AWS Device Farm: Developer Guide
Copyright © Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

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

What Is AWS Device Farm? .............................................................................................................. 1
Automated App Testing .................................................................................................................. 1
Remote Access Interaction .............................................................................................................. 1
Terminology .................................................................................................................................... 1
Setting Up ...................................................................................................................................... 2

Setting Up ...................................................................................................................................... 3
Step 1: Sign Up for AWS .................................................................................................................. 3
Step 2: Create or Use an IAM User in Your AWS Account .............................................................. 3
Step 3: Give the IAM User Permission to Access Device Farm .................................................... 3
Next Step ........................................................................................................................................ 4

Getting started ................................................................................................................................. 5
Prerequisites .................................................................................................................................... 5
Step 1: Sign in to the console .......................................................................................................... 5
Step 2: Create a project .................................................................................................................... 5
Step 3: Create and start a run ......................................................................................................... 6
Step 4: View the run's results ....................................................................................................... 7
Next steps ........................................................................................................................................ 7

Purchase device slots ...................................................................................................................... 8
Purchase device slots (console) ...................................................................................................... 8
Purchase a device slot (AWS CLI) .................................................................................................. 9
Purchase a device slot (API) .......................................................................................................... 12

Concepts ......................................................................................................................................... 13
Devices .............................................................................................................................................. 13
Supported Devices ........................................................................................................................... 13
Device Pools .................................................................................................................................... 13
Private Devices ............................................................................................................................... 13
Device Branding ............................................................................................................................. 14
Device Slots .................................................................................................................................... 14
Preinstalled Device Apps ................................................................................................................ 14
Device Capabilities ........................................................................................................................ 14

Test Environments .......................................................................................................................... 14
Standard Test Environment ............................................................................................................ 15
Custom Test Environment .............................................................................................................. 15

Runs .................................................................................................................................................. 15
Run Configuration ............................................................................................................................ 15
Run Files Retention .......................................................................................................................... 16
Run Device State .............................................................................................................................. 16
Parallel Runs .................................................................................................................................... 16
Setting the Execution Timeout ....................................................................................................... 16
Instrumenting Apps ......................................................................................................................... 16
Re-Signing Apps in Runs .................................................................................................................. 16
Obfuscated Apps in Runs ................................................................................................................. 16
Ads in Runs ...................................................................................................................................... 16
Media in Runs ................................................................................................................................. 17
Common Tasks for Runs .................................................................................................................. 17

Reports ............................................................................................................................................. 17
Report Retention .............................................................................................................................. 17
Report Components ........................................................................................................................ 17
Performance Samples in Reports ..................................................................................................... 17
Logs in Reports ............................................................................................................................... 17
Common Tasks for Reports .............................................................................................................. 17

Sessions ........................................................................................................................................... 18
Supported Devices for Remote Access ........................................................................................... 18
Session Files Retention .................................................................................................................... 18

API Version 2015-06-23
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appium</td>
<td>53</td>
</tr>
<tr>
<td>Version Support</td>
<td>53</td>
</tr>
<tr>
<td>Configure Your Appium Test Package</td>
<td>54</td>
</tr>
<tr>
<td>Create a Zipped Package File</td>
<td>60</td>
</tr>
<tr>
<td>Upload Your Test Package to Device Farm</td>
<td>62</td>
</tr>
<tr>
<td>Take Screenshots of Your Tests (Optional)</td>
<td>64</td>
</tr>
<tr>
<td>Working with Remote Access</td>
<td>85</td>
</tr>
<tr>
<td>Working with Custom Test Environments</td>
<td>76</td>
</tr>
<tr>
<td>Get Session Results</td>
<td>87</td>
</tr>
<tr>
<td>Use a Session</td>
<td>86</td>
</tr>
<tr>
<td>Create a Session</td>
<td>85</td>
</tr>
<tr>
<td>Extending Custom Mode</td>
<td>82</td>
</tr>
<tr>
<td>Migrating Tests</td>
<td>81</td>
</tr>
<tr>
<td>Android Tests</td>
<td>65</td>
</tr>
<tr>
<td>Built-in Tests</td>
<td>65</td>
</tr>
<tr>
<td>Built-in Test Types for Android</td>
<td>65</td>
</tr>
<tr>
<td>Instrumentation</td>
<td>66</td>
</tr>
<tr>
<td>UI Automator</td>
<td>67</td>
</tr>
<tr>
<td>iOS Tests</td>
<td>68</td>
</tr>
<tr>
<td>Built-in Test Types for iOS</td>
<td>68</td>
</tr>
<tr>
<td>UI Automation</td>
<td>69</td>
</tr>
<tr>
<td>XCTest</td>
<td>70</td>
</tr>
<tr>
<td>XCTest UI</td>
<td>72</td>
</tr>
<tr>
<td>Web App Tests</td>
<td>73</td>
</tr>
<tr>
<td>Rules for Metered and Unmetered Devices</td>
<td>73</td>
</tr>
<tr>
<td>Built-in Tests</td>
<td>73</td>
</tr>
<tr>
<td>Built-in Test Types</td>
<td>73</td>
</tr>
<tr>
<td>Built-in: Explorer (Android)</td>
<td>73</td>
</tr>
<tr>
<td>Built-in: Fuzz (Android and iOS)</td>
<td>74</td>
</tr>
<tr>
<td>Working with Custom Test Environments</td>
<td>76</td>
</tr>
<tr>
<td>Test Spec Syntax</td>
<td>76</td>
</tr>
<tr>
<td>Test Spec Example</td>
<td>77</td>
</tr>
<tr>
<td>Environment Variables</td>
<td>79</td>
</tr>
<tr>
<td>Common Environment Variables</td>
<td>79</td>
</tr>
<tr>
<td>Appium Java JUnit Environment Variables</td>
<td>80</td>
</tr>
<tr>
<td>Appium Java TestNG Environment Variables</td>
<td>81</td>
</tr>
<tr>
<td>XCUItestEnvironment Variables</td>
<td>81</td>
</tr>
<tr>
<td>Migrating Tests</td>
<td>81</td>
</tr>
<tr>
<td>Appium Framework</td>
<td>81</td>
</tr>
<tr>
<td>Android Instrumentation</td>
<td>81</td>
</tr>
<tr>
<td>Migrating Existing iOS XCUItest Tests</td>
<td>81</td>
</tr>
<tr>
<td>Extending Custom Mode</td>
<td>82</td>
</tr>
<tr>
<td>Setting a PIN</td>
<td>82</td>
</tr>
<tr>
<td>Speeding Up Appium-based Tests through Server Capabilities</td>
<td>82</td>
</tr>
<tr>
<td>Using Webhooks and other APIs after Your Tests Run</td>
<td>83</td>
</tr>
<tr>
<td>Working with Remote Access</td>
<td>85</td>
</tr>
<tr>
<td>Create a Session</td>
<td>85</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>85</td>
</tr>
<tr>
<td>Create a Session with the Device Farm Console</td>
<td>85</td>
</tr>
<tr>
<td>Next Steps</td>
<td>86</td>
</tr>
<tr>
<td>Use a Session</td>
<td>86</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>86</td>
</tr>
<tr>
<td>Use a Session in the Device Farm Console</td>
<td>86</td>
</tr>
<tr>
<td>Next Steps</td>
<td>87</td>
</tr>
<tr>
<td>Tips and Tricks</td>
<td>87</td>
</tr>
<tr>
<td>Get Session Results</td>
<td>87</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>87</td>
</tr>
</tbody>
</table>
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 Working with Test Types in AWS Device Farm (p. 52).

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 (p. 17).

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 (p. 15).

session

A real-time interaction with an actual, physical device through your web browser. For more information, see Sessions (p. 18).

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 (p. 13).

Setting Up

To use Device Farm, see Setting Up (p. 3).
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 (p. 3)
- Step 2: Create or Use an IAM User in Your AWS Account (p. 3)
- Step 3: Give the IAM User Permission to Access Device Farm (p. 3)
- Next Step (p. 4)

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
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.

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.
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 (p. 187).

Next Step

You are now ready to start using Device Farm. See Getting started with Device Farm (p. 5).
Prerequisites

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

• Complete the steps in Setting Up (p. 3). 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 Working with Test Types in AWS Device Farm (p. 52).

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.
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 (p. 15).

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 Types and Frameworks (p. 52).
   - 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)” (p. 74).
   - 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.

   **Note**
   Setting the device radio state is available only for Android native tests at this time.
   - 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.
10. Choose Next.

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 Working with test reports in Device Farm (p. 40).

Next steps

For more information about Device Farm, see Concepts (p. 13).
Purchase 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.

Topics
- Purchase device slots (console) (p. 8)
- Purchase a device slot (AWS CLI) (p. 9)
- Purchase a device slot (API) (p. 12)

Purchase device slots (console)

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 appears. 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:

```json
{
  "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:

```json
{
  "offerings": [
    {
      "recurringCharges": [
        {
          "cost": {
          }
        }
      ]
    }
  ]
}
```
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 (p. 113).
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 (p. 115).
**AWS Device Farm Concepts**

This section describes important Device Farm concepts.

- **Devices** (p. 13)
- **Test Environments** (p. 14)
- **Runs** (p. 15)
- **Reports** (p. 17)
- **Sessions** (p. 18)

For more information about supported test types in Device Farm, see *Working with Test Types in AWS Device Farm* (p. 52).

**Device Support in AWS Device Farm**

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

**Topics**

- **Supported Devices** (p. 13)
- **Device Pools** (p. 13)
- **Private Devices** (p. 13)
- **Device Branding** (p. 14)
- **Device Slots** (p. 14)
- **Preinstalled Device Apps** (p. 14)
- **Device Capabilities** (p. 14)

**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. 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 Working with Private Devices in AWS Device Farm (p. 89).

Device Branding

Device Farm runs tests on physical, non-rooted devices that are both OEM- and carrier-branded.

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 (p. 15)
- Custom Test Environment (p. 15)
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 Working with Custom Test Environments (p. 76).

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 Create a test run in Device Farm (p. 22).

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 (p. 15)
- Run Files Retention (p. 16)
- Run Device State (p. 16)
- Parallel Runs (p. 16)
- Setting the Execution Timeout (p. 16)
- Instrumenting Apps (p. 16)
- Re-Signing Apps in Runs (p. 16)
- Obfuscated Apps in Runs (p. 16)
- Ads in Runs (p. 16)
- Media in Runs (p. 17)
- Common Tasks for Runs (p. 17)

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 Set the Execution Timeout for Test Runs in AWS Device Farm (p. 32).

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.

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 Create a test run in Device Farm (p. 22) and Working with Test Runs (p. 22).

Reports in AWS Device Farm

The following sections provide information about Device Farm test reports.

Topics
- Report Retention (p. 17)
- Report Components (p. 17)
- Performance Samples in Reports (p. 17)
- Logs in Reports (p. 17)
- Common Tasks for Reports (p. 17)

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.

Performance Samples in Reports

During a test run, Device Farm captures performance samples every second.

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 Working with test reports in Device Farm (p. 40).
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 (p. 18)
• Session Files Retention (p. 18)
• Instrumenting Apps (p. 18)
• Re-Signing Apps in Sessions (p. 18)
• Obfuscated Apps in Sessions (p. 18)

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 Devices (p. 13).

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.
Working with 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
- Create a Project in AWS Device Farm (p. 19)
- View the Projects List in AWS Device Farm (p. 20)

Create 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 (p. 3).

Create a Project (Console)

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 Set the Execution Timeout for Test Runs in AWS Device Farm (p. 32).

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.
Create a Project (API)

- Call the CreateProject API.

For information about using the Device Farm API, see Automating Device Farm (p. 115).

View 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 (p. 20)
- View the Projects List (Console) (p. 20)
- View the Projects List (AWS CLI) (p. 20)
- View the Projects List (API) (p. 21)

Prerequisites

- Create at least one project in Device Farm. Follow the instructions in Create a Project in AWS Device Farm (p. 19), and then return to this page.

View the Projects List (Console)

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 (p. 113).

**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 (p. 115).
Working with 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 (p. 15).

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

Topics
- create a test run in Device Farm (p. 22)
- Set the Execution Timeout for Test Runs in AWS Device Farm (p. 32)
- Simulate Network Connections and Conditions for your AWS Device Farm Runs (p. 33)
- Stop a Run in AWS Device Farm (p. 35)
- View a List of Runs in AWS Device Farm (p. 37)
- Create a Device Pool in AWS Device Farm (p. 38)
- Analyzing Results in AWS Device Farm (p. 40)

Create 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 (p. 206). For information about runs, see Runs (p. 15).

Topics
- Prerequisites (p. 22)
- Create a test run (console) (p. 22)
- Create a test run (AWS CLI) (p. 24)
- Create a test run (API) (p. 31)
- Next steps (p. 32)

Prerequisites

You must have a project in Device Farm. Follow the instructions in Create a Project in AWS Device Farm (p. 19), and then return to this page.

Create a test run (console)

2. In the navigation pane, choose Mobile Device Testing, and then choose Projects.
Create a test run (console)

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 **Working with Test Types in AWS Device Farm (p. 52)**.

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 (p. 14)**.

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 Create a Device Pool (p. 38), 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 **Devices** (p. 13).

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.

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 Set the Execution Timeout for Test Runs in AWS Device Farm (p. 32).

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 Stop a Run in AWS Device Farm (p. 35).

## Create a test run (AWS CLI)

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

**Topics**

- Step 1: Choose a project (p. 25)
- Step 2: Choose a device pool (p. 25)
- Step 3: Upload your application file (p. 26)
- Step 4: Upload your test scripts package (p. 27)
- Step 5: (Optional) Upload your custom test spec (p. 28)
- Step 6: Schedule a test run (p. 30)
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 Create a Project in AWS Device Farm (p. 19).

   Example:
   ```
   aws devicefarm list-projects
   ```

   The response includes a list of your Device Farm projects.
   ```json
   {
   "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:
   ```json
   {
   "devicePools": [
   {
   "rules": [
   {
   "attribute": "ARN",
   "operator": "IN",
   }
   ,
   "type": "CURATED",
   "name": "Top Devices",
   "arn": "arn:aws:devicefarm:us-west-2::devicepool:example",
   "description": "Top devices"
   }
   ,
   ```
Create a test run (AWS CLI)

"rules": [
    {
        "attribute": "PLATFORM",
        "operator": "EQUALS",
        "value": "\\ANDROID\\"
    },
    {"type": "PRIVATE",
     "name": "MyAndroidDevices",
     "arn": "arn:aws:devicefarm:us-west-2:605403973111:devicepool:example2"
    }
],
"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) (p. 39).

