SOASTA

TouchTest^{TM} Appcelerator Jenkins CI Tutorial

Table of Contents

Prerequisites

This tutorial guides the user through the process of using the Jenkins continuous integration tool combined with the CloudTest Jenkins/Hudson Plugin in tandem with an example Titanium project and preconfigured CloudTest test compositions.

This tutorial provides guidance for two audiences:

- Users who would like to add Titanium iOS or Android Testing to a pre-existing Jenkins setup
- Users who are either Titanium iOS Developers or Android Developers starting out with TouchTest who would also like to add TouchTest to a continuous integration setup

  TIP: If your organization is not already using Jenkins—refer to the documentation on the Use Jenkins page to get started. Additional Jenkins installation references are included at the end of this guide.

A Jenkins job will be defined that uses git to retrieve the sample project, makes that project touchtestable using the MakeAppTouchTestable utility, and which then deploys the compiled app to multiple devices using a deployment script; after which several pre-existing test compositions are called to run silently in CloudTest and on the specified devices. Finally, CloudTest results are inspected inline in Jenkins, also via the CloudTest Jenkins/Hudson Plugin.

Note: The Appcelerator sample mobile app, KitchenSink, which is available from GitHub, is used as the example project in this guide.

Third Party Prerequisites

To get started, you'll need:

- For iOS, a Macintosh computer with sufficient USB ports to run the desired number of devices, possibly using a USB hub
- For Android, a computer with sufficient USB ports to run the desired number of devices, possibly using a USB hub
- Jenkins continuous integration software (this can be on the same Mac that is running Titanium or on a different Mac node)

This tutorial assumes a minimal familiarity with Appcelerator Titanium Studio and the following prerequisites:

- Titanium Studio is installed with Titanium SDK version 2.1.3 or above.

iOS-Only Prerequisites

The remaining iOS prerequisites are common to any Appcelerator project that utilizes the iOS Developer Program to install an iOS app on a device:

- The developer is enrolled in the iOS Developer Program (so that the Appcelerator app can be pushed to the iOS Device via iTunes).

- The Unique Device Identifier (UDID) of the iOS Device that will be used to test must be registered at the Apple Provisioning Portal.
- iTunes 10.x or greater must be installed on the desktop client that is running Titanium Studio.

One of the key steps during an iOS automated build is deploying the app to your test device, *without requiring any human interaction*. Typical solutions (e.g. over-the-air distribution) require that the user accept a prompt. SOASTA TouchTest includes a tool (built using AppleScript) that *silently* deploys an app. It does this by automating the process of opening the project in Titanium and running it, which causes Xcode to deploy.

To use the deploy script, you will need the following:

1. A dedicated machine running Mac OS X, with Xcode. If you are using Jenkins or Hudson, this can be either the master node or a slave.
2. One or more **tethered** devices. If you have more devices than USB inputs, you can use a USB hub. Note also that sufficient power to prevent the device from running down unexpectedly should be available via that USB input.

   A note on tethering: SOASTA TouchTest™ does **not** require tethering for recording or playback. However, you do need to tether the device for silent deployment of your app.
3. The device(s) should have the iOS “Auto-Lock” setting set to “Never”.

About Shell Steps and the iOS Signing/Provisioning Prerequisite

In general, the best practice for all shell steps presented in this tutorial is to first execute all of them from the command line.

With respect to iOS signing and provisioning, you must perform the following one-time procedure from Terminal to sign each profile/application combination that is in its first use. Provisioning will not be silent until this is done.

TIP: If signing is not done, then a "User interaction not allowed" error will occur in the build. This is because the Operating System requires a user to sign an application using a specific Provisioning profile for the first time. In that case, the Operating System needs a user to authorize the signing process.

Use the following steps:

1. In Terminal, run the xcrun command from your Jenkins job manually.

   Note: If you are using the Jenkins $WORKSPACE variable, you'll need to change it to a Mac OS X path to work here.

   When you do that, OS will popup a dialog box asking permission for the signing process.
2. Choose "Always allow" and this error will not show up again.

Android-Only Prerequisites

- This tutorial uses Apache Ant with Android SDK. Learn more about the Android SDK here, and Apache Ant here.
    - The Minimum Android Version supported for use with TouchTest™ is 2.3.3 (Gingerbread).
    - The Minimum Apache Ant version required for using the SOASTA CloudTest Jenkins Plugin is 1.8.0 or later.
- The Jenkins GitHub Plugin is used to retrieve the project used in this tutorial (instructions for installing it are presented below). Refer to the git site here.
- Each Android device must have the TouchTest Agent app installed and Connected at runtime (if mobile web testing will occur then the TouchTest Web app is also required).

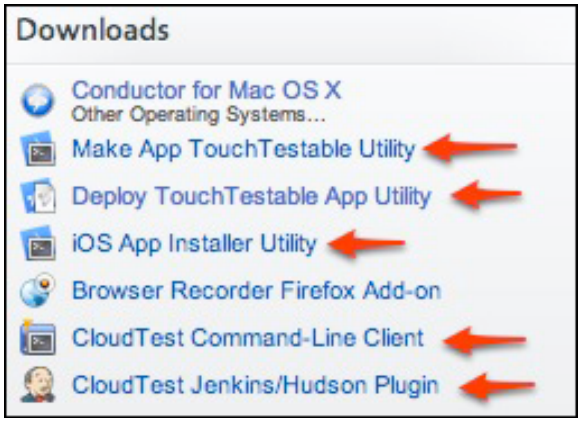
Each Android device should have the following Settings:

- In the device settings, tap Developer Options and check the USB Debugging box.
- In the device settings, tap "Security" and then tap to check the Unknown sources box.

CloudTest Utilities and Plugins

Before proceeding, download the following CloudTest plugin and utility software from the CloudTest Welcome page, Downloads section.

- **CloudTest Jenkins/Hudson Plugin** (this Jenkins/Hudson plugin will be installed using the Jenkins plugin interface). Jenkins CloudTest Plugin version 2.9 or later and TouchTest 51.07 or later are required for dynamic instrumentation.


Downloads
Conductor for Mac OS X
Other Operating Systems...
Make App TouchTestable Utility
Deploy TouchTestable App Utility
iOS App Installer Utility
Browser Recorder Firefox Add-on
CloudTest Command-Line Client
CloudTest Jenkins/Hudson Plugin

- **CloudTest MakeAppTouchTestable Utility** (this utility will be called at the appropriate time via a Jenkins job using an Execute Shell build step)

Note: The CloudTest user specified to run the MakeAppTouchTestable utility must be a user with Mobile Device Administrator rights.

- **CloudTest Command Line Client** (also known as sCommand, this command line interface utility will be called at the appropriate time via a Jenkins job using the CloudTest Jenkins Plugin's MakeAppTouchTestable, Play Composition(s) build step). It is not necessary to download the command-line utility since the Plugin also handles this task.
- For iOS only, the CloudTest **iOS App Installer Utility** is required (this utility contains two executables; the `ios_app_installer`, which is used to install IPA files to iOS physical devices; and the `ios_sim_launcher`, which is used to install compiled APP bundle files). This archive contains two executables:
    - For deployment to Simulators, use the `ios_sim_launcher` executable found in the iOS App Installer Utility at the appropriate time(s) via a Jenkins job using an Execute Shell build step.
    - For deployment to iPhone and iPad devices, use the iOS App Installer Utility to deploy .ipa archives to the physical device(s). This executable can be called at the appropriate time(s) via a Jenkins job using an Execute Shell build step.

  The appropriate executable will be called at the appropriate time(s) via a Jenkins job using an Execute Shell build step.

Test Composition Prerequisites

The test compositions you will use must already exist on the CloudTest instance that you specify and you must use the correct SOASTA Repository path to invoke them. The test composition specifies the device(s) that it will run on.

If you're a new TouchTest user, refer to the following documentation before proceeding with this tutorial.

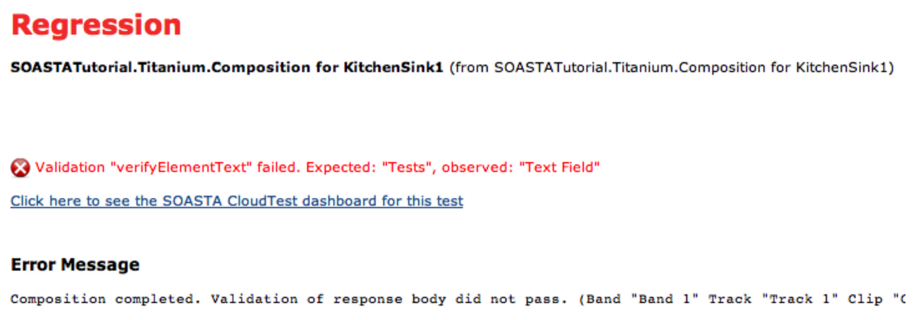
CloudTest Continuous Integration Support

SOASTA CloudTest includes first-class support for including test output in build reports for Jenkins and Hudson via the SCommand utility. Additionally, the Jenkins/Hudson Plugin provides visual integration with CloudTest dashboards within Jenkins itself.

When you run test compositions via SCommand, CloudTest automatically outputs JUnitXML-compatible test output. Since Jenkins provides out-of-the-box JUnit support these result details from a given test composition run in Jenkins using SCommand will appear on the corresponding Test Results page in Jenkins.

By placing all its XML into a directory that we also provide to Jenkins as a part of defining a given job, we can easily display these JUnit-friendly test results inside Jenkins. Using this configuration, a Jenkins Test Result detail page will display details from CloudTest.

While it is not necessary to install the CloudTest Jenkins/Hudson Plugin to utilize CloudTest's JUnitXML-formatted output in Jenkins in this manner, plugin installation adds the capability to also jump to specific test composition errors in the CloudTest Result Details dashboard from within Jenkins.


Regression

SOASTATutorial.Titanium.Composition for KitchenSink1 (from SOASTATutorial.Titanium.Composition for KitchenSink1)

Validation "verifyElementText" failed. Expected: "Tests", observed: "Text Field"

Click here to see the SOASTA CloudTest dashboard for this test

Error Message

```
Composition completed. Validation of response body did not pass. (Band "Band 1" Track "Track 1" Clip "C
```

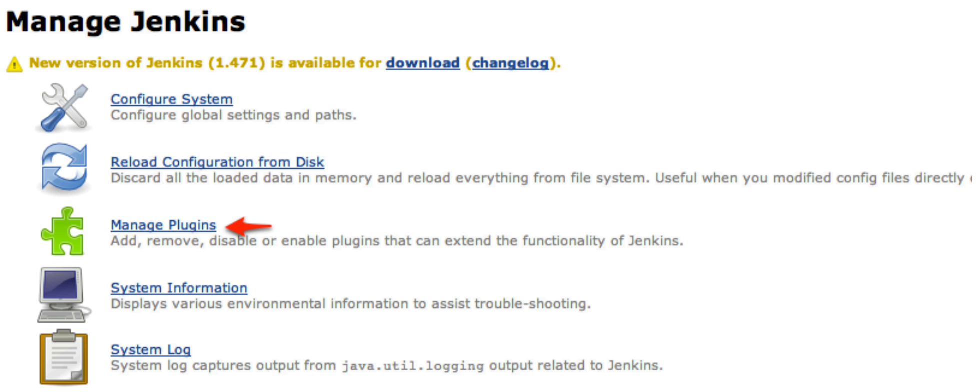
In the screenshot above, the error heading and detail text are an output of SCommand for the test shown.

Clicking the link, and providing CloudTest credentials, will then display the precise failure inline in Jenkins for the given CloudTest result.

Installing the SOASTA CloudTest Jenkins/Hudson Plugin

Use the following steps to install the CloudTest Jenkins Plugin (version 2.9 or later are required for dynamic instrumentation as well as to automatically update the CloudTest Jenkins Plugin's MakeAppTouchTestable) from within your Jenkins instance. The MATT module is auto-updated, so once it's installed the most current version is assured.

1. In Jenkins, click Manage Jenkins, and then click Manage Plugins.



Manage Jenkins

⚠ New version of Jenkins (1.471) is available for download (changelog).

Configure System
Configure global settings and paths.

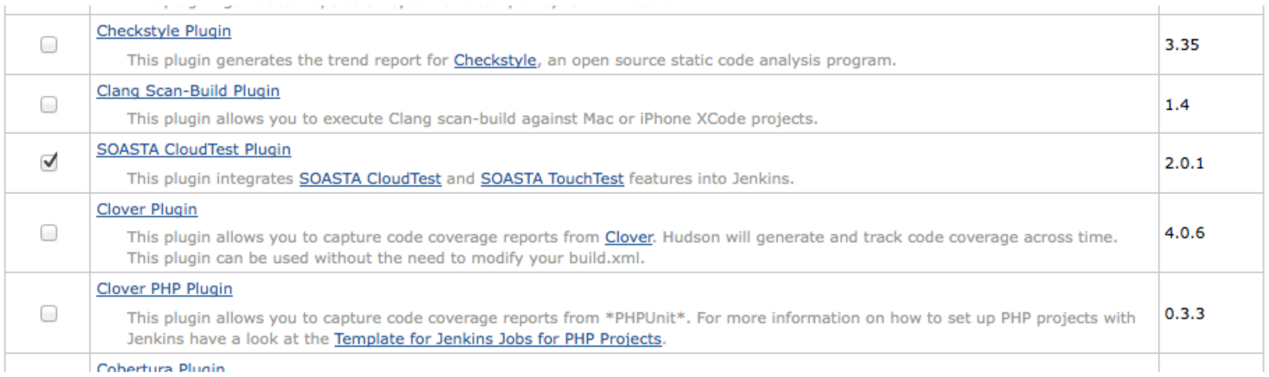
Reload Configuration from Disk
Discard all the loaded data in memory and reload everything from file system. Useful when you modified config files directly

Manage Plugins
Add, remove, disable or enable plugins that can extend the functionality of Jenkins.

