Developer Guide

AWS Device Farm

API Version 2015-06-23

AWS Device Farm: Developer Guide

Table of Contents

What is AWS Device Farm?

Device Farm is an app testing service that you can use to test and interact with your Android, iOS, and web apps on real, physical phones and tablets that are hosted by Amazon Web Services (AWS).

There are two main ways to use Device Farm:

- Automated testing of apps using a variety of testing frameworks.
- Remote access of devices onto which you can load, run, and interact with apps in real time.

> **Note**
>
> Device Farm is only available in the `us-west-2` (Oregon) region.

Automated app testing

Device Farm allows you to upload your own tests or use built-in, script-free compatibility tests. Because testing is performed in parallel, tests on multiple devices begin in minutes.

As tests are completed, a test report that contains high-level results, low-level logs, pixel-to-pixel screenshots, and performance data is updated.

Device Farm supports testing of native and hybrid Android and iOS apps, including those created with PhoneGap, Titanium, Xamarin, Unity, and other frameworks. It supports remote access of Android and iOS apps for interactive testing. For more information about supported test types, see Test frameworks and built-in tests in AWS Device Farm.

Remote access interaction

Remote access allows you to swipe, gesture, and interact with a device through your web browser in real time. There are a number of situations where real-time interaction with a device is useful. For example, customer service representatives can guide customers through the use or setup of their device. They can also walk customers through the use of apps running on a specific device. You can install apps on a device running in a remote access session and then reproduce customer problems or reported bugs.

During a remote access session, Device Farm collects details about actions that take place as you interact with the device. Logs with these details and a video capture of the session are produced at the end of the session.

Terminology

Device Farm introduces the following terms that define the way information is organized:

device pool

A collection of devices that typically share similar characteristics, such as platform, manufacturer, or model.

job

A request for Device Farm to test a single app against a single device. A job contains one or more suites.

metering

Refers to billing for devices. You might see references to metered devices or unmetered devices in the documentation and API reference. For more information about pricing, see AWS Device Farm Pricing.

project

A logical workspace that contains runs, one run for each test of a single app against one or more devices. You can use projects to organize workspaces in whatever way you choose. For example, you can have one project per app title or one project per platform. You can create as many projects as you need.

report

Contains information about a run, which is a request for Device Farm to test a single app against one or more devices. For more information, see Reports in AWS Device Farm.

run

A specific build of your app, with a specific set of tests, to be run on a specific set of devices. A run produces a report of the results. A run contains one or more jobs. For more information, see Runs.

session

A real-time interaction with an actual, physical device through your web browser. For more information, see Sessions.

suite

The hierarchical organization of tests in a test package. A suite contains one or more tests.

test

An individual test case in a test package.

For more information about Device Farm, see Concepts.

Setting up

To use Device Farm, see Setting up.

Setting up AWS Device Farm

Before you use Device Farm for the first time, you must complete the following tasks:

Topics

Step 1: Sign up for AWS

Sign up for Amazon Web Services (AWS).

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

Step 2: Create or use an IAM user in your AWS account

We recommend that you do not use your AWS root account to access Device Farm. Instead, create an AWS Identity and Access Management (IAM) user (or use an existing one) in your AWS account, and then access Device Farm with that IAM user.

For more information, see Creating an IAM User (AWS Management Console).

Step 3: Give the IAM user permission to access Device Farm

Give the IAM user permission to access Device Farm. To do this, create an access policy in IAM, and then assign the access policy to the IAM user, as follows.

> **Note**
>
> The AWS root account or IAM user that you use to complete the following steps must have permission to create the following IAM policy and attach it to the IAM user. For more information, see Working with Policies.

1. Create a policy with the following JSON body. Give it a descriptive title, such as *DeviceFarmAdmin*.

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

   For more information on creating IAM policies, see Creating IAM Policies in the IAM User Guide.

2. Attach the IAM policy you created to your new user. For more information on attaching IAM policies to users, see Adding and Removing IAM Policies in the IAM User Guide.

Attaching the policy provides the IAM user with access to all Device Farm actions and resources associated with that IAM user. For information about how to restrict IAM users to a limited set of Device Farm actions and resources, see Identity and access management in AWS Device Farm.

Next step

You are now ready to start using Device Farm. See Getting started with Device Farm.

Getting started with Device Farm

This walkthrough shows you how to use Device Farm to test a native Android or iOS app. You use the Device Farm console to create a project, upload an .apk or .ipa file, run a suite of standard tests, and then view the results.

> **Note**
>
> Device Farm is available only in the `us-west-2` (Oregon) AWS Region.

Topics

Prerequisites

Before you begin, make sure you have completed the following requirements:

- Complete the steps in Setting up. You need an AWS account and an AWS Identity and Access Management (IAM) user with permission to access Device Farm.

- For Android, you need an .apk (Android app package) file. For iOS, you need an .ipa (iOS app archive) file. You upload the file to Device Farm later in this walkthrough.

  > **Note**
  >
  > Make sure that your .ipa file is built for an iOS device and not for a simulator.

- (Optional) You need a test from one of the testing frameworks that Device Farm supports. You upload this test package to Device Farm, and then run the test later in this walkthrough. If you don't have a test package available, you can specify and run a standard built-in test suite. For more information, see Test frameworks and built-in tests in AWS Device Farm.

Step 1: Sign in to the console

You can use the Device Farm console to create and manage projects and runs for testing. You learn about projects and runs later in this walkthrough.

- Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.

Step 2: Create a project

To test an app in Device Farm, you must first create a project.

1. In the navigation pane, choose **Mobile Device Testing**, and then choose **Projects**.
2. Under **Mobile Device Testing Projects**, choose **New project**.
3. Under **Create project**, enter a **Project Name** (for example, **`MyDemoProject`**).
4. Choose **Create**.

   The console opens the **Automated tests** page of your newly created project.

Step 3: Create and start a run

Now that you have a project, you can create and then start a run. For more information, see Runs.

1. On the **Automated tests** page, choose **Create a new run**.
2. On the **Choose application** page, under **Mobile App**, choose **Choose File**, and then choose an Android (.apk) or iOS (.ipa) file from your computer. Or, drag the file from your computer and drop it in the console.
3. Enter a **Run name**, such as **`my first test`**. By default, the Device Farm console uses the file name.
4. Choose **Next**.
5. On the **Configure** page, under **Setup test framework**, choose one of the testing frameworks or built-in test suites. For information about each option, see *Test frameworks and built-in tests*.

- If you have not yet packaged your tests for Device Farm, choose **Built-in: Fuzz** to run a standard, built-in test suite. You can keep the default values for **Event count**, **Event throttle**, and **Randomizer seed**. For more information, see the section called “Built-in: fuzz (Android and iOS)”.
- If you have a test package from one of the supported testing frameworks, choose the corresponding testing framework, and then upload the file that contains your tests.

6. Choose **Next**.
7. On the **Select devices** page, for **Device pool**, choose **Top Devices**.
8. Choose **Next**.
9. On the **Specify device state** page, do any of the following:
   - To provide additional data for Device Farm to use during the run, under **Add extra data**, upload a .zip file.
   - To install other apps for the run, under **Install other apps**, upload the .apk or .ipa files for the apps. To change the installation order, drag and drop the files.
   - To turn on Wi-Fi, Bluetooth, GPS, or NFC radios for the run, under **Set radio states**, select the corresponding check boxes.
   - To test location-specific behavior during the run, under **Device location**, specify preset **Latitude** and **Longitude** coordinates.
   - To preset device language and region for the run, under **Device locale**, choose a locale.
   - To preset the network profile for the run, under **Network profile**, choose a curated profile. Or, choose **Create network profile** to create your own.

   > **Note**
   >
   > Setting the device radio state and locale are options only available for Android native tests at this time.
10. Choose **Next**.
11. On the **Review and start run** page, choose **Confirm and start run**.

Device Farm starts the run as soon as devices are available, typically within a few minutes. To view the run status, on the **Automated tests** page of your project, choose the name of your run. One the run page, under **Devices**, each device starts with the pending icon in the device table, then switches to the running icon

when the test begins. As each test finishes, the console displays a test result icon next to the device name. When all tests are complete, the pending icon next to the run changes to a test result icon.

Step 4: View the run's results

To view test results from the run, on the **Automated tests** page of your project, choose the name of your run. A summary page displays:

- The total number of tests, by outcome.
- Lists of tests with unique warnings or failures.
- A list of devices with test results for each.
- Any screenshots captured during the run, grouped by device.
- A section to download the parsing result.

For more information, see Viewing test reports in Device Farm.

Next steps

For more information about Device Farm, see Concepts.

Purchasing a device slot in Device Farm

You can use the Device Farm console, AWS Command Line Interface (AWS CLI), or Device Farm API to purchase a device slot.

Purchase device slots (console)

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. In the navigation pane, choose **Mobile Device Testing**, and then choose **Device slots**.
3. On the **Purchase and manage device slots** page, you can create your own custom package by choosing the number of slots of **Automated testing** and **Remote access** devices that you want to purchase. Specify slot amounts for both the current and next billing periods.

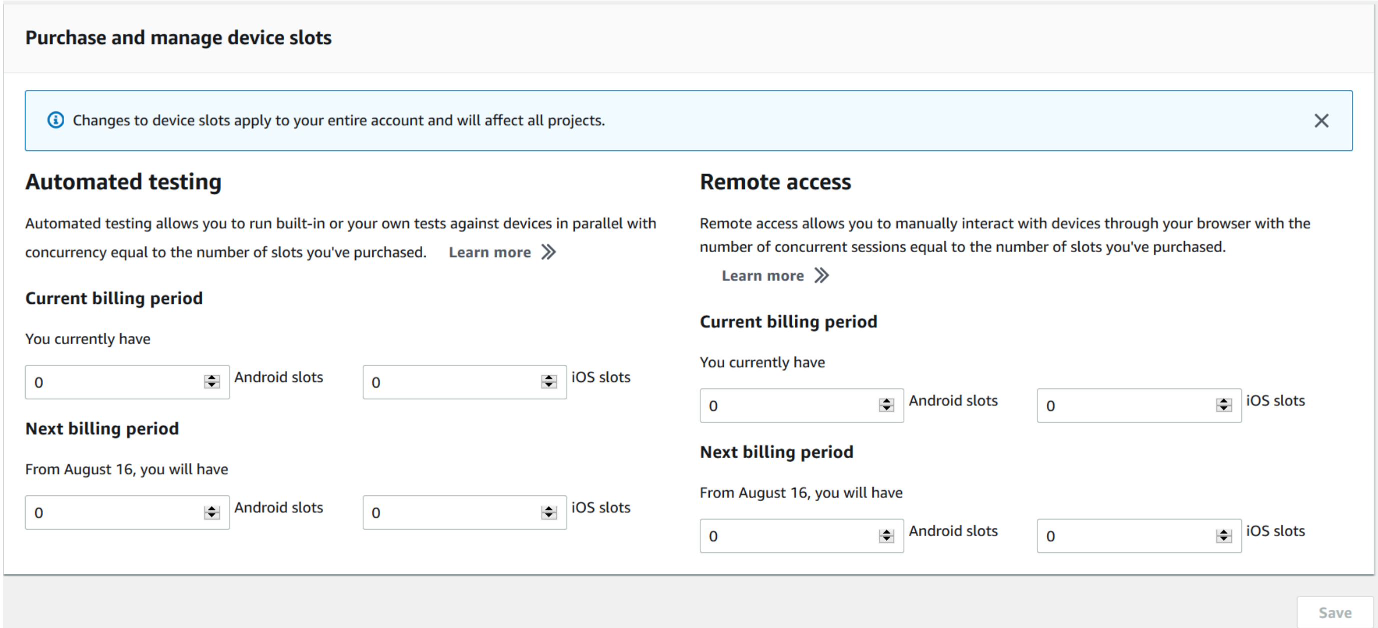
   As you change the slot amount, the text dynamically updates with the billing amount. For more information, see AWS Device Farm pricing.

   Important

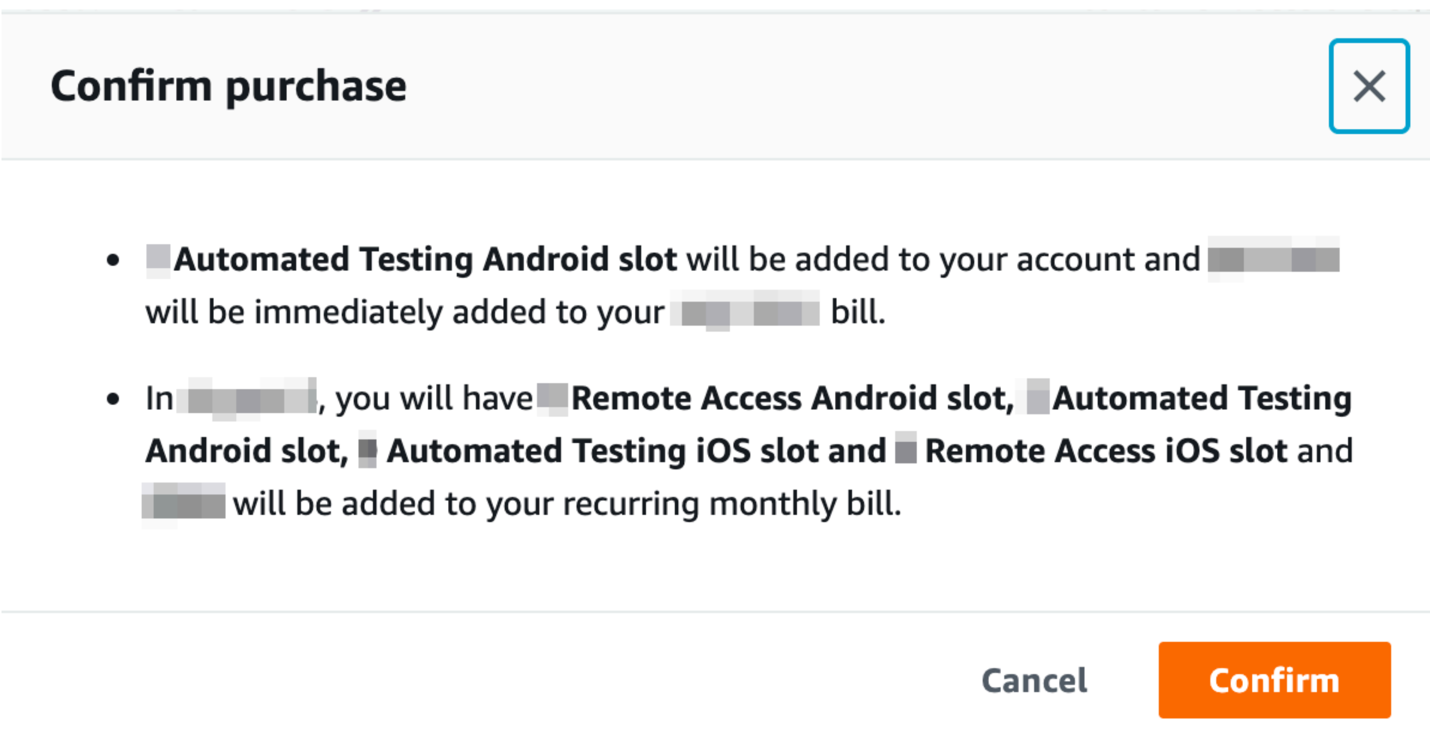
   If you change the number of device slots but see a **contact us** or **contact us to purchase** message, your AWS account is not yet approved to purchase the number of device slots that you requested.
   These options prompt you to send an email to the Device Farm support team. In the email, specify the number of each device type that you want to purchase and for which billing cycle.

   Note

   Changes to the device slots apply to your entire account and affect all projects.



4. Choose **Purchase**. A **Confirm purchase** window will appear. Review the information and then choose **Confirm** to complete the transaction.



On the **Purchase and manage device slots** page, you can see the number of device slots that you currently have. If you increased or decreased the number of slots, you'll see the number of slots that you'll have one month after the date you made the change.

Purchase a device slot (AWS CLI)

You can run the **purchase-offering** command to purchase the offering.

To list your Device Farm account settings, including the maximum number of device slots that you can purchase and the number of remaining free trial minutes, run the **get-account-settings** command. You'll see output similar to the following:

```
{
    "accountSettings": {
        "maxSlots": {
            "GUID": 1,
            "GUID": 1,
            "GUID": 1,
            "GUID": 1
        },
        "unmeteredRemoteAccessDevices": {
            "ANDROID": 0,
            "IOS": 0
        },
        "maxJobTimeoutMinutes": 150,
        "trialMinutes": {
            "total": 1000.0,
            "remaining": 954.1
        },
        "defaultJobTimeoutMinutes": 150,
        "awsAccountNumber": "AWS-ACCOUNT-NUMBER",
        "unmeteredDevices": {
            "ANDROID": 0,
            "IOS": 0
        }
    }
}
```

To list the device slot offerings that are available to you, run the **list-offerings** command. You should see output similar to the following:

```
{
```

```
    "offerings": [
        {
            "recurringCharges": [
                {
                    "cost": {
                        "amount": 250.0,
                        "currencyCode": "USD"
                    },
                    "frequency": "MONTHLY"
                }
            ],
            "platform": "IOS",
            "type": "RECURRING",
            "id": "GUID",
            "description": "iOS Unmetered Device Slot"
        },
        {
            "recurringCharges": [
                {
                    "cost": {
                        "amount": 250.0,
                        "currencyCode": "USD"
                    },
                    "frequency": "MONTHLY"
                }
            ],
            "platform": "ANDROID",
            "type": "RECURRING",
            "id": "GUID",
            "description": "Android Unmetered Device Slot"
        },
        {
            "recurringCharges": [
                {
                    "cost": {
                        "amount": 250.0,
                        "currencyCode": "USD"
                    },
                    "frequency": "MONTHLY"
                }
            ],
            "platform": "ANDROID",
            "type": "RECURRING",
            "id": "GUID",
```

```
            "description": "Android Remote Access Unmetered Device Slot"
        },
        {
            "recurringCharges": [
                {
                    "cost": {
                        "amount": 250.0,
                        "currencyCode": "USD"
                    },
                    "frequency": "MONTHLY"
                }
            ],
            "platform": "IOS",
            "type": "RECURRING",
            "id": "GUID",
            "description": "iOS Remote Access Unmetered Device Slot"
        }
    ]
}
```

To list the available offering promotions, run the **list-offering-promotions** command.

> **Note**
>
> This command returns only promotions that you haven't yet purchased. As soon as you purchase one or more slots across any offering using a promotion, that promotion no longer appears in the results.

You should see output similar to the following:

```
{
    "offeringPromotions": [
        {
            "id": "2FREEMONTHS",
            "description": "New device slot customers get 3 months for the price of 1."
        }
    ]
}
```

To get the offering status, run the **get-offering-status** command. You should see output similar to the following:

```
{
    "current": {
        "GUID": {
            "offering": {
                "platform": "IOS",
                "type": "RECURRING",
                "id": "GUID",
                "description": "iOS Unmetered Device Slot"
            },
            "quantity": 1
        },
        "GUID": {
            "offering": {
                "platform": "ANDROID",
                "type": "RECURRING",
                "id": "GUID",
                "description": "Android Unmetered Device Slot"
            },
            "quantity": 1
        }
    },
    "nextPeriod": {
        "GUID": {
            "effectiveOn": 1459468800.0,
            "offering": {
                "platform": "IOS",
                "type": "RECURRING",
                "id": "GUID",
                "description": "iOS Unmetered Device Slot"
            },
            "quantity": 1
        },
        "GUID": {
            "effectiveOn": 1459468800.0,
            "offering": {
                "platform": "ANDROID",
                "type": "RECURRING",
                "id": "GUID",
                "description": "Android Unmetered Device Slot"
            },
            "quantity": 1
        }
    }
```

```
}
```

The **renew-offering** and **list-offering-transactions** commands are also available for this feature. For more information, see the AWS CLI reference.

Purchase a device slot (API)

1. Call the GetAccountSettings operation to list your account settings.
2. Call the ListOfferings operation to list the device slot offerings available to you.
3. Call the ListOfferingPromotions operation to list the offering promotions that are available.

   > **Note**
   >
   > This command returns only promotions that you haven't yet purchased. As soon as you purchase one or more slots using an offering promotion, that promotion no longer appears in the results.

4. Call the PurchaseOffering operation to purchase an offering.
5. Call the GetOfferingStatus operation to get the offering status.

The RenewOffering and ListOfferingTransactions commands are also available for this feature.

For information about using the Device Farm API, see Automating Device Farm.

Cancelling a device slot in Device Farm

You can cancel the number of device slots for both automated testing and remote access. For instructions, see one of the following sections. The amount charged to your account for the next billing cycle will be listed underneath the billing period field.

For more information about device slots, see Purchasing a device slot in Device Farm.

Cancel a device slot (console)

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. In the navigation pane, choose **Mobile Device Testing**, and then choose **Device slots**.

3. On the **Purchase and manage device slots** page, you can decrease the number of device slots for both automated testing and remote access by decreasing the value under **Next billing period**. The amount charged to your account for the next billing cycle will be listed underneath the billing period field.
4. Choose **Save**. A **Confirm Change** window will appear. Review the information and then choose **Confirm** to complete the transaction.

Cancel a device slot (AWS CLI)

You can run the **renew-offering** command to change the amount of devices for the next billing cycle.

Cancel a device slot (API)

Call the RenewOffering operation to change the quantity of devices in your account.

AWS Device Farm concepts

Device Farm is an app testing service that you can use to test and interact with your Android, iOS, and web apps on real, physical phones and tablets that are hosted by Amazon Web Services (AWS).

This section describes important Device Farm concepts.

For more information about supported test types in Device Farm, see Test frameworks and built-in tests in AWS Device Farm.

Device support in AWS Device Farm

The following sections provide information about device support in Device Farm.

Topics

Supported devices

Device Farm provides support for hundreds of unique, popular Android and iOS devices and operating system combinations. The list of available devices grows as new devices enter the market. For the full list of devices, see Device List.

Device pools

Device Farm organizes its devices into device pools that you can use for your testing. These device pools contain related devices, such as devices that run only on Android or only on iOS. Device Farm provides curated device pools, such as those for top devices. You can also create device pools that mix public and private devices.

Private devices

Private devices allow you to specify exact hardware and software configurations for your testing needs. Certain configurations, such as rooted Android devices, can be supported as private devices. Each private device is a physical device that Device Farm deploys on your behalf in an Amazon data center. Your private devices are available exclusively to you for both automated and manual testing. After you choose to end your subscription, the hardware is removed from our environment. For more information, see Private Devices and Private devices in AWS Device Farm.

Device branding

Device Farm runs tests on physical mobile and tablet devices from a variety of OEMs.

Device slots

Device slots correspond to concurrency in which the number of device slots you have purchased determines how many devices you can run in tests or remote access sessions.

There are two types of device slots:

- A *remote access device slot* is one you can run in remote access sessions concurrently.

  If you have one remote access device slot, you can only run one remote access session at a time. If you purchase additional remote testing device slots, you can run multiple sessions concurrently.
- An *automated testing device slot* is one on which you can run tests concurrently.

If you have one automated testing device slot, you can only run tests on one device at a time. If you purchase additional automated testing device slots, you can run multiple tests concurrently, on multiple devices, to get test results faster.

You can purchase device slots based on the device family (Android or iOS devices for automated testing and Android or iOS devices for remote access). For more information, see Device Farm Pricing.

Preinstalled device apps

Devices in Device Farm include a small number of apps that are already installed by manufacturers and carriers.

Device capabilities

All devices have a Wi-Fi connection to the internet. They do not have carrier connections and cannot make phone calls or send SMS messages.

You can take photos with any device that supports a front- or rear-facing camera. Due to the way the devices are mounted, photos might look dark and blurry.

Google Play Services is installed on devices that support it, but these devices do not have an active Google account.

Test environments in AWS Device Farm

AWS Device Farm provides both custom and standard test environments for running your automated tests. You can choose a custom test environment for complete control over your automated tests. Or, you can choose the Device Farm default standard test environment, which offers granular reporting of each test in your automated test suite.

Topics

- Standard test environment
- Custom test environment

Standard test environment

When you run a test in the standard environment, Device Farm provides detailed logs and reporting for every case in your test suite. You can view performance data, videos, screenshots, and logs for each test to pinpoint and fix issues in your app.

> **Note**
>
> Because Device Farm provides granular reporting in the standard environment, test execution times can be longer than when you run your tests locally. If you want faster execution times, run your tests in a custom test environment.

Custom test environment

When you customize the test environment, you can specify the commands Device Farm should run to execute your tests. This ensures that tests on Device Farm run in a way that is similar to tests run on your local machine. Running your tests in this mode also enables live log and video streaming of your tests. When you run tests in a customized test environment, you do not get granular reports for each test case. For more information, see Custom test environments in AWS Device Farm.

You have the option to use a custom test environment when you use the Device Farm console, AWS CLI, or Device Farm API to create a test run.

For more information, see Uploading a Custom Test Spec Using the AWS CLI and Creating a test run in Device Farm.

Runs in AWS Device Farm

The following sections contain information about runs in Device Farm.

A run in Device Farm represents a specific build of your app, with a specific set of tests, to be run on a specific set of devices. A run produces a report that contains information about the results of the run. A run contains one or more jobs.

Topics

Run configuration

As part of a run, you can supply settings Device Farm can use to override current device settings. These include latitude and longitude coordinates, locale, radio states (such as Bluetooth, GPS, NFC, and Wi-Fi), extra data (contained in a .zip file), and auxiliary apps (apps that should be installed before the app to be tested).

Run files retention

Device Farm stores your apps and files for 30 days and then deletes them from its system. You can delete your files at any time, however.

Device Farm stores your run results, logs, and screenshots for 400 days and then deletes them from its system.

Run device state

Device Farm always reboots a device before making it available for the next job.

Parallel runs

Device Farm runs tests in parallel as devices become available.

Setting the execution timeout

You can set a value for how long a test run should execute before you stop each device from running a test. For example, if your tests take 20 minutes per device to complete, you should choose a timeout of 30 minutes per device.

For more information, see Setting the execution timeout for test runs in AWS Device Farm.

Ads in runs

We recommend that you remove ads from your apps before you upload them to Device Farm. We cannot guarantee that ads are displayed during runs.

Media in runs

You can provide media or other data to accompany your app. Additional data must be provided in a .zip file no more than 4 GB in size.

Common tasks for runs

For more information, see Creating a test run in Device Farm and Test runs in AWS Device Farm.

Apps in AWS Device Farm

The following sections contain information about app behaviors in Device Farm.

Topics

Instrumenting apps

You do not need to instrument your apps or provide Device Farm with the source code for your apps. Android apps can be submitted unmodified. iOS apps must be built with the **iOS Device** target instead of with the simulator.

Re-signing apps in runs

For iOS apps, you do not need to add any Device Farm UUIDs to your provisioning profile. Device Farm replaces the embedded provisioning profile with a wildcard profile and then re-signs the app. If you provide auxiliary data, Device Farm adds it to the app's package before Device Farm installs it, so that the auxiliary exists in your app's sandbox. Re-signing the app removes entitlements such as App Group, Associated Domains, Game Center, HealthKit, HomeKit, Wireless Accessory Configuration, In-App Purchase, Inter-App Audio, Apple Pay, Push Notifications, and VPN Configuration & Control.

For Android apps, Device Farm re-signs the app. This might break any functionality that depends on the app's signature, such as the Google Maps Android API, or it might trigger antipiracy or antitamper detection from products such as DexGuard.

Obfuscated apps in runs

For Android apps, if the app is obfuscated, you can still test it with Device Farm if you use ProGuard. However, if you use DexGuard with antipiracy measures, Device Farm cannot re-sign and run tests against the app.

Reports in AWS Device Farm

The following sections provide information about Device Farm test reports.

Topics

Report retention

Device Farm stores your reports for 400 days. These reports include metadata, logs, screenshots, and performance data.

Report components

Reports in Device Farm contain pass and fail information, crash reports, test and device logs, screenshots, and performance data.

Reports include detailed per-device data and high-level results, such as the number of occurrences of a given problem.

Logs in reports

Reports include complete logcat captures for Android tests and complete Device Console logs for iOS tests.

Common tasks for reports

For more information, see Viewing test reports in Device Farm.

Sessions in AWS Device Farm

You can use Device Farm to perform interactive testing of Android and iOS apps through remote access sessions in a web browser. This kind of interactive testing helps support engineers on a customer call to walk through, step by step, the customer's issue. Developers can reproduce a problem on a specific device to isolate possible sources of the problem. You can use remote sessions to conduct usability tests with your target customers.

Topics

Supported devices for remote access

Device Farm provides support for a number of unique, popular Android and iOS devices. The list of available devices grows as new devices enter the market. The Device Farm console displays the current list of Android and iOS devices available for remote access. For more information, see Device support in AWS Device Farm.

Session files retention

Device Farm stores your apps and files for 30 days and then deletes them from its system. You can delete your files at any time, however.

Device Farm stores your session logs and captured video for 400 days and then deletes them from its system.

Instrumenting apps

You do not need to instrument your apps or provide Device Farm with the source code for your apps. Android and iOS apps can be submitted unmodified.

Re-signing apps in sessions

Device Farm re-signs Android and iOS apps. This can break functionality that depends on the app's signature. For example, the Google Maps API for Android depends on your app's signature. App re-signing can also trigger antipiracy or antitamper detection from products such as DexGuard for Android devices.

Obfuscated apps in sessions

For Android apps, if the app is obfuscated, you can still test it with Device Farm if you use ProGuard. However, if you use DexGuard with antipiracy measures, Device Farm cannot re-sign the app.

Projects in AWS Device Farm

A project in Device Farm represents a logical workspace in Device Farm that contains runs, one run for each test of a single app against one or more devices. Projects enable you to organize workspaces in whatever way you choose. For example, there can be one project per app title, or there can be one project per platform. You can create as many projects as you need.

You can use the AWS Device Farm console, AWS Command Line Interface (AWS CLI), or AWS Device Farm API to work with projects.

Topics

Creating a project in AWS Device Farm

You can create a project by using the AWS Device Farm console, AWS CLI, or AWS Device Farm API.

Prerequisites

- Complete the steps in Setting up.

Create a project (console)

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. Choose **New project**.
4. Enter a name for your project, then choose **Submit**.
5. To specify settings for the project, choose **Project settings**. These settings include the default timeout for test runs. After the settings are applied, they are used by all test runs for the project. For more information, see Setting the execution timeout for test runs in AWS Device Farm.

Create a project (AWS CLI)

- Run **create-project**, specifying the project name.

  Example:

  ```
  aws devicefarm create-project --name MyProjectName
  ```

  The AWS CLI response includes the Amazon Resource Name (ARN) of the project.

  ```
  {
      "project": {
          "name": "MyProjectName",
          "arn": "arn:aws:devicefarm:us-west-2:123456789101:project:5e01a8c7-
  c861-4c0a-b1d5-12345EXAMPLE",
          "created": 1535675814.414
      }
  }
  ```

  For more information, see **create-project** and AWS CLI reference.

Create a project (API)

- Call the `CreateProject` API.

For information about using the Device Farm API, see Automating Device Farm.

Viewing the projects list in AWS Device Farm

You can use the AWS Device Farm console, AWS CLI, or AWS Device Farm API to view the list of projects.

Topics

Prerequisites

- Create at least one project in Device Farm. Follow the instructions in Creating a project in AWS Device Farm, and then return to this page.

View the projects list (console)

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. To find the list of available projects, do the following:
   - For mobile device testing projects, on the Device Farm navigation menu, choose **Mobile Device Testing**, then choose **Projects**.
   - For desktop browser testing projects, on the Device Farm navigation menu, choose **Desktop Browser Testing**, then choose **Projects**.

View the projects list (AWS CLI)

- To view the projects list, run the **list-projects** command.

  To view information about a single project, run the **get-project** command.

For information about using Device Farm with the AWS CLI, see AWS CLI reference.

View the projects list (API)

- To view the projects list, call the `ListProjects` API.

  To view information about a single project, call the `GetProject` API.

For information about the AWS Device Farm API, see Automating Device Farm.

Test runs in AWS Device Farm

A run in Device Farm represents a specific build of your app, with a specific set of tests, to be run on a specific set of devices. A run produces a report that contains information about the results of the run. A run contains one or more jobs. For more information, see Runs.

You can use the AWS Device Farm console, AWS Command Line Interface (AWS CLI), or AWS Device Farm API to work with test runs.

Topics

Creating a test run in Device Farm

You can use the Device Farm console, AWS CLI, or Device Farm API to create a test run. You can also use a supported plugin, such as the Jenkins or Gradle plugins for Device Farm. For more information about plugins, see Tools and plugins. For information about runs, see Runs.

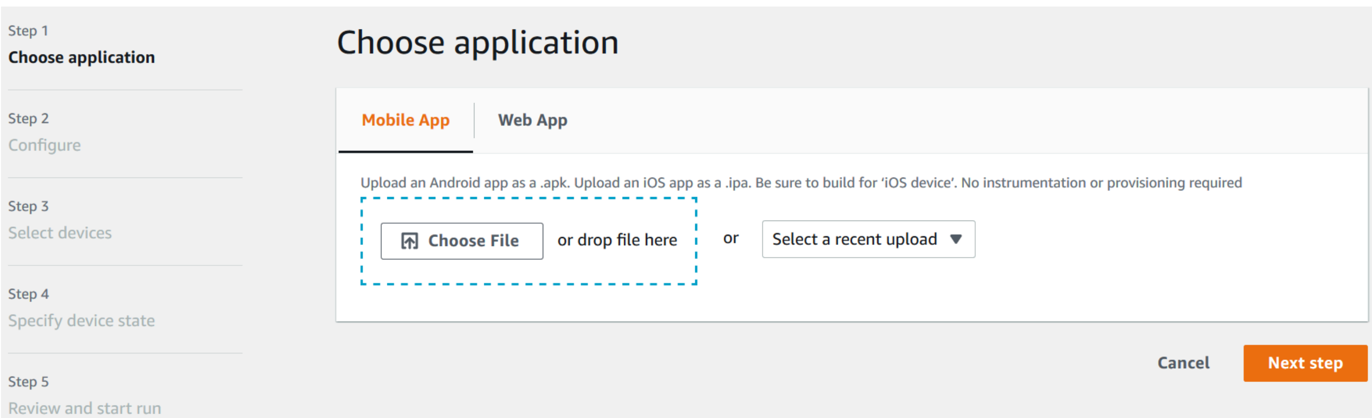
Topics

Prerequisites

You must have a project in Device Farm. Follow the instructions in Creating a project in AWS Device Farm, and then return to this page.

Create a test run (console)

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. In the navigation pane, choose **Mobile Device Testing**, and then choose **Projects**.
3. If you already have a project, you can upload your tests to it. Otherwise, choose **New project**, enter a **Project Name**, and then choose **Create**.
4. Open your project, and then choose **Create a new run**.
5. On the **Choose application** page, choose **Mobile App** or **Web App**.



6. Upload your application file. You can also drag and drop your file or choose a recent upload. If you're uploading an iOS app, be sure to choose **iOS device**, as opposed to a simulator.
7. (Optional) In **Run name**, enter a name. By default, Device Farm uses the app file name.
8. Choose **Next**.
9. On the **Configure** page, choose one of the available test suites.

> **Note**
>
> If you don't have any tests available, choose **Built-in: Fuzz** to run a standard, built-in test suite. If you choose **Built-in: Fuzz**, and the **Event count**, **Event throttle**, and **Randomizer seed** boxes appear, you can change or keep the values.

For information about the available test suites, see Test frameworks and built-in tests in AWS Device Farm.

10. If you didn't choose **Built-in: Fuzz**, select **Choose File**, and then browse to and choose the file that contains your tests.
11. For your testing environment, choose **Run your test in our standard environment** or **Run your test in a custom environment**. For more information, see Test environments in AWS Device Farm.
12. If you're using the standard test environment, skip to step 13. If you're using a custom test environment with the default test spec YAML file, skip to step 13.

    a. If you want to edit the default test spec in a custom test environment, choose **Edit** to update the default YAML specification.

    b. If you changed the test spec, choose **Save as New** to update it.
13. If you want to configure the video recording or performance data capture options, choose **Advanced Configuration**.

    a. Select **Enable video recording** to record video during testing.

    b. Select **Enable app performance data capture** to capture performance data from the device.

> **Note**
>
> If you have private devices, **Configuration specific to Private Devices** is also displayed.

14. Choose **Next**.
15. On the **Select devices** page, do one of the following:

    - To choose a built-in device pool to run the tests against, for **Device pool**, choose **Top Devices**.
    - To create your own device pool to run the tests against, follow the instructions in Creating a device pool, and then return to this page.
    - If you created your own device pool earlier, for **Device pool**, choose your device pool.

    For more information, see Device support in AWS Device Farm.

16. Choose **Next**.
17. On the **Specify device state** page:

    - To provide other data for Device Farm to use during the run, next to **Add extra data**, choose **Choose File**, and then browse to and choose the .zip file that contains the data.
    - To install an additional app for Device Farm to use during the run, next to **Install other apps**, choose **Choose File**, and then browse to and choose the .apk or .ipa file that contains the app. Repeat this for other apps you want to install. You can change the installation order by dragging and dropping the apps after you upload them.
    - To specify whether Wi-Fi, Bluetooth, GPS, or NFC is enabled during the run, next to **Set radio states**, select the appropriate boxes.
    - To preset the device latitude and longitude for the run, next to **Device location**, enter the coordinates.
    - To preset the device locale for the run, in **Device locale**, choose the locale.

    > **Note**
    >
    > Setting the device radio state and locale are options only available for Android native tests at this time.

18. Choose **Next**.
19. On the **Review and start run** page, you can specify the execution timeout for your test run. If you're using unlimited testing slots, confirm that **Run on unmetered slots** is selected.
20. Enter a value or use the slider bar to change the execution timeout. For more information, see Setting the execution timeout for test runs in AWS Device Farm.
21. Choose **Confirm and start run**.

Device Farm starts the run as soon as devices are available, typically within a few minutes. During your test run, the Device Farm console displays a pending icon

in the run table. Each device in the run will also start with the pending icon, then switch to the running icon when the test begins. As each test finishes, a test result icon is displayed next to the device name. When all tests have been completed, the pending icon next to the run changes to a test result icon.

If you want to stop the test run, see Stopping a run in AWS Device Farm.

Create a test run (AWS CLI)

You can use the AWS CLI to create a test run.

Topics

Step 1: Choose a project

You must associate your test run with a Device Farm project.

1. To list your Device Farm projects, run **list-projects**. If you do not have a project, see Creating a project in AWS Device Farm.

   Example:

   ```
   aws devicefarm list-projects
   ```

   The response includes a list of your Device Farm projects.

   ```
   {
       "projects": [
           {
               "name": "MyProject",
               "arn": "arn:aws:devicefarm:us-west-2:123456789101:project:5e01a8c7-
   c861-4c0a-b1d5-12345EXAMPLE",
               "created": 1503612890.057
           }
   ```

```
    ]
}
```

2. Choose a project to associate with your test run, and make a note of its Amazon Resource Name (ARN).

Step 2: Choose a device pool

You must choose a device pool to associate with your test run.

1. To view your device pools, run **list-device-pools**, specifying your project ARN.

   Example:

```
aws devicefarm list-device-pools --arn arn:MyProjectARN
```

   The response includes the built-in Device Farm device pools, such as **Top Devices**, and any device pools previously created for this project:


```
{
    "devicePools": [
        {
            "rules": [
                {
                    "attribute": "ARN",
                    "operator": "IN",
                    "value": "[\"arn:aws:devicefarm:us-west-2::device:example1\",
\"arn:aws:devicefarm:us-west-2::device:example2\",\"arn:aws:devicefarm:us-
west-2::device:example3\"]"
                }
            ],
            "type": "CURATED",
            "name": "Top Devices",
            "arn": "arn:aws:devicefarm:us-west-2::devicepool:example",
            "description": "Top devices"
        },
        {
            "rules": [
                {
                    "attribute": "PLATFORM",
                    "operator": "EQUALS",
                    "value": "\"ANDROID\""
```

```
                }
            ],
            "type": "PRIVATE",
            "name": "MyAndroidDevices",
            "arn": "arn:aws:devicefarm:us-west-2:605403973111:devicepool:example2"
        }
    ]
}
```

2. Choose a device pool, and make a note of its ARN.

   You can also create a device pool, and then return to this step. For more information, see [Create a device pool (AWS CLI)](#).

Step 3: Upload your application file

To create your upload request and get an Amazon Simple Storage Service (Amazon S3) presigned upload URL, you need:

- Your project ARN.
- The name of your app file.
- The type of the upload.

For more information, see **create-upload**.

1. To upload a file, run **create-upload** with the `--project-arn`, `--name`, and `--type` parameters.

   This example creates an upload for an Android app:

   ```
   aws devicefarm create-upload --project-arn arn:MyProjectArn --name MyAndroid.apk --
   type ANDROID_APP
   ```

   The response includes your app upload ARN and a presigned URL.

   ```
   {
       "upload": {
           "status": "INITIALIZED",
           "name": "MyAndroid.apk",
           "created": 1535732625.964,
   ```

```
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "ANDROID_APP",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE"
    }
}
```

2. Make a note of the app upload ARN and the presigned URL.
3. Upload your app file using the Amazon S3 presigned URL. This example uses **curl** to upload an Android .apk file:

```
curl -T MyAndroid.apk "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL"
```

   For more information, see Uploading objects using presigned URLs in the *Amazon Simple Storage Service User Guide*.
4. To check the status of your app upload, run **get-upload** and specify the ARN of the app upload.

```
aws devicefarm get-upload --arn arn:MyAppUploadARN
```

   Wait until the status in the response is **SUCCEEDED** before you upload your test scripts package.

```
{
    "upload": {
        "status": "SUCCEEDED",
        "name": "MyAndroid.apk",
        "created": 1535732625.964,
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "ANDROID_APP",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
        "metadata": "{"valid": true}"
    }
}
```

Step 4: Upload your test scripts package

Next, you upload your test scripts package.

1. To create your upload request and get an Amazon S3 presigned upload URL, run **create-upload** with the `--project-arn`, `--name`, and `--type` parameters.

   This example creates an Appium Java TestNG test package upload:

   ```
   aws devicefarm create-upload --project-arn arn:MyProjectARN --name MyTests.zip --
   type APPIUM_JAVA_TESTNG_TEST_PACKAGE
   ```

   The response includes your test package upload ARN and a presigned URL.

   ```
   {
       "upload": {
           "status": "INITIALIZED",
           "name": "MyTests.zip",
           "created": 1535738627.195,
           "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
   ExampleURL",
           "type": "APPIUM_JAVA_TESTNG_TEST_PACKAGE",
           "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
   c861-4c0a-b1d5-12345EXAMPLE"
       }
   }
   ```

2. Make a note of the ARN of the test package upload and the presigned URL.
3. Upload your test scripts package file using the Amazon S3 presigned URL. This example uses **curl** to upload a zipped Appium TestNG scripts file:

   ```
   curl -T MyTests.zip "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
   ExampleURL"
   ```

4. To check the status of your test scripts package upload, run **get-upload** and specify the ARN of the test package upload from step 1.

   ```
   aws devicefarm get-upload --arn arn:MyTestsUploadARN
   ```

Wait until the status in the response is **SUCCEEDED** before you continue to the next, optional step.

```
{
    "upload": {
        "status": "SUCCEEDED",
        "name": "MyTests.zip",
        "created": 1535738627.195,
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "APPIUM_JAVA_TESTNG_TEST_PACKAGE",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
        "metadata": "{"valid": true}"
    }
}
```

Step 5: (Optional) Upload your custom test spec

If you're running your tests in a standard test environment, skip this step.

Device Farm maintains a default test spec file for each supported test type. Next, you download your default test spec and use it to create a custom test spec upload for running your tests in a custom test environment. For more information, see Test environments in AWS Device Farm.

1. To find the upload ARN for your default test spec, run **list-uploads** and specify your project ARN.

```
aws devicefarm list-uploads --arn arn:MyProjectARN
```

The response contains an entry for each default test spec:

```
{
    "uploads": [
        {

            {
                "status": "SUCCEEDED",
                "name": "Default TestSpec for Android Appium Java TestNG",
                "created": 1529498177.474,
```

```
                "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
                "type": "APPIUM_JAVA_TESTNG_TEST_SPEC",
                "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE"
            }
        }
    ]
}
```

2. Choose your default test spec from the list. Make a note of its upload ARN.
3. To download your default test spec, run **get-upload** and specify the upload ARN.

   Example:

   ```
   aws devicefarm get-upload --arn arn:MyDefaultTestSpecARN
   ```

   The response contains a presigned URL where you can download your default test spec.
4. This example uses **curl** to download the default test spec and save it as `MyTestSpec.yml`:

   ```
   curl "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/ExampleURL" >
    MyTestSpec.yml
   ```
5. You can edit the default test spec to meet your testing requirements, and then use your modified test spec in future test runs. Skip this step to use the default test spec as-is in a custom test environment.
6. To create an upload of your custom test spec, run **create-upload**, specifying your test spec name, test spec type, and project ARN.

   This example creates an upload for an Appium Java TestNG custom test spec:

   ```
   aws devicefarm create-upload --name MyTestSpec.yml --type
    APPIUM_JAVA_TESTNG_TEST_SPEC --project-arn arn:MyProjectARN
   ```

   The response includes the test spec upload ARN and presigned URL:

   ```
   {
       "upload": {
           "status": "INITIALIZED",
           "category": "PRIVATE",
   ```

```
        "name": "MyTestSpec.yml",
        "created": 1535751101.221,
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "APPIUM_JAVA_TESTNG_TEST_SPEC",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE"
    }
}
```

7. Make a note of the ARN for the test spec upload and the presigned URL.
8. Upload your test spec file using the Amazon S3 presigned URL. This example uses **curl** to upload an Appium JavaTestNG test spec:

```
curl -T MyTestSpec.yml "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL"
```

9. To check the status of your test spec upload, run **get-upload** and specify the upload ARN.

```
aws devicefarm get-upload --arn arn:MyTestSpecUploadARN
```

Wait until the status in the response is **SUCCEEDED** before you schedule your test run.

```
{
    "upload": {
        "status": "SUCCEEDED",
        "name": "MyTestSpec.yml",
        "created": 1535732625.964,
        "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
        "type": "APPIUM_JAVA_TESTNG_TEST_SPEC",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
        "metadata": "{"valid": true}"
    }
}
```

To update your custom test spec, run **update-upload**, specifying the upload ARN for the test spec. For more information, see update-upload.

Step 6: Schedule a test run

To schedule a test run with the AWS CLI, run **schedule-run**, specifying:

- The project ARN from step 1.
- The device pool ARN from step 2.
- The app upload ARN from step 3.
- The test package upload ARN from step 4.

If you are running tests in a custom test environment, you also need your test spec ARN from step 5.

To schedule a run in a standard test environment

- Run **schedule-run**, specifying your project ARN, device pool ARN, application upload ARN, and test package information.

  Example:

  ```
  aws devicefarm schedule-run --project-arn arn:MyProjectARN --app-
  arn arn:MyAppUploadARN --device-pool-arn arn:MyDevicePoolARN --name MyTestRun --
  test type=APPIUM_JAVA_TESTNG,testPackageArn=arn:MyTestPackageARN
  ```

  The response contains a run ARN that you can use to check the status of your test run.

  ```
  {
      "run": {
          "status": "SCHEDULING",
          "appUpload": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
  c861-4c0a-b1d5-12345appEXAMPLE",
          "name": "MyTestRun",
          "radios": {
              "gps": true,
              "wifi": true,
              "nfc": true,
              "bluetooth": true
          },
          "created": 1535756712.946,
          "totalJobs": 179,
          "completedJobs": 0,
  ```

```
        "platform": "ANDROID_APP",
        "result": "PENDING",
        "devicePoolArn": "arn:aws:devicefarm:us-
west-2:123456789101:devicepool:5e01a8c7-c861-4c0a-b1d5-12345devicepoolEXAMPLE",
        "jobTimeoutMinutes": 150,
        "billingMethod": "METERED",
        "type": "APPIUM_JAVA_TESTNG",
        "testSpecArn": "arn:aws:devicefarm:us-west-2:123456789101:upload:5e01a8c7-
c861-4c0a-b1d5-12345specEXAMPLE",
        "arn": "arn:aws:devicefarm:us-west-2:123456789101:run:5e01a8c7-c861-4c0a-
b1d5-12345runEXAMPLE",
        "counters": {
            "skipped": 0,
            "warned": 0,
            "failed": 0,
            "stopped": 0,
            "passed": 0,
            "errored": 0,
            "total": 0
        }
    }
}
```

For more information, see **schedule-run**.

