A guia AWS CDK de camadas

AWS Recomendações

AWS Recomendações: A guia AWS CDK de camadas

Table of Contents

O guia de camadas do AWS CDK

Steven Guggenheimer, Amazon Web Services (AWS)

Dezembro de 2023 (histórico do documento)

Um dos principais conceitos por trás do AWS Cloud Development Kit (AWS CDK) é muito parecido com o conceito por trás de se manter aquecido em um dia frio. Esse conceito é chamado de camadas. Em um dia frio, você coloca uma camisa, um casaco e, às vezes, um sobretudo, dependendo do frio. Então, se você entrar e o aquecedor estiver muito quente, você pode tirar uma ou as duas camadas da jaqueta para não sentir muito calor. O AWS CDK usa camadas para fornecer diferentes níveis de abstração para o uso de componentes de nuvem. A disposição em camadas garante que você nunca precise escrever muito código ou ter pouco acesso às propriedades dos recursos ao implantar suas pilhas de infraestrutura como código (IaC).

Se você não usa o AWS CDK, precisa escrever seus modelos do AWS CloudFormation manualmente, ou seja, você está aproveitando apenas uma única camada que o força a escrever muito mais código do que normalmente é necessário. Por outro lado, se o AWS CDK abstrair tudo no CloudFormation que você normalmente não precisa escrever, você não será capaz de lidar com nenhum caso extremo.

Para resolver esse problema, o AWS CDK divide o provisionamento de recursos em três camadas separadas e distintas:

- Camada 1 – A camada do CloudFormation: a camada mais básica em que o recurso do CloudFormation e o recurso do AWS CDK são quase idênticos.
- Camada 2 – A camada selecionada: a camada em que os recursos do CloudFormation são abstraídos em classes programáticas que simplificam grande parte da sintaxe padrão do CloudFormation internamente. Essa camada compõe a maior parte do AWS CDK.
- Camada 3 — A camada padrão: a camada mais abstrata em que você pode usar os blocos de criação fornecidos pelas camadas 1 e 2 para personalizar o código para seu caso de uso específico.

Cada item de cada camada é uma instância de uma classe especial do AWS CDKchamada um `Construct`. De acordo com a documentação da AWS, os contructos são “os blocos básicos de criação das aplicações do AWS CDK. Um constructo representa um “componente de nuvem” e encapsula tudo que o AWS CloudFormation precisa para criar o componente.” Os constructos dentro

dessas camadas são conhecidos como constructos L1, L2 e L3, dependendo da camada a que pertencem. Neste guia, faremos um tour por cada camada do AWS CDK para descobrir para que elas são usadas e por que são importantes.

Este guia é destinado a gerentes técnicos, líderes e desenvolvedores interessados em se aprofundar nos principais conceitos que compõem o trabalho do AWS CDK. O AWS CDK é uma ferramenta pconhecida, mas é muito comum que as equipes percam uma grande parte do que ela tem a oferecer. Ao começar a entender os conceitos descritos neste guia, você pode abrir as portas para um novo mundo de possibilidades e otimizar os processos de provisionamento de recursos de suas equipes.

Neste guia:

Constructos da camada 1

Os constructos L1 são os blocos de criação do AWS CDK e são facilmente distinguidos de outros constructos pelo prefixo `Cfn`. Por exemplo, o pacote do Amazon DynamoDB no AWS CDK contém um constructo `Table`, que é um constructo L2. O constructo L1 correspondente é chamado `CfnTable`, e representa diretamente uma `Table` do CloudFormation DynamoDB. É impossível usar o AWS CDK sem acessar essa primeira camada, embora uma aplicação do AWS CDK normalmente nunca use um constructo L1 diretamente. No entanto, na maioria dos casos, os constructos L2 e L3 que os desenvolvedores estão acostumados a usar dependem muito dos constructos L1. Portanto, você pode pensar nos constructos L1 como a ponte entre o CloudFormation e o AWS CDK.

O único propósito do AWS CDK é gerar modelos do CloudFormation usando linguagens de codificação padrão. Depois de executar o comando cdk synth da CLI e gerar os modelos resultantes do CloudFormation, o trabalho do AWS CDK estará concluído. O comando cdk deploy existe apenas por conveniência, mas o que você está fazendo ao executar esse comando acontece inteiramente no CloudFormation. A peça do quebra-cabeça que converte o código do AWS CDK no formato que o CloudFormation entende é o constructo L1.

O ciclo de vida do AWS CDK–CloudFormation para constructos L1

O processo para criar e usar constructos L1 consiste nestas etapas:

1. O processo de criação do AWS CDK converte as especificações do CloudFormation em código programático na forma de constructos L1.
2. Os desenvolvedores escrevem código que faz referência direta ou indireta aos constructos L1 como parte de uma aplicação do AWS CDK.
3. Os desenvolvedores executam o comando cdk synth para converter o código programático novamente no formato ditado pelas especificações (modelos) do CloudFormation.
4. Os desenvolvedores executam o comando cdk deploy para implantar as pilhas do CloudFormation dentro desses modelos nos ambientes da conta da AWS.

Vamos fazer um pequeno exercício. Acesse o repositório de código aberto do AWS CDK no GitHub, escolha um serviço aleatório da AWS e, em seguida, acesse o pacote do AWS CDK desse serviço (localizado na pasta `packages`, `aws-cdk-lib`, `aws-<servicename>`, `lib`). Neste exemplo, vamos escolher o Amazon S3, mas isso funciona para qualquer serviço. Se você olhar o arquivo index.ts principal desse pacote, verá uma linha que diz:

```
export * from './s3.generated';
```

No entanto, você não verá o arquivo `s3.generated` em nenhum lugar do diretório correspondente. Isso ocorre porque os constructos L1 são gerados automaticamente da especificação de recursos do CloudFormation durante o processo de criação do AWS CDK. Portanto, você verá `s3.generated` no pacote somente depois de executar o comando de criação do AWS CDK para o pacote.

