MuleSoft Best Practices

DRAFT

Capgemini

21-08-2017

Table of Contents

Double Click Here to Change Text

[1 Introduction 1](#_Toc491082127)

[1.1 Purpose 1](#_Toc491082128)

[1.2 Scope 1](#_Toc491082129)

[1.3 Intended Audience 1](#_Toc491082130)

[1.4 Lists of used terms and abbreviations 1](#_Toc491082131)

[1.5 Related Documents/References 1](#_Toc491082132)

[2 Design and Architecture 2](#_Toc491082133)

[2.1 Bottom-up approach 2](#_Toc491082134)

[2.2 Mule Flow Design 2](#_Toc491082135)

[2.3 API design 3](#_Toc491082136)

[2.4 Service design 3](#_Toc491082137)

[2.4.1 Service Proxy Pattern 4](#_Toc491082138)

[2.4.2 Service Selection Pattern 4](#_Toc491082139)

[2.5 Canonical Data Model 4](#_Toc491082140)

[2.6 Design for flexibility 5](#_Toc491082141)

[2.7 Exposing services 5](#_Toc491082142)

[2.8 Reliability Pattern 6](#_Toc491082143)

[3 Development 8](#_Toc491082144)

[3.1 Development Best Practices 8](#_Toc491082145)

[3.1.1 Peer reviews 8](#_Toc491082146)

[3.1.2 Source control 8](#_Toc491082147)

[3.1.3 Use dependency management tools 9](#_Toc491082148)

[3.1.4 Reproducible Builds 9](#_Toc491082149)

[3.1.5 Continuous Integration 10](#_Toc491082150)

[3.2 Mule Development 10](#_Toc491082151)

[3.2.1 Configuration File 10](#_Toc491082152)

[3.2.2 Parameters in Configuration Files 12](#_Toc491082153)

[3.2.3 Sharing Custom Code 13](#_Toc491082154)

[3.2.4 Coding best practices 14](#_Toc491082155)

[4 Performance stand point 16](#_Toc491082156)

[4.1.1 Keep application synchronous if possible 16](#_Toc491082157)

[4.1.2 Use flow references instead of VM endpoints 16](#_Toc491082158)

[4.1.3 Cache aggressively 16](#_Toc491082159)

[4.1.4 Configure message processors and endpoints at the global level 16](#_Toc491082160)

[4.1.5 Avoid creating a large volume of business events 16](#_Toc491082161)

[4.1.6 Use the async scope when appropriate 16](#_Toc491082162)

[4.1.7 Shared Resources 17](#_Toc491082163)

[5 When to create a new API and when to update the existing one- Scenarios 18](#_Toc491082164)

# Introduction

## Purpose

This document outlines the recommendations, best practices and guidelines for Architecting, Designing and Developing Enterprise Integration solutions based on MuleSoft and also describes the things to be considered from the Performance stand point.

## Scope

This document describes the MuleSoft ESB architecture, design and development best practices and things to be considered from the Performance stand point and intended for use by Capgemini. It does not cover all ESB implementation across all Capgemini subsidiaries and divisions, as each is governed independently and some may already have their own ESB/Integration foundation in place or underway. This document does not intend to be prescriptive to how to adopt MuleSoft, but instead establish what foundation must be in place to adopt MuleSoft based APIs and Services.

## Intended Audience

This document is intended for use primarily by architects, implementation teams, including developers, analysts from Capgemini who work on some aspect of the MuleSoft ESB. Specifically, enterprise architects, developers should be familiar with this document as ESB best practices should be included in their scope to assist in guiding MuleSoft ESB projects moving forward.

## Lists of used terms and abbreviations

| Abbreviation | Decription |
| --- | --- |
| ESB | Enterprise Service Bus |
| SOA | Service Oriented Architecture |
| ROI | Return on Investment |
| HA | High Availability |
| FO | Fail Over |
| DR | Disaster Recovery |

## Related Documents/References

| Document | Author/Location |
| --- | --- |
| MuleSoft User Guide | <https://docs.mulesoft.com/mule-user-guide/v/3.8/> |
| MuleSoft Best Practices | <https://www.mulesoft.com/whitepaper/integration-use-cases> |
| MuleSoft Development Best Practices | <https://www.mulesoft.com> |

# Design and Architecture

This section discusses the design principles and architectural concepts that can be applied to new integration projects.

## Bottom-up approach

MuleSoft recommends a bottom-up approach for creating new integration applications.

As one of the core components of a SOA stack, Mule ESB enables the creation and orchestration of services without requiring an application server or other infrastructure components.

A key aspect of the bottom-up approach is that the ESB enables service creation and orchestration. Naturally, the ESB will also perform its primary task: integration. So the ESB is really the centerpiece of the integration solution, providing an integration platform and an orchestration platform. Establishing a services model according to the bottom-up approach consists of these basic steps.

## Mule Flow Design

Services and compositions should be designed from the ground up, thus starting with the integration points to the actual business logic and/or services from legacy system, applications, databases etc.