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:

   ```bash
   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.

   ```json
   {
     "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:

   ```bash
   ```
For more information, see Uploading objects using presigned URLs in the Amazon Simple Storage Service Developer Guide.

4. To check the status of your app upload, run `get-upload` and specify the ARN of the app upload.

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

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

```json
{
   "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:

   ```bash
   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.

   ```json
   {
      "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:

   ```bash
   ```

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.
Create a test run (AWS CLI)

```bash
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](#).

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

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

   The response contains an entry for each default test spec:

   ```json
   {
      "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:

   ```bash
   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`:

   ```bash
   ```

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:

   ```bash
   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:

   ```json
   {
     "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:

   ```bash
   ```

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

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

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

   ```json
   {
     "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 (p. 25).
- The device pool ARN from step 2 (p. 25).
- The app upload ARN from step 3 (p. 26).
- The test package upload ARN from step 4 (p. 27).

If you are running tests in a custom test environment, you also need your test spec ARN from step 5 (p. 28).

**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:

```bash
```

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

```json
{
    "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,
```
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) (p. 24).

You need this information to call the ScheduleRun API:

- A project ARN. See Create a Project (API) (p. 20) and CreateProject.
- An application upload ARN. See CreateUpload.
- A test package upload ARN. See CreateUpload.
- A device pool ARN. See Create a Device Pool (p. 38) 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 (p. 28) and CreateUpload.

For information about using the Device Farm API, see Automating Device Farm (p. 115).
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 (p. 17).

To use the report, follow the instructions in Working with test reports in Device Farm (p. 40).

Set 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 (p. 3).
2. Create a project in Device Farm. Follow the instructions in Create a Project in AWS Device Farm (p. 19), and then return to this page.

**Set the Execution Timeout for a Project**

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

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.

### Simulate 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**
- Set Up Network Shaping When Scheduling a Test Run (p. 33)
- Create a Network Profile (p. 34)
- Change Network Conditions During Your Test (p. 35)

### 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 Create a Project in AWS Device Farm (p. 19).
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.
Create a Network Profile

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

1. Choose **Create network profile**.

   - **Name**: MyNetworkProfile
   - **Description** (optional): Please enter a short description.
   - **Uplink bandwidth (bps)**: Data throughput rate in bits per second as a number from 0 to 104857600.
     - Value: 104857600
   - **Downlink bandwidth (bps)**: Data throughput rate in bits per second as a number from 0 to 104857600.
     - Value: 104857600
   - **Uplink delay (ms)**: Delay time for all packets to destination in milliseconds as a number from 0 to 2000.
     - Value: 0
   - **Downlink delay (ms)**: Delay time for all packets to destination in milliseconds as a number from 0 to 2000.
     - Value: 0
   - **Uplink jitter (ms)**: Time variation in the delay of received packets in milliseconds as a number from 0 to 2000.
     - Value: 0
   - **Downlink jitter (ms)**: Time variation in the delay of received packets in milliseconds as a number from 0 to 2000.
     - Value: 0
   - **Uplink loss (%)**: Proportion of transmitted packets that fail to arrive from 0 to 100 percent.
     - Value: 0
   - **Downlink loss (%)**: Proportion of received packets that fail to arrive from 0 to 100 percent.
     - Value: 0

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 or Calabash to simulate dynamic network conditions such as reduced bandwidth during your test run. For more information, see CreateNetworkProfile.

Stop 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) (p. 35)
- Stop a Run (AWS CLI) (p. 36)
- Stop a Run (API) (p. 37)

Stop a Run (Console)

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.

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

![Example Devices section](image.png)

**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:

```json
{
  "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 (p. 113).

### Stop a Run (API)

- Call the `StopRun` operation to the test run.

For information about using the Device Farm API, see Automating Device Farm (p. 115).

### View 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) (p. 37)
- View a List of Runs (AWS CLI) (p. 38)
- View a List of Runs (API) (p. 38)

**View a List of Runs (Console)**

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 (p. 113).

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 (p. 115).

Create 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 (p. 38)
- Create a Device Pool (Console) (p. 38)
- Create a Device Pool (AWS CLI) (p. 39)
- Create a Device Pool (API) (p. 39)

Prerequisites

- Create a run in the Device Farm console. Follow the instructions in Create a test run in Device Farm (p. 22). When you get to the Select devices page, continue with the instructions in this section.

Create a Device Pool (Console)

1. On the Select devices page, 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 type value, choose **Form Factor**.

d. For **Operator** (second drop-down list), choose **EQUALS** to include devices where the **Field** value equals the **Value** value.

e. For **Value** (third drop-down list), enter or choose the value you want to specify for the **Field** and **Operator** values. 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**.

g. To delete a rule, choose the X icon next to the 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.

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.

**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)

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

  **Note**
  For Appium tests, you can use `APPIUM_VERSION` in the `rules` field.

For information about using Device Farm with the AWS CLI, see [AWS CLI Reference](p. 113).

### Create a Device Pool (API)

- Call the `CreateDevicePool` API.

  **Note**
  For Appium tests, you can use `APPIUM_VERSION` in the `rules` field.

For information about using the Device Farm API, see [Automating Device Farm](p. 115).
Analyzing 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.

Device Farm also gathers other artifacts such as files, logs, and images that you can download when your test run is complete.

Topics

- Working with test reports in Device Farm (p. 40)
- Working With Artifacts in Device Farm (p. 46)

Working with test reports in Device Farm

Use the Device Farm console to view your test reports. For more information, see Reports (p. 17).

Topics

- Prerequisites (p. 40)
- Understanding test results (p. 40)
- Viewing reports (p. 41)

Prerequisites

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

1. To create a run, see Create a test run in Device Farm (p. 22), 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 Understanding test results (p. 40).

Understanding test results

The Device Farm console displays icons that help you quickly assess the state of your completed test run.

Topics

- Reporting results for an individual test (p. 40)
- Reporting results for multiple tests (p. 41)

Reporting results for an individual test

For reports that describe an individual test, Device Farm displays an icon:

<table>
<thead>
<tr>
<th>Description</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>The test succeeded.</td>
<td>✔</td>
</tr>
</tbody>
</table>
### Reporting results for multiple tests

If you choose a finished run, Device Farm displays a test results summary graph.

![Test results summary graph](image)

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.

### Viewing reports

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

**Topics**

- View the test run summary page (p. 42)
- View unique problem reports (p. 42)
- View device reports (p. 43)
- View test suite reports (p. 43)
- View test reports (p. 44)
- View performance data for a problem, device, suite, or test in a report (p. 44)
- View log information for a problem, device, suite, or test in a report (p. 44)
View the test run summary page

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 (p. 42).
   - 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 (p. 43).

   - 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 Reporting results for an individual test (p. 40).

   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 (p. 44).

   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 (p. 44).

   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 Reporting results for multiple tests (p. 41).

To view the full results by suite, follow the instructions in View test suite reports (p. 43).

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 (p. 44).

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 (p. 44).

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 Reporting results for multiple tests (p. 41).

To view the full results by test, follow the instructions in View test reports (p. 44).

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 (p. 44).

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 (p. 44).

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 Reporting results for an individual test (p. 40).

   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 (p. 44).

   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 (p. 44).

   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 Android devices at this time.

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**.
Working With 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 Working with test reports in Device Farm (p. 40).

**Logs**

Output from each test in the test run.

**Screenshots**

Screen images recorded for each test in the test run.

Using 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 Using Artifacts in a Custom Test Environment (p. 48).

Using Artifacts (AWS CLI)

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

**Topics**

- Step 1: Get Your Amazon Resource Names (ARN) (p. 46)
- Step 2: List Your Artifacts (p. 47)
- Step 3: Download Your Artifacts (p. 48)

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:
<table>
<thead>
<tr>
<th>AWS CLI List Command</th>
<th>Required ARN</th>
</tr>
</thead>
<tbody>
<tr>
<td>list-projects</td>
<td>This command returns all projects and does not require an ARN.</td>
</tr>
<tr>
<td>list-runs</td>
<td>project</td>
</tr>
<tr>
<td>list-jobs</td>
<td>run</td>
</tr>
<tr>
<td>list-suites</td>
<td>job</td>
</tr>
<tr>
<td>list-tests</td>
<td>suite</td>
</tr>
</tbody>
</table>

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

Example:

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

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

```json
{
  "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`

Using 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.

Using 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 Working With Artifacts in Device Farm (p. 46).
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 (p. 50)
- Looking Up Resources by Tags (p. 51)
- Removing Tags from Resources (p. 51)

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:

```python
import boto3
client = boto3.client('resourcegroupstaggingapi')

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 Tags

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/resourcegroupstaggingapi/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`:

```python
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:

```python
import boto3
client = boto3.client('resourcegroupstaggingapi')
```
Working with Test Types in AWS Device Farm

This section describes Device Farm support for testing frameworks, as well as for two built-in test types.

Testing Frameworks

Device Farm supports these mobile automation testing frameworks:

Android Application Testing Frameworks

- Working with Appium and AWS Device Farm (p. 53)
- Working with Calabash tests on AWS Device Farm (p. 64)
- Working with Instrumentation for Android and AWS Device Farm (p. 66)
- UI Automator (p. 67)

iOS Application Testing Frameworks

- Working with Appium and AWS Device Farm (p. 53)
- Working with Calabash tests on AWS Device Farm (p. 64)
- UI Automation (p. 69)
- Working with XCTest for iOS and AWS Device Farm (p. 70)
- XCTest UI (p. 72)

Web Application Testing Frameworks

Web applications are supported using Appium. For more information on bringing your tests to Appium, see Working with Appium and AWS Device Farm (p. 53).

Frameworks in a Custom Test Environment

Device Farm does not provide support for customizing the test environment for Calabash, XCTests, and UI Automator testing frameworks. For more information, see Working with Custom Test Environments (p. 76).

Appium Version Support

For tests running in a custom environment, Device Farm supports Appium v1.8.1. For tests running in the standard environment, Device Farm supports v1.7.2. For more information, see Test Environments (p. 14).
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 two built-in test types:

- Built-in: Explorer (Android) (p. 73)
- Built-in: Fuzz (Android and iOS) (p. 74)

Working with Appium 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.

Version Support

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

Device Farm supports all Appium server versions 1.6.5 and above. You can choose any Appium version by using the `avm` command. For example, to use Appium server version 1.9.1, add these commands to your test spec YAML file:

```yaml
phases:
  install:
    commands:
      - export APPIUM_VERSION=1.9.1
      - avm $APPIUM_VERSION
      - ln -s /usr/local/avm/versions/$APPIUM_VERSION/node_modules/.bin/appium /usr/local/avm/versions/$APPIUM_VERSION/node_modules/appium/bin/appium.js
```

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.

Java (TestNG)

Currently, Device Farm supports Java 8 for running Appium Java tests.

Java (JUnit)

Currently, Device Farm supports Java 8 for running Appium Java tests.

Node.js

You can use any Node.js version when you run Appium Node.js tests on Device Farm. Specify the Node.js version in the test spec YAML file.

Python

In standard mode, Device Farm supports Python version 2.7 for running tests. In custom mode, both Python 2.7 and Python 3 (3.7) are available.

Ruby

You can use any Ruby version when you run Appium Ruby tests on Device Farm. Specify the Ruby version (for example, 2.5.1) in the test spec YAML file.
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:

   ```xml
   <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:

   ```xml
   <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:

   ```xml
   <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>
   ```
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:

```xml
<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:

```xml
<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:

```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:

```xml
<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:

```xml
<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:

```xml
<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:

```xml
<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:

```xml
<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:

```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)**

  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

   ```bash
   command -v nvm
   ```

   You should see `nvm` as output.

   For more information, see [nvm on GitHub](https://github.com/nvm-sh/nvm).