System Information
Displays various environmental information to assist trouble-shooting.

System Log
System log captures output from `java.util.logging` output related to Jenkins.

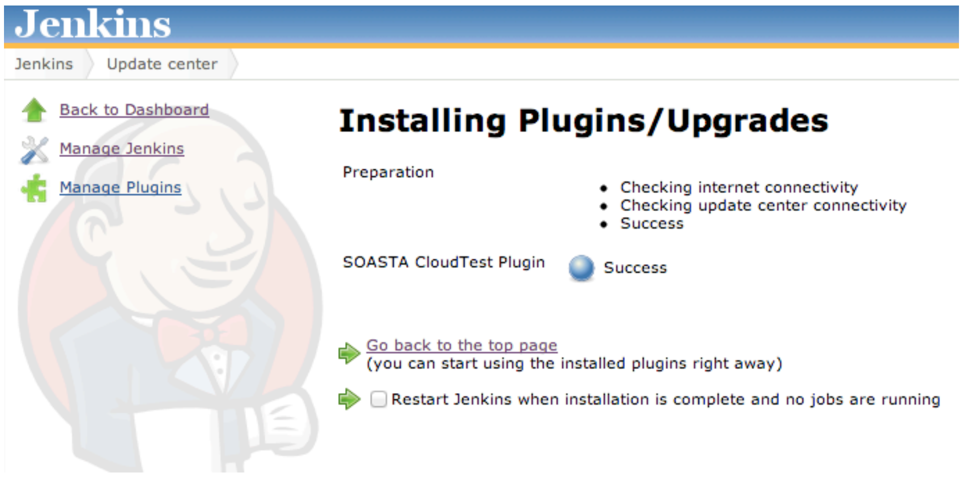
2. Click the Plugin Manager, Advanced tab.
3. Locate the SOASTA CloudTest Plugin in the list (using Cmd+F and "Soasta" is a quick way to locate it).



☐	Checkstyle Plugin This plugin generates the trend report for Checkstyle, an open source static code analysis program.	3.35
☐	Clang Scan-Build Plugin This plugin allows you to execute Clang scan-build against Mac or iPhone XCode projects.	1.4
☑	SOASTA CloudTest Plugin This plugin integrates SOASTA CloudTest and SOASTA TouchTest features into Jenkins.	2.0.1
☐	Clover Plugin This plugin allows you to capture code coverage reports from Clover. Hudson will generate and track code coverage across time. This plugin can be used without the need to modify your build.xml.	4.0.6
☐	Clover PHP Plugin This plugin allows you to capture code coverage reports from *PHPUnit*. For more information on how to set up PHP projects with Jenkins have a look at the Template for Jenkins Jobs for PHP Projects.	0.3.3

4. Click the Install without Restart button at the bottom of the page.

The Installing Plugins/Upgrades page appears and indicates success once the install completes.



Installing Plugins/Upgrades

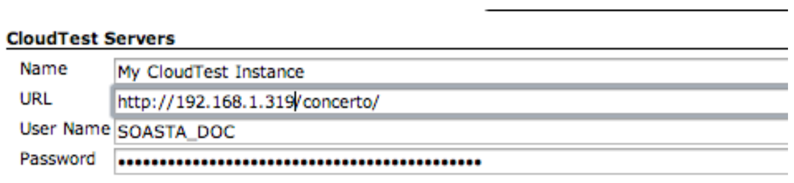
Preparation
- Checking internet connectivity
- Checking update center connectivity
- Success

SOASTA CloudTest Plugin ● Success

Go back to the top page
(you can start using the installed plugins right away)

☐ Restart Jenkins when installation is complete and no jobs are running

Before using the CloudTest Jenkins Plugin, you will need to provide the CloudTest server URL and user credentials via the Manage Jenkins > Configure System page, CloudTest section. *We recommend creating a dedicated CloudTest account for Jenkins to use.*



CloudTest Servers

Name	My CloudTest Instance
URL	http://192.168.1.319/concerto/
User Name	SOASTA_DOC
Password	•••

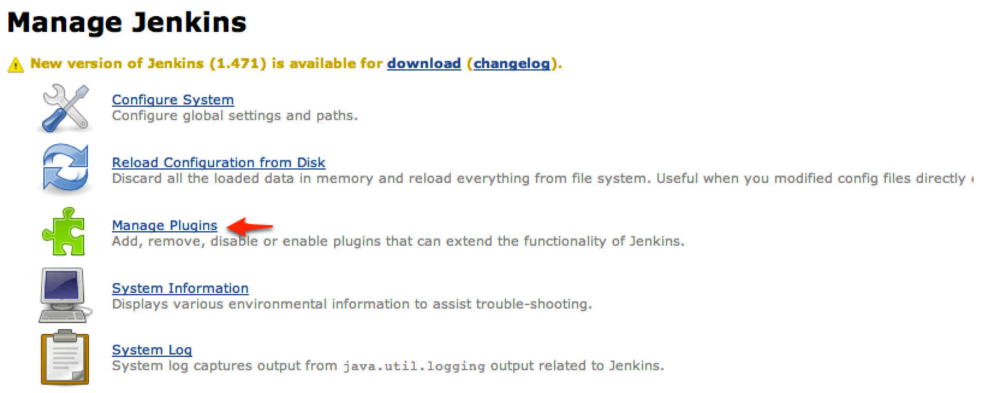
Installing the GitHub Plugin

Use the following steps to install the GitHub Plugin from within your Jenkins instance. This will allow us to use GitHub as a Source Code Repository to retrieve the example project, KitchenSink.

TIP: Git is a distributed version control system used for software development. It is not necessary to signup or login to GitHub in order to checkout the code using the following command. In the Source Code Management section, click Git.

If you are using your own app, you can skip this requirement and then substitute the SCM tool and repository to use in the place of Git.

1. In Jenkins, click Manage Jenkins, and then click Manage Plugins.



Manage Jenkins

New version of Jenkins (1.471) is available for download (changelog).

Configure System
Configure global settings and paths.

Reload Configuration from Disk
Discard all the loaded data in memory and reload everything from file system. Useful when you modified config files directly

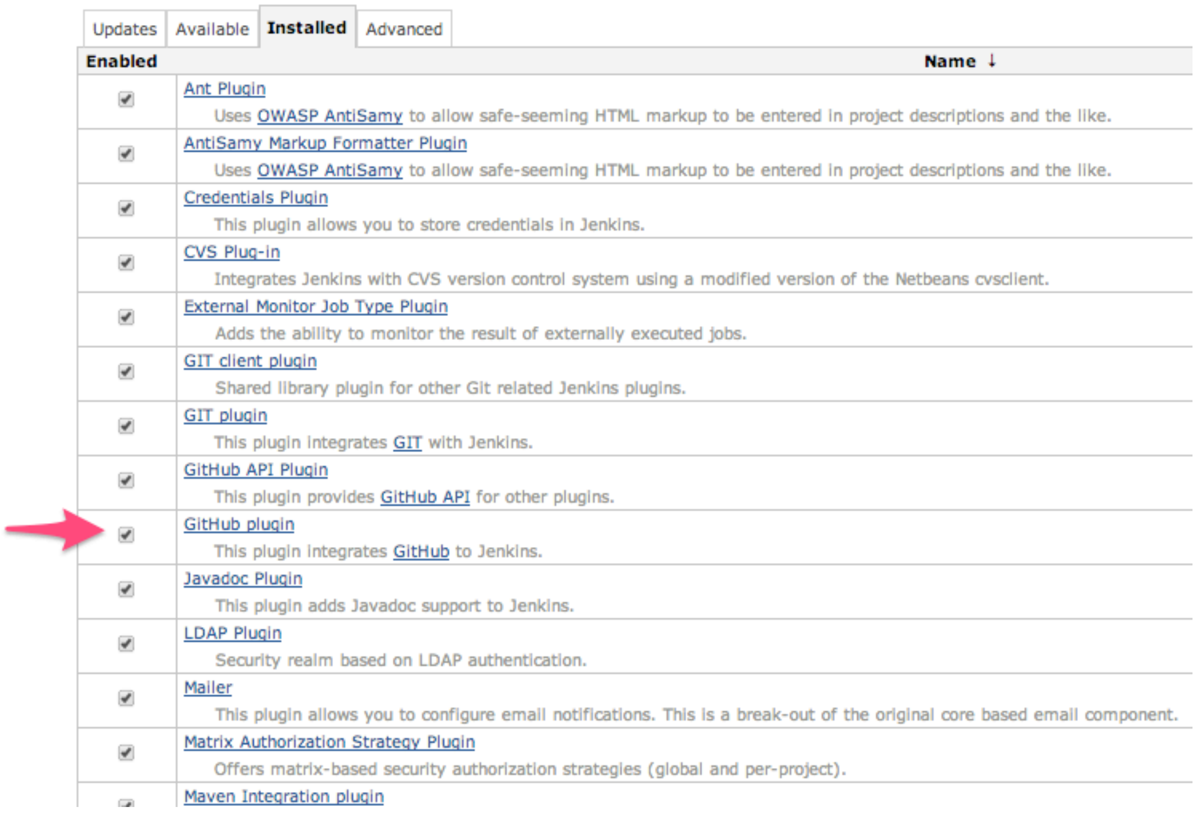
Manage Plugins
Add, remove, disable or enable plugins that can extend the functionality of Jenkins.

System Information
Displays various environmental information to assist trouble-shooting.

System Log
System log captures output from `java.util.logging` output related to Jenkins.

2. Click the Plugin Manager, Advanced tab
3. Locate the GitHub Plugin and check it as well. Installing the GitHub Plugin will also install the GitHub Plugin.

You can verify plugin installation on the Manage Plugins, Installed tab:



Enabled	Name ↓
✓	Ant Plugin Uses OWASP AntiSamy to allow safe-seeming HTML markup to be entered in project descriptions and the like.
✓	AntiSamy Markup Formatter Plugin Uses OWASP AntiSamy to allow safe-seeming HTML markup to be entered in project descriptions and the like.
✓	Credentials Plugin This plugin allows you to store credentials in Jenkins.
✓	CVS Plug-in Integrates Jenkins with CVS version control system using a modified version of the Netbeans cvsclient.
✓	External Monitor Job Type Plugin Adds the ability to monitor the result of externally executed jobs.
✓	GIT client plugin Shared library plugin for other Git related Jenkins plugins.
✓	GIT plugin This plugin integrates GIT with Jenkins.
✓	GitHub API Plugin This plugin provides GitHub API for other plugins.
✓	GitHub plugin This plugin integrates GitHub to Jenkins.
✓	Javadoc Plugin This plugin adds Javadoc support to Jenkins.
✓	LDAP Plugin Security realm based on LDAP authentication.
✓	Mailer This plugin allows you to configure email notifications. This is a break-out of the original core based email component.
✓	Matrix Authorization Strategy Plugin Offers matrix-based security authorization strategies (global and per-project).
✓	Maven Integration plugin

Note: We will specify a Git repository to clone in a later step in the Jenkins job.

Static vs. Dynamic Instrumentation

The CloudTest Jenkins Plugin's MATT module supports two instrumentation methods: static and dynamic. The MATT module plays a role in making either the iOS or Android project or its compiled APP bundle, IPA, or APK TouchTestable.

- *Dynamic instrumentation* occurs when MATT instruments a compiled file (i.e. an APP bundle folder, an IPA file, or APK file).

  This method requires that you compile your project first to create an APP, IPA, or APK, after which it can be instrumented using SOASTA 51.07 or later (TouchTest 7040.58). Dynamic instrumentation is available for all supported Android versions, while for iOS version 6 or later is required.
- *Static instrumentation* occurs when MATT instruments an iOS Android project. This method requires that you apply MATT to the project prior to building the APK. Static instrumentation is available in all TouchTest releases and for all supported Android versions.

1. Determine whether to instrument the mobile app using the MATT input type project or to instrument it using the APP, IPA, or APK options. The subsequent steps will differ since MATT is applied at a different time in the workflow.

Jenkins Workflows for TouchTest

The CloudTest Jenkins Plugin is used to make the mobile app TouchTestable—either by applying it to the project or to the APP bundle, IPA, or APK file itself (as discussed below). OS-specific tools are used where applicable.

- *In Android*, Ant is used to build the APK and the command used depends whether MATT has already been applied. After which, adb is used in an Execute Shell step, followed by a final pre-build step that uses the CloudTest Jenkins Plugin, Play Composition(s) command.
- *In iOS*, xcodebuild and xcrun are used, and the iOSAppInstaller utility command-line commands are used to build and deploy.

Because there are different possible Jenkins job workflows, the *possible* steps are presented *a la carte.* For each Jenkins job there will be (minimally):

- A First step; used to retrieve the source project (all workflows)
- In between the first and last steps, each Jenkins job will have:
  - A CloudTest Plugin, MakeAppTouchTestable step;
  - with one (or more) Build step using Ant (in Execute Shell)
  - and one (or more) Install steps using adb (in Execute Shell)
- A Last step; to Play the Composition

Note: Signing the app is done using the CloudTest Jenkins Plugin, MATT, Advanced Options to enter the optional MATT parameters (refer to the relevant sections). Signing can also be done from the command line using an Execute Shell step and the tool of your choice.

Note the following terminology:

- *Static instrumentation* applies MATT to the iOS project file or to te Android project folder
- Dynamic instrumentation applies MATT to the iOS APP bundle file, IPA file, or to the Android APK file

Dynamic Instrumentation of an APP file

In this workflow, you'll apply MATT to the compiled APP file using the `appbundle` parameter. If you're also deploying to physical devices, you'll need to mix and match steps to do both, keeping in mind to build the APP first.