It is recommended to create multiple flow files in Mule projects, containing the various endpoints, connector definitions etc. By breaking down a large, monolithic Mule application into smaller flow files, reusability is ensured. For example, create ‘common’ flow files for often-used functionalities, such as basic transformations, audit logging etc. Having multiple flow files in projects can be considered a best practice.

Keep new flows short and make sure that each flow follows single responsibility principle. For complex solution split the main configuration file (XML files containing flows) into smaller ones. Think in functionality and business logic terms.

Summarizing, the following design principles should be applied:

* Separation of logic: Implementation logic must be separated from integration and routing logic, both in Java code and Mule flow files. For example, a high level flow that is used for service orchestration should not contain any business logic or implementation logic.
* Break down flows: External invocations must be wrapped in services using Mule flows. Each invocation should be in a separate flow.
* Use a data or domain model: Business related data that is being sent between components, services etc., and should be captured in a canonical data model (XML or POJO). Message or routing related data should be stored in the metadata of the message payload.
* Design for efficiency: Mule flows files should be kept as small as possible. All reusable flow elements, or duplicate code, should be extracted into new flow files.
* Ensure reusability: Common functionalities are extracted to reusable flow files.
* Reduce coupling: Separate core business logic from base services, like data stores. Build services in layers and give users as much access as possible to each layer.
* Encourage modularity: Do not create needlessly complex services. Build simple services first, and only then build the complex orchestrations on top of them. Users will then have the option to use either.

Integration design principles should be applied

* Avoid point-to-point integrations: Build services by orchestrating other services, and build services that could be integrated together in the future. Reuse services whenever possible, and plan for services to be reused by others.
* Avoid hardcoding configuration values: Externalize all tunable values, and configuration parameters, to properties files. There should be one properties file to store parameters common to all environments, and a set of environment-specific property files.
* Use both synchronous and asynchronous integration: Know which integration pattern to apply in each case. Synchronous integrations will give more feedback, but asynchronous integrations will work faster and will work better when integrating slow services.
* Use the right constructs for the job: use flows for interactive services and batch processes to maximize throughput in cases where latency doesn't matter.
* Error handling: Anticipate possible errors in data, endpoint systems and connectivity. Build robust error handlers to recover from errors when possible. Have a Dead Letter Queue to send messages when errors are unrecoverable.
* Build APIs into existing integrations, to allow further expansion and interoperability. For new integrations, an API-first approach is recommended.
* Choose what to log: Always log failures and errors, use log aggregators (Splunk) when possible.
* Monitoring: Use a robust monitoring platform to stay ahead of CPU and memory shortages due to increased load. Plan ahead for load spikes.

## API design

Every API should meet a requirement and follow a common principle that brings everyone together.   
Here are some general best practices for making API’s effective and successful:

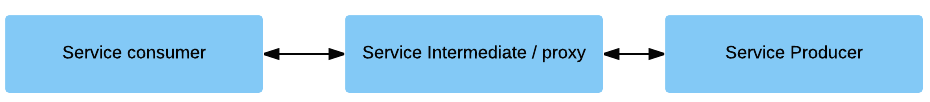
* Make it effective: Great APIs make products successful. If the API is not effective, it impacts the product or solution.
* Build it with user’s focused: Great API’s are designed with intent, not a reproduction of internal data. Therefore, use a top-down approach: design the API without keeping the underlying technology in mind. After the API design, implement the backend binding with Mule ESB.
* Be thoughtful: Get away from the anti pattern “we’ll expose our data and see what people do with it”. Instead while framing the design pour good thoughts into the possibilities of using the API
* Make it intuitive: API has to be intuitive, should follow principles of similarity across the board. Similar things should look similar. What users or consumers experienced in one place should behave the same all over.
* Simplicity: Do not push complexity to users or consumer, but try to wrap complex operations into a simple API.
* Documentation: Make sure the API is well documented, as it will be the starting points for most users.
* Definition: use RAML for API modeling. Do not start from the technology side, but define the API in a technology-agnostic way.

Below link can be referred for more details:

<https://www.mulesoft.com/resources/api/development-best-practices>

## Service design

When designing services, the concept of Service virtualization should be applied. Service virtualization is the abstraction of a physical service through a proxy or intermediary service. With service virtualization, the service proxy interacts with the service consumer (i.e. the service or application making a request) and hides the physical location of the service provider (i.e. the service or application that provides functionality). In other words, communication between the service provider and service consumer is done through a service proxy, creating a gateway for application integration.



### Service Proxy Pattern

The most basic service virtualization pattern is a simple service proxy that decouples the service consumer from the service provider by adding a layer of indirection. The service consumer interacts with the interface of the service intermediary, which is the same as the interface of the actual service provider. Because the service consumer only knows the location of the service intermediary and not the endpoint address of the service provider, this pattern of service virtualization is commonly used when services are invoked by consumers that reside beyond the enterprise, providing security and control over access to such services. It is also used to prevent disruptions when changes are made to a service provider (for example, moving a service provider from one server to another), allowing the service consumer to continue interacting with the service proxy without being impacted.

