



Guide for Desktop Browser Testing

Device Farm desktop browser testing



Device Farm desktop browser testing: Guide for Desktop Browser Testing

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

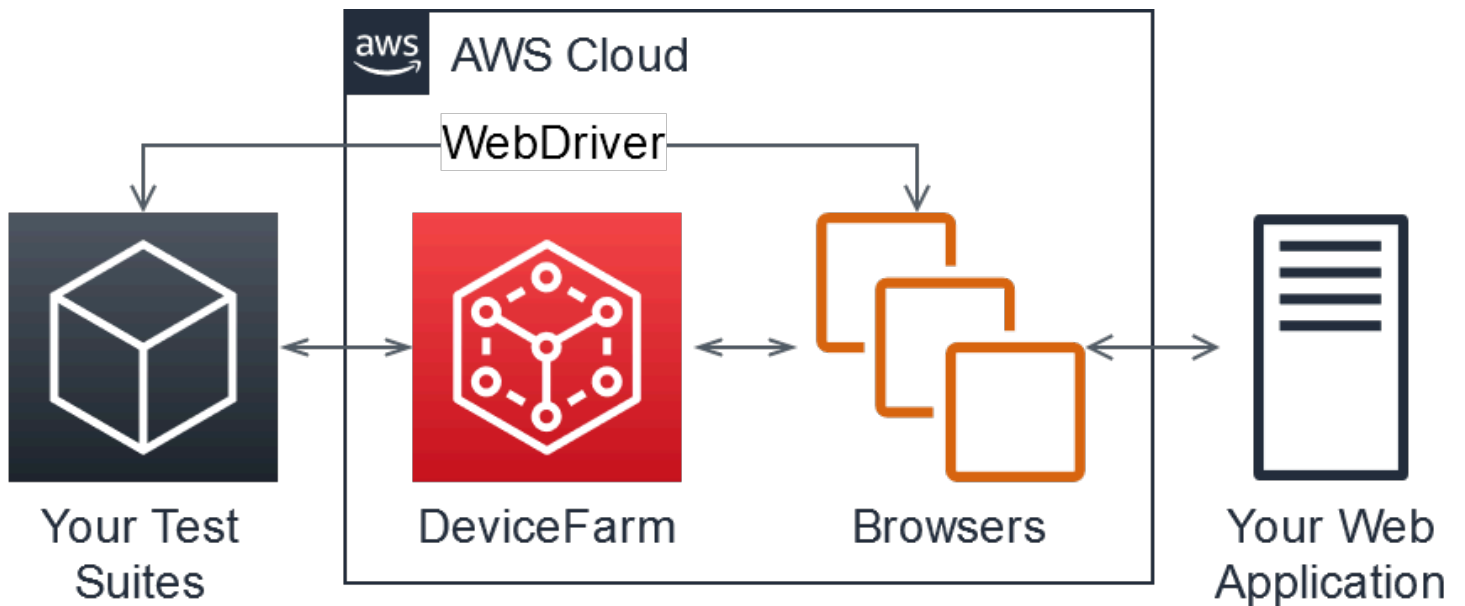
Table of Contents

What is Device Farm desktop browser testing?	1
Before you begin	1
Terminology	2
Accessing Device Farm desktop browser testing	2
Pricing for Device Farm desktop browser testing	2
Getting started	3
Selenium Grid users	3
Local testers	6
Migrating Selenium tests to Device Farm desktop browser testing	13
JUnit	13
Python	16
Node.js	18
Using selenium-webdriver	18
Using Webdriver.IO	19
Ruby	23
Working with Device Farm desktop browser testing	26
Projects	26
Creating a project	26
Deleting a Device Farm desktop browser testing project	28
Sessions	29
Looking up sessions	30
Actions	31
Artifacts	31
Technical reference	33
Browser support	33
Supported browsers	33
Supported capabilities	34
Supported platforms	35
Limitations	35
API reference	36
Using Amazon VPC with Device Farm desktop browser testing	37
Configuring your project to use Amazon VPC endpoints	38
Removing an Amazon VPC configuration from a Device Farm desktop browser testing project	41

Tagging in Device Farm	42
Tagging resources	42
Looking up resources by tag	43
Removing tags from resources	44
Troubleshooting	45
Can't connect to a RemoteWebDriver	45
Timeouts when setting up RemoteWebDriver	45
Rate limit errors during session creation	45
Session creation rate management	46
Actions per second rate management	46
I get errors during session creation	48
Browser can't connect to my app	48
Selenium 3 WebDriver testing framework is not working properly with Microsoft Edge (Chromium)	49
My tests using XPath are slow	49
Quotas	50
Security	51
Your data in Device Farm desktop browser testing	51
Access control and IAM	51
Using service-linked roles	53
Service-linked role permissions for Device Farm	54
Creating a service-linked role for Device Farm	57
Editing a service-linked role for Device Farm	57
Deleting a service-linked role for Device Farm	57
Supported Regions for Device Farm service-linked roles	58
Document history	60

What is Device Farm desktop browser testing?

This guide shows you how to run your Selenium tests on multiple desktop browsers hosted on AWS. The feature scales seamlessly so you can run your tests in parallel on multiple browser instances to speed up the execution of your test suite. For every browser the test is executed on, Device Farm generates video recordings and Selenium logs to help you quickly identify any issues with your web app.



Topics

- [Before you begin](#)
- [Terminology](#)
- [Accessing Device Farm desktop browser testing](#)
- [Pricing for Device Farm desktop browser testing](#)

Before you begin

We recommend that you read the following sections before you use this feature:

- If you're using Selenium Grid or another browser testing provider, see [Migrating to Device Farm desktop browser testing from Selenium Grid](#) and [Supported capabilities, browsers, and platforms in Device Farm desktop browser testing](#).

- If you're already using Selenium for local testing, see [Migrating to Device Farm desktop browser testing from local Selenium WebDrivers](#).

Device Farm desktop browser testing supports using Amazon VPC to test applications in an isolated environment. For more information, see [Using Amazon VPC with Device Farm desktop browser testing](#).

Terminology

The following terms are used throughout this guide:

Project

A grouping of Selenium sessions in Device Farm. For more information, see [Projects in Device Farm desktop browser testing](#).

Session

A single instance of a browser under your control. For more information, see [Working with Device Farm desktop browser testing sessions](#).

Artifact

A record (recordings, logs, and so on) produced by Device Farm during the execution of your tests. For more information, see [Artifacts in Device Farm desktop browser testing](#).

Action

A record of your test suite interacting with Device Farm using the W3C WebDriver protocol. For more information, see [Actions in Device Farm desktop browser testing](#).

Accessing Device Farm desktop browser testing

You access this feature through the AWS SDK. For more information, see [AWS SDK and Tools](#).

Pricing for Device Farm desktop browser testing

This feature is billed on a per-minute basis. For more information, see [Device Farm Pricing](#).

Getting started with Selenium testing on Device Farm

For most Selenium users, using Device Farm desktop browser testing requires only minor changes to their testing configuration, specifically one API call to create a limited-time use URL for the Selenium `RemoteWebDriver`.

If you're new to Selenium testing, see the [Selenium documentation](#).

This section covers:

- [Migrating to Device Farm desktop browser testing from Selenium Grid](#)
- [Migrating to Device Farm desktop browser testing from local Selenium WebDrivers](#)

Migrating to Device Farm desktop browser testing from Selenium Grid

To migrate Selenium test suites in an environment that uses the `RemoteWebDriver`, you must modify your test suite setup procedure. Using the AWS SDK, you request a signed command executor (hub) URL from the Device Farm desktop browser testing API. Then you pass that URL along with your requested capabilities to your `RemoteWebDriver`.

To create your first project


1. Run your tests locally to observe and confirm your current test suite behavior.
2. To use the desktop browser testing feature, you must install and configure the [AWS SDK](#) for the language appropriate for your tests.
3. Use the console or CLI to create a project:

Console

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
2. In the navigation pane, choose **Desktop Browser Testing**, and then choose **Projects**.
3. If you already have a project, under **Desktop browser testing projects**, choose the name of your project.

Otherwise, to create a new project, choose **New project**. Then, on the **Create Project** page, do the following:

- a. Enter a **Project name**.
 - b. (Optional) Enter a project **Description**.
 - c. (Optional) Under **Virtual Private Cloud (VPC) Settings**, you can configure your project's VPC peering settings by choosing the **VPC**, its **Subnets**, and its **Security Groups**. For instructions on connecting Device Farm to a VPC, see [Working with Amazon Virtual Private Cloud across Regions](#) in the *Device Farm Developer Guide*.
 - d. Choose **Create**.
4. In the project details, note the project's Amazon Resource Name (ARN). It looks like this: `arn:aws:devicefarm:us-west-2:111122223333:testgrid-project:123e4567-e89b-12d3-a456-426655440000`.

 **Note**

For instructions on updating your project configuration, see [Configuring your project to use Amazon VPC endpoints](#).

CLI

The following creates a project:


```
aws devicefarm create-test-grid-project --name "Peculiar Things"
```

 **Note**

To update your project configuration, see [Configuring your project to use Amazon VPC endpoints](#).

4. Modify your RemoteWebDriver initialization to use the WebDriver endpoint.

Java

 **Note**

This example uses JUnit 5 and the AWS SDK for Java 2.x. For more information about the AWS SDK for Java 2.x, see [AWS SDK for Java 2.x API Reference](#). If you are

using a different test framework, be aware that `@Before` and `@After` are called before and after each test, respectively.

```
// Import the AWS SDK for Java 2.x Device Farm client:
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.devicefarm.*;
import software.amazon.awssdk.services.devicefarm.model.*;
import java.net.URL;

// in your tests ...
public class MyTests {
    // ... When you set up your test suite
    private static RemoteWebDriver driver;

    @Before
    void setUp() {
        String myProjectARN = "arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655440000";
        DeviceFarmClient client =
DeviceFarmClient.builder().region(Region.US_WEST_2).build();
        CreateTestGridUrlRequest request = CreateTestGridUrlRequest.builder()
            .expiresInSeconds(300)
            .projectArn(myProjectARN)
            .build();
        CreateTestGridUrlResponse response = client.createTestGridUrl(request);
        URL testGridUrl = new URL(response.url());
        // You can now pass this URL into RemoteWebDriver.
        WebDriver driver = new RemoteWebDriver(testGridUrl,
DesiredCapabilities.firefox());
    }

    @After
    void tearDown() {
        // make sure to close your WebDriver:
        driver.quit();
    }
}
```

For more information, see [Using Device Farm desktop browser testing in Java](#).