2. Run this command to install Node.js:

   ```bash
   nvm install node
   ```

   You can specify a particular version of Node.js:

   ```bash
   nvm install 11.4.0
   ```

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

   ```bash
   node -v
   ```

4. Install `npm-bundle` globally:

   ```bash
   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.

   ```bash
   $ 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 want 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

   ```bash
   command -v rvm
   ```
   
   You should see `rvm` as output.

3. If you want to install a specific version of Ruby (for example, 2.5.1) run the following command:

   ```bash
   rvm install ruby 2.5.1 --autolibs=0
   ```
   
   Verify that you are on the requested version of Ruby:

   ```bash
   ruby -v
   ```

4. The `bundler` gem is usually installed by default. If it is not, install it:

   ```bash
   gem install bundler
   ```

---

**Create a Zipped Test Package File**

Now, bundle your tests for Device Farm.

**Java (JUnit)**

Build and package your tests:

```bash
$ 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:

```bash
$ 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.

```bash
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.

```bash
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:

```bash
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:

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

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

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

**Python 3**

Package your tests and pip requirements into a zip file:

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

**Ruby**

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

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

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

```bash
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`:

```bash
bundle install
```

**Note**
At this point, you should be able to run your tests locally. Use this command to run a test locally:

```bash
bundle exec $test_command
```

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

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

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

```bash
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.

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 (p. 54).
10. Choose to either **Run your test in our standard environment** or **Run your test in a custom environment**. The standard environment has granular, per-test reporting, while the custom environment is capable of running custom test harnesses built on top of any framework.

Using the custom environment allows for full control over test setup, teardown, and invocation, as well as choosing specific versions of runtimes and the Appium server. Some language features (such as Python 3 or custom TestNG execution) are only available through the custom environment:

Java (JUnit)

JUnit runs in both the standard and custom environments without issue

Java (TestNG)

Invoking TestNG using a testng.xml configuration is only available in Custom Mode.

Node.JS

The Custom Mode allows you to specify additional parameters or use nonstandard invocations of your test framework.

Python

Python 2.7 is supported in both the standard environment and using Custom Mode. It is the default in both when specifying Python.

Python 3 is only supported in Custom Mode. To choose Python 3 as your python version, change the test spec to set the PYTHON_VERSION to 3, as shown here:

```
phases:
  install:
    commands:
      # ...
      - export PYTHON_VERSION=3
      - export APPIUM_VERSION=1.14.2
      # Activate the Virtual Environment that Device Farm sets up for Python 3, then use Pip to install required packages.
      - cd $DEVICEFARM_TEST_PACKAGE_PATH
      - . bin/activate
      - pip install -r requirements.txt
      # ...
```

Ruby

In Custom Mode, you may specify a specific version of Ruby with the `rvm` command:

```
phases:
  install:
    commands:
      rvm install version --autolibs=0
      rvm use version
```

11. Choose **Next**, and then follow the instructions to select devices and start the run. For more information, see Create a test run in Device Farm (p. 22).

**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 `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.

Working with Calabash tests on AWS Device Farm

Device Farm supports Calabash for iOS and Android tests. For a sample test suite using Calabash to test an application using Calabash, see the Device Farm sample app.

What is Calabash?

Calabash is a mobile testing framework you can use to run automated interface acceptance tests. Tests are written in the Cucumber description language and Ruby.

Device farm supports Calabash version 0.20.5.

Preparing your Calabash tests

To bring your Calabash tests to Device Farm, they must be zipped with the following structure:

- A directory called `features`, containing
  - Feature definitions (`my-feature.feature`)
  - A directory called `step_definitions` containing ruby step definitions
  - A directory called `support` containing ruby support files
  - Any other supporting files

Upload your Calabash tests to AWS Device Farm

Use the Device Farm console to upload your tests.

2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Projects**.
3. Choose the project for the test run to be associated with.
   
   **Tip**
   
   To create a project, follow the instructions in Create a Project in AWS Device Farm (p. 19).
4. Choose **Create a new run**.
5. On the **Choose application** page, select **Choose File**.
6. Browse to and choose your app file. The file must be an .ipa file for iOS applications or an .apk file for Android applications.
   
   **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 Calabash, and then select Choose File.
9. Browse to and choose the .zip file that contains your tests.
10. Choose Next, and then complete the remaining instructions to select the devices to run your tests on and start the run.

Taking Screenshots in Android Calabash Tests

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

Calabash provides a set of predefined steps for taking screenshots. For details, see the "Screenshots" section of the iOS Predefined Steps and Android Predefined Steps page in the Calabash repositories on GitHub.

Alternatively, you can define a custom step inside of a Ruby (.rb) file to call the screenshot_embed function, which creates a screenshot and saves it to a directory you define. For example, the following code example creates a screenshot and saves it to the /my/custom/path directory with a file name of screenshot_seconds-since-Epoch:

```ruby
screenshot_embed(:prefix => "\my/custom/path", :name => "screenshot_#{Time.now.to_i}")
```

Additional Considerations for Calabash Tests

Device Farm replaces some Calabash hooks so that Calabash tests can run on devices in Device Farm, but does not modify tests.

Working with Android Tests in AWS Device Farm

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

Android Application Testing Frameworks

The following tests are available for Android devices.

- Working with Appium and AWS Device Farm (p. 53)
- Working with Instrumentation for Android and AWS Device Farm (p. 66)
- UI Automator (p. 67)

Built-in Test Types for Android

There are two built-in test types available for Android devices.

- Built-in: Explorer (Android) (p. 73)
- Built-in: Fuzz (Android and iOS) (p. 74)
Working with 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.

**Topics**
- What Is Instrumentation? (p. 66)
- Upload Your Android Instrumentation Tests (p. 66)
- Taking Screenshots in Android Instrumentation Tests (p. 66)
- Additional Considerations for Android Instrumentation Tests (p. 67)

**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 Instrumentation in the "Testing Fundamentals" section of the Android Developer Tools documentation.

**Upload Your Android Instrumentation Tests**

Use the Device Farm console to upload your tests.

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 Create a Project in AWS Device Farm (p. 19).

4. If the Create a new run button is displayed, choose it.
5. On the Choose application page, select Choose 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.

**Taking 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:

```java
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`

### Additional Considerations for Android Instrumentation Tests

#### 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.

### Working with UI Automator for Android and AWS Device Farm

Device Farm provides support for UI Automator for Android.

**Note**

This framework is currently in preview. It might not work with all scripts and apps.

#### Topics

- What Is UI Automator? (p. 67)
- Prepare Your Android UI Automator Tests (p. 68)
- Upload Your Android UI Automator Tests (p. 68)
- Taking Screenshots in Android UI Automator Tests (p. 68)
- Additional Considerations for Android UI Automator Tests (p. 68)

#### What Is UI Automator?

The UI Automator testing framework provides a set of APIs to build user interface tests that perform interactions on user and system apps for Android. The UI Automator APIs allow you to perform operations such as opening the `Settings` menu or the app launcher in a test device. For more information, see UI Automator in the "Testing Support Library" section of the Android Developer Tools documentation.
Prepare Your Android UI Automator Tests

The Android UI Automator tests must be contained in a single JAR file. The package name in this file must match the package name used by the Android app. For example, if the Android app’s package name is `com.my.android.app.MyMobileApp`, the Android UI Automator tests must be in a package named `com.my.android.app`.

Upload Your Android UI Automator Tests

Use the Device Farm console to upload your tests.

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 Create a Project in AWS Device Farm (p. 19).
4. If the Create a new run button is displayed, choose it.
5. On the Choose application page, select Choose File.
7. Choose Next.
8. On the Configure page, in the Setup test framework section, choose UI Automator, and then select Choose File.
9. Browse to and choose the JAR file that contains your tests.
10. Choose Next, and then complete the remaining instructions to select devices and start the run.

Taking Screenshots in Android UI Automator Tests

You can take screenshots as part of your Android UI Automator tests.

To take a screenshot, call the `takeScreenshot` method (for example, `takeScreenshot("/sdcard/uiautomator-screenshots/home-screen-1234.png");`).

**Note**

All screenshots must be stored in the `/sdcard/uiautomator-screenshots` directory. You must specify the full path (including the file name) of the screenshot to be stored.

The `takeScreenshot` method works for API Levels 17 and higher only. For API Level 16, UI Automator is supported, but screenshots are not supported.

Additional Considerations for Android UI Automator Tests

Device Farm re-signs Android UI Automator test packages, but it does not modify Android UI Automator tests.

Working with iOS Tests in AWS Device Farm

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

iOS Application Testing Frameworks

The following tests are available for iOS devices.
Built-in Test Types for iOS

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

- Built-in: Fuzz (Android and iOS) (p. 74)

Working with UI Automation for iOS and AWS Device Farm

Device Farm provides support for UI Automation for iOS.

Topics

- What Is UI Automation? (p. 69)
- Upload Your iOS UI Automation Tests (p. 69)
- Taking Screenshots in iOS UI Automation Tests (p. 70)
- Additional Considerations for iOS UI Automation Tests (p. 70)

What Is UI Automation?

You can use the Automation instrument to automate user interface tests in your iOS app through test scripts that you write. These scripts run outside of your app and simulate user interaction by calling the UI Automation API. The API is a JavaScript programming interface that specifies actions to be performed in your app as it runs in the simulator or on a connected device. For more information, see About Instruments in the Instruments User Guide of the iOS Developer Library.

Upload Your iOS UI Automation Tests

Use the Device Farm console to upload your tests.

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 Create a Project in AWS Device Farm (p. 19).
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 UI Automation, and then select Choose File.
9. Browse to and choose the .js file for a single test.
10. Choose Next, and then complete the remaining instructions to select the devices to run your tests on and start the run.

Taking Screenshots in iOS UI Automation Tests

You can take screenshots as part of your iOS UI Automation tests.

To take a screenshot, call the captureScreenWithName function (for example, target.capturseriesWithNamem(name + "_home"));, where name is the current language name).

Additional Considerations for iOS UI Automation Tests

Device Farm adds logging hooks so that it can monitor the execution flow of iOS UI Automation tests, but Device Farm does not modify iOS UI Automation tests themselves.

Working with XCTest for iOS and AWS Device Farm

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.

Topics
- Creating the Packages for Your XCTest Run (p. 70)
- Uploading the Packages for Your XCTest Run to Device Farm (p. 71)

Creating 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.

Uploading the Packages for Your XCTest Run to Device Farm

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

2. If you don't have a project already, create one. For the steps to create a project, see **Create a Project in AWS Device Farm** (p. 19).

   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.
Working with XCTest UI Testing Framework for iOS and AWS Device Farm

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

Topics
- What Is XCTest UI Testing Framework? (p. 72)
- Prepare Your iOS XCTest UI Tests (p. 72)
- Upload Your iOS XCTest UI Tests (p. 72)
- Taking Screenshots in iOS XCTest UI Tests (p. 73)

What Is XCTest UI Testing Framework?

XCTest UI framework is the new testing framework introduced with Xcode 7. This framework extends XCTest with UI testing capabilities. For more information, see User Interface Testing in the iOS Developer Library.

Prepare Your iOS XCTest UI Tests

Your iOS XCTest UI test runner bundle must be contained in a properly formatted .ipa file.

To create an .ipa file, place your my-project-nameUITest-Runner.app bundle in an empty Payload directory. Next, archive the Payload directory into a .zip file and then change the file extension to .ipa. The *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.

Upload Your iOS XCTest UI Tests

Use the Device Farm console to upload your tests.

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 Create a Project in AWS Device Farm (p. 19)

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 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.
Taking Screenshot in iOS XCTest UI Tests

XCTest UI tests capture screenshots automatically for every step of your tests. These screenshots are displayed in your Device Farm test report. No additional code is required.

Working with 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” (p. 53).

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.

Working with Built-in Tests in AWS Device Farm

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

Built-in Test Types

Built-in tests make it possible for you to test your apps without writing scripts.

- Built-in: Explorer (Android) (p. 73)
- Built-in: Fuzz (Android and iOS) (p. 74)

Working with the Built-in Explorer Test for Device Farm

Device Farm provides a built-in explorer test type.

What Is the Built-in Explorer Test?

The built-in explorer test crawls your app by analyzing each screen and interacting with it as if it were an end user. It takes screenshots as it explores, and you can provide Device Farm with credentials so the test can sign in.

Parameters
• **Username** (Optional). Specifies a user name for the explorer to use if it encounters a sign-in screen in your app. If no user name is provided, Device Farm does not insert a user name.
• **Password** (Optional). Specifies a password for the explorer to use if it encounters a sign-in screen in your app. If no password is provided, Device Farm does not insert a password.

### Use the Built-in Explorer Test Type

Use the Device Farm console to run the built-in explorer test.

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 explorer test.
   
   **Tip**
   
   You can use the search bar to filter the project list by name.
   
   To create a project, follow the instructions in [Create a Project in AWS Device Farm](p. 19).
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 explorer test.
7. Choose **Next**.
8. On the **Configure** page, in the **Setup test framework** section, choose **Built-in: Explorer**.
9. Choose **Next**, and then complete the remaining instructions to select devices and start the run.

### Working with the Built-in Fuzz Test for Device Farm

Device Farm provides a built-in fuzz test type.

**What Is the Built-in Fuzz Test?**

The built-in fuzz test randomly sends user interface events to devices and then reports results.