A especificação do recurso AWS CloudFormation

A especificação do recurso do AWS CloudFormation define a infraestrutura como código (IAC) para a AWS e determina como o código nos modelos do CloudFormation é convertido em recursos em uma conta da AWS. Essa especificação define os recursos da AWS no formato JSON em um nível por região. Cada recurso recebe um nome de tipo de recurso exclusivo que segue o formato `provider::service::resource`. Por exemplo, o nome do tipo de recurso para um bucket do Amazon S3 seria `AWS::S3::Bucket`, e o nome do tipo de recurso para um ponto de acesso Amazon S3 seria `AWS::S3::AccessPoint`. Esses tipos de recursos podem ser renderizados em um modelo do CloudFormation usando a sintaxe definida na especificação do recurso do AWS CloudFormation. Quando o processo de criação do AWS CDK é executado, cada tipo de recurso também se torna um constructo L1.

Consequentemente, cada constructo L1 é uma imagem espelhada programática de seu recurso correspondente do CloudFormation. Cada propriedade que você aplicaria em um modelo do CloudFormation está disponível quando você instancia um constructo L1, e cada propriedade necessária do CloudFormation também é exigida como argumento quando você instancia o constructo L1 correspondente. A tabela a seguir compara um bucket do S3, conforme representado em um modelo do CloudFormation, com o mesmo bucket do S3, conforme definido como um constructo L1 do AWS CDK.

Modelo do CloudFormation

```
"amzns3demobucket": {
    "Type": "AWS::S3::Bucket",
    "Properties": {
      "BucketName": "amzn-s3-demo-
bucket",
      "BucketEncryption": {
```

Constructo L1

```
new CfnBucket(this, "amzns3de
mobucket", {
  bucketName: "amzn-s3-demo-bucket",
  bucketEncryption: {
    serverSideEncryptionConfigu
ration: [
      {
```

```
        "ServerSideEncryptionConfig
uration": [
          {
            "ServerSideEncrypt
ionByDefault": {
              "SSEAlgorithm": "AES256"
            }
          }
        ]
      },
      "MetricsConfigurations": [
        {
          "Id": "myConfig"
        }
      ],
      "OwnershipControls": {
        "Rules": [
          {
            "ObjectOwnership":
 "BucketOwnerPreferred"
          }
        ]
      },
      "PublicAccessBlockConfigura
tion": {
        "BlockPublicAcls": true,
        "BlockPublicPolicy": true,
        "IgnorePublicAcls": true,
        "RestrictPublicBuckets": true
      },
      "VersioningConfiguration": {
        "Status": "Enabled"
      }
    }
  }
```

```
        serverSideEncryptionByDefau
lt: {
          sseAlgorithm: "AES256"
        }
      }
    ]
  },
  metricsConfigurations: [
    {
      id: "myConfig"
    }
  ],
  ownershipControls: {
    rules: [
      {
        objectOwnership: "BucketOw
nerPreferred"
      }
    ]
  },
  publicAccessBlockConfiguration: {
    blockPublicAcls: true,
    blockPublicPolicy: true,
    ignorePublicAcls: true,
    restrictPublicBuckets: true
  },
  versioningConfiguration: {
    status: "Enabled"
  }
});
```

Como você pode ver, o constructo L1 é a manifestação exata no código do recurso do CloudFormation. Não há atalhos ou simplificações, então a quantidade de texto padronizado que deve ser escrita é praticamente a mesma. No entanto, uma das grandes vantagens de usar o AWS CDK é que ele ajuda a eliminar grande parte da sintaxe padronizada do CloudFormation. Então, como isso acontece? É aí que entra o constructo L2.

Constructos da camada 2

O repositório de código aberto do AWS CDK é escrito principalmente usando a linguagem de programação TypeScript e consiste em vários pacotes e módulos. A biblioteca de pacotes principal, chamada `aws-cdk-lib`, é aproximadamente dividida em um pacote por serviço da AWS, embora isso nem sempre seja o caso. Conforme discutido anteriormente, os constructos L1 são gerados automaticamente durante o processo de criação. Então, qual é todo esse código que você vê quando olha dentro do repositório? Estes são os constructos L2, que são abstrações dos constructos L1.

Os pacotes também contêm uma coleção de tipos, enums e interfaces do TypeScript, bem como classes auxiliares que adicionam mais funcionalidades, mas todos esses itens fornecem o constructo L2. Todos os constructos L2 chamam seus constructos L1 correspondentes em seus construtores após a instanciação, e o constructo L1 resultante que é criada pode ser acessado da camada 2 da seguinte forma:

```
const role = new Bucket(this, "amzn-s3-demo-bucket", {/*...BucketProps*/});
const cfnBucket = role.node.defaultChild;
```

O constructo L2 pega as propriedades padrão, os métodos de conveniência e outros açúcares sintáticos e os aplica ao constructo L1. Isso elimina grande parte da repetição e da verbosidade necessárias para provisionar recursos diretamente no CloudFormation.

Todos os constructos L2 criam seus constructos L1 correspondentes internamente. No entanto, os constructos L2, na verdade, não estendem os constructos L1. Os constructos L1 e L2 herdam uma classe especial chamada Construct. Na versão 1 do AWS CDK, a classe `Construct` foi incorporada ao kit de desenvolvimento, mas na versão 2 é um pacote independente separado. Isso é para que outros pacotes, como o Cloud Development Kit for Terraform (CDKTF), possam incluí-lo como uma dependência. Qualquer classe que herda a classe `Construct` é um constructo L1, L2 ou L3. Os constructos L2 estendem essa classe diretamente, enquanto os constructos L1 estendem uma classe chamada `CfnResource`, conforme mostrado na tabela a seguir.

Árvore de herança L1	Árvore de herança L2
Constructo L1	Constructo L2
→ classe https://docs.aws.amazon.com/cdk/api/v2/docs/aws-cdk-lib.CfnResource.htmlCfnResource	→ classe Construct

→→ classe abstrata CfnRefElement

→→→ classe abstrata CfnElement

→→→→ classe Construct

Se os constructos L1 e L2 herdam a classe `Construct`, por que os constructos L2 simplesmente não estendem L1? Bem, as classes entre a classe `Construct` e a camada 1 bloqueiam o constructo L1 como uma imagem espelhada do recurso do CloudFormation. Elas contêm métodos abstratos (métodos que as classes downstream devem incluir), como `_toCloudFormation`, o que força o constructo a gerar diretamente a sintaxe do CloudFormation. Os constructos L2 ignoram essas classes e estendem a classe `Construct` diretamente. Isso lhes dá a flexibilidade de abstrair grande parte do código necessário para constructos L1, construindo-os separadamente em seus construtores.

