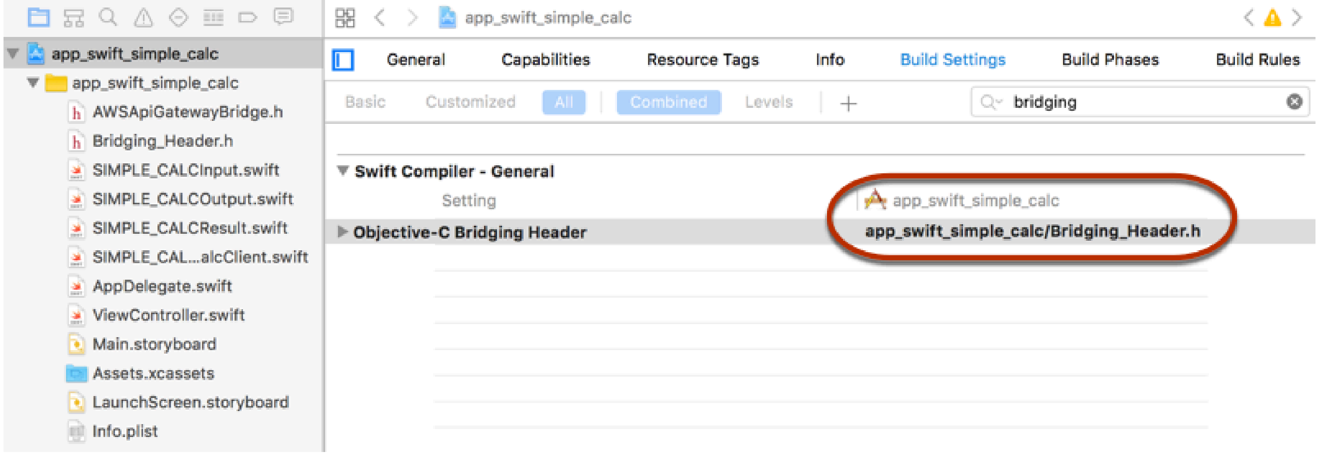
d. 关闭 Xcode 项目，然后打开 `*.xcworkspace` 文件，以重新启动 Xcode。

e. 将已提取 `.h` 目录中所有开发工具包的标头文件 (`.swift`) 和 Swift 源代码文件 (`generated-src`) 添加到您的 Xcode 项目中。



```
/*
 Copyright 2010-2016 Amazon.com, Inc. or its affiliates. All Rights Reserved.

 Licensed under the Apache License, Version 2.0 (the "License").
 You may not use this file except in compliance with the License.
 A copy of the License is located at

 http://aws.amazon.com/apache2.0

 or in the "license" file accompanying this file. This file is distributed
 on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
 express or implied. See the License for the specific language governing
 permissions and limitations under the License.
 */


import AWSCore
import AWSAPIGateway

public class SIMPLE_CALCSimpleCalcClient: AWSAPIGatewayClient {

    private static let _serviceClients = AWSSynchronizedMutableDictionary()
    private static var token: dispatch_once_t = 0
    private static var _defaultClient:SIMPLE_CALCSimpleCalcClient?

    /**
     Returns the singleton service client. If the singleton object does not exist, the SDK
         instantiates the default service client with `defaultServiceConfiguration` from
         `AWSServiceManager.defaultServiceManager()`. The reference to this object is
         maintained by the SDK, and you do not need to retain it manually.

     If you want to enable AWS Signature, set the default service configuration in `func
         application(application: UIApplication, didFinishLaunchingWithOptions
```

f. 为了能够从 Swift 代码项目调用 AWS 移动开发工具包的 Objective-C 库，请在 Xcode 项目配置的 `Bridging_Header.h`Swift Compiler - General 设置下，在 Objective-C Bridging Header 属性上设置 文件路径：



Swift Compiler - General

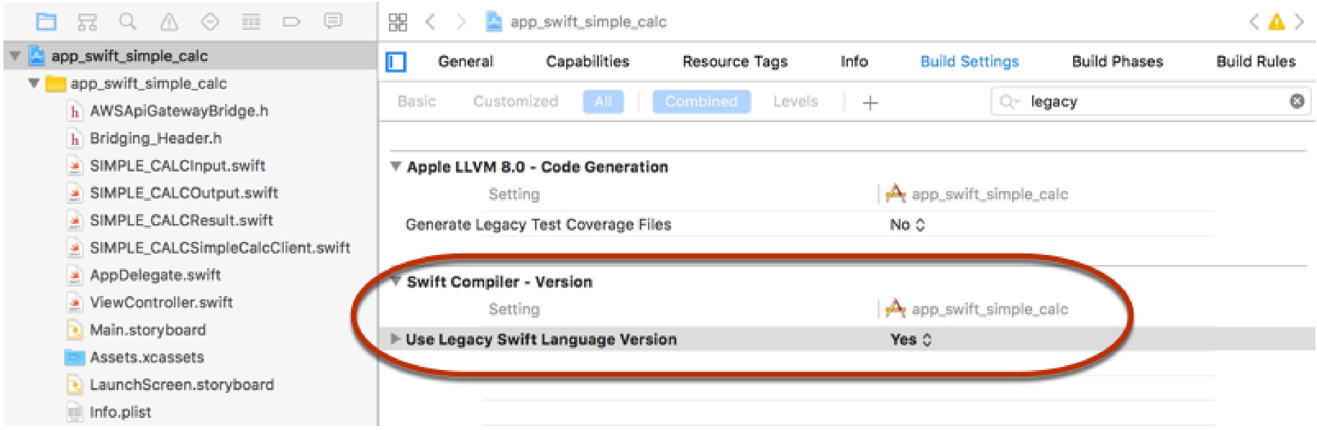
Setting: app_swift_simple_calc

Objective-C Bridging Header: app_swift_simple_calc/Bridging_Header.h

> **Tip**
>
> 您可以通过在 Xcode 的搜索框中键入 **`bridging`** 来找到 Objective-C Bridging Header 属性。

g. 构建 Xcode 项目，以在继续进行操作之前验证它是否正确配置。如果与 AWS 移动开发工具包支持的 Swift 版本相比，您的 Xcode 使用了更新的版本，那么您将收到 Swift 编译器错误。

在这种情况下，在 Swift Compiler - Version 设置下方将 Use Legacy Swift Language Version (使用传统 Swift 语言版本) 属性设置为是：



Apple LLVM 8.0 - Code Generation

Generate Legacy Test Coverage Files No

Swift Compiler - Version

Use Legacy Swift Language Version Yes

要通过显式下载AWS移动软件开发工具包或使用 Carthage，将 Swift 中的AWS移动版 SDK for iOS 导入您的项目中，请按照开发工具包随附的 `README.md` 文件中的说明操作。确保只使用以下选项之一来导入 AWS 移动开发工具包。

在 Swift 项目中通过 API Gateway 生成的 iOS 开发工具包调用 API 方法

当您为此 SimpleCalc API 生成具有 `SIMPLE_CALC` 前缀的开发工具包，并通过两个模型描述 API 请求和响应的输入 (`Input`) 和输出 (`Result`) 时，在开发工具包中，由此产生的 API 客户端类将变成 `SIMPLE_CALCSimpleCalcClient`，而且相应数据类分别是 `SIMPLE_CALCInput` 和 `SIMPLE_CALCResult`。API 请求和响应映射到开发工具包方法，如下所示：

- 以下 API 请求

  ```
  GET /?a=...&b=...&op=...
  ```

  将变为以下开发工具包方法

  ```
  public func rootGet(op: String?, a: String?, b: String?) -> AWSTask
  ```

  `AWSTask.result` 属性的类型是 `SIMPLE_CALCResult`，前提是 `Result` 模型已添加到该方法响应中。否则，它的类型将是 `NSDictionary`。
- 以下 API 请求

  ```
  POST /
  ```

```
{
   "a": "Number",
   "b": "Number",
   "op": "String"
}
```

将变为以下开发工具包方法

```
public func rootPost(body: SIMPLE_CALCInput) -> AWSTask
```

- 以下 API 请求

  ```
  GET /{a}/{b}/{op}
  ```

  将变为以下开发工具包方法

  ```
  public func aBOpGet(a: String, b: String, op: String) -> AWSTask
  ```

以下过程将介绍如何在 Swift 应用程序源代码中调用 API 方法；例如，作为 `viewDidLoad()` 文件中 `ViewController.m` 委派的一部分进行调用。

通过 API Gateway 生成的 iOS 开发工具包调用 API

1. 实例化 API 客户端类：

   ```
   let client = SIMPLE_CALCSimpleCalcClient.default()
   ```

   要将 Amazon Cognito 与 API 结合使用，请设置默认 AWS 服务配置（如下所示），然后再获取 `default` 方法（如之前所示）：

   ```
   let credentialsProvider =
    AWSCognitoCredentialsProvider(regionType: AWSRegionType.USEast1, identityPoolId:
    "my_pool_id")
   let configuration = AWSServiceConfiguration(region: AWSRegionType.USEast1,
    credentialsProvider: credentialsProvider)
   AWSServiceManager.defaultServiceManager().defaultServiceConfiguration =
    configuration
   ```

2. 调用 `GET /?a=1&b=2&op=+` 方法以执行 `1+2`：

   ```
   client.rootGet("+", a: "1", b:"2").continueWithBlock {(task: AWSTask) -> AnyObject?
    in
       self.showResult(task)
       return nil
   }
   ```

   其中，帮助程序函数 `self.showResult(task)` 会将结果或错误打印到控制台；例如：

   ```
   func showResult(task: AWSTask) {
       if let error = task.error {
           print("Error: \(error)")
       } else if let result = task.result {
           if result is SIMPLE_CALCResult {
               let res = result as! SIMPLE_CALCResult
               print(String(format:"%@ %@ %@ = %@", res.input!.a!, res.input!.op!,
    res.input!.b!, res.output!.c!))
           } else if result is NSDictionary {
               let res = result as! NSDictionary
               print("NSDictionary: \(res)")
           }
       }
   }
   ```

   在生产应用程序中，您可以在文本字段中显示结果或错误。最终显示为 `1 + 2 = 3`。

3. 使用负载调用 `POST /` 以执行 `1-2`：

   ```
   let body = SIMPLE_CALCInput()
   body.a=1
   body.b=2
   body.op="-"
   client.rootPost(body).continueWithBlock {(task: AWSTask) -> AnyObject? in
       self.showResult(task)
       return nil
   }
   ```

   结果显示为 `1 - 2 = -1`。

4. 调用 `GET /{a}/{b}/{op}` 以执行 `1/2`：

```
client.aBOpGet("1", b:"2", op:"div").continueWithBlock {(task: AWSTask) ->
 AnyObject? in
    self.showResult(task)
    return nil
}
```

最终显示为 `1 div 2 = 0.5`。在这里，`div` 用于代替 `/`，因为后端的简单 Lambda 函数不会处理 URL 编码的路径变量。

使用 OpenAPI 配置 REST API

您可以使用 API Gateway 将 REST API 从外部定义文件导入到 API Gateway。目前，API Gateway 支持 OpenAPI v2.0 和 OpenAPI v3.0 定义文件，但Amazon API Gateway 关于 REST API 的重要说明中列出了例外。您可以使用新定义覆盖 API 以进行更新，还可以将定义与现有 API 合并。您使用 `mode` 查询参数指定请求 URL 中的选项。

有关从 API Gateway 控制台使用导入 API 功能的教程，请参阅 教程：通过导入示例创建 REST API。

主题

- 将边缘优化的 API 导入 API Gateway
- 将区域 API 导入到 API Gateway 中
- 导入 OpenAPI 文件以更新现有 API 定义
- 设置 OpenAPI basePath 属性
- 用于 OpenAPI 导入的 AWS 变量
- 导入期间的错误和警告
- 从 API Gateway 导出 REST API

将边缘优化的 API 导入 API Gateway

您可以导入 API 的 OpenAPI 定义文件，在 OpenAPI 文件之外通过指定 `EDGE` 终端节点类型作为导入操作的附加输入，创建新的边缘优化的 API。您可以使用 API Gateway 控制台、AWS CLI 或 AWS SDK 执行此操作。

有关从 API Gateway 控制台使用导入 API 特征的教程，请参阅 教程：通过导入示例创建 REST API。

主题

- 使用 API Gateway 控制台导入边缘优化的 API
- 使用 AWS CLI 导入边缘优化的 API

使用 API Gateway 控制台导入边缘优化的 API

要使用 API Gateway 控制台导入边缘优化的 API，请执行以下操作：

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择创建 API。
3. 在REST API中，选择 Import（导入）。
4. 复制 API 的 OpenAPI 定义并将其粘贴到代码编辑器中，或者选择选择文件以从本地驱动器加载 OpenAPI 文件。
5. 对于 API 端点类型，选择边缘优化。
6. 选择创建 API 以开始导入 OpenAPI 定义。

使用 AWS CLI 导入边缘优化的 API

要使用 AWS CLI 从 OpenAPI 定义文件导入 API 创建新的边缘优化的 API，请使用 `import-rest-api` 命令，如下所示：

```
aws apigateway import-rest-api \
    --fail-on-warnings \
    --body 'file://path/to/API_OpenAPI_template.json'
```

或者使用到 `EDGE` 的 `endpointConfigurationTypes` 查询字符串参数的明确规范：

```
aws apigateway import-rest-api \
    --parameters endpointConfigurationTypes=EDGE \
    --fail-on-warnings \
    --body 'file://path/to/API_OpenAPI_template.json'
```

将区域 API 导入到 API Gateway 中

导入 API 时，您可以为 API 选择区域终端节点配置。您可以使用 API Gateway 控制台、AWS CLI 或 AWS SDK。

导出 API 时，API 终端节点配置不包括在导出的 API 定义中。

有关从 API Gateway 控制台使用导入 API 特征的教程，请参阅 教程：通过导入示例创建 REST API。

主题

- 使用 API Gateway 控制台导入区域 API
- 使用 AWS CLI 导入区域 API

使用 API Gateway 控制台导入区域 API

要使用 API Gateway 控制台导入区域终端节点的 API，请执行以下操作：

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择创建 API。
3. 在REST API中，选择 Import（导入）。
4. 复制 API 的 OpenAPI 定义并将其粘贴到代码编辑器中，或者选择选择文件以从本地驱动器加载 OpenAPI 文件。
5. 对于 API 端点类型，选择区域。
6. 选择创建 API 以开始导入 OpenAPI 定义。

使用 AWS CLI 导入区域 API

要使用 AWS CLI 从 OpenAPI 定义文件导入 API，请使用 `import-rest-api` 命令：

```
aws apigateway import-rest-api \
    --parameters endpointConfigurationTypes=REGIONAL \
    --fail-on-warnings \
    --body 'file://path/to/API_OpenAPI_template.json'
```

导入 OpenAPI 文件以更新现有 API 定义

您只能导入 API 定义来更新现有 API，无需更改其终端节点配置以及阶段或阶段变量，或者引用 API 密钥。

导入到更新操作可以使用两种模式进行：合并和覆盖。

当一个 API (A) 合并到另一个 (B) 中时，如果两个 API 中没有互相冲突的定义，生成的 API 会保留 A 和 B 两者的定义。如果出现冲突，合并 API (A) 的方法定义会覆盖接受并入的 API (B) 的相应方法定义。例如，假设 B 声明了以下方法，用于返回 `200` 和 `206` 响应：

```
GET /a
POST /a
```

A 声明了以下方法，用于返回 `200` 和 `400` 响应：

```
GET /a
```

如果 A 并入 B 中，生成的 API 会生成以下方法：

```
GET /a
```

返回 `200` 和 `400` 响应，以及

```
POST /a
```

返回 `200` 和 `206` 响应。

当您将外部 API 定义分解为多个较小的部分，并希望一次只应用其中一个部分的更改时，合并 API 非常有用。例如，如果多个团队负责一个 API 的不同部分并以不同的速度提供更改，则可能会出现此情况。在这种模式下，没有在导入的定义中明确定义的现有 API 中的项目会被忽略。

如果一个 API (A) 覆盖另一个 API (B)，生成的 API 将采纳覆盖方 API (A) 的定义。当外部 API 定义包含一个 API 的完整定义时，覆盖 API 非常有用。在这种模式下，没有在导入的定义中明确定义的现有 API 中的项目会被删除。

要合并 API，请将一个 PUT 请求提交至 `https://apigateway.`*`<region>`*`.amazonaws.com/restapis/`*`<restapi_id>`*`?mode=merge`。`restapi_id` 路径参数值指定了将要与提供的 API 定义合并的 API。

以下代码段显示了一个 PUT 请求的示例，该请求将作为负载的 JSON 格式的 OpenAPI API 定义与 API Gateway 中已指定的 API 合并。

```
PUT /restapis/<restapi_id>?mode=merge
Host:apigateway.<region>.amazonaws.com
Content-Type: application/json
```

```
Content-Length: ...

An OpenAPI API definition in JSON
```

合并更新操作需要提取两个完整的 API 定义并将它们合并到一起。对于小型增量变更，您可以使用资源更新操作。

要覆盖 API，请将一个 PUT 请求提交至 `https://apigateway.<region>.amazonaws.com/restapis/<restapi_id>?mode=overwrite`。`restapi_id` 路径参数指定了将要被提供的 API 定义覆盖的 API。

以下代码段显示了一个覆盖请求的示例，该请求的负载为 JSON 格式 OpenAPI 定义：

```
PUT /restapis/<restapi_id>?mode=overwrite
Host:apigateway.<region>.amazonaws.com
Content-Type: application/json
Content-Length: ...

An OpenAPI API definition in JSON
```

如果未指定 `mode` 查询参数，则系统会假定合并。

> **Note**
>
> PUT 操作是幂等操作，不是原子操作。这意味着如果在处理过程中出现系统错误，则 API 可能会以不良状态结束。但是，重复该操作会成功将 API 置于相同的最终状态，如同第一次操作已成功一样。

设置 OpenAPI `basePath` 属性

在 OpenAPI 2.0 中，您可以使用 `basePath` 属性来提供 `paths` 属性中定义的每个路径之前的一个或多个路径部分。由于 API Gateway 具有多种表达资源路径的方式，因此导入 API 功能可提供以下选项用于解释导入过程中的 `basePath` 属性：ignore、prepend 和 split。

在 OpenAPI 3.0 中，`basePath` 不再是顶级属性。相反，作为惯例，API Gateway 使用服务器变量。导入 API 功能提供了相同选项用于解释导入过程中的基本路径。按如下所示标识基本路径：

- 如果 API 不包含任何 basePath 变量，则导入 API 功能将检查 `server.url` 字符串以查看其是否包含 "/" 之外的路径。如果包含，则将该路径用作基本路径。
- 如果 API 仅包含一个 basePath 变量，则导入 API 功能将使用其作为基本路径，即使该变量未在 `server.url` 中进行引用。
- 如果 API 包含多个 basePath 变量，则导入 API 功能将仅使用第一个变量作为基本路径。

忽略

如果 OpenAPI 文件的 `basePath` 值为 `/a/b/c` 且 `paths` 属性包含 `/e` 和 `/f`，则以下 `POST` 或 `PUT` 请求：

```
POST /restapis?mode=import&basepath=ignore
```

```
PUT /restapis/api_id?basepath=ignore
```

将在 API 中生成以下资源：

- `/`
- `/e`
- `/f`

效果是将 `basePath` 视为不存在，所有声明的 API 资源均相对于主机提供。例如，如果您有一个自定义域名，其 API 映射不包含基础路径和表示生产阶段的阶段值，则可以使用这一选项。

> **Note**
>
> API Gateway 自动为您创建一个根资源，即使该资源未在定义文件中明确声明。

如未指定，`basePath` 以 `ignore` 为默认值。

前置

如果 OpenAPI 文件的 `basePath` 值为 `/a/b/c` 且 `paths` 属性包含 `/e` 和 `/f`，则以下 `POST` 或 `PUT` 请求：

```
POST /restapis?mode=import&basepath=prepend
```

```
PUT /restapis/api_id?basepath=prepend
```

将在 API 中生成以下资源：

- `/`
- `/a`
- `/a/b`
- `/a/b/c`
- `/a/b/c/e`
- `/a/b/c/f`

效果是将 `basePath` 视为指定其他资源 (不含方法) 并将这些资源添加到声明的资源组中。例如，如果不同的团队负责一个 API 的不同部分且 `basePath` 可以为每个团队所负责 API 部分引用路径位置，则可以使用这一选项。

> **Note**
>
> API Gateway 自动为您创建中间资源，即使这些资源未在定义中明确声明。

Split

如果 OpenAPI 文件的 `basePath` 值为 `/a/b/c` 且 `paths` 属性包含 `/e` 和 `/f`，则以下 `POST` 或 `PUT` 请求：

```
POST /restapis?mode=import&basepath=split
```

```
PUT /restapis/api_id?basepath=split
```

将在 API 中生成以下资源：

- `/`
- `/b`
- `/b/c`
- `/b/c/e`

- `/b/c/f`

效果是将最顶层的路径部分 `/a` 视为每个资源路径的开始，并在 API 自身内创建其他资源 (不含方法)。例如，如果 a 是一个您想在 API 中使用的阶段名称，则可以使用这一选项。

用于 OpenAPI 导入的 AWS 变量

您可以在 OpenAPI 定义中使用以下 AWS 变量。API Gateway 在导入 API 时解析变量。要指定变量，请使用 `${variable-name}`。

AWS 变量

变量名称	说明
`AWS::AccountId`	导入 API 的 AWS 账户 ID（例如，123456789012）。
`AWS::Partition`	在其中导入 API 的 AWS 分区。对于标准 AWS 区域，分区是 `aws`。
`AWS::Region`	在其中导入 API 的 AWS 区域，例如 `us-east-2` 。

AWS 变量示例

以下示例使用 AWS 变量为集成指定 AWS Lambda 函数。

OpenAPI 3.0

```
openapi: "3.0.1"
info:
  title: "tasks-api"
  version: "v1.0"
paths:
  /:
    get:
      summary: List tasks
      description: Returns a list of tasks
      responses:
```

```
        200:
          description: "OK"
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: "#/components/schemas/Task"
        500:
          description: "Internal Server Error"
          content: {}
      x-amazon-apigateway-integration:
        uri:
          arn:${AWS::Partition}:apigateway:${AWS::Region}:lambda:path/2015-03-31/
functions/arn:${AWS::Partition}:lambda:${AWS::Region}:
${AWS::AccountId}:function:LambdaFunctionName/invocations
        responses:
          default:
            statusCode: "200"
        passthroughBehavior: "when_no_match"
        httpMethod: "POST"
        contentHandling: "CONVERT_TO_TEXT"
        type: "aws_proxy"
components:
  schemas:
    Task:
      type: object
      properties:
        id:
          type: integer
        name:
          type: string
        description:
          type: string
```

导入期间的错误和警告

导入过程中出现错误

在导入过程中，可能会因 OpenAPI 文档无效等严重问题而产生错误。系统会在不成功响应中将错误作为异常（如 `BadRequestException`）返回。出现错误时，新的 API 定义会被丢弃，现有 API 不会发生更改。

导入过程中出现警告

在导入过程中，可能会因缺失模型引用等轻微问题而产生警告。出现警告时，如果请求 URL 中附加了 `failonwarnings=false` 查询表达式，则操作将会继续。否则更新就会回滚。默认情况下，将 `failonwarnings` 设置为 `false`。在这种情况下，系统会在生成的 RestApi 资源中以字段形式返回警告。否则，系统会将警告作为异常中的一条消息返回。

从 API Gateway 导出 REST API

使用 API Gateway 控制台或其他方式在 API Gateway 中创建和配置 REST API 后，您可以使用 API Gateway 导出 API（该 API 是 Amazon API Gateway 控制服务的一部分）将其导出到 OpenAPI 文件。要使用 API Gateway 导出 API，您需要签署您的 API 请求。有关签署请求的更多信息，请参阅《IAM 用户指南》中的签署 AWS API 请求。您可以在导出的 OpenAPI 定义文件中包含 API Gateway 集成扩展以及 Postman 扩展。

> **Note**
>
> 使用 AWS CLI 导出 API 时，请务必包含扩展参数，如以下示例所示，以确保包含 `x-amazon-apigateway-request-validator` 扩展：
>
> ```
> aws apigateway get-export --parameters extensions='apigateway' --rest-api-id
>  abcdefg123 --stage-name dev --export-type swagger latestswagger2.json
> ```

如果 API 的负载并非 `application/json` 类型，则您无法将其导出。如果尝试导出，您将收到一条错误响应，指出未找到 JSON 正文模型。

请求导出 REST API

借助 Export API，您可以提交 GET 请求，将要导出的 REST 指定为 URL 路径的一部分，以此来导出现有 REST API。请求 URL 的格式如下：

OpenAPI 3.0

```
https://<host>/restapis/<restapi_id>/stages/<stage_name>/exports/oas30
```

OpenAPI 2.0

```
https://<host>/restapis/<restapi_id>/stages/<stage_name>/exports/swagger
```

您可以附加 `extensions` 查询字符串，以指定是否要包含 API Gateway 扩展（含 `integration` 值）或 Postman 扩展（含 `postman` 值）。

此外，您还可以将 `Accept` 标头设置为 `application/json` 或 `application/yaml`，以分别接收 JSON 格式或 YAML 格式的 API 定义输出。

有关使用 API Gateway 导出 API 提交 GET 请求的更多信息，请参阅 GetExport。

> **Note**
>
> 如果您在 API 中定义模型，那么这些模型的内容类型必须为“application/json”，这样 API Gateway 才能将其导出。否则，API Gateway 会引发异常，并显示“仅找到适用于……的非 JSON 正文模型”的错误消息。
> 模型必须包含属性或者被定义为特定 JSONSchema 类型。

下载 JSON 格式的 REST API OpenAPI 定义

要下载 JSON 格式的 REST API OpenAPI 定义，请执行以下操作：

OpenAPI 3.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/oas30
Host: apigateway.<region>.amazonaws.com
Accept: application/json
```

OpenAPI 2.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/swagger
Host: apigateway.<region>.amazonaws.com
```

```
Accept: application/json
```

这里的 *<region>* 可以是 (比如说) `us-east-1`。有关提供 API Gateway 的所有区域，请参阅 Regions and Endpoints。

下载 YAML 格式的 REST API OpenAPI 定义

要下载 YAML 格式的 REST API OpenAPI 定义，请执行以下操作：

OpenAPI 3.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/oas30
Host: apigateway.<region>.amazonaws.com
Accept: application/yaml
```

OpenAPI 2.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/swagger
Host: apigateway.<region>.amazonaws.com
Accept: application/yaml
```

借助 Postman 扩展下载 JSON 格式的 REST API OpenAPI 定义

借助 Postman 导出并下载 JSON 格式的 REST API OpenAPI 定义，请执行以下操作：

OpenAPI 3.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/oas30?extensions=postman
Host: apigateway.<region>.amazonaws.com
Accept: application/json
```

OpenAPI 2.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/swagger?extensions=postman
Host: apigateway.<region>.amazonaws.com
Accept: application/json
```

借助 API Gateway 集成下载 YAML 格式的 REST API OpenAPI 定义

要借助 API Gateway 集成导出并下载 YAML 格式的 REST API OpenAPI 定义，请执行以下操作：

OpenAPI 3.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/oas30?extensions=integrations
Host: apigateway.<region>.amazonaws.com
Accept: application/yaml
```

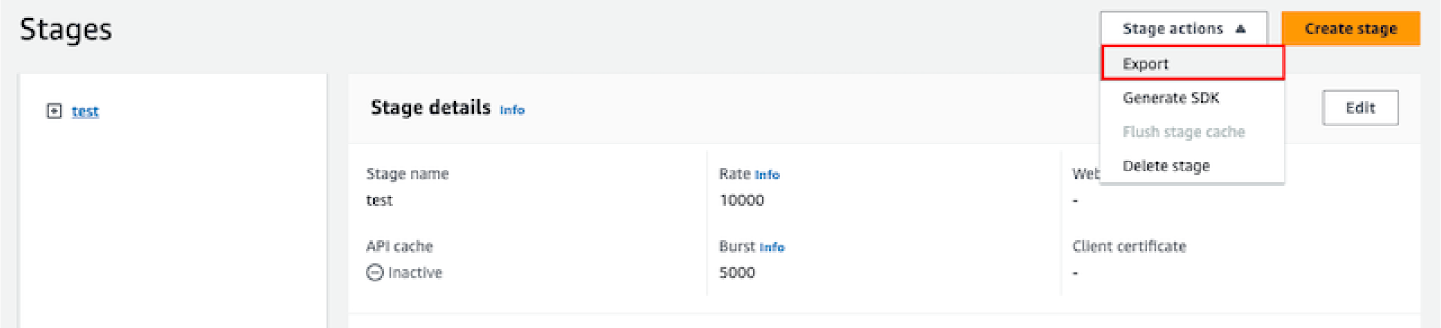
OpenAPI 2.0

```
GET /restapis/<restapi_id>/stages/<stage_name>/exports/swagger?
extensions=integrations
Host: apigateway.<region>.amazonaws.com
Accept: application/yaml
```

使用 API Gateway 控制台导出 REST API

将 REST API 部署到一个阶段之后，您可以使用 API Gateway 控制台将此阶段中的 API 导出到 OpenAPI 文件。

在 API Gateway 控制台的阶段窗格中，选择阶段操作，然后选择导出。



Stage name
test

API cache
Inactive

Rate
10000

Burst
5000

Client certificate
-

指定 API 规范类型、格式和扩展以下载您 API 的 OpenAPI 定义。

发布 REST API 供客户调用

仅仅创建和开发 API Gateway API 并不会自动使其可供用户调用。要使 API 可供调用，您必须将其部署到一个阶段。此外，建议您自定义用户将用于访问 API 的 URL。您可以为其提供一个与品牌一致的域，或者使其比 API 的默认 URL 更容易记住。

在本节中，您可以了解如何部署 API 并自定义您提供给用户以访问 API 的 URL。

> **Note**
>
> 为了增强您的 API Gateway API 的安全性，将在公共后缀列表 (PSL) 中注册 `execute-api.{region}.amazonaws.com` 域。为进一步增强安全性，如果您需要在 API Gateway API 的默认域名中设置敏感 Cookie，我们建议您使用带 `__Host-` 前缀的 Cookie。这将有助于保护您的域，防范跨站点请求伪造 (CSRF) 攻击。要了解更多信息，请参阅 Mozilla 开发者网络中的 Set-Cookie 页面。

主题

- 在 Amazon API Gateway 中部署 REST API
- 为 REST API 设置自定义域名

在 Amazon API Gateway 中部署 REST API

创建 API 之后，您必须对其进行部署，以便您的用户可以调用它。

要部署 API，您可以创建 API 部署并将其与阶段关联。一个阶段是对您 API 生命周期状态（例如，`dev`、`prod`、`beta`、`v2`）的一次逻辑引用。API 阶段由 API ID 和阶段名称标识。它们包含在您用于调用 API 的 URL 中。每个阶段都是一个对 API 部署的命名引用，可供客户端应用程序调用。

⚠ Important

每次更新 API 时，您都必须将 API 重新部署到现有阶段或新阶段。更新 API 涉及修改路由、方法、集成、授权方、资源策略以及除阶段设置之外的任何其他内容。

随着 API 的发展，您可以继续将其作为 API 的不同版本部署到不同阶段。您还可以将 API 更新部署为 Canary 版本部署。这使您的 API 客户端可在相同阶段上，通过生产版本访问正式版本，并通过 Canary 版本访问更新的版本。

为调用已部署 API，客户端对 API 的 URL 提交请求。URL 由 API 的协议（HTTP(S) 或 (WSS)）、主机名、阶段名称和（对于 REST API）资源路径确定。主机名和阶段名称确定 API 的基本 URL。

使用 API 的默认域名，给定阶段 (*{stageName}*) 中的 REST API 的基本 URL（例如）采用以下格式：

```
https://{restapi-id}.execute-api.{region}.amazonaws.com/{stageName}
```

要使 API 的默认基本 URL 对用户更友好，您可以创建自定义域名（例如 `api.example.com`）来替换该 API 的默认主机名。要支持自定义域名下的多个 API，您必须将 API 阶段映射到基本路径。

使用自定义域名 *{api.example.com}*，以及在自定义域名下映射到基本路径 (*{basePath}*) 的 API 阶段，REST API 的基本 URL 将如下所示：

```
https://{api.example.com}/{basePath}
```

对于每个阶段，您可以通过调整默认账户级别请求限制并启用 API 缓存来优化 API 性能。您还可以启用对 CloudTrail 或 CloudWatch 的 API 调用的日志记录，并为后端选择客户端证书以对 API 请求进行身份验证。此外，您也可以覆盖各个方法的阶段级别设置并定义阶段变量，以便在运行时将特定于阶段的环境上下文传递到 API 集成中。

阶段实现了对 API 的可靠版本控制。例如，您可以将 API 部署到 `test` 阶段和 `prod` 阶段，并使用 `test` 阶段作为测试版本，使用 `prod` 阶段作为稳定版本。更新通过测试之后，您可以将 `test` 阶段提升到 `prod` 阶段。可以通过将 API 重新部署到 `prod` 阶段或者将阶段变量值从 `test` 的阶段名称更新为 `prod` 的阶段名称来完成提升。

在本部分中，我们讨论如何使用 API Gateway 控制台或调用 API Gateway REST API 来部署 API。要使用其他工具，请参阅AWS CLI 的文档或者AWS开发工具包。

主题

- 在 API Gateway 中创建 REST API。
- 设置 REST API 的阶段
- 设置 API Gateway Canary 版本部署
- 需要重新部署的对 REST API 的更新

在 API Gateway 中创建 REST API。

在 API Gateway 中，以部署资源来表示REST API 部署。它类似于由 RestApi 资源表示的 API 的可执行文件。

要让客户端调用 API，您必须创建部署并将阶段与其关联。阶段由阶段资源表示。它代表 API 的快照，包括方法、集成、模型、映射模板、Lambda 授权方（以前称为自定义授权方）。更新 API 时，您可以通过将新阶段与现有部署关联来重新部署 API。我们在the section called “设置阶段”中介绍了创建阶段。

主题

- 使用 AWS CLI 创建部署
- 从 API Gateway 控制台部署 REST API

使用 AWS CLI 创建部署

创建部署时，您实例化部署资源。您可以使用 API Gateway 控制台、AWS CLI、AWS开发工具包或 API Gateway REST API 创建部署。

要使用 CLI 创建部署，请使用 create-deployment 命令：

```
aws apigateway create-deployment --rest-api-id <rest-api-id> --region <region>
```

在您将此部署与阶段关联之前，API 不可调用。对于现有阶段，您可以使用新创建的部署 ID (`deploymentId`) 更新阶段的 `<deployment-id>` 属性来完成此操作。

```
aws apigateway update-stage --region <region> \
   --rest-api-id <rest-api-id> \
   --stage-name <stage-name> \
   --patch-operations op='replace',path='/deploymentId',value='<deployment-id>'
```

首次部署 API 时，您可以将阶段创建和部署创建结合起来同时进行：

```
aws apigateway create-deployment --region <region> \
   --rest-api-id <rest-api-id> \
   --stage-name <stage-name>
```

这是在您首次部署 API 或者将 API 重新部署到新阶段时，API Gateway 控制台在后台完成的任务。

从 API Gateway 控制台部署 REST API

您必须先创建 REST API，然后才能对其进行首次部署。有关更多信息，请参阅 在 API Gateway 中开发 REST API。

主题

- 将 REST API 部署到阶段
- 将 REST API 重新部署到阶段
- 更新 REST API 部署的阶段配置
- 为 REST API 部署设置阶段变量
- 将阶段与不同的 REST API 部署相关联

将 REST API 部署到阶段

借助 API Gateway 控制台，您可以创建部署并将其与新的或现有阶段相关联，从而部署 API。

> **Note**
>
> 要在 API Gateway 中将阶段与不同的部署相关联，请改为参阅将阶段与不同的 REST API 部署相关联。

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在 APIs 导航窗格中，选择您要部署的 API。
3. 在资源窗格中，选择 Deploy API (部署 API)。
4. 对于阶段，请从以下选项中进行选择：

    a. 要创建新阶段，请选择新建阶段，然后在阶段名称中输入名称。或者，您可以在部署描述中为部署提供描述。

   b. 要选择现有阶段，请从下拉菜单中选择阶段名称。您可能需要在部署描述中为新部署提供描述。
   c. 要创建没有与阶段关联的部署，请选择没有阶段。稍后，您可以将此部署与阶段相关联。
5. 选择部署。

将 REST API 重新部署到阶段

要重新部署 API，请执行与the section called “将 REST API 部署到阶段”中相同的步骤。您可以根据需要多次重复使用同一个阶段。

更新 REST API 部署的阶段配置

部署完 API 后，您可以修改阶段设置，以启用或禁用 API 缓存、日志记录或请求限制。您还可以为后端选择客户端证书以对 API Gateway 进行身份验证，并设置阶段变量，从而在运行时将部署上下文传递至 API 集成。有关更多信息，请参阅 更新阶段设置。

> ⚠ Important
>
> 修改阶段设置后，您必须重新部署 API 才能使更改生效。

> ⓘ Note
>
> 如果启用日志记录等更新的设置要求使用新的 IAM 角色，您无需重新部署 API 即可添加所需的 IAM 角色。但是，新的 IAM 角色可能需要几分钟才能生效。在该角色生效之前，即使您已启用日志记录选项，系统也不会记录对 API 调用的跟踪。

为 REST API 部署设置阶段变量

对于部署，您可以设置或修改阶段变量，从而在运行时将特定于部署的数据传递至 API 集成。您可以在 Stage Editor (阶段编辑器) 中的 Stage Variables (阶段变量) 选项卡上执行此操作。有关更多信息，请参阅为 REST API 部署设置阶段变量中的说明。

将阶段与不同的 REST API 部署相关联

由于部署代表 API 快照，而阶段可定义到快照的路径，因此您可以选择不同的部署阶段组合，以控制用户如何调用不同版本的 API。这非常有用，例如，如果您想将 API 状态回滚至上一个部署，或者将 API 的“私有分支”合并到公有分支中，就可以这样做。

以下过程介绍如何在 API Gateway 控制台中使用阶段编辑器执行此操作。我们假定您已多次部署 API。

1. 如果您尚未打开阶段窗格，请在主导航窗格中选择阶段。
2. 选择要更新的阶段。
3. 在部署历史记录选项卡上，选择您希望阶段使用的部署。
4. 选择更改活动部署。
5. 确认要更改活动部署，然后在设为活动部署对话框中，选择更改活动部署。

设置 REST API 的阶段

阶段是对某个部署的指定引用，是 API 的快照。您可以使用阶段来管理和优化特定的部署。例如，您可以配置阶段设置，以便启用缓存、自定义请求限制、配置日志记录、定义阶段变量或者附加 Canary 版本进行测试。

主题

- 使用 API Gateway 控制台设置阶段
- 在 API Gateway 中为 API 阶段设置标签
- 为 REST API 部署设置阶段变量

使用 API Gateway 控制台设置阶段

主题

- 创建新阶段
- 更新阶段设置
- 覆盖阶段级别设置
- 删除阶段

创建新阶段

初始部署后，您可以添加更多阶段，并将它们与现有部署进行关联。在部署 API 时，您可以使用 API Gateway 控制台来创建新阶段或选择现有阶段。一般来说，重新部署 API 之前，您可以向 API 部署中添加新阶段。要使用 API Gateway 控制台创建新阶段，请按照下列步骤操作：

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 在主导航窗格中，选择某个 API 下的阶段。
4. 从阶段导航窗格中，选择创建阶段。
5. 对于阶段名称，请输入名称，例如 **`prod`**。

> Note
>
> 阶段名称只能包含字母数字字符、连字符和下划线。最大长度为 128 个字符。

6. （可选）。对于描述，请输入阶段描述。
7. 从部署中，选择要与此阶段关联的现有 API 部署的日期和时间。
8. 在其它设置下，您可以为阶段指定其它设置。
9. 选择创建阶段。

更新阶段设置

成功部署 API 后，系统将使用默认设置填充阶段。您可以使用控制台或 API Gateway REST API 更改阶段设置，包括 API 缓存和日志记录。以下步骤说明如何使用 API Gateway 控制台的阶段编辑器来执行这一操作。

使用 API Gateway 控制台更新阶段设置

这些步骤假设您已将 API 部署到阶段。

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 在主导航窗格中，选择某个 API 下的阶段。
4. 在阶段窗格中，选择该阶段的名称。
5. 在阶段详细信息部分中，选择编辑。
6. （可选）对于阶段描述，输入描述。
7. 对于其他设置，修改以下设置：

缓存设置

要为阶段启用 API 缓存，请开启配置 API 缓存。然后，配置默认方法级缓存、缓存容量、加密缓存数据、缓存生存时间(TTL)，以及每个密钥缓存失效的任何要求。

在开启默认方法级缓存或为特定方法开启方法级缓存之前，缓存不会处于活动状态。

有关缓存设置的更多信息，请参阅 启用 API 缓存以增强响应能力。

> Note
>
> 如果您为 API 阶段启用 API 缓存，系统可能会针对 API 缓存向您的 AWS 账户收费。缓存没有资格享受 AWS 免费套餐。

节流设置

要为与该 API 相关联的所有方法设置阶段级别节流目标，请开启节流。

针对比率部分，请输入目标率。这是将令牌添加到令牌桶的频率，以每秒请求数计。阶段级别频率不得超过用于配置和运行 REST API 的 API Gateway 配额中规定的账户级别频率。

针对突增部分，请输入目标突增率。突增频率是令牌桶的容量。这允许在一段时间内通过比目标速率更多的请求。该阶段级别突增率不得超过在 用于配置和运行 REST API 的 API Gateway 配额 中指定的账户级别突增比率。

> Note
>
> 节流费率不是硬限制，应基于最佳效果来应用。在某些情况下，客户端可能会超过您设置的目标。不要依靠使用节流来控制成本或阻止对 API 的访问。考虑使用 AWS Budgets 监控成本和 AWS WAF 来管理 API 请求。

防火墙和证书设置

要将 AWS WAF Web ACL 与阶段关联，请从 Web ACL 下拉列表中选择一个 Web ACL。如果需要，请选择如果无法评估 WebACL，则阻止 API 请求(失败 - 关闭)。

要为您的阶段选择客户端证书，请从客户端证书下拉菜单中选择证书。

8. 选择保存。
9. 要为与此 API Gateway API 的此阶段相关联的所有方法启用 Amazon CloudWatch Logs，请在日志和跟踪部分选择编辑。

> **Note**
>
> 要启用 CloudWatch Logs，您还必须指定 IAM 角色的 ARN，该角色使 API Gateway 能够代表您的用户将信息写入 CloudWatch Logs。为此，请从 APIs 主导航窗格中选择设置。然后，对于 CloudWatch 日志角色，输入 IAM 角色的 ARN。
> 对于常见的应用程序场景，IAM 角色可以附加 `AmazonAPIGatewayPushToCloudWatchLogs` 的托管策略，其中包括以下访问策略语句：
>
> ```
> {
>     "Version": "2012-10-17",
>     "Statement": [
>         {
>             "Effect": "Allow",
>             "Action": [
>                 "logs:CreateLogGroup",
>                 "logs:CreateLogStream",
>                 "logs:DescribeLogGroups",
>                 "logs:DescribeLogStreams",
>                 "logs:PutLogEvents",
>                 "logs:GetLogEvents",
>                 "logs:FilterLogEvents"
>             ],
>             "Resource": "*"
>         }
>     ]
> }
> ```
>
> IAM 角色还必须包含以下信任关系语句：
>
> ```
> {
>   "Version": "2012-10-17",
>   "Statement": [
>     {
>       "Sid": "",
>       "Effect": "Allow",
>       "Principal": {
> ```

```
            "Service": "apigateway.amazonaws.com"
          },
          "Action": "sts:AssumeRole"
        }
      ]
    }
```

有关 CloudWatch 的更多信息，请参阅 Amazon CloudWatch 用户指南。

10. 从 CloudWatch Logs 下拉菜单中选择日志记录级别。日志记录级别如下所示：

    - 关闭 - 不对此阶段开启日志记录。
    - 仅限错误 - 仅对错误启用日志记录。
    - 错误和信息日志 - 对所有事件启用日志记录。
    - 完整的请求和响应日志 - 对所有事件启用详细日志记录。这对于排除 API 故障非常有用，但可能会导致记录敏感数据。

    > **Note**
    >
    > 我们建议不要为生产 API 使用完整请求和响应日志。

11. 选择详细指标，让 API Gateway 向 CloudWatch 报告 `API calls`、`Latency`、`Integration latency`、`400 errors` 和 `500 errors` 的 API 指标。有关 CloudWatch 的更多信息，请参阅《Amazon CloudWatch 用户指南》中的基本监控和详细监控。

    > **Important**
    >
    > 我们将针对访问方法级别 CloudWatch 指标而非 API 级或存储级指标向您的账户收费。

12. 要启用对目标的访问日志记录，请打开自定义访问日志记录。

13. 对于访问日志目标 ARN，输入日志组或 Firehose 流的 ARN。

    Firehose 的 ARN 格式为 `arn:aws:firehose:`*`{region}`*`:`*`{account-id}`*`:deliverystream/amazon-apigateway-`*`{your-stream-name}`*。Firehose 流的名称必须为 `amazon-apigateway-`*`{your-stream-name}`*。

14. 在日志格式中，输入日志格式。要了解有关示例日志格式的更多信息，请参阅the section called “用于 API Gateway 的 CloudWatch 日志格式”。

15. 要为 API 阶段启用 AWS X-Ray 跟踪，请选择 X-Ray 跟踪。有关更多信息，请参阅 使用 X-Ray 跟踪用户对 REST API 的请求。
16. 选择 Save changes（保存更改）。重新部署 API 以使新设置生效。

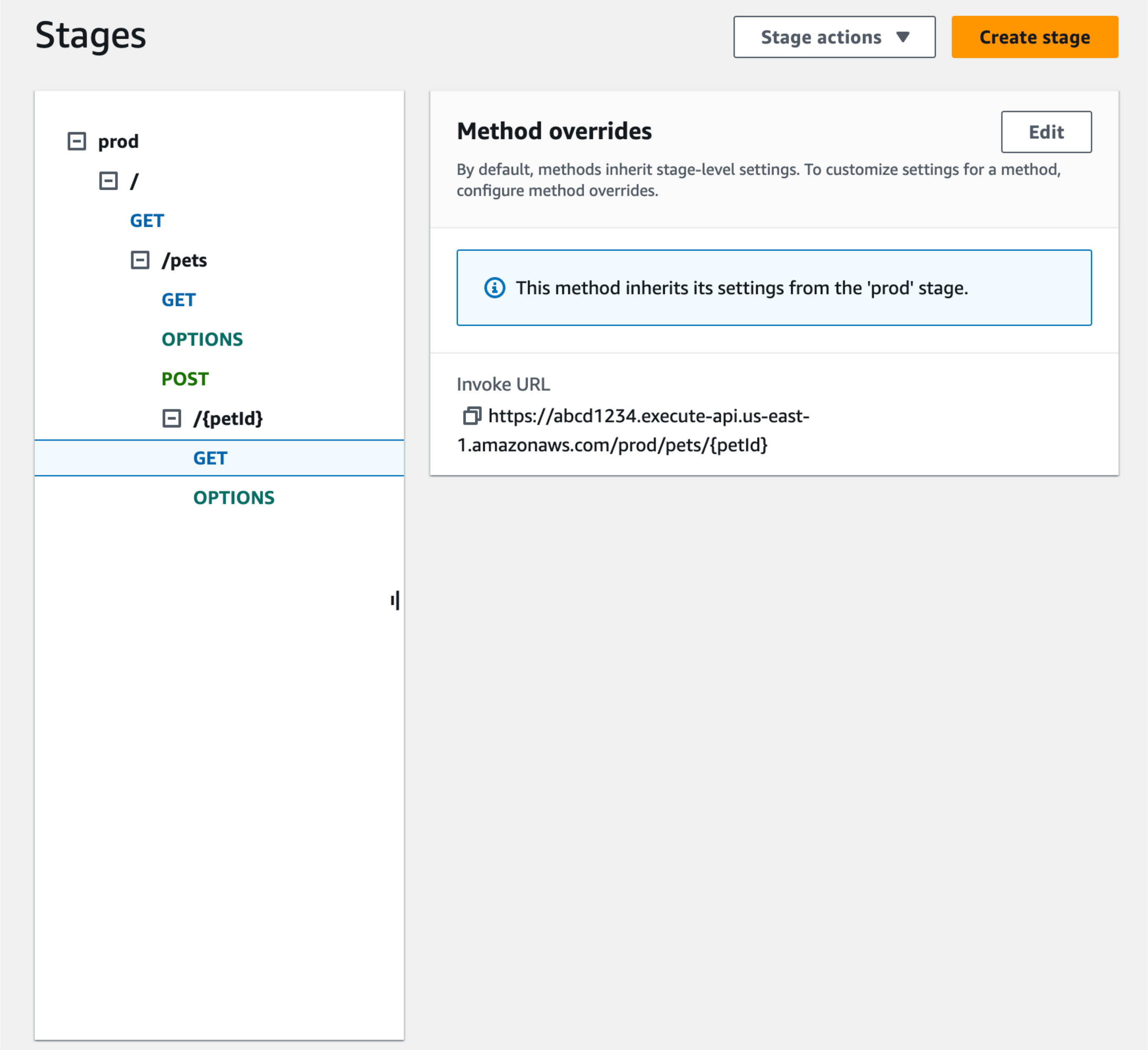
覆盖阶段级别设置

您可以覆盖以下已启用的阶段级别设置。其中一些选项可能会导致您的 AWS 账户产生额外费用。

使用 API Gateway 控制台覆盖阶段级别设置

使用 API Gateway 控制台覆盖阶段级别设置

1. 要配置方法覆盖，请在二级导航窗格下展开阶段，然后选择一种方法。



By default, methods inherit stage-level settings. To customize settings for a method, configure method overrides.

This method inherits its settings from the 'prod' stage.

Invoke URL

https://abcd1234.execute-api.us-east-1.amazonaws.com/prod/pets/{petId}

2. 对于方法覆盖，选择编辑。
3. 要开启方法级别 CloudWatch 设置，请为 CloudWatch Logs 选择一个日志记录级别。
4. 要开启方法级别详细指标，请选择详细指标。我们将针对访问方法级别 CloudWatch 指标而非 API 级或存储级指标向您的账户收费。
5. 要开启方法级别的节流，请选择节流。输入适合的方法级别选项。要了解有关节流的更多信息，请参阅 the section called “限制”。
6. 要配置方法级别缓存，请选择启用方法缓存。如果您在阶段详细信息中更改默认方法级缓存设置，它不会影响此设置。
7. 选择保存。

删除阶段

当您不再需要某个阶段时，可以将其删除，以免为未使用的资源付费。以下步骤介绍如何使用 API Gateway 控制台删除阶段。

> ⚠ Warning
>
> 删除阶段可能会导致 API 调用方无法使用部分或全部相应的 API。删除阶段无法撤销，但您可以重新创建阶段并将其与相同的部署相关联。

使用 API Gateway 控制台删除阶段

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 在主导航窗格中，选择阶段。
4. 在阶段窗格中，选择您要删除的阶段，然后依次选择阶段操作和删除阶段。
5. 当系统提示您时，输入 **`confirm`**，然后选择删除。

在 API Gateway 中为 API 阶段设置标签

在 API Gateway 中，您可以将标签添加到 API 阶段、从阶段中删除标签或者查看标签。为此，您可以使用 API Gateway 控制台、AWS CLI/开发工具包或 API Gateway REST API。

阶段还可以从其父 REST API 继承标签。有关更多信息，请参阅 the section called “Amazon API Gateway V1 API 中的标签继承”。

有关标记 API Gateway 资源的更多信息，请参阅 Tagging。

主题

- 使用 API Gateway 控制台为 API 阶段设置标签
- 使用 AWS CLI 为 API 阶段设置标签
- 使用 API Gateway REST API 为 API 阶段设置标签

使用 API Gateway 控制台为 API 阶段设置标签

以下过程将介绍如何为 API 阶段设置标签。

使用 API Gateway 控制台为 API 阶段设置标签

1. 登录 API Gateway 控制台。
2. 选择一个现有 API，或者创建包括资源、方法和对应集成的新 API。
3. 选择阶段或者将 API 部署到新阶段。
4. 在主导航窗格中，选择阶段。
5. 选择标签选项卡。您可能需要选择右箭头按钮以显示该选项卡。
6. 选择管理标签。
7. 在标签编辑器中，选择添加标签。在键字段中输入标签键（例如 `Department`），在值字段中输入标签值（例如 `Sales`）。选择保存以保存标签。
8. 如果需要，请重复第 5 步，将更多标签添加到 API 阶段。每个阶段的最大标签数是 50。
9. 要从阶段中移除现有标签，请选择移除。
10. 如果先前已在 API Gateway 控制台中部署了 API，则需要重新进行部署，以使更改生效。

使用 AWS CLI 为 API 阶段设置标签

您可以使用 AWS CLI，通过 create-stage 命令或 tag-resource 命令为 API 阶段设置标签。您可以使用 untag-resource 命令从 API 阶段中删除一个或多个标签。

以下示例在创建 `test` 阶段时添加一个标签：

```
aws apigateway create-stage --rest-api-id abc1234 --stage-name test --description
 'Testing stage' --deployment-id efg456 --tag Department=Sales
```

以下示例向 `prod` 阶段添加一个标签：

```
aws apigateway tag-resource --resource-arn arn:aws:apigateway:us-east-2::/
restapis/abc123/stages/prod --tags Department=Sales
```

以下示例从 `test` 阶段移除 `Department=Sales` 标签。

```
aws apigateway untag-resource --resource-arn arn:aws:apigateway:us-east-2::/
restapis/abc123/stages/test --tag-keys Department
```

使用 API Gateway REST API 为 API 阶段设置标签

您可以执行以下操作之一，使用 API Gateway REST API 为 API 阶段设置标签：

- 调用 `tags:tag` 为 API 阶段添加标签。
- 调用 `tags:untag` 以从 API 阶段中删除一个或多个标签。
- 调用 `stage:create` 以将一个或多个标签添加到您正创建的 API 阶段。

您还可以调用 `tags:get` 以描述 API 阶段中的标签。

为 API 阶段添加标签

将 API (`m5zr3vnks7`) 部署到阶段 (`test`) 之后，通过调用 `tags:tag` 为阶段添加标签。所需的阶段 Amazon Resource Name (ARN) (`arn:aws:apigateway:us-east-1::/restapis/m5zr3vnks7/stages/test`) 必须为 URL 编码 (`arn%3Aaws%3Aapigateway%3Aus-east-1%3A%3A%2Frestapis%2Fm5zr3vnks7%2Fstages%2Ftest`)。

```
PUT /tags/arn%3Aaws%3Aapigateway%3Aus-east-1%3A%3A%2Frestapis%2Fm5zr3vnks7%2Fstages
%2Ftest

{
  "tags" : {
    "Department" : "Sales"
  }
}
```

您还可以使用之前的请求，将现有标签更新为新值。

在调用 `stage:create` 创建阶段时，您可以将标签添加到阶段。

```
POST /restapis/<restapi_id>/stages

{
  "stageName" : "test",
  "deploymentId" : "adr134",
  "description" : "test deployment",
  "cacheClusterEnabled" : "true",
  "cacheClusterSize" : "500",
  "variables" : {
    "sv1" : "val1"
  },
  "documentationVersion" : "test",

  "tags" : {
    "Department" : "Sales",
```

```
    "Division" : "Retail"
  }
}
```

取消 API 阶段的标签

要从阶段中删除 `Department` 标签，请调用 [`tags:untag`]：

```
DELETE /tags/arn%3Aaws%3Aapigateway%3Aus-east-1%3A%3A%2Frestapis%2Fm5zr3vnks7%2Fstages
%2Ftest?tagKeys=Department
Host: apigateway.us-east-1.amazonaws.com
Authorization: ...
```

要删除多个标签，请在查询表达式中使用逗号分隔的标签键列表，例如？`tagKeys=Department,Division,…`。

描述 API 阶段的标签

要描述指定阶段上的现有标签，请调用 [`tags:get`]：

```
GET /tags/arn%3Aaws%3Aapigateway%3Aus-east-1%3A%3A%2Frestapis%2Fm5zr3vnks7%2Fstages
%2Ftags
Host: apigateway.us-east-1.amazonaws.com
Authorization: ...
```

成功响应的形式与下方类似：

```
200 OK

{
    "_links": {
        "curies": {
            "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/
restapi-tags-{rel}.html",
            "name": "tags",
            "templated": true
        },
        "tags:tag": {
            "href": "/tags/arn%3Aaws%3Aapigateway%3Aus-east-1%3A%3A%2Frestapis
%2Fm5zr3vnks7%2Fstages%2Ftags"
        },
        "tags:untag": {
```

```
            "href": "/tags/arn%3Aaws%3Aapigateway%3Aus-east-1%3A%3A%2Frestapis
%2Fm5zr3vnks7%2Fstages%2Ftags{?tagKeys}",
            "templated": true
        }
    },
    "tags": {
        "Department": "Sales"
    }
}
```

为 REST API 部署设置阶段变量

阶段变量是您可以定义为与 REST API 部署阶段关联的配置属性的名称-值对。它们与环境变量的功能类似，可用于 API 设置和映射模板。

例如，您可以在阶段配置中定义一个阶段变量，然后针对 REST API 中的方法将其值设置为 HTTP 集成的 URL 字符串。之后，您可以使用 API 设置中的关联阶段变量名称引用该 URL 字符串。这样，您就可以通过将阶段变量值重置为相应的 URL，在每个阶段对不同端点使用同一 API 设置。

此外，您还可以访问映射模板中的阶段变量，或者将配置参数传递给 AWS Lambda 或 HTTP 后端。

有关映射模板的更多信息，请参阅API Gateway 映射模板和访问日志记录变量引用。

> Note
>
> 阶段变量不适用于敏感数据，例如凭证。要将敏感数据传递给集成，请使用 AWS Lambda 授权方。您可以在 Lambda 授权方的输出中将敏感数据传递给集成。要了解更多信息，请参阅“the section called “来自 API Gateway Lambda 授权方的输出””。

使用案例

使用 API Gateway 中的部署阶段，您可以管理各 API 的多个发布阶段，例如内部测试、测试和生产。通过阶段变量，您可以将 API 部署阶段配置为与不同的后端端点交互。

例如，您的 API 可以将 GET 请求作为 HTTP 代理传递给后端 Web 主机 (例如 `http://example.com`)。在这种情况下，后端 Web 主机将在阶段变量中配置，这样一来，当开发人员调用生产端点时，API Gateway 将调用 example.com。当您调用测试端点时，API Gateway 将使用在测试阶段的阶段变量中配置的值，并调用不同的 Web 主机（例如 `beta.example.com`）。同样，阶段变量可用于为 API 中的每个阶段指定不同的 AWS Lambda 函数名称。

您还可以使用阶段变量，通过映射模板将配置参数传递给 Lambda 函数。例如，您可能需要针对 API 中的多个阶段重复使用同一个 Lambda 函数，但是该函数应根据调用的阶段，从不同的 Amazon DynamoDB 表中读取数据。在为 Lambda 函数生成请求的映射模板中，您可以使用阶段变量来将表的名称传递给 Lambda。

示例

要使用阶段变量自定义 HTTP 集成端点，您必须首先配置具有指定名称的阶段变量（例如 **`url`**），然后为其分配一个值（例如 **`example.com`**）。接下来，从您的方法配置中，设置 HTTP 代理集成。您可以告知 API Gateway 使用阶段变量值 **`http://${stageVariables.url}`**，而不是输入端点的 URL。此值将指示 API Gateway 在运行时替换您的阶段变量 `${}`，具体取决于 API 正在哪个阶段运行。

您可以通过与在凭证字段中指定 Lambda 函数名称、AWS服务代理路径或AWS角色 ARN 类似的方式引用阶段变量。

将 Lambda 函数名称指定为阶段变量值时，您必须手动配置对 Lambda 函数的权限。当您在 API Gateway 控制台中指定 Lambda 函数时，将弹出一条 AWS CLI 命令，让您配置正确的权限。您还可以使用 AWS Command Line Interface (AWS CLI) 来执行此操作。

```
aws lambda add-permission --function-name "arn:aws:lambda:us-
east-2:123456789012:function:my-function" --source-arn "arn:aws:execute-api:us-
east-2:123456789012:api_id/*/HTTP_METHOD/resource" --principal apigateway.amazonaws.com
 --statement-id apigateway-access --action lambda:InvokeFunction
```

使用 Amazon API Gateway 控制台设置阶段变量

在本教程中，您将了解如何使用 Amazon API Gateway 控制台为示例 API 的两个部署阶段设置阶段变量。在您开始之前，确保您满足以下先决条件：

- API Gateway 中必须有可用的 API。按照中的说明进行操作在 API Gateway 中开发 REST API
- 您必须至少部署过一次 API。按照中的说明进行操作在 Amazon API Gateway 中部署 REST API
- 您必须为已部署的 API 创建了第一个阶段。按照中的说明进行操作创建新阶段

使用 API Gateway 控制台声明阶段变量

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。

2. 创建 API，然后在 API 的根资源上创建 GET 方法。将集成类型设置为 HTTP，并将端点 URL 设置为 **`http://${stageVariables.url}`**。
3. 将 API 部署到名为 **`beta`** 的新阶段。
4. 在主导航窗格中，选择阶段，然后选择 beta 阶段。
5. 在阶段变量选项卡上，选择编辑。
6. 选择添加阶段变量。
7. 对于名称，请输入 **`url`**。对于值，输入 **`httpbin.org/get`**。
8. 选择添加阶段变量，然后执行以下操作：

   对于名称，请输入 **`stageName`**。对于值，输入 **`beta`**。
9. 选择添加阶段变量，然后执行以下操作：

   对于名称，请输入 **`function`**。对于值，输入 **`HelloWorld`**。

   > **Note**
   >
   > 在将 Lambda 函数设置为阶段变量的值时，请使用函数的本地名称，可能包括其别名或版本规范，例如 **`HelloWorld`**、**`HelloWorld:1`** 或 **`HelloWorld:alpha`**。请勿使用该函数的 ARN（例如 **`arn:aws:lambda:us-east-1:123456789012:function:HelloWorld`**）。API Gateway 控制台将 Lambda 函数的阶段变量值假定为非限定的函数名称，并且会将给定的阶段变量扩展到 ARN 中。
10. 选择保存。
11. 现在创建第二个阶段。从阶段导航窗格中，选择创建阶段。对于阶段名称，输入 **`prod`**。从部署中选择一个新近部署，然后选择创建阶段。
12. 与 beta 阶段一样，将相同的三个阶段变量（url、stageName 和 function）分别设置为不同的值（**`petstore-demo-endpoint.execute-api.com/petstore/pets`**、**`prod`** 和 **`HelloEveryone`**）。

    要了解如何使用阶段变量，请参阅the section called “使用阶段变量 ”。

使用 Amazon API Gateway 阶段变量

您可以使用 API Gateway 阶段变量访问不同 API 部署阶段的 HTTP 和 Lambda 后端。您还可以使用阶段变量将特定于阶段的配置元数据作为查询参数传递给 HTTP 后端，同时作为在输入映射模板中生成的负载传递到 Lambda 函数中。

先决条件

您必须创建两个阶段，并将 url 阶段变量设置为两个不同的 HTTP 端点：一个为分配到两个不同的 Lambda 函数的 function 阶段变量，一个为包含特定于阶段的元数据的 stageName 阶段变量。

使用阶段变量通过 API 访问 HTTP 端点

1. 在阶段导航窗格中，选择 beta 阶段。在阶段详细信息下，选择复制图标以复制 API 的调用 URL，然后在 Web 浏览器中输入 API 的调用 URL。这将启动 API 根资源上的 beta 阶段 GET 请求。

   > **Note**
   >
   > Invoke URL 链接指向 API 在其 beta 阶段的根资源。在 Web 浏览器中输入 URL 会对根资源调用 beta 阶段 GET 方法。如果方法是在子资源而非根资源本身上定义的，则在 Web 浏览器中输入 URL 将返回 `{"message":"Missing Authentication Token"}` 错误响应。在这种情况下，您必须将特定子资源的名称附加到调用 URL 链接。

2. beta 阶段 GET 请求的响应如下所示。您还可以使用浏览器导航到 http://httpbin.org/get，以验证结果。此值已分配到 beta 阶段中的 `url` 变量。这两个响应完全相同。
3. 在阶段导航窗格中，选择 prod 阶段。在阶段详细信息下，选择复制图标以复制 API 的调用 URL，然后在 Web 浏览器中输入 API 的调用 URL。这将启动 API 根资源上的 prod 阶段 GET 请求。
4. prod 阶段 GET 请求的响应如下所示。您可以使用浏览器导航到 http://petstore-demo-endpoint.execute-api.com/petstore/pets，以验证结果。此值已分配到 prod 阶段中的 `url` 变量。这两个响应完全相同。

在查询参数表达式中通过阶段变量将特定于阶段的元数据传递给 HTTP 后端

此过程介绍如何在查询参数表达式中使用阶段变量值将特定于阶段的元数据传递到 HTTP 后端。我们将使用已在 `stageName` 中声明的 使用 Amazon API Gateway 控制台设置阶段变量 阶段变量。

1. 在资源导航窗格中，选择 GET 方法。

   要向方法的 URL 添加查询字符串参数，请选择方法请求选项卡，然后在方法请求设置部分中选择编辑。
2. 选择 URL 查询字符串参数并执行以下操作：
   a. 选择添加查询字符串。
   b. 在名称中，输入 **`stageName`**。

   c. 保持必填和缓存为已关闭状态。

3. 选择保存。
4. 选择集成请求选项卡，然后在集成请求设置部分中，选择编辑。
5. 对于端点 URL，将 **`?stageName=${stageVariables.stageName}`** 附加到先前定义的 URL 值，因此整个端点 URL 为 **`http://${stageVariables.url}?stageName=${stageVariables.stageName}`**。
6. 选择部署 API，然后选择 beta 阶段。
7. 在主导航窗格中，选择阶段。在阶段导航窗格中，选择 beta 阶段。在阶段详细信息下，选择复制图标以复制 API 的调用 URL，然后在 Web 浏览器中输入 API 的调用 URL。

> **Note**
>
> 我们之所以在这里使用测试阶段，是因为 HTTP 端点（由 `url` 变量“http://httpbin.org/get”指定）接受查询参数表达式，并在响应中将其作为 `args` 对象返回。

8. 您将收到以下响应。注意，分配给 `beta` 阶段变量的 `stageName` 作为 `stageName` 参数传递到后端。

```
{
  "args": {
    "stageName": "beta"
  },
  "headers": {
    "Accept": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "AmazonAPIGateway_abcd1234",
    "X-Amzn-Apigateway-Api-Id": "abcd1234",
    "X-Amzn-Trace-Id": "Self=1-abcd-11111111111111;Root=1-11111111-111111111111111"
  },
  "origin": "192.0.2.9",
  "url": "http://httpbin.org/get?stageName=beta"
}
```

使用阶段变量通过 API 调用 Lambda 函数

此过程介绍了如何使用阶段变量调用 Lambda 函数作为 API 后端。我们将使用先前声明的 `function` 阶段变量。有关更多信息，请参阅 [使用 Amazon API Gateway 控制台设置阶段变量]。

1. 使用默认 Node.js 运行时系统创建名为 **`HelloWorld`** 的 Lambda 函数。代码必须包含以下内容：

```
export const handler = function(event, context, callback) {
```

```
    if (event.stageName)
        callback(null, 'Hello, World! I\'m calling from the ' + event.stageName + '
 stage.');
    else
        callback(null, 'Hello, World! I\'m not sure where I\'m calling from...');
};
```

有关如何创建 Lambda 函数的更多信息，请参阅 REST API 控制台入门。

2. 在资源窗格中，选择创建资源，然后执行以下操作：

   a. 对于资源路径，选择 /。

   b. 对于资源名称，输入 **`lambdav1`**。

   c. 选择创建资源。

3. 选择 /lambdav1 资源，然后选择创建方法。

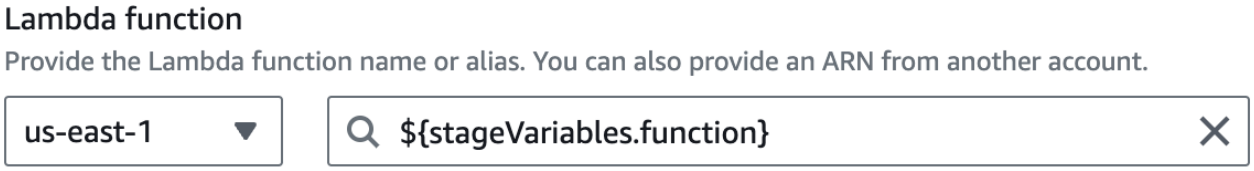
   然后执行以下操作：

   a. 对于方法类型，选择 GET。

   b. 对于集成类型，选择 Lambda 函数。

   c. 保持 Lambda 代理集成处于关闭状态。

   d. 对于 Lambda 函数，输入 `${stageVariables.function}`。



Lambda function

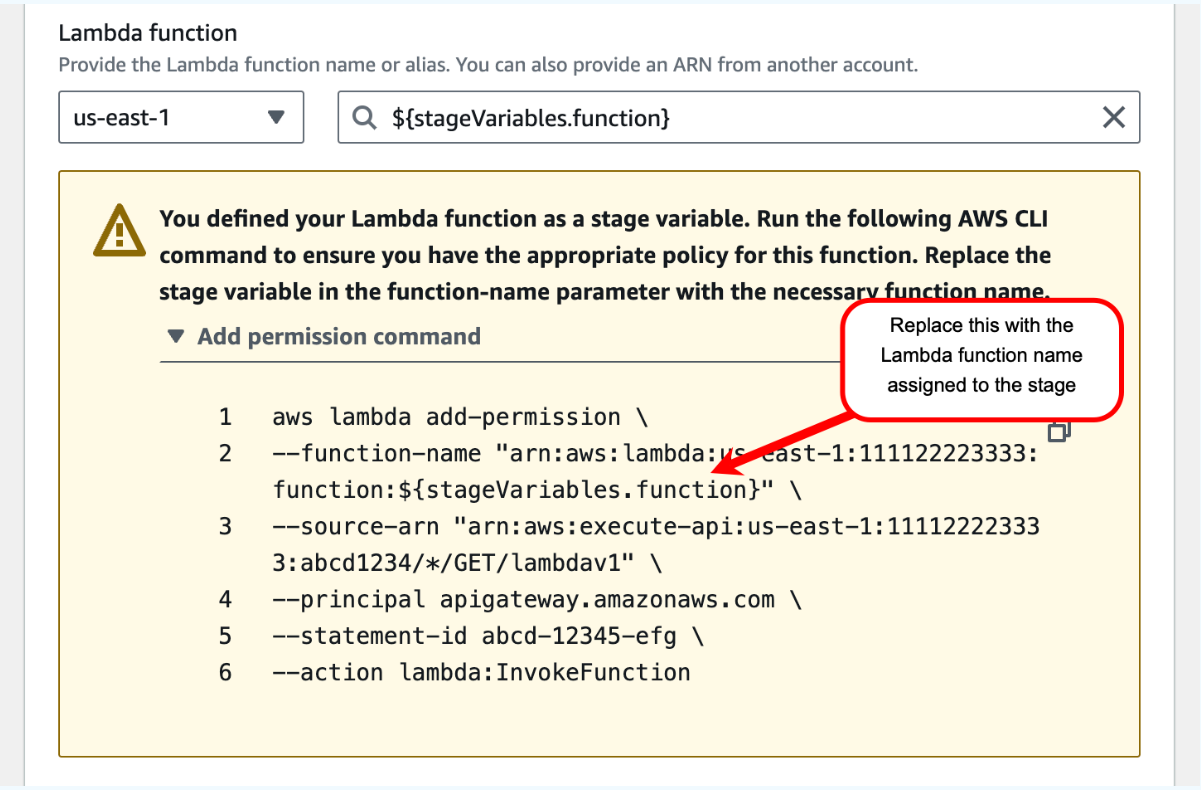
Provide the Lambda function name or alias. You can also provide an ARN from another account.

us-east-1

${stageVariables.function}

> **Tip**
>
> 当提示使用添加权限命令时，复制 AWS CLI 命令。对将分配给 `function` 阶段变量的每个 Lambda 函数运行此命令。例如，如果 `$stageVariables.function` 值为 `HelloWorld`，则运行以下 AWS CLI 命令：
>
> ```
> aws lambda add-permission --function-name arn:aws:lambda:us-
> east-1:account-id:function:HelloWorld --source-arn arn:aws:execute-
> api:us-east-1:account-id:api-id/*/GET/lambdav1 --principal
>  apigateway.amazonaws.com --statement-id statement-id-guid --action
>  lambda:InvokeFunction
> ```

否则会导致在调用该方法时出现 `500 Internal Server Error` 响应。将 `${stageVariables.function}` 替换为已分配给阶段变量的 Lambda 函数名称。



You defined your Lambda function as a stage variable. Run the following AWS CLI command to ensure you have the appropriate policy for this function. Replace the stage variable in the function-name parameter with the necessary function name.

```
aws lambda add-permission \
--function-name "arn:aws:lambda:us-east-1:111122223333:function:${stageVariables.function}" \
--source-arn "arn:aws:execute-api:us-east-1:111122223333:abcd1234/*/GET/lambdav1" \
--principal apigateway.amazonaws.com \
--statement-id abcd-12345-efg \
--action lambda:InvokeFunction
```

Replace this with the Lambda function name assigned to the stage

   e. 选择创建方法。

4. 将 API 部署到 **prod** 和 **beta** 阶段。

5. 在主导航窗格中，选择阶段。在阶段导航窗格中，选择 beta 阶段。在阶段详细信息下，选择复制图标以复制 API 的调用 URL，然后在 Web 浏览器中输入 API 的调用 URL。在按 Enter 之前，将 **/lambdav1** 附加到 URL。

   您将收到以下响应。

```
"Hello, World! I'm not sure where I'm calling from..."
```

通过阶段变量将特定于阶段的元数据传递到 Lambda 函数中

此过程介绍了如何使用阶段变量将特定于阶段的配置元数据传递到 Lambda 函数中。您创建 `POST` 方法和输入映射模板，以使用先前声明的 `stageName` 阶段变量生成负载。

1. 选择 /lambdav1 资源，然后选择创建方法。

然后执行以下操作：

   a. 对于方法类型，选择 POST。
   b. 对于集成类型，选择 Lambda 函数。
   c. 保持 Lambda 代理集成处于关闭状态。
   d. 对于 Lambda 函数，输入 `${stageVariables.function}`。
   e. 当提示使用添加权限命令时，复制 AWS CLI 命令。对将分配给 `function` 阶段变量的每个 Lambda 函数运行此命令。
   f. 选择创建方法。

2. 选择集成请求选项卡，然后在集成请求设置部分中，选择编辑。
3. 选择映射模板，然后选择添加映射模板。
4. 对于内容类型，输入 **`application/json`**。
5. 对于模板正文，输入以下模板：

```
#set($inputRoot = $input.path('$'))
{
    "stageName" : "$stageVariables.stageName"
}
```

> **Note**
>
> 在映射模板中，阶段变量必须在引号中引用（如 `"$stageVariables.stageName"` 或 `"${stageVariables.stageName}"` 中所示）。在其他位置，必须不带引号引用（如 `${stageVariables.function}` 中所示）。

6. 选择保存。
7. 将 API 部署到 **`beta`** 和 **`prod`** 阶段。
8. 要使用 REST API 客户端传递特定于阶段的元数据，请执行以下操作：

   a. 在阶段导航窗格中，选择 beta 阶段。在阶段详细信息下，选择复制图标以复制 API 的调用 URL，然后在 REST API 客户端的输入字段中输入 API 的调用 URL。在提交请求之前附加 **`/lambdav1`**。

   您将收到以下响应。

```
"Hello, World! I'm calling from the beta stage."
```

b. 在阶段导航窗格中，选择 prod。在阶段详细信息下，选择复制图标以复制 API 的调用 URL，然后在 REST API 客户端的输入字段中输入 API 的调用 URL。在提交请求之前附加 **`/lambdav1`**。

   您将收到以下响应。

   ```
   "Hello, World! I'm calling from the prod stage."
   ```

9. 要使用测试特征传递特定于阶段的元数据，请执行以下操作：

   a. 在资源导航窗格中，选择测试选项卡。您可能需要选择右箭头按钮以显示该选项卡。

   b. 对于 function，请输入 **`HelloWorld`**。

   c. 对于 stageName，请输入 **`beta`**。

   d. 选择测试。您无需将正文添加到您的 `POST` 请求。

      您将收到以下响应。

      ```
      "Hello, World! I'm calling from the beta stage."
      ```

   e. 您可以重复前面的步骤来测试 Prod 阶段。对于 stageName，请输入 **`Prod`**。

      您将收到以下响应。

      ```
      "Hello, World! I'm calling from the prod stage."
      ```

Amazon API Gateway 阶段变量参考

在以下情况下，您可以使用 API Gateway 阶段变量。

参数映射表达式

阶段变量可在参数映射表达式中用于 API 方法的请求或响应标头参数，无需进行任何部分替换。在以下示例中，引用阶段变量时未使用 `$` 和封闭的 `{...}`。

- `stageVariables.<variable_name>`

映射模板

阶段变量可在映射模板中的任何位置使用，如以下示例所示。

- `{ "name" : "$stageVariables.<variable_name>"}`
- `{ "name" : "${stageVariables.<variable_name>}"}`

HTTP 集成 URI

阶段变量可用作 HTTP 集成 URL 的一部分，如以下示例所示：

- 不带协议的完整 URI – `http://${stageVariables.<variable_name>}`
- 完整域 – `http://${stageVariables.<variable_name>}/resource/operation`
- 子域 – `http://${stageVariables.<variable_name>}.example.com/resource/operation`
- 路径 – `http://example.com/${stageVariables.<variable_name>}/bar`
- 查询字符串 – `http://example.com/foo?q=${stageVariables.<variable_name>}`

AWS 集成 URI

阶段变量可用作 AWS URI 操作或路径组件的一部分，如以下示例所示。

- `arn:aws:apigateway:<region>:<service>:${stageVariables.<variable_name>}`

AWS 集成 URI（Lambda 函数）

阶段变量可用于替代 Lambda 函数名称或版本/别名，如以下示例所示。

- `arn:aws:apigateway:<region>:lambda:path/2015-03-31/functions/arn:aws:lambda:<region>:<account_id>:function:${stageVariables.<function_variable_name>}/invocations`
- `arn:aws:apigateway:<region>:lambda:path/2015-03-31/functions/arn:aws:lambda:<region>:<account_id>:function:<function_name>:${stageVariables.<version_variable_name>}/invocations`

> **Note**
>
> 要将阶段变量用于 Lambda 函数，该函数必须与 API 位于同一账户中。阶段变量不支持跨账户 Lambda 函数。

Amazon Cognito 用户池

阶段变量可用来代替 `COGNITO_USER_POOLS` 授权方的 Amazon Cognito 用户池。

- `arn:aws:cognito-idp:<region>:<account_id>:userpool/${stageVariables.<variable_name>}`

AWS 集成凭证

阶段变量可用作 AWS 用户/角色凭证 ARN 的一部分，如以下示例所示。

- `arn:aws:iam::<account_id>:${stageVariables.<variable_name>}`

设置 API Gateway Canary 版本部署

Canary 版本是一种软件开发战略，部署新版本的 API（以及其他软件）用于测试，而在相同阶段基础版本仍部署为生产版本用于正常运营。在本文档中，为了方便讨论，我们将基础版本称为生产版本。虽然这样做是合理的，您也可以随意在任意非生产版本上应用 Canary 版本进行测试。

在 Canary 版本部署中，全部 API 流量按照预配置的比率，随机拆分到生产版本和 Canary 版本中。通常，Canary 版本接收少量百分比的 API 流量，生产版本接受剩余部分。更新后的 API 特征仅对流经 Canary 的 API 流量可见。您可以调整 Canary 流量百分比来优化测试覆盖范围或性能。

通过将 Canary 流量保持在较小的比例并随机进行选择，大部分用户在任何时间都不会受到新版本中潜在错误的不利影响，而且没有任何一个用户会始终受到不利影响。

在测试指标满足您的要求之后，可以将金丝雀版本提升为生产版本，并在部署中禁用 Canary 版本。这使得新特征在生产阶段中可用。

主题

- API Gateway 中的 Canary 版本部署
- 创建 Canary 版本部署

- 更新 Canary 版本
- 提升 Canary 版本
- 关闭金丝雀版本

API Gateway 中的 Canary 版本部署

在 API Gateway 中，Canary 版本部署将部署阶段用于 API 基础版本的生产版本，并将 Canary 版本连接到阶段，用于提供 API 相对于基础版本的新版本。该阶段与初始部署关联，Canary 与后续部署关联。在一开始，阶段和 Canary 均指向相同的 API 版本。在这一部分中，我们所说的阶段和生产版本是可以互换的，Canary 和 Canary 版本也是可以互换的。

要部署带有 Canary 版本的 API，您必须将 Canary 设置添加到常规部署的阶段。Canary 设置描述基本金丝雀版本，阶段表示此部署中 API 的生产版本。要添加 Canary 设置，请在部署阶段上设置 `canarySettings` 并指定以下内容：

- 部署 ID，最初与阶段上设置的基础版本部署的 ID 相同。
- Canary 版本的 API 流量百分比，在 0.0 到 100.0（含）之间。
- Canary 版本的阶段变量，可以覆盖生产版本阶段变量。
- 为 Canary 版本请求使用的阶段缓存（如果设置了 useStageCache 并在阶段上启用了 API 缓存）。

启用 Canary 版本之后，除非禁用 Canary 版本并从阶段中删除 Canary 设置，否则部署阶段无法与其他非 Canary 版本部署关联。

当您启用 API 执行日志记录时，Canary 版本为所有 Canary 请求生成自己的日志和指标。它们报告给生产阶段 CloudWatch Logs 日志组，以及 Canary 版本特定的 CloudWatch Logs 日志组。访问日志记录也是如此。单独的 Canary 特定日志非常有用，可用于验证新 API 更改以及决定是否接受更改并将 Canary 版本提升为生产版本，或者放弃更改并从生产阶段还原 Canary 版本。

生产阶段执行日志组名为 `API-Gateway-Execution-Logs/`*`{rest-api-id}`*`/`*`{stage-name}`*，Canary 版本执行日志组名为 `API-Gateway-Execution-Logs/`*`{rest-api-id}`*`/`*`{stage-name}`*`/Canary`。对于访问日志记录，您必须创建新日志组或者选择现有日志组。Canary 版本访问日志组名称具有 `/Canary` 后缀，附加到选定的日志组名中。

Canary 发布可以使用阶段缓存（如果启用），并在预定义的生存时间 (TTL) 周期内，使用缓存的条目将结果返回到下一个 Canary 发布请求。

在 Canary 版本部署中，API 的生产版本和 Canary 版本可以关联到同一个版本，也可以关联到不同版本。当它们与不同版本关联时，来自生产和 Canary 请求的响应单独缓存，阶段缓存返回生产和

Canary 请求的对应结果。当生产版本和 Canary 版本关联到同一个部署时，阶段缓存为两种类型的请求使用单个缓存键，并为来自生产版本和 Canary 版本的相同请求返回相同响应。

创建 Canary 版本部署

在部署具有 Canary 设置的 API 作为对部署创建操作的额外输入时，您可以创建 Canary 版本部署。

您还可以通过发出 `stage:update` 请求在阶段上添加 Canary 版本设置，从现有非 Canary 版本部署创建 Canary 版本。

创建非 Canary 版本部署时，您可以指定非现有阶段名称。如果指定的阶段不存在，API Gateway 将创建一个。但是，在创建 Canary 版本部署时，您无法指定任何非现有阶段名称。您会收到错误，并且 API Gateway 不创建任何 Canary 版本部署。

您可以使用 API Gateway 控制台、AWS CLI 或AWS开发工具包在 API Gateway 中创建 Canary 版本部署。

主题

- 使用 API Gateway 控制台创建 Canary 版本部署
- 使用 AWS CLI 创建 Canary 版本部署

使用 API Gateway 控制台创建 Canary 版本部署

要使用 API Gateway 控制台创建 Canary 版本部署，请按照以下说明操作：

创建初始 Canary 版本部署

1. 登录 API Gateway 控制台。
2. 选择现有 REST API 或创建新的 REST API。
3. 在主导航窗格中，选择资源，然后选择部署 API。按照 Deploy API (部署 API) 中屏幕上的说明操作，将 API 部署到新阶段。

   到目前为止，您已将 API 部署到生产版本阶段。接下来，您需要在阶段上配置 Canary 设置，并且在需要时还要启用缓存、设置阶段变量或者配置 API 执行或访问日志。
4. 要启用 API 缓存或将 AWS WAF Web ACL 与阶段关联，请在阶段详细信息部分中选择编辑。有关更多信息，请参阅the section called “缓存设置”或the section called “使用 API Gateway 控制台将 AWS WAF Web ACL 与 API Gateway API 阶段相关联”。
5. 要配置执行或访问日志记录，请在日志和跟踪部分中选择编辑，并按照屏幕上的说明操作。有关更多信息，请参阅 在 API Gateway 中为 REST API 设置 CloudWatch 日志记录。

6. 要设置阶段变量，请选择阶段变量选项卡，并按照屏幕上的说明添加或修改阶段变量。有关更多信息，请参阅 the section called “设置阶段变量”。
7. 选择金丝雀选项卡，然后选择创建金丝雀。您可能需要选择右箭头按钮以显示金丝雀选项卡。
8. 在金丝雀设置下，对于金丝雀，输入要转移到金丝雀的请求的百分比。
9. 如果需要，请选择阶段缓存以便为金丝雀版本开启缓存。除非启用 API 缓存，否则缓存对 Canary 版本不可用。
10. 要覆盖现有的阶段变量，对于金丝雀覆盖，请输入新的阶段变量值。

在部署阶段上初始化 Canary 版本之后，您更改了 API 并希望测试更改。您可以将 API 重新部署到相同阶段，这样可以通过同一个阶段访问更新后的版本和基础版本。以下步骤说明了如何完成此操作。

将最新 API 版本部署到 Canary

1. 每次更新 API 时，请选择部署 API。
2. 在部署 API 中，从部署阶段下拉列表中选择包含金丝雀的阶段。
3. （可选）为部署描述输入描述。
4. 选择 Deploy (部署) 将最新 API 版本推送到 Canary 版本。
5. 如果需要，可重新配置阶段设置、日志或 Canary 设置，如创建初始 Canary 版本部署中所述。

此时，Canary 版本指向最新版本，而生产版本仍指向 API 的初始版本。canarySettings 现在具有新的 deploymentId 值，而阶段仍具有最初的 deploymentId 值。在后台，控制台调用 stage:update。

使用 AWS CLI 创建 Canary 版本部署

首先创建具有两个阶段变量但没有任何 Canary 的基准部署：

```
aws apigateway create-deployment \
    --variables sv0=val0,sv1=val1 \
    --rest-api-id abcd1234 \
    --stage-name 'prod' \
```

该命令返回生成的 `Deployment` 的表示，与以下类似：

```
{
    "id": "du4ot1",
    "createdDate": 1511379050
```

```
}
```

生成的部署 id 标识 API 的一个快照 (或版本)。

现在，在 prod 阶段上创建 Canary 部署：

```
aws apigateway create-deployment  --rest-api-id abcd1234 \
    --canary-settings \
    '{
        "percentTraffic":10.5,
        "useStageCache":false,
        "stageVariableOverrides":{
            "sv1":"val2",
            "sv2":"val3"
        }
    }' \
    --stage-name 'prod'
```

如果指定的阶段 (prod) 不存在，则之前的命令返回错误。否则，它会返回新创建的部署资源表示，与以下内容类似：

```
{
    "id": "a6rox0",
    "createdDate": 1511379433
}
```

生成的部署 id 标识 Canary 版本的 API 测试版本。此时关联的阶段启用了 Canary。您可以通过调用 `get-stage` 命令来查看此阶段，与以下类似：

```
aws apigateway get-stage --rest-api-id acbd1234 --stage-name prod
```

下面说明了以命令输出表示的 `Stage`：

```
{
    "stageName": "prod",
    "variables": {
        "sv0": "val0",
        "sv1": "val1"
    },
    "cacheClusterEnabled": false,
```

```
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": "du4ot1",
    "lastUpdatedDate": 1511379433,
    "createdDate": 1511379050,
    "canarySettings": {
        "percentTraffic": 10.5,
        "deploymentId": "a6rox0",
        "useStageCache": false,
        "stageVariableOverrides": {
            "sv2": "val3",
            "sv1": "val2"
        }
    },
    "methodSettings": {}
}
```

在本例中，API 的基础版本将使用阶段变量 `{"sv0":val0", "sv1":val1"}`，而测试版本使用阶段变量 `{"sv1":val2", "sv2":val3"}`。生产版本和 Canary 版本均使用相同的阶段变量 `sv1`，但分别具有不同值 `val1` 和 `val2`。阶段变量 `sv0` 仅用于生产版本中，阶段变量 `sv2` 仅用于 Canary 版本中。

您可通过更新阶段来启用 Canary，从现有常规部署创建 Canary 版本部署。为了演示此操作，首先创建一个常规部署：

```
aws apigateway create-deployment \
    --variables sv0=val0,sv1=val1 \
    --rest-api-id abcd1234 \
    --stage-name 'beta'
```

该命令返回基础版本部署的表示：

```
{
    "id": "cifeiw",
    "createdDate": 1511380879
}
```

关联的测试版阶段没有任何 Canary 设置：

```
{
    "stageName": "beta",
    "variables": {
```

```
        "sv0": "val0",
        "sv1": "val1"
    },
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": "cifeiw",
    "lastUpdatedDate": 1511380879,
    "createdDate": 1511380879,
    "methodSettings": {}
}
```

现在，通过在阶段上连接 Canary 创建新的 Canary 版本部署：

```
aws apigateway update-stage \
    --rest-api-id abcd1234 \
    --stage-name 'beta' \
    --patch-operations '[{
            "op": "replace",
            "value": "0.0",
            "path": "/canarySettings/percentTraffic"
        }, {
            "op": "copy",
            "from": "/canarySettings/stageVariableOverrides",
            "path": "/variables"
        }, {
            "op": "copy",
            "from": "/canarySettings/deploymentId",
            "path": "/deploymentId"
        }]'
```

更新后阶段的表示如下所示：

```
{
    "stageName": "beta",
    "variables": {
        "sv0": "val0",
        "sv1": "val1"
    },
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": "cifeiw",
    "lastUpdatedDate": 1511381930,
    "createdDate": 1511380879,
```

```
    "canarySettings": {
        "percentTraffic": 10.5,
        "deploymentId": "cifeiw",
        "useStageCache": false,
        "stageVariableOverrides": {
            "sv2": "val3",
            "sv1": "val2"
        }
    },
    "methodSettings": {}
}
```

由于我们刚刚在 API 的现有版本上启用了 Canary，生产版本 (`Stage`) 和 Canary 版本 (`canarySettings`) 指向相同的部署，即 API 的相同版本 (`deploymentId`)。在您更改 API 并将其再次部署到此阶段之后，新版本将为 Canary 版本，而基础版本保持为生产版本。在阶段演变中，Canary 的 `deploymentId` 更新为新部署 `id`，而生产版本的 `deploymentId` 保持不变，证明了这一点。

更新 Canary 版本

在部署了 Canary 版本之后，您可能需要调整 Canary 流量的百分比，或者启用或禁用阶段缓存以优化测试性能。在更新执行上下文时，您还可以修改在 Canary 版本中使用的阶段变量。要进行此类更新，请在 canarySettings 上使用新值调用 stage:update 操作。

您可以使用 API Gateway 控制台、AWS CLI update-stage 命令或AWS开发工具包更新 Canary 版本。

主题

- 使用 API Gateway 控制台更新 Canary 版本
- 使用 AWS CLI 更新 Canary 版本

使用 API Gateway 控制台更新 Canary 版本

要使用 API Gateway 控制台更新阶段上的现有 Canary 设置，请执行以下操作：

更新现有的金丝雀设置

1. 登录 API Gateway 控制台并选择现有的 REST API。
2. 在主导航窗格中，选择阶段，然后选择一个现有阶段。
3. 选择金丝雀选项卡，然后选择编辑。您可能需要选择右箭头按钮以显示金丝雀选项卡。
4. 通过在 0.0 到 100.0 之间（含）增加或减少百分比数字，更新请求分布。

5. 选中或清除阶段缓存复选框。
6. 添加、移除或修改金丝雀阶段变量。
7. 选择保存。

使用 AWS CLI 更新 Canary 版本

要使用 AWS CLI 更新 Canary，请调用 `update-stage` 命令。

要为 Canary 启用或禁用阶段缓存，请按以下所示调用 `update-stage` 命令：

```
aws apigateway update-stage \
    --rest-api-id {rest-api-id} \
    --stage-name '{stage-name}'  \
    --patch-operations op=replace,path=/canarySettings/useStageCache,value=true
```

要调整 Canary 流量百分比，请调用 `update-stage` 以替换阶段上的 `/canarySettings/percentTraffic` 值。

```
aws apigateway update-stage \
    --rest-api-id {rest-api-id} \
    --stage-name '{stage-name}' \
    --patch-operations op=replace,path=/canarySettings/percentTraffic,value=25.0
```

更新 Canary 阶段变量，包括添加、替换或删除 Canary 阶段变量：

```
aws apigateway update-stage  \
    --rest-api-id {rest-api-id} \
    --stage-name '{stage-name}'  \
    --patch-operations '[{
        "op": "replace",
        "path": "/canarySettings/stageVariableOverrides/newVar",
        "value": "newVal"
      }, {
        "op": "replace",
        "path": "/canarySettings/stageVariableOverrides/var2",
        "value": "val4"
      }, {
        "op": "remove",
        "path": "/canarySettings/stageVariableOverrides/var1"
      }]'
```

您可以通过将操作组合到单个 `patch-operations` 值中，更新以上所有内容。

```
aws apigateway update-stage  \
    --rest-api-id {rest-api-id} \
    --stage-name '{stage-name}' \
    --patch-operations '[{
        "op": "replace",
        "path": "/canarySettings/percentTraffic",
        "value": "20.0"
    }, {
        "op": "replace",
        "path": "/canarySettings/useStageCache",
        "value": "true"
    }, {
        "op": "remove",
        "path": "/canarySettings/stageVariableOverrides/var1"
    }, {
        "op": "replace",
        "path": "/canarySettings/stageVariableOverrides/newVar",
        "value": "newVal"
    }, {
        "op": "replace",
        "path": "/canarySettings/stageVariableOverrides/val2",
        "value": "val4"
      }]'
```

提升 Canary 版本

提升 Canary 版本，使其在正测试的 API 版本的生产阶段中可用。操作涉及到以下任务：

- 使用 Canary 的部署 ID 设置重置阶段的部署 ID。这会使用 Canary 的快照更新阶段的 API 快照，使得测试版本也成为生产版本。
- 使用 Canary 阶段变量更新阶段变量（如有）。这会使用 Canary 的内容更新阶段的 API 执行上下文。没有此更新时，如果测试版本使用不同的阶段变量或者现有阶段变量的不同值，新 API 版本可能会造成意外的结果。
- 将 Canary 流量的百分比设置为 0.0%。

提升 Canary 版本不会在阶段上禁用 Canary。要禁用 Canary，您必须在阶段上删除 Canary 设置。

主题

- 使用 API Gateway 控制台提升 Canary 版本
- 使用 AWS CLI 提升 Canary 版本

使用 API Gateway 控制台提升 Canary 版本

要使用 API Gateway 控制台提升 Canary 版本部署，请执行以下步骤：

提升金丝雀版本部署

1. 登录 API Gateway 控制台并在主导航窗格中选择现有 API。
2. 在主导航窗格中，选择阶段，然后选择一个现有阶段。
3. 选择金丝雀选项卡。
4. 选择提升金丝雀。
5. 确认所做的更改，然后选择提升金丝雀。

提升之后，生产版本引用相同的 API 版本 (deploymentId) 作为 Canary 版本。您可以使用 AWS CLI 验证此项。有关示例，请查看the section called “使用 AWS CLI 提升 Canary 版本”。

使用 AWS CLI 提升 Canary 版本

要使用 AWS CLI 命令将 Canary 版本提升到生产版本，请调用 `update-stage` 命令以将与 Canary 版本关联的 `deploymentId` 复制到与阶段关联的 `deploymentId`，将 Canary 版本流量百分比重置为零 (`0.0`)，并且将任意 Canary 版本绑定阶段变量复制到对应的阶段绑定变量。

假设我们有一个 Canary 版本部署，由类似于以下内容的阶段描述：

```
{
    "_links": {
        ...
    },
    "accessLogSettings": {
        ...
    },
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "canarySettings": {
        "deploymentId": "eh1sby",
        "useStageCache": false,
        "stageVariableOverrides": {
            "sv2": "val3",
```

```
            "sv1": "val2"
        },
        "percentTraffic": 10.5
    },
    "createdDate": "2017-11-20T04:42:19Z",
    "deploymentId": "nfcn0x",
    "lastUpdatedDate": "2017-11-22T00:54:28Z",
    "methodSettings": {
        ...
    },
    "stageName": "prod",
    "variables": {
        "sv1": "val1"
    }
}
```

我们调用以下 `update-stage` 请求来提升它：

```
aws apigateway update-stage  \
    --rest-api-id {rest-api-id}  \
    --stage-name '{stage-name}'  \
    --patch-operations '[{
        "op": "replace",
        "value": "0.0",
        "path": "/canarySettings/percentTraffic"
      }, {
        "op": "copy",
        "from": "/canarySettings/stageVariableOverrides",
        "path": "/variables"
      }, {
        "op": "copy",
        "from": "/canarySettings/deploymentId",
        "path": "/deploymentId"
      }]'
```

提升后，阶段现在看起来与以下内容类似：

```
{
    "_links": {
        ...
    },
    "accessLogSettings": {
        ...
```

```
    },
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "canarySettings": {
        "deploymentId": "eh1sby",
        "useStageCache": false,
        "stageVariableOverrides": {
            "sv2": "val3",
            "sv1": "val2"
        },
        "percentTraffic": 0
    },
    "createdDate": "2017-11-20T04:42:19Z",
    "deploymentId": "eh1sby",
    "lastUpdatedDate": "2017-11-22T05:29:47Z",
    "methodSettings": {
        ...
    },
    "stageName": "prod",
    "variables": {
        "sv2": "val3",
        "sv1": "val2"
    }
}
```

正如您所看到的，将 Canary 版本提升到阶段并不会禁用该金丝雀版本，部署仍然是 Canary 版本部署。要使其成为常规生产版本部署，您必须禁用 Canary 设置。有关如何禁用 Canary 版本部署的更多信息，请参阅 the section called “关闭金丝雀版本”。

关闭金丝雀版本

要关闭金丝雀版本部署，请将 `canarySettings` 设置为 Null 以从阶段中删除它。

您可以使用 API Gateway 控制台、AWS CLI 或AWS开发工具包禁用 Canary 版本部署。

主题

- 使用 API Gateway 控制台关闭金丝雀版本
- 使用 AWS CLI 关闭金丝雀版本

使用 API Gateway 控制台关闭金丝雀版本

要使用 API Gateway 控制台关闭金丝雀版本部署，请执行以下步骤：

关闭金丝雀版本部署

1. 登录 API Gateway 控制台并在主导航窗格中选择现有 API。
2. 在主导航窗格中，选择阶段，然后选择一个现有阶段。
3. 选择金丝雀选项卡。
4. 选择 删除。
5. 选择 Delete (删除) 确认您要删除 Canary。

结果，`canarySettings` 属性变为 `null`，并从部署阶段中删除。您可以使用 AWS CLI 验证此项。有关示例，请查看the section called “使用 AWS CLI 关闭金丝雀版本”。

使用 AWS CLI 关闭金丝雀版本

要使用 AWS CLI 关闭金丝雀版本部署，请按以下所示调用 `update-stage` 命令：

```
aws apigateway update-stage \
    --rest-api-id abcd1234 \
    --stage-name canary \
    --patch-operations '[{"op":"remove", "path":"/canarySettings"}]'
```

成功的响应返回与以下类似的负载：

```
{
    "stageName": "prod",
    "accessLogSettings": {
        ...
    },
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": "nfcn0x",
    "lastUpdatedDate": 1511309280,
    "createdDate": 1511152939,
    "methodSettings": {
        ...
    }
}
```

如输出中所示，`canarySettings` 属性不再存在于已禁用 Canary 的部署的阶段中。

需要重新部署的对 REST API 的更新

维护 API 相当于查看、更新和删除现有的 API 设置。您可以使用 API Gateway 控制台、AWS CLI、开发工具包或 API Gateway REST API 来维护 API。更新 API 涉及到修改 API 的特定资源属性或配置设置。资源更新需要重新部署 API，而配置更新不需要。

下表中详细介绍了可以更新的 API 资源。

需要重新部署 API 的 API 资源更新

资源	备注
ApiKey	有关适用的属性和支持的操作，请参阅 apikey:update。此更新需要重新部署 API。
授权方	有关适用的属性和支持的操作，请参阅 authorizer:update。此更新需要重新部署 API。
DocumentationPart	有关适用的属性和支持的操作，请参阅 documentationpart:update。此更新需要重新部署 API。
DocumentationVersion	有关适用的属性和支持的操作，请参阅 documentationversion:update。此更新需要重新部署 API。
GatewayResponse	有关适用的属性和支持的操作，请参阅 gatewayresponse:update。此更新需要重新部署 API。
集成	有关适用的属性和支持的操作，请参阅 integration:update。此更新需要重新部署 API。
IntegrationResponse	有关适用的属性和支持的操作，请参阅 integrationresponse:update。此更新需要重新部署 API。
方法	有关适用的属性和支持的操作，请参阅 method:update。此更新需要重新部署 API。
MethodResponse	有关适用的属性和支持的操作，请参阅 methodresponse:update。此更新需要重新部署 API。
模型	有关适用的属性和支持的操作，请参阅 model:update。此更新需要重新部署 API。

资源	备注
RequestValidator	有关适用的属性和支持的操作，请参阅 requestvalidator:update。此更新需要重新部署 API。
资源	有关适用的属性和支持的操作，请参阅 resource:update。此更新需要重新部署 API。
RestApi	有关适用的属性和支持的操作，请参阅 restapi:update。此更新需要重新部署 API。
VpcLink	有关适用的属性和支持的操作，请参阅 vpclink:update。此更新需要重新部署 API。

下表中详细介绍了可以更新的 API 配置。

不需要重新部署 API 的 API 配置更新

配置	备注
账户	有关适用的属性和支持的操作，请参阅 account:update。此更新不需要重新部署 API。
部署	有关适用的属性和支持的操作，请参阅 deployment:update。
DomainName	有关适用的属性和支持的操作，请参阅 domainname:update。此更新不需要重新部署 API。
BasePathMapping	有关适用的属性和支持的操作，请参阅 basepathmapping:update。此更新不需要重新部署 API。
阶段	有关适用的属性和支持的操作，请参阅 stage:update。此更新不需要重新部署 API。
用法	有关适用的属性和支持的操作，请参阅 usage:update。此更新不需要重新部署 API。
UsagePlan	有关适用的属性和支持的操作，请参阅 usageplan:update。此更新不需要重新部署 API。

为 REST API 设置自定义域名

自定义域名 是您可以提供给 API 用户的更简单、更直观的 URL。

部署 API 后，您（和您的客户）可以使用以下格式的默认基本 URL 调用 API：

```
https://api-id.execute-api.region.amazonaws.com/stage
```

其中 `api-id` 由 API Gateway 生成，`region`（AWS 区域）由您在创建 API 时指定，`stage` 由您在部署 API 时指定。

URL 的主机名部分（即 `api-id.execute-api.region.amazonaws.com`）是指 API 端点。默认 API 端点难于重新调用，对用户不友好。

使用自定义域名，您可以设置 API 的主机名，并选择基本路径（例如 `myservice`）以将备用 URL 映射到 API。例如，一个更为用户友好的 API 基本 URL 可以变成：

```
https://api.example.com/myservice
```

Note

区域自定义域可以与 REST API 和 HTTP API 相关联。您可以使用 API Gateway 版本 2 API 创建和管理 REST API 的区域自定义域名。
私有 API不支持自定义域名。
您可以选择 REST API 支持的最低 TLS 版本。对于 REST API，您可以选择 TLS 1.2 或 TLS 1.0。

注册域名

您必须拥有已注册的 Internet 域名，以便为 API 设置自定义域名。域名必须遵循 RFC 1035 规范，每个标签最多可以有 63 个八位字节，总共可以有 255 个八位字节。如果需要，您可以使用 Amazon Route 53 或使用您选择的第三方域注册商注册互联网域。API 的自定义域名可以是已注册 Internet 域的子域或根域（也称为“顶级域”）的名称。

在 API Gateway 中创建自定义域名后，您必须创建或更新 DNS 提供商的资源记录以映射到 API 端点。如果没有此类映射，针对自定义域名的 API 请求无法到达 API Gateway。

ⓘ Note

自定义域名跨所有 AWS 账户在一个区域内必须是唯一的。

要在区域或 AWS 账户之间移动边缘优化的自定义域名，必须删除现有 CloudFront 分配和创建新的分配。该过程可能需要大约 30 分钟，然后新的自定义域名才变为可用。有关更多信息，请参阅更新 CloudFront 分配。

边缘优化的自定义域名

当您部署边缘优化的 API 时，API Gateway 会设置 Amazon CloudFront 分配和 DNS 记录，以将 API 域名映射到 CloudFront 分配域名。然后，对 API 的请求将通过映射的 CloudFront 分配路由到 API Gateway。

当您为边缘优化的 API 创建自定义域名时，API Gateway 会设置 CloudFront 分配。但是，您必须设置 DNS 记录以将自定义域名映射到 CloudFront 分配域名。此映射适用于针对要通过映射的 CloudFront 分配路由到 API Gateway 的自定义域名绑定的 API 请求。您还必须为自定义域名提供证书。

ⓘ Note

API Gateway 创建的 CloudFront 分配由与 API Gateway 关联的区域特定账户拥有。当跟踪操作以在 CloudWatch Logs 中创建和更新此类 CloudFront 分配时，您必须使用此 API Gateway 账户 ID。有关更多信息，请参阅 在 CloudTrail 中记录自定义域名的创建操作。

要设置边缘优化的自定义域名或更新其证书，您必须有权更新 CloudFront 分配。

要提供访问权限，请为您的用户、组或角色添加权限：

- AWS IAM Identity Center 中的用户和群组：

  创建权限集合。按照《AWS IAM Identity Center 用户指南》中创建权限集的说明进行操作。
- 通过身份提供商在 IAM 中托管的用户：

  创建适用于身份联合验证的角色。按照《IAM 用户指南》中为第三方身份提供商创建角色（联合身份验证）的说明进行操作。
- IAM 用户：
  - 创建您的用户可以担任的角色。按照《IAM 用户指南》中为 IAM 用户创建角色的说明进行操作。

- （不推荐使用）将策略直接附加到用户或将用户添加到用户组。按照《IAM 用户指南》中向用户添加权限（控制台）中的说明进行操作。

更新 CloudFront 分配需要以下权限。

```
{
    "Version": "2012-10-17",
    "Statement": [
         {
            "Sid": "AllowCloudFrontUpdateDistribution",
            "Effect": "Allow",
            "Action": [
                "cloudfront:updateDistribution"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

API Gateway 在 CloudFront 分配上通过利用服务器名称指示 (SNI) 来支持边缘优化的自定义域名。有关在 CloudFront 分配上使用自定义域名的更多信息，其中包括所需证书格式和证书密钥最大长度，请参阅 Amazon CloudFront 开发人员指南 中的使用备用域名和 HTTPS。

要设置自定义域名作为 API 的主机名，API 拥有者必须针对自定义域名提供 SSL/TLS 证书。

要为边缘优化的自定义域名提供此证书，您可以请求 AWS Certificate Manager (ACM) 在 ACM 中生成新证书，或向 ACM 导入由 `us-east-1` 区域（美国东部（弗吉尼亚北部））中的第三方证书颁发机构颁发的证书。

区域自定义域名

为区域 API 创建自定义域名时，API Gateway 为 API 创建区域域名。您必须设置将自定义域名映射到区域域名的 DNS 记录。您还必须为自定义域名提供证书。

通配符自定义域名

使用通配符自定义域名，您可以在不超过默认配额的情况下支持几乎无限数量的域名。例如，您可以为每位客户提供自己的域名 `*customername*.api.example.com`。

要创建通配符自定义域名，可以指定通配符 (*) 作为表示根域所有可能子域的自定义域的第一个子域。

例如，通配符自定义域名 `*.example.com` 会生成子域，如 `a.example.com`、`b.example.com` 和 `c.example.com`，这些子域都会路由到同一个域。

通配符自定义域名支持与 API Gateway 的标准自定义域名不同的配置。例如，在单个 AWS 账户中，您可以对 `*.example.com` 和 `a.example.com` 进行不同的配置。

您可以使用 `$context.domainName` 和 `$context.domainPrefix` 上下文变量来确定客户端用于调用 API 的域名。要了解有关上下文变量的更多信息，请参阅 API Gateway 映射模板和访问日志记录变量引用。

要创建通配符自定义域名，您必须提供已使用 DNS 或电子邮件验证方法验证的由 ACM 颁发的证书。

> **Note**
>
> 如果其他 AWS 账户已经创建了与通配符自定义域名冲突的自定义域名，则无法创建通配符自定义域名。例如，如果账户 A 已经创建了 `a.example.com`，则账户 B 无法创建通配符自定义域名 `*.example.com`。
> 如果账户 A 和账户 B 共享拥有者，您可以联系 AWS Support 中心请求例外。

自定义域名的证书

> **Important**
>
> 您为您的自定义域名配置了证书。如果您的应用程序使用证书固定（有时称为 SSL 固定）来固定 ACM 证书，则在 AWS 续订证书后，应用程序可能无法连接到您的域。有关更多信息，请参阅《AWS Certificate Manager 用户指南》中的证书固定问题。

要为支持 ACM 的区域中的自定义域名提供证书，您必须从 ACM 请求证书。要为不支持 ACM 的区域中的区域自定义域名提供证书，您必须在该区域中将证书导入到 API Gateway。

要导入 SSL/TLS 证书，您必须针对自定义域名提供 PEM 格式的 SSL/TLS 证书文本、其私有密钥和证书链。存储在 ACM 中的每个证书均由其 ARN 标识。要针对域名使用 AWS 托管的证书，您只需参考其 ARN 即可。

通过 ACM 可以轻松地为 API 设置和使用自定义域名。您可以为给定的域名创建证书（或导入证书），使用 ACM 提供的证书的 ARN 在 API Gateway 中设置域名，然后将自定义域名下的基本路径映射到

API 的已部署阶段。如果拥有 ACM 颁发的证书，那么您就无需担心公开任何敏感的证书详细信息，如私有密钥。

主题

在 中准备好证书AWS Certificate Manager

为 API 设置自定义域名之前，您必须先在 中准备好 SSL/TLS 证书AWS Certificate Manager 以下步骤介绍了如何完成上述操作。有关更多信息，请参阅 AWS Certificate Manager 用户指南。

> **Note**
>
> 要将 ACM 证书与 API Gateway 边缘优化的自定义域名结合使用，您必须在美国东部（弗吉尼亚北部）(`us-east-1`) 区域中请求或导入证书。对于 API Gateway 区域自定义域名，您必须在与 API 相同的区域中请求或导入证书。证书必须由公开信任的证书颁发机构签名并涵盖自定义域名。

首先，注册您的互联网域，例如 `*example*.com`。您可以使用 Amazon Route 53 或获得认可的第三方域注册商。要获取此类注册商的列表，请参阅 ICANN 网站上获得认可的注册商目录。

要为域名创建 SSL/TLS 证书或将证书导入 ACM 中，请执行以下操作之一：

为域名请求 ACM 提供的证书

1. 登录到 AWS Certificate Manager 控制台。
2. 选择请求证书。
3. 在域名中为您的 API 输入自定义域名，例如 `api.example.com`。

4. （可选）选择向此证书添加一个名称。
5. 选择审核并请求。
6. 选择确认并请求。
7. 如果请求有效，Internet 域中注册的拥有者必须同意请求，然后 ACM 才能颁发证书。

将域名的证书导入 ACM 中

1. 从证书颁发机构为自定义域名获取 PEM 编码的 SSL/TLS 证书。要获取此类 CA 的部分列表，请参阅 Mozilla 内置 CA 列表
   a. 使用 OpenSSL 网站中的 OpenSSL 工具包生成证书的私有密钥并将输出结果保存到文件中：

      ```
      openssl genrsa -out private-key-file 2048
      ```

      > **Note**
      >
      > Amazon API Gateway 利用 Amazon CloudFront 支持自定义域名的证书。因此，自定义域名 SSL/TLS 证书的要求和限制由 CloudFront 决定。例如，公有密钥的最大大小为 2048，而私有密钥大小可以是 1024、2048 和 4096。公有密钥的大小取决于所用的证书颁发机构。要求证书颁发机构返回大小不同于默认长度的密钥。有关更多信息，请参阅安全访问对象和创建签名 URL 和签名 Cookie。

   b. 使用 OpenSSL 通过之前生成的私有密钥生成证书签名请求 (CSR)：

      ```
      openssl req -new -sha256 -key private-key-file -out CSR-file
      ```

   c. 将 CSR 提交给证书颁发机构并保存生成的证书。
   d. 从证书颁发机构下载证书链。

   > **Note**
   >
   > 如果您通过其他方式获取了私有密钥并且密钥已加密，则您可以使用以下命令解密密钥，然后再将其提交到 API Gateway 以便设置自定义域名。
   >
   > ```
   > openssl pkcs8 -topk8 -inform pem -in MyEncryptedKey.pem -outform pem -
   > nocrypt -out MyDecryptedKey.pem
   > ```

2. 将证书上传到 AWS Certificate Manager：

   a. 登录到 AWS Certificate Manager 控制台。

   b. 选择导入证书。

   c. 对于证书正文，输入或粘贴从证书颁发机构提供的 PEM 格式的服务器证书的正文。下面显示了此类证书的简短示例。

      ```
      -----BEGIN CERTIFICATE-----
      EXAMPLECA+KgAwIBAgIQJ1XxJ8Pl++gOfQtj0IBoqDANBgkqhkiG9w0BAQUFADBB
      ...
      az8Cg1aicxLBQ7EaWIhhgEXAMPLE
      -----END CERTIFICATE-----
      ```

   d. 对于证书私有密钥，输入或粘贴 PEM 格式的证书的私有密钥。下面显示了此类密钥的简短示例。

      ```
      -----BEGIN RSA PRIVATE KEY-----
      EXAMPLEBAAKCAQEA2Qb3LDHD7StY7Wj6U2/opV6Xu37qUCCkeDWhwpZMYJ9/nETO
      ...
      1qGvJ3u04vdnzaYN5WoyN5LFckrlA71+CszD1CGSqbVDWEXAMPLE
      -----END RSA PRIVATE KEY-----
      ```

   e. 对于证书链，输入或粘贴 PEM 格式的中间证书和根证书（可选），一个接一个，不带任何空白行。如果包含了根证书，您的证书链必须以中间证书开始，以根证书结尾。使用证书颁发机构提供的中间证书。不要包含未在信任路径链中的任何中间证书。下面显示了一个简短示例。

      ```
      -----BEGIN CERTIFICATE-----
      EXAMPLECA4ugAwIBAgIQWrYdrB5NogYUx1U9Pamy3DANBgkqhkiG9w0BAQUFADCB
      ...
      8/ifBlIK3se2e4/hEfcEejX/arxbx1BJCHBvlEPNnsdw8EXAMPLE
      -----END CERTIFICATE-----
      ```

      以下是另一个示例。

      ```
      -----BEGIN CERTIFICATE-----
      Intermediate certificate 2
      -----END CERTIFICATE-----
      -----BEGIN CERTIFICATE-----
      Intermediate certificate 1
      -----END CERTIFICATE-----
      -----BEGIN CERTIFICATE-----
      ```

```
Optional: Root certificate
-----END CERTIFICATE-----
```

f. 选择审核并导入。

成功创建或导入证书后，请记下证书的 ARN。您在设置自定义域名时会需要它。

在 API Gateway 中为自定义域选择安全策略

为了提高您的 Amazon API Gateway 自定义域的安全性，您可以在 API Gateway 控制台、AWS CLI 或 AWS SDK 中选择安全策略。

安全策略 是 API Gateway 提供的最低 TLS 版本和密码套件的预定义组合。您可以选择 TLS 版本 1.2 或 TLS 版本 1.0 安全策略。TLS 协议解决了网络安全问题，例如客户端和服务器之间的篡改和窃听。当您的客户端通过自定义域建立与您 API 的 TLS 握手时，安全策略实施客户端可以选择使用的 TLS 版本和密码套件选项。

在自定义域设置中，安全策略确定两个设置：

- API Gateway 用来与 API 客户端通信的最低 TLS 版本
- API Gateway 用来加密其返回到 API 客户端的内容的密码

如果您选择 TLS 1.0 安全策略，则该安全策略接受 TLS 1.0、TLS 1.2 和 TLS 1.3 流量。如果您选择 TLS 1.2 安全策略，则该安全策略接受 TLS 1.2 和 TLS 1.3 流量，但拒绝 TLS 1.0 流量。

> Note
>
> 您只能为自定义域指定安全策略。对于使用默认端点的 API，API Gateway 使用以下安全策略：
>
> - 对于边缘优化的 API：`TLS-1-0`
> - 对于区域 API：`TLS-1-0`
> - 对于私有 API：`TLS-1-2`

主题

- 如何为自定义域指定安全策略
- 边缘优化的自定义域支持的安全策略、TLS 协议版本和密码

- 区域自定义域支持的安全策略、TLS 协议版本和密码
- 私有 API 支持的 TLS 协议版本和密码
- OpenSSL 和 RFC 密码名称
- 有关 HTTP API 和 WebSocket API 的信息

如何为自定义域指定安全策略

当您创建自定义域名时，您为其指定安全策略。要了解如何创建自定义域，请参阅the section called "创建边缘优化的自定义域名"或the section called "设置区域自定义域名"。

要更改自定义域名的安全策略，请更新自定义域设置。您可以使用 AWS Management Console、AWS CLI 或 AWS SDK 更新您的自定义域名设置。

当您使用 API Gateway REST API 或 AWS CLI 时，请在 `securityPolicy` 参数中指定新的 TLS 版本 `TLS_1_0` 或 `TLS_1_2`。有关更多信息，请参阅《Amazon API Gateway REST API 参考》中的 domainname:update 或《AWS CLI 参考》中的 update-domain-name。

更新操作可能需要几分钟才能完成。

边缘优化的自定义域支持的安全策略、TLS 协议版本和密码

下表介绍了可为边缘优化的自定义域名指定的安全策略。

安全策略	TLS_1_0	TLS_1_2
TLS 协议		
TLSv1.3	◆	◆
TLSv1.2	◆	◆
TLSv1.1	◆	
TLSv1	◆	
TLS 密码		
TLS_AES_128_GCM_SHA256	◆	◆
TLS_AES_256_GCM_SHA384	◆	◆

安全策略	TLS_1_0	TLS_1_2
TLS_CHACHA20_POLY1305_SHA256	◆	◆
ECDHE-ECDSA-AES128-GCM-SHA256	◆	◆
ECDHE-ECDSA-AES128-SHA256	◆	◆
ECDHE-ECDSA-AES128-SHA	◆	
ECDHE-ECDSA-AES256-GCM-SHA384	◆	◆
ECDHE-ECDSA-CHACHA20-POLY1305	◆	◆
ECDHE-ECDSA-AES256-SHA384	◆	◆
ECDHE-ECDSA-AES256-SHA	◆	
ECDHE-RSA-AES128-GCM-SHA256	◆	◆
ECDHE-RSA-AES128-SHA256	◆	◆
ECDHE-RSA-AES128-SHA	◆	
ECDHE-RSA-AES256-GCM-SHA384	◆	◆
ECDHE-RSA-CHACHA20-POLY1305	◆	◆
ECDHE-RSA-AES256-SHA384	◆	◆

安全策略	TLS_1_0	TLS_1_2
ECDHE-RSA-AES256-SHA	◆	
AES128-GCM-SHA256	◆	
AES256-GCM-SHA384	◆	◆
AES128-SHA256	◆	◆
AES256-SHA	◆	
AES128-SHA	◆	
DES-CBC3-SHA	◆	

区域自定义域支持的安全策略、TLS 协议版本和密码

下表介绍了可为区域自定义域名指定的安全策略。

安全策略	TLS_1_0	TLS_1_2
TLS 协议		
TLSv1.3	◆	◆
TLSv1.2	◆	◆
TLSv1.1	◆	
TLSv1	◆	
TLS 密码		
TLS_AES_128_GCM_SHA256	◆	◆
TLS_AES_256_GCM_SHA384	◆	◆
TLS_CHACHA20_POLY1305_SHA256	◆	◆

安全策略	TLS_1_0	TLS_1_2
ECDHE-ECDSA-AES128-GCM-SHA256	♦	♦
ECDHE-RSA-AES128- GCM-SHA256	♦	♦
ECDHE-ECDSA-AES128-SHA256	♦	♦
ECDHE-RSA-AES128-SHA256	♦	♦
ECDHE-ECDSA-AES128-SHA	♦	
ECDHE-RSA-AES128-SHA	♦	
ECDHE-ECDSA-AES256-GCM-SHA384	♦	♦
ECDHE-RSA-AES256- GCM-SHA384	♦	♦
ECDHE-ECDSA-AES256-SHA384	♦	♦
ECDHE-RSA-AES256-SHA384	♦	♦
ECDHE-RSA-AES256-SHA	♦	
ECDHE-ECDSA-AES256-SHA	♦	
AES128-GCM-SHA256	♦	♦
AES128-SHA256	♦	♦
AES128-SHA	♦	
AES256-GCM-SHA384	♦	♦

安全策略	TLS_1_0	TLS_1_2
AES256-SHA256	♦	♦
AES256-SHA	♦	

私有 API 支持的 TLS 协议版本和密码

下表介绍了私有 API 支持的 TLS 协议和密码。不支持为私有 API 指定安全策略。

安全策略	TLS_1_2
TLS 协议	
TLSv1.2	♦
TLS 密码	
ECDHE-ECDSA-AES128- GCM-SHA256	♦
ECDHE-RSA-AES128- GCM-SHA256	♦
ECDHE-ECDSA-AES128-SHA256	♦
ECDHE-RSA-AES128-SHA256	♦
ECDHE-ECDSA-AES256-GCM-SHA384	♦
ECDHE-RSA-AES256-GCM-SHA384	♦
ECDHE-ECDSA-AES256-SHA384	♦
ECDHE-RSA-AES256-SHA384	♦
AES128-GCM-SHA256	♦
AES128-SHA256	♦
AES256-GCM-SHA384	♦
AES256-SHA256	♦