### Service Selection Pattern

In aservice selectionpattern, multiple implementations (or instances) of a service provider with the same interface are exposed to the service consumer at a common endpoint address. Each instance of the service provider provides a different level of service and the service intermediary routes a request to the appropriate endpoint based on the context of the service consumer’s invocation. In other words, the service consumer doesn’t know that there are multiple instances of the service provider and it is the service intermediary’s job to select the appropriate version based on the service consumer’s needs. This pattern can be used to distinguish service levels between different kinds of customers.

#### Service Translation Pattern

In aservice translationpattern, a service provider is made available through a service proxy with a different interface than the existing interface. This allows for only some, rather than all, functionalities to be exposed as well as the modification of an interface to meet the local requirements of a service consumer. The service proxy then translates the original request to meet the requirements of the existing interface before routing it to the service provider. This service virtualization pattern might be used when an enterprise teams up with a partner enterprise and decides to limit exposure of its services or discovers that the service consumers in its partner systems require data transformation for their different interface requirements.

## Canonical Data Model

Establishing canonical data models for key objects will be a critical part of building a foundation of reusable services. The concept basically is as follows: rather than mapping the PeopleSoft definition of a Student directly to the Blackboard definition of a student and also to each other system, the mapping should always be done to one, system-agnostic, master data model for each data object (such as a Student, Class, Person, etc.). The canonical definition of that data object will include all the attributes needed by any system so that all systems can properly map into that object. If one system changes or has a mapping change from an upgrade, all that needs to be done is modify the mapping for that one system to the common object.

Using a (canonical) data model allows the usage of metadata information without the need of creating flow or sessions variables. Also, when using many flows or subflows, the usage of flow/sessions variables should be as minimal as possible (it will actually result in better performance).

## Design for flexibility

When designing and structuring Mule applications, it’s important to consider the frequency with which parts of the solution will need to change and grow. Failure to do so may lead to monolithic applications that are extremely hard to manage and troubleshoot. The benefit of partitioning flows and services that may change or grow gives much more deployment flexibility should an architecture need to change to result in increased load or a change in requirements.

Example: We have a requirement to expose a service via HTTP endpoint, which needs to perform complex validation logic (on HTTP Headers, Request Body etc.), then transform the provided data into a proprietary structure, backing it up with data coming from external REST Service. Eventually send a notification email to anyone who it may concern.

The main flow can accommodate the whole application process into few single flow references, which can be easily read and understood. Every of the bigger, complex tasks are extracted to a different configuration file. With such a structure we can execute end-to-end tests or test only main processing paths of full solution (when flow has choice components as presented) by mocking out all referenced implementations.

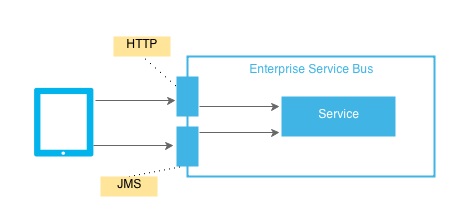
Breaking down flows into smaller subflows will lead to more readable code, a modular approach and better testing possibilities using the MUnit framework. Also, modular programming limits the impact of version changes on your tested code, such as when you upgrade a particular Connector.

Each of that configuration files is responsible for different phase of the solution. We can do the same with them if the flows would get bigger: split into more compact flows / subflows or even into another configuration files if necessary. At some point in time, it may be needed that part of the application should be extracted into one or more separate Mule applications. Keeping flows modularized makes this task relatively easy.

## Exposing services

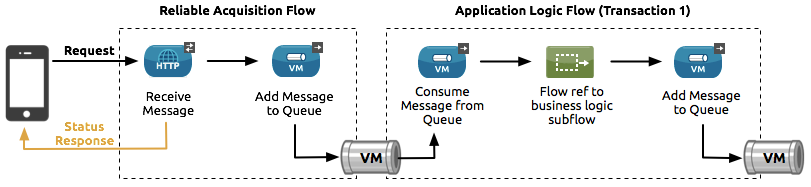
In order to facilitate the modular approach and maintain a focus on performance and resilience, the following messaging approach is implemented:

* It is recommended that all external interfaces/endpoints are exposed through more than on one protocol, for example both JMS and SOAP/HTTP(S)/REST.



## Reliability Pattern

A reliability pattern is a design that results in reliable messaging for an application even if the application receives messages from a non-transactional transport. A reliability pattern couples a reliable acquisition flow with an application logic flow, as shown in the following diagram.



