

Medtronic

**MuleSoft Development Best Practices**

**v1.1**

Document Control Information

|  |  |
| --- | --- |
| Document Identification |  |
| Document Name | MuleSoft Development Best Practices |
| Project Name |  |
| Client | Medtronic |
| Document Author | Kishore Reddy Paluri |
| Document Contributors | MuleSoft Professional Services Team |
| Document Version | 1.1 |
| Document Status | Draft |
| Date Released |  |

Document Edit History

| Version | Date | Additions/Modifications | Prepared/Revised by |
| --- | --- | --- | --- |
| 1.0 | 25-May-2020 | Initial Draft | Kishore Reddy Paluri |
| 1.1 | 01-Jul-2020 | Added API Taxonomy | Kishore Reddy Paluri |

Document Review/Approval History

| Date | Name | Organization/Title | Comments |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[1. Overview 4](#_Toc44500698)

[2. Anypoint Studio 5](#_Toc44500699)

[2.1 Studio Updates 5](#_Toc44500700)

[2.2 Mule Runtime 5](#_Toc44500701)

[2.3 Mule Projects in Anypoint Studio 5](#_Toc44500702)

[2.3.1 Mule Projects in Anypoint Studio 5](#_Toc44500703)

[2.3.2 Project File Structure 6](#_Toc44500704)

[2.3.3 XML Indentation and Formatting 8](#_Toc44500705)

[2.3.4 Secure Application Properties 9](#_Toc44500706)

[2.4 Naming Conventions 9](#_Toc44500707)

[2.4.1 Naming Snapshot 9](#_Toc44500708)

[2.4.2 Application Naming Conventions 9](#_Toc44500709)

[2.4.3 Project Naming for CloudHub Deployments 10](#_Toc44500710)

[2.4.4 API Versioning 10](#_Toc44500711)

[2.4.5 Mule Configuration Files 11](#_Toc44500712)

[2.4.6 Testing 11](#_Toc44500713)

[2.4.7 Anypoint Exchange 11](#_Toc44500714)

[2.4.8 Properties Per Environment 11](#_Toc44500715)

[3. Mule Components 13](#_Toc44500716)

[3.1 Dataweave 13](#_Toc44500717)

[3.2 HTTP(s) Listener 13](#_Toc44500718)

[3.3 HTTP Request 14](#_Toc44500719)

[3.3.1 Request to APIs or HTTPs endpoint with self-signed certificates 14](#_Toc44500720)

[3.3.2 HTTP Request Status Code Validator 14](#_Toc44500721)

[3.4 Web Service Consumer 14](#_Toc44500722)

[3.5 Database Connector 14](#_Toc44500723)

[3.6 APIKit 15](#_Toc44500724)

[3.7 Validator 15](#_Toc44500725)

[3.8 Flow Reusability 15](#_Toc44500726)

[3.9 Java Custom Coding/Scripting 15](#_Toc44500727)

[3.10 Unit Testing 15](#_Toc44500728)

[3.11 Mule Deploy Maven Plugin 16](#_Toc44500729)

[3.12 Debugging & Trouble-shooting 16](#_Toc44500730)

[3.13 Wire Logging 16](#_Toc44500731)

[4. Error Handling in API-led Connectivity – Best Practices 17](#_Toc44500732)

[4.1 API-led Connectivity 17](#_Toc44500733)

[4.2 REST API Error Handling 18](#_Toc44500734)

[4.2.1 HTTP Status 18](#_Toc44500735)

[4.2.2 Error Types 18](#_Toc44500736)

[4.2.3 Error Response 19](#_Toc44500737)

[5. API Taxonomy 20](#_Toc44500738)

# 1. Overview

The following guide is intended to set and describe the best practices and standards to apply in the implementation phase of the API Development Life Cycle.

# 2. Anypoint Studio

## 2.1 Studio Updates

* Keep your Anypoint Studio up-to-date (Install Studio Updates when available).
* Use the recommended Java version for the specific Studio version.

