## Table of Contents

What Is AWS Device Farm? ........................................................................................................ 1
Automated App Testing ........................................................................................................... 1
  Supported Test Types and Built-in Tests ................................................................. 1
Remote Access Interaction ............................................................................................... 2
Terminology ..................................................................................................................... 2
Setting Up ....................................................................................................................... 3
  Step 1: Sign Up for AWS .......................................................................................... 4
  Step 2: Create or Use an IAM User in Your AWS Account .......................................... 4
  Step 3: Give the IAM User Permission to Access Device Farm ...................................... 4
  Next Step ................................................................................................................... 5
Getting Started ................................................................................................................. 6
  Prerequisites ............................................................................................................... 6
  Step 1: Sign in to the Console .................................................................................. 6
  Step 2: Create a Project ......................................................................................... 6
  Step 3: Create and Start a Run .......................................................... .............................. 7
  Step 4: View the Run’s Results ................................................................................ 8
  Next Steps ............................................................................................................... 8
Concepts ............................................................................................................................. 9
  Devices ....................................................................................................................... 9
    Supported Devices ................................................................................................... 9
    Device Pools ........................................................................................................... 9
    Private Devices ......................................................................................................... 9
    Device Branding ......................................................................................................... 9
    Device Slots ............................................................................................................... 10
    Pre-Installed Device Apps ....................................................................................... 10
    Device Capabilities .................................................................................................. 10
Test Types ....................................................................................................................... 10
  Android Test Types .................................................................................................... 10
  iOS Test Types ........................................................................................................... 11
  Web Application Test Types ..................................................................................... 11
Runs ..................................................................................................................................... 11
  Run Configuration ..................................................................................................... 11
  Run Files Retention ................................................................................................... 12
  Run Device State ........................................................................................................ 12
  Parallel Runs .............................................................................................................. 12
  Setting the execution timeout in test runs ................................................................. 12
  Instrumenting Apps .................................................................................................... 12
  Re-Signing Apps in Runs ....................................................................................... 12
  Obfuscated Apps in Runs ....................................................................................... 12
  Ads in Runs ............................................................................................................... 12
  Media in Runs ........................................................................................................... 13
  Common Tasks for Runs ......................................................................................... 13
Reports ............................................................................................................................. 13
  Report Retention ....................................................................................................... 13
  Report Components ................................................................................................. 13
  Performance Samples in Reports ................................................................................ 13
  Logs in Reports ........................................................................................................... 13
  Common Tasks for Reports ...................................................................................... 13
Sessions ............................................................................................................................ 14
  Supported Devices for Remote Access ..................................................................... 14
  Session Files Retention ............................................................................................ 14
  Instrumenting Apps .................................................................................................. 14
  Re-Signing Apps in Sessions ..................................................................................... 14
<table>
<thead>
<tr>
<th>Working with Test Types</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build-in Test Types</td>
<td>42</td>
</tr>
<tr>
<td>Custom Test Types</td>
<td>42</td>
</tr>
<tr>
<td>Custom Android Test Types</td>
<td>42</td>
</tr>
<tr>
<td>Custom iOS Test Types</td>
<td>42</td>
</tr>
<tr>
<td>Custom Web Application Test Types</td>
<td>42</td>
</tr>
<tr>
<td>Android Tests</td>
<td>43</td>
</tr>
<tr>
<td>Built-in Test Types for Android</td>
<td>43</td>
</tr>
<tr>
<td>Custom Test Types for Android</td>
<td>43</td>
</tr>
<tr>
<td>Appium Java JUnit</td>
<td>43</td>
</tr>
<tr>
<td>Appium Java TestNG</td>
<td>47</td>
</tr>
<tr>
<td>Appium Python</td>
<td>51</td>
</tr>
<tr>
<td>Calabash</td>
<td>54</td>
</tr>
<tr>
<td>Instrumentation</td>
<td>56</td>
</tr>
<tr>
<td>UI Automator</td>
<td>57</td>
</tr>
<tr>
<td>iOS Tests</td>
<td>59</td>
</tr>
<tr>
<td>Built-in Test Types for iOS</td>
<td>59</td>
</tr>
<tr>
<td>Custom Test Types</td>
<td>59</td>
</tr>
<tr>
<td>Appium Java JUnit</td>
<td>59</td>
</tr>
<tr>
<td>Appium Java TestNG</td>
<td>63</td>
</tr>
<tr>
<td>Appium Python</td>
<td>67</td>
</tr>
<tr>
<td>Calabash</td>
<td>70</td>
</tr>
<tr>
<td>UI Automation</td>
<td>72</td>
</tr>
<tr>
<td>XCTest</td>
<td>73</td>
</tr>
<tr>
<td>XCTest UI</td>
<td>74</td>
</tr>
<tr>
<td>Web App Tests</td>
<td>76</td>
</tr>
<tr>
<td>Rules for Metered and Unmetered Devices</td>
<td>76</td>
</tr>
<tr>
<td>Appium Java JUnit</td>
<td>76</td>
</tr>
<tr>
<td>Appium Java TestNG</td>
<td>78</td>
</tr>
<tr>
<td>Appium Python</td>
<td>80</td>
</tr>
<tr>
<td>Built-in Tests</td>
<td>84</td>
</tr>
<tr>
<td>Built-in Test Types</td>
<td>84</td>
</tr>
<tr>
<td>Built-in: Explorer (Android)</td>
<td>84</td>
</tr>
<tr>
<td>Built-in: Fuzz (Android and iOS)</td>
<td>85</td>
</tr>
<tr>
<td>Working with Remote Access</td>
<td>86</td>
</tr>
<tr>
<td>Create a Session</td>
<td>86</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>86</td>
</tr>
<tr>
<td>Create a Session with the Device Farm Console</td>
<td>86</td>
</tr>
<tr>
<td>Next Steps</td>
<td>87</td>
</tr>
<tr>
<td>Use a Session</td>
<td>87</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>87</td>
</tr>
<tr>
<td>Use a Session in the Device Farm Console</td>
<td>87</td>
</tr>
<tr>
<td>Next Steps</td>
<td>88</td>
</tr>
<tr>
<td>Tips and Tricks</td>
<td>88</td>
</tr>
<tr>
<td>Get Session Results</td>
<td>88</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>88</td>
</tr>
<tr>
<td>Viewing Session Details</td>
<td>89</td>
</tr>
<tr>
<td>Downloading Session Video or Logs</td>
<td>89</td>
</tr>
<tr>
<td>Working with Direct Device Access</td>
<td>90</td>
</tr>
<tr>
<td>Create a Remote Debugging Session</td>
<td>90</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>90</td>
</tr>
<tr>
<td>Replace Your Copy of ADB</td>
<td>91</td>
</tr>
<tr>
<td>Launch the Device Farm App and Start a Remote Debugging Session</td>
<td>94</td>
</tr>
<tr>
<td>How to Stop a Remote Debugging Session</td>
<td>100</td>
</tr>
<tr>
<td>Tips and Tricks</td>
<td>101</td>
</tr>
</tbody>
</table>
What Is AWS Device Farm?

Device Farm is an app testing service that enables you 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 available testing frameworks
- Remote access of devices onto which you can load, run, and interact with apps in real time

Automated App Testing

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

A test report containing high-level results, low-level logs, pixel-to-pixel screenshots, and performance data is updated as tests are completed.

Device Farm supports testing of native and hybrid Android, iOS, and Fire OS apps, including those created with PhoneGap, Titanium, Xamarin, Unity, and other frameworks. It supports remote access of Android apps for interactive testing.

Supported Test Types and Built-in Tests

Device Farm currently provides support for the following test types:

For Android:

- Appium Java JUnit (p. 43)
- Appium Java TestNG (p. 47)
- Appium Python (p. 51)
- Calabash (p. 54)
- Instrumentation (p. 56) (JUnit, Espresso, Robotium, or any instrumentation-based tests)
- UI Automator (p. 57)
- Explorer (p. 84)

For iOS:

- Appium Java JUnit (p. 59)
- Appium Java TestNG (p. 63)
- Appium Python (p. 67)
- Calabash (p. 70)
- UI Automation (p. 72)
- XCTest (p. 73) (including KIF)
- XCTest UI (p. 74)

For Web Apps:

- Appium Java JUnit (p. 76)
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 how to use or set up their device. They can also walk customers through how to use 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 for your review.

Initially, a limited number of Android and Fire OS devices are supported for remote access. However, the list of devices will grow during the beta period and as new devices enter the market.

Terminology

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

project
A logical workspace 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.

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 that contains information about the results of the run. A run contains one or more jobs. For more information, see Runs (p. 11).

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. 13).

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

meter
Metering refers to billing for devices, and you may encounter references to "metered devices" or "unmetered devices" in the documentation and API reference. For more information about pricing, see AWS Device Farm Pricing.

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

session
An interactive session with a single device in the console.

Setting Up
To get set up to use Device Farm, see Setting Up (p. 4).
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. 4)
- Step 2: Create or Use an IAM User in Your AWS Account (p. 4)
- Step 3: Give the IAM User Permission to Access Device Farm (p. 4)
- Next Step (p. 5)

**Step 1: Sign Up for AWS**

Sign up for Amazon Web Services (AWS).

If you do not have an AWS account, use the following procedure to create one.

**To sign up for AWS**

2. Follow the online instructions.

**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 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)*.

**Step 3: Give the IAM User Permission to Access Device Farm**

Give the IAM user permission to access Device Farm. To do this, 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**

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 **Provides access to all Device Farm actions associated with the IAM user**.
7. For **Policy Document**, type the following statement:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "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, choose **Add permissions**.
5. In the **Grant permissions** area, choose **Attach existing policies directly**.
6. Select the policy you just created (for example, **AWSDeviceFarmAccessPolicy**).
7. Choose **Next: Review**.
8. In the **Permissions summary** area, choose **Add permissions**.

**Note**

Attaching the policy provides the IAM user with access to all Device Farm actions associated with that IAM user. To learn how to restrict IAM users to a limited set of Device Farm actions, see Access Permissions Reference (p. 197).

**Next Step**

You are now ready to start using Device Farm. See **Getting Started** (p. 6).
Getting Started with AWS Device Farm

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

Topics
• Prerequisites (p. 6)
• Step 1: Sign in to the Console (p. 6)
• Step 2: Create a Project (p. 6)
• Step 3: Create and Start a Run (p. 7)
• Step 4: View the Run's Results (p. 8)
• Next Steps (p. 8)

Prerequisites

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

• Complete the steps in Setting Up (p. 4), which include signing up for an AWS account, creating or using an IAM user in the AWS account, and giving the IAM user permission to access Device Farm.

• For Android, you will need an .apk (Android app package) file, and for iOS you will need an .ipa (iOS app archive) file, which you will upload 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.

• Optionally, you will need a test from one of the test types supported by Device Farm. You will upload this test package to Device Farm, and then run the test later in this walkthrough. (If you do not 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. 42).

Step 1: Sign in to the Console

You can use the Device Farm console to create and manage projects and runs for testing. You will 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.

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.

1. In the Device Farm console page, type a name for your project (for example, **MyDemoProject**).
   
   **Note**
   
   If you type a project name other than **MyDemoProject**, be sure to use it throughout this walkthrough.

2. Choose **Create project**. Refresh the page to see your new project.

---

### Step 3: Create and Start a Run

Now that you have a project, you can create and then start a run.

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. 11)**.

1. Choose **MyDemoProject**.
2. On the **Automated tests** page, choose **Create a new run**.
3. On the **Choose your application** page, choose **Upload**.
4. Browse to and choose your Android or iOS app file. For Android, the file must be an .apk file. For iOS, the file must be an .ipa file built for a device, not the simulator.
5. Choose **Next step**.
6. On the **Configure a test** page, choose one of the test suites.
   
   **Note**
   
   If you do not have any tests available, choose **Built-in: Fuzz** to run a standard built-in test suite. For this walkthrough, if you choose **Built-in: Fuzz**, leave the **Event count**, **Event throttle**, and **Randomizer seed** boxes at their default values.
7. If you did not choose **Built-in: Fuzz**, then choose **Upload**, and browse to and choose the file that contains your tests.
8. Choose **Next step**.
9. On the **Select devices** page, for **Device pool**, choose **Top Devices** to select the device pool, and then choose **Next step**.

   A device pool in Device Farm represents a collection of devices that typically share similar characteristics such as platform, manufacturer, or model. For more information, see **Devices (p. 9)**.
10. On the **Specify device state** page, do any of the following:
   
   - To provide additional data for Device Farm to use during the run, next to **Add extra data**, choose **Upload**, and then browse to and choose the .zip file.
• To install an additional app for Device Farm to use during the run, next to Install other apps, choose Upload, and then browse to and choose the .apk or .ipa file that contains the app. Repeat for any additional apps you want to install. You can change the installation order by dragging and dropping the apps.

• To specify whether Wi-Fi, Bluetooth, GPS, or NFC will be 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, type the coordinates.

• To preset the device locale for the run, choose the locale in Device Locale.

• To preset the network profile for the run, choose a profile in Network profile or choose Create a new profile to create your own.

11. Choose Next step.

Device Farm should start the run as soon as devices are available, typically within a few minutes. Until the run starts, Device Farm will display a calendar icon. After the run starts, results will appear as tests are completed. During this time, Device Farm will display a progress icon.

Step 4: View the Run's Results

You'll know the run is complete when the progress icon changes to a result icon.

To view the run's results, choose the completed run in the Device Farm console. A summary page that includes the following information is displayed.

• The total number of tests, by outcome.
• Lists of tests with unique warnings or failures.
• A list of devices and test results for each.
• Any screenshots captured during the run, grouped by device.

For more information, see Analyze a Report (p. 35).

You have now completed this walkthrough.

Next Steps

To learn more about Device Farm, see Concepts (p. 9).
AWS Device Farm Concepts

This section describes important Device Farm concepts.

- Devices (p. 9)
- Test Types in AWS Device Farm (p. 10)
- Runs (p. 11)
- Reports (p. 13)
- Sessions (p. 14)

Device Support in AWS Device Farm

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

Topics

- Supported Devices (p. 9)
- Device Pools (p. 9)
- Private Devices (p. 9)
- Device Branding (p. 9)
- Device Slots (p. 10)
- Pre-Installed Device Apps (p. 10)
- Device Capabilities (p. 10)

Supported Devices

Device Farm provides support for hundreds of unique, popular Android, iOS, and Fire OS devices and operating system combinations. The list of available devices grows as new devices enter the market. The full list of devices can be found here: 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 that run 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. Once you choose to end your subscription, the hardware is removed from our environment. For more information about using private devices, see Private devices.

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, remote access device slots and automated testing device slots. An automated testing device slot is one on which you can run tests concurrently. A remote access device slot is one you can run in remote access sessions concurrently.

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

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.

Pre-Installed Device Apps

Devices in Device Farm include a small number of apps pre-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 may 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 Types in AWS Device Farm

AWS Device Farm provides many different built-in and custom test types for Android, iOS, and Web applications. Built-in tests enable you to test your apps without writing scripts. Custom tests allow you to test specific flows and business logic within your app. For more information, see Working with Test Types in AWS Device Farm (p. 42).

Android Test Types

Device Farm provides the following built-in and custom test types for Android devices.

- Built-in: Explorer (Android) (p. 84)
- Built-in: Fuzz (Android and iOS) (p. 85)
- Appium Java JUnit (p. 43)
- Appium Java TestNG (p. 47)
- Appium Python (p. 51)
- Calabash (p. 54)
- Instrumentation (p. 56)
- UI Automator (p. 57)
iOS Test Types

Device Farm provides the following built-in and custom test types for iOS devices.

- Built-in: Fuzz (Android and iOS) (p. 85)
- Appium Java JUnit (p. 59)
- Appium Java TestNG (p. 63)
- Appium Python (p. 67)
- Calabash (p. 70)
- UI Automation (p. 72)
- XCTest (p. 73)
- XCTest UI (p. 74)

Web Application Test Types

Device Farm provides the following custom test types for Web applications.

- Appium Java JUnit (p. 76)
- Appium Java TestNG (p. 78)
- Appium Python (p. 80)

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. 11)
- Run Files Retention (p. 12)
- Run Device State (p. 12)
- Parallel Runs (p. 12)
- Setting the execution timeout in test runs (p. 12)
- Instrumenting Apps (p. 12)
- Re-Signing Apps in Runs (p. 12)
- Obfuscated Apps in Runs (p. 12)
- Ads in Runs (p. 12)
- Media in Runs (p. 13)
- Common Tasks for Runs (p. 13)

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 that will 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 in test runs

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.

To learn more, see Set the Execution Timeout for Test Runs in AWS Device Farm (p. 26).

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 will add it to the app's package before Device Farm installs it, so that the auxiliary will exist 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 may break any functionality that depends on the app's signature, such as the Google Maps Android API, or it may 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 will not be able to 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 will be 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 (p. 23) and Working with Test Runs (p. 23).

Reports in AWS Device Farm

The following sections contain information about Device Farm reports.

A report in Device Farm contains information about a run, which is a request for Device Farm to test a single app against one or more devices.

Topics

- Report Retention (p. 13)
- Report Components (p. 13)
- Performance Samples in Reports (p. 13)
- Logs in Reports (p. 13)
- Common Tasks for Reports (p. 13)

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 both detailed per-device data as well as 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 Analyze a Report (p. 35).
Sessions in AWS Device Farm

You can use Device Farm to perform interactive testing of Android apps through remote access sessions in a web browser. This kind of interactive testing helps support engineers on a customer call to walk step by step through 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.

A session in Device Farm is a real-time interaction with an actual, physical device hosted in a web browser.

Topics

- Supported Devices for Remote Access (p. 14)
- Session Files Retention (p. 14)
- Instrumenting Apps (p. 14)
- Re-Signing Apps in Sessions (p. 14)
- Obfuscated Apps in Sessions (p. 14)

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 and will grow beyond the initial set during the beta period. The current list of Android and iOS devices available for remote access is displayed in the console. For more information about devices, see Devices (p. 9).

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 apps can be submitted unmodified.

Re-Signing Apps in Sessions

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

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 will not be able to re-sign the app.
Purchase a Device Slot in AWS Device Farm

To purchase a device slot, you can use the Device Farm console, the AWS CLI, or the Device Farm API.

**Topics**
- Purchase Device Slots with the Device Farm Console (p. 15)
- Purchase a Device Slot with the AWS CLI (p. 16)
- Purchase a Device Slot with the Device Farm API (p. 19)

## Purchase Device Slots with the Device Farm Console

2. Scroll down to the **UNLIMITED ACCESS** section and choose the **Learn more about unlimited testing** link to get to the **packages** page.

![Device Farm Console Packages](image)

3. On the **Packages** page, you can either choose one of the preconfigured packages (**SINGLE PLATFORM**, **CROSS PLATFORM**, or **CONTINUOUS INTEGRATION** package) or create your own **custom package** by choosing the number of slots of each type you wish to purchase.

   **Note**
   If you choose one of the preconfigured packages, you must check **Automated testing**, **Remote access**, or both.

   The text dynamically updates with the amount that will be added to your bill for each device slot purchased. For more information, see Device Farm Pricing.
4. Choose **Buy now** for the package you wish to purchase.
If you choose **Buy now**, you'll see a **Complete your purchase** dialog. Choose **Complete purchase** to complete your purchase.

Instead of **Buy now**, you may see **Contact us** or **Contact us to purchase**. This indicates that your account is not yet approved to purchase the number of device slots you have requested.

If you choose **Contact us** or **Contact us to purchase**, you'll see a **Send us feedback for the Device Farm Console** dialog. Tell us how many slots of each type you'd like to purchase and choose **Contact Support**.

Once you have successfully purchased device slots, you'll see the **Account settings** page.

In the **Account settings** page, you'll see the **You have xxx FREE TRIAL MINUTES remaining** message only if you have free trial minutes remaining. The number of minutes remaining is an estimate that doesn't reflect usage by tests that are currently running.

You'll also see the number of device slots that you have currently. If you have increased or decreased the number of slots, you'll also see the number of slots that you will have one month after the date you made the change.

---

**Purchase a Device Slot with the 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 you can purchase before you need to contact us and the number of remaining free trial minutes that you have, run the **get-account-settings** command. You will see output similar to the following:

```json
{
    "accountSettings": {
        "maxSlots": {
            "GUID": 1,
            "GUID": 1,
            "GUID": 1,
            "GUID": 1
        },
        "unmeteredRemoteAccessDevices": {
            "ANDROID": 0,
            "IOS": 0
        }
    }
}
```
To list the device slot offerings available to you, run the `list-offerings` command. You will see output similar to the following:

```json
{
   "offerings": [
      {
         "recurringCharges": [
            {
               "cost": {
                  "amount": 250.0,
                  "currencyCode": "USD"
               },
               "frequency": "MONTHLY"
            }
         ],
         "platform": "IOS",
         "type": "RECURRING",
         "id": "GUID",
         "description": "iOS Unmetered Device Slot"
      },
      {
         "recurringCharges": [
            {
               "cost": {
                  "amount": 250.0,
                  "currencyCode": "USD"
               },
               "frequency": "MONTHLY"
            }
         ],
         "platform": "ANDROID",
         "type": "RECURRING",
         "id": "GUID",
         "description": "Android Unmetered Device Slot"
      },
      {
         "recurringCharges": [
            {
               "cost": {
                  "amount": 250.0,
                  "currencyCode": "USD"
               },
               "frequency": "MONTHLY"
            }
         ],
         "platform": "ANDROID",
         "type": "RECURRING",
         "id": "GUID",
         "description": "Android Remote Access Unmetered Device Slot"
      }
   ]
}
```
To list offering promotions that are available, run the `list-offering-promotions` command.  

**Note**

This command returns only promotions that you have not yet purchased. As soon as you purchase one or more slots across any offering using a promotion, that promotion will no longer appear in the results.