**Use the Built-in Fuzz Test Type**

Use the Device Farm console to run the built-in fuzz test.

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 [Create a Project in AWS Device Farm](p. 19).
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.
Working with Custom Test Environments in AWS Device Farm

In addition to providing a default standard test environment, AWS Device Farm also allows you to configure a custom environment for your automated tests. For more information, see Test Environments (p. 14).

You can set up and configure your custom automated test environment with a YAML-formatted test specification (test spec) file. Device Farm provides a default test spec for each supported test type. You can use the default test spec as-is or edit it, if you prefer. If you edit the test spec file, you can save it for future runs.

For more information, see Uploading a Custom Test Spec Using the AWS CLI and Create a test run in Device Farm (p. 22).

Topics
- Test Spec Syntax (p. 76)
- Test Spec Example (p. 77)
- Environment Variables (p. 79)
- Migrating Tests from a Standard Test Environment to a Custom Test Environment (p. 81)
- Extending Custom Test Environments in Device Farm (p. 82)

Test Spec Syntax

This is the YAML test spec file structure:

```
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 Using Artifacts in a Custom Test Environment (p. 48).

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.

Test Spec Example

This is an example of a Device Farm YAML test spec that configures an Appium Java TestNG test run:

```yaml
version: 0.1

# Phases are a collection of commands that get executed on Device Farm.
phases:
  # The install phase includes commands that install dependencies that your tests use.
  # Default dependencies for testing frameworks supported on Device Farm are already installed.
```

API Version 2015-06-23
install:
  commands:
  # By default, Appium server version used is 1.7.2.
  # You can switch to an alternate supported version from 1.6.5, 1.7.1, 1.7.2, 1.8.0 or
  # 1.8.1 by using a command like "avm 1.7.1"
  # OR
  # To install a newer version of Appium use the following commands:
  # - export APPIUM_VERSION=1.8.1
  # - avm $APPIUM_VERSION
  # - ln -s /usr/local/avm/versions/$APPIUM_VERSION/node_modules/.bin/appium /usr/
  # local/avm/versions/$APPIUM_VERSION/node_modules/appium/bin/appium.js
  # The pre-test phase includes commands that setup your test environment.
pre_test:
  commands:
  # Setup environment variables for java
  - export CLASSPATH=$CLASSPATH:$DEVICEFARM_TESTNG_JAR
  - export CLASSPATH=$CLASSPATH:$DEVICEFARM_TEST_PACKAGE_PATH/*
  - export CLASSPATH=$CLASSPATH:$DEVICEFARM_TEST_PACKAGE_PATH/dependency-jars/*
  # We recommend starting appium server process in the background using the command
  # below.
  # Appium server log will go to $DEVICEFARM_LOG_DIR directory.
  # The environment variables below will be auto-populated during run time.
  # echo "Start appium server"
  # >-
  # appium --log-timestamp --device-name $DEVICEFARM_DEVICE_NAME
  # --platform-name $DEVICEFARM_DEVICE_PLATFORM_NAME --app $DEVICEFARM_APP_PATH
  # --udid $DEVICEFARM_DEVICE_UDID --chromedriver-executable
  # $DEVICEFARM_CHROMEDRIVER_EXECUTABLE >> $DEVICEFARM_LOG_DIR/appiumlog.txt 2>&1 &
  # >-
  # start_appium_timeout=0;
  # while [ true ];
  # do
  #     if [ $start_appium_timeout -gt 30 ];
  #     then
  #         echo "appium server never started in 30 seconds. Exiting";
  #         exit 1;
  #     fi;
  #     grep -i "Appium REST http interface listener started on 0.0.0.0:4723"
  #     $DEVICEFARM_LOG_DIR/appiumlog.txt >> /dev/null 2>&1;
  #     if [ $? -eq 0 ];
  #     then
  #         echo "Appium REST http interface listener started on 0.0.0.0:4723";
  #         break;
  #     else
  #         echo "Waiting for appium server to start. Sleeping for 1 second";
  #         sleep 1;
  #         start_appium_timeout=$((start_appium_timeout+1));
  #     fi;
  # done;
  # The test phase includes commands that start your test suite execution.
  test:
  commands:
  # Your test package is downloaded in $DEVICEFARM_TEST_PACKAGE_PATH so we first change
directory to that path.
  - echo "Navigate to test package directory"
  - cd $DEVICEFARM_TEST_PACKAGE_PATH
  # By default, the following command is used by Device Farm to run your Appium TestNG
  # test.
  # The goal is to run to your tests jar file with all the dependencies jars in the
  # CLASSPATH.
  # Alternatively, You may specify your customized command.
  # Note: For most use cases, the default command works fine.
Environment Variables

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 (p. 79)
- Appium Java JUnit Environment Variables (p. 80)
- Appium Java TestNG Environment Variables (p. 81)
- XCUITest Environment Variables (p. 81)

Common Environment Variables

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 identifier (UDID) of the device.

$DEVICEFARMDEVICE 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.

$DEVICEFARM_CHROMEDRIVER_EXECUTABLE
The version of the Chrome browser.

$DEVICEFARM_TEST_PACKAGE_NAME
The name of the package (e.g. com.example.myapp) under test.

Note
This is only available when using android native instrumentation tests. For more information, see Working with Instrumentation for Android and AWS Device Farm (p. 66).

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 identifier (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
This section describes environment variables used by the Appium Java JUnit tests in a custom test environment.
Appium Java TestNG Environment Variables

This section describes environment variables used by the Appium Java TestNG tests in a custom test environment.

$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

$DEVICEFARM_XCUITESTRUN_FILE
Path to the Device Farm .xctestrun file. It is generated from your app and test packages.

$DEVICEFARM_DERIVED_DATA_PATH
Expected path of Device Farm xcodebuild output.

Migrating Tests from a Standard Test Environment to a Custom Test Environment

Running a test in a custom environment requires that all information used in the test run is included either in your test spec YAML file or test code. Here are some things to consider when you move a test to a custom environment.

Appium Framework

In a custom test environment, 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

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.

Setting a PIN

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:

```plaintext
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.

Speeding Up Appium-based Tests through Server 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 Driver capabilities in your test spec. For example, the following sets the `usePrebuiltWDA` capability to `true` in an iOS test suite to speed up initial start time:

```plaintext
phases:
  pre_test:
    - # ... Start up Appium
    - appium --log-timestamp
    - 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\"}
```

**API Version 2015-06-23**

82
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.

For more information on the capabilities that Appium supports, see Appium Desired Capabilities in the Appium documentation.

### Using Webhooks and other APIs after Your Tests Run

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:

```bash
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.

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:

```bash
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.
Working with 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.

**Topics**
- Create a Remote Access Session in AWS Device Farm (p. 85)
- Use a Remote Access Session in AWS Device Farm (p. 86)
- Get Results of a Remote Access Session in AWS Device Farm (p. 87)

Create a Remote Access Session in AWS Device Farm

For information about remote access sessions, see Sessions (p. 18).

- Prerequisites (p. 85)
- Create a test run (console) (p. 85)
- Next Steps (p. 86)

**Prerequisites**

- Create a project in Device Farm. Follow the instructions in Create a Project in AWS Device Farm (p. 19), and then return to this page.

Create a Session with the Device Farm Console

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 Create a Project in AWS Device Farm (p. 19).
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.

Use 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 (p. 18).

- Prerequisites (p. 86)
- Use a Session in the Device Farm Console (p. 86)
- Next Steps (p. 87)
- Tips and Tricks (p. 87)

Prerequisites

- Create a session. Follow the instructions in Create a Session (p. 85), 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 Get Session Results (p. 87).

**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.

**Get Results of a Remote Access Session in AWS Device Farm**

For information about sessions, see Sessions (p. 18).

- Prerequisites (p. 87)
- Viewing Session Details (p. 87)
- Downloading Session Video or Logs (p. 88)

**Prerequisites**

- Complete a session. Follow the instructions in Use a Remote Access Session in AWS Device Farm (p. 86), 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 (p. 44).
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.
Working with 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're interested in using a fleet of one or more private devices, contact us. The Device Farm team must work with you to set up and deploy a fleet of one or more private devices for your AWS account.

If you have a private device fleet, you can create remote access sessions and schedule test runs with your private devices. 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 Managing Private Devices in AWS Device Farm (p. 89).

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 Virtual Private Cloud Endpoint Services with AWS Device Farm (p. 97).

**Topics**
- Managing Private Devices in AWS Device Farm (p. 89)
- Skipping App Re-Signing on Private Devices in AWS Device Farm (p. 95)
- Using Amazon Virtual Private Cloud Endpoint Services with AWS Device Farm (p. 97)
- Working with Amazon Virtual Private Cloud Across Regions (p. 100)

Managing 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 in the AWS US West (Oregon) Region (`us-west-2`) only.

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.

This topic explains how to create an instance profile and perform other common device management tasks.

**Topics**
- Creating an Instance Profile (p. 90)
- Managing a Private Device Instance (p. 91)
- Creating a Test Run or Starting a Remote Access Session (p. 93)
Creating an Instance Profile

To control the behavior of private devices during a test run or remote access session, you can create or modify an instance profile in Device Farm. You do not need an instance profile to start using your private devices.

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.
   
   ![Create a new instance profile form](image)

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.
Managing a Private Device Instance

If you already have one or more private devices in your fleet, you can view information about and manage certain settings for each device instance. You can also request an additional private device instance.

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. (Optional) 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.
Edit device instances

Instance ID
ID for the private device instance.

Mobile
Model of the private device.
Google Pixel 4 XL (Unlocked)

Platform
Platform of the private device.
Android

OS Version
OS version of the private device.
10

Status
Status of the private device.
Available

Profile
Choose a profile to attach to the device.

Instance profile details

Name:

Reboot after use: false

Package Cleanup: false

Excluded Packages:

Labels
Labels are custom strings that can be attached to private devices.

Example

+ Add new
6. (Optional) For **Profile**, choose an instance profile to attach to the device instance. This can be helpful if you want to always exclude a specific app package from cleanup tasks, for example.

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

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.

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” (p. 93).
   - 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**.

## Creating a Private Device Pool

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.

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.

6. Select **Create static device pool**, then select **Private device instances only** to limit the list of devices to only those private devices that the Device Farm team set up and associated with your AWS account.

7. Select each device that you want to include in the test run.

8. (Optional) To create a device pool that changes dynamically as new devices become available, choose **Create dynamic device pool**, then choose **Add rule**. Set the values for **Field**, **Operator**, and **Value** according to your requirements.

   **Important**
   The **Private device instances only** option is not available when creating dynamic device pools.

9. Choose **Create**.

**Next Steps**

After you set up your private devices, you can also manage your private devices in the following ways:
• Skip app re-signing on private devices (p. 95)
• Use Amazon Virtual Private Cloud endpoint services with Device Farm (p. 97)

To delete an instance profile, on the Instance profiles menu, choose the toggle option next to the instance that you want to delete, then choose Delete.

Skipping App Re-Signing on Private Devices in AWS Device Farm

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 AWS Device Farm FAQs.

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 Working with XCTest for iOS and AWS Device Farm (p. 70).

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" (p. 96).

- 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, open the Device Farm console and navigate to the **Device instances** tab of the **Project settings** page or 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** (p. 90) 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 Create a Session (p. 85) 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.

### Using Amazon Virtual Private Cloud Endpoint Services with AWS Device Farm

**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.

If you use Amazon Virtual Private Cloud (Amazon VPC) to host private applications in the AWS US West (Oregon) Region (us-west-2), you can establish a private connection between your VPC and AWS 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.

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. 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 in the Amazon VPC User 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
Before You Begin

The following information is for users of Amazon VPC in the AWS US West (Oregon) 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 it can be used 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 Availability Zones for a VPC endpoint service are determined by the Network Load Balancer that's associated with the 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 whitelisted principals for the endpoint service must include the Amazon Resource Name of the Device Farm VPC endpoint (service ARN). After you create your endpoint service, add the Device Farm VPC endpoint service ARN to your whitelist 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 the Require acceptance for endpoint setting is enabled, you must manually accept each connection request that Device Farm sends to the endpoint service. You can disable this setting when you create the VPC endpoint service. You can also change this setting for an existing endpoint service. To change this setting, choose the endpoint service on the Amazon VPC console, choose Actions, and then choose Modify endpoint acceptance setting.

The next section explains how to create an Amazon VPC endpoint service that meets these requirements.

Step 1: Creating an Amazon VPC Endpoint Service

The first step in establishing a private connection between your VPC and Device Farm is to 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. If you already have a Network Load Balancer that you want the endpoint service to use, choose it in the list, and then skip to step 11.
5. Next to Associate Network Load Balancers, choose Create new Network Load Balancers.
6. Choose Create Load Balancer, and then choose Network Load Balancer.
7. Enter a name and settings for the Network Load Balancer. Under Availability Zones, enable the us-west-2a, us-west-2b, and us-west-2c Availability Zones.
8. Follow the instructions to configure the Network Load Balancer.
9. Repeat steps 1 through 3 to restart the process of creating an endpoint service that uses the new Network Load Balancer.
10. Choose the Network Load Balancer that you created for the endpoint service.
11. For **Included Availability Zones**, verify that **us-west-2a**, **us-west-2b**, and **us-west-2c** appear in the list.

