
AWS Device Farm

Developer Guide

API Version 2015-06-23



AWS Device Farm: Developer Guide

Copyright © 2017 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

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

Table of Contents

What Is AWS Device Farm?	1
Automated App Testing	1
Supported Test Types and Built-in Tests	1
Remote Access Interaction	2
Terminology	2
Setting Up	3
Setting Up	4
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	7
Step 2: Create a Project	7
Step 3: Create and Start a Run	7
Step 4: View the Run's Results	8
Next Steps	9
Concepts	10
Devices	10
Supported Devices	10
Device Pools	10
Device Branding	11
Device Slots	11
Pre-Installed Device Apps	11
Device Capabilities	11
Test Types	11
Android Test Types	11
iOS Test Types	12
Web Application Test Types	12
Runs	12
Run Configuration	13
Run Files Retention	13
Run Device State	13
Parallel Runs	13
Setting the execution timeout in test runs	13
Instrumenting Apps	13
Re-Signing Apps in Runs	13
Obfuscated Apps in Runs	14
Ads in Runs	14
Media in Runs	14
Common Tasks for Runs	14
Reports	14
Report Retention	14
Report Components	14
Performance Samples in Reports	15
Logs in Reports	15
Common Tasks for Reports	15
Sessions	15
Supported Devices for Remote Access	15
Session Files Retention	15
Instrumenting Apps	15
Re-Signing Apps in Sessions	16
Obfuscated Apps in Sessions	16

Purchase Device Slots	17
Purchase Device Slots with the Device Farm Console	17
Purchase a Device Slot with the AWS CLI	18
Purchase a Device Slot with the Device Farm API	21
Working with Projects	22
Create a Project	22
Prerequisites	22
Create a Project with the Device Farm Console	22
Create a Project with the AWS CLI	23
Create a Project with the Device Farm API	23
View the Projects List	23
Prerequisites	23
View the Projects List with the Device Farm Console	24
View the Projects List with the AWS CLI	24
View the Projects List with the Device Farm API	24
Working with Test Runs	25
Create a Test Run	25
Prerequisites	26
Create a Test Run with the Device Farm Console	26
Create a Run with the AWS CLI	27
Create a Run with the Device Farm API	28
Next Steps	28
Set Execution Timeout	28
Prerequisites	28
Set the Execution Timeout for a Project	29
Set the Execution Timeout for a Test Run	29
Simulate Network Connections and Conditions	30
Set up Network Shaping When Scheduling a Test Run	30
Create your own Network Profile	31
Change Network Conditions During your Test	32
Stop a Run	33
Prerequisites	33
Stop a Run with the Device Farm Console	33
Stop a Run with the AWS CLI	34
Stop a Run with the Device Farm API	35
View a Runs List	35
Prerequisites	36
View a Runs List with the Device Farm Console	36
View a Runs List with the AWS CLI	36
View a Runs List with the Device Farm API	36
Create a Device Pool	36
Prerequisites	37
Create a Device Pool with the Device Farm Console	37
Create a Device Pool with the AWS CLI	38
Create a Device Pool with the Device Farm API	38
Analyze a Report	38
Prerequisites	38
Console Icons	38
Open a Report with the Device Farm Console	38
Analyze a Report's Summary Page with the Device Farm Console	39
Analyze a Report's Unique Problems with the Device Farm Console	39
Analyze a Report by Device with the Device Farm Console	40
Analyze a Report by Suite with the Device Farm Console	40
Analyze a Report by Test with the Device Farm Console	41
Analyze Performance Data for a Problem, Device, Suite, or Test in a Report with the Device Farm Console	42

Analyze Log Information for a Problem, Device, Suite, or Test in a Report with the Device Farm Console	42
Working with Test Types	45
Built-in Test Types	45
Custom Test Types	45
Custom Android Test Types	45
Custom iOS Test Types	45
Custom Web Application Test Types	46
Android Tests	46
Built-in Test Types for Android	46
Custom Test Types for Android	46
Appium Java JUnit	46
Appium Java TestNG	50
Appium Python	54
Calabash	57
Instrumentation	59
UI Automator	60
iOS Tests	62
Built-in Test Types for iOS	62
Custom Test Types	62
Appium Java JUnit	62
Appium Java TestNG	66
Appium Python	70
Calabash	73
UI Automation	75
XCTest	76
XCTest UI	77
Web App Tests	78
Rules for Metered and Unmetered Devices	78
Appium Java JUnit	79
Appium Java TestNG	81
Appium Python	83
Built-in Tests	86
Built-in Test Types	86
Built-in: Explorer (Android)	86
Built-in: Fuzz (Android and iOS)	87
Working with Remote Access	89
Create a Session	89
Prerequisites	90
Create a Session with the Device Farm Console	90
Next Steps	90
Use a Session	90
Prerequisites	90
Use a Session in the Device Farm Console	91
Next Steps	91
Tips and Tricks	91
Get Session Results	91
Prerequisites	92
Viewing Session Details	92
Downloading Session Video or Logs	92
CloudTrail Integration	93
Device Farm Information in CloudTrail	93
Understanding Device Farm Log File Entries	94
AWS CLI Reference	96
Windows PowerShell Reference	97
API Reference	98
Troubleshooting	99

Android Applications	99
ANDROID_APP_UNZIP_FAILED	99
ANDROID_APP_AAPT_DEBUG_BADGING_FAILED	100
ANDROID_APP_PACKAGE_NAME_VALUE_MISSING	101
ANDROID_APP_SDK_VERSION_VALUE_MISSING	101
ANDROID_APP_AAPT_DUMP_XMLTREE_FAILED	102
ANDROID_APP_DEVICE_ADMIN_PERMISSIONS	102
Appium Java JUnit	103
APPIUM_JAVA_JUNIT_TEST_PACKAGE_PACKAGE_UNZIP_FAILED	103
APPIUM_JAVA_JUNIT_TEST_PACKAGE_DEPENDENCY_DIR_MISSING	104
APPIUM_JAVA_JUNIT_TEST_PACKAGE_JAR_MISSING_IN_DEPENDENCY_DIR	105
APPIUM_JAVA_JUNIT_TEST_PACKAGE_TESTS_JAR_FILE_MISSING	105
APPIUM_JAVA_JUNIT_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TESTS_JAR	106
APPIUM_JAVA_JUNIT_TEST_PACKAGE_JUNIT_VERSION_VALUE_UNKNOWN	107
APPIUM_JAVA_JUNIT_TEST_PACKAGE_INVALID_JUNIT_VERSION	108
Appium Java JUnit Web	109
APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_UNZIP_FAILED	109
APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_DEPENDENCY_DIR_MISSING	109
APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_JAR_MISSING_IN_DEPENDENCY_DIR	110
APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_TESTS_JAR_FILE_MISSING	111
APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TESTS_JAR	111
APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_JUNIT_VERSION_VALUE_UNKNOWN	112
APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_INVALID_JUNIT_VERSION	113
Appium Java TestNG	114
APPIUM_JAVA_TESTNG_TEST_PACKAGE_UNZIP_FAILED	114
APPIUM_JAVA_TESTNG_TEST_PACKAGE_DEPENDENCY_DIR_MISSING	115
APPIUM_JAVA_TESTNG_TEST_PACKAGE_JAR_MISSING_IN_DEPENDENCY_DIR	115
APPIUM_JAVA_TESTNG_TEST_PACKAGE_TESTS_JAR_FILE_MISSING	116
APPIUM_JAVA_TESTNG_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TESTS_JAR	117
Appium Java TestNG Web	118
APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_UNZIP_FAILED	118
APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_DEPENDENCY_DIR_MISSING	119
APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_JAR_MISSING_IN_DEPENDENCY_DIR	119
APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_TESTS_JAR_FILE_MISSING	120
APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_CLASS_FILE_MISSING_IN_TESTS_JAR	121
Appium Python	122
APPIUM_PYTHON_TEST_PACKAGE_UNZIP_FAILED	122
APPIUM_PYTHON_TEST_PACKAGE_DEPENDENCY_WHEEL_MISSING	122
APPIUM_PYTHON_TEST_PACKAGE_INVALID_PLATFORM	123
APPIUM_PYTHON_TEST_PACKAGE_TEST_DIR_MISSING	124
APPIUM_PYTHON_TEST_PACKAGE_INVALID_TEST_FILE_NAME	124
APPIUM_PYTHON_TEST_PACKAGE_REQUIREMENTS_TXT_FILE_MISSING	125
APPIUM_PYTHON_TEST_PACKAGE_INVALID_PYTEST_VERSION	126
APPIUM_PYTHON_TEST_PACKAGE_INSTALL_DEPENDENCY_WHEELS_FAILED	127
APPIUM_PYTHON_TEST_PACKAGE_PYTEST_COLLECT_FAILED	127
Appium Python Web	128
APPIUM_WEB_PYTHON_TEST_PACKAGE_UNZIP_FAILED	129
APPIUM_WEB_PYTHON_TEST_PACKAGE_DEPENDENCY_WHEEL_MISSING	129
APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_PLATFORM	130
APPIUM_WEB_PYTHON_TEST_PACKAGE_TEST_DIR_MISSING	131
APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_TEST_FILE_NAME	131
APPIUM_WEB_PYTHON_TEST_PACKAGE_REQUIREMENTS_TXT_FILE_MISSING	132
APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_PYTEST_VERSION	133
APPIUM_WEB_PYTHON_TEST_PACKAGE_INSTALL_DEPENDENCY_WHEELS_FAILED	133
APPIUM_WEB_PYTHON_TEST_PACKAGE_PYTEST_COLLECT_FAILED	134
Calabash	135
CALABASH_TEST_PACKAGE_UNZIP_FAILED_UNZIP_FAILED	135

CALABASH_TEST_PACKAGE_FEATURES_DIR_MISSING_FEATURES_DIR_MISSING	136
CALABASH_TEST_PACKAGE_FEATURE_FILE_MISSING	137
CALABASH_TEST_PACKAGE_STEP_DEFINITIONS_DIR_MISSING	137
CALABASH_TEST_PACKAGE_SUPPORT_DIR_MISSING	138
CALABASH_TEST_PACKAGE_RUBY_FILE_MISSING_IN_STEP_DEFINITIONS_DIR	138
CALABASH_TEST_PACKAGE_RUBY_FILE_MISSING_IN_SUPPORT_DIR	139
CALABASH_TEST_PACKAGE_EMBEDDED_SERVER_MISSING	140
CALABASH_TEST_PACKAGE_DRY_RUN_FAILED	140
Instrumentation	141
INSTRUMENTATION_TEST_PACKAGE_UNZIP_FAILED	141
INSTRUMENTATION_TEST_PACKAGE_AAPT_DEBUG_BADGING_FAILED	142
INSTRUMENTATION_TEST_PACKAGE_INSTRUMENTATION_RUNNER_VALUE_MISSING	143
INSTRUMENTATION_TEST_PACKAGE_AAPT_DUMP_XMLTREE_FAILED	143
INSTRUMENTATION_TEST_PACKAGE_TEST_PACKAGE_NAME_VALUE_MISSING	144
iOS Applications	145
IOS_APP_UNZIP_FAILED	145
IOS_APP_PAYLOAD_DIR_MISSING	145
IOS_APP_APP_DIR_MISSING	146
IOS_APP_PLIST_FILE_MISSING	147
IOS_APP_CPU_ARCHITECTURE_VALUE_MISSING	147
IOS_APP_PLATFORM_VALUE_MISSING	148
IOS_APP_WRONG_PLATFORM_DEVICE_VALUE	149
IOS_APP_FORM_FACTOR_VALUE_MISSING	150
IOS_APP_PACKAGE_NAME_VALUE_MISSING	151
IOS_APP_EXECUTABLE_VALUE_MISSING	151
UI Automator	152
UIAUTOMATOR_TEST_PACKAGE_UNZIP_FAILED	152
XCTest	153
XCTEST_TEST_PACKAGE_UNZIP_FAILED	153
XCTEST_TEST_PACKAGE_XCTEST_DIR_MISSING	154
XCTEST_TEST_PACKAGE_PLIST_FILE_MISSING	154
XCTEST_TEST_PACKAGE_PACKAGE_NAME_VALUE_MISSING	155
XCTEST_TEST_PACKAGE_EXECUTABLE_VALUE_MISSING	156
XCTest UI	156
XCTEST_UI_TEST_PACKAGE_UNZIP_FAILED	157
XCTEST_UI_TEST_PACKAGE_PAYLOAD_DIR_MISSING	157
XCTEST_UI_TEST_PACKAGE_APP_DIR_MISSING	158
XCTEST_UI_TEST_PACKAGE_PLUGINS_DIR_MISSING	158
XCTEST_UI_TEST_PACKAGE_XCTEST_DIR_MISSING_IN_PLUGINS_DIR	159
XCTEST_UI_TEST_PACKAGE_PLIST_FILE_MISSING	160
XCTEST_UI_TEST_PACKAGE_PLIST_FILE_MISSING_IN_XCTEST_DIR	160
XCTEST_UI_TEST_PACKAGE_CPU_ARCHITECTURE_VALUE_MISSING	161
XCTEST_UI_TEST_PACKAGE_PLATFORM_VALUE_MISSING	162
XCTEST_UI_TEST_PACKAGE_WRONG_PLATFORM_DEVICE_VALUE	163
XCTEST_UI_TEST_PACKAGE_FORM_FACTOR_VALUE_MISSING	164
XCTEST_UI_TEST_PACKAGE_PACKAGE_NAME_VALUE_MISSING	165
XCTEST_UI_TEST_PACKAGE_EXECUTABLE_VALUE_MISSING	165
XCTEST_UI_TEST_PACKAGE_TEST_PACKAGE_NAME_VALUE_MISSING	166
XCTEST_UI_TEST_PACKAGE_TEST_EXECUTABLE_VALUE_MISSING	167
Access Permissions Reference	169
Create and Attach a Policy to an IAM User	169
Action Syntax for Performing Actions in Device Farm	170
Limits	172
Tools and Plugins	173
Jenkins CI Plugin	173
Step 1: Install the Plugin	176
Step 2: Create an IAM User	177

Step 3: First-time configuration instructions	178
Step 4: Use the Plugin	178
Dependencies	178
Device Farm Gradle Plugin	178
Building the Device Farm Gradle Plugin	179
Setting up the Device Farm Gradle Plugin	179
Generating an IAM user	180
Configuring Test Types	181
Dependencies	183
Document History	184
AWS Glossary	186

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 \(p. 46\)](#)
- [Appium Java TestNG \(p. 50\)](#)
- [Appium Python \(p. 54\)](#)
- [Calabash \(p. 57\)](#)
- [Instrumentation \(p. 59\)](#) (JUnit, Espresso, Robotium, or any instrumentation-based tests)
- [UI Automator \(p. 60\)](#)
- [Explorer \(p. 86\)](#)

For iOS:

- [Appium Java JUnit \(p. 62\)](#)
- [Appium Java TestNG \(p. 66\)](#)
- [Appium Python \(p. 70\)](#)
- [Calabash \(p. 73\)](#)
- [UI Automation \(p. 75\)](#)
- [XCTest \(p. 76\) \(including KIF\)](#)
- [XCTest UI \(p. 77\)](#)

For Web Apps:

- [Appium Java JUnit \(p. 79\)](#)
- [Appium Java TestNG \(p. 81\)](#)
- [Appium Python \(p. 83\)](#)

If you do not have your own tests, you can use a built-in fuzz test. For more information, see [Built-in: Fuzz \(Android and iOS\) \(p. 87\)](#).