A seção anterior apresentou uma comparação lado a lado de um bucket do S3 de um modelo do CloudFormation, e esse mesmo bucket do S3 renderizado como um constructo L1. Essa comparação mostrou que as propriedades e a sintaxe são quase idênticas, e o constructo L1 economiza apenas três ou quatro linhas em comparação com o constructo do CloudFormation. Agora vamos comparar o constructo L1 com o constructo L2 para o mesmo bucket do S3:

Constructo L1 para o bucket do S3

```
new CfnBucket(this, "amzns3de
mobucket", {
  bucketName: "amzn-s3-demo-bucket",
  bucketEncryption: {
    serverSideEncryptionConfigu
ration: [
      {
        serverSideEncryptionByDefau
lt: {
          sseAlgorithm: "AES256"
        }
      }
    ]
  },
  metricsConfigurations: [
    {
```

Constructo L2 para o bucket do S3

```
new Bucket(this, "amzns3demobucket",
 {
  bucketName: "amzn-s3-demo-bucket",
  encryption: BucketEncryption.S
3_MANAGED,
  metrics: [
    {
      id: "myConfig"
    },
  ],
  objectOwnership: ObjectOwnership.BU
CKET_OWNER_PREFERRED,
  blockPublicAccess: BlockPubl
icAccess.BLOCK_ALL,
  versioned: true
});
```

```
      id: "myConfig"
    }
  ],
  ownershipControls: {
    rules: [
      {
        objectOwnership: "BucketOw
nerPreferred"
      }
    ]
  },
  publicAccessBlockConfiguration: {
    blockPublicAcls: true,
    blockPublicPolicy: true,
    ignorePublicAcls: true,
    restrictPublicBuckets: true
  },
  versioningConfiguration: {
    status: "Enabled"
  }
});
```

Como você pode ver, o constructo L2 tem menos da metade do tamanho do constructo L1. Os constructos L2 usam várias técnicas para realizar essa consolidação. Algumas dessas técnicas se aplicam a um único constructo L2, mas outras podem ser reutilizadas em vários constructos para que sejam separadas em sua própria classe para reutilização. Os constructos L2 consolidam a sintaxe do CloudFormation de várias maneiras, conforme anaçisado nas seções a seguir.

Propriedades padrão

A maneira mais simples de consolidar o código para provisionar um recurso é transformar as configurações de propriedade mais comuns em padrões. O AWS CDK tem acesso a linguagens de programação avançadas e o CloudFormation não, então esses padrões geralmente são de natureza condicional. Às vezes, várias linhas de configuração do CloudFormation podem ser eliminadas do código do AWS CDK porque essas configurações podem ser inferidas dos valores de outras propriedades que são passadas para o constructo.

Estruturas, tipos e interfaces

Embora o AWS CDK esteja disponível em várias linguagens de programação, ele é escrito nativamente em TypeScript, de modo que o sistema de tipos de linguagem é usado para definir os tipos que compõem os constructos L2. Aprofundar-se nesse sistema de tipos está além do escopo deste guia. Consulte a documentação do TypeScript para obter detalhes. Para resumir, um TypeScript `type` descreve o tipo de dados que uma determinada variável contém. Podem ser dados básicos, como uma `string`, ou dados mais complexos, como um `object`. Um TypeScript `interface` é outra forma de expressar o tipo de objeto TypeScript, e um `struct` é outro nome para uma interface.

O TypeScript não usa o termo struct, mas se você olhar na Referência de APIs do AWS CDK, verá que um struct é, na verdade, apenas outra interface do TypeScript dentro do código. A Referência de APIs também se refere a determinadas interfaces como interfaces. Se structs e interfaces são a mesma coisa, por que a documentação do AWS CDK faz uma distinção entre eles?

O que o AWS CDK chama de structs são interfaces que representam qualquer objeto usado por um constructo L2. Isso inclui os tipos de objeto para os argumentos de propriedade que são passados para o constructo L2 durante a instanciação, como `BucketProps` para o constructo do bucket do S3 e `TableProps` para o constructo da tabela do DynamoDB, bem como outras interfaces do TypeScript usadas no AWS CDK. Resumindo, se for uma interface do TypeScript no AWS CDK, e seu nome não for prefixado pela letra `I`, o AWS CDK o chamará de struct.

Por outro lado, o AWS CDK usa o termo interface para representar os elementos básicos de que um objeto simples precisaria para ser considerado uma representação adequada de um determinado constructo ou classe auxiliar. Ou seja, uma interface descreve quais devem ser as propriedades públicas de um constructo L2. Todos os nomes de interface do AWS CDK são nomes de constructos ou classes auxiliares existentes prefixadas pela letra `I`. Todos os constructos L2 estendem a classe `Construct`, mas também implementam a interface correspondente. Portanto, o constructo L2 `Bucket` implementa a interface `IBucket`.

Métodos estáticos

Cada instância de um constructo L2 também é uma instância de sua interface correspondente, mas o inverso não é verdadeiro. Isso é importante ao examinar um struct para ver quais tipos de dados são necessários. Se um struct tiver uma propriedade chamada `bucket`, que requer o tipo de dados`IBucket`, você poderá passar um objeto que contenha as propriedades listadas na interface `IBucket` ou uma instância de um L2 `Bucket`. Qualquer um funcionará. No entanto, se

essa propriedade `bucket` exigir um L2 `Bucket`, você poderá passar somente uma instância `Bucket` nesse campo.

Essa distinção torna-se muito importante quando você importa recursos preexistentes para sua pilha. Você pode criar um constructo L2 para qualquer recurso nativo da sua pilha, mas se precisar referenciar um recurso que foi criado fora da pilha, precisará usar a interface desse constructo L2. Isso porque a criação de um constructo L2 cria um novo recurso, caso ainda não exista um dentro dessa pilha. As referências aos recursos existentes devem ser objetos simples que estejam em conformidade com a interface do constructo L2.