12. If you don't want to manually accept or deny each connection request that is sent to the endpoint service, clear **Require acceptance for endpoint**. If you clear this check box, the endpoint service automatically accepts each connection request that it receives.

13. Choose **Create service**.

14. Choose **Close**.

15. In the list of endpoint services, choose the endpoint service that you created.

16. Choose **Whitelisted principals**.

17. Contact us to get the Amazon Resource Name of the Device Farm VPC endpoint (service ARN) to add to the whitelist for the endpoint service, and then add that service ARN to the whitelist for the service.

18. 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 for use with Device Farm.

**Step 2: Creating a VPC Endpoint Configuration in Device Farm**

After you create an endpoint service in Amazon VPC, you can create an Amazon VPC endpoint (VPCE) configuration in Device Farm.


2. On the Device Farm navigation panel, choose **Mobile Device Testing**, then choose **Private devices**.

3. Choose **VPCE configurations**.

4. Choose **Create VPCE configuration**.

5. Enter a name for the VPCE configuration.

6. Enter the name of the Amazon VPC endpoint service (**service name**) that you noted on the Amazon VPC console. The name looks like *com.amazonaws.vpce.us-west-2.vpce-svc-id*.

7. 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 VPCE configuration**.
Step 3: Creating a Test Run

After you save the VPCE configuration, you can use the configuration to create test runs or remote access sessions. For more information, see Create a Test Run (p. 22) or Create a Remote Access Session (p. 85).

Working with Amazon Virtual Private Cloud Across Regions

This topic describes how to reach an Amazon Virtual Private Cloud (Amazon VPC) endpoint in another AWS Region. If AWS Device Farm and your VPC endpoint are in the same AWS Region, see the section called “Using VPC Endpoint Services” (p. 97).

You can connect any two VPCs in different AWS Regions, as long as they have distinct, non-overlapping CIDR blocks. This ensures that all of the private IP addresses are unique and allows all of the resources in the VPCs to address each other without the need for any form of network address translation. For more information about CIDR notation, see RFC 4632.

This topic includes a cross-region example scenario in which Device Farm is located in the AWS US West (Oregon) Region (us-west-2) and is referred to as VPC-1. The second VPC in this example is in another AWS Region and is referred to as VPC-2.

Device Farm VPC Cross-Region Example

<table>
<thead>
<tr>
<th>VPC Component</th>
<th>VPC-1</th>
<th>VPC-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIDR</td>
<td>10.0.0.0/16</td>
<td>172.16.0.0/16</td>
</tr>
<tr>
<td>Public subnet</td>
<td>10.0.0.0/24</td>
<td>172.16.0.0/24</td>
</tr>
</tbody>
</table>
The following diagram shows the components in the example and the interactions between these components.

**Topics**
- Prerequisites (p. 101)
- Step 1: Connect Device Farm to a VPC in the Same Region (p. 101)
- Step 2: Set Up an OpenVPN Server in the Device Farm Region (VPC-1) (p. 102)
- Step 3: Set Up an OpenVPN Server in a Second Region (VPC-2) (p. 103)
- Step 4: Configure VPC-1 (p. 103)
- Step 5: Configure VPC-2 (p. 105)
- Step 6: Create a Test Run (p. 105)

### Prerequisites

This example requires the following:
- Two VPCs that are configured with public and private subnets.
- An Elastic IP address that's associated with each of the VPC-1 and VPC-2 public subnets.

### Step 1: Connect Device Farm to a VPC in the Same Region

Establish a private connection, as an AWS PrivateLink connection, between Device Farm and an endpoint in your VPC. For more information, see the section called “Using VPC Endpoint Services” (p. 97).
Step 2: Set Up an OpenVPN Server in the Device Farm Region (VPC-1)

1. Open the Amazon VPC console. You might be prompted for your AWS credentials.
2. From the VPC Dashboard, choose Launch EC2 Instances.
3. From the left navigation bar, choose AWS Marketplace.

5. Choose Select to display the OpenVPN summary page, and then choose Continue.
6. Choose an Amazon EC2 instance type.
7. Choose Next: Configure Instance Details. For Subnet, choose your public subnet.
8. Accept the defaults on these pages:
   a. Choose Next: Add Storage.
   b. Choose Next: Add Tags.
9. Confirm the OpenVPN security group settings:
   • SSH – Port 22
   • Custom TCP Rule – Port 943
   • HTTPS – Port 443
   • Custom UDP Rule – Port 1194
11. Choose any media type, and then choose Next.
12. Choose Launch.
13. Choose or create an instance, and then choose Launch Instances.
14. It can take some time for the instance to launch. Choose View Instances to track the status of your Amazon EC2 instance.
15. Disable the source and destination IP address checks for your VPC traffic:
Step 3: Set Up VPC-2

a. On the EC2 Instances page, from Actions, choose Networking, and then choose Change Source/Dest Check.
b. Choose Yes, Disable.

To configure your OpenVPN server

1. Sign in to your OpenVPN Amazon EC2 instance using SSH, the user name openvpnas, and the key pair that you set for this instance. For more information, see Connecting to Your Linux Instance Using SSH.
2. The OpenVPN Access Server Setup wizard runs automatically when you first sign in. Use this command to run it again:

   sudo vpn-init --ec2

Step 3: Set Up an OpenVPN Server in a Second Region (VPC-2)

Use the information in step 2 (p. 102) to set up an OpenVPN server in the public subnet of your second region (VPC-2).

Step 4: Configure VPC-1

1. Open the Amazon VPC console. You might be prompted for your AWS credentials.
2. Choose Customer Gateways, enter the gateway settings, and create the customer gateway:
   a. Choose Create Customer Gateway.
   b. For Routing, choose Static.
c. For **Name**, enter a name for your gateway.

d. For **IP Address**, enter the public IP address of your OpenVPN Access Server instance.

e. Choose **Create Customer Gateway**.

f. If successful, the customer gateway ID is displayed. Choose **Close**.

3. Choose **Virtual Private Gateways**, and then create the virtual private gateway (VPG):

   a. For **Name**, enter a name for your VPG.

   b. Choose **Create Virtual Private Gateway**.

4. Choose the VPG that you just created, and attach it to the VPC:

   a. From **Actions**, choose **Attach to VPC**.

   b. From **VPC**, choose your VPC.

   c. From **Routing Options**, choose **Static**. Enter your IP address in CIDR notation.

   d. Choose **Yes, Attach**.

5. Choose **Route Tables**, and then configure the routing settings:

   a. Choose the routing table that corresponds to your subnet.

   b. On the **Route Propagation** tab, choose **Edit Route Propagation**.

   c. Next to the VGW identifier for the virtual private gateway that you created earlier, choose **Enable**.

   d. Choose **Save**.

6. Choose **Site-to-Site VPN Connections**, and then create the VPN connection:

   a. Choose **Create VPN Connection**.

   b. From **Virtual Private Gateway**, choose your virtual private gateway.

   c. From **Customer Gateway ID**, choose your existing customer gateway.

   d. From **Routing Options**, choose **Static**. For **Static IP Addresses**, enter your Elastic IP address. For example, if your static IP address is 10.12.34.56, then your CIDR notation for the IP prefix is 10.12.34.0/24.

   e. Choose **Create VPN Connection**.

   f. If successful, a VPN connection ID is displayed. Choose **Close**.

7. Choose **Use Static Routing**.

8. Enter the Elastic IP address of the OpenVPN Access VPN server. Choose the VPN connection, and make a note of the Tunnel 1 and Tunnel 2 IP addresses in the console. You need them later in this procedure.

9. Choose **Download Configuration**.

10. Use SSH to connect to your OpenVPN Access Server instance, and then open the `/etc/ipsec.conf` file:

```
sudo /etc/ipsec.conf
```

11. Edit the `rightsubnet=` value to point to your VPC CIDR mask.

12. Under the VPC-CUST-GW1 and VPC-CUST-GW2 sections, add the Tunnel 1 and Tunnel 2 IP addresses and save the file.

13. Open the `/etc/ipsec.secrets` file, and then enter the preshared keys from the VPC-1 configuration file that you downloaded earlier.

14. To start the VPN connection, use the `ipsec start` command.

You can see the status of your VPN connection entries in the Amazon VPC console.
Step 5: Configure VPC-2

Use the information in step 4 (p. 103) to configure VPC-2. Configure the routing tables in both VPCs to send traffic to the other VPC through the VPC EC2 instances.

Note
You might need to configure multiple routing tables for your public and private subnets, depending on which subnets you want to route traffic between.

For more information about this example scenario and an alternative VPN implementation of it, see Connecting Multiple VPCs with EC2 Instances (SSL).

Step 6: Create a Test Run

You can create test runs that use the VPCE configuration described in step 1 (p. 101). For more information, see Create a Test Run (p. 22) or Create a Remote Access Session (p. 85).
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 (p. 113) and the Automating Device Farm (p. 115). 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": "AKIAI44QH8H8DEXAMPLE",
        "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": {
      "responseElements": {
        "runs": [ 
          {
            "created": "Jul 8, 2015 11:26:12 PM",
            "name": "example.apk",
            "completedJobs": 2,
            "counters": {
              "stopped": 0,
              "warned": 0,
              "failed": 0,
              "pass": 4,
              "skipped": 0,
              "total": 4,
              "errored": 0
            }
          },
          {
            "type": "BUILTIN_FUZZ",
          {
            "counters": {
              "stopped": 0,
              "warned": 0,
              "failed": 0,
              "pass": 4,
              "skipped": 0,
              "total": 4,
              "errored": 0
            }
          }
        ]
      }
    }
  ]
}``
"status": "RUNNING",
"totalJobs": 3,
"platform": "ANDROID_APP",
"result": "PENDING"
},
... additional entries ...
]}
}
}
]}
}
Using AWS Device Farm in an 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.

Configure CodePipeline to Use Your Device Farm Tests

In these steps, we assume that you have configured a Device Farm project (p. 19) 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 a 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**.

![Add action panel](image)

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.

![Device Farm Test](image)

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.

![App file path](image)
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.

```python
import boto3
import os
import requests
import string
import random
import time
import datetime

# 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",
    # Since we care about the most popular devices, we'll use a curated pool.
    "testSpecArn": "arn:aws:devicefarm:us-west-2::upload:4f8bd6b2-7be5-11e8-adc0-fa7ae01bbbc",
    "poolArn": "arn:aws:devicefarm:us-west-2::devicepool:082d10e5-d7d7-48a5-ba5c-b33d6e6e1f5",
    "namePrefix": "MyAppTest",
    # This is our test package. This tutorial won't go into how to make these.
    "testPackage": "tests.zip"
}

client = boto3.client('devicefarm')
```

API Version 2015-06-23
115
Example: Using the AWS SDK to start a Device Farm run and collect artifacts

```python
unique = config['namePrefix']+'-'+(datetime.date.today().isoformat())+''.join(random.sample(string.ascii_letters,8))
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']}...','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:
                    \"{response['upload']['message']}
            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...

```python
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']"arm']
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)
# 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.

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 Working with Android Tests in AWS Device Farm (p. 65).

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'
maint
  supports-screens: 'small' 'normal' 'large' 'xlarge'
supports-any-density: 'true'
local: '--_--'
densities: '160' '213' '240' '320' '480' '640'
```

For more information, see Working with Android Tests in AWS Device Farm (p. 65).

**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 Working with Android Tests in AWS Device Farm (p. 65).
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 `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 [Working with Android Tests in AWS Device Farm](p. 65).

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:

```
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")
```
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:

  ```bash
  $ aapt dump xmltree app-debug.apk AndroidManifest.xml
  ```

  You should find output like the following:

  ```xml
  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 **Working with Android Tests in AWS Device Farm (p. 65)**.
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_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 Working with Appium and AWS Device Farm (p. 53).

**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 Working with Appium and AWS Device Farm (p. 53).

---

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)
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 **Working with Appium and AWS Device Farm (p. 53)**.

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:

```bash
# 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`.

```bash
|-- 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:

```bash
# 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 Working with Appium and AWS Device Farm (p. 53).

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:

```bash
$ 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:

```bash
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 Working with Appium and AWS Device Farm (p. 53).

---

**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:

```bash
$ 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:

```bash
$ 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:
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" (p. 53).

**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:

   ```bash
   # 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:

   ```bash
   # 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)
      |-- 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 Working with Appium and AWS Device Farm (p. 53).
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:

   ```bash
   $ 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:

   ```bash
   $ 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
```

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:

   ```bash
   $ 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:

   ```bash
   $ 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)
`-- 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
```
If the Appium Java JUnit package is valid, you will find at least one jar file inside the dependency-jars 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)
|--- 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
```

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`.

```
$ 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)
|--- 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
```
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
   |— 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
   ```
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:

```bash
$ 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:

```bash
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:

```bash
$ 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:

```bash
$ tree .
```

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.
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)
  `-- 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
```

**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 Working with Appium and AWS Device Farm (p. 53).

### 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:

   ```bash
   # 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:

   ```bash
   # tree .
   ```

   A valid Appium Java JUnit package should produce output like the following:

   ```
   .
   ```
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:

   ```bash
   # 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:

   ```bash
   # 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 Working with Appium and AWS Device Farm (p. 53).
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:

   ```bash
   $ 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:

   ```bash
   $ tree .
   ```

If the Appium Java JUnit package is valid, you will find at least one `jar` file inside the `dependency-jars` directory.