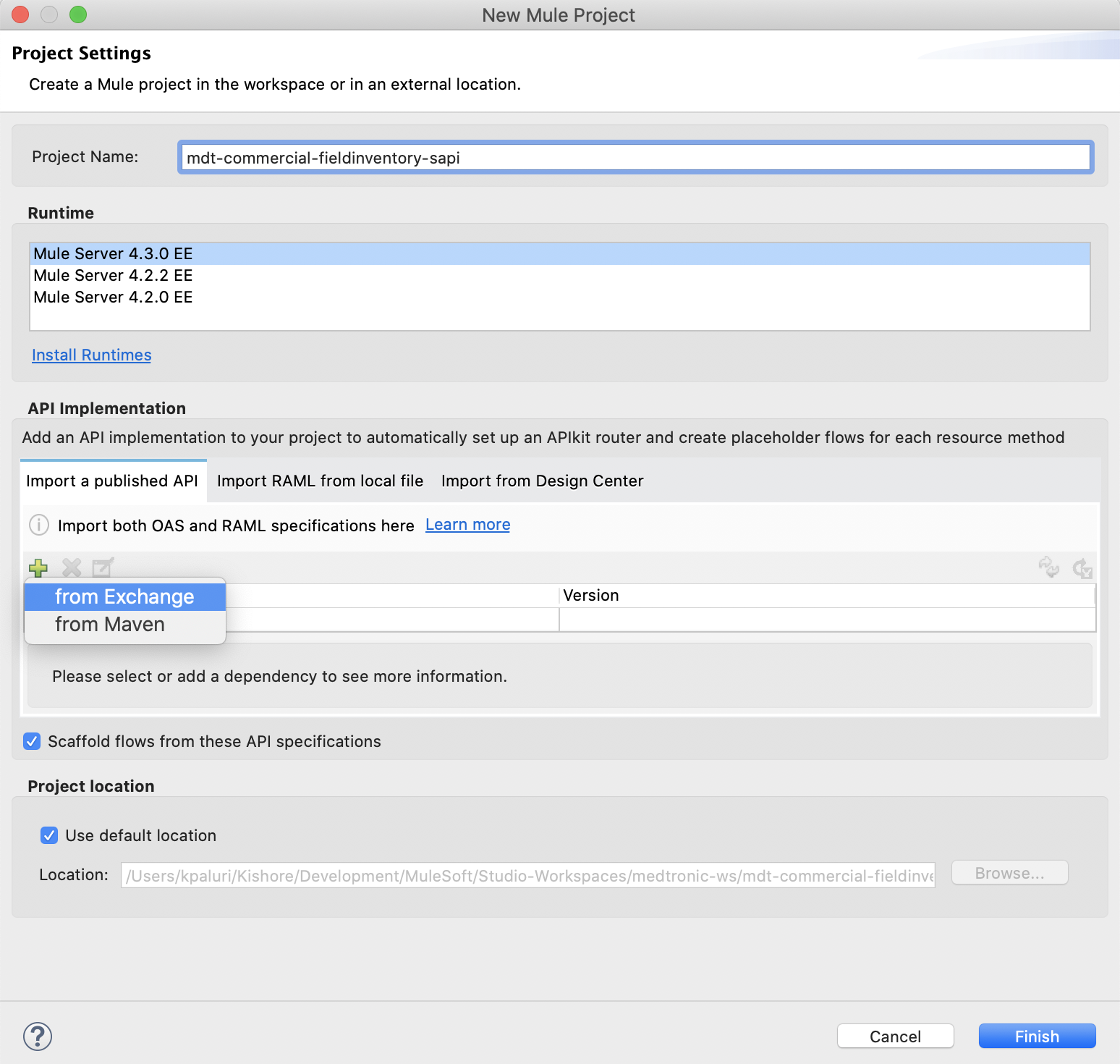
## 2.2 Mule Runtime

* It's really important the coherence of the Mule Runtimes version within the SDLC. If you developed and tested your application using a specific Mule Runtime version, deploy it to a worker with the same Runtime version
* Upgrade your projects along with rest of the team to the latest patch-version (Install the new Runtimes from the Studio Update Site or from the Mule Runtime Update Site)
* After updating mule runtime inside Studio, it is the developer’s responsibility to retest all test cases using the updated mule runtime before deploying to the target mule runtime with the same (updated) mule runtime version
* If you want to migrate your applications to a minor-new-version or a major-new-version follow the migration guides provided in the MuleSoft public documentation <https://docs.mulesoft.com/release-notes/>