Python

Important

This example is written with the assumption you are using Python 3 with pytest. If you are using another testing framework, see the documentation.

```
# Include boto3, the Python SDK's main package:
import boto3, pytest

# in your tests:
# Set up the Device Farm client, get a driver URL:
class myTestSuite:
    def setup_method(self, method):
        devicefarm_client = boto3.client("devicefarm", region_name="us-west-2")
        testgrid_url_response = devicefarm_client.create_test_grid_url(
            projectArn="arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655440000",
            expiresInSeconds=300)
        self.driver =
selenium.webdriver.Remote(testgrid_url_response["url"], selenium.webdriver.DesiredCapabilities)

# later, make sure to end your WebDriver session:
def teardown_method(self, method):
    self.driver.quit()
```

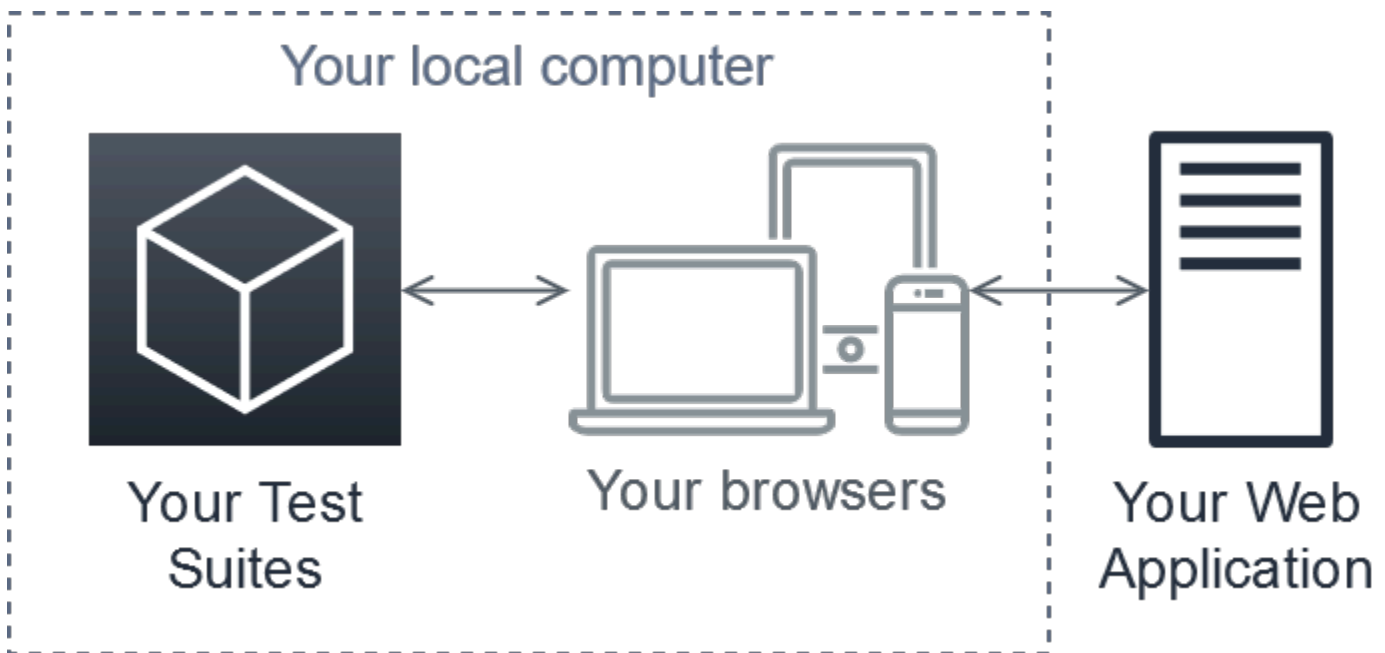
For more information, see [Using Device Farm desktop browser testing in Python](#).

5. Make sure that the environment variables `AWS_ACCESS_KEY` and `AWS_SECRET_KEY` are configured in your testing environment.
6. Run your tests.

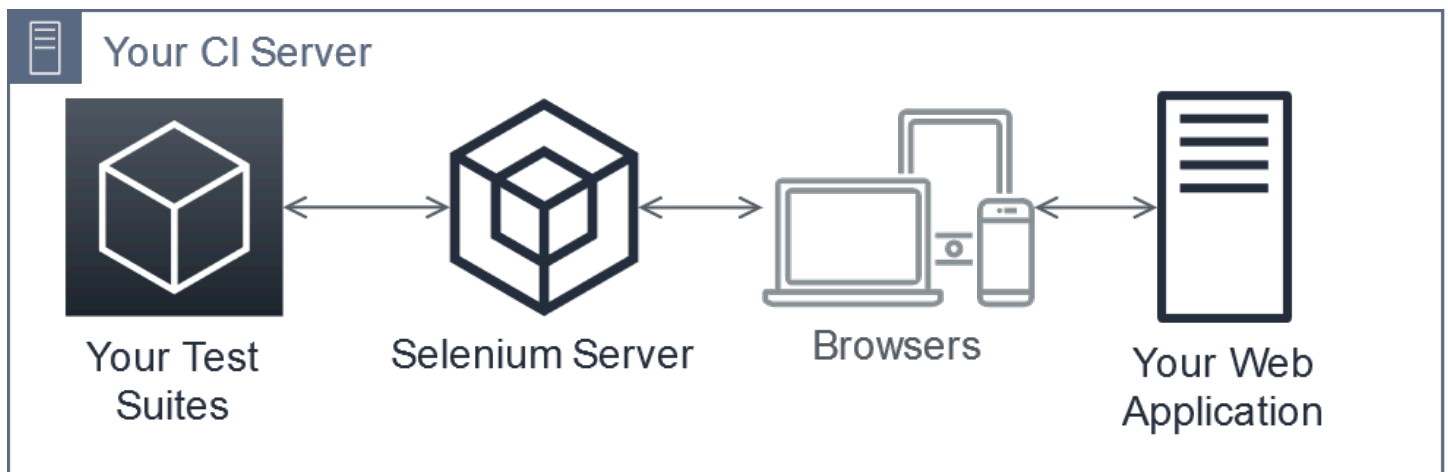
Migrating to Device Farm desktop browser testing from local Selenium WebDrivers

If you're testing browsers locally, you can use the desktop browser testing feature to test on Firefox, Edge, and Chrome without setting up those browsers.

Traditional local testing with Selenium involves tests that start a WebDriver component, such as `GeckoDriver` or `ChromeDriver`. These components directly interact with a browser under test, without the use of an intermediary. This means that your browsers are running where your tests are running:



A common solution is to add an intermediary, Selenium Server, that runs browsers remotely. Often, this results in your tests being run on your CI server with a headless browser. Your infrastructure now looks like this:



When you use Selenium Server (or Selenium Grid), you create a `RemoteWebDriver` instance that acts as a stand-in for your browser-specific `WebDriver`.

This section shows you how to configure your local WebDriver tests to use Selenium's RemoteWebDriver with a URL from the GetTestGridUrl API call.

To perform these steps, you need an AWS account and a working set of tests.

Important

We recommend that you follow the standard security advice of granting least privilege—that is, granting only the permissions required to perform a task—when you configure the AWS SDK and AWS CLI with credentials. For more information, see [AWS Security Credentials](#) and [IAM Best Practices](#).

1. To keep track of your sessions, you must create a project.

Console

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
2. In the navigation pane, choose **Desktop Browser Testing**, and then choose **Projects**.
3. If you already have a project, under **Desktop browser testing projects**, choose the name of your project.

Otherwise, to create a new project, choose **New project**. Then, on the **Create Project** page, do the following:

- a. Enter a **Project name**.
 - b. (Optional) Enter a project **Description**.
 - c. (Optional) Under **Virtual Private Cloud (VPC) Settings**, you can configure your project's VPC peering settings by choosing the **VPC**, its **Subnets**, and its **Security Groups**. For instructions on connecting Device Farm to a VPC, see [Working with Amazon Virtual Private Cloud across Regions](#) in the *Device Farm Developer Guide*.
 - d. Choose **Create**.
4. In the project details, note the project's Amazon Resource Name (ARN). It looks like this: `arn:aws:devicefarm:us-west-2:111122223333:testgrid-project:123e4567-e89b-12d3-a456-426655440000`.

Note

For instructions on updating your project configuration, see [Configuring your project to use Amazon VPC endpoints](#).

AWS CLI

Use the `create-test-grid-project` command to create a project:

```
aws devicefarm create-test-grid-project --name "Peculiar Things"
```

Make a note of the project ARN, which is used by your application:

```
{
  "testGridProject": {
    "arn": "arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655440000",
    "name": "Peculiar Things"
  }
}
```

Note

To update your project configuration, see [Configuring your project to use Amazon VPC endpoints](#).

2. To use the desktop browser testing feature, you must install and configure the [AWS SDK](#) for the language appropriate for your tests.
3. Modify your environment to include your AWS access and secret keys. The steps vary depending on your configuration, but involve setting two environment variables:

Important

We recommend that you follow the standard security advice of granting least privilege—that is, granting only the permissions required to perform a task—when you

configure the AWS SDK and AWS CLI with credentials. For more information, see [AWS Security Credentials](#) and [IAM Best Practices](#).

```
AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY
```

4. Instead of creating a `WebDriver` for your browser (for example, `GeckoDriver`) create a `RemoteWebDriver` and get a URL from the desktop browser testing feature.

