# Mulesoft Development Guidelines

Version 1.0

Revision History Page

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Description of Changes | Authors |
| 3/5/2019 | 1.0 | Initial Version | Ankur Parashar |
|  |  |  |  |
|  |  |  |  |

## Table of Contents

[Table of Contents 3](#_Toc7800008)

[Introduction 4](#_Toc7800009)

[Software Development Lifecycle Tools 4](#_Toc7800010)

[Coding Conventions 16](#_Toc7800011)

## Introduction

The following document outlines best practices and coding standards for a mulesoft project.

## Software Development Lifecycle Tools

The following tools should be used in the development of MuleSoft solutions.

**Development IDEs**

**MuleSoft Anypoint Studio**

<https://www.mulesoft.com/platform/studio>

MuleSoft solutions can be developed using a range of tools, from simple text editors to Integrated Development Environments (IDE).

For the most comprehensive development experience, it is recommended that MuleSoft Anypoint Studio is used; Primarily because it offers the following.

1. Visual development
2. XML editing
3. Development tool integration
4. Debugging support
5. Testing framework support

As Anypoint Studio is based on Eclipse, there is a plugin which will extend Eclipse to provide many of the features of Anypoint Studio.

<https://docs.mulesoft.com/mule-user-guide/v/3.7/studio-in-eclipse>

**Build Technologies**

**Apache Maven**

[https://maven.apache.org](https://maven.apache.org/)

Apache Maven 3.x will be used for build and dependency management. This is recommended by MuleSoft and provides many hooks and tooling for managing Mule applications.

Apache Maven is a software project management and comprehension tool.

Based on the concept of a project object model (POM), Maven can manage a project’s build, reporting and documentation from a central piece of configuration.

Maven projects are created following conventions. This means that source code and other project resources reside in a standard folder structure and never change from project to project.

**Continues Integration**

**Jenkins**

[https://jenkins-ci.org](https://jenkins-ci.org/)

Jenkins is a continuous integration application that monitors the execution of repeated jobs, such as building a software project or jobs run by cron.

Among those things, current Jenkin versions focus on the following two jobs:

1. Building/testing software projects continuously, Jenkins provides an easy-to-use so-called continuous integration system, making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build. The automated, continuous build increases productivity.
2. Monitoring executions of externally-run jobs, such as cron jobs or deployment jobs, even those that are run on a remote machine. For example, with cron, all you receive is regular e-mails that capture the output, and it is up to you to look at them diligently and notice when it broke. Jenkins keeps those outputs and makes it easy for you to notice when something is wrong.

**Dependency Management**

**Artifactory**

<https://www.jfrog.com/open-source>

Artifactory is what is known as a Repository Manager. It is used to store released application binaries, libraries and other assets. Artifactory also has the ability to act as a proxy to remote repositories, allowing for a central system to retrieve project dependencies.

**Software Configuration Management**

**GIT**

[https://git-scm.com](https://git-scm.com/)

GIT is a Software Configuration Management (SCM) system. This is used to track changes to software systems and provide advanced code management functions.

**SCM Repositories**

SCM repositories allow a centralized storage location for all software and change management.

GitLab and GitHub should be used as SCM Repositories

**Branching and Tagging Strategy**

The branching and tagging strategy that will be used initially is the “Github flow”

The main points of this flow are:

* Anything in the master branch is deployable.
* To work on something new, create a descriptively named branch off of master.
* Commit to that branch locally and regularly push your work to the same named branch on the server.
* When you need feedback or help, or you think the branch is ready for merging, open a pull request.
* After someone else has reviewed and signed off on the feature, you can merge it into master.
* Once it is merged and pushed to ‘master’, you can and should deploy immediately.

Depending on how complex the solution and live implementations become it may be necessary to move to the Git Flow.

