



SST Unit Tester User Guide

Document Revision History

Revision Date	Written/Edited By	Comments
September 2016	Andy Dunfee	Initial release with SSD v2

© Copyright 2017 SailPoint Technologies, Inc., All Rights Reserved.

SailPoint Technologies, Inc. makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. SailPoint Technologies shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Restricted Rights Legend. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of SailPoint Technologies. The information contained in this document is subject to change without notice.

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

Regulatory/Export Compliance. The export and reexport of this software is controlled for export purposes by the U.S. Government. By accepting this software and/or documentation, licensee agrees to comply with all U.S. and foreign export laws and regulations as they relate to software and related documentation. Licensee will not export or reexport outside the United States software or documentation, whether directly or indirectly, to any Prohibited Party and will not cause, approve or otherwise intentionally facilitate others in so doing. A Prohibited Party includes: a party in a U.S. embargoed country or country the United States has named as a supporter of international terrorism; a party involved in proliferation; a party identified by the U.S. Government as a Denied Party; a party named on the U.S. Government's Entities List; a party prohibited from participation in export or reexport transactions by a U.S. Government General Order; a party listed by the U.S. Government's Office of Foreign Assets Control as ineligible to participate in transactions subject to U.S. jurisdiction; or any party that licensee knows or has reason to know has violated or plans to violate U.S. or foreign export laws or regulations. Licensee shall ensure that each of its software users complies with U.S. and foreign export laws and regulations as they relate to software and related documentation.

Trademark Notices. Copyright © 2017 SailPoint Technologies, Inc. All rights reserved. SailPoint, the SailPoint logo, SailPoint IdentityIQ, and SailPoint Identity Analyzer are trademarks of SailPoint Technologies, Inc. and may not be used without the prior express written permission of SailPoint Technologies, Inc. All other trademarks shown herein are owned by the respective companies or persons indicated.

Table of Contents

Overview.....	4
The Section You Can Skip	4
Development Paradigm	5
Usage.....	8
Setup	16
Inventory Contents	16
Initial Setup Steps.....	19
Target Properties	22
SST_VIEW_UNIT_TESTS_WF_TRACE	22
SST_UT_OUTPUT_PATH.....	22
SST_UT_OUTPUT_PATH_BUILD	22
SST_RUN_ALL_UT_RECIPIENTS	22
SST_IIQ_INSTANCE_NAME	23
SST_UT_ARGS_PATH	23
SST_UT_TARGET	23
FILE_SEPARATOR	23
Configuring Your First Test	25
Advanced Topics	35
Void Methods	35
Multiple Updates	35
Repeatability	35
Test Other Things	36

Overview

This quick guide details the setup and usage of the SST Unit Tester. It assumes the reader has a more than basic understanding of IdentityIQ and the Services Standard Build.

The SST Unit Tester is a set of IdentityIQ configuration objects that provides a standard and repeatable way to test “atomic” units of IdentityIQ code, such as rules and methods in rule libraries. It is not intended for load testing or mimicking end user UI interactions. It should not replace any of the customer’s testing. It is, mostly, a supplemental tool for developers to better unit test their code and provide a way to quickly regression test code changes throughout the development process.

The Section You Can Skip

Some may wonder why this exists or what was the big idea. If you wonder this or are looking for a reason to use this thing and thus keep reading this document, continue reading this section. If you already get it, don’t care about the why, or are only using this because you were told to, skip to the next section.

Oh, you’re still here? Okay, great. Let’s go. Imagine the very common use case of onboarding a new user. This typically means a new account shows up in an HR system, IdentityIQ consumes this record, initializes a bunch of identity attributes, kicks off a Joiner workflow, assigns a Business Role, provisions an account, most likely Active Directory, and finally sends the new user’s manager an email containing the initial login credentials of the new user. To make all that happen, you have to build quite a few individual components:

- HR application and schema
- HR correlation rule
- HR identity creation rule
- The HR aggregation task
- The Identity object config
- Any number of identity attribute rules
- A lifecycle event with an identity trigger rule
- A workflow with
 - A build plan step
 - A call to the Initialize step, which invokes the plan compiler and the target application’s provisioning policy (see below)
 - Call to provision step
 - An after provision step with logic to send the email
- Business role
- IT Role
- Target application provisioning policy with any number of fields and field value rules
- Refresh task

Quite a list. Within that list of components are a number of places where you, the developer, will need to write some code, specifically Beanshell scripts. It’s not uncommon for there to be, just in that list above, as many as 25-30 individual chunks of script logic.

How does one test all that?

At the end of the day, unit test framework or not, you'll have to assembly test all of it and the way that's going to be done is quite simple:

- Write a new record to the HR system or table
- Run an aggregation against the HR system
- Run a refresh against your new user
- Evaluate the results

Honestly, when looking at it like that, it's not that bad... the first 5 or 6 times. But if you are using this methodology to slowly move the dial in terms of what line or point in the overall solution your code breaks, and every time you go to retest you either need to completely delete all of your test results or have to create a brand new entry in the HR system, you are going to get fatigued and might start to get lazy and might limit the breadth of your test cases.

The other problem with this approach is that you tend to need to build up a certain critical mass of code before you can even start to test some things. You might write something on a Monday that you can't test until Wednesday. We believe you should aim to get feedback much sooner, like every hour of every day. No, seriously.

What's the answer to all that? Well, a unit test framework. A way to test the individual pieces of code by themselves with a way to quickly change the input parameters to cover a broad array of scenarios.

That's what this is all about: providing a way for developers to test very small units of code in a repeatable manner with a wide range of inputs and results in a way that provides an easy-to-read report of success and failure with the reason why something failed (so you can then go back and fix it).

The other thing this allows for is for Solution Architects working on a large team with multiple developers to better track, in a more qualified manner, the code written by the other developers.

We'll assume a burning question at this point is: Will it save me time?

The answer is No and Yes. It is more up front work to write each test for each thing you want to test. There's no getting around that. But we believe, by giving immediate feedback on smaller chunks of code, versus sifting through a larger test and set of logs, by allowing for a wider range of test cases, by basically giving a means of writing a much better tested and overall cleaner solution, you will save tons of time on the backend. And your customers will love you for it.

All that being said, let's get started.

Development Paradigm

For this unit test tool, a certain development paradigm is assumed, though not required. The central tenet of this paradigm is that any script code, no matter how simple or complex, will live inside of a method inside of a rule library and then that method is called by a given rule or workflow.

This design paradigm can have some drawbacks, namely performance impacts (during very specific use cases, namely when a rule is called by the Multi-threaded beanshell kit) and greater levels of abstraction. But there are also advantages: cleaner code (in general and very specifically in workflows), more reuse, better logging, and code that is easier to unit test.

The following are examples of code that does and does not follow this given paradigm.

```
<Step name="Set IIQ Password">
  <Script>
    <Source>
      import sailpoint.object.Identity;

      Identity identity = context.getObject(Identity.class, identityName);
      identity.setPassword(password);

      context.saveObject(identity);
      context.commitTransaction();
      context.detach(identity);

      identity = null;
    </Source>
  </Script>
</Step>
```

Figure 1 - Workflow step with code embedded

The above image shows a workflow step with the code embedded directly into the script's source tags. This is technically fine but can't be called by the unit tester and requires at least 3 more lines of code to turn on a unique (to this code) log4j logger.

```
<RuleLibraries>
  <Reference class="sailpoint.object.Rule" name="SP.Provisioning Rules Library"/>
  <Reference class="sailpoint.object.Rule" name="DEC Update Access Rules Library"/>
  <Reference class="sailpoint.object.Rule" name="DEC Util Rules Library"/>
  <Reference class="sailpoint.object.Rule" name="%%SP LCE COMMON RULES OBJECT NAME%%"/>
</RuleLibraries>
<Step icon="Start" monitored="true" name="Start" posX="2" posY="135">
  <Transition to="Get Request Type"/>
</Step>
<Step monitored="true" name="Get Request Type" resultVariable="requestType">
  <Script>
    <Source>
      return getRequestTypeRule(workflow);
    </Source>
  </Script>
  <Transition to="Build Plan" />
</Step>
<Step name="Build Plan" resultVariable="plan">
  <Script>
    <Source>
      return buildUpdateAccessPlan(identityName, ou);
    </Source>
  </Script>
  <Transition to="No Request Failure" when="plan == null || plan.getAccountRequests() == null"/>
  <Transition to="Process Plan"/>
</Step>
```

Figure 2 - Workflow using libraries and method calls

The above image shows a workflow that references multiple libraries and where each script step is simply a method call.

```
import sailpoint.object.*;
import sailpoint.api.SailPointContext;
import java.util.*;
import java.text.*;
import sailpoint.tools.GeneralException;
import sailpoint.tools.Util;
import sailpoint.tools.xml.XMLObjectFactory;
import sailpoint.object.ProvisioningPlan;
import sailpoint.object.ProvisioningPlan.AttributeRequest;
import sailpoint.object.ProvisioningPlan.AccountRequest;
import sailpoint.tools.Util;
import sailpoint.tools.Message;
import sailpoint.dec.monitoring.Monitor;
import java.io.*;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.DriverManager;
import sailpoint.tools.JdbcUtil;
import sailpoint.object.ProvisioningPlan.Operation;
import sailpoint.api.*;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

private static Log uallogger = LogFactory.getLog("rule.DEC.UnitTest.RulesLibrary");
```

Figure 3 - Rule library with single set of imports and log4j logger

```
/******
GET REQUEST TYPE
******/
public static String getRequestTypeRule(Workflow workflow){
    return "LCE WF No Approvals";
}

/******
BUILD PLAN METHODS
******/
public static ProvisioningPlan buildUpdateAccessPlan(String identityName, String ou){
    uallogger.trace("Enter buildUpdateAccessPlan");
    ProvisioningPlan plan = new ProvisioningPlan();

    String appName = "";
    String nativeIdentity = "";
    AccountRequest.Operation op = AccountRequest.Operation.Modify;

    uallogger.trace("Get identity: " + identityName);
    Identity identity = context.getObjectByName(Identity.class, identityName);

    IdentityService is = new IdentityService(context);
    uallogger.trace("Get the primary AD link. ");
    Link link = getPrimaryADLink(identity);

    if (link == null){
        uallogger.warn("No primary AD link for " + identityName);
        return null;
    }
}
```

Figure 4 - Rule library methods can be called

The above two images show how using a rule library allows for declaring imports once and reusing one logger for multiple methods and shows the methods in the library that can now be called by the workflow, other rules and the SST Unit Tester, which we can now talk about.