Java

 **Note**

This example uses JUnit 5 and the AWS SDK for Java 2.x. For more information about the AWS SDK for Java 2.x, see [AWS SDK for Java 2.x API Reference](#). If you are using a different test framework, be aware that `@Before` and `@After` are called before and after each test, respectively.

```
// Import the AWS SDK for Java 2.x Device Farm client:
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.devicefarm.*;
import software.amazon.awssdk.services.devicefarm.model.*;
import java.net.URL;

// in your tests ...
public class MyTests {
    // ... When you set up your test suite
    private static RemoteWebDriver driver;

    @Before
    void setUp() {
        String myProjectARN = "arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655440000";
        DeviceFarmClient client =
        DeviceFarmClient.builder().region(Region.US_WEST_2).build();
        CreateTestGridUrlRequest request = CreateTestGridUrlRequest.builder()
            .expiresInSeconds(300)
```

```
        .projectArn(myProjectARN)
        .build();
    CreateTestGridUrlResponse response = client.createTestGridUrl(request);
    URL testGridUrl = new URL(response.url());
    // You can now pass this URL into RemoteWebDriver.
    WebDriver driver = new RemoteWebDriver(testGridUrl,
DesiredCapabilities.firefox());
}

@After
void tearDown() {
    // make sure to close your WebDriver:
    driver.quit();
}
}
```

For more information, see [Using Device Farm desktop browser testing in Java](#).

Python

Important

This example is written with the assumption you are using Python 3 with pytest. If you are using another testing framework, see the documentation.

```
# Include boto3, the Python SDK's main package:
import boto3, pytest

# in your tests:
# Set up the Device Farm client, get a driver URL:
class myTestSuite:
    def setup_method(self, method):
        devicefarm_client = boto3.client("devicefarm", region_name="us-west-2")
        testgrid_url_response = devicefarm_client.create_test_grid_url(
            projectArn="arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655440000",
            expiresInSeconds=300)
        self.driver =
selenium.webdriver.Remote(testgrid_url_response["url"], selenium.webdriver.DesiredCapab
```

```
# later, make sure to end your WebDriver session:  
def teardown_method(self, method):  
    self.driver.quit()
```

For more information, see [Using Device Farm desktop browser testing in Python](#).

For a list of supported capabilities, see [Supported capabilities, browsers, and platforms in Device Farm desktop browser testing](#)

5. Run your tests as you would normally.

Migrating Selenium tests in testing frameworks to Device Farm desktop browser testing

This section describes how to migrate existing Selenium test suites in testing frameworks to use AWS Device Farm desktop browser testing.

Topics

- [Using Device Farm desktop browser testing in Java](#)
- [Using Device Farm desktop browser testing in Python](#)
- [Using Device Farm desktop browser testing with Node.js](#)
- [Using Device Farm desktop browser testing in Ruby](#)

Using Device Farm desktop browser testing in Java

Follow the steps in this topic to migrate your Java test suite to using the desktop browser testing feature with `RemoteWebDriver`.

Important

This topic is written with the assumption you are using the AWS SDK for Java 2.x and JUnit 5. If you are using the 1.x release of the SDK for Java, see the [AWS SDK for Java API Reference](#). For more information about the SDK for Java 2.x, see the [AWS SDK for Java 2.x API Reference](#). If you are using another testing framework (for example, TestNG), `@BeforeAll` and `@AfterAll` annotated methods are run at the start and end, respectively, of a test class.

To migrate your existing tests

1. Add the AWS SDK for Java 2.x to your Maven `pom.xml`:

```
<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>aws-sdk-java</artifactId>
  <version>2.10.60</version>
```

```
</dependency>
```

If you're using another build manager such as Gradle, see [AWS SDK for Java 2.x Developer Guide](#) and [AWS SDK for Java 2.x on MVNRepository](#).

2. Wherever you initialize a `WebDriver` instance, configure a `RemoteWebDriver` instance using the endpoint generated by the Device Farm API.
 - a. Import the Device Farm classes:

```
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.devicefarm.DeviceFarmClient;
import
    software.amazon.awssdk.services.devicefarm.model.CreateTestGridUrlRequest;
import
    software.amazon.awssdk.services.devicefarm.model.CreateTestGridUrlResponse;
```

- b. Instantiate a new `DeviceFarmClient` where you create your `WebDriver`:

```
public class myTestSuite {
    /** Set up the test suite. */
    @BeforeAll
    public void setUp() {
        DeviceFarmClient client =
            DeviceFarmClient.builder().region(Region.US_WEST_2).build();
    }
}
```

- c. Create a request for a signed `WebDriver` hub URL:

```
CreateTestGridUrlRequest request = CreateTestGridUrlRequest.builder()
    .expiresInSeconds(300) // 5 minutes
    .projectArn("arn:aws:devicefarm:us-
west-2:111122223333:testgrid-project:1111111-2222-3333-4444-5555555555")
    .build();
```

- d. Get a response:

```
URL testGridUrl = null;
try {
    CreateTestGridUrlResponse response =
        client.createTestGridUrl(request);
    testGridUrl = new URL(response.url());
} catch (Exception e) {
    e.printStackTrace();
}
```

```
}
    Assertions.assertNotNull(testGridUrl);
```

- e. Create a `DesiredCapabilities` object to hold the capabilities you want or use a preconfigured set:

```
DesiredCapabilities desired_capabilities = new DesiredCapabilities();
desired_capabilities.setCapability("browserName", "firefox");
desired_capabilities.setCapability("browserVersion", "latest");
desired_capabilities.setCapability("platform", "windows");
// Or
DesiredCapabilities desired_capabilities =
DesiredCapabilities.firefox();
```

- f. Use `RemoteWebDriver` in place of your existing `WebDriver` implementation.

```
driver = new RemoteWebDriver(testGridUrl, desired_capabilities);
```

- g. Make sure to close your session after the tests complete:

```
@AfterAll
public static void teardown() {
    driver.quit();    // tear down the driver and its parts.
}
```

3. Modify your environment to include your AWS access and secret keys. The steps vary depending on your configuration, but involve setting two environment variables:

Important

We recommend that you follow the standard security advice of granting least privilege—that is, granting only the permissions required to perform a task—when you configure the AWS SDK and AWS CLI with credentials. For more information, see [AWS Security Credentials](#) and [IAM Best Practices](#).

```
AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY
```

4. Run your tests.

Using Device Farm desktop browser testing in Python

Follow the steps in this topic to get your tests running on Python.

Important

This topic is written with the assumption you are using Python 3, Boto3, and pytest. As of January 1, 2020, Python 2 is officially no longer supported. For more information, see [Sunsetting Python 2](#) from the Python Software Foundation.

To migrate your existing tests

1. Add the AWS SDK for Python (Boto3) to your requirements. If you're using pipenv for your requirements management, run the following:

```
pipenv install boto3
```

If you're using pip, add the following to your `requirements.txt`:

```
boto3 >= 1.10.44
```

Make sure that the version listed here is correct and then run `pip install -r requirements.txt` to install the Boto3.

2. Modify your test suite to use `RemoteWebDriver`. Wherever you initialize a `WebDriver` instance, configure a `RemoteWebDriver` instance using the endpoint generated by the Device Farm API.

- a. Import the Device Farm classes:

```
import boto3
from selenium.webdriver import DesiredCapabilities
from selenium.webdriver import Remote
```

- b. Instantiate a new `DeviceFarmClient` where you create your `WebDriver`

```
class MyPytestTests():
    def setup_method(self, method):
        # step 2: Set up a client for boto3
```

```
# The AWS_ACCESS_KEY and AWS_SECRET_KEY will be inferred from the command
line.
devicefarm_client = boto3.client("devicefarm")
```

c. Get a signed WebDriver hub URL:

```
testgrid_url_response = devicefarm_client.create_test_grid_url(
    projectArn= "arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655440000",
    expiresInSeconds=300
)
```

d. Use the `DesiredCapabilities` class to specify the browser to test against:

```
desired_capabilities = DesiredCapabilities.FIREFOX
desired_capabilities["platform"] = "windows"
```

e. Create your `RemoteWebDriver` in place of your existing `GeckoDriver`, `ChromeDriver`, or similar.

```
self.driver = Remote(testgrid_url_response['url'], desired_capabilities)
```

f. Make sure to close your session after the tests are complete:

```
def teardown_method(self, method):
    self.driver.quit()
```

3. Modify your environment to include your AWS access and secret keys. The steps vary depending on your configuration, but involve setting two environment variables:

⚠ Important

We recommend that you follow the standard security advice of granting least privilege—that is, granting only the permissions required to perform a task—when you configure the AWS SDK and AWS CLI with credentials. For more information, see [AWS Security Credentials](#) and [IAM Best Practices](#).

```
AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY
```

4. Run your tests, including the following:

```
pytest -s
```

For more information, see [SDK for Python \(Boto3\) API Reference](#).

Using Device Farm desktop browser testing with Node.js

Device Farm desktop browser testing can be used alongside the AWS SDK for JavaScript in Node.js and W3C WebDriver Specification compatible libraries, such as the official Selenium WebDriver package and Webdriver.IO.

Topics

- [Using the official Selenium WebDriver Node.js package](#)
- [Using Webdriver.IO](#)

Using the official Selenium WebDriver Node.js package

You can use Device Farm desktop browser testing with the Selenium Node.js `selenium-webdriver` package. The following example shows how to use the `selenium-webdriver` npm package to interact with Device Farm desktop browser testing.