You will see output similar to the following:

```json
{
  "offeringPromotions": [
    {
      "i": "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 will see output similar to the following:

```json
{
  "current": {
    "GUID": {
      "offering": {
        "platform": "IOS",
        "type": "RECURRING",
        "i": "GUID",
        "description": "iOS Unmetered Device Slot"
      },
      "quantity": 1
    },
    "GUID": {
      "offering": {
        "platform": "ANDROID",
        "type": "RECURRING",
        "i": "GUID",
        "description": "Android Unmetered Device Slot"
      },
      "quantity": 1
    }
  },
  "nextPeriod": {
    "GUID": {
      "effectiveOn": 1459468800.0,
      "description": "iOS Unmetered Device Slot"
    }
  }
}
```
Additional commands for this feature include `renew-offering` and `list-offering-transactions`. For more information about specific operations, see the AWS CLI reference for Device Farm.

For information about using Device Farm with the AWS CLI, see AWS CLI Reference (p. 124).

### Purchase a Device Slot with the Device Farm 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 have not yet purchased. As soon as you purchase one or more slots using an offering promotion, that promotion will no longer appear in the results.

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

Additional commands for this feature include `RenewOffering` and `ListOfferingTransactions`.

For information about using the Device Farm API, see API Reference (p. 126).
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 Device Farm console, the AWS CLI, or the Device Farm service API to work with projects.

- Create a Project (p. 20)
- View the Projects List (p. 21)

Create a Project in AWS Device Farm

To create a project, you can use the Device Farm console, the AWS CLI, or the Device Farm API.

Topics
- Prerequisites (p. 20)
- Create a Project with the Device Farm Console (p. 20)
- Create a Project with the AWS CLI (p. 21)
- Create a Project with the Device Farm API (p. 21)

Prerequisites

- Complete the steps in Setting Up (p. 4), which include signing up for an AWS account, creating or using an IAM user in the AWS account, and giving the IAM user permission to access Device Farm.

Create a Project with the Device Farm Console

1. Make sure you have set up an AWS account and an IAM user to access Device Farm.
3. In the Device Farm console page, type a name for your project.

Note
You can also specify project settings, including the default timeout for a test run. Once applied, these settings will apply to all test runs in a project. For more information, see Set the Execution Timeout for Test Runs in AWS Device Farm (p. 26).
Create a Project with the AWS CLI

1. Make sure you have set up an AWS account and an IAM user to access Device Farm.
2. Run the `create-project` command.

   **Note**
   For information about using Device Farm with the AWS CLI, see AWS CLI Reference (p. 124).

Create a Project with the Device Farm API

1. Make sure you have set up an AWS account and an IAM user to access Device Farm.
2. Call the `CreateProject` API.

   For information about using the Device Farm API, see API Reference (p. 126).

View the Projects List in AWS Device Farm

To view the list of available projects, you can use the Device Farm console, the AWS CLI, or the Device Farm API.

**Topics**

- Prerequisites (p. 21)
- View the Projects List with the Device Farm Console (p. 21)
- View the Projects List with the AWS CLI (p. 22)
- View the Projects List with the Device Farm API (p. 22)

Prerequisites

- Create at least one project in Device Farm. To create a project, follow the instructions in Create a Project (p. 20), and then return to this page.

View the Projects List with the Device Farm Console

1. Make sure that you have completed at least one project.
3. If the list of available projects is not displayed, then on the secondary navigation bar, do one of the following:
   - Choose the Device Farm console home button.
   - For Projects, choose View all projects.

View the Projects List with the AWS CLI

1. Make sure that you have completed at least one project.
2. To view the projects list, run the list-projects command.

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

Note
For information about using Device Farm with the AWS CLI, see AWS CLI Reference (p. 124).

View the Projects List with the Device Farm API

1. Make sure that you have completed at least one project.
2. To view the projects list, call the ListProjects API.

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

For information about the Device Farm API, see API Reference (p. 126).
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. 11).

You can use the Device Farm console, the AWS CLI, or the Device Farm service API to work with runs.

Topics

• Create a Run in AWS Device Farm (p. 23)
• Set the Execution Timeout for Test Runs in AWS Device Farm (p. 26)
• Simulate Network Connections and Conditions for your AWS Device Farm Runs (p. 28)
• Stop a Run in AWS Device Farm (p. 30)
• View a Runs List in AWS Device Farm (p. 33)
• Create a Device Pool in AWS Device Farm (p. 34)
• Analyze a Report in AWS Device Farm (p. 35)

Create a Run in AWS Device Farm

To create a run, you can use the Device Farm console, the AWS CLI, or the Device Farm API.

For information about runs, see Runs (p. 11).

Topics

• Prerequisites (p. 23)
• Create a Test Run with the Device Farm Console (p. 23)
• Create a Run with the AWS CLI (p. 25)
• Create a Run with the Device Farm API (p. 25)
• Next Steps (p. 26)

Prerequisites

• Create a project in Device Farm. Follow the instructions in Create a Project (p. 20), and then return to this page.

Create a Test Run with the Device Farm Console

2. If you see the AWS Device Farm console home page, type a name for your project and choose Create project. Refresh the page to see your new project.
3. If you already have a project, you can upload your tests to an existing project.
   Otherwise, choose Create a new project and specify a name to create your project.
4. Open your project, and then choose Create a new run.
5. On the Choose your application page, choose either Native application (the Android and Apple button) or Web application (the HTML5 button).
6. Upload your application file. You can also drag and drop your file or choose a recent upload.
   If you are uploading an iOS app, be sure to build for iOS device, as opposed to a simulator.
7. Optionally, you can provide a Run name.
   If you don't specify a Run name, Device Farm uses the app filename as the run name.
8. Choose Next step.
9. On the Configure a test page, choose one of the available test suites.
   Note
   If you do not have any tests available, then choose Built-in: Fuzz to run a standard test suite that is built-in to Device Farm. If you choose Built-in: Fuzz, and the Event count, Event throttle, and Randomizer seed boxes appear, you can change or leave the values as desired.
   For information about the available test suites, see Working with Test Types in AWS Device Farm (p. 42).
10. If you did not choose Built-in: Fuzz, then choose Upload, and browse to and choose the file that contains your tests.
11. Choose Next step.
12. 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. 34), 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. 9).
13. Choose Next step.
14. On the Specify device state page, do none, some, or all of the following:
   • To provide any additional data that Device Farm will use during the run, choose Upload next to Add extra data, and then browse to and choose the .zip file that contains the additional data.
   • To install an additional app that Device Farm will use during the run, choose Upload next to Install other apps, and then browse to and choose the .apk, or .ipa, file that contains the app. Repeat this for any additional apps that you want to install. You can change the apps' installation order by dragging and dropping the apps after you upload them.
   • To specify whether Wi-Fi, Bluetooth, GPS, or NFC will be 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, type the coordinates.
   • To preset the device locale for the run, choose the locale in Device Locale.
15. Choose Review and start run.
16. On this page, you can specify the execution timeout for your test run.
17. Change the execution timeout by typing a value or using the slider bar. For more information, see Set the Execution Timeout for Test Runs in AWS Device Farm (p. 26).

18. Choose **Confirm and start run**.

Device Farm will start the run as soon as devices are available, typically within a few minutes. Until the run starts, Device Farm will display a calendar icon. After the run starts, results will appear as tests are completed. During this time, Device Farm will display a progress icon.

**Note**
If you need to stop the test run, see Stop a Run in AWS Device Farm (p. 30).

### Create a Run with the AWS CLI

For a tutorial on using the AWS CLI to create a test run, see this AWS Mobile blog post.

1. Make sure that you have created a project. For more information, see Create a Project with the AWS CLI (p. 21).
2. Upload your application file by running the `create-upload` command.
3. Upload your tests by running the `create-upload` command.
4. Make sure that you have created a device pool. For more information, see Create a Device Pool with the AWS CLI (p. 35).
5. Schedule a test run by running the `schedule-run` command.

**Note**
For information about using Device Farm with the AWS CLI, see AWS CLI Reference (p. 124).

### Create a Run with the Device Farm API

1. Make sure that you have created a project. For more information, see Create a Project with the Device Farm API (p. 21).
2. Call the `ScheduleRun` API.

For information about using the Device Farm API, see API Reference (p. 126).
Next Steps

You’ll know the run is complete when the progress icon changes to a result icon. A report corresponding to the run will appear as soon as tests are complete. For more information, see Reports (p. 13).

To use the report, follow the instructions in Analyze a Report (p. 35).

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 60 minutes per device, but you can set a value as low as 5 minutes using the console, the AWS Command Line Interface, or the API. If you need to set a duration value longer than 60 minutes, contact us directly to set that up.

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

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

You may 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 where no tests are executing. In other words, using the execution timeout feature lets you stop that run if the test run is taking longer than expected.

You can set the execution timeout in two places: at the project level and at the test run level. The following procedures show you how to set up both using the Device Farm console.

**Prerequisites**

1. Complete the steps in Setting Up (p. 4), which include signing up for an AWS account, creating or using an IAM user in the AWS account, and giving the IAM user permission to access Device Farm.
2. Create a project in Device Farm. Follow the instructions in Create a Project (p. 20), and then return to this page.

**Set the Execution Timeout for a Project**

2. If you have a project already, choose that project from the Device Farm page.
   Otherwise, choose Create project and specify a name.
3. Choose Project settings.
4. Choose the General tab of your project.
5. Change the execution timeout by typing a value or using the slider bar.
6. Choose **Save changes**.

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

## Set the Execution Timeout for a Test Run

2. If you have a project already, choose that project from the **Device Farm** page.

   Otherwise, choose **Create project** and specify a name.
3. Choose **Create a new run**.
4. Follow the steps to choose an application, configure your test, select your devices, and specify a device state.
5. When you get to **Review and start run**, you can specify the execution timeout for your test run.
6. Change the execution timeout by typing a value or using the slider bar.
7. Choose Confirm and start run.

Simulate Network Connections and Conditions for your AWS Device Farm Runs

You can simulate network connections and conditions while testing your Android, iOS, FireOS, and web apps using network shaping in Device Farm. For example, you may want to test your app behavior and performance in different customer environments.

When you create a run using the default network settings, each device has a full, unhindered WiFi connection with Internet connectivity. When you use network shaping, you can change the WiFi 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. 28)
- Create your own Network Profile (p. 29)
- Change Network Conditions During your Test (p. 30)

**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 any projects yet, see Create a Project (p. 20).
2. Choose your application, and then choose Next step.
3. Configure your test, and then choose Next step.
4. Select your devices, and then choose Next step.
5. Choose a **Network profile** or choose **Create a new network profile** to create your own.

![Network profile selection](image)

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

### Create your own Network Profile

When you create a test run, you can choose to create a new network profile.

1. Choose **Create a new network profile**.

   The **Create a new network profile** dialog box appears.

![Create a new network profile dialog](image)
2. Specify the name and settings for your network profile.
3. Choose **Save network profile**.
4. Finish creating your test run and start the run.

Once created, you’ll be able to see and manage your network profiles on the **Project settings** page.

![Network profiles](image)

### Change Network Conditions During your Test

You can also simulate dynamic network conditions during your test execution. For example, you may simulate a dropped connection or fluctuating network types. You can do this by utilizing an API from the device host using a framework like Appium or Calabash.

We are still working on the API that simulates these conditions during your test execution, but you can try them out and let us know about your experience. For more information about this API, please contact us.

### Stop a Run in AWS Device Farm

You may want to stop a run after you have started it. For example, you may notice an issue while your tests are running and wish to restart the run with an updated test script. This topic describes how to stop a run and what the implications are for billing.

To stop a run, you can use the Device Farm console, the AWS CLI, or the Device Farm API.

For information about runs, see [Runs](p. 11).

#### Topics

- Prerequisites (p. 30)
- Stop a Run with the Device Farm Console (p. 30)
- Stop a Run with the AWS CLI (p. 31)
- Stop a Run with the Device Farm API (p. 33)

#### Prerequisites

- To stop a test run, you must have a test run created and actively running. For more information, see [Create a Test Run](p. 23).

### Stop a Run with the Device Farm Console

2. From the Device Farm console home page, choose the project where you have an active test run.

3. On the Run results page, choose the test run.

   Your screen should look like the following image (with the pending or in-progress icon to the left of the device name).

   ![DeviceFarm Run Results](image)

4. Choose **Stop run**.

   The button changes to **Stopping**, and after a short time the icon next to the device name also changes to the **Stopping** icon (a pulsing orange circle with a square inside it). When completely finished, the icon changes to an orange square.

   **Important**

   If a test has already finished, Device Farm cannot stop it. If a test is in progress, Device Farm will stop the test and you will see the total minutes for which you will be billed in the Devices section. In addition, you will still 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.

   ![DeviceFarm Devices Section](image)

---

**Stop a Run with the 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 will see output similar to the following:
To get the ARN of your run, use the `list-runs` command. The output will be similar to the following:

```json
{
   "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
         }
      }
   ]
}
```
**Stop a Run with the Device Farm API**

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

For information about using the Device Farm API, see API Reference (p. 126).

**View a Runs List in AWS Device Farm**

To view a list of available runs for a project, you can use the Device Farm console, the AWS CLI, or the Device Farm API.

**Topics**
- Prerequisites (p. 33)
- View a Runs List with the Device Farm Console (p. 33)
- View a Runs List with the AWS CLI (p. 33)
- View a Runs List with the Device Farm API (p. 34)

**Prerequisites**
- Create at least one run in Device Farm. To create a run, follow the instructions in Create a Test Run (p. 23), and then return to this page.

**View a Runs List with the Device Farm Console**

1. Make sure that you complete the prerequisites (p. 33), including the creation of at least one run.
3. In the list of projects, choose the project that corresponds to the runs list you want to view.

   **Tip**
   - If the list of available projects is not displayed, then on the secondary navigation bar, do one of the following:
     - Choose the Device Farm console home button, and then choose the project.
     - For Projects, choose View all projects, and then choose the project.

**View a Runs List with the AWS CLI**

1. Make sure that you have completed the prerequisites (p. 33), including the creation of at least one run.
2. To view a list of runs, run the `list-runs` command.

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

For general information about using Device Farm with the AWS CLI, see AWS CLI Reference (p. 124).
View a Runs List with the Device Farm API

1. Make sure that you have completed the prerequisites (p. 33), including the creation at least one run.
2. To view a list of runs, call the ListRuns API.
   
   Tip
   To view information about a single run, call the GetRun API.

For general information about the Device Farm API, see API Reference (p. 126).

Create a Device Pool in AWS Device Farm

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

Topics
- Prerequisites (p. 34)
- Create a Device Pool with the Device Farm Console (p. 34)
- Create a Device Pool with the AWS CLI (p. 35)
- Create a Device Pool with the Device Farm API (p. 35)

Prerequisites

- Start by creating a run in the Device Farm console. Follow the instructions in Create a Test Run (p. 23). When you get to the Select devices page in the Create a new run wizard, continue with the instructions in this section.

Create a Device Pool with the Device Farm Console

1. Make sure you have started to create a run and stopped on the Select devices page in the Create a new run wizard.
2. On the Select devices page, choose Create new device pool.
3. For Name, type a name that will make this device pool easy to identify in the future.
4. For Description, type a description that will make this device pool easy to identify in the future.
5. If you want to use one or more selection criteria for the devices in this device pool, do the following:
   1. Choose Add rule.
   2. For Field, choose one of the following:
      - Choose Manufacturer to include devices by their manufacturer name.
      - Choose Type to include devices by their Type value.
   3. For Operator, choose the following:
      - Choose EQUALS to include devices where the Field value equals the Operand value.
   4. For Operand, type or choose the value you want to specify for the Field and Operator values. Note that if you choose Platform for Field, then the only available selections are ANDROID and IOS. Similarly, if you choose Type for Field, then the only available selections are PHONE and TABLET.
5. To add another rule, choose Add rule again.
6. To delete a rule, choose the X icon next to the rule you want to delete.

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

6. If you want to manually include or exclude individual devices, select or clear the box next to each device.
   
   **Note**
   You can select or clear the boxes only if you do not have any rules specified.

7. 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 will be the only ones included or excluded. To confirm which devices will be included or excluded, be sure to clear the contents of all of the boxes in the column header row. Then browse the boxes to see which devices will be included or excluded.

8. Choose Save device pool.

**Create a Device Pool with the AWS CLI**

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

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

For information about using Device Farm with the AWS CLI, see AWS CLI Reference (p. 124).

**Create a Device Pool with the Device Farm API**

- Call the `CreateDevicePool` API.

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

For information about using the Device Farm API, see API Reference (p. 126).

**Analyze a Report in AWS Device Farm**

Use the Device Farm console to analyze a report. For more information, see Reports (p. 13).

**Prerequisites**

- Create a run in Device Farm, and verify the run is complete. Follow the instructions in Create a Test Run (p. 23), and then return to this page.
Console Icons

<table>
<thead>
<tr>
<th>Icon</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green check mark inside of a circle</td>
<td>Success</td>
</tr>
<tr>
<td>Orange exclamation mark inside of a triangle</td>
<td>Warning</td>
</tr>
<tr>
<td>Red exclamation mark inside of a circle</td>
<td>Failure</td>
</tr>
<tr>
<td>Blue circle with a slash through it</td>
<td>Skipped</td>
</tr>
<tr>
<td>Orange square</td>
<td>Stopped</td>
</tr>
</tbody>
</table>

Open a Report with the Device Farm Console

1. Make sure the run is complete.
3. In the list of projects, choose the project for the run that corresponds to the report that you want to access.
   
   **Tip**
   
   If the list of projects is not displayed, then on the secondary navigation bar, choose the Device Farm console home button, and then choose the project.
4. In the list, choose the run with the finished icon that corresponds to the report you want to access. The report’s summary page is displayed.

To analyze the various parts of the report, follow the instructions in the following sections.

Analyze a Report's Summary Page with the Device Farm Console

1. If the report's summary page is not already displayed, follow the instructions in Open a Report with the Device Farm Console (p. 36).
2. At the beginning of the summary page, the total number of tests, by outcome, is displayed.
   - An exclamation mark is displayed next to the number of tests with errors.
   - A square is displayed next to the number of stopped tests.
   - An exclamation mark inside of a circle is displayed next to the number of failed tests.
   - A check mark is displayed next to the number of successful tests.
   - A circle with a slash through it is displayed next to the number of skipped tests.
   - An exclamation mark inside of a triangle is displayed next to the number of tests with warnings.
3. The summary page displays a list of test results as follows:
   - The **Unique problems** section lists unique warnings and failures. To analyze unique problems, follow the instructions in Analyze a Report's Unique Problems with the Device Farm Console (p. 37).
   - The **Devices** section displays the total number of tests, by outcome, for each device.
   - Next to the device's name, one of the following icons is displayed:
     - If there is at least one stopped test for the device, an orange square is displayed.
Analyze a Report's Unique Problems with the Device Farm Console

1. If the report's summary page is not already displayed, follow the instructions in Open a Report with the Device Farm Console (p. 36).
2. Following the total number of tests by outcome for the run, for **Unique problems**, choose the problem that you want to analyze. The list of devices for the problem is displayed.
3. Choose the device whose results you want to analyze. The report displays information about the problem.
4. The **Video** section displays a downloadable video recording of the test.
5. The **Logs** section displays any information Device Farm logged during the test. To analyze this information, follow the instructions in Analyze Log Information for a Problem, Device, Suite, or Test in a Report with the Device Farm Console (p. 40).
6. The **Performance** section displays information about any performance data Device Farm generated during the test. To analyze this performance data, follow the instructions in Analyze Performance Data for a Problem, Device, Suite, or Test in a Report with the Device Farm Console (p. 39).
7. The **Files** section displays a list of tests for the suite and any associated files (such as log files) that can be downloaded. To download a file, choose the file's link in the list.
8. The **Screenshots** section displays a list of any screenshots Device Farm captured during the test.

Analyze a Report by Device with the Device Farm Console

1. If the report's summary page is not already displayed, follow the instructions in Open a Report with the Device Farm Console (p. 36).
2. In the **Devices** section, choose the device whose results you want to analyze.
3. The **Video** section displays a downloadable video recording of the test.
4. The **Suites** section displays information about the suites for the device. For each suite, the following test results are displayed:
   - For **Test Results**, the total number of tests for the suite is displayed by outcome.
   - Next to the suite's name, one of the following icons is displayed:
     - An orange square is displayed if there is at least one stopped test for the suite.
     - A red exclamation mark is displayed if there is at least one test with errors for the suite.
     - A red exclamation mark inside of a circle is displayed if there is at least one failed test for the suite.
• An orange exclamation mark inside of a triangle is displayed if there is at least one test for the
suite with warnings.
• Otherwise, a green check mark inside of a circle is displayed.

To analyze the results by suite, follow the instructions in Analyze a Report by Suite with the Device Farm
Console (p. 38).

The Logs section displays any information Device Farm logged during the run. To
analyze this information, follow the instructions in Analyze Log Information for a Problem, Device,
Suite, or Test in a Report with the Device Farm Console (p. 40).

5. The Performance section displays information about any performance data Device Farm generated
for the device during the run. To analyze this performance data, follow the instructions in
Analyze Performance Data for a Problem, Device, Suite, or Test in a Report with the Device Farm
Console (p. 39).

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

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

**Analyze a Report by Suite with the Device Farm Console**

1. If the report's summary page is not already displayed, follow the instructions in Open a Report with
the Device Farm Console (p. 36).

2. In the Devices section, choose the device that corresponds to the suite whose results you want to
analyze. The device's results page is displayed.

3. In the Suites section, choose the suite that you want to analyze for results. The suite's results page is
displayed.

4. The suite's results page displays information about the tests for the suite. For each test, the
following test results are displayed:

   • For Tests, the outcome for the test is displayed as follows:
     • If the test succeeded, the number 1 is displayed next to a green check mark inside of a circle.
     • If the test has warnings, the number 1 is displayed next to an orange exclamation mark inside of
       a triangle.
     • If the test was skipped, the number 1 is displayed next to a blue circle with a slash through it.
     • If the test failed, the number 1 is displayed next to a red exclamation mark inside of a circle.
     • If the test has errors, the number 1 is displayed next to a red exclamation mark.
     • If the test was stopped, the number 1 is displayed next to an orange square.
   • These icons are also displayed next to the test's name.

To analyze the results by test, follow the instructions in Analyze a Report by Test with the Device Farm
Console (p. 39).

5. The Logs section displays any information Device Farm logged during the run for the suite. To
analyze this information, follow the instructions in Analyze Log Information for a Problem, Device,
Suite, or Test in a Report with the Device Farm Console (p. 40).

6. The Performance section displays information about any performance data Device Farm
generated during the run for the suite. To analyze this performance data, follow the instructions
7. The **Files** section displays a list of tests for the suite and any associated files (such as log files) that can be downloaded. To download a file, choose the file's link in the list.

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

---

### Analyze a Report by Test with the Device Farm Console

1. If the report's summary page is not already displayed, open the report by following the instructions in **Open a Report with the Device Farm Console** (p. 36).

2. In the **Devices** section, choose the device that corresponds to the test you want to analyze for results.

3. In the **Suites** section, choose the suite that corresponds to the test you want to analyze for results.

4. The **Tests** tab displays information about the test.
   - If the test was stopped, an orange square is displayed.
   - If the test contains errors, a red exclamation mark is displayed.
   - If the test failed, a red exclamation mark inside of a circle is displayed.
   - If the test had a warning, an orange exclamation mark inside of a triangle is displayed.
   - Otherwise, a green check mark inside of a circle is displayed.

   The **Logs** section displays any information Device Farm logged during the test. To analyze this information, follow the instructions in **Analyze Log Information for a Problem, Device, Suite, or Test in a Report with the Device Farm Console** (p. 40).

5. The **Performance** tab displays information about any performance data Device Farm generated during the test. To analyze this performance data, follow the instructions in **Analyze Performance Data for a Problem, Device, Suite, or Test in a Report with the Device Farm Console** (p. 39).

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

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

---

### Analyze Performance Data for a Problem, Device, Suite, or Test in a Report with the Device Farm Console

1. If the **Performance** tab is not already displayed, follow one of these sets of instructions and choose the **Performance** tab:
   - **Analyze a Report's Unique Problems with the Device Farm Console** (p. 37)
   - **Analyze a Report by Device with the Device Farm Console** (p. 37)
   - **Analyze a Report by Suite with the Device Farm Console** (p. 38)
   - **Analyze a Report by Test with the Device Farm Console** (p. 39)

2. The following information is displayed:
• The CPU graph displays the percentage of CPU 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 may exceed 100% if the app used more than one core. For example, if three cores are at 60% usage, this percentage will be displayed as 180%.

• The FPS graph displays the frame rate in frames per second (FPS) during the selected problem, device, suite, or test (along the vertical axis) over time (along the horizontal axis).

The vertical axis is expressed in FPS from 0 FPS to the maximum number of recorded FPS.

• The Memory graph displays the number of MB 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 0 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.

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

**Analyze Log Information for a Problem, Device, Suite, or Test in a Report with the Device Farm Console**

1. If the Logs section is not already displayed, follow one of these sets of instructions and choose the Logs tab:

   • Analyze a Report's Unique Problems with the Device Farm Console (p. 37).
   • Analyze a Report by Device with the Device Farm Console (p. 37)
   • Analyze a Report by Suite with the Device Farm Console (p. 38)
   • Analyze a Report by Test with the Device Farm Console (p. 39)

2. The following information is displayed:

   • **Source** represents the source of a log entry. Possible values include:
     • **Harness** represents a log entry Device Farm created. These log entries are typically created during start and stop events.
     • **Device** represents a log entry 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 will have the same PID.
     • **Level** represents the logging level for the log entry. For example, Logger.debug("This is a message!") would log a Level of Debug. Possible values include the following:
Analyze Log Information for a Problem, Device, Suite, or Test in a Report with the Device Farm Console

- 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!") would log a Message of "Hello, World!".

3. To display only a portion of the information, do one or more of the following:

- To show all log entries that match a value for a specific column, type the value into the corresponding column header box. For example, to show all log entries with a Source value of Harness, type Harness in the Source column header box. Similarly, to show all log entries with a PID value of 969 and a Tag value of ActivityManager, type 969 in the PID column header box, and type ActivityManager in the Tag column header box.

- To show all log entries that contain zero or more unknown characters for a specific column, use the wildcard character (*) to represent the unknown characters. For example, to show all log entries with a Source value that contain an es (such as Harness and Test), type *es* in the Source column header box. Similarly, to show all log entries that start with a Source value of H (such as Harness) and have a Level value that contains an e (such as Passed), type H* in the PID column header box, and type *e* in the Level column header box.

- To show log entries that contain a choice between one or more known characters for a specific column, surround the set of choices in parentheses (()), and use the pipe character (|) to separate each choice. For example, to show log entries with a Message value that contains either started or starting, type *start(ed|ing)* in the Message column header box. Similarly, to show all log entries with a Log value of Info or Debug, type *(Info|Debug)* in the Log column header box.

- 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 typing * in that column header box.

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

**Note**
Even if you display only a portion of the information, if you choose **Download logs**, all log information for the device will be downloaded.
Working with Test Types in AWS Device Farm

Device Farm provides support for several automation test types.

Built-in Test Types

Built-in tests enable you to test your apps without writing scripts.

- Built-in: Explorer (Android) (p. 84)
- Built-in: Fuzz (Android and iOS) (p. 85)

Custom Test Types

Custom tests allow you to test specific flows and business logic within your app.

Custom Android Test Types

- Appium Java JUnit (p. 43)
- Appium Java TestNG (p. 47)
- Appium Python (p. 51)
- Calabash (p. 54)
- Instrumentation (p. 56)
- UI Automator (p. 57)

Custom iOS Test Types

- Appium Java JUnit (p. 59)
- Appium Java TestNG (p. 63)
- Appium Python (p. 67)
- Calabash (p. 70)
- UI Automation (p. 72)
- XCTest (p. 73)
- XCTest UI (p. 74)

Custom Web Application Test Types

- Appium Java JUnit (p. 76)
- Appium Java TestNG (p. 78)
- Appium Python (p. 80)
Working with Android Tests in AWS Device Farm

Device Farm provides support for several automation test types.

Built-in Test Types for Android

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

- Built-in: Explorer (Android) (p. 84)
- Built-in: Fuzz (Android and iOS) (p. 85)

Custom Test Types for Android

The following custom tests are available for Android devices.

- Appium Java JUnit (p. 43)
- Appium Java TestNG (p. 47)
- Appium Python (p. 51)
- Calabash (p. 54)
- Instrumentation (p. 56)
- UI Automator (p. 57)

Working with Appium Java JUnit for Android and AWS Device Farm

Device Farm provides support for Appium Java JUnit for Android.

Device Farm also provides a sample Android application along with links to working tests in three Android automation frameworks, including Appium. You can download the Device Farm sample app for Android on GitHub.

Topics

- What Is Appium Java JUnit? (p. 43)
- Version Information (p. 43)
- Prepare Your Android Appium Java JUnit Tests (p. 44)
- Upload Your Android Appium Java JUnit Tests (p. 46)
- Taking Screenshots in Android Appium Java JUnit Tests (p. 46)
- Additional Considerations for Android Appium Java JUnit Tests (p. 47)

What Is Appium Java JUnit?

Appium is an open-source tool for automating native, mobile web, and hybrid applications on platforms such as Android. For more information about Appium, see Introduction to Appium.

Version Information

Currently, Device Farm supports Appium versions 1.6.3, 1.6.5, and 1.4.16 and Java 8.
Note
If desired, you can use the rules field in the create-device-pool command or the CreateDevicePool API to specify the APPium_VERSION.

Prepare Your Android Appium Java JUnit Tests

Your Android Appium Java JUnit tests must be contained in a .zip file.

Build the Appium Java Test Package

The Appium Java test package you upload to Device Farm must be in .zip format and contain all of the tests’ dependencies. The following instructions will show you how to meet these requirements during the package stage of a Maven build.

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