Usage

Once the SST Unit Tester is setup, the general flow of its usage goes like this:

1. Write a method in a library
2. Write a corresponding unit tester rule to call the method
3. Write one or more argument files (inputs and expected result) to be passed into the unit tester rule for each unit test case
4. Register the rule and argument files in the SST Unit Tester mapping object
5. Run the unit tester rule from the console, the dashboard or a task
6. Analyze the results and repeat

Now, in more detail:

1. Write a method that does something. It accepts certain inputs and will spit out, based on those inputs, a certain output, leaving the possibility of any number of expected results.

For example, you write a method called `getFullName(String lastName, String firstName, String mi)`.

2. Write a unit test rule that will call the method. This rule will need to reference the rule library containing the method and will require input arguments for any value that needs to be different per test case when calling the method. The rule can contain whatever logic is required to complete the test but ultimately the rule must call the method and then return the value of that method (won't get into the complexities of testing void methods, methods that do more than just build one value).

For example, write a rule called `SST UT getFullName Rule`. The rule references the library containing the method from above. The input arguments are: `firstName`, `lastName`, `mi`, and `expectedResult`. The logic of the rule is fairly straightforward as it simply maps the input arguments to the method call and returns the result of the method call (much more complex unit test rules can be written).


```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Rule PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Rule name="SST UT getFullName Rule" language="beanshell">
  <ReferencedRules>
    <Reference class="sailpoint.object.Rule" name="CST Util Identity Attributes Library"/>
  </ReferencedRules>
  <Signature returnType="Map">
    <Inputs>
      <Argument name="argsFileName"/>
      <Argument name="lastName"/>
      <Argument name="firstName"/>
      <Argument name="mi"/>
      <Argument name="expectedResult"/>
    </Inputs>
    <Returns/>
  </Signature>
  <Source>
    <![CDATA[
      import sailpoint.object.*;
      import java.util.*;

      import org.apache.commons.logging.Log;
      import org.apache.commons.logging.LogFactory;

      private static Log utlogger = LogFactory.getLog("rule.SST.UnitTest.RulesLibrary");

      utlogger.trace("Enter SST Sample Unit Test Rule");

      utlogger.trace("Have the last name: " + lastName);
      utlogger.trace("Have the first name: " + firstName);
      utlogger.trace("Have the mi: " + mi);
      utlogger.trace("Have the expectedResult: " + expectedResult);

      String returnVal = getFullName(lastName, firstName, mi);

      utlogger.trace("Exit SST Sample Unit Test Rule: " + returnVal);
      return returnVal;
    ]]>
  </Source>
</Rule>
```

Figure 5 - Sample unit test rule

3. Write an argument file for each unique test case. The arg file is a mapping object that will contain each unique input and the expectedResult, which is absolutely required; it's what the unit tester is going to compare to determine success or failure. Additional fields, argsFileName and useCase, are used in the output report to add more context to the given context.

For example, in the given method, the results may be different based on different values being null. Or logic might need to truncate extra long strings. The following are two different args files that are used for the same unit test rule.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Map PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Map>
  <entry key="argsFileName" value="sst UT getFullName - Args 1 - All Present" />
  <entry key="useCase" value="Calculate full name with all values present" />
  <entry key='lastName' value="Smith" />
  <entry key='firstName' value="John" />
  <entry key='mi' value="M" />
  <entry key="expectedResult">
    <value>
      <String>John M Smith</String>
    </value>
  </entry>
</Map>
```

Figure 6 - Sample argument file

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Map PUBLIC "sailpoint.dtd" "sailpoint.dtd">
<Map>
  <entry key="argsFileName" value="sst UT getFullName - Args 1 - No Mi" />
  <entry key="useCase" value="Calculate full name with no middle initial" />
  <entry key='lastName' value="Smith" />
  <entry key='firstName' value="John" />
  <entry key='mi' value="" />
  <entry key="expectedResult">
    <value>
      <String>John Smith</String>
    </value>
  </entry>
</Map>
```

Figure 7 - Sample argument file

4. "Register" the rule and all arguments files in the SST Unit Tester Mapping Object. In the mapping object, there is an entry for every unit test rule. The name of the entry is the name of the rule object. Inside of this entry is a list of all the pertinent arg files. Each arg file entry is the path to the actual file. The correct file path will actually point to the location of the file after the build is executed and uses target properties to set the path in each environment. The rule entry also includes a category and the rule can be turned off by setting Enabled to false.

```
<entry key="SST UT getFullName Rule">
  <value>
    <Attributes>
      <Map>
        <entry key="Args">
          <value>
            <List>
              <String>%%SST_UT_ARGS_PATH%%Sample%%FILE_SEPARATOR%%SST_UT_getFullName_Args1.xml</String>
              <String>%%SST_UT_ARGS_PATH%%Sample%%FILE_SEPARATOR%%SST_UT_getFullName_Args2.xml</String>
            </List>
          </value>
        </entry>
        <entry key="Category" value="Identity Attribute" />
        <entry key="Enabled" value="true" />
      </Map>
    </Attributes>
  </value>
</entry>
```

Figure 8 - Register unit tester rule

5. Run the unit tests:
 - a. **Run from the iiq console:** rule "SST Unit Test All Rule" (See note on running individual rules below)

```
aduntee-mbpr:bin andy.duntee$ ./iiq console
2016-01-12 09:45:26,255 ERROR main sailpoint
> rule "SST Unit Test All Rule"
2016-01-12 09:45:47,823 DEBUG main rule.SST.
2016-01-12 09:45:47,828 DEBUG main rule.SST.
2016-01-12 09:45:47,834 DEBUG main rule.SST.
2016-01-12 09:45:47,842 DEBUG main rule.SST.
```

Figure 9 - Run from console

...bunch of output...

```
2016-01-12 09:45:47,903 DEBUG main rule.SST.UnitTest.RulesLibrary:? - SEND THE RESULTS TO SYSTEM.OUT
2016-01-12 09:45:47,903 DEBUG main rule.SST.UnitTest.RulesLibrary:? - {message=OBJECT NOT FOUND, THIS UNIT TEST RULE HASN'T B
EEN CODED YET., bgColor=#FF9900, category=Miscellaneous, useCase=N/A, argFile=N/A, status=FAILURE, ruleName=NON-EXISTENT RULE
ENTRY)
2016-01-12 09:45:47,903 DEBUG main rule.SST.UnitTest.RulesLibrary:? - {message=__, bgColor=#73B873, category=Miscellaneous, us
eCase=Simple Return Test Value, argFile=Sample/SST_SampleUnitTest_Args1.xml, status=SUCCESS, ruleName=SST Sample Unit Test Rul
e)
2016-01-12 09:45:47,903 DEBUG main rule.SST.UnitTest.RulesLibrary:? - {message=OBJECT NOT FOUND, THIS UNIT TEST ARG FILE HASN
'T BEEN CODED YET., bgColor=#FF9900, category=Miscellaneous, useCase=N/A, argFile=Sample/SST_SampleUnitTest_Args2NOTEXIST5.xml
, status=FAILURE, ruleName=SST Sample Unit Test Rule}
finished
> █
```

Figure 10 - View console results

The console results will return a map. In that map will be the rule being called, the arg file being called, the status, and any error message.

- b. **Run from the Dashboard:** Click dashboard link, SST Unit Test Reports. Click Run Tests button. Report will be a link at bottom of list, named based on the time run and the box the test was run on.

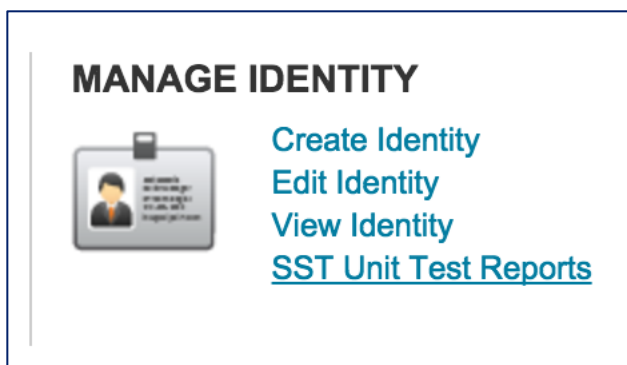


Figure 11 - Link to view/run reports from dashboard (6.4 view)