## 2.3 Mule Projects in Anypoint Studio

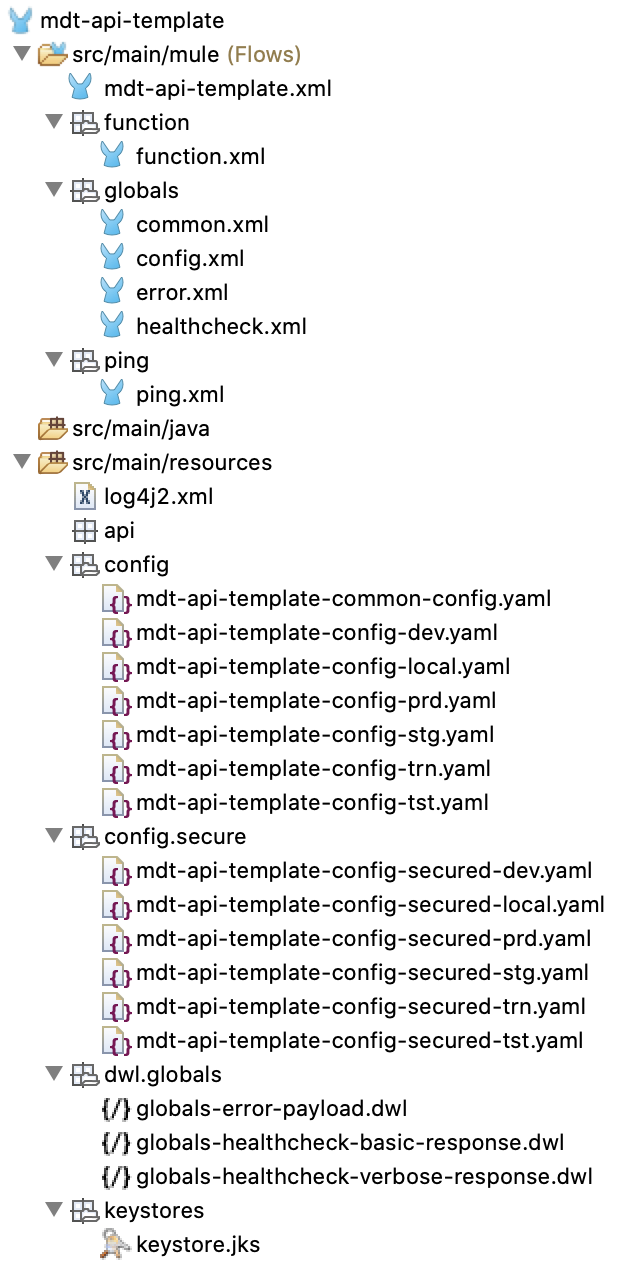
### 2.3.1 Mule Projects in Anypoint Studio

• Add APIKit components getting the RAML from the Anypoint Exchange when developing APIs.