**Mule Project Development Best Practices**

### Modularization

**Application Modularization**

Mule allows you to run applications side-by-side in the same instance. Each Mule application should represent a coherent set of business or technical functions and, as such, should be coded, tested, built, released, versioned and deployed as a whole. Splitting particular functions into individual applications allows a coarse-grained approach to modularity and is useful when keeping elements of your application running while others could go through some maintenance operations. The problem of packaging single flows into individual applications is that, currently, you lose the possibility to use the VM transport across your flows.

The recommended approach is to use a message queuing system (JMS, AMQP) to tie your flows together. With this in place, you can deploy (and redeploy) them as single applications: they will stop, upgrade and resume consuming messages during this redeployment.

For optimum modularity:

1. Consider what functions are tightly interrelated and keep them together in the same Mule application: they will form sub-systems of your whole solution.
2. Establish communication channels between the different Mule applications: the VM transport will not be an option here, as it can’t be used across different applications. Prefer the TCP or HTTP transports for synchronous channels and JMS for asynchronous ones.

**Configuration Modularization**

Though it may seem convenient to keep all Mule configurations in one place, the reality is that a gigantic XML file quickly becomes unmanageable. This is why it is recommended to split monolithic configurations into several files and leverage Mule’s capacity to load multiple configuration files at application start-up time. Moreover, splitting configurations into multiple fragments encourages re-use across teams.

Mule offers two options for loading several configuration files:

* side-by-side: provide a list of independent configuration files to load.
* imported: have one configuration file import several others, which in-turn can import other files.

In practice, it is common to use both approaches simultaneously.

Don’t forget that all the configuration files end up loaded in the same context; therefore you should be careful and use unique names for all your configuration elements. Mule will refuse to load an application whose configuration files contain name conflicts.

How can you determine what constitutes good separation lines between configuration fragments? Here are a few rules of thumb:

* Business domains usually form a natural border that can be used to separated configuration elements
* Keeping together elements that have similar reasons for change reduces the risk of impacting unrelated aspects of your application
* Technical aspects, like administrative components, security or Spring beans configuration, define good lines of demarcation
* Extracting a side-by-side transport configuration (connectors and endpoints) facilitates functional testing. Note that it is not intended to take care of environment specific transport configuration, which is dealt with properties files
* Reusability across teams and projects

Mule relies on Spring XML configuration for importing configuration files into each other.

Here is the main configuration file illustrated above, which takes care of importing the three other configuration elements:

|  |
| --- |
| <mule xmlns=”<http://www.mulesoft.org/schema/mule/core&#8221>;  xmlns:xsi=”<http://www.w3.org/2001/XMLSchema-instance&#8221>;  xmlns:spring=”<http://www.springframework.org/schema/beans&#8221>;  xsi:schemaLocation=”  <http://www.mulesoft.org/schema/mule/core>  <http://www.mulesoft.org/schema/mule/core/3.1/mule.xsd>  <http://www.springframework.org/schema/beans>  [http://www.springframework.org/schema/beans/spring-beans-current.xsd”&gt](http://www.springframework.org/schema/beans/spring-beans-current.xsd%E2%80%9D&gt);    <spring:beans>  <spring:import resource=”domain-A-config.xml” />  <spring:import resource=”domain-B-config.xml” />  <spring:import resource=”admin-config.xml” />  </spring:beans>  </mule> |

**Flow Modularization**

When implementing complex integrations, it will become increasingly important to logically separate functionality into units.Much like traditional programming models, each unit/flow should represent a single function. As such, it is important to break down larger application flows into smaller flows.This is achieved by grouping related message processors into individual flows and or subflows. However, care must be taken if transactional message processing is required.

For more information on flow creation, please review:

<https://docs.mulesoft.com/mule-fundamentals/v/3.7/flows-and-subflows>

**Custom DevKit Modules/Connectors**

Custom Mule modules and connectors built using the Mule DevKit are an artifact in their own right and should be source controlled separate from any other project or module.

**Custom Business components**

 Generic (non Mule aware) business components that need to be shared should be separated into their own project, e.g. business-commons.

