# 7. Unit Testing Guidelines

## 7.1 Unit Testing Approach

Following Test-Driven Development (TDD) approach to API development provides a key benefit of being able to have the API contract defined, mock service created, and test plan with test cases and steps ready before even the full blown API development is under way. This allows for any API contract and code changes down the road to be fully regression test-ready, reducing the opportunity to uncover new code defects with frequently changing API contracts and code.

**Example of the TDD Approach:**

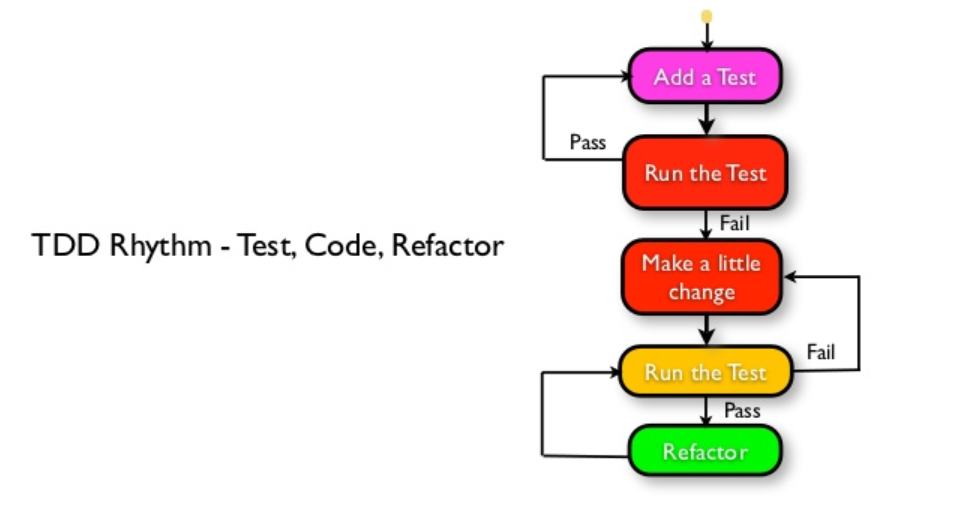
As an example, as a developer you are given a task to design and develop a particular System API to interact with the backend relational database and expose the Create/Read/Update/Delete (CRUD) operations in the database via RESTful API operations.

Following the TDD approach to designing and developing the API, first thing you do as an API developer is gather the requirements for the API operations and create an API contract (in RAML or Swagger). Once the contract is created, you generate a mock API. In Swagger, you would run a NodeJS or Java utility to generate source code based on the Swagger contract. In Mulesoft Anypoint Platform, using Design Center, you would generate a mock API using an designed RAML contract.

Once the mock service is up and available for consuming its operations, you create initial mock-driven unit tests using one of the unit testing frameworks. In Java, you would use JUnit to create unit test classes based your generated Java objects and methods. In NodeJS, you would use Mocha and Chai unit test frameworks. In Mule 4.x, you would use MUnit 4.x unit testing framework to create unit test flows and sub-flows, once you have developed the initial basic APIKit router and flows from your RAML contract. Once the unit tests have been created, you validate them by executing them inside your Integrated Development Environment.

All of the above steps are performed even before you started your actual API operation implementations. At this stage, you already have an API contract, mock API with operation returning mock data, and unit tests developed and fully tested. Notice that all of the above can be performed in a single “Planning” Sprint, before even the actual development Sprints have started.

Next, let’s assume that you have completed your Planning Sprint and started the actual API implementation to expose relational database CRUD operations via the REST API operations. Now, in the middle of your development, a new requirement is provided by the future API consumer to change the database schema of that relational database. You are forced to update your API contract and implementation of the API operation. Without unit tests that you have already developed, you are not able to validate whether all of your API contract changes are reflected correctly in your developed code. On the contract, if you have unit tests already created if following the TDD, you would just re-run your unit tests after any API contract and code changes. Executing unit tests is fairly quick and efficient way to validate any change you have to make to your RAML or Swagger contract and to the underlying codebase.



