MuleSoft MUnit Guidelines and Best Practices

**Version 0.1**

Eldee Khowong

Accenture Federal Services

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Author | Description | Version |
| 03/15/2019 | Khowong, Eldee | Initial draft | 0.1 |

Table of Contents

[Objectives 4](#_Toc3764846)

[1. Why is MUnit Important? 4](#_Toc3764847)

[2. Installing the MUnit Framework 5](#_Toc3764848)

[3. Creating and Structuring MUnit Tests 6](#_Toc3764849)

[3.1 Naming Convention 6](#_Toc3764850)

[3.2 Creating MUnit Tests in Anypoint Studio 6](#_Toc3764851)

[4. MUnit Test Suite Components 8](#_Toc3764852)

[4.1 Imports 8](#_Toc3764853)

[4.2 Bean Definitions 8](#_Toc3764854)

[4.3 Before/After Suites 8](#_Toc3764855)

[4.4 Before/After Tests 9](#_Toc3764856)

[4.5 MUnit Tests 10](#_Toc3764857)

[5. MUnit Test Components 12](#_Toc3764858)

[5.1 Mock Message Processors 12](#_Toc3764859)

[5.2 Spy Message Processor 13](#_Toc3764860)

[5.3 Verify Event Processor 14](#_Toc3764861)

[5.4 Assert Message Processors 14](#_Toc3764862)

[6. Executing MUnit Tests 16](#_Toc3764863)

[6.1 Executing MUnit Tests from Anypoint Studio 16](#_Toc3764864)

[6.2 Configuring MUnit for Maven 17](#_Toc3764865)

[6.3 Executing MUnit Tests via Maven 18](#_Toc3764866)

[7. Test Coverage 20](#_Toc3764867)

[7.1 Defining Test Coverage Metric 20](#_Toc3764868)

[7.2 Test Coverage Reporting in Maven 20](#_Toc3764869)

[7.3 Enforcing Test Coverage in Maven 23](#_Toc3764870)

[8. MUnit Support for Domain Projects 25](#_Toc3764871)

[9. Tips for Writing Great MUnit Tests 26](#_Toc3764872)

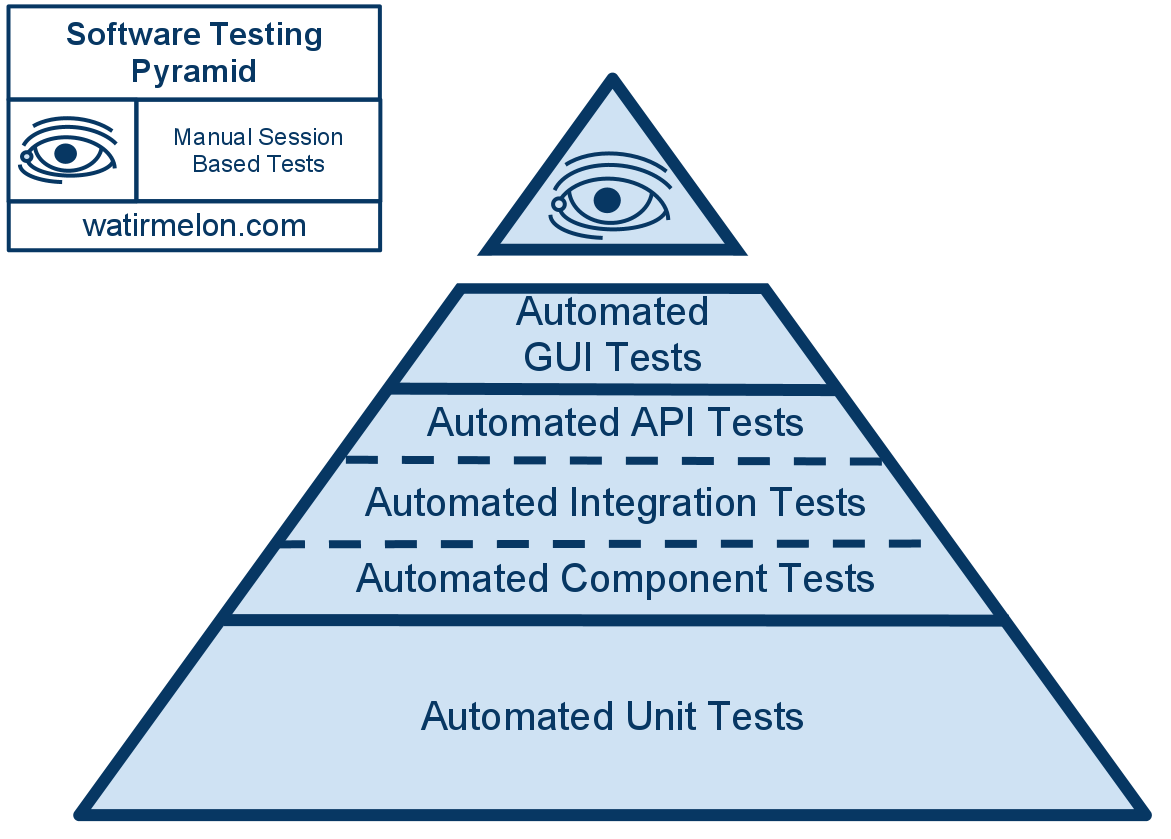
[10. Summary of Best Practices 27](#_Toc3764873)