- ○ *Build the APP* with an Execute Shell step using xcodebuild
- ○ *Apply MATT* to that compiled APP using `iOS APP Bundle`
- ○ *Run App on iOS Simulator* command

Dynamic Instrumentation of an IPA file

In this workflow, you'll delay applying MATT until the IPA is created

- ○ *Build the APP* with an Execute Shell step using xcodebuild
- ○ *Build the IPA* file with an Execute Shell step using xcrun,
- ○ *Apply MATT* to that compiled IPA using `ipa`
- ○ Install App on iOS Device command.

In the remainder of the Job Creation steps, mix and match the tasks that you need to build your Jenkins job.

Static Instrumentation of an Xcode Project

In this workflow, you'll apply MATT to the Xcode project itself, using the `project` parameter. After which, you'll add the steps necessary to build and install to your simulators and devices.

- ○ *Apply MATT* using the `project` parameter (static only)
- ○ *Build the APP* with an Execute Shell step using xcodebuild
- ○ (optional) *Build the IPA* with an Execute Shell step using xcrun
- ○ Either *Run App in iOS Simulator* or *Install iOS App on Device* (using IPA)

Dynamic Instrumentation of an APK file

In this workflow, you'll apply MATT to the compiled APK file using the appbundle parameter. If you're also deploying to physical devices, you'll need to mix and match steps to do both, keeping in mind to build the APK first.

- Get the source code (in this scenario, using git)
- *Build the APK* with an Execute Shell step using Ant
- *Apply MATT* to that compiled APK using MATT's APK input type
- *Install the APK*
- *Run the composition(s)*
- *Post results (optional)*

In the following sections, choose only those steps that match your workflow.

Static Instrumentation of an Android Project

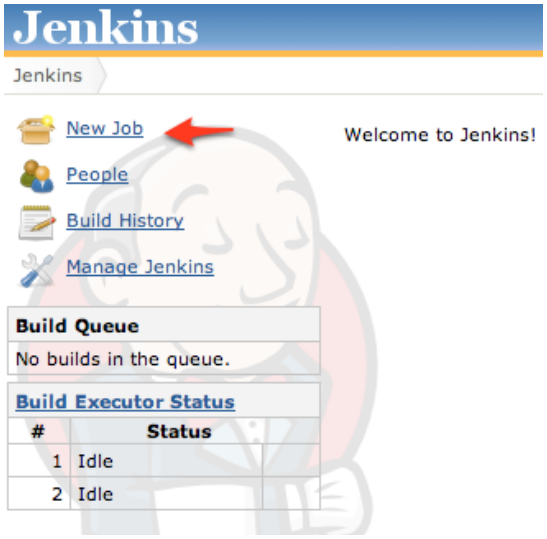
In this workflow, you'll apply MATT to the Android project itself, using the project input type. This is done before the APK build step. After which, add the step(s) necessary to build and install to your simulators and devices.

- Get the source code (in this scenario, using git)
- *Apply MATT* using the project input type (static only)
- *Build the APK* with an Execute Shell step using Ant
- Install the APK
- Run the composition(s)
- Post results (optional)

Creating a Jenkins Job

The Jenkins job created below will run on the desktop machine with Titanium and one or more properly provisioned USB-attached devices. The job will have build steps using Execute Shell that will utilize git to download a project from source, run the Make AppTouchTestable utility on that project, build and deploy using the deploy script, and use the Plugin's Play Composition(s) command to silently play a list of compositions. Additionally, a Post-build action is used to publish the JUnit test result report.

1. In top-level Jenkins dashboard, click New Job.

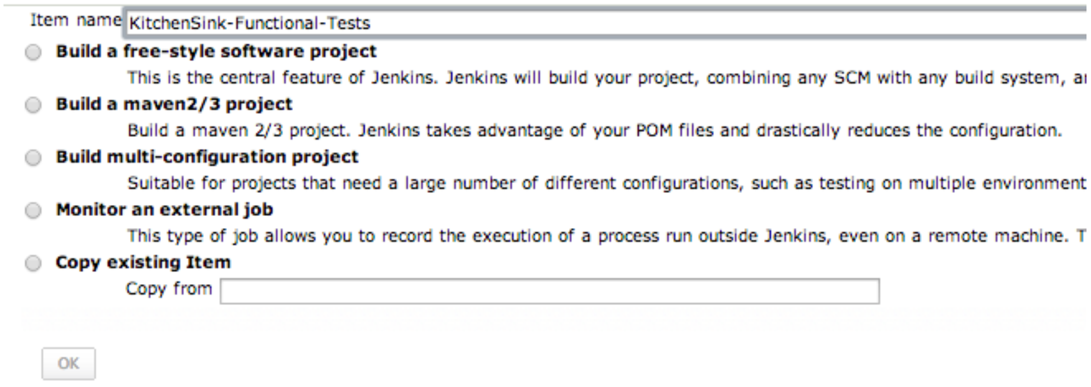


Build Queue
No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

The Job name page appears.

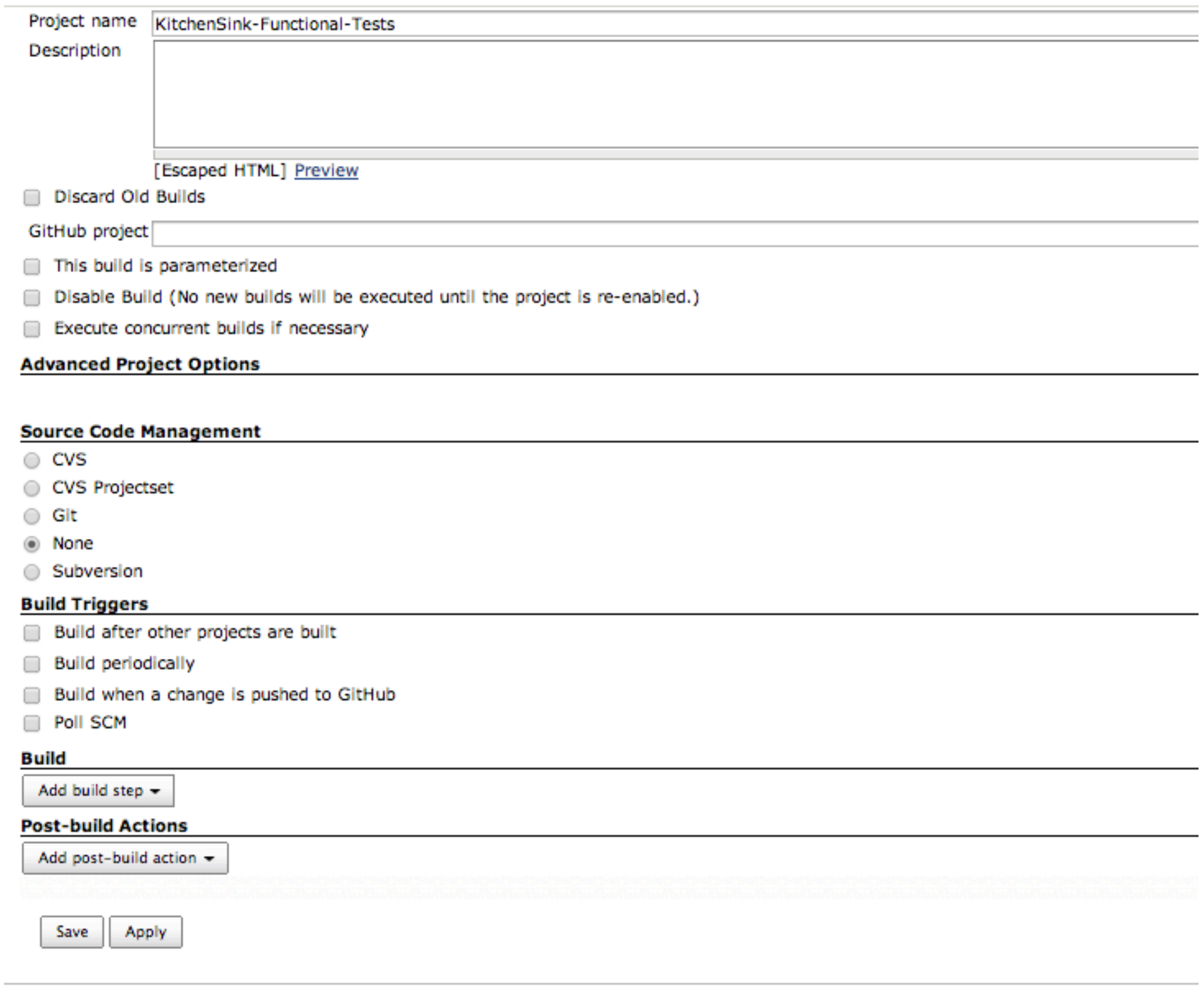


Item name KitchenSink-Functional-Tests

- **Build a free-style software project**
  This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, a
- **Build a maven2/3 project**
  Build a maven 2/3 project. Jenkins takes advantage of your POM files and drastically reduces the configuration.
- **Build multi-configuration project**
  Suitable for projects that need a large number of different configurations, such as testing on multiple environment
- **Monitor an external job**
  This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. T
- **Copy existing Item**
  Copy from

1. Enter a job name without spaces and select the first option, "Build a free-style software project".

The Job Details page appears.



Project name: KitchenSink-Functional-Tests

Description:

[Escaped HTML] Preview

☐ Discard Old Builds

GitHub project:

☐ This build is parameterized

☐ Disable Build (No new builds will be executed until the project is re-enabled.)

☐ Execute concurrent builds if necessary

Advanced Project Options

Source Code Management

○ CVS
○ CVS Projectset
○ Git
● None
○ Subversion

Build Triggers

☐ Build after other projects are built

☐ Build periodically

☐ Build when a change is pushed to GitHub

☐ Poll SCM

Build

Post-build Actions

2. Enter the following configuration details for this job:

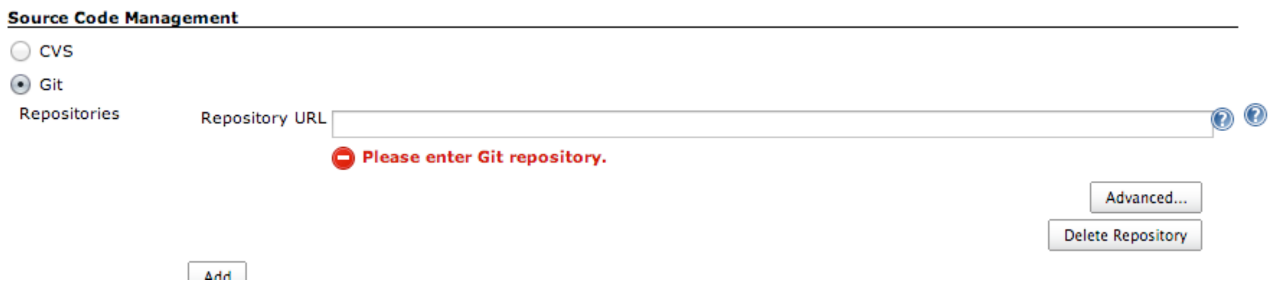
- A description. For example, “For example, "Checks out the KitchenSink source code, makes the project or compiled file TouchTestable, deploys the app, and runs a suite of CloudTest compositions."
- Click the Add build step drop-down and select Execute shell.

Get KitchenSink using the GitHub Plugin (All Workflows)

Next, we will add a step that will get the KitchenSink project that will be used in the remainder of this tutorial. The following instructions use the Jenkins Git Plugin (installed above). If you are using your own Source Code Management system simply select its type and enter its repository URL, as you would normally do.

With the Git Plugin installed in our Jenkins instance, we will add the Source Code Management step as we would with any SCM tool.

1. In the Source Code Management section, check the Git radio button.



Source Code Management

○ CVS
[● Git]

Repositories — Repository URL:

Please enter Git repository.

Note: You can specify your own SCM tool and Repository URL here.

2. Enter the GitHub Repository URL in the entry field:

```
https://github.com/appcelerator/KitchenSink
```

3. Save the Jenkins job and proceed with the OS-specific instructions below.

Using the CloudTest Jenkins Plugin, MakeAppTouchTestable

As noted in the prerequisites above, the CloudTest Jenkins Plugin's MakeAppTouchTestable module is used to automate portions of the Jenkins job.

Note: The CloudTest user specified to run the CloudTest Jenkins Plugin, MakeAppTouchTestable module must be a user with Mobile Device Administrator rights. CloudTest Lite users have admin rights for the given device on their own instance.

In the following sections, select only those steps necessary to complete your Jenkins job. The workflows are organized first by mobile OS (iOS or Android) and then by the instrumentation type.

About the Titanium SDK Path

In some cases where the Titanium SDK path is not automatically detected, you will need to add your Titanium SDK path to the examples included below. In such cases, use the `-titaniumsdk` parameter.

The Titanium SDK argument can be appended to the CloudTest Jenkins Plugin's MATT step using the following syntax:

```
-titaniumsdk <Path of the Titanium SDK to use>
```

where the value of the Titanium SDK is either /Library/Application Support/ Titanium/ mobilesdk/osx/ or ~/Library/Application Support/Titanium/mobilesdk/osx/ with the flavor or SDK appended at the end of the path.