Note: Image created by Naresh Jain, Agile Faqs

## Agnostic Unit Testing

There are different unit testing frameworks used for API development, depending on what programming language and platform has been targeted for hosting the API-driven microservice. For Java, an obvious and widely used unit testing framework is JUnit. For NodeJS, it is Mocha and Chai. For Mulesoft API development, it is MUnit. No matter which unit testing framework is chosen, there are key unit test development guidelines to follow:

* Develop your unit tests to prove API implementation is working correctly as expected. Defect avoidance is the key goal of developing thorough unit tests.
* Ensure that code coverage with your unit tests is close to 100%. In other words, when developing your unit tests, almost every line of your code should be executed when running your unit tests. Test coverage reports are usually provided as part of a unit test framework of choice, or added on as part of the overall test harness.
* Ensure that your unit tests are targeting requirements provided by your target API consumers. In other words, don’t develop unit tests for methods and functions that are not directly relevant to the requirements. Build a traceability report from requirement or user story to a test case and step in your functional and unit test harness.
* Execute unit tests every time your API service operation has been modified, to prevent code regression between API release updates.
* For minor API release updates, execute focused unit test harness specific to the component or operation being changed. For major API release updates, execute full regression testing by running all unit tests for your API.

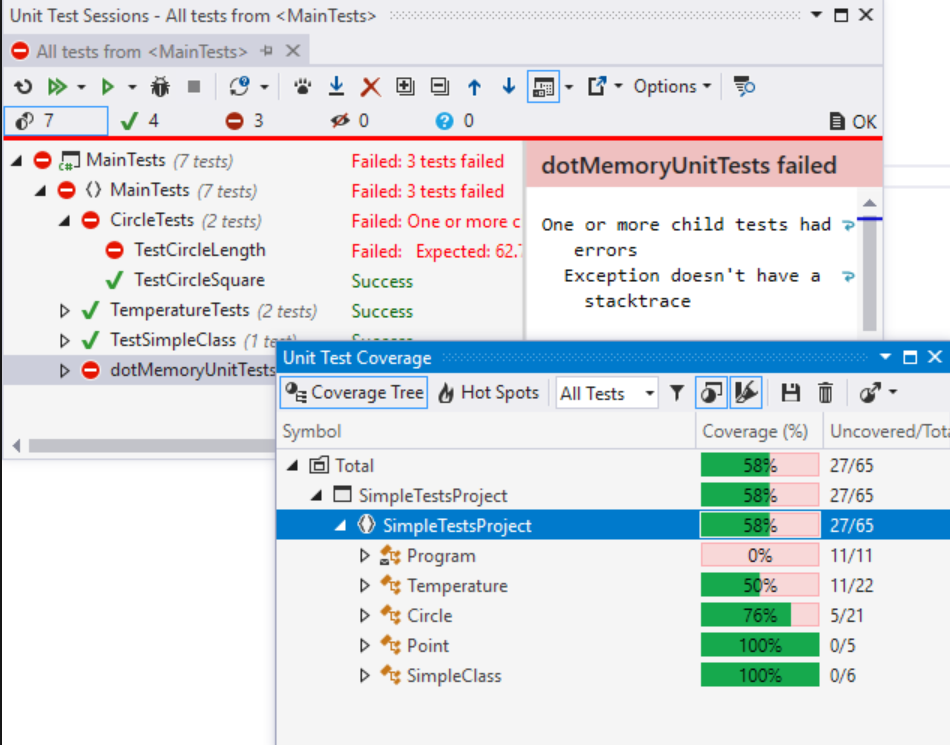
To develop unit tests, as a developer you would create the test classes in your main codebase associated with each of the key source code components and methods in your API source. For example, you have developed source code to implement a specific API operation to perform an HTTP POST of XML data to the backend system. Your corresponding unit test class will be contain the call to the method within your API operation that performs the HTTP POST. As the best practice, your unit test class won’t actually perform a POST to the backend, instead you would mock the request and response using a mock unit test framework such as Mockito or similar.

The point of unit tests is not simulate the behavior of the actual API operation using live operations, but rather to test the source code that you have written as a developer to perform any of the API operations. Therefore, you don’t need to actually make an HTTP POST call in your unit test class, but rather mock the HTTP request and response from the backend service, since you are not actually unit testing the backend source code, but rather you are unit testing your API source code.

The focus of the unit tests should be on more complex components in your API source code, such as security authentication/authorization, complex algorithms, transformations (e.g., XML to JSON), database operations, and service orchestration and business rules. Those components and operations are frequently subject to change due to updated requirements, and hence require extra regression testing via your unit and functional test harnesses.

The target goal, however, should be unit test coverage of most of your source code. Recommended test coverage percentage should be close to 95% of all source code. This means that close to 95% of all lines of source code should be tested during your complete unit test execution.

An example unit test coverage report looks like this (using JetBrains IDEA as an integrated development environment for Java development):



Unit tests are frequently plugged into automated Continuous Integration (CI) pipeline using Jenkins, Bamboo or TeamCity. The tests are executed during the test stage of the CI pipeline to guarantee that the source code delivered to a given environment is bug-free. Therefore, it is critical to develop your unit tests to cover most of your code to avoid any possibility of code delivered with bugs to QA or Production environments.

A screenshot below shows example Jenkins pipeline stages that execute unit tests, then functional tests, and finally load and security tests as three separate pipeline stages, before final deploy readiness:

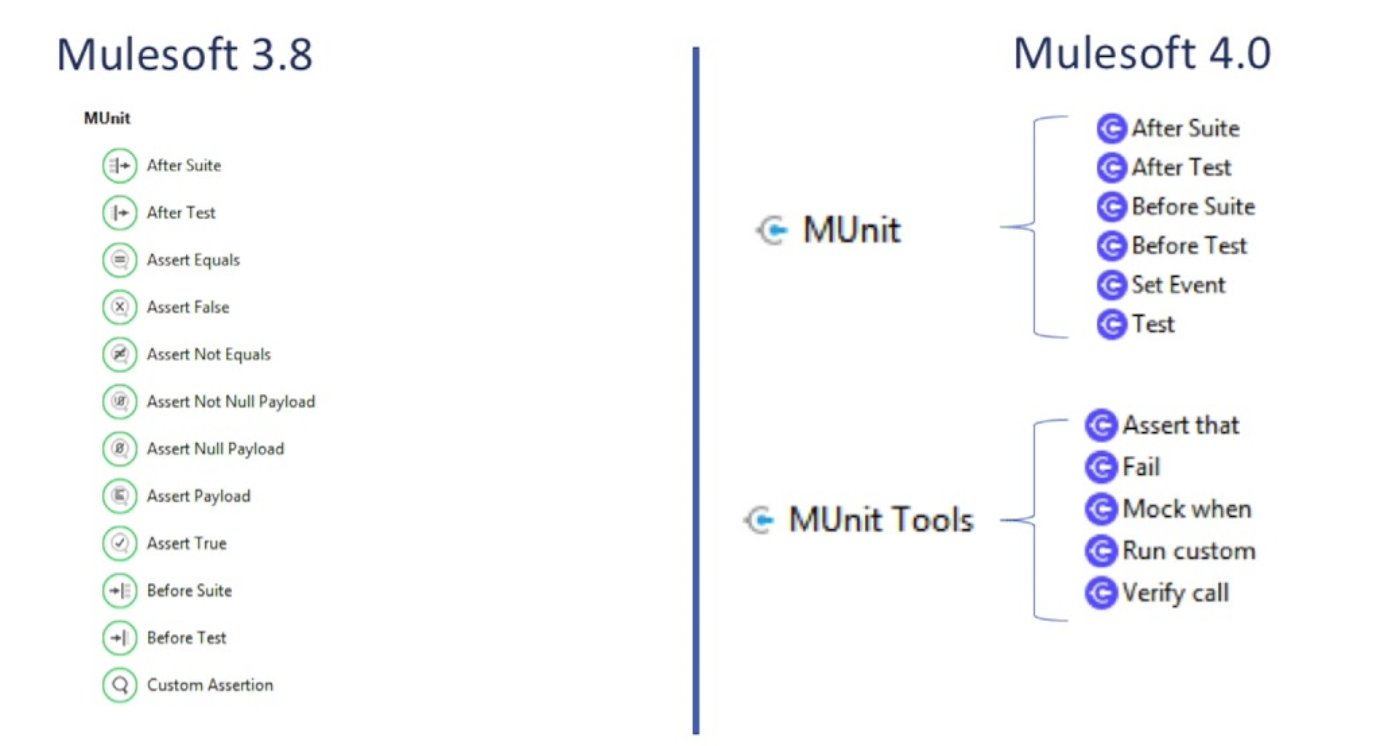


## 7.3 MuleSoft MUnit

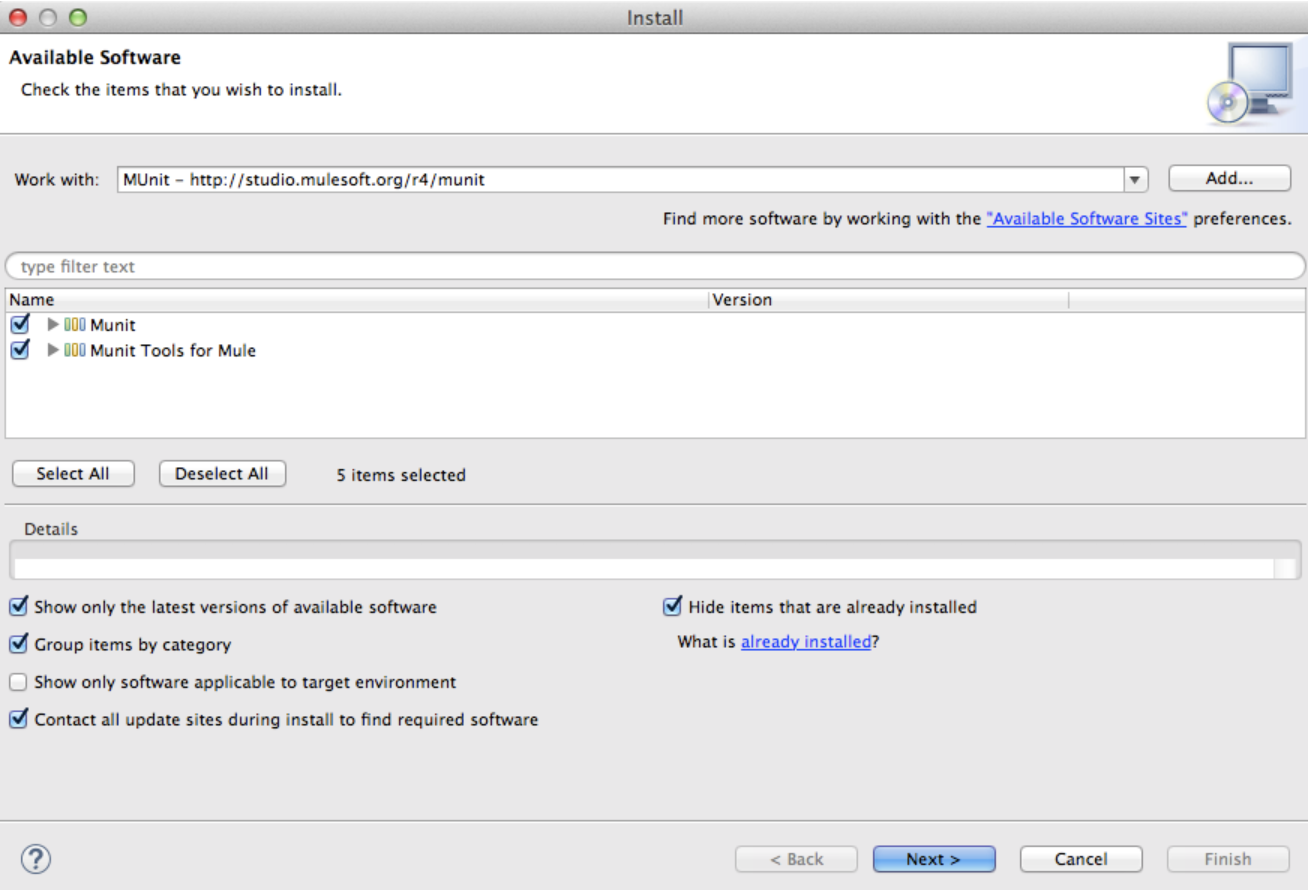
Prior to deploying your Mule applications and APIs, conduct unit and functional tests using MUnit, a native testing framework for Mule. As a developer, you would create the MUnit tests to be executed in your local environment, or in your continuous integration and continuous delivery (CI/CD) settings.