# Objectives

MUnit is MuleSoft’s application testing framework that allows you to easily build automated unit tests for your integrations and APIs. It provides a full suite of integration and unit test capabilitiesand is fully integrated with Maven and Surefire for integration with continuous deployment environments.

This document provides a high-level overview of the MUnit testing framework. It will help developers/analysts configure MUnit and build simple MUnit tests. It also includes information about enforcing code coverage and tips for writing effective test cases. This best practice document is meant for MUnit 1.3.x running on Mule 3.x.

# Why is MUnit Important?



The software testing pyramid above illustratesthe value of automated testing. Unit tests are the most granular of your testing and make up the base of the testing pyramid.The more components that we cover with automated unit test cases, the less testing that needs to be done as you go up the pyramid. Unit tests are meant to validate the correctness of an individual unit of source code. For Mule, that translates to a Mule flow or sub-flow.

Unit testing should never be considered an afterthought. Anypoint Studio comes with a built-intesting framework that allows you to build MUnit test cases at the same time as you develop. This approach is called test driven development (TDD) and allows you to validate your requirements using unit tests.Some features of the MUnit testing framework include:

* Creating Mule test by writing Mule flows
* Mockingindividual outbound endpoints
* Mockingindividual message processors
* Spyingon message processor
* Verifying message processor calls
* Assertingresults
* Assertion/error reports with Mule stack trace
* Disabling/enabling particular tests
* Disabling/enabling flow inbound endpoints
* Disabling/enabling endpoint connectors
* Checking visual coverage in Anypoint Studio
* Reporting code coverage metrics
* Debugging tests with AnypointStudio
* Executing tests from Anypoint Studio
* Executing tests from Maven or a CI environment
* Failing builds when unit test cases fail, or code coverage is insufficient

# Installing the MUnit Framework

The MUnit framework comes pre-packaged with Anypoint Studio. However, you should always make sure that you have the most recent version of the testing framework installed. You can do this by adding the MUnit update site to the list of sites to check for updates and by installing any updates in Anypoint Studio.

A screenshot of a cell phone

Description automatically generated

# Creating and Structuring MUnit Tests

## Naming Convention



The MUnit framework enables developers to create test suites, which are grouped into unit tests as resources within the Mule project. Each test suite results in a separate configuration file (similar to Mule Configuration files), whereas each test is analogous to an ordinary flow/sub-flow. Because of that relationship, it to recommended to use a naming convention that mirrors the XML configuration or Mule flow that is being tested:

* Test flows: {name-of-the-flow}-unit-test
* Files: {name-of-the-mule-file}-unit-test

## Creating MUnit Tests in Anypoint Studio

There are two ways to create MUnit tests in Anypoint Studio.

1. Select the specific flow, right click, and select MUnit



1. Directly create MUnit from the wizard and specify flows for which you want to create MUnit tests



Once you create the MUnit test, you will be presented with a sample testautomatically created in **src/test/unit**.



# MUnit Test Suite Components

An MUnit test suite file may contain any combination of the following components:

* Imports
* Bean Definitions
* Before/After Suites
* Before/After Tests
* MUnit Tests

## Imports

Each MUnit test suite is meant to cover a particular set of Mule flows/sub-flows in a Mule configuration file. You need to tell MUnit where those flows are defined in your application structure using spring imports.

<spring:beans>

<spring:import resource="classpath:Calculator.xml"/>

</spring:beans>

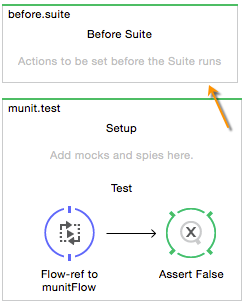
If there are dependent files that are needed for the flow to run, those also need to be imported. If your application is spread across multiple files and the components in those files cross-reference each other, you need to import each file.