OpenSSL 和 RFC 密码名称

OpenSSL 和 IETF RFC 5246 为相同的密码使用不同的名称。下表为每个密码列出了 OpenSSL 名称及对应的 RFC 名称。

OpenSSL 密码名称	RFC 密码名称
TLS_AES_128_GCM_SHA256	TLS_AES_128_GCM_SHA256
TLS_AES_256_GCM_SHA384	TLS_AES_256_GCM_SHA384
TLS_CHACHA20_POLY1305_SHA256	TLS_CHACHA20_POLY1305_SHA256
ECDHE-RSA-AES128- GCM-SHA256	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
ECDHE-RSA-AES128-SHA256	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
ECDHE-RSA-AES128-SHA	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
ECDHE-RSA-AES256- GCM-SHA384	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
ECDHE-RSA-AES256-SHA384	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
ECDHE-RSA-AES256-SHA	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA

OpenSSL 密码名称	RFC 密码名称
AES128-GCM-SHA256	TLS_RSA_WITH_AES_128_GCM_SHA256
AES256-GCM-SHA384	TLS_RSA_WITH_AES_256_GCM_SHA384
AES128-SHA256	TLS_RSA_WITH_AES_128_CBC_SHA256
AES256-SHA	TLS_RSA_WITH_AES_256_CBC_SHA
AES128-SHA	TLS_RSA_WITH_AES_128_CBC_SHA
DES-CBC3-SHA	TLS_RSA_WITH_3DES_EDE_CBC_SHA

有关 HTTP API 和 WebSocket API 的信息

有关 HTTP API 和 WebSocket API 的更多信息，请参阅the section called “HTTP API 的安全策略”和the section called “WebSocket API 的安全策略”。

创建边缘优化的自定义域名

主题

- 为 API Gateway API 设置边缘优化自定义域名
- 在 CloudTrail 中记录自定义域名的创建操作
- 配置 API 的基本路径映射，将自定义域名作为主机名
- 轮换 ACM 中导入的证书
- 调用具有自定义域名的 API

为 API Gateway API 设置边缘优化自定义域名

以下过程介绍了如何使用 API Gateway 控制台为 API 创建自定义域名。

使用 API Gateway 控制台创建自定义域名

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 从主导航窗格中选择自定义域名。
3. 选择创建。
4. 对于域名，输入一个域名。
5. 在“配置”下，选择边缘优化。
6. 选择最低 TLS 版本。
7. 选择 ACM 证书。

   > **Note**
   >
   > 要将 ACM 证书与 API Gateway 边缘优化的自定义域名结合使用，您必须在 `us-east-1` 区域（美国东部（弗吉尼亚北部））中请求或导入证书。

8. 选择创建域名。
9. 创建自定义域名后，控制台将显示关联的 CloudFront 分配域名（形式为 `*distribution-id*.cloudfront.net`）以及证书 ARN。请记下输出中显示的 CloudFront 分配域名。您在下一步设置自定义域的 CNAME 值或 DNS 中的 A 记录别名目标时会需要此域名。

   > **Note**
   >
   > 新创建的自定义域名需要大约 40 分钟才能就绪。同时，您可以配置 DNS 记录别名以将自定义域名映射到关联的 CloudFront 分配域名，并在自定义域名初始化的同时设置自定义域名的基本路径映射。

10. 接下来，您可以使用 DNS 提供商配置 DNS 记录，以将自定义域名映射到关联的 CloudFront 分配。有关 Amazon Route 53 的说明，请参阅 Amazon Route 53 开发人员指南中的使用域名将流量路由到 Amazon API Gateway API。

    对于大多数 DNS 提供程序，将自定义域名添加到托管区域作为 CNAME 资源记录集。CNAME 记录名称指定您之前在域名中输入的自定义域名（如 `api.example.com`）。CNAME 记录值指定 CloudFront 分配的域名。但是，如果自定义域是顶级域 (例如，是 `example.com` 而非 `api.example.com`)，则无法使用 CNAME 记录。顶级域通常还称为组织的根域。对于顶级域，您需要使用 A 记录别名 (如果受 DNS 提供程序支持)。

借助 Route 53，您可以为自定义域名创建 A 记录别名并指定 CloudFront 分配域名作为别名目标。这意味着，Route 53 可以路由自定义域名，即使是顶级域名也是如此。有关更多信息，请参阅 Amazon Route 53 开发人员指南 中的在别名资源记录集和非别名资源记录集之间进行选择。

使用 A 记录别名还不需要公开底层 CloudFront 分配域名，因为域名映射仅发生在 Route 53 中。出于这些原因，我们建议您尽可能使用 Route 53 A 记录别名。

除了使用 API Gateway 控制台之外，您还可以使用 API Gateway REST API、AWS CLI 或某种AWS开发工具包为 API 设置自定义域名。例如，以下过程概述了使用 REST API 调用执行此操作的步骤。

使用 API Gateway REST API 设置自定义域名

1. 调用 domainname:create，并指定自定义域名和 AWS Certificate Manager 中所存储证书的 ARN。

   成功的 API 调用将返回 `201 Created` 响应，其中包含证书 ARN 以及负载中关联的 CloudFront 分配名称。
2. 请记下输出中显示的 CloudFront 分配域名。您在下一步设置自定义域的 CNAME 值或 DNS 中的 A 记录别名目标时会需要此域名。
3. 按照上述过程设置 A 记录别名以将自定义域名映射到 CloudFront 分配名称。

有关此 REST API 调用的代码示例，请参阅 domainname:create。

在 CloudTrail 中记录自定义域名的创建操作

启用 CloudTrail 以记录您的账户发出的 API Gateway 调用时，API Gateway 会在为 API 创建或更新自定义域名时记录关联的 CloudFront 分配更新。由于这些 CloudFront 分配归 API Gateway 拥有，因此每个报告的 CloudFront 分配都由以下特定于区域的 API Gateway 账户 ID 之一而不是 API 拥有者的账户 ID 来标识。

区域	账户 ID
us-east-1	392220576650
us-east-2	718770453195
us-west-1	968246515281

区域	账户 ID
us-west-2	109351309407
ca-central-1	796887884028
eu-west-1	631144002099
eu-west-2	544388816663
eu-west-3	061510835048
eu-central-1	474240146802
eu-central-2	166639821150
eu-north-1	394634713161
eu-south-1	753362059629
eu-south-2	359345898052
ap-northeast-1	969236854626
ap-northeast-2	020402002396
ap-northeast-3	360671645888
ap-southeast-1	195145609632
ap-southeast-2	798376113853
ap-southeast-3	652364314486
ap-southeast-4	849137399833
ap-south-1	507069717855
ap-south-2	644042651268
ap-east-1	174803364771

区域	账户 ID
sa-east-1	287228555773
me-south-1	855739686837
me-central-1	614065512851

配置 API 的基本路径映射，将自定义域名作为主机名

您可以使用一个自定义域名作为多个 API 的主机名。您可以通过在自定义域名上配置基本路径映射来实现这一目的。借助基本路径映射，可通过自定义域名和关联的基本路径访问自定义域下的 API。

例如，如果您创建了一个名为 `PetStore` 的 API 和一个名为 `PetShop` 的 API 并在 API Gateway 中设置了一个自定义域名 `api.example.com`，则您可以将 `PetStore` API 的 URL 设置为 `https://api.example.com` 或 `https://api.example.com/myPetStore`。`PetStore` API 与自定义域名 `myPetStore` 下的空字符串或 `api.example.com` 基本路径关联。同样，您可以为 `yourPetShop` API 分配 `PetShop` 基本路径。然后，`https://api.example.com/yourPetShop` 的 URL 就成了 `PetShop` API 的根 URL。

在为 API 设置基本路径之前，请先完成 中的步骤为 API Gateway API 设置边缘优化自定义域名

以下过程设置 API 映射，将您的自定义域名的路径映射到 API 阶段。

使用 API Gateway 控制台创建 API 映射

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择自定义域名。
3. 选择配置 API 映射。
4. 选择添加新映射。
5. 指定映射的 API、阶段和路径（可选）。
6. 选择保存。

此外，您还可以调用 API Gateway REST API、AWS CLI 或一个AWS开发工具包来设置 API（自定义域名作为其主机名）的基本路径映射。例如，以下过程概述了使用 REST API 调用执行此操作的步骤。

使用 API Gateway REST API 设置 API 的基本路径映射

- 在特定自定义域名上调用 basepathmapping:create，并指定 `basePath`、`restApiId` 和请求负载中的一个部署 `stage` 属性。

  成功的 API 调用将返回 `201 Created` 响应。

有关 REST API 调用的代码示例，请参阅 basepathmapping:create。

轮换 ACM 中导入的证书

ACM 会自动处理其所颁发证书的续订事宜。您不需要为自定义域名轮换 ACM 颁发的任何证书。CloudFront 会代表您处理这一事宜。

但是，如果您将证书导入到 ACM 并将其用于自定义域名，则您必须在证书到期前进行轮换。这包括导入域名的新第三方证书和将现有证书轮换为新证书。新导入的证书到期后，您需要重复上述过程。或者，您也可以请求 ACM 为域名颁发新证书并将现有证书轮换为 ACM 颁发的新证书。之后，您就可以让 ACM 和 CloudFront 自动为您处理证书轮换。要创建或导入新的 ACM 证书，请按照步骤为指定域名请求或导入新的 ACM 证书。

要为域名轮换证书，您可以使用 API Gateway 控制台、API Gateway REST API、AWS CLI 或某种 AWS开发工具包。

使用 API Gateway 控制台轮换 ACM 中导入的到期证书

1. 在 ACM 中请求或导入证书。
2. 回到 API Gateway 控制台。
3. 从 API Gateway 控制台的主导航窗格中选择自定义域名。
4. 选择自定义域名。
5. 选择编辑。
6. 从 ACM 证书下拉列表中选择所需证书。
7. 选择保存，开始轮换自定义域名的证书。

> Note
>
> 该过程大约需要 40 分钟才能完成。轮换完成后，您可以选择 ACM 证书旁边的双向箭头图标回滚到原始证书。

为了说明如何以编程方式轮换自定义域名的导入证书，我们概括了使用 API Gateway REST API 轮换证书的步骤。

使用 API Gateway REST API 轮换导入的证书

- 调用 domainname:update 操作，并为特定域名指定新 ACM 证书的 ARN。

调用具有自定义域名的 API

如果使用的 URL 正确，则调用具有自定义域名的 API 与调用具有默认域名的 API 是一样的。

以下示例针对两个 API（`udxjef` 和 `qf3duz`），将它们在指定区域 (`us-east-1`) 的一组默认 URL 与其给定自定义域名 (`api.example.com`) 下的相应自定义 URL 进行了对比。

具有默认域名和自定义域名的 API 的根 URL

API ID	阶段	默认 URL	基本路径	自定义 URL
udxjef	prod	https://udxjef.execute-api.us-east-1.amazonaws.com/prod	/petstore	https://api.example.com/petstore
udxjef	tst	https://udxjef.execute-api.us-east-1.amazonaws.com/tst	/petdepot	https://api.example.com/petdepot
qf3duz	dev	https://qf3duz.execute-api.us-east-1.amazonaws.com/dev	/bookstore	https://api.example.com/bookstore
qf3duz	tst	https://qf3duz.execute-api.us-east-1	/bookstand	https://api.example.com/bookstand

API ID	阶段	默认 URL	基本路径	自定义 URL
		.amazonaws.com/tst		

API Gateway 通过使用服务器名称指示 (SNI) 来支持 API 的自定义域名。您可以使用浏览器或支持 SNI 的客户端库调用具有自定义域名的 API。

API Gateway 在 CloudFront 分配中强制实施 SNI。有关 CloudFront 如何使用自定义域名的信息，请参阅 Amazon CloudFront 自定义 SSL。

在 API Gateway 中设置区域自定义域名

您可以为区域 API 端点（对于 AWS 区域）创建自定义域名。要创建自定义域名，您必须提供特定于区域的 ACM 证书。有关创建或上传自定义域名证书的更多信息，请参阅在 中准备好证书AWS Certificate Manager。

> ⚠ Important
>
> 对于 API Gateway 区域自定义域名，您必须在与 API 相同的区域中请求或导入证书。

在您创建（或迁移）包含 ACM 证书的区域自定义域名时，如果不存在服务相关角色，API Gateway 会在您的账户中创建一个这样的角色。需要使用服务相关角色，才能将 ACM 证书附加到您的区域端点。该角色名为 AWSServiceRoleForAPIGateway，将对其附加 APIGatewayServiceRolePolicy 托管策略。有关使用服务相关角色的更多信息，请参阅使用服务相关角色。

> ⚠ Important
>
> 您必须创建将自定义域名指向区域域名的 DNS 记录。这使绑定到自定义域名的流量可以路由到 API 的区域主机名。DNS 记录可以是 CNAME 或“A”类型。

主题

- 使用 API Gateway 控制台设置带 ACM 证书的区域自定义域名
- 使用 AWS CLI 设置带 ACM 证书的区域自定义域名

使用 API Gateway 控制台设置带 ACM 证书的区域自定义域名

要使用 API Gateway 控制台设置区域自定义域名，请使用以下过程。

使用 API Gateway 控制台设置区域自定义域名

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 从主导航窗格中选择自定义域名。
3. 选择创建。
4. 对于域名，输入一个域名。
5. 在“配置”下，选择区域。
6. 选择最低 TLS 版本。
7. 选择 ACM 证书。证书必须与 API 位于同一区域。
8. 选择创建。
9. 请按照配置 Route 53 以将流量路由到 API Gateway 的 Route 53 文档进行操作。

以下过程设置 API 映射，将您的自定义域名的路径映射到 API 阶段。

使用 API Gateway 控制台创建 API 映射

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择自定义域名。
3. 选择配置 API 映射。
4. 选择添加新映射。
5. 指定映射的 API、阶段和路径。
6. 选择保存。

要了解有关为自定义域设置基本路径映射的信息，请参阅配置 API 的基本路径映射，将自定义域名作为主机名。

使用 AWS CLI 设置带 ACM 证书的区域自定义域名

要使用 AWS CLI 为区域 API 设置自定义域名，请使用以下过程。

1. 调用 `create-domain-name`，并指定自定义域名和区域证书的 ARN。

   ```
   aws apigatewayv2 create-domain-name \
       --domain-name 'regional.example.com' \
       --domain-name-configurations CertificateArn=arn:aws:acm:us-
   west-2:123456789012:certificate/123456789012-1234-1234-1234-12345678
   ```

   请注意，指定的证书来自 `us-west-2` 区域，在此示例中，我们假定底层 API 来自同一区域。

   如果成功，调用返回的结果类似于以下内容：

   ```
   {
       "ApiMappingSelectionExpression": "$request.basepath",
       "DomainName": "regional.example.com",
       "DomainNameConfigurations": [
           {
               "ApiGatewayDomainName": "d-id.execute-api.us-west-2.amazonaws.com",
               "CertificateArn": "arn:aws:acm:us-west-2:123456789012:certificate/id",
               "DomainNameStatus": "AVAILABLE",
               "EndpointType": "REGIONAL",
               "HostedZoneId": "id",
               "SecurityPolicy": "TLS_1_2"
           }
       ]
   }
   ```

   `DomainNameConfigurations` 属性值返回区域 API 的主机名。您必须创建将您的自定义域名指向此区域域名的 DNS 记录。这使指向自定义域名的流量可以路由到该区域 API 的主机名。

2. 创建将自定义域名与区域域名进行关联的 DNS 记录。这使指向到自定义域名的请求可以路由到 API 的区域主机名。

3. 添加基本路径映射，以在部署阶段 (例如 `0qzs2sy7bh`) 在指定的自定义域名 (例如 `test`) 下公开指定的 API (例如 `regional.example.com`)。

   ```
   aws apigatewayv2 create-api-mapping \
       --domain-name 'regional.example.com' \
       --api-mapping-key 'myApi' \
       --api-id 0qzs2sy7bh \
       --stage 'test'
   ```

这样，在此阶段部署的使用 API 的自定义域名的基本 URL 会变为 `https://regional.example.com/myAPI`。

4. 配置 DNS 记录，以将区域自定义域名映射到给定的托管区域 ID 的主机名。首先创建一个 JSON 文件，其中包含为区域域名设置 DNS 记录的配置。以下示例显示了如何创建 DNS A 记录，以将区域自定义域名 (`regional.example.com`) 映射到在创建自定义域名时为其预配置的区域主机名 (`d-numh1z56v6.execute-api.us-west-2.amazonaws.com`)。`DNSName` 的 `HostedZoneId` 和 `AliasTarget` 属性可分别采用自定义域名的 `regionalDomainName` 和 `regionalHostedZoneId` 值。您也可以在 Amazon API Gateway 端点和配额中获取区域 Route 53 托管区域 ID。

```
{
  "Changes": [
    {
      "Action": "CREATE",
      "ResourceRecordSet": {
        "Name": "regional.example.com",
        "Type": "A",
        "AliasTarget": {
          "DNSName": "d-numh1z56v6.execute-api.us-west-2.amazonaws.com",
          "HostedZoneId": "Z2OJLYMUO9EFXC",
          "EvaluateTargetHealth": false
        }
      }
    }
  ]
}
```

5. 运行以下 CLI 命令：

```
aws route53 change-resource-record-sets \
    --hosted-zone-id {your-hosted-zone-id} \
    --change-batch file://path/to/your/setup-dns-record.json
```

其中，*`{your-hosted-zone-id}`* 是您账户中的 DNS 记录集的 Route 53 托管区域 ID。`change-batch` 参数值指向文件夹 (*`path/to/your`*) 中的 JSON 文件 (*`setup-dns-record.json`*)。

将自定义域名迁移至不同 API 终端节点

您可以在边缘优化和区域终端节点之间迁移自定义域名。首先，您向自定义域名的现有 `endpointConfiguration.types` 列表添加新终端节点配置类型。接下来，您设置 DNS 记录，将自定义域名指向新预配置的终端节点。可选的最后一步是删除过时的自定义域名配置数据。

规划迁移时，请记住对于边缘优化的 API 的自定义域名，ACM 提供的必需证书来自美国东部（弗吉尼亚北部）区域 (`us-east-1`)。此证书分发到所有地理位置。但是，对于区域 API，区域域名的 ACM 证书必须来自托管 API 的同一个区域。您可以先从 API 本地的区域请求新 ACM 证书，将不在 `us-east-1` 区域中的边缘优化自定义域名迁移到区域自定义域名。

在 API Gateway 中，最多可能需要 60 秒才能完成从边缘优化自定义域名与区域自定义域名之间的迁移。要让新创建的终端节点准备好接受流量，迁移时间还取决于您何时更新 DNS 记录。

主题

- 使用 AWS CLI 迁移自定义域名

使用 AWS CLI 迁移自定义域名

要使用 AWS CLI 将自定义域名从边缘优化的终端节点迁移到区域终端节点或者反之，请调用 `update-domain-name` 命令添加新终端节点类型，并（可选）调用 `update-domain-name` 命令以移除旧终端节点类型。

主题

- 将边缘优化自定义域名迁移为区域
- 将区域自定义域名迁移为边缘优化

将边缘优化自定义域名迁移为区域

要将边缘优化的自定义域名迁移为区域自定义域名，请调用 `update-domain-name` CLI 命令，如下所示：

```
aws apigateway update-domain-name \
    --domain-name 'api.example.com' \
    --patch-operations  [ \
        { op:'add', path: '/endpointConfiguration/types',value: 'REGIONAL' }, \
        { op:'add', path: '/regionalCertificateArn', value: 'arn:aws:acm:us-
west-2:123456789012:certificate/cd833b28-58d2-407e-83e9-dce3fd852149' } \
      ]
```

区域证书必须与区域 API 位于相同区域。

成功的响应包含 `200 OK` 状态代码以及与以下类似的正文：

```
{
    "certificateArn": "arn:aws:acm:us-
east-1:123456789012:certificate/34a95aa1-77fa-427c-aa07-3a88bd9f3c0a",
    "certificateName": "edge-cert",
    "certificateUploadDate": "2017-10-16T23:22:57Z",
    "distributionDomainName": "d1frvgze7vy1bf.cloudfront.net",
    "domainName": "api.example.com",
    "endpointConfiguration": {
        "types": [
            "EDGE",
            "REGIONAL"
        ]
    },
    "regionalCertificateArn": "arn:aws:acm:us-west-2:123456789012:certificate/
cd833b28-58d2-407e-83e9-dce3fd852149",
    "regionalDomainName": "d-fdisjghyn6.execute-api.us-west-2.amazonaws.com"
}
```

对于迁移后的区域自定义域名，生成的 `regionalDomainName` 属性返回区域 API 主机名。您必须设置 DNS 记录以将区域自定义域名指向此区域主机名。这使绑定到自定义域名的流量可以路由到区域主机。

设置 DNS 记录之后，您可以调用 [`update-domain-name`] 的 AWS CLI 命令删除边缘优化自定义域名：

```
aws apigateway update-domain-name \
    --domain-name api.example.com \
    --patch-operations [ \
            {op:'remove', path:'/endpointConfiguration/types', value:'EDGE'}, \
            {op:'remove', path:'certificateName'}, \
            {op:'remove', path:'certificateArn'} \
        ]
```

将区域自定义域名迁移为边缘优化

要将区域自定义域名迁移到边缘优化的自定义域名，请调用 `update-domain-name` 的 AWS CLI 命令，如下所示：

```
aws apigateway update-domain-name \
    --domain-name 'api.example.com' \
    --patch-operations  [ \
            { op:'add', path:'/endpointConfiguration/types',value: 'EDGE' }, \
            { op:'add', path:'/certificateName', value:'edge-cert'}, \
            { op:'add', path:'/certificateArn', value: 'arn:aws:acm:us-
east-1:123456789012:certificate/34a95aa1-77fa-427c-aa07-3a88bd9f3c0a' } \
        ]
```

边缘优化的域证书必须在 `us-east-1` 区域中创建。

成功的响应包含 `200 OK` 状态代码以及与以下类似的正文：

```
{
    "certificateArn": "arn:aws:acm:us-
east-1:738575810317:certificate/34a95aa1-77fa-427c-aa07-3a88bd9f3c0a",
    "certificateName": "edge-cert",
    "certificateUploadDate": "2017-10-16T23:22:57Z",
    "distributionDomainName": "d1frvgze7vy1bf.cloudfront.net",
    "domainName": "api.example.com",
    "endpointConfiguration": {
        "types": [
            "EDGE",
            "REGIONAL"
        ]
    },
    "regionalCertificateArn": "arn:aws:acm:us-
east-1:123456789012:certificate/3d881b54-851a-478a-a887-f6502760461d",
    "regionalDomainName": "d-cgkq2qwgzf.execute-api.us-east-1.amazonaws.com"
}
```

对于指定的自定义域名，API Gateway 将返回边缘优化的 API 主机名作为 `distributionDomainName` 属性值。您必须设置 DNS 记录以将边缘优化自定义域名指向此分配域名。这使绑定到边缘优化的自定义域名的流量可以路由到边缘优化的 API 主机名。

设置 DNS 记录之后，您可以删除自定义域名的 `REGION` 终端节点类型：

```
aws apigateway update-domain-name \
    --domain-name api.example.com \
    --patch-operations [ \
        {op:'remove', path:'/endpointConfiguration/types', value:'REGIONAL'}, \
        {op:'remove', path:'regionalCertificateArn'} \
```

```
    ]
```

此命令的结果类似于以下输出，只有边缘优化域名配置数据：

```
{
    "certificateArn": "arn:aws:acm:us-
east-1:738575810317:certificate/34a95aa1-77fa-427c-aa07-3a88bd9f3c0a",
    "certificateName": "edge-cert",
    "certificateUploadDate": "2017-10-16T23:22:57Z",
    "distributionDomainName": "d1frvgze7vy1bf.cloudfront.net",
    "domainName": "regional.haymuto.com",
    "endpointConfiguration": {
        "types": "EDGE"
    }
}
```

对 REST API 使用 API 映射

您可以使用 API 映射将 API 阶段连接到自定义域名。创建域名并配置 DNS 记录后，您可以使用 API 映射通过自定义域名向 API 发送流量。

API 映射指定了用于映射的 API、阶段以及可选的路径。例如，您可以将 API 的 `production` 阶段映射到 `https://api.example.com/orders`。

您可以将 HTTP 和 REST API 阶段映射到相同的自定义域名。

在创建 API 映射之前，您必须拥有 API、阶段和自定义域名。要了解有关创建自定义域名的更多信息，请参阅 the section called “设置区域自定义域名”。

路由 API 请求

您可以使用多个级别配置 API 映射，例如 `orders/v1/items` 和 `orders/v2/items`。

> **Note**
>
> 要配置多个级别的 API 映射，您的自定义域名必须是区域性的，并使用 TLS 1.2 安全策略。

对于具有多个级别的 API 映射，API Gateway 将请求路由到匹配路径最长的 API 映射。选择要调用的 API 时，API Gateway 仅考虑为 API 映射配置的路径，而不考虑 API 路由。如果没有与请求匹配的路径，API Gateway 会将请求发送到已映射到空路径（none）的 API。

对于使用具有多个级别的 API 映射的自定义域名，API Gateway 将请求路由到匹配前缀最长的 API 映射。

例如，考虑使用以下 API 映射的自定义域名 `https://api.example.com`：

1. (none) 映射到 API 1。
2. `orders` 映射到 API 2。
3. `orders/v1/items` 映射到 API 3。
4. `orders/v2/items` 映射到 API 4。
5. `orders/v2/items/categories` 映射到 API 5。

请求	选定的 API	说明
`https://api.example.com/orders`	`API 2`	请求完全匹配此 API 映射。
`https://api.example.com/orders/v1/items`	`API 3`	请求完全匹配此 API 映射。
`https://api.example.com/orders/v2/items`	`API 4`	请求完全匹配此 API 映射。
`https://api.example.com/orders/v1/items/123`	`API 3`	API Gateway 选择匹配路径最长的映射。请求结尾处的 `123` 不影响选择。
`https://api.example.com/orders/v2/items/categories/5`	`API 5`	API Gateway 选择匹配路径最长的映射。
`https://api.example.com/customers`	`API 1`	API Gateway 使用空映射作为“捕获全部”。
`https://api.example.com/ordersandmore`	`API 2`	API Gateway 选择匹配前缀最长的映射。对于配置了单

请求	选定的 API	说明
		级映射的自定义域名（例如，仅 `https://api.example.com/orders` 和 `https://api.example.com/`），API Gateway 会选择 `API 1`，因为没有与 `ordersandmore` 匹配的路径。

限制

- 在 API 映射中，自定义域名和映射的 API 必须位于同一个AWS账户中。
- API 映射必须仅包含字母、数字和以下字符：`$-_.+!*'()/`。
- API 映射中路径的最大长度为 300 个字符。
- 每个域名可以有 200 个具有多个级别的 API 映射。
- 您只能使用 TLS 1.2 安全策略将 HTTP API 映射到区域自定义域名。
- 您不能将 WebSocket API 映射到与 HTTP API 或 REST API 相同的自定义域名。

创建 API 映射

要创建 API 映射，您必须首先创建自定义域名、API 和阶段。有关使用自定义域名的更多信息，请参阅 the section called “设置区域自定义域名”。

例如，创建所有资源的 AWS Serverless Application Model 模板，请参阅 GitHub 上的使用 SAM 的会话。

AWS Management Console

创建 API 映射

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择自定义域名。
3. 选择您已经创建的自定义域名。

4. 选择 API 映射。
5. 选择 Configure API mappings (配置 API 映射)。
6. 选择 Add new mapping (添加新映射)。
7. 输入 API、阶段以及可选的路径。
8. 选择 Save。

AWS CLI

以下 AWS CLI 命令创建一个 API 映射。在此示例中，API Gateway 将请求发送到 `api.example.com/v1/orders`，到指定的 API 和阶段。

> **Note**
>
> 要创建具有多个级别的 API 映射，必须使用 `apigatewayv2`。

```
aws apigatewayv2 create-api-mapping \
    --domain-name api.example.com \
    --api-mapping-key v1/orders \
    --api-id a1b2c3d4 \
    --stage test
```

AWS CloudFormation

以下 AWS CloudFormation 示例会创建一个 API 映射。

> **Note**
>
> 要创建具有多个级别的 API 映射，必须使用 `AWS::ApiGatewayV2`。

```
MyApiMapping:
  Type: 'AWS::ApiGatewayV2::ApiMapping'
  Properties:
    DomainName: api.example.com
    ApiMappingKey: 'orders/v2/items'
    ApiId: !Ref MyApi
```

```
    Stage: !Ref MyStage
```

禁用 REST API 的默认终端节点

默认情况下，客户端可以通过使用 API Gateway 为 API 生成的 `execute-api` 终端节点来调用您的 API。为确保客户端只能通过使用自定义域名访问您的 API，请禁用默认 `execute-api` 终端节点。客户端仍然可以连接到您的默认端点，但它们会收到 `403 Forbidden` 状态码。

> **Note**
>
> 禁用默认终端节点时，它会影响 API 的所有阶段。

以下 AWS CLI 命令会禁用 REST API 的默认终端节点。

```
aws apigateway update-rest-api \
    --rest-api-id abcdef123 \
    --patch-operations op=replace,path=/disableExecuteApiEndpoint,value='True'
```

禁用默认终端节点后，必须部署 API 才能使更改生效。

以下 AWS CLI 命令会创建部署。

```
aws apigateway create-deployment \
    --rest-api-id abcdef123 \
    --stage-name dev
```

为 DNS 故障转移配置自定义运行状况检查

您可以使用 Amazon Route 53 运行状况检查，来控制从主要 AWS 区域的 API Gateway API 到辅助区域的 API Gateway API 的 DNS 故障转移。这可以帮助减轻在出现区域性问题时的影响。如果您使用自定义域，则无需客户端更改 API 终端节点，即可执行故障转移。

当您为别名记录选择评估目标运行状况时，只有当 API Gateway 服务在该区域不可用时，这些记录才会失败。某些情况下，您自己的 API Gateway API 可能会在此之前遇到中断。要直接控制 DNS 故障转移，请为您的 API Gateway API 配置自定义 Route 53 运行状况检查。在本示例中，您使用 CloudWatch 警报来帮助操作员控制 DNS 故障转移。有关配置故障转移的更多示例和其他注意事项，

请参阅使用 Route 53 创建灾难恢复机制和使用 AWS Lambda 和 CloudWatch 对 VPC 中的私有资源执行 Route 53 运行状况检查。

主题

先决条件

要完成此过程，您必须创建和配置以下资源：

- 您拥有的域名。
- 该域名在两个 AWS 区域中的 ACM 证书。有关更多信息，请参阅the section called “在 中准备好证书AWS Certificate Manager”。
- 您的域名的 Route 53 托管区域。有关更多信息，请参阅《Amazon Route 53 开发人员指南》中的使用托管区域。

有关如何为域名创建 Route 53 失效转移 DNS 记录的更多信息，请参阅《Amazon Route 53 开发人员指南》中的选择路由策略。有关如何监控 CloudWatch 警报的更多信息，请参阅《Amazon Route 53 开发人员指南》中的监控 CloudWatch 警报。

步骤 1：设置资源

在此示例中，您将创建以下资源来为您的域名配置 DNS 故障转移：

- 两个 AWS 区域中的 API Gateway API
- 两个 AWS 区域中同名的 API Gateway 自定义域名
- 将您的 API Gateway API 连接到自定义域名的 API Gateway API 映射
- 域名的 Route 53 故障转移 DNS 记录
- 辅助区域的 CloudWatch 警报

- 在辅助区域进行的基于 CloudWatch 警报的 Route 53 运行状况检查

首先，确保您的主区域和辅助区域都拥有所有必需的资源。辅助区域应包含警报和运行状况检查。这样，您无需依赖主区域即可进行故障转移。有关创建这些资源的 AWS CloudFormation 模板示例，请参阅`primary.yaml`和`secondary.yaml`。

> ⚠ Important
>
> 在故障转移到辅助区域之前，请确保所有必需的资源都可用。否则，您的 API 将无法为辅助区域的流量做好准备。

步骤 2：启动到辅助区域的故障转移

在以下示例中，备用区域收到 CloudWatch 指标并启动故障转移。我们使用需要操作员干预才能启动故障转移的自定义指标。

```
aws cloudwatch put-metric-data \
    --metric-name Failover \
    --namespace HealthCheck \
    --unit Count \
    --value 1 \
    --region us-west-1
```

将指标数据替换为您配置的 CloudWatch 警报的相应数据。

步骤 3：测试故障转移

调用您的 API 并验证您是否收到辅助区域的响应。如果您在步骤 1 中使用了示例模板，则故障转移后，响应将从 `{"message": "Hello from the primary Region!"}` 更改为 `{"message": "Hello from the secondary Region!"}`。

```
curl https://my-api.example.com

{"message": "Hello from the secondary Region!"}
```

步骤 4：返回主区域

要返回主区域，请发送使运行状况检查顺利通过的 CloudWatch 指标。

```
aws cloudwatch put-metric-data \
    --metric-name Failover \
    --namespace HealthCheck \
    --unit Count \
    --value 0 \
    --region us-west-1
```

将指标数据替换为您配置的 CloudWatch 警报的相应数据。

调用您的 API 并验证您是否收到主区域的响应。如果您在步骤 1 中使用了示例模板，则响应将从 `{"message": "Hello from the secondary Region!"}` 更改为 `{"message": "Hello from the primary Region!"}`。

```
curl https://my-api.example.com

{"message": "Hello from the primary Region!"}
```

后续步骤：自定义和定期测试

此示例演示了一种配置 DNS 故障转移的方法。您可以使用各种 CloudWatch 指标或 HTTP 终端节点进行运行状况检查，以管理故障转移。定期测试您的故障转移机制，确保它们按预期运行，并且操作员熟悉您的故障转移过程。

优化 REST API 的性能

在您将 API 设置为可供调用之后，您可能会意识到需要对其进行优化以提高响应能力。API Gateway 提供了一些用于优化 API 的策略，例如响应缓存和负载压缩。在本节中，您可以了解如何启用这些功能。

主题

- 启用 API 缓存以增强响应能力
- 为 API 启用负载压缩

启用 API 缓存以增强响应能力

您可以在 Amazon API Gateway 中启用 API 缓存以缓存您的端点的响应。借助缓存，您可以减少向端点发起的调用数量，同时减少向 API 发出的请求的延迟。

为某个阶段启用缓存时，API Gateway 会在指定的生存时间 (TTL) 期间（以秒为单位）缓存来自端点的响应。然后，在响应请求时，API Gateway 会从缓存中查找端点响应，而不会向端点发出请求。API 缓存的默认 TTL 值为 300 秒。最大 TTL 值为 3600 秒。TTL = 0 表示缓存功能处于禁用状态。

> **Note**
>
> 缓存是尽力而为。您可以使用 Amazon CloudWatch 中的 `CacheHitCount` 和 `CacheMissCount` 指标，以监控 API Gateway 从 API 缓存中提供的请求。

可缓存的响应的最大大小为 1048576 字节。缓存数据加密可能会增加缓存时的响应的大小。

这是一项符合 HIPAA 要求的服务。有关AWS、《1996 年健康保险可携性与责任法》(HIPAA) 以及使用AWS服务处理、存储和传输受保护的医疗信息 (PHI) 的更多信息，请参阅 HIPAA 概述。

> **Important**
>
> 为某个阶段启用缓存时，默认情况下，仅 GET 方法已启用缓存。这有助于确保您的 API 的安全性和可用性。您可以通过覆盖方法设置为其他方法启用缓存。

> **Important**
>
> 缓存功能基于您选择的缓存大小按小时计费。缓存没有资格享受 AWS 免费套餐。有关更多信息，请参阅 API Gateway 定价。

启用 Amazon API Gateway 缓存

在 API Gateway 中，您可以为特定阶段启用缓存。

启用缓存时，您必须选择一个缓存容量。一般而言，容量越大，性能越高，但成本也更高。有关支持的缓存大小，请参阅《API Gateway API 参考》中的 cacheClusterSize。

API Gateway 通过创建专用的缓存实例来实现缓存功能。这一过程耗时最多 4 分钟。

API Gateway 通过删除现有缓存实例并重新创建一个具有修改后的容量的新实例来更改缓存容量。所有现有的缓存数据均将被删除。

> **Note**
>
> 缓存容量会影响缓存实例的 CPU、内存和网络带宽。因此，缓存容量会影响缓存的性能。API Gateway 建议您运行 10 分钟的负载测试，来验证缓存容量是否适用于您的工作负载。确保负载测试期间的流量能够反映生产流量。例如，包括流量增加、流量恒定和流量高峰。负载测试应包括缓存可提供的响应以及向缓存添加项的唯一响应。监控负载测试期间的延迟、4xx、5xx、缓存命中和缓存未命中指标。根据这些指标，按需调整缓存容量。有关负载测试的更多信息，请参阅如何选择最佳 API Gateway 缓存容量以避免达到速率限制？

在 API Gateway 控制台中，您可以在阶段页面上配置缓存。您可以预调配阶段缓存，并指定默认的方法级缓存设置。如果您开启默认的方法级别缓存，则会为阶段上的所有 GET 方法开启方法级缓存，除非该方法具有方法覆盖。部署到阶段的任何其它 GET 方法都将具有方法级缓存。要为阶段的特定方法配置方法级缓存设置，可以使用方法覆盖。有关方法覆盖的更多信息，请参阅the section called “覆盖方法缓存的阶段缓存”。

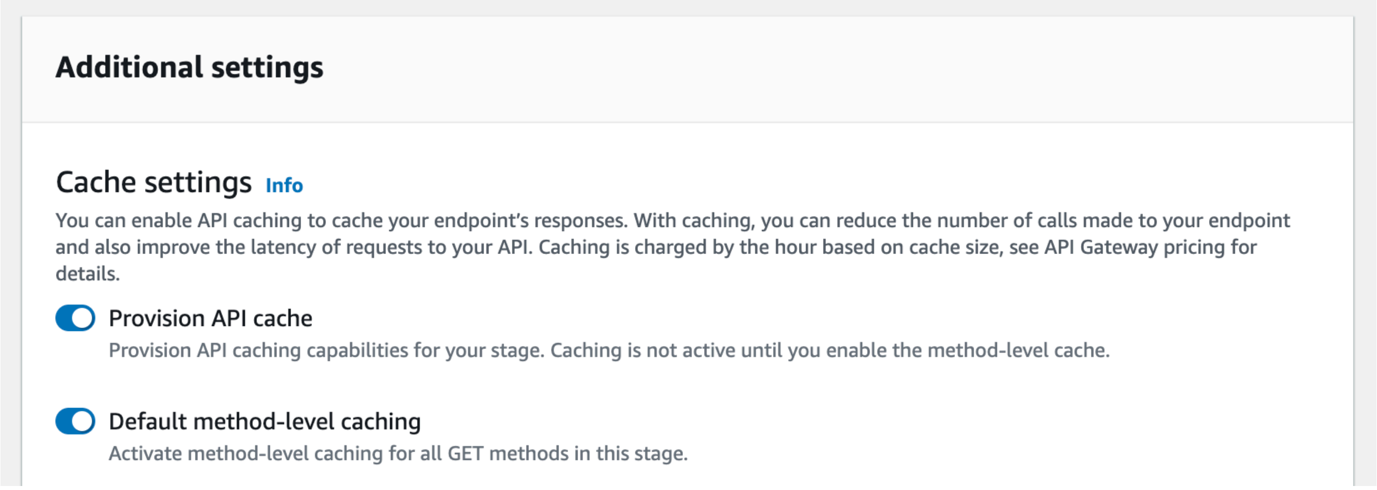
要为指定阶段配置 API 缓存，请执行以下操作：

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择阶段。
3. 在 API 的阶段列表中，选择阶段。
4. 在阶段详细信息部分中，选择编辑。
5. 在其它设置下，对于缓存设置，开启配置 API 缓存。

   这会为您的阶段预调配一个缓存集群。
6. 要为阶段激活缓存，请开启默认方法级缓存。

   这将为阶段上的所有 GET 方法开启方法级别缓存。您部署到此阶段的任何其它 GET 方法都将具有方法级缓存。

   > **Note**
   >
   > 如果您有方法级缓存的现有设置，则更改默认的方法级缓存设置不会影响该现有设置。



Additional settings

Cache settings Info

You can enable API caching to cache your endpoint's responses. With caching, you can reduce the number of calls made to your endpoint and also improve the latency of requests to your API. Caching is charged by the hour based on cache size, see API Gateway pricing for details.

● Provision API cache
Provision API caching capabilities for your stage. Caching is not active until you enable the method-level cache.

● Default method-level caching
Activate method-level caching for all GET methods in this stage.

7. 选择 Save changes（保存更改）。

> Note
>
> API Gateway 大约需要 4 分钟来完成对缓存的创建或删除。
>
> 创建缓存后，缓存集群值会从 `Create in progress` 变为 `Active`。完成缓存删除后，缓存集群值会从 `Delete in progress` 变为 `Inactive`。
>
> 当您为阶段上的所有方法开启方法级缓存时，默认方法级缓存值将变为 `Active`。如果您为阶段上的所有方法关闭方法级缓存，默认方法级缓存值将变为 `Inactive`。如果您有方法级缓存的现有设置，则更改缓存的状态不会影响该设置。

当您在阶段的缓存设置中启用缓存时，仅对 `GET` 方法进行缓存。要确保您的 API 的安全性和可用性，我们建议您不要更改此设置。不过，您可以通过 覆盖方法设置 为其他方法启用缓存。

如果想要验证缓存是否按预期正常运行，您有两种常规选择：

- 针对您的 API 和阶段，检查 CacheHitCount 和 CacheMissCount 的 CloudWatch 指标。
- 在响应中放置一个时间戳。

> Note
>
> 您不应使用来自 CloudFront 响应的 `X-Cache` 标头来确定您的 API 是否由 API Gateway 缓存实例提供服务。

覆盖方法级缓存的 API Gateway 阶段级缓存

您可以通过为特定方法开启或关闭缓存来覆盖阶段级缓存设置。您还可以修改 TTL 期间，或者为缓存的响应开启或关闭加密。

如果您在阶段详细信息中更改默认方法级缓存设置，则不会影响具有覆盖功能的方法级缓存设置。

如果您预计某个进行缓存的方法将在其响应中接收敏感数据，请在缓存设置中选择加密缓存数据。

要使用控制台为各个方法配置 API 缓存，请执行以下操作：

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 API。
3. 选择阶段。
4. 在 API 的阶段列表中，展开阶段，然后在 API 中选择方法。
5. 在方法覆盖部分，选择编辑。
6. 在方法设置部分，打开或关闭启用方法缓存或自定义任何其他所需选项。

   > **Note**
   >
   > 在您为阶段预调配缓存集群之前，缓存不会处于活动状态。

7. 选择保存。

将方法或集成参数用作索引缓存响应的缓存键

当缓存的方法或集成具有参数 (可以是自定义标头、URL 路径或查询字符串格式) 时，您可以使用部分或全部参数来构建缓存键。API Gateway 可以缓存方法的响应，具体取决于使用的参数值。

> **Note**
>
> 在资源上设置缓存时需要缓存键。

例如，假设您在以下格式中提出一个请求：

```
GET /users?type=... HTTP/1.1
host: example.com
...
```

在这个请求中，`type` 的值可以是 `admin` 或 `regular`。如果您添加 `type` 参数作为缓存键的组成部分，则 `GET /users?type=admin` 的响应将与 `GET /users?type=regular` 的响应分开缓存。

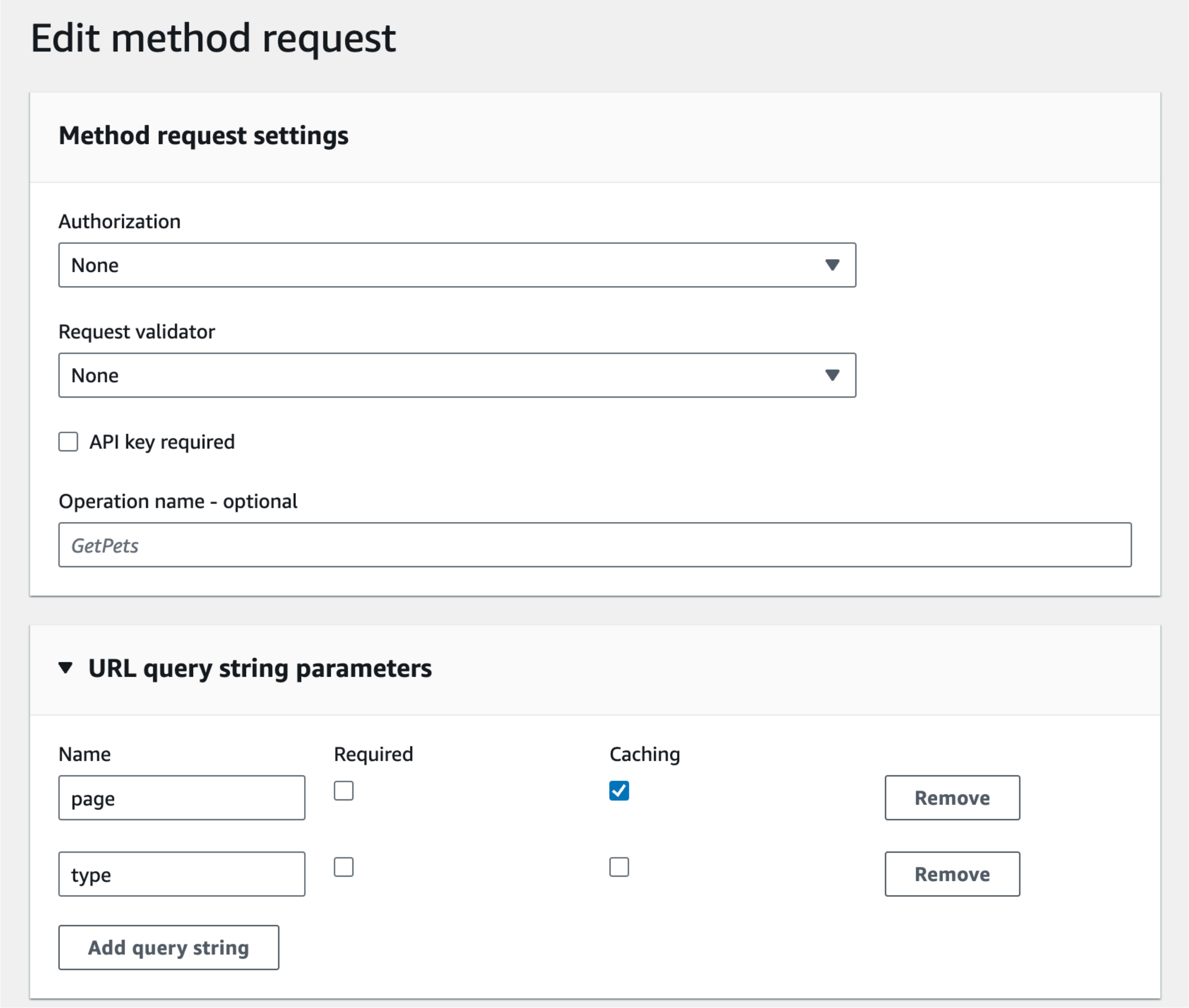
当某种方法或集成请求采用多个参数时，您可以选择添加部分或全部参数来创建缓存键。例如，对于在 TTL 期内按列出的顺序提出的以下请求，您可以在缓存键中只添加 `type` 参数：

```
GET /users?type=admin&department=A HTTP/1.1
host: example.com
...
```

此请求的响应将被缓存，并用于服务以下请求：

```
GET /users?type=admin&department=B HTTP/1.1
host: example.com
...
```

要在 API Gateway 控制台中将一个方法或集成请求参数添加为缓存键的一部分，请在添加参数后选择缓存。



Edit method request

Method request settings

Authorization: None

Request validator: None

☐ API key required

Operation name - optional: *GetPets*

▼ **URL query string parameters**

Name	Required	Caching	
page	☐	☑	Remove
type	☐	☐	Remove

刷新 API Gateway 中的 API 阶段缓存

启用 API 缓存时，您可以刷新 API 阶段的整个缓存，以确保您的 API 客户端可以从集成端点获得最新响应。

要刷新 API 阶段缓存，请选择阶段操作菜单，然后选择刷新阶段缓存。

> **Note**
>
> 刷新缓存之后，在重新构建缓存之前，从集成端点为响应提供服务。在此期间，发送到集成端点的请求数量可能会增加。这可能会临时增加 API 的整体延迟。

使 API Gateway 缓存条目失效

您的 API 客户端可以使某个现有缓存条目失效，也可以从各个请求的集成端点重新加载该条目。客户端必须发送一个包含 `Cache-Control: max-age=0` 标头的请求。客户端将直接从集成端点 (而非缓存) 收到响应，前提是客户端获得授权执行此操作。这会将现有缓存条目替换为从集成端点获得的新响应。

要为客户端授予权限，请将以下格式的策略附加到用户的 IAM 执行角色。

Note

不支持跨账户缓存无效化。

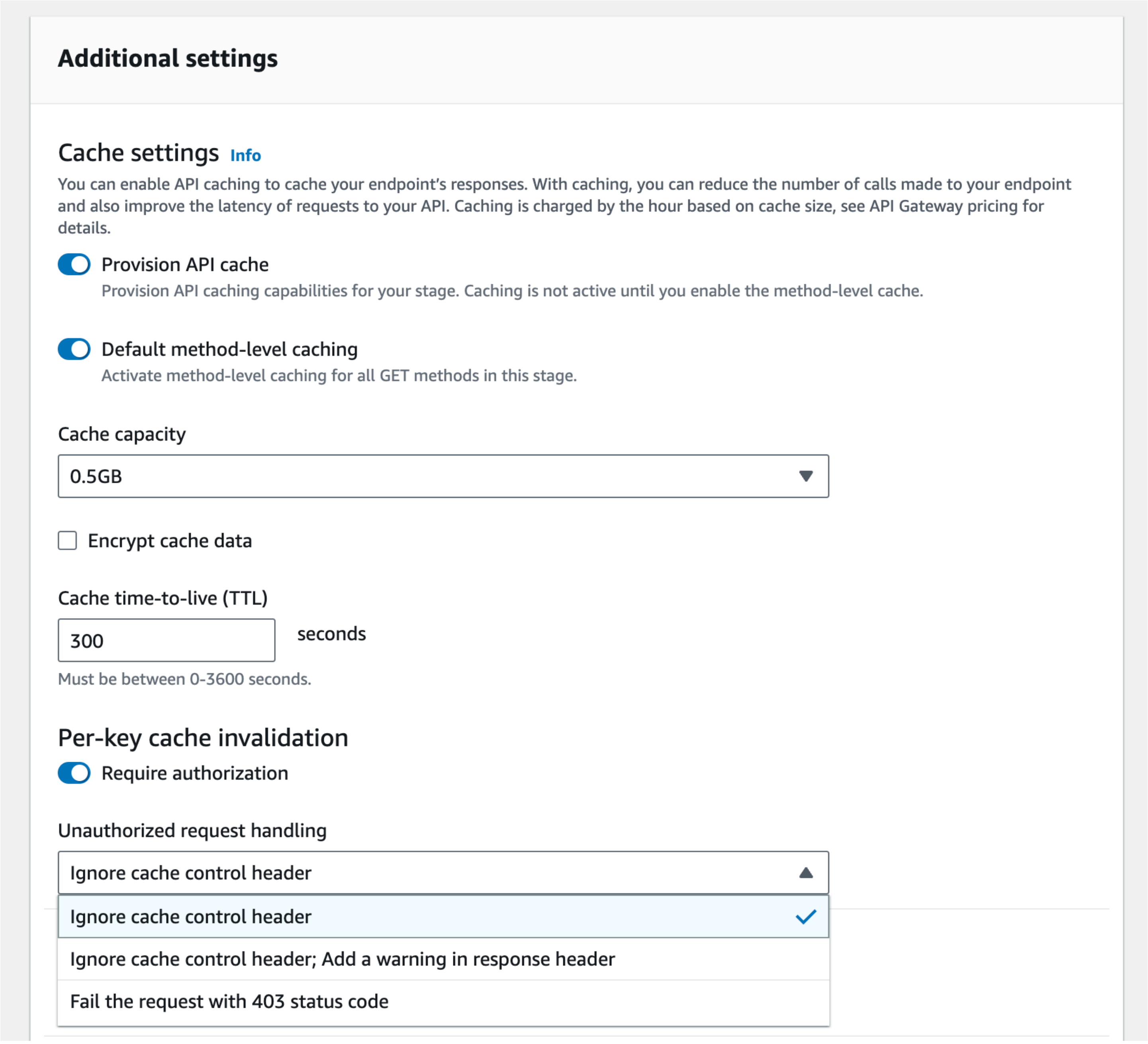
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:InvalidateCache"
      ],
      "Resource": [
        "arn:aws:execute-api:region:account-id:api-id/stage-name/GET/resource-path-
specifier"
      ]
    }
  ]
}
```

此策略允许 API Gateway 执行服务让指定资源上的请求对应的缓存失效。要指定一组目标资源，请将 `account-id` 的 ARN 值中的 `api-id`、`Resource` 和其他条目替换为通配符 (*)。有关如何为 API Gateway 执行服务设置权限的更多信息，请参阅使用 IAM 权限控制对 API 的访问。

如果您未应用 `InvalidateCache` 策略（或在控制台中选中需要授权复选框），则任何客户端均可使 API 缓存失效。如果大部分或全部客户端都使 API 缓存失效，这会显著增加 API 的延迟。

策略到位后，将启用缓存并需要授权。

您可以从 API Gateway 控制台的未经授权的请求处理中选择相应选项，来控制对未经授权的请求的处理方式。



Additional settings

Cache settings Info

You can enable API caching to cache your endpoint's responses. With caching, you can reduce the number of calls made to your endpoint and also improve the latency of requests to your API. Caching is charged by the hour based on cache size, see API Gateway pricing for details.

● Provision API cache
Provision API caching capabilities for your stage. Caching is not active until you enable the method-level cache.

● Default method-level caching
Activate method-level caching for all GET methods in this stage.

Cache capacity: 0.5GB

☐ Encrypt cache data

Cache time-to-live (TTL): 300 seconds
Must be between 0-3600 seconds.

Per-key cache invalidation

● Require authorization

Unauthorized request handling: Ignore cache control header

- Ignore cache control header ✓
- Ignore cache control header; Add a warning in response header
- Fail the request with 403 status code

这三个选项会引发以下行为：

- 请求失败，显示 403 状态代码：返回“403 Unauthorized”响应。

  要使用 API 设置该选项，请使用 `FAIL_WITH_403`。
- 忽略缓存控制标头；在响应中添加一条警告标头：处理请求并在响应中添加一条警告标头。

要使用 API 设置该选项，请使用 `SUCCEED_WITH_RESPONSE_HEADER`。

- 忽略缓存控制标头：处理请求，但不在响应中添加警告标头。

  要使用 API 设置该选项，请使用 `SUCCEED_WITHOUT_RESPONSE_HEADER`。

为 API 启用负载压缩

API Gateway 允许您的客户端使用支持的内容编码之一调用具有压缩负载的 API。默认情况下，API Gateway 支持解压缩方法请求负载。但是，您必须配置 API 以启用方法响应负载的压缩。

要在 `API` 上启用压缩，请在创建 API 时或在创建 API 之后，将 `minimumCompressionsSize` 属性设置为介于 0 到 10485760（10M 字节）之间的非负整数。要在 API 上禁用压缩，请将 `minimumCompressionSize` 设置为空值或者将属性与值一起删除。您可以使用 API Gateway 控制台、AWS CLI 或 API Gateway REST API 为 API 启用或禁用 API 压缩。

如果您希望在任意大小的负载上应用压缩，请将 `minimumCompressionSize` 值设置为零。不过，压缩较小大小的数据实际上可能会增加最终数据大小。此外，API Gateway 中的压缩和客户端中的解压缩可能会增加总体延迟并需要更多的计算时间。您应该根据 API 运行测试用例，确定最佳值。

客户端可以针对 API Gateway 提交具有压缩负载及合适 `Content-Encoding` 标头的 API 请求，以解压缩并应用适用的映射模板，然后再将请求传递到集成端点。启用解压缩并部署了 API 之后，如果在方法请求中指定了合适的 `Accept-Encoding` 标头，客户端可以接收具有压缩负载的 API 响应。

当集成终端节点预期并返回了未压缩的 JSON 负载时，为未压缩 JSON 负载配置的任何映射模板适用于压缩负载。对于压缩方法请求负载，API Gateway 解压缩负载，应用映射模板，然后将映射的请求传递到集成终端节点。对于未压缩的集成响应负载，API Gateway 应用映射模板，压缩映射的负载，并将压缩的负载返回客户端。

主题

为 API 启用负载压缩

您可以使用 API Gateway 控制台、AWS CLI 或AWS开发工具包为 API 启用压缩。

对于现有 API，您必须在启用压缩后部署 API，这样更改才能生效。对于新 API，您可以在 API 设置完成后部署该 API。

> **Note**
>
> 最高优先级内容编码必须是 API Gateway 支持的编码。如果不是，将不对响应负载应用压缩。

主题

- 使用 API Gateway 控制台为 API 启用负载压缩
- 使用 AWS CLI 为 API 启用负载压缩
- API Gateway 支持的内容编码

使用 API Gateway 控制台为 API 启用负载压缩

以下过程介绍如何为 API 启用负载压缩。

使用 API Gateway 控制台启用负载压缩

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择现有 API 或者创建新 API。
3. 在主导航窗格中，选择 API 设置。
4. 在 API 详细信息部分中，选择编辑。
5. 开启内容编码以启用负载压缩。对于最小正文大小，为最小压缩大小输入数字（字节数）。要关闭压缩，请关闭内容编码选项。
6. 选择 Save changes（保存更改）。

使用 AWS CLI 为 API 启用负载压缩

要使用 AWS CLI 创建新 API 并启用压缩，请按以下所示调用 `create-rest-api` 命令：

```
aws apigateway create-rest-api \
    --name "My test API" \
    --minimum-compression-size 0
```

要使用 AWS CLI 在现有 API 上启用压缩，请按以下所示调用 update-rest-api 命令：

```
aws apigateway update-rest-api \
    --rest-api-id 1234567890 \
    --patch-operations op=replace,path=/minimumCompressionSize,value=0
```

`minimumCompressionSize` 属性具有 0 到 10485760（10M 字节）之间的非负整数值。它可测量压缩阈值。如果负载大小小于该值，则不会对负载应用压缩或解压缩。将它设置为零允许对任何负载大小进行压缩。

要使用 AWS CLI 禁用压缩，请按以下所示调用 update-rest-api 命令：

```
aws apigateway update-rest-api \
    --rest-api-id 1234567890 \
    --patch-operations op=replace,path=/minimumCompressionSize,value=
```

您也可以将 `value` 设置为空字符串 `""` 或在上一调用中忽略 `value` 属性。

API Gateway 支持的内容编码

API Gateway 支持以下内容编码：

- `deflate`
- `gzip`
- `identity`

根据 [RFC 7231](RFC 7231) 规范，API Gateway 还支持以下 `Accept-Encoding` 标头格式：

- `Accept-Encoding:deflate,gzip`
- `Accept-Encoding:`
- `Accept-Encoding:*`
- `Accept-Encoding:deflate;q=0.5,gzip;q=1.0`
- `Accept-Encoding:gzip;q=1.0,identity;q=0.5,*;q=0`

使用压缩负载调用 API 方法

要使用压缩负载发出 API 请求，客户端必须使用[支持的内容编码](支持的内容编码)之一设置 `Content-Encoding` 标头。

假设您使用 API 客户端并希望调用 PetStore API 方法 (`POST /pets`)。不要使用以下 JSON 输出来调用方法：

```
POST /pets
Host: {petstore-api-id}.execute-api.{region}.amazonaws.com
Content-Length: ...

{
  "type": "dog",
  "price": 249.99
}
```

相反，您可以通过使用 GZIP 编码压缩的相同负载来调用方法：

```
POST /pets
Host: {petstore-api-id}.execute-api.{region}.amazonaws.com
Content-Encoding:gzip
Content-Length: ...

���RPP*�,HU�RPJ�OW��e&���L,�,-y�j
```

API Gateway 收到请求时，它会验证是否支持指定的内容编码。然后，它尝试解压缩具有指定内容编码的负载。如果解压缩成功，它会将请求分派到集成终端节点。如果不支持指定的编码或者提供的负载未使用指定编码压缩，API Gateway 返回 `415 Unsupported Media Type` 错误响应。如果此错误在解压缩的早期阶段发生（尚未确定您的 API 和阶段），则此错误不会记录到 CloudWatch Logs。

接收具有压缩负载的 API 响应

在启用压缩的 API 上发出请求时，客户端可以指定具有 支持内容编码 的 `Accept-Encoding` 标头，选择接收特定格式的压缩响应负载。

API Gateway 仅在满足以下条件时压缩响应负载：

- 传入请求具有 `Accept-Encoding` 标头和支持的内容编码及格式。

> **Note**
>
> 如果未设置标头，默认值为 `*`（在 RFC 7231 中定义）。在这种情况下，API Gateway 不会压缩有效负载。某些浏览器或客户端可能会为启用了压缩的请求自动添加 `Accept-Encoding` (例如 `Accept-Encoding:gzip, deflate, br`)。这会在 API Gateway 中触

> 发负载压缩。如果没有对支持的 `Accept-Encoding` 标头值的明确规范，API Gateway 不会压缩负载。

- 在 API 上设置了 `minimumCompressionSize` 以启用压缩。
- 集成响应没有 `Content-Encoding` 标头。
- 集成响应负载的大小，在应用了适用的映射模板之后，大于或等于指定的 `minimumCompressionSize` 值。

API Gateway 在压缩负载之前，应用为集成响应配置的任意映射模板。如果集成响应包含 `Content-Encoding` 标头，API Gateway 假设集成响应负载已经压缩并跳过压缩处理。

以 PetStore API 示例及以下请求为例：

```
GET /pets
Host: {petstore-api-id}.execute-api.{region}.amazonaws.com
Accept: application/json
```

后端对具有未压缩 JSON 负载的请求的响应类似于以下内容：

```
200 OK

[
  {
    "id": 1,
    "type": "dog",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "cat",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
```

要将此输出作为压缩负载接收，您的 API 客户端可以按以下所示提交请求：

```
GET /pets
Host: {petstore-api-id}.execute-api.{region}.amazonaws.com
Accept-Encoding:gzip
```

客户端接收具有 `Content-Encoding` 标头和 GZIP 编码负载的响应，类似于以下内容：

```
200 OK
Content-Encoding:gzip
...

���RP�

J�)JV
�:P^IeA*������+(�L �X�YZ�ku0L0B7!9��C#�&����Y��a���^�X
```

在压缩响应负载之后，只收取压缩后数据大小的数据传输费用。

将 REST API 分发给客户端

本节提供有关向客户分发 API Gateway API 的详细信息。分发 API 包括生成开发工具包供客户下载并与其客户端应用程序集成，记录您的 API 以便客户了解如何从客户端应用程序调用它，并将 API 作为产品的一部分提供。

主题

- 创建和使用带 API 密钥的使用计划
- 记录 REST API
- 在 API Gateway 中为 REST API 生成开发工具包
- 通过 AWS Marketplace 销售 API Gateway API

创建和使用带 API 密钥的使用计划

创建、测试和部署 API 后，您可以实施 API Gateway 使用计划，将它们作为面向客户的产品/服务提供。您可以配置使用计划和 API 密钥，以允许客户访问选定的 API，并根据定义的限制和配额开始对这些API的请求进行节流。这些可以在 API 或 API 方法级别设置。

什么是使用计划和 API 密钥？

一个使用计划指定谁可以访问一个或多个部署的 API 阶段和方法 — 可以选择设置目标请求速率来启动限制请求。该计划使用 API 密钥来标识 API 客户端，以及针对每个密钥对关联的 API 阶段的访问人员。

API 密钥 是字母数字字符串值，可将它分发给应用程序开发人员（要向其授予对您的 API 的访问权的客户）。您可以将 API 密钥与 Lambda 授权方、IAM 角色或 Amazon Cognito 一起使用，以控制对 API 的访问。API Gateway 可以代表您生成 API 密钥，也可以从 CSV 文件中导入 API 密钥。您可以在 API Gateway 中生成 API 密钥，或从外部源将其导入 API Gateway。有关更多信息，请参阅 the section called “使用 API Gateway 控制台设置 API 密钥”。

API 密钥具有一个名称和值。（术语“API 密钥”和“API 密钥值”经常互换使用。）名称不能超过 1024 个字符。值为一个长度在 20 到 128 个字符之间的字母数字字符串，例如，`apikey1234abcdefghij0123456789`。

> **Important**
>
> API 密钥值必须是唯一的。如果您尝试使用不同的名称和相同的值来创建两个 API 密钥，API Gateway 会将它们视为同一 API 密钥。
>
> 一个 API 密钥可与多个使用计划关联。一个使用计划可与多个阶段关联。但是，对于 API 的每个阶段，给定 API 密钥只能与一个使用计划关联。

一个限制设置请求限制应该开始的目标点。这可以在 API 或 API 方法级别设置。

配额限制可设置在指定的时间间隔内提交的包含给定 API 密钥的最大目标请求数。您可以配置单独的 API 方法，要求根据使用计划配置进行 API 密钥授权。

限制和配额限制适用于跨一个使用计划内的所有 API 阶段聚合的各个 API 密钥的请求。

> **Note**
>
> 使用计划节流和配额不是硬性限制，而是在尽力而为的基础上应用的。在某些情况下，客户端可能会超过您设置的配额。不要依靠使用计划配额或节流来控制成本或阻止对 API 的访问。考虑使用 AWS Budgets 监控成本和 AWS WAF 来管理 API 请求。

API 密钥和使用计划的最佳实践

以下是使用 API 密钥和使用计划时要遵循的建议的最佳实践。

> ⚠ Important
>
> - 请勿使用 API 密钥进行身份验证或授权以控制对 API 的访问权限。如果您在一个使用计划中有多个 API，则具有该使用计划中的一个 API 的有效 API 密钥的用户可以访问该使用计划中的所有 API。相反，要控制对 API 的访问权限，请使用 IAM 角色、Lambda 授权方或 Amazon Cognito 用户群体。
> - 使用 API Gateway 生成的 API 密钥。API 密钥不应包含机密信息；客户端通常使用可记录的标头传输这些信息。

- 如果您使用开发人员门户发布 API，请注意，给定使用计划中的所有 API 均可由客户订阅，即使您尚未向您的客户显示它们。
- 在某些情况下，客户端可能会超过您设置的配额。不要依靠使用计划来控制成本。考虑使用 AWS Budgets 监控成本和 AWS WAF 来管理 API 请求。
- 将 API 密钥添加到使用计划后，更新操作可能需要几分钟才能完成。

配置使用计划的步骤

以下步骤概述了作为 API 所有者，您如何为客户创建和配置使用计划。

配置使用计划

1. 创建一个或多个 API，将方法配置为需要 API 密钥，并将 API 部署到各阶段。
2. 生成或导入 API 密钥以分发给将使用您的 API 的应用程序开发人员（您的客户）。
3. 创建具有所需限制和配额限制的使用计划。
4. 将 API 阶段和 API 密钥与使用计划关联。

API 调用方必须在 API 请求的 `x-api-key` 标头中提供一个已分配的 API 密钥。

Note

要在使用计划中包括 API 方法，您必须将单独的 API 方法配置为需要 API 密钥。有关可考虑的最佳实践，请参阅 the section called “API 密钥和使用计划的最佳实践”。

选择一个 API 密钥源

在将使用计划与 API 关联并在 API 方法上启用 API 密钥时，针对 API 的传入请求必须包含 API 密钥。API Gateway 将读取密钥并将它与使用计划中的密钥进行比较。如果匹配，API Gateway 将基于计划的请求限制和配额限制请求。否则，将引发 `InvalidKeyParameter` 异常。作为结果，调用方将收到 `403 Forbidden` 响应。

您的 API Gateway API 可从下面两个源之一接收 API 密钥：

`HEADER`

您将 API 密钥分发给您的客户，并要求其将 API 密钥作为每个传入请求的 `X-API-Key` 标头传递。

`AUTHORIZER`

您可以让 Lambda 授权方将 API 密钥作为授权响应的一部分返回。有关授权响应的更多信息，请参阅the section called “来自 API Gateway Lambda 授权方的输出”。

Note

有关可考虑的最佳实践，请参阅 the section called “API 密钥和使用计划的最佳实践”。

使用 API Gateway 控制台为 API 选择 API 密钥源

1. 登录 API Gateway 控制台。
2. 选择现有 API 或者创建新 API。
3. 在主导航窗格中，选择 API 设置。
4. 在 API 详细信息部分中，选择编辑。
5. 在 API 密钥源下，从下拉列表中选择 `Header` 或 `Authorizer`。
6. 选择 Save changes（保存更改）。

要使用 AWS CLI 为 API 选择 API 密钥源，请按以下所示调用 `update-rest-api` 命令：

```
aws apigateway update-rest-api --rest-api-id 1234123412 --patch-operations
 op=replace,path=/apiKeySource,value=AUTHORIZER
```

要让客户端提交 API 密钥，请在以上 CLI 命令中将 `value` 设置为 `HEADER`。

要使用 API Gateway REST API 为 API 选择 API 密钥源，请按以下所示调用 `restapi:update`：

```
PATCH /restapis/fugvjdxtri/ HTTP/1.1
Content-Type: application/json
Host: apigateway.us-east-1.amazonaws.com
X-Amz-Date: 20160603T205348Z
Authorization: AWS4-HMAC-SHA256 Credential={access_key_ID}/20160603/us-east-1/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature={sig4_hash}

{
  "patchOperations" : [
    {
        "op" : "replace",
        "path" : "/apiKeySource",
        "value" : "HEADER"
    }
  ]
}
```

要让授权方返回 API 密钥，请在之前的 `value` 输入中将 `AUTHORIZER` 设置为 `patchOperations`。

请使用下列过程之一以在方法调用中使用以标头为源的 API 密钥或授权方返回的 API 密钥，具体取决于您选择的 API 密钥源类型：

使用以标头为源的 API 密钥：

1. 使用所需的 API 方法创建 API，然后将 API 部署到某个阶段。
2. 新建使用计划，或选择现有使用计划。将部署的 API 阶段添加到使用计划。为使用计划附加 API 密钥，或在计划中选择现有 API 密钥。请记下所选的 API 密钥值。
3. 设置 API 方法以获得 API 密钥。
4. 将此 API 重新部署到同一阶段。如果将此 API 部署到新阶段，请确保更新使用计划，附加新的 API 阶段。

客户端现在可以调用 API 方法，同时为 `x-api-key` 标头提供所选的 API 密钥作为标头值。

使用以授权方为源的 API 密钥：

1. 使用所需的 API 方法创建 API，然后将 API 部署到某个阶段。
2. 新建使用计划，或选择现有使用计划。将部署的 API 阶段添加到使用计划。为使用计划附加 API 密钥，或在计划中选择现有 API 密钥。请记下所选的 API 密钥值。
3. 创建基于令牌的 Lambda 授权方。包括，`usageIdentifierKey:`*`{api-key}`* 作为授权响应的根级属性。有关创建基于令牌的授权方的说明，请参阅the section called “TOKEN 授权方 Lambda 函数示例”。
4. 设置 API 方法以获得 API 密钥，并针对这些方法启用 Lambda 授权方。
5. 将此 API 重新部署到同一阶段。如果将此 API 部署到新阶段，请确保更新使用计划，附加新的 API 阶段。

客户端现在可以调用获得了 API 密钥的方法，而无需明确提供 API 密钥。返回授权方的 API 密钥将自动使用。

使用 API Gateway 控制台设置 API 密钥

要设置 API 键，请执行以下操作：

- 将 API 方法配置为需要 API 键。
- 为区域的 API 创建或导入 API 键。

在设置 API 键之前，您必须先创建一个 API 并将其部署到某个阶段。创建 API 密钥值后，无法更改此值。

有关如何使用 API Gateway 控制台创建和部署 API 的说明，请分别参阅在 API Gateway 中开发 REST API和在 Amazon API Gateway 中部署 REST API。

创建 API 密钥后，必须将其与使用计划相关联。有关更多信息，请参阅 使用 API Gateway 控制台创建、配置和测试使用计划。

> **Note**
>
> 有关可考虑的最佳实践，请参阅 the section called “API 密钥和使用计划的最佳实践”。

主题

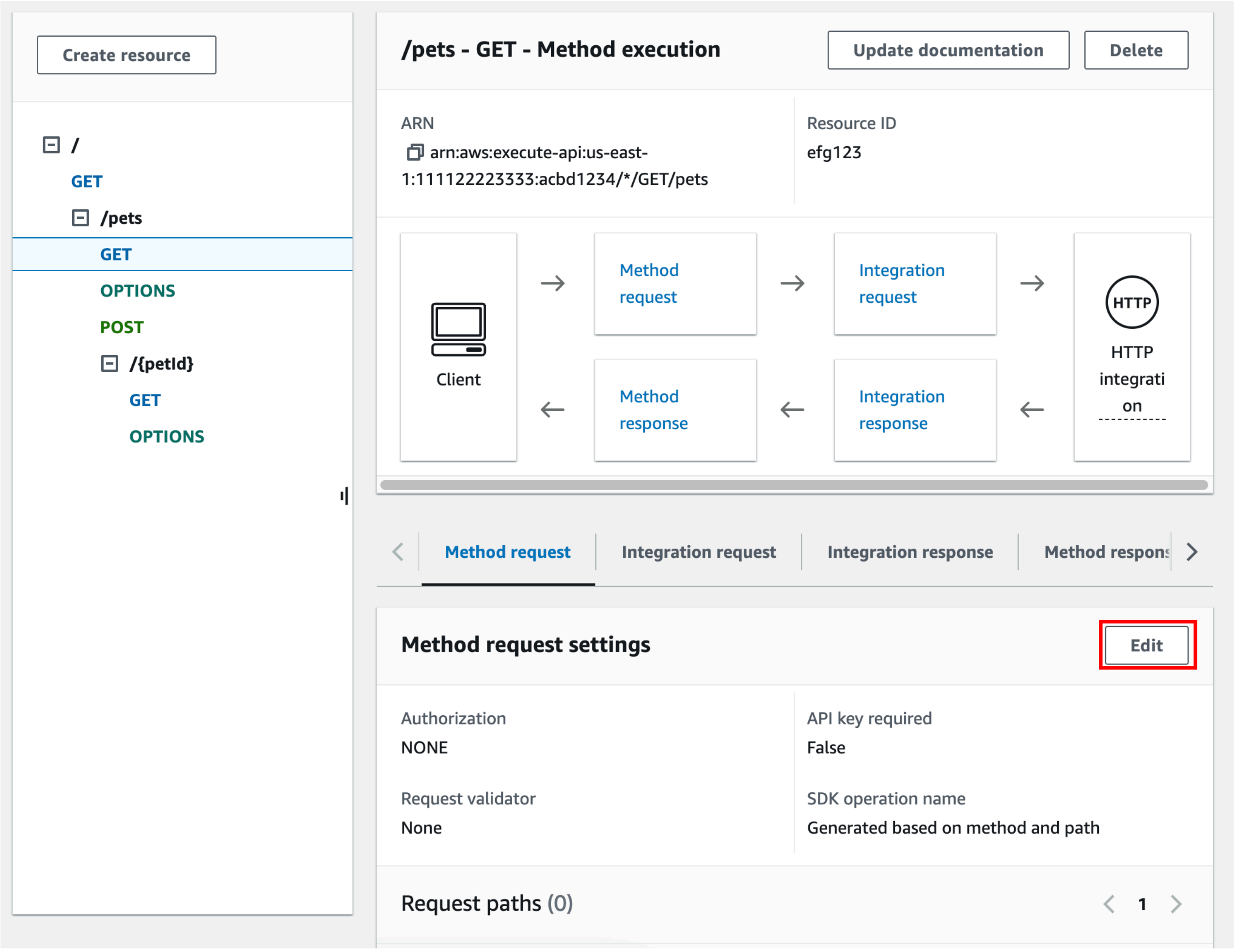
- 要求方法使用 API
- 创建 API 密钥
- 导入 API 密钥

要求方法使用 API

以下过程介绍如何将 API 方法配置为需要 API 密钥。

将 API 方法配置为需要 API 密钥

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 在 API Gateway 主导航窗格中，选择资源。
4. 在资源下，创建新方法或选择现有方法。
5. 在方法请求选项卡上的方法请求设置下，选择编辑。



/pets - GET - Method execution

ARN
arn:aws:execute-api:us-east-1:111122223333:acbd1234/*/GET/pets

Resource ID
efg123

Method request settings

Authorization
NONE

API key required
False

Request validator
None

SDK operation name
Generated based on method and path

6. 选择需要 API 密钥。
7. 选择保存。
8. 部署或重新部署 API 以使该要求生效。

如果需要 API 密钥选项设置为 `false` 并且您没有执行上述步骤，则与 API 阶段关联的任何 API 密钥都不会用于该方法。

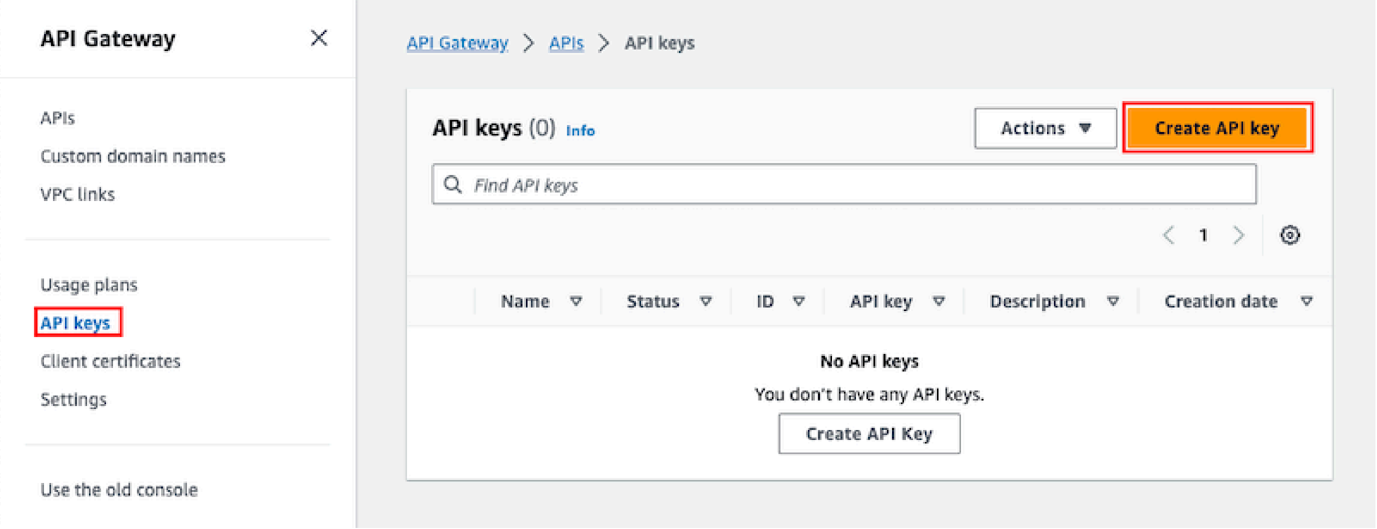
创建 API 密钥

如果您已创建或导入 API 密钥以用于使用计划，则可跳过此过程和下一过程。

创建 API 密钥

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。

2. 选择一个 REST API。
3. 在 API Gateway 主导航窗格中，选择 API 密钥。
4. 选择创建 API 密钥。



5. 对于名称，输入名称。
6. （可选）对于描述，输入描述。
7. 对于 API 密钥，选择自动生成可让 API Gateway 生成密钥值，或者选择自定义以创建您自己的密钥值。
8. 选择保存。

导入 API 密钥

以下过程介绍如何导入 API 密钥以用于使用计划。

导入 API 密钥

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 在主导航窗格中，选择 API 密钥。
4. 选择操作下拉菜单，然后选择导入 API 密钥。
5. 要加载逗号分隔的密钥文件，请选择选择文件。您也可以在文本编辑器中输入密钥。有关文件格式的信息，请参阅 the section called “API Gateway API 密钥文件格式”。
6. 选择收到警告停止以在出现错误时停止导入，或选择忽略警告以在出现警告时继续导入有效的密钥条目。

7. 选择导入以导入您的 API 密钥。

使用 API Gateway 控制台创建、配置和测试使用计划

创建使用计划之前，请确保您已设置所需的 API 密钥。有关更多信息，请参阅 使用 API Gateway 控制台设置 API 密钥。

本节介绍如何使用 API Gateway 控制台创建和实施使用计划。

主题

- 将您的 API 迁移到默认使用计划（如果需要）
- 创建使用计划
- 测试使用计划
- 维护使用计划

将您的 API 迁移到默认使用计划（如果需要）

如果您在 2016 年 8 月 11 日推出使用计划特征之后 开始使用 API Gateway，则会自动在所有受支持区域为您启用使用计划。

如果您在此日期之前开始使用 API Gateway，则可能需要迁移到默认使用计划。在选定区域中首次使用计划之前，系统将提示您选择启用使用计划选项。在您启用此选项后，您将为与现有 API 密钥关联的每个唯一 API 阶段创建默认使用计划。在默认使用计划中，最初不设置限制或配额限制，API 密钥和 API 阶段之间的关联将复制到使用计划中。API 的行为方式将与之前相同。但是，您必须使用 `UsagePlan` `apiStages` 属性（而不是使用 `ApiKey` `stageKeys` 属性）将指定的 API 阶段值（`apiId` 和 `stage`）与所含的 API 密钥进行关联（通过 `UsagePlanKey`）。

要检查您是否已迁移到默认使用计划，请使用 `get-account` CLI 命令。在命令输出中，当启用使用计划时，`features` 列表将包括 `"UsagePlans"` 的条目。

您还可以使用 AWS CLI 将您的 API 迁移到默认使用计划，如下所示：

使用 AWS CLI 迁移到默认使用计划

1. 调用此 CLI 命令：`update-account`
2. 对于 `cli-input-json` 参数，使用以下 JSON：

```
[
```

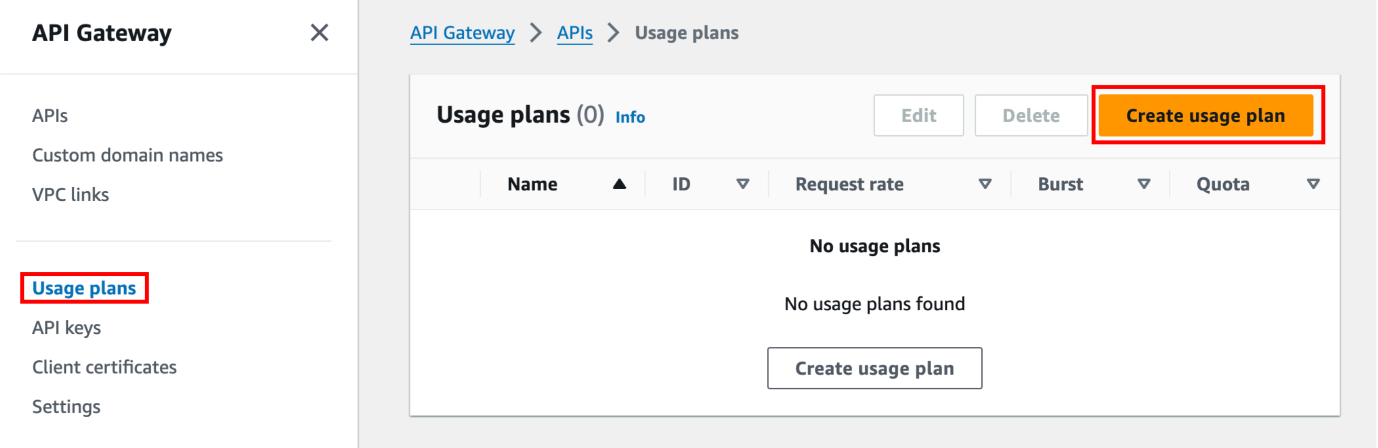
```
    {
        "op": "add",
        "path": "/features",
        "value": "UsagePlans"
    }
]
```

创建使用计划

以下过程介绍如何创建使用计划。

创建使用计划

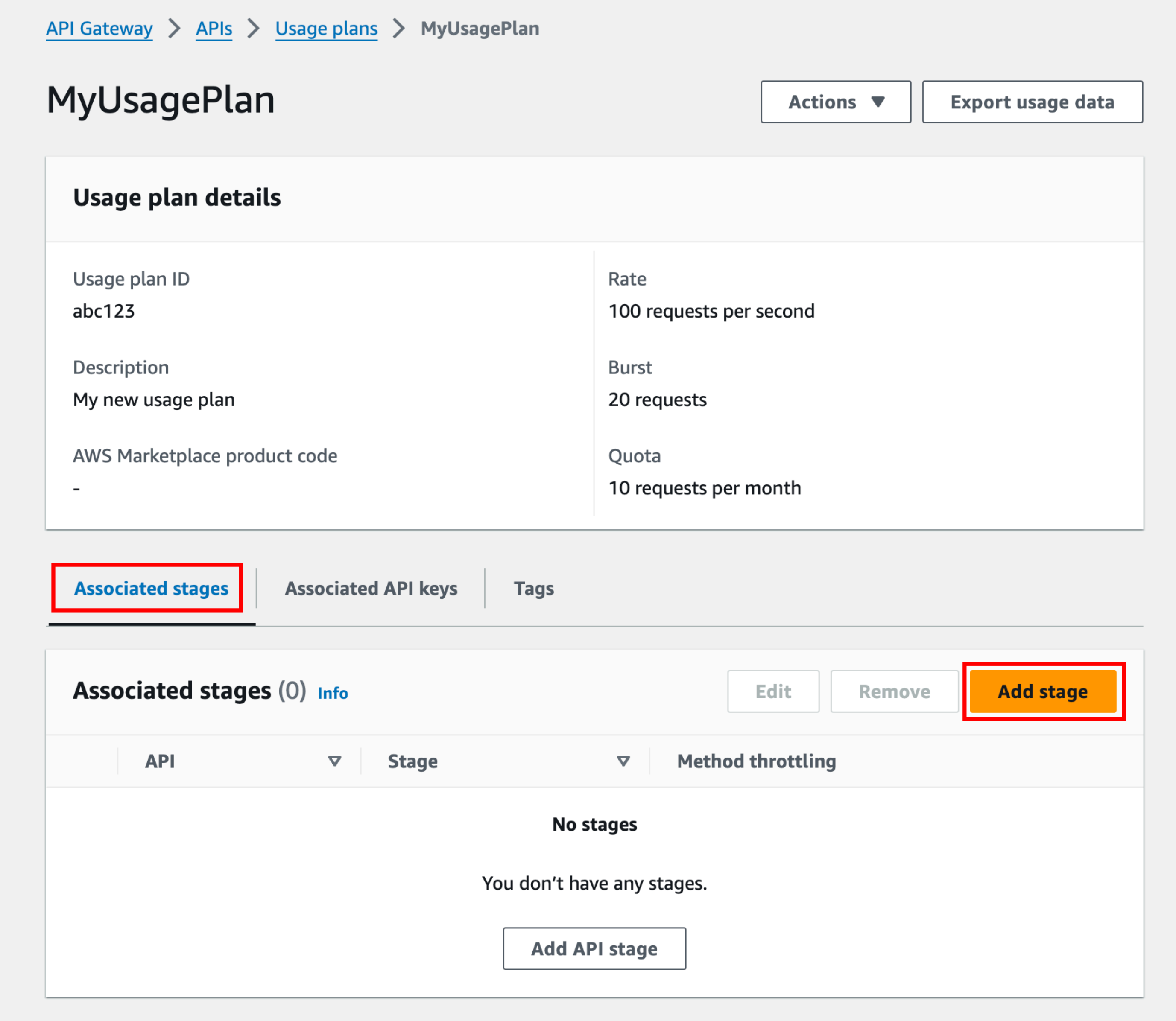
1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在 API Gateway 主导航窗格中，选择使用计划，然后选择创建使用计划。



3. 对于名称，输入名称。
4. （可选）对于描述，输入描述。
5. 默认情况下，使用计划会启用节流。为您的使用计划输入速率和突增。选择节流可关闭节流。
6. 默认情况下，使用计划会针对一个时间段启用配额。对于请求，输入用户在使用计划的时间段内，可以发出的请求总数。选择配额可关闭配额。
7. 选择创建使用计划。

向使用计划添加阶段

1. 选择您的使用计划。
2. 在关联的阶段选项卡下，选择添加阶段。



Usage plan details

Usage plan ID
abc123

Description
My new usage plan

AWS Marketplace product code
-

Rate
100 requests per second

Burst
20 requests

Quota
10 requests per month

Associated stages (0)

API	Stage	Method throttling

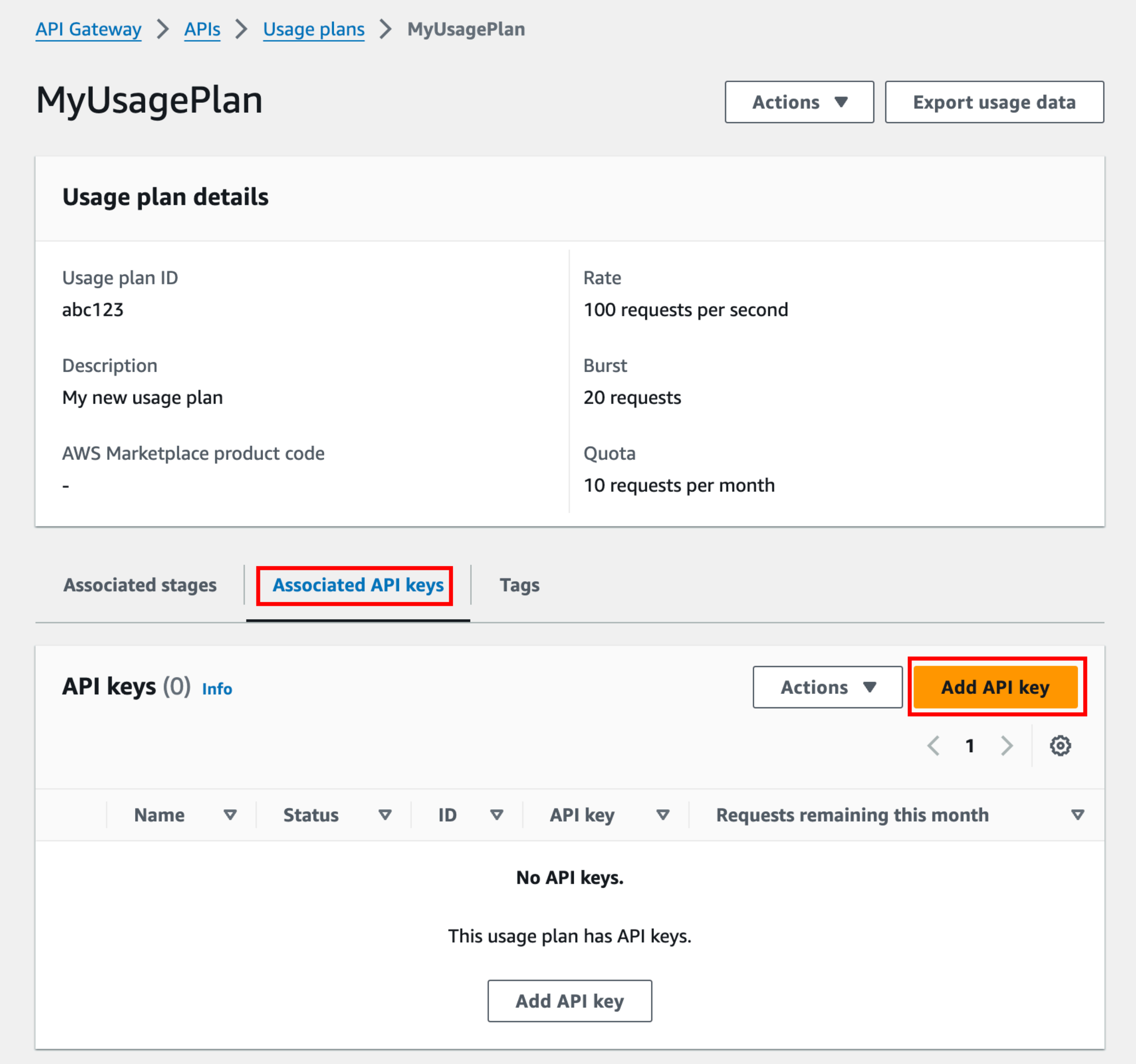
No stages

You don't have any stages.

3. 对于 API，选择一个 API。
4. 对于阶段，选择一个阶段。
5. （可选）要开启方法级别的节流，请执行以下操作：
   a. 选择方法级别节流，然后选择添加方法。
   b. 对于资源，从您的 API 中选择一个资源。
   c. 对于方法，从您的 API 中选择一种方法。
   d. 为您的使用计划输入速率和突增。
6. 选择添加至使用计划。

向使用计划添加密钥

1. 在关联的 API 密钥选项卡下，选择添加 API 密钥。



Usage plan details

Usage plan ID abc123	Rate 100 requests per second
Description My new usage plan	Burst 20 requests
AWS Marketplace product code -	Quota 10 requests per month

Name	Status	ID	API key	Requests remaining this month

No API keys.

This usage plan has API keys.

2. a. 要将现有密钥与您的使用计划关联，请选择添加现有密钥，然后从下拉菜单中选择您的现有密钥。

   b. 要创建新 API 密钥，请选择创建并添加新密钥，然后创建新密钥。有关如何创建新密钥的更多信息，请参阅创建 API 密钥。

3. 选择添加 API 密钥。

测试使用计划

要测试使用计划，您可以使用 AWS 软件开发工具包、AWS CLI 或类似于 Postman 的 REST API 客户端。有关使用 Postman 测试使用计划的示例，请参阅 测试使用计划。

维护使用计划

维护使用计划涉及监控给定时间段内的已用配额和剩余配额，并（如果需要）将剩余配额扩展指定的量。以下过程介绍如何监控配额。

监控已用配额和剩余配额

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在 API Gateway 主导航窗格中，选择使用计划。
3. 选择一个使用计划。
4. 选择关联的 API 密钥选项卡，查看每个密钥在时间段内剩余的请求数。
5. （可选）选择导出使用数据，然后选择开始日期和结束日期。接下来，为导出的数据选择 JSON 或 CSV 格式，然后选择导出。

   以下示例显示了一个导出的文件。

```
{
  "thisPeriod": {
    "px1KW6...qBazOJH": [
      [
        0,
        5000
      ],
      [
        0,
        5000
      ],
      [
        0,
        10
      ]
    ]
  },
  "startDate": "2016-08-01",
  "endDate": "2016-08-03"
```

```
}
```

示例中的使用率数据显示了某 API 客户端在 2016 年 8 月 1 日至 2016 年 8 月 3 日期间的每日使用率数据，由 API 密钥 (`px1KW6...qBazOJH`) 标识。每个每日使用率数据均显示已用配额和剩余配额。在本例中，订阅者尚未使用任何分配的配额，并且 API 所有者或管理员已在第三天将剩余配额从 5000 减至 10。

以下过程介绍如何修改配额。

扩展剩余配额

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在 API Gateway 主导航窗格中，选择使用计划。
3. 选择一个使用计划。
4. 选择关联的 API 密钥选项卡，查看每个密钥在时间段内剩余的请求数。
5. 选择 API 密钥，然后选择授予使用延期。
6. 为剩余请求配额输入一个数字。对于使用计划中的时间段，您可以增加剩余的请求数或减少剩余的请求数。
7. 选择更新配额。

使用 API Gateway REST API 设置 API 密钥

要设置 API 键，请执行以下操作：

- 将 API 方法配置为需要 API 键。
- 为区域的 API 创建或导入 API 键。

在设置 API 键之前，您必须先创建一个 API 并将其部署到某个阶段。创建 API 密钥值后，无法更改此值。

要使用 REST API 调用创建和部署 API，请分别参阅 `restapi:create` 和 `deployment:create`。

Note

有关可考虑的最佳实践，请参阅 the section called “API 密钥和使用计划的最佳实践”。

主题

- 要求方法使用 API
- 创建或导入 API 密钥

要求方法使用 API

要让某个方法使用 API，请执行以下操作之一：

- 调用 `method:put` 以创建方法。在请求负载中将 `apiKeyRequired` 设置为 `true`。
- 调用 `method:update` 将 `apiKeyRequired` 设置为 `true`。

创建或导入 API 密钥

要创建或导入 API 密钥，请执行以下操作之一：

- 调用 `apikey:create` 以创建 API 密钥。
- 调用 `apikey:import` 以从某个文件导入 API 密钥。有关文件格式的信息，请参阅 API Gateway API 密钥文件格式。

您无法更改新 API 密钥的值。要了解如何配置使用计划，请参阅使用 API Gateway CLI 和 REST API 创建、配置和测试使用计划。

使用 API Gateway CLI 和 REST API 创建、配置和测试使用计划

配置使用计划之前，必须已执行以下操作：将选定 API 的方法设置为需要 API 密钥，将 API 部署或重新部署到某个阶段，并且创建或导入一个或多个 API 密钥。有关更多信息，请参阅 使用 API Gateway REST API 设置 API 密钥。

要使用 API Gateway REST API 配置使用计划，请按照以下说明操作（假设您已创建要添加到使用计划的 API）。

主题

- 迁移到默认使用计划
- 创建使用计划
- 使用 AWS CLI 管理使用计划
- 测试使用计划

迁移到默认使用计划

首次创建使用计划时，可通过调用包含以下正文的 `account:update`，将与选定 API 密钥关联的现有 API 阶段迁移到某个使用计划：

```
{
  "patchOperations" : [ {
    "op" : "add",
    "path" : "/features",
    "value" : "UsagePlans"
  } ]
}
```

有关迁移与 API 密钥关联的 API 阶段的更多信息，请参阅迁移到 API Gateway 控制台中的默认使用计划。

创建使用计划

以下过程介绍如何创建使用计划。

使用 REST API 创建使用计划

1. 调用 `usageplan:create` 以创建使用计划。在负载中指定该计划的名称和描述、关联的 API 阶段、速率限制和配额。

   记下生成的使用计划标识符。您在下一个步骤中需要用到它。

2. 请执行下列操作之一：

   a. 调用 `usageplankey:create` 以向使用计划中添加 API 密钥。在负载中指定 `keyId` 和 `keyType`。

      要向使用计划添加更多 API 密钥，请重复之前的调用，每次添加一个 API 密钥。

   b. 调用 `apikey:import` 以直接向指定的使用计划添加一个或多个 API 密钥。请求负载应包含 API 密钥值、关联的使用计划标识符、布尔值标记 (用于表明已为使用计划启用密钥)，还可能包含 API 密钥名称和描述。

      以下 `apikey:import` 请求示例将向一个使用计划 (用 `key` 进行标识) 添加三个 API 密钥 (用 `name`、`description` 和 `usageplanIds` 进行标识)：

      ```
      POST /apikeys?mode=import&format=csv&failonwarnings=fase HTTP/1.1
      ```

```
Host: apigateway.us-east-1.amazonaws.com
Content-Type: text/csv
Authorization: ...

key,name, description, enabled, usageplanIds
abcdef1234ghijklmnop8901234567, importedKey_1, firstone,  tRuE, n371pt
abcdef1234ghijklmnop0123456789, importedKey_2, secondone, TRUE, n371pt
abcdef1234ghijklmnop9012345678, importedKey_3,          , true, n371pt
```

因此，将创建三个 `UsagePlanKey` 资源并将其添加到 `UsagePlan`。

您也可以通过这种方式向多个使用计划添加 API 密钥。要执行此操作，请将每个 `usageplanIds` 列值更改为逗号分隔的字符串，其中包含选定的使用计划标识符并用引号引起来 (`"n371pt,m282qs"` 或 `'n371pt,m282qs'`)。

> **Note**
>
> 一个 API 密钥可与多个使用计划关联。一个使用计划可与多个阶段关联。但是，对于 API 的每个阶段，给定 API 密钥只能与一个使用计划关联。

使用 AWS CLI 管理使用计划

以下代码示例说明如何通过调用 [update-usage-plan](https://docs.aws.amazon.com/cli/latest/reference/apigateway/update-usage-plan.html) 命令在使用计划中添加、删除或修改方法级别的限制设置。

> **Note**
>
> 确保为您的 API 将 `us-east-1` 更改为相应的区域值。

要添加或替换用于限制单个资源和方法的速率限制，请执行以下操作：

```
aws apigateway --region us-east-1 update-usage-plan --usage-plan-id <planId> --patch-operations
                        op="replace",path="/apiStages/<apiId>:<stage>/throttle/<resourcePath>/<httpMethod>/rateLimit",value="0.1"
```

要添加或替换用于限制单个资源和方法的突增限制，请执行以下操作：

```
aws apigateway --region us-east-1 update-usage-plan --usage-plan-id <planId>
 --patch-operations op="replace",path="/apiStages/<apiId>:<stage>/
throttle/<resourcePath>/<httpMethod>/burstLimit",value="1"
```

要删除单个资源和方法的方法级别限制设置，请执行以下操作：

```
aws apigateway --region us-east-1 update-usage-plan --usage-plan-id <planId>
 --patch-operations op="remove",path="/apiStages/<apiId>:<stage>/
throttle/<resourcePath>/<httpMethod>",value=""
```

要删除 API 的所有方法级别限制设置，请执行以下操作：

```
aws apigateway --region us-east-1 update-usage-plan --usage-plan-id <planId> --patch-
operations op="remove",path="/apiStages/<apiId>:<stage>/throttle ",value=""
```

下面是使用 Pet Store 示例 API 的示例：

```
aws apigateway --region us-east-1 update-usage-plan --usage-plan-id <planId> --patch-
operations
                op="replace",path="/apiStages/<apiId>:<stage>/throttle",value='"{\"/
pets/GET\":{\"rateLimit\":1.0,\"burstLimit\":1},\"//GET\":{\"rateLimit\":1.0,
\"burstLimit\":1}}"'
```

测试使用计划

为了举例说明，我们将使用在教程：通过导入示例创建 REST API中创建的 PetStore API。假定 API 被配置为使用 API 密钥 `Hiorr45VR...c4GJc`。以下步骤介绍如何测试使用计划。

测试您的使用计划

- 使用 GET 查询参数对使用计划中的 API (例如 `/pets`) 的 Pets 资源 (`?type=...&page=...`) 发出 `xbvxlpijch` 请求：

  ```
  GET /testStage/pets?type=dog&page=1 HTTP/1.1
  x-api-key: Hiorr45VR...c4GJc
  Content-Type: application/x-www-form-urlencoded
  Host: xbvxlpijch.execute-api.ap-southeast-1.amazonaws.com
  X-Amz-Date: 20160803T001845Z
  ```

```
Authorization: AWS4-HMAC-SHA256 Credential={access_key_ID}/20160803/ap-southeast-1/
execute-api/aws4_request, SignedHeaders=content-type;host;x-amz-date;x-api-key,
 Signature={sigv4_hash}
```

> **Note**
>
> 必须将此请求提交至 API Gateway 的 `execute-api` 组件，并在所需的 `Hiorr45VR...c4GJc` 标头中提供所需的 API 密钥（例如 `x-api-key`）。

成功的响应会返回 `200 OK` 状态代码以及包含来自后端的请求结果的负载。如果您忘记设置 `x-api-key` 标头或使用不正确的密钥进行设置，则会收到 `403 Forbidden` 响应。不过，如果您未将方法配置为需要 API 密钥，则可能会收到 `200 OK` 响应 (无论是否正确设置 `x-api-key` 标头)，并且会绕过使用计划的限制和配额限制。

有时，当出现使 API Gateway 无法对请求实施使用计划限制或配额的内部错误时，API Gateway 将处理该请求，而不会应用使用计划中指定的限制或配额。不过，它会在 CloudWatch 中记录错误消息 `Usage Plan check failed due to an internal error`。您可以忽略此类偶尔出现的错误。

使用 AWS CloudFormation 创建和配置 API 密钥和使用计划。

您可以使用 AWS CloudFormation 对于 API 方法要求 API 密钥并为 API 创建使用计划。示例 AWS CloudFormation 模板执行以下操作：

- 使用 `GET` 和 `POST` 方法创建 API Gateway API。
- `GET` 和 `POST` 方法需要 API 密钥。此 API 从每个传入请求的 `X-API-KEY` 标头接收密钥。
- 创建 API 密钥。
- 创建使用计划，以指定每月限额为每月 1000 个请求，节流速率限制为每秒 100 个请求，而节流突增限制为每秒 200 个请求。
- 为 `GET` 方法指定每秒 50 个请求的方法级节流速率限制和每秒 100 个请求的方法级节流突增限制。
- 将 API 阶段和 API 密钥与使用计划关联。

```
AWSTemplateFormatVersion: 2010-09-09
Parameters:
  StageName:
```

```
      Type: String
      Default: v1
      Description: Name of API stage.
    KeyName:
      Type: String
      Default: MyKeyName
      Description: Name of an API key
  Resources:
    Api:
      Type: 'AWS::ApiGateway::RestApi'
      Properties:
        Name: keys-api
        ApiKeySourceType: HEADER
    PetsResource:
      Type: 'AWS::ApiGateway::Resource'
      Properties:
        RestApiId: !Ref Api
        ParentId: !GetAtt Api.RootResourceId
        PathPart: 'pets'
    PetsMethodGet:
      Type: 'AWS::ApiGateway::Method'
      Properties:
        RestApiId: !Ref Api
        ResourceId: !Ref PetsResource
        HttpMethod: GET
        ApiKeyRequired: true
        AuthorizationType: NONE
        Integration:
          Type: HTTP_PROXY
          IntegrationHttpMethod: GET
          Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets/
    PetsMethodPost:
      Type: 'AWS::ApiGateway::Method'
      Properties:
        RestApiId: !Ref Api
        ResourceId: !Ref PetsResource
        HttpMethod: POST
        ApiKeyRequired: true
        AuthorizationType: NONE
        Integration:
          Type: HTTP_PROXY
          IntegrationHttpMethod: GET
          Uri: http://petstore-demo-endpoint.execute-api.com/petstore/pets/
    ApiDeployment:
```

```
    Type: 'AWS::ApiGateway::Deployment'
    DependsOn:
      - PetsMethodGet
    Properties:
      RestApiId: !Ref Api
      StageName: !Sub '${StageName}'
  UsagePlan:
    Type: AWS::ApiGateway::UsagePlan
    DependsOn:
      - ApiDeployment
    Properties:
      Description: Example usage plan with a monthly quota of 1000 calls and method-level throttling for /pets GET
      ApiStages:
        - ApiId: !Ref Api
          Stage: !Sub '${StageName}'
          Throttle:
            "/pets/GET":
              RateLimit: 50.0
              BurstLimit: 100
      Quota:
        Limit: 1000
        Period: MONTH
      Throttle:
        RateLimit: 100.0
        BurstLimit: 200
      UsagePlanName: "My Usage Plan"
  ApiKey:
    Type: AWS::ApiGateway::ApiKey
    Properties:
      Description: API Key
      Name: !Sub '${KeyName}'
      Enabled: True
  UsagePlanKey:
    Type: AWS::ApiGateway::UsagePlanKey
    Properties:
      KeyId: !Ref ApiKey
      KeyType: API_KEY
      UsagePlanId: !Ref UsagePlan
Outputs:
  ApiRootUrl:
    Description: Root Url of the API
    Value: !Sub 'https://${Api}.execute-api.${AWS::Region}.amazonaws.com/${StageName}'
```

配置一个将 API 密钥与 OpenAPI 定义结合使用的方法

您可以使用 OpenAPI 定义来要求对方法使用 API 密钥。

对于每个方法，创建一个安全要求对象，来要求使用 API 密钥调用该方法。然后，在安全定义中定义 `api_key`。创建 API 后，将新的 API 阶段添加到您的使用计划中。

以下示例创建一个 API，并要求为 `POST` 和 `GET` 方法提供 API 密钥：

OpenAPI 2.0

```
{
  "swagger" : "2.0",
  "info" : {
    "version" : "2024-03-14T20:20:12Z",
    "title" : "keys-api"
  },
  "basePath" : "/v1",
  "schemes" : [ "https" ],
  "paths" : {
    "/pets" : {
      "get" : {
        "responses" : { },
        "security" : [ {
          "api_key" : [ ]
        } ],
        "x-amazon-apigateway-integration" : {
          "type" : "http_proxy",
          "httpMethod" : "GET",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets/",
          "passthroughBehavior" : "when_no_match"
        }
      },
      "post" : {
        "responses" : { },
        "security" : [ {
          "api_key" : [ ]
        } ],
        "x-amazon-apigateway-integration" : {
          "type" : "http_proxy",
          "httpMethod" : "GET",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets/",
          "passthroughBehavior" : "when_no_match"
        }
```

```
      }
    }
  },
  "securityDefinitions" : {
    "api_key" : {
      "type" : "apiKey",
      "name" : "x-api-key",
      "in" : "header"
    }
  }
}
```

OpenAPI 3.0

```
{
  "openapi" : "3.0.1",
  "info" : {
    "title" : "keys-api",
    "version" : "2024-03-14T20:20:12Z"
  },
  "servers" : [ {
    "url" : "{basePath}",
    "variables" : {
      "basePath" : {
        "default" : "v1"
      }
    }
  } ],
  "paths" : {
    "/pets" : {
      "get" : {
        "security" : [ {
          "api_key" : [ ]
        } ],
        "x-amazon-apigateway-integration" : {
          "httpMethod" : "GET",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets/",
          "passthroughBehavior" : "when_no_match",
          "type" : "http_proxy"
        }
      },
      "post" : {
        "security" : [ {
```

```
          "api_key" : [ ]
        } ],
        "x-amazon-apigateway-integration" : {
          "httpMethod" : "GET",
          "uri" : "http://petstore-demo-endpoint.execute-api.com/petstore/pets/",
          "passthroughBehavior" : "when_no_match",
          "type" : "http_proxy"
        }
      }
    }
  },
  "components" : {
    "securitySchemes" : {
      "api_key" : {
        "type" : "apiKey",
        "name" : "x-api-key",
        "in" : "header"
      }
    }
  }
}
```

API Gateway API 密钥文件格式

API Gateway 可以从逗号分隔值 (CSV) 格式的外部文件导入 API 密钥，然后将导入的密钥与一个或多个使用计划关联。导入的文件必须包含 `Name` 和 `Key` 列。列标头名称不区分大小写，并且这些列可以采用任何顺序，如以下示例所示：

```
Key,name
apikey1234abcdefghij0123456789,MyFirstApiKey
```

`Key` 值必须介于 20 到 128 个字符之间。`Name` 值不能超过 1024 个字符。

API 密钥文件也可以包含 `Description`、`Enabled` 或 `UsagePlanIds` 列，如以下示例所示：

```
Name,key,description,Enabled,usageplanIds
MyFirstApiKey,apikey1234abcdefghij0123456789,An imported key,TRUE,c7y23b
```

当密钥与多个使用计划关联时，`UsagePlanIds` 值是用双引号或单引号引起来的使用计划 ID 的逗号分隔字符串，如以下示例所示：

```
Enabled,Name,key,UsageplanIds
true,MyFirstApiKey,apikey1234abcdefghij0123456789,"c7y23b,glvrsr"
```

允许使用无法识别的列，但会忽略这些列。默认值为空字符串或 `true` 布尔值。

同一个 API 密钥可以多次导入，最新版本将覆盖前一个版本。如果两个 API 密钥具有相同的 `key` 值，则这两个密钥相同。

ⓘ Note

有关可考虑的最佳实践，请参阅 the section called “API 密钥和使用计划的最佳实践”。

记录 REST API

为帮助客户了解和使用您的 API，您应记录 API。为帮助您记录 API，API Gateway 支持您为各个 API 实体添加和更新帮助内容，这是 API 开发流程中不可或缺的一部分。API Gateway 将存储源内容，并使您能够存档不同版本的文档。您可以将一个文档版本与一个 API 阶段关联，将特定于阶段的文档快照导出为外部 OpenAPI 文件，并将该文件以文档发布的形式分发。

要记录您的 API，您可以调用 API Gateway REST API，使用其中一个 AWS SDK，使用适用于 API Gateway 的 AWS CLI，或者使用 API Gateway 控制台。此外，您还可以导入或导出在外部 OpenAPI 文件中定义的文档部分。

要与开发人员共享 API 文档，您可以使用开发人员门户。有关示例，请参阅 AWS 合作伙伴网络 (APN) 博客上的 Integrating ReadMe with API Gateway to Keep Your Developer Hub Up to Date（将 ReadMe 与 API Gateway 集成，让您的开发人员中心保持最新状态）。

主题

- API 文档在 API Gateway 中的表示形式
- 使用 API Gateway 控制台记录 API
- 使用 API Gateway 控制台发布 API 文档
- 使用 API Gateway REST API 记录 API
- 使用 API Gateway REST API 发布 API 文档
- 导入 API 文档
- 控制对 API 文档的访问权限

API 文档在 API Gateway 中的表示形式

API Gateway API 文档包含多个与特定 API 实体关联的独立文档部分，其中包括 API、资源、方法、请求、响应、消息参数（即路径、查询、标头），以及授权方和模型。

在 API Gateway 中，文档部分以 DocumentationPart 资源表示。整个 API 文档以 DocumentationParts 集合表示。

记录 API 的过程涉及创建 `DocumentationPart` 实例，将其添加至 `DocumentationParts` 集合，以及随着 API 的发展维护文档部分的各个版本。

主题

- 文档部分
- 文档版本

文档部分

DocumentationPart 资源是一个 JSON 对象，存储着适用于各个 API 实体的文档内容。它的 `properties` 字段包含作为键/值对映射的文档内容。它的 `location` 属性用于标识关联的 API 实体。

内容映射的形状由您 (API 开发人员) 决定。键/值对的值可以是字符串、数字、布尔值、对象或数组。`location` 对象的形状取决于目标实体类型。

`DocumentationPart` 资源支持内容继承：API 实体的文档内容适用于该 API 实体的子级。有关子实体和内容继承的定义的更多信息，请参阅从包含更通用规格的 API 实体中继承内容。

文档部分的位置

DocumentationPart 实例的 location 属性用于标识要将关联内容应用于哪个 API 实体。API 实体可以是 API Gateway REST API 资源，例如 RestApi、Resource、Method、MethodResponse、Authorizer 或 Model。实体也可以是消息参数，例如 URL 路径参数、查询字符串参数、请求或响应标头参数、请求或响应正文或响应状态代码。

要指定一个 API 实体，请将 `location` 对象的 type 属性设置为 `API`、`AUTHORIZER`、`MODEL`、`RESOURCE`、`METHOD`、`PATH_PARAMETER`、`QUERY_PARAMETER`、`REQUEST_` 或 `RESPONSE_BODY` 中的一个。

根据 API 实体的 `type`，您可以指定其他 `location` 属性，包括 method、name、path 和 statusCode。并非所有这些属性都对给定 API 实体有效。举例来说，`type`、`path`、`name` 和

`statusCode` 是 `RESPONSE` 实体的有效属性；仅 `type` 和 `path` 是 `RESOURCE` 实体的有效位置属性。在 `location` 的 `DocumentationPart` 中为给定 API 实体添加无效字段的操作是错误的。

并非所有有效的 `location` 字段都是必需字段。例如，`type` 是所有 API 实体的有效和必需 `location` 字段。但是，`method`、`path` 和 `statusCode` 是 `RESPONSE` 实体的有效属性，却并非必需属性。在未明确规定时，有效的 `location` 字段将作为其默认值。默认的 `path` 值为 `/`，即 API 的根资源。`method` 或 `statusCode` 的默认值为 `*`，分别表示任意方法值或任意状态代码值。

文档部分的内容

`properties` 值以 JSON 字符串的格式编码。`properties` 值包含您为满足文档要求而选择的任何信息。例如，以下是一个有效的内容映射：

```
{
  "info": {
    "description": "My first API with Amazon API Gateway."
  },
  "x-custom-info" : "My custom info, recognized by OpenAPI.",
  "my-info" : "My custom info not recognized by OpenAPI."
}
```

虽然 API Gateway 接受将任意有效 JSON 字符串作为内容映射，但内容属性仍分为两类：可以被 OpenAPI 识别的属性以及不可被 OpenAPI 识别的属性。在上一示例中，`info`、`description` 和 `x-custom-info` 均被 OpenAPI 识别为标准 OpenAPI 对象、属性或扩展。相比之下，`my-info` 不符合 OpenAPI 规范。API Gateway 会将符合 OpenAPI 规范的内容属性从关联的 `DocumentationPart` 实例传播到 API 实体定义中。API Gateway 不会将不符合规范的内容属性传播到 API 实体定义中。

再给一个示例，以下是面向 `DocumentationPart` 实体的 `Resource`：

```
{
    "location" : {
        "type" : "RESOURCE",
        "path": "/pets"
    },
    "properties" : {
        "summary" : "The /pets resource represents a collection of pets in PetStore.",
        "description": "... a child resource under the root...",
    }
}
```

在这里，`type` 和 `path` 均为有效字段，用于识别 `RESOURCE` 类型的目标。对于根资源 (`/`)，您可以省略 `path` 字段。

```
{
    "location" : {
        "type" : "RESOURCE"
    },
    "properties" : {
        "description" : "The root resource with the default path specification."
    }
}
```

以下 `DocumentationPart` 实例也是如此：

```
{
    "location" : {
        "type" : "RESOURCE",
        "path": "/"
    },
    "properties" : {
        "description" : "The root resource with an explicit path specification"
    }
}
```

从包含更通用规格的 API 实体中继承内容

可选 `location` 字段的默认值提供了 API 实体的模式化描述。使用 `location` 对象中的默认值，您可以通过这类 `properties` 模式，在到 `DocumentationPart` 实例的 `location` 映射中添加一般性描述。API Gateway 从通用 API 实体的 `DocumentationPart` 中提取适用的 OpenAPI 文档属性，并通过与一般 `location` 模式匹配或与精确值匹配的 `location` 字段将其注入特定 API 实体中，除非特定实体已拥有与其关联的 `DocumentationPart` 实例。此行为也称为从包含更通用规格的 API 实体中继承内容。

内容继承不适用于某些 API 实体类型。有关详细信息，请参阅下表。

当 API 实体与多个 `DocumentationPart` 的位置模式匹配时，该实体将继承具有最高优先级和特定性的位置字段的文档部分。优先顺序是 `path` > `statusCode`。为与 `path` 字段匹配，API Gateway 会选择拥有最具体的路径值的实体。下表通过几个示例展示了这一点。

案例	**path**	**statusCc e**	**name**	备注
1	`/pets`	`*`	`id`	与此位置模式关联的文档将由与该位置模式匹配的实体继承。
2	`/pets`	`200`	`id`	当案例 1 和案例 2 相匹配时，由于案例 2 比案例 1 更具体，与此位置模式关

案例	`path`	`statusCo` `e`	`name`	备注
				联的文档将由与该位置模式匹配的实体继承。

案例	path	statusCo e	name	备注
3	/pets/petId	*	id	当案例1、2和案例3匹配时，由于案例3的优先级比案例2高，并且比案例1更具体，与此位置模式关联的文档将由与该位置模式匹配的实

案例	**path**	**statusCc e**	**name**	备注
				体继承。

以下是另一个示例，它对比了更通用的 `DocumentationPart` 实例与更具体的实例。以下一般性错误消息 `"Invalid request error"` 已注入 `400` 错误响应的 OpenAPI 定义中（除非被覆盖）。

```
{
    "location" : {
        "type" : "RESPONSE",
        "statusCode": "400"
    },
    "properties" : {
        "description" : "Invalid request error."
    }"
}
```

通过以下覆盖，对 `400` 资源上任何方法的 `/pets` 响应都会改为 `"Invalid petId specified"` 这一描述。

```
{
    "location" : {
        "type" : "RESPONSE",
        "path": "/pets",
        "statusCode": "400"
    },
    "properties" : "{
        "description" : "Invalid petId specified."
    }"
}
```

`DocumentationPart` 的有效位置字段

下表显示与给定 API 实体类型关联的 DocumentationPart 资源的有效及必需字段和适用的默认值。

API 实体	有效位置字段	必需位置字段	默认字段值	是否为可继承的内容
API		type	不适用	否
资源		type	path 的默认值为 /。	否
方法		type	path 和 method 的默认值分别为 / 和 *。	是，按前缀匹配 path，匹配任何值的 method。
查询参数		type	path 和 method 的默认值分别为 / 和 *。	是，按前缀匹配 path，按精确值匹配 method。

API

```
{
    "location": {
        "type": "API"
    },
    ...
}
```

资源

```
{
    "location": {
        "type": "RESOURCE
",
        "path":
 "resource_path "
    },
    ...
}
```

方法

```
{
    "location": {
        "type":
 "METHOD",
        "path":
 "resource_path ",
        "method":
 "http_verb "
    },
    ...
}
```

查询参数

```
{
    "location": {
        "type": "QUERY_PA
RAMETER",
        "path":
 "resource_path ",
        "method":
 "HTTP_verb ",
```

API 实体	有效位置字段	必需位置字段	默认字段值	是否为可继承的内容
	（代码见下方）			
请求正文	（代码见下方）	type	path 和 method 的默认值分别为 / 和 *。	是，按前缀匹配 path，按精确值匹配 method。
请求标头参数	（代码见下方）	type, name	path 和 method 的默认值分别为 / 和 *。	是，按前缀匹配 path，按精确值匹配 method。

（续）

```
            "name": "query_parameter_name "
        },
        ...
}
```

请求正文

```
{
    "location": {
        "type": "REQUEST_BODY",
        "path": "resource_path ",
        "method": "http_verb "
    },
    ...
}
```

请求标头参数

```
{
    "location": {
        "type": "REQUEST_HEADER",
        "path": "resource_path ",
        "method": "HTTP_verb ",
        "name": "header_name "
    },
    ...
}
```

API 实体	有效位置字段	必需位置字段	默认字段值	是否为可继承的内容
请求路径参数	见下方代码	type, name	path 和 method 的默认值分别为 / 和 *。	是，按前缀匹配 path，按精确值匹配 method。
响应	见下方代码	type	path、method 和 statusCode 的默认值分别为 /、* 和 *。	是，按前缀匹配 path，按精确值匹配 method 和 statusCode 。

请求路径参数

```
{
    "location": {
        "type": "PATH_PARAMETER",
        "path": "resource/{path_parameter_name }",
        "method": "HTTP_verb ",
        "name": "path_parameter_name "
    },
    ...
}
```

响应

```
{
    "location": {
        "type": "RESPONSE",
        "path": "resource_path ",
        "method": "http_verb ",
        "statusCode": "status_code "
    },
    ...
}
```

API 实体	有效位置字段	必需位置字段	默认字段值	是否为可继承的内容
响应标头		type, name	path、method 和 statusCode 的默认值分别为 /、* 和 *。	是，按前缀匹配 path，按精确值匹配 method 和 statusCode 。
响应正文		type	path、method 和 statusCode 的默认值分别为 /、* 和 *。	是，按前缀匹配 path，按精确值匹配 method 和 statusCode 。

响应标头