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 \(p. 12\)](#).

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 \(p. 14\)](#).

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

1. Open <https://aws.amazon.com/> and choose **Create an AWS Account**.
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

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

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "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, 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 \(p. 169\)](#).

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 \(p. 7\)](#)
- [Step 2: Create a Project \(p. 7\)](#)
- [Step 3: Create and Start a Run \(p. 7\)](#)
- [Step 4: View the Run's Results \(p. 8\)](#)
- [Next Steps \(p. 9\)](#)

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 \(p. 45\)](#).

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

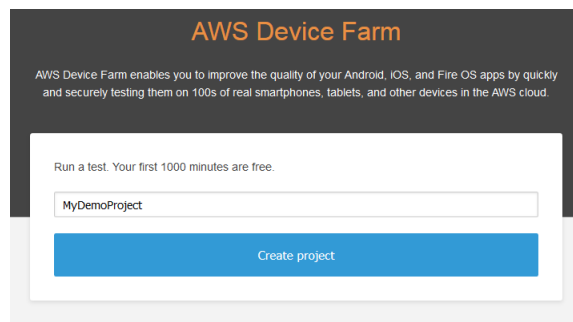
To test an app in Device Farm, you must first create a project.

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 \(p. 12\)](#).

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 \(p. 10\)](#).

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**.
12. On the **Review and start run** page, choose **Confirm and start run**.

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 \(p. 38\)](#).

You have now completed this walkthrough.

Next Steps

To learn more about Device Farm, see [Concepts \(p. 10\)](#).

AWS Device Farm Concepts

This section describes important Device Farm concepts.

- [Devices](#) (p. 10)
- [Test Types in AWS Device Farm](#) (p. 11)
- [Runs](#) (p. 12)
- [Reports](#) (p. 14)
- [Sessions](#) (p. 15)

Device Support in AWS Device Farm

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

Topics

- [Supported Devices](#) (p. 10)
- [Device Pools](#) (p. 10)
- [Device Branding](#) (p. 11)
- [Device Slots](#) (p. 11)
- [Pre-Installed Device Apps](#) (p. 11)
- [Device Capabilities](#) (p. 11)

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 your own private device pools. For more information about using private devices, see [Device Farm Pricing](#).

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. 45).

Android Test Types

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

- [Built-in: Explorer \(Android\)](#) (p. 86)

- [Built-in: Fuzz \(Android and iOS\) \(p. 87\)](#)
- [Appium Java JUnit \(p. 46\)](#)
- [Appium Java TestNG \(p. 50\)](#)
- [Appium Python \(p. 54\)](#)
- [Calabash \(p. 57\)](#)
- [Instrumentation \(p. 59\)](#)
- [UI Automator \(p. 60\)](#)

iOS Test Types

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

- [Built-in: Fuzz \(Android and iOS\) \(p. 87\)](#)
- [Appium Java JUnit \(p. 62\)](#)
- [Appium Java TestNG \(p. 66\)](#)
- [Appium Python \(p. 70\)](#)
- [Calabash \(p. 73\)](#)
- [UI Automation \(p. 75\)](#)
- [XCTest \(p. 76\)](#)
- [XCTest UI \(p. 77\)](#)

Web Application Test Types

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

- [Appium Java JUnit \(p. 79\)](#)
- [Appium Java TestNG \(p. 81\)](#)
- [Appium Python \(p. 83\)](#)

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 \(p. 13\)](#)
- [Run Files Retention \(p. 13\)](#)
- [Run Device State \(p. 13\)](#)
- [Parallel Runs \(p. 13\)](#)
- [Setting the execution timeout in test runs \(p. 13\)](#)
- [Instrumenting Apps \(p. 13\)](#)
- [Re-Signing Apps in Runs \(p. 13\)](#)

- [Obfuscated Apps in Runs \(p. 14\)](#)
- [Ads in Runs \(p. 14\)](#)
- [Media in Runs \(p. 14\)](#)
- [Common Tasks for Runs \(p. 14\)](#)

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 \(p. 28\)](#).

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 \(p. 25\)](#) and [Working with Test Runs \(p. 25\)](#).

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 \(p. 14\)](#)
- [Report Components \(p. 14\)](#)
- [Performance Samples in Reports \(p. 15\)](#)
- [Logs in Reports \(p. 15\)](#)
- [Common Tasks for Reports \(p. 15\)](#)

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 \(p. 38\)](#).

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 \(p. 15\)](#)
- [Session Files Retention \(p. 15\)](#)
- [Instrumenting Apps \(p. 15\)](#)
- [Re-Signing Apps in Sessions \(p. 16\)](#)
- [Obfuscated Apps in Sessions \(p. 16\)](#)

Supported Devices for Remote Access

Device Farm provides support for a number of unique popular Android and Fire OS devices and operating system combinations. 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 Fire OS devices available for remote access is displayed in the console. For more information about devices, see [Devices \(p. 10\)](#).

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

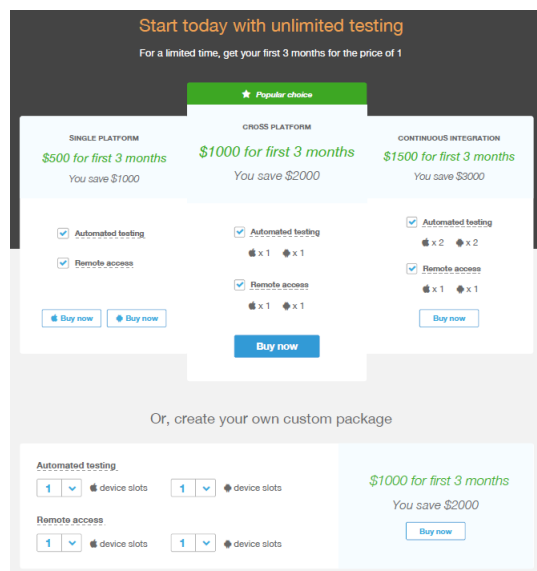
To purchase a device slot, you can use the Device Farm console, the AWS CLI, or the Device Farm API.

Topics

- [Purchase Device Slots with the Device Farm Console \(p. 17\)](#)
- [Purchase a Device Slot with the AWS CLI \(p. 18\)](#)
- [Purchase a Device Slot with the Device Farm API \(p. 21\)](#)

Purchase Device Slots with the Device Farm Console

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
2. Scroll down to the **UNLIMITED ACCESS** section and choose the **Learn more about unlimited testing** link to get to the **packages** page.



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](#).

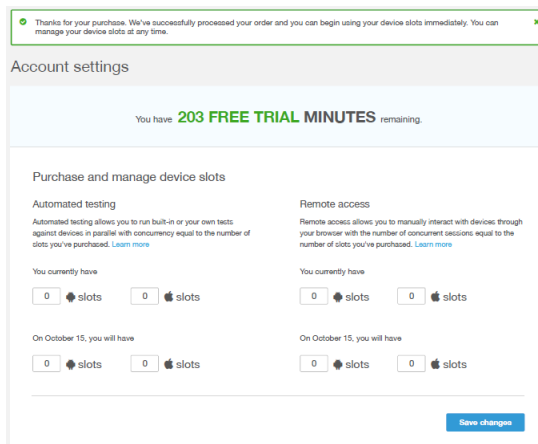
4. Choose **Buy now** for the package you wish to purchase.

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:

```
{
  "accountSettings": {
    "maxSlots": {
      "GUID": 1,
      "GUID": 1,
      "GUID": 1,
      "GUID": 1
    },
    "unmeteredRemoteAccessDevices": {
      "ANDROID": 0,
      "IOS": 0
    },
    "maxJobTimeoutMinutes": 60,
    "trialMinutes": {
      "total": 1000.0,
      "remaining": 954.1
    },
    "defaultJobTimeoutMinutes": 60,
    "awsAccountNumber": "AWS-ACCOUNT-NUMBER",
    "unmeteredDevices": {
      "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:

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

```

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

To list 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:

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

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

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

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

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 \(p. 96\)](#).

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 \(p. 98\)](#).

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 \(p. 22\)](#)
- [View the Projects List \(p. 23\)](#)

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 \(p. 22\)](#)
- [Create a Project with the Device Farm Console \(p. 22\)](#)
- [Create a Project with the AWS CLI \(p. 23\)](#)
- [Create a Project with the Device Farm API \(p. 23\)](#)

Prerequisites

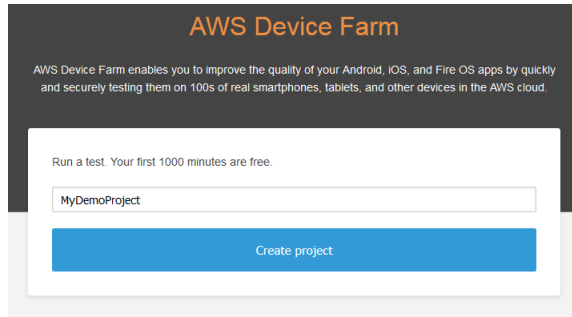
- Complete the steps in [Setting Up \(p. 4\)](#), which include signing up for an AWS account, creating or using an IAM user in the AWS account, and giving the IAM user permission to access Device Farm.

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.
2. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 28\)](#).



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

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 \(p. 96\)](#).

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 \(p. 98\)](#).

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 \(p. 23\)](#)
- [View the Projects List with the Device Farm Console \(p. 24\)](#)
- [View the Projects List with the AWS CLI \(p. 24\)](#)
- [View the Projects List with the Device Farm API \(p. 24\)](#)

Prerequisites

- Create at least one project in Device Farm. To create a project, follow the instructions in [Create a Project \(p. 22\)](#), 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.
2. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 96\)](#).

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 \(p. 98\)](#).

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 \(p. 12\)](#).

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 \(p. 25\)](#)
- [Set the Execution Timeout for Test Runs in AWS Device Farm \(p. 28\)](#)
- [Simulate Network Connections and Conditions for your AWS Device Farm Runs \(p. 30\)](#)
- [Stop a Run in AWS Device Farm \(p. 33\)](#)
- [View a Runs List in AWS Device Farm \(p. 35\)](#)
- [Create a Device Pool in AWS Device Farm \(p. 36\)](#)
- [Analyze a Report in AWS Device Farm \(p. 38\)](#)

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 \(p. 12\)](#).

Topics

- [Prerequisites \(p. 26\)](#)
- [Create a Test Run with the Device Farm Console \(p. 26\)](#)
- [Create a Run with the AWS CLI \(p. 27\)](#)
- [Create a Run with the Device Farm API \(p. 28\)](#)

- [Next Steps \(p. 28\)](#)

Prerequisites

- Create a project in Device Farm. Follow the instructions in [Create a Project \(p. 22\)](#), and then return to this page.

Create a Test Run with the Device Farm Console

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 45\)](#).

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 \(p. 36\)](#), 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 \(p. 10\)](#).

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.

Create a new run

- 1 Choose application
- 2 Configure test
- 3 Select devices
- 4 Specify device state
- 5 **Review and start run**

Review and start run

Review your run below. Look good? Confirm to start your run. Interested in unlimited, unmetered testing? [Learn more](#).

Execution timeout

Limit the maximum number of device minutes that your run can use by setting a timeout on each device. If execution exceeds your timeout, execution on that device will be forcibly stopped. Partial results will be available if possible.

30 × 2 = 60
MAXIMUM MINUTES PER DEVICE DEVICES MAXIMUM DEVICE MINUTES FOR THIS RUN

5m 32m 60m

You should choose a value that is greater than the anticipated duration of your tests. For example, if your tests take 20 minutes to complete you should choose a timeout of 30 minutes. Need to run tests longer than 60 minutes? Please [contact us](#).

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 \(p. 28\)](#).
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 \(p. 33\)](#).

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 \(p. 23\)](#).
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 \(p. 38\)](#).

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 \(p. 96\)](#).

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 \(p. 23\)](#).
2. Call the `ScheduleRun` API.

For information about using the Device Farm API, see [API Reference \(p. 98\)](#).

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 \(p. 14\)](#).

To use the report, follow the instructions in [Analyze a Report \(p. 38\)](#).

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.

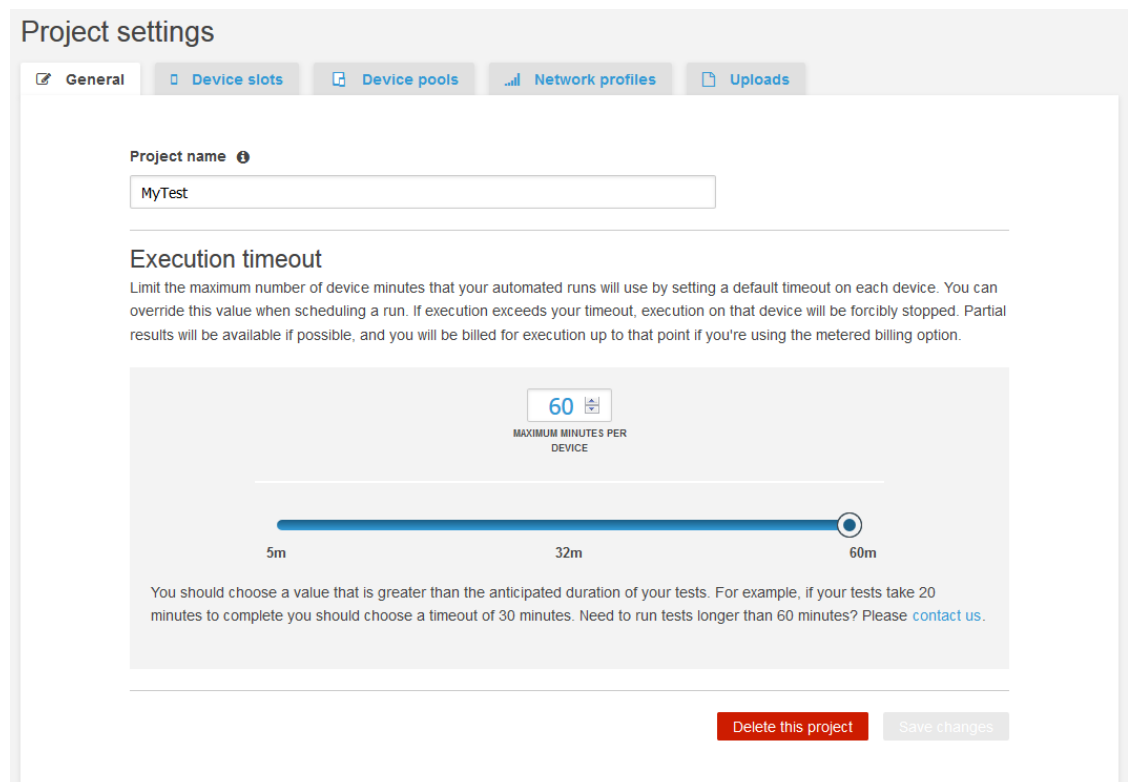
2. Create a project in Device Farm. Follow the instructions in [Create a Project \(p. 22\)](#), and then return to this page.