To schedule a run in a custom test environment

- The steps are the almost the same as those for the standard test environment, with an additional `testSpecArn` attribute in the `--test` parameter.

  Example:

```
aws devicefarm schedule-run --project-arn arn:MyProjectARN --app-
arn arn:MyAppUploadARN --device-pool-arn arn:MyDevicePoolARN --name MyTestRun --
test
 testSpecArn=arn:MyTestSpecUploadARN,type=APPIUM_JAVA_TESTNG,testPackageArn=arn:MyTestPacka
```

To check the status of your test run

- Use the **get-run** command and specify the run ARN:

```
aws devicefarm get-run --arn arn:aws:devicefarm:us-
west-2:111122223333:run:5e01a8c7-c861-4c0a-b1d5-12345runEXAMPLE
```

For more information, see **get-run**. For information about using Device Farm with the AWS CLI, see AWS CLI reference.

Create a test run (API)

The steps are the same as those described in the AWS CLI section. See Create a test run (AWS CLI).

You need this information to call the `ScheduleRun` API:

- A project ARN. See Create a project (API) and `CreateProject`.
- An application upload ARN. See `CreateUpload`.
- A test package upload ARN. See `CreateUpload`.
- A device pool ARN. See Creating a device pool and `CreateDevicePool`.

> **Note**
>
> If you're running tests in a custom test environment, you also need your test spec upload ARN. For more information, see Step 5: (Optional) Upload your custom test spec and `CreateUpload`.

For information about using the Device Farm API, see Automating Device Farm.

Next steps

In the Device Farm console, the clock icon

changes to a result icon such as success

when the run is complete. A report for the run appears as soon as tests are complete. For more information, see Reports in AWS Device Farm.

To use the report, follow the instructions in Viewing test reports in Device Farm.

Setting the execution timeout for test runs in AWS Device Farm

You can set a value for how long a test run should execute before you stop each device from running a test. The default execution timeout is 150 minutes per device, but you can set a value as low as 5 minutes. You can use the AWS Device Farm console, AWS CLI, or AWS Device Farm API to set the execution timeout.

> **Important**
>
> The execution timeout option should be set to the *maximum duration* for a test run, along with some buffer. For example, if your tests take 20 minutes per device, you should choose a timeout of 30 minutes per device.

If the execution exceeds your timeout, the execution on that device is forcibly stopped. Partial results are available, if possible. You are billed for execution up to that point, if you're using the metered billing option. For more information about pricing, see Device Farm Pricing.

You might want to use this feature if you know how long a test run is supposed to take to execute on each device. When you specify an execution timeout for a test run, you can avoid the situation where a test run is stuck for some reason and you are being billed for device minutes when no tests are being executed. In other words, using the execution timeout feature lets you stop that run if it's taking longer than expected.

You can set the execution timeout in two places, at the project level and the test run level.

Prerequisites

1. Complete the steps in Setting up.
2. Create a project in Device Farm. Follow the instructions in Creating a project in AWS Device Farm, and then return to this page.

Set the execution timeout for a project

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. If you already have a project, choose it from the list. Otherwise, choose **New project**, enter a name for your project, then choose **Submit**.

4. Choose **Project settings**.
5. On the **General** tab, for **Execution timeout**, enter a value or use the slider bar.
6. Choose **Save**.

   All the test runs in your project now use the execution timeout value that you specified, unless you override the timeout value when you schedule a run.

Set the execution timeout for a test run

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. If you already have a project, choose it from the list. Otherwise, choose **New project**, enter a name for your project, then choose **Submit**.
4. Choose **Create a new run**.
5. Follow the steps to choose an application, configure your test, select your devices, and specify a device state.
6. On **Review and start run**, for **Set execution timeout**, enter a value or use the slider bar.
7. Choose **Confirm and start run**.

Simulating network connections and conditions for your AWS Device Farm runs

You can use network shaping to simulate network connections and conditions while testing your Android, iOS, FireOS, and web apps in Device Farm. For example, you can test your app in less than perfect network conditions.

When you create a run using the default network settings, each device has a full, unhindered Wi-Fi connection with internet connectivity. When you use network shaping, you can change the Wi-Fi connection to specify a network profile like **3G** or **Lossy WiFi** that controls throughput, delay, jitter, and loss for both inbound and outbound traffic.

Topics

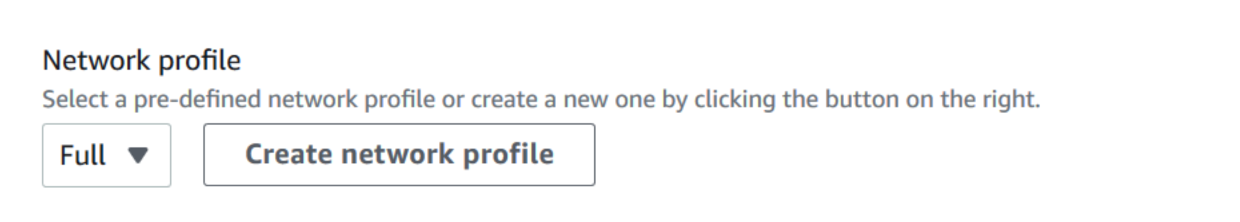
- Change network conditions during your test

Set up network shaping when scheduling a test run

When you schedule a run, you can choose from any of the Device Farm-curated profiles, or you can create and manage your own.

1. From any Device Farm project, choose **Create a new run**.

   If you don't have a project yet, see Creating a project in AWS Device Farm.
2. Choose your application, and then choose **Next**.
3. Configure your test, and then choose **Next**.
4. Select your devices, and then choose **Next**.
5. In the **Location and network settings** section, choose a network profile or choose **Create network profile** to create your own.

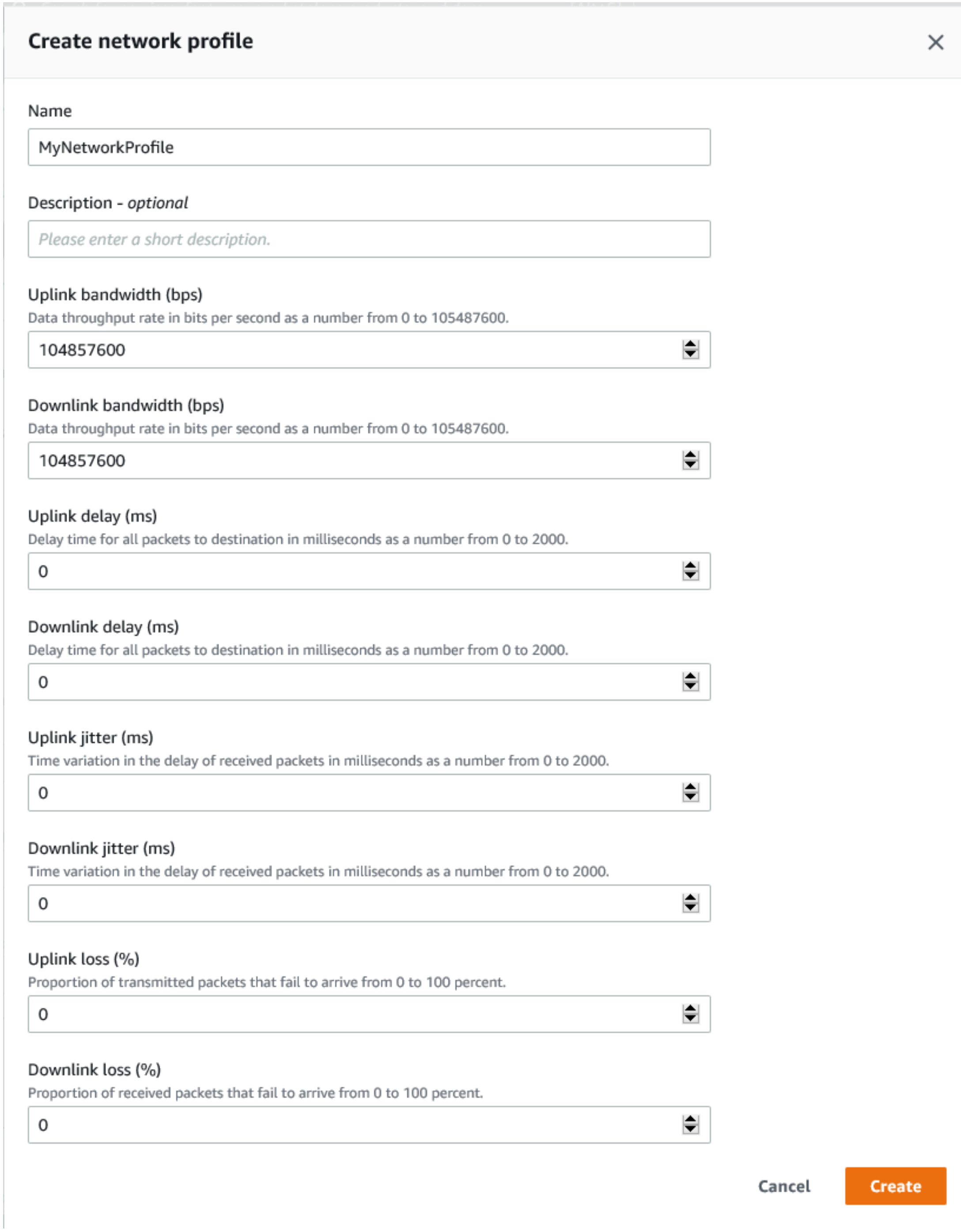


6. Choose **Next**.
7. Review and start your test run.

Create a network profile

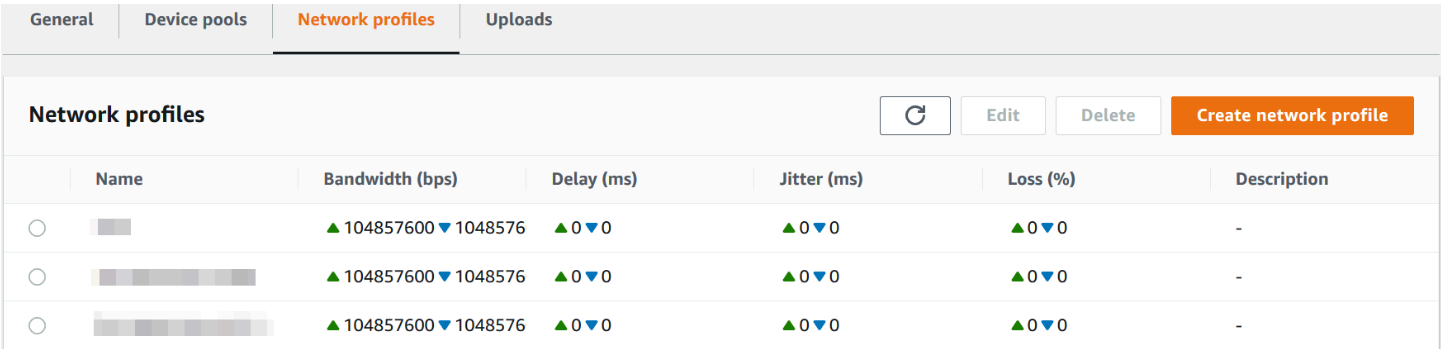
When you create a test run, you can create a network profile.

1. Choose **Create network profile**.



2. Enter a name and settings for your network profile.
3. Choose **Create**.
4. Finish creating your test run and start the run.

After you have created a network profile, you'll be able to see and manage it on the **Project settings** page.



Change network conditions during your test

You can call an API from your device host using a framework like Appium to simulate dynamic network conditions such as reduced bandwidth during your test run. For more information, see CreateNetworkProfile.

Stopping a run in AWS Device Farm

You might want to stop a run after you have started it. For example, if you notice an issue while your tests are running you might want to restart the run with an updated test script.

You can use the Device Farm console, AWS CLI, or API to stop a run.

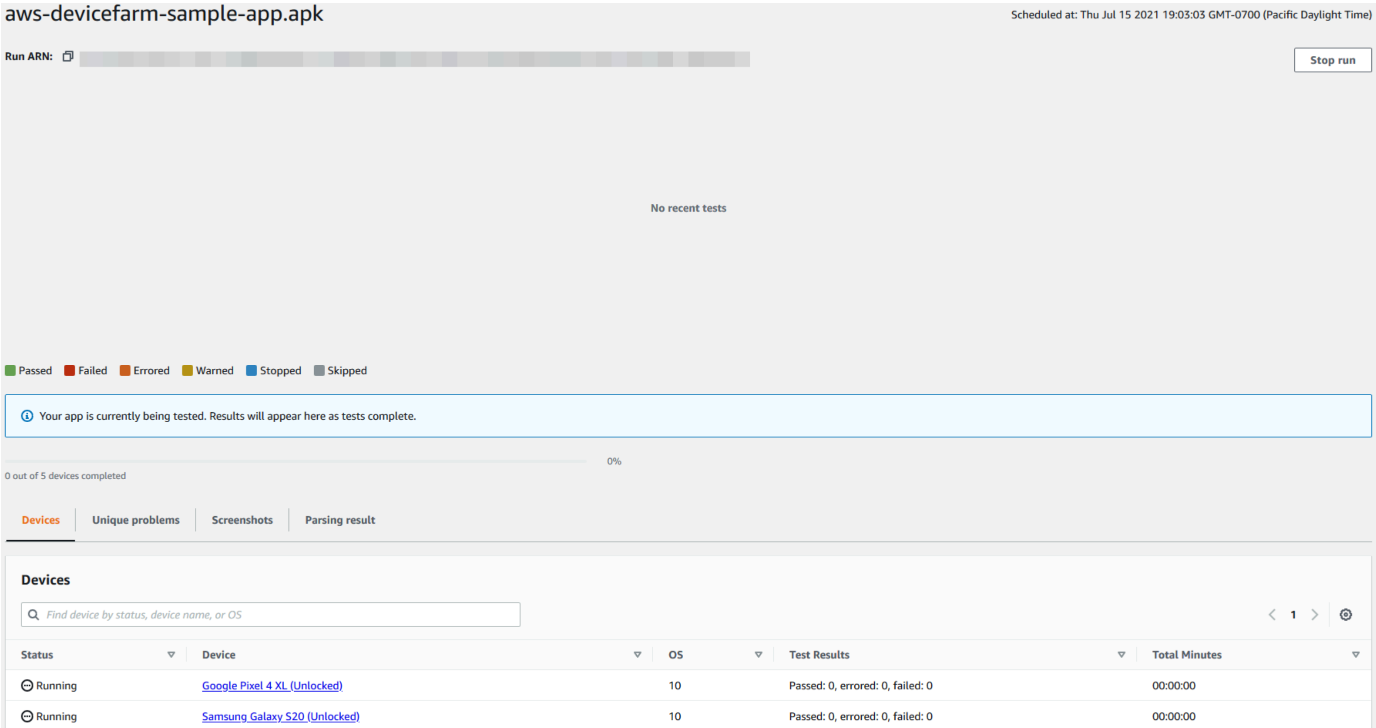
Topics

- Stop a run (console)
- Stop a run (AWS CLI)
- Stop a run (API)

Stop a run (console)

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. Choose the project where you have an active test run.
4. On the **Automated tests** page, choose the test run.

   The pending or running icon should appear to the left of the device name.

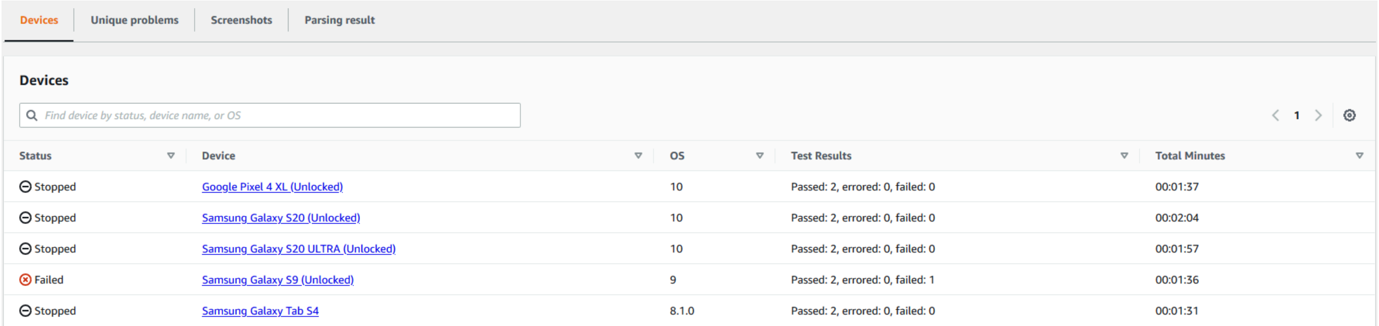


5. Choose **Stop run**.

   After a short time, an icon with a red circle with a minus inside it appears next to the device name. When the run has been stopped, the icon color changes from red to black.

   > **Important**
   >
   > If a test has already been run, Device Farm cannot stop it. If a test is in progress, Device Farm stops the test. The total minutes for which you will be billed appears in the **Devices** section. In addition, you will also be billed for the total minutes that Device Farm takes to run the setup suite and the teardown suite. For more information, see [Device Farm Pricing](https://aws.amazon.com/device-farm/faq/).

   The following image shows an example **Devices** section after a test run was successfully stopped.



Stop a run (AWS CLI)

You can run the following command to stop the specified test run, where *myARN* is the Amazon Resource Name (ARN) of the test run.

```
$ aws devicefarm stop-run --arn myARN
```

You should see output similar to the following:

```
{
    "run": {
        "status": "STOPPING",
        "name": "Name of your run",
        "created": 1458329687.951,
        "totalJobs": 7,
        "completedJobs": 5,
        "deviceMinutes": {
            "unmetered": 0.0,
            "total": 0.0,
            "metered": 0.0
        },
        "platform": "ANDROID_APP",
        "result": "PENDING",
        "billingMethod": "METERED",
        "type": "BUILTIN_EXPLORER",
        "arn": "myARN",
        "counters": {
            "skipped": 0,
            "warned": 0,
            "failed": 0,
            "stopped": 0,
            "passed": 0,
```

```
            "errored": 0,
            "total": 0
        }
    }
}
```

To get the ARN of your run, use the `list-runs` command. The output should be similar to the following:

```
{
    "runs": [
        {
            "status": "RUNNING",
            "name": "Name of your run",
            "created": 1458329687.951,
            "totalJobs": 7,
            "completedJobs": 5,
            "deviceMinutes": {
                "unmetered": 0.0,
                "total": 0.0,
                "metered": 0.0
            },
            "platform": "ANDROID_APP",
            "result": "PENDING",
            "billingMethod": "METERED",
            "type": "BUILTIN_EXPLORER",
            "arn": "Your ARN will be here",
            "counters": {
                "skipped": 0,
                "warned": 0,
                "failed": 0,
                "stopped": 0,
                "passed": 0,
                "errored": 0,
                "total": 0
            }
        }
    ]
}
```

For information about using Device Farm with the AWS CLI, see AWS CLI reference.

Stop a run (API)

- Call the StopRun operation to the test run.

For information about using the Device Farm API, see Automating Device Farm.

Viewing a list of runs in AWS Device Farm

You can use the Device Farm console, AWS CLI, or API to view a list of runs for a project.

Topics

View a list of runs (console)

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. In the list of projects, choose the project that corresponds to the list you want to view.

> **Tip**
>
> You can use the search bar to filter the project list by name.

View a list of runs (AWS CLI)

- Run the **list-runs** command.

  To view information about a single run, run the **get-run** command.

For information about using Device Farm with the AWS CLI, see AWS CLI reference.

View a list of runs (API)

- Call the `ListRuns` API.

  To view information about a single run, call the `GetRun` API.

For information about the Device Farm API, see Automating Device Farm.

Creating a device pool in AWS Device Farm

You can use the Device Farm console, AWS CLI, or API to create a device pool.

Topics

Prerequisites

- Create a run in the Device Farm console. Follow the instructions in Creating a test run in Device Farm. When you get to the **Select devices** page, continue with the instructions in this section.

Create a device pool (console)

1. On the **Projects** page, choose your project. In the **Project details** page, choose **Project settings**. In the **Device pools** tab, choose **Create device pool**.
2. For **Name**, enter a name that makes this device pool easy to identify.
3. For **Description**, enter a description that makes this device pool easy to identify.
4. If you want to use one or more selection criteria for the devices in this device pool, do the following:

   a. Choose **Create dynamic device pool**.

   b. Choose **Add a rule**.

   c. For **Field** (first drop-down list), choose one of the following:

- To include devices by their manufacturer name, choose **Device Manufacturer**.
- To include devices by their form factor (tablet or phone), choose **Form Factor**.
- To include devices by their availability status based on load, choose **Availability**.
- To include only public or private devices, choose **Fleet Type**.
- To include devices by their operating system, choose **Platform**.
- Some devices have an additional label tag or description about the device. You can find devices based on their label contents by choosing **Instance labels**.
- To include devices by their operating system version, choose **OS Version**.
- To include devices by their model, choose **Model**.

d. For **Operator** (second drop-down list), choose a logical operation (EQUALS, CONTAINS, etc.) to include devices based on the query. For example, you could choose *`Availability EQUALS AVAILABLE`* to include devices that currently have the `Available` status.

e. For **Value** (third drop-down list), enter or choose the value you want to specify for the **Field** and **Operator** values. Values are limited based on your **Field** choice. For example, if you choose **Platform** for **Field**, the only available selections are **ANDROID** and **IOS**. Similarly, if you choose **Form Factor** for **Field**, the only available selections are **PHONE** and **TABLET**.

f. To add another rule, choose **Add a rule**.

   After you create the first rule, in the list of devices, the box next to each device that matches the rule is selected. After you create or change rules, in the list of devices, the box next to each device that matches those combined rules is selected. Devices with selected boxes are included in the device pool. Devices with cleared boxes are excluded.

g. Under **Max devices**, enter the number of devices you want to use in your device pool. If you do not enter the max number of devices, Device Farm will pick all devices in the fleet that match the rule(s) you created. To avoid additional charges, set this number to an amount that matches your actual parallel execution and device variety requirements.

h. To delete a rule, choose **Remove rule**.

5. If you want to manually include or exclude individual devices, do the following:

   a. Choose **Create static device pool**.

   b. Select or clear the box next to each device. You can select or clear the boxes only if you do not have any rules specified.

6. If you want to include or exclude all displayed devices, select or clear the box in the column header row of the list. If you want to view private device instances only, choose **See private device instances only**.

> **Important**
>
> Although you can use the boxes in the column header row to change the list of displayed devices, this does not mean that the remaining displayed devices are the only ones included or excluded. To confirm which devices are included or excluded, be sure to clear the contents of all of the boxes in the column header row, and then browse the boxes.

7. Choose **Create**.

Create a device pool (AWS CLI)

> **Tip**
>
> If you do not enter the max number of devices, Device Farm will pick all devices in the fleet that match the rule(s) you created. To avoid additional charges, set this number to an amount that matches your actual parallel execution and device variety requirements.

- Run the **create-device-pool** command.

For information about using Device Farm with the AWS CLI, see AWS CLI reference.

Create a device pool (API)

> **Tip**
>
> If you do not enter the max number of devices, Device Farm will pick all devices in the fleet that match the rule(s) you created. To avoid additional charges, set this number to an amount that matches your actual parallel execution and device variety requirements.

- Call the `CreateDevicePool` API.

For information about using the Device Farm API, see Automating Device Farm.

Analyzing test results in AWS Device Farm

In the standard test environment, you can use the Device Farm console to view reports for each test in your test run. Vewing the reports helps you understand which tests passed or failed, and provides you with details on the performance and behavior of your app across different device configurations.

Device Farm also gathers other artifacts such as files, logs, and images that you can download when your test run is complete. This information can help you analyze how your app is behaving on real devices, identify issues or bugs, and diagnose problems.

Topics

- Viewing test reports in Device Farm
- Downloading artifacts in Device Farm

Viewing test reports in Device Farm

Use the Device Farm console to view your test reports. For more information, see Reports in AWS Device Farm.

Topics

- Prerequisites
- View reports
- Device Farm test result statuses

Prerequisites

Set up a test run and verify that it is complete.

1. To create a run, see Creating a test run in Device Farm, and then return to this page.
2. Verify that the run is complete. During your test run, the Device Farm console displays a pending icon

   for runs that are in progress. Each device in the run will also start with the pending icon, then switch to the running

icon when the test begins. As each test finishes, a test result icon is displayed next to the device name. When all tests have been completed, the pending icon next to the run changes to a test result icon. For more information, see [Device Farm test result statuses](#) .

View reports

You can view the results of your test in the Device Farm console.

Topics

- View the test run summary page
- View unique problem reports
- View device reports
- View test suite reports
- View test reports
- View performance data for a problem, device, suite, or test in a report
- View log information for a problem, device, suite, or test in a report

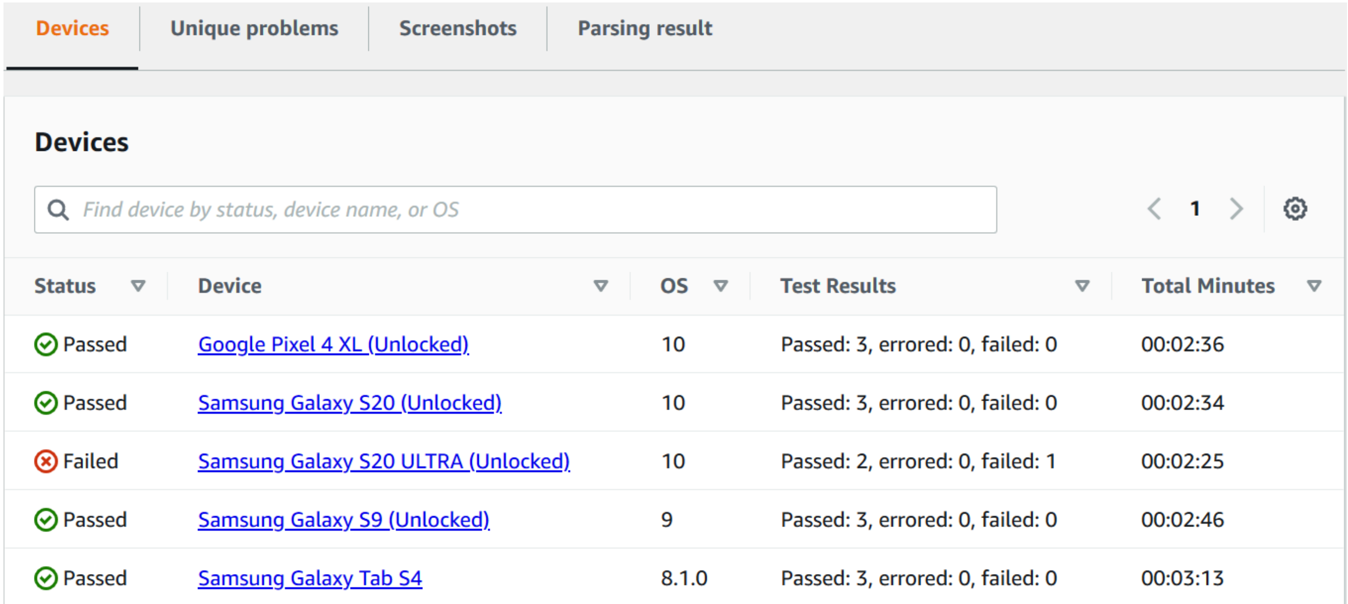
View the test run summary page

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. In the navigation pane, choose **Mobile Device Testing**, and then choose **Projects**.
3. In the list of projects, choose the project for the run.

> **Tip**
>
> To filter the project list by name, use the search bar.

4. Choose a completed run to view its summary report page.
5. The test run summary page displays an overview of your test results.

    - The **Unique problems** section lists unique warnings and failures. To view unique problems, follow the instructions in View unique problem reports.
    - The **Devices** section displays the total number of tests, by outcome, for each device.



In this example, there are several devices. In the first table entry, the Google Pixel 4 XL device running Android version 10 reports three successful tests that took 02:36 minutes to run.

To view the results by device, follow the instructions in View device reports.

- The **Screenshots** section displays a list of any screenshots that Device Farm captured during the run, grouped by device.
- In the **Parsing result** section, you can download the parsing result.

View unique problem reports

1. In **Unique problems**, choose the problem that you want to view.
2. Choose the device. The report displays information about the problem.

   The **Video** section displays a downloadable video recording of the test.

   The **Result** section displays the result of the test. The status is represented as a result icon. For more information, see Statuses of an individual test.

   The **Logs** section displays any information that Device Farm logged during the test. To view this information, follow the instructions in View log information for a problem, device, suite, or test in a report.

The **Performance** tab displays information about any performance data that Device Farm generated during the test. To view this performance data, follow the instructions in View performance data for a problem, device, suite, or test in a report.

The **Files** tab displays a list of any of the test's associated files (such as log files) that you can download. To download a file, choose the file's link in the list.

The **Screenshots** tab displays a list of any screenshots that Device Farm captured during the test.

View device reports

- In the **Devices** section, choose the device.

  The **Video** section displays a downloadable video recording of the test.

  The **Suites** section displays a table containing information about the suites for the device.

  In this table, the **Test results** column summarizes the number of tests by outcome for each of the test suites that have run on the device. This data also has a graphical component. For more information, see Statuses for multiple tests.

  To view the full results by suite, follow the instructions in View test suite reports.

  The **Logs** section displays any information that Device Farm logged for the device during the run. To view this information, follow the instructions in View log information for a problem, device, suite, or test in a report.

  The **Performance** section displays information about any performance data that Device Farm generated for the device during the run. To view this performance data, follow the instructions in View performance data for a problem, device, suite, or test in a report.

  The **Files** section displays a list of suites for the device and any associated files (such as log files) that you can download. To download a file, choose the file's link in the list.

  The **Screenshots** section displays a list of any screenshots that Device Farm captured during the run for the device, grouped by suite.

View test suite reports

1. In the **Devices** section, choose the device.
2. In the **Suites** section, choose the suite from the table.

   The **Video** section displays a downloadable video recording of the test.

   The **Tests** section displays a table containing information about the tests in the suite.

   In the table, the **Test results** column displays the result. This data also has a graphical component. For more information, see Statuses for multiple tests.

   To view the full results by test, follow the instructions in View test reports.

   The **Logs** section displays any information that Device Farm logged during the run for the suite. To view this information, follow the instructions in View log information for a problem, device, suite, or test in a report.

   The **Performance** section displays information about any performance data that Device Farm generated during the run for the suite. To view this performance data, follow the instructions in View performance data for a problem, device, suite, or test in a report.

   The **Files** section displays a list of tests for the suite and any associated files (such as log files) that you can download. To download a file, choose the file's link in the list.

   The **Screenshots** section displays a list of any screenshots that Device Farm captured during the run for the suite, grouped by test.

View test reports

1. In the **Devices** section, choose the device.
2. In the **Suites** section, choose the suite.
3. In the **Tests** section, choose the test.
4. The **Video** section displays a downloadable video recording of the test.

   The **Result** section displays the result of the test. The status is represented as a result icon. For more information, see Statuses of an individual test.

The **Logs** section displays any information that Device Farm logged during the test. To view this information, follow the instructions in View log information for a problem, device, suite, or test in a report.

The **Performance** tab displays information about any performance data that Device Farm generated during the test. To view this performance data, follow the instructions in View performance data for a problem, device, suite, or test in a report.

The **Files** tab displays a list of any of the test's associated files (such as log files) that you can download. To download a file, choose the file's link in the list.

The **Screenshots** tab displays a list of any screenshots that Device Farm captured during the test.

View performance data for a problem, device, suite, or test in a report

> **Note**
>
> Device Farm collects device performance data only for legacy Android test hosts that don't use the latest `amazon_linux_2` test host. This feature is not supported on iOS.

The **Performance** tab displays the following information:

- The **CPU** graph displays the percentage of CPU that the app used on a single core during the selected problem, device, suite, or test (along the vertical axis) over time (along the horizontal axis).

  The vertical axis is expressed in percentages, from 0% to the maximum recorded percentage.

  This percentage might exceed 100% if the app used more than one core. For example, if three cores are at 60% usage, this percentage is displayed as 180%.
- The **Memory** graph displays the number of MB that the app used during the selected problem, device, suite, or test (along the vertical axis) over time (along the horizontal axis).

  The vertical axis is expressed in MB, from 0 MB to the maximum number of recorded MB.
- The **Threads** graph displays the number of threads used during the selected problem, device, suite, or test (along the vertical axis) over time (along the horizontal axis).

The vertical axis is expressed in number of threads, from zero threads to the maximum number of recorded threads.

In all cases, the horizontal axis is represented, in seconds, from the start and end of the run for the selected problem, device, suite, or test.

To display information for a specific data point, pause in the desired graph at the desired second along the horizontal axis.

View log information for a problem, device, suite, or test in a report

The **Logs** section displays the following information:

- **Source** represents the source of a log entry. Possible values include:
  - **Harness** represents a log entry that Device Farm created. These log entries are typically created during start and stop events.
  - **Device** represents a log entry that the device created. For Android, these log entries are logcat-compatible. For iOS, these log entries are syslog-compatible.
  - **Test** represents a log entry that either a test or its test framework created.
- **Time** represents the elapsed time between the first log entry and this log entry. The time is expressed in *`MM:SS.SSS`* format, where *`M`* represents minutes and *`S`* represents seconds.
- **PID** represents the process identifier (PID) that created the log entry. All log entries created by an app on a device have the same PID.
- **Level** represents the logging level for the log entry. For example, `Logger.debug("This is a message!")` logs a **Level** of `Debug`. These are the possible values:
  - **Alert**
  - **Critical**
  - **Debug**
  - **Emergency**
  - **Error**
  - **Errored**
  - **Failed**
  - **Info**
  - **Internal**

  - **Notice**
  - **Passed**
  - **Skipped**
  - **Stopped**
  - **Verbose**
  - **Warned**
  - **Warning**
- **Tag** represents arbitrary metadata for the log entry. For example, Android logcat can use this to describe which part of the system created the log entry (for example, `ActivityManager`).
- **Message** represents the message or data for the log entry. For example, `Logger.debug("Hello, World!")` logs a **Message** of `"Hello, World!"`.

To display only a portion of the information:

- To show all log entries that match a value for a specific column, enter the value into the search bar. For example, to show all log entries with a **Source** value of `Harness`, enter **`Harness`** in the search bar.
- To remove all of the characters from a column header box, choose the **X** in that column header box. Removing all of the characters from a column header box is the same as entering ***** in that column header box.

To download all of the log information for the device, including all of the suites and tests that you ran, choose **Download logs**.

Device Farm test result statuses

The Device Farm console displays icons that help you quickly assess the state of your completed test run. For more information about tests in Device Farm, see Reports in AWS Device Farm.

Topics

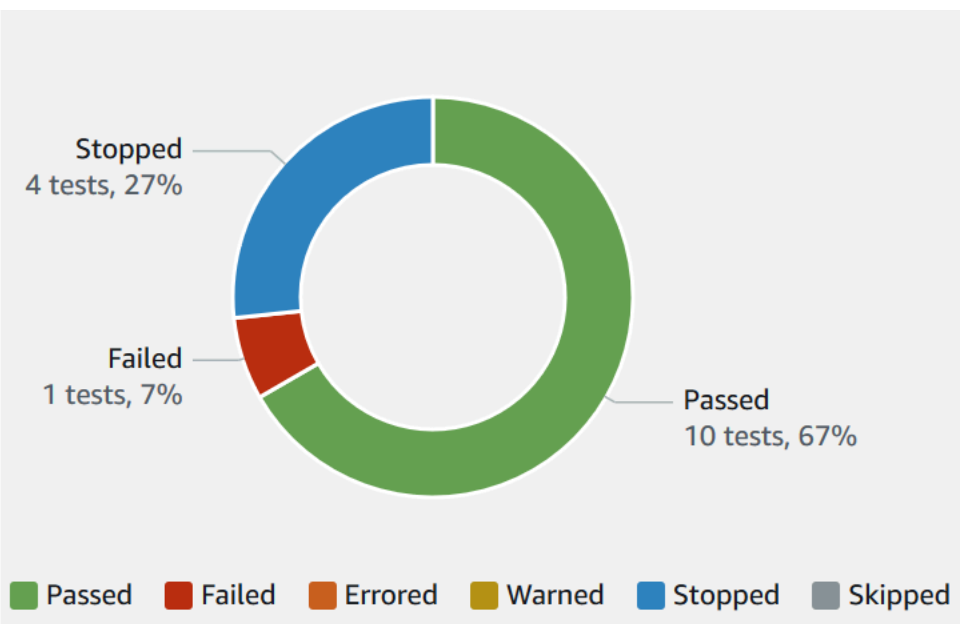
Statuses of an individual test

For reports that describe an individual test, Device Farm displays an icon representing the test result status:

Description	Icon
The test succeeded.	
The test failed.	
Device Farm skipped the test.	
The test stopped.	
Device Farm returned a warning.	
Device Farm returned an error.	

Statuses for multiple tests

If you choose a finished run, Device Farm displays a summary graph showing the percentage of tests in various states.



For example, this test run results graph shows that the run had 4 stopped tests, 1 failed test, and 10 successful tests.

Graphs are always color coded and labeled.

Downloading artifacts in Device Farm

Device Farm gathers artifacts such as reports, log files, and images for each test in the run.

You can download artifacts created during your test run:

Files

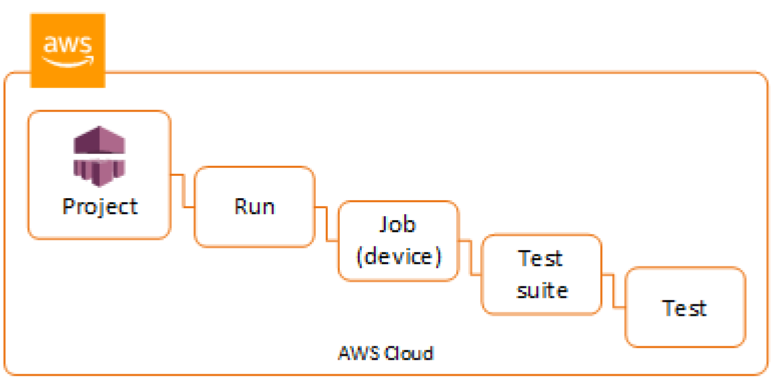
Files generated during the test run including Device Farm reports. For more information, see Viewing test reports in Device Farm.

Logs

Output from each test in the test run.

Screenshots

Screen images recorded for each test in the test run.



Download artifacts (console)

1. On the test run report page, from **Devices**, choose a mobile device.
2. To download a file, choose one from **Files**.
3. To download the logs from your test run, from **Logs**, choose **Download logs**.
4. To download a screenshot, choose a screenshot from **Screenshots**.

For more information about downloading artifacts in a custom test environment, see Downloading artifacts in a custom test environment.

Download artifacts (AWS CLI)

You can use the AWS CLI to list your test run artifacts.

Topics

Step 1: Get your Amazon Resource Names (ARN)

You can list your artifacts by run, job, test suite, or test. You need the corresponding ARN. This table shows the input ARN for each of the AWS CLI list commands:

AWS CLI List Command	Required ARN
list-projects	This command returns all projects and does not require an ARN.
list-runs	`project`
list-jobs	`run`
list-suites	`job`
list-tests	`suite`

For example, to find a test ARN, run **list-tests** using your test suite ARN as an input parameter.

Example:

```
aws devicefarm list-tests --arn arn:MyTestSuiteARN
```

The response includes a test ARN for each test in the test suite.

```
{
```

```
    "tests": [
        {
            "status": "COMPLETED",
            "name": "Tests.FixturesTest.testExample",
            "created": 1537563725.116,
            "deviceMinutes": {
                "unmetered": 0.0,
                "total": 1.89,
                "metered": 1.89
            },
            "result": "PASSED",
            "message": "testExample passed",
            "arn": "arn:aws:devicefarm:us-west-2:123456789101:test:5e01a8c7-c861-4c0a-
b1d5-12345EXAMPLE",
            "counters": {
                "skipped": 0,
                "warned": 0,
                "failed": 0,
                "stopped": 0,
                "passed": 1,
                "errored": 0,
                "total": 1
            }
        }
    ]
}
```

Step 2: List your artifacts

The AWS CLI list-artifacts command returns a list of artifacts, such as files, screenshots, and logs. Each artifact has a URL so you can download the file.

- Call **list-artifacts** specifying a run, job, test suite, or test ARN. Specify a type of FILE, LOG, or SCREENSHOT.

  This example returns a download URL for each artifact available for an individual test:

  ```
  aws devicefarm list-artifacts --arn arn:MyTestARN --type "FILE"
  ```

  The response contains a download URL for each artifact.

  ```
  {
  ```

```
    "artifacts": [
        {
            "url": "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/
ExampleURL",
            "extension": "txt",
            "type": "APPIUM_JAVA_OUTPUT",
            "name": "Appium Java Output",
            "arn": "arn:aws:devicefarm:us-west-2:123456789101:artifact:5e01a8c7-
c861-4c0a-b1d5-12345EXAMPLE",
        }
    ]
}
```

Step 3: Download your artifacts

- Download your artifact using the URL from the previous step. This example uses **curl** to download an Android Appium Java output file:

```
curl "https://prod-us-west-2-uploads.s3-us-west-2.amazonaws.com/ExampleURL"
 > MyArtifactName.txt
```

Download artifacts (API)

The Device Farm API ListArtifacts method returns a list of artifacts, such as files, screenshots, and logs. Each artifact has a URL so you can download the file.

Downloading artifacts in a custom test environment

In a custom test environment, Device Farm gathers artifacts such as custom reports, log files, and images. These artifacts are available for each device in the test run.

You can download these artifacts created during your test run:

Test spec output

The output from running the commands in the test spec YAML file.

Customer artifacts

A zipped file that contains the artifacts from the test run. It is configured in the **artifacts:** section of your test spec YAML file.

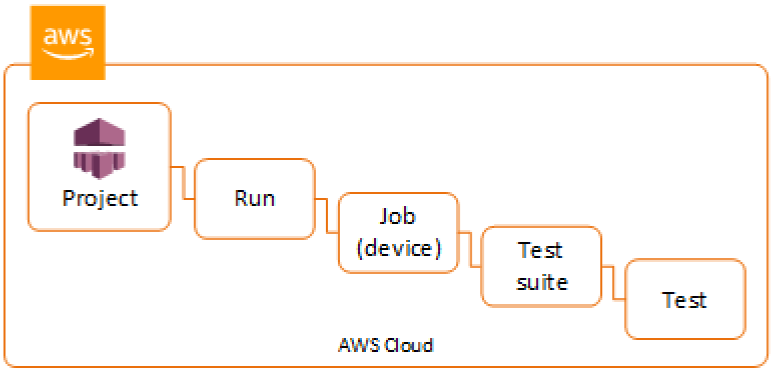
Test spec shell script

An intermediate shell script file created from your YAML file. Because it is used in the test run, the shell script file can be used for debugging the YAML file.

Test spec file

The YAML file used in the test run.

For more information, see Downloading artifacts in Device Farm.



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

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 `resourcegrouptaggingapi` 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](https://docs.aws.amazon.com/resourcegroupstagging/latest/APIReference/).

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"])
```

Test frameworks and built-in tests in AWS Device Farm

This section describes Device Farm support for testing frameworks and built-in test types.

For more information about how Device Farm runs tests, see Test environments in AWS Device Farm.

Testing frameworks

Device Farm supports these mobile automation testing frameworks:

Android application testing frameworks

- Appium
- Instrumentation

iOS application testing frameworks

- Appium
- XCTest
- XCTest UI

Web application testing frameworks

Web applications are supported using Appium. For more information on bringing your tests to Appium, see Appium tests and AWS Device Farm.

Frameworks in a custom test environment

Device Farm does not provide support for customizing the test environment for the XCTest framework. For more information, see Custom test environments in AWS Device Farm.

Appium version support

For tests running in a custom environment, Device Farm supports Appium version 1. For more information, see Test environments in AWS Device Farm.

Built-in test types

With built-in tests, you can test your application on multiple devices without having to write and maintain test automation scripts. Device Farm offers one built-in test type:

- Built-in: fuzz (Android and iOS)

Appium tests and AWS Device Farm

This section describes how to configure, package, and upload your Appium tests to Device Farm. Appium is an open source tool for automating native and mobile web applications. For more information, see Introduction to Appium on the Appium website.

For a sample app and links to working tests, see Device Farm Sample App for Android and Device Farm Sample App for iOS on GitHub.

For more information about testing in Device Farm, see Test frameworks and built-in tests in AWS Device Farm.

Version support

Support for various frameworks and programming languages depends on the language used.

Device Farm supports all Appium 1.x and 2.x server versions. For Android, you can choose any major Appium version with `devicefarm-cli`. For example, to use Appium server version 2, add these commands to your test spec YAML file:

```
phases:
  install:
    commands:
      # To install a newer version of Appium such as version 2:
      - export APPIUM_VERSION=2
      - devicefarm-cli use appium $APPIUM_VERSION
```

For iOS, you can choose specific Appium versions with the `avm` or `npm` commands. For example, to use the `avm` command to set the Appium server version to 2.1.2, add these commands to your test spec YAML file:

```
phases:
  install:
```

```
    commands:
      # To install a newer version of Appium such as version 2.1.2:
      - export APPIUM_VERSION=2.1.2
      - avm $APPIUM_VERSION
```

Using the npm command to use the latest version of Appium 2, add these commands to your test spec YAML file:

```
phases:
  install:
    commands:
      - export APPIUM_VERSION=2
      - npm install -g appium@$APPIUM_VERSION
```

For more information about `devicefarm-cli` or any other CLI commands, see the AWS CLI reference.

To use all the features of the framework, like annotations, choose a custom test environment, and use the AWS CLI or the `Device Farm` console to upload a custom test spec.

Integrating Appium tests with Device Farm

Use the following instructions to integrate Appium tests with AWS Device Farm. For more information about using Appium tests in Device Farm, see Appium tests and AWS Device Farm.

Configure your Appium test package

Use the following instructions to configure your test package.

Java (JUnit)

1. Modify `pom.xml` to set packaging to a JAR file:

   ```
   <groupId>com.acme</groupId>
   <artifactId>acme-myApp-appium</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>jar</packaging>
   ```

2. Modify `pom.xml` to use `maven-jar-plugin` to build your tests into a JAR file.

   The following plugin builds your test source code (anything in the `src/test` directory) into a JAR file:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.6</version>
  <executions>
    <execution>
      <goals>
        <goal>test-jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

3. Modify `pom.xml` to use `maven-dependency-plugin` to build dependencies as JAR files.

   The following plugin copies your dependencies into the `dependency-jars` directory:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>2.10</version>
  <executions>
    <execution>
      <id>copy-dependencies</id>
      <phase>package</phase>
      <goals>
        <goal>copy-dependencies</goal>
      </goals>
      <configuration>
        <outputDirectory>${project.build.directory}/dependency-jars/</
outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

4. Save the following XML assembly to `src/main/assembly/zip.xml`.

   The following XML is an assembly definition that, when configured, instructs Maven to build a .zip file that contains everything in the root of your build output directory and the `dependency-jars` directory:

```
<assembly
    xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/
assembly/1.1.0 http://maven.apache.org/xsd/assembly-1.1.0.xsd">
  <id>zip</id>
  <formats>
    <format>zip</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory>./</outputDirectory>
      <includes>
        <include>*.jar</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory>./</outputDirectory>
      <includes>
        <include>/dependency-jars/</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```

5. Modify `pom.xml` to use `maven-assembly-plugin` to package tests and all dependencies into a single .zip file.

   The following plugin uses the preceding assembly to create a .zip file named `zip-with-dependencies` in the build output directory every time **mvn package** is run:

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.5.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
```

```
        </goals>
        <configuration>
          <finalName>zip-with-dependencies</finalName>
          <appendAssemblyId>false</appendAssemblyId>
          <descriptors>
            <descriptor>src/main/assembly/zip.xml</descriptor>
          </descriptors>
        </configuration>
      </execution>
    </executions>
</plugin>
```

Note

If you receive an error that says annotation is not supported in 1.3, add the following to `pom.xml`:

```
<plugin>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.7</source>
    <target>1.7</target>
  </configuration>
</plugin>
```

Java (TestNG)

1. Modify `pom.xml` to set packaging to a JAR file:

```
<groupId>com.acme</groupId>
<artifactId>acme-myApp-appium</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
```

2. Modify `pom.xml` to use `maven-jar-plugin` to build your tests into a JAR file.

   The following plugin builds your test source code (anything in the `src/test` directory) into a JAR file:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.6</version>
  <executions>
    <execution>
      <goals>
        <goal>test-jar</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

3. Modify `pom.xml` to use `maven-dependency-plugin` to build dependencies as JAR files.

   The following plugin copies your dependencies into the `dependency-jars` directory:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>2.10</version>
  <executions>
    <execution>
      <id>copy-dependencies</id>
      <phase>package</phase>
      <goals>
        <goal>copy-dependencies</goal>
      </goals>
      <configuration>
        <outputDirectory>${project.build.directory}/dependency-jars/</
outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

4. Save the following XML assembly to `src/main/assembly/zip.xml`.

   The following XML is an assembly definition that, when configured, instructs Maven to build a .zip file that contains everything in the root of your build output directory and the `dependency-jars` directory:

```
<assembly
    xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/
assembly/1.1.0 http://maven.apache.org/xsd/assembly-1.1.0.xsd">
  <id>zip</id>
  <formats>
    <format>zip</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
  <fileSets>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory>./</outputDirectory>
      <includes>
        <include>*.jar</include>
      </includes>
    </fileSet>
    <fileSet>
      <directory>${project.build.directory}</directory>
      <outputDirectory>./</outputDirectory>
      <includes>
        <include>/dependency-jars/</include>
      </includes>
    </fileSet>
  </fileSets>
</assembly>
```

5. Modify `pom.xml` to use `maven-assembly-plugin` to package tests and all dependencies into a single .zip file.

   The following plugin uses the preceding assembly to create a .zip file named `zip-with-dependencies` in the build output directory every time **mvn package** is run:

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.5.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
```

```
        </goals>
        <configuration>
          <finalName>zip-with-dependencies</finalName>
          <appendAssemblyId>false</appendAssemblyId>
          <descriptors>
            <descriptor>src/main/assembly/zip.xml</descriptor>
          </descriptors>
        </configuration>
      </execution>
    </executions>
  </plugin>
```

> **Note**
>
> If you receive an error that says annotation is not supported in 1.3, add the following to `pom.xml`:
>
> ```
> <plugin>
>   <artifactId>maven-compiler-plugin</artifactId>
>   <configuration>
>     <source>1.7</source>
>     <target>1.7</target>
>   </configuration>
> </plugin>
> ```

Node.JS

To package your Appium Node.js tests and upload them to Device Farm, you must install the following on your local machine:

- [Node Version Manager (nvm)](https://github.com/nvm-sh/nvm)

  Use this tool when you develop and package your tests so that unnecessary dependencies are not included in your test package.
- Node.js
- npm-bundle (installed globally)

1. Verify that nvm is present

```
command -v nvm
```

You should see nvm as output.