```
{
    "location": {
        "type": "RESPONSE
_HEADER",
        "path":
 "resource_path ",
        "method":
 "http_verb ",
        "statusCode":
 "status_code ",
        "name":
 "header_name "
    },
    ...
}
```

响应正文

```
{
    "location": {
        "type": "RESPONSE
_BODY",
        "path":
 "resource_path ",
        "method":
 "http_verb ",
        "statusCode":
 "status_code "
    },
    ...
}
```

<table>
<tr><th>API 实体</th><th>有效位置字段</th><th>必需位置字段</th><th>默认字段值</th><th>是否为可继承的内容</th></tr>
<tr><td>授权方</td><td><pre>{
    "location": {
        "type": "AUTHORIZER",
        "name": "authorizer_name "
    },
    ...
}</pre></td><td><code>type</code></td><td>不适用</td><td>否</td></tr>
<tr><td>模型</td><td><pre>{
    "location": {
        "type": "MODEL",
        "name": "model_name "
    },
    ...
}</pre></td><td><code>type</code></td><td>不适用</td><td>否</td></tr>
</table>

文档版本

文档版本是 API 的 DocumentationParts 集合的快照，标记有版本标识符。发布 API 文档的过程涉及创建一个文档版本，将其与一个 API 阶段关联，并将特定于阶段的 API 文档版本导出为外部 OpenAPI 文件。在 API Gateway 中，文档快照表示为 DocumentationVersion 资源。

当您更新 API 时，您便创建了 API 的新版本。在 API Gateway 中，您使用 DocumentationVersions 集合维护所有文档版本。

使用 API Gateway 控制台记录 API

在本节中，我们将介绍如何使用 API Gateway 控制台创建和维护 API 的文档部分。

创建和编辑 API 文档的一个先决条件是您必须已创建 API。在本节中，我们使用 PetStore API 作为示例。要使用 API Gateway 控制台创建 API，请按照 教程：通过导入示例创建 REST API中的说明操作。

主题

- 记录 API 实体
- 记录 RESOURCE 实体
- 记录 METHOD 实体
- 记录 QUERY_PARAMETER 实体
- 记录 PATH_PARAMETER 实体
- 记录 REQUEST_HEADER 实体
- 记录 REQUEST_BODY 实体
- 记录 RESPONSE 实体
- 记录 RESPONSE_HEADER 实体
- 记录 RESPONSE_BODY 实体
- 记录 MODEL 实体
- 记录 AUTHORIZER 实体

记录 `API` 实体

要为 `API` 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择文档，然后选择创建文档部分。
2. 对于文档类型，请选择 API。

   如果尚未为 `API` 创建文档部分，您将看到文档部分的 `properties` 映射编辑器。在文本编辑器中输入以下 `properties` 映射。

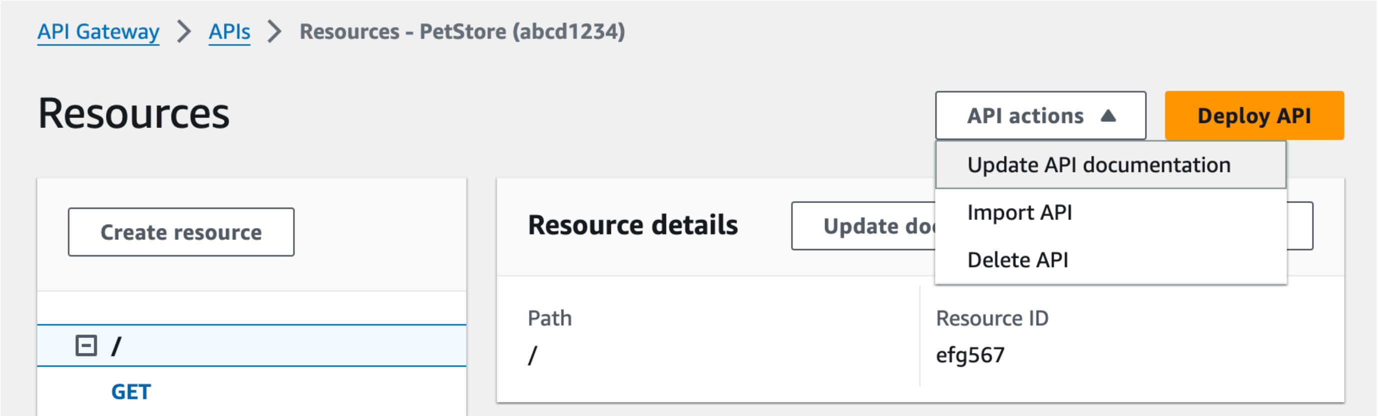
   ```
   {
     "info": {
       "description": "Your first API Gateway API.",
       "contact": {
           "name": "John Doe",
           "email": "john.doe@api.com"
       }
     }
   }
   ```

> **Note**
>
> 您无需将 `properties` 映射编码成 JSON 字符串。API Gateway 控制台会对 JSON 对象进行字符串化。

3. 选择创建文档部分。

要在资源窗格中为 API 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择资源。
2. 选择 API 操作菜单，然后选择更新 API 文档。



要编辑现有文档部分，请执行以下操作：

1. 在文档窗格中，选择资源和方法选项卡。
2. 选择 API 的名称，然后在 API 卡片上选择编辑。

记录 **RESOURCE** 实体

要为 RESOURCE 实体添加新的文档部分，请执行以下操作：

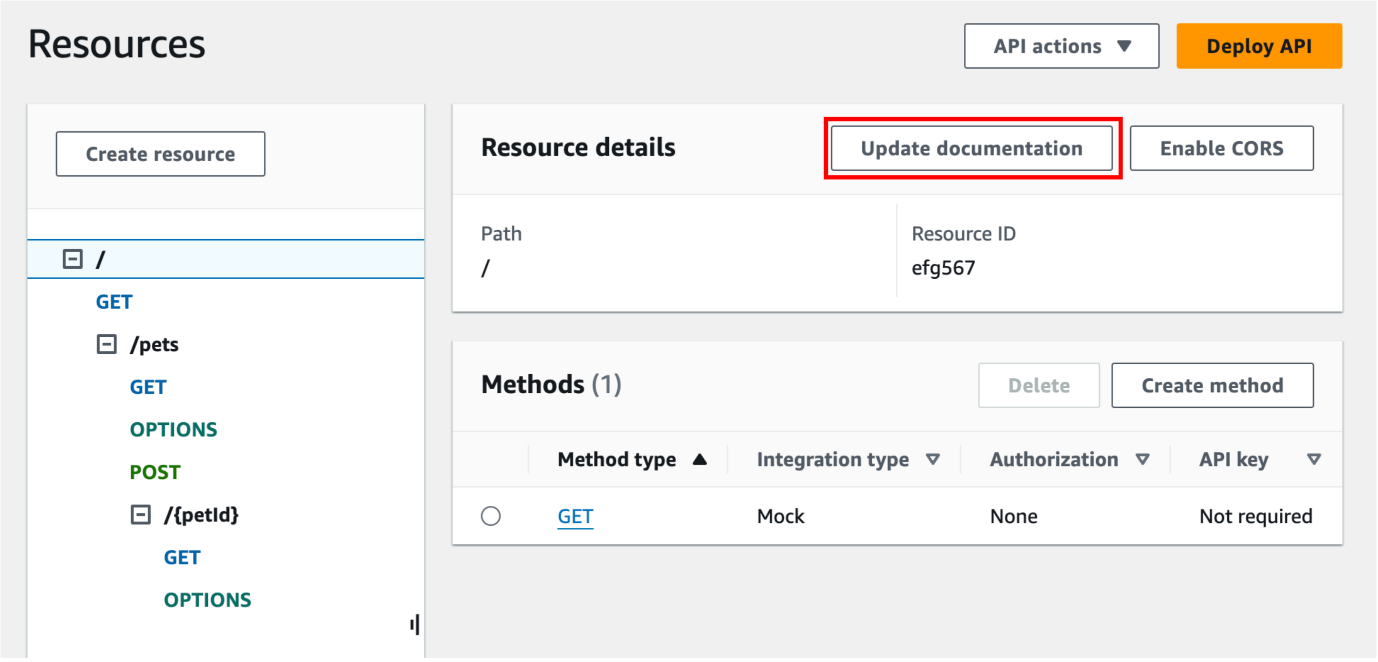
1. 在主导航窗格中，选择文档，然后选择创建文档部分。
2. 对于文档类型，选择资源。
3. 对于路径，输入路径。
4. 在文本编辑器中输入描述，例如：

```
{
    "description": "The PetStore's root resource."
}
```

5. 选择创建文档部分。您可以为未列出的资源创建文档。
6. 如果需要，请重复上述步骤，以添加或编辑另一个文档部分。

要在资源窗格中为 RESOURCE 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择资源。
2. 选择资源，然后选择更新文档。



Path
/

Resource ID
efg567

	Method type ▲	Integration type ▽	Authorization ▽	API key ▽
○	GET	Mock	None	Not required

要编辑现有文档部分，请执行以下操作：

1. 在文档窗格中，选择资源和方法选项卡。
2. 选择包含文档部分的资源，然后选择编辑。

记录 **METHOD** 实体

要为 METHOD 实体添加新的文档部分，请执行以下操作：

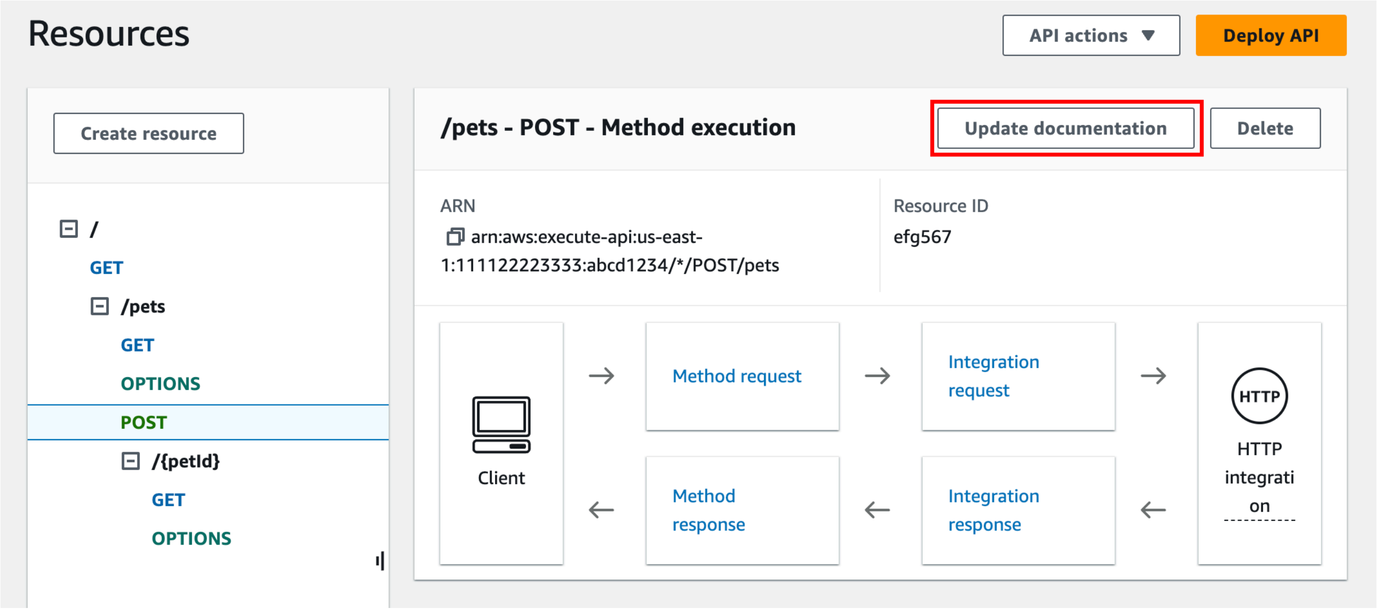
1. 在主导航窗格中，选择文档，然后选择创建文档部分。
2. 对于文档类型，选择方法。
3. 对于路径，输入路径。
4. 对于方法，选择 HTTP 动词。
5. 在文本编辑器中输入描述，例如：

```
{
  "tags" : [ "pets" ],
  "summary" : "List all pets"
}
```

6. 选择创建文档部分。您可以为未列出的方法创建文档。
7. 如果需要，请重复上述步骤，以添加或编辑另一个文档部分。

要在资源窗格中为 `METHOD` 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择资源。
2. 选择方法，然后选择更新文档。



ARN
arn:aws:execute-api:us-east-1:111122223333:abcd1234/*/POST/pets

Resource ID
efg567

要编辑现有文档部分，请执行以下操作：

1. 在文档窗格中，选择资源和方法选项卡。

2. 您可以选择方法或选择包含该方法的资源，然后使用搜索栏查找并选择您的文档部分。
3. 选择编辑。

记录 **QUERY_PARAMETER** 实体

要为 `QUERY_PARAMETER` 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择文档，然后选择创建文档部分。
2. 对于文档类型，选择查询参数。
3. 对于路径，输入路径。
4. 对于方法，选择 HTTP 动词。
5. 对于名称，输入名称。
6. 在文本编辑器中输入描述。
7. 选择创建文档部分。您可以为未列出的查询参数创建文档。
8. 如果需要，请重复上述步骤，以添加或编辑另一个文档部分。

要编辑现有文档部分，请执行以下操作：

1. 在文档窗格中，选择资源和方法选项卡。
2. 您可以选择查询参数或选择包含查询参数的资源，然后使用搜索栏查找和选择您的文档部分。
3. 选择编辑。

记录 **PATH_PARAMETER** 实体

要为 `PATH_PARAMETER` 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择文档，然后选择创建文档部分。
2. 对于文档类型，选择路径参数。
3. 对于路径，输入路径。
4. 对于方法，选择 HTTP 动词。
5. 对于名称，输入名称。
6. 在文本编辑器中输入描述。
7. 选择创建文档部分。您可以为未列出的路径参数创建文档。

8. 如果需要，请重复上述步骤，以添加或编辑另一个文档部分。

要编辑现有文档部分，请执行以下操作：

1. 在文档窗格中，选择资源和方法选项卡。
2. 您可以选择路径参数或选择包含路径参数的资源，然后使用搜索栏查找并选择您的文档部分。
3. 选择编辑。

记录 **REQUEST_HEADER** 实体

要为 `REQUEST_HEADER` 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择文档，然后选择创建文档部分。
2. 对于文档类型，选择请求标头。
3. 对于路径，输入请求标头的路径。
4. 对于方法，选择 HTTP 动词。
5. 对于名称，输入名称。
6. 在文本编辑器中输入描述。
7. 选择创建文档部分。您可以为未列出的请求标头创建文档。
8. 如果需要，请重复上述步骤，以添加或编辑另一个文档部分。

要编辑现有文档部分，请执行以下操作：

1. 在文档窗格中，选择资源和方法选项卡。
2. 您可以选择请求标头或选择包含请求标头的资源，然后使用搜索栏查找并选择您的文档部分。
3. 选择编辑。

记录 **REQUEST_BODY** 实体

要为 `REQUEST_BODY` 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择文档，然后选择创建文档部分。
2. 对于文档类型，选择请求正文。
3. 对于路径，输入请求正文的路径。

4. 对于方法，选择 HTTP 动词。
5. 在文本编辑器中输入描述。
6. 选择创建文档部分。您可以为未列出的请求正文创建文档。
7. 如果需要，请重复上述步骤，以添加或编辑另一个文档部分。

要编辑现有文档部分，请执行以下操作：

1. 在文档窗格中，选择资源和方法选项卡。
2. 您可以选择请求正文或选择包含请求正文的资源，然后使用搜索栏查找并选择您的文档部分。
3. 选择编辑。

记录 **RESPONSE** 实体

要为 `RESPONSE` 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择文档，然后选择创建文档部分。
2. 对于文档类型，选择响应(状态代码)。
3. 对于路径，输入响应的路径。
4. 对于方法，选择 HTTP 动词。
5. 对于状态代码，输入 HTTP 状态代码。
6. 在文本编辑器中输入描述。
7. 选择创建文档部分。您可以为未列出的响应状态代码创建文档。
8. 如果需要，请重复上述步骤，以添加或编辑另一个文档部分。

要编辑现有文档部分，请执行以下操作：

1. 在文档窗格中，选择资源和方法选项卡。
2. 您可以选择响应状态代码或选择包含响应状态代码的资源，然后使用搜索栏查找并选择您的文档部分。
3. 选择编辑。

记录 **RESPONSE_HEADER** 实体

要为 `RESPONSE_HEADER` 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择文档，然后选择创建文档部分。
2. 对于文档类型，选择响应标头。
3. 对于路径，输入响应标头的路径。
4. 对于方法，选择 HTTP 动词。
5. 对于状态代码，输入 HTTP 状态代码。
6. 在文本编辑器中输入描述。
7. 选择创建文档部分。您可以为未列出的响应标头创建文档。
8. 如果需要，请重复上述步骤，以添加或编辑另一个文档部分。

要编辑现有文档部分，请执行以下操作：

1. 在文档窗格中，选择资源和方法选项卡。
2. 您可以选择响应标头或选择包含响应标头的资源，然后使用搜索栏查找并选择您的文档部分。
3. 选择编辑。

记录 **`RESPONSE_BODY`** 实体

要为 `RESPONSE_BODY` 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择文档，然后选择创建文档部分。
2. 对于文档类型，选择响应正文。
3. 对于路径，输入响应正文的路径。
4. 对于方法，选择 HTTP 动词。
5. 对于状态代码，输入 HTTP 状态代码。
6. 在文本编辑器中输入描述。
7. 选择创建文档部分。您可以为未列出的响应正文创建文档。
8. 如果需要，请重复上述步骤，以添加或编辑另一个文档部分。

要编辑现有文档部分，请执行以下操作：

1. 在文档窗格中，选择资源和方法选项卡。
2. 您可以选择响应正文或选择包含响应正文的资源，然后使用搜索栏查找并选择您的文档部分。
3. 选择编辑。

记录 **MODEL** 实体

记录 `MODEL` 实体的过程涉及创建和管理模型的 `DocumentPart` 实例以及每个模型的 `properties`。例如，对于每个 API 默认附带的 `Error` 模型，它包含以下架构定义，

```
{
  "$schema" : "http://json-schema.org/draft-04/schema#",
  "title" : "Error Schema",
  "type" : "object",
  "properties" : {
    "message" : { "type" : "string" }
  }
}
```

且需要两个 `DocumentationPart` 实例，其中一个用于 `Model`，另一个用于其 `message` 属性：

```
{
  "location": {
    "type": "MODEL",
    "name": "Error"
  },
  "properties": {
    "title": "Error Schema",
    "description": "A description of the Error model"
  }
}
```

和

```
{
  "location": {
    "type": "MODEL",
    "name": "Error.message"
  },
  "properties": {
    "description": "An error message."
  }
}
```

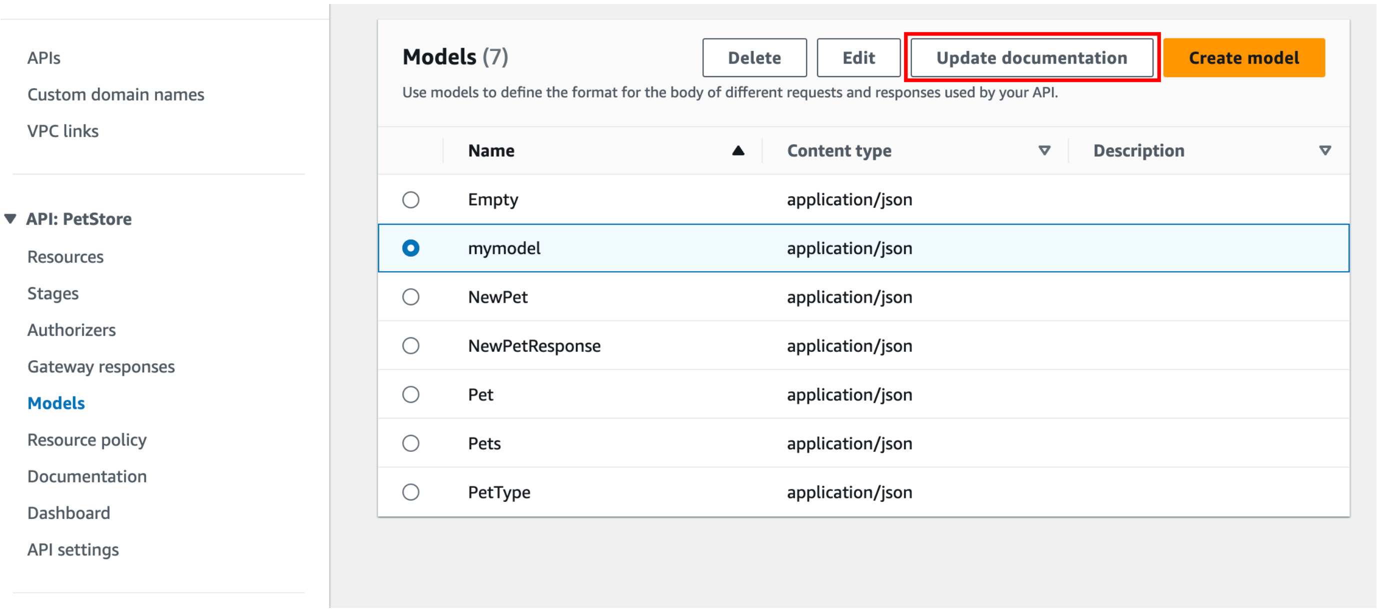
导出 API 后，`DocumentationPart` 的属性将覆盖原始架构中的值。

要为 `MODEL` 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择文档，然后选择创建文档部分。
2. 对于文档类型，选择模型。
3. 对于名称，输入模型的名称。
4. 在文本编辑器中输入描述。
5. 选择创建文档部分。您可以为未列出的模型创建文档。
6. 如果需要，请重复上述步骤，以将文档部分添加到其他模型，或编辑文档部分。

要在模型窗格中为 `MODEL` 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择模型。
2. 选择模型，然后选择更新文档。



Use models to define the format for the body of different requests and responses used by your API.

	Name	Content type	Description
○	Empty	application/json	
●	mymodel	application/json	
○	NewPet	application/json	
○	NewPetResponse	application/json	
○	Pet	application/json	
○	Pets	application/json	
○	PetType	application/json	

要编辑现有文档部分，请执行以下操作：

1. 在文档窗格中，选择模型选项卡。
2. 使用搜索栏或选择模型，然后选择编辑。

记录 `AUTHORIZER` 实体

要为 `AUTHORIZER` 实体添加新的文档部分，请执行以下操作：

1. 在主导航窗格中，选择文档，然后选择创建文档部分。

2. 对于文档类型，选择授权方。
3. 对于名称，输入授权方的名称。
4. 在文本编辑器中输入描述。为授权方的有效 `location` 字段指定一个值。
5. 选择创建文档部分。您可以为未列出的授权方创建文档。
6. 如果需要，请重复上述步骤，以将文档部分添加到其他授权方，或编辑文档部分。

要编辑现有文档部分，请执行以下操作：

1. 在文档窗格中，选择授权方选项卡。
2. 使用搜索栏或选择授权方，然后选择编辑。

使用 API Gateway 控制台发布 API 文档

以下过程介绍了如何发布文档版本。

使用 API Gateway 控制台发布文档版本

1. 在主导航窗格中，选择文档。
2. 选择发布文档。
3. 设置发布：
   a. 对于阶段，选择一个阶段。
   b. 对于版本，输入版本标识符，例如 `1.0.0`。
   c. （可选）对于描述，输入描述。
4. 选择 Publish。

现在，您可以通过将文档导出为外部 OpenAPI 文件来继续下载已发布的文档。要了解更多信息，请参阅“the section called “导出 REST API””。

使用 API Gateway REST API 记录 API

在本节中，我们将介绍如何使用 API Gateway REST API 创建和维护 API 的文档部分。

在创建和编辑 API 的文档之前，请先创建 API。在本节中，我们使用 PetStore API 作为示例。要使用 API Gateway 控制台创建 API，请按照 教程：通过导入示例创建 REST API中的说明操作。

主题

- 记录 API 实体
- 记录 RESOURCE 实体
- 记录 METHOD 实体
- 记录 QUERY_PARAMETER 实体
- 记录 PATH_PARAMETER 实体
- 记录 REQUEST_BODY 实体
- 记录 REQUEST_HEADER 实体
- 记录 RESPONSE 实体
- 记录 RESPONSE_HEADER 实体
- 记录 AUTHORIZER 实体
- 记录 MODEL 实体
- 更新文档部分
- 列出文档部分

记录 `API` 实体

要为 API 添加文档，请为 API 实体添加 DocumentationPart 资源：

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "API"
    },
    "properties": "{\n\t\"info\": {\n\t\t\"description\" : \"Your first API with Amazon
 API Gateway.\"\n\t}\n}"
}
```

如果成功，操作将返回 `201 Created` 响应，其中的有效载荷包含新创建的 `DocumentationPart` 实例。例如：

```
{
  ...
  "id": "s2e5xf",
  "location": {
    "path": null,
    "method": null,
    "name": null,
    "statusCode": null,
    "type": "API"
  },
  "properties": "{\n\t\"info\": {\n\t\t\"description\" : \"Your first API with Amazon
 API Gateway.\"\n\t}\n}"
}
```

如果已添加文档部分，则将返回 `409 Conflict` 响应，其中包含错误消息 `Documentation part already exists for the specified location: type 'API'."`。在这种情况下，您必须调用 documentationpart:update 操作。

```
PATCH /restapis/4wk1k4onj3/documentation/parts/part_id HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
  "patchOperations" : [ {
    "op" : "replace",
    "path" : "/properties",
    "value" : "{\n\t\"info\": {\n\t\t\"description\" : \"Your first API with Amazon API
 Gateway.\"\n\t}\n}"
  } ]
}
```

成功的响应返回 `200 OK` 状态代码，其中的有效载荷包含更新后的 `DocumentationPart` 实例。

记录 RESOURCE 实体

要为 API 的根资源添加文档，请添加面向对应的 Resource 资源的 DocumentationPart 资源：

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
```

```
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
       "type" : "RESOURCE",
    },
    "properties" : "{\n\t\"description\" : \"The PetStore root resource.\"\n}"
}
```

如果成功，操作将返回 201 Created 响应，其中的有效载荷包含新创建的 DocumentationPart 实例。例如：

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/p76vqo"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/p76vqo"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/p76vqo"
    }
  },
  "id": "p76vqo",
  "location": {
    "path": "/",
    "method": null,
    "name": null,
    "statusCode": null,
    "type": "RESOURCE"
  },
```

```
  "properties": "{\n\t\"description\" : \"The PetStore root resource.\"\n}"
}
```

如果未指定资源路径，则该资源将作为根资源。您可以将 `"path": "/"` 添加到 `properties`，以明确规范。

要为 API 的子资源创建文档，请添加面向对应的 Resource 资源的 DocumentationPart 资源：

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "RESOURCE",
         "path" : "/pets"
    },
    "properties": "{\n\t\"description\" : \"A child resource under the root of
 PetStore.\"\n}"
}
```

如果成功，操作将返回 `201 Created` 响应，其中的有效载荷包含新创建的 `DocumentationPart` 实例。例如：

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/qcht86"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/qcht86"
    },
```

```
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/qcht86"
    }
  },
  "id": "qcht86",
  "location": {
    "path": "/pets",
    "method": null,
    "name": null,
    "statusCode": null,
    "type": "RESOURCE"
  },
  "properties": "{\n\t\"description\" : \"A child resource under the root of PetStore.
\"\n}"
}
```

要为由路径参数指定的子资源添加文档，请添加面向 [Resource](#) 资源的 [DocumentationPart](#) 资源：

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "RESOURCE",
         "path" : "/pets/{petId}"
    },
    "properties": "{\n\t\"description\" : \"A child resource specified by the petId
 path parameter.\"\n}"
}
```

如果成功，操作将返回 `201 Created` 响应，其中的有效载荷包含新创建的 `DocumentationPart` 实例。例如：

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
```

```
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/k6fpwb"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/k6fpwb"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/k6fpwb"
    }
  },
  "id": "k6fpwb",
  "location": {
    "path": "/pets/{petId}",
    "method": null,
    "name": null,
    "statusCode": null,
    "type": "RESOURCE"
  },
  "properties": "{\n\t\"description\" : \"A child resource specified by the petId path
 parameter.\"\n}"
}
```

> **Note**
>
> RESOURCE 实体的 DocumentationPart 实例无法由其任意子资源继承。

记录 **METHOD** 实体

要为 API 的方法添加文档，请添加面向对应的 Method 资源的 DocumentationPart 资源：

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

```
{
    "location" : {
         "type" : "METHOD",
         "path" : "/pets",
         "method" : "GET"
    },
    "properties": "{\n\t\"summary\" : \"List all pets.\"\n}"
}
```

如果成功，操作将返回 `201 Created` 响应，其中的有效载荷包含新创建的 DocumentationPart 实例。例如：

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    }
  },
  "id": "o64jbj",
  "location": {
    "path": "/pets",
    "method": "GET",
    "name": null,
    "statusCode": null,
    "type": "METHOD"
  },
  "properties": "{\n\t\"summary\" : \"List all pets.\"\n}"
}
```

如果成功，操作将返回 `201 Created` 响应，其中的有效载荷包含新创建的 DocumentationPart 实例。例如：

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    }
  },
  "id": "o64jbj",
  "location": {
    "path": "/pets",
    "method": "GET",
    "name": null,
    "statusCode": null,
    "type": "METHOD"
  },
  "properties": "{\n\t\"summary\" : \"List all pets.\"\n}"
}
```

如果未在前面的请求中指定 `location.method` 字段，则将其假定为以通配符 ANY 字符表示的 * 方法。

要更新 METHOD 实体的文档内容，请调用 documentationpart:update 操作，并提供新的 `properties` 映射：

```
PATCH /restapis/4wk1k4onj3/documentation/parts/part_id HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

```
{
  "patchOperations" : [ {
    "op" : "replace",
    "path" : "/properties",
    "value" : "{\n\t\"tags\" : [ \"pets\" ], \n\t\"summary\" : \"List all pets.\"\n}"
  } ]
}
```

成功的响应返回 `200 OK` 状态代码，其中的有效载荷包含更新后的 `DocumentationPart` 实例。例如：

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/o64jbj"
    }
  },
  "id": "o64jbj",
  "location": {
    "path": "/pets",
    "method": "GET",
    "name": null,
    "statusCode": null,
    "type": "METHOD"
  },
  "properties": "{\n\t\"tags\" : [ \"pets\" ], \n\t\"summary\" : \"List all pets.\"\n}"
}
```

记录 **QUERY_PARAMETER** 实体

要为请求查询参数添加文档，请添加面向 QUERY_PARAMETER 类型的 DocumentationPart 资源以及 path 和 name 的有效字段。

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "QUERY_PARAMETER",
         "path" : "/pets",
         "method" : "GET",
         "name" : "page"
    },
    "properties": "{\n\t\"description\" : \"Page number of results to return.\"\n}"
}
```

如果成功，操作将返回 201 Created 响应，其中的有效载荷包含新创建的 DocumentationPart 实例。例如：

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/h9ht5w"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/h9ht5w"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/h9ht5w"
    }
```

```
  },
  "id": "h9ht5w",
  "location": {
    "path": "/pets",
    "method": "GET",
    "name": "page",
    "statusCode": null,
    "type": "QUERY_PARAMETER"
  },
  "properties": "{\n\t\"description\" : \"Page number of results to return.\"\n}"
}
```

`properties` 实体文档部分的 QUERY_PARAMETER 映射可由它的一个子 QUERY_PARAMETER 实体继承。例如，如果您在 `treats` 之后添加 `/pets/{petId}` 资源，对 GET 启用 `/pets/{petId}/treats` 方法，并呈现 page 查询参数，则此子查询参数将从 `DocumentationPart` 方法名称类似的查询参数中继承 `properties` 的 GET `/pets` 映射，除非您将 `DocumentationPart` 资源明确添加到 page 方法的 GET `/pets/{petId}/treats` 查询参数中。

记录 **PATH_PARAMETER** 实体

要为路径参数添加文档，请为 PATH_PARAMETER 实体添加 DocumentationPart 资源。

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "PATH_PARAMETER",
         "path" : "/pets/{petId}",
         "method" : "*",
         "name" : "petId"
    },
    "properties": "{\n\t\"description\" : \"The id of the pet to retrieve.\"\n}"
}
```

如果成功，操作将返回 `201 Created` 响应，其中的有效载荷包含新创建的 `DocumentationPart` 实例。例如：

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/ckpgog"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/ckpgog"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/ckpgog"
    }
  },
  "id": "ckpgog",
  "location": {
    "path": "/pets/{petId}",
    "method": "*",
    "name": "petId",
    "statusCode": null,
    "type": "PATH_PARAMETER"
  },
  "properties": "{\n  \"description\" : \"The id of the pet to retrieve\"\n}"
}
```

记录 **REQUEST_BODY** 实体

要为请求正文添加文档，请为请求正文添加 DocumentationPart 资源。

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
```

```
          "type" : "REQUEST_BODY",
          "path" : "/pets",
          "method" : "POST"
     },
     "properties": "{\n\t\"description\" : \"A Pet object to be added to PetStore.\"\n}"
}
```

如果成功，操作将返回 201 Created 响应，其中的有效载荷包含新创建的 DocumentationPart 实例。例如：

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/kgmfr1"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/kgmfr1"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/kgmfr1"
    }
  },
  "id": "kgmfr1",
  "location": {
    "path": "/pets",
    "method": "POST",
    "name": null,
    "statusCode": null,
    "type": "REQUEST_BODY"
  },
  "properties": "{\n\t\"description\" : \"A Pet object to be added to PetStore.\"\n}"
}
```

记录 **REQUEST_HEADER** 实体

要为请求标头添加文档，请为请求标头添加 DocumentationPart 资源。

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "REQUEST_HEADER",
         "path" : "/pets",
         "method" : "GET",
         "name" : "x-my-token"
    },
    "properties": "{\n\t\"description\" : \"A custom token used to authorization the
 method invocation.\"\n}"
}
```

如果成功，操作将返回 `201 Created` 响应，其中的有效载荷包含新创建的 `DocumentationPart` 实例。例如：

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/h0m3uf"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/h0m3uf"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/h0m3uf"
    }
  },
  "id": "h0m3uf",
  "location": {
    "path": "/pets",
```

```
    "method": "GET",
    "name": "x-my-token",
    "statusCode": null,
    "type": "REQUEST_HEADER"
  },
  "properties": "{\n\t\"description\" : \"A custom token used to authorization the
 method invocation.\"\n}"
}
```

记录 RESPONSE 实体

要为状态代码的响应添加文档，请添加面向对应的 MethodResponse 资源的 DocumentationPart 资源。

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location": {
      "path": "/",
      "method": "*",
      "name": null,
      "statusCode": "200",
      "type": "RESPONSE"
    },
    "properties": "{\n  \"description\" : \"Successful operation.\"\n}"
}
```

如果成功，操作将返回 `201 Created` 响应，其中的有效载荷包含新创建的 `DocumentationPart` 实例。例如：

```
{
    "_links": {
        "self": {
            "href": "/restapis/4wk1k4onj3/documentation/parts/lattew"
        },
        "documentationpart:delete": {
```

```
            "href": "/restapis/4wk1k4onj3/documentation/parts/lattew"
        },
        "documentationpart:update": {
            "href": "/restapis/4wk1k4onj3/documentation/parts/lattew"
        }
    },
    "id": "lattew",
    "location": {
        "path": "/",
        "method": "*",
        "name": null,
        "statusCode": "200",
        "type": "RESPONSE"
    },
    "properties": "{\n  \"description\" : \"Successful operation.\"\n}"
}
```

记录 **RESPONSE_HEADER** 实体

要为响应标头添加文档，请为响应标头添加 DocumentationPart 资源。

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

  "location": {
    "path": "/",
    "method": "GET",
    "name": "Content-Type",
    "statusCode": "200",
    "type": "RESPONSE_HEADER"
  },
  "properties": "{\n  \"description\" : \"Media type of request\"\n}"
```

如果成功，操作将返回 `201 Created` 响应，其中的有效载荷包含新创建的 `DocumentationPart` 实例。例如：

```
{
  "_links": {
```

```
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/fev7j7"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/fev7j7"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/fev7j7"
    }
  },
  "id": "fev7j7",
  "location": {
    "path": "/",
    "method": "GET",
    "name": "Content-Type",
    "statusCode": "200",
    "type": "RESPONSE_HEADER"
  },
  "properties": "{\n  \"description\" : \"Media type of request\"\n}"
}
```

`Content-Type` 响应标头的文档是 API 任意响应的 `Content-Type` 标头的默认文档。

记录 `AUTHORIZER` 实体

要为 API 授权方添加文档，请添加面向指定授权方的 DocumentationPart 资源。

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "AUTHORIZER",
```

```
            "name" : "myAuthorizer"
    },
    "properties": "{\n\t\"description\" : \"Authorizes invocations of configured
  methods.\"\n}"
}
```

如果成功，操作将返回 201 Created 响应，其中的有效载荷包含新创建的 DocumentationPart 实例。例如：

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/pw3qw3"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/pw3qw3"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/pw3qw3"
    }
  },
  "id": "pw3qw3",
  "location": {
    "path": null,
    "method": null,
    "name": "myAuthorizer",
    "statusCode": null,
    "type": "AUTHORIZER"
  },
  "properties": "{\n\t\"description\" : \"Authorizes invocations of configured methods.
\"\n}"
}
```

> **Note**
>
> AUTHORIZER 实体的 DocumentationPart 实例无法由其任意子资源继承。

记录 `MODEL` 实体

记录 `MODEL` 实体的过程涉及创建和管理模型的 `DocumentPart` 实例以及每个模型的 `properties`。例如，对于每个 API 默认附带的 `Error` 模型，它包含以下架构定义，

```
{
  "$schema" : "http://json-schema.org/draft-04/schema#",
  "title" : "Error Schema",
  "type" : "object",
  "properties" : {
    "message" : { "type" : "string" }
  }
}
```

且需要两个 `DocumentationPart` 实例，其中一个用于 `Model`，另一个用于其 `message` 属性：

```
{
  "location": {
    "type": "MODEL",
    "name": "Error"
  },
  "properties": {
    "title": "Error Schema",
    "description": "A description of the Error model"
  }
}
```

和

```
{
  "location": {
    "type": "MODEL",
    "name": "Error.message"
  },
  "properties": {
    "description": "An error message."
  }
}
```

导出 API 后，`DocumentationPart` 的属性将覆盖原始架构中的值。

要为 API 模型添加文档，请添加面向指定模型的 DocumentationPart 资源。

```
POST /restapis/restapi_id/documentation/parts HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "location" : {
         "type" : "MODEL",
         "name" : "Pet"
    },
    "properties": "{\n\t\"description\" : \"Data structure of a Pet object.\"\n}"
}
```

如果成功，操作将返回 `201 Created` 响应，其中的有效载荷包含新创建的 DocumentationPart 实例。例如：

```
{
  "_links": {
    "curies": {
      "href": "http://docs.aws.amazon.com/apigateway/latest/developerguide/restapi-
documentationpart-{rel}.html",
      "name": "documentationpart",
      "templated": true
    },
    "self": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/lkn4uq"
    },
    "documentationpart:delete": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/lkn4uq"
    },
    "documentationpart:update": {
      "href": "/restapis/4wk1k4onj3/documentation/parts/lkn4uq"
    }
  },
  "id": "lkn4uq",
  "location": {
    "path": null,
    "method": null,
    "name": "Pet",
    "statusCode": null,
```

```
    "type": "MODEL"
  },
  "properties": "{\n\t\"description\" : \"Data structure of a Pet object.\"\n}"
}
```

重复同一步骤，为任意模型的属性创建 DocumentationPart 实例。

> **Note**
>
> `MODEL` 实体的 DocumentationPart 实例无法由其任意子资源继承。

更新文档部分

要更新任何一类 API 实体的文档部分，请对具有指定部分标识符的 DocumentationPart 实例提交 PATCH 请求，以将现有的 `properties` 映射替换为新的映射。

```
PATCH /restapis/4wk1k4onj3/documentation/parts/part_id HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
  "patchOperations" : [ {
    "op" : "replace",
    "path" : "RESOURCE_PATH",
    "value" : "NEW_properties_VALUE_AS_JSON_STRING"
  } ]
}
```

成功的响应返回 `200 OK` 状态代码，其中的有效载荷包含更新后的 `DocumentationPart` 实例。

您可以在单个 `PATCH` 请求中更新多个文档部分。

列出文档部分

要列出任何一类 API 实体的文档部分，请对 DocumentationParts 集合提交 GET 请求。

```
GET /restapis/restapi_id/documentation/parts HTTP/1.1
```

```
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

成功的响应返回 `200 OK` 状态代码，其中的有效载荷包含可用的 `DocumentationPart` 实例。

使用 API Gateway REST API 发布 API 文档

要发布 API 文档，请创建、更新或获取文档快照，然后将该文档快照与一个 API 阶段关联。创建文档快照时，您也可以同时将其与一个 API 阶段关联。

主题

- 创建文档快照并将其与一个 API 阶段关联
- 创建文档快照
- 更新文档快照
- 获取文档快照
- 将文档快照与一个 API 阶段关联
- 下载与阶段关联的文档快照

创建文档快照并将其与一个 API 阶段关联

要创建 API 文档部分的快照，并同时将其与一个 API 阶段关联，请提交以下 `POST` 请求：

```
POST /restapis/restapi_id/documentation/versions HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "documentationVersion" : "1.0.0",
    "stageName": "prod",
    "description" : "My API Documentation v1.0.0"
}
```

如果成功，操作将返回 200 OK 响应，其中包含新创建的 DocumentationVersion 实例作为有效载荷。

或者，您可以先创建一个文档快照，但不将其与一个 API 阶段关联，然后调用 restapi:update，以便将该快照与指定的 API 阶段关联。您也可以更新或查询现有的文档快照，然后更新其阶段关联。我们将在接下来的四个部分中展示相关步骤。

创建文档快照

要创建 API 文档部分的快照，请创建新的 DocumentationVersion 资源，并将其添加到 API 的 DocumentationVersions 集合中：

```
POST /restapis/restapi_id/documentation/versions HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "documentationVersion" : "1.0.0",
    "description" : "My API Documentation v1.0.0"
}
```

如果成功，操作将返回 200 OK 响应，其中包含新创建的 DocumentationVersion 实例作为有效载荷。

更新文档快照

您只需修改对应的 DocumentationVersion 资源的 description 属性，即可更新文档快照。以下示例展示了如何更新由版本标识符 *version* (例如 1.0.0) 进行标识的文档快照的描述。

```
PATCH /restapis/restapi_id/documentation/versions/version HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "patchOperations": [{
```

```
        "op": "replace",
        "path": "/description",
        "value": "My API for testing purposes."
    }]
}
```

如果成功，操作将返回 `200 OK` 响应，其中包含更新后的 `DocumentationVersion` 实例作为有效载荷。

获取文档快照

要获取文档快照，请提交针对指定 DocumentationVersion 资源的 `GET` 请求。以下示例展示了如何获取给定版本标识符 (1.0.0) 的文档快照。

```
GET /restapis/<restapi_id>/documentation/versions/1.0.0 HTTP/1.1
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

将文档快照与一个 API 阶段关联

要发布 API 文档，请将文档快照与一个 API 阶段关联。您必须创建 API 阶段，然后才能将文档版本与该阶段关联。

要使用 API Gateway REST API 将文档快照与一个 API 阶段关联，请调用 stage:update 操作，以在 `stage.documentationVersion` 属性上设置所需的文档版本：

```
PATCH /restapis/RESTAPI_ID/stages/STAGE_NAME
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
    "patchOperations": [{
        "op": "replace",
        "path": "/documentationVersion",
        "value": "VERSION_IDENTIFIER"
```

```
    }]
}
```

下载与阶段关联的文档快照

在某一版本的文档部分与某个阶段相关联后，您可以使用 API Gateway 控制台、API Gateway REST API、某种开发工具包或适用于 API Gateway 的 AWS CLI，将文档部分与 API 实体定义导出到外部文件。该过程与导出 API 的过程一样。导出的文件格式可以是 JSON 或 YAML。

使用 API Gateway REST API，还可以明确设置 `extension=documentation,integrations,authorizers` 查询参数，以将 API 文档部分、API 集成和授权方包含在 API 导出中。默认情况下，在导出 API 时会包含文档部分，但不包括集成和授权方。API 导出的默认输出适用于文档的分发。

要使用 API Gateway REST API 将 API 文档导出到外部 JSON OpenAPI 文件中，请提交以下 GET 请求：

```
GET /restapis/restapi_id/stages/stage_name/exports/swagger?extensions=documentation
 HTTP/1.1
Accept: application/json
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

在这里，`x-amazon-apigateway-documentation` 对象包含文档部分，而 API 实体定义包含 OpenAPI 支持的文档属性。输出并不包含集成或 Lambda 授权方（以前称为自定义授权方）的详细信息。要包含这两项详细信息，请设置 `extensions=integrations,authorizers,documentation`。要包含集成的详细信息，但不包含授权方的详细信息，请设置 `extensions=integrations,documentation`。

您必须在请求中设置 `Accept:application/json` 标头，才能以 JSON 文件格式输出结果。要生成 YAML 输出，请将请求标头更改为 `Accept:application/yaml`。

例如，我们来查看一个在根资源 (GET) 上显示简单 / 方法的 API。此 API 拥有四个在 OpenAPI 定义文件中定义的 API 实体，其类型分别为 `API`、`MODEL`、`METHOD` 和 `RESPONSE`。每个 `API`、`METHOD` 和 `RESPONSE` 实体都已添加文档部分。通过调用前面的 documentation-exporting 命令，我们将获得以下输出，而文档部分将作为标准 OpenAPI 文件的扩展列在 `x-amazon-apigateway-documentation` 对象中。

OpenAPI 3.0

```
{
   "openapi": "3.0.0",
   "info": {
      "description": "API info description",
      "version": "2016-11-22T22:39:14Z",
      "title": "doc",
      "x-bar": "API info x-bar"
   },
   "paths": {
      "/": {
         "get": {
            "description": "Method description.",
            "responses": {
               "200": {
                  "description": "200 response",
                  "content": {
                     "application/json": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
                        }
                     }
                  }
               }
            },
            "x-example": "x- Method example"
         },
         "x-bar": "resource x-bar"
      }
   },
   "x-amazon-apigateway-documentation": {
      "version": "1.0.0",
      "createdDate": "2016-11-22T22:41:40Z",
      "documentationParts": [
         {
            "location": {
               "type": "API"
            },
            "properties": {
               "description": "API description",
               "foo": "API foo",
               "x-bar": "API x-bar",
               "info": {
```

```
                "description": "API info description",
                "version": "API info version",
                "foo": "API info foo",
                "x-bar": "API info x-bar"
             }
          }
       },
       {
          "location": {
             "type": "METHOD",
             "method": "GET"
          },
          "properties": {
             "description": "Method description.",
             "x-example": "x- Method example",
             "foo": "Method foo",
             "info": {
                "version": "method info version",
                "description": "method info description",
                "foo": "method info foo"
             }
          }
       },
       {
          "location": {
             "type": "RESOURCE"
          },
          "properties": {
             "description": "resource description",
             "foo": "resource foo",
             "x-bar": "resource x-bar",
             "info": {
                "description": "resource info description",
                "version": "resource info version",
                "foo": "resource info foo",
                "x-bar": "resource info x-bar"
             }
          }
       }
    ]
 },
 "x-bar": "API x-bar",
 "servers": [
    {
```

```
         "url": "https://rznaap68yi.execute-api.ap-southeast-1.amazonaws.com/
{basePath}",
         "variables": {
            "basePath": {
              "default": "/test"
            }
         }
      }
   ],
   "components": {
      "schemas": {
         "Empty": {
            "type": "object",
            "title": "Empty Schema"
         }
      }
   }
}
```

OpenAPI 2.0

```
{
  "swagger" : "2.0",
  "info" : {
    "description" : "API info description",
    "version" : "2016-11-22T22:39:14Z",
    "title" : "doc",
    "x-bar" : "API info x-bar"
  },
  "host" : "rznaap68yi.execute-api.ap-southeast-1.amazonaws.com",
  "basePath" : "/test",
  "schemes" : [ "https" ],
  "paths" : {
    "/" : {
      "get" : {
        "description" : "Method description.",
        "produces" : [ "application/json" ],
        "responses" : {
          "200" : {
            "description" : "200 response",
            "schema" : {
              "$ref" : "#/definitions/Empty"
            }
```

```
          }
        },
        "x-example" : "x- Method example"
      },
      "x-bar" : "resource x-bar"
    }
  },
  "definitions" : {
    "Empty" : {
      "type" : "object",
      "title" : "Empty Schema"
    }
  },
  "x-amazon-apigateway-documentation" : {
    "version" : "1.0.0",
    "createdDate" : "2016-11-22T22:41:40Z",
    "documentationParts" : [ {
      "location" : {
        "type" : "API"
      },
      "properties" : {
        "description" : "API description",
        "foo" : "API foo",
        "x-bar" : "API x-bar",
        "info" : {
          "description" : "API info description",
          "version" : "API info version",
          "foo" : "API info foo",
          "x-bar" : "API info x-bar"
        }
      }
    }, {
      "location" : {
        "type" : "METHOD",
        "method" : "GET"
      },
      "properties" : {
        "description" : "Method description.",
        "x-example" : "x- Method example",
        "foo" : "Method foo",
        "info" : {
          "version" : "method info version",
          "description" : "method info description",
          "foo" : "method info foo"
```

```
        }
      }
    }, {
      "location" : {
        "type" : "RESOURCE"
      },
      "properties" : {
        "description" : "resource description",
        "foo" : "resource foo",
        "x-bar" : "resource x-bar",
        "info" : {
          "description" : "resource info description",
          "version" : "resource info version",
          "foo" : "resource info foo",
          "x-bar" : "resource info x-bar"
        }
      }
    } ]
  },
  "x-bar" : "API x-bar"
}
```

对于在文档部分的 `properties` 映射中定义且符合 OpenAPI 规范的属性，API Gateway 会将该属性插入关联的 API 实体定义中。`x-something` 的一个属性是标准 OpenAPI 扩展。此扩展将传播到 API 实体定义中。例如，请查看 `x-example` 方法的 `GET` 属性。类似 `foo` 的属性不属于 OpenAPI 规范，也不会插入到关联的 API 实体定义中。

如果某个文档呈现工具（例如 OpenAPI UI）通过解析 API 实体定义来提取文档属性，则该工具将无法使用 `properties` 实例所有不符合 OpenAPI 规范的 `DocumentationPart` 属性。但是，如果某个文档呈现工具通过解析 `x-amazon-apigateway-documentation` 对象来获取内容，或者通过调用 restapi:documentation-parts 和 documenationpart:by-id 从 API Gateway 中检索文档部分，则该工具可以展示所有文档属性。

要将具备 API 实体定义（包含集成详细信息）的文档导出为 JSON OpenAPI 文件，请提交以下 `GET` 请求：

```
GET /restapis/restapi_id/stages/stage_name/exports/swagger?
extensions=integrations,documentation HTTP/1.1
Accept: application/json
Host: apigateway.region.amazonaws.com
```

```
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

要将具备 API 实体定义（包含集成和授权方的详细信息）的文档导出为 YAML OpenAPI 文件，请提交以下 `GET` 请求：

```
GET /restapis/restapi_id/stages/stage_name/exports/swagger?
extensions=integrations,authorizers,documentation HTTP/1.1
Accept: application/yaml
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret
```

要使用 API Gateway 控制台导出和下载 API 的已发布文档，请按照使用 API Gateway 控制台导出 REST API 中的说明操作。

导入 API 文档

与导入 API 实体定义一样，您可以在 API Gateway 中将文档部分从外部 OpenAPI 文件导入到 API 中。您需要在有效的 OpenAPI 定义文件中指定 x-amazon-apigateway-documentation 对象 扩展中待导入的文档部分。导入文档的过程不会更改现有的 API 实体定义。

您可以选择在 API Gateway 中将新指定的文档部分合并到现有的文档部分中，也可以覆盖现有的文档部分。在 `MERGE` 模式下，在 OpenAPI 文件中定义的新文档部分会添加到 API 的 `DocumentationParts` 集合中。如果存在已导入的 `DocumentationPart`，在两者属性不同的情况下，导入的属性将替代现有属性。其他现有的文档属性不受影响。在 `OVERWRITE` 模式中，整个 `DocumentationParts` 集合将被替代，具体取决于导入的 OpenAPI 定义文件。

使用 API Gateway REST API 导入文档部分

要使用 API Gateway REST API 导入 API 文档，请调用 documentationpart:import 操作。以下示例显示了如何使用单个 `GET /` 方法覆盖 API 的现有文档部分，并在成功后返回 `200 OK` 响应。

OpenAPI 3.0

```
PUT /restapis/<restapi_id>/documentation/parts&mode=overwrite&failonwarnings=true
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
   "openapi": "3.0.0",
   "info": {
      "description": "description",
      "version": "1",
      "title": "doc"
   },
   "paths": {
      "/": {
         "get": {
            "description": "Method description.",
            "responses": {
               "200": {
                  "description": "200 response",
                  "content": {
                     "application/json": {
                        "schema": {
                           "$ref": "#/components/schemas/Empty"
                        }
                     }
                  }
               }
            }
         }
      }
   },
   "x-amazon-apigateway-documentation": {
      "version": "1.0.3",
      "documentationParts": [
         {
            "location": {
               "type": "API"
            },
            "properties": {
```

```
                "description": "API description",
                "info": {
                   "description": "API info description 4",
                   "version": "API info version 3"
                }
             }
          },
          {
             "location": {
                "type": "METHOD",
                "method": "GET"
             },
             "properties": {
                "description": "Method description."
             }
          },
          {
             "location": {
                "type": "MODEL",
                "name": "Empty"
             },
             "properties": {
                "title": "Empty Schema"
             }
          },
          {
             "location": {
                "type": "RESPONSE",
                "method": "GET",
                "statusCode": "200"
             },
             "properties": {
                "description": "200 response"
             }
          }
       ]
    },
    "servers": [
       {
          "url": "/"
       }
    ],
    "components": {
       "schemas": {
```

```
          "Empty": {
             "type": "object",
             "title": "Empty Schema"
          }
       }
    }
}
```

OpenAPI 2.0

```
PUT /restapis/<restapi_id>/documentation/parts&mode=overwrite&failonwarnings=true
Host: apigateway.region.amazonaws.com
Content-Type: application/json
X-Amz-Date: YYYYMMDDTttttttZ
Authorization: AWS4-HMAC-SHA256 Credential=access_key_id/YYYYMMDD/region/
apigateway/aws4_request, SignedHeaders=content-length;content-type;host;x-amz-date,
 Signature=sigv4_secret

{
  "swagger": "2.0",
  "info": {
    "description": "description",
    "version": "1",
    "title": "doc"
  },
  "host": "",
  "basePath": "/",
  "schemes": [
    "https"
  ],
  "paths": {
    "/": {
      "get": {
        "description": "Method description.",
        "produces": [
          "application/json"
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Empty"
            }
```

```
          }
        }
      }
    }
  },
  "definitions": {
    "Empty": {
      "type": "object",
      "title": "Empty Schema"
    }
  },
  "x-amazon-apigateway-documentation": {
    "version": "1.0.3",
    "documentationParts": [
      {
        "location": {
          "type": "API"
        },
        "properties": {
          "description": "API description",
          "info": {
            "description": "API info description 4",
            "version": "API info version 3"
          }
        }
      },
      {
        "location": {
          "type": "METHOD",
          "method": "GET"
        },
        "properties": {
          "description": "Method description."
        }
      },
      {
        "location": {
          "type": "MODEL",
          "name": "Empty"
        },
        "properties": {
          "title": "Empty Schema"
        }
      },
```

```
      {
        "location": {
          "type": "RESPONSE",
          "method": "GET",
          "statusCode": "200"
        },
        "properties": {
          "description": "200 response"
        }
      }
    ]
  }
}
```

如果成功，此请求将返回一个 200 OK 响应，并在其负载中包含导入的 `DocumentationPartId`。

```
{
  "ids": [
    "kg3mth",
    "796rtf",
    "zhek4p",
    "5ukm9s"
  ]
}
```

另外，您也可以调用 restapi:import 或 restapi:put，并在 `x-amazon-apigateway-documentation` 对象中提供文档部分，作为 API 定义的输入 OpenAPI 文件的一部分。要从 API 导入中排除文档部分，请在请求查询参数中设置 `ignore=documentation`。

使用 API Gateway 控制台导入文档部分

以下说明介绍了如何导入文档部分。

使用控制台从外部文件导入 API 文档部分的步骤

1. 在主导航窗格中，选择文档。
2. 选择 Import（导入）。
3. 如果您已有文档，请选择覆盖或合并新文档。
4. 选择选择文件以从驱动器加载文件，或将文件内容输入到文件视图中。有关示例，请参阅使用 API Gateway REST API 导入文档部分中示例请求的负载。

5. 选择如何处理导入时的警告。选择警告时失败或忽略警告。有关更多信息，请参阅 the section called “导入期间的错误和警告”。
6. 选择 Import。

控制对 API 文档的访问权限

如果您在撰写和编辑 API 文档方面有专门的文档团队，您可以为开发人员 (API 开发) 以及撰稿者或编辑 (内容开发) 配置单独的访问权限。这尤其适合第三方供应商参与文档创建的情况。

要向您的文档团队授予创建、更新和发布 API 文档的访问权限，您可以使用以下 IAM 策略为文档团队分配一个 IAM 角色，其中 *account_id* 是文档团队的AWS账户 ID。

```
{
  "Version": "2012-10-17",
  "Statement": [

    {
      "Sid": "StmtDocPartsAddEditViewDelete",
      "Effect": "Allow",
      "Action": [
        "apigateway:GET",
        "apigateway:PUT",
        "apigateway:POST",
        "apigateway:PATCH",
        "apigateway:DELETE"
      ],
      "Resource": [
        "arn:aws:apigateway::account_id:/restapis/*/documentation/*"
      ]
    }
  ]
}
```

有关设置对于 API Gateway 资源的访问权限的信息，请参阅 the section called “Amazon API Gateway 如何与 IAM 配合使用”。

在 API Gateway 中为 REST API 生成开发工具包

要以特定于平台或特定于语言的方式调用您的 REST API，您必须为 API 生成特定于平台或语言的软件开发工具包。在创建、测试 API 并将其部署到某个阶段后，您将生成 SDK。目前，API Gateway 支

持使用 Java、JavaScript、Java for Android，以及 Objective-C for iOS 或 Swift for iOS 以及 Ruby 为 API 生成开发工具包。

本节介绍如何生成 API Gateway API 的开发工具包。其中还演示如何在 Java 应用程序、Java for Android 应用程序、Objective-C for iOS 应用程序、Swift for iOS 应用程序以及 JavaScript 应用程序中使用生成的开发工具包。

为了便于讨论，我们将使用此 API Gateway API，它公开了这个简单计算器 Lambda 函数。

继续操作之前，请在 API Gateway 中创建或导入 API 并至少部署一次。有关说明，请参阅 在 Amazon API Gateway 中部署 REST API。

主题

- 简单的计算器 Lambda 函数
- API Gateway 中的简单计算器 API
- 简单计算器 API OpenAPI 定义
- 生成 API 的 Java 开发工具包
- 生成 API 的 Android 开发工具包
- 生成 API 的 iOS 开发工具包
- 生成 REST API 的 JavaScript 开发工具包
- 生成 API 的 Ruby 开发工具包
- 使用 AWS CLI 命令为 API 生成开发工具包

简单的计算器 Lambda 函数

我们将使用一个执行加、减、乘、除二进制运算的 Node.js Lambda 函数作为示例。

主题

- 简单计算器 Lambda 函数的输入格式
- 简单计算器 Lambda 函数的输出格式
- 简单计算器 Lambda 函数的实施

简单计算器 Lambda 函数的输入格式

此函数使用以下格式的输入：

```
{ "a": "Number", "b": "Number", "op": "string"}
```

其中 op 可以是任意 (`+, -, *, /, add, sub, mul, div`)。

简单计算器 Lambda 函数的输出格式

如果运算成功，则返回以下格式的结果：

```
{ "a": "Number", "b": "Number", "op": "string", "c": "Number"}
```

其中 c 对应着计算结果。

简单计算器 Lambda 函数的实施

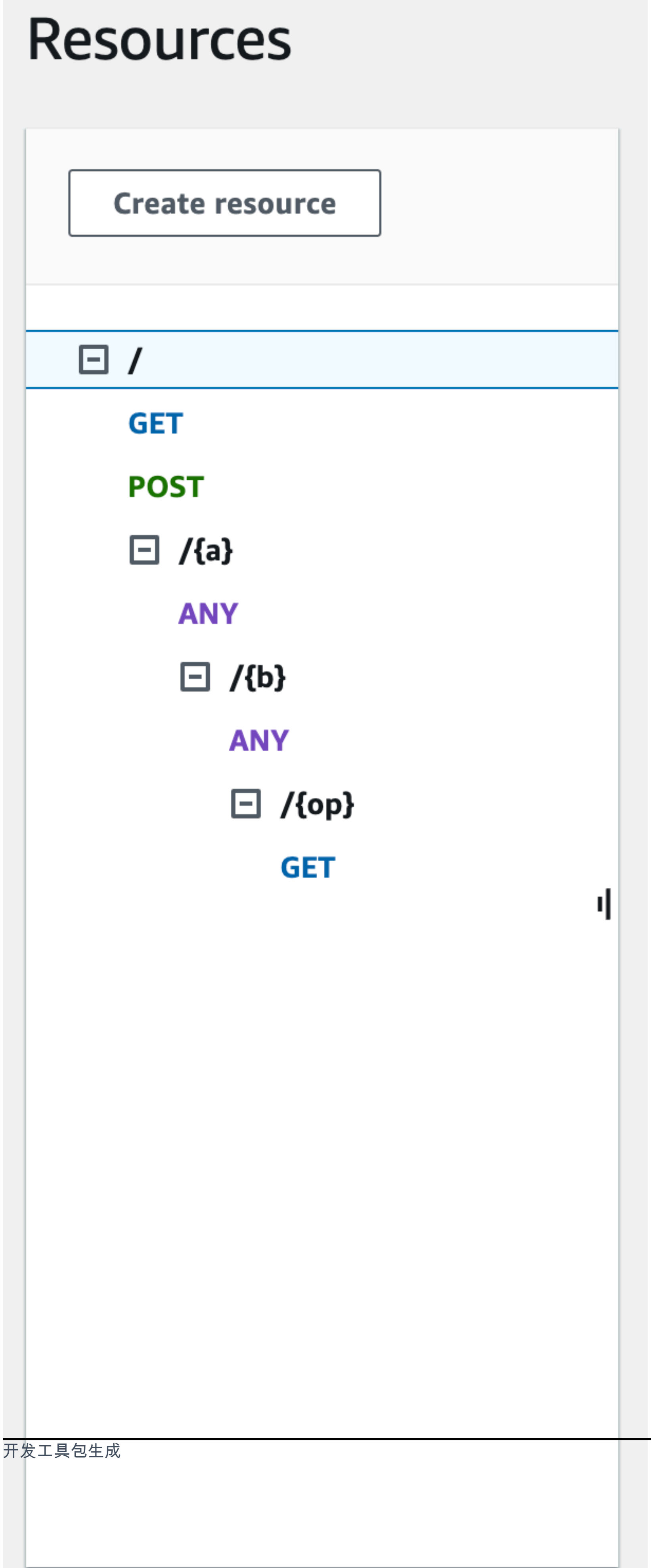
Lambda 函数的实施方式如下：

```
export const handler = async function (event, context) {
  console.log("Received event:", JSON.stringify(event));

  if (
    event.a === undefined ||
    event.b === undefined ||
    event.op === undefined
  ) {
    return "400 Invalid Input";
  }

  const res = {};
  res.a = Number(event.a);
  res.b = Number(event.b);
  res.op = event.op;
  if (isNaN(event.a) || isNaN(event.b)) {
    return "400 Invalid Operand";
  }
  switch (event.op) {
    case "+":
    case "add":
      res.c = res.a + res.b;
      break;
    case "-":
    case "sub":
      res.c = res.a - res.b;
```

```
      break;
    case "*":
    case "mul":
      res.c = res.a * res.b;
      break;
    case "/":
    case "div":
      if (res.b == 0) {
        return "400 Divide by Zero";
      } else {
        res.c = res.a / res.b;
      }
      break;
    default:
      return "400 Invalid Operator";
  }

  return res;
};
```

API Gateway 中的简单计算器 API

我们的简单计算器 API 公开了三种方法（GET、POST、GET）来调用 the section called “简单的计算器 Lambda 函数”。此 API 的图形表示如下：



Resources
Create resource
/
GET
POST
/{a}
ANY
/{b}
ANY
/{op}
GET

这三种方法显示为后端 Lambda 函数提供输入来执行相同操作的不同方式：

- `GET /?a=...&b=...&op=...` 方法使用查询参数来指定输入。
- `POST /` 方法使用 JSON 负载 `{"a":"Number", "b":"Number", "op":"string"}` 来指定输入。
- `GET /{a}/{b}/{op}` 方法使用路径参数来指定输入。

如果未定义，API Gateway 通过将 HTTP 方法和路径部分组合起来，生成对应的开发工具包方法名称。根路径部分 (/) 称为 `Api Root`。例如，API 方法 `GET /?a=...&b=...&op=...` 的默认 Java 开发工具包方法名称为 `getABOp`，`POST /` 的默认开发工具包方法名称为 `postApiRoot`，`GET /{a}/{b}/{op}` 的默认开发工具包方法名称为 `getABOp`。单独的开发工具包可以自定义约定。对于开发工具包特定的方法名称，请参考生成的开发工具包源中的文档。

您可以并且应该通过在各个 API 方法上指定 operationName 属性来覆盖默认开发工具包方法名称。您可在使用 API Gateway REST API 创建 API 方法或更新 API 方法时这样做。在 API Swagger 定义中，您可以设置 `operationId` 以实现相同结果。

在演示如何使用 API Gateway 为该 API 生成的开发工具包来调用这些方法之前，我们先简单回想一下如何对其进行设置。有关详细说明，请参阅在 API Gateway 中开发 REST API。如果您是 API Gateway 的新用户，请先参阅选择 AWS Lambda 集成教程。

创建用于输入和输出的模型

为了在 SDK 中指定强类型输入，我们为该 API 创建了一个 `Input` 模型。为了描述响应正文数据类型，我们创建了一个 `Output` 模型和一个 `Result` 模型。

创建用于输入、输出和结果的模型

1. 在主导航窗格中，选择模型。
2. 选择创建模型。
3. 对于名称，请输入 **`input`**。
4. 对于内容类型，输入 **`application/json`**。

   如果未找到匹配的内容类型，则不执行请求验证。要使用同一模型而不考虑内容类型，请输入 **`$default`**。
5. 对于模型架构，输入以下模型：

```
{
```

```
    "$schema" : "$schema": "http://json-schema.org/draft-04/schema#",
    "type":"object",
    "properties":{
        "a":{"type":"number"},
        "b":{"type":"number"},
        "op":{"type":"string"}
    },
    "title":"Input"
}
```

6. 选择创建模型。
7. 重复以下步骤以创建 `Output` 模型和 `Result` 模型。

   对于 `Output` 模型，为模型架构输入以下内容：

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type": "object",
    "properties": {
        "c": {"type":"number"}
    },
    "title": "Output"
}
```

   对于 `Result` 模型，为模型架构输入以下内容。将 API ID `abc123` 替换为您的 API ID。

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "type":"object",
    "properties":{
        "input":{
            "$ref":"https://apigateway.amazonaws.com/restapis/abc123/models/Input"
        },
        "output":{
            "$ref":"https://apigateway.amazonaws.com/restapis/abc123/models/Output"
        }
    },
    "title":"Result"
}
```

设置 GET/方法查询参数

对于 `GET /?a=..&b=..&op=..` 方法，查询参数在 Method Request (方法请求) 中声明：

设置 GET / URL 查询字符串参数

1. 在方法请求部分，为根 (/) 资源上的 GET 方法选择编辑。
2. 选择 URL 查询字符串参数并执行以下操作：
   a. 选择添加查询字符串。
   b. 在名称中，输入 **a**。
   c. 保持必填和缓存为已关闭状态。
   d. 将缓存保持为关闭状态。

   重复相同的步骤，创建一个名为 **b** 的查询字符串和一个名为 **op** 的查询字符串。
3. 选择保存。

将负载的数据模型设置为后端的输入

对于 `POST /` 方法，我们创建了 `Input` 模型，并将其添加到方法请求中以定义输入数据的形状。

将负载的数据模型设置为后端的输入

1. 在方法请求部分，为根 (/) 资源上的 POST 方法选择编辑。
2. 选择请求正文。
3. 选择添加模型。
4. 对于内容类型，输入 **application/json**。
5. 对于模型，选择输入。
6. 选择保存。

使用此模型，您的 API 客户可以通过实例化 `Input` 对象来调用开发工具包以指定输入。若不使用此模型，您的客户将需要创建词典对象来表示 Lambda 函数的 JSON 输入。

为后端的结果输出设置数据模型

对于所有三种方法，我们将创建 `Result` 模型并将其添加到方法的 `Method Response` 中，以定义 Lambda 函数返回的输出的形状。

为后端的结果输出设置数据模型

1. 选择 /{a}/{b}/{op} 资源，然后选择 GET 方法。
2. 在方法响应选项卡的响应 200 下，选择编辑。
3. 在响应正文下，选择添加模型。
4. 对于内容类型，输入 **`application/json`**。
5. 对于模型，选择结果。
6. 选择保存。

使用此模型，您的 API 客户可以通过读取 `Result` 对象的属性来解析成功的输出。若不使用此模型，您的客户将需要创建词典对象来表示 JSON 输出。

简单计算器 API OpenAPI 定义

以下是简单计算器 API 的 OpenAPI 定义。您可以将其导入到您的账户中。但是，您需要在导入后对 Lambda 函数重置基于资源的权限。要执行此操作，请从 API Gateway 控制台中的集成请求重新选择您在账户中创建的 Lambda 函数。这会让 API Gateway 控制台重置所需的权限。或者，您可以使用 AWS Command Line Interface 来执行 add-permission 的 Lambda 命令。

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "info": {
    "version": "2016-09-29T20:27:30Z",
    "title": "SimpleCalc"
  },
  "host": "t6dve4zn25.execute-api.us-west-2.amazonaws.com",
  "basePath": "/demo",
  "schemes": [
    "https"
  ],
  "paths": {
    "/": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
```

```
        ],
        "parameters": [
          {
            "name": "op",
            "in": "query",
            "required": false,
            "type": "string"
          },
          {
            "name": "a",
            "in": "query",
            "required": false,
            "type": "string"
          },
          {
            "name": "b",
            "in": "query",
            "required": false,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Result"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "requestTemplates": {
            "application/json": "#set($inputRoot = $input.path('$'))\n{\n
  \"a\" : $input.params('a'),\n  \"b\" : $input.params('b'),\n  \"op\" :
 \"$input.params('op')\"\n}"
          },
          "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "passthroughBehavior": "when_no_templates",
          "httpMethod": "POST",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseTemplates": {
```

```
                "application/json": "#set($inputRoot = $input.path('$'))\n{\n
 \"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
 \"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "type": "aws"
        }
      },
      "post": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "in": "body",
            "name": "Input",
            "required": true,
            "schema": {
              "$ref": "#/definitions/Input"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Result"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "passthroughBehavior": "when_no_match",
          "httpMethod": "POST",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseTemplates": {
```

```
                "application/json": "#set($inputRoot = $input.path('$'))\n{\n
\"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
\"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "type": "aws"
        }
      }
    },
    "/{a}": {
      "x-amazon-apigateway-any-method": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "a",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "404": {
            "description": "404 response"
          }
        },
        "x-amazon-apigateway-integration": {
          "requestTemplates": {
            "application/json": "{\"statusCode\": 200}"
          },
          "passthroughBehavior": "when_no_match",
          "responses": {
            "default": {
              "statusCode": "404",
              "responseTemplates": {
                "application/json": "{ \"Message\" : \"Can't $context.httpMethod
$context.resourcePath\" }"
              }
            }
```

```
          },
          "type": "mock"
        }
      }
    },
    "/{a}/{b}": {
      "x-amazon-apigateway-any-method": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "a",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "b",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "404": {
            "description": "404 response"
          }
        },
        "x-amazon-apigateway-integration": {
          "requestTemplates": {
            "application/json": "{\"statusCode\": 200}"
          },
          "passthroughBehavior": "when_no_match",
          "responses": {
            "default": {
              "statusCode": "404",
              "responseTemplates": {
                "application/json": "{ \"Message\" : \"Can't $context.httpMethod
  $context.resourcePath\" }"
              }
```

```
            }
          },
          "type": "mock"
        }
      }
    },
    "/{a}/{b}/{op}": {
      "get": {
        "consumes": [
          "application/json"
        ],
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "a",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "b",
            "in": "path",
            "required": true,
            "type": "string"
          },
          {
            "name": "op",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "schema": {
              "$ref": "#/definitions/Result"
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "requestTemplates": {
```

```
            "application/json": "#set($inputRoot = $input.path('$'))\n{\n
  \"a\" : $input.params('a'),\n  \"b\" : $input.params('b'),\n  \"op\" :
 \"$input.params('op')\"\n}"
          },
          "uri": "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:Calc/invocations",
          "passthroughBehavior": "when_no_templates",
          "httpMethod": "POST",
          "responses": {
            "default": {
              "statusCode": "200",
              "responseTemplates": {
                "application/json": "#set($inputRoot = $input.path('$'))\n{\n
 \"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
 \"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "type": "aws"
        }
      }
    }
  },
  "definitions": {
    "Input": {
      "type": "object",
      "properties": {
        "a": {
          "type": "number"
        },
        "b": {
          "type": "number"
        },
        "op": {
          "type": "string"
        }
      },
      "title": "Input"
    },
    "Output": {
      "type": "object",
      "properties": {
        "c": {
          "type": "number"
```

```
        }
      },
      "title": "Output"
    },
    "Result": {
      "type": "object",
      "properties": {
        "input": {
          "$ref": "#/definitions/Input"
        },
        "output": {
          "$ref": "#/definitions/Output"
        }
      },
      "title": "Result"
    }
  }
}
```

OpenAPI 3.0

```
{
  "openapi" : "3.0.1",
  "info" : {
    "title" : "SimpleCalc",
    "version" : "2016-09-29T20:27:30Z"
  },
  "servers" : [ {
    "url" : "https://t6dve4zn25.execute-api.us-west-2.amazonaws.com/{basePath}",
    "variables" : {
      "basePath" : {
        "default" : "demo"
      }
    }
  } ],
  "paths" : {
    "/{a}/{b}" : {
      "x-amazon-apigateway-any-method" : {
        "parameters" : [ {
          "name" : "a",
          "in" : "path",
          "required" : true,
          "schema" : {
```

```
            "type" : "string"
          }
        }, {
          "name" : "b",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "404" : {
            "description" : "404 response",
            "content" : { }
          }
        },
        "x-amazon-apigateway-integration" : {
          "type" : "mock",
          "responses" : {
            "default" : {
              "statusCode" : "404",
              "responseTemplates" : {
                "application/json" : "{ \"Message\" : \"Can't $context.httpMethod 
$context.resourcePath\" }"
              }
            }
          },
          "requestTemplates" : {
            "application/json" : "{\"statusCode\": 200}"
          },
          "passthroughBehavior" : "when_no_match"
        }
      }
    },
    "/{a}/{b}/{op}" : {
      "get" : {
        "parameters" : [ {
          "name" : "a",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
```

```
          "name" : "b",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "op",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "200" : {
            "description" : "200 response",
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Result"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
          "type" : "aws",
          "httpMethod" : "POST",
          "uri" : "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:111122223333:function:Calc/invocations",
          "responses" : {
            "default" : {
              "statusCode" : "200",
              "responseTemplates" : {
                "application/json" : "#set($inputRoot = $input.path('$'))\n{\n
 \"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
 \"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "requestTemplates" : {
```

```
          "application/json" : "#set($inputRoot = $input.path('$'))\n{\n
 \"a\" : $input.params('a'),\n  \"b\" : $input.params('b'),\n  \"op\" :
\"$input.params('op')\"\n}"
        },
        "passthroughBehavior" : "when_no_templates"
      }
    }
  },
  "/" : {
    "get" : {
      "parameters" : [ {
        "name" : "op",
        "in" : "query",
        "schema" : {
          "type" : "string"
        }
      }, {
        "name" : "a",
        "in" : "query",
        "schema" : {
          "type" : "string"
        }
      }, {
        "name" : "b",
        "in" : "query",
        "schema" : {
          "type" : "string"
        }
      } ],
      "responses" : {
        "200" : {
          "description" : "200 response",
          "content" : {
            "application/json" : {
              "schema" : {
                "$ref" : "#/components/schemas/Result"
              }
            }
          }
        }
      },
      "x-amazon-apigateway-integration" : {
        "type" : "aws",
        "httpMethod" : "POST",
```

```
          "uri" : "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:111122223333:function:Calc/invocations",
          "responses" : {
            "default" : {
              "statusCode" : "200",
              "responseTemplates" : {
                "application/json" : "#set($inputRoot = $input.path('$'))\n{\n
 \"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
 \"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "requestTemplates" : {
            "application/json" : "#set($inputRoot = $input.path('$'))\n{\n
  \"a\" : $input.params('a'),\n  \"b\" : $input.params('b'),\n  \"op\" :
 \"$input.params('op')\"\n}"
          },
          "passthroughBehavior" : "when_no_templates"
        }
      },
      "post" : {
        "requestBody" : {
          "content" : {
            "application/json" : {
              "schema" : {
                "$ref" : "#/components/schemas/Input"
              }
            }
          },
          "required" : true
        },
        "responses" : {
          "200" : {
            "description" : "200 response",
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Result"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration" : {
```

```
          "type" : "aws",
          "httpMethod" : "POST",
          "uri" : "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:111122223333:function:Calc/invocations",
          "responses" : {
            "default" : {
              "statusCode" : "200",
              "responseTemplates" : {
                "application/json" : "#set($inputRoot = $input.path('$'))\n{\n
 \"input\" : {\n    \"a\" : $inputRoot.a,\n    \"b\" : $inputRoot.b,\n    \"op\" :
 \"$inputRoot.op\"\n  },\n  \"output\" : {\n    \"c\" : $inputRoot.c\n  }\n}"
              }
            }
          },
          "passthroughBehavior" : "when_no_match"
        }
      }
    },
    "/{a}" : {
      "x-amazon-apigateway-any-method" : {
        "parameters" : [ {
          "name" : "a",
          "in" : "path",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "404" : {
            "description" : "404 response",
            "content" : { }
          }
        },
        "x-amazon-apigateway-integration" : {
          "type" : "mock",
          "responses" : {
            "default" : {
              "statusCode" : "404",
              "responseTemplates" : {
                "application/json" : "{ \"Message\" : \"Can't $context.httpMethod
 $context.resourcePath\" }"
              }
            }
```

```
          },
          "requestTemplates" : {
            "application/json" : "{\"statusCode\": 200}"
          },
          "passthroughBehavior" : "when_no_match"
        }
      }
    }
  },
  "components" : {
    "schemas" : {
      "Input" : {
        "title" : "Input",
        "type" : "object",
        "properties" : {
          "a" : {
            "type" : "number"
          },
          "b" : {
            "type" : "number"
          },
          "op" : {
            "type" : "string"
          }
        }
      },
      "Output" : {
        "title" : "Output",
        "type" : "object",
        "properties" : {
          "c" : {
            "type" : "number"
          }
        }
      },
      "Result" : {
        "title" : "Result",
        "type" : "object",
        "properties" : {
          "input" : {
            "$ref" : "#/components/schemas/Input"
          },
          "output" : {
            "$ref" : "#/components/schemas/Output"
```

```
                }
              }
            }
          }
        }
      }
```

生成 API 的 Java 开发工具包

在 API Gateway 中生成 API 的 Java 开发工具包

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 选择 Stages (阶段)。
4. 在阶段窗格中，选择阶段的名称。
5. 打开阶段操作菜单，然后选择生成 SDK。
6. 对于平台，选择 Java 平台并执行以下操作：
   a. 对于服务名称，指定您的开发工具包的名称。例如，**`SimpleCalcSdk`**。这将成为开发工具包客户端类的名称。该名称与 pom.xml 文件 (位于开发工具包的项目文件夹内) 中 `<name>` 下的 `<project>` 标记相对应。请勿包括连字符。
   b. 对于 Java Package Name (Java 包名称)，指定您的开发工具包的程序包名称。例如，**`examples.aws.apig.simpleCalc.sdk`**。此程序包名称将用作开发工具包库的命名空间。请勿包括连字符。
   c. 对于 Java Build System (Java 构建系统)，输入 **`maven`** 或 **`gradle`** 以指定生成系统。
   d. 对于 Java Group Id (Java 组 ID)，输入您开发工具包项目的组标识符。例如，输入 **`my-apig-api-examples`**。此标识符与 `<groupId>` 文件 (位于开发工具包的项目文件夹内) 中 `<project>` 下的 `pom.xml` 标记相对应。
   e. 对于 Java Artifact Id (Java 构件 ID)，输入您开发工具包项目的构件标识符。例如，输入 **`simple-calc-sdk`**。此标识符与 `<artifactId>` 文件 (位于开发工具包的项目文件夹内) 中 `<project>` 下的 `pom.xml` 标记相对应。
   f. 对于 Java Artifact Version (Java 构件版本)，输入版本标识符字符串。例如，**`1.0.0`**。此版本标识符与 `<version>` 文件 (位于开发工具包的项目文件夹内) 中 `<project>` 下的 `pom.xml` 标记相对应。

   g. 对于 Source Code License Text (源代码许可证文本)，输入源代码的许可证文本（如果有）。

7. 选择生成开发工具包，然后按照屏幕上的指示下载 API Gateway 生成的开发工具包。

按照使用由 API Gateway 为 REST API 生成的 Java 开发工具包中的说明使用生成的开发工具包。

每次更新 API 后，您必须重新部署 API，并重新生成开发工具包，才能添加这些更新。

生成 API 的 Android 开发工具包

在 API Gateway 中生成 API 的 Android 开发工具包

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 选择 Stages (阶段)。
4. 在阶段窗格中，选择阶段的名称。
5. 打开阶段操作菜单，然后选择生成 SDK。
6. 对于平台，选择 Android 平台并执行以下操作：
   a. 对于 Group ID (组 ID)，输入对应项目的唯一标识符。这将在 `pom.xml` 文件中使用（例如，**`com.mycompany`**）。
   b. 对于 Invoker package (调用程序包)，请为生成的客户端类输入命名空间（例如 **`com.mycompany.clientsdk`**）。
   c. 对于 Artifact ID (构件 ID)，输入已编译的 .jar 文件的名称（不含版本）。这将在 `pom.xml` 文件中使用（例如，**`aws-apigateway-api-sdk`**）。
   d. 对于 Artifact version (构件版本)，输入已生成客户端的构件版本号。此标识符将在 `pom.xml` 文件中使用且应遵循 *`major`*.*`minor`*.*`patch`* 模式（例如，**`1.0.0`**）。
7. 选择生成开发工具包，然后按照屏幕上的指示下载 API Gateway 生成的开发工具包。

按照使用由 API Gateway 为 REST API 生成的 Android 开发工具包中的说明使用生成的开发工具包。

每次更新 API 后，您必须重新部署 API，并重新生成开发工具包，才能添加这些更新。

生成 API 的 iOS 开发工具包

在 API Gateway 中生成 API 的 iOS 开发工具包

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 选择 Stages (阶段)。
4. 在阶段窗格中，选择阶段的名称。
5. 打开阶段操作菜单，然后选择生成 SDK。
6. 对于平台，选择 iOS (Objective-C) 或 iOS (Swift) 平台，然后执行以下操作：

    - 在前缀框中键入唯一前缀。

      前缀效果如下所示：举例来说，如果您为 SimpleCalc API 的开发工具包分配 **`SIMPLE_CALC`** 前缀，并使用 `input`、`output` 和 `result` 模型，则生成的开发工具包将包含封装 API 的 `SIMPLE_CALCSimpleCalcClient` 类，其中包括方法请求/响应。此外，生成的开发工具包将包含分别表示输入、输出和结果的 `SIMPLE_CALCinput`、`SIMPLE_CALCoutput` 和 `SIMPLE_CALCresult`，从而表示请求输入和响应输出。有关更多信息，请参阅 在 Objective-C 或 Swift 中使用由 API Gateway 为 REST API 生成的 iOS 开发工具包。
7. 选择生成开发工具包，然后按照屏幕上的指示下载 API Gateway 生成的开发工具包。

按照在 Objective-C 或 Swift 中使用由 API Gateway 为 REST API 生成的 iOS 开发工具包中的说明使用生成的开发工具包。

每次更新 API 后，您必须重新部署 API，并重新生成开发工具包，才能添加这些更新。

生成 REST API 的 JavaScript 开发工具包

在 API Gateway 中生成 API 的 JavaScript 开发工具包

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 选择 Stages (阶段)。
4. 在阶段窗格中，选择阶段的名称。
5. 打开阶段操作菜单，然后选择生成 SDK。

6. 对于平台，选择 JavaScript 平台。
7. 选择生成开发工具包，然后按照屏幕上的指示下载 API Gateway 生成的开发工具包。

按照使用由 API Gateway 为 REST API 生成的 JavaScript 开发工具包中的说明使用生成的开发工具包。

每次更新 API 后，您必须重新部署 API，并重新生成开发工具包，才能添加这些更新。

生成 API 的 Ruby 开发工具包

在 API Gateway 中生成 API 的 Ruby 开发工具包

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 选择 Stages (阶段)。
4. 在阶段窗格中，选择阶段的名称。
5. 打开阶段操作菜单，然后选择生成 SDK。
6. 对于平台，选择 Rubya 平台并执行以下操作：
   a. 对于服务名称，指定您的开发工具包的名称。例如，**`SimpleCalc`**。这用于生成您的 API 的 Ruby Gem 命名空间。该名称必须是全字母形式 (`a-zA-Z`)，不含任何特殊字符或数字。
   b. 对于 Ruby Gem Name (Ruby Gem 名称)，指定 Ruby Gem 的名称，其中将包含为您 API 生成的开发工具包源代码。默认情况下，这是小写的服务名称加上 `-sdk` 后缀 — 例如 **`simplecalc-sdk`**。
   c. 对于 Ruby Gem Version (Ruby Gem 版本)，指定所生成 Ruby Gem 的版本号。默认情况下，将它设置为 `1.0.0`。
7. 选择生成开发工具包，然后按照屏幕上的指示下载 API Gateway 生成的开发工具包。

按照使用由 API Gateway 为 REST API 生成的 Ruby 开发工具包中的说明使用生成的开发工具包。

每次更新 API 后，您必须重新部署 API，并重新生成开发工具包，才能添加这些更新。

使用 AWS CLI 命令为 API 生成开发工具包

您可以使用 AWS CLI，通过调用 get-sdk 命令为支持的平台生成 API 的开发工具包并下载。在下文中我们将针对一些支持的平台演示此操作。

主题

- 使用 AWS CLI 生成和下载 Java for Android 开发工具包
- 使用 AWS CLI 生成和下载 JavaScript 开发工具包
- 使用 AWS CLI 生成和下载 Ruby 开发工具包

使用 AWS CLI 生成和下载 Java for Android 开发工具包

要在指定阶段 (udpuvvzbkc) 生成并下载由 API 的 API Gateway (`test`) 生成的 Java for Android 开发工具包，请按以下所示调用命令：

```
aws apigateway get-sdk \
            --rest-api-id udpuvvzbkc \
            --stage-name test \
            --sdk-type android \
            --parameters groupId='com.mycompany',\
                invokerPackage='com.mycompany.myApiSdk',\
                artifactId='myApiSdk',\
                artifactVersion='0.0.1' \
            ~/apps/myApi/myApi-android-sdk.zip
```

`~/apps/myApi/myApi-android-sdk.zip` 的最后输入是名为 `myApi-android-sdk.zip` 的已下载开发工具包文件的路径。

使用 AWS CLI 生成和下载 JavaScript 开发工具包

要在指定阶段 (udpuvvzbkc) 生成并下载由 API 的 API Gateway (`test`) 生成的 JavaScript 开发工具包，请按以下所示调用命令：

```
aws apigateway get-sdk \
            --rest-api-id udpuvvzbkc \
            --stage-name test \
            --sdk-type javascript \
            ~/apps/myApi/myApi-js-sdk.zip
```

`~/apps/myApi/myApi-js-sdk.zip` 的最后输入是名为 `myApi-js-sdk.zip` 的已下载开发工具包文件的路径。

使用 AWS CLI 生成和下载 Ruby 开发工具包

要在指定阶段 (udpuvvzbkc) 上，生成并下载 API (`test`) 的 Ruby 开发工具包，请按以下所示调用命令：

```
aws apigateway get-sdk \
            --rest-api-id udpuvvzbkc \
            --stage-name test  \
            --sdk-type ruby \
            --parameters service.name=myApiRubySdk,ruby.gem-name=myApi,ruby.gem-
version=0.01 \
            ~/apps/myApi/myApi-ruby-sdk.zip
```

`~/apps/myApi/myApi-ruby-sdk.zip` 的最后输入是名为 `myApi-ruby-sdk.zip` 的已下载开发工具包文件的路径。

接下来，我们将说明如何使用生成的开发工具包来调用底层 API。有关更多信息，请参阅 在 Amazon API Gateway 中调用 REST API。

通过 AWS Marketplace 销售 API Gateway API

构建、测试和部署 API 后，您可以将其打包到 API Gateway usage plan（使用计划）中，并通过 AWS Marketplace 将该计划作为软件即服务 (SaaS) 产品进行销售。AWS Marketplace 根据向使用计划发出的请求数量，对订阅您的产品/服务的 API 买家进行计费。

要在 AWS Marketplace 上销售 API，您必须设置销售渠道以将 AWS Marketplace 与 API Gateway 集成。一般而言，这涉及在 AWS Marketplace 上列出您的产品，使用适当的策略设置 IAM 角色以允许 API Gateway 向 AWS Marketplace 发送用量指标，将 AWS Marketplace 产品与 API Gateway 使用计划进行关联，以及将 AWS Marketplace 买家与 API Gateway API 密钥进行关联。以下各节讨论了相关详细信息。

有关在 AWS Marketplace 上将您的 API 作为 SaaS 产品进行销售的更多信息，请参阅 AWS Marketplace 用户指南。

主题

- 初始化 AWS Marketplace 与 API Gateway 的集成
- 处理使用计划的客户订阅

初始化 AWS Marketplace 与 API Gateway 的集成

以下任务是为了一次性初始化 AWS Marketplace 与 API Gateway 的集成，以便您将 API 作为 SaaS 产品进行销售。

在 中列出产品AWS Marketplace

要将您的使用计划作为 SaaS 产品列出，请通过 AWS Marketplace 提交产品加载表单。产品必须包含 `apigateway` 类型且名为 `requests` 的维度。此维度定义基于请求的定价，并由 API Gateway 用于计量对您的 API 提出的请求。

创建计量角色

使用以下执行策略和信任策略，创建一个名为 `ApiGatewayMarketplaceMeteringRole` 的 IAM 角色。此角色允许 API 代表您向 AWS Marketplace 发送用量指标。

计量角色的执行策略

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "aws-marketplace:BatchMeterUsage",
        "aws-marketplace:ResolveCustomer"
      ],
      "Resource": "*",
      "Effect": "Allow"
    }
  ]
}
```

计量角色的可信关系策略

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
```

```
        "Action": "sts:AssumeRole"
      }
    ]
}
```

将使用计划与 AWS Marketplace 产品关联

在 AWS Marketplace 上列出产品时，您会收到一个 AWS Marketplace 产品代码。要将 API Gateway 与 AWS Marketplace 集成，请将您的使用计划与 AWS Marketplace 产品代码关联。您可以通过使用 API Gateway 控制台、API Gateway REST API、适用于 API Gateway 的 AWS CLI 或适用于 API Gateway 的 AWS SDK 将 API Gateway `UsagePlan` 的 `productCode` 字段设为 AWS Marketplace 产品代码来启用关联。以下代码示例使用 API Gateway REST API：

```
PATCH /usageplans/USAGE_PLAN_ID
Host: apigateway.region.amazonaws.com
Authorization: ...

{
    "patchOperations" : [{
        "path" : "/productCode",
        "value" : "MARKETPLACE_PRODUCT_CODE",
        "op" : "replace"
    }]
}
```

处理使用计划的客户订阅

以下任务由您的开发人员门户应用程序处理。

当客户通过 AWS Marketplace 订阅您的产品时，AWS Marketplace 会将 `POST` 请求转发至 SaaS 订阅您在 AWS Marketplace 中上架产品时注册的 URL。`POST` 请求附带一个包含买家信息的 `x-amzn-marketplace-token` 参数。按照 SaaS 客户登记中的说明在您的开发人员门户应用程序中处理此重定向。

作为对客户订阅请求的响应，AWS Marketplace 会向您可以订阅的 Amazon SNS 主题发送 `subscribe-success` 通知。（请参阅 SaaS 客户登记）。要接受客户订阅请求，您应创建或检索客户的 API Gateway API 密钥，将客户的 AWS Marketplace 预置 `customerId` 与 API 密钥进行关联，然后将该 API 密钥添加到您的使用计划中，以此来处理 `subscribe-success` 通知。

当客户的订阅请求完成后，开发人员门户应用程序应向客户展示关联的 API 密钥，并告知客户该 API 密钥必须包含在对 API 提出的请求的 `x-api-key` 标头中。

如果客户取消对使用计划的订阅，AWS Marketplace 会向 SNS 主题发送 `unsubscribe-success` 通知。要完成客户取消订阅的过程，应通过从使用计划中删除客户的 API 密钥来处理 `unsubscribe-success` 通知。

授权客户访问使用计划

要授权给定客户访问您的使用计划，请使用 API Gateway API 为该客户获取或创建 API 密钥，然后将该 API 密钥添加到使用计划中。

以下示例介绍了如何调用 API Gateway REST API 以创建具有特定 AWS Marketplace `customerId` 值 (*`MARKETPLACE_CUSTOMER_ID`*) 的新 API 密钥。

```
POST apikeys HTTP/1.1
Host: apigateway.region.amazonaws.com
Authorization: ...

{
  "name" : "my_api_key",
  "description" : "My API key",
  "enabled" : "false",
  "stageKeys" : [ {
    "restApiId" : "uycll6xg9a",
    "stageName" : "prod"
  } ],
  "customerId" : "MARKETPLACE_CUSTOMER_ID"
}
```

以下示例介绍了如何获取具有特定 AWS Marketplace `customerId` 值 (*`MARKETPLACE_CUSTOMER_ID`*) 的 API 密钥。

```
GET apikeys?customerId=MARKETPLACE_CUSTOMER_ID HTTP/1.1
Host: apigateway.region.amazonaws.com
Authorization: ...
```

要向使用计划添加 API 密钥，请通过相关使用计划的 API 密钥创建一个 [`UsagePlanKey`]。以下示例介绍如何使用 API Gateway REST API 完成此操作，其中 `n371pt` 是使用计划 ID，`q5ugs7qjjh` 是从之前的示例返回的示例 API `keyId`。

```
POST /usageplans/n371pt/keys HTTP/1.1
Host: apigateway.region.amazonaws.com
```

```
Authorization: ...

{
    "keyId": "q5ugs7qjjh",
    "keyType": "API_KEY"
}
```

将客户与 API 密钥关联

必须将 [ApiKey](#) 的 `customerId` 字段更新为客户的 AWS Marketplace 客户 ID。这可将 API 密钥与 AWS Marketplace 客户关联，从而可对买家进行计量和计费。以下代码示例调用 API Gateway REST API 来执行此操作。

```
PATCH /apikeys/q5ugs7qjjh
Host: apigateway.region.amazonaws.com
Authorization: ...

{
    "patchOperations" : [{
        "path" : "/customerId",
        "value" : "MARKETPLACE_CUSTOMER_ID",
        "op" : "replace"
    }]
}
```

保护您的 REST API

API Gateway 提供了多种方法来保护您的 API 免受某些威胁，例如恶意用户或流量高峰。您可以使用生成 SSL 证书、配置 Web 应用程序防火墙、设置节流目标以及仅允许从 Virtual Private Cloud (VPC) 访问 API 等策略来保护 API。在本节中，您可以了解如何使用 API Gateway 启用这些功能。

主题

为 REST API 配置双向 TLS 身份验证

双向 TLS 身份验证要求在客户端和服务器之间进行双向身份验证。使用双向 TLS，客户端必须提供 X.509 证书来验证其身份才能访问您的 API。双向 TLS 是物联网 (IoT) 和企业对企业应用程序的常见要求。

您可以使用双向 TLS 以及 API Gateway 支持的其他授权和身份验证操作。API Gateway 将客户端提供的证书转发给 Lambda 授权方和后端集成。

⚠ Important

默认情况下，客户端可以通过使用 API Gateway 为 API 生成的 `execute-api` 端点来调用您的 API。要确保客户端只能通过使用具有双向 TLS 的自定义域名访问您的 API，请禁用默认 `execute-api` 端点。要了解更多信息，请参阅“the section called “禁用默认终端节点””。

主题

- 使用双向 TLS 的先决条件
- 为自定义域名配置双向 TLS
- 使用需要双向 TLS 的自定义域名调用 API
- 更新您的信任存储库
- 禁用双向 TLS
- 排查证书警告问题
- 域名冲突故障排除
- 域名状态消息故障排除

使用双向 TLS 的先决条件

要配置双向 TLS，您需要：

- 自定义域名
- 至少在 AWS Certificate Manager 中为您的自定义域名配置了一个证书
- 已配置信任存储库并上载到 Amazon S3

自定义域名

要为 REST API 启用双向 TLS，您必须为 API 配置自定义域名。您可以为自定义域名启用双向 TLS，然后向客户端提供自定义域名。要使用启用了双向 TLS 的自定义域名访问 API，客户端必须提供您在 API 请求中信任的证书。您可以在 the section called “自定义域名” 中找到更多信息。

使用 AWS Certificate Manager 颁发的证书

您可以直接从 ACM 请求公开可信的证书，也可以导入公开证书或自签名证书。要在 ACM 中设置证书，请前往 ACM。如果要导入证书，请继续阅读以下部分中的内容。

使用导入的证书或 AWS Private Certificate Authority 证书

要使用导入至 ACM 中的证书或来自 AWS Private Certificate Authority 的带有双向 TLS 的证书，API Gateway 需要由 ACM 颁发的 `ownershipVerificationCertificate`。此所有权证书仅用于验证您是否有权使用域名。它不用于 TLS 握手。如果您还没有 `ownershipVerificationCertificate`，请前往 https://console.aws.amazon.com/acm/ 来设置一个。

您需要确保此证书在域名生命周期内有效。如果证书过期且自动续订失败，则域名的所有更新都将被锁定。您需要使用有效 `ownershipVerificationCertificate` 更新 `ownershipVerificationCertificateArn`，然后才能进行任何其他更改。`ownershipVerificationCertificate` 不能用作 API Gateway 中另一个双向 TLS 域的服务器证书。如果直接向 ACM 中重新导入证书，发布者必须保持不变。

配置信任存储库

信任存储库是带有 `.pem` 文件扩展名的文本文件。它们是来自证书颁发机构的证书的受信任列表。要使用双向 TLS，请创建您信任的 X.509 证书的信任存储库以访问您的 API。

您必须在信任存储库中包含完整的信任链，从颁发的 CA 证书到根 CA 证书。API Gateway 接受信任链中存在的任何 CA 发布的客户端证书。证书可以来自公有或私有证书颁发机构。证书的最大链长度可为四。您还可以提供自签名证书。信任库支持以下算法：

- SHA-256 或更强
- RSA-2048 或更强
- ECDSA-256 或 ECDSA-384

API Gateway 验证许多证书属性。您可以使用 Lambda 授权方在客户端调用 API 时执行其他检查，包括检查证书是否已被吊销。API Gateway 验证以下属性：

验证	说明
X.509 语法	证书必须满足 X.509 语法要求。
完整性	证书的内容不得与信任存储库中证书颁发机构签名的内容有所差异。
有效性	证书的有效期必须是最新的。
名称链接/键链接	证书的名称和主题必须形成一个完整的链条。证书的最大链长度可为四。

以单个文件的形式将信任存储库上载到 Amazon S3 存储桶

以下是 .pem 文件具体形式的示例。

Example certificates.pem

```
-----BEGIN CERTIFICATE-----
<Certificate contents>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Certificate contents>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Certificate contents>
-----END CERTIFICATE-----
...
```

以下 AWS CLI 命令会将 `certificates.pem` 上载到 Amazon S3 存储桶。

```
aws s3 cp certificates.pem s3://bucket-name
```

Amazon S3 存储桶必须具有 API Gateway 的读取权限，才能允许 API Gateway 访问您的信任库。

为自定义域名配置双向 TLS

要为 REST API 配置双向 TLS，必须通过 TLS_1_2 安全策略为 API 使用区域自定义域名。有关选择安全策略的更多信息，请参阅the section called “选择安全策略”。

Note

私有 API 不支持双向 TLS。

将信任存储库上传到 Amazon S3 后，您可以将自定义域名配置为使用双向 TLS。将以下内容（包括斜杠）粘贴到终端中：

```
aws apigateway create-domain-name --region us-east-2 \
    --domain-name api.example.com \
    --regional-certificate-arn arn:aws:acm:us-
east-2:123456789012:certificate/123456789012-1234-1234-1234-12345678 \
    --endpoint-configuration types=REGIONAL \
    --security-policy TLS_1_2 \
    --mutual-tls-authentication truststoreUri=s3://bucket-name/key-name
```

创建域名后，您必须为 API 操作配置 DNS 记录和基本路径映射。要了解更多信息，请参阅 在 API Gateway 中设置区域自定义域名。

使用需要双向 TLS 的自定义域名调用 API

要调用启用了双向 TLS 的 API，客户端必须在 API 请求中提供受信任证书。当客户端尝试调用您的 API 时，API Gateway 会在您的信任存储库中查找客户端证书的发布者。为使 API Gateway 继续处理请求，证书的发布者和直至根 CA 证书的完整信任链必须位于您的信任存储库中。

以下示例 `curl` 命令将请求发送到在请求中包含 `api.example.com,` 的 `my-cert.pem`。`my-key.key` 是证书的私有密钥。

```
curl -v --key ./my-key.key --cert ./my-cert.pem api.example.com
```

仅当您的信任存储库信任证书时，才会调用您的 API。以下情况将导致 API Gateway 使 TLS 握手失败，并以 `403` 状态代码拒绝请求。如果您的证书：

- 不可信
- 已过期
- 未使用支持的算法

Note

API Gateway 不验证证书是否已被吊销。

更新您的信任存储库

要更新信任存储库中的证书，请将新的证书服务包上载到 Amazon S3。然后，您可以更新自定义域名以使用更新后的证书。

使用 Amazon S3 版本控制来维护信任存储库的多个版本。当您更新自定义域名以使用新的信任存储库版本时，如果证书无效，则 API Gateway 返回警告。

API Gateway 仅在您更新域名时才生成证书警告。如果先前上传的证书过期，API Gateway 不会通知您。

以下 AWS CLI 命令将自定义域名更新为使用新的信任存储库版本。

```
aws apigateway update-domain-name \
    --domain-name api.example.com \
    --patch-operations op='replace',path='/mutualTlsAuthentication/
truststoreVersion',value='abcdef123'
```

禁用双向 TLS

要为自定义域名禁用双向 TLS，请从自定义域名中删除信任存储库，如以下命令所示。

```
aws apigateway update-domain-name \
    --domain-name api.example.com \
    --patch-operations op='replace',path='/mutualTlsAuthentication/
truststoreUri',value=''
```

排查证书警告问题

当使用双向 TLS 创建自定义域名时，如果信任存储库中的证书无效，则 API Gateway 会返回警告。在更新自定义域名以使用新的信任存储库时，也可能会出现这种情况。警告指示证书存在问题以及生成警告的证书的主题。仍然为您的 API 启用双向 TLS，但某些客户端可能无法访问您的 API。

要标识生成警告的证书，您需要解码信任存储库中的证书。您可以使用诸如 `openssl` 等工具对证书进行解码和标识其主题。

以下命令显示证书的内容，包括其主题：

```
openssl x509 -in certificate.crt -text -noout
```

更新或删除生成警告的证书，然后将新信任存储库上传到 Amazon S3。上载新的信任存储库后，请更新您的自定义域名以使用新的信任存储库。

域名冲突故障排除

错误 `"The certificate subject <certSubject> conflicts with an existing certificate from a different issuer."` 表示多个证书颁发机构已发布此域的证书。对于证书中的每个主题，双向 TLS 域的 API Gateway 中只能有一个发布者。您需要通过单个发布者获取该主题的所有证书。如果您无法控制的证书出现问题，但您可以证明域名的所有权，请联系 AWS Support 以提出支持请求。

域名状态消息故障排除

`PENDING_CERTIFICATE_REIMPORT`：这意味着您将证书重新导入到 ACM，并且验证失败，因为新证书具有 SAN（主题备用名称），而 `ownershipVerificationCertificate` 不涵盖该 SAN，或是证书中的主题或 SAN 不涵盖域名。某些内容可能配置不正确，或导入了无效的证书。您需要将有效证书重新导入 ACM。有关验证的更多信息，请参阅验证域名所有权。

`PENDING_OWNERSHIP_VERIFICATION`：这意味着您之前验证的证书已过期，ACM 无法自动续订。您需要续订证书或请求新证书。有关证书续订的更多信息，请查看 ACM 对托管式证书续订进行故障排除指南。

生成和配置 SSL 证书用于后端身份验证

您可以使用 API Gateway 生成一个 SSL 证书，然后在后端使用其公有密钥来验证发往后端系统的 HTTP 请求是否来自 API Gateway。这样，您的 HTTP 后端便可控制且仅接受来自 Amazon API Gateway 的请求（即使该后端可公开访问）。

> Note
>
> 某些后端服务器可能不像 API Gateway 那样支持 SSL 客户端身份验证，并且可能返回 SSL 证书错误。有关不兼容后端服务器的列表，请参阅 the section called “重要提示”。

API Gateway 生成的 SSL 证书为自签名证书，只有证书的公有密钥在 API Gateway 控制台中可见或通过 API 显示。

主题

- 使用 API Gateway 控制台生成客户端证书
- 配置 API 以使用 SSL 证书
- 测试调用，以验证客户端证书配置
- 配置后端 HTTPS 服务器以验证客户端证书
- 轮换即将过期的客户端证书
- API Gateway 支持的 HTTP 和 HTTP 代理集成证书的颁发机构

使用 API Gateway 控制台生成客户端证书

1. 通过以下网址打开 API Gateway 控制台：https://console.aws.amazon.com/apigateway/。
2. 选择一个 REST API。
3. 在主导航窗格中，选择客户端证书。
4. 从客户端证书页面中，选择生成证书。
5. （可选）对于描述，输入描述。
6. 选择生成证书以生成证书。API Gateway 会生成新证书，并返回新证书的 GUID 以及 PEM 编码的公有密钥。

您现在可以配置 API 以使用该证书。

配置 API 以使用 SSL 证书

这些说明假设您已完成使用 API Gateway 控制台生成客户端证书。

1. 在 API Gateway 控制台中，创建或打开一个您要对其使用客户端证书的 API。确保 API 已部署到某个阶段。
2. 在主导航窗格中，选择阶段。
3. 在阶段详细信息部分中，选择编辑。
4. 对于客户端证书，选择一个证书。
5. 选择 Save changes（保存更改）。

   如果已在 API Gateway 控制台中部署了 API，则需要重新进行部署，以使更改生效。有关更多信息，请参阅 the section called “将 REST API 重新部署到阶段”。

为 API 选择证书并保存后，API Gateway 会将该证书用于对您 API 中的 HTTP 集成的所有调用。

测试调用，以验证客户端证书配置

1. 选择 API 方法。选择测试选项卡。您可能需要选择右箭头按钮以显示测试选项卡。
2. 对于客户端证书，选择一个证书。
3. 选择测试。

API Gateway 将为 HTTP 后端呈现所选的 SSL 证书以对 API 进行身份验证。

配置后端 HTTPS 服务器以验证客户端证书

这些说明假设您已完成使用 API Gateway 控制台生成客户端证书，并已下载客户端证书的副本。您可以通过调用 API Gateway REST API 的 `clientcertificate:by-id` 或 AWS CLI 的 `get-client-certificate` 来下载客户端证书。

在配置用于验证 API Gateway 的客户端 SSL 证书的后端 HTTPS 服务器之前，您必须获得 PEM 编码的私有密钥，以及可信证书颁发机构提供的服务器端证书。

如果服务器域名是 `myserver.mydomain.com`，则服务器证书的 CNAME 值必须是 `myserver.mydomain.com` 或 `*.mydomain.com`。

支持的证书颁发机构包含 Let's Encrypt (进行加密) 或 the section called “受支持的 HTTP 和 HTTP 代理集成证书颁发机构” 之一。

例如，假设客户端证书文件为 `apig-cert.pem`，服务器私有密钥和证书文件分别为 `server-key.pem` 和 `server-cert.pem`。对于后端中的 Node.js 服务器，您可以配置类似于以下的服务器：

```
var fs = require('fs');
var https = require('https');
var options = {
    key: fs.readFileSync('server-key.pem'),
    cert: fs.readFileSync('server-cert.pem'),
    ca: fs.readFileSync('apig-cert.pem'),
    requestCert: true,
    rejectUnauthorized: true
};
https.createServer(options, function (req, res) {
    res.writeHead(200);
    res.end("hello world\n");
```

```
}).listen(443);
```

对于 node-express 应用程序，您可以使用 client-certificate-auth 模块，利用 PEM 编码证书对客户端请求进行身份验证。

对于其他 HTTPS 服务器，请参阅服务器文档。

轮换即将过期的客户端证书

API Gateway 生成的客户端证书有效期为 365 天。您必须在 API 阶段的客户端证书过期之前轮换证书，以避免该 API 停止工作。您可以通过调用 API Gateway REST API 的 clientCertificate:by-id，或 AWS CLI 命令 get-client-certificate 并检查返回的 expirationDate 属性，来查看证书的到期日期。

要轮换客户端证书，请执行以下操作：

1. 调用 API Gateway REST API 的 clientcertificate:generate 或 AWS CLI 命令 generate-client-certificate 生成新的客户端证书。在本教程中，我们将假设新的客户端证书 ID 是 `ndiqef`。
2. 更新后端服务器，以包括新的客户端证书。目前请不要删除现有客户端证书。

   有些服务器可能需要重启以完成更新。请参考服务器文档，了解在更新期间是否必须重启服务器。
3. 用新的客户端证书 ID (`ndiqef`) 调用 API Gateway REST API 的 stage:update，使用新客户端证书更新 API 阶段：

   ```
   PATCH /restapis/{restapi-id}/stages/stage1 HTTP/1.1
   Content-Type: application/json
   Host: apigateway.us-east-1.amazonaws.com
   X-Amz-Date: 20170603T200400Z
   Authorization: AWS4-HMAC-SHA256 Credential=...

