

## EFPF: European Connected Factory Platform for Agile Manufacturing



European Factory  
Platform

### WP3: EFPF Architecture

## D3.12: EFPF Data Spine Realisation – Final Report Vs: 1.0

**Deliverable Lead and Editor:** Rohit Deshmukh, Fraunhofer FIT

**Contributing Partners:** FIT, LINKS, CNET, SRFG, ICE, NXW, C2K, SRDC, VLC, UOS-ITI, ALM, CERTH, ASC, ELN, SIE, CMS, FOR

**Date:** 2022-06-30

**Dissemination:** Public

**Status:** <Draft | Consortium Approved | EU Approved>

#### Short Abstract

This deliverable presents an update to the architecture of the EFPF ecosystem since D3.11: EFPF Data Spine Realisation - I. The deliverable describes how the Data Spine, together with other Ecosystem Enablers, enables the creation of and communication in the EFPF ecosystem. It further details the methodologies for the integration of tools, services, and platforms with the ecosystem.

Grant Agreement:  
825075



## Document Status

<b>Deliverable Lead</b>	Rohit Deshmukh, Fraunhofer FIT
<b>Internal Reviewer 1</b>	Usman Wajid, ICE
<b>Internal Reviewer 2</b>	Raluca Maria Repanovici, SIE
<b>Type</b>	Deliverable
<b>Work Package</b>	WP3: EFPP Architecture
<b>ID</b>	D3.12: EFPP Data Spine Realisation – Final Report
<b>Due Date</b>	2022-06-30
<b>Delivery Date</b>	2022-06-30
<b>Status</b>	<Draft   Consortium Approved   EU Approved>

## History

See Annex B.

## Status

This deliverable is subject to final acceptance by the European Commission.

## Further Information

[www.efpf.org](http://www.efpf.org)

## Disclaimer

The views represented in this document only reflect the views of the authors and not the views of the European Union. The European Union is not liable for any use that may be made of the information contained in this document.

Furthermore, the information is provided “as is” and no guarantee or warranty is given that the information is fit for any particular purpose. The user of the information uses it at its sole risk and liability.

## Project Partners:



## Executive Summary

This deliverable presents the final state of the EFPF ecosystem architecture with focus on the categorization of the EFPF components into two major types for an easy and efficient administration; the design and realisation of the Data Spine, the integration and interoperability and communications layer in the EFPF ecosystem; the interfaces for the connected tools, systems, and platforms that enable their usage for the realisation of the use cases and finally, the data model interoperability layer that analyses the alignment of the data models and dataflows in the pilots and Open Call experimentation scenarios.

This consolidated deliverable reports on the work done in the following tasks in the EFPF project, solely because all of these tasks perform highly interrelated activities that contribute towards the establishment of the EFPF federation through the realisation of an open interoperability mechanism i.e., the Data Spine.

T3.1: EFPF Architecture

T3.2: Design and Realisation of Interoperable Data Spine

T3.4: Interfaces for Tools, Systems and Platforms

T3.5: Data Model Interoperability Layer

This deliverable illustrates the systematic approach taken for the refinement of the architecture, i.e., the definition of the vision based on the motivation, the identification of the challenges and definition of requirements based on the challenges, the identification of the core components called ‘Ecosystem Enablers’ that fulfil these requirements to establish the EFPF ecosystem. The deliverable describes the updated interfaces for the connected tools, services, systems, and platforms, that are the building blocks for the realisation of use case scenarios in the EFPF ecosystem and the data model interoperability layer that aligns the data models of these tools and specifies how data transformation can be performed to facilitate the interplay and interconnectivity between the distributed technologies.

One important objective of this deliverable is to provide necessary information to the EFPF project participants as well as the 3<sup>rd</sup> parties who might be interested in interlinking their tools, services, or platforms by using the Ecosystem Enablers and making them part of the EFPF ecosystem. To enable this, the deliverable highlights the integration and interaction methodologies for various stakeholders such as service providers and service consumers, etc.

The architecture of the EFPF ecosystem and the Data Spine has been designed with modularity, scalability, and extensibility in mind to meet the need for incorporating new tools, services, and platforms in the EFPF ecosystem, with minimum effort, and also for adding interoperability support for new aspects, such as communication protocols.

This deliverable also summarises the use of the EFPF Ecosystem Enablers and the Smart Factory Tools and Services from the connected platforms for realising the real-world use cases from the pilots as well as the Open Call experimentation scenarios.

## Table of Contents

0	Introduction .....	1
0.1	EFPF Project Overview .....	1
0.2	Deliverable Purpose and Scope .....	1
0.3	Target Audience .....	2
0.4	Deliverable Context .....	2
0.5	Document Structure.....	3
0.6	Document Status .....	3
0.7	Document Dependencies .....	3
0.8	Glossary and Abbreviations.....	3
0.9	External Annexes and Supporting Documents .....	4
0.10	Reading Notes.....	4
1	EFPF Architecture Update .....	5
1.1	Motivation and Requirements.....	5
1.1.1	Motivation .....	5
1.1.2	Offerings and Stakeholders .....	7
1.1.3	Challenges, Requirements, and Core Components .....	10
1.2	EFPF Architecture Context View .....	13
1.3	EFPF Architecture Functional View .....	15
1.3.1	Functional Overview of the Ecosystem Enablers.....	16
1.3.2	Functional Overview of the Smart Factory Tools & Services in the EFPF Ecosystem .....	25
1.4	EFPF Architecture Information View.....	36
1.4.1	High-level Dataflow Patterns in the EFPF Ecosystem .....	36
1.4.2	Federated Search Indexing Dataflows.....	38
1.4.3	Aerospace Pilot: Workplace Environment Monitoring.....	39
1.4.4	Furniture Pilot: Analytics & Predictive Maintenance.....	41
1.4.5	Open Call Experiment: DNET Labs .....	42
1.5	EFPF Architecture Development and Deployment View .....	43
1.6	Integration Methodologies & Documentation Structure .....	45
1.6.1	Integration Methodology for Platform Providers.....	47
1.6.2	Integration Methodology for Tool/Service Providers .....	49
1.6.3	Integration Methodology for Composite Application Developers .....	51
2	Design and Realisation of Interoperable Data Spine .....	55
2.1	Vision and Objectives .....	55
2.2	Requirements .....	56
2.3	Interoperability Approach.....	60
2.4	Design of Interoperable Data Spine.....	60
2.4.1	Components of the Data Spine.....	61
2.4.2	Data Spine Architecture and Components' Interaction .....	66
2.5	Realisation of Interoperable Data Spine .....	68
2.5.1	EFPF Security Portal .....	68
2.5.2	Integration Flow Engine .....	73
2.5.3	API Security Gateway.....	83
2.5.4	Service Registry .....	85
2.5.5	Message Bus.....	95
2.5.6	Summary .....	102
2.6	Deployment .....	103
2.7	Testing Scenarios & Framework.....	105

2.7.1	Integration Testing Scenarios .....	105
2.7.2	Integration Testing Framework .....	105
2.7.3	Performance Testing Scenarios .....	107
2.7.4	Performance Testing Framework .....	107
2.8	Platform/Service Integration & Dataflow through Data Spine .....	110
2.9	Data Spine Usage in Pilots and Open Call Experiments .....	116
2.10	Evaluation.....	117
2.10.1	Quantitative Evaluation .....	117
2.10.2	Data Spine Usage Survey.....	119
3	Interfaces for Tools, Systems and Platforms .....	125
3.1	Introduction.....	125
3.2	Interfaces for Tools, Systems and Platforms .....	125
3.2.1	Interfaces for Tools & Services in the EFPF Ecosystem.....	125
3.2.2	Interfaces for Platforms in the EFPF Ecosystem .....	250
3.3	API Management.....	261
3.4	Interface Contracts and Their Management .....	264
3.4.1	Introduction.....	264
3.4.2	Interface Contract Management Tool .....	265
4	Data Model Interoperability Layer .....	267
4.1	Introduction.....	267
4.2	Final Pilot scenarios analysis.....	267
4.2.1	Working Environment Monitoring.....	267
4.2.2	Production Optimisation Pilot.....	268
4.2.3	Bins' Fill Level Monitoring .....	272
4.2.4	Matchmaking .....	274
4.2.5	Projects which did not implement interoperability workflows .....	275
4.2.6	Conclusion from Pilot scenarios analysis.....	276
4.3	Open-Call Project Analysis .....	277
4.3.1	Open Call LORTEK .....	277
4.3.2	Open Call DNET .....	277
4.4	Data Model Interoperability Surveys.....	278
4.4.1	Pilot Survey Analysis .....	278
4.4.2	Open Call Survey Analysis .....	279
4.4.3	Data Model Interoperability Tools Adoption .....	280
5	Summary of Architectural Considerations & Implications.....	282
6	Conclusion and Outlook.....	285
	Annex A: History .....	287
	Annex B: References .....	289
	Annex C: Data Spine Testing Scenarios.....	293

## 0 Introduction

### 0.1 EFPF Project Overview

EFPF – European Connected Factory Platform for Agile Manufacturing – is a project funded by the H2020 Framework Programme of the European Commission under Grant Agreement 825075 and conducted from January 2019 until December 2022. It engages 30 partners (Users, Technology Providers, Consultants and Research Institutes) from 11 countries with a total budget of circa 16M€. Further information: [www.efpf.org](http://www.efpf.org)

In order to foster the growth of a pan-European platform ecosystem that enables the transition from “analogue-first” mass production, to “digital twins” and lot-size-one manufacturing, the EFPF project will design, build and operate a federated digital manufacturing platform. The Platform will be bootstrapped by interlinking the four base platforms from FoF-11-2016 cluster funded by the European Commission, early on. This will set the foundation for the development of EFPF Data Spine and the associated toolsets to fully connect the existing platforms, toolsets and user communities of the 4 base platforms. The federated EFPF platform will also be offered to new users through a unified Portal with value-added features such as single sign-on (SSO), user access management functionalities to hide the complexity of dealing with different platform and solution providers.

### 0.2 Deliverable Purpose and Scope

The purpose of this document, “D3.12: EFPF Data Spine Realisation - Final Report”, is to present four different aspects of the EFPF ecosystem: the architecture, the design and realisation of the Interoperable Data Spine, the interfaces for tools, systems and platforms in the ecosystem and the data model interoperability layer in the form of four dedicated sections. First, an overview of the architecture of the EFPF ecosystem with focus on updates from the previous version of the architecture that was presented in the architecture deliverable D3.11 (M18) is presented. This includes the identification of the core components called ‘Ecosystem Enablers’ that enable the creation and functioning of the ecosystem. Second, the detailed design of the Data Spine, its conceptual components and their relationships and interactions with each other, the overview, architecture, interfaces, configuration, and operation of the technologies selected to realise these conceptual components of the Data Spine, integration, and interaction methodologies for the stakeholders such as the service providers and service consumers, etc., are elaborated. In addition, the work done on setting up the automated deployment process and the testing framework to ensure that the Data Spine DevOps process is as comprehensive as possible, is highlighted. Furthermore, Data Spine’s usage in realising the pilot and Open Call experimentation scenarios is briefly mentioned. Third, the interfaces for tools, services, systems, and platforms which are the building blocks of the EFPF ecosystem, the management of their APIs and APIs contracts between them are described. Finally, the data model interoperability layer that aligns the data models of the federated platforms to support meaningful message exchange and viable business processes that spread across two or more of the existing EFPF platforms is explained.

The scope of this deliverable includes the updates to architectures of the tools, services, systems, and platforms in the EFPF ecosystem and their APIs and not their detailed description and information related to their configuration, deployments, and operation, etc.

In addition, this deliverable also addresses all aspects related to the design and realisation of the Data Spine and the data model interoperability layer.

### 0.3 Target Audience

This document aims primarily at project participants and external entities that are interested in interlinking their tools, services, and/or platforms using the Data Spine and the other Ecosystem Enablers and making them part of the EFPF ecosystem. In addition, this deliverable provides the European Commission (including appointed independent experts) with an overview of the underlying architecture of the EFPF ecosystem.

### 0.4 Deliverable Context

This document is one of the cornerstones for achieving the project results. Its relationship to other documents is as follows:

- **D2.1: Project Vision and Roadmap for Realising Integrated EFPF Platform:** Provides an overview of the EFPF project and platform
- **D2.2: Platform Interoperation Challenge Report:** Describes a small set of business challenges that involve different usage scenarios across the participating platforms to kickstart EFPF
- **D2.3: Requirements of Embedded Pilot Scenarios:** Provide an overview of the pilot requirements on the federated EFPF platform
- **D2.4: EFPF Platform Requirements:** Defines the requirements to develop, enhance and deliver the EFPF ecosystem that is maintained as a living JIRA document with final submission at M42.
- **D3.1: EFPF Architecture-I:** Presents the baseline architecture of the EFPF ecosystem with focus on the EFPF platform and the Data Spine
- **D3.11: EFPF Data Spine Realisation - I:** presents an update to the architecture of EFPF, the design and realisation of the Interoperable Data Spine, interfaces for tools, systems, and platforms in the ecosystem and the data model interoperability layer at M18.
- **D4.13: Smart Factory Solutions in the EFPF Ecosystem - I:** Provides a report of the Tools and Services available within the EFPF Ecosystem that can be used to provide Smart Factory solutions
- **D5.13: EFPF Interfacing, Evolution and Extension:** Provides requirements for the EFPF Ecosystem, its evolution and extension and includes detailed information about the EFPF Marketplace and Portal
- **D6.1: EFPF Integration and Deployment - I:** Presents the development and deployment architecture of the EFPF ecosystem with focus on the EFPF platform and the Data Spine at M12.
- **D6.2: EFPF Integration and Deployment- Final Report:** Presents the development and deployment architecture of the EFPF ecosystem at M42.
- **D9.1: Implementation and Validation through Pilot-1:** Aerospace Pilot
- **D9.2: Implementation and Validation through Pilot-2:** Furniture Pilot



- **D9.3: Implementation and Validation through Pilot-3: Circular Economy Pilot**

## 0.5 Document Structure

This deliverable is broken down into the following sections:

- **Section 1: EFPF Architecture Update:** Presents an overview of the architecture of the EFPF ecosystem with focus on updates from the previous version of the architecture that was presented in the architecture deliverable D3.11
- **Section 2: Design and Realisation of Interoperable Data Spine:** Presents the design and realisation of Data Spine – the interoperability backbone of EFPF
- **Section 3: Interfaces for Tools, Systems and Platforms:** Describes the interfaces for tools, systems, and platforms – the building blocks of the EFPF ecosystem
- **Section 4: Data Model Interoperability Layer:** Describes the data model interoperability layer that aligns the data models of the federated platforms
- **Section 5: Summary of Architectural Considerations & Implications:** Presents a summary of architectural considerations and implications based on the requirements and from the perspectives of the users
- **Section 6: Conclusion and Outlook:** Concludes the deliverable and mentions the future work.
- **Annexes:**
  - **Annex A:** History
  - **Annex B:** References
  - **Annex C:** Data Spine Testing Scenarios

## 0.6 Document Status

This document is listed in the Description of Action (DoA) as “public”.

## 0.7 Document Dependencies

This document is the final of the three deliverables that describe the architecture of the EFPF ecosystem. The first deliverable, D3.1 submitted at Month 9 of the EFPF project described the baseline architecture of the EFPF ecosystem. The second deliverable, D3.11 submitted at Month 18 described an update to the EFPF ecosystem architecture, the first report on the design and realisation of the Data Spine, the interfaces for the federated tools, services, and platforms, and the data model interoperability layer. This, the third and the final architecture deliverable at Month 42 provides the final architecture and the final report on the above-mentioned tasks and aspects.

## 0.8 Glossary and Abbreviations

A definition of common terms related to EFPF, as well as a list of abbreviations, is available in the supplementary and separate document “EFPF Glossary and Abbreviations”.

Some terms that are extensively used in this document are defined below:

- **Digital Platform:** A platform that provides offerings such as digital tools, services and data and secures access to them using its own Identity and Access Management service.

- Ecosystem of Digital Platforms: A federation of digital platforms that enables an easy creation of cross-platform applications.
- Ecosystem Enablers: The core central components that enable the creation and functioning of the ecosystem.
- Data Spine: A federated interoperability enabler that bridges the interoperability gaps between the services of heterogeneous platforms and enables an easy and intuitive creation of cross-platform applications.

Further information can be found at [www.efpf.org](http://www.efpf.org)

## **0.9 External Annexes and Supporting Documents**

Annexes and Supporting Documents:

- None

## **0.10 Reading Notes**

- None

# 1 EFPF Architecture Update

This section provides an update to the architecture of the EFPF ecosystem from the earlier versions presented in the previous deliverables D3.1 (M9) and D3.11 (M18). The methodology used to define the architecture that is based on IEEE 1471 "Recommended Practice for Architectural Description for Software-Intensive Systems" [Hil00], ISO/IEC/IEEE 42010:2011 "Systems and software engineering - Architecture description" [IEEE11], and the "Architectural Perspectives" chapter by Rozanski and Woods [RW05], is described in D3.1. In order to preserve the context and ensure readability, some information from D3.1 and D3.11 is also included in this section.

## 1.1 Motivation and Requirements

This section, at first, presents the motivation behind the creation of an ecosystem of digital manufacturing platforms. Second, it identifies the challenges that need to be addressed in order to form the ecosystem and enable interoperability and communication among its components. Third, it identifies the requirements for the central core components and finally, it maps the pilot requirements to the Smart Factory Tools and Services in the ecosystem that are used to fulfil them.

### 1.1.1 Motivation

Manufacturing is one of the key driving forces of the European economy. It creates over 30 million jobs (about 20% of all jobs in Europe) and generates a turnover of €7000 billion in 25 industrial sectors and over 2 million companies, dominated by SMEs [Sim17]. In the digital age, it is crucial for the manufacturing companies to adopt advanced manufacturing processes and technologies to remain sustainable and at the forefront of innovation.

With the advent and pervasiveness of the Internet of Things (IoT), big data and cloud computing technologies, digitalization has indeed increased in the factories and enterprises from the manufacturing domain in the last few years. This opens new opportunities for companies to collaborate, share their data, reuse resources, find new suppliers, and optimize their supply chains in order to deliver innovative solutions. However, enabling collaboration among the manufacturing companies to realise innovative B2B scenarios is still challenging because of the interoperability gaps between their digital resources such as tools, services, systems, platforms, and data APIs.

Today's digital manufacturing platforms (DMPs) are largely heterogeneous, vendor-specific, functionality-wise fragmented, vertically oriented silos of resources that are locked behind their private identity and access management solutions. Mass production continues to be the norm in the manufacturing domain. Each company at any level of the supply pyramid, as illustrated in Figure 1, has its small number of preferred suppliers usually from its local cluster. For example, in the aerospace sector, the large OEMs such as Airbus at the top of the supply pyramid have streamlined their supply chains to include only a small amount of preferred large suppliers such as Thales, Honeywell, etc., as 1<sup>st</sup> tier suppliers. The SMEs often do not have the necessary digital infrastructure (e.g., ERP systems) needed to directly become a part of the supply chain network of the large OEMs. In addition, other constraints such as extensive risk sharing requirements, complex procurement and collaboration procedures, rules and diversity of IT systems create strong barriers for SMEs to become a 1<sup>st</sup> tier supplier of the OEMs. On the other hand, the customers' demands of highly customised products requires a close collaboration among the OEMs and small but high-tech SMEs that can deliver such products within a very short time. However, the formation

of such ad hoc collaborative clusters becomes very challenging because of the underlying heterogeneity of available digital resources.

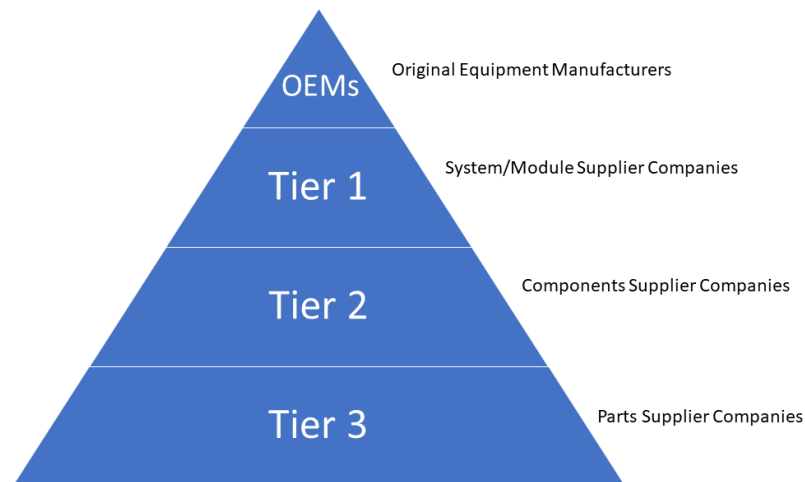


Figure 1. Supply Pyramid

In the furniture manufacturing sector, the companies are under pressure to meet the market demands of highly customised products. The new business models require tools for facilitating search and selection of suppliers and/or products with certain characteristics, which allow monitoring of manufacturing processes through supply chain transparency, coordination of deliveries, and planning of both internal and external activities throughout the supply network. In the circular economy scenarios, the formation of Closed-Loop Supply Chains (CLSC) requires collaboration between suppliers, consumers, and prosumers such as waste producing companies, waste management companies, bio-energy companies, etc., from different domains as illustrated by the CLSC pilot scenario in Figure 2. Such objectives of industry 4.0, lot size one and sustainable manufacturing can be achieved by creating an ecosystem of DMPs that enables the integration of heterogeneous platforms, interoperability among their services and the creation of cross-platform applications in an easy way.

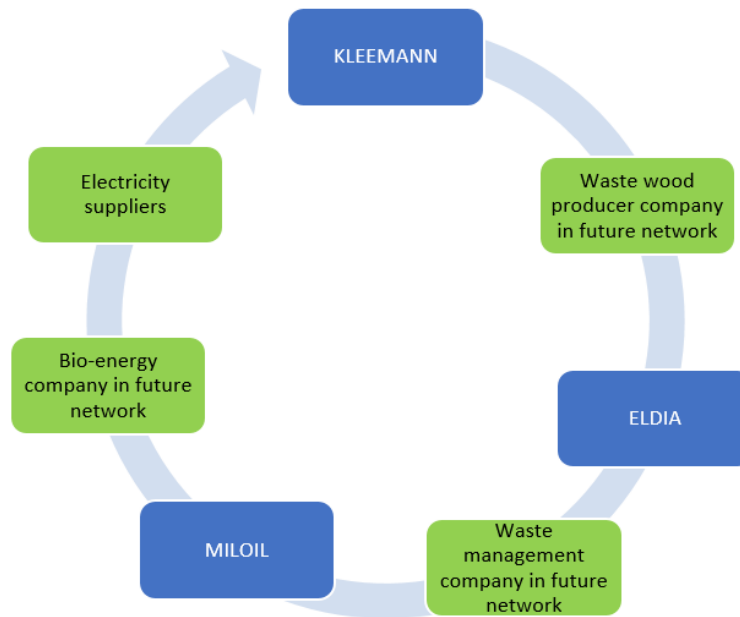


Figure 2. CLSC scenario in EFPF Circular Economy Pilot

The objective of the EFPF ecosystem is to interlink the heterogeneous DMPs and enable interoperability, communication and sharing of resources in order to enable companies to make a transition from traditional mass production to a lot-size-one manufacturing. The primary objectives of the EFPF architecture definition task are to make the design of the EFPF ecosystem modular, scalable, and extensible and to define the enablers and the methodologies that are necessary for its creation and for sustaining its operation.

### 1.1.2 Offerings and Stakeholders

Figure 3 illustrates the core stakeholders that interact with the ecosystem and the offerings provided by them. In general, all the stakeholders except for the Ecosystem Administrator (Admin) are referred to as users. A user is the representative of his/her company and interacts with the ecosystem on behalf of the company. The offerings and the stakeholders are defined below.

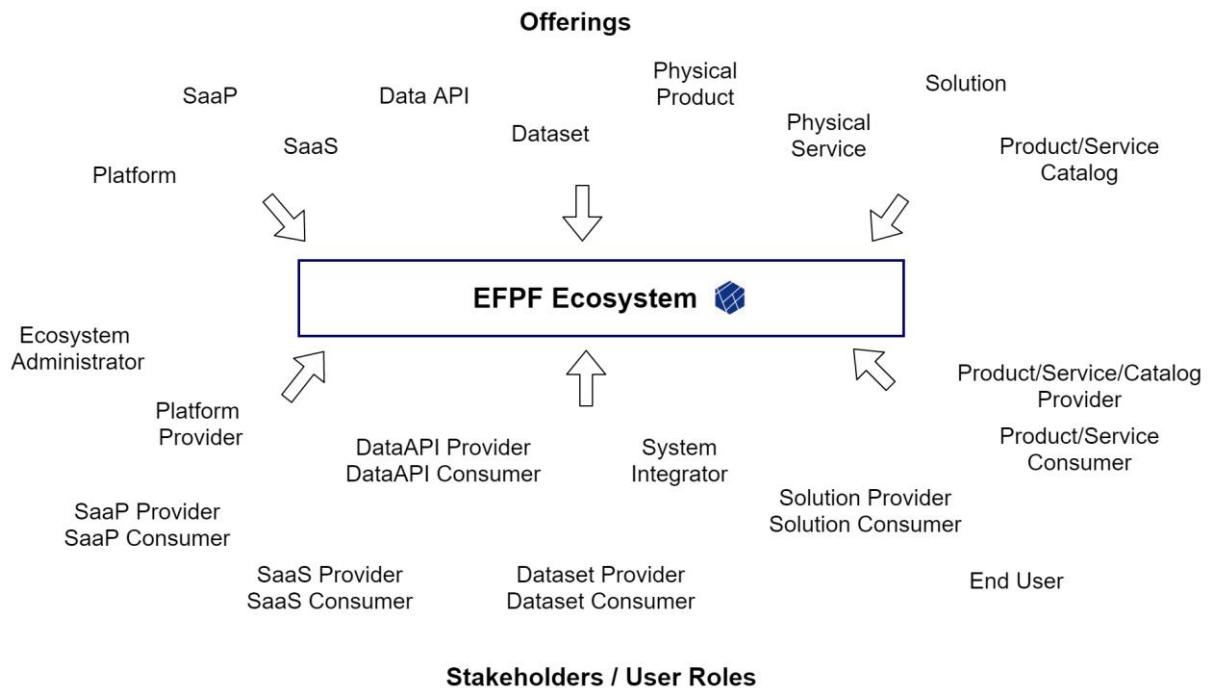


Figure 3. Offerings & Stakeholders in the EFPF Ecosystem

### Offerings:

- Platform: A digital platform that provides offerings such as digital tools, services and data and secures access to them by using its own Identity and Access Management service.
- Tool/Service/Data:
  - Software as a Product (SaaP): Software tools that are sold/given to users as products and therefore, they generally intend to follow a one-time pricing/licensing model. E.g., Factory connectors or IoT Gateways such as TSMatch, Symphony HAL, Industweb Collect or other products such as Google Chrome browser; Microsoft Office 2010, etc.
  - Software as a Service (SaaS): Software tools/services centrally hosted on cloud, e.g., GitHub, Skype, Docker Hub, Microsoft Office 365, etc., which generally intend to follow the subscription-based or access-based (pay-as-you-go) pricing/licensing model. This includes the tools/services that make processed or value-added data available to other users over an API.
  - Data API: Data provided over an API. E.g., sensor measurements available over an MQTT topic, weather data available over an HTTP API, etc.
  - Dataset: Data provided in the form of a downloadable blob. E.g., historical stock data downloadable as a CSV file.
- Physical Product: A physical product offered by a company. E.g., a wooden cupboard.
- Physical Service: A physical service offered by a company. E.g., transport and logistics service.
- Product/Service Catalogue: A catalogue of physical products and services provided by one or more companies.

- **Solution:** A solution provided by a company or a group of companies to the requirements specified by another company or a group of companies. The solution provision/consumption workflow often involves a tendering mechanism. A tender is an offer to provide solutions such as carrying out work, supply goods, or another asset, etc., at an agreed price.

#### **Entities:**

- **Application:** The final outcome of a defined use case scenario that is used by the end user, e.g., the pilot applications. A Solution can be an Application or a part of an Application.
- **Teams & Projects:** The users of different companies can create groups or 'teams' that work collaboratively on 'projects' for the realisation of a Solution or an Application.

#### **Stakeholders / User roles:**

- **Ecosystem Administrator (Admin):** A user who has administrator-level access to the deployments and the APIs of the Ecosystem Enablers.
- **Platform Provider:** A user who is the provider of a digital platform.
- **Tool/Service/Data Provider:**
  - **SaaP Provider:** A user who is the provider of a downloadable SaaP software.
  - **Service Provider:** A user who is the provider of a SaaS software.
  - **Data API Provider:** A user who is the provider of a data API.
  - **Dataset Provider:** A user who is the provider of a downloadable dataset.
- **Tool/Service/Data Consumer:**
  - **SaaP Consumer:** A user who is the consumer of a downloadable SaaP software.
  - **Service Consumer:** A user who is the consumer of a SaaS software.
  - **Data API Consumer:** A user who is the consumer of a data API.
  - **Dataset Consumer:** A user who is the consumer of a downloadable dataset.
- **Product/Service/Catalogue Provider:** A user who provides a physical product, a physical service or a catalogue of physical products and services.
- **Solution Consumer:** A company seeking solution for its requirements or use cases. This often involves a tendering mechanism. Therefore, it involves the Solution Consumer companies inviting tenders for solutions required by them, evaluating the submitted bids, accepting a tender, facilitating the solution provider to work on the project of delivering the solution and finally, consuming the provided solution.
- **Solution Provider:** A company providing solutions to the Solution Consumer companies such as consultancy services (includes analysis, design), implementation (includes sensor/actuator installation, application development, etc.), deployment, maintenance, administration, etc. This often involves a tendering mechanism. Therefore, it involves the Solution Provider companies looking up the tender invitations submitted by Solution Seekers/Consumers, submitting tender bids, working on projects emerging from the

accepted tenders to deliver the agreed solution, and finally, delivering the solution to the Solution Consumers.

- **System Integrator:** A user who integrates a resource (i.e., a platform, a tool/service, or a data API, etc., on behalf of the resource providers) with the ecosystem or creates composite applications using the existing resources. The System Integrator is a general user role that can be a combination of one/more of the user roles described above. E.g., a System Integrator could be a Tool Provider who him/herself integrates their tool with the ecosystem. The user role ‘Composite Application Developer’ is also associated with this user role.
- **End User:** The consumer or beneficiary of the Application that realises the use case scenario defined by him/her/them on behalf of their company/companies.

In addition to these user roles defined at the ecosystem level, the connected platforms can have their own internal user roles. When interacting with the ecosystem, their user roles translate to one/more of the user roles defined above. Moreover, various use case scenarios, projects and solutions can define their own user roles. For example, the functional user roles defined in the aerospace, furniture manufacturing and circular economy pilot scenarios in the EFPF project can be found in the following deliverables respectively: “D9.1: Implementation and Validation through Pilot-1”, “D9.2: Implementation and Validation through Pilot-2”, and “D9.3: Implementation and Validation through Pilot-3”.

### 1.1.3 Challenges, Requirements, and Core Components

The deliverable ‘D2.1: Project Vision and Roadmap for Realising Integrated EFPF Platform’ defined the vision and roadmap for realising the integrated EFPF ecosystem at M3 and ‘D2.2: Platform Interoperation Challenge Report’ identified the initial interoperation challenges at M5 of the project. In addition, ‘D2.3: Requirements of Embedded Pilot Scenarios’, at M5, defined the initial requirements of the three embedded pilot scenarios, which were added to Fraunhofer FIT’s JIRA as a live document and elaborated further. The architecture definition task identified the following major challenges that need to be addressed in order to fulfil these requirements:

- How to integrate the heterogeneous platforms, tools, and services provided by independent entities to form the EFPF ecosystem and enable communication among them while ensuring that the system is scalable and extensible?
- How to enable the creation of cross-platform applications in an easy manner, without making any changes to the existing tools and services?
- What core components are needed to ensure that the ecosystem keeps running and providing the core functionality even if the connected tools/services/platforms are disconnected from the ecosystem?
- How to enable the realisation of specific use case scenarios from the manufacturing domain such as the pilot applications?

To effectively address these challenges in order to fulfil the requirements, the architecture definition task classified the components required into two major categories:

1. **Ecosystem Enablers:** The common core components that are required to integrate the heterogeneous platforms, tools and services provided by independent entities to form the EFPF ecosystem and enable communication among them while ensuring that the system is scalable and extensible.



**2. Smart Factory Tools and Services:** The components from the connected platforms that provide the use case specific functionality required to realise the domain-specific pilot and Open Call experimentation scenarios. The providers of these components can make use of the Ecosystem Enablers to manage their development and operations lifecycle. The System Integrators make use of the Ecosystem Enablers to integrate these components with the ecosystem, to establish communication with the other components and to create composite applications.

Section 1.1.3.1 below identifies the requirements that should be fulfilled by the Ecosystem Enablers, whereas Section 1.1.3.2 maps the pilot requirements to the tools and services in the EFPF ecosystem.

### 1.1.3.1 Requirements for Ecosystem Enablers

The Ecosystem Enablers are the building blocks that enable the creation and functioning of the ecosystem. The EFPF architecture categorises the Ecosystem Enablers into 6 types based on the functionality they should offer. The requirements for each type are listed below:

1. Data Spine: Identity Federation, Cross-Platform Interoperability & Service Composition
  - Creation of a holistic framework for security, privacy, and management of data as well as users
  - Enabling Single Sign-On (SSO) across the connected platforms
  - Enabling service-level cross-platform interoperability
  - Ensuring that the defined interoperability mechanism enables the ecosystem to be scalable and extensible
  - Methodology and tooling support for an easy creation of cross-platform applications
2. DevOps, Maintenance & Support
  - Platform for source code, deployment, and configuration management
  - Tools and procedures for infrastructure management such as monitoring and data backup and recovery routines
  - Documentation and support for users
3. API Management
  - Easy provisioning, lifecycle management and discovery of service metadata
  - Uniformity across and completeness of API specifications
  - Delegation of access consent requests directly to the service/data providers
  - Management of interface contracts among service providers and consumers
4. Unified Functionality
  - Single point of entry to the ecosystem

- Availability of coherent functionality at the ecosystem-level, e.g., the uniform view of all the offerings of the connected platforms, a unified/integrated marketplace, etc.

## 5. Essential Platform-Based Functionality

- Identification of functionality that is provided by one/more of the connected platforms, but is necessary for the realisation of multiple use cases from the manufacturing domain

## 6. Governance Rules & Trust Mechanisms

- Definition of effective governance mechanisms to enable the ecosystem to reach its major goals and create sustainable outcomes
- To ensure that the governance mechanisms reflect on the lawful interactions of key stakeholders, be they owners of the platforms, companies using the platform, or developers, users, advertisers, economists, computer scientists, governments, or regulators

Thus, the Ecosystem Enablers provide the core functionality that enables the smart factory tools, services, and platforms to integrate with the ecosystem, enable communication and the creation of composite applications to realise various smart factory applications and use case scenarios such as the pilot applications. Some generic use case scenarios that, for example, involve searching for product/service supplier companies across platforms, can be realised directly using the Ecosystem Enabler components. Whereas, for realising the use case scenarios that involve specific requirements such as predictive maintenance, risk detection, supply chain transparency, or creation of a customised dashboard, etc., the smart factory tools and services from the connected platforms can be used.

The next section presents a mapping of the pilot use case scenarios to the smart factory tools and services.

### 1.1.3.2 Mapping of Pilot Requirements to the Smart Factory Tools & Services in the EFPF Ecosystem

From the pilot requirements documented in Fraunhofer FIT's JIRA as a live document which were described in the pilot deliverables D9.1, D9.2 and D9.3, various pilot solutions were defined and implemented. Figure 4 (adapted from the tables in the pilot deliverables) illustrates the mapping between these pilot solutions and the Smart Factory Tools and Services. The details from the live document such as the EPICs, the user stories, and their implementation and validation can be found in the pilot deliverables.

No	Solution	Relates to Pilot	Smart Factory Tools and Services
<b>S 1a</b>	Solution 1a: Production Optimisation (Predictive Maintenance)	Furniture	Industreweb Collect Factory Connector, FCGMT, Anomaly Detection Solution (ADS), Visual & Data Analytics Service, Deep Learning Toolkit (DLT), ROAM Tool, Secure Data Storage Solution
<b>S 1b</b>	Solution 1b: Production Optimisation (Operator Error)	Furniture	Industreweb Collect Factory Connector, FCGMT, Industreweb Display
<b>S 2</b>	Solution 2: Bin Fill Level Monitoring	Furniture CE	Visual and Data Analytics Tool, Symphony HAL, Symphony Event Reactor

<b>S 3</b>	Solution 3: Workflow and Service Automation Platform	Furniture Aero-space	WASP
<b>S 4</b>	Solution 4: Matchmaking Service	Aero-space CE	Base platform Marketplaces, Federated Search, Product Catalogue Service, Business Opportunities Service
<b>S 5a</b>	Solution 5a: Efficient Resources Management Solutions (Visual Detection)	Aero-space	Industreweb Collect Factory Connector, AI Vision Service (FC component) Secure Data Storage Solution
<b>S 5b</b>	Solution 5b: Efficient Resources Management Solutions (Stores Monitoring)	Aero-space	Industreweb Collect Factory Connector, ROAM Tool
<b>S 6</b>	Solution 6: Workplace Environment Monitoring	Aero-space	TSMatch Gateway Factory Connector, Symphony Platform
<b>S 7</b>	Solution 7: Tendering & Bid Management	All domains	Federated Search, Business Opportunities Service
<b>S 8</b>	Solution 8: Almende Risk Analysis & Management (ROAM) Tool	All domains	ROAM Tool
<b>S 9</b>	Solution 9: Catalogue Service	All domains	Product Catalogue Service
<b>S 10</b>	Solution 10: Business Network Intelligence	All domains	ValueChain Network Portal (previously known as 'iQluster') SDK Business Intelligence App
<b>S 11</b>	Solution 11: Data Analytics	CE	Visual and Data Analytics Tool, Deep Learning Toolkit
<b>S 12</b>	Solution 12: Blockchain Application	CE Aero-space	DAML, Blockchain DApp (web and mobile application)
<b>S 13</b>	Solution 13: Online Bidding Process	CE	Matchmaker, Agents, Marketplace
<b>S 14</b>	Solution 14: System Security Modelling	CE	SSM Tool

Figure 4. Mapping of Pilot Requirements to the EFPF Ecosystem Components

It should be noted that in the latest version of the architecture presented in this document, the FCGMT (Factory Connector Gateway Management Tool) has been replaced by the Pub/Sub Security Service. The smart factory tools and services mentioned in the table are described in Sections 1.3 and 3.2.

## 1.2 EFPF Architecture Context View

Figure 5 presents an overview of the high-level architecture of the EFPF ecosystem that consists of the Ecosystem Enablers and tools, services, and platforms from various providers. In contrast to the high-level architecture diagrams from the previous deliverable D3.11, the architecture in Figure 5 extends the central box of Data Spine to include more components which are collectively called as 'Ecosystem Enablers'. The Ecosystem Enablers are the core components that enable the creation and the functioning of the ecosystem. In addition, it consists of separate blocks for tools/services/data APIs indicating that the

ecosystem enables the integration of individual tools and services together with full-fledged platforms.

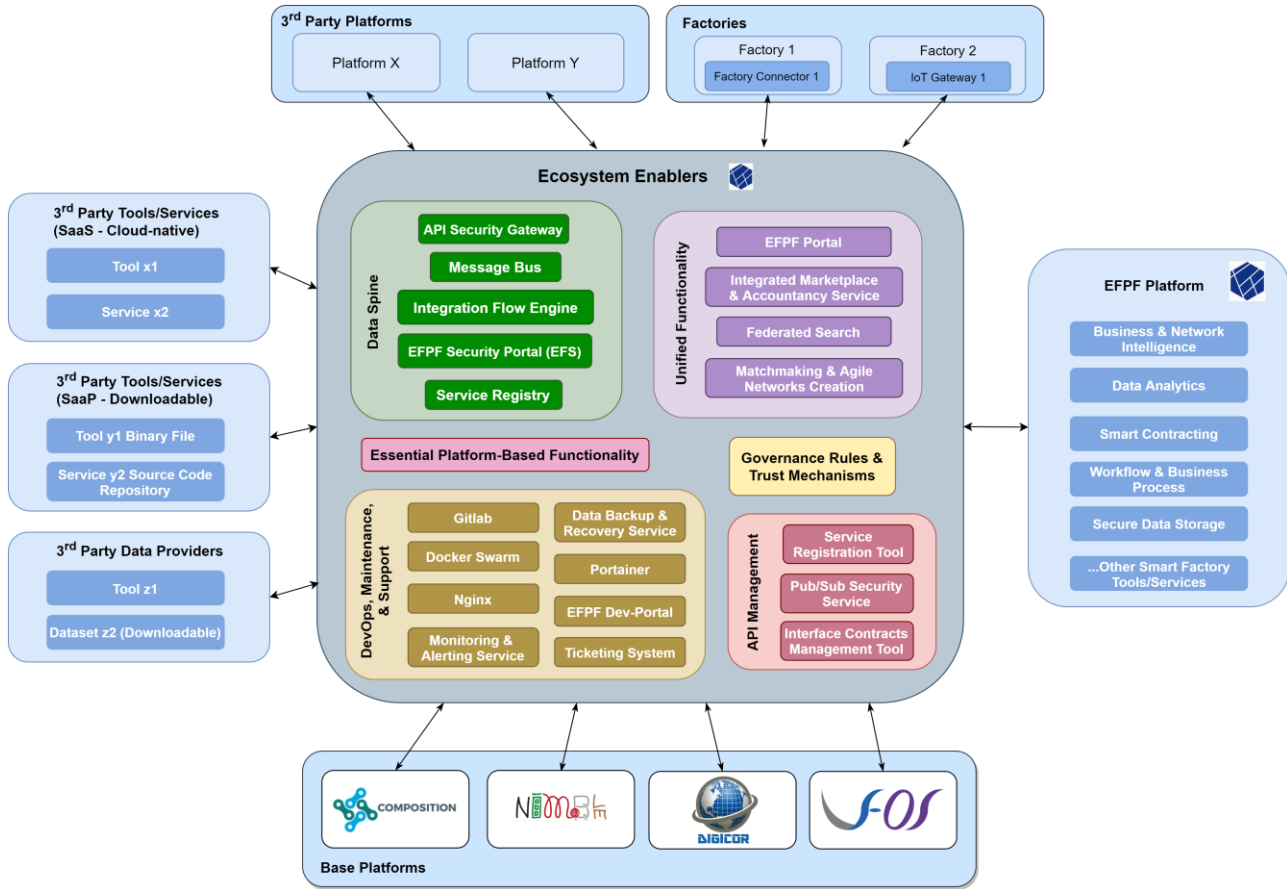


Figure 5. High-level Architecture of the EFPF Ecosystem

The EFPF platform follows the microservices architecture approach in which different functional modules implement individual functionalities that can be composed based on specific user needs. In order to implement this approach, all components in the EFPF ecosystem are prescribed to implement and publish open interfaces, preferably REST interfaces, allowing the exchange of data.

The EFPF ecosystem is designed considering the federation approach where the distributed heterogeneous digital manufacturing platforms developed, provided, and managed by different independent entities permit the creation of added value within the ecosystem. To enable communication among them, an integration and communication layer, i.e., the Data Spine that acts as a translator/adaptor between them is used. In addition, the rest of the Ecosystem Enablers provide the common core functionality and the digital infrastructure that is needed for the efficient operation of the ecosystem. Thus, the EFPF ecosystem follows the Service-oriented architecture (SOA) style. The main elements in the EFPF federation are:

- Ecosystem Enablers:** In the previous version of the EFPF ecosystem architecture, the Data Spine, that provides the interoperability infrastructure that interlinks and establishes interoperability between heterogeneous tools, services, and platforms and enables the creation of composite applications was illustrated as the only central core entity. In the latest version of architecture presented in this final report, the architectural vision was extended beyond interoperability and service composition to also include

DevOps for easy deployment, clustering for high availability, automation for better usability, and components for an efficient infrastructure monitoring, management, operations, etc.

- **Data Spine:** This Ecosystem Enabler is the central entity or gluing mechanism in the EFPF federation. The Data Spine provides the interoperability infrastructure that initially interlinks and establishes interoperability between the four base platforms: COMPOSITION, DIGICOR, NIMBLE and vf-OS (see D3.1 for more details). It adheres to common industry standards and follows a modular approach to enable the creation of a modular, flexible, and extensible ecosystem. Therefore, it can be easily extended beyond interconnecting the base platforms to “plug in” new 3<sup>rd</sup> party platforms and interlink them with the already connected platforms. Figure 5 also highlights the platform agnostic nature of the Data Spine, i.e., it is evident from the high-level architecture that as far as interactions with the Data Spine are concerned, there is no distinction between the EFPF platform and the base platforms or any other 3<sup>rd</sup> party platforms. Thus, the Data Spine would be independent from the rest of the EFPF platform. This hypothetically means that even if the EFPF platform were “switched-off” in the future, the Data Spine would not be affected and therefore would continue to support an interconnected ecosystem.
- **EFPF Platform:** This is a digital platform that provides unified access to dispersed (IoT, digital manufacturing, data analytics, blockchain, distributed workflow, business intelligence, matchmaking, etc.) tools and services through the Ecosystem Enabler called ‘EFPF Portal’ that acts as the single point of entry for the ecosystem. The tools and services brought together in the EFPF platform are the market ready or reference implementations of the Smart Factory and Industry 4.0 tools from the EFPF project partners. The collection of enhanced versions of such tools and services from the base or 3<sup>rd</sup> party platforms deployed together as microservices would constitute the EFPF platform. These micro-services are made accessible through the EFPF Portal using the Single Sign-On (SSO) functionality offered by the Data Spine.
- **Base Platforms:** The EFPF ecosystem is created by initially interlinking the four digital manufacturing platforms from the European Factories-of-Future (FoF-11-2016) cluster focused on supply chains and logistics [DMC22]—namely NIMBLE [NIM22], COMPOSITION [COM22], DIGICOR [DIG22], and vf-OS [VFO22]. These are termed as the ‘Base Platforms’. The base platforms provide functionality that is complementary to each other with minimum overlap and hence by interlinking them, the EFPF ecosystem is able to offer a comprehensive set of business functions.
- **3<sup>rd</sup> Party Platforms:** In addition to the four base platforms, the EFPF ecosystem enables interlinking of other 3<sup>rd</sup> party platforms that address the specific needs of connected smart factories. The examples of 3<sup>rd</sup> party platforms that joined the EFPF ecosystem include ValueChain’s Network Portal platform [VLC22], Nextworks’ Symphony platform [NXT22] and SMECluster’s Industreweb platform [C2K22].
- **3<sup>rd</sup> Party Tools, Services, and Data:** The EFPF ecosystem can also be extended by connecting individual tools, services, and data APIs, etc. that do not belong to an existing platform.

### 1.3 EFPF Architecture Functional View

The EFPF ecosystem architecture categorises the components into two main types depending upon the functionality they offer as illustrated in Figure 5. The Ecosystem Enablers provide the core functionality while the other tools and services from the connected platforms provide functionality that is used to realise specific use case scenarios from the

manufacturing domain. This section provides a functional overview of these components and detailed description of their interfaces can be found in Section 2.10.1.

### 1.3.1 Functional Overview of the Ecosystem Enablers

The Ecosystem Enablers are responsible for providing the core functionality that enables the efficient (re)use of the smart factory tools and services to realise various use case scenarios regardless of the platforms they belong to. This core functionality is grouped into 6 different categories as illustrated in Figure 6.



Figure 6. Overview of the Ecosystem Enablers' Functionality

#### 1.3.1.1 Identity Federation, Cross-Platform Interoperability & Easy Service Composition

In the EFPF ecosystem that connects heterogeneous platforms, the services of these platforms are behind their closed Identity Providers and hence, not directly accessible to the users of other platforms. Moreover, as the platforms are developed and provided by independent entities, their services have interoperability gaps at the levels of interfaces, communication protocols, data formats, data models, etc. Therefore, enabling uniform access to the cross-platform services by bridging the interoperability gaps among them becomes a prerequisite to enabling the creation of cross-platform applications. The Interoperable Data Spine is a federated interoperability enabler that provides these functionalities in the EFPF ecosystem.

## Identity Federation & Cross-Platform Interoperability

The Data Spine is the interoperability backbone of the EFPF ecosystem that interlinks and establishes interoperability between the services of different platforms. It is aimed at bridging the interoperability gaps between services at the following levels:

- **Security interoperability:** The Data Spine federates the Identity Provides of the connected platforms, enables SSO among them that facilitates the EFPF ecosystem users to access the resources of the connected platforms with as single set of credentials.
- **Protocol interoperability:** The Data Spine supports two communication patterns:
  1. Synchronous request-response pattern
  2. Asynchronous Pub/Sub (publish-subscribe) pattern

While the Data Spine supports standard application layer protocols that are widely used in the industry, it employs an easily extensible mechanism for adding support for new protocols. For supporting lower-level protocols and IoT networking technologies such as BLE (Bluetooth Low Energy), Z-Wave, ZigBee, LoRa (Long Range), etc. the Data Spine relies on the Factory Connectors and IoT Gateways deployed at the edge.
- **Data Model interoperability:** The Data Spine provides a mechanism to transform between the message formats, data structures and data models of different services thereby bridging the syntactic and semantic gaps for data transfer.
- **Interaction approach interoperability:** Even when two different services use the same communication protocol, there might be a mismatch between their interaction approaches. For example, one service might make its data available at an HTTP GET API endpoint, while the other service provides a webhook functionality that expects data over an HTTP POST endpoint instead. Another example could be that the services of once platform expect separate steps for discovery and data retrieval, while the services of another platform combine these steps and provide a direct querying functionality for data retrieval instead. Such gaps between the interaction approaches followed by different services are bridged by the Data Spine.

## Easy Service Composition

Together with enabling interoperability, enabling the integration of tools and services with the ecosystem and the creation of cross-platform applications in an easy and effortless manner is crucial for realising the use cases from agile manufacturing. This way, the existing tools, services, and solutions can be applied to realising similar use cases in different domains, contexts, companies, or factories with minimal cost, time, and effort.