Para facilitar isso na prática, a maioria dos constructos L2 tem um conjunto de métodos estáticos associados a eles que retornam a interface desse constructo L2. Esses métodos estáticos geralmente começam com a palavra `from`. Os dois primeiros argumentos passados para esses métodos são os mesmos argumentos `scope` e `id` necessários para um contructo L2 padrão. No entanto, o terceiro argumento não é `props`, mas sim um pequeno subconjunto de propriedades (ou às vezes apenas uma propriedade) que define uma interface. Por esse motivo, quando você passa um constructo L2, na maioria dos casos, somente os elementos da interface são necessários. Isso é para que você também possa usar recursos importados, sempre que possível.

```
// Example of referencing an external S3 bucket
const preExistingBucket = Bucket.fromBucketName(this, "external-bucket", "name-of-
bucket-that-already-exists");
```

No entanto, você não deve depender muito de interfaces. Você deve importar recursos e usar interfaces diretamente somente quando for absolutamente necessário, porque as interfaces não fornecem muitas das propriedades, como métodos auxiliares, que conferem tanta capacidade a um constructo L2.

Métodos auxiliares

Um constructo L2 é uma classe programática em vez de um objeto simples, portanto, ele pode expor métodos de classe que permitem manipular a configuração do recurso após a instanciação. Um bom exemplo disso é o constructo L2 Role do AWS Identity and Access Management (IAM). Os trechos a seguir mostram duas maneiras de criar o mesmo perfil do IAM usando o constructo L2 `Role`.

Sem um método auxiliar:

```
const role = new Role(this, "my-iam-role", {
```

```
   assumedBy: new FederatedPrincipal('my-identity-provider.com'),
   managedPolicies: [
      ManagedPolicy.fromAwsManagedPolicyName("ReadOnlyAccess")
   ],
   inlinePolicies: {
      lambdaPolicy: new PolicyDocument({
         statements: [
            new PolicyStatement({
               effect: Effect.ALLOW,
               actions: [ 'lambda:UpdateFunctionCode' ],
               resources: [ 'arn:aws:lambda:us-east-1:123456789012:function:my-
function' ]
            })
         ]
      })
   }
});
```

Com um método auxiliar:

```
const role = new Role(this, "my-iam-role", {
   assumedBy: new FederatedPrincipal('my-identity-provider.com')
});

role.addManagedPolicy(ManagedPolicy.fromAwsManagedPolicyName("ReadOnlyAccess"));
role.attachInlinePolicy(new Policy(this, "lambda-policy", {
   policyName: "lambdaPolicy",
   statements: [
      new PolicyStatement({
         effect: Effect.ALLOW,
         actions: [ 'lambda:UpdateFunctionCode' ],
         resources: [ 'arn:aws:lambda:us-east-1:123456789012:function:my-function' ]
      })
   ]
 }));
```

A capacidade de usar métodos de instância para manipular a configuração de recursos após a instanciação dá aos constructos L2 muita flexibilidade adicional em relação à camada anterior. Os constructos L1 também herdam alguns métodos de recursos (como `addPropertyOverride`), mas é somente na camada dois que você obtém métodos projetados especificamente para esse recurso e suas propriedades.

Enumerações

A sintaxe do CloudFormation geralmente exige que você especifique muitos detalhes para provisionar um recurso adequadamente. No entanto, a maioria dos casos de uso geralmente é abrangida por apenas algumas configurações. Representar essas configurações usando uma série de valores enumerados pode reduzir consideravelmente a quantidade necessária de código.

Por exemplo, no exemplo de código L2 do bucket do S3 apresentado anteriormente nesta seção, você precisa usar a propriedade `bucketEncryption` do modelo do CloudFormation para fornecer todos os detalhes, incluindo o nome do algoritmo de criptografia a ser usado. Em vez disso, o AWS CDK fornece o enum `BucketEncryption`, que usa as cinco formas mais comuns de criptografia de bucket e permite que você expresse cada uma usando nomes de variáveis únicas.

E quanto aos casos de exceção que não são abrangidos pelos enums? Um dos objetivos de um constructo L2 é simplificar a tarefa de provisionar um recurso de camada 1, de modo que certos casos de exceção que são menos usados podem não ser suportados na camada 2. Para oferecer suporte a esses casos de exceção, o AWS CDK permite manipular diretamente as propriedades dos recursos subjacentes do CloudFormation usando o método addPropertyOverride. Para saber mais sobre substituições de propriedades, consulte a seção Práticas recomendadas deste guia e a seção Abstrações e portas de escape na documentação do AWS CDK.

Classes auxiliares

Às vezes, um enum não consegue realizar a lógica programática necessária para configurar um recurso para um determinado caso de uso. Nessas situações, o AWS CDK geralmente oferece uma classe auxiliar. Um enum é um objeto simples que oferece uma série de pares de chave/valor, enquanto uma classe auxiliar oferece todos os recursos de uma classe TypeScript. Uma classe auxiliar ainda pode agir como um enum expondo propriedades estáticas, mas essas propriedades poderão então ter seus valores definidos internamente com lógica condicional no construtor da classe auxiliar ou em um método auxiliar.

Portanto, embora ao enum `BucketEncryption` possa reduzir a quantidade de código necessária para definir um algoritmo de criptografia em um bucket do S3, essa mesma estratégia não funcionaria para definir durações de tempo porque simplesmente há muitos valores possíveis para escolher. Criar um enum para cada valor seria muito mais problemático do que vale a pena. Por esse motivo, uma classe auxiliar é usada para as configurações padrão do Bloqueio de Objetos do S3 de um bucket do S3, conforme representado pela classe ObjectLockRetention. `0bjectLockRetention` contém dois métodos estáticos: um para retenção de conformidade e

outro para retenção de governança. Ambos os métodos usam uma instância da classe auxiliar Duration como argumento para expressar a quantidade de tempo para a qual o bloqueio deve ser configurado.

Outro exemplo é a classe auxiliar Runtime do AWS Lambda. À primeira vista, pode parecer que as propriedades estáticas associadas a essa classe podem ser tratadas por um enum. No entanto, internamente, cada valor de propriedade representa uma instância da própria classe `Runtime`, portanto, a lógica executada no construtor da classe não poderia ser alcançada em um enum.

Constructos da camada 3

Se as construções L1 realizam uma tradução literal de CloudFormation recursos em código programático e as construções L2 substituem grande parte da CloudFormation sintaxe detalhada por métodos auxiliares e lógica personalizada, o que as construções L3 fazem? A resposta para isso é limitada apenas pela sua imaginação. Você pode criar a camada 3 para se adequar a qualquer caso de uso específico. Se seu projeto precisar de um recurso que tenha um subconjunto específico de propriedades, você poderá criar um constructo L3 reutilizável para atender a essa necessidade.