For more information, see nvm on GitHub.

2. Run this command to install Node.js:

```
nvm install node
```

You can specify a particular version of Node.js:

```
nvm install 11.4.0
```

3. Verify that the correct version of Node is in use:

```
node -v
```

4. Install **npm-bundle** globally:

```
npm install -g npm-bundle
```

Python

1. We strongly recommend that you set up Python virtualenv for developing and packaging tests so that unnecessary dependencies are not included in your app package.

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

> **Tip**
>
> - Do not create a Python virtualenv with the `--system-site-packages` option, because it inherits packages from your global site-packages directory. This can result in including dependencies in your virtual environment that are not required by your tests.

- You should also verify that your tests do not use dependencies that are dependent on native libraries, because these native libraries might not be present on the instance where these tests run.

2. Install **py.test** in your virtual environment.

```
$ pip install pytest
```

3. Install the Appium Python client in your virtual environment.

```
$ pip install Appium-Python-Client
```

4. Unless you specify a different path in custom mode, Device Farm expects your tests to be stored in `tests/`. You can use `find` to show all files inside a folder:

```
$ find tests/
```

   Confirm that these files contain test suites you wand to run on Device Farm

```
tests/
tests/my-first-tests.py
tests/my-second-tests/py
```

5. Run this command from your virtual environment workspace folder to show a list of your tests without running them.

```
$ py.test --collect-only tests/
```

   Confirm the output shows the tests that you want to run on Device Farm.

6. Clean all cached files under your tests/ folder:

```
$ find . -name '__pycache__' -type d -exec rm -r {} +
$ find . -name '*.pyc' -exec rm -f {} +
$ find . -name '*.pyo' -exec rm -f {} +
$ find . -name '*~' -exec rm -f {} +
```

7. Run the following command in your workspace to generate the requirements.txt file:

```
$ pip freeze > requirements.txt
```

Ruby

To package your Appium Ruby tests and upload them to Device Farm, you must install the following on your local machine:

- Ruby Version Manager (RVM)

  Use this command-line tool when you develop and package your tests so that unnecessary dependencies are not included in your test package.
- Ruby
- Bundler (This gem is typically installed with Ruby.)

1. Install the required keys, RVM, and Ruby. For instructions, see Installing RVM on the RVM website.

   After the installation is complete, reload your terminal by signing out and then signing in again.

   > **Note**
   >
   > RVM is loaded as a function for the bash shell only.

2. Verify that **rvm** is installed correctly

   ```
   command -v rvm
   ```

   You should see `rvm` as output.

3. If you want to install a specific version of Ruby, such as *`2.5.3`*, run the following command:

   ```
   rvm install ruby 2.5.3 --autolibs=0
   ```

   Verify that you are on the requested version of Ruby:

   ```
   ruby -v
   ```

4. Configure the bundler to compile packages for your desired testing platforms:

   ```
   bundle config specific_platform true
   ```

5. Update your .lock file to add the platforms needed to run tests.

   - If you're compiling tests to run on Android devices, then run this command to configure the Gemfile to use dependencies for the Android test host:

     ```
     bundle lock --add-platform x86_64-linux
     ```

   - If you're compiling tests to run on iOS devices, then run this command to configure the Gemfile to use dependencies for the iOS test host:

     ```
     bundle lock --add-platform x86_64-darwin
     ```

6. The **bundler** gem is usually installed by default. If it is not, install it:

   ```
   gem install bundler -v 2.3.26
   ```

Create a zipped test package file

Warning

In Device Farm, the folder structure of files in your zipped test package matters, and some archival tools will change the structure of your ZIP file implicitly. We recommend that you follow the specified command-line utilities below rather than use the archival utilities built into the file manager of your local desktop (such as Finder or Windows Explorer).

Now, bundle your tests for Device Farm.

Java (JUnit)

Build and package your tests:

```
$ mvn clean package -DskipTests=true
```

The file `zip-with-dependencies.zip` will be created as a result. This is your test package.

Java (TestNG)

Build and package your tests:

```
$ mvn clean package -DskipTests=true
```

The file `zip-with-dependencies.zip` will be created as a result. This is your test package.

Node.JS

1. Check out your project.

   Make sure you are at the root directory of your project. You can see `package.json` at the root directory.

2. Run this command to install your local dependencies.

   ```
   npm install
   ```

   This command also creates a `node_modules` folder inside your current directory.

   > **Note**
   >
   > At this point, you should be able to run your tests locally.

3. Run this command to package the files in your current folder into a *.tgz file. The file is named using the `name` property in your `package.json` file.

   ```
   npm-bundle
   ```

   This tarball (.tgz) file contains all your code and dependencies.

4. Run this command to bundle the tarball (*.tgz file) generated in the previous step into a single zipped archive:

   ```
   zip -r MyTests.zip *.tgz
   ```

   This is the `MyTests.zip` file that you upload to Device Farm in the following procedure.

Python

Python 2

Generate an archive of the required Python packages (called a "wheelhouse") using pip:

```
$ pip wheel --wheel-dir wheelhouse -r requirements.txt
```

Package your wheelhouse, tests, and pip requirements into a zip archive for Device Farm:

```
$ zip -r test_bundle.zip tests/ wheelhouse/ requirements.txt
```

Python 3

Package your tests and pip requirements into a zip file:

```
$ zip -r test_bundle.zip tests/ requirements.txt
```

Ruby

1. Run this command to create a virtual Ruby environment:

```
# myGemset is the name of your virtual Ruby environment
rvm gemset create myGemset
```

2. Run this command to use the environment you just created:

```
rvm gemset use myGemset
```

3. Check out your source code.

   Make sure you are at the root directory of your project. You can see `Gemfile` at the root directory.

4. Run this command to install your local dependencies and all gems from the `Gemfile`:

```
bundle install
```

> **Note**
>
> At this point, you should be able to run your tests locally. Use this command to run a test locally:
>
> ```
> bundle exec $test_command
> ```

5. Package your gems in the `vendor/cache` folder.

```
# This will copy all the .gem files needed to run your tests into the vendor/
cache directory
bundle package --all-platforms
```

6. Run the following command to bundle your source code, along with all your dependencies, into a single zipped archive:

```
zip -r MyTests.zip Gemfile vendor/ $(any other source code directory files)
```

   This is the `MyTests.zip` file that you upload to Device Farm in the following procedure.

Upload your test package to Device Farm

You can use the Device Farm console to upload your tests.

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. If you are a new user, choose **New project**, enter a name for the project, then choose **Submit**.

   If you already have a project, you can choose it to upload your tests to it.
4. Open your project, and then choose **Create a new run**.
5. For native Android and iOS tests

   On the **Choose application** page, choose **Mobile App**, then select **Choose File** to upload your application's distributable package.

   > **Note**
   >
   > The file must be either an Android `.apk` or an iOS `.ipa`. iOS Applications must be built for real devices, not the Simulator.

   For Mobile Web application tests

   On the **Choose application** page, choose **Web App**.

6. Give your test an appropriate name. This may contain any combination of spaces or punctuation.
7. Choose **Next**.
8. On the **Configure** page, in the **Setup test framework** section, choose **Appium *language***, then **Choose File**.
9. Browse to and choose the .zip file that contains your tests. The .zip file must follow the format described in Configure your Appium test package.
10. Choose **Run your test in a custom environment**. This execution environment allows for full control over test setup, teardown, and invocation, as well as choosing specific versions of runtimes and the Appium server. You can configure your custom environment via the test spec file. For more information, see Working with custom test environments in AWS Device Farm.
11. Choose **Next**, and then follow the instructions to select devices and start the run. For more information, see Creating a test run in Device Farm.

> **Note**
>
> Device Farm does not modify Appium tests.

Take screenshots of your tests (Optional)

You can take screenshots as part of your tests.

Device Farm sets the `DEVICEFARM_SCREENSHOT_PATH` property to a fully qualified path on the local file system where Device Farm expects Appium screenshots to be saved. The test-specific directory where the screenshots are stored is defined at runtime. The screenshots are pulled into your Device Farm reports automatically. To view the screenshots, in the Device Farm console, choose the **Screenshots** section.

For more information on taking screenshots in Appium tests, see Take Screenshot in the Appium API documentation.

Android tests in AWS Device Farm

Device Farm provides support for several automation test types for Android devices, and two built-in tests.

For more information about testing in Device Farm, see Test frameworks and built-in tests in AWS Device Farm.

Android application testing frameworks

The following tests are available for Android devices.

- Appium
- Instrumentation

Built-in test types for Android

There's one built-in test type available for Android devices:

- Built-in: fuzz (Android and iOS)

Instrumentation for Android and AWS Device Farm

Device Farm provides support for Instrumentation (JUnit, Espresso, Robotium, or any Instrumentation-based tests) for Android.

Device Farm also provides a sample Android application and links to working tests in three Android automation frameworks, including Instrumentation (Espresso). The Device Farm sample app for Android is available for download on GitHub.

For more information about testing in Device Farm, see Test frameworks and built-in tests in AWS Device Farm.

Topics

- What is instrumentation?
- Considerations for Android instrumentation tests
- Standard mode test parsing
- Integrating Android Instrumentation with Device Farm

What is instrumentation?

Android instrumentation makes it possible for you to invoke callback methods in your test code so you can run through the lifecycle of a component step by step, as if you were debugging the component. For more information, see Instrumented tests in the *Test types and locations* section of the *Android Developer Tools* documentation.

Considerations for Android instrumentation tests

When using Android instrumentation, consider the following recommendations and notes.

Check Android OS Compatibility

Check the Android documentation , to ensure Instrumentation is compatible with your Android OS version.

Running from the Command Line

To run Instrumentation tests from the command line, please follow the Android documentation.

System Animations

Per the Android documentation for Espresso testing, it is recommended that system animations are turned off when testing on real devices. Device Farm automatically disables **Window Animation Scale**, **Transition Animation Scale**, and **Animator Duration Scale** settings when it executes with the android.support.test.runner.AndroidJUnitRunner instrumentation test runner.

Test Recorders

Device Farm supports frameworks, such as Robotium, that have record-and-playback scripting tools.

Standard mode test parsing

In the standard mode of a run, Device Farm parses your test suite and identifies the unique test classes and methods that it will run. This is done through a tool called Dex Test Parser.

When given an Android instrumentation .apk file as input, the parser returns the fully qualified method names of the tests that match JUnit 3 and JUnit 4 conventions.

To test this in a local environment:

1. Download the `dex-test-parser` binary.
2. Run the following command to get the list of test methods that will run on Device Farm:

```
java -jar parser.jar path/to/apk path/for/output
```

Integrating Android Instrumentation with Device Farm

Note

Use the following instructions to integrate Android instrumentation tests with AWS Device Farm. For more information about using instrumentation tests in Device Farm, see Instrumentation for Android and AWS Device Farm.

Upload your Android instrumentation tests

Use the Device Farm console to upload your tests.

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. In the list of projects, choose the project that you want to upload your tests to.

   Tip

   You can use the search bar to filter the project list by name.
   To create a project, follow the instructions in Creating a project in AWS Device Farm.

4. If the **Create a new run** button is displayed, choose it.
5. On the **Choose application** page, select **Choose File**.
6. Browse to and choose your Android app file. The file must be an .apk file.
7. Choose **Next**.
8. On the **Configure** page, in the **Setup test framework** section, choose **Instrumentation**, and then select **Choose File**.
9. Browse to and choose the .apk file that contains your tests.
10. Choose **Next**, and then complete the remaining instructions to select devices and start the run.

(Optional) Take screenshots in Android instrumentation tests

You can take screenshots as part of your Android Instrumentation tests.

To take screenshots, call one of the following methods:

- For Robotium, call the `takeScreenShot` method (for example, `solo.takeScreenShot();`).
- For Spoon, call the `screenshot` method, for example:

```
Spoon.screenshot(activity, "initial_state");
/* Normal test code... */
Spoon.screenshot(activity, "after_login");
```

During a test run, Device Farm gets screenshots from the following locations on the devices, if they exist, and then adds them to the test reports:

- `/sdcard/robotium-screenshots`
- `/sdcard/test-screenshots`
- `/sdcard/Download/spoon-screenshots/test-class-name/test-method-name`
- `/data/data/application-package-name/app_spoon-screenshots/test-class-name/test-method-name`

iOS tests in AWS Device Farm

Device Farm provides support for several automation test types for iOS devices, and a built-in test.

For more information about testing in Device Farm, see Test frameworks and built-in tests in AWS Device Farm.

iOS application testing frameworks

The following tests are available for iOS devices.

- Appium
- XCTest
- XCTest UI

Built-in test types for iOS

There is currently one built-in test type available for iOS devices.

- Built-in: fuzz (Android and iOS)

Integrating Device Farm with XCTest for iOS

With Device Farm, you can use the XCTest framework to test your app on real devices. For more information about XCTest, see Testing Basics in *Testing with Xcode*.

To run a test, you create the packages for your test run, and you upload these packages to Device Farm.

For more information about testing in Device Farm, see Test frameworks and built-in tests in AWS Device Farm.

Topics

Create the packages for your XCTest run

To test your app by using the XCTest framework, Device Farm requires the following:

- Your app package as a `.ipa` file.
- Your XCTest package as a `.zip` file.

You create these packages by using the build output that Xcode generates. Complete the following steps to create the packages so that you can upload them to Device Farm.

To generate the build output for your app

1. Open your app project in Xcode.
2. In the scheme dropdown menu in the Xcode toolbar, choose **Generic iOS Device** as the destination.
3. In the **Product** menu, choose **Build For**, and then choose **Testing**.

To create the app package

1. In the project navigator in Xcode, under **Products**, open the contextual menu for the file named *`app-project-name`*`.`app. Then, choose **Show in Finder**. Finder opens a folder named

`Debug-iphoneos`, which contains the output that Xcode generated for your test build. This folder includes your `.app` file.

2. In Finder, create a new folder, and name it `Payload`.
3. Copy the *`app-project-name`*`.app` file, and paste it in the `Payload` folder.
4. Open the contextual menu for the `Payload` folder and choose **Compress "Payload"**. A file named `Payload.zip` is created.
5. Change the file name and extension of `Payload.zip` to *`app-project-name`*`.ipa`.

   In a later step, you provide this file to Device Farm. To make the file easier to find, you might want to move it to another location, such as your desktop.
6. Optionally, you can delete the `Payload` folder and the `.app` file in it.

To create the XCTest package

1. In Finder, in the `Debug-iphoneos` directory, open the contextual menu for the *`app-project-name`*`.app` file. Then, choose **Show Package Contents**.
2. In the package contents, open the `Plugins` folder. This folder contains a file named *`app-project-name`*`.xctest`.
3. Open the contextual menu for this file and choose **Compress "*`app-project-name`*`.xctest`"**. A file named *`app-project-name`*`.xctest.zip` is created.

   In a later step, you provide this file to Device Farm. To make the file easier to find, you might want to move it to another location, such as your desktop.

Upload the packages for your XCTest run to Device Farm

Use the Device Farm console to upload the packages for your test.

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. If you don't have a project already, create one. For the steps to create a project, see Creating a project in AWS Device Farm.

   Otherwise, on the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. Choose the project that you want to use to run the test.

4. Choose **Create a new run**.
5. On the **Choose application** page, choose **Mobile App**.
6. Select **Choose File**.
7. Browse to the `.ipa` file for your app and upload it.

> **Note**
>
> Your `.ipa` package must be built for testing.

8. After the upload completes, choose **Next**.
9. On the **Configure** page, in the **Setup test framework** section, choose **XCTest**. Then, select **Choose File**.
10. Browse to the `.zip` file that contains the XCTest package for your app and upload it.
11. After the upload completes, choose **Next**.
12. Complete the remaining steps in the project creation process. You will select the devices that you want to test on and specify the device state.
13. After you configure your run, on the **Review and start run** page, choose **Confirm and start run**.

    Device Farm runs your test and shows the results in the console.

Integrating XCTest UI for iOS with Device Farm

Device Farm provides support for the XCTest UI testing framework. Specifically, Device Farm supports XCTest UI tests written in both Objective-C and Swift.

The XCTest UI framework enables UI testing in iOS development, built on top of XCTest. For more information, see User Interface Testing in the iOS Developer Library.

For general information about testing in Device Farm, see Test frameworks and built-in tests in AWS Device Farm.

Use the following instructions to integrate Device Farm with the XCTest UI testing framework for iOS.

Topics

- Prepare your iOS XCTest UI tests
- Option 1: Creating an XCTest UI .ipa package
- Option 2: Creating an XCTest UI .zip package
- Upload your iOS XCTest UI tests

Prepare your iOS XCTest UI tests

You can either upload an `.ipa` file or a `.zip` file for your XCTEST_UI test package.

An `.ipa` file is an application archive containing the iOS Runner app in bundle format. *Additional files cannot be included inside the `.ipa` file.*

If you upload a `.zip` file, it can contain either the iOS Runner app directly or an `.ipa` file. You can also include other files within the `.zip` file if you want to use them during the tests. For example you can include files like `.xctestrun`, `.xcworkspace` or `.xcodeproj` inside `.zip` file to run XCUI Test Plans on device farm. Detailed instructions on how to run Test Plans are available in the default test specification file for the XCUI Test type.

Option 1: Creating an XCTest UI .ipa package

The *yourAppName*UITest-Runner.app bundle is produced by Xcode when you build your project for testing. It can be found in the Products directory for your project.

To create an .ipa file:

1. Create a directory called *`Payload`*.
2. Add your app directory to the Payload directory.
3. Archive the Payload directory into a `.zip` file and then change the file extension to `.ipa`.

The following folder structure shows how an example app named *`my-project-nameUITest-Runner.app`* would be packaged as an `.ipa` file:

```
.
### my-project-nameUITest.ipa
    ### Payload (directory)
        ### my-project-nameUITest-Runner.app
```

Option 2: Creating an XCTest UI .zip package

Device Farm automatically generates a `.xctestrun` file for you for running your full XCTest UI test suite. If you want to use your own `.xctestrun` file on Device Farm, you can compress your `.xctestrun` files and app directory into a `.zip` file. If you already have a `.ipa` file for your test package you can include that here instead of *`*-Runner.app`*.

```
.
### swift-sample-UI.zip (directory)
   ### my-project-nameUITest-Runner.app [OR] my-project-nameUITest.ipa
   ### SampleTestPlan_2.xctestrun
   ### SampleTestPlan_1.xctestrun
   ### (any other files)
```

If you want to run an Xcode test plan for your XCUI tests on Device Farm, you can create a zip containing your *my-project-nameUITest-Runner.app* **or** *my-project-nameUITest.ipa* file and xcode source code files required to run XCTEST_UI with test plans, including either a `.xcworkspace` or `.xcodeproj` file.

Here is a sample zip using a `.xcodeproj` file:

```
.
### swift-sample-UI.zip (directory)
   ### my-project-nameUITest-Runner.app [OR] my-project-nameUITest.ipa
   ### (any directory)
   ### SampleXcodeProject.xcodeproj
        ### Testplan_1.xctestplan
        ### Testplan_2.xctestplan
        ### (any other source code files created by xcode with .xcodeproj)
```

Here is a sample zip using a `.xcworkspace` file:

```
.
###swift-sample-UI.zip (directory)
   ### my-project-nameUITest-Runner.app [OR] my-project-nameUITest.ipa
   ### (any directory)
```

```
#   ### SampleXcodeProject.xcodeproj
#   ### Testplan_1.xctestplan
#   ### Testplan_2.xctestplan
|   ### (any other source code files created by xcode with .xcodeproj)
### SampleWorkspace.xcworkspace
    ### contents.xcworkspacedata
```

Note

Please ensure that you do not have a directory named "Payload" inside your XCTest UI .zip package.

Upload your iOS XCTest UI tests

Use the Device Farm console to upload your tests.

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. In the list of projects, choose the project that you want to upload your tests to.

   Tip

   You can use the search bar to filter the project list by name.
   To create a project, follow the instructions in Creating a project in AWS Device Farm
4. If the **Create a new run** button is displayed, choose it.
5. On the **Choose application** page, select **Choose File**.
6. Browse to and choose your iOS app file. The file must be an .ipa file.

   Note

   Make sure that your .ipa file is built for an iOS device and not for a simulator.
7. Choose **Next**.
8. On the **Configure** page, in the **Setup test framework** section, choose **XCTest UI**, and then select **Choose File**.

9. Browse to and choose the .ipa or .zip file that contains your iOS XCTest UI test runner.
10. Choose **Next**, and then complete the remaining instructions to select the devices to run your tests on and start the run.

Web app tests in AWS Device Farm

Device Farm provides testing with Appium for web applications. For more information on setting up your Appium tests on Device Farm, see the section called “Appium”.

For more information about testing in Device Farm, see Test frameworks and built-in tests in AWS Device Farm.

Rules for metered and unmetered devices

Metering refers to billing for devices. By default, Device Farm devices are metered and you are charged per minute after the free trial minutes are used up. You can also choose to purchase unmetered devices, which allow unlimited testing for a flat monthly fee. For more information about pricing, see AWS Device Farm Pricing.

If you choose to start a run with a device pool that contains both iOS and Android devices, there are rules for metered and unmetered devices. For example, if you have five unmetered Android devices and five unmetered iOS devices, your web test runs use your unmetered devices.

Here is another example: Suppose you have five unmetered Android devices and 0 unmetered iOS devices. If you select only Android devices for your web run, your unmetered devices are used. If you select both Android and iOS devices for your web run, the billing method is metered, and your unmetered devices are not used.

Built-in tests in AWS Device Farm

Device Farm provides support for built-in test types for Android and iOS devices.

With built-in tests, you can test your application on multiple devices without having to write and maintain test automation scripts. This can save you time and effort, especially when you're getting started with Device Farm. Device Farm offers the following built-in test type:

- Built-in: fuzz (Android and iOS) – The fuzz test randomly sends user interface events to devices and then reports results.

For more information about tests and testing frameworks in Device Farm, see Test frameworks and built-in tests in AWS Device Farm.

Running Device Farm's built-in fuzz test (Android and iOS)

Device Farm's built-in fuzz test randomly sends user interface events to devices and then reports results.

For more information about testing in Device Farm, see Test frameworks and built-in tests in AWS Device Farm.

To run the built-in fuzz test

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. In the list of projects, choose the project where you want to run the built-in fuzz test.

   > **Tip**
   >
   > You can use the search bar to filter the project list by name.
   > To create a project, follow the instructions in Creating a project in AWS Device Farm.

4. If the **Create a new run** button is displayed, choose it.
5. On the **Choose application** page, select **Choose File**.
6. Browse to and choose your app file where you want to run the built-in fuzz test.
7. Choose **Next**.
8. On the **Configure** page, in the **Setup test framework** section, choose **Built-in: Fuzz**.
9. If any of the following settings appear, you can either accept the default values or specify your own:

   - **Event count**: Specify a number between 1 and 10,000, representing the number of user interface events for the fuzz test to perform.
   - **Event throttle**: Specify a number between 0 and 1,000, representing the number of milliseconds for the fuzz test to wait before performing the next user interface event.
   - **Randomizer seed**: Specify a number for the fuzz test to use for randomizing user interface events. Specifying the same number for subsequent fuzz tests ensures identical event sequences.

10. Choose **Next**, and then complete the remaining instructions to select devices and start the run.

Custom test environments in AWS Device Farm

AWS Device Farm enables configuring a custom environment for automated testing (custom mode), which is the recommended approach for all Device Farm users. To learn more about environments in Device Farm, see Test environments.

Benefits of the Custom Mode as opposed to the Standard Mode include:

- **Faster end-to-end test execution**: The test package isn't parsed to detect every test in the suite, avoiding preprocessing/postprocessing overhead.
- **Live log and video streaming**: Your client-side test logs and video are live streamed when using Custom Mode. This feature isn't available in the standard mode.
- **Captures all artifacts**: On the host and device, Custom Mode allows you to capture all test artifacts. This may not be possible in the standard mode.
- **More consistent and replicable local environment**: When in Standard Mode, artifacts will be provided for each individual test separately, which can be beneficial under certain circumstances. However, your local test environment may deviate from the original configuration as Device Farm handles each executed test differently.

  In contrast, Custom Mode enables you to make your Device Farm test execution environment consistently in line with with your local test environment.

Custom environments are configured using a YAML-formatted test specification (test spec) file. Device Farm provides a default test spec file for each supported test type that can be used as is or customized; customizations like test filters or config files can be added to the test spec. Edited test specs can be saved for future test runs.

For more information, see Uploading a Custom Test Spec Using the AWS CLI and Creating a test run in Device Farm.

Topics

- Extending custom test environments in Device Farm

Test spec syntax in Device Farm

The test spec is a file that you use to define custom test environments in AWS Device Farm. For more information about the custom environments and the test spec file, see Custom test environments in AWS Device Farm.

The following is the YAML test spec file structure. Following the structure is a description of each property.

To see an example test spec file, refer to Example of the Device Farm test spec file.

```
version: 0.1

phases:
  install:
    commands:
      - command
      - command
  pre_test:
    commands:
      - command
      - command
  test:
    commands:
      - command
      - command
  post_test:
    commands:
      - command
      - command

artifacts:
  - location
  - location
```

The test spec contains the following:

`version`

Reflects the Device Farm supported test spec version. The current version number is 0.1.

phases

This section contains groups of commands executed during a test run.

The allowed test phase names are:

install

Optional.

Default dependencies for testing frameworks supported by Device Farm are already installed. This phase contains additional commands, if any, that Device Farm runs during installation.

pre_test

Optional.

The commands, if any, executed before your automated test run.

test

Optional.

The commands executed during your automated test run. If any command in the test phase fails, the test is marked as failed.

post_test

Optional.

The commands, if any, executed after your automated test run.

artifacts

Optional.

Device Farm gathers artifacts such as custom reports, log files, and images from a location specified here. Wildcard characters are not supported as part of an artifact location, so you must specify a valid path for each location.

These test artifacts are available for each device in your test run. For information about retrieving your test artifacts, see [Downloading artifacts in a custom test environment](#).

Important

A test spec must be formatted as a valid YAML file. If the indenting or spacing in your test spec are invalid, your test run can fail. Tabs are not allowed in YAML files. You can use a YAML validator to test whether your test spec is a valid YAML file. For more information, see the YAML website.

Example of the Device Farm test spec file

The test spec is a file that you use to define custom test environments in AWS Device Farm. For more information about the custom environments and the test spec file, see Custom test environments in AWS Device Farm.. For more information about the test spec file's structure and contents, see Test spec syntax in Device Farm.

The following is an example of a Device Farm YAML test spec that configures an Appium Java TestNG test run.

```
version: 0.1

# This flag enables your test to run using Device Farm's Amazon Linux 2 test host when
 scheduled on
# Android devices. By default, iOS device tests will always run on Device Farm's macOS
 test hosts.
# For Android, you can explicitly select your test host to use our Amazon Linux 2
 infrastructure.
# For more information, please see:
# https://docs.aws.amazon.com/devicefarm/latest/developerguide/amazon-linux-2.html
android_test_host: amazon_linux_2

# Phases represent collections of commands that are executed during your test run on
 the test host.
phases:

  # The install phase contains commands for installing dependencies to run your tests.
  # For your convenience, certain dependencies are preinstalled on the test host.

  # For Android tests running on the Amazon Linux 2 test host, many software libraries
 are available
  # from the test host using the devicefarm-cli tool. To learn more, please see:
```

```
  # https://docs.aws.amazon.com/devicefarm/latest/developerguide/amazon-linux-2-
devicefarm-cli.html

  # For iOS tests, you can use the Node.JS tools nvm, npm, and avm to setup your
 environment. By
  # default, Node.js versions 16.20.2 and 14.19.3 are available on the test host.
  install:
    commands:
      # The Appium server is written using Node.js. In order to run your desired
 version of Appium,
      # you first need to set up a Node.js environment that is compatible with your
 version of Appium.
      - |-
        if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "Android" ];
        then
          devicefarm-cli use node 16;
        else
          # For iOS, use "nvm use" to switch between the two preinstalled NodeJS
 versions 14 and 16,
          # and use "nvm install" to download a new version of your choice.
          nvm use 16;
        fi;
      - node --version

      # Use the devicefarm-cli to select a preinstalled major version of Appium on
 Android.
      # Use avm or npm to select Appium for iOS.
      - |-
        if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "Android" ];
        then
          # For Android, the Device Farm service automatically updates the preinstalled
 Appium versions
          # over time to incorporate the latest minor and patch versions for each major
 version. If you
          # wish to select a specific version of Appium, you can instead use NPM to
 install it:
          # npm install -g appium@2.1.3;
          devicefarm-cli use appium 2;
        else
          # For iOS, Appium versions 1.22.2 and 2.2.1 are preinstalled and selectable
 through avm.
          # For all other versions, please use npm to install them. For example:
          # npm install -g appium@2.1.3;
```

```
          # Note that, for iOS devices, Appium 2 is only supported on iOS version 14 
and above using
          # NodeJS version 16 and above.
          avm 2.2.1;
        fi;
      - appium --version

      # For Appium version 2, for Android tests, Device Farm automatically updates the 
preinstalled
      # UIAutomator2 driver over time to incorporate the latest minor and patch 
versions for its major
      # version 2. If you want to install a specific version of the driver, you can use 
the Appium
      # extension CLI to uninstall the existing UIAutomator2 driver and install your 
desired version:
      # - |-
      #   if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "Android" ];
      #   then
      #     appium driver uninstall uiautomator2;
      #     appium driver install uiautomator2@2.34.0;
      #   fi;

      # For Appium version 2, for iOS tests, the XCUITest driver is preinstalled using 
version 5.7.0
      # If you want to install a different version of the driver, you can use the 
Appium extension CLI
      # to uninstall the existing XCUITest driver and install your desired version:
      # - |-
      #   if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "iOS" ];
      #   then
      #     appium driver uninstall xcuitest;
      #     appium driver install xcuitest@5.8.1;
      #   fi;

      # We recommend setting the Appium server's base path explicitly for accepting 
commands.
      - export APPIUM_BASE_PATH=/wd/hub

      # Install the NodeJS dependencies.
      - cd $DEVICEFARM_TEST_PACKAGE_PATH
      # First, install dependencies which were packaged with the test package using 
npm-bundle.
      - npm install *.tgz
      # Then, optionally, install any additional dependencies using npm install.
```

```
      # If you do run these commands, we strongly recommend that you include your 
package-lock.json
      # file with your test package so that the dependencies installed on Device Farm 
match
      # the dependencies you've installed locally.
      # - cd node_modules/*
      # - npm install

  # The pre-test phase contains commands for setting up your test environment.
  pre_test:
    commands:
      # Device farm provides different pre-built versions of WebDriverAgent, an 
essential Appium
      # dependency for iOS devices, and each version is suggested for different 
versions of Appium:
      # DEVICEFARM_WDA_DERIVED_DATA_PATH_V8: this version is suggested for Appium 2
      # DEVICEFARM_WDA_DERIVED_DATA_PATH_V7: this version is suggested for Appium 1
      # Additionally, for iOS versions 16 and below, the device unique identifier 
(UDID) needs
      # to be slightly modified for Appium tests.
      - |-
        if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "iOS" ];
        then
          if [ $(appium --version | cut -d "." -f1) -ge 2 ];
          then
            DEVICEFARM_WDA_DERIVED_DATA_PATH=$DEVICEFARM_WDA_DERIVED_DATA_PATH_V8;
          else
            DEVICEFARM_WDA_DERIVED_DATA_PATH=$DEVICEFARM_WDA_DERIVED_DATA_PATH_V7;
          fi;

          if [ $(echo $DEVICEFARM_DEVICE_OS_VERSION | cut -d "." -f 1) -le 16 ];
          then
            DEVICEFARM_DEVICE_UDID_FOR_APPIUM=$(echo $DEVICEFARM_DEVICE_UDID | tr -d 
"-");
          else
            DEVICEFARM_DEVICE_UDID_FOR_APPIUM=$DEVICEFARM_DEVICE_UDID;
          fi;
        fi;

      # Appium downloads Chromedriver using a feature that is considered insecure for 
multitenant
      # environments. This is not a problem for Device Farm because each test host is 
allocated
```

```
      # exclusively for one customer, then terminated entirely. For more information,
 please see
      # https://github.com/appium/appium/blob/master/packages/appium/docs/en/guides/
security.md

      # We recommend starting the Appium server process in the background using the
 command below.
      # The Appium server log will be written to the $DEVICEFARM_LOG_DIR directory.
      # The environment variables passed as capabilities to the server will be
 automatically assigned
      # during your test run based on your test's specific device.
      # For more information about which environment variables are set and how they're
 set, please see
      # https://docs.aws.amazon.com/devicefarm/latest/developerguide/custom-test-
environment-variables.html
      - |-
        if [ $DEVICEFARM_DEVICE_PLATFORM_NAME = "Android" ];
        then
          appium --base-path=$APPIUM_BASE_PATH --log-timestamp \
            --log-no-colors --relaxed-security --default-capabilities \
            "{\"appium:deviceName\": \"$DEVICEFARM_DEVICE_NAME\", \
            \"platformName\": \"$DEVICEFARM_DEVICE_PLATFORM_NAME\", \
            \"appium:app\": \"$DEVICEFARM_APP_PATH\", \
            \"appium:udid\":\"$DEVICEFARM_DEVICE_UDID\", \
            \"appium:platformVersion\": \"$DEVICEFARM_DEVICE_OS_VERSION\", \
            \"appium:chromedriverExecutableDir\":
 \"$DEVICEFARM_CHROMEDRIVER_EXECUTABLE_DIR\", \
            \"appium:automationName\": \"UiAutomator2\"}" \
            >> $DEVICEFARM_LOG_DIR/appium.log 2>&1 &
        else
          appium --base-path=$APPIUM_BASE_PATH --log-timestamp \
            --log-no-colors --relaxed-security --default-capabilities \
            "{\"appium:deviceName\": \"$DEVICEFARM_DEVICE_NAME\", \
            \"platformName\": \"$DEVICEFARM_DEVICE_PLATFORM_NAME\", \
            \"appium:app\": \"$DEVICEFARM_APP_PATH\", \
            \"appium:udid\":\"$DEVICEFARM_DEVICE_UDID_FOR_APPIUM\", \
            \"appium:platformVersion\": \"$DEVICEFARM_DEVICE_OS_VERSION\", \
            \"appium:derivedDataPath\": \"$DEVICEFARM_WDA_DERIVED_DATA_PATH\", \
            \"appium:usePrebuiltWDA\": true, \
            \"appium:automationName\": \"XCUITest\"}" \
            >> $DEVICEFARM_LOG_DIR/appium.log 2>&1 &
        fi;

      # This code will wait until the Appium server starts.
```

```
      - |-
        appium_initialization_time=0;
        until curl --silent --fail "http://0.0.0.0:4723${APPIUM_BASE_PATH}/status"; do
          if [[ $appium_initialization_time -gt 30 ]]; then
            echo "Appium did not start within 30 seconds. Exiting...";
            exit 1;
          fi;
          appium_initialization_time=$((appium_initialization_time + 1));
          echo "Waiting for Appium to start on port 4723...";
          sleep 1;
        done;

  # The test phase contains commands for running your tests.
  test:
    commands:
      # Your test package is downloaded and unpackaged into the
 $DEVICEFARM_TEST_PACKAGE_PATH directory.
      # When compiling with npm-bundle, the test folder can be found in the
 node_modules/*/ subdirectory.
      - cd $DEVICEFARM_TEST_PACKAGE_PATH/node_modules/*
      - echo "Starting the Appium NodeJS test"

      # Enter your command below to start the tests. The command should be the same
 command as the one
      # you use to run your tests locally from the command line. An example, "npm
 test", is given below:
      - npm test

  # The post-test phase contains commands that are run after your tests have completed.
  # If you need to run any commands to generating logs and reports on how your test
 performed,
  # we recommend adding them to this section.
  post_test:
    commands:

# Artifacts are a list of paths on the filesystem where you can store test output and
 reports.
# All files in these paths will be collected by Device Farm.
# These files will be available through the ListArtifacts API as your "Customer
 Artifacts".
artifacts:
  # By default, Device Farm will collect your artifacts from the $DEVICEFARM_LOG_DIR
 directory.
```

```
  - $DEVICEFARM_LOG_DIR
```

Amazon Linux 2 test environment for Android tests

AWS Device Farm utilizes Amazon Elastic Compute Cloud (EC2) host machines running Amazon Linux 2 to execute Android tests. When you schedule a test run, Device Farm allocates a dedicated host for each device to independently run tests. The host machines terminate after the test run along with any generated artifacts.

The Amazon Linux 2 test host is the newest Android test environment, replacing the previous Ubuntu-based system. Using your test spec file, you can choose to run your Android tests on the Amazon Linux 2 environment.

The Amazon Linux 2 host provides several advantages:

- **Faster, more reliable testing**: Compared to the legacy host, the new test host significantly improves test speed, especially reducing test start times. The Amazon Linux 2 host also demonstrates greater stability and reliability during testing.
- **Enhanced Remote Access for manual testing**: Upgrades to the latest test host and improvements lead to lower latency and better video performance for Android manual testing.
- **Standard software version selection**: Device Farm now standardizes major programming language support on the test host as well as Appium framework versions. For supported languages (currently Java, Python, Node.js, and Ruby) and Appium, the new test host provides long-term stable releases soon after launch. Centralized version management through the `devicefarm-cli` tool enables test spec file development with a consistent experience across frameworks.

Topics

Pre-installed software libraries to support Device Farm tests of Android devices

AWS Device Farm uses Amazon Elastic Compute Cloud (EC2) host machines running Amazon Linux 2 to execute Android tests. The Amazon Linux 2 test host comes pre-installed with many of the necessary software libraries to support Device Farm test frameworks, providing a ready testing environment on launch. For any other required software, you can modify the test spec file to install from your test package, download from the internet, or access private sources within your VPC (see VPC ENI for more information). For more information, see the Test spec file example.

The following software versions are currently available on the host:

Software Library	Software Version	Command to use in your test spec file
Python	3.8	`devicefarm-cli use python 3.8`
	3.9	`devicefarm-cli use python 3.9`
	3.10	`devicefarm-cli use python 3.10`
	3.11	`devicefarm-cli use python 3.11`
Java	8	`devicefarm-cli use java 8`
	11	`devicefarm-cli use java 11`
	17	`devicefarm-cli use java 17`
NodeJS	16	`devicefarm-cli use node 16`

	18	`devicefarm-cli use node 18`
	20	`devicefarm-cli use node 20`
Ruby	2.7	`devicefarm-cli use ruby 2.7`
	3.2	`devicefarm-cli use ruby 3.2`
Appium	1	`devicefarm-cli use appium 1`
	2	`devicefarm-cli use appium 2`

The test host also includes commonly used supporting tools for each software version, such as the `pip` and `npm` package managers (included with Python and Node.js respectively) and dependencies (such as the Appium UIAutomator2 Driver) for tools like Appium. This ensures you have the tools needed to work with the supported test frameworks.

Supported IP ranges for the Amazon Linux 2 test environment in Device Farm

Customers often need to know the IP range from which Device Farm's traffic originates, particularly for configuring their firewalls and security settings. For Amazon EC2 test hosts, the IP range encompasses the entire `us-west-2` region. For Amazon Linux 2 test hosts, which is the default option for new Android runs, the ranges have been restricted. The traffic now originates from a specific set of NAT gateways, restricting the IP range to the following addresses:

IP Ranges
44.236.137.143
52.13.151.244

IP Ranges
52.35.189.191
54.201.250.26

For more information about Android test environments in Device Farm, see [Amazon Linux 2 test environment for Android tests](#).

Using the `devicefarm-cli` tool in AWS Device Farm

AWS Device Farm uses Amazon Elastic Compute Cloud (EC2) host machines running Amazon Linux 2 to execute Android tests. The Amazon Linux 2 test host uses a standardized version management tool called `devicefarm-cli` to select software versions. This tool is separate from the AWS CLI and only available on the Device Farm Test Host. With `devicefarm-cli`, you can switch to any pre-installed software version on the test host. This provides a straightforward way to maintain your Device Farm test spec file over time and gives you a predictable mechanism to upgrade software versions in the future.

The snippet below shows the `help` page of `devicefarm-cli`:

```
$ devicefarm-cli help
 Usage: devicefarm-cli COMMAND [ARGS]

     Commands:
         help                         Prints this usage message.
         list                         Lists all versions of software configurable
                                      via this CLI.
         use <software> <version>     Configures the software for usage within the
                                      current shell's environment.
```

Let's review a couple of examples using `devicefarm-cli`. To use the tool to change the Python version from *3.10* to *3.9* in your test spec file, run the following commands:

```
$ python --version
Python 3.10.12
$ devicefarm-cli use python 3.9
$ python --version
Python 3.9.17
```

To change the Appium version from *1* to *2*:

```
$ appium --version
1.22.3
$ devicefarm-cli use appium 2
$ appium --version
2.1.2
```

> **Tip**
>
> Note that when you select a software version, `devicefarm-cli` also switches the supporting tools for those languages, such as `pip` for Python and `npm` for NodeJS.

For more information about how Device Farm tests Android devices, see Amazon Linux 2 test environment for Android tests.

For more information about the preinstalled software on the Amazon Linux 2 test host, see Pre-installed software libraries to support Device Farm tests of Android devices.

Selecting which Android test host to use in Device Farm

> **Warning**
>
> The legacy Android Test Host will no longer be available on October 21, 2024. Note that the process for deprecation is split across several dates:
>
> - On April 22, 2024, jobs from any new account will be directed to the upgraded test host.
> - On September 2, 2024, all new or modified test spec files must target the upgraded test host.
> - On October 21, 2024, jobs will no longer be able to run on the legacy test host.
>
> Set your test spec files to the `amazon_linux_2` host to prevent compatibility issues. Please note that the Legacy Android Test Host only supports Android versions 14 and lower. Use the amazon_linux_2 host for Android versions 15 and higher.

AWS Device Farm uses Amazon Elastic Compute Cloud (EC2) host machines running Amazon Linux 2 to execute Android tests. For Android tests, Device Farm requires the following field in your test spec file to choose the Amazon Linux 2 test host:

```
android_test_host: amazon_linux_2 | legacy
```

Use `amazon_linux_2` to run your tests on the Amazon Linux 2 test host:

```
android_test_host: amazon_linux_2
```

Learn more about the benefits of Amazon Linux 2 here.

Device Farm recommends using the Amazon Linux 2 host for Android tests instead of the legacy host environment. If you'd prefer to use the legacy environment, then use `legacy` to run your tests on the legacy test host:

```
android_test_host: legacy
```

By default, test spec files without a test host selection will run on the legacy test host.

Deprecated syntax

Below is the deprecated syntax for choosing Amazon Linux 2 in your test spec file:

```
preview_features:
  android_amazon_linux_2_host: true
```

If you're using this flag, your tests will continue to run on Amazon Linux 2. However, we **strongly recommend** removing the `preview_features` flags section and replacing it with the new `android_test_host` field in order to avoid maintenance overhead in the future.

> **Warning**
>
> Using both the `android_test_host` and `android_amazon_linux_2_host` flags in your test spec file will return an error. Only one should be used; we recommend `android_test_host`.