   {
     "patchOperations" : [
       {
           "op" : "replace",
           "path" : "/clientCertificateId",
           "value" : "ndiqef"
       }
     ]
   }
   ```

   或通过调用 CLI 命令 update-stage。

4. 更新后端服务器，以删除旧证书。
5. 通过调用 API Gateway REST API 的 clientcertificate:delete，同时指定旧证书的 clientCertificateId (a1b2c3)，从 API Gateway 删除旧证书：

   ```
   DELETE /clientcertificates/a1b2c3
   ```

   或通过调用 CLI 命令 delete-client-certificate：

   ```
   aws apigateway delete-client-certificate --client-certificate-id a1b2c3
   ```

要在控制台中为之前部署的 API 轮换客户端证书，请执行以下操作：

1. 在主导航窗格中，选择客户端证书。
2. 在客户端证书窗格中，选择生成证书。
3. 打开您要对其使用客户端证书的 API。
4. 在选定 API 下选择 Stages (阶段)，然后选择一个阶段。
5. 在阶段详细信息部分中，选择编辑。
6. 对于客户端证书，请选择新证书。
7. 要保存设置，请选择保存更改。

   您将需要重新部署 API 才能使更改生效。有关更多信息，请参阅 the section called “将 REST API 重新部署到阶段”。

API Gateway 支持的 HTTP 和 HTTP 代理集成证书的颁发机构

以下列表显示了受 API Gateway 支持的 HTTP、HTTP 代理和私有集成证书颁发机构。

```
Alias name: accvraiz1
  SHA1: 93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
  SHA256:
 9A:6E:C0:12:E1:A7:DA:9D:BE:34:19:4D:47:8A:D7:C0:DB:18:22:FB:07:1D:F1:29:81:49:6E:D1:04:38:41:1
Alias name: acraizfnmtrcm
  SHA1: EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
  SHA256:
 EB:C5:57:0C:29:01:8C:4D:67:B1:AA:12:7B:AF:12:F7:03:B4:61:1E:BC:17:B7:DA:B5:57:38:94:17:9B:93:F
Alias name: actalis
  SHA1: F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
```

```
  SHA256:
 55:92:60:84:EC:96:3A:64:B9:6E:2A:BE:01:CE:0B:A8:6A:64:FB:FE:BC:C7:AA:B5:AF:C1:55:B3:7F:D7:60:6
Alias name: actalisauthenticationrootca
  SHA1: F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
  SHA256:
 55:92:60:84:EC:96:3A:64:B9:6E:2A:BE:01:CE:0B:A8:6A:64:FB:FE:BC:C7:AA:B5:AF:C1:55:B3:7F:D7:60:6
Alias name: addtrustclass1ca
  SHA1: CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
  SHA256:
 8C:72:09:27:9A:C0:4E:27:5E:16:D0:7F:D3:B7:75:E8:01:54:B5:96:80:46:E3:1F:52:DD:25:76:63:24:E9:A
Alias name: addtrustexternalca
  SHA1: 02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
  SHA256:
 68:7F:A4:51:38:22:78:FF:F0:C8:B1:1F:8D:43:D5:76:67:1C:6E:B2:BC:EA:B4:13:FB:83:D9:65:D0:6D:2F:F
Alias name: addtrustqualifiedca
  SHA1: 4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
  SHA256:
 80:95:21:08:05:DB:4B:BC:35:5E:44:28:D8:FD:6E:C2:CD:E3:AB:5F:B9:7A:99:42:98:8E:B8:F4:DC:D0:60:1
Alias name: affirmtrustcommercial
  SHA1: F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
  SHA256:
 03:76:AB:1D:54:C5:F9:80:3C:E4:B2:E2:01:A0:EE:7E:EF:7B:57:B6:36:E8:A9:3C:9B:8D:48:60:C9:6F:5F:A
Alias name: affirmtrustcommercialca
  SHA1: F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
  SHA256:
 03:76:AB:1D:54:C5:F9:80:3C:E4:B2:E2:01:A0:EE:7E:EF:7B:57:B6:36:E8:A9:3C:9B:8D:48:60:C9:6F:5F:A
Alias name: affirmtrustnetworking
  SHA1: 29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
  SHA256:
 0A:81:EC:5A:92:97:77:F1:45:90:4A:F3:8D:5D:50:9F:66:B5:E2:C5:8F:CD:B5:31:05:8B:0E:17:F3:F0:B4:1
Alias name: affirmtrustnetworkingca
  SHA1: 29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
  SHA256:
 0A:81:EC:5A:92:97:77:F1:45:90:4A:F3:8D:5D:50:9F:66:B5:E2:C5:8F:CD:B5:31:05:8B:0E:17:F3:F0:B4:1
Alias name: affirmtrustpremium
  SHA1: D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
  SHA256:
 70:A7:3F:7F:37:6B:60:07:42:48:90:45:34:B1:14:82:D5:BF:0E:69:8E:CC:49:8D:F5:25:77:EB:F2:E9:3B:9
Alias name: affirmtrustpremiumca
  SHA1: D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
  SHA256:
 70:A7:3F:7F:37:6B:60:07:42:48:90:45:34:B1:14:82:D5:BF:0E:69:8E:CC:49:8D:F5:25:77:EB:F2:E9:3B:9
Alias name: affirmtrustpremiumecc
  SHA1: B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
```

```
  SHA256:
 BD:71:FD:F6:DA:97:E4:CF:62:D1:64:7A:DD:25:81:B0:7D:79:AD:F8:39:7E:B4:EC:BA:9C:5E:84:88:82:14:2
Alias name: affirmtrustpremiumeccca
  SHA1: B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
  SHA256:
 BD:71:FD:F6:DA:97:E4:CF:62:D1:64:7A:DD:25:81:B0:7D:79:AD:F8:39:7E:B4:EC:BA:9C:5E:84:88:82:14:2
Alias name: amazon-ca-g4-acm1
  SHA1: F2:0D:28:B6:29:C2:2C:5E:84:05:E6:02:4D:97:FE:8F:A0:84:93:A0
  SHA256:
 B0:11:A4:F7:29:6C:74:D8:2B:F5:62:DF:87:D7:28:C7:1F:B5:8C:F4:E6:73:F2:78:FC:DA:F3:FF:83:A6:8C:8
Alias name: amazon-ca-g4-acm2
  SHA1: A7:E6:45:32:1F:7A:B7:AD:C0:70:EA:73:5F:AB:ED:C3:DA:B4:D0:C8
  SHA256:
 D7:A8:7C:69:95:D0:E2:04:2A:32:70:A7:E2:87:FE:A7:E8:F4:C1:70:62:F7:90:C3:EB:BB:53:F2:AC:39:26:B
Alias name: amazon-ca-g4-acm3
  SHA1: 7A:DB:56:57:5F:D6:EE:67:85:0A:64:BB:1C:E9:E4:B0:9A:DB:9D:07
  SHA256:
 6B:EB:9D:20:2E:C2:00:70:BD:D2:5E:D3:C0:C8:33:2C:B4:78:07:C5:82:94:4E:7E:23:28:22:71:A4:8E:0E:C
Alias name: amazon-ca-g4-legacy
  SHA1: EA:E7:DE:F9:0A:BE:9F:0B:68:CE:B7:24:0D:80:74:03:BF:6E:B1:6E
  SHA256:
 CD:72:C4:7F:B4:AD:28:A4:67:2B:E1:86:47:D4:40:E9:3B:16:2D:95:DB:3C:2F:94:BB:81:D9:09:F7:91:24:5
Alias name: amazon-root-ca-ecc-384-1
  SHA1: F9:5E:4A:AB:9C:2D:57:61:63:3D:B2:57:B4:0F:24:9E:7B:E2:23:7D
  SHA256:
 C6:BD:E5:66:C2:72:2A:0E:96:E9:C1:2C:BF:38:92:D9:55:4D:29:03:57:30:72:40:7F:4E:70:17:3B:3C:9B:6
Alias name: amazon-root-ca-rsa-2k-1
  SHA1: 8A:9A:AC:27:FC:86:D4:50:23:AD:D5:63:F9:1E:AE:2C:AF:63:08:6C
  SHA256:
 0F:8F:33:83:FB:70:02:89:49:24:E1:AA:B0:D7:FB:5A:BF:98:DF:75:8E:0F:FE:61:86:92:BC:F0:75:35:CC:8
Alias name: amazon-root-ca-rsa-4k-1
  SHA1: EC:BD:09:61:F5:7A:B6:A8:76:BB:20:8F:14:05:ED:7E:70:ED:39:45
  SHA256:
 36:AE:AD:C2:6A:60:07:90:6B:83:A3:73:2D:D1:2B:D4:00:5E:C7:F2:76:11:99:A9:D4:DA:63:2F:59:B2:8B:C
Alias name: amazon1
  SHA1: 8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
  SHA256:
 8E:CD:E6:88:4F:3D:87:B1:12:5B:A3:1A:C3:FC:B1:3D:70:16:DE:7F:57:CC:90:4F:E1:CB:97:C6:AE:98:19:6
Alias name: amazon2
  SHA1: 5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
  SHA256:
 1B:A5:B2:AA:8C:65:40:1A:82:96:01:18:F8:0B:EC:4F:62:30:4D:83:CE:C4:71:3A:19:C3:9C:01:1E:A4:6D:B
Alias name: amazon3
  SHA1: 0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
```

```
  SHA256:
 18:CE:6C:FE:7B:F1:4E:60:B2:E3:47:B8:DF:E8:68:CB:31:D0:2E:BB:3A:DA:27:15:69:F5:03:43:B4:6D:B3:A
Alias name: amazon4
  SHA1: F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
  SHA256:
 E3:5D:28:41:9E:D0:20:25:CF:A6:90:38:CD:62:39:62:45:8D:A5:C6:95:FB:DE:A3:C2:2B:0B:FB:25:89:70:9
Alias name: amazonrootca1
  SHA1: 8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
  SHA256:
 8E:CD:E6:88:4F:3D:87:B1:12:5B:A3:1A:C3:FC:B1:3D:70:16:DE:7F:57:CC:90:4F:E1:CB:97:C6:AE:98:19:6
Alias name: amazonrootca2
  SHA1: 5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
  SHA256:
 1B:A5:B2:AA:8C:65:40:1A:82:96:01:18:F8:0B:EC:4F:62:30:4D:83:CE:C4:71:3A:19:C3:9C:01:1E:A4:6D:B
Alias name: amazonrootca3
  SHA1: 0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
  SHA256:
 18:CE:6C:FE:7B:F1:4E:60:B2:E3:47:B8:DF:E8:68:CB:31:D0:2E:BB:3A:DA:27:15:69:F5:03:43:B4:6D:B3:A
Alias name: amazonrootca4
  SHA1: F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
  SHA256:
 E3:5D:28:41:9E:D0:20:25:CF:A6:90:38:CD:62:39:62:45:8D:A5:C6:95:FB:DE:A3:C2:2B:0B:FB:25:89:70:9
Alias name: amzninternalinfoseccag3
  SHA1: B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
  SHA256:
 81:03:0B:C7:E2:54:DA:7B:F8:B7:45:DB:DD:41:15:89:B5:A3:81:86:FB:4B:29:77:1F:84:0A:18:D9:67:6D:6
Alias name: amzninternalrootca
  SHA1: A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06
  SHA256:
 0E:DE:63:C1:DC:7A:8E:11:F1:AB:BC:05:4F:59:EE:49:9D:62:9A:2F:DE:9C:A7:16:32:A2:64:29:3E:8B:66:A
Alias name: aolrootca1
  SHA1: 39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
  SHA256:
 77:40:73:12:C6:3A:15:3D:5B:C0:0B:4E:51:75:9C:DF:DA:C2:37:DC:2A:33:B6:79:46:E9:8E:9B:FA:68:0A:E
Alias name: aolrootca2
  SHA1: 85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
  SHA256:
 7D:3B:46:5A:60:14:E5:26:C0:AF:FC:EE:21:27:D2:31:17:27:AD:81:1C:26:84:2D:00:6A:F3:73:06:CC:80:B
Alias name: atostrustedroot2011
  SHA1: 2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
  SHA256:
 F3:56:BE:A2:44:B7:A9:1E:B3:5D:53:CA:9A:D7:86:4A:CE:01:8E:2D:35:D5:F8:F9:6D:DF:68:A6:F4:1A:A4:7
Alias name: autoridaddecertificacionfirmaprofesionalcifa62634068
  SHA1: AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
```

```
   SHA256:
 04:04:80:28:BF:1F:28:64:D4:8F:9A:D4:D8:32:94:36:6A:82:88:56:55:3F:3B:14:30:3F:90:14:7F:5D:40:E
Alias name: baltimorecodesigningca
   SHA1: 30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D
   SHA256:
 A9:15:45:DB:D2:E1:9C:4C:CD:F9:09:AA:71:90:0D:18:C7:35:1C:89:B3:15:F0:F1:3D:05:C1:3A:8F:FB:46:8
Alias name: baltimorecybertrustca
   SHA1: D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
   SHA256:
 16:AF:57:A9:F6:76:B0:AB:12:60:95:AA:5E:BA:DE:F2:2A:B3:11:19:D6:44:AC:95:CD:4B:93:DB:F3:F2:6A:E
Alias name: baltimorecybertrustroot
   SHA1: D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
   SHA256:
 16:AF:57:A9:F6:76:B0:AB:12:60:95:AA:5E:BA:DE:F2:2A:B3:11:19:D6:44:AC:95:CD:4B:93:DB:F3:F2:6A:E
Alias name: buypassclass2ca
   SHA1: 49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
   SHA256:
 9A:11:40:25:19:7C:5B:B9:5D:94:E6:3D:55:CD:43:79:08:47:B6:46:B2:3C:DF:11:AD:A4:A0:0E:FF:15:FB:4
Alias name: buypassclass2rootca
   SHA1: 49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
   SHA256:
 9A:11:40:25:19:7C:5B:B9:5D:94:E6:3D:55:CD:43:79:08:47:B6:46:B2:3C:DF:11:AD:A4:A0:0E:FF:15:FB:4
Alias name: buypassclass3ca
   SHA1: DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
   SHA256:
 ED:F7:EB:BC:A2:7A:2A:38:4D:38:7B:7D:40:10:C6:66:E2:ED:B4:84:3E:4C:29:B4:AE:1D:5B:93:32:E6:B2:4
Alias name: buypassclass3rootca
   SHA1: DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
   SHA256:
 ED:F7:EB:BC:A2:7A:2A:38:4D:38:7B:7D:40:10:C6:66:E2:ED:B4:84:3E:4C:29:B4:AE:1D:5B:93:32:E6:B2:4
Alias name: cadisigrootr2
   SHA1: B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
   SHA256:
 E2:3D:4A:03:6D:7B:70:E9:F5:95:B1:42:20:79:D2:B9:1E:DF:BB:1F:B6:51:A0:63:3E:AA:8A:9D:C5:F8:07:0
Alias name: camerfirmachambersca
   SHA1: 78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
   SHA256:
 06:3E:4A:FA:C4:91:DF:D3:32:F3:08:9B:85:42:E9:46:17:D8:93:D7:FE:94:4E:10:A7:93:7E:E2:9D:96:93:C
Alias name: camerfirmachamberscommerceca
   SHA1: 6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
   SHA256:
 0C:25:8A:12:A5:67:4A:EF:25:F2:8B:A7:DC:FA:EC:EE:A3:48:E5:41:E6:F5:CC:4E:E6:3B:71:B3:61:60:6A:C
Alias name: camerfirmachambersignca
   SHA1: 4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
```

```
   SHA256:
 13:63:35:43:93:34:A7:69:80:16:A0:D3:24:DE:72:28:4E:07:9D:7B:52:20:BB:8F:BD:74:78:16:EE:BE:BA:C
Alias name: certigna
   SHA1: B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
   SHA256:
 E3:B6:A2:DB:2E:D7:CE:48:84:2F:7A:C5:32:41:C7:B7:1D:54:14:4B:FB:40:C1:1F:3F:1D:0B:42:F5:EE:A1:2
Alias name: certignarootca
   SHA1: 2D:0D:52:14:FF:9E:AD:99:24:01:74:20:47:6E:6C:85:27:27:F5:43
   SHA256:
 D4:8D:3D:23:EE:DB:50:A4:59:E5:51:97:60:1C:27:77:4B:9D:7B:18:C9:4D:5A:05:95:11:A1:02:50:B9:31:6
Alias name: certplusclass2primaryca
   SHA1: 74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
   SHA256:
 0F:99:3C:8A:EF:97:BA:AF:56:87:14:0E:D5:9A:D1:82:1B:B4:AF:AC:F0:AA:9A:58:B5:D5:7A:33:8A:3A:FB:C
Alias name: certplusclass3pprimaryca
   SHA1: 21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
   SHA256:
 CC:C8:94:89:37:1B:AD:11:1C:90:61:9B:EA:24:0A:2E:6D:AD:D9:9F:9F:6E:1D:4D:41:E5:8E:D6:DE:3D:02:8
Alias name: certsignrootca
   SHA1: FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
   SHA256:
 EA:A9:62:C4:FA:4A:6B:AF:EB:E4:15:19:6D:35:1C:CD:88:8D:4F:53:F3:FA:8A:E6:D7:C4:66:A9:4E:60:42:B
Alias name: certsignrootcag2
   SHA1: 26:F9:93:B4:ED:3D:28:27:B0:B9:4B:A7:E9:15:1D:A3:8D:92:E5:32
   SHA256:
 65:7C:FE:2F:A7:3F:AA:38:46:25:71:F3:32:A2:36:3A:46:FC:E7:02:09:51:71:07:02:CD:FB:B6:EE:DA:33:0
Alias name: certum2
   SHA1: D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
   SHA256:
 B6:76:F2:ED:DA:E8:77:5C:D3:6C:B0:F6:3C:D1:D4:60:39:61:F4:9E:62:65:BA:01:3A:2F:03:07:B6:D0:B8:0
Alias name: certumca
   SHA1: 62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
   SHA256:
 D8:E0:FE:BC:1D:B2:E3:8D:00:94:0F:37:D2:7D:41:34:4D:99:3E:73:4B:99:D5:65:6D:97:78:D4:D8:14:36:2
Alias name: certumtrustednetworkca
   SHA1: 07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
   SHA256:
 5C:58:46:8D:55:F5:8E:49:7E:74:39:82:D2:B5:00:10:B6:D1:65:37:4A:CF:83:A7:D4:A3:2D:B7:68:C4:40:8
Alias name: certumtrustednetworkca2
   SHA1: D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
   SHA256:
 B6:76:F2:ED:DA:E8:77:5C:D3:6C:B0:F6:3C:D1:D4:60:39:61:F4:9E:62:65:BA:01:3A:2F:03:07:B6:D0:B8:0
Alias name: cfcaevroot
   SHA1: E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
```

```
  SHA256:
 5C:C3:D7:8E:4E:1D:5E:45:54:7A:04:E6:87:3E:64:F9:0C:F9:53:6D:1C:CC:2E:F8:00:F3:55:C4:C5:FD:70:F
Alias name: chambersofcommerceroot2008
  SHA1: 78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
  SHA256:
 06:3E:4A:FA:C4:91:DF:D3:32:F3:08:9B:85:42:E9:46:17:D8:93:D7:FE:94:4E:10:A7:93:7E:E2:9D:96:93:C
Alias name: chunghwaepkirootca
  SHA1: 67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
  SHA256:
 C0:A6:F4:DC:63:A2:4B:FD:CF:54:EF:2A:6A:08:2A:0A:72:DE:35:80:3E:2F:F5:FF:52:7A:E5:D8:72:06:DF:D
Alias name: cia-crt-g3-01-ca
  SHA1: 2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
  SHA256:
 20:48:AD:4C:EC:90:7F:FA:4A:15:D4:CE:45:E3:C8:E4:2C:EA:78:33:DC:C7:D3:40:48:FC:60:47:27:42:99:E
Alias name: cia-crt-g3-02-ca
  SHA1: 96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
  SHA256:
 93:F1:72:FB:BA:43:31:5C:06:EE:0F:9F:04:89:B8:F6:88:BC:75:15:3C:BE:B4:80:AC:A7:14:3A:F6:FC:4A:C
Alias name: comodo-ca
  SHA1: AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
  SHA256:
 52:F0:E1:C4:E5:8E:C6:29:29:1B:60:31:7F:07:46:71:B8:5D:7E:A8:0D:5B:07:27:34:63:53:4B:32:B4:02:3
Alias name: comodoaaaca
  SHA1: D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
  SHA256:
 D7:A7:A0:FB:5D:7E:27:31:D7:71:E9:48:4E:BC:DE:F7:1D:5F:0C:3E:0A:29:48:78:2B:C8:3E:E0:EA:69:9E:F
Alias name: comodoaaaservicesroot
  SHA1: D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
  SHA256:
 D7:A7:A0:FB:5D:7E:27:31:D7:71:E9:48:4E:BC:DE:F7:1D:5F:0C:3E:0A:29:48:78:2B:C8:3E:E0:EA:69:9E:F
Alias name: comodocertificationauthority
  SHA1: 66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
  SHA256:
 0C:2C:D6:3D:F7:80:6F:A3:99:ED:E8:09:11:6B:57:5B:F8:79:89:F0:65:18:F9:80:8C:86:05:03:17:8B:AF:6
Alias name: comodoecccertificationauthority
  SHA1: 9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
  SHA256:
 17:93:92:7A:06:14:54:97:89:AD:CE:2F:8F:34:F7:F0:B6:6D:0F:3A:E3:A3:B8:4D:21:EC:15:DB:BA:4F:AD:C
Alias name: comodorsacertificationauthority
  SHA1: AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
  SHA256:
 52:F0:E1:C4:E5:8E:C6:29:29:1B:60:31:7F:07:46:71:B8:5D:7E:A8:0D:5B:07:27:34:63:53:4B:32:B4:02:3
Alias name: cybertrustglobalroot
  SHA1: 5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
```

```
   SHA256:
  96:0A:DF:00:63:E9:63:56:75:0C:29:65:DD:0A:08:67:DA:0B:9C:BD:6E:77:71:4A:EA:FB:23:49:AB:39:3D:A
 Alias name: deprecateditsecca
   SHA1: 12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
   SHA256:
  9A:59:DA:86:24:1A:FD:BA:A3:39:FA:9C:FD:21:6A:0B:06:69:4D:E3:7E:37:52:6B:BE:63:C8:BC:83:74:2E:C
 Alias name: deutschetelekomrootca2
   SHA1: 85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
   SHA256:
  B6:19:1A:50:D0:C3:97:7F:7D:A9:9B:CD:AA:C8:6A:22:7D:AE:B9:67:9E:C7:0B:A3:B0:C9:D9:22:71:C1:70:D
 Alias name: digicertassuredidrootca
   SHA1: 05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
   SHA256:
  3E:90:99:B5:01:5E:8F:48:6C:00:BC:EA:9D:11:1E:E7:21:FA:BA:35:5A:89:BC:F1:DF:69:56:1E:3D:C6:32:5
 Alias name: digicertassuredidrootg2
   SHA1: A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F
   SHA256:
  7D:05:EB:B6:82:33:9F:8C:94:51:EE:09:4E:EB:FE:FA:79:53:A1:14:ED:B2:F4:49:49:45:2F:AB:7D:2F:C1:8
 Alias name: digicertassuredidrootg3
   SHA1: F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
   SHA256:
  7E:37:CB:8B:4C:47:09:0C:AB:36:55:1B:A6:F4:5D:B8:40:68:0F:BA:16:6A:95:2D:B1:00:71:7F:43:05:3F:C
 Alias name: digicertglobalrootca
   SHA1: A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
   SHA256:
  43:48:A0:E9:44:4C:78:CB:26:5E:05:8D:5E:89:44:B4:D8:4F:96:62:BD:26:DB:25:7F:89:34:A4:43:C7:01:6
 Alias name: digicertglobalrootg2
   SHA1: DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
   SHA256:
  CB:3C:CB:B7:60:31:E5:E0:13:8F:8D:D3:9A:23:F9:DE:47:FF:C3:5E:43:C1:14:4C:EA:27:D4:6A:5A:B1:CB:5
 Alias name: digicertglobalrootg3
   SHA1: 7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E
   SHA256:
  31:AD:66:48:F8:10:41:38:C7:38:F3:9E:A4:32:01:33:39:3E:3A:18:CC:02:29:6E:F9:7C:2A:C9:EF:67:31:D
 Alias name: digicerthighassuranceevrootca
   SHA1: 5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
   SHA256:
  74:31:E5:F4:C3:C1:CE:46:90:77:4F:0B:61:E0:54:40:88:3B:A9:A0:1E:D0:0B:A6:AB:D7:80:6E:D3:B1:18:C
 Alias name: digicerttrustedrootg4
   SHA1: DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
   SHA256:
  55:2F:7B:DC:F1:A7:AF:9E:6C:E6:72:01:7F:4F:12:AB:F7:72:40:C7:8E:76:1A:C2:03:D1:D9:D2:0A:C8:99:8
 Alias name: dstrootcax3
   SHA1: DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
```

```
   SHA256:
 06:87:26:03:31:A7:24:03:D9:09:F1:05:E6:9B:CF:0D:32:E1:BD:24:93:FF:C6:D9:20:6D:11:BC:D6:77:07:3
Alias name: dtrustrootclass3ca22009
   SHA1: 58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
   SHA256:
 49:E7:A4:42:AC:F0:EA:62:87:05:00:54:B5:25:64:B6:50:E4:F4:9E:42:E3:48:D6:AA:38:E0:39:E9:57:B1:C
Alias name: dtrustrootclass3ca2ev2009
   SHA1: 96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
   SHA256:
 EE:C5:49:6B:98:8C:E9:86:25:B9:34:09:2E:EC:29:08:BE:D0:B0:F3:16:C2:D4:73:0C:84:EA:F1:F3:D3:48:8
Alias name: ecacc
   SHA1: 28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:F2:E8
   SHA256:
 88:49:7F:01:60:2F:31:54:24:6A:E2:8C:4D:5A:EF:10:F1:D8:7E:BB:76:62:6F:4A:E0:B7:F9:5B:A7:96:87:9
Alias name: emsigneccrootcac3
   SHA1: B6:AF:43:C2:9B:81:53:7D:F6:EF:6B:C3:1F:1F:60:15:0C:EE:48:66
   SHA256:
 BC:4D:80:9B:15:18:9D:78:DB:3E:1D:8C:F4:F9:72:6A:79:5D:A1:64:3C:A5:F1:35:8E:1D:DB:0E:DC:0D:7E:B
Alias name: emsigneccrootcag3
   SHA1: 30:43:FA:4F:F2:57:DC:A0:C3:80:EE:2E:58:EA:78:B2:3F:E6:BB:C1
   SHA256:
 86:A1:EC:BA:08:9C:4A:8D:3B:BE:27:34:C6:12:BA:34:1D:81:3E:04:3C:F9:E8:A8:62:CD:5C:57:A3:6B:BE:6
Alias name: emsignrootcac1
   SHA1: E7:2E:F1:DF:FC:B2:09:28:CF:5D:D4:D5:67:37:B1:51:CB:86:4F:01
   SHA256:
 12:56:09:AA:30:1D:A0:A2:49:B9:7A:82:39:CB:6A:34:21:6F:44:DC:AC:9F:39:54:B1:42:92:F2:E8:C8:60:8
Alias name: emsignrootcag1
   SHA1: 8A:C7:AD:8F:73:AC:4E:C1:B5:75:4D:A5:40:F4:FC:CF:7C:B5:8E:8C
   SHA256:
 40:F6:AF:03:46:A9:9A:A1:CD:1D:55:5A:4E:9C:CE:62:C7:F9:63:46:03:EE:40:66:15:83:3D:C8:C8:D0:03:6
Alias name: entrust2048ca
   SHA1: 50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
   SHA256:
 6D:C4:71:72:E0:1C:BC:B0:BF:62:58:0D:89:5F:E2:B8:AC:9A:D4:F8:73:80:1E:0C:10:B9:C8:37:D2:1E:B1:7
Alias name: entrustevca
   SHA1: B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
   SHA256:
 73:C1:76:43:4F:1B:C6:D5:AD:F4:5B:0E:76:E7:27:28:7C:8D:E5:76:16:C1:E6:E6:14:1A:2B:2C:BC:7D:8E:4
Alias name: entrustnetpremium2048secureserverca
   SHA1: 50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
   SHA256:
 6D:C4:71:72:E0:1C:BC:B0:BF:62:58:0D:89:5F:E2:B8:AC:9A:D4:F8:73:80:1E:0C:10:B9:C8:37:D2:1E:B1:7
Alias name: entrustrootcag2
   SHA1: 8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
```

```
  SHA256:
 43:DF:57:74:B0:3E:7F:EF:5F:E4:0D:93:1A:7B:ED:F1:BB:2E:6B:42:73:8C:4E:6D:38:41:10:3D:3A:A7:F3:3
Alias name: entrustrootcertificationauthority
  SHA1: B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
  SHA256:
 73:C1:76:43:4F:1B:C6:D5:AD:F4:5B:0E:76:E7:27:28:7C:8D:E5:76:16:C1:E6:E6:14:1A:2B:2C:BC:7D:8E:4
Alias name: entrustrootcertificationauthorityec1
  SHA1: 20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
  SHA256:
 02:ED:0E:B2:8C:14:DA:45:16:5C:56:67:91:70:0D:64:51:D7:FB:56:F0:B2:AB:1D:3B:8E:B0:70:E5:6E:DF:F
Alias name: entrustrootcertificationauthorityg2
  SHA1: 8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
  SHA256:
 43:DF:57:74:B0:3E:7F:EF:5F:E4:0D:93:1A:7B:ED:F1:BB:2E:6B:42:73:8C:4E:6D:38:41:10:3D:3A:A7:F3:3
Alias name: entrustrootcertificationauthorityg4
  SHA1: 14:88:4E:86:26:37:B0:26:AF:59:62:5C:40:77:EC:35:29:BA:96:01
  SHA256:
 DB:35:17:D1:F6:73:2A:2D:5A:B9:7C:53:3E:C7:07:79:EE:32:70:A6:2F:B4:AC:42:38:37:24:60:E6:F0:1E:8
Alias name: epkirootcertificationauthority
  SHA1: 67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
  SHA256:
 C0:A6:F4:DC:63:A2:4B:FD:CF:54:EF:2A:6A:08:2A:0A:72:DE:35:80:3E:2F:F5:FF:52:7A:E5:D8:72:06:DF:D
Alias name: equifaxsecureebusinessca1
  SHA1: AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
  SHA256:
 2E:3A:2B:B5:11:25:05:83:6C:A8:96:8B:E2:CB:37:27:CE:9B:56:84:5C:6E:E9:8E:91:85:10:4A:FB:9A:F5:9
Alias name: equifaxsecureglobalebusinessca1
  SHA1: 3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
  SHA256:
 86:AB:5A:65:71:D3:32:9A:BC:D2:E4:E6:37:66:8B:A8:9C:73:1E:C2:93:B6:CB:A6:0F:71:63:40:A0:91:CE:A
Alias name: eszignorootca2017
  SHA1: 89:D4:83:03:4F:9E:9A:48:80:5F:72:37:D4:A9:A6:EF:CB:7C:1F:D1
  SHA256:
 BE:B0:0B:30:83:9B:9B:C3:2C:32:E4:44:79:05:95:06:41:F2:64:21:B1:5E:D0:89:19:8B:51:8A:E2:EA:1B:9
Alias name: etugracertificationauthority
  SHA1: 51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
  SHA256:
 B0:BF:D5:2B:B0:D7:D9:BD:92:BF:5D:4D:C1:3D:A2:55:C0:2C:54:2F:37:83:65:EA:89:39:11:F5:5E:55:F2:3
Alias name: gd-class2-root.pem
  SHA1: 27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
  SHA256:
 C3:84:6B:F2:4B:9E:93:CA:64:27:4C:0E:C6:7C:1E:CC:5E:02:4F:FC:AC:D2:D7:40:19:35:0E:81:FE:54:6A:E
Alias name: gd_bundle-g2.pem
  SHA1: 27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8
```

```
  SHA256:
 97:3A:41:27:6F:FD:01:E0:27:A2:AA:D4:9E:34:C3:78:46:D3:E9:76:FF:6A:62:0B:67:12:E3:38:32:04:1A:A
Alias name: gdcatrustauthr5root
  SHA1: 0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
  SHA256:
 BF:FF:8F:D0:44:33:48:7D:6A:8A:A6:0C:1A:29:76:7A:9F:C2:BB:B0:5E:42:0F:71:3A:13:B9:92:89:1D:38:9
Alias name: gdroot-g2.pem
  SHA1: 47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
  SHA256:
 45:14:0B:32:47:EB:9C:C8:C5:B4:F0:D7:B5:30:91:F7:32:92:08:9E:6E:5A:63:E2:74:9D:D3:AC:A9:19:8E:D
Alias name: geotrustglobalca
  SHA1: DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
  SHA256:
 FF:85:6A:2D:25:1D:CD:88:D3:66:56:F4:50:12:67:98:CF:AB:AA:DE:40:79:9C:72:2D:E4:D2:B5:DB:36:A7:3
Alias name: geotrustprimaryca
  SHA1: 32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
  SHA256:
 37:D5:10:06:C5:12:EA:AB:62:64:21:F1:EC:8C:92:01:3F:C5:F8:2A:E9:8E:E5:33:EB:46:19:B8:DE:B4:D0:6
Alias name: geotrustprimarycag2
  SHA1: 8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
  SHA256:
 5E:DB:7A:C4:3B:82:A0:6A:87:61:E8:D7:BE:49:79:EB:F2:61:1F:7D:D7:9B:F9:1C:1C:6B:56:6A:21:9E:D7:6
Alias name: geotrustprimarycag3
  SHA1: 03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
  SHA256:
 B4:78:B8:12:25:0D:F8:78:63:5C:2A:A7:EC:7D:15:5E:AA:62:5E:E8:29:16:E2:CD:29:43:61:88:6C:D1:FB:D
Alias name: geotrustprimarycertificationauthority
  SHA1: 32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
  SHA256:
 37:D5:10:06:C5:12:EA:AB:62:64:21:F1:EC:8C:92:01:3F:C5:F8:2A:E9:8E:E5:33:EB:46:19:B8:DE:B4:D0:6
Alias name: geotrustprimarycertificationauthorityg2
  SHA1: 8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
  SHA256:
 5E:DB:7A:C4:3B:82:A0:6A:87:61:E8:D7:BE:49:79:EB:F2:61:1F:7D:D7:9B:F9:1C:1C:6B:56:6A:21:9E:D7:6
Alias name: geotrustprimarycertificationauthorityg3
  SHA1: 03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
  SHA256:
 B4:78:B8:12:25:0D:F8:78:63:5C:2A:A7:EC:7D:15:5E:AA:62:5E:E8:29:16:E2:CD:29:43:61:88:6C:D1:FB:D
Alias name: geotrustuniversalca
  SHA1: E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
  SHA256:
 A0:45:9B:9F:63:B2:25:59:F5:FA:5D:4C:6D:B3:F9:F7:2F:F1:93:42:03:35:78:F0:73:BF:1D:1B:46:CB:B9:1
Alias name: geotrustuniversalca2
  SHA1: 37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
```

```
   SHA256:
  A0:23:4F:3B:C8:52:7C:A5:62:8E:EC:81:AD:5D:69:89:5D:A5:68:0D:C9:1D:1C:B8:47:7F:33:F8:78:B9:5B:0
 Alias name: globalchambersignroot2008
   SHA1: 4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
   SHA256:
  13:63:35:43:93:34:A7:69:80:16:A0:D3:24:DE:72:28:4E:07:9D:7B:52:20:BB:8F:BD:74:78:16:EE:BE:BA:C
 Alias name: globalsignca
   SHA1: B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
   SHA256:
  EB:D4:10:40:E4:BB:3E:C7:42:C9:E3:81:D3:1E:F2:A4:1A:48:B6:68:5C:96:E7:CE:F3:C1:DF:6C:D4:33:1C:9
 Alias name: globalsigneccrootcar4
   SHA1: 69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
   SHA256:
  BE:C9:49:11:C2:95:56:76:DB:6C:0A:55:09:86:D7:6E:3B:A0:05:66:7C:44:2C:97:62:B4:FB:B7:73:DE:22:8
 Alias name: globalsigneccrootcar5
   SHA1: 1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
   SHA256:
  17:9F:BC:14:8A:3D:D0:0F:D2:4E:A1:34:58:CC:43:BF:A7:F5:9C:81:82:D7:83:A5:13:F6:EB:EC:10:0C:89:2
 Alias name: globalsignr2ca
   SHA1: 75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
   SHA256:
  CA:42:DD:41:74:5F:D0:B8:1E:B9:02:36:2C:F9:D8:BF:71:9D:A1:BD:1B:1E:FC:94:6F:5B:4C:99:F4:2C:1B:9
 Alias name: globalsignr3ca
   SHA1: D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
   SHA256:
  CB:B5:22:D7:B7:F1:27:AD:6A:01:13:86:5B:DF:1C:D4:10:2E:7D:07:59:AF:63:5A:7C:F4:72:0D:C9:63:C5:3
 Alias name: globalsignrootca
   SHA1: B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
   SHA256:
  EB:D4:10:40:E4:BB:3E:C7:42:C9:E3:81:D3:1E:F2:A4:1A:48:B6:68:5C:96:E7:CE:F3:C1:DF:6C:D4:33:1C:9
 Alias name: globalsignrootcar2
   SHA1: 75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
   SHA256:
  CA:42:DD:41:74:5F:D0:B8:1E:B9:02:36:2C:F9:D8:BF:71:9D:A1:BD:1B:1E:FC:94:6F:5B:4C:99:F4:2C:1B:9
 Alias name: globalsignrootcar3
   SHA1: D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
   SHA256:
  CB:B5:22:D7:B7:F1:27:AD:6A:01:13:86:5B:DF:1C:D4:10:2E:7D:07:59:AF:63:5A:7C:F4:72:0D:C9:63:C5:3
 Alias name: globalsignrootcar6
   SHA1: 80:94:64:0E:B5:A7:A1:CA:11:9C:1F:DD:D5:9F:81:02:63:A7:FB:D1
   SHA256:
  2C:AB:EA:FE:37:D0:6C:A2:2A:BA:73:91:C0:03:3D:25:98:29:52:C4:53:64:73:49:76:3A:3A:B5:AD:6C:CF:6
 Alias name: godaddyclass2ca
   SHA1: 27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
```

```
  SHA256:
 C3:84:6B:F2:4B:9E:93:CA:64:27:4C:0E:C6:7C:1E:CC:5E:02:4F:FC:AC:D2:D7:40:19:35:0E:81:FE:54:6A:E
Alias name: godaddyrootcertificateauthorityg2
  SHA1: 47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
  SHA256:
 45:14:0B:32:47:EB:9C:C8:C5:B4:F0:D7:B5:30:91:F7:32:92:08:9E:6E:5A:63:E2:74:9D:D3:AC:A9:19:8E:D
Alias name: godaddyrootg2ca
  SHA1: 47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
  SHA256:
 45:14:0B:32:47:EB:9C:C8:C5:B4:F0:D7:B5:30:91:F7:32:92:08:9E:6E:5A:63:E2:74:9D:D3:AC:A9:19:8E:D
Alias name: gtsrootr1
  SHA1: E1:C9:50:E6:EF:22:F8:4C:56:45:72:8B:92:20:60:D7:D5:A7:A3:E8
  SHA256:
 2A:57:54:71:E3:13:40:BC:21:58:1C:BD:2C:F1:3E:15:84:63:20:3E:CE:94:BC:F9:D3:CC:19:6B:F0:9A:54:7
Alias name: gtsrootr2
  SHA1: D2:73:96:2A:2A:5E:39:9F:73:3F:E1:C7:1E:64:3F:03:38:34:FC:4D
  SHA256:
 C4:5D:7B:B0:8E:6D:67:E6:2E:42:35:11:0B:56:4E:5F:78:FD:92:EF:05:8C:84:0A:EA:4E:64:55:D7:58:5C:6
Alias name: gtsrootr3
  SHA1: 30:D4:24:6F:07:FF:DB:91:89:8A:0B:E9:49:66:11:EB:8C:5E:46:E5
  SHA256:
 15:D5:B8:77:46:19:EA:7D:54:CE:1C:A6:D0:B0:C4:03:E0:37:A9:17:F1:31:E8:A0:4E:1E:6B:7A:71:BA:BC:E
Alias name: gtsrootr4
  SHA1: 2A:1D:60:27:D9:4A:B1:0A:1C:4D:91:5C:CD:33:A0:CB:3E:2D:54:CB
  SHA256:
 71:CC:A5:39:1F:9E:79:4B:04:80:25:30:B3:63:E1:21:DA:8A:30:43:BB:26:66:2F:EA:4D:CA:7F:C9:51:A4:B
Alias name: hellenicacademicandresearchinstitutionseccrootca2015
  SHA1: 9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
  SHA256:
 44:B5:45:AA:8A:25:E6:5A:73:CA:15:DC:27:FC:36:D2:4C:1C:B9:95:3A:06:65:39:B1:15:82:DC:48:7B:48:3
Alias name: hellenicacademicandresearchinstitutionsrootca2011
  SHA1: FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
  SHA256:
 BC:10:4F:15:A4:8B:E7:09:DC:A5:42:A7:E1:D4:B9:DF:6F:05:45:27:E8:02:EA:A9:2D:59:54:44:25:8A:FE:7
Alias name: hellenicacademicandresearchinstitutionsrootca2015
  SHA1: 01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
  SHA256:
 A0:40:92:9A:02:CE:53:B4:AC:F4:F2:FF:C6:98:1C:E4:49:6F:75:5E:6D:45:FE:0B:2A:69:2B:CD:52:52:3F:3
Alias name: hongkongpostrootca1
  SHA1: D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
  SHA256:
 F9:E6:7D:33:6C:51:00:2A:C0:54:C6:32:02:2D:66:DD:A2:E7:E3:FF:F1:0A:D0:61:ED:31:D8:BB:B4:10:CF:B
Alias name: hongkongpostrootca3
  SHA1: 58:A2:D0:EC:20:52:81:5B:C1:F3:F8:64:02:24:4E:C2:8E:02:4B:02
```

```
    SHA256:
  5A:2F:C0:3F:0C:83:B0:90:BB:FA:40:60:4B:09:88:44:6C:76:36:18:3D:F9:84:6E:17:10:1A:44:7F:B8:EF:D
 Alias name: identrustcommercialrootca1
    SHA1: DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
    SHA256:
  5D:56:49:9B:E4:D2:E0:8B:CF:CA:D0:8A:3E:38:72:3D:50:50:3B:DE:70:69:48:E4:2F:55:60:30:19:E5:28:A
 Alias name: identrustpublicsectorrootca1
    SHA1: BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
    SHA256:
  30:D0:89:5A:9A:44:8A:26:20:91:63:55:22:D1:F5:20:10:B5:86:7A:CA:E1:2C:78:EF:95:8F:D4:F4:38:9F:2
 Alias name: isrgrootx1
    SHA1: CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
    SHA256:
  96:BC:EC:06:26:49:76:F3:74:60:77:9A:CF:28:C5:A7:CF:E8:A3:C0:AA:E1:1A:8F:FC:EE:05:C0:BD:DF:08:C
 Alias name: izenpecom
    SHA1: 2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
    SHA256:
  25:30:CC:8E:98:32:15:02:BA:D9:6F:9B:1F:BA:1B:09:9E:2D:29:9E:0F:45:48:BB:91:4F:36:3B:C0:D4:53:1
 Alias name: keynectisrootca
    SHA1: 9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
    SHA256:
  42:10:F1:99:49:9A:9A:C3:3C:8D:E0:2B:A6:DB:AA:14:40:8B:DD:8A:6E:32:46:89:C1:92:2D:06:97:15:A3:3
 Alias name: microseceszignorootca2009
    SHA1: 89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
    SHA256:
  3C:5F:81:FE:A5:FA:B8:2C:64:BF:A2:EA:EC:AF:CD:E8:E0:77:FC:86:20:A7:CA:E5:37:16:3D:F3:6E:DB:F3:7
 Alias name: mozillacert0.pem
    SHA1: 97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
    SHA256:
  A5:31:25:18:8D:21:10:AA:96:4B:02:C7:B7:C6:DA:32:03:17:08:94:E5:FB:71:FF:FB:66:67:D5:E6:81:0A:3
 Alias name: mozillacert1.pem
    SHA1: 23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
    SHA256:
  B4:41:0B:73:E2:E6:EA:CA:47:FB:C4:2F:8F:A4:01:8A:F4:38:1D:C5:4C:FA:A8:44:50:46:1E:ED:09:45:4D:E
 Alias name: mozillacert10.pem
    SHA1: 5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
    SHA256:
  21:DB:20:12:36:60:BB:2E:D4:18:20:5D:A1:1E:E7:A8:5A:65:E2:BC:6E:55:B5:AF:7E:78:99:C8:A2:66:D9:2
 Alias name: mozillacert100.pem
    SHA1: 58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
    SHA256:
  49:E7:A4:42:AC:F0:EA:62:87:05:00:54:B5:25:64:B6:50:E4:F4:9E:42:E3:48:D6:AA:38:E0:39:E9:57:B1:C
 Alias name: mozillacert101.pem
    SHA1: 99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
```

```
  SHA256:
 62:F2:40:27:8C:56:4C:4D:D8:BF:7D:9D:4F:6F:36:6E:A8:94:D2:2F:5F:34:D9:89:A9:83:AC:EC:2F:FF:ED:5
Alias name: mozillacert102.pem
  SHA1: 96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
  SHA256:
 EE:C5:49:6B:98:8C:E9:86:25:B9:34:09:2E:EC:29:08:BE:D0:B0:F3:16:C2:D4:73:0C:84:EA:F1:F3:D3:48:8
Alias name: mozillacert103.pem
  SHA1: 70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
  SHA256:
 3C:FC:3C:14:D1:F6:84:FF:17:E3:8C:43:CA:44:0C:00:B9:67:EC:93:3E:8B:FE:06:4C:A1:D7:2C:90:F2:AD:B
Alias name: mozillacert104.pem
  SHA1: 4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E
  SHA256:
 1C:01:C6:F4:DB:B2:FE:FC:22:55:8B:2B:CA:32:56:3F:49:84:4A:CF:C3:2B:7B:E4:B0:FF:59:9F:9E:8C:7A:F
Alias name: mozillacert105.pem
  SHA1: 77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC
  SHA256:
 F0:9B:12:2C:71:14:F4:A0:9B:D4:EA:4F:4A:99:D5:58:B4:6E:4C:25:CD:81:14:0D:29:C0:56:13:91:4C:38:4
Alias name: mozillacert106.pem
  SHA1: E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
  SHA256:
 D9:5F:EA:3C:A4:EE:DC:E7:4C:D7:6E:75:FC:6D:1F:F6:2C:44:1F:0F:A8:BC:77:F0:34:B1:9E:5D:B2:58:01:5
Alias name: mozillacert107.pem
  SHA1: 8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
  SHA256:
 F9:6F:23:F4:C3:E7:9C:07:7A:46:98:8D:5A:F5:90:06:76:A0:F0:39:CB:64:5D:D1:75:49:B2:16:C8:24:40:C
Alias name: mozillacert108.pem
  SHA1: B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
  SHA256:
 EB:D4:10:40:E4:BB:3E:C7:42:C9:E3:81:D3:1E:F2:A4:1A:48:B6:68:5C:96:E7:CE:F3:C1:DF:6C:D4:33:1C:9
Alias name: mozillacert109.pem
  SHA1: B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
  SHA256:
 E2:3D:4A:03:6D:7B:70:E9:F5:95:B1:42:20:79:D2:B9:1E:DF:BB:1F:B6:51:A0:63:3E:AA:8A:9D:C5:F8:07:0
Alias name: mozillacert11.pem
  SHA1: 05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
  SHA256:
 3E:90:99:B5:01:5E:8F:48:6C:00:BC:EA:9D:11:1E:E7:21:FA:BA:35:5A:89:BC:F1:DF:69:56:1E:3D:C6:32:5
Alias name: mozillacert110.pem
  SHA1: 93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
  SHA256:
 9A:6E:C0:12:E1:A7:DA:9D:BE:34:19:4D:47:8A:D7:C0:DB:18:22:FB:07:1D:F1:29:81:49:6E:D1:04:38:41:1
Alias name: mozillacert111.pem
  SHA1: 9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
```

```
    SHA256:
  59:76:90:07:F7:68:5D:0F:CD:50:87:2F:9F:95:D5:75:5A:5B:2B:45:7D:81:F3:69:2B:61:0A:98:67:2F:0E:1
Alias name: mozillacert112.pem
    SHA1: 43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
    SHA256:
  DD:69:36:FE:21:F8:F0:77:C1:23:A1:A5:21:C1:22:24:F7:22:55:B7:3E:03:A7:26:06:93:E8:A2:4B:0F:A3:8
Alias name: mozillacert113.pem
    SHA1: 50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
    SHA256:
  6D:C4:71:72:E0:1C:BC:B0:BF:62:58:0D:89:5F:E2:B8:AC:9A:D4:F8:73:80:1E:0C:10:B9:C8:37:D2:1E:B1:7
Alias name: mozillacert114.pem
    SHA1: 51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
    SHA256:
  B0:BF:D5:2B:B0:D7:D9:BD:92:BF:5D:4D:C1:3D:A2:55:C0:2C:54:2F:37:83:65:EA:89:39:11:F5:5E:55:F2:3
Alias name: mozillacert115.pem
    SHA1: 59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
    SHA256:
  91:E2:F5:78:8D:58:10:EB:A7:BA:58:73:7D:E1:54:8A:8E:CA:CD:01:45:98:BC:0B:14:3E:04:1B:17:05:25:5
Alias name: mozillacert116.pem
    SHA1: 2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
    SHA256:
  F3:56:BE:A2:44:B7:A9:1E:B3:5D:53:CA:9A:D7:86:4A:CE:01:8E:2D:35:D5:F8:F9:6D:DF:68:A6:F4:1A:A4:7
Alias name: mozillacert117.pem
    SHA1: D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
    SHA256:
  16:AF:57:A9:F6:76:B0:AB:12:60:95:AA:5E:BA:DE:F2:2A:B3:11:19:D6:44:AC:95:CD:4B:93:DB:F3:F2:6A:E
Alias name: mozillacert118.pem
    SHA1: 7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45
    SHA256:
  5F:0B:62:EA:B5:E3:53:EA:65:21:65:16:58:FB:B6:53:59:F4:43:28:0A:4A:FB:D1:04:D7:7D:10:F9:F0:4C:0
Alias name: mozillacert119.pem
    SHA1: 75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
    SHA256:
  CA:42:DD:41:74:5F:D0:B8:1E:B9:02:36:2C:F9:D8:BF:71:9D:A1:BD:1B:1E:FC:94:6F:5B:4C:99:F4:2C:1B:9
Alias name: mozillacert12.pem
    SHA1: A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
    SHA256:
  43:48:A0:E9:44:4C:78:CB:26:5E:05:8D:5E:89:44:B4:D8:4F:96:62:BD:26:DB:25:7F:89:34:A4:43:C7:01:6
Alias name: mozillacert120.pem
    SHA1: DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
    SHA256:
  CF:56:FF:46:A4:A1:86:10:9D:D9:65:84:B5:EE:B5:8A:51:0C:42:75:B0:E5:F9:4F:40:BB:AE:86:5E:19:F6:7
Alias name: mozillacert121.pem
    SHA1: CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
```

```
  SHA256:
 8C:72:09:27:9A:C0:4E:27:5E:16:D0:7F:D3:B7:75:E8:01:54:B5:96:80:46:E3:1F:52:DD:25:76:63:24:E9:A
Alias name: mozillacert122.pem
  SHA1: 02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
  SHA256:
 68:7F:A4:51:38:22:78:FF:F0:C8:B1:1F:8D:43:D5:76:67:1C:6E:B2:BC:EA:B4:13:FB:83:D9:65:D0:6D:2F:F
Alias name: mozillacert123.pem
  SHA1: 2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10:DD:6B:DF:99:72:2C:96:E5
  SHA256:
 07:91:CA:07:49:B2:07:82:AA:D3:C7:D7:BD:0C:DF:C9:48:58:35:84:3E:B2:D7:99:60:09:CE:43:AB:6C:69:2
Alias name: mozillacert124.pem
  SHA1: 4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
  SHA256:
 80:95:21:08:05:DB:4B:BC:35:5E:44:28:D8:FD:6E:C2:CD:E3:AB:5F:B9:7A:99:42:98:8E:B8:F4:DC:D0:60:1
Alias name: mozillacert125.pem
  SHA1: B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
  SHA256:
 73:C1:76:43:4F:1B:C6:D5:AD:F4:5B:0E:76:E7:27:28:7C:8D:E5:76:16:C1:E6:E6:14:1A:2B:2C:BC:7D:8E:4
Alias name: mozillacert126.pem
  SHA1: 25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
  SHA256:
 AF:8B:67:62:A1:E5:28:22:81:61:A9:5D:5C:55:9E:E2:66:27:8F:75:D7:9E:83:01:89:A5:03:50:6A:BD:6B:4
Alias name: mozillacert127.pem
  SHA1: DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
  SHA256:
 FF:85:6A:2D:25:1D:CD:88:D3:66:56:F4:50:12:67:98:CF:AB:AA:DE:40:79:9C:72:2D:E4:D2:B5:DB:36:A7:3
Alias name: mozillacert128.pem
  SHA1: A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D
  SHA256:
 CA:2D:82:A0:86:77:07:2F:8A:B6:76:4F:F0:35:67:6C:FE:3E:5E:32:5E:01:21:72:DF:3F:92:09:6D:B7:9B:8
Alias name: mozillacert129.pem
  SHA1: E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
  SHA256:
 A0:45:9B:9F:63:B2:25:59:F5:FA:5D:4C:6D:B3:F9:F7:2F:F1:93:42:03:35:78:F0:73:BF:1D:1B:46:CB:B9:1
Alias name: mozillacert13.pem
  SHA1: 06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
  SHA256:
 6C:61:DA:C3:A2:DE:F0:31:50:6B:E0:36:D2:A6:FE:40:19:94:FB:D1:3D:F9:C8:D4:66:59:92:74:C4:46:EC:9
Alias name: mozillacert130.pem
  SHA1: E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
  SHA256:
 F4:C1:49:55:1A:30:13:A3:5B:C7:BF:FE:17:A7:F3:44:9B:C1:AB:5B:5A:0A:E7:4B:06:C2:3B:90:00:4C:01:0
Alias name: mozillacert131.pem
  SHA1: 37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
```

```
  SHA256:
 A0:23:4F:3B:C8:52:7C:A5:62:8E:EC:81:AD:5D:69:89:5D:A5:68:0D:C9:1D:1C:B8:47:7F:33:F8:78:B9:5B:0
Alias name: mozillacert132.pem
  SHA1: 39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
  SHA256:
 77:40:73:12:C6:3A:15:3D:5B:C0:0B:4E:51:75:9C:DF:DA:C2:37:DC:2A:33:B6:79:46:E9:8E:9B:FA:68:0A:E
Alias name: mozillacert133.pem
  SHA1: 85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
  SHA256:
 7D:3B:46:5A:60:14:E5:26:C0:AF:FC:EE:21:27:D2:31:17:27:AD:81:1C:26:84:2D:00:6A:F3:73:06:CC:80:B
Alias name: mozillacert134.pem
  SHA1: 70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
  SHA256:
 69:FA:C9:BD:55:FB:0A:C7:8D:53:BB:EE:5C:F1:D5:97:98:9F:D0:AA:AB:20:A2:51:51:BD:F1:73:3E:E7:D1:2
Alias name: mozillacert135.pem
  SHA1: 62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
  SHA256:
 D8:E0:FE:BC:1D:B2:E3:8D:00:94:0F:37:D2:7D:41:34:4D:99:3E:73:4B:99:D5:65:6D:97:78:D4:D8:14:36:2
Alias name: mozillacert136.pem
  SHA1: D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
  SHA256:
 D7:A7:A0:FB:5D:7E:27:31:D7:71:E9:48:4E:BC:DE:F7:1D:5F:0C:3E:0A:29:48:78:2B:C8:3E:E0:EA:69:9E:F
Alias name: mozillacert137.pem
  SHA1: 4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A1:D1
  SHA256:
 BD:81:CE:3B:4F:65:91:D1:1A:67:B5:FC:7A:47:FD:EF:25:52:1B:F9:AA:4E:18:B9:E3:DF:2E:34:A7:80:3B:E
Alias name: mozillacert138.pem
  SHA1: E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
  SHA256:
 3F:06:E5:56:81:D4:96:F5:BE:16:9E:B5:38:9F:9F:2B:8F:F6:1E:17:08:DF:68:81:72:48:49:CD:5D:27:CB:6
Alias name: mozillacert139.pem
  SHA1: DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
  SHA256:
 A4:5E:DE:3B:BB:F0:9C:8A:E1:5C:72:EF:C0:72:68:D6:93:A2:1C:99:6F:D5:1E:67:CA:07:94:60:FD:6D:88:7
Alias name: mozillacert14.pem
  SHA1: 5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
  SHA256:
 74:31:E5:F4:C3:C1:CE:46:90:77:4F:0B:61:E0:54:40:88:3B:A9:A0:1E:D0:0B:A6:AB:D7:80:6E:D3:B1:18:C
Alias name: mozillacert140.pem
  SHA1: CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
  SHA256:
 85:A0:DD:7D:D7:20:AD:B7:FF:05:F8:3D:54:2B:20:9D:C7:FF:45:28:F7:D6:77:B1:83:89:FE:A5:E5:C4:9E:8
Alias name: mozillacert141.pem
  SHA1: 31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
```

```
  SHA256:
 58:D0:17:27:9C:D4:DC:63:AB:DD:B1:96:A6:C9:90:6C:30:C4:E0:87:83:EA:E8:C1:60:99:54:D6:93:55:59:6
Alias name: mozillacert142.pem
  SHA1: 1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
  SHA256:
 18:F1:FC:7F:20:5D:F8:AD:DD:EB:7F:E0:07:DD:57:E3:AF:37:5A:9C:4D:8D:73:54:6B:F4:F1:FE:D1:E1:8D:3
Alias name: mozillacert143.pem
  SHA1: 36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
  SHA256:
 E7:5E:72:ED:9F:56:0E:EC:6E:B4:80:00:73:A4:3F:C3:AD:19:19:5A:39:22:82:01:78:95:97:4A:99:02:6B:6
Alias name: mozillacert144.pem
  SHA1: 37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
  SHA256:
 79:08:B4:03:14:C1:38:10:0B:51:8D:07:35:80:7F:FB:FC:F8:51:8A:00:95:33:71:05:BA:38:6B:15:3D:D9:2
Alias name: mozillacert145.pem
  SHA1: 10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:3A:04
  SHA256:
 D4:1D:82:9E:8C:16:59:82:2A:F9:3F:CE:62:BF:FC:DE:26:4F:C8:4E:8B:95:0C:5F:F2:75:D0:52:35:46:95:A
Alias name: mozillacert146.pem
  SHA1: 21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A
  SHA256:
 48:98:C6:88:8C:0C:FF:B0:D3:E3:1A:CA:8A:37:D4:E3:51:5F:F7:46:D0:26:35:D8:66:46:CF:A0:A3:18:5A:E
Alias name: mozillacert147.pem
  SHA1: 58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
  SHA256:
 85:FB:2F:91:DD:12:27:5A:01:45:B6:36:53:4F:84:02:4A:D6:8B:69:B8:EE:88:68:4F:F7:11:37:58:05:B3:4
Alias name: mozillacert148.pem
  SHA1: 04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
  SHA256:
 6E:A5:47:41:D0:04:66:7E:ED:1B:48:16:63:4A:A3:A7:9E:6E:4B:96:95:0F:82:79:DA:FC:8D:9B:D8:81:21:3
Alias name: mozillacert149.pem
  SHA1: 6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
  SHA256:
 0C:25:8A:12:A5:67:4A:EF:25:F2:8B:A7:DC:FA:EC:EE:A3:48:E5:41:E6:F5:CC:4E:E6:3B:71:B3:61:60:6A:C
Alias name: mozillacert15.pem
  SHA1: 74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
  SHA256:
 0F:99:3C:8A:EF:97:BA:AF:56:87:14:0E:D5:9A:D1:82:1B:B4:AF:AC:F0:AA:9A:58:B5:D5:7A:33:8A:3A:FB:C
Alias name: mozillacert150.pem
  SHA1: 33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
  SHA256:
 EF:3C:B4:17:FC:8E:BF:6F:97:87:6C:9E:4E:CE:39:DE:1E:A5:FE:64:91:41:D1:02:8B:7D:11:C0:B2:29:8C:E
Alias name: mozillacert151.pem
  SHA1: AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
```

```
    SHA256:
 7F:12:CD:5F:7E:5E:29:0E:C7:D8:51:79:D5:B7:2C:20:A5:BE:75:08:FF:DB:5B:F8:1A:B9:68:4A:7F:C9:F6:6
Alias name: mozillacert16.pem
    SHA1: DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
    SHA256:
 06:87:26:03:31:A7:24:03:D9:09:F1:05:E6:9B:CF:0D:32:E1:BD:24:93:FF:C6:D9:20:6D:11:BC:D6:77:07:3
Alias name: mozillacert17.pem
    SHA1: 40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
    SHA256:
 76:7C:95:5A:76:41:2C:89:AF:68:8E:90:A1:C7:0F:55:6C:FD:6B:60:25:DB:EA:10:41:6D:7E:B6:83:1F:8C:4
Alias name: mozillacert18.pem
    SHA1: 79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
    SHA256:
 44:04:E3:3B:5E:14:0D:CF:99:80:51:FD:FC:80:28:C7:C8:16:15:C5:EE:73:7B:11:1B:58:82:33:A9:B5:35:A
Alias name: mozillacert19.pem
    SHA1: B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
    SHA256:
 C4:70:CF:54:7E:23:02:B9:77:FB:29:DD:71:A8:9A:7B:6C:1F:60:77:7B:03:29:F5:60:17:F3:28:BF:4F:6B:E
Alias name: mozillacert2.pem
    SHA1: 22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
    SHA256:
 69:DD:D7:EA:90:BB:57:C9:3E:13:5D:C8:5E:A6:FC:D5:48:0B:60:32:39:BD:C4:54:FC:75:8B:2A:26:CF:7F:7
Alias name: mozillacert20.pem
    SHA1: D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
    SHA256:
 62:DD:0B:E9:B9:F5:0A:16:3E:A0:F8:E7:5C:05:3B:1E:CA:57:EA:55:C8:68:8F:64:7C:68:81:F2:C8:35:7B:9
Alias name: mozillacert21.pem
    SHA1: 9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
    SHA256:
 BE:6C:4D:A2:BB:B9:BA:59:B6:F3:93:97:68:37:42:46:C3:C0:05:99:3F:A9:8F:02:0D:1D:ED:BE:D4:8A:81:D
Alias name: mozillacert22.pem
    SHA1: 32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
    SHA256:
 37:D5:10:06:C5:12:EA:AB:62:64:21:F1:EC:8C:92:01:3F:C5:F8:2A:E9:8E:E5:33:EB:46:19:B8:DE:B4:D0:6
Alias name: mozillacert23.pem
    SHA1: 91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
    SHA256:
 8D:72:2F:81:A9:C1:13:C0:79:1D:F1:36:A2:96:6D:B2:6C:95:0A:97:1D:B4:6B:41:99:F4:EA:54:B7:8B:FB:9
Alias name: mozillacert24.pem
    SHA1: 59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
    SHA256:
 66:8C:83:94:7D:A6:3B:72:4B:EC:E1:74:3C:31:A0:E6:AE:D0:DB:8E:C5:B3:1B:E3:77:BB:78:4F:91:B6:71:6
Alias name: mozillacert25.pem
    SHA1: 4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
```

```
  SHA256:
 9A:CF:AB:7E:43:C8:D8:80:D0:6B:26:2A:94:DE:EE:E4:B4:65:99:89:C3:D0:CA:F1:9B:AF:64:05:E4:1A:B7:D
Alias name: mozillacert26.pem
  SHA1: 87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
  SHA256:
 F1:C1:B5:0A:E5:A2:0D:D8:03:0E:C9:F6:BC:24:82:3D:D3:67:B5:25:57:59:B4:E7:1B:61:FC:E9:F7:37:5D:7
Alias name: mozillacert27.pem
  SHA1: 3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
  SHA256:
 42:00:F5:04:3A:C8:59:0E:BB:52:7D:20:9E:D1:50:30:29:FB:CB:D4:1C:A1:B5:06:EC:27:F1:5A:DE:7D:AC:6
Alias name: mozillacert28.pem
  SHA1: 66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
  SHA256:
 0C:2C:D6:3D:F7:80:6F:A3:99:ED:E8:09:11:6B:57:5B:F8:79:89:F0:65:18:F9:80:8C:86:05:03:17:8B:AF:6
Alias name: mozillacert29.pem
  SHA1: 74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
  SHA256:
 15:F0:BA:00:A3:AC:7A:F3:AC:88:4C:07:2B:10:11:A0:77:BD:77:C0:97:F4:01:64:B2:F8:59:8A:BD:83:86:0
Alias name: mozillacert3.pem
  SHA1: 87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
  SHA256:
 39:DF:7B:68:2B:7B:93:8F:84:71:54:81:CC:DE:8D:60:D8:F2:2E:C5:98:87:7D:0A:AA:C1:2B:59:18:2B:03:1
Alias name: mozillacert30.pem
  SHA1: E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
  SHA256:
 A7:12:72:AE:AA:A3:CF:E8:72:7F:7F:B3:9F:0F:B3:D1:E5:42:6E:90:60:B0:6E:E6:F1:3E:9A:3C:58:33:CD:4
Alias name: mozillacert31.pem
  SHA1: 9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
  SHA256:
 17:93:92:7A:06:14:54:97:89:AD:CE:2F:8F:34:F7:F0:B6:6D:0F:3A:E3:A3:B8:4D:21:EC:15:DB:BA:4F:AD:C
Alias name: mozillacert32.pem
  SHA1: 60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C
  SHA256:
 B9:BE:A7:86:0A:96:2E:A3:61:1D:AB:97:AB:6D:A3:E2:1C:10:68:B9:7D:55:57:5E:D0:E1:12:79:C1:1C:89:3
Alias name: mozillacert33.pem
  SHA1: FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
  SHA256:
 A2:2D:BA:68:1E:97:37:6E:2D:39:7D:72:8A:AE:3A:9B:62:96:B9:FD:BA:60:BC:2E:11:F6:47:F2:C6:75:FB:3
Alias name: mozillacert34.pem
  SHA1: 59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
  SHA256:
 41:C9:23:86:6A:B4:CA:D6:B7:AD:57:80:81:58:2E:02:07:97:A6:CB:DF:4F:FF:78:CE:83:96:B3:89:37:D7:F
Alias name: mozillacert35.pem
  SHA1: 2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
```

```
  SHA256:
 92:BF:51:19:AB:EC:CA:D0:B1:33:2D:C4:E1:D0:5F:BA:75:B5:67:90:44:EE:0C:A2:6E:93:1F:74:4F:2F:33:C
Alias name: mozillacert36.pem
  SHA1: 23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D
  SHA256:
 32:7A:3D:76:1A:BA:DE:A0:34:EB:99:84:06:27:5C:B1:A4:77:6E:FD:AE:2F:DF:6D:01:68:EA:1C:4F:55:67:D
Alias name: mozillacert37.pem
  SHA1: B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
  SHA256:
 E3:B6:A2:DB:2E:D7:CE:48:84:2F:7A:C5:32:41:C7:B7:1D:54:14:4B:FB:40:C1:1F:3F:1D:0B:42:F5:EE:A1:2
Alias name: mozillacert38.pem
  SHA1: CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36
  SHA256:
 A6:C5:1E:0D:A5:CA:0A:93:09:D2:E4:C0:E4:0C:2A:F9:10:7A:AE:82:03:85:7F:E1:98:E3:E7:69:E3:43:08:5
Alias name: mozillacert39.pem
  SHA1: AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
  SHA256:
 E6:B8:F8:76:64:85:F8:07:AE:7F:8D:AC:16:70:46:1F:07:C0:A1:3E:EF:3A:1F:F7:17:53:8D:7A:BA:D3:91:B
Alias name: mozillacert4.pem
  SHA1: E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B
  SHA256:
 0B:5E:ED:4E:84:64:03:CF:55:E0:65:84:84:40:ED:2A:82:75:8B:F5:B9:AA:1F:25:3D:46:13:CF:A0:80:FF:3
Alias name: mozillacert40.pem
  SHA1: 80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
  SHA256:
 8D:A0:84:FC:F9:9C:E0:77:22:F8:9B:32:05:93:98:06:FA:5C:B8:11:E1:C8:13:F6:A1:08:C7:D3:36:B3:40:8
Alias name: mozillacert41.pem
  SHA1: 6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
  SHA256:
 EB:F3:C0:2A:87:89:B1:FB:7D:51:19:95:D6:63:B7:29:06:D9:13:CE:0D:5E:10:56:8A:8A:77:E2:58:61:67:E
Alias name: mozillacert42.pem
  SHA1: 85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
  SHA256:
 B6:19:1A:50:D0:C3:97:7F:7D:A9:9B:CD:AA:C8:6A:22:7D:AE:B9:67:9E:C7:0B:A3:B0:C9:D9:22:71:C1:70:D
Alias name: mozillacert43.pem
  SHA1: F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
  SHA256:
 50:79:41:C7:44:60:A0:B4:70:86:22:0D:4E:99:32:57:2A:B5:D1:B5:BB:CB:89:80:AB:1C:B1:76:51:A8:44:D
Alias name: mozillacert44.pem
  SHA1: 5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
  SHA256:
 96:0A:DF:00:63:E9:63:56:75:0C:29:65:DD:0A:08:67:DA:0B:9C:BD:6E:77:71:4A:EA:FB:23:49:AB:39:3D:A
Alias name: mozillacert45.pem
  SHA1: 67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
```

```
   SHA256: 
  C0:A6:F4:DC:63:A2:4B:FD:CF:54:EF:2A:6A:08:2A:0A:72:DE:35:80:3E:2F:F5:FF:52:7A:E5:D8:72:06:DF:D
 Alias name: mozillacert46.pem
   SHA1: 40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
   SHA256: 
  EC:C3:E9:C3:40:75:03:BE:E0:91:AA:95:2F:41:34:8F:F8:8B:AA:86:3B:22:64:BE:FA:C8:07:90:15:74:E9:3
 Alias name: mozillacert47.pem
   SHA1: 1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
   SHA256: 
  E4:C7:34:30:D7:A5:B5:09:25:DF:43:37:0A:0D:21:6E:9A:79:B9:D6:DB:83:73:A0:C6:9E:B1:CC:31:C7:C5:2
 Alias name: mozillacert48.pem
   SHA1: A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC
   SHA256: 
  0F:4E:9C:DD:26:4B:02:55:50:D1:70:80:63:40:21:4F:E9:44:34:C9:B0:2F:69:7E:C7:10:FC:5F:EA:FB:5E:3
 Alias name: mozillacert49.pem
   SHA1: 61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71
   SHA256: 
  B7:B1:2B:17:1F:82:1D:AA:99:0C:D0:FE:50:87:B1:28:44:8B:A8:E5:18:4F:84:C5:1E:02:B5:C8:FB:96:2B:2
 Alias name: mozillacert5.pem
   SHA1: B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
   SHA256: 
  CE:CD:DC:90:50:99:D8:DA:DF:C5:B1:D2:09:B7:37:CB:E2:C1:8C:FB:2C:10:C0:FF:0B:CF:0D:32:86:FC:1A:A
 Alias name: mozillacert50.pem
   SHA1: 8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
   SHA256: 
  35:AE:5B:DD:D8:F7:AE:63:5C:FF:BA:56:82:A8:F0:0B:95:F4:84:62:C7:10:8E:E9:A0:E5:29:2B:07:4A:AF:B
 Alias name: mozillacert51.pem
   SHA1: FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
   SHA256: 
  EA:A9:62:C4:FA:4A:6B:AF:EB:E4:15:19:6D:35:1C:CD:88:8D:4F:53:F3:FA:8A:E6:D7:C4:66:A9:4E:60:42:B
 Alias name: mozillacert52.pem
   SHA1: 8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F
   SHA256: 
  E2:83:93:77:3D:A8:45:A6:79:F2:08:0C:C7:FB:44:A3:B7:A1:C3:79:2C:B7:EB:77:29:FD:CB:6A:8D:99:AE:A
 Alias name: mozillacert53.pem
   SHA1: 7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
   SHA256: 
  2D:47:43:7D:E1:79:51:21:5A:12:F3:C5:8E:51:C7:29:A5:80:26:EF:1F:CC:0A:5F:B3:D9:DC:01:2F:60:0D:1
 Alias name: mozillacert54.pem
   SHA1: 03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
   SHA256: 
  B4:78:B8:12:25:0D:F8:78:63:5C:2A:A7:EC:7D:15:5E:AA:62:5E:E8:29:16:E2:CD:29:43:61:88:6C:D1:FB:D
 Alias name: mozillacert55.pem
   SHA1: AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
```

```
   SHA256:
  A4:31:0D:50:AF:18:A6:44:71:90:37:2A:86:AF:AF:8B:95:1F:FB:43:1D:83:7F:1E:56:88:B4:59:71:ED:15:5
 Alias name: mozillacert56.pem
   SHA1: F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
   SHA256:
  4B:03:F4:58:07:AD:70:F2:1B:FC:2C:AE:71:C9:FD:E4:60:4C:06:4C:F5:FF:B6:86:BA:E5:DB:AA:D7:FD:D3:4
 Alias name: mozillacert57.pem
   SHA1: D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
   SHA256:
  F9:E6:7D:33:6C:51:00:2A:C0:54:C6:32:02:2D:66:DD:A2:E7:E3:FF:F1:0A:D0:61:ED:31:D8:BB:B4:10:CF:B
 Alias name: mozillacert58.pem
   SHA1: 8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
   SHA256:
  5E:DB:7A:C4:3B:82:A0:6A:87:61:E8:D7:BE:49:79:EB:F2:61:1F:7D:D7:9B:F9:1C:1C:6B:56:6A:21:9E:D7:6
 Alias name: mozillacert59.pem
   SHA1: 36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
   SHA256:
  23:99:56:11:27:A5:71:25:DE:8C:EF:EA:61:0D:DF:2F:A0:78:B5:C8:06:7F:4E:82:82:90:BF:B8:60:E8:4B:3
 Alias name: mozillacert6.pem
   SHA1: 27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
   SHA256:
  C3:84:6B:F2:4B:9E:93:CA:64:27:4C:0E:C6:7C:1E:CC:5E:02:4F:FC:AC:D2:D7:40:19:35:0E:81:FE:54:6A:E
 Alias name: mozillacert60.pem
   SHA1: 3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
   SHA256:
  BF:0F:EE:FB:9E:3A:58:1A:D5:F9:E9:DB:75:89:98:57:43:D2:61:08:5C:4D:31:4F:6F:5D:72:59:AA:42:16:1
 Alias name: mozillacert61.pem
   SHA1: E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
   SHA256:
  03:95:0F:B4:9A:53:1F:3E:19:91:94:23:98:DF:A9:E0:EA:32:D7:BA:1C:DD:9B:C8:5D:B5:7E:D9:40:0B:43:4
 Alias name: mozillacert62.pem
   SHA1: A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
   SHA256:
  A4:B6:B3:99:6F:C2:F3:06:B3:FD:86:81:BD:63:41:3D:8C:50:09:CC:4F:A3:29:C2:CC:F0:E2:FA:1B:14:03:0
 Alias name: mozillacert63.pem
   SHA1: 89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
   SHA256:
  3C:5F:81:FE:A5:FA:B8:2C:64:BF:A2:EA:EC:AF:CD:E8:E0:77:FC:86:20:A7:CA:E5:37:16:3D:F3:6E:DB:F3:7
 Alias name: mozillacert64.pem
   SHA1: 62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
   SHA256:
  AB:70:36:36:5C:71:54:AA:29:C2:C2:9F:5D:41:91:16:3B:16:2A:22:25:01:13:57:D5:6D:07:FF:A7:BC:1F:7
 Alias name: mozillacert65.pem
   SHA1: 69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:35:FB
```

```
  SHA256:
 BC:23:F9:8A:31:3C:B9:2D:E3:BB:FC:3A:5A:9F:44:61:AC:39:49:4C:4A:E1:5A:9E:9D:F1:31:E9:9B:73:01:9
Alias name: mozillacert66.pem
  SHA1: DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34
  SHA256:
 E6:09:07:84:65:A4:19:78:0C:B6:AC:4C:1C:0B:FB:46:53:D9:D9:CC:6E:B3:94:6E:B7:F3:D6:99:97:BA:D5:9
Alias name: mozillacert67.pem
  SHA1: D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
  SHA256:
 CB:B5:22:D7:B7:F1:27:AD:6A:01:13:86:5B:DF:1C:D4:10:2E:7D:07:59:AF:63:5A:7C:F4:72:0D:C9:63:C5:3
Alias name: mozillacert68.pem
  SHA1: AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
  SHA256:
 04:04:80:28:BF:1F:28:64:D4:8F:9A:D4:D8:32:94:36:6A:82:88:56:55:3F:3B:14:30:3F:90:14:7F:5D:40:E
Alias name: mozillacert69.pem
  SHA1: 2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
  SHA256:
 25:30:CC:8E:98:32:15:02:BA:D9:6F:9B:1F:BA:1B:09:9E:2D:29:9E:0F:45:48:BB:91:4F:36:3B:C0:D4:53:1
Alias name: mozillacert7.pem
  SHA1: AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
  SHA256:
 14:65:FA:20:53:97:B8:76:FA:A6:F0:A9:95:8E:55:90:E4:0F:CC:7F:AA:4F:B7:C2:C8:67:75:21:FB:5F:B6:5
Alias name: mozillacert70.pem
  SHA1: 78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
  SHA256:
 06:3E:4A:FA:C4:91:DF:D3:32:F3:08:9B:85:42:E9:46:17:D8:93:D7:FE:94:4E:10:A7:93:7E:E2:9D:96:93:C
Alias name: mozillacert71.pem
  SHA1: 4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
  SHA256:
 13:63:35:43:93:34:A7:69:80:16:A0:D3:24:DE:72:28:4E:07:9D:7B:52:20:BB:8F:BD:74:78:16:EE:BE:BA:C
Alias name: mozillacert72.pem
  SHA1: 47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
  SHA256:
 45:14:0B:32:47:EB:9C:C8:C5:B4:F0:D7:B5:30:91:F7:32:92:08:9E:6E:5A:63:E2:74:9D:D3:AC:A9:19:8E:D
Alias name: mozillacert73.pem
  SHA1: B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
  SHA256:
 2C:E1:CB:0B:F9:D2:F9:E1:02:99:3F:BE:21:51:52:C3:B2:DD:0C:AB:DE:1C:68:E5:31:9B:83:91:54:DB:B7:F
Alias name: mozillacert74.pem
  SHA1: 92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
  SHA256:
 56:8D:69:05:A2:C8:87:08:A4:B3:02:51:90:ED:CF:ED:B1:97:4A:60:6A:13:C6:E5:29:0F:CB:2A:E6:3E:DA:B
Alias name: mozillacert75.pem
  SHA1: D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
```

```
   SHA256:
 08:29:7A:40:47:DB:A2:36:80:C7:31:DB:6E:31:76:53:CA:78:48:E1:BE:BD:3A:0B:01:79:A7:07:F9:2C:F1:7
Alias name: mozillacert76.pem
   SHA1: F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
   SHA256:
 03:76:AB:1D:54:C5:F9:80:3C:E4:B2:E2:01:A0:EE:7E:EF:7B:57:B6:36:E8:A9:3C:9B:8D:48:60:C9:6F:5F:A
Alias name: mozillacert77.pem
   SHA1: 13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
   SHA256:
 EB:04:CF:5E:B1:F3:9A:FA:76:2F:2B:B1:20:F2:96:CB:A5:20:C1:B9:7D:B1:58:95:65:B8:1C:B9:A1:7B:72:4
Alias name: mozillacert78.pem
   SHA1: 29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
   SHA256:
 0A:81:EC:5A:92:97:77:F1:45:90:4A:F3:8D:5D:50:9F:66:B5:E2:C5:8F:CD:B5:31:05:8B:0E:17:F3:F0:B4:1
Alias name: mozillacert79.pem
   SHA1: D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
   SHA256:
 70:A7:3F:7F:37:6B:60:07:42:48:90:45:34:B1:14:82:D5:BF:0E:69:8E:CC:49:8D:F5:25:77:EB:F2:E9:3B:9
Alias name: mozillacert8.pem
   SHA1: 3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:6A:0F
   SHA256:
 C7:66:A9:BE:F2:D4:07:1C:86:3A:31:AA:49:20:E8:13:B2:D1:98:60:8C:B7:B7:CF:E2:11:43:B8:36:DF:09:E
Alias name: mozillacert80.pem
   SHA1: B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
   SHA256:
 BD:71:FD:F6:DA:97:E4:CF:62:D1:64:7A:DD:25:81:B0:7D:79:AD:F8:39:7E:B4:EC:BA:9C:5E:84:88:82:14:2
Alias name: mozillacert81.pem
   SHA1: 07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
   SHA256:
 5C:58:46:8D:55:F5:8E:49:7E:74:39:82:D2:B5:00:10:B6:D1:65:37:4A:CF:83:A7:D4:A3:2D:B7:68:C4:40:8
Alias name: mozillacert82.pem
   SHA1: 2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
   SHA256:
 FC:BF:E2:88:62:06:F7:2B:27:59:3C:8B:07:02:97:E1:2D:76:9E:D1:0E:D7:93:07:05:A8:09:8E:FF:C1:4D:1
Alias name: mozillacert83.pem
   SHA1: A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:EA:46
   SHA256:
 8C:4E:DF:D0:43:48:F3:22:96:9E:7E:29:A4:CD:4D:CA:00:46:55:06:1C:16:E1:B0:76:42:2E:F3:42:AD:63:0
Alias name: mozillacert84.pem
   SHA1: D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2
   SHA256:
 79:3C:BF:45:59:B9:FD:E3:8A:B2:2D:F1:68:69:F6:98:81:AE:14:C4:B0:13:9A:C7:88:A7:8A:1A:FC:CA:02:F
Alias name: mozillacert85.pem
   SHA1: CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
```

```
  SHA256:
 BF:D8:8F:E1:10:1C:41:AE:3E:80:1B:F8:BE:56:35:0E:E9:BA:D1:A6:B9:BD:51:5E:DC:5C:6D:5B:87:11:AC:4
Alias name: mozillacert86.pem
  SHA1: 74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
  SHA256:
 E7:68:56:34:EF:AC:F6:9A:CE:93:9A:6B:25:5B:7B:4F:AB:EF:42:93:5B:50:A2:65:AC:B5:CB:60:27:E4:4E:7
Alias name: mozillacert87.pem
  SHA1: 5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
  SHA256:
 51:3B:2C:EC:B8:10:D4:CD:E5:DD:85:39:1A:DF:C6:C2:DD:60:D8:7B:B7:36:D2:B5:21:48:4A:A4:7A:0E:BE:F
Alias name: mozillacert88.pem
  SHA1: FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
  SHA256:
 BC:10:4F:15:A4:8B:E7:09:DC:A5:42:A7:E1:D4:B9:DF:6F:05:45:27:E8:02:EA:A9:2D:59:54:44:25:8A:FE:7
Alias name: mozillacert89.pem
  SHA1: C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
  SHA256:
 E3:89:36:0D:0F:DB:AE:B3:D2:50:58:4B:47:30:31:4E:22:2F:39:C1:56:A0:20:14:4E:8D:96:05:61:79:15:0
Alias name: mozillacert9.pem
  SHA1: F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
  SHA256:
 76:00:29:5E:EF:E8:5B:9E:1F:D6:24:DB:76:06:2A:AA:AE:59:81:8A:54:D2:77:4C:D4:C0:B2:C0:11:31:E1:B
Alias name: mozillacert90.pem
  SHA1: F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
  SHA256:
 55:92:60:84:EC:96:3A:64:B9:6E:2A:BE:01:CE:0B:A8:6A:64:FB:FE:BC:C7:AA:B5:AF:C1:55:B3:7F:D7:60:6
Alias name: mozillacert91.pem
  SHA1: 3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
  SHA256:
 C1:B4:82:99:AB:A5:20:8F:E9:63:0A:CE:55:CA:68:A0:3E:DA:5A:51:9C:88:02:A0:D3:A6:73:BE:8F:8E:55:7
Alias name: mozillacert92.pem
  SHA1: A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0
  SHA256:
 E1:78:90:EE:09:A3:FB:F4:F4:8B:9C:41:4A:17:D6:37:B7:A5:06:47:E9:BC:75:23:22:72:7F:CC:17:42:A9:1
Alias name: mozillacert93.pem
  SHA1: 31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17
  SHA256:
 C7:BA:65:67:DE:93:A7:98:AE:1F:AA:79:1E:71:2D:37:8F:AE:1F:93:C4:39:7F:EA:44:1B:B7:CB:E6:FD:59:9
Alias name: mozillacert94.pem
  SHA1: 49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
  SHA256:
 9A:11:40:25:19:7C:5B:B9:5D:94:E6:3D:55:CD:43:79:08:47:B6:46:B2:3C:DF:11:AD:A4:A0:0E:FF:15:FB:4
Alias name: mozillacert95.pem
  SHA1: DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
```

```
    SHA256:
  ED:F7:EB:BC:A2:7A:2A:38:4D:38:7B:7D:40:10:C6:66:E2:ED:B4:84:3E:4C:29:B4:AE:1D:5B:93:32:E6:B2:4
Alias name: mozillacert96.pem
    SHA1: 55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
    SHA256:
  FD:73:DA:D3:1C:64:4F:F1:B4:3B:EF:0C:CD:DA:96:71:0B:9C:D9:87:5E:CA:7E:31:70:7A:F3:E9:6D:52:2B:B
Alias name: mozillacert97.pem
    SHA1: 85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
    SHA256:
  83:CE:3C:12:29:68:8A:59:3D:48:5F:81:97:3C:0F:91:95:43:1E:DA:37:CC:5E:36:43:0E:79:C7:A8:88:63:8
Alias name: mozillacert98.pem
    SHA1: C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
    SHA256:
  3E:84:BA:43:42:90:85:16:E7:75:73:C0:99:2F:09:79:CA:08:4E:46:85:68:1F:F1:95:CC:BA:8A:22:9B:8A:7
Alias name: mozillacert99.pem
    SHA1: F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
    SHA256:
  97:8C:D9:66:F2:FA:A0:7B:A7:AA:95:00:D9:C0:2E:9D:77:F2:CD:AD:A6:AD:6B:A7:4A:F4:B9:1C:66:59:3C:5
Alias name: netlockaranyclassgoldfotanusitvany
    SHA1: 06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
    SHA256:
  6C:61:DA:C3:A2:DE:F0:31:50:6B:E0:36:D2:A6:FE:40:19:94:FB:D1:3D:F9:C8:D4:66:59:92:74:C4:46:EC:9
Alias name: networksolutionscertificateauthority
    SHA1: 74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
    SHA256:
  15:F0:BA:00:A3:AC:7A:F3:AC:88:4C:07:2B:10:11:A0:77:BD:77:C0:97:F4:01:64:B2:F8:59:8A:BD:83:86:0
Alias name: oistewisekeyglobalrootgaca
    SHA1: 59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
    SHA256:
  41:C9:23:86:6A:B4:CA:D6:B7:AD:57:80:81:58:2E:02:07:97:A6:CB:DF:4F:FF:78:CE:83:96:B3:89:37:D7:F
Alias name: oistewisekeyglobalrootgbca
    SHA1: 0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
    SHA256:
  6B:9C:08:E8:6E:B0:F7:67:CF:AD:65:CD:98:B6:21:49:E5:49:4A:67:F5:84:5E:7B:D1:ED:01:9F:27:B8:6B:D
Alias name: oistewisekeyglobalrootgcca
    SHA1: E0:11:84:5E:34:DE:BE:88:81:B9:9C:F6:16:26:D1:96:1F:C3:B9:31
    SHA256:
  85:60:F9:1C:36:24:DA:BA:95:70:B5:FE:A0:DB:E3:6F:F1:1A:83:23:BE:94:86:85:4F:B3:F3:4A:55:71:19:8
Alias name: quovadisrootca
    SHA1: DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
    SHA256:
  A4:5E:DE:3B:BB:F0:9C:8A:E1:5C:72:EF:C0:72:68:D6:93:A2:1C:99:6F:D5:1E:67:CA:07:94:60:FD:6D:88:7
Alias name: quovadisrootca1g3
    SHA1: 1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67
```

```
  SHA256:
 8A:86:6F:D1:B2:76:B5:7E:57:8E:92:1C:65:82:8A:2B:ED:58:E9:F2:F2:88:05:41:34:B7:F1:F4:BF:C9:CC:7
Alias name: quovadisrootca2
  SHA1: CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
  SHA256:
 85:A0:DD:7D:D7:20:AD:B7:FF:05:F8:3D:54:2B:20:9D:C7:FF:45:28:F7:D6:77:B1:83:89:FE:A5:E5:C4:9E:8
Alias name: quovadisrootca2g3
  SHA1: 09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
  SHA256:
 8F:E4:FB:0A:F9:3A:4D:0D:67:DB:0B:EB:B2:3E:37:C7:1B:F3:25:DC:BC:DD:24:0E:A0:4D:AF:58:B4:7E:18:4
Alias name: quovadisrootca3
  SHA1: 1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
  SHA256:
 18:F1:FC:7F:20:5D:F8:AD:DD:EB:7F:E0:07:DD:57:E3:AF:37:5A:9C:4D:8D:73:54:6B:F4:F1:FE:D1:E1:8D:3
Alias name: quovadisrootca3g3
  SHA1: 48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D
  SHA256:
 88:EF:81:DE:20:2E:B0:18:45:2E:43:F8:64:72:5C:EA:5F:BD:1F:C2:D9:D2:05:73:07:09:C5:D8:B8:69:0F:4
Alias name: secomevrootca1
  SHA1: FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
  SHA256:
 A2:2D:BA:68:1E:97:37:6E:2D:39:7D:72:8A:AE:3A:9B:62:96:B9:FD:BA:60:BC:2E:11:F6:47:F2:C6:75:FB:3
Alias name: secomscrootca1
  SHA1: 36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
  SHA256:
 E7:5E:72:ED:9F:56:0E:EC:6E:B4:80:00:73:A4:3F:C3:AD:19:19:5A:39:22:82:01:78:95:97:4A:99:02:6B:6
Alias name: secomscrootca2
  SHA1: 5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
  SHA256:
 51:3B:2C:EC:B8:10:D4:CD:E5:DD:85:39:1A:DF:C6:C2:DD:60:D8:7B:B7:36:D2:B5:21:48:4A:A4:7A:0E:BE:F
Alias name: secomvalicertclass1ca
  SHA1: E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
  SHA256:
 F4:C1:49:55:1A:30:13:A3:5B:C7:BF:FE:17:A7:F3:44:9B:C1:AB:5B:5A:0A:E7:4B:06:C2:3B:90:00:4C:01:0
Alias name: secureglobalca
  SHA1: 3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
  SHA256:
 42:00:F5:04:3A:C8:59:0E:BB:52:7D:20:9E:D1:50:30:29:FB:CB:D4:1C:A1:B5:06:EC:27:F1:5A:DE:7D:AC:6
Alias name: securesignrootca11
  SHA1: 3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
  SHA256:
 BF:0F:EE:FB:9E:3A:58:1A:D5:F9:E9:DB:75:89:98:57:43:D2:61:08:5C:4D:31:4F:6F:5D:72:59:AA:42:16:1
Alias name: securetrustca
  SHA1: 87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
```

```
  SHA256:
 F1:C1:B5:0A:E5:A2:0D:D8:03:0E:C9:F6:BC:24:82:3D:D3:67:B5:25:57:59:B4:E7:1B:61:FC:E9:F7:37:5D:7
Alias name: securitycommunicationrootca
  SHA1: 36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
  SHA256:
 E7:5E:72:ED:9F:56:0E:EC:6E:B4:80:00:73:A4:3F:C3:AD:19:19:5A:39:22:82:01:78:95:97:4A:99:02:6B:6
Alias name: securitycommunicationrootca2
  SHA1: 5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
  SHA256:
 51:3B:2C:EC:B8:10:D4:CD:E5:DD:85:39:1A:DF:C6:C2:DD:60:D8:7B:B7:36:D2:B5:21:48:4A:A4:7A:0E:BE:F
Alias name: soneraclass1ca
  SHA1: 07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
  SHA256:
 CD:80:82:84:CF:74:6F:F2:FD:6E:B5:8A:A1:D5:9C:4A:D4:B3:CA:56:FD:C6:27:4A:89:26:A7:83:5F:32:31:3
Alias name: soneraclass2ca
  SHA1: 37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
  SHA256:
 79:08:B4:03:14:C1:38:10:0B:51:8D:07:35:80:7F:FB:FC:F8:51:8A:00:95:33:71:05:BA:38:6B:15:3D:D9:2
Alias name: soneraclass2rootca
  SHA1: 37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
  SHA256:
 79:08:B4:03:14:C1:38:10:0B:51:8D:07:35:80:7F:FB:FC:F8:51:8A:00:95:33:71:05:BA:38:6B:15:3D:D9:2
Alias name: sslcomevrootcertificationauthorityecc
  SHA1: 4C:DD:51:A3:D1:F5:20:32:14:B0:C6:C5:32:23:03:91:C7:46:42:6D
  SHA256:
 22:A2:C1:F7:BD:ED:70:4C:C1:E7:01:B5:F4:08:C3:10:88:0F:E9:56:B5:DE:2A:4A:44:F9:9C:87:3A:25:A7:C
Alias name: sslcomevrootcertificationauthorityrsar2
  SHA1: 74:3A:F0:52:9B:D0:32:A0:F4:4A:83:CD:D4:BA:A9:7B:7C:2E:C4:9A
  SHA256:
 2E:7B:F1:6C:C2:24:85:A7:BB:E2:AA:86:96:75:07:61:B0:AE:39:BE:3B:2F:E9:D0:CC:6D:4E:F7:34:91:42:5
Alias name: sslcomrootcertificationauthorityecc
  SHA1: C3:19:7C:39:24:E6:54:AF:1B:C4:AB:20:95:7A:E2:C3:0E:13:02:6A
  SHA256:
 34:17:BB:06:CC:60:07:DA:1B:96:1C:92:0B:8A:B4:CE:3F:AD:82:0E:4A:A3:0B:9A:CB:C4:A7:4E:BD:CE:BC:6
Alias name: sslcomrootcertificationauthorityrsa
  SHA1: B7:AB:33:08:D1:EA:44:77:BA:14:80:12:5A:6F:BD:A9:36:49:0C:BB
  SHA256:
 85:66:6A:56:2E:E0:BE:5C:E9:25:C1:D8:89:0A:6F:76:A8:7E:C1:6D:4D:7D:5F:29:EA:74:19:CF:20:12:3B:6
Alias name: staatdernederlandenevrootca
  SHA1: 76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
  SHA256:
 4D:24:91:41:4C:FE:95:67:46:EC:4C:EF:A6:CF:6F:72:E2:8A:13:29:43:2F:9D:8A:90:7A:C4:CB:5D:AD:C1:5
Alias name: staatdernederlandenrootcag3
  SHA1: D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
```

```
  SHA256:
 3C:4F:B0:B9:5A:B8:B3:00:32:F4:32:B8:6F:53:5F:E1:72:C1:85:D0:FD:39:86:58:37:CF:36:18:7F:A6:F4:2
Alias name: starfieldclass2ca
  SHA1: AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
  SHA256:
 14:65:FA:20:53:97:B8:76:FA:A6:F0:A9:95:8E:55:90:E4:0F:CC:7F:AA:4F:B7:C2:C8:67:75:21:FB:5F:B6:5
Alias name: starfieldrootcertificateauthorityg2
  SHA1: B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
  SHA256:
 2C:E1:CB:0B:F9:D2:F9:E1:02:99:3F:BE:21:51:52:C3:B2:DD:0C:AB:DE:1C:68:E5:31:9B:83:91:54:DB:B7:F
Alias name: starfieldrootg2ca
  SHA1: B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
  SHA256:
 2C:E1:CB:0B:F9:D2:F9:E1:02:99:3F:BE:21:51:52:C3:B2:DD:0C:AB:DE:1C:68:E5:31:9B:83:91:54:DB:B7:F
Alias name: starfieldservicesrootcertificateauthorityg2
  SHA1: 92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
  SHA256:
 56:8D:69:05:A2:C8:87:08:A4:B3:02:51:90:ED:CF:ED:B1:97:4A:60:6A:13:C6:E5:29:0F:CB:2A:E6:3E:DA:B
Alias name: starfieldservicesrootg2ca
  SHA1: 92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
  SHA256:
 56:8D:69:05:A2:C8:87:08:A4:B3:02:51:90:ED:CF:ED:B1:97:4A:60:6A:13:C6:E5:29:0F:CB:2A:E6:3E:DA:B
Alias name: swisssigngoldcag2
  SHA1: D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
  SHA256:
 62:DD:0B:E9:B9:F5:0A:16:3E:A0:F8:E7:5C:05:3B:1E:CA:57:EA:55:C8:68:8F:64:7C:68:81:F2:C8:35:7B:9
Alias name: swisssigngoldg2ca
  SHA1: D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
  SHA256:
 62:DD:0B:E9:B9:F5:0A:16:3E:A0:F8:E7:5C:05:3B:1E:CA:57:EA:55:C8:68:8F:64:7C:68:81:F2:C8:35:7B:9
Alias name: swisssignplatinumg2ca
  SHA1: 56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66
  SHA256:
 3B:22:2E:56:67:11:E9:92:30:0D:C0:B1:5A:B9:47:3D:AF:DE:F8:C8:4D:0C:EF:7D:33:17:B4:C1:82:1D:14:3
Alias name: swisssignsilvercag2
  SHA1: 9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
  SHA256:
 BE:6C:4D:A2:BB:B9:BA:59:B6:F3:93:97:68:37:42:46:C3:C0:05:99:3F:A9:8F:02:0D:1D:ED:BE:D4:8A:81:D
Alias name: swisssignsilverg2ca
  SHA1: 9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
  SHA256:
 BE:6C:4D:A2:BB:B9:BA:59:B6:F3:93:97:68:37:42:46:C3:C0:05:99:3F:A9:8F:02:0D:1D:ED:BE:D4:8A:81:D
Alias name: szafirrootca2
  SHA1: E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE
```

```
  SHA256:
 A1:33:9D:33:28:1A:0B:56:E5:57:D3:D3:2B:1C:E7:F9:36:7E:B0:94:BD:5F:A7:2A:7E:50:04:C8:DE:D7:CA:F
Alias name: teliasonerarootcav1
  SHA1: 43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
  SHA256:
 DD:69:36:FE:21:F8:F0:77:C1:23:A1:A5:21:C1:22:24:F7:22:55:B7:3E:03:A7:26:06:93:E8:A2:4B:0F:A3:8
Alias name: thawtepersonalfreemailca
  SHA1: E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
  SHA256:
 5B:38:BD:12:9E:83:D5:A0:CA:D2:39:21:08:94:90:D5:0D:4A:AE:37:04:28:F8:DD:FF:FF:FA:4C:15:64:E1:8
Alias name: thawtepremiumserverca
  SHA1: E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:66:66
  SHA256:
 3F:9F:27:D5:83:20:4B:9E:09:C8:A3:D2:06:6C:4B:57:D3:A2:47:9C:36:93:65:08:80:50:56:98:10:5D:BC:E
Alias name: thawteprimaryrootca
  SHA1: 91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
  SHA256:
 8D:72:2F:81:A9:C1:13:C0:79:1D:F1:36:A2:96:6D:B2:6C:95:0A:97:1D:B4:6B:41:99:F4:EA:54:B7:8B:FB:9
Alias name: thawteprimaryrootcag2
  SHA1: AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
  SHA256:
 A4:31:0D:50:AF:18:A6:44:71:90:37:2A:86:AF:AF:8B:95:1F:FB:43:1D:83:7F:1E:56:88:B4:59:71:ED:15:5
Alias name: thawteprimaryrootcag3
  SHA1: F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
  SHA256:
 4B:03:F4:58:07:AD:70:F2:1B:FC:2C:AE:71:C9:FD:E4:60:4C:06:4C:F5:FF:B6:86:BA:E5:DB:AA:D7:FD:D3:4
Alias name: thawteserverca
  SHA1: 9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
  SHA256:
 87:C6:78:BF:B8:B2:5F:38:F7:E9:7B:33:69:56:BB:CF:14:4B:BA:CA:A5:36:47:E6:1A:23:25:BC:10:55:31:6
Alias name: trustcenterclass2caii
  SHA1: AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
  SHA256:
 E6:B8:F8:76:64:85:F8:07:AE:7F:8D:AC:16:70:46:1F:07:C0:A1:3E:EF:3A:1F:F7:17:53:8D:7A:BA:D3:91:B
Alias name: trustcenterclass4caii
  SHA1: A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
  SHA256:
 32:66:96:7E:59:CD:68:00:8D:9D:D3:20:81:11:85:C7:04:20:5E:8D:95:FD:D8:4F:1C:7B:31:1E:67:04:FC:3
Alias name: trustcenteruniversalcai
  SHA1: 6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
  SHA256:
 EB:F3:C0:2A:87:89:B1:FB:7D:51:19:95:D6:63:B7:29:06:D9:13:CE:0D:5E:10:56:8A:8A:77:E2:58:61:67:E
Alias name: trustcoreca1
  SHA1: 58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
```

```
  SHA256:
 5A:88:5D:B1:9C:01:D9:12:C5:75:93:88:93:8C:AF:BB:DF:03:1A:B2:D4:8E:91:EE:15:58:9B:42:97:1D:03:9
Alias name: trustcorrootcertca1
  SHA1: FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
  SHA256:
 D4:0E:9C:86:CD:8F:E4:68:C1:77:69:59:F4:9E:A7:74:FA:54:86:84:B6:C4:06:F3:90:92:61:F4:DC:E2:57:5
Alias name: trustcorrootcertca2
  SHA1: B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
  SHA256:
 07:53:E9:40:37:8C:1B:D5:E3:83:6E:39:5D:AE:A5:CB:83:9E:50:46:F1:BD:0E:AE:19:51:CF:10:FE:C7:C9:6
Alias name: trustisfpsrootca
  SHA1: 3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
  SHA256:
 C1:B4:82:99:AB:A5:20:8F:E9:63:0A:CE:55:CA:68:A0:3E:DA:5A:51:9C:88:02:A0:D3:A6:73:BE:8F:8E:55:7
Alias name: ttelesecglobalrootclass2
  SHA1: 59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
  SHA256:
 91:E2:F5:78:8D:58:10:EB:A7:BA:58:73:7D:E1:54:8A:8E:CA:CD:01:45:98:BC:0B:14:3E:04:1B:17:05:25:5
Alias name: ttelesecglobalrootclass2ca
  SHA1: 59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
  SHA256:
 91:E2:F5:78:8D:58:10:EB:A7:BA:58:73:7D:E1:54:8A:8E:CA:CD:01:45:98:BC:0B:14:3E:04:1B:17:05:25:5
Alias name: ttelesecglobalrootclass3
  SHA1: 55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
  SHA256:
 FD:73:DA:D3:1C:64:4F:F1:B4:3B:EF:0C:CD:DA:96:71:0B:9C:D9:87:5E:CA:7E:31:70:7A:F3:E9:6D:52:2B:B
Alias name: ttelesecglobalrootclass3ca
  SHA1: 55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
  SHA256:
 FD:73:DA:D3:1C:64:4F:F1:B4:3B:EF:0C:CD:DA:96:71:0B:9C:D9:87:5E:CA:7E:31:70:7A:F3:E9:6D:52:2B:B
Alias name: tubitakkamusmsslkoksertifikasisurum1
  SHA1: 31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
  SHA256:
 46:ED:C3:68:90:46:D5:3A:45:3F:B3:10:4A:B8:0D:CA:EC:65:8B:26:60:EA:16:29:DD:7E:86:79:90:64:87:1
Alias name: twcaglobalrootca
  SHA1: 9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
  SHA256:
 59:76:90:07:F7:68:5D:0F:CD:50:87:2F:9F:95:D5:75:5A:5B:2B:45:7D:81:F3:69:2B:61:0A:98:67:2F:0E:1
Alias name: twcarootcertificationauthority
  SHA1: CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
  SHA256:
 BF:D8:8F:E1:10:1C:41:AE:3E:80:1B:F8:BE:56:35:0E:E9:BA:D1:A6:B9:BD:51:5E:DC:5C:6D:5B:87:11:AC:4
Alias name: ucaextendedvalidationroot
  SHA1: A3:A1:B0:6F:24:61:23:4A:E3:36:A5:C2:37:FC:A6:FF:DD:F0:D7:3A
```

```
  SHA256:
 D4:3A:F9:B3:54:73:75:5C:96:84:FC:06:D7:D8:CB:70:EE:5C:28:E7:73:FB:29:4E:B4:1E:E7:17:22:92:4D:2
Alias name: ucaglobalg2root
  SHA1: 28:F9:78:16:19:7A:FF:18:25:18:AA:44:FE:C1:A0:CE:5C:B6:4C:8A
  SHA256:
 9B:EA:11:C9:76:FE:01:47:64:C1:BE:56:A6:F9:14:B5:A5:60:31:7A:BD:99:88:39:33:82:E5:16:1A:A0:49:3
Alias name: usertrustecc
  SHA1: D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0
  SHA256:
 4F:F4:60:D5:4B:9C:86:DA:BF:BC:FC:57:12:E0:40:0D:2B:ED:3F:BC:4D:4F:BD:AA:86:E0:6A:DC:D2:A9:AD:7
Alias name: usertrustecccertificationauthority
  SHA1: D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0
  SHA256:
 4F:F4:60:D5:4B:9C:86:DA:BF:BC:FC:57:12:E0:40:0D:2B:ED:3F:BC:4D:4F:BD:AA:86:E0:6A:DC:D2:A9:AD:7
Alias name: usertrustrsa
  SHA1: 2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
  SHA256:
 E7:93:C9:B0:2F:D8:AA:13:E2:1C:31:22:8A:CC:B0:81:19:64:3B:74:9C:89:89:64:B1:74:6D:46:C3:D4:CB:D
Alias name: usertrustrsacertificationauthority
  SHA1: 2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
  SHA256:
 E7:93:C9:B0:2F:D8:AA:13:E2:1C:31:22:8A:CC:B0:81:19:64:3B:74:9C:89:89:64:B1:74:6D:46:C3:D4:CB:D
Alias name: utndatacorpsgcca
  SHA1: 58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
  SHA256:
 85:FB:2F:91:DD:12:27:5A:01:45:B6:36:53:4F:84:02:4A:D6:8B:69:B8:EE:88:68:4F:F7:11:37:58:05:B3:4
Alias name: utnuserfirstclientauthemailca
  SHA1: B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
  SHA256:
 43:F2:57:41:2D:44:0D:62:74:76:97:4F:87:7D:A8:F1:FC:24:44:56:5A:36:7A:E6:0E:DD:C2:7A:41:25:31:A
Alias name: utnuserfirsthardwareca
  SHA1: 04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
  SHA256:
 6E:A5:47:41:D0:04:66:7E:ED:1B:48:16:63:4A:A3:A7:9E:6E:4B:96:95:0F:82:79:DA:FC:8D:9B:D8:81:21:3
Alias name: utnuserfirstobjectca
  SHA1: E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B:AC:1D:81:D8:38:5E:2D:46
  SHA256:
 6F:FF:78:E4:00:A7:0C:11:01:1C:D8:59:77:C4:59:FB:5A:F9:6A:3D:F0:54:08:20:D0:F4:B8:60:78:75:E5:8
Alias name: valicertclass2ca
  SHA1: 31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
  SHA256:
 58:D0:17:27:9C:D4:DC:63:AB:DD:B1:96:A6:C9:90:6C:30:C4:E0:87:83:EA:E8:C1:60:99:54:D6:93:55:59:6
Alias name: verisignc1g1.pem
  SHA1: 90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:55:6F
```

```
  SHA256:
 D1:7C:D8:EC:D5:86:B7:12:23:8A:48:2C:E4:6F:A5:29:39:70:74:2F:27:6D:8A:B6:A9:E4:6E:E0:28:8F:33:5
Alias name: verisignc1g2.pem
  SHA1: 27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
  SHA256:
 34:1D:E9:8B:13:92:AB:F7:F4:AB:90:A9:60:CF:25:D4:BD:6E:C6:5B:9A:51:CE:6E:D0:67:D0:0E:C7:CE:9B:7
Alias name: verisignc1g3.pem
  SHA1: 20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
  SHA256:
 CB:B5:AF:18:5E:94:2A:24:02:F9:EA:CB:C0:ED:5B:B8:76:EE:A3:C1:22:36:23:D0:04:47:E4:F3:BA:55:4B:6
Alias name: verisignc1g6.pem
  SHA1: 51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
  SHA256:
 9D:19:0B:2E:31:45:66:68:5B:E8:A8:89:E2:7A:A8:C7:D7:AE:1D:8A:AD:DB:A3:C1:EC:F9:D2:48:63:CD:34:B
Alias name: verisignc2g1.pem
  SHA1: 67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:14:2A
  SHA256:
 BD:46:9F:F4:5F:AA:E7:C5:4C:CB:D6:9D:3F:3B:00:22:55:D9:B0:6B:10:B1:D0:FA:38:8B:F9:6B:91:8B:2C:E
Alias name: verisignc2g2.pem
  SHA1: B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
  SHA256:
 3A:43:E2:20:FE:7F:3E:A9:65:3D:1E:21:74:2E:AC:2B:75:C2:0F:D8:98:03:05:BC:50:2C:AF:8C:2D:9B:41:A
Alias name: verisignc2g3.pem
  SHA1: 61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
  SHA256:
 92:A9:D9:83:3F:E1:94:4D:B3:66:E8:BF:AE:7A:95:B6:48:0C:2D:6C:6C:2A:1B:E6:5D:42:36:B6:08:FC:A1:B
Alias name: verisignc2g6.pem
  SHA1: 40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F
  SHA256:
 CB:62:7D:18:B5:8A:D5:6D:DE:33:1A:30:45:6B:C6:5C:60:1A:4E:9B:18:DE:DC:EA:08:E7:DA:AA:07:81:5F:F
Alias name: verisignc3g1.pem
  SHA1: A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
  SHA256:
 A4:B6:B3:99:6F:C2:F3:06:B3:FD:86:81:BD:63:41:3D:8C:50:09:CC:4F:A3:29:C2:CC:F0:E2:FA:1B:14:03:0
Alias name: verisignc3g2.pem
  SHA1: 85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
  SHA256:
 83:CE:3C:12:29:68:8A:59:3D:48:5F:81:97:3C:0F:91:95:43:1E:DA:37:CC:5E:36:43:0E:79:C7:A8:88:63:8
Alias name: verisignc3g3.pem
  SHA1: 13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
  SHA256:
 EB:04:CF:5E:B1:F3:9A:FA:76:2F:2B:B1:20:F2:96:CB:A5:20:C1:B9:7D:B1:58:95:65:B8:1C:B9:A1:7B:72:4
Alias name: verisignc3g4.pem
  SHA1: 22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
```

```
  SHA256:
 69:DD:D7:EA:90:BB:57:C9:3E:13:5D:C8:5E:A6:FC:D5:48:0B:60:32:39:BD:C4:54:FC:75:8B:2A:26:CF:7F:7
Alias name: verisignc3g5.pem
  SHA1: 4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
  SHA256:
 9A:CF:AB:7E:43:C8:D8:80:D0:6B:26:2A:94:DE:EE:E4:B4:65:99:89:C3:D0:CA:F1:9B:AF:64:05:E4:1A:B7:D
Alias name: verisignc4g2.pem
  SHA1: 0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52
  SHA256:
 44:64:0A:0A:0E:4D:00:0F:BD:57:4D:2B:8A:07:BD:B4:D1:DF:ED:3B:45:BA:AB:A7:6F:78:57:78:C7:01:19:6
Alias name: verisignc4g3.pem
  SHA1: C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
  SHA256:
 E3:89:36:0D:0F:DB:AE:B3:D2:50:58:4B:47:30:31:4E:22:2F:39:C1:56:A0:20:14:4E:8D:96:05:61:79:15:0
Alias name: verisignclass1ca
  SHA1: CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45:3E:64:09:EA:E8:7D:60:F1
  SHA256:
 51:84:7C:8C:BD:2E:9A:72:C9:1E:29:2D:2A:E2:47:D7:DE:1E:3F:D2:70:54:7A:20:EF:7D:61:0F:38:B8:84:2
Alias name: verisignclass1g2ca
  SHA1: 27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
  SHA256:
 34:1D:E9:8B:13:92:AB:F7:F4:AB:90:A9:60:CF:25:D4:BD:6E:C6:5B:9A:51:CE:6E:D0:67:D0:0E:C7:CE:9B:7
Alias name: verisignclass1g3ca
  SHA1: 20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
  SHA256:
 CB:B5:AF:18:5E:94:2A:24:02:F9:EA:CB:C0:ED:5B:B8:76:EE:A3:C1:22:36:23:D0:04:47:E4:F3:BA:55:4B:6
Alias name: verisignclass2g2ca
  SHA1: B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
  SHA256:
 3A:43:E2:20:FE:7F:3E:A9:65:3D:1E:21:74:2E:AC:2B:75:C2:0F:D8:98:03:05:BC:50:2C:AF:8C:2D:9B:41:A
Alias name: verisignclass2g3ca
  SHA1: 61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
  SHA256:
 92:A9:D9:83:3F:E1:94:4D:B3:66:E8:BF:AE:7A:95:B6:48:0C:2D:6C:6C:2A:1B:E6:5D:42:36:B6:08:FC:A1:B
Alias name: verisignclass3ca
  SHA1: A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
  SHA256:
 A4:B6:B3:99:6F:C2:F3:06:B3:FD:86:81:BD:63:41:3D:8C:50:09:CC:4F:A3:29:C2:CC:F0:E2:FA:1B:14:03:0
Alias name: verisignclass3g2ca
  SHA1: 85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
  SHA256:
 83:CE:3C:12:29:68:8A:59:3D:48:5F:81:97:3C:0F:91:95:43:1E:DA:37:CC:5E:36:43:0E:79:C7:A8:88:63:8
Alias name: verisignclass3g3ca
  SHA1: 13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
```

```
    SHA256:
  EB:04:CF:5E:B1:F3:9A:FA:76:2F:2B:B1:20:F2:96:CB:A5:20:C1:B9:7D:B1:58:95:65:B8:1C:B9:A1:7B:72:4
 Alias name: verisignclass3g4ca
    SHA1: 22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
    SHA256:
  69:DD:D7:EA:90:BB:57:C9:3E:13:5D:C8:5E:A6:FC:D5:48:0B:60:32:39:BD:C4:54:FC:75:8B:2A:26:CF:7F:7
 Alias name: verisignclass3g5ca
    SHA1: 4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
    SHA256:
  9A:CF:AB:7E:43:C8:D8:80:D0:6B:26:2A:94:DE:EE:E4:B4:65:99:89:C3:D0:CA:F1:9B:AF:64:05:E4:1A:B7:D
 Alias name: verisignclass3publicprimarycertificationauthorityg4
    SHA1: 22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
    SHA256:
  69:DD:D7:EA:90:BB:57:C9:3E:13:5D:C8:5E:A6:FC:D5:48:0B:60:32:39:BD:C4:54:FC:75:8B:2A:26:CF:7F:7
 Alias name: verisignclass3publicprimarycertificationauthorityg5
    SHA1: 4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
    SHA256:
  9A:CF:AB:7E:43:C8:D8:80:D0:6B:26:2A:94:DE:EE:E4:B4:65:99:89:C3:D0:CA:F1:9B:AF:64:05:E4:1A:B7:D
 Alias name: verisignroot.pem
    SHA1: 36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
    SHA256:
  23:99:56:11:27:A5:71:25:DE:8C:EF:EA:61:0D:DF:2F:A0:78:B5:C8:06:7F:4E:82:82:90:BF:B8:60:E8:4B:3
 Alias name: verisigntsaca
    SHA1: 20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
    SHA256:
  CB:6B:05:D9:E8:E5:7C:D8:82:B1:0B:4D:B7:0D:E4:BB:1D:E4:2B:A4:8A:7B:D0:31:8B:63:5B:F6:E7:78:1A:9
 Alias name: verisignuniversalrootca
    SHA1: 36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
    SHA256:
  23:99:56:11:27:A5:71:25:DE:8C:EF:EA:61:0D:DF:2F:A0:78:B5:C8:06:7F:4E:82:82:90:BF:B8:60:E8:4B:3
 Alias name: verisignuniversalrootcertificationauthority
    SHA1: 36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
    SHA256:
  23:99:56:11:27:A5:71:25:DE:8C:EF:EA:61:0D:DF:2F:A0:78:B5:C8:06:7F:4E:82:82:90:BF:B8:60:E8:4B:3
 Alias name: xrampglobalca
    SHA1: B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
    SHA256:
  CE:CD:DC:90:50:99:D8:DA:DF:C5:B1:D2:09:B7:37:CB:E2:C1:8C:FB:2C:10:C0:FF:0B:CF:0D:32:86:FC:1A:A
 Alias name: xrampglobalcaroot
    SHA1: B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
    SHA256:
  CE:CD:DC:90:50:99:D8:DA:DF:C5:B1:D2:09:B7:37:CB:E2:C1:8C:FB:2C:10:C0:FF:0B:CF:0D:32:86:FC:1A:A
```

使用 AWS WAF 保护 API

AWS WAF 是一个 Web 应用程序防火墙，可帮助保护 Web 应用程序和 API 免受攻击。通过它，您可以配置一组规则（称为 Web 访问控制列表，即 Web ACL），基于可自定义的 Web 安全规则以及您定义的条件，允许、阻止或统计 Web 请求。有关更多信息，请参阅 AWS WAF 的工作原理。

您可以使用 AWS WAF 保护 API Gateway REST API 免受常见的 Web 漏洞攻击，例如 SQL 注入和跨站脚本攻击（XSS）。这些威胁可能会影响 API 的可用性和性能、损害安全性或消耗过多的资源。例如，您可以创建规则以允许或阻止以下请求：来自指定 IP 地址范围的请求；来自 CIDR 块的请求；源自特定国家/地区或区域的请求；包含恶意 SQL 代码的请求；或者包含恶意脚本的请求。

您还可以创建与 HTTP 标头、方法、查询字符串、URI 和请求正文中的指定字符串或正则表达式模式匹配的规则（限制为前 64 KB）。此外，您可以创建规则来阻止来自特定用户代理、恶意机器人和内容抓取程序的攻击。例如，您可以使用基于速率的规则来指定每个客户端 IP 在尾随的、不断更新的 5 分钟期间内允许的 Web 请求数。

> ⚠ Important
>
> AWS WAF 是抵御 Web 漏洞攻击的第一道防线。在 API 上启用 AWS WAF 时，会先评估 AWS WAF 规则，然后再评估其他访问控制特征，例如资源策略、IAM 策略、Lambda 授权方和 Amazon Cognito 授权方。例如，如果 AWS WAF 阻止从资源策略允许的 CIDR 块进行访问，AWS WAF 将优先进行并且不评估资源策略。

要为 API 启用 AWS WAF，您需要执行以下操作：

1. 使用 AWS WAF 控制台、AWS SDK 或 CLI 创建一个 Web ACL，其中包含 AWS WAF 托管式规则和您自己的自定义规则的所需组合。有关更多信息，请参阅 Getting Started with AWS WAF 和 Web access control lists (web ACLs)。

   > ⚠ Important
   >
   > API Gateway 需要区域应用程序的 AWS WAFV2 Web ACL 或 AWS WAF Classic Regional Web ACL。

2. 将 AWS WAF Web ACL 与 API 阶段关联。您可以使用 AWS WAF 控制台、AWS SDK、CLI 或使用 API Gateway 控制台来完成此操作。

使用 API Gateway 控制台将 AWS WAF Web ACL 与 API Gateway API 阶段相关联

要使用 API Gateway 控制台将 AWS WAF Web ACL 与现有的 API Gateway API 阶段相关联，请按以下步骤操作：

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择现有 API 或者创建新 API。
3. 在主导航窗格中，选择阶段，然后选择一个阶段。
4. 在阶段详细信息部分中，选择编辑。
5. 在 Web 应用程序防火墙（AWS WAF）下，选择您的 Web ACL。

   如果您正在使用 AWS WAFV2，请为区域应用程序选择 AWS WAFV2 Web ACL。Web ACL 及其使用的任何其它 AWS WAFV2 资源都必须与您的 API 位于同一区域中。

   如果您使用的是 AWS WAF Classic Regional，请选择区域 Web ACL。
6. 选择 Save changes（保存更改）。

使用 AWS CLI 将 AWS WAF Web ACL 与 API Gateway API 阶段相关联

要使用 AWS CLI 将区域应用程序的 AWS WAFV2 Web ACL 与现有的 API Gateway API 阶段相关联，请调用 associate-web-acl 命令，如以下示例中所示：

```
aws wafv2 associate-web-acl \
--web-acl-arn  arn:aws:wafv2:{region}:111122223333:regional/webacl/test-cli/
a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
--resource-arn arn:aws:apigateway:{region}::/restapis/4wk1k4onj3/stages/prod
```

要使用 AWS CLI 将 AWS WAF Classic Regional Web ACL 与现有的 API Gateway API 阶段相关联，请调用 associate-web-acl 命令，如以下示例中所示：

```
aws waf-regional associate-web-acl \
--web-acl-id 'aabc123a-fb4f-4fc6-becb-2b00831cadcf' \
--resource-arn 'arn:aws:apigateway:{region}::/restapis/4wk1k4onj3/stages/prod'
```

使用 AWS WAF REST API 将 AWS WAF Web ACL 与 API 阶段相关联

要使用 AWS WAFV2 REST API 将区域应用程序的 AWS WAFV2 Web ACL 与现有的 API Gateway API 阶段相关联，请使用 AssociateWebACL 命令，如以下示例中所示：

```
import boto3

wafv2 = boto3.client('wafv2')

wafv2.associate_web_acl(
    WebACLArn='arn:aws:wafv2:{region}:111122223333:regional/webacl/test/abc6aa3b-
fc33-4841-b3db-0ef3d3825b25',
    ResourceArn='arn:aws:apigateway:{region}::/restapis/4wk1k4onj3/stages/prod'
)
```

要使用 AWS WAF REST API 将 AWS WAF Classic Regional Web ACL 与现有的 API Gateway API 阶段相关联，请使用 AssociateWebACL 命令，如以下示例中所示：

```
import boto3

waf = boto3.client('waf-regional')

waf.associate_web_acl(
    WebACLId='aabc123a-fb4f-4fc6-becb-2b00831cadcf',
    ResourceArn='arn:aws:apigateway:{region}::/restapis/4wk1k4onj3/stages/prod'
)
```

限制 API 请求以获得更高的吞吐量

您可以为 API 配置节流和配额，以帮助防止它们因过多请求而不堪重负。节流和配额都是在尽最大努力的基础上应用的，应被视为目标而不是保证的请求上限。

API Gateway 使用令牌桶算法（其中，一个令牌即一个请求）限制对 API 的请求。具体来说，API Gateway 根据您账户中的所有 API，按区域检查请求提交的速率和突发事件。在令牌桶算法中，突发可以允许这些限制的预定义超出，但在某些情况下，其他因素也可能导致限制超支。

如果请求提交超过稳态请求速率和突增限制，则 API Gateway 将开始限制请求。此时客户可能会收到 `429 Too Many Requests` 个错误响应。捕获此类异常后，客户端能够以限制速率的方式重新提交失败的请求。

作为 API 开发人员，您可以针对各个 API 阶段或方法设置目标限制，以提高账户中所有 API 的整体性能。或者，您可以启用使用计划，根据指定的请求速率和配额设置客户端请求提交的节流。

主题

- 如何在 API Gateway 中应用限制设置
- 每个区域的账户级别限制
- 在使用计划中配置 API 级别限制和阶段级别节流目标
- 配置阶段级别节流目标
- 在使用计划中配置方法级别节流目标

如何在 API Gateway 中应用限制设置

在为 API 配置限制和配额设置之前，了解 Amazon API Gateway 如何应用这些节流和配额设置很有用。

Amazon API Gateway 提供四种基本类型的限制相关设置：

- AWS 节流限制适用于某个区域的所有账户和客户。这些限制设置旨在防止您的 API 和账户因过多请求而不堪重负。这些限制是由 AWS 设置，且客户无法更改。
- 每个账户限制适用于指定区域内账户中的所有 API。客户可以请求我们放宽账户级别的速率限制 —— 如果具有更短的超时和较小的有效负载的 API，则可以提高限制。要请求增加每个区域的账户级别限制，请联系 AWS Support 中心。有关更多信息，请参阅 配额和重要提示。请注意，这些限制不能高于 AWS 节流限制。
- 每个 API、每阶段的节流限制应用于某个阶段的 API 方法级别。您可以为所有方法配置相同的设置，也可以为每种方法配置不同的限制设置。请注意，这些限制不能高于 AWS 节流限制。
- 每客户端限制应用于将与使用计划关联的 API 密钥用作客户端标识符的客户端。请注意，这些限制不能高于每个账户限制。

API Gateway 限制相关的设置将按以下顺序应用：

1. 在使用计划中为 API 阶段设置的每个客户端或每个方法限制
2. 为 API 阶段设置的每个方法的节流限制
3. 每个区域的账户级别限制
4. AWS 区域节流

每个区域的账户级别限制

默认情况下，API Gateway 针对每个区域限制 AWS 账户内所有 API 的每秒稳态请求 (RPS)。它还对于每个区域限制一个 AWS 账户中所有 API 的突增（即最大存储桶大小）。在 API Gateway 中，突增限制代表 API Gateway 在返回 `429 Too Many Requests` 错误响应之前可以完成目标的最大并发请求提交数量。有关限制配额的更多信息，请参阅配额和重要提示。

在使用计划中配置 API 级别限制和阶段级别节流目标

在使用计划中，您可以在 API 或阶段级别为所有方法设置每种方法的节流目标。您可以指定节流速率，也即，将令牌添加到令牌桶的速率（以每秒请求数为单位）。您还可以指定节流突发，即令牌桶的容量。

您可以使用 AWS CLI、SDK 和 AWS Management Console来创建使用计划。有关如何创建使用计划的更多信息，请参阅???。

配置阶段级别节流目标

您可以使用 AWS CLI、SDK 和 AWS Management Console 来创建阶段级别的节流目标。

有关如何使用 AWS Management Console创建阶段级节流目标的更多信息，请参阅???。有关如何使用 AWS CLI 创建阶段级节流目标的更多信息，请参阅 create-stage。

在使用计划中配置方法级别节流目标

您可以在使用计划中的方法级别设置其他目标，如 创建使用计划 中所示。在 API Gateway 控制台中，通过在配置方法限制设置中指定 `Resource=<resource>` 和 `Method=<method>` 来设置这些限制。例如，对于 PetStore 示例，您可以指定 `Resource=/pets` 和 `Method=GET`。

Amazon API Gateway 中的私有 REST API

私有 API 是指只能从 Amazon VPC 内部调用的 REST API。您可以使用接口 VPC 端点访问 API，该端点是您在 VPC 中创建的端点网络接口。接口端点由 AWS PrivateLink 提供支持，您可以使用该技术通过私有 IP 地址私密访问 AWS。

您还可以使用 AWS Direct Connect 建立从本地网络到 Amazon VPC 的连接，然后通过该连接访问私有 API。在所有情况下，私有 API 的流量使用安全连接，并与公共互联网隔离开来。流量不会离开 Amazon 网络。

私有 API 的最佳实践

在创建私有 API 时，我们建议您使用以下最佳实践：

- 使用单个 VPC 端点访问多个私有 API。这样可减少您可能需要的 VPC 端点的数量。
- 将您的 VPC 端点与您的 API 关联。这样可创建 Route 53 别名 DNS 记录，并简化调用私有 API 的过程。
- 为您的 VPC 开启私有 DNS。这样，您就可以在 VPC 内调用 API，而不必传递 Host 或 `x-apigw-api-id` 标头。如果您选择不启用私有 DNS，您将只能通过公有 DNS 访问您的 API。
- 将私有 API 的访问权限限制为特定 VPC 或 VPC 端点。向 API 的资源策略中添加 `aws:SourceVpc` 或 `aws:SourceVpce` 条件来限制访问权限。
- 为了最大程度地保障数据边界的安全，您可以创建 VPC 端点策略。这可以控制对 VPC 端点的访问，从而限制调用私有 API。

私有 API 的注意事项

以下注意事项可能会影响您对私有 API 的使用：

- 仅支持 REST API。
- 私有 API 不支持自定义域名。
- 您不能将私有 API 转换为边缘优化 API。
- 私有 API 仅支持 TLS 1.2。不支持早期 TLS 版本。
- 私有 API 的 VPC 端点与其他接口 VPC 端点的限制相同。有关更多信息，请参阅《AWS PrivateLink 指南》中的使用接口 VPC 终端节点访问 AWS 服务。有关将 API Gateway 用于共享 VPC 和共享子网的更多信息，请参阅《AWS PrivateLink 指南》中的共享子网。

私有 API 的后续步骤

要了解如何创建私有 API 以及如何关联 VPC 端点，请参阅the section called “创建私有 API”。要了解在 AWS CloudFormation 中创建依赖项以及在 AWS Management Console中创建私有 API 的教程，请参阅the section called “教程：构建私有 REST API”。

创建私有 API

在创建私有 API 之前，您首先为 API Gateway 创建一个 VPC 端点。接下来，您要创建私有 API 并向其附加资源策略。（可选）您可以将 VPC 端点与私有 API 关联，来简化调用 API 的过程。最后，您部署 API。

以下过程介绍了如何执行完成此操作。您可以使用 AWS Management Console、AWS CLI 或 AWS SDK 创建私有 REST API。

先决条件

要执行以下步骤，您必须拥有完全配置的 VPC。要了解如何创建 VPC，请参阅《Amazon VPC 用户指南》中的仅创建 VPC。要在创建 VPC 时遵循所有推荐的步骤，请启用私有 DNS。这样，您就可以在 VPC 内调用 API，而不必传递 Host 或 `x-apigw-api-id` 标头。

要启用私有 DNS，VPC 的 `enableDnsSupport` 和 `enableDnsHostnames` 属性必须设置为 `true`。有关更多信息，请参阅 VPC 中的 DNS 支持和更新 VPC 的 DNS 支持。

步骤 1：在 VPC 中为 API Gateway 创建 VPC 端点

以下过程展示如何为 API Gateway 创建 VPC 端点。要为 API Gateway 创建 VPC 端点，您需要为在其中创建私有 API 的 AWS 区域指定 `execute-api` 域。`execute-api` 域是用于 API 执行的 API Gateway 组件服务。

在为 API Gateway 创建 VPC 端点时，需要指定 DNS 设置。如果您关闭私有 DNS，则只能使用公有 DNS 访问您的 API。有关更多信息，请参阅 the section called “问题：我无法从 API Gateway VPC 端点连接到我的公有 API”。

AWS Management Console

为 API Gateway 创建接口 VPC 端点

1. 通过 https://console.aws.amazon.com/vpc/ 登录到AWS Management Console并打开 Amazon VPC 控制台。
2. 在导航窗格中的虚拟私有云下，选择端点。
3. 选择创建端点。
4. （可选）对于名称标签，输入名称以协助标识您的 VPC 端点。
5. 对于服务类别，选择 AWS 服务。
6. 在服务下的搜索栏中，输入 **`execute-api`**。然后，在您将创建 API 的 AWS 区域中选择 API Gateway 服务端点。服务名称应类似于 `com.amazonaws.us-east-1.execute-api`，类型应为接口。
7. 对于 VPC，选择要在其中创建端点的 VPC。
8. （可选）要关闭启用私有 DNS 名称，请选择其他设置，然后清除启用私有 DNS 名称。
9. 对于子网，选择在其中创建了端点网络接口的可用区。要提高 API 的可用性，请选择多个子网。
10. 对于安全组，选择要与 VPC 端点网络接口关联的安全组。

您选择的安全组必须设置为允许来自您的 VPC 中的 IP 范围或您的 VPC 中的其他安全组的 TCP 端口 443 入站 HTTPS 通信。

11. 对于策略，请执行以下操作之一：
    - 如果您尚未创建私有 API 或者不想配置自定义 VPC 端点策略，请选择完全访问权限。
    - 如果您已经创建了私有 API 并想要配置自定义 VPC 端点策略，则可以输入自定义 VPC 端点策略。有关更多信息，请参阅 the section called “为私有 API 使用 VPC 端点策略”。

    创建 VPC 端点后，您可以更新 VPC 端点策略。有关更多信息，请参阅 Update a VPC endpoint policy。
12. 选择创建端点。
13. 复制生成的 VPC 端点 ID，以便在将来的步骤中使用。

AWS CLI

以下 create-vpc-endpoint 命令可用于创建 VPC 端点：

```
aws ec2 create-vpc-endpoint \
    --vpc-id vpc-1a2b3c4d \
    --vpc-endpoint-type Interface \
    --service-name com.amazonaws.us-east-1.execute-api \
    --subnet-ids subnet-7b16de0c \
    --security-group-id sg-1a2b3c4d
```

复制生成的 VPC 端点 ID，以便在将来的步骤中使用。

步骤 2：创建私有密钥

创建 VPC 端点后，您创建一个私有 REST API。以下过程说明了如何创建私有 API。

AWS Management Console

要创建私有 API

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择创建 API。
3. 在 REST API 下，选择 Build (生成)。

4. 对于名称，输入名称。
5. （可选）对于描述，输入描述。
6. 对于 API 端点类型，选择私有。
7. （可选）对于 VPC 端点 ID，输入 VPC 端点 ID。

   如果您将 VPC 端点 ID 与私有 API 关联，则可以从 VPC 内调用 API，而无需覆盖 `Host` 标头或传递 `x-apigw-api-id header`。有关更多信息，请参阅the section called "（可选）将 VPC 端点与私有 API 关联或取消关联"。
8. 选择创建 API。

完成前面的步骤之后，您可以按照the section called "开始使用 REST API 控制台"中的说明来为此 API 设置方法和集成，但您无法部署 API。要部署您的 API，请执行步骤 3 并将资源策略附加到 API。

AWS CLI

以下 update-rest-api 命令显示如何创建私有 API：