Note

This example assumes you have the SDK for JavaScript in Node.js configured and installed. For more information, see [AWS SDK for JavaScript in Node.js Getting Started Guide](#).

```
var webdriver = require('selenium-webdriver');
var AWS = require("aws-sdk");

var PROJECT_ARN = "arn:aws:devicefarm:us-west-2:111122223333:testgrid-project:123e4567-
e89b-12d3-a456-426655440000";
var devicefarm = new AWS.DeviceFarm({ region: "us-west-2" });

(async () => {
```

```
// Get the endpoint to create a new session
const testGridUrlResult = await devicefarm.createTestGridUrl({
  projectArn: PROJECT_ARN,
  expiresInSeconds: 600
}).promise();

console.log("Created url result:", testGridUrlResult);
runExample(testGridUrlResult.url);

})();catch((e) => console.error(e));

var runExample = async (urlString) => {
  console.log("Starting WebDriverJS remote driver")
  driver = await new webdriver.Builder()
    .usingServer(urlString)
    .withCapabilities({ browserName: 'chrome' })
    .build();

  console.log("New session created:", driver.getSession());

  await driver.get('https://aws.amazon.com/');
  const title = await driver.getTitle();
  console.log('Title was: ' + title);

  console.log("Deleting session...");
  await driver.quit();
}
```

Using Webdriver.IO

Device Farm desktop browser testing is supported in Webdriver.IO with some slight configuration changes. The following two examples show how to use Device Farm desktop browser testing with and without the Mocha test framework.

Using Webdriver.IO and Device Farm desktop browser testing without Mocha

Using Device Farm desktop browser testing with Webdriver.IO does not require the inherent use of the Mocha test suite. The following example demonstrates how to configure the Webdriver.IO API for use with Device Farm desktop browser testing and interact with a browser session.

Note

This example assumes you have the SDK for JavaScript in Node.js configured and installed. For more information, see [AWS SDK for JavaScript in Node.js Getting Started Guide](#).

```
const webdriverio = require('webdriverio');
const AWS = require("aws-sdk");

var PROJECT_ARN = process.env.TEST_GRID_PROJECT
var devicefarm = new AWS.DeviceFarm({ region: "us-west-2" });

(async () => {
  // Get a new signed WebDriver hub URL.
  const testGridUrlResult = await devicefarm.createTestGridUrl({
    projectArn: PROJECT_ARN,
    expiresInSeconds: 600
  }).promise();

  console.log("Created url result:", testGridUrlResult);
  runExample(testGridUrlResult.url);

})();

var runExample = async (urlString) => {
  var url = new URL(urlString)

  console.log("Starting a new remote driver")
  const browser = await webdriverio.remote({
    logLevel: 'trace',
    hostname: url.host,
    path: url.pathname,
    protocol: 'https',
    port: 443,
    connectionRetryTimeout: 180000,
    capabilities: {
      browserName: 'chrome',
    }
  })

  await browser.url('https://aws.amazon.com/')
}
```



```
const title = await browser.getTitle()
console.log('Title was: ' + title)

await browser.deleteSession()
}
```

Integrating Device Farm desktop browser testing into Mocha and Webdriver.IO test suites

To integrate Device Farm into your existing Mocha and Webdriver.IO test suite, you'll need to add support for the Device Farm desktop browser testing service backend. The following steps will guide you through the process:

1. Install the SDK for JavaScript in Node.js. For more information, see [AWS SDK for JavaScript in Node.js Getting Started Guide](#)

```
$ npm install aws-sdk
```

2. Modify your environment to include your AWS access and secret keys. The steps vary depending on your configuration, but involve setting two environment variables:

Important

We recommend that you follow the standard security advice of granting least privilege—that is, granting only the permissions required to perform a task—when you configure the AWS SDK and AWS CLI with credentials. For more information, see [AWS Security Credentials](#) and [IAM Best Practices](#).

```
AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE
AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxrFiCYEXAMPLEKEY
```

3. Add `services/devicefarm.service.js` with the following:

```
const AWS = require("aws-sdk")

class AwsDeviceFarmService {

  constructor() {
```

```
    this.projectArn = process.env.TEST_GRID_PROJECT
    this.devicefarm = new AWS.DeviceFarm({ region: "us-west-2" })
  }

  // Get a test grid URL once for all test suites / test cases during this wdio
  testrunner invocation
  onPrepare(config, capabilities) {
    return new Promise((resolve, reject) => this.devicefarm.createTestGridUrl({
      projectArn: this.projectArn,
      expiresInSeconds: 300
    }), (err, createTestGridUrlResult) => {
      if (err) {
        return reject(err)
      }

      console.log("CreateTestGridUrlResult:", createTestGridUrlResult)
      process.env.AWS_DEVICE_FARM_SERVICE_TEST_GRID_URL =
createTestGridUrlResult.url

      resolve()
    })
  }

  // Set the connection settings on the beforeSession hook. See more: https://github.com/webdriverio/webdriverio/issues/5113
  beforeSession(config, capabilities, specs) {
    return new Promise((resolve, reject) => {

      // Set the default connection timeout to at least 3 minutes
      config.connectionRetryTimeout = 180000 // millis

      const testGridUrl = new
URL(process.env.AWS_DEVICE_FARM_SERVICE_TEST_GRID_URL)

      config.protocol = 'https'
      config.hostname = testGridUrl.host
      config.path = testGridUrl.pathname
      config.port = 443

      resolve()
    })
  }
}
```

```
module.exports = AwsDeviceFarmService
```

4. Set the environment variable TEST_GRID_PROJECT to the ARN of your project.
5. Modify your wdio.conf.js

```
const AwsDeviceFarmService = require('./service/devicefarm.service')

exports.config = {
  runner: 'local',

  services: [
    // Use the devicefarm.service.js to automatically generate a remote URL
    // that will be used to create all web drivers from this testrunner
    [AwsDeviceFarmService]
  ],

  // Define which test specs should run.
  specs: [
    './test/specs/**/*.js'
  ],

  capabilities: [{
    browserName: 'firefox',
  }],

  // Level of logging verbosity: trace | debug | info | warn | error | silent
  logLevel: 'info',
}
```

6. Your tests will now run using Device Farm desktop browser testing!

Using Device Farm desktop browser testing in Ruby

Follow the steps in this topic to get your tests running on ruby.

To migrate your existing tests

1. Install the appropriate gems:

```
gem install selenium-webdriver
```

```
gem install aws-sdk-devicefarm
```

If you are using Bundler, add the following to your Gemfile:

```
gem 'selenium-webdriver'  
gem 'aws-sdk-devicefarm'
```

and update your installed gems with

```
bundle install
```

2. Modify your test suite to use RemoteWebDriver. Wherever you initialize a WebDriver instance, configure a RemoteWebDriver instance using the endpoint generated by the Device Farm API.

a. Import the Device Farm classes:

```
require 'aws-sdk-devicefarm'  
require 'selenium-webdriver'
```

b. Instantiate a new DeviceFarm::Client where you create your WebDriver

```
devicefarm = Aws::DeviceFarm::Client.new(region: 'us-west-2')
```

c. Get a signed WebDriver hub URL:

```
test_grid_url_response = devicefarm.create_test_grid_url(  
  project_arn: "arn:aws:devicefarm:us-west-2:111122223333:testgrid-  
project:123e4567-e89b-12d3-a456-426655440000",  
  expires_in_seconds: 300)  
remote_url = test_grid_url_response.url
```

d. Use the DesiredCapabilities class to specify the browser to test against:

```
capabilities = Selenium::WebDriver::Remote::Capabilities.firefox
```

e. Create your RemoteWebDriver in place of your existing GeckoDriver, ChromeDriver, or similar. To avoid timeouts, create a new HTTP client with an extended timeout:

```
client = Selenium::WebDriver::Remote::Http::Default.new  
client.read_timeout = 180 # seconds - make sure to set a timeout here of at  
least 3 minutes (default is 60 seconds)
```

```
driver = Selenium::WebDriver.for :remote, http_client: client, url: remote_url,  
  desired_capabilities: capabilities
```

Note

If you do not set an increased timeout, the session creation will time out before your session is ready.

- f. Make sure to close your session after the tests are complete:

```
driver.quit
```

3. Modify your environment to include your AWS access and secret keys. The steps vary depending on your configuration, but involve setting two environment variables:

Important

We recommend that you follow the standard security advice of granting least privilege—that is, granting only the permissions required to perform a task—when you configure the AWS SDK and AWS CLI with credentials. For more information, see [AWS Security Credentials](#) and [IAM Best Practices](#).

```
AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE  
AWS_SECRET_ACCESS_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

4. Run your tests. They're on Device Farm!

For more information, see [AWS SDK for Ruby Developer Guide](#).

Working with Device Farm desktop browser testing

The desktop browser testing feature is an interface between your testing framework and an on-demand pool of highly available combinations of browsers and operating systems.

This section covers:

- [Projects in Device Farm desktop browser testing](#)
- [Working with Device Farm desktop browser testing sessions](#)
- [Actions in Device Farm desktop browser testing](#)
- [Artifacts in Device Farm desktop browser testing](#)

Projects in Device Farm desktop browser testing

Selenium projects created, organized, and managed separately from native testing projects in Device Farm.

You can use the console, AWS CLI, and AWS SDK to manage projects.

Important

We recommend that you follow the standard security advice of granting least privilege—that is, granting only the permissions required to perform a task—when you configure the AWS SDK and AWS CLI with credentials. For more information, see [AWS Security Credentials](#) and [IAM Best Practices](#).

Creating a Device Farm desktop browser testing project

You can use the Device Farm console or the AWS CLI to create a project.

Device Farm console

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
2. In the navigation pane, choose **Desktop Browser Testing**, and then choose **Projects**.
3. If you already have a project, under **Desktop browser testing projects**, choose the name of your project.

Otherwise, to create a new project, choose **New project**. Then, on the **Create Project** page, do the following:

- a. Enter a **Project name**.
 - b. (Optional) Enter a project **Description**.
 - c. (Optional) Under **Virtual Private Cloud (VPC) Settings**, you can configure your project's VPC peering settings by choosing the **VPC**, its **Subnets**, and its **Security Groups**. For instructions on connecting Device Farm to a VPC, see [Working with Amazon Virtual Private Cloud across Regions](#) in the *Device Farm Developer Guide*.
 - d. Choose **Create**.
4. In the project details, note the project's Amazon Resource Name (ARN). It looks like this:
`arn:aws:devicefarm:us-west-2:111122223333:testgrid-project:123e4567-e89b-12d3-a456-426655440000`.

 **Note**

For instructions on updating your project configuration, see [Configuring your project to use Amazon VPC endpoints](#).

AWS CLI

Use the `create-test-grid-project` command to create a project:

```
aws devicefarm create-test-grid-project --name "My Web App"
```

The result contains the project that you just created:

```
{
  "testGridProject": {
    "arn": "arn:aws:devicefarm:us-west-2:111122223333:testgrid-project:123e4567-e89b-12d3-a456-426655440000",
    "name": "My Web App"
  }
}
```

Note

To update your project configuration, see [Configuring your project to use Amazon VPC endpoints](#).

Deleting a Device Farm desktop browser testing project

To delete a project using the AWS CLI, you need the project ARN.

Warning

Deleting a project is not reversible.
You cannot delete a project that has in-progress sessions.

Device Farm Console

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
2. On the Device Farm navigation panel, choose **Desktop Browser Testing**, then choose **Projects**.
3. Choose the project you want to delete.
4. Choose the **Delete** action.
5. Confirm that you want to delete the project and any associated content, then choose **Delete**.

AWS CLI

1. If you don't know the ARN, use the `list-test-grid-projects` command to list your current Device Farm Selenium projects:

```
aws devicefarm list-test-grid-projects
```

The result contains your current Selenium testing projects:

```
{  
  "testGridProjects": [  

```



```
{
  "arn": "arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655440000",
  "name": "Peculiar Things"
},
{
  "arn": "arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655441111",
  "name": "Wumbo's Wild WiFi Emporium"
},
{
  "arn": "arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655442222",
  "name": "Banana Stand 2: This is Bananas"
}
]
```

2. Use the `delete-test-grid-project` command to delete the project and any sessions and artifacts associated with the project:

```
aws devicefarm delete-test-grid-project --project-arn arn:aws:devicefarm:us-
west-2:111122223333:testgrid-project:123e4567-e89b-12d3-a456-426655440000
```

Working with Device Farm desktop browser testing sessions

Sessions are used to keep track of two things:

- Artifacts, which are records of the results of events. For more information, see [the section called "Artifacts"](#).
- Actions, which are the series of events that created artifacts. For more information, see [the section called "Actions"](#).

You create a session when you start a `RemoteWebDriver`. Sessions are also the base unit of billing. Billing is determined by the time between your first `WebDriver` action (for example, getting a page) and the closing or timing out of the session, whichever comes first.

Looking up sessions

You can use the `getTestGridSession` API action or `get-test-grid-session` CLI command to look up a session. This action can take two sets of parameters: a session ARN or a project ARN and a session ID. If you do not know your session ARN or session ID, you can use the `listTestGridSessions` action or `list-test-grid-sessions` CLI command to search for sessions. This command can search by session start, end, or status.

The following examples show how you can use the command to find sessions.

Look up sessions by their creation time and status

The following example shows how to look up active sessions created after a specified time:

```
aws devicefarm list-test-grid-sessions \
  --project-arn "arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655440000" \
  --creation-time-after "2019-12-04T19:21:13" \
  --status "ACTIVE"
```

Look up sessions by the time they ended

The following example shows how to look up sessions that ended before a specified date and time:

```
aws devicefarm list-test-grid-sessions \
  --project-arn "arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655440000" \
  --end-time-before "2020-04-21T00:20:00" ; The state, "CLOSED", can be assumed
here.
```

Look up a session by ARN

The following example shows how to look up a session by its ARN:

```
aws devicefarm get-test-grid-session --session-arn arn:aws:devicefarm:us-
west-2:111122223333:testgrid-session:123e4567-e89b-12d3-a456-426655440000/123e4567-
e89b-12d3-a456-426655441111
```

Look up a session by project ARN and session ID

The following example shows how to look up a session by its project ARN and session ID:

```
aws devicefarm get-test-grid-session --project-arn arn:aws:devicefarm:us-west-2:111122223333:testgrid-project:123e4567-e89b-12d3-a456-426655440000 --session-id 123e4567-e89b-12d3-a456-426655441111
```

Actions in Device Farm desktop browser testing

When you interact with the W3C WebDriver protocol, Device Farm logs the calls made as actions.

An action represents the WebDriver endpoint requested by your test's RemoteWebDriver.

Actions consist of:

- The HTTP requestType (method) used for the call.
- The HTTP statusCode returned by the WebDriver API.
- The time that the call was made.
- The number, in milliseconds, it took to make the call.

Artifacts in Device Farm desktop browser testing

Artifacts are created automatically in Device Farm when your tests perform actions like opening or closing a session.

The following kinds of artifacts are produced:

- Video recording of your test. Recording starts when your session starts and ends when your session times out or is closed.

Note

For long-running test sessions, videos are split at even intervals. Even test runs that go just over one of these intervals are still split.

- Selenium logs produced by the Selenium WebDriver.

Session artifacts have the following properties:

- Type (video, log, other).

- File name.
- Limited lifetime S3-signed URL that can be used to retrieve the artifact.

A new URL is generated whenever the artifact is requested as a part of an API request. This URL expires after an hour.

Technical reference

In the W3C WebDriver model, the feature operates as an intermediary node. It provides a means to start, end, and manage sessions.

Here is the functional flow for using the feature:

1. Use the `createTestGridProject` API to create a project.
2. Use the `createTestGridUrl` API to create a signed WebDriver hub URL.
3. Pass the WebDriver URL to your Selenium `RemoteWebDriver` configuration.
4. Run your tests.
5. Use the `listTestGridSessions` API to retrieve the sessions created in the running of your tests.
6. Use the `listTestGridSessionArtifacts` API to collect any artifacts such as Selenium logs or video.

Supported capabilities, browsers, and platforms in Device Farm desktop browser testing

This topic lists the supported configurations of browsers and operating systems. An exception is raised if you request a browser and operating system combination that is not supported.

Values here are suitable for parametrized test suites.

Supported browsers

The following browsers are supported:

Name	browserName value	Supported platforms
Google Chrome	chrome	windows
Mozilla Firefox	firefox	windows
Microsoft Edge (Chromium)	MicrosoftEdge	windows

Note

AWS Device Farm no longer supports Microsoft Internet Explorer because Microsoft has stopped providing security updates and technical support for older versions of Internet Explorer. For more information, see [Microsoft's announcement of IE end of support](#).

Supported capabilities

The desktop browser testing feature does not implement all capabilities outlined in the [W3C WebDriver documentation](#). Extension capabilities used by Device Farm desktop browser testing are prefixed with `aws:`. The following capabilities are used by the Device Farm WebDriver node:

`browserName` (string, required, no default)

Name of the browser to use for the session. Must be one of the [supported browsers](#). *This value is always lowercase.*

`browserVersion` (string, required)

Version of the browser to launch; one of `latest`, `latest-1`, `latest-2`

Note

Device Farm desktop browser testing does not support requesting specific releases of browsers.

`platform` (string, optional, defaults to `windows`)

Platform (operating system) to use when creating the session. Must be one of the [supported platforms](#).

`aws:maxDurationSecs` (integer, optional, defaults to `900`)

Maximum duration of the session before it is forcibly closed, in seconds. Range: `180` to `2400`.

`aws:idleTimeoutSecs` (integer, optional, defaults to `120`)

Maximum delay between WebDriver commands before the session is forcibly closed. Range: `30` to `900`

The following are not allowed:

w3c (in Chrome)

Setting this capability to `false` will result in an error.

binary

Setting this capability will result in an error.

args

The following browser command line parameters are not allowed:

- `--headless` (Chrome)
- `-headless` (Firefox)
- `--user-data-dir`
- `--no-sandbox` (Chrome and Firefox)

All other capabilities are passed directly to the target browser-specific WebDriver. These arguments are not officially supported by Device Farm. Their use might result in unexpected behavior.

Supported platforms

The following platforms are supported:

Name	platform value
Microsoft Windows	windows

Limitations of Device Farm desktop browser testing

Keep these limitations in mind when you use the desktop browser testing feature:

- The feature is only available in the `us-west-2` (Oregon) region.
- Not all Selenium interfaces are supported. The `pytest-selenium` package, for example, does not allow a command execution URL to be specified.
- Each session you create is isolated from other sessions. Testing that involves multi-window or multi-session interaction is not supported.

Device Farm desktop browser testing API

The [Device Farm API Reference](#) includes a `CreateProject` action and a `CreateTestGridProject` action. For desktop browser and Selenium testing, the following API calls are used:

Project Management APIs

[CreateTestGridProject](#)

Creates a desktop browser testing project.

[DeleteTestGridProject](#)

Deletes a desktop browser testing project.

[GetTestGridProject](#)

Gets information about a desktop browser testing project.

[UpdateTestGridProject](#)

Updates attributes (name, description) of a desktop browser testing project.

[ListTestGridProjects](#)

Lists desktop browser testing projects, including ARNs, names, and descriptions.

Session Management APIs

[CreateTestGridUrl](#)

Creates a limited-time desktop browser testing WebDriver path for creating sessions.

[ListTestGridSessions](#)

Lists your desktop browser testing sessions.

[GetTestGridSession](#)

Gets a desktop browser testing session.

[ListTestGridSessionActions](#)

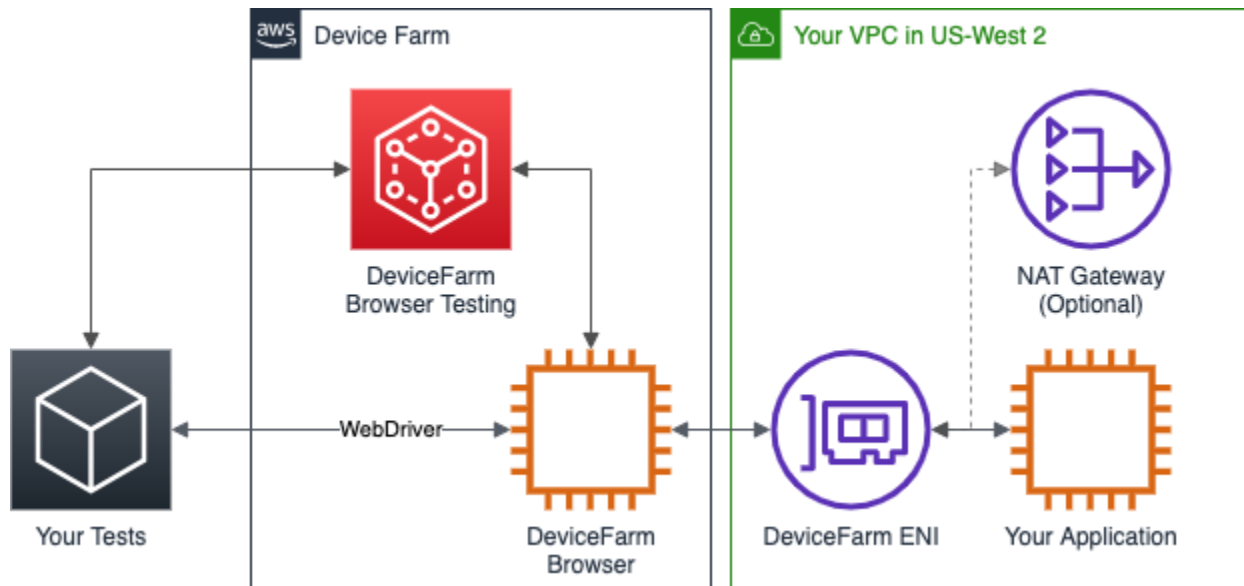
Gets a list of the actions performed during a session.

[ListTestGridSessionArtifacts](#)

Lists the artifacts (video, Selenium logs, and so on) associated with a session.

Using Amazon VPC with Device Farm desktop browser testing

You can give Device Farm desktop browser testing access to an Amazon Virtual Private Cloud (Amazon VPC) environment, enabling testing of isolated, non-internet-facing services and apps through an [elastic network interface](#). For more information on VPCs, see the [Amazon VPC User Guide](#).

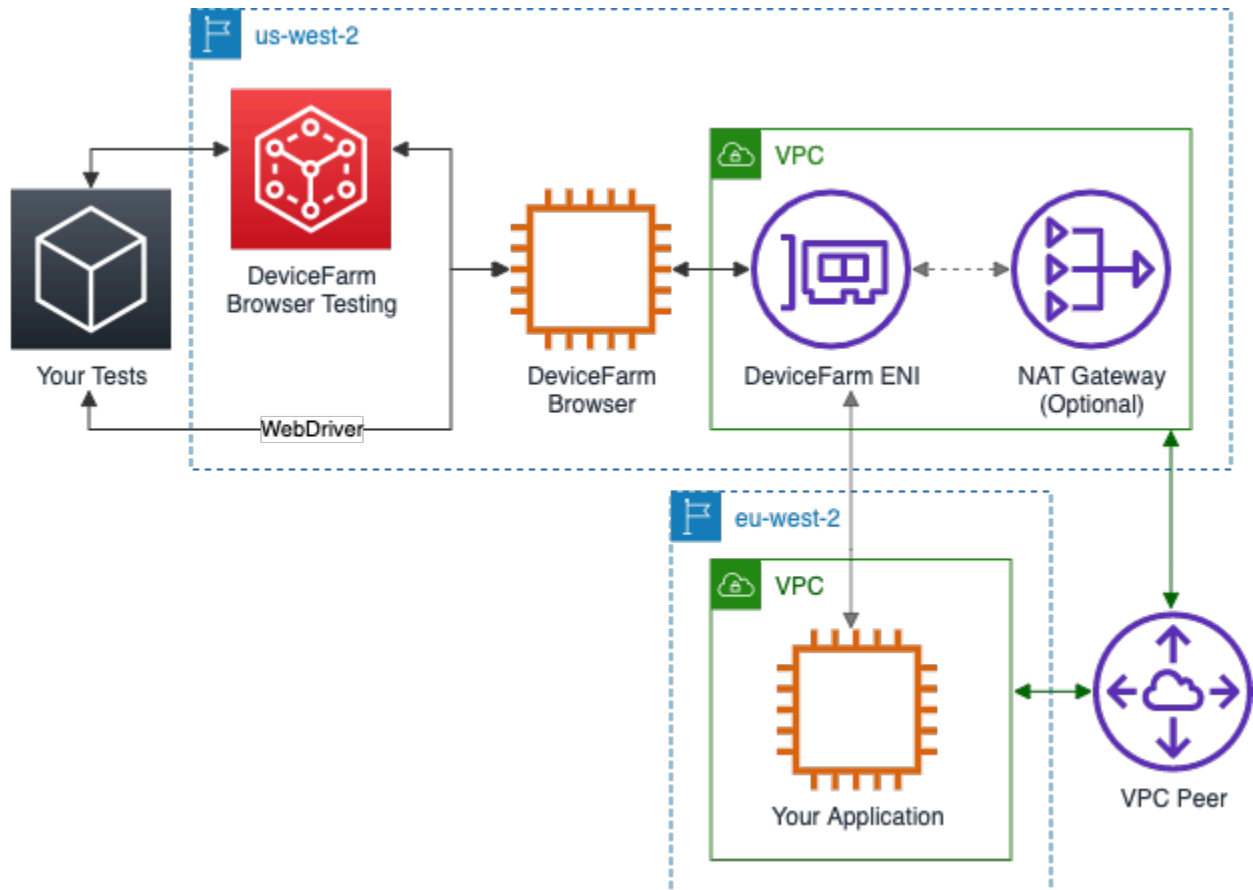


If you have private DNS enabled within your VPC, you can use the DNS names within the VPC to access those resources.

Once you configure VPC access, the browsers that you use for your tests won't be able to connect to resources outside of the VPC, such as public CDNs, unless there is a NAT gateway that you specify within the VPC. For more information, see [NAT gateways](#) in the *Amazon VPC User Guide*.

As part of using Amazon VPC endpoints with Device Farm desktop browser testing, Device Farm creates an AWS Identity and Access Management (IAM) service-linked role. For more information, see [Using service-linked roles for Device Farm](#).

Device Farm can connect to VPCs only within the us-west-2 AWS Region. To access resources in a VPC in another Region, you must create a VPC in the us-west-2 Region and peer the VPCs. For information on peering VPCs, see the [Amazon VPC Peering Guide](#).



For information on using AWS CloudFormation to automatically create and peer VPCs, see the [VPC Peering templates](#) in the AWS CloudFormation template repository on GitHub.

Topics

- [Configuring your project to use Amazon VPC endpoints](#)
- [Removing an Amazon VPC configuration from a Device Farm desktop browser testing project](#)

Configuring your project to use Amazon VPC endpoints

You must configure Amazon VPC connections on a per-project basis. At this time, you can configure only one endpoint per project. When you configure a VPC, Device Farm creates an interface within your VPC and assigns it to the specified subnets and security groups. All future sessions associated with the project use the configured VPC connection.

Important

If you use your VPC with a TestGrid session, you may incur additional bandwidth charges if your VPC has a public-facing NAT gateway and isn't using an S3 gateway endpoint. The reason for this is desktop browser sessions provide test artifacts after your tests are complete and, to make them readily available after your session has been closed, the host used for your desktop browser test session will periodically synchronize your session's artifacts into Device Farm's S3 bucket. When you use a public-facing NAT gateway without an S3 gateway endpoint with your VPC, all traffic for test artifact synchronization traverses through the NAT gateway, which may incur additional bandwidth charges. For more information, see [Amazon VPC Pricing](#).

To avoid incurring additional bandwidth charges, we recommend that you use an S3 gateway endpoint in your VPC if your VPC has a public-facing NAT gateway. For more information, see [Gateway endpoints](#) in the *AWS PrivateLink Guide*.

To configure VPC access for a project, you must know:

- The VPC ID where your app is hosted.
- The applicable security groups to apply to the connection.
- The subnets which will be associated with the connection. When a session starts, the largest available subnet is used.

Additionally, to verify that you have access to your specified VPC when you configure the connection, you must configure certain Amazon Elastic Compute Cloud (Amazon EC2) permissions for Device Farm. For more information, see the [relevant IAM policy](#) in this guide for configuring VPC connections.

For existing Device Farm desktop browser testing projects, you can update the Amazon VPC configuration using the console or the AWS Command Line Interface (AWS CLI):

Console

To update the Amazon VPC configuration using the console

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
2. In the navigation pane, choose **Desktop Browser Testing**, and then choose **Projects**.

3. Under **Desktop browser testing projects**, choose the name of your project.
4. Choose **Project settings**.
5. In the **Virtual Private Cloud (VPC) Settings** section, you can change the **VPC**, **Subnets**, and **Security Groups**.
6. Choose **Save**.

CLI

To update the Amazon VPC configuration using the AWS CLI

Use the following AWS CLI command to update the Amazon VPC configuration:

```
$ aws devicefarm update-test-grid-project \
  --project-arn arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655440000 \
  --vpc-config '{
    "securityGroupIds": ["sg-123456789", ...],
    "subnetIds": ["subnet-123456789", ...],
    "vpcId": "vpc-1234abcd5678"
  }'
```

You can also configure Amazon VPC when creating your project:

```
$ aws devicefarm create-test-grid-project \
  --name "My Testing Project" \
  --vpc-config '{
    "securityGroupIds": ["sg-123456789", ...],
    "subnetIds": ["subnet-123456789", ...],
    "vpcId": "vpc-1234abcd5678"
  }'
```

Note

The JSON presented here is written over multiple lines for readability.

Removing an Amazon VPC configuration from a Device Farm desktop browser testing project

Console

To remove the Amazon VPC configuration through the console

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
2. In the navigation pane, choose **Desktop Browser Testing**, and then choose **Projects**.
3. Under **Desktop browser testing projects**, choose the name of your project.
4. Choose **Project settings**.
5. Under **Virtual Private Cloud (VPC) Settings**, for **VPC**, choose **No VPC**.
6. Choose **Save**.

CLI

To remove the Amazon VPC configuration through the AWS CLI

To remove the Amazon VPC configuration using the AWS CLI, use the `update-test-grid-project` command and pass a blank `vpc-config` parameter:

```
$ aws devicefarm update-test-grid-project \
  --project-arn arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655440000 \
  --vpc-config ''
```

To delete the service-linked role that Device Farm created for accessing your Amazon VPC resources, use the following AWS CLI command:

```
$ aws iam delete-service-linked-role --role-name AWSServiceRoleForDeviceFarmTestGrid
```

Tagging AWS Device Farm resources

AWS Device Farm works with the AWS Resource Groups Tagging API. This API allows you to manage resources in your AWS account with *tags*. You can add tags to resources, such as projects and test runs.

You can use tags to:

- Organize your AWS bill to reflect your own cost structure. To do this, sign up to get your AWS account bill with tag key values included. Then, to see the cost of combined resources, organize your billing information according to resources with the same tag key values. For example, you can tag several resources with an application name, and then organize your billing information to see the total cost of that application across several services. For more information, see [Cost Allocation and Tagging](#) in *About AWS Billing and Cost Management*.
- Control access through IAM policies. To do so, create a policy that allows access to a resource or set of resources using a tag value condition.
- Identify and manage runs that have certain properties as tags, such as the branch used for testing.

For more information about tagging resources, see the [Tagging Best Practices](#) whitepaper.

Topics

- [Tagging resources](#)
- [Looking up resources by tag](#)
- [Removing tags from resources](#)

Tagging resources

The AWS Resource Group Tagging API allows you to add, remove, or modify tags on resources. For more information, see the [AWS Resource Group Tagging API Reference](#).

To tag a resource, use the [TagResources](#) operation from the `resourcegroupstaggingapi` endpoint. This operation takes a list of ARNs from supported services and a list of key-value pairs. The value is optional. An empty string indicates that there should be no value for that tag. For example, the following Python example tags a series of project ARNs with the tag `build-config` with the value `release`:

```
import boto3

client = boto3.client('resourcegroupstaggingapi')

client.tag_resources(ResourceARNList=["arn:aws:devicefarm:us-
west-2:111122223333:project:123e4567-e89b-12d3-a456-426655440000",
                                   "arn:aws:devicefarm:us-
west-2:111122223333:project:123e4567-e89b-12d3-a456-426655441111",
                                   "arn:aws:devicefarm:us-
west-2:111122223333:project:123e4567-e89b-12d3-a456-426655442222"],
                    Tags={"build-config":"release", "git-commit":"8fe28cb"})
```

A tag value is not required. To set a tag with no value, use an empty string ("") when specifying a value. A tag can only have one value. Any previous value a tag has for a resource will be overwritten with the new value.

Looking up resources by tag

To look up resources by their tags, use the `GetResources` operation from the `resourcegroupstaggingapi` endpoint. This operation takes a series of filters, none of which are required, and returns the resources that match the given criteria. With no filters, all tagged resources are returned. The `GetResources` operation allows you to filter resources based on

- Tag value
- Resource type (for example, `devicefarm:run`)

For more information, see the [AWS Resource Group Tagging API Reference](#).

The following example looks up Device Farm desktop browser testing sessions (`devicefarm:testgrid-session` resources) with the tag stack that have the value `production`:

```
import boto3
client = boto3.client('resourcegroupstaggingapi')
sessions = client.get_resources(ResourceTypeFilters=['devicefarm:testgrid-session'],
                               TagFilters=[
                                   {"Key":"stack","Values":["production"]}
                               ])
```

Removing tags from resources

To remove a tag, use the `UntagResources` operation, specifying a list of resources and the tags to remove:

```
import boto3
client = boto3.client('resourcegroupstaggingapi')
client.UntagResources(ResourceARNList=["arn:aws:devicefarm:us-
west-2:111122223333:project:123e4567-e89b-12d3-a456-426655440000"], TagKeys=["RunCI"])
```


Troubleshooting Device Farm desktop browser testing

In this section, you'll find some common problems you can run into and how to address them.

Topics

- [Can't connect to a RemoteWebDriver](#)
- [Timeouts when setting up RemoteWebDriver](#)
- [Rate limit errors during session creation](#)
- [I get errors during session creation](#)
- [Browser can't connect to my app](#)
- [Selenium 3 WebDriver testing framework is not working properly with Microsoft Edge \(Chromium\)](#)
- [My tests using XPath are slow](#)

Can't connect to a RemoteWebDriver

You might see this error when you have an expired URL from `CreateTestGridUrl`. The URL is valid only for a short time, as specified in the API call. Even if a session is created before the expiry, it cannot be accessed after the expiration of the `RemoteWebDriver` URL.

Timeouts when setting up RemoteWebDriver

If your testing framework enforces strict timeouts for test completion, you might see tests time out when setting up a new session. When you start a session, there is a 60 to 120 delay between the request to create the session and its availability for use. In situations where you cannot increase timeouts, we recommend that you request a session and open a page before your test framework sets up your tests, and then re-use that session for your test suite. Make sure to close your session when your tests are complete.

Rate limit errors during session creation

You have too many open sessions, are opening too many sessions per second, or are making too many `WebDriver` requests per second. This is common in large, highly parallel test suites where many browsers are being controlled at once.

For more information, see [Quotas in Device Farm desktop browser testing](#).

Session creation rate management

If you are opening more than 5 sessions at a time, we recommend adding a random delay before your session creation. A uniformly random delay of 0.5 to 5 seconds will stagger your session creation and avoid rate limiting during this time.

Actions per second rate management

To mitigate the rate limit of 150 actions per second across WebDriver sessions, you can use a custom `CommandExecutor` that automatically limits the rate of outgoing requests. The following example demonstrates how to accomplish this:

Java

```
import org.openqa.selenium.remote.HttpCommandExecutor;
import org.openqa.selenium.remote.Response;
import org.openqa.selenium.remote.Command;
import java.io.IOException;
import java.time.Instant;
import java.util.concurrent.TimeUnit;

class RateLimitExecutor extends HttpCommandExecutor {

    public static final int CONCURRENT_SESSIONS = 100;
    public static final int ACTIONS_RATE_LIMIT_PER_SECOND = 150;

    public static final double SECONDS_PER_ACTION = ((double) CONCURRENT_SESSIONS)
        / ((double) ACTIONS_RATE_LIMIT_PER_SECOND);
    private long lastExecutionTime;

    public RateLimitExecutor(URL addressOfRemoteServer) {
        super(addressOfRemoteServer);
        lastExecutionTime = 0;
    }

    public Response execute(Command command) throws IOException {
        long currentTime = Instant.now().toEpochMilli();
        double elapsedTime = TimeUnit.MILLISECONDS.toSeconds(currentTime -
lastExecutionTime);
```

```

        if (elapsedTime < SECONDS_PER_ACTION) {
            try {
                Thread.sleep(TimeUnit.SECONDS.toMillis((long)(SECONDS_PER_ACTION -
elapsedTime)));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        lastExecutionTime = Instant.now().toEpochMilli();
        return super.execute(command);
    }
}

```

When calling the constructor for the remote `WebDriver`, use an instance of this new class as the `command_executor` parameter:

```

RemoteWebDriver driver = new RemoteWebDriver(new RateLimitExecutor(testGridUrl), new
FirefoxOptions());

```

Python

```

CONCURRENT_SESSIONS = 100
ACTIONS_RATE_LIMIT_PER_SECOND = 150

class RateLimitExecutor(RemoteConnection):

    def __init__(self, *args):
        super().__init__(*args)
        self.last_execution_time = None
        self.seconds_per_action = CONCURRENT_SESSIONS /
ACTIONS_RATE_LIMIT_PER_SECOND

    def execute(self, *args):
        if self.last_execution_time:
            current_time = time.time()
            elapsed_time = current_time - self.last_execution_time
            if elapsed_time < self.seconds_per_action:
                time.sleep(self.seconds_per_action - elapsed_time)
        self.last_execution_time = time.time()
        return super().execute(*args)

```

When calling the constructor for the remote `WebDriver`, use an instance of this new class as the `command_executor` parameter:

```
driver = webdriver.Remote(command_executor=CustomCommandExecutor(test_grid_url),
    desired_capabilities=DesiredCapabilities.CHROME)
```

I get errors during session creation

If you get errors during session creation or are getting a browser that you don't expect, check that your requested capabilities are valid. Make sure:

- the `browserName` capability contains a supported browser name
- the `browserName` capability value is lower case
- the `browserVersion` capability is set to `latest`, `latest-1`, or `latest-2`
- the `platform` capability is set to a supported platform
- the URL of the `WebDriver` hub has not expired

For more information, see

- [Supported browsers](#)
- [Supported platforms](#)

Browser can't connect to my app

You might see this in the following cases:

- Verify the DNS hostname that is being used to connect to your application. If you are using Amazon VPC, verify that private DNS is enabled and configured properly.
- If you are using a Amazon VPC configuration, verify that the correct subnets and security groups have been applied and that the specified VPC is in the `us-west-2` region. Additionally, if your application requires access to the internet, for example to access a CDN, those resources must be accessible from within the VPC or through a NAT gateway. For more information on NAT gateways, see [NAT gateways](#) in the *Amazon VPC user guide*.
- You must not block traffic from Amazon EC2 or other AWS services.

Selenium 3 WebDriver testing framework is not working properly with Microsoft Edge (Chromium)

Selenium 3 requires the *Selenium Tools for Microsoft Edge* package to be able to support the Edge browser. For more information, see [Using Selenium 3](#).

Note that Selenium 4 contains built-in support for Edge.

My tests using XPath are slow

Due to the design of Selenium and the desktop browser testing feature, tests using XPath are slow if they are not optimized properly.

If you use `find_elements_by_xpath`, consider the following optimizations:

- Use an ID selector instead of XPath.
- In Python, avoid looping over the results multiple times or casting the results to a `list`, using `map`, or other late-binding iterators without caching the results. Use `str(your-element.getAttribute(...))` when requesting strings.
- If you are using the list multiple times, cache the results if possible.

Selenium makes a full round-trip web request for each action you take on the returned elements on an XPath query. For example, if you request every link in a page (`//a`) and want to validate that every link points to a real resource, each call to `getAttribute` from your test results in a full round-trip request to the desktop browser testing feature running your test and the WebDriver automating your browser, even if the content has not changed.

Quotas in Device Farm desktop browser testing

Exceeding the following limits will result in session creation failure:

- You may have up to 50 sessions in an active state at any time.
- You may create up to 5 sessions per second.
- You may call `createTestGridUrl` up to 10 times a second.
- No POST payload may be greater than 30MB.

If you have too many open sessions or create them too fast, session creation will fail. If you require more than 50 concurrent sessions, open a technical support case with your use case. For more information about increasing your service quota, see [AWS Service Quotas](#).

Security in Device Farm desktop browser testing

This section describes data collection and how to control access to resources when you use browser testing in AWS Device Farm.

Topics

- [Your data in Device Farm desktop browser testing](#)
- [Access control and IAM](#)
- [Using service-linked roles for Device Farm](#)

Your data in Device Farm desktop browser testing

Device Farm does not collect the content of your web application except what's required to run the service.

Your tests are run in isolated instances. They are not shared by any other user or test.

Artifacts (logs, video, and so on) are associated with your account. Files that you download in your tests are not collected. Any content that is saved in the browser (for example, cookies or LocalDB storage) is lost when your session ends.

Access control and IAM

Be aware of the following when you use the desktop browser testing feature:

- Your AWS access and secret keys are used for all actions with the SDK and CLI.
- URLs generated with the `CreateTestGridUrl` API call can be used to create sessions on your behalf until the URL expires.
- Artifact URLs can be viewed by anyone with the limited lifetime URL.

The desktop browser testing feature is integrated with AWS Identity and Access Management (IAM), which you can use to:

- Create users and groups in your AWS account.
- Easily share your AWS resources between the users in your AWS account.

- Assign unique security credentials to each user.
- Control each user's access to services and resources.
- Get a single bill for all users in your AWS account.

IAM is used to restrict access to operations and resources. The following example IAM policy grants the user access only to Device Farm resources:

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

The following IAM Policy allows access to Device Farm resources as well as the required permissions for VPC connection configuration:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": [
      "devicefarm:*",
      "ec2:DescribeVpcs",
      "ec2:DescribeSubnets",
      "ec2:DescribeSecurityGroups",
      "ec2:CreateNetworkInterface"
    ],
    "Resource": [
      "*"
    ]
  },
  {
```



```
    "Effect": "Allow",
    "Action": "iam:CreateServiceLinkedRole",
    "Resource": "arn:aws:iam::*:role/aws-service-role/
testgrid.devicefarm.amazonaws.com/AWSServiceRoleForDeviceFarmTestGrid",
    "Condition": {
      "StringLike": {
        "iam:AWSServiceName": "testgrid.devicefarm.amazonaws.com"
      }
    }
  }
]
```

The following IAM policy allows the user to create URLs with the `CreateTestGridUrl` API call:

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

For more information about IAM, see the following:

- [AWS Identity and Access Management \(IAM\)](#)
- [Getting started](#)
- [IAM User Guide](#)

Using service-linked roles for Device Farm

AWS Device Farm uses AWS Identity and Access Management (IAM) [service-linked roles](#). A service-linked role is a unique type of IAM role that is linked directly to Device Farm. Service-linked roles

are predefined by Device Farm and include all the permissions that the service requires to call other AWS services on your behalf.

A service-linked role makes setting up Device Farm easier because you don't have to manually add the necessary permissions. Device Farm defines the permissions of its service-linked roles, and unless defined otherwise, only Device Farm can assume its roles. The defined permissions include the trust policy and the permissions policy, and that permissions policy cannot be attached to any other IAM entity.

You can delete a service-linked role only after first deleting their related resources. This protects your Device Farm resources because you can't inadvertently remove permission to access the resources.

For information about other services that support service-linked roles, see [AWS Services That Work with IAM](#) and look for the services that have **Yes** in the **Service-Linked Role** column. Choose a **Yes** with a link to view the service-linked role documentation for that service.

Service-linked role permissions for Device Farm

Device Farm uses the service-linked role named **AWSServiceRoleForDeviceFarmTestGrid** – Allows Device Farm to access AWS resources on your behalf.

The AWSServiceRoleForDeviceFarmTestGrid service-linked role trusts the following services to assume the role:

- `testgrid.devicefarm.amazonaws.com`

The role permissions policy allows Device Farm to complete the following actions:

- For your account
 - Create network interfaces
 - Describe network interfaces
 - Describe VPCs
 - Describe subnets
 - Describe security groups
 - Delete interfaces
 - Modify network interfaces
- For network interfaces

- Create tags
- For EC2 network interfaces managed by Device Farm
 - Create network interface permissions

The full IAM policy reads:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:DescribeNetworkInterfaces",
        "ec2:DescribeVpcs",
        "ec2:DescribeSubnets",
        "ec2:DescribeSecurityGroups"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:CreateNetworkInterface"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:subnet/*",
        "arn:aws:ec2:*:*:security-group/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:CreateNetworkInterface"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:network-interface/*"
      ],
      "Condition": {
        "StringEquals": {
          "aws:RequestTag/AWSDeviceFarmManaged": "true"
        }
      }
    }
  ]
}
```

```
},
{
  "Effect": "Allow",
  "Action": [
    "ec2:CreateTags"
  ],
  "Resource": "arn:aws:ec2:*:*:network-interface/*",
  "Condition": {
    "StringEquals": {
      "ec2:CreateAction": "CreateNetworkInterface"
    }
  }
},
{
  "Effect": "Allow",
  "Action": [
    "ec2:CreateNetworkInterfacePermission",
    "ec2>DeleteNetworkInterface"
  ],
  "Resource": "arn:aws:ec2:*:*:network-interface/*",
  "Condition": {
    "StringEquals": {
      "aws:ResourceTag/AWSDeviceFarmManaged": "true"
    }
  }
},
{
  "Effect": "Allow",
  "Action": [
    "ec2:ModifyNetworkInterfaceAttribute"
  ],
  "Resource": [
    "arn:aws:ec2:*:*:security-group/*",
    "arn:aws:ec2:*:*:instance/*"
  ]
},
{
  "Effect": "Allow",
  "Action": [
    "ec2:ModifyNetworkInterfaceAttribute"
  ],
  "Resource": "arn:aws:ec2:*:*:network-interface/*",
  "Condition": {
    "StringEquals": {
```

```
    "aws:ResourceTag/AWSDeviceFarmManaged": "true"  
  }  
}  
}  
]  
}
```

You must configure permissions to allow an IAM entity (such as a user, group, or role) to create, edit, or delete a service-linked role. For more information, see [Service-Linked Role Permissions](#) in the *IAM User Guide*.

Creating a service-linked role for Device Farm

When you provide a VPC config for a TestGridProject, you don't need to manually create a service-linked role. When you create your first Device Farm resource in the AWS Management Console, the AWS CLI, or the AWS API, Device Farm creates the service-linked role for you.

If you delete this service-linked role, and then need to create it again, you can use the same process to recreate the role in your account. When you create your first Device Farm resource, Device Farm creates the service-linked role for you again.

You can also use the IAM console to create a service-linked role with the **Device Farm** use case. In the AWS CLI or the AWS API, create a service-linked role with the `testgrid.devicefarm.amazonaws.com` service name. For more information, see [Creating a Service-Linked Role](#) in the *IAM User Guide*. If you delete this service-linked role, you can use this same process to create the role again.

Editing a service-linked role for Device Farm

Device Farm does not allow you to edit the `AWSServiceRoleForDeviceFarmTestGrid` service-linked role. After you create a service-linked role, you cannot change the name of the role because various entities might reference the role. However, you can edit the description of the role using IAM. For more information, see [Editing a Service-Linked Role](#) in the *IAM User Guide*.

Deleting a service-linked role for Device Farm

If you no longer need to use a feature or service that requires a service-linked role, we recommend that you delete that role. That way you don't have an unused entity that is not actively monitored

or maintained. However, you must clean up the resources for your service-linked role before you can manually delete it.

Note

If the Device Farm service is using the role when you try to delete the resources, then the deletion might fail. If that happens, wait for a few minutes and try the operation again.

To manually delete the service-linked role using IAM

Use the IAM console, the AWS CLI, or the AWS API to delete the `AWSServiceRoleForDeviceFarmTestGrid` service-linked role. For more information, see [Deleting a Service-Linked Role](#) in the *IAM User Guide*.

Supported Regions for Device Farm service-linked roles

Device Farm supports using service-linked roles in all of the regions where the service is available. For more information, see [AWS Regions and Endpoints](#).

Device Farm does not support using service-linked roles in every region where the service is available. You can use the `AWSServiceRoleForDeviceFarmTestGrid` role in the following regions.

Region name	Region identity	Support in Device Farm
US East (N. Virginia)	us-east-1	No
US East (Ohio)	us-east-2	No
US West (N. California)	us-west-1	No
US West (Oregon)	us-west-2	Yes
Asia Pacific (Mumbai)	ap-south-1	No
Asia Pacific (Osaka)	ap-northeast-3	No
Asia Pacific (Seoul)	ap-northeast-2	No
Asia Pacific (Singapore)	ap-southeast-1	No

Region name	Region identity	Support in Device Farm
Asia Pacific (Sydney)	ap-southeast-2	No
Asia Pacific (Tokyo)	ap-northeast-1	No
Canada (Central)	ca-central-1	No
Europe (Frankfurt)	eu-central-1	No
Europe (Ireland)	eu-west-1	No
Europe (London)	eu-west-2	No
Europe (Paris)	eu-west-3	No
South America (São Paulo)	sa-east-1	No
AWS GovCloud (US)	us-gov-west-1	No

Document history for Device Farm desktop browser testing

The following table describes the documentation for this release of the guide.

- **API version: 2015-06-23**
- **Latest documentation update:** December 20, 2019

Change	Description	Date
Rate Limit Workarounds	We've added workarounds for common ways that test suites run into rate limits. For more information, see ??? .	September 25, 2020
Initial Release	This is the initial release of the guide.	December 20, 2019