Example: Device Farm test spec file showing an Android test host

The following snippet is an example of a Device Farm test spec file that configures an Appium NodeJS test run using the Amazon Linux 2 test host for Android:

```
version: 0.1

# This flag enables your test to run using Device Farm's Amazon Linux 2 test host. For
 more information,
# please see https://docs.aws.amazon.com/devicefarm/latest/developerguide/amazon-
linux-2.html
android_test_host: amazon_linux_2

# Phases represent collections of commands that are executed during your test run on
 the test host.
phases:

  # The install phase contains commands for installing dependencies to run your tests.
  # For your convenience, certain dependencies are preinstalled on the test host. To
 lean about which
  # software is included with the host, and how to install additional software, please
 see:
  # https://docs.aws.amazon.com/devicefarm/latest/developerguide/amazon-linux-2-
supported-software.html

  # Many software libraries you may need are available from the test host using the
 devicefarm-cli tool.
  # To learn more about what software is available from it and how to use it, please
 see:
  # https://docs.aws.amazon.com/devicefarm/latest/developerguide/amazon-linux-2-
devicefarm-cli.html

  install:
    commands:
      # The Appium server is written using Node.js. In order to run your desired
 version of Appium,
      # you first need to set up a Node.js environment that is compatible with your
 version of Appium.
      - devicefarm-cli use node 18
      - node --version

      # Use the devicefarm-cli to select a preinstalled major version of Appium.
      - devicefarm-cli use appium 2
```

```
      - appium --version

      # The Device Farm service automatically updates the preinstalled Appium versions
 over time to
      # incorporate the latest minor and patch versions for each major version. If you
 wish to
      # select a specific version of Appium, you can use NPM to install it.
      # - npm install -g appium@2.1.3

      # For Appium version 2, Device Farm automatically updates the preinstalled
 UIAutomator2 driver
      # over time to incorporate the latest minor and patch versions for its major
 version 2. If you
      # want to install a specific version of the driver, you can use the Appium
 extension CLI to
      # uninstall the existing UIAutomator2 driver and install your desired version:
      # - appium driver uninstall uiautomator2
      # - appium driver install uiautomator2@2.34.0

      # We recommend setting the Appium server's base path explicitly for accepting
 commands.
      - export APPIUM_BASE_PATH=/wd/hub

      # Install the NodeJS dependencies.
      - cd $DEVICEFARM_TEST_PACKAGE_PATH
      # First, install dependencies which were packaged with the test package using
 npm-bundle.
      - npm install *.tgz
      # Then, optionally, install any additional dependencies using npm install.
      # If you do run these commands, we strongly recommend that you include your
 package-lock.json
      # file with your test package so that the dependencies installed on Device Farm
 match
      # the dependencies you've installed locally.
      # - cd node_modules/*
      # - npm install

  # The pre-test phase contains commands for setting up your test environment.
  pre_test:
    commands:

      # Appium downloads Chromedriver using a feature that is considered insecure for
 multitenant
```

```
      # environments. This is not a problem for Device Farm because each test host is
 allocated
      # exclusively for one customer, then terminated entirely. For more information,
 please see
      # https://github.com/appium/appium/blob/master/packages/appium/docs/en/guides/
security.md

      # We recommend starting the Appium server process in the background using the
 command below.
      # The Appium server log will be written to the $DEVICEFARM_LOG_DIR directory.
      # The environment variables passed as capabilities to the server will be
 automatically assigned
      # during your test run based on your test's specific device.
      # For more information about which environment variables are set and how they're
 set, please see
      # https://docs.aws.amazon.com/devicefarm/latest/developerguide/custom-test-
environment-variables.html
      - |-
        appium --base-path=$APPIUM_BASE_PATH --log-timestamp \
          --log-no-colors --relaxed-security --default-capabilities \
          "{\"appium:deviceName\": \"$DEVICEFARM_DEVICE_NAME\", \
          \"platformName\": \"$DEVICEFARM_DEVICE_PLATFORM_NAME\", \
          \"appium:app\": \"$DEVICEFARM_APP_PATH\", \
          \"appium:udid\":\"$DEVICEFARM_DEVICE_UDID\", \
          \"appium:platformVersion\": \"$DEVICEFARM_DEVICE_OS_VERSION\", \
          \"appium:chromedriverExecutableDir\":
 \"$DEVICEFARM_CHROMEDRIVER_EXECUTABLE_DIR\", \
          \"appium:automationName\": \"UiAutomator2\"}" \
          >> $DEVICEFARM_LOG_DIR/appium.log 2>&amp;1 &amp;

      # This code will wait until the Appium server starts.
      - |-
        appium_initialization_time=0;
        until curl --silent --fail "http://0.0.0.0:4723${APPIUM_BASE_PATH}/status"; do
          if [[ $appium_initialization_time -gt 30 ]]; then
            echo "Appium did not start within 30 seconds. Exiting...";
            exit 1;
          fi;
          appium_initialization_time=$((appium_initialization_time + 1));
          echo "Waiting for Appium to start on port 4723...";
          sleep 1;
        done;

  # The test phase contains commands for running your tests.
```

```
  test:
    commands:
      # Your test package is downloaded and unpackaged into the
 $DEVICEFARM_TEST_PACKAGE_PATH directory.
      # When compiling with npm-bundle, the test folder can be found in the
 node_modules/*/ subdirectory.
      - cd $DEVICEFARM_TEST_PACKAGE_PATH/node_modules/*
      - echo "Starting the Appium NodeJS test"

      # Enter your command below to start the tests. The command should be the same
 command as the one
      # you use to run your tests locally from the command line. An example, "npm
 test", is given below:
      - npm test

  # The post-test phase contains commands that are run after your tests have completed.
  # If you need to run any commands to generating logs and reports on how your test
 performed,
  # we recommend adding them to this section.
  post_test:
    commands:

# Artifacts are a list of paths on the filesystem where you can store test output and
 reports.
# All files in these paths will be collected by Device Farm.
# These files will be available through the ListArtifacts API as your "Customer
 Artifacts".
artifacts:
  # By default, Device Farm will collect your artifacts from the $DEVICEFARM_LOG_DIR
 directory.
  - $DEVICEFARM_LOG_DIR
```

Migrating to the Amazon Linux 2 Test Host in AWS Device Farm

Warning

The legacy Android Test Host will no longer be available on October 21, 2024. Note that the process for deprecation is split across several dates:

- On April 22, 2024, jobs from any new account will be directed to the upgraded test host.
- On September 2, 2024, all new or modified test spec files must target the upgraded test host.

- On October 21, 2024, jobs will no longer be able to run on the legacy test host.

Set your test spec files to the `amazon_linux_2` host to prevent compatibility issues.

To migrate existing tests from the legacy host to the new Amazon Linux 2 host, develop new test spec files based on your pre-existing ones. The recommended approach is to start with the new default test spec files for your test types. Then, migrate relevant commands from your old test spec file to the new one, saving the old file as a backup. This lets you leverage the optimized default spec for the new host while reusing your existing code. It ensures you get the full benefits of the new host configured optimally for your tests, while retaining your legacy test spec for reference as you adapt commands to the new environment.

The following steps can be used to create a new Amazon Linux 2 test spec file while reusing commands from your old test spec file:

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. Navigate to the Device Farm project containing your automation tests.
3. Choose **Create a new test run** in the project.
4. Choose a previously used app and test package for your test framework.
5. Choose **Run your test in a custom environment**.
6. Choose the test spec file you are currently using for tests on the legacy test host from the test spec drop-down menu.
7. Copy the contents of this file and paste them locally in a text editor for reference later on.
8. In the test spec drop-down menu, change your test spec selection to the most recent *default* test spec file.
9. Choose **Edit**, and you will enter the test spec editing interface. You will notice that, in the first few lines of the test spec file, it has already opted in to the new test host:

   ```
   android_test_host: amazon_linux_2
   ```

10. Review the syntax for selecting test hosts here and the key differences between the test hosts here.

11. Selectively add and edit commands from your locally-saved test spec file from step 6 into the new default test spec file. Then, choose **Save as** to save the new spec file. You can now schedule test runs on the Amazon Linux 2 test host.

Differences between the new and legacy test hosts

When you're editing your test spec file to use the Amazon Linux 2 test host and transitioning your tests from the legacy test host, be aware of these key environment differences:

- **Selecting software versions**: In many instances, the default software versions have changed, so if you weren't explicitly selecting your software version in the Legacy test host before, you may want to specify it now in the Amazon Linux 2 test host using `devicefarm-cli`. In the vast majority of use cases, we recommend that customers explicitly select the versions of software they use. By selecting a software version with `devicefarm-cli`, you'll have a predictable and consistent experience with it and receive ample amounts of warnings if Device Farm plans to remove that version from the test host.

  Moreover, software selection tools like `nvm`, `pyenv`, `avm`, and `rvm` have been removed in favor of the new `devicefarm-cli` software selection system.
- **Available software versions**: Many versions of previously pre-installed software have been removed, and many new versions have been added. So, ensure that when using the `devicefarm-cli` to select your software versions, you select versions which are in the supported version list.
- Any **file paths that are hard-coded** in your Legacy host test spec file as absolute paths will most likely not work as expected in the Amazon Linux 2 test host; they're generally not recommended for test spec file use. We recommend that you use relative paths and environment variables for all test spec file code. Moreover, note that most binaries you need for your test can be found in the PATH of the host so that they are immediately runnable from the spec file using just their name (such as appium).
- **Performance data** collection isn't supported on the new test host at this time.
- **Operating System version**: The legacy test host was based on the Ubuntu operating system, whereas the new one is based on Amazon Linux 2. As a result, users may notice some differences in the available system libraries and system library versions.
- For **Appium Java** users, the new test host does not contain any pre-installed JAR files in its class path, whereas the previous host contained one for the TestNG framework (via an environment variable `$DEVICEFARM_TESTNG_JAR`). We recommend that customers package the necessary

JAR files for their test frameworks inside their test package and remove instances of the $DEVICEFARM_TESTNG_JAR variable from their test spec files. For more information, see Working with Appium and AWS Device Farm.

- For **Appium** users, the $DEVICEFARM_CHROMEDRIVER_EXECUTABLE environment variable has been removed in favor of a new approach to enabling customers to access Chromedriver for Android. See our Default test spec file for an example, which uses a new environment variable $DEVICEFARM_CHROMEDRIVER_EXECUTABLE_DIR.

> **Note**
>
> We strongly recommend keeping the existing Appium server command from the default test spec file as is.

We recommend reaching out to the service team through a support case if you have any feedback or questions about the differences between the test hosts from a software perspective.

Environment variables in Device Farm

Environment variables represent values that are used by your automated tests. You can use these environment variables in your YAML files and test code. In a custom test environment, Device Farm dynamically populates environment variables at runtime.

Topics

Common environment variables in Device Farm

This section describes environment variables that are common to Android platform tests and to iOS platform tests in AWS Device Farm. For more information about environment variables in Device Farm, see Environment variables in Device Farm.

Android tests

This section describes custom environment variables common to Android platform tests supported by Device Farm.

`$DEVICEFARM_DEVICE_NAME`

Name of the device on which your tests run. It represents the unique device identifer (UDID) of the device.

`$DEVICEFARM_DEVICE_PLATFORM_NAME`

The device platform name. It is either Android or iOS.

`$DEVICEFARM_DEVICE_OS_VERSION`

The device OS version.

`$DEVICEFARM_APP_PATH`

The path to the mobile app on the host machine where the tests are being executed. The app path is available for mobile apps only.

`$DEVICEFARM_DEVICE_UDID`

The unique identifier of the mobile device running the automated test.

`$DEVICEFARM_LOG_DIR`

The path to the log files generated during the test run. By default, all files in this directory are archived in a ZIP file and made available as an artifact after your test run.

`$DEVICEFARM_SCREENSHOT_PATH`

The path to the screenshots, if any, captured during the test run.

`$DEVICEFARM_CHROMEDRIVER_EXECUTABLE_DIR`

The location of a directory which contains the necessary Chromedriver executables for use in Appium web and hybrid tests.

`$ANDROID_HOME`

The path to the Android SDK installation directory.

> **Note**
>
> The `ANDROID_HOME` environment variable is only available on the Amazon Linux 2 test host for Android.

iOS tests

This section describes custom environment variables common to iOS platform tests supported by Device Farm.

`$DEVICEFARM_DEVICE_NAME`

Name of the device on which your tests run. It represents the unique device identifer (UDID) of the device.

`$DEVICEFARM_DEVICE_PLATFORM_NAME`

The device platform name. It is either Android or iOS.

`$DEVICEFARM_APP_PATH`

The path to the mobile app on the host machine where the tests are being executed. The app path is available for mobile apps only.

`$DEVICEFARM_DEVICE_UDID`

The unique identifier of the mobile device running the automated test.

`$DEVICEFARM_LOG_DIR`

The path to the log files generated during the test run.

`$DEVICEFARM_SCREENSHOT_PATH`

The path to the screenshots, if any, captured during the test run.

Appium Java JUnit environment variables in Device Farm

This section describes environment variables used by the Appium Java JUnit tests in a custom test environment in AWS Device Farm. For more information about environment variables in Device Farm, see Environment variables in Device Farm.

$DEVICEFARM_TESTNG_JAR

The path to the TestNG .jar file.

$DEVICEFARM_TEST_PACKAGE_PATH

The path to the unzipped contents of the test package file.

Appium Java TestNG environment variables in Device Farm

This section describes environment variables used by the Appium Java TestNG tests in a custom test environment in Device Farm. For more information about environment variables in Device Farm, see Environment variables in Device Farm.

$DEVICEFARM_TESTNG_JAR

The path to the TestNG .jar file.

$DEVICEFARM_TEST_PACKAGE_PATH

The path to the unzipped contents of the test package file.

XCUITest environment variables in Device Farm

This section describes environment variables used by the XCUITest test in a custom test environment in Device Farm. For more information about environment variables in Device Farm, see Environment variables in Device Farm.

$DEVICEFARM_XCUITESTRUN_FILE

Path to the Device Farm `.xctestun` file. It is generated from your app and test packages.

$DEVICEFARM_DERIVED_DATA_PATH

Expected path of Device Farm xcodebuild output.

$DEVICEFARM_XCTEST_BUILD_DIRECTORY

The path to the unzipped contents of the test package file.

Migrating tests from a standard to custom test environment

You can switch from a standard test execution mode to a custom execution mode in AWS Device Farm. Migration primarily involves two different forms of execution:

1. **Standard mode**: This test execution mode is primarily built to provide customers with granular reporting and a fully-managed environment.
2. **Custom mode**: This test execution mode is built for different use cases that require faster test runs, the ability to lift and shift and achieve parity with their local environment, and live video streaming.

For more information about standard and custom modes in Device Farm, see [Test environments in AWS Device Farm] and [Custom test environments in AWS Device Farm].

Considerations when migrating

This section lists some of the prominent use cases to consider when migrating to the custom mode:

1. **Speed**: In the standard mode of execution, Device Farm parses the metadata of the tests that you have packaged and uploaded using the packaging instructions for your particular framework. Parsing detects the number of tests in your package. Thereafter, Device Farm runs each test separately and presents the logs, videos, and other result artifacts individually for each test. However, this steadily adds to the total end-to-end test execution time as there are the pre- and post-processing of tests and result artifacts on the service end.

   By contrast, the custom mode of execution doesn't parse your test package; this means no pre-processing and minimal post-processing for tests or result artifacts. This results in total end-to-end execution times close to your local setup. The tests are executed in the same format as they would be if they ran on your local machine(s). The results of the tests are the same as what you get locally and are available for download at the end of the job execution.
2. **Customization or Flexibility**: The standard mode of execution parses your test package to detect the number of tests and then runs each test separately. Note that there is no guarantee that the tests will run in the order you specified. As a result, tests requiring a particular sequence of execution may not work as expected. In addition, there is no way to customize the host machine environment or pass config files that may be needed to run your tests a certain way.

   By contrast, custom mode lets you configure the host machine environment including the ability to install additional software, pass filters to your tests, pass config files, and control the test execution setup. It achieves this via a yaml file (also called the testspec file) that you can modify by adding shell commands to it. This yaml file gets converted to a shell script which gets executed on the test host machine. You can save multiple yaml files and choose one dynamically as per your requirements when you schedule a run.

3. **Live video and logging**: Both standard and custom modes of execution provide you with videos and logs for your tests. However, in standard mode, you get the video and predefined logs of your tests only after your tests are completed.

   By contrast, custom mode gives you a live stream of the video and client-side logs of your tests. In addition, you are able to download the video and other artifacts at the end of the test(s).

> **Tip**
>
> If your use case involves at least one of the factors above, we strongly recommend switching to the custom mode of execution.

Migration steps

To migrate from standard to custom mode, do the following:

1. Sign in to the AWS Management Console and open the Device Farm console at https://console.aws.amazon.com/devicefarm/.
2. Choose your project and then start a new automation run.
3. Upload your app (or select `web app`), choose your test framework type, upload your test package, then under the `Choose your execution environment` parameter, choose the option to `Run your test in a custom environment`.
4. By default, Device Farm's example test spec file will appear for you to view and edit. This example file can be used as a starting place to try your tests out in custom environment mode. Then, once you've verified that your tests are working properly from the console, you can then alter any of your API, CLI, and pipeline integrations with Device Farm to use this test spec file as a parameter when scheduling test runs. For information on how to add a test spec file as a parameter for your runs, see the `testSpecArn` parameter section for the `ScheduleRun` API in our API guide.

Appium framework

In a custom test environnment, Device Farm does not insert or override any Appium capabilities in your Appium framework tests. You must specify your test's Appium capabilities in either the test spec YAML file or your test code.

Android instrumentation

You do not need to make changes to move your Android instrumentation tests to a custom test environment.

iOS XCUITest

You do not need to make changes to move your iOS XCUITest tests to a custom test environment.

Extending custom test environments in Device Farm

AWS Device Farm enables configuring a custom environment for automated testing (custom mode), which is the recommended approach for all Device Farm users. The Device Farm custom mode enables you to run more than just your test suite. In this section, you learn how to extend your test suite and optimize your tests.

For more information about custom test environments in Device Farm, see Custom test environments in AWS Device Farm.

Topics

- Setting a device PIN when running tests in Device Farm
- Speeding up Appium-based tests in Device Farm through desired capabilities
- Using Webhooks and other APIs after your tests run in Device Farm
- Adding extra files to your test package in Device Farm

Setting a device PIN when running tests in Device Farm

Some applications require that you set a PIN on the device. Device Farm does not support setting a PIN on devices natively. However, this is possible with the following caveats:

- The device must be running Android 8 or above.

- The PIN must be removed after the test is complete.

To set the PIN in your tests, use the `pre_test` and `post_test` phases to set and remove the PIN, as shown following:

```
phases:
    pre_test:
      - # ... among your pre_test commands
      - DEVICE_PIN_CODE="1234"
      - adb shell locksettings set-pin "$DEVICE_PIN_CODE"
    post_test:
      - # ... Among your post_test commands
      - adb shell locksettings clear --old "$DEVICE_PIN_CODE"
```

When your test suite begins, the PIN 1234 is set. After your test suite exits, the PIN is removed.

Warning

If you don't remove the PIN from the device after the test is complete, the device and your account will be quarantined.

For more ways to extend your test suite and optimize your tests, see [Extending custom test environments in Device Farm](#).

Speeding up Appium-based tests in Device Farm through desired capabilities

When using Appium, you might find that the standard mode test suite is very slow. This is because Device Farm applies the default settings and doesn't make any assumptions about how you want to use the Appium environment. While these defaults are built around industry best practices, they might not apply to your situation. To fine-tune the parameters of the Appium server, you can adjust the default Appium capabilities in your test spec. For example, the following sets the `usePrebuildWDA` capability to `true` in an iOS test suite to speed up initial start time:

```
phases:
  pre_test:
    - # ... Start up Appium
    - >-
    appium --log-timestamp
    --default-capabilities "{\"usePrebuiltWDA\": true, \"derivedDataPath\":
\"$DEVICEFARM_WDA_DERIVED_DATA_PATH\",
    \"deviceName\": \"$DEVICEFARM_DEVICE_NAME\", \"platformName\":
\"$DEVICEFARM_DEVICE_PLATFORM_NAME\", \"app\":\"$DEVICEFARM_APP_PATH\",
    \"automationName\":\"XCUITest\", \"udid\":\"$DEVICEFARM_DEVICE_UDID_FOR_APPIUM\",
 \"platformVersion\":\"$DEVICEFARM_DEVICE_OS_VERSION\"}"
    >> $DEVICEFARM_LOG_DIR/appiumlog.txt 2>&1 &
```

Appium capabilities must be a shell-escaped, quoted JSON structure.

The following Appium capabilities are common sources of performance improvements:

`noReset` and `fullReset`

These two capabilities, which are mutually exclusive, describe the behavior of Appium after each session is complete. When `noReset` is set to `true`, the Appium server doesn't remove data from your application when an Appium session ends, effectively doing no cleanup whatsoever. `fullReset` uninstalls and clears all application data from the device after the session has closed. For more information, see [Reset Strategies](#) in the Appium documentation.

`ignoreUnimportantViews` (Android only)

Instructs Appium to compress the Android UI hierarchy only to *relevant* views for the test, speeding up certain element lookups. However, this can break some XPath-based test suites because the hierarchy of the UI layout has been changed.

`skipUnlock` (Android only)

Informs Appium that there is no PIN code currently set, which speeds up tests after a screen off event or other lock event.

`webDriverAgentUrl` (iOS only)

Instructs Appium to assume that an essential iOS dependency, `webDriverAgent`, is already running and available to accept HTTP requests at the specified URL. If `webDriverAgent` isn't already up and running, it can take Appium some time at the beginning of a test suite to start the webDriverAgent. If you start webDriverAgent yourself and set webDriverAgentUrl to http://localhost:8100 when starting Appium, you can boot up your test suite faster. Note that this capability should never be used alongside the useNewWDA capability.

You can use the following code to start webDriverAgent from your test spec file on the device's local port 8100, then forward it to the test host's local port 8100 (this allows you to set webDriverAgentUrl's value to http://localhost:8100). This code should be run during the install phase after any code for setting up the Appium and webDriverAgent environment variables has been defined:

```
      # Start WebDriverAgent and iProxy
      - >-
        xcodebuild test-without-building -project /usr/local/avm/versions/
$APPIUM_VERSION/node_modules/appium/node_modules/appium-webdriveragent/
WebDriverAgent.xcodeproj
        -scheme WebDriverAgentRunner -derivedDataPath
 $DEVICEFARM_WDA_DERIVED_DATA_PATH
        -destination id=$DEVICEFARM_DEVICE_UDID_FOR_APPIUM
 IPHONEOS_DEPLOYMENT_TARGET=$DEVICEFARM_DEVICE_OS_VERSION
        GCC_TREAT_WARNINGS_AS_ERRORS=0 COMPILER_INDEX_STORE_ENABLE=NO >>
 $DEVICEFARM_LOG_DIR/webdriveragent_log.txt 2>&1 &

        iproxy 8100 8100 >> $DEVICEFARM_LOG_DIR/iproxy_log.txt 2>&1 &
```

Then, you can add the following code to your test spec file to ensure that webDriverAgent started successfully. This code should be run at the end of the pre-test phase after ensuring that Appium started successfully:

```
      # Wait for WebDriverAgent to start
      - >-
        start_wda_timeout=0;
        while [ true ];
        do
          if [ $start_wda_timeout -gt 60 ];
          then
              echo "WebDriverAgent server never started in 60 seconds.";
              exit 1;
          fi;
          grep -i "ServerURLHere" $DEVICEFARM_LOG_DIR/webdriveragent_log.txt >> /
dev/null 2>&1;
          if [ $? -eq 0 ];
```

```
            then
                echo "WebDriverAgent REST http interface listener started";
                break;
            else
                echo "Waiting for WebDriverAgent server to start. Sleeping for 1
seconds";
                sleep 1;
                start_wda_timeout=$((start_wda_timeout+1));
            fi;
        done;
```

For more information on the capabilities that Appium supports, see Appium Desired Capabilities in the Appium documentation.

For more ways to extend your test suite and optimize your tests, see Extending custom test environments in Device Farm.

Using Webhooks and other APIs after your tests run in Device Farm

You can have Device Farm call a webhook after every test suite finishes using **curl**. The process to do this varies with the destination and formatting. For your specific webhook, see the documentation for that webhook. The following example posts a message each time a test suite has finished to a Slack webhook:

```
phases:
  post_test:
    - curl -X POST -H 'Content-type: application/json' --data '{"text":"Tests on
 '$DEVICEFARM_DEVICE_NAME' have finished!"}' https://hooks.slack.com/services/
T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXXX
```

For more information on using webhooks with Slack, see Sending your first Slack message using Webhook in the Slack API reference.

For more ways to extend your test suite and optimize your tests, see Extending custom test environments in Device Farm.

You are not limited to using **curl** to call webhooks. Test packages can include extra scripts and tools, as long as they are compatible with the Device Farm execution environment. For example, your test package may include auxiliary scripts that make requests to other APIs. Make sure that any required packages are installed alongside your test suite's requirements. To add a script that runs after your test suite is complete, include the script in your test package and add the following to your test spec:

```
phases:
  post_test:
    - python post_test.py
```

Note

Maintaining any API keys or other authentication tokens used in your test package is your responsibility. We recommend that you keep any form of security credential out of source control, use credentials with the fewest possible privileges, and use revokable, short-lived tokens whenever possible. To verify security requirements, see the documentation for the third-party APIs that you use.

If you plan on using AWS services as a part of your test execution suite, you should use IAM temporary credentials, generated outside of your test suite and included in your test package. These credentials should have the fewest granted permissions and shortest lifespan possible. For more information on creating temporary credentials, see Requesting temporary security credentials in the *IAM User Guide*.

For more ways to extend your test suite and optimize your tests, see Extending custom test environments in Device Farm.

Adding extra files to your test package in Device Farm

You may want to use additional files as a part of your tests either as extra configuration files or additional test data. You can add these additional files to your test package before uploading it to AWS Device Farm, then access them from the custom environment mode. Fundamentally, all test package upload formats (ZIP, IPA, APK, JAR, etc.) are package archive formats that support standard ZIP operations.

You can add files to your test archive before uploading it to AWS Device Farm by using the following command:

```
$ zip zip-with-dependencies.zip extra_file
```

For a directory of extra files:

```
$ zip -r zip-with-dependencies.zip extra_files/
```

These commands work as expected for all test package upload formats except for IPA files. For IPA files, especially when used with XCUITests, we recommend that you put any extra files in a slightly different location due to how AWS Device Farm resigns iOS test packages. When building your iOS test, the test application directory will be located inside of another directory named *Payload*.

For example, this is how one such iOS test directory may look:

```
$ tree
.
### Payload
    ### ADFiOSReferenceAppUITests-Runner.app
        ### ADFiOSReferenceAppUITests-Runner
        ### Frameworks
        #   ### XCTAutomationSupport.framework
        #   #   ### Info.plist
        #   #   ### XCTAutomationSupport
        #   #   ### _CodeSignature
        #   #   #   ### CodeResources
        #   #   ### version.plist
        #   ### XCTest.framework
        #       ### Info.plist
        #       ### XCTest
        #       ### _CodeSignature
        #       #   ### CodeResources
        #       ### en.lproj
        #       #   ### InfoPlist.strings
        #       ### version.plist
        ### Info.plist
        ### PkgInfo
        ### PlugIns
        #   ### ADFiOSReferenceAppUITests.xctest
        #   #   ### ADFiOSReferenceAppUITests
        #   #   ### Info.plist
        #   #   ### _CodeSignature
        #   #       ### CodeResources
        #   ### ADFiOSReferenceAppUITests.xctest.dSYM
        #       ### Contents
        #           ### Info.plist
        #           ### Resources
```

```
        #               ### DWARF
        #                   ### ADFiOSReferenceAppUITests
        ### _CodeSignature
        #   ### CodeResources
        ### embedded.mobileprovision
```

For these XCUITest packages, add any extra files to the directory ending in *.app* inside of the *Payload* directory. For example, the following commands show how you can add a file to this test package:

```
$ mv extra_file Payload/*.app/
$ zip -r my_xcui_tests.ipa Payload/
```

When you add a file to your test package, you can expect slightly different interaction behavior in AWS Device Farm based on its upload format. If the upload used the ZIP file extension, AWS Device Farm will automatically unzip the upload before your test and leave the unzipped files at the location with the *$DEVICEFARM_TEST_PACKAGE_PATH* environment variable. (This means that if you added a file called *extra_file* to the root of the archive as in the first example, it would be located at *$DEVICEFARM_TEST_PACKAGE_PATH/extra_file* during the test).

To use a more practical example, if you're an Appium TestNG user who wants to include a *testng.xml* file with your test, you can include it in your archive using the following command:

```
$ zip zip-with-dependencies.zip testng.xml
```

Then, you can change your test command in the custom environment mode to the following:

```
java -D appium.screenshots.dir=$DEVICEFARM_SCREENSHOT_PATH org.testng.TestNG -testjar
 *-tests.jar -d $DEVICEFARM_LOG_DIR/test-output $DEVICEFARM_TEST_PACKAGE_PATH/
testng.xml
```

If your test package upload extension isn't ZIP (e.g., APK, IPA, or JAR file), the uploaded package file itself is found at *$DEVICEFARM_TEST_PACKAGE_PATH*. Because these are still archive format files, you can unzip the file in order to access the additional files from within. For example, the following command will unzip the contents of the test package (for APK, IPA, or JAR files) to the */tmp* directory:

```
unzip $DEVICEFARM_TEST_PACKAGE_PATH -d /tmp
```

In the case of an APK or JAR file, you would find your extra files unzipped to the `/tmp` directory (e.g., `/tmp/extra_file`). In the case of an IPA file, as explained before, extra files would be in a slightly different location inside the folder ending in `.app`, which is inside of the `Payload` directory. For example, based on the IPA example above, the file would be found at the location `/tmp/Payload/ADFiOSReferenceAppUITests-Runner.app/extra_file` (referenceable as `/tmp/Payload/*.app/extra_file`).

For more ways to extend your test suite and optimize your tests, see Extending custom test environments in Device Farm.

Remote access in AWS Device Farm

Remote access allows you to swipe, gesture, and interact with a device through your web browser in real time to test functionality and reproduce customer issues. You interact with a specific device by creating a remote access session with that device.

A session in Device Farm is a real-time interaction with an actual, physical device hosted in a web browser. A session displays the single device you select when you start the session. A user can start more than one session at a time with the total number of simultaneous devices limited by the number of device slots you have. You can purchase device slots based on the device family (Android or iOS devices). For more information, see Device Farm Pricing.

Device Farm currently offers a subset of devices for remote access testing. New devices are added to the device pool all the time.

Device Farm captures video of each remote access session and generates logs of activity during the session. These results include any information you provide during a session.

> **Note**
>
> For security reasons, we recommend that you avoid providing or entering sensitive information, such as account numbers, personal login information, and other details during a remote access session. If possible, use alternatives developed specifically for testing, such as test accounts.

Topics

Creating a remote access session in AWS Device Farm

For information about remote access sessions, see Sessions.

- Prerequisites
- Create a test run (console)

- Next steps

Prerequisites

- Create a project in Device Farm. Follow the instructions in Creating a project in AWS Device Farm, and then return to this page.

Create a session with the 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 **Mobile Device Testing**, then choose **Projects**.
3. If you already have a project, choose it from the list. Otherwise, create a project by following the instructions in Creating a project in AWS Device Farm.
4. On the **Remote access** tab, choose **Start a new session**.
5. Choose a device for your session. You can choose from the list of available devices or search for a device using the search bar at the top of the list. You can search by:
    - Name
    - Platform
    - Form factor
    - Fleet type
6. In **Session name**, enter a name for the session.
7. Choose **Confirm and start session**.

Next steps

Device Farm starts the session as soon as the requested device is available, typically within a few minutes. The **Device Requested** dialog box appears until the session starts. To cancel the session request, choose **Cancel request**.

After a session starts, if you should close the browser or browser tab without stopping the session or if the connection between the browser and the internet is lost, the session remains active for five minutes. After that, Device Farm ends the session. Your account is charged for the idle time.

After the session starts, you can interact with the device in the web browser.

Using a remote access session in AWS Device Farm

For information about performing interactive testing of Android and iOS apps through remote access sessions, see Sessions.

- Prerequisites
- Use a session in the Device Farm console
- Next steps
- Tips and tricks

Prerequisites

- Create a session. Follow the instructions in Creating a session, and then return to this page.

Use a session in the Device Farm console

As soon as the device that you requested for a remote access session becomes available, the console displays the device screen. The session has a maximum length of 150 minutes. The time remaining in the session appears in the **Time Left** field near the device name.

Installing an application

To install an application on the session device, in **Install applications**, select **Choose File**, and then choose the .apk file (Android) or the .ipa file (iOS) that you want to install. Applications that you run in a remote access session don't require any test instrumentation or provisioning.

> **Note**
>
> AWS Device Farm doesn't display a confirmation after an app is installed. Try interacting with the app icon to see if the app is ready to use.
> When you upload an app, there's sometimes a delay before the app is available. Look at the system tray to determine whether the app is available.

Controlling the device

You can interact with the device displayed in the console as you would the actual physical device, by using your mouse or a comparable device for touch and the device's on-screen keyboard. For Android devices, there are buttons in **View controls** that function like the **Home** and **Back** buttons on an Android device. For iOS devices, there is a **Home** button that functions like the home button on an iOS device. You can also switch between applications running on the device by choosing **Recent Apps**.

Switching between portrait and landscape mode

You can also switch between portrait (vertical) and landscape (horizontal) mode for the devices that you're using.

Next steps

Device Farm continues the session until you stop it manually or the 150-minute time limit is reached. To end the session, choose **Stop Session**. After the session stops, you can access the video that was captured and the logs that were generated. For more information, see Retrieving session results from remote access sessions.

Tips and tricks

You might experience performance issues with the remote access session in some AWS Regions. This is due, in part, to latency in some Regions. If you experience performance issues, give the remote access session a chance to catch up before you interact with the app again.

Retrieving the results of a remote access session in AWS Device Farm

For information about sessions, see Sessions.

- Prerequisites
- Viewing session details
- Downloading session video or logs

Prerequisites

- Complete a session. Follow the instructions in Using a remote access session in AWS Device Farm, and then return to this page.

Viewing session details

When a remote access session ends, the Device Farm console displays a table that contains details about activity during the session. For more information, see Analyzing Log Information.

To return to the details of a session at a later time:

1. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
2. Choose the project containing the session.
3. Choose **Remote access**, and then choose the session you want to review from the list.

Downloading session video or logs

When a remote access session ends, the Device Farm console provides access to a video capture of the session and activity logs. In the session results, choose the **Files** tab for a list of links to the session video and logs. You can view these files in the browser or save them locally.

Private devices in AWS Device Farm

A private device is a physical mobile device that AWS Device Farm deploys on your behalf in an Amazon data center. This device is exclusive to your AWS account.

> **Note**
>
> Currently, private devices are available only in the AWS US West (Oregon) Region (`us-west-2`).

If you have a private device fleet, you can create remote access sessions and schedule test runs with your private devices. For more information, see Creating a test run or starting a remote access session in AWS Device Farm. You can also create instance profiles to control the behavior of your private devices during a remote access session or a test run. For more information, see Creating an instance profile in AWS Device Farm. Optionally, you can request that certain Android private devices be deployed as rooted devices.

You can also create an Amazon Virtual Private Cloud endpoint service to test private apps that your company has access to, but are not reachable through the internet. For example, you might have a web application running in your VPC that you want to test on mobile devices. For more information, see Using Amazon VPC endpoint services with Device Farm - Legacy (not recommended).

If you're interested in using a fleet of private devices, contact us. The Device Farm team must work with you to set up and deploy a fleet of private devices for your AWS account.

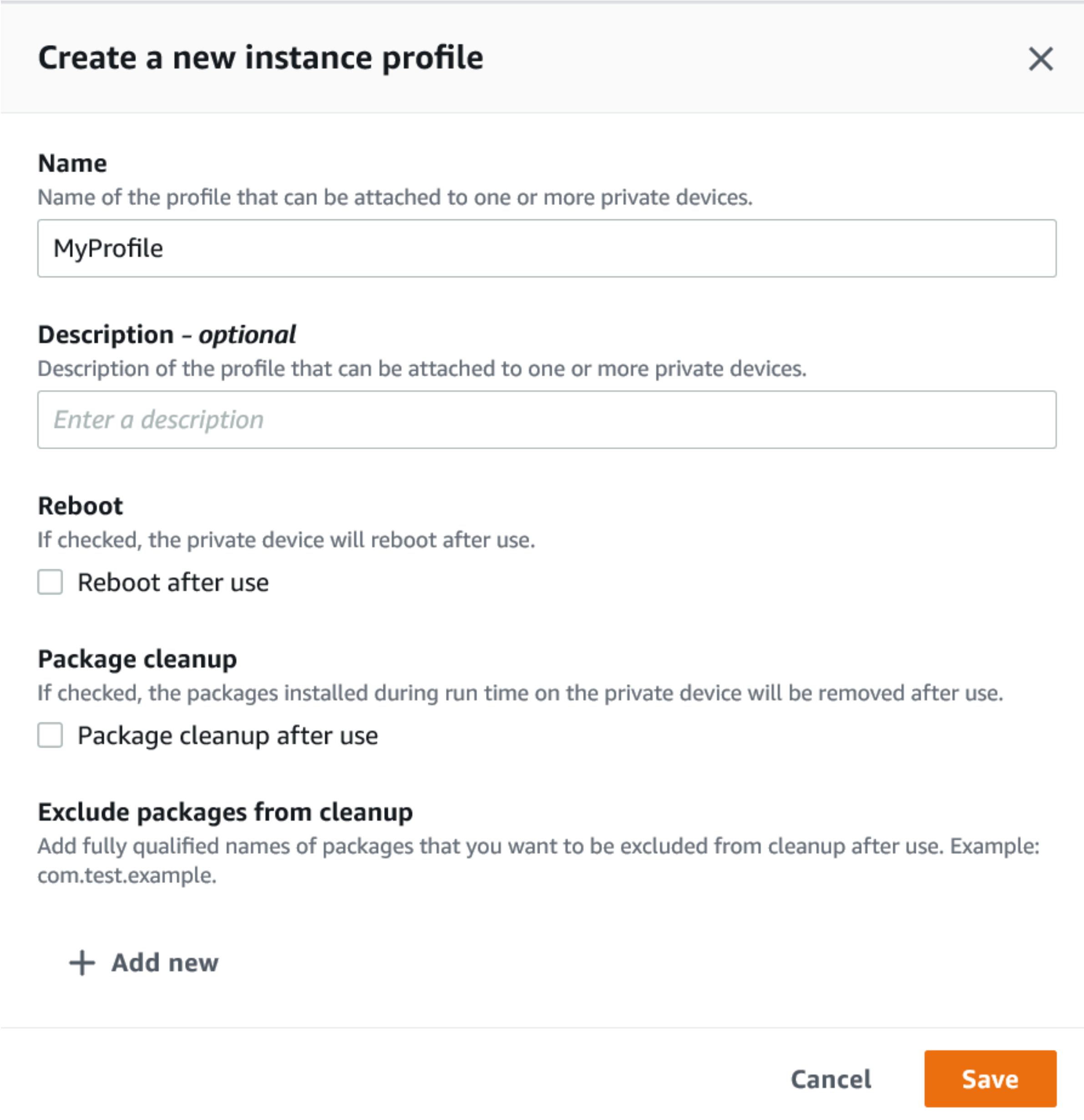
Topics

Creating an instance profile in AWS Device Farm

You can set up a fleet that contains one or more private devices. These devices are dedicated to your AWS account. After you set up the devices, you can optionally create one or more instance profiles for them. Instance profiles can help you automate test runs and consistently apply the same settings to device instances. Instances profiles can also help you control the behavior of remote access session. For more information about private devices in Device Farm, see Private devices in AWS Device Farm.

To create an instance

1. Open the Device Farm console at https://console.aws.amazon.com/devicefarm/.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Private devices**.
3. Choose **Instance profiles**.
4. Choose **Create instance profile**.
5. Enter a name for the instance profile.



6. (Optional) Enter a description for the instance profile.
7. (Optional) Change any of the following settings to specify which actions you want Device Farm to take on a device after each test run or session ends:
   - **Reboot after use** – To reboot the device, select this check box. By default, this check box is cleared (`false`).
   - **Package cleanup** – To remove all the app packages that you installed on the device, select this check box. By default, this check box is cleared (`false`). To keep all the app packages that you installed on the device, leave this check box cleared.

- **Exclude packages from cleanup –** To keep only selected app packages on the device, select the **Package Cleanup** check box, and then choose **Add new**. For the package name, enter the fully qualified name of the app package that you want to keep on the device (for example, `com.test.example`). To keep more app packages on the device, choose **Add new**, and then enter the fully qualified name of each package.

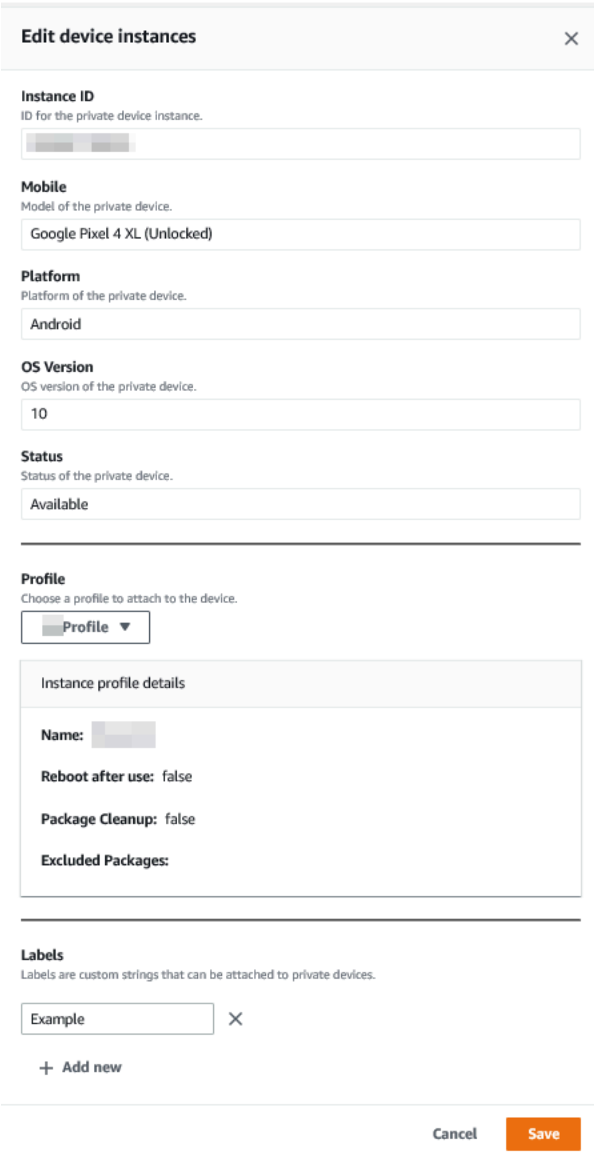
8. Choose **Save**.

Request additional private devices in AWS Device Farm

In AWS Device Farm, you can request an additional private device instances to be added to your fleet. You can also view and change the settings of existing private device instances in your fleet. For more information about private devices, see Private devices in AWS Device Farm.

To request additional private devices or change their settings

1. Open the Device Farm console at https://console.aws.amazon.com/devicefarm/.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Private devices**.
3. Choose **Device instances**. The **Device instances** tab displays a table of the private devices that are in your fleet. To quickly search or filter the table, enter search terms in the search bar above the columns.
4. To request a new private device instance, choose **Request device instance** or contact us. Private devices require additional setup with help from the Device Farm team.
5. In the table of device instances, choose the toggle option next to the instance that you want to view information about or manage, then choose **Edit**.



6. To attach an instance profile to the device instance, choose it from the **Profile** drop-down list. Attaching an instance profile can be helpful if you want to always exclude a specific app package from cleanup tasks, for example. For more information about using instance profiles with devices, see Creating an instance profile in AWS Device Farm.

7. (Optional) Under **Labels**, choose **Add new** to add a label to the device instance. Labels can help you categorize your devices and find specific devices more easily.

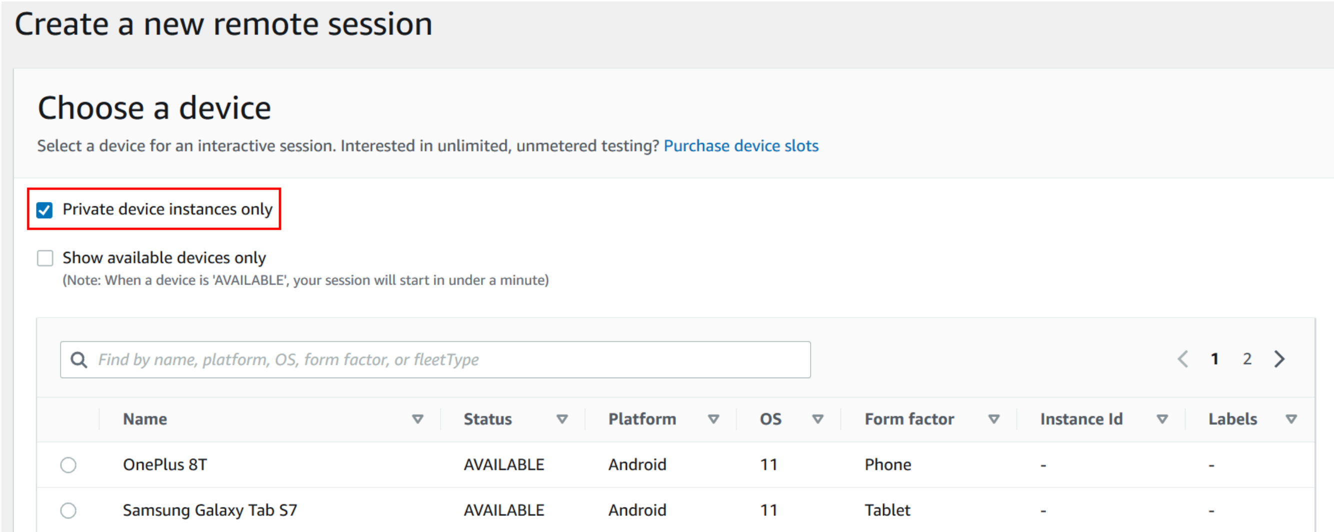
8. Choose **Save**.

Creating a test run or starting a remote access session in AWS Device Farm

In AWS Device Farm, after you set up a private device fleet, you can create test runs or start remote access sessions with one or more private devices in your fleet. For more information about private devices, see Private devices in AWS Device Farm.

To create a test run or start a remote access session

1. Open the Device Farm console at https://console.aws.amazon.com/devicefarm/.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. Choose an existing project from the list or create a new one. To create a new project, choose **New project**, enter a name for the project, and then choose **Submit**.
4. Do one of the following:
   - To create a test run, choose **Automated tests**, and then choose **Create a new run**. The wizard guides you through the steps to create the run. For the **Select devices** step, you can edit an existing device pool or create a new device pool that includes only those private devices that the Device Farm team set up and associated with your AWS account. For more information, see the section called "Create a private device pool".
   - To start a remote access session, choose **Remote access**, and then choose **Start a new session**. On the **Choose a device** page, select **Private device instances only** to limit the list to only those private devices that the Device Farm team set up and associated with your AWS account. Then, choose the device that you want to access, enter a name for the remote access session, and choose **Confirm and start session**.



Selecting private devices in a device pool in AWS Device Farm

To use private devices in your test run, you can create a device pool that selects your private devices. Device pools enable you to select private devices primarily through three types of device pool rules:

1. Rules based on the device ARN
2. Rules based on the device instance label
3. Rules based on the device instance ARN

In the following sections, each rule type and their use cases are described in depth. You can use the Device Farm console, AWS Command Line Interface (AWS CLI), or the Device Farm API to create or modify a device pool with private devices using these rules.

Topics

Device ARN

A device ARN is an identifier representing a type of device rather than any specific physical device instance. A device type is defined by the following attributes:

- The device’s fleet ID
- The device’s OEM
- The device’s model number
- The device’s operating system version
- The device's state that indicates whether it's rooted or not

Many physical device instances can be represented by a single device type where every instance of that type has the same values for these attributes. For example, if you had three *Apple iPhone 13* devices on iOS version *16.1.0* in your private fleet, each device would share the same device ARN. If any devices were added or removed from your fleet with these same attributes, the device ARN would continue to represent whatever available devices you had in your fleet for that device type.

The device ARN is the most robust way of selecting private devices for a device pool because it allows the device pool to continue selecting devices regardless of the specific device instances you have deployed at any given time. Individual private device instances can experience hardware failures, prompting Device Farm to automatically replace them with new working instances of the same device type. In these scenarios, the device ARN rule ensures that your device pool can continue to select devices in the event of a hardware failure.

When you use a device ARN rule for private devices in your device pool and schedule a test run with that pool, Device Farm will automatically check which private device instances are represented by that device ARN. Of the instances that are currently available, one of them will be assigned to run your test. If no instances are currently available, Device Farm will wait for the first available instance of that device ARN to become available, and assign it to run your test.

Device instance labels

A device instance label is a textual identifier you can attach as metadata for a device instance. You can attach multiple labels to each device instance and the same label to multiple device instances. For more information about adding, modifying, or removing device labels from device instances, see Managing private devices.

The device instance label can be a robust way of selecting private devices for a device pool because, if you have multiple device instances with the same label, then it allows the device pool to select from any one of them for your test. If the device ARN isn't a good rule for your use case (for example, if you want to select from devices of multiple device types, or if you want to select from a subset of all devices of a device type), then device instance labels can enable you to select from multiple devices for your device pool with greater granularity. Individual private device instances can experience hardware failures, prompting Device Farm to automatically replace them with new working instances of the same device type. In these scenarios, the replacement device instance will not retain any instance label metadata of the replaced device. So, if you apply the same device instance label to multiple device instances, then the device instance label rule ensures that your device pool can continue to select device instances in the event of a hardware failure.

When you use a device instance label rule for private devices in your device pool and schedule a test run with that pool, Device Farm will automatically check which private device instances are represented by that device instance label, and of those instances, randomly select one which is available to run your test. If none are available, Device Farm will randomly select any device instance with the device instance label to run your test and queue the test to run on the device once it's available.

Instance ARN

A device instance ARN is an identifier representing a physical bare metal device instance deployed in a private fleet. For example, if you had three `iPhone 13` devices on OS `15.0.0` in your private fleet, while each device would share the same device ARN, each device would also have its own instance ARN representing that instance alone.