Os constructos L3 são chamados de padrões no AWS CDK. Um padrão é qualquer objeto que estende a `Construct` classe no AWS CDK (ou estende uma classe que estende a `Construct` classe) para realizar qualquer lógica abstrata além da camada 2. Ao usar a AWS CDK CLI para executar cdk init para iniciar um novo AWS CDK projeto, você deve escolher entre três tipos de AWS CDK aplicativos:`app`, e. `lib` `sample-app`

```
Available templates:
* app: Template for a CDK Application
   └ cdk init app --language=[csharp|fsharp|go|java|javascript|python|typescript]
* lib: Template for a CDK Construct Library
   └ cdk init lib --language=typescript
* sample-app: Example CDK Application with some constructs
   └ cdk init sample-app --language=[csharp|fsharp|go|java|javascript|python|typescript]
```

`app`e `sample-app` ambos representam AWS CDK aplicativos clássicos nos quais você cria e implanta CloudFormation pilhas em ambientes da AWS. Ao escolher`lib`, você está optando por construir uma nova construção L3. `app`e `sample-app` permitem que você escolha qualquer idioma AWS CDK compatível, mas você só pode escolher TypeScript com`lib`. Isso ocorre porque o AWS CDK é escrito de forma nativa TypeScript e usa um sistema de código aberto chamado JSii para traduzir o código original para os outros idiomas suportados. Quando você escolhe `lib` para iniciar seu projeto, você está optando por criar uma extensão para o AWS CDK.

Qualquer classe que estenda a classe `Construct` pode ser um constructo L3, mas os casos de uso mais comuns da camada 3 são interações de recursos, extensões de recursos e recursos personalizados. A maioria dos constructos L3 usa um ou mais desses três casos para estender a funcionalidade do AWS CDK .

Interações de recursos

Uma solução normalmente emprega vários serviços da AWS que funcionam juntos. Por exemplo, uma CloudFront distribuição da Amazon geralmente usa um bucket S3 como origem e AWS WAF como proteção contra explorações comuns. AWS AppSync e o Amazon API Gateway costumam usar tabelas do Amazon DynamoDB como fontes de dados para suas APIs. Um pipeline AWS CodePipeline geralmente usa o Amazon S3 como fonte e AWS CodeBuild para seus estágios de construção. Nesses casos, geralmente é útil criar um único constructo L3 que manipule o provisionamento de dois ou mais constructos L2 interconectados.

Aqui está um exemplo de uma construção L3 que provisiona uma CloudFront distribuição junto com sua origem S3, um AWS WAF registro do Amazon Route 53 e um certificado Gerenciador de certificados da AWS (ACM) para adicionar um endpoint personalizado com criptografia em trânsito, tudo em uma construção reutilizável:

```
// Define the properties passed to the L3 construct
export interface CloudFrontWebsiteProps {
     distributionProps: DistributionProps
     bucketProps: BucketProps
     wafProps: CfnWebAclProps
     zone: IHostedZone
}

// Define the L3 construct
export class CloudFrontWebsite extends Construct {
     public distribution: Distribution

     constructor(
        scope: Construct,
        id: string,
        props: CloudFrontWebsiteProps
     ) {
         super(scope, id);

         const certificate = new Certificate(this, "Certificate", {
            domainName: props.zone.zoneName,
            validation: CertificateValidation.fromDns(props.zone)
         });
         const defaultBehavior = {
            origin: new S3Origin(new Bucket(this, "bucket", props.bucketProps))
         }
```

```
        const waf = new CfnWebACL(this, "waf", props.wafProps);
        this.distribution = new Distribution(this, id, {
           ...props.distributionProps,
           defaultBehavior,
           certificate,
           domainNames: [this.domainName],
           webAclId: waf.attrArn,
         });
     }
}
```

Observe que o Amazon S3 CloudFront, o Route 53 e o ACM usam construções L2, mas a ACL da web (que define regras para lidar com solicitações da web) usa uma construção L1. Isso ocorre porque AWS CDK é um pacote de código aberto em evolução que não está totalmente completo e ainda não existe uma construção L2. `WebAcl` No entanto, qualquer pessoa pode contribuir com a criação AWS CDK de novas construções L2. Então, até que AWS CDK ofereça uma construção L2 para`WebAcl`, você precisa usar uma construção L1. Para criar um novo site usando o constructo L3 `CloudFrontWebsite`, você usa o seguinte código:

```
const siteADotCom = new CloudFrontWebsite(stack, "siteA", siteAProps);
const siteBDotCom = new CloudFrontWebsite(stack, "siteB", siteBProps);
const siteCDotCom = new CloudFrontWebsite(stack, "siteC", siteCProps);
```

Neste exemplo, a construção CloudFront `Distribution` L2 é exposta como uma propriedade pública da construção L3. Ainda haverá casos em que você precisará expor propriedades do L3 como esta, conforme necessário. Na verdade, veremos `Distribution` novamente mais tarde, na seção Recursos personalizados.

AWS CDK Isso inclui alguns exemplos de padrões de interação de recursos, como este. Além do pacote `aws-ecs` que contém os constructos L2 para o Amazon Elastic Container Service (Amazon ECS), o AWS CDK tem um pacote chamado aws-ecs-patterns. Esse pacote contém vários constructos L3 que combinam o Amazon ECS com Application Load Balancers, Network Load Balancers e grupos de destino, ao mesmo tempo em que oferecem versões diferentes que são predefinidas para o Amazon Elastic Compute Cloud (Amazon EC2) e o AWS Fargate. Como muitas aplicações sem servidor usam o Amazon ECS somente com o Fargate, esses constructos L3 oferecem uma conveniência que pode economizar o tempo dos desenvolvedores e o dinheiro dos clientes.

Extensões de recursos

Alguns casos de uso exigem que os recursos tenham configurações padrão específicas que não sejam nativas do constructo L2. No nível da pilha, isso pode ser tratado usando aspects, mas outra maneira conveniente de dar novos padrões a um constructo L2 é estendendo a camada 2. Como um constructo é qualquer classe que herda a classe `Construct`, e os constructos L2 estendem essa classe, você também pode criar um constructo L3 estendendo diretamente um constructo L2.