The Data Spine:

- Defines a standardised approach, methodology and tooling support for enabling the creation of composite applications in an easy and intuitive manner, while ensuring that the system remains modular, scalable, and extensible.
- Provides the necessary technical infrastructure to ensure that the existing services can be composed together without (1) any additional local deployments and (2) the need to make any changes to their code or deployment configuration, regardless of the platform they belong to, or the location they are deployed at.
- Enables collaboration among the users from different platforms or companies for the creation of applications.

## Data Spine Components

The Data Spine consists of the following components:

- **The EFPF Security Portal (EFS)** component is responsible for providing a SSO facility across the ecosystem. It enables the users to access the resources, i.e., tools, services, and data, of the connected platforms with a single set of credentials. The EFS is realised using Keycloak.
- **The Integration Flow Engine (IFE)** component of the Data Spine provides a platform to the users, allowing them to create dataflows or “integration flows” for interconnecting the APIs of different services in order to create composite applications. The IFE is realised using Apache NiFi.
- **The Message Bus** component is used for mediating the transfer of messages or data between asynchronous services communicating through the Data Spine. The Message Bus is realised using RabbitMQ.
- **The Service Registry** component allows the service providers to register their service endpoints and metadata such as API specifications. The Service Registry provides a facility for the service consumers or composite application developers to discover these services and retrieve their metadata information, which is required to create the integration flows. The Service Registry is realised using LinkSmart Service Catalog.
- **The API Security Gateway (ASG)** component acts as the policy enforcement point (PEP) for the APIs that are exposed by the integration flows created by users. It can also be used as a permalink reverse proxy endpoint for the services in the ecosystem whose direct endpoints can change. The ASG is realised using Apache APISIX.

### 1.3.1.2 DevOps, Maintenance & Support

The Ecosystem Enablers are central components in the EFPF ecosystem that are used by multiple smart factory tools, services, and platforms. Ensuring their rapid deployment, integration, version upgrades, efficient maintenance and high availability is of utmost importance. In addition, providing documentation such as user guides and asking for support with the integration activities is necessary. With multitenancy support, the same DevOps, Maintenance and Support infrastructure can also be made available to the providers of smart factory tools, services, and platforms that have similar requirements.

The DevOps, Maintenance and Support infrastructure provides the following functionalities:

- Development Management
  - Source Code Version Control System that enables collaboration among software development teams
  - Project Management features such as grouping and sub-grouping of projects, wiki for internal documentation and issue tracking functionality for management of technical issues, milestone management functionality for project planning and release management at group-level, etc.
  - Multitenancy and a fine-grained access control
- Deployment Management
  - Continuous Integration (CI), Continuous Delivery (CD), Continuous Deployment (CD)



- Container Registry
- Remote Deployment & OTA (Over-the-Air) Updates
- Infrastructure Management
  - Web Server for Reverse Proxying and URL Path Management
  - Remote Deployment Management Interface
  - Monitoring & Alerting
- Documentation & Support
  - Documentation Portal
  - Ticketing System for users to report technical issues and get support

The technologies used to realise the DevOps, Maintenance and Support infrastructure are illustrated in Figure 5. The detailed descriptions of these components can be found in Section 3.2.1 and also in the deliverables D6.1 (M12), D7.1 (M18), D6.2 (due at M42) and D7.2 (due at M48).

### 1.3.1.3 API Management

In the EFPF ecosystem, the services communicate with each other by consuming their APIs. APIs are the contracts between service providers and service consumers that determine how a particular service can be consumed. Therefore, the management and discovery of service APIs is crucial for enabling service-level communication. In the EFPF ecosystem, the API management process is concerned with creating, publishing, monitoring, and securing the APIs of the smart factory tools, and services as well as the Ecosystem Enablers.

#### 1.3.1.3.1 API Metadata Management: Service Registry & Service Registration Tool

In the interconnected EFPF ecosystem, the services of different platforms need to be composed together to achieve common objectives. For this purpose, the service consumer users should be able to discover the available services and consume them without the active involvement of the service provider users. The Data Spine Service Registry provides the following functionalities to fulfil these requirements:

- Registration and lifecycle management of service/API metadata for synchronous (Request-Response) as well as asynchronous (Pub/Sub) services in a uniform manner
- The discovery, lookup, and filtering of services
- Use of industry standard API specifications (spec) to capture service metadata to ensure the completeness of and uniformity across the API descriptions

The Service Registration Tool, the front end for the Service Registry, provides:

- An easy-to-use GUI for service registration, and
- API spec validation functionality.

#### 1.3.1.3.2 Access Consent Delegation Framework: Pub/Sub Security Service

The EFPF ecosystem consists of several smart factory tools and services that make IoT data available to be consumed by other by publishing it to the Data Spine Message Bus. As

per the traditional approach, the data consumers need to ask permission from the administrator in order to access this data. The administrators need to check with the data owners before granting the access permission. Therefore, the process becomes slow and cumbersome, and the data owners do not have direct control over who accesses their data. The Pub/Sub Security Service provides the access consent delegation functionality to automate this process and take the administrator out of the loop, enabling the data publishers and the potential data consumers to directly communicate with each other.

In the EFPF ecosystem, the users, tools, and services often require the ability to communicate in an asynchronous fashion. To facilitate these requirements, the Message Bus (DS RabbitMQ) is offered as part of the Data Spine. However, this presented three main challenges within the ecosystem:

- How to enable the synchronisation of user accounts, security mechanisms, and terminologies between the EFS and RabbitMQ
- How to manage the lifecycle of resources (topics, queues, exchanges, binding keys), vhosts
- How to manage access control for the resources (topics, queues, exchanges, binding keys), vhosts, users

The EFPF Pub Sub Security Service has been provisioned in the ecosystem that provides the following functionalities to address these challenges:

- Automated synchronisation of EFS and RabbitMQ accounts.
- Register and manage resources requiring access to the Message Bus.
- Create and Manage Topics in the EFPF Message Bus for resources to publish or subscribe to.
- Discover and request permission to consumer topics created by other resources in the tool.
- Manage permission requests to owned topics, including approve, reject, and revoke operations.

#### **1.3.1.3.3 Interface Contracts Management Tool**

APIs serve as the contracts between Service Providers and Service Consumers and a successful collaboration depends on adhering to them. In some cases, the changes to APIs can disrupt the communication. The Interface Contract Management Tool (ICMT) is useful for tracking changes to the APIs of the tools and services in the EFPF ecosystem. Furthermore, it helps the API consumers in keeping track of the software interfaces they make use of.

#### **1.3.1.4 Unified Functionality**

The EFPF ecosystem consists of full-fledged platforms that have their own portals, marketplaces, etc., that display their offerings in a variety of formats and interfaces and enable their sale. The EFPF ecosystem offers the following unified functionalities to its users:

- A single point of entry
- A unified view of all the offerings of the connected platforms

- A unified search interface to search for potential collaborators and their offerings by using custom, user-defined search filters
- A unified accountancy mechanism that enables them to track and trace their purchases across the platforms
- An ability to collaborate with users regardless of the platforms or companies they belong to

#### 1.3.1.4.1 EFPF Portal

The EFPF Portal component is the unification point of distributed tools and platforms in the EFPF ecosystem. It allows the user to access connected tools, base platforms, marketplaces, experiments, and pilots through a unified interface. The EFPF Portal is accessible at: <https://portal.efpf.org/> and provides the following functionalities to fulfil these requirements:

- Modern UI web application
- Keycloak integration for authentication and authorization
- Integration of EFPF Marketplace providing products from connected marketplaces
- Provisioning of Value Proposition pages for improved experience of new users
- Provisioning of Tools and other EFPF-related applications via
- Integration of EFPF Portal Backend providing
  - Accountancy Service integration for Telemetry and other event logging
  - MailJet integration for User management email notifications

#### 1.3.1.4.2 EFPF Marketplace: Integrated Marketplace & Accountancy Service

##### Integrated Marketplace

The Integrated Marketplace provides access to products listed on connected marketplaces at different platforms provided by the EFPF partners. The Integrated Marketplace provides the following functionalities to fulfil these requirements:

- Integration of Service Registry for retrieving external marketplaces dynamically
- Displaying products in a unified way
- Providing pagination and filtering options

##### Accountancy Service

Tracking the user behaviour enables businesses to make productive decisions and develop effective business strategies, since it allows to understand and focus on the needs of the customers and help to become more user-oriented. Moreover, tracking the money flow within large business networks and ecosystems and federated digital platforms pose different challenges that need to be addressed. With these goals in mind, the Accountancy Service is developed as a part of the EFPF Marketplace Framework and provides insight into users' interactions with the EFPF Platform by tracking & tracing the user journey, particularly any transactions that EFPF users make on different marketplaces linked with the

EFPF Marketplace Framework. The Accountancy Service addresses the abovementioned requirements by implementing following mechanisms:

- The Accountancy Service collects data about the specific actions of the user to determine their journey across the EFPF ecosystem. The collected data are then used to carry out a cashback mechanism which enables to charge the marketplace by a commission charge or a referral fee where an EFPF user carries out a business transaction.
- The Accountancy Service provides interactive dashboards to visualize collected information and allows you to pull insights out of user behaviour and transaction data. The interactive dashboards provide advanced filtering and data analysis mechanisms to help you to reach specific information in a faster way.
- The Service offers automated tools for preparing monthly reports based on the accumulated data and generating monthly invoices in accordance with the commissions calculated for successful transactions that users perform on the marketplaces connected to the EFPF Platform.

#### **1.3.1.4.3 Federated Search**

The EFPF Marketplace combines and integrates a variety of distinct platforms each of which manages sellable items and companies or manufactures selling those items. The Federated Search Service acts as the starting point when searching products and manufacturers of all platform providers with a unified search service combining the searchable information.

For this purpose, the Federated Search Service provides the following functionalities:

- Possibility for bulk updating of platform data with a generic data model
- Provision of a unified search API for product items, companies
- Possibility to manage accompanying meta-information for arbitrary classifications, customized properties including multilingual naming.
- Provision of this accompanying meta information for user friendly search interfaces.
- Collection of the above data with the Data Spine (Apache NiFi) services consuming individual service endpoints of the distinct platforms.

#### **1.3.1.4.4 Matchmaking and Agile Networks Creation**

For the Matchmaking and Agile Networks Creation, the Federated Search Service collects both, company data and their responsible contact persons. With the federated search interface, the team formation process is initiated by selecting two or more companies (including contact persons) in order to trigger the agile network creation. The Matchmaking Service offers the following functionalities:

- Re-use the collected company data from Federated Search Service
- Ability to select two or more companies in the Federated Search Interface
- Ability to initiate the team formation process and send invitation to contact persons from the selected companies
- Visualization of existing teams and their members

Note: business opportunities will be integrated into federated search which allows users to search for an opportunity and form a team to apply for the opportunity accordingly.

### 1.3.1.5 Essential Platform-Based Functionality

The Ecosystem Enablers, in general, provide the common core functionality that is necessary for keeping the ecosystem functioning as expected. The EFPF ecosystem body (EFPF project, followed by the European Factory Foundation – EFF [EFF22]) is the “owner” of such Ecosystem Enablers and the Ecosystem Administrators are directly responsible for managing them. Whereas the Smart Factory Tools and Services from the connected platforms provide the domain-specific and use case specific functionalities and are owned and managed by their respective providers. However, there are certain sets of functionalities provided by the Smart Factory Tools and Services that are necessary for the realisation of multiple use cases from the manufacturing domain. The Essential Platform-Based Functionality category of the Ecosystem Enablers identifies such functionalities. Based on the pilot use case scenarios in the EFPF project, the Essential Platform-Based Functionality category consists of the following functionalities:

- **Factory Connectivity:** Getting the shop floor data from the sensors and making it available to rest of the Smart Factory Tools & Services in the EFPF ecosystem is crucial for realising multiple use case scenarios. The Factory Connectors and IoT Gateways available as SaaS from the connected platforms can be installed in factories and configured to push data to the Data Spine. The EFPF ecosystem consists of three different Factory Connectivity tools: (1) Industweb Collect, (2) TSMatch Gateway, and (3) Symphony HAL. Further information on these tools can be found in Sections 1.3.2.1 and 3.2.1.5.
- **Data Storage:** The data collected from the shopfloors and the processed or analysed data needs to be stored in a database to be made available to other services at a later point in time. For example, the analytics services make use of the historical data to train their machine learning models for specific purposes such as anomaly detection. The Secure Data Storage Solution (SDSS) that provides these functionalities is available as a cloud-native SaaS service that supports a fine-grained access control as well as a SaaS software that can be deployed on-premises. Further information on SDSS can be found in Sections 1.3.2.2 and 3.2.1.6.
- **Metadata Management for Physical Products and Services:** The EFPF ecosystem aims to enable a B2B collaboration. The consumers often search for companies as potential suppliers based on their physical offerings such as products or services. The Product Catalogue Service in the EFPF ecosystem, described in Sections 1.3.2.3 and 3.2.1.7, enables the suppliers to create, manage and share catalogues of products and services.
- **Tendering and Bid Management:** The EFPF ecosystem opens opportunities for companies to collaboratively work on projects and benefit from each other’s offerings and capabilities. At times, the matchmaking is not as simple as just searching for companies and forming teams. Prior negotiations to find the best suited collaborator and agreements are needed. Also, in some cases, whether the expected solution can be delivered by any of the companies in the ecosystem is not directly known. The Business Opportunities tool described in Sections 1.3.2.6 and 3.2.1.10 and the Online Bidding Process tool described in Sections 1.3.2.5 and 3.2.1.9 provide the Tendering and Bid Management functionalities for fulfilling such requirements.

### 1.3.1.6 Governance Rules & Trust Mechanisms

One of the major ingredients to enable a platform ecosystem is the choice, design, and implementation of governance mechanisms. This is all the truer for multi-sided platforms

such as EFPF, where there are several stakeholders in “producer”, “consumer” and “prosumer” roles. Interactions between these actors can be measured, monitored, and controlled at least partly, via data generated by user activity on the platform. Based on such data, the platform managers and owners are able to put in place, a system of incentives and disincentives in order to optimise the platforms utility both for its users and for its owners. With respect to the EFPF Data Spine, we focus in this report on those governance mechanisms that are supported technically – as computable functions – by the Data Spine. A more comprehensive treatment of governance overall, in EFPF is subject to a separate project deliverable.

In the following, we describe the main set of actors in EFPF, governance issues and mechanisms that are pertinent to them:

- **Manufacturing companies** (as primary EFPF Service Customers): These companies are the primary customers and thus, the *raison d'être* for EFPF. All governance mechanisms should add to the value they derive from EFPF. In the workshops carried on with user representatives, it was established that their main governance requirements were trustworthiness of business partners and trust in the platform services, mostly w.r.t data security. There are different ways in which *trustworthiness of companies* can be measured directly or indirectly. In many countries, there are creditor protection agencies that offer information on likely liquidity of companies. On platforms like EFPF, a convincing and authentic video-presentation of on-site manufacturing capabilities can help in assessing work practices. A thorough presentation of PPAP compliance<sup>1</sup> would also be an indicator, particularly in the automotive sector.

*Data security* is of course, of utmost importance for the EFPF platform itself, because it acts as the security vault for a significant part of every customer's business information. EFPF's security concept starts with identifying provision and *Keycloak* as the main access management tool. Access to partner platforms is possible via an EFPF account, but not vice versa. The platform needs to ensure data *confidentiality*, *integrity*, and *availability* to legitimate users, at the same time. The actual implementation of EFPF's security features directly dependent on the final hosting of the platform since many security aspects will be dealt with at the level of the cloud provider, leaving mostly the formulation of the platform specific access rules to EFPF itself.

- **Third Party Platforms** as external legal entities engaging with EFPF to form a larger platform ecosystem: The relationship between EFPF and 3<sup>rd</sup> party platforms is ideally one of complementarity but in some cases, there will be overlaps in services offered and the question then arises how such a cooperation/competition dilemma can be resolved by means of governance. In terms of cooperation, Single-Sign On (SSO) has been implemented to allow access to EFPF services via trusted partner platforms.
- **Third Party Tools/Services** as external legal entities offering additional utility via EFPF: 3<sup>rd</sup> party providers of tools and services are part of the overall platform offering and therefore, will be seen as part of the platform by the primary customers. In the case of problems with their offered services, this is likely to reflect badly on the platform overall. Hence, monitoring of the customer-facing performance of external tools and services is desirable, in order to prevent loss of reputation. Direct measures are *user satisfaction ratings* whereas indirect metrics could be the *service re-use rate* (existing customers coming back) or the *customer fluctuation rate* (initial customers moving to a similar, other service).

<sup>1</sup> <https://www.en-standard.eu/ppap-production-part-approval-process/>

- **Third Party Data Providers** as external legal entities offering data as a service via EFPP: Since EFPP will also provide several data points to their customers, 3<sup>rd</sup> party data providers may also (like tools and services) impact the platform’s reputation if customers are dissatisfied. Therefore, there is essentially the same need for monitoring the performance of their services and data as with external tool providers.
- **The European Factory Foundation EFF** as provider of EFPP with its:
  - Integrated Marketplace for any transactions between the above actors
  - Matchmaking & Federated Search
  - Accountancy Service
  - Permissions Dashboard
  - Monitoring & Alerting Service

The Integrated Market Place is the obvious source for contract- or payment-related transactional data among all actors on the platform. Also, the Accountancy Service that calculates revenue-shares between platform and services, will have access to such transactional data.

- **EFF as provider of Smart Factory Services & Data for:**
  - Business & Network Intelligence
  - Data Analytics
  - Smart Contracting
  - Workflow & Business Process
  - Secure Data Storage

With these add-ons, EFF provides added-value services that customers may consume and pay for, in addition to the basic services available to all users of EFPP. When seen in conjunction with the marketplace, EFF has (at least) two roles which can rise a potential conflict: as platform providers, they have a legitimate interest in monitoring third parties – for security as well as in-good-faith business reasons. As providers of services where there are also third-party providers present on the platform, there is an issue concerning a “level playing field” if the service-selling EFF uses data they have gathered through monitoring their competitors, to then improve their own services and thus, their relative competitive position. This is what the *EU Digital Markets Act* would call a “gatekeeper function” requiring special attention from regulatory bodies.

- The **Base Platforms** can be viewed as external 3<sup>rd</sup> party platform entities. With respect to governance, they play a minor role now. The only existing marketplace based on one of the base platforms is b2bmarket [AID22] and this is run as a service to their customers, by AIDIMME, with (at least currently) a strong focus on Spanish furniture manufacturers.

### 1.3.2 Functional Overview of the Smart Factory Tools & Services in the EFPP Ecosystem

This section provides the functional overview of the Smart Factory Tools and Services from the connected platforms in the EFPP ecosystem.

### 1.3.2.1 Factory Connectors & IoT Gateways

In order to utilise the functionality of the tools and services data from the numerous data sources available within, the manufacturing environment needs to be made available. To achieve this, it is necessary to utilise a Factory Connector or IoT Gateway that can interface with the specific devices, sensors, and systems the user wishes to gather data from. The connectors and gateways then communicate with the EFPF Data Spine using a predefined data model over MQTT.

In EFPF there are three implementations of Factory Connectors & Gateways, each supporting different connectivity options (including the most widely used industry standards and systems such as OPC UA, Modbus and propriety PLC protocols), and offering different functions such as data thresholds the ability to make a calculation or scaling the data values.

The following sections will describe each of the existing Factory Connectors and Gateways and their functionality, followed by the management tool used to configure the Pub/Sub communications.

#### 1.3.2.1.1 Industreweb Collect

Industreweb (IW) Collect is a high-speed data engine that interfaces with a range of systems and devices with the aim of extracting business critical data. The primary objective of IW Collect is to solve getting data from sources that may prove to be normally difficult or require a bespoke solution.

The functionalities of the Industreweb Collect Factory Connector are:

- Support for interfacing with diverse range of industrial control systems and open protocols such as OPC UA, Modbus and Ethernet/IP
- Support for interfacing with legacy equipment via the use of
- Support for interfacing with common data protocols such as SQL, REST, MQTT AMQP
- High speed logic engine to allow calculations to be made to determine actions to be carried out
- Raise alerts via SMS or Email
- Cloud based administration tools
- Interconnect Collect nodes via MQTT broker

#### 1.3.2.1.2 TSMATCH Gateway

TSMATCH is a software-based solution that supports the semantic matchmaking of IoT data to services. The main goal of TSMATCH is to automate the data supply between IoT data sources and services, while satisfying the service needs. For that, TSMATCH v1.0 [TSM22] follows a semantic similarity matchmaking approach, where semantic descriptions of sensors are matched to semantic descriptions of services, based on an ontological approach. TSMATCH is composed of the following elements:

- TSMATCH Engine, currently a server-side component of TSMATCH that can reside, for instance, on an Edge server, on an IoT gateway, or even on a Cloud server.
- TSMATCH Things Registry, corresponding to a database system (GraphDB) where Things descriptions are periodically stored.
- TSMATCH Client, an end-user application, currently available as an Android apk.



TSMATCH provides the following functionalities:

- Discovery of IoT sensors, via the use of the middleware coaty.io.
- Support for an automated integration of IoT sensors and EFPF services, via the TSMATCH engine and via MQTT.
- Interconnection to the EFPF Data Spine, via a connector developed to support interconnection between Mosquitto and RabbitMQ.
- Support for MQTT Sparkplug [BNO22] to better support industrial environments.

#### **1.3.2.1.3 Symphony Hardware Abstraction Layer (HAL)**

The Symphony Hardware Abstraction Layer (HAL), a tool offered by Nextworks, provides a unified interface to access devices connected through different low-level bus technologies. This component implements the lower layer to read/write data points (sensors and actuators) for a typical Building Management System. The HAL interacts with the devices using their own protocols, field buses and interfaces, and provides a common interface and data model to its users. More specifically, HAL provides the following functionality:

- It abstracts the low-level details of various heterogeneous fieldbus technologies and provides a common interface to its users
- It allows the development of modules that can be plugged to the HAL's core in order to extend the available fieldbuses
- It provides the necessary logic to manage the devices according to their constraints and proper optimizations (e.g., jam avoidance, timing constraints)

#### **1.3.2.2 Secure Data Store Solution (SDSS)**

The Secure Data Store Solution provides mechanisms to capture data messages sent over the Message Bus and store key values in a Timeseries Database, which is specially optimized for data queries time-based operations. Subsequently data can be retrieved and served, allowing for analysis tools to be brought online and trained on historical data. This has the key advantage to allow for analysis tools to be selected on-demand for a specific, immediate problem, as opposed to preparing a myriad of analysis tools ahead of time.

The SDSS provides the following functionality:

- Listen to channels on the message bus on behalf of the user from which to extract key datapoints into a specialized timeseries database.
- Allow users to authorize the sharing of their data with other EFPF users.
- Is deployable on-site, allowing data owners to retain physical control of the systems storing their data.

#### **1.3.2.3 Product Catalogue Service**

Product Catalogue Service is a platform for product / service publishing, and it is the main enabler of the partner discovery phase as it allows companies to introduce themselves to the EFPF platform with the products they supply and the services they provide. To achieve this, Product Catalogue Service provides the following functionalities:

- To enable users to find what they are looking for quickly, Product Catalogue Service offers publishing products with semantically relevant annotations. It makes use of generic

and sector-specific taxonomies as knowledge bases from which relevant annotations can be obtained automatically given a product category. The main taxonomy used in Product Catalogue Service is eClass which is an ISO/IEC compliant industry standard for cross-industry product and service classification. Further, available taxonomies can be extended with domain-specific taxonomies such as Furniture Taxonomy, Textile Taxonomy and Aerospace Taxonomy, as well.

- Product Catalogue Service makes use of Universal Business Language (UBL), a world-wide standard providing a royalty-free library of standard electronic XML business documents that are commonly used in supply chain operations, as the common data model since it contains appropriate data elements for catalogue/product management such as catalogues, products, product properties and so on. Moreover, products and services as well as catalogues are persisted on a UBL-compliant relational database.

The data and metadata regarding products and services are managed in different ways. While metadata are kept in a global registry; raw data, which could have varying formats, are kept in disparate repositories. Maintaining all the metadata in a single repository enables querying on products having heterogeneous structures initially. Once a product is identified, its complete, structured definition can be fetched from the respective repository.

#### **1.3.2.4 Matchmaking Service**

The Matchmaking Service in this context is the combination of the Data Collection and Data Transformation steps with the Data Spine, the storage of the aligned data in the Federated Search Service and the Team Formation procedures that are provided with the participating platforms. Thus, the Matchmaking Service can be divided into the following tasks:

1. Data gathering / federation of the relevant information and data alignment for the use with the Federated Search Service.
2. Provision of stored information with a unified search interface, and
3. Agile Network Creation with selected participants.

For more details, see Section 3.2.1.9.

#### **1.3.2.5 Online Bidding Process**

Online Bidding Process service provides an automated matchmaking mechanism for information requests from buyers to suppliers, to execute negotiations and business transactions automatically via configured agents. It is a web-based application, which achieves automated negotiations and business transactions between interested stakeholders by matching the available suppliers for a request, enabling offers submission as a live auction and suggest the best available offer based on various criteria.

The core functionalities of bidding process are:

- Companies' representation by virtual agents
- Companies/Agents communication using agents common exchange language
- Semantic modelling of companies and their services
- Semantic Matching of requesters and suppliers
- Live offers submissions during the bidding process or use of predefined ones

- Suggestion of best available offer best on multi-criteria from requester based on semantics and best score algorithms
- Easy to use interfaces for setting up an agent and participate in the online bidding process as a provider or requester

#### **1.3.2.6 Business Opportunity Tool**

The Business Opportunity Tool offered by the SMECluster platform allows Companies to post Business Opportunities so that they can be searched and allow application bids to be made by Supplier companies with the required capabilities and accreditations. Business Opportunities and Companies from the Tool are also aggregated to the EFPF platform through the Federated Search Component to allow EFPF members to find them based on keywords.

The functionalities of the Industweb Collect Factory Connector are:

- Business Opportunity creation wizard to allow procuring companies to post Opportunities
- Flexible framework to allow the following Opportunity types to be created: Procure Products, Procure Service, Offer Products, Offer Service, Raw Material / Consumable, Group Purchase
- “My Applications” dashboard for Procurers to review bid applications
- Opportunity search utility with filters based on keyword, category, and accreditations to allow suppliers to find relevant Opportunities
- Direct invitation of compatible Suppliers to apply via the company search utility
- Team formations tool for Procurers to communicate and share documents with Suppliers

#### **1.3.2.7 Business & Network Intelligence**

Within the EFPF ecosystem, user journeys between the connected platforms, tools, and services are tracked and recorded through the EFPF Accountancy Service. This availability of data presented a novel opportunity to analyse the platform’s traffic to generate and provide actionable intelligence surrounding not only the trends within the platform, but also within its ecosystem of connected platforms through the federated search functionality.

The Business & Network Intelligence Service includes the following functionalities to provide meaningful insights into the platform’s usage:

- Dashboard providing insights into overall usage of the EFPF Platform
- Dashboard providing personalised insights into a user’s usage of the EFPF platform
- Dashboard proving insights in the usage of the EFPF platform by a user’s company

Within the service, each of the mentioned dashboards provides insights into the following aspects of the platform’s usage: Logins, Search Events, Payments, Platform Visits, Tool/Service Visits, User & Company Registrations.

#### **1.3.2.8 Blockchain & Smart Contracting**

The EFPF Blockchain and Smart Contracting services provide also provide the means for applications in the EFPF Ecosystem to incorporate blockchains and smart contracts for audit

trails for manufacturing and supply chain data, product data traceability and smart contracting agile networks. It is built on the foundation of blockchain components from base platforms COMPOSITION and NIMBLE as well as adaptation of more recent advances in smart contracting. It offers integration of blockchains and smart contracting in three service levels:

- NIMBLE Blockchain: Integrated process traceability for users of the NIMBLE Platform.
- Blockchain As a Service (BaaS): A blockchain service API for managing identities, schemas, and data, providing the means to build audit trails for manufacturing and supply chain data, product passports, identity management and other immutable transaction logging applications.
- DAML Integration: to build systems with secure multi-party transactions and smart contracting agile networks, DAML applications can be built and deployed for the EFPF Ecosystem, integrated with the EFPF Security Portal (EFS).

### 1.3.2.9 Data Analytics

This section provides a functional overview of the various analytics services in the EFPF ecosystem.

#### 1.3.2.9.1 Anomaly Detection Service

Anomaly detection allows the users to find uncommon conditions or anomalies in their data that can provide actionable insight into the errors and non-desired events captured in the (machine or IoT) data. The anomaly detection solution can help the users to improve the quality of processes, spot malfunctioning equipment or spot faulty raw material, the anomaly detection service can help you. Anomaly Detection Service is powered by AI algorithms that can predict in real time sources of defects. Through the anomaly detection solution, the users can:

- Create Models for anomaly detection, using clustering or deep learning algorithms.
- Connect models to broker topics to allow for stream data to be processed.
- Create csv datasets by collecting a set number of messages from a topic.
- Test deployed models using a publisher to send messages to your own topic.
- Visualise messages processed by models.

#### 1.3.2.9.2 Visual & Data Analytics Service

The Visual and Data Analytics tool provides a complete solution for analyse and visualize various types of data. It is a web-based framework to analyse and visualize both industrial and supply chain data.

The functionalities of Visual & Data Analytics Tool are:

- Connection to a data source for analysis and visualization. The user can select one of the available connected Databases, (MongoDB, InfluxDB) brokers or to load a .csv file containing historical data
- Selection of an analytics methods in the case that multiple are available. This depends on the data source that the user has chosen to load
- Selection of a machine(s) and available sensor(s) as data sources for the visualization
- Selection of the date or range of dates for data visualization
- Selection of various available graph types for the visualization of the analysis

#### **1.3.2.9.3 Deep Learning Toolkit (DLT)**

The Deep Learning Toolkit provides an easy way to develop and integrate Deep Learning (DL) models with the EFPF platform. The DL models supported by the DLT consume time series data encoded in a JSON formatted OGC-Sensor Things data model. These models allow users to perform activities such as Predictive Maintenance or Price Forecasting. The interaction of the user with the DLT consist of:

- Managing the different DL models which the DLT can handle
- Tweak the available models on the edge and then push those to the cloud instance of the tool
- Collect the metrics to get useful insights about the performances of the models

#### **1.3.2.9.4 Customer Trend Analysis**

The Elanyo EFPF Behavioral Predictive Framework (BPF) provides predictive insights for specific business use cases, initially only for the churn prediction use case. The framework has been built to potentially serve several use cases in one coherent place, allowing users to gain predictive insights by providing data resources upon which various models can be created that utilize the framework's underlying algorithms. These models can then be used to score data, thus estimating the probabilities for different events, such as customer churn.

Currently, only one use case is offered within the framework: the Churn Prediction. This use case is the prediction of potential churners of a company. Machine learning algorithms analyse the probabilities of customers becoming potential churners and therefore no longer have a business relationship with the company. The algorithm achieves this by taking information about the customer (e.g., transaction data over the last 6 month, interactions with the company website, etc.) and by creating an instance of a model that is able to predict the outcome of a target variable which defines churned customers. This target variable is individually set based on the company's definition of a churned customer. The definition can vary from customers that stop buying the offered products to customers that stop interacting with the communication channels. The users of the BPF can later score their latest customer data in order to predict potential churners with the help of the model that has previously been created.

#### **1.3.2.9.5 Analytics Integrator Platform (AIP)**

The Analytics Integrator Platform, developed by Siemens, is a batch-oriented analytics workflow creator which allows the EFPF's users to create their own custom analytical pipelines containing various modules used in order to parse data, extract meaningful observations which can be then published in to the main EFPF communication services for others to use in further domain specific activities. The Analytics Integrator Platform provides the following functionalities to fulfil these requirements:

- Batch-oriented
- Scheduling and manual triggering
- Docker operator and containers
- Data Spine sink/source interaction

#### **1.3.2.10 Workflow and Service Automation Platform (WASP)**

The Workflow and Service Automation Platform (WASP) can be logically split into:

- Designer components
- Runtime components

Together, these components are responsible for allowing users to model multiple manufacturing workflows to orchestrate the various assets available within a collaborative framework.

The WASP Process Designer is a visual online reactive canvas allowing a business process designer to pull in existing models from a library representing the virtualised manufacturing assets. Each asset may support additional properties that can be defined. The defined workflow can consist of sub workflows and be saved and versioned within the storage as (e.g., BPMN 2.0) model definitions.

The WASP Form Designer is a visual online reactive canvas allowing the User to design forms that can be associated with process User Tasks, and which are displayed to the User in the Tasks runtime component. (See below).

The WASP Runtime is based on the Camunda open source BPMN engine (Process Engine) and a number of UI components (Control Panel & Tasks) that communicate with the Process Engine via a REST API to provide a visual layer that allows the User to manage processes & instances, and to directly interact with a process by entering information & making decisions that control the flow of the process.

#### **Major Benefits:**

- Design, execute and monitor multiple processes/workflows on the intuitive interface
- Store and reuse the designed processes/workflows
- Assign responsibilities (manual tasks) either to people in your (virtual) organisation or to external suppliers in WASP
- Introduce your own REST services in the WASP marketplace and use them in workflows
- Cloud-based service, used to design, execute, and monitor distributed workflows

#### **EFPP Services Used in WASP:**

- EFPP Service Registry
- EFPP Security Portal

#### **1.3.2.11 Software Development Kit (SDK)**

The EFPP SDK is a user-friendly visual software module used to promote the development of applications that use the EFPP services. It includes a visual software development environment with code management, parsing, execution, and testing functionalities, as well as reusable libraries, making it easy for users to compose applications. The main purpose of the SDK is to provide EFPP users a centralised and platform level facility to develop smart factory applications, either by composing the existing building blocks (e.g., open-source applications) or by custom development of new applications.

The SDK is composed by four components:

- The SDK core libraries, which centralise endpoints and services available in the EFPP project, making them available to the developers of new applications

- The SDK Studio, an Integrated Development Environment, based on state-of-the-art technologies and customised for the development of manufacturing applications using EFPF
- The SDK Frontend editor, a WYSIWYG editor intended to build the frontends of the applications being built using the SDK, including standard GUI elements like tabs, buttons and other interactive elements but also reporting elements such as graphic charts of multiple types
- The SDK Engagement Hub, a portal which has the purpose to engage the EFPF application developers into collaboration activities, allowing them to host and disseminate the source code of their developed applications with the community

#### **1.3.2.12 Risk, Opportunity, Analysis and Monitoring (ROAM) Tool**

Many users of the EFPF platform deal with real time processes, such as assembly, supply chains, and construction, involving machines and sensors that gather data.

Users are then able to process such data using the ROAM Tool by publishing it to the Message Bus. The data is processed using user-defined workflows, consisting of configurable recipes, resulting in various outputs. These include useful insights such as statistics, figures, forecasts, and notifications:

- High customizability of workflows through configurable recipes
- Workflow and recipe definition through frontend, or REST API
- Web Push & Email Notifications
- Workflow and recipe sharing with other users and within company
- Enhanced predictive maintenance by using the output of predictive maintenance tools
- Fully integrated with Data Spine, including automatic topic management using Pub/Sub Security Service.
- Upcoming integration with Secure Data Storage for historical data

#### **1.3.2.13 System Security Modeler (SSM)**

The system security model is a risk assessment tool that is provided as a service to the EFPF platform.

The SSM automates much of a cyber-security risk assessment. As well as looking for cyber threats it will also check for compliance (e.g., GDPR). It follows the process of ISO 27005 and thereby supports ISO 27001 compliance.



Figure 7. System Security Model Tool Functionalities

#### 1.3.2.14 Industreweb Global

Industreweb Global (IW Global) is a web framework that provides data visualisation, storage, workflow co-ordination, as well as Administration and Security Management tools for the Industreweb Ecosystem. Deployed either on the edge device or in the cloud the framework allows Collect Factory Connectors to be set up, and then deployed with a single click.

The functionalities of the Industreweb Global are:

- Administration of Industreweb Collect Nodes
- Definition of protocol connectors, and data tags to allow real time data access by the Factory Connector
- Editing of Collect programs to orchestrate data flows at the edge
- Pull deployment over REST to enable Collect Nodes to update their configuration
- Integrated Audit trail so configurations can be edited, deployed, or rolled back
- Security Administration tools to control User access based on Role permissions
- Built in Data Monitoring tools to monitor Tag based data published from Industreweb Collect
- Administration tools for Industreweb Display dashboards

#### 1.3.2.15 Industreweb Visual Resource Monitoring Tool

The Visual Detection and Alerting system makes use of an Industreweb (IW) Collect factory connector running in the business premises or manufacturing facility to monitor using a camera and to recognise objects within its field of vision. It uses an edge-based AI component to detect objects that it recognises from a pre-learnt AI model.

IW Collect then detects these events and based on a set of rules determines what Actions to perform. This could be to notify by email or SMS, to sound a siren, to light a warning lamp, push data to the EFPF cloud or display a message on a screen or dashboard.

The functionalities of the Industreweb Visual Resource Monitoring Tool are:



- Detection of objects base on pre-trained model
- Define match accuracy threshold for detection acceptance
- Define match duration to ensure that an object is detected for a solid period of time before acceptance
- Support for colour matching where distinction based on colour is required
- Publish events to the Data Spine for Business Intelligence applications
- Control factory outputs to support Error Proofing of shopfloor production applications

#### 1.3.2.16 Symphony Event Reactor

The Symphony Event Reactor is capable to create complex rule chains to process data and events coming from devices in a Building Management System. It allows the user to define scenarios based on configurable events, which are automatically triggered by monitoring the remote objects. The main functionalities of the Symphony Event Reactor are:

- Define complex rules through a Blockly-based GUI, using these rules to create complex scenarios in a Building Management System
- Define alarms to inform the user through SMS, Email and VoIP to remotely control the status of the monitoring system

#### 1.3.2.17 Symphony Data Storage

The Symphony Data Storage provides a solution to store large amounts of data through a high customizable setup in terms of additional services. The component supports AMQP/MQTT and REST interfaces for both data ingestion and historical data access. In terms of backends, it supports PostgreSQL and Elastic Search. The main functionalities of the Symphony Data Storage are:

- on-the-fly data aggregation and sub-sampling functions on incoming data
- flexible retention and storage quota enforcement policies
- primary / secondary data replication between an edge and a cloud instance

#### 1.3.2.18 Symphony Resource Catalogue

The Symphony Resource Catalogue is a software module used to store types, relations, and endpoints for a set of IoT objects in a given environment. The module contains two components:

1. **Object Catalogue:** A service containing the list of the objects (ID) and a complete ontology describing their types and relations
2. **Object Registry:** A service mapping the object IDs to one or more protocol-specific endpoints to access the object's interfaces.

The main functionalities of the Symphony Resource Catalogue are:

- REST APIs for objects creation and objects and services registration
- It contains both static information (what the object is) as well as dynamic runtime information (e.g., if another node controls an object in a high-availability setup)

- It relies on a set of standards (RDS, Apache Jena Triplestore) and it uses oneM2M + SAREF + SAREF4B + Nextworks' proprietary protocol as data model.

## 1.4 EFPF Architecture Information View

This section describes the high-level view of information flow in the EFPF ecosystem. It explains the implications of the Data Spine's Federated Interoperability approach and how it influences the dataflows. It also describes how the data is manipulated and where it is stored. Some of the data models and alignment of APIs is described as a part of use case scenarios from the internal Ecosystem Enabler dataflows, pilots, and Open Call experiments. Further details on the APIs and the data models of components can be found in Section 2.10.1 and the analysis of dataflows in the pilot and Open Call experimentation scenarios is included in Section 4.

### 1.4.1 High-level Dataflow Patterns in the EFPF Ecosystem

The EFPF ecosystem follows the Federated Interoperability Approach to ensure that the system remains scalable even when multiple new tools, services, and platforms join the ecosystem. This means that there is no common canonical data model prescribed at the ecosystem-level. The tools, services, and platforms can choose any standard or proprietary/custom data models that suit their needs and evolve independently. The Data Spine provides the necessary interoperability support for creating cross-platform applications. For realising their use cases, the System Integrators can make use of the integration flows in the Data Spine to establish interoperability and communication among the tools and services involved in order to create composite applications. While in other cases, the Data Spine can be used only for providing security or establishing "security interoperability" and the data transfer between the services is established in a peer-to-peer manner. The Ecosystem Enablers provide the generic integration, interoperability, and functional infrastructure. Therefore, the data models and the dataflows depend upon the use case scenario to be realised and the smart factory tools and services involved.

#### Basic High-level Dataflow Pattern

The services in the EFPF ecosystem make use of one or more of the functionalities provided by the Data Spine to enable communication with each other. In some cases, the Data Spine is used only security and SSO purposes (getting an access token, token introspection, etc.), while in other cases it is used for establishing interoperability at protocol and/or data model levels, message brokering and/or service discovery, as illustrated in Figure 8. Thus, the synchronous request-response as well as the asynchronous Pub/Sub dataflow patterns are supported.

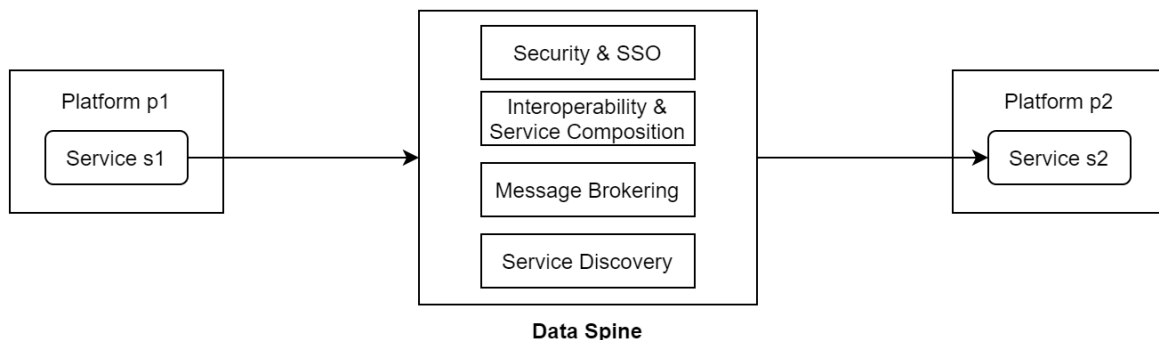


Figure 8. High-level Dataflow in the EFPF Ecosystem

### Unified Index Pattern

In some cases, the Data Spine is used to collect data from the connected platforms and then storing it in an ecosystem-level unified index, as illustrated in Figure 9. The data from the unified index can be made available to other services through an API. The Federated Search service in the EFPF ecosystem makes use of this pattern, as explained in the next section.

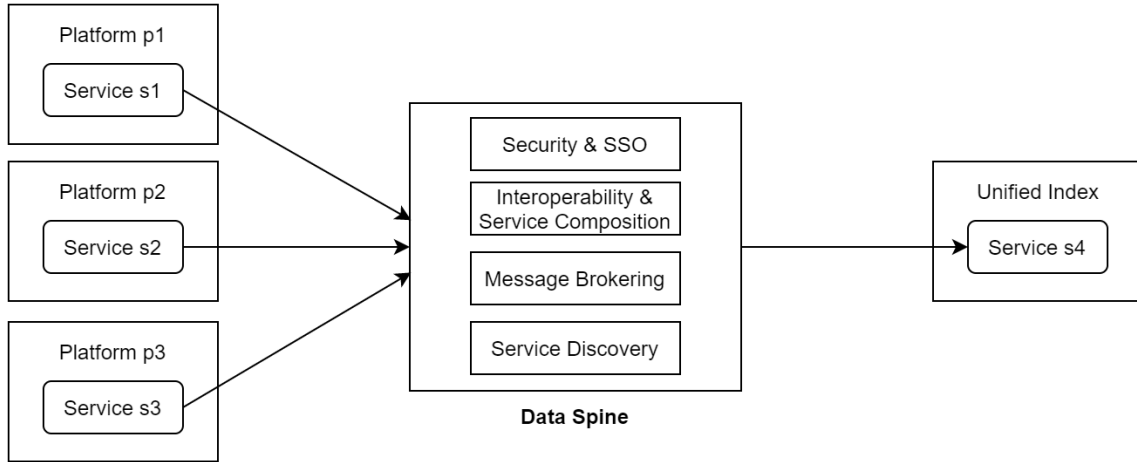


Figure 9. Unified Index Pattern

### Reusable “Interoperability Proxy” API Pattern

In some cases, multiple services expect data adhering to a certain standard data model which is not followed by the data providing service. For example, an IoT Gateway makes sensor measurements available that follow a proprietary data model, but multiple consumers expect the data that follows the OGC SensorThings data model. In such cases, the Data Spine can be used to transform the data and expose an “interoperability proxy” API endpoint that can be consumed by multiple consumers, as illustrated in Figure 10. The Production Optimisation Predictive Maintenance pilot scenario (Section 1.4.4) follows this pattern.

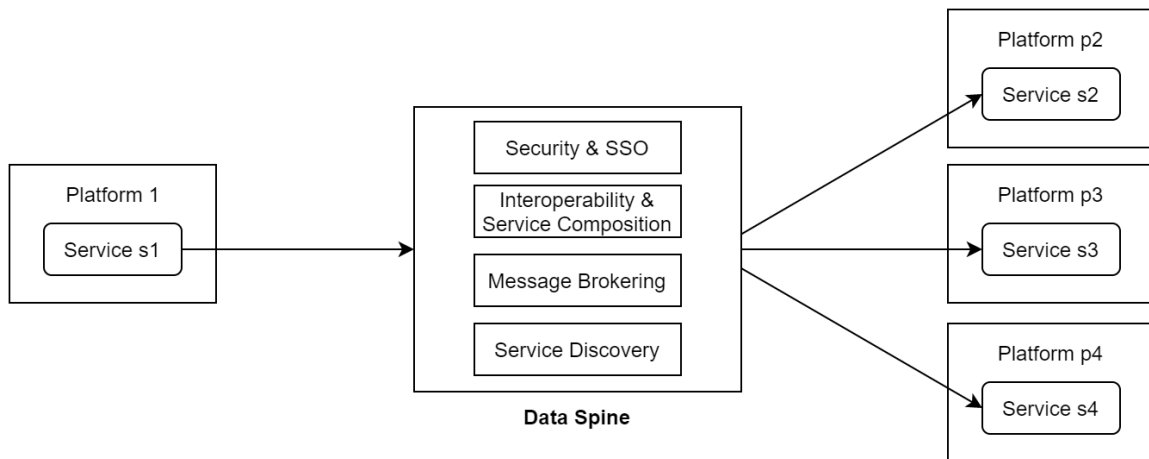


Figure 10. Reusable “Interoperability Proxy” API Pattern

### Data Storage

The user data in the EFPF ecosystem is stored in the EFPF Security Portal (EFS) as well as in the Identity Providers of the connected platforms. The Service Registry stores the technical metadata of services such as their API specifications, while the metadata of

resources (e.g., tools, Factory Connectors, integration flows) that perform publish or subscribe operations using the Data Spine Message Bus, is stored in the Pub/Sub Security Service's database. The federated/unified metadata from the connected platforms that consists of platforms' participants (suppliers/service provider companies) & their value-units (products/services) is stored in the Federated Search service's index. The other Ecosystem Enablers and Smart Factory Tools and Services store data that is necessary for their operations in their local databases. The sensor measurements data collected from the factory shopfloors by Factory Connectors or IoT Gateways is typically pushed to the Data Spine Message Bus, (optionally) transformed through an integration flow and then stored in the Secure Data Storage Solution (SDSS) from the EFPF platform, as illustrated in Figure 11. This pattern is followed by the Workplace Environment Monitoring pilot as described in Section 1.4.3.

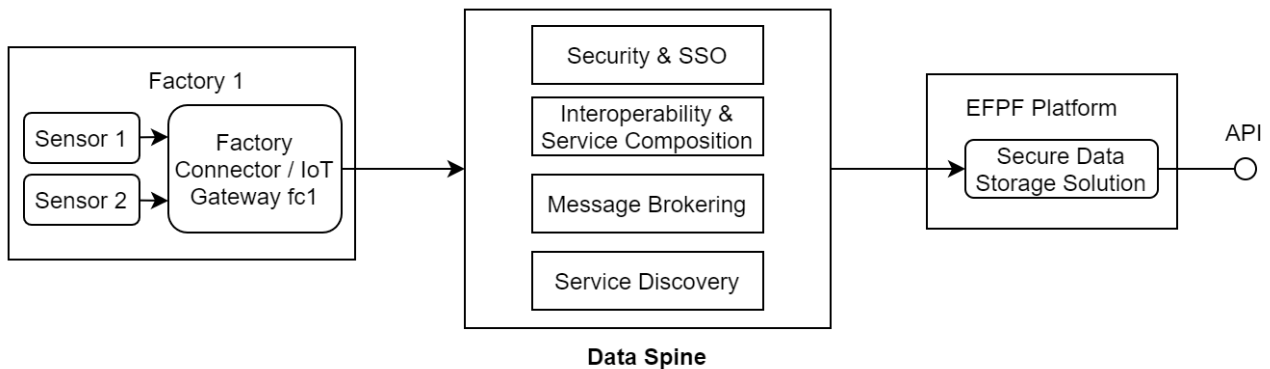


Figure 11. A Typical Sensor Data Collection and Storage Dataflow

### 1.4.2 Federated Search Indexing Dataflows

The goal of Federated Search and Matchmaking services in the EFPF ecosystem is to facilitate EFPF users to find the best suited suppliers and enable them to transact with them efficiently and effectively. The Federated Search functionality enables search for products and services across the connected platforms, by using custom, user-defined search filters. The Federated Search service's integration flows in the Data Spine contain data indexing/re-indexing workflows from the connected platforms. Different connected platforms have different data sources and data models. These heterogeneous data models need to be transformed and indexed into the common data model that is used in EFPF federated search index which is an Apache Solr based data model. There are two types of data the Federated Search service typically retrieves from a connected platform: company data and products data. These two types of data are indexed into two Solr collections in EFPF federated search index (party, item). In summary, the Federated Search service's integration flows contains the extract-transform-load (ETL) workflows for each individual base platform data to the EFPF federated search index as illustrated in Figure 12.