The device instance ARN is the least robust way to select private devices for a device pool and is only recommended if the device ARNs and device instance labels don't fit your use case. Device instance ARNs are often used as rules for device pools when a specific device instance is configured in a unique and specific way as a prerequisite for your test and if that configuration needs to be known and verified before the test is ran on it. Individual private device instances can experience hardware failures, prompting Device Farm to automatically replace them with new working instances of the same device type. In these scenarios, the replacement device instance will have a different device instance ARN than the replaced device. So, if you rely on device instance ARNs for your device pool, then you'll need to manually change your device pool's rule definition from using the old ARN to using the new ARN. If you do need to manually preconfigure the device for its test, then this can be an effective workflow (compared to device ARNs). For testing at scale, it is recommended to try to adapt these use cases to work with device instance labels and if possible, have multiple device instances preconfigured for testing.

When you use a device instance ARN rule for private devices in your device pool and schedule a test run with that pool, Device Farm will automatically assign that test to that device instance. If that device instance isn't available, Device Farm will queue the test on the device once it's available.

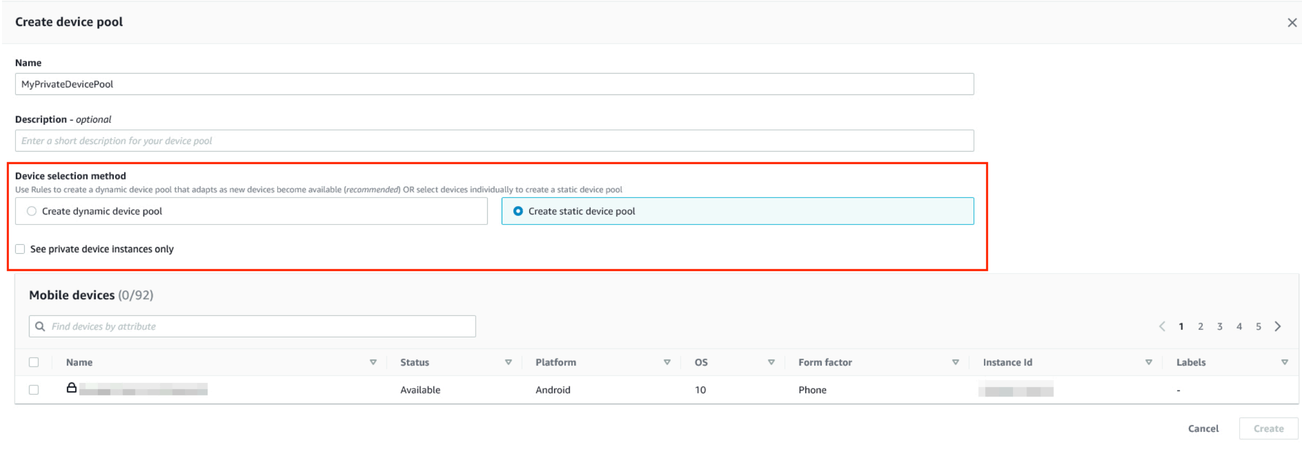
Creating a private device pool with private devices (console)

When you create a test run, you can create a device pool for the test run and ensure that the pool includes only your private devices.

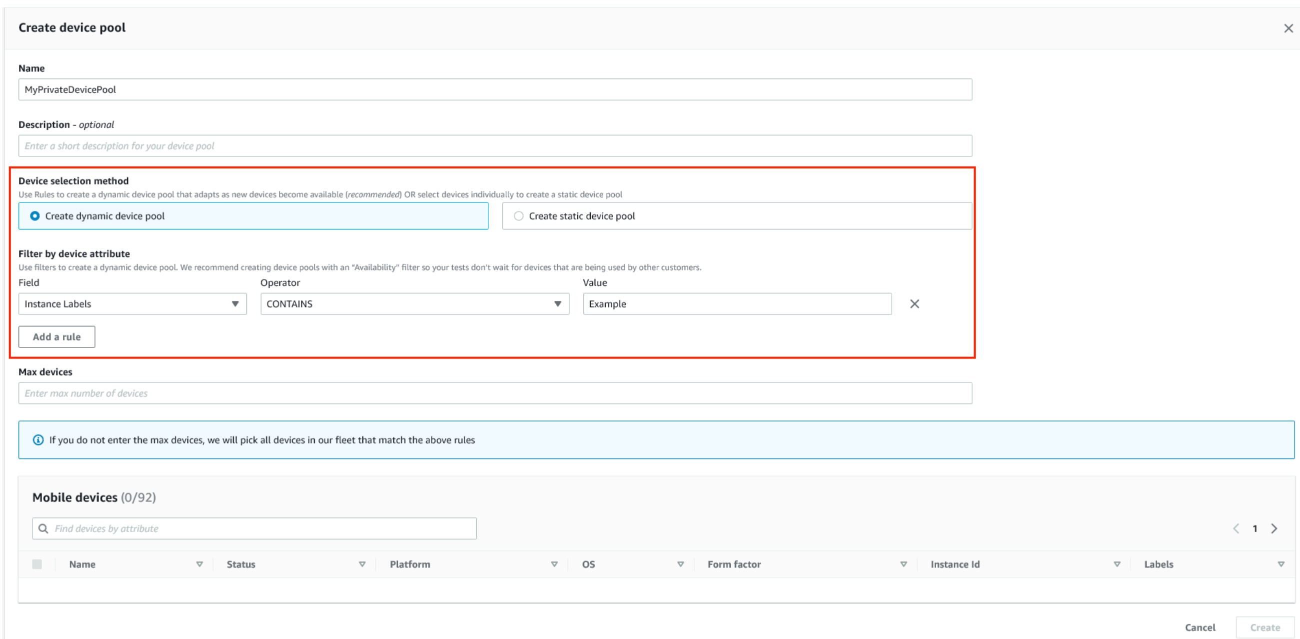
> **Note**
>
> When creating a device pool with private devices in the console, you can only use any one of the three available rules for selecting private devices. If you want to create a device pool that contains multiple types of rules for private devices (for example, device pools that contain rules for device ARNs and device instance ARNs), then you need to create the pool through the CLI or API.

1. Open the Device Farm console at https://console.aws.amazon.com/devicefarm/.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. Choose an existing project from the list or create a new one. To create a new project, choose **New project**, enter a name for the project, and then choose **Submit**.
4. Choose **Automated tests**, and then choose **Create a new run**. The wizard guides you through the steps to choose your application and configure the test that you want to run.
5. For the **Select devices** step, choose **Create device pool**, and enter a name and optional description for your device pool.

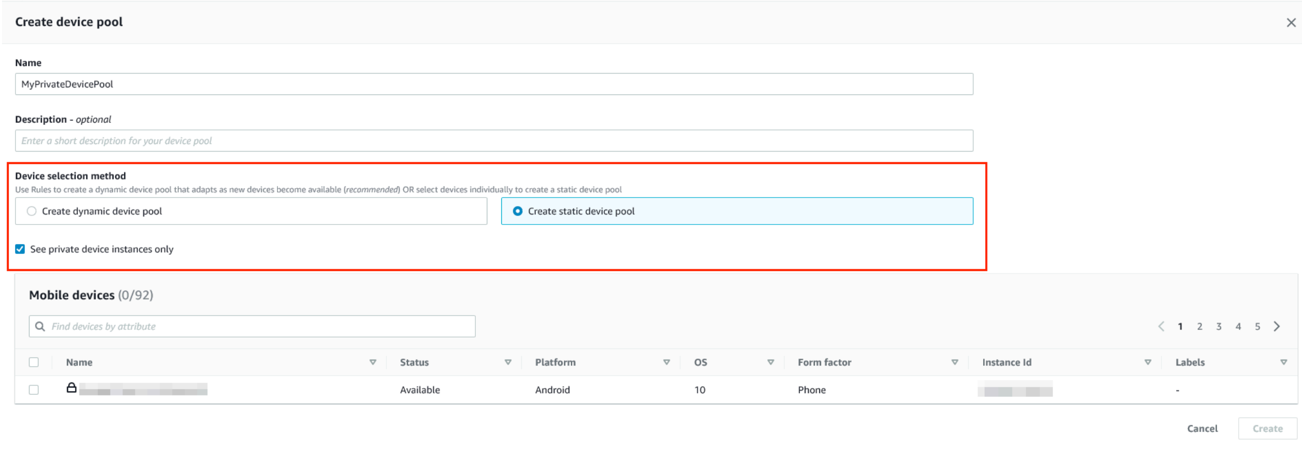
    a. To use device ARN rules for your device pool, choose **Create static device pool**, then select the specific device types from the list that you would like to use in the device pool. Do not select **Private device instances only** because this option causes the device pool to be created with device instance ARN rules (instead of device ARN rules).



b. To use device instance label rules for your device pool, choose **Create dynamic device pool**. Then, for each label you would like to use in the device pool, choose **Add a rule**. For each rule, choose **Instance Labels** as the `Field`, choose **Contains** as the `Operator`, and specify your desired device instance label as the `Value`.



c. To use device instance ARN rules for your device pool, choose **Create static device pool**, then select **Private device instances only** to limit the list of devices to only those private device instances that Device Farm has associated with your AWS account.



6. Choose **Create**.

Creating a private device pool with private devices (AWS CLI)

- Run the **create-device-pool** command.

For information about using Device Farm with the AWS CLI, see AWS CLI reference.

Creating a private device pool with private devices (API)

- Call the `CreateDevicePool` API.

For information about using the Device Farm API, see Automating Device Farm.

Skipping app re-signing on private devices in AWS Device Farm

App signing is a process that involves digitally signing an app package (e.g., APK, IPA) with a private key before it can be installed on a device or published to an app store like the Google Play Store or the Apple App Store. To streamline testing by reducing the number of signatures and profiles needed and increase data security on remote devices, AWS Device Farm will re-sign your app after it has been uploaded to the service.

Once you upload your app to AWS Device Farm, the service will generate a new signature for the app using its own signing certificates and provisioning profiles. This process replaces the original app signature with AWS Device Farm's signature. The re-signed app is then installed on the test devices provided by AWS Device Farm. The new signature allows the app to be installed and run on these devices without the need for the original developer's certificates.

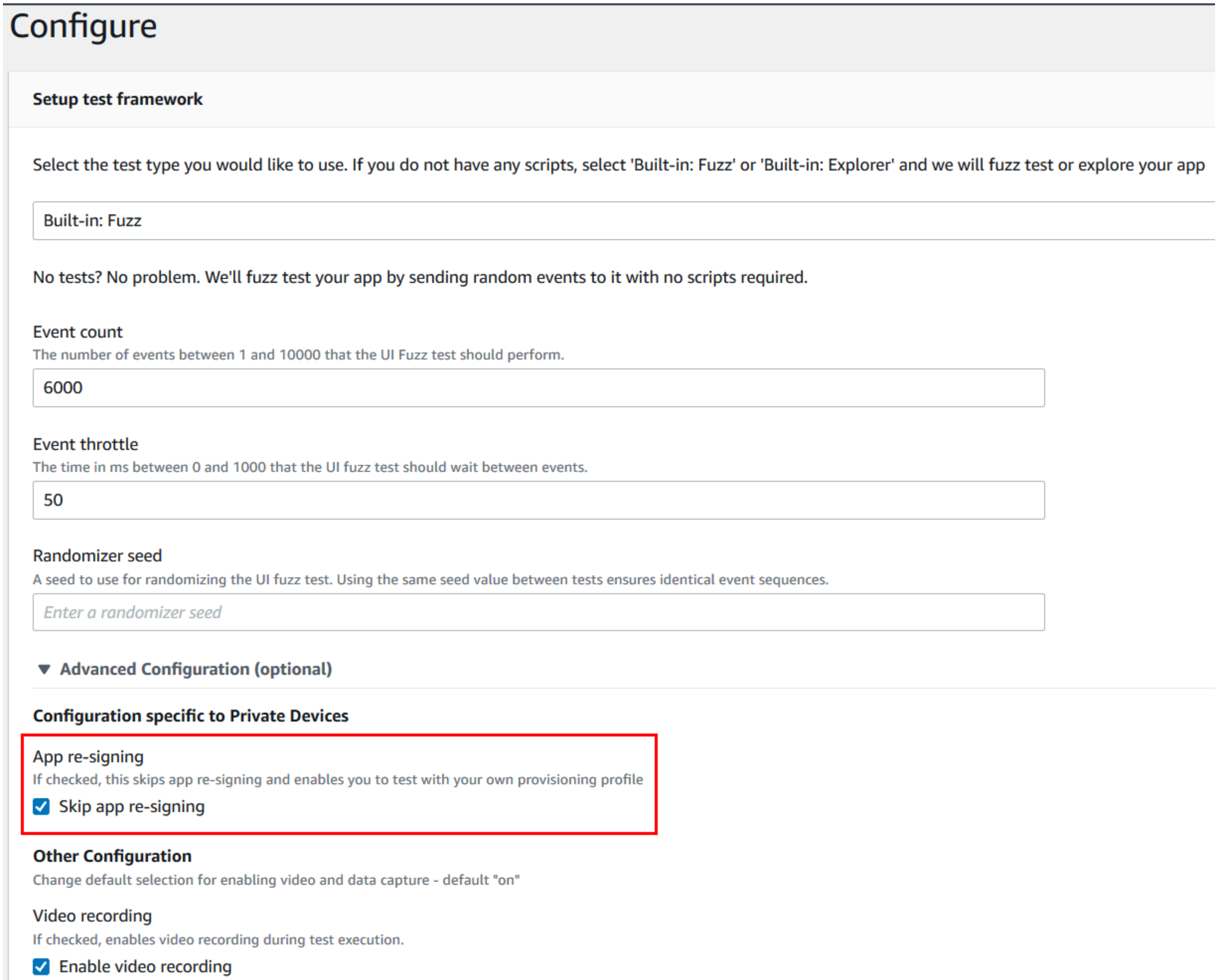
On iOS, we replace the embedded provisioning profile with a wildcard profile and resign the app. If you provide it, we will add auxiliary data to the application package before installation so the data will be present in your app’s sandbox. Resigning the iOS app results in the removal of certain entitlements. This includes App Group, Associated Domains, Game Center, HealthKit, HomeKit, Wireless Accessory Configuration, In-App Purchase, Inter-App Audio, Apple Pay, Push Notifications, and VPN Configuration & Control.

On Android, we resign the app. This may break functionality that depends on the app signature, such as the Google Maps Android API. It may also trigger anti-piracy and anti-tamper detection available from products such as DexGuard. For built-in tests, we may modify the manifest to include permissions required to capture and save screenshots.

When you use private devices, you can skip the step where AWS Device Farm re-signs your app. This is different from public devices, where Device Farm always re-signs your app on the Android and iOS platforms.

You can skip app re-signing when you create a remote access session or a test run. This can be helpful if your app has functionality that breaks when Device Farm re-signs your app. For example, push notifications might not work after re-signing. For more information about the changes that Device Farm makes when it tests your app, see the AWS Device Farm FAQs or the Apps page.

To skip app re-signing for a test run, select **Skip app re-signing** on the **Configure** page when you create the test run.



> **Note**
>
> If you're using the XCTest framework, the **Skip app re-signing** option is not available. For more information, see Integrating Device Farm with XCTest for iOS.

Additional steps for configuring your app-signing settings vary, depending on whether you're using private Android or iOS devices.

Skipping app re-signing on Android devices

If you're testing your app on a private Android device, select **Skip app re-signing** when you create your test run or your remote access session. No other configuration is required.

Skipping app re-signing on iOS devices

Apple requires you to sign an app for testing before you load it onto a device. For iOS devices, you have two options for signing your app.

- If you're using an in-house (Enterprise) developer profile, you can skip to the next section, the section called “Create a remote access session to trust your app”.
- If you're using an ad hoc iOS app development profile, you must first register the device with your Apple developer account, and then update your provisioning profile to include the private device. You must then re-sign your app with the provisioning profile that you updated. You can then run your re-signed app in Device Farm.

To register a device with an ad hoc iOS app development provisioning profile

1. Sign in to your Apple developer account.
2. Navigate to the **Certificates, IDs, and Profiles** section of the console.
3. Go to **Devices**.
4. Register the device in your Apple developer account. To get the name and UDID of the device, use the `ListDeviceInstances` operation of the Device Farm API.
5. Go to your provisioning profile and choose **Edit**.
6. Choose the device from the list.
7. In Xcode, fetch your updated provisioning profile, and then re-sign the app.

No other configuration is required. You can now create a remote access session or a test run and select **Skip app re-signing**.

Creating a remote access session to trust your iOS app

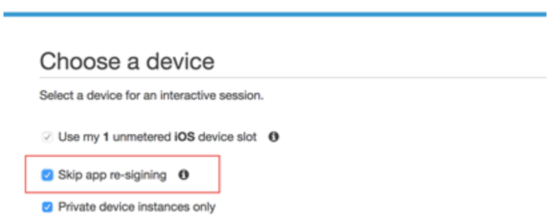
If you're using an in-house (Enterprise) developer provisioning profile, you must perform a one-time procedure to trust the in-house app developer certificate on each of your private devices.

To do so, you can either install the app that you want to test on the private device, or you can install a dummy app that's signed with the same certificate as the app that you want to test. There is an advantage to installing a dummy app that's signed with the same certificate. After you trust the configuration profile or enterprise app developer, all apps from that developer are trusted on the private device until you delete them. Therefore, when you upload a new version of the app that you want to test, you won't have to trust the app developer again. This is particularly useful if you run test automations and you don't want to create a remote access session each time you test your app.

Before you start your remote access session, follow the steps in [Creating an instance profile in AWS Device Farm](#) to create or modify an instance profile in Device Farm. In the instance profile, add the bundle ID of the test app or dummy app to the **Exclude packages from cleanup** setting. Then, attach the instance profile to the private device instance to ensure that Device Farm doesn't remove this app from the device before it starts a new test run. This ensures that your developer certificate remains trusted.

You can upload the dummy app to the device by using a remote access session, which allows you to launch the app and trust the developer.

1. Follow the instructions in [Creating a session](#) to create a remote access session that uses the private device instance profile that you created. When you create your session, be sure to select **Skip app re-signing**.

   > **Important**
   >
   > To filter the list of devices to include only private devices, select **Private device instances only** to ensure that you use a private device with the correct instance profile.

   Be sure to also add the dummy app or the app that you want to test to the **Exclude packages from cleanup** setting for the instance profile that's attached to this instance.

2. When your remote session starts, choose **Choose File** to install an application that uses your in-house provisioning profile.
3. Launch the app that you just uploaded.
4. Follow the instructions to trust the developer certificate.

All apps from this configuration profile or enterprise app developer are now trusted on this private device until you delete them.

Amazon VPC across AWS Regions in AWS Device Farm

Device Farm services are located only in the US West (Oregon) (`us-west-2`) Region. You can use Amazon Virtual Private Cloud (Amazon VPC) to reach a service in your Amazon Virtual Private Cloud in another AWS Region using Device Farm. If Device Farm and your service are in the same Region, see Using Amazon VPC endpoint services with Device Farm - Legacy (not recommended).

There are two ways to access your private services located in a different Region. If you have services located in one other Region that's not `us-west-2`, you can use VPC Peering in order to peer that Region's VPC to another VPC that is interfacing with Device Farm in `us-west-2`. However, if you have services in multiple Regions, a Transit Gateway will allow you to access those services with a simpler network configuration.

For more information, see VPC peering scenarios in the *Amazon VPC Peering Guide*.

VPC peering overview for VPCs in different Regions in AWS Device Farm

You can peer any two VPCs in different Regions as long as they have distinct, non-overlapping CIDR blocks. This ensures that all of the private IP addresses are unique, and it allows all of the resources in the VPCs to address each other without the need for any form of network address translation (NAT). For more information about CIDR notation, see RFC 4632.

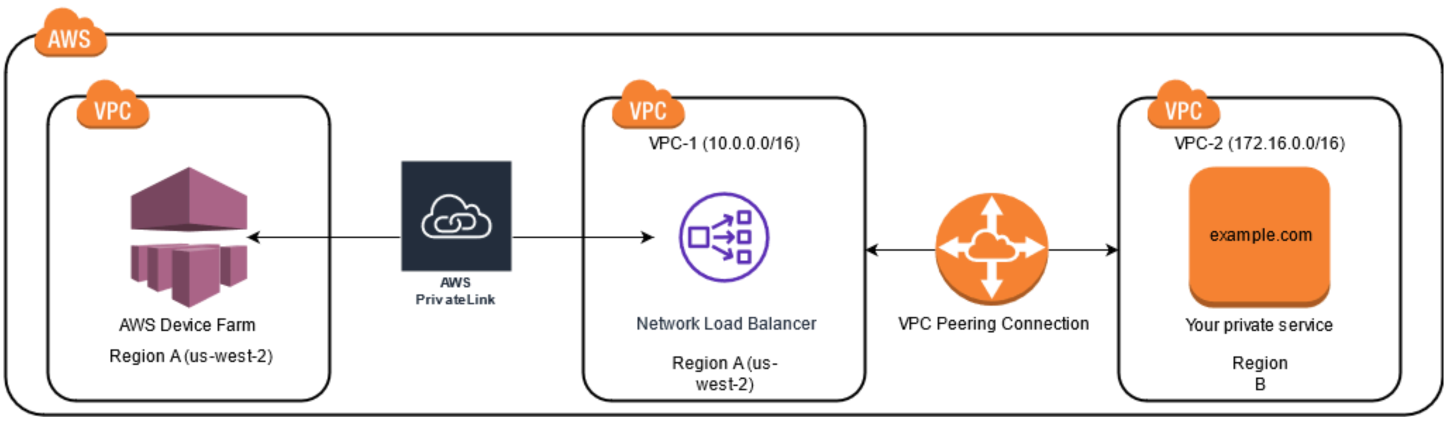
This topic includes a cross-Region example scenario in which Device Farm (referred to as *VPC-1*) is located in the US West (Oregon) (`us-west-2`) Region. The second VPC in this example (referred to as *VPC-2*) is in another Region.

Device Farm VPC cross-Region example

VPC Component	VPC-1	VPC-2
CIDR	10.0.0.0/16	172.16.0.0/16

> **Important**
>
> Establishing a peering connection between two VPCs can change the security posture of the VPCs. In addition, adding new entries to their route tables can change the security posture of the resources within the VPCs. It is your responsibility to implement these configurations in a way that meets your organization's security requirements. For more information, please see the Shared responsibility model.

The following diagram shows the components in the example and the interactions between these components.



Topics

Prerequisites for using Amazon VPC in AWS Device Farm

This example requires the following:

- Two VPCs that are configured with subnets containing non-overlapping CIDR blocks.
- *VPC-1* must be in the `us-west-2` Region and contain subnets for Availability Zones `us-west-2a`, `us-west-2b`, and `us-west-2c`.

For more information on creating VPCs and configuring subnets, see Working with VPCs and subnets in the *Amazon VPC Peering Guide*.

Step 1: Setting up a peering connection between VPC-1 and VPC-2

Establish a peering connection between the two VPCs containing non-overlapping CIDR blocks. To do this, see Create and accept VPC peering connections in the *Amazon VPC Peering Guide*. Using this topic's cross-Region scenario and the *Amazon VPC Peering Guide*, the following example peering connection configuration is created:

Name

```
Device-Farm-Peering-Connection-1
```

VPC ID (Requester)

```
vpc-0987654321gfedcba (VPC-2)
```

Account

```
My account
```

Region

```
US West (Oregon) (us-west-2)
```

VPC ID (Accepter)

```
vpc-1234567890abcdefg (VPC-1)
```

> **Note**
>
> Ensure that you consult your VPC peering connection quotas when establishing any new peering connections. For more information, please see Amazon VPC quotas in the *Amazon VPC Peering Guide*.

Step 2: Updating the route tables in VPC-1 and VPC-2

After setting up a peering connection, you must establish a destination route between the two VPCs to transfer data between them. To establish this route, you can manually update the route table of *VPC-1* to point to the subnet of *VPC-2* and vice versa. To do this, see [Update your route tables for a VPC peering connection]() in the *Amazon VPC Peering Guide*. Using this topic's cross-Region scenario and the *Amazon VPC Peering Guide*, the following example route table configuration is created:

Device Farm VPC route table example

VPC component	VPC-1	VPC-2
Route table ID	rtb-1234567890abcdefg	rtb-0987654321gfedcba
Local address range	10.0.0.0/16	172.16.0.0/16
Destination address range	172.16.0.0/16	10.0.0.0/16

Step 3: Creating a target group

After you set up your destination routes, you can configure a Network Load Balancer in *VPC-1* to route requests to *VPC-2*.

The Network Load Balancer must first contain a target group that contains the IP addresses to which requests are sent.

To create a target group

1. Identify the IP addresses of the service that you want to target in *VPC-2*.
   - These IP addresses must be members of the subnet used in the peering connection.
   - The targeted IP addresses must be static and immutable. If your service has dynamic IP addresses, consider targeting a static resource (such as a Network Load Balancer) and having that static resource route requests to your true target.

> **Note**
>
> - If you're targeting one or more stand-alone Amazon Elastic Compute Cloud (Amazon EC2) instances, open the Amazon EC2 console at https://console.aws.amazon.com/ec2/, then choose **Instances**.
> - If you're targeting an Amazon EC2 Auto Scaling group of Amazon EC2 instances, you must associate the Amazon EC2 Auto Scaling group to a Network Load Balancer. For more information, see Attaching a load balancer to your Auto Scaling group in the *Amazon EC2 Auto Scaling User Guide*.
>
>   Then, you can open the Amazon EC2 console at https://console.aws.amazon.com/ec2/, and then choose **Network Interfaces**. From there you can view the IP addresses for each of the Network Load Balancer's network interfaces in each **Availability Zone**.

2. Create a target group in *VPC-1*. To do this, see Create a target group for your Network Load Balancer in the *User Guide for Network Load Balancers*.

   Target groups for services in a different VPC require the following configuration:

   - For **Choose a target type**, choose **IP addresses**.
   - For **VPC**, choose the VPC that will host the load balancer. For the topic example, this will be *VPC-1*.
   - On the **Register targets** page, register a target for each IP address in *VPC-2*.

     For **Network**, choose **Other private IP address**.

     For **Availability Zone**, choose your desired zones in *VPC-1*.

     For **IPv4 address**, choose the *VPC-2* IP address.

     For **Ports**, choose your ports.
   - Choose **Include as pending below**. When you're finished specifying addresses, choose **Register pending targets**.

Using this topic's cross-Region scenario and the *User Guide for Network Load Balancers*, the following values are used in the target group configuration:

Target type

```
IP addresses
```

Target group name

```
my-target-group
```

Protocol/Port

```
TCP : 80
```

VPC

```
vpc-1234567890abcdefg (VPC-1)
```

Network

```
Other private IP address
```

Availability Zone

```
all
```

IPv4 address

```
172.16.100.60
```

Ports

```
80
```

Step 4: Creating a Network Load Balancer

Create a Network Load Balancer using the target group described in step 3. To do this, see Creating a Network Load Balancer.

Using this topic's cross-Region scenario, the following values are used in an example Network Load Balancer configuration:

Load balancer name

```
my-nlb
```

Scheme

```
Internal
```

VPC

`vpc-1234567890abcdefg (VPC-1)`

Mapping

`us-west-2a` - `subnet-4i23iuufkdiufsloi`

`us-west-2b` - `subnet-7x989pkjj78nmn23j`

`us-west-2c` - `subnet-0231ndmas12bnnsds`

Protocol/Port

`TCP : 80`

Target Group

`my-target-group`

Step 5: Creating a VPC endpoint service to connect your VPC to Device Farm

You can use the Network Load Balancer to create a VPC endpoint service. Through this VPC endpoint service, Device Farm can connect to your service in *VPC-2* without any additional infrastructure, such as an internet gateway, NAT instance, or VPN connection.

To do this, see Creating an Amazon VPC endpoint service.

Step 6: Create a VPC endpoint configuration between your VPC and Device Farm

Now you can establish a private connection between your VPC and Device Farm. You can use Device Farm to test private services without exposing them through the public internet. To do this, see Creating a VPC endpoint configuration in Device Farm.

Using this topic's cross-Region scenario, the following values are used in an example VPC endpoint configuration:

Name

`My VPCE Configuration`

VPCE service name

```
com.amazonaws.vpce.us-west-2.vpce-svc-1234567890abcdefg
```

Service DNS name

```
devicefarm.com
```

Step 7: Creating a test run to use the VPC endpoint configuration

You can create test runs that use the VPC endpoint configuration described in step 6. For more information, see Creating a test run in Device Farm or Creating a session.

Creating a scalable network with Transit Gateway

To create a scalable network using more than two VPCs, you can use Transit Gateway to act as a network transit hub to interconnect your VPCs and on-premises networks. To configure a VPC in the same region as Device Farm to use a Transit Gateway, you can follow the Amazon VPC endpoint services with Device Farm guide to target resources in another region based on their private IP addresses.

For more information about Transit Gateway, see What is a transit gateway? in the *Amazon VPC Transit Gateways Guide*.

Terminating private devices in Device Farm

To terminate a private device after your initial agreed term, you must provide a 30-day notice of non-renewal via our email at `<aws-devicefarm-support@amazon.com>`. For more information about private devices, see Private devices in AWS Device Farm.

> **Important**
>
> These instructions **only** apply to terminating private device agreements. For all other AWS services and billing issues, see the respective documentation for those products or contact AWS support.

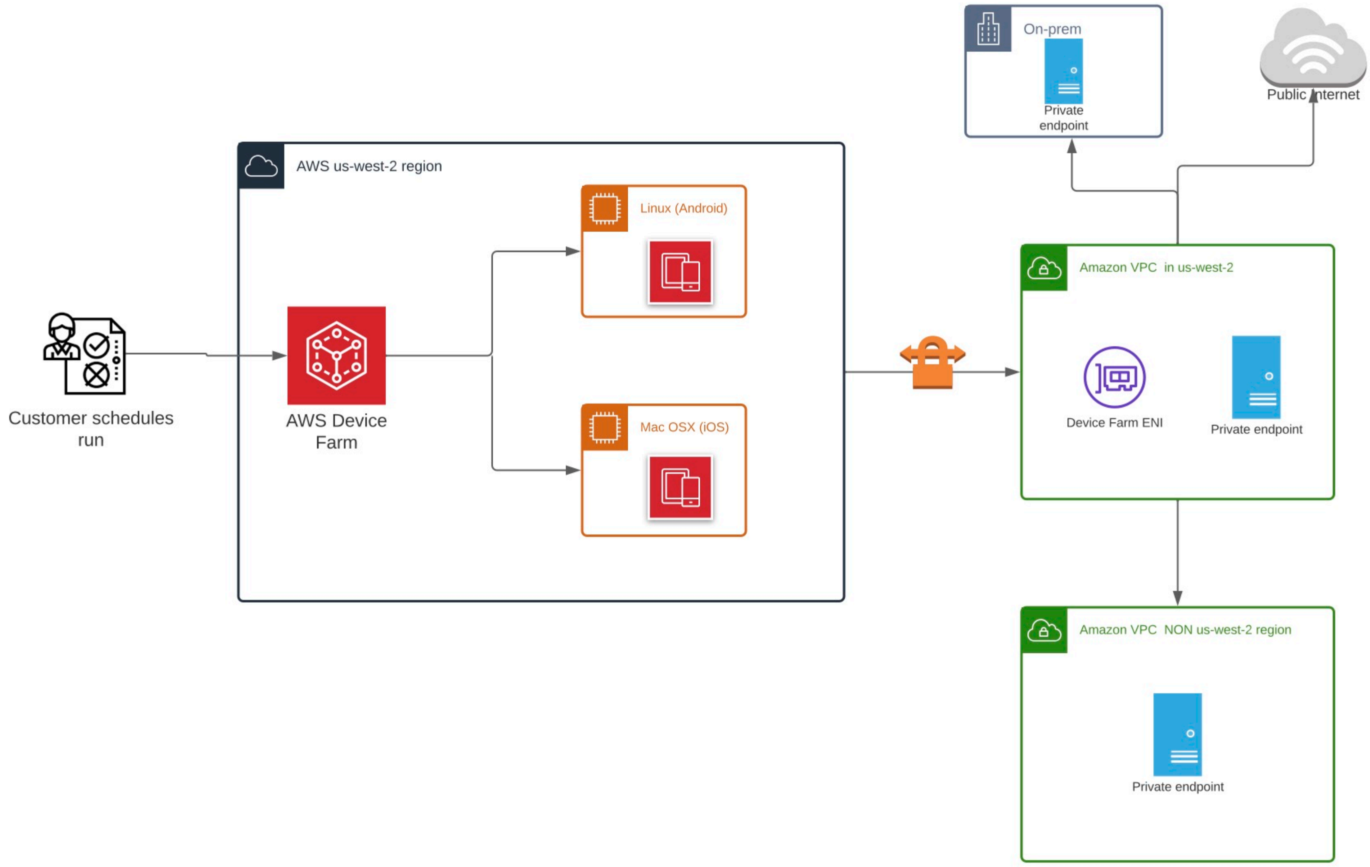
VPC-ENI in AWS Device Farm

> **Warning**
>
> This feature is only available on private devices. To request private device use on your AWS account, please contact us. If you already have private devices added to your AWS account, we strongly recommend using this method of VPC connectivity.

AWS Device Farm's VPC-ENI connectivity feature helps customers securely connect to their private endpoints hosted on AWS, on-premise software, or another cloud provider.

You can connect both Device Farm mobile devices and their host machines to an Amazon Virtual Private Cloud (Amazon VPC) environment in the `us-west-2` Region, which enables access to isolated, non-internet-facing services and applications through an elastic network interface. For more information on VPCs, see the Amazon VPC User Guide.

If your private endpoint or VPC is not in the `us-west-2` Region, you can link it with a VPC in the `us-west-2` Region using solutions such as a Transit Gateway or VPC Peering. In such situations, Device Farm will create an ENI in a subnet you provide for your `us-west-2` Region VPC, and you'll be responsible for ensuring that a connection can be established between the `us-west-2` Region VPC and the VPC in the other Region.



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

> **Note**
>
> Device Farm doesn't charge anything for creating ENIs in a customer's VPC in `us-west-2`. The cost for cross-Region or external inter-VPC connectivity isn't included in this feature.

Once you configure VPC access, the devices and host machines that you use for your tests won't be able to connect to resources outside of the VPC (e.g., 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*.

Topics

AWS access control and IAM

AWS Device Farm allows you to use AWS Identity and Access Management (IAM) to create policies granting or restricting access to Device Farm's features. To use the VPC Connectivity feature with AWS Device Farm, the following IAM Policy is required for the user account or role that you are using to access AWS Device Farm:

```
{
  "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/devicefarm.amazonaws.com/
AWSServiceRoleForDeviceFarm",
      "Condition": {
        "StringLike": {
          "iam:AWSServiceName": "devicefarm.amazonaws.com"
        }
```

```
      }
    }
  ]
}
```

To create or update a Device Farm project with a VPC configuration, your IAM policy must allow you to call the following actions against the resources listed in the VPC configuration:

```
"ec2:DescribeVpcs"
"ec2:DescribeSubnets"
"ec2:DescribeSecurityGroups"
"ec2:CreateNetworkInterface"
```

Additionally, your IAM policy must also allow for the creation of the service-linked role:

```
"iam:CreateServiceLinkedRole"
```

Note

None of these permissions are required for users who don't use VPC configurations in their projects.

Service-linked roles

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 **AWSServiceRoleForDeviceFarm** – Allows Device Farm to access AWS resources on your behalf.

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

- `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 mobile testing project, 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 `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 AWSServiceRoleForDeviceFarm 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 AWSServiceRoleForDeviceFarm 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 AWSServiceRoleForDeviceFarm 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
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

Region name	Region identity	Support in Device Farm
AWS GovCloud (US)	us-gov-west-1	No

Prerequisites

The following list describes some requirements and suggestions to review when creating VPC-ENI configurations:

- Private devices must be assigned to your AWS Account.
- You must have an AWS account user or role with permissions to create a Service-linked role. When using Amazon VPC endpoints with Device Farm mobile testing features, Device Farm creates an AWS Identity and Access Management (IAM) service-linked role.
- Device Farm can connect to VPCs only in the `us-west-2` Region. If you don't have a VPC in the `us-west-2` Region, you need to create one. Then, to access resources in a VPC in another Region, you must establish a peering connection between the VPC in the `us-west-2` Region and the VPC in the other Region. For information on peering VPCs, see the [Amazon VPC Peering Guide](#).

  You should 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.
- DNS resolution is required in the VPC that you use.
- Once your VPC has been created, you will need the following information about the VPC in the `us-west-2` Region:
  - VPC ID
  - Subnet IDs (private subnets only)
  - Security group IDs
- You must configure Amazon VPC connections on a per-project basis. At this time, you can configure only one VPC configuration per project. When you configure a VPC, Amazon VPC creates an interface within your VPC and assigns it to the specified subnets and security groups. All future sessions associated with the project will use the configured VPC connection.
- You cannot use VPC-ENI configurations along with the legacy VPCE feature.

- We strongly recommend **not updating an existing project** with a VPC-ENI configuration as existing projects may have VPCE settings that persist on the run level. Instead, if you already use the existing VPCE features, use VPC-ENI for all new projects.

Connecting to Amazon VPC

You can configure and update your project to use Amazon VPC endpoints. The VPC-ENI configuration is configured on a per-project basis. A project can have only one VPC-ENI endpoint at any given time. To configure VPC access for a project, you must know the following details:

- The VPC ID in `us-west-2` if your app is hosted there or the `us-west-2` VPC ID that connects to some other VPC in a different Region.
- The applicable security groups to apply to the connection.
- The subnets that will be associated with the connection. When a session starts, the largest available subnet is used. We recommend having multiple subnets associated with different availability zones to improve the availability posture of your VPC connectivity.

Once you have created your VPC-ENI configuration, you can update its details using the console or CLI using the steps below.

Console

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. Under **Mobile Testing projects**, choose the name of your project from the list.
4. Choose **Project settings**.
5. In the **Virtual Private Cloud (VPC) Settings** section, you can change the `VPC`, `Subnets` (private subnets only), and `Security Groups`.
6. Choose **Save**.

CLI

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

```
$  aws devicefarm update-project \
```

```
--arn arn:aws:devicefarm:us-
west-2:111122223333:project:12345678-1111-2222-333-456789abcdef \
--vpc-config \
securityGroupIds=sg-02c1537701a7e3763,sg-005dadf9311efda25,\
subnetIds=subnet-09b1a45f9cac53717,subnet-09b1a45f9cac12345,\
vpcId=vpc-0238fb322af81a368
```

You can also configure an Amazon VPC when creating your project:

```
$  aws devicefarm create-project \
--name VPCDemo \
--vpc-config \
securityGroupIds=sg-02c1537701a7e3763,sg-005dadf9311efda25,\
subnetIds=subnet-09b1a45f9cac53717,subnet-09b1a45f9cac12345,\
vpcId=vpc-0238fb322af81a368
```

Limits

The following limitations are applicable to the VPC-ENI feature:

- You can provide up to five security groups in the VPC configuration of a Device Farm project.
- You can provide up to eight subnets in the VPC configuration of a Device Farm project.
- When configuring a Device Farm project to work with your VPC, the smallest subnet you can provide must have a minimum of five available IPv4 addresses.
- Public IP addresses aren't supported at this time. Instead, we recommend that you use private subnets in your Device Farm projects. If your need public internet access during your tests, use a network address translation (NAT) gateway. Configuring a Device Farm project with a public subnet doesn't give your tests internet access or a public IP address.
- VPC-ENI integration only supports private subnets in your VPC.
- Only outgoing traffic from the service-managed ENI is supported. This means that the ENI cannot receive unsolicited inbound requests from the VPC.

Using Amazon VPC endpoint services with Device Farm - Legacy (not recommended)

Warning

We strongly recommend using the VPC-ENI connectivity described on this page for private endpoint connectivity as VPCE is now considered a legacy feature. VPC-ENI provides more flexibility, simpler configurations, is more cost efficient, and requires significantly less maintenance overhead when compared to the VPCE connectivity method.

Note

Using Amazon VPC Endpoint Services with Device Farm is only supported for customers with configured private devices. To enable your AWS account to use this feature with private devices, please contact us.

Amazon Virtual Private Cloud (Amazon VPC) is an AWS service that you can use to launch AWS resources in a virtual network that you define. With a VPC, you have control over your network settings, such as the IP address range, subnets, routing tables, and network gateways.

If you use Amazon VPC to host private applications in the US West (Oregon) (`us-west-2`) AWS Region, you can establish a private connection between your VPC and Device Farm. With this connection, you can use Device Farm to test private applications without exposing them through the public internet. To enable your AWS account to use this feature with private devices, contact us.

To connect a resource in your VPC to Device Farm, you can use the Amazon VPC console to create a VPC endpoint service. This endpoint service lets you provide the resource in your VPC to Device Farm through a Device Farm VPC endpoint. The endpoint service provides reliable, scalable connectivity to Device Farm without requiring an internet gateway, network address translation (NAT) instance, or VPN connection. For more information, see VPC endpoint services (AWS PrivateLink) in the *AWS PrivateLink Guide*.

Important

The Device Farm VPC endpoint feature helps you securely connect private internal services in your VPC to the Device Farm public VPC by using AWS PrivateLink connections. Although the connection is secure and private, that security depends on your protection of your AWS credentials. If your AWS credentials are compromised, an attacker can access or expose your service data to the outside world.

After you create a VPC endpoint service in Amazon VPC, you can use the Device Farm console to create a VPC endpoint configuration in Device Farm. This topic shows you how to create the Amazon VPC connection and the VPC endpoint configuration in Device Farm.

Before you begin

The following information is for Amazon VPC users in the US West (Oregon) (`us-west-2`) Region, with a subnet in each of the following Availability Zones: us-west-2a, us-west-2b, and us-west-2c.

Device Farm has additional requirements for the VPC endpoint services that you can use it with. When you create and configure a VPC endpoint service to work with Device Farm, make sure that you choose options that meet the following requirements:

- The Availability Zones for the service must include us-west-2a, us-west-2b, and us-west-2c. The Network Load Balancer that's associated with a VPC endpoint service determines the Availability Zones for that VPC endpoint service. If your VPC endpoint service doesn't show all three of these Availability Zones, you must re-create your Network Load Balancer to enable these three zones, and then reassociate the Network Load Balancer with your endpoint service.
- The allowed principals for the endpoint service must include the Amazon Resource Name (ARN) of the Device Farm VPC endpoint (service ARN). After you create your endpoint service, add the Device Farm VPC endpoint service ARN to your allow list to give Device Farm permission to access your VPC endpoint service. To get the Device Farm VPC endpoint service ARN, contact us.

In addition, if you keep the **Acceptance required** setting turned on when you create your VPC endpoint service, you must manually accept each connection request that Device Farm sends to the endpoint service. To change this setting for an existing endpoint service, choose the endpoint service on the Amazon VPC console, choose **Actions**, and then choose **Modify endpoint acceptance setting**. For more information, see Change the load balancers and acceptance settings in the *AWS PrivateLink Guide*.

The next section explains how to create an Amazon VPC endpoint service that meets these requirements.

Step 1: Creating a Network Load Balancer

The first step in establishing a private connection between your VPC and Device Farm is to create a Network Load Balancer to route requests to a target group.

New console

To create a Network Load Balancer using the new console

1. Open the Amazon Elastic Compute Cloud (Amazon EC2) console at https://console.aws.amazon.com/ec2/.
2. In the navigation pane, under **Load balancing**, choose **Load balancers**.
3. Choose **Create load balancer**.
4. Under **Network load balancer**, choose **Create**.
5. On the **Create network load balancer** page, under **Basic configuration**, do the following:
    a. Enter a load balancer **Name**.
    b. For **Scheme**, choose **Internal**.
6. Under **Network mapping**, do the following:
    a. Choose the **VPC** for your target group.
    b. Select the following **Mappings**:
        - `us-west-2a`
        - `us-west-2b`
        - `us-west-2c`
7. Under **Listeners and routing**, use the **Protocol** and **Port** options to choose your target group.

> **Note**
>
> By default, cross-availability zone load balancing is disabled.
> Because the load balancer uses the Availability Zones `us-west-2a`, `us-west-2b`, and `us-west-2c`, it either requires targets to be registered in each of those Availability Zones, or, if you register targets in less than all three zones, it requires

that you enable cross-zone load balancing. Otherwise, the load balancer might not work as expected.

8. Choose **Create load balancer**.

Old console

To create a Network Load Balancer using the old console

1. Open the Amazon Elastic Compute Cloud (Amazon EC2) console at https://console.aws.amazon.com/ec2/.
2. In the navigation pane, under **Load balancing**, choose **load balancers**.
3. Choose **Create load balancer**.
4. Under **Network load balancer**, choose **Create**.
5. On the **Configure load balancer** page, under **Basic configuration**, do the following:

   a. Enter a load balancer **Name**.

   b. For **Scheme**, choose **Internal**.
6. Under **Listeners**, select the **Protocol** and **Port** that your target group is using.
7. Under **Availability zones**, do the following:

   a. Choose the **VPC** for your target group.

   b. Select the following **Availability zones**:

      - `us-west-2a`
      - `us-west-2b`
      - `us-west-2c`

   c. Choose **Next: configure security settings**.
8. (Optional) Configure your security settings, then choose **Next: configure routing**.
9. On the **Configure Routing** page, do the following:

   a. For **Target group**, choose **Existing target group**.

   b. For **Name**, choose your target group.

   c. Choose **Next: register targets**.
10. On the **Register targets** page, review your targets, then choose **Next: review**.

> **Note**
>
> By default, cross-availability zone load balancing is disabled.
> Because the load balancer uses the Availability Zones `us-west-2a`, `us-west-2b`, and `us-west-2c`, it either requires targets to be registered in each of those Availability Zones, or, if you register targets in less than all three zones, it requires that you enable cross-zone load balancing. Otherwise, the load balancer might not work as expected.

11. Review your load balancer configuration, then choose **Create**.

Step 2: Creating an Amazon VPC endpoint service

After creating the Network Load Balancer, use the Amazon VPC console to create an endpoint service in your VPC.

1. Open the Amazon VPC console at https://console.aws.amazon.com/vpc/.
2. Under **Resources by region**, choose **Endpoint services**.
3. Choose **Create endpoint service**.
4. Do one of the following:
   - If you already have a Network Load Balancer that you want the endpoint service to use, choose it under **Available load balancers**, and then continue to step 5.
   - If you haven't yet created a Network Load Balancer, choose **Create new load balancer**. The Amazon EC2 console opens. Follow the steps in Creating a Network Load Balancer beginning with step 3, then continue with these steps in the Amazon VPC console.
5. For **Included availability zones**, verify that `us-west-2a`, `us-west-2b`, and `us-west-2c` appear in the list.
6. If you don't want to manually accept or deny each connection request that is sent to the endpoint service, under **Additional settings**, clear **Acceptance required**. If you clear this check box, the endpoint service automatically accepts each connection request that it receives.
7. Choose **Create**.
8. In the new endpoint service, choose **Allow principals**.

9. [Contact us](#) to get the ARN of the Device Farm VPC endpoint (service ARN) to add to the allow list for the endpoint service, and then add that service ARN to the allow list for the service.
10. On the **Details** tab for the endpoint service, make a note of the name of the service (**service name**). You need this name when you create the VPC endpoint configuration in the next step.