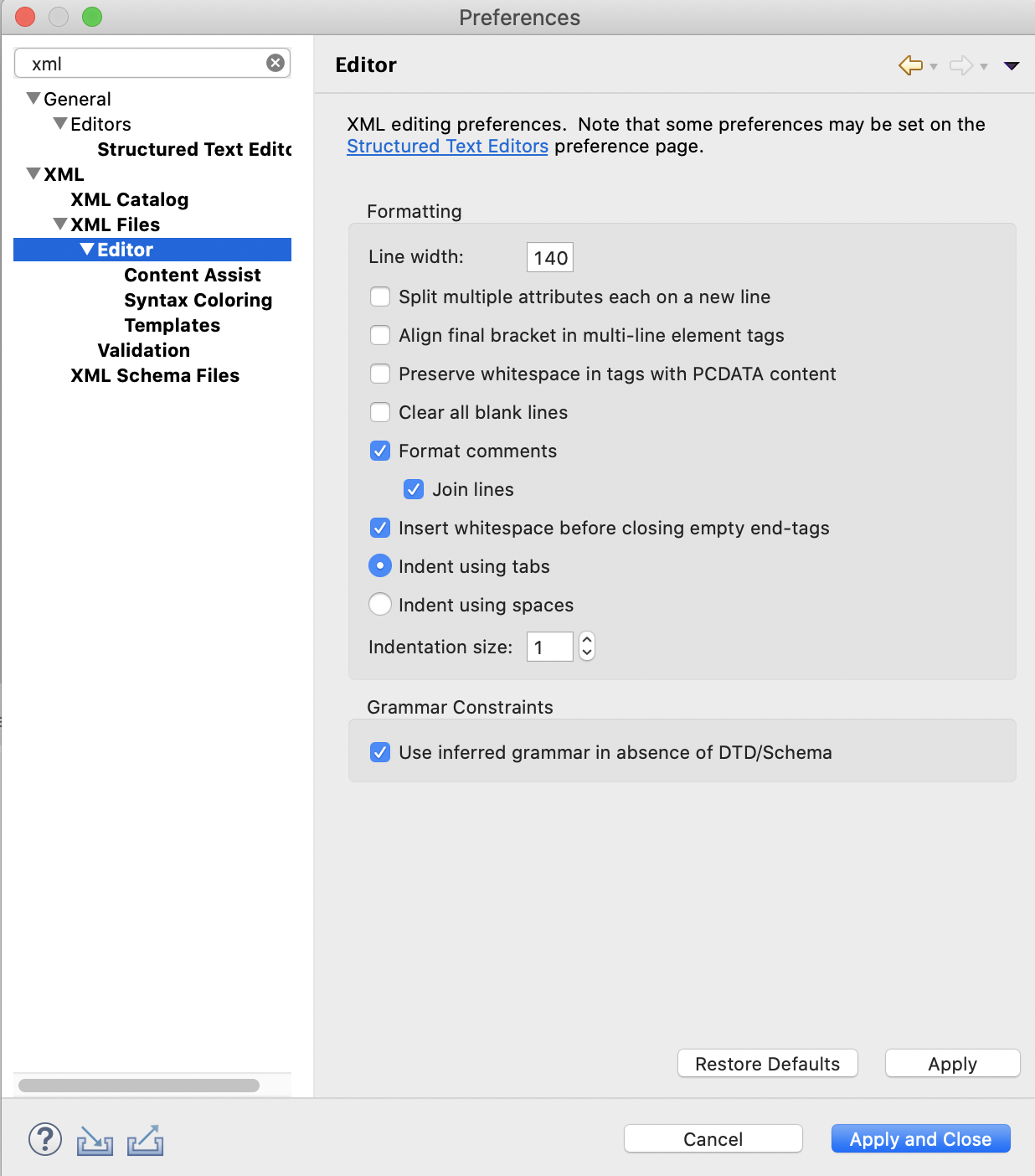
### 2.3.2 Project File Structure

* Create a separate xml for the global elements (e.g. configuration elements)
* Create a separate xml for each use-case or resource-implementation
* Create separate xmls for common structures/logic
* Create different packages for the resources (dataweave, wsdls, examples, etc)



### 2.3.3 XML Indentation and Formatting

Indent all your xml (mule xmls, pom.xml, log4j2.xml, etc) files before committing to the source code repository



### 2.3.4 Secure Application Properties

Sensitive application properties such as access credentials and passwords can be obfuscated in CloudHub by using the Secure Application Properties feature.

Such feature can be used in conjunction with the Mule Credentials Vault that allows users to encrypt properties within the application properties files.

Use AES algorithm with a CBC mode: Define a 256 bits key.

For more information:  
• <https://docs.mulesoft.com/mule-runtime/4.3/secure-configuration-properties>

## 2.4 Naming Conventions

### 2.4.1 Naming Snapshot

|  |  |  |
| --- | --- | --- |
| **Category** | **Pattern** | **Examples** |
| GitHub Repositories | lower case, hyphen | mdt-commercial-customers-xapi |
| Mule Projects | lower case, hyphen | mdt-commercial-customers-xapi |
| Mule xml files | lower case, hyphen | global.xml |
| Flow name | camel case, prefixed with folder fqn | customers:getCustomersFlow |
| Sub flow name | camel case, prefixed with folder fqn | customers:getRefIdSubFlow |
| Mule properties files | lower case, hyphen and prefixed with app name | mdt-commercial-customers-xapi-config-dev.yaml |
| Mule properties keys | lower case, separation with dots and prefixed with folder fqn | customers.api.name |
| Dataweave Scripts | lower case, hyphen and prefixed with folder fqn | customers-response-payload.dwl |
| Variable name | lower camel case, prefixed with var | varOrderType |

### 2.4.2 Application Naming Conventions

The general naming structure for projects is as follows, which depends on the type of project and where it resides in the stack:

<enterprise-initials>-<domain>-<sub-domain>-<type-of-app> Explanation of structure:

1. Enterprise initials – (Mandatory) All application names should be prefixed with initials of the enterprise
2. Domain – (Mandatory) The domain to which the project
3. Sub-domain – (Mandatory) The entity or consumer your project represents
4. Type of project – (Mandatory) Indicates whether it is an API and the type of API. To

be consistent use ‘sapi’ for system APIs, ‘papi’ for process APIS, and ‘xapi’ for experience APIs. You may have other types of projects like batches that you might want to categorize and come up with a convention for consistency.

Example:

* **Experience API**: mdt-commercial-web-xapi
* **Process API**: mdt-commercial-customers-papi
* **System API**: mdt-commercial-fieldinventory-sapi
* **Batch**: mdt-commercial-customers-reconcillation-process

### 2.4.3 Project Naming for CloudHub Deployments

An additional consideration for project naming is when it comes to deploying applications to CloudHub (NOTE: you would physically name your projects as previously indicated, this naming would just be for the name of the CloudHub deployment of the given API). All project names must be unique in CloudHub, so it is recommended to append the environment to the end of the project name, with the exception of your production deployments, which will adhere to the naming structure already discussed. This project naming convention for deployments should be adhered to in your CICD/deployment strategy.

Examples of project naming for applications deployed to CloudHub, given a project name of ‘mdt-commercial-fieldinventory-sapi’:

Development environment: mdt-commercial-fieldinventory-sapi-dev (suffixed with ‘-dev’)

Test environment: mdt-commercial-ecommerce-xapi-test (suffixed with ‘-tst’)

Staging environment: mdt-commercial-ecommerce-xapi-stg (suffixed with ‘-stg’) Production environment: mdt-commercial-ecommerce-xapi (no suffix for production)

### 2.4.4 API Versioning

It has been agreed upon that APIs will be versioned via use of the RAML and URLs within the RAML. For example if you were creating a customers API, the base resource would be /customers and for the first version of the API endpoints, you would use /v1 and then after that provide your additional endpoints. When modifying endpoints that would require a new version, add the next version in the RAML (Ie; /v2) and provide the implementations for that version beneath the new version.

Here is an example of this structure:

/customers:

/v1:

/endpoint1:

/endpoint2:

/endpoint3:

### 2.4.5 Mule Configuration Files

Each Mule application should contain a Mule configuration file with all of your application configurations in it (such as database configurations, ApiKit configurations, property files, etc..). These files are sometime called either global.xml or mule-config.xml.

Keep your Mule configuration files and flows relatively small and for a specific purpose. Think of your Mule configuration files as classes and the flows within as methods. Often times when MUnit testing is being done you will find that you need to refactor your Mule flows.

### 2.4.6 Testing

Always unit test your code. Write tests as much as you possibly can, MUnit is a powerful tool that can reduce coding errors if used to its’ full extent. Some recent documentation on MUnit can be viewed at this [link](https://docs.mulesoft.com/munit/latest/).

### 2.4.7 Anypoint Exchange

All APIs should be published and documented in Anypoint Exchange. Please refer to the Anypoint Exchange strategy document that has been provided to the team.

### 2.4.8 Properties Per Environment

It is better to over-configure your applications than have to modify your code when a property needs to be changed. These configurations would exist in a property file for your applications. It is most common to just have one property file per Mule application per environment. Secure any properties that contain sensitive data.

Create a folder called ‘config’ under src/main/resources. Ie; src/main/resources/config.

Put all of your property files inside of the folder mentioned above.

Your property file(s) should be referenced something like as follows: /config/<folder- fqn>-config-<usecase>-${mule.env}.yaml (however your environment variable is configured and is suggested to just use ‘mule.env’). For local development, this would default to the ‘local’ property file. There would be an environment variable of the same name on the Mule servers with an environment specific value or injected into your CloudHub deployment.

For [secure properties](https://docs.mulesoft.com/mule-runtime/4.3/secure-configuration-properties), follow a similar naming convention, with secure in the file name. /config/<folder-fqn>-secure-config-${mule.env}.yaml. When configuration ‘local’ secure files, use an environment variable called ‘mule.key’ as the key to encrypt/decrypt properties. All developers should agree on this value for local property files. For property files for higher regions, have an admin generate these secure properties with a key that will get injected via the CICD build.

# 3. Mule Components

## 3.1 Dataweave

* To transform/enrich the data, dates transformation and build error/response messages instead of custom scripting code (Java, Groovy, etc.)
* When dealing with large payloads, include the directive indent=false to improve the client's parsing performance and to reduce the response payload size.
* Define the input content-type to avoid verbose messages in the Log e.g.

<ee:transform doc:name=*"set verbose response payload"* doc:id=*"5f899d85-fd1e-4344-8ef1-cc207de93c45"*>

<ee:message>

<ee:set-payload resource=*"dwl/globals/globals-healthcheck-verbose-response.dwl"* />

</ee:message>

</ee:transform>

* Use inline dataweave scripts on the development phase, then pass it to an external file to have a clean xml and reusable scripts.

For more information: <https://docs.mulesoft.com/mule-runtime/4.3/dataweave>

## 3.2 HTTP(s) Listener

In MuleSoft CloudHub deployment target, the following default port forwarding rules are in place. The last two rules should be removed in Medtronic’s environments.

* noBalancer (http.private.port is a reserved word in CloudHub that resolves to 8091)
* ${https.private.port} when deploying to CloudHub using HTTPS and using a Dedicated Load Balancer (https.private.port is a reserved word in CloudHub that resolves to 8092)

For MuleSoft CloudHub networking guide, please refer to https://docs.mulesoft.com/runtime- manager/cloudhub-networking-guide

If HTTPS is required (e.g. in the DLB upstream protocol or for the CloudHub shared load balancer) use the HTTPS Listener with a TLS Context reference, configuring the key-store with a key-pair values generated with keytool.

*<tls:context name="TLS\_Context" doc:name="TLS Context">*

*<tls:key-store type="jks" path="company-keystore.jks" alias="${jks.alias}" keyPassword="${jks.key.password}" password="${jks.password}"/>*

*</tls:context>*

SSL can also be terminated at Load Balancer level when using a Dedicated Load Balancer in CloudHub or if there's an on-prem load balancer (e.g. F5) for hybrid scenarios. If this is the case, the HTTPS configuration can be avoided.

For more information: Mule 4: https://docs.mulesoft.com/connectors/http-connector

## 3.3 HTTP Request

### 3.3.1 Request to APIs or HTTPs endpoint with self-signed certificates

When consuming APIs that are exposed with a self-signed certificate, use the HTTP Requester with a TLS Context reference, configuring the trust store with the self-signed certificate or including the "insecure=true" attribute to avoid the certificate validation (useful when running locally, don't use this in prod).

### 3.3.2 HTTP Request Status Code Validator

By default, the HTTP Request expects a 2xx status code from the target system, if there's a different status code in the response it will throw an Exception. If you want to control different responses by adding some business logic, you have to configure the range of status codes considered as 'valid', by doing that you can put some logic after the HTTP Request, e.g. a Choice to make decisions based on the status code.

<http:request *method="GET" doc:name="Request" doc:id="80ece5fb-956a-4f8f-b374-4c929f32916c" config-ref="HTTP\_Request\_Configuration">*

*<http:response-validator >*

*<http:success-status-code-validator values="200..400" />*

*</http:response-validator>*

*</http:request>*

## 3.4 Web Service Consumer

Import the wsdls and the xsds in your project, use the Webservice Consumer module and point the configuration to the local wsdl.

For more information: <https://docs.mulesoft.com/connectors/web-service-consumer>

## 3.5 Database Connector

* Use parameterized queries for better performance.
* Use dynamic queries only if you don't have another choice, remember to validate the inputs previously (using dynamic queries could cause SQL-Injection vulnerability https://www.owasp.org/index.php/SQL\_Injection)

For more information: https://docs.mulesoft.com/connectors/db-connector-index

## 3.6 APIKit

* Use the APIKit features to generate the flows automatically Use the APIKit Exception Strategy to catch connector related exceptions
* It's a good practice to disable the APIKit Console under all exposed environments (e.g. CloudHub environments, onPrem production environments) as:
  + If the API Console is used without caution,you can affectreal objects/data due to the fact that you don't have a 'mocking-services’ option. When using API Portals (Exchange Public/ Private Portal) you can enable mocking services.
  + If an API Portal exists, the API Console is a redundancy.
  + If there are no policies applied, the API Console is a risk, due to point#1.

For more information: https://docs.mulesoft.com/anypoint-platform-for-apis/apikit- tutorial

## 3.7 Validator

For simple single-item check, use Validator.  
For more information: https://docs.mulesoft.com/connectors/validation-connector.

## 3.8 Flow Reusability

Use Flow references to separate and reuse common logic. The flow diagram should be clear, showing the steps of the use case.

For more information: <https://docs.mulesoft.com/mule-runtime/4.3/flowref-about#configure-flow-ref>

## 3.9 Java Custom Coding/Scripting

Do not to use Java or script custom coding unless it’s absolutely necessary.

## 3.10 Unit Testing

Create significative Unit tests using MUnit. Use MUnit, mock all the Connectors/Transports that make an outbound call.

* If there are query parameters, uri parameters or headers, use the set-message component to mock the incoming parameters
* Use Dataweave to simulate requests.

## 3.11 Mule Deploy Maven Plugin

Prepare your applications for automated deployments (executed by CI/CD processes), including the Mule Maven Deploy Plugin

● CloudHub deployments: <https://docs.mulesoft.com/mule-runtime/4.3/deploy-to-cloudhub>

## 3.12 Debugging & Trouble-shooting

Use Anypoint Studio Debugger to debug your Mule Applications.  
For more information: <https://docs.mulesoft.com/studio/7.5/visual-debugger-concept>

## 3.13 Wire Logging

To enable a more detailed logging to see all the HTTP requests and responses, configure the following loggers in src/main/resources/log4j2.xml

*<AsyncLogger name="org.mule.module.http.internal.HttpMessageLogger" level="DEBUG" /> <AsyncLogger name="com.ning.http" level="DEBUG" />*

# 4. Error Handling in API-led Connectivity – Best Practices

## 4.1 API-led Connectivity

API-led connectivity is a methodical way to connect data to applications through reusable and purposeful APIs. These APIs are developed to play a specific role – unlocking data from systems, composing data into processes, or delivering an experience.

The APIs used in an API-led approach fall into three categories:

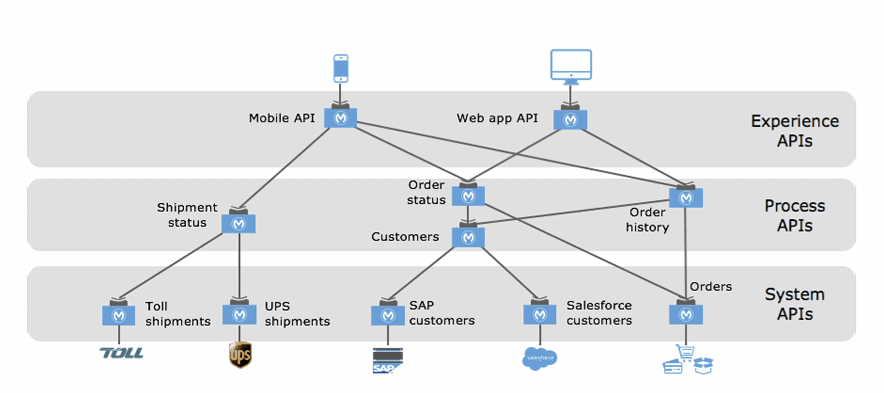
● System APIs: these usually access the core systems of record and provide a means of insulating the user from the complexity or any changes to the underlying systems. Once built, many users can access data without any need to learn the underlying systems and can reuse these APIs in multiple projects.

● Process APIs: These APIs interact with and shape data within a single system or across systems (breaking down data silos) and are created here without a dependence on the source systems from which that data originates, as well as the target channels through which that data is delivered.

● Experience APIs: Experience APIs are the means by which data can be reconfigured so that it is most easily consumed by its intended audience, all from a common data source, rather than setting up separate point-to-point integrations for each channel. An Experience API is usually created with API-first design principles where the API is designed for the specific user experience in mind.

A set of APIs following this approach will look like this:

Based on this, we should consider how we'll handle the possible errors that will be raised during the APIs execution. For example, a user consumes the Web App, calling the Experience API and creating an order. The Experience API calls the "Order Status" API (Process). This API consumes the "Customers" process API and retrieves a customer from the "Salesforce customers" API.