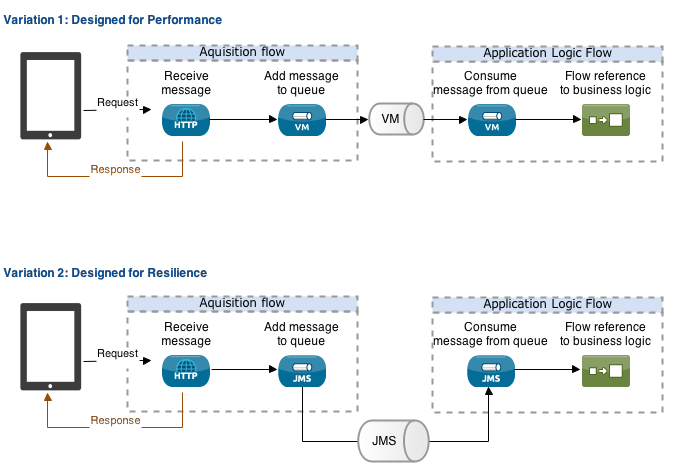
The reliable acquisition flow (that is, the left-hand part of the diagram) delivers a message reliably from an inbound endpoint to an outbound endpoint, even though the inbound endpoint is for a non-transactional transport. The outbound endpoint can be any type of transactional endpoint such as VM or JMS.

If the reliable acquisition flow cannot deliver the message, it ensures that the message is not lost:

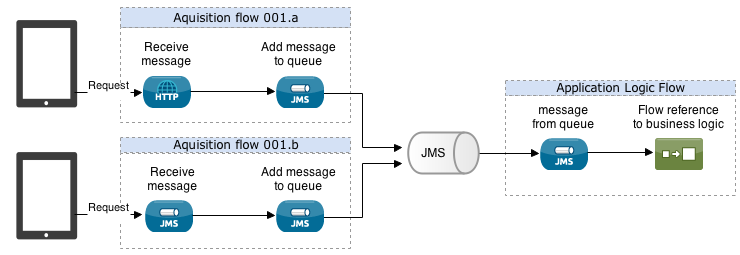
* For socket-based transports like HTTP, this means returning an "unsuccessful request" response to the client so that the client can retry the request.
* For resource-based transports like File or FTP, it means not deleting the file, so that it can be reprocessed.

The application logic flow (that is, the right-hand side of the diagram) delivers the message from the inbound endpoint (which uses a transactional transport) to the business logic for the application.

Based on requirements, there are two possible implementations of this pattern, one with a focus on performance, and another implementation with a focus on persistence. The images below describe both variations of the pattern.



As a matter of fact, this pattern makes it easier to implement the example standard that dictates that all front-facing services must be exposed through SOAP/REST and JMS. The image below shows a simplified integration flow that incorporates both principles:



# Development

## Development Best Practices

Development phases includes:

* Acceptance Criteria: All acceptance criteria for the requirement have been met. The acceptance criteria for the user requirement must be demoed and reviewed by the Product Owner and/or ScrumMaster for acceptance. The user requirement can only be closed by the Product Owner and/or ScrumMaster for the team.
* Supporting Documentation: Each Work record must have supporting comments documenting what work was performed and/or decision that was made. Support documents or reference links must be attached to the Work record. For example, technical specification, functional requirements, links to external resources.
* JavaDocs: All code must have well defined JavaDoc comments, which can be incorporated.
* Meaningful names in Mule flows and elements: Elements of Mule applications should have names and labels that reflect their purpose, rather than their type.
* Automated tests: The code must include a set of automated Unit or Integration tests that exercise the different code paths. Tests provide 100% code coverage.

### Peer reviews

All committed code should undergo peer review; either by making all changes goes through an approver or through regular all-team meetings.

In addition to letting other developers inspect newly checked-in code, peer review meetings facilitate the adoption of coding standards and design patterns across a team, by letting all developers discuss advantages and disadvantages of specific approaches to problems.

### Source control

Source control is one of the most commonly used best practices in the industry today. Mule's projects can be checked into any standard source control system, including CVS, SVN, Git and Github among others.

#### What to check in, what to ignore

Not all files in a Mule project need to be checked into source control system. Some files are metadata generated by Studio and other tools in order to keep track of the project; others are temporary information generated by the build process and are completely machine-dependent. Checking in these files will cause conflicts in other team members' projects, or will cause their builds to fail.

Check in these files and folders:

* src/ folder (source code).
* mappings/ folder (Data Mapper maps).
* The mule-project.xml file (Studio project descriptor).
* pom.xml (for Maven-based projects, the project's build configuration).