```
.        # the working directory tree structure
|-- 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 Working with Appium and AWS Device Farm (p. 53).

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:

   ```bash
   $ 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:

   ```bash
   $ 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`.

API Version 2015-06-23
134
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:

```bash
$ 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:

```bash
$ 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`.

3. To extract files from the jar file, you can run the following command:

```bash
$ jar xf acme-android-appium-1.0-SNAPSHOT-tests.jar
```

4. After you successfully extract the files, run the following command:

```bash
$ tree .
```
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:

   ```bash
   $ 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:

   ```bash
   $ 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)
├── 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_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 Working with Appium and AWS Device Farm (p. 53).

APPIUM_WEB_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 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 Working with Appium and AWS Device Farm (p. 53).
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 Working with Appium and AWS Device Farm (p. 53).

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)
```
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:

   ```bash
   # 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:

   ```bash
   # 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:

   ```bash
   # jar xf acme-android-appium-1.0-SNAPSHOT-tests.jar
   ```

4. After you successfully extract the files, run the following command:

   ```bash
   # 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)
   ```
For more information, see Working with Appium and AWS Device Farm (p. 53).

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.

**APIUM_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:

   ```bash
   $ unzip test_bundle.zip
   ``

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ 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 Working with Appium and AWS Device Farm (p. 53).
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:

   ```bash
   $ unzip test_bundle.zip
   ``

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ 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 Working with Appium and AWS Device Farm (p. 53).

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:

   ```bash
   $ unzip test_bundle.zip
   ``

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
```
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.

For more information, see Working with Appium and AWS Device Farm (p. 53).

**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:

   ```bash
   # unzip test_bundle.zip
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   # tree .
   ```

   If the Appium Python package is valid, you will find the tests directory inside the working directory.
For more information, see Working with Appium and AWS Device Farm (p. 53).

**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 Working with Appium and AWS Device Farm (p. 53).

**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 Working with Appium and AWS Device Farm (p. 53).

---

**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:
You should find output like the following:

```plaintext
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 Working with Appium and AWS Device Farm (p. 53).

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:

```bash
# 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:

```bash
# unzip test_bundle.zip
```

2. To test installing wheel files, you can run the following command:

```bash
# pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./requirements.txt
```

A valid Appium Python package should produce output like the following:

```plaintext
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 Working with Appium and AWS Device Farm (p. 53).

API Version 2015-06-23

---

**Warning**

We failed to collect tests in the tests directory. Please unzip your test package, verify 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 Working with Appium and AWS Device Farm (p. 53).
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:

```
$ tree .
|-- 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 Working with Appium and AWS Device Farm (p. 53).

**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:
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 Working with Appium and AWS Device Farm (p. 53).

**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)
```
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:

   ```bash
   $ unzip test_bundle.zip
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ tree .
   ```

   If the Appium Python package is valid, you will find the `tests` directory inside the working directory.

```bash
.
|-- 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 Working with Appium and AWS Device Farm (p. 53).

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`.

```bash
.
|-- 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
```
1. Copy your test package to your working directory, and then run the following command:

   ```bash
   # unzip test_bundle.zip
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ 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`.

```plaintext
|-- 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 Working with Appium and AWS Device Farm (p. 53).

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:

   ```bash
   # unzip test_bundle.zip
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ tree .
   ```

If the Appium Python package is valid, you will find the `requirements.txt` file inside the working directory.

```plaintext
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
   |-- `-- test_unittest.py
   `-- wheelhouse (directory)
      |-- Appium_Python_Client-0.20-cp27-none-any.whl
```
For more information, see Working with Appium and AWS Device Farm (p. 53).

### 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:

   ```bash
   $ unzip test_bundle.zip
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ 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:

   ```bash
   $ 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 Working with Appium and AWS Device Farm (p. 53).
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 [Working with Appium and AWS Device Farm](p. 53).

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, verify 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 Working with Appium and AWS Device Farm (p. 53).

Troubleshooting Calabash Tests in AWS Device Farm

The following topic lists error messages that occur during the upload of Calabash tests and recommends workarounds to resolve each error.
CALABASH_TEST_PACKAGE_UNZIP_FAILED_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 `features.zip`.

1. Copy your test package to your working directory, and then run the following command:

   ```bash
   $ unzip features.zip
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ tree .
   ```

A valid Calabash package should produce output like the following:

```
|-- features (directory)
   |-- my-feature-1-file-name.feature
   |-- my-feature-2-file-name.feature
   |-- my-feature-N-file-name.feature
   |-- step_definitions (directory)
   |   `-- (.rb files)
   |-- support (directory)
   |   `-- (.rb files)
   `-- (any other supporting files)
```

For more information, see Android tests with Calabash (p. 64) or iOS test with Calabash (p. 64).

CALABASH_TEST_PACKAGE_FEATURES_DIR_MISSING_FEATURES_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

Warning
We could not find the features directory inside your test package tree. Please unzip your test package, verify that the features directory is inside the package, and try again.

In the following example, the package's name is `features.zip`.

1. Copy your test package to your working directory, and then run the following command:

   ```bash
   $ unzip features.zip
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ tree .
   ```

If the Calabash package is valid, you will find the `features` directory inside the working directory.

.
CALABASH_TEST_PACKAGE_FEATURE_FILE_MISSING

If you see the following message, follow these steps to fix the issue.

**Warning**

We could not find a .feature file in the features directory tree. Please unzip your test package and open the features directory, verify that at least one .feature file is in the directory, and try again.

In the following example, the package's name is `features.zip`.

1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip features.zip
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```
   $ tree .
   ```

   If the Calabash package is valid, you will find at least one `.feature` file inside the `features` directory.

   ```
   `-- features (directory)
       |-- my-feature-1-file-name.feature
       |-- my-feature-2-file-name.feature
       |-- my-feature-N-file-name.feature
       |-- step_definitions (directory)
           |-- (.rb files)
           |-- support (directory)
           |-- (.rb files)
           `-- (any other supporting files)
   ```

   For more information, see Android tests with Calabash (p. 64) or iOS test with Calabash (p. 64).

CALABASH_TEST_PACKAGE_STEP_DEFINITIONS_DIR_MISSING

If you see the following message, follow these steps to fix the issue.

**Warning**

We could not find the step_definitions directory inside your test package. Please unzip your test package and open the features directory, verify that the step_definitions directory is inside the package, and try again.

```
`-- features (directory)
    |-- my-feature-1-file-name.feature
    |-- my-feature-2-file-name.feature
    |-- my-feature-N-file-name.feature
    |-- step_definitions (directory)
        |-- (.rb files)
        |-- support (directory)
        |-- (.rb files)
        `-- (any other supporting files)
```

For more information, see Android tests with Calabash (p. 64) or iOS test with Calabash (p. 64).
In the following example, the package's name is **features.zip**.

1. Copy your test package to your working directory, and then run the following command:

   ```bash
   $ unzip features.zip
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ tree .
   ```

If the Calabash package is valid, you will find the `step_definitions` directory inside the `features` directory.

```
|-- features (directory)
   |-- my-feature-1-file-name.feature
   |-- my-feature-2-file-name.feature
   |-- my-feature-N-file-name.feature
   `-- step_definitions (directory)
      `-- (.rb files)
   `-- support (directory)
      `-- (.rb files)
      `-- (any other supporting files)
```

For more information, see Android tests with Calabash (p. 64) or iOS test with Calabash (p. 64).

---

**CALABASH_TEST_PACKAGE_SUPPORT_DIR_MISSING**

If you see the following message, follow these steps to fix the issue.

**Warning**
We could not find the support directory inside your test package. Please unzip your test package and open the features directory, verify that the support directory is inside the package, and try again.

In the following example, the package's name is **features.zip**.

1. Copy your test package to your working directory, and then run the following command:

   ```bash
   $ unzip features.zip
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ tree .
   ```

If the Calabash package is valid, you will find the `support` directory inside the `features` directory.

```
|-- features (directory)
   |-- my-feature-1-file-name.feature
   |-- my-feature-2-file-name.feature
   |-- my-feature-N-file-name.feature
   `-- step_definitions (directory)
      `-- (.rb files)
   `-- support (directory)
      `-- (any other supporting files)
```
CALABASH_TEST_PACKAGE_RUBY_FILE_MISSING_IN_STEP_DEFINITIONS_DIR

If you see the following message, follow these steps to fix the issue.

**Warning**

We could not find a ruby file in the step_definitions directory tree. Please unzip your test package and open the step_definitions directory, verify that at least one ruby file is in the directory, and try again.

In the following example, the package's name is `features.zip`.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip features.zip
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

If the Calabash package is valid, you will find at least one `ruby` file inside the `step_definitions` directory.

```
|-- features (directory)
 |   |-- my-feature-1-file-name.feature
 |   |-- my-feature-2-file-name.feature
 |   |-- my-feature-N-file-name.feature
 |   `-- step_definitions (directory)
 |       |-- one-ruby.rb
 |       |-- folder
 |       `-- `another-ruby.rb
 |           |-- (any other supporting files)
 |       `-- support (directory)
 |           `-- (.rb files)
 |                 `-- (any other supporting files)
```

For more information, see Android tests with Calabash (p. 64) or iOS test with Calabash (p. 64).

CALABASH_TEST_PACKAGE_RUBY_FILE_MISSING_IN_SUPPORT_DIR

If you see the following message, follow these steps to fix the issue.

**Warning**

We could not find a ruby file in the support directory tree. Please unzip your test package and open the support directory, verify that at least one ruby file is in the directory, and try again.

In the following example, the package's name is `features.zip`.

1. Copy your test package to your working directory, and then run the following command:

```
$ unzip features.zip
```

For more information, see Android tests with Calabash (p. 64) or iOS test with Calabash (p. 64).
# unzip features.zip

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```bash
# tree .
```

If the Calabash package is valid, you will find at least one *ruby* file inside the *support* directory.

```
|-- features (directory)
 |  |-- my-feature-1-file-name.feature
 |  |-- my-feature-2-file-name.feature
 |  |-- my-feature-N-file-name.feature
 |  |-- step_definitions (directory)
 |  |  `-- (.rb files)
 |  |-- support (directory)
 |  |     |-- one-ruby.rb
 |  |     |
 |  |     `-- another-ruby.rb
 |  |     `-- (any other supporting files)
 |  `-- (any other supporting files)
```

For more information, see [Android tests with Calabash](#) (p. 64) or [iOS test with Calabash](#) (p. 64).

## CALABASH_TEST_PACKAGE_EMBEDDED_SERVER_MISSING

If you see the following message, follow these steps to fix the issue.

**Warning**

We could not find the embedded server inside your test package. Please verify that the server is inside the package by running the command "calabash-ios check <path to your test package>" , and try again after finding the calabash framework.

Calabash tests contain an embedded web server within the iOS application.

Make sure that the embedded web server is inside your iOS application. In the following example, the iOS application’s name is `AWSDeviceFarmiOSReferenceApp.ipa`.

- Copy your iOS application to your working directory, and then run the following command:

```bash
$ calabash-ios check AWSDeviceFarmiOSReferenceApp.ipa
```

A valid iOS application should produce output like the following:

```
Ipa: AWSDeviceFarmiOSReferenceApp.ipa *contains* calabash.framework
0.19.0
```

For more information, see [Android tests with Calabash](#) (p. 64) or [iOS test with Calabash](#) (p. 64).

## CALABASH_TEST_PACKAGE_DRY_RUN_FAILED

If you see the following message, follow these steps to fix the issue.
Warning
We failed to quickly scan your .feature files. Please unzip your test package, verify that the files are valid by running the command "calabash --dry-run <path to your features directory>"; and try again after the command does not print any error.

During the upload validation process, Device Farm quickly scans your features without actually running them.

Make sure that your features are valid. In the following example, the package's name is features.zip.

1. Copy your test package to your working directory, and then run the following command:

   ```sh
   $ unzip features.zip
   ```

   After you successfully unzip your package, you will find the features directory inside the working directory.

2. To scan your features, run the following command:

   ```sh
   $ cucumber-ios --dry-run --format json features
   ```

   A valid Calabash package should produce output like the following:

   ```json
   {
     "uri": "features/homepage.feature",
     "id": "home-page",
     "keyword": "Feature",
     "name": "Home Page",
     "description": "As a Device Farm user\n I would like to be able to see examples of testing a static homepage\n So I can apply it to my future tests.\",
     "line": 1,
     "elements": [
       {
         "id": "home-page;a-valid-homepage",
         "keyword": "Scenario",
         "name": "A Valid Homepage",
         "description": "",
         "line": 6,
         "type": "scenario",
         "steps": [
           {
             "keyword": "Given ",
             "name": "that I navigate to the \"Home\" menu category",
             "line": 7,
             "match": {
               "location": "/Library/Ruby/Gems/2.0.0/gems/cucumber-2.4.0/lib/cucumber/step_match.rb:98"
             },
             "result": {
               "status": "skipped",
               "duration": 16000
             }
           }
         ]
       }
     ]
   }
   ```

   For more information, see Android tests with Calabash (p. 64) or iOS test with Calabash (p. 64).
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.

**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 [Working with Instrumentation for Android and AWS Device Farm](p. 66).

**INSTRUMENTATION_TEST_PACKAGE_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 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-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 Working with Instrumentation for Android and AWS Device Farm (p. 66).

INSTRUMENTATION_TEST_PACKAGE_INSTRUMENTATION_RUNNER_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

**Warning**

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")
```

API Version 2015-06-23
161
If you see the following message, follow these steps to fix the issue.