**Custom Mule Components**

Besides business components, there are Mule specific programmatic artifacts that are worth separating into their own project, e.g.  a “mule-commons” project such as:

* Custom transformers – performing operations that none of the Mule stock transformers can perform.
* Custom components – typically Mule-aware or non-business oriented components, as business components are usually simple POJOs coming from pre-existing projects.

**Application Structure**

All Mule applications will follow the standard Maven/Mule project structure. Both Anypoint Studio and the Mule App-kit Command Line Interface (CLI) will automatically create projects with the required Maven structure and files needed similar to the structure detailed in Listing 1.1.

Listing 1.1: The recommended structure for a Mule application.

|  |
| --- |
| |\_\_\_\_pom.xml  |\_\_\_\_src  | |\_\_\_\_main  | | |\_\_\_\_app  | | | |\_\_\_\_functional-usecase.xml  | | | |\_\_\_\_global.xml  | | | |\_\_\_\_mule-deploy.properties  | | |\_\_\_\_java  | | | |\_\_\_\_com  | | | | |\_\_\_\_orgName  | | | | | |\_\_\_\_immediate  | | | | | | |\_\_\_\_SomeClass.java  | | |\_\_\_\_resources  | | | |\_\_\_\_mule-app.properties  | | | |\_\_\_\_local  | | | | |\_\_\_\_mule-app.properties  | | | |\_\_\_\_dev  | | | | |\_\_\_\_mule-app.properties  | | | |\_\_\_\_test  | | | | |\_\_\_\_mule-app.properties  | | | |\_\_\_\_prod (we call it demo)  | | | | |\_\_\_\_mule-app.properties  | | | |\_\_\_\_wsdl  | | | | |\_\_\_\_SomeService.wsdl  | | | |\_\_\_\_xslt  | | | | |\_\_\_\_SomeStylesheet.xsl  | |\_\_\_\_test  | | |\_\_\_\_java  | | | |\_\_\_\_com  | | | | |\_\_\_\_orgName  | | | | | |\_\_\_\_immediate  | | | | | | |\_\_\_\_UnitFunctionalTest.java  | | |\_\_\_\_munit  | | | |\_\_\_\_my-munit-test.xml  | | |\_\_\_\_resources  | | | |\_\_\_\_functional-usecase-test.xml  | | | |\_\_\_\_LoadTestScript.jmx  | | | |\_\_\_\_IntegrationTestScript.jmx  |\_\_\_\_target |

**Directory Structure**

The use of Maven dictates that a standard directory structure is followed.

The project will consist of the following main directories:

* src/main/java Application/Library sources such as business components, transformers and custom components.
* src/main/resources Application resources such as schemas, wsdls and stylesheets.
* src/main/app Mule xml configuration files and Mule property runtime files such as mule-app.properties, mule-deploy.properties.
* src/test/java Test sources – Mule Functional and Unit tests.
* src/test/resources Test resources – Sample responses, mocked configuration files and integration test scripts such as Jmeter jmx files or SoapUI scripts.

**Dependency Management**

Maven is used for all dependency management. The following details how and where common dependencies should be located.

**External Dependencies**

Mule is very modular and requires many dependencies. The default-generated pom.xml will contain the necessary Mule repositories for downloading it’s dependencies. If you are not starting with the generated POM, then you can find repository information here:

<http://www.mulesoft.org/documentation/display/33X/Mule-Maven+Dependencies>

**Enterprise Dependencies**

 Mule Enterprise edition is not freely available, neither are it’s dependencies. Enterprise dependencies are bundled along with the EE runtime when downloaded. In order to use these with Maven they must be installed into your Maven repo. This can be accomplished by running the populate\_m2\_repo script from $MULE\_HOME/bin directory that will install each library as a dependency within your local Maven repo.

These dependencies can then be pushed to a privately hosted artifact repo such as Artifactory and the private repository can then be configured within your pom.