Set the Execution Timeout for a Project

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
2. 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

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 30\)](#)
- [Create your own Network Profile \(p. 31\)](#)
- [Change Network Conditions During your Test \(p. 32\)](#)

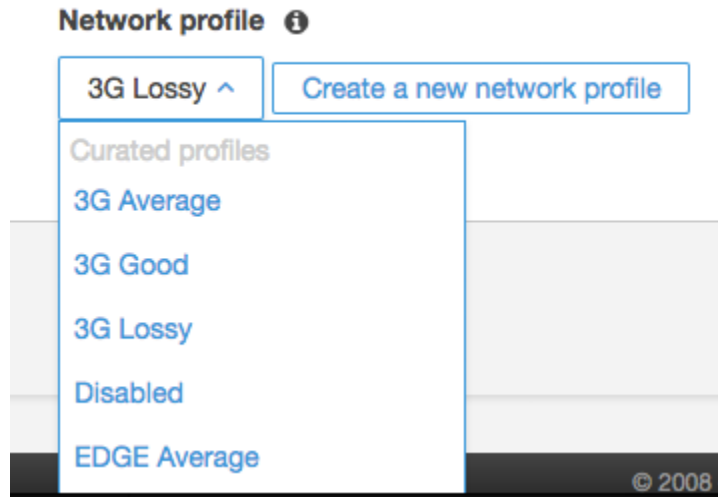
Set up Network Shaping When Scheduling a Test Run

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 \(p. 22\)](#).

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.



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 run

Create a new network profile

Network profile name: Please enter a new network profile name.

Description: Please enter a short description (optional).

Uplink bandwidth (bps): 104857600

Downlink bandwidth (bps): 104857600

Uplink delay (ms): 0

Downlink delay (ms): 0

Uplink jitter (ms): 0

Downlink jitter (ms): 0

Uplink loss (%): 0

Downlink loss (%): 0

Cancel Save network profile

Cancel Previous Next Step

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.

Project settings

General Device slots Network profiles Uploads Device pools

Create a new network profile

Name	Bandwidth (bit/s)	Delay (ms)	Jitter (ms)	Loss (%)	Description
Field-captured (T-Mobile)	↑ 52857600 ↓ 104857600	↑ 20 ↓ 5	↑ 1 ↓ 1	↑ 0 ↓ 0	T-Mobile connection s...
My new profile	↑ 104857600 ↓ 104857600	↑ 0 ↓ 0	↑ 0 ↓ 0	↑ 0 ↓ 0	

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 \(p. 12\)](#).

Topics

- [Prerequisites \(p. 33\)](#)
- [Stop a Run with the Device Farm Console \(p. 33\)](#)
- [Stop a Run with the AWS CLI \(p. 34\)](#)
- [Stop a Run with the Device Farm API \(p. 35\)](#)

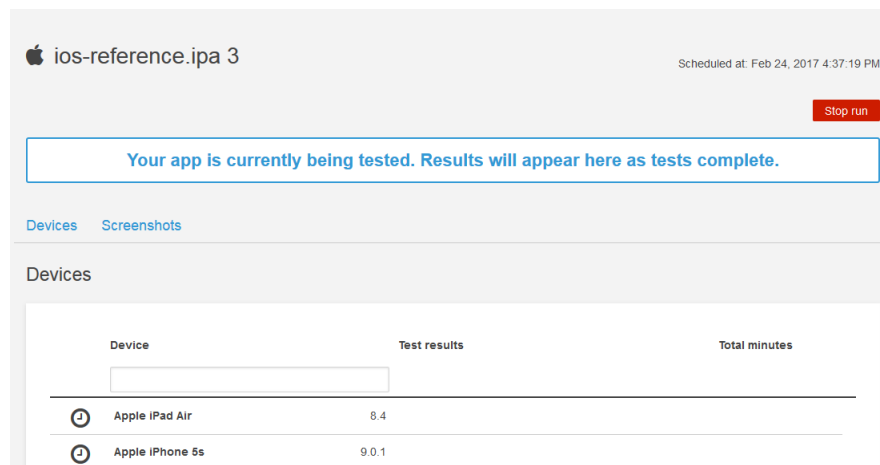
Prerequisites

- To stop a test run, you must have a test run created and actively running. For more information, see [Create a Test Run \(p. 25\)](#).

Stop a Run with the Device Farm Console

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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).



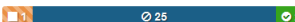

4. Choose **Stop run**.

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.

Device	Test results	Total minutes
Apple iPad Air	8.4 	00:00:40
Apple iPhone 5s	9.0.1  This device was unavailable and skipped	

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:

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

```
}
```

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

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

Note

For information about using Device Farm with the AWS CLI, see [AWS CLI Reference \(p. 96\)](#).

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 \(p. 98\)](#).

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 \(p. 36\)](#)
- [View a Runs List with the Device Farm Console \(p. 36\)](#)
- [View a Runs List with the AWS CLI \(p. 36\)](#)
- [View a Runs List with the Device Farm API \(p. 36\)](#)

Prerequisites

- Create at least one run in Device Farm. To create a run, follow the instructions in [Create a Test Run \(p. 25\)](#), and then return to this page.

View a Runs List with the Device Farm Console

1. Make sure that you complete the [prerequisites \(p. 36\)](#), including the creation of at least one run.
2. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 36\)](#), 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 \(p. 96\)](#).

View a Runs List with the Device Farm API

1. Make sure that you have completed the [prerequisites \(p. 36\)](#), 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 \(p. 98\)](#).

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 \(p. 37\)](#)
- [Create a Device Pool with the Device Farm Console \(p. 37\)](#)
- [Create a Device Pool with the AWS CLI \(p. 38\)](#)

- [Create a Device Pool with the Device Farm API \(p. 38\)](#)

Prerequisites

- Start by creating a run in the Device Farm console. Follow the instructions in [Create a Test Run \(p. 25\)](#). 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.

For information about using Device Farm with the AWS CLI, see [AWS CLI Reference \(p. 96\)](#).

Create a Device Pool with the Device Farm API

- Call the `CreateDevicePool` API.

For information about using the Device Farm API, see [API Reference \(p. 98\)](#).

Analyze a Report in AWS Device Farm

Use the Device Farm console to analyze a report. For more information, see [Reports \(p. 14\)](#).

Prerequisites

- Create a run in Device Farm, and verify the run is complete. Follow the instructions in [Create a Test Run \(p. 25\)](#), and then return to this page.

Console Icons

Icon	Description
Green check mark inside of a circle	Success
Orange exclamation mark inside of a triangle	Warning
Red exclamation mark inside of a circle	Failure
Blue circle with a slash through it	Skipped
Orange square	Stopped

Open a Report with the Device Farm Console

1. Make sure the run is complete.
2. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 38\)](#).
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 \(p. 39\)](#).
 - 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.
 - If there is at least one test with errors, a red exclamation mark is displayed.
 - If there is at least one failed test, a red exclamation mark inside of a circle is displayed.
 - If there is at least one test with warnings, an orange exclamation mark inside of a triangle is displayed.
 - Otherwise, a green check mark inside of a circle is displayed.
 - To analyze the results by device, follow the instructions in [Analyze a Report by Device with the Device Farm Console \(p. 40\)](#).
 - The **Screenshots** section displays a list of any screenshots Device Farm captured during the run, grouped by device.

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 \(p. 38\)](#).
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 \(p. 42\)](#).
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 \(p. 42\)](#).

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 \(p. 38\)](#).
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 \(p. 40\)](#).

The **Logs** section displays any information Device Farm logged for the device 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 \(p. 42\)](#).

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 \(p. 42\)](#).
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 \(p. 38\)](#).
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.

- 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 \(p. 41\)](#).

- 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 \(p. 42\)](#).
- 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 in [Analyze Performance Data for a Problem, Device, Suite, or Test in a Report with the Device Farm Console \(p. 42\)](#).
- 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.
- 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

- 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 \(p. 38\)](#).
- In the **Devices** section, choose the device that corresponds to the test you want to analyze for results.
- In the **Suites** section, choose the suite that corresponds to the test you want to analyze for results.
- 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 \(p. 42\)](#).

- 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 \(p. 42\)](#).

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 \(p. 39\)](#)
 - [Analyze a Report by Device with the Device Farm Console \(p. 40\)](#)
 - [Analyze a Report by Suite with the Device Farm Console \(p. 40\)](#)
 - [Analyze a Report by Test with the Device Farm Console \(p. 41\)](#)
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 \(p. 39\)](#).

- [Analyze a Report by Device with the Device Farm Console \(p. 40\)](#)
 - [Analyze a Report by Suite with the Device Farm Console \(p. 40\)](#)
 - [Analyze a Report by Test with the Device Farm Console \(p. 41\)](#)
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:
 - **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 `e` (such as `Harness` and `Test`), type `*e*` 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 `s` (such as `Passed`), type `Hs*` 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 \(Android\) \(p. 86\)](#)
- [Built-in: Fuzz \(Android and iOS\) \(p. 87\)](#)

Custom Test Types

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

Custom Android Test Types

- [Appium Java JUnit \(p. 46\)](#)
- [Appium Java TestNG \(p. 50\)](#)
- [Appium Python \(p. 54\)](#)
- [Calabash \(p. 57\)](#)
- [Instrumentation \(p. 59\)](#)
- [UI Automator \(p. 60\)](#)

Custom iOS Test Types

- [Appium Java JUnit \(p. 62\)](#)
- [Appium Java TestNG \(p. 66\)](#)
- [Appium Python \(p. 70\)](#)

- [Calabash \(p. 73\)](#)
- [UI Automation \(p. 75\)](#)
- [XCTest \(p. 76\)](#)
- [XCTest UI \(p. 77\)](#)

Custom Web Application Test Types

- [Appium Java JUnit \(p. 79\)](#)
- [Appium Java TestNG \(p. 81\)](#)
- [Appium Python \(p. 83\)](#)

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 \(Android\) \(p. 86\)](#)
- [Built-in: Fuzz \(Android and iOS\) \(p. 87\)](#)

Custom Test Types for Android

The following custom tests are available for Android devices.

- [Appium Java JUnit \(p. 46\)](#)
- [Appium Java TestNG \(p. 50\)](#)
- [Appium Python \(p. 54\)](#)
- [Calabash \(p. 57\)](#)
- [Instrumentation \(p. 59\)](#)
- [UI Automator \(p. 60\)](#)

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? \(p. 47\)](#)
- [Version Information \(p. 47\)](#)
- [Prepare Your Android Appium Java JUnit Tests \(p. 47\)](#)

- [Upload Your Android Appium Java JUnit Tests \(p. 49\)](#)
- [Taking Screenshots in Android Appium Java JUnit Tests \(p. 50\)](#)
- [Additional Considerations for Android Appium Java JUnit Tests \(p. 50\)](#)

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, see [About Appium](#).

Version Information

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

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:

```
<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:

```
<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:

```
<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:

```

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

```

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

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

```

<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.5.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <finalName>zip-with-dependencies</finalName>
      </configuration>
    </execution>
  </executions>
</plugin>

```



```
<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:

```
<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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 22\)](#).

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. 47).
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 several Java system properties that describe the configuration of the Appium server with which you're communicating. For example, Device Farm sets the `appium.screenshots.dir` 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.

```
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. 50)
- [Version Information](#) (p. 51)
- [Prepare Your Android Appium Java TestNG Tests](#) (p. 51)
- [Upload Your Android Appium Java TestNG Tests](#) (p. 53)
- [Taking Screenshots in Android Appium Java TestNG Tests](#) (p. 53)
- [Additional Considerations for Android Appium Java TestNG Tests](#) (p. 54)

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, see [About Appium](#).

Version Information

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

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:

```
<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:

```
<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:

```
<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:

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

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

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

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.5.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <finalName>zip-with-dependencies</finalName>
        <appendAssemblyId>>false</appendAssemblyId>
        <descriptors>
          <descriptor>src/main/assembly/zip.xml</descriptor>
        </descriptors>
      </configuration>
    </execution>
  </executions>
</plugin>
```

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`:

```
<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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 22\)](#).

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 \(p. 51\)](#).
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 TestNG test, the service sets several Java system properties that describe the configuration of the Appium server with which you're communicating. For example, Device Farm sets the `appium.screenshots.dir` 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.

```
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? \(p. 54\)](#)
- [Version Information \(p. 54\)](#)
- [Prepare Your Android Application Appium Python Tests \(p. 54\)](#)
- [Build the Appium Python Test Package \(p. 55\)](#)
- [Upload Your Android Application Appium Python Tests \(p. 56\)](#)
- [Taking Screenshots in Android Appium Python Tests \(p. 57\)](#)
- [Additional Considerations for Android Appium Python Tests \(p. 57\)](#)

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, see [About 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).

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/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
```

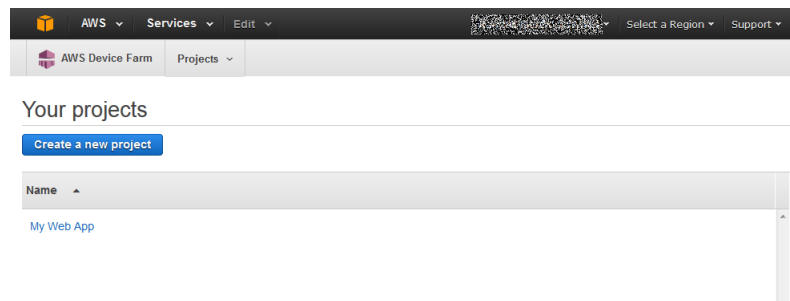
Your work space will eventually look like this:

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

Upload Your Android Application Appium Python Tests

Use the Device Farm console to upload your tests.

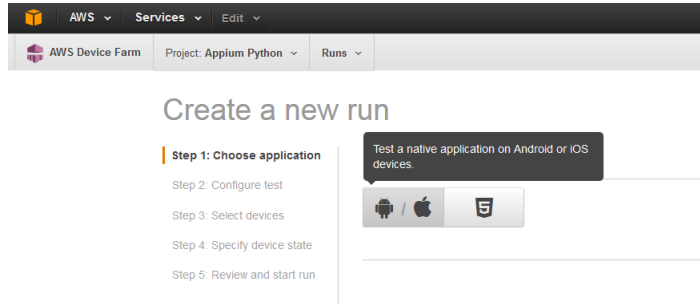
1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
2. If you see the AWS Device Farm console home page, choose **Get started**.
3. If you already have a project, you can upload your tests to an existing project or choose **Create a new project**.



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 project, follow the instructions in [Create a Project \(p. 22\)](#).

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 \(p. 25\)](#).

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 several system properties that describe the configuration of the Appium server with which you're communicating. For example, Device Farm sets the `SCREENSHOT_PATH` property to a fully qualified path on the local file system where Device Farm expects Appium screenshots to be saved. The test-specific directory where the screenshots are stored is defined at runtime. The screenshots are pulled into your Device Farm reports automatically. To view the screenshots, in the Device Farm console, choose the **Screenshots** section.

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.