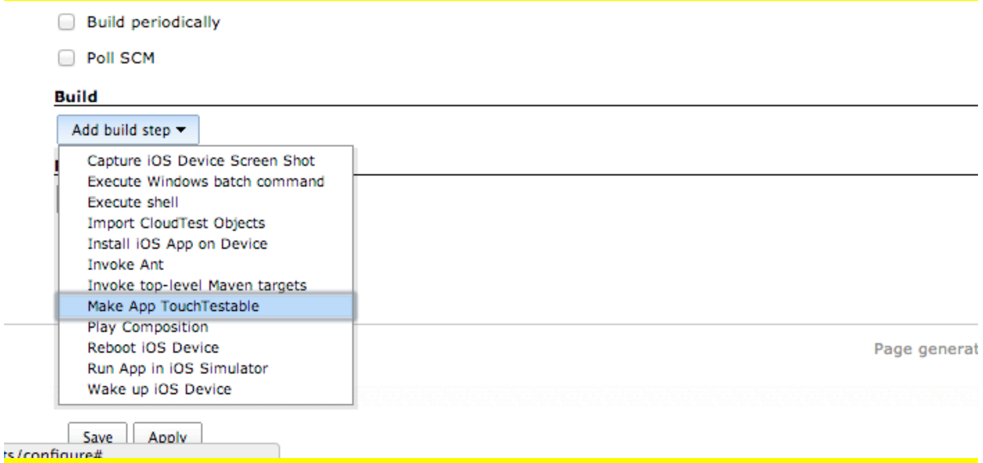
For example,

“/Library/Application Support/Titanium/mobilesdk/osx/2.0.1.GA2”.

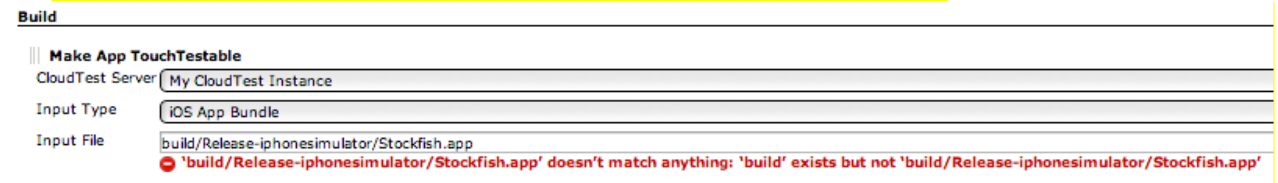
Using MATT on an iOS APP Bundle (Dynamic Instrumentation)

Use the following steps to dynamically instrument an APP bundle. This is typically done before the APP is run on a simulator but after the project APP is built. For example, you can use this step after an Execute Shell step using xcodebuild.

1. Add a MakeAppTouchTestable step to the job.



2. Apply MATT to the compiled APP by selecting the Input Type, iOS App Bundle (the MATT command-line equivalent is the `appbundle` parameter).



CloudTest Server: My CloudTest Instance
Input Type: iOS App Bundle
Input File: build/Release-iphonesimulator/Stockfish.app

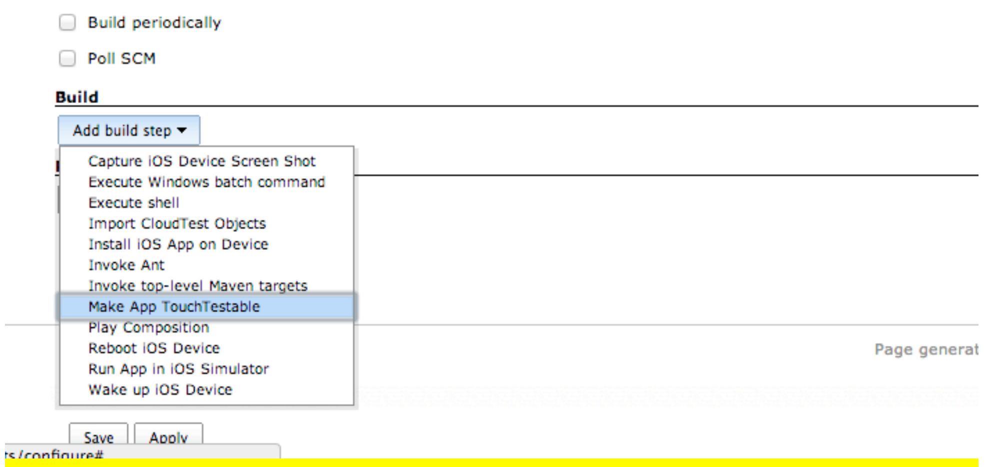
'build/Release-iphonesimulator/Stockfish.app' doesn't match anything: 'build' exists but not 'build/Release-iphonesimulator/Stockfish.app'

1. Enter the APP name as the Input File (from the workspace root).
2. Optionally, click Advanced to display additional MATT configuration fields.
3. Save the Jenkins job.

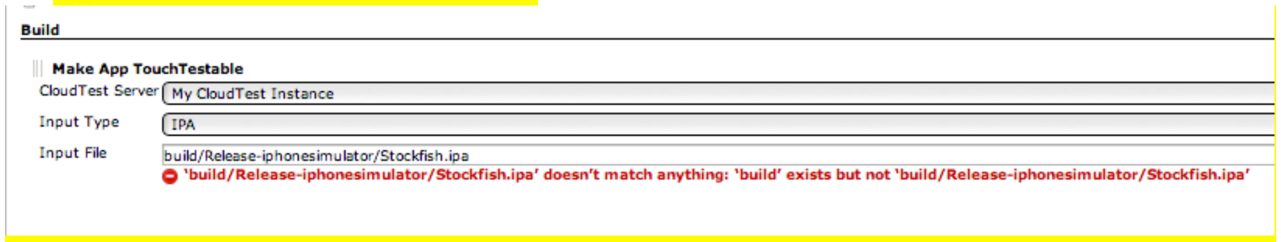
Using MATT on an IPA (Dynamic Instrumentation for Device)

Use the following steps to dynamically instrument an IPA file. This is typically done as the last step before it is installed on a device. For example, you can use this step after an Execute Shell step using xcrun.

3. Add a MakeAppTouchTestable step to the job.



4. Apply MATT to the compiled IPA by selecting the Input Type, IPA (e.g. the MATT equivalent is the `ipa` parameter).



Build

Make App TouchTestable

CloudTest Server: My CloudTest Instance

Input Type: IPA

Input File: build/Release-iphonesimulator/Stockfish.ipa

'build/Release-iphonesimulator/Stockfish.ipa' doesn't match anything: 'build' exists but not 'build/Release-iphonesimulator/Stockfish.ipa'

4. Enter the IPA name as the Input File (from the workspace root).

5. Optionally, click Advanced to display additional MATT configuration fields.

- **Launch URL** – Same as MATT `launchurl`. For example: `my-app://launch`
- **Back up modified files** – Check this to keep backups in the project).
- **Additional options** - Enter any additional MATT command line parameters.

  Most notably, you can use MATT to provision and code sign the dynamically instrumented IPA file (code signing and provisioning can, of course, be done using xcrun, which is discussed in the Execute Shell step, Building the IPA for a Device).

  Use the following MATT optional IPA parameters

  - `-provisioningprofile <profilepath>` – Path of the Provisioning profile to be used for building IPA file. The provisioning profile you input MUST to be a Distribution profile.
  - `-signingidentity <signingidentityname>` – Name of the signing identity to be used for codesigning the application. (e.g. "iOS Distribution: Developer Name")
  - `-entitlementsfile <entitlementsfilepath>` - path of the entitlements file to be used for codesigning the application

  For more about using additional MATT parameters, use:

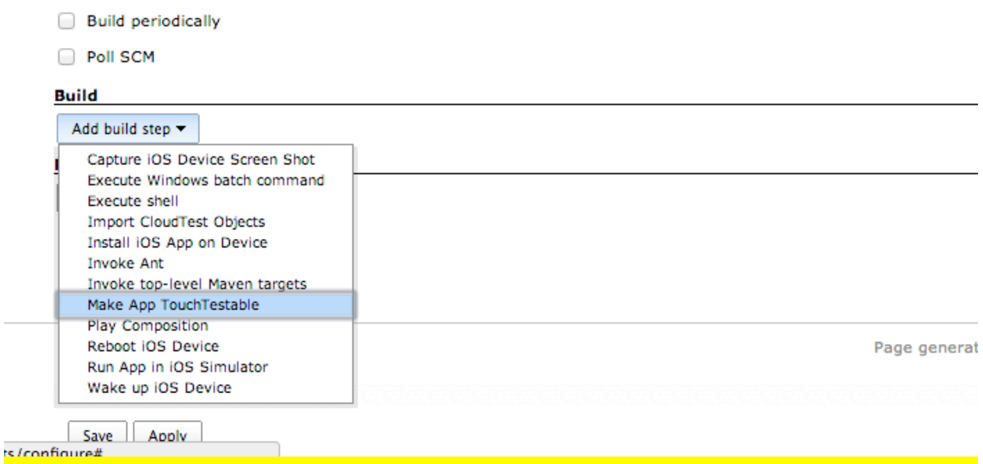
```
sh MakeAppTouchTestable/bin/MakeAppTouchTestable -help
```

6. Save the Jenkins job.

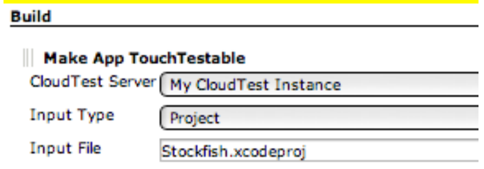
Using MATT on a Project (Static Instrumentation for Simulators/ Devices)

This utility is automatically downloaded by the SOASTA CloudTest Jenkins Plugin and can be easily specified in your Jenkins job using the following steps.

7. Click Add a Build Step, and select MakeAppTouchTestable from the drop-down list. This MakeAppTouchTestable step will always be the first step in a Static Instrumentation workflow, but will come after the APP or IPA is built where those files are in use on a simulator(s) or device(s).



The Make App TouchTestable form appears.



Build

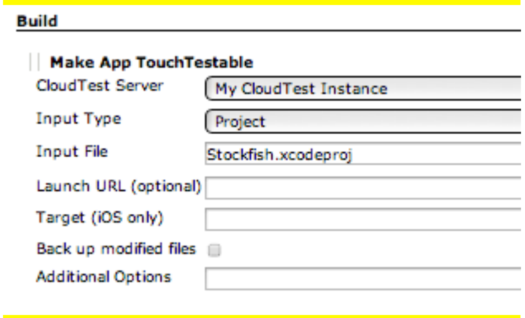
Make App TouchTestable

CloudTest Server	My CloudTest Instance
Input Type	Project
Input File	Stockfish.xcodeproj

8. Enter the Xcode project name as the Input File.

9. Optionally, click Advanced to display additional MATT configuration fields.

- **Launch URL** – Same as MATT `launchurl`. For example: `my-app://launch`
- **Target** – Same as MATT `target`. For example: `Stockfish copy`



Build

Make App TouchTestable

CloudTest Server	My CloudTest Instance
Input Type	Project
Input File	Stockfish.xcodeproj
Launch URL (optional)	
Target (iOS only)	
Back up modified files	☐
Additional Options	

- **Back up modified files** – Check this to keep backups in the project folder (where .xcodeproj resides).
- **Additional options** - Enter any additional MATT command line parameters. For more about using additional MATT parameters, use:

```
sh MakeAppTouchTestable/bin/MakeAppTouchTestable -help
```

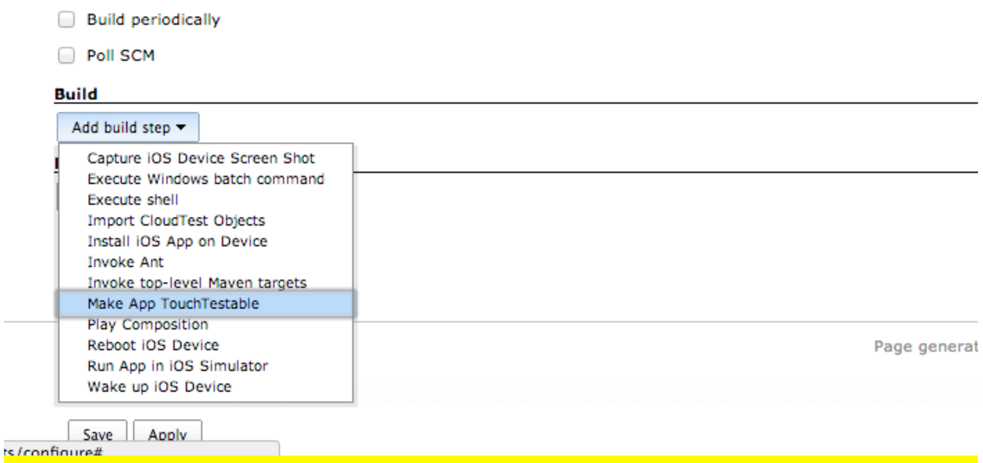
10. Save the Jenkins job.

Using MATT on an APK (Dynamic)

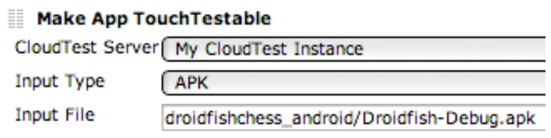
Use the following steps to dynamically instrument an APK file at the proper point in your workflow.

- If you are using dynamic instrumentation, you must first do the APK step prior to this step. But, if you'd rather follow along sequentially, you can add the MATT step now so long as you place an APK build step prior to it before building the Jenkins job.
- If you are using static instrumentation, you must apply MATT to the project prior to this step.

1. Add a MakeAppTouchTestable step to the job.



2. Select the Input Type, APK (e.g. the MATT equivalent is the APK parameter).

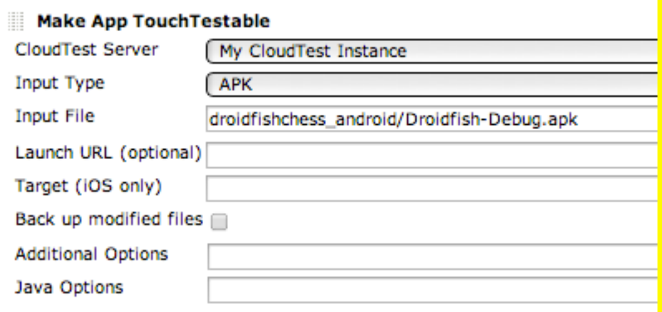


Make App TouchTestable

CloudTest Server	My CloudTest Instance
Input Type	APK
Input File	droidfishchess_android/Droidfish-Debug.apk

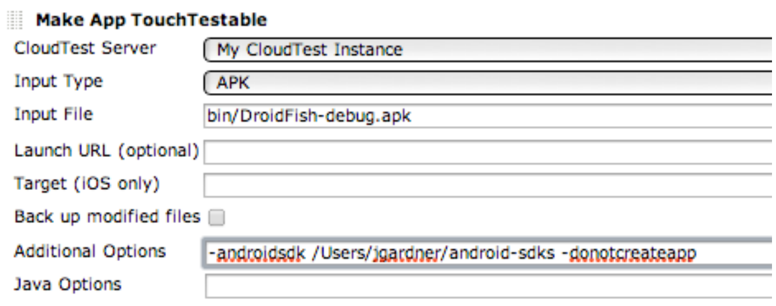
3. Enter the APK name as the Input File (from the workspace root).

4. Optionally, click Advanced to display additional MATT configuration fields.



Make App TouchTestable	
CloudTest Server	My CloudTest Instance
Input Type	APK
Input File	droidfishchess_android/Droidfish-Debug.apk
Launch URL (optional)	
Target (iOS only)	
Back up modified files	☐
Additional Options	
Java Options	

5. Specify MATT flags as required. For example, `androidsdk` when installing a debug APK to a physical device and either `overwriteapp` or `donotcreateapp` to prevent Jenkins from marking the MATT step as `FAILURE` (even though the TouchTestable app is created as expected).



Make App TouchTestable	
CloudTest Server	My CloudTest Instance
Input Type	APK
Input File	bin/DroidFish-debug.apk
Launch URL (optional)	
Target (iOS only)	
Back up modified files	☐
Additional Options	-androidsdk /Users/jgardner/android-sdks -donotcreateapp
Java Options	

- **Launch URL** – Same as MATT launchurl. For example: my-app://launch
- **Backup modified files** – Check this to keep backups in the project.
- **Additional options** - Enter any additional MATT command line parameters.

  Most notably, you can use MATT to add keystore, keypass, and storepass arguments to sign the dynamically instrumented APK file.

  Use the following MATT optional APK parameters

  - -keystore <keystorepath> - Path of the keystore to be used to sign the APK file..
  - -storepass <keystorepassword> - Password of the keystore to be used to sign the APK file.
  - -keypass <privatekeypassword> - Password of the private key (if different than the keystore password) to be used to sign the APK file.

  For more about using additional MATT parameters, use:

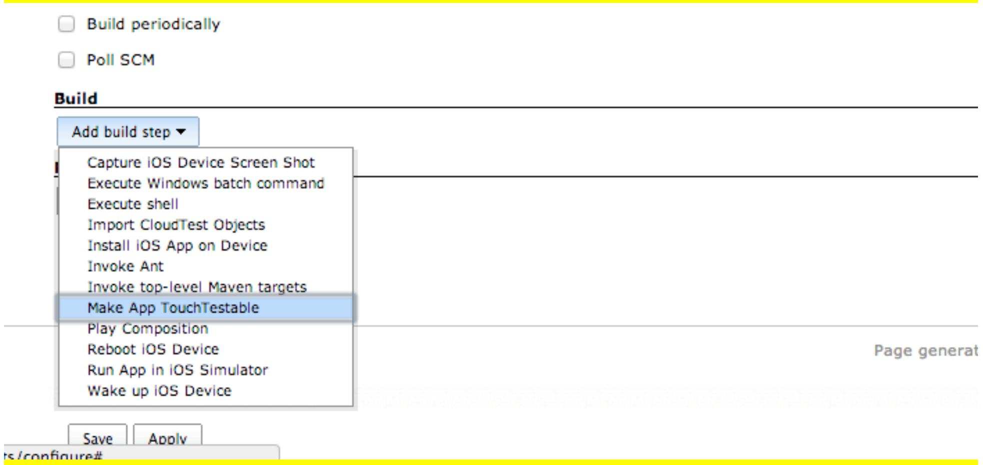
  sh MakeAppTouchTestable/bin/MakeAppTouchTestable - help

6. Save the Jenkins job.

Using MATT on a Project (Static)

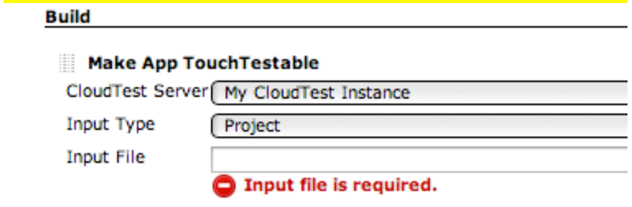
Use the following steps if you'll be instrumenting the project using the static method via the Input Type, project. This step will always proceed the build APK step while using static instrumentation.

1. Click Add a Build Step, and select Make App TouchTestable from the drop-down list.



The Make App TouchTestable form appears.

TIP: Click the Help icons for any row to get tip text.

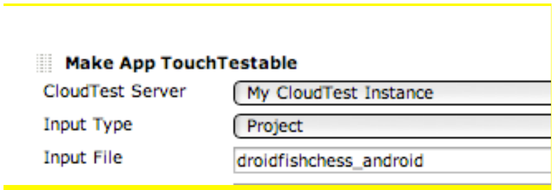


CloudTest Server: My CloudTest Instance
Input Type: Project
Input File:
Input file is required.

2. Select the CloudTest Server from among those configured.

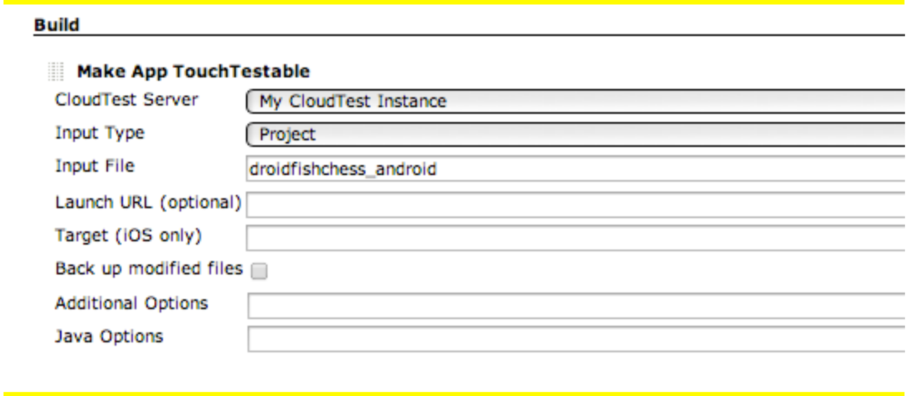
   Note: Indicating the CloudTest server was done as part of the CloudTest Jenkins Plugin, Configure System step. If not entries appear here return to Manage Jenkins > Configure System, and fill in the information in the CloudTest Servers section. Be sure to save these changes.

3. Specify the current project folder to use. "droidfishchess_android" to indicate the project folder in the Jenkins workspace.


Make App TouchTestable

CloudTest Server	My CloudTest Instance
Input Type	Project
Input File	droidfishchess_android

4. Optionally, specify additional parameters by first clicking the Advanced button (page right).


Build

Make App TouchTestable

CloudTest Server	My CloudTest Instance
Input Type	Project
Input File	droidfishchess_android
Launch URL (optional)	
Target (iOS only)	
Back up modified files	☐
Additional Options	
Java Options	

5. Select the CloudTest Server from among those configured.

   TIP: Indicating the CloudTest server was done as part of the CloudTest Jenkins Plugin, Configure System step. If not entries appear here return to Manage Jenkins > Configure System, and fill in the information in the CloudTest Servers section. Be sure to save these changes.

6. Optionally, click Advanced to display additional MATT configuration fields.

- **Launch URL** – Same as MATT launchurl. For example: my-app://launch
- **Back up modified files** – Check this to keep backups in the project folder (where build.xml resides).
- **Additional options** - Enter any additional MATT command line parameters.

  Most notably, you can use MATT to add keystore, keypass, and storepass arguments to sign the dynamically instrumented APK file.

  Use the following MATT optional APK parameters

  - -keystore <keystorepath> - Path of the keystore to be used to sign the APK file..
  - -storepass <keystorepassword> - Password of the keystore to be used to sign the APK file.
  - -keypass <privatekeypassword> - Password of the private key (if different than the keystore password) to be used to sign the APK file.

  For more about using additional MATT parameters, use:

sh MakeAppTouchTestable/bin/MakeAppTouchTestable - help

7. Click Apply and then Save.

Build the Titanium Project

Now that we've revised our script to make the directory, clone the KitchenSink app source, and to make the app source touchtestable, the next section will be used to build the Titanium project using the Python builder.py script that comes with the Titanium SDK in use.

> **Note:** The builder.py file is available as part of all Titanium SDKs. However, you must use the one that corresponds to the Titanium SDK your project uses. As noted in Prerequisites, TouchTest supports only Titanium SDK 2.1.3 and later.

This section of our Execute Shell script will generate both an Xcode project and store it in the "build/iphone" sub-directory of your Titanium project (e.g. "~/dev/Jenkins/workspace/JenkinsFucntionalTests/KitchenSink/build/iphone/KitchenSink.xcodeproj").

Below is a sample section that uses all the parameters above (as commented fields) followed by the command itself.

```
# Build the Titanium project.

#

# Parameters:

# mode

# Minimum iOS version ("5.1" in this example)

# Path to the Titanium project directory

# Titanium app ID

# Titanium app name

# iOS provisioning profile identifier (see comments below)

# iOS provisioning profile owner (see comments below)

$ "/Library/Application Support/Titanium/mobilesdk/osx/2.1.3.GA/iphone/
builder.py" install 5.1 "~/dev/jenkins/workspace/JenkinsFucntionalTests/
KitchenSink" com.appcelerator.kitchensink KitchenSink b92b4cc0-fc88-11e1-
a21f-0800200c9a66 "John Doe" universal
```

1. Determine all of the following values and then paste the revised command into the Execute Shell script using the sample at the end of this section.

   You will need the following information to proceed:

- Mode – Users have 3 options as a mode. Choose the mode based on your Apple Profile (use Xcode's Organizer to review your profile options).
  - **install** (requires Apple **Developer** Profile)
  - **adhoc** (requires Apple **Distribution** Profile)

    - **distribute** (requires Apple **Distribution** Profile).
- Minimum iOS version – For example, 5.1.
- Path to the Titanium project directory on the Jenkins node (will be the same as the one used for MakeAppTouchTestable above).
- Titanium app ID - the app ID can be found in tiapp.xml (in the <id/> element). For this example, the ID com.appcelerator.titanium
- Titanium app Name – the app Name can be found in tiapp.xml (in the name/> element. For this example, KitchenSink
- iOS provisioning profile identifier - The provisioning profile identifier is typically located in the "~/Library/MobileDevice/Provisioning Profiles" directory. You'll find one or more files with the extension ".mobileprovision". The profile ID is the name of the file, minus the extension.
- iOS provisioning profile owner Team Name- the profile "owner" is the name of the iOS development team to which the profile is registered.

Preparing the App for Simulators and Devices

The iOS App Installer Utility contains two executable files—`ios_sim_launcher` and `ios_app_installer`. Deployment is achieved for both Simulators and iPad/IPhone devices using these executables and the steps described below. Unzip the utility at this time if you have yet to do so and note the contents of the resulting iOSAppInstaller folder.

- For a Simulator, the `./bin/ios_sim_launcher -app` command is used

  In order to deploy to a simulator, we must first build an APP file and then use that APP file with the `ios_sim_launcher`

- For an iPhone or iPad, the `./bin/ios_app_installer -ipa`

  In order to deploy to a physical device, we must first build an APP file, followed by building an IPA file, after which we can use the `ios_app_installer`.

Build the APP for a Simulator

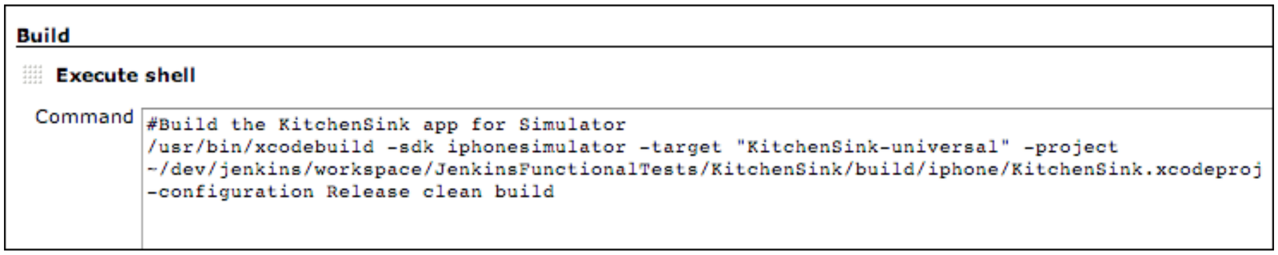
The app file is a requirement for deployment to all iOS simulators and devices.

In order to build the app, the command line xcodebuild command is used.