**Private dependencies**

Custom and internal artifacts should be deployed to a privately hosted artifact repository such as Artifactory and the resulting repository configured within each projects POM in order to resolve dependencies.

**Maven Group Id**

For each Maven artifact the groupId that must be used is:

com.org.project

**Environment properties**

Environment properties will be handled via Maven profiles. Profiles are specified using a subset of the elements available in the POM itself (plus one extra section), and are triggered in any of a variety of ways.  They modify the POM at build time, and are designed to be used in complementary sets to give equivalent-but-different parameters for a set of target environments (providing, for example, the path of the app server root in the development, testing, and production environments). As such, profiles can easily lead to differing build results from different members of your team.

However, used properly, profiles can be used while still preserving project portability. This will also minimize the use of -f option of maven, which allows user to create another POM with different parameters or configuration to build which makes it more maintainable since it is running with one POM only.

The root mule-app.properties file will contain recursive links to any properties that are environment specific, for example:

|  |
| --- |
| db.host=${db.host}  db.env=${db.env}  db.user=${db.user}  db.password=${db.password} |

Additionally an environment specific properties file containing the actual property values will be created in it’s own package representing that environment.

For example in Listing 1.1 the project structure contains three additional mule-app.properties files in the following folders: local, dev, test and demo each containing their specific environments properties. These directories are then configured as profiles within the projects pom.xml as follows:

|  |
| --- |
| <profiles>  <profile>  <id>dev</id>  <activation>  <activeByDefault>true</activeByDefault>  </activation>  <properties>  <profile.name>local</profile.name>  </properties>  </profile>  <profile>  <id>test</id>  <properties>  <profile.name>dev</profile.name>  </properties>  </profile>  <profile>  <id>test</id>  <properties>  <profile.name>test</profile.name>  </properties>  </profile>  <profile>  <id>prod</id>  <properties>  <profile.name>demo</profile.name>  </properties>  </profile>  </profiles> |

A default profile can be activated using the activeByDefault element, in this case: dev. When using MuleStudio ensure you run your projects using “Run as Mule application with Maven” option so it can correctly import the properties. Alternatively you will need to install the m2eclipse plugin for Eclipse and set the default profile via navigating to project/properties/Maven and set the necessary profile.

The profile.name element representing the directory name for the properties file location is then injected by Maven at build time using a filter:

|  |
| --- |
| <build>  <finalName>${project.artifactId}</finalName>  <filters>  <filter>src/main/${profile.name}/mule-app.properties</filter>  </filters>  …  </build> |

Building with a specific profile is done via -P argument as the following command demonstrates:

|  |
| --- |
| mvn clean install -P prod |

## Coding Conventions

**Code Format**

Our Eclipse code formatter configuration is stored in the following location:

|  |
| --- |
| <GITHUB\_HOME>/xxx/build/formatter.xml |

This configuration is based on the default “Java Conventions” formatter that comes with Eclipse, with one or two changes. To enable this formatter in MuleStudio follow these steps:

* From the top menu bar, select “Window” –> “Preferences” –> “Java” –> “Code Style” –> “Formatter”.
* Select the “Import” button and navigate to the above XML file, then click “OK”. The “iihformatter” should be selected as the active profile.
* To format your code, with a Java class open simply use the following keyboard shortcut (in Ubuntu): SHIFT + CONTROL + F.

**Checkstyle**

Our Checkstyle configuration is kept as part of the Maven build-tools project in the following locations:

|  |
| --- |
| <GITHUB\_HOME>/xxx/mule/commons/src/main/resources/checkstyleconfig.xml |

This configuration is based on the default “Sun Checks” config that comes with Checkstyle, but with some rules relaxed.

The build-tools.jar file should be added as a dependency to any project where Checkstyle analysis is required during the Maven build, so that the Maven Checkstyle plugin itself can find the config.