Warning
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:

```
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:targetPackage(0x01010021)="com.amazon.aws.adf.android.referenceapp" (Raw: "com.amazon.aws.adf.android.referenceapp")
  A: android:name(0x01010003)="android.support.test.runner.AndroidJUnitRunner" (Raw: "android.support.test.runner.AndroidJUnitRunner")
E: application (line=16)
  A: android:label(0x01010001)=@0x7f020000
  A: android:debuggable(0x0101000f)=(type 0x12)0xffffffff
E: uses-library (line=17)
  A: android:name(0x01010001)="android.test.runner" (Raw: "android.test.runner")
```

For more information, see Working with Instrumentation for Android and AWS Device Farm (p. 66).
If you see the following message, follow these steps to fix the issue.

**Warning**
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:

  ```bash
  $ aapt debug badging app-debug-androidTest-unaligned.apk | grep "package: name=
  ```

  A valid Instrumentation test package will produce output like the following:

  ```bash
  package: name='com.amazon.aws.adf.android.referenceapp.test' versionCode='' versionName='' platformBuildVersionName='5.1.1-1819727'
  ```

  For more information, see Working with Instrumentation for Android and AWS Device Farm (p. 66).

**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:

   ```bash
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ 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 Working with iOS Tests in AWS Device Farm (p. 68).

**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:

   ```bash
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ 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 Working with iOS Tests in AWS Device Farm (p. 68).

**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:

   ```bash
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```
2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```bash
$ 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 Working with iOS Tests in AWS Device Farm (p. 68).

**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:

```bash
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```bash
$ 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 Working with iOS Tests in AWS Device Farm (p. 68).

**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:

   ```bash
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ 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:

   ```bash
   $ pip install biplist
   ```

   Next, open Python and run the following command:

   ```python
   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 Working with iOS Tests in AWS Device Farm (p. 68).

---

**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:

   ```bash
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:
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 Working with iOS Tests in AWS Device Farm (p. 68).

---

**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:

   ```bash
   # pip install biplist
   ``

4. Next, open Python and run the following command:

   ```python
   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:

   ```python
   ['iPhoneOS']
   ```

   If the iOS application is valid, the value should not contain the keyword `simulator`.

   For more information, see Working with iOS Tests in AWS Device Farm (p. 68).

## 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:

   ```bash
   # unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   # 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:

```sh
$ pip install biplist
```

4. Next, open Python and run the following command:

```python
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 Working with iOS Tests in AWS Device Farm (p. 68).

**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:

```sh
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```sh
$ 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:

```sh
$ pip install biplist
```

4. Next, open Python and run the following command:

```python
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 Working with iOS Tests in AWS Device Farm (p. 68).

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
   ```
Troubleshooting UI Automator Tests in AWS Device Farm

The following topic lists error messages that occur during the upload of UI Automator tests and recommends workarounds to resolve each error.

**UIAUTOMATOR_TEST_PACKAGE_UNZIP_FAILED**

If you see the following message, follow these steps to fix the issue.

**Warning**
We could not open your test JAR file. Please verify that the file is valid and try again.

**Note**
The instructions below are based on Linux x86_64 and Mac.

Make sure that you can unzip the application package without errors. In the following example, the package's name is `com.uiautomator.example.jar`.

1. Copy your application package to your working directory, and then run the following command:

   ```bash
   $ unzip com.uiautomator.example.jar
   ```

2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```bash
   $ tree .
   ```

   A valid UI Automator package should produce output like the following:

   
   ```
   .
   |-- classes.dex
   |-- META-INF (directory)
   |   `-- MANIFEST.MF
   `- (any other files)
   ```

   For more information, see [UI Automator](p. 67).

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.

**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.

**Note**
The instructions below assume you are using MacOS.
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:

   ```bash
   $ 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:

   ```bash
   $ tree .
   ```

   A valid XCTest package should produce output like the following:

   ```
   |-- swiftExampleTests.xctest (directory)
     |-- Info.plist
     |-- (any other files)
   ```

   For more information, see Working with XCTest for iOS and AWS Device Farm (p. 70).

**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:

   ```bash
   $ 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:

   ```bash
   $ 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 Working with XCTest for iOS and AWS Device Farm (p. 70).

**XCTEST_TEST_PACKAGE_PLIST_FILE_MISSING**

If you see the following message, follow these steps to fix the issue.

API Version 2015-06-23

172
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:

   ```bash
   $ 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:

   ```bash
   $ 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 Working with XCTest for iOS and AWS Device Farm (p. 70).

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:

   ```bash
   $ 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:

   ```bash
   $ 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:

```python
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 Working with XCTest for iOS and AWS Device Farm (p. 70).

**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:

```bash
$ 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:

```bash
$ 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:

   ```bash
   $ pip install biplist
   ```

4. Next, open Python and run the following command:

```python
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:
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.

**Warning**
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:

```bash
$ 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:

```bash
$ 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 **XCTest UI** (p. 72).

**XCTEST_UI_TEST_PACKAGE_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 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 **XCTest UI (p. 72)**.

---

**XCTEST_UI_TEST_PACKAGE_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 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)
   ```

API Version 2015-06-23
**XCTEST_UI_TEST_PACKAGE_PLUGINS_DIR_MISSING**

If you see the following message, follow these steps to fix the issue.

**Warning**
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 **XCTest UI (p. 72)**.

**XCTEST_UI_TEST_PACKAGE_XCTEST_DIR_MISSING_IN_PLUGINS_DIR**

If you see the following message, follow these steps to fix the issue.

**Warning**
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 the .xctest directory inside the plugins 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 **XCTest UI (p. 72)**.
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 XCTest UI (p. 72).

**XCTEST_UI_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 .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:

   ```bash
   $ 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:

   ```bash
   $ 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 XCTest UI (p. 72).

**XCTEST_UI_TEST_PACKAGE_PLIST_FILE_MISSING_IN_XCTEST_DIR**

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 swift-sample-UI.ipa.

1. Copy your test package to your working directory, and then run the following command:

   ```bash
   $ 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:

   ```bash
   $ 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 XCTest UI (p. 72).

XCTEST_UI_TEST_PACKAGE_CPU_ARCHITECTURE_VALUE_MISSING

If you see the following message, follow these steps to fix the issue.

Warning
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 “UIRequiredDeviceCapabilties” 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:

   ```bash
   $ 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:

   ```bash
   $ tree .
   ```

   You should find the Info.plist file inside an .app directory like swift-sampleUITests-Runner.app in our example:

```
`. `. Payload (directory)
```
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:

   ```python
   $ pip install biplist
   ```

4. Next, open Python and run the following command:

   ```python
   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:

   ```python
   ['armv7']
   ```

   For more information, see XCTest UI (p. 72).

---

**XCTEST_UI_TEST_PACKAGE_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. Please unzipped 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:

   ```bash
   $ 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:

   ```bash
   $ tree .
   ```

   You should find the **Info.plist** file inside an .app directory like **swift-sampleUITests-Runner.app** in our example:

   ```bash
   `-- Payload (directory)
     `-- swift-sampleUITests-Runner.app (directory)
         `-- Info.plist
         `-- Plugins (directory)
          `-- swift-sampleUITests.xctest (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:

```shell
$ pip install biplist
```

4. Next, open Python and run the following command:

```python
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:

```python
['iPhoneOS']
```

For more information, see XCTest UI (p. 72).

**XCTEST_UI_TEST_PACKAGE_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 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:

```shell
$ 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:

```shell
$ 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:

```python
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 XCTest UI (p. 72).

Warning
---

If you see the following message, follow these steps to fix the issue.

```
Warning
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:

```python
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 XCTest UI (p. 72).

---

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 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:

```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
A valid XCTest UI package should produce output like the following:

```bash
com.apple.test.swift-sampleUITests-Runner
```

For more information, see XCTest UI (p. 72).

**XCTEST_UI_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 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:

```bash
$ 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:

```bash
$ tree .
```

You should find the `Info.plist` file inside an `.app` directory like `swift-sampleUITests-Runner.app` in our example:

```markdown`
`-- 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:

```bash
$ pip install biplist
```

4. Next, open Python and run the following command:

```python
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 XCTest UI (p. 72).

**XCTest UI_TEST_PACKAGE_TEST_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 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 XCTest UI (p. 72).
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 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:

   ```bash
   # 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:

   ```bash
   # 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:

   ```bash
   # pip install biplist
   ```

4. Next, open Python and run the following command:

   ```python
   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 **XCTest UI (p. 72)**.
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 (p. 187)
- Compliance Validation for AWS Device Farm (p. 199)
- Data Protection in AWS Device Farm (p. 199)
- Resilience in AWS Device Farm (p. 202)
- Infrastructure Security in AWS Device Farm (p. 202)
- Configuration Vulnerability Analysis and Management in Device Farm (p. 203)
- Incident Response in Device Farm (p. 203)
- Logging and Monitoring in Device Farm (p. 204)
- Security Best Practices for Device Farm (p. 204)

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 (p. 197).

**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 employees 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 (p. 190).

**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 (p. 194).

### Authenticating with Identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see Signing in to the AWS Management Console as an IAM user or root user in the [IAM User Guide](#). You must be authenticated (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly. To sign in directly to the AWS Management Console, use your password with your root user email address or your IAM user name. You can access AWS programmatically using your root user or IAM users access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don’t use AWS tools, you must sign the request yourself. Do this using Signature Version 4, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 signing process in the [AWS General Reference](#).

Regardless of the authentication method that you use, you might also 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 Using multi-factor authentication (MFA) in AWS in the [IAM User Guide](#).

**AWS account root user**

When you first create an AWS account, you begin with a single 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 do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

**IAM Users and Groups**

An IAM user is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see Managing access keys for IAM users in the IAM User Guide. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.
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 **When to create an IAM user (instead of a role)** 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. You can temporarily assume an IAM role in the AWS Management Console by **switching roles**. 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 **Using IAM roles** in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as **federated users**. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see **Federated users and roles** in the *IAM User Guide*.
- **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 **How IAM roles differ from resource-based policies** 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 is 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.
- **Principal permissions** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. Policies grant permissions to a principal. When you use some services, you might perform an action that then triggers another action in a different service. In this case, you must have permissions to perform both actions. To see whether an action requires additional dependent actions in a policy, see **Actions, Resources, and Condition Keys for AWS Device Farm** in the *Service Authorization Reference*.
- **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 **Creating 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 IAM 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 **Using an IAM role to grant permissions to applications running on Amazon EC2 instances** in the *IAM User Guide*.
To learn whether to use IAM roles or IAM users, see When to create an IAM role (instead of a user) 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 (p. 190)
- Device Farm Resource-Based Policies (p. 192)
- Access Control Lists (p. 192)
- Authorization Based on Device Farm Tags (p. 192)
- Device Farm IAM Roles (p. 192)

## 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:

```json
"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:

```json
"Action": "devicefarm:List*"
```

To see a list of Device Farm actions, see Actions Defined by AWS Device Farm in the IAM User Guide.
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 Resources Defined by AWS Device Farm in the IAM User Guide. To learn with which actions you can specify the ARN of each resource, see Actions Defined by AWS Device Farm.

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 User Guide. To learn with which actions and resources you can use a condition key, see Actions Defined by AWS Device Farm.

Examples

To view examples of Device Farm identity-based policies, see AWS Device Farm Identity-Based Policy Examples (p. 194).

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 (p. 50).

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 (p. 197).

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 or cross-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 IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. You can sign in as the root user or an IAM user, or you can assume an IAM role. When you then make a request, AWS evaluates the related identity-based or resource-based policies. 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.

Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

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 Creating IAM 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 Choosing between managed policies and inline policies in the [IAM User Guide](#).

The following table outlines the Device Farm AWS managed policies.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>AWSDeviceFarmFullAccess</td>
<td>Provides full access to all AWS Device Farm operations.</td>
<td>July 15, 2015</td>
</tr>
</tbody>
</table>
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 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 How SCPs work 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 (p. 195)
- Allow Users to View Their Own Permissions (p. 195)
- Accessing One Device Farm Desktop Browser Testing Project (p. 196)
- Viewing Device Farm Desktop Browser Testing Projects Based on Tags (p. 197)
Policy Best Practices

Identity-based policies are very powerful. They 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 using AWS managed policies** – To start using Device Farm quickly, use AWS managed policies to give your employees the permissions they need. These policies are already available in your account and are maintained and updated by AWS. For more information, see Get started using permissions with AWS managed policies in the IAM User Guide.

- **Grant least privilege** – When you create custom policies, grant only the permissions required to perform a task. Start with a minimum set of permissions and grant additional permissions as necessary. Doing so is more secure than starting with permissions that are too lenient and then trying to tighten them later. For more information, see Grant least privilege in the IAM User Guide.

- **Enable MFA for sensitive operations** – For extra security, require IAM users to use multi-factor authentication (MFA) to access sensitive resources or API operations. For more information, see Using multi-factor authentication (MFA) in AWS in the IAM User Guide.

- **Use policy conditions for extra security** – To the extent that it's practical, define the conditions under which your identity-based policies allow access to a resource. For example, you can write conditions to specify a range of allowable IP addresses that a request must come from. You can also write conditions to allow requests only within a specified date or time range, or to require the use of SSL or MFA. For more information, see IAM JSON policy elements: Condition 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.