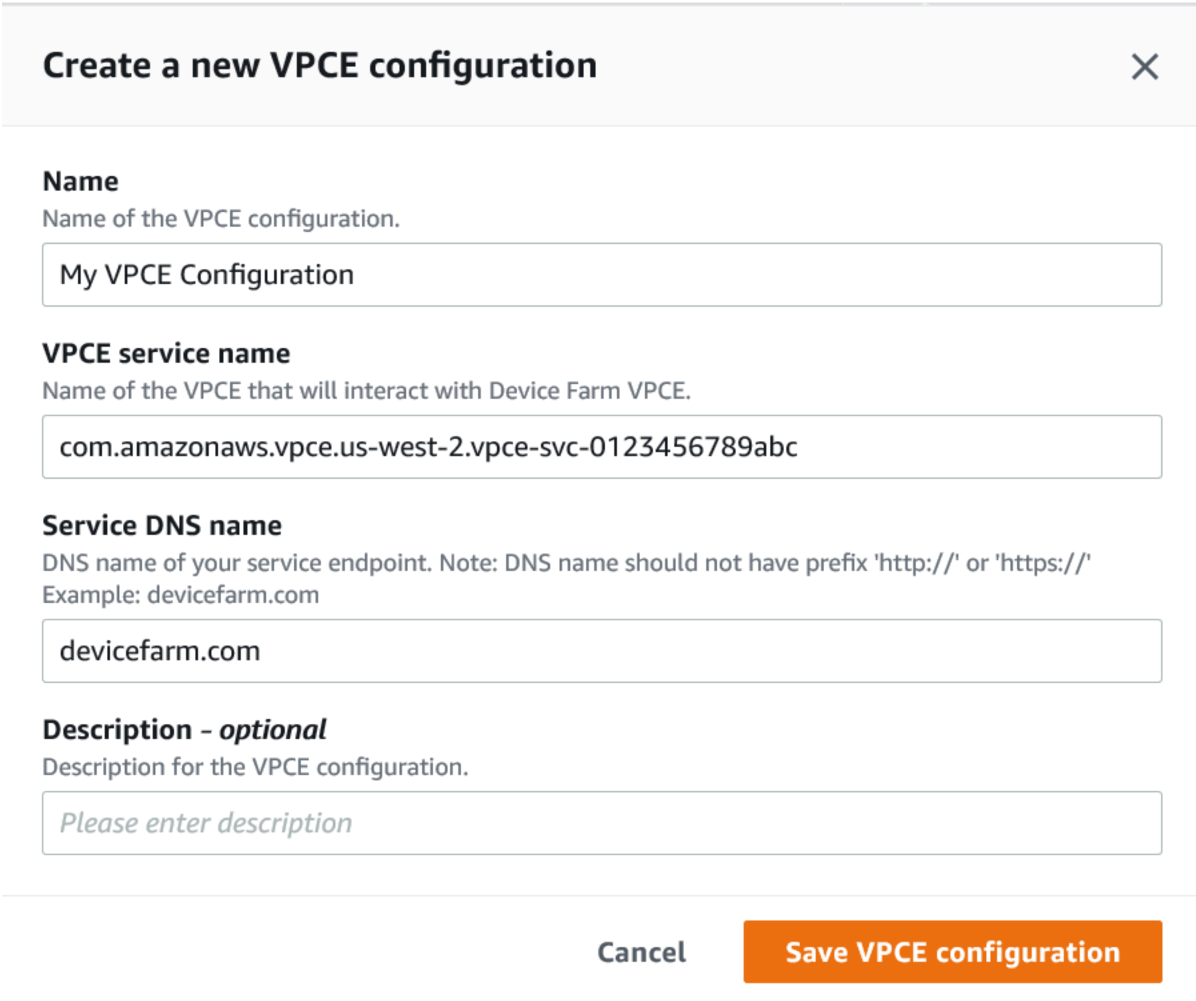
Your VPC endpoint service is now ready to use with Device Farm.

Step 3: Creating a VPC endpoint configuration in Device Farm

After you create an endpoint service in Amazon VPC, you can create an Amazon VPC endpoint configuration in Device Farm.

1. Sign in to the Device Farm console at https://console.aws.amazon.com/devicefarm.
2. In the navigation pane, choose **Mobile device testing**, then **Private devices**.
3. Choose **VPCE configurations**.
4. Choose **Create VPCE configuration**.
5. Under **Create a new VPCE configuration**, enter a **Name** for the VPC endpoint configuration.
6. For **VPCE service name**, enter the name of the Amazon VPC endpoint service (**service name**) that you noted in the Amazon VPC console. The name looks like `com.amazonaws.vpce.us-west-2.vpce-svc-id`.
7. For **Service DNS name**, enter the service DNS name for the app that you want to test (for example, `devicefarm.com`). Don't specify `http` or `https` before the service DNS name.

   The domain name is not accessible through the public internet. In addition, this new domain name, which maps to your VPC endpoint service, is generated by Amazon Route 53 and is available exclusively for you in your Device Farm session.
8. Choose **Save**.



Step 4: Creating a test run

After you save the VPC endpoint configuration, you can use the configuration to create test runs or remotely access sessions. For more information, see Creating a test run in Device Farm or Creating a session.

Logging AWS Device Farm API calls with AWS CloudTrail

AWS Device Farm is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in AWS Device Farm. CloudTrail captures all API calls for AWS Device Farm as events. The calls captured include calls from the AWS Device Farm console and code calls to the AWS Device Farm API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for AWS Device Farm. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to AWS Device Farm, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

AWS Device Farm information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in AWS Device Farm, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing Events with CloudTrail Event History.

For an ongoing record of events in your AWS account, including events for AWS Device Farm, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for Creating a Trail
- CloudTrail Supported Services and Integrations
- Configuring Amazon SNS Notifications for CloudTrail
- Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts

When CloudTrail logging is enabled in your AWS account, API calls made to Device Farm actions are tracked in log files. Device Farm records are written together with other AWS service records in a log file. CloudTrail determines when to create and write to a new file based on a time period and file size.

All of the Device Farm actions are logged and documented in the AWS CLI reference and the Automating Device Farm. For example, calls to create a new project or run in Device Farm generate entries in CloudTrail log files.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity Element.

Understanding AWS Device Farm log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the Device Farm `ListRuns` action:

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "Root",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam::123456789012:root",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
          "attributes": {
```

```
            "mfaAuthenticated": "false",
            "creationDate": "2015-07-08T21:13:35Z"
          }
        }
      },
      "eventTime":"2015-07-09T00:51:22Z",
      "eventSource": "devicefarm.amazonaws.com",
      "eventName":"ListRuns",
      "awsRegion":"us-west-2",
      "sourceIPAddress":"203.0.113.11",
      "userAgent":"example-user-agent-string",
      "requestParameters": {
        "arn":"arn:aws:devicefarm:us-west-2:123456789012:project:a9129b8c-
df6b-4cdd-8009-40a25EXAMPLE"},
        "responseElements": {
          "runs": [
            {
              "created": "Jul 8, 2015 11:26:12 PM",
              "name": "example.apk",
              "completedJobs": 2,
              "arn": "arn:aws:devicefarm:us-west-2:123456789012:run:a9129b8c-
df6b-4cdd-8009-40a256aEXAMPLE/1452d105-e354-4e53-99d8-6c993EXAMPLE",
              "counters": {
                "stopped": 0,
                "warned": 0,
                "failed": 0,
                "passed": 4,
                "skipped": 0,
                "total": 4,
                "errored": 0
              },
              "type": "BUILTIN_FUZZ",
              "status": "RUNNING",
              "totalJobs": 3,
              "platform": "ANDROID_APP",
              "result": "PENDING"
            },
            ... additional entries ...
          ]
        }
      }
    }
  ]
```

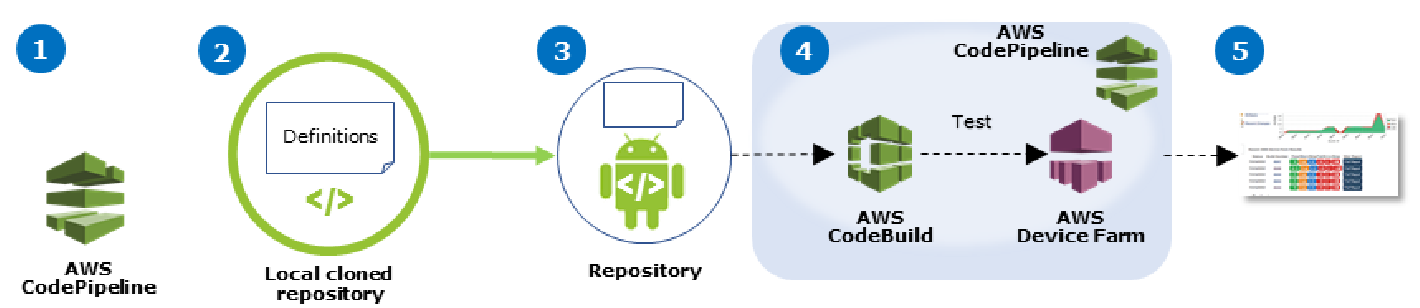
```
}
```

Integrating AWS Device Farm in a CodePipeline test stage

You can use AWS CodePipeline to incorporate mobile app tests configured in Device Farm into an AWS-managed automated release pipeline. You can configure your pipeline to run tests on demand, on a schedule, or as part of a continuous integration flow.

The following diagram shows the continuous integration flow in which an Android app is built and tested each time a push is committed to its repository. To create this pipeline configuration, see the Tutorial: Build and Test an Android App When Pushed to GitHub.

Workflow to Set Up Android Application Test



1. Configure	2. Add definitions	3. Push	4. Build and test	5. Report
Configure pipeline resources	*Add build and test definitions to your package*	*Push a package to your repository*	*App build and test of build output artifact kicked off automatically*	*View test results*

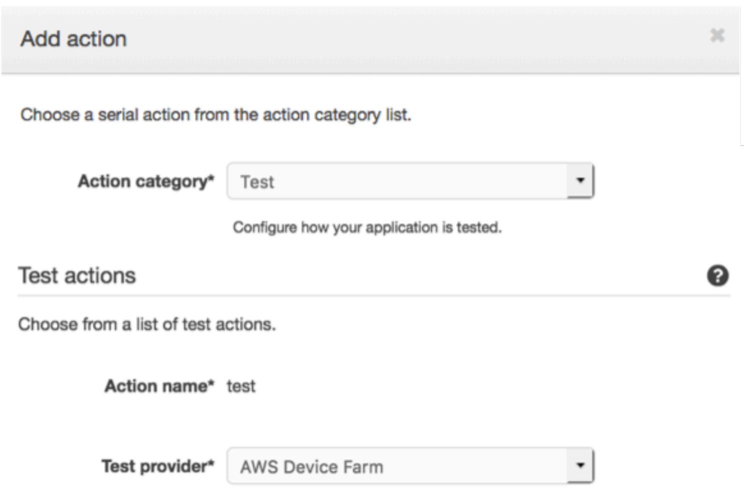
To learn how to configure a pipeline that continually tests a compiled app (such as an iOS `.ipa` or Android `.apk` file) as its source, see Tutorial: Test an iOS App Each Time You Upload an .ipa File to an Amazon S3 Bucket.

Configure CodePipeline to use your Device Farm tests

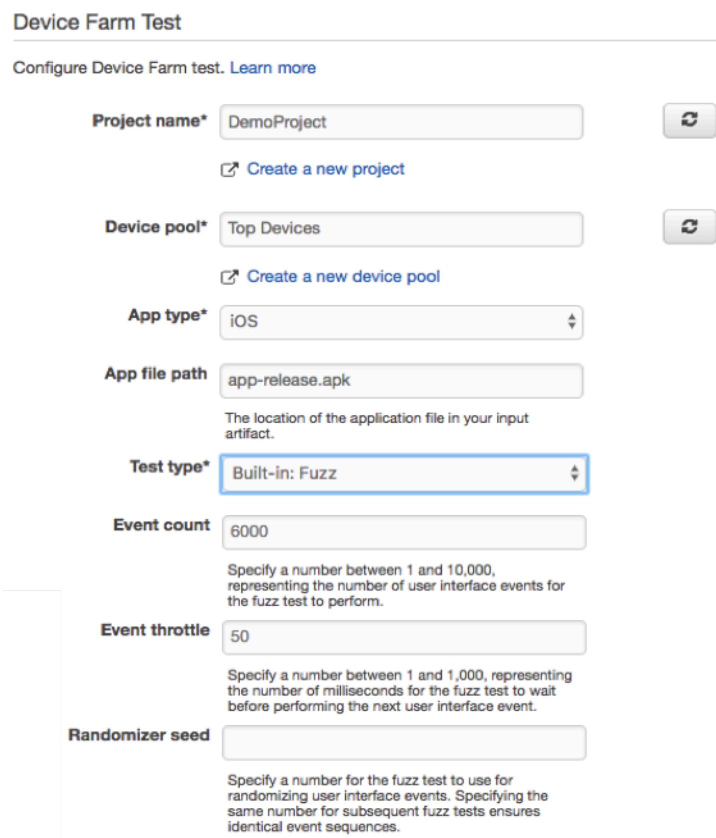
In these steps, we assume that you have configured a Device Farm project and created a pipeline. The pipeline should be configured with a test stage that receives an input artifact that contains your test definition and compiled app package files. The test stage input artifact can be the output artifact of either a source or build stage configured in your pipeline.

To configure a Device Farm test run as an CodePipeline test action

1. Sign in to the AWS Management Console and open the CodePipeline console at https://console.aws.amazon.com/codepipeline/.
2. Choose the pipeline for your app release.
3. On the test stage panel, choose the pencil icon, and then choose **Action**.
4. On the **Add action** panel, for **Action category**, choose **Test**.
5. In **Action name**, enter a name.
6. In **Test provider**, choose **AWS Device Farm**.



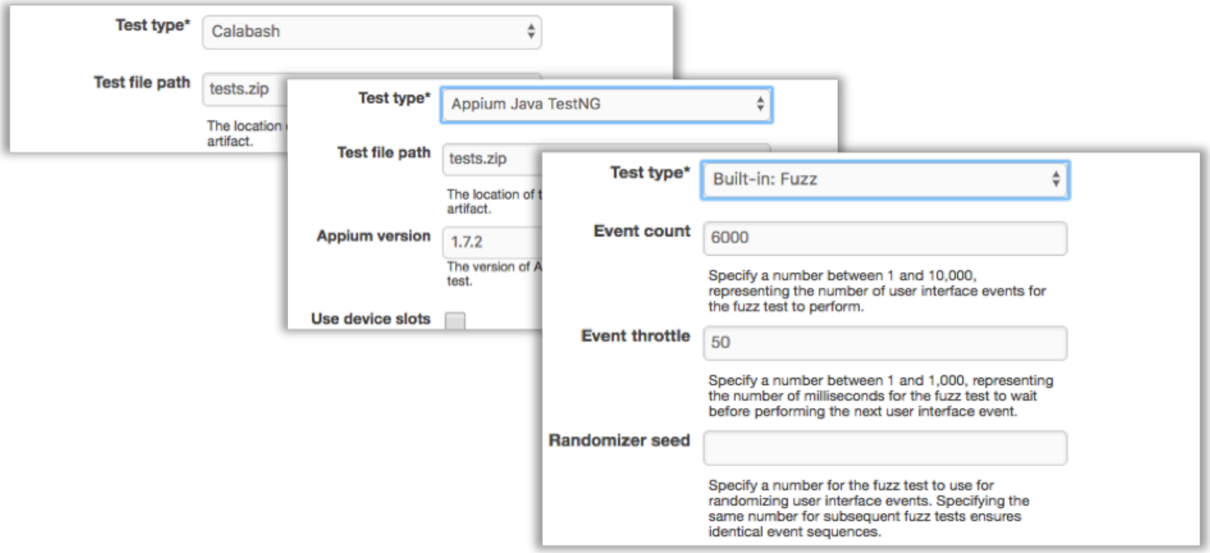
7. In **Project name**, choose your existing Device Farm project or choose **Create a new project**.
8. In **Device pool**, choose your existing device pool or choose **Create a new device pool**. If you create a device pool, you need to select a set of test devices.
9. In **App type**, choose the platform for your app.



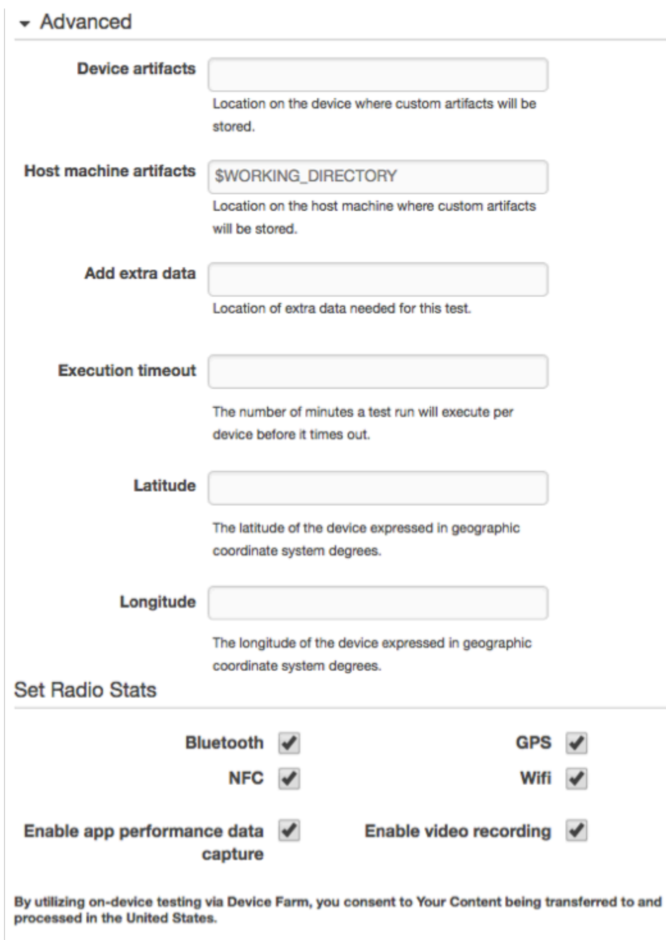
10. In **App file path**, enter the path of the compiled app package. The path is relative to the root of the input artifact for your test.

11. In **Test type**, do one of the following:

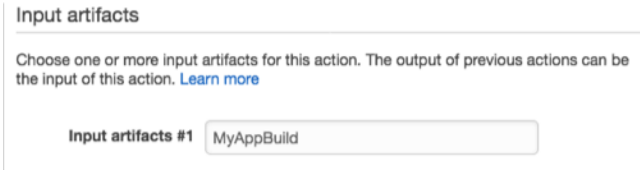
    - If you're using one of the built-in Device Farm tests, choose the type of test configured in your Device Farm project.
    - If you aren't using one of the Device Farm built-in tests, in the **Test file path**, enter the path of the test definition file. The path is relative to the root of the input artifact for your test.



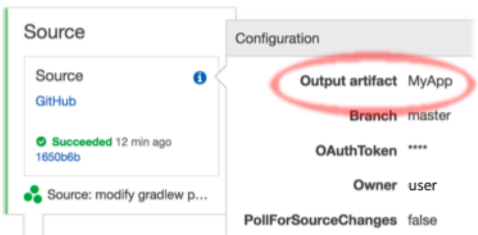
12. In the remaining fields, provide the configuration that is appropriate for your test and application type.
13. (Optional) In **Advanced**, provide detailed configuration for your test run.



14. In **Input artifacts**, choose the input artifact that matches the output artifact of the stage that comes before the test stage in the pipeline.



In the CodePipeline console, you can find the name of the output artifact for each stage by hovering over the information icon in the pipeline diagram. If your pipeline tests your app directly from the **Source** stage, choose **MyApp**. If your pipeline includes a **Build** stage, choose **MyAppBuild**.



15. At the bottom of the panel, choose **Add Action**.
16. In the CodePipeline pane, choose **Save pipeline change**, and then choose **Save change**.
17. To submit your changes and start a pipeline build, choose **Release change**, and then choose **Release**.

AWS CLI reference for AWS Device Farm

To use the AWS Command Line Interface (AWS CLI) to run Device Farm commands, see the AWS CLI Reference for AWS Device Farm.

For general information about the AWS CLI, see the AWS Command Line Interface User Guide and the AWS CLI Command Reference.

Windows PowerShell reference for AWS Device Farm

To use Windows PowerShell to run Device Farm commands, see the Device Farm Cmdlet Reference in the AWS Tools for Windows PowerShell Cmdlet Reference. For more information, see Setting up the AWS Tools for Windows PowerShell in the *AWS Tools for Windows PowerShell User Guide*.

Automating AWS Device Farm

Programmatic access to Device Farm is a powerful way to automate the common tasks that you need to accomplish, such as scheduling a run or downloading the artifacts for a run, suite, or test. The AWS SDK and AWS CLI provide means to do so.

The AWS SDK provides access to every AWS service, including Device Farm, Amazon S3, and more. For more information, see

- the AWS tools and SDKs
- the AWS Device Farm API Reference

Example: Using the AWS SDK to start a Device Farm run and collect artifacts

The following example provides a beginning-to-end demonstration of how you can use the AWS SDK to work with Device Farm. This example does the following:

- Uploads a test and application packages to Device Farm
- Starts a test run and waits for its completion (or failure)
- Downloads all artifacts produced by the test suites

This example depends on the third-party `requests` package to interact with HTTP.

```
import boto3
import os
import requests
import string
import random
import time
import datetime
import time
import json

# The following script runs a test through Device Farm
#
# Things you have to change:
config = {
```

```
    # This is our app under test.
    "appFilePath":"app-debug.apk",
    "projectArn": "arn:aws:devicefarm:us-
west-2:111122223333:project:1b99bcff-1111-2222-ab2f-8c3c733c55ed",
    # Since we care about the most popular devices, we'll use a curated pool.
    "testSpecArn":"arn:aws:devicefarm:us-west-2::upload:101e31e8-12ac-11e9-ab14-
d663bd873e83",
    "poolArn":"arn:aws:devicefarm:us-west-2::devicepool:082d10e5-d7d7-48a5-ba5c-
b33d66efa1f5",
    "namePrefix":"MyAppTest",
    # This is our test package. This tutorial won't go into how to make these.
    "testPackage":"tests.zip"
}

client = boto3.client('devicefarm')

unique =
 config['namePrefix']+"-"+(datetime.date.today().isoformat())+(''.join(random.sample(string.asc

print(f"The unique identifier for this run is going to be {unique} -- all uploads will
 be prefixed with this.")

def upload_df_file(filename, type_, mime='application/octet-stream'):
    response = client.create_upload(projectArn=config['projectArn'],
        name = (unique)+"_"+os.path.basename(filename),
        type=type_,
        contentType=mime
        )
    # Get the upload ARN, which we'll return later.
    upload_arn = response['upload']['arn']
    # We're going to extract the URL of the upload and use Requests to upload it
    upload_url = response['upload']['url']
    with open(filename, 'rb') as file_stream:
        print(f"Uploading {filename} to Device Farm as {response['upload']['name']}...
 ",end='')
        put_req = requests.put(upload_url, data=file_stream, headers={"content-
type":mime})
        print(' done')
        if not put_req.ok:
            raise Exception("Couldn't upload, requests said we're not ok. Requests
 says: "+put_req.reason)
    started = datetime.datetime.now()
    while True:
```

```
        print(f"Upload of {filename} in state {response['upload']['status']} after
 "+str(datetime.datetime.now() - started))
        if response['upload']['status'] == 'FAILED':
            raise Exception("The upload failed processing. DeviceFarm says reason
 is: \n"+(response['upload']['message'] if 'message' in response['upload'] else
 response['upload']['metadata']))
        if response['upload']['status'] == 'SUCCEEDED':
            break
        time.sleep(5)
        response = client.get_upload(arn=upload_arn)
    print("")
    return upload_arn

our_upload_arn = upload_df_file(config['appFilePath'], "ANDROID_APP")
our_test_package_arn = upload_df_file(config['testPackage'],
 'APPIUM_PYTHON_TEST_PACKAGE')
print(our_upload_arn, our_test_package_arn)
# Now that we have those out of the way, we can start the test run...
response = client.schedule_run(
    projectArn = config["projectArn"],
    appArn = our_upload_arn,
    devicePoolArn = config["poolArn"],
    name=unique,
    test = {
        "type":"APPIUM_PYTHON",
        "testSpecArn": config["testSpecArn"],
        "testPackageArn": our_test_package_arn
        }
    )
run_arn = response['run']['arn']
start_time = datetime.datetime.now()
print(f"Run {unique} is scheduled as arn {run_arn} ")

try:

    while True:
        response = client.get_run(arn=run_arn)
        state = response['run']['status']
        if state == 'COMPLETED' or state == 'ERRORED':
            break
        else:
            print(f" Run {unique} in state {state}, total time
 "+str(datetime.datetime.now()-start_time))
            time.sleep(10)
```

```
except:
    # If something goes wrong in this process, we stop the run and exit.

    client.stop_run(arn=run_arn)
    exit(1)
print(f"Tests finished in state {state} after "+str(datetime.datetime.now() -
 start_time))
# now, we pull all the logs.
jobs_response = client.list_jobs(arn=run_arn)
# Save the output somewhere. We're using the unique value, but you could use something
 else
save_path = os.path.join(os.getcwd(), unique)
os.mkdir(save_path)
# Save the last run information
for job in jobs_response['jobs'] :
    # Make a directory for our information
    job_name = job['name']
    os.makedirs(os.path.join(save_path, job_name), exist_ok=True)
    # Get each suite within the job
    suites = client.list_suites(arn=job['arn'])['suites']
    for suite in suites:
        for test in client.list_tests(arn=suite['arn'])['tests']:
            # Get the artifacts
            for artifact_type in ['FILE','SCREENSHOT','LOG']:
                artifacts = client.list_artifacts(
                    type=artifact_type,
                    arn = test['arn']
                )['artifacts']
                for artifact in artifacts:
                    # We replace : because it has a special meaning in Windows & macos
                    path_to = os.path.join(save_path, job_name, suite['name'],
 test['name'].replace(':','_') )
                    os.makedirs(path_to, exist_ok=True)
                    filename =
 artifact['type']+"_"+artifact['name']+"."+artifact['extension']
                    artifact_save_path = os.path.join(path_to, filename)
                    print("Downloading "+artifact_save_path)
                    with open(artifact_save_path, 'wb') as fn,
 requests.get(artifact['url'],allow_redirects=True) as request:
                        fn.write(request.content)
                    #/for artifact in artifacts
                #/for artifact type in []
            #/ for test in ()[]
        #/ for suite in suites
```

```
    #/ for job in _[]
# done
print("Finished")
```

Troubleshooting Device Farm errors

In this section, you will find error messages and procedures to help you fix common problems with Device Farm.

Topics

Troubleshooting Android application tests in AWS Device Farm

The following topic lists error messages that occur during the upload of Android application tests and recommends workarounds to resolve each error.

> **Note**
>
> The instructions below are based on Linux x86_64 and Mac.

ANDROID_APP_UNZIP_FAILED

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not open your application. Please verify that the file is valid and try again.

Make sure that you can unzip the application package without errors. In the following example, the package's name is **app-debug.apk**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip app-debug.apk
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   A valid Android application package should produce output like the following:

```
.
|-- AndroidManifest.xml
|-- classes.dex
|-- resources.arsc
|-- assets (directory)
|-- res (directory)
`-- META-INF (directory)
```

   For more information, see [Android tests in AWS Device Farm](#).

ANDROID_APP_AAPT_DEBUG_BADGING_FAILED

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not extract information about your application. Please verify that the application is valid by running the command `aapt debug badging` *<path to your test package>*, and try again after the command does not print any error.

During the upload validation process, AWS Device Farm parses out information from the output of an `aapt debug badging` *<path to your package>* command.

Make sure that you can run this command on your Android application successfully. In the following example, the package's name is **app-debug.apk**.

- Copy your application package to your working directory, and then run the command:

  ```
  $ aapt debug badging app-debug.apk
  ```

  A valid Android application package should produce output like the following:

  ```
  package: name='com.amazon.aws.adf.android.referenceapp' versionCode='1'
   versionName='1.0' platformBuildVersionName='5.1.1-1819727'
  sdkVersion:'9'
  application-label:'ReferenceApp'
  application: label='ReferenceApp' icon='res/mipmap-mdpi-v4/ic_launcher.png'
  application-debuggable
  launchable-activity:
   name='com.amazon.aws.adf.android.referenceapp.Activities.MainActivity'
   label='ReferenceApp' icon=''
  uses-feature: name='android.hardware.bluetooth'
  uses-implied-feature: name='android.hardware.bluetooth' reason='requested
   android.permission.BLUETOOTH permission, and targetSdkVersion > 4'
  main
  supports-screens: 'small' 'normal' 'large' 'xlarge'
  supports-any-density: 'true'
  locales: '--_--'
  densities: '160' '213' '240' '320' '480' '640'
  ```

  For more information, see Android tests in AWS Device Farm.

ANDROID_APP_PACKAGE_NAME_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the package name value in your application. Please verify that the application is valid by running the command `aapt debug badging` *<path to your*

test package>, and try again after finding the package name value behind the keyword "package: name."

During the upload validation process, AWS Device Farm parses out the package name value from the output of an `aapt debug badging` *<path to your package>* command.

Make sure that you can run this command on your Android application and find the package name value successfully. In the following example, the package's name is **app-debug.apk**.

- Copy your application package to your working directory, and then run the following command:

```
$ aapt debug badging app-debug.apk | grep "package: name="
```

  A valid Android application package should produce output like the following:

```
package: name='com.amazon.aws.adf.android.referenceapp' versionCode='1'
 versionName='1.0' platformBuildVersionName='5.1.1-1819727'
```

  For more information, see Android tests in AWS Device Farm.

ANDROID_APP_SDK_VERSION_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the SDK version value in your application. Please verify that the application is valid by running the command `aapt debug badging` *<path to your test package>*, and try again after finding the SDK version value behind the keyword `sdkVersion`.

During the upload validation process, AWS Device Farm parses out the SDK version value from the output of an `aapt debug badging` *<path to your package>* command.

Make sure that you can run this command on your Android application and find the package name value successfully. In the following example, the package's name is **app-debug.apk**.

- Copy your application package to your working directory, and then run the following command:

  ```
  $ aapt debug badging app-debug.apk | grep "sdkVersion"
  ```

  A valid Android application package should produce output like the following:

  ```
  sdkVersion:'9'
  ```

  For more information, see Android tests in AWS Device Farm.

ANDROID_APP_AAPT_DUMP_XMLTREE_FAILED

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the valid AndroidManifest.xml in your application. Please verify that the test package is valid by running the command `aapt dump xmltree <path to your test package> AndroidManifest.xml`, and try again after the command does not print any error.

During the upload validation process, AWS Device Farm parses out information from the XML parse tree for an XML file contained within the package using the command `aapt dump xmltree <path to your package> AndroidManifest.xml`.

Make sure that you can run this command on your Android application successfully. In the following example, the package's name is **app-debug.apk**.

- Copy your application package to your working directory, and then run the following command:

  ```
  $ aapt dump xmltree app-debug.apk. AndroidManifest.xml
  ```

A valid Android application package should produce output like the following:

```
N: android=http://schemas.android.com/apk/res/android
  E: manifest (line=2)
    A: android:versionCode(0x0101021b)=(type 0x10)0x1
    A: android:versionName(0x0101021c)="1.0" (Raw: "1.0")
    A: package="com.amazon.aws.adf.android.referenceapp" (Raw:
 "com.amazon.aws.adf.android.referenceapp")
    A: platformBuildVersionCode=(type 0x10)0x16 (Raw: "22")
    A: platformBuildVersionName="5.1.1-1819727" (Raw: "5.1.1-1819727")
    E: uses-sdk (line=7)
      A: android:minSdkVersion(0x0101020c)=(type 0x10)0x9
      A: android:targetSdkVersion(0x01010270)=(type 0x10)0x16
    E: uses-permission (line=11)
      A: android:name(0x01010003)="android.permission.INTERNET" (Raw:
 "android.permission.INTERNET")
    E: uses-permission (line=12)
      A: android:name(0x01010003)="android.permission.CAMERA" (Raw:
 "android.permission.CAMERA")
```

For more information, see [Android tests in AWS Device Farm](#).

ANDROID_APP_DEVICE_ADMIN_PERMISSIONS

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We found that your application requires device admin permissions. Please verify that the permissions are not required by run the command `aapt dump xmltree` *<path to your test package>* `AndroidManifest.xml`, and try again after making sure that output does not contain the keyword `android.permission.BIND_DEVICE_ADMIN`.

During the upload validation process, AWS Device Farm parses out permission information from the xml parse tree for an xml file contained within the package using the command `aapt dump xmltree` *<path to your package>* `AndroidManifest.xml`.

Make sure that your application does not require device admin permission. In the following example, the package's name is **app-debug.apk**.

- Copy your application package to your working directory, and then run the following command:

```
$ aapt dump xmltree app-debug.apk AndroidManifest.xml
```

You should find output like the following:

```
N: android=http://schemas.android.com/apk/res/android
  E: manifest (line=2)
    A: android:versionCode(0x0101021b)=(type 0x10)0x1
    A: android:versionName(0x0101021c)="1.0" (Raw: "1.0")
    A: package="com.amazonaws.devicefarm.android.referenceapp" (Raw:
 "com.amazonaws.devicefarm.android.referenceapp")
    A: platformBuildVersionCode=(type 0x10)0x16 (Raw: "22")
    A: platformBuildVersionName="5.1.1-1819727" (Raw: "5.1.1-1819727")
    E: uses-sdk (line=7)
      A: android:minSdkVersion(0x0101020c)=(type 0x10)0xa
      A: android:targetSdkVersion(0x01010270)=(type 0x10)0x16
    E: uses-permission (line=11)
      A: android:name(0x01010003)="android.permission.INTERNET" (Raw:
 "android.permission.INTERNET")
    E: uses-permission (line=12)
      A: android:name(0x01010003)="android.permission.CAMERA" (Raw:
 "android.permission.CAMERA")
        ……
```

If the Android application is valid, the output should not contain the following: `A: android:name(0x01010003)="android.permission.BIND_DEVICE_ADMIN" (Raw: "android.permission.BIND_DEVICE_ADMIN")`.

For more information, see Android tests in AWS Device Farm.

Certain windows in my Android application show a blank or black screen

If you are testing an Android application and notice that certain windows in the application appear with a black screen in Device Farm's video recording of your test, your application may be using Android's `FLAG_SECURE` feature. This flag (as described in Android's official documentation) is used to prevent certain windows of an application from being recorded by screen recording tools. As a result, Device Farm's screen recording feature (for both automation and remote access testing) may show a black screen in place of your application's window if the window uses this flag.

This flag is often used by developers for pages in their application that contain sensitive information such as login pages. If you are seeing a black screen in place of your application's screen for certain pages like its login page, work with your developer(s) to obtain a build of the application that doesn't use this flag for testing.

Additionally, note that Device Farm can still interact with application windows that have this flag. So, if your application's login page appears as a black screen, you may still be able to enter your credentials in order to log in to the application (and thus view pages not blocked by the `FLAG_SECURE` flag).

Troubleshooting Appium Java JUnit tests in AWS Device Farm

The following topic lists error messages that occur during the upload of Appium Java JUnit tests and recommends workarounds to resolve each error.

> **Note**
>
> The instructions below are based on Linux x86_64 and Mac.

APPIUM_JAVA_JUNIT_TEST_PACKAGE_UNZIP_FAILED

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not open your test ZIP file. Please verify that the file is valid and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

A valid Appium Java JUnit package should produce output like the following:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

For more information, see [Appium tests and AWS Device Farm](#).

APPIUM_JAVA_JUNIT_TEST_PACKAGE_DEPENDENCY_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the dependency-jars directory inside your test package. Please unzip your test package, verify that the dependency-jars directory is inside the package, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   If the Appium Java JUnit package is valid, you will find the *dependency-jars* directory inside the working directory:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

   For more information, see Appium tests and AWS Device Farm.

APPIUM_JAVA_JUNIT_TEST_PACKAGE_JAR_MISSING_IN_DEPENDENCY_DIR

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a JAR file in the dependency-jars directory tree. Please unzip your test package and then open the dependency-jars directory, verify that at least one JAR file is in the directory, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find at least one *jar* file inside the *dependency-jars* directory:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_JAVA_JUNIT_TEST_PACKAGE_TESTS_JAR_FILE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a *-tests.jar file in your test package. Please unzip your test package, verify that at least one *-tests.jar file is in the package, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find at least one `jar` file like `acme-android-appium-1.0-SNAPSHOT-tests.jar` in our example. The file's name may be different, but it should end with `-tests.jar`.

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_JAVA_JUNIT_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TESTS_JAR

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a class file within the tests JAR file. Please unzip your test package and then unjar the tests JAR file, verify that at least one class file is within the JAR file, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   You should find at least one jar file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *–tests.jar*.

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

3. After you successfully extract the files, you should find at least one class in the working directory tree by running the command:

```
$ tree .
```

   You should see output like this:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
everything built from the ./src/test directory)
|- one-class-file.class
```

```
|- folder
|    `-another-class-file.class
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_JAVA_JUNIT_TEST_PACKAGE_JUNIT_VERSION_VALUE_UNKNOWN

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a JUnit version value. Please unzip your test package and open the dependency-jars directory, verify that the JUnit JAR file is inside the directory, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working-directory tree structure by running the following command:

```
tree .
```

The output should look like this:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
```

```
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
    |— junit-4.10.jar
    |— com.some-dependency.bar-4.1.jar
    |— com.another-dependency.thing-1.0.jar
    |— joda-time-2.7.jar
    `— log4j-1.2.14.jar
```

If the Appium Java JUnit package is valid, you will find the JUnit dependency file that is similar to the jar file *junit-4.10.jar* in our example. The name should consist of the keyword *junit* and its version number, which in this example is 4.10.

For more information, see Appium tests and AWS Device Farm.

APPIUM_JAVA_JUNIT_TEST_PACKAGE_INVALID_JUNIT_VERSION

If you see the following message, follow these steps to fix the issue.

Warning

We found the JUnit version was lower than the minimum version 4.10 we support. Please change the JUnit version and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find a JUnit dependency file like *junit-4.10.jar* in our example and its version number, which in our example is 4.10:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— junit-4.10.jar
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

Note

Your tests may not execute correctly if the JUnit version specified in your test package is lower than the minimum version 4.10 we support.

For more information, see Appium tests and AWS Device Farm.

Troubleshooting Appium Java JUnit web application tests in AWS Device Farm

The following topic lists error messages that occur during the upload of Appium Java JUnit Web application tests and recommends workarounds to resolve each error. For more information on using Appium with Device Farm, see the section called “Appium”.

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_UNZIP_FAILED

If you see the following message, follow these steps to fix the issue.

Warning

We could not open your test ZIP file. Please verify that the file is valid and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   A valid Appium Java JUnit package should produce output like the following:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_DEPENDENCY_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the dependency-jars directory inside your test package. Please unzip your test package, verify that the dependency-jars directory is inside the package, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find the *dependency-jars* directory inside the working directory:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_JAR_MISSING_IN_DEPENDENCY_

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a JAR file in the dependency-jars directory tree. Please unzip your test package and then open the dependency-jars directory, verify that at least one JAR file is in the directory, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find at least one *jar* file inside the *dependency-jars* directory:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_TESTS_JAR_FILE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a *-tests.jar file in your test package. Please unzip your test package, verify that at least one *-tests.jar file is in the package, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find at least one *jar* file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *-tests.jar*.

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TESTS_JAR

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a class file within the tests JAR file. Please unzip your test package and then unjar the tests JAR file, verify that at least one class file is within the JAR file, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find at least one jar file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *-tests.jar*.

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

3. After you successfully extract the files, you should find at least one class in the working directory tree by running the command:

```
$ tree .
```

You should see output like this:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
everything built from the ./src/test directory)
|- one-class-file.class
|- folder
|    `-another-class-file.class
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
```

```
        |— com.another-dependency.thing-1.0.jar
        |— joda-time-2.7.jar
        `— log4j-1.2.14.jar
```

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_JUNIT_VERSION_VALUE_UNKNO

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a JUnit version value. Please unzip your test package and open the dependency-jars directory, verify that the JUnit JAR file is inside the directory, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working-directory tree structure by running the following command:

```
tree .
```

The output should look like this:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
        |— junit-4.10.jar
        |— com.some-dependency.bar-4.1.jar
        |— com.another-dependency.thing-1.0.jar
        |— joda-time-2.7.jar
```

```
        `— log4j-1.2.14.jar
```

If the Appium Java JUnit package is valid, you will find the JUnit dependency file that is similar to the jar file *junit-4.10.jar* in our example. The name should consist of the keyword *junit* and its version number, which in this example is 4.10.

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_INVALID_JUNIT_VERSION

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We found the JUnit version was lower than the minimum version 4.10 we support. Please change the JUnit version and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find a JUnit dependency file like *junit-4.10.jar* in our example and its version number, which in our example is 4.10:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— junit-4.10.jar
```

```
    |— com.some-dependency.bar-4.1.jar
    |— com.another-dependency.thing-1.0.jar
    |— joda-time-2.7.jar
    `— log4j-1.2.14.jar
```

> **Note**
>
> Your tests may not execute correctly if the JUnit version specified in your test package is lower than the minimum version 4.10 we support.

For more information, see Appium tests and AWS Device Farm.

Troubleshooting Appium Java TestNG tests in AWS Device Farm

The following topic lists error messages that occur during the upload of Appium Java TestNG tests and recommends workarounds to resolve each error.

> **Note**
>
> The instructions below are based on Linux x86_64 and Mac.

APPIUM_JAVA_TESTNG_TEST_PACKAGE_UNZIP_FAILED

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not open your test ZIP file. Please verify that the file is valid and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

A valid Appium Java JUnit package should produce output like the following:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_JAVA_TESTNG_TEST_PACKAGE_DEPENDENCY_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

Warning

We could not find the `dependency-jars` directory inside your test package. Please unzip your test package, verify that the `dependency-jars` directory is inside the package, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find the *dependency-jars* directory inside the working directory.

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_JAVA_TESTNG_TEST_PACKAGE_JAR_MISSING_IN_DEPENDENCY_DIR

If you see the following message, follow these steps to fix the issue.

Warning

We could not find a JAR file in the dependency-jars directory tree. Please unzip your test package and then open the dependency-jars directory, verify that at least one JAR file is in the directory, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find at least one *jar* file inside the *dependency-jars* directory.

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_JAVA_TESTNG_TEST_PACKAGE_TESTS_JAR_FILE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a *-tests.jar file in your test package. Please unzip your test package, verify that at least one *-tests.jar file is in the package, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find at least one *jar* file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *-tests.jar*.

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_JAVA_TESTNG_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TESTS_JA

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a class file within the tests JAR file. Please unzip your test package and then unjar the tests JAR file, verify that at least one class file is within the JAR file, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find at least one jar file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *–tests.jar*.

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

3. To extract files from the jar file, you can run the following command:

```
$ jar xf acme-android-appium-1.0-SNAPSHOT-tests.jar
```

4. After you successfully extract the files, run the following command:

```
$ tree .
```

You should find at least one class in the working directory tree:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
everything built from the ./src/test directory)
|- one-class-file.class
```

```
|- folder
|     `– another-class-file.class
|– zip-with-dependencies.zip (this .zip file contains all of the items)
`– dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |– com.some-dependency.bar-4.1.jar
      |– com.another-dependency.thing-1.0.jar
      |– joda-time-2.7.jar
      `– log4j-1.2.14.jar
```

For more information, see Appium tests and AWS Device Farm.

Troubleshooting Appium Java TestNG web applications in AWS Device Farm

The following topic lists error messages that occur during the upload of Appium Java TestNG Web application tests and recommends workarounds to resolve each error.

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_UNZIP_FAILED

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not open your test ZIP file. Please verify that the file is valid and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

A valid Appium Java JUnit package should produce output like the following:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_DEPENDENCY_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the dependency-jars directory inside your test package. Please unzip your test package, verify that the dependency-jars directory is inside the package, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find the *dependency-jars* directory inside the working directory.

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_JAR_MISSING_IN_DEPENDENC

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a JAR file in the dependency-jars directory tree. Please unzip your test package and then open the dependency-jars directory, verify that at least one JAR file is in the directory, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find at least one *jar* file inside the *dependency-jars* directory.

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_TESTS_JAR_FILE_MISSING

If you see the following message, follow these steps to fix the issue.

Warning

We could not find a *-tests.jar file in your test package. Please unzip your test package, verify that at least one *-tests.jar file is in the package, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find at least one *jar* file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *-tests.jar*.

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TEST

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a class file within the tests JAR file. Please unzip your test package and then unjar the tests JAR file, verify that at least one class file is within the JAR file, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip zip-with-dependencies.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find at least one jar file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *-tests.jar*.