1. Enter the following lines in the end of the Execute Shell field (revise the paths to match that of your own environment):

   ```
   #Build the KitchenSink app for Simulator
   ```

   ```
   /usr/bin/xcodebuild -sdk iphonesimulator -target "KitchenSink-universal"
   -project ~/dev/jenkins/workspace/JenkinsFunctionalTests/KitchenSink/
   build/iphone/KitchenSink.xcodeproj -configuration Release clean build
   ```



```
#Build the KitchenSink app for Simulator
/usr/bin/xcodebuild -sdk iphonesimulator -target "KitchenSink-universal" -project
~/dev/jenkins/workspace/JenkinsFunctionalTests/KitchenSink/build/iphone/KitchenSink.xcodeproj
-configuration Release clean build
```

   where (as in the MakeAppTouchTestable command above):

   - `<sdk>` in this case is iphonesimulator
   - `<target>` is the name of the target in the ".xcodeproj" file representing your project. Note that you need to either specify the sdk as an argument (recommended) or create a Target specifically for Simulators
   - `<project>` is the path of the ".xcodeproj" file to build
   - <configuration> is the type of build. Refer to `/usr/bin/xcodebuild –help` for more information.

After the APP has been successfully created, it can be deployed to a simulator using the steps in the next section. If you are deploying to an iPhone or iPad, you must first create an IPA archive before deployment.

Deploying the APP File on a Simulator

Once the APP file has been created, use the following step to run the `ios_sim_launcher` to deploy the compiled APP file.

Before starting, note the path to the unarchived iOS App Installer Utility folder where `ios_sim_launcher` resides.

1. In the Execute Shell field, enter the following (be sure to use your own paths as well as to specify the iOS SDK and family of the simulator to use):

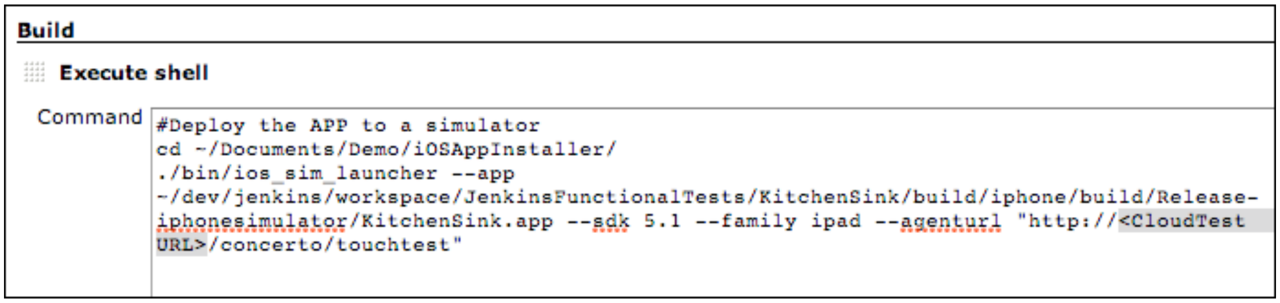
   ```
   #Deploy the APP to a simulator

   cd ~/Documents/Demo/iOSAppInstaller/

   ./bin/ios_sim_launcher --app ~/dev/jenkins/workspace/
   JenkinsFunctionalTests/KitchenSink/build/iphone/build/Release-
   iphonesimulator/KitchenSink.app --sdk 5.1 --family ipad --agenturl
   "http://<CloudTest URL>/concerto/touchtest"
   ```

The `ios_sim_launcher` requires the app path:

- `--app <app path>` - The path to the compiled APP.



Build

Execute shell

Command
```
#Deploy the APP to a simulator
cd ~/Documents/Demo/iOSAppInstaller/
./bin/ios_sim_launcher --app
~/dev/jenkins/workspace/JenkinsFunctionalTests/KitchenSink/build/iphone/build/Release-
iphonesimulator/KitchenSink.app --sdk 5.1 --family ipad --agenturl "http://<CloudTest
URL>/concerto/touchtest"
```

Note: The highlighted CloudTest URL placeholder in the above shot, which originates in the appendix script at the end of this tutorial, must be replaced with your own CloudTest URL.

The `ios_sim_launcher` takes the following additional parameters. These are used in the example above to specify the SDK and family:

- `--sdk <version>` - The iOS SDK version to use. For example, 5.1.
- `--family <list>` - The simulated device family, values are ‘iphone’ (default), ipad, iphone_retina, and ipad_retina.
- `--agenturl` – The agenturl parameter specifies the TouchTest Agent URL to use to launch the device agent, which is a requirement for continuous integration. The value is the CloudTest URL with the /concerto/touchtest URL string appended.

Installing the IPA Archive on iPhones and iPads

In this section, we'll run the ios_app_installer to deploy the IPA archive created in the prior section. Before starting, note the path to the downloaded iOS App Installer Utility.

In the example below, the IPA archive is in the Jenkins workspace we already defined as part of this Jenkins job.

1. In the Execute Shell field, enter the following (be sure to revise this example to use your own paths):

```
#Deploy the IPA to all devices

cd ~/Documents/Demo/iOSAppInstaller/

./bin/ios_app_installer --ipa ~/dev/jenkins/workspace/
JenkinsFunctionalTests/KitchenSink/build/iphone/build/Debug-iphoneos/
KitchenSink.ipa
```

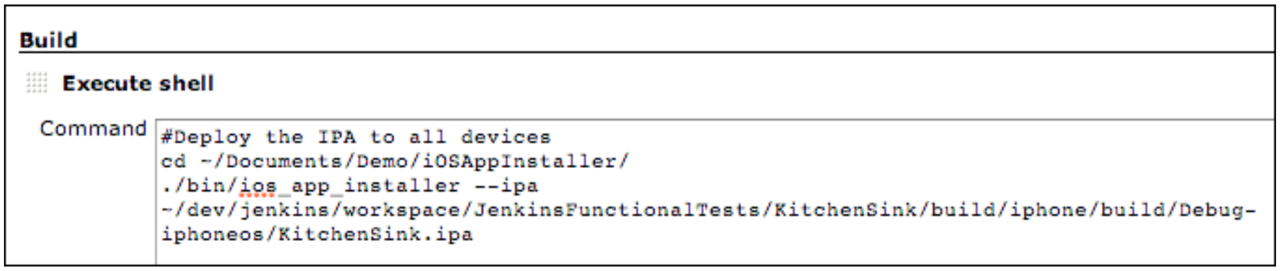
The script requires the following parameters:

- --ipa <ipapath> - The path to the IPA archive.

Note: Your user-configurable IPA archive location will vary. In the above example, the file is in the Jenkins workspace that we specified as ~/dev/jenkins/workspace/JenkinsFunctionalTests/KitchenSink/build/iphone/build/Debug-iphoneos/.

The iOS App Installer Utility will deploy to all the tethered provisioned devices by default. If you'd like to limit the deployment to specific devices use the following optional parameters:

- --udid <list> - One or more device UDID in a comma-separated list, if unspecified it install on all the connected iOS devices
- --device <list> : device name list comma-separated, if unspecified it install on all the connected iOS devices



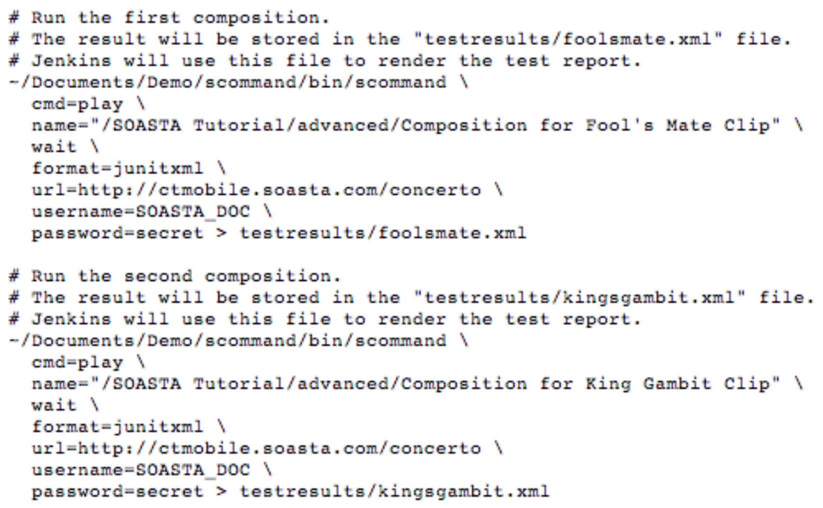
```
#Deploy the IPA to all devices
cd ~/Documents/Demo/iOSAppInstaller/
./bin/ios_app_installer --ipa
~/dev/jenkins/workspace/JenkinsFunctionalTests/KitchenSink/build/iphone/build/Debug-
iphoneos/KitchenSink.ipa
```

Using SCommand to Play One or More Compositions

Next, we will add SCommand lines that will silently play the specified test compositions on the specific CloudTest instance. Additionally, we will add arguments that will output junitxml-compatible XML code that will appear in Jenkins for each test result.

1. In the Execute Shell field, enter the following (be sure to substitute the full path for the composition you'd like to use):

```
# Run the first composition.
# The result will be stored in the "testresults/kitchensink1.xml" file.
# Jenkins will use this file to render the test report.
~/Documents/Demo/scommand/bin/scommand \
  cmd=play \
  name="/SOASTATutorial/Titanium/Composition for KitchenSink1" \
  wait \
  format=junitxml \
  url=http://ctmobile.soasta.com/concerto \
  username=SOASTA_DOC
  password=secret >testresults/kitchensink1.xml
```



```
# Run the first composition.
# The result will be stored in the "testresults/foolsmate.xml" file.
# Jenkins will use this file to render the test report.
~/Documents/Demo/scommand/bin/scommand \
  cmd=play \
  name="/SOASTA Tutorial/advanced/Composition for Fool's Mate Clip" \
  wait \
  format=junitxml \
  url=http://ctmobile.soasta.com/concerto \
  username=SOASTA_DOC \
  password=secret > testresults/foolsmate.xml

# Run the second composition.
# The result will be stored in the "testresults/kingsgambit.xml" file.
# Jenkins will use this file to render the test report.
~/Documents/Demo/scommand/bin/scommand \
  cmd=play \
  name="/SOASTA Tutorial/advanced/Composition for King Gambit Clip" \
  wait \
  format=junitxml \
  url=http://ctmobile.soasta.com/concerto \
  username=SOASTA_DOC \
  password=secret > testresults/kingsgambit.xml
```

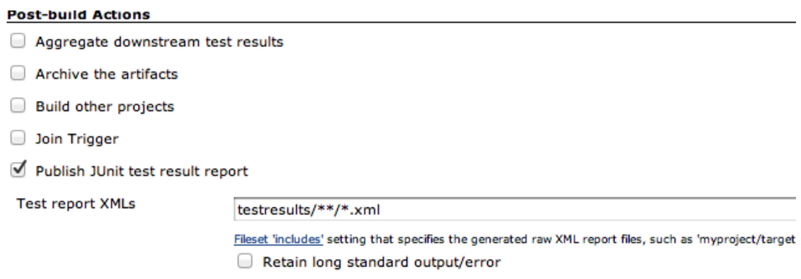
2. Enter any additional compositions for the job:

```
# Run the second composition.
# The result will be stored in the "testresults/kitchensink2.xml" file.
# Jenkins will use this file to render the test report.
~/Documents/Demo/scommand/bin/scommand \
  cmd=play \
  name="/SOASTATutorial/Titanium/Composition for KitchenSink2" \
  wait \
  format=junitxml \
  url=http://ctmobile.soasta.com/concerto \
  username=SOASTA_DOC \
  password=secret >testresults/kitchensink2.xml
```

Adding the Publish Junit Test Result Reports Step

Next, we will add a Post-build action that will display SCommand output in Jenkins and opt in to plugin display as well.

1. In the Post-build action section, check "Publish JUnit test result report."



Post-build Actions

- [ ] Aggregate downstream test results
- [ ] Archive the artifacts
- [ ] Build other projects
- [ ] Join Trigger
- [✓] Publish JUnit test result report

Test report XMLs: testresults/**/*.xml

Fileset 'includes' setting that specifies the generated raw XML report files, such as 'myproject/target

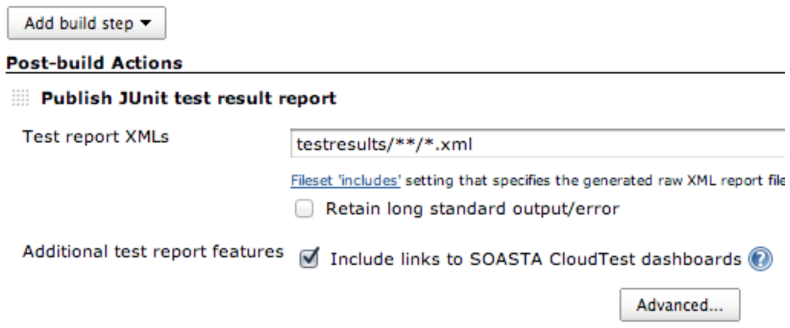
- [ ] Retain long standard output/error

2. In the Test report XMLs field, enter a path where the JUnit XML will be created. This should be in the Jenkins workspace. The folder need not exist prior to the first build.

   For example, the folder `testresults/**/*.xml.`

   Note: The error shown below will display the no match error until after the first build.

3. Under the Additional test report features section, check the 'Include links to SOASTA CloudTest dashboards' box.



Post-build Actions

Publish JUnit test result report

Test report XMLs: testresults/**/*.xml

Fileset 'includes' setting that specifies the generated raw XML report file

- [ ] Retain long standard output/error

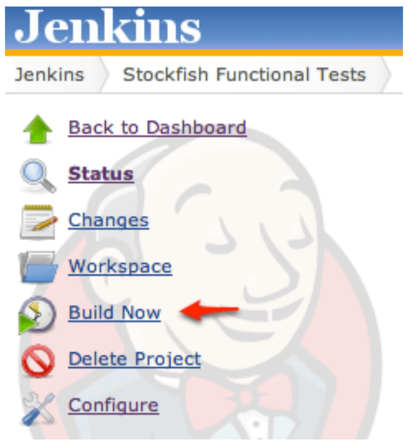
Additional test report features: [✓] Include links to SOASTA CloudTest dashboards

4. Check the Include links to SOASTA CloudTest dashboards box to opt into the plugin for this job.
5. Click Save to exit the job.