```xml
<groupId>com.acme</groupId>
<artifactId>acme-android-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 will build 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 will copy 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 containing 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>
```

6. Build, package, and verify. For example:

```
$ mvn clean package -DskipTests=true
```
# tree target

.  
  |- 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
       |- (and so on...)

7. Use the Device Farm console to upload the test package.

**Tip**
If you receive an error saying that 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>
```

### Upload Your Android Appium Java JUnit Tests

Use the Device Farm console to upload your tests:

2. In the list of projects, choose the option next to the project where you want to upload your tests.

   **Tip**
   If the list of projects is not displayed, then on the secondary navigation bar, for **Projects**, choose the name of the project where you want to upload your tests.
   To create a new project, follow the instructions in **Create a Project** (p. 20).

3. If the **Create a new run** button is displayed, then choose it.
4. On the **Choose your application** page, choose **Upload**.
5. Browse to and choose your Android app file. The file must be an .apk file.
6. Choose **Next step**.
7. On the **Configure a test** page, choose **Appium Java JUnit**, and then choose **Upload**.
8. Browse to and choose the .zip file that contains your tests. The .zip file must follow the format described in **Prepare Your Android Appium Java JUnit Tests** (p. 44).
9. Choose the Appium version you are using from the **Appium version** dropdown list.
10. Choose **Next step**, and then complete the remaining on-screen instructions to select devices and start the run.

### Taking Screenshots in Android Appium Java JUnit Tests

You can take screenshots as part of your Android Appium Java JUnit tests.
When Device Farm runs your Appium Java JUnit test, the service sets the following system properties that describe the configuration of the Appium server with which you’re communicating:

- **appium.screenshots.dir**: path to which the screenshots are saved
- **appium.server.address**: host address of the Appium server
- **appium.server.port**: port on which the Appium server is listening

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 automatically pulled into your Device Farm reports automatically. To view the screenshots, in the Device Farm console, choose the Screenshots section.

The following example shows how to use and consume the `appium.screenshots.dir` property to capture an Appium screenshot that is pulled into your Device Farm report.

```java
public boolean takeScreenshot(final String name) {
    String screenshotDirectory = System.getProperty("appium.screenshots.dir", System.getProperty("java.io.tmpdir", ""));
    File screenshot = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
    return screenshot.renameTo(new File(screenshotDirectory, String.format("%s.png", name)));
}
```

### Additional Considerations for Android Appium Java JUnit Tests

Device Farm does not modify Android Appium Java JUnit tests.

#### Working with Appium Java TestNG for Android and AWS Device Farm

Device Farm provides support for Appium Java TestNG for Android.

Device Farm also provides a sample Android application along with links to working tests in three Android automation frameworks, including Appium. You can download the Device Farm sample app for Android on GitHub.

### Topics

- What Is Appium Java TestNG? (p. 47)
- Version Information (p. 47)
- Prepare Your Android Appium Java TestNG Tests (p. 48)
- Upload Your Android Appium Java TestNG Tests (p. 50)
- Taking Screenshots in Android Appium Java TestNG Tests (p. 50)
- Additional Considerations for Android Appium Java TestNG Tests (p. 51)

#### What Is Appium Java TestNG?

Appium is an open-source tool for automating native, mobile web, and hybrid applications on platforms such as Android. For more information about Appium, see Introduction to Appium.

#### Version Information

Currently, Device Farm supports Appium versions 1.6.3, 1.6.5, and 1.4.16 and Java 8.
**Note**

If desired, you can use the `rules` field in the `create-device-pool` command or the the `CreateDevicePool` API to specify the `APPIUM_VERSION`.

## Prepare Your Android Appium Java TestNG Tests

Your Android Appium Java TestNG tests must be contained in a .zip file before you upload them to Device Farm.

### Build the Appium Java Test Package

The Appium Java test package you upload to Device Farm must be in .zip format and contain all of the tests’ dependencies. The following instructions will show you how to meet these requirements during the package stage of a Maven build.

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

   ```xml
   <groupId>com.acme</groupId>
   <artifactId>acme-android-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 will build 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 will copy 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 containing 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>
```

6. Build, package, and verify. For example:

```bash
# mvn clean package -DskipTests=true
```
# tree target
  
  |- 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
      `-- (and so on...)

7. Use the Device Farm console to upload the test package.

   **Tip**
   If you receive an error saying that 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>
   ```

---

**Upload Your Android Appium Java TestNG Tests**

Use the Device Farm console to upload your tests:

2. In the list of projects, choose the option next to the project where you want to upload your tests.
   
   **Tip**
   If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project where you want to upload your tests.
   
   To create a new project, follow the instructions in [Create a Project (p. 20)](#).

3. If the **Create a new run** button is displayed, then choose it.
4. On the **Choose your application** page, choose **Upload**.
5. Browse to and choose your Android app file. The file must be an .apk file.
6. Choose **Next step**.
7. On the **Configure a test** page, choose **Appium Java TestNG**, and then choose **Upload**.
8. Browse to and choose the .zip file that contains your tests. The .zip file must follow the format described in [Prepare Your Android Appium Java TestNG Tests (p. 48)](#).
9. Choose the Appium version you are using from the **Appium version** dropdown list.
10. Choose **Next step**, and then complete the remaining on-screen instructions to select devices and start the run.

---

**Taking Screenshots in Android Appium Java TestNG Tests**

You can take screenshots as part of your Android Appium Java TestNG tests.
When Device Farm runs your Appium Java JUnit test, the service sets the following system properties that describe the configuration of the Appium server with which you’re communicating:

- **appium.screenshots.dir**: path to which the screenshots are saved
- **appium.server.address**: host address of the Appium server
- **appium.server.port**: port on which the Appium server is listening

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 automatically pulled into your Device Farm reports automatically. To view the screenshots, in the Device Farm console, choose the **Screenshots** section.

The following example shows how to use and consume the **appium.screenshots.dir** property to capture an Appium screenshot that is pulled into your Device Farm report.

```java
public boolean takeScreenshot(final String name) {
    String screenshotDirectory = System.getProperty("appium.screenshots.dir", 
        System.getProperty("java.io.tmpdir", ")");
    File screenshot = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
    return screenshot.renameTo(new File(screenshotDirectory, String.format("%s.png", 
        name)));
}
```

**Additional Considerations for Android Appium Java TestNG Tests**

Device Farm does not modify Android Appium Java TestNG tests.

**Working with Appium Python for Android Applications and AWS Device Farm**

Device Farm provides support for Appium Python for Android apps.

**Topics**
- *What Is Appium Python?* (p. 51)
- *Version Information* (p. 51)
- *Prepare Your Android Application Appium Python Tests* (p. 52)
- *Build the Appium Python Test Package* (p. 52)
- *Upload Your Android Application Appium Python Tests* (p. 53)
- *Taking Screenshots in Android Appium Python Tests* (p. 54)
- *Additional Considerations for Android Appium Python Tests* (p. 54)

**What Is Appium Python?**

Appium is an open-source tool for automating native, mobile web, and hybrid applications on platforms like Android. For more information about Appium, see *Introduction to Appium*.

**Version Information**

Currently, Device Farm supports Appium versions 1.6.3, 1.6.5, and 1.4.16 and Python version 2.7.6 (pip version 1.5.4).
**Prepare Your Android Application Appium Python Tests**

The Appium Python tests for your Android application must be contained in a .zip file.

**Build the Appium Python Test Package**

The Appium Python test packages you upload to Device Farm must be in .zip format and contain all the dependencies of your test. The following instructions show you how to meet these requirements.

**Note**
The instructions below are based on Linux x86_64 and Mac. In the currently supported scheme, Device Farm requires that the packaging of your Appium Python tests be done on Linux x86_64 if your tests contain non-universal Python wheels dependencies. For the platform on which you execute a command, the wheels tools gather your .whl dependent files under the `wheelhouse/` folder. When you execute the Python wheel command on any platform other than Linux x86_64, you would gather the flavor of a non-universal wheel dependency for that particular platform and may cause undesired effects. This would most likely lead to errors when executing your tests on Device Farm.

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.**

   **Tip**
   - Do not create a Python virtualenv with the `--system-site-packages` option, because it will inherit packages from `/usr/lib/python.x.x/site-packages` or wherever your global site-packages directory is. This can lead to you including dependencies in your virtual environment that are not needed by your tests.
   - You should also verify that your tests do not use dependencies that are dependent on native libraries, as these native libraries may or may not be present on the instance where these tests run.

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

   An example flow of creating a virtual environment using Python virtualenv and installing pytest in that virtual environment would look like the following:

   ```
   $ virtualenv workspace
   $ cd workspace
   $ source bin/activate
   $ pip install pytest
   ```

3. **Store all Python test scripts under the `tests/` folder in your work space.**

   ```
   # workspace
   ## tests/ (your tests go here)
   ```

4. **Make sure you have py.test installed in your virtual environment and test cases are discoverable by the following command, which you should run from your virtual environment `workspace` folder.**

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

   Make sure the output of py.test command shows you the tests that you want to execute on Device Farm.
5. Go to your work space and run the following command to generate the requirements.txt file:

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

6. Go to your work space and run the following command to generate the wheelhouse/ folder:

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

7. You can use the following commands to 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 {} +
```

8. Zip the tests/ folder, wheelhouse/ folder, and the requirements.txt file into a single archive:

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

Your work space will eventually look like this:

```
# workspace
## tests/
## test_bundle.zip
## requirements.txt
## wheelhouse/
```

### Upload Your Android Application Appium Python Tests

Use the Device Farm console to upload your tests.

2. If you see the AWS Device Farm console home page, choose Get started.
3. If you already have a project, you can upload your tests to an existing project or choose Create a new project.

   ![Create a new project](image)

**Tip**

If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project where you want to upload your tests. To create a project, follow the instructions in Create a Project (p. 20).

4. If the Create a new run button is displayed, choose it.
5. On the Choose your application page, choose Native Application (the Android and Apple logos).
6. Next, choose **Upload** to upload your .apk file.
   
   Device Farm processes your .apk file before continuing.

7. In the **Run name** field, type a name for your run.

   **Tip**
   
   Give the run a name that will help you identify a specific build of your app (for example, Beta-0.1). For more information, see Working with Test Runs (p. 23).

8. Choose **Appium Python** to configure your test,

9. To add your Appium test scripts to the test run, choose **Upload**.

10. Choose the Appium version you are using from the **Appium version** dropdown list.

11. Choose **Next step**, and then complete the instructions to select devices and start the run.

### Taking Screenshots in Android Appium Python Tests

You can take screenshots as part of your Android Appium Python tests.

When Device Farm runs your Appium Python test, the service 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.

The following example shows how to use and consume the `SCREENSHOT_PATH` property to capture an Appium screenshot that is pulled into your Device Farm report.

```python
screenshot_folder = os.getenv('SCREENSHOT_PATH', '')
self.driver.save_screenshot(screenshot_folder + '/screenshot.png')
```

### Additional Considerations for Android Appium Python Tests

Device Farm does not modify Android Appium Python tests.

### Working with Calabash for Android and AWS Device Farm

Device Farm provides support for Calabash for Android.

Device Farm also provides a sample Android application along with links to working tests in three Android automation frameworks, including **Calabash**. You can download the Device Farm sample app for Android on GitHub.
What Is Calabash?

Calabash is a mobile testing framework that enables automated user interface acceptance tests written in Cucumber to be run on Android apps. For more information, see the Welcome to Calabash for Android repository on GitHub.

Version Information

Currently, Device Farm supports Calabash version 0.9.0.

Prepare Your Android Calabash Tests

Your Android Calabash tests must be contained in a .zip file before you upload them to Device Farm. This .zip file must contain the following structure:

```
my-zip-file-name.zip
  |-- 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)
```

Upload Your Android Calabash Tests

Use the Device Farm console to upload your tests:

2. In the list of projects, choose the option next to the project where you want to upload your tests.
   
   Tip
   
   If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project where you want to upload your tests.
   
   To create a new project, follow the instructions in Create a Project (p. 20).
3. If the Create a new run button is displayed, then choose it.
4. On the Choose your application page, choose Upload.
5. Browse to and choose your Android app file. The file must be an .apk file.
6. Choose Next step.
7. On the Configure a test page, choose Calabash, and then choose Upload.
8. Browse to and choose the .zip file that contains your tests. The .zip file must follow the format described in Prepare Your Android Calabash Tests (p. 55).
9. Choose Next step, and then complete the remaining on-screen instructions to select devices and start the run.

Taking Screenshots in Android Calabash Tests

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

Android Calabash provides a set of predefined steps for taking screenshots. For details, see the "Screenshots" section of the Canned steps page in the Calabash Android repository 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 Android Calabash Tests

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

### 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 along with links to working tests in three Android automation frameworks, including Instrumentation (Espresso). You can download the Device Farm sample app for Android on GitHub.

#### Topics
- What Is Instrumentation? (p. 56)
- Upload Your Android Instrumentation Tests (p. 56)
- Taking Screenshots in Android Instrumentation Tests (p. 57)
- Additional Considerations for Android Instrumentation Tests (p. 57)

### What Is Instrumentation?

Android instrumentation enables you to invoke callback methods in your test code. This allows you to 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. In the list of projects, choose the option next to the project where you want to upload your tests.

   **Tip**
   - If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project where you want to upload your tests.
   - To create a new project, follow the instructions in Create a Project (p. 20).
3. If the Create a new run button is displayed, then choose it.
4. On the Choose your application page, choose Upload.
5. Browse to and choose your Android app file. The file must be an .apk file.
6. Choose Next step.
7. On the Configure a test page, choose Instrumentation, and then choose Upload.
8. Browse to and choose the .apk file that contains your tests.
9. Choose Next step, and then complete the remaining on-screen 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 automatically 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 will automatically disable Window Animation Scale, Transition Animation Scale, and Animator Duration Scale settings when executing 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 and may not work with all scripts and apps.

Topics

- What Is UI Automator? (p. 58)
- Prepare Your Android UI Automator Tests (p. 58)
- Upload Your Android UI Automator Tests (p. 58)
- Taking Screenshots in Android UI Automator Tests (p. 58)
- Additional Considerations for Android UI Automator Tests (p. 58)
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 before you upload them to Device Farm. The package name in this file must match the package name used by the related Android app. For example, if the Android app's package name is com.my.android.app.MyMobileApp, then 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. In the list of projects, choose the option next to the project where you want to upload your tests.
   
   Tip
   
   If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project where you want to upload your tests. To create a new project, follow the instructions in Create a Project (p. 20).

3. If the Create a new run button is displayed, then choose it.
4. On the Choose your application page, choose Upload.
5. Browse to and choose your Android app file. The file must be an .apk file.
6. Choose Next step.
7. On the Configure a test page, choose uiautomator, and then choose Upload.
8. Browse to and choose the JAR file that contains your tests. Make sure the Android tests are organized according to the instructions in Prepare Your Android UI Automator Tests (p. 58).
9. Choose Next step, and then complete the remaining on-screen 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.

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. 85)

Custom Test Types

The following custom tests are available for iOS devices.

- Appium Java JUnit (p. 59)
- Appium Java TestNG (p. 63)
- Appium Python (p. 67)
- Calabash (p. 70)
- UI Automation (p. 72)
- XCTest (p. 73)
- XCTest UI (p. 74)

Working with Appium Java JUnit for iOS and AWS Device Farm

Device Farm provides support for Appium Java JUnit for iOS. The following information describes how to use this test framework with Device Farm test types.

Topics

- What is Appium Java JUnit? (p. 59)
- Version Information (p. 59)
- Prepare Your iOS Appium Java JUnit Tests (p. 60)
- Upload Your iOS Appium Java JUnit Tests (p. 62)
- Taking Screenshots in iOS Appium Java JUnit Tests (p. 62)
- Additional Considerations for iOS Appium Java JUnit Tests (p. 63)

What is Appium Java JUnit?