Isso pode ser especialmente útil para uma lógica de negócios personalizada que ofereça suporte às necessidades singulares de um cliente. Suponhamos que uma empresa tenha um repositório que armazene todo o seu código de AWS Lambda função em um único diretório chamado `src/lambda` e que a maioria das funções do Lambda reutilize sempre o mesmo tempo de execução e nome de manipulador. Em vez de configurar o caminho do código toda vez que você configura uma nova função do Lambda, você pode criar um novo constructo L3:

```
export class MyCompanyLambdaFunction extends Function {
      constructor(
        scope: Construct,
        id: string,
        props: Partial<FunctionProps> = {}
     ) {
         super(scope, id, {
            handler: 'index.handler',
            runtime: Runtime.NODEJS_LATEST,
            code: Code.fromAsset(`src/lambda/${props.functionName || id}`),
            ...props
     });
}
```

Você pode então substituir o constructo L2 `Function` em todos os lugares do repositório da seguinte forma:

```
new MyCompanyLambdaFunction(this, "MyFunction");
new MyCompanyLambdaFunction(this, "MyOtherFunction");
new MyCompanyLambdaFunction(this, "MyThirdFunction", {
   runtime: Runtime.PYTHON_3_11
});
```

Os padrões permitem que você crie novas funções do Lambda em uma única linha, e o constructo L3 é configurado para que você ainda possa substituir as propriedades padrão, se necessário.

Estender constructos L2 diretamente funciona melhor quando você deseja apenas adicionar novos padrões aos constructos L2 existentes. Se você também precisar de outra lógica personalizada, é melhor estender a classe `Construct`. A razão para isso decorre do método `super`, que é chamado dentro do construtor. Em classes que estendem outras classes, o método `super` é usado para chamar o construtor da classe primária, e isso deve ser a primeira coisa que acontece em seu construtor. Isso significa que qualquer manipulação dos argumentos passados ou de outra lógica personalizada só pode acontecer após a criação do constructo L2 original. Se você precisar executar alguma dessas lógicas personalizadas antes de instanciar seu constructo L2, é melhor seguir o padrão descrito anteriormente na seção Interações de recursos.

Recursos personalizados

Os recursos personalizados são um recurso poderoso CloudFormation que permite executar a lógica personalizada a partir de uma função Lambda que é ativada durante a implantação da pilha. Sempre que precisar de algum processo durante a implantação que não seja diretamente suportado CloudFormation, você pode usar um recurso personalizado para que isso aconteça. O AWS CDK oferece classes que também permitem criar recursos personalizados de forma programática. Ao usar recursos personalizados em um construtor L3, você poderá criar um constructo de quase tudo.

Uma das vantagens de usar a Amazon CloudFront são seus fortes recursos globais de armazenamento em cache. Se você quiser redefinir manualmente esse cache para que seu site reflita imediatamente as novas alterações feitas em sua origem, você pode usar uma CloudFront invalidação. No entanto, as invalidações são processos executados em uma CloudFront distribuição em vez de serem propriedades de uma CloudFront distribuição. Elas podem ser criadas e aplicadas a uma distribuição existente a qualquer momento, portanto, não são parte nativa do processo de provisionamento e implantação.

Nesse cenário, talvez você queira criar e executar uma invalidação após cada atualização na origem de uma distribuição. Por causa dos recursos personalizados, você pode criar um constructo L3 que seja parecido com:

```
export interface CloudFrontInvalidationProps {
     distribution: Distribution
     region?: string
     paths?: string[]
}

export class CloudFrontInvalidation extends Construct {
      constructor(
```

```
        scope: Construct,
        id: string,
        props: CloudFrontInvalidationProps
     ) {
          super(scope, id);
          const policy = AwsCustomResourcePolicy.fromSdkCalls({
               resources:AwsCustomResourcePolicy.ANY_RESOURCE
          });
          new AwsCustomResource(scope, `${id}Invalidation`, {
             policy,
             onUpdate: {
                  service: 'CloudFront',
                  action: 'createInvalidation',
                  region: props.region || 'us-east-1',
                  physicalResourceId:
 PhysicalResourceId.fromResponse('Invalidation.Id'),
                  parameters: {
                     DistributionId: props.distribution.distributionId,
                     InvalidationBatch: {
                         Paths: {
                            Quantity: props.paths?.length || 1,
                            Items: props.paths || ['/*']
                         },
                         CallerReference: crypto.randomBytes(5).toString('hex')
                     }
                  }
             }
          }
     }
}
```

Usando a distribuição que criamos anteriormente no constructo L3 `CloudFrontWebsite`, você pode fazer isso com muita facilidade:

```
new CloudFrontInvalidation(this, 'MyInvalidation', {
       distribution: siteADotCom.distribution
});
```

Essa construção L3 usa uma construção AWS CDK L3 chamada AwsCustomResourcepara criar um recurso personalizado que executa a lógica personalizada. `AwsCustomResource`é muito conveniente quando você precisa fazer exatamente uma chamada do AWS SDK, pois permite que você faça isso sem precisar escrever nenhum código Lambda. Se você tiver requisitos mais

complexos e quiser implementar sua própria lógica, poderá usar a CustomResourceclasse básica diretamente.

Outro bom exemplo do AWS CDK uso de uma construção L3 de recurso personalizada é a implantação do bucket do S3. A função Lambda criada pelo recurso personalizado dentro do construtor dessa construção L3 adiciona funcionalidades que não CloudFormation seriam capazes de lidar de outra forma: ela adiciona e atualiza objetos em um bucket do S3. Sem a implantação do bucket do S3, você não conseguiria colocar conteúdo no bucket do S3 que acabou de criar como parte da sua pilha, o que seria muito inconveniente.