Building the Project

After you click the Save button, you will be taken to the project page for the job you just created.

1. To build the project, click the "Build Now" link.



The build will start. After a short delay, you should see a progress bar appear on the left side of the page. Click this progress bar to watch the build process "live" in the Console view.

You should see the following happen:

a. Jenkins checks out the source code from Git, and runs the MakeAppTouchTestable utility.

b. Jenkins runs the DeployProjectToDevice script. On the build node, you should see Xcode immediately launch and deploy the KitchenSink app to the tethered device. **Do not interact with Xcode while this is happening.** Once the app has been deployed, Xcode will automatically exit.

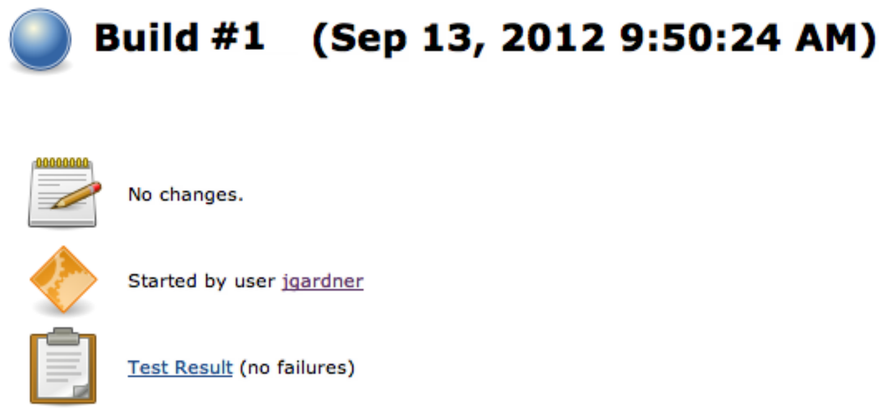
c. On the tethered device, you should see the KitchenSink app briefly appear. It will immediately switch over to the SOASTA TouchTest Agent web page (in Safari), with the Status showing "Connected" <insert screen shot>.

d. Jenkins plays the CloudTest compositions using SCommand. On the tethered device, you should see the KitchenSink app launch and run through the test steps. When the test finishes, KitchenSink will exit, and the SOASTA TouchTest Agent page will re-open.

Inspecting Test Results in Jenkins

On the build summary page, the Test Result link (added by the steps in the previous section) appears.

1. Click Test Result.



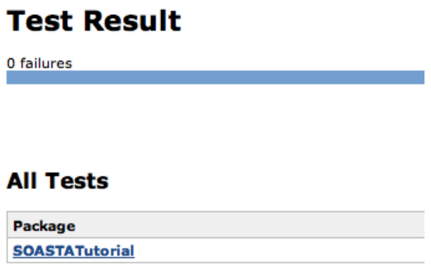
Build #1 (Sep 13, 2012 9:50:24 AM)

No changes.

Started by user jgardner

Test Result (no failures)

For a successful test with no failures, the Test Result page merely lists the All Tests section with the given package (i.e. in this case the package equates to a CloudTest repository folder).



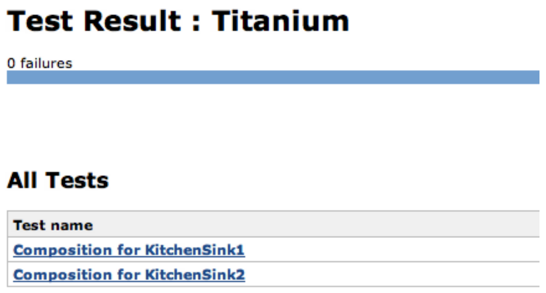
Test Result

0 failures

All Tests

Package
SOASTATutorial

2. Click the Package link until you reach the composition folders. Each click opens the subsequent CloudTest folder.



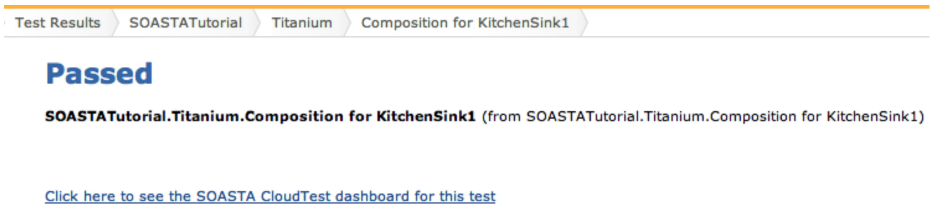
Test Result : Titanium

0 failures

All Tests

Test name
Composition for KitchenSink1
Composition for KitchenSink2

The composition was a success in this case.

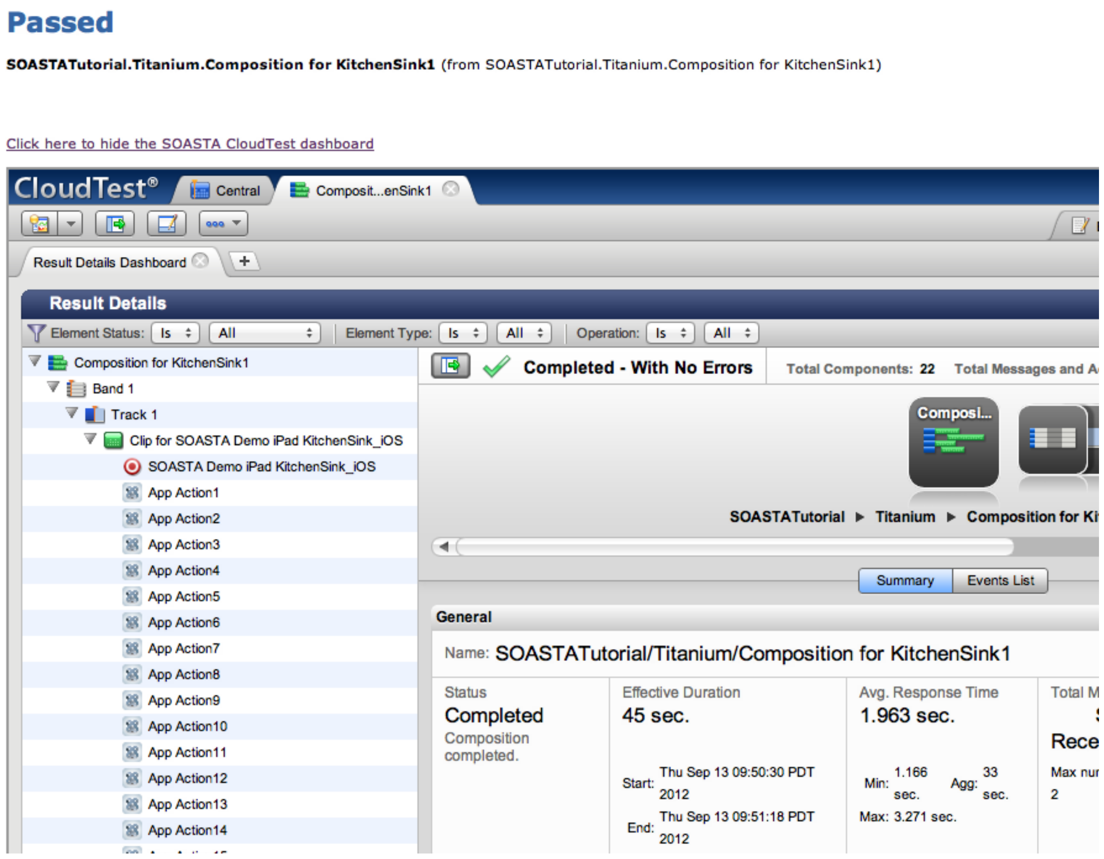


Passed

SOASTATutorial.Titanium.Composition for KitchenSink1 (from SOASTATutorial.Titanium.Composition for KitchenSink1)

Click here to see the SOASTA CloudTest dashboard for this test

3. Use the “Click here…” dashboard link to view CloudTest inline. Login using your credentials if you get the Login page. After doing so, the dashboard for the given composition is shown.



Passed

SOASTATutorial.Titanium.Composition for KitchenSink1 (from SOASTATutorial.Titanium.Composition for KitchenSink1)

Click here to hide the SOASTA CloudTest dashboard

Completed - With No Errors

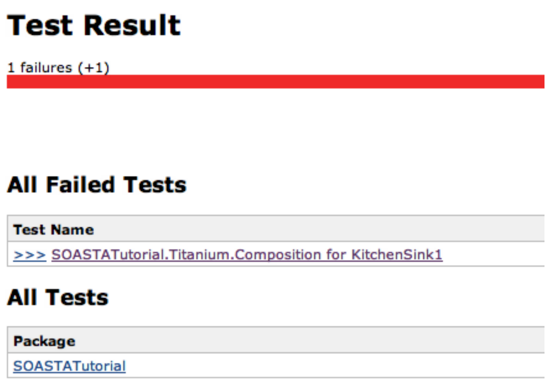
Total Components: 22 Total Messages and A

General

Name: SOASTATutorial/Titanium/Composition for KitchenSink1

Status	Effective Duration	Avg. Response Time	Total M
Completed	45 sec.	1.963 sec.	
Composition completed.	Start: Thu Sep 13 09:50:30 PDT 2012	Min: 1.166 sec. Agg: 33 sec.	Rece
	End: Thu Sep 13 09:51:18 PDT 2012	Max: 3.271 sec.	Max nu 2

Things get more interesting when an error in the test occurs. The subsequent SCommand output is displayed (in text) on the Jenkins Test Result page (as discussed above).


Test Result

1 failures (+1)

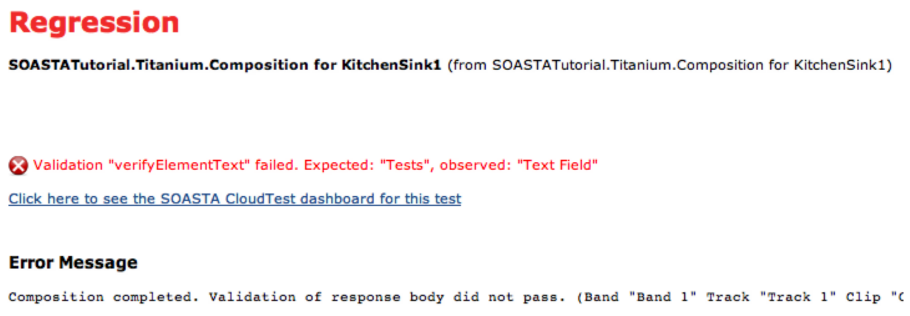
All Failed Tests

Test Name
>>> SOASTATutorial.Titanium.Composition for KitchenSink1

All Tests

Package
SOASTATutorial

In this case, the All Failed Tests section is added with the name of the test listed with a link to more of the SCommand details. Clicking the link under the Test Name section where the composition is named displays an Error detail page (for the given error).


Regression

SOASTATutorial.Titanium.Composition for KitchenSink1 (from SOASTATutorial.Titanium.Composition for KitchenSink1)

Validation "verifyElementText" failed. Expected: "Tests", observed: "Text Field"

Click here to see the SOASTA CloudTest dashboard for this test

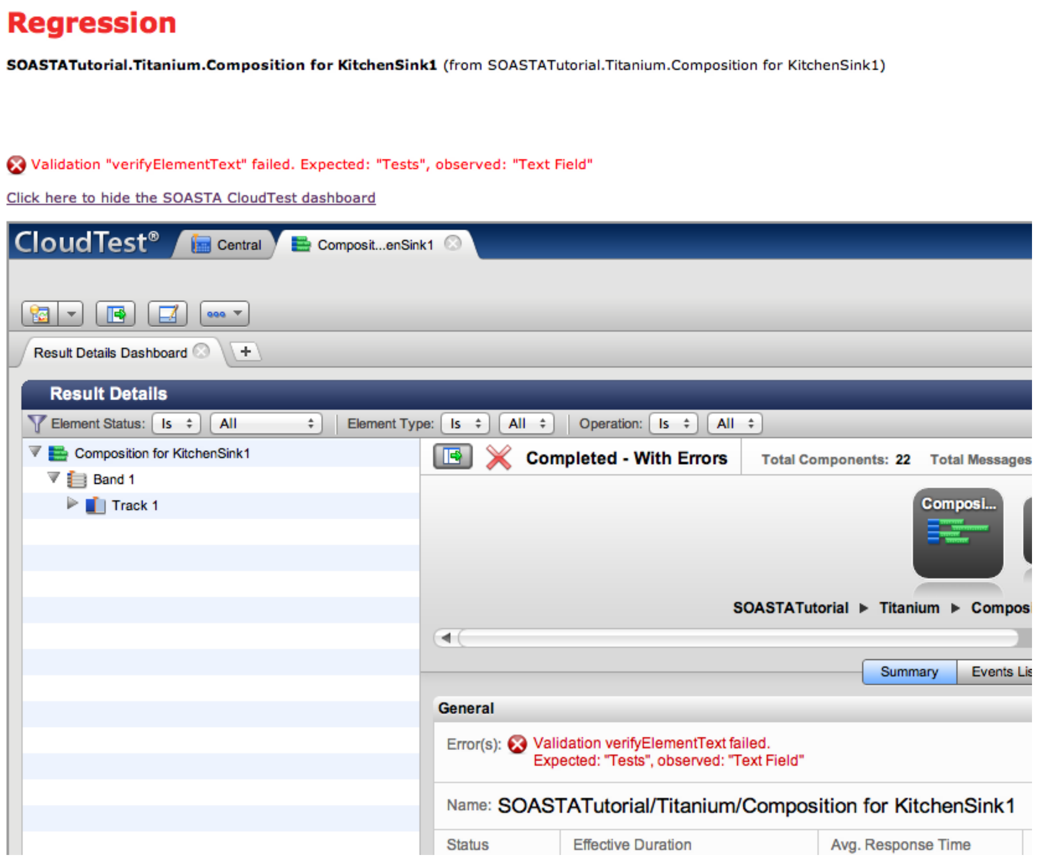
Error Message

```
Composition completed. Validation of response body did not pass. (Band "Band 1" Track "Track 1" Clip "C
```

In the error above a validation in the Composition for KitchenSink1 has failed.