Appium is an open-source tool for automating native, mobile web, and hybrid applications on platforms such as iOS. For more information about Appium, see Introduction to Appium.

Version Information

Currently, Device Farm supports Appium versions 1.6.3 and 1.6.5 (for iOS 10 and later) and 1.4.16 (for iOS 9 and earlier) and Java 8.
Note
If desired, you can use the rules field in the create-device-pool command or the CreateDevicePool API to specify the APPium_VERSION.

Prepare Your iOS Appium Java JUnit Tests

Before you upload your iOS Appium Java JUnit tests to Device Farm for testing, make sure that your iOS Appium Java JUnit tests are contained within a .zip file.

Build the Appium Java Test Package

The Appium Java test package you upload to Device Farm must be in .zip format and contain all of the tests' dependencies. The following instructions will show you how to meet these requirements during the package stage of a Maven build.

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

```xml
<groupId>com.acme</groupId>
<artifactId>acme-android-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 will build 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 will copy 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 containing 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>./dependency-jars/</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>
```

6. Build, package, and verify. For example:

```bash
$ mvn clean package -DskipTests=true
```
# tree target

```
|— 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
    |— (and so on...)
```

7. Use the Device Farm console to upload the test package.

**Tip**
If you receive an error saying that 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>
```

## Upload Your iOS Appium Java JUnit Tests

To run your iOS Appium Java JUnit tests on a set of iOS devices in Device Farm, you upload your tests with the Device Farm console as follows:

2. In the list of projects, choose the option next to the project that you want to upload your tests to.
   **Tip**
   If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project that you want to upload your tests to.
   To create a new project, follow the instructions in Create a Project (p. 20).
3. If the Create a new run button is displayed, then choose it.
4. On the Choose your application page, choose Upload.
5. Browse to and choose your iOS app file. The file must be an .ipa file.
   **Note**
   Make sure that your app file is built for an iOS device and not for a simulator.
6. Choose Next step.
7. On the Configure a test page, choose Appium Java JUnit, and then choose Upload.
8. Choose the Appium version you are using from the Appium version dropdown list.
9. Choose Next step, and then complete the remaining on-screen instructions to select the devices to run your tests on and to then start the run.

## Taking Screenshots in iOS Appium Java JUnit Tests

You can take screenshots as part of your iOS Appium Java JUnit tests.
When Device Farm runs your Appium Java JUnit test, the service sets the following system properties that describe the configuration of the Appium server with which you're communicating:

- `appium.screenshots.dir`: path to which the screenshots are saved
- `appium.server.address`: host address of the Appium server
- `appium.server.port`: port on which the Appium server is listening

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 automatically pulled into your Device Farm reports automatically. To view the screenshots, in the Device Farm console, choose the Screenshots section.

The following example shows how to use and consume the `appium.screenshots.dir` property to capture an Appium screenshot that is pulled into your Device Farm report.

```java
public boolean takeScreenshot(final String name) {
    String screenshotDirectory = System.getProperty("appium.screenshots.dir",
            System.getProperty("java.io.tmpdir", ")");
    File screenshot = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
    return screenshot.renameTo(new File(screenshotDirectory,
            String.format("%s.png", name)));
}
```

Additional Considerations for iOS Appium Java JUnit Tests

Device Farm does not modify iOS Appium Java JUnit tests.

Working with Appium Java TestNG for iOS and AWS Device Farm

Device Farm provides support for Appium Java TestNG for iOS. The following information describes how to use this test framework with Device Farm test types.

Topics

- What is Appium Java TestNG? (p. 63)
- Version Information (p. 63)
- Prepare Your iOS Appium Java TestNG Tests (p. 64)
- Upload Your iOS Appium Java TestNG Tests (p. 66)
- Taking Screenshots in iOS Appium Java TestNG Tests (p. 66)
- Additional Considerations for iOS Appium Java TestNG Tests (p. 67)

What is Appium Java TestNG?

Appium is an open-source tool for automating native, mobile web, and hybrid applications on platforms such as iOS. For more information about Appium, see Introduction to Appium.

Version Information

Currently, Device Farm supports Appium versions 1.6.3 and 1.6.5 (for iOS 10 and later) and 1.4.16 (for iOS 9 and earlier) and Java 8.
Prepare Your iOS Appium Java TestNG Tests

Before you upload your iOS Appium Java TestNG tests to Device Farm for testing, make sure that your iOS Appium Java TestNG tests are contained within a .zip file.

Build the Appium Java Test Package

The Appium Java test package you upload to Device Farm must be in .zip format and contain all of the tests’ dependencies. The following instructions will show you how to meet these requirements during the package stage of a Maven build.

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

   ```xml
   <groupId>com.acme</groupId>
   <artifactId>acme-android-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 will build 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 will copy 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 containing 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>
```

6. Build, package, and verify. For example:

```
$ mvn clean package -DskipTests=true
```
7. Use the Device Farm console to upload the test package.

Tip
If you receive an error saying that 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>
```

### Upload Your iOS Appium Java TestNG Tests

To run your iOS Appium Java TestNG tests on a set of iOS devices in Device Farm, you upload your tests with the Device Farm console as follows:

2. In the list of projects, choose the option next to the project that you want to upload your tests to.

   Tip
   If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project that you want to upload your tests to.

   To create a new project, follow the instructions in Create a Project (p. 20).

3. If the Create a new run button is displayed, then choose it.
4. On the Choose your application page, choose Upload.
5. Browse to and choose your iOS app file. The file must be an .ipa file.

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

6. Choose Next step.
7. On the Configure a test page, choose Appium Java TestNG, and then choose Upload.
8. Choose the Appium version you are using from the Appium version dropdown list.
9. Choose Next step, and then complete the remaining on-screen instructions to select the devices to run your tests on and to then start the run.

### Taking Screenshots in iOS Appium Java TestNG Tests

You can take screenshots as part of your iOS Appium Java TestNG tests.
When Device Farm runs your Appium Java JUnit test, the service sets the following system properties that describe the configuration of the Appium server with which you're communicating:

- `appium.screenshots.dir`: path to which the screenshots are saved
- `appium.server.address`: host address of the Appium server
- `appium.server.port`: port on which the Appium server is listening

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 automatically pulled into your Device Farm reports automatically. To view the screenshots, in the Device Farm console, choose the **Screenshots** section.

The following example shows how to use and consume the `appium.screenshots.dir` property to capture an Appium screenshot that is pulled into your Device Farm report.

```java
public boolean takeScreenshot(final String name) {
    String screenshotDirectory = System.getProperty("appium.screenshots.dir",
            System.getProperty("java.io.tmpdir", ""));
    File screenshot = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
    return screenshot.renameTo(new File(screenshotDirectory, String.format("%s.png", name)));
}
```

**Additional Considerations for iOS Appium Java TestNG Tests**

Device Farm does not modify iOS Appium Java TestNG tests.

**Working with Appium Python for iOS Applications and AWS Device Farm**

Device Farm provides support for Appium Python for iOS apps.

**Topics**

- **What Is Appium Python?** (p. 67)
- **Version Information** (p. 67)
- **Prepare Your iOS Application Appium Python Tests** (p. 68)
- **Build the Appium Python Test Package** (p. 68)
- **Upload Your iOS Application Appium Python Tests** (p. 69)
- **Taking Screenshots in iOS Appium Python Tests** (p. 70)
- **Additional Considerations for Android Appium Python Tests** (p. 70)

**What Is Appium Python?**

Appium is an open-source tool for automating native, mobile web, and hybrid applications on platforms like web applications. For more information about Appium, see **Introduction to Appium**.

**Version Information**

Currently, Device Farm supports Appium versions 1.6.3 and 1.6.5 (for iOS 10 and later) and 1.4.16 (for iOS 9 and earlier) and Python version 2.7.6 (pip version 1.5.4).
Note
If desired, you can use the rules field in the create-device-pool command or the CreateDevicePool API to specify the APPIUM_VERSION.

Prepare Your iOS Application Appium Python Tests

The Appium Python tests for your iOS application must be contained in a .zip file.

Build the Appium Python Test Package

The Appium Python test packages you upload to Device Farm must be in .zip format and contain all the dependencies of your test. The following instructions show you how to meet these requirements.

Note
The instructions below are based on Linux x86_64 and Mac. In the currently supported scheme, Device Farm requires that the packaging of your Appium Python tests be done on Linux x86_64 if your tests contain non-universal Python wheels dependencies. For the platform on which you execute a command, the wheels tools gather your .whl dependent files under the wheelhouse/ folder. When you execute the Python wheel command on any platform other than Linux x86_64, you would gather the flavor of a non-universal wheel dependency for that particular platform and may cause undesired effects. This would most likely lead to errors when executing your tests on Device Farm.

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.

   Tip
   • Do not create a Python virtualenv with the --system-site-packages option, because it will inherit packages from /usr/lib/python.x.x/site-packages or wherever your global site-packages directory is. This can lead to you including dependencies in your virtual environment that are not needed by your tests.
   • You should also verify that your tests do not use dependencies that are dependent on native libraries, as these native libraries may or may not be present on the instance where these tests run.

2. Install py.test in your virtual environment.

   An example flow of creating a virtual environment using Python virtualenv and installing pytest in that virtual environment would look like the following:

   ```
   $ virtualenv workspace
   $ cd workspace
   $ source bin/activate
   $ pip install pytest
   ```

3. Store all Python test scripts under the tests/ folder in your work space.

   ```
   # workspace
   ## tests/ (your tests go here)
   ```

4. Make sure you have py.test installed in your virtual environment and test cases are discoverable by the following command, which you should run from your virtual environment workspace folder.

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

   Make sure the output of py.test command shows you the tests that you want to execute on Device Farm.
5. Go to your work space and run the following command to generate the requirements.txt file:

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

6. Go to your work space and run the following command to generate the wheelhouse/ folder:

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

7. You can use the following commands to clean all cached files under your tests/ folder:

```bash
$ 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 {} +
```

8. Zip the tests/ folder, wheelhouse/ folder, and the requirements.txt file into a single archive:

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

Your work space will eventually look like this:

```
# workspace
  ## tests/
  ## test_bundle.zip
  ## requirements.txt
  ## wheelhouse/
```

---

**Upload Your iOS Application Appium Python Tests**

Use the Device Farm console to upload your tests.

2. If you see the AWS Device Farm console home page, choose **Get started**.
3. If you already have a project, you can upload your tests to an existing project or choose **Create a new project**.

   ![Create a new project](image)

   **Tip**
   If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project where you want to upload your tests.
   To create a project, follow the instructions in **Create a Project** (p. 20).

4. If the **Create a new run** button is displayed, choose it.
5. On the **Choose your application** page, choose **Native Application** (the Android and Apple logos).
6. Next, choose **Upload** to upload your .ipa file.

   Device Farm processes your .ipa file before continuing.

7. In the **Run name** field, type a name for your run.

   **Tip**
   
   Give the run a name that will help you identify a specific build of your app (for example, **Beta-0.1**). For more information, see **Working with Test Runs** (p. 23).

8. Choose **Appium Python** to configure your test.

9. To add your Appium test scripts to the test run, choose **Upload**.

10. Choose the Appium version you are using from the **Appium version** dropdown list.

11. Choose **Next step**, and then complete the instructions to select devices and start the run.

---

### Taking Screenshots in iOS Appium Python Tests

You can take screenshots as part of your iOS Appium Python tests.

When Device Farm runs your Appium Python test, the service 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.

The following example shows how to use and consume the `SCREENSHOT_PATH` property to capture an Appium screenshot that is pulled into your Device Farm report.

```python
screenshot_folder = os.getenv('SCREENSHOT_PATH', '')
self.driver.save_screenshot(screenshot_folder + '/screenshot.png')
```

---

### Additional Considerations for Android Appium Python Tests

Device Farm does not modify iOS Appium Python tests.

---

### Working with Calabash for iOS and AWS Device Farm

Device Farm provides support for Calabash for iOS. The following information describes how to use this test framework with Device Farm test types.

**Topics**

- **What is Calabash?** (p. 71)
- **Version Information** (p. 71)
What is Calabash?

Calabash is a mobile testing framework that enables automated user interface acceptance tests that are written in Cucumber to be run on iOS apps. For more information, see the Welcome to Calabash iOS repository on GitHub.

Version Information

Currently Device Farm supports Calabash version 0.20.5.

Prepare Your iOS Calabash Tests

Before you upload your iOS Calabash tests to Device Farm for testing, make sure that your iOS Calabash tests are contained within a .zip file. This .zip file must contain the following structure:

```
my-zip-file-name.zip
  |-- 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)
```

Upload Your iOS Calabash Tests

To run your iOS Calabash tests on a set of iOS devices in Device Farm, you upload your tests with the Device Farm console as follows:

2. In the list of projects, choose the option next to the project that you want to upload your tests to.
   
   **Tip**
   
   If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project that you want to upload your tests to. To create a new project, follow the instructions in Create a Project (p. 20).

3. If the Create a new run button is displayed, then choose it.
4. On the Choose your application page, choose Upload.
5. 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.

6. Choose Next step.
7. On the Configure a test page, choose Calabash, and then choose Upload.
8. Browse to and choose the .zip file that contains your tests. The .zip file must follow the format as described in Prepare Your iOS Calabash Tests (p. 71).
9. Choose **Next step**, and then complete the remaining on-screen instructions to select the devices to run your tests on and to then start the run.

**Taking Screenshots in iOS Calabash Tests**

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

iOS Calabash provides a predefined step for taking screenshots. For details, see the "Screenshots" section of the Predefined steps page in the Calabash iOS repository 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 that you define. For example, the following code example creates a screenshot in PNG format 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 iOS Calabash Tests**

Device Farm replaces certain Calabash hooks so that iOS Calabash tests will run on devices in Device Farm, but Device Farm does not modify iOS Calabash tests themselves.

**Working with UI Automation for iOS and AWS Device Farm**

Device Farm provides support for UI Automation for iOS. The following information describes how to use this test framework with Device Farm test types.

**Topics**
- What is UI Automation? (p. 72)
- Upload Your iOS UI Automation Tests (p. 72)
- Taking Screenshots in iOS UI Automation Tests (p. 73)
- Additional Considerations for iOS UI Automation Tests (p. 73)

**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, 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 section of the iOS Developer Library.

**Upload Your iOS UI Automation Tests**

To run your iOS UI Automation tests on a set of iOS devices in Device Farm, you upload your tests with the Device Farm console as follows:

2. In the list of projects, choose the option next to the project that you want to upload your tests to.

   **Tip**
   If the list of projects is not displayed, then on the secondary navigation bar, for **Projects**, choose the name of the project that you want to upload your tests to.
To create a new project, follow the instructions in Create a Project (p. 20).

3. If the Create a new run button is displayed, then choose it.
4. On the Choose your application page, choose Upload.
5. 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.

6. Choose Next step.
7. On the Configure a test page, choose UI Automation, and then choose Upload.
8. Browse to and choose the .js file for a single test.
9. Choose Next step, and then complete the remaining on-screen instructions to select the devices to run your tests on and to then 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.captureScreenWithName(lang + "_home");
```
where `lang` 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**

Device Farm provides support for XCTest (including KIF) for iOS, written both Objective-C and Swift. The following information describes how to use this test framework with Device Farm test types.

**Topics**
- What is XCTest (and KIF)? (p. 73)
- Prepare Your iOS XCTest Tests (p. 74)
- Upload Your iOS XCTest Tests (p. 74)
- Taking Screenshots in iOS XCTest Tests (p. 74)
- Additional Considerations for iOS XCTest Tests (p. 74)

**What is XCTest (and KIF)?**

XCTest is the new testing framework introduced with Xcode 5. XCTest is a modernized reimplementation of OCUnit, the previous-generation testing framework. For more information, see XCTest—the Xcode Testing Framework and Transitioning from OCUnit to XCTest in the Testing with Xcode section of the iOS Developer Library.

KIF (which stands for Keep It Functional) is a related iOS integration test framework. It allows for easy automation of iOS apps by leveraging the accessibility attributes that the operating system makes available for those with visual disabilities. KIF builds and performs the tests using a standard XCTest testing target. For more information, see the KIF iOS Integration Testing Framework repository on GitHub.
Prepare Your iOS XCTest Tests

Before you upload iOS XCTest tests to Device Farm for testing, make sure that your iOS XCTest tests are contained within a .zip file. This .zip file must contain your *my-project-name*.xctest directory at the root of the .zip file. The actual iOS XCTest bundle must be located within this *my-project-name*.xctest directory.

Upload Your iOS XCTest Tests

To run your iOS XCTest tests on a set of iOS devices in Device Farm, you upload your tests with the Device Farm console as follows:

2. In the list of projects, choose the option next to the project that you want to upload your tests to.
   
   **Tip**
   If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project that you want to upload your tests to.
   
   To create a new project, follow the instructions in Create a Project (p. 20).

3. If the **Create a new run** button is displayed, then choose it.
4. On the **Choose your application** page, choose **Upload**.
5. 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.

6. Choose **Next step**.
7. On the **Configure a test** page, choose **XCTest**, and then choose **Upload**.
8. Browse to and choose the .zip file that contains your iOS XCTest tests. In this .zip file, make sure that the contents are organized according to the instructions as described in Prepare Your iOS XCTest Tests (p. 74).
9. Choose **Next step**, and then complete the remaining on-screen instructions to select the devices to run your tests on and to then start the run.

Taking Screenshots in iOS XCTest Tests

Device Farm currently supports taking screenshots as part of your iOS XCTest tests using KIF. By default, KIF will automatically capture screenshots after any failed steps during your tests, and these will be included in your Device Farm report. If you wish to take on-demand screenshots within your tests, you should call the `captureScreenshotWithDescription` method.

Additional Considerations for iOS XCTest Tests

Device Farm supports any version of KIF that is based on OCUnit or XCTest.

Device Farm supports XCTest tests that are written in Objective-C and Swift.

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

Device Farm provides support for XCTest UI testing framework for iOS, written in both Objective-C and Swift. The following information describes how to use this test framework with Device Farm test types.

**Topics**
• What is XCTest UI Testing Framework? (p. 75)
• Prepare Your iOS XCTest UI Tests (p. 75)
• Upload Your iOS XCTest UI Tests (p. 75)
• Taking Screenshots in iOS XCTest UI Tests (p. 75)
• Additional Considerations for iOS XCTest UI Tests (p. 76)

What is XCTest UI Testing Framework?

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

Prepare Your iOS XCTest UI Tests

Before you upload iOS XCTest UI tests to Device Farm for testing, make sure that your iOS XCTest UI test runner bundle is contained within a properly formatted .ipa file. To create an .ipa file, you can 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, and it can be found in the Products directory for your project.

Upload Your iOS XCTest UI Tests

To run your iOS XCTest UI tests on a set of iOS devices in Device Farm, you upload your tests with the Device Farm console as follows:

2. In the list of projects, choose the option next to the project that you want to upload your tests to.
   
   **Tip**
   
   If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project that you want to upload your tests to.
   
   To create a new project, follow the instructions in Create a Project (p. 20).

3. If the Create a new run button is displayed, then choose it.
4. On the Choose your application page, choose Upload.
5. 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.

6. Choose Next step.
7. On the Configure a test page, choose XCTest UI, and then choose Upload.
8. Browse to and choose the .ipa file that contains your iOS XCTest UI test runner. In this .ipa file, make sure that the contents are organized according to the instructions as described in Prepare Your iOS XCTest UI Tests (p. 75).
9. Choose Next step, and then complete the remaining on-screen instructions to select the devices to run your tests on and to then start the run.

Taking Screenshots in iOS XCTest UI Tests

XCTest UI tests capture screenshots automatically for every step of your tests. These screenshots will be displayed in your Device Farm test report automatically. No additional code is required.
Additional Considerations for iOS XCTest UI Tests

Device Farm supports XCTest UI tests that are written in Objective-C and Swift.

Working with Custom Web App Tests in AWS Device Farm

Device Farm provides support for the following test types for working with Web applications.

- Appium Java JUnit (p. 76)
- Appium Java TestNG (p. 78)
- Appium Python (p. 80)

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 5 unmetered Android devices and 5 unmetered iOS devices, your Web test runs will use your unmetered devices.

Here is another example: suppose you have 5 unmetered Android devices and 0 unmetered iOS devices. If you select only Android devices for your Web run, your unmetered devices will be used. If you select both Android and iOS devices for your Web run, the billing method will be metered, and your unmetered devices will not be used.

Working with Appium Java JUnit for Web Applications and AWS Device Farm

Device Farm provides support for Appium Java JUnit for Web apps.

Topics
- What Is Appium Java JUnit? (p. 76)
- Version Information (p. 77)
- Prepare Your Web Application Appium Java JUnit Tests (p. 77)
- Upload Your Web Application Appium Java JUnit Tests (p. 77)
- Taking Screenshots in Web Application Appium Java JUnit Tests (p. 78)
- Additional Considerations for Web Application Appium Java JUnit Tests (p. 78)

What Is Appium Java JUnit?

Appium is an open-source tool for automating native, mobile web, and hybrid applications on platforms such as Web applications. For more information about Appium, see Introduction to Appium.
Version Information

Currently, Device Farm supports Appium versions 1.6.3 and 1.4.16 and Java 8.

Note
If desired, you can use the rules field in the create-device-pool command or the the CreateDevicePool API to specify the APPJUNIT_VERSION.

Prepare Your Web Application Appium Java JUnit Tests

Your Web Application Appium Java JUnit tests must be contained in a .zip file.

Upload Your Web Application Appium Java JUnit Tests

Use the Device Farm console to upload your tests:

2. If you see the AWS Device Farm console home page, choose Get started.
3. If you already have a project, you can upload your tests to an existing project or choose Create a new project.

   Tip
   If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project where you want to upload your tests. To create a new project, follow the instructions in Create a Project (p. 20).

4. If the Create a new run button is displayed, then choose it.
5. On the Choose your application page, choose Web application (the HTML5 button).

   Tip
   Name the run something that helps you easily identify a specific build of your app (for example, Beta-0.1. For more information, see Working with Test Runs (p. 23).
6. Provide a name for your run in the Run name field.

   Tip
   Name the run something that helps you easily identify a specific build of your app (for example, Beta-0.1. For more information, see Working with Test Runs (p. 23).
7. Configure your test by choosing Appium Java JUnit.
8. Next, choose **Upload** to upload your .zip file.

   Device Farm processes your .zip file before continuing.

9. Choose the Appium version you are using from the **Appium version** dropdown list.

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

### Taking Screenshots in Web Application Appium Java Junit Tests

You can take screenshots as part of your Web Application Appium Java JUnit tests.

When Device Farm runs your Appium Java JUnit test, the service sets the following system properties that describe the configuration of the Appium server with which you're communicating:

- `appium.screenshots.dir`: path to which the screenshots are saved
- `appium.server.address`: host address of the Appium server
- `appium.server.port`: port on which the Appium server is listening

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 automatically pulled into your Device Farm reports automatically. To view the screenshots, in the Device Farm console, choose the **Screenshots** section.

The following example shows how to use and consume the `SCREENSHOT_PATH` property to capture an Appium screenshot that is pulled into your Device Farm report.