Do NOT check in these files and folders. They are either temporary or studio metadata:

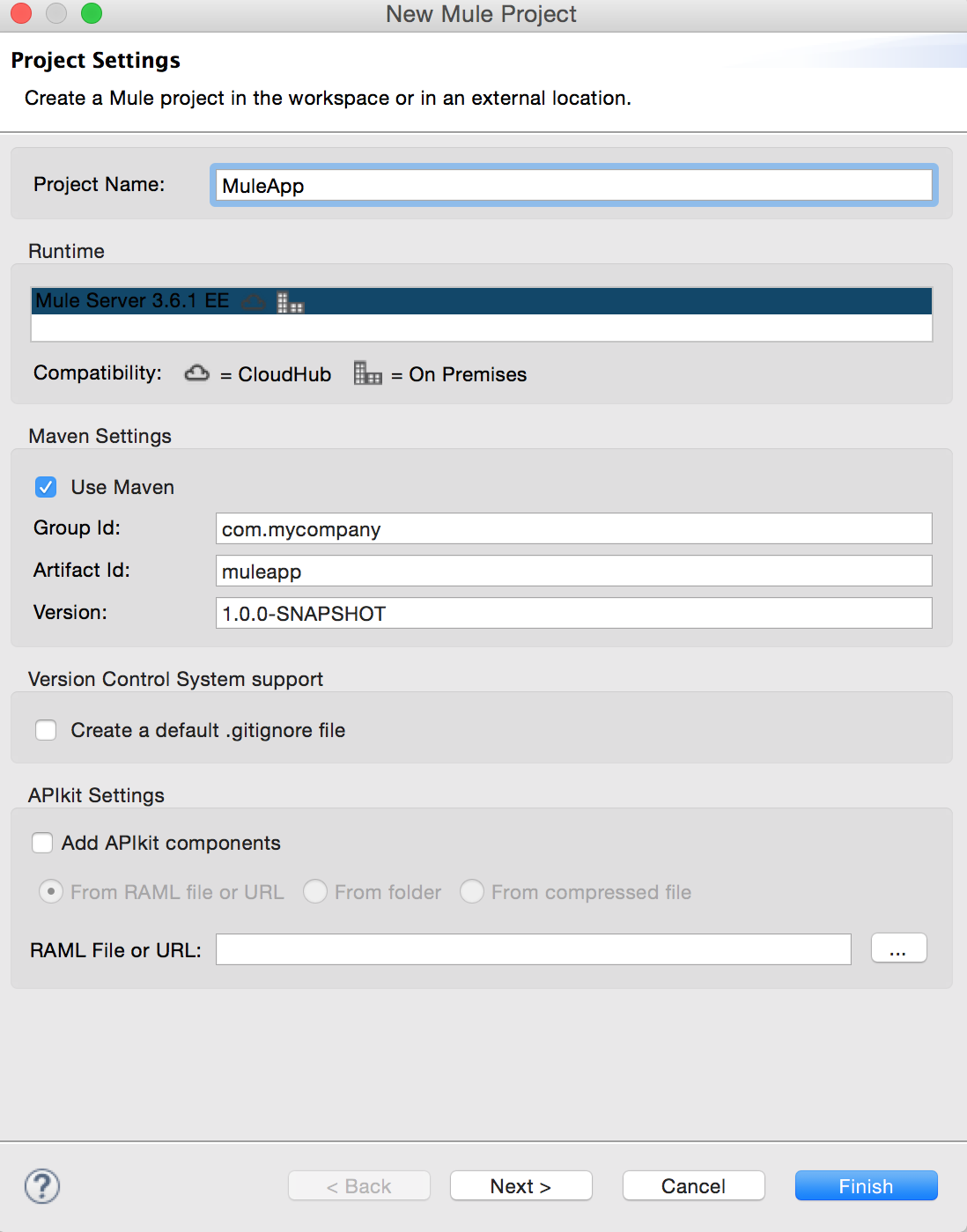
* .classpath (Studio metadata)
* .settings (Studio metadata)
* target/ folder (contains compiled code)
* bin/ folder (contains compiled code)

If Git is being used for source control management, it is possible to create a file called “.gitignore”. In this file, all assets that are to be ignored and should be not checked in, can be defined here.

### Use dependency management tools

Most development projects use third-party libraries. MuleSoft officially supports and recommends Maven for dependency management in all Mule projects. Studio can create and maintain your project's pom.xml file if you select “Use Maven” when creating your Mule project.

Studio will then create a base pom.xml, configure it to use Mule's libraries and synchronize all dependencies and dependency resolution between the Studio project and the pom.xml.



### Reproducible Builds

We should be able to build a particular version of Mule project at any point of time. This facilitates maintenance, as the versions of projects in production will certainly not be the latest ones.

The following will help achieve this goal:

* Source control all Mule projects, like any other project. Branching, merging and tagging are all applicable practices to Mule projects
* Manage dependencies strictly: using Maven and an in-house repository manager (like Github).

### Continuous Integration

Setting up continuous integration for Mule project is highly recommended. By using Maven as primary build tool, we will be able to set-up a build that gets triggered on every project change and run all its unit and functional tests automatically.

There are plenty of continuous integration tools out there. To name a few: Jenkins, TeamCity and Bamboo are popular choices.

If you've opted for using real transports in functional test cases, we will have to pay attention to potential port conflicts that could occur in continuous build server.