```
aws apigateway create-rest-api \
        --name 'Simple PetStore (AWS CLI, Private)' \
        --description 'Simple private PetStore API' \
        --region us-west-2 \
        --endpoint-configuration '{ "types": ["PRIVATE"] }'
```

成功调用返回类似于以下内容的输出：

```
{
    "createdDate": "2017-10-13T18:41:39Z",
    "description": "Simple private PetStore API",
    "endpointConfiguration": {
        "types": "PRIVATE"
    },
    "id": "0qzs2sy7bh",
    "name": "Simple PetStore (AWS CLI, Private)"
}
```

完成前面的步骤之后，您可以按照the section called "教程：使用 AWS SDK 或 AWS CLI 设置边缘优化的 API"中的说明来为此 API 设置方法和集成，但您无法部署 API。要部署您的 API，请执行步骤 3 并将资源策略附加到 API。

SDK JavaScript v3

以下示例说明了如何使用适用于 JavaScript 的 AWS SDK v3 创建私有 API：

```
import {APIGatewayClient, CreateRestApiCommand} from "@aws-sdk/client-api-gateway";
const apig = new APIGatewayClient({region:"us-east-1"});

const input = { // CreateRestApiRequest
  name: "Simple PetStore (JavaScript v3 SDK, private)", // required
  description: "Demo private API created using the AWS SDK for JavaScript v3",
  version: "0.00.001",
  endpointConfiguration: { // EndpointConfiguration
    types: [ "PRIVATE"],
  },
};

export const handler = async (event) => {
const command = new CreateRestApiCommand(input);
try {
  const result = await apig.send(command);
  console.log(result);
} catch (err){
  console.error(err)
 }
};
```

成功调用返回类似于以下内容的输出：

```
{
  apiKeySource: 'HEADER',
  createdDate: 2024-04-03T17:56:36.000Z,
  description: 'Demo private API created using the AWS SDK for JavaScript v3',
  disableExecuteApiEndpoint: false,
  endpointConfiguration: { types: [ 'PRIVATE' ] },
  id: 'abcd1234',
  name: 'Simple PetStore (JavaScript v3 SDK, private)',
  rootResourceId: 'efg567',
  version: '0.00.001'
}
```

完成前面的步骤之后，您可以按照the section called “教程：使用 AWS SDK 或 AWS CLI 设置边缘优化的 API”中的说明来为此 API 设置方法和集成，但您无法部署 API。要部署您的 API，请执行步骤 3 并将资源策略附加到 API。

Python SDK

以下示例说明了如何使用适用于 Python 的 AWS SDK 创建私有 API：

```
import json
import boto3
import logging

logger = logging.getLogger()
apig = boto3.client('apigateway')

def lambda_handler(event, context):
    try:
      result = apig.create_rest_api(
      name='Simple PetStore (Python SDK, private)',
      description='Demo private API created using the AWS SDK for Python',
      version='0.00.001',
      endpointConfiguration={
          'types': [
             'PRIVATE',
          ],
      },
      )
    except botocore.exceptions.ClientError as error:
           logger.exception("Couldn't create private API %s.", error)
           raise
    attribute=["id", "name", "description", "createdDate", "version",
 "apiKeySource", "endpointConfiguration"]
    filtered_data ={key:result[key] for key in attribute}
    result = json.dumps(filtered_data, default=str, sort_keys='true')
    return result
```

成功调用返回类似于以下内容的输出：

```
"{\"apiKeySource\": \"HEADER\", \"createdDate\": \"2024-04-03 17:27:05+00:00\",
 \"description\": \"Demo private API created using the AWS SDK for \",
 \"endpointConfiguration\": {\"types\": [\"PRIVATE\"]}, \"id\": \"abcd1234\", \"name
\": \"Simple PetStore (Python SDK, private)\", \"version\": \"0.00.001\"}"
```

完成前面的步骤之后，您可以按照the section called “教程：使用 AWS SDK 或 AWS CLI 设置边缘优化的 API”中的说明来为此 API 设置方法和集成，但您无法部署 API。要部署您的 API，请执行步骤 3 并将资源策略附加到 API。

步骤 3：为私有 API 设置资源策略

所有 VPC 都无法访问您当前的私有 API。使用资源策略授予您的 VPC 和 VPC 端点访问私有 API 的权限。您可以向任何 AWS 账户中的 VPC 端点授予访问权限。

您的资源策略应包含 `aws:SourceVpc` 或 `aws:SourceVpce` 条件来限制访问权限。我们建议您标识特定的 VPC 和 VPC 端点，而不要创建允许所有 VPC 和 VPC 端点访问的资源策略。

以下过程展示如何向 API 附加资源策略。

AWS Management Console

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 REST API。
3. 在主导航窗格中，选择资源策略。
4. 选择创建策略。
5. 选择选择模板，然后选择源 VPC。
6. 将 `{{vpceID}}`（包括大括号）替换为您的 VPC 端点 ID。
7. 选择 Save changes（保存更改）。

AWS CLI

以下 update-rest-api 命令显示如何将资源策略附加到现有 API：

```
aws apigateway update-rest-api \
    --rest-api-id a1b2c3 \
    --patch-operations op=replace,path=/
policy,value='"{\"jsonEscapedPolicyDocument\"}"'
```

您可能还要控制哪些资源可以访问您的 VPC 端点。要控制哪些资源可以访问 VPC 端点，请向 VPC 端点附加端点策略。有关更多信息，请参阅 the section called “为私有 API 使用 VPC 端点策略”。

（可选）将 VPC 端点与私有 API 关联或取消关联

当您将 VPC 端点与私有 API 关联时，API Gateway 将生成新的 Route 53 别名 DNS 记录。您可以使用此记录调用私有 API，就像使用公有 API 一样，而无需覆盖 `Host` 标头或传递 `x-apigw-api-id` 标头。

生成的基本 URL 采用以下格式：

```
https://{rest-api-id}-{vpce-id}.execute-api.{region}.amazonaws.com/{stage}
```

Associate a VPC endpoint (AWS Management Console)

您可以在创建私有 API 时或创建后，将 VPC 端点与私有 API 关联起来。以下过程展示如何将 VPC 端点与先前创建的 API 相关联。

将 VPC 端点与私有 API 关联

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择您的私有 API。
3. 在主导航窗格中，选择资源策略。
4. 编辑您的资源策略以允许来自其他 VPC 端点的调用。
5. 在主导航窗格中，选择 API 设置。
6. 在 API 详细信息部分中，选择编辑。
7. 对于 VPC 端点 ID，请选择其他 VPC 端点 ID。
8. 选择保存。
9. 重新部署 API 以使更改生效。

Dissociate a VPC endpoint (AWS Management Console)

将 VPC 端点与私有 REST API 取消关联

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择您的私有 API。
3. 在主导航窗格中，选择资源策略。

4. 编辑您的资源策略，对于您要与私有 API 取消关联的 VPC 端点，删除提及该端点的内容。
5. 在主导航窗格中，选择 API 设置。
6. 在 API 详细信息部分中，选择编辑。
7. 对于 VPC 端点 ID，选择 X 以与 VPC 端点取消关联。
8. 选择保存。
9. 重新部署 API 以使更改生效。

Associate a VPC endpoint (AWS CLI)

以下 create-rest-api 命令显示如何在创建 API 时关联 VPC 端点。

```
aws apigateway create-rest-api \
    --name Petstore \
    --endpoint-configuration '{ "types": ["PRIVATE"], "vpcEndpointIds" :
 ["vpce-0212a4ababd5b8c3e", "vpce-0393a628149c867ee"] }' \
    --region us-west-2
```

输出将与以下内容类似：

```
{
    "apiKeySource": "HEADER",
    "endpointConfiguration": {
        "types": [
            "PRIVATE"
        ],
        "vpcEndpointIds": [
            "vpce-0212a4ababd5b8c3e",
            "vpce-0393a628149c867ee"
        ]
    },
    "id": "u67n3ov968",
    "createdDate": 1565718256,
    "name": "Petstore"
}
```

以下 update-rest-api 命令显示如何将 VPC 端点与已经创建的 API 关联：

```
aws apigateway update-rest-api \
```

```
    --rest-api-id u67n3ov968 \
    --patch-operations "op='add',path='/endpointConfiguration/
vpcEndpointIds',value='vpce-01d622316a7df47f9'" \
    --region us-west-2
```

输出将与以下内容类似：

```
{
    "name": "Petstore",
    "apiKeySource": "1565718256",
    "tags": {},
    "createdDate": 1565718256,
    "endpointConfiguration": {
        "vpcEndpointIds": [
            "vpce-0212a4ababd5b8c3e",
            "vpce-0393a628149c867ee",
            "vpce-01d622316a7df47f9"
        ],
        "types": [
            "PRIVATE"
        ]
    },
    "id": "u67n3ov968"
}
```

重新部署 API 以使更改生效。

Disassociate a VPC endpoint (AWS CLI)

以下 update-rest-api 命令显示如何将 VPC 端点与私有 API 取消关联：

```
aws apigateway update-rest-api \
    --rest-api-id u67n3ov968 \
    --patch-operations "op='remove',path='/endpointConfiguration/
vpcEndpointIds',value='vpce-0393a628149c867ee'" \
    --region us-west-2
```

输出将与以下内容类似：

```
{
    "name": "Petstore",
    "apiKeySource": "1565718256",
```

```
    "tags": {},
    "createdDate": 1565718256,
    "endpointConfiguration": {
        "vpcEndpointIds": [
            "vpce-0212a4ababd5b8c3e",
            "vpce-01d622316a7df47f9"
        ],
        "types": [
            "PRIVATE"
        ]
    },
    "id": "u67n3ov968"
}
```

重新部署 API 以使更改生效。

步骤 4：部署私有 API

要部署 API，您可以创建 API 部署并将其与阶段关联。以下过程显示如何部署私有 API。

AWS Management Console

部署私有 API

1. 选择 API。
2. 选择部署 API。
3. 对于阶段，选择新建阶段。
4. 对于阶段名称，输入阶段名称。
5. （可选）对于描述，输入描述。
6. 选择部署。

AWS CLI

以下 create-deployment 命令显示如何部署私有 API：

```
aws apigateway create-deployment --rest-api-id a1b2c3 \
  --stage-name test \
  --stage-description 'Private API test stage' \
  --description 'First deployment'
```

对私有 API 进行故障排除

以下内容为您在创建私有 API 时可能遇到的错误和问题提供故障排除建议。

问题：我无法从 API Gateway VPC 端点连接到我的公有 API

创建 VPC 时，您可以配置 DNS 设置。我们建议您为 VPC 开启私有 DNS。如果您选择关闭私有 DNS，您只能通过公有 DNS 访问您的 API。

如果您启用私有 DNS，则无法从 VPC 端点访问公有 API Gateway API 的默认端点。您可以使用自定义域名访问 API。

如果您创建区域自定义域名，请使用 A 类型别名记录；如果您创建边缘优化的自定义域名，则不限制记录类型。您可以在启用私有 DNS 的情况下访问这些公有 API。有关更多信息，请参阅问题：我无法从 API Gateway VPC 端点连接到我的公有 API。

问题：我的 API 返回 **`{"Message":"User: anonymous is not authorized to perform: execute-api:Invoke on resource: arn:aws:execute-api:us-east-1:********/****/****/"}`**

在资源策略中，如果您将主体设置为 AWS 主体，如下所示：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            ""Principal": {
                "AWS": [
                    "arn:aws:iam::account-id:role/developer",
                    "arn:aws:iam::account-id:role/Admin"
                ]
            },
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ]
        },
        ...
        }
```

您必须对 API 中的每个方法使用 `AWS_IAM` 授权，否则 API 会返回之前的错误消息。有关如何为方法开启 `AWS_IAM` 授权的更多说明，请参阅 the section called “方法”。

问题：我无法分辨我的 VPC 端点是否与我的 API 关联

如果您将 VPC 端点与私有 API 关联或取消关联，则需要重新部署 API。由于 DNS 传播，更新操作可能需要几分钟才能完成。在这段时间内，您的 API 将可用，但新生成的 DNS URL 的 DNS 传播可能仍在进行中。如果几分钟后，您的新 URL 仍无法在 DNS 中解析，我们建议您重新部署 API。

调用私有 API

您只能从 VPC 内部调用私有 API。私有 API 必须具有允许特定 VPC 和 VPC 端点调用 API 的资源策略。

您可以通过下列方式调用私有 API：

- 使用 Route53 别名调用 API。只有将您的 VPC 端点与 API 关联后，此功能才可用。有关更多信息，请参阅 the section called "（可选）将 VPC 端点与私有 API 关联或取消关联"。
- 使用私有 DNS 调用 API。仅当您为 VPC 启用私有 DNS 时，此功能才可用。
- 使用 AWS Direct Connect 调用 API。
- 使用特定于端点的公有 DNS 主机名调用您的 API。

要使用 DNS 名称调用私有 API，您需要识别 API 的 DNS 名称。以下步骤将演示如何查找您的 DNS 名称。

AWS Management Console

查找 DNS 名称

1. 通过 https://console.aws.amazon.com/vpc/ 登录到AWS Management Console并打开 Amazon VPC 控制台。
2. 在主导航窗格中，选择端点，然后为 API Gateway 选择接口 VPC 端点。
3. 在详细信息窗格中，您将看到 DNS 名称字段中有五个值。前三个值是您的 API 的公有 DNS 名称。另外两个值则是 API 的私有 DNS 名称。

AWS CLI

使用以下 describe-vpc-endpoints 命令列出您的 DNS 值。

```
aws ec2 describe-vpc-endpoints --filters vpc-endpoint-id=vpce-01234567abcdef012
```

前三个值是您的 API 的公有 DNS 名称。另外两个值则是 API 的私有 DNS 名称。

使用 Route53 别名调用私有 API

您可以将 VPC 端点与私有 API 关联或取消关联。有关更多信息，请参阅 the section called “（可选）将 VPC 端点与私有 API 关联或取消关联”。

将 VPC 端点与私有 API 关联后，您可以使用以下基本 URL 来调用 API：

```
https://{rest-api-id}-{vpce-id}.execute-api.{region}.amazonaws.com/{stage}
```

例如，如果您为 `test` 阶段设置了 `GET /pets` 方法，并且 REST API ID 为 `01234567ab`，VPC 端点 ID 为 `vpce-01234567abcdef012`，区域为 `us-west-2`，则可以按以下方式调用您的 API：

```
curl -v https://01234567ab-vpce-01234567abcdef012.execute-api.us-west-2.amazonaws.com/
test/pets
```

使用私有 DNS 名称调用私有 API

如果您已启用了私有 DNS，则可以使用以下私有 DNS 名称访问私有 API：

```
{restapi-id}.execute-api.{region}.amazonaws.com
```

用于调用 API 的基本 URL 采用以下格式：

```
https://{restapi-id}.execute-api.{region}.amazonaws.com/{stage}
```

例如，如果您为 `test` 阶段设置了 `GET /pets` 方法，并且 REST API ID 为 `01234567ab`，区域为 `us-west-2`，则可以通过在浏览器中输入以下 URL 来调用私有 API：

```
https://01234567ab.execute-api.us-west-2.amazonaws.com/test/pets
```

或者，您可以使用以下 cURL 命令来调用私有 API：

```
curl -X GET https://01234567ab.execute-api.us-west-2.amazonaws.com/test/pets
```

> ⚠ Warning
>
> 如果您为 VPC 端点启用私有 DNS，您将无法访问公有 API 的默认端点。有关更多信息，请参阅为何无法从 API Gateway VPC 端点连接到我的公有 API？

使用 AWS Direct Connect 调用私有 API

您可以使用 AWS Direct Connect 建立从本地网络到 Amazon VPC 的专用私有连接，并使用公有 DNS 名称通过该连接访问私有 API 端点。

您还可以使用私有 DNS 名称从本地网络访问您的私有 API，方法是设置 Amazon Route 53 Resolver 入站端点并从远程网络转发所有私有 DNS 的 DNS 查询。有关更多信息，请参阅 Amazon Route 53 开发人员指南中的将入站 DNS 查询转发到您的 VPC。

使用特定于端点的公有 DNS 主机名调用私有 API

您可以使用特定于端点的 DNS 主机名访问您的私有 API。这些是包含您的私有 API 的 VPC 端点 ID 或 API ID 的公有 DNS 主机名。

生成的基本 URL 采用以下格式：

```
https://{public-dns-hostname}.execute-api.{region}.vpce.amazonaws.com/{stage}
```

例如，如果您为 `test` 阶段设置 GET `/pets` 方法，并且 REST API ID 为 `abc1234`，其公有 DNS 主机名为 `vpce-def-01234567`，区域为 `us-west-2`，则可以通过在 cURL 命令中使用 `Host` 标头，使用 VPCe ID 调用私有 API：

```
curl -v https://vpce-def-01234567.execute-api.us-west-2.vpce.amazonaws.com/test/pets -H
 'Host: abc1234.execute-api.us-west-2.amazonaws.com'
```

或者，您可以在 cURL 命令中使用 `x-apigw-api-id` 标头，以下面的格式通过 API ID 调用私有 API：

```
curl -v https://{public-dns-hostname}.execute-api.{region}.vpce.amazonaws.com/{stage} -
H 'x-apigw-api-id:{api-id}'
```

监控 REST API

在本节中，您可以了解如何使用 CloudWatch 指标、CloudWatch Logs、Firehose 和 AWS X-Ray 监控您的 API。通过结合 CloudWatch 执行日志和 CloudWatch 指标，您可以记录错误和执行跟踪，并监控 API 的性能。您可能还需要将 API 调用记录到 Firehose。您也可以使用 AWS X-Ray，通过构成 API 的下游服务跟踪调用。

> **Note**
>
> 在以下情况下，API Gateway 可能无法生成日志和指标：
>
> - “413 请求实体过大”错误
> - 过多的“429 请求太多”错误
> - 发送到没有 API 映射的自定义域的请求中出现 400 系列错误
> - 由内部故障造成的 500 系列错误
>
> 在测试 REST API 方法时，API Gateway 不会生成日志和指标。模拟 CloudWatch 条目。有关更多信息，请参阅 the section called “使用控制台测试 REST API 方法”。

主题

- 使用 Amazon CloudWatch 指标监控 REST API 执行
- 在 API Gateway 中为 REST API 设置 CloudWatch 日志记录
- 将 API 调用记录到 Amazon Data Firehose
- 使用 X-Ray 跟踪用户对 REST API 的请求

使用 Amazon CloudWatch 指标监控 REST API 执行

您可以使用 CloudWatch 监控 API 执行，这将从 API Gateway 收集原始数据，并将数据处理为便于阅读的近乎实时的指标。这些统计数据会保存 15 个月，从而使您能够访问历史信息，并能够更好地了解您的 Web 应用程序或服务的执行情况。默认情况下，API Gateway 指标数据会在一分钟时段内自动发送到 CloudWatch。有关更多信息，请参阅 Amazon CloudWatch 用户指南 中的什么是 Amazon CloudWatch？

您可以通过多种方式分析 API Gateway 报告的指标所提供的信息。下面的列表显示了指标的一些常见用法，这些内容是帮助您开始使用的建议：

- 监控 IntegrationLatency 指标以衡量后端的响应能力。
- 监控 Latency 指标以衡量 API 调用的整体响应能力。
- 监控 CacheHitCount 和 CacheMissCount 指标以优化缓存容量，从而实现所需的性能。

主题

- Amazon API Gateway 维度和指标
- 使用 API Gateway 中的 API 控制面板查看 CloudWatch 指标
- 在 CloudWatch 控制台中查看 API Gateway 指标
- 在 CloudWatch 控制台中查看 API Gateway 日志事件
- 中的监控工具AWS

Amazon API Gateway 维度和指标

下面列出 API Gateway 发送给 Amazon CloudWatch 的指标和维度。有关更多信息，请参阅 使用 Amazon CloudWatch 指标监控 REST API 执行。

API Gateway 指标

Amazon API Gateway 每分钟向 CloudWatch 发送一次指标数据。

`AWS/ApiGateway` 命名空间包括以下指标。

指标	说明
4XXError	在给定期间捕获的客户端错误数。 API Gateway 将修改后的网关响应状态代码计为 4XXError 错误。 `Sum` 统计数据表示此指标，即给定期间内 4XXError 错误的总计数。`Average` 统计数据表示 4XXError 错误率，即 4XXError 错误的总计数除以该期间中的请求总数。分母对应于 Count 指标 (见下)。 Unit: Count
5XXError	在给定期间捕获的服务器端错误数。 `Sum` 统计数据表示此指标，即给定期间内 5XXError 错误的总计数。`Average` 统计数据表示 5XXError 错误率，即 5XXError 错误的总计数除以该期间中的请求总数。分母对应于 Count 指标 (见下)。 Unit: Count

指标	说明
CacheHitCount	在给定期间内从 API 缓存中提供的请求数。 `Sum` 统计数据表示此指标，即给定期间内缓存命中的总计数。`Average` 统计数据表示缓存命中率，即缓存命中的总计数除以该期间中的请求总数。分母对应于 Count 指标 (见下)。 Unit: Count
CacheMissCount	在启用 API 缓存时，在给定期间内由后端所服务的请求的数量。 `Sum` 统计数据表示此指标，即指定期间内缓存未命中的总计数。`Average` 统计数据表示缓存未命中率，即缓存未命中的总计数除以该期间中的请求总数。分母对应于 Count 指标 (见下)。 Unit: Count
Count	给定期间内的 API 请求总数。 `SampleCount` 统计数据表示此指标。 Unit: Count
IntegrationLatency	从 API Gateway 将请求中继到后端到其从后端收到响应所经过的时间。 Unit: Millisecond
Latency	从 API Gateway 从客户端收到请求到其将响应返回给客户端所经过的时间。延迟包括集成延迟和其他 API Gateway 开销。 Unit: Millisecond

指标的维度

您可以使用下表中的维度筛选 API Gateway 指标。

> **Note**
>
> API Gateway 先从 ApiName 维度中删除非 ASCII 字符，然后再将指标发送到 CloudWatch。如果 APIName 不包含任何 ASCII 字符，则 API ID 将用作 ApiName。

维度	说明
ApiName	针对具有指定 API 名称的 REST API 筛选 API Gateway 指标。
ApiName, Method, Resource, Stage	针对具有指定 API 名称、阶段、资源和方法的 API 方法筛选 API Gateway 指标。 除非您明确启用了详细的 CloudWatch 指标，否则 API Gateway 不会发送这些指标。在控制台中，选择一个阶段，然后对于日志和跟踪，选择编辑。选择详细指标，然后选择保存更改。或者，您也可以调用 update-stage AWS CLI 命令，以将 `metricsEnabled` 属性更新为 `true`。 启用这些指标会对您的账户额外计费。有关定价信息，请参阅 Amazon CloudWatch 定价。
ApiName, Stage	针对具有指定 API 名称和阶段的 API 方法筛选 API Gateway 指标。

使用 API Gateway 中的 API 控制面板查看 CloudWatch 指标

您可以使用 API Gateway 控制台中的 API 控制面板在 API Gateway 中显示已部署 API 的 CloudWatch 指标。它们显示为一段时间内 API 活动的总结。

主题

- 先决条件

- 在控制面板中检查 API 活动

先决条件

1. 您必须已在 API Gateway 中创建 API。按照中的说明进行操作在 API Gateway 中开发 REST API
2. 您必须至少部署一次 API。按照中的说明进行操作在 Amazon API Gateway 中部署 REST API

在控制面板中检查 API 活动

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择一个 API。
3. 在主导航窗格中，选择控制面板。
4. 对于阶段，选择所需的阶段。
5. 选择日期范围以指定日期范围。
6. 如果需要请刷新，并查看标题分别为 API 调用、延迟、集成延迟、延迟、4xx 错误和 5xx 错误的独立图表中显示的各个指标。

> **Tip**
>
> 要检查方法级别的 CloudWatch 指标，请确保您已在方法级别启用 CloudWatch Logs。有关如何设置方法级别日志记录的更多信息，请参阅使用 API Gateway 控制台更新阶段设置。

在 CloudWatch 控制台中查看 API Gateway 指标

指标的分组首先依据服务命名空间，然后依据每个命名空间内的各种维度组合。要查看 API 的指标级指标，请开启详细指标。有关更多信息，请参阅 the section called “更新阶段设置”。

使用 CloudWatch 控制台查看 API Gateway 指标

1. 通过以下网址打开 CloudWatch 控制台：https://console.aws.amazon.com/cloudwatch/。
2. 如果需要，更改 AWS 区域。从导航栏中，选择 AWS 资源所在的区域。
3. 在导航窗格中，选择指标。
4. 在全部指标选项卡上，选择 API Gateway。

5. 要按阶段查看指标，请选择按阶段面板。然后，选择您的 API 和指标名称。
6. 要按特定 API 查看指标，请选择按 Api 名称面板。然后，选择您的 API 和指标名称。

使用AWS CLI 查看指标

1. 在命令提示符处使用以下命令列出指标：

   ```
   aws cloudwatch list-metrics --namespace "AWS/ApiGateway"
   ```

   创建指标后，请等待最多 15 分钟让指标显示出来。要更快地查看指标统计信息，请使用 get-metric-data 或 get-metric-statistics。
2. 要按 5 分钟间隔查看一段时间的特定统计数据（例如，`Average`），请调用以下命令：

   ```
   aws cloudwatch get-metric-statistics --namespace AWS/ApiGateway --metric-name Count
    --start-time 2011-10-03T23:00:00Z --end-time 2017-10-05T23:00:00Z --period 300 --
   statistics Average
   ```

在 CloudWatch 控制台中查看 API Gateway 日志事件

先决条件

1. 您必须已在 API Gateway 中创建 API。按照中的说明进行操作在 API Gateway 中开发 REST API
2. 必须至少部署和调用一次 API。按照在 Amazon API Gateway 中部署 REST API和在 Amazon API Gateway 中调用 REST API中的说明操作。
3. 您必须为某个阶段启用了 CloudWatch 日志。按照在 API Gateway 中为 REST API 设置 CloudWatch 日志记录中的说明进行操作。

使用 CloudWatch 控制台查看记录的 API 请求和响应

1. 通过以下网址打开 CloudWatch 控制台：https://console.aws.amazon.com/cloudwatch/。
2. 如果需要，更改 AWS 区域。从导航栏中，选择 AWS 资源所在的区域。有关更多信息，请参阅区域和端点。
3. 在导航窗格中，依次选择日志和日志组。
4. 在日志组表中，选择 API-Gateway-Execution-Logs_{rest-api-id}/{stage-name} 名称的日志组。
5. 在日志流表下，选择日志流。您可以使用时间戳来帮助查找感兴趣的日志流。

6. 选择文本可查看原始文本，选择行可逐行查看事件。

> ⚠ Important
>
> CloudWatch 允许您删除日志组或流。请勿手动删除 API Gateway API 日志组或流；让 API Gateway 管理这些资源。手动删除日志组或流可能会导致未记录 API 请求和响应。如果出现这种情况，您可以删除 API 的整个日志组并重新部署 API。这是因为 API Gateway 会在部署时为某个 API 阶段创建日志组或日志流。

中的监控工具AWS

AWS 为您提供了各种可用于监控 API Gateway 的工具。您可以配置其中的一些工具来为您自动执行监控任务，但其他工具需要手动干预。建议您尽可能实现监控任务自动化。

中的自动监控工具AWS

您可以使用以下自动化监控工具来监控 API Gateway，并在出现错误时进行报告：

- Amazon CloudWatch 警报 – 按您指定的时间段观察单个指标，并根据相对于给定阈值的指标值在若干时间段内执行一项或多项操作。具体操作是：通知已发送到 Amazon Simple Notification Service（Amazon SNS）主题或 Amazon EC2 Auto Scaling 策略。CloudWatch 告警不调用操作，因为这些操作处于特定状态；状态必须改变并保持指定时间。有关更多信息，请参阅使用 Amazon CloudWatch 指标监控 REST API 执行。
- Amazon CloudWatch Logs – 监控、存储和访问来自 AWS CloudTrail 或其他来源的日志文件。有关更多信息，请参阅《Amazon CloudWatch 用户指南》中的 What is CloudWatch Logs?
- Amazon EventBridge（以前称为 CloudWatch Events）– 匹配事件并将事件传送到一个或多个目标函数或流，来进行更改、捕获状态信息和采取纠正措施。有关更多信息，请参阅《EventBridge 用户指南》中的 What Is Amazon EventBridge?
- AWS CloudTrail 日志监控——在账户间共享日志文件，通过将 CloudTrail 日志文件发送到 CloudWatch Logs 来进行实时监控，用 Java 编写日志处理应用程序，验证 CloudTrail 提供的日志文件未发生更改。有关更多信息，请参见《AWS CloudTrail 用户指南》的使用 CloudTrail 日志文件。

手动监控工具

监控 API Gateway 的另一个重要环节是手动监控 CloudWatch 警报未涵盖的那些项。API Gateway、CloudWatch 和其他AWS控制台控制面板提供您的AWS环境状态的概览视图。建议您还要查看有关 API 执行的日志文件。

- API Gateway 控制面板显示指定时间段内给定 API 阶段的以下统计数据：
  - API 调用
  - 缓存命中（仅当启用 API 缓存时）。
  - 缓存未命中（仅当启用 API 缓存时）。
  - 延迟
  - 集成延迟
  - 4XX 错误
  - 5XX 错误
- CloudWatch 主页显示：
  - 当前警报和状态
  - 告警和资源图表
  - 服务运行状况

  此外，还可以使用 CloudWatch 执行以下操作：

  - 创建自定义控制面板以监控您关心的服务
  - 绘制指标数据图，以排除问题并弄清楚趋势
  - 搜索并浏览您所有的 AWS 资源指标
  - 创建和编辑警报以接收有关问题的通知

创建 CloudWatch 警报以监控 API Gateway 指标

您可以创建在警报改变状态时发送 Amazon SNS 消息的 CloudWatch 警报。警报会每隔一段时间（由您指定）监控一个指标，并根据相对于给定阈值的指标值每隔若干个时间段执行一项或多项操作。操作是一个发送到 Amazon SNS 主题或自动扩缩策略的通知。警报只会调用操作进行持续的状态变更。CloudWatch 告警不调用操作，因为这些操作处于特定状态；状态必须改变并保持指定时间。

在 API Gateway 中为 REST API 设置 CloudWatch 日志记录

为了帮助调试与请求执行或客户端对 API 的访问的相关问题，您可以启用 Amazon CloudWatch Logs 以记录 API 调用。有关 CloudWatch 的更多信息，请参阅the section called “CloudWatch 指标”。

用于 API Gateway 的 CloudWatch 日志格式

在 CloudWatch 中有两种类型的 API 日志记录：执行日志记录和访问日志记录。在执行日志记录中，API Gateway 管理 CloudWatch Logs。该过程包括创建日志组和日志流，以及向日志流报告任意调用方的请求和响应。

记录的数据包括错误或执行跟踪（例如，请求或响应的参数值或负载）、Lambda 授权方（以前称为自定义授权方）使用的数据、是否需要 API 密钥、是否启用了使用计划以及其它信息。API Gateway 会从记录的数据中删除授权标头、API 密钥值和类似的敏感请求参数。

部署 API 时，API Gateway 创建日志组和日志组下的日志流。日志组以 `API-Gateway-Execution-Logs_{rest-api-id}/{stage_name}` 格式命名。在每个日志组内，日志进一步划分到日志流中，后者在报告记录的数据时按照上次事件时间排序。

在访问日志记录中，作为 API 开发人员，您想要记录谁访问了您的 API 以及调用方访问 API 的方式。您可以创建自己的日志组，或者选择可由 API Gateway 管理的现有日志组。要指定访问详细信息，可以选择 `$context` 变量、日志格式和日志组目标。

访问日志格式必须至少包括 `$context.requestId` 或 `$context.extendedRequestId`。作为最佳实践，请在日志格式中包含 `$context.requestId` 和 `$context.extendedRequestId`。

`$context.requestId`

这会将值记录在 `x-amzn-RequestId` 标头中。客户端可以用通用唯一标识符 (UUID) 格式的值覆盖 `x-amzn-RequestId` 标头中的值。API Gateway 返回 `x-amzn-RequestId` 响应标头中的此请求 ID。API Gateway 将不采用 UUID 格式的被覆盖的请求 ID 替换为访问日志中的 *`UUID`*`_REPLACED_INVALID_REQUEST_ID`。

`$context.extendedRequestId`

extendedRequestID 是 API Gateway 生成的唯一 ID。API Gateway 返回 `x-amz-apigw-id` 响应标头中的此请求 ID。API 调用者无法提供或覆盖此请求 ID。您可能需要向 AWS Support 提供此值，来协助排查 API 的问题。有关更多信息，请参阅 the section called “适用于数据模型、授权方、映射模板和 CloudWatch 访问日志记录的 `$context` 变量”。

> **Note**
>
> 仅支持 `$context` 变量。

选择您的分析后端也采用的日志格式，例如常用日志格式 (CLF)、JSON、XML 或 CSV。然后，您可以将访问日志直接输送到其中，以计算和呈现您的指标。要定义日志格式，请在阶段的 accessLogSettings/destinationArn 属性上设置日志组 ARN。您可以在 CloudWatch 控制台中获取日志组 ARN。要定义访问日志格式，请在阶段的 accessLogSetting/format 属性上设置选定格式。

一些常用访问日志格式的示例在 API Gateway 控制台中显示，下面列出了这些格式。

- CLF（常用日志格式）：

```
$context.identity.sourceIp $context.identity.caller $context.identity.user
 [$context.requestTime]"$context.httpMethod $context.resourcePath
 $context.protocol" $context.status $context.responseLength $context.requestId
 $context.extendedRequestId
```

- JSON:

```
{ "requestId":"$context.requestId",
 "extendedRequestId":"$context.extendedRequestId","ip": "$context.identity.sourceIp",
 "caller":"$context.identity.caller", "user":"$context.identity.user",
 "requestTime":"$context.requestTime", "httpMethod":"$context.httpMethod",
 "resourcePath":"$context.resourcePath", "status":"$context.status",
 "protocol":"$context.protocol", "responseLength":"$context.responseLength" }
```

- XML:

```
<request id="$context.requestId"> <extendedRequestId>$context.extendedRequestId</
extendedRequestId> <ip>$context.identity.sourceIp</ip> <caller>
$context.identity.caller</caller> <user>$context.identity.user</user> <requestTime>
$context.requestTime</requestTime> <httpMethod>$context.httpMethod</httpMethod>
 <resourcePath>$context.resourcePath</resourcePath> <status>$context.status</status>
 <protocol>$context.protocol</protocol> <responseLength>$context.responseLength</
responseLength> </request>
```

- CSV（逗号分隔值）：

```
$context.identity.sourceIp,$context.identity.caller,$context.identity.user,
$context.requestTime,$context.httpMethod,$context.resourcePath,$context.protocol,
$context.status,$context.responseLength,$context.requestId,$context.extendedRequestId
```

CloudWatch 日志记录的权限

要启用 CloudWatch Logs，您必须向 API Gateway 授予权限，才能针对您的账户读取日志和将日志写入到 CloudWatch。`AmazonAPIGatewayPushToCloudWatchLogs`托管策略（ARN 为 `arn:aws:iam::aws:policy/service-role/AmazonAPIGatewayPushToCloudWatchLogs`）具有全部必需的权限：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:DescribeLogGroups",
                "logs:DescribeLogStreams",
                "logs:PutLogEvents",
                "logs:GetLogEvents",
                "logs:FilterLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

> ⓘ Note
>
> API Gateway 将调用 AWS Security Token Service 以恢复 IAM 角色，因此请确保已为区域启用 AWS STS。有关更多信息，请参阅在AWS区域中管理AWS STS。

要将这些权限授予您的账户，请使用 `apigateway.amazonaws.com` 作为其可信实体创建一个 IAM 角色，将之前的策略附加到此 IAM 角色，然后在您账户的 cloudWatchRoleArn 属性上设置 IAM 角色

ARN。您必须为要在其中启用 CloudWatch Logs 的每个AWS区域单独设置 cloudWatchRoleArn 属性。

如果您在设置 IAM 角色 ARN 时收到错误，请检查您的 AWS Security Token Service 账户设置，以确保在您所使用的区域中启用了 AWS STS。有关启用 AWS STS 的更多信息，请参阅 IAM 用户指南中的在AWS区域中管理AWS STS。

使用 API Gateway 控制台设置 CloudWatch API 日志记录

要设置 CloudWatch API 日志记录，您必须已经将 API 部署到某个阶段。您还必须已经为账户配置了合适的 CloudWatch Logs 角色 ARN。

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 在主导航窗格上选择设置，然后在日志记录下选择编辑。
3. 对于 CloudWatch 日志角色 ARN，输入具有相应权限的 IAM 角色的 ARN。对于每个使用 API Gateway 创建 API 的 AWS 账户，您都需要执行一次此操作。
4. 在主导航窗格中，选择 API，然后执行以下操作之一：

    a. 选择现有 API，然后选择一个阶段。

    b. 创建 API，然后将其部署到阶段。
5. 在主导航窗格中，选择阶段。
6. 在日志和跟踪部分中，选择编辑。
7. 启用执行日志记录：

    a. 从 CloudWatch Logs 下拉菜单中选择日志记录级别。日志记录级别如下所示：

        - 关闭 - 不对此阶段开启日志记录。
        - 仅限错误 - 仅对错误启用日志记录。
        - 错误和信息日志 - 对所有事件启用日志记录。
        - 完整的请求和响应日志 - 对所有事件启用详细日志记录。这对于排除 API 故障非常有用，但可能会导致记录敏感数据。

> **Note**
>
> 我们建议不要为生产 API 使用完整请求和响应日志。

   b. 如果需要，请选择详细指标以开启详细的 CloudWatch 指标。

   有关 CloudWatch 指标的更多信息，请参阅the section called “CloudWatch 指标”。

8. 启用访问日志记录：

   a. 开启自定义访问日志记录。
   b. 对于访问日志目标 ARN，输入日志组的 ARN。ARN 格式为 `arn:aws:logs:{region}:{account-id}:log-group:log-group-name`。
   c. 对于日志格式，输入日志格式。您可以选择 CLF、JSON、XML 或 CSV。要了解有关示例日志格式的更多信息，请参阅the section called “用于 API Gateway 的 CloudWatch 日志格式”。

9. 选择 Save changes（保存更改）。

> **Note**
>
> 您可以分别启用执行日志记录和访问日志记录，这两者相互独立。

API Gateway 现已准备好记录对您 API 的请求。在更新阶段设置、日志或阶段变量时，您无需重新部署 API。

使用 AWS CloudFormation 设置 CloudWatch API 日志记录

使用以下示例 AWS CloudFormation 模板创建 Amazon CloudWatch Logs 日志组并配置阶段的执行和访问日志记录。要启用 CloudWatch Logs，您必须向 API Gateway 授予权限，才能针对您的账户读取日志和将日志写入到 CloudWatch。要了解更多信息，请参阅《AWS CloudFormation 用户指南》中的将账户与 IAM 角色关联。

```
TestStage:
  Type: AWS::ApiGateway::Stage
  Properties:
    StageName: test
    RestApiId: !Ref MyAPI
    DeploymentId: !Ref Deployment
    Description: "test stage description"
    MethodSettings:
      - ResourcePath: "/*"
        HttpMethod: "*"
```

```
          LoggingLevel: INFO
      AccessLogSetting:
        DestinationArn: !GetAtt MyLogGroup.Arn
        Format: $context.extendedRequestId $context.identity.sourceIp
$context.identity.caller $context.identity.user [$context.requestTime]
"$context.httpMethod $context.resourcePath $context.protocol" $context.status
$context.responseLength $context.requestId
 MyLogGroup:
   Type: AWS::Logs::LogGroup
   Properties:
     LogGroupName: !Join
       - '-'
       - - !Ref MyAPI
         - access-logs
```

将 API 调用记录到 Amazon Data Firehose

为有助于调试与客户端对您的 API 的访问相关的问题，您可以将 API 调用记录到 Amazon Data Firehose。有关 Firehose 的更多信息，请参阅 What Is Amazon Data Firehose?

对于访问日志记录，您只能启用 CloudWatch 或 Firehose - 而不能同时启用两者。但是，您可以启用 CloudWatch 来实施执行日志记录，并启用 Firehose 来实施访问日志记录。

主题

- 适用于 API Gateway 的 Firehose 日志格式
- Firehose 日志记录的权限
- 使用 API Gateway 控制台设置 Firehose 访问日志记录

适用于 API Gateway 的 Firehose 日志格式

Firehose 日志记录与 CloudWatch 日志记录使用相同的格式。

Firehose 日志记录的权限

当在某个阶段启用 Firehose 访问日志记录时，如果服务相关角色尚不存在，则 API Gateway 会在您的账户中创建此角色。该角色被命名为 `AWSServiceRoleForAPIGateway`，并且已将 `APIGatewayServiceRolePolicy` 托管策略附加到该角色。有关服务相关角色的更多信息，请参阅使用服务相关角色。

> **Note**
>
> Firehose 流的名称必须为 `amazon-apigateway-{your-stream-name}`。

使用 API Gateway 控制台设置 Firehose 访问日志记录

要设置 API 日志记录，您必须已经将 API 部署到某个阶段。您还必须创建 Firehose 流。

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 请执行以下操作之一：
    a. 选择现有 API，然后选择一个阶段。
    b. 创建 API 并将其部署到阶段。
3. 在主导航窗格中，选择阶段。
4. 在日志和跟踪部分中，选择编辑。
5. 要启用 Firehose 流的访问日志记录，请执行以下操作：
    a. 开启自定义访问日志记录。
    b. 对于访问日志目标 ARN，输入 Firehose 流的 ARN。ARN 格式为 `arn:aws:firehose:{region}:{account-id}:deliverystream/amazon-apigateway-{your-stream-name}`。

    > **Note**
    >
    > Firehose 流的名称必须为 `amazon-apigateway-{your-stream-name}`。

    c. 对于日志格式，输入日志格式。您可以选择 CLF、JSON、XML 或 CSV。要了解有关示例日志格式的更多信息，请参阅the section called “用于 API Gateway 的 CloudWatch 日志格式”。
6. 选择 Save changes（保存更改）。

API Gateway 现已准备好将对 API 的请求记录到 Firehose。在更新阶段设置、日志或阶段变量时，您无需重新部署 API。

使用 X-Ray 跟踪用户对 REST API 的请求

当用户请求通过 Amazon API Gateway REST API 传输到底层服务时，您可以使用 AWS X-Ray 对用户请求进行跟踪和分析。API Gateway 支持对所有 API Gateway REST API 终端节点类型进行 X-Ray 跟踪：区域、边缘优化和私有。您可以在提供 X-Ray 的所有AWS区域中将 X-Ray 与 Amazon API Gateway 结合使用。

X-Ray 为您提供了整个请求的端到端视图，因此您可以分析 API 及其后端服务中的延迟。您可以使用 X-Ray 服务地图，以查看整个请求的延迟及集成了 X-Ray 的下游服务的延迟。您还可以配置采样规则，以告知 X-Ray 根据您指定的标准以哪种采样率记录哪些请求。

如果您从正在跟踪的服务中调用 API Gateway，则 API Gateway 会通过跟踪，即使 X-Ray 跟踪在 API 上未启用。

您可以使用 API Gateway 控制台或使用 API Gateway API 或 CLI 为 API 阶段启用 X-Ray。

主题

- 使用 API Gateway REST API 设置 AWS X-Ray
- 通过 API Gateway 使用 AWS X-Ray 服务地图和跟踪视图
- 为 API Gateway API 配置 AWS X-Ray 采样规则
- 了解 Amazon API Gateway API 的 AWS X-Ray 跟踪

使用 API Gateway REST API 设置 AWS X-Ray

在本节中，您可以找到有关如何使用 API Gateway REST API 设置 AWS X-Ray 的详细信息。

主题

- API Gateway 的 X-Ray 跟踪模式
- X-Ray 跟踪的权限
- 在 API Gateway 控制台中启用 X-Ray 跟踪
- 使用 API Gateway CLI 启用 AWS X-Ray 跟踪

API Gateway 的 X-Ray 跟踪模式

通过跟踪 ID 来跟踪请求在您的应用程序中传输的路径。跟踪会收集单个请求（通常是 HTTP `GET` 或 `POST` 请求）生成的所有分段。

API Gateway API 的跟踪有两种模式：

- 被动：如果您尚未在 API 阶段上启用 X-Ray 跟踪，则此模式为默认设置。此方法意味着，只有在上游服务上启用 X-Ray 后才会跟踪 API Gateway API。
- 主动：如果 API Gateway API 阶段具有此设置， API Gateway 会自动根据 X-Ray 指定的采样算法对 API 调用请求进行采样。

  在一个阶段上启用主动跟踪时，如果服务相关角色不存在，API Gateway 会在您的账户中创建该角色。该角色被命名为 `AWSServiceRoleForAPIGateway`，且将 `APIGatewayServiceRolePolicy` 托管策略附加于此。有关服务相关角色的更多信息，请参阅使用服务相关角色。

  Note

  X-Ray 应用采样算法确保跟踪有效，同时为 API 所接收的请求提供代表性样本。默认的采样算法是每秒钟 1 个请求，超过此限制的请求采样 5%。

您可以使用 API Gateway 管理控制台、API Gateway CLI 或AWS开发工具包更改 API 的跟踪模式。

X-Ray 跟踪的权限

当您在一个阶段上启用 X-Ray 跟踪时，如果服务相关角色不存在，则 API Gateway 会在您的账户中创建该角色。该角色被命名为 `AWSServiceRoleForAPIGateway`，且将 `APIGatewayServiceRolePolicy` 托管策略附加于此。有关服务相关角色的更多信息，请参阅使用服务相关角色。

在 API Gateway 控制台中启用 X-Ray 跟踪

您可以使用 Amazon API Gateway 控制台在 API 阶段上启用活动的跟踪。

这些步骤假设您已将 API 部署到阶段。

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择您的 API，然后在主导航窗格中选择阶段。
3. 在阶段窗格中，选择阶段。
4. 在日志和跟踪部分中，选择编辑。
5. 要启用活动的 X-Ray 跟踪，请选择 X-Ray 跟踪以开启 X-Ray 跟踪。
6. 选择 Save changes（保存更改）。

一旦您已为 API 阶段启用了 X-Ray，则可以使用 X-Ray 管理控制台查看跟踪和服务地图。

使用 API Gateway CLI 启用 AWS X-Ray 跟踪

创建阶段时，要针对 API 阶段使用 AWS CLI 来启用活动的 X-Ray 跟踪，请调用 `create-stage` 命令，如以下示例所示：

```
aws apigateway create-stage \
    --rest-api-id {rest-api-id} \
    --stage-name {stage-name} \
    --deployment-id {deployment-id} \
    --region {region} \
    --tracing-enabled=true
```

下面是成功调用的示例输出：

```
{
    "tracingEnabled": true,
    "stageName": {stage-name},
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": {deployment-id},
    "lastUpdatedDate": 1533849811,
    "createdDate": 1533849811,
    "methodSettings": {}
}
```

创建阶段时，要针对 API 阶段使用 AWS CLI 来禁用活动的 X-Ray 跟踪，请调用 `create-stage` 命令，如以下示例所示：

```
aws apigateway create-stage \
    --rest-api-id {rest-api-id} \
    --stage-name {stage-name} \
    --deployment-id {deployment-id} \
    --region {region} \
    --tracing-enabled=false
```

下面是成功调用的示例输出：

```
{
    "tracingEnabled": false,
```

```
    "stageName": {stage-name},
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": {deployment-id},
    "lastUpdatedDate": 1533849811,
    "createdDate": 1533849811,
    "methodSettings": {}
}
```

针对已部署的 API，要使用 AWS CLI 来启用活动的 X-Ray 跟踪，请调用 `update-stage` 命令，如以下示例所示：

```
aws apigateway update-stage \
    --rest-api-id {rest-api-id} \
    --stage-name {stage-name} \
    --patch-operations op=replace,path=/tracingEnabled,value=true
```

针对已部署的 API，要使用 AWS CLI 来禁用活动的 X-Ray 跟踪，请调用 `update-stage` 命令，如以下示例所示：

```
aws apigateway update-stage \
    --rest-api-id {rest-api-id} \
    --stage-name {stage-name} \
    --region {region} \
    --patch-operations op=replace,path=/tracingEnabled,value=false
```

下面是成功调用的示例输出：

```
{
    "tracingEnabled": false,
    "stageName": {stage-name},
    "cacheClusterEnabled": false,
    "cacheClusterStatus": "NOT_AVAILABLE",
    "deploymentId": {deployment-id},
    "lastUpdatedDate": 1533850033,
    "createdDate": 1533849811,
    "methodSettings": {}
}
```

一旦您已为 API 阶段启用了 X-Ray，请使用 X-Ray CLI 检索跟踪信息。有关更多信息，请参阅将 AWS X-Ray API 与AWS CLI 一起使用。

通过 API Gateway 使用 AWS X-Ray 服务地图和跟踪视图

在本节中，您可以找到有关如何借助 API Gateway 使用 AWS X-Ray 服务映射和跟踪视图的详细信息。

有关服务地图和跟踪视图以及如何对其进行解析的详细信息，请参阅 AWS X-Ray 控制台。

主题

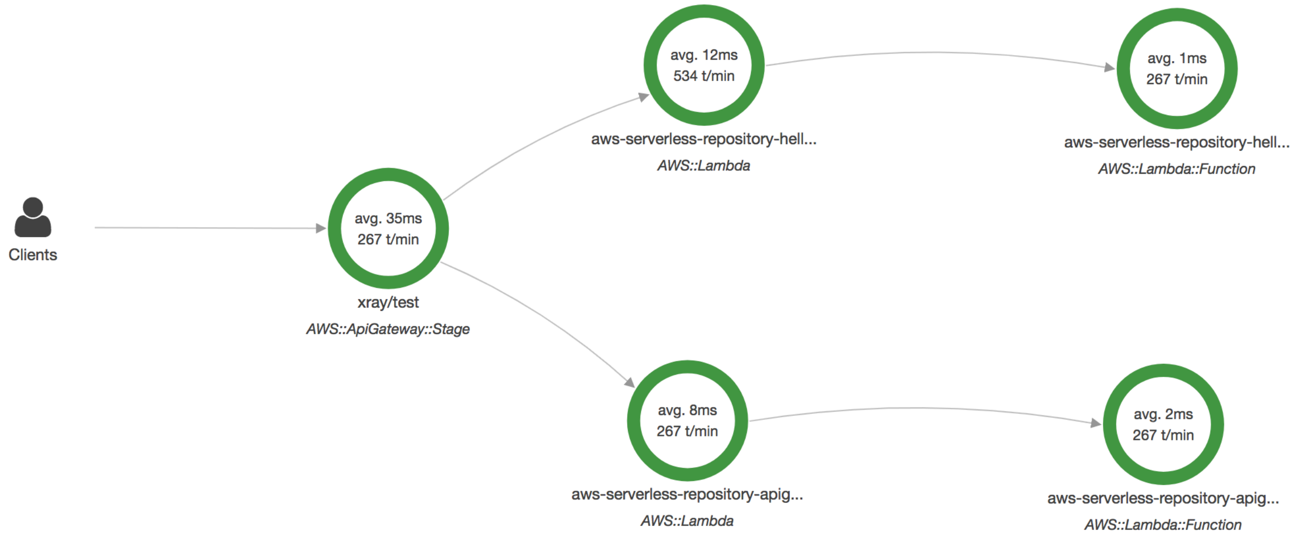
- X-Ray 服务地图示例
- X-Ray 跟踪视图示例

X-Ray 服务地图示例

AWS X-Ray 服务地图显示有关您的 API 及其所有下游服务的信息。在 API Gateway 中为 API 阶段启用 X-Ray 时，在服务地图中，您将看到一个包含有关 API Gateway 服务耗费的全部时间的节点。您可以获取有关响应状态和直方图所选时间范围的 API 响应时间。对于与AWS服务（如 AWS Lambda 和 Amazon DynamoDB）集成的 API，您将看到更多节点提供与这些服务相关的性能指标。每个 API 阶段将有一个服务地图。

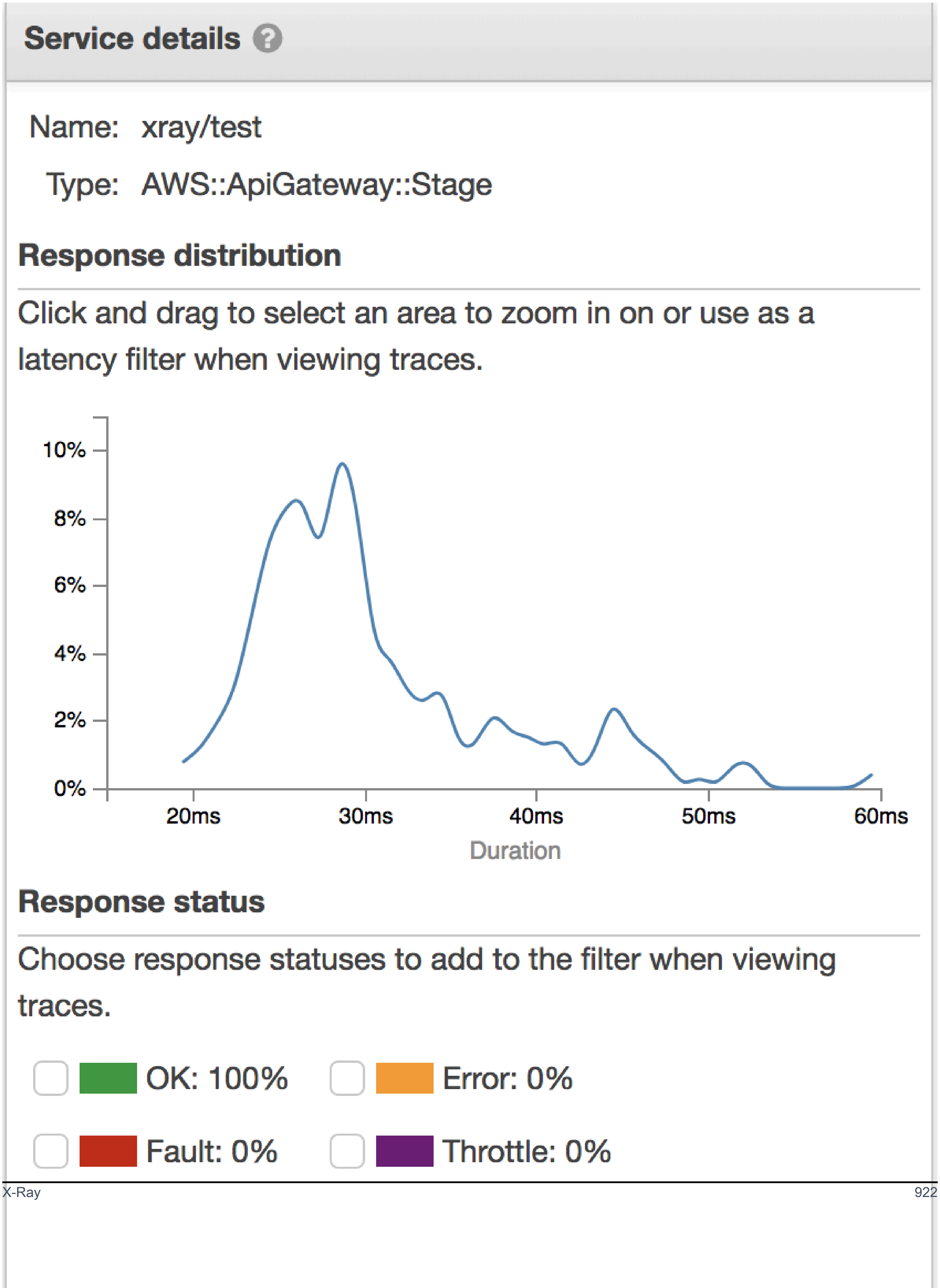
以下示例显示了名为 `test` API 的 `xray` 阶段的一个服务地图。此 API 与 Lambda 授权方函数和 Lambda 后端函数进行了 Lambda 集成。节点表示 API Gateway 服务、Lambda 服务和两个 Lambda 函数。

有关服务地图结构的详细说明，请参阅查看服务地图。



您可以在服务地图中进行放大，查看您的 API 阶段的跟踪视图。跟踪将显示与您的 API 相关的深度信息，以分段和子分段的方式显示。例如，上面所示的服务地图跟踪包括 Lambda 服务和 Lambda 函数的分段。有关更多信息，请参阅AWS Lambda 和AWS X-Ray。

如果您在 X-Ray 服务地图上选择一个节点或边缘，X-Ray 控制台会显示一个延迟分布直方图。您可以使用延迟直方图查看一项服务所需的时长以完成其请求。下面是 API Gateway 阶段的直方图，在之前的服务地图中名为 `xray/test`。有关延迟分配直方图的详细说明，请参阅在 AWS X-Ray 控制台中使用延迟直方图。



Service details

Name: xray/test

Type: AWS::ApiGateway::Stage

Response distribution

Click and drag to select an area to zoom in on or use as a latency filter when viewing traces.

Response status

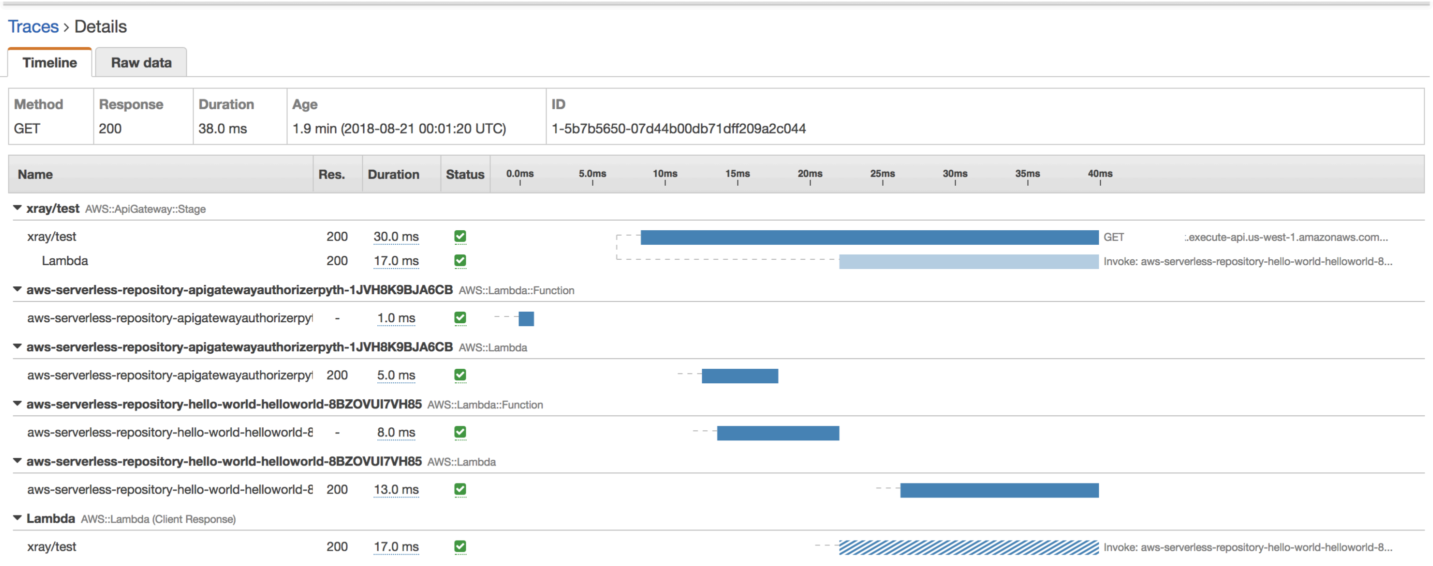
Choose response statuses to add to the filter when viewing traces.

☐ 🟩 OK: 100%
☐ 🟧 Error: 0%
☐ 🟥 Fault: 0%
☐ 🟪 Throttle: 0%

X-Ray 跟踪视图示例

下图显示了为上述的示例 API 生成的跟踪视图，其中具有 Lambda 后端函数和 Lambda 授权方函数。成功的 API 方法请求的响应代码显示为 200。

有关跟踪视图的详细说明，请参阅查看跟踪。



Traces › Details

Method	Response	Duration	Age	ID
GET	200	38.0 ms	1.9 min (2018-08-21 00:01:20 UTC)	1-5b7b5650-07d44b00db71dff209a2c044

Name	Res.	Duration	Status
xray/test AWS::ApiGateway::Stage			
xray/test	200	30.0 ms	✓
Lambda	200	17.0 ms	✓
aws-serverless-repository-apigatewayauthorizerpyth-1JVH8K9BJA6CB AWS::Lambda::Function			
aws-serverless-repository-apigatewayauthorizerpyt	-	1.0 ms	✓
aws-serverless-repository-apigatewayauthorizerpyth-1JVH8K9BJA6CB AWS::Lambda			
aws-serverless-repository-apigatewayauthorizerpyt	200	5.0 ms	✓
aws-serverless-repository-hello-world-helloworld-8BZOVUI7VH85 AWS::Lambda::Function			
aws-serverless-repository-hello-world-helloworld-8	-	8.0 ms	✓
aws-serverless-repository-hello-world-helloworld-8BZOVUI7VH85 AWS::Lambda			
aws-serverless-repository-hello-world-helloworld-8	200	13.0 ms	✓
Lambda AWS::Lambda (Client Response)			
xray/test	200	17.0 ms	✓

为 API Gateway API 配置 AWS X-Ray 采样规则

您可以使用 AWS X-Ray 控制台或开发工具包为您的 Amazon API Gateway API 配置采样规则。采样规则指定了 X-Ray 应为您的 API 记录哪些请求。通过自定义采样规则，您可以控制您记录的数据量，并即刻修改采样行为，而不修改或重新部署您的代码。

指定您的 X-Ray 采样规则之前，请阅读《X-Ray 开发人员指南》中的以下主题：

- 在 AWS X-Ray 控制台配置采样规则
- 通过 X-Ray API 使用采样规则

主题

- API Gateway API 的 X-Ray 采样规则选项值
- X-Ray 采样规则示例

API Gateway API 的 X-Ray 采样规则选项值

以下 X-Ray 采样选项与 API Gateway 相关。字符串值可以使用通配符来匹配单个字符 (?) 或零或多个字符 (*)。有关更多详细信息，包括如何使用容器和速率设置的详细说明，请参阅 AWS X-Ray 控制台中的配置采样规则 。

- 规则名称（字符串）— 一个唯一的规则名称。
- 优先级（1 和 9999 之间的整数）— 采样规则的优先级。服务按优先级的上升顺序评估规则，并与匹配的第一条规则进行抽样决策。
- 容器（非负整数）— 在应用固定速率之前，每秒要分析的匹配请求的固定数量。该容器不由服务直接使用，但适用于所有使用该规则的服务。
- 速率（0 到 100 之间的数字）— 容器耗尽后，要分析的匹配请求的百分比。
- 服务名称（字符串）— API 阶段名称，形式为 ***{api-name}*/*{stage-name}***。例如，如果您要将 PetStore 示例 API 部署到名为 `test` 的阶段，则要在您的采样规则中指定的 Service name (服务名称) 值应为 **`pets/test`**。
- 服务类型（字符串）— 对于 API Gateway API，可以指定 **`AWS::ApiGateway::Stage`** 或 **`AWS::ApiGateway::*`**。
- 主机（字符串）— HTTP 主机标头中的主机名。请将 **`*`** 设置为匹配所有的主机名。或者，您可以指定完整的或部分主机名进行匹配，例如 **`api.example.com`** 或 **`*.example.com`**。
- 资源 ARN（字符串）– API 阶段的 ARN，例如 **`arn:aws:apigateway:`*`region`*`::/restapis/`*`api-id`*`/stages/`*`stage-name`***。

  该阶段名可以从控制台或 API Gateway CLI 或 API 获取。有关 ARN 格式的更多信息，请参阅 Amazon Web Services 一般参考。
- HTTP 方法（字符串）— 要采样的方法，例如 **`GET`**。
- URL path (URL 路径)（字符串） — 请求的 URL 路径。
- （可选）属性（密钥和值）— 来自原始 HTTP 请求的标头，例如 **`Connection`**、**`Content-Length`** 或 **`Content-Type`**。每个属性值的长度最多为 32 个字符。

X-Ray 采样规则示例

采样规则示例 #1

此规则在 `GET` 阶段针对 `testxray` API 所有的 `test` 请求采样。

- 规则名称 — **`test-sampling`**

- 优先级 — **17**
- 容器大小 — **10**
- 固定速率 — **10**
- 服务名称 — **testxray/test**
- 服务类型 — **AWS::ApiGateway::Stage**
- HTTP 方法 — **GET**
- 资源 ARN — *****
- 主机 — *****

采样规则示例 #2

此规则在 `testxray` 阶段针对 `prod` API 的所有请求采样。

- 规则名称 — **prod-sampling**
- 优先级 — **478**
- 容器大小 — **1**
- 固定速率 — **60**
- 服务名称 — **testxray/prod**
- 服务类型 — **AWS::ApiGateway::Stage**
- HTTP 方法 — *****
- 资源 ARN — *****
- 主机 — *****
- 属性 — **{}**

了解 Amazon API Gateway API 的 AWS X-Ray 跟踪

本部分讨论 Amazon API Gateway API 的 AWS X-Ray 跟踪分段、子段和其他跟踪字段。

阅读本节之前，请查看《X-Ray 开发人员指南》中的以下主题中：

- AWS X-Ray 控制台
- AWS X-Ray 分段文档
- X-Ray 概念

主题

- 针对 API Gateway API 的跟踪对象示例
- 了解跟踪

针对 API Gateway API 的跟踪对象示例

本节讨论您可能在 API Gateway API 跟踪中看到的一些对象。

注释

注解可在分段和子分段中出现。它们在采样规则中用作过滤表达式，以过滤跟踪。有关更多信息，请参阅在 AWS X-Ray 控制台中配置采样规则。

下面是 `annotations` 对象的一个示例，其中，API 阶段由 API ID 和 API 阶段名称标识：

```
"annotations": {
    "aws:api_id": "a1b2c3d4e5",
    "aws:api_stage": "dev"
}
```

AWS 资源数据

`aws` 对象仅存在于分段中。以下是与默认的取样规则匹配的 `aws` 对象的一个示例。有关采样规则的深入说明，请参阅 AWS X-Ray 控制台中配置采样规则。

```
"aws": {
    "xray": {
        "sampling_rule_name": "Default"
    },
    "api_gateway": {
        "account_id": "123412341234",
        "rest_api_id": "a1b2c3d4e5",
        "stage": "dev",
        "request_id": "a1b2c3d4-a1b2-a1b2-a1b2-a1b2c3d4e5f6"
    }
}
```

了解跟踪

以下是 API Gateway 阶段的跟踪分段。有关组成跟踪分段的字段的详细说明，请参阅“AWS X-Ray 开发人员指南”中的 AWS X-Ray 分段文档。

```
        {
            "Document": {
                "id": "a1b2c3d4a1b2c3d4",
                "name": "testxray/dev",
                "start_time": 1533928226.229,
                "end_time": 1533928226.614,
                "metadata": {
                    "default": {
                        "extended_request_id": "abcde12345abcde=",
                        "request_id": "a1b2c3d4-a1b2-a1b2-a1b2-a1b2c3d4e5f6"
                    }
                },
                "http": {
                    "request": {
                        "url": "https://example.com/dev?
username=demo&message=hellofromdemo/",
                        "method": "GET",
                        "client_ip": "192.0.2.0",
                        "x_forwarded_for": true
                    },
                    "response": {
                        "status": 200,
                        "content_length": 0
                    }
                },
                "aws": {
                    "xray": {
                        "sampling_rule_name": "Default"
                    },
                    "api_gateway": {
                        "account_id": "123412341234",
                        "rest_api_id": "a1b2c3d4e5",
                        "stage": "dev",
                        "request_id": "a1b2c3d4-a1b2-a1b2-a1b2-a1b2c3d4e5f6"
                    }
                },
                "annotations": {
                    "aws:api_id": "a1b2c3d4e5",
                    "aws:api_stage": "dev"
                },
                "trace_id": "1-a1b2c3d4-a1b2c3d4a1b2c3d4a1b2c3d4",
                "origin": "AWS::ApiGateway::Stage",
```

```
                "resource_arn": "arn:aws:apigateway:us-east-1::/restapis/a1b2c3d4e5/
stages/dev",
                "subsegments": [
                    {
                        "id": "abcdefgh12345678",
                        "name": "Lambda",
                        "start_time": 1533928226.233,
                        "end_time": 1533928226.6130002,
                        "http": {
                            "request": {
                                "url": "https://example.com/2015-03-31/functions/
arn:aws:lambda:us-east-1:123412341234:function:xray123/invocations",
                                "method": "GET"
                            },
                            "response": {
                                "status": 200,
                                "content_length": 62
                            }
                        },
                        "aws": {
                            "function_name": "xray123",
                            "region": "us-east-1",
                            "operation": "Invoke",
                            "resource_names": [
                                "xray123"
                            ]
                        },
                        "namespace": "aws"
                    }
                ]
            },
            "Id": "a1b2c3d4a1b2c3d4"
        }
```

使用 HTTP API

REST API 和 HTTP API 都是 RESTful API 产品。REST API 支持的功能比 HTTP API 多，而 HTTP API 在设计时功能就极少，因此能够以更低的价格提供。有关更多信息，请参阅 the section called “在 REST API 和 HTTP API 之间选择 ”。

您可以使用 HTTP API 将请求发送到 AWS Lambda 函数或任何可路由的 HTTP 终端节点。例如，您可以创建一个与后端上的 Lambda 函数集成的 HTTP API。当客户端调用您的 API 时，API Gateway 将请求发送到 Lambda 函数并将该函数的响应返回给客户端。

HTTP API 支持 OpenID Connect 和 OAuth 2.0 授权。它们内置了对跨域资源共享 (CORS) 和自动部署的支持。

您可以使用AWS管理控制台、AWS CLI、API、AWS CloudFormation 或开发工具包创建 HTTP API。

主题

- 在 API Gateway 中开发 HTTP API
- 发布 HTTP API 以供客户调用
- 保护您的 HTTP API
- 监控您的 HTTP API
- 排查 HTTP API 的问题

在 API Gateway 中开发 HTTP API

本节提供有关开发 API Gateway API 时所需的 API Gateway 功能的详细信息。

在开发 API Gateway API 时，您可以决定 API 的许多特征。这些特征取决于 API 的使用案例。例如，您可能希望仅允许某些客户端调用您的 API，或者您可能希望它对所有人都可用。您可能需要 API 调用来执行 Lambda 函数、进行数据库查询或调用应用程序。

主题

- 创建 HTTP API
- 使用 HTTP API 的路由
- 控制和管理对 API Gateway 中 HTTP API 的访问
- 为 HTTP API 配置集成
- 为 HTTP API 配置 CORS

- 转换 API 请求和响应
- 将 OpenAPI 定义用于 HTTP API

创建 HTTP API

要创建功能 API，您必须至少有一个路由、集成、阶段和部署。

以下示例显示如何创建具有 AWS Lambda 或 HTTP 集成、路由和默认阶段（配置为自动部署更改）的 API。

本指南以您已熟悉 API Gateway 和 Lambda 为前提。如需更详细的指南，请参阅开始使用。

主题

- 使用 创建 HTTP APIAWS Management Console
- 使用AWS CLI 创建 HTTP API

使用 创建 HTTP APIAWS Management Console

1. 打开 API Gateway 控制台。
2. 选择 Create API (创建 API)。
3. 在 HTTP API 下，选择 Build (构建)。
4. 选择 Add integration (添加集成)，然后选择一个 AWS Lambda 函数或输入 HTTP 终端节点。
5. 对于 Name (名称)，输入 API 的名称。
6. 选择 Review and create。
7. 选择创建。

现在，您的 API 已准备好，可进行调用。您可以通过在浏览器中输入其调用 URL 或使用 Crul 来测试您的 API。

```
curl https://api-id.execute-api.us-east-2.amazonaws.com
```

使用AWS CLI 创建 HTTP API

您可以使用“快速创建”功能，创建具有 Lambda 或 HTTP 集成、默认“捕获全部”路由和默认阶段（配置为自动部署更改）的 API。以下命令使用快速创建来创建与后端上的 Lambda 函数集成的 API。

Note

要调用 Lambda 集成，API Gateway 必须具有所需的权限。您可以使用基于资源的策略或 IAM 角色，为 API Gateway 授予权限以调用 Lambda 函数。如需了解更多信息，请参阅 AWS Lambda 开发人员指南中的 AWS Lambda 权限。

Example

```
aws apigatewayv2 create-api --name my-api --protocol-type HTTP --target
 arn:aws:lambda:us-east-2:123456789012:function:function-name
```

现在，您的 API 已准备好，可进行调用。您可以通过在浏览器中输入其调用 URL 或使用 Crul 来测试您的 API。

```
curl https://api-id.execute-api.us-east-2.amazonaws.com
```

使用 HTTP API 的路由

将直接传入 API 请求路由到后端资源。路由包含两部分：HTTP 方法和资源路径，例如，`GET /pets`。您可以为路由定义特定的 HTTP 方法。或者，您可以使用 ANY 方法匹配尚未为资源定义的所有方法。您可以创建一个 `$default` 路由，用作与任何其他路由不匹配的请求的“捕获全部”方法。

Note

API Gateway 在将 URL 编码的请求参数传递给后端集成之前对其进行解码。

使用路径变量

您可以在 HTTP API 路由中使用路径变量。

例如，`GET /pets/{petID}` 路由捕获客户端提交给 `GET` 的 `https://api-id.execute-api.us-east-2.amazonaws.com/pets/6` 请求。

贪婪的路径变量 捕获路由的所有子资源。要创建“贪婪”路径变量，请将 `+` 添加到变量名称，例如 `{proxy+}`。“贪婪”路径变量必须位于资源路径的末尾。

使用查询字符串参数

默认情况下，如果查询字符串参数包含在对 HTTP API 的请求中，则 API Gateway 会将查询字符串参数发送到您的后端集成。

例如，当客户端向 `https://`*`api-id`*`.execute-api.`*`us-east-2`*`.amazonaws.com/pets`*`?id=4&type=dog`* 发送请求时，查询字符串参数 `?id=4&type=dog` 将发送到您的集成。

使用 `$default` 路由

`$default` 路由捕获与 API 中的其他路由未显式匹配的请求。

当 `$default` 路由收到请求时，API Gateway 将完整的请求路径发送到集成。例如，您可以创建仅包含 `$default` 路由的 API，并将其与 ANY HTTP 终端节点集成到 `https://petstore-demo-endpoint.execute-api.com` 方法上。当您向 `https://`*`api-id`*`.execute-api.`*`us-east-2`*`.amazonaws.com/store/checkout` 发送请求时，API Gateway 会向 `https://petstore-demo-endpoint.execute-api.com/store/checkout` 发送请求。

要了解有关 HTTP 集成的更多信息，请参阅 使用 HTTP API 的 HTTP 代理集成。

路由 API 请求

当客户端发送 API 请求时，API Gateway 首先确定要将请求路由到哪个阶段 。如果请求显式匹配某个阶段，则 API Gateway 将请求发送到该阶段。如果没有阶段与请求完全匹配，API Gateway 将请求发送到 `$default` 阶段。如果没有 `$default` 阶段，则 API 返回 `{"message":"Not Found"}` 并且不生成 CloudWatch 日志。

选择阶段后，API Gateway 将选择路由。API Gateway 使用以下优先级选择具有最佳匹配项的路由：

1. 路由和方法的完全匹配。
2. 匹配具有贪婪路径变量 (`{proxy+}`) 的路由和方法。
3. `$default` 路由。

如果没有与请求匹配的路由，API Gateway 将 `{"message":"Not Found"}` 返回到客户端。

例如，考虑一个具有 `$default` 阶段和以下示例路由的 API：

1. `GET /pets/dog/1`
2. `GET /pets/dog/{id}`

3. `GET /pets/{proxy+}`
4. `ANY /{proxy+}`
5. `$default`

下表汇总了 API Gateway 如何将请求路由到示例路由。

请求	选定的路由	说明
`GET https://`*`api-id`*`.execute-api.`*`region`*`.amazonaws.com/pets/dog/1`	`GET /pets/dog/1`	请求与此静态路由完全匹配。
`GET https://`*`api-id`*`.execute-api.`*`region`*`.amazonaws.com/pets/dog/2`	`GET /pets/dog/{id}`	请求与此路由完全匹配。
`GET https://`*`api-id`*`.execute-api.`*`region`*`.amazonaws.com/pets/cat/1`	`GET /pets/{proxy+}`	请求与路由不完全匹配。具有 `GET` 方法和贪婪路径变量的路由捕获此请求。
`POST https://`*`api-id`*`.execute-api.`*`region`*`.amazonaws.com/test/5`	`ANY /{proxy+}`	`ANY` 方法匹配尚未为路由定义的所有方法。具有贪婪路径变量的路由比 `$default` 路由的优先级更高。

控制和管理对 API Gateway 中 HTTP API 的访问

API Gateway 支持多种用于控制和管理对 HTTP API 的访问的机制：

- Lambda 授权方 使用 Lambda 函数来控制对 API 的访问。有关更多信息，请参阅 使用适用于 HTTP API 的 AWS Lambda 授权方。
- JWT 授权方 使用 JSON Web 令牌来控制对 API 的访问。有关更多信息，请参阅 使用 JWT 授权方控制对 HTTP API 的访问。

- 标准AWS IAM 角色和策略提供灵活且稳健的访问控制。您可以使用 IAM 角色和策略来控制哪些人可以创建和管理您的 API，以及谁可以调用它们。有关更多信息，请参阅 使用 IAM 授权。

使用适用于 HTTP API 的 AWS Lambda 授权方

您可以使用 Lambda 授权方以通过 Lambda 函数控制对 HTTP API 的访问。然后，当客户端调用您的 API 时，API Gateway 会调用您的 Lambda 函数。API Gateway 使用来自 Lambda 函数的响应来确定客户端是否可以访问您的 API。

负载格式版本

授权方负载格式版本指定 API Gateway 发送到 Lambda 授权方的数据格式，以及 API Gateway 如何解释来自 Lambda 的响应。如果未指定负载格式版本，则默认情况下 AWS Management Console 使用最新版本。如果您通过使用 AWS CLI、AWS CloudFormation 或开发工具包创建 Lambda 授权方，则必须指定 `authorizerPayloadFormatVersion`。支持的值是 `1.0` 和 `2.0`。

如果您需要与 REST API 兼容，请使用版本 `1.0`。

以下示例显示了每个负载格式版本的结构。

2.0

```
{
  "version": "2.0",
  "type": "REQUEST",
  "routeArn": "arn:aws:execute-api:us-east-1:123456789012:abcdef123/test/GET/
request",
  "identitySource": ["user1", "123"],
  "routeKey": "$default",
  "rawPath": "/my/path",
  "rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",
  "cookies": ["cookie1", "cookie2"],
  "headers": {
    "header1": "value1",
    "header2": "value2"
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
  "requestContext": {
    "accountId": "123456789012",
```

```
    "apiId": "api-id",
    "authentication": {
      "clientCert": {
        "clientCertPem": "CERT_CONTENT",
        "subjectDN": "www.example.com",
        "issuerDN": "Example issuer",
        "serialNumber": "1",
        "validity": {
          "notBefore": "May 28 12:30:02 2019 GMT",
          "notAfter": "Aug  5 09:36:04 2021 GMT"
        }
      }
    },
    "domainName": "id.execute-api.us-east-1.amazonaws.com",
    "domainPrefix": "id",
    "http": {
      "method": "POST",
      "path": "/my/path",
      "protocol": "HTTP/1.1",
      "sourceIp": "IP",
      "userAgent": "agent"
    },
    "requestId": "id",
    "routeKey": "$default",
    "stage": "$default",
    "time": "12/Mar/2020:19:03:58 +0000",
    "timeEpoch": 1583348638390
  },
  "pathParameters": { "parameter1": "value1" },
  "stageVariables": { "stageVariable1": "value1", "stageVariable2": "value2" }
}
```

1.0

```
{
  "version": "1.0",
  "type": "REQUEST",
  "methodArn": "arn:aws:execute-api:us-east-1:123456789012:abcdef123/test/GET/
request",
  "identitySource": "user1,123",
  "authorizationToken": "user1,123",
  "resource": "/request",
  "path": "/request",
```

```
  "httpMethod": "GET",
  "headers": {
    "X-AMZ-Date": "20170718T062915Z",
    "Accept": "*/*",
    "HeaderAuth1": "headerValue1",
    "CloudFront-Viewer-Country": "US",
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Is-Mobile-Viewer": "false",
    "User-Agent": "..."
  },
  "queryStringParameters": {
    "QueryString1": "queryValue1"
  },
  "pathParameters": {},
  "stageVariables": {
    "StageVar1": "stageValue1"
  },
  "requestContext": {
    "path": "/request",
    "accountId": "123456789012",
    "resourceId": "05c7jb",
    "stage": "test",
    "requestId": "...",
    "identity": {
      "apiKey": "...",
      "sourceIp": "...",
      "clientCert": {
        "clientCertPem": "CERT_CONTENT",
        "subjectDN": "www.example.com",
        "issuerDN": "Example issuer",
        "serialNumber": "a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1",
        "validity": {
          "notBefore": "May 28 12:30:02 2019 GMT",
          "notAfter": "Aug  5 09:36:04 2021 GMT"
        }
      }
    },
    "resourcePath": "/request",
    "httpMethod": "GET",
    "apiId": "abcdef123"
  }
}
```

Lambda 授权方响应格式

负载格式版本还确定您必须从 Lambda 函数返回的响应的结构。

格式 1.0 的 Lambda 函数响应

如果您选择 `1.0` 格式版本， Lambda 授权方必须返回允许或拒绝访问您的 API 路由的 IAM 策略。您可以在策略中使用标准 IAM 语法。有关 IAM 策略的示例，请参阅 the section called " 针对调用 API 的访问控制"。您可以使用 `$context.authorizer.`*`property`* 将上下文属性传递到 Lambda 集成或访问日志。`context` 对象是可选的，`claims` 是保留的占位符，不能用作上下文对象。要了解更多信息，请参阅 the section called "日志记录变量"。

Example

```
{
  "principalId": "abcdef", // The principal user identification associated with the
 token sent by the client.
  "policyDocument": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action": "execute-api:Invoke",
        "Effect": "Allow|Deny",
        "Resource": "arn:aws:execute-api:{regionId}:{accountId}:{apiId}/{stage}/
{httpVerb}/[{resource}/[{child-resources}]]"
      }
    ]
  },
  "context": {
    "exampleKey": "exampleValue"
  }
}
```

格式 2.0 的 Lambda 函数响应

如果选择 `2.0` 格式版本，则可以从 Lambda 函数中返回使用标准 IAM 策略语法的布尔值或 IAM 策略。要返回布尔值，请为授权方启用简单响应。以下示例演示您必须对 Lambda 函数进行编码才能返回的格式。`context` 对象是可选的。您可以使用 `$context.authorizer.`*`property`* 将上下文属性传递到 Lambda 集成或访问日志。要了解更多信息，请参阅"the section called "日志记录变量""。

Simple response

```
{
  "isAuthorized": true/false,
  "context": {
    "exampleKey": "exampleValue"
  }
}
```

IAM policy

```
{
  "principalId": "abcdef", // The principal user identification associated with the
 token sent by the client.
  "policyDocument": {
    "Version": "2012-10-17",
    "Statement": [
      {
        "Action": "execute-api:Invoke",
        "Effect": "Allow|Deny",
        "Resource": "arn:aws:execute-api:{regionId}:{accountId}:{apiId}/{stage}/
{httpVerb}/[{resource}/[{child-resources}]]"
      }
    ]
  },
  "context": {
    "exampleKey": "exampleValue"
  }
}
```

Lambda 授权方函数示例

以下示例 Node.js Lambda 函数演示您需要从 `2.0` 负载格式版本的 Lambda 函数返回的所需响应格式。

Simple response - Node.js

```
export const handler = async(event) => {
    let response = {
        "isAuthorized": false,
        "context": {
            "stringKey": "value",
```

```
            "numberKey": 1,
            "booleanKey": true,
            "arrayKey": ["value1", "value2"],
            "mapKey": {"value1": "value2"}
        }
    };

    if (event.headers.authorization === "secretToken") {
        console.log("allowed");
        response = {
            "isAuthorized": true,
            "context": {
                "stringKey": "value",
                "numberKey": 1,
                "booleanKey": true,
                "arrayKey": ["value1", "value2"],
                "mapKey": {"value1": "value2"}
            }
        };
    }

    return response;

};
```

Simple response - Python

```
import json


def lambda_handler(event, context):
    response = {
        "isAuthorized": False,
        "context": {
            "stringKey": "value",
            "numberKey": 1,
            "booleanKey": True,
            "arrayKey": ["value1", "value2"],
            "mapKey": {"value1": "value2"}
        }
    }

    try:
```

```
        if (event["headers"]["authorization"] == "secretToken"):
            response = {
                "isAuthorized": True,
                "context": {
                    "stringKey": "value",
                    "numberKey": 1,
                    "booleanKey": True,
                    "arrayKey": ["value1", "value2"],
                    "mapKey": {"value1": "value2"}
                }
            }
            print('allowed')
            return response
        else:
            print('denied')
            return response
    except BaseException:
        print('denied')
        return response
```

IAM policy - Node.js

```
export const handler = async(event) => {
  if (event.headers.authorization == "secretToken") {
    console.log("allowed");
    return {
      "principalId": "abcdef", // The principal user identification associated with
 the token sent by the client.
      "policyDocument": {
        "Version": "2012-10-17",
        "Statement": [{
          "Action": "execute-api:Invoke",
          "Effect": "Allow",
          "Resource": event.routeArn
        }]
      },
      "context": {
        "stringKey": "value",
        "numberKey": 1,
        "booleanKey": true,
        "arrayKey": ["value1", "value2"],
        "mapKey": { "value1": "value2" }
      }
```

```
    };
  }
  else {
    console.log("denied");
    return {
      "principalId": "abcdef", // The principal user identification associated with
 the token sent by the client.
      "policyDocument": {
        "Version": "2012-10-17",
        "Statement": [{
          "Action": "execute-api:Invoke",
          "Effect": "Deny",
          "Resource": event.routeArn
        }]
      },
      "context": {
        "stringKey": "value",
        "numberKey": 1,
        "booleanKey": true,
        "arrayKey": ["value1", "value2"],
        "mapKey": { "value1": "value2" }
      }
    };
  }
};
```

IAM policy - Python

```
import json


def lambda_handler(event, context):
    response = {
        # The principal user identification associated with the token sent by
        # the client.
        "principalId": "abcdef",
        "policyDocument": {
            "Version": "2012-10-17",
            "Statement": [{
                "Action": "execute-api:Invoke",
                "Effect": "Deny",
                "Resource": event["routeArn"]
            }]
```

```
        },
        "context": {
            "stringKey": "value",
            "numberKey": 1,
            "booleanKey": True,
            "arrayKey": ["value1", "value2"],
            "mapKey": {"value1": "value2"}
        }
    }

    try:
        if (event["headers"]["authorization"] == "secretToken"):
            response = {
                # The principal user identification associated with the token
                # sent by the client.
                "principalId": "abcdef",
                "policyDocument": {
                    "Version": "2012-10-17",
                    "Statement": [{
                        "Action": "execute-api:Invoke",
                        "Effect": "Allow",
                        "Resource": event["routeArn"]
                    }]
                },
                "context": {
                    "stringKey": "value",
                    "numberKey": 1,
                    "booleanKey": True,
                    "arrayKey": ["value1", "value2"],
                    "mapKey": {"value1": "value2"}
                }
            }
            print('allowed')
            return response
        else:
            print('denied')
            return response
    except BaseException:
        print('denied')
        return response
```

身份来源

您可以选择指定 Lambda 授权方的身份来源。身份来源指定对请求进行授权所需的数据的位置。例如，您可以将标头或查询字符串值指定为身份来源。如果您指定身份来源，则客户端必须将其包含在请求中。如果客户端的请求不包括身份源，则 API Gateway 不会调用您的 Lambda 授权方，客户端会收到 401 错误。支持以下身份来源：

选择表达式

类型	示例	备注
标头值	$request.header.*name*	标头名称不区分大小写。
查询字符串值	$request.querystring.*name*	查询字符串名称区分大小写。
上下文变量	$context.*variableName*	受支持的上下文变量的值。
阶段变量	$stageVariables.*variableName*	阶段变量的值。

缓存授权方响应

您可以通过指定 authorizerResultTtlInSeconds 为 Lambda 授权方启用缓存。为授权方启用缓存时，API Gateway 使用授权方的身份来源作为缓存密钥。如果客户端在配置的 TTL 内在身份来源中指定了相同的参数，则 API Gateway 使用缓存的授权方结果，而不调用 Lambda 函数。

要启用缓存，授权方必须至少有一个身份来源。

如果为授权方启用简单响应，则授权方的响应将完全允许或拒绝与缓存的身份来源值匹配的所有 API 请求。要获得更精细的权限，请禁用简单响应并返回 IAM 策略。

默认情况下，API Gateway 对使用授权方的 API 的所有路由使用缓存的授权方响应。要缓存每条路由的响应，请将 `$context.routeKey` 添加到授权方的身份来源中。

创建 Lambda 授权方

创建 Lambda 授权方时，需要指定供 API Gateway 使用的 Lambda 函数。您必须使用函数的资源策略或 IAM 角色向 API Gateway 授予调用 Lambda 函数的权限。在此示例中，我们更新函数的资源策略，以便它授予 API Gateway 调用 Lambda 函数的权限。

```
aws apigatewayv2 create-authorizer \
```

```
    --api-id abcdef123 \
    --authorizer-type REQUEST \
    --identity-source '$request.header.Authorization' \
    --name lambda-authorizer \
    --authorizer-uri 'arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-west-2:123456789012:function:my-function/invocations' \
    --authorizer-payload-format-version '2.0' \
    --enable-simple-responses
```

以下命令授予 API Gateway 调用 Lambda 函数的权限。如果 API Gateway 没有调用函数的权限，客户端会收到 `500 Internal Server Error`。

```
aws lambda add-permission \
    --function-name my-authorizer-function \
    --statement-id apigateway-invoke-permissions-abc123 \
    --action lambda:InvokeFunction \
    --principal apigateway.amazonaws.com \
    --source-arn "arn:aws:execute-api:us-west-2:123456789012:api-
id/authorizers/authorizer-id"
```

创建授权方并授予 API Gateway 调用授权方的权限后，请更新您的路由以使用此授权方。

```
aws apigatewayv2 update-route \
    --api-id abcdef123 \
    --route-id acd123 \
    --authorization-type CUSTOM \
    --authorizer-id def123
```

Lambda 授权方故障排除

如果 API Gateway 无法调用 Lambda 授权方，或者您的 Lambda 授权方返回无效格式的响应，则客户端将收到 `500 Internal Server Error`。

要排除错误，请为 API 阶段启用访问日志记录。以 `$context.authorizer.error` 日志格式包含日志记录变量。

如果日志表明 API Gateway 无权调用您的函数，请更新函数的资源策略或提供 IAM 角色，以授予 API Gateway 调用授权方的权限。

如果日志表明您的 Lambda 函数返回无效响应，请验证您的 Lambda 函数以所需格式返回响应。

使用 JWT 授权方控制对 HTTP API 的访问

您可以使用 JSON Web 令牌 (JWT) 作为 OpenID Connect (OIDC) 和 OAuth 2.0 框架的一部分来限制客户端对您的 API 的访问。

如果您为 API 的路由配置 JWT 授权方，API Gateway 将验证客户端通过 API 请求提交的 JWT。API Gateway 根据令牌中的令牌验证和（可选）作用域来允许或拒绝请求。如果为路由配置作用域，则令牌必须至少包含路由的作用域之一。

您可以为 API 的每个路由配置不同的授权方，也可以为多个路由使用同一个授权方。

> **Note**
>
> 没有标准的机制可以将 JWT 访问令牌与其他类型的 JWT（如 OpenID Connect ID 令牌）区分开来。除非您需要 ID 令牌进行 API 授权，否则我们建议您将路由配置为需要授权范围。您还可以将 JWT 授权方配置为需要相应的发布者或受众（身份提供商仅在发布 JWT 访问令牌时才使用他们）。

使用 JWT 授权方授权 API 请求

API Gateway 使用以下常规工作流程向配置为使用 JWT 授权方的路由授权请求。

1. 检查 `identitySource` 是否有令牌。`identitySource` 只能包含令牌或前缀为 `Bearer` 的令牌。
2. 对令牌解码。
3. 使用从发布者的 `jwks_uri` 中提取的公有密钥检查令牌的算法和签名。目前仅支持基于 RSA 的算法。API Gateway 可以将公有密钥缓存两个小时。最佳实践是，在轮换密钥时，留出一段宽限期，在此期间旧密钥和新密钥均有效。
4. 验证声明。API Gateway 评估以下令牌声明：
    - `kid` – 令牌必须具有与签署令牌的 `jwks_uri` 中的键匹配的标头声明。
    - `iss` – 必须匹配为授权方配置的 `issuer`。
    - `aud` 或 `client_id` – 必须匹配为授权方配置的其中一个 `audience` 条目。API Gateway 只有在 `aud` 不存在时才验证 `client_id`。当 `aud` 和 `client_id` 同时存在时，API Gateway 会评估 `aud`。
    - `exp` – 必须在 UTC 中的当前时间之后。
    - `nbf` – 必须在 UTC 中的当前时间之前。
    - `iat` – 必须在 UTC 中的当前时间之前。
    - `scope` 或 `scp` – 令牌必须至少包含路由的 `authorizationScopes` 中的一个作用域。

如果这些步骤中的任何一个步骤失败，API Gateway 都会拒绝 API 请求。

验证 JWT 后，API Gateway 将令牌中的声明传递给 API 路由的集成。后端资源（如 Lambda 函数）可以访问 JWT 声明。例如，如果 JWT 包含了身份声明 `emailID`，则它在 `$event.requestContext.authorizer.jwt.claims.emailID` 中可供 Lambda 集成使用。有关 API Gateway 向 Lambda 集成发送的负载的更多信息，请参阅the section called “AWS Lambda 集成”。

创建 JWT 授权方

在创建 JWT 授权方之前，必须向身份提供商注册客户端应用程序。您必须还创建了 HTTP API。有关创建 HTTP API 的示例，请参阅 创建 HTTP API。

使用控制台创建 JWT 授权方

以下步骤说明如何使用控制台创建 JWT 授权方。

使用控制台创建 JWT 授权方

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 HTTP API。
3. 在主导航窗格中，选择授权。
4. 选择管理授权方选项卡。
5. 选择创建。
6. 对于授权方类型，选择 JWT。
7. 配置您的 JWT 授权方，并指定用于定义令牌来源的身份来源。
8. 选择创建。

使用 AWS CLI 创建 JWT 授权方

以下 AWS CLI 命令将创建 JWT 授权方。对于 `jwt-configuration`，为您的身份提供商指定 `Audience` 和 `Issuer`。如果您使用 Amazon Cognito 作为身份提供者，则 `IssuerUrl` 为 `https://cognito-idp.`*`us-east-2`*`.amazonaws.com/`*`userPoolID`*。

```
aws apigatewayv2 create-authorizer \
    --name authorizer-name \
    --api-id api-id \
    --authorizer-type JWT \
```

```
    --identity-source '$request.header.Authorization' \
    --jwt-configuration Audience=audience,Issuer=IssuerUrl
```

使用 AWS CloudFormation 创建 JWT 授权方

以下 AWS CloudFormation 模板通过将 Amazon Cognito 用作身份提供者的 JWT 授权方创建 HTTP API。

AWS CloudFormation 模板的输出是 Amazon Cognito 托管用户界面的 URL，客户可以在其中注册并登录来接收 JWT。客户端登录后，会使用 URL 中的访问令牌将客户端重定向到您的 HTTP API。要使用访问令牌调用 API，请将 URL 中的 # 更改为 ?，来使用该令牌作为查询字符串参数。

示例 AWS CloudFormation 模板

```
AWSTemplateFormatVersion: '2010-09-09'
Description: |
  Example HTTP API with a JWT authorizer. This template includes an Amazon Cognito user
 pool as the issuer for the JWT authorizer
  and an Amazon Cognito app client as the audience for the authorizer. The outputs
 include a URL for an Amazon Cognito hosted UI where clients can
  sign up and sign in to receive a JWT. After a client signs in, the client is
 redirected to your HTTP API with an access token
  in the URL. To invoke the API with the access token, change the '#' in the URL to a
 '?' to use the token as a query string parameter.

Resources:
  MyAPI:
    Type: AWS::ApiGatewayV2::Api
    Properties:
      Description: Example HTTP API
      Name: api-with-auth
      ProtocolType: HTTP
      Target: !GetAtt MyLambdaFunction.Arn
  DefaultRouteOverrides:
    Type: AWS::ApiGatewayV2::ApiGatewayManagedOverrides
    Properties:
      ApiId: !Ref MyAPI
      Route:
        AuthorizationType: JWT
        AuthorizerId: !Ref JWTAuthorizer
  JWTAuthorizer:
    Type: AWS::ApiGatewayV2::Authorizer
    Properties:
```

```
      ApiId: !Ref MyAPI
      AuthorizerType: JWT
      IdentitySource:
        - '$request.querystring.access_token'
      JwtConfiguration:
        Audience:
        - !Ref AppClient
        Issuer: !Sub https://cognito-idp.${AWS::Region}.amazonaws.com/${UserPool}
      Name: test-jwt-authorizer
  MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      Runtime: nodejs18.x
      Role: !GetAtt FunctionExecutionRole.Arn
      Handler: index.handler
      Code:
        ZipFile: |
          exports.handler = async (event) => {
              const response = {
                  statusCode: 200,
                  body: JSON.stringify('Hello from the ' + event.routeKey + ' route!'),
              };
              return response;
          };
  APIInvokeLambdaPermission:
    Type: AWS::Lambda::Permission
    Properties:
      FunctionName: !Ref MyLambdaFunction
      Action: lambda:InvokeFunction
      Principal: apigateway.amazonaws.com
      SourceArn: !Sub arn:${AWS::Partition}:execute-api:${AWS::Region}:
${AWS::AccountId}:${MyAPI}/$default/$default
  FunctionExecutionRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service:
              - lambda.amazonaws.com
            Action:
              - 'sts:AssumeRole'
```

```
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
  UserPool:
    Type: AWS::Cognito::UserPool
    Properties:
      UserPoolName: http-api-user-pool
      AutoVerifiedAttributes:
        - email
      Schema:
        - Name: name
          AttributeDataType: String
          Mutable: true
          Required: true
        - Name: email
          AttributeDataType: String
          Mutable: false
          Required: true
  AppClient:
    Type: AWS::Cognito::UserPoolClient
    Properties:
      AllowedOAuthFlows:
        - implicit
      AllowedOAuthScopes:
        - aws.cognito.signin.user.admin
        - email
        - openid
        - profile
      AllowedOAuthFlowsUserPoolClient: true
      ClientName: api-app-client
      CallbackURLs:
        - !Sub https://${MyAPI}.execute-api.${AWS::Region}.amazonaws.com
      ExplicitAuthFlows:
        - ALLOW_USER_PASSWORD_AUTH
        - ALLOW_REFRESH_TOKEN_AUTH
      UserPoolId: !Ref UserPool
      SupportedIdentityProviders:
        - COGNITO
  HostedUI:
    Type: AWS::Cognito::UserPoolDomain
    Properties:
      Domain: !Join
        - '-'
        - - !Ref MyAPI
          - !Ref AppClient
```

```
      UserPoolId: !Ref UserPool
Outputs:
  SignupURL:
    Value: !Sub https://${HostedUI}.auth.${AWS::Region}.amazoncognito.com/login?
client_id=${AppClient}&response_type=token&scope=email+profile&redirect_uri=https://
${MyAPI}.execute-api.${AWS::Region}.amazonaws.com
```

更新路由来使用 JWT 授权方

您可以使用控制台、AWS CLI 或 AWS SDK 更新路由来使用 JWT 授权方。

使用控制台更新路由来使用 JWT 授权方

以下步骤说明如何使用控制台更新路由来使用 JWT 授权方。

使用控制台创建 JWT 授权方

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 HTTP API。
3. 在主导航窗格中，选择授权。
4. 选择一个方法，然后从下拉菜单中选择您的授权方，然后选择附加授权方。

使用 AWS CLI 更新路由来使用 JWT 授权方

以下命令使用 AWS CLI 更新路由来使用 JWT 授权方。

```
aws apigatewayv2 update-route \
   --api-id api-id  \
   --route-id route-id  \
   --authorization-type JWT \
   --authorizer-id authorizer-id \
   --authorization-scopes user.email
```

使用 IAM 授权

您可以为 HTTP API 路由启用 IAM 授权。启用 IAM 授权后，客户端必须使用签名版本 4（SigV4），才能使用 AWS 凭证签署其请求。仅当客户端对路由具有 `execute-api` 权限时，API Gateway 才会调用您的 API 路由。

用于 HTTP API 的 IAM 授权类似于用于 REST API 的授权。

ⓘ Note

HTTP API 目前不支持资源策略。

有关授予客户端调用 API 的 IAM 权限的策略示例，请参阅 the section called " 针对调用 API 的访问控制"。

为路由启用 IAM 授权

以下 AWS CLI 命令会为 HTTP API 路由启用 IAM 授权。

```
aws apigatewayv2 update-route \
    --api-id abc123 \
    --route-id abcdef \
    --authorization-type AWS_IAM
```

为 HTTP API 配置集成

集成 将路由连接到后端资源。HTTP API 支持 Lambda 代理、AWS服务和 HTTP 代理集成。例如，您可以配置对 API 的 `POST` 路由的 `/signup` 请求，以便与处理注册客户的 Lambda 函数集成。

主题

- 使用 HTTP API 的 AWS Lambda 代理集成
- 使用 HTTP API 的 HTTP 代理集成
- 使用针对 HTTP API 的 AWS 服务集成
- 使用 HTTP API 的私有集成

使用 HTTP API 的 AWS Lambda 代理集成

通过 Lambda 代理集成，您可以将 API 路由与 Lambda 函数集成。当客户端调用您的 API 时，API Gateway 将请求发送到 Lambda 函数并将该函数的响应返回给客户端。有关创建 HTTP API 的示例，请参阅 创建 HTTP API。

负载格式版本

负载格式版本指定 API Gateway 发送到 Lambda 集成的事件的格式，以及 API Gateway 如何解释来自 Lambda 的响应。如果未指定负载格式版本，则默认情况下 AWS Management Console 使用最新

版本。如果您通过使用 AWS CLI、AWS CloudFormation 或开发工具包创建 Lambda 集成，则必须指定 `payloadFormatVersion`。支持的值是 `1.0` 和 `2.0`。

有关如何设置 `payloadFormatVersion` 的更多信息，请参阅 create-integration。有关如何确定现有集成的 `payloadFormatVersion` 的更多信息，请参阅 get-integration。

负载格式差异

以下列表显示了 `1.0` 和 `2.0` 负载格式版本之间的差异：

- 格式 `2.0` 没有 `multiValueHeaders` 或 `multiValueQueryStringParameters` 字段。重复的标头使用逗号组合，包含在 `headers` 字段中。重复的查询字符串使用逗号组合，包含在 `queryStringParameters` 字段中。
- 格式 `2.0` 具有 `rawPath`。如果您使用 API 映射将阶段连接到自定义域名，`rawPath` 不会提供 API 映射值。使用格式 `1.0` 和 `path` 可访问自定义域名的 API 映射。
- 格式 `2.0` 包含一个新 `cookies` 字段。请求中的所有 Cookie 标头均使用逗号组合并添加到 `cookies` 字段中。在对客户端的响应中，每个 Cookie 都成为一个 `set-cookie` 标头。

负载格式结构

以下示例显示了每个负载格式版本的结构。所有标头名称均为小写。

2.0

```
{
  "version": "2.0",
  "routeKey": "$default",
  "rawPath": "/my/path",
  "rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",
  "cookies": [
    "cookie1",
    "cookie2"
  ],
  "headers": {
    "header1": "value1",
    "header2": "value1,value2"
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
```

```
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "api-id",
    "authentication": {
      "clientCert": {
        "clientCertPem": "CERT_CONTENT",
        "subjectDN": "www.example.com",
        "issuerDN": "Example issuer",
        "serialNumber": "a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1",
        "validity": {
          "notBefore": "May 28 12:30:02 2019 GMT",
          "notAfter": "Aug  5 09:36:04 2021 GMT"
        }
      }
    },
    "authorizer": {
      "jwt": {
        "claims": {
          "claim1": "value1",
          "claim2": "value2"
        },
        "scopes": [
          "scope1",
          "scope2"
        ]
      }
    },
    "domainName": "id.execute-api.us-east-1.amazonaws.com",
    "domainPrefix": "id",
    "http": {
      "method": "POST",
      "path": "/my/path",
      "protocol": "HTTP/1.1",
      "sourceIp": "192.0.2.1",
      "userAgent": "agent"
    },
    "requestId": "id",
    "routeKey": "$default",
    "stage": "$default",
    "time": "12/Mar/2020:19:03:58 +0000",
    "timeEpoch": 1583348638390
  },
  "body": "Hello from Lambda",
  "pathParameters": {
```

```
    "parameter1": "value1"
  },
  "isBase64Encoded": false,
  "stageVariables": {
    "stageVariable1": "value1",
    "stageVariable2": "value2"
  }
}
```

1.0

```
{
  "version": "1.0",
  "resource": "/my/path",
  "path": "/my/path",
  "httpMethod": "GET",
  "headers": {
    "header1": "value1",
    "header2": "value2"
  },
  "multiValueHeaders": {
    "header1": [
      "value1"
    ],
    "header2": [
      "value1",
      "value2"
    ]
  },
  "queryStringParameters": {
    "parameter1": "value1",
    "parameter2": "value"
  },
  "multiValueQueryStringParameters": {
    "parameter1": [
      "value1",
      "value2"
    ],
    "parameter2": [
      "value"
    ]
  },
  "requestContext": {
```

```
    "accountId": "123456789012",
    "apiId": "id",
    "authorizer": {
      "claims": null,
      "scopes": null
    },
    "domainName": "id.execute-api.us-east-1.amazonaws.com",
    "domainPrefix": "id",
    "extendedRequestId": "request-id",
    "httpMethod": "GET",
    "identity": {
      "accessKey": null,
      "accountId": null,
      "caller": null,
      "cognitoAuthenticationProvider": null,
      "cognitoAuthenticationType": null,
      "cognitoIdentityId": null,
      "cognitoIdentityPoolId": null,
      "principalOrgId": null,
      "sourceIp": "192.0.2.1",
      "user": null,
      "userAgent": "user-agent",
      "userArn": null,
      "clientCert": {
        "clientCertPem": "CERT_CONTENT",
        "subjectDN": "www.example.com",
        "issuerDN": "Example issuer",
        "serialNumber": "a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1:a1",
        "validity": {
          "notBefore": "May 28 12:30:02 2019 GMT",
          "notAfter": "Aug  5 09:36:04 2021 GMT"
        }
      }
    },
    "path": "/my/path",
    "protocol": "HTTP/1.1",
    "requestId": "id=",
    "requestTime": "04/Mar/2020:19:15:17 +0000",
    "requestTimeEpoch": 1583349317135,
    "resourceId": null,
    "resourcePath": "/my/path",
    "stage": "$default"
  },
  "pathParameters": null,
```

```
  "stageVariables": null,
  "body": "Hello from Lambda!",
  "isBase64Encoded": false
}
```

Lambda 函数响应格式

负载格式版本确定 Lambda 函数必须返回的响应结构。

格式 1.0 的 Lambda 函数响应

对于 `1.0` 格式版本，Lambda 集成必须返回采用以下 JSON 格式的响应：

Example

```
{
    "isBase64Encoded": true|false,
    "statusCode": httpStatusCode,
    "headers": { "headername": "headervalue", ... },
    "multiValueHeaders": { "headername": ["headervalue", "headervalue2", ...], ... },
    "body": "..."
}
```

格式 2.0 的 Lambda 函数响应

使用 `2.0` 格式版本，API Gateway 可以推断您的响应格式。如果您的 Lambda 函数返回有效的 JSON 并且没有返回 `statusCode`，API Gateway 会做出以下假设：

- `isBase64Encoded` 为 `false`。
- `statusCode` 为 `200`。
- `content-type` 为 `application/json`。
- `body` 是函数的响应。

以下示例显示 Lambda 函数的输出和 API Gateway 的解释。

Lambda 函数输出	API Gateway 解释
`"Hello from Lambda!"`	`{` `  "isBase64Encoded": false,`

Lambda 函数输出	API Gateway 解释

API Gateway 解释：

```
  "statusCode": 200,
  "body": "Hello from Lambda!",
  "headers": {
    "content-type": "application/json"
  }
}
```

Lambda 函数输出：

```
{ "message": "Hello from Lambda!" }
```

API Gateway 解释：

```
{
  "isBase64Encoded": false,
  "statusCode": 200,
  "body": "{ \"message\": \"Hello from Lambda!\" }",
  "headers": {
    "content-type": "application/json"
  }
}
```

要自定义响应，您的 Lambda 函数应返回以下格式的响应。

```
{
    "cookies" : ["cookie1", "cookie2"],
    "isBase64Encoded": true|false,
    "statusCode": httpStatusCode,
    "headers": { "headername": "headervalue", ... },
    "body": "Hello from Lambda!"
}
```

使用 HTTP API 的 HTTP 代理集成

通过 HTTP 代理集成，您可以将 API 路由连接到可公开路由的 HTTP 终端节点。使用这种集成类型，API Gateway 在前端和后端之间传递完整的请求和响应。

要创建 HTTP 代理集成，请提供可公开路由 HTTP 终端节点的 URL。

具有路径变量的 HTTP 代理集成

您可以在 HTTP API 路由中使用路径变量。

例如，路由 `/pets/{petID}` 捕获对 `/pets/6` 的请求。您可以在集成 URI 中引用路径变量，以将变量的内容发送到集成。示例是 `/pets/extendedpath/{petID}`。

您可以使用“贪婪”路径变量来捕获路由的所有子资源。要创建“贪婪”路径变量，请将 + 添加到变量名称，例如 `{proxy+}`。

要设置具有捕获所有请求的 HTTP 代理集成的路由，请使用“贪婪”路径变量创建 API 路由（例如，`/parent/{proxy+}`）。将路由与 `https://petstore-demo-endpoint.execute-api.com/petstore/{proxy}` 方法上的 HTTP 终端节点（例如 ANY）集成。“贪婪”路径变量必须位于资源路径的末尾。

使用针对 HTTP API 的 AWS 服务集成

您可以使用一流的集成将 HTTP API 与AWS服务集成。一流的集成会将 HTTP API 路由连接到AWS服务 API。当客户端调用由一流集成支持的路由时，API Gateway 会为您调用AWS服务 API。例如，您可以使用一流的集成向 Amazon Simple Queue Service 队列发送消息，或启动 AWS Step Functions 状态机。有关支持的服务操作，请参阅 the section called “AWS 服务集成参考”。

映射请求参数

一流的集成具有必需和可选的参数。必须配置所有必需的参数才能创建集成。您可以使用在运行时动态评估的静态值或映射参数。有关支持的集成和参数的完整列表，请参阅the section called “AWS 服务集成参考”。

参数映射

类型	示例	备注
标头值	$request.header.*name*	标头名称不区分大小写。API Gateway 将多个标头值与逗号组合在一起，例如 `"header1": "value1,value2"` 。
查询字符串值	$request.querystring.*name*	查询字符串名称区分大小写。API Gateway 将多个值与逗号组合在一起，例如 `"querystring1": "Value1,Value2"` 。
路径参数	$request.path.*name*	请求中路径参数的值。例如，如果路由为 `/pets/`

类型	示例	备注
		`{petId}`，，则可以从具有 *`$request.path.petId`* 的请求中映射 `petId` 参数。
请求正文传递	$request.body	API Gateway 传递整个请求正文。
请求正文	$request.body.*name*	JSON 路径表达式。不支持递归下降 (`$request.body..` *`name`*) 和筛选表达式 (`?(`*`expression`*`)`)。 Note 当您指定 JSON 路径时，API Gateway 会在请求正文的 100 KB 处将其截断，然后应用选择表达式。要发送大于 100 KB 的负载，请指定 `$request.body` 。
上下文变量	$context.*`variableName`*	受支持的上下文变量的值。
阶段变量	$stageVariables.*`variableName`*	阶段变量的值。
静态值	*`string`*	常量值。

创建一流的集成

在创建一流集成之前，您必须创建一个 IAM 角色，该角色向 API Gateway 授予调用您要集成的AWS服务操作的权限。如需了解详情，请参阅为AWS服务创建角色。

要创建一流集成，请选择受支持的AWS服务操作（例如 `SQS-SendMessage`），配置请求参数，然后提供一个角色来授予 API Gateway 调用集成的AWS服务 API 的权限。根据集成子类型，需要不同的请求参数。要了解更多信息，请参阅the section called “AWS 服务集成参考”。

以下 AWS CLI 命令会创建一个用于发送 Amazon SQS 消息的集成。

```
aws apigatewayv2 create-integration \
    --api-id abcdef123 \
    --integration-subtype SQS-SendMessage \
    --integration-type AWS_PROXY \
    --payload-format-version 1.0 \
    --credentials-arn arn:aws:iam::123456789012:role/apigateway-sqs \
    --request-parameters '{"QueueUrl": "$request.header.queueUrl", "MessageBody":
 "$request.body.message"}'
```

使用 AWS CloudFormation 创建一流的集成

以下示例显示了一个 AWS CloudFormation 代码段，该代码段可创建一个与 Amazon EventBridge 进行一流集成的 `/{source}/{detailType}` 路径。

`Source` 参数映射到 `{source}` 路径参数，`DetailType` 映射到 `{DetailType}` 路径参数，`Detail`参数映射到请求正文。

该代码段不显示事件总线或授予 API Gateway 调用 `PutEvents` 操作的权限的 IAM 角色。

```
Route:
    Type: AWS::ApiGatewayV2::Route
    Properties:
      ApiId: !Ref HttpApi
      AuthorizationType: None
      RouteKey: 'POST /{source}/{detailType}'
      Target: !Join
        - /
        - - integrations
          - !Ref Integration
  Integration:
    Type: AWS::ApiGatewayV2::Integration
    Properties:
      ApiId: !Ref HttpApi
      IntegrationType: AWS_PROXY
      IntegrationSubtype: EventBridge-PutEvents
      CredentialsArn: !GetAtt EventBridgeRole.Arn
```

```
      RequestParameters:
        Source: $request.path.source
        DetailType: $request.path.detailType
        Detail: $request.body
        EventBusName: !GetAtt EventBus.Arn
      PayloadFormatVersion: "1.0"
```

集成子类型参考

HTTP API 支持以下集成子类型。

集成子类型

- EventBridge-PutEvents
- SQS-SendMessage
- SQS-ReceiveMessage
- SQS-DeleteMessage
- SQS-PurgeQueue
- AppConfig-GetConfiguration
- Kinesis-PutRecord
- StepFunctions-StartExecution
- StepFunctions-StartSyncExecution
- StepFunctions-StopExecution

EventBridge-PutEvents

将自定义事件发送到 Amazon EventBridge，以便将其与规则进行匹配。

EventBridge-PutEvents 1.0

参数	必需
Detail	True
DetailType	True
源	True
Time	False

参数	必需
EventBusName	False
资源	False
区域	False
TraceHeader	False

要了解更多信息，请参阅 Amazon EventBridge API 参考 中的 PuteVentts。

SQS-SendMessage

向指定的队列传送了一条消息。

SQS-SendMessage 1.0

参数	必需
QueueUrl	True
MessageBody	True
DelaySeconds	False
MessageAttributes	False
MessageDeduplicationId	False
MessageGroupId	False
MessageSystemAttributes	False
区域	False

要了解更多信息，请参阅 Amazon Simple Queue Service API 参考 中的 SendMessage。

SQS-ReceiveMessage

从指定的队列中检索一条或多条消息（最多 10 条）。

SQS-ReceiveMessage 1.0

参数	必需
QueueUrl	True
AttributeNames	False
MaxNumberOfMessages	False
MessageAttributeNames	False
ReceiveRequestAttemptId	False
VisibilityTimeout	False
WaitTimeSeconds	False
区域	False

要了解更多信息，请参阅 Amazon Simple Queue Service API 参考 中的 ReceiveMessage。

SQS-DeleteMessage

从指定的队列中删除指定的消息。

SQS-DeleteMessage 1.0

参数	必需
ReceiptHandle	True
QueueUrl	True
区域	False

要了解更多信息，请参阅 Amazon Simple Queue Service API 参考 中的 DeleteMessage。

SQS-PurgeQueue

删除指定队列中的所有消息。

SQS-PurgeQueue 1.0

参数	必需
QueueUrl	True
区域	False

要了解更多信息，请参阅 Amazon Simple Queue Service API 参考 中的 PurgeQueue。

AppConfig-GetConfiguration

接收有关配置的信息。

AppConfig-GetConfiguration 1.0

参数	必需
应用程序	True
环境	True
配置	True
ClientId	True
ClientConfigurationVersion	False
区域	False

如需了解详情，请参阅AWS AppConfig API 参考中的 GetConfiguration。

Kinesis-PutRecord

将单个数据记录写入 Amazon Kinesis 数据流。

Kinesis-PutRecord 1.0

参数	必需
StreamName	True

参数	必需
数据	True
PartitionKey	True
SequenceNumberForOrdering	False
ExplicitHashKey	False
区域	False

要了解更多信息，请参阅 Amazon Kinesis Data Streams API 参考 中的 PutRecord。

StepFunctions-StartExecution

启动状态机的一次执行。

StepFunctions-StartExecution 1.0

参数	必需
StateMachineArn	True
名称	False
输入	False
区域	False

如需了解详情，请参阅 AWS Step Functions API 参考中的 StartExecution。

StepFunctions-StartSyncExecution

启动同步状态机执行。

StepFunctions-StartSyncExecution 1.0

参数	必需
StateMachineArn	True

参数	必需
名称	False
输入	False
区域	False
TraceHeader	False

如需了解详情，请参阅 AWS Step Functions API 参考中的 StartSyncExecution。

StepFunctions-StopExecution

停止一次执行。

StepFunctions-StopExecution 1.0

参数	必需
ExecutionArn	True
原因	False
错误	False
区域	False

如需了解详情，请参阅 AWS Step Functions API 参考中的 StopExecution。

使用 HTTP API 的私有集成

私有集成使您能够与 VPC 中的私有资源（如 Application Load Balancer 或基于 Amazon ECS 容器的应用程序）创建 API 集成。

您可以使用私有集成公开 VPC 中的资源，以便 VPC 外部的客户端访问。您可以使用 API Gateway 支持的任何授权方法来控制对 API 的访问。

要创建私有集成，您必须首先创建 VPC 链接。要了解有关 VPC 链接的更多信息，请参阅 为 HTTP API 使用 VPC 链接。

创建 VPC 链接后，您可以设置私有集成，以连接到 Application Load Balancer、Network Load Balancer 或注册到 AWS Cloud Map 服务的资源。

要创建私有集成，所有资源必须归同一AWS账户所有（包括负载均衡器或 AWS Cloud Map 服务、VPC 链接和 HTTP API）。

默认情况下，私有集成流量使用 HTTP 协议。如果您要求私有集成流量使用 HTTPS，则可以指定 `tlsConfig`。

> **Note**
>
> 对于私有集成，API Gateway 在对后端资源的请求中包括 API 终端节点的阶段部分。例如，对 API 的 `test` 阶段的请求在对私有集成的请求中包含 `test/*route-path*`。要从对后端资源的请求中删除阶段名，请使用参数映射覆盖 `$request.path` 的请求路径。

使用 Application Load Balancer 或 Network Load Balancer 创建私有集成

在创建私有集成之前，您必须创建 VPC 链接。要了解有关 VPC 链接的更多信息，请参阅 为 HTTP API 使用 VPC 链接。

要创建与 Application Load Balancer 或 Network Load Balancer 的私有集成，请创建 HTTP 代理集成，指定要使用的 VPC 链接，并提供负载均衡器的侦听器 ARN。

使用以下命令创建私有集成，该私有集成通过 VPC 链接连接到负载均衡器。

```
aws apigatewayv2 create-integration --api-id api-id --integration-type HTTP_PROXY \
    --integration-method GET --connection-type VPC_LINK \
    --connection-id VPC-link-ID \
    --integration-uri arn:aws:elasticloadbalancing:us-east-2:123456789012:listener/app/
my-load-balancer/50dc6c495c0c9188/0467ef3c8400ae65
    --payload-format-version 1.0
```

使用 AWS Cloud Map 服务发现创建私有集成

在创建私有集成之前，您必须创建 VPC 链接。要了解有关 VPC 链接的更多信息，请参阅 为 HTTP API 使用 VPC 链接。

为了与 AWS Cloud Map 集成，API Gateway 使用 `DiscoverInstances` 来识别资源。您可以使用查询参数来定位特定资源。已注册资源的属性必须包括 IP 地址和端口。API Gateway 在从

`DiscoverInstances` 返回的运行状况良好的资源之间分发请求。如需了解详情，请参阅 AWS Cloud Map API 参考中的 DiscoverInstances。

> **Note**
>
> 如果您使用 Amazon ECS 填充 AWS Cloud Map 中的条目，则必须将您的 Amazon ECS 任务配置为通过 Amazon ECS 服务发现使用 SRV 记录，或者开启 Amazon ECS Service Connect。有关更多信息，请参阅《Amazon Elastic Container Service 开发人员指南》中互连服务。

要使用 AWS Cloud Map 创建私有集成，请创建 HTTP 代理集成，指定要使用的 VPC 链接，并提供 AWS Cloud Map 服务的 ARN。

使用以下命令创建私有集成，该私有集成通过 AWS Cloud Map 服务发现来标识资源。

```
aws apigatewayv2 create-integration --api-id api-id --integration-type HTTP_PROXY  \
    --integration-method GET --connection-type VPC_LINK \
    --connection-id VPC-link-ID \
    --integration-uri arn:aws:servicediscovery:us-east-2:123456789012:service/srv-id?
stage=prod&deployment=green_deployment
    --payload-format-version 1.0
```

为 HTTP API 使用 VPC 链接

通过 VPC 链接，您可以创建私有集成，将 HTTP API 路由连接到 VPC 中的私有资源，例如 Application Load Balancer 或基于 Amazon ECS 容器的应用程序。要了解有关创建私有集成的更多信息，请参阅 使用 HTTP API 的私有集成。

私有集成使用 VPC 链接来封装 API Gateway 与目标 VPC 资源之间的连接。您可以跨不同的路由和 API 重复使用 VPC 链接。

创建 VPC 链接时，API Gateway 在您的账户中为 VPC 链接创建和管理弹性网络接口。此过程可能耗时数分钟。当 VPC 链接可供使用时，其状态将从 `PENDING` 转换为 `AVAILABLE`。

> **Note**
>
> 如果 60 天内未通过 VPC 链接发送任何流量，其状态会变为 `INACTIVE`。当 VPC 链接处于 `INACTIVE` 状态时，API Gateway 删除 VPC 链接的所有网络接口。这会导致依赖于 VPC 链接

的 API 请求失败。如果 API 请求恢复，API Gateway 将重新预置网络接口。创建网络接口和重新激活 VPC 链接可能需要几分钟时间。您可以使用 VPC 链接状态来监控 VPC 链接的状态。

使用 AWS CLI 创建 VPC 链接

使用以下命令创建 VPC 链接。要创建 VPC 链接，涉及的所有资源必须由同一 AWS 账户拥有。

```
aws apigatewayv2 create-vpc-link --name MyVpcLink \
    --subnet-ids subnet-aaaa subnet-bbbb \
    --security-group-ids sg1234 sg5678
```

Note

VPC 链接是不可变的。创建 VPC 链接后，您无法更改其子网或安全组。

使用 AWS CLI 删除 VPC 链接

使用以下命令删除 VPC 链接。

```
aws apigatewayv2 delete-vpc-link --vpc-link-id abcd123
```

按区域的可用性

以下区域和可用区支持适用于 HTTP API 的 VPC 链接：

区域名称	区域	支持的可用区
美国东部（俄亥俄州）	us-east-2	use2-az1、use2-az2、use2-az3
美国东部（弗吉尼亚州北部）	us-east-1	use1-az1、use1-az2、use1-az4、use1-az5、use1-az6
美国西部（北加利福尼亚）	us-west-1	usw1-az1、usw1-az3

区域名称	区域	支持的可用区
美国西部（俄勒冈州）	us-west-2	usw2-az1、usw2-az2、usw2-az3、usw2-az4
亚太地区（香港）	ap-east-1	ape1-az2、ape1-az3
亚太地区（孟买）	ap-south-1	aps1-az1、aps1-az2、aps1-az3
Asia Pacific (Seoul)	ap-northeast-2	apne2-az1、apne2-az2、apne2-az3
亚太地区（新加坡）	ap-southeast-1	apse1-az1、apse1-az2、apse1-az3
亚太地区（悉尼）	ap-southeast-2	apse2-az1、apse2-az2、apse2-az3
亚太地区（东京)	ap-northeast-1	apne1-az1、apne1-az2、apne1-az4
加拿大（中部）	ca-central-1	cac1-az1、cac1-az2
欧洲（法兰克福）	eu-central-1	uc1-az1、eu1az2、eu1-az3
欧洲地区（爱尔兰）	eu-west-1	euw1-az1、euw1-az2、euw1-az3
欧洲（伦敦）	eu-west-2	euw2-az1、euw2-az2、euw2-az3
欧洲（巴黎）	eu-west-3	euw3-az1、euw3-az3
欧洲地区（斯德哥尔摩）	eu-north-1	eun1-az1、eun1-az2、eun1-az3
中东（巴林）	me-south-1	mes1-az1、mes1-az2、mes1-az3

区域名称	区域	支持的可用区
南美洲（圣保罗）	sa-east-1	sae1-az1、sae1-az2、sae1-az3
AWS GovCloud (美国西部)	us-gov-west-1	usgw1-az1、usgw1-az2、usgw1-az3

为 HTTP API 配置 CORS

跨源资源共享 (CORS) 是一项浏览器安全特征，该特征限制从在浏览器中运行的脚本启动的 HTTP 请求。如果您无法访问自己的 API 并收到包含 `Cross-Origin Request Blocked` 的错误消息，则可能需要启用 CORS。有关更多信息，请参阅什么是 CORS？。

通常需要 CORS 以构建 Web 应用程序来访问托管在不同域或源上的 API。您可以启用 CORS 以允许从托管在不同域上的 Web 应用程序发出对 API 的请求。例如，如果您的 API 托管在 `https://{api_id}.execute-api.{region}.amazonaws.com/` 上，并且您希望从托管在 `example.com` 上的 Web 应用程序调用 API，您的 API 必须支持 CORS。

如果您为 API 配置 CORS，即使没有为 API 配置 OPTIONS 路由，API Gateway 也会自动向预检 OPTIONS 请求发送响应。对于 CORS 请求，API Gateway 将已配置的 CORS 标头添加到来自集成的响应中。

> **Note**
>
> 如果您为 API 配置 CORS，则 API Gateway 忽略从后端集成返回的 CORS 标头。

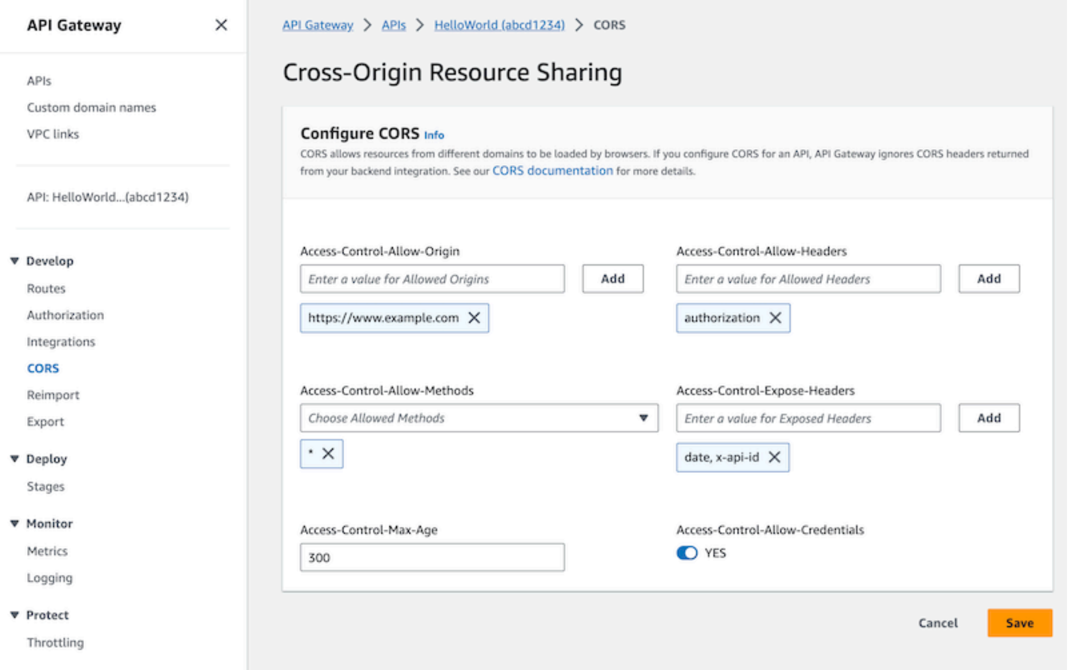
您可以在 CORS 配置中指定以下参数。要使用 API Gateway HTTP API 控制台添加这些参数，请在输入值后选择添加。

CORS 标头	CORS 配置属性	示例值
Access-Control-Allow-Origin	allowOrigins	• `https://www.example.com` • `*`（允许所有源）

CORS 标头	CORS 配置属性	示例值
		• `https://*` （允许以 https:// 开头的任何源） • `http://*`（允许以 http:// 开头的任何源）
Access-Control-Allow-Credentials	allowCredentials	true
Access-Control-Expose-Headers	exposeHeaders	Date、x-api-id*
Access-Control-Max-Age	maxAge	300
Access-Control-Allow-Methods	allowMethods	GET、POST、DELETE*
Access-Control-Allow-Headers	allowHeaders	Authorization*

要返回 CORS 标头，您的请求必须包含 `origin` 标头。

您的 CORS 配置可能类似以下内容：



Cross-Origin Resource Sharing

Configure CORS

CORS allows resources from different domains to be loaded by browsers. If you configure CORS for an API, API Gateway ignores CORS headers returned from your backend integration. See our CORS documentation for more details.