```java
public boolean takeScreenshot(final String name) {
    String screenshotDirectory = System.getProperty("appium.screenshots.dir", System.getProperty("java.io.tmpdir", ""));
    File screenshot = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
    return screenshot.renameTo(new File(screenshotDirectory, String.format("%s.png", name)));
}
```

### Additional Considerations for Web Application Appium Java JUnit Tests

Device Farm does not modify Web application Appium Java JUnit tests.

### Working with Appium Java TestNG for Web Applications and AWS Device Farm

Device Farm provides support for Appium Java TestNG for Web applications.

**Topics**

- **What Is Appium Java TestNG?** (p. 79)
- **Version Information** (p. 79)
- **Prepare Your Web Application Appium Java TestNG Tests** (p. 79)
- **Upload Your Web Application Appium Java TestNG Tests** (p. 79)
- **Taking Screenshots in Web Application Appium TestNG Tests** (p. 80)
- **Additional Considerations for Web Application Appium TestNG Tests** (p. 80)
What Is Appium Java TestNG?

Appium is an open-source tool for automating native, mobile web, and hybrid applications on platforms such as Web applications. For more information about Appium, see Introduction to Appium.

Version Information

Currently, Device Farm supports Appium versions 1.6.3 and 1.4.16 and Java 8.

Note
If desired, you can use the rules field in the create-device-pool command or the the CreateDevicePool API to specify the APPIUM_VERSION.

Prepare Your Web Application Appium Java TestNG Tests

Your Web application Appium Java TestNG tests must be contained in a .zip file before you upload them to Device Farm.

Upload Your Web Application Appium Java TestNG Tests

Use the Device Farm console to upload your tests:

2. If you see the AWS Device Farm console home page, choose Get started.
3. If you already have a project, you can upload your tests to an existing project or choose Create a new project.
   
   Tip
   If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project where you want to upload your tests. To create a new project, follow the instructions in Create a Project (p. 20).
4. If the Create a new run button is displayed, then choose it.
5. On the Choose your application page, choose Web application (the HTML5 button).
6. Provide a name for your run in the Run name field.

   **Tip**
   Name the run something that helps you easily identify a specific build of your app (for example, Beta-0.1. For more information, see Working with Test Runs (p. 23).

7. Configure your test by choosing Appium Java TestNG.

8. Next, choose Upload to upload your .zip file.

   Device Farm processes your .zip file before continuing.

9. Choose the Appium version you are using from the Appium version dropdown list.

10. Choose Next step, and then complete the remaining on-screen instructions to select devices and start the run.

### Taking Screenshots in Web Application Appium TestNG Tests

You can take screenshots as part of your Web Application Appium TestNG tests.

When Device Farm runs your Appium Java JUnit test, the service sets the following system properties that describe the configuration of the Appium server with which you're communicating:

- `appium.screenshots.dir`: path to which the screenshots are saved
- `appium.server.address`: host address of the Appium server
- `appium.server.port`: port on which the Appium server is listening

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 automatically pulled into your Device Farm reports automatically. To view the screenshots, in the Device Farm console, choose the Screenshots section.

The following example shows how to use and consume the `SCREENSHOT_PATH` property to capture an Appium screenshot that is pulled into your Device Farm report.

```java
public boolean takeScreenshot(final String name) {
    String screenshotDirectory = System.getProperty("appium.screenshots.dir",
    System.getProperty("java.io.tmpdir", "");
    File screenshot = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
    return screenshot.renameTo(new File(screenshotDirectory, String.format("%s.png", name)));
}
```

### Additional Considerations for Web Application Appium TestNG Tests

Device Farm does not modify Web application Appium TestNG tests.

### Working with Appium Python for Web Applications and AWS Device Farm

Device Farm provides support for Appium Python for web applications.

**Topics**
What Is Appium Python?

Appium is an open-source tool for automating native, mobile web, and hybrid applications on platforms like web applications. For more information about Appium, see Introduction to Appium.

Version Information

Currently, Device Farm supports Appium versions 1.6.3 and 1.4.16 and Python version 2.7.6 (pip version 1.5.4).

Note

If desired, you can use the rules field in the create-device-pool command or the CreateDevicePool API to specify the APPIUM_VERSION.

Prepare Your Web Application Appium Python Tests

The Appium Python tests for your web application must be contained in a .zip file.

Build the Appium Python Test Package

The Appium Python test packages you upload to Device Farm must be in .zip format and contain all the dependencies of your test. The following instructions show you how to meet these requirements.

Note

The instructions below are based on Linux x86_64 and Mac. In the currently supported scheme, Device Farm requires that the packaging of your Appium Python tests be done on Linux x86_64 if your tests contain non-universal Python wheels dependencies. For the platform on which you execute a command, the wheels tools gather your .whl dependent files under the wheelhouse/ folder. When you execute the Python wheel command on any platform other than Linux x86_64, you would gather the flavor of a non-universal wheel dependency for that particular platform and may cause undesired effects. This would most likely lead to errors when executing your tests on Device Farm.

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.

   Tip

   - Do not create a Python virtualenv with the --system-site-packages option, because it will inherit packages from /usr/lib/pythonx.x/site-packages or wherever your global site-packages directory is. This can lead to you including dependencies in your virtual environment that are not needed by your tests.
   - You should also verify that your tests do not use dependencies that are dependent on native libraries, as these native libraries may or may not be present on the instance where these tests run.
2. Install `py.test` in your virtual environment.

   An example flow of creating a virtual environment using Python virtualenv and installing `pytest` in that virtual environment would look like the following:

   ```
   $ virtualenv workspace
   $ cd workspace
   $ source bin/activate
   $ pip install pytest
   ```

3. Store all Python test scripts under the `tests/` folder in your work space.

   ```
   # workspace
   ## tests/ (your tests go here)
   ```

4. Make sure you have `py.test` installed in your virtual environment and test cases are discoverable by the following command, which you should run from your virtual environment `workspace` folder.

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

   Make sure the output of `py.test` command shows you the tests that you want to execute on Device Farm.

5. Go to your work space and run the following command to generate the `requirements.txt` file:

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

6. Go to your work space and run the following command to generate the `wheelhouse/` folder:

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

7. You can use the following commands to 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 {} +
   ```

8. Zip the `tests/` folder, `wheelhouse/` folder, and the `requirements.txt` file into a single archive:

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

   Your work space will eventually look like this:

   ```
   # workspace
   ## tests/
   ## test_bundle.zip
   ## requirements.txt
   ## wheelhouse/
   ```

---

### Upload Your Web Application Appium Python Tests

Use the Device Farm console to upload your tests.

2. If you see the AWS Device Farm console home page, choose **Get started**.
3. If you already have a project, you can upload your tests to an existing project or choose **Create a new project**.

   ![Image of projects](image1.png)

   **Tip**
   If the list of projects is not displayed, then on the secondary navigation bar, for **Projects**, choose the name of the project where you want to upload your tests. To create a project, follow the instructions in Create a Project (p. 20).

4. If the **Create a new run** button is displayed, choose it.

5. On the **Choose your application** page, choose **Web application** (the HTML5 button).

   ![Image of new run](image2.png)

6. In the **Run name** field, type a name for your run.

   **Tip**
   Give the run a name that will help you identify a specific build of your app (for example, Beta-0.1). For more information, see Working with Test Runs (p. 23).

7. Choose **Appium Python** to configure your test.

8. Next, choose **Upload** to upload your .zip file.

   Device Farm processes your .zip file before continuing.

9. Choose the Appium version you are using from the **Appium version** dropdown list.

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

### Taking Screenshots in Web Application Appium Python Tests

You can take screenshots as part of your Appium Python tests for your web application.

When Device Farm runs your Appium Python test, the service 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.
The following example shows how to use and consume the SCREENSHOT_PATH property to capture an Appium screenshot that is pulled into your Device Farm report.

```python
screenshot_folder = os.getenv('SCREENSHOT_PATH', '')
screenshot_folder + "screenshot.png")
```

Additional Considerations for Web Application Appium Python Tests

Device Farm does not modify Appium Python tests for your web application.

Working with Built-in Tests in AWS Device Farm

Device Farm provides support for several automation test types.

**Built-in Test Types**

Built-in tests enable you to test your apps without writing scripts.

- Built-in: Explorer (Android) (p. 84)
- Built-in: Fuzz (Android and iOS) (p. 85)

**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 log in.

**Parameters**

- **Username** (Optional). Specifies a user name the explorer will use if it encounters a login screen within your app. If no user name is provided, Device Farm will not insert a user name.
- **Password** (Optional). Specifies a password the explorer will use if it encounters a login screen within your app. If no password is provided, Device Farm will not insert a password.

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

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

2. In the list of projects, choose the option next to the project where you want to run the built-in explorer test.

**Tip**

If the list of projects is not displayed, then on the secondary navigation bar, for **Projects**, choose the name of the project where you to run the built-in explorer test.
To create a new project, follow the instructions in Create a Project (p. 20).

3. If the Create a new run button is displayed, then choose it.
4. On the Choose your application page, choose Upload.
5. Browse to and choose your app file where you want to run the built-in explorer test.
6. Choose Next step.
8. Choose Next step, and then complete the remaining on-screen 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. In the list of projects, choose the option next to the project where you want to run the built-in fuzz test.

   Tip
   If the list of projects is not displayed, then on the secondary navigation bar, for Projects, choose the name of the project where you want to run the built-in fuzz test.
   To create a new project, follow the instructions in Create a Project (p. 20).
3. If the Create a new run button is displayed, then choose it.
4. On the Choose your application page, choose Upload.
5. Browse to and choose your app file where you want to run the built-in fuzz test.
6. Choose Next step.
8. 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 1 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.
9. Choose Next step, and then complete the remaining on-screen instructions to select devices and start the run.
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 in order 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 (for example, Android or iOS devices). For more information, see Device Farm Pricing.

We currently offer a subset of our devices for remote access testing. We continue to add new devices to the device pool all the time.

Device Farm captures video of each remote access session and generates logs of activity taking place 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.

- Create a Session (p. 86)
- Use a Session (p. 87)
- Get Session Results (p. 88)

Create a Remote Access Session in AWS Device Farm

For information about remote access sessions, see Sessions (p. 14).

- Prerequisites (p. 86)
- Create a Test Run with the Device Farm Console (p. 86)
- Next Steps (p. 87)

Prerequisites

- Create a project in Device Farm. Follow the instructions in Create a Project (p. 20), and then return to this page.

Create a Session with the Device Farm Console

2. If you see the AWS Device Farm console home page, choose Get started.
3. Choose a project from the Projects drop-down or choose Create a new project.
4. Choose the **Remote access** tab.
5. Choose the **Start a new session** button.
6. Choose a device for your session. You can choose from the list of available devices or search for a device using the fields at the top of the list. You can search by:
   - Name
   - Platform
   - Operating system
   - Form factor
7. Type a name for the session in **Session name**.
8. Choose **Confirm and start session** to begin the session.

**Next Steps**

Device Farm will start 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 from that time. After that, Device Farm ends the session automatically. Your account will be charged for the idle time, however.

After the session begins, you can start interacting with the device in the web browser.

**Use a Remote Access Session in AWS Device Farm**

For information about sessions, see Sessions (p. 14).

- **Prerequisites** (p. 87)
- **Use a Session in the Device Farm Console** (p. 87)
- **Next Steps** (p. 88)
- **Tips and Tricks** (p. 88)

**Prerequisites**

- Create a session. Follow the instructions in **Create a Session** (p. 86), and then return to this page.

**Use a Session in the Device Farm Console**

As soon the device you requested for a remote access session becomes available, the console displays the device screen. The session has a maximum length of 60 minutes. The time remaining in the session appears below the menu bar on the right side of the console.

**Installing an Application**

To install an application on the session device, in **Install applications**, choose **Upload**, and then choose either the .apk file for an Android application or the .ipa file for an iOS application you want to install. Applications you run in a remote access session don’t require any test instrumentation or provisioning.
Note
At present, we do not display a confirmation once an app is finished installing. Try interacting with the app icon to see if the app is ready to use.

Tip
When you upload an app, there is 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 using your mouse for touch and the device's on-screen keyboard. For Android devices, there are buttons in View controls that function just as the Home and Back buttons on an Android device do. For iOS devices, there is a Home button that functions just 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 you are using.

Next Steps
Device Farm will continue the session until you stop it manually or until the sixty-minute time limit is reached. To end the session, choose the Stop session button. After it stops, you can access captured video and logs generated by the session. For more information about getting session results, see Get Session Results (p. 88).

Tips and Tricks
In some regions, you may experience performance issues with the remote access session. This is due in part to latency in some regions. If you experience performance issues, give the remote session a chance to catch up before interacting with the app again.

Get Results of a Remote Access Session in AWS Device Farm
For information about sessions, see Sessions (p. 14).

- Prerequisites (p. 88)
- Viewing Session Details (p. 89)
- Downloading Session Video or Logs (p. 89)

Prerequisites
- Complete a session. Follow the instructions in Use a Session (p. 87), and then return to this page.
Viewing Session Details

When a remote access session ends, the Device Farm console displays a table containing details about activity that took place during the session. For more information about the details displayed, see Analyzing Log Information (p. 40).

To return to the details of a session at a later time:

1. Choose the project you want to review from the Project drop-down list.
2. Choose the session you want to review from the list or View all sessions from the Runs & sessions drop-down list then choose the session you want to review from the list of sessions displayed.

Downloading Session Video or Logs

When a remote access session ends, the Device Farm console provides access to a video capture of the session along with logs about activity in the session. 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 Direct Device Access in AWS Device Farm

Direct device access is a Device Farm feature that allows you to:

- Run and debug Android and iOS apps from your preferred IDE, such as Xcode or Android Studio, on remote devices.
- Run test automation, using the framework of your choice, against multiple private Android and iOS remote devices concurrently.

With direct device access, your Device Farm private devices are accessible using platform tools (such as ADB for Android and instruments for iOS) on your local machine. The devices are visible to you as if they were connected directly to your computer.

**Note**
Direct device access is only available for customers who have Device Farm private devices. A private device is a physical device that Device Farm deploys on your behalf in an Amazon data center. This device is exclusive to your account.

You interact with a specific device in real time by creating a remote debugging session with the device. The device is hosted in a Device Farm app window. The session displays the single device that you select when you start the session. One device can be hosted in each session. You can run multiple remote debugging sessions in parallel.

Only Device Farm private devices can be used for direct device access. For more information about using private devices, see Device Farm Pricing.

Device Farm direct device access supports any local test frameworks that you choose, and it supports most third-party test tools.

**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 debugging session.

**Topics**
- Create a Remote Debugging Session in AWS Device Farm (p. 90)
- Use a Remote Debugging Session in AWS Device Farm (p. 103)
- AWS Device Farm Direct Device Access FAQ (p. 118)

Create a Remote Debugging Session in AWS Device Farm

**Topics**
- Prerequisites (p. 91)
- Replace Your Copy of ADB (p. 91)
- Launch the Device Farm App and Start a Remote Debugging Session (p. 94)
- How to Stop a Remote Debugging Session (p. 100)
Prerequisites

1. Complete the steps in Setting Up (p. 4), which include signing up for an AWS account, creating or using an IAM user in the AWS account, and giving the IAM user permission to access Device Farm.

2. Download the Device Farm app from one of the following locations:
   - Windows: https://s3-us-west-2.amazonaws.com/prod-us-west-2-system-resources/AWSDeviceFarmClient/AWS_Device_Farm.exe. SHA-256: ec34f8c7f98dc51ac6529f1fd76b7aa63695c5f3d8ecfeb8d883588fd0d6648a
   - MacOS: https://s3-us-west-2.amazonaws.com/prod-us-west-2-system-resources/AWSDeviceFarmClient/AWS_Device_Farm.dmg. SHA-256: 41123da58f9f688562c11ff1add37dd1582f51fadb13cb39c0bdf8d977209c956

3. Install the Device Farm app. On Macintosh, you'll need to drag it into your Applications folder.

On Windows, you'll need to double-click the .exe file to install the app. The installer creates a shortcut on the desktop and one in the Start menu.

4. If you're using Android Studio or ADB, you'll need to replace your copy of ADB with the version that Device Farm currently supports.

Replace Your Copy of ADB

1. Download the Device Farm-supported version of ADB.
Replace Your Copy of ADB

- MacOS: https://s3-us-west-2.amazonaws.com/gamma-us-west-2-devicefarm-adb/adb_32_mac.zip
  SHA-256: 6eccd1b02195d2eaa343c6fddd244ce31da03faa9f51e317e31e4db63dcafb

2. Extract the files from the downloaded .zip file. The MacOS version has one file. The Windows version has three files (adb.exe, AdbWinApi.dll, and AdbWinUsbApi.dll).

3. If you are using Android Studio, open Android Studio and perform the following steps:

   Replacing ADB for Android Studio

1. In the Tools menu, choose Android, and then choose Android SDK Manager.

2. Choose the SDK Platforms tab.

3. Navigate to the folder specified in the Android SDK location field. You can do this by copying the folder path into an open File Explorer window.

4. Open the platform-tools folder.

5. If you're on MacOS, make a backup of the original ADB binary in the platform-tools folder and copy the Device Farm-supported one into the platform-tools folder.
If you’re on Windows, make backups of the original `adb.exe`, `AdbWinApi.dll`, and `AdbWinUsbApi.dll` in the `platform-tools` folder, and copy the Device Farm-supported one into the `platform-tools` folder.

4. If you’re accessing devices using ADB from the command line, perform the following steps on Macintosh or Windows:

**Replacing command-line ADB on Macintosh**

1. Open a terminal window and enter `which adb`.
2. Copy the location returned by the `which` command and navigate to it by using *File Explorer*.
3. Make a backup of the original ADB library and copy the Device Farm-supported one into the folder.
Replacing command-line ADB on Windows

1. Open a command-prompt window and enter `where adb`.

2. Copy the location returned by the `which` command and navigate to it by using **File Explorer**.

3. Make backups of the original `adb.exe`, `AdbWinApi.dll`, and `AdbWinUsbApi.dll`, and copy the Device Farm-supported one into the folder.

Launch the Device Farm App and Start a Remote Debugging Session

1. Double-click the app desktop shortcut to open it.

2. Enter your AWS credentials if prompted to do so.

When you open the app, it looks for stored AWS credentials in two locations:

- If you have already installed the AWS CLI on your computer, your AWS credentials are stored in your **AWS CLI configuration file** located at `~/.aws/credentials` on Linux and OS X systems or `C:\Users\USERNAME\.aws\credentials` on Windows systems.

- Otherwise, the first time you open the app, you'll be prompted to enter your AWS credentials. When you do that, your credentials are stored in the app.

**Note**

We recommend that you create an IAM user with full Device Farm access.
3. (OPTIONAL) If you have an existing Device Farm project, you can choose the project from the Projects drop-down menu. The default project is Remote Debug.

4. Choose the Start a new session button.

5. Choose devices for your remote debugging sessions. You can choose multiple devices. Each device will get its own session. You can choose from the list of available devices or search for a device. You can filter the list by platform or type.

You can also choose the interaction mode for iOS devices:
Launch the Device Farm App and Start a Remote Debugging Session

- Interactive: You can interact with the iOS device by viewing, touching, and rotating the screen. You **cannot** run XCUItest framework-based tests in this mode.
- Video Only: You can view the screen but cannot touch or rotate it. You **can** run XCUItest framework-based tests and watch the screen in this mode.
- No Video: You are connected to the device but cannot interact with it or view the screen. This mode has the fastest test execution speed. You **can** run XCUItest framework-based tests in this mode.

6. Choose the **Start session with n devices** button.

Device Farm will start the session as soon as the requested device is available, typically within a few minutes. The requested device appears in the **Pending Sessions** list until the session starts.
To determine the status of a pending session, hover over the session entry in the **Pending Sessions** list. Pending session status can be:

- **Scheduling** - Device Farm is scheduling a session
- **Pending** - the session is pending
- **Preparing** - Device Farm is getting the device ready
- **Waiting for available device** - waiting for a device to become available

To cancel the session request, choose **Cancel request**.
Launch the Device Farm App and Start a Remote Debugging Session

If the session starts successfully, it moves from the Pending Sessions list to the Active Sessions list. An active session is one with a device that you can connect to.

Once the session appears in the Active Sessions list, you can connect to the device and debug it. See Use a Remote Debugging Session in AWS Device Farm (p. 103).

Once a session is completed, it moves to the Completed Sessions list.
Note
If a session fails, it moves directly from the Pending Sessions list to the Completed Sessions list.

To determine the status of a completed session, hover over the session entry in the Completed Sessions list. Completed session status can be:

- **Stopping** - the session is in the process of being stopped
- **Passed** - the session has completed successfully
- **Errored** - the session encountered a problem

To view the results of a session in the Completed Sessions list, choose View report to see the results in the Device Farm console.
How to Stop a Remote Debugging Session

To stop a session that is still pending, choose **Cancel request**.

To stop an active session, choose **Terminate**.
After a session starts (becomes active), if you close the Device Farm app without stopping the session or if the connection between the app and the Internet is lost, the session remains active for five minutes from that time. After that, Device Farm ends the session automatically. Your account will be charged for the idle time, however.

**Tips and Tricks**

At the bottom of the Device Farm app screen are the following dropdown menus:

- The **AWS CLI Profile** menu can be used to switch from the default profile to an **AWS CLI profile** that you have configured.
• The Project menu can be used to switch from the default Remote Debug project to another project that you previously created.
Use a Remote Debugging Session in AWS Device Farm

Topics

- Prerequisites (p. 103)
- View and Interact with Devices Remotely (p. 103)
- Connect to Remote Devices (p. 105)
- Switching Between Remote and Local Devices (p. 106)
- Debugging Your App Remotely with Android Studio (p. 106)
- Debugging Your App Remotely with Xcode (p. 107)
- Using Test Automation (p. 108)
- Debugging Web Apps (p. 113)

Prerequisites

- Create a remote debugging session. Follow the instructions in Create a Remote Debugging Session in AWS Device Farm (p. 90), and then return to this page.