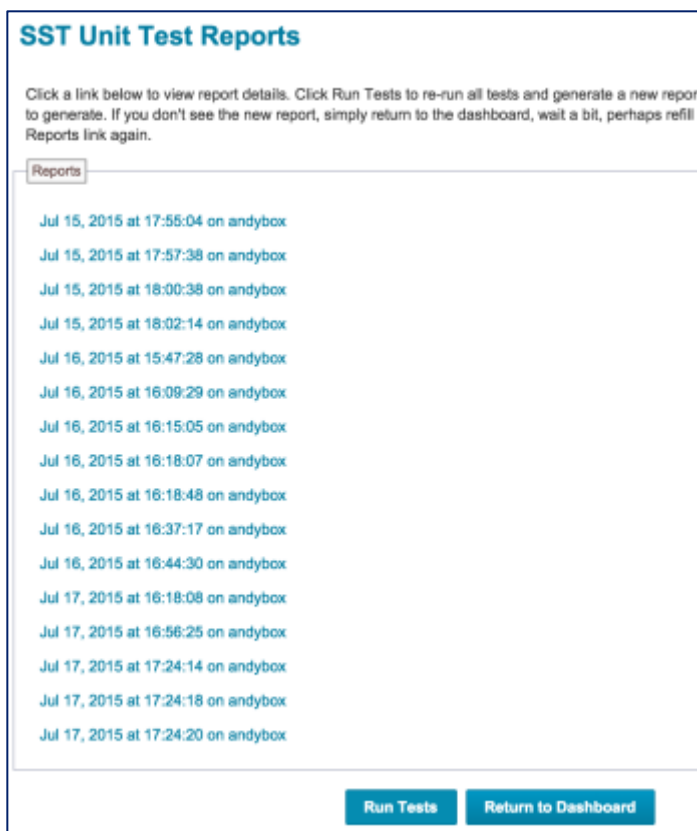


Figure 12 - Run Tests (6.4 view)

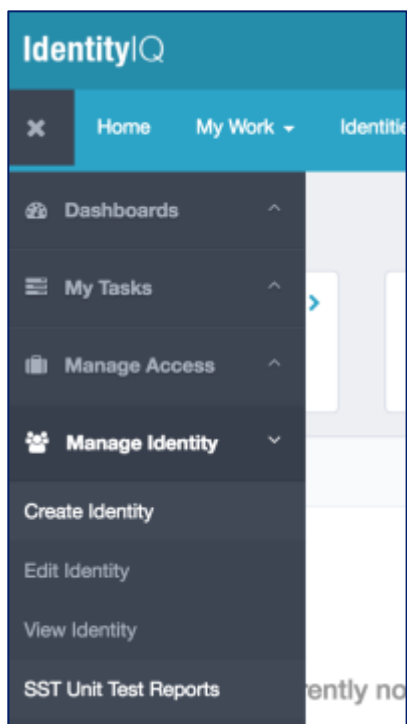


Figure 13 - Link to view/run reports from dashboard (7.x view)

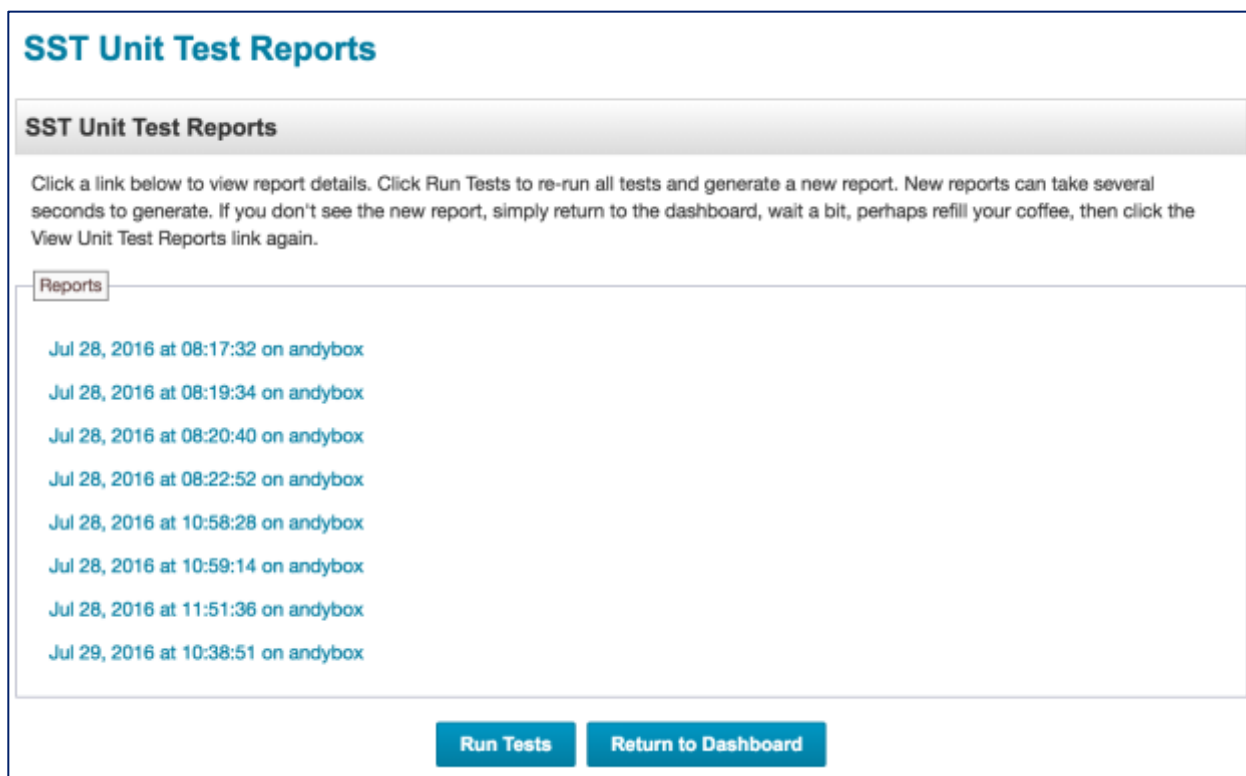


Figure 14 - Run Tests (7.x view)

- c. Run the task (task can also be scheduled): Go to Monitor>Tasks. Locate the task, SST Unit Tester All Rule. Click it. Confirm the rule selected and ruleConfig entry and click Save and Execute.

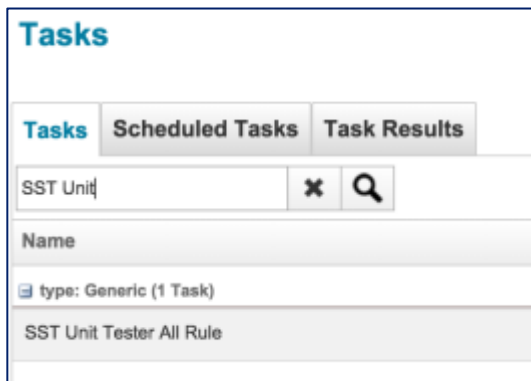


Figure 15 - Locate the task

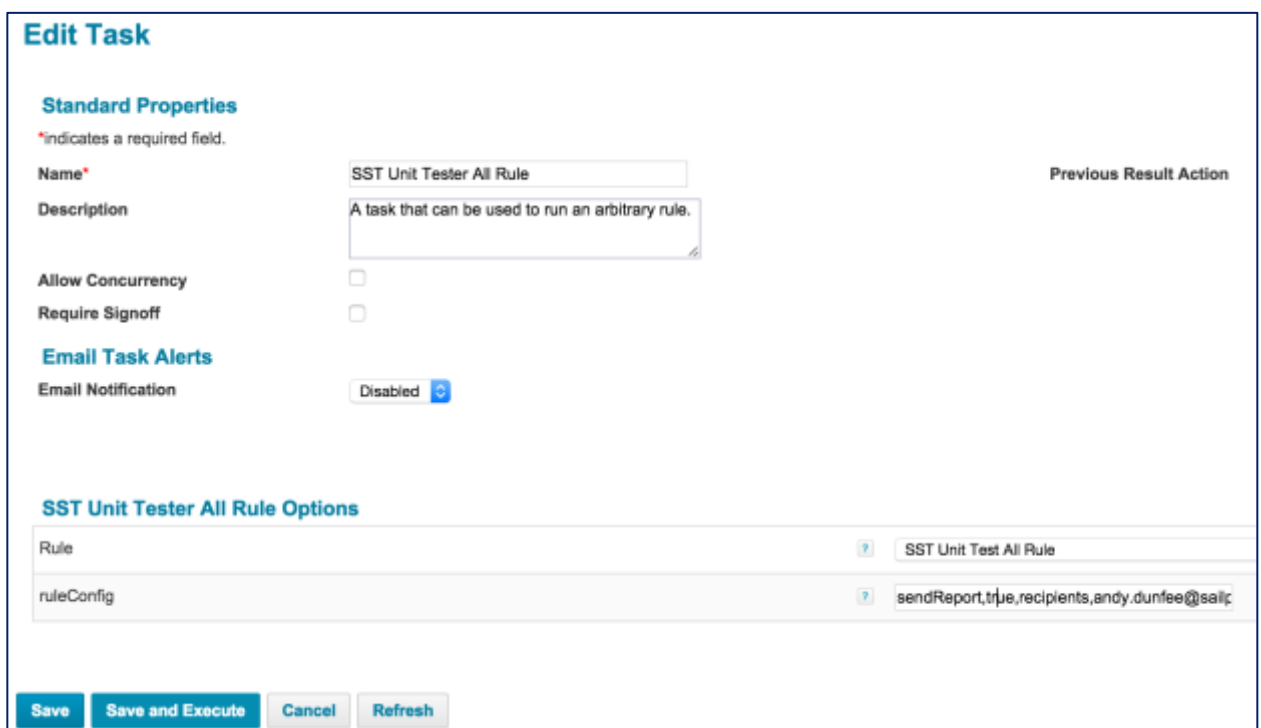


Figure 16 - Confirm and run task

6. Analyze the results. If running from console, see above on what to look for in console output. If running from the dashboard or from the task, an html report will be created. It will be available by going to the SST Unit Tester Reports link and clicking the link for the given report, named according to the time and machine where it was run. The report will display a summary of the number of tests run, the number successful and the number failed; numbers based on argument files, not rules. It will then show a more detailed breakdown of every test, including for each: the rule, argument file, use case, category, status (success/failure, and notes (error message or reason for failure).