Access-Control-Allow-Origin: https://www.example.com

Access-Control-Allow-Headers: authorization

Access-Control-Allow-Methods: *

Access-Control-Expose-Headers: date, x-api-id

Access-Control-Max-Age: 300

Access-Control-Allow-Credentials: YES

使用 **`$default`** 路由和授权方为 HTTP API 配置 CORS

您可以为 HTTP API 的任何路由启用 CORS 并配置授权。当您为 `$default` 路由启用 CORS 和授权时，需要注意一些特殊事项。`$default` 路由捕获对所有尚未显式定义的方法和路由的请求，包括 `OPTIONS` 请求。要支持未经授权的 `OPTIONS` 请求，请向 API 添加一条不需要授权的 `OPTIONS /{proxy+}` 路由，并向该路由附加一个集成。`OPTIONS /{proxy+}` 路由的优先级高于 `$default` 路由。因此，它允许客户端在未经授权的情况下向您的 API 提交 `OPTIONS` 请求。有关路由优先级的更多信息，请参阅 路由 API 请求。

使用 AWS CLI 为 HTTP API 配置 CORS

您可以使用以下 update-api 命令从 `https://www.example.com` 启用 CORS 请求。

Example

```
aws apigatewayv2 update-api --api-id api-id --cors-configuration AllowOrigins="https://
www.example.com"
```

有关更多信息，请参阅 Amazon API Gateway 版本 2 API 参考中的 CORS。

转换 API 请求和响应

您可以在客户端 API 请求到达后端集成之前修改它们。您还可以在 API Gateway 将响应返回给客户端之前更改集成的响应。您可以使用参数映射修改 HTTP API 的 API 请求和响应。要使用参数映射，您需要指定要修改的 API 请求或响应参数，并指定如何修改这些参数。

转换 API 请求

在请求到达后端集成之前，您可以使用请求参数更改请求。您可以修改标头、查询字符串或请求路径。

请求参数是键值映射。键用于标识要更改的请求参数的位置以及如何更改它。值指定参数的新数据。

下表显示了支持的键。

参数映射键

类型	语法
标头	`append|overwrite|remove:header.` *`headername`*

类型	语法
查询字符串	`append|overwrite|remove:querystring.` *`querystring-name`*
路径	`overwrite:path`

下表显示了可以映射到参数的支持值。

请求参数映射值

类型	语法	备注
标头值	$request.header.*name* 或 ${request.header.*name*}	标头名称不区分大小写。API Gateway 将多个标头值与逗号组合在一起，例如 `"header1": "value1,value2"` 。保留一些标头。要了解更多信息，请参阅“[the section called “保留的标头”](#)”。
查询字符串值	$request.querystring.*name* 或 ${request.querystring.*name*}	查询字符串名称区分大小写。API Gateway 将多个值与逗号组合在一起，例如 `"querystring1" "Value1,Value2"` 。
请求正文	$request.body.*name* 或 ${request.body.*name*}	JSON 路径表达式 不支持递归下降 (`$request.body..name`) 和筛选表达式 (`?(expression)`)。 Note 当您指定 JSON 路径时，API Gateway 会在请求正文的 100 KB 处将其截断，然后应

类型	语法	备注
		用选择表达式。要发送大于 100 KB 的负载，请指定 `$request.body` 。
请求路径	$request.path 或 $ {request.path}	请求路径，没有阶段名称。
路径参数	$request.path.*name* 或 ${request.path.*name*}	请求中路径参数的值。例如，如果路由为 `/pets/{petId}` ，则可以从具有 `petId` 的请求中映射 `$request.path.petId` 参数。
上下文变量	$context.*variableName* 或 ${context.*variableName* }	上下文变量的值。 Note 仅支持特殊字符 `.` 和 `_`。
阶段变量	$stageVariables.*variableName* 或 ${stageVariables.*variableName* }	阶段变量的值。
静态值	*string*	常量值。

> Note
>
> 要在选择表达式中使用多个变量，请将该变量括在括号中。例如，`${request.path.name} ${request.path.id}`。

转换 API 响应

在将响应返回给客户端之前，您可以使用响应参数转换来自后端集成的 HTTP 响应。在 API Gateway 将响应返回给客户端之前，您可以修改响应的标头或状态码。

您可以为集成返回的每个状态代码配置响应参数。响应参数是键值映射。键用于标识要更改的请求参数的位置以及如何更改它。值指定参数的新数据。

下表显示了支持的键。

响应参数映射键

类型	语法
标头	`append|overwrite|remove:header.` *`headername`*
状态代码	`overwrite:statuscode`

下表显示了可以映射到参数的支持值。

响应参数映射值

类型	语法	备注
标头值	$response.header.*name* 或 ${response.header.*name*}	标头名称不区分大小写。API Gateway 将多个标头值与逗号组合在一起，例如 `"header1": "value1,value2"`。保留一些标头。要了解更多信息，请参阅“the section called “保留的标头””。
响应正文	$response.body.*name* 或 ${response.body.*name*}	JSON 路径表达式。不支持递归下降 (`$response.body..name`) 和筛选表达式 (`?(expression)`)。

类型	语法	备注
		Note 当您指定 JSON 路径时，API Gateway 会在响应正文的 100 KB 处将其截断，然后应用选择表达式。要发送大于 100 KB 的负载，请指定 `$response.body` 。
上下文变量	$context.*variableName* 或 ${context.*variableName* }	受支持的上下文变量的值。
阶段变量	$stageVariables.*variableName* 或 ${stageVariables.*variableName* }	阶段变量的值。
静态值	*string*	常量值。

> Note
>
> 要在选择表达式中使用多个变量，请将该变量括在括号中。例如，`${request.path.name} ${request.path.id}`。

保留的标头

保留以下标头。您无法为这些标头配置请求或响应映射。

- access-control-*
- apigw-*
- 授权
- Connection

- Content-Encoding
- 内容长度
- Content-Location
- 已转发
- Keep-Alive
- Origin
- Proxy-Authenticate
- Proxy-Authorization
- TE
- Trailers
- Transfer-Encoding
- 升级
- x-amz-*
- x-amzn-*
- X-Forwarded-For
- X-Forwarded-Host
- X-Forwarded-Proto
- Via

示例

以下 AWS CLI 示例会配置参数映射。如需示例 AWS CloudFormation 模板，请参阅 GitHub。

向 API 请求添加标头

以下示例在 API 请求到达后端集成之前将名为 `header1` 的标头添加到该请求中。API Gateway 使用请求 ID 填充标头。

```
aws apigatewayv2 create-integration \
    --api-id abcdef123 \
    --integration-type HTTP_PROXY \
    --payload-format-version 1.0 \
```

```
    --integration-uri 'https://api.example.com' \
    --integration-method ANY \
    --request-parameters '{ "append:header.header1": "$context.requestId" }'
```

重命名请求标头

以下示例将请求标头从 `header1` 重命名为 `header2`。

```
aws apigatewayv2 create-integration \
    --api-id abcdef123 \
    --integration-type HTTP_PROXY \
    --payload-format-version 1.0 \
    --integration-uri 'https://api.example.com' \
    --integration-method ANY \
    --request-parameters '{ "append:header.header2": "$request.header.header1",
 "remove:header.header1": "''"}'
```

更改集成的响应

以下示例为集成配置响应参数。当集成返回 500 状态码时，API Gateway 会将状态码更改为 403，然后在响应中添加 `header11`。当集成返回 404 状态码时，API Gateway 会向响应添加 `error` 标头。

```
aws apigatewayv2 create-integration \
    --api-id abcdef123 \
    --integration-type HTTP_PROXY \
    --payload-format-version 1.0 \
    --integration-uri 'https://api.example.com' \
    --integration-method ANY \
    --response-parameters '{"500" : {"append:header.header1": "$context.requestId",
 "overwrite:statuscode" : "403"}, "404" : {"append:header.error" :
 "$stageVariables.environmentId"}  }'
```

删除配置的参数映射

以下示例命令删除以前为 `append:header.header1` 配置的请求参数。它还删除了先前为 200 状态码配置的响应参数。

```
aws apigatewayv2 update-integration \
    --api-id abcdef123 \
    --integration-id hijk456 \
    --request-parameters '{"append:header.header1" : ""}' \
```

```
    --response-parameters '{"200" : {}}'
```

将 OpenAPI 定义用于 HTTP API

您可以使用 OpenAPI 3.0 定义文件来定义您的 HTTP API。然后，您可以将定义导入 API Gateway 中以创建 API。要了解有关 OpenAPI 的 API Gateway 扩展的更多信息，请参阅 OpenAPI 扩展。

导入 HTTP API

您可以通过导入 OpenAPI 3.0 定义文件来创建 HTTP API。

要从 REST API 迁移到 HTTP API，您可以将 REST API 导出为 OpenAPI 3.0 定义文件。然后，将 API 定义导入为 HTTP API。要了解有关导出 REST API 的更多信息，请参阅 从 API Gateway 导出 REST API。

> **Note**
>
> HTTP API 与 REST API 支持相同的AWS变量。要了解更多信息，请参阅“用于 OpenAPI 导入的 AWS 变量”。

导入验证信息

导入 API 时，API Gateway 提供三类验证信息。

Info

根据 OpenAPI 规范，属性是有效的，但对于 HTTP API 不支持该属性。

例如，以下 OpenAPI 3.0 代码段生成有关导入的信息，因为 HTTP API 不支持请求验证。API Gateway 忽略 `requestBody` 和 `schema` 字段。

```
"paths": {
  "/": {
    "get": {
      "x-amazon-apigateway-integration": {
        "type": "AWS_PROXY",
        "httpMethod": "POST",
        "uri": "arn:aws:lambda:us-east-2:123456789012:function:HelloWorld",
        "payloadFormatVersion": "1.0"
```

```
      },
      "requestBody": {
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Body"
            }
          }
        }
      }
    }
  }
  ...
},
"components": {
  "schemas": {
    "Body": {
      "type": "object",
      "properties": {
        "key": {
          "type": "string"
        }
      }
    }
    ...
  }
  ...
}
```

警告

根据 OpenAPI 规范，属性或结构是无效的，但它不会阻止 API 创建。您可以指定 API Gateway 是应忽略这些警告并继续创建 API，还是在出现警告时停止创建 API。

以下 OpenAPI 3.0 文档在导入时生成警告，因为 HTTP API 只支持 Lambda 代理和 HTTP 代理集成。

```
"x-amazon-apigateway-integration": {
  "type": "AWS",
  "httpMethod": "POST",
  "uri": "arn:aws:lambda:us-east-2:123456789012:function:HelloWorld",
  "payloadFormatVersion": "1.0"
}
```

错误

OpenAPI 规范无效或格式错误。API Gateway 无法从格式错误的文档创建任何资源。您必须修复错误，然后重试。

以下 API 定义会在导入时产生错误，因为 HTTP API 只支持 OpenAPI 3.0 规范。

```
{
  "swagger": "2.0.0",
  "info": {
    "title": "My API",
    "description": "An Example OpenAPI definition for Errors/Warnings/ImportInfo",
    "version": "1.0"
  }
  ...
}
```

在另一个示例中，虽然 OpenAPI 允许用户定义一个具有与特定操作相关的多个安全要求的 API，但 API Gateway 不支持这一点。每个操作只能有一个 IAM 授权、一个 Lambda 授权者或一个 JWT 授权者。尝试对多个安全要求进行建模会导致错误。

使用 AWS CLI 导入 API

以下命令将 OpenAPI 3.0 定义文件 `api-definition.json` 导入为 HTTP API。

Example

```
aws apigatewayv2 import-api --body file://api-definition.json
```

Example

您可以导入以下示例 OpenAPI 3.0 定义来创建 HTTP API。

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "Example Pet Store",
    "description": "A Pet Store API.",
    "version": "1.0"
  },
  "paths": {
    "/pets": {
```

```
      "get": {
        "operationId": "GET HTTP",
        "parameters": [
          {
            "name": "type",
            "in": "query",
            "schema": {
              "type": "string"
            }
          },
          {
            "name": "page",
            "in": "query",
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "headers": {
              "Access-Control-Allow-Origin": {
                "schema": {
                  "type": "string"
                }
              }
            },
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Pets"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "HTTP_PROXY",
          "httpMethod": "GET",
          "uri": "http://petstore.execute-api.us-west-1.amazonaws.com/petstore/pets",
          "payloadFormatVersion": 1.0
        }
      },
```

```
      "post": {
        "operationId": "Create Pet",
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/NewPet"
              }
            }
          },
          "required": true
        },
        "responses": {
          "200": {
            "description": "200 response",
            "headers": {
              "Access-Control-Allow-Origin": {
                "schema": {
                  "type": "string"
                }
              }
            },
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/NewPetResponse"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "HTTP_PROXY",
          "httpMethod": "POST",
          "uri": "http://petstore.execute-api.us-west-1.amazonaws.com/petstore/pets",
          "payloadFormatVersion": 1.0
        }
      }
    },
    "/pets/{petId}": {
      "get": {
        "operationId": "Get Pet",
        "parameters": [
          {
```

```
            "name": "petId",
            "in": "path",
            "required": true,
            "schema": {
              "type": "string"
            }
          }
        ],
        "responses": {
          "200": {
            "description": "200 response",
            "headers": {
              "Access-Control-Allow-Origin": {
                "schema": {
                  "type": "string"
                }
              }
            },
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Pet"
                }
              }
            }
          }
        },
        "x-amazon-apigateway-integration": {
          "type": "HTTP_PROXY",
          "httpMethod": "GET",
          "uri": "http://petstore.execute-api.us-west-1.amazonaws.com/petstore/pets/
{petId}",
          "payloadFormatVersion": 1.0
        }
      }
    }
  },
  "x-amazon-apigateway-cors": {
    "allowOrigins": [
      "*"
    ],
    "allowMethods": [
      "GET",
      "OPTIONS",
```

```
      "POST"
    ],
    "allowHeaders": [
      "x-amzm-header",
      "x-apigateway-header",
      "x-api-key",
      "authorization",
      "x-amz-date",
      "content-type"
    ]
  },
  "components": {
    "schemas": {
      "Pets": {
        "type": "array",
        "items": {
          "$ref": "#/components/schemas/Pet"
        }
      },
      "Empty": {
        "type": "object"
      },
      "NewPetResponse": {
        "type": "object",
        "properties": {
          "pet": {
            "$ref": "#/components/schemas/Pet"
          },
          "message": {
            "type": "string"
          }
        }
      },
      "Pet": {
        "type": "object",
        "properties": {
          "id": {
            "type": "string"
          },
          "type": {
            "type": "string"
          },
          "price": {
            "type": "number"
```

```
          }
        }
      },
      "NewPet": {
        "type": "object",
        "properties": {
          "type": {
            "$ref": "#/components/schemas/PetType"
          },
          "price": {
            "type": "number"
          }
        }
      },
      "PetType": {
        "type": "string",
        "enum": [
          "dog",
          "cat",
          "fish",
          "bird",
          "gecko"
        ]
      }
    }
  }
}
```

从 API Gateway 导出 HTTP API

创建 HTTP API 后，您可以从 API Gateway 导出 API 的 OpenAPI 3.0 定义。您可以选择要导出的阶段，也可以导出 API 的最新配置。还可以将导出的 API 定义导入到 API Gateway 中，以创建另一个相同的 API。要了解有关导入 API 定义的更多信息，请参阅 导入 HTTP API。

使用AWS CLI 导出阶段的 OpenAPI 3.0 定义

以下命令将名为 `prod` 的 API 阶段的 OpenAPI 定义导出到名为 `stage-definition.yaml` 的 YAML 文件。默认情况下，导出的定义文件包含 API Gateway 扩展名。

```
aws apigatewayv2 export-api \
    --api-id api-id  \
    --output-type YAML  \
    --specification OAS30 \
```

```
    --stage-name prod \
    stage-definition.yaml
```

使用AWS CLI 导出 API 的最新更改的 OpenAPI 3.0 定义

以下命令将 HTTP API 的 OpenAPI 定义导出到名为 `latest-api-definition.json` 的 JSON 文件。由于命令未指定阶段，因此 API Gateway 导出 API 的最新配置，无论它是否已部署到阶段。导出的定义文件不包含 API Gateway 扩展名。

```
aws apigatewayv2 export-api \
    --api-id api-id  \
    --output-type JSON  \
    --specification OAS30 \
    --no-include-extensions \
    latest-api-definition.json
```

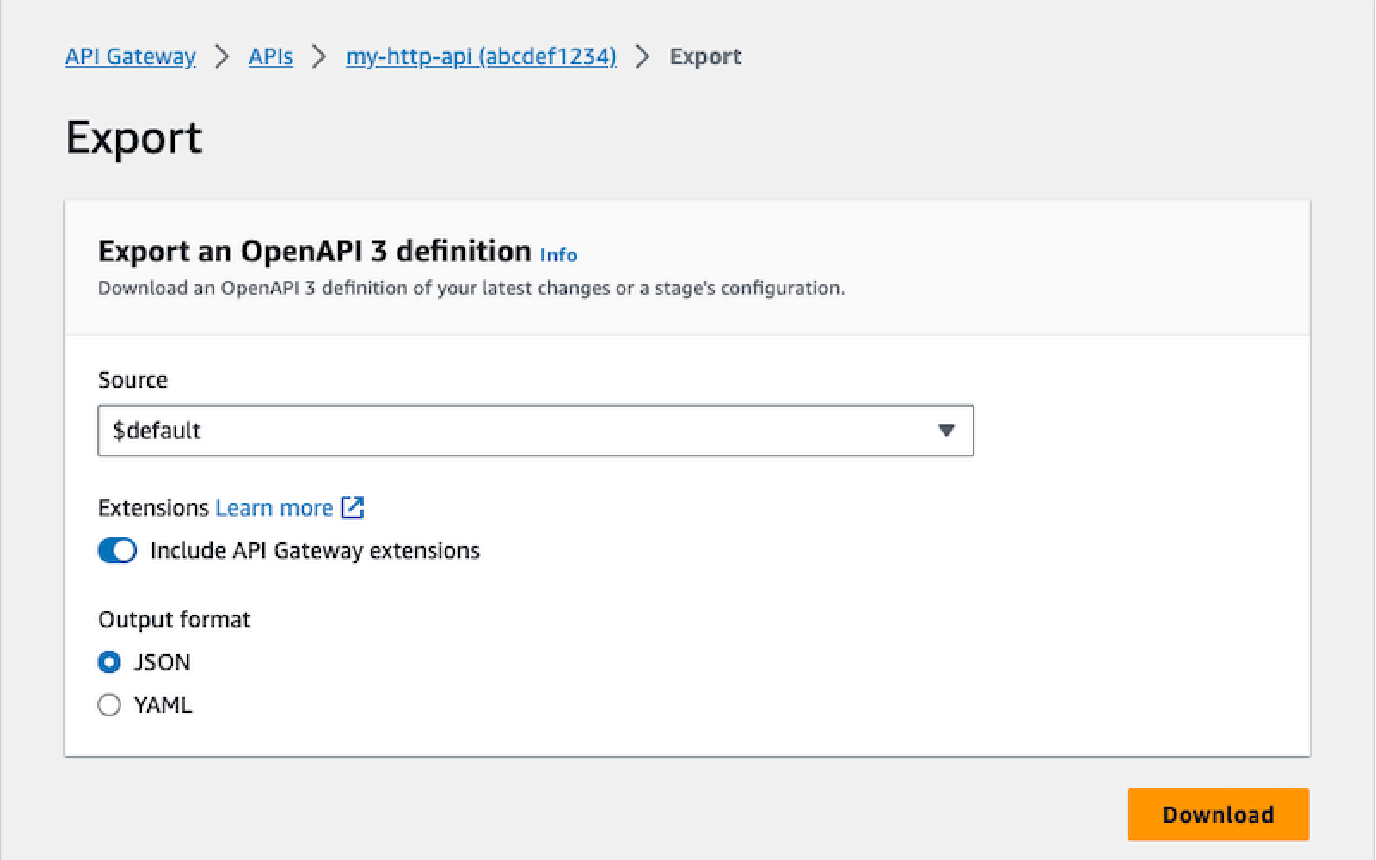
有关更多信息，请参阅 Amazon API Gateway 版本 2 API 参考 中的 ExportAPI。

使用 API Gateway 控制台导出 OpenAPI 3.0 定义

以下过程显示了如何导出 HTTP API 的 OpenAPI 定义。

使用 API Gateway 控制台导出 OpenAPI 3.0 定义

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 HTTP API。
3. 在主导航窗格的开发下，选择导出。
4. 在以下选项中选择，以导出您的 API：



Export

Export an OpenAPI 3 definition

Download an OpenAPI 3 definition of your latest changes or a stage's configuration.

Source

$default

Extensions

● Include API Gateway extensions

Output format

● JSON

○ YAML

   a. 在来源中，选择 OpenAPI 3.0 定义的来源。您可以选择要导出的阶段，也可以导出 API 的最新配置。
   b. 打开包括 API Gateway 扩展以包含 API 网关扩展。
   c. 在输出格式中，选择一种输出格式。

5. 选择下载。

发布 HTTP API 以供客户调用

您可以使用阶段和自定义域名发布 API 以供客户端调用。

一个 API 阶段是对您 API 生命周期状态（例如，`dev`、`prod`、`beta` 或 `v2`）的一次逻辑引用。每个阶段都是一个对 API 部署的命名引用，可供客户端应用程序调用。您可以为 API 的每个阶段配置不同的集成和设置。

您可以使用自定义域名来提供比默认 URL `https://api-id.execute-api.region.amazonaws.com/stage` 更简单、更直观的 URL，以供客户端调用 API。

Note

为了增强您的 API Gateway API 的安全性，将在公共后缀列表 (PSL) 中注册 `execute-api.{region}.amazonaws.com` 域。为进一步增强安全性，如果您需要在 API Gateway API 的默认域名中设置敏感 Cookie，我们建议您使用带 `__Host-` 前缀的 Cookie。这将有助于保护您的域，防范跨站点请求伪造 (CSRF) 攻击。要了解更多信息，请参阅 Mozilla 开发者网络中的 Set-Cookie 页面。

主题

- 使用 HTTP API 的阶段
- HTTP API 的安全策略
- 为 HTTP API 设置自定义域名

使用 HTTP API 的阶段

一个 API 阶段是对您 API 生命周期状态（例如，`dev`、`prod`、`beta` 或 `v2`）的一次逻辑引用。API 阶段通过 API ID 和阶段名称标识，包含在您用于调用 API 的 URL 中。每个阶段都是一个对 API 部署的命名引用，可供客户端应用程序调用。

您可以创建一个 `$default` 阶段，该阶段可以从 API 的 URL 的基本部分得出 — 例如 `https://{api_id}.execute-api.{region}.amazonaws.com/`。您可以使用此 URL 调用 API 阶段。

部署是 API 配置的快照。将 API 部署到阶段后，客户端可以调用该 API。您必须部署 API 才能使更改生效。如果启用自动部署，则会自动为您发布对 API 的更改。

阶段变量

阶段变量是您可以为 HTTP API 的阶段定义的键/值对。它们与环境变量的功能类似，可用于 API 设置。

例如，您可以定义阶段变量，然后将其值设置为某个 HTTP 代理集成的 HTTP 终端节点。稍后，您可以使用关联的阶段变量名称引用终端节点。通过执行此操作，您可以在各个阶段对不同终端节点使用相同的 API 设置。同样，您可以使用阶段变量，为 API 的各个阶段指定不同的 AWS Lambda 函数集成。

Note

阶段变量不适用于敏感数据，例如凭证。要将敏感数据传递给集成，请使用 AWS Lambda 授权方。您可以在 Lambda 授权方的输出中将敏感数据传递给集成。要了解更多信息，请参阅“the section called “Lambda 授权方响应格式””。

示例

要使用阶段变量自定义 HTTP 集成终端节点，您必须首先设置阶段变量的名称和值（例如 `url`，值为 `example.com`）。接下来，设置 HTTP 代理集成。您可以告知 API Gateway 使用阶段变量值 **`http://${stageVariables.url}`**，而不是输入终端节点的 URL。此值将指示 API Gateway 在运行时替换您的阶段变量 `${}`，具体取决于 API 所处的阶段。

您可以通过类似的方式引用阶段变量，用于指定 Lambda 函数名称或AWS角色 ARN。

将 Lambda 函数名称指定为阶段变量值时，您必须手动配置对 Lambda 函数的权限。您可以使用 AWS Command Line Interface (AWS CLI) 来执行此操作。

```
aws lambda add-permission --function-name arn:aws:lambda:XXXXXX:your-lambda-function-
name --source-arn arn:aws:execute-api:us-east-1:YOUR_ACCOUNT_ID:api_id/*/HTTP_METHOD/
resource --principal apigateway.amazonaws.com --statement-id apigateway-access --action
 lambda:InvokeFunction
```

API Gateway 阶段变量参考

HTTP 集成 URI

您可将阶段变量用作 HTTP 集成 URI 的一部分，如以下示例所示。

- 不带协议的完整 URI – `http://${stageVariables.<variable_name>}`
- 完整域 – `http://${stageVariables.<variable_name>}/resource/operation`
- 子域 – `http://${stageVariables.<variable_name>}.example.com/resource/operation`
- 路径 – `http://example.com/${stageVariables.<variable_name>}/bar`
- 查询字符串 – `http://example.com/foo?q=${stageVariables.<variable_name>}`

Lambda 函数

您可以使用阶段变量代替 Lambda 函数集成名称或别名，如以下示例所示。

- `arn:aws:apigateway:<region>:lambda:path/2015-03-31/functions/arn:aws:lambda:<region>:<account_id>:function:${stageVariables.<function_variable_name>}/invocations`
- `arn:aws:apigateway:<region>:lambda:path/2015-03-31/functions/arn:aws:lambda:<region>:<account_id>:function:<function_name>:${stageVariables.<version_variable_name>}/invocations`

> **Note**
>
> 要将阶段变量用于 Lambda 函数，该函数必须与 API 位于同一账户中。阶段变量不支持跨账户 Lambda 函数。

AWS 集成凭证

您可以在 AWS 用户或角色凭证 ARN 中使用阶段变量，如以下示例所示。

- `arn:aws:iam::<account_id>:${stageVariables.<variable_name>}`

HTTP API 的安全策略

API Gateway 对所有 HTTP API 端点强制执行 TLS_1_2 的安全策略。

安全策略 是 Amazon API Gateway 提供的最低 TLS 版本和密码套件的预定义组合。TLS 协议解决了网络安全问题，例如客户端和服务器之间的篡改和窃听。当您的客户端通过自定义域建立与您 API 的 TLS 握手时，安全策略实施客户端可以选择使用的 TLS 版本和密码套件选项。此安全策略接受 TLS 1.2 和 TLS 1.3 流量并拒绝 TLS 1.0 流量。

HTTP API 支持的 TLS 协议和密码

下表描述了 HTTP API 支持的 TLS 协议和密码。

安全策略	TLS_1_2
TLS 协议	
TLSv1.3	♦
TLSv1.2	♦
TLS 密码	
TLS-AES-128-GCM-SHA256	♦
TLS-AES-256-GCM-SHA384	♦
TLS-CHACHA20-POLY1305-SHA256	♦
ECDHE-ECDSA-AES128- GCM-SHA256	♦
ECDHE-RSA-AES128- GCM-SHA256	♦
ECDHE-ECDSA-AES128-SHA256	♦
ECDHE-RSA-AES128-SHA256	♦
ECDHE-ECDSA-AES256- GCM-SHA384	♦
ECDHE-RSA-AES256- GCM-SHA384	♦
ECDHE-ECDSA-AES256-SHA384	♦
ECDHE-RSA-AES256-SHA384	♦
AES128-GCM-SHA256	♦
AES128-SHA256	♦
AES256-GCM-SHA384	♦
AES256-SHA256	♦

OpenSSL 和 RFC 密码名称

OpenSSL 和 IETF RFC 5246 为相同的密码使用不同的名称。有关密码名称的列表，请参阅the section called “OpenSSL 和 RFC 密码名称”。

有关 REST API 和 WebSocket API 的信息

有关 REST API 和 WebSocket API 的更多信息，请参阅the section called “选择安全策略”和the section called “WebSocket API 的安全策略”。

为 HTTP API 设置自定义域名

自定义域名 是您可以提供给 API 用户的更简单、更直观的 URL。

部署 API 后，您（和您的客户）可以使用以下格式的默认基本 URL 调用 API：

```
https://api-id.execute-api.region.amazonaws.com/stage
```

其中 `api-id` 由 API Gateway 生成，`region`（AWS 区域）由您在创建 API 时指定，`stage` 由您在部署 API 时指定。

URL 的主机名部分（即 `api-id.execute-api.region.amazonaws.com`）是指 API 端点。默认 API 端点难于重新调用，对用户不友好。

使用自定义域名，您可以设置 API 的主机名，并选择基本路径（例如 `myservice`）以将备用 URL 映射到 API。例如，一个更为用户友好的 API 基本 URL 可以变成：

```
https://api.example.com/myservice
```

Note

自定义域可以与 REST API 和 HTTP API 相关联。您可以使用 API Gateway 版本 2 API 创建和管理 REST API 和 HTTP API 的区域自定义域名。
对于 HTTP API，TLS 1.2 是唯一受支持的 TLS 版本。

注册域名

您必须拥有已注册的 Internet 域名，以便为 API 设置自定义域名。域名必须遵循 RFC 1035 规范，每个标签最多可以有 63 个八位字节，总共可以有 255 个八位字节。如果需要，您可以使用 Amazon

Route 53 或使用您选择的第三方域注册商注册互联网域。API 的自定义域名可以是已注册 Internet 域的子域或根域（也称为“顶级域”）的名称。

在 API Gateway 中创建自定义域名后，您必须创建或更新 DNS 提供商的资源记录以映射到 API 端点。如果没有此类映射，针对自定义域名的 API 请求无法到达 API Gateway。

区域自定义域名

为区域 API 创建自定义域名时，API Gateway 为 API 创建区域域名。您必须设置将自定义域名映射到区域域名的 DNS 记录。您还必须为自定义域名提供证书。

通配符自定义域名

使用通配符自定义域名，您可以在不超过默认配额的情况下支持几乎无限数量的域名。例如，您可以为每位客户提供自己的域名 `*customername*.api.example.com`。

要创建通配符自定义域名，可以指定通配符 (*) 作为表示根域所有可能子域的自定义域的第一个子域。

例如，通配符自定义域名 `*.example.com` 会生成子域，如 `a.example.com`、`b.example.com` 和 `c.example.com`，这些子域都会路由到同一个域。

通配符自定义域名支持与 API Gateway 的标准自定义域名不同的配置。例如，在单个 AWS 账户中，您可以对 `*.example.com` 和 `a.example.com` 进行不同的配置。

要创建通配符自定义域名，您必须提供已使用 DNS 或电子邮件验证方法验证的由 ACM 颁发的证书。

> **Note**
>
> 如果其他 AWS 账户已经创建了与通配符自定义域名冲突的自定义域名，则无法创建通配符自定义域名。例如，如果账户 A 已经创建了 `a.example.com`，则账户 B 无法创建通配符自定义域名 `*.example.com`。
> 如果账户 A 和账户 B 共享拥有者，您可以联系 AWS Support 中心请求例外。

自定义域名的证书

> **Important**
>
> 您为您的自定义域名配置了证书。如果您的应用程序使用证书固定（有时称为 SSL 固定）来固定 ACM 证书，则在 AWS 续订证书后，应用程序可能无法连接到您的域。有关更多信息，请参阅《AWS Certificate Manager 用户指南》中的证书固定问题。

要为支持 ACM 的区域中的自定义域名提供证书，您必须从 ACM 请求证书。要为不支持 ACM 的区域中的区域自定义域名提供证书，您必须在该区域中将证书导入到 API Gateway。

要导入 SSL/TLS 证书，您必须针对自定义域名提供 PEM 格式的 SSL/TLS 证书文本、其私有密钥和证书链。存储在 ACM 中的每个证书均由其 ARN 标识。要针对域名使用 AWS 托管的证书，您只需参考其 ARN 即可。

通过 ACM 可以轻松地为 API 设置和使用自定义域名。您可以为给定的域名创建证书（或导入证书），使用 ACM 提供的证书的 ARN 在 API Gateway 中设置域名，然后将自定义域名下的基本路径映射到 API 的已部署阶段。如果拥有 ACM 颁发的证书，那么您就无需担心公开任何敏感的证书详细信息，如私有密钥。

有关设置自定义域名的详细信息，请参阅在 中准备好证书AWS Certificate Manager和在 API Gateway 中设置区域自定义域名。

对 HTTP API 使用 API 映射

您可以使用 API 映射将 API 阶段连接到自定义域名。创建域名并配置 DNS 记录后，您可以使用 API 映射通过自定义域名向 API 发送流量。

API 映射指定了用于映射的 API、阶段以及可选的路径。例如，您可以将 API 的 `production` 阶段映射到 `https://api.example.com/orders`。

您可以将 HTTP 和 REST API 阶段映射到相同的自定义域名。

在创建 API 映射之前，您必须拥有 API、阶段和自定义域名。要了解有关创建自定义域名的更多信息，请参阅 the section called “设置区域自定义域名”。

路由 API 请求

您可以使用多个级别配置 API 映射，例如 `orders/v1/items` 和 `orders/v2/items`。

对于具有多个级别的 API 映射，API Gateway 将请求路由到匹配路径最长的 API 映射。选择要调用的 API 时，API Gateway 仅考虑为 API 映射配置的路径，而不考虑 API 路由。如果没有与请求匹配的路径，API Gateway 会将请求发送到已映射到空路径 (none) 的 API。

对于使用具有多个级别的 API 映射的自定义域名，API Gateway 将请求路由到匹配前缀最长的 API 映射。

例如，考虑使用以下 API 映射的自定义域名 `https://api.example.com`：

1. (none) 映射到 API 1。
2. `orders` 映射到 API 2。
3. `orders/v1/items` 映射到 API 3。
4. `orders/v2/items` 映射到 API 4。
5. `orders/v2/items/categories` 映射到 API 5。

请求	选定的 API	说明
`https://api.example.com/orders`	`API 2`	请求完全匹配此 API 映射。
`https://api.example.com/orders/v1/items`	`API 3`	请求完全匹配此 API 映射。
`https://api.example.com/orders/v2/items`	`API 4`	请求完全匹配此 API 映射。
`https://api.example.com/orders/v1/items/123`	`API 3`	API Gateway 选择匹配路径最长的映射。请求结尾处的 `123` 不影响选择。
`https://api.example.com/orders/v2/items/categories/5`	`API 5`	API Gateway 选择匹配路径最长的映射。
`https://api.example.com/customers`	`API 1`	API Gateway 使用空映射作为“捕获全部”。
`https://api.example.com/ordersandmore`	`API 2`	API Gateway 选择匹配前缀最长的映射。对于配置了单级映射的自定义域名（例如，仅 `https://api.example.com/orders` 和 `https://api.example.com/` ），API Gateway

请求	选定的 API	说明
		会选择 API 1，因为没有与 `ordersandmore` 匹配的路径。

限制

- 在 API 映射中，自定义域名和映射的 API 必须位于同一个AWS账户中。
- API 映射必须仅包含字母、数字和以下字符：`$-_.+!*'()/`。
- API 映射中路径的最大长度为 300 个字符。
- 每个域名可以有 200 个具有多个级别的 API 映射。
- 您只能使用 TLS 1.2 安全策略将 HTTP API 映射到区域自定义域名。
- 您不能将 WebSocket API 映射到与 HTTP API 或 REST API 相同的自定义域名。

创建 API 映射

要创建 API 映射，您必须首先创建自定义域名、API 和阶段。有关使用自定义域名的更多信息，请参阅 the section called “设置区域自定义域名”。

例如，创建所有资源的 AWS Serverless Application Model 模板，请参阅 GitHub 上的使用 SAM 的会话。

AWS Management Console

创建 API 映射

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择自定义域名。
3. 选择您已经创建的自定义域名。
4. 选择 API 映射。
5. 选择 Configure API mappings (配置 API 映射)。
6. 选择 Add new mapping (添加新映射)。
7. 输入 API、阶段以及可选的路径。
8. 选择 Save。

AWS CLI

以下 AWS CLI 命令创建一个 API 映射。在此示例中，API Gateway 将请求发送到 `api.example.com/v1/orders`，到指定的 API 和阶段。

```
aws apigatewayv2 create-api-mapping \
    --domain-name api.example.com \
    --api-mapping-key v1/orders \
    --api-id a1b2c3d4 \
    --stage test
```

AWS CloudFormation

以下 AWS CloudFormation 示例会创建一个 API 映射。

```
MyApiMapping:
  Type: 'AWS::ApiGatewayV2::ApiMapping'
  Properties:
    DomainName: api.example.com
    ApiMappingKey: 'orders/v2/items'
    ApiId: !Ref MyApi
    Stage: !Ref MyStage
```

禁用 HTTP API 的默认终端节点

默认情况下，客户端可以通过使用 API Gateway 为 API 生成的 `execute-api` 终端节点来调用您的 API。为确保客户端只能通过使用自定义域名访问您的 API，请禁用默认 `execute-api` 终端节点。

> **Note**
>
> 禁用默认终端节点时，它会影响 API 的所有阶段。

以下 AWS CLI 命令会禁用 HTTP API 的默认终端节点。

```
aws apigatewayv2 update-api \
    --api-id abcdef123 \
    --disable-execute-api-endpoint
```

禁用默认终端节点后，除非启用了自动部署，否则必须部署 API 才能使更改生效。

以下 AWS CLI 命令会创建部署。

```
aws apigatewayv2 create-deployment \
    --api-id abcdef123 \
    --stage-name dev
```

保护您的 HTTP API

API Gateway 提供了多种方法来保护您的 API 免受某些威胁，例如恶意用户或流量高峰。您可以使用诸如设置目标和启用双向 TLS 等策略来保护您的 API。在本节中，您可以了解如何使用 API Gateway 启用这些功能。

主题

- 限制发送给 HTTP API 的请求
- 为 HTTP API 配置双向 TLS 身份验证

限制发送给 HTTP API 的请求

您可以为 API 配置节流，以帮助防止它们因请求过多而不堪重负。节流是在尽最大努力的基础上应用的，应被视为目标而不是保证的请求上限。

API Gateway 使用令牌桶算法（其中，一个令牌即一个请求）限制对 API 的请求。具体来说，API Gateway 根据您账户中的所有 API，按区域检查请求提交的速率和突发事件。在令牌桶算法中，突发可以允许这些限制的预定义超出，但在某些情况下，其他因素也可能导致限制超支。

如果请求提交超过稳态请求速率和突增限制，则 API Gateway 将开始限制请求。此时客户可能会收到 `429 Too Many Requests` 个错误响应。捕获此类异常后，客户端能够以限制速率的方式重新提交失败的请求。

作为 API 开发人员，您可以针对各个 API 阶段或方法设置目标限制，以提高账户中所有 API 的整体性能。

每个区域的账户级别限制

默认情况下，API Gateway 针对每个区域限制 AWS 账户内所有 API 的每秒稳态请求 (RPS)。它还对于每个区域限制一个 AWS 账户中所有 API 的突增（即最大存储桶大小）。在 API Gateway 中，突增限制代表 API Gateway 在返回 `429 Too Many Requests` 错误响应之前可以完成目标的最大并发请求提交数量。有关限制配额的更多信息，请参阅配额和重要提示。

每个账户限制适用于指定区域内账户中的所有 API。客户可以请求我们放宽账户级别的速率限制 —— 如果具有更短的超时和较小的有效负载的 API，则可以提高限制。要请求增加每个区域的账户级别限制，请联系 AWS Support 中心。有关更多信息，请参阅 配额和重要提示。请注意，这些限制不能高于 AWS 节流限制。

路由级别限制

您可以设置路由级别限制，用于覆盖 API 中特定阶段或各个路由的账户级别请求限制。原定设置的路由节流限制不能超过账户级别的费率限制。

您可以使用 AWS CLI 配置路由级限制。以下命令为 API 的指定阶段和路由配置自定义限制。

```
aws apigatewayv2 update-stage \
    --api-id a1b2c3d4 \
    --stage-name dev \
    --route-settings '{"GET /pets":
{"ThrottlingBurstLimit":100,"ThrottlingRateLimit":2000}}'
```

为 HTTP API 配置双向 TLS 身份验证

双向 TLS 身份验证要求在客户端和服务器之间进行双向身份验证。使用双向 TLS，客户端必须提供 X.509 证书来验证其身份才能访问您的 API。双向 TLS 是物联网 (IoT) 和企业对企业应用程序的常见要求。

您可以使用双向 TLS 以及 API Gateway 支持的其他授权和身份验证操作。API Gateway 将客户端提供的证书转发给 Lambda 授权方和后端集成。

> ⚠ Important
>
> 默认情况下，客户端可以通过使用 API Gateway 为 API 生成的 `execute-api` 端点来调用您的 API。要确保客户端只能通过使用具有双向 TLS 的自定义域名访问您的 API，请禁用默认 `execute-api` 端点。要了解更多信息，请参阅“the section called “禁用默认终端节点””。

使用双向 TLS 的先决条件

要配置双向 TLS，您需要：

- 自定义域名
- 至少在 AWS Certificate Manager 中为您的自定义域名配置了一个证书

- 已配置信任存储库并上载到 Amazon S3

自定义域名

要为 HTTP API 启用双向 TLS，您必须为 API 配置自定义域名。您可以为自定义域名启用双向 TLS，然后向客户端提供自定义域名。要使用启用了双向 TLS 的自定义域名访问 API，客户端必须提供您在 API 请求中信任的证书。您可以在 the section called “自定义域名” 中找到更多信息。

使用 AWS Certificate Manager 颁发的证书

您可以直接从 ACM 请求公开可信的证书，也可以导入公开证书或自签名证书。要在 ACM 中设置证书，请前往 ACM。如果要导入证书，请继续阅读以下部分中的内容。

使用导入的证书或 AWS Private Certificate Authority 证书

要使用导入至 ACM 中的证书或来自 AWS Private Certificate Authority 的带有双向 TLS 的证书，API Gateway 需要由 ACM 颁发的 `ownershipVerificationCertificate`。此所有权证书仅用于验证您是否有权使用域名。它不用于 TLS 握手。如果您还没有 `ownershipVerificationCertificate`，请前往 https://console.aws.amazon.com/acm/ 来设置一个。

您需要确保此证书在域名生命周期内有效。如果证书过期且自动续订失败，则域名的所有更新都将被锁定。您需要使用有效 `ownershipVerificationCertificate` 更新 `ownershipVerificationCertificateArn`，然后才能进行任何其他更改。`ownershipVerificationCertificate` 不能用作 API Gateway 中另一个双向 TLS 域的服务器证书。如果直接向 ACM 中重新导入证书，发布者必须保持不变。

配置信任存储库

信任存储库是带有 `.pem` 文件扩展名的文本文件。它们是来自证书颁发机构的证书的受信任列表。要使用双向 TLS，请创建您信任的 X.509 证书的信任存储库以访问您的 API。

您必须在信任存储库中包含完整的信任链，从颁发的 CA 证书到根 CA 证书。API Gateway 接受信任链中存在的任何 CA 发布的客户端证书。证书可以来自公有或私有证书颁发机构。证书的最大链长度可为四。您还可以提供自签名证书。信任存储库支持以下哈希算法：

- SHA-256 或更强
- RSA-2048 或更强
- ECDSA-256 或更强

API Gateway 验证许多证书属性。您可以使用 Lambda 授权方在客户端调用 API 时执行其他检查，包括检查证书是否已被吊销。API Gateway 验证以下属性：

验证	说明
X.509 语法	证书必须满足 X.509 语法要求。
完整性	证书的内容不得与信任存储库中证书颁发机构签名的内容有所差异。
有效性	证书的有效期必须是最新的。
名称链接/键链接	证书的名称和主题必须形成一个完整的链条。证书的最大链长度可为四。

以单个文件的形式将信任存储库上载到 Amazon S3 存储桶

Example certificates.pem

```
-----BEGIN CERTIFICATE-----
<Certificate contents>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Certificate contents>
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
<Certificate contents>
-----END CERTIFICATE-----
...
```

以下 AWS CLI 命令会将 `certificates.pem` 上载到 Amazon S3 存储桶。

```
aws s3 cp certificates.pem s3://bucket-name
```

为自定义域名配置双向 TLS

要为 HTTP API 配置双向 TLS，必须为 API 使用区域自定义域名，且 TLS 最低版本为 1.2。要了解有关创建和配置自定义域名的更多信息，请参阅 the section called “设置区域自定义域名”。

> **Note**
>
> 私有 API 不支持双向 TLS。

将信任存储库上传到 Amazon S3 后，您可以将自定义域名配置为使用双向 TLS。将以下内容（包括斜杠）粘贴到终端中：

```
aws apigatewayv2 create-domain-name \
    --domain-name api.example.com \
    --domain-name-configurations CertificateArn=arn:aws:acm:us-
west-2:123456789012:certificate/123456789012-1234-1234-1234-12345678 \
    --mutual-tls-authentication TruststoreUri=s3://bucket-name/key-name
```

创建域名后，您必须为 API 操作配置 DNS 记录和基本路径映射。要了解更多信息，请参阅 在 API Gateway 中设置区域自定义域名。

使用需要双向 TLS 的自定义域名调用 API

要调用启用了双向 TLS 的 API，客户端必须在 API 请求中提供受信任证书。当客户端尝试调用您的 API 时，API Gateway 会在您的信任存储库中查找客户端证书的发布者。为使 API Gateway 继续处理请求，证书的发布者和直至根 CA 证书的完整信任链必须位于您的信任存储库中。

以下示例 `curl` 命令将请求发送到在请求中包含 `api.example.com,` 的 `my-cert.pem`。`my-key.key` 是证书的私有密钥。

```
curl -v --key ./my-key.key --cert ./my-cert.pem api.example.com
```

仅当您的信任存储库信任证书时，才会调用您的 API。以下情况将导致 API Gateway 使 TLS 握手失败，并以 `403` 状态代码拒绝请求。如果您的证书：

- 不可信
- 已过期
- 未使用支持的算法

> **Note**
>
> API Gateway 不验证证书是否已被吊销。

更新您的信任存储库

要更新信任存储库中的证书，请将新的证书服务包上载到 Amazon S3。然后，您可以更新自定义域名以使用更新后的证书。

使用 Amazon S3 版本控制来维护信任存储库的多个版本。当您更新自定义域名以使用新的信任存储库版本时，如果证书无效，则 API Gateway 返回警告。

API Gateway 仅在您更新域名时才生成证书警告。如果先前上传的证书过期，API Gateway 不会通知您。

以下 AWS CLI 命令将自定义域名更新为使用新的信任存储库版本。

```
aws apigatewayv2 update-domain-name \
    --domain-name api.example.com \
    --domain-name-configurations CertificateArn=arn:aws:acm:us-
west-2:123456789012:certificate/123456789012-1234-1234-1234-12345678 \
    --mutual-tls-authentication TruststoreVersion='abcdef123'
```

禁用双向 TLS

要为自定义域名禁用双向 TLS，请从自定义域名中删除信任存储库，如以下命令所示。

```
aws apigatewayv2 update-domain-name \
    --domain-name api.example.com \
    --domain-name-configurations CertificateArn=arn:aws:acm:us-
west-2:123456789012:certificate/123456789012-1234-1234-1234-12345678 \
    --mutual-tls-authentication TruststoreUri=''
```

排查证书警告问题

当使用双向 TLS 创建自定义域名时，如果信任存储库中的证书无效，则 API Gateway 会返回警告。在更新自定义域名以使用新的信任存储库时，也可能会出现这种情况。警告指示证书存在问题以及生成警告的证书的主题。仍然为您的 API 启用双向 TLS，但某些客户端可能无法访问您的 API。

要标识生成警告的证书，您需要解码信任存储库中的证书。您可以使用诸如 `openssl` 等工具对证书进行解码和标识其主题。

以下命令显示证书的内容，包括其主题：

```
openssl x509 -in certificate.crt -text -noout
```

更新或删除生成警告的证书，然后将新信任存储库上传到 Amazon S3。上载新的信任存储库后，请更新您的自定义域名以使用新的信任存储库。

域名冲突故障排除

错误 `"The certificate subject <certSubject> conflicts with an existing certificate from a different issuer."` 表示多个证书颁发机构已发布此域的证书。对于证书中的每个主题，双向 TLS 域的 API Gateway 中只能有一个发布者。您需要通过单个发布者获取该主题的所有证书。如果您无法控制的证书出现问题，但您可以证明域名的所有权，请联系 AWS Support 以提出支持请求。

域名状态消息故障排除

`PENDING_CERTIFICATE_REIMPORT`：这意味着您将证书重新导入到 ACM，并且验证失败，因为新证书具有 SAN（主题备用名称），而 `ownershipVerificationCertificate` 不涵盖该 SAN，或是证书中的主题或 SAN 不涵盖域名。某些内容可能配置不正确，或导入了无效的证书。您需要将有效证书重新导入 ACM。有关验证的更多信息，请参阅验证域名所有权。

`PENDING_OWNERSHIP_VERIFICATION`：这意味着您之前验证的证书已过期，ACM 无法自动续订。您需要续订证书或请求新证书。有关证书续订的更多信息，请查看 ACM 对托管式证书续订进行故障排除指南。

监控您的 HTTP API

您可以使用 CloudWatch 指标和 CloudWatch Logs 来监控 HTTP API。通过合并日志和指标，您可以记录错误并监控 API 的性能。

> **Note**
>
> 在以下情况下，API Gateway 可能无法生成日志和指标：
>
> - “413 请求实体过大”错误
> - 过多的“429 请求太多”错误
> - 发送到没有 API 映射的自定义域的请求中出现 400 系列错误
> - 由内部故障造成的 500 系列错误

主题

- 使用 HTTP API 的指标
- 配置 HTTP API 的日志记录

使用 HTTP API 的指标

您可以使用 CloudWatch 监控 API 执行，这将从 API Gateway 收集原始数据，并将数据处理为便于阅读的近乎实时的指标。这些统计数据会保存 15 个月，从而使您能够访问历史信息，并能够更好地了解您的 Web 应用程序或服务的执行情况。默认情况下，API Gateway 指标数据会在一分钟时段内自动发送到 CloudWatch。要监控您的指标，请为您的 API 创建 CloudWatch 控制面板。有关如何创建 CloudWatch 控制面板的更多信息，请参阅《Amazon CloudWatch 用户指南》中的创建 CloudWatch 控制面板。有关更多信息，请参阅 Amazon CloudWatch 用户指南 中的什么是 Amazon CloudWatch？

HTTP API 支持以下指标。您还可以启用详细指标，以便将路由级指标写入到 Amazon CloudWatch。

指标	说明
4xx	在给定期间捕获的客户端错误数。
5xx	在给定期间捕获的服务器端错误数。
计数	给定期间内的 API 请求总数。
IntegrationLatency	从 API Gateway 将请求中继到后端到其从后端收到响应所经过的时间。
延迟	从 API Gateway 从客户端收到请求到其将响应返回给客户端所经过的时间。延迟包括集成延迟和其他 API Gateway 开销。
DataProcessed	处理的数据量（以字节为单位）。

您可以使用下表中的维度筛选 API Gateway 指标。

维度	说明
ApiId	筛选具有指定 API ID 的 API 的 API Gateway 指标。
ApiId、阶段	针对具有指定 API ID 和阶段 ID 的 API 阶段筛选 API Gateway 指标。
ApiId、方法、资源、阶段	使用指定的 API ID、阶段 ID、资源路径和路由 ID 筛选 API 方法的 API Gateway 指标。 除非您明确启用了详细的 CloudWatch 指标，否则 API Gateway 不会发送这些指标。要执行此操作，您可以通过调用 API Gateway V2 REST API 的 UpdateStage 操作，将 `detailedMetricsEnabled` 属性更新为 `true`。或者，您也可以调用 update-stage AWS CLI 命令，以将 `DetailedMetricsEnabled` 属性更新为 `true`。启用这些指标会对您的账户额外计费。有关定价信息，请参阅 Amazon CloudWatch 定价。

配置 HTTP API 的日志记录

您可以开启日志记录以将日志写入 CloudWatch Logs。您可以使用日志记录变量来自定义日志的内容。

要为 HTTP API 开启日志记录，您必须执行以下操作。

1. 确保您的用户具有激活日志记录所需的权限。

2. 创建 CloudWatch Logs 日志组。
3. 为 API 的某个阶段提供 CloudWatch Logs 日志组的 ARN。

激活日志记录的权限

要开启 API 的日志记录，您的用户必须具有以下权限。

Example

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:DescribeLogGroups",
                "logs:DescribeLogStreams",
                "logs:GetLogEvents",
                "logs:FilterLogEvents"
            ],
            "Resource": "arn:aws:logs:us-east-2:123456789012:log-group:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogDelivery",
                "logs:PutResourcePolicy",
                "logs:UpdateLogDelivery",
                "logs:DeleteLogDelivery",
                "logs:CreateLogGroup",
                "logs:DescribeResourcePolicies",
                "logs:GetLogDelivery",
                "logs:ListLogDeliveries"
            ],
            "Resource": "*"
        }
    ]
}
```

创建日志组并激活 HTTP API 的日志记录

您可以使用 AWS Management Console 或 AWS CLI 创建日志组并激活访问日志记录。

AWS Management Console

1. 创建 日志组。

   要了解如何使用控制台建立日志组，请参阅《Amazon CloudWatch Logs 用户指南》中的“创建日志组”。
2. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
3. 选择 HTTP API。
4. 在主导航面板的 Monitor（监控）选项卡下，选择 Logging（日志记录）。
5. 选择要激活日志记录的阶段，然后选择 Select（选择）。
6. 选择 Edit（编辑）以激活访问日志。
7. 开启 Access logging（访问日志），进入 CloudWatch Logs，然后选择日志格式。
8. 选择保存。

AWS CLI

以下 AWS CLI 命令创建日志组。

```
aws logs create-log-group --log-group-name my-log-group
```

您需要日志组的 Amazon 资源名称（ARN）才能开启日志记录。ARN 格式为 arn:aws:logs:*region*:*account-id*:log-group:*log-group-name*。

以下 AWS CLI 命令会为 HTTP API 的 `$default` 阶段开启日志记录。

```
aws apigatewayv2 update-stage --api-id abcdef \
    --stage-name '$default' \
    --access-log-settings '{"DestinationArn": "arn:aws:logs:region:account-id:log-group:log-group-name", "Format": "$context.identity.sourceIp - - [$context.requestTime] \"$context.httpMethod $context.routeKey $context.protocol\" $context.status $context.responseLength $context.requestId"}'
```

日志格式示例

一些常用的访问日志格式的示例在 API Gateway 控制台中显示，下面列出了这些格式。

- CLF (常用日志格式)：

```
$context.identity.sourceIp - - [$context.requestTime] "$context.httpMethod
 $context.routeKey $context.protocol" $context.status $context.responseLength
 $context.requestId $context.extendedRequestId
```

- JSON:

```
{ "requestId":"$context.requestId", "ip": "$context.identity.sourceIp",
 "requestTime":"$context.requestTime",
 "httpMethod":"$context.httpMethod","routeKey":"$context.routeKey",
 "status":"$context.status","protocol":"$context.protocol",
 "responseLength":"$context.responseLength", "extendedRequestId":
 "$context.extendedRequestId" }
```

- XML:

```
<request id="$context.requestId"> <ip>$context.identity.sourceIp</ip> <requestTime>
$context.requestTime</requestTime> <httpMethod>$context.httpMethod</httpMethod>
 <routeKey>$context.routeKey</routeKey> <status>$context.status</status> <protocol>
$context.protocol</protocol> <responseLength>$context.responseLength</responseLength>
 <extendedRequestId>$context.extendedRequestId</extendedRequestId> </request>
```

- CSV（逗号分隔值）：

```
$context.identity.sourceIp,$context.requestTime,$context.httpMethod,
$context.routeKey,$context.protocol,$context.status,$context.responseLength,
$context.requestId,$context.extendedRequestId
```

自定义 HTTP API 访问日志

您可以使用以下变量自定义 HTTP API 访问日志。要了解有关 HTTP API 的访问日志的更多信息，请参阅 配置 HTTP API 的日志记录。

参数	说明
`$context.accountId`	API 拥有者的 AWS 账户 ID。
`$context.apiId`	API Gateway 分配给您的 API 的标识符。
`$context.authorizer.claims.` *`property`*	成功对方法调用方进行身份验证后从 JSON Web 令牌（JWT）返回的声明的属性，如 `$context.authorizer.claims.username` 。有关更多信息，请参阅 使用 JWT 授权方控制对 HTTP API 的访问。 **Note** 调用 `$context.authorizer.claims` 将返回 null。
`$context.authorizer.error`	从授权方返回的错误消息。
`$context.authorizer.principalId`	Lambda 授权方返回的委托人用户标识。
`$context.authorizer.` *`property`*	从 API Gateway Lambda 授权方函数返回的 `context` 映射的指定键/值对的值。例如，如果授权方返回以下 `context` 映射： （见下方代码） 调用 `$context.authorizer.key` 将返回 `"value"` 字符串，调用 `$context.authorizer.numKey` 将返回 `1`，调用 `$context.authorizer.boolKey` 将返回 `true`。

```
"context" : {
  "key": "value",
  "numKey": 1,
  "boolKey": true
}
```

参数	说明
`$context.awsEndpointRequestId`	`x-amz-request-id` 或 `x-amzn-requestId` 标头中AWS端点的请求 ID。
`$context.awsEndpointRequestId2`	来自 `x-amz-id-2` 标头的AWS端点的请求 ID。
`$context.customDomain.basePathMatched`	传入请求所匹配的 API 映射路径。适用于客户端使用自定义域名访问 API 的情况。例如，如果客户端向 `https://api.example.com/v1/orders/1234` 发送请求，且该请求匹配路径为 `v1/orders` 的 API 映射，则值为 `v1/orders` 。要了解更多信息，请参阅“the section called “API 映射””。
`$context.dataProcessed`	处理的数据量（以字节为单位）。
`$context.domainName`	用于调用 API 的完整域名。这应与传入的 `Host` 标头相同。
`$context.domainPrefix`	`$context.domainName` 的第一个标签。
`$context.error.message`	包含 API Gateway 错误消息的字符串。
`$context.error.messageString`	`$context.error.message` 的带引号的值，即 `"$context.error.message"` 。
`$context.error.responseType`	一种 `GatewayResponse` 类型。有关更多信息，请参阅 the section called “指标” 和 the section called “设置网关响应以自定义错误响应”。
`$context.extendedRequestId`	等效于 `$context.requestId` 。
`$context.httpMethod`	所用的 HTTP 方法。有效值包括：`DELETE`、`GET`、`HEAD`、`OPTIONS`、`PATCH`、`POST` 和 `PUT`。

参数	说明
`$context.identity.accountId`	与请求关联的 AWS 账户 ID。对于使用 IAM 授权的路由支持此项。
`$context.identity.caller`	签发请求的调用方的委托人标识符。对于使用 IAM 授权的路由支持此项。
`$context.identity.cognitoAuthenticationProvider`	发出请求的调用方使用的 Amazon Cognito 身份验证提供商的逗号分隔列表。仅当使用 Amazon Cognito 凭证对请求签名时才可用。 例如，对于 Amazon Cognito 身份池中的身份，`cognito-idp. *region*.amazonaws.com/ *user_pool_id* ,cognito-idp. *region*.amazonaws.com/ *user_pool_id* :CognitoSignIn: *token subject claim*` 有关更多信息，请参阅 Amazon Cognito 开发人员指南 中的使用联合身份。
`$context.identity.cognitoAuthenticationType`	发出请求的调用方的 Amazon Cognito 身份验证类型。仅当使用 Amazon Cognito 凭证对请求签名时才可用。可能的值包括经过身份验证的身份的 `authenticated` 和未经身份验证的身份的 `unauthenticated` 。
`$context.identity.cognitoIdentityId`	发出请求的调用方的 Amazon Cognito 身份 ID。仅当使用 Amazon Cognito 凭证对请求签名时才可用。
`$context.identity.cognitoIdentityPoolId`	发出请求的调用方的 Amazon Cognito 身份池 ID。仅当使用 Amazon Cognito 凭证对请求签名时才可用。
`$context.identity.principalOrgId`	AWS 组织 ID。对于使用 IAM 授权的路由支持此项。

参数	说明
`$context.identity.clientCert.clientCertPem`	客户端在双向 TLS 身份验证过程中提供的 PEM 编码的客户端证书。当客户端使用已启用双向 TLS 的自定义域名访问 API 时提供。
`$context.identity.clientCert.subjectDN`	客户端提供的证书的主题的可分辨名称。当客户端使用已启用双向 TLS 的自定义域名访问 API 时提供。
`$context.identity.clientCert.issuerDN`	客户端提供的证书的颁发者的可分辨名称。当客户端使用已启用双向 TLS 的自定义域名访问 API 时提供。
`$context.identity.clientCert.serialNumber`	证书的序列号。当客户端使用已启用双向 TLS 的自定义域名访问 API 时提供。
`$context.identity.clientCert.validity.notBefore`	证书无效之前的日期。当客户端使用已启用双向 TLS 的自定义域名访问 API 时提供。
`$context.identity.clientCert.validity.notAfter`	证书无效后的日期。当客户端使用已启用双向 TLS 的自定义域名访问 API 时提供。
`$context.identity.sourceIp`	向 API Gateway 端点发出请求的即时 TCP 连接的源 IP 地址。
`$context.identity.user`	将获得资源访问权限授权的用户的委托人标识符。对于使用 IAM 授权的路由支持此项。
`$context.identity.userAgent`	API 调用方的 `User-Agent` 标头。
`$context.identity.userArn`	身份验证后标识的有效用户的 Amazon Resource Name (ARN)。对于使用 IAM 授权的路由支持此项。有关更多信息，请参阅 https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users.html。
`$context.integration.error`	从集成返回的错误消息。等效于 `$context.integrationErrorMessage` 。

参数	说明
`$context.integration.integrationStatus`	对于 Lambda 代理集成，从 AWS Lambda（而不是从后端 Lambda 函数代码）返回的状态代码。
`$context.integration.latency`	集成延迟（毫秒）。等效于 `$context.integrationLatency` 。
`$context.integration.requestId`	AWS 端点的请求 ID。等效于 `$context.awsEndpointRequestId` 。
`$context.integration.status`	从集成返回的状态代码。对于 Lambda 代理集成，这是 Lambda 函数代码返回的状态代码。
`$context.integrationErrorMessage`	包含集成错误消息的字符串。
`$context.integrationLatency`	集成延迟（毫秒）。
`$context.integrationStatus`	对于 Lambda 代理集成，此参数表示从 AWS Lambda（而不是从后端 Lambda 函数）返回的状态代码。
`$context.path`	请求路径。例如，`/{stage}/root/child` 。
`$context.protocol`	请求的协议，例如，`HTTP/1.1`。 Note API Gateway API 可以接受 HTTP/2 请求，但 API Gateway 使用 HTTP/1.1 向后端集成发送请求。因此，即使客户端发送的请求使用 HTTP/2，请求协议也会记录为 HTTP/1.1。
`$context.requestId`	API Gateway 分配给 API 请求的 ID。

参数	说明
`$context.requestTime`	CLF 格式的请求时间 (dd/MMM/yyyy:HH:mm:ss +-hhmm)。
`$context.requestTimeEpoch`	Epoch 格式的请求时间。
`$context.responseLatency`	响应延迟（毫秒）。
`$context.responseLength`	响应负载长度（以字节为单位）。
`$context.routeKey`	API 请求的路由密钥，例如 `/pets`。
`$context.stage`	API 请求的部署阶段（例如，`beta` 或 `prod`）。
`$context.status`	方法响应状态。

排查 HTTP API 的问题

以下主题为您在使用 HTTP API 时可能遇到的错误和问题提供问题排查建议。

主题

- 对 HTTP API Lambda 集成的问题进行故障排查
- 排查 HTTP API JWT 授权方的问题

对 HTTP API Lambda 集成的问题进行故障排查

以下内容为您在将 AWS Lambda 集成 与 HTTP API 结合使用时可能遇到的错误和问题提供故障排除建议。

问题：我的 API 与 Lambda 集成返回 `{"message":"Internal Server Error"}`

要解决此内部服务器错误，请将`$context.integrationErrorMessage`日志记录变量添加到日志格式中，然后查看 HTTP API 的日志。为此，请执行以下操作：

要使用 创建日志组AWS Management Console

1. 通过以下网址打开 CloudWatch 控制台：https://console.aws.amazon.com/cloudwatch/。

2. 选择日志组。
3. 选择创建日志组。
4. 输入日志组名称，然后选择创建。
5. 记下您的日志组的 Amazon 资源名称 (ARN)。ARN 格式为 arn:aws:logs:*region*: *account-id*:log-group:*log-group-name*。您需要日志组 ARN 才能为 HTTP API 启用访问日志记录。

添加 `$context.integrationErrorMessage` 日志记录变量

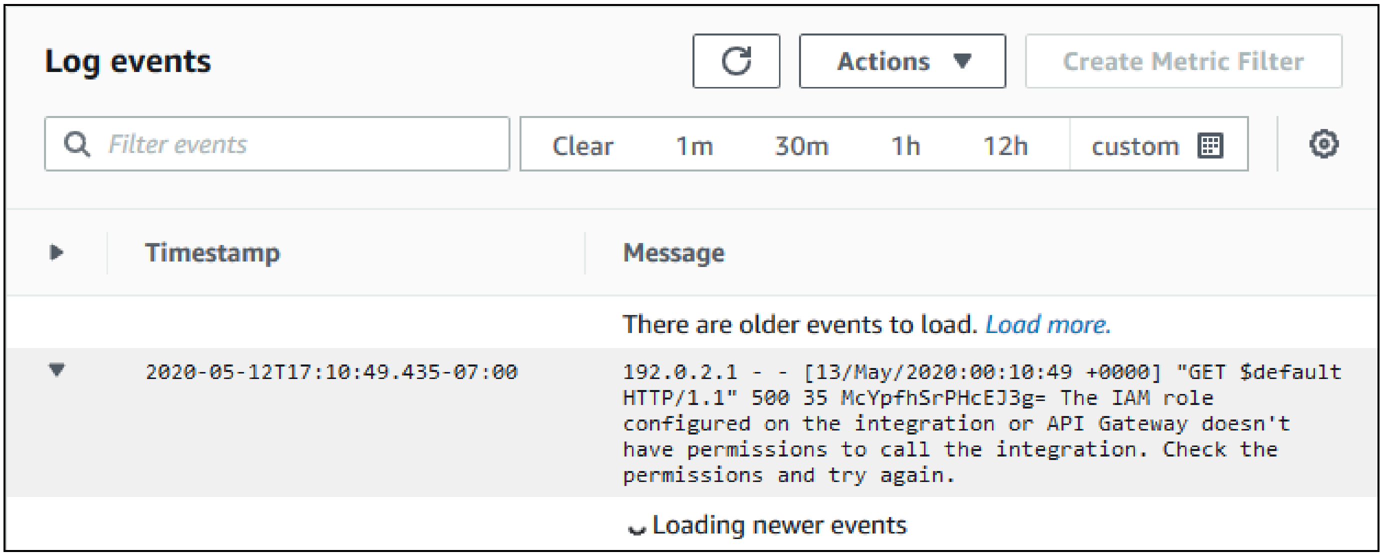
1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择您的 HTTP API。
3. 在监控下，选择日志记录。
4. 选择 API 的一个阶段。
5. 选择编辑，然后启用访问日志记录。
6. 为日志目标输入您在上一步中创建的安全组的 ARN。
7. 对于日志格式，选择 CLF。API Gateway 创建一个示例日志格式。
8. 将 `$context.integrationErrorMessage` 添加到日志格式的末尾。
9. 选择保存。

查看 API 的日志

1. 生成日志。使用浏览器或 `curl` 调用您的 API。

   ```
   $curl https://api-id.execute-api.us-west-2.amazonaws.com/route
   ```

2. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
3. 选择您的 HTTP API。
4. 在监控下，选择日志记录。
5. 选择您启用了日志记录的 API 阶段。
6. 选择查看 CloudWatch 中的日志。
7. 选择最新的日志流以查看 HTTP API 的日志。
8. 您的日志条目应类似于以下内容：



```
2020-05-12T17:10:49.435-07:00
192.0.2.1 - - [13/May/2020:00:10:49 +0000] "GET $default
HTTP/1.1" 500 35 McYpfhSrPHcEJ3g= The IAM role
configured on the integration or API Gateway doesn't
have permissions to call the integration. Check the
permissions and try again.
```

因为我们已将 `$context.integrationErrorMessage` 添加到日志格式，所以我们在日志中会看到一条总结此问题的错误消息。

您的日志可能包含不同的错误消息，以指示您的 Lambda 函数代码存在问题。在这种情况下，检查您的 Lambda 函数代码，并验证您的 Lambda 函数以所需格式返回响应。如果您的日志不包含错误消息，请在日志格式中添加 `$context.error.message` 和 `$context.error.responseType` 以获取更多信息以帮助进行故障排除。

在这种情况下，日志显示 API Gateway 不具备调用 Lambda 函数所需的权限。

当您在 API Gateway 控制台中创建 Lambda 集成时，API Gateway 会自动配置权限以调用 Lambda 函数。当您使用 AWS CLI、AWS CloudFormation 或开发工具包创建 Lambda 集成时，您必须授予 API Gateway 调用函数的权限。以下示例 AWS CLI 命令为不同的 HTTP API 路由授予调用 Lambda 函数的权限。

Example 示例 – 针对 HTTP API 的 **`$default`** 阶段和 **`$default`** 路由

```
aws lambda add-permission \
    --function-name my-function \
    --statement-id apigateway-invoke-permissions \
    --action lambda:InvokeFunction \
    --principal apigateway.amazonaws.com \
    --source-arn "arn:aws:execute-api:us-west-2:123456789012:api-id/\$default/\
$default"
```

Example 示例 – 针对 HTTP API 的 **prod** 阶段和 **test** 路由

```
aws lambda add-permission \
    --function-name my-function \
    --statement-id apigateway-invoke-permissions \
    --action lambda:InvokeFunction \
    --principal apigateway.amazonaws.com \
    --source-arn "arn:aws:execute-api:us-west-2:123456789012:api-id/prod/*/test"
```

在 Lambda 控制台的权限选项卡中确认函数策略。

尝试再次调用您的 API。您应该看到 Lambda 函数的响应。

排查 HTTP API JWT 授权方的问题

以下内容为您在将 JSON Web 令牌 (JWT) 授权方与 HTTP API 结合使用时可能遇到的错误和问题提供故障排除建议。

问题：我的 API 返回 `401 {"message":"Unauthorized"}`

检查来自 API 的响应中的 `www-authenticate` 标头。

以下命令使用 `curl` 将请求发送到具有 JWT 授权方（`$request.header.Authorization` 作为其身份源）的 API。

```
$curl -v -H "Authorization: token" https://api-id.execute-api.us-
west-2.amazonaws.com/route
```

来自 API 的响应包含一个 `www-authenticate` 标头。

```
...
< HTTP/1.1 401 Unauthorized
< Date: Wed, 13 May 2020 04:07:30 GMT
< Content-Length: 26
< Connection: keep-alive
< www-authenticate: Bearer scope="" error="invalid_token" error_description="the token
 does not have a valid audience"
< apigw-requestid: Mc7UVioPPHcEKPA=
<
* Connection #0 to host api-id.execute-api.us-west-2.amazonaws.com left intact
{"message":"Unauthorized"}}
```

在这种情况下，`www-authenticate` 标头显示未为有效的受众颁发令牌。为使 Lambda 对请求授权，JWT 的 `aud` 或 `client_id` 声明必须与为授权方配置的受众条目之一匹配。API Gateway 只有在 `aud` 不存在时才验证 `client_id`。当 `aud` 和 `client_id` 同时存在时，API Gateway 会评估 `aud`。

您还可以对 JWT 进行解码，并验证它与 API 所需的发布者、受众和作用域匹配。网站 jwt.io 可以在浏览器中调试 JWT。OpenID Foundation 维护用于处理 JWT 的库列表。

要了解有关 JWT 授权方的更多信息，请参阅 使用 JWT 授权方控制对 HTTP API 的访问。

使用 WebSocket API

API Gateway 中的 WebSocket API 是与后端 HTTP 终端节点、Lambda 函数或其他 AWS 服务集成的 WebSocket 路由的集合。您可以使用 API Gateway 功能，帮助您处理 API 生命周期从创建到监控生产 API 的各个方面。

API Gateway WebSocket API 是双向的。客户端可以向服务发送消息，服务可以独立向客户端发送消息。这种双向行为可实现更丰富的客户端/服务交互，因为服务无需客户端发出明确请求即可将数据推送到客户端。WebSocket API 通常用于聊天应用程序、协作平台、多人游戏和金融交易平台等实时应用程序。

有关要开始使用的示例应用程序，请参阅教程：使用 WebSocket API、Lambda 和 DynamoDB 构建无服务器聊天应用程序。

在本节中，您可以了解如何使用 API Gateway 开发、发布、保护和监控 WebSocket API。

主题

关于 API Gateway 中的 WebSocket API

在 API Gateway 中，您可以创建 WebSocket API 作为AWS服务（如 Lambda 或 DynamoDB）或 HTTP 终端节点的有状态前端。WebSocket API 根据从客户端应用程序收到的消息内容来调用您的后端。

与接收和响应请求的 REST API 不同，WebSocket API 支持客户端应用程序与后端之间的双向通信。后端可以向连接的客户端发送回调消息。

在 WebSocket API 中，传入的 JSON 消息将根据您配置的路由定向到后端集成。（非 JSON 消息将定向到您配置的 `$default` 路由。）

路由包含一个路由键，这是在评估路由选择表达式时预期的值。`routeSelectionExpression` 是在 API 级别定义的属性。它指定了预期存在于消息负载中的 JSON 属性。有关路由选择表达式的更多信息，请参阅the section called “”。

例如，如果您的 JSON 消息包含一个 `action` 属性，并且您想要根据此属性执行不同操作，则您的路由选择表达式可能是 `${request.body.action}`。您的路由表将通过将 `action` 属性的值与您在表中定义的自定义路由键值相匹配来指定要执行的操作。

可以使用三个预定义路由：`$connect`、`$disconnect` 和 `$default`。此外，您还可以创建自定义路由。

- API Gateway 会在客户端和 WebSocket API 之间的持久连接处于启动状态时调用 `$connect` 路由。
- API Gateway 会在客户端或服务器与 API 断开连接时调用 `$disconnect` 路由。
- 如果找到匹配的路由，则 API Gateway 会在针对消息评估路由选择表达式之后调用自定义路由；匹配项确定调用哪个集成。
- 如果无法针对消息评估路径选择表达式或未找到匹配的路由，则 API Gateway 会调用 `$default` 路由。

有关 `$connect` 和 `$disconnect` 路由的更多信息，请参阅the section called “管理连接的用户和客户端应用程序”。

有关 `$default` 路由和自定义路由的更多信息，请参阅the section called “调用您的后端集成”。

后端服务可以将数据发送到连接的客户端应用程序。有关更多信息，请参阅 the section called “从后端服务向连接的客户端发送数据”。

管理连接的用户和客户端应用程序：`$connect` 和 `$disconnect` 路由

主题

- $connect 路由
- 从 $connect 路由传递连接信息
- $disconnect 路由

`$connect` 路由

客户端应用程序通过发送 WebSocket 升级请求连接到 WebSocket API。如果请求成功，则在建立连接时会执行 `$connect` 路由。

由于 WebSocket 连接是有状态连接，因此您只能在 `$connect` 路由上配置授权。`AuthN/AuthZ` 仅在连接时执行。

在执行完与 `$connect` 路由关联的集成之前，升级请求处于待处理状态，将不会建立实际连接。如果 `$connect` 请求失败（例如，由于 AuthN/AuthZ 失败或集成失败），则不会建立连接。

> **Note**
>
> 如果授权在 `$connect` 上失败，则不会建立连接，客户端将收到 `401` 或 `403` 响应。

为 `$connect` 设置集成是可选的。在以下情况下，您应该考虑设置 `$connect` 集成：

- 您希望使客户端能够使用 `Sec-WebSocket-Protocol` 字段指定子协议。有关示例代码，请参阅设置需要 WebSocket 子协议的 `$connect` 路由。
- 您希望在客户端连接时收到通知。
- 您希望限制连接或控制谁会连接。
- 您希望后端使用回调 URL 将消息发送回客户端。
- 您希望将每个连接 ID 和其他信息存储到数据库中（例如，Amazon DynamoDB）。

从 $connect 路由传递连接信息

您可以使用代理和非代理集成将信息从 `$connect` 路由传递到数据库或其他 AWS 服务。

使用代理集成传递连接信息

您可以通过事件中的 Lambda 代理集成访问连接信息。使用其他 AWS 服务 或 AWS Lambda 函数发布到连接。

以下 Lambda 函数显示如何使用 `requestContext` 对象记录连接 ID、域名、阶段名和查询字符串。

Node.js

```
export const handler = async(event, context) => {
   const connectId = event["requestContext"]["connectionId"]
   const domainName = event["requestContext"]["domainName"]
   const stageName = event["requestContext"]["stage"]
   const qs = event['queryStringParameters']
   console.log('Connection ID: ', connectId, 'Domain Name: ', domainName, 'Stage
Name: ', stageName, 'Query Strings: ', qs )
   return {"statusCode" : 200}
```

```
};
```

Python

```
import json
import logging
logger = logging.getLogger()
logger.setLevel("INFO")


def lambda_handler(event, context):
    connectId = event["requestContext"]["connectionId"]
    domainName = event["requestContext"]["domainName"]
    stageName = event["requestContext"]["stage"]
    qs = event['queryStringParameters']
    connectionInfo = {
        'Connection ID': connectId,
        'Domain Name': domainName,
        'Stage Name': stageName,
        'Query Strings': qs}
    logging.info(connectionInfo)
    return {"statusCode": 200}
```

使用非代理集成传递连接信息

- 您可以使用非代理集成访问连接信息。设置集成请求并提供 WebSocket API 请求模板。以下 Velocity 模板语言（VTL）映射模板提供了集成请求。此请求将以下详细信息发送给非代理集成：

  - 连接 ID
  - 域名
  - 阶段名称
  - 路径
  - 标头
  - 查询字符串

  此请求将连接 ID、域名、阶段名、路径、标头和查询字符串发送到非代理集成。

```
{
    "connectionId": "$context.connectionId",
```

```
    "domain": "$context.domainName",
    "stage": "$context.stage",
    "params": "$input.params()"
}
```

有关设置数据转换的更多信息，请参阅the section called “数据转换”。

要完成集成请求，请为集成响应设置 `StatusCode: 200`。要了解有关设置集成响应的更多信息，请参阅使用 API Gateway 控制台设置集成响应。

`$disconnect` 路由

在连接关闭后，会执行 `$disconnect` 路由。

连接可以由服务器或客户端关闭。由于连接在执行时已经关闭，因此 `$disconnect` 是最适合的事件。API Gateway 将尽最大努力将 `$disconnect` 事件传送到您的集成，但它不能保证交付。

后端可以使用 `@connections` API 启动断开连接。有关更多信息，请参阅 the section called “在后端服务中使用 @connections 命令”。

调用您的后端集成：`$default` 路由和自定义路由

主题

- 使用路由处理消息
- $default 路由
- 自定义路由
- 使用 API Gateway WebSocket API 集成连接到您的业务逻辑
- WebSocket API 和 REST API 之间的重要区别

使用路由处理消息

在 API Gateway WebSocket API 中，消息可以从客户端发送到后端服务，反之亦然。与 HTTP 的请求/响应模型不同，在 WebSocket 中，后端可以在客户端不采取任何操作的情况下向客户端发送消息。

消息可以是 JSON 或非 JSON。但是，只有 JSON 消息可以根据消息内容路由到特定的集成。非 JSON 消息通过 `$default` 路由传递到后端。

> **Note**
>
> API Gateway 支持最大 128 KB 的消息负载，最大帧大小为 32 KB。如果消息超过 32 KB，则必须将其拆分为多个帧，每个 32 KB 或更小。如果接收到更大的消息（或帧），则连接会关闭并显示代码 1009。
> 目前不支持二进制负载。如果接收到二进制帧，则连接会关闭并显示代码 1003。但是，可以将二进制负载转换为文本。请参阅 the section called “二进制媒体类型”。

使用 API Gateway 中的 WebSocket API，可以路由 JSON 消息以根据消息内容执行特定的后端服务。当客户端通过其 WebSocket 连接发送消息时，这会导致对 WebSocket API 的路由请求。请求将与 API Gateway 中具有相应路由键的路由匹配。您可以在 API Gateway 控制台中使用 AWS CLI 或AWS 开发工具包为 WebSocket API 设置路由请求。

> **Note**
>
> 在 AWS CLI 和 AWS 开发工具包中，您可以在创建集成之前或之后创建路由。目前，控制台不支持重用集成，因此您必须先创建路由，然后为该路由创建集成。

您可以配置 API Gateway，使其在继续集成请求之前对路由请求执行验证。如果验证失败，API Gateway 会在不调用后端的情况下使请求失败，向客户端发送类似于以下内容的 `"Bad request body"` 网关响应，并在 CloudWatch Logs 中发布验证结果：

```
{"message" : "Bad request body", "connectionId": "{connectionId}", "messageId":
 "{messageId}"}
```

这样可以减少对后端的不必要调用，并让您专注于 API 的其他要求。

您还可以为 API 的路由定义路由响应，以启用双向通信。路由响应描述了在完成特定路由的集成后将向客户端发送的数据。例如，如果您希望客户端向后端发送消息而不收到响应（单向通信），则无需为路由定义响应。但是，如果您不提供路由响应，API Gateway 将不会向您的客户端发送有关您的集成结果的任何信息。

`$default` 路由

每个 API Gateway WebSocket API 都可以有一个 `$default` 路由。这是一个特殊的路由值，可以通过以下方式使用：

- 您可以将它与已定义的路由键一起使用，来为与任何已定义路由键不匹配的传入消息指定“回退”路由（例如，返回特定错误消息的通用模拟集成）。
- 您可以在没有任何已定义路由键的情况下使用它，来指定将路由委派给后端组件的代理模型。
- 您可以使用它为非 JSON 负载指定路由。

自定义路由

如果您希望根据消息内容来调用特定集成，则可以通过创建自定义路由来实现。

自定义路由使用您指定的路由键和集成。当传入消息包含 JSON 属性，并且该属性的计算结果为与路由键值匹配的值时，API Gateway 将会调用集成。（有关更多信息，请参阅 the section called “关于 WebSocket API”。）

例如，假设您要创建聊天室应用程序。您可以从创建 WebSocket API 开始，其路径选择表达式为 `$request.body.action`。然后，您可以定义两个路由：`joinroom` 和 `sendmessage`。客户端应用程序可以通过发送如下消息来调用 `joinroom` 路由：

```
{"action":"joinroom","roomname":"developers"}
```

而且，它可以通过发送如下消息来调用 `sendmessage` 路由：

```
{"action":"sendmessage","message":"Hello everyone"}
```

使用 API Gateway WebSocket API 集成连接到您的业务逻辑

为 API Gateway WebSocket API 设置路由后，您必须指定您想使用的集成。与路由可以具有路由请求和路由响应一样，集成可以具有集成请求和集成响应。集成请求包含后端预期的信息，用以处理来自客户端的请求。集成响应 包含后端返回到 API Gateway 的数据，可用于构造要发送给客户端的消息（如果定义了路由响应）。

有关设置集成的更多信息，请参阅the section called “集成”。

WebSocket API 和 REST API 之间的重要区别

WebSocket API 的集成类似于 REST API 的集成，但有以下区别：

- 目前，在 API Gateway 控制台中，您必须先创建一个路径，然后创建一个集成作为该路由的目标。但是，在 API 和 CLI 中，您可以按任何顺序独立创建路由和集成。

- 您可以对多个路由使用单个集成。例如，如果您有一组彼此密切相关的操作，您可能希望所有这些路由都转到单个 Lambda 函数。您可以一次指定集成的详细信息并将其分配给每个相关路由，而不是多次定义它。

> **Note**
>
> 目前，控制台不支持重用集成，因此您必须先创建路由，然后为该路由创建集成。
> 在 AWS CLI 和 AWS 开发工具包中，您可以通过将路由的目标设置 `"integrations/{integration-id}"` 的值来重用集成，其中 `{integration-id}"` 是要与路由关联的集成的唯一 ID。

- API Gateway 提供多个选择表达式，您可以在路由和集成中使用它们。您无需依赖内容类型来选择输入模板或输出映射。与路径选择表达式一样，您可以定义要由 API Gateway 评估的选择表达式以选择正确的项。如果找不到匹配的模板，则所有这些都将回退到 `$default` 模板。
  - 在集成请求中，模板选择表达式支持 `$request.body.<json_path_expression>` 和静态值。
  - 在集成响应中，模板选择表达式支持 `$request.body.<json_path_expression>`、`$integration.response.statuscode`、`$integ` 和静态值。

在 HTTP 协议中，请求和响应是同步发送的，通信基本上是单向的。在 WebSocket 协议中，通信是双向的。响应是异步的，客户端不一定以与发送客户端消息相同的顺序接收响应。此外，后端可以向客户端发送消息。

> **Note**
>
> 对于配置为使用 `AWS_PROXY` 或 `LAMBDA_PROXY` 集成的路由，通信是单向的，而 API Gateway 不会自动将后端响应传递给路由响应。例如，对于 `LAMBDA_PROXY` 集成，Lambda 函数返回的正文将不会返回到客户端。如果希望客户端接收集成响应，则必须定义路由响应以使双向通信成为可能。

从后端服务向连接的客户端发送数据

API Gateway WebSocket API 提供了以下方法供您将数据从后端服务发送到连接的客户端：

- 集成可以发送响应，该响应通过您定义的路由响应返回到客户端。

- 您可以使用 `@connections` API 发送 POST 请求。有关更多信息，请参阅 the section called “在后端服务中使用 `@connections` 命令”。

API Gateway 中的 WebSocket 选择表达式

主题

API Gateway 使用选择表达式作为一种评估请求和响应上下文并生成键的方法。然后，该键用于从通常由您，即 API 开发人员提供的一组可能值中进行选择。确切的受支持变量集将因特定表达式而异。下文更为详细地描述了每个表达式。

对于所有表达式，该语言遵循相同的规则集：

- 变量以 `"$"` 为前缀。
- 大括号可用于明确定义变量边界，例如，`"${request.body.version}-beta"`。
- 支持多个变量，但评估仅发生一次（无递归评估）。
- 可以使用 `$` 对美元符号 (`"\"`) 进行转义。这在定义映射到保留的 `$default` 键（例如 `"\$default"`）的表达式时非常有用。
- 在某些情况下，需要模式格式。在这种情况下，表达式应该用正斜杠 (`"/"`) 包装起来，例如，`"/2\d\d/"`，以便匹配 2*XX* 状态代码。

路由响应选择表达式

路由响应用于对从后端到客户端的响应建模。对于 WebSocket API，路由响应是可选的。定义后，它向 API Gateway 发出信号，表示它应该在收到 WebSocket 消息时向客户端返回响应。

路由响应选择表达式的求解会产生路由响应键。最终，此密钥将用于从与 API 相关联的一个 `RouteResponses` 中进行选择。但是，目前仅支持 `$default` 键。

API 键选择表达式

如果服务确定仅当客户端提供有效的 API 键时给定的请求才应继续，则会求解此表达式。

目前，仅支持的两个值是 `$request.header.x-api-key` 和 `$context.authorizer.usageIdentifierKey`。

API 映射选择表达式

将会求解估此表达式以确定在使用自定义域发出请求时选择哪个 API 阶段。

目前，唯一支持的值是 `$request.basepath`。

WebSocket 选择表达式摘要

下表总结了 WebSocket API 中的选择表达式的用例：

选择表达式	求解为以下对象的键	备注	示例使用案例
`Api.RouteSelectionExpression`	`Route.RouteKey`	`$defaul` 持 作为包罗万象的路由。	根据客户端请求的上下文路由 WebSocket 消息。
`Route.ModelSelectionExpression`	`Route.RequestModels` 的键	可选。 如果是为非代理集成提供的，则会发生模型验证。 `$defaul` 持 作为包罗	在同一路径中动态执行请求验证。

选择表达式	求解为以下对象的键	备注	示例使用案例
		万象的路由。	
`Integration.TemplateSelectionExpression`	`Integration.RequestTemplates` 的键	可选。 可以为非代理集成提供，用于处理传入的负载。 `${reque` `.body.` `nPath}` 持 和静态值。 `$defau` 持 作为包罗万象的路由。	根据请求的动态属性处理调用方的请求。

选择表达式	求解为以下对象的键	备注	示例使用案例
`Integration.IntegrationResponseSelectionExpression`	`IntegrationResponse.IntegrationResponseKey`	可选。可以为非代理集成提供。 充当错误消息（来自 Lambda 或状态代码（来自 HTTP 集成）的模式匹配。 `$defaul` 代理集成需要来充当包罗万象的成功响应。	处理来自后端的响应。 选择根据后端的动态响应发生的操作（例如，明显地处理某些错误）。

选择表达式	求解为以下对象的键	备注	示例使用案例
`IntegrationResponse.TemplateSelectionExpression`	`IntegrationResponse.ResponseTemplates` 的键	可选。可以为非代理集成提供。 支持 $default。	在某些情况下，响应的动态属性可能决定相同路径和相关集成内的不同变换。 支持 `${request.body.jsonPath}` 、`${i` `tion.response.statuscode}` 、`${` `tion.response.header.headerName}` 、`${in` `tion.response.multivalueheader.headerName}` 和静态值。 `$default`支持 作

选择表达式	求解为以下对象的键	备注	示例使用案例
			为包罗万象的路由。
`Route.RouteResponseSelectionExpression`	`RouteResponse.RouteResponseKey`	应提供以对 WebSoc 路由启动双向通信。 目前，此值仅限于 `$defaul`	
`RouteResponse.ModelSelectionExpression`	`RouteResponse.RequestModels` 的键	目前不受支持。	

在 API Gateway 中开发 WebSocket API

本节提供有关开发 API Gateway API 时所需的 API Gateway 功能的详细信息。

在开发 API Gateway API 时，您可以决定 API 的许多特征。这些特征取决于 API 的使用案例。例如，您可能希望仅允许某些客户端调用您的 API，或者您可能希望它对所有人都可用。您可能需要 API 调用来执行 Lambda 函数、进行数据库查询或调用应用程序。

主题

- 在 API Gateway 中创建 WebSocket API
- 在 WebSocket API 中使用路由
- 控制和管理对 API Gateway 中 WebSocket API 的访问

- 设置 WebSocket API 集成
- 请求验证
- 为 WebSocket API 设置数据转换
- 为 WebSocket API 使用二进制媒体类型
- 调用 WebSocket API

在 API Gateway 中创建 WebSocket API

您可以使用 AWS CLI create-api 命令或使用 AWS SDK 中的 `CreateApi` 命令，在 API Gateway 控制台中创建 WebSocket API。以下过程说明如何创建新的 WebSocket API。

> **Note**
>
> WebSocket API 仅支持 TLS 1.2。不支持早期 TLS 版本。

使用 AWS CLI 命令创建 WebSocket API

使用 AWS CLI 创建 WebSocket API 需要调用 create-api 命令，如以下示例所示（该示例使用 `$request.body.action` 路由选择表达式创建 API）：

```
aws apigatewayv2 --region us-east-1 create-api --name "myWebSocketApi3" --protocol-type
 WEBSOCKET --route-selection-expression '$request.body.action'
```

输出示例：

```
{
    "ApiKeySelectionExpression": "$request.header.x-api-key",
    "Name": "myWebSocketApi3",
    "CreatedDate": "2018-11-15T06:23:51Z",
    "ProtocolType": "WEBSOCKET",
    "RouteSelectionExpression": "'$request.body.action'",
    "ApiId": "aabbccddee"
}
```

使用 API Gateway 控制台创建 WebSocket API

您可以通过选择 WebSocket 协议并为 API 命名来在控制台中创建 WebSocket API。

> **Important**
>
> 创建 API 后，您无法更改为其选择的协议。无法将 WebSocket API 转换为 REST API，反之亦然。

使用 API Gateway 控制台创建 WebSocket API

1. 登录 API Gateway 控制台，然后选择创建 API。
2. 在 WebSocket API 下，选择构建。仅支持区域端点。
3. 对于 API 名称，输入 API 的名称。
4. 对于路由选择表达式，输入一个值。例如，`$request.body.action`。

   有关路由选择表达式的更多信息，请参阅the section called “”。
5. 请执行以下操作之一：
   - 选择创建空白 API，以创建没有路由的 API。
   - 选择下一步，将路由附加到 API。

   您可以在创建 API 之后附加路由。

在 WebSocket API 中使用路由

在 WebSocket API 中，传入的 JSON 消息将根据您配置的路由定向到后端集成。（非 JSON 消息将定向到您配置的 `$default` 路由。）

路由包含一个路由键，这是在评估路由选择表达式时预期的值。`routeSelectionExpression` 是在 API 级别定义的属性。它指定了预期存在于消息负载中的 JSON 属性。有关路由选择表达式的更多信息，请参阅the section called “”。

例如，如果您的 JSON 消息包含一个 `action` 属性，并且您想要根据此属性执行不同操作，则您的路由选择表达式可能是 `${request.body.action}`。您的路由表将通过将 `action` 属性的值与您在表中定义的自定义路由键值相匹配来指定要执行的操作。

可以使用三个预定义路由：`$connect`、`$disconnect` 和 `$default`。此外，您还可以创建自定义路由。

- API Gateway 会在客户端和 WebSocket API 之间的持久连接处于启动状态时调用 `$connect` 路由。

- API Gateway 会在客户端或服务器与 API 断开连接时调用 `$disconnect` 路由。
- 如果找到匹配的路由，则 API Gateway 会在针对消息评估路由选择表达式之后调用自定义路由；匹配项确定调用哪个集成。
- 如果无法针对消息评估路径选择表达式或未找到匹配的路由，则 API Gateway 会调用 `$default` 路由。

路由选择表达式

当服务正在为传入消息选择要遵循的路由时，将会对路由选择表达式进行求解。该服务使用其 `routeKey` 与求解值完全匹配的路由。如果没有匹配项，并且存在具有 `$default` 路由键的路由，则将选择该路由。如果没有路由与求解值匹配，并且没有 `$default` 路由，则该服务将返回错误。对于基于 WebSocket 的 API，表达式的格式应为 `$request.body.`*`{path_to_body_element}`*。

例如，假设您要发送以下 JSON 消息：

```
{
    "service" : "chat",
    "action" : "join",
    "data" : {
        "room" : "room1234"
   }
}
```

您可能希望根据 `action` 属性选择 API 的行为。在这种情况下，您可以定义以下路由选择表达式：

```
$request.body.action
```

在此示例中，`request.body` 是指您的消息的 JSON 负载，`.action` 是 JSONPath 表达式。您可以在 `request.body` 之后使用任何 JSON 路径表达式，但请记住，结果将会字符串化。例如，如果您的 JSONPath 表达式返回包含两个元素的数组，那么它将显示为字符串 `"[item1, item2]"`。因此，最好将表达式求解为值而不是数组或对象。

您可以仅使用一个静态值，也可以使用多个变量。下表显示了针对上述负载的示例及其求解结果。

表达式	求解结果	说明
`$request.body.action`	`join`	未包装的变量

表达式	求解结果	说明
`${request.body.action}`	`join`	包装的变量
`${request.body.service}/${request.body.action}`	`chat/join`	具有静态值的多个变量
`${request.body.action}-${request.body.invalidPath}`	`join-`	如果找不到 JSONPath，则变量将解析为 ""。
`action`	`action`	静态值
`\$default`	`$default`	静态值

求解结果将用于查找路由。如果存在具有匹配路由键的路由，则将选择该路由来处理消息。如果找不到匹配的路由，则 API Gateway 尝试查找 `$default` 路由（如果可用）。如果未定义 `$default` 路由，则 API Gateway 将返回错误。

在 API Gateway 中为 WebSocket API 设置路由

首次创建新的 WebSocket API 时，有三个预定义的路由：`$connect`、`$disconnect` 和 `$default`。您可以使用控制台、API 或 AWS CLI 创建它们。如果需要，您可以创建自定义路由。有关更多信息，请参阅 the section called “关于 WebSocket API”。

> Note
>
> 在 CLI 中，您可以在创建集成之前或之后创建路由，并且可以为多个路由重用相同的集成。

使用 API Gateway 控制台创建路由

使用 API Gateway 控制台创建路由

1. 登录到 API Gateway 控制台，选择 API，然后选择路由。
2. 选择创建路由。
3. 在路径密钥中，输入路径密钥名称。您可以创建预定义路由（`$connect`、`$disconnect` 和 `$default`），也可以创建自定义路由。

   > **Note**
   >
   > 当您创建自定义路由时，请勿在路由键名称中使用 $ 前缀。此前缀是专为预定义路由预留的。

4. 选择并配置路径的集成类型。有关更多信息，请参阅 the section called “使用 API Gateway 控制台设置 WebSocket API 集成请求”。

使用 AWS CLI 创建路由

要使用 AWS CLI 创建路由，请调用 `create-route`，如以下示例所示：

```
aws apigatewayv2 --region us-east-1 create-route --api-id aabbccddee --route-key
 $default
```

输出示例：

```
{
    "ApiKeyRequired": false,
    "AuthorizationType": "NONE",
    "RouteKey": "$default",
    "RouteId": "1122334"
}
```

为 `$connect` 指定路由请求设置

当您为 API 设置 `$connect` 路由时，可以使用以下可选设置为 API 启用授权。有关更多信息，请参阅 the section called “`$connect` 路由”。

- Authorization (授权)：如果不需要授权，您可以指定 `NONE`。否则，您可以指定：

- `AWS_IAM`，以使用标准AWS IAM 策略来控制对 API 的访问。
- `CUSTOM`，以通过指定先前创建的 Lambda 授权方函数来实现 API 的授权。授权方可以驻留在您自己的AWS账户或其他AWS账户中。有关 Lambda 授权方的更多信息，请参阅 使用 API Gateway Lambda 授权方。

  > **Note**
  >
  > 在 API Gateway 控制台中，只有在您设置了如`CUSTOM`中所述的授权方函数后，the section called “配置 Lambda 授权方（控制台）” 设置才可见。

> **Important**
>
> Authorization (授权) 设置将会应用于整个 API，而不仅仅是 `$connect` 路由。`$connect` 路由保护其他路由，因为它在每个连接上都会被调用。

- API Key Required (需要 API 键)：您可以要求（可选）API 的 `$connect` 路由有 API 键。您可以将 API 键与使用计划一起使用来控制和跟踪对 API 的访问。有关更多信息，请参阅 the section called “使用计划”。

使用 API Gateway 控制台设置 **`$connect`** 路由请求

要使用 API Gateway 控制台为 WebSocket API 设置 `$connect` 路由请求，请执行以下操作：

1. 登录到 API Gateway 控制台，选择 API，然后选择路由。
2. 在路由下选择 `$connect`，或按照the section called “使用 API Gateway 控制台创建路由”创建 `$connect` 路由。
3. 在路由请求设置部分中，选择编辑。
4. 对于授权，选择一种授权类型。
5. 要为 `$connect` 路由要求 API，请选择需要 API 密钥。
6. 选择 Save changes（保存更改）。

在 API Gateway 中为 WebSocket API 设置路由响应

WebSocket 路由可以配置为双向或单向通信。除非您设置了路由响应，否则 API Gateway 不会将后端响应传递给路由响应。

> ⓘ Note
>
> 您只能为 WebSocket API 定义 `$default` 路由响应。您可以使用集成响应来处理来自后端服务的响应。有关更多信息，请参阅 the section called “集成响应概述”。

您可以使用 API Gateway 控制台、AWS CLI 或AWS开发工具包配置路由响应和响应选择表达式。

有关路由响应选择表达式的更多信息，请参阅the section called “”。

主题

- 使用 API Gateway 控制台设置路由响应
- 使用 AWS CLI 设置路由响应

使用 API Gateway 控制台设置路由响应

在创建 WebSocket API 并将代理 Lambda 函数附加到默认路由后，您可以使用 API Gateway 控制台设置路由响应：

1. 登录 API Gateway 控制台，对于 `$default` 路由选择集成了代理 Lambda 函数的 WebSocket API。
2. 在 Routes（路由）下，选择 `$default` 路由。
3. 选择启用双向通信。
4. 选择部署 API。
5. 将 API 部署到阶段。

使用以下 wscat 命令连接到您的 API。有关 `wscat` 的更多信息，请参阅 the section called “使用 `wscat` 连接到 WebSocket API 并向其发送消息”。

```
wscat -c wss://api-id.execute-api.us-east-2.amazonaws.com/test
```

按 Enter 键以调用默认路由。应返回您的 Lambda 函数的主体。

使用 AWS CLI 设置路由响应

要使用 AWS CLI 为 WebSocket API 设置路由响应，请调用 `create-route-response` 命令，如以下示例所示。您可以通过调用 `get-apis` 和 `get-routes` 来识别 API ID 和路由 ID。

```
aws apigatewayv2 create-route-response \
    --api-id aabbccddee \
    --route-id 1122334  \
    --route-response-key '$default'
```

输出示例：

```
{
    "RouteResponseId": "abcdef",
    "RouteResponseKey": "$default"
}
```

设置需要 WebSocket 子协议的 `$connect` 路由

在连接到 WebSocket API 期间，客户端可以使用 `Sec-WebSocket-Protocol` 字段请求 WebSocket 子协议。您可以为 `$connect` 路由设置集成，以便仅当客户端请求您的 API 支持的子协议时允许连接。

以下示例 Lambda 函数将 `Sec-WebSocket-Protocol` 标头返回给客户端。仅当客户端指定 `myprotocol` 子协议时，此函数才会建立与 API 的连接。

如需创建此示例 API 和 Lambda 代理集成的 AWS CloudFormation 模板，请参阅 `ws-subprotocol.yaml`。

```
export const handler = async (event) => {
    if (event.headers != undefined) {
        const headers = toLowerCaseProperties(event.headers);

        if (headers['sec-websocket-protocol'] != undefined) {
            const subprotocolHeader = headers['sec-websocket-protocol'];
            const subprotocols = subprotocolHeader.split(',');

            if (subprotocols.indexOf('myprotocol') >= 0) {
                const response = {
                    statusCode: 200,
                    headers: {
                        "Sec-WebSocket-Protocol" : "myprotocol"
                    }
                };
                return response;
            }
```

```
        }
    }

    const response = {
        statusCode: 400
    };

    return response;
};

function toLowerCaseProperties(obj) {
    var wrapper = {};
    for (var key in obj) {
        wrapper[key.toLowerCase()] = obj[key];
    }
    return wrapper;
}
```

您可以使用 `wscat` 测试您的 API，以确定是否仅当客户端请求 API 支持的子协议时允许连接。以下命令在连接过程中使用 `-s` 标志指定子协议。

以下命令尝试使用不受支持的子协议进行连接。由于客户端指定了 `chat1` 子协议，因此，Lambda 集成将返回 400 错误，并且连接将失败。

```
wscat -c wss://api-id.execute-api.region.amazonaws.com/beta -s chat1
error: Unexpected server response: 400
```

以下命令在连接请求中包含支持的子协议。Lambda 集成允许连接。

```
wscat -c wss://api-id.execute-api.region.amazonaws.com/beta -s chat1,myprotocol
connected (press CTRL+C to quit)
```

要了解有关调用 WebSocket API 的更多信息，请参阅调用 WebSocket API。

控制和管理对 API Gateway 中 WebSocket API 的访问

API Gateway 支持多种用于控制和管理对 WebSocket API 的访问的机制：

您可以使用以下机制进行身份验证和授权：

- 标准AWS IAM 角色和策略提供灵活且稳健的访问控制。提供灵活且稳健的访问控制。您可以使用 IAM 角色和策略来控制哪些人可以创建和管理您的 API，以及谁可以调用它们。有关更多信息，请参阅 使用 IAM 授权。
- IAM 标签 可与 IAM 策略结合使用来控制访问权限。有关更多信息，请参阅 使用标签控制对 API Gateway REST API 资源的访问。
- Lambda 授权方 是控制对 API 的访问的 Lambda 函数。有关更多信息，请参阅 创建 Lambda REQUEST 授权方函数。

主题

- 使用 IAM 授权
- 创建 Lambda REQUEST 授权方函数

使用 IAM 授权

WebSocket API 中的 IAM 授权类似于 REST API 的授权，但有以下例外情况：

- 除了现有操作（`execute-api`、`ManageConnections`）之外，`Invoke` 操作也支持 `InvalidateCache`。`ManageConnections` 控制对 @connections API 的访问。
- WebSocket 路由使用不同的 ARN 格式：

  ```
  arn:aws:execute-api:region:account-id:api-id/stage-name/route-key
  ```

- `@connections` API 使用与 REST API 相同的 ARN 格式：

  ```
  arn:aws:execute-api:region:account-id:api-id/stage-name/POST/@connections
  ```

> **Important**
>
> 在使用 IAM 授权时，必须使用签名版本 4 (SigV4) 对请求进行签名。

例如，您可以为客户端设置以下策略。此示例能让每个人为 `Invoke` 阶段中除密钥路由之外的所有路由发送消息 (`prod`)，并针对所有阶段阻止每个人将消息发送回连接的客户端 (`ManageConnections`)。

```
{
```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:account-id:api-id/prod/*"
      ]
    },
    {
      "Effect": "Deny",
      "Action": [
        "execute-api:Invoke"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:account-id:api-id/prod/secret"
      ]
    },
    {
      "Effect": "Deny",
      "Action": [
        "execute-api:ManageConnections"
      ],
      "Resource": [
        "arn:aws:execute-api:us-east-1:account-id:api-id/*"
      ]
    }
  ]
}
```

创建 Lambda **REQUEST** 授权方函数

WebSocket API 中的 Lambda 授权方函数类似于 REST API 的授权方函数，但有以下例外情况：

- 您只能对 `$connect` 路由使用 Lambda 授权方函数。
- 您不能使用路径变量 (`event.pathParameters`)，因为路径是固定的。
- `event.methodArn` 与 REST API 等效变量不同，因为它没有 HTTP 方法。如果是 `$connect`，则 `methodArn` 以 `"$connect"` 结尾：

```
arn:aws:execute-api:region:account-id:api-id/stage-name/$connect
```

- `event.requestContext` 中的上下文变量与 REST API 的上下文变量不同。

以下示例显示了适用于 WebSocket API 的 `REQUEST` 授权方的输入：

```
{
    "type": "REQUEST",
    "methodArn": "arn:aws:execute-api:us-east-1:123456789012:abcdef123/default/
$connect",
    "headers": {
        "Connection": "upgrade",
        "content-length": "0",
        "HeaderAuth1": "headerValue1",
        "Host": "abcdef123.execute-api.us-east-1.amazonaws.com",
        "Sec-WebSocket-Extensions": "permessage-deflate; client_max_window_bits",
        "Sec-WebSocket-Key": "...",
        "Sec-WebSocket-Version": "13",
        "Upgrade": "websocket",
        "X-Amzn-Trace-Id": "...",
        "X-Forwarded-For": "...",
        "X-Forwarded-Port": "443",
        "X-Forwarded-Proto": "https"
    },
    "multiValueHeaders": {
        "Connection": [
            "upgrade"
        ],
        "content-length": [
            "0"
        ],
        "HeaderAuth1": [
            "headerValue1"
        ],
        "Host": [
            "abcdef123.execute-api.us-east-1.amazonaws.com"
        ],
        "Sec-WebSocket-Extensions": [
            "permessage-deflate; client_max_window_bits"
        ],
        "Sec-WebSocket-Key": [
            "..."
        ],
        "Sec-WebSocket-Version": [
            "13"
```

```
        ],
        "Upgrade": [
            "websocket"
        ],
        "X-Amzn-Trace-Id": [
            "..."
        ],
        "X-Forwarded-For": [
            "..."
        ],
        "X-Forwarded-Port": [
            "443"
        ],
        "X-Forwarded-Proto": [
            "https"
        ]
    },
    "queryStringParameters": {
        "QueryString1": "queryValue1"
    },
    "multiValueQueryStringParameters": {
        "QueryString1": [
            "queryValue1"
        ]
    },
    "stageVariables": {},
    "requestContext": {
        "routeKey": "$connect",
        "eventType": "CONNECT",
        "extendedRequestId": "...",
        "requestTime": "19/Jan/2023:21:13:26 +0000",
        "messageDirection": "IN",
        "stage": "default",
        "connectedAt": 1674162806344,
        "requestTimeEpoch": 1674162806345,
        "identity": {
            "sourceIp": "..."
        },
        "requestId": "...",
        "domainName": "abcdef123.execute-api.us-east-1.amazonaws.com",
        "connectionId": "...",
        "apiId": "abcdef123"
    }
```

```
}
```

以下示例 Lambda 授权方函数是[the section called “其他 Lambda 授权方函数示例”]中 REST API 的 Lambda 授权方函数的 WebSocket 版本：

Node.js

```
  // A simple REQUEST authorizer example to demonstrate how to use request
  // parameters to allow or deny a request. In this example, a request is
  // authorized if the client-supplied HeaderAuth1 header and QueryString1 query
parameter
  // in the request context match the specified values of
  // of 'headerValue1' and 'queryValue1' respectively.
           export const handler = function(event, context, callback) {
   console.log('Received event:', JSON.stringify(event, null, 2));

  // Retrieve request parameters from the Lambda function input:
  var headers = event.headers;
  var queryStringParameters = event.queryStringParameters;
  var stageVariables = event.stageVariables;
  var requestContext = event.requestContext;

  // Parse the input for the parameter values
  var tmp = event.methodArn.split(':');
  var apiGatewayArnTmp = tmp[5].split('/');
  var awsAccountId = tmp[4];
  var region = tmp[3];
  var ApiId = apiGatewayArnTmp[0];
  var stage = apiGatewayArnTmp[1];
  var route = apiGatewayArnTmp[2];

  // Perform authorization to return the Allow policy for correct parameters and
  // the 'Unauthorized' error, otherwise.
  var authResponse = {};
  var condition = {};
   condition.IpAddress = {};

  if (headers.HeaderAuth1 === "headerValue1"
      && queryStringParameters.QueryString1 === "queryValue1") {
       callback(null, generateAllow('me', event.methodArn));
   }  else {
       callback("Unauthorized");
   }
```

```
}

// Helper function to generate an IAM policy
var generatePolicy = function(principalId, effect, resource) {
   // Required output:
   var authResponse = {};
    authResponse.principalId = principalId;
   if (effect && resource) {
       var policyDocument = {};
        policyDocument.Version = '2012-10-17'; // default version
       policyDocument.Statement = [];
       var statementOne = {};
        statementOne.Action = 'execute-api:Invoke'; // default action
       statementOne.Effect = effect;
        statementOne.Resource = resource;
        policyDocument.Statement[0] = statementOne;
        authResponse.policyDocument = policyDocument;
    }
   // Optional output with custom properties of the String, Number or Boolean type.
   authResponse.context = {
       "stringKey": "stringval",
       "numberKey": 123,
       "booleanKey": true
    };
   return authResponse;
}

var generateAllow = function(principalId, resource) {
   return generatePolicy(principalId, 'Allow', resource);
}

var generateDeny = function(principalId, resource) {
   return generatePolicy(principalId, 'Deny', resource);
}
```

Python

```
# A simple REQUEST authorizer example to demonstrate how to use request
# parameters to allow or deny a request. In this example, a request is
# authorized if the client-supplied HeaderAuth1 header and QueryString1 query
 parameter
# in the request context match the specified values of
# of 'headerValue1' and 'queryValue1' respectively.
```

```
import json


def lambda_handler(event, context):
    print(event)

    # Retrieve request parameters from the Lambda function input:
    headers = event['headers']
    queryStringParameters = event['queryStringParameters']
    stageVariables = event['stageVariables']
    requestContext = event['requestContext']

    # Parse the input for the parameter values
    tmp = event['methodArn'].split(':')
    apiGatewayArnTmp = tmp[5].split('/')
    awsAccountId = tmp[4]
    region = tmp[3]
    ApiId = apiGatewayArnTmp[0]
    stage = apiGatewayArnTmp[1]
    route = apiGatewayArnTmp[2]

    # Perform authorization to return the Allow policy for correct parameters
    # and the 'Unauthorized' error, otherwise.

    authResponse = {}
    condition = {}
    condition['IpAddress'] = {}

    if (headers['HeaderAuth1'] ==
            "headerValue1" and queryStringParameters["QueryString1"] ==
 "queryValue1"):
        response = generateAllow('me', event['methodArn'])
        print('authorized')
        return json.loads(response)
    else:
        print('unauthorized')
        return 'unauthorized'

    # Help function to generate IAM policy


def generatePolicy(principalId, effect, resource):
    authResponse = {}
```

```
    authResponse['principalId'] = principalId
    if (effect and resource):
        policyDocument = {}
        policyDocument['Version'] = '2012-10-17'
        policyDocument['Statement'] = []
        statementOne = {}
        statementOne['Action'] = 'execute-api:Invoke'
        statementOne['Effect'] = effect
        statementOne['Resource'] = resource
        policyDocument['Statement'] = [statementOne]
        authResponse['policyDocument'] = policyDocument

    authResponse['context'] = {
        "stringKey": "stringval",
        "numberKey": 123,
        "booleanKey": True
    }

    authResponse_JSON = json.dumps(authResponse)

    return authResponse_JSON


def generateAllow(principalId, resource):
    return generatePolicy(principalId, 'Allow', resource)


def generateDeny(principalId, resource):
    return generatePolicy(principalId, 'Deny', resource)
```

要将前面的 Lambda 函数配置为 WebSocket API 的 `REQUEST` 授权方函数，请遵循与 REST API 相同的过程。

要将 `$connect` 路由配置为在控制台中使用此 Lambda 授权方，请选择或创建 `$connect` 路由。在路由请求设置部分中，选择编辑。在授权下拉菜单中选择您的授权方，然后选择保存更改。

要测试授权方，您将需要创建一个新连接。在 `$connect` 中更改授权方不会影响已连接的客户端。当您连接到 WebSocket API 时，需要为任何已配置的身份源提供值。例如，您可以通过使用 `wscat` 发送有效的查询字符串和标头进行连接，如以下示例所示：

```
wscat -c 'wss://myapi.execute-api.us-east-1.amazonaws.com/beta?
QueryString1=queryValue1' -H HeaderAuth1:headerValue1
```

如果您尝试在没有有效身份值的情况下进行连接，您将收到 `401` 响应：