View and Interact with Devices Remotely

Once the device you requested for a remote debugging session becomes available, choose Interact to open the device window to interact with the device (but not debug it). You can interact remotely with the device as you would the actual physical device using your mouse for touch and the device's on-screen keyboard or the physical keyboard on your laptop.

The session has a default timeout value of 60 minutes. You can change this value. For more information, see Set the Execution Timeout for Test Runs in AWS Device Farm.
The device window allows you to view and control the device via streaming video.

**Note**
The device won't be visible in Android Studio or Xcode until you connect to it.
Connect to Remote Devices

You can connect to your remote devices and debug them locally, as if they were connected directly to your computer. To connect to remote devices, choose the Connect button. This creates an ssh tunnel to the EC2 instance that hosts your devices, and you can now use them locally using your preferred debugging tools. You will lose any local device connections that you already had.

Now the devices are visible to your IDE (for example, Android Studio or Xcode) and other tools (such as ADB for Android and Instruments for iOS).

**Note**
The first time you connect to the ssh tunnel on Macintosh, you'll need to enter the local machine's Macintosh root password.

If you encounter a "Connect to remote devices failed" error, you can troubleshoot the error as follows:

1. In the Advanced menu, choose View Logs.

2. In the logs, look for the following line, where `host_ip` is the host IP address:

   ```
   [date time] [info] Creating ssh tunnel to host address: host_ip
   ```

   Copy the host IP address.

3. Open a terminal window and run the following command, where `host_ip` is the host IP address you copied from the logs:

   ```
   ssh -v -N -oStrictHostKeyChecking=no -oUserKnownHostsFile=/dev/null -L 5037:localhost:5038 -i ~/.aws/devicefarm/prikey.pem device-farm@host_ip
   ```
4. Check the command output to see why the connection attempt failed.

**Switching Between Remote and Local Devices**

To disconnect from a remote device, choose the **Disconnect** button.

When you disconnect from your remote devices, you will no longer see them in your debugger. You will only see your local devices.

To reconnect to your remote devices, choose **Connect**.

---

**Debugging Your App Remotely with Android Studio**

Once you’ve connected to a remote device in the Device Farm app, you can begin to debug your app with Android Studio as follows:

1. Launch Android Studio.
2. Choose the **Run** button to run your app. A dialog box will appear that lists your connected devices. Your remote Device Farm devices will appear as if they were local devices physically connected to your computer.
3. Connect to a device and debug your app.
Debugging Your App Remotely with Xcode

Once you've connected to a remote device in the Device Farm app, you can begin to debug your app with Xcode as follows:

1. Launch Xcode. Your remote Device Farm devices will appear in device selection menus as if they were local devices physically connected to your computer.
2. Connect to a device and debug your app.

**Note**
Following are some guidelines for remote debugging by using Xcode:

1. When using the remote iOS device for the first time, you may see the warning "Processing symbol files" in Xcode. This will take some time and prevent you from running or debugging your app on the remote device. Wait until it is done before you use your remote device.
2. When you connect to a remote device, you might not see the device in Xcode if Xcode was open before you connected to the first remote device. If this happens, restart Xcode, and you should be able to see the device in Xcode.
3. If you get a failure interacting with the device using the libimobiledevice library, open the video stream for the device and accept the trust pop-up if one appears. You should then be able to list the device using the libimobiledevice library.

4. In order to launch an app on the iOS device, you can either use the Xcode-managed provisioning profile or use your own provisioning profile. If you want to use your own provisioning profile, you should add the udid of the device to your provisioning profile. You can get the udid of the device by hovering over the device name in the **Active Sessions** list in the Device Farm client app. After adding the udid of the device to your provisioning profile, update your Xcode to use the latest provisioning profile.

### Using Test Automation

Once you've connected to your remote devices, they should appear as if they were connected locally. You can run your test in concurrent fashion against multiple Android and iOS devices using your preferred test automation framework, such as Calabash, Appium, or Spoon.
To verify that your device is connected, use a command such as `adb` or `instrument`.

**Verifying Device Connection Using ADB for Android**

To verify the connection, type `adb devices` at the command prompt. Once you can list the devices locally, you should be able to run test automation as if the devices are connected locally.

**Verifying Device Connection Using `instruments` for iOS**

To verify the connection, type `instruments -s` at the command prompt. Once you can list the devices locally, you should be able to run test automation as if the devices are connected locally.
Using Spoon Test Automation to Debug Multiple Android Devices

Here is a screenshot that shows test automation running on multiple devices on Spoon.
Using XCode-Build Automation to Debug Multiple iOS Devices

Here is a screenshot that shows test automation running on multiple iOS devices on XCode-Build.
## Remote Debugging Sessions

### Active Sessions

<table>
<thead>
<tr>
<th>Session Name</th>
<th>Platform</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple iPhone SE</td>
<td>iOS</td>
<td>10.3.2</td>
</tr>
<tr>
<td>Apple iPhone 6 Plus</td>
<td>iOS</td>
<td>10.3.2</td>
</tr>
<tr>
<td>Apple iPhone 6</td>
<td>iOS</td>
<td>10.3.2</td>
</tr>
</tbody>
</table>

### Pending Sessions

<table>
<thead>
<tr>
<th>Session Name</th>
<th>Platform</th>
<th>Version</th>
</tr>
</thead>
</table>

### Completed Sessions

<table>
<thead>
<tr>
<th>Session Name</th>
<th>Platform</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple iPhone SE</td>
<td>iOS</td>
<td>10.3.2</td>
</tr>
<tr>
<td>Apple iPhone 6 Plus</td>
<td>iOS</td>
<td>10.3.2</td>
</tr>
<tr>
<td>Apple iPhone 6</td>
<td>iOS</td>
<td>10.3.2</td>
</tr>
</tbody>
</table>

AWS CLI Profile: `default`
Debugging Web Apps

In addition to native apps, you can also debug web apps on remote devices.

Debugging Web Apps on Android

Once you've connected to a remote Android device in the Device Farm app, you can begin to debug your web app as follows:

1. Open the Chrome browser on your computer and navigate to chrome://inspect#devices to see if you're connected to the remote device.

2. In the Device Farm app device window, perform the following steps on your remote device:
   1. Choose Interact to get a device window.
   2. Open the Chrome browser on the remote device.
   3. Open the UI for your web app, for example, www.google.com.

3. In the Chrome browser on your computer, you should now see that Chrome is also running on the remote device:
4. In the chrome://inspect#devices window on your computer, choose Inspect to open Chrome developer tools.

Here's what the tools look like:
5. Start debugging your app.

Debugging Web Apps on iOS

Once you've connected to a remote iOS device in the Device Farm app, you can begin to debug your web app as follows:

1. Open the Safari browser on your computer.
2. In the Develop menu, choose the entry that corresponds to your remote device.
3. In the Device Farm app device window, perform the following steps on your remote device:

1. Choose **Interact** to get a device window.
2. Open the Safari browser.
3. Open the UI for your web app, for example, www.google.com.

4. In the Safari window on your computer, in the **Develop** menu, open the UI for your web app by choosing the entry corresponding to the remote device.
This will open a **Web Inspector** showing the source code for the web page on the remote device.
5. Start debugging your app.

**AWS Device Farm Direct Device Access FAQ**

**Topics**

- Do I need to install additional software on my machine to use this feature? (p. 119)
- Can I use this feature in the Device Farm console? (p. 119)
- Is direct device access available for public fleets of devices? (p. 119)
- Are both iOS and Android supported? (p. 119)
- Can I connect to multiple Android devices from my local machine? (p. 119)
- Can I connect to multiple iOS devices from my local machine? (p. 119)
- Can I use ADB forward and ADB reverse on Device Farm devices in my remote debugging sessions? (p. 119)
- Can I debug and run on cloud devices from Android Studio, Xcode, Chrome Developer Tools, and Safari Web Inspector? (p. 120)
- What test frameworks are supported for Android devices? (p. 120)
- What test frameworks are supported for iOS devices? (p. 120)
- Is there a command-line interface (CLI) client to check out devices? (p. 120)
- How is direct device access different from remote access? (p. 120)
- What interaction modes are available for iOS devices? (p. 120)
- How Do I Troubleshoot a Direct Device Access Session? (p. 120)
Do I need to install additional software on my machine to use this feature?

Yes. You can download the Device Farm app from one of the following locations:

- **Windows**: [https://s3-us-west-2.amazonaws.com/prod-us-west-2-system-resources/AWSDeviceFarmClient/AWS_Device_Farm.exe](https://s3-us-west-2.amazonaws.com/prod-us-west-2-system-resources/AWSDeviceFarmClient/AWS_Device_Farm.exe). SHA-256: ec34f8c7f98dc51ac6529f1fd76b7aa63695c5f3d8ecfeb8d883588fd0d6648a
- **MacOS**: [https://s3-us-west-2.amazonaws.com/prod-us-west-2-system-resources/AWSDeviceFarmClient/AWS_Device_Farm.dmg](https://s3-us-west-2.amazonaws.com/prod-us-west-2-system-resources/AWSDeviceFarmClient/AWS_Device_Farm.dmg). SHA-256: 41123da58f9f688562c11ff1add37dd1582f51fdb13cb39c0bdf8d977209e956

For more information, see Create a Remote Debugging Session in AWS Device Farm (p. 90).

Can I use this feature in the Device Farm console?

No. You must download and run the Device Farm app. However, we are planning to add command-line support soon.

Is direct device access available for public fleets of devices?

Direct device access is available only for customers who have AWS Device Farm private devices. For more information about using private devices, see Device Farm Pricing.

Are both iOS and Android supported?

Yes.

Can I connect to multiple Android devices from my local machine?

Yes.

Can I connect to multiple iOS devices from my local machine?

Yes.

Can I use ADB forward and ADB reverse on Device Farm devices in my remote debugging sessions?

Yes. All ADB commands are supported.
Can I debug and run on cloud devices from Android Studio, Xcode, Chrome Developer Tools, and Safari Web Inspector?

Yes.

What test frameworks are supported for Android devices?

Once you connect to your devices, you can run any framework you choose.

What test frameworks are supported for iOS devices?

Once you connect to your devices, you can run any framework you choose.

Is there a command-line interface (CLI) client to check out devices?

We do not currently have a CLI client. However, we are planning to add CLI support soon.

How is direct device access different from remote access?

Remote access allows you to swipe, gesture, and interact with a device through your web browser in real time in order to test functionality and reproduce customer issues. In addition to the features that remote access provides, with direct device access your Device Farm private devices are accessible using platform tools (such as ADB for Android and instruments for iOS) on your local machine. You can run and debug Android and iOS apps from IDEs such as Xcode and Android Studio. The devices are visible on your local machine as if they were physically connected to your local machine.

What interaction modes are available for iOS devices?

There are three interaction modes for iOS devices in Direct Device Access sessions:

- INTERACTIVE: You can interact with the iOS device by viewing, touching, and rotating the screen. You cannot run XCUITest framework-based tests in this mode.
- NO_VIDEO: You are connected to the iOS device but cannot interact with it or view the screen. This mode has the fastest test execution speed. You can run XCUITest framework-based tests in this mode.
- VIDEO_ONLY: You can view the iOS device's screen but cannot touch or rotate it. You can run XCUITest framework-based tests and watch the screen in this mode.

How Do I Troubleshoot a Direct Device Access Session?

To view client app logs for your sessions, go to the menu bar at the top of the Device Farm app screen and choose View Logs from the Advanced menu.
Why Don't I See My Local Devices? How Do I Switch from Debugging Remote Devices to Debugging Local Ones?

When you connect to your Device Farm remote devices, you lose any connections to your local devices that are physically connected to your computer. When you disconnect from remote devices, you can see your local devices again.

To switch from remote devices to local ones, choose **Disconnect**.
Logging AWS Device Farm API Calls by Using AWS CloudTrail

Device Farm is integrated with CloudTrail, a service that captures API calls made by or on behalf of Device Farm in your AWS account and delivers the log files to an Amazon S3 bucket you specify. Examples of these API calls include creating a new project or run in Device Farm. CloudTrail captures API calls from the Device Farm console or the Device Farm APIs. Using the information collected by CloudTrail, you can determine which request was made to Device Farm, the source IP address from which the request was made, who made the request, when it was made, and so on. To learn more about CloudTrail, including how to configure and enable it, see the AWS CloudTrail User Guide.

Device Farm Information in CloudTrail

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. 124) and the API Reference (p. 126). For example, calls to create a new project or run in Device Farm generate entries in CloudTrail log files.

Every log entry contains information about who generated the request. The user identity information in the log helps you determine whether the request was made with root or IAM user credentials, with temporary security credentials for a role or federated user, or by another AWS service. For more information, see the userIdentity field in the CloudTrail Event Reference.

You can store log files in your bucket for as long as you want, but you can also define Amazon S3 lifecycle rules to archive or delete log files automatically. By default, your log files are encrypted by using Amazon S3 server-side encryption (SSE).

If you want to take quick action upon log file delivery, you can have CloudTrail publish Amazon SNS notifications when new log files are delivered. For more information, see Configuring Amazon SNS Notifications.

You can also aggregate Device Farm log files from multiple AWS regions and multiple AWS accounts into a single Amazon S3 bucket. For more information, see Aggregating CloudTrail Log Files to a Single Amazon S3 Bucket.

Understanding Device Farm Log File Entries

CloudTrail log files can contain one or more log entries where each entry is made up of multiple JSON-formatted events. A log entry represents a single request from any source and includes information about the requested action, any parameters, the date and time of the action, and so on. The log entries are not guaranteed to be in any particular order. That is, they are not an ordered stack trace of the public API calls.

The following example shows a CloudTrail log entry that demonstrates the Device Farm ListRuns action:
Understanding Device Farm Log File Entries

```
{
    "Records": [
        {
            "eventVersion": "1.03",
            "userIdentity": {
                "type": "Root",
                "principalId": "AKIAI44QH8DHBEXAMPLE",
                "arn": "arn:aws:iam::123456789012:root",
                "accountId": "123456789012",
                "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
                "sessionContext": {
                    "attributes": {
                        "mfaAuthenticated": "false",
                        "creationDate": "2015-07-08T21:13:35Z"
                    }
                },
            },
            "eventTime": "2015-07-09T00:51:22Z",
            "eventSource": "devicefarm.amazonaws.com",
            "eventName": "ListRuns",
            "awsRegion": "us-west-2",
            "sourceIPAddress": "203.0.113.11",
            "userAgent": "example-user-agent-string",
            "requestParameters": {
            "responseElements": {
                "runs": [
                    {
                        "created": "Jul 8, 2015 11:26:12 PM",
                        "name": "example.apk",
                        "completedJobs": 2,
                        "counters": {
                            "stopped": 0,
                            "warned": 0,
                            "failed": 0,
                            "passed": 4,
                            "skipped": 0,
                            "total": 4,
                            "errored": 0
                        },
                        "type": "BUILTIN_FUZZ",
                        "status": "RUNNING",
                        "totalJobs": 3,
                        "platform": "ANDROID_APP",
                        "result": "PENDING"
                    }
                ],
            }...additional entries ...
        }
    ]
}
```
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.
API Reference for AWS Device Farm

Use HTTP to call the Device Farm APIs. For more information, see the AWS Device Farm API Reference.
Troubleshooting Device Farm Errors

In this section, you will find error messages and procedures to help you fix common problems with Device Farm.

Topics
- Troubleshooting Android Application Tests in AWS Device Farm (p. 127)
- Troubleshooting Appium Java JUnit Tests in AWS Device Farm (p. 131)
- Troubleshooting Appium Java JUnit Web Application Tests in AWS Device Farm (p. 136)
- Troubleshooting Appium Java TestNG Tests in AWS Device Farm (p. 142)
- Troubleshooting Appium Java TestNG Web Applications in AWS Device Farm (p. 146)
- Troubleshooting Appium Python Tests in AWS Device Farm (p. 149)
- Troubleshooting Appium Python Web Application Tests in AWS Device Farm (p. 156)
- Troubleshooting Calabash Tests in AWS Device Farm (p. 163)
- Troubleshooting Instrumentation Tests in AWS Device Farm (p. 169)
- Troubleshooting IOS Application Tests in AWS Device Farm (p. 173)
- Troubleshooting UI Automator Tests in AWS Device Farm (p. 180)
- Troubleshooting XCTest Tests in AWS Device Farm (p. 181)
- Troubleshooting XCTest UI Tests in AWS Device Farm (p. 184)

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. 43).

**ANDROID_APP_AAPT_DEBUG_BADGING_FAILED**

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

**Warning**

We could not extract information about your application. Please verify that the application is valid by running the command `aapt debug badging <path to your test package>`, and try again after the command does not print any error.

During the upload validation process, AWS Device Farm parses out information from the output of an `aapt debug badging <path to your package>` command.

Make sure that you can run this command on your Android application successfully. In the following example, the package's name is `app-debug.apk`.

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

  ```
  # aapt debug badging app-debug.apk
  ```

A valid Android application package should produce output like the following:

```
package: name='com.amazon.aws.adf.android.referenceapp' versionCode='1'
  versionName='1.0' platformBuildVersionName='5.1.1-1819727'
sdkVersion:'9'
application-label:'ReferenceApp'
application: label='ReferenceApp' icon='res/mipmap-mdpi-v4/ic_launcher.png'
application-debuggable
launchable-activity:
  name='com.amazon.aws.adf.android.referenceapp.Activities.MainActivity'
  label='ReferenceApp' icon='' uses-feature: name='android.hardware.bluetooth'
uses-implied-feature: name='android.hardware.bluetooth' reason='requested
  android.permission.BLUETOOTH permission, and targetSdkVersion > 4'
main
supports-screens: 'small' 'normal' 'large' 'xlarge'
supports-any-density: 'true'
locales: '---_--'
densities: '160' '213' '240' '320' '480' '640'
```

For more information, see Working with Android Tests in AWS Device Farm (p. 43).

**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:

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

  A valid Android application package should produce output like the following:

  ```xml
  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. 43).

**ANDROID_APP_SDK_VERSION.VALUE MISSING**

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

Warning
We could not find the SDK version value in your application. Please verify that the application is valid by running the command `aapt debug badging <path to your test package>`, and try again after finding the SDK version value behind the keyword `sdkVersion`.

During the upload validation process, AWS Device Farm parses out the SDK version value from the output of an `aapt debug badging <path to your package>` command.

Make sure that you can run this command on your Android application and find the package name value successfully. In the following example, the package's name is `app-debug.apk`.

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

  ```bash
  $ aapt debug badging app-debug.apk | grep "sdkVersion"
  ```

  A valid Android application package should produce output like the following:

  ```text
  sdkVersion:'9'
  ```

  For more information, see Working with Android Tests in AWS Device Farm (p. 43).

**ANDROID_APP_AAPT DUMP XMLTREE FAILED**

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

Warning
We could not find the valid AndroidManifest.xml in your application. Please verify that the test package is valid by running the command `aapt dump xmltree <path to your test package>` AndroidManifest.xml, and try again after the command does not print any error.
During the upload validation process, AWS Device Farm parses out information from the XML parse tree for an XML file contained within the package using the command `aapt dump xmltree <path to your package> AndroidManifest.xml`.

Make sure that you can run this command on your Android application successfully. In the following example, the package's name is `app-debug.apk`.

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

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

A valid Android application package should produce output like the following:

```
E: manifest (line=2)
  A: android:versionCode(0x0101021b)=(type 0x10)0x1
  A: android:versionName(0x0101021c)="1.0" (Raw: "1.0")
  A: package="com.amazon.aws.adf.android.referenceapp" (Raw: "com.amazon.aws.adf.android.referenceapp")
  A: platformBuildVersionCode=(type 0x10)0x16 (Raw: "22")
  A: platformBuildVersionName="5.1.1-1819727" (Raw: "5.1.1-1819727")
E: uses-sdk (line=7)
  A: android:minSdkVersion(0x0101020c)=(type 0x10)0x9
  A: android:targetSdkVersion(0x01010270)=(type 0x10)0x16
E: uses-permission (line=11)
  A: android:name(0x01010003)="android.permission.INTERNET" (Raw: "android.permission.INTERNET")
E: uses-permission (line=12)
  A: android:name(0x01010003)="android.permission.CAMERA" (Raw: "android.permission.CAMERA")
```

For more information, see Working with Android Tests in AWS Device Farm (p. 43).

**ANDROID_APP_DEVICE_ADMIN_PERMISSIONS**

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

**Warning**

We found that your application requires device admin permissions. Please verify that the permissions are not required by run the command `aapt dump xmltree <path to your test package> AndroidManifest.xml`, and try again after making sure that output does not contain the keyword `android.permission.BIND_DEVICE_ADMIN`.

During the upload validation process, AWS Device Farm parses out permission information from the XML parse tree for an XML file contained within the package using the command `aapt dump xmltree <path to your package> AndroidManifest.xml`.

Make sure that your application does not require device admin permission. In the following example, the package's name is `app-debug.apk`.

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

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

You should find output like the following:

```
```

API Version 2015-06-23

130
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:

   ```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)
```
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
```

For more information, see Appium Java JUnit (p. 43) or Appium Java JUnit (p. 59).
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:

   ```
   .
   `|-- acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built from the ./src/main directory)
   `|-- acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything built from the ./src/test directory)
   `|-- zip-with-dependencies.zip (this .zip file contains all of the items)
   `|-- dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
   `|   |`-- com.some-dependency.bar-4.1.jar
   `|   |`-- com.another-dependency.thing-1.0.jar
   `|   `-- joda-time-2.7.jar
   `|`-- log4j-1.2.14.jar
   ```

   For more information, see Appium Java JUnit (p. 43) or Appium Java JUnit (p. 59).

---

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`.

   ```
   .
   `|-- 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 Appium JUnit (p. 43) or Appium JUnit (p. 59).

**APPium_JAVA_JUNIT_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TESTS_JAR**

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

**Warning**

We could not find a class file within the tests JAR file. Please unzip your test package and then unjar the tests JAR file, verify that at least one class file is within the JAR file, and try again.

In the following example, the package's name is **zip-with-dependencies.zip**.

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

   ```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. 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)
   | `- 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 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 Appium Java JUnit (p. 43) and Appium Java JUnit (p. 59)
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:

   ```
   .
   |— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built from the ./src/main directory)
   |— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything built from the ./src/test directory)
   |— zip-with-dependencies.zip (this .zip file contains all of the items)
   `— dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
   |— junit-4.10.jar
   |— com.some-dependency.bar-4.1.jar
   |— com.another-dependency.thing-1.0.jar
   |— joda-time-2.7.jar
   `— log4j-1.2.14.jar
   ```

Note
Your tests may not execute correctly if the JUnit version specified in your test package is lower than the minimum version 4.10 we support.