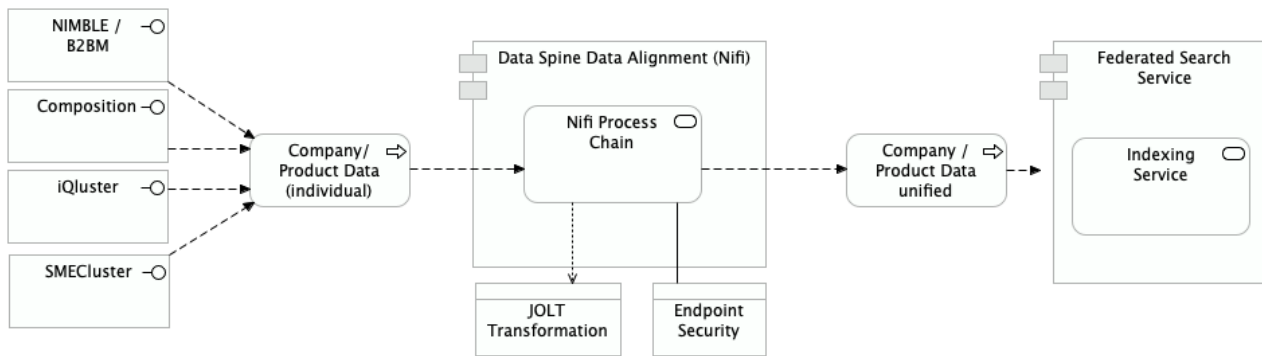


Figure 12. Federated Search Indexing Dataflows

### 1.4.3 Aerospace Pilot: Workplace Environment Monitoring

In the aerospace sector, large OEMs such as Airbus and Boeing set detailed product specifications for suppliers. In some cases, certain production steps are only permitted under very specific and monitored environmental conditions. For example, aerospace paints may only be processed within a specific temperature range. The overall goal of this pilot scenario is to ensure that specific parameters in specific production machines and production environment are analysed in real-time to provide effective decision support.

The pilot scenario involves two dataflows:

- Dataflow 1: Sensing:** The first dataflow, illustrated in Figure 13, involves sensing and data collection and visualisation operations using the federated EFPF smart factory tools and services. The TSMatch Factory Connector collects the shopfloor from the sensors installed in the factory premises and publishes it to the Data Spine Message Bus to make it available to the other EFPF tools. The data is transformed using an integration flow in the IFE and made available to the Symphony platform's GUI integrated with the EFPF Portal. The GUI shows a real-time visualisation of the collected data to the users.
- Dataflow 2: Actuation:** The second dataflow, illustrated Figure 14, involves actuation of the Alarm System. The data collected as a part of the first dataflow is made available by the Symphony Event Reactor component that compares the reading with a predefined threshold. If a certain measurement exceeds the threshold value, the Alarm System is activated to notify the factory managers for taking an appropriate action.

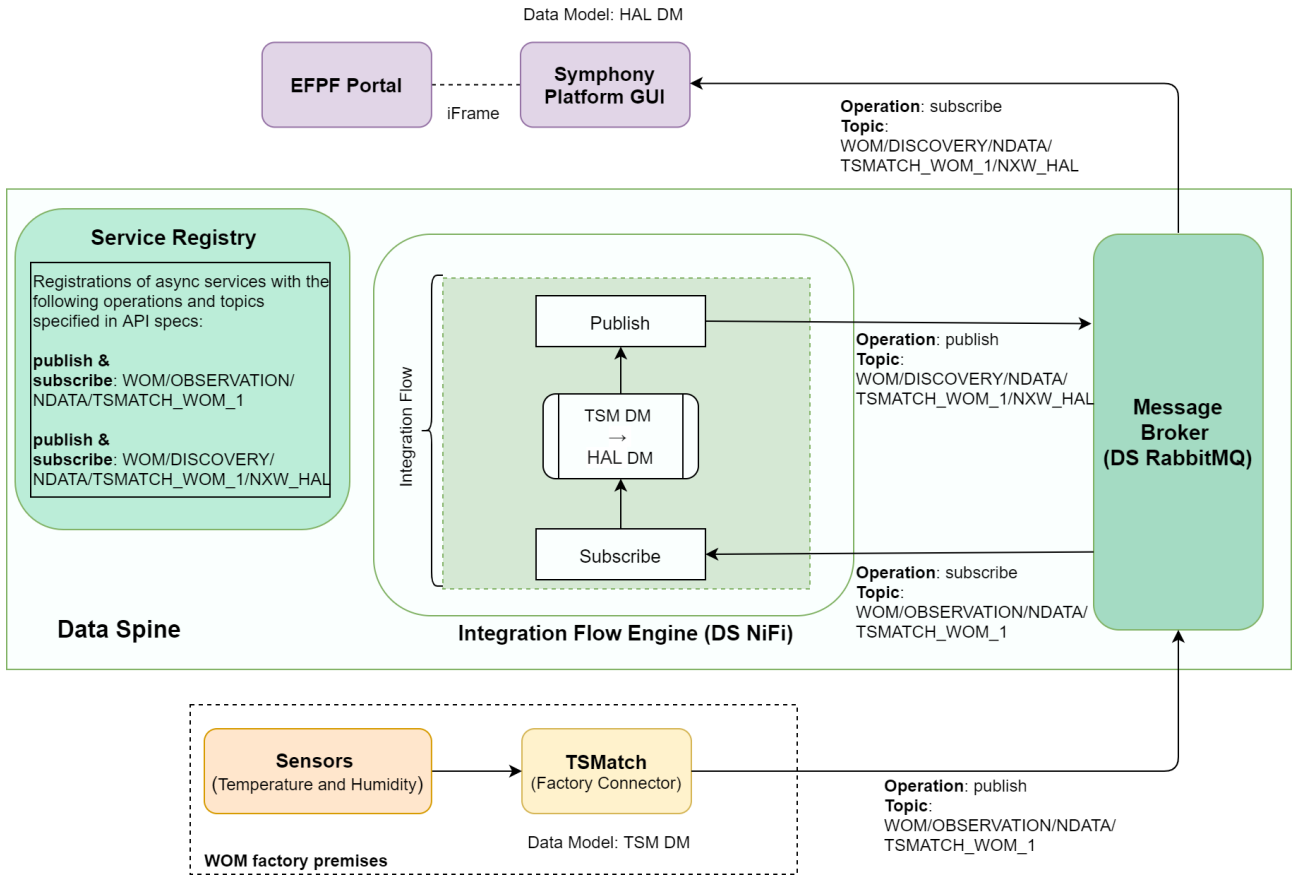


Figure 13. Workplace Environment Monitoring Dataflow 1: Sensing

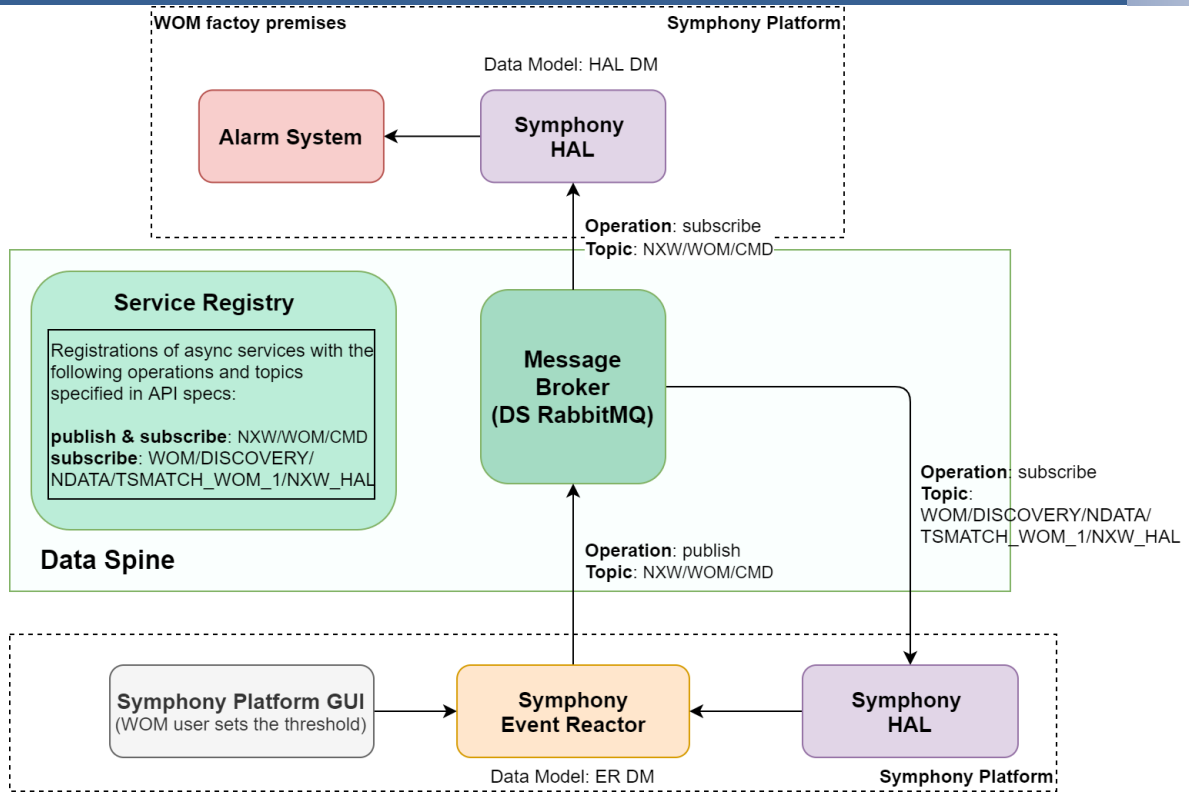


Figure 14. Workplace Environment Monitoring Dataflow 2: Actuation

#### 1.4.4 Furniture Pilot: Analytics & Predictive Maintenance

In this pilot project the Lagrama company wanted to use multiple predictive maintenance and analytics solutions with their data collected from the shop floor. In order to do so the company resorted to some solutions available on the EFPF platform and its main component, the Data Spine.

Since these solutions used different data models the main challenge has been to integrate those with the data coming from the shopfloor which used a data model of its own. All of the tools and services in this pilot project used the MQTT protocol so there was no protocol bridge to cover.

As shown in Figure 15, the data reaches the Data Spine encoded with a property data model created by C2K. the first operation performed on the data is to scale the values of the sensors from the raw readings performed on the machine to a set of standard units of measurement. Once this operation has been performed, using a JOLT processor on NiFi, the data is published again to a different topic on the Message Bus.

At this point in the pipeline, three tools can already consume this data. The Risk Analysis Tool, the Visual Analytics Solution, and the Anomaly Detection Solution.

Since the deep learning tool kit uses DOGC sensor things standard data model the scaled data must be converted to this data model before it can be consumed by the tool. This is how quite complex transformation, and it happens outside the data spine on a dedicated micro service. Once this updated data is fetched from the micro services rest API then it can be published again to the message bus. The data is finally consumed by the deep learning tool kit and used for making predictive maintenance forecasts.

The forecasts made by the deep learning tool kit are done pushed back to the message bus and consumed by the risk analysis tool directly using the proprietary data model of the deep learning toolkit and are used for generating notifications about possible breakdowns.

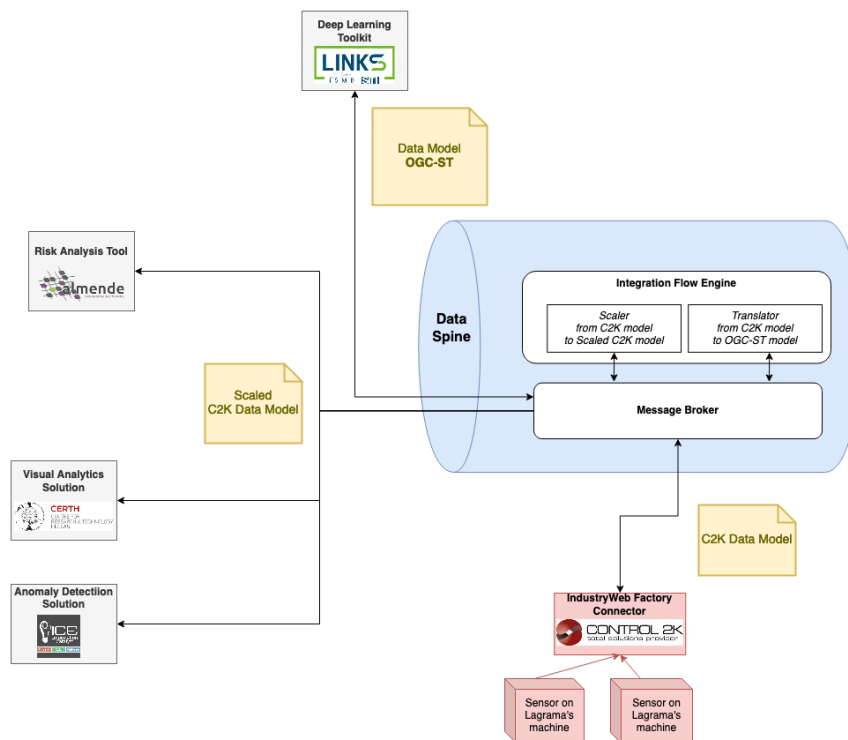


Figure 15. Furniture Pilot: Analytics & Predictive Maintenance Dataflows

### 1.4.5 Open Call Experiment: DNET Labs

The goal of this open call project is to integrate DNET's Product Passport service with the EFPF infrastructure and with a predictive maintenance service already available. The dataflows are illustrated in Figure 16. In order to demonstrate this integration between this service and the EFPF platform the readings from different sensors coming from the Metalac source have been used.

This data comes with a custom proprietary data model and is pushed to the EFPF Data Spine using the AMQP protocol. The first challenge to tackle has been to bridge the protocol gap between the DNET service and the predictive maintenance solution since this one uses the MQTT protocol. To solve this issue a NiFi flow has been developed which subscribes from that queue and publishes to a similarly named one.

The second challenge was that the DNET services uses a custom data model while the Deep Learning Toolkit that uses a standard OGC Sensor Things based data model. To solve the second issue the data was pushed from the Data Spine to a custom micro service developed for this purpose. The rest API provided by the service allows to perform the translation process in real time.

Now the transformed data is ready to be sent to the Deep Learning Toolkit using the MQTT protocol. Once the Deep Learning Toolkit has made predictions using this data, the predictions themselves are published to the message bus using the MQTT protocol.

Before the end service can consume these predictions, these shall be collected from the MQTT topic on which they are published and have to be pushed to another AMQP queue



since that service uses that protocol to communicate. The predictions are finally consumed by the DNET product passport service and presented to the end users.

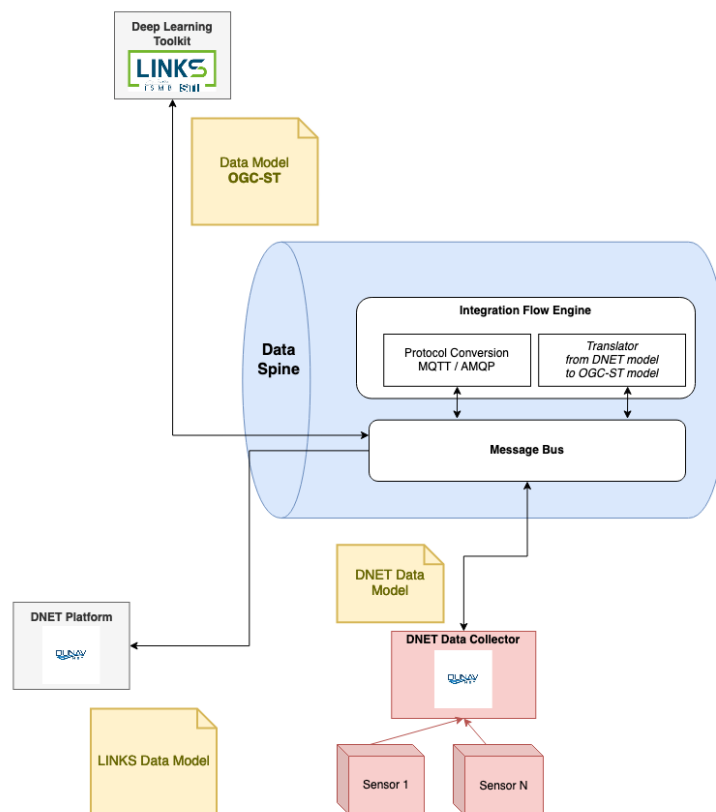


Figure 16. Open Call Experiment: DNET Labs Dataflows

## 1.5 EFPF Architecture Development and Deployment View

The design goal of the development and deployment architecture for the integrated EFPF platform (i.e., Ecosystem Enablers) has been to enable EFPF, and later EFF, to perform continuous integration of the Ecosystem Enablers, manage deployment on partner infrastructure and release incremental versions of the Ecosystem Enablers for functional testing. The owners of 3<sup>rd</sup> party tools and services, base platforms and EFPF Platform tools and services will manage the respective development and deployment of their resources. The major architectural concerns have been to enable a modular and extensible infrastructure for the Ecosystem Enablers where new modules and components can be added, development can be distributed, deployment can be made on premise, cloud or distributed, and quality attributes like scalability, maintainability and availability are ensured. The heterogeneous nature of the development organization with many organizations in different locations, using different tools, had to be considered when designing the architecture.

The Data Spine has served as the main use case and architectural proof-of-concept when developing the development and deployment architecture. Stable operation, repeatable deployment and co-hosting of the Data Spine components has been a primary concern. The other Ecosystem Enablers have been adapted to the defined architecture.

The development process and codeline organization for the Ecosystem Enablers use the GitLab tool. It provides code repository, image repository for docker components, agile

development issue management tool, and runs the CI/CD pipeline. Ecosystem Enabler code is managed by EFPF. There are a few exceptions, but the code for business-critical components is in the EFPF repository or available as public open-source code in the case of external components, e.g., NiFi. The EFPF Platform components primarily use in-house development tools and processes.

Deployment and testing are further covered in Section 2.6 of this document.

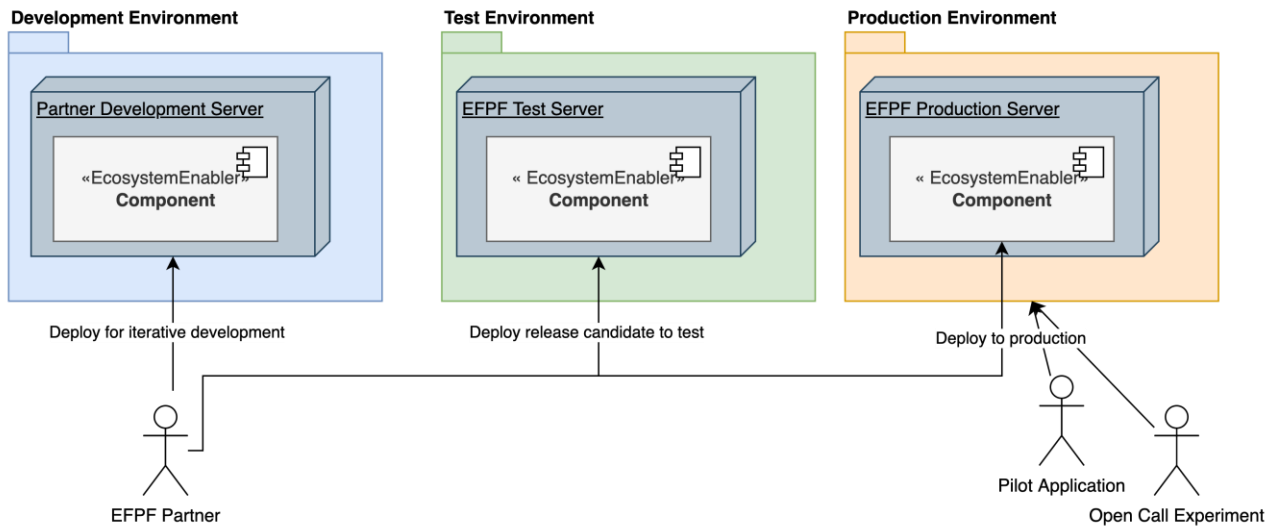


Figure 17. Runtime environment

Development of the EFPF Ecosystem and integration of base platforms started at the same time as the specification of the development and deployment architecture. The initial versions of the Data Spine, Portal and other central components were distributed over the hosting resources of the technical partners, using the available deployment and hosting processes and resources. This configuration now comprises the Development Environment, where iterative development, experimentation and proof of technical feasibility takes place. There are no policies guaranteeing that the deployed version of any component is stable, ad hoc deployment is allowed, and the configuration of services, network, or security may change without notice. (However, it has proven to be quite stable.)

The project needed a staging runtime environment where release candidates approved for production and a stable runtime environment used for external users, Pilots and Open Call experiments. These are called the Test Environment and Production Environment, respectively. There is only one production environment, as a limited number of runtime environments require less effort to manage and support. It was decided to limit Test and Production hosting to the Ecosystem Enablers, specifically the Data Spine and Portal. This was to ensure performance efficiency and co-existence by having sufficient resources for the most business-critical components.

The decision taken was to use on-premises hosting at C2K rather than relying on cloud resources, e.g., AWS or Azure Cloud. This was motivated by having predictable costs and a guaranteed continuous environment when transitioning to EFF.

The choice of a common container technology provides maintainability and portability for the Ecosystem Enablers. The project selected the widely used Docker container technology and after evaluating Kubernetes, decided on Docker Swarm for container management.

The automated and configurable deployment pipeline together with the container technology enables EFPF and EFF to switch to cloud hosting or deploy multiple instances of the platform if this is desired. E.g., the on-premises hosting can be complemented with instances hosted in the cloud for resource demanding customers.

It was a highly prioritized task to move all Data Spine components as well as the Portal and Marketplace components to the CI/CD Pipeline and deploy these together on the Test and Production Environments.

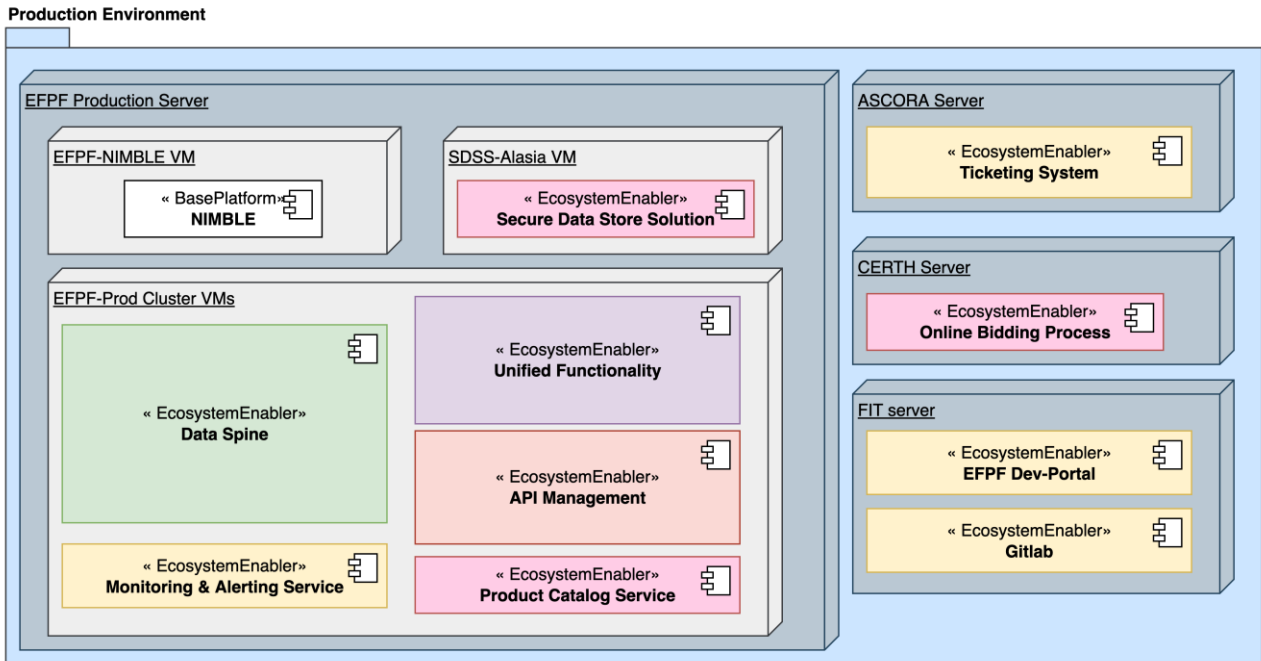


Figure 18. The Production Environment (simplified)

The Production Environment hosts the stable versions of the Ecosystem Enablers. This primarily means the Data Spine and Portal. However, other Ecosystem Enablers can be hosted there if they implement the CI/CD pipeline and container technology. If they have high cohesion with the Data Spine and Portal and resource consumption is estimated to be low and stable, they may be hosted together with the Portal and Data Spine on the same nodes. Otherwise, a separate virtual machine is set up on the server. Performance efficiency and reliability for the Data Spine and Portal is the deciding factor when allowing other components to be deployed in the Production Environment.

The diagrams above (Figure 17 and Figure 18) have been simplified and only Ecosystem Enablers are shown. The infrastructure and container management tools – Docker Swarm, Portainer, Nginx - located on the test and production virtual machines with Data Spine have been omitted from the diagrams. Detailed information on the development and deployment views is provided in deliverable “D6.2: Integration and Deployment - Final Report”.

## 1.6 Integration Methodologies & Documentation Structure

The EFPF ecosystem consists of a large number of tools, services, and platforms. It can be difficult for new users to integrate their tools and/or to create composite applications using the separate documentations of the available tools, services, and the infrastructure, especially if they are not homogeneous. To ensure consistency among the documentations

of the Smart Factory Tools and Services, standard templates illustrated in Figure 19 was followed.

1. **Index Page (Overview)**
  - Introduction, Features, USP, etc.
  - High-level architecture
2. **Service1 Quickstart Guide**
  - Getting started
  - Hello World
3. **Service1 Admin Guide**
  - How to install, deploy, configure, etc.
  - Where to find Docker images or other binaries
4. **Service1 User Guide**
  - How to use/consume
  - Functional information
  - GUI related documentation
  - API documentation
5. **Service1 Developer Guide**
  - How to maintain, enhance, extend, etc.
  - Detailed architecture
  - Link to the source code repo
6. **Service1 Miscellaneous**
  - Any other documentation that is not covered in the sections above

Figure 19. Documentation Template for EFPF Ecosystem Components

The documentation for the EFPF components that follows this template was published onto the EFPF Dev-Portal, which is integrated with the EFPF Portal as shown in Figure 20.

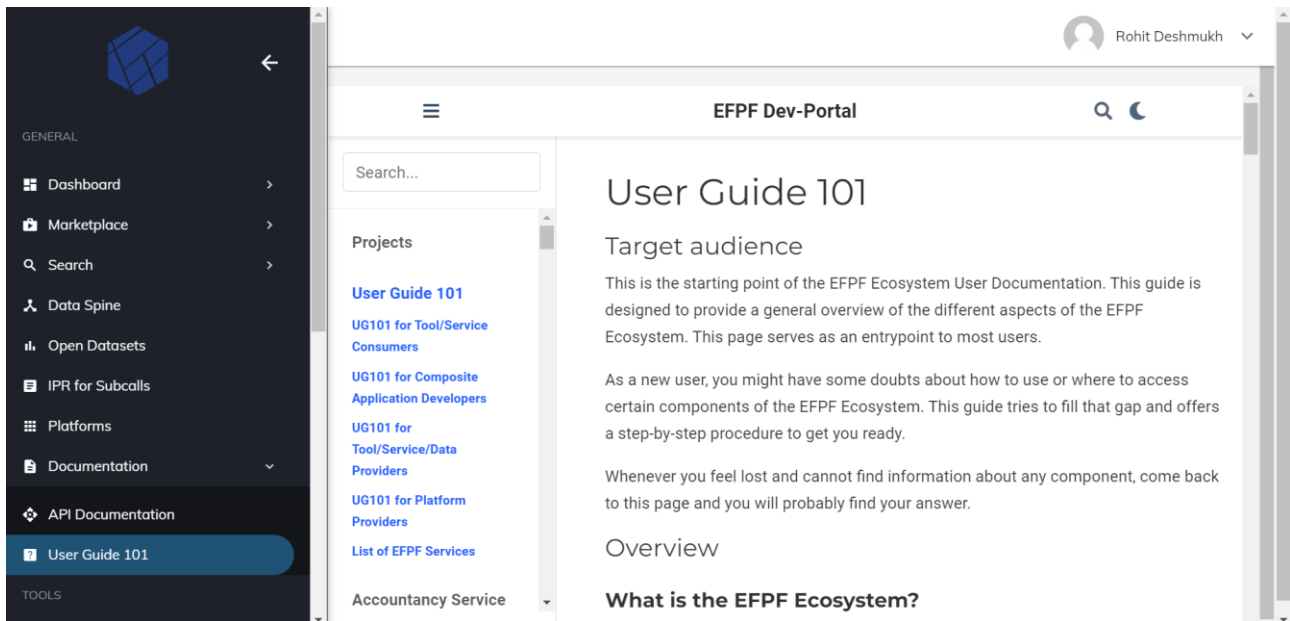


Figure 20. EFPF Dev-Portal

In addition, in order to avoid the users “getting lost” into the documentation and for providing them with an orientation, a new User Guide (UG) 101 that acts as the starting point of the EFPF ecosystem user documentation was created. The UG101 acts as the index, describes the EFPF ecosystem architecture in brief, introduces the core components and the deployment environments and provides links to the detailed documentation of the components. In order to provide the current orientation for the users depending upon how they want to interact with the ecosystem, the UG101 directs them to one of four further

UG101s, based on their user role, as shown in Figure 21. The integration/interaction methodologies contained in these four UG101s are described in the subsequent sections.

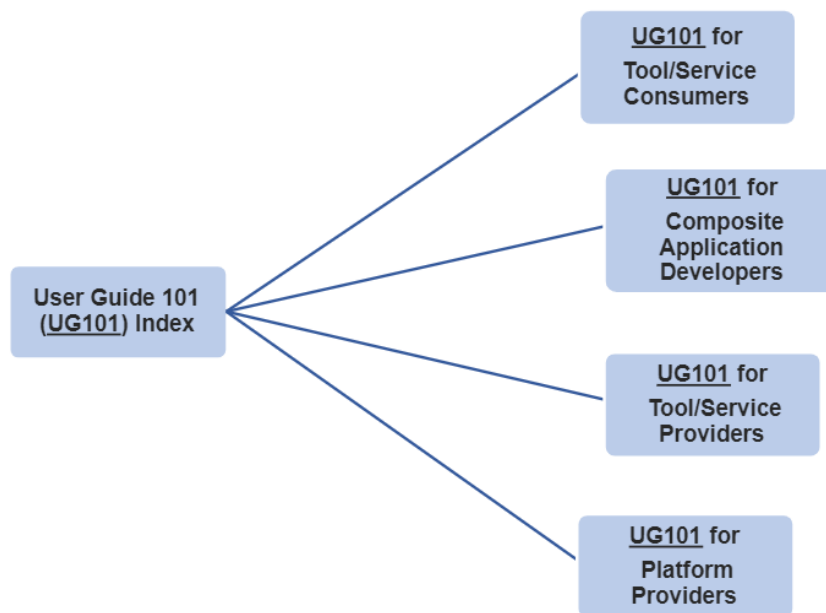


Figure 21. User Guide 101 Documentation Structure

### 1.6.1 Integration Methodology for Platform Providers

This section lists the integration steps for the ‘Platform Providers’, i.e., the users who want to connect/integrate/federate their digital platform with the EFPF ecosystem. In the context of the EFPF ecosystem, the definition of a digital platform is “A platform that provides offerings such as digital tools, services and data, and secures access to them using its own Identity and Access Management service”.

#### Prerequisites and notes:

- You must have an EFPF user account
- Refer to the documentation of the individual components for detailed integration steps

#### Steps:

##### 1. Data Spine: Enabling Single Sign-On (SSO) Functionality

- Integrating/connecting/federating a platform (which has its own private Identity Provider) with the EFPF Ecosystem means making its tools and services accessible with a single set of EFPF credentials, i.e., enabling the users of EFPF ecosystem to access the tools and services of that platform with their EFPF (EFS) user account.
- The EFS acts as the central Identity Provider for the EFPF ecosystem. It federates the identity providers of all platforms in the EFPF ecosystem in order to enable SSO functionality.
- Refer to the EFS SSO documentation to federate your platform’s Identity Provider with the EFS [EFS22].
- Enabling SSO for the platform mainly enables two functionalities:

- Login with EFPF: These actions would enable SSO and add ‘Login with EFPF’ option to the login page of the integrated platform.
- Access to Service APIs: These actions also enable access to the APIs of the tools/services in the platform with (authorized) EFPF user accounts.

## 2. Portal integration

- The EFPF Portal acts as the single point of entry for the EFPF ecosystem.
- An entry with platform’s name, logo, short description, etc., can be added to the ‘Platforms’ page of the EFPF Portal.
- The ‘Value Proposition’ pages can be updated.

## 3. Marketplace integration

- The EFPF Marketplace retrieves the list of products and services from the marketplace services of the connected platforms and displays them coherently.
- After SSO is enabled for the platform, its marketplace service can be integrated with the EFPF ecosystem’s Marketplace.

## 4. Accountancy Service integration

- The Accountancy Service aims to track and trace a user’s journey across the EFPF ecosystem and collect data about the transactions they make on different marketplaces to enable a cashback mechanism
- The Logstash endpoint of the Accountancy Service should be available through the Data Spine Service Registry

## 5. Matchmaking/Federated Search integration

- The goal of Matchmaking in the EFPF ecosystem is to facilitate EFPF users to find the best suited suppliers and enable them to transact with them efficiently and effectively.
- The Federated Search functionality enables search for products and services across the connected platforms, by using custom, user-defined search filters.
- The metadata of the newly connected platform’s participants (suppliers/service providers) & their value-units (products/services) can be added to the Matchmaking service’s index.

## 6. Data Spine: Registration of Services / APIs

- The technical metadata of the Services and their APIs (OpenAPI (Swagger) or AsyncAPI specs) can then be registered to the Data Spine Service Registry, so that the potential service consumers can discover these services and retrieve their technical metadata such as the API endpoints, the API specs, etc., which are needed for consuming them directly or through the integration flows.

## 7. Use of other Ecosystem Enablers

- The use of other Ecosystem Enablers such as the depends upon use cases involving service-level integration/communication. E.g., if your use case involves Pub/Sub communication, you can use the Data Spine Message Bus, if you need an SCM and CI/CD pipelines, you can use the DevOps infrastructure, if you need a coherent listing of the ecosystem offerings, you can use the Federated Search Index APIs, etc.

## 1.6.2 Integration Methodology for Tool/Service Providers

This section lists the integration steps for the Tool/Service/Data Providers, i.e., the users who want to integrate/provide their tool/service/data, that does not have an associated Identity Provider (IdP), as a part of the EFPF ecosystem. The Tool/Service/Data Providers can be of different types as explained in Section 1.1.2 and the steps for each type are listed below.

### Prerequisites and notes:

- You must have an EFPF user account
- Refer to the documentation of the individual components for detailed integration steps
- The central, core services called 'Ecosystem Enablers' are hosted centrally on the EFPF servers. The rest of the smart factory tools, services and platforms are self-hosted by the respective providers on their servers, and they connect to the EFPF ecosystem through the Ecosystem Enablers.

### Steps:

#### SaaS Providers:

##### 1. Deployment

- Deploy your tool/service on the servers managed by you
- The EFPF DevOps infrastructure can be used for development, deployment, and operational management

##### 2. Single sign-on (SSO) - Authentication and authorization for request-response APIs (if applicable)

- Your tool's interfaces (GUI/APIs) must be accessible using EFPF user accounts
- If your tool offers a GUI, add functionality to it to redirect to the EFS login page (if the user is not logged in already) for authentication
- This will also require registration of a new client for your tool in the EFS Keycloak. To do that, send an email to the EFPF Support Team with the details of the client to be created.
- Configure your tool's GUI to use the OAuth2.0 Authorization Code Grant Flow to request an Access Token from the EFS Keycloak
- Use this access/bearer token to access the tool's API
- If your tool offers only an API and no GUI, you can assume that the user who wants to access the API is already in possession of an access token
- Add functionality to your tool to perform authentication for incoming API access requests. This can be done in various ways:
  - using a proxy route/endpoint in a local instance of Apache APISIX,
  - using a Policy Enforcement Point (PEP) embedded into the tool,
  - using an external library integrated into the tool that takes care of authentication (e.g., Keycloak client adapters, LinkSmart go-sec, Quarkus library),
  - using a locally deployed proxy microservice that performs authentication (e.g., oauth2-proxy), etc.

- EFS Keycloak's public key can be retrieved from <https://<Prod environment EFS Keycloak URL>/auth/realms/<realm name>> (realm name would be either 'efpf' or 'master').
  - Add functionality to your tool to perform authorization for incoming API access requests
3. Authentication and authorization for Pub/Sub APIs (if applicable)
    - Use the Pub/Sub Security Service dashboard from the EFPF Portal to get credentials for the Data Spine Message Bus (DS RabbitMQ) and to get permissions to publish/subscribe to topics/queues in DS RabbitMQ
    - Configure your tool to publish/subscribe to DS RabbitMQ
    - The users who want to subscribe to your topics can make use of the Pub/Sub Security Service dashboard to ask for access, and you can see and approve/reject the access requests using the dashboard
  4. Service/API registration
    - Register your tool/service's APIs (including Pub/Sub APIs) to the Service Registry
  5. Data enrichment (if needed)
    - If you want to offer enriched data or data conforming to some other data model (e.g., making OGC SensorThings compliant data available over another API in addition to the proprietary data model served by your tool/service's API) to the potential service/data consumers, you can use the Data Spine Integration Flow Engine (DS NiFi)

#### **SaaS Providers:**

1. Tool Provision
  - Make the tool artifacts such as the binaries or source code downloadable to authenticated and authorized users, or publicly (e.g., if free and/or open source)
  - Publish the documentation for the tool such as admin, developer, and user guides, including the API specifications
2. Advertise
  - Advertise your SaaS tool/service on the EFPF Portal/Marketplace to enable sale/discovery

#### **Data Providers/Publishers:**

1. Search for a tool (if needed)
  - Search for a tool (e.g., a Factory Connector / an IoT Gateway) on the EFPF Portal/Marketplace
2. Get the tool (if needed)
  - Purchase/download the tool using the link from the EFPF Portal/Marketplace
3. Deployment



- Deploy your tool on the servers managed by you (e.g., on factory premises)
  - You can follow the admin guide of the tool from the EFPF Dev-Portal for deployment, initial setup, and administration
4. Collect data
    - Connect your tool to data sources in order to collect data. E.g., connect to sensors to collect shop floor data.
  5. Authentication and authorization for request-response APIs (if applicable)
    - Same as step (2) for SaaS Providers above
  6. Authentication and authorization for Pub/Sub APIs (if applicable)
    - Same as step (3) for SaaS Providers above
  7. Service/API registration
    - Same as step (4) for SaaS Providers above
  8. Data enrichment (if needed)
    - Same as step (5) for SaaS Providers above

### 1.6.3 Integration Methodology for Composite Application Developers

This section lists the integration/interaction steps for the Composite Application Developers, that is, the users who want to create applications that use the existing tools/services from the EFPF Ecosystem using the Data Spine. If a tool/service consumer expects data adhering to a different data model than what the tool/service provides, she/he can make use of Data Spine (DS) NiFi to create an integration flow (i.e., a workflow/dataflow) that uses the built-in data model transformation tools (e.g., Jolt, XSLT, ExecuteScript, etc. Details. DS NiFi provides an intuitive drag-and-drop GUI to create integration flows easily. An overall flowchart of the steps is illustrated in Figure 22.

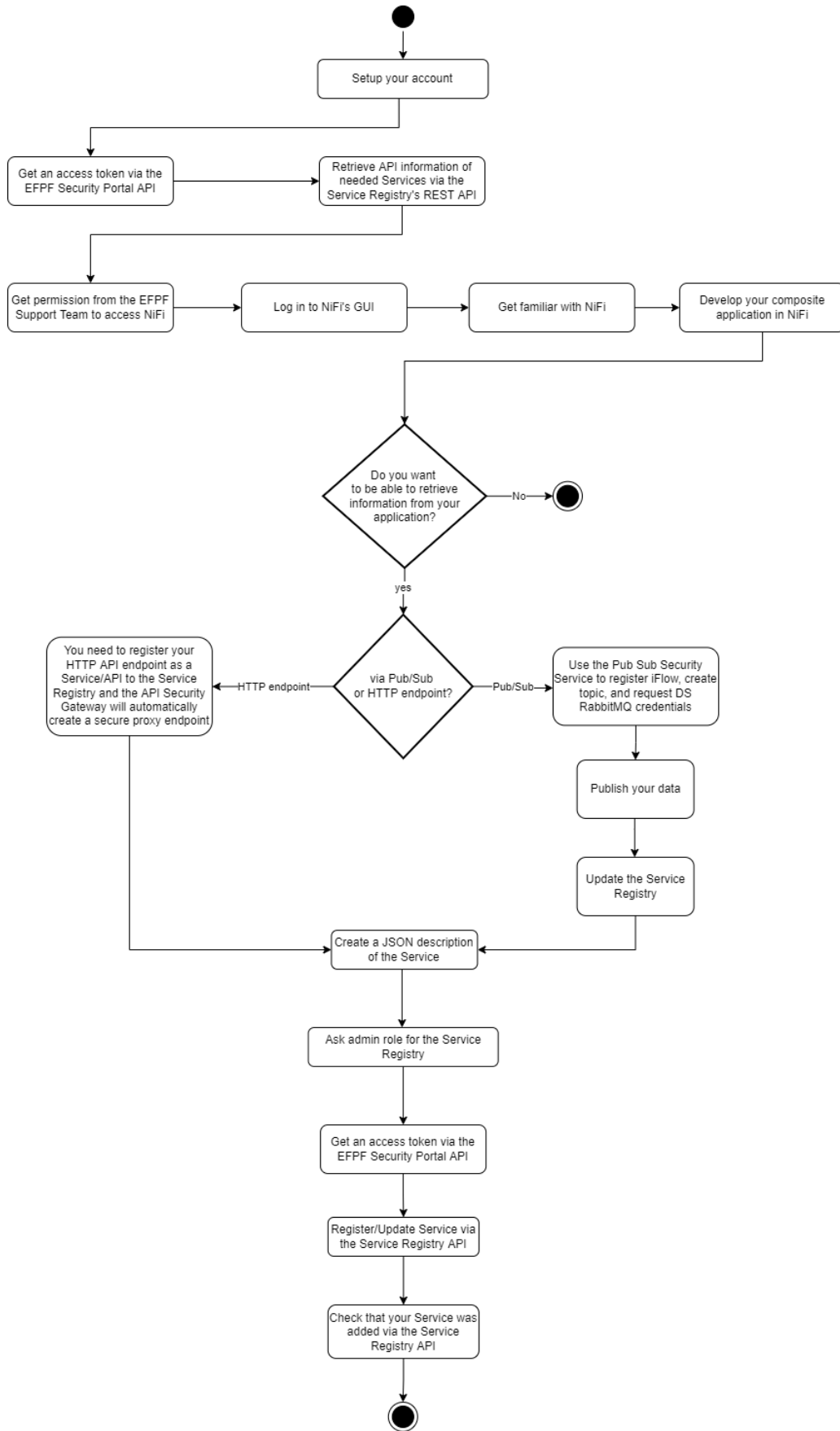


Figure 22. Flowchart of Integration Steps for Composite Application Developers

### Prerequisites and notes:

- You must have an EFPF user account
- Refer to the documentation of the individual components for detailed integration steps

### Steps:

1. Set up your EFPF account
2. Get an access token via the EFPF Security Portal (EFS) API
3. Retrieve the needed information from the Service Registry: Use the Service Registry REST API to search for the existing services you want to use and retrieve the needed data.
4. Get permissions to access NiFi: Get the necessary permissions to access NiFi. To create Integration Flows in DS NiFi, a collaboration space called 'Process Group (PG)' would be given to you. You would be given admin privileges for your PG enabling you to not only create and execute integration flows, but also give other users access to your PG in order to collaborate.
5. Log in to NiFi: After getting the necessary access permissions, log in to NiFi's GUI. To develop your application that uses the existing services in the EFPF Ecosystem, you will need to create integration flows using the drag-and-drop GUI of the Integration Flow Engine (NiFi).
6. Get familiar with NiFi tool: The Integration Flow Engine of the DS is realised using the Apache NiFi dataflow management tool. To get familiar with Apache NiFi, take a look at the provided examples [EX122, EX222].
7. Develop your composite application using the existing services: This is the core part of the work to be done. This is where you have to create one or more integration flows to implement the application logic in NiFi. NiFi provides built-in processors to accomplish various tasks with ease, e.g., for data transformation to achieve interoperability at Data Model level, it provides processors such as Jolt, TransformXml (XSLT), ExecuteScript, etc. To easily create integration flows, you can refer to the example integration flows available in the same instance of NiFi you are using. Inside the 'p1' Process Group in NiFi you will find another Process Group called 'Examples'. This Process Group contains some simple, most commonly used integration flows.

In case you need to make some information from your application accessible from outside of NiFi (i.e., you take on the role of a Service Provider), you need to follow the next steps, depending on how you plan on doing this. You have two options:

- Using Pub/Sub: You can make information from your integration flow in NiFi accessible from the outside via the DS Message Bus (RabbitMQ) using MQTT/AMQP.
  1. Use the Pub/Sub Security Service dashboard: Use the Pub/Sub Security Service Dashboard from the EFPF Portal to get credentials for the Data Spine Message Bus (DS RabbitMQ) and to get permissions to publish/subscribe to topics/queues in DS RabbitMQ. Follow the Pub Sub Security Service Quickstart Guide for steps on how to use the Pub Sub Security Service.
  2. Publish your data: In your integration flow in NiFi, publish your data to RabbitMQ with the configuration details and credentials provided by the Pub Sub Security Service.

3. Update the Service Registry: It is recommended to register the new Pub/Sub API to the Service Registry. If the data being published is to be consumed by other companies, the registration is mandatory. Otherwise, if the data being published is meant to be consumed by the same users who published it or by users in the same company, it is not mandatory but still recommended to register the API containing information such as the topic name, payload syntax, etc., to the SR as it would be useful in the long run.
- Using your HTTP API
    1. You can configure your integration flow in NiFi to expose an HTTP API endpoint that can be accessed like any other service/API in the EFPF Ecosystem.
    2. Register your Endpoint as a Service.

## 2 Design and Realisation of Interoperable Data Spine

This section highlights the vision and objectives for the creation of the Data Spine, explores the challenges and formulates the requirements that result in the definition of the conceptual components of the Data Spine. The architectures of these components, the technologies selected to realise them, are described followed by their DevOps and testing mechanisms. Then, the service integration and dataflow through the Data Spine are explained and finally, the usage of the Data Spine in the pilots as well as the Open Call experimentation scenarios is briefly highlighted.

### 2.1 Vision and Objectives

The recently increased digitalisation in the manufacturing domain opens new opportunities for companies to collaborate, find new suppliers, establish value chains and ad hoc collaborative networks, optimise their supply chains, streamline production processes, and reuse resources such as tools, services, and data in their platforms in order to create innovative B2B applications. However, today's digital manufacturing platforms are largely heterogeneous with their resources closed behind their Identity Providers. Because of the interoperability gaps among the platforms, it becomes very challenging to achieve the objectives of hyperconnected factories, lot-size-one manufacturing and Industry 4.0. The full potential of the platform resources remains untapped. The already existing and established tools, services and solutions cannot be used across platforms, contexts, or domains. The individual platforms can provide only a limited set of functionalities and in order to avail extended functionalities, the users must join multiple platforms and deal with the interoperability gaps themselves. As the reusability is limited, the cost of joining the platforms increases, barring the entry of SMEs which are considered strong drivers of innovation and are crucial for realising the objectives of agile manufacturing. Therefore, creation of an ecosystem of such heterogeneous digital manufacturing platforms that enables interoperability, and an easy creation of cross-platform is needed.

The EFPF ecosystem consists of distributed, heterogeneous digital platforms, tools, and components provided and hosted by independent entities. The technical features, including its interfaces, protocols, data formats, data models, and identity and access management mechanisms, etc., differ significantly from each other, thus making a direct communication between EFPF services is challenging. There could be many different ways to address this problem – one such way could be to design standardized APIs based on the identification of common standards and abstractions and ask the Service Providers and the Service Consumers to implement specific connectors/plugins, in order to align their proprietary APIs to these standard APIs and enable communication. However, such approaches are not desirable as they need significant modifications to the existing tools, services, systems, and platforms that need to communicate with each other, among other shortcomings. Hence, a novel solution enabling an interoperability and communications layer that acts as a translator/adaptor between these heterogeneous tools, services, systems, and platforms, and providing data handling, routing capabilities and API adaptation functionalities is needed. The Federated Interoperability Enabler, Data Spine, is designed to address these challenges. The Data Spine also aims to make the EFPF ecosystem scalable and extensible, so that more 3<sup>rd</sup> party tools, services, and platforms can join the ecosystem. The next section identifies the challenges and formulates the requirements that guide the design of the Data Spine.

## 2.2 Requirements

In this section, the challenges in creating an ecosystem/federation of digital platforms that must be addressed by the Data Spine are identified. From each challenge, a requirement is formulated. To fulfil each requirement, a component is defined. These components become the building blocks of the Data Spine.

The lack of cross-platform interoperability is a major roadblock in the creation of an ecosystem of digital platforms to enable an easy creation of applications using services from multiple platforms. Interoperability gaps exist between the services of heterogeneous platforms mainly at the levels of security mechanisms, communication protocols, data models, data formats, data values, interaction approaches, and interaction and communication patterns, etc., as illustrated in Figure 23.

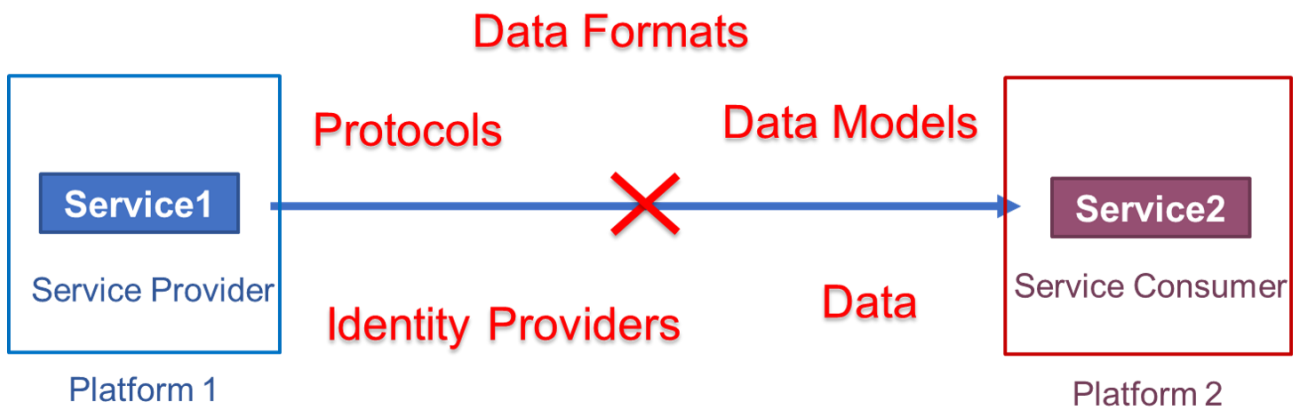


Figure 23. An Illustration of Interoperability Gaps between the Services of Digital Platforms

**Challenge 1)** Security interoperability: The services of each digital platform are behind their own, closed Identity Providers and a user needs to get user accounts for multiple platforms to access their services.

**Requirement 1:** The users should be able to seamlessly access tools and services from different platforms using a single set of credentials.

**Solution:** This challenge is addressed by using a component called 'EFPF Security Portal (EFS)' that federates the Identity Providers of the platforms in the ecosystem.

**Challenge 2)** Protocol and Data Model Interoperability and tooling support for an easy service composition: The interoperability gaps between services of heterogeneous platforms in the ecosystem prevent cross-platform service-level communication.

**Requirement 2:** It should be possible for users to create cross-platform applications using the existing services that may use different communication protocols and data models, without making any changes to them.

**Requirement 3:** The users should not be required to deploy and maintain additional components on their self-managed servers.

**Requirement 4:** The creation of cross-platform applications should be possible easily and intuitively, with minimal coding effort.

**Requirement 5:** It should be possible for users/developers from the same or different platforms/companies to collaboratively create composite applications.

**Solution:** This challenge is addressed by using a component called 'Integration Flow Engine (IFE)' that provides a low-code development environment for the creation of composite applications.

**Challenge 3)** Support for asynchronous Pub/Sub communication pattern and messaging protocols

**Requirement 6:** The ecosystem should not only support message-based communication, but also enable the composition of tools/services that use the messaging pattern.

**Solution:** This challenge is addressed by using components called 'Message Bus and 'Integration Flow Engine' that support Pub/Sub messaging pattern and protocols.

**Challenge 4)** Service/API discovery and metadata

**Requirement 7:** The users who create cross-platform applications using the existing services need a mechanism for discovering the existing services and retrieving their technical metadata such as the API specifications.

**Solution:** This challenge is addressed by using a component called 'Service Registry' that provides a mechanism for lifecycle management and discovery of service/API metadata and endpoints.

**Challenge 5)** Decoupling of application logic and policy enforcement and permanent hyperlinks

**Requirement 8:** The users should be able to easily secure the HTTP-based APIs of their integration flows and the Ecosystem Administrators should be able to secure the APIs of the Data Spine components.

**Requirement 9:** The services that do not have fixed/permanent API endpoints, should still be discoverable and accessible to the consumers.

**Solution:** This challenge can be addressed by using policy enforcement points embedded into the components or by using a component called 'API Security Gateway (ASG)' that intercepts incoming requests to component APIs and enforces access policies with the help of the Data Spine Security Portal. In addition, the ASG can also provide permanent reverse proxy endpoints for the services that do not have fixed/permanent API endpoints.

Additional requirements:

- **Federation approach:** The ecosystem should follow a federation approach, enabling "on-demand" interoperability between different tools/services, i.e., when required by a use case. No common data model or format should be imposed, so that there is no overhead on the system administrators of maintaining such a complex canonical model

and on the services to understand it and adhere to it. This is further explained in Section 2.3.

- **Agility and flexibility:** The ecosystem should allow tools/services to use neutral APIs, which are not strongly tied to any specific implementation. This will allow them to upgrade their APIs without any dependency concerns, thereby giving them the flexibility to evolve independently. The Data Spine should provide an intelligent and flexible infrastructure to align APIs “on-demand”, by creating workflows or “integration flows”.
- **Usability and multitenancy:** The Data Spine should provide an intuitive, low-code development environment to align the APIs of services and enable communication among them. It should be possible for the system integrator users to collaborate, but at the same time to limit access to their integration flows, when required.
- **Built-in functionality and tool/service integration effort:** The Data Spine should take care of the boilerplate code for protocol translation, routing, and mediation, etc., and facilitate the system integrator users for integrating their services by configuring only the service-specific parts of their integration flows with minimal coding effort.
- **Integration effort:** No local deployments of any Data Spine components should be needed to integrate 3<sup>rd</sup> party tools, services, or platforms with the ecosystem.
- **API management:** The system integrator users need to refer to the technical specifications of service APIs to create integration flows. The Service Registry component of the Data Spine should ensure uniformity across and completeness of the API specifications.
- **Modularity and extensibility:** The architecture of the Data Spine should be designed with modularity and extensibility in mind to meet the need for incorporating new tools, services, and platforms into the EFPF ecosystem, with minimal effort.
- **Performance, scalability, and availability:** As the Data Spine is a central entity of the platform ecosystem, it should be highly performant and should support high throughput. The performance critical components of the Data Spine should have the capability to operate within a cluster to support high availability.
- **Maintainability:** In the view of maintainability, the Data Spine should facilitate the creation of a loosely coupled, modular and an easily extensible ecosystem.
- **Documentation:** A comprehensive documentation that describes how different services can be integrated together using the Data Spine should be made available to the users.

Along with these initial guiding requirements for establishing a federated platform ecosystem listed above, the concrete technical requirements for the design and realisation of the Data Spine are derived from the four base platforms in the EFPF project. These four platforms are functionality complementary to each other and they offer services with minimum overlap, as shown in Figure 24. Thus, the EFPF ecosystem formed by initially integrating these platforms offers a rich set of functionalities to its users from the manufacturing domain.



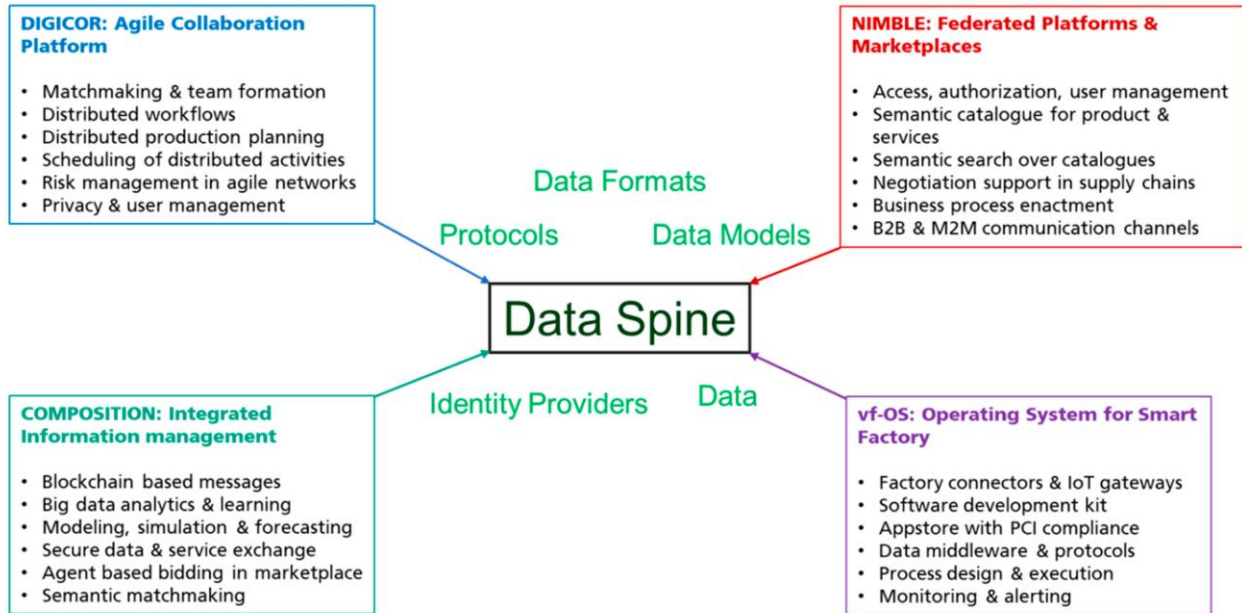


Figure 24. Conceptual Overview of the EFPF ecosystem

The technical profiles of the four base platforms are documented, including the specification of their tools, services and components, their maturity levels, exposed interfaces, protocols, data models, data formats, access control mechanisms, authentication providers supported, dependencies, programming environment, technical documentation, etc. The documented platform profiles are included in Annex C of ‘D3.1: EFPF Architecture-I’ and Figure 25 presents a summary of the platform profiles.

Technical Aspect	Summary of Adaptation by Services
Protocol	HTTP (REST)
	AMQP
	MQTT
	Minor adaptation: WebSockets, RPC, COBRA, RAW
Data Format	JSON
	Minor adaptation: XML, OPC-UP Binary, Proprietary (oneM2M/SAREF)
Data Model	UBL
	BPMN
	OGC-SensorThings
	OPC-UA
	Minor adaptation: Proprietary
Security Method	OAuth 2.0
	OpenID Connect
	Basic MQTT Authentication
	Minor adaptation: Basic Auth
Identity Provider	Keycloak
	Minor adaptation: Proprietary

Figure 25. Summary of the Technical Profiles of the Base Platforms

## 2.3 Interoperability Approach

The CEN/ISO 11354 Framework for Enterprise Interoperability [ISO11354] defines three different approaches to enterprise interoperability. These can also be applied to digital platform interoperability if the digital platforms are considered as heterogeneous enterprises.

- **Integrated Interoperability Approach:** This approach defines a common data model or a common Application Programming Interface (API) and all the digital platforms in the ecosystem must adhere to it.
- **Unified Interoperability Approach:** This approach defines a common, canonical data model at the ecosystem-level similar to the Integrated Interoperability Approach. However, in this approach, the common data model is defined at the meta-level and therefore it is a non-executable entity.
- **Federated Interoperability Approach:** This approach does not prescribe a common data model at the ecosystem and the all the digital platforms in the ecosystem are free to define and use their own data models. Interoperability needs to be enabled on-demand, when required by a use case, through data model transformation. The platforms need to share an ontology to map between their data models to establish interoperability.

The EFPF ecosystem consists of heterogeneous digital platforms that are owned and managed by independent entities. Therefore, enforcing a common data model or API at the ecosystem-level is not feasible. Defining a common canonical meta-model at the ecosystem-level is very difficult. With new tools, services and/or platforms joining the ecosystem, the administrator is burdened with updating the meta-model frequently, while also ensuring backwards compatibility with the already connected platforms. Therefore, Integrated and Unified Interoperability Approaches do not scale well with the increasing number of connected tools, services, and platforms in the ecosystem. In contrast, the Federated Interoperability Approach distributes the burden of establishing interoperability among the service consumers, when they want to consume a particular service, making it scalable with the rapid growth of the ecosystem. Therefore, the Data Spine interoperability approach aligns closely with the Federated Interoperability Approach.

## 2.4 Design of Interoperable Data Spine

The Data Spine is a collection of components that work together to form an integration, interoperability, and communications layer for the EFPF ecosystem. Figure 26 illustrates the architecture of the Data Spine showing a high-level conceptual view of the following core components that provide the expected functionality of the Data Spine:

- EFPF Security Portal (EFS)
- The Integration Flow Engine
- API Security Gateway
- Service Registry
- Message Bus

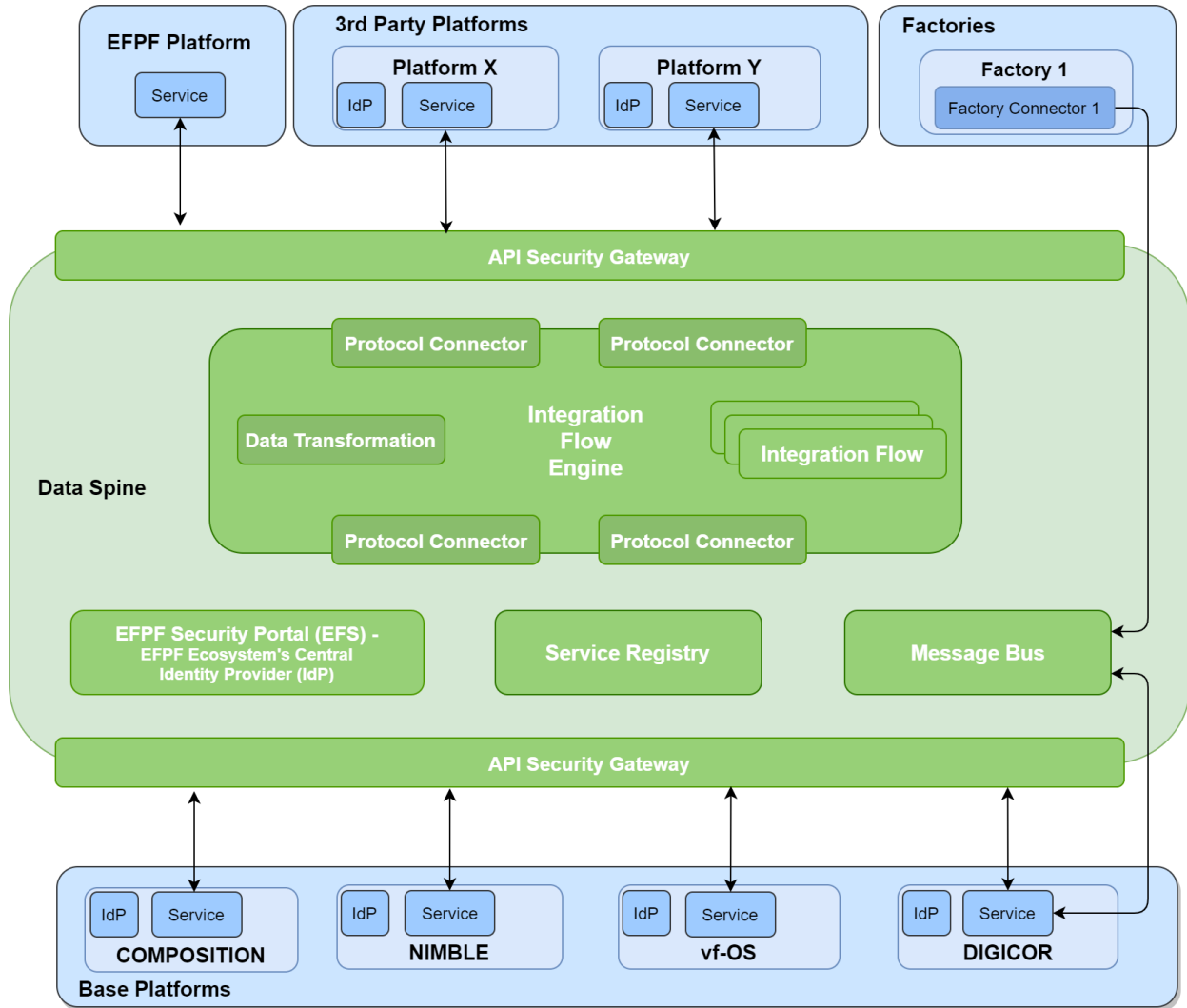


Figure 26. High-level Architecture of the Data Spine.

## 2.4.1 Components of the Data Spine

This section describes the core conceptual components of the Data Spine, their functionality, and the role they play in establishing the EFPF ecosystem and enabling cross-platform communication.

### 2.4.1.1 EFPF Security Portal (EFS)

The EFPF Security Portal implements a federated identity management mechanism that is designed to bridge security-related platform interoperability gaps. The platform ecosystem in EFPF consists of heterogeneous platforms owned by independent entities. Each of these platforms has its own Identity Provider (IdP). To enable collaboration and data exchange among the platforms, a user of one platform needs to access a service of another platform, which is provided implementing an SSO functionality in the EFPF ecosystem.

The EFPF ecosystem is designed to be an extensible platform ecosystem, enabling numerous login options for individual platforms (e.g., login to platform A, B, C, ...), which would require continuous updates to the authentication and authorization workflows for each platform in the ecosystem. The Security Portal is designed as a “distributed single point of

trust” that federates the identity providers (IdPs) of the connected platforms in order to enable an SSO connection among them. This federation of the IdPs supports users to seamlessly access the resources, i.e., tools, services, data, by using a single set of credentials.

The EFPF Security Portal takes on the role of a central identity provider solution for the EFPF ecosystem. The Web portal of each platform provides an additional “Login with EFS” option to allow logging in with a user account, as illustrated in Figure 27 below.

In case of adding new independent tools/services to the EFPF platform ecosystem, without using their own identity provider, the EFPF Security Portal acts as an identity and access management solution.

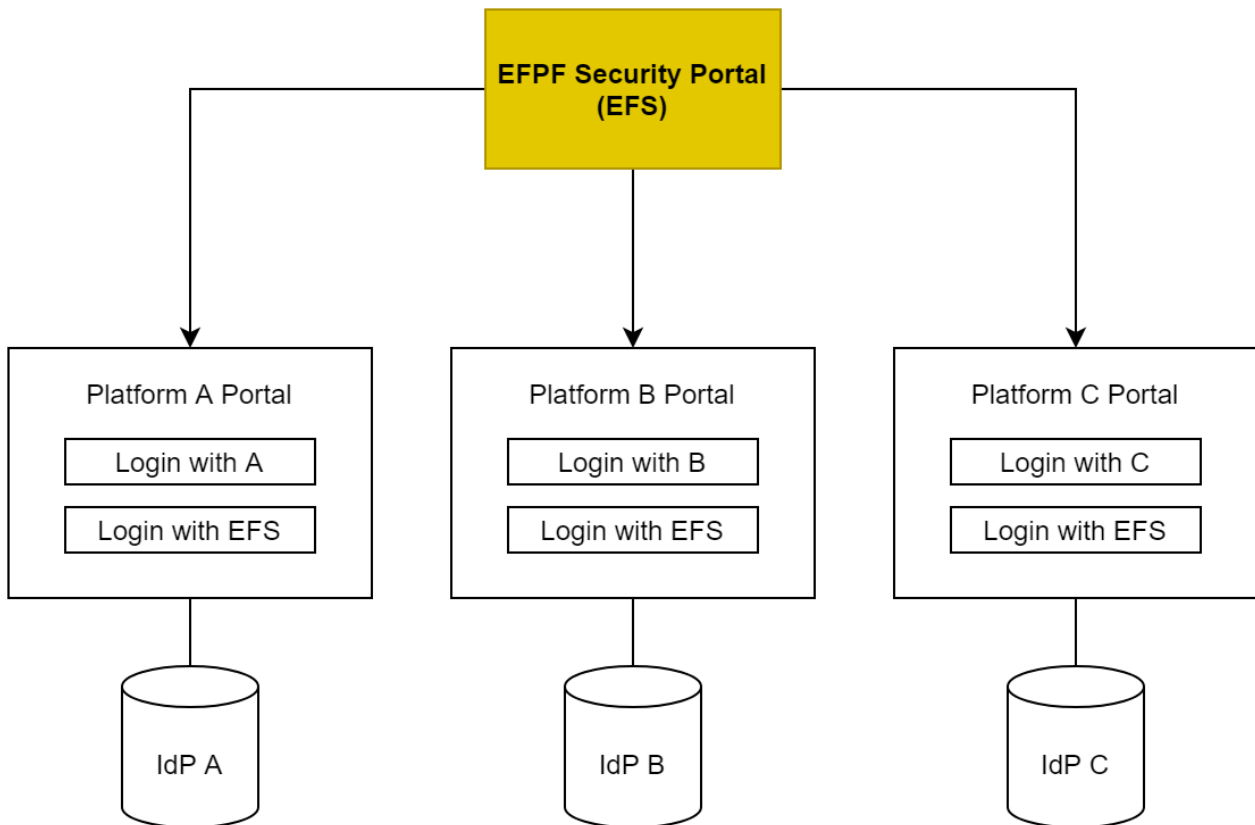


Figure 27. “Login with EFS” Option on the Platform’s Web portals

#### 2.4.1.2 Integration Flow Engine (IFE)

Integration Flow Engine (IFE) component of the Data Spine aims to bridge the interoperability gaps at the protocol level and the data model level between the heterogeneous services communicating through the Data Spine, in order to create composite applications. To enable interoperability among the services on the fly, the IFE makes use of dataflows or “Integration Flows”. The IFE is thus designed as a dataflow management system based on the concepts from flow-based programming. It provides functionalities such as connectivity, data routing, data transformation and system mediation.

The integration flows are designed and implemented as directed graphs that have ‘processors’ at their vertices and the edges represent the direction of the dataflow. The processors are of different types depending upon the functionality they provide: The processors of type ‘Protocol Connector’ address the issue of interlinking the services that

use heterogeneous communication protocols, the processors of type ‘Data Transformation Processor’ provide the means for transforming between data models and message formats, etc. The edges that represent the flow of information support routing of data based on certain parameters. The IFE provides functionality for the lifecycle management of the integration flows.

The IFE supports the Request-Response as well as the Pub/Sub communication pattern. An instance of the Integration Flow Engine should have built-in Protocol Connectors for standard communication protocols that are widely used in the industry. The functionality of the IFE can be extended by writing and adding custom processors to the IFE instance. For example, support for a new protocol can be added by writing a new Protocol Connector and adding it to the IFE instance.

The IFE offers an intuitive, drag-and-drop style, Web-based Graphical User Interface (GUI) to the service consumers or system integrators to create the integration flows which is based on the concepts from visual programming. This interface, the use of integration flows, and the built-in building blocks for connecting and aligning the APIs enable the users to create composite applications quickly, easily, and intuitively.

The IFE and its GUI support a fine-grained access control. It is possible to define access policies to allow or restrict visibility of and/or access to certain GUI elements. In addition, the IFE supports multitenancy and grouping of integration flows into “Development Spaces”. The access to the development spaces at view or modify levels can be restricted per user or user-group. This enables the ecosystem administrator to assign development spaces for companies where the system integrators from those companies can create their own integration flows. This helps shifting the data transformation burden across the Service Consumers making the architecture scalable and extensible. This also enables collaboration among the users who create the integration flows. Figure 28 illustrates the process for the creation of an integration flow using the drag-and-drop style GUI of the IFE. The process steps are described below:

1. Development space: The ecosystem administrator allots a development space for the company of the Service Consumer ‘consumer1’. consumer1 logs in to the GUI of the IFE and navigates to his/her company’s development space.
2. Reusable processors: The built-in reusable processors are loaded by the IFE and are displayed to the users on the GUI.
3. Creation of the integration flow: consumer1 drags and drops the processors in the development space, configures them, and connects them together to create an integration flow.
4. Functionality offered by the integration flow: The integration flow in this example consumes the API endpoint EP1-a provided by the Service Provider ‘provider1’, performs data model transformation, and makes the transformed data available over EP1-b, that is the “interoperability proxy” endpoint for EP1-a.

Thus, the IFE enables interoperability and creation of applications in an easy and intuitive manner.

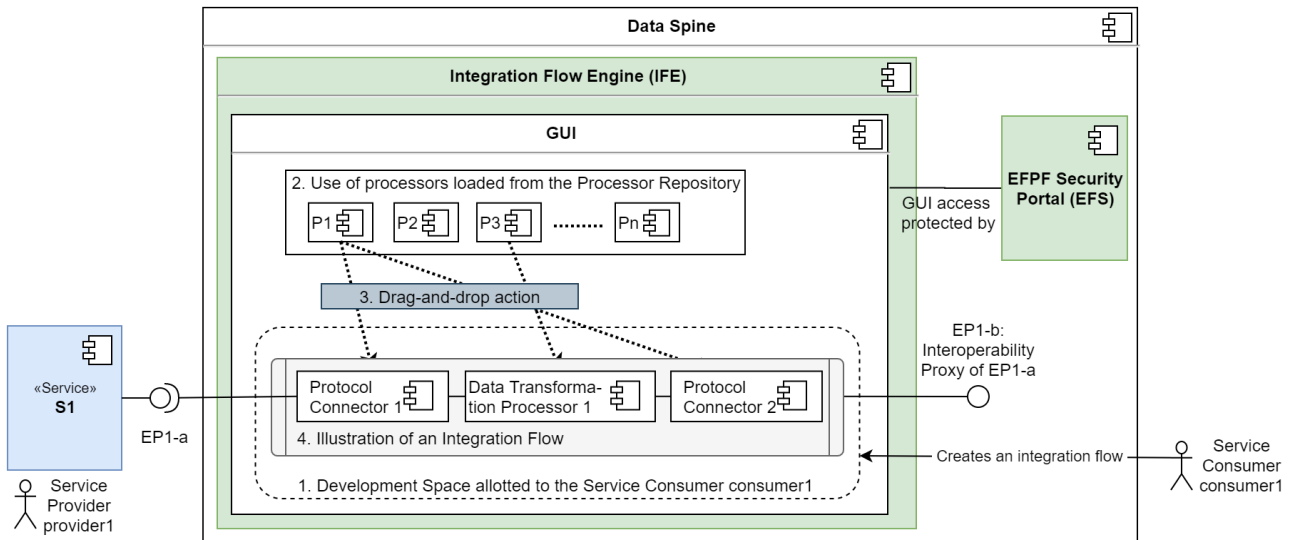


Figure 28. An illustration of the Creation of an Integration Flow

Furthermore, the IFE supports standard authentication and authorization protocols such as OpenID Connect (OIDC) and OAuth2.0 to secure access to its GUI using a pluggable identity provider. This enables connecting the IFE with the Security Portal in order to enable user authentication. Finally, to ensure high availability, throughput, and low latency, an instance of the IFE should be scalable and capable of operating in a clustered fashion.

### 2.4.1.3 API Security Gateway (ASG)

The API Security Gateway (ASG) acts as the Policy Enforcement Point (PEP) for the synchronous HTTP-based APIs, which are exposed by the integration flows created by users in the IFE. It connects to the Security Portal for making authentication and authorization decisions. The ASG can also provide permanent reverse proxy endpoints for the services that do not have fixed/permanent API endpoints.

Moreover, the ASG can be used to secure communication to the HTTP-based APIs of other components deployed in the same internal network, such as the Service Registry. The ASG also automates the process of creation of secure proxy endpoints for the services registered in the Service Registry.

### 2.4.1.4 Service Registry

In an interconnected platform ecosystem such as EFPF, the services of different platforms need to be composed together to achieve common objectives. For this purpose, the service consumers should be able to discover the available services, retrieve their API metadata, and consume them without the active involvement of the service providers. The Data Spine Service Registry provides the following mechanisms to fulfil these requirements:

- Registration and lifecycle management of service/API metadata for synchronous (Request-Response) as well as asynchronous (Pub/Sub) services in a uniform manner
- Discovery, lookup, and filtering of services
- Use of standard API specifications (specs) to capture service metadata to ensure the completeness of and uniformity across the API descriptions.

Figure 29 shows the abstract class diagram for the Service Registry that illustrates composition relationship between its classes. The notation '0..\*' in the diagram denotes 'zero

or more instances' of the concerned entity. As illustrated in the diagram, the Catalog of the Service Registry can have zero or more services. Each Service has zero or more APIs, and each API has exactly one API specification (Spec).

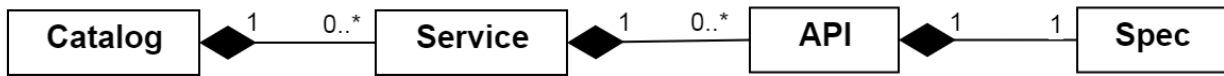


Figure 29. Abstract Class Diagram for the Service Registry

Figure 30 further shows the abstract schema for the Service object. The API Spec is obtained from an API Spec document. This API Spec document needs to conform to one of the following standards in order to ensure uniformity across and completeness of API specifications:

- For synchronous (Request-Response) services: OpenAPI/Swagger Spec [OAS22]
- For asynchronous (Pub/Sub) services: AsyncAPI Spec [AAS22]

Thus, this design makes the schema capable of managing metadata for synchronous (Request-Response) as well as asynchronous (Pub/Sub) type of services. All the technical metadata for the APIs of services that is needed for consuming the API can be obtained from these API Spec documents.

The 'type' could be used to categorise the services by giving a type to them based on the functionality they offer. In addition, any additional functional metadata related to the services, or the individual APIs can be stored in the respective 'meta' objects as key-value pairs. Thus, the basic schema can be extended to include additional metadata for the entire service or for a specific API.

```

{
  "id": "<unique id - custom or uuid>",
  "type": "string",
  "meta": {},
  "apis": [{
    "id": "string",
    "url": "<base url of the API>",
    "spec": {
      "mediaType": "<mediaType type of the API Spec document>",
      "url": "<url to API Spec document>"
    },
    "meta": {}
  }],
  "created": "2020-06-05T15:46:36.793Z",
  "updated": "2020-06-05T15:46:36.793Z"
}

```

Figure 30. Abstract Service Description Schema of the Service Registry

### 2.4.1.5 Message Bus

In the manufacturing domain, the Pub/Sub communication pattern is widely used. The shop-floor data from sensors is typically collected by a Factory Connector or an IoT Gateway and is made available to other services through a message-oriented middleware. The Data Spine Message Bus supports the Pub/Sub communication pattern. In addition, the Message Bus supports multitenancy and fine-grained access control. It provides interfaces for topic, user, and policy management.

An implementation of the Message Bus should support messaging protocols such as MQTT, AMQP, etc. that are widely used in the industry. The Message Bus can be extended to add support for new protocols via a plugin mechanism.

Figure 31 shows the conceptual components of the Message Bus. At the core of the Message Bus is the 'PubSub Service'. The PubSub Service provides a Pub/Sub API. The published messages are accepted by the PubSub Service and these messages are stored in queues/buffers provided by the Message Bus until they are forwarded to the designated subscribers. The Message Bus is capable of having multiple topics/channels and also sub-topics over which multiple publishers can publish messages and each topic/sub-topic can have multiple subscribers. In addition, the Message Bus also has an Identity and Access Management component so that the identities of the publishers and subscribers can be verified, and their publications and subscriptions can be access controlled. The Message Bus supports the use of username-password based authentication. Finally, the Message Bus provides interfaces for user and topic administration, management and monitoring which could be HTTP APIs or GUIs or even CLIs.

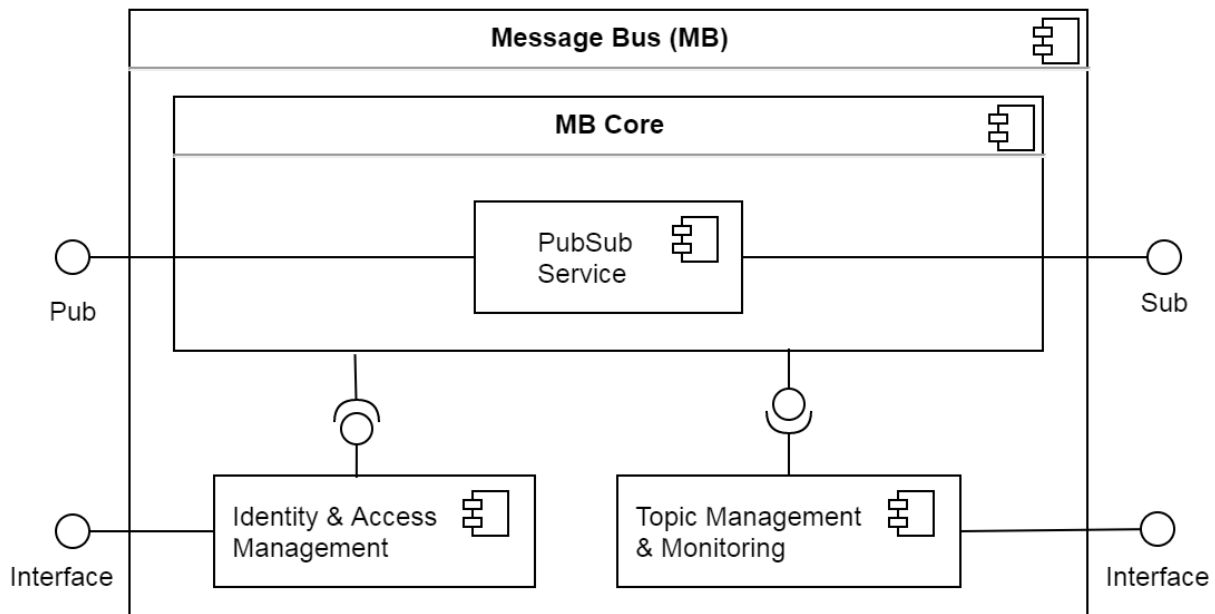


Figure 31. High-level Architecture of the Message Bus

## 2.4.2 Data Spine Architecture and Components' Interaction

Figure 32 illustrates the architecture of the Data Spine and interaction among its components. The access to the GUI of the IFE and its elements is protected by the Security Portal. The ASG acts as the Policy Enforcement Point and relies on the Security Portal to make the access control related decisions. The ASG secures the REST API of the Service Registry and offers a secure proxy endpoint to access it. The API Security Gateway is configured to check the Service Registry for new service registrations periodically in order to automatically create secure proxy endpoints for them. The Service Registry publishes service status announcement related messages to the Message Broker.



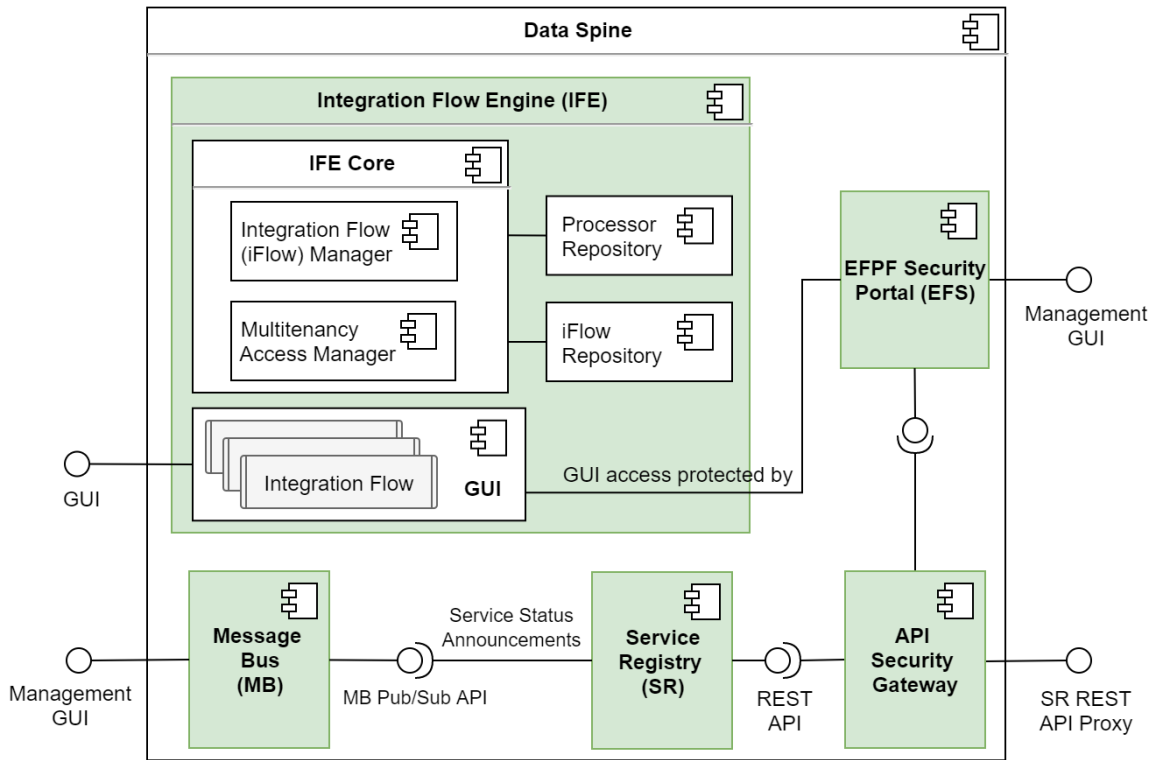


Figure 32. Architecture of the Data Spine

Figure 33 illustrates how the interoperability proxy endpoint EP1-b exposed by the integration flow is secured by the ASG. When the Service Consumer, consumer1 registers EP1-b to the Service Registry, the ASG automatically creates a security proxy endpoint EP1-c for EP1-b. consumer1 can then invoke EP1-c in through S2 with an access token obtained from the Security Portal.

In this way, the components of the Data Spine work together to enable integration of and communication between the services of different platforms.

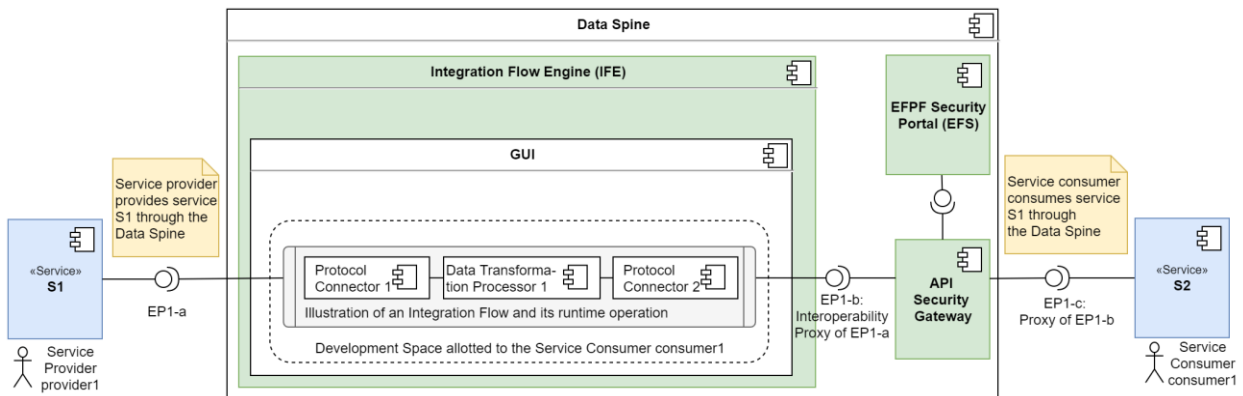


Figure 33. Illustration of an Integration Flow and its Runtime Operation

In summary, the Data Spine provides the following functionalities:

- Authentication, authorization and SSO
- Service/API metadata lifecycle management and discovery

- Infrastructure and tooling for protocol connection, data transformation, routing, and system mediation
- Multitenant, Web-based, drag-and-drop style GUI for an easy and intuitive creation of applications with minimal coding effort
- Message brokering

## 2.5 Realisation of Interoperable Data Spine

This section presents and describes the open-source technologies chosen to realise these conceptual components of the Data Spine as shown in Figure 34. In order to make this section as comprehensive and independent as possible, the sections from ‘D3.11: EFPF Data Spine Realisation – I’ have been taken and updates have been added. The work on Data Spine after D3.11 (M18) focused not only on improving the Data Spine’s technology stack and its integration, but also on automating the deployment, defining, and automating integration, system, and performance tests, improving the user documentation and supporting the implementation of the pilot and Open Call experimentation scenarios. The subsequent sections briefly describe these activities. The process followed for the selection of the open-source technologies to realise the components of the Data Spine is described in D3.1 (M9) and D3.11 (M18).

Conceptual Component	Technology	Version
<b>EFS</b>	Keycloak	3.4.0
<b>Integration Flow Engine</b>	Apache NiFi	1.11.4
<b>API Security Gateway</b>	Apache APISIX & asg-importer	2.3.0 & 1.0.0
<b>Service Registry</b>	LinkSmart Service Catalog	3.0.0-beta.1
<b>Message Bus</b>	RabbitMQ	3.8.5

Figure 34. Technologies Selected to Realise the Data Spine

### 2.5.1 EFPF Security Portal

The EFPF project uses the Keycloak identity provider to realise the EFS. The following lists the main functionalities performed by the EFS.

- User Management
- User Federation Service, SSO, Loggers
- Policy Enforcement Service

#### 2.5.1.1 Architecture and Interfaces

##### User Management

Keycloak is an OpenID connect (OIDC) and User-Managed Access (UMA) compliant identity provider. Keycloak handles user management and authentication, and also provides an Admin API to perform user management and a fully extensible plugin-based ecosystem.

##### User Federation Service

To enable user’s login to any of the connected platforms, the user can select one of the two procedures for the authorization:

- Login through a connected platform (native users), and
- Login through the EFPF platform (federated users).

The platform-level interoperability in EFPF can be achieved by following workflows 1 and 2, as shown in Figure 35 and Figure 36 respectively. Note that both workflows require the user to be registered on the EFPF platform.

Figure 35 illustrates the workflow that follows the bottom-up approach for federation. Here, the user logs in to the connected platform using his/her EFPF credentials. The EFPF credentials are issued by the EFS and are not propagated to the connected platforms. The initial representation of the user will be created when the user opts to login with EFPF credentials in a connected platform. If a user is already present in the connected platform, then a linked user will be created with the existing roles of the connected platform.

The second workflow, presented in Figure 36, enables the user to login to the EFPF platform (via EFS) and then visit any connected platform in the same browser session, e.g., PLATFORM 1, PLATFORM 2, etc. In this approach, the user logs in to the connected platforms using his/her previously provided EFPF credentials. By keeping the common browser session, the EFPF user can achieve SSO capability when logging in to other connected platforms.

The implementation of the EFS using Keycloak follows the OIDC based identity and federation method. Here, the user who tries to login to the connected platform using EFPF credentials, is redirected to the EFPF platform’s Identity Provider, i.e., the EFS, for the validation of his/her credentials. After verifying the credentials, the user is redirected to the relevant connected platform. Furthermore, the connected platform identifies the user and provides required roles based on predefined policies.

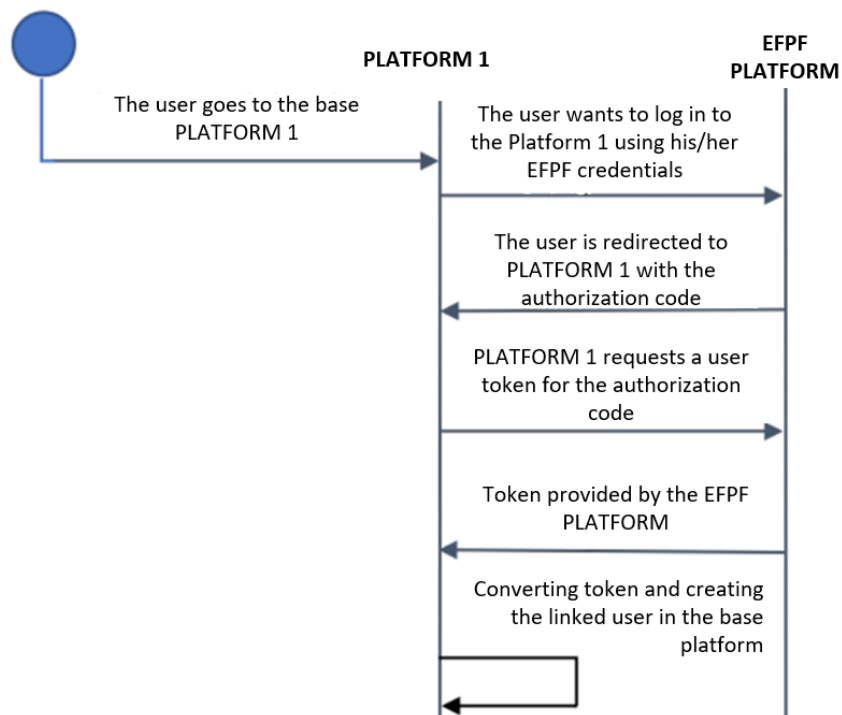


Figure 35. Workflow 1: Login to PLATFORM 1 using EFPF platform credentials

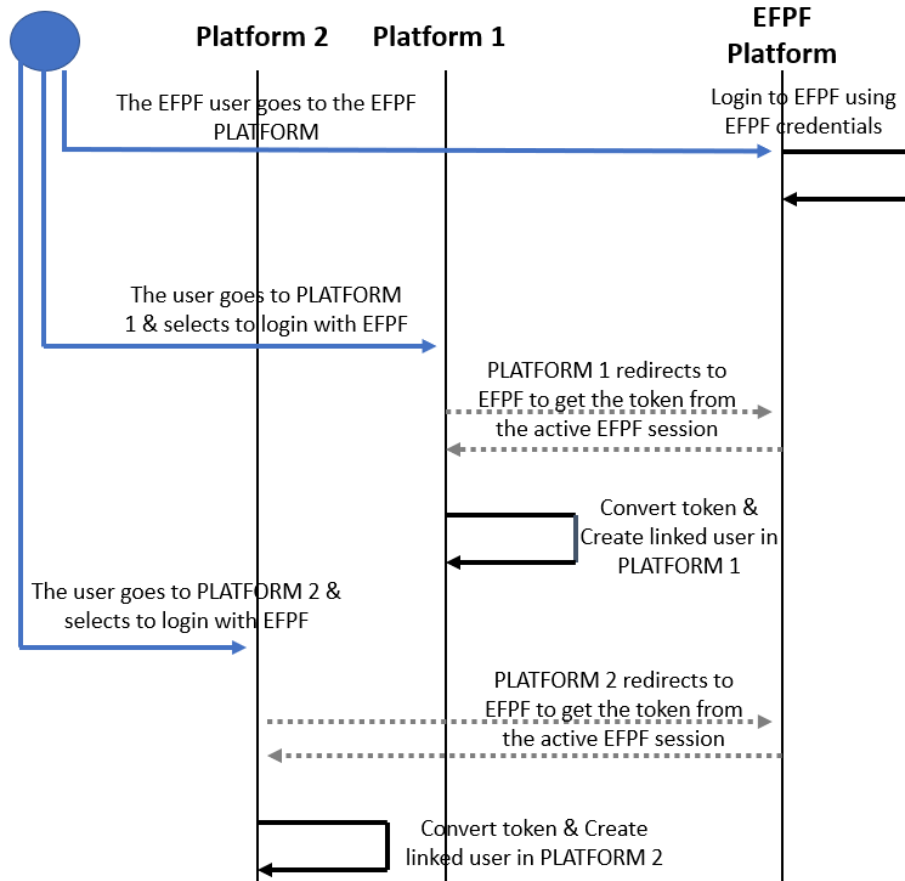


Figure 36. Workflow 2: Login to EFPP using EFPP platform credentials, then login to PLATFORM 1 and PLATFORM 2

### Policy Enforcement Service

Policy Enforcement Service acts as the first contact point from the API Security Gateway. This allows the enforcement of the policies for the HTTP-based APIs exposed by the Data Spine. EFS allows 2 types of permissions to be created:

1. Resource-Based: The permission can be directly applied to a resource created in the identity provider.
2. Scope-Based: The permission can be assigned to scopes or scope(s) and resource.

Scopes represent a set of rights at a protected resource. Scopes can be resource specific or can be shared between multiple resources. Figure 37 shows the architecture of the policy enforcement service.

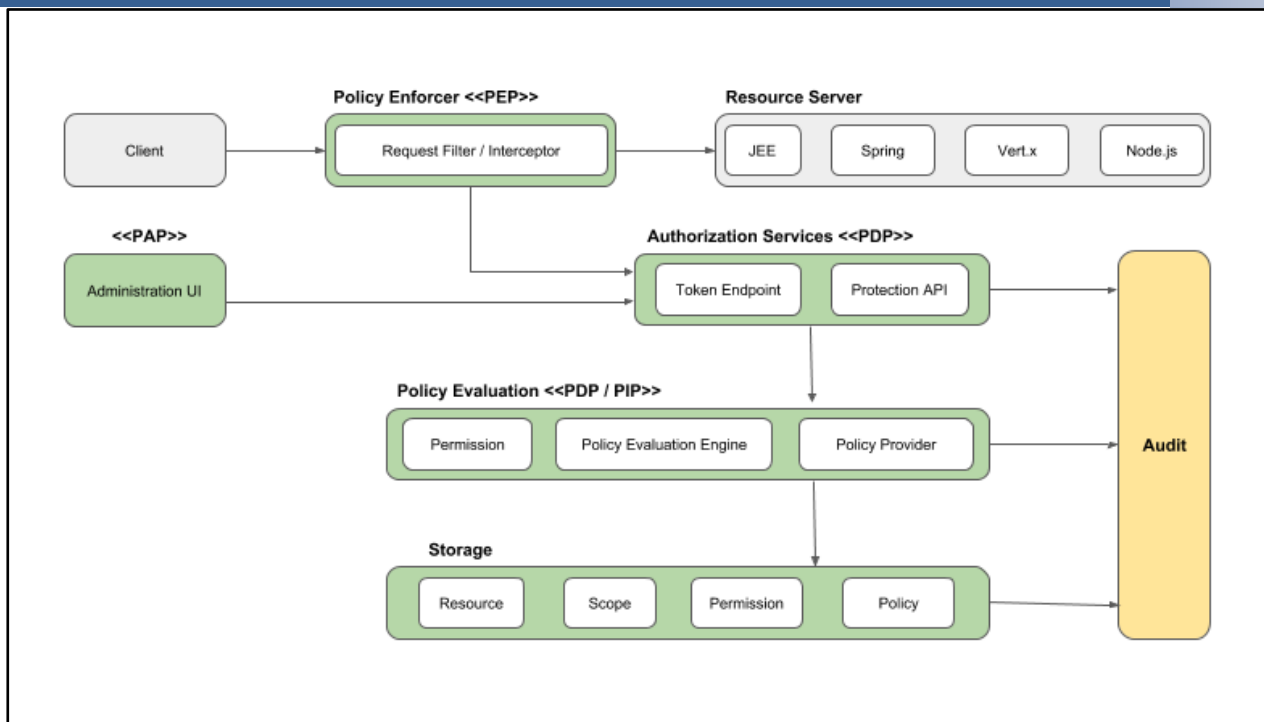


Figure 37. Policy Enforcement Architecture

### 2.5.1.2 Configuration

#### Identity Provider Configuration

The EFS Keycloak identity provider is configured to use the OIDC-based mechanism to authenticate and authorize the users. Trusted clients should be created in the identity provider for the applications to authenticate users and micro-services in the ecosystem.

#### SSO Configurations

To enable SSO, as illustrated in Figure 38, a dedicated client for each connected platform should be created in the EFS Keycloak identity provider (IdP). The SSO client will have the redirect URL and other OIDC configurations such as the OAuth flows supported for the client. Then, if a connected platform has a Keycloak-based IdP, the EFS Keycloak IdP must be added as a trusted IdP to it, as depicted in Figure 39. For non-Keycloak-based IdPs, the steps below should be followed:

1. Obtain an authorization code for the user
  - Redirect the user to the following URL and replace the values 'your\_client\_id', 'your\_redirect\_uri', and 'ccd9' with your custom values.  
`https://<EFS Keycloak URL>/auth/realms/<realm name>/protocol/openid-connect/auth?client_id=your_client_id&redirect_uri=your_redirect_uri&response_type=code&scope=openid&nonce=ccd9`
  - Once the user logs in to the EFS, the EFS will redirect the call to the given redirect\_uri with an authorization code. Extract the code value from the URL.
2. Obtain an ID Token with the Authorization Code Grant Type using the code value from step (1).
3. Decode the ID Token to obtain the user details

#### 4. Create the User in the IdP of the connected platform

Clients > nimble-federation-client

Nimble-federation-client

Settings | Credentials | Roles | Client Scopes | Mappers | Scope | Authorization | Revocation | Sessions | Offline Access

Client ID : nimble-federation-client

Name :

Description :

Enabled :

Consent Required :

Login Theme :

Client Protocol : openid-connect

Access Type : confidential

Standard Flow Enabled :

Implicit Flow Enabled :

Direct Access Grants Enabled :

Service Accounts Enabled :

Authorization Enabled :

Root URL :

\* Valid Redirect URIs :

Figure 38. A Sample Client in EFS Keycloak to Enable SSO

Login with EFPF

Settings | Mappers | Permissions

Redirect URI : http://nimble-staging-neu.salzburgresearch.at:8080/auth/realms/master/broker/EFS/endpoint

\* Alias : EFS

Display Name : Login with EFPF

Enabled :

Store Tokens :

Stored Tokens Readable :

Trust Email :

Account Linking Only :

Hide on Login Page :

GUI order :

First Login Flow : first broker login

Post Login Flow :

OpenID Connect Config

\* Authorization URL : https://efpf-security-portal.salzburgresearch.at/auth/realms/master/protocol/openid-connect/auth

Pass login\_hint :

\* Token URL : https://efpf-security-portal.salzburgresearch.at/auth/realms/master/protocol/openid-connect/token

Logout URL :

Backchannel Logout :

Disable User Info :

User Info URL : https://efpf-security-portal.salzburgresearch.at/auth/realms/master/protocol/openid-connect/userinfo

\* Client ID : nimble-federation-client

\* Client Secret : .....

Issuer :

Default Scopes :

Figure 39. Adding EFS Keycloak as a Trusted IdP in a Connected Platform's IdP for Enabling SSO

By default, the EFS Keycloak identity server supports the standard flows. However, if the base platform does not have a back end, then an implicit flow can be enabled to exchange SSO tokens.

### 2.5.1.3 Operation

The components of the EFS are packaged and containerized as a docker-compose solution. All the sub-components of the EFS are dependent on the Identity Server. Therefore, the rest of the EFS components are initiated after the Identity Server runs. The components of the EFS are monitored using the access logs. The Identity Server also provides several functionalities to ensure smooth operation such as detection of brute forcing or any suspicious activities.

## 2.5.2 Integration Flow Engine

Apache NiFi was selected to be the Integration Flow Engine of the Data Spine. It is a dataflow management platform based on the concepts of Flow-based programming. It automates the flow of information between systems through directed graphs called dataflows. The dataflows support communication, data routing, data transformation and system mediation logic with the help of ‘processors’ as their vertices.

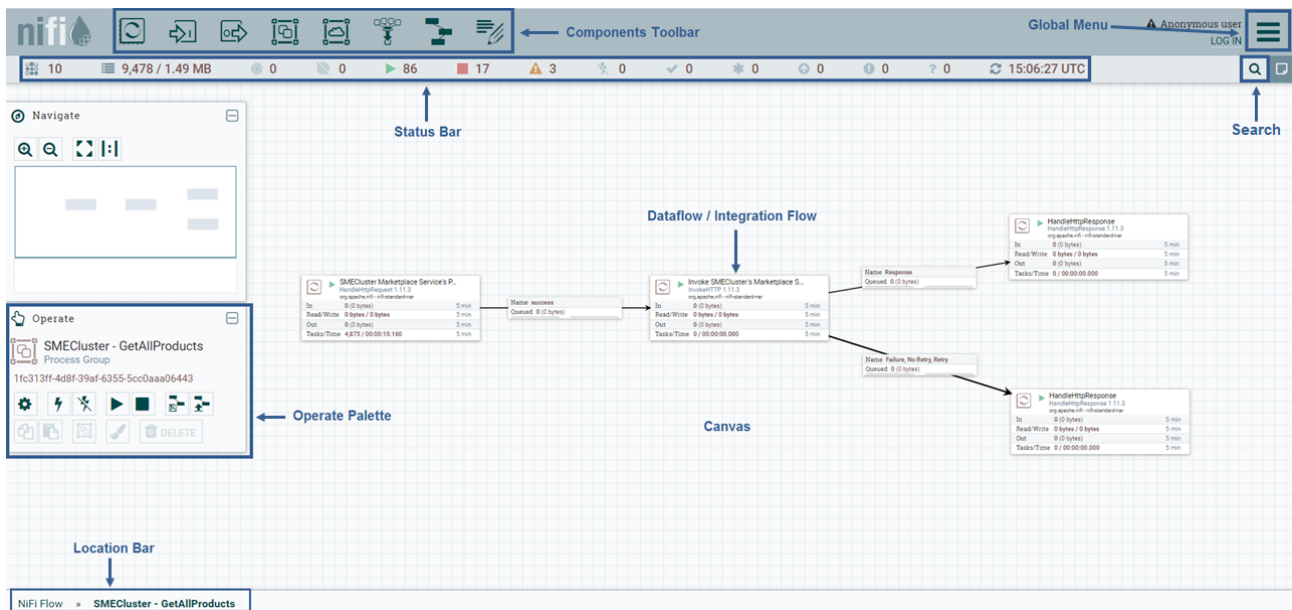


Figure 40. Apache NiFi GUI Elements

The processors are responsible for handling data ingress, egress, routing, mediation, and transformation. The edges that connect these processors with each other are called ‘Connections’. Apache NiFi offers a Web-based, highly configurable, drag-and-drop style GUI for creating such dataflows. Figure 40 highlights the elements of NiFi’s GUI and also shows a sample dataflow. NiFi’s GUI offers a functionality to search for a particular processor and view its short description to include it in a dataflow as shown in Figure 41. NiFi contains as many as 284 different processors as of version 1.11.3. In the context of Data Spine, the Integration Flows translate to dataflows in NiFi. Henceforth, dataflows would be referred to as Integration Flows in this document.

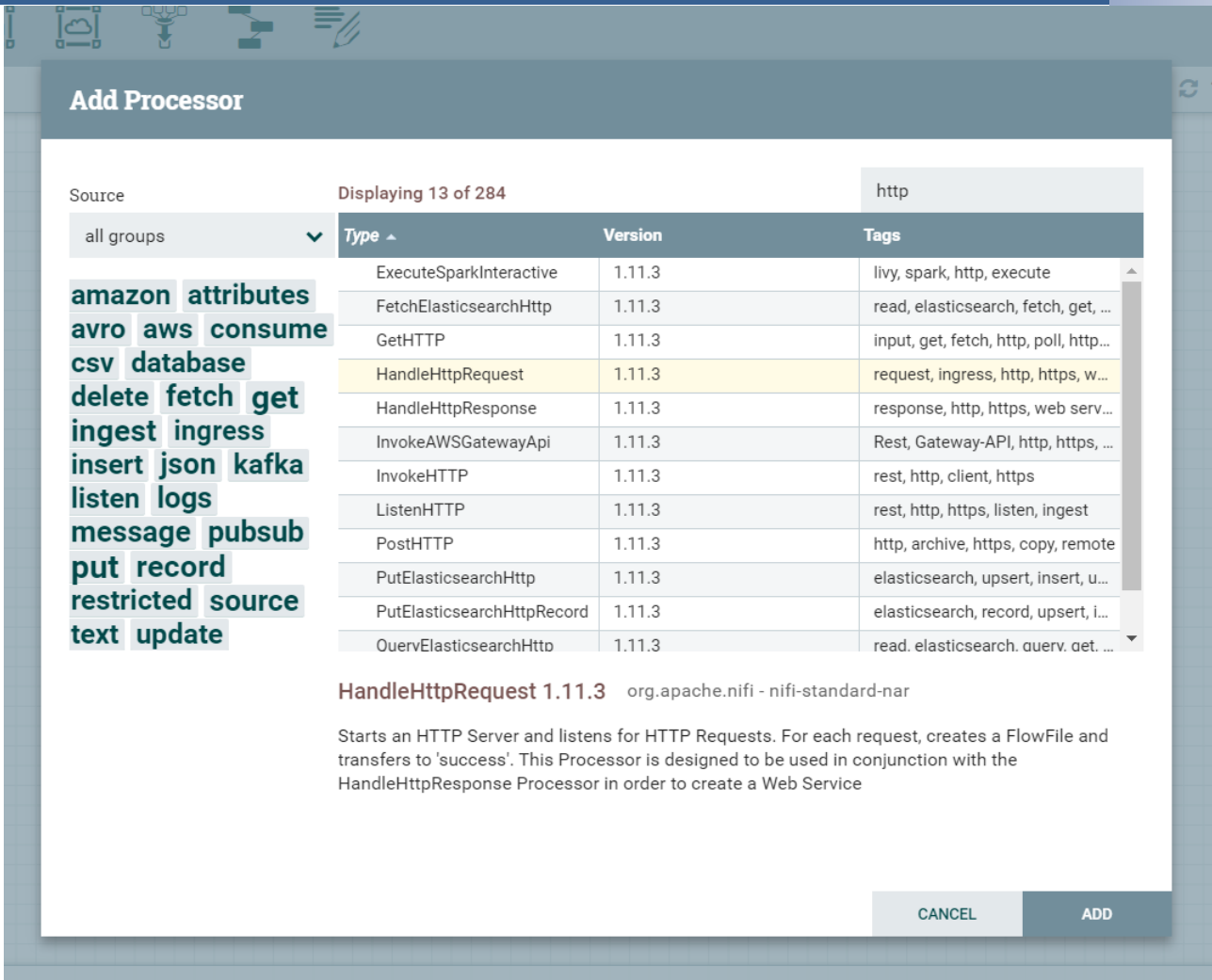


Figure 41. Apache NiFi Processors

Apache NiFi’s conformity to the fundamental design requirements for Integration Flow Engine identified in Section 2.4.1.2 is discussed below:

- **License:** Apache NiFi comes with Apache License v2.0.
- **Usability:** Apache NiFi provides an intuitive, drag-and-drop style GUI to the developers to create the Integration Flows with minimal effort. The collaboration of work concerning a particular Integration Flow among different developers is easy to manage as NiFi provides a Web-based GUI for creating Integration Flows and a Multi-tenant authorization capability that enables different groups of users to command, control, and observe different parts of the dataflow, with different levels of authorization. Therefore, NiFi was found to be in compliance of the requirements of usability, developer productivity and ease of collaboration.
- **Built-in Protocol Connectors:** NiFi provides connectors for standard communication protocols such as HTTP, MQTT, AMQP, etc. that are widely used in the industry. In addition, it provides processors for directly connecting with widely used industrial grade systems such as Apache Kafka, MongoDB, Elasticsearch, AWS DynamoDB, AWS S3, etc.
- **Built-in Data Transformation Processors:** NiFi provides several data transformation processors: JoltTransformJSON, TransformXml, ExecuteScript, ReplaceText, MergeContent, etc. These include TransformXml processor that supports



transformations with XSLT which is a WC3 standard, a Turing complete language for transformations and widely known in the industry. But XSLT has a steeper learning curve. Whereas, JoltTransformJSON, a processor which uses Jolt transformation rules to transform one JSON data model to another JSON data model, is easier to learn, but not Turing complete. Apart from these two processors, NiFi also offers ExecuteScript processor that facilitates users in writing a script for performing data transformation.

- **Extensibility:** NiFi is at its core built with extensibility in consideration. Points of extension include: Processors, Controller Services, Reporting Tasks, Prioritizers, and Customer User Interfaces. For example, it is possible to write a custom processor for NiFi in order to connect to an OPC-UA server (based on OPC-UA Java Stack) and read the data.
- **Performance and scalability:** NiFi was observed to work seamlessly with resource allocation of 8GB RAM and 2 CPU cores. NiFi is also able to operate within a cluster.
- **Identity and access management:** NiFi supports a pluggable OpenID Connect based authentication provider such as Keycloak. Alternatively, NiFi also supports user authentication via client certificates, via username/password with pluggable Login Identity Provider options for Lightweight Directory Access Protocol (LDAP) and Kerberos or via Apache Knox.
- **Component integration effort:** NiFi provides connectors for integration with external components. E.g., for integration with Kafka, NiFi has 20 built-in processors. Integration of NiFi with REST APIs of other components such as EFS was done with minimal effort.
- **Maintainability and Documentation:** NiFi's GUI is very simple, intuitive, drag-and-drop style and easy to manage. NiFi has a comprehensive documentation that covers different aspects of the Platform and different perspectives. NiFi has a Getting Started Guide, a User Guide, an Expression Language Guide, RecordPath Guide, Administrator's Guide, a Developer's Guide, In Depth Guide and also the documentation of its REST API. NiFi has a strong community and has frequent source code releases.

Thus, Apache NiFi complies with the foundational design requirements identified for the Integration Flow Engine of the Data Spine.

Some other additional key features of NiFi include:

- **Flow Management**
  - NiFi supports guaranteed delivery with the help of persistent write-ahead log and content repository, even at a very high scale.
  - The Connection queues of NiFi support data buffering and can be configured to apply back pressure upon reaching a certain limit or can age off data.
  - NiFi supports prioritized queuing where data can be retrieved from queues based on various strategies such as oldest first, newest first, largest first, or some other custom scheme.
  - NiFi supports Flow Specific QoS i.e., it can be configured to prefer low latency vs high throughput or loss tolerant vs guaranteed delivery.
- **Ease of Use**
  - Apart from the easy to use drag-and-drop style GUI, NiFi also supports visual command, control and debugging where parts of the Integration Flow can be

stopped at runtime and queues can be examined. Also, changes can be made to any Integration Flow at real-time and those changes immediately take effect.

- Visually created Integration Flows in NiFi are represented as XML documents in the backend. NiFi supports download-upload of these XML Integration Flows and also saving them as templates – thereby enabling reuse and collaboration.
- **Data Provenance**
  - Data supports automatic recording of provenance of data related to the Integration Flows – a feature that would prove to be very useful in Production Environments for debugging, finding out the history of changes to a particular Integration Flow for troubleshooting and for ensuring compliance.
- **Flexible Scaling Model**
  - Scale-out (Clustering): NiFi supports scaling-out through the use of clustering.
  - Scale-up and down: NiFi also be scaled-up and down in a flexible manner. To handle increasing throughput, the processors in an Integration Flow can be configured to increase the number of concurrent processors.

Thus, Apache NiFi was identified as the most suitable candidate to realise the Integration Flow Engine of the Data Spine.

### 2.5.2.1 Architecture and Interfaces

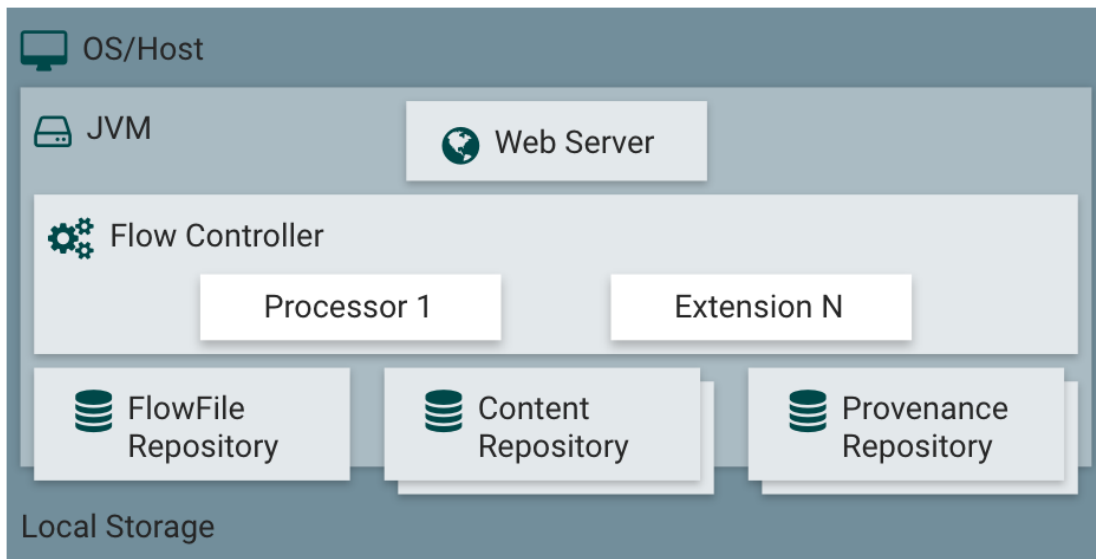


Figure 42. Architecture of Apache NiFi [NOG20]

Figure 42 shows the primary internal components of Apache NiFi. NiFi executes inside a JVM on the host operating system. The Web Server component hosts NiFi's RESTful HTTP-based command and control API. The Flow Controller is the central component that manages the execution of processors and extensions. It provides threads for the execution of processors and handles their scheduling when the processors or extensions receive resources to execute. As discussed before, NiFi supports custom extensions. These extensions also run within the same JVM as the in-built components of NiFi. NiFi has three different types of repositories for storing different types of data. The FlowFile Repository captures the runtime state of NiFi where it stores the metadata state of its FlowFiles at a

given time. FlowFile is the data serialization format used by NiFi. The FlowFile Repository implementation instance is pluggable, and it uses Write-Ahead Log located on a specified disk partition. The Content Repository stores the actual contents of a given FlowFile. The Content Repository construct is also pluggable. Finally, the Provenance Repository stores provenance data for all the events and actions. The implementation of Provenance Repository is pluggable as well. The event data stored in the Provenance Repository is indexed and searchable.

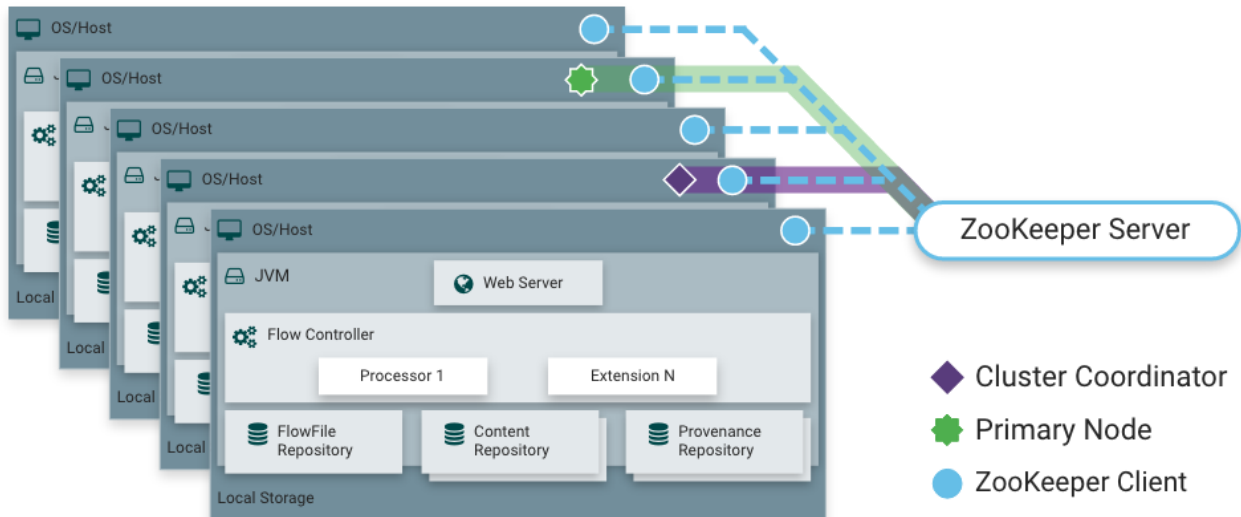


Figure 43. Apache NiFi Cluster

NiFi also supports scaling-out through the use of clustering. Figure 43 shows the operation of NiFi within a cluster. Apache ZooKeeper can be used to enable clustering. However, in the EFPF Production Environment, NiFi cluster is enabled using the Docker Swarm technology, which is followed as the uniform clustering technology for the other Data Spine components as well.

NiFi offers a RESTful HTTP-based API [NAR20] that provides a functionality to programmatically send commands to control a NiFi instance at runtime. The API provides endpoints for lifecycle management of Integration Flows, processors, Process Groups, users, access policies, templates, etc. The API also provides User authentication and token endpoints e.g., to authenticate a request through the plugged OpenID Connect provider. Moreover, the API offers control endpoints that can be used e.g., to start and stop processors at real-time; debugging endpoints that can be used to monitor queues, query provenance data, etc.

### 2.5.2.2 Configuration

In order for two heterogeneous services to be able to communicate, they must be integrated and interoperated through the Data Spine beforehand which is accomplished with the help of Integration Flows. The Integration Flow Engine needs to be configured for allowing its usage and facilitating collaboration among system integrator users who create the Integration Flows. Such design-time aspects and prerequisites to the run-time operation of Integration Flows are discussed in this section.

#### Securing a NiFi Instance:

In order to setup a secure instance of NiFi, an SSL certificate, a keystore, a truststore, etc. need to be setup at first. To automatically generate the required certificate, keystores, truststore, and relevant configuration files, a `tls-toolkit` [NTT20] command line utility provided by NiFi can be used. The `tls-toolkit` also alters `nifi.properties` file to set `nifi.web.https.port=9443` and remove `nifi.web.http.port=8080`. Once this configuration is done, NiFi's GUI becomes accessible over https. The `tls-toolkit` is especially useful for securing multiple NiFi nodes.

### User Authentication:

Users using NiFi's Web-based GUI to create Integration Flows need to be authenticated first. NiFi supports user authentication via client certificates using 2-way SSL, via username/password with Login Identity Providers such as LDAP and Kerberos, via Apache Knox, or via OpenID Connect (OIDC). NiFi can be configured to use one of these at a given time. In EFPP, NiFi's GUI is secured using EFS's Identity Provider via OIDC. To enable user authenticated via OIDC, the properties as shown in Figure 44 are configured in `nifi.properties` file.

```
# OpenId Connect SSO Properties #
nifi.security.user.oidc.discovery.url=http://localhost:9090/auth/realms/master/.well-known/openid-configuration
nifi.security.user.oidc.connect.timeout=100 secs
nifi.security.user.oidc.read.timeout=5 secs
nifi.security.user.oidc.client.id=nifi-client
nifi.security.user.oidc.client.secret=c73d0448-e53c-4c92-a31e-8545f2b0868e
nifi.security.user.oidc.preferred.jwsalgorithm=RS256
```

Figure 44. Sample OIDC Properties Configuration for NiFi

After this configuration, NiFi would redirect to EFS's Identity Provider i.e., Keycloak for login and after a successful login display 'Insufficient Permissions' error as access policies for logged in users still need to be configured. Thus, NiFi is able to use the same user-base from EFS and user lifecycle management can take place at a single place i.e., EFS.

### Multi-Tenant Authorization:

NiFi's Web-based GUI is intended to be used by multiple users for creating Integration Flows. The access and visibility of such Integration Flows needs to be restricted to their creators only and, restricted and hidden from the other users. Furthermore, user collaboration over an Integration Flow or a Process Group containing several Integration Flows needs to be facilitated. This requires not only configuring who has access to the Process Groups but also the level of their access. NiFi provides this functionality through its 'Multi-Tenant Authorization' policy governance framework. The Multi-Tenant Authorization enables multiple groups of users to collaboratively view, control and manipulate different parts of the Integration Flows, with different levels of authorization. Thus, when a logged in user attempts to view or update a particular resource through NiFi's GUI, NiFi, based on the configured privileges for the user allows or denies that particular action. To define such privileges for individual users or user groups, the access policies need to be defined.

For Data Spine, two different user roles would be defined in the EFS: DS-Admin and DS-User. These would be mapped to user groups in NiFi named 'DS-Admins' and 'DS-Users' respectively. Upon login to NiFi through EFS, with no user accounts in NiFi (and hence with no Authorization policies defined), 'Insufficient Permissions' message is displayed. Thus, the First User ('Initial User Identity') and access policies for that user need to be hard-coded into the config files of NiFi. The access policies for the users belonging to the DS-

Admins group are then configured by this First User enabling them to configure access policies for DS-Users. In EFPF, the access policies for these user roles/groups need to be defined at three different levels:

- **Global Access Policies:** The Global Access Policies define privileges for uses that are applicable system-wide. Figure 45 shows the Global Access Policies defined for NiFi in the EFPF Project.

Role (Group)				Policy	Privilege
DS-Admin (Group: DS-Admins)		DS-User (Group: DS-Users)			
View	Modify	View	Modify		
Y	NA	Y	NA	view the UI	Allow users to view the UI
Y	Y	Y	Y	access the controller	Allows users to view/modify the controller including Reporting Tasks, Controller Services, Parameter Contexts and Nodes in the Cluster
Y	Y	Y	Y	access parameter contexts	Allows users to view/modify Parameter Contexts. Access to Parameter Contexts is inherited from the "access the controller" policies unless overridden.
Y	Y	N	N	query provenance	Allows users to submit a Provenance Search and request Event Lineage
Y	Y	Y	N	access restricted components	Allows users to create/modify restricted components assuming other permissions are sufficient. The restricted components may indicate which specific permissions are required. Permissions can be granted for specific restrictions or be granted regardless of restrictions. If permission is granted regardless of restrictions, the user can create/modify all restricted components.
Y	Y	N	N	access all policies	Allows users to view/modify the policies for all components
Y	Y	Y	N	access users/user groups	Allows users to view/modify the users and user groups
Y	Y	N	NA	retrieve site-to-site details	Allows other NiFi instances to retrieve Site-To-Site details
Y	Y	N	NA	view system diagnostics	Allows users to view System Diagnostics
Y	Y	N	N	proxy user requests	Allows proxy machines to send requests on the behalf of others
Y	Y	N	N	access counters	Allows users to view/modify Counters

Figure 45: NiFi Global Access Policies

- **Component-level Access Policies for the Root Process Group ('NiFi Flow'):** These policies define privileges for uses that are applicable to the Root Process Group of NiFi and the components (Process Groups, Integration Flows, processors, etc.) present in it. Figure 46 shows the Component-level Access Policies for the Root Process Group 'NiFi Flow' in the EFPF Project.

DS-Admin (Group: DS-Admins)	DS-User (Group: DS-Users)	Policy	Privilege
Y	N	view the component	Allows users to view component configuration details
Y	N	modify the component	Allows users to modify component configuration details
Y	N	operate the component	Allows users to operate components by changing component run status (start/stop/enable/disable), remote port transmission status, or terminating processor threads
Y	N	view provenance	Allows users to view provenance events generated by this component
Y	N	view the data	Allows users to view metadata and content for this component in flowfile queues in outbound connections and through provenance events
Y	N	modify the data	Allows users to empty flowfile queues in outbound connections and submit replays through provenance events
Y	N	view the policies	Allows users to view the list of users who can view/modify a component
Y	N	modify the policies	Allows users to modify the list of users who can view/modify a component
Y	N	receive data via site-to-site	Allows a port to receive data from NiFi instances
Y	N	send data via site-to-site	Allows a port to send data from NiFi instances

Figure 46. Component-level Access Policies for the Root Process Group ('NiFi Flow')

- Component-level Access Policies for a particular component (Process Group) PG\_X:** These policies define privileges for users that are applicable to that particular component (Process Group, Template, etc.). Figure 47 shows the recommended component-level access policies to be configured by the Component-Owner. These can be customized based on requirements by the Component-Owner. When a new System Integrator user in EFPF wants to create Integration Flows in NiFi, DS-Admin creates a new component 'PG\_X' for his/her company/project and grants him/her 'admin' level privileges for PG\_X and he/she are regarded as the component-owner for PG\_X. The component-owner can then grant permissions for PG\_X to others (e.g., by creating a user group such as 'PG\_X-Admins', 'PG\_X-Users', etc.).

Component-Owner for PG_X User-Group: PG_X-Owners	Component-User for PG_X User-Group: PG_X-Users	Policy	Privilege
Y	Y	view the component	Allows users to view component configuration details
Y	Y	modify the component	Allows users to modify component configuration details
Y	Y/N	operate the component	Allows users to operate components by changing component run status (start/stop/enable/disable), remote port transmission status, or terminating processor threads

Y	Y/N	view provenance	Allows users to view provenance events generated by this component
Y	Y	view the data	Allows users to view metadata and content for this component in flowfile queues in outbound connections and through provenance events
Y	Y/N	modify the data	Allows users to empty flowfile queues in outbound connections and submit replays through provenance events
Y	Y/N	view the policies	Allows users to view the list of users who can view/modify a component
Y	Y/N	modify the policies	Allows users to modify the list of users who can view/modify a component
Y	N	receive data via site-to-site	Allows a port to receive data from NiFi instances
Y	N	send data via site-to-site	Allows a port to send data from NiFi instances

Figure 47. Component-level Access Policies for a particular component (Process Group) PG\_X

Figure 48 illustrates an example of NiFi’s Multi-Tenant Authorization. The users ‘admin’ and ‘user1’ have roles DS-Admin and DS-User respectively and user1 is the Component-Owner for the Process Group ‘ProjectA’. Therefore, only ProjectA component is visible and accessible to user1 and not other Process Groups and processors and not even some of the GUI elements.

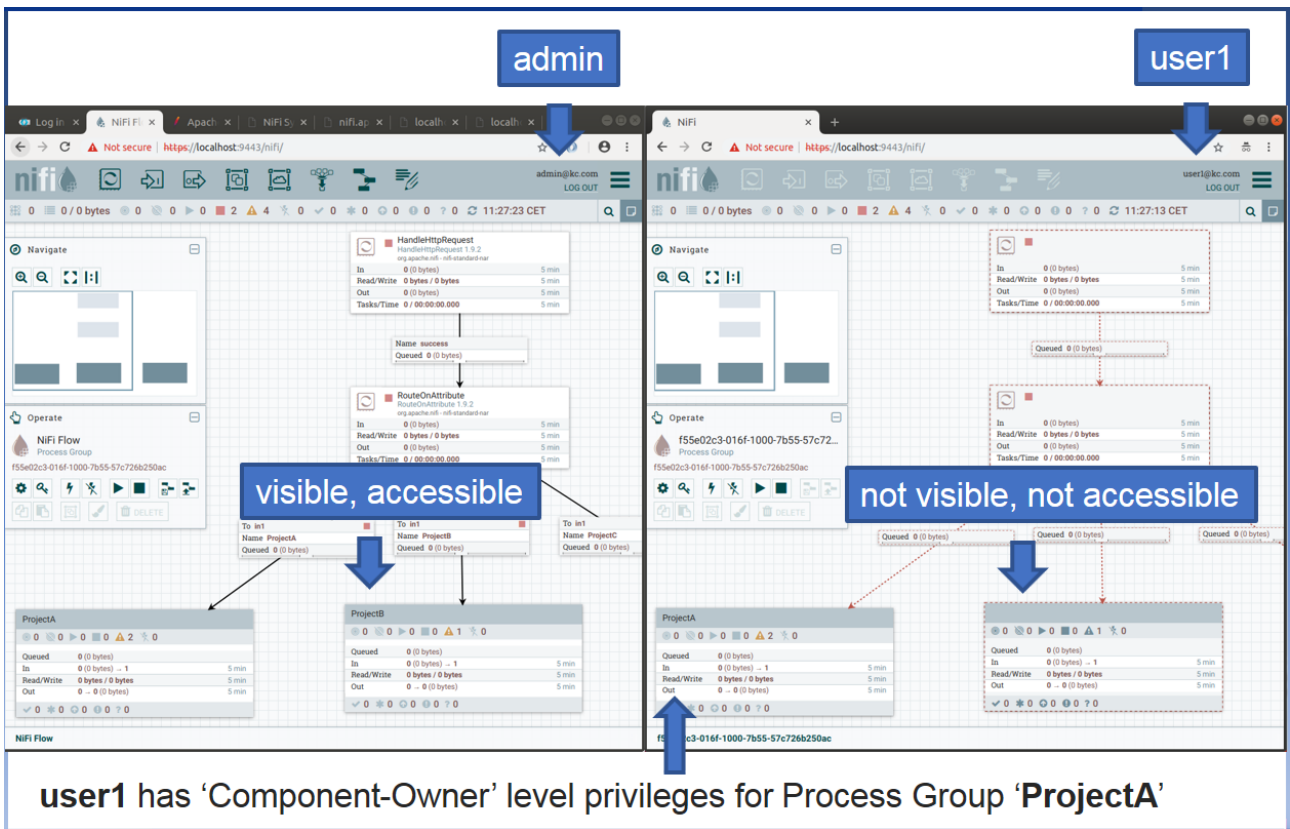


Figure 48. NiFi Multi-Tenant Authorization

This completes the Multi-Tenant Authorization of NiFi and its GUI can be used to create and execute Integration Flows.

## Securing Integration Flow API Endpoints

The Integration Flows often expose new endpoints which are proxy or interoperability-proxy<sup>2</sup> endpoints for external services and access to these endpoints needs to be secured. As shown in Figure 49, the access to these endpoints is secured with the API Security Gateway. When these endpoints are registered in the Service Registry, the API Security Gateway automatically creates proxy endpoints for these endpoints and the service consumers are required to invoke these proxy endpoints in API Security Gateway instead. This completes the security integration of Data Spine's Integration Flow Engine – Apache NiFi.

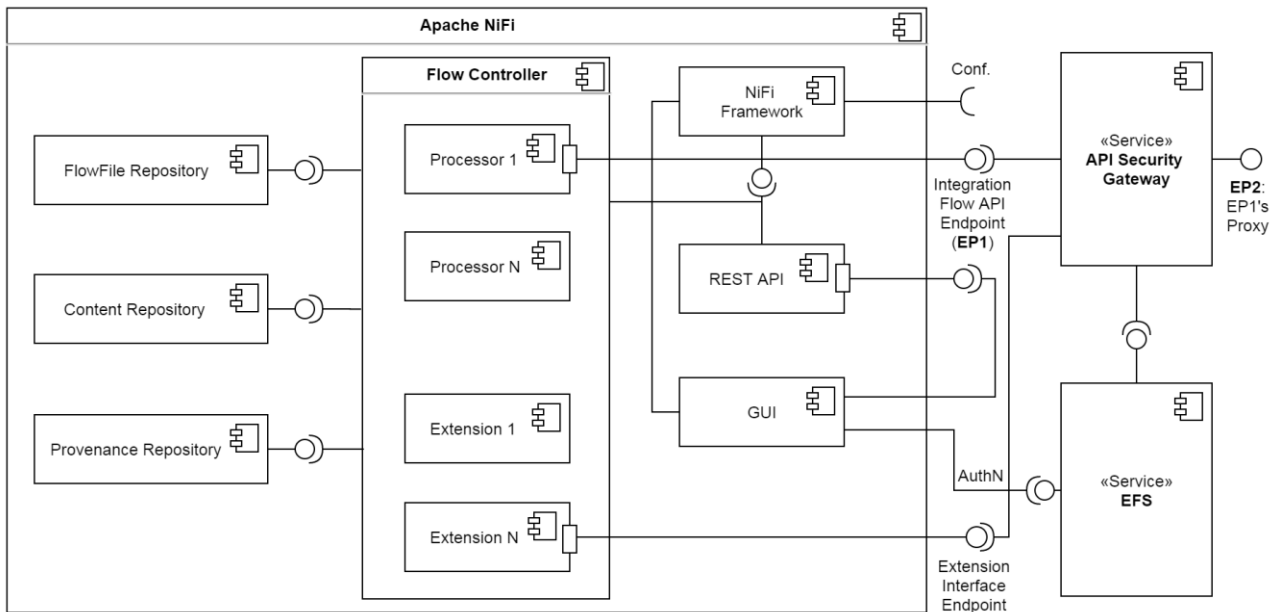


Figure 49. Data Spine NiFi Security Integration

The design-time configuration of Data Spine Integration Flow Engine's instance NiFi is now complete.

### 2.5.2.3 Operation

Figure 50 shows a basic Integration Flow that provides an interoperability-proxy endpoint for an external endpoint. All the processors can be started with selecting them and pressing start (▶) button on Operate Palette. The HandleHTTPRequest processor Starts an HTTP Server and listens for HTTP Requests. Once it receives a request, it creates a FlowFile and forwards it to 'success' relationship whose other end is connected with the InvokeHTTP processor. This processor invokes the preconfigured external endpoint and if it receives a response, it forwards it to Jolt Data Transformation Processor which does the data transformation and delegates the outcome to HandleHttpResponse processor which returns the response to the caller, else to HandleHttpResponse processor which returns the response/error to the caller.

<sup>2</sup> same data is made available over a new endpoint/topic but adhering to a different data model (and/or format) and/or over different protocol



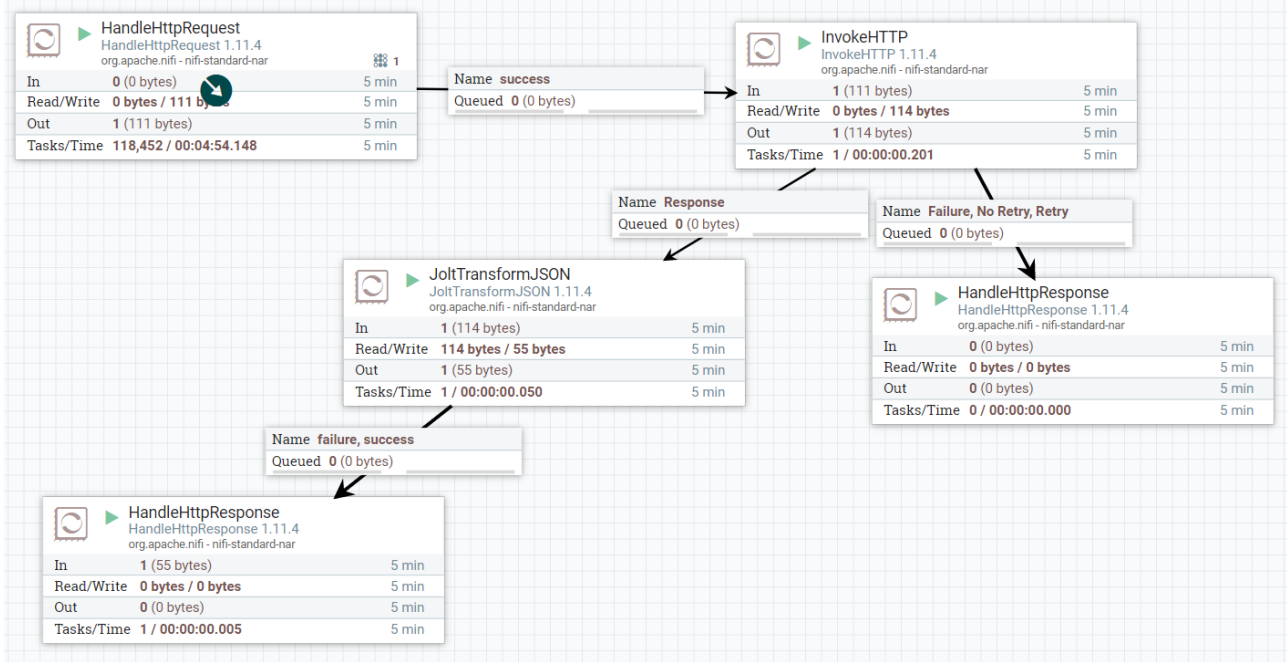


Figure 50. Integration Flow Example

### 2.5.3 API Security Gateway

The API Security Gateway (ASG) acts as a border gateway ahead of all HTTP-based API calls to the integration flows in the Data Spine. Its role is to enforce security policies on the service calls. In EFPF, ASG is implemented using Apache APISIX and the asg-importer [ASGI22] microservice. Apache APISIX offers the following features:

- **Speed:** As the ASG will proxy calls from Data Spine to other platforms in the ecosystem, the latency for the calls should be minimized
- **Custom plugins:** The ASG should depend on minimal code/configuration for the development of custom security plugins
- **License:** A permissive license is preferred (Apache / MIT) for the implementation of the ASG; and
- **MQTT support.**

Figure 51 compares core features of Apache APISIX and Kong 2.0 Open-Source API Gateway [KON20].

Features	Apache APISIX	Kong 2.0
<b>Technology</b>	Nginx, etcd (for service discovery)	Nginx, Postgres
<b>Latency</b>	0.2 ms	2 ms
<b>Plugin setup</b>	Minimal effort	Multiple file changes
<b>Plugin hot-loading</b>	Yes	No
<b>MQTT support</b>	Yes	No
<b>License</b>	Apache 2.0	Apache 2.0

Figure 51. Comparison of Apache APISIX and Kong 2.0 API Gateways

Due to the above advantages the ASG in EFPF is implemented using Apache APISIX. APISIX is a cloud-based microservices API gateway that delivers the ultimate performance, security, open-source, and scalable platform for APIs and microservices. It can be used as a traffic entrance to process all business data, including dynamic routing, dynamic upstream, dynamic certificates, A/B testing, canary release, blue-green deployment, limit rate, defence against malicious attacks, metrics, monitoring alarms, service observability, service governance, etc. Compared with the traditional API gateways, APISIX has dynamic routing and plug-in hot loading, which is especially suitable for API management under microservice systems.

The asg-importer microservice was additionally developed to scan the Service Registry for new service registrations or updates and create secure proxy endpoints for those services in Apache APISIX. This empowers the users to secure the HTTP-based APIs exposed by their integration flows, without the intervention of an administrator.

### 2.5.3.1 Architecture and Interfaces

Figure 52 shows the workflow diagram of the API Security Gateway.

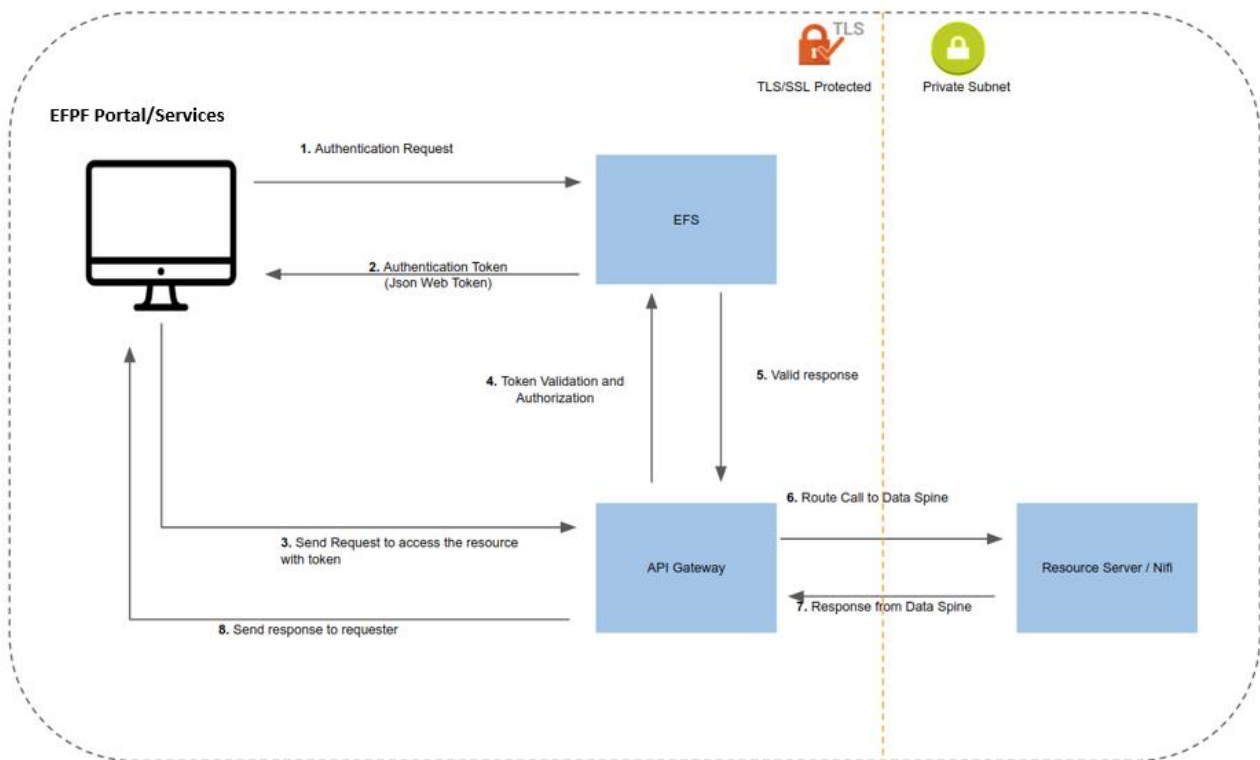


Figure 52. General Communication Workflow involving the API Security Gateway

Moreover, Figure 52 shows the basic communication workflow around the ASG (note, ASG is represented by the API Gateway box in the figure).

The ASG automatically creates the routes for services that are based on the Service Registry from the Data Spine (note, the Data Spine is represented by the Resource Server/NiFi box in the figure). Any routes which are not exposed to the ASG will result in a 404 response (“Not found”).

The ASG has two custom plugins for security enforcement, Open ID Connect plugin and Policy Enforcement plugin dealing with the corresponding services.

**The Open ID Connect plugin** provides token introspection. The token introspection can be done either through communicating with the identity server or importing the public key of the token. This plugin verifies if the token is generated from the EFS identity server and does basic authorization via JSON web token (JWT) scopes.

**The Policy Enforcement plugin** provides additional security for the routes defined by the ASG. The identity server allows to define policies based on the user's role or user's attributes. This plugin communicates with the policy engine to allow or reject the call based on user's privileges.

### 2.5.3.2 Configuration

The configuration of the ASG involves three main steps.

1. **Configure connecting with Service Registry:** Service registry contains the services registered in the EFPF platform. The ASG performs a periodic scanning of the services to create secure proxy routes for the services.
2. **Configure Open ID connect Plugin:** Open ID connect plugin performs introspection of the EFS token. The ASG should be configured to communicate with the EFS to validate the tokens in each API call.
3. **Configure Policy Enforcement Plugin:** This is a complementary plugin for the Identity Server to enforce policies to routes exposed via the ASG. This plugin should be configured by stating the upstream resource of the route and the scope of the operation.

### 2.5.3.3 Operation

Apache APISIX and asg-importer are available as Docker containers for cloud-native deployments. The ASG comes with the admin dashboard to monitor the operations of the ASG. The routes will be automatically configured when enabling the connectivity to the Service Registry. The routes will be dynamically modified as and when the Service Registry is modified with new services. The access logs of the ASG can be exported via using the HTTP or Kafka logger for monitoring.

Additionally, more plugins can be enabled to ensure smooth operation of the ASG, such as IP block listing and request rate limiting plugins.

## 2.5.4 Service Registry

LinkSmart Service Catalog [LSC20] was chosen to realise the Service Registry component of Data Spine. Service Catalog is the entry point for Web services, and the Service Registration Tool (SRT) is an easy-to-use GUI for the Service Registry. The main functionality of the Service Registry covers the lifecycle management of services, i.e., the registration, viewing, updating and deregistration of services' metadata. In addition, it supports browsing of the registered service entries and provides a service filtering functionality that can be used by service consumers to search services by known capabilities.

Figure 53 shows the flow of service metadata. Services that register themselves can be discovered by other components within or beyond the local network.

The LinkSmart Service Catalog was enhanced further to fulfil the design requirements for Service Registry mentioned in Section 2.4.1.4.

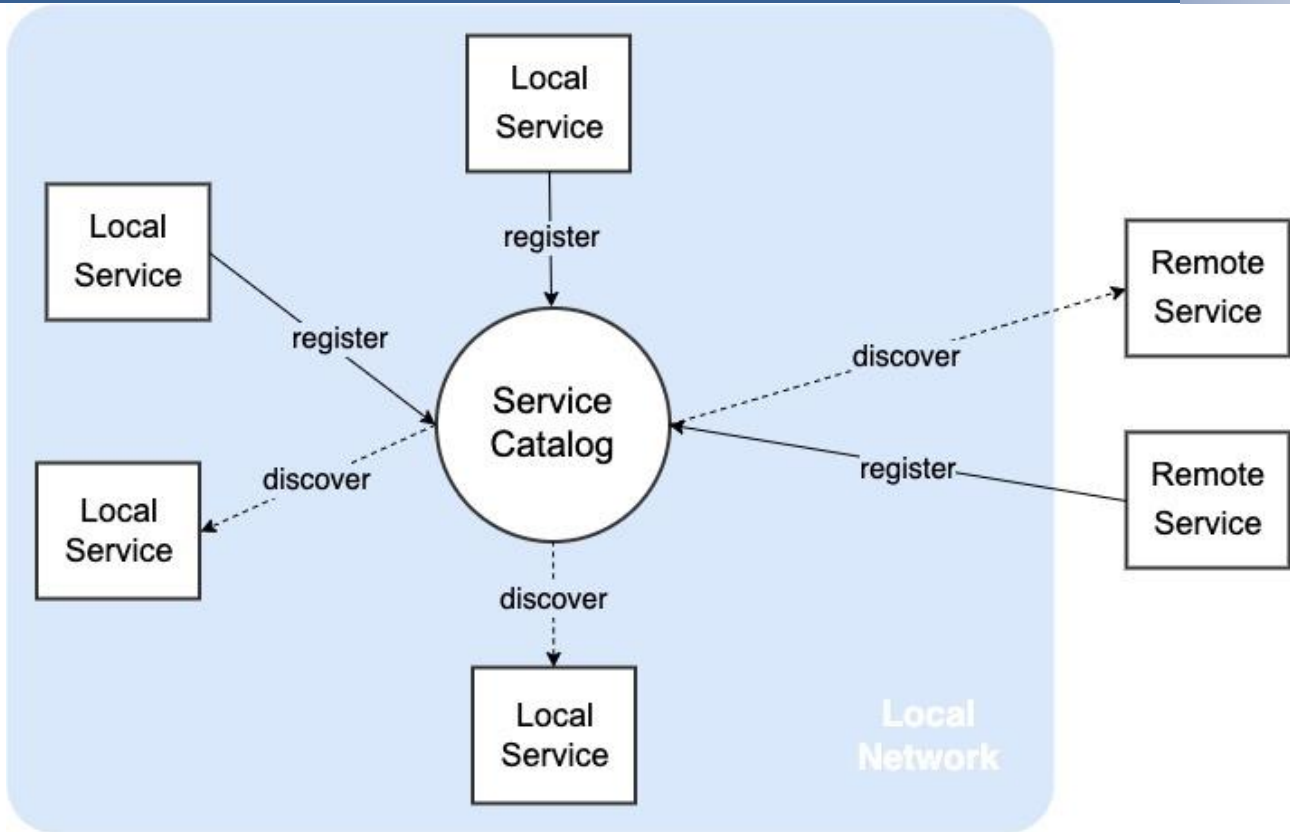


Figure 53. LinkSmart Service Catalog

### 2.5.4.1 Architecture and Interfaces

The schema of LinkSmart Service Catalog was updated as per the design requirements identified in Section 2.4.1.4. The new schema, depicted in Figure 54, is capable of managing metadata for synchronous (request-response) as well as asynchronous (publish-subscribe) type of services. This schema can be extended to include additional metadata for the entire service or for a specific API. E.g., Figure 55 shows an extended schema for the Service Registry to include certain attributes applicable to asynchronous services such as Factory Connectors/Gateways.

```
{
  "id": "string",
  "type": "string",
  "title": "string",
  "description": "string",
  "meta": {},
  "apis": [{
    "id": "string",
    "title": "string",
    "description": "string",
    "protocol": "<protocol - e.g., MQTT>",
    "url": "<base url of the API>",
    "spec": {
      "mediaType": "<mediaType type of the API Spec document>",
      "url": "<url to external API Spec document>",
      "schema": {}
    },
    "meta": {}
  }],
  "doc": "string",
  "ttl": 864000,
}
```

```

"created": "2020-06-05T15:46:36.793Z",
"updated": "2020-06-05T15:46:36.793Z",
"expires": "2020-06-06T15:46:36.793Z"
}

```

Figure 54. Service Description Schema of the Service Registry

```

{
  "id": "string",
  "type": "string",
  "title": "string",
  "description": "string",
  "meta": {
    "async": {
      "location": {
        "description": "string",
        "latitude": "string",
        "longitude": "string"
      },
      "manufacturer": "string"
    }
  }
  "apis": [{
    "id": "string",
    "title": "string",
    "description": "string",
    "protocol": "<protocol - e.g., MQTT>",
    "url": "<base url of the API>",
    "spec": {
      "mediaType": "<mediaType type of the API Spec document>",
      "url": "<url to external API Spec document>",
      "schema": {}
    },
    "meta": {}
  }],
  "doc": "string",
  "ttl": 864000,
  "created": "2020-06-05T15:46:36.793Z",
  "updated": "2020-06-05T15:46:36.793Z",
  "expires": "2020-06-06T15:46:36.793Z"
}

```

Figure 55. Extended Service Description Schema of the Service Registry

The architecture of the Service Registry and the SRT are illustrated in Figure 56. The Service Registry provides an HTTP REST API for Lifecycle Management and Discovery of Services, an MQTT Service Registration/De-registration API through the Data Spine Message Bus and an MQTT Service Status Announcements API for also through the Data Spine Message Bus. The Service Registry can be configured by using a JSON configuration file and it uses LevelDB on-disk key-value store to persist data. Finally, the access to Service Registry's APIs is secured by using the API Security Gateway and EFS.

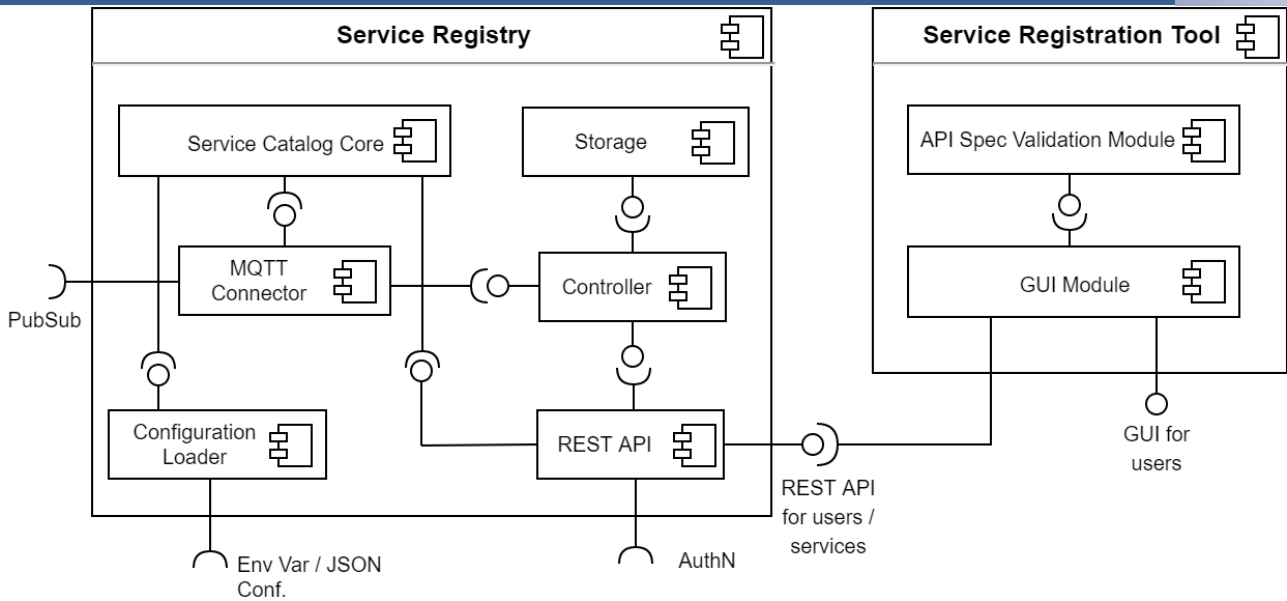


Figure 56. Architecture of the Service Registry and the Service Registration Tool

**SR’s HTTP REST API for Lifecycle Management and Discovery of Services:**

Figure 57 provides an insight into the HTTP REST API of Service Registry. As illustrated, create, read, update, and delete operations can be performed on the Service object in a RESTful manner. The service filtering API endpoint enables service filtering based on a given path, operator, and value.

Examples:

- Filter all services belonging to PlatformX (convention for 'type' followed: <platform-name>.<service-type>): /type/prefix/PlatformX
- Filter all services that have MQTT API(s): /apis.protocol>equals/MQTT
- Filter all services based on address meta field: /meta.address/contains/Bonn

REST Endpoint	HTTP Method	Description
/	GET	Retrieves API index.
/{id}	POST	Creates new 'Service' object with a random UUID (Universally Unique Identifier).
/{id}	GET	Retrieves a 'Service' object
/{id}	PUT	Updates the existing 'Service' or creates a new one (with the provided ID)
/{id}	DELETE	Deletes the 'Service'
/{jsonpath}/{operator}/{value}	GET	Service filtering API endpoint

Figure 57. Service Registry HTTP REST API

Figure 58 shows the data model of the Service Registry.

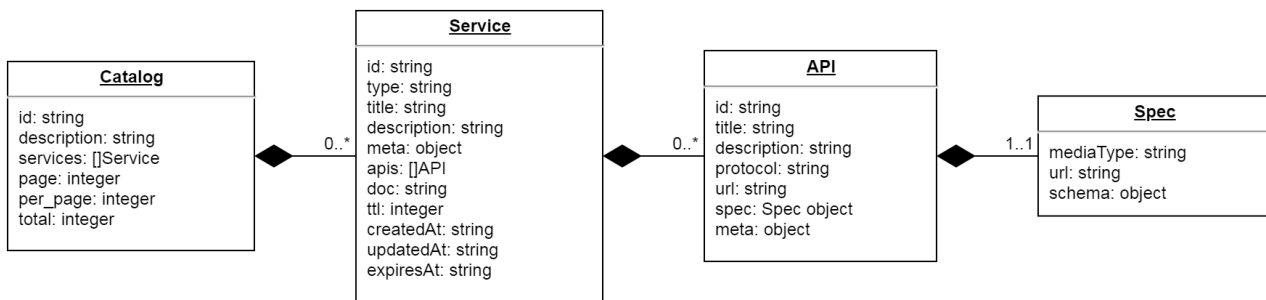


Figure 58. Data Model of the Service Registry

The attributes are described below:

The Catalog object consists of:

- id: unique id of the catalog
- description: a friendly name or description of the service
- services: an array of Service objects
- page: the current page in catalog
- per\_page: number of items in each page
- total: total number of registered services

A Service object consists of:

- id: unique id of the service
- type: type of the service, preferably in the form <platform>.<service-type>  
E.g., “composition.marketplace-service”
- title: human-readable name of the service
- description: human-readable description of the service
- meta: a hash-map for optional meta-information
- apis: an array of API objects specifying the service’s APIs
- doc: url to service documentation
- ttl: time in Seconds after which the service should be removed from the SR, unless it is updated within the ttl timeframe. ttl serves as a keepalive mechanism to detect failures/unavailability of registered services. I.e., as per the current setting, the registered services are obliged to update themselves within the ttl. If they fail to do so, they are concluded to be unavailable. The Service Provider based on the availability requirements of his/her service should determine the most suitable value for the ttl of a particular service.
- createdAt: RFC3339 time of service creation
- updatedAt: RFC3339 time in which the service was lastly updated
- expiresAt: RFC3339 time in which the service expires and is removed from the SR

An API object consists of:

- id: unique id of the API

- title: human-readable name of the API
- description: human-readable description of the API
- protocol: the communication protocol used by the API (E.g., HTTP, MQTT, etc.)
- url: A URL to the server/target host (E.g., <https://services.example.com>, <tcp://broker.example.com:1883>, etc.) as defined by 'Server Object' in OpenAPI/AsyncAPI specifications
- spec: the specification of the API as per the Open API Specification (Swagger) standard for synchronous (Request-Response) services or the AsyncAPI Specification standard for asynchronous (PubSub) services
- meta: a hash-map for optional meta-information

A Spec object consists of:

- mediaType: The media type for the spec url below
  1. For OpenAPI/Swagger Spec: application/vnd.oai.openapi;version=3.0 (YAML variant) or application/vnd.oai.openapi+json;version=3.0 (JSON only variant)
  2. For AsyncAPI Spec: application/vnd.aai.asyncapi;version=2.0.0 or application/vnd.aai.asyncapi+yaml;version=2.0.0 (YAML variant) or application/vnd.aai.asyncapi+json;version=2.0.0 (JSON only variant)
- url: url to external spec document
- schema: the JSON object for the spec can be added here in case if the external document is not available. In case both are present, the spec in the url takes precedence

### **SRT's GUI for Service Registration:**

To make it easy for human users to registration, the SRT provides a GUI as illustrated in Figure 59 and Figure 60.



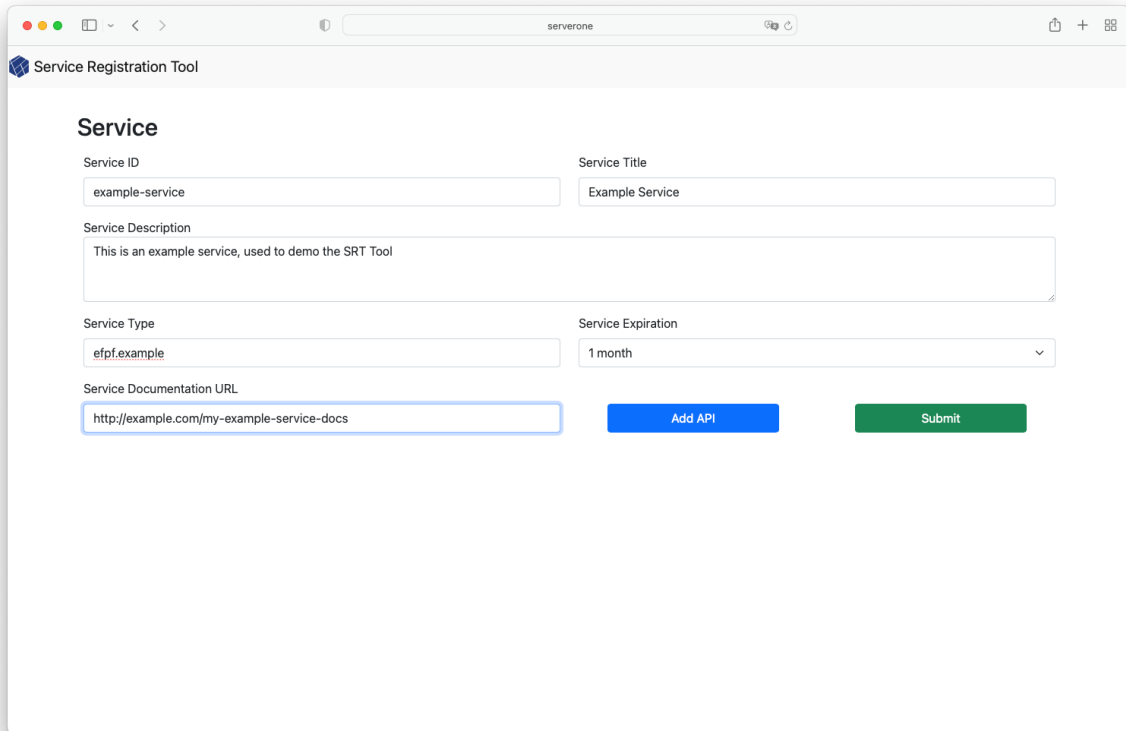


Figure 59. Service Registration Tool's GUI: Register a Service

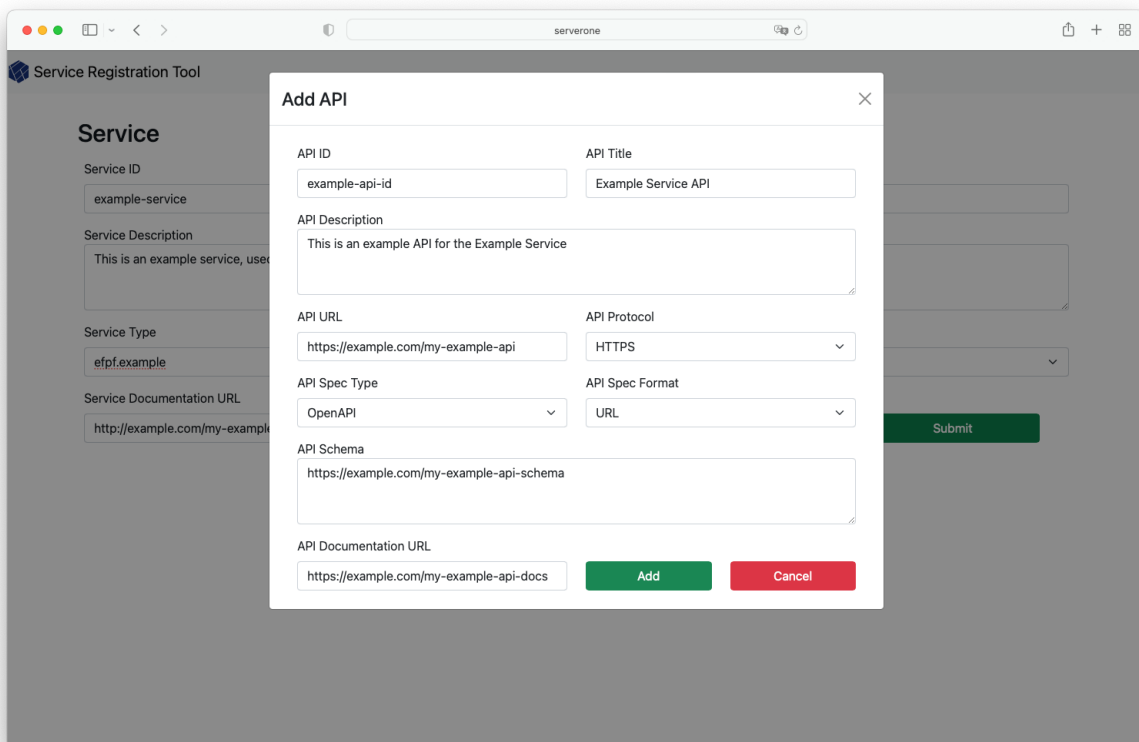


Figure 60. Service Registration Tool's GUI: Register Add an API

## SR's MQTT Service Registration/De-registration API:

Service Registry (SR) also supports MQTT for service registration, updates, and de-registration.

**Registration:** Service registration is similar to PUT method of REST API. Here, a service uses a pre-configured topic defined in the config file (see `commonRegTopics` and `regTopics`) for publishing the message.

Example:

```
mosquitto_pub -h localhost -p 1883 -t 'sr/v3/cud/reg/id1' -f
./service_object.json
```

Here, the `service_object.json` file contains the service (JSON) object.

**Deregistration:** The will message of the registered service is used to de-register it from the SR. The will topic(s) are defined in the config file (see `commonWillTopics` and `willTopics`).

Example:

```
mosquitto_pub -h localhost -p 1883 -t 'sr/v3/cud/dereg/id1' -m 'deleting service
with id: id1'
```

## SR's MQTT Service Status Announcements API

Service Registry announces the service registration status via MQTT using retain messages.

The message topics follow following patterns:

- `<topicPrefix>/<service type>/<service_id>/alive:` (Retained message) The body contains service description of alive service
- `<topicPrefix>/<service type>/<service_id>/dead:` (Not retained message) The body contains service description of alive service

The retained messages are removed whenever service de-registers. The 'topicPrefix' can be configured via the config file.

Examples:

```
mosquitto_sub -h localhost -p 1883 -t 'sr/v3/announcement/efpf.data-spine-
service/eb647488-a53b-4223-89ef-63ae2ce826ae/alive'
```

```
mosquitto_sub -h localhost -p 1883 -t 'sr/v3/announcement/efpf.data-spine-
service/eb647488-a53b-4223-89ef-63ae2ce826ae/dead'
```

```
mosquitto_sub -h localhost -p 1883 -t 'sr/v3/announcement/efpf.data-spine-
service/+/alive'
```

```
mosquitto_sub -h localhost -p 1883 -t 'sr/v3/announcement/efpf.data-spine-
service/+/dead'
```

```
mosquitto_sub -h localhost -p 1883 -t 'sr/v3/announcement/+/+/alive'
```

```
mosquitto_sub -h localhost -p 1883 -t 'sr/v3/announcement/+/+/dead'
```

### 2.5.4.2 Configuration

The Service Catalog (SC) consumes a JSON configuration file which is shown in Figure 61.

```
{
  "description": "string",
  "dnssdEnabled": "boolean",
  "storage": {
    "type": "string",
    "dsn": "string"
  },
  "http" : {
    "bindAddr": "string",
    "bindPort": "int"
  },
  "mqtt":{
    "client": {
      "brokerID": "string",
      "brokerURI":"string",
      "regTopics": ["string"],
      "willTopics": ["string"],
      "qos": "int",
      "username": "",
      "password": ""
    },
    "additionalClients": [],
    "commonRegTopics": ["string"],
    "commonWillTopics": ["string"],
    "topicPrefix": "string"
  },
  "auth": {
    "enabled": "bool",
    "provider": "string",
    "providerURL": "string",
    "serviceID": "string",
    "basicEnabled": "bool",
    "authorization": {}
  }
}
```

Figure 61: LinkSmart Service Catalog Configuration

The configuration file is primarily used to specify the configuration details of MQTT broker, the storage, and the optional authentication provider. The attributes are explained below:

- **description** is a human-readable description for the SC
- **dnssdEnabled** is a flag enabling DNS-SD advertisement of the Catalog on the network
- **storage** is the configuration of the storage backend
  - **type** is the type of the backend (supported backends are memory and LevelDB)

- **dsn** is the Data Source Name for storage backend (ignored for memory, "file:///path/to/ldb" for leveldb)
- **http** is the configuration of HTTP API
  - **bindAddr** is the bind address which the server listens on
  - **bindPort** is the bind port
- **mqtt** is the configuration of MQTT API
  - **client** is the configuration for the main MQTT client
    - **brokerID** is the service ID of the broker (Optional)
    - **brokerURI** is the URL of the broker
    - **regTopics** is an array of topics that the client should subscribe to for addition/update of services
      - Example: "regTopics": ["topic/reg/+"]
      - While publishing a service registration message over this topic, '+' should be replaced by id of the service to be added/updated. id passed in message payload takes precedence over id in the topic
        - E.g., `mosquitto_pub -h localhost -p 1883 -t 'topic/reg/custom_id1' -f ./service_object.json`
    - **willTopics** is an array of will topics that the client should subscribe to for removal of services (Optional in case TTL is used for registration)
      - Example: "willTopics": ["topic/dereg/+"]
      - While publishing a service deregistration message over this topic, '+' should be replaced by id of the service to be removed
        - E.g., `mosquitto_pub -h localhost -p 1883 -t 'topic/dereg/custom_id1' -m 'something'`
  - **qos** is the MQTT Quality of Service (QoS) for all reg and will topics
  - **username** is username for MQTT client
  - **password** is the password for MQTT client
- **additionalClients** is an array of additional brokers objects.
- **commonRegTopics** is an array of topics that all clients should subscribe to for addition/update of services (Optional)
  - Example: same as the example for 'regTopics' above
- **commonWillTopics** is an array of will topic that the client should subscribe to for removal of services (Optional in case commonRegTopics not used or TTL is used for registration)
  - Example: same as the example for 'willTopics' above
- **topicPrefix** is the string describing the prefix of service announcement topics
- **auth** is the Authentication configuration
  - **enabled** is a boolean flag enabling/disabling the authentication
  - **provider** is the name of a supported auth provider

- **providerURL** is the URL of the auth provider endpoint
- **serviceID** is the ID of the service in the authentication provider (used for validating auth tokens provided by the clients)
- **basicEnabled** is a boolean flag enabling/disabling the Basic Authentication
- **authorization** - optional, see authorization configuration

All configuration fields (except for arrays of objects) can be overridden using environment variables. E.g.: `SC_STORAGE_TYPE=leveldb`

Having configured the Service Catalog in this way, the next step is to secure access to its APIs. To secure the REST API, proxy endpoints are configured for its REST endpoints in the API Security Gateway and access policies are defined in the EFS. The access to its MQTT APIs is secured by the policies configured in EFS; however, the access is not enforced using the API Security Gateway but using the Message Bus itself. The SC subscribes to or publishes to the Message Bus using keys that are issued by the Message Bus when the corresponding topics are created by calling Message Bus's HTTP API. The service providers who want to register their services by publishing through the MQTT API or the users who want to subscribe to the service status announcements need to obtain the respective keys by calling Message Bus's HTTP API through the API Security Gateway.

### 2.5.4.3 Operation

When a call is made to an endpoint of Service Catalog's proxy API in the API Security Gateway (ASG) with an EFPF token, the API Security Gateway checks for authentication and authorization with the EFS. Upon receiving a positive reply from the EFS, ASG invokes the corresponding endpoint of the Service Catalog. The Service Catalog processes the call and returns the reply to its caller, the ASG. The ASG then forwards this reply to the original caller.

For registrations through MQTT API, the user/client publishes the service object over the preconfigured registration topic to the Message Bus with the given key, the Message Bus verifies the key for authentication and authorization and once verified, the message is published. The Service Catalog receives this message from the Message Bus, and the service is registered. The Service Catalog publishes the service status announcements to the Message Bus over the preconfigured topics using the given keys, the users/clients need to subscribe to these topics using the keys issued by the Message Bus.

### 2.5.5 Message Bus

RabbitMQ [RMQ20] Message Broker satisfies the design requirements the Data Spine Message Bus enlisted in Section 2.4.1.5. RabbitMQ is a message broker or message-oriented middleware that implements AMQP (Advanced Message Queuing Protocol) 0-9-1. It is one of the most popular and most widely deployed open-source message broker. Many partners involved in the EFPF project, especially the partners that provide Factory Connector/Gateway solutions had first-hand experience with using RabbitMQ and also RabbitMQ is being used for supporting asynchronous communication in COMPOSITION and SMECluster platforms. Therefore, RabbitMQ was first chosen for experimentation in the EFPF project and with a positive first-hand experience, was selected to realise the Message Bus.

Some of the features of RabbitMQ and their significance in EFPF are explored below:

- **Support for protocols**

- In EFPF, we primarily make use of MQTT, MQTTS and AMQP (0-9-1)
- RabbitMQ supports AMQP 0-9-1 inherently and AMQP 1.0 and MQTT/MQTTS via plugins
- It also supports STOMP, AMQP 1.0, HTTP and WebSockets
- **Deployment**
  - Docker container has been selected as a deployment unit in EFPF
  - RabbitMQ's Docker images are made available with each release
- **Management and Monitoring**
  - RabbitMQ provides a management GUI and an HTTP-based API for administration, management and monitoring of channels/topics, users, dataflow stats, etc. via a plugin.
  - In addition, RabbitMQ also provides management command line tools such as 'rabbitmqadmin' and 'rabbitmqctl' that enable easy administration.
- **Identity and Access Management**
  - RabbitMQ supports multiple SASL (Simple Authentication and Security Layer) authentication mechanisms out of which, three are built into the server - PLAIN, AMQPLAIN and RABBIT-CR-DEMO and one 'EXTERNAL' is supported via a plugin. More such authentication mechanisms are supported via plugins. In essence, RabbitMQ supports widely used password-based, token-based and client certificates-based authentication.
  - RabbitMQ also supports multi/tenant authorization with the help of 'virtual hosts' which enable logical grouping and separation of resources such as connections, exchanges, queues, bindings, user permissions, policies, etc.
- **Performance and Scalability**
  - RabbitMQ supports clustered deployment for high availability and throughput.
- **Extensibility, Tools & Plugins**
  - RabbitMQ's flexible plug-in-approach supports extension of functionality through the use of plugins.
  - It provides official client libraries for many programming languages and also various developer tools for supporting frameworks such as the Spring Framework.
- **Documentation**
  - The documentation provided by the RabbitMQ developers and community is comprehensive and it covers tutorials and guides from installation, setup, and usage of RabbitMQ to developments of new plugins.

#### 2.5.5.1 Architecture and Interfaces

RabbitMQ is an implementation of the AMQP 0-9-1 protocol. The AMQP 0-9-1 model followed by RabbitMQ is shown in Figure 62. The model defines messaging brokers that act as middleware, publishers/producers that publish messages to the messaging brokers and these messaging brokers route these messages to the consumers/subscribers. The

messages are published to 'exchanges' component of RabbitMQ, the exchanges then route the messages to 'queues' based on routing rules called 'bindings'. RabbitMQ then pushes these messages in the queues to the subscribers or these messages can even be pulled on demand by the subscribers. RabbitMQ also uses message acknowledgments to ensure reliable exchange of messages over unreliable networks. The subscriber application notifies the broker as soon as it receives a message and then only the broker will remove those messages from a queue.

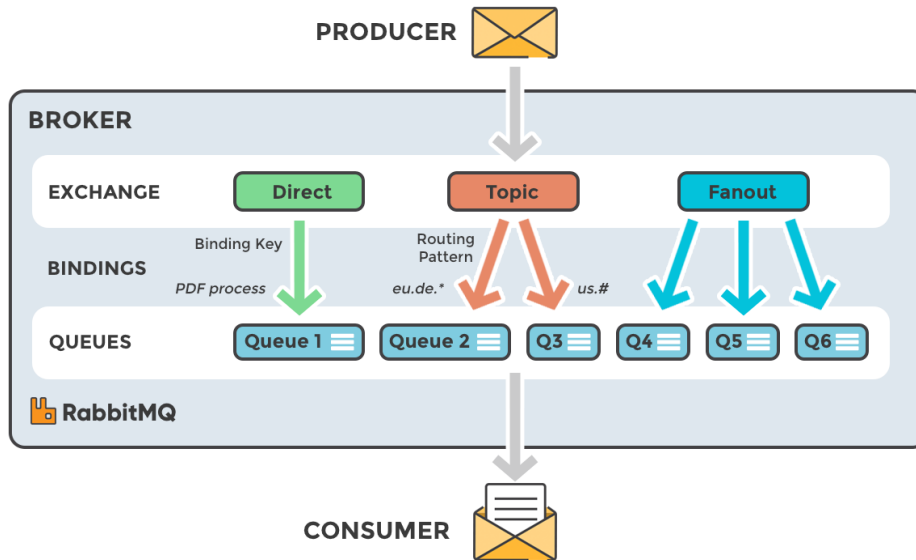


Figure 62. AMQP 0-9-1 Model Followed by RabbitMQ [CAQ20]

These exchanges are of different types and the type of an exchange is specified when it is created:

In 'Direct' type of exchanges, the message is routed to the queue whose binding key exactly matches with the routing key of the message. In 'Topic' type of exchanges, a wildcard match between the routing key of the message and the routing pattern specified in the binding is done by the exchange. In 'Fanout' type of exchange, messages are routed to all of the queues bound to the exchange. In 'Header' type of exchange, the messages are routed based on message header attributes.

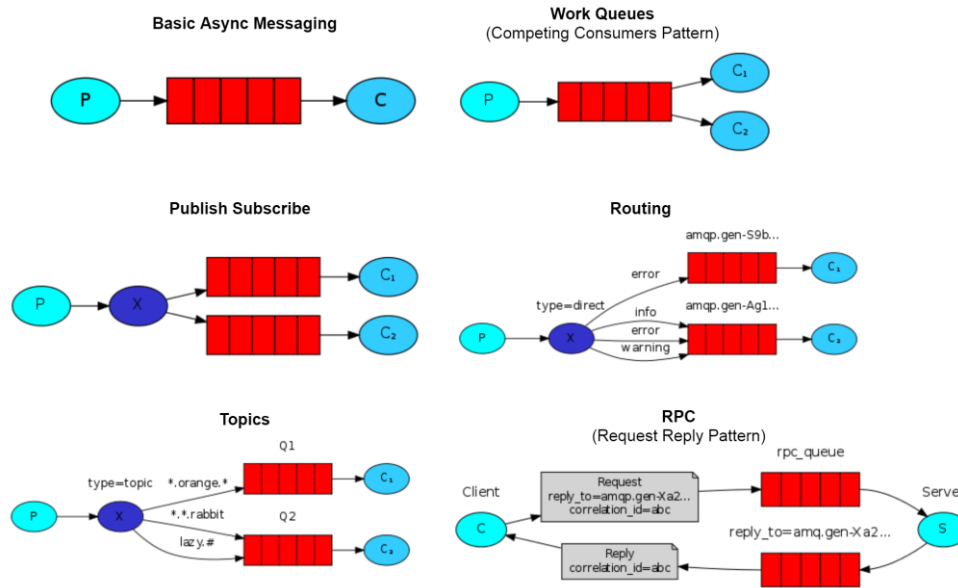


Figure 63. RabbitMQ Messaging Patterns [RMP20]

RabbitMQ supports several messaging patterns as shown in Figure 63. It supports the basic asynchronous pattern where it provides a queue to which a publisher can publish a message and the subscriber can receive it through the broker’s queue. The Work Queues or Competing Consumer pattern enables distribution of messages/tasks among several consumers/workers. The Publish Subscribe pattern enables the dispatch of messages to many consumers at once. The Routing pattern enables the selective routing of messages to different queues. The Topic pattern enables routing of messages to different queues based on patterns/topics. RabbitMQ also supports the RPC or Request-Response pattern that supports call-backs and also the Publisher Confirms pattern to enable reliable publishing of messages.

RabbitMQ offers multiple command line tools that provide CLIs for service management, general operator tasks, diagnostics and health checking, plugin-management, maintenance tasks on queues and related to upgrades, and, for management and monitoring of RabbitMQ nodes and clusters. The primary RabbitMQ CLI tool ‘rabbitmqctl’ provides an interface for managing RabbitMQ nodes and clusters. Its CLI supports user management which provides commands for adding users, authenticating users, updating passwords, listing users, setting user tags, etc. The CLI also supports many other functionalities such as access control, monitoring, observability and health checks, management of runtime parameters and policies, management of virtual hosts, configuration, etc.



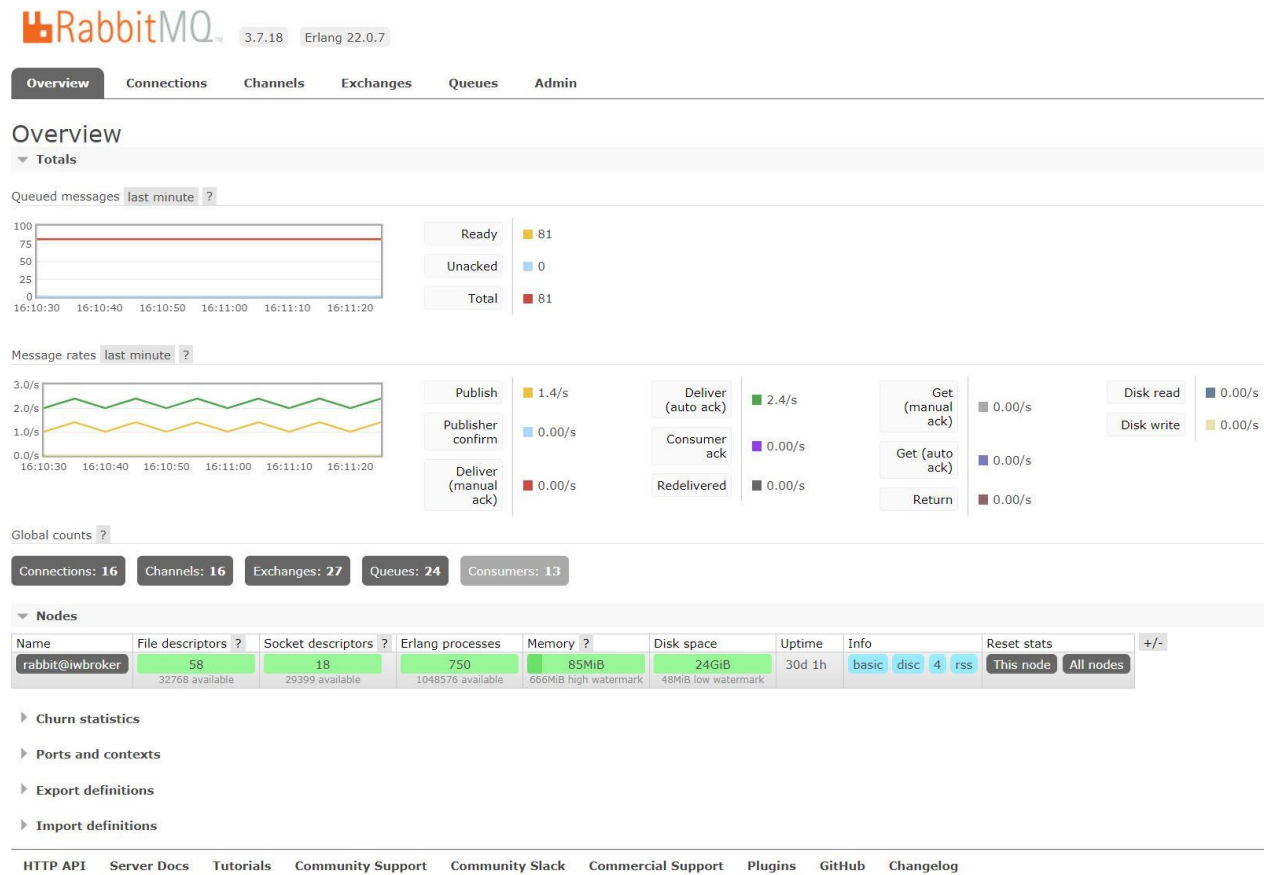


Figure 64. RabbitMQ Management GUI

RabbitMQ’s Management plugin provides an HTTP API, a Web-based GUI (Figure 64) and a CLI for management and monitoring of RabbitMQ nodes and clusters. The CLI is provided via a command line tool ‘rabbitmqadmin’. rabbitmqadmin provides capabilities to list exchanges, queues, bindings, vhosts, users, permissions, connections and channels, view overview information, declare and delete exchanges, queues, bindings, vhosts, users and permissions, publish and get messages from queues, close connections, import and export configuration, etc. The HTTP API is primarily used for monitoring and alerting purposes. It can be used to collect data related to the state of nodes, connections, channels, queues, consumers, etc. and aggregate it periodically. These stats can be used for alerting, visualization, monitoring, analysis, etc. through the provided GUI or through external monitoring systems such as Prometheus and Grafana or ELK stack. The management GUI is a single page application that consumes the HTTP API. It is intended to be used for monitoring and debugging purposes.

### 2.5.5.2 Configuration

RabbitMQ ships with built-in settings that are most commonly used across applications by default. These can be used readily for environments such as Development and Testing where performance and fine tuning is not very essential as compared to the functioning. For performance and security critical environments such as Production, RabbitMQ provides ways through which the broker server and the plugins can be configured. These configuration mechanisms for RabbitMQ include configuration files, environment variables, command line tools such as rabbitmqctl, rabbitmq-queues, rabbitmq-plugins, rabbitmq-

diagnostics, etc. The compilation of the exact configuration details for RabbitMQ to be used in the Production environment of EFPF are work in progress.

After the initial setup and deployment related configuration is done, some design-time setup activities need to be performed which are a prerequisite to RabbitMQ's operation. RabbitMQ supports multi-tenancy through the use of virtual hosts or vhosts. When the RabbitMQ server is started for the first time, it realises that the database is uninitialized or has been deleted and creates (i) a fresh database, (ii) a default '/' vhost and (iii) a default 'guest' user with full access to the '/' vhost. For security reasons, by default, the guest user can only operate from localhost. New vhosts need to be created or the default '/' vhost can also be used prior to the run-time operation. New users that can connect from remote hosts need to be created and they need to be given access permissions to specific vhosts for performing specific operations. This pre-configuration of a number of vhosts, users and user permissions is called 'seeding' operation. Such a seeding operation is typically done for production environments. This kind of seeding can be done with the help of various command line tools provided by RabbitMQ.

Once these design-time configurations on the broker side are done, the client sides can establish connections with it. The clients can make use of client libraries provided by RabbitMQ or any other tools to interface with the broker. Every client connection with the broker has an associated user and an associated vhost. The user is authenticated by the broker and its access permissions for that vhost are examined for authorization. There are two primary ways of authentication: username-password and X.509 certificates. RabbitMQ enforces access control in a layered fashion. The first layer of authorization checks whether the user has access to the specified vhost or not. The second layer is concerned with checking user access to resources such as exchanges, queues, etc. and operations to be performed on them.

### **Use of the Pub/Sub Security Service for Design-time Access Configuration**

RabbitMQ provides the administrators with a management GUI that can be used to create user accounts, vhosts, topics, assigning publish/subscribe permissions for topics to different users, etc. However, this involves intervention from the administrators, making the process slow and inefficient. In addition, the data publishers cannot be in directly control of determining who is authorized to use their data. This process was automated using the Pub/Sub Security Service. At design-time, the user can get a new user account for RabbitMQ, and/or pub/sub permissions for the topics of interest using the Pub/Sub Security Service's GUI and at runtime, the user's tool can be configured to pub/sub directly to RabbitMQ over those topics. The Pub/Sub Security Service's interfaces and its interactions with the Message Bus RabbitMQ are described further in Sections 3.2.1.4 and 3.3 respectively.

### **Configuration of Operator Policies**

In the EFPF ecosystem, the Message Bus RabbitMQ can experience a very high volume of data at times. To deal with such high data volumes, the Data Spine components including RabbitMQ are deployed in a clustered fashion using the Docker Swarm technology. In addition, to prevent running out of memory and disk space, and ensure a fair distribution and usage of available resources, RabbitMQ enables the administrators to configure operator policies. The operator policies are also useful for clearing out unused queues and message and queue indices from the persistence layer. Figure 65 illustrates the four types of operator policies that can be set and Figure 66 illustrates the scripts that can be used to set these policies for all vhosts using the rabbitmqctl CLI tool.

Virtual Host	Name	Pattern	Apply to	Definition	Priority	Clear
default_vhost	queues-expire-1-hour	.*	queues	expires: 3600000	0	<input type="button" value="Clear"/>
default_vhost	queues-max-length-10-messages	.*	queues	max-length: 10	0	<input type="button" value="Clear"/>
default_vhost	queues-max-length-bytes-50-MiB	.*	queues	max-length-bytes: 52428800	0	<input type="button" value="Clear"/>
default_vhost	queues-message-ttl-1-hour	.*	queues	message-ttl: 3600000	0	<input type="button" value="Clear"/>

Figure 65. Sample Operator Policies for RabbitMQ

```
#!/bin/bash

# set queues-expire-1-hour policy for all vhosts
rabbitmqctl --silent list_vhosts name | awk '{ print $1 }' | xargs -L1 rabbitmqctl
set_operator_policy queues-expire-1-hour ".*" '{"expires": 3600000}' --apply-to
queues -p

# set queues-max-length-10-messages policy for all vhosts
rabbitmqctl --silent list_vhosts name | awk '{ print $1 }' | xargs -L1 rabbitmqctl
set_operator_policy queues-max-length-10-messages ".*" '{"max-length": 10}' --apply-
to queues -p

# set queues-max-length-bytes-50-MiB for all vhosts
rabbitmqctl --silent list_vhosts name | awk '{ print $1 }' | xargs -L1 rabbitmqctl
set_operator_policy queues-max-length-bytes-50-MiB ".*" '{"max-length-bytes":
52428800}' --apply-to queues -p

# set queues-message-ttl-1-hour for all vhosts
rabbitmqctl --silent list_vhosts name | awk '{ print $1 }' | xargs -L1 rabbitmqctl
set_operator_policy queues-message-ttl-1-hour ".*" '{"message-ttl": 3600000}' --
apply-to queues -p
```

Figure 66. Sample Script to Set the four Operator Policies for all the Existing Vhosts using the rabbitmqctl CLI Tool

### 2.5.5.3 Operation

Once the configuration on both the RabbitMQ side and the clients' side is done, the communication can start. In the case of MQTT, after getting the necessary access permissions, the clients can directly publish and subscribe to the intended topics and can exchange data through the broker. In the case of AMQP, a few more steps are involved:

1. A TCP connection is set up between the client application and RabbitMQ where the key/credentials, connection URL, port, etc. is specified by the client.
2. The connection interface is used to create a channel in the TCP connection. Messages can now be sent or received through this channel.
3. A queue is declared or created if it does not exist.
4. An exchange is declared and setup.
5. The exchange is bound to a queue.
6. The publisher publishes message to the exchange and the consumer consumes it from the queue.

7. The channel is closed followed by the connection.

### 2.5.6 Summary

Figure 67 illustrates the technologies selected to realise the components of the Data Spine and mapping between the concepts and terminologies. The integration flows of the IFE are realised using the dataflows in Apache NiFi, the Protocol Connectors are realised using NiFi processors such as HandleHTTPRequest, InvokeHTTP, ConsumeMQTT, PublishMQTT, etc., while the Data Transformation processors are realised using NiFi processors such as Jolt, TransformXml, ExecuteScript, etc.

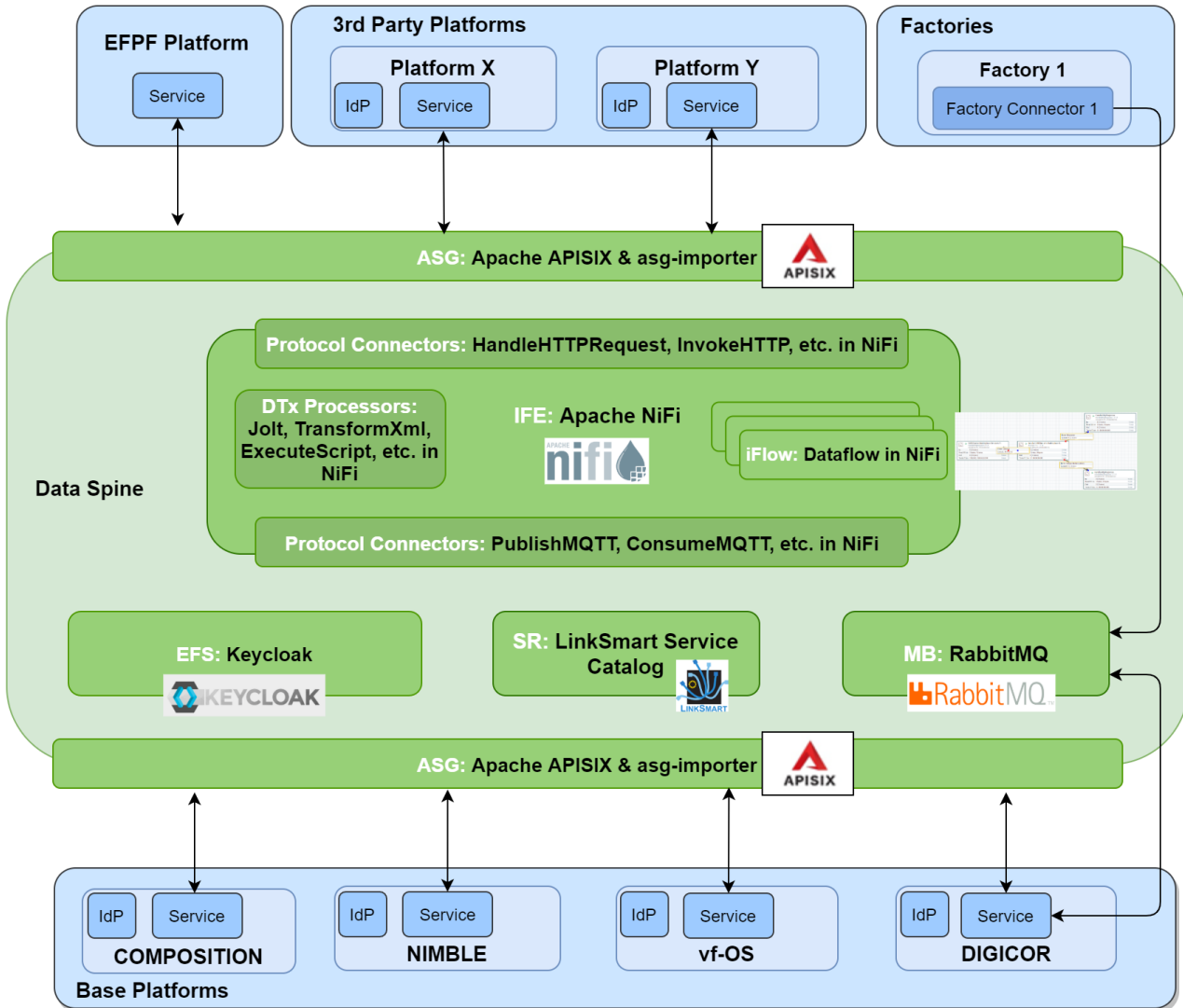


Figure 67. Data Spine Realisation & Mapping of Concepts

Figure 68 further illustrates how the Data Spine components interact with each other and how synchronous and asynchronous services interface with the components of the Data Spine. The methodology for integration of services through the Data Spine is described in detail in Section 2.8.

In this way, the Data Spine provides the necessary integration infrastructure to bridge the interoperability gaps between heterogeneous services and enables communication in the EFPF ecosystem.

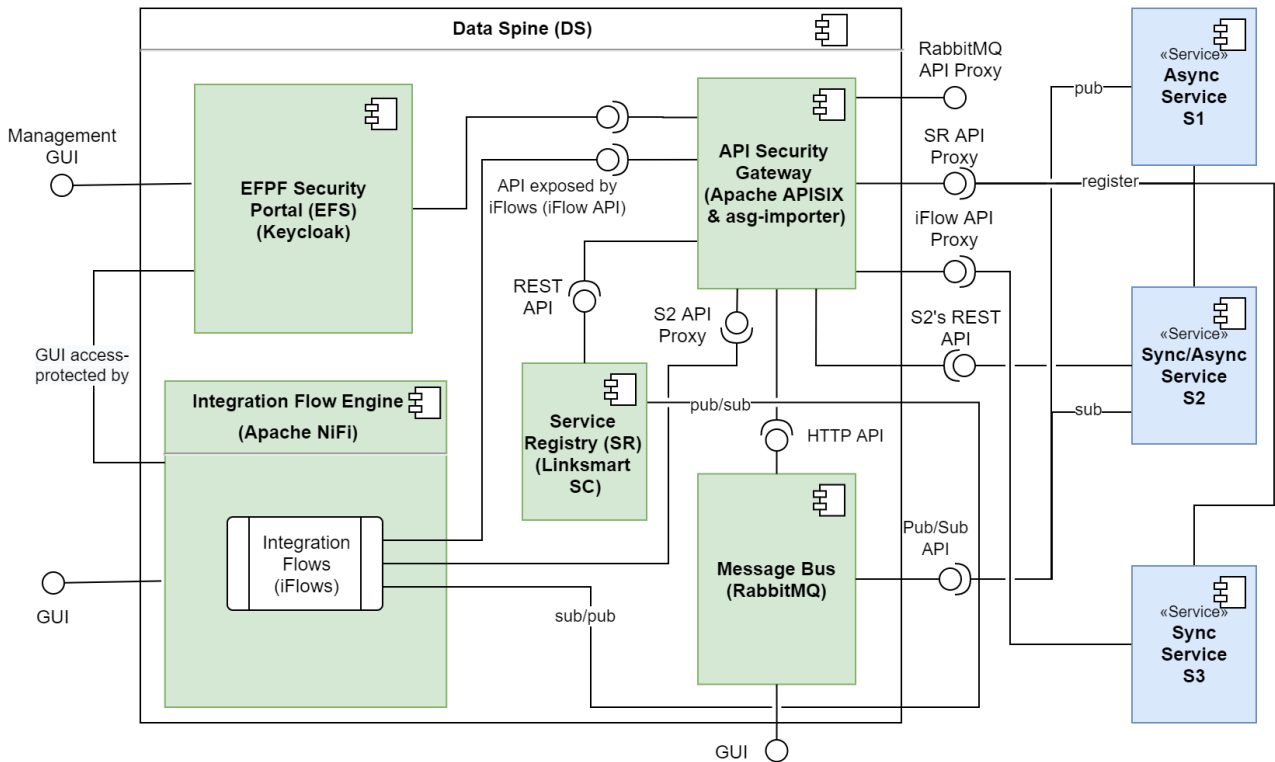


Figure 68. Interfacing of Sync and Async Services with the Data Spine Components

## 2.6 Deployment

The deployment is the process of making an application work on a specific target environment. Deployment refers to a set of activities which start from the code release (once a consistent version of a software code is ready to be executed) up to the execution beginning of the application on the target (physical or virtual) machine.

Usually, this operation is automated using deployment pipelines, a system of automated processes designed to transfer the new code modification quickly and accurately from the repository to the execution environment. During the pipeline, the code is analysed and verified against coding standards. Unit testing of the compiled binaries are performed to be sure that the released software meets the functional and performances requirement.

The pipelines can also be split in two main categories:

- **Continuous Integration (CI) Pipelines:** A set of operation in which all the developers merge code changes in a central repository. Each code change triggers an automated build and test system to provide feedbacks to the developer who made the change.
- **Continuous Delivery and Deployment (CD) Pipelines:** A set of operation to move the built code to the target environment, which can be the staging environment (to perform the final tests and make the software ready for the production environment) or directly the production environment, if the acceptance tests have already been executed.

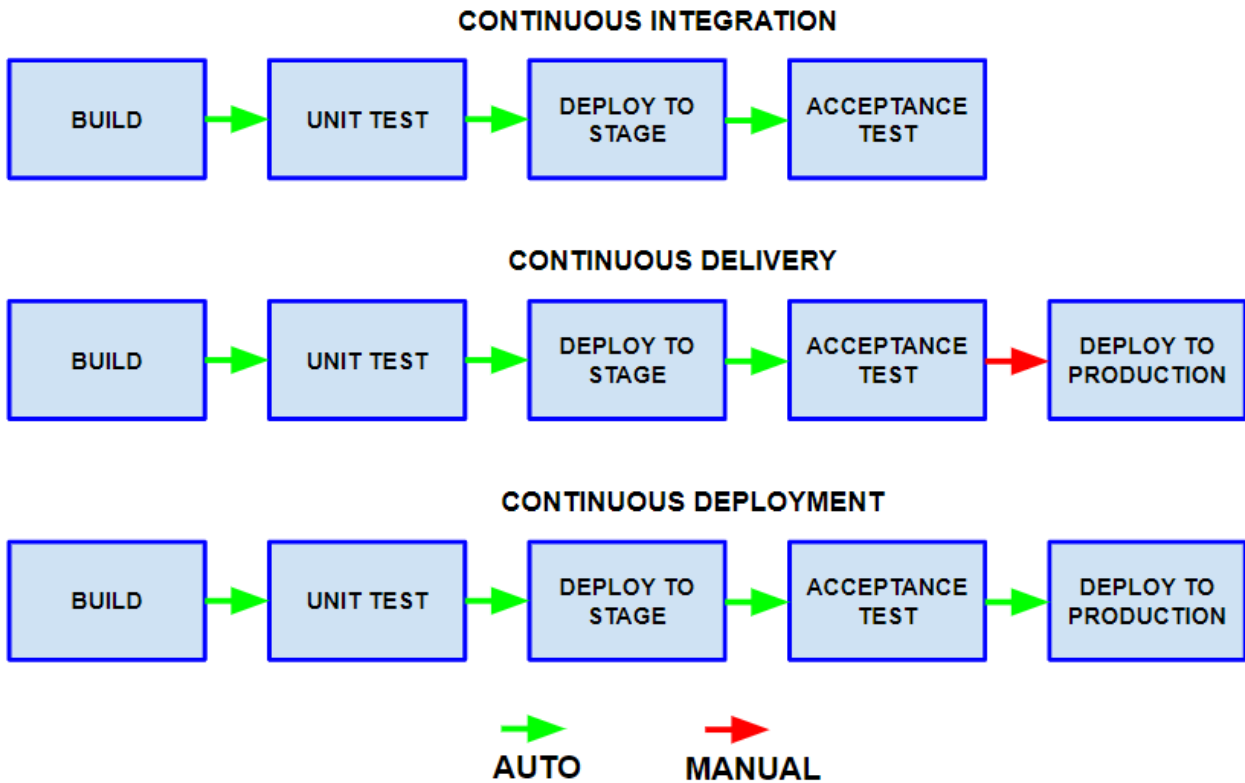


Figure 69. Continuous Integration, Delivery and Deployment Pipelines

The EFPP ecosystem is composed by a set of software components that need to be deployed and kept updated over the time. A centralized Gitlab repository contains all the tools and configuration necessary to deploy or update a component to the corresponding environment (test and production). A high-level overview of the EFPP CI/CD Pipeline system is shown in Figure 70.

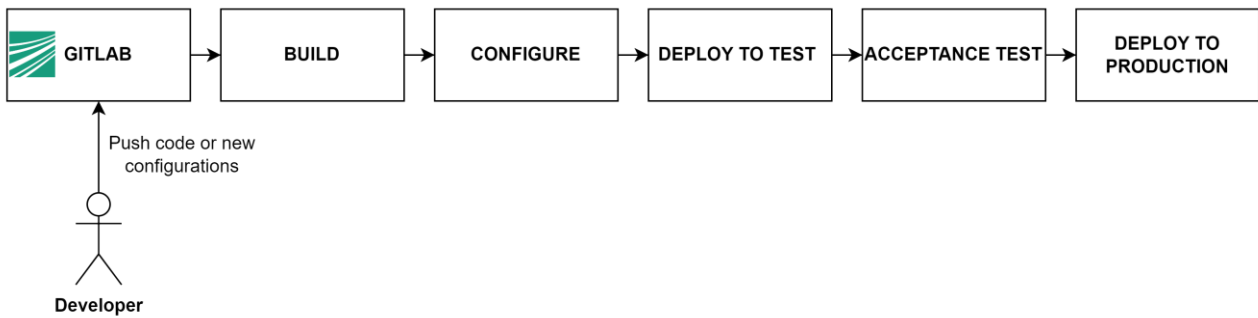


Figure 70. High-level Overview of the EFPP CI/CD Pipeline System

All the related operations are described in Figure 71:

CI/CD Operation	Tool	Description
<b>BUILD</b>	Docker	The EFPP Components have been delivered as Docker Containers, therefore, a Dockerfile is responsible to produce the artifacts
<b>CONFIGURE</b>	Ansible + Docker Compose	The software is pre-configured using a set of ansible scripts, in combination of other specific Container's configurations described by the docker-compose file (produced by the ansible script itself)

<b>DEPLOY TO TEST</b>	Gitlab CI/CD Pipelines	The tools provided by GitLab (Gitlab CI/CD System) are responsible to transfer all the artifacts and their configuration files to the target machine and execute the boot up of the service
<b>ACCEPTANCE TEST</b>	Human validation of the system + automatic tests	Once the service has been deployed, a first human evaluation is performed to ensure the correctness of the deployment in terms of functionalities. Moreover, a periodic execution of some tests ensure that the system is constantly up and running as expected
<b>DEPLOY TO PRODUCTION</b>	Gitlab CI/CD Pipelines	The tools provided by GitLab (Gitlab CI/CD System) are responsible to transfer all the artifacts and their configuration files to the target machine and execute the boot up of the service

Figure 71. EFPF CI/CD Pipeline Operations

Initially, at the beginning of the project, the deployment was completely distributed with each technical partner providing hosting and using the in-house deployment process for their components. With the development of the EFPF DevOps architecture with defined container management and deployment pipeline, the Ecosystem Enablers have been migrated to this deployment infrastructure using the Data Spine as an architecture proof-of-concept and template. The deployment of the core Ecosystem Enablers is now centralized and coordinated using the deployment pipeline. More details on the Data Spine deployment can be found in 'D6.2: EFPF Integration and Deployment - Final Report'.

## 2.7 Testing Scenarios & Framework

The successful operation of the Data Spine depends upon the correctness of the integration of its components. Therefore, after a fresh deployment and initial setup, or after the upgradation of one or more components, it becomes very important to ensure that the Data Spine continues to provide the expected functionality. The automated Integration Testing Framework runs the predefined integration tests to ensure this. In addition, as Data Spine is deployed as a cloud-native component, it must be highly performant. The Performance Testing Framework in EFPF runs stress/load tests to evaluate the Data Spine's performance and recognizes needs for infrastructure upgradation, if any.

### 2.7.1 Integration Testing Scenarios

The testing scenarios for the Data Spine are of two types:

1. **Integration Testing Scenarios:** To test the correction of deployment and configuration of the individual components of the Data Spine, integration testing scenarios were defined.
2. **System Testing Scenarios:** To test the correctness of the integrated Data Spine stack, system scenarios were defined.

These testing scenarios have been included in Annex C to ensure readability of the document.

### 2.7.2 Integration Testing Framework

In order to implement the selected Integration Testing scenarios, the choice for the platform to run them on has fallen on the GitLab CI/CD infrastructure. A dedicated GitLab project has been created for this purpose. More details about how to run these tests is provided in the deliverable 'D6.2: EFPF Integration and Deployment - Final Report'.

Since the Integration Testing scenarios have been implemented using the Python language, in order to run those tests, it has been required to create containers containing the appropriate packages and libraries for each scenario. The GitLab CI/CD Pipeline configuration is mentioned in Figure 72 below:

### GitLab CI/CD Pipeline config

```

service-registry:
  stage: test
  when: manual
  image:
    name: python:3.8
  script:
    - cd service-registry
    - pip install -r requirements.txt
    - python ./src/main.py $EFS_URL $CLIENT_SECRET $BROKER_URL $BROKER_PORT $BROKER_U
SER $BROKER_PASS $TOKEN_ADMIN_USER $TOKEN_ADMIN_PASS
  tags:
    - container

efs:
  stage: test
  when: manual
  image:
    name: python:3.8
  script:
    - cd efs
    - pip install -r requirements.txt
    - python ./src/main.py $EFS_URL $CLIENT_SECRET $TOKEN_ADMIN_USER $TOKEN_ADMIN_PAS
S
  tags:
    - container

api-security-gateway:
  stage: test
  when: manual
  image:
    name: python:3.8
  script:
    - cd api-security-gateway
    - pip install -r requirements.txt
    - python ./src/main.py $EFS_URL $CLIENT_SECRET $TOKEN_BASIC_USER $TOKEN_BASIC_PAS
S
  tags:
    - container

message-bus:
  stage: test
  when: manual
  image:

```



```

    name: python:3.8
  script:
    - cd message-bus
    - pip install -r requirements.txt
    - python ./src/main.py $EFS_URL $CLIENT_SECRET $TOKEN_ADMIN_USER $TOKEN_ADMIN_PAS
  S $TOKEN_BASIC_USER $TOKEN_BASIC_PASS
  tags:
    - container

data-spine:
  stage: test
  when: manual
  image:
    name: python:3.8
  script:
    - cd data-spine
    - pip install -r requirements.txt
    - python ./src/main.py $EFS_URL $CLIENT_SECRET $TOKEN_ADMIN_USER $TOKEN_ADMIN_PAS
  S $TOKEN_BASIC_USER $TOKEN_BASIC_PASS
  tags:
    - container

```

Figure 72. GitLab CI/CD Pipeline Configuration for the Data Spine Integration Testing Framework

### 2.7.3 Performance Testing Scenarios

The performance testing focus on the core of the EFPF infrastructure, the Data Spine, and its components. Since the goal of these tests is to test the platform as a whole, three scenarios of testing have been identified:

- Backend synchronous communication scenario
- Backend asynchronous communication scenario
- Frontend interaction scenario

The first two scenarios cover the two communication ways in which the Data Spine can be involved. These scenarios cover the communication between different tools and services. The third scenario covers instead the situation in which users interact directly with the platform's user interface endpoints.

### 2.7.4 Performance Testing Framework

A similar approach was chosen for the performance testing as for the integration testing. The k6 framework has been selected for implementing the performance tests. These tests have been developed to be run inside docker containers. This allowed for an easy distribution of these tests and for running them on Gitlab runners as well. More details about how to run these tests are provided in the document Deliverable 'D6.2: EFPF Integration and Deployment - Final Report'.

### 2.7.4.1 Backend Synchronous Scenario

This scenario, as illustrated in Figure 74, involves a Producer service which has been created just for the purpose of serving a payload. This Payload consists of a JSON body which will be requested by the Synchronous Test Service. Since this request will happen through a NiFi register endpoint the body will have to pass through NiFi which will proxy the request to the original service.

To add an element of complexity and to better simulate a real workflow which makes use of the capabilities of NiFi a JOLT transformation (Figure 73) has been added to the scenario as well.

#### Jolt Spec

```
[
  {
    "operation": "modify-overwrite-beta",
    "spec": {
      //
      // Sums
      "sumIntData": "=intSum(@(1,intData))",
      "sumLongData": "=intSum(@(1,intData))",
      "sumDoubleData": "=doubleSum(@(1,doubleData))",
      //
      // Averages
      "avgIntData": "=avg(@(1,intData))", // note this returns a double
      "avgDoubleData": "=avg(@(1,doubleData))",
      //
      // Sort ascending
      "sortedIntScores": "=sort(@(1,intData))",
      //
      // Min, Max, Absolute Value
      "minAB": "=min(@(1,a),@(1,b))", // should be 5
      "maxAB": "=max(@(1,a),@(1,b))", // should be 10
      "abs": "=abs(@(1,negative))",
      //
      // Divide
      "aDivB": "=divide(@(1,a),@(1,b))",
      "aDivC": "=divide(@(1,a),@(1,c))", // will be 3.3333
      //
      // Divide and Round : decimal point to round to is first param
      "aDivCRounded4": "=divideAndRound(4,(@(1,a),@(1,c)))"
    }
  }
]
```

]

Figure 73. Jolt Spec for Backend Synchronous Testing Scenario

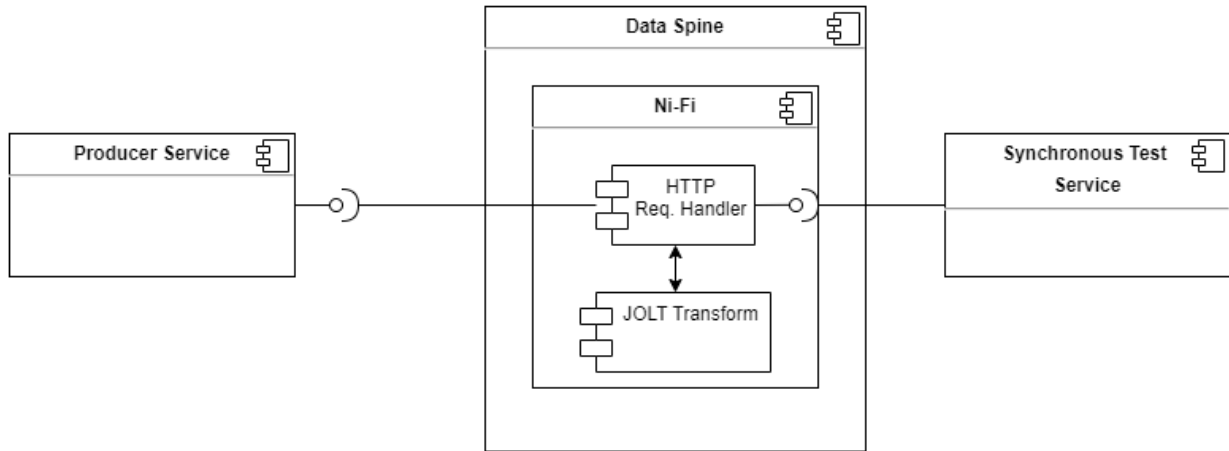


Figure 74. Architecture for Backend Synchronous Scenario

#### 2.7.4.2 Backend Asynchronous Scenario

This scenario, depicted in Figure 75 involves a Producer service which has been created just for the purpose of serving a payload as well. In this case however the service generating the payload consisting of a JSON body is integrated directly in the Asynchronous Test Service.

To add an element of complexity and to better simulate a real workflow which makes use of the capabilities of NiFi a JOLT transformation has been added to this scenario as well.

The spec of the JSON message generated for this scenario and the relative JOLT transformation spec are the same as with the Synchronous Scenario.

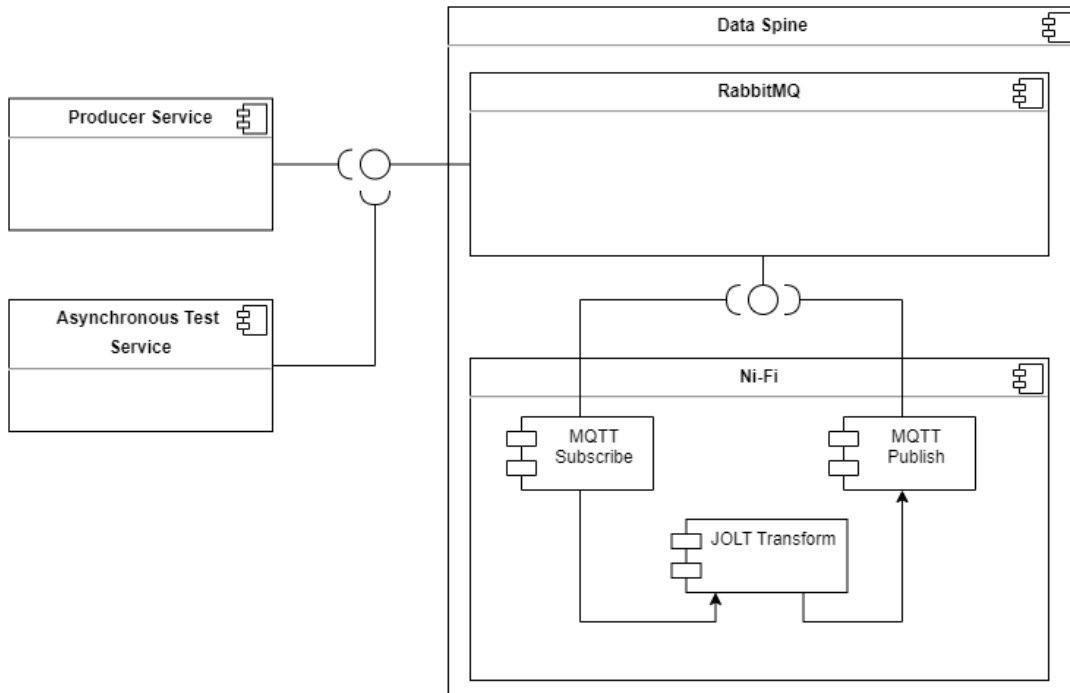


Figure 75. Architecture for Backend Asynchronous Scenario

### 2.7.4.3 Frontend Interaction Scenario

This scenario, illustrated in Figure 76 makes use of the capabilities of the k6 framework and directly tests the provided endpoints of the NiFi interface. One of the capabilities of this service is the possibility to integrate the login phase to the testing process which means that that component is tested as well.

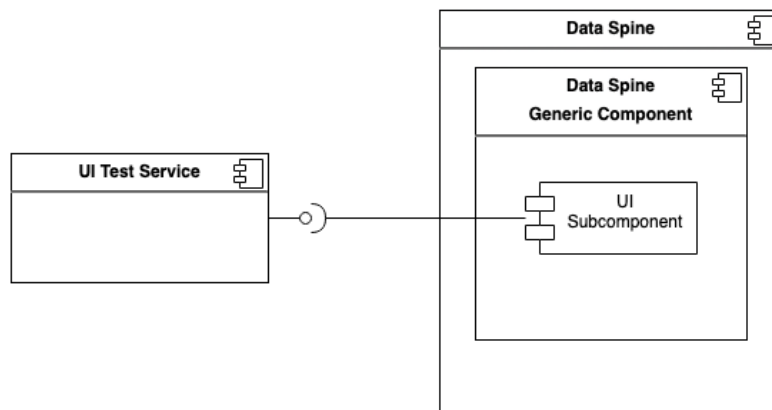


Figure 76. Architecture for Frontend Interaction Scenario

## 2.8 Platform/Service Integration & Dataflow through Data Spine

The high-level integration methodologies for different stakeholders in the EFPF ecosystem are described in Section 1.6. This section focuses on describing the integration steps specific to the Data Spine for the provision and consumption of services.

Together with enabling cross-platform interoperability, the Data Spine also enables the creation of cross-platforms in an easy and intuitive manner. In order to create composite applications using the services of different platforms, those platforms need to be integrated with the EFPF ecosystem beforehand. The prerequisite for the creation of composite applications is the federation of the platforms' identity providers with the EFS to enable SSO. It is also possible to integrate the services that do not belong to any platform, or in other words, do not have an associated identity provider, in which case the EFS becomes their default identity provider during their integration. Once this integration is complete, the services can be composed together using the Data Spine to create applications.

The design-time steps for the creation of composite applications and dataflow through the Data Spine at runtime for both synchronous (Request-Response) and asynchronous (Pub/Sub) communication patterns are described below.

### Synchronous (Request-Response) Communication

Figure 77 shows how provider1's service 'PS1' and consumer1's service 'SC1' interact with the components of the Data Spine, in order to provide and consume services respectively. The actions to be performed for service provision and consumption through the Data Spine are described below.

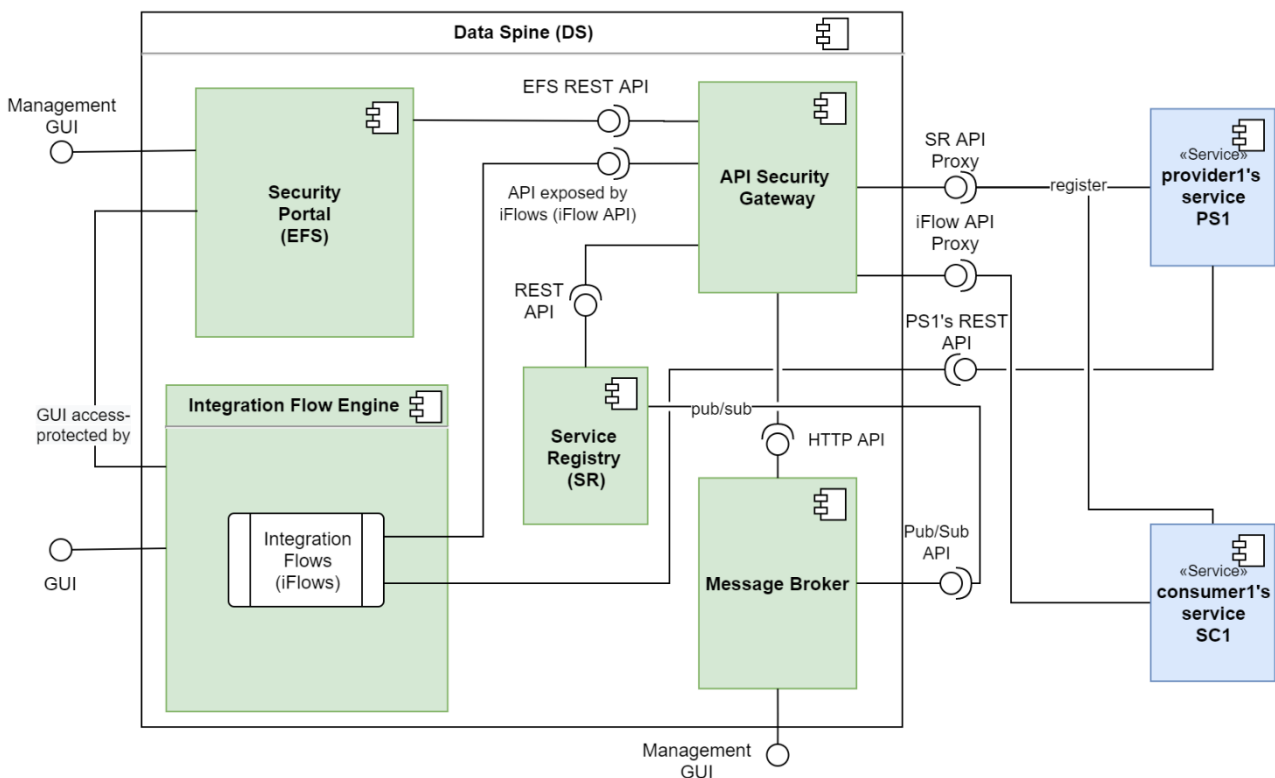


Figure 77. Synchronous Services' Integration through the Data Spine

#### Prerequisites:

- The service provider 'provider1' and service consumer 'consumer1' both have user accounts for the EFS.
- provider1 and consumer1 have the necessary permissions required to access the Service Registry.

*Design-time service integration activities:*

1. Service Registration: provider1 registers his/her service 'PS1' to the Service Registry with an appropriate service 'type' (e.g., 'marketplace-service'). Here, PS1's REST API endpoint, EP1 is already secured by its platform's identity and access management service (not shown in the figure for simplicity).
2. Service Lookup and Metadata Retrieval: consumer1 uses Service Registry's filtering API to find PS1, decides to consume it, and retrieves its technical metadata including its API spec from the Service Registry.
3. Access Configuration: consumer1 requests for and acquires the necessary access permissions to invoke EP1.
4. Access Configuration: consumer1 requests for and acquires a development space in the IFE to create integration flows.
5. Integration Flow Creation: consumer1 creates an integration flow using the GUI of the IFE that invokes EP1, performs data transformation to align request/response payload to its own data model/format, and finally creates and exposes an "interoperability-proxy" endpoint EP1-C for EP1.
6. Service/API Registration: consumer1 registers this new EP1-C API endpoint to the Service Registry.
7. Creation of a Secure Proxy API: ASG automatically creates a secure proxy API endpoint EP1-C<sub>P</sub> for EP1-C.
8. Access Configuration: consumer1 requests for and acquires the necessary access permissions to invoke EP1-C<sub>P</sub>.
9. Integration Complete: provider1's service PS1 and consumer1's service CS1 are now integrated through the Data Spine and CS1 can start invoking PS1 and obtain a response in the format required by it as illustrated in Figure 78.

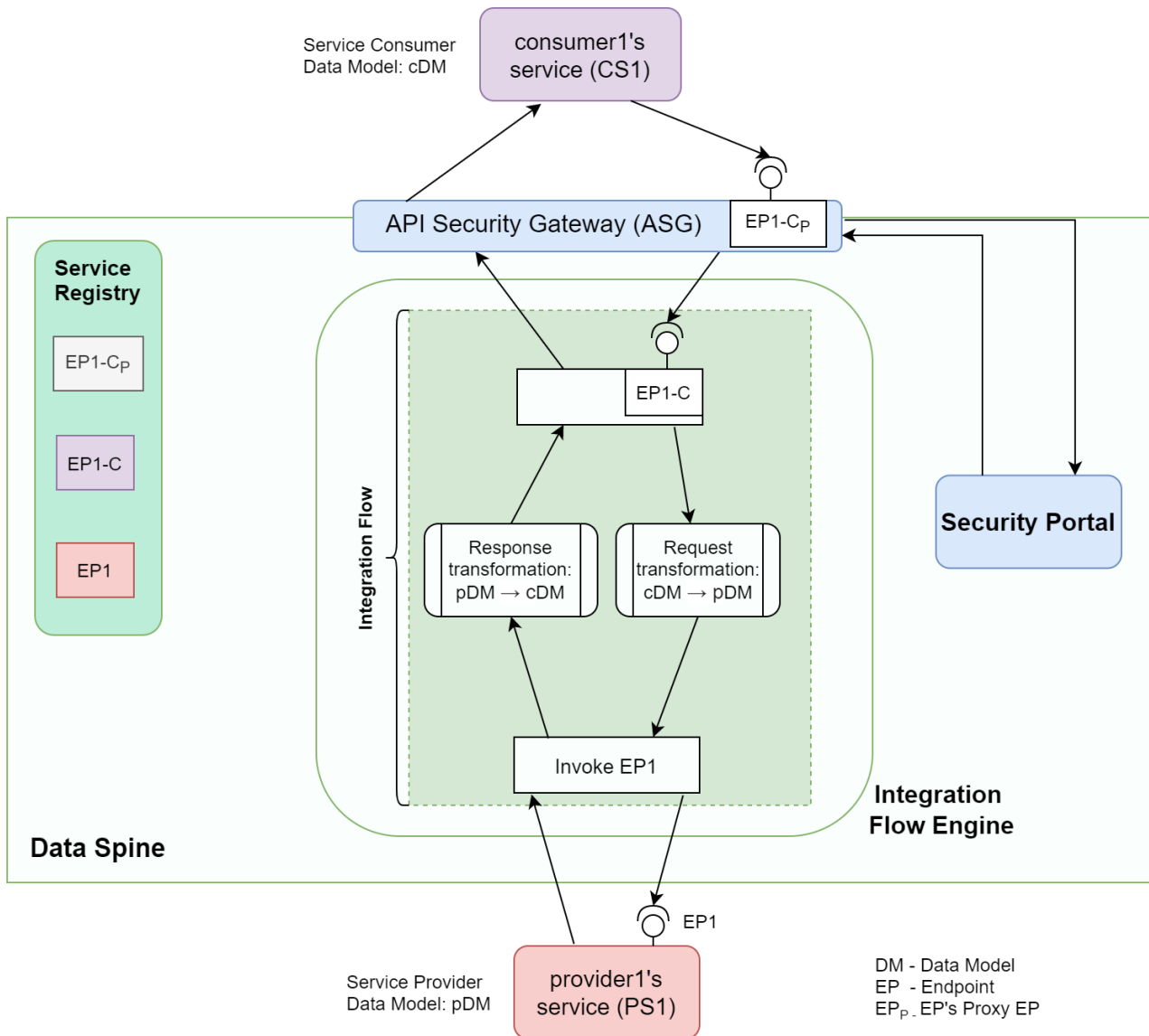


Figure 78. Example of Synchronous Communication Dataflow through the Data Spine

### Asynchronous (Pub/Sub) Communication

Figure 79 shows how publisher1's service 'pub1' and subscriber1's service 'sub1' interact with the components of the Data Spine to provide and consume services, respectively. The actions to be performed for service provision and consumption through the Data Spine are described below.

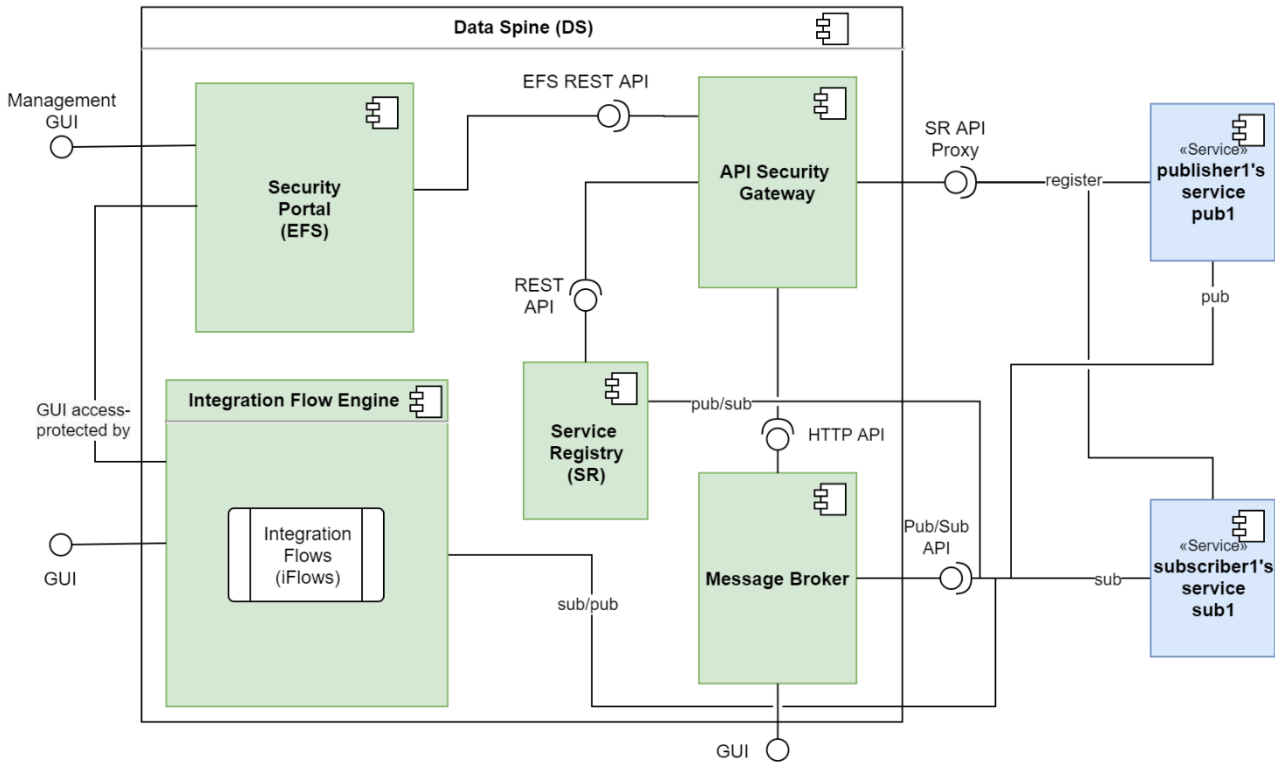


Figure 79. Asynchronous Services' Integration through Data Spine

*Prerequisites:*

- The service provider 'provider1' and service consumer 'consumer1' both have user accounts for the EFPF Interop Security Portal.
- provider1 and consumer1 have the necessary permissions required to access the Service Registry.

*Design-time service integration activities:*

1. Access Configuration and MB User Account: publisher1 visits the Pub/Sub Security Service Dashboard. A new Message Bus (MB) user account and company vhost would be created for publisher1 if it does not exist already. Otherwise, publisher1 will be granted access to the existing company vhost. publisher1 then uses the Pub/Sub Security Service Dashboard to register their service 'pub1' and create a topic 'companyx-com/p1/DDATA/topic1' in the MB for it to publish to. The credentials and configuration details needed to publish to the MB are then requested in the Pub/Sub Security Service.
2. Publisher Configuration: publisher1 configures his/her service 'pub1' to publish to the Message Bus over the topic 'companyx-com/p1/DDATA/topic1' using the credentials and configuration details obtained from the Pub/Sub Security Service.
3. Service Registration: publisher1 registers pub1 that consists of this Pub/Sub API containing its publication information to the Service Registry.
4. Service Metadata Retrieval: subscriber1 decides to subscribe to pub1's topic 'companyx-com/p1/DDATA/topic1' and gets the technical metadata for pub1 including its API spec from the Service Registry.



5. Access Configuration: subscriber1 wants to subscribe to topic 'companyx-com/p1/DDATA/topic1', perform data transformation using the Integration Flow Engine, and publish back transformed data to the MB over the topic 'companyz-org/s1/DDATA/topic1'.
6. Access Configuration (and MB User Account): subscriber1 visits the Pub/Sub Security Service Dashboard. A new Message Bus (MB) user account and company vhost would be created for subscriber1 if it does not exist already. Otherwise, subscriber1 will be granted access to the existing company vhost.
7. Topic Subscription: subscriber1 visits the Pub Sub Security Service Dashboard and searches for the topic 'companyx-com/p1/DDATA/topic1' in the View Topics page. subscriber1 then selects the request consume permission button for topic 'companyx-com/p1/DDATA/topic1'. subscriber1 then waits for approval from the topic owner and downloads the credentials/configuration details needed to subscribe to the topic.
8. Topic Creation: subscriber1 uses the Pub/Sub Security Service Dashboard to register their iFlow and creates a topic 'companyz-org/s1/DDATA/topic1' for it to publish to. subscriber1 then downloads the credentials/configuration details needed to publish to topic 'companyz-org/s1/DDATA/topic1'.
9. Access Configuration: subscriber1 requests for and acquires the necessary access permissions to create integration flows in the Integration Flow Engine (NiFi) component of Data Spine. (Details: DS NiFi User Guide)
10. Integration Flow Creation: subscriber1 creates an integration flow using the GUI of the Integration Flow Engine to subscribe to 'companyx-com/p1/DDATA/topic1', perform data transformation and finally to publish the resulting data to the Message Bus over the topic 'companyz-org/s1/DDATA/topic1' using the credentials/configuration details provided by the Pub Sub Security Service.
11. Service Registration: subscriber1 registers his/her service with the APIs containing its subscription and publication information to the Service Registry.
12. Integration Complete: publisher1's service and subscriber1's service are now integrated through the Data Spine and, sub1 can subscribe to the topic 'companyz-org/s1/DDATA/topic1' and obtain data in the format required by it as illustrated in Figure 80.

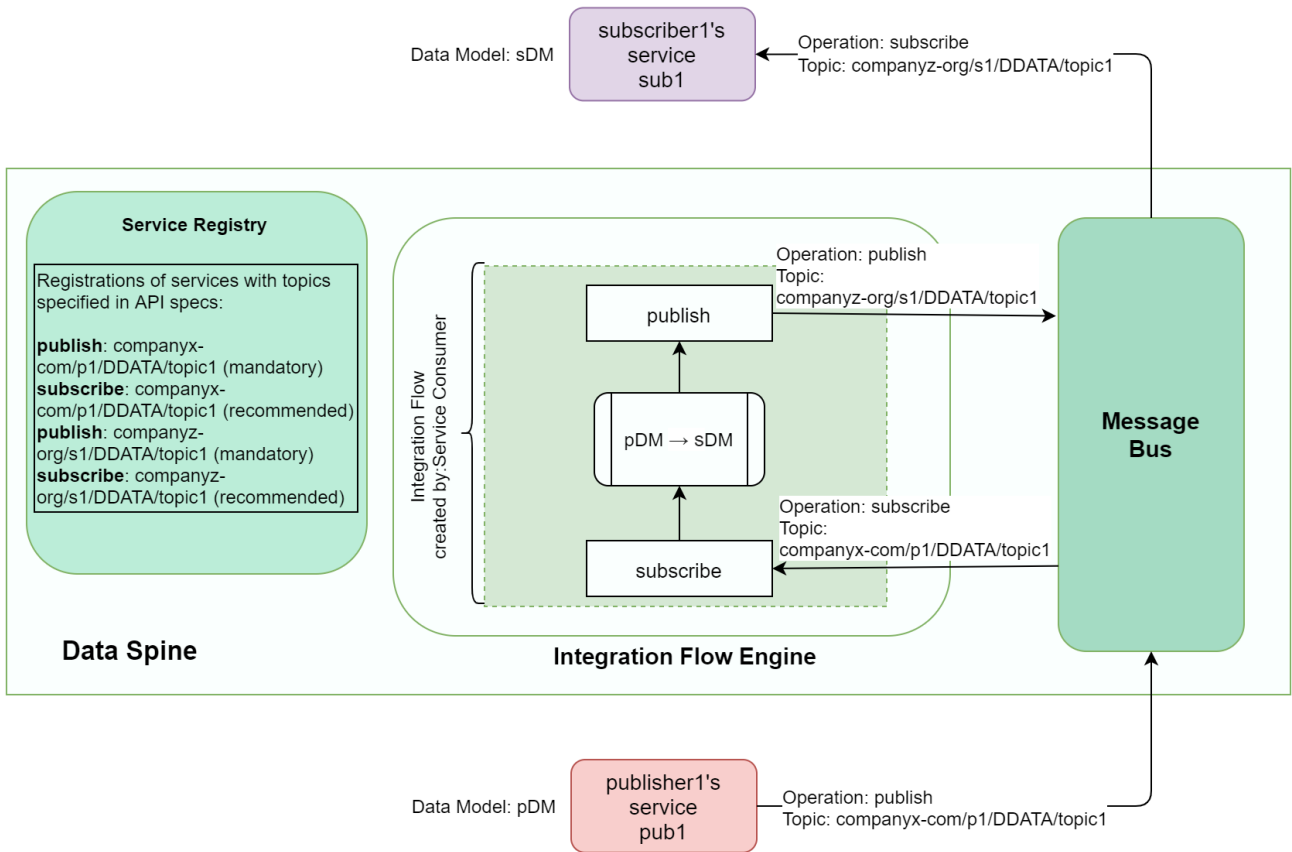


Figure 80. Example of Asynchronous Communication Dataflow through the Data Spine

## 2.9 Data Spine Usage in Pilots and Open Call Experiments

The Data Spine was and remains extensively used in the pilots as well as in the Open Call experimentation scenarios. In order for the pilots and Open Call experimentation activity to have a smooth start, comprehensive documentation and examples were published onto the EFPF Dev-Portal and active support was / is being given during the experimentation phase. Figure 81 illustrates the usage of Data Spine components for realising the pilot use case scenarios. Moreover, all the Open Call experimenters are making use of the Data Spine components in their experiments. A brief description of dataflows and data models used in these scenarios can be found in Sections 1.4 and 4 where the usage of Data Spine is also highlighted. The detailed description of these scenarios and their implementation details can be found in the respective pilot and Open Call deliverables.

No	Solution	Relates to Pilot	Data Spine Components Used
<b>S 1a</b>	Solution 1a: Production Optimisation (Predictive Maintenance)	Furniture	Message Bus, Integration Flow Engine, EFPF Security Portal (EFS), Service Registry
<b>S 1b</b>	Solution 1b: Production Optimisation (Operator Error)	Furniture	Message Bus, EFPF Security Portal (EFS), Service Registry
<b>S 2</b>	Solution 2: Bin Fill Level Monitoring	Furniture CE	Message Bus, Integration Flow Engine, EFPF Security Portal (EFS), Service Registry
<b>S 3</b>	Solution 3: Workflow and Service Automation Platform	Furniture Aero-space	EFPF Security Portal (EFS), Service Registry

<b>S 4</b>	Solution 4: Matchmaking Service	Aero-space CE	EFPP Security Portal (EFS), Service Registry
<b>S 5a</b>	Solution 5a: Efficient Resources Management Solutions (Visual Detection)	Aero-space	Message Bus, Service Registry
<b>S 5b</b>	Solution 5b: Efficient Resources Management Solutions (Stores Monitoring)	Aero-space	Message Bus, Service Registry
<b>S 6</b>	Solution 6: Workplace Environment Monitoring	Aero-space	Message Bus, Integration Flow Engine, EFPP Security Portal (EFS), Service Registry
<b>S 7</b>	Solution 7: Tendering & Bid Management	All domains	EFPP Security Portal (EFS), Service Registry
<b>S 8</b>	Solution 8: Almende Risk Analysis & Management (ROAM) Tool	All domains	Message Bus, Integration Flow Engine, EFPP Security Portal (EFS), Service Registry
<b>S 9</b>	Solution 9: Catalogue Service	All domains	EFPP Security Portal (EFS), Service Registry
<b>S 10</b>	Solution 10: Business Network Intelligence	All domains	Message Bus, EFPP Security Portal (EFS)
<b>S 11</b>	Solution 11: Data Analytics	CE	Message Bus, Integration Flow Engine, EFPP Security Portal (EFS), Service Registry
<b>S 12</b>	Solution 12: Blockchain Application	CE Aero-space	EFPP Security Portal (EFS)
<b>S 13</b>	Solution 13: Online Bidding Process	CE	Integration Flow Engine, EFPP Security Portal (EFS), Service Registry
<b>S 14</b>	Solution 14: System Security Modelling	CE	EFPP Security Portal (EFS), Service Registry

Figure 81. Usage of Data Spine in the Pilot Solutions

## 2.10 Evaluation

The Data Spine is a core central component that enables security and communication in the EFPP ecosystem and therefore, evaluating the realised Data Spine technology stack concerning its performance and usability is very important. This section presents the quantitative performance evaluation of the Data Spine followed by its usability evaluation through surveys that were launched for pilots and Open Call experimenters.

### 2.10.1 Quantitative Evaluation

In this section, we evaluate the performance of the Data Spine approach by using the integrated marketplace use case example illustrated in Figure 82. In the sample realisation of the integrated marketplace solution, the Data Spine, through SSO, enables invoking the endpoints of the base platforms' marketplace services with a single set of EFPP credentials and facilitates the data model transformation process using the integration flows.

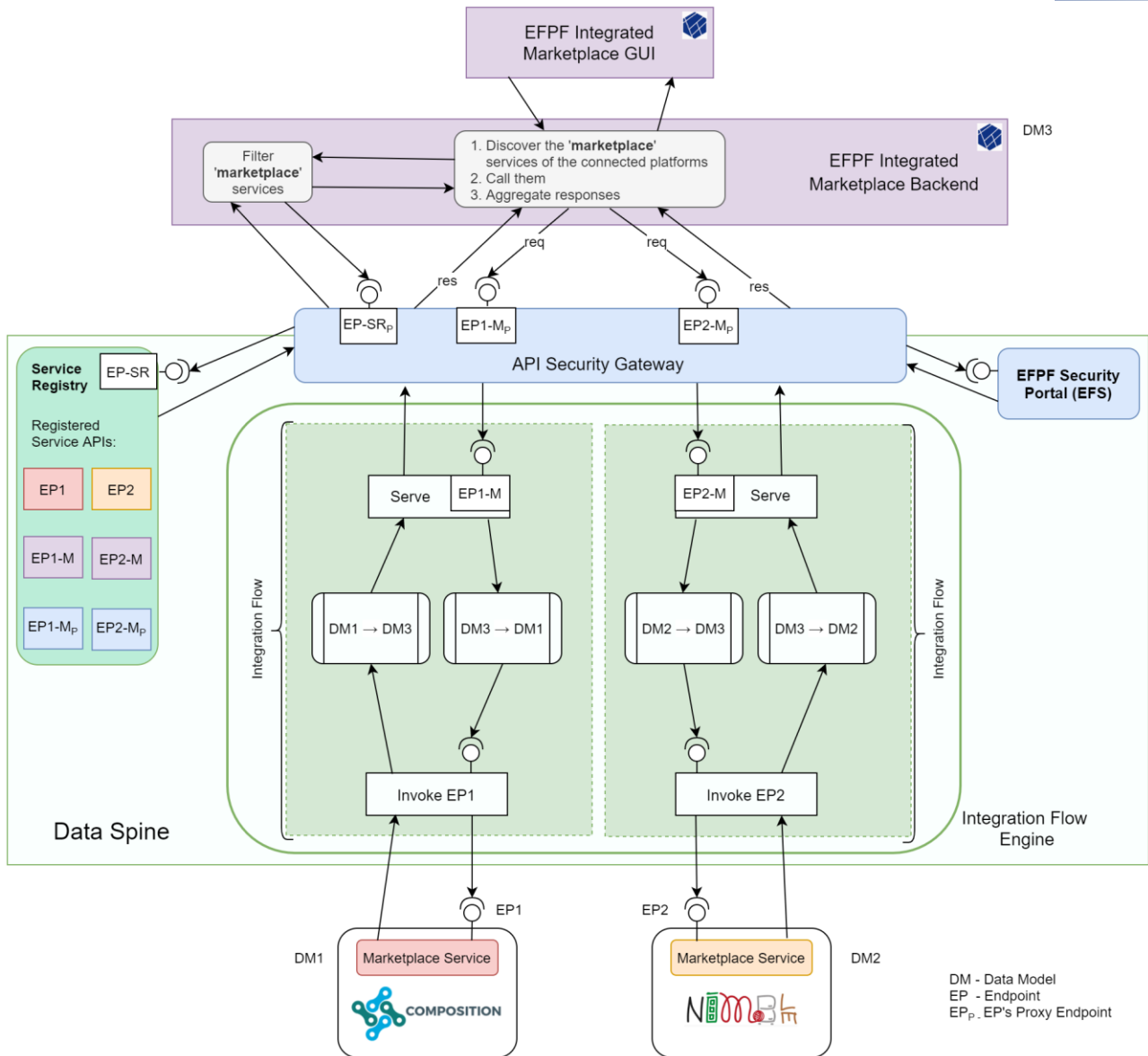


Figure 82. Integrated Marketplace Realisation Example

Without the use of the Data Spine, in the traditional approach, the developer of the integrated marketplace must obtain user accounts for each of the base platforms, use those separate sets of credentials for invoking the individual marketplace services of the base platforms and write additional source code to perform the data model transformation locally. If the developer uses specialised data model transformation tools locally, their deployment also needs to be managed separately. The high-level workflows and API calls required for getting a response from the base platforms' marketplace services in the format expected by the integrated marketplace for (a) the traditional approach, and (b) the Data Spine approach are shown in Figure 83.

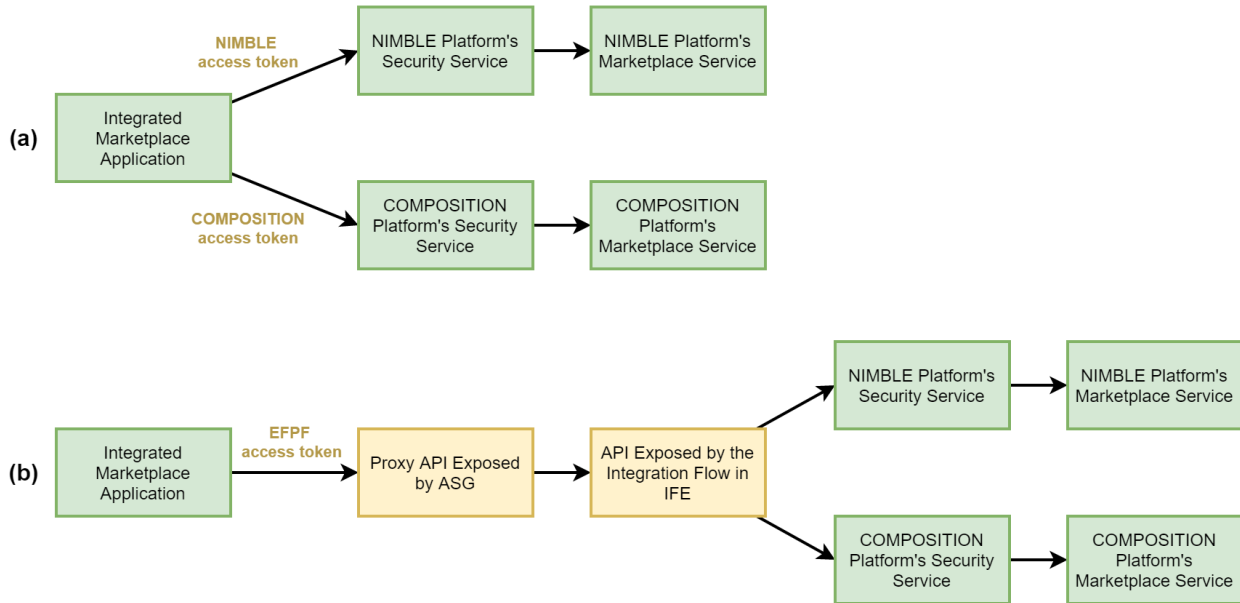


Figure 83. High-level workflows for realising the integrated marketplace solution: (a) the traditional approach, and (b) the Data Spine approach.