```
wscat -c wss://myapi.execute-api.us-east-1.amazonaws.com/beta
error: Unexpected server response: 401
```

设置 WebSocket API 集成

设置 API 路由之后，您必须将其与后端中的端点集成。后端端点也称为集成端点，它可以是 Lambda 函数、HTTP 端点或AWS服务操作。API 集成有一个集成请求和一个集成响应。

在本节中，您将了解如何为 WebSocket API 设置集成请求和集成响应。

主题

- 在 API Gateway 中设置 WebSocket API 集成请求
- 在 API Gateway 中设置 WebSocket API 集成响应

在 API Gateway 中设置 WebSocket API 集成请求

设置集成请求涉及以下内容：

- 选择要集成到后端的路由键。
- 指定要调用的后端端点。WebSocket API 支持以下集成类型：
  - `AWS_PROXY`
  - `AWS`
  - `HTTP_PROXY`
  - `HTTP`
  - `MOCK`

  有关集成类型的更多信息，请参阅《API Gateway V2 REST API》中的 IntegrationType。
- 通过指定一个或多个请求模板，配置如何根据需要将路径请求数据转换为集成请求数据。

使用 API Gateway 控制台设置 WebSocket API 集成请求

使用 API Gateway 控制台向 WebSocket API 中的路由添加集成请求

1. 登录到 API Gateway 控制台，选择 API，然后选择路由。
2. 在路由中，选择所需路由。
3. 选择集成请求选项卡，然后在集成请求设置部分中，选择编辑。
4. 对于集成类型，选择下列选项之一：
   - 仅当您的 API 将与您已在此账户或其它账户中创建的 AWS Lambda 函数集成时，才选择 Lambda 函数。

     要在 AWS Lambda 中创建新的 Lambda 函数，要为 Lambda 函数设置资源权限，或者要执行任何其他 Lambda 服务操作，请改为选择 AWS 服务。
   - 如果您的 API 将与现有 HTTP 端点集成，请选择 HTTP。有关更多信息，请参阅 在 API Gateway 中设置 HTTP 集成。
   - 如果要直接从 API Gateway 生成 API 响应，而无需集成后端，请选择模拟。有关更多信息，请参阅 在 API Gateway 中设置模拟集成。
   - 如果您的 API 将与某项 AWS 服务集成，则选择 AWS 服务。
   - 如果您的 API 将使用 `VpcLink` 作为私有集成端点，请选择 VPC 链接。有关更多信息，请参阅 设置 API Gateway 私有集成。
5. 如果您选择 Lambda 函数，请执行以下操作：
   a. 对于使用 Lambda 代理集成，如果您打算使用 Lambda 代理集成或跨账户 Lambda 代理集成，请选中此复选框。
   b. 对于 Lambda 函数，请通过以下方式之一指定函数：
      - 如果您的 Lambda 函数在同一账户中，请输入函数名称，然后从下拉列表中选择函数。

        > Note
        >
        > 函数名称可以包含（可选）其别名或版本规范，如在 `HelloWorld`、`HelloWorld:1` 或 `HelloWorld:alpha` 中。
      - 如果该函数位于不同账户，请输入该函数的 ARN。
   c. 要使用默认超时值 29 秒，请保持默认超时处于开启状态。要设置自定义超时，请选择默认超时，然后输入一个介于 `50` 到 `29000` 毫秒之间的超时值。

6. 如果您选择了 HTTP，请遵循the section called “ 使用控制台设置集成请求”的步骤 4 中的说明。
7. 如果您选择了模拟，请继续执行请求模板步骤。
8. 如果您选择了 AWS 服务，请遵循the section called “ 使用控制台设置集成请求”的步骤 6 中的说明操作。
9. 如果您选择了 VPC 链接，请执行以下操作：

   a. 对于 VPC 代理集成，如果要将请求通过代理连接到 `VPCLink` 的端点，请选中该复选框。

   b. 对于 HTTP 方法，选择与 HTTP 后端中的方法最匹配的 HTTP 方法类型。

   c. 从 VPC 链接下拉列表中，选择一个 VPC 链接。您可以选择 `[Use Stage Variables]` 并在列表下方的文本框中输入 **`${stageVariables.vpcLinkId}`**。

      您可以在将 API 部署到阶段之后定义 `vpcLinkId` 阶段变量，并将其值设置为 `VpcLink` 的 ID。

   d. 对于端点 URL，请输入您希望此集成使用的 HTTP 后端的 URL。

   e. 要使用默认超时值 29 秒，请保持默认超时处于开启状态。要设置自定义超时，请选择默认超时，然后输入一个介于 `50` 到 `29000` 毫秒之间的超时值。

10. 选择 Save changes（保存更改）。
11. 在请求模板下方，执行以下操作：

    a. 要输入模板选择表达式，请在请求模板下选择编辑。

    b. 输入模板选择表达式。使用 API Gateway 在消息负载中查找的表达式。如果找到，则对其进行评估，结果是模板键值，用于选择要应用于消息负载中的数据的数据映射模板。您将在下一步中创建数据映射模板。选择编辑以保存所做更改。

    c. 选择创建模板以创建数据映射模板。对于模板密钥，输入一个模板密钥值，用于选择要应用于消息负载中的数据的数据映射模板。然后，输入映射模板。选择创建模板。

       有关模板选择表达式的信息，请参阅the section called “模板选择表达式”。

使用 AWS CLI 设置集成请求

您可以使用 AWS CLI 为 WebSocket API 中的路由设置集成请求，如以下示例所示（这将创建模拟集成）：

1. 使用以下内容创建名为 `integration-params.json` 的文件：

```
{"PassthroughBehavior": "WHEN_NO_MATCH", "TimeoutInMillis": 29000,
 "ConnectionType": "INTERNET", "RequestTemplates": {"application/json":
 "{\"statusCode\":200}"}, "IntegrationType": "MOCK"}
```

2. 运行 create-integration 命令，如以下示例所示：

```
aws apigatewayv2 --region us-east-1 create-integration --api-id aabbccddee --cli-
input-json file://integration-params.json
```

以下是此示例的示例输出：

```
{
    "PassthroughBehavior": "WHEN_NO_MATCH",
    "TimeoutInMillis": 29000,
    "ConnectionType": "INTERNET",
    "IntegrationResponseSelectionExpression": "${response.statuscode}",
    "RequestTemplates": {
        "application/json": "{\"statusCode\":200}"
    },
    "IntegrationId": "0abcdef",
    "IntegrationType": "MOCK"
}
```

或者，您可以使用 AWS CLI 为代理集成设置集成请求，如以下示例所示：

1. 在 Lambda 控制台中创建 Lambda 函数，并为其提供基本的 Lambda 执行角色。
2. 运行 create-integration 命令，如以下示例所示：

```
aws apigatewayv2 create-integration --api-id aabbccddee --integration-type
 AWS_PROXY --integration-method POST --integration-uri arn:aws:apigateway:us-
east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-
east-1:123412341234:function:simpleproxy-echo-e2e/invocations
```

以下是此示例的示例输出：

```
{
    "PassthroughBehavior": "WHEN_NO_MATCH",
    "IntegrationMethod": "POST",
    "TimeoutInMillis": 29000,
```

```
    "ConnectionType": "INTERNET",
    "IntegrationUri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123412341234:function:simpleproxy-echo-e2e/invocations",
    "IntegrationId": "abcdefg",
    "IntegrationType": "AWS_PROXY"
}
```

适用于 WebSocket API 的代理集成的 Lambda 函数的输入格式

使用 Lambda 代理集成，API Gateway 可以将整个客户端请求映射到后端 Lambda 函数的输入 `event` 参数：以下示例显示了 API Gateway 发送到 Lambda 代理集成的 `$connect` 路由中输入事件和 `$disconnect` 路由中输入事件的结构。

Input from the $connect route

```
{
    headers: {
      Host: 'abcd123.execute-api.us-east-1.amazonaws.com',
      'Sec-WebSocket-Extensions': 'permessage-deflate; client_max_window_bits',
      'Sec-WebSocket-Key': '...',
      'Sec-WebSocket-Version': '13',
      'X-Amzn-Trace-Id': '...',
      'X-Forwarded-For': '192.0.2.1',
      'X-Forwarded-Port': '443',
      'X-Forwarded-Proto': 'https'
    },
    multiValueHeaders: {
      Host: [ 'abcd123.execute-api.us-east-1.amazonaws.com' ],
      'Sec-WebSocket-Extensions': [ 'permessage-deflate; client_max_window_bits' ],
      'Sec-WebSocket-Key': [ '...' ],
      'Sec-WebSocket-Version': [ '13' ],
      'X-Amzn-Trace-Id': [ '...' ],
      'X-Forwarded-For': [ '192.0.2.1' ],
      'X-Forwarded-Port': [ '443' ],
      'X-Forwarded-Proto': [ 'https' ]
    },
    requestContext: {
      routeKey: '$connect',
      eventType: 'CONNECT',
      extendedRequestId: 'ABCD1234=',
      requestTime: '09/Feb/2024:18:11:43 +0000',
      messageDirection: 'IN',
      stage: 'prod',
```

```
      connectedAt: 1707502303419,
      requestTimeEpoch: 1707502303420,
      identity: { sourceIp: '192.0.2.1' },
      requestId: 'ABCD1234=',
      domainName: 'abcd1234.execute-api.us-east-1.amazonaws.com',
      connectionId: 'AAAA1234=',
      apiId: 'abcd1234'
    },
    isBase64Encoded: false
  }
```

Input from the $disconnect route

```
{
    headers: {
      Host: 'abcd1234.execute-api.us-east-1.amazonaws.com',
      'x-api-key': '',
      'X-Forwarded-For': '',
      'x-restapi': ''
    },
    multiValueHeaders: {
      Host: [ 'abcd1234.execute-api.us-east-1.amazonaws.com' ],
      'x-api-key': [ '' ],
      'X-Forwarded-For': [ '' ],
      'x-restapi': [ '' ]
    },
    requestContext: {
      routeKey: '$disconnect',
      disconnectStatusCode: 1005,
      eventType: 'DISCONNECT',
      extendedRequestId: 'ABCD1234=',
      requestTime: '09/Feb/2024:18:23:28 +0000',
      messageDirection: 'IN',
      disconnectReason: 'Client-side close frame status not set',
      stage: 'prod',
      connectedAt: 1707503007396,
      requestTimeEpoch: 1707503008941,
      identity: { sourceIp: '192.0.2.1' },
      requestId: 'ABCD1234=',
      domainName: 'abcd1234.execute-api.us-east-1.amazonaws.com',
      connectionId: 'AAAA1234=',
      apiId: 'abcd1234'
```

```
    },
    isBase64Encoded: false
  }
```

在 API Gateway 中设置 WebSocket API 集成响应

主题

- 集成响应概述
- 双向通信的集成响应
- 使用 API Gateway 控制台设置集成响应
- 使用 AWS CLI 设置集成响应

集成响应概述

API Gateway 的集成响应是一种用于对后端服务的响应进行建模和处理的方法。REST API 与 WebSocket API 集成响应的设置存在一些差别，但从概念上讲，行为是相同的。

WebSocket 路由可以配置为双向或单向通信。

- 当路由配置为双向通信时，集成响应能让您对返回的消息负载配置转换，类似于 REST API 的集成响应。
- 如果路由配置为单向通信，则无论是否有任何集成响应配置，在处理消息后都不会通过 WebSocket 通道返回响应。

除非您设置了路由响应，否则 API Gateway 不会将后端响应传递给路由响应。要了解有关设置路径响应的信息，请参阅 the section called “设置 WebSocket API 路由响应”。

双向通信的集成响应

集成可以分为代理集成和非代理集成。

⚠ Important

对于代理集成，API Gateway 会自动将后端输出作为完整的负载传递给调用方。没有集成响应。

对于非代理集成，您必须至少设置一个集成响应：

- 理想情况下，您的一个集成响应应该在无法进行明确的选择时是包罗万象的响应。此默认情况是通过设置 `$default` 的集成响应键来表示的。
- 在所有其他情况下，集成响应键起正则表达式的作用。它应遵循 `"/expression/"` 格式。

对于非代理 HTTP 集成：

- API Gateway 将尝试匹配后端响应的 HTTP 状态代码。在这种情况下，集成响应键将起正则表达式的作用。如果找不到匹配项，则会选择 `$default` 作为集成响应。
- 如上所述，模板选择表达式的作用相同。例如：
  - `/2\d\d/`：接收并转换成功响应
  - `/4\d\d/`：接收并转换不正确的请求错误
  - `$default`：接收并转换所有意外响应

有关模板选择表达式的更多信息，请参阅 the section called “模板选择表达式”。

使用 API Gateway 控制台设置集成响应

要使用 API Gateway 控制台为 WebSocket API 设置路由集成响应，请执行以下操作：

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 WebSocket API，然后选择路由。
3. 选择集成请求选项卡，然后在集成响应设置部分中，选择创建集成响应。
4. 对于响应密钥，输入一个值，该值将在评估响应选择表达式后在传出消息中的响应密钥中找到。例如，您可以输入 **`/4\d\d/`** 以接收和转换错误的请求错误，也可以输入 **`$default`** 以接收和转换与模板选择表达式匹配的所有响应。
5. 对于模板选择表达式，输入选择表达式来评估传出的消息。
6. 选择创建响应。
7. 您还可以定义映射模板来配置返回的消息有效负载的转换。选择创建模板。
8. 输入键名称。如果您选择的是默认模板选择表达式，请输入 **`\$default`**。
9. 对于响应模板，请在代码编辑器中输入您的映射模板。
10. 选择创建模板。
11. 选择部署 API 以部署您的 API。

使用以下 wscat 命令连接到您的 API。有关 `wscat` 的更多信息，请参阅 the section called “使用 `wscat` 连接到 WebSocket API 并向其发送消息”。

```
wscat -c wss://api-id.execute-api.us-east-2.amazonaws.com/test
```

当您调用路由时，应返回消息有效负载。

使用 AWS CLI 设置集成响应

要使用 AWS CLI 为 WebSocket API 设置集成响应，请调用 create-integration-response 命令：以下 CLI 命令显示创建 `$default` 集成响应的示例：

```
aws apigatewayv2 create-integration-response \
    --api-id vaz7da96z6 \
    --integration-id a1b2c3 \
    --integration-response-key '$default'
```

请求验证

您可以配置 API Gateway，使其在继续集成请求之前对路由请求执行验证。如果验证失败，API Gateway 将在不调用后端的情况下使请求失败，向客户端发送“错误请求正文”网关响应，并在 CloudWatch Logs 中发布验证结果。通过这种方式使用验证可减少对 API 后端的不必要调用。

模型选择表达式

您可以使用模型选择表达式来动态验证同一路由中的请求。如果您为代理集成或非代理集成提供模型选择表达式，则会发生模型验证。当未找到匹配模型时，您可能需要将 `$default` 模型定义为回退。如果没有匹配模型且未定义 `$default`，则验证将失败。选择表达式类似于 `Route.ModelSelectionExpression`，并计算为 `Route.RequestModels` 的键。

当您为 WebSocket API 定义路由时，可以指定（可选）模型选择表达式。将会求解此表达式以选择在收到请求时用于正文验证的模型。表达式的求值结果为路由的 `requestmodels` 中的条目之一。

模型采用 JSON 架构表示，描述了请求正文的数据结构。此选择表达式的性质能让您为特定路由，在运行时动态选择要用于对照进行验证的模型。有关如何创建模型的信息，请参阅the section called “了解数据模型”。

使用 API Gateway 控制台设置请求验证

以下示例演示了如何在路径上设置请求验证。

首先，您创建一个模型，然后创建路由。接下来，在刚创建的路径上配置请求验证。最后，您部署并测试您的 API。要完成本教程，您需要一个将 `$request.body.action` 作为路由选择表达式的 WebSocket API 和一个用于新路由的集成端点。

还需要 `wscat` 以连接到 API。有关更多信息，请参阅 the section called “使用 `wscat` 连接到 WebSocket API 并向其发送消息”。

创建模型

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择 WebSocket API。
3. 在主导航窗格中，选择模型。
4. 选择创建模型。
5. 对于名称，请输入 **`emailModel`**。
6. 对于内容类型，输入 **`application/json`**。
7. 对于模型架构，输入以下模型：

   ```
   {
       "$schema": "http://json-schema.org/draft-04/schema#",
       "type" : "object",
       "required" : [ "address"],
       "properties" : {
           "address": {
               "type": "string"
           }
       }
   }
   ```

   此模型要求请求包含电子邮件地址。
8. 选择保存。

在本步骤中，您将为 WebSocket API 创建路由。

要创建路由

1. 在主导航窗格中，选择路由。
2. 选择创建路由。

3. 对于路由键，请输入 **`sendMessage`**。
4. 选择集成类型并指定集成端点。有关更多信息，请参阅the section called “集成”。
5. 选择创建路由。

在此步骤中，您将为 `sendMessage` 路径设置请求验证。

设置请求验证

1. 在路由请求选项卡上的路由请求设置下，选择编辑。
2. 在模型选择表达式中，输入 **`${request.body.messageType}`**。

   API Gateway 使用 `messageType` 属性来验证传入的请求。
3. 选择添加请求模型。
4. 对于模型密钥，输入 **`email`**。
5. 对于模型，选择 emailModel。

   API Gateway 会根据此模型验证 `messageType` 属性设置为 `email` 的传入消息。

   > **Note**
   >
   > 如果 API Gateway 无法将模型选择表达式与模型密钥相匹配，则它会选择 `$default` 模型。如果没有 `$default` 模型，则验证将失败。对于生产 API，我们建议您创建一个 `$default` 模型。
6. 选择 Save changes（保存更改）。

在本步骤中，您将部署并测试您的 API。

部署和测试您的 API

1. 选择部署 API。
2. 从下拉列表中选择所需的阶段，或输入新阶段的名称。
3. 选择部署。
4. 在主导航窗格中，选择阶段。
5. 复制 API 的 WebSocket URL。该 URL 应类似于 `wss://abcdef123.execute-api.us-east-2.amazonaws.com/production`。

6. 打开一个新终端，并使用以下参数运行 wscat 命令。

```
wscat -c wss://abcdef123.execute-api.us-west-2.amazonaws.com/production
```

```
Connected (press CTRL+C to quit)
```

7. 使用以下命令来测试您的 API：

```
{"action": "sendMessage", "messageType": "email"}
```

```
{"message": "Invalid request body", "connectionId":"ABCD1=234",
 "requestId":"EFGH="}
```

API Gateway 将使请求失败。

使用下一个命令向您的 API 发送有效的请求。

```
{"action": "sendMessage", "messageType": "email", "address":
 "mary_major@example.com"}
```

为 WebSocket API 设置数据转换

在 API Gateway 中，WebSocket API 的方法请求采用的负载格式可能与后端所需的相应集成请求负载的格式不同。同样，后端返回的集成响应负载可能不同于前端希望的方法响应负载。

API Gateway 允许您使用映射模板将负载从方法请求映射到相应的集成请求，以及从集成响应映射到相应的方法响应。您可以指定模板选择表达式来确定执行必要的数据转换时所用的模板。

您可以使用数据映射将路由请求中的数据映射到后端集成。要了解更多信息，请参阅“the section called “数据映射””。

映射模板和模型

映射模板是一个用 Velocity 模板语言 (VTL) 表示的脚本，应用于使用 JSONPath 表达式的负载。有关 API Gateway 映射模板的更多信息，请参阅了解映射模板。

负载可以拥有符合 JSON 架构草案 4 的数据模型。您不必定义模型来创建映射模板。但是，模型可以帮助您创建模板，因为 API Gateway 根据提供的模型生成模板蓝图。有关 API Gateway 模型的更多信息，请参阅了解数据模型。

模板选择表达式

要使用映射模板转换负载，请在集成请求或集成响应中指定 WebSocket API 模板选择表达式。将会求解此表达式以确定用于将请求正文转换为集成请求正文（通过输入模板）或将响应正文转换为路由响应正文（通过输出模板）的输入或输出模板（如果有）。

`Integration.TemplateSelectionExpression` 支持 `${request.body.jsonPath}` 和静态值。

`IntegrationResponse.TemplateSelectionExpression` 支持 `${request.body.jsonPath}`、`${integration.response.statuscode}`、`${integration.respo`
和静态值。

集成响应选择表达式

当您为 WebSocket API 设置集成响应时，您可以指定（可选）集成响应选择表达式。此表达式确定在集成返回时应选择什么 `IntegrationResponse`。此表达式的值当前受 API Gateway 限制，定义如下。认识到此表达式只与非代理集成相关；代理集成无需建模或修改便会将响应负载传递回调用方。

与前面的其他选择表达式不同，此表达式当前支持模式匹配 格式。表达式应该用正斜杠包装起来。

目前，该值是固定的，具体取决于 `integrationType`：

- 对于基于 Lambda 的集成，它是 `$integration.response.body.errorMessage`。
- 对于 `HTTP` 和 `MOCK` 集成，它是 `$integration.response.statuscode`。
- 对于 `HTTP_PROXY` 和 `AWS_PROXY`，不会使用此表达式，因为您请求将负载传递给调用方。

为 WebSocket API 设置数据映射

数据映射 使您能够将数据从路由请求映射到后端集成。

> Note
>
> 中不支持 WebSocket API 的数据映射AWS Management Console 必须使用 AWS CLI、AWS CloudFormation 或开发工具包配置数据映射。

主题

- 将路由请求数据映射至集成请求参数

- 示例

将路由请求数据映射至集成请求参数

可以从任何定义的路由请求参数、请求正文、context 或 stage 变量以及静态值映射集成请求参数。

在下表中，*PARAM_NAME* 是给定参数类型的路由请求参数的名称。它必须匹配正则表达式 `'^[a-zA-Z0-9._$-]+$]'`。*JSONPath_EXPRESSION* 是请求正文的 JSON 字段的 JSONPath 表达式。

集成请求数据映射表达式

映射数据源	映射表达式
请求查询字符串（仅对于 $connect 路由才支持）	`route.request.querystring.` *PARAM_NAME*
请求标头（仅对于 $connect 路由才支持）	`route.request.header.` *PARAM_NAME*
多值请求查询字符串（仅对于 $connect 路由才支持）	`route.request.multivaluequerystring.` *PARAM_NAME*
多值请求标头（仅对于 $connect 路由才支持）	`route.request.multivalueheader.` *PARAM_NAME*
请求正文	`route.request.body.` *JSONPath_EXPRESSION*
阶段变量	`stageVariables.` *VARIABLE_NAME*
上下文变量	`context.`*VARIABLE_NAME* ，必须为受支持的上下文变量之一。
静态值	*'STATIC_VALUE'* 。*STATIC_VALUE* 为字符串文本值，必须括在单引号内。

示例

以下 AWS CLI 示例配置数据映射。有关示例 AWS CloudFormation 模板，请参阅 websocket-data-mapping.yaml。

将客户端的 connectionId 映射到集成请求中的标头

以下示例命令将客户端的 `connectionId` 映射到后端集成请求中的 `connectionId` 标头。

```
aws apigatewayv2 update-integration \
    --integration-id abc123 \
    --api-id a1b2c3d4 \
    --request-parameters
 'integration.request.header.connectionId'='context.connectionId'
```

将查询字符串参数映射到集成请求中的标头

以下示例命令将 `authToken` 查询字符串参数映射到集成请求中的 `authToken` 标头。

首先，将 `authToken` 查询字符串参数添加到路由的请求参数。

```
aws apigatewayv2 update-route --route-id 0abcdef \
    --api-id a1b2c3d4 \
    --request-parameters '{"route.request.querystring.authToken": {"Required": false}}'
```

接下来，将查询字符串参数映射到后端集成请求中的 `authToken` 标头。

```
aws apigatewayv2 update-integration \
    --integration-id abc123 \
    --api-id a1b2c3d4 \
    --request-parameters
 'integration.request.header.authToken'='route.request.querystring.authToken'
```

如有必要，请从路由的请求参数中删除 `authToken` 查询字符串参数。

```
aws apigatewayv2 delete-route-request-parameter \
    --route-id 0abcdef \
    --api-id a1b2c3d4 \
    --request-parameter-key 'route.request.querystring.authToken'
```

API Gateway WebSocket API 映射模板参考

本节总结了 API Gateway 中 WebSocket API 当前支持的变量集。

参数	说明
`$context.connectionId`	连接的唯一 ID，可用于对客户端进行回调。
`$context.connectedAt`	Epoch 格式的连接时间。
`$context.domainName`	WebSocket API 的域名。这可用于对客户端进行回调（而不是硬编码值）。
`$context.eventType`	事件类型：`CONNECT`、`MESSAGE` 或 `DISCONNECT`。
`$context.messageId`	消息的唯一服务器端 ID。仅当 `$context.eventType` 为 `MESSAGE` 时才可用。
`$context.routeKey`	选定的路由键。
`$context.requestId`	与 `$context.extendedRequestId` 相同。
`$context.extendedRequestId`	为 API 调用自动生成的 ID，其中包含用于调试/故障排除的更有用的信息。
`$context.apiId`	API Gateway 分配给您的 API 的标识符。
`$context.authorizer.principalId`	与由客户端发送的令牌关联并从 API Gateway Lambda 授权方（以前称为自定义授权方）Lambda 函数返回的委托人用户标识。
`$context.authorizer.`*property*	从 API Gateway Lambda 授权方函数返回的 `context` 映射的指定键/值对的字符串化值。例如，如果授权方返回以下 `context` 映射：

```
"context" : {
  "key": "value",
  "numKey": 1,
  "boolKey": true
}
```

参数	说明
	调用 `$context.authorizer.key` 将返回 "value" 字符串，调用 `$context.authorizer.numKey` 将返回 "1" 字符串，而调用 `$context.authorizer.boolKey` 将返回 "true" 字符串。
`$context.error.messageString`	`$context.error.message` 的带引号的值，即 "`$context.error.message`"。
`$context.error.validationErrorString`	包含详细验证错误消息的字符串。
`$context.identity.accountId`	与请求关联的 AWS 账户 ID。
`$context.identity.apiKey`	API 所有者密钥与启用密钥的 API 请求关联。
`$context.identity.apiKeyId`	API 密钥 ID 与启用密钥的 API 请求关联
`$context.identity.caller`	发出请求的调用方的委托人标识符。
`$context.identity.cognitoAuthenticationProvider`	发出请求的调用方使用的 Amazon Cognito 身份验证提供商的逗号分隔列表。仅当使用 Amazon Cognito 凭证对请求签名时才可用。 例如，对于 Amazon Cognito 身份池中的身份，`cognito-idp.`*`region`*`.amazonaws.com/`*`user_pool_id`*`,cognito-idp.`*`region`*`.amazonaws.com/`*`user_pool_id`*`:CognitoSignIn:`*`token subject claim`* 有关更多信息，请参阅 Amazon Cognito 开发人员指南 中的使用联合身份。

参数	说明
`$context.identity.cognitoAuthenticationType`	发出请求的调用方的 Amazon Cognito 身份验证类型。仅当使用 Amazon Cognito 凭证对请求签名时才可用。可能的值包括经过身份验证的身份的 `authenticated` 和未经身份验证的身份的 `unauthenticated` 。
`$context.identity.cognitoIdentityId`	发出请求的调用方的 Amazon Cognito 身份 ID。仅当使用 Amazon Cognito 凭证对请求签名时才可用。
`$context.identity.cognitoIdentityPoolId`	发出请求的调用方的 Amazon Cognito 身份池 ID。仅当使用 Amazon Cognito 凭证对请求签名时才可用。
`$context.identity.sourceIp`	向 API Gateway 终端节点发出请求的即时 TCP 连接的源 IP 地址。
`$context.identity.user`	发出请求的用户的委托人标识符。
`$context.identity.userAgent`	API 调用方的用户代理。
`$context.identity.userArn`	身份验证后标识的有效用户的 Amazon Resource Name (ARN)。
`$context.requestTime`	CLF 格式的请求时间 (`dd/MMM/yyyy:HH:mm:ss +-hhmm`)。
`$context.requestTimeEpoch`	Epoch 格式的请求时间，以毫秒为单位。
`$context.stage`	API 调用的部署阶段（例如测试或生产）。
`$context.status`	响应状态。
`$input.body`	以字符串形式返回原始负载。

参数	说明
`$input.json(x)`	此函数计算 JSONPath 表达式并以 JSON 字符串形式返回结果。 例如，`$input.json('$.pets')` 将返回一个表示宠物结构的 JSON 字符串。 有关 JSONPath 的更多信息，请参阅 JSONPath 或适用于 Java 的 JSONPath。

参数	说明
`$input.path(x)`	获取一个 JSONPath 表达式字符串 (x) 并返回结果的 JSON 对象表达式。这样，您便可通过 [Apache Velocity 模板语言 (VTL)](#) 在本机访问和操作负载的元素。 例如，如果表达式 `$input.path('$.pets')` 返回一个如下所示的对象：

```
[
  {
    "id": 1,
    "type": "dog",
    "price": 249.99
  },
  {
    "id": 2,
    "type": "cat",
    "price": 124.99
  },
  {
    "id": 3,
    "type": "fish",
    "price": 0.99
  }
]
```

`$input.path('$.pets').count()` 将返回 `"3"`。

有关 JSONPath 的更多信息，请参阅 [JSONPath](#) 或[适用于 Java 的 JSONPath](#)。

参数	说明
`$stageVariables.` *<variable_name>*	*<variable_name>* 表示阶段变量名称。
`$stageVariables['` *<variable_name>* `']`	*<variable_name>* 表示任何阶段变量名称。

参数	说明
`${stageVariables['<variable_name>']}`	*<variable_name>* 表示任何阶段变量名称。
`$util.escapeJavaScript()`	使用 JavaScript 字符串规则对字符串中的字符进行转义。 **Note** 此函数会将任何常规单引号 (') 变成转义单引号 (\\')。但是，转义单引号在 JSON 中无效。因此，当此函数的输出用于 JSON 属性时，必须将任何转义单引号 (\\') 变回常规单引号 (')。如下例所示: `$util.escapeJavaScript( data).replaceAll("\\'","'")`

参数	说明
`$util.parseJson()`	获取“字符串化的”JSON 并返回结果的对象表示形式。您可以使用此函数的结果通过 Apache Velocity 模板语言 (VTL) 在本机访问和操作负载的元素。例如，如果您具有以下负载：

```
{"errorMessage":"{\"key1\":\"var1\",
\"key2\":{\"arr\":[1,2,3]}}"}
```

并使用以下映射模板

```
#set ($errorMessageObj = $util.par
seJson($input.path('$.error
Message')))
{
   "errorMessageObjKey2ArrVal" :
 $errorMessageObj.key2.arr[0]
}
```

您将得到以下输出：

```
{
   "errorMessageObjKey2ArrVal" : 1
}
```

参数	说明
`$util.urlEncode()`	将字符串转换为“application/x-www-form-urlencoded”格式。
`$util.urlDecode()`	对“application/x-www-form-urlencoded”字符串进行解码。
`$util.base64Encode()`	将数据编码为 base64 编码的字符串。
`$util.base64Decode()`	对 base64 编码字符串中的数据进行解码。

为 WebSocket API 使用二进制媒体类型

API Gateway WebSocket API 目前在传入消息负载中不支持二进制帧。如果客户端应用程序发送二进制帧，API Gateway 会拒绝它并断开客户端，而且显示代码 1003。

此行为有一种解决方法。如果客户端发送文本编码二进制数据（例如，Base64）作为文本帧，您可以将集成的 `contentHandlingStrategy` 属性设置为 `CONVERT_TO_BINARY`，以将负载从 Base64 编码的字符串转换为二进制。

要在非代理集成中返回二进制负载的路由响应，您可以将集成响应的 `contentHandlingStrategy` 属性设置为 `CONVERT_TO_TEXT`，以将负载从二进制转换为 Base64 编码字符串。

调用 WebSocket API

部署 WebSocket API 后，客户端应用程序可以连接并向其发送消息，而且您的后端服务可以向连接的客户端应用程序发送消息：

- 您可以使用 `wscat` 连接到 WebSocket API 并向其发送消息以模拟客户端行为。请参阅the section called “使用 `wscat` 连接到 WebSocket API 并向其发送消息”。
- 您可以使用后端服务中的 @connections API 向连接的客户端发送回调消息、获取连接信息或断开客户端连接。请参阅the section called “在后端服务中使用 `@connections` 命令”。
- 客户端应用程序可以使用自己的 WebSocket 库来调用您的 WebSocket API。

使用 **`wscat`** 连接到 WebSocket API 并向其发送消息

`wscat` 实用程序是一个方便的工具，用于测试您已在 API Gateway 中创建和部署的 WebSocket API。您可以按如下方式安装和使用 `wscat`：

1. 从 https://www.npmjs.com/package/wscat 中下载 `wscat`。
2. 通过运行以下命令安装 `wscat`：

   ```
   npm install -g wscat
   ```

3. 要连接到 API，请运行 `wscat` 命令，如以下示例所示。请注意，此示例假定 `Authorization` 设置是 `NONE`。

   ```
   wscat -c wss://aabbccddee.execute-api.us-east-1.amazonaws.com/test/
   ```

您需要将 *aabbccddee* 替换为实际的 API ID，该 ID 显示在 API Gateway 控制台中或由 AWS CLI `create-api` 命令返回。

此外，如果您的 API 位于 `us-east-1` 以外的区域，则需要替换正确的区域。

4. 要测试您的 API，请在连接时输入以下消息：

```
{"{jsonpath-expression}":"{route-key}"}
```

其中 *{jsonpath-expression}* 是一个 JSONPath 表达式，*{route-key}* 是 API 的路由键。例如：

```
{"action":"action1"}
{"message":"test response body"}
```

有关 JSONPath 的更多信息，请参阅 JSONPath 或适用于 Java 的 JSONPath。

5. 要从 API 断开连接，请输入 `ctrl-C`。

在后端服务中使用 `@connections` 命令

您的后端服务可以使用以下 WebSocket 连接 HTTP 请求向连接的客户端发送回调消息、获取连接信息或断开客户端连接。

> ⚠ Important
>
> 这些请求使用 IAM 授权，因此您必须使用签名版本 4 (SigV4) 对其进行签名。为此，您可以使用 API Gateway 管理 API。有关更多信息，请参阅 ApiGatewayManagementApi。

在以下命令中，您需要将 *{api-id}* 替换为实际的 API ID，该 ID 显示在 API Gateway 控制台中或由 AWS CLI create-api 命令返回。在使用此命令之前，必须先建立连接。

要向客户端发送回调消息，请使用：

```
POST https://{api-id}.execute-api.us-east-1.amazonaws.com/{stage}/
@connections/{connection_id}
```

您可以通过使用 `Postman` 或通过调用 `awscurl` 来测试此请求，如以下示例所示：

```
awscurl --service execute-api -X POST -d "hello world" https://{prefix}.execute-api.us-east-1.amazonaws.com/{stage}/@connections/{connection_id}
```

您需要对命令进行 URL 编码，如以下示例所示：

```
awscurl --service execute-api -X POST -d "hello world" https://aabbccddee.execute-api.us-east-1.amazonaws.com/prod/%40connections/R0oXAdfD0kwCH6w%3D
```

要获取客户端的最新连接状态，请使用：

```
GET https://{api-id}.execute-api.us-east-1.amazonaws.com/{stage}/
@connections/{connection_id}
```

要断开客户端连接，请使用：

```
DELETE https://{api-id}.execute-api.us-east-1.amazonaws.com/{stage}/
@connections/{connection_id}
```

您可以通过在集成中使用 `$context` 变量来动态构建回调 URL。例如，如果您将 Lambda 代理集成与 `Node.js` Lambda 函数一起使用，则可以按如下方式构建 URL 并向连接的客户端发送消息：

```
import {
  ApiGatewayManagementApiClient,
  PostToConnectionCommand,
} from "@aws-sdk/client-apigatewaymanagementapi";

export const handler = async (event) => {
  const domain = event.requestContext.domainName;
  const stage = event.requestContext.stage;
  const connectionId = event.requestContext.connectionId;
  const callbackUrl = `https://${domain}/${stage}`;
  const client = new ApiGatewayManagementApiClient({ endpoint: callbackUrl });

  const requestParams = {
    ConnectionId: connectionId,
    Data: "Hello!",
  };

  const command = new PostToConnectionCommand(requestParams);
```

```
  try {
    await client.send(command);
  } catch (error) {
    console.log(error);
  }

  return {
    statusCode: 200,
  };
};
```

发送回调消息时，您的 Lambda 函数必须有权调用 API Gateway 管理 API。如果您在连接建立之前或客户端断开连接后发布消息，则可能会收到一条包含 `GoneException` 的错误。

发布 WebSocket API 以供客户调用

仅仅创建和开发 API Gateway API 并不会自动使其可供用户调用。要使 API 可供调用，您必须将其部署到一个阶段。此外，建议您自定义用户将用于访问 API 的 URL。您可以为其提供一个与品牌一致的域，或者使其比 API 的默认 URL 更容易记住。

在本节中，您可以了解如何部署 API 并自定义您提供给用户以访问 API 的 URL。

> **Note**
>
> 为了增强您的 API Gateway API 的安全性，将在公共后缀列表 (PSL) 中注册 `execute-api.{region}.amazonaws.com` 域。为进一步增强安全性，如果您需要在 API Gateway API 的默认域名中设置敏感 Cookie，我们建议您使用带 `__Host-` 前缀的 Cookie。这将有助于保护您的域，防范跨站点请求伪造 (CSRF) 攻击。要了解更多信息，请参阅 Mozilla 开发者网络中的 Set-Cookie 页面。

主题

- 使用 WebSocket API 阶段
- 在 API Gateway 中部署 WebSocket API
- WebSocket API 的安全策略
- 为 WebSocket API 设置自定义域名

使用 WebSocket API 阶段

一个 API 阶段是对您 API 生命周期状态（例如，`dev`、`prod`、`beta` 或 `v2`）的一次逻辑引用。API 阶段通过 API ID 和阶段名称标识，包含在您用于调用 API 的 URL 中。每个阶段都是一个对 API 部署的命名引用，可供客户端应用程序调用。

部署是 API 配置的快照。将 API 部署到阶段后，客户端可以调用该 API。您必须部署 API 才能使更改生效。

阶段变量

阶段变量是您可以为 WebSocket API 的阶段定义的键/值对。它们与环境变量的功能类似，可用于 API 设置。

例如，您可以定义阶段变量，然后将其值设置为某个 HTTP 代理集成的 HTTP 终端节点。稍后，您可以使用关联的阶段变量名称引用终端节点。通过执行此操作，您可以在各个阶段对不同终端节点使用相同的 API 设置。同样，您可以使用阶段变量，为 API 的各个阶段指定不同的 AWS Lambda 函数集成。

> **Note**
>
> 阶段变量不适用于敏感数据，例如凭证。要将敏感数据传递给集成，请使用 AWS Lambda 授权方。您可以在 Lambda 授权方的输出中将敏感数据传递给集成。要了解更多信息，请参阅“the section called “Lambda 授权方响应格式””。

示例

要使用阶段变量自定义 HTTP 集成终端节点，您必须首先设置阶段变量的名称和值（例如 `url`，值为 `example.com`）。接下来，设置 HTTP 代理集成。您可以告知 API Gateway 使用阶段变量值 **`http://${stageVariables.url}`**，而不是输入终端节点的 URL。此值将指示 API Gateway 在运行时替换您的阶段变量 `${}`，具体取决于 API 所处的阶段。

您可以通过类似的方式引用阶段变量，用于指定 Lambda 函数名称或AWS角色 ARN。

将 Lambda 函数名称指定为阶段变量值时，您必须手动配置对 Lambda 函数的权限。您可以使用 AWS Command Line Interface (AWS CLI) 来执行此操作。

```
aws lambda add-permission --function-name arn:aws:lambda:XXXXXX:your-lambda-function-
name --source-arn arn:aws:execute-api:us-east-1:YOUR_ACCOUNT_ID:api_id/*/HTTP_METHOD/
```

```
resource --principal apigateway.amazonaws.com --statement-id apigateway-access --action
 lambda:InvokeFunction
```

API Gateway 阶段变量参考

HTTP 集成 URI

您可将阶段变量用作 HTTP 集成 URI 的一部分，如以下示例所示。

- 不带协议的完整 URI – `http://${stageVariables.<variable_name>}`
- 完整域 – `http://${stageVariables.<variable_name>}/resource/operation`
- 子域 – `http://${stageVariables.<variable_name>}.example.com/resource/operation`
- 路径 – `http://example.com/${stageVariables.<variable_name>}/bar`
- 查询字符串 – `http://example.com/foo?q=${stageVariables.<variable_name>}`

Lambda 函数

您可以使用阶段变量代替 Lambda 函数名称或别名，如以下示例所示。

- `arn:aws:apigateway:<region>:lambda:path/2015-03-31/functions/arn:aws:lambda:<region>:<account_id>:function:${stageVariables.<function_variable_name>}/invocations`
- `arn:aws:apigateway:<region>:lambda:path/2015-03-31/functions/arn:aws:lambda:<region>:<account_id>:function:<function_name>:${stageVariables.<version_variable_name>}/invocations`

> **Note**
>
> 要将阶段变量用于 Lambda 函数，该函数必须与 API 位于同一账户中。阶段变量不支持跨账户 Lambda 函数。

AWS 集成凭证

您可以在 AWS 用户或角色凭证 ARN 中使用阶段变量，如以下示例所示。

- `arn:aws:iam::<account_id>:${stageVariables.<variable_name>}`

在 API Gateway 中部署 WebSocket API

在创建 WebSocket API 之后，您必须对其进行部署，以便您的用户可以调用它。

要部署 API，您可以创建 API 部署并将其与阶段关联。每个阶段都是 API 的一个快照，可供客户端应用程序调用。

> ⚠ Important
>
> 每次更新 API 后，都必须重新部署 API。对阶段设置以外的任何内容进行更改都需要重新部署，包括修改以下资源：
>
> - 路线
> - 集成
> - 授权方
>
> 默认情况下，您将受到每个 API 10 个阶段的限制。我们建议您为部署重用阶段。

为调用已部署的 WebSocket API，客户端会向 API 的 URL 发送消息。URL 由 API 的主机名和阶段名称确定。

> ⓘ Note
>
> API Gateway 将支持最大 128 KB 的负载，最大帧大小为 32 KB。如果消息超过 32 KB，则必须将其拆分为多个帧，每个 32 KB 或更小。

使用 API 的默认域名，给定阶段 (*{stageName}*) 中的 WebSocket API 的 URL（例如）采用以下格式：

```
wss://{api-id}.execute-api.{region}.amazonaws.com/{stageName}
```

要使 WebSocket API 的 URL 对用户更友好，您可以创建自定义域名（如 `api.example.com`）来替换该 API 的默认主机名。配置过程与 REST API 相同。有关更多信息，请参阅 the section called “自定义域名”。

阶段实现了对 API 的可靠版本控制。例如，您可以将 API 部署到 `test` 阶段和 `prod` 阶段，并使用 `test` 阶段作为测试版本，使用 `prod` 阶段作为稳定版本。更新通过测试之后，您可以将 `test` 阶段提升到 `prod` 阶段。可以通过将 API 重新部署到 `prod` 阶段来完成升级。有关阶段的详细信息，请参阅the section called “设置阶段”。

主题

- 使用 AWS CLI 创建 WebSocket API 部署
- 使用 API Gateway 控制台创建 WebSocket API 部署

使用 AWS CLI 创建 WebSocket API 部署

要使用 AWS CLI 来创建部署，请使用 create-deployment 命令，如以下示例所示：

```
aws apigatewayv2 --region us-east-1 create-deployment --api-id aabbccddee
```

输出示例：

```
{
    "DeploymentId": "fedcba",
    "DeploymentStatus": "DEPLOYED",
    "CreatedDate": "2018-11-15T06:49:09Z"
}
```

在您将此部署与阶段关联之前，部署的 API 不可调用。您可以创建新阶段或重用之前创建的阶段。

要创建新的阶段并将其与部署关联，请使用 create-stage 命令，如以下示例所示：

```
aws apigatewayv2 --region us-east-1 create-stage --api-id aabbccddee --deployment-id
 fedcba --stage-name test
```

输出示例：

```
{
    "StageName": "test",
    "CreatedDate": "2018-11-15T06:50:28Z",
    "DeploymentId": "fedcba",
    "DefaultRouteSettings": {
        "MetricsEnabled": false,
        "ThrottlingBurstLimit": 5000,
        "DataTraceEnabled": false,
```

```
        "ThrottlingRateLimit": 10000.0
    },
    "LastUpdatedDate": "2018-11-15T06:50:28Z",
    "StageVariables": {},
    "RouteSettings": {}
}
```

要重用现有阶段，请使用 update-stage 命令，利用新创建的部署 ID (*{deployment-id}*) 更新阶段的 `deploymentId` 属性。

```
aws apigatewayv2 update-stage --region {region} \
   --api-id {api-id} \
   --stage-name {stage-name} \
   --deployment-id {deployment-id}
```

使用 API Gateway 控制台创建 WebSocket API 部署

要使用 API Gateway 控制台为 WebSocket API 创建部署，请执行以下操作：

1. 登录 API Gateway 控制台并选择 API。
2. 选择部署 API。
3. 从下拉列表中选择所需的阶段，或输入新阶段的名称。

WebSocket API 的安全策略

API Gateway 对所有 WebSocket API 端点强制执行 TLS_1_2 的安全策略。

安全策略 是 Amazon API Gateway 提供的最低 TLS 版本和密码套件的预定义组合。TLS 协议解决了网络安全问题，例如客户端和服务器之间的篡改和窃听。当您的客户端通过自定义域建立与您 API 的 TLS 握手时，安全策略实施客户端可以选择使用的 TLS 版本和密码套件选项。此安全策略接受 TLS 1.2 和 TLS 1.3 流量并拒绝 TLS 1.0 流量。

WebSocket API 支持的 TLS 协议和密码

下表描述了 WebSocket API 支持的 TLS 协议和密码。

安全策略	TLS_1_2
TLS 协议	

安全策略	TLS_1_2
TLSv1.3	♦
TLSv1.2	♦
TLS 密码	
TLS_AES_128_GCM_SHA256	♦
TLS_AES_256_GCM_SHA384	♦
TLS_CHACHA20_POLY1305_SHA256	♦
ECDHE-ECDSA-AES128- GCM-SHA256	♦
ECDHE-RSA-AES128- GCM-SHA256	♦
ECDHE-ECDSA-AES128-SHA256	♦
ECDHE-RSA-AES128-SHA256	♦
ECDHE-ECDSA-AES256- GCM-SHA384	♦
ECDHE-RSA-AES256- GCM-SHA384	♦
ECDHE-ECDSA-AES256-SHA384	♦
ECDHE-RSA-AES256-SHA384	♦
AES128-GCM-SHA256	♦
AES128-SHA256	♦
AES256-GCM-SHA384	♦
AES256-SHA256	♦

OpenSSL 和 RFC 密码名称

OpenSSL 和 IETF RFC 5246 为相同的密码使用不同的名称。有关密码名称的列表，请参阅the section called “OpenSSL 和 RFC 密码名称”。

有关 REST API 和 HTTP API 的信息

有关 REST API 和 HTTP API 的更多信息，请参阅the section called “选择安全策略”和the section called “HTTP API 的安全策略”。

为 WebSocket API 设置自定义域名

自定义域名 是您可以提供给 API 用户的更简单、更直观的 URL。

部署 API 后，您（和您的客户）可以使用以下格式的默认基本 URL 调用 API：

```
https://api-id.execute-api.region.amazonaws.com/stage
```

其中 `api-id` 由 API Gateway 生成，`region`（AWS 区域）由您在创建 API 时指定，`stage` 由您在部署 API 时指定。

URL 的主机名部分（即 `api-id.execute-api.region.amazonaws.com`）是指 API 端点。默认 API 端点难于重新调用，对用户不友好。

使用自定义域名，您可以设置 API 的主机名，并选择基本路径（例如 `myservice`）以将备用 URL 映射到 API。例如，一个更为用户友好的 API 基本 URL 可以变成：

```
https://api.example.com/myservice
```

> **Note**
>
> 无法将 WebSocket API 的自定义域名映射到 REST API 或 HTTP API。
> 对于 WebSocket API，支持区域自定义域名。
> 对于 WebSocket API，TLS 1.2 是唯一受支持的 TLS 版本。

注册域名

您必须拥有已注册的 Internet 域名，以便为 API 设置自定义域名。域名必须遵循 RFC 1035 规范，每个标签最多可以有 63 个八位字节，总共可以有 255 个八位字节。如果需要，您可以使用 Amazon Route 53 或使用您选择的第三方域注册商注册互联网域。API 的自定义域名可以是已注册 Internet 域的子域或根域（也称为“顶级域”）的名称。

在 API Gateway 中创建自定义域名后，您必须创建或更新 DNS 提供商的资源记录以映射到 API 端点。如果没有此类映射，针对自定义域名的 API 请求无法到达 API Gateway。

区域自定义域名

为区域 API 创建自定义域名时，API Gateway 为 API 创建区域域名。您必须设置将自定义域名映射到区域域名的 DNS 记录。您还必须为自定义域名提供证书。

通配符自定义域名

使用通配符自定义域名，您可以在不超过默认配额的情况下支持几乎无限数量的域名。例如，您可以为每位客户提供自己的域名 *customername*`.api.example.com`。

要创建通配符自定义域名，可以指定通配符 (`*`) 作为表示根域所有可能子域的自定义域的第一个子域。

例如，通配符自定义域名 `*.example.com` 会生成子域，如 `a.example.com`、`b.example.com` 和 `c.example.com`，这些子域都会路由到同一个域。

通配符自定义域名支持与 API Gateway 的标准自定义域名不同的配置。例如，在单个 AWS 账户中，您可以对 `*.example.com` 和 `a.example.com` 进行不同的配置。

您可以使用 `$context.domainName` 和 `$context.domainPrefix` 上下文变量来确定客户端用于调用 API 的域名。要了解有关上下文变量的更多信息，请参阅 API Gateway 映射模板和访问日志记录变量引用。

要创建通配符自定义域名，您必须提供已使用 DNS 或电子邮件验证方法验证的由 ACM 颁发的证书。

> **Note**
>
> 如果其他 AWS 账户已经创建了与通配符自定义域名冲突的自定义域名，则无法创建通配符自定义域名。例如，如果账户 A 已经创建了 `a.example.com`，则账户 B 无法创建通配符自定义域名 `*.example.com`。
> 如果账户 A 和账户 B 共享拥有者，您可以联系 AWS Support 中心请求例外。

自定义域名的证书

> **Important**
>
> 您为您的自定义域名配置了证书。如果您的应用程序使用证书固定（有时称为 SSL 固定）来固定 ACM 证书，则在 AWS 续订证书后，应用程序可能无法连接到您的域。有关更多信息，请参阅《AWS Certificate Manager 用户指南》中的证书固定问题。

要为支持 ACM 的区域中的自定义域名提供证书，您必须从 ACM 请求证书。要为不支持 ACM 的区域中的区域自定义域名提供证书，您必须在该区域中将证书导入到 API Gateway。

要导入 SSL/TLS 证书，您必须针对自定义域名提供 PEM 格式的 SSL/TLS 证书文本、其私有密钥和证书链。存储在 ACM 中的每个证书均由其 ARN 标识。要针对域名使用 AWS 托管的证书，您只需参考其 ARN 即可。

通过 ACM 可以轻松地为 API 设置和使用自定义域名。您可以为给定的域名创建证书（或导入证书），使用 ACM 提供的证书的 ARN 在 API Gateway 中设置域名，然后将自定义域名下的基本路径映射到 API 的已部署阶段。如果拥有 ACM 颁发的证书，那么您就无需担心公开任何敏感的证书详细信息，如私有密钥。

设置自定义域名

有关设置自定义域名的详细信息，请参阅在 中准备好证书AWS Certificate Manager和在 API Gateway 中设置区域自定义域名。

对 WebSocket API 使用 API 映射

您可以使用 API 映射将 API 阶段连接到自定义域名。创建域名并配置 DNS 记录后，您可以使用 API 映射通过自定义域名向 API 发送流量。

API 映射指定了用于映射的 API、阶段以及可选的路径。例如，您可以将 API 的 `production` 阶段映射到 `wss://api.example.com/orders`。

在创建 API 映射之前，您必须拥有 API、阶段和自定义域名。要了解有关创建自定义域名的更多信息，请参阅 the section called “设置区域自定义域名”。

限制

- 在 API 映射中，自定义域名和映射的 API 必须位于同一个AWS账户中。
- API 映射必须仅包含字母、数字和以下字符：`$-_.+!*'()`。
- API 映射中路径的最大长度为 300 个字符。
- 您不能将 WebSocket API 映射到与 HTTP API 或 REST API 相同的自定义域名。

创建 API 映射

要创建 API 映射，您必须首先创建自定义域名、API 和阶段。有关使用自定义域名的更多信息，请参阅 the section called “设置区域自定义域名”。

AWS Management Console

创建 API 映射

1. 通过以下网址登录到 Amazon API Gateway 控制台：https://console.aws.amazon.com/apigateway。
2. 选择自定义域名。
3. 选择您已经创建的自定义域名。
4. 选择 API 映射。
5. 选择 Configure API mappings (配置 API 映射)。
6. 选择 Add new mapping (添加新映射)。
7. 输入 API、阶段以及可选的路径。
8. 选择 Save。

AWS CLI

以下 AWS CLI 命令创建一个 API 映射。在此示例中，API Gateway 将请求发送到 `api.example.com/v1`，到指定的 API 和阶段。

```
aws apigatewayv2 create-api-mapping \
    --domain-name api.example.com \
    --api-mapping-key v1 \
    --api-id a1b2c3d4 \
    --stage test
```

AWS CloudFormation

以下 AWS CloudFormation 示例会创建一个 API 映射。

```
MyApiMapping:
  Type: 'AWS::ApiGatewayV2::ApiMapping'
  Properties:
    DomainName: api.example.com
    ApiMappingKey: 'v1'
    ApiId: !Ref MyApi
    Stage: !Ref MyStage
```

禁用 WebSocket API 的默认终端节点

默认情况下，客户端可以通过使用 API Gateway 为 API 生成的 `execute-api` 终端节点来调用您的 API。为确保客户端只能通过使用自定义域名访问您的 API，请禁用默认 `execute-api` 终端节点。

> **Note**
>
> 禁用默认终端节点时，它会影响 API 的所有阶段。

以下 AWS CLI 命令会禁用 WebSocket API 的默认终端节点。

```
aws apigatewayv2 update-api \
    --api-id abcdef123 \
    --disable-execute-api-endpoint
```

禁用默认终端节点后，必须部署 API 才能使更改生效。

以下 AWS CLI 命令会创建部署。

```
aws apigatewayv2 create-deployment \
    --api-id abcdef123 \
    --stage-name dev
```

保护您的 WebSocket API

您可以为 API 配置节流，以帮助防止它们因请求过多而不堪重负。节流是在尽最大努力的基础上应用的，应被视为目标而不是保证的请求上限。

API Gateway 使用令牌桶算法（其中，一个令牌即一个请求）限制对 API 的请求。具体来说，API Gateway 根据您账户中的所有 API，按区域检查请求提交的速率和突发事件。在令牌桶算法中，突发可以允许这些限制的预定义超出，但在某些情况下，其他因素也可能导致限制超支。

如果请求提交超过稳态请求速率和突增限制，则 API Gateway 将开始限制请求。此时客户可能会收到 `429 Too Many Requests` 个错误响应。捕获此类异常后，客户端能够以限制速率的方式重新提交失败的请求。

作为 API 开发人员，您可以针对各个 API 阶段或方法设置目标限制，以提高账户中所有 API 的整体性能。

每个区域的账户级别限制

默认情况下，API Gateway 针对每个区域限制 AWS 账户内所有 API 的每秒稳态请求 (RPS)。它还对于每个区域限制一个 AWS 账户中所有 API 的突增（即最大存储桶大小）。在 API Gateway 中，突增限制代表 API Gateway 在返回 `429 Too Many Requests` 错误响应之前可以完成目标的最大并发请求提交数量。有关限制配额的更多信息，请参阅配额和重要提示。

每个账户限制适用于指定区域内账户中的所有 API。客户可以请求我们放宽账户级别的速率限制 —— 如果具有更短的超时和较小的有效负载的 API，则可以提高限制。要请求增加每个区域的账户级别限制，请联系 AWS Support 中心。有关更多信息，请参阅 配额和重要提示。请注意，这些限制不能高于 AWS 节流限制。

路由级别限制

您可以设置路由级别限制，用于覆盖 API 中特定阶段或各个路由的账户级别请求限制。原定设置的路由节流限制不能超过账户级别的费率限制。

您可以使用 AWS CLI 配置路由级限制。以下命令为 API 的指定阶段和路由配置自定义限制。

```
aws apigatewayv2 update-stage \
    --api-id a1b2c3d4 \
    --stage-name dev \
    --route-settings '{"messages":
{"ThrottlingBurstLimit":100,"ThrottlingRateLimit":2000}}'
```

监控 WebSocket API

您可以使用 CloudWatch 指标和 CloudWatch Logs 来监控 WebSocket API。通过合并日志和指标，您可以记录错误并监控 API 的性能。

> **Note**
>
> 在以下情况下，API Gateway 可能无法生成日志和指标：
>
> - “413 请求实体过大”错误
> - 过多的“429 请求太多”错误
> - 发送到没有 API 映射的自定义域的请求中出现 400 系列错误
> - 由内部故障造成的 500 系列错误

主题

- 使用 CloudWatch 指标监控 WebSocket API 执行
- 配置 WebSocket API 的日志记录

使用 CloudWatch 指标监控 WebSocket API 执行

您可以使用 Amazon CloudWatch 指标来监控 WebSocket API。该配置类似于 REST API 使用的配置。有关更多信息，请参阅 使用 Amazon CloudWatch 指标监控 REST API 执行。

WebSocket API 支持以下指标：

指标	说明
ConnectCount	发送到 `$connect` 路由集成的消息数。
MessageCount	从客户端发送到 WebSocket API 的消息数。
IntegrationError	从集成返回 4XX/5XX 响应的请求数。
ClientError	在调用集成之前由 API Gateway 返回 4XX 响应的请求数。
ExecutionError	调用集成时发生的错误。
IntegrationLatency	API Gateway 向集成发送请求和 API Gateway 从集成接收响应之间的时间差。已对回调和模拟集成禁止。

您可以使用下表中的维度筛选 API Gateway 指标。

维度	说明
ApiId	筛选具有指定 API ID 的 API 的 API Gateway 指标。
ApiId、阶段	针对具有指定 API ID 和阶段 ID 的 API 阶段筛选 API Gateway 指标。
ApiId、方法、资源、阶段	使用指定的 API ID、阶段 ID、资源路径和路由 ID 筛选 API 方法的 API Gateway 指标。 除非您明确启用了详细的 CloudWatch 指标，否则 API Gateway 不会发送这些指标。要执行此操作，您可以通过调用 API Gateway V2 REST API 的 UpdateStage 操作，将 `detailedMetricsEnabled` 属性更新为 `true`。或者，您也可以调用 update-stage AWS CLI 命令，以将 `DetailedMetricsEnabled` 属性更新为 `true`。启用这些指标会对您的账户额外计费。有关定价信息，请参阅 Amazon CloudWatch 定价。

配置 WebSocket API 的日志记录

您可以启用日志记录以将日志写入 CloudWatch Logs。在 CloudWatch 中有两种类型的 API 日志记录：执行日志记录和访问日志记录。在执行日志记录中，API Gateway 管理 CloudWatch Logs。该过程包括创建日志组和日志流，以及向日志流报告任意调用方的请求和响应。

在访问日志记录中，作为 API 开发人员，您想要记录谁访问了您的 API 以及调用方访问 API 的方式。您可以创建自己的日志组，或者选择可由 API Gateway 管理的现有日志组。要指定访问详细信息，您可以选择 `$context` 变量（以您选择的格式表示），并选择日志组作为目标。

有关如何设置 CloudWatch 日志记录的说明，请参阅the section called “使用 API Gateway 控制台设置 CloudWatch API 日志记录”。

当您指定 Log Format (日志格式) 时，您可以选择要记录的上下文变量。支持以下变量。

参数	说明
`$context.apiId`	API Gateway 分配给您的 API 的标识符。
`$context.authorize.error`	授权错误消息。
`$context.authorize.latency`	授权延迟时间（以毫秒为单位）。
`$context.authorize.status`	从授权尝试返回的状态代码。
`$context.authorizer.error`	从授权方返回的错误消息。
`$context.authorizer.integrationLatency`	Lambda 授权方延迟（以毫秒为单位）。
`$context.authorizer.integrationStatus`	从 Lambda 授权方返回的状态代码。
`$context.authorizer.latency`	授权方延迟（以毫秒为单位）。
`$context.authorizer.requestId`	AWS 端点的请求 ID。
`$context.authorizer.status`	从授权方返回的状态代码。
`$context.authorizer.principalId`	与由客户端发送的令牌相关联的委托人用户标识，从API Gateway Lambda 授权方 Lambda 函数返回。（Lambda 授权方以前称为自定义授权方。）
`$context.authorizer.` *property*	从 API Gateway Lambda 授权方函数返回的 `context` 映射的指定键/值对的字符串化值。例如，如果授权方返回以下 `context` 映射：

```
"context" : {
                             "key":
 "value",
                             "numKey":
 1,
                             "boolKey":
 true
                             }
```

参数	说明
	调用 `$context.authorizer.key` 将返回 `"value"` 字符串，调用 `$context.authorizer.numKey` 将返回 `"1"` 字符串，而调用 `$context.authorizer.boolKey` 将返回 `"true"` 字符串。
`$context.authenticate.error`	从身份验证尝试返回的错误消息。
`$context.authenticate.latency`	身份验证延迟时间（以毫秒为单位）。
`$context.authenticate.status`	从身份验证尝试返回的状态代码。
`$context.connectedAt`	Epoch 格式的连接时间。
`$context.connectionId`	连接的唯一 ID，可用于对客户端进行回调。
`$context.domainName`	WebSocket API 的域名。这可用于对客户端进行回调（而不是硬编码值）。
`$context.error.message`	包含 API Gateway 错误消息的字符串。
`$context.error.messageString`	`$context.error.message` 的带引号的值，即 `"$context.error.message"` 。
`$context.error.responseType`	错误响应类型。
`$context.error.validationErrorString`	包含详细验证错误消息的字符串。

参数	说明
`$context.eventType`	事件类型：`CONNECT`、`MESSAGE` 或 `DISCONNECT` 。
`$context.extendedRequestId`	等效于 `$context.requestId` 。
`$context.identity.accountId`	与请求关联的 AWS 账户 ID。
`$context.identity.apiKey`	API 所有者密钥与启用密钥的 API 请求关联。
`$context.identity.apiKeyId`	API 密钥 ID 与启用密钥的 API 请求关联
`$context.identity.caller`	签发请求的调用方的委托人标识符。对于使用 IAM 授权的路由支持此项。
`$context.identity.cognitoAuthenticationProvider`	发出请求的调用方使用的 Amazon Cognito 身份验证提供商的逗号分隔列表。仅当使用 Amazon Cognito 凭证对请求签名时才可用。 例如，对于 Amazon Cognito 身份池中的身份，`cognito-idp.` *`region`* `.amazonaws.com/` *`user_pool_id`* `,cognito-idp.` *`region`* `.amazonaws.com/` *`user_pool_id`* `:CognitoSignIn:` *`token subject claim`* 有关更多信息，请参阅 Amazon Cognito 开发人员指南 中的使用联合身份。
`$context.identity.cognitoAuthenticationType`	发出请求的调用方的 Amazon Cognito 身份验证类型。仅当使用 Amazon Cognito 凭证对请求签名时才可用。可能的值包括经过身份验证的身份的 `authenticated` 和未经身份验证的身份的 `unauthenticated` 。
`$context.identity.cognitoIdentityId`	发出请求的调用方的 Amazon Cognito 身份 ID。仅当使用 Amazon Cognito 凭证对请求签名时才可用。

参数	说明
`$context.identity.cognitoIdentityPoolId`	发出请求的调用方的 Amazon Cognito 身份池 ID。仅当使用 Amazon Cognito 凭证对请求签名时才可用。
`$context.identity.principalOrgId`	AWS 组织 ID。对于使用 IAM 授权的路由支持此项。
`$context.identity.sourceIp`	向 API Gateway 发出请求的 TCP 连接的源 IP 地址。
`$context.identity.user`	将获得资源访问权限授权的用户的委托人标识符。对于使用 IAM 授权的路由支持此项。
`$context.identity.userAgent`	API 调用方的用户代理。
`$context.identity.userArn`	身份验证后标识的有效用户的 Amazon 资源名称 (ARN)。
`$context.integration.error`	从集成返回的错误消息。
`$context.integration.integrationStatus`	对于 Lambda 代理集成，从 AWS Lambda（而不是从后端 Lambda 函数代码）返回的状态代码。
`$context.integration.latency`	集成延迟（毫秒）。等效于 `$context.integrationLatency` 。
`$context.integration.requestId`	AWS 端点的请求 ID。等效于 `$context.awsEndpointRequestId` 。
`$context.integration.status`	从集成返回的状态代码。对于 Lambda 代理集成，这是 Lambda 函数代码返回的状态代码。等效于 `$context.integrationStatus` 。
`$context.integrationLatency`	集成延迟（毫秒），仅可用于访问日志记录。
`$context.messageId`	消息的唯一服务器端 ID。仅当 `$context.eventType` 为 `MESSAGE` 时才可用。

参数	说明
`$context.requestId`	与 `$context.extendedRequestId` 相同。
`$context.requestTime`	CLF 格式的请求时间 (`dd/MMM/yyyy:HH:mm:ss +-hhmm`)。
`$context.requestTimeEpoch`	Epoch 格式的请求时间，以毫秒为单位。
`$context.routeKey`	选定的路由键。
`$context.stage`	API 调用的部署阶段（例如测试或生产）。
`$context.status`	响应状态。
`$context.waf.error`	从 返回的错误消息AWS WAF
`$context.waf.latency`	AWS WAF 延迟时间（以毫秒为单位）。
`$context.waf.status`	从 返回的状态代码AWS WAF

一些常用访问日志格式的示例在 API Gateway 控制台中显示，下面列出了这些格式。

- `CLF` (常用日志格式)：

```
$context.identity.sourceIp $context.identity.caller \
$context.identity.user [$context.requestTime] "$context.eventType $context.routeKey
 $context.connectionId" \
$context.status $context.requestId
```

  继续符 (\) 用作视觉辅助。日志格式必须为单行。您可以在日志格式末尾添加换行符 (\n)，以便在每个日志条目末尾添加换行符。

- `JSON`:

```
{
"requestId":"$context.requestId", \
"ip": "$context.identity.sourceIp", \
"caller":"$context.identity.caller", \
"user":"$context.identity.user", \
```

```
"requestTime":"$context.requestTime", \
"eventType":"$context.eventType", \
"routeKey":"$context.routeKey", \
"status":"$context.status", \
"connectionId":"$context.connectionId"
}
```

继续符 (\) 用作视觉辅助。日志格式必须为单行。您可以在日志格式末尾添加换行符 (\n)，以便在每个日志条目末尾添加换行符。

- `XML`:

```
<request id="$context.requestId"> \
 <ip>$context.identity.sourceIp</ip> \
 <caller>$context.identity.caller</caller> \
 <user>$context.identity.user</user> \
 <requestTime>$context.requestTime</requestTime> \
 <eventType>$context.eventType</eventType> \
 <routeKey>$context.routeKey</routeKey> \
 <status>$context.status</status> \
 <connectionId>$context.connectionId</connectionId> \
</request>
```

继续符 (\) 用作视觉辅助。日志格式必须为单行。您可以在日志格式末尾添加换行符 (\n)，以便在每个日志条目末尾添加换行符。

- `CSV` (逗号分隔值)：

```
$context.identity.sourceIp,$context.identity.caller, \
$context.identity.user,$context.requestTime,$context.eventType, \
$context.routeKey,$context.connectionId,$context.status, \
$context.requestId
```

继续符 (\) 用作视觉辅助。日志格式必须为单行。您可以在日志格式末尾添加换行符 (\n)，以便在每个日志条目末尾添加换行符。

API Gateway Amazon 资源名称 (ARN) 参考

下表列出 API Gateway 资源的 Amazon 资源名称 (ARN)。要了解有关在 AWS Identity and Access Management 策略中使用 ARN 的更多信息，请参阅Amazon API Gateway 如何与 IAM 配合使用和使用 IAM 权限控制对 API 的访问。

HTTP API 和 WebSocket API 资源

资源	ARN
AccessLogSettings	`arn:partition :apigateway: region::/apis/api-id/stages/stage-name /accesslogsettings`
API	`arn:partition :apigateway: region::/apis/api-id`
Api	`arn:partition :apigateway: region::/apis`
ApiMapping	`arn:partition :apigateway: region::/domainnames/ domain-name /apimappings/ id`
ApiMappings	`arn:partition :apigateway: region::/domainnames/ domain-name /apimappings`
授权方	`arn:partition :apigateway: region::/apis/api-id/authorizers/ id`
授权方	`arn:partition :apigateway: region::/apis/api-id/authorizers`

资源	ARN
Cors	`arn:partition :apigateway: region::/apis/api-id/cors`
部署	`arn:partition :apigateway: region::/apis/api-id/deployments/ id`
部署	`arn:partition :apigateway: region::/apis/api-id/deployments`
DomainName	`arn:partition :apigateway: region::/domainnames/ domain-name`
DomainNames	`arn:partition :apigateway: region::/domainnames`
ExportedAPI	`arn:partition :apigateway: region::/apis/api-id/exports/ specification`
集成	`arn:partition :apigateway: region::/apis/api-id/integrations/ integration-id`
集成	`arn:partition :apigateway: region::/apis/api-id/integrations`
IntegrationResponse	`arn:partition :apigateway: region::/apis/api-id/integrationresponses/  integration-response`

资源	ARN
IntegrationResponses	`arn:partition :apigateway: region::/apis/api-id/integrationresponses`
模型	`arn:partition :apigateway: region::/apis/api-id/models/id`
模型	`arn:partition :apigateway: region::/apis/api-id/models`
ModelTemplate	`arn:partition :apigateway: region::/apis/api-id/models/id/template`
路线	`arn:partition :apigateway: region::/apis/api-id/routes/id`
路线	`arn:partition :apigateway: region::/apis/api-id/routes`
RouteRequestParameter	`arn:partition :apigateway: region::/apis/api-id/routes/id/requestparameters/ key`
RouteResponse	`arn:partition :apigateway: region::/apis/api-id/routes/id/routeresponses/ id`
RouteResponses	`arn:partition :apigateway: region::/apis/api-id/routes/id/routeresponses`
RouteSettings	`arn:partition :apigateway: region::/apis/api-id/stages/stage-name /routesettings/ route-key`

资源	ARN
舞台	`arn:`*`partition`*` :apigateway: `*`region`*`::/apis/`*`api-id`*`/stages/`*`stage-name`*
阶段	`arn:`*`partition`*` :apigateway: `*`region`*`::/apis/`*`api-id`*`/stages`
VpcLink	`arn:`*`partition`*` :apigateway: `*`region`*`::/vpclinks/ `*`vpclink-id`*
VpcLinks	`arn:`*`partition`*` :apigateway: `*`region`*`::/vpclinks`

Rest API 资源 ID

资源	ARN
账户	`arn:`*`partition`*` :apigateway: `*`region`*`::/account`
ApiKey	`arn:`*`partition`*` :apigateway: `*`region`*`::/apikeys/ `*`id`*
ApiKeys	`arn:`*`partition`*` :apigateway: `*`region`*`::/apikeys`
授权方	`arn:`*`partition`*` :apigateway: `*`region`*`::/restapis/ `*`api-id`*`/authorizers/ `*`id`*
授权方	`arn:`*`partition`*` :apigateway: `*`region`*`::/restapis/ `*`api-id`*`/authorizers`

资源	ARN
BasePathMapping	arn:*partition* :apigateway: *region*::/domainnames/ *domain-name* /basepathmappings/ *basepath*
BasePathMappings	arn:*partition* :apigateway: *region*::/domainnames/ *domain-name* /basepathmappings
ClientCertificate	arn:*partition* :apigateway: *region*::/clientcertificates/ *id*
ClientCertificates	arn:*partition* :apigateway: *region*::/clientcertificates
部署	arn:*partition* :apigateway: *region*::/restapis/ *api-id*/ deployments/ *id*
部署	arn:*partition* :apigateway: *region*::/restapis/ *api-id*/ deployments
DocumentationPart	arn:*partition* :apigateway: *region*::/restapis/ *api-id*/ documentation/parts/ *id*
DocumentationParts	arn:*partition* :apigateway: *region*::/restapis/ *api-id*/ documentation/parts
DocumentationVersion	arn:*partition* :apigateway: *region*::/restapis/ *api-id*/ documentation/versions/ *version*

资源	ARN
DocumentationVersions	`arn:partition :apigateway: region::/restapis/ api-id/documentation/versions`
DomainName	`arn:partition :apigateway: region::/domainnames/ domain-name`
DomainNames	`arn:partition :apigateway: region::/domainnames`
GatewayResponse	`arn:partition :apigateway: region::/restapis/ api-id/gatewayresponses/ response-type`
GatewayResponses	`arn:partition :apigateway: region::/restapis/ api-id/gatewayresponses`
集成	`arn:partition :apigateway: region::/restapis/ api-id/resources/ resource-id /methods/ http-method /integration`
IntegrationResponse	`arn:partition :apigateway: region::/restapis/ api-id/resources/ resource-id /methods/ http-method /integration/responses/ status-code`
方法	`arn:partition :apigateway: region::/restapis/ api-id/resources/ resource-id /methods/ http-method`

资源	ARN
MethodResponse	`arn:`*`partition`*`:apigateway:`*`region`*`::/restapis/`*`api-id`*`/resources/`*`resource-id`*`/methods/`*`http-method`*`/responses/`*`status-code`*
模型	`arn:`*`partition`*`:apigateway:`*`region`*`::/restapis/`*`api-id`*`/models/`*`model-name`*
模型	`arn:`*`partition`*`:apigateway:`*`region`*`::/restapis/`*`api-id`*`/models`
RequestValidator	`arn:`*`partition`*`:apigateway:`*`region`*`::/restapis/`*`api-id`*`/requestvalidators/`*`id`*
RequestValidators	`arn:`*`partition`*`:apigateway:`*`region`*`::/restapis/`*`api-id`*`/requestvalidators`
资源	`arn:`*`partition`*`:apigateway:`*`region`*`::/restapis/`*`api-id`*`/resources/`*`id`*
资源	`arn:`*`partition`*`:apigateway:`*`region`*`::/restapis/`*`api-id`*`/resources`
RestApi	`arn:`*`partition`*`:apigateway:`*`region`*`::/restapis/`*`api-id`*
RestApis	`arn:`*`partition`*`:apigateway:`*`region`*`::/restapis`

资源	ARN
舞台	`arn:partition :apigateway: region::/restapis/ api-id/stages/stage-name`
阶段	`arn:partition :apigateway: region::/restapis/ api-id/stages`
标签	`arn:partition :apigateway: region::/tags/url-encoded-resource-arn`
模板	`arn:partition :apigateway: region::/restapis/models/ model-name /template`
UsagePlan	`arn:partition :apigateway: region::/usageplans/ usageplan-id`
UsagePlans	`arn:partition :apigateway: region::/usageplans`
UsagePlanKey	`arn:partition :apigateway: region::/usageplans/ usageplan-id /keys/id`
UsagePlanKeys	`arn:partition :apigateway: region::/usageplans/ usageplan-id /keys`
VpcLink	`arn:partition :apigateway: region::/vpclinks/ vpclink-id`
VpcLinks	`arn:partition :apigateway: region::/vpclinks`

`execute-api`（HTTP API、WebSocket API 和 REST API）

资源	ARN
WebSocket API 终端节点	`arn:partition :execute-api: region:account-id :api-id/stage/route-key`
HTTP API 和 REST API 端点 *	`arn:partition :execute-api: region:account-id :api-id/stage/http-method /resource-path`
Lambda 授权方 **	`arn:partition :execute-api: region:account-id :api-id/authorizers/ authorizer-id`

* HTTP API 的 `$default` 路由端点的 ARN 是 `arn:partition:execute-api:region:account-id:api-id/*/$default`。

** 此 ARN 仅在资源策略中为 Lambda 授权方函数设置 `SourceArn` 条件时适用。有关示例，请参阅 the section called “创建 Lambda 授权方”。

使用基于 OpenAPI 的 API Gateway 扩展

API Gateway 扩展针对 REST API 和 HTTP API 支持AWS特定的授权和 API Gateway 特定的 API 集成。在本节中，我们将介绍基于 OpenAPI 规范的 API Gateway 扩展。

> Tip
>
> 要了解如何在应用程序中使用 API Gateway 扩展，您可以使用 API Gateway 控制台创建 REST API 或 HTTP API 并将其导出到 OpenAPI 定义文件中。有关如何导出 API 的更多信息，请参阅 从 API Gateway 导出 REST API 和 从 API Gateway 导出 HTTP API。

主题

- x-amazon-apigateway-integration.responses 对象
- x-amazon-apigateway-integration.response 对象
- x-amazon-apigateway-integration.responseTemplates 对象
- x-amazon-apigateway-integration.responseParameters 对象
- x-amazon-apigateway-integration.tlsConfig object
- x-amazon-apigateway-minimum-compression-size
- x-amazon-apigateway-policy
- x-amazon-apigateway-request-validator 属性
- x-amazon-apigateway-request-validators 对象
- x-amazon-apigateway-request-validators.requestValidator 对象
- x-amazon-apigateway-tag-value 属性

x-amazon-apigateway-any-method 对象

在 OpenAPI 路径项目对象中为 API Gateway“捕获全部”的 ANY 方法指定 OpenAPI 操作对象。该对象可与其他操作对象共存，并能捕获未明确声明的任何 HTTP 方法。

下表列出了由 API Gateway 扩展的属性。有关其他 OpenAPI 操作属性，请参阅 OpenAPI 规范。

属性

属性名称	类型	说明
`isDefaultRoute`	`Boolean`	指定路由是否为 `$default` 路由。仅 HTTP API 支持。要了解更多信息，请参阅“使用 HTTP API 的路由”。
`x-amazon-apigateway-integration`	x-amazon-apigateway-integration 对象	指定该方法与后端的集成。这是 OpenAPI 操作对象的扩展属性。集成的类型可以是 `AWS`、`AWS_PROXY`、`HTTP`、`HTTP_PROXY` 或 `MOCK`。

x-amazon-apigateway-any-method 示例

以下示例集成了代理资源 ANY 上的 `{proxy+}` 方法与 Lambda 函数 `TestSimpleProxy`。

```
    "/{proxy+}": {
      "x-amazon-apigateway-any-method": {
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "proxy",
            "in": "path",
            "required": true,
            "type": "string"
          }
        ],
        "responses": {},
        "x-amazon-apigateway-integration": {
          "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:TestSimpleProxy/invocations",
          "httpMethod": "POST",
          "type": "aws_proxy"
        }
```

以下示例为与 Lambda 函数 `$default` 集成的 HTTP API 创建 `HelloWorld` 路由。

```
"/$default": {
    "x-amazon-apigateway-any-method": {
      "isDefaultRoute": true,
      "x-amazon-apigateway-integration": {
        "type": "AWS_PROXY",
        "httpMethod": "POST",
        "uri": "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:123456789012:function:HelloWorld/invocations",
        "timeoutInMillis": 1000,
        "connectionType": "INTERNET",
        "payloadFormatVersion": 1.0
      }
   }
}
```

x-amazon-apigateway-cors 对象

为 HTTP API 指定跨域资源共享 (CORS) 配置。扩展适用于根级 OpenAPI 结构。要了解更多信息，请参阅“为 HTTP API 配置 CORS”。

属性

属性名称	类型	说明
`allowOrigins`	`Array`	指定允许的源。
`allowCredentials`	`Boolean`	指定是否在 CORS 请求中包含凭证。
`exposeHeaders`	`Array`	指定公开的标头。
`maxAge`	`Integer`	指定浏览器应缓存预检请求结果的秒数。
`allowMethods`	`Array`	指定允许的 HTTP 方法。
`allowHeaders`	`Array`	指定允许的标头。

x-amazon-apigateway-cors 示例

以下是用于 HTTP API 的 CORS 配置示例。

```
"x-amazon-apigateway-cors": {
    "allowOrigins": [
      "https://www.example.com"
    ],
    "allowCredentials": true,
    "exposeHeaders": [
      "x-apigateway-header",
      "x-amz-date",
      "content-type"
    ],
    "maxAge": 3600,
    "allowMethods": [
      "GET",
      "OPTIONS",
```

```
      "POST"
    ],
    "allowHeaders": [
      "x-apigateway-header",
      "x-amz-date",
      "content-type"
    ]
}
```

x-amazon-apigateway-api-key-source 属性

指定源来接收 API 密钥，用于限制获得密钥的 API 方法。此 API 级别属性是 `String` 类型。有关将方法配置为需要 API 密钥的更多信息，请参阅the section called "配置一个将 API 密钥与 OpenAPI 定义结合使用的方法"。

为请求指定 API 密钥的源。有效值为：

- `HEADER` 用于从请求的 `X-API-Key` 标头接收 API 密钥。
- `AUTHORIZER` 用于从 Lambda 授权方（以前称为自定义授权方）接收 `UsageIdentifierKey` 的 API 密钥。

x-amazon-apigateway-api-key-source 示例

以下示例将 `X-API-Key` 标头设置为 API 密钥源。

OpenAPI 2.0

```
{
  "swagger" : "2.0",
  "info" : {
    "title" : "Test1"
   },
  "schemes" : [ "https" ],
  "basePath" : "/import",
  "x-amazon-apigateway-api-key-source" : "HEADER",
   .
   .
   .
```

```
}
```

OpenAPI 3.0.1

```
{
  "openapi" : "3.0.1",
  "info" : {
    "title" : "Test1"
  },
  "servers" : [ {
    "url" : "/{basePath}",
    "variables" : {
      "basePath" : {
        "default" : "import"
      }
    }
  } ],
  "x-amazon-apigateway-api-key-source" : "HEADER",
   .
   .
   .
}
```

x-amazon-apigateway-auth 对象

在 API Gateway 中定义为方法调用的授权应用的授权类型。

属性

属性名称	类型	说明
`type`	`string`	指定授权类型。为开放访问指定 `"NONE"`。指定 `"AWS_IAM"` 以使用 IAM 权限。值不区分大小写。

x-amazon-apigateway-auth 示例

下面的示例为 API 方法设置授权类型。

OpenAPI 3.0.1

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "openapi3",
    "version": "1.0"
  },
  "paths": {
    "/protected-by-iam": {
      "get": {
        "x-amazon-apigateway-auth": {
          "type": "AWS_IAM"
        }
      }
    }
  }
}
```

x-amazon-apigateway-authorizer 对象

定义 Lambda 授权方、Amazon Cognito 用户群体或 JWT 授权方，用于对 API Gateway 中的方法调用进行授权。此扩展适用于 OpenAPI 2 和 OpenAPI 3 中的安全定义。

属性

属性名称	类型	说明
`type`	`string`	授权方的类型。这是一个必需属性。 对于 REST API，为具有已嵌入授权令牌中的调用方身份的授权方指定 `token`。为具有请求参数中包含的调用方身份的授权方指定 `request`。为使用 Amazon Cognito 用户群体控制对 API 的访问权限的授权方指定 `cognito_user_pools` 。

属性名称	类型	说明
		对于 HTTP API，为具有请求参数中包含的调用方身份的 Lambda 授权方指定 `request`。为 JWT 授权方指定 `jwt`。
`authorizerUri`	`string`	授权方 Lambda 函数的统一资源标识符 (URI)。语法如下： `"arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/arn:aws:lambda:us-east-1:account-id:function:auth_function_name/invocations"`
`authorizerCredentials`	`string`	调用授权方所需的凭证（如果有），该凭证采用 IAM 执行角色的 ARN 形式。例如，“arn:aws:iam::*account-id*:*IAM_role*”。
`authorizerPayloadFormatVersion`	`string`	对于 HTTP API，指定 API Gateway 发送到 Lambda 授权方的数据格式，以及 API Gateway 如何解释来自 Lambda 的响应。要了解更多信息，请参阅“the section called “负载格式版本””。

属性名称	类型	说明
`enableSimpleResponses`	`Boolean`	对于 HTTP API，指定 `request` 授权方是返回布尔值还是 IAM 策略。仅支持 `authorizerPayloadFormatVersion` 为 `2.0` 的授权方。如果启用，Lambda 授权方函数将返回一个布尔值。要了解更多信息，请参阅“the section called “格式 2.0 的 Lambda 函数响应””。
`identitySource`	`string`	请求参数（作为身份来源）的映射表达式的逗号分隔列表。仅适用于 `request` 和 `jwt` 类型的授权方。
`jwtConfiguration`	`Object`	指定 JWT 授权方的发布者和受众。要了解更多信息，请参阅 API Gateway 版本 2 API 参考中的 JWTConfiguration。仅 HTTP API 支持。
`identityValidationExpression`	`string`	一个正则表达式，用于验证作为传入身份的令牌。例如，“^x-[a-z]+”。仅 REST API 支持 `TOKEN` 授权方。
`authorizerResultTtlInSeconds`	`string`	对授权方结果进行缓存的秒数。
`providerARNs`	`string` 数组	`COGNITO_USER_POOLS` 的 Amazon Cognito 用户群体 ARN 列表。

REST API 的 x-amazon-apigateway-authorizer 示例

以下 OpenAPI 安全定义示例指定了一个名为 `test-authorizer` 的“token”类型的 Lambda 授权方。

```
"securityDefinitions" : {
    "test-authorizer" : {
      "type" : "apiKey",                         // Required and the value must be
 "apiKey" for an API Gateway API.
      "name" : "Authorization",                  // The name of the header containing
 the authorization token.
      "in" : "header",                           // Required and the value must be
 "header" for an API Gateway API.
      "x-amazon-apigateway-authtype" : "oauth2", // Specifies the authorization
 mechanism for the client.
      "x-amazon-apigateway-authorizer" : {       // An API Gateway Lambda authorizer
 definition
        "type" : "token",                        // Required property and the value
 must "token"
        "authorizerUri" : "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:account-id:function:function-name/invocations",
        "authorizerCredentials" : "arn:aws:iam::account-id:role",
        "identityValidationExpression" : "^x-[a-z]+",
        "authorizerResultTtlInSeconds" : 60
      }
    }
  }
```

以下 OpenAPI 操作对象代码段设置 `GET /http` 来使用上述 Lambda 授权方。

```
"/http" : {
   "get" : {
     "responses" : { },
     "security" : [ {
       "test-authorizer" : [ ]
     } ],
     "x-amazon-apigateway-integration" : {
       "type" : "http",
       "responses" : {
         "default" : {
           "statusCode" : "200"
         }
```

```
          },
          "httpMethod" : "GET",
          "uri" : "http://api.example.com"
        }
      }
    }
```

以下 OpenAPI 安全定义示例指定“request”类型的 Lambda 授权方作为身份来源，带单个标头参数 (`auth`)。`securityDefinitions` 名为 `request_authorizer_single_header`。

```
"securityDefinitions": {
    "request_authorizer_single_header" : {
      "type" : "apiKey",
      "name" : "auth",               // The name of a single header or query parameter
 as the identity source.
      "in" : "header",               // The location of the single identity source
 request parameter. The valid value is "header" or "query"
      "x-amazon-apigateway-authtype" : "custom",
      "x-amazon-apigateway-authorizer" : {
        "type" : "request",
        "identitySource" : "method.request.header.auth",   // Request parameter mapping
 expression of the identity source. In this example, it is the 'auth' header.
        "authorizerCredentials" : "arn:aws:iam::123456789012:role/AWSepIntegTest-CS-
LambdaRole",
        "authorizerUri" : "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:123456789012:function:APIGateway-Request-
Authorizer:vtwo/invocations",
        "authorizerResultTtlInSeconds" : 300
      }
    }
}
```

以下 OpenAPI 安全定义示例指定“request”类型的 Lambda 授权方作为身份来源，带有一个标头 (`HeaderAuth1`) 和一个查询字符串参数 `QueryString1`。

```
"securityDefinitions": {
    "request_authorizer_header_query" : {
      "type" : "apiKey",
      "name" : "Unused",             // Must be "Unused" for multiple identity sources
 or non header or query type of request parameters.
      "in" : "header",               // Must be "header" for multiple identity sources
 or non header or query type of request parameters.
```

```
      "x-amazon-apigateway-authtype" : "custom",
      "x-amazon-apigateway-authorizer" : {
        "type" : "request",
        "identitySource" : "method.request.header.HeaderAuth1,
 method.request.querystring.QueryString1",   // Request parameter mapping expressions
 of the identity sources.
        "authorizerCredentials" : "arn:aws:iam::123456789012:role/AWSepIntegTest-CS-
LambdaRole",
        "authorizerUri" : "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:123456789012:function:APIGateway-Request-
Authorizer:vtwo/invocations",
        "authorizerResultTtlInSeconds" : 300
      }
    }
}
```

以下 OpenAPI 安全定义示例指定“request”类型的 API Gateway Lambda 授权方作为身份来源，带单个阶段变量 (`stage`)。

```
"securityDefinitions": {
    "request_authorizer_single_stagevar" : {
      "type" : "apiKey",
      "name" : "Unused",             // Must be "Unused", for multiple identity sources
 or non header or query type of request parameters.
      "in" : "header",               // Must be "header", for multiple identity sources
 or non header or query type of request parameters.
      "x-amazon-apigateway-authtype" : "custom",
      "x-amazon-apigateway-authorizer" : {
        "type" : "request",
        "identitySource" : "stageVariables.stage",   // Request parameter mapping
 expression of the identity source. In this example, it is the stage variable.
        "authorizerCredentials" : "arn:aws:iam::123456789012:role/AWSepIntegTest-CS-
LambdaRole",
        "authorizerUri" : "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:123456789012:function:APIGateway-Request-
Authorizer:vtwo/invocations",
        "authorizerResultTtlInSeconds" : 300
      }
    }
}
```

以下 OpenAPI 安全定义示例将 Amazon Cognito 用户群体指定为授权方。

```
"securityDefinitions": {
   "cognito-pool": {
     "type": "apiKey",
     "name": "Authorization",
     "in": "header",
     "x-amazon-apigateway-authtype": "cognito_user_pools",
     "x-amazon-apigateway-authorizer": {
       "type": "cognito_user_pools",
       "providerARNs": [
         "arn:aws:cognito-idp:us-east-1:123456789012:userpool/us-east-1_ABC123"
       ]
     }
   }
```

以下 OpenAPI 操作对象片段将 `GET /http` 设置为使用前面的 Amazon Cognito 用户群体作为授权方，而没有自定义范围。

```

  "/http" : {
     "get" : {
       "responses" : { },
       "security" : [ {
         "cognito-pool" : [ ]
       } ],
       "x-amazon-apigateway-integration" : {
         "type" : "http",
         "responses" : {
           "default" : {
             "statusCode" : "200"
           }
         },
         "httpMethod" : "GET",
         "uri" : "http://api.example.com"
       }
     }
   }
```

HTTP API 的 x-amazon-apigateway-authorizer 示例

以下 OpenAPI 3.0 示例为使用 Amazon Cognito 作为身份提供商的 HTTP API 创建 JWT 授权方，并将 `Authorization` 标头作为身份来源。

```
"securitySchemes": {
  "jwt-authorizer-oauth": {
    "type": "oauth2",
     "x-amazon-apigateway-authorizer": {
       "type": "jwt",
       "jwtConfiguration": {
          "issuer": "https://cognito-idp.region.amazonaws.com/userPoolId",
          "audience": [
            "audience1",
            "audience2"
          ]
        },
        "identitySource": "$request.header.Authorization"
    }
  }
}
```

以下 OpenAPI 3.0 示例生成的 JWT 授权方与上一个示例相同。但是，此示例使用 OpenAPI 的 `openIdConnectUrl` 属性来自动检测发布者。`openIdConnectUrl` 必须完全形成。

```
"securitySchemes": {
  "jwt-authorizer-autofind": {
    "type": "openIdConnect",
    "openIdConnectUrl": "https://cognito-idp.region.amazonaws.com/userPoolId/.well-
known/openid-configuration",
    "x-amazon-apigateway-authorizer": {
      "type": "jwt",
      "jwtConfiguration": {
        "audience": [
          "audience1",
          "audience2"
        ]
      },
      "identitySource": "$request.header.Authorization"
    }
  }
}
```

以下示例为 HTTP API 创建一个 Lambda 授权方。此示例授权方使用 `Authorization` 标头作为其身份来源。授权方使用 `2.0` 负载格式版本，并返回布尔值，因为 `enableSimpleResponses` 设置为 `true`。

```
"securitySchemes" : {
  "lambda-authorizer" : {
    "type" : "apiKey",
    "name" : "Authorization",
    "in" : "header",
    "x-amazon-apigateway-authorizer" : {
      "type" : "request",
      "identitySource" : "$request.header.Authorization",
      "authorizerUri" : "arn:aws:apigateway:us-west-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-west-2:123456789012:function:function-name/invocations",
      "authorizerPayloadFormatVersion" : "2.0",
      "authorizerResultTtlInSeconds" : 300,
      "enableSimpleResponses" : true
    }
  }
}
```

x-amazon-apigateway-authtype 属性

对于 REST API，此扩展可用于定义 Lambda 授权方的自定义类型。在这种情况下，该值为自由格式。例如，一个 API 可能有多个使用不同内部方案的 Lambda 授权方。您可以使用此扩展来标识 Lambda 授权方的内部方案。

更常见的是，在 HTTP API 和 REST API 中，它还可以用作在共享同一安全方案的多个操作中定义 IAM 授权的方法。在这种情况下，术语 `awsSigv4` 是保留术语，以及前缀为 `aws` 的任何术语。

此扩展适用于 OpenAPI 2 和 OpenAPI 3 中的 `apiKey` 类型安全方案。

x-amazon-apigateway-authtype 示例

以下 OpenAPI 3 示例定义了 REST API 或 HTTP API 中的多个资源的 IAM 授权：

```
{
  "openapi" : "3.0.1",
  "info" : {
    "title" : "openapi3",
    "version" : "1.0"
  },
  "paths" : {
    "/operation1" : {
      "get" : {
```

```
        "responses" : {
          "default" : {
            "description" : "Default response"
          }
        },
        "security" : [ {
          "sigv4Reference" : [ ]
        } ]
      }
    },
    "/operation2" : {
      "get" : {
        "responses" : {
          "default" : {
            "description" : "Default response"
          }
        },
        "security" : [ {
          "sigv4Reference" : [ ]
        } ]
      }
    }
  },
  "components" : {
    "securitySchemes" : {
      "sigv4Reference" : {
        "type" : "apiKey",
        "name" : "Authorization",
        "in" : "header",
        "x-amazon-apigateway-authtype": "awsSigv4"
      }
    }
  }
}
```

以下 OpenAPI 3 示例使用 REST API 的自定义方案定义了 Lambda 授权方：

```
{
  "openapi" : "3.0.1",
  "info" : {
    "title" : "openapi3 for REST API",
    "version" : "1.0"
  },
```

```
  "paths" : {
    "/protected-by-lambda-authorizer" : {
      "get" : {
        "responses" : {
          "200" : {
            "description" : "Default response"
          }
        },
        "security" : [ {
          "myAuthorizer" : [ ]
        } ]
      }
    }
  },
  "components" : {
    "securitySchemes" : {
      "myAuthorizer" : {
        "type" : "apiKey",
        "name" : "Authorization",
        "in" : "header",
        "x-amazon-apigateway-authorizer" : {
          "identitySource" : "method.request.header.Authorization",
          "authorizerUri" : "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/
functions/arn:aws:lambda:us-east-1:account-id:function:function-name/invocations",
          "authorizerResultTtlInSeconds" : 300,
          "type" : "request",
          "enableSimpleResponses" : false
        },
        "x-amazon-apigateway-authtype": "Custom scheme with corporate claims"
      }
    }
  },
  "x-amazon-apigateway-importexport-version" : "1.0"
}
```

另请参阅

[authorizer.authType](#)

x-amazon-apigateway-binary-media-types 属性

指定 API Gateway 支持的二进制媒体类型列表，例如 `application/octet-stream` 和 `image/jpeg`。该扩展是一个 JSON 数组。它应作为 OpenAPI 文档的顶级供应商扩展包含在内。

x-amazon-apigateway-binary-media-types 示例

以下示例显示了 API 的编码查找顺序。

```
"x-amazon-apigateway-binary-media-types": [ "application/octet", "image/jpeg" ]
```

x-amazon-apigateway-documentation 对象

定义要导入到 API Gateway 中的文档部分。此对象是一个包含 `DocumentationPart` 示例数组的 JSON 对象。

属性

属性名称	类型	说明
`documentationParts`	`Array`	导出或导入的 `DocumentationPart` 实例数组。
`version`	`String`	导出文档部分的快照的版本标识符。

x-amazon-apigateway-documentation 示例

以下基于 OpenAPI 的 API Gateway 扩展示例将 `DocumentationParts` 实例定义为从 API Gateway 中的 API 导入或导出。

```
{ ...
  "x-amazon-apigateway-documentation": {
    "version": "1.0.3",
    "documentationParts": [
       {
         "location": {
           "type": "API"
```

```
       },
        "properties": {
          "description": "API description",
          "info": {
            "description": "API info description 4",
            "version": "API info version 3"
          }
        }
      },
      {
         … // Another DocumentationPart instance
      }
    ]
  }
}
```

x-amazon-apigateway-endpoint-configuration 对象

指定 API 的端点配置的详细信息。此扩展是 OpenAPI 操作对象的扩展属性。此对象应该存在于 Swagger 2.0 的顶级供应商扩展中。对于 OpenAPI 3.0，它应该存在于服务器对象的供应商扩展下。

属性

属性名称	类型	描述
`disableExecuteApiEndpoint`	Boolean	指定客户端是否可以使用默认 `execute-api` 端点调用您的 API。默认情况下，客户端可以使用默认 `https://{api_id}.execute-api.{region}.amazonaws.com` 端点调用您的 API。如果要求客户端使用自定义域名来调用 API，请指定 `true`。
`vpcEndpointIds`	`String` 数组	VpcEndpoint 标识符的列表，可针对这些标识符为 REST API 创建 Route 53 别名。仅 `PRIVATE` 端点类型的 REST API 支持它。

x-amazon-apigateway-endpoint-configuration 示例

以下示例将指定的 VPC 端点与 REST API 关联。

```
"x-amazon-apigateway-endpoint-configuration": {
    "vpcEndpointIds": ["vpce-0212a4ababd5b8c3e", "vpce-01d622316a7df47f9"]
}
```

以下示例禁用 API 的默认端点。

```
"x-amazon-apigateway-endpoint-configuration": {
    "disableExecuteApiEndpoint": true
}
```

x-amazon-apigateway-gateway-responses 对象

将 API 的网关响应定义为键/值对的字符串到 GatewayResponse 映射。扩展适用于根级 OpenAPI 结构。

属性

属性名称	类型	说明
responseType	x-amazon-apigateway-gateway-responses.gatewayResponse	指定 *responseType* 的 `GatewayResponse`。

x-amazon-apigateway-gateway-responses 示例

以下基于 OpenAPI 的 API Gateway 扩展示例定义了一个 GatewayResponses 映射，该映射包含两个 GatewayResponse 实例，其中一个为 `DEFAULT_4XX` 类型，一个为 `INVALID_API_KEY` 类型。

```
{
  "x-amazon-apigateway-gateway-responses": {
    "DEFAULT_4XX": {
      "responseParameters": {
        "gatewayresponse.header.Access-Control-Allow-Origin": "'domain.com'"
      },
```

```
      "responseTemplates": {
        "application/json": "{\"message\": test 4xx b }"
      }
    },
    "INVALID_API_KEY": {
      "statusCode": "429",
      "responseTemplates": {
        "application/json": "{\"message\": test forbidden }"
      }
    }
  }
}
```

x-amazon-apigateway-gateway-responses.gatewayResponse 对象

定义了给定响应类型的网关响应，包括状态代码、任何适用的响应参数或响应模板。

属性

属性名称	类型	说明
`responseParameters`	x-amazon-apigateway-gateway-responses.responseParameters	指定 GatewayResponse 参数，即标头参数。参数值可以采用任何传入请求参数值或静态自定义值。
`responseTemplates`	x-amazon-apigateway-gateway-responses.responseTemplates	指定网关响应的映射模板。这些模板不由 VTL 引擎处理。
`statusCode`	`string`	网关响应的 HTTP 状态代码。

x-amazon-apigateway-gateway-responses.gatewayResponse 示例

以下基于 OpenAPI 的 API Gateway 扩展示例定义了 GatewayResponse 来自定义 `INVALID_API_KEY` 响应，以返回状态代码 `456`、传入请求的 `api-key` 标头值以及 `"Bad api-key"` 消息。

```
    "INVALID_API_KEY": {
      "statusCode": "456",
      "responseParameters": {
        "gatewayresponse.header.api-key": "method.request.header.api-key"
      },
      "responseTemplates": {
        "application/json": "{\"message\": \"Bad api-key\" }"
      }
    }
```

x-amazon-apigateway-gateway-responses.responseParameters 对象

定义键/值对的字符串到字符串映射，从传入请求参数或使用文本字符串生成网关响应参数。仅支持用于 REST API 。

属性

属性名称	类型	说明
`gatewayresponse. param-position .param-name`	`string`	*param-position* 可以是 `header`、`path` 或 `querystring` 。有关更多信息，请参阅 将方法请求数据映射至集成请求参数。

x-amazon-apigateway-gateway-responses.responseParameters 示例

以下 OpenAPI 扩展示例显示了一个 GatewayResponse 响应参数映射表达式，可用于为 `*.example.domain` 域中的资源启用 CORS 支持。

```
      "responseParameters": {
        "gatewayresponse.header.Access-Control-Allow-Origin": '*.example.domain',
        "gatewayresponse.header.from-request-header" : method.request.header.Accept,
        "gatewayresponse.header.from-request-path" : method.request.path.petId,
        "gatewayresponse.header.from-request-query" : method.request.querystring.qname
      }
```

x-amazon-apigateway-gateway-responses.responseTemplates 对象

对于给定的网关响应，将 GatewayResponse 映射模板定义为采用键/值对的字符串到字符串映射。对于每个键/值对，“键”指内容类型。例如“application/json”，而“值”指简单变量替换的字符串化映射模板。`GatewayResponse` 映射模板不由 Velocity 模板语言 (VTL) 引擎处理。

属性

属性名称	类型	说明
`content-type`	`string`	`GatewayResponse` 正文映射模板，仅支持简单变量替换以自定义网关响应正文。

x-amazon-apigateway-gateway-responses.responseTemplates 示例

下面的 OpenAPI 扩展示例显示了一个 GatewayResponse 映射模板，用于将 API Gateway 生成的错误响应自定义为特定于应用程序的格式。

```
      "responseTemplates": {
        "application/json": "{ \"message\": $context.error.messageString, \"type\":
$context.error.responseType, \"statusCode\": '488' }"
      }
```

下面的 OpenAPI 扩展示例显示了一个 GatewayResponse 映射模板，用于使用静态错误消息覆盖 API Gateway 生成的错误响应。

```
      "responseTemplates": {
        "application/json": "{ \"message\": 'API-specific errors' }"
      }
```

x-amazon-apigateway-importexport-version

指定 HTTP API 的 API Gateway 导入和导出算法的版本。目前，唯一支持的值是 `1.0`。要了解更多信息，请参阅 API Gateway 版本 2 API 参考中的 exportVersion。

x-amazon-apigateway-importexport-version 示例

以下示例将导入和导出版本设置为 `1.0`。

```
{
    "openapi": "3.0.1",
    "x-amazon-apigateway-importexport-version": "1.0",
    "info": { ...
```

x-amazon-apigateway-integration 对象

指定用于该方法的后端集成的详细信息。此扩展是 OpenAPI 操作对象的扩展属性。结果是 API Gateway 集成对象。

属性

属性名称	类型	说明
`cacheKeyParameters`	`string` 数组	一个需要缓存值的请求参数列表。
`cacheNamespace`	`string`	相关缓存参数的 API 特定的标记组。
`connectionId`	`string`	私有集成的 VpcLink 的 ID。
`connectionType`	`string`	集成连接类型。对于私有集成，有效值为 `"VPC_LINK"`，否则为 `"INTERNET"`。
`credentials`	`string`	对于基于 AWS IAM 角色的凭证，指定一个适当的 IAM 角色的 ARN。若未指定，凭证将默认为基于资源的许可，该许可必须手动添加，以使 API 能访问相应资源。有关更多信息，请参阅使用资源策略授予许可。

属性名称	类型	说明
		注意：使用 IAM 凭证时，请确保已对部署 API 的区域启用 AWS STS 区域端点以实现最佳性能。
contentHandling	string	请求负载编码转换类型。有效值为 1) `CONVERT_TO_TEXT` ，用于将二进制负载转换为 Base64 编码字符串，或者将文本负载转换为 `utf-8` 编码字符串，或者在无任何修改的情况下直接传递文本负载，以及 2) `CONVERT_TO_BINARY` ，用于将文本负载转换为 Base64 解码的二进制大型对象，或者在无任何修改的情况下直接传递二进制负载。
httpMethod	string	集成请求中使用的 HTTP 方法。对于 Lambda 函数调用，值必须是 `POST`。
integrationSubtype	string	指定 AWS 服务集成的集成子类型。仅 HTTP API 支持。有关支持的集成子类型，请参阅 the section called “AWS 服务集成参考”。

属性名称	类型	说明
`passthroughBehavior`	`string`	指定如何在无任何修改的情况下通过集成请求传递未映射内容类型的请求负载。支持的值有 `when_no_templates` 、`when_no_match` 和 `never`。有关更多信息，请参阅 Integration.passthroughBehavior。
`payloadFormatVersion`	`string`	指定发送到集成的负载的格式。对 HTTP API 是必需的。对于 HTTP API，Lambda 代理集成支持的值为 `1.0` 和 `2.0`。对于所有其他集成，`1.0` 是唯一受支持的值。要了解更多信息，请参阅the section called “AWS Lambda 集成”和the section called “AWS 服务集成参考”。

属性名称	类型	说明
`requestParameters`	x-amazon-apigateway-integration.requestParameters 对象	对于 REST API，指定从方法请求参数到集成请求参数的映射。支持的请求参数有 `querystring` 、`path`、`header` 和 `body`。 对于 HTTP API，请求参数是键/值映射，用于指定通过指定的 `AWS_PROXY` 传递给 `integrationSubtype` 集成的参数。您可以提供静态值，或者在运行时评估的映射请求数据、阶段变量或上下文变量。要了解更多信息，请参阅"the section called "AWS 服务集成""。
`requestTemplates`	x-amazon-apigateway-integration.requestTemplates 对象	指定的 MIME 类型请求负载的映射模板。
`responses`	x-amazon-apigateway-integration.responses 对象	定义方法的响应，并指定从集成响应到方法响应的所需的参数映射或负载映射。
`timeoutInMillis`	`integer`	集成超时介于 50 毫秒到 29000 毫秒之间。

属性名称	类型	说明
`type`	`string`	与指定后端的集成的类型。有效值为： • `http` 或 `http_proxy`：适用于与 HTTP 后端集成。 • `aws_proxy`，适用于与 AWS Lambda 函数集成。 • `aws`，用于与 AWS Lambda 函数或其他AWS服务（如 Amazon DynamoDB、Amazon Simple Notification Service 或 Amazon Simple Queue Service）集成。 • `mock`，适用于与 API Gateway 集成，而无需调用任何后端。 有关集成类型的更多信息，请参阅 integration:type。
`tlsConfig`	the section called “x-amazon-apigateway-integration.tlsConfig”	指定集成的 TLS 配置。
`uri`	`string`	后端的端点 URI。对于 `aws` 类型的集成，此为一个 ARN 值。对于 HTTP 集成，此为 HTTP 端点的 URL，包括 `https` 或 `http` 方案。

x-amazon-apigateway-integration 示例

对于 HTTP API，您可以在 OpenAPI 定义的组件部分中定义集成。要了解更多信息，请参阅“x-amazon-apigateway-integrations 对象”。

```
"x-amazon-apigateway-integration": {
    "$ref": "#/components/x-amazon-apigateway-integrations/integration1"
}
```

以下示例创建了与 Lambda 函数的集成。出于演示目的，假定 `requestTemplates` 中显示的示例映射模板和下面示例中的 `responseTemplates` 适用于以下 JSON 格式的负载：`{ "name":"value_1", "key":"value_2", "redirect": {"url" :"..."} }`，以生成 `{ "stage":"value_1", "user-id":"value_2" }` 的 JSON 输出或 `<stage>value_1</stage>` 的 XML 输出。

```


"x-amazon-apigateway-integration" : {
    "type" : "aws",
    "uri" : "arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-1:012345678901:function:HelloWorld/invocations",
    "httpMethod" : "POST",
    "credentials" : "arn:aws:iam::012345678901:role/apigateway-invoke-lambda-exec-
role",
    "requestTemplates" : {
                "application/json" : "#set ($root=$input.path('$')) { \"stage\":
 \"$root.name\", \"user-id\": \"$root.key\" }",
                "application/xml" : "#set ($root=$input.path('$')) <stage>$root.name</
stage> "
    },
    "requestParameters" : {
        "integration.request.path.stage" : "method.request.querystring.version",
        "integration.request.querystring.provider" :
 "method.request.querystring.vendor"
    },
    "cacheNamespace" : "cache namespace",
    "cacheKeyParameters" : [],
    "responses" : {
        "2\\d{2}" : {
            "statusCode" : "200",
            "responseParameters" : {
                "method.response.header.requestId" : "integration.response.header.cid"
            },
            "responseTemplates" : {
                "application/json" : "#set ($root=$input.path('$')) { \"stage\":
 \"$root.name\", \"user-id\": \"$root.key\" }",
                "application/xml" : "#set ($root=$input.path('$')) <stage>$root.name</
stage> "
```

```
            }
        },
        "302" : {
            "statusCode" : "302",
            "responseParameters" : {
                "method.response.header.Location" :
 "integration.response.body.redirect.url"
            }
        },
        "default" : {
            "statusCode" : "400",
            "responseParameters" : {
                "method.response.header.test-method-response-header" : "'static value'"
            }
        }
    }
}
```

注意，映射模板中 JSON 字符串的双引号 (") 必须是转义字符串 (\")。

x-amazon-apigateway-integrations 对象

定义集成的集合。您可以在 OpenAPI 定义的组件部分中定义集成，并对多个路由重复使用集成。仅 HTTP API 支持。

属性

属性名称	类型	说明
##	x-amazon-apigateway-integration 对象	集成对象的集合。

x-amazon-apigateway-integrations 示例

以下示例创建定义两个集成的 HTTP API，并通过使用 `$ref": "#/components/x-amazon-apigateway-integrations/`*`integration-name`* 引用集成。

```
{
  "openapi": "3.0.1",
  "info":
```

```
    {
      "title": "Integrations",
      "description": "An API that reuses integrations",
      "version": "1.0"
    },
  "servers": [
  {
    "url": "https://example.com/{basePath}",
    "description": "The production API server",
    "variables":
      {
        "basePath":
          {
            "default": "example/path"
          }
      }
  }],
  "paths":
    {
      "/":
        {
          "get":
            {
              "x-amazon-apigateway-integration":
                {
                  "$ref": "#/components/x-amazon-apigateway-integrations/integration1"

                }
            }
        },
       "/pets":
        {
          "get":
            {
              "x-amazon-apigateway-integration":
                {
                  "$ref": "#/components/x-amazon-apigateway-integrations/integration1"

                }
            }
        },
       "/checkout":
        {
          "get":
```

```
          {
            "x-amazon-apigateway-integration":
              {
                "$ref": "#/components/x-amazon-apigateway-integrations/integration2"
              }
          }
        }
    },
    "components": {
      "x-amazon-apigateway-integrations":
        {
          "integration1":
            {
              "type": "aws_proxy",
              "httpMethod": "POST",
              "uri": "arn:aws:apigateway:us-east-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-2:123456789012:function:my-function/invocations",
              "passthroughBehavior": "when_no_templates",
              "payloadFormatVersion": "1.0"
            },
          "integration2":
            {
              "type": "aws_proxy",
              "httpMethod": "POST",
              "uri": "arn:aws:apigateway:us-east-2:lambda:path/2015-03-31/functions/
arn:aws:lambda:us-east-2:123456789012:function:example-function/invocations",
              "passthroughBehavior": "when_no_templates",
              "payloadFormatVersion" : "1.0"
            }
        }
    }
}
```

x-amazon-apigateway-integration.requestTemplates 对象

为给定 MIME 类型的请求负载指定映射模板。

属性

属性名称	类型	说明
MIME type	`string`	MIME 类型的一个示例是 `application/json` 。有关创建映射模板的信息，请参阅 PetStore 映射模板。

x-amazon-apigateway-integration.requestTemplates 示例

下面的示例为 `application/json` 和 `application/xml` MIME 类型的请求负载设置了映射模板。

```
"requestTemplates" : {
    "application/json" : "#set ($root=$input.path('$')) { \"stage\": \"$root.name\",
 \"user-id\": \"$root.key\" }",
    "application/xml" : "#set ($root=$input.path('$')) <stage>$root.name</stage> "
}
```

x-amazon-apigateway-integration.requestParameters 对象

对于 REST API，指定从给定的方法请求参数到集成请求参数的映射。要引用方法请求参数，必须先对其进行定义。

对于 HTTP API，指定传递给具有指定 `AWS_PROXY` 的 `integrationSubtype` 集成的参数。

属性

属性名称	类型	说明
`integration.request.` *<param-type>* `.`*<param-name>*	`string`	对于 REST API，该值通常是 `method.request.` *<param-type>* `.`*<param-name>* 格式的预定义方法请求

属性名称	类型	说明
		参数，其中 `<param-type>` 可以是 `querystring`、`path`、`header` 或 `body`。但是，`$context.` *`VARIABLE_NAME`*、`$stageVariables.` *`VARIABLE_NAME`* 和 *`STATIC_VALUE`* 也有效。对于 `body` 参数，`<param-name>` 是不带 `$.` 前缀的 JSON 路径表达式。
`parameter`	`string`	对于 HTTP API，请求参数是键/值映射，用于指定通过指定的 `AWS_PROXY` 传递给 `integrationSubtype` 集成的参数。您可以提供静态值，或者在运行时评估的映射请求数据、阶段变量或上下文变量。要了解更多信息，请参阅“the section called “AWS 服务集成””。

`x-amazon-apigateway-integration.requestParameters` 示例

以下请求参数映射示例将方法请求的查询 (`version`)、标头 (`x-user-id`) 和路径 (`service`) 参数分别转换为了集成请求的查询 (`stage`)、标头 (`x-userid`) 和路径 (`op`) 参数。

> **Note**
>
> 如果您要通过 OpenAPI 或 AWS CloudFormation 创建资源，静态值应括在单引号中。
>
> 要从控制台添加此值，请在框中输入 `application/json`，无需引号。

```
"requestParameters" : {
    "integration.request.querystring.stage" : "method.request.querystring.version",
    "integration.request.header.x-userid" : "method.request.header.x-user-id",
    "integration.request.path.op" : "method.request.path.service"
},
```

x-amazon-apigateway-integration.responses 对象

定义方法的响应，并指定从集成响应到方法响应的参数映射或负载映射。

属性

属性名称	类型	说明
######	x-amazon-apigateway-integration.response 对象	可以是用于将集成响应与方法响应匹配的正则表达式，或者用于捕获尚未配置的任何响应的 `default`。对于 HTTP 集成，该正则表达式适用于集成响应状态代码。对于 Lambda 调用，当 Lambda 函数执行发生异常时，该正则表达式适用于 AWS Lambda 作为失败响应正文返回的错误信息对象的 `errorMessage` 字段。 Note ######属性名称是指响应状态代码或描述一组响应状态代码的正则表达式。它不对应 API Gateway REST API 中 IntegrationResponse 资源的任意标识符。

x-amazon-apigateway-integration.responses 示例

下面的示例显示了一列 2xx 响应和 302 响应。对于 2xx 响应，从 application/json 或 application/xml MIME 类型的集成响应负载映射方法响应。此响应使用提供的映射模板。对于 302 响应，方法响应会返回一个 Location 标头，该标头的值是由集成响应负载上的 redirect.url 属性派生出来的。

```
"responses" : {
    "2\\d{2}" : {
        "statusCode" : "200",
        "responseTemplates" : {
            "application/json" : "#set ($root=$input.path('$')) { \"stage\":
 \"$root.name\", \"user-id\": \"$root.key\" }",
            "application/xml" : "#set ($root=$input.path('$')) <stage>$root.name</
stage> "
        }
    },
    "302" : {
        "statusCode" : "302",
        "responseParameters" : {
            "method.response.header.Location": "integration.response.body.redirect.url"
        }
    }
}
```

x-amazon-apigateway-integration.response 对象

定义响应并指定从集成响应到方法响应的参数映射或负载映射。

属性

属性名称	类型	说明
statusCode	string	方法响应的 HTTP 状态代码，例如 "200"。必须对应 OpenAPI 操作responses 字段中的匹配响应。

属性名称	类型	说明
`responseTemplates`	x-amazon-apigateway-integration.responseTemplates 对象	为响应的负载指定特定于 MIME 类型的映射模板。
`responseParameters`	x-amazon-apigateway-integration.responseParameters 对象	指定响应的参数映射。仅集成响应的 `header` 和 `body` 参数可以映射到方法的 `header` 参数。
`contentHandling`	`string`	响应负载编码转换类型。有效值为 1) `CONVERT_TO_TEXT` ，用于将二进制负载转换为 Base64 编码字符串，或者将文本负载转换为 `utf-8` 编码字符串，或者在无任何修改的情况下直接传递文本负载，以及 2) `CONVERT_TO_BINARY` ，用于将文本负载转换为 Base64 解码的二进制大型对象，或者在无任何修改的情况下直接传递二进制负载。

`x-amazon-apigateway-integration.response` 示例

下面的示例为从后端获得 `302` 或 `application/json` MIME 类型负载的方法定义了 `application/xml` 响应。该响应使用提供的映射模板，从方法的 `Location` 标头中的集成响应中返回重定向 URL。

```
{
    "statusCode" : "302",
    "responseTemplates" : {
         "application/json" : "#set ($root=$input.path('$')) { \"stage\": \"$root.name
\", \"user-id\": \"$root.key\" }",
         "application/xml" : "#set ($root=$input.path('$')) <stage>$root.name</stage> "
    },
    "responseParameters" : {
```

```
        "method.response.header.Location": "integration.response.body.redirect.url"
    }
}
```

x-amazon-apigateway-integration.responseTemplates 对象

为给定的 MIME 类型响应负载指定映射模板。

属性

属性名称	类型	说明
MIME type	`string`	指定映射模板，将集成响应正文转换为给定 MIME 类型的方法响应正文。有关创建映射模板的信息，请参阅 PetStore 映射模板。*MIME ##*的一个示例是 `application/json` 。

x-amazon-apigateway-integration.responseTemplate 示例

下面的示例为 `application/json` 和 `application/xml` MIME 类型的请求负载设置了映射模板。

```
"responseTemplates" : {
    "application/json" : "#set ($root=$input.path('$')) { \"stage\": \"$root.name\",
 \"user-id\": \"$root.key\" }",
    "application/xml" : "#set ($root=$input.path('$')) <stage>$root.name</stage> "
}
```

x-amazon-apigateway-integration.responseParameters 对象

指定从集成方法响应参数到方法响应参数的映射。您可以将 `header`、`body` 或静态值映射到 `header` 类型的方法响应。仅支持用于 REST API 。

属性

属性名称	类型	描述
`method.response.header. <param-name>`	`string`	指定的参数值可以源自 `header` 和 `body` 类型的集成响应参数。

`x-amazon-apigateway-integration.responseParameters` 示例

下面的示例将集成响应的 `body` 和 `header` 参数映射到了方法响应的两个 `header` 参数。

```
"responseParameters" : {
    "method.response.header.Location" : "integration.response.body.redirect.url",
    "method.response.header.x-user-id" : "integration.response.header.x-userid"
}
```

x-amazon-apigateway-integration.tlsConfig object

指定集成的 TLS 配置。

属性

属性名称	类型	说明
`insecureSkipVerification`	`Boolean`	仅支持用于 REST API 。指定 API Gateway 是否跳过集成端点的证书由受支持的证书颁发机构颁发的验证。建议不要这样做，但它使您能够使用由私有证书颁发机构签名的证书或自签名的证书。如果启用，API Gateway 仍会执行基本证书验证，其中包括检查证书的到期日期、主机名以及是否存在根证书颁发机构。属于私有颁发

属性名称	类型	说明
		机构的根证书必须满足以下限制： • x509 扩展 `keyUsage` 必须具有 `keyCertSign` 。 • x509 扩展 `basicConstraints` 必须具有 `CA:TRUE`。 仅支持用于 HTTP 和 `HTTP_PROXY` 集成。 **Warning** 不推荐启用 `insecureSkipVerification` ，特别是用于与公共 HTTPS 端点的集成。如果您启用 `insecureSkipVerification` ，将增加中间人攻击的风险。
`serverNameToVerify`	`string`	仅支持用于 HTTP API 私有集成。如果指定服务器名称，API Gateway 将使用它来验证集成证书上的主机名。TLS 握手中也包含服务器名称，以支持服务器名称指示 (SNI) 或虚拟主机。

x-amazon-apigateway-integration.tlsConfig examples

下面的 OpenAPI 3.0 示例为 REST API HTTP 代理集成启用 `insecureSkipVerification`。

```
"x-amazon-apigateway-integration": {
  "uri": "http://petstore-demo-endpoint.execute-api.com/petstore/pets",
  "responses": {
     default": {
       "statusCode": "200"
      }
  },
  "passthroughBehavior": "when_no_match",
  "httpMethod": "ANY",
  "tlsConfig" : {
    "insecureSkipVerification" : true
  }
  "type": "http_proxy",
}
```

下面的 OpenAPI 3.0 示例为 HTTP API 私有集成指定 `serverNameToVerify`。

```
"x-amazon-apigateway-integration" : {
  "payloadFormatVersion" : "1.0",
  "connectionId" : "abc123",
  "type" : "http_proxy",
  "httpMethod" : "ANY",
  "uri" : "arn:aws:elasticloadbalancing:us-west-2:123456789012:listener/app/my-load-
balancer/50dc6c495c0c9188/0467ef3c8400ae65",
  "connectionType" : "VPC_LINK",
  "tlsConfig" : {
     "serverNameToVerify" : "example.com"
  }
}
```

x-amazon-apigateway-minimum-compression-size

为 REST API 指定最小压缩大小。要启用压缩，请指定一个介于 0 和 10485760 之间的整数。要了解更多信息，请参阅“为 API 启用负载压缩”。

x-amazon-apigateway-minimum-compression-size example

以下示例指定 REST API 的最小压缩大小 5242880（字节）。

```
"x-amazon-apigateway-minimum-compression-size": 5242880
```

x-amazon-apigateway-policy

为 REST API 指定资源策略。要了解有关资源策略的更多信息，请参阅 使用 API Gateway 资源策略控制对 API 的访问。有关资源策略示例，请参阅 API Gateway 资源策略示例。