For more information, see Appium Java JUnit (p. 43) or Appium Java JUnit (p. 59).

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.

**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:
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)
`-- dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
    |-- com.some-dependency.bar-4.1.jar
    |-- com.another-dependency.thing-1.0.jar
    |-- joda-time-2.7.jar
    `-- log4j-1.2.14.jar
```

For more information, see Appium Java JUnit (p. 76).
For more information, see Appium Java JUnit (p. 76).

**APPIUM_WEB_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:

   ```shell
   $ 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:

   ```shell
   $ tree .
   ```

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

   ```
   .
   |— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built from the ./src/main directory)
   |— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything built from the ./src/test directory)
   |— zip-with-dependencies.zip (this .zip file contains all of the items)
   `— dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
      |— com.some-dependency.bar-4.1.jar
      |— com.another-dependency.thing-1.0.jar
      |— joda-time-2.7.jar
      `— log4j-1.2.14.jar
   ```

   For more information, see Appium Java JUnit (p. 76).

**APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_TESTS_JAR_FILE_MISSING**

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

**Warning**
We could not find a *-tests.jar file in your test package. Please unzip your test package, verify that at least one *-tests.jar file is in the package, and try again.

In the following example, the package's name is `zip-with-dependencies.zip`.

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

   ```shell
   $ 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
```

For more information, see Appium Java JUnit (p. 76).

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
```
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

For more information, see Appium Java JUnit (p. 76).

**APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_JUNIT_VERSION_VALUE_UNKNOWN**

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

**Warning**

We could not find a JUnit version value. Please unzip your test package and open the dependency-jars directory, verify that the JUnit JAR file is inside the directory, and try again.

In the following example, the package's name is `zip-with-dependencies.zip`.

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

```
$ unzip zip-with-dependencies.zip
```

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

```
tree .
```

The output should look like this:

```
.
| ` acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built from the ./src/main directory)
| ` acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything built from the ./src/test directory)
| ` zip-with-dependencies.zip (this .zip file contains all of the items)
| `- dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
| | ` `-junit-4.10.jar
| | `- com.some-dependency.bar-4.1.jar

API Version 2015-06-23
140
If the Appium Java JUnit package is valid, you will find the JUnit dependency file that is similar to the jar file `junit-4.10.jar` in our example. The name should consist of the keyword `junit` and its version number, which in this example is 4.10.

For more information, see Appium Java JUnit (p. 76).

**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:

   ```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)
   │           └── junit-4.10.jar
   │           └── com.some-dependency.bar-4.1.jar
   │           └── com.another-dependency.thing-1.0.jar
   │           └── joda-time-2.7.jar
   │                   └── log4j-1.2.14.jar
   ```

**Note**

Your tests may not execute correctly if the JUnit version specified in your test package is lower than the minimum version 4.10 we support.

For more information, see Appium Java JUnit (p. 76).
Troubleshooting Appium Java TestNG Tests in AWS Device Farm

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

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

**APPIUM_JAVA_TESTNG_TEST_PACKAGE_UNZIP_FAILED**

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

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

Make sure that you can unzip the test package without errors. In the following example, the package's name is `zip-with-dependencies.zip`.

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

   ```
   $ unzip zip-with-dependencies.zip
   ```

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

   ```
   $ tree .
   ```

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

```
.  
|-- acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built from the ./src/main directory)
|-- acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything built from the ./src/test directory)
|-- zip-with-dependencies.zip (this .zip file contains all of the items)
   `-- dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
       |-- com.some-dependency.bar-4.1.jar
       |-- com.another-dependency.thing-1.0.jar
       |-- joda-time-2.7.jar
       `-- log4j-1.2.14.jar
```

For more information, see Appium Java TestNG (p. 47) or Appium Java TestNG (p. 63).

**APPIUM_JAVA_TESTNG_TEST_PACKAGE_DEPENDENCY_DIR_MISSING**

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

**Warning**
We could not find the `dependency-jars` directory inside your test package. Please unzip your test package, verify that the `dependency-jars` directory is inside the package, and try again.

In the following example, the package's name is `zip-with-dependencies.zip`.

1. Copy your test package to your working directory, and then run the following command:
# unzip zip-with-dependencies.zip

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

```
# tree .
```

If the Appium Java JUnit package is valid, you will find the `dependency-jars` directory inside the working directory.

```
|- acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built from the ./src/main directory)
|- acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything built from the ./src/test directory)
|- zip-with-dependencies.zip (this .zip file contains all of the items)
 `-- dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
    |- com.some-dependency.bar-4.1.jar
    |- com.another-dependency.thing-1.0.jar
    |- joda-time-2.7.jar
    `-- log4j-1.2.14.jar
```

For more information, see Appium TestNG (p. 47) or Appium TestNG (p. 63).

---

**APPIUM_JAVA_TESTNG_TEST_PACKAGE_JAR_MISSING_IN_DEPENDENCY_DIR**

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

**Warning**

We could not find a JAR file in the dependency-jars directory tree. Please unzip your test package and then open the dependency-jars directory, verify that at least one JAR file is in the directory, and try again.

In the following example, the package's name is `zip-with-dependencies.zip`.

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

```
# unzip zip-with-dependencies.zip
```

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

```
# tree .
```

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

```
|- acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built from the ./src/main directory)
|- acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything built from the ./src/test directory)
|- zip-with-dependencies.zip (this .zip file contains all of the items)
 `-- dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
```
For more information, see Appium Java TestNG (p. 47) or Appium Java TestNG (p. 63).

**APPIUM_JAVA_TESTNG_TEST_PACKAGE_TESTS_JAR_FILE_MISSING**

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

**Warning**
We could not find a `*-tests.jar` file in your test package. Please unzip your test package, verify that at least one `*-tests.jar` file is in the package, and try again.

In the following example, the package's name is `zip-with-dependencies.zip`.

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

   ```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`.

```
| acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built from the ./src/main directory)
|-- acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything built from the ./src/test directory)
|-- zip-with-dependencies.zip (this .zip file contains all of the items)
 `-- dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
     |-- com.some-dependency.bar-4.1.jar
     |-- com.another-dependency.thing-1.0.jar
     |-- joda-time-2.7.jar
     `-- log4j-1.2.14.jar
```

For more information, see Appium Java TestNG (p. 47) or Appium Java TestNG (p. 63).

**APPIUM_JAVA_TESTNG_TEST_PACKAGE_CLASS_FILE_MISSING**

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
   ```

   ```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
```
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)
   `-- one-class-file.class
      `-- folder
           |-- another-class-file.class
           `-- zip-with-dependencies.zip (this .zip file contains all of the items)
               `-- dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
                   |-- com.some-dependency.bar-4.1.jar
                   |-- com.another-dependency.thing-1.0.jar
                   |-- joda-time-2.7.jar
                   `-- log4j-1.2.14.jar
```

For more information, see Appium Java TestNG (p. 47) or Appium Java TestNG (p. 63).
Troubleshooting Appium Java TestNG Web Applications in AWS Device Farm

The following topic lists error messages that occur during the upload of Appium Java TestNG Web application tests and recommends workarounds to resolve each error.

**APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_UNZIP_FAILED**

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

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

Make sure that you can unzip the test package without errors. In the following example, the package's name is `zip-with-dependencies.zip`.

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

```
$ unzip zip-with-dependencies.zip
```

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

```
$ tree .
```

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

```
.
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
 `— dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
   |— com.some-dependency.bar-4.1.jar
   |— com.another-dependency.thing-1.0.jar
   |— joda-time-2.7.jar
   `— log4j-1.2.14.jar
```

For more information, see Appium Java TestNG (p. 78).

**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:

```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 Appium Java TestNG (p. 78).

### 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:

```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.

```
. |
  |-- 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:

   ```sh
   # 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:

   ```sh
   # tree .
   ```

If the Appium Java JUnit package is valid, you will find at least one jar file like **acme-android-appium-1.0-SNAPSHOT-tests.jar** in our example. The file's name may be different, but it should end with **-tests.jar**.

```
|-- acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built from the ./src/main directory)
|-- acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything built from the ./src/test directory)
|-- zip-with-dependencies.zip (this .zip file contains all of the items)
  `-- dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
    |-- com.some-dependency.bar-4.1.jar
    |-- com.another-dependency.thing-1.0.jar
    |-- joda-time-2.7.jar
    `-- log4j-1.2.14.jar
```

For more information, see **Appium Java TestNG (p. 78)**.

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:

   ```sh
   # unzip zip-with-dependencies.zip
   ```

For more information, see **Appium Java TestNG (p. 78)**.
2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

```
$ tree .
```

You should find at least one jar file like `acme-android-appium-1.0-SNAPSHOT-tests.jar` in our example. The file's name may be different, but it should end with `-tests.jar`.

```
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything built from the ./src/test directory)
|— zip-with-dependencies.zip (this .zip file contains all of the items)
 `— dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
    |— com.some-dependency.bar-4.1.jar
    |— com.another-dependency.thing-1.0.jar
    |— joda-time-2.7.jar
    `— log4j-1.2.14.jar
```

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

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

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

```
$ tree .
```

You should find at least one class in the working directory tree:

```
|— acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built from the ./src/main directory)
|— acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything built from the ./src/test directory)
|— one-class-file.class
|— folder
 `— another-class-file.class
|— zip-with-dependencies.zip (this .zip file contains all of the items)
 `— dependency-jars (this is the directory that contains all of your dependencies, built as JAR files)
    |— com.some-dependency.bar-4.1.jar
    |— com.another-dependency.thing-1.0.jar
    |— joda-time-2.7.jar
    `— log4j-1.2.14.jar
```

For more information, see Appium Java TestNG (p. 78).

**Troubleshooting Appium Python Tests in AWS Device Farm**

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

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

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

Make sure that you can unzip the test package without errors. In the following example, the package's name is `test_bundle.zip`.

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

   ```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 Appium Python (p. 51) or Appium Python (p. 67).

APPIUM_PYTHON_TEST_PACKAGE_DEPENDENCY_WHEEL_MISSING

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

**Warning**
We could not find a dependency wheel file in the wheelhouse directory tree. Please unzip your test package and then open the wheelhouse directory, verify that at least one wheel file is in the directory, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is `test_bundle.zip`.

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

   ```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 .
   ```

For more information, see Appium Python (p. 51) or Appium Python (p. 67).
If the Appium Python package is valid, you will find at least one `.whl` dependent file like the highlighted files inside the `wheelhouse` directory.

```
|--- requirements.txt
|--- test_bundle.zip
|--- tests (directory)
 |`--- test_unittest.py
 |`--- wheelhouse (directory)
 |    |-- Appium_Python_Client-0.20-cp27-none-any.whl
 |    |-- py-1.4.31-py2.py3-none-any.whl
 |    |-- pytest-2.9.0-py2.py3-none-any.whl
 |    |-- selenium-2.52.0-cp27-none-any.whl
 |    |-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium Python (p. 51) or Appium Python (p. 67).

**APPIUM_PYTHON_TEST_PACKAGE_INVALID_PLATFORM**

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

**Warning**
We found at least one wheel file specified a platform that we do not support. Please unzip your test package and then open the wheelhouse directory, verify that names of wheel files end with `-any.whl` or `-linux_x86_64.whl`, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is `test_bundle.zip`.

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

   ```
   # unzip test_bundle.zip
   ```

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

   ```
   # tree .
   ```

If the Appium Python package is valid, you will find at least one `.whl` dependent file like the highlighted files inside the `wheelhouse` directory. The file's name may be different, but it should end with `-any.whl` or `-linux_x86_64.whl`, which specifies the platform. Any other platforms like windows are not supported.

```
|--- requirements.txt
|--- test_bundle.zip
|--- tests (directory)
 |`--- test_unittest.py
 |`--- wheelhouse (directory)
 |    |-- Appium_Python_Client-0.20-cp27-none-any.whl
 |    |-- py-1.4.31-py2.py3-none-any.whl
 |    |-- pytest-2.9.0-py2.py3-none-any.whl
 |    |-- selenium-2.52.0-cp27-none-any.whl
 |    |-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium Python (p. 51) or Appium Python (p. 67).
APPIUM_PYTHON_TEST_PACKAGE_TEST_DIR_MISSING

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

**Warning**

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

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

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

   ```
   # unzip test_bundle.zip
   ```

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

   ```
   # tree .
   ```

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

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|   |-- test_unittest.py
|   `-- wheelhouse (directory)
`-- Appium_Python_Client-0.20-cp27-none-any.whl
     |-- py-1.4.31-py2.py3-none-any.whl
     |-- pytest-2.9.0-py2.py3-none-any.whl
     |-- selenium-2.52.0-cp27-none-any.whl
     `-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium Python (p. 51) or Appium Python (p. 67).

APPIUM_PYTHON_TEST_PACKAGE_INVALID_TEST_FILE_NAME

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

**Warning**

We could not find a valid test file in the tests directory tree. Please unzip your test package and then open the tests directory, verify that at least one file's name starts or ends with the keyword "test", and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

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

   ```
   # unzip test_bundle.zip
   ```

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

   ```
   # tree .
   ```
If the Appium Python package is valid, you will find the `tests` directory inside the working directory. The file's name may be different, but it should start with `test_` or end with `_test.py`.

```
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
| `-- test_unittest.py
| `-- wheelhouse (directory)
|   |-- Appium_Python_Client-0.20-cp27-none-any.whl
|   |-- py-1.4.31-py2.py3-none-any.whl
|   |-- pytest-2.9.0-py2.py3-none-any.whl
|   |-- selenium-2.52.0-cp27-none-any.whl
|   `-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium Python (p. 51) or Appium Python (p. 67).

**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:

   ```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.

```
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
| `-- test_unittest.py
| `-- wheelhouse (directory)
|   |-- Appium_Python_Client-0.20-cp27-none-any.whl
|   |-- py-1.4.31-py2.py3-none-any.whl
|   |-- pytest-2.9.0-py2.py3-none-any.whl
|   |-- selenium-2.52.0-cp27-none-any.whl
|   `-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium Python (p. 51) or Appium Python (p. 67).
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 Appium Python (p. 51) or Appium Python (p. 67).

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:

   ```
   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:

   ```bash
   $ deactivate
   ```

   For more information, see Appium Python (p. 51) or Appium Python (p. 67).

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

**Warning**

We failed to collect tests in the tests directory. Please unzip your test package, very that the test package is valid by running the command `py.test --collect-only <path to your tests directory>`, and try again after the command does not print any error.

We strongly recommend that you set up Python virtualenv for packaging tests. Here is an example flow of creating a virtual environment using Python virtualenv and then activating it:

```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:

   ```
   $ unzip test_bundle.zip
   ```

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

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

3. To collect tests, you can run the following command:

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

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

   ```
   ===================== test session starts =====================
   platform darwin -- Python 2.7.11, pytest-2.9.0, py-1.4.31, pluggy-0.3.1
   rootdir: /Users/zhena/Desktop/Ios/tests, inifile:
   collected 1 items
   <Module 'test_unittest.py'>
   <UnitTestCase 'DeviceFarmAppiumWebTests'>
   <TestCaseFunction 'test_devicefarm'>
   ===================== no tests ran in 0.11 seconds =====================
   ```

4. To deactivate the virtual environment, you can run the following command:

   ```
   $ deactivate
   ```

For more information, see Appium Python (p. 51) or Appium Python (p. 67).

**Troubleshooting Appium Python Web Application Tests in AWS Device Farm**

The following topic lists error messages that occur during the upload of Appium Python Web application tests and recommends workarounds to resolve each error.

**APPIUM_WEB_PYTHON_TEST_PACKAGE_UNZIP_FAILED**

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

**Warning**

We could not open your Appium test ZIP file. Please verify that the file is valid and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is `test_bundle.zip`.

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

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

```
$ tree .
```

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

```
./
|  |-- requirements.txt
|  |-- test_bundle.zip
|  `-- tests (directory)
|      |-- test_unittest.py
|      `-- wheelhouse (directory)
|      |  |-- Appium_Python_Client-0.20-cp27-none-any.whl
|      |  |-- py-1.4.31-py2.py3-none-any.whl
|      |  |-- pytest-2.9.0-py2.py3-none-any.whl
|      |  |-- selenium-2.52.0-cp27-none-any.whl
|      |  `-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium Python (p. 80).

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

**Warning**

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

Make sure that you can unzip the test package without errors. In the following example, the package's name is `test_bundle.zip`.

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

```
$ unzip test_bundle.zip
```

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

```
$ tree .
```

If the Appium Python package is valid, you will find at least one `.whl` dependent file like the highlighted files inside the `wheelhouse` directory.

```
./
|  |-- requirements.txt
|  |-- test_bundle.zip
|  `-- tests (directory)
|      |-- test_unittest.py
|      `-- wheelhouse (directory)
|          |-- Appium_Python_Client-0.20-cp27-none-any.whl
|          |-- py-1.4.31-py2.py3-none-any.whl
|          |-- pytest-2.9.0-py2.py3-none-any.whl
|          |-- selenium-2.52.0-cp27-none-any.whl
|          `-- wheel-0.26.0-py2.py3-none-any.whl
```
For more information, see Appium Python (p. 80).

**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)

   |-- AppiumPythonClient-0.20-cp27-none-any.whl
   |-- py-1.4.31-py2.py3-none-any.whl
   |-- pytest-2.9.0-py2.py3-none-any.whl
   |-- selenium-2.52.0-cp27-none-any.whl
   `-- wheel-0.26.0-py2.py3-none-any.whl
   ```

For more information, see Appium Python (p. 80).

**APPIUM_WEB_PYTHON_TEST_PACKAGE_TEST_DIR_MISSING**

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

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

Make sure that you can unzip the test package without errors. In the following example, the package's name is `test_bundle.zip`.

1. Copy your test package to your working directory, and then run the following command:
2. After you successfully unzip the package, you can find the working directory tree structure by running the following command:

   ```
   # tree .
   ```

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

   ```
   `-- requirements.txt
   |-- test_bundle.zip
   `-- tests (directory)
      |-- test_unittest.py
   `-- wheelhouse (directory)
      |-- Appium_Python_Client-0.20-cp27-none-any.whl
      |-- py-1.4.31-py2.py3-none-any.whl
      |-- pytest-2.9.0-py2.py3-none-any.whl
      |-- selenium-2.52.0-cp27-none-any.whl
      `-- wheel-0.26.0-py2.py3-none-any.whl
   ```

   For more information, see [Appium Python](p. 80).

---

**APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_TEST_FILE_NAME**

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

**Warning**

We could not find a valid test file in the tests directory tree. Please unzip your test package and then open the tests directory, verify that at least one file's name starts or ends with the keyword "test", and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is `test_bundle.zip`.

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

   ```
   # unzip test_bundle.zip
   ```

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

   ```
   # tree .
   ```

   If the Appium Python package is valid, you will find the `tests` directory inside the working directory. The file's name may be different, but it should start with `test_` or end with `_test.py`.

   ```
   `-- requirements.txt
   |-- test_bundle.zip
   `-- tests (directory)
      |-- test_unittest.py
   `-- wheelhouse (directory)
      |-- Appium_Python_Client-0.20-cp27-none-any.whl
   ```
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.

```
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
 | | `-- test_unittest.py
 | `-- wheelhouse (directory)
    |-- Appium_Python_Client-0.20-cp27-none-any.whl
    |-- py-1.4.31-py2.py3-none-any.whl
    |-- pytest-2.9.0-py2.py3-none-any.whl
    |-- selenium-2.52.0-cp27-none-any.whl
     `-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see Appium Python (p. 80).

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
```

For more information, see Appium Python (p. 80).
# 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 [Appium Python](p. 80).

---

**APPIUM_WEB_PYTHON_TEST_PACKAGE_INSTALL_DEPENDENCY_WHEELS_FAILED**

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

**Warning**

We failed to install the dependency wheels. Please unzip your test package and then open the requirements.txt file and the wheelhouse directory, verify that the dependency wheels specified in the requirements.txt file exactly match the dependency wheels inside the wheelhouse directory, and try again.

We strongly recommend that you set up Python virtualenv for packaging tests. Here is an example flow of creating a virtual environment using Python virtualenv and then activating it:

```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:
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:

```bash
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:

```bash
$ deactivate
```

For more information, see Appium Python (p. 80).

**APPIUM_WEB_PYTHON_TEST_PACKAGE_PYTEST_COLLECT_FAILED**

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

**Warning**

We failed to collect tests in the tests directory. Please unzip your test package, 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:

```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 install wheel files, you can run the following command:

```bash
$ pip install --use-wheel --no-index --find-links=./wheelhouse --requirement=./requirements.txt
```
3. To collect tests, you can run the following command:

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

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

```
==================== test session starts ====================
platform darwin -- Python 2.7.11, pytest-2.9.0, py-1.4.31, pluggy-0.3.1
rootdir: /Users/zhena/Desktop/Ios/tests, inifile:
collected 1 items
<Module 'test_unittest.py'>
<UnitTestCase 'DeviceFarmAppiumWebTests'>
<TestCaseFunction 'test_devicefarm'>

==================== no tests ran in 0.11 seconds ====================
```

4. To deactivate the virtual environment, you can run the following command:

```
$ deactivate
```

For more information, see Appium Python (p. 80).

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:

```
$ unzip features.zip
```

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

```
$ 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
```
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.

   ```
   .
   |-- 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 Calabash (p. 54) or Calabash (p. 70).

---

**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:

   ```bash
   # unzip features.zip
   ```

   For more information, see Calabash (p. 54) or Calabash (p. 70).
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 Calabash (p. 54) or Calabash (p. 70).

**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.

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 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 Calabash (p. 54) or Calabash (p. 70).
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:

   ```
   # 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 the **support** directory inside the **features** directory.

   ```
   $ tree .
   .
     `-- 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 Calabash (p. 54) or Calabash (p. 70).

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.

   ```
   $ tree .
   .
     `-- 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)
         `-- (any other supporting files)
   ```
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:

   ```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 at least one ruby file inside the `support` directory.

   ```bash
   |-- 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
   |   `-- support (directory)
   |       |-- (any other supporting files)
   |       `-- (any other supporting files)
   ```

For more information, see Calabash (p. 54) or Calabash (p. 70).