## Bean Definitions

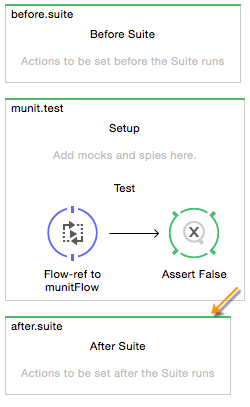
If there are any bean definitions, those need to be defined and imported as well.

## Before/After Suites

Each MUnit test suite has a Before Suite scope which contains logic that is meant to execute **before** the entire test suite runs. For instance, let’s suppose you have an MUnit test suite file with four MUnit tests. The logic inside the MUnit Before Suite scope runs just once, before your four MUnit tests execute. You can have multiple Before Suites. They all follow the same execution rule: Run before the tests, just once. **If there is more than one Before Suite, MUnit does NOT guarantee the order in which the Before Suites run amongst themselves.**

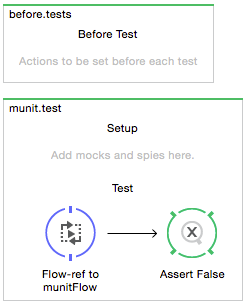


Each MUnit test suite has an After Suite scope which contains logic that is meant to execute **after** the entire test suite runs. For instance, let’s suppose you have an MUnit test suite file with four MUnit tests. The logic inside the MUnit After Suite scope runs just once, after your four MUnit tests complete executing. You can have multipleAfter Suites. They all follow the same execution rule: Run after the tests, just once. **If there is more than one After Suite, MUnit does NOT guarantee the order in which the After Suites run amongst themselves.**

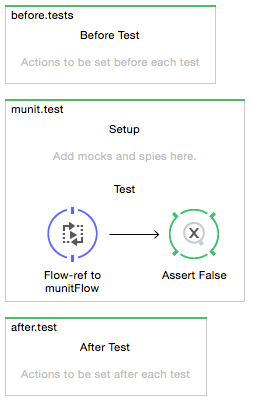


## Before/After Tests

Each MUnit test has a Before Test scope which containslogic that is meant to execute**before** each test.For instance, let’s suppose you have an MUnit test suite file with four tests. The logic inside the MUnit Before Test scope runs before each of your four tests; it runs four times. You can have multiple Before Test scopes. They all follow the same execution rule: Run before each test. **If there is more than one Before Test scope, MUnit does NOT guarantee the order in which the Before Test scopes run amongst themselves.**



Each MUnit test has an After Test scope which containslogic that is meant to execute**after** each test.For instance, let’s suppose you have an MUnit test suite file with four tests. The logic inside the MUnit AfterTest scope runs after each of your four tests; it runs four times. You can have multiple AfterTest scopes. They all follow the same execution rule: Run after each test. **If there is more than one After Test scope, MUnit does NOT guarantee the order in which the After Test scopes run amongst themselves.**



## MUnit Tests

Each MUnit test suite can contain one or more MUnit tests. Individual MUnit tests should correspond to a flow/sub-flow within the Mule configuration file that is covered by this test suite. More information about MUnit tests can be found in the next section.

# MUnit Test Components

There are a variety of components and message processors that can be referenced inside each MUnit test. Some of the more useful and popular ones are covered in this section.

## Mock Message Processors

MUnit comes with the ability to mock message responses while a unit test is running. Mocking allows you to define the mocked behavior of message processor in the flow. Normal behavior will be replaced by the one defined in the mock. Steps to define a mock:

1. First drag the **Mock** activity from the palate to your test suite. Place the **Mock** activity in the Setup section of your unit test
2. In the **When message processor matches section**, select the message processor from your flow
3. Optionally, you may specify attributes to control when the mock is executed. For example, you can decide to execute the mock only when the payload or flow variable is a certain value.
4. In the **Then return message with payload** section, you can specify the value returned when the mock is executed.

A screenshot of a cell phone

Description automatically generated

Some additional information about mocks:

* If you want to return the same payload as that of original processor use **#[samePayload()]**. Please note that if you do NOT specify anything in **Then return message with payload**,the mock will also returnthe original payload.
* If you want to returnfile content (in case of large payload), you can use the **getResource()** method:

<mock:then-return payload=”#[getResource(‘abc.xml’).asStream()]”/>

Instead of the asStream() method; asString() or asByteArray() can also be used, based on requirements.