O melhor exemplo de como AWS CDK eliminar a necessidade de escrever uma grande quantidade de CloudFormation sintaxe é esse básico: `S3BucketDeployment`

```
new BucketDeployment(this, 'BucketObjects', {
  sources: [Source.asset('./path/to/amzn-s3-demo-bucket')],
  destinationBucket: amzn-s3-demo-bucket
});
```

Compare isso com o CloudFormation código que você precisaria escrever para realizar a mesma coisa:

```
"lambdapolicyA5E98E09": {
   "Type": "AWS::IAM::Policy",
   "Properties": {
    "PolicyDocument": {
     "Statement": [
      {
       "Action": "lambda:UpdateFunctionCode",
       "Effect": "Allow",
       "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function"
      }
     ],
     "Version": "2012-10-17"
    },
    "PolicyName": "lambdaPolicy",
    "Roles": [
     {
      "Ref": "myiamroleF09C7974"
     }
    ]
   },
   "Metadata": {
```

```
    "aws:cdk:path": "CdkScratchStack/lambda-policy/Resource"
   }
  },
  "BucketObjectsAwsCliLayer8C081206": {
   "Type": "AWS::Lambda::LayerVersion",
   "Properties": {
    "Content": {
     "S3Bucket": {
      "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
     },
     "S3Key": "e2277687077a2abf9ae1af1cc9565e6715e2ebb62f79ec53aa75a1af9298f642.zip"
    },
    "Description": "/opt/awscli/aws"
   },
   "Metadata": {
    "aws:cdk:path": "CdkScratchStack/BucketObjects/AwsCliLayer/Resource",
    "aws:asset:path":
"asset.e2277687077a2abf9ae1af1cc9565e6715e2ebb62f79ec53aa75a1af9298f642.zip",
    "aws:asset:is-bundled": false,
    "aws:asset:property": "Content"
   }
  },
  "BucketObjectsCustomResourceB12E6837": {
   "Type": "Custom::CDKBucketDeployment",
   "Properties": {
    "ServiceToken": {
     "Fn::GetAtt": [
      "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C81C01536",
      "Arn"
     ]
    },
    "SourceBucketNames": [
     {
      "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
     }
    ],
    "SourceObjectKeys": [
     "f888a9d977f0b5bdbc04a1f8f07520ede6e00d4051b9a6a250860a1700924f26.zip"
    ],
    "DestinationBucketName": {
     "Ref": "amzn-s3-demo-bucket77F80CC0"
    },
    "Prune": true
   },
```

```
    "UpdateReplacePolicy": "Delete",
    "DeletionPolicy": "Delete",
    "Metadata": {
     "aws:cdk:path": "CdkScratchStack/BucketObjects/CustomResource/Default"
    }
   },
   "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRole89A01265": {
    "Type": "AWS::IAM::Role",
    "Properties": {
     "AssumeRolePolicyDocument": {
      "Statement": [
       {
        "Action": "sts:AssumeRole",
        "Effect": "Allow",
        "Principal": {
         "Service": "lambda.amazonaws.com"
        }
       }
      ],
      "Version": "2012-10-17"
     },
     "ManagedPolicyArns": [
      {
       "Fn::Join": [
        "",
        [
         "arn:",
         {
          "Ref": "AWS::Partition"
         },
         ":iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
        ]
       ]
      }
     ]
    },
    "Metadata": {
     "aws:cdk:path": "CdkScratchStack/
Custom::CDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C/ServiceRole/Resource"
    }
   },

  "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRoleDefaultPolicy88902FDF":
  {
```

```
    "Type": "AWS::IAM::Policy",
    "Properties": {
     "PolicyDocument": {
      "Statement": [
       {
        "Action": [
         "s3:GetBucket*",
         "s3:GetObject*",
         "s3:List*"
        ],
        "Effect": "Allow",
        "Resource": [
         {
          "Fn::Join": [
           "",
           [
            "arn:",
            {
             "Ref": "AWS::Partition"
            },
            ":s3:::",
            {
             "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
            },
            "/*"
           ]
          ]
         },
         {
          "Fn::Join": [
           "",
           [
            "arn:",
            {
             "Ref": "AWS::Partition"
            },
            ":s3:::",
            {
             "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
            }
           ]
          ]
         }
        ]
```

```
      },
      {
       "Action": [
        "s3:Abort*",
        "s3:DeleteObject*",
        "s3:GetBucket*",
        "s3:GetObject*",
        "s3:List*",
        "s3:PutObject",
        "s3:PutObjectLegalHold",
        "s3:PutObjectRetention",
        "s3:PutObjectTagging",
        "s3:PutObjectVersionTagging"
       ],
       "Effect": "Allow",
       "Resource": [
        {
         "Fn::GetAtt": [
          "amzns3demobucket77F80CC0",
          "Arn"
         ]
        },
        {
         "Fn::Join": [
          "",
          [
           {
            "Fn::GetAtt": [
             "amzns3demobucket77F80CC0",
             "Arn"
            ]
           },
           "/*"
          ]
         ]
        }
       ]
      }
     ],
     "Version": "2012-10-17"
    },
    "PolicyName":
"CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRoleDefaultPolicy88902FDF",
    "Roles": [
```

```
     {
      "Ref":
 "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRole89A01265"
     }
    ]
   },
   "Metadata": {
    "aws:cdk:path": "CdkScratchStack/
Custom::CDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C/ServiceRole/DefaultPolicy/
Resource"
   }
  },
  "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C81C01536": {
   "Type": "AWS::Lambda::Function",
   "Properties": {
    "Code": {
     "S3Bucket": {
      "Fn::Sub": "cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}"
     },
     "S3Key": "9eb41a5505d37607ac419321497a4f8c21cf0ee1f9b4a6b29aa04301aea5c7fd.zip"
    },
    "Role": {
     "Fn::GetAtt": [
      "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRole89A01265",
      "Arn"
     ]
    },
    "Environment": {
     "Variables": {
      "AWS_CA_BUNDLE": "/etc/pki/ca-trust/extracted/pem/tls-ca-bundle.pem"
     }
    },
    "Handler": "index.handler",
    "Layers": [
     {
      "Ref": "BucketObjectsAwsCliLayer8C081206"
     }
    ],
    "Runtime": "python3.9",
    "Timeout": 900
   },
   "DependsOn": [

 "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRoleDefaultPolicy88902FDF",
```

```
    "CustomCDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756CServiceRole89A01265"
   ],
   "Metadata": {
    "aws:cdk:path": "CdkScratchStack/
Custom::CDKBucketDeployment8693BB64968944B69AAFB0CC9EB8756C/Resource",
    "aws:asset:path":
 "asset.9eb41a5505d37607ac419321497a4f8c21cf0ee1f9b4a6b29aa04301aea5c7fd",
    "aws:asset:is-bundled": false,
    "aws:asset:property": "Code"
   }
  }
```

4 linhas versus 241 linhas é uma grande diferença! E este é apenas um exemplo do que é possível quando você aproveita a camada 3 para personalizar suas pilhas.