This is likely to be done in a parent pom.xml (so all child projects can make use of it). Additionally, the maven-checkstyle-plugin should be added as a plugin in the pom.xml of   project where Checkstyle analysis is required.

Jenkins jobs which invoke the Maven build should be configured to publish the results of the Checkstyle analysis, to ensure visibility of the analysis so errors and warnings do not get ignored. However, developers should be fixing Checkstyle errors and warnings as and when they appear, and this can be done using the Eclipse Checkstyle plugin.

To install the plugin, follow the steps given here:

<http://eclipse-cs.sourceforge.net/downloads.html>

* From the top menu bar, select “Window” –> “Preferences” –> “Checkstyle”. Click the “New” button.
* Select “Type” as “External Configuration File”, give it the name “IIHCheckstyleConfig” and location pointing to the above checkstyleconfig.xml file.
* Click “OK” and finally highlight the new entry and “Set as Default”.

Existing projects should already be configured to use this config via a checkstyle file in the project root. This is done by right clicking on the project within Mule Studio, selecting “Properties” –> “Checkstyle” and ensuring that the “Checkstyle active for this project” and “Use simple configuration” checkboxes are checked, and that it the IIHCheckstyleConfig checks are being used (from the drop down list).

**Java Code Conventions**

**Naming Classes**

As per the example below, include all standard naming conventions.

|  |
| --- |
| AbstractClassName (which enforces Abstract prefix or Factory suffix) and ConstantName (which enforces all CAPS) may be excluded.    <!– <http://checkstyle.sf.net/config_naming.html> –>  <module name=”LocalFinalVariableName”/>  <module name=”LocalVariableName”/>  <module name=”MethodName”/>  <module name=”PackageName”/>  <module name=”ParameterName”/>  <module name=”StaticVariableName”/>  <module name=”TypeName”/>  <module name=”MemberName”/>  <!– <module name=”ConstantName”/> –> |

**Importing Classes**

As per the example below, include AvoidStarImport. Though it makes Java files long, it provides clear information about classes and their use. Further, include IllegalImport, RedundantImport and UnusedImports.

|  |
| --- |
| <!– <http://checkstyle.sf.net/config_import.html> –>  <module name=”AvoidStarImport”/>  <module name=”IllegalImport”/>  <module name=”RedundantImport”/>  <module name=”UnusedImports”/> |

**File Size**

Include a methodLength check to ensure that files contain no more than 2000 lines and methods contain no more then 7 parameters.

|  |
| --- |
| <!– <http://checkstyle.sf.net/config_sizes.html> –>  <module name=”FileLength”/>  <module name=”ParameterNumber”/> |

**Whitespace**

As per the example below, set the tabWidth property to 4.

|  |
| --- |
| <!– <http://checkstyle.sf.net/config_whitespace.html> –>  <module name=”EmptyForIteratorPad”/>  <module name=”NoWhitespaceAfter”/>  <module name=”NoWhitespaceBefore”/>  <module name=”OperatorWrap”/>  <module name=”TabCharacter”/>  <module name=”WhitespaceAfter”/>  <module name=”WhitespaceAround”/> |

**Modifier order**

To avoid redundant modifiers on interfaces and annotations, follow the order of the Java Language specification, as per the following example.

|  |
| --- |
| <!– <http://checkstyle.sf.net/config_modifiers.html> –>  <module name=”ModifierOrder”/>  <module name=”RedundantModifier”/> |

**Block checks**

As per the following example, ensure block checks are applied so that there are no empty blocks, and all blocks are contained within braces

|  |
| --- |
| <\!-\- [http://checkstyle.sf.net/config\_blocks.html] \–>  <module name=”AvoidNestedBlocks”/>  <module name=”EmptyBlock”/>  <module name=”NeedBraces”/>  <module name=”LeftCurly”>  <property name=”option” value=”nl”/>  </module>  <module name=”RightCurly”>  <property name=”option” value=”alone”/>  </module> |