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
 everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

3. To extract files from the jar file, you can run the following command:

```
$ jar xf acme-android-appium-1.0-SNAPSHOT-tests.jar
```

4. After you successfully extract the files, run the following command:

```
$ tree .
```

You should find at least one class in the working directory tree:

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything
 built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
everything built from the ./src/test directory)
|- one-class-file.class
|- folder
|     `— another-class-file.class
|— zip-with-dependencies.zip (this .zip file contains all of the items)
`— dependency-jars  (this is the directory that contains all of your dependencies,
 built as JAR files)
```

```
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
```

For more information, see Appium tests and AWS Device Farm.

Troubleshooting Appium Python tests in AWS Device Farm

The following topic lists error messages that occur during the upload of Appium Python tests and recommends workarounds to resolve each error.

APPIUM_PYTHON_TEST_PACKAGE_UNZIP_FAILED

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not open your Appium test ZIP file. Please verify that the file is valid and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

A valid Appium Python package should produce output like the following:

```
.
|-- requirements.txt
|-- test_bundle.zip
```

```
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_PYTHON_TEST_PACKAGE_DEPENDENCY_WHEEL_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a dependency wheel file in the wheelhouse directory tree. Please unzip your test package and then open the wheelhouse directory, verify that at least one wheel file is in the directory, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Python package is valid, you will find at least one `.whl` dependent file like the highlighted files inside the `wheelhouse` directory.

```
.
|-- requirements.txt
```

```
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_PYTHON_TEST_PACKAGE_INVALID_PLATFORM

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We found at least one wheel file specified a platform that we do not support. Please unzip your test package and then open the wheelhouse directory, verify that names of wheel files end with -any.whl or -linux_x86_64.whl, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Python package is valid, you will find at least one `.whl` dependent file like the highlighted files inside the `wheelhouse` directory. The file's name may be different, but it should end with `-any.whl` or `-linux_x86_64.whl`, which specifies the platform. Any other platforms like `windows` are not supported.

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_PYTHON_TEST_PACKAGE_TEST_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the tests directory inside your test package. Please unzip your test package, verify that the tests directory is inside the package, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   If the Appium Python package is valid, you will find the *tests* directory inside the working directory.

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_PYTHON_TEST_PACKAGE_INVALID_TEST_FILE_NAME

If you see the following message, follow these steps to fix the issue.

Warning

We could not find a valid test file in the tests directory tree. Please unzip your test package and then open the tests directory, verify that at least one file's name starts or ends with the keyword "test", and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Python package is valid, you will find the *tests* directory inside the working directory. The file's name may be different, but it should start with *test_* or end with *_test.py*.

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_PYTHON_TEST_PACKAGE_REQUIREMENTS_TXT_FILE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the requirements.txt file inside your test package. Please unzip your test package, verify that the requirements.txt file is inside the package, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Python package is valid, you will find the *requirements.txt* file inside the working directory.

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
       |-- Appium_Python_Client-0.20-cp27-none-any.whl
       |-- py-1.4.31-py2.py3-none-any.whl
       |-- pytest-2.9.0-py2.py3-none-any.whl
       |-- selenium-2.52.0-cp27-none-any.whl
       `-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_PYTHON_TEST_PACKAGE_INVALID_PYTEST_VERSION

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We found the pytest version was lower than the minimum version 2.8.0 we support. Please change the pytest version inside the requirements.txt file, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find the *requirements.txt* file inside the working directory.

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `--test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

3. To get the pytest version, you can run the following command:

```
$ grep "pytest" requirements.txt
```

You should find output like the following:

```
pytest==2.9.0
```

It shows the pytest version, which in this example is 2.9.0. If the Appium Python package is valid, the pytest version should be larger than or equal to 2.8.0.

For more information, see Appium tests and AWS Device Farm.

APPIUM_PYTHON_TEST_PACKAGE_INSTALL_DEPENDENCY_WHEELS_FAILED

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We failed to install the dependency wheels. Please unzip your test package and then open the requirements.txt file and the wheelhouse directory, verify that the dependency wheels

specified in the requirements.txt file exactly match the dependency wheels inside the wheelhouse directory, and try again.

We strongly recommend that you set up Python virtualenv for packaging tests. Here is an example flow of creating a virtual environment using Python virtualenv and then activating it:

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip test_bundle.zip
   ```

2. To test installing wheel files, you can run the following command:

   ```
   $ pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./
   requirements.txt
   ```

   A valid Appium Python package should produce output like the following:

   ```
   Ignoring indexes: https://pypi.python.org/simple
   Collecting Appium-Python-Client==0.20 (from -r ./requirements.txt (line 1))
   Collecting py==1.4.31 (from -r ./requirements.txt (line 2))
   Collecting pytest==2.9.0 (from -r ./requirements.txt (line 3))
   Collecting selenium==2.52.0 (from -r ./requirements.txt (line 4))
   Collecting wheel==0.26.0 (from -r ./requirements.txt (line 5))
   Installing collected packages: selenium, Appium-Python-Client, py, pytest, wheel
     Found existing installation: wheel 0.29.0
       Uninstalling wheel-0.29.0:
         Successfully uninstalled wheel-0.29.0
   Successfully installed Appium-Python-Client-0.20 py-1.4.31 pytest-2.9.0
    selenium-2.52.0 wheel-0.26.0
   ```

3. To deactivate the virtual environment, you can run the following command:

```
$ deactivate
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_PYTHON_TEST_PACKAGE_PYTEST_COLLECT_FAILED

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We failed to collect tests in the tests directory. Please unzip your test package, very that the test package is valid by running the command `py.test --collect-only <path to your tests directory>`, and try again after the command does not print any error.

We strongly recommend that you set up Python virtualenv for packaging tests. Here is an example flow of creating a virtual environment using Python virtualenv and then activating it:

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip test_bundle.zip
   ```

2. To install wheel files, you can run the following command:

   ```
   $ pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./
   requirements.txt
   ```

3. To collect tests, you can run the following command:

   ```
   $ py.test --collect-only tests
   ```

A valid Appium Python package should produce output like the following:

```
==================== test session starts =====================
platform darwin -- Python 2.7.11, pytest-2.9.0, py-1.4.31, pluggy-0.3.1
rootdir: /Users/zhena/Desktop/Ios/tests, inifile:
collected 1 items
<Module 'test_unittest.py'>
  <UnitTestCase 'DeviceFarmAppiumWebTests'>
    <TestCaseFunction 'test_devicefarm'>

==================== no tests ran in 0.11 seconds ====================
```

4. To deactivate the virtual environment, you can run the following command:

```
$ deactivate
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_PYTHON_TEST_PACKAGE_DEPENDENCY_WHEELS_INSUFFICIENT

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find enough wheel dependencies in the wheelhouse directory. Please unzip your test package, and then open the wheelhouse directory. Verify that you have all the wheel dependencies specified in the requirements.txt file.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. Check the length of the `requirements.txt` file as well as the number of `.whl` dependent files in the wheelhouse directory:

```
$ cat requirements.txt | egrep "." |wc -l
    12
$ ls wheelhouse/ | egrep ".+\.whl" | wc -l
    11
```

If the number of *.whl* dependent files is less than the number of non-empty rows in your *requirements.txt* file, then you need to ensure the following:

- There is a *.whl* dependent file corresponding to each row in the *requirements.txt* file.
- There are no other lines in the *requirements.txt* file that contain information other than the dependency package names.
- No dependency names are duplicated in multiple lines in the *requirements.txt* file such that two lines in the file may correspond to one *.whl* dependent file.

AWS Device Farm doesn't support lines in the *requirements.txt* file that don't directly correspond to dependency packages, such as lines that specify global options for the `pip install` command. See Requirements file format for a list of global options.

For more information, see Appium tests and AWS Device Farm.

Troubleshooting Appium Python web application tests in AWS Device Farm

The following topic lists error messages that occur during the upload of Appium Python Web application tests and recommends workarounds to resolve each error.

APPIUM_WEB_PYTHON_TEST_PACKAGE_UNZIP_FAILED

If you see the following message, follow these steps to fix the issue.

Warning

We could not open your Appium test ZIP file. Please verify that the file is valid and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

    ```
    $ unzip test_bundle.zip
    ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

    ```
    $ tree .
    ```

    A valid Appium Python package should produce output like the following:

    ```
    .
    |-- requirements.txt
    |-- test_bundle.zip
    |-- tests (directory)
    |       `-- test_unittest.py
    `-- wheelhouse (directory)
            |-- Appium_Python_Client-0.20-cp27-none-any.whl
            |-- py-1.4.31-py2.py3-none-any.whl
            |-- pytest-2.9.0-py2.py3-none-any.whl
            |-- selenium-2.52.0-cp27-none-any.whl
            `-- wheel-0.26.0-py2.py3-none-any.whl
    ```

    For more information, see Appium tests and AWS Device Farm.

APPIUM_WEB_PYTHON_TEST_PACKAGE_DEPENDENCY_WHEEL_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a dependency wheel file in the wheelhouse directory tree. Please unzip your test package and then open the wheelhouse directory, verify that at least one wheel file is in the directory, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Appium Python package is valid, you will find at least one *.whl* dependent file like the highlighted files inside the *wheelhouse* directory.

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|      `-- test_unittest.py
`-- wheelhouse (directory)
       |-- Appium_Python_Client-0.20-cp27-none-any.whl
       |-- py-1.4.31-py2.py3-none-any.whl
       |-- pytest-2.9.0-py2.py3-none-any.whl
       |-- selenium-2.52.0-cp27-none-any.whl
       `-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium tests and AWS Device Farm.

APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_PLATFORM

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We found at least one wheel file specified a platform that we do not support. Please unzip your test package and then open the wheelhouse directory, verify that names of wheel files end with -any.whl or -linux_x86_64.whl, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   If the Appium Python package is valid, you will find at least one *.whl* dependent file like the highlighted files inside the *wheelhouse* directory. The file's name may be different, but it should end with *-any.whl* or *-linux_x86_64.whl*, which specifies the platform. Any other platforms like `windows` are not supported.

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

   For more information, see Appium tests and AWS Device Farm.

APPIUM_WEB_PYTHON_TEST_PACKAGE_TEST_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the tests directory inside your test package. Please unzip your test package, verify that the tests directory is inside the package, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   If the Appium Python package is valid, you will find the *`tests`* directory inside the working directory.

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
       |-- Appium_Python_Client-0.20-cp27-none-any.whl
       |-- py-1.4.31-py2.py3-none-any.whl
       |-- pytest-2.9.0-py2.py3-none-any.whl
       |-- selenium-2.52.0-cp27-none-any.whl
       `-- wheel-0.26.0-py2.py3-none-any.whl
```

   For more information, see Appium tests and AWS Device Farm.

APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_TEST_FILE_NAME

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find a valid test file in the tests directory tree. Please unzip your test package and then open the tests directory, verify that at least one file's name starts or ends with the keyword "test", and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   If the Appium Python package is valid, you will find the `tests` directory inside the working directory. The file's name may be different, but it should start with `test_` or end with `_test.py`.

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

   For more information, see Appium tests and AWS Device Farm.

APPIUM_WEB_PYTHON_TEST_PACKAGE_REQUIREMENTS_TXT_FILE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the requirements.txt file inside your test package. Please unzip your test package, verify that the requirements.txt file is inside the package, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   If the Appium Python package is valid, you will find the *requirements.txt* file inside the working directory.

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|       `-- test_unittest.py
`-- wheelhouse (directory)
        |-- Appium_Python_Client-0.20-cp27-none-any.whl
        |-- py-1.4.31-py2.py3-none-any.whl
        |-- pytest-2.9.0-py2.py3-none-any.whl
        |-- selenium-2.52.0-cp27-none-any.whl
        `-- wheel-0.26.0-py2.py3-none-any.whl
```

   For more information, see Appium tests and AWS Device Farm.

APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_PYTEST_VERSION

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We found the pytest version was lower than the minimum version 2.8.0 we support. Please change the pytest version inside the requirements.txt file, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip test_bundle.zip
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```
   $ tree .
   ```

   You should find the *requirement.txt* file inside the working directory.

   ```
   .
   |-- requirements.txt
   |-- test_bundle.zip
   |-- tests (directory)
   |       `--test_unittest.py
   `-- wheelhouse (directory)
          |-- Appium_Python_Client-0.20-cp27-none-any.whl
          |-- py-1.4.31-py2.py3-none-any.whl
          |-- pytest-2.9.0-py2.py3-none-any.whl
          |-- selenium-2.52.0-cp27-none-any.whl
          `-- wheel-0.26.0-py2.py3-none-any.whl
   ```

3. To get the pytest version, you can run the following command:

   ```
   $ grep "pytest" requirements.txt
   ```

   You should find output like the following:

   ```
   pytest==2.9.0
   ```

   It shows the pytest version, which in this example is 2.9.0. If the Appium Python package is valid, the pytest version should be larger than or equal to 2.8.0.

   For more information, see Appium tests and AWS Device Farm.

APPIUM_WEB_PYTHON_TEST_PACKAGE_INSTALL_DEPENDENCY_WHEELS_FA

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We failed to install the dependency wheels. Please unzip your test package and then open the requirements.txt file and the wheelhouse directory, verify that the dependency wheels specified in the requirements.txt file exactly match the dependency wheels inside the wheelhouse directory, and try again.

We strongly recommend that you set up Python virtualenv for packaging tests. Here is an example flow of creating a virtual environment using Python virtualenv and then activating it:

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip test_bundle.zip
   ```

2. To test installing wheel files, you can run the following command:

   ```
   $ pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./
   requirements.txt
   ```

   A valid Appium Python package should produce output like the following:

   ```
   Ignoring indexes: https://pypi.python.org/simple
   Collecting Appium-Python-Client==0.20 (from -r ./requirements.txt (line 1))
   Collecting py==1.4.31 (from -r ./requirements.txt (line 2))
   Collecting pytest==2.9.0 (from -r ./requirements.txt (line 3))
   Collecting selenium==2.52.0 (from -r ./requirements.txt (line 4))
   Collecting wheel==0.26.0 (from -r ./requirements.txt (line 5))
   ```

```
Installing collected packages: selenium, Appium-Python-Client, py, pytest, wheel
  Found existing installation: wheel 0.29.0
    Uninstalling wheel-0.29.0:
      Successfully uninstalled wheel-0.29.0
Successfully installed Appium-Python-Client-0.20 py-1.4.31 pytest-2.9.0
 selenium-2.52.0 wheel-0.26.0
```

3. To deactivate the virtual environment, you can run the following command:

```
$ deactivate
```

   For more information, see Appium tests and AWS Device Farm.

APPIUM_WEB_PYTHON_TEST_PACKAGE_PYTEST_COLLECT_FAILED

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We failed to collect tests in the tests directory. Please unzip your test package, very that the test package is valid by running the command "py.test --collect-only <path to your tests directory>", and try again after the command does not print any error.

We strongly recommend that you set up Python virtualenv for packaging tests. Here is an example flow of creating a virtual environment using Python virtualenv and then activating it:

```
$ virtualenv workspace
$ cd workspace
$ source bin/activate
```

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip test_bundle.zip
```

2. To install wheel files, you can run the following command:

```
$ pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./
requirements.txt
```

3. To collect tests, you can run the following command:

```
$ py.test --collect-only tests
```

   A valid Appium Python package should produce output like the following:

```
==================== test session starts ====================
platform darwin -- Python 2.7.11, pytest-2.9.0, py-1.4.31, pluggy-0.3.1
rootdir: /Users/zhena/Desktop/Ios/tests, inifile:
collected 1 items
<Module 'test_unittest.py'>
  <UnitTestCase 'DeviceFarmAppiumWebTests'>
    <TestCaseFunction 'test_devicefarm'>

==================== no tests ran in 0.11 seconds ====================
```

4. To deactivate the virtual environment, you can run the following command:

```
$ deactivate
```

   For more information, see Appium tests and AWS Device Farm.

Troubleshooting instrumentation tests in AWS Device Farm

The following topic lists error messages that occur during the upload of Instrumentation tests and recommends workarounds to resolve each error.

> **Note**
>
> For important considerations when using Instrumentation tests in AWS Device Farm, see Instrumentation for Android and AWS Device Farm.

INSTRUMENTATION_TEST_PACKAGE_UNZIP_FAILED

If you see the following message, follow these steps to fix the issue.

```
Warning: We could not open your test APK file. Please verify that the file is valid and
    try again.
```

Make sure that you can unzip the test package without errors. In the following example, the package's name is **app-debug-androidTest-unaligned.apk**.

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip app-debug-androidTest-unaligned.apk
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```
   $ tree .
   ```

   A valid Instrumentation test package will produce output like the following:

   ```
   .
   |-- AndroidManifest.xml
   |--  classes.dex
   |-- resources.arsc
   |-- LICENSE-junit.txt
   |-- junit (directory)
   `-- META-INF (directory)
   ```

   For more information, see Instrumentation for Android and AWS Device Farm.

INSTRUMENTATION_TEST_PACKAGE_AAPT_DEBUG_BADGING_FAILED

If you see the following message, follow these steps to fix the issue.

```
We could not extract information about your test package. Please verify that the
      test package is valid by running the command "aapt debug badging <path to your
 test
      package>", and try again after the command does not print any error.
```

During the upload validation process, Device Farm parses out information from the output of the `aapt debug badging <path to your package>` command.

Make sure that you can run this command on your Instrumentation test package successfully.

In the following example, the package's name is **app-debug-androidTest-unaligned.apk**.

- Copy your test package to your working directory, and then run the following command:

  ```
  $ aapt debug badging app-debug-androidTest-unaligned.apk
  ```

  A valid Instrumentation test package will produce output like the following:

  ```
  package: name='com.amazon.aws.adf.android.referenceapp.test' versionCode=''
   versionName='' platformBuildVersionName='5.1.1-1819727'
  sdkVersion:'9'
  targetSdkVersion:'22'
  application-label:'Test-api'
  application: label='Test-api' icon=''
  application-debuggable
  uses-library:'android.test.runner'
  feature-group: label=''
  uses-feature: name='android.hardware.touchscreen'
  uses-implied-feature: name='android.hardware.touchscreen' reason='default feature
   for all apps'
  supports-screens: 'small' 'normal' 'large' 'xlarge'
  supports-any-density: 'true'
  locales: '--_--'
  densities: '160'
  ```

  For more information, see Instrumentation for Android and AWS Device Farm.

INSTRUMENTATION_TEST_PACKAGE_INSTRUMENTATION_RUNNER_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

```
We could not find the instrumentation runner value in the AndroidManifest.xml.
      Please verify the test package is valid by running the command "aapt dump xmltree
 <path to
      your test package> AndroidManifest.xml", and try again after finding the
 instrumentation
```

```
runner value behind the keyword "instrumentation."
```

During the upload validation process, Device Farm parses out the instrumentation runner value from the XML parse tree for an XML file contained within the package. You can use the following command: `aapt dump xmltree <path to your package> AndroidManifest.xml`.

Make sure that you can run this command on your Instrumentation test package and find the instrumentation value successfully.

In the following example, the package's name is **app-debug-androidTest-unaligned.apk**.

- Copy your test package to your working directory, and then run the following command:

  ```
  $ aapt dump xmltree app-debug-androidTest-unaligned.apk AndroidManifest.xml | grep
   -A5 "instrumentation"
  ```

  A valid Instrumentation test package will produce output like the following:

  ```
  E: instrumentation (line=9)
        A: android:label(0x01010001)="Tests for
   com.amazon.aws.adf.android.referenceapp" (Raw: "Tests for
   com.amazon.aws.adf.android.referenceapp")
        A:
   android:name(0x01010003)="android.support.test.runner.AndroidJUnitRunner" (Raw:
   "android.support.test.runner.AndroidJUnitRunner")
        A:
   android:targetPackage(0x01010021)="com.amazon.aws.adf.android.referenceapp" (Raw:
   "com.amazon.aws.adf.android.referenceapp")
        A: android:handleProfiling(0x01010022)=(type 0x12)0x0
        A: android:functionalTest(0x01010023)=(type 0x12)0x0
  ```

  For more information, see [Instrumentation for Android and AWS Device Farm](#).

INSTRUMENTATION_TEST_PACKAGE_AAPT_DUMP_XMLTREE_FAILED

If you see the following message, follow these steps to fix the issue.

```
We could not find the valid AndroidManifest.xml in your test package. Please
      verify that the test package is valid by running the command "aapt dump xmltree
 <path to
```

```
      your test package> AndroidManifest.xml", and try again after the command does not
print any
      error.
```

During the upload validation process, Device Farm parses out information from the XML parse tree for an XML file contained within the package using the following command: `aapt dump xmltree <path to your package> AndroidManifest.xml`.

Make sure that you can run this command on your instrumentation test package successfully.

In the following example, the package's name is **app-debug-androidTest-unaligned.apk**.

- Copy your test package to your working directory, and then run the following command:

  ```
  $ aapt dump xmltree app-debug-androidTest-unaligned.apk AndroidManifest.xml
  ```

  A valid Instrumentation test package will produce output like the following:

  ```
  N: android=http://schemas.android.com/apk/res/android
    E: manifest (line=2)
      A: package="com.amazon.aws.adf.android.referenceapp.test" (Raw:
   "com.amazon.aws.adf.android.referenceapp.test")
      A: platformBuildVersionCode=(type 0x10)0x16 (Raw: "22")
      A: platformBuildVersionName="5.1.1-1819727" (Raw: "5.1.1-1819727")
      E: uses-sdk (line=5)
        A: android:minSdkVersion(0x0101020c)=(type 0x10)0x9
        A: android:targetSdkVersion(0x01010270)=(type 0x10)0x16
      E: instrumentation (line=9)
        A: android:label(0x01010001)="Tests for
   com.amazon.aws.adf.android.referenceapp" (Raw: "Tests for
   com.amazon.aws.adf.android.referenceapp")
        A:
   android:name(0x01010003)="android.support.test.runner.AndroidJUnitRunner" (Raw:
   "android.support.test.runner.AndroidJUnitRunner")
        A:
   android:targetPackage(0x01010021)="com.amazon.aws.adf.android.referenceapp" (Raw:
   "com.amazon.aws.adf.android.referenceapp")
        A: android:handleProfiling(0x01010022)=(type 0x12)0x0
        A: android:functionalTest(0x01010023)=(type 0x12)0x0
      E: application (line=16)
        A: android:label(0x01010001)=@0x7f020000
        A: android:debuggable(0x0101000f)=(type 0x12)0xffffffff
  ```

```
      E: uses-library (line=17)
        A: android:name(0x01010003)="android.test.runner" (Raw:
 "android.test.runner")
```

For more information, see Instrumentation for Android and AWS Device Farm.

INSTRUMENTATION_TEST_PACKAGE_TEST_PACKAGE_NAME_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

```
We could not find the package name in your test package. Please verify that the
      test package is valid by running the command "aapt debug badging <path to your
 test
      package>", and try again after finding the package name value behind the keyword
 "package:
      name."
```

During the upload validation process, Device Farm parses out the package name value from the output of the following command: `aapt debug badging <path to your package>`.

Make sure that you can run this command on your Instrumentation test package and find the package name value successfully.

In the following example, the package's name is **app-debug-androidTest-unaligned.apk**.

- Copy your test package to your working directory, and then run the following command:

```
$ aapt debug badging app-debug-androidTest-unaligned.apk | grep "package: name="
```

A valid Instrumentation test package will produce output like the following:

```
package: name='com.amazon.aws.adf.android.referenceapp.test' versionCode=''
 versionName='' platformBuildVersionName='5.1.1-1819727'
```

For more information, see Instrumentation for Android and AWS Device Farm.

Troubleshooting iOS application tests in AWS Device Farm

The following topic lists error messages that occur during the upload of iOS application tests and recommends workarounds to resolve each error.

> **Note**
>
> The instructions below are based on Linux x86_64 and Mac.

IOS_APP_UNZIP_FAILED

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not open your application. Please verify that the file is valid and try again.

Make sure that you can unzip the application package without errors. In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

1. Copy your application package to your working directory, and then run the following command:

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```
   $ tree .
   ```

   A valid iOS application package should produce output like the following:

   ```
   .
   `-- Payload (directory)
           `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                          |-- Info.plist
                          `-- (any other files)
   ```

For more information, see iOS tests in AWS Device Farm.

IOS_APP_PAYLOAD_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the Payload directory inside your application. Please unzip your application, verify that the Payload directory is inside the package, and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

1. Copy your application package to your working directory, and then run the following command:

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   If the iOS application package is valid, you will find the *Payload* directory inside the working directory.

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                      |-- Info.plist
                      `-- (any other files)
```

   For more information, see iOS tests in AWS Device Farm.

IOS_APP_APP_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the .app directory inside the Payload directory. Please unzip your application and then open the Payload directory, verify that the .app directory is inside the directory, and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

1. Copy your application package to your working directory, and then run the following command:

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   If the iOS application package is valid, you will find an *.app* directory like *AWSDeviceFarmiOSReferenceApp.app* in our example inside the *Payload* directory.

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                      |-- Info.plist
                      `-- (any other files)
```

   For more information, see iOS tests in AWS Device Farm.

IOS_APP_PLIST_FILE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the Info.plist file inside the .app directory. Please unzip your application and then open the .app directory, verify that the Info.plist file is inside the directory, and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

1. Copy your application package to your working directory, and then run the following command:

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   If the iOS application package is valid, you will find the *`Info.plist`* file inside the *`.app`* directory like *`AWSDeviceFarmiOSReferenceApp.app`* in our example.

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                      |-- Info.plist
                      `-- (any other files)
```

   For more information, see iOS tests in AWS Device Farm.

IOS_APP_CPU_ARCHITECTURE_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the CPU architecture value in the Info.plist file. Please unzip your application and then open Info.plist file inside the .app directory, verify that the key "UIRequiredDeviceCapabilities" is specified, and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

1. Copy your application package to your working directory, and then run the following command:

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   You should find the *Info.plist* file inside an *.app* directory like *AWSDeviceFarmiOSReferenceApp.app* in our example:

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                      |-- Info.plist
                      `-- (any other files)
```

3. To find the CPU architecture value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

4. Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
Info.plist')
print info_plist['UIRequiredDeviceCapabilities']
```

A valid iOS application package should produce output like the following:

```
['armv7']
```

For more information, see iOS tests in AWS Device Farm.

IOS_APP_PLATFORM_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the platform value in the Info.plist file. Please unzip your application and then open Info.plist file inside the .app directory, verify that the key "CFBundleSupportedPlatforms" is specified, and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

1. Copy your application package to your working directory, and then run the following command:

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find the *Info.plist* file inside an *.app* directory like *AWSDeviceFarmiOSReferenceApp.app* in our example:

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                     |-- Info.plist
                     `-- (any other files)
```

3. To find the platform value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

   ```
   $ pip install biplist
   ```

4. Next, open Python and run the following command:

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
   Info.plist')
   print info_plist['CFBundleSupportedPlatforms']
   ```

   A valid iOS application package should produce output like the following:

   ```
   ['iPhoneOS']
   ```

   For more information, see iOS tests in AWS Device Farm.

IOS_APP_WRONG_PLATFORM_DEVICE_VALUE

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We found the platform device value was wrong in the Info.plist file. Please unzip your application and then open Info.plist file inside the .app directory, verify that the value of the key "CFBundleSupportedPlatforms" does not contain the keyword "simulator", and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

1. Copy your application package to your working directory, and then run the following command:

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find the *Info.plist* file inside an *.app* directory like *AWSDeviceFarmiOSReferenceApp.app* in our example:

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                      |-- Info.plist
                      `-- (any other files)
```

3. To find the platform value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

4. Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
Info.plist')
print info_plist['CFBundleSupportedPlatforms']
```

   A valid iOS application package should produce output like the following:

```
['iPhoneOS']
```

   If the iOS application is valid, the value should not contain the keyword `simulator`.

   For more information, see [iOS tests in AWS Device Farm](#).

IOS_APP_FORM_FACTOR_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the form factor value in the Info.plist file. Please unzip your application and then open Info.plist file inside the .app directory, verify that the key "UIDeviceFamily" is specified, and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

1. Copy your application package to your working directory, and then run the following command:

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   You should find the *Info.plist* file inside an *.app* directory like *AWSDeviceFarmiOSReferenceApp.app* in our example:

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                     |-- Info.plist
                     `-- (any other files)
```

3. To find the form factor value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

4. Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
Info.plist')
print info_plist['UIDeviceFamily']
```

A valid iOS application package should produce output like the following:

```
[1, 2]
```

For more information, see iOS tests in AWS Device Farm.

IOS_APP_PACKAGE_NAME_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the package name value in the Info.plist file. Please unzip your application and then open Info.plist file inside the .app directory, verify that the key "CFBundleIdentifier" is specified, and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

1. Copy your application package to your working directory, and then run the following command:

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find the *Info.plist* file inside an *.app* directory like *AWSDeviceFarmiOSReferenceApp.app* in our example:

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                      |-- Info.plist
                      `-- (any other files)
```

3. To find the package name value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

   ```
   $ pip install biplist
   ```

4. Next, open Python and run the following command:

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
   Info.plist')
   print info_plist['CFBundleIdentifier']
   ```

   A valid iOS application package should produce output like the following:

   ```
   Amazon.AWSDeviceFarmiOSReferenceApp
   ```

   For more information, see iOS tests in AWS Device Farm.

IOS_APP_EXECUTABLE_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the executable value in the Info.plist file. Please unzip your application and then open Info.plist file inside the .app directory, verify that the key "CFBundleExecutable" is specified, and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

1. Copy your application package to your working directory, and then run the following command:

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find the *Info.plist* file inside an *.app* directory like *AWSDeviceFarmiOSReferenceApp.app* in our example:

```
.
`-- Payload (directory)
        `-- AWSDeviceFarmiOSReferenceApp.app (directory)
                      |-- Info.plist
                      `-- (any other files)
```

3. To find the executable value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

4. Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
Info.plist')
print info_plist['CFBundleExecutable']
```

   A valid iOS application package should produce output like the following:

```
AWSDeviceFarmiOSReferenceApp
```

   For more information, see iOS tests in AWS Device Farm.

Troubleshooting XCTest tests in AWS Device Farm

The following topic lists error messages that occur during the upload of XCTest tests and recommends workarounds to resolve each error.

Note

The instructions below assume you are using MacOS.

XCTEST_TEST_PACKAGE_UNZIP_FAILED

If you see the following message, follow these steps to fix the issue.

Warning

We could not open your test ZIP file. Please verify that the file is valid and try again.

Make sure that you can unzip the application package without errors. In the following example, the package's name is **swiftExampleTests.xctest-1.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swiftExampleTests.xctest-1.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   A valid XCTest package should produce output like the following:

```
.
`-- swiftExampleTests.xctest (directory)
              |-- Info.plist
              `-- (any other files)
```

   For more information, see Integrating Device Farm with XCTest for iOS.

XCTEST_TEST_PACKAGE_XCTEST_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the .xctest directory inside your test package. Please unzip your test package, verify that the .xctest directory is inside the package, and try again.

In the following example, the package's name is **swiftExampleTests.xctest-1.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swiftExampleTests.xctest-1.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

   If the XCTest package is valid, you will find a directory with a name similar to *swiftExampleTests.xctest* inside the working directory. The name should end with *.xctest*.

```
.
`-- swiftExampleTests.xctest (directory)
              |-- Info.plist
              `-- (any other files)
```

   For more information, see Integrating Device Farm with XCTest for iOS.

XCTEST_TEST_PACKAGE_PLIST_FILE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the Info.plist file inside the .xctest directory. Please unzip your test package and then open the .xctest directory, verify that the Info.plist file is inside the directory, and try again.

In the following example, the package's name is **swiftExampleTests.xctest-1.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swiftExampleTests.xctest-1.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the XCTest package is valid, you will find the *Info.plist* file inside the *.xctest* directory. In our example below, the directory is called *swiftExampleTests.xctest*.

```
.
`-- swiftExampleTests.xctest (directory)
              |-- Info.plist
              `-- (any other files)
```

For more information, see Integrating Device Farm with XCTest for iOS.

XCTEST_TEST_PACKAGE_PACKAGE_NAME_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

Warning

We could not find the package name value in the Info.plist file. Please unzip your test package and then open Info.plist file, verify that the key "CFBundleIdentifier" is specified, and try again.

In the following example, the package's name is **swiftExampleTests.xctest-1.zip**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swiftExampleTests.xctest-1.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find the *Info.plist* file inside an *.xctest* directory like *swiftExampleTests.xctest* in our example:

```
.
`-- swiftExampleTests.xctest (directory)
              |-- Info.plist
              `-- (any other files)
```

3. To find the package name value, you can open Info.plist using Xcode or Python.

For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

4. Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('swiftExampleTests.xctest/Info.plist')
print info_plist['CFBundleIdentifier']
```

A valid XCtest application package should produce output like the following:

```
com.amazon.kanapka.swiftExampleTests
```

For more information, see Integrating Device Farm with XCTest for iOS.

XCTEST_TEST_PACKAGE_EXECUTABLE_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

> **Warning**
>
> We could not find the executable value in the Info.plist file. Please unzip your test package and then open Info.plist file, verify that the key "CFBundleExecutable" is specified, and try again.

In the following example, the package's name is **swiftExampleTests.xctest-1.zip**.

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip swiftExampleTests.xctest-1.zip
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```
   $ tree .
   ```

   You should find the *Info.plist* file inside an *.xctest* directory like *swiftExampleTests.xctest* in our example:

   ```
   .
   `-- swiftExampleTests.xctest (directory)
                 |-- Info.plist
                 `-- (any other files)
   ```

3. To find the package name value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

   ```
   $ pip install biplist
   ```

4. Next, open Python and run the following command:

   ```
   import biplist
   info_plist = biplist.readPlist('swiftExampleTests.xctest/Info.plist')
   print info_plist['CFBundleExecutable']
   ```

   A valid XCtest application package should produce output like the following:

```
swiftExampleTests
```

For more information, see Integrating Device Farm with XCTest for iOS.

Troubleshooting XCTest UI tests in AWS Device Farm

The following topic lists error messages that occur during the upload of XCTest UI tests and recommends workarounds to resolve each error.

> **Note**
>
> The instructions below are based on Linux x86_64 and Mac.

XCTEST_UI_TEST_PACKAGE_UNZIP_FAILED

If you see the following message, follow these steps to fix the issue.

`We could not open your test IPA file. Please verify that the file is valid and try again.`

Make sure that you can unzip the application package without errors. In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

    ```
    $ unzip swift-sample-UI.ipa
    ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

    ```
    $ tree .
    ```

    A valid iOS application package should produce output like the following:

    ```
    .
    `-- Payload (directory)
            `-- swift-sampleUITests-Runner.app (directory)
    ```

```
                    |-- Info.plist
                    |-- Plugins (directory)
                    |       `swift-sampleUITests.xctest (directory)
                    |                       |-- Info.plist
                    |                       `-- (any other files)
                    `-- (any other files)
```

For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_PAYLOAD_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

`We could not find the Payload directory inside your test package. Please unzip your test package, verify that the Payload directory is inside the package, and try again.`

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swift-sample-UI.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the XCTest UI package is valid, you will find the *Payload* directory inside the working directory.

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                    |-- Info.plist
                    |-- Plugins (directory)
                    |       `swift-sampleUITests.xctest (directory)
                    |                       |-- Info.plist
                    |                       `-- (any other files)
                    `-- (any other files)
```

For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_APP_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

```
We could not find the .app directory inside the Payload directory. Please
unzip your test package and then open the Payload directory, verify that
the .app directory is inside the directory, and try again.
```

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```
   $ tree .
   ```

   If the XCTest UI package is valid, you will find an `.app` directory like `swift-sampleUITests-Runner.app` in our example inside the `Payload` directory.

   ```
   .
   `-- Payload (directory)
           `-- swift-sampleUITests-Runner.app (directory)
                         |-- Info.plist
                         |-- Plugins (directory)
                         |       `swift-sampleUITests.xctest (directory)
                         |                         |-- Info.plist
                         |                         `-- (any other files)
                         `-- (any other files)
   ```

   For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_PLUGINS_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

```
We could not find the Plugins directory inside the .app directory. Please
unzip your test package and then open the .app directory, verify that the
Plugins directory is inside the directory, and try again.
```

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```
   $ tree .
   ```

   If the XCTest UI package is valid, you will find the *Plugins* directory inside an *.app* directory. In our example, the directory is called *swift-sampleUITests-Runner.app*.

   ```
   .
   `-- Payload (directory)
           `-- swift-sampleUITests-Runner.app (directory)
                          |-- Info.plist
                          |-- Plugins (directory)
                          |       `swift-sampleUITests.xctest (directory)
                          |                       |-- Info.plist
                          |                       `-- (any other files)
                          `-- (any other files)
   ```

   For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_XCTEST_DIR_MISSING_IN_PLUGINS_DIR

If you see the following message, follow these steps to fix the issue.

```
We could not find the .xctest directory inside the plugins directory.
Please unzip your test package and then open the plugins directory, verify
that the .xctest directory is inside the directory, and try again.
```

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```
   $ tree .
   ```

   If the XCTest UI package is valid, you will find an *.xctest* directory inside the *Plugins* directory. In our example, the directory is called *swift-sampleUITests.xctest*.

   ```
   .
   `-- Payload (directory)
           `-- swift-sampleUITests-Runner.app (directory)
                           |-- Info.plist
                           |-- Plugins (directory)
                           |       `swift-sampleUITests.xctest (directory)
                           |                       |-- Info.plist
                           |                       `-- (any other files)
                           `-- (any other files)
   ```

   For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_PLIST_FILE_MISSING

If you see the following message, follow these steps to fix the issue.

```
We could not find the Info.plist file inside the .app directory. Please
unzip your test package and then open the .app directory, verify that the
Info.plist file is inside the directory, and try again.
```

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swift-sample-UI.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the XCTest UI package is valid, you will find the *Info.plist* file inside the *.app* directory. In our example below, the directory is called *swift-sampleUITests-Runner.app*.

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                      |-- Info.plist
                      |-- Plugins (directory)
                      |       `swift-sampleUITests.xctest (directory)
                      |                       |-- Info.plist
                      |                       `-- (any other files)
                      `-- (any other files)
```

For more information, see [Integrating XCTest UI for iOS with Device Farm](#).

XCTEST_UI_TEST_PACKAGE_PLIST_FILE_MISSING_IN_XCTEST_DIR

If you see the following message, follow these steps to fix the issue.

`We could not find the Info.plist file inside the .xctest directory. Please unzip your test package and then open the .xctest directory, verify that the Info.plist file is inside the directory, and try again.`

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swift-sample-UI.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the XCTest UI package is valid, you will find the *Info.plist* file inside the *.xctest* directory. In our example below, the directory is called *swift-sampleUITests.xctest*.

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                      |-- Info.plist
                      |-- Plugins (directory)
                      |       `swift-sampleUITests.xctest (directory)
                      |                       |-- Info.plist
                      |                       `-- (any other files)
                      `-- (any other files)
```

For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_CPU_ARCHITECTURE_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

`We could not the CPU architecture value in the Info.plist file. Please unzip your test package and then open the Info.plist file inside the .app directory, verify that the key "UIRequiredDeviceCapabilities" is specified, and try again.`

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swift-sample-UI.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find the *Info.plist* file inside an *.app* directory like *swift-sampleUITests-Runner.app* in our example:

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                      |-- Info.plist
                      |-- Plugins (directory)
                      |       `swift-sampleUITests.xctest (directory)
                      |                       |-- Info.plist
                      |                       `-- (any other files)
                      `-- (any other files)
```

3. To find the CPU architecture value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

4. Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/
Info.plist')
print info_plist['UIRequiredDeviceCapabilities']
```

   A valid XCtest UI package should produce output like the following:

```
['armv7']
```

   For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_PLATFORM_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

`We could not find the platform value in the Info.plist. Please unzip your test package and then open the Info.plist file inside the .app directory,` `verify that the key "CFBundleSupportedPlatforms" is specified, and try again.`

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```
   $ tree .
   ```

   You should find the *`Info.plist`* file inside an *`.app`* directory like *`swift-sampleUITests-Runner.app`* in our example:

   ```
   .
   `-- Payload (directory)
           `-- swift-sampleUITests-Runner.app (directory)
                          |-- Info.plist
                          |-- Plugins (directory)
                          |       `swift-sampleUITests.xctest (directory)
                          |                         |-- Info.plist
                          |                         `-- (any other files)
                          `-- (any other files)
   ```

3. To find the platform value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

   ```
   $ pip install biplist
   ```

4. Next, open Python and run the following command:

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
   print info_plist['CFBundleSupportedPlatforms']
   ```

   A valid XCtest UI package should produce output like the following:

```
['iPhoneOS']
```

For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_WRONG_PLATFORM_DEVICE_VALUE

If you see the following message, follow these steps to fix the issue.

```
We found the platform device value was wrong in the Info.plist file. Please
unzip your test package and then open the Info.plist file inside the .app
directory, verify that the value of the key "CFBundleSupportedPlatforms"
does not contain the keyword "simulator", and try again.
```

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```
   $ tree .
   ```

   You should find the *Info.plist* file inside an *.app* directory like *swift-sampleUITests-Runner.app* in our example:

   ```
   .
   `-- Payload (directory)
           `-- swift-sampleUITests-Runner.app (directory)
                           |-- Info.plist
                           |-- Plugins (directory)
                           |       `swift-sampleUITests.xctest (directory)
                           |                       |-- Info.plist
                           |                       `-- (any other files)
                           `-- (any other files)
   ```

3. To find the platform value, you can open Info.plist using Xcode or Python.

For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

4. Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['CFBundleSupportedPlatforms']
```

A valid XCtest UI package should produce output like the following:

```
['iPhoneOS']
```

If the XCTest UI package is valid, the value should not contain the keyword `simulator`.

For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_FORM_FACTOR_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

`We could not the form factor value in the Info.plist. Please unzip your test package and then open the Info.plist file inside the .app directory, verify that the key "UIDeviceFamily" is specified, and try again.`

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swift-sample-UI.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find the *`Info.plist`* file inside an *`.app`* directory like *`swift-sampleUITests-Runner.app`* in our example:

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                      |-- Info.plist
                      |-- Plugins (directory)
                      |       `swift-sampleUITests.xctest (directory)
                      |                       |-- Info.plist
                      |                       `-- (any other files)
                      `-- (any other files)
```

3. To find the form factor value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

   ```
   $ pip install biplist
   ```

4. Next, open Python and run the following command:

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
   print info_plist['UIDeviceFamily']
   ```

   A valid XCtest UI package should produce output like the following:

   ```
   [1, 2]
   ```

   For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_PACKAGE_NAME_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

`We could not find the package name value in the Info.plist file. Please unzip your test package and then open the Info.plist file inside the .app directory, verify that the key "CFBundleIdentifier" is specified, and try again.`

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swift-sample-UI.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find the *Info.plist* file inside an *.app* directory like *swift-sampleUITests-Runner.app* in our example:

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                      |-- Info.plist
                      |-- Plugins (directory)
                      |       `swift-sampleUITests.xctest (directory)
                      |                       |-- Info.plist
                      |                       `-- (any other files)
                      `-- (any other files)
```

3. To find the package name value, you can open Info.plist using Xcode or Python.

For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

4. Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['CFBundleIdentifier']
```

A valid XCtest UI package should produce output like the following:

```
com.apple.test.swift-sampleUITests-Runner
```

For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_EXECUTABLE_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

```
We could not find the executable value in the Info.plist file. Please
unzip your test package and then open the Info.plist file inside the .app
directory, verify that the key "CFBundleExecutable" is specified, and try
again.
```

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip swift-sample-UI.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```
   $ tree .
   ```

   You should find the *`Info.plist`* file inside an *`.app`* directory like *`swift-sampleUITests-Runner.app`* in our example:

   ```
   .
   `-- Payload (directory)
           `-- swift-sampleUITests-Runner.app (directory)
                         |-- Info.plist
                         |-- Plugins (directory)
                         |       `swift-sampleUITests.xctest (directory)
                         |                        |-- Info.plist
                         |                        `-- (any other files)
                         `-- (any other files)
   ```

3. To find the executable value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

   ```
   $ pip install biplist
   ```

4. Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['CFBundleExecutable']
```

A valid XCtest UI package should produce output like the following:

```
XCTRunner
```

For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_TEST_PACKAGE_NAME_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

```
We could not find the package name value in the Info.plist file inside
the .xctest directory. Please unzip your test package and then open
the Info.plist file inside the .xctest directory, verify that the key
"CFBundleIdentifier" is specified, and try again.
```

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swift-sample-UI.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find the `Info.plist` file inside an `.app` directory like `swift-sampleUITests-Runner.app` in our example:

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                      |-- Info.plist
                      |-- Plugins (directory)
```

```
                          |       `swift-sampleUITests.xctest (directory)
                          |                       |-- Info.plist
                          |                       `-- (any other files)
                          `-- (any other files)
```

3. To find the package name value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

4. Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Plugins/
swift-sampleUITests.xctest/Info.plist')
print info_plist['CFBundleIdentifier']
```

   A valid XCtest UI package should produce output like the following:

```
com.amazon.swift-sampleUITests
```

   For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_TEST_EXECUTABLE_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

```
We could not find the executable value in the Info.plist file inside
the .xctest directory. Please unzip your test package and then open
the Info.plist file inside the .xctest directory, verify that the key
"CFBundleExecutable" is specified, and try again.
```

In the following example, the package's name is **swift-sample-UI.ipa**.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swift-sample-UI.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find the *Info.plist* file inside an *.app* directory like *swift-sampleUITests-Runner.app* in our example:

```
.
`-- Payload (directory)
        `-- swift-sampleUITests-Runner.app (directory)
                      |-- Info.plist
                      |-- Plugins (directory)
                      |       `swift-sampleUITests.xctest (directory)
                      |                       |-- Info.plist
                      |                       `-- (any other files)
                      `-- (any other files)
```

3. To find the executable value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

4. Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Plugins/
swift-sampleUITests.xctest/Info.plist')
print info_plist['CFBundleExecutable']
```

   A valid XCtest UI package should produce output like the following:

```
swift-sampleUITests
```

   For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_MULTIPLE_APP_DIRS

If you see the following message, follow these steps to fix the issue.

```
We found multiple .app directories inside your test package. Please unzip
your test package, verify that only a single .app directory is present
inside the package, then try again.
```

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip swift-sample-UI.zip
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```
   $ tree .
   ```

   If the XCTest UI package is valid, you should find only single `.app` directory like `swift-sampleUITests-Runner.app` in our example inside the .zip test package.

   ```
   .
   `--swift-sample-UI.zip--(directory)
       `-- swift-sampleUITests-Runner.app (directory)
               |-- Info.plist
               |-- Plugins (directory)
               |       `swift-sampleUITests.xctest (directory)
              |            |-- Info.plist
              |            `-- (any other files)
               `-- (any other files)
       `-- (any other files)
   ```

   For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_MULTIPLE_IPA_DIRS

If you see the following message, follow these steps to fix the issue.

```
We found multiple .ipa directories inside your test package. Please unzip
your test package, verify that only a single .ipa directory is present
inside the package, then try again.
```

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swift-sample-UI.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the XCTest UI package is valid, you should find only single `.ipa` directory like `sampleUITests.ipa` in our example inside the .zip test package.

```
.
`--swift-sample-UI.zip--(directory)
    `-- sampleUITests.ipa (directory)
            `-- Payload (directory)
                `-- swift-sampleUITests-Runner.app (directory)
    `-- (any other files)
```

For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_BOTH_APP_AND_IPA_DIR_PRESENT

If you see the following message, follow these steps to fix the issue.

```
We found both .app and .ipa files inside your test package. Please unzip
your test package, verify that only a single .app or .ipa file is present
inside the package, then try again.
```

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swift-sample-UI.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the XCTest UI package is valid, you should find either `.ipa` directory like `sampleUITests.ipa` or `.app` directory like `swift-sampleUITests-Runner.app` in our example inside the .zip test package. You can refer to an example of valid XCTEST_UI Test package in our documentation on Integrating XCTest UI for iOS with Device Farm.

```
.
`--swift-sample-UI.zip--(directory)
    `-- sampleUITests.ipa (directory)
            `-- Payload (directory)
                `-- swift-sampleUITests-Runner.app (directory)
   `-- (any other files)
```

or

```
.
`--swift-sample-UI.zip--(directory)
    `-- swift-sampleUITests-Runner.app (directory)
            |-- Info.plist
            |-- Plugins (directory)
            `-- (any other files)
   `-- (any other files)
```

For more information, see Integrating XCTest UI for iOS with Device Farm.

XCTEST_UI_TEST_PACKAGE_PAYLOAD_DIR_PRESENT_IN_ZIP

If you see the following message, follow these steps to fix the issue.

```
We found a Payload directory inside your .zip test package. Please unzip
your test package, ensure that a Payload directory is not present in the
package, then try again.
```

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip swift-sample-UI.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the XCTest UI package is valid, you should not find a Payload Directory inside your test package.

```
.
`--swift-sample-UI.zip--(directory)
    `-- swift-sampleUITests-Runner.app (directory)
            |-- Info.plist
            |-- Plugins (directory)
            `-- (any other files)
   `-- Payload (directory) [This directory should not be present]
            |-- (any other files)
   `-- (any other files)
```

For more information, see Integrating XCTest UI for iOS with Device Farm.

Security in AWS Device Farm

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to AWS Device Farm, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Device Farm. The following topics show you how to configure Device Farm to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Device Farm resources.

Topics

Identity and access management in AWS Device Farm

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in Device Farm.

Service user – If you use the Device Farm service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Device Farm features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Device Farm, see Troubleshooting AWS Device Farm identity and access.

Service administrator – If you're in charge of Device Farm resources at your company, you probably have full access to Device Farm. It's your job to determine which Device Farm features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Device Farm, see How AWS Device Farm works with IAM.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Device Farm. To view example Device Farm identity-based policies that you can use in IAM, see AWS Device Farm identity-based policy examples.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and AWS Multi-factor authentication in IAM in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

IAM Users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. To temporarily assume an IAM role in the AWS Management Console, you can switch from a user to an IAM role (console). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Methods to assume a role in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Create a role for a third-party identity provider (federation) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that

requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

- **Service role –** A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Create a role to delegate permissions to an AWS service in the *IAM User Guide*.
- **Service-linked role –** A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

- **Applications running on Amazon EC2 –** You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Use an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

How AWS Device Farm works with IAM

Before you use IAM to manage access to Device Farm, you should understand which IAM features are available to use with Device Farm. To get a high-level view of how Device Farm and other AWS services work with IAM, see AWS Services That Work with IAM in the *IAM User Guide*.

Topics

- Device Farm identity-based policies
- Device Farm resource-based policies
- Access control lists
- Authorization based on Device Farm tags
- Device Farm IAM roles

Device Farm identity-based policies

With IAM identity-based policies, you can specify allowed or denied actions and resources and the conditions under which actions are allowed or denied. Device Farm supports specific actions, resources, and condition keys. To learn about all of the elements that you use in a JSON policy, see IAM JSON Policy Elements Reference in the *IAM User Guide*.

Actions

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

Policy actions in Device Farm use the following prefix before the action: `devicefarm:`. For example, to grant someone permission to start Selenium sessions with the Device Farm desktop browser testing `CreateTestGridUrl` API operation, you include the `devicefarm:CreateTestGridUrl` action in the policy. Policy statements must include either an `Action` or `NotAction` element. Device Farm defines its own set of actions that describe tasks that you can perform with this service.

To specify multiple actions in a single statement, separate them with commas as follows:

```
"Action": [
      "devicefarm:action1",
      "devicefarm:action2"
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word `List`, include the following action:

```
"Action": "devicefarm:List*"
```

To see a list of Device Farm actions, see Actions defined by AWS Device Farm in the *IAM Service Authorization Reference*.

Resources

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its [Amazon Resource Name (ARN)](). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

The Amazon EC2 instance resource has the following ARN:

```
arn:${Partition}:ec2:${Region}:${Account}:instance/${InstanceId}
```

For more information about the format of ARNs, see [Amazon Resource Names (ARNs) and AWS Service Namespaces]().

For example, to specify the `i-1234567890abcdef0` instance in your statement, use the following ARN:

```
"Resource": "arn:aws:ec2:us-east-1:123456789012:instance/i-1234567890abcdef0"
```

To specify all instances that belong to an account, use the wildcard (*):

```
"Resource": "arn:aws:ec2:us-east-1:123456789012:instance/*"
```

Some Device Farm actions, such as those for creating resources, cannot be performed on a resource. In those cases, you must use the wildcard (*).

```
"Resource": "*"
```

Many Amazon EC2 API actions involve multiple resources. For example, `AttachVolume` attaches an Amazon EBS volume to an instance, so an IAM user must have permissions to use the volume and the instance. To specify multiple resources in a single statement, separate the ARNs with commas.