## Mule Development

When working on a multiple project, the Mule projects will increase in size, will increase in its number of developers, and must run in other environments, such as test and production. Here are some practices that will make such growth possible.

The different approaches to modularizing Mule configurations and applications are all opportunities for splitting work across teams, whether these teams work on the same overarching project or on different projects with an accent put on reuse.

### Configuration File

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 we should be careful and use unique names for all configuration elements. Mule will refuse to load an application whose configuration files contain name conflicts.

How can we 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
* And, last but not least, re-use across teams and projects

#### Import configuration files

Mule relies on Spring XML configuration for importing configuration files.

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

|  |  |
| --- | --- |
| |  | | --- | | <mule xmlns="        xmlns:xsi="        xmlns:spring="    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>>    <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>  ... | |

#### Using Side-by-Side Configuration Files

Side by side configurations are independent and require nothing specific to work except to let Mule know it should load them.

To do so, create a file named mule-deploy.properties in the same directory that contains configuration files and add configuration similar to the following one, but of course with the app configuration file names:

|  |
| --- |
| config.resources=mule-main-config.xml,mule-sub-config.xml |

With this in place, Mule will know which configuration files it should load when deploying your application.

If you're starting your application from Eclipse, go to the parameters screen of the "Run" configuration you use and select your files there.

### Parameters in Configuration Files

When an application gets deployed in different environments, like QA, pre-production or production, it usually needs to be configured differently as server names, credentials and other similar parameters will vary.

As a developer our goal is to produce a single Mule application for all your environments and to externalize all the environment-specific configuration parameters. This is the key to reproducible deployments.

Consider externalizing other aspects of configuration, like time-out values, polling frequencies, etc. Even if they don't vary between environments. This will facilitate tuning, as the whole Mule application would become configurable through a single properties file.

#### Spring's property placeholder

Consider the following Mule configuration fragment that defines an HTTP endpoint pointing to a password protected web resource:

|  |
| --- |
| <http:endpoint name="HTTPResource"  user="${http.rsc.user}"  password="${http.rsc.password}"  host="${http.rsc.host}"  port="${http.port}"  path="path/to/resource" /> |

The variable bits are clearly visible: the user, password and host can vary for each environment where this endpoint gets deployed in. To provide values for these variables, use a standard Java properties file:

|  |
| --- |
| http.rsc.user=mule  http.rsc.password=welcome123  http.rsc.host=www.mulesoft.com  http.port= 80 |

Use a consistent naming strategy for properties and make them unique across applications: this will greatly facilitate re-use across teams.

#### Springs override property placeholder

Instead of configuring spring to load a single properties file we can create override property placeholder to override the properties, follow below approach:

* Configure Spring to load a default properties file and another file containing overrides.
* Ship a default properties file with values applicable for developers' workstations inside Mule application deployable.
* Create the properties override file only in the environments where it's needed and with only the properties that actually need to be overridden.

Advantages of this approach:

* Developers don't need to deploy and run the application locally.
* The ops team only needs to work with the set of properties they have to configure for a particular environment.

Example:

|  |
| --- |
| <mule xmlns="http://www.mulesoft.org/schema/mule/core"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:spring="http://www.springframework.org/schema/beans"  xmlns:context="http://www.springframework.org/schema/context"  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  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context-3.0.xsd">  <spring:beans>  <context:property-placeholder  location="classpath:my-mule-app.properties,  classpath:my-mule-app-override.properties" />  </spring:beans>  </mule> |

With this in place, add a my-mule-app.properties file in application resources directory (src/main/resources for a Mule application Maven project) and put default and development environment values in it. To override some values, create a my-mule-app-override.properties file and drop it in $MULE\_HOME/conf.

If ops team can't drop files in Mule's directory hierarchy, the alternative is to configure the placeholder to pick up the override file from a well-known location, as shown here:

|  |
| --- |
| <context:property-placeholder  location="classpath:my-mule-app.properties,  file:///var/mule/conf/my-mule-app-override.properties" /> |

Use unique file names for properties files to ease the burden on ops team. A good strategy is to use the application name or ID in the default and override properties file names.

Should you need to encrypt passwords in your properties file, consider using the Mule Credentials Vault. Refer to the documentation for Anypoint Enterprise Security for more information on securing applications in Mule.

### Sharing Custom Code

Besides all the common code that exists in a company, there is Mule specific programmatic artifacts that are worth considering sharing.

Example:

* 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.