```json
{
  "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": "*"
    }
  ]
}
```

API Version 2015-06-23
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 desktop 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.

```json
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid":"GetTestGridProject",
      "Effect":"Allow",
      "Action": [ "devicefarm:GetTestGridProject" ],
    },
    {
      "Sid":"ViewProjectInfo",
      "Effect":"Allow",
      "Action": [ "devicefarm:ListTestGridSessions", "devicefarm:ListTestGridSessionActions", "devicefarm:ListTestGridSessionArtifacts" ],
    }
  ]
}
```

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:

```json
{
  "Version":"2012-10-17",
  "Statement": [ 
    {
      "$id":"scheduleTestRuns",
      "Effect":"Allow",
      "Action": [ "devicefarm:CreateUpload","devicefarm:ScheduleRun" ],
    }
  ]
}
```
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.
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, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password. Ask that person to update your policies 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 granted by a service role. Mary does not have permissions to pass the role to the service.

In this case, Mary asks her administrator to update her policies to allow her to perform the `iam:PassRole` action.

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/bPxRfiCYEXAMPIEKEY`). 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 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 create an IAM entity (user or role) for the person or application that needs access. They will use the credentials for that entity to access AWS. You must then attach a policy to the entity that grants them the correct permissions in Device Farm.

To get started right away, see Creating your first IAM delegated user and group 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 (p. 190).
- 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 How IAM roles differ from resource-based policies 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. This content includes 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 user accounts with 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 recommend TLS 1.2 or later.
- Set up API and user activity logging with AWS CloudTrail.
- 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 personal data that is stored in Amazon S3.
- If you require FIPS 140-2 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) 140-2.

We strongly recommend that you never put confidential or sensitive information, such as your customers’ email addresses, into tags or free-form 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 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 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 backing storage, but any metadata (ARNs, upload dates, file names, and so on) is preserved for future use. The following table lists the retention period for various content types.

<table>
<thead>
<tr>
<th>Content type</th>
<th>Retention period (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uploaded applications</td>
<td>30</td>
</tr>
<tr>
<td>Uploaded test packages</td>
<td>30</td>
</tr>
</tbody>
</table>
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.

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.

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 Working with Private Devices in AWS Device Farm (p. 89).

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 procedures that are described in the Amazon Web Services: Overview of Security Processes whitepaper.

You use AWS published API calls to access Device Farm through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

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 \texttt{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 \textit{rooting} or \textit{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 (p. 106).

**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 Working with Custom Test Environments (p. 76).
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.
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

• AWS Device Farm Integration with Jenkins CI Plugin (p. 206)
• AWS Device Farm Gradle Plugin (p. 209)

AWS Device Farm Integration with Jenkins CI Plugin

This 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: Install the Plugin (p. 206).

This section contains a series of procedures to set up and use the Jenkins CI plugin with AWS Device Farm.

Topics

• Step 1: Installing the Plugin (p. 206)
• Step 2: Creating an AWS Identity and Access Management User for your Jenkins CI Plugin (p. 207)
• Step 3: First-time configuration instructions (p. 208)
• Step 4: Using the Plugin in a Jenkins Job (p. 208)
• Dependencies (p. 209)

The following images show the features of the Jenkins CI plugin.

The plugin can also pull down all the test artifacts (logs, screenshots, etc.) locally:

Step 1: Installing the Plugin

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

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

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:

```json
{
   "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**

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: First-time configuration instructions**

The first time you run your Jenkins server, you will need to configure the system as follows.

**Note**

If you are using device slots (p. 8), 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 **Step 2: Create an IAM User** (p. 207) 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.

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.

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
To download the Gradle plugin, go to GitHub and follow the instructions in Building the Device Farm Gradle Plugin (p. 209).

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
- Step 1: Building the AWS Device Farm Gradle Plugin (p. 209)
- Step 2: Setting up the AWS Device Farm Gradle Plugin (p. 210)
- Step 3: Generating an IAM User (p. 211)
- Step 4: Configuring Test Types (p. 212)
- Dependencies (p. 214)

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 (p. 210).

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.
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 (p. 209).

To configure the AWS Device Farm Gradle Plugin

1. Add the plugin artifact to your dependency list in build.gradle.

```groovy
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:

```groovy
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 "wJalrXUtznFEMI/K7MDENG/bFxRfjCExAMPLEKEY"
        // OR
    }
}
```
Generating an IAM user

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 (p. 211)

Step 3: Generating an IAM User

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.

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

15. Choose Apply Policy.

Next step: Configuring Test Types (p. 212).

For more information, see Creating an IAM User (AWS Management Console) or Setting Up (p. 3).

Step 4: Configuring Test Types

By default, the AWS Device Farm Gradle plugin runs the Working with Instrumentation for Android and AWS Device Farm (p. 66) 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 Working with Test Types in AWS Device Farm (p. 52).

If you haven't done so already, complete steps 1 – 3 before configuring test types.

Note
If you are using device slots (p. 8), the device slots feature is disabled by default.

Appium

Device Farm provides support for Appium Java JUnit and TestNG for Android.

- Appium (Java (JUnit)) (p. 53)
• **Appium (Java (TestNG)) (p. 53)**

You can choose `useTestNG()` or `useJUnit()`. JUnit is the default and does not need to be explicitly specified.

```java
appium {
  tests file("path to zip file") // required
  useTestNG() // or useJUnit()
}
```

**Built-in: Explorer**

Device Farm provides a built-in app explorer to test user flows through your app without writing custom test scripts. You can specify a user name and password to test scenarios that require a login. Here is how you configure user name and password:

```java
appexplorer {
  username "my-username"
  password "my-password"
}
```

For more information, see **Built-in: Explorer (Android) (p. 73).**

**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.

```java
fuzz {
  eventThrottle 50 // optional default
  eventCount 6000 // optional default
  randomizerSeed 1234 // optional default blank
}
```

For more information, see **Built-in: Fuzz (Android and iOS) (p. 74).**

**Calabash**

Device Farm provides support for Calabash for Android. For information about how to prepare your Android Calabash tests, see **Android tests with Calabash (p. 64).**

```java
calabash {
  tests file("path to zip file") // required
  tags "my tags" // optional calabash tags
  profile "my profile" // optional calabash profile
}
```

**Instrumentation**

Device Farm provides support for instrumentation (JUnit, Espresso, Robotium, or any instrumentation-based tests) for Android. For more information, see **Working with Instrumentation for Android and AWS Device Farm (p. 66).**
When running an instrumentation test in Gradle, Device Farm uses the .apk file generated from your androidTest directory as the source of your tests.

```gradle
instrumentation {
    filter "test filter per developer docs" // optional
}
```

### UI Automator

Upload your app and UI Automator-based tests in a .jar file.

```gradle
uiautomator {
    tests file("path to uiautomator jar file") // required
    filter "test filter per developer docs" // optional
}
```

For more information, see UI Automator (p. 67).

### 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
# Document History

The following table describes the important changes to the documentation since the last release of this guide.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Date Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polaris UI updates</td>
<td>The Device Farm console now supports the Polaris framework.</td>
<td>July 28, 2021</td>
</tr>
<tr>
<td>Python 3 support</td>
<td>Device Farm now supports Python 3 in custom mode tests. Learn more about using Python 3 in your test packages:</td>
<td>April 20, 2020</td>
</tr>
<tr>
<td></td>
<td>• Appium (Python) (p. 53)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Appium (Python) (p. 53)</td>
<td></td>
</tr>
<tr>
<td>New security information and information on</td>
<td>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 (p. 187)</td>
<td>March 27, 2020</td>
</tr>
<tr>
<td>tagging AWS resources.</td>
<td>A new section on tagging in Device Farm has been added. To learn more about tagging, see Tagging in Device Farm (p. 50).</td>
<td></td>
</tr>
<tr>
<td>Removal of Direct Device Access.</td>
<td>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.</td>
<td>September 9, 2019</td>
</tr>
<tr>
<td>Update Gradle plugin configuration</td>
<td>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 (p. 210).</td>
<td>August 16, 2019</td>
</tr>
<tr>
<td>New requirement for test runs with XCTest</td>
<td>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” (p. 70).</td>
<td>February 4, 2019</td>
</tr>
<tr>
<td>Support for Appium Node.js and Appium Ruby</td>
<td>You can now run your tests in both Appium Node.js and Appium Ruby custom test environments. Learn more about Working with Test Types in AWS Device Farm (p. 52).</td>
<td>January 10, 2019</td>
</tr>
<tr>
<td>test types in custom environments</td>
<td>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 Working with Test Types in AWS Device Farm (p. 52).</td>
<td>October 2, 2018</td>
</tr>
<tr>
<td>Custom test environments</td>
<td>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</td>
<td>August, 16 2018</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date Changed</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Support for using Device Farm as an AWS CodePipeline test provider</td>
<td>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 Using AWS Device Farm in an CodePipeline Test Stage (p. 109).</td>
<td>July 19, 2018</td>
</tr>
<tr>
<td>Support for Private Devices</td>
<td>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 Working with Private Devices in AWS Device Farm (p. 89).</td>
<td>May 2, 2018</td>
</tr>
<tr>
<td>Support for Appium 1.6.3</td>
<td>You can now set the Appium version for your Appium custom tests.</td>
<td>March 21, 2017</td>
</tr>
<tr>
<td>Set the execution timeout for test runs</td>
<td>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 (p. 32).</td>
<td>February 9, 2017</td>
</tr>
<tr>
<td>Network Shaping</td>
<td>You can now simulate network connections and conditions for a test run. Learn more about Simulating Network Connections and Conditions (p. 33).</td>
<td>December 8, 2016</td>
</tr>
<tr>
<td>New Troubleshooting Section</td>
<td>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 (p. 118).</td>
<td>August 10, 2016</td>
</tr>
<tr>
<td>Remote Access Sessions</td>
<td>You can now remotely access and interact with a single device in the console. Learn more about Working with Remote Access (p. 85).</td>
<td>April 19, 2016</td>
</tr>
<tr>
<td>Device Slots Self-Service</td>
<td>You can now purchase device slots using the AWS Management Console, the AWS Command Line Interface, or the API. Learn more about how to Purchase a device slot in Device Farm (p. 8).</td>
<td>March 22, 2016</td>
</tr>
<tr>
<td>How to stop test runs</td>
<td>You can now stop test runs using the AWS Management Console, the AWS Command Line Interface, or the API. Learn more about how to Stop a Run in AWS Device Farm (p. 35).</td>
<td>March 22, 2016</td>
</tr>
<tr>
<td>New XCTest UI test types</td>
<td>You can now run XCTest UI custom tests on iOS applications. Learn more about the XCTest UI (p. 72) test type.</td>
<td>March 8, 2016</td>
</tr>
<tr>
<td>New Appium Python test types</td>
<td>You can now run Appium Python custom tests on Android, iOS, and web applications. Learn more about Working with Test Types in AWS Device Farm (p. 52).</td>
<td>January 19, 2016</td>
</tr>
<tr>
<td>Web Application test types</td>
<td>You can now run Appium Java JUnit and TestNG custom tests on web applications. Learn more about Working with Web App Tests in AWS Device Farm (p. 73).</td>
<td>November 19, 2015</td>
</tr>
<tr>
<td>Change</td>
<td>Description</td>
<td>Date Changed</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>AWS Device Farm Gradle Plugin</td>
<td>Learn more about how to install and use the <a href="https://docs.aws.amazon.com/devicefarm/latest/userguide/gradle-plugin.html">Device Farm Gradle Plugin (p. 209)</a></td>
<td>September 28, 2015</td>
</tr>
<tr>
<td>New Android Built-in Test: Explorer</td>
<td>Learn more about <a href="https://docs.aws.amazon.com/devicefarm/latest/userguide/functional-testing-android.html">Built-in: Explorer (Android) (p. 73)</a>. The explorer test crawls your app by analyzing each screen as if it were an end user and takes screenshots as it explores.</td>
<td>September 16, 2015</td>
</tr>
<tr>
<td>iOS support added</td>
<td>Learn more about testing iOS devices and running iOS tests (including XCTest) in <a href="https://docs.aws.amazon.com/devicefarm/latest/userguide/ios-testing.html">Working with Test Types in AWS Device Farm (p. 52)</a></td>
<td>August 4, 2015</td>
</tr>
<tr>
<td>Initial public release</td>
<td>This is the initial public release of the <a href="https://docs.aws.amazon.com/devicefarm/latest/userguide">AWS Device Farm Developer Guide</a>.</td>
<td>July 13, 2015</td>
</tr>
</tbody>
</table>
AWS glossary

For the latest AWS terminology, see the AWS glossary in the AWS General Reference.