```
"Resource": [
      "resource1",
      "resource2"
```

To see a list of Device Farm resource types and their ARNs, see [Resource types defined by AWS Device Farm](#) in the *IAM Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see [Actions defined by AWS Device Farm](#) in the *IAM Service Authorization Reference*.

Condition keys

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use [condition operators](#), such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see [IAM policy elements: variables and tags](#) in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see [AWS global condition context keys](#) in the *IAM User Guide*.

Device Farm defines its own set of condition keys and also supports the use of some global condition keys. To see all AWS global condition keys, see [AWS Global Condition Context Keys](#) in the *IAM User Guide*.

To see a list of Device Farm condition keys, see [Condition keys for AWS Device Farm](#) in the *IAM Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see [Actions defined by AWS Device Farm](#) in the *IAM Service Authorization Reference*.

Examples

To view examples of Device Farm identity-based policies, see AWS Device Farm identity-based policy examples.

Device Farm resource-based policies

Device Farm does not support resource-based policies.

Access control lists

Device Farm does not support access control lists (ACLs).

Authorization based on Device Farm tags

You can attach tags to Device Farm resources or pass tags in a request to Device Farm. To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys. For more information about tagging Device Farm resources, see *Tagging in Device Farm*.

To view an example identity-based policy for limiting access to a resource based on the tags on that resource, see Viewing Device Farm desktop browser testing projects based on tags.

Device Farm IAM roles

An IAM role is an entity in your AWS account that has specific permissions.

Using temporary credentials with Device Farm

Device Farm supports the use of temporary credentials.

You can use temporary credentials to sign in with federation to assume an IAM role orcross-account role. You obtain temporary security credentials by calling AWS STS API operations such as AssumeRole or GetFederationToken.

Service-linked roles

Service-linked roles allow AWS services to access resources in other services to complete an action on your behalf. Service-linked roles appear in your IAM account and are owned by the service. An IAM administrator can view, but cannot edit, the permissions for service-linked roles.

Device Farm uses service-linked roles in the Device Farm desktop browser testing feature. For information on these roles, see Using Service-Linked Roles in Device Farm desktop browser testing in the developer guide.

Service roles

Device Farm does not support service roles.

This feature allows a service to assume a service role on your behalf. This role allows the service to access resources in other services to complete an action on your behalf. Service roles appear in your IAM account and are owned by the account. This means that an IAM administrator can change the permissions for this role. However, doing so might break the functionality of the service.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choose between managed policies and inline policies in the *IAM User Guide*.

The following table outlines the Device Farm AWS managed policies.

Change	Description	Date
AWSDeviceFarmFullAccess	Provides full access to all AWS Device Farm operations.	July 15, 2015
AWSServiceRoleForDeviceFarmTestGrid	Allows Device Farm to access AWS resources on your behalf.	May 20, 2021

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root

user, regardless of whether they belong to your organization. For more information about Organizations and RCPs, including a list of AWS services that support RCPs, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.

- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

AWS Device Farm identity-based policy examples

By default, IAM users and roles don't have permission to create or modify Device Farm resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the IAM users or groups that require those permissions.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating Policies on the JSON Tab in the *IAM User Guide*.

Topics

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Device Farm resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see Validate policies with IAM Access Analyzer in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Secure API access with MFA in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Accessing one Device Farm desktop browser testing project

In this example, you want to grant an IAM user in your AWS account access to one of your Device Farm destktop browser testing projects, `arn:aws:devicefarm:us-west-2:111122223333:testgrid-project:123e4567-e89b-12d3-a456-426655441111`. You want the account to be able to see items related to the project.

In addition to the `devicefarm:GetTestGridProject` endpoint, the account must have the `devicefarm:ListTestGridSessions`, `devicefarm:GetTestGridSession`, `devicefarm:ListTestGridSessionActions`, and `devicefarm:ListTestGridSessionArtifacts` endpoints.

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"GetTestGridProject",
         "Effect":"Allow",
         "Action":[
            "devicefarm:GetTestGridProject"
         ],
         "Resource":"arn:aws:devicefarm:us-west-2:111122223333:testgrid-
project:123e4567-e89b-12d3-a456-426655441111"
      },
      {
         "Sid":"ViewProjectInfo",
         "Effect":"Allow",
         "Action":[
            "devicefarm:ListTestGridSessions",
            "devicefarm:ListTestGridSessionActions",
            "devicefarm:ListTestGridSessionArtifacts"
         ],
         "Resource":"arn:aws:devicefarm:us-west-2:111122223333:testgrid-*:123e4567-
e89b-12d3-a456-426655441111/*"
      }
   ]
}
```

If you are using CI systems, you should give each CI runner unique access credentials. For example, a CI system is unlikely to need more permissions than `devicefarm:ScheduleRun` or `devicefarm:CreateUpload`. The following IAM policy outlines a minimal policy to allow a CI runner to start a test of a new Device Farm native app test by creating an upload and using it to schedule a test run:

```
{
   "Version":"2012-10-17",
   "Statement": [
      {
         "$id":"scheduleTestRuns",
         "effect":"Allow",
         "Action": [ "devicefarm:CreateUpload","devicefarm:ScheduleRun" ],
```

```
            "Resource": [
               "arn:aws:devicefarm:us-west-2:111122223333:project:123e4567-e89b-12d3-
a456-426655440000",
               "arn:aws:devicefarm:us-west-2:111122223333:*:123e4567-e89b-12d3-
a456-426655440000/*",
               ]
          }
     ]
}
```

Viewing Device Farm desktop browser testing projects based on tags

You can use conditions in your identity-based policy to control access to Device Farm resources based on tags. This example shows how you might create a policy that allows the viewing of projects and sessions. Permission is granted if the `Owner` tag of the requested resource matches the username of the requesting account.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ListTestGridProjectSessions",
            "Effect": "Allow",
            "Action": [
              "devicefarm:ListTestGridSession*",
              "devicefarm:GetTestGridSession",
              "devicefarm:ListTestGridProjects"
              ],
            "Resource": [
               "arn:aws:devicefarm:us-west-2:testgrid-project:*/*"
               "arn:aws:devicefarm:us-west-2:testgrid-session:*/*"
               ],
            "Condition": {
               "StringEquals": {"aws:TagKey/Owner":"${aws:username}"}
            }
        }
    ]
}
```

You can attach this policy to the IAM users in your account. If a user named richard-roe attempts to view a Device Farm project or session, the project must be tagged Owner=richard-roe or owner=richard-roe. Otherwise, the user is denied access. The condition tag key Owner matches both Owner and owner because condition key names are not case sensitive. For more information, see IAM JSON Policy Elements: Condition in the *IAM User Guide*.

Troubleshooting AWS Device Farm identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Device Farm and IAM.

I am not authorized to perform an action in Device Farm

If you receive an error in the AWS Management Console that says you're not authorized to perform an action, you must contact your administrator for assistance. Your administrator is the person who provided you with your user name and password.

The following example error occurs when the IAM user, mateojackson, tries to use the console to view details about a run, but does not have devicefarm:GetRun permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 devicefarm:GetRun on resource: arn:aws:devicefarm:us-west-2:123456789101:run:123e4567-
e89b-12d3-a456-426655440000/123e4567-e89b-12d3-a456-426655441111
```

In this case, Mateo asks his administrator to update his policies to allow him to access the devicefarm:GetRun on resource arn:aws:devicefarm:us-west-2:123456789101:run:123e4567-e89b-12d3-a456-426655440000/123e4567-e89b-12d3-a456-426655441111 resource using the devicefarm:GetRun action.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the iam:PassRole action, your policies must be updated to allow you to pass a role to Device Farm.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Device Farm. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to view my access keys

After you create your IAM user access keys, you can view your access key ID at any time. However, you can't view your secret access key again. If you lose your secret key, you must create a new access key pair.

Access keys consist of two parts: an access key ID (for example, `AKIAIOSFODNN7EXAMPLE`) and a secret access key (for example, `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`). Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests. Manage your access keys as securely as you do your user name and password.

> **Important**
>
> Do not provide your access keys to a third party, even to help find your canonical user ID. By doing this, you might give someone permanent access to your AWS account.

When you create an access key pair, you are prompted to save the access key ID and secret access key in a secure location. The secret access key is available only at the time you create it. If you lose your secret access key, you must add new access keys to your IAM user. You can have a maximum of two access keys. If you already have two, you must delete one key pair before creating a new one. To view instructions, see Managing access keys in the *IAM User Guide*.

I'm an administrator and want to allow others to access Device Farm

To allow others to access Device Farm, you must grant permission to the people or applications that need access. If you are using AWS IAM Identity Center to manage people and applications, you assign permission sets to users or groups to define their level of access. Permission sets automatically create and assign IAM policies to IAM roles that are associated with the person or application. For more information, see Permission sets in the *AWS IAM Identity Center User Guide*.

If you are not using IAM Identity Center, you must create IAM entities (users or roles) for the people or applications that need access. You must then attach a policy to the entity that grants them the correct permissions in Device Farm. After the permissions are granted, provide the credentials to the user or application developer. They will use those credentials to access AWS. To learn more about creating IAM users, groups, policies, and permissions, see IAM Identities and Policies and permissions in IAM in the *IAM User Guide*.

I want to allow people outside of my AWS account to access my Device Farm resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Device Farm supports these features, see How AWS Device Farm works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see Cross account resource access in IAM in the *IAM User Guide*.

Compliance validation for AWS Device Farm

Third-party auditors assess the security and compliance of AWS Device Farm as part of multiple AWS compliance programs. These include SOC, PCI, FedRAMP, HIPAA, and others. AWS Device Farm is not in scope of any AWS compliance programs.

For a list of AWS services in scope of specific compliance programs, see AWS Services in Scope by Compliance Program. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using Device Farm is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security and Compliance Quick Start Guides – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- Evaluating Resources with Rules in the *AWS Config Developer Guide* – AWS Config assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS that helps you check your compliance with security industry standards and best practices.

Data protection in AWS Device Farm

The AWS shared responsibility model applies to data protection in AWS Device Farm (Device Farm). As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.

- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-3.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Device Farm or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Encryption in transit

The Device Farm endpoints only support signed HTTPS (SSL/TLS) requests except where otherwise noted. All content retrieved from or placed in Amazon S3 through upload URLs is encrypted using SSL/TLS. For more information on how HTTPS requests are signed in AWS, see Signing AWS API requests in the AWS General Reference.

It is your responsibility to encrypt and secure any communications that your tested applications make and any applications installed in the process of running on-device tests.

Encryption at rest

Device Farm's desktop browser testing feature supports encryption at rest for artifacts generated during tests.

Device Farm's physical mobile device testing data is not encrypted at rest.

Data retention

Data in Device Farm is retained for a limited time. After the retention period expires, the data is removed from Device Farm's backing storage.

Content type	Retention period (days)	Metadata Retention period (days)
Uploaded applications	30	30
Uploaded test packages	30	30
Logs	400	400
Video recordings and other artifacts	400	400

It is your responsibility to archive any content that you want to retain for longer periods.

Data management

Data in Device Farm is managed differently depending on which features are used. This section explains how data is managed while and after you use Device Farm.

Desktop browser testing

Instances used during Selenium sessions are not saved. All data generated as a result of browser interactions is discarded when the session ends.

This feature currently supports encryption at rest for artifacts generated during the test.

Physical device testing

The following sections provide information about the steps AWS takes to clean up or destroy devices after you have used Device Farm.

Device Farm's physical mobile device testing data is not encrypted at rest.

Public device fleets

After test execution is complete, Device Farm performs a series of cleanup tasks on each device in the public device fleet, including uninstallation of your app. If we cannot verify uninstallation of your app or any of the other cleanup steps, the device receives a factory reset before it is put back into use.

> **Note**
>
> It is possible for data to persist between sessions in some cases, especially if you make use of the device system outside the context of your app. For this reason, and because Device Farm captures video and logs of activity taking place during your use of each device, we recommend that you do not enter sensitive information (for example, Google account or Apple ID), personal information, and other security-sensitive details during your automated test and remote access sessions.

Private devices

After expiration or termination of your private device contract, the device is removed from use and securely destroyed in accordance with AWS destruction policies. For more information, see Private devices in AWS Device Farm.

Key management

Currently, Device Farm does not offer any external key management for encryption of data, at rest or in transit.

Internetwork traffic privacy

Device Farm can be configured, for private devices only, to use Amazon VPC endpoints to connect to your resources in AWS. Access to any non-public AWS infrastructure associated with your account (for example, Amazon EC2 instances without a public IP address) must use an Amazon VPC endpoint. Regardless of VPC endpoint configuration, Device Farm isolates your traffic from other users throughout the Device Farm network.

Your connections outside the AWS network are not guaranteed to be secured or safe, and it is your responsibility to secure any internet connections your applications make.

Resilience in AWS Device Farm

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

Because Device Farm is available in the `us-west-2` Region only, we strongly recommend that you implement backup and recovery processes. Device Farm should not be the only source of any uploaded content.

Device Farm makes no guarantees of the availability of public devices. These devices are taken in and out of the public device pool depending on a variety of factors, such as failure rate and quarantine status. We do not recommend that you depend on the availability of any one device in the public device pool.

Infrastructure security in AWS Device Farm

As a managed service, AWS Device Farm is protected by the AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well‐Architected Framework*.

You use AWS published API calls to access Device Farm through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

Infrastructure security for physical device testing

Devices are physically separated during physical device testing. Network isolation prevents cross-device communication over wireless networks.

Public devices are shared, and Device Farm makes a best-effort attempt at keeping devices safe over time. Certain actions, such as attempts to acquire complete administrator rights on a device (a practice referred to as *rooting* or *jailbreaking*), cause public devices to become quarantined. They are removed from the public pool automatically and placed into manual review.

Private devices are accessible only by AWS accounts explicitly authorized to do so. Device Farm physically isolates these devices from other devices and keeps them on a separate network.

On privately managed devices, tests can be configured to use an Amazon VPC endpoint to secure connections in and out of your AWS account.

Infrastructure security for desktop browser testing

When you use the desktop browser testing feature, all test sessions are separated from one another. Selenium instances cannot cross-communicate without an intermediate third party, external to AWS.

All traffic to Selenium WebDriver controllers must be made through the HTTPS endpoint generated with `createTestGridUrl`.

The desktop browser testing feature does not support Amazon VPC endpoint configuration at this time. You are responsible for making sure that each Device Farm test instance has secure access to resources it tests.

Configuration vulnerability analysis and management in Device Farm

Device Farm allows you to run software that is not actively maintained or patched by the vendor, such as the OS vendor, hardware vendor, or phone carrier. Device Farm makes a best-effort attempt to maintain up to date software, but makes no guarantees that any particular version of the software on a physical device is up to date, by design allowing potentially vulnerable software to be put into use.

For example, if a test is performed on a device running Android 4.4.2, Device Farm makes no guarantee that the device is patched against the vulnerability in Android known as StageFright. It is up to the vendor (and sometimes the carrier) of the device to provide security updates to devices. A malicious application that uses this vulnerability is not guaranteed to be caught by our automated quarantining.

Private devices are maintained as per your agreement with AWS.

Device Farm makes a best-faith effort to prevent customer applications from actions such as *rooting* or *jailbreaking*. Device Farm removes devices that are quarantined from the public pool until they have been manually reviewed.

You are responsible for keeping any libraries or versions of software that you use in your tests, such as Python wheels and Ruby gems, up to date. Device Farm recommends that you update your test libraries.

These resources can help keep your test dependencies up to date:

- For information about how to secure Ruby gems, see Security Practices on the RubyGems website.
- For information about the safety package used by Pipenv and endorsed by the Python Packaging Authority to scan your dependency graph for known vulnerabilities, see the Detection of Security Vulnerabilities on GitHub.
- For information about the Open Web Application Security Project (OWASP) Maven dependency checker, see OWASP DependencyCheck on the OWASP website.

It is important to remember that even if an automated system does not believe there are any known security issues, it does not mean that there are no security issues. Always use due diligence when using libraries or tools from third parties and verify cryptographic signatures when possible or reasonable.

Incident response in Device Farm

Device Farm continuously monitors devices for behaviors that might indicate security issues. If AWS is made aware of a case where customer data, such as test results or files written to a public device, is accessible by another customer, AWS contacts affected customers, according to the standard incident alerting and reporting policies used throughout AWS services.

Logging and monitoring in Device Farm

This service supports AWS CloudTrail, which is a service that records AWS calls for your AWS account and delivers log files to an Amazon S3 bucket. By using information collected by CloudTrail, you can determine what requests were successfully made to AWS services, who made the request, when it was made, and so on. To learn more about CloudTrail, including how to turn it on and find your log files, see the AWS CloudTrail User Guide.

For information about using CloudTrail with Device Farm, see Logging AWS Device Farm API calls with AWS CloudTrail.

Security best practices for Device Farm

Device Farm provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

- Grant any continuous integration (CI) system you use the least privilege possible under IAM. Consider using temporary credentials for each CI system test so that even if a CI system is compromised, it cannot make spurious requests. For more information about temporary credentials, see the IAM User Guide.
- Use `adb` commands in a custom test environment to clean up any content created by your application. For more information about custom test environments, see *Custom test environments*.

Limits in AWS Device Farm

The following list describes current AWS Device Farm limits:

- The maximum file size of an app that you can upload is 4 GB.
- There is no limit to the number of devices that you can include in a test run. However, the maximum number of devices that Device Farm will test simultaneously during a test run is five. (This number can be increased upon request.)
- There is no limit to the number of runs that you can schedule.
- There is a 150-minute limit to the duration of a remote access session.
- There is a 150-minute limit to the duration of an automated test run.
- The maximum number of in-flight jobs, including pending queued jobs across your account, is 250. This is a soft limit.
- There's no limit to the number of devices you can include in a test run. The number of devices (jobs) that can execute your tests in parallel at any given time is equal to your account-level concurrency. The default account-level concurrency for metered use in Device Farm is five.

  The metered concurrency limit can be increased upon request up to a certain threshold depending on the use case. The default account-level concurrency for unmetered use is equal to the number of slots you are subscribed to for that platform.

  For more information concerning the default metered concurrency limits or quotas in general, see the Quotas page.
- Device Farm follows a token-bucket algorithm to throttle API call rates. For example, imagine creating a bucket that holds tokens. Each token represents one transaction, and one API call uses up a token. Tokens are added to the bucket at a fixed rate (e.g., 10 tokens per second), and the bucket has a maximum capacity (e.g., 100 tokens). When a request or packet arrives, it must claim a token from the bucket to be processed. If there are enough tokens, the request is allowed through and tokens are removed. If there aren't enough tokens, the request is either delayed or dropped, depending on the implementation.

  In Device Farm, this is how the algorithm is implemented:

  - The **Burst API requests** is the maximum number of requests the service is able to respond to for a specified API in a specified customer account ID. In other words, this is the capacity of the bucket. You can call the API as many times as there are tokens remaining in the bucket, and each request consumes one token.

- The **Transactions-per-second (TPS) rate** is the minimum rate at which your API requests can be executed. In other words, this is the rate at which the bucket refills with tokens per second. For example, if an API has a burst number of ten but a TPS of one, you could call it ten times instantly. However, the bucket would only regain tokens at a rate of one token per second, resulting in being throttled to one call per second unless you stopped calling the API to let the bucket refill.

Here are the rates for Device Farm APIs:

- For List and Get APIs, the **Burst API requests** capacity is `50`, and the **Transactions-per-second (TPS) rate** is `10`.
- For all other APIs, the **Burst API requests** capacity is `10`, and the **Transactions-per-second (TPS) rate** is `1`.

Tools and plugins for AWS Device Farm

This section contains links and information about working with AWS Device Farm tools and plugins. You can find Device Farm plugins on AWS Labs on GitHub.

If you are an Android developer, we also have an AWS Device Farm sample app for Android on GitHub. You can use the app and example tests as a reference for your own Device Farm test scripts.

Topics

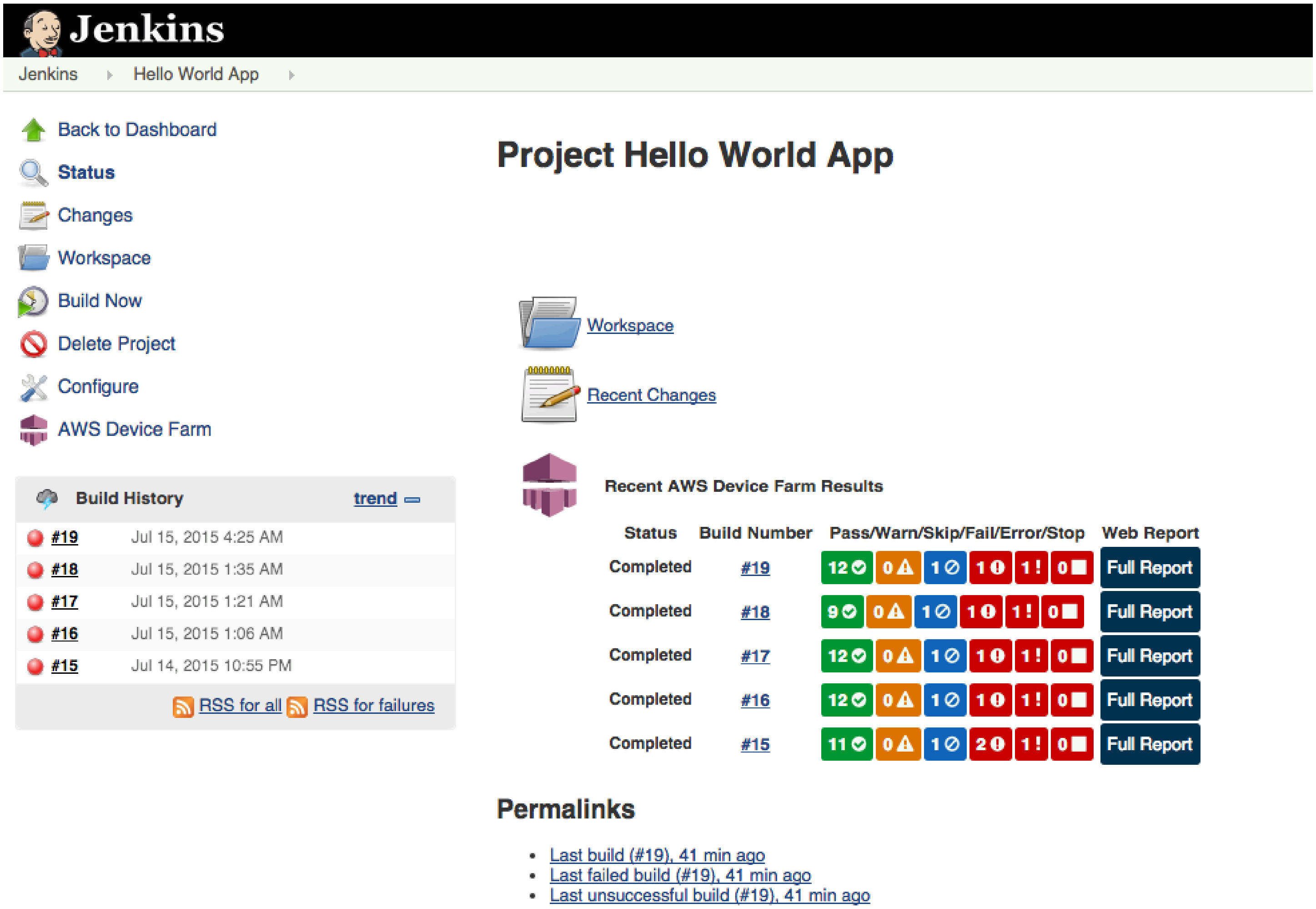
Integrating Device Farm with a Jenkins CI server

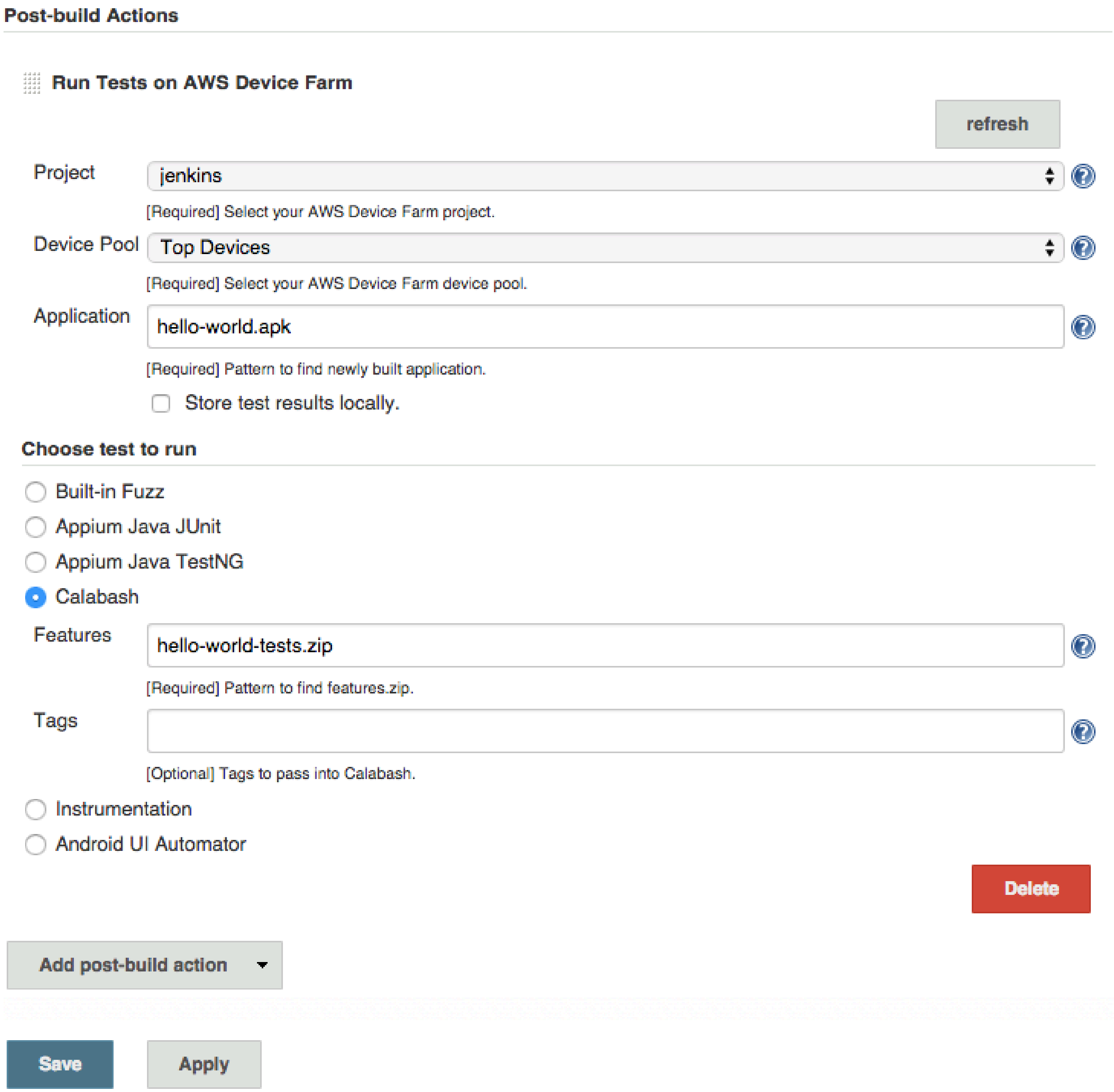
The Jenkins CI plugin provides AWS Device Farm functionality from your own Jenkins continuous integration (CI) server. For more information, see Jenkins (software).

> **Note**
>
> To download the Jenkins plugin, go to GitHub and follow the instructions in Step 1: Installing the Jenkins CI plugin for AWS Device Farm.

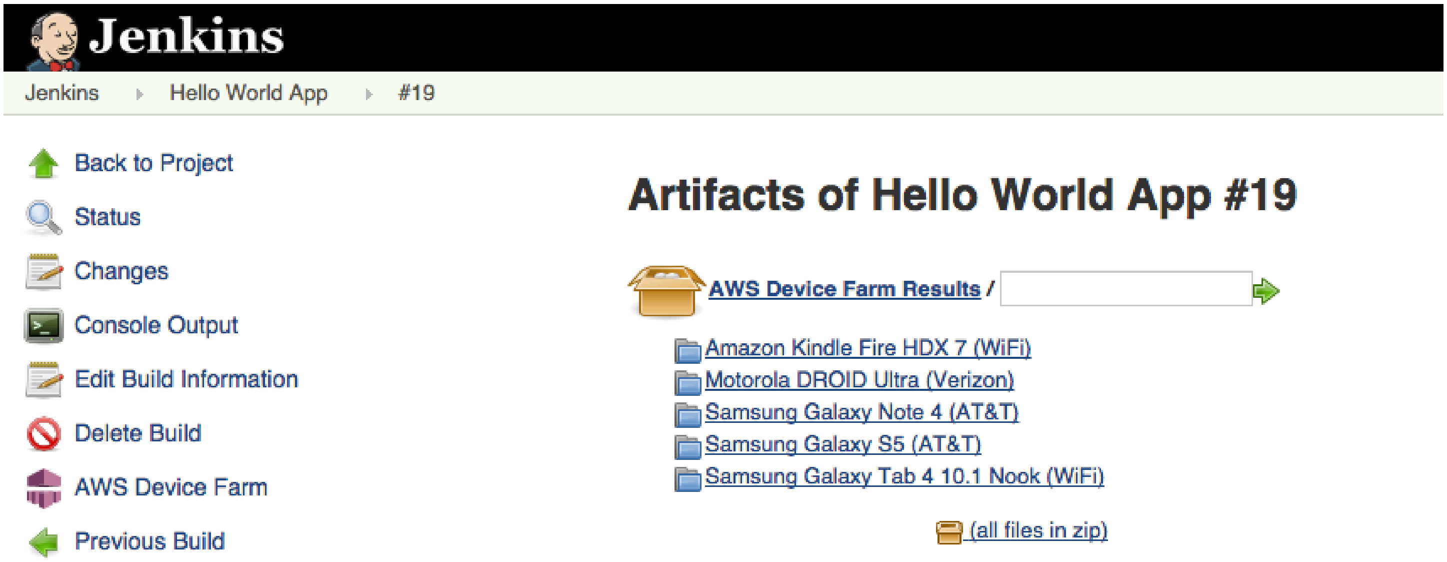
This section contains a series of procedures to set up and use the Jenkins CI plugin with AWS Device Farm.

The following images show the features of the Jenkins CI plugin.


Jenkins
Jenkins
Hello World App
Back to Dashboard
Status
Changes
Workspace
Build Now
Delete Project
Configure
AWS Device Farm
Build History
trend
#19 Jul 15, 2015 4:25 AM
#18 Jul 15, 2015 1:35 AM
#17 Jul 15, 2015 1:21 AM
#16 Jul 15, 2015 1:06 AM
#15 Jul 14, 2015 10:55 PM
RSS for all
RSS for failures
Project Hello World App
Workspace
Recent Changes
Recent AWS Device Farm Results
Status Build Number Pass/Warn/Skip/Fail/Error/Stop Web Report
Completed #19 12 0 1 1 1 0 Full Report
Completed #18 9 0 1 1 1 0 Full Report
Completed #17 12 0 1 1 1 0 Full Report
Completed #16 12 0 1 1 1 0 Full Report
Completed #15 11 0 1 2 1 0 Full Report
Permalinks
Last build (#19), 41 min ago
Last failed build (#19), 41 min ago
Last unsuccessful build (#19), 41 min ago



The plugin can also pull down all the test artifacts (logs, screenshots, etc.) locally:



Topics

- [Dependencies]
- [Step 1: Installing the Jenkins CI plugin for AWS Device Farm]
- [Step 2: Creating an AWS Identity and Access Management user for your Jenkins CI Plugin for AWS Device Farm]
- [Step 3: Configuring the Jenkins CI plugin for the first time in AWS Device Farm]
- [Step 4: Using the plugin in a Jenkins job]

Dependencies

The Jenkins CI Plugin requires the AWS Mobile SDK 1.10.5 or later. For more information and to install the SDK, see AWS Mobile SDK.

Step 1: Installing the Jenkins CI plugin for AWS Device Farm

There are two options for installing the Jenkins continuous integration (CI) plugin for AWS Device Farm. You can search for the plugin from within the **Available Plugins** dialog in the Jenkins Web UI, or you can download the `hpi` file and install it from within Jenkins.

Install from within the Jenkins UI

1. Find the plugin within the Jenkins UI by choosing **Manage Jenkins**, **Manage Plugins**, and then choose **Available**.

2. Search for **aws-device-farm**.
3. Install the AWS Device Farm plugin.
4. Ensure that the plugin is owned by the `Jenkins` user.
5. Restart Jenkins.

Download the plugin

1. Download the `hpi` file directly from http://updates.jenkins-ci.org/latest/aws-device-farm.hpi.
2. Ensure that the plugin is owned by the `Jenkins` user.
3. Install the plugin using one of the following options:
   - Upload the plugin by choosing **Manage Jenkins**, **Manage Plugins**, **Advanced**, and then choose **Upload plugin**.
   - Place the `hpi` file in the Jenkins plugin directory (usually `/var/lib/jenkins/plugins`).
4. Restart Jenkins.

Step 2: Creating an AWS Identity and Access Management user for your Jenkins CI Plugin for AWS Device Farm

We recommend that you do not use your AWS root account to access Device Farm. Instead, create a new AWS Identity and Access Management (IAM) user (or use an existing IAM user) in your AWS account, and then access Device Farm with that IAM user.

To create a new IAM user, see [Creating an IAM User (AWS Management Console)](). Be sure to generate an access key for each user and download or save the user security credentials. You will need the credentials later.

Give the IAM user permission to access Device Farm

To give the IAM user permission to access Device Farm, create a new access policy in IAM, and then assign the access policy to the IAM user as follows.

> **Note**
>
> The AWS root account or IAM user that you use to complete the following steps must have permission to create the following IAM policy and attach it to the IAM user. For more information, see [Working with Policies](#)

To create the access policy in IAM

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Policies**.
3. Choose **Create Policy**. (If a **Get Started** button appears, choose it, and then choose **Create Policy**.)
4. Next to **Create Your Own Policy**, choose **Select**.
5. For **Policy Name**, type a name for the policy (for example, **`AWSDeviceFarmAccessPolicy`**).
6. For **Description**, type a description that helps you associate this IAM user with your Jenkins project.
7. For **Policy Document**, type the following statement:

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

8. Choose **Create Policy**.

To assign the access policy to the IAM user

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Users**.

3. Choose the IAM user to whom you will assign the access policy.
4. In the **Permissions** area, for **Managed Policies**, choose **Attach Policy**.
5. Select the policy you just created (for example, **AWSDeviceFarmAccessPolicy**).
6. Choose **Attach Policy**.

Step 3: Configuring the Jenkins CI plugin for the first time in AWS Device Farm

The first time you run your Jenkins server, you will need to configure the system as follows.

> **Note**
>
> If you are using device slots, the device slots feature is disabled by default.

1. Log into your Jenkins Web user interface.
2. On the left-hand side of the screen, choose **Manage Jenkins**.
3. Choose **Configure System**.
4. Scroll down to the **AWS Device Farm** header.
5. Copy your security credentials from Creating an IAM user for your Jenkins CI Plugin and paste your Access Key ID and Secret Access Key into their respective boxes.
6. Choose **Save**.

Step 4: Using the plugin in a Jenkins job

Once you have installed the Jenkins plugin, follow these instructions to use the plugin in a Jenkins job.

1. Log into your Jenkins web UI.
2. Click the job you want to edit.
3. On the left-hand side of the screen, choose **Configure**.
4. Scroll down to the **Post-build Actions** header.
5. Click **Add post-build action** and select **Run Tests on AWS Device Farm**.
6. Select the project you would like to use.

7. Select the device pool you would like to use.
8. Select whether you'd like to have the test artifacts (such as the logs and screenshots) archived locally.
9. In **Application**, fill in the path to your compiled application.
10. Select the test you would like run and fill in all the required fields.
11. Choose **Save**.

Integrating Device Farm with a Gradle build system

The Device Farm Gradle plugin provides AWS Device Farm integration with the Gradle build system in Android Studio. For more information, see Gradle.

> **Note**
>
> To download the Gradle plugin, go to GitHub and follow the instructions in Building the Device Farm Gradle plugin.

The Device Farm Gradle Plugin provides Device Farm functionality from your Android Studio environment. You can kick off tests on real Android phones and tablets hosted by Device Farm.

This section contains a series of procedures to set up and use the Device Farm Gradle Plugin.

Topics

Dependencies

Runtime

- The Device Farm Gradle Plugin requires the AWS Mobile SDK 1.10.15 or later. For more information and to install the SDK, see AWS Mobile SDK.

- Android tools builder test api 0.5.2
- Apache Commons Lang3 3.3.4

For Unit Tests

- Testng 6.8.8
- Jmockit 1.19
- Android gradle tools 1.3.0

Step 1: Building the AWS Device Farm Gradle plugin

This plugin provides AWS Device Farm integration with the Gradle build system in Android Studio. For more information, see Gradle.

> **Note**
>
> Building the plugin is optional. The plugin is published through Maven Central. If you wish to allow Gradle to download the plugin directly, skip this step and jump to Step 2: Setting up the AWS Device Farm Gradle plugin.

To build the plugin

1. Go to GitHub and clone the repository.
2. Build the plugin using `gradle install`.

   The plugin is installed to your local maven repository.

Next step: Step 2: Setting up the AWS Device Farm Gradle plugin

Step 2: Setting up the AWS Device Farm Gradle plugin

If you haven't done so already, clone the repository and install the plugin using the procedure here: Building the Device Farm Gradle plugin.

To configure the AWS Device Farm Gradle Plugin

1. Add the plugin artifact to your dependency list in `build.gradle`.

```
    buildscript {

        repositories {
            mavenLocal()
            mavenCentral()
        }

        dependencies {
            classpath 'com.android.tools.build:gradle:1.3.0'
            classpath 'com.amazonaws:aws-devicefarm-gradle-plugin:1.0'
        }
    }
```

2. Configure the plugin in your `build.gradle` file. The following test specific configuration should serve as your guide:

```
apply plugin: 'devicefarm'

devicefarm {

    // Required. The project must already exist. You can create a project in the
 AWS Device Farm console.
    projectName "My Project" // required: Must already exist.

    // Optional. Defaults to "Top Devices"
    // devicePool "My Device Pool Name"

    // Optional. Default is 150 minutes
    // executionTimeoutMinutes 150

    // Optional. Set to "off" if you want to disable device video recording during
 a run. Default is "on"
    // videoRecording "on"

    // Optional. Set to "off" if you want to disable device performance monitoring
 during a run. Default is "on"
    // performanceMonitoring "on"

    // Optional. Add this if you have a subscription and want to use your unmetered
 slots
    // useUnmeteredDevices()
```

```
    // Required. You must specify either accessKey and secretKey OR roleArn.
roleArn takes precedence.
    authentication {
        accessKey "AKIAIOSFODNN7EXAMPLE"
        secretKey "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY"

        // OR

        roleArn "arn:aws:iam::111122223333:role/DeviceFarmRole"
    }

    // Optionally, you can
    // - enable or disable Wi-Fi, Bluetooth, GPS, NFC radios
    // - set the GPS coordinates
    // - specify files and applications that must be on the device when your test
runs
    devicestate {
        // Extra files to include on the device.
        // extraDataZipFile file("path/to/zip")

        // Other applications that must be installed in addition to yours.
        // auxiliaryApps files(file("path/to/app"), file("path/to/app2"))

        // By default, Wi-Fi, Bluetooth, GPS, and NFC are turned on.
        // wifi "off"
        // bluetooth "off"
        // gps "off"
        // nfc "off"

        // You can specify GPS location. By default, this location is 47.6204,
-122.3491
        // latitude 44.97005
        // longitude -93.28872
    }

    // By default, the Instrumentation test is used.
    // If you want to use a different test type, configure it here.
    // You can set only one test type (for example, Calabash, Fuzz, and so on)

    // Fuzz
    // fuzz { }

    // Calabash
    // calabash { tests file("path-to-features.zip") }
```

```
}
```

3. Run your Device Farm test using the following task: `gradle devicefarmUpload`.

   The build output will print out a link to the Device Farm console where you can monitor your test execution.

Next step: [Generating an IAM user in the Device Farm Gradle plugin]

Step 3: Generating an IAM user in the Device Farm Gradle plugin

AWS Identity and Access Management (IAM) helps you manage permissions and policies for working with AWS resources. This topic walks you through generating an IAM user with permissions to access AWS Device Farm resources.

If you haven't done so already, complete steps 1 and 2 before generating an IAM user.

We recommend that you do not use your AWS root account to access Device Farm. Instead, create a new IAM user (or use an existing IAM user) in your AWS account, and then access Device Farm with that IAM user.

> **Note**
>
> The AWS root account or IAM user that you use to complete the following steps must have permission to create the following IAM policy and attach it to the IAM user. For more information, see [Working with Policies].

To create a new user with the proper access policy in IAM

1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose **Users**.
3. Choose **Create New Users**.
4. Enter the user name of your choice.

   For example, **`GradleUser`**.

5. Choose **Create**.
6. Choose **Download Credentials** and save them in a location where you can easily retrieve them later.
7. Choose **Close**.
8. Choose the user name in the list.
9. Under **Permissions**, expand the **Inline Policies** header by clicking the down arrow on the right.
10. Choose **Click here** where it says, **There are no inline policies to show. To create one, click here**.
11. On the **Set Permissions** screen, choose **Custom Policy**.
12. Choose **Select**.
13. Give your policy a name, such as **`AWSDeviceFarmGradlePolicy`**.
14. Paste the following policy into **Policy Document**.

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

15. Choose **Apply Policy**.

Next step: Configuring test types.

For more information, see Creating an IAM User (AWS Management Console) or Setting up.

Step 4: Configuring test types

By default, the AWS Device Farm Gradle plugin runs the Instrumentation for Android and AWS Device Farm test. If you want to run your own tests or specify additional parameters, you can choose to configure a test type. This topic provides information about each available test type and what you need to do in Android Studio to configure it for use. For more information about the available test types in Device Farm, see Test frameworks and built-in tests in AWS Device Farm.

If you haven't done so already, complete steps 1 – 3 before configuring test types.

Note

If you are using device slots, the device slots feature is disabled by default.

Appium

Device Farm provides support for Appium Java JUnit and TestNG for Android.

- Appium (under Java (JUnit))
- Appium (under Java (TestNG))

You can choose `useTestNG()` or `useJUnit()`. `JUnit` is the default and does not need to be explicitly specified.

```
appium {
    tests file("path to zip file") // required
    useTestNG() // or useJUnit()
}
```

Built-in: fuzz

Device Farm provides a built-in fuzz test type, which randomly sends user interface events to devices and then reports the results.

```
fuzz {

   eventThrottle 50 // optional default
   eventCount 6000  // optional default
   randomizerSeed 1234 // optional default blank

 }
```

For more information, see Running Device Farm's built-in fuzz test (Android and iOS).

Instrumentation

Device Farm provides support for instrumentation (JUnit, Espresso, Robotium, or any instrumentation-based tests) for Android. For more information, see Instrumentation for Android and AWS Device Farm.

When running an instrumentation test in Gradle, Device Farm uses the `.apk` file generated from your **androidTest** directory as the source of your tests.

```
instrumentation {

    filter "test filter per developer docs" // optional

}
```

AWS Device Farm document history

The following table describes the important changes to the documentation since the last release of this guide.

Change	Description	Date Changed
AL2 support	Device Farm now supports the AL2 test environment for Android. Learn more about AL2.	November 6, 2023
Migration from Standard to Custom test environments	Updated migration guide to document deprecation for standard mode tests in December 2023.	September 3, 2023
VPC ENI support	Device Farm now allows private devices to use the VPC-ENI connectivity feature to help customers securely connect to their private endpoints hosted on AWS, on-premise software, or another cloud provider. Learn nore about VPC-ENI.	May 15, 2023
Polaris UI updates	The Device Farm console now supports the Polaris framework.	July 28, 2021
Python 3 support	Device Farm now supports Python 3 in custom mode tests. Learn more about using Python 3 in your test packages: • Appium (Python) • Appium (Python)	April 20, 2020
New security information and information on tagging AWS resources.	To make securing AWS services easier and more comprehensive, a new section on security has been built. To read more, see Security in AWS Device Farm A new section on tagging in Device Farm has been added. To learn more about tagging, see Tagging in Device Farm.	March 27, 2020

Change	Description	Date Changed
Removal of Direct Device Access.	Direct Device Access (remote debugging on private devices) is no longer available for general usage. For inquiries into future availability of Direct Device Access, please contact us.	September 9, 2019
Update Gradle plugin configura tion	A revised Gradle plugin configuration now includes a customizable version of the gradle configuration, with optional parameters commented out. Learn more about Setting up the Device Farm Gradle plugin.	August 16, 2019
New requirement for test runs with XCTest	For test runs that use the XCTest framework, Device Farm now requires an app package that is built for testing. Learn more about the section called "XCTest".	February 4, 2019
Support for Appium Node.js and Appium Ruby test types in custom environme nts	You can now run your tests in both Appium Node.js and Appium Ruby custom test environments. Learn more about Test frameworks and built-in tests in AWS Device Farm.	January 10, 2019
Support for Appium server version 1.7.2 in both standard and custom environme nts. Support for version 1.8.1 using a custom test spec YAML file in a custom test environment.	You can now run your tests in both standard and custom test environments with Appium server versions 1.72, 1.71, and 1.6.5. You can also run your tests with versions 1.8.1 and 1.8.0 using a custom test spec YAML file in a custom test environment. Learn more about Test frameworks and built-in tests in AWS Device Farm.	October 2, 2018

Change	Description	Date Changed
Custom test environments	With a custom test environment, you can ensure your tests run like they do in your local environment. Device Farm now provides support for live log and video streaming, so you can get instant feedback on your tests that are run in a custom test environment. Learn more about Custom test environments in AWS Device Farm.	August, 16 2018
Support for using Device Farm as an AWS CodePipeline test provider	You can now configure a pipeline in AWS CodePipeline to use AWS Device Farm runs as test actions in your release process. CodePipeline enables you to quickly link your repository to build and test stages to achieve a continuous integration system customized to your needs. Learn more about Integrating AWS Device Farm in a CodePipeline test stage.	July, 19 2018
Support for Private Devices	You can now use private devices to schedule test runs and start remote access sessions. You can manage profiles and settings for these devices, create Amazon VPC endpoints to test private apps, and create remote debugging sessions. Learn more about Private devices in AWS Device Farm.	May 2, 2018
Support for Appium 1.6.3	You can now set the Appium version for your Appium custom tests.	March 21, 2017
Set the execution timeout for test runs	You can set the execution timeout for a test run or for all tests in a project. Learn more about Setting the execution timeout for test runs in AWS Device Farm.	February 9, 2017
Network Shaping	You can now simulate network connections and conditions for a test run. Learn more about Simulating network connections and conditions for your AWS Device Farm runs.	December 8, 2016

Change	Description	Date Changed
New Troublesh ooting Section	You can now troubleshoot test package uploads using a set of procedures designed to resolve error messages you might encounter in the Device Farm console. Learn more about Troubleshooting Device Farm errors.	August 10, 2016
Remote Access Sessions	You can now remotely access and interact with a single device in the console. Learn more about Remote access.	April 19, 2016
Device Slots Self-Service	You can now purchase device slots using the AWS Management Console, the AWS Command Line Interface, or the API. Learn more about how to Purchasing a device slot in Device Farm.	March 22, 2016
How to stop test runs	You can now stop test runs using the AWS Managemen t Console, the AWS Command Line Interface, or the API. Learn more about how to Stopping a run in AWS Device Farm.	March 22, 2016
New XCTest UI test types	You can now run XCTest UI custom tests on iOS applicati ons. Learn more about the Integrating XCTest UI for iOS with Device Farm test type.	March 8, 2016
New Appium Python test types	You can now run Appium Python custom tests on Android, iOS, and web applications. Learn more about Test frameworks and built-in tests in AWS Device Farm.	January 19, 2016
Web Application test types	You can now run Appium Java JUnit and TestNG custom tests on web applications. Learn more about Web app tests in AWS Device Farm.	November 19, 2015
AWS Device Farm Gradle Plugin	Learn more about how to install and use the Device Farm Gradle plugin.	September 28, 2015
New Android Built-in Test: Explorer	The explorer test crawls your app by analyzing each screen as if it were an end user and takes screenshots as it explores.	September 16, 2015

Change	Description	Date Changed
iOS support added	Learn more about testing iOS devices and running iOS tests (including XCTest) in Test frameworks and built-in tests in AWS Device Farm.	August 4, 2015
Initial public release	This is the initial public release of the *AWS Device Farm Developer Guide*.	July 13, 2015

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.