* If you want to return payload as a result of some script then use **resultOfScript()**. For this, please set the script in the file with name ‘groovyScript’
  + Mocks can also be used to respond with an exception

<mock:throw-an whenCalling=”.\*:.” exception-ref=”#[new java.lang.Exception()]” doc:name=”Throw an Exception”

<mock:with-attributes>

<mock:with-attribute name=”doc:name” whereValue=”#[‘Transform Message’]” />

</mock:with-atrributes>

</mock:throw-an>

* + Inbound endpoints should be mocked by setting **mock-inbounds=true.** Let’s say you have a JMS inbound endpoint in your Mule flow that you are testing. When the MUnit test case is run, it will start that Mule flow and if the JMS inbound endpoint is NOT mocked, there is a possibility that it will dequeue active messages. Similar behavior can happen with other endpoints that unexpectedly start processing real messages. To prevent this behavior, youshould mock your inbound endpoints and test your Mule flows using a Flow Reference. **mock-connectors=true**will also mock all your connectorsfor similar reasons why we mock inbound endpoints.

## Spy Message Processor

The Spy Message Processor allows you to view what happens before and after a message processor is called. The Spy Message Processor only validates and does NOT modify the original Mule Message.If you set variables, attributes, or modify the payload inside a Spy Message Processor, these changes won’t persist outside the processor.The Spy Message Processor also supports executing when certain attribute criteria are met.

Assertions and verifications can be included inside the Spy Message Processor, to validate the values before and after executing a message processor such as a DataWeave transformation. Different asserts such as assert equals and assert true can be placed inside the Before call to evaluate the message **before** the actual message processor is executed. Similarly, the same asserts can be placed inside the After call to evaluate the message **after** the actual message processor is executed.

<munit:test name="Calculator-Spy-Dataweave" description="MUnit Test">

<mock:spy messageProcessor=".\*:.\*" doc:name="Spy - Double the Payload">

<mock:with-attributes>

<mock:with-attribute name="doc:name" whereValue="#[‘Double the Payload’]/>

</mock:with-attributes>

<mock:assertions-before-call>

<munit:assert-payload-equals expectedValue="5" doc:name="Assert Payload = 5"/>

</mock:assertions-before-call>

<mock:assertions-after-call>

<munit:assert-true condition="#[flowVars.DoublePayload == 10]” doc:name="Assert flowVars.DoublePayload Equals 10"/>

</mock:assertions-after-call>

</mock:spy>

<set-payload value="5" doc:name="Set Payload = 5"/>

<flow-ref name="Calculator-Flow" doc:name="Calculator-Flow"/>

</munit:test>

## Verify Event Processor

The Verify Event Processor allows you to verify if a message processor has been called with a particular set of attributes a specific number of times. For example, you can verify if a “Sent Email” activity has been triggered in your flow. When defining a verification, you are telling MUnit to fail a test if the verification is NOT successful. You can define Verify Event Processorsfor any message processor, even if you haven’t created a mock for it.

<munit:test name="Calculator-Verify-Email" description="MUnit Test">

<mock:when messageProcessor=".\*:.\*" doc:name="Send E-mail">

<mock:with-attributes>

<mock:with-attribute name="doc:name" whereValue="#[‘Send Notification’]"/>

</mock:with-attributes>

<mock:then-return payload="#[&quot;Sending E-mail&quot;]”/>

</mock:when>

<set-payload value="5" doc:name="Set Payload = 5"/>

<flow-ref name="Calculator-Flow" doc:name="Calculator-Flow"/>

<mock:verify-call messageProcessor=".\*:.\*" times="0" doc:name="Verify E-mail Never Sent">

<mock:with-attributes>

<mock:with-attribute name="doc:name" whereValue="#[‘Send Notification’]"/>

</mock:with-attributes>

</mock:verify-call>

</munit:test>

|  |  |
| --- | --- |
| times | This attribute verifies that the message processor is called exactly a certain number of times. The default value is 1. If the value is set to 0, it means the message processor should never be called. |
| atLeast | This attribute verifies that the message processor is called at least a certain number of times |
| atMost | This attribute verifies calls as successful when the message processor is called a maximum defined number of times |

**Note: The times, atLeast, and atMost attributes are mutually exclusive**

## Assert Message Processors

Assertion message processors are used to test conditions on a mule message and fail when specific criteria are NOT met. Assertions are commonly used inside the Spy Message Processor or at the end of an MUnit test. The MUnit framework includes several different types of Asserts: