Who Are We?

Founded in January 2010, Mountain State Software Solutions, LLC (MS³) is a Global IT consulting firm based in the Washington DC metropolitan area that specializes in engineering future proof solutions for both commercial and federal customers. MS³ manages a global team and has the ability to operate on a 24-hour development life cycle with over 50 engineers armed with a wide range of capabilities within the software development arena in North America alone. Additionally, MS³ also employs a resource team located in the Philippines to offer additional cost effective solutions to our clients and partners. With extensive expertise in API integration and enablement, Big Data, Dev/Ops, onPrem and Cloud solutions as well as ongoing Operations Support, MS³ is a leading provider of enterprise ready mission critical software solutions providing globally distributed organizations the ability to meet today’s most complex business challenges.

What We Do?

- Integration
- Portal
- Cloud
- Big Data

MuleSoft MUnit: Development Best Practices
ABSTRACT

This article, intended for software developers and architects who have a basic knowledge of the MuleSoft Anypoint® development environment, provides useful development best practices about the MUnit testing framework [1]. A very high-level overview of the framework itself is given first for ease of reading.
MUnit Overview

MUnit General Concepts

*MuleSoft MUnit*® testing framework [1] is being increasingly used, due to the significant amount of “intelligent automation” it brings to the Software Development Lifecycle (SDLC). This is especially true in the area of Continuous Integration and Continuous Deployment (CI/CD), as typically dictated by Test-Driven Development (TDD) patterns [4].

MUnit tests are very similar to JUnit tests, which are widely used in the Java community. The framework adds additional robustness to the CI/CD chain by using frameworks, such as Jenkins, to avoid untested code getting built and deployed.

As the diagram suggests, build errors and failed unit tests will prevent deployment. MUnit, which is tightly integrated into the MuleSoft Anytime Studio development environment, allows a command line interface to be invoked within the scope of the deployment framework in order to determine where tests are failing (and why).
MUnit Object Model

The MUnit® framework enables the developer to create test suites, which are grouped into unit tests as resources included in the Mule project.

The following screenshot shows a sample of what these resources looks like in the project.

Each Test Suite results in a separate configuration file (similar to Mule Configuration files), where each test is analogous to an ordinary Process Flow. It actually IS a process flow, responsible for a specific test scenario.
MUnit Best Practices

Use Maven

Maven is a software packaging framework that specifies how an application (usually Java) should be assembled, including any dependencies it may have. By default, most Mule applications are built using Maven because of its many useful features, such as its efficient support to dependency management. The CI/CD process relies on frameworks like this, to assemble applications to be continuously deployed.

Plan

Don’t forget the piece you are building is just a part of something bigger: the CI/CD chain! You want to avoid creating blockers to the DevOps team. So, the earlier in the project lifecycle you give them the hooks they need, the better.

Pro Tip:

Anticipate the number and names of test suites to be created and stub them with a single dummy test which always succeed. This creates placeholders for better tests later.

It’s possible to execute a specific test in a test suite, but Jenkins will typically be configured to run one, some, or all of the tests. This allows you to unlock the DevOps team by setting up the automations needed on their side. As time allows, go back and write more meaningful MUnit code, replacing the previous stubs with real tests. You can add more tests to an existing test suite in a manner that is transparent to the rest of the DevOps team.
Structuring Test Suites

There are no hard and fast rules for how you should organize your test suites. Generally, test suites are organized in one of two ways:

- **By functionality**
  (Suites are grouped by functions or subfunctions that they test.)

- **By interface**
  (Suites are grouped by the endpoint system or set of systems being tested.)

Depending on the specific project, both of these criteria may make sense, so you should consider that a mix of criteria may apply. Use common sense and keep in mind that the ultimate goal is to keep the CI/CD chain flexible. Having a granular structure of test suites may help in adopting gradual approaches, where the suites you use will depend on the point you’re at in the SDLC process.

**EXAMPLE:**

“Let’s not use the db-tests.xml test suite before the related development is complete.”

Code Coverage: Acceptance Percentage

Depending on the projects, reasonable numbers can be in the 80-90% range. Do not spend an excessive amount of energy to hit the magic 100% code coverage objective, or even any specific number for that matter. This is often an overrated and misleading metric. You could get a 100% by hitting all the process flows exactly one time but still be missing important particular sequences in which those lines are hit. Developers should focus on the most frequent scenarios and be thorough in testing those. If you have a use case that happens twice a year with negligible business impact, it should not be a big deal if it is not covered, even if it takes 40% of the code. So... use common sense!
Code Coverage: Output Generation

Properly configured, MUnit can generate a coverage report in the “target” directory of your build. By default, the report comes in HTML format but I recommend also creating it in JSON format, which better supports automated tasks.

Code Coverage: POM configuration

The code coverage behavior can be configured in the POM file using the Maven MUnit Plugin:

```xml
<project ..... ...
<plugins>
  <plugin> ..... 
  </plugin>
  ...
  <plugin>
    <groupId>com.mulesoft.munit.tools</groupId>
    <artifactId>munit-maven-plugin</artifactId>
    <version>${munit.version}</version>
    ...
    <configuration>
      <coverage>
        <runCoverage>true</runCoverage>
        <failBuild>true</failBuild>
        <requiredApplicationCoverage><numeric>
        <requiredResourceCoverage><numeric>
        <requiredFlowCoverage><numeric>
        <formats>
          <format>html</format>
          <format>json</format>
        </formats>
      </coverage>
    </configuration>
  </plugin>
</plugins>
```
How Much MUnit Code to Write

Another interesting metric to consider is the LPC/LTC ratio, which is number of Lines of Production Code (LPC) versus the number of Lines of Test Code (LTC).

Several studies (publicly available on the web) suggest that a ratio of about 1:4 (LPC:LTC) is sufficient for 80% test coverage.

My recommendation is to avoid overly focusing on a metric like this. Metrics cannot magically create good code. The approach I suggest is to make sure you are thoroughly testing the most frequently occurring scenarios. Only then should you start improving the ratio.

Avoid overly focusing on a metric like LPC:LTC. Metrics cannot magically create good code.

Command Line Testing

MUnit allows tests to be run inside of MuleSoft Anypoint Studio®, which is certainly a useful step to perform while coding. However, PRIOR to committing/pushing your software to the code repository you need to test the MUnit artifacts in the actual way they are going to be executed in the scope of your project. For example, you would need to test from the command line if you are using Jenkins. Once the environment has been set properly, to launch a single suite, use:

```
mvn clean test -Dmunit .test=<MUnit test suite name>.xml
```

Omit the -D option to launch them all.

Note: Although the Eclipse environment correctly manages dependencies, you might experience some unresolved dependencies in the command line context. You might not normally see these, which is why command line testing is useful. These dependencies can be fixed in the Maven pom.xml.
When using MUnit for APIKit-based projects, the developer can right-click on the APIKit Router to generate useful test skeletons. While you may want to take advantage of this feature, make sure that the customer’s expectations are set properly. These are just skeletons and further development work will be needed to get the exact tests you want. You might also need more copies of the individual tests depending on the various logical paths to be examined.

Mocking Database Query Activities

Debug your application code to get the exact structure of what the payload will look like AFTER the query results set has been returned. This is the structure that you will need to mock up. Similarity is not enough; the structure must be exact. As an example, consider the following debugging screenshot.

Here you can see the database returning a `java.util.LinkedList` as a result set where each item (row) is contained in a `org.mule.util.CaseInsensitiveHashMap`. Note, the mockup would fail if we use a MEL expression to create a payload like the following.

```
{PHONE:'(815)-254-0356', DISTANCE:'1.3 miles', ADDRESS2:'Plainfield IL 60544', UID:'1000', ADDRESS1:'13546 State Route 30', SERVICES:'Gift Cards'}
```

While you may want to take advantage of this feature, make sure that the customer’s expectations are set properly. These are just skeletons and further development work will be needed to get the exact tests you want. You might also need more copies of the individual tests depending on the various logical paths to be examined.
This will fail because this MEL expression returned a `java.util.ArrayList` as a result where each item (row) sits in a `java.util.HashMap`. These are not exactly the required types.

An alternative is to create a groovy script as follows, named "mockDbRowScript" which, in this example, returns the proper structure:

```java
import org.mule.util.CaseInsensitiveHashMap;
import java.util.LinkedList;

LinkedList<CaseInsensitiveHashMap<String, String>> rSet = new LinkedList<CaseInsensitiveHashMap<String, String>>();
CaseInsensitiveHashMap<String, String> rowDataMap = new CaseInsensitiveHashMap<String, String>();
rowDataMap.put("PHONE", "(815)-254-0356");
rowDataMap.put("DISTANCE", "1.3 miles");
rowDataMap.put("ADDRESS2", "Romeoville IL 60446");
rowDataMap.put("UID", "1000");
rowDataMap.put("ADDRESS1", "13546 State Route 30");
rowDataMap.put("SERVICES", "Gift Cards");
rSet.add(rowDataMap);
return rSet;
```

This is referenced from the mock code with a call to the "resultOfScript" function:
Mocking Database with Insert/Update/Delete Activities

A similar approach is recommended for INSERT/UPDATE/DELETE scenarios. Make sure to correctly mimic your payload, which in this case would have a much simpler structure of `java.lang.Integer`. This value represents the number of rows inserted, updated, or deleted respectively.

Mocking Other Database Activities

The mocking of Stored procedure, Bulk Execute, Execute DDL operations deserves more articulate examples and is out of the scope of this article, but the approach is the same. As long as you mimic accurately the structure of the generated payload (see it by debugging), your solution will work.

Preventing Naming Conflicts

Naming conflicts are usually reported as “problems” in Eclipse IDE but they are not always managed properly! For example, when launching MUnit wizard to create the RAML-based test suite skeletons (the one you launched by right-clicking the APIKit Router), AP Studio will try to create an HTTP Request Connector named “HTTP_Request_Connector.” If another HTTP request exists with the same name in the project or any Mule library referenced by the project, the connector will be considered valid. This assumption is not always true but it can lead to some very tricky scenarios. Hopefully, MuleSoft will fix this in a future release, but the best practice is to check your connector names so that they are unique in the scope of your application (including libraries), and double-check that everything works as expected.

References

[1] https://docs.mulesoft.com/munit/v/1.3
[3] https://jenkins.io
CONCLUSION

We have provided an overview of the MuleSoft Unit [1] test suite, illustrating how, in a completely seamless way and while being fully integrated with the MuleSoft Anypoint development environment, programmers can be enabled to efficiently adopt well known TDD practices, with explicit reference to their usage in a CI/CD context. MUnit development best practices and code coverage best practices were provided.
FUTURE PROOF
CLOUD INTEGRATION

www.ms3-inc.com
contact@ms3-inc.com
304-579-8100
308 S Fairfax Blvd.
Ranson, WV 25438