SST Unit Tester Report

Summary

Total Run:	3
Total Success:	1
Total Failed:	2

Individual Results

Rule	Arg File	Use Case	Category	Status	Notes
NON-EXISTENT RULE ENTRY	N/A	N/A	Miscellaneous	FAILURE	OBJECT NOT FOUND, THIS UNIT TEST RULE HASN'T BEEN CODED YET.
SST Sample Unit Test Rule	Sample/SST_SampleUnitTest_Args1.xml	Simple Return Test Value	Miscellaneous	SUCCESS	—
SST Sample Unit Test Rule	Sample/SST_SampleUnitTest_Args2NOTEXISTS.xml	N/A	Miscellaneous	FAILURE	OBJECT NOT FOUND, THIS UNIT TEST ARG FILE HASN'T BEEN CODED YET.

Figure 17 - Unit Test Report

SST Unit Tester Report

Summary

Total Run:	2
Total Success:	2
Total Failed:	0

Individual Results

Rule	Arg File	Use Case	Category	Status	Notes
UT_getSomeDumbVal Rule	UT_getSomeDumbVal_Args-happy.xml	happy == smile	Miscellaneous	SUCCESS	—
UT_getSomeDumbVal Rule	UT_getSomeDumbVal_Args-sad.xml	sad == rain	Miscellaneous	SUCCESS	—

Figure 18 - Unit Test Report

Setup

The following details how to do the initial setup.

Inventory Contents

First, it is important to inventory everything associated with the tool. The main configurations are found under /config/SSF_Tools/SST_UnitTester. The main folders are: Configure, SampleInputs and Source.

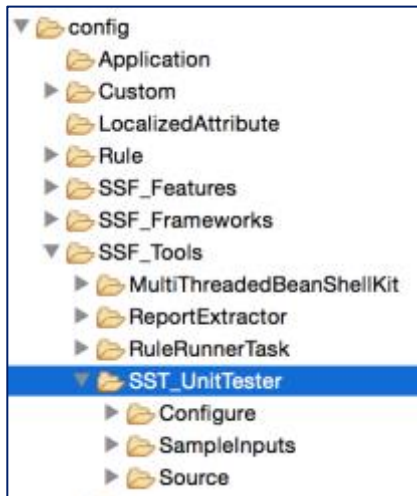


Figure 19 - Package contents

Each can be described as follows:

- **Configure** – the files/folders that need to be moved and updated
- **SampleInputs** – example tester rules and arguments files
- **Source** – the underlying code that need not be touched

Additionally, there are target and ignorefiles property entries in the unittester.target.properties and unittester.ignorefiles.properties files under SST_UnitTester. In the target.properties file for your environment, the following should be add and updated:

```
%%SST_VIEW_UNIT_TESTS_WF_TRACE%%=true
%%SST_UT_OUTPUT_PATH%%=/usr/local/tomcat/webapps/essd/UTReports/
%%SST_UT_OUTPUT_PATH_BUILD%%=/Users/andy.dunfee/Documents/workspace/ExplodeSSD/web/UTReports/
%%SST_RUN_ALL_UT_RECIPIENTS%%=andy.dunfee@sailpoint.com
%%SST_IIQ_INSTANCE_NAME%%=essd
%%SST_UT_ARGS_PATH%%=/Users/andy.dunfee/Documents/workspace/ExplodeSSD/build/extract/WEB-INF/config/custom/ZUnitTests/Args/
%%SST_UT_TARGET%%=andybox
%%FILE_SEPARATOR%%=/
```

In the ignorefiles.properties for your environment, the following should be available:

custom/SSF_Tools/SST_UnitTester/SampleInputs/SST_SampleUnitTest_Args1.xml
 custom/SSF_Tools/SST_UnitTester/SampleInputs/SST_SampleUnitTest_Rule.xml
 custom/SSF_Tools/SST_UnitTester/Configure/SST_UnitTest_Mappings_Custom.xml

Note that if you are using the SSD Deployer tool the contents of the `unittester.target.properties` and `unittester.ignorefiles.properties` will be automatically copied over to the properties files for your environment.

As a sanity check, review the file `<Project Root>/scripts/build.config.xml` and validate it includes the custom exclusions for unit test objects.

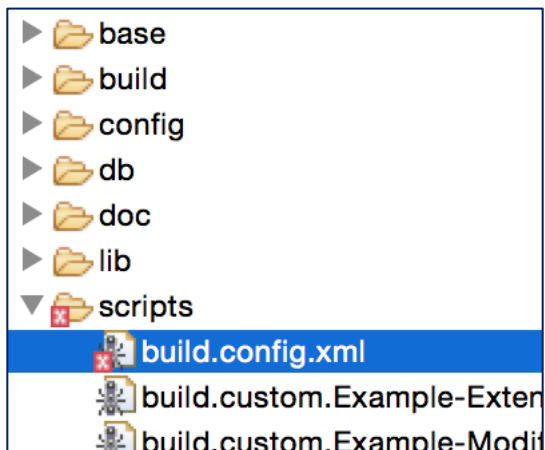


Figure 20 - Locate build.config.xml

```

64  <target name="build-inits" depends="validate">
65    <build-init initFile="${build.iqBinaryExtract}/WEB-INF/config/sp.init-custom.xml">
66      <fileset dir="${build.iqBinaryExtract}/WEB-INF/config">
67        <include name="custom/**/*.xml"/>
68        <include name="custom/**/*.jspxml"/>
69        <exclude name="custom/**/*-init.xml"/>
70        <exclude name="custom/**/*.hbm.xml"/>
71        <exclude name="custom/**/*.cert.xml"/>
72        <exclude name="custom/**/*.template/**/*.xml"/>
73        <!-- Do not move over the template xml into the build dir -->
74        <exclude name="custom/**/*.Template.xml"/>
75        <!-- Exclude items named in this target's excludes file: -->
76        <excludesfile name="${target}.ignorefiles.properties" if="ignorefiles.is.found"/>
77
78        <!-- A.DUNFEE = - CUSTOM EXCLUDE FOR UNIT TEST OBJECTS -->
79        <exclude name="custom/**/*.Args/**"/>
80        <exclude name="custom/**/*.Args*"/>
81      </fileset>
82    </build-init>
  
```

Figure 21 - Validate Args folder is excluded

```
164<target name="validate" depends="-strip-ids">
165    <echo message="Validating Custom XML objects"/>
166    <echo message="${build.customXMLDir}"/>
167    <xmlvalidate>
168        <!--
169        <fileset dir="${build.customXMLDir}" includes="**/*.xml" />
170        -->
171    <fileset dir="${build.customXMLDir}">
172        <include name="**/*.xml"/>
173
174        <!-- A.DUNFEE = - CUSTOM EXCLUDE FOR UNIT TEST OBJECTS -->
175        <exclude name="**/Args/**"/>
176        <exclude name="**/*Args*"/>
177
178    </fileset>
179    <xmlcatalog>
180        <dtd publicId="sailpoint.dtd" location="${dtd}"/>
181    </xmlcatalog>
182    </xmlvalidate>
183</target>
```

Figure 22 - Validate Args folder is excluded

Initial Setup Steps

The Unit Tester can be set up with the SSD Deployer tool (see the SSD Deployer User Guide that ships with the SSD) or manually. First you should review the inventory and ensure everything is in place (see above). Then follow these steps (if you are using the SSD Deployer it will automate steps 1 to 5):

1. In the build.properties file, set 'deploySSTUnitTester=true'
2. Under config, create or validate existence of folder called "Custom"

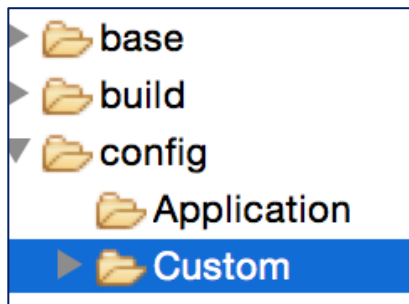


Figure 23 - Create /config/Custom folder

3. Copy the file /config/SSF_Tools/SST_UnitTester/Configure/SST_UnitTest_Mappings_Custom.xml to the folder /config/Custom