The most convenient way to share custom code across team is to rely on Maven's dependency management mechanism. Here is an extract of a pom.xml referring to common code stored in a shared Maven library:

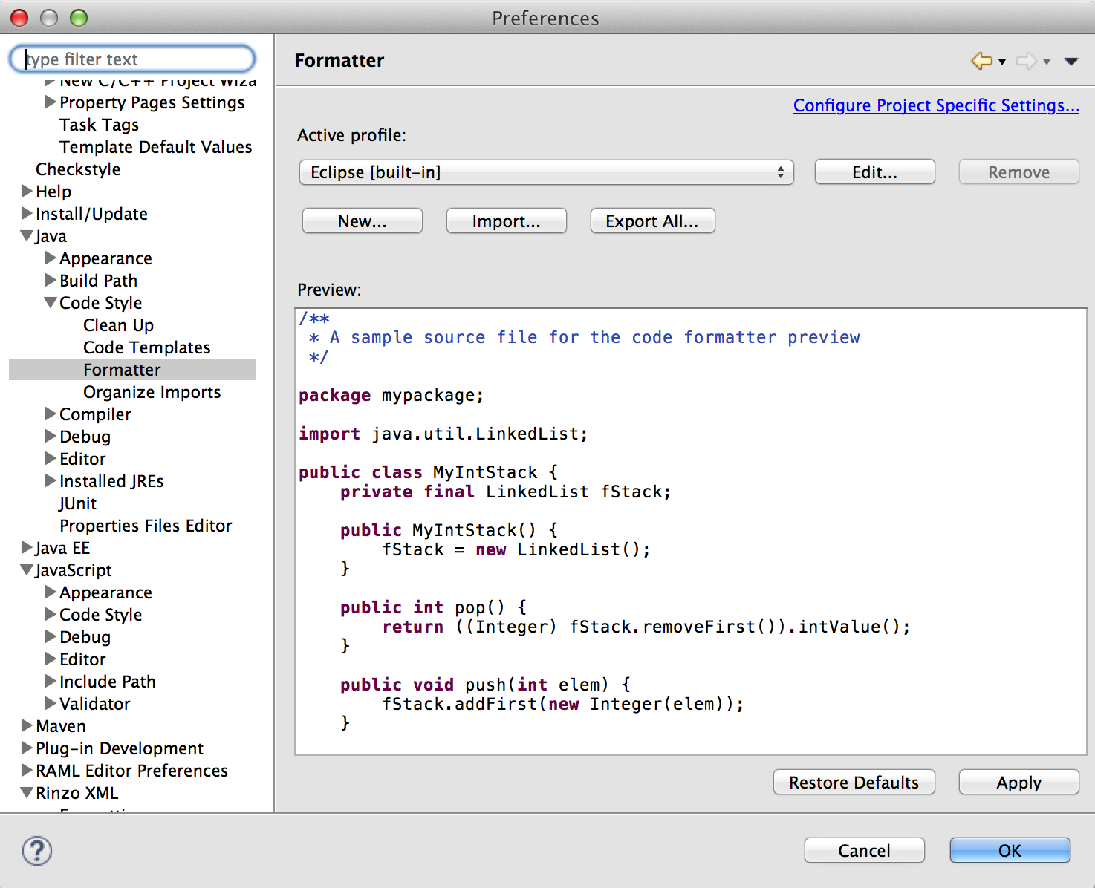
|  |
| --- |
| <dependency>  <groupId>com.mulesoft </groupId>  <artifactId>common-mule-project</artifactId>  <version>1.2</version>  </dependency> |

The Mule build plug-in will automatically bundle these extra dependencies in the /lib directory of the deployable application. In this case, the common-mule-project-1.2.jar will be added to this directory at build time, ready to be deployed and made available to the application running on Mule.

### Coding best practices

#### Code formatting

Mule Studio ships with a pre-configured formatter for both Java code and Mule.

Java → Code Style → Formatter 

#### Coding standards

In a typical development project, names have to be given to many different constructs: from the Mule Application itself, to individual flow files, assets, and message processors.

Every tool and language has its own naming conventions.

Here are some conventions on how to format names for different entities in Mule applications:

##### Application name

##### Dashes-between-words: - <client>-<project>-<application>-<esb/api> (e.g. Capgemini-peoplesoft-personpub-esb)

##### Flow (xml) files

##### Dashes-between-words: -<purpose>.xml (e.g. main-flow.xml/ /Capgemini-person-salesforcesub-esb-flow.xml)

##### Flow and subflow names

Dashes-between-words (e.g. Get-Person-Affil-Count-flow)

##### Message Processor name

camelCase (e.g. convertPayloadToJson)

##### Java bean names

camelCase (e.g. <spring:bean name="myBean" />

##### DataMapper's map definitions

Dashes-between-words (e.g. pspersontransaction-to-personcdm.grf)

##### Other resource file names:

Dashes-between-words**: -** <client>-<requirement>-api.raml (e.g. Capgemini-GetEmployee-api.raml)

As a more general rule: use camelCase for java-related entities (Classes, Beans, Message Processors) and dashes-between-words for Mule-related entities (flows, message queues, resources)

By applying these naming rules all your applications will be more consistent and maintainable.