```
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 0.7.2.

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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 22\)](#).

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 \(p. 58\)](#).
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`:

```
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? \(p. 59\)](#)
- [Upload Your Android Instrumentation Tests \(p. 59\)](#)
- [Taking Screenshots in Android Instrumentation Tests \(p. 60\)](#)
- [Additional Considerations for Android Instrumentation Tests \(p. 60\)](#)

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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 22\)](#).

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:

```
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? \(p. 61\)](#)
- [Prepare Your Android UI Automator Tests \(p. 61\)](#)

- [Upload Your Android UI Automator Tests \(p. 61\)](#)
- [Taking Screenshots in Android UI Automator Tests \(p. 61\)](#)
- [Additional Considerations for Android UI Automator Tests \(p. 62\)](#)

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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 22\)](#).

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 \(p. 61\)](#).
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 \(Android and iOS\) \(p. 87\)](#)

Custom Test Types

The following custom tests are available for iOS devices.

- [Appium Java JUnit \(p. 62\)](#)
- [Appium Java TestNG \(p. 66\)](#)
- [Appium Python \(p. 70\)](#)
- [Calabash \(p. 73\)](#)
- [UI Automation \(p. 75\)](#)
- [XCTest \(p. 76\)](#)
- [XCTest UI \(p. 77\)](#)

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? \(p. 62\)](#)
- [Version Information \(p. 63\)](#)
- [Prepare Your iOS Appium Java JUnit Tests \(p. 63\)](#)
- [Upload Your iOS Appium Java JUnit Tests \(p. 65\)](#)
- [Taking Screenshots in iOS Appium Java JUnit Tests \(p. 66\)](#)
- [Additional Considerations for iOS Appium Java JUnit Tests \(p. 66\)](#)

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, see [About Appium](#).

Version Information

Currently, Device Farm supports Appium versions 1.6.3 (for iOS 10 and later) and 1.4.16 (for iOS 9 and earlier) and Java version Java 8.

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:

```
<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:

```
<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:

```
<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:

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

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

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

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.5.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <finalName>zip-with-dependencies</finalName>
        <appendAssemblyId>>false</appendAssemblyId>
        <descriptors>
          <descriptor>src/main/assembly/zip.xml</descriptor>
        </descriptors>
      </configuration>
    </execution>
  </executions>
</plugin>
```

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:

```
<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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 22\)](#).

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 several Java system properties that describe the configuration of the Appium server with which you're communicating. For example, Device Farm sets the `appium.screenshots.dir` 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.

```
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? \(p. 66\)](#)
- [Version Information \(p. 66\)](#)
- [Prepare Your iOS Appium Java TestNG Tests \(p. 66\)](#)
- [Upload Your iOS Appium Java TestNG Tests \(p. 69\)](#)
- [Taking Screenshots in iOS Appium Java TestNG Tests \(p. 69\)](#)
- [Additional Considerations for iOS Appium Java TestNG Tests \(p. 70\)](#)

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, see [About Appium](#).

Version Information

Currently, Device Farm supports Appium versions 1.6.3 (for iOS 10 and later) and 1.4.16 (for iOS 9 and earlier) and Java version 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:

```
<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:

```
<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:

```
<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:

```
<assembly
  xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

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

```

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

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

```

<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.5.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <finalName>zip-with-dependencies</finalName>
        <appendAssemblyId>>false</appendAssemblyId>
        <descriptors>
          <descriptor>src/main/assembly/zip.xml</descriptor>
        </descriptors>
      </configuration>
    </execution>
  </executions>
</plugin>

```

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`:

```
<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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 22\)](#).

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 TestNG test, the service sets several Java system properties that describe the configuration of the Appium server with which you're communicating. For example, Device Farm sets the `appium.screenshots.dir` 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.

```
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? \(p. 70\)](#)
- [Version Information \(p. 70\)](#)
- [Prepare Your iOS Application Appium Python Tests \(p. 70\)](#)
- [Build the Appium Python Test Package \(p. 70\)](#)
- [Upload Your iOS Application Appium Python Tests \(p. 72\)](#)
- [Taking Screenshots in iOS Appium Python Tests \(p. 73\)](#)
- [Additional Considerations for Android Appium Python Tests \(p. 73\)](#)

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, see [About Appium](#).

Version Information

Currently, Device Farm supports Appium versions 1.6.3 (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).

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/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 iOS Application Appium Python Tests

Use the Device Farm console to upload your tests.

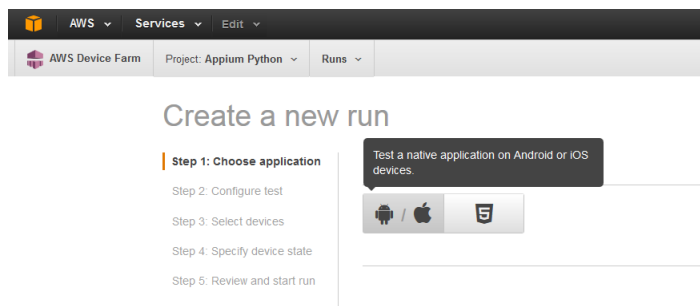
1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
2. If you see the AWS Device Farm console home page, choose **Get started**.
3. If you already have a project, you can upload your tests to an existing project or choose **Create a new project**.



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 project, follow the instructions in [Create a Project \(p. 22\)](#).

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 \(p. 25\)](#).

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 several system properties that describe the configuration of the Appium server with which you're communicating. For example, Device Farm sets the `SCREENSHOT_PATH` property to a fully qualified path on the local file system where Device Farm expects Appium screenshots to be saved. The test-specific directory where the screenshots are stored is defined at runtime. The screenshots are pulled into your Device Farm reports automatically. To view the screenshots, in the Device Farm console, choose the **Screenshots** section.

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.

```
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? \(p. 73\)](#)
- [Version Information \(p. 73\)](#)
- [Prepare Your iOS Calabash Tests \(p. 73\)](#)
- [Upload Your iOS Calabash Tests \(p. 74\)](#)
- [Taking Screenshots in iOS Calabash Tests \(p. 74\)](#)
- [Additional Considerations for iOS Calabash Tests \(p. 74\)](#)

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 0.20.3.

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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 22\)](#).

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 \(p. 73\)](#).
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`:

```
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? \(p. 75\)](#)
- [Upload Your iOS UI Automation Tests \(p. 75\)](#)
- [Taking Screenshots in iOS UI Automation Tests \(p. 75\)](#)
- [Additional Considerations for iOS UI Automation Tests \(p. 76\)](#)

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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 22\)](#).

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. 76)
- [Prepare Your iOS XCTest Tests](#) (p. 76)
- [Upload Your iOS XCTest Tests](#) (p. 76)
- [Taking Screenshots in iOS XCTest Tests](#) (p. 77)
- [Additional Considerations for iOS XCTest Tests](#) (p. 77)

What is XCTest (and KIF)?

XCTest is the new testing framework introduced with Xcode 5. XCTest is a modernized reimplementaion of OCUnt, the previous-generation testing framework. For more information, see [XCTest—the Xcode Testing Framework](#) and [Transitioning from OCUnt 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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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. 22).

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 \(p. 76\)](#).
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? \(p. 77\)](#)
- [Prepare Your iOS XCTest UI Tests \(p. 77\)](#)
- [Upload Your iOS XCTest UI Tests \(p. 78\)](#)
- [Taking Screenshots in iOS XCTest UI Tests \(p. 78\)](#)
- [Additional Considerations for iOS XCTest UI Tests \(p. 78\)](#)

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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 22\)](#).

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 \(p. 77\)](#).
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 \(p. 79\)](#)
- [Appium Java TestNG \(p. 81\)](#)
- [Appium Python \(p. 83\)](#)

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? \(p. 79\)](#)
- [Version Information \(p. 79\)](#)
- [Prepare Your Web Application Appium Java JUnit Tests \(p. 79\)](#)
- [Upload Your Web Application Appium Java JUnit Tests \(p. 79\)](#)
- [Taking Screenshots in Web Application Appium Java JUnit Tests \(p. 80\)](#)
- [Additional Considerations for Web Application Appium Java JUnit Tests \(p. 81\)](#)

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, see [About Appium](#).

Version Information

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

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:

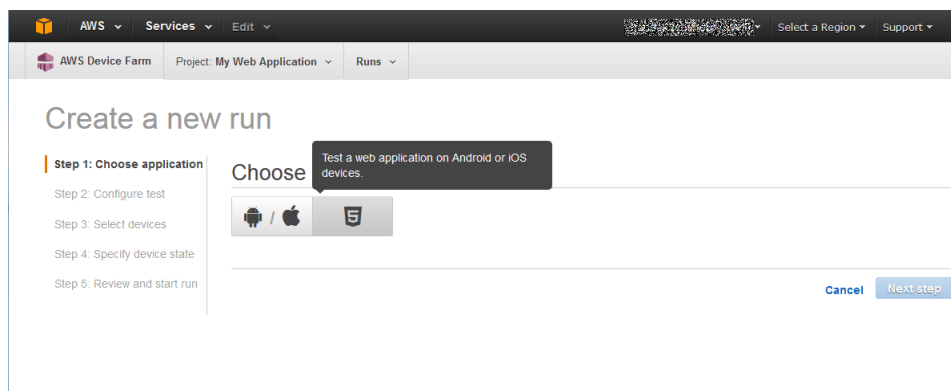
1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
2. If you see the AWS Device Farm console home page, choose **Get started**.
3. If you already have a project, you can upload your tests to an existing project or choose **Create a new project**.



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 \(p. 22\)](#).

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 \(p. 25\)](#).

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 several system properties that describe the configuration of the Appium server with which you're communicating. For example, 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.

```
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? \(p. 81\)](#)
- [Version Information \(p. 81\)](#)
- [Prepare Your Web Application Appium Java TestNG Tests \(p. 81\)](#)
- [Upload Your Web Application Appium Java TestNG Tests \(p. 81\)](#)
- [Taking Screenshots in Web Application Appium TestNG Tests \(p. 82\)](#)
- [Additional Considerations for Web Application Appium TestNG Tests \(p. 83\)](#)

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, see [About Appium](#).

Version Information

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

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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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**.

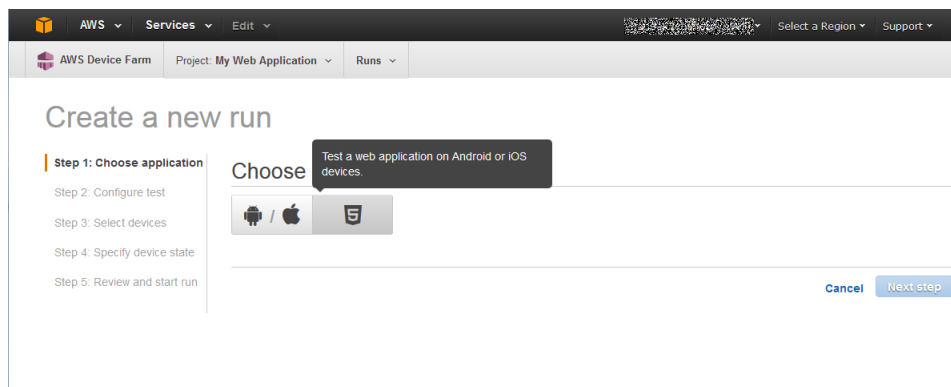


Tip

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 \(p. 22\)](#).

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 \(p. 25\)](#).

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 TestNG test, the service sets several system properties that describe the configuration of the Appium server with which you're communicating. For example, 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.

```
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? \(p. 83\)](#)
- [Version Information \(p. 83\)](#)
- [Prepare Your Web Application Appium Python Tests \(p. 83\)](#)
- [Build the Appium Python Test Package \(p. 83\)](#)
- [Upload Your Web Application Appium Python Tests \(p. 85\)](#)
- [Taking Screenshots in Web Application Appium Python Tests \(p. 86\)](#)
- [Additional Considerations for Web Application Appium Python Tests \(p. 86\)](#)

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, see [About 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).

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.

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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**.

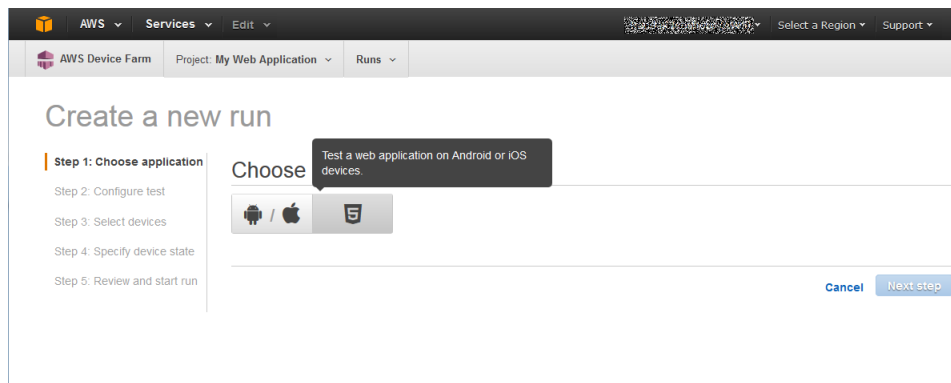


Tip

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 project, follow the instructions in [Create a Project \(p. 22\)](#).

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).



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 \(p. 25\)](#).

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 several system properties that describe the configuration of the Appium server with which you're communicating. For example, Device Farm sets the `SCREENSHOT_PATH` property to a fully qualified path on the local file system where Device Farm expects Appium screenshots to be saved. The test-specific directory where the screenshots are stored is defined at runtime. The screenshots are pulled into your Device Farm reports automatically. To view the screenshots, in the Device Farm console, choose the **Screenshots** section.

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.

```
screenshot_folder = os.getenv('SCREENSHOT_PATH', '')
self.driver.save_screenshot(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 \(Android\) \(p. 86\)](#)
- [Built-in: Fuzz \(Android and iOS\) \(p. 87\)](#)

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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 \(p. 22\)](#).

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**.
7. On the **Configure a test** page, choose **Built-in: Explorer**.
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:

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
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 to run the built-in fuzz test.

To create a new project, follow the instructions in [Create a Project \(p. 22\)](#).