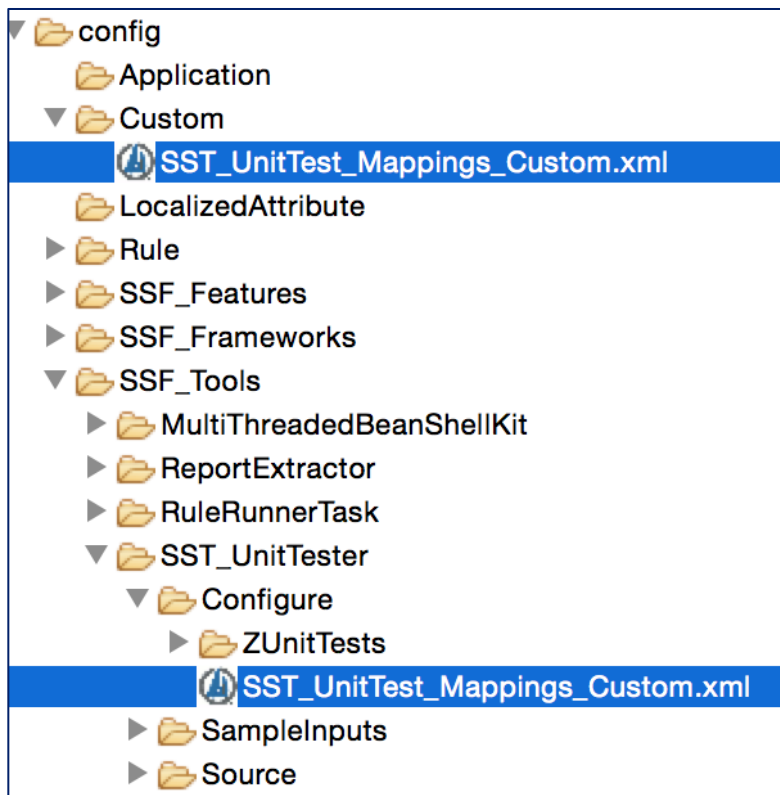


Figure 24 - Copy/paste custom mapping object to /config/Custom folder

4. Copy the folder config/SSF_Tools/SST_UnitTester/Configure/ZUnitTests to the /config folder

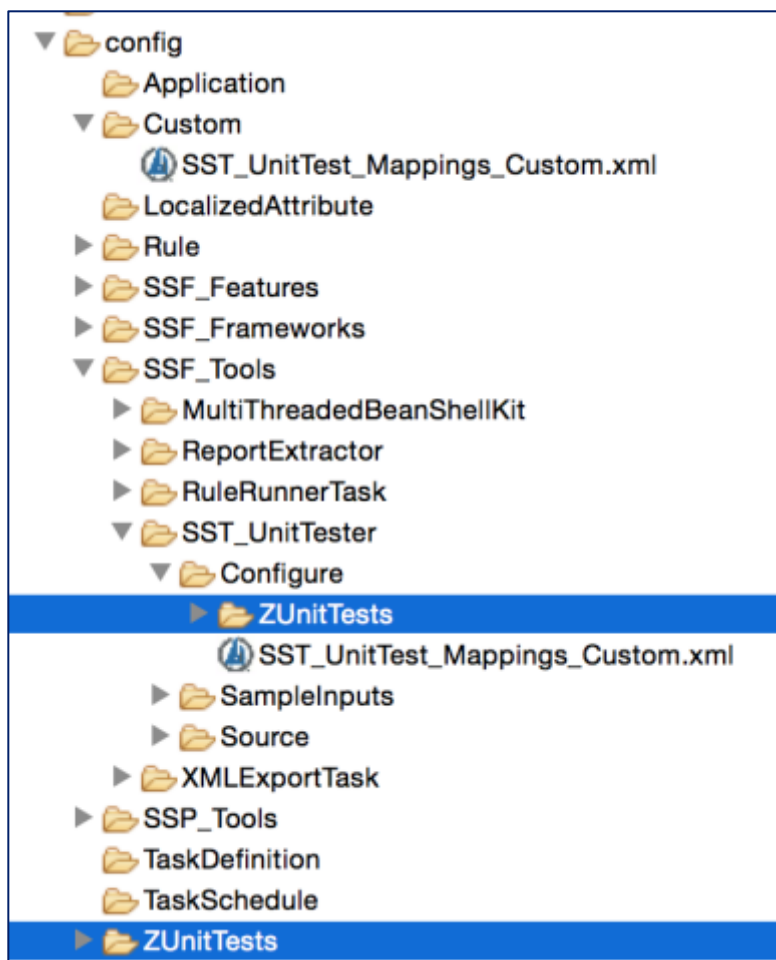


Figure 25 - Copy/paste ZUnitTests folder to /config

5. Under <Project Root>/web, create a folder called UTReports

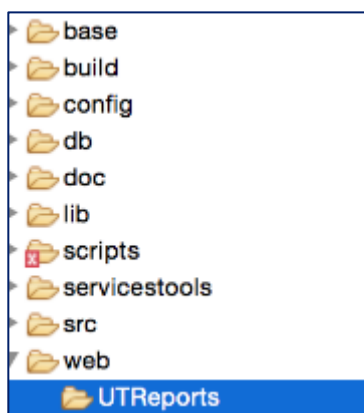


Figure 26 - Create UTReports folder in build for html output

6. If you've already deployed a WAR to your app server, you may want to also create this folder underneath the application in your app server

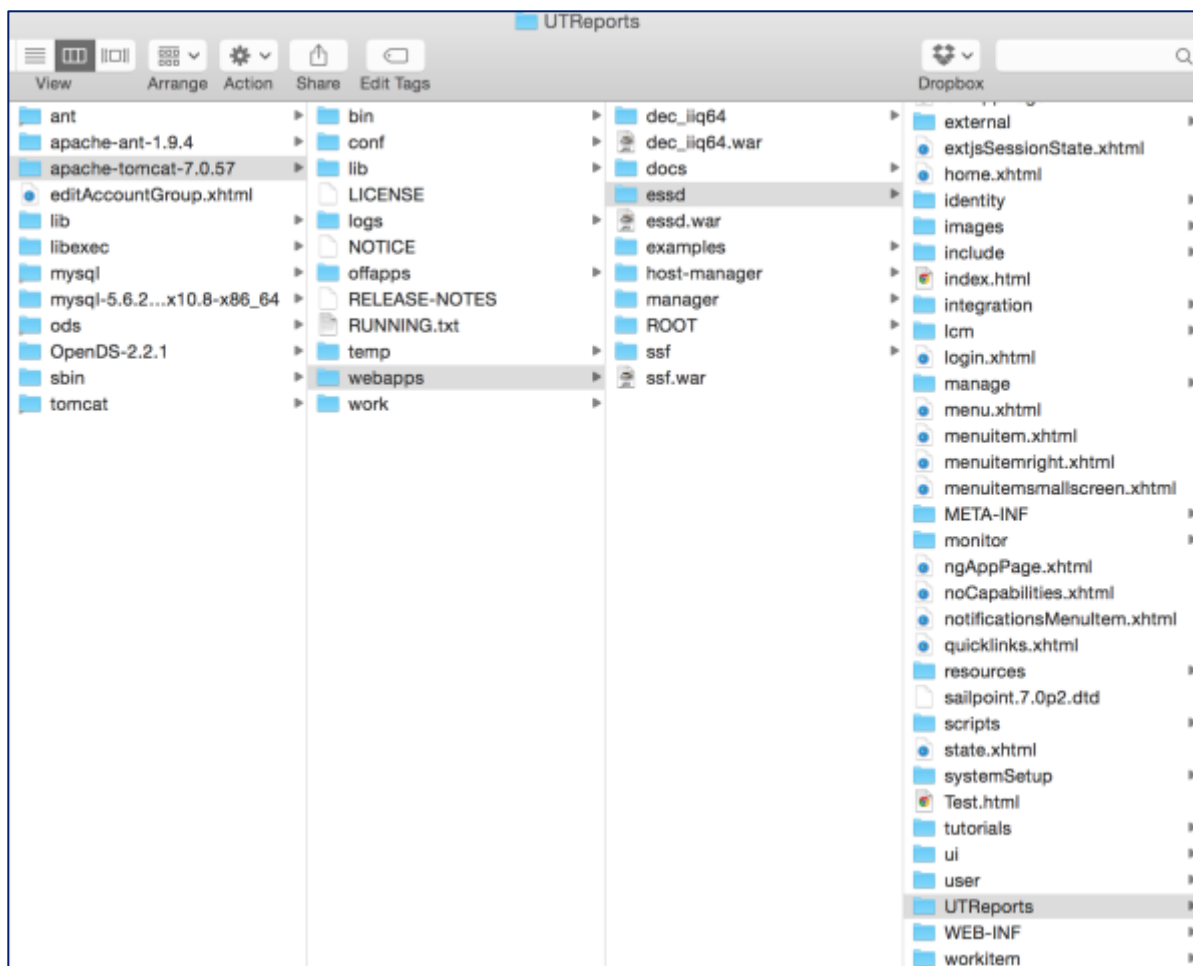


Figure 27 - Create UTRports folder in deployed application for html output

7. Configure the target properties (see Target Properties below).

```
#####
# SST UNIT TESTER
#####
%%SST_VIEW_UNIT_TESTS_WF_TRACE%%=true
%%SST_UT_OUTPUT_PATH%%=/usr/local/tomcat/webapps/essd/UTReports/
%%SST_UT_OUTPUT_PATH_BUILD%%=/Users/andy.dunfee/Documents/workspace/ExplodeSSD/web/UTReports/
%%SST_RUN_ALL_UT_RECIPIENTS%%=andy.dunfee@sailpoint.com
%%SST_IIQ_INSTANCE_NAME%%=essd
%%SST_UT_ARGS_PATH%%=/Users/andy.dunfee/Documents/workspace/ExplodeSSD/build/extract/WEB-INF/config/custom/ZUnitTests/Args/
%%SST_UT_TARGET%%=andybox
%%FILE_SEPARATOR%%=/
```

Figure 28 - Update target.properties

8. Configure your first test (see Configuring Your First Test)

Target Properties

The list below describes the target properties used by the SST Unit Tester. In the given example environment, the following paths are used:

Project Root: /Users/andy.dunfee/Documents/workspace/ExplodeSSD

Deployed Application Root: /usr/local/tomcat/webapps/essd

The following details each property:

SST_VIEW_UNIT_TESTS_WF_TRACE

Used to turn on/off trace in the SST Unit Test WF. The possible values are true or false.

This workflow is available from the dashboard via a quick link and allows for running the full set of unit tests as well as viewing all of the existing unit test reports.

Example:

```
%%SST_VIEW_UNIT_TESTS_WF_TRACE%%=true
```

SST_UT_OUTPUT_PATH

Used to set the path where reports will be written when running the tests from the dashboard or task definition. This path should be within the context of your deployed application.

Example:

```
%%SST_UT_OUTPUT_PATH%%=/usr/local/tomcat/webapps/essd/UTReports/
```

SST_UT_OUTPUT_PATH_BUILD

Used to set the path where reports will be written when running the tests from the dashboard or task definition. This property allows for setting a path in your local build so that if you build and re-deploy the war, your past tests aren't lost.

Example:

```
%%SST_UT_OUTPUT_PATH_BUILD%%=/Users/andy.dunfee/Documents/workspace/ExplodeSSD/web/UTReports/
```

SST_RUN_ALL_UT_RECIPIENTS

Used to set a recipient email address that will receive an email containing the latest test run. This would really only be necessary if running in a dev/test environment configured to talk to an SMTP server.

Example:

```
%%SST_RUN_ALL_UT_RECIPIENTS%%=andy.dunfee@sailpoint.com
```

SST_IIQ_INSTANCE_NAME

Used to map the web location of each individual file. Should match the name of the application instance (or deployed .war file's name, minus '.war').