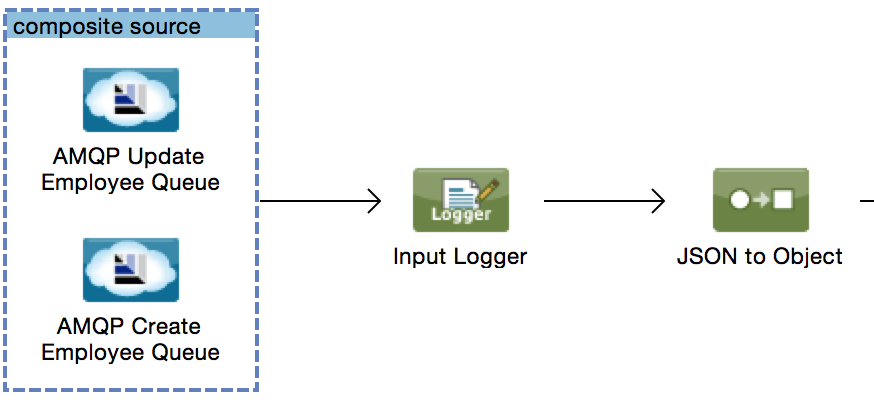
Refer the following document for more information –

<https://docs.mulesoft.com/mule-contributors-guide/v/3.4/mule-coding-conventions>

<https://docs.mulesoft.com/mule-contributors-guide/v/3.4/xml-coding-style-guide>

#### Logging

As mentioned earlier, services should be wrapped in mule flows, and service compositions consist of Mule flows as well. In practice, there will be many levels of subflows. As a best practice, logging should be applied after each subflow or service calls to log/check the results of the subflow. Furthermore, make sure to apply log level “debug” to the log statements, in order to prevent “log pollution”.



# Performance stand point

### Keep application synchronous if possible

Synchronous flows avoid serialization/deserialization of messages sent through VM queues, do not cause context switches, and do not cause contention when messages move across thread pools.

### Use flow references instead of VM endpoints

To communicate between flows internally within an application, use flow references instead of VM endpoints. The VM connector, even though it is an in-memory protocol, emulates transport semantics that serialize and de-serialize parts of your messages, most notably the Session scope. This makes it slower than a flow reference, which just injects messages into the referenced flow with no intermediate steps.

### Cache aggressively

Take advantage of Mule’s caching scope when making requests to external resources like Web services or databases. Also consider caching reusable assets such as security tokens or ephemeral API keys and cookies.  Mule’s Notification subsystem can additionally be used to “warm up” a cache when Mule starts. For example, consider doing this for situations where a initial cache miss is not acceptable.

### Configure message processors and endpoints at the global level

Some connectors allow you to configure some parameters at both the global and the endpoint/message processor level. We recommend placing the configuration at a global level to avoid repeated initialization of resources. The MongoDB connector, for instance, will bypass internal connection pooling if usernames/passwords are configured on an individual message processor as opposed to the global connector configuration.

### Avoid creating a large volume of business events

Business events incur performance overhead in Mule and in MMC when MMC’s internal event buffer overflows. Thus, avoid using either default flow-level business events or a large volume of custom business events in a high message volume project.

Business events store the information on the Mule DB. Follow the below best practices.

* Correctly size the database that will be used to store events. Mule stores event-related data in a default database, which persists the data. However, we also have the option to persist the data in an Oracle database. This can give control over how much data to store. It can also provide better scalability and performance than the default data storage.
* Fine-tune the cleanup script. The management console provides a cleanup script as part of the Admin Shell in the Administration tab. The script periodically cleans old data from the tracking database. By default, the script runs once a day and cleans all data from the database that is older than one week. We can customize the cleanup script so that it specifically meets our requirements.

### Use the async scope when appropriate

If a flow is performing processing on a message that is neither modifying the message nor changing how it is routed, then it could be wrapped that in an async block.  This will cause the processing to occur in a different thread and will avoid adding unnecessary overhead to processing the message.

### Shared Resources

Share the same connector (JMS, JDBC, HTTP, VM are supported) across multiple applications within the same Mule ESB instance; consider using the new Shared Resources feature available in Mule ESB 3.5.x.

Follow below documentation:

<https://docs.mulesoft.com/mule-user-guide/v/3.8/shared-resources>

Note: For sensitive information we may not want to share the resource pool

# 5 When to create a new API and when to update the existing one- Scenarios

A few of the many required characteristics listed below. Need to work more on it to come up with the scenarios for the particular customer business we are looking this for. We can modify/update the list going forward.

If the existing API has below basic characteristics and attributes then we may think of exploring more so that we come to a conclusion to update the existing API instead of creating a new one.

1. Existing API endpoints should be simple and provide parameters to support a wide range of use cases required for the new API requirement.
2. Existing API endpoints should be consistently structured for SQL, NoSQL and file stores.
3. New API may require to handle high transaction volume so the existing API must have the capability for high transaction volume.
4. Requests and responses should include JSON or XML with objects, arrays and sub-arrays
5. All HTTP verbs (GET, PUT, DELETE, etc.) need to be implemented for every use case.
6. Support for web standards like OAuth, CORS, GZIP and SSL is also important and should be checked.