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**.
7. On the **Configure a test** page, choose **Built-in: Fuzz**.
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 \(p. 89\)](#)
- [Use a Session \(p. 90\)](#)
- [Get Session Results \(p. 91\)](#)

Create a Remote Access Session in AWS Device Farm

For information about remote access sessions, see [Sessions \(p. 15\)](#).

- [Prerequisites \(p. 90\)](#)
- [Create a Test Run with the Device Farm Console \(p. 90\)](#)
- [Next Steps \(p. 90\)](#)

Prerequisites

- Create a project in Device Farm. Follow the instructions in [Create a Project \(p. 22\)](#), and then return to this page.

Create a Session with the Device Farm Console

1. Sign in to the Device Farm console at <https://console.aws.amazon.com/devicefarm>.
2. 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 \(p. 15\)](#).

- [Prerequisites \(p. 90\)](#)
- [Use a Session in the Device Farm Console \(p. 91\)](#)
- [Next Steps \(p. 91\)](#)
- [Tips and Tricks \(p. 91\)](#)

Prerequisites

- Create a session. Follow the instructions in [Create a Session \(p. 89\)](#), 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 \(p. 91\)](#).

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 \(p. 15\)](#).

- [Prerequisites \(p. 92\)](#)

- [Viewing Session Details \(p. 92\)](#)
- [Downloading Session Video or Logs \(p. 92\)](#)

Prerequisites

- Complete a session. Follow the instructions in [Use a Session \(p. 90\)](#), 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 \(p. 42\)](#).

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.

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 \(p. 96\)](#) and the [API Reference \(p. 98\)](#). 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:

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "Root",
        "principalId": "AKIAI44QH8DHBEXAMPLE",
        "arn": "arn:aws:iam:123456789012:root",
        "accountId": "123456789012",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
          "attributes": {
            "mfaAuthenticated": "false",
            "creationDate": "2015-07-08T21:13:35Z"
          }
        }
      },
      "eventTime": "2015-07-09T00:51:22Z",
      "eventSource": "devicefarm.amazonaws.com",
      "eventName": "ListRuns",
      "awsRegion": "us-west-2",
      "sourceIPAddress": "203.0.113.11",
      "userAgent": "example-user-agent-string",
      "requestParameters": {
        "arn": "arn:aws:devicefarm:us-west-2:123456789012:project:a9129b8c-
df6b-4cdd-8009-40a25EXAMPLE",
        "responseElements": {
          "runs": [
            {
              "created": "Jul 8, 2015 11:26:12 PM",
              "name": "example.apk",
              "completedJobs": 2,
              "arn": "arn:aws:devicefarm:us-west-2:123456789012:run:a9129b8c-
df6b-4cdd-8009-40a256aEXAMPLE/1452d105-e354-4e53-99d8-6c993EXAMPLE",
              "counters": {
                "stopped": 0,
                "warned": 0,
                "failed": 0,
                "passed": 4,
                "skipped": 0,
                "total": 4,
                "errored": 0
              },
              "type": "BUILTIN_FUZZ",
              "status": "RUNNING",
              "totalJobs": 3,
              "platform": "ANDROID_APP",
              "result": "PENDING"
            },
            ... additional entries ...
          ]
        }
      }
    }
  ]
}
```

```
}  
 ]  
}
```

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 Command Line Interface 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 \(p. 99\)](#)
- [Troubleshooting Appium Java JUnit Tests in AWS Device Farm \(p. 103\)](#)
- [Troubleshooting Appium Java JUnit Web Application Tests in AWS Device Farm \(p. 109\)](#)
- [Troubleshooting Appium Java TestNG Tests in AWS Device Farm \(p. 114\)](#)
- [Troubleshooting Appium Java TestNG Web Applications in AWS Device Farm \(p. 118\)](#)
- [Troubleshooting Appium Python Tests in AWS Device Farm \(p. 122\)](#)
- [Troubleshooting Appium Python Web Application Tests in AWS Device Farm \(p. 128\)](#)
- [Troubleshooting Calabash Tests in AWS Device Farm \(p. 135\)](#)
- [Troubleshooting Instrumentation Tests in AWS Device Farm \(p. 141\)](#)
- [Troubleshooting iOS Application Tests in AWS Device Farm \(p. 145\)](#)
- [Troubleshooting UI Automator Tests in AWS Device Farm \(p. 152\)](#)
- [Troubleshooting XCTest Tests in AWS Device Farm \(p. 153\)](#)
- [Troubleshooting XCTest UI Tests in AWS Device Farm \(p. 156\)](#)

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 \(p. 46\)](#).

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-implies-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 \(p. 46\)](#).

ANDROID_APP_PACKAGE_NAME_VALUE_MISSING

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

Warning

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

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

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

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

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

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

```
package: name='com.amazon.aws.adf.android.referenceapp' versionCode='1'  
  versionName='1.0' platformBuildVersionName='5.1.1-1819727'
```

For more information, see [Working with Android Tests in AWS Device Farm \(p. 46\)](#).

ANDROID_APP_SDK_VERSION_VALUE_MISSING

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

Warning

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

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

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

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

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

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

```
sdkVersion:'9'
```

For more information, see [Working with Android Tests in AWS Device Farm \(p. 46\)](#).

ANDROID_APP_AAPT_DUMP_XMLTREE_FAILED

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

Warning

We could not find the valid AndroidManifest.xml in your application. Please verify that the test package is valid by running the command `aapt dump xmltree <path to your test package> AndroidManifest.xml`, and try again after the command does not print any error.

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

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

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

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

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

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

For more information, see [Working with Android Tests in AWS Device Farm \(p. 46\)](#).

ANDROID_APP_DEVICE_ADMIN_PERMISSIONS

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

Warning

We found that your application requires device admin permissions. Please verify that the permissions are not required by run the command `aapt dump xmltree <path to your test`

`package`> `AndroidManifest.xml`, and try again after making sure that output does not contain the keyword `android.permission.BIND_DEVICE_ADMIN`.

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

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

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

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

You should find output like the following:

```
N: android=http://schemas.android.com/apk/res/android
E: manifest (line=2)
  A: android:versionCode(0x0101021b)=(type 0x10)0x1
  A: android:versionName(0x0101021c)="1.0" (Raw: "1.0")
  A: package="com.amazonaws.devicefarm.android.referenceapp" (Raw:
"com.amazonaws.devicefarm.android.referenceapp")
  A: platformBuildVersionCode=(type 0x10)0x16 (Raw: "22")
  A: platformBuildVersionName="5.1.1-1819727" (Raw: "5.1.1-1819727")
E: uses-sdk (line=7)
  A: android:minSdkVersion(0x0101020c)=(type 0x10)0xa
  A: android:targetSdkVersion(0x01010270)=(type 0x10)0x16
E: uses-permission (line=11)
  A: android:name(0x01010003)="android.permission.INTERNET" (Raw:
"android.permission.INTERNET")
E: uses-permission (line=12)
  A: android:name(0x01010003)="android.permission.CAMERA" (Raw:
"android.permission.CAMERA")
.....
```

If the Android application is valid, the output should not contain the following: `A: android:name(0x01010003)="android.permission.BIND_DEVICE_ADMIN" (Raw: "android.permission.BIND_DEVICE_ADMIN")`.

For more information, see [Working with Android Tests in AWS Device Farm \(p. 46\)](#).

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_FAIL

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 JUnit \(p. 46\)](#) or [Appium Java JUnit \(p. 62\)](#).

APPIUM_JAVA_JUNIT_TEST_PACKAGE_DEPENDENCY_DIR_M

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 Java JUnit \(p. 46\)](#) or [Appium Java JUnit \(p. 62\)](#).

APPIUM_JAVA_JUNIT_TEST_PACKAGE_JAR_MISSING_IN_DEP

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

Warning

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

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

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

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

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

```
$ tree .
```

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

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

For more information, see [Appium Java JUnit \(p. 46\)](#) or [Appium Java JUnit \(p. 62\)](#).

APPIUM_JAVA_JUNIT_TEST_PACKAGE_TESTS_JAR_FILE_MIS

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

Warning

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

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

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

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

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

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find at least one *jar* file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *-tests.jar*.

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

For more information, see [Appium Java JUnit \(p. 46\)](#) or [Appium Java JUnit \(p. 62\)](#).

APPIUM_JAVA_JUNIT_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:

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

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

```
$ tree .
```

You should find at least one jar file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *-tests.jar*.

```
.
|- acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built
from the ./src/main directory)
|- acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything
built from the ./src/test directory)
|- zip-with-dependencies.zip (this .zip file contains all of the items)
```

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

3. After you successfully extract the files, you should find at least one class in the working directory tree by running the command:

```
$ tree .
```

You should see output like this:

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

For more information, see [Appium Java JUnit \(p. 46\)](#) or [Appium Java JUnit \(p. 62\)](#).

APPIUM_JAVA_JUNIT_TEST_PACKAGE_JUNIT_VERSION_VAL

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

Warning

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

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

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

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

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

```
tree .
```

The output should look like this:

```
.
├─ acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built
  from the ./src/main directory)
├─ acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything
  built from the ./src/test directory)
```

```
|- zip-with-dependencies.zip (this .zip file contains all of the items)
^- dependency-jars (this is the directory that contains all of your dependencies,
  built as JAR files)
  |- junit-4.10.jar
  |- com.some-dependency.bar-4.1.jar
  |- com.another-dependency.thing-1.0.jar
  |- joda-time-2.7.jar
  ^- log4j-1.2.14.jar
```

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

For more information, see [Appium Java JUnit \(p. 46\)](#) and [Appium Java JUnit \(p. 62\)](#)

APPIUM_JAVA_JUNIT_TEST_PACKAGE_INVALID_JUNIT_VERS

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

Warning

We found the JUnit version was lower than the minimum version 4.10 we support. Please change the JUnit version and try again.

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

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

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

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

```
$ tree .
```

You should find a JUnit dependency file like `junit-4.10.jar` in our example and its version number, which in our example is 4.10:

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

Note

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

For more information, see [Appium Java JUnit \(p. 46\)](#) or [Appium Java JUnit \(p. 62\)](#).

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:

```
$ 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 JUnit \(p. 79\)](#).

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_DEPENDENCY_D

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 Java JUnit \(p. 79\)](#).

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:

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

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

```
$ tree .
```

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

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

```
└─ log4j-1.2.14.jar
```

For more information, see [Appium Java JUnit \(p. 79\)](#).

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_TESTS_JAR_FILE_MISSING

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

Warning

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

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

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

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

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

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find at least one *jar* file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *-tests.jar*.

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

For more information, see [Appium Java JUnit \(p. 79\)](#).

APPIUM_WEB_JAVA_JUNIT_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:

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

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

```
$ tree .
```

You should find at least one jar file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *-tests.jar*.

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

3. After you successfully extract the files, you should find at least one class in the working directory tree by running the command:

```
$ tree .
```

You should see output like this:

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

For more information, see [Appium Java JUnit \(p. 79\)](#).

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_JUNIT_VERSION

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

Warning

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

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

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


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

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

```
tree .
```

The output should look like this:

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

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

For more information, see [Appium Java JUnit \(p. 79\)](#).

APPIUM_WEB_JAVA_JUNIT_TEST_PACKAGE_INVALID_JUNIT

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

Warning

We found the JUnit version was lower than the minimum version 4.10 we support. Please change the JUnit version and try again.

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

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

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

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

```
$ tree .
```

You should find a JUnit dependency file like `junit-4.10.jar` in our example and its version number, which in our example is 4.10:

```
.
|- acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built
from the ./src/main directory)
|- acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing everything
built from the ./src/test directory)
```

```
| - zip-with-dependencies.zip (this .zip file contains all of the items)
|- dependency-jars (this is the directory that contains all of your dependencies, built
  as JAR files)
  | - junit-4.10.jar
  | - com.some-dependency.bar-4.1.jar
  | - com.another-dependency.thing-1.0.jar
  | - joda-time-2.7.jar
  ` - log4j-1.2.14.jar
```

Note

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

For more information, see [Appium Java JUnit \(p. 79\)](#).

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 \(p. 50\)](#) or [Appium Java TestNG \(p. 66\)](#).

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 Java TestNG \(p. 50\)](#) or [Appium Java TestNG \(p. 66\)](#).

APPIUM_JAVA_TESTNG_TEST_PACKAGE_JAR_MISSING_IN_DIRECTORY

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

Warning

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

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

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

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

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

```
$ tree .
```

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

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

For more information, see [Appium Java TestNG \(p. 50\)](#) or [Appium Java TestNG \(p. 66\)](#).

APPIUM_JAVA_TESTNG_TEST_PACKAGE_TESTS_JAR_FILE_M

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

Warning

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

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

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

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

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

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find at least one *jar* file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *-tests.jar*.

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

```
└─ log4j-1.2.14.jar
```

For more information, see [Appium Java TestNG \(p. 50\)](#) or [Appium Java TestNG \(p. 66\)](#).

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:

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

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

```
$ tree .
```

You should find at least one jar file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *-tests.jar*.

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

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

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

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

```
$ tree .
```

You should find at least one class in the working directory tree:

```
.
├─ acme-android-appium-1.0-SNAPSHOT.jar (this is the JAR containing everything built
  from the ./src/main directory)
├─ acme-android-appium-1.0-SNAPSHOT-tests.jar (this is the JAR containing
  everything built from the ./src/test directory)
├─ one-class-file.class
├─ folder
└─ └─ another-class-file.class
```

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

For more information, see [Appium Java TestNG \(p. 50\)](#) or [Appium Java TestNG \(p. 66\)](#).

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 \(p. 81\)](#).

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_DEPENDENCY

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 Java TestNG \(p. 81\)](#).

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_JAR_MISSING

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

Warning

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

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

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

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

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

```
$ tree .
```

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

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

For more information, see [Appium Java TestNG \(p. 81\)](#).

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_TESTS_JAR_F

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

Warning

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

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

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

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

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

```
$ tree .
```

If the Appium Java JUnit package is valid, you will find at least one *jar* file like *acme-android-appium-1.0-SNAPSHOT-tests.jar* in our example. The file's name may be different, but it should end with *-tests.jar*.

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

For more information, see [Appium Java TestNG \(p. 81\)](#).

APPIUM_WEB_JAVA_TESTNG_TEST_PACKAGE_CLASS_FILE_M

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

Warning

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

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

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

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

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

```
$ tree .
```

You should find at least one jar file like `acme-android-appium-1.0-SNAPSHOT-tests.jar` in our example. The file's name may be different, but it should end with `-tests.jar`.

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

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

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

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

```
$ tree .
```

You should find at least one class in the working directory tree:

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

```
^- log4j-1.2.14.jar
```

For more information, see [Appium Java TestNG \(p. 81\)](#).