Now, if the customer sent by the user in the Web App has been deleted from Salesforce, the Salesforce connector will fail, and the "Salesforce customers" API will return an error. We have to analyze what should be returned to the Web App user, and how the error will be propagated. The Salesforce error message won't be designed for an end user and could contain sensitive information.

## 4.2 REST API Error Handling

REST APIs use the Status of an HTTP response message to inform clients of their request’s result.

### 4.2.1 HTTP Status

HTTP RFC defines over 40 standard status codes that can be used to convey the results of a client’s request. The status codes are divided into the five categories presented here:

* 1xx: Informational - Communicates transfer protocol-level information.
* 2xx: Success - Indicates that the client’s request was accepted successfully.
* 3xx: Redirection - Indicates that the client must take some additional action in order to complete their request.
* 4xx: Client Error - This category of error status codes points the finger at clients. It's commonly used for business validations and business errors.
* 5xx: Server Error - The server takes responsibility for these error status codes. These errors are not expected and should never happen.

The most common HTTP error codes are:

* 400 - Bad Request: An error in the client request (Mostly due to validations)
* 401 - Unauthorized: User can't be authenticated
* 403 - Forbidden: The server cannot give access to the resource
* 404 - Not Found: The resource defined in the URL doesn't exist
* 412 - Precondition Failed: One of the validations in the request failed (Sometimes used instead of 400)
* 500 - Internal Server Error: The server encountered an unexpected condition
* 502 - Bad Gateway: The server, while acting as a gateway or proxy, received an invalid response from an inbound server
* 504 - Gateway Timeout: The server tried to access an upstream service and it took more than the expected

### 4.2.2 Error Types

For the purposes of this guide, it's important to distinguish between two kinds of errors.

**Business Error**

It's an error expected by the business domain, for example, a validation not being satisfied, a required field not being sent, or a record not being found. The API should notify the user about this error in a proper way.

**System Error**

These kinds of errors aren't expected. Development support or operation teams have to be notified when one of these errors is raised. These exceptions could be thrown when a Database connection fails, an upstream service returns a server error, or due to a code error.

### 4.2.3 Error Response

A REST API should reply with an error code and a message to be shown to the consumer. To add more information, a description field could be returned. A transaction or request identifier could be useful to provide traceability, but we will see this later.

Here are some examples of how an error response should be:

{

"success": false,

"apiName": "medtronic-api-template",

"version": "1.0.0",

"correlationId": "690cf550-4789-11ea-abac-38f9d346c463",

"timestamp": "2020-05-18T17:03:29.325-05:00",

"errorDetails": [

{

"code": 400,

"message": "There was an issue with your request message",

"additionalInfo": "There was an issue with your request message : additional error info"

}

]

}

# 5. API Taxonomy

The API Taxonomy should follow REST Design patterns, summarized in the following table:

|  |  |
| --- | --- |
| **Topic** | **Description** |
| Behaviour | Create the resources with the right verb logic.  - GET: For obtaining data  - PUT: (Idempotent): To update data (the entire instance)  - POST (Not Idempotent): To store data  - PATCH: To update partial data of an instance  - DELETE: To delete an instance |
| CamelCase | Use lowerCamelCase for all fields, preferably don’t use underscores. |
| Date/Time representation | - Use standard date formats: ISO8601  - Use UTC |
| Media Types | - For the request: use ‘Content-Type’  - For the response: use ‘Content-Type’ |
| Pagination | When dealing with large responses, force pagination to avoid performance issues (long response times), define the offset and limit.  \*\*Based on backend capabilities |
| Query Parameters | - Use camelCase (example: dateFrom, dateTo)  - When dealing with collections, use comma separated values,  e.g. /accounts?fields=name,lastName |
| Resource types | **Collection resource**  /customers  **Instance resource**  /customers/{id} (example: /customers/189654 ) |
| Resources | - Use nouns, not verbs  - Coarse grained, not fine grained  - Architectural style for use-case scalability - Use lower case (example: /accounts)  - For resources with more than 2 words  - use lowercase for both words (example: /line-items) or  - use kebab-case (aka spinal-case) (example: /line-items) |

**Reference:** <https://restfulapi.net/>