---

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

---

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

---

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

---

For more information, see Calabash (p. 54) or Calabash (p. 70).
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:

  ```
  $ 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 Calabash (p. 54) or Calabash (p. 70).

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:

  ```
  $ 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:

  ```
  $ 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
I would like to be able to see examples of testing a static homepage
So I can apply it to my future tests."
  }
]```
For more information, see Calabash (p. 54) or Calabash (p. 70).

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:

   ```bash
   $ 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:

   ```bash
   $ tree .
   ```

   A valid Instrumentation test package will produce output like the following:
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: label='Test-api' icon=''
application-debuggable
uses-library:'android.test.runner'
feature-group: label=''
uses-feature: name='android.hardware.touchscreen'
uses-implied-feature: name='android.hardware.touchscreen' reason='default feature for all apps'
supports-screens: 'small' 'normal' 'large' 'xlarge'
supports-any-density: 'true'
locales: '---_--'
densities: '160'
```

For more information, see Instrumentation (p. 56).

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: label='Test-api' icon=''
application-debuggable
uses-library:'android.test.runner'
feature-group: label=''
uses-feature: name='android.hardware.touchscreen'
uses-implied-feature: name='android.hardware.touchscreen' reason='default feature for all apps'
supports-screens: 'small' 'normal' 'large' 'xlarge'
supports-any-density: 'true'
locales: '---_--'
densities: '160'
```

For more information, see Instrumentation (p. 56).
**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:

  ```bash
  $ aapt dump xmltree app-debug-androidTest-unaligned.apk AndroidManifest.xml | grep -A5 "instrumentation"
  ``

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

  ```bash
  E: instrumentation (line=9)
  A: android:label(0x01010001)="Tests for com.amazon.aws.adf.android.referenceapp" (Raw: "Tests for com.amazon.aws.adf.android.referenceapp")
  A: android:name(0x01010003)="android.support.test.runner.AndroidJUnitRunner" (Raw: "android.support.test.runner.AndroidJUnitRunner")
  A: android:targetPackage(0x01010021)="com.amazon.aws.adf.android.referenceapp" (Raw: "com.amazon.aws.adf.android.referenceapp")
  A: android:handleProfiling(0x01010022)=(type 0x12)0x0
  A: android:functionalTest(0x01010023)=(type 0x12)0x0
  ```

  For more information, see [Instrumentation](p. 56).

**INSTRUMENTATION_TEST_PACKAGE_AAPT_DUMP_XMLTREE_FAILED**

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

**Warning**

We could not find the valid AndroidManifest.xml in your 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:

  ```bash
  # 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=(type 0x10)0x9 (Raw: "9")
A: android:targetSdkVersion=(type 0x10)0x16 (Raw: "16")
E: instrumentation (line=9)
A: android:handleProfiling=(type 0x12)0x0
A: android:functionalTest=(type 0x12)0x0
E: application (line=16)
A: android:label=app-debug-androidTest-unaligned (Raw: "app-debug-androidTest-unaligned")
A: android:debuggable=(type 0x12)0xffffffff
E: uses-library (line=17)
A: android:name=android.test.runner (Raw: "android.test.runner")
```

For more information, see Instrumentation (p. 56).

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:

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

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

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

For more information, see Instrumentation (p. 56).
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. 59).

**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:
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. 59).

**IOS_APP_APP_DIR_MISSING**

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

**Warning**
We could not find the .app directory inside the Payload directory. Please unzip your application and then open the Payload directory, verify that the .app directory is inside the directory, and try again.

In the following example, the package's name is `AWSDeviceFarmiOSReferenceApp.ipa`.

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

```
# unzip AWSDeviceFarmiOSReferenceApp.ipa
```

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

```
# tree .
```

If the iOS application package is valid, you will find an .app directory like `AWSDeviceFarmiOSReferenceApp.app` in our example inside the `Payload` directory.

```
|-- Payload (directory)
   `-- AWSDeviceFarmiOSReferenceApp.app (directory)
       |-- Info.plist
       `-- (any other files)
```

For more information, see Working with iOS Tests in AWS Device Farm (p. 59).

**IOS_APP_PLIST_FILE_MISSING**

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

**Warning**
We could not find the Info.plist file inside the .app directory. Please unzip your application and then open the .app directory, verify that the Info.plist file is inside the directory, and try again.

In the following example, the package's name is `AWSDeviceFarmiOSReferenceApp.ipa`.
1. Copy your application package to your working directory, and then run the following command:

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

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

   ```
   $ tree .
   ```

   If the iOS application package is valid, you will find the `Info.plist` file inside the `.app` directory like `AWSDeviceFarmiOSReferenceApp.app` in our example.

   ```
   |-- Payload (directory)
   `-- AWSDeviceFarmiOSReferenceApp.app (directory)
       |-- Info.plist
       `-- (any other files)
   ```

   For more information, see Working with iOS Tests in AWS Device Farm (p. 59).

### IOS_APP_CPU_ARCHITECTURE_VALUE_MISSING

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

**Warning**

We could not find the CPU architecture value in the Info.plist file. Please unzip your application and then open Info.plist file inside the .app directory, verify that the key "UIRequiredDeviceCapabilities" is specified, and try again.

In the following example, the package's name is `AWSDeviceFarmiOSReferenceApp.ipa`.

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

   ```
   $ unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

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

   ```
   $ tree .
   ```

   You should find the `Info.plist` file inside an `.app` directory like `AWSDeviceFarmiOSReferenceApp.app` in our example:

   ```
   |-- Payload (directory)
   `-- AWSDeviceFarmiOSReferenceApp.app (directory)
       |-- Info.plist
       `-- (any other files)
   ```

3. To find the CPU architecture value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

   ```
   # pip install biplist
   ```

4. Next, open Python and run the following command:
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/Info.plist')
print info_plist['UIRequiredDeviceCapabilities']

A valid iOS application package should produce output like the following:

[['armv7']]

For more information, see Working with iOS Tests in AWS Device Farm (p. 59).

**IOS_APP_PLATFORM_VALUE_MISSING**

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

**Warning**

We could not find the platform value in the Info.plist file. Please unzip your application and then open Info.plist file inside the .app directory, verify that the key "CFBundleSupportedPlatforms" is specified, and try again.

In the following example, the package's name is AWSDeviceFarmiOSReferenceApp.ipa.

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

   ```
   # unzip AWSDeviceFarmiOSReferenceApp.ipa
   ```

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

   ```
   # tree .
   ```

   You should find the `Info.plist` file inside an `.app` directory like AWSDeviceFarmiOSReferenceApp.app in our example:

   ```
   `-- Payload (directory)
   | `-- AWSDeviceFarmiOSReferenceApp.app (directory)
   |   `-- Info.plist
   |     `-- (any other files)
   ```

3. To find the platform value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

   ```
   # pip install biplist
   ```

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

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/Info.plist')
   print info_plist['CFBundleSupportedPlatforms']
   ```

   A valid iOS application package should produce output like the following:
If you see the following message, follow these steps to fix the issue.

**Warning**
We found the platform device value was wrong in the Info.plist file. Please unzip your application and then open Info.plist file inside the .app directory, verify that the value of the key "CFBundleSupportedPlatforms" does not contain the keyword "simulator", and try again.

In the following example, the package's name is *AWSDeviceFarmiOSReferenceApp.ipa*.

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

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

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

```
$ tree .
```

You should find the *Info.plist* file inside an *.app* directory like *AWSDeviceFarmiOSReferenceApp.app* in our example:

```
|-- Payload (directory)
`-- AWSDeviceFarmiOSReferenceApp.app (directory)
    |-- Info.plist
    `-- (any other files)
```

3. To find the platform value, you can open Info.plist using Xcode or Python.

For Python, you can install the biplist module by running the following command:

```
# pip install biplist
```

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

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/Info.plist')
print info_plist['CFBundleSupportedPlatforms']
```

A valid iOS application package should produce output like the following:

```
["iPhoneOS"]
```

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

For more information, see *Working with iOS Tests in AWS Device Farm (p. 59)*.
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:
   ```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['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. 59).

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:

   ```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 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('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. 59).

### 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:

   ```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 executable value, you can open `Info.plist` using Xcode or Python.

For Python, you can install the `biplist` module by running the following command:

```
$ pip install biplist
```

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

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
                           Info.plist')
print info_plist['CFBundleExecutable']
```

A valid iOS application package should produce output like the following:

```
AWSDeviceFarmiOSReferenceApp
```

For more information, see Working with iOS Tests in AWS Device Farm (p. 59).

**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:

```
# 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:

```
$ 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. 57).

**Troubleshooting XCTest Tests in AWS Device Farm**

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

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

**XCTEST_TEST_PACKAGE_UNZIP_FAILED**

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

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

Make sure that you can unzip the application package without errors. In the following example, the package's name is `swiftExampleTests.xctest-1.zip`.

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

```
$ unzip swiftExampleTests.xctest-1.zip
```

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

```
$ tree .
```

A valid XCTest package should produce output like the following:

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

For more information, see XCTest (p. 73).

**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 XCTest (p. 73).

---

XCTEST_TEST_PACKAGE_PLIST_FILE_MISSING

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

Warning
We could not find the Info.plist file inside the .xctest directory. Please unzip your test package and then open the .xctest directory, verify that the Info.plist file is inside the directory, and try again.

In the following example, the package's name is swiftExampleTests.xctest-1.zip.

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

   ```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 XCTest (p. 73).
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:

   ```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['CFBundleIdentifier']
   ```

   A valid XCTest application package should produce output like the following:

   ```
   com.amazon.kanapka.swiftExampleTests
   ```

   For more information, see **XCTest** (p. 73).

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

```
$ tree .
```

You should find the `Info.plist` file inside an `.xctest` directory like `swiftExampleTests.xctest` in our example:

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

3. To find the package name value, you can open Info.plist using Xcode or Python. For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

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

```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:

```
swiftExampleTests
```

For more information, see XCTest (p. 73).

### 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. 74).

**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:

```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 `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. 74).
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)
      `-- (any other files)
```

For more information, see **XCTest UI (p. 74)**.

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)
```

For more information, see XCTest UI (p. 74).

### 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 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)
```

For more information, see XCTest UI (p. 74).

### 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 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)
```

For more information, see XCTest UI (p. 74).
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. 74)](#).

---

**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
```

For more information, see [XCTest UI (p. 74)](#).
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 "UIRequiredDeviceCapabilities" is specified, and try again.

In the following example, the package's name is **swift-sample-ui.ipa**.

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

   ```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.xc test (directory)
         `-- Info.plist
         `-- (any other files)
     `-- (any other files)
   ```

3. To find the CPU architecture value, you can open Info.plist using Xcode or Python.

   For Python, you can install the biplist module by running the following command:

   ```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['UIRequiredDeviceCapabilities']
   ```

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

   ```python
   ['armv7']
   ```
For more information, see XCTest UI (p. 74).

**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 unzip your test package and then open the Info.plist file inside the .app directory, verify that the key "CFBundleSupportedPlatforms" is specified, and try again.

In the following example, the package's name is **swift-sample-UI.ipa**.

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

```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 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/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. 74).
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:

   ```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 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/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. 74).

---

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

**XCTEST_UI_TEST_PACKAGE_FORM_FACTOR_VALUE_MISSING**

API Version 2015-06-23

191
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:

   ```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 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:

   ```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['UIDeviceFamily']
   ```

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

   ```
   [1, 2]
   ```

   For more information, see XCTest UI (p. 74).

### XCTEST UI 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 the Info.plist file inside the .app directory, verify that the key "CFBundleIdentifier" is specified, and try again.

In the following example, the package's name is swift-sample-UI.ipa.
1. Copy your test package to your working directory, and then run the following command:

   ```
   $ unzip swift-sample-UI.ipa
   ```

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

   ```
   # tree .
   ```

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

   ```
   |-- Payload (directory)
   |   |-- swift-sampleUITests-Runner.app (directory)
   |       |-- Info.plist
   |       |-- Plugins (directory)
   |       |   `swift-sampleUITests.xctest (directory)
   |       |       |-- Info.plist
   |       |       |-- (any other files)
   |       `-- (any other files)
   |-- (any other files)
   ```

3. To find the package name value, you can open `Info.plist` using Xcode or Python.

   For Python, you can install the `biplist` module by running the following command:

   ```
   # pip install biplist
   ```

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

   ```
   import biplist
   info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
   print info_plist['CFBundleIdentifier']
   ```

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

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

   For more information, see XCTest UI (p. 74).

---

**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:

   ```
   # 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/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. 74).

**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:

```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 .
```

---

**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:

```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 .
```

---

API Version 2015-06-23
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 `bplist` module by running the following command:

   ```bash
   $ pip install bplist
   ```

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

   ```python
   import bplist
   info_plist = bplist.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. 74).

---

**XCTEST_UI_TEST_PACKAGE_TEST_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 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:
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:

```plaintext
swift-sampleUITests
```

For more information, see XCTest UI (p. 74).
User Access Permissions for AWS Device Farm

You can use IAM to enable IAM users in your AWS account to perform only certain actions in Device Farm. You may want to do this, for example, if you have a set of IAM users that you want to allow to list, but not create, resources in Device Farm; you may have another set of IAM users you want to allow to list and create new resources; and so on.

For example, in the Setting Up (p. 4) instructions, you attached an access policy to an IAM user in your AWS account that contains a policy statement similar to this:

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

The preceding statement allows the IAM user in your AWS account to perform actions in Device Farm to which your AWS account has access. In practice, you may not want to give the IAM users in your AWS account this much access.

The following information shows how you can attach a policy to an IAM user to restrict the actions the IAM user can perform in Device Farm.

Create and Attach a Policy to an IAM User

To create and attach an access policy to an IAM user that restricts the actions the IAM user can perform in Device Farm, do the following:

1. Sign in to the AWS Management Console and open the IAM console at [https://console.aws.amazon.com/iam/](https://console.aws.amazon.com/iam/).
2. Choose Policies, and then choose Create Policy. (If a Get Started button appears, choose it, and then choose Create Policy.)
3. Next to Create Your Own Policy, choose Select.
4. For Policy Name, type any value that will be easy for you to refer to later, if needed.
5. For Policy Document, type a policy statement with the following format, and then choose Create Policy:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "devicefarm:*"
      ],
      "Resource": [
        "*"
      ]
    }
  ]
}
```
In the preceding statement, substitute action-statement as needed, and add additional statements as needed, to specify the actions in Device Farm the IAM user can perform. (By default, the IAM user will not have the desired permissions unless a corresponding Allow statement is explicitly stated.) The following section describes the format of allowed actions for Device Farm.

**Note**
Currently, the only allowed value for resource-statement in the preceding example is the asterisk character (*). This means that while you can restrict the actions an IAM user can perform in Device Farm, you cannot also restrict the Device Farm resources the IAM user can access.

6. Choose Users.
7. Choose the IAM user to whom you want to attach the policy.
8. In the Permissions area, for Managed Policies, choose Attach Policy.
9. Select the policy you just created, and then choose Attach Policy.

### Action Syntax for Performing Actions in Device Farm

The following information describes the format for specifying actions an IAM user can perform in Device Farm.

Actions follow this general format:

```
devicefarm:action
```

Where action is an available Device Farm action:

- An asterisk character (*), which represents all of the available Device Farm actions.
- One of the available Device Farm actions, as described in the AWS Device Farm API Reference.
- A combination of an available Device Farm action prefix and an asterisk character (*). For example, specifying List* enables the IAM user to perform all available Device Farm actions that begin with List.

Some example action statements include:
• `devicefarm:*` for all Device Farm actions.
• `devicefarm:Get*` for only the Device Farm actions that begin with `Get`.
• `devicefarm:ListProjects` for just the `ListProjects` Device Farm action.

For example, the following policy statement gives the IAM user permission to get information about all Device Farm resources that are available to the user's AWS account:

```json
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "devicefarm:Get*",
        "devicefarm:List*"
      ],
      "Resource": [
        "*
      ]
    }
  ]
}
```
Limits in AWS Device Farm

The following list describes current Device Farm limits.

- The maximum app file size you can upload is 4 GB.
- There is no limit to the number of devices you can include in a test run. However, the maximum number of devices that Device Farm will simultaneously test during a run is 5. (This number can be increased on request.)
- There is no limit to the number of runs you can schedule.
- There is a sixty-minute limit to length of a remote access session.
- There is a 60 minute limit on automated testing. (This number can be increased to 150 minutes on request.)
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. 201)
- AWS Device Farm Gradle Plugin (p. 206)

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. 204).

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. 204)
- Step 2: Creating an AWS Identity and Access Management User for your Jenkins CI Plugin (p. 205)
- Step 3: First-time configuration instructions (p. 206)
- Step 4: Using the Plugin in a Jenkins Job (p. 206)
- Dependencies (p. 206)

The following images show the features of the Jenkins CI plugin.
Project Hello World App

Build History

<table>
<thead>
<tr>
<th>Build</th>
<th>Date/Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>#19</td>
<td>Jul 15, 2015 4:25 AM</td>
</tr>
<tr>
<td>#18</td>
<td>Jul 15, 2015 1:35 AM</td>
</tr>
<tr>
<td>#17</td>
<td>Jul 15, 2015 1:21 AM</td>
</tr>
<tr>
<td>#16</td>
<td>Jul 15, 2015 1:06 AM</td>
</tr>
<tr>
<td>#15</td>
<td>Jul 14, 2015 10:55 PM</td>
</tr>
</tbody>
</table>

Recent AWS Device Farm Results

- Completed #19
- Completed #18
- Completed #17
- Completed #16
- Completed #15

Permalinks

- Last build (#19), 41 min ago
- Last failed build (#19), 41 min ago
- Last unsuccessful build (#19), 41 min ago
The plugin can also pull down all the test artifacts (logs, screenshots, etc.) locally:
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:
   a. Upload the plugin by choosing Manage Jenkins, Manage Plugins, Advanced, and then choose Upload plugin.
   b. 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:

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

8. Choose Create Policy.

To assign the access policy to the IAM user
1. Open the IAM console at https://console.aws.amazon.com/iam/.
2. Choose Users.
3. Choose the IAM user to whom you will assign the access policy.
4. In the Permissions area, for Managed Policies, choose Attach Policy.
5. Select the policy you just created (for example, AWSDeviceFarmAccessPolicy).
6. Choose Attach Policy.
Step 3: 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. 15), 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. 205) 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. 207).

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.
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 Setting up the Device Farm Gradle Plugin (p. 207).

**To build the plugin**

1. Go to GitHub and clone the repository.
2. Build the plugin using `gradle install`.

The plugin is installed to your local maven repository.

Next step: Setting up the Device Farm Gradle Plugin (p. 207)

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. 207).

**To configure the AWS Device Farm Gradle Plugin**

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

   ```gradle
   buildscript {

       repositories {
           mavenLocal()
           mavenCentral()
       }

       dependencies {
           classpath 'com.android.tools.build:gradle:1.3.0'
           classpath 'com.amazonaws:aws-devicefarm-gradle-plugin:1.0'
       }
   }
   ```

2. Configure the plugin in your `build.gradle` file. The following test specific configuration should serve as your guide:

   ```gradle
   apply plugin: 'devicefarm'
   ```
devicefarm {

    projectName "My Project" // required: Must already exists.

devicePool "My Device Pool Name" // optional: Defaults to "Top Devices"

useUnmeteredDevices() // optional if you wish to use your un-metered devices

    authentication {
        accessKey "aws-iam-user-accesskey"
        secretKey "aws-iam-user-secretkey"

        // or
        roleArn "My role arn" // Optional, if role arn is specified, it will be used.

    } // Otherwise use access and secret keys

    // optional block, radios default to 'on' state, all parameters optional
devicestate {

        extraDataZipFile file("relative/path/to/zip") // default null
        auxiliaryApps [file("path1"), file("path2")]) // default empty list
        wifi on
        bluetooth off
        gps off
        nfc on
        latitude 47.6204 // default
        longitude -122.3491 // default

    } // Configure test type, if none default to instrumentation
    // Fuzz
    // fuzz {

    } // Instrumentation
    // See AWS Developer docs for filter (optional)
    // instrumentation { filter "my-filter" }

    // Calabash
calabash {

        tests file("path-to-features.zip")

    }
。

3. Run your Device Farm test using the following task: gradle devicefarmUpload.

   The build output will print out a link to the Device Farm console where you can monitor your test execution.

Next step: Generating an IAM user (p. 208)

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. 209).

For more information, see Creating an IAM User (AWS Management Console) or Setting Up (p. 4).

**Step 4: Configuring Test Types**

By default, the AWS Device Farm Gradle plugin runs the Instrumentation (p. 56) 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 to configure it for use from within Android Studio. For more information about the available test types in Device Farm, see Working with Test Types in AWS Device Farm (p. 42).
If you haven’t done so already, complete steps 1 – 3 before configuring test types.

**Note**
If you are using device slots (p. 15), 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. 43)
- Appium Java TestNG (p. 47)

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

```yaml
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:

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

For more information:
- Built-in: Explorer (Android) (p. 84)

### 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.

```yaml
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. 85).

### Calabash

Device Farm provides support for Calabash for Android. To learn how to prepare your Android Calabash tests, see Calabash (p. 54)
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 Instrumentation (p. 56).

When running an instrumentation test in Gradle, Device Farm uses the .apk file generated from your androidTest directory as the source of your tests.

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

UI Automator

Upload your app, as well as your UI Automator-based tests, packaged in a .jar file.

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

For more information, see UI Automator (p. 57).

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>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 Set the Execution Timeout for Test Runs in AWS Device Farm (p. 26).</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 Simulate Network Connections and Conditions (p. 28).</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. 127).</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. 86).</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 Device Slots (p. 15).</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. 30).</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. 74) 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 and iOS applications, as well as Web applications. Learn more about Test Types in AWS Device Farm (p. 10).</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 Custom Web App Tests in AWS Device Farm (p. 76).</td>
<td>November 19, 2015</td>
</tr>
<tr>
<td>AWS Device Farm Gradle Plugin</td>
<td>Learn more about how to install and use the Device Farm Gradle Plugin (p. 206).</td>
<td>September 28, 2015</td>
</tr>
<tr>
<td>New Android Built-in Test: Explorer</td>
<td>Learn more about Built-in: Explorer (Android) (p. 84). 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>Change</td>
<td>Description</td>
<td>Date Changed</td>
</tr>
<tr>
<td>------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>iOS support added</td>
<td>Learn more about testing iOS devices and running iOS tests (including XCTest) in Working with Test Types in AWS Device Farm (p. 42).</td>
<td>August 4, 2015</td>
</tr>
<tr>
<td>Initial public release</td>
<td>This is the initial public release of the AWS Device Farm Developer Guide.</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.