



**Technical Paper**

**Working Draft**

**Example Implementation of the Dynamic  
Binding for the MEF LSO APIs**

**September 2021**

**This draft represents MEF work in progress and  
is subject to change.**

17 Disclaimer

18 © MEF Forum 2021. All Rights Reserved.

19 The information in this publication is freely available for reproduction and use by any recipient  
20 and is believed to be accurate as of its publication date. Such information is subject to change  
21 without notice and MEF Forum (MEF) is not responsible for any errors. MEF does not assume  
22 responsibility to update or correct any information in this publication. No representation or war-  
23 ranty, expressed or implied, is made by MEF concerning the completeness, accuracy, or applica-  
24 bility of any information contained herein and no liability of any kind shall be assumed by MEF  
25 as a result of reliance upon such information.

26 The information contained herein is intended to be used without modification by the recipient or  
27 user of this document. MEF is not responsible or liable for any modifications to this document  
28 made by any other party.

29 The receipt or any use of this document or its contents does not in any way create, by implication  
30 or otherwise:

- 31 a) any express or implied license or right to or under any patent, copyright, trademark or  
32 trade secret rights held or claimed by any MEF member which are or may be associated  
33 with the ideas, techniques, concepts or expressions contained herein; nor
- 34 b) any warranty or representation that any MEF members will announce any product(s)  
35 and/or service(s) related thereto, or if such announcements are made, that such an-  
36 nounced product(s) and/or service(s) embody any or all of the ideas, technologies, or  
37 concepts contained herein; nor
- 38 c) any form of relationship between any MEF member and the recipient or user of this  
39 document.

40 Implementation or use of specific MEF standards, specifications, or recommendations will be vol-  
41 untary, and no Member shall be obliged to implement them by virtue of participation in MEF  
42 Forum. MEF is a non-profit international organization to enable the development and worldwide  
43 adoption of agile, assured and orchestrated network services. MEF does not, expressly or other-  
44 wise, endorse or promote any specific products or services.

## Table of Contents

<b>1</b>	<b>Introduction.....</b>	<b>1</b>
<b>2</b>	<b>Architecture.....</b>	<b>2</b>
<b>3</b>	<b>Implementation .....</b>	<b>3</b>
<b>4</b>	<b>Installation &amp; Running.....</b>	<b>6</b>
<b>5</b>	<b>Use Cases.....</b>	<b>6</b>
5.1	Use Case 1: Lack of product specification .....	7
5.2	Use Case 2: Add the product specification.....	7
5.3	Use Case 3: New product specification support.....	7
5.4	Use Case 4: New product payload validation.....	7
<b>6</b>	<b>Program Details .....</b>	<b>7</b>
6.1	Product Schemas.....	7
6.2	Common Schemas .....	8
6.3	Schema Repository .....	8
6.4	Codegen .....	9
<b>7</b>	<b>Future Considerations .....</b>	<b>10</b>
<b>8</b>	<b>Support.....</b>	<b>10</b>

## List of Figures

Figure 1 – Cantata and Sonata API Framework .....	1
Figure 2 – Architecture .....	2
Figure 3 – Implementation.....	4
Figure 4 – Application flow.....	5
Figure 5 – Architecture with Catalog API .....	10

## 1 Abstract

This non-normative document provides an example implementation of the so called dynamic binding approach to combining the LSO APIs with LSO Payloads. It's purpose is to help Software Architects, Analysts and Developers to better understand and familiarize with the dynamic binding concept and what are the advantages and disadvantages it brings.

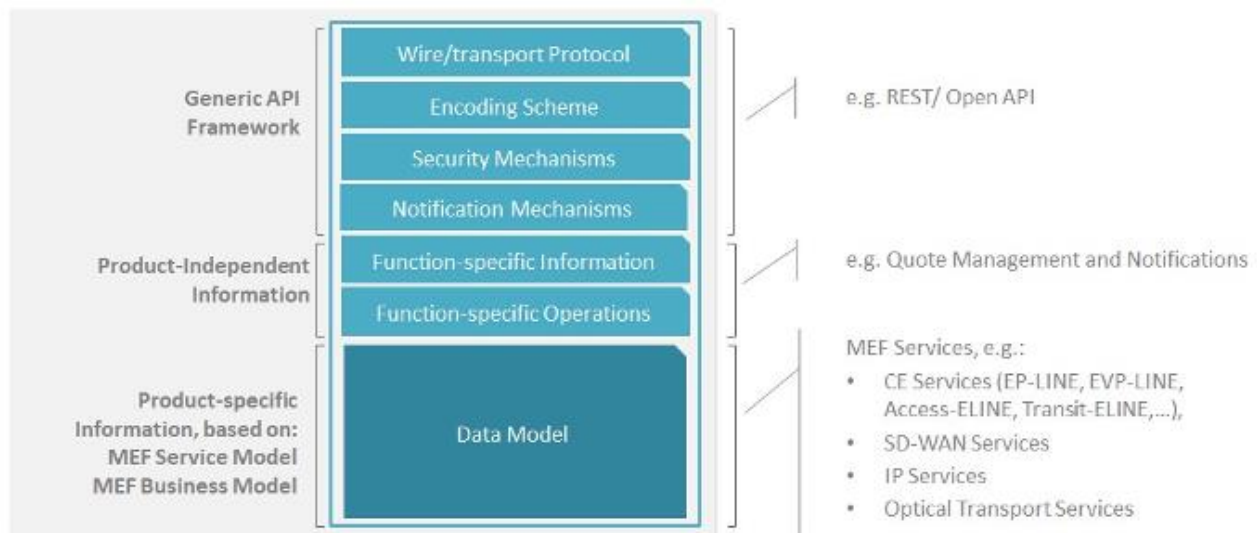
## 2 Introduction

This document's resources: source code, Postman collection, and this document, can be found on MEF GitHub (available for registered MEF members. Please visit [How do I get access to MEF GitHub?](https://github.com/MEF-GIT/Example-LSO-Dynamic-Binding-Implementation) to get the access):

<https://github.com/MEF-GIT/Example-LSO-Dynamic-Binding-Implementation>

As presented in Figure 1. MEF LSO APIs are composed of three structural components:

- Generic API framework
- Product-independent functional API – called the envelope. Function-specific information and Function-specific operations, e.g., the Product Offering Qualification (POQ), Quote, Product Order)
- Product-specific information – MEF product specification data model, e.g., Access E-Line



**Figure 1 – Cantata and Sonata API Framework**

They need to be used together. There are two ways to accomplish that.

*Dynamic binding* is an approach in which supported product definitions can be on-boarded or removed at runtime without the need to redeploy software components that support the validation process. It is opposed to *static binding* approach in which selected product-specific definitions are integrated within the functional API definition before the implementation starts.

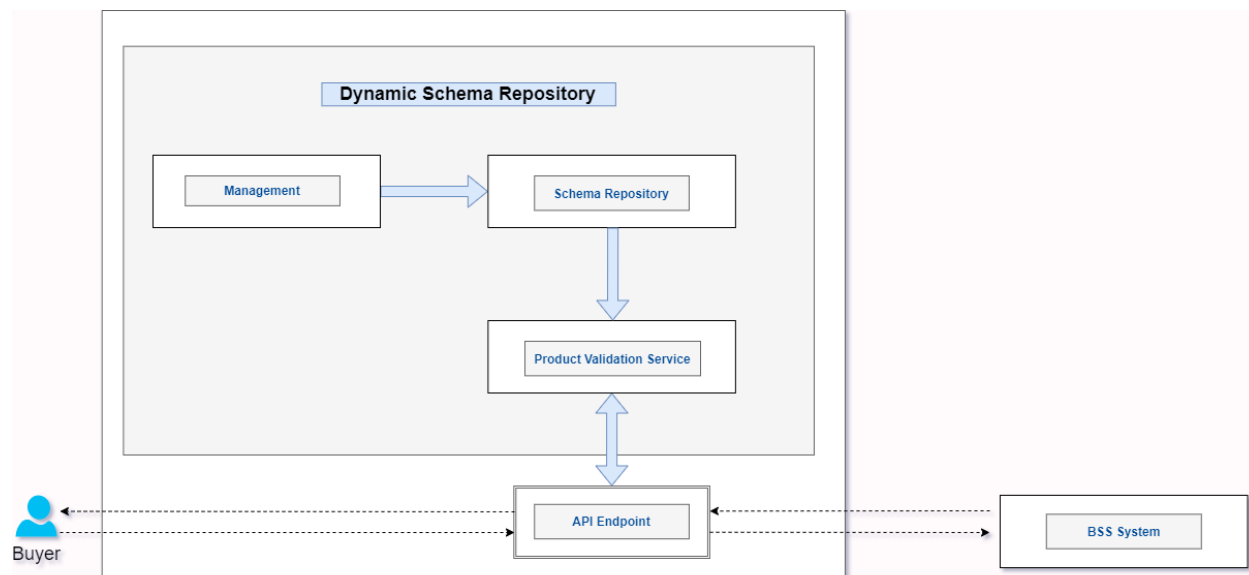
Understanding this document requires a good knowledge of the envelope, product payload, static and dynamic binding concepts and patterns. To acquaint with the, please refer to the [MEF W87 LSO Cantata and LSO Sonata Product Offering Qualification API – Developer Guide](#), in particular with chapters:

- 4.4 Approach
- 5.2.3 Integration of Product Specifications into Product Offering Qualification Management API

This example implements the SDK [Billie Release](#) of Product Offering Qualification API but instead of supporting all requirements defined in [MEF W87 Developer Guide](#), it focuses only on demonstrating how dynamic binding can be used to implement validation of incoming data against additional modules added at run-time, it doesn't help actually implement the module-specific business behavior.

### 3 Architecture

Conceptual architecture is presented in Figure 2 (simplified).



**Figure 2 – Architecture**

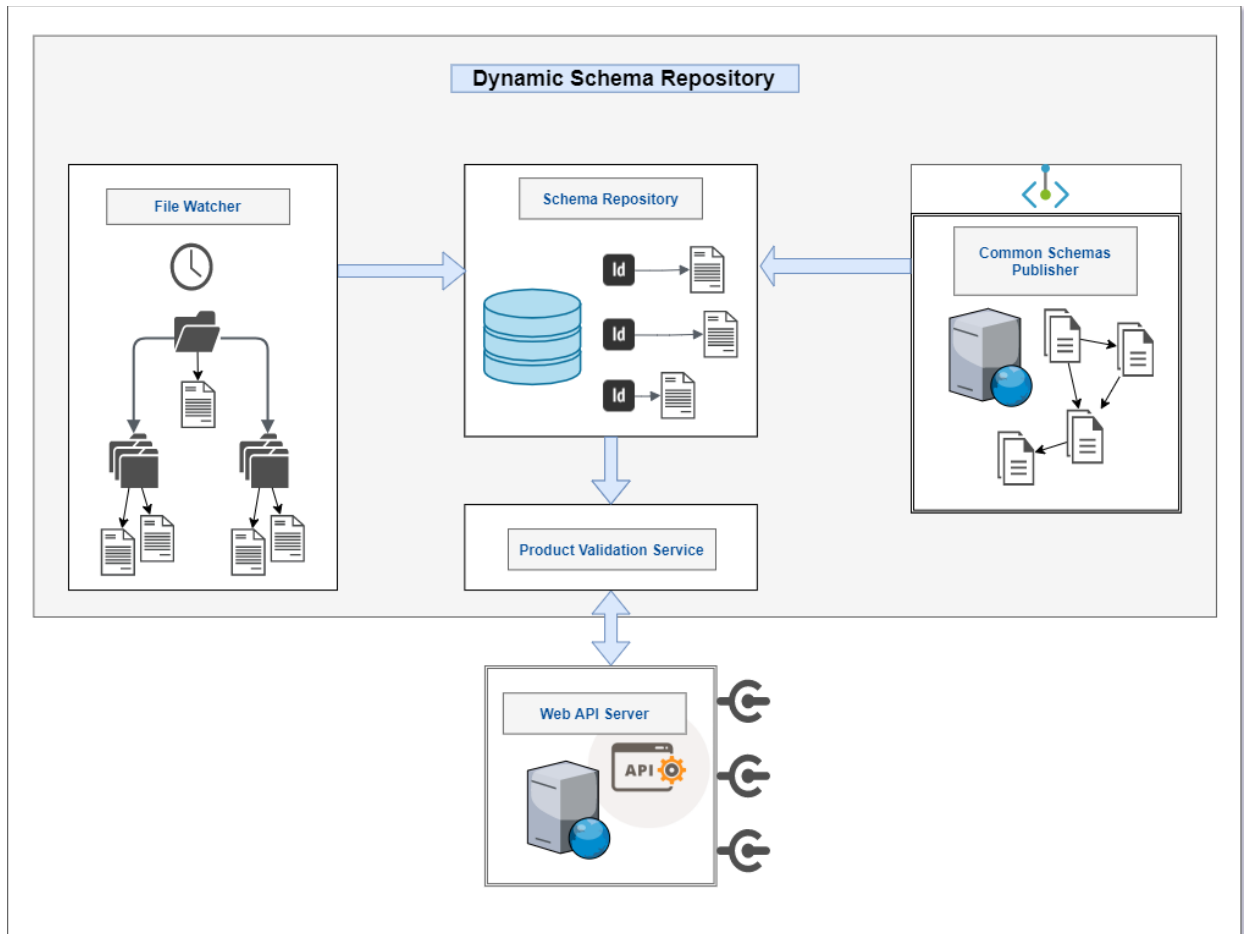
*API Endpoint* implements POQ endpoints defined in [MEF W87](#). POQ Endpoint is using *Product Validation Service* which is responsible for validating the payload against appropriate product specification. The product specifications are served from *Schema Repository*. There is a *Management* component that gives control over which product specifications are available in the platform.

The *BSS System* handles the actual business processing of the request.

## 4 Implementation

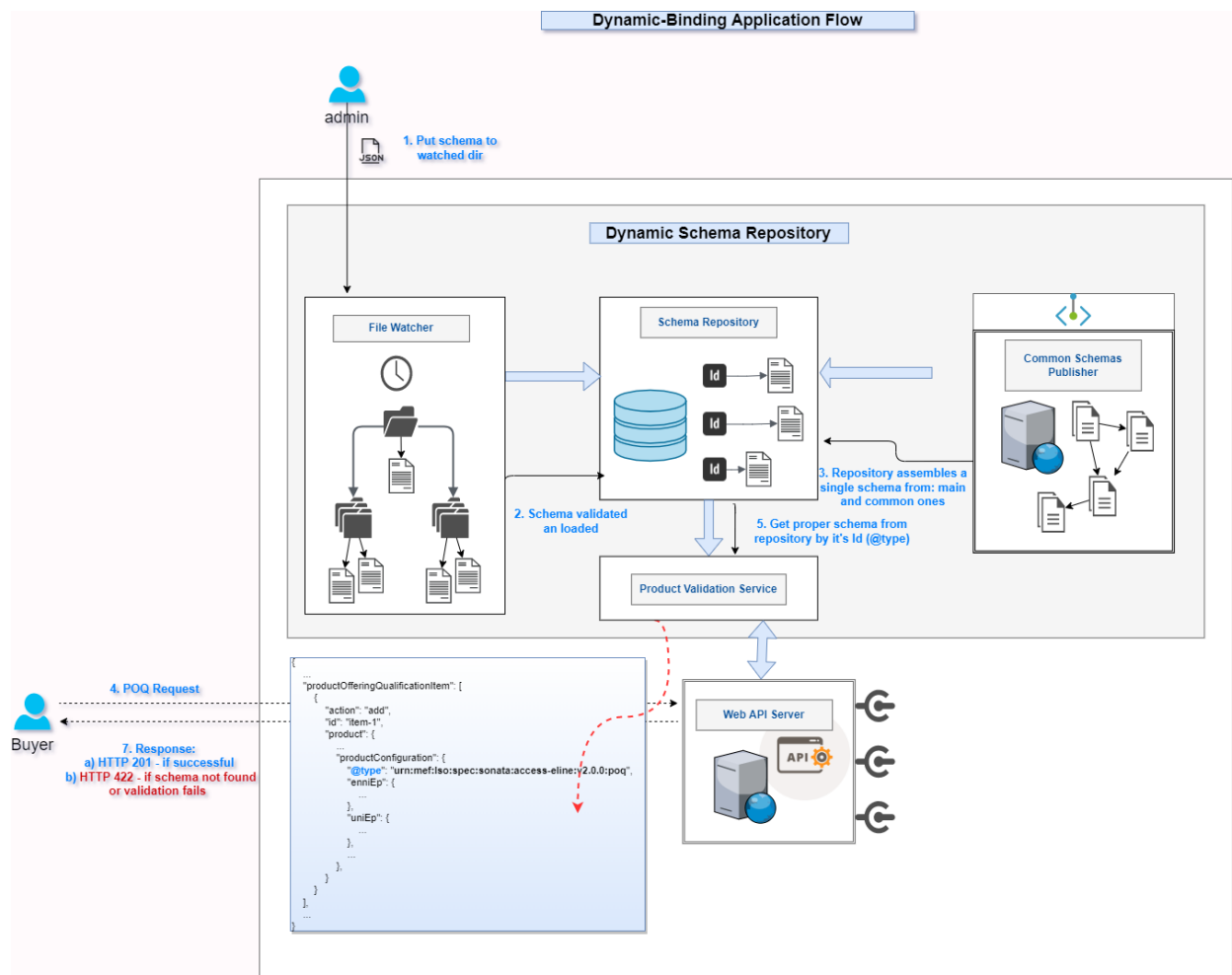
The example implements the abovementioned architecture in the following way:

- *Management* is implemented as a *File Watcher* – a specific file repository for product yaml schemas. It is monitored for changes and new schemas are automatically parsed and loaded into the *Schema Repository*
- *Schema Repository* – the central point. A place that acts as a logical store of all available product specifications.
- *Common Schemas Publisher* – implementation-specific internal part of the repository that stores the Common Schemas – the data model and dictionaries shared between the schemas.
- *Product Validation Service* – a logical component used by the request processing logic to validate incoming payload with supported schemas.
- *WEB API Server* that serves the MEF Product Offering Qualification API and uses the *Dynamic Schema Repository* for the Product specific payload validation.
- *BSS System* – is not present and is replaced with a simple hard-coded response logic.



**Figure 3 – Implementation**

The functionalities supported by this example implementation are described in Figure 4:



**Figure 4 – Application flow**

1. New schema is put into the file system into the watch dir. This is done by some Seller admin user.
2. The schema is noticed by the file watcher, interpreted, validated, and loaded into the *Schema Repository*.
3. The *Schema Repository* uses the shared schemas available in *Common Schemas Publisher* and builds a single resolved schema for the product. After this step, the Product Specification is available to use.
4. A Buyer sends a POQ Request with the new product specification.
5. The Server uses the `product.productConfiguration.@type` to identify the proper schema in the *Schema Repository*.
6. The Server validates the request according to the schema found.
7. The response is provided with code 201 if successful, or an error response if the schema is not found or the payload is not validated properly.

The *WEB API Server* uses the `product.productConfiguration.@type` property to identify the schema needed to validate the product payload. The MEF standard product

specifications are identified by the @type in specific URN format:

urn:mef:lso:spec:sonata:access-eline:v2.0.0:poq, it contains:

- the name of the interface it applies: urn:mef:lso:spec:sonata
- the name of the product: access-eline
- version of the specification: 2.0.0
- function to which it applies (products may have different required attributes per different contexts like POQ or Order): poq

The schema is checked with every request. This means that the product specification can be dynamically and freely added or multiple versions can be supported.

## 5 Installation & Running

- Java 11 required

To configure and build the program, please do the following:

1. Create/choose a root monitoring directory and put its path to: `src/main/resources/application.yml` (`watch.dir` key). The pre-configured is an existing empty directory: `./productSchema`.
2. Run from CLI: `mvn clean install` – to build the application and produce the result JAR file (in `/target` directory). It should generate single (fat) JAR, containing all compiled classes/resources + all dependencies, for example: *dynamic-binding-example-0.0.1-SNAPSHOT.jar*
3. Run from CLI: `java -jar target/dynamic-binding-example-0.0.1-SNAPSHOT.jar` – it will start the application  
NOTE: alternatively, the program may be run from IDE `mvn spring-boot:run`
4. Open a *Postman* collection (`/extras/ProductOfferingQualificationManagement.postman_collection.json`) and send a POST request to check if the server is up and running.

NOTE: One can also provide their own: product + common schemas

## 6 Use Cases

Use Cases listed below should be executed in presented order as they depend on each other.

First run the application as described above, without any modifications (empty `./productSchema` and `extras/main_schemas/` unchanged, or adapt to your changes if any). This will start a server without any product specification loaded.

## 6.1 Use Case 1: Lack of product specification

- Open the provided Postman collection. There are two POST requests prepared.
- Run the *Valid Request*.
- Since there is no Product Specification loaded, the server response should be Error422 with the message: No schema found for id=...

## 6.2 Use Case 2: Add the product specification

- The example request uses the *Access Eline* product with "\$id":  
`urn:mef:lso:spec:sonata:access-eline:v2.0.0:poq`.
- Copy the appropriate product specification `accessElineOvc.yaml` from `extras/main_schemas/accessEline/poq/` to the `watch.dir (/productSchema/)`.
- Product specification is automatically loaded into the *Dynamic Schema Repository*.
- Verify the fact by checking the application log output (Saved schema:  
`[urn:mef:lso:spec:sonata:access-eline:v2.0.0:poq -> productSchema/accessElineOvc.yaml]`)

## 6.3 Use Case 3: New product specification support

- Now that the new product specification is supported, run again the *Valid Request* from the Postman collection.
- This time the response should be 201 Created and a full server response provided.

## 6.4 Use Case 4: New product payload validation

- The second POST request available, the *Invalid request*, contains an intentional error in line 28: `colorMode` has value `WRONG_VALUE` which does not match the field's enumeration `[COLOR_BLIND, COLOR_AWARE]`.
- The automatically generated code rejects this request based on the API specification and throws an Error422 with appropriate message  
`("$.enniEp.ingressBandwidthProfilePerClassOfServiceName[0].bwpFlow.colorMode: does not have a value in the enumeration [COLOR_BLIND, COLOR_AWARE]")`

# 7 Program Details

## 7.1 Product Schemas

MEF delivers product schemas in a way that the common parts shared among different products are extracted to separate files. All actual product schemas are those that contain the "\$id" parameter available. All product schemas available in the Billie release are placed in the `extras/main_schemas/`. This folder is not loaded automatically. From here the user can

copy and paste given schema into the `watch_dir` so that it can be dynamically loaded into the *Schema Repository*. Note that the product specification comes in different flavors depending on the context API (poq, quote, order, inventory) and it is reflected in the `$id` – this implementation does not validate if the `@type` provided in the request is dedicated to POQ (this is not the point of the example).

### Schema files watcher/monitor

- When the program starts, it immediately starts to monitor the directory provided by the user (`watch_dir`).
- The monitored directory can be populated by the user at any time – also before starting the application.
- The monitor directory should contain product schemas (those containing `$id` property).
- The program is not only capable to discover the creation/update/deletion of single YAML schemas, but also discovers the whole directory structure (with schemas). In that case, also the sub-directories are watched for changes.
- Generally, when the program discovers any filesystem change on the 'root directory' or its descendants it:
  - adds other sub-directories for further monitoring
  - creates or updates one or many YAML schemas - in the repository
  - removes one or many YAML schemas from the repository
- The system accepts JSON schemas in YAML format

## 7.2 Common Schemas

Schemas that do not contain the "`$id`" parameter and are referred to as the common schemas. All common schemas available in the Billie release are placed in `src/main/resources/schemas/` and they are automatically loaded to the *Common Schemas Publisher* at the application start.

- The product schema may consist of links (`$ref`'s) to the other, common schemas – and these may also have links to other ones, etc. Product and common schemas are composed together to form a single, fat schema – which finally – goes to the repository.
- The application utilizes 3rd party software for schema validation: `json-schema-validator` which doesn't support local/filesystem `$ref` links – only URLs are supported. That's the reason the common schemas are published by the webserver. This also required modification of the MEF original schemas so that the `$ref` uses URLs instead of local references.
- To properly serve all common schemas – a Schema REST controller is introduced which publishes the schemas as static content (under *Tomcat* webserver).

## 7.3 Schema Repository

- Holds YAML schemas for further use. Each schema repository item contains:

- unique schema ID
  - monitored file path
  - fully composed JSON schema content as `JsonSchema` object
- Supports basic CRUD operations on the repository
- Uses builtin *Spring* key-value cache

## 7.4 Codegen

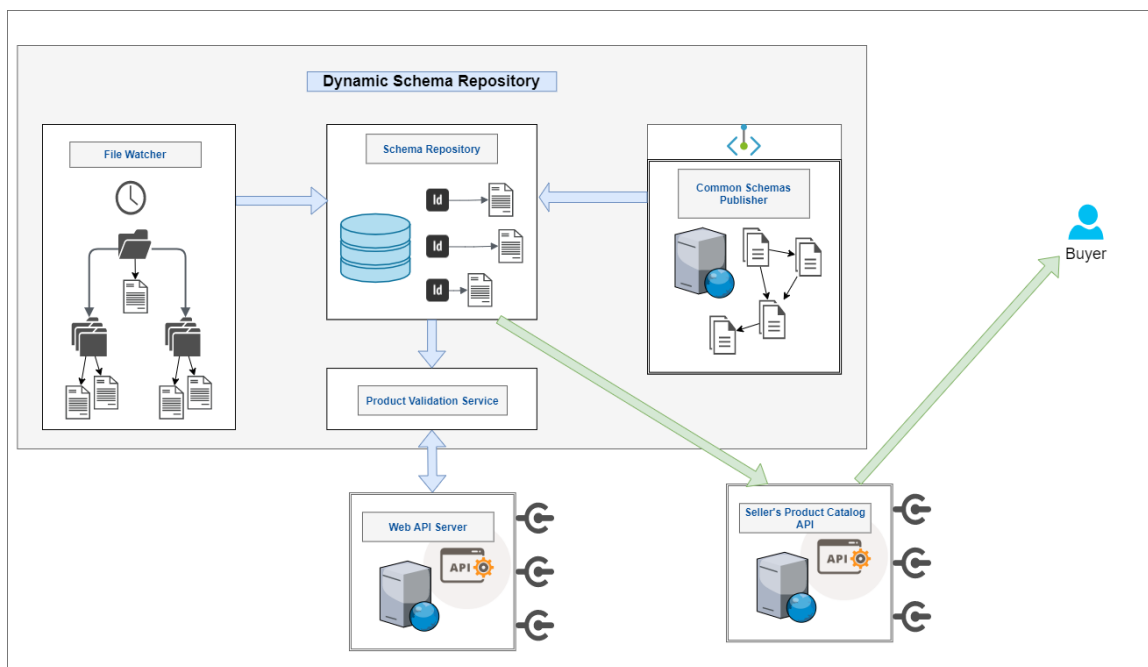
- *CodeGen* is used to generate the code and API from the 'Swagger/OpenAPI' document. This example utilizes the Sonata POQ:  
(`src/main/resources/productOfferingQualificationManagement.api.yaml`)
- This file is pointed in `pom.xml` in the property of `openapi-generator-maven-plugin`
- The *CodeGen* is instructed to generate only Web API. The REST controllers (implementing the API) are provided by this program as custom ones.
- The *CodeGen* is also configured to not produce very important model object: 'MEFProductConfiguration'. Also here, the custom version is provided by the application.
- Custom 'MEFProductConfiguration' object should consist of:
  - `@type` property – containing the ID of the validation schema
  - any JSON content/payload – which is to be verified by the validation schema

## 8 Future Considerations

At the time of this example's implementation, the MEF Catalog API was not available. Thus there was no standardized process of Product Specifications' exchange. Product specifications are provided by MEF as part of the standard documents and also in the releases of the SDKs as YAML coded JSON documents. The Seller can exchange the list and details of offered products during the onboarding process. The example assumes that a standard MEF product is used and that the Buyer already has the specification.

Once the Catalog API is available, the Seller can expose the specification and offering via an API from where the Buyer can easily pick it.

The example diagram of the target architecture is presented in Figure 5:



**Figure 5 – Architecture with Catalog API**

## 9 Support

This example implementation is provided as open-source and MEF members are free to further improve it or provide fixes.

If you have any questions, please contact:

Michał Łączyński, Amartus

[michal.laczynski@amartus.com](mailto:michal.laczynski@amartus.com)