Example:

```
%%SST_IIQ_INSTANCE_NAME%%=essd
```

SST_UT_ARGS_PATH

Used to set the file location of the argument files used for each unit test case. The key to this property is that it should be set to the destination folder of the arguments AFTER a build is run. If the location in the build is <Project Root>/config/ZUnitTests/Args, the value of this property should be <Project Root>/build/extract/WEB-INF/config/custom/ZUnitTests/Args/.

The reason for this distinction is that this file location allows for the use of target properties in argument files. (NOTE: THIS IS ALSO THE REASON FOR THE EXCLUSION EDITS IN THE BUILD.CONFIG.XML FILE IN THE SSB)

Example:

```
%%SST_UT_ARGS_PATH%%=/Users/andy.dunfee/Documents/workspace/ExplodeSSD/build/extract/WEB-INF/config/custom/ZUnitTests/Args/
```

SST_UT_TARGET

Used in the name of each html report that is created to denote which environment the test was run on.

Example:

```
%%SST_UT_TARGET%%=andybox
```

FILE_SEPARATOR

Used to denote which way your environment's slashes go. In all beanshell, the code more likely uses the java call to get the underlying separator. What this is really for is adding sub paths to the Args folder and being able to add an environment-agnostic separator in the custom mapping object.

Example:

%%FILE_SEPARATOR%%= /

Configuring Your First Test

The following walks through an example to show how to setup your first test.

In this example, we've written a rule library called ESSD Tester Rules Library, stored in the file location /config/Rule/ESSD_Tester_RulesLibrary.xml.

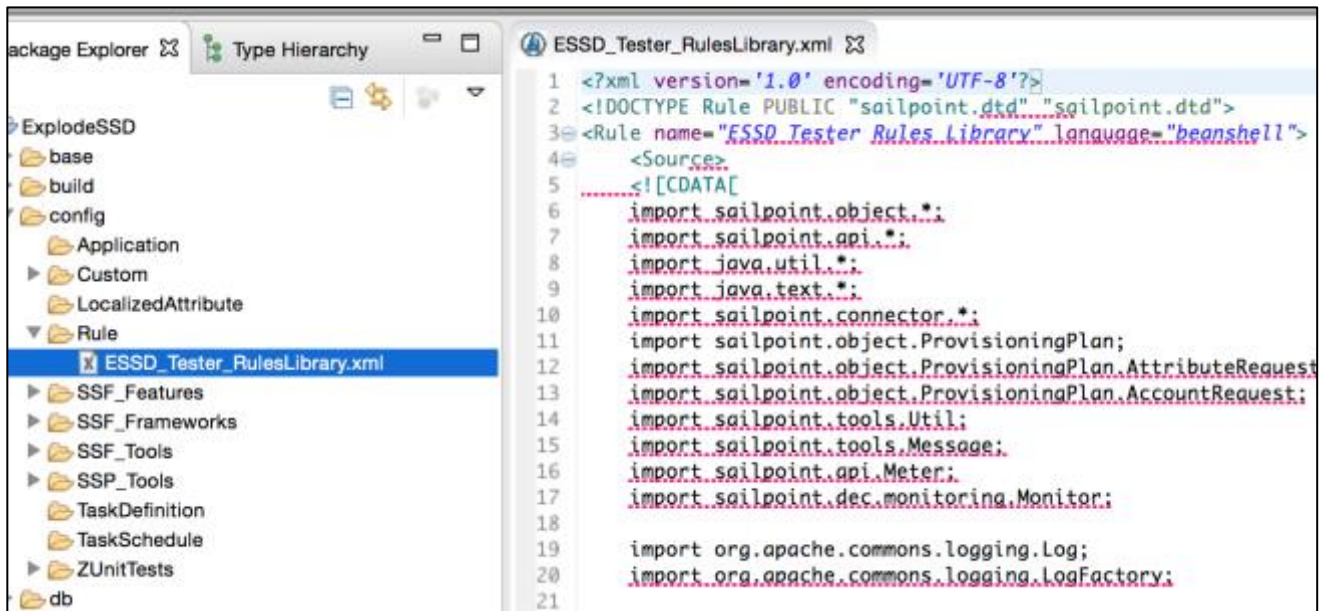


Figure 29 - New library with code to test

In this library, we've written a method. In this case, it's a rather dumb, simple method that accepts one input, a String, and uses that to do a get against a statically coded HashMap and one return value. For instance, if the input is "happy", the return value will be "smile".

```
public String getSomeDumbVal(String input){
    gglogger.trace("Enter getSomeDumbVal");
    String retVal = "NOMATCH";

    if (input ==null){
        retVal = "NULLNOGOOD";
        gglogger.trace("Exit getSomeDumbVal: " + retVal);
        return retVal;
    }

    Map someMap = new HashMap();
    someMap.put("scary", "boo");
    someMap.put("happy", "smile");
    someMap.put("sad", "clown");

    if (someMap.containsKey(input)){
        retVal = someMap.get(input);
    }

    gglogger.trace("Exit getSomeDumbVal: " + retVal);
    return retVal;
}
```

Figure 30 - Example method to test

The easiest place to start to write a unit test is to go to the SampleInputs folder under /config/SSF_Tools/SST_UnitTester:

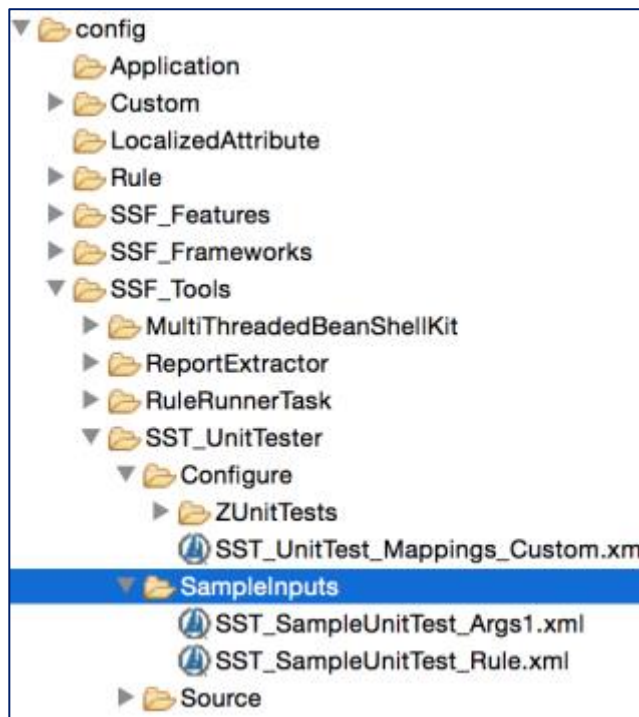


Figure 31 - Sample rules and args files

Copy the rule file to /config/ZUnitTests/Rules and rename it to something meaningful. In this case, we've named it UT_getSomeDumbVal_Rule.xml. The logic here is: "UT" clearly identifies it as a unit

test, “getSomeDumbVal” denotes which method is being test, and “Rule” identities the file content’s object type.

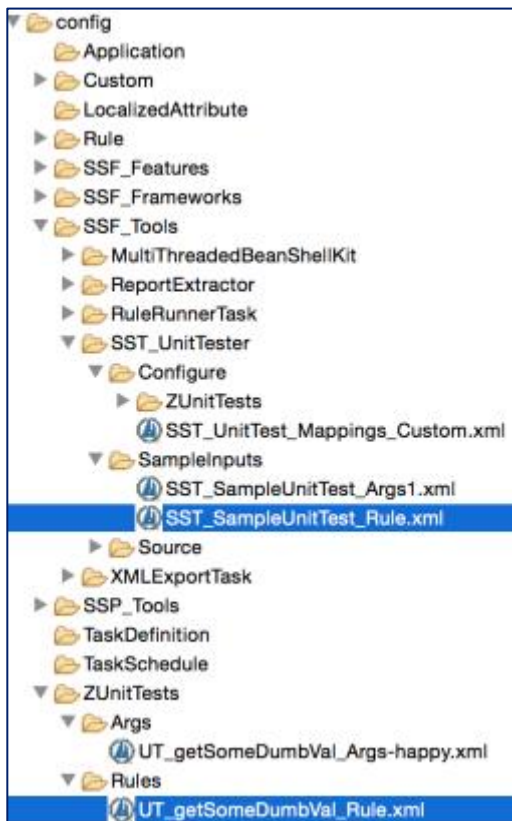


Figure 32 - Copy the sample rule

Edit the tester rule and make the required changes:

- Change the name of the object. It should roughly match the file name and it is recommended to use the same logic of making it obvious it’s a unit test, denoting what method/thing you are testing, and denoting that it’s a rule. In the given example, we’ve chosen the name *UT getSomeDumbVal Rule*:

```
<Rule name="UT getSomeDumbVal Rule" language="beanshell">
```

- Add the rule reference to the library of the method you want to test. In this case, we added the following library reference:

```
<ReferencedRules>
<Reference class="sailpoint.object.Rule" name="ESSD Tester Rules Library"/>
</ReferencedRules>
```

- Add each input argument that is required by the method you want to test. In this case, the method accepts one input string, therefore the rule must also accept one input string. The

name of the argument in the rule must match the name of the argument passed into the method call.