Mule 4.x reorganizes the MUnit assertions into two groupings, but majority of assertions remain the same as in MUnit 3.x. Instead of having separate assertions for True, False, Equals, etc, MUnit 4.x simplifies the assertions for equality or inequality into a single “Assert that” MUnit tool.

“Mock when” MUnit tool becomes quite useful to mock a response or statement when a specific variable or parameter contains a given value. For example, as an MUnit developer you may want to mock the response when a passed-in parameter equals to a specific value.

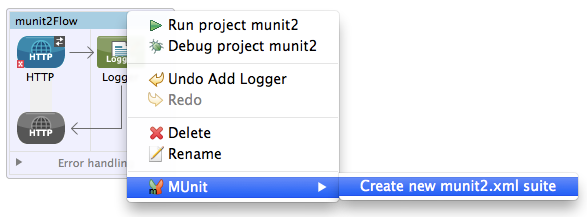


To create a new MUnit test, follow the Mule guidelines documented online for MUnit generation. There is a built-in utility within Anypoint Studio to generate MUnit tests. With previous versions of Anypoint Studio, you would had to manually download and install the MUnit Tools for Mule plugin:



However, with the new Anypoint Studio 7.x, MUnit already comes pre-bundled with the Studio.

To generate the MUnit flows, simply select a given flow in your code, right-click and select option to generate MUnit flow:



This will generate a “skeleton” version of the MUnit flow. Next, you would add custom logic and assertions using MUnit tools such as “Assert that”, which is the most frequently using assertion type.

You can then run and debug your MUnit tests directly inside Anypoint Studio, with various test data and conditions.

### 7.3.1 Mocking Features

* 1. As per best practices outlined in the previous section, mocking backend system responses from a downstream service or platform is recommended. Remember that as a developer, your goal with MUnit is not to test your backend platform, but to test your own Mule flows and sub-flows. Therefore, when your Mule flow is making an HTTP Request or a JDBC call to the database, you would want to mock the response from that HTTP service or database, instead of making the actual call.

To mock a response in MUnit, you would use the Mock Message Processor. The Mock feature provided by MUnit allows you to define mocked behavior for a message processor. In this case, MUnit replaces the normal behavior of the message processor with the behavior you define. Therefore you can modify how a specific message processor responds when it is called with a particular set of attributes. You would use “Mock when” MUnit tool to enable the mock message processor:

<mock:when messageProcessor="mule:set-payload">

</mock:when>

A mock definition is based on ***matchers***, that is, parameters that match features of the desired message processor. Defining a mock solely on the name of the message processor largely limits your scope and actions regarding the mock. For this reason, MUnit allows you to define a mock by defining matchers over the value of a message processor’s attributes.