Troubleshooting Appium Python Tests in AWS Device Farm

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

APPIUM_PYTHON_TEST_PACKAGE_UNZIP_FAILED

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

Warning

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

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

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

```
$ unzip test_bundle.zip
```

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

```
$ tree .
```

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

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|   |-- test_unittest.py
|-- wheelhouse (directory)
|   |-- Appium_Python_Client-0.20-cp27-none-any.whl
|   |-- py-1.4.31-py2.py3-none-any.whl
|   |-- pytest-2.9.0-py2.py3-none-any.whl
|   |-- selenium-2.52.0-cp27-none-any.whl
|   |-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see [Appium Python \(p. 54\)](#) or [Appium Python \(p. 70\)](#).

APPIUM_PYTHON_TEST_PACKAGE_DEPENDENCY_WHEEL_M

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 \(p. 54\)](#) or [Appium Python \(p. 70\)](#).

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 \(p. 54\)](#) or [Appium Python \(p. 70\)](#).

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 \(p. 54\)](#) or [Appium Python \(p. 70\)](#).

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 \(p. 54\)](#) or [Appium Python \(p. 70\)](#).

APPIUM_PYTHON_TEST_PACKAGE_REQUIREMENTS_TXT_FILE_MISSING

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

Warning

We could not find the `requirements.txt` file inside your test package. Please unzip your test package, verify that the `requirements.txt` file is inside the package, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

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

```
$ unzip test_bundle.zip
```

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

```
$ tree .
```

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

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
```

```
|      ^-- test_unittest.py
|-- wheelhouse (directory)
|   |-- Appium_Python_Client-0.20-cp27-none-any.whl
|   |-- py-1.4.31-py2.py3-none-any.whl
|   |-- pytest-2.9.0-py2.py3-none-any.whl
|   |-- selenium-2.52.0-cp27-none-any.whl
|   ^-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see [Appium Python \(p. 54\)](#) or [Appium Python \(p. 70\)](#).

APPIUM_PYTHON_TEST_PACKAGE_INVALID_PYTEST_VERSION

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

Warning

We found the pytest version was lower than the minimum version 2.8.0 we support. Please change the pytest version inside the requirements.txt file, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

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

```
$ unzip test_bundle.zip
```

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

```
$ tree .
```

You should find the *requirements.txt* file inside the working directory.

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|   ^--test_unittest.py
|-- wheelhouse (directory)
|   |-- Appium_Python_Client-0.20-cp27-none-any.whl
|   |-- py-1.4.31-py2.py3-none-any.whl
|   |-- pytest-2.9.0-py2.py3-none-any.whl
|   |-- selenium-2.52.0-cp27-none-any.whl
|   ^-- wheel-0.26.0-py2.py3-none-any.whl
```

3. To get the pytest version, you can run the following command:

```
$ grep "pytest" requirements.txt
```

You should find output like the following:

```
pytest==2.9.0
```

It shows the pytest version, which in this example is 2.9.0. If the Appium Python package is valid, the pytest version should be larger than or equal to 2.8.0.

For more information, see [Appium Python \(p. 54\)](#) or [Appium Python \(p. 70\)](#).

APPIUM_PYTHON_TEST_PACKAGE_INSTALL_DEPENDENCY_WHEELS_FAILED

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

Warning

We failed to install the dependency wheels. Please unzip your test package and then open the requirements.txt file and the wheelhouse directory, verify that the dependency wheels specified in the requirements.txt file exactly match the dependency wheels inside the wheelhouse directory, and try again.

We strongly recommend that you set up [Python virtualenv](#) for packaging tests. Here is an example flow of creating a virtual environment using Python virtualenv and then activating it:

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

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

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

```
$ unzip test_bundle.zip
```

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

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

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

```
Ignoring indexes: https://pypi.python.org/simple
Collecting Appium-Python-Client==0.20 (from -r ./requirements.txt (line 1))
Collecting py==1.4.31 (from -r ./requirements.txt (line 2))
Collecting pytest==2.9.0 (from -r ./requirements.txt (line 3))
Collecting selenium==2.52.0 (from -r ./requirements.txt (line 4))
Collecting wheel==0.26.0 (from -r ./requirements.txt (line 5))
Installing collected packages: selenium, Appium-Python-Client, py, pytest, wheel
  Found existing installation: wheel 0.29.0
    Uninstalling wheel-0.29.0:
      Successfully uninstalled wheel-0.29.0
Successfully installed Appium-Python-Client-0.20 py-1.4.31 pytest-2.9.0 selenium-2.52.0 wheel-0.26.0
```

3. To deactivate the virtual environment, you can run the following command:

```
$ deactivate
```

For more information, see [Appium Python \(p. 54\)](#) or [Appium Python \(p. 70\)](#).

APPIUM_PYTHON_TEST_PACKAGE_PYTEST_COLLECT_FAIL

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

Warning

We failed to collect tests in the tests directory. Please unzip your test package, verify that the test package is valid by running the command `py.test --collect-only <path to your tests directory>`, and try again after the command does not print any error.

We strongly recommend that you set up [Python virtualenv](#) for packaging tests. Here is an example flow of creating a virtual environment using Python virtualenv and then activating it:

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

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

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

```
$ unzip test_bundle.zip
```

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

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

3. To collect tests, you can run the following command:

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

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

```
===== test session starts =====
platform darwin -- Python 2.7.11, pytest-2.9.0, py-1.4.31, pluggy-0.3.1
rootdir: /Users/zhen/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 \(p. 54\)](#) or [Appium Python \(p. 70\)](#).

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 \(p. 83\)](#).

APPIUM_WEB_PYTHON_TEST_PACKAGE_DEPENDENCY_WH

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 \(p. 83\)](#).

APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_PLATFORM

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

Warning

We found at least one wheel file specified a platform that we do not support. Please unzip your test package and then open the wheelhouse directory, verify that names of wheel files end with `-any.whl` or `-linux_x86_64.whl`, and try again.

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

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

```
$ unzip test_bundle.zip
```

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

```
$ tree .
```

If the Appium Python package is valid, you will find at least one `.whl` dependent file like the highlighted files inside the `wheelhouse` directory. The file's name may be different, but it should end with `-any.whl` or `-linux_x86_64.whl`, which specifies the platform. Any other platforms like `windows` are not supported.

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|   |-- test_unittest.py
|-- wheelhouse (directory)
|   |-- Appium_Python_Client-0.20-cp27-none-any.whl
|   |-- py-1.4.31-py2.py3-none-any.whl
|   |-- pytest-2.9.0-py2.py3-none-any.whl
|   |-- selenium-2.52.0-cp27-none-any.whl
|   |-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see [Appium Python \(p. 83\)](#).

APPIUM_WEB_PYTHON_TEST_PACKAGE_TEST_DIR_MISSING

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

Warning

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

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

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

```
$ unzip test_bundle.zip
```

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

```
$ tree .
```

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

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|   |-- test_unittest.py
|-- wheelhouse (directory)
|   |-- Appium_Python_Client-0.20-cp27-none-any.whl
|   |-- py-1.4.31-py2.py3-none-any.whl
|   |-- pytest-2.9.0-py2.py3-none-any.whl
|   |-- selenium-2.52.0-cp27-none-any.whl
|   |-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see [Appium Python \(p. 83\)](#).

APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_TEST_FILE

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 \(p. 83\)](#).

APPIUM_WEB_PYTHON_TEST_PACKAGE_REQUIREMENTS_TXT_FILE_MISSING

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

Warning

We could not find the requirements.txt file inside your test package. Please unzip your test package, verify that the requirements.txt file is inside the package, and try again.

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

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

```
$ unzip test_bundle.zip
```

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

```
$ tree .
```

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

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|   |-- test_unittest.py
|-- wheelhouse (directory)
|   |-- Appium_Python_Client-0.20-cp27-none-any.whl
|   |-- py-1.4.31-py2.py3-none-any.whl
|   |-- pytest-2.9.0-py2.py3-none-any.whl
|   |-- selenium-2.52.0-cp27-none-any.whl
|   |-- wheel-0.26.0-py2.py3-none-any.whl
```

For more information, see [Appium Python \(p. 83\)](#).

APPIUM_WEB_PYTHON_TEST_PACKAGE_INVALID_PYTEST_V

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

Warning

We found the pytest version was lower than the minimum version 2.8.0 we support. Please change the pytest version inside the requirements.txt file, and try again.

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

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

```
$ unzip test_bundle.zip
```

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

```
$ tree .
```

You should find the `requirements.txt` file inside the working directory.

```
.
|-- requirements.txt
|-- test_bundle.zip
|-- tests (directory)
|   |--test_unittest.py
|-- wheelhouse (directory)
|   |-- Appium_Python_Client-0.20-cp27-none-any.whl
|   |-- py-1.4.31-py2.py3-none-any.whl
|   |-- pytest-2.9.0-py2.py3-none-any.whl
|   |-- selenium-2.52.0-cp27-none-any.whl
|   |-- wheel-0.26.0-py2.py3-none-any.whl
```

3. To get the pytest version, you can run the following command:

```
$ grep "pytest" requirements.txt
```

You should find output like the following:

```
pytest==2.9.0
```

It shows the pytest version, which in this example is 2.9.0. If the Appium Python package is valid, the pytest version should be larger than or equal to 2.8.0.

For more information, see [Appium Python \(p. 83\)](#).

APPIUM_WEB_PYTHON_TEST_PACKAGE_INSTALL_DEPENDE

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

Warning

We failed to install the dependency wheels. Please unzip your test package and then open the requirements.txt file and the wheelhouse directory, verify that the dependency wheels specified in the requirements.txt file exactly match the dependency wheels inside the wheelhouse directory, and try again.

We strongly recommend that you set up [Python virtualenv](#) for packaging tests. Here is an example flow of creating a virtual environment using Python virtualenv and then activating it:

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

Make sure that you can unzip the test package without errors. In the following example, the package's name is **test_bundle.zip**.

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

```
$ unzip test_bundle.zip
```

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

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

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

```
Ignoring indexes: https://pypi.python.org/simple
Collecting Appium-Python-Client==0.20 (from -r ./requirements.txt (line 1))
Collecting py==1.4.31 (from -r ./requirements.txt (line 2))
Collecting pytest==2.9.0 (from -r ./requirements.txt (line 3))
Collecting selenium==2.52.0 (from -r ./requirements.txt (line 4))
Collecting wheel==0.26.0 (from -r ./requirements.txt (line 5))
Installing collected packages: selenium, Appium-Python-Client, py, pytest, wheel
  Found existing installation: wheel 0.29.0
    Uninstalling wheel-0.29.0:
      Successfully uninstalled wheel-0.29.0
Successfully installed Appium-Python-Client-0.20 py-1.4.31 pytest-2.9.0 selenium-2.52.0 wheel-0.26.0
```

3. To deactivate the virtual environment, you can run the following command:

```
$ deactivate
```

For more information, see [Appium Python \(p. 83\)](#).

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:

```
$ 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/zhen/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 \(p. 83\)](#).

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
|   |-- step_definitions (directory)
|       |-- (.rb files)
|   |-- support (directory)
|       |-- (.rb files)
|-- (any other supporting files)
```

For more information, see [Calabash \(p. 57\)](#) or [Calabash \(p. 73\)](#).

CALABASH_TEST_PACKAGE_FEATURES_DIR_MISSING_FEATU

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:

```
$ 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 *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 \(p. 57\)](#) or [Calabash \(p. 73\)](#).

CALABASH_TEST_PACKAGE_FEATURE_FILE_MISSING

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

Warning

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

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

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

```
$ unzip features.zip
```

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

```
$ tree .
```

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

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

For more information, see [Calabash \(p. 57\)](#) or [Calabash \(p. 73\)](#).

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 \(p. 57\)](#) or [Calabash \(p. 73\)](#).

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.

```
.
|-- 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 \(p. 57\)](#) or [Calabash \(p. 73\)](#).

CALABASH_TEST_PACKAGE_RUBY_FILE_MISSING_IN_STEP_D

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

Warning

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

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

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

```
$ unzip features.zip
```

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

```
$ tree .
```

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

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

For more information, see [Calabash \(p. 57\)](#) or [Calabash \(p. 73\)](#).

CALABASH_TEST_PACKAGE_RUBY_FILE_MISSING_IN_SUPPORT_DIR

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

Warning

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

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

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

```
$ unzip features.zip
```

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 *support* directory.

```
.
|-- features (directory)
    |-- my-feature-1-file-name.feature
    |-- my-feature-2-file-name.feature
    |-- my-feature-N-file-name.feature
    |-- step_definitions (directory)
```

```
|      |-- (.rb files)
|-- support (directory)
|      |-- one-ruby.rb
|      |-- folder
|      |-- another-ruby.rb
|-- (any other supporting files)
|-- (any other supporting files)
```

For more information, see [Calabash \(p. 57\)](#) or [Calabash \(p. 73\)](#).

CALABASH_TEST_PACKAGE_EMBEDDED_SERVER_MISSING

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

Warning

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

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

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

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

```
$ 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 \(p. 57\)](#) or [Calabash \(p. 73\)](#).

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:

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

For more information, see [Calabash \(p. 57\)](#) or [Calabash \(p. 73\)](#).

Troubleshooting Instrumentation Tests in AWS Device Farm

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

INSTRUMENTATION_TEST_PACKAGE_UNZIP_FAILED

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

Warning

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

Make sure that you can unzip the test package without errors. In the following example, the package's name is **app-debug-androidTest-unaligned.apk**.

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

```
$ unzip app-debug-androidTest-unaligned.apk
```

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

```
$ tree .
```

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

```
.
|-- AndroidManifest.xml
|-- classes.dex
|-- resources.arsc
|-- LICENSE-junit.txt
|-- junit (directory)
`-- META-INF (directory)
```

For more information, see [Instrumentation \(p. 59\)](#).

INSTRUMENTATION_TEST_PACKAGE_AAPT_DEBUG_BADGIN

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-implieed-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 \(p. 59\)](#).

INSTRUMENTATION_TEST_PACKAGE_INSTRUMENTATION_R

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

Warning

We could not find the instrumentation runner value in the AndroidManifest.xml. Please verify the test package is valid by running the command "aapt dump xmltree <path to your test package> AndroidManifest.xml", and try again after finding the instrumentation runner value behind the keyword "instrumentation."

During the upload validation process, Device Farm parses out the instrumentation runner value from the XML parse tree for an XML file contained within the package. You can use the following command: `aapt dump xmltree <path to your package> AndroidManifest.xml`.

Make sure that you can run this command on your Instrumentation test package and find the instrumentation value successfully.

In the following example, the package's name is **app-debug-androidTest-unaligned.apk**.

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

```
$ aapt dump xmltree app-debug-androidTest-unaligned.apk AndroidManifest.xml | grep -A5  
"instrumentation"
```

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

```
E: instrumentation (line=9)  
  A: android:label(0x01010001)="Tests for  
com.amazon.aws.adf.android.referenceapp" (Raw: "Tests for  
com.amazon.aws.adf.android.referenceapp")  
  A:  
  android:name(0x01010003)="android.support.test.runner.AndroidJUnitRunner" (Raw:  
"android.support.test.runner.AndroidJUnitRunner")  
  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 \(p. 59\)](#).

INSTRUMENTATION_TEST_PACKAGE_AAPT_DUMP_XMLTREE

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

Warning

We could not find the valid AndroidManifest.xml in your test package. Please verify that the test package is valid by running the command "aapt dump xmltree <path to your test package> AndroidManifest.xml", and try again after the command does not print any error.

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

Make sure that you can run this command on your instrumentation test package successfully.

In the following example, the package's name is **app-debug-androidTest-unaligned.apk**.

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

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

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

```
N: android=http://schemas.android.com/apk/res/android
  E: manifest (line=2)
    A: package="com.amazon.aws.adf.android.referenceapp.test" (Raw:
"com.amazon.aws.adf.android.referenceapp.test")
    A: platformBuildVersionCode=(type 0x10)0x16 (Raw: "22")
    A: platformBuildVersionName="5.1.1-1819727" (Raw: "5.1.1-1819727")
    E: uses-sdk (line=5)
      A: android:minSdkVersion(0x0101020c)=(type 0x10)0x9
      A: android:targetSdkVersion(0x01010270)=(type 0x10)0x16
    E: instrumentation (line=9)
      A: android:label(0x01010001)="Tests for
com.amazon.aws.adf.android.referenceapp" (Raw: "Tests for
com.amazon.aws.adf.android.referenceapp")
      A:
      android:name(0x01010003)="android.support.test.runner.AndroidJUnitRunner" (Raw:
"android.support.test.runner.AndroidJUnitRunner")
      A:
      android:targetPackage(0x01010021)="com.amazon.aws.adf.android.referenceapp" (Raw:
"com.amazon.aws.adf.android.referenceapp")
      A: android:handleProfiling(0x01010022)=(type 0x12)0x0
      A: android:functionalTest(0x01010023)=(type 0x12)0x0
    E: application (line=16)
      A: android:label(0x01010001)=@0x7f020000
      A: android:debuggable(0x0101000f)=(type 0x12)0xffffffff
    E: uses-library (line=17)
      A: android:name(0x01010003)="android.test.runner" (Raw: "android.test.runner")
```

For more information, see [Instrumentation \(p. 59\)](#).

INSTRUMENTATION_TEST_PACKAGE_TEST_PACKAGE_NAME

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 \(p. 59\)](#).

Troubleshooting iOS Application Tests in AWS Device Farm

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

Note

The instructions below are based on Linux x86_64 and Mac.

IOS_APP_UNZIP_FAILED

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

Warning

We could not open your application. Please verify that the file is valid and try again.

Make sure that you can unzip the application package without errors. In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

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

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

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

```
$ tree .
```

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

```
.  
|-- Payload (directory)  
    |-- AWSDeviceFarmiOSReferenceApp.app (directory)  
        |-- Info.plist  
        |-- (any other files)
```

For more information, see [Working with iOS Tests in AWS Device Farm \(p. 62\)](#).

IOS_APP_PAYLOAD_DIR_MISSING

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

Warning

We could not find the Payload directory inside your application. Please unzip your application, verify that the Payload directory is inside the package, and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

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

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

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

```
$ tree .
```

If the iOS application package is valid, you will find the *Payload* directory inside the working directory.

```
.
|-- Payload (directory)
    |-- AWSDeviceFarmiOSReferenceApp.app (directory)
        |-- Info.plist
        |-- (any other files)
```

For more information, see [Working with iOS Tests in AWS Device Farm \(p. 62\)](#).

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 \(p. 62\)](#).

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 \(p. 62\)](#).

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 \(p. 62\)](#).

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
```

- Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/Info.plist')
print info_plist['CFBundleSupportedPlatforms']
```

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

```
['iPhoneOS']
```

For more information, see [Working with iOS Tests in AWS Device Farm \(p. 62\)](#).

IOS_APP_WRONG_PLATFORM_DEVICE_VALUE

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

Warning

We found the platform device value was wrong in the Info.plist file. Please unzip your application and then open Info.plist file inside the .app directory, verify that the value of the key "CFBundleSupportedPlatforms" does not contain the keyword "simulator", and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

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

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

- 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)
```

- 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
```

- 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 \(p. 62\)](#).

IOS_APP_FORM_FACTOR_VALUE_MISSING

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

Warning

We could not find the form factor value in the Info.plist file. Please unzip your application and then open Info.plist file inside the .app directory, verify that the key "UIDeviceFamily" is specified, and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

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

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

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

```
$ tree .
```

You should find the *Info.plist* file inside an *.app* directory like *AWSDeviceFarmiOSReferenceApp.app* in our example:

```
.
|-- Payload (directory)
    |-- AWSDeviceFarmiOSReferenceApp.app (directory)
        |-- Info.plist
        |-- (any other files)
```

3. To find the form factor value, you can open Info.plist using Xcode or Python.

For Python, you can install the `biplist` module by running the following command:

```
$ pip install biplist
```

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

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/Info.plist')
print info_plist['UIDeviceFamily']
```

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

```
[1, 2]
```

For more information, see [Working with iOS Tests in AWS Device Farm \(p. 62\)](#).

IOS_APP_PACKAGE_NAME_VALUE_MISSING

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

Warning

We could not find the package name value in the Info.plist file. Please unzip your application and then open Info.plist file inside the .app directory, verify that the key "CFBundleIdentifier" is specified, and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

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

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

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

```
$ tree .
```

You should find the *Info.plist* file inside an *.app* directory like *AWSDeviceFarmiOSReferenceApp.app* in our example:

```
.
|-- Payload (directory)
    |-- AWSDeviceFarmiOSReferenceApp.app (directory)
        |-- Info.plist
        |-- (any other files)
```

3. To find the package name value, you can open Info.plist using Xcode or Python.

For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

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

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/Info.plist')
print info_plist['CFBundleIdentifier']
```

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

```
Amazon.AWSDeviceFarmiOSReferenceApp
```

For more information, see [Working with iOS Tests in AWS Device Farm \(p. 62\)](#).

IOS_APP_EXECUTABLE_VALUE_MISSING

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

Warning

We could not find the executable value in the Info.plist file. Please unzip your application and then open Info.plist file inside the .app directory, verify that the key "CFBundleExecutable" is specified, and try again.

In the following example, the package's name is **AWSDeviceFarmiOSReferenceApp.ipa**.

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

```
$ unzip AWSDeviceFarmiOSReferenceApp.ipa
```

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

```
$ tree .
```

You should find the *Info.plist* file inside an *.app* directory like *AWSDeviceFarmiOSReferenceApp.app* in our example:

```
.
|-- Payload (directory)
    |-- AWSDeviceFarmiOSReferenceApp.app (directory)
        |-- Info.plist
        |-- (any other files)
```

3. To find the executable value, you can open Info.plist using Xcode or Python.

For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

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

```
import biplist
info_plist = biplist.readPlist('Payload/AWSDeviceFarmiOSReferenceApp-cal.app/
Info.plist')
print info_plist['CFBundleExecutable']
```

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

```
AWSDeviceFarmiOSReferenceApp
```

For more information, see [Working with iOS Tests in AWS Device Farm \(p. 62\)](#).

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 \(p. 60\)](#).

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. 76).

XCTEST_TEST_PACKAGE_XCTEST_DIR_MISSING

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

Warning

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

In the following example, the package's name is **swiftExampleTests.xctest-1.zip**.

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

```
$ unzip swiftExampleTests.xctest-1.zip
```

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

```
$ tree .
```

If the XCTest package is valid, you will find a directory with a name similar to *swiftExampleTests.xctest* inside the working directory. The name should end with *.xctest*.

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

For more information, see [XCTest](#) (p. 76).

XCTEST_TEST_PACKAGE_PLIST_FILE_MISSING

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

Warning

We could not find the Info.plist file inside the .xctest directory. Please unzip your test package and then open the .xctest directory, verify that the Info.plist file is inside the directory, and try again.

In the following example, the package's name is **swiftExampleTests.xctest-1.zip**.

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

```
$ unzip swiftExampleTests.xctest-1.zip
```

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

```
$ tree .
```

If the XCTest package is valid, you will find the *Info.plist* file inside the *.xctest* directory. In our example below, the directory is called *swiftExampleTests.xctest*.

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

For more information, see [XCTest](#) (p. 76).

XCTEST_TEST_PACKAGE_PACKAGE_NAME_VALUE_MISSING

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

Warning

We could not find the package name value in the Info.plist file. Please unzip your test package and then open Info.plist file, verify that the key "CFBundleIdentifier" is specified, and try again.

In the following example, the package's name is **swiftExampleTests.xctest-1.zip**.

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

```
$ unzip swiftExampleTests.xctest-1.zip
```

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

```
$ tree .
```

You should find the *Info.plist* file inside an *.xctest* directory like *swiftExampleTests.xctest* in our example:

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

3. To find the package name value, you can open Info.plist using Xcode or Python.

For Python, you can install the *biplist* module by running the following command:

```
$ pip install biplist
```

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

```
import biplist  
info_plist = biplist.readPlist('swiftExampleTests.xctest/Info.plist')  
print info_plist['CFBundleIdentifier']
```

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

```
com.amazon.kanapka.swiftExampleTests
```

For more information, see [XCTest](#) (p. 76).

XCTEST_TEST_PACKAGE_EXECUTABLE_VALUE_MISSING

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

Warning

We could not find the executable value in the Info.plist file. Please unzip your test package and then open Info.plist file, verify that the key "CFBundleExecutable" is specified, and try again.

In the following example, the package's name is `swiftExampleTests.xctest-1.zip`.

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

```
$ unzip swiftExampleTests.xctest-1.zip
```

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

```
$ tree .
```

You should find the `Info.plist` file inside an `.xctest` directory like `swiftExampleTests.xctest` in our example:

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

3. To find the package name value, you can open Info.plist using Xcode or Python.

For Python, you can install the `biplist` module by running the following command:

```
$ pip install biplist
```

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

```
import biplist
info_plist = biplist.readPlist('swiftExampleTests.xctest/Info.plist')
print info_plist['CFBundleExecutable']
```

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

```
swiftExampleTests
```

For more information, see [XCTest](#) (p. 76).

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:

```
$ unzip swift-sample-UI.ipa
```

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

```
$ tree .
```

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

```
.
|-- Payload (directory)
    |-- swift-sampleUITests-Runner.app (directory)
        |-- Info.plist
        |-- Plugins (directory)
            |-- swift-sampleUITests.xctest (directory)
                |-- Info.plist
                |-- (any other files)
            |-- (any other files)
```

For more information, see [XCTest UI \(p. 77\)](#).

XCTEST_UI_TEST_PACKAGE_PAYLOAD_DIR_MISSING

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

Warning

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

In the following example, the package's name is **swift-sample-UI.ipa**.

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

```
$ unzip swift-sample-UI.ipa
```

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

```
$ tree .
```

If the XCTest UI package is valid, you will find the **Payload** directory inside the working directory.

```
.
|-- Payload (directory)
    |-- swift-sampleUITests-Runner.app (directory)
        |-- Info.plist
        |-- Plugins (directory)
            |-- swift-sampleUITests.xctest (directory)
                |-- Info.plist
                |-- (any other files)
        |-- (any other files)
```

For more information, see [XCTest UI \(p. 77\)](#).

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 \(p. 77\)](#).

XCTEST_UI_TEST_PACKAGE_PLUGINS_DIR_MISSING

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

Warning

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

In the following example, the package's name is **swift-sample-UI.ipa**.

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

```
$ unzip swift-sample-UI.ipa
```

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

```
$ tree .
```

If the XCTest UI package is valid, you will find the *Plugins* directory inside an *.app* directory. In our example, the directory is called *swift-sampleUITests-Runner.app*.

```
.
|-- Payload (directory)
    |-- swift-sampleUITests-Runner.app (directory)
        |-- Info.plist
        |-- Plugins (directory)
            |-- swift-sampleUITests.xctest (directory)
                |-- Info.plist
                |-- (any other files)
            |-- (any other files)
```

For more information, see [XCTest UI \(p. 77\)](#).

XCTEST_UI_TEST_PACKAGE_XCTEST_DIR_MISSING_IN_PLUG

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)
```

```
^-- (any other files)
```

For more information, see [XCTest UI \(p. 77\)](#).

XCTEST_UI_TEST_PACKAGE_PLIST_FILE_MISSING

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

Warning

We could not find the Info.plist file inside the .app directory. Please unzip your test package and then open the .app directory, verify that the Info.plist file is inside the directory, and try again.

In the following example, the package's name is **swift-sample-UI.ipa**.

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

```
$ unzip swift-sample-UI.ipa
```

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

```
$ tree .
```

If the XCTest UI package is valid, you will find the *Info.plist* file inside the *.app* directory. In our example below, the directory is called *swift-sampleUITests-Runner.app*.

```
.
|-- Payload (directory)
    |-- swift-sampleUITests-Runner.app (directory)
        |-- Info.plist
        |-- Plugins (directory)
            |-- swift-sampleUITests.xctest (directory)
                |-- Info.plist
                |-- (any other files)
            |-- (any other files)
        |-- (any other files)
```

For more information, see [XCTest UI \(p. 77\)](#).

XCTEST_UI_TEST_PACKAGE_PLIST_FILE_MISSING_IN_XCTES

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:

```
$ unzip swift-sample-UI.ipa
```

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


```
$ tree .
```

If the XCTest UI package is valid, you will find the *Info.plist* file inside the *.xctest* directory. In our example below, the directory is called *swift-sampleUITests.xctest*.

```
.
|-- Payload (directory)
    |-- swift-sampleUITests-Runner.app (directory)
        |-- Info.plist
        |-- Plugins (directory)
            |-- swift-sampleUITests.xctest (directory)
                |-- Info.plist
                |-- (any other files)
            |-- (any other files)
        |-- (any other files)
    |-- (any other files)
```

For more information, see [XCTest UI \(p. 77\)](#).

XCTEST_UI_TEST_PACKAGE_CPU_ARCHITECTURE_VALUE_M

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:

```
$ 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)
    |-- (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
```

- Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['UIRequiredDeviceCapabilities']
```

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

```
['armv7']
```

For more information, see [XCTest UI \(p. 77\)](#).

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**.

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

```
$ unzip swift-sample-UI.ipa
```

- 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)
```

- 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
```

- Next, open Python and run the following command:

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['CFBundleSupportedPlatforms']
```

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

```
['iPhoneOS']
```

For more information, see [XCTest UI \(p. 77\)](#).

XCTEST_UI_TEST_PACKAGE_WRONG_PLATFORM_DEVICE_V

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:

```
$ 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 platform value, you can open Info.plist using Xcode or Python.

For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

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

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['CFBundleSupportedPlatforms']
```

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

```
['iPhoneOS']
```

If the XCTest UI package is valid, the value should not contain the keyword `simulator`.

For more information, see [XCTest UI \(p. 77\)](#).

XCTEST_UI_TEST_PACKAGE_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. Please unzip your test package and then open the Info.plist file inside the .app directory, verify that the key "UIDeviceFamily" is specified, and try again.

In the following example, the package's name is **swift-sample-UI.ipa**.

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

```
$ unzip swift-sample-UI.ipa
```

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

```
$ tree .
```

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

```
.
|-- Payload (directory)
    |-- swift-sampleUITests-Runner.app (directory)
        |-- Info.plist
        |-- Plugins (directory)
            |-- swift-sampleUITests.xctest (directory)
                |-- Info.plist
                |-- (any other files)
            |-- (any other files)
        |-- (any other files)
```

3. To find the form factor value, you can open Info.plist using Xcode or Python.

For Python, you can install the `biplist` module by running the following command:

```
$ pip install biplist
```

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

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['UIDeviceFamily']
```

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

```
[1, 2]
```

For more information, see [XCTest UI \(p. 77\)](#).

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 \(p. 77\)](#).

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:

```
$ 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 executable value, you can open Info.plist using Xcode or Python.

For Python, you can install the biplist module by running the following command:

```
$ pip install biplist
```

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

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Info.plist')
print info_plist['CFBundleExecutable']
```

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

```
XCTRunner
```

For more information, see [XCTest UI \(p. 77\)](#).

XCTEST_UI_TEST_PACKAGE_TEST_PACKAGE_NAME_VALUE_

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

Warning

We could not find the package name value in the Info.plist file inside the .xctest directory. Please unzip your test package and then open the Info.plist file inside the .xctest directory, verify that the key "CFBundleIdentifier" is specified, and try again.

In the following example, the package's name is **swift-sample-UI.ipa**.

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

```
$ unzip swift-sample-UI.ipa
```

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

```
$ tree .
```

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

```
.
|-- Payload (directory)
    |-- swift-sampleUITests-Runner.app (directory)
        |-- Info.plist
        |-- Plugins (directory)
            |-- swift-sampleUITests.xctest (directory)
                |-- Info.plist
                |-- (any other files)
            |-- (any other files)
```

3. To find the package name value, you can open Info.plist using Xcode or Python.

For Python, you can install the `biplist` module by running the following command:

```
$ pip install biplist
```

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

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Plugins/swift-
sampleUITests.xctest/Info.plist')
print info_plist['CFBundleIdentifier']
```

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

```
com.amazon.swift-sampleUITests
```

For more information, see [XCTest UI \(p. 77\)](#).

XCTEST_UI_TEST_PACKAGE_TEST_EXECUTABLE_VALUE_MIS

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:

```
$ unzip swift-sample-UI.ipa
```

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

```
$ tree .
```

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

```
.
|-- Payload (directory)
    |-- swift-sampleUITests-Runner.app (directory)
        |-- Info.plist
        |-- Plugins (directory)
            |-- swift-sampleUITests.xctest (directory)
                |-- Info.plist
                |-- (any other files)
            |-- (any other files)
```

3. To find the executable value, you can open Info.plist using Xcode or Python.

For Python, you can install the *biplist* module by running the following command:

```
$ pip install biplist
```

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

```
import biplist
info_plist = biplist.readPlist('Payload/swift-sampleUITests-Runner.app/Plugins/swift-
sampleUITests.xctest/Info.plist')
print info_plist['CFBundleExecutable']
```

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

```
swift-sampleUITests
```

For more information, see [XCTest UI \(p. 77\)](#).

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:

```
{
  "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 IAM console at <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**:

```
{
  "Version": "2012-10-17",
  "Statement" : [
    {
      "Effect" : "Allow",
      "Action" : [
        "action-statement"
      ],
      "Resource" : [
        "resource-statement"
      ]
    },
    {
      "Effect" : "Allow",
      "Action" : [
        "action-statement"
      ],
      "Resource" : [
        "resource-statement"
      ]
    }
  ]
}
```

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:

```
{
  "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 \(p. 173\)](#)
- [AWS Device Farm Gradle Plugin \(p. 178\)](#)

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 \(p. 176\)](#).

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 \(p. 176\)](#)
- [Step 2: Creating an AWS Identity and Access Management User for your Jenkins CI Plugin \(p. 177\)](#)

- [Step 3: First-time configuration instructions \(p. 178\)](#)
- [Step 4: Using the Plugin in a Jenkins Job \(p. 178\)](#)
- [Dependencies \(p. 178\)](#)

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

The screenshot shows the Jenkins CI plugin interface for a project named 'Hello World App'. The interface includes a navigation menu on the left with options like 'Back to Dashboard', 'Status', 'Changes', 'Workspace', 'Build Now', 'Delete Project', 'Configure', and 'AWS Device Farm'. The main content area is titled 'Project Hello World App' and features a 'Workspace' link, a 'Recent Changes' link, and a 'Recent AWS Device Farm Results' table. The table lists build status, build numbers, and pass/fail counts. Below the table is a 'Permalinks' section with links to the last build, last failed build, and last unsuccessful build.

Build History

Build Number	Timestamp
#19	Jul 15, 2015 4:25 AM
#18	Jul 15, 2015 1:35 AM
#17	Jul 15, 2015 1:21 AM
#16	Jul 15, 2015 1:06 AM
#15	Jul 14, 2015 10:55 PM

[RSS for all](#) [RSS for failures](#)

Recent AWS Device Farm Results

Status	Build Number	Pass/Fail
Completed	#19	12 ✓
Completed	#18	9 ✓
Completed	#17	12 ✓
Completed	#16	12 ✓
Completed	#15	11 ✓

Permalinks

- [Last build \(#19\), 41 min ago](#)
- [Last failed build \(#19\), 41 min ago](#)
- [Last unsuccessful build \(#19\), 41 min ago](#)

Post-build Actions

Run Tests on AWS Device Farm

Project

[Required] Select your AWS Device Farm project.

Device Pool

[Required] Select your AWS Device Farm device pool.

Application

[Required] Pattern to find newly built application.

Store test results locally.

Choose test to run

- Built-in Fuzz
- Appium Java JUnit
- Appium Java TestNG
- Calabash

Features

[Required] Pattern to find features.zip.

Tags

[Optional] Tags to pass into Calabash.

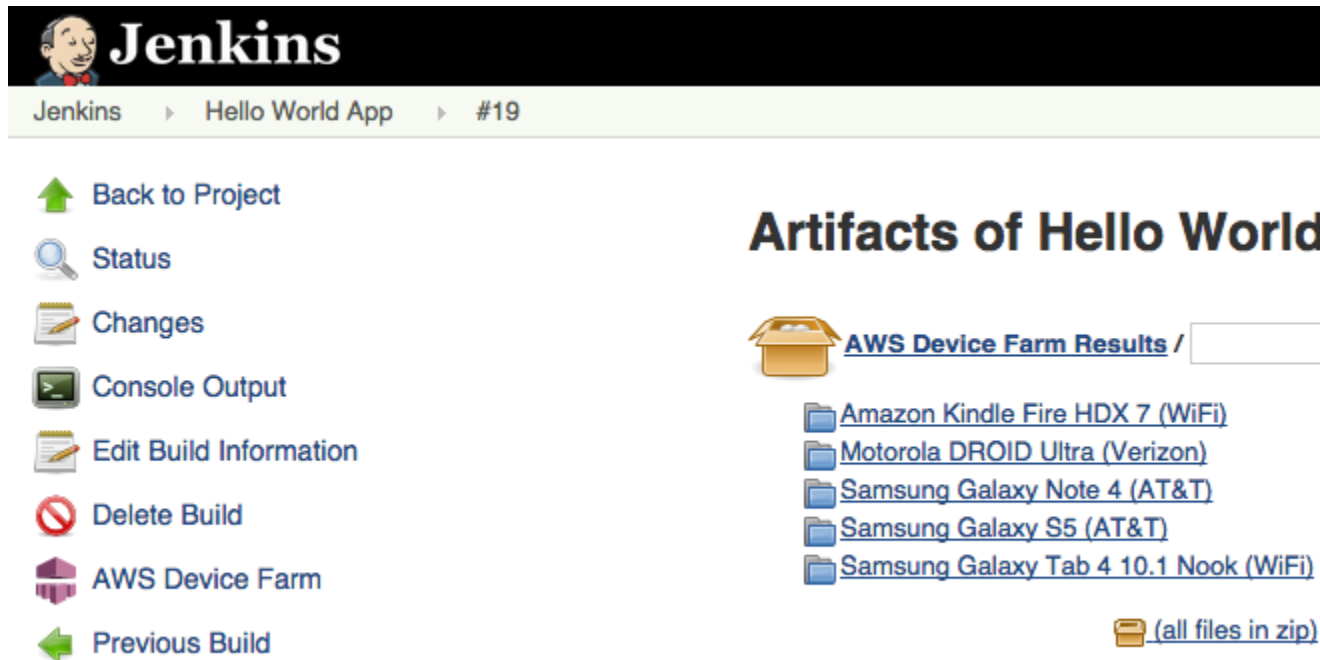
- Instrumentation
- Android UI Automator

Add post-build action ▼

Save

Apply

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



The screenshot shows the Jenkins web interface. At the top, the Jenkins logo is visible. Below it, the breadcrumb navigation shows 'Jenkins > Hello World App > #19'. On the left side, there is a sidebar with several navigation options: 'Back to Project', 'Status', 'Changes', 'Console Output', 'Edit Build Information', 'Delete Build', 'AWS Device Farm', and 'Previous Build'. The main content area is titled 'Artifacts of Hello World'. Below the title, there is a folder icon and the text 'AWS Device Farm Results /'. Underneath, there is a list of folders representing different devices: 'Amazon Kindle Fire HDX 7 (WiFi)', 'Motorola DROID Ultra (Verizon)', 'Samsung Galaxy Note 4 (AT&T)', 'Samsung Galaxy S5 (AT&T)', and 'Samsung Galaxy Tab 4 10.1 Nook (WiFi)'. At the bottom right of the artifacts list, there is a link '(all files in zip)' with a folder icon.

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

1. Download the `hpi` file directly from <http://updates.jenkins-ci.org/latest/aws-device-farm.hpi>.
2. Ensure that the plugin is owned by the `Jenkins` user.
3. Install the plugin using one of the following options:
 - Upload the plugin by choosing **Manage Jenkins, Manage Plugins, Advanced**, and then choose **Upload plugin**.
 - Place the `hpi` file in the Jenkins plugin directory (usually `/var/lib/jenkins/plugins`).
4. Restart Jenkins.

Step 2: Creating an AWS Identity and Access Management User for your Jenkins CI Plugin

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 \(p. 17\)](#), 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 \(p. 177\)](#) 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 \(p. 179\)](#).

The Device Farm Gradle Plugin provides Device Farm functionality from your Android Studio environment. You can kick off tests on real Android phones and tablets hosted by Device Farm.

This section contains a series of procedures to set up and use the Device Farm Gradle Plugin.

Topics

- [Step 1: Building the AWS Device Farm Gradle Plugin \(p. 179\)](#)
- [Step 2: Setting up the AWS Device Farm Gradle Plugin \(p. 179\)](#)
- [Step 3: Generating an IAM User \(p. 180\)](#)
- [Step 4: Configuring Test Types \(p. 181\)](#)
- [Dependencies \(p. 183\)](#)

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 \(p. 179\)](#).

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 \(p. 179\)](#)

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 \(p. 179\)](#).

To configure the AWS Device Farm Gradle Plugin

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

```
buildscript {  
    repositories {  
        mavenLocal()  
        mavenCentral()  
    }  
  
    dependencies {  
        classpath 'com.android.tools.build:gradle:1.3.0'  
        classpath 'com.amazonaws:aws-devicefarm-gradle-plugin:1.0'  
    }  
}
```

2. Configure the plugin in your `build.gradle` file. The following test specific configuration should serve as your guide:

```
apply plugin: 'devicefarm'
```

```
devicefarm {  
  
    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 \(p. 180\)](#)

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**.

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

15. Choose **Apply Policy**.

Next step: [Configuring Test Types \(p. 181\)](#).

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 \(p. 59\)](#) 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 \(p. 45\)](#).

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

Note

If you are using [device slots \(p. 17\)](#), the device slots feature is disabled by default.

Appium

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

- [Appium Java JUnit \(p. 46\)](#)
- [Appium Java TestNG \(p. 50\)](#)

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

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

Built-in: 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:

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

For more information:

- [Built-in: Explorer \(Android\) \(p. 86\)](#)

Built-in: Fuzz

Device Farm provides a built-in fuzz test type, which randomly sends user interface events to devices and then reports the results.

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

For more information, see [Built-in: Fuzz \(Android and iOS\) \(p. 87\)](#).

Calabash

Device Farm provides support for Calabash for Android. To learn how to prepare your Android Calabash tests, see [Calabash \(p. 57\)](#)

```
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 \(p. 59\)](#).

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 \(p. 60\)](#).

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.

- **API version:** 2015-06-23
- **Latest documentation update:** March 21, 2017

Change	Description	Date Changed
Support for Appium 1.6.3	You can now set the Appium version for your Appium custom tests.	March 21, 2017
Set the execution timeout for test runs	You can set the execution timeout for a test run or for all tests in a project. Learn more about Set the Execution Timeout for Test Runs in AWS Device Farm (p. 28) .	February 9, 2017
Network Shaping	You can now simulate network connections and conditions for a test run. Learn more about Simulate Network Connections and Conditions (p. 30) .	December 8, 2016
New Troubleshooting Section	You can now troubleshoot test package uploads using a set of procedures designed to resolve error messages you might encounter in the Device Farm console. Learn more about Troubleshooting (p. 99) .	August 10, 2016
Remote Access Sessions	You can now remotely access and interact with a single device in the console. Learn more about Working with Remote Access (p. 89) .	April 19, 2016
Device Slots Self-Service	You can now purchase device slots using the AWS Management Console, the AWS Command Line Interface, or the API. Learn more about how to Purchase Device Slots (p. 17) .	March 22, 2016
How to stop test runs	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. 33) .	March 22, 2016

Change	Description	Date Changed
New XCTest UI test types	You can now run XCTest UI custom tests on iOS applications. Learn more about the XCTest UI (p. 77) test type.	March 8, 2016
New Appium Python test types	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. 11) .	January 19, 2016
Web Application test types	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. 78) .	November 19, 2015
AWS Device Farm Gradle Plugin	Learn more about how to install and use the Device Farm Gradle Plugin (p. 178) .	September 28, 2015
New Android Built-in Test: Explorer	Learn more about Built-in: Explorer (Android) (p. 86) . The explorer test crawls your app by analyzing each screen as if it were an end user and takes screenshots as it explores.	September 16, 2015
iOS support added	Learn more about testing iOS devices and running iOS tests (including XCTest) in Working with Test Types in AWS Device Farm (p. 45) .	August 4, 2015
Initial public release	This is the initial public release of the <i>AWS Device Farm Developer Guide</i> .	July 13, 2015

AWS Glossary

For the latest AWS terminology, see the [AWS Glossary](#) in the *AWS General Reference*.