Input args:

```
<Signature returnType="Map">
  <Inputs>
    <Argument name="argsFileName"/>
    <Argument name="inputValue"/> <!-- Main input -->
    <Argument name="expectedResult"/>
  </Inputs>
  <Returns/>
</Signature>
```

Method call:

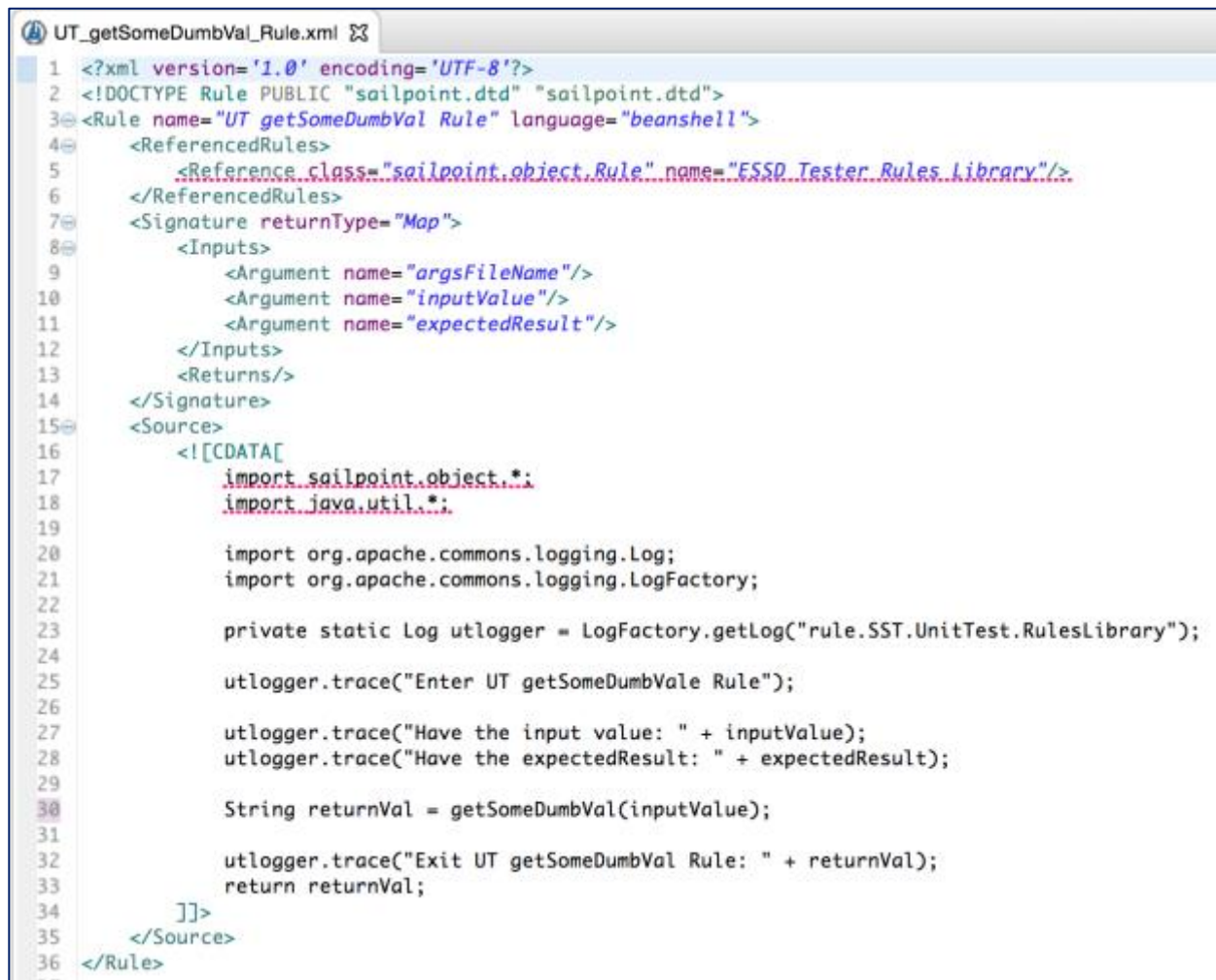
```
String returnVal = getSomeDumbVal(inputValue);
```

- Additionally, it's often helpful to pass in two other arguments, so that you can log them in the tester rule:
 - argsFileName – this is something that can be customized in the args file to denote what test you are running
 - expectedResult – the value you expect to get back from the method call
- Write the test rule. The contents of the rule can be complex if there are things that need to be done to setup additional test data or convert inputs into specific arguments for your given test, and it is recommended to wrap logic in clear log statements, but often the logic can be as simple as:

```
Object returnVal = yourMethodCall(yourInputValues);

return returnVal;
```

The following shows our full example rule:



```

1 <?xml version='1.0' encoding='UTF-8'?>
2 <!DOCTYPE Rule PUBLIC "sailpoint.dtd" "sailpoint.dtd">
3 <Rule name="UT_getSomeDumbVal Rule" language="beanshell">
4   <ReferencedRules>
5     <Reference class="sailpoint.object.Rule" name="ESSD Tester Rules Library"/>
6   </ReferencedRules>
7   <Signature returnType="Map">
8     <Inputs>
9       <Argument name="argsFileName"/>
10      <Argument name="inputValue"/>
11      <Argument name="expectedResult"/>
12    </Inputs>
13    <Returns/>
14  </Signature>
15  <Source>
16    <![CDATA[
17      import sailpoint.object.*;
18      import java.util.*;
19
20      import org.apache.commons.logging.Log;
21      import org.apache.commons.logging.LogFactory;
22
23      private static Log utlogger = LogFactory.getLog("rule.SST.UnitTest.RulesLibrary");
24
25      utlogger.trace("Enter UT_getSomeDumbVal Rule");
26
27      utlogger.trace("Have the input value: " + inputValue);
28      utlogger.trace("Have the expectedResult: " + expectedResult);
29
30      String returnVal = getSomeDumbVal(inputValue);
31
32      utlogger.trace("Exit UT_getSomeDumbVal Rule: " + returnVal);
33      return returnVal;
34    ]]>
35  </Source>
36 </Rule>

```

Figure 33 - Example unit test rule

After the unit test rule is created, it's time to create some argument files. Each file should represent a unique test case, each containing a different input value or set of input arguments. Similar to the test rule, the best place to start is in the SampleInputs folder (see above) and the best first step is to copy the sample args file.

In this case, copy the file SST_SampleUnitTest_Args1.xml into the folder /config/ZUnitTests/Args. Rename it to something meaningful. The recommendation is to start with the same name as the tester rule, but then change "Rule" to "Args" and then add something brief to denote the specific use case. For example, UT_getSomeDumbVal_Args-happy.xml or UT_getSomeDumbVal_Args-sad.xml.

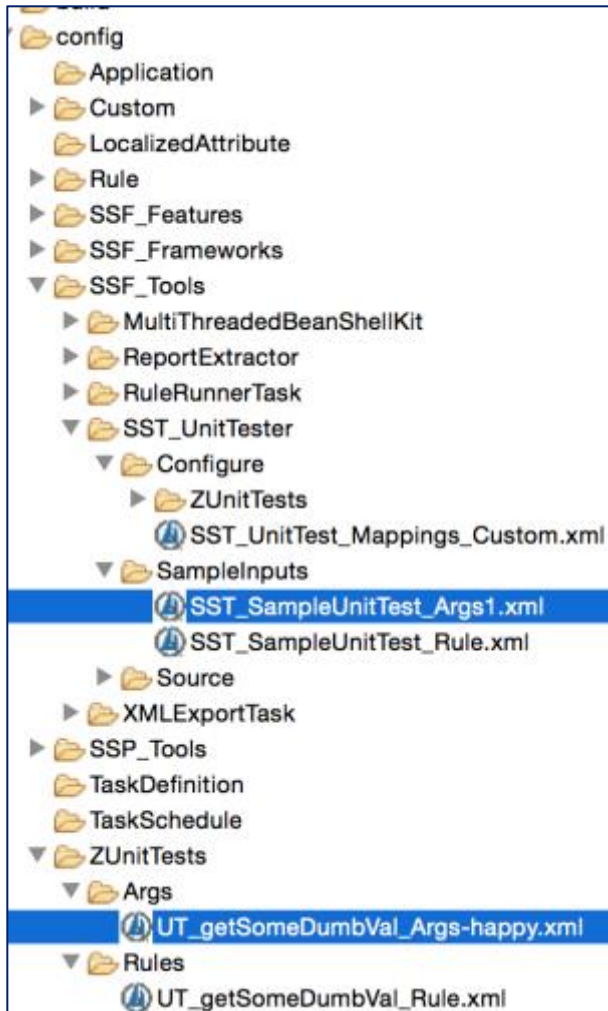


Figure 34 - Copy, paste and rename the arguments file

Edit the new args file. There are three common entries:

1. **argsFileName** – This denotes the name of the file being tested. Its usage is mostly informational in the report output
2. **useCase** – Similarly, this is informational for denoting what use case is being run, what is expected in terms of input and output (keep it short though)
3. **expectedResult** – This is REQUIRED and is the object that will be compared to the return value of the tester rule. This object must be equal to the return value in order for the test to be considered a success

All other entries are optional, in terms of the underlying test framework. However, all other inputs must match the inputs of the method or thing you want to test, or at least must match those things that are unique to your test; it is certainly possible and okay to hard-code values that are the same for every test.

In our example, we have a method with one input value. We have written the test rule to have the input argument called “inputValue”. Therefore, the args file needs to contain an entry called “inputValue”. This is our mechanism for creating multiple test cases.