Práticas recomendadas

Constructos L1

- Nem sempre é possível evitar o uso direto de constructos L1, mas você deve evitá-los sempre que possível. Se um constructo L2 específico não for compatível com seu caso de exceção, você pode explorar estas duas opções em vez de usar o constructo L1 diretamente:
  - Acessar `defaultChild`: se a propriedade do CloudFormation de que você precisa não estiver disponível em um constructo L2, você poderá acessar o constructo L1 subjacente usando `L2Construct.node.defaultChild`. Você pode atualizar qualquer propriedade pública do constructo L1 acessando-a por meio dessa propriedade, em vez de ter o trabalho de criar por conta própria o constructo L1.
  - Usar substituições de propriedade: e se a propriedade que você deseja atualizar não for pública? O melhor recurso final de escape que permite ao AWS CDK fazer qualquer coisa que um modelo do CloudFormation possa fazer é usar um método disponível em cada constructo L1: addPropertyOverride. Você pode manipular sua pilha no nível do modelo do CloudFormation passando o nome e o valor da propriedade do CloudFormation diretamente para esse método.

Constructos L2

- Lembre-se de aproveitar os métodos auxiliares que as constructos L2 geralmente oferecem. Com a camada 2, você não precisa passar todas as propriedades na instanciação. Os métodos auxiliares L2 podem tornar o provisionamento de recursos exponencialmente mais conveniente, especialmente quando a lógica condicional é necessária. Um dos métodos auxiliares mais convenientes é derivado da classe Grant. Essa classe não é usada diretamente, mas muitos constructos L2 a usam para fornecer métodos auxiliares que tornam as permissões muito mais fáceis de implementar. Por exemplo, se você quiser dar permissão para uma função L2 do Lambda para acessar um bucket L2 do S3, você pode chamar `s3Bucket.grantReadWrite(lambdaFunction)` em vez de criar um novo perfil e política.

Constructos L3

- Embora os constructos L3 possam ser muito convenientes quando você deseja tornar suas pilhas mais reutilizáveis e personalizáveis, recomendamos que você as use com cautela. Considere de qual tipo de constructo L3 você precisa ou se você realmente precisa de um constructo L3:

- Se você não estiver interagindo diretamente com os recursos da AWS, geralmente é mais apropriado criar uma classe auxiliar em vez de estender a classe `Construct`. Isso ocorre porque a classe `Construct` executa muitas ações por padrão que serão necessárias somente se você estiver interagindo diretamente com os recursos da AWS. Portanto, se você não precisa que essas ações sejam executadas, é mais eficiente evitá-las.
- Se você determinar que a criação de um novo constructo L3 é apropriada, na maioria dos casos, você desejará estender a classe `Construct` diretamente. Estenda outros constructos L2 somente quando quiser atualizar as propriedades padrão do constructo. Se outros constructos L2 ou a lógica personalizada estiverem envolvidos, estenda `Construct` diretamente e instancie todos os recursos no constructo.

Perguntas frequentes

Não posso usar as camadas AWS CDK sem entender?

Pode sim. Mas, como acontece com as ferramentas mais poderosas, elas AWS CDK se tornam mais poderosas quanto mais você as conhece. Aprender como as AWS CDK camadas interagem libera um novo nível de compreensão que ajuda a simplificar suas implantações de pilha muito além do que você pode fazer com apenas o conhecimento básico. AWS CDK

Posso fazer constructos L2 de L1 da mesma forma que faço constructos L3 de L2?

Se um recurso já tiver um constructo L2, recomendamos que você o use e faça suas personalizações na camada 3. Isso ocorre porque muitas pesquisas já foram feitas para descobrir as melhores maneiras de configurar constructos L2 existentes para um recurso específico. No entanto, existem várias constructos L1 cujos constructos L2 ainda não existem. Nesses casos, incentivamos você a criar seus próprios constructos L2 e compartilhá-los com outras pessoas, tornando-se um colaborador da biblioteca de código aberto do AWS CDK . Você pode encontrar tudo o que precisa para começar nas diretrizes de contribuição do AWS CDK.

Quais AWS recursos ainda não têm construções L2 oficiais?

O número de AWS recursos que não têm construções L2 está diminuindo a cada dia, mas se você estiver interessado em ajudar a criar uma construção L2 para um desses recursos, visite a AWS CDK Referência da API. Confira a lista de recursos no painel esquerdo. Os recursos que têm o sobrescrito 1 ao lado de seus nomes não têm constructos L2 oficiais.

Posso criar uma construção L2 ou L3 em qualquer idioma compatível? AWS CDK

O AWS CDK suporta várias linguagens de programação TypeScript JavaScript, incluindo Python, Java, C# e Go. Você pode criar suas construções L3 pessoais usando o AWS CDK código compilado na linguagem relevante. No entanto, se você quiser contribuir AWS CDK ou criar AWS CDK construções nativas, você deve usar TypeScript. Isso ocorre porque TypeScript é o único

idioma nativo do AWS CDK. As AWS CDK versões para outras linguagens são criadas a partir do TypeScript código nativo usando uma AWS biblioteca chamada JSii.

Onde posso encontrar constructos L3 existentes fora do AWS CDK?

Há muitos locais para compartilhar aqui, mas você pode encontrar muitas das construções mais populares no site da AWS Solutions Constructs e na AWS CDK seção do Construct Hub.

Recursos

- AWS CDK Referência de API do
- AWS CloudFormation Especificação do recurso
- Documentação de constructos do AWS CDK
- Abstrações e portas de escape do AWS CDK
- Leverage L2 constructs to reduce the complexity of your AWS CDK application (publicação do Blog da AWS)
- Recursos personalizados do AWS CloudFormation
- Construstos das soluções da AWS
- Hub de constructos
- AWS CDK Examples (repositório do GitHub)

Histórico do documento

A tabela a seguir descreve alterações significativas feitas neste guia. Se desejar receber notificações sobre futuras atualizações, inscreva-se em um feed RSS.

Alteração	Descrição	Data
Publicação inicial	—	4 de dezembro de 2023

As traduções são geradas por tradução automática. Em caso de conflito entre o conteúdo da tradução e da versão original em inglês, a versão em inglês prevalecerá.