<mock:when messageProcessor="mule:set-payload">

<mock:with-attributes>

<mock:with-attribute whereValue="#['Real Set Payload']" name="doc:name"/>

</mock:with-attributes>

</mock:when>

You can define as many attributes as you deem necessary to make the mock as representative as possible. When defining an attribute, you do so by defining these two parameters in the “with-attribute” XML tag of the message processor:

* **name** – the name of the attribute
* **whereValue** – the value of the attribute

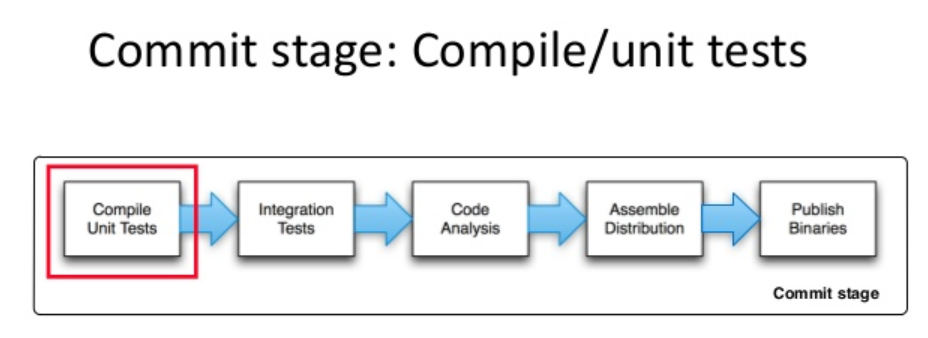
## Integration with Continuous Integration (CI) / Continuous Deployment (CD)

The key here is automation - irrespective of various techniques employed by QA teams, humans are not capable of executing complex test scenarios as consistently, frequently and as quickly as machines can.

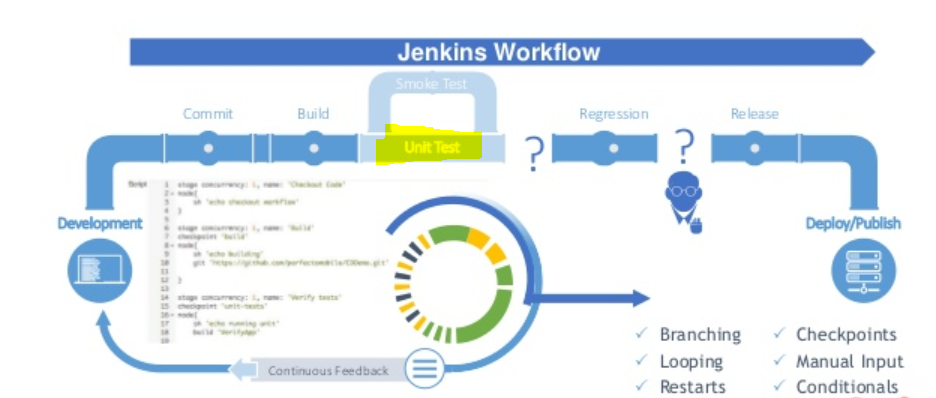
* There are tools available to continuously run unit tests whenever a code file is saved on disk.
* There are continuous integration systems to run unit and integration tests when code is committed to local branch or to central repo.
* Test script can be scheduled to run every XX minutes continuously to check for certain conditions 24x7.

Unit test harness is typically executed as part of the CI pipeline stage called “Commit Stage”, which is triggered after the code is pulled automatically from a source code repository (e.g., Git or GitHub or SVN) onto a build server and compiled.

After the source code is pulled and compiled on the CI build server, the next stage in the CI pipeline is to execute the unit tests that you have developed as part of your API development lifecycle.



A Continuous Integration (CI) workflow with a unit test stage is shown below, using Jenkins CI Workflow as an example:



When the unit test stage is completed, the unit test coverage report is automatically generated as part of the CI stage. That report is available for viewing on the CI Management Console, such as inside Jenkins CI Console. Finally, the Regression and Release stages are executed before the final Deploy and Publish stages. The Continuous Delivery automation can be integrated together with the Continuous Integration, in a single CI/CD workflow. CD portion of the automation workflow is dealing with actual deployment of the source code into a virtual container (e.g., Docker or OpenShift Pod) or onto the “bare metal” server instance if virtualization is not used.