`x-amazon-apigateway-policy` 示例

以下示例为 REST API 指定资源策略。此资源策略拒绝（阻止）从指定的源 IP 地址块到 API 的传入流量。导入时，将使用当前区域、您的AWS账户 ID 和当前 REST API ID 将 `"execute-api:/*"` 转换为 `arn:aws:execute-api:`*`region`*`:`*`account-id`*`:`*`api-id`*`/*`。

```
"x-amazon-apigateway-policy": {
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
                "execute-api:/*"
            ]
        },
        {
            "Effect": "Deny",
            "Principal": "*",
            "Action": "execute-api:Invoke",
            "Resource": [
               "execute-api:/*"
            ],
            "Condition" : {
                "IpAddress": {
                    "aws:SourceIp": "192.0.2.0/24"
                }
            }
```

```
        }
    ]
}
```

x-amazon-apigateway-request-validator 属性

引用 *request_validator_name* 映射的 [x-amazon-apigateway-request-validators 对象](x-amazon-apigateway-request-validators) 指定请求验证程序，对所包含的 API 或某个方法启用请求验证。此扩展的值是一个 JSON 字符串。

可在 API 级别或方法级别指定此扩展。API 级别的验证程序适用于所有方法，除非它被方法级别的验证程序覆盖。

`x-amazon-apigateway-request-validator` 示例

下面的示例使用了 API 级别的 `basic` 请求验证程序，同时对 `parameter-only` 请求启用了 `POST /validation` 请求验证程序。

OpenAPI 2.0

```
{
  "swagger": "2.0",
  "x-amazon-apigateway-request-validators" : {
    "basic" : {
      "validateRequestBody" : true,
      "validateRequestParameters" : true
    },
    "params-only" : {
      "validateRequestBody" : false,
      "validateRequestParameters" : true
    }
  },
  "x-amazon-apigateway-request-validator" : "basic",
  "paths": {
    "/validation": {
      "post": {
        "x-amazon-apigateway-request-validator" : "params-only",
       ...
     }
}
```

x-amazon-apigateway-request-validators 对象

将所包含的 API 支持的请求验证程序定义为验证程序名称与关联的请求验证规则之间的映射。此扩展适用于 REST API。

属性

属性名称	类型	说明
request_validator_name	x-amazon-apigateway-request-validators.requestValidator 对象	指定包含给定验证程序的验证规则。例如： （代码见下） 要将该验证程序应用于特定方法，请将验证程序名称 (basic) 引用为 x-amazon-apigateway-request-validator 属性 属性的值。

request_validator_name：

```
    "basic" : {
      "validateRequestBody" : true,
      "validateRequestParameters" : true
    },
```

x-amazon-apigateway-request-validators 示例

下面的示例以验证程序名称与关联的请求验证规则之间的映射的形式，显示了一组 API 的请求验证程序。

OpenAPI 2.0

```
{
  "swagger": "2.0",
  ...
  "x-amazon-apigateway-request-validators" : {
    "basic" : {
      "validateRequestBody" : true,
      "validateRequestParameters" : true
```

```
    },
    "params-only" : {
      "validateRequestBody" : false,
      "validateRequestParameters" : true
    }
  },
  ...
}
```

x-amazon-apigateway-request-validators.requestValidator 对象

将请求验证程序的验证规则指定为 x-amazon-apigateway-request-validators 对象 映射定义的一部分。

属性

属性名称	类型	说明
`validateRequestBody`	`Boolean`	指定是否要验证请求正文 (`true` 或 `false`)。
`validateRequestParameters`	`Boolean`	指定是否要验证所需的请求参数 (`true` 或 `false`)。

`x-amazon-apigateway-request-validators.requestValidator` 示例

下面的示例显示了一个仅限参数的请求验证程序：

```
"params-only": {
    "validateRequestBody" : false,
    "validateRequestParameters" : true
}
```

x-amazon-apigateway-tag-value 属性

指定 HTTP API 的AWS标签的值。您可以使用 `x-amazon-apigateway-tag-value` 属性作为根级别 OpenAPI 标签对象的一部分，用于指定 HTTP API 的AWS标签。如果您指定一个没有 `x-amazon-apigateway-tag-value` 属性的标签名称，则 API Gateway 将创建一个以空字符串为值的标签。

要了解添加标签的更多信息，请参阅 为 API Gateway 资源添加标签。

属性

属性名称	类型	说明
`name`	`String`	指定标签键。
`x-amazon-apigateway-tag-value`	`String`	指定标签值。

`x-amazon-apigateway-tag-value` 示例

以下示例为 HTTP API 指定两个标签：

- "Owner": "Admin"
- "Prod": ""

```
"tags": [
    {
      "name": "Owner",
      "x-amazon-apigateway-tag-value": "Admin"
    },
    {
      "name": "Prod"
    }
]
```

Amazon API Gateway 中的安全性

AWS 十分重视云安全性。作为 AWS 客户，您将从专为满足大多数安全敏感型企业的要求而打造的数据中心和网络架构中受益。

安全性是 AWS 和您的共同责任。责任共担模型将其描述为云的安全性和云中的安全性。

- 云的安全性 — AWS 负责保护在 AWS 云中运行 AWS 服务的基础设施。AWS 还向您提供可安全使用的服务。第三方审核员定期测试和验证我们的安全性的有效性，作为 AWS 合规性计划的一部分。要了解适用于 Amazon API Gateway 的合规性计划，请参阅合规性计划范围内的 AWS 服务。
- 云中的安全性：您的责任由您使用的 AWS 服务决定。您还需要对其他因素负责，包括您的数据的敏感性、您公司的要求以及适用的法律法规。

此文档将帮助您了解如何在使用 API Gateway 时应用责任共担模式。以下主题说明如何配置 API Gateway 以实现您的安全性和合规性目标。您还会了解如何使用其他 AWS 服务以帮助您监控和保护 API Gateway 资源。

有关更多信息，请参阅 Amazon API Gateway 安全概述。

主题

Amazon API Gateway 中的数据保护

AWS责任共担模式适用于 Amazon API Gateway 中的数据保护。如该模式中所述，AWS 负责保护运行所有 AWS Cloud 的全球基础设施。您负责维护对托管在此基础设施上的内容的控制。您还负责您所使用的 AWS 服务 的安全配置和管理任务。有关数据隐私的更多信息，请参阅数据隐私常见问题。有关欧洲数据保护的信息，请参阅 AWS 安全性博客 上的 AWS 责任共担模式和 GDPR 博客文章。

出于数据保护目的，我们建议您保护 AWS 账户 凭证并使用 AWS IAM Identity Center 或 AWS Identity and Access Management（IAM）设置单个用户。这样，每个用户只获得履行其工作职责所需的权限。我们还建议您通过以下方式保护数据：

- 对每个账户使用 multi-factor authentication（MFA）。
- 使用 SSL/TLS 与 AWS 资源进行通信。我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 使用 AWS CloudTrail 设置 API 和用户活动日记账记录。
- 使用 AWS 加密解决方案以及 AWS 服务 中的所有默认安全控制。
- 使用高级托管安全服务（例如 Amazon Macie），它有助于发现和保护存储在 Amazon S3 中的敏感数据。
- 如果在通过命令行界面或 API 访问 AWS 时需要经过 FIPS 140-2 验证的加密模块，请使用 FIPS 端点。有关可用的 FIPS 端点的更多信息，请参阅《美国联邦信息处理标准（FIPS）第 140-2 版》。

我们强烈建议您切勿将机密信息或敏感信息（如您客户的电子邮件地址）放入标签或自由格式文本字段（如名称字段）。这包括使用控制台、API、AWS CLI 或 AWS SDK 处理 API Gateway 或其他 AWS 服务 时。在用于名称的标签或自由格式文本字段中输入的任何数据都可能会用于计费或诊断日志。如果您向外部服务器提供网址，强烈建议您不要在网址中包含凭证信息来验证对该服务器的请求。

Amazon API Gateway 中的数据加密

数据保护是指，在数据传输（传入和传出 API Gateway 时）和处于静态（存储在 中时）期间保护数据 AWS

Amazon API Gateway 中的静态数据加密

如果选择为 REST API 启用缓存，则可以启用缓存加密。要了解更多信息，请参阅“启用 API 缓存以增强响应能力”。

有关数据保护的更多信息，请参阅AWS安全性博客上的AWS责任共担模式和 GDPR 博客文章。

Amazon API Gateway 中的传输中数据加密

通过 Amazon API Gateway 创建的 API 只公开 HTTPS 终端节点。API Gateway 不支持未加密的 (HTTP) 终端节点。

API Gateway 管理原定设置 `execute-api` 端点的证书。如果您配置自定义域名，请为该域名指定证书。作为最佳实践，请勿固定证书。

为了获得更高的安全性，您可以选择要为您的 API Gateway 自定义域实施的最低传输层安全性 (TLS) 协议版本。WebSocket API 和 HTTP API 仅支持 TLS 1.2。要了解更多信息，请参阅“在 API Gateway 中为自定义域选择安全策略”。

您还可以在账户中使用自定义 SSL 证书设置 Amazon CloudFront 分配，并将其用于区域 API。然后，您可以根据您的安全性和合规性要求，为具有 TLS 1.1 或更高版本的 CloudFront 分配配置安全策略。

有关数据保护的更多信息，请参阅 保护您的 REST API 以及AWS安全性博客上的AWS责任共担模式和 GDPR 博客文章。

互联网络流量隐私保护

使用 Amazon API Gateway，您可以创建只能从您的 Amazon Virtual Private Cloud (VPC) 访问的私有 REST API。VPC 使用接口 VPC 终端节点，该终端节点是您在 VPC 中创建的终端节点网络接口。通过使用资源策略，您可以允许或拒绝从选定的 VPC 和 VPC 终端节点（包括跨 AWS 账户）对您的 API 进行访问。每个终端节点都可用于访问多个私有 API。您还可以使用 AWS Direct Connect 建立从本地网络到 Amazon VPC 的连接，并在该连接上访问您的私有 API。在所有情况下，您的私有 API 的流量使用安全的连接，不会离开 Amazon 网络；它与公有 Internet 隔离开来。要了解更多信息，请参阅“the section called “私有 REST API””。

适用于 Amazon API Gateway 的 Identity and Access Management

AWS Identity and Access Management（IAM）是一项 AWS 服务，可以帮助管理员安全地控制对 AWS 资源的访问。IAM 管理员控制谁可以通过身份验证（登录）和授权（具有权限）以使用 API Gateway 资源。IAM 是一项无需额外费用即可使用的 AWS 服务。

主题

受众

使用 AWS Identity and Access Management (IAM) 的方式因您可以在 API Gateway 中执行的操作而异。

服务用户 - 如果您使用 API Gateway 服务来完成工作，则管理员会为您提供所需的凭证和权限。随着您使用更多 API Gateway 功能来完成工作，您可能需要额外的权限。了解如何管理访问权限可帮助您向管理员请求适合的权限。如果您无法访问 API Gateway 中的功能，请参阅Amazon API Gateway 身份和访问疑难解答。

服务管理员 – 如果您在公司负责管理 API Gateway 资源，您可能对 API Gateway 具有完全访问权限。您有责任确定您的服务用户应访问哪些 API Gateway 功能和资源。然后，您必须向 IAM 管理员提交请求以更改服务用户的权限。请查看该页面上的信息以了解 IAM 的基本概念。要了解有关您的公司如何将 IAM 与 API Gateway 搭配使用的更多信息，请参阅Amazon API Gateway 如何与 IAM 配合使用。

IAM 管理员 - 如果您是 IAM 管理员，您可能希望了解如何编写策略以管理对 API Gateway 的访问的详细信息。要查看您可在 IAM 中使用的 API Gateway 基于身份的策略示例，请参阅Amazon API Gateway 基于身份的策略示例。

使用身份进行身份验证

身份验证是您使用身份凭证登录 AWS 的方法。您必须作为 AWS 账户根用户、IAM 用户或通过代入 IAM 角色进行身份验证（登录到 AWS）。

您可以使用通过身份源提供的凭证以联合身份登录到 AWS。AWS IAM Identity Center（IAM Identity Center）用户、您公司的单点登录身份验证以及您的 Google 或 Facebook 凭证都是联合身份的示例。当您以联合身份登录时，您的管理员以前使用 IAM 角色设置了身份联合验证。当您使用联合身份验证访问 AWS 时，您就是在间接代入角色。

根据您的用户类型，您可以登录 AWS Management Console 或 AWS 访问门户。有关登录到 AWS 的更多信息，请参阅《AWS 登录 用户指南》中的 如何登录到您的 AWS 账户。

如果您以编程方式访问 AWS，则 AWS 将提供软件开发工具包（SDK）和命令行界面（CLI），以便使用您的凭证以加密方式签署您的请求。如果您不使用 AWS 工具，则必须自行对请求签名。有关使用推荐的方法自行签署请求的更多信息，请参阅《IAM 用户指南》中的签署 AWS API 请求。

无论使用何种身份验证方法，您可能需要提供其他安全信息。例如，AWS 建议您使用多重身份验证（MFA）来提高账户的安全性。要了解更多信息，请参阅《AWS IAM Identity Center 用户指南》中的多重身份验证 和《IAM 用户指南》中的 在 AWS 中使用多重身份验证（MFA）。

AWS 账户 根用户

当您创建 AWS 账户 时，最初使用的是一个对账户中所有 AWS 服务 和资源拥有完全访问权限的登录身份。此身份称为 AWS 账户 根用户，使用您创建账户时所用的电子邮件地址和密码登录，即可获得该身份。强烈建议您不要使用根用户执行日常任务。保护好根用户凭证，并使用这些凭证来执行仅根用户可以执行的任务。有关需要您以根用户身份登录的任务的完整列表，请参阅 IAM 用户指南 中的 需要根用户凭证的任务。

IAM 用户和群组

IAM 用户是 AWS 账户 内对某个人员或应用程序具有特定权限的一个身份。在可能的情况下，我们建议使用临时凭证，而不是创建具有长期凭证（如密码和访问密钥）的 IAM 用户。但是，如果您有一些特定的使用场景需要长期凭证以及 IAM 用户，我们建议您轮换访问密钥。有关更多信息，请参阅《IAM 用户指南》中的对于需要长期凭证的使用场景定期轮换访问密钥。

IAM 组是一个指定一组 IAM 用户的身份。您不能使用组的身份登录。您可以使用组来一次性为多个用户指定权限。如果有大量用户，使用组可以更轻松地管理用户权限。例如，您可能具有一个名为 IAMAdmins 的组，并为该组授予权限以管理 IAM 资源。

用户与角色不同。用户唯一地与某个人员或应用程序关联，而角色旨在让需要它的任何人代入。用户具有永久的长期凭证，而角色提供临时凭证。要了解更多信息，请参阅 IAM 用户指南中的何时创建 IAM 用户（而不是角色）。

IAM 角色

IAM 角色是 AWS 账户 中具有特定权限的身份。它类似于 IAM 用户，但与特定人员不关联。您可以通过切换角色，在 AWS Management Console 中暂时代入 IAM 角色。您可以调用 AWS CLI 或 AWS API 操作或使用自定义网址以担任角色。有关使用角色的方法的更多信息，请参阅《IAM 用户指南》中的 使用 IAM 角色。

具有临时凭证的 IAM 角色在以下情况下很有用：

- 联合用户访问 – 要向联合身份分配权限，请创建角色并为角色定义权限。当联合身份进行身份验证时，该身份将与角色相关联并被授予由此角色定义的权限。有关联合身份验证的角色的信息，请参阅《IAM 用户指南》中的 为第三方身份提供商创建角色。如果您使用 IAM Identity Center，则需要配置权限集。为控制您的身份在进行身份验证后可以访问的内容，IAM Identity Center 将权限集与 IAM 中的角色相关联。有关权限集的信息，请参阅《AWS IAM Identity Center 用户指南》中的权限集。
- 临时 IAM 用户权限 – IAM 用户可代入 IAM 用户或角色，以暂时获得针对特定任务的不同权限。
- 跨账户存取 – 您可以使用 IAM 角色以允许不同账户中的某个人（可信主体）访问您的账户中的资源。角色是授予跨账户访问权限的主要方式。但是，对于某些 AWS 服务，您可以将策略直接附加到

资源（而不是使用角色作为代理）。要了解用于跨账户访问的角色和基于资源的策略之间的差别，请参阅《IAM 用户指南》中的 IAM 角色与基于资源的策略有何不同。

- 跨服务访问 – 某些 AWS 服务使用其它 AWS 服务 中的特征。例如，当您在某个服务中进行调用时，该服务通常会在 Amazon EC2 中运行应用程序或在 Amazon S3 中存储对象。服务可能会使用发出调用的主体的权限、使用服务角色或使用服务相关角色来执行此操作。
  - 转发访问会话：当您使用 IAM 用户或角色在 AWS 中执行操作时，您将被视为主体。使用某些服务时，您可能会执行一个操作，此操作然后在不同服务中启动另一个操作。FAS 使用主体调用 AWS 服务 的权限，结合请求的 AWS 服务，向下游服务发出请求。只有在服务收到需要与其他 AWS 服务 或资源交互才能完成的请求时，才会发出 FAS 请求。在这种情况下，您必须具有执行这两个操作的权限。有关发出 FAS 请求时的策略详情，请参阅转发访问会话。
  - 服务角色 - 服务角色是服务代表您在您的账户中执行操作而分派的 IAM 角色。IAM 管理员可以在 IAM 中创建、修改和删除服务角色。有关更多信息，请参阅《IAM 用户指南》中的创建向 AWS 服务 委派权限的角色。
  - 服务相关角色 – 服务相关角色是与 AWS 服务 关联的一种服务角色。服务可以代入代表您执行操作的角色。服务相关角色显示在您的 AWS 账户 中，并由该服务拥有。IAM 管理员可以查看但不能编辑服务相关角色的权限。
- 在 Amazon EC2 上运行的应用程序 – 您可以使用 IAM 角色管理在 EC2 实例上运行并发出 AWS CLI 或 AWS API 请求的应用程序的临时凭证。这优先于在 EC2 实例中存储访问密钥。要将 AWS 角色分配给 EC2 实例并使其对该实例的所有应用程序可用，您可以创建一个附加到实例的实例配置文件。实例配置文件包含角色，并使 EC2 实例上运行的程序能够获得临时凭证。有关更多信息，请参阅《IAM 用户指南》中的 使用 IAM 角色为 Amazon EC2 实例上运行的应用程序授予权限。

要了解是使用 IAM 角色还是 IAM 用户，请参阅 IAM 用户指南中的何时创建 IAM 角色（而不是用户）。

使用策略管理访问

您将创建策略并将其附加到 AWS 身份或资源，以控制 AWS 中的访问。策略是 AWS 中的对象；在与身份或资源相关联时，策略定义它们的权限。在主体（用户、根用户或角色会话）发出请求时，AWS 将评估这些策略。策略中的权限确定是允许还是拒绝请求。大多数策略在 AWS 中存储为 JSON 文档。有关 JSON 策略文档的结构和内容的更多信息，请参阅《IAM 用户指南》中的 JSON 策略概览。

管理员可以使用 AWS JSON 策略来指定谁有权访问什么内容。也就是说，哪个主体 可以对什么资源执行操作，以及在什么条件下执行。

默认情况下，用户和角色没有权限。要授予用户对所需资源执行操作的权限，IAM 管理员可以创建 IAM policy。管理员随后可以向角色添加 IAM policy，用户可以代入角色。

IAM policy 定义操作的权限，无关乎您使用哪种方法执行操作。例如，假设您有一个允许 `iam:GetRole` 操作的策略。具有该策略的用户可以从 AWS Management Console、AWS CLI 或 AWS API 获取角色信息。

基于身份的策略

基于身份的策略是可附加到身份（如 IAM 用户、用户组或角色）的 JSON 权限策略文档。这些策略控制用户和角色可在何种条件下对哪些资源执行哪些操作。要了解如何创建基于身份的策略，请参阅 IAM 用户指南中的创建 IAM policy。

基于身份的策略可以进一步归类为内联策略或托管式策略。内联策略直接嵌入单个用户、组或角色中。托管式策略是可以附加到 AWS 账户 中的多个用户、组和角色的独立策略。托管式策略包括 AWS 托管式策略和客户托管式策略。要了解如何在托管式策略和内联策略之间进行选择，请参阅《IAM 用户指南》中的在托管式策略与内联策略之间进行选择。

基于资源的策略

基于资源的策略是附加到资源的 JSON 策略文档。基于资源的策略的示例包括 IAM 角色信任策略和 Simple Storage Service（Amazon S3）存储桶策略。在支持基于资源的策略的服务中，服务管理员可以使用它们来控制对特定资源的访问。对于在其中附加策略的资源，策略定义指定主体可以对该资源执行哪些操作以及在什么条件下执行。您必须在基于资源的策略中指定主体。主体可以包括账户、用户、角色、联合用户或 AWS 服务。

基于资源的策略是位于该服务中的内联策略。您不能在基于资源的策略中使用来自 IAM 的 AWS 托管式策略。

访问控制列表 (ACL)

访问控制列表（ACL）控制哪些主体（账户成员、用户或角色）有权访问资源。ACL 与基于资源的策略类似，尽管它们不使用 JSON 策略文档格式。

Simple Storage Service（Amazon S3）、AWS WAF 和 Amazon VPC 是支持 ACL 的服务示例。要了解有关 ACL 的更多信息，请参阅 Amazon Simple Storage Service 开发人员指南 中的访问控制列表（ACL）概览。

其它策略类型

AWS 支持额外的、不太常用的策略类型。这些策略类型可以设置更常用的策略类型向您授予的最大权限。

- 权限边界 – 权限边界是一个高级特征，用于设置基于身份的策略可以为 IAM 实体（IAM 用户或角色）授予的最大权限。您可为实体设置权限边界。这些结果权限是实体基于身份的策略及其权限边界的交集。在 `Principal` 中指定用户或角色的基于资源的策略不受权限边界限制。任一项策略中的显式拒绝将覆盖允许。有关权限边界的更多信息，请参阅 IAM 用户指南中的 IAM 实体的权限边界。
- 服务控制策略（SCP）– SCP 是 JSON 策略，指定了组织或组织单位（OU）在 AWS Organizations 中的最大权限。AWS Organizations 服务可以分组和集中管理您的企业拥有的多个 AWS 账户 账户。如果在组织内启用了所有特征，则可对任意或全部账户应用服务控制策略（SCP）。SCP 限制成员账户中实体（包括每个 AWS 账户根用户）的权限。有关 Organizations 和 SCP 的更多信息，请参阅 AWS Organizations 用户指南中的 SCP 的工作原理。
- 会话策略 – 会话策略是当您以编程方式为角色或联合身份用户创建临时会话时作为参数传递的高级策略。结果会话的权限是用户或角色的基于身份的策略和会话策略的交集。权限也可以来自基于资源的策略。任一项策略中的显式拒绝将覆盖允许。有关更多信息，请参阅 IAM 用户指南中的会话策略。

多个策略类型

当多个类型的策略应用于一个请求时，生成的权限更加复杂和难以理解。要了解AWS如何确定在涉及多种策略类型时是否允许请求，请参阅 IAM 用户指南中的策略评估逻辑。

Amazon API Gateway 如何与 IAM 配合使用

在使用 IAM 管理对 API Gateway 的访问权限之前，您应该了解哪些 IAM 功能可用于 API Gateway。要大致了解 API Gateway 和其他AWS服务如何与 IAM 一起使用，请参阅 IAM 用户指南中的与 IAM 一起使用的AWS服务。

主题

- API Gateway 基于身份的策略
- API Gateway 基于资源的策略
- 基于 API Gateway 标签的授权
- API Gateway IAM 角色

API Gateway 基于身份的策略

通过使用 IAM 基于身份的策略，您可以指定允许或拒绝的操作和资源以及允许或拒绝操作的条件。API Gateway 支持特定的操作、资源和条件键。有关 API Gateway 特定的操作、资源和条件键的更多信息，请参阅 Amazon API Gateway Management 的操作、资源和条件键和 Amazon API Gateway

Management V2 的操作、资源和条件键。如需在 JSON 策略中使用的所有元素，请参阅 IAM 用户指南中的 IAM JSON 策略元素参考。

以下示例显示了一个基于身份的策略，该策略允许用户仅创建或更新私有 REST API。有关更多示例，请参阅 the section called “基于身份的策略示例”。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ScopeToPrivateApis",
      "Effect": "Allow",
      "Action": [
        "apigateway:PATCH",
        "apigateway:POST",
        "apigateway:PUT"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/restapis",
        "arn:aws:apigateway:us-east-1::/restapis/??????????"
      ],
      "Condition": {
        "ForAllValues:StringEqualsIfExists": {
          "apigateway:Request/EndpointType": "PRIVATE",
          "apigateway:Resource/EndpointType": "PRIVATE"
        }
      }
    },
    {
      "Sid": "AllowResourcePolicyUpdates",
      "Effect": "Allow",
      "Action": [
          "apigateway:UpdateRestApiPolicy"
      ],
      "Resource": [
          "arn:aws:apigateway:us-east-1::/restapis/*"
      ]
    }
  ]
}
```

操作

JSON 策略的 `Action` 元素描述可用于在策略中允许或拒绝访问的操作。

API Gateway 中的策略操作在操作前面使用以下前缀：`apigateway:`。策略语句必须包含 `Action` 或 `NotAction` 元素。API Gateway 定义了一组自己的操作，以描述您可以使用该服务执行的任务。

API 管理 `Action` 表达式的格式为 `apigateway:`*`action`*，其中 *`action`* 是以下 API Gateway 操作之一：GET、POST、PUT、DELETE、PATCH（用于更新资源）或 *（即前面的所有操作）。

`Action` 表达式的一些示例包括：

- **`apigateway:*`**，表示所有 API Gateway 操作。
- **`apigateway:GET`**，仅表示 API Gateway 中的 GET 操作。

要在单个语句中指定多项操作，请使用逗号将它们隔开，如下所示：

```
"Action": [
      "apigateway:action1",
      "apigateway:action2"
```

有关用于特定 API Gateway 操作的 HTTP 动词的信息，请参阅 Amazon API Gateway 第 1 版 API 参考 (REST API) 和 Amazon API Gateway 第 2 版 API 参考（WebSocket 和 HTTP API）。

有关更多信息，请参阅 the section called “基于身份的策略示例”。

资源

管理员可以使用 AWS JSON 策略来指定谁有权访问什么内容。也就是说，哪个主体 可以对什么资源执行操作，以及在什么条件 下执行。

`Resource` JSON 策略元素指定要向其应用操作的一个或多个对象。语句必须包含 `Resource` 或 `NotResource` 元素。作为最佳实践，请使用其 Amazon 资源名称（ARN）指定资源。对于支持特定资源类型（称为资源级权限）的操作，您可以执行此操作。

对于不支持资源级权限的操作（如列出操作），请使用通配符 (*) 指示语句应用于所有资源。

```
"Resource": "*"
```

API Gateway 资源具有以下 ARN 格式：

```
arn:aws:apigateway:region::resource-path-specifier
```

例如，要指定带有 id *api-id* 及其子资源的 REST API（例如语句中的授权方），请使用以下 ARN：

```
"Resource": "arn:aws:apigateway:us-east-2::/restapis/api-id/*"
```

要指定属于特定账户的所有 REST API 和子资源，请使用通配符 (*)：

```
"Resource": "arn:aws:apigateway:us-east-2::/restapis/*"
```

有关 API Gateway 资源类型及其 ARN 的列表，请参阅 API Gateway Amazon 资源名称 (ARN) 参考。

条件键

管理员可以使用 AWS JSON 策略来指定谁有权访问什么内容。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

在 `Condition` 元素（或 `Condition` 块）中，可以指定语句生效的条件。`Condition` 元素是可选的。您可以创建使用条件运算符（例如，等于或小于）的条件表达式，以使策略中的条件与请求中的值相匹配。

如果您在一个语句中指定多个 `Condition` 元素，或在单个 `Condition` 元素中指定多个键，则 AWS 使用逻辑 `AND` 运算评估它们。如果您为单个条件键指定多个值，则 AWS 使用逻辑 `OR` 运算来评估条件。在授予语句的权限之前必须满足所有的条件。

在指定条件时，您也可以使用占位符变量。例如，只有在使用 IAM 用户名标记 IAM 用户时，您才能为其授予访问资源的权限。有关更多信息，请参阅 IAM 用户指南中的 IAM policy 元素：变量和标签。

AWS支持全局条件键和特定于服务的条件键。要查看所有 AWS 全局条件键，请参阅 IAM 用户指南 中的 AWS 全局条件上下文键。

API Gateway 定义了自己的一组条件键，还支持使用一些全局条件键。有关 API Gateway 条件键的列表，请参阅 IAM 用户指南中的 Amazon API Gateway 的条件键。有关您可以将哪些操作和资源与条件键结合使用，请参阅 Amazon API Gateway 定义的操作。

有关标记（包括基于属性的访问控制）的信息，请参阅 Tagging。

示例

有关 API Gateway 基于身份的策略示例，请参阅 Amazon API Gateway 基于身份的策略示例。

API Gateway 基于资源的策略

基于资源的策略是 JSON 策略文档，它们指定了指定的委托人可在 API Gateway 资源上执行的操作以及在什么条件下可执行。API Gateway 对于 REST API 支持基于资源的权限策略。您可以使用资源策略来控制谁可以调用 REST API。有关更多信息，请参阅 the section called “使用 API Gateway 资源策略”。

示例

有关基于 API Gateway 资源的策略示例，请参阅 API Gateway 资源策略示例。

基于 API Gateway 标签的授权

您可以将标签附加到 API Gateway 资源或将请求中的标签传递给 API Gateway。要基于标签控制访问，您需要使用 `apigateway:ResourceTag/key-name`、`aws:RequestTag/key-name` 或 `aws:TagKeys` 条件键在策略的条件元素中提供标签信息。有关标记 API Gateway 资源的更多信息，请参阅 the section called “基于属性的访问控制”。

有关基于身份的策略（用于根据资源上的标签来限制对该资源的访问）的示例，请参阅 使用标签控制对 API Gateway REST API 资源的访问。

API Gateway IAM 角色

IAM 角色是 AWS 账户中具有特定权限的实体。

将临时凭证用于 API Gateway

您可以使用临时凭证进行联合身份登录，担任 IAM 角色或担任跨账户角色。您可以通过调用 AWS STS API 操作（如 AssumeRole 或 GetFederationToken）获得临时安全凭证。

API Gateway 支持使用临时凭证。

服务相关角色

服务相关角色允许 AWS 服务访问其他服务中的资源以代表您完成操作。服务相关角色显示在 IAM 账户中，并归该服务所有。IAM 管理员可以查看但不能编辑服务相关角色的权限。

API Gateway 支持服务相关角色。有关创建或管理 API Gateway 服务相关角色的信息，请参阅 使用 API Gateway 的服务相关角色。

服务角色

服务可以代表您代入服务角色。此服务角色允许服务访问其他服务中的资源以代表您完成操作。服务角色显示在您的 IAM 账户中并由该账户拥有，因此管理员可以更改此角色的权限。但是，这样做可能会中断服务的功能。

API Gateway 支持服务角色。

Amazon API Gateway 基于身份的策略示例

默认情况下，IAM 用户和角色无权创建或修改 API Gateway 资源。它们还无法使用AWS Management Console、AWS CLI 或者AWS开发工具包执行任务。IAM 管理员必须创建 IAM 策略，以便为用户和角色授予权限以对所需的指定资源执行特定的 API 操作。然后，管理员必须将这些策略附加到需要这些权限的 IAM 用户或组。

有关如何创建 IAM 策略的信息，请参阅 IAM 用户指南中的在 JSON 选项卡上创建策略。有关 API Gateway 特定的操作、资源和条件的信息，请参阅 Amazon API Gateway Management 的操作、资源和条件键和 Amazon API Gateway Management V2 的操作、资源和条件键。

主题

- 策略最佳实践
- 允许用户查看他们自己的权限
- 简单读取许可
- 仅创建 REQUEST 或 JWT 授权方
- 要求禁用默认 execute-api 终端节点
- 只允许用户创建或更新私有 REST API
- 要求 API 路由具有授权
- 防止用户创建或更新 VPC 链接

策略最佳实践

基于身份的策略确定某个人是否可以创建、访问或删除您账户中的 API Gateway 资源。这些操作可能会使 AWS 账户产生成本。创建或编辑基于身份的策略时，请遵循以下准则和建议：

- AWS 托管策略及转向最低权限许可入门 - 要开始向用户和工作负载授予权限，请使用 AWS 托管策略来为许多常见使用场景授予权限。您可以在 AWS 账户 中找到这些策略。我们建议通过定义特定于您的使用场景的 AWS 客户管理型策略来进一步减少权限。有关更多信息，请参阅《IAM 用户指南》中的 AWS 托管式策略 或 工作职能的 AWS 托管式策略。
- 应用最低权限 – 在使用 IAM policy 设置权限时，请仅授予执行任务所需的权限。为此，您可以定义在特定条件下可以对特定资源执行的操作，也称为最低权限许可。有关使用 IAM 应用权限的更多信息，请参阅《IAM 用户指南》中的 IAM 中的策略和权限。
- 使用 IAM policy 中的条件进一步限制访问权限 – 您可以向策略添加条件来限制对操作和资源的访问。例如，您可以编写策略条件来指定必须使用 SSL 发送所有请求。如果通过特定（AWS 服务例如 AWS CloudFormation）使用服务操作，您还可以使用条件来授予对服务操作的访问权限。有关更多信息，请参阅《IAM 用户指南》中的 IAM JSON 策略元素：条件。
- 使用 IAM Access Analyzer 验证您的 IAM policy，以确保权限的安全性和功能性 – IAM Access Analyzer 会验证新策略和现有策略，以确保策略符合 IAM policy 语言 (JSON) 和 IAM 最佳实践。IAM Access Analyzer 提供 100 多项策略检查和可操作的建议，以帮助您制定安全且功能性强的策略。有关更多信息，请参阅《IAM 用户指南》中的 IAM Acess Analyzer 策略验证。
- Require multi-factor authentication（MFA）[需要多重身份验证（MFA）] – 如果您所处的场景要求您的 AWS 账户 中有 IAM 用户或根用户，请启用 MFA 来提高安全性。若要在调用 API 操作时需要 MFA，请将 MFA 条件添加到您的策略中。有关更多信息，请参阅《IAM 用户指南》中的 配置受 MFA 保护的 API 访问。

有关 IAM 中的最佳实操的更多信息，请参阅《IAM 用户指南》中的 IAM 中的安全最佳实操。

允许用户查看他们自己的权限

该示例说明了您如何创建策略，以允许 IAM 用户查看附加到其用户身份的内联和托管式策略。此策略包括在控制台上完成此操作或者以编程方式使用 AWS CLI 或 AWS API 所需的权限。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
```

```
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

简单读取许可

此示例策略授予用户在 us-east-1 的AWS区域中获取有关标识符为 `a123456789` 的 HTTP 或 WebSocket API 的所有资源的信息的权限。资源 `arn:aws:apigateway:`*`us-east-1`*`::/apis/a123456789/*` 包括 API 的所有子资源，例如授权方和部署。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "apigateway:GET"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/apis/a123456789/*"
      ]
    }
  ]
}
```

仅创建 REQUEST 或 JWT 授权方

此示例策略允许用户仅使用 REQUEST 或 JWT 授权方创建 API，包括通过导入进行操作。在策略的 `Resource` 部分中，`arn:aws:apigateway:us-east-1::/apis/??????????` 要求资源最多具有 10 个字符，这不包括 API 的子资源。此示例在 `ForAllValues` 部分中使用 `Condition`，因为用户可以通过导入 API 同时创建多个授权方。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "OnlyAllowSomeAuthorizerTypes",
      "Effect": "Allow",
      "Action": [
        "apigateway:PUT",
        "apigateway:POST",
        "apigateway:PATCH"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/apis",
        "arn:aws:apigateway:us-east-1::/apis/??????????",
        "arn:aws:apigateway:us-east-1::/apis/*/authorizers",
        "arn:aws:apigateway:us-east-1::/apis/*/authorizers/*"
      ],
      "Condition": {
        "ForAllValues:StringEqualsIfExists": {
          "apigateway:Request/AuthorizerType": [
            "REQUEST",
            "JWT"
          ]
        }
      }
    }
  ]
}
```

要求禁用默认 `execute-api` 终端节点

此示例策略允许用户在 `DisableExecuteApiEndpoint` 为 `true` 的前提下创建、更新或导入 API。当 `DisableExecuteApiEndpoint` 为 `true` 时，客户端无法使用默认 `execute-api` 终端节点来调用 API。

我们使用 `BoolIfExists` 条件来处理调用，以更新未填充 `DisableExecuteApiEndpoint` 条件键的 API。当用户尝试创建或导入 API 时，`DisableExecuteApiEndpoint` 条件键始终填充。

由于 `apis/*` 资源还捕获授权方或方法等子资源，因此我们明确地将其范围限定为带有 `Deny` 语句的 API。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DisableExecuteApiEndpoint",
      "Effect": "Allow",
      "Action": [
        "apigateway:PATCH",
        "apigateway:POST",
        "apigateway:PUT"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/apis",
        "arn:aws:apigateway:us-east-1::/apis/*"
      ],
      "Condition": {
        "BoolIfExists": {
          "apigateway:Request/DisableExecuteApiEndpoint": true
        }
      }
    },
    {
      "Sid": "ScopeDownToJustApis",
      "Effect": "Deny",
      "Action": [
        "apigateway:PATCH",
        "apigateway:POST",
        "apigateway:PUT"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/apis/*/*"
      ]
    }
  ]
}
```

只允许用户创建或更新私有 REST API

此示例策略使用条件键来要求用户只创建 `PRIVATE` API，并防止可能将 API 从 `PRIVATE` 更改为其他类型（如 `REGIONAL`）的更新。

我们使用 `ForAllValues` 要求每个添加到 API 的 `EndpointType` 必须为 `PRIVATE`。我们使用资源条件键来允许对 API 进行任何更新，只要它是 `PRIVATE` 即可。`ForAllValues` 仅适用于存在条件键的情况。

我们使用非贪婪匹配器 (?) 来显式匹配 API ID，以防止允许授权方等非 API 资源。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ScopePutToPrivateApis",
            "Effect": "Allow",
            "Action": [
                "apigateway:PUT"
            ],
            "Resource": [
                "arn:aws:apigateway:us-east-1::/restapis",
                "arn:aws:apigateway:us-east-1::/restapis/??????????"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "apigateway:Resource/EndpointType": "PRIVATE"
                }
            }
        },
        {
            "Sid": "ScopeToPrivateApis",
            "Effect": "Allow",
            "Action": [
                "apigateway:DELETE",
                "apigateway:PATCH",
                "apigateway:POST"
            ],
            "Resource": [
                "arn:aws:apigateway:us-east-1::/restapis",
                "arn:aws:apigateway:us-east-1::/restapis/??????????"
            ],
            "Condition": {
```

```
                "ForAllValues:StringEquals": {
                    "apigateway:Request/EndpointType": "PRIVATE",
                    "apigateway:Resource/EndpointType": "PRIVATE"
                }
            }
        },
        {
            "Sid": "AllowResourcePolicyUpdates",
            "Effect": "Allow",
            "Action": [
                "apigateway:UpdateRestApiPolicy"
            ],
            "Resource": [
                "arn:aws:apigateway:us-east-1::/restapis/*"
            ]
        }
    ]
}
```

要求 API 路由具有授权

如果路由没有授权，此策略会导致创建或更新路由（包括通过导入）的尝试失败。如果密钥不存在，例如没有创建或更新路由时，则 `ForAnyValue` 的计算结果为 false。我们使用 `ForAnyValue` 是因为可以通过导入来创建多个路由。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowUpdatesOnApisAndRoutes",
      "Effect": "Allow",
      "Action": [
        "apigateway:POST",
        "apigateway:PATCH",
        "apigateway:PUT"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/apis",
        "arn:aws:apigateway:us-east-1::/apis/??????????",
        "arn:aws:apigateway:us-east-1::/apis/*/routes",
        "arn:aws:apigateway:us-east-1::/apis/*/routes/*"
      ]
    },
```

```
    {
      "Sid": "DenyUnauthorizedRoutes",
      "Effect": "Deny",
      "Action": [
        "apigateway:POST",
        "apigateway:PATCH",
        "apigateway:PUT"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/apis",
        "arn:aws:apigateway:us-east-1::/apis/*"
      ],
      "Condition": {
        "ForAnyValue:StringEqualsIgnoreCase": {
          "apigateway:Request/RouteAuthorizationType": "NONE"
        }
      }
    }
  ]
}
```

防止用户创建或更新 VPC 链接

此策略防止用户创建或更新 VPC 链接。利用 VPC 链接，您可以向 VPC 之外的客户端公开 Amazon VPC 之内的资源。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "DenyVPCLink",
      "Effect": "Deny",
      "Action": [
        "apigateway:POST",
        "apigateway:PUT",
        "apigateway:PATCH"
      ],
      "Resource": [
        "arn:aws:apigateway:us-east-1::/vpclinks",
        "arn:aws:apigateway:us-east-1::/vpclinks/*"
      ]
    }
  ]
```

```
}
```

Amazon API Gateway 基于资源的策略示例

有关基于资源的策略示例，请参阅the section called “API Gateway 资源策略示例”。

Amazon API Gateway 身份和访问疑难解答

可以使用以下信息，以帮助您诊断和修复在使用 API Gateway 和 IAM 时可能遇到的常见问题。

主题

- 我无权在 API Gateway 中执行操作
- 我无权执行 iam:PassRole
- 我希望允许我的AWS账户以外的用户访问我的 API Gateway 资源

我无权在 API Gateway 中执行操作

如果您收到错误提示，表明您无权执行某个操作，则您必须更新策略以允许执行该操作。

当 `mateojackson` IAM 用户尝试使用控制台查看有关虚构 *`my-example-widget`* 资源的详细信息，但不拥有虚构 `apigateway:`*`GetWidget`* 权限时，会发生以下示例错误。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform: 
 apigateway:GetWidget on resource: my-example-widget
```

在此情况下，必须更新 `mateojackson` 用户的策略，以允许使用 `apigateway:`*`GetWidget`* 操作访问 *`my-example-widget`* 资源。

如果您需要帮助，请联系 AWS 管理员。您的管理员是提供登录凭证的人。

我无权执行 iam:PassRole

如果您收到一个错误，表明您无权执行 `iam:PassRole` 操作，则必须更新策略以允许您将角色传递给 API Gateway。

有些 AWS 服务 允许将现有角色传递到该服务，而不是创建新服务角色或服务相关角色。为此，您必须具有将角色传递到服务的权限。

当名为 `marymajor` 的 IAM 用户尝试使用控制台在 API Gateway 中执行操作时，会发生以下示例错误。但是，服务必须具有服务角色所授予的权限才可执行此操作。Mary 不具有将角色传递到服务的权限。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

在这种情况下，必须更新 Mary 的策略以允许她执行 `iam:PassRole` 操作。

如果您需要帮助，请联系您的 AWS 管理员。您的管理员是提供登录凭证的人。

我希望允许我的AWS账户以外的用户访问我的 API Gateway 资源

您可以创建一个角色，以便其他账户中的用户或您组织外的人员可以使用该角色来访问您的资源。您可以指定谁值得信赖，可以担任角色。对于支持基于资源的策略或访问控制列表（ACL）的服务，您可以使用这些策略向人员授予对您的资源的访问权。

要了解更多信息，请参阅以下内容：

- 要了解 API Gateway 是否支持这些功能，请参阅 Amazon API Gateway 如何与 IAM 配合使用。
- 要了解如何为您拥有的AWS 账户中的资源提供访问权限，请参阅IAM 用户指南中的为您拥有的另一个AWS 账户中的 IAM 用户提供访问权限。
- 要了解如何为第三方AWS 账户提供您的资源的访问权限，请参阅 IAM 用户指南中的为第三方拥有的AWS 账户提供访问权限。
- 要了解如何通过联合身份验证提供访问权限，请参阅 IAM 用户指南中的为经过外部身份验证的用户（联合身份验证）提供访问权限。
- 要了解使用角色和基于资源的策略进行跨账户访问之间的差别，请参阅《IAM 用户指南》中的 IAM 角色与基于资源的策略有何不同。

使用 API Gateway 的服务相关角色

Amazon API Gateway 使用 AWS Identity and Access Management (IAM) 服务相关角色。服务相关角色是一种与 API Gateway 直接关联的独特类型的 IAM 角色。服务相关角色是由 API Gateway 预定义的，并包含服务代表您调用其他AWS服务所需的所有权限。

您可以使用服务相关角色轻松设置 API Gateway，因为您不必手动添加所需的权限。API Gateway 定义其服务相关角色的权限，除非另外定义，否则只有 API Gateway 可以代入该角色。定义的权限包括信任策略和权限策略，以及不能附加到任何其他 IAM 实体的权限策略。

只有在先删除相关资源后，才能删除服务相关角色。这将保护您的 API Gateway 资源，因为您不会无意中删除对资源的访问权限。

有关支持服务链接角色的其他服务的信息，请参阅[使用 IAM 的 AWS 服务]，并在服务链接角色列中查找是的服务。选择 Yes 与查看该服务的服务相关角色文档的链接。

API Gateway 的服务相关角色权限

API Gateway 使用名为 AWSServiceRoleForAPIGateway 的服务相关角色 – 允许 API Gateway 代表您访问 Elastic Load Balancing、Amazon Data Firehose 和其它服务资源。

AWSServiceRoleForAPIGateway 服务相关角色信任以下服务以代入该角色：

- `ops.apigateway.amazonaws.com`

角色权限策略允许 API Gateway 对指定的资源完成以下操作：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "elasticloadbalancing:AddListenerCertificates",
                "elasticloadbalancing:RemoveListenerCertificates",
                "elasticloadbalancing:ModifyListener",
                "elasticloadbalancing:DescribeListeners",
                "elasticloadbalancing:DescribeLoadBalancers",
                "xray:PutTraceSegments",
                "xray:PutTelemetryRecords",
                "xray:GetSamplingTargets",
                "xray:GetSamplingRules",
                "logs:CreateLogDelivery",
                "logs:GetLogDelivery",
                "logs:UpdateLogDelivery",
                "logs:DeleteLogDelivery",
                "logs:ListLogDeliveries",
                "servicediscovery:DiscoverInstances"
            ],
            "Resource": [
                "*"
            ]
        },
```

```
        {
            "Effect": "Allow",
            "Action": [
                "firehose:DescribeDeliveryStream",
                "firehose:PutRecord",
                "firehose:PutRecordBatch"
            ],
            "Resource": "arn:aws:firehose:*:*:deliverystream/amazon-apigateway-*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "acm:DescribeCertificate",
                "acm:GetCertificate"
            ],
            "Resource": "arn:aws:acm:*:*:certificate/*"
        },
        {
            "Effect": "Allow",
            "Action": "ec2:CreateNetworkInterfacePermission",
            "Resource": "arn:aws:ec2:*:*:network-interface/*"
        },
        {
            "Effect": "Allow",
            "Action": "ec2:CreateTags",
            "Resource": "arn:aws:ec2:*:*:network-interface/*",
            "Condition": {
                "ForAllValues:StringEquals": {
                    "aws:TagKeys": [
                        "Owner",
                        "VpcLinkId"
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:ModifyNetworkInterfaceAttribute",
                "ec2:DeleteNetworkInterface",
                "ec2:AssignPrivateIpAddresses",
                "ec2:CreateNetworkInterface",
                "ec2:DeleteNetworkInterfacePermission",
                "ec2:DescribeNetworkInterfaces",
```

```
                "ec2:DescribeAvailabilityZones",
                "ec2:DescribeNetworkInterfaceAttribute",
                "ec2:DescribeVpcs",
                "ec2:DescribeNetworkInterfacePermissions",
                "ec2:UnassignPrivateIpAddresses",
                "ec2:DescribeSubnets",
                "ec2:DescribeRouteTables",
                "ec2:DescribeSecurityGroups"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": "servicediscovery:GetNamespace",
            "Resource": "arn:aws:servicediscovery:*:*:namespace/*"
        },
        {
            "Effect": "Allow",
            "Action": "servicediscovery:GetService",
            "Resource": "arn:aws:servicediscovery:*:*:service/*"
        }
    ]
}
```

您必须配置权限以允许 IAM 实体（例如，用户、组或角色）创建、编辑或删除服务相关角色。有关更多信息，请参阅《IAM 用户指南》中的服务相关角色权限。

为 API Gateway 创建服务相关角色

您无需手动创建服务相关角色。当您在AWS Management Console、AWS CLI 中或者使用AWS API 创建 API、自定义域名或 VPC 链接时，API Gateway 会再次为您创建服务相关角色。

如果您删除了此服务相关角色然后需要再次创建它，则可以使用相同的流程在您的账户中重新创建此角色。当您创建 API、自定义域名或 VPC 链接时，API Gateway 会再次为您创建服务相关角色。

编辑 API Gateway 的服务相关角色

API Gateway 不允许您编辑 AWSServiceRoleForAPIGateway 服务相关角色。在创建服务相关角色后，您将无法更改角色的名称，因为可能有多种实体引用该角色。不过，您可以使用 IAM 编辑角色的说明。有关更多信息，请参阅《IAM 用户指南》中的编辑服务相关角色。

删除 API Gateway 的服务相关角色

如果您不再需要使用某个需要服务相关角色的功能或服务，我们建议您删除该角色。这样您就没有未被主动监控或维护的未使用实体。但是，您必须先清除服务相关角色的资源，然后才能手动删除它。

> **Note**
>
> 如果在您试图删除资源时 API Gateway 服务正在使用该角色，则删除操作可能会失败。如果发生这种情况，则请等待几分钟后重试。

删除 AWSServiceRoleForAPIGateway 使用的 API Gateway 资源

1. 通过以下网址打开 API Gateway 控制台：https://console.aws.amazon.com/apigateway/。
2. 导航到使用服务相关角色的 API、自定义域名或 VPC 链接。
3. 使用控制台删除资源。
4. 重复此过程以删除使用服务相关角色的所有 API、自定义域名或 VPC 链接。

使用 IAM 手动删除服务相关角色

使用 IAM 控制台、AWS CLI 或AWS API 删除 AWSServiceRoleForAPIGateway 服务相关角色。有关更多信息，请参阅 IAM 用户指南 中的删除服务相关角色。

API Gateway 服务相关角色支持的区域

API Gateway 支持在服务可用的所有区域中使用服务相关角色。有关更多信息，请参阅 AWS 服务端点。

API Gateway 更新到AWS托管策略

查看有关 API Gateway 的 AWS 托管策略更新的详细信息（从该服务开始跟踪这些更改开始）。有关此页面更改的提示，请订阅文档历史记录页面上的 API Gateway RSS 源。

更改	描述	日期
添加了 `acm:GetCertificate` 支持到	`AWSServiceRoleForAPIGateway` 策略现在包	2021 年 7 月 12 日

更改	描述	日期
`AWSServiceRoleForAPIGateway` 策略。	含调用 ACM `GetCertificate` API 操作的权限。	
API Gateway 已开启跟踪更改	API Gateway 为其 AWS 托管策略开启了跟踪更改。	2021 年 7 月 12 日

Amazon API Gateway 中的日志记录和监控

监控是保持 API Gateway 和AWS解决方案的可靠性、可用性和性能的重要环节。您应该从AWS解决方案的各个部分收集监控数据，以便您可以更轻松地调试多点故障（如果发生）。AWS提供了多种工具来监控您的 API Gateway 资源并对潜在事件做出响应。

Amazon CloudWatch Logs

> 为了帮助调试与请求执行或客户端对您 API 访问的相关问题，您可以启用 CloudWatch Logs 以记录 API 调用。有关更多信息，请参阅 the section called “CloudWatch 日志”。

Amazon CloudWatch 警报

> 使用 CloudWatch 警报，可以观看单个指标在指定时间段内的变化。如果指标超过给定阈值，则会向 Amazon Simple Notification Service 主题或 AWS Auto Scaling 策略发送通知。当指标处于特定状态时，CloudWatch 警报不会调用操作。而是必须在状态已改变并在指定的若干个时间段内保持不变后才调用。有关更多信息，请参阅 the section called “CloudWatch 指标”。

访问 Firehose 的日志记录

> 为了协助调试与客户端对您的 API 的访问相关的问题，您可以启用 Firehose 来记录 API 调用。有关更多信息，请参阅 the section called “Firehose”。

AWS CloudTrail

> CloudTrail 提供了用户、角色或AWS服务在 API Gateway 中所执行操作的记录。使用 CloudTrail 收集的信息，您可以确定向 API Gateway 发出了什么请求、发出请求的 IP 地址、何人发出的请求、请求的发出时间以及其他详细信息。有关更多信息，请参阅 the section called “使用 CloudTrail”。

AWS X-Ray

X-Ray 是一项AWS服务，用于收集有关应用程序服务的请求的数据，并使用它来构建服务地图，以便您发现应用程序问题和优化机会。有关更多信息，请参阅 the section called “设置 AWS X-Ray”。

AWS Config

AWS Config 可以提供关于您的账户中的 AWS 资源配置的详细信息。您可以查看资源的关联方式、获取配置更改的历史记录并了解关系和配置如何随时间的推移而变化。您可以使用 AWS Config 定义评估资源配置是否符合数据的规则。AWS Config 规则代表 API Gateway 资源的理想配置设置。如果某个资源违反了某规则并且被标记为“不合规”，则 AWS Config 可能提醒您使用 Amazon Simple Notification Service (Amazon SNS) 主题。有关详细信息，请参阅the section called “使用 AWS Config”。

使用 AWS CloudTrail 记录 Amazon API Gateway API 调用

Amazon API Gateway 与 AWS CloudTrail 集成，后者是一项提供用户、角色或 AWS 服务所采取操作的记录的服务。CloudTrail 将 API Gateway 服务的所有 REST API 调用作为事件捕获。捕获的调用包含来自 API Gateway 控制台的调用和对 API Gateway 服务 API 的代码调用。借助通过 CloudTrail 收集的信息，您可以确定向 API Gateway 发出的请求、发出请求的 IP 地址、请求的发出时间以及其它详细信息。

> Note
>
> 未在 CloudTrail 中记录 TestInvokeAuthorizer 和 TestInvokeMethod。

每个事件或日记账条目都包含有关生成请求的人员信息。身份信息有助于您确定以下内容：

- 请求是使用根用户凭证还是用户凭证发出的。
- 请求是否代表 IAM Identity Center 用户发出。
- 请求是使用角色还是联合用户的临时安全凭证发出的。
- 请求是否由其他 AWS 服务 发出。

当您创建账户并可以自动访问 CloudTrail 事件历史记录时，CloudTrail 在您的 AWS 账户 中处于活动状态。CloudTrail 事件历史记录提供对 AWS 区域 中过去 90 天的已记录管理事件的可查看、可搜索、

可下载和不可变记录。有关更多信息，请参见《AWS CloudTrail 用户指南》的 使用 CloudTrail 事件历史记录。查看事件历史记录不会收取 CloudTrail 费用。

要持续记录您的 AWS 账户 过去 90 天的事件，请创建跟踪或 CloudTrail Lake 事件数据存储。

CloudTrail 跟踪

通过跟踪记录，CloudTrail 可将日志文件传送至 Simple Storage Service (Amazon S3) 存储桶。使用 AWS Management Console 创建的所有跟踪均具有多区域属性。您可以通过使用 AWS CLI 创建单区域或多区域跟踪。建议创建多区域跟踪，因为您可记录您账户中的所有 AWS 区域 的活动。如果您创建单区域跟踪，则只能查看跟踪的 AWS 区域 中记录的事件。有关跟踪的更多信息，请参阅《AWS CloudTrail 用户指南》中的为您的 AWS 账户 创建跟踪和为组织创建跟踪。

通过创建跟踪，您可以从 CloudTrail 免费向您的 Amazon S3 存储桶传送一份正在进行的管理事件的副本，但会收取 Amazon S3 存储费用。有关 CloudTrail 定价的更多信息，请参阅 AWS CloudTrail 定价。有关 Amazon S3 定价的信息，请参阅 Amazon S3 定价。

CloudTrail Lake 事件数据存储

CloudTrail Lake 允许您对事件运行基于 SQL 的查询。CloudTrail Lake 可将基于行的 JSON 格式的现有事件转换为 Apache ORC 格式。ORC 是一种针对快速检索数据进行优化的列式存储格式。事件将被聚合到事件数据存储中，它是基于您通过应用高级事件选择器选择的条件的不可变的事件集合。应用于事件数据存储的选择器用于控制哪些事件持续存在并可供您查询。有关 CloudTrail Lake 的更多信息，请参阅《AWS CloudTrail 用户指南》中的使用 AWS CloudTrail Lake。

CloudTrail Lake 事件数据存储和查询会产生费用。创建事件数据存储时，您可以选择要用于事件数据存储的定价选项。定价选项决定了摄取和存储事件的成本，以及事件数据存储的默认和最长保留期。有关 CloudTrail 定价的更多信息，请参阅 AWS CloudTrail 定价。

CloudTrail 中的 API Gateway 管理事件

管理事件提供对您的 AWS 账户内资源所执行管理操作的相关信息。这些也称为控制层面操作。默认情况下，CloudTrail 会记录管理事件。

Amazon API Gateway 将所有 API Gateway 操作记录为管理事件，但 TestInvokeAuthorizer 和 TestInvokeMethod 除外。有关 API Gateway 记录到 CloudTrail 的 Amazon API Gateway 操作的列表，请参阅 Amazon API Gateway API Reference。

API Gateway 事件示例

一个事件表示一个来自任何源的请求，包括有关所请求的 API 操作、操作的日期和时间、请求参数等方面的信息。CloudTrail 日志文件不是公用 API 调用的有序堆栈跟踪，因此事件不会按任何特定顺序显示。

下面的示例显示了一个 CloudTrail 事件，该事件演示了 API Gateway `GetResource` 操作。

```
{
    Records: [
        {
            eventVersion: "1.03",
            userIdentity: {
                type: "Root",
                principalId: "AKIAI44QH8DHBEXAMPLE",
                arn: "arn:aws:iam::123456789012:root",
                accountId: "123456789012",
                accessKeyId: "AKIAIOSFODNN7EXAMPLE",
                sessionContext: {
                    attributes: {
                        mfaAuthenticated: "false",
                        creationDate: "2015-06-16T23:37:58Z"
                    }
                }
            },
            eventTime: "2015-06-17T00:47:28Z",
            eventSource: "apigateway.amazonaws.com",
            eventName: "GetResource",
            awsRegion: "us-east-1",
            sourceIPAddress: "203.0.113.11",
            userAgent: "example-user-agent-string",
            requestParameters: {
                restApiId: "3rbEXAMPLE",
                resourceId: "5tfEXAMPLE",
                template: false
            },
            responseElements: null,
            requestID: "6d9c4bfc-148a-11e5-81b6-7577cEXAMPLE",
            eventID: "4d293154-a15b-4c33-9e0a-ff5eeEXAMPLE",
            readOnly: true,
            eventType: "AwsApiCall",
            recipientAccountId: "123456789012"
        },
```

```
        ... additional entries ...
    ]
}
```

有关 CloudTrail 记录内容的信息，请参阅《AWS CloudTrail 用户指南》中的 CloudTrail 记录内容。

使用 AWS Config 监控 API Gateway API 配置

您可以使用 AWS Config 来记录对您的 API Gateway API 资源所做的配置更改，并根据资源更改发送通知。维护 API Gateway 资源的配置更改历史记录对于运行问题排查、审计与合规性使用案例非常有用。

AWS Config 可以跟踪以下项的更改：

- API 阶段配置，如：
  - 缓存集群设置
  - 限制设置
  - 访问日志设置
  - 为阶段设置的活动部署
- API 配置，如：
  - 终端节点配置
  - 版本
  - 协议
  - 标签

此外，AWS Config 规则 功能让您能够定义配置规则并自动检测、跟踪和提醒对于这些规则的违反行为。通过跟踪对这些资源配置属性的更改，您还可以为 API Gateway 资源编写更改触发的 AWS Config 规则，并根据最佳实践测试资源配置。

您可以使用 AWS Config 控制台或 AWS Config 为账户启用 AWS CLI。选择要跟踪更改的资源类型。如果您之前配置了 AWS Config 来记录所有资源类型，那么这些 API Gateway 资源将会自动记录到您的账户中。所有AWS 公共区域和 AWS GovCloud (US) 均提供对 AWS Config 中的 Amazon API Gateway 的支持。有关受支持区域的完整列表，请参阅《AWS 一般参考》中的 Amazon API Gateway 端点和限额。

主题

- 设置 AWS Config
- 配置 AWS Config 以记录 API Gateway 资源
- 在 AWS Config 控制台中查看 API Gateway 配置详细信息
- 使用 AWS Config 规则评估 API Gateway 资源

支持的资源类型

以下 API Gateway 资源类型与 AWS Config 集成并记录到 AWS Config 支持的AWS资源类型和资源关系中：

- `AWS::ApiGatewayV2::Api`（WebSocket 和 HTTP API）
- `AWS::ApiGateway::RestApi` (REST API)
- `AWS::ApiGatewayV2::Stage`（WebSocket 和 HTTP API 阶段）
- `AWS::ApiGateway::Stage`（REST API 阶段）

有关 AWS Config 的更多信息，请参阅 AWS Config 开发人员指南。有关定价信息，请参阅 AWS Config 定价信息页。

> ⚠ Important
>
> 如果您在部署 API 后更改以下任一 API 属性，则必须 重新部署 API 以传播更改。否则，您将在 AWS Config 控制台中看到属性更改，但之前的属性设置仍有效；API 的运行时行为将保持不变。
>
> - **`AWS::ApiGateway::RestApi`** – `binaryMediaTypes`, `minimumCompressionSize`, `apiKeySource`
> - **`AWS::ApiGatewayV2::Api`** – `apiKeySelectionExpression`

设置 AWS Config

要初次设置 AWS Config，请参阅 AWS Config 开发人员指南中的以下主题。

- 使用控制台设置 AWS Config
- 使用 AWS CLI 设置 AWS Config

配置 AWS Config 以记录 API Gateway 资源

默认情况下，AWS Config 会记录在环境的运行区域中发现的所有受支持类型的区域性资源的配置更改。您可以自定义 AWS Config，使其仅记录特定资源类型的更改或仅记录对全局资源的更改。

要了解区域与全局资源以及了解如何自定义您的 AWS Config 配置，请参阅选择 AWS Config 记录的资源。

在 AWS Config 控制台中查看 API Gateway 配置详细信息

您可使用 AWS Config 控制台来查找 API Gateway 资源，并获取有关其配置的当前和历史详细信息。以下过程显示了如何查找有关 API Gateway API 的信息。

在 AWS Config 控制台中查找 API Gateway 资源

1. 打开 AWS Config 控制台。
2. 选择 Resources (资源)。
3. 在 Resource (资源) 清单页面上，选择 Resources (资源)。
4. 打开 Resource type (资源类型) 菜单，滚动到 APIGateway 或 APIGatewayV2，然后选择一个或多个 API Gateway 资源类型。
5. 选择 Look up (查找)。
6. 选择 AWS Config 显示的资源列表中的一个资源 ID。AWS Config 将显示配置详细信息以及有关所选资源的其他信息。
7. 要查看记录的配置的完整详细信息，请选择 View Details (查看详细信息)。

要了解在此页面上查找资源和查看信息的更多方法，请参阅“AWS Config 开发人员指南”中的查看AWS 资源配置和历史记录。

使用 AWS Config 规则评估 API Gateway 资源

您可以创建 AWS Config 规则，这些规则表示 API Gateway 资源的理想配置设置。您可以使用预定义的 AWS Config 托管规则，也可以定义自定义规则。AWS Config 会持续跟踪对您的资源配置的更改，以确定这些更改是否符合规则中设定的所有条件。AWS Config 控制台将显示您的规则与资源的合规性状态。

如果某个资源违反了某规则并且被标记为“不合规”，AWS Config 可提醒您使用 Amazon Simple Notification Service 开发人员指南 (Amazon SNS) 主题。要以编程方式使用这些 AWS Config 提醒中

的数据，请使用 Amazon Simple Queue Service (Amazon SQS) 队列作为 Amazon SNS 主题的通知终端节点。

要了解有关设置和使用规则的更多信息，请参阅 AWS Config 开发人员指南中的使用规则评估资源。

Amazon API Gateway 的合规性验证

要了解某个 AWS 服务是否在特定合规性计划范围内，请参阅 合规性计划范围内的 AWS 服务，然后选择您感兴趣的合规性计划。有关常规信息，请参阅AWS合规性计划。

您可以使用 AWS Artifact 下载第三方审计报告。有关更多信息，请参阅在 AWS Artifact 中下载报告、。

您使用 AWS 服务 的合规性责任取决于您数据的敏感度、贵公司的合规性目标以及适用的法律法规。AWS 提供以下资源来帮助满足合规性：

- 安全性与合规性快速入门指南 – 这些部署指南讨论了架构注意事项，并提供了在 AWS 上部署以安全性和合规性为重点的基准环境的步骤。
- Amazon Web Services 上的 HIPAA 安全性和合规性架构设计– 该白皮书介绍了公司如何使用 AWS 创建符合 HIPAA 标准的应用程序。

> **Note**
>
> 并非所有 AWS 服务都符合 HIPAA 要求。有关更多信息，请参阅符合 HIPAA 要求的服务参考。

- AWS 合规性资源 – 此业务手册和指南集合可能适用于您的行业和位置。
- AWS 客户合规指南：从合规角度了解责任共担模式。这些指南总结了保护 AWS 服务 的最佳实践，并将指南映射到跨多个框架的安全控制，包括美国国家标准与技术研究院（NIST）、支付卡行业安全标准委员会（PCI）和国际标准化组织（ISO）。
- AWS Config 开发人员指南中的使用规则评估资源 - 此 AWS Config 服务评测您的资源配置对内部实践、行业指南和法规的遵循情况。
- AWS Security Hub – 此 AWS 服务 向您提供 AWS 中安全状态的全面视图。Security Hub 通过安全控件评估您的 AWS 资源并检查其是否符合安全行业标准和最佳实践。有关受支持服务及控件的列表，请参阅 Security Hub 控件参考。
- Amazon GuardDuty – 该 AWS 服务 通过监控您的环境中是否存在可疑和恶意活动，来检测您的 AWS 账户、工作负载、容器和数据面临的潜在威胁。GuardDuty 可以通过满足某些合规性框架规定的入侵检测要求，来协助您满足各种合规性要求，如 PCI DSS。

- AWS Audit Manager – 此 AWS 服务 可帮助您持续审核您的 AWS 使用情况，以简化管理风险以及与相关法规和行业标准的合规性的方式。

Amazon API Gateway 中的弹性

AWS全球基础设施围绕 AWS 区域和可用区构建。AWS区域提供多个在物理上独立且隔离的可用区，这些可用区通过延迟低、吞吐量高且冗余性高的网络连接在一起。利用可用区，您可以设计和操作在可用区之间无中断地自动实现失效转移的应用程序和数据库。与传统的单个或多个数据中心基础设施相比，可用区具有更高的可用性、容错性和可扩展性。

有关 AWS 区域和可用区的更多信息，请参阅 AWS全球基础设施。

为了防止您的 API 被过多的请求所淹没，API Gateway 限制对您的 API 的请求。具体来说，API Gateway 对于一个账户中的所有 API 设置请求提交稳态速率限制和突增限制。您可以为 API 配置自定义限制。要了解更多信息，请参阅 限制 API 请求以获得更高的吞吐量。

您可以使用 Route 53 运行状况检查，来控制从主要区域的 API Gateway API 到辅助区域的 API Gateway API 的 DNS 故障转移。有关示例，请参阅 the section called “DNS 故障转移”。

Amazon API Gateway 中的基础设施安全

作为一项托管式服务，Amazon API Gateway 受 AWS 全球网络安全保护。有关 AWS 安全服务以及 AWS 如何保护基础设施的信息，请参阅 AWS 云安全。要按照基础设施安全最佳实践设计您的 AWS 环境，请参阅《安全性支柱 AWS Well‐Architected Framework》中的 基础设施保护。

您可以使用 AWS 发布的 API 调用通过网络访问 API Gateway。客户端必须支持以下内容：

- 传输层安全性协议 (TLS) 我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 具有完全向前保密 (PFS) 的密码套件，例如 DHE（临时 Diffie-Hellman）或 ECDHE（临时椭圆曲线 Diffie-Hellman）。大多数现代系统（如 Java 7 及更高版本）都支持这些模式。

此外，必须使用访问密钥 ID 和与 IAM 委托人关联的秘密访问密钥来对请求进行签名。或者，您可以使用 AWS Security Token Service（AWS STS）生成临时安全凭证来对请求进行签名。

您可以从任何网络位置调用这些 API 操作，但 API Gateway 不支持基于资源的访问策略，其中可以包含基于源 IP 地址的限制。您还可以使用基于资源的策略以控制来自特定 Amazon Virtual Private Cloud (Amazon VPC) 终端节点或特定 VPC 的访问。事实上，这隔离了在 AWS 网络中仅从特定 VPC 到给定 API Gateway 资源的网络访问。

Amazon API Gateway 中的漏洞分析

配置和 IT 控制是 AWS 和您（我们的客户）之间的共同责任。有关更多信息，请参阅 AWS 责任共担模型。

Amazon API Gateway 中的安全最佳实践

API Gateway 提供了在您开发和实施自己的安全策略时需要考虑的大量安全功能。以下最佳实践是一般准则，并不代表完整的安全解决方案。由于这些最佳实践可能不适合您的环境或不满足您的环境要求，因此将其视为有用的考虑因素而不是惯例。

实施最低权限访问

> 使用 IAM 策略实施最低权限访问，以创建、读取、更新或删除 API Gateway API。要了解更多信息，请参阅“适用于 Amazon API Gateway 的 Identity and Access Management”。API Gateway 提供了多个选项来控制对您创建的 API 的访问。要了解更多信息，请参阅在 API Gateway 中控制和管理对 REST API 的访问、控制和管理对 API Gateway 中 WebSocket API 的访问以及 使用 JWT 授权方控制对 HTTP API 的访问。

实施日志记录

> 使用 CloudWatch Logs 或 Amazon Data Firehose 记录对 API 的请求。要了解更多信息，请参阅监控 REST API、配置 WebSocket API 的日志记录以及 配置 HTTP API 的日志记录。

实施 Amazon CloudWatch 警报

> 使用 CloudWatch 警报，可以观看单个指标在指定时间段内的变化。如果指标超过给定阈值，则会向 Amazon Simple Notification Service 主题或 AWS Auto Scaling 策略发送通知。当指标处于特定状态时，CloudWatch 警报不会调用操作。而是必须在状态已改变并在指定的若干个时间段内保持不变后才调用。有关更多信息，请参阅 the section called “CloudWatch 指标”。

启用 AWS CloudTrail

> CloudTrail 提供了用户、角色或AWS服务在 API Gateway 中所执行操作的记录。使用 CloudTrail 收集的信息，您可以确定向 API Gateway 发出了什么请求、发出请求的 IP 地址、何人发出的请求、请求的发出时间以及其他详细信息。有关更多信息，请参阅 the section called “使用 CloudTrail”。

启用 AWS Config

> AWS Config 可以提供关于您的账户中的 AWS 资源配置的详细信息。您可以查看资源的关联方式、获取配置更改的历史记录并了解关系和配置如何随时间的推移而变化。您可以使用 AWS

Config 定义评估资源配置是否符合数据的规则。AWS Config 规则代表 API Gateway 资源的理想配置设置。如果某个资源违反了某规则并且被标记为“不合规”，则 AWS Config 可能提醒您使用 Amazon Simple Notification Service (Amazon SNS) 主题。有关详细信息，请参阅the section called “使用 AWS Config”。

使用 AWS Security Hub

监控 API Gateway 的使用情况，因为它与使用 AWS Security Hub 的安全最佳实践相关。Security Hub 使用安全控件来评估资源配置和安全标准，以帮助您遵守各种合规框架。有关使用 Security Hub 评估 API Gateway 资源的更多信息，请参阅《AWS Security Hub 用户指南》中的 Amazon API Gateway 控件。

为 API Gateway 资源添加标签

标签是您或 AWS 为 AWS 资源分配的元数据标记。每个标签具有两个部分：

- 标签键（例如，`CostCenter`、`Environment` 或 `Project`）。标签键区分大小写。
- 一个称为标签值 的可选字段（例如，`111122223333` 或 `Production`）。省略标签值与使用空字符串相同。与标签键一样，标签值区分大小写。

标签可帮助您：

- 根据分配给资源的标签控制对资源的访问。您可以通过在 AWS Identity and Access Management (IAM) 策略的条件中指定标签键和值来控制访问权限。有关基于标签的访问控制的更多信息，请参阅 IAM 用户指南 中的使用标签控制访问。
- 跟踪您的 AWS 成本。您可以在 AWS Billing and Cost Management 控制面板上激活这些标签。AWS 使用标签对您的成本进行分类，并向您提供每月成本分配报告。有关更多信息，请参阅 AWS Billing 用户指南中的使用成本分配标签。
- 标识和整理您的 AWS 资源。许多 AWS 服务支持标记，因此，您可以将同一标签分配给来不同服务的资源，以指示这些资源是相关的。例如，您可以将相同的标签分配给您分配给 CloudWatch Events 规则的 API Gateway 阶段。

有关使用标签的提示，请参阅白皮书 AWS 标记策略。

以下各部分提供有关 Amazon API Gateway 的标签的更多信息。

主题

可以标记的 API Gateway 资源

可以在 Amazon API Gateway V2 API 中的以下 HTTP API 或 WebSocket API 资源上设置标签：

- `Api`
- `DomainName`
- `Stage`

- `VpcLink`

此外，在 Amazon API Gateway V1 API 中可以在以下 REST API 资源上设置标签：

- `ApiKey`
- `ClientCertificate`
- `DomainName`
- `RestApi`
- `Stage`
- `UsagePlan`
- `VpcLink`

不能直接对其他资源设置标签。但是，在 Amazon API Gateway V1 API 中，子资源继承对父资源设置的标签。例如：

- 如果对 `RestApi` 资源设置标签，该标签会由 `RestApi` 的以下子资源继承以用于基于属性的访问控制：
  - `Authorizer`
  - `Deployment`
  - `Documentation`
  - `GatewayResponse`
  - `Integration`
  - `Method`
  - `Model`
  - `Resource`
  - `ResourcePolicy`
  - `Setting`
  - `Stage`
- 如果对 `DomainName` 设置标签，则此标签将由其下的任意 `BasePathMapping` 资源继承。
- 如果对 `UsagePlan` 设置标签，则此标签将由其下的任意 `UsagePlanKey` 资源继承。

Note

标签继承仅适用于基于属性的访问控制。例如，您不能使用继承的标签来监控 AWS Cost Explorer 中的成本。当您调用某个资源的 GetTags 时，API Gateway 不会返回继承的标签。

Amazon API Gateway V1 API 中的标签继承

以前，只能对阶段设置标签。现在，您还可以对其他资源设置标签，`Stage` 可以通过两种方式接收标签：

- 可以直接对 `Stage` 设置标签。
- 阶段可以从其父 `RestApi` 继承标签。

如果某个阶段以两种方式收到标签，直接对阶段设置的标签优先。例如，假设一个阶段从其父 REST API 继承以下标签：

```
{
 'foo': 'bar',
 'x':'y'
}
```

假设它也直接对此阶段设置了以下标签：

```
{
 'foo': 'bar2',
 'hello': 'world'
}
```

净结果是阶段将具有以下标签（具有以下值）：

```
{
 'foo': 'bar2',
 'hello': 'world'
 'x':'y'
}
```

标签限制和使用约定

以下限制和使用约定适用于将标签与 API Gateway 资源一起使用：

- 每个资源最多可以有 50 个标签。
- 对于每个资源，每个标签键都必须是唯一的，每个标签键只能有一个值。
- 最大标签键长度为 128 个 Unicode 字符 (采用 UTF-8 格式)。
- 最大标签值长度为 256 个 Unicode 字符 (采用 UTF-8 格式)。
- 键和值允许使用的字符包括可用 UTF-8 格式表示的字母、数字和空格，以及以下字符：. : + = @ _ / -（连字符）。Amazon EC2 资源允许任何字符。
- 标签键和值区分大小写。最佳实践是，决定利用标签的策略并在所有资源类型中一致地实施该策略。例如，决定是使用 `Costcenter`、`costcenter` 还是 `CostCenter`，以及是否对所有标签使用相同的约定。避免将类似的标签用于不一致的案例处理。
- 对标签禁止使用 `aws:` 前缀；它是为使用 AWS 而保留的。您无法编辑或删除带此前缀的标签键或值。具有此前缀的标签不计入每个资源的标签数限制。

使用标签控制对 API Gateway REST API 资源的访问

AWS Identity and Access Management 策略中的条件是所需语法的一部分，您可以使用它们指定对 API Gateway 资源的权限。有关指定 IAM 策略的详情，请参阅the section called “使用 IAM 权限”。在 API Gateway 中，资源可以具有标签，而且某些操作可以包括标签。在创建 IAM 策略时，您可以使用标签条件键来控制：

- 哪些用户可以基于资源已有的标签对 API Gateway 资源执行操作。
- 哪些标签可以在操作的请求中传递。
- 是否特定标签键可在请求中使用。

通过将标签用于基于属性的访问控制，可以实现比 API 级别控制更精细的控制以及比基于资源的访问控制更动态的控制。可以创建 IAM 策略，以允许或拒绝根据请求中提供的标签（请求标签）或正在操作的资源的标签（资源标签）执行操作。一般而言，资源标签用于已经存在的资源。请求标签适用于您创建新资源时。

有关标签条件键的完整请求和语义，请参阅 IAM 用户指南中的使用标签控制访问。

以下示例演示如何为 API Gateway 用户指定策略中的标签条件。

基于资源标签限制操作

以下示例策略将向用户授予对所有资源执行所有操作的权限，前提是这些资源不具有值为 `prod` 的标签 `Environment`。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "apigateway:*",
      "Resource": "*"
    },
    {
      "Effect": "Deny",
      "Action": [
        "apigateway:*"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Environment": "prod"
        }
      }
    }
  ]
}
```

基于资源标签允许操作

以下示例策略允许用户针对 API Gateway 资源执行所有操作，前提是这些资源具有值为 `Development` 的标签 `Environment`。`Deny` 语句防止用户更改 `Environment` 标签的值。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ConditionallyAllow",
      "Effect": "Allow",
      "Action": [
        "apigateway:*"
      ],
```

```
      "Resource": [
        "arn:aws:apigateway:*::*"
      ],
      "Condition": {
        "StringEquals": {
          "aws:ResourceTag/Environment": "Development"
        }
      }
    },
    {
      "Sid": "AllowTagging",
      "Effect": "Allow",
      "Action": [
        "apigateway:*"
      ],
      "Resource": [
        "arn:aws:apigateway:*::/tags/*"
      ]
    },
    {
      "Sid": "DenyChangingTag",
      "Effect": "Deny",
      "Action": [
        "apigateway:*"
      ],
      "Resource": [
        "arn:aws:apigateway:*::/tags/*"
      ],
      "Condition": {
        "ForAnyValue:StringEquals": {
          "aws:TagKeys": "Environment"
        }
      }
    }
  ]
}
```

拒绝标记操作

以下示例策略允许用户执行所有 API Gateway 操作，更改标签除外。

```
{
  "Version": "2012-10-17",
```

```
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "apigateway:*"
      ],
      "Resource": [
        "*"
      ],
    },
    {
      "Effect": "Deny",
      "Action": [
        "apigateway:*"
      ],
      "Resource": "arn:aws:apigateway:*::/tags*",
    }
  ]
}
```

允许标记操作

以下示例策略允许用户获取所有 API Gateway 资源并更改这些资源的标签。要获取资源的标签，用户必须拥有该资源的 GET 权限。要更新资源的标签，用户必须拥有该资源的 PATCH 权限。

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "apigateway:GET",
        "apigateway:PUT",
        "apigateway:POST",
        "apigateway:DELETE"
      ],
      "Resource": [
        "arn:aws:apigateway:*::/tags/*",
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
```

```
        "apigateway:GET",
        "apigateway:PATCH",
      ],
      "Resource": [
        "arn:aws:apigateway:*::*",
      ]
    }
  ]
}
```

API 参考

Amazon API Gateway 提供用于创建和部署您自己的 HTTP 和 WebSocket API 的 API。此外，API Gateway API 在标准 AWS 开发工具包中提供。

如果您使用的是 AWS 开发工具包支持的语言，您可能更愿意使用该开发工具包而不是直接使用 API Gateway REST API。开发工具包可简化身份验证、轻松与您的开发环境集成，并可让您轻松访问 API Gateway 命令。

以下是查找 AWS 开发工具包和 API Gateway REST API 参考文档的位置：

- 用于 Amazon Web Services 的工具
- Amazon API Gateway REST API 参考
- Amazon API Gateway WebSocket 和 HTTP API 参考

Amazon API Gateway 配额和重要说明

主题

除非另有说明，否则可根据请求提高配额。要申请提高配额，您可以使用 Service Quotas 或联系 AWS Support 中心。

为某个方法启用授权后，该方法的 ARN（例如，`arn:aws:execute-api:{region-id}:{account-id}:{api-id}/{stage-id}/{method}/{resource}/{path}`）的最大长度为 1600 字节。路径参数值（其大小在运行时确定）可能导致 ARN 长度超过限制。出现这种情况时，API 客户端将收到 `414 Request URI too long` 响应。

> Note
>
> 这将限制使用资源策略时的 URI 长度。当私有 API 中需要资源策略时，这会限制所有私有 API 的 URI 长度。

API Gateway 账户级别配额，每个区域

以下配额按 Amazon API Gateway 中的每个账户、每个区域应用。

资源或操作	默认配额	能否增加
跨 HTTP API、REST API、WebSocket API 和 WebSocket	每秒 10000 个请求 (RPS)，额外的突增容量由令牌存储桶算法提供，使用的最大存储桶容量为 5000 个请求。*	是

资源或操作	默认配额	能否增加
回调 API 的每账户、每区域的限制配额	Note 突增配额由 API Gateway 服务团队根据区域中账户的总体 RPS 配额确定。它不是客户可以控制或请求更改的配额。	
区域 API	600	否
边缘优化的 API	120	否

* 对于以下区域，默认节流配额为 2500 RPS，而默认爆增配额为 1250 RPS：非洲（开普敦）、欧洲地区（米兰）、亚太地区（雅加达）、中东（阿联酋）、亚太地区（海得拉巴）、亚太地区（墨尔本）、欧洲（西班牙）、欧洲（苏黎世）、以色列（特拉维夫）和加拿大西部（卡尔加里）。

HTTP API 配额

以下配额适用于在 API Gateway 中配置和运行 HTTP API。

资源或操作	默认配额	能否增加
每 API 的路由数	300	是
每 API 的集成数	300	否
最大集成超时	30 秒	否
每个 API 的阶段	10	是
每个域的多级 API 映射	200	否
每阶段的标签数	50	否

资源或操作	默认配额	能否增加
请求行和标头值的总组合大小	10240 字节	否
负载大小	10 MB	否
每个区域每个账户的自定义域数量	120	是
访问日志模板大小	3 KB	否
Amazon CloudWatch Logs 日志条目	1MB	否
每个 API 的授权方	10	是
每个授权方的受众	50	否
每个路由的范围	10	否
JSON Web 密钥集端点的超时	1500 毫秒	否
JSON Web 密钥集端点的响应大小	150000 字节	否
OpenID Connect 发现端点的超时	1500 毫秒	否
Lambda 授权方响应超时	10000 毫秒	否
每区域每账户的 VPC 链接数	10	是
每个 VPC 链接的子网数量	10	是

资源或操作	默认配额	能否增加
每个阶段的阶段变量	100	否
阶段变量中以字符数表示的键长度	64	否
阶段变量中以字符数表示的值长度	512	否

用于配置和运行 WebSocket API 的 API Gateway 配额

在 Amazon API Gateway 中配置和运行 WebSocket API 时适用以下配额。

资源或操作	默认配额	能否增加
每区域每账户（跨所有 WebSocket API）的每秒新连接数	500	是
并发连接	不适用 *	不适用
AWS Lambda每个API 的 授权方	10	是
AWS Lambda 授权方结果大小	8 KB	否
每 API 的路由数	300	是
每 API 的集成数	300	是
集成超时	50 毫秒 - 对于所有集成类型（包括 Lambda、Lambda 代理、HTTP、HTTP 代理和AWS集成）均为 29 秒。	否

资源或操作	默认配额	能否增加
每个 API 的阶段	10	是
WebSocket 帧大小	32 KB	否
消息负载大小	128 KB **	否
WebSocket API 的连接持续时间	2 小时	否
空闲连接超时	10 分钟	否
以字符数表示的 WebSocket API 的 URL 长度	4096	否

* API Gateway 不对并发连接强制执行配额。并发连接的最大数量取决于每秒的新连接速率和两小时的最长连接持续时间。例如，默认配额为每秒 500 个新连接，如果客户端在两小时内以最大速率连接，则 API Gateway 最多可提供 3600000 个并发连接。

** 由于 WebSocket 帧大小配额为 32 KB，因此必须将大于 32 KB 的消息拆分为多个帧，每个帧大小为 32 KB 或更小。这适用于 `@connections` 命令。如果接收到更大的消息（或更大的帧大小），则连接会关闭并显示代码 1009。

用于配置和运行 REST API 的 API Gateway 配额

在 Amazon API Gateway 中配置和运行 REST API 时适用以下配额。对于 restapi:import 或 restapi:put，API 定义文件的最大大小为 6 MB。

只能对特定 API 提高所有的每 API 配额。

资源或操作	默认配额	能否增加
每个区域每个账户的自定义域名数	120	是

资源或操作	默认配额	能否增加
每个域的多级 API 映射	200	否
以字符数表示的边缘优化 API 的 URL 长度	8192	否
以字符数表示的区域 API 的 URL 长度	10240	否
每个区域每个账户的私有 API 数	600	否
API Gateway 资源策略的字符长度	8192	是
每区域每账户的 API 密钥数	10000	否
每区域每账户的客户端证书数	60	是
每个 API 的授权方（AWS Lambda 和 Amazon Cognito）	10	是
每个 API 的文档部分	2000	是
每个 API 的资源	300	是
每个 API 的阶段	10	是
每个阶段的阶段变量	100	否

资源或操作	默认配额	能否增加
阶段变量中以字符数表示的键长度	64	否
阶段变量中以字符数表示的值长度	512	否
每区域每账户的使用计划数	300	是
每个 API 密钥的使用计划	10	是
每区域每账户的 VPC 链接数	20	是
API 缓存 TTL	默认情况下为 300 秒，可由 API 所有者配置为 0 到 3600 秒之间。	上限 (3600) 处不可增加
缓存响应大小	1048576 字节。缓存数据加密可能会增加正在缓存的项目的大小。	否
集成超时	50 毫秒 - 对于所有集成类型（包括 Lambda、Lambda 代理、HTTP、HTTP 代理和AWS集成）均为 29 秒。	是*
所有标头值的总组合大小	10240 字节	否
私有 API 的所有标头值的总组合大小	8000 字节	否
负载大小	10 MB	否
每阶段的标签数	50	否

资源或操作	默认配额	能否增加
映射模板的 `#foreach ... #end` 循环中迭代的数量	1000	否
带授权方法的 ARN 长度	1600 字节	否
使用计划中某个阶段方法一级的节流设置	20	是
每个 API 的模型大小	400KB	否
信任存储中的证书数量	1000 个证书，对象总大小不超过 1 MB。	否

* 您不能将集成超时设置为小于 50 毫秒。您可以将区域 API 和私有 API 的集成超时提高到 29 秒以上，但这可能需要降低账户级别的节流配额限制。

API Gateway 在创建、部署和管理 API 方面的配额

以下固定配额适用于使用 AWS CLI、API Gateway 控制台或 API Gateway REST API 及其开发工具包在 API Gateway 中创建、部署和管理 API。这些配额不能提高。

操作	默认配额	能否增加
CreateApiKey	每账户每秒 5 个请求	否
CreateDeployment	每账户每 5 秒 1 个请求	否
CreateDocumentationVersion	每账户每 20 秒 1 个请求	否
CreateDomainName	每账户每 30 秒 1 个请求	否

操作	默认配额	能否增加
CreateResource	每账户每秒 5 个请求	否
CreateRestApi	区域或私有 API • 每账户每 3 秒 1 个请求 边缘优化的 API • 每账户每 30 秒 1 个请求	否
CreateVpcLink (V2)	每账户每 15 秒 1 个请求	否
DeleteApiKey	每账户每秒 5 个请求	否
DeleteDomainName	每账户每 30 秒 1 个请求	否
DeleteResource	每账户每秒 5 个请求	否
DeleteRestApi	每账户每 30 秒 1 个请求	否
GetResources	每账户每 2 秒 5 个请求	否
DeleteVpcLink (V2)	每账户每 30 秒 1 个请求	否
ImportDocumentationParts	每账户每 30 秒 1 个请求	否
ImportRestApi	区域或私有 API • 每账户每 3 秒 1 个请求 边缘优化的 API • 每账户每 30 秒 1 个请求	否
PutRestApi	每账户每秒 1 个请求	否
UpdateAccount	每账户每 20 秒 1 个请求	否

操作	默认配额	能否增加
UpdateDomainName	每账户每 30 秒 1 个请求	否
UpdateUsagePlan	每账户每 20 秒 1 个请求	否
其他操作	无配额到不超过账户总配额。	否
总操作	每秒 10 个请求，突增配额为每秒 40 个请求。	否

Amazon API Gateway 重要说明

主题

Amazon API Gateway 关于 REST API、HTTP API 和 WebSocket API 的重要提示

- 签名版本 4A 不受 Amazon API Gateway 正式支持。

Amazon API Gateway 关于 REST 和 WebSocket API 的重要说明

- API Gateway 不支持跨 REST 和 WebSocket API 共享自定义域名。
- 阶段名称只能包含字母数字字符、连字符和下划线。最大长度为 128 个字符。
- 系统会保留 `/ping` 和 `/sping` 路径用于服务运行状况检查。将这些路径用于带有自定义域的 API 根级资源将无法产生预期的结果。
- 目前，API Gateway 将日志事件限制为 1024 个字节。超过 1024 个字节的日志事件 (如请求和响应正文) 将被 API Gateway 截断，然后再提交到 CloudWatch 日志。

- CloudWatch 指标当前将维度名称和值限制为 255 个有效的 XML 字符。（有关更多信息，请参阅 CloudWatch 用户指南。）维度值是用户定义名称的函数，包括 API 名称、标签（阶段）名称和资源名称。选择这些名称时，请注意不要超出 CloudWatch 指标限制。
- 映射模板的最大大小为 300 KB。

Amazon API Gateway 关于 WebSocket API 的重要说明

- API Gateway 支持最大 128 KB 的消息负载，最大帧大小为 32 KB。如果消息超过 32 KB，则必须将其拆分为多个帧，每个 32 KB 或更小。如果接收到更大的消息，则连接会关闭并显示代码 1009。

Amazon API Gateway 关于 REST API 的重要说明

- 任何请求 URL 查询字符串都不支持纯文本竖线字符 (|)，必须对该字符进行 URL 编码。
- 不支持分号字符 (`;`) 的任何请求中的 URL 查询字符串和结果数据被拆分。
- REST API 在将 URL 编码的请求参数传递给后端集成之前对其进行解码。对于 UTF-8 请求参数，REST API 会对参数进行解码，然后将其作为 unicode 传递给后端集成。
- 使用 API Gateway 控制台测试 API 时，如果向后端提供自签名证书、证书链中缺少中间证书，或后端引发了任何其他无法识别的证书相关异常情况，那么您可能会收到“未知端点错误”响应。
- 对于具有私有集成的 API `Resource` 或 `Method` 实体，您应删除对 `VpcLink` 的任意硬编码引用。否则，您会具有不固定的集成，并收到错误，说明 VPC 链接仍在使用，即使 `Resource` 或 `Method` 实体已删除。在私有集成通过阶段变量引用 `VpcLink` 时，此行为不适用。
- 以下后端可能无法通过与 API Gateway 兼容的方式来支持 SSL 客户端身份验证：
  - NGINX
  - Heroku
- API Gateway 支持大多数 OpenAPI 2.0 规范和 OpenAPI 3.0 规范，但下列情况除外：
  - 路径段只能包含字母数字字符、下划线、连字符、句点、逗号、冒号和大括号。路径参数必须为单独的路径段。例如，“resource/{path_parameter_name}”为有效；而“resource{path_parameter_name}” 为无效。
  - 模型名称只能包含字母数字字符。
  - 对于输入参数，仅支持以下属性：`name`、`in`、`required`、`type`、`description`。其他属性将被忽略。
  - `securitySchemes` 类型（如果使用）必须为 `apiKey`。但是，支持通过 Lambda 授权方进行 OAuth 2 和 HTTP 基本身份验证；OpenAPI 配置通过供应商扩展实现。

- `deprecated` 字段不受支持且将在导出的 API 中删除。
- API Gateway 模型是使用 JSON 架构草案 4 定义的，而不是 OpenAPI 使用的 JSON 架构。
- 任何架构对象均不支持 `discriminator` 参数。
- 不支持 `example` 标记。
- `exclusiveMinimum`API Gateway 不支持 。
- 简单请求验证中不包括 `maxItems` 和 `minItems` 标记。要解决此问题，请在导入之后、执行验证之前更新模型。
- `oneOf` 不支持 OpenAPI 2.0 或生成 SDK。
- 不支持 `readOnly` 字段。
- `$ref` 不能用于引用其他文件。
- OpenAPI 文档根目录不支持 `"500": {"$ref": "#/responses/UnexpectedError"}` 表单的响应定义。要解决此问题，请使用内联架构替换参考。
- 不支持 `Int32` 或 `Int64` 类型的数字。下面展示了一个示例：

```
"elementId": {
    "description": "Working Element Id",
    "format": "int32",
    "type": "number"
}
```

- 架构定义不支持十进制数字格式类型 (`"format": "decimal"`)。
- 在方法响应中，架构定义必须为对象类型，不能是基元类型。例如，不支持 `"schema": { "type": "string"}`。但是，您可以使用以下对象类型解决此问题：

```
"schema": {
     "$ref": "#/definitions/StringResponse"
            }

 "definitions": {
    "StringResponse": {
      "type": "string"
    }
  }
```

- API Gateway 不使用在 OpenAPI 规范中定义的根级别安全性。因此，需要在操作级别上定义安全性以正确应用。
- 不支持该 `default` 关键字。

- 当使用 Lambda 集成或 HTTP 集成处理方法时，API Gateway 会施加以下限制：
  - 对于标头名称和查询参数，会以区分大小写的方式进行处理。
  - 下表列出了在发送到集成端点或由集成端点发回时，可能会被删除、重新映射或以其他方式修改的标头：在此表中：
    - `Remapped` 表示标头名称从 *<string>* 更改为 `X-Amzn-Remapped-`*<string>*。

      `Remapped Overwritten` 表示标头名称从 *<string>* 更改为 `X-Amzn-Remapped-`*<string>*，并且值被覆盖。

标头名称	请求 (**http/http_proxy /lambda**)	响应 (**http/http_pr y /lambda**)
Age	传递	传递
Accept	传递	已删除/传递/传递
Accept-Charset	传递	传递
Accept-Encoding	传递	传递
Authorization	传递*	已重新映射
Connection	传递/传递/已删除	已重新映射
Content-Encoding	传递/已删除/传递	传递
Content-Length	传递（基于正文生成）	传递

标头名称	请求 (**http**/**http_proxy** /**lambda**)	响应 (**http**/**http_pr** **y** /**lambda**)
`Content-MD5`	已删除	已重新映射
`Content-Type`	传递	传递
`Date`	传递	重新映射被覆盖
`Expect`	已删除	已删除
`Host`	已覆盖到集成端点	已删除
`Max-Forwards`	已删除	已重新映射
`Pragma`	传递	传递
`Proxy-Authenticate`	已删除	已删除
`Range`	传递	传递
`Referer`	传递	传递
`Server`	已删除	重新映射被覆盖
`TE`	已删除	已删除
`Transfer-Encoding`	已删除/已删除/例外	已删除
`Trailer`	已删除	已删除

标头名称	请求 (**http**/**http_proxy** /**lambda**)	响应 (**http**/**http_pr y** /**lambda**)
`Upgrade`	已删除	已删除
`User-Agent`	传递	已重新映射
`Via`	已删除/已删除/传递	传递/已删除/已删除
`Warn`	传递	传递
`WWW-Authenticate`	已删除	已重新映射

* 如果 `Authorization` 标头包含签名版本 4 签名，或者如果使用 `AWS_IAM` 授权，则会删除此标头。

- 由 API Gateway 生成的 API 的 Android 开发工具包使用 `java.net.HttpURLConnection` 类。如果将 `WWW-Authenticate` 标头重新映射到 `X-Amzn-Remapped-WWW-Authenticate` 导致了 401 响应，那么该类将在运行 Android 4.4 及更早版本的设备上引起未处理的异常。
- 与 API Gateway 生成的 API 的 Java、Android 和 iOS 开发工具包不同，API Gateway 生成的 API 的 JavaScript 开发工具包不支持针对 500 级错误进行重试。
- 方法的测试调用使用默认内容类型 `application/json` 并忽略任何其他内容类型的规范。
- 通过传递 `X-HTTP-Method-Override` 标头将请求发送到 API 时，API Gateway 覆盖方法。因此，要将标头传递到后端，该标头需要添加到集成请求中。
- 如果某个请求的 `Accept` 标头包含多个媒体类型，API Gateway 将只接受第一个 `Accept` 媒体类型。如果无法控制 `Accept` 媒体类型的顺序并且二进制内容的媒体类型不是列表中的第一个，您可以添加 API 的 `Accept` 列表中的第一个 `binaryMediaTypes` 媒体类型，API Gateway 将您的内容以二进制形式返回。例如，要在浏览器中使用 <img> 元素发送 JPEG 文件，该浏览器可能会在请求中发送 `Accept:image/webp,image/*,*/*;q=0.8`。将 `image/webp` 添加到 `binaryMediaTypes` 列表后，端点将收到二进制形式的 JPEG 文件。
- 当前不支持自定义 `413 REQUEST_TOO_LARGE` 的默认网关响应。

- API Gateway 对所有集成响应使用 `Content-Type` 标头。默认情况下，内容类型为 `application/json`。

文档历史记录

下表描述了自 Amazon API Gateway 上一次发布以来对文档所做的重要更改。如需此文档更新的通知，您可以通过在顶部菜单面板中选择 RSS 按钮来订阅 RSS 源。

- 文档最新更新时间：2024 年 2 月 15 日

变更	说明	日期
增加了对 TLS 1.3 的支持	API Gateway 现在可在区域 REST API、HTTP API 和 WebSocket API 上支持 TLS 1.3。有关更多信息，请参阅在 API Gateway 中为自定义域选择安全策略。	2024 年 2 月 15 日
REST API 和 WebSocket API 控制台更新	更新了 REST API 和 WebSocket API 的控制台信息。	2023 年 12 月 10 日
文档更新	更新了概念信息，并为 API Gateway REST API 的数据转换和请求验证主题创建了新的教程。有关更多信息，请参阅在 API Gateway 中使用请求验证和为 REST API 设置数据转换。	2023 年 6 月 22 日
为多区域 API Gateway 配置 DNS 故障转移	添加了以下支持：使用 Amazon Route 53 运行状况检查，控制从主要 AWS 区域中的 API Gateway REST API 到辅助区域中的 API Gateway REST API 的 DNS 故障转移。有关更多信息，请参阅为 DNS	2022 年 10 月 31 日

	故障转移配置自定义运行状况检查。	
文档更新	更新了 REST API 和 HTTP API 的核心功能摘要。有关更多信息，请参阅在 REST API 和 HTTP API 之间进行选择。	2022 年 5 月 31 日
托管式策略更新	添加了 `acm:GetCertificate` 支持到 `AWSServiceRoleForAPIGateway` 策略。有关更多信息，请参阅使用 API Gateway 的服务相关角色。	2021 年 7 月 12 日
HTTP API 的参数映射	添加了对 HTTP API 参数映射的支持。有关更多信息，请参阅转换 API 请求和响应。	2021 年 1 月 7 日
禁用 REST API 的默认端点	添加了对禁用 REST API 的默认端点的支持。有关更多信息，请参阅禁用 REST API 的默认端点。	2020 年 10 月 29 日
双向 TLS 身份验证	增加了对 REST API 和 HTTP API 的双向 TLS 身份验证的支持。有关更多信息，请参阅为 REST API 配置双向 TLS 身份验证和为 HTTP API 配置双向 TLS 身份验证。	2020 年 9 月 17 日
HTTP API AWS Lambda 授权方	添加了对 HTTP API 的 AWS Lambda 授权方的支持。有关更多信息，请参阅与 HTTP API AWS Lambda 授权方协同工作。	2020 年 9 月 9 日

HTTP API AWS 服务集成	添加了对适用于 HTTP API 的 AWS 服务集成的支持。有关更多信息，请参阅使用适用于 HTTP API 的AWS服务集成。	2020 年 8 月 20 日
HTTP API 通配符自定义域	添加了对 HTTP API 的通配符自定义域名的支持。有关更多信息，请参阅通配符自定义域名。	2020 年 8 月 10 日
无服务器开发人员门户改进	已将用户管理添加到管理员面板，并支持导出 API 定义。有关更多信息，请参阅使用无服务器开发人员门户为 API Gateway API 编目。	2020 年 6 月 25 日
WebSocket API `Sec-WebSocket-Protocol` 支持	增加了对 `Sec-WebSocket-Protocol` 字段的支持。有关更多信息，请参阅设置需要 WebSocket 子协议的 $connect 路由。	2020 年 6 月 16 日
HTTP API 导出	添加了对导出 HTTP API 的 OpenAPI 3.0 定义的支持。有关更多信息，请参阅从 API Gateway 导出 HTTP API。	2020 年 4 月 20 日
安全性文档	添加了安全性文档。有关更多信息，请参阅 Amazon API Gateway 中的安全性。	2020 年 3 月 31 日
重新组织了文档。	重新组织了开发人员指南。	2020 年 3 月 12 日
HTTP API 正式发布	HTTP API 已正式发布。有关更多信息，请参阅使用 HTTP API。	2020 年 3 月 12 日

HTTP API 日志记录	添加了对 HTTP API 日志中 `$context.integrationErrorMessage` 的支持。有关更多信息，请参阅HTTP API 日志记录变量。	2020 年 2 月 26 日
AWS用于 OpenAPI 导入的变量	在 OpenAPI 定义中添加了对 AWS 变量的支持。有关更多信息，请参阅适用于 OpenAPI 导入的AWS变量。	2020 年 2 月 17 日
HTTP API	在测试版中发布了 HTTP API。有关更多信息，请参阅HTTP API。	2019 年 12 月 4 日
通配符自定义域名	添加了对通配符自定义域名的支持。有关更多信息，请参阅通配符自定义域名。	2019 年 10 月 21 日
Amazon Data Firehose 日志记录	添加了对将 Amazon Data Firehose 作为访问日志记录数据的目标的支持。有关更多信息，请参阅使用 Amazon Data Firehose 作为 API Gateway 访问日志记录的目标。	2019 年 10 月 15 日
用于调用私有 API 的 Route53 别名	添加了对用于调用私有 API 的其他 Route53 别名 DNS 记录的支持。有关更多信息，请参阅使用 Route53 别名访问您的私有 API。	2019 年 9 月 18 日
针对 WebSocket API 的基于标签的访问控制	增加了对 WebSocket API 基于标签的访问控制的支持。有关更多信息，请参阅可加标签的 API Gateway 资源。	2019 年 6 月 27 日

自定义域的 TLS 版本选择	增加了相应支持，可以对部署到自定义域的 API 的传输层安全性 (TLS) 选择版本。请参阅在 API Gateway 中为自定义域选择最小 TLS 版本中的说明。	2019 年 6 月 20 日
针对私有 API 的 VPC 端点策略	增加了对通过附加端点策略到接口 VPC 端点来改进私有 API 安全性的支持。有关更多信息，请参阅在 API Gateway 中为私有 API 使用 VPC 端点策略。	2019 年 6 月 4 日
更新的文档	重新编写了开始使用 Amazon API Gateway。已将教程移至 Amazon API Gateway 教程。	2019 年 5 月 29 日
针对 REST API 的基于标签的访问控制	增加了对 REST API 基于标签的访问控制的支持。有关更多信息，请参阅使用标签与 IAM 策略控制对 API Gateway 资源的访问。	2019 年 5 月 23 日
更新的文档	重新编写了以下 6 个主题：什么是 Amazon API Gateway？、教程：使用 HTTP 代理集成构建 API、教程：使用三个非代理集成创建 Calc REST API、API Gateway 映射模板和访问日志记录变量参考、使用 API Gateway Lambda 授权方和为 API Gateway REST API 资源启用 CORS。	2019 年 4 月 5 日

无服务器开发人员门户改进	增加了管理员面板以及更方便在 Amazon API Gateway 开发人员门户中发布 API 的其他改进。有关更多信息，请参阅使用开发人员门户为您的 API 编目。	2019 年 3 月 28 日
支持 AWS Config	增加了对 AWS Config 的支持。有关更多信息，请参阅使用 AWS Config 监控 API Gateway API 配置。	2019 年 3 月 20 日
支持 AWS CloudFormation	将 API Gateway V2 API 添加到 AWS CloudFormation 模板参考中。有关更多信息，请参阅 Amazon API Gateway V2 资源类型参考。	2019 年 2 月 7 日
支持 WebSocket API	添加了对 WebSocket API 的支持。有关更多信息，请参阅在 Amazon API Gateway 中创建 WebSocket API。	2018 年 12 月 18 日
无服务器开发人员门户可用，通过AWS Serverless Application Repository	现在，除 GitHub 之外，Amazon API Gateway 开发人员门户无服务器应用程序还可通过 AWS Serverless Application Repository 使用。有关更多信息，请参阅使用开发人员门户为您的 API Gateway API 编目。	2018 年 11 月 16 日
支持 AWS WAF	添加了对 AWS WAF（Web 应用程序防火墙）的支持。有关更多信息，请参阅使用 AWS WAF 控制对 API 的访问。	2018 年 11 月 5 日

无服务器开发人员门户	Amazon API Gateway 现在提供了完全可定制的开发人员门户作为无服务器应用程序，您可以部署它以发布您的 API Gateway API。有关更多信息，请参阅使用开发人员门户为您的 API Gateway API 编目。	2018 年 10 月 29 日
支持多值标头和查询字符串参数	Amazon API Gateway 现在支持多个具有相同名称的标头和查询字符串参数。有关更多信息，请参阅支持多值标头和查询字符串参数。	2018 年 10 月 4 日
OpenAPI 支持	Amazon API Gateway 现在支持 OpenAPI 3.0 以及 OpenAPI (Swagger) 2.0。	2018 年 9 月 27 日
更新的文档	添加了一个新主题：Amazon API Gateway 资源策略如何影响授权工作流程。	2018 年 9 月 27 日
活动的 AWS X-Ray 集成	现在，您可以使用 AWS X-Ray 来跟踪和分析用户请求通过您的 API 传输到基础服务时的延迟。有关更多信息，请参阅使用 AWS X-Ray 跟踪 API Gateway API 的执行情况。	2018 年 9 月 6 日
缓存改进	只有当您为 API 阶段启用缓存时，`GET` 方法才会默认启用缓存。这有助于确保您的 API 的安全。您可以通过 覆盖方法设置 为其他方法启用缓存。有关更多信息，请参阅启用 API 缓存以提高响应能力。	2018 年 8 月 20 日

修改了服务限制	已修改几个限制：增加了每账户 API 数。提高了创建/导入/部署 API 的 API 速率限制。已更正一些从每分钟到每秒的速率。有关更多信息，请参阅限制。	2018 年 7 月 13 日
覆盖 API 请求和响应参数以及标头	添加了对覆盖请求标头、查询字符串和路径以及响应标头和状态代码的支持。有关更多信息，请参阅使用映射模板覆盖 API 的请求和响应参数以及标头。	2018 年 7 月 12 日
使用计划的方法级别限制	添加了对设置默认每方法限制以及在使用计划设置中设定单个 API 方法的限制的支持。除了现有的账户级别限制和默认方法级别限制之外，还可以在阶段设置中设定这些设置。有关更多信息，请参阅限制 API 请求以获得更高的吞吐量。	2018 年 7 月 11 日
现在可通过 RSS 获得《API Gateway 开发人员指南》更新通知	HTML 版本的 《API Gateway 开发人员指南》现在支持更新的 RSS 源，此类更新显示在该文档历史记录 页面上。RSS 源包括 2018 年 6 月 27 日及此日期之后发布的更新。在此之前发布的更新仍会显示在该页面上。使用顶部菜单面板中的 RSS 按钮，订阅此源。	2018 年 27 月 6 日

早期更新

下表描述 2018 年 6 月 27 日之前发布的每个 API Gateway 开发人员指南 中的重要变化。

更改	描述	更改日期
私有 API	添加了对您通过接口 VPC 端点公开的私有 API 的支持。您的私有 API 的流量不会离开 Amazon 网络；它与公有 Internet 隔离开来。	2018 年 14 月 6 日
跨账户 Lambda 授权方和集成以及跨账户 Amazon Cognito 用户池授权方	使用另一个 AWS 账户中的 AWS Lambda 函数作为 Lambda 授权方函数或 API 集成后端。或使用 Amazon Cognito 用户池作为授权方。另一个账户可以位于 Amazon API Gateway 可用的任何区域中。有关更多信息，请参阅 the section called “配置跨账户 Lambda 授权方”、the section called “教程：使用跨账户 Lambda 代理集成构建 API” 和 the section called “为 REST API 配置跨账户 Amazon Cognito 授权方”。	2018 年 4 月 2 日
用于 API 的资源策略	使用 API Gateway 资源策略可以允许其他 AWS 账户的用户安全地访问您的 API，或者仅允许从指定的源 IP 地址范围或 CIDR 块调用 API。有关更多信息，请参阅 the section called “使用 API Gateway 资源策略”。	2018 年 4 月 2 日
为 API Gateway 资源添加标签	在 API Gateway 中可使用最多 50 个标签为 API 阶段添加标签，用于 API 请求的成本分配和缓存。有关更多信息，请参阅the section called “设置标签”。	2017 年 12 月 19 日
负载压缩和解压缩	允许使用支持的内容编码之一调用具有压缩负载的 API。如果指定了正文映射模板，则压缩负载需要进行映射。有关更多信息，请参阅 the section called “内容编码”。	2017 年 12 月 19 日
API 密钥来自自定义授权方	从自定义授权方将 API 密钥返回到 API Gateway，以便为需要密钥的 API 方法应用使用计划。有关更多信息，请参阅 the section called “选择一个 API 密钥源”。	2017 年 12 月 19 日
使用 OAuth 2 范围授权	使用 OAuth 2 范围和 `COGNITO_USER_POOLS` 授权方启用对方法调用的授权。有关更多信息，请参阅 the section called “使用 Amazon Cognito 用户池作为 REST API 的授权方”。	2017 年 12 月 14 日

更改	描述	更改日期
私有集成和 VPC 链接	使用 API Gateway 私有集成创建 API，向位于 Amazon VPC 外部的客户端，通过 VpcLink 资源提供对此 VPC 中 HTTP/HTTPS 资源的访问。有关更多信息，请参阅the section called “教程：使用私有集成构建 API” 和the section called “私有集成”。	2017 年 11 月 30 日
为 API 测试部署金丝雀版本	将金丝雀版本添加到现有 API 部署，用于测试 API 的较新版本，同时确保运营中的当前版本保持在相同的阶段。您可以为 Canary 版本设置阶段流量百分比，以及启用在单独的 CloudWatch Logs 日志中记录特定于 Canary 的执行和访问。有关更多信息，请参阅 the section called “设置 Canary 版本部署”。	2017 年 11 月 28 日
访问日志记录	使用您选择的格式，通过从 $context 变量派生的数据记录对 API 的客户端访问。有关更多信息，请参阅 the section called “CloudWatch 日志”。	2017 年 11 月 21 日
API 的 Ruby 开发工具包	为您的 API 生成 Ruby 开发工具包并将其用于调用 API 方法。有关更多信息，请参阅the section called “生成 API 的 Ruby 开发工具包” 和the section called “使用由 API Gateway 为 REST API 生成的 Ruby 开发工具包”。	2017 年 11 月 20 日
区域 API 端点	指定区域 API 端点来为非移动客户端创建 API。非移动客户端，例如 EC2 实例，在与部署 API 的相同 AWS 区域中运行。与边缘优化的 API 一样，您可为区域 API 创建自定义域名。有关更多信息，请参阅the section called “API Gateway 端点类型” 和the section called “设置区域自定义域名”。	2017 年 11 月 2 日
自定义请求授权方	使用自定义请求授权方可提供请求参数中的用户身份验证信息以授权 API 方法调用。请求参数包括标头和查询字符串参数以及阶段和上下文变量。有关更多信息，请参阅 使用 API Gateway Lambda 授权方。	2017 年 9 月 15 日

更改	描述	更改日期
自定义网关响应	针对无法到达集成后端的 API 请求自定义 API Gateway 生成的网关响应。自定义网关消息可以为调用方提供特定于 API 的自定义错误消息，包括返回所需的 CORS 标头，或者也可以将网关响应数据转换为外部交换的格式。有关更多信息，请参阅 设置网关响应以自定义错误响应。	2017 年 6 月 6 日
将 Lambda 自定义错误属性映射到方法响应标头	使用 `integration.response.body` 参数（依赖于 API Gateway 在运行时反序列化字符串化的自定义错误对象），将从 Lambda 返回的单独自定义错误属性映射到方法响应标头参数。有关更多信息，请参阅 处理 API Gateway 中的自定义 Lambda 错误。	2017 年 6 月 6 日
限制提高	账户级别稳态请求速率限制提高至每秒 10000 个请求 (rps)，突发限制提高至 5000 个并发请求。有关更多信息，请参阅 限制 API 请求以获得更高的吞吐量。	2017 年 6 月 6 日
验证方法请求	在 API 级别或方法级别配置基本请求验证程序，以便 API Gateway 验证传入请求。API Gateway 会验证必需的参数是否已设置且未留空，并验证适用负载的格式是否符合配置的模型。有关更多信息，请参阅 在 API Gateway 中使用请求验证。	2017 年 4 月 11 日
与 ACM 集成	针对您的 API 自定义域名使用 ACM 证书。您可以在 AWS Certificate Manager 中创建一个证书，或将现有的 PEM 格式证书导入 ACM。然后，您可以在为 API 设置自定义域名时引用证书的 ARN。有关更多信息，请参阅 为 REST API 设置自定义域名。	2017 年 9 月 3 日
生成并调用 API 的 Java 开发工具包	让 API Gateway 为您的 API 生成 Java 开发工具包，并使用该开发工具包在 Java 客户端中调用 API。有关更多信息，请参阅 使用由 API Gateway 为 REST API 生成的 Java 开发工具包。	2017 年 1 月 13 日

更改	描述	更改日期
集成 AWS Marketplace	通过 AWS Marketplace，在使用计划中将您的 API 作为 SaaS 产品进行销售。使用 AWS Marketplace 扩展您的 API 的范围。依赖 AWS Marketplace 以代表您处理客户账单。让 API Gateway 处理用户授权和使用情况计量。有关更多信息，请参阅 将您的 API 作为 SaaS 进行销售。	2016 年 12 月 1 日
为您的 API 启用文档支持	在 API Gateway 中，为 DocumentationPart 资源中的 API 实体添加文档。将集合 `DocumentationPart` 实例的快照与 API 阶段相关联以创建 DocumentationVersion。通过将文档版本导出到外部文件 (如 Swagger 文件)，发布 API 文档。有关更多信息，请参阅 记录 REST API。	2016 年 12 月 1 日
更新了自定义授权方	现在，客户授权方 Lambda 函数会返回调用方的委托人标识符。该函数还会将其他信息作为 `context` 映射和 IAM 策略的键/值对返回。有关更多信息，请参阅 来自 API Gateway Lambda 授权方的输出。	2016 年 12 月 1 日
支持二进制负载	设置您 API 的 binaryMediaTypes，以支持请求或响应的二进制负载。设置 Integration 或 IntegrationResponse 的 `contentHandling` 属性，以指定是将二进制负载处理为本地二进制 blob、Base64 编码字符串，还是无需修改即直接传递。有关更多信息，请参阅 使用 REST API 的二进制媒体类型。	2016 年 11 月 17 日
通过 API 的代理资源启用与 HTTP 或 Lambda 后端的代理集成。	使用 `{proxy+}` 形式的“贪婪”路径参数和“捕获全部”ANY 方法创建一个代理资源。代理资源分别使用 HTTP 或 Lambda 代理集成与 HTTP 或 Lambda 后端集成。有关更多信息，请参阅 设置具有代理资源的代理集成。	2016 年 9 月 20 日
通过提供一个或多个使用计划，将 API Gateway 内的选定 API 扩展为面向客户的产品服务。	在 API Gateway 内创建使用计划，以使选定的 API 客户端能够按照商定的请求速率和配额访问指定的 API 阶段。有关更多信息，请参阅 创建和使用带 API 密钥的使用计划。	2016 年 8 月 11 日

更改	描述	更改日期
通过 Amazon Cognito 中的用户池启用方法级别授权	在 Amazon Cognito 内创建用户池，并将其用作您自己的身份提供商。您可以将用户池配置为方法级别授权方，以便向在用户池中注册的用户授予访问权限。有关更多信息，请参阅 使用 Amazon Cognito 用户池作为授权方控制对 REST API 的访问。	2016 年 7 月 28 日
在 `AWS/ApiGateway` 命名空间下启用 Amazon CloudWatch 指标和维度。	API Gateway 指标现在已在 `AWS/ApiGateway` 的 CloudWatch 命名空间下进行了标准化。您可以在 API Gateway 控制台和 Amazon CloudWatch 控制台中查看它们。有关更多信息，请参阅 Amazon API Gateway 维度和指标。	2016 年 7 月 28 日
为自定义域名启用证书轮换	借助证书轮换，您可以为自定义域名上传证书和续订过期证书。有关更多信息，请参阅 轮换 ACM 中导入的证书。	2016 年 4 月 27 日
记录对更新的 Amazon API Gateway 控制台的更改。	了解如何使用更新的 API Gateway 控制台创建和设置 API。有关更多信息，请参阅教程：通过导入示例创建 REST API 和教程：使用 HTTP 非代理集成构建 REST API。	2016 年 4 月 5 日
启用“导入 API”功能以根据外部 API 定义创建新的 API 或更新现有 API。	借助“导入 API”功能，您可以通过 API Gateway 扩展上传 Swagger 2.0 中表述的外部 API 定义，从而创建新 API 或更新现有 API。有关“导入 API”的更多信息，请参阅使用 OpenAPI 配置 REST API。	2016 年 4 月 5 日
公开 `$input.body` 变量以访问字符串形式的原始负载和 `$util.parseJson()` 函数，从而将 JSON 字符串转换为映射模板内的 JSON 对象。	有关 `$input.body` 和 `$util.parseJson()` 的更多信息，请参阅 API Gateway 映射模板和访问日志记录变量引用。	2016 年 4 月 5 日

更改	描述	更改日期
启用使方法级别缓存失效的客户端请求，提高请求限制管理。	刷新 API 阶段级别缓存，并使各个缓存条目失效。有关更多信息，请参阅刷新 API Gateway 中的 API 阶段缓存 和使 API Gateway 缓存条目失效。改善管理 API 请求限制方面的控制台体验。有关更多信息，请参阅 限制 API 请求以获得更高的吞吐量。	2016 年 3 月 25 日
使用自定义授权启用和调用 API Gateway API	创建和配置 AWS Lambda 函数以实施自定义授权。该函数会返回一个 IAM 策略文档，该文档会向 API Gateway API 的客户端请求授予“允许”或“拒绝”权限。有关更多信息，请参阅 使用 API Gateway Lambda 授权方。	2016 年 2 月 11 日
使用 Swagger 定义文件和扩展导入和导出 API Gateway API	结合使用 Swagger 规范和 API Gateway 扩展来创建和更新您的 API Gateway API。使用 API Gateway 导入程序导入 Swagger 定义。使用 API Gateway 控制台或 API Gateway 导出 API 将 API Gateway API 导出到 Swagger 定义文件。有关更多信息，请参阅使用 OpenAPI 配置 REST API 和从 API Gateway 导出 REST API。	2015 年 12 月 18 日
将请求、响应正文或正文的 JSON 字段映射到请求或响应参数。	将方法请求正文或其 JSON 字段映射到集成请求的路径、查询字符串或标头。将集成响应正文或其 JSON 字段映射到请求响应的标头。有关更多信息，请参阅 Amazon API Gateway API 请求和响应数据映射参考。	2015 年 12 月 18 日
在 Amazon API Gateway 中使用阶段变量	了解如何将配置属性与 Amazon API Gateway 中 API 的部署阶段相关联。有关更多信息，请参阅 为 REST API 部署设置阶段变量。	2015 年 11 月 5 日
如何：为方法启用 CORS	现在，您可以在 Amazon API Gateway 中更轻松地为方法启用跨源资源共享 (CORS)。有关更多信息，请参阅 为 REST API 资源启用 CORS。	2015 年 11 月 3 日
如何：使用客户端 SSL 身份验证	使用 Amazon API Gateway 生成 SSL 证书，您可以用它来对 HTTP 后端调用进行身份验证。有关更多信息，请参阅 生成和配置 SSL 证书用于后端身份验证。	2015 年 9 月 22 日

更改	描述	更改日期
模拟方法集成	了解如何将 API 与 Amazon API Gateway 进行模拟集成。借助此功能，开发人员可以直接从 API Gateway 生成 API 响应，而无需提前在后端进行最终集成。	2015 年 9 月 1 日
Amazon Cognito 身份支持	Amazon API Gateway 扩展了 `$context` 变量的范围，以便在使用 Amazon Cognito 凭证对请求签名时，它现在会返回有关 Amazon Cognito 身份的信息。此外，我们添加了 `$util` 变量，用于对 JavaScript 中的字符串进行转义，以及对 URL 和字符串进行编码。有关更多信息，请参阅 API Gateway 映射模板和访问日志记录变量引用。	2015 年 8 月 28 日
Swagger 集成	使用 GitHub 上的 Swagger 导入工具将 Swagger API 定义导入 Amazon API Gateway。了解有关 使用基于 OpenAPI 的 API Gateway 扩展 的更多信息，以使用导入工具创建和部署 API 和方法。利用 Swagger 导入程序工具，您还可以更新现有 API。	2015 年 7 月 21 日
映射模板参考	在 `$input` 中，阅读有关 API Gateway 映射模板和访问日志记录变量引用 参数及其函数的信息。	2015 年 7 月 18 日
第一个公开发布版	这是 API Gateway 开发人员指南的第一个公开发行版。	2015 年 7 月 9 日

AWS 术语表

有关最新的 AWS 术语，请参阅 AWS 词汇表参考 中的 AWS 词汇表。