4. To view this error in the CloudTest, Result Details dashboard, click the plugin link provided (i.e. “Click here to see the SOASTA CloudTest dashboard for this test).”

After the plugin link is clicked, enter CloudTest credentials whenever required.


Regression

SOASTATutorial.Titanium.Composition for KitchenSink1 (from SOASTATutorial.Titanium.Composition for KitchenSink1)

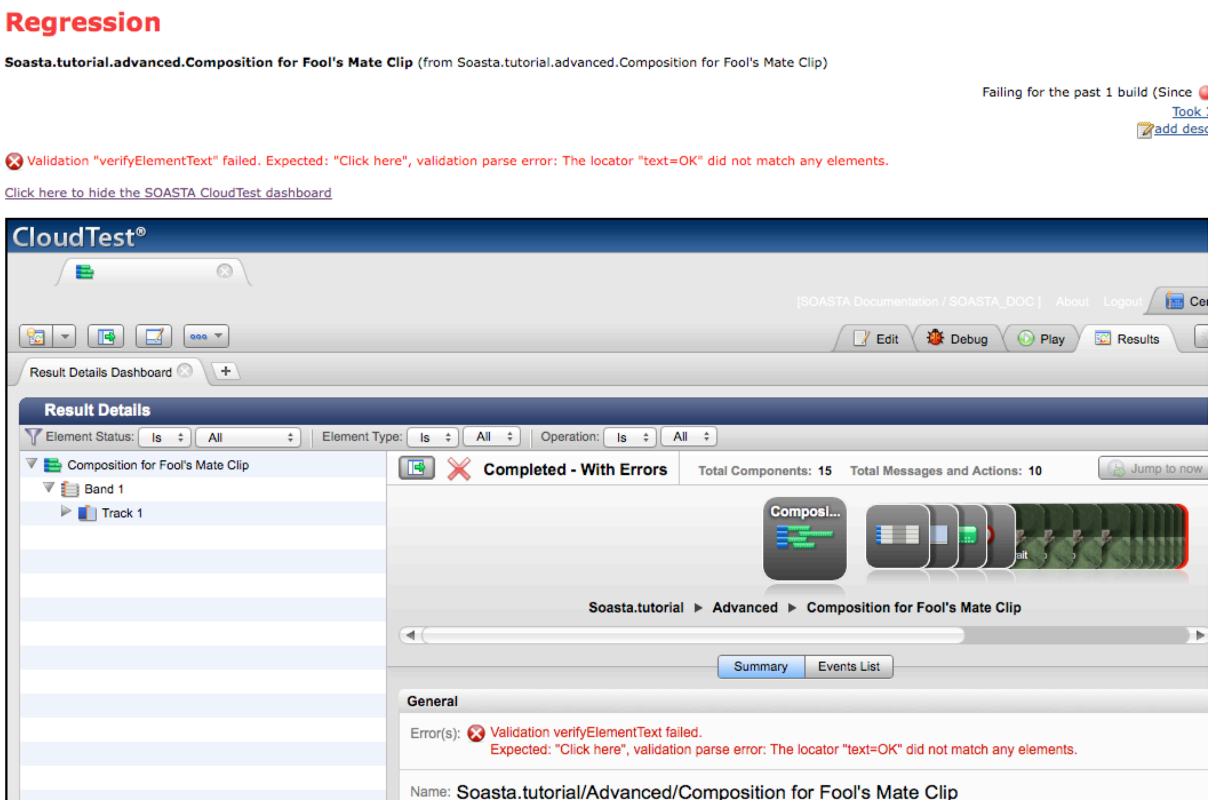
Validation "verifyElementText" failed. Expected: "Tests", observed: "Text Field"

Click here to hide the SOASTA CloudTest dashboard

Error(s): Validation verifyElementText failed. Expected: "Tests", observed: "Text Field"

Name: SOASTATutorial/Titanium/Composition for KitchenSink1

- After credentials are entered, the dashboard tab opens, displays the test result, and jumps to the relevant error.


Regression

Soasta.tutorial.advanced.Composition for Fool's Mate Clip (from Soasta.tutorial.advanced.Composition for Fool's Mate Clip)

Failing for the past 1 build (Since

Took

add desc

Validation "verifyElementText" failed. Expected: "Click here", validation parse error: The locator "text=OK" did not match any elements.

Click here to hide the SOASTA CloudTest dashboard

Completed - With Errors Total Components: 15 Total Messages and Actions: 10

General

Error(s): Validation verifyElementText failed.
Expected: "Click here", validation parse error: The locator "text=OK" did not match any elements.

Name: Soasta.tutorial/Advanced/Composition for Fool's Mate Clip

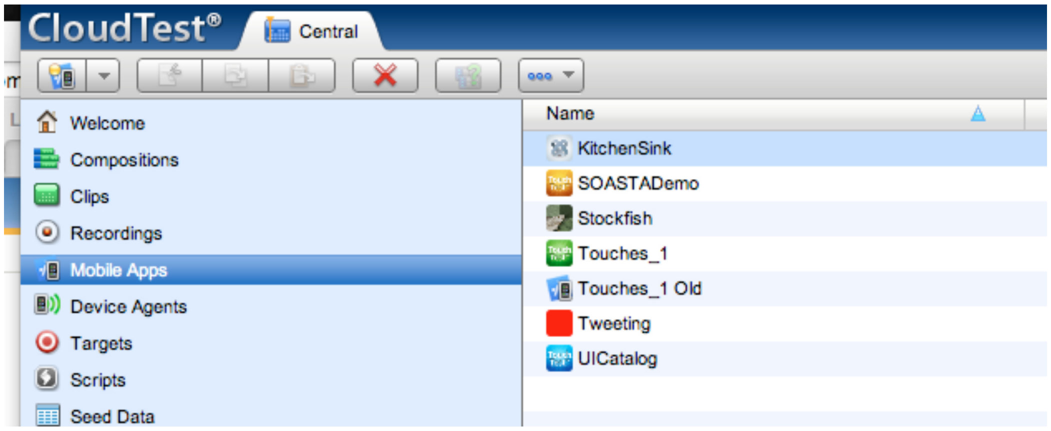
From here, the Result Details dashboard can be navigated as within any CloudTest dashboard. Refer to Result Details Dashboard for a quick review of Result Details features.

Appendix: Inspecting the Mobile App in CloudTest®

In the steps above at the end of each run of the MakeAppTouchTestable.jar we were notified that the “Mobile App Object” had been created in the CloudTest® Repository.

> **TIP:** This mobile app will appear in the Choose Device Agent and Mobile App box whenever end-users start a mobile app recording. Selecting which mobile app to launch on which test devices is a crucial end-user step.

1. Optionally, verify that the Mobile App has been added by logging into CloudTest® and looking for its entry in the Central > Mobile Apps list. For example, in the screenshot below the *KitchenSink copy* app appears as expected.

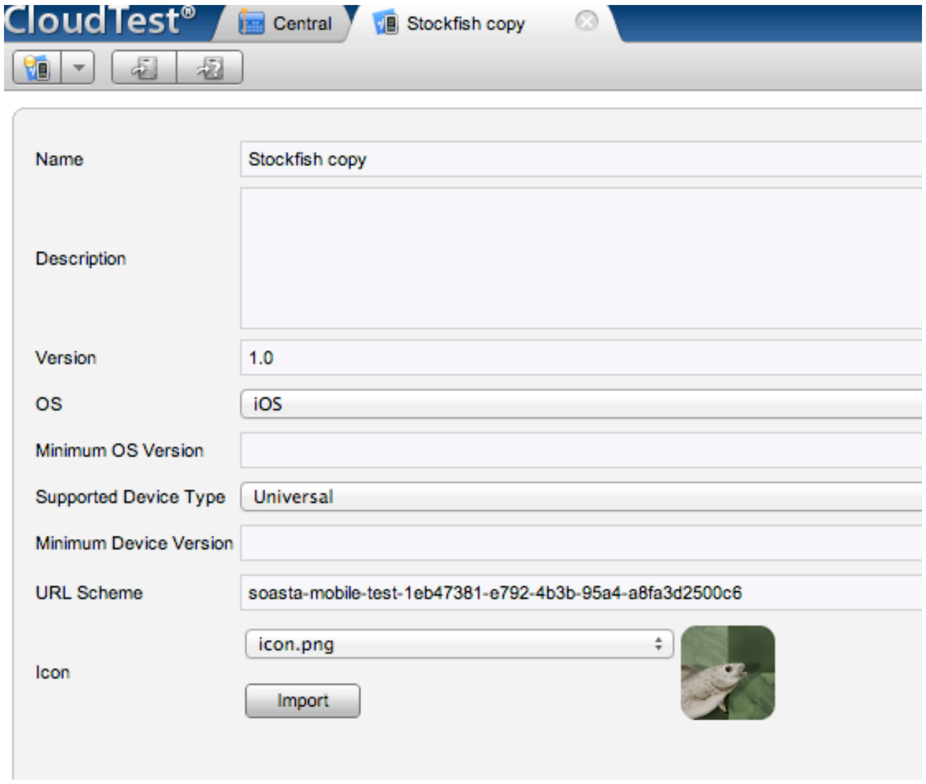


2. Double-click the KitchenSink Mobile App to inspect its details.

   The Mobile App detail form appears.

   - All of the fields shown were populated from the Xcode project, with the exception of Supported Device Type and Minimum OS Version.

- The default Supported Device Type is Universal (e.g. both iPhone and iPad). Leave this as is, since we will be specifying multiple devices later.



Name	Stockfish copy
Description	
Version	1.0
OS	iOS
Minimum OS Version	
Supported Device Type	Universal
Minimum Device Version	
URL Scheme	soasta-mobile-test-1eb47381-e792-4b3b-95a4-a8fa3d2500c6
Icon	icon.png

The default Minimum OS Version supported in TouchTest™ beta is iOS 5.0

Appendix: Completed Execute Shell Script

The completed Execute Shell script described above is presented in its entirety below. You can use this as a template to insert your own paths as well as your own CloudTest URL wherever it is required.

```
#Continue playing subsequent compositions if a prior composition fails

set +e


#Create the test result reports folder if it doesn't already exist

mkdir -p testresults


#Remove the version we downloaded as part of last build (if any)

rm -rf "KitchenSink"


#Download KitchenSink from Github

/usr/local/git/bin/git clone https://github.com/appcelerator/KitchenSink


#Run MakeAppTouchTestable (be sure to enter your CloudTest URL here)

sh MakeAppTouchTestable/bin/MakeAppTouchTestable -project $WORKSPACE/
KitchenSink -target "KitchenSink" -url http://<CloudTest URL>/concerto -
username SOASTA_DOC -password secret


# Build the Titanium project.

#

# Parameters:

# mode (should be "adhoc" for CI)

# Minimum iOS version ("5.1" in this example)

# Path to the Titanium project directory

# Titanium app ID

# Titanium app name

# iOS provisioning profile identifier

# iOS provisioning profile team name

"/Library/Application Support/Titanium/mobilesdk/osx/2.1.3.GA/iphone/
builder.py" install 5.1 ~/dev/jenkins/workspace/JenkinsFunctionalTests/
KitchenSink com.appcelerator.kitchensink KitchenSink 20433FAB-7A9C-4996-BDD3-
E660DB279261 "James GARDNER" universal
```

```
#Build the KitchenSink app for Simulator

/usr/bin/xcodebuild -sdk iphonesimulator -target "KitchenSink-universal" -
project ~/dev/jenkins/workspace/JenkinsFunctionalTests/KitchenSink/build/
iphone/KitchenSink.xcodeproj -configuration Release clean build


#Deploy the APP to a simulator (be sure to enter your CloudTest URL here)

cd ~/Documents/Demo/iOSAppInstaller/

./bin/ios_sim_launcher --app ~/dev/jenkins/workspace/JenkinsFunctionalTests/
KitchenSink/build/iphone/build/Release-iphonesimulator/KitchenSink.app --sdk
5.1 --family ipad --agenturl "http://<CloudTest URL>/concerto/touchtest"


#Deploy the IPA to all devices

cd ~/Documents/Demo/iOSAppInstaller/

./bin/ios_app_installer --ipa ~/dev/jenkins/workspace/JenkinsFunctionalTests/
KitchenSink/build/iphone/build/Debug-iphoneos/KitchenSink.ipa


# Run the first composition.

# The result will be stored in the "testresults/kitchensink1.xml" file.

# Jenkins will use this file to render the test report.

~/Documents/Demo/scommand/bin/scommand \

  cmd=play \

  name="/SOASTATutorial/Titanium/Composition for KitchenSink1" \

  wait \

  format=junitxml \

  url=http://ctmobile.soasta.com/concerto \

  username=SOASTA_DOC \

  password=secret >testresults/kitchensink1.xml


# Run the second composition.

# The result will be stored in the "testresults/kitchensink2.xml" file.

# Jenkins will use this file to render the test report.

~/Documents/Demo/scommand/bin/scommand \

  cmd=play \

  name="/SOASTATutorial/Titanium/Composition for KitchenSink2" \

  wait \

  format=junitxml \

  url=http://ctmobile.soasta.com/concerto \
  username=SOASTA_DOC \
  password=secret >testresults/kitchensink2.xml
```

SOASTA, Inc.

444 Castro St.

Mountain View, CA 94041

866.344.8766

http://www.soasta.com