The following shows what our arg file ends up looking like:

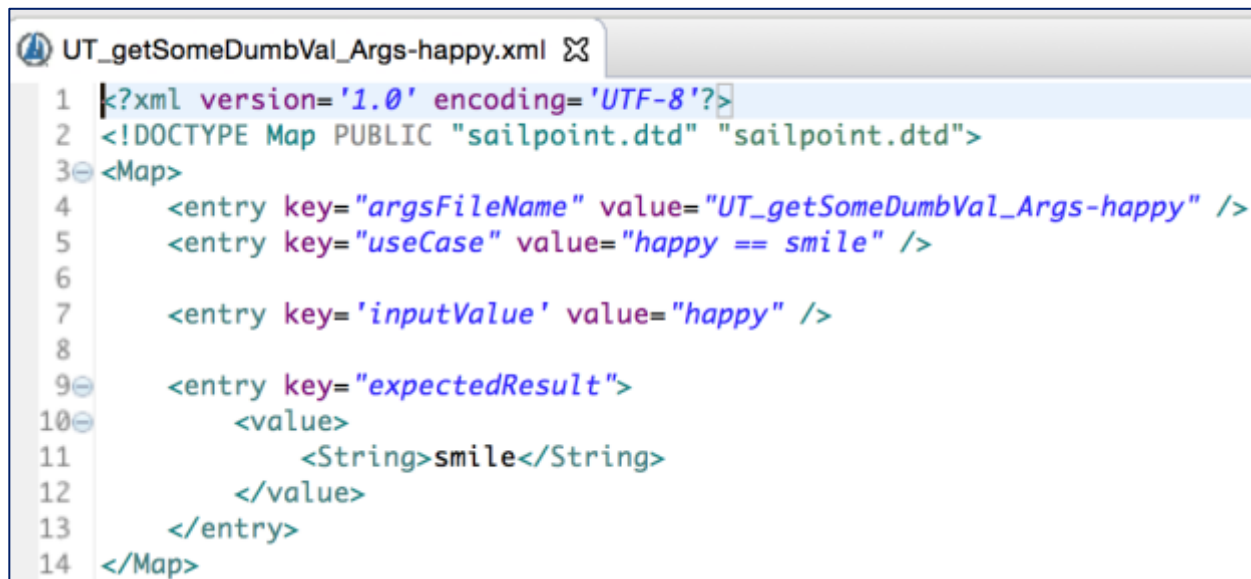


Figure 35 - Sample args file

Once the rule and args file(s) are created, the final step is to register the rule and arg files. Locate the mapping object at /config/Custom/SST_UnitTest_Mappings_Custom.xml (you should've copied this here as part of the initial setup).

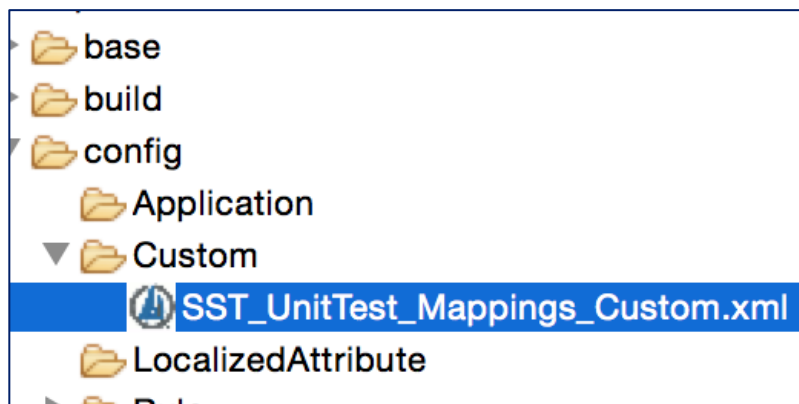


Figure 36 - Locate the unit test mappings custom object

Edit this file. Add an entry for the new test rule. Within that entry, add the list of argument files for each test case. The following should be noted:

- The entry key is the name of the rule object (not the rule file)
- Each args entry is the name of the file (not the name of the object) and should utilize the path found in the target property %%SST_UT_ARGS_PATH%%
- Enter an appropriate category based on the Categories entry at the top (this is editable, btw)
- Set Enabled to true if you want this test to run and false if you don't

The following shows our example's entry:

```
<entry key="Rules">
  <value>
    <Attributes>
      <Map>
        <entry key="UT_getSomeDumbVal Rule">
          <value>
            <Attributes>
              <Map>
                <entry key="Args">
                  <value>
                    <List>
                      <String>XXSST_UT_ARGS_PATHXXUT_getSomeDumbVal_Args-happy.xml</String>
                    </List>
                  </value>
                </entry>
                <entry key="Category" value="Miscellaneous" />
                <entry key="Enabled" value="true" />
              </Map>
            </Attributes>
          </value>
        </entry>
      </entry>
    </value>
  </entry>
```

Figure 37 - Register your test rule and use cases

THAT'S IT!

You've created your first test rule and use case. Run and deploy the build. Run the tester (see Usage). After you run, you should be able to see the result in the report:

SST Unit Tester Report

Summary

Total Run:	1
Total Success:	1
Total Failed:	0

Individual Results

Rule	Arg File	Use Case	Category	Status
UT_getSomeDumbVal Rule	UT_getSomeDumbVal_Args-happy.xml	happy == smile	Miscellaneous	SUCCESS

Figure 38 - First run of the report

Once you have the first rule and args file in place, adding more tests is fairly easy. Simply copy the args file you just created, giving it a new name. Change the input value and expected result and add the new file path to the custom mapping object.

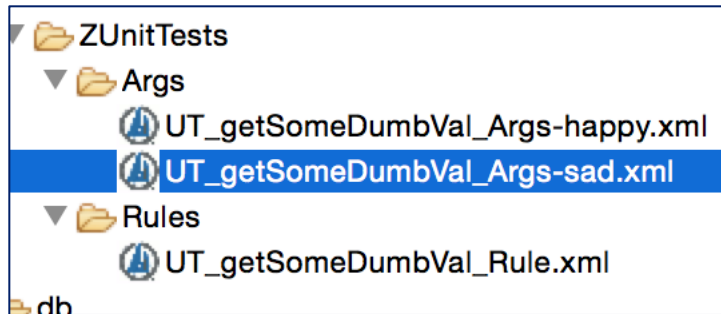


Figure 39 - Copy and rename for new test case



Figure 40 - Update the args file appropriately



Figure 41 - Register new args file

SST Unit Tester Report

Summary

Total Run:	2
Total Success:	1
Total Failed:	1

Individual Results

Rule	Arg File	Use Case	Category	Status	Notes
UT_getSomeDumbVal Rule	UT_getSomeDumbVal_Args-happy.xml	happy == smile	Miscellaneous	SUCCESS	—
UT_getSomeDumbVal Rule	UT_getSomeDumbVal_Args-sad.xml	sad == rain	Miscellaneous	FAILURE	EXPECTED RESULT == rain. ACTUAL RESULT == clown.

Figure 42 - Review results

In the example above, the expected result entered was intentionally made to be incorrect to show a failure. The developer now would need to examine which is incorrect, the args file or the logic being tested.

After updating the mapping in the method to match the test case, we rerun the unit tester and get two successes:

SST Unit Tester Report

Summary

Total Run:	2
Total Success:	2
Total Failed:	0

Individual Results

Rule	Arg File	Use Case	Category	Status	Notes
UT_getSomeDumbVal Rule	UT_getSomeDumbVal_Args-happy.xml	happy == smile	Miscellaneous	SUCCESS	—
UT_getSomeDumbVal Rule	UT_getSomeDumbVal_Args-sad.xml	sad == rain	Miscellaneous	SUCCESS	—

Figure 43 - Review results again

Now, that's progress.

Advanced Topics

All of the above stuff was pretty simple. What about some more complex scenarios. What about void methods? What about methods that update objects in the repository? What if I want to test things other than methods in libraries? What if I'm testing the update of something, don't I need to revert the something to its original state so the test can be repeated?

Can the SST Unit Tester really be used for these scenarios?

Yes, but...

You, the developer, are responsible for how and what you write in your unit test rule and you, the developer, simply have to be careful and cover your tracks.

But we can give you some tips...

Void Methods

Void methods don't return anything so how do you compare nothing to some sort of expected result?

This requires creativity and not every solution is going to be the same. Generally, a void method is going to do one of two things:

1. Update something somewhere: setting an attribute or many attributes on an identity cube, writing to an external database, etc.
2. Update one of the input arguments to the method or thing you're calling

In these scenarios, the unit test rule must now do extra stuff after calling the thing you are testing and, often in these cases, it must do some initialization before calling the thing you are testing.

The most common example is doing some sort of update to an identity cube. Thus, after calling your method, the test rule would need to get the object you just updated and the value of the thing you updated and use that to set the return value.

Likewise, if the void method is updating one of the input arguments, simply use the attribute, after calling your method, to get the thing that was updated and use that to set the return value.

Multiple Updates

But what if I'm updating multiple attributes?

Get both or all of them and concatenate as a string.

Return the entire object you are updating? The expected result doesn't have to be a String. It can be any object.

Repeatability

Ok, but if I'm updating something, how do I rerun the same test with the same starting point?

Here's another thing to note: the inputs don't need to be just Strings either. You can pass in any object. Even an Identity object.

Putting these things together, we can now do some interesting things. Let's say we are testing a call to a void method that does multiple updates to an Identity object. The best approach is to do the following:

- Make one of the input arguments the identity in its before state. This can be a smaller cube, only containing the things you care about. If you're only updating identity attributes and have no need for any links, don't include any links (it's a unit test, right?)
- Make the expected result argument a map object of each value you are updating, with each key being something you are updating and each value being the expected result
- Ensure, based on the naming of the identity, that there's no way it is an identity that would already be in your repository
- In the tester rule, save the input identity to the repository
- Run your method
- Get the identity from the repository. This gives you the updated version.
- Get the values you expect to be updated and do a put for each into a return map object
- Delete the identity from the repository (this is your cleanup step so you can rerun)
- Return the map

Note: if the use of a map doesn't work because of the ordering of entries, another approach would be to simply create a concatenated string, such as: "key1=value1,key2=value2".

Test Other Things

What if I don't want to test a method?

The tester rule is plain old Beanshell, so you can write whatever you want.

Want to test a rule, use `context.runRule();`

Want to run a workflow, use the `Workflow`, `WorkflowLaunch` classes.

Want to run a task, use `TaskManager.run();`

The possibilities are only limited to your imagination. But be careful. And always know the SST Unit Tester isn't going to write your tests for you nor should it replace assembly testing, performance testing, or any other customer-led test. It is a simple framework to better enable you to pound on individual pieces of code. Use your discretion as to where this fits into your build and test process.