Using the EFPF Development Environment setup, we performed a quantitative evaluation of both approaches where the response times were measured as a sum of: (1) the time taken for calling NIMBLE platform's marketplace service and (2) the time taken for transforming the response to adhere to the Integrated Marketplace's data model. The integrated marketplace applications were realised as Java programs that recorded the response times. The Data Spine components were realised using the technologies presented in Section 2.5. Apache NiFi, LinkSmart Service Catalog and RabbitMQ were deployed on a machine with 2 vCPUs and 8 GiB RAM alongside 5 other Docker containers. Apache APISIX and Keycloak were deployed on another machine at a different physical location with 4 vCPUs and 16 GiB RAM alongside 14 other Docker containers. To minimise the impact of public network traffic variations, the experiment was repeated 500 times for each of the two approaches and the average response times were calculated. For approach (a), the average response time was 192.36 ms, whereas for (b), it was 228.03 ms. This overhead of 35.67 ms relates to the two additional proxy endpoints introduced in the Data Spine approach as highlighted in Figure 83. We have reduced this overhead further by deploying all the components of Data Spine at the same local network in the EFPF Test and Production Environments. Thus, the Data Spine provides multitude of advantages at the cost of a reasonable performance overhead.

## 2.10.2 Data Spine Usage Survey

Data Spine usage surveys were launched for pilots and Open Call experimenters that were intended to gather data about:

- who is using the Data Spine and for what purpose,
- the ease of use of the Data Spine components and potential improvements,
- the understanding of the interoperability approach followed by the Data Spine and awareness about the data transformation tools provided, etc.

A brief summary of responses for the most important questions is included below. The surveys also included questionnaire regarding Data Model Interoperability. The summary of responses for those questions has been included in Section 4.

### Usability evaluation through Data Spine Survey for Pilots

The survey received 18 responses.

**Question:** If you are using the Data Spine to realise pilot solutions or any other composite applications, please select/enter their names below.

**Response:** Figure 84

- Production Optimisation (Predic... 6
- Production Optimisation (Opera... 2
- Bin Fill Level Monitoring 3
- Workflow and Service Automati... 4
- Matchmaking Service / Product ... 1
- Efficient Resources Managemen... 2
- Workplace Environment Monito... 3
- Tendering & Bid Management 3
- Almende Risk Analysis & Manag... 2
- Catalogue Service 3
- Business Network Intelligence 1
- Data Analytics 4
- Blockchain Application 4
- Online Bidding Process 2
- Other 1

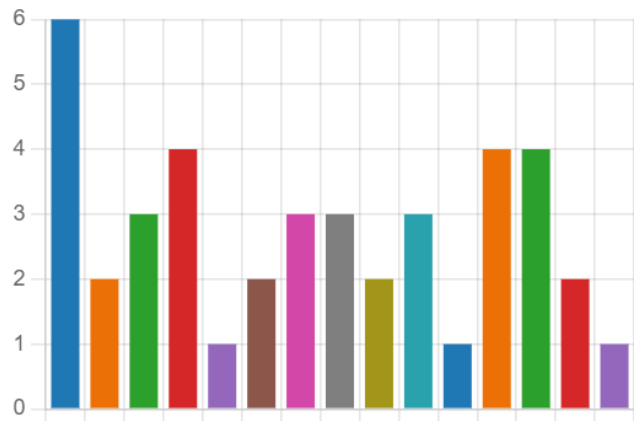


Figure 84. Data Spine Survey for Pilots: Pilot System Integrators that Use the Data Spine

**Question:** Please rate the 'ease of use' for Data Spine to integrate services, tools, or platforms

**Response:** Figure 85

- Very easy
- Easy
- Moderate
- Difficult
- Very difficult



Figure 85. Data Spine Survey for Pilots: Ease of Use Rating for the Data Spine

**Question:** More details on the ease of use for the components of Data Spine

**Response: Figure 86**

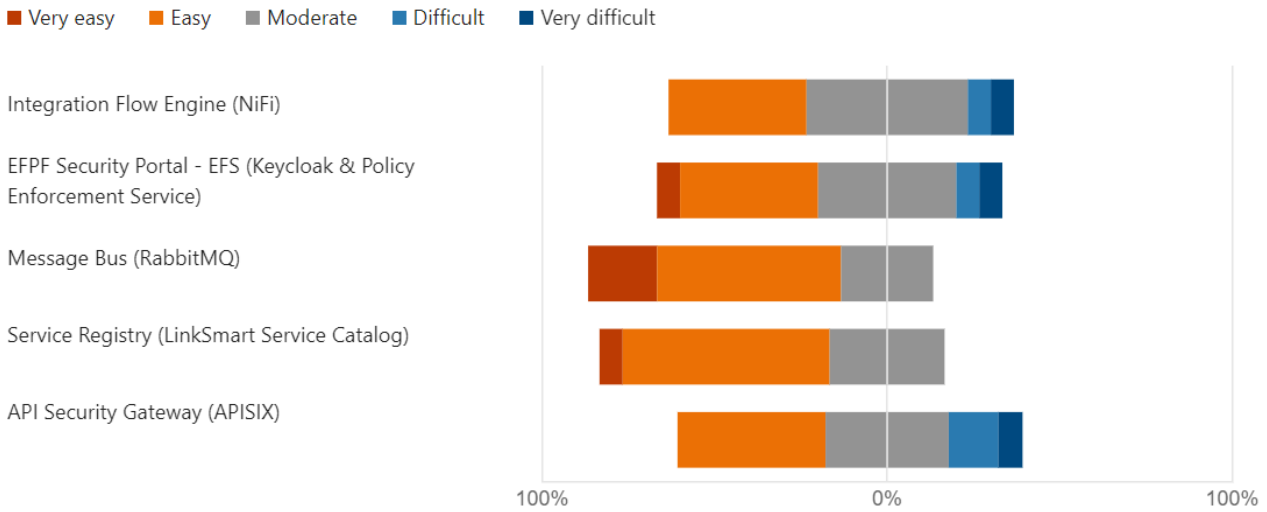


Figure 86. Data Spine Survey for Pilots: Ease of Use Rating for the Individual Data Spine Components

It should be noted that many survey respondents were using APISIX without even knowing it, which is good, as the API Security Gateway is intended to be consciously used only by the admins, and not by the users. Usage of the other services' secure proxy APIs, e.g., Service Registry's API through APISIX, without even knowing it, is a good sign towards APISIX's "transparency" towards the users in the Data Spine stack.

In order to improve the usability, the documentation was further improved. A review activity for the documentation on the EFPF Dev-Portal was launched in WP6: Integration and Deployment to ensure the comprehensiveness and readability of the published documentation. New User Guide 101s were creating as the starting point of the EFPF ecosystem user documentation as explained in Section 1.6. The Data Spine user documentation is arranged as illustrated in Figure 87. Moreover, several (executable) examples that also serve as templates were directly added to the IFE (DS NiFi) in the Production Environment as illustrated in Figure 88.

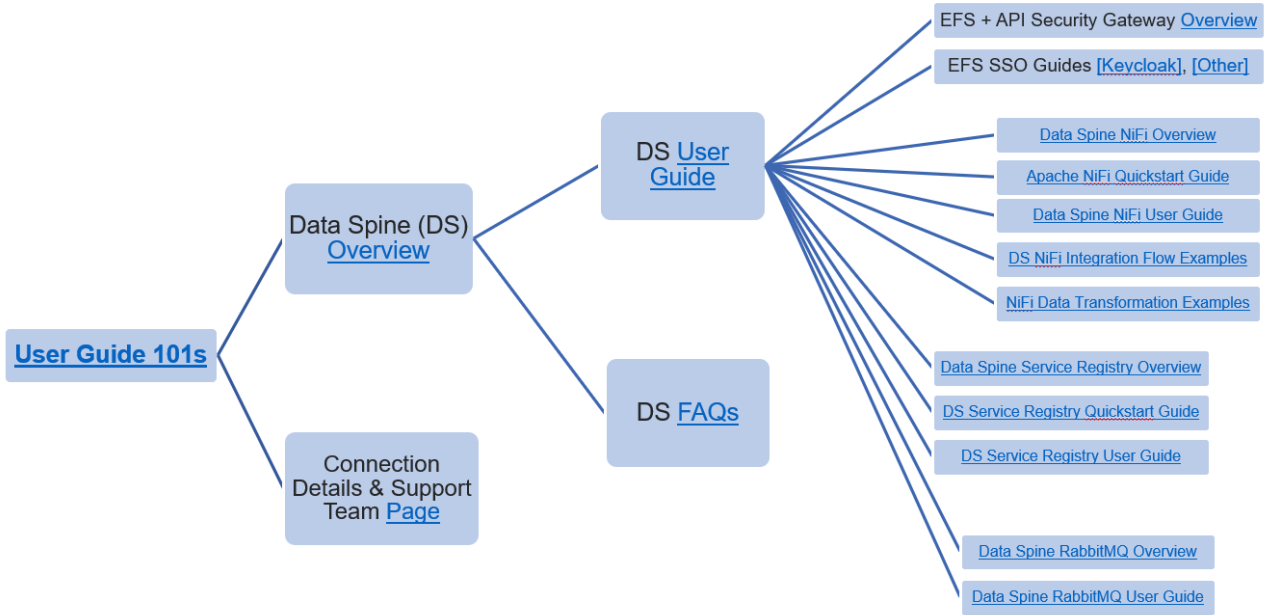


Figure 87. Data Spine Documentation on the EFPF Dev-Portal

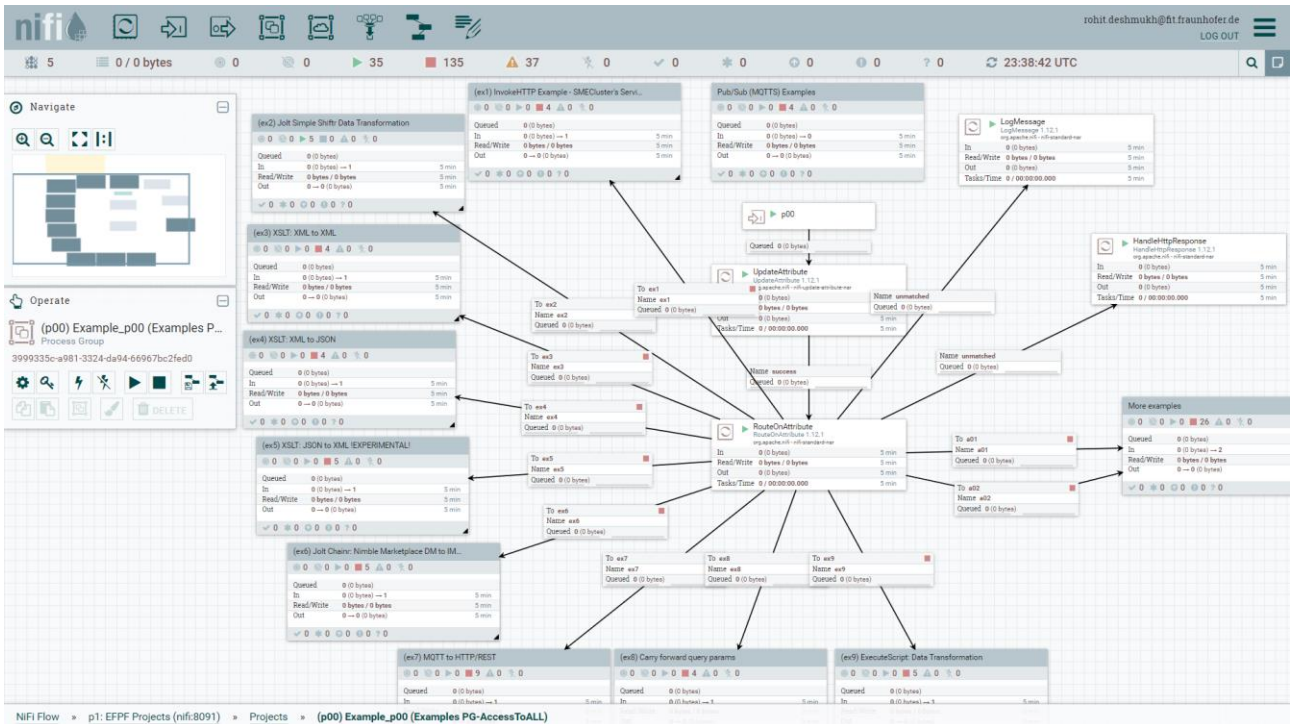


Figure 88. Executable Example Templates on the Production Environment Instance of DS NiFi

### Usability evaluation through Data Spine Survey for Open Call Experiment

The survey had received 12 responses at the time of writing this deliverable (June 2022). Some Open Call experiments are still work in progress.

**Question:** Which Open Call sub-project are you involved in?

**Response:** Figure 89



ID	Responses
1	DAWN - Data-driven service for quality monitoring and Assessment in robotic arc WeldiNg for steel joints of industrial components.
2	Digitanimal
3	ExtraCash
4	Sappi Scailable Edge Deployment project
5	PREMAR (Cabomar + CTAG)
6	Smart Metal
7	Nissatech Innovation Centre
8	Predictive and Connected Continuous Galvanization Line with EFPF Services
9	XAI-FS
10	Digitisation of the meat processing chain from Farm to Fork
11	GoGreen-Smart platform for quantification of CO2 and H2O footprint in industry
12	XAI-FS

Figure 89. Data Spine Survey for Open Call: Respondents

**Question:** Please rate the 'ease of use' for the Data Spine to integrate tools, services, or platforms and create composite applications.

**Response:** Figure 90

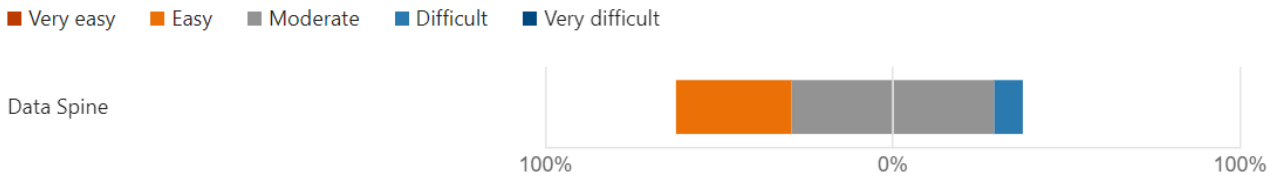


Figure 90. Data Spine Survey for Open Call: Ease of Use Rating for the Data Spine

**Question:** More details on the ease of use for the components of Data Spine

**Response:** Figure 91

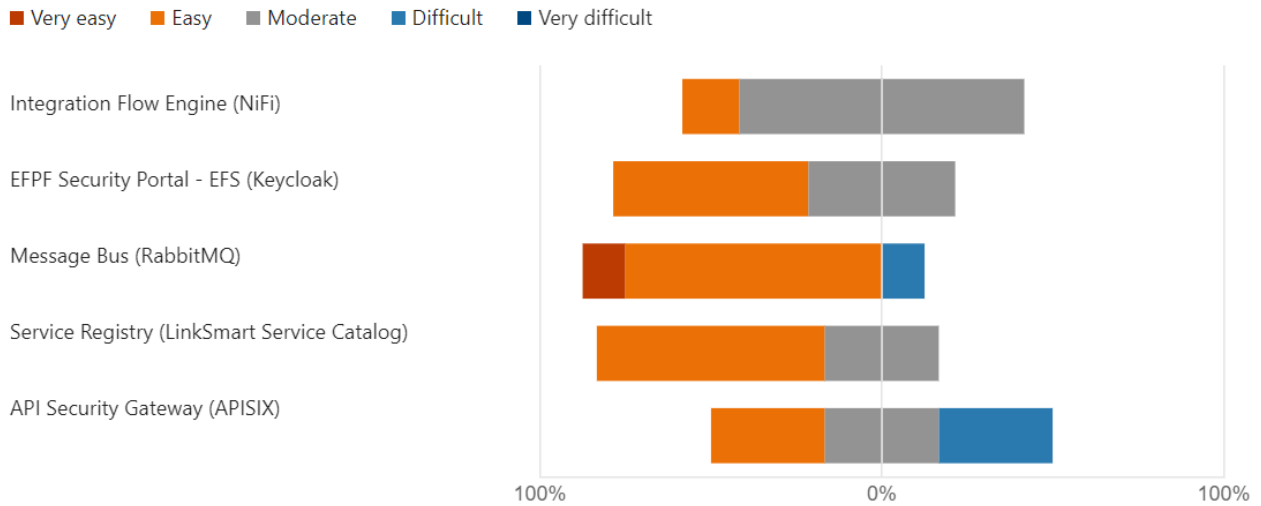


Figure 91. Data Spine Survey for Open Call: Ease of Use Rating for the Individual Data Spine Components

## 3 Interfaces for Tools, Systems and Platforms

This section presents the interfaces for tools, services, systems, and platforms in the EFPF ecosystem. It also discusses some other topics closely concerned with APIs such as API Management and Interface Contracts.

### 3.1 Introduction

The individual tools, services, systems, and platforms, provided by different partners (and 3<sup>rd</sup> parties) to the EFPF project are the building blocks of the EFPF ecosystem. These tools, services, systems, and platforms must be able to communicate with each other through the Data Spine. In that respect, relevant interfaces and APIs need to be defined.

The definitions of these APIs need to be complete, i.e., the API definition should cover all the possible aspects of the API that a consumer needs to know to consume the service. Moreover, the vocabulary for defining the APIs should be uniform as otherwise the consumer might need to go through API definitions following different vocabularies that would prove to be more time consuming, tedious and error prone.

To avoid such issues, in EFPF, API specification standards are used to specify the APIs. For specifying APIs of services that follow synchronous request-response communication pattern, OpenAPI Specification [OAS22] standard is used whereas for specifying APIs of services that follow asynchronous Pub/Sub communication pattern, AsyncAPI Specification [AAS22] standard is used. These industrial-grade standards provide standard vocabulary, structure, and formats for specifying the APIs. Moreover, they provide tooling such as editors for writing and verifying the correctness of the API specifications, documentation generation tools for visualising the APIs, code generation tools for generating server and client-side source for providing or consuming APIs in various programming languages, testing tools to perform functional tests on APIs, etc. Thus, use of these standards for specifying APIs ensures uniformity across and completeness of the API specifications in EFPF.

The next section presents a brief description of the interfaces and APIs of tools, services, systems, and platforms that together constitute the EFPF ecosystem. A brief description of the APIs of base platforms is also given in the next section. Other topics that are closely related to API Management such as the lifecycle management of APIs, discovery of APIs, monitoring of API endpoints, API contracts, and API access consent delegation, etc. are also presented in the subsequent sections.

### 3.2 Interfaces for Tools, Systems and Platforms

This section presents a brief description of the architectures of tools, services, systems, and platforms in the EFPF ecosystem using UML Component Diagrams or high-level architecture diagrams that highlight their interfaces such as GUIs and APIs.

#### 3.2.1 Interfaces for Tools & Services in the EFPF Ecosystem

##### 3.2.1.1 EFPF Portal

The EFPF portal consists of an Angular web application (frontend) and a .NET-Core-based backend. The frontend does not provide any API endpoints. The backend provides an HTTP REST API for user registration and event logging. This API is provided only for the frontend and does not provide endpoints to other components at the current time.

Figure 92 provides an overview of the existing HTTP REST API of the Portal Backend, which shows REST API for creating and updating user and create events for logging purposes.

REST Endpoint	HTTP Method	Description
/user	POST	Creates new user
/user	PUT	Updates an existing user
/event	POST	Creates new event

Figure 92. EFPF Portal Backend HTTP REST API

### 3.2.1.1.1 HTTP REST API for User Registration

The backend provides API endpoints required for creating and updating users. While updating a new user only contacts the EFPF Security Portal creating a new user is using a 3<sup>rd</sup> party mail service and the Smart Contracting.

Figure 93 shows the data model for creating a user profile and Figure 94 shows the data model for updating a user profile.

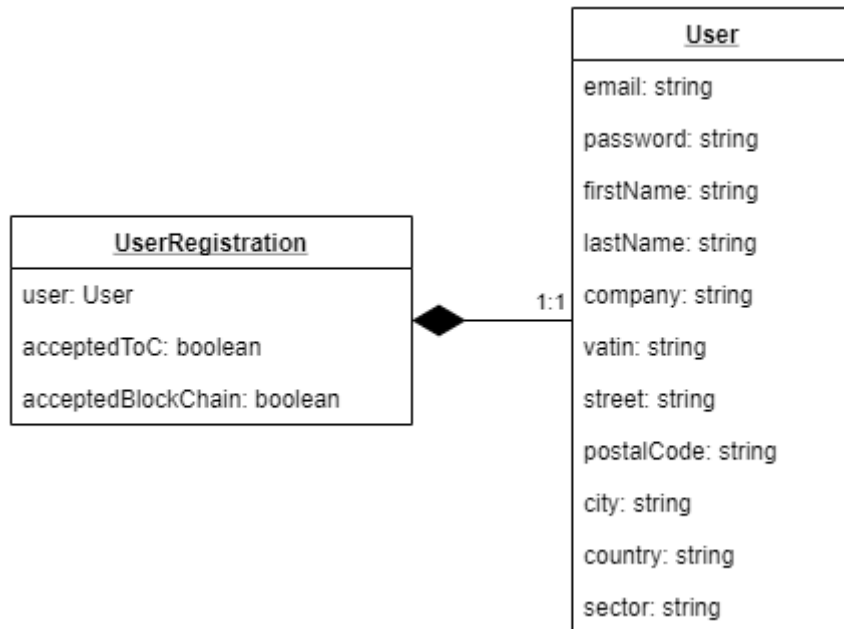


Figure 93. EFPF Portal - User Registration Data Model

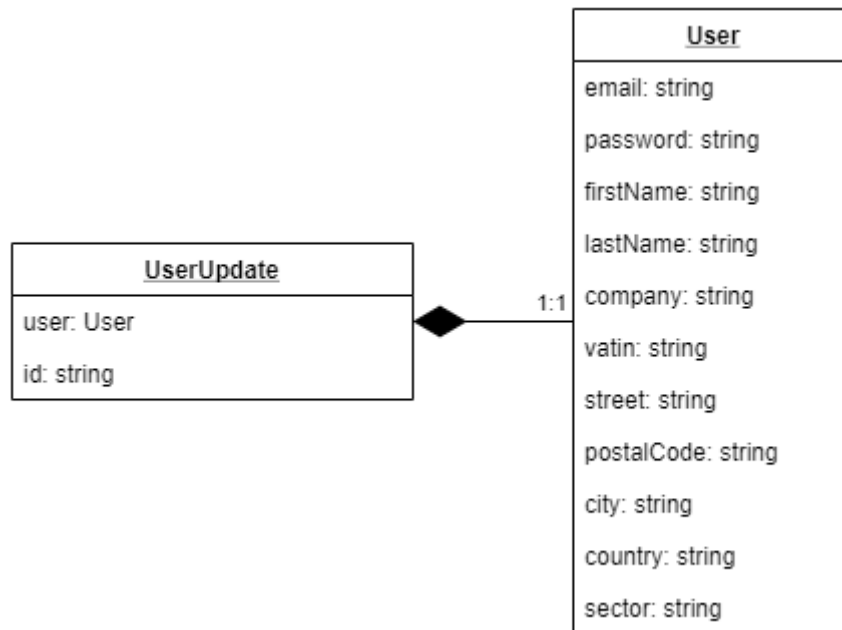


Figure 94. EFPF Portal - User Update Data Model

The attributes are described below:

The UserRegistration object consists of:

- user: user object to be created
- acceptedToC: Whether or not the user agreed to the terms and conditions at registration time
- acceptedBlockChain: Whether or not the user agreed to the BlockChain conditions at registration time

The UserUpdate object consists of:

- user: user object including attributes to be updated
- id: user id of the user object to be updated

The User object consists of:

- email: The users email address, also used as username at registration time
- password: The password set up by the user at registration time
- firstName: First name of the user
- lastName: Last name of the user
- company: Company name
- vatin: VAT identification number of the company
- street: Street information of the company
- postalCode: Postal code of the company
- city: City of the company
- country: Country of the company; Format: ISO 3166-1 Alpha-2 code

- sector: Industry sector of the company; Format: NACE Rev.2<sup>3</sup>

### 3.2.1.1.2 HTTP REST API for Event Logging

The backend provides an API endpoint for logging events to the Accountancy Service.

Event
userId: string
action: string
platform: string
timeStamp: string
visitedPlatform: string
visitedTool: string
query: string
facetQuery: string
queriedPlatforms: string
searchResponse:string
searchType: string

Figure 95. EFPF Portal – Event Data Model

The attributes of the event object are described below:

- userId: unique id of the user provided by the token
- action: Defined value of what the action the user executed; Currently the following options are available:
  - PLATFORM\_VISIT: User visited a platform
  - TOOL\_VISIT: User opened a tool
  - SEARCH\_EVENT: User conducted a product search
- platform: A defined value in which platform the event occurred
- timestamp: Timestamp when the event occurred
- visitedPlatform: A defined value for the platform the user has visited
- visitedTool: A defined value for the tool the user has opened

The following attributes are only provided in case a search has been conducted:

- query: The query string the user has entered
- facetQuery: Facet query string

<sup>3</sup>

[https://ec.europa.eu/eurostat/ramon/nomenclatures/index.cfm?TargetUrl=LST\\_NOM\\_DTL&StrNom=NACE\\_REV2&StrLanguageCode=EN](https://ec.europa.eu/eurostat/ramon/nomenclatures/index.cfm?TargetUrl=LST_NOM_DTL&StrNom=NACE_REV2&StrLanguageCode=EN)

- queriedPlatforms: Platforms included in the search
- searchResponse: Response of the search query
- searchType: A defined value if products or companies has been searched for

### 3.2.1.2 EFPF Marketplace

The EFPF Marketplace is retrieving and showing products from external marketplaces in a unified manner. It consists of two components:

- Marketplace UI
- Marketplace Backend

The Marketplace UI, as illustrated in Figure 96, is implemented as an Angular Web Component showing products in a unified manner and providing filter and sorting options. It retrieves content from the Marketplace Backend component, which endpoint must be provided by the hosting website.

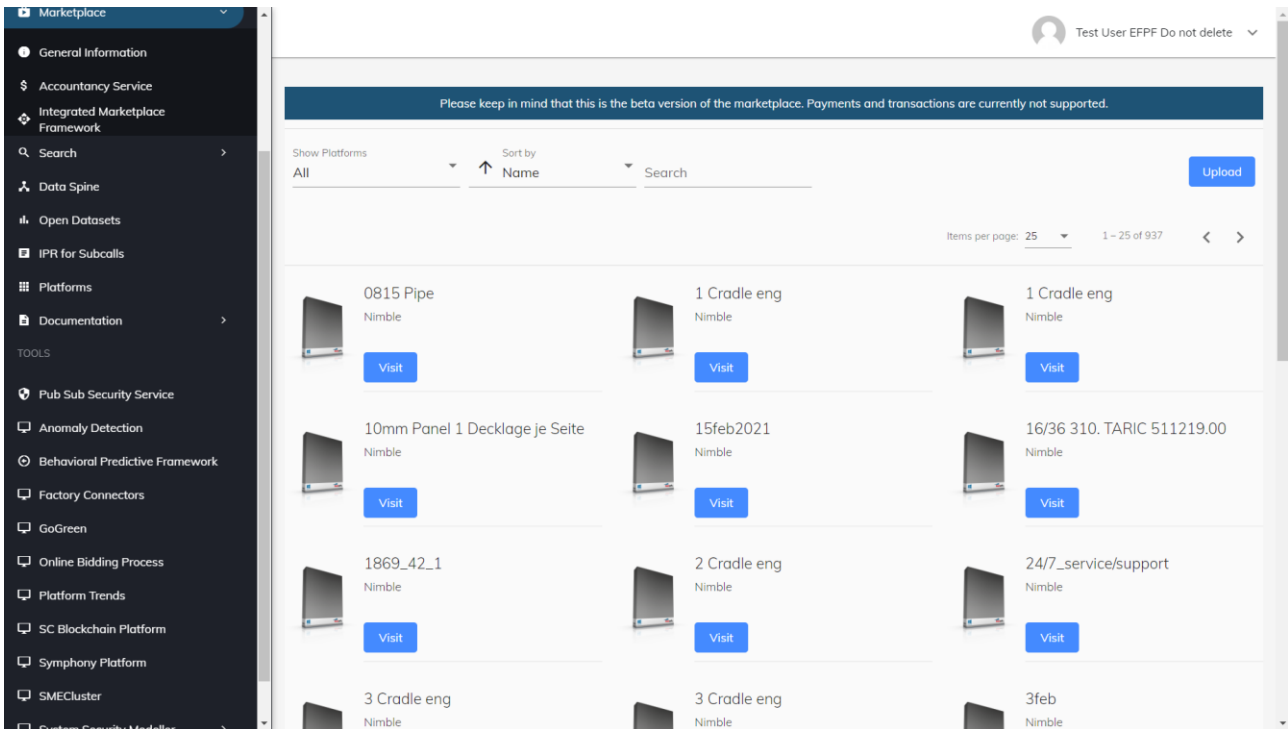


Figure 96. Marketplace UI Screenshot

The required configuration object has the following structure, as depicted in Figure 97:

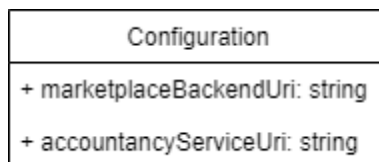


Figure 97. Marketplace UI Configuration

The Marketplace Backend is implemented as Node server application consuming the API of the Service Registry. Products provided by external marketplaces will be converted internally to a unified structure, so it can be consumed by the Marketplace UI. The

conversion results will be stored internally and updated on an hourly basis to keep the workload low.

To be able to query the Service Registry, the Marketplace Backend requires the following configuration at start:

- EFPF Security Portal URI
- EFPF Security Portal Client ID
- EFPF Security Portal Client Secret
- EFPF Service Registry URI

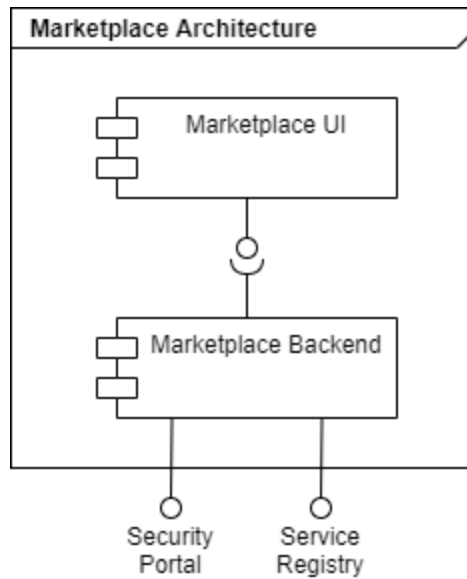


Figure 98, Marketplace UML Architecture

### Marketplace’s HTTP REST API for providing products

The Marketplace Backend provides a single HTTP REST endpoint providing products to the Marketplace UI, as shown in Figure 99.

REST Endpoint	HTTP Method	Description
/products	GET	Provides products from marketplaces registered in the Service Registry

Figure 99. Marketplace Backend Products Endpoint

#### 3.2.1.3 Accountancy Service

The Accountancy Service is a standalone component that runs independently of existing EFPF tools and services and can be integrated with unlimited number of external marketplaces. It consists of 3 main components:

- **Log Aggregator:** Gathers user behaviour data from various components of the EFPF Platform, executes different transformations and filters the content, before sending the data to the Log Persistence component



- **Log Persistence:** Stores, indexes, provides, and manages user logs to be later analysed. Since relational databases are not well-suited for managing log data, a NoSQL database like Elasticsearch is preferred due to their flexible and schema-free document structures, enabling analytics of the log data.
- **Visualization:** Enables interactive dashboards, filters and advanced data analysis and exploration of user logs.

In addition, the following custom modules listed below were developed to provide additional functionality:

- **Reporting Component:** Creates periodic (i.e., monthly) reports for each dashboard at the end of each month in PDF format and sends it as an email
- **Invoicing Component:** Processes all the payment data accumulated within each month, sums all the amounts from successful transactions realized on each marketplace, calculates a corresponding cashback amount, and creates a detailed invoice with the information including purchased products, dates of transactions as well as the calculated commission for each product. The invoice will then be used to charge marketplaces.

In order to process the accumulated log data and provide advanced visualization mechanisms, the Accountancy Service uses Elastic Stack (Elasticsearch, Logstash, Kibana) as an advanced log persistence, monitoring, processing, and visualization framework. The Logstash component is utilized as a data ingestion and server-side data processing pipeline. The data sent to Logstash from various EFPF components are forwarded to Elasticsearch for persistence after executing certain ingestion pipelines; and then Kibana dashboards are automatically updated based on the certain fields of documents stored on Elasticsearch. In addition to the central functionalities offered by the Elastic Stack, the custom modules were developed with JavaScript and they are running on Node JS environment.

The architecture of the Accountancy Service is presented in Figure 100 below:

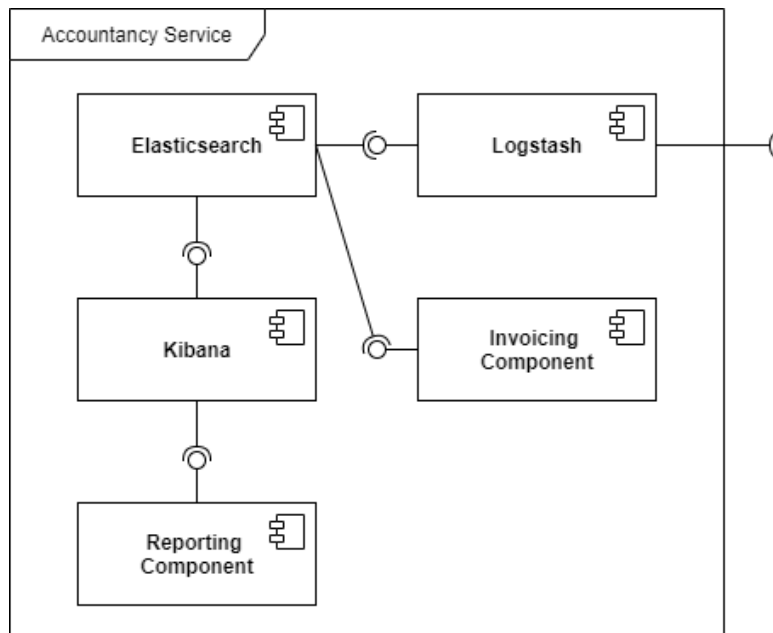


Figure 100. UML Component Diagram of the Accountancy Service

### Accountancy Service's HTTP REST API for Log Persistence

The Accountancy Service provide a single HTTP REST endpoint to capture user logs. This endpoint is registered to the Data Spine Service Registry so that it can be discovered from various components of EFPF ecosystem. For the integration, all there is needed to do is to send a POST request to the Logstash endpoint with a JSON body related with the related action. This will be enough for the Accountancy Service to capture the data and update the dashboards. Additionally, this simplicity also allows Accountancy Service to be integrated with an unlimited number of external marketplaces.

REST Endpoint	HTTP Method	Description
/logstash	POST	Persists logs to the Elasticsearch after executing data ingestion pipelines

Figure 101: Accountancy Service Log Persistence Endpoint

Furthermore, following event types can be tracked by the Accountancy Service, and they must conform the presented data model so that Kibana dashboards can be updated automatically:

**LOGIN**

This event is sent to Logstash when a user logs in to the EFPF Portal.

LoginEvent
action: string
userId: string
platform: string

Figure 102. Accountancy Service Login Event Data Model

The attributes of the LoginEvent object are described below:

- action: Default "LOGIN" value for identifying login events among all the data kept on Elasticsearch
- userId: Identifier of the user assigned by the platform
- platform: Default "EFPF" value as the name of the platform

**REGISTER\_COMPANY**

This event is sent to Logstash when a new company is registered to the EFPF Portal.

RegisterCompanyEvent
action: string
companyId: string
platform: string

Figure 103. Accountancy Service Company Registration Event Data Model

The attributes of the RegisterCompanyEvent object are described below:

- action: Default “REGISTER\_COMPANY” value for identifying company registrations among all the data kept on Elasticsearch
- companyId: Identifier of the company assigned by the platform
- platform: Default “EFPP” value as the name of the platform

## REGISTER\_USER

This event is sent to Logstash when a user registers to the EFPP Portal.

RegisterUserEvent
action: string
userId: string
companyId: string
platform: string

Figure 104. Accountancy Service User Registration Event Data Model

The attributes of the RegisterUserEvent object are described below:

- action: Default "REGISTER\_USER" value for identifying user registrations among all the data kept on Elasticsearch
- userId: Identifier of the user assigned by the platform
- companyId: Identifier of the company that the user belongs to
- platform: Default “EFPP” value as the name of the platform

## PLATFORM\_VISIT

This event is sent to Logstash when a user visits a base platform through the EFPP Portal.

PlatformVisitEvent
action: string
userId: string
platform: string
visitedPlatform: string

Figure 105. Accountancy Service Platform Visit Event Data Model

The attributes of the PlatformVisitEvent object are described below:

- action: Default “PLATFORM\_VISIT” value for identifying platform visit events among all the data kept on Elasticsearch
- userId: Identifier of the user visiting the platform
- platform: Default “EFPP” value as the name of the platform
- visitedPlatform: The name of the platform visited by the authenticated user

## TOOL\_VISIT

This event is sent to Logstash when a user visits a tool/service offered by the EFPF platform.

ToolVisitedEvent
action: string
userId: string
platform: string
visitedTool: string

Figure 106. Accountancy Service Tool/Service Visit Data Model

The attributes of the ToolVisitEvent object are described below:

- action: Default "TOOL\_VISIT" value for identifying tool/service visit events among all the data kept on Elasticsearch
- userId: Identifier of the user visiting the tool/service
- platform: Default "EFPF" value as the name of the platform
- visitedTool: The name of the tool/service visited by the authenticated user

## SEARCH\_EVENT

This event is sent to Logstash when a user searches for products/services or companies on EFPF Portal.

SearchEvent
action: string
userId: string
query: string
searchType: string
queriedPlatforms: string
platform: string

Figure 107. Accountancy Service Search Event Data Model

The attributes of the SearchEvent object are described below:

- action: Default "SEARCH\_EVENT" value for identifying product/service/company search events among all the data kept on Elasticsearch
- userId: Identifier of the user performing the search operation
- query: The search keyword
- searchType: Depending on the type of the search, it can be either COMPANY\_SEARCH or PRODUCT\_SEARCH
- queriedPlatforms: Names of the target platforms that the current search operation executed on
- platform: Default "EFPF" value as the name of the platform

## PAYMENT

This event is sent to Logstash when a user purchases products/services on base marketplace if the user initiated his/her journey from EFPF Portal.

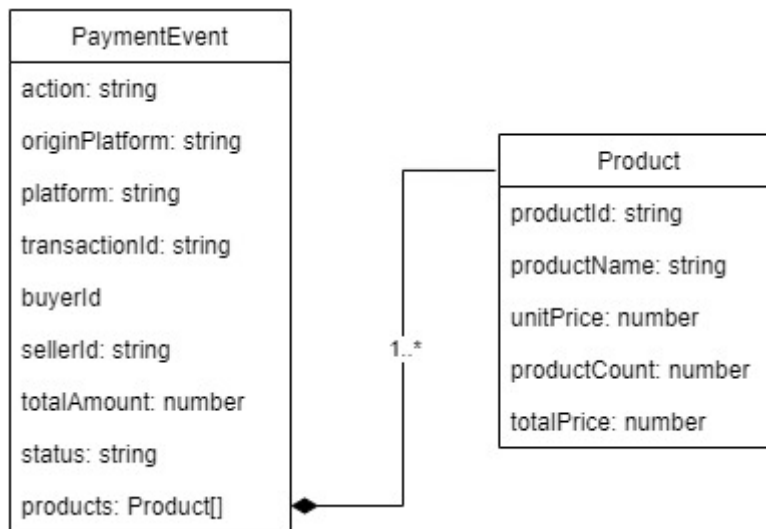


Figure 108. Accountancy Service Payment Event Data Model

The attributes of the PaymentEvent object are described below:

- action: Default "PAYMENT" value for identifying payment events among all the data kept on Elasticsearch
- originPlatform: If the user is coming from EFPF platform, "EFPF" constant value must be used since this is essential for the cashback mechanism to work, otherwise the value will be null or this field is omitted
- platform: Name of the platform that realizes the transaction: NIMBLE, SMECluster, VF-OS, etc.
- transactionId: If available. Indicates the identifier of this transaction.
- buyerId: If available. Indicates the identifier of the buying company/user on the platform that realizes the transaction
- sellerId: If available. Indicates the identifier of the selling company/user on the platform that realizes the transaction
- totalAmount: The total amount paid by the buyer company/user. Please note that, this field should be a numeric value
- status: Status of the payment and can be "pending", "completed" or "cancelled"
- products: List of products purchased
  - productId: Id of the product
  - productName: Name of the product
  - unitPrice: Price of the single product
  - productCount: Number of the current products purchased in this transaction
  - totalPrice: Total price of products (unitPrice x productCount)

### Accountancy Service's GUI (Kibana Dashboards)

Accountancy Service provides different predefined dashboards with charts to display user interactions with different parts of the EFPF Ecosystem. The charts in each dashboard visualizes specific information and allows users to interact with the dashboards. For example, any data displayed on charts can be selected to filter other charts so that a detailed user behaviour analysis can be performed. Moreover, Kibana dashboards also enable users to display data in different predefined periods (Last 1 week, Last 90 days, etc.) or absolute time intervals. Examples of Kibana dashboards can be seen below in Figure 109, Figure 110, Figure 111, Figure 112:

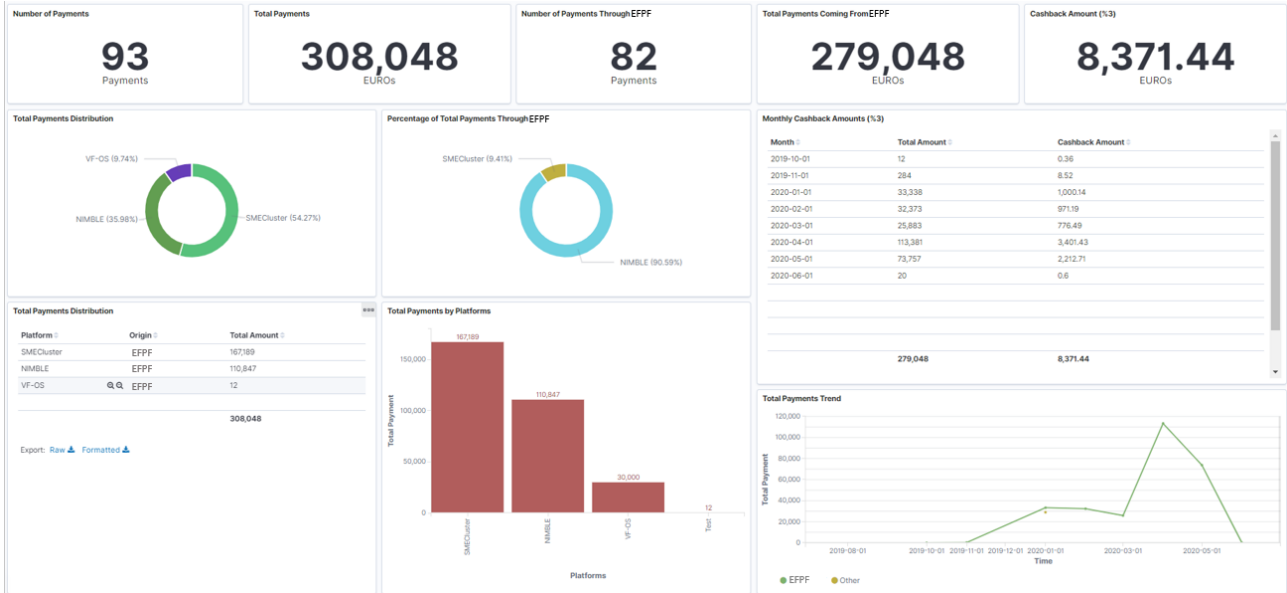


Figure 109. Accountancy Service - Payments Dashboard

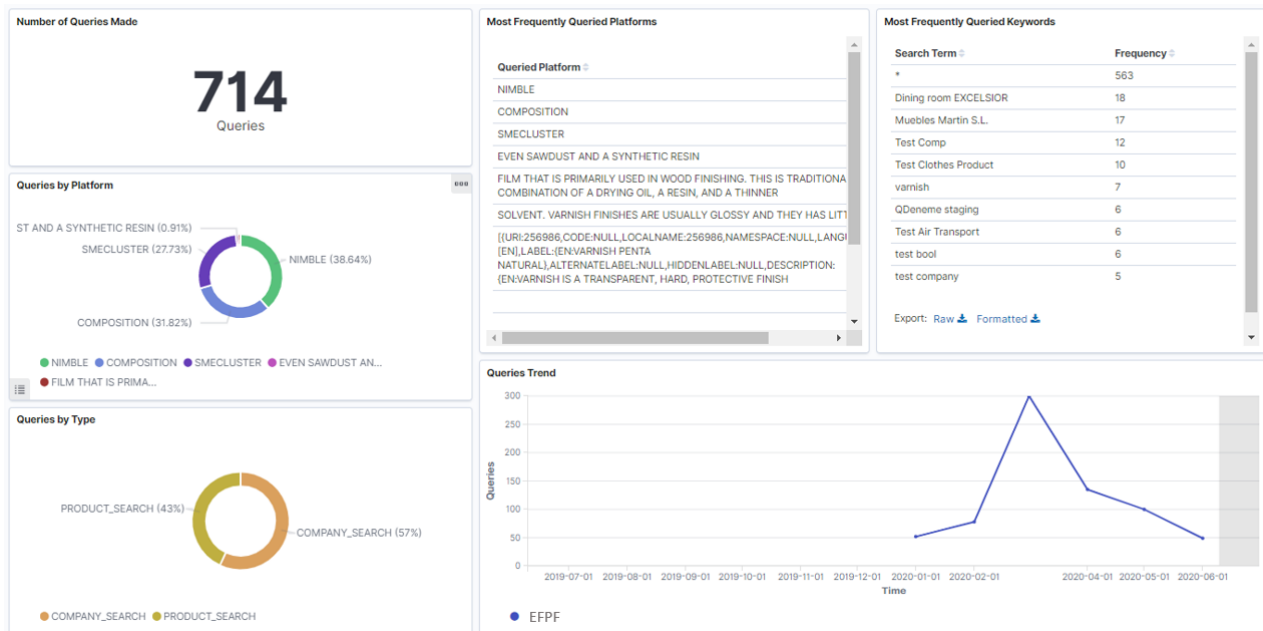


Figure 110. Accountancy Service Marketplace Usage Dashboard

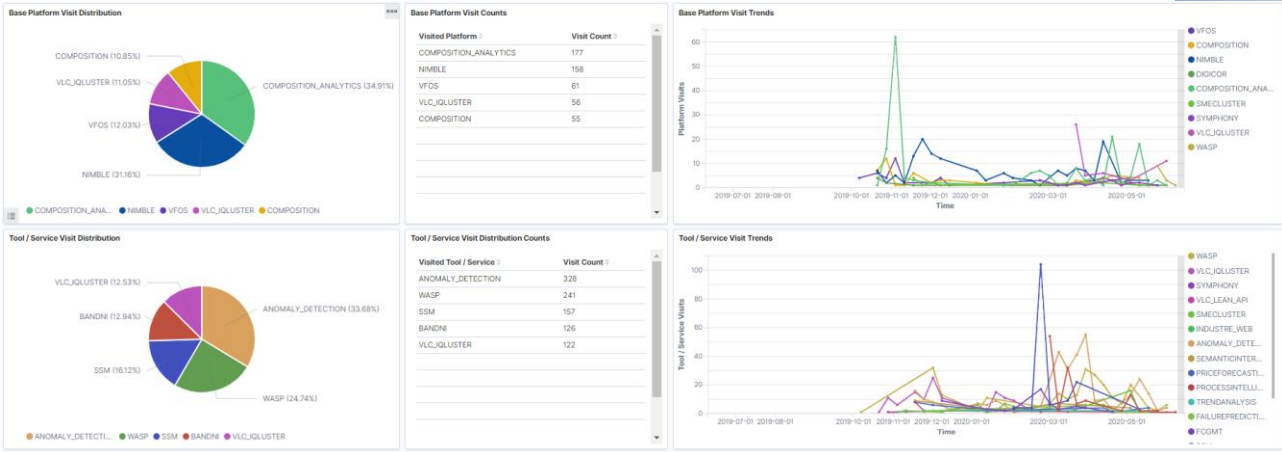


Figure 111. Accountancy Service - Platform Engagement Dashboard



Figure 112. Accountancy Service - User Activities Dashboard

### 3.2.1.4 Pub/Sub Security Service

The Pub Sub Security Service was based on and extended from early developments in the EFPF Project, namely, the Factory Connector Gateway Management Tool (FCGMT), and the Factory Connector Gateway – API (FCG-API).

The Pub Sub Security Service is the entry point for interaction with the DS Message Bus provided by the EFPF platform. The Service consists of three core components, the Pub Sub Backend, Pub Sub Frontend and Pub Sub Database. The overall architecture has been illustrated in Figure 113. The Pub Sub Security Service provides a front-end component that allows for the management of resources, topics, and permissions, within an intuitive GUI. The Pub Sub Backend component then acts as the main point interaction point for the frontend tool, with the provision of a HTTP REST API, following Open API Specification 3.0. The backend component is then responsible for interaction with other components, including authentication of the requests with the EFS, executing configuration operations in RabbitMQ and persisting the data in the Pub Sub Database, the final component of the Pub Sub Security Service. Finally access to the Pub Sub Security Service is secured through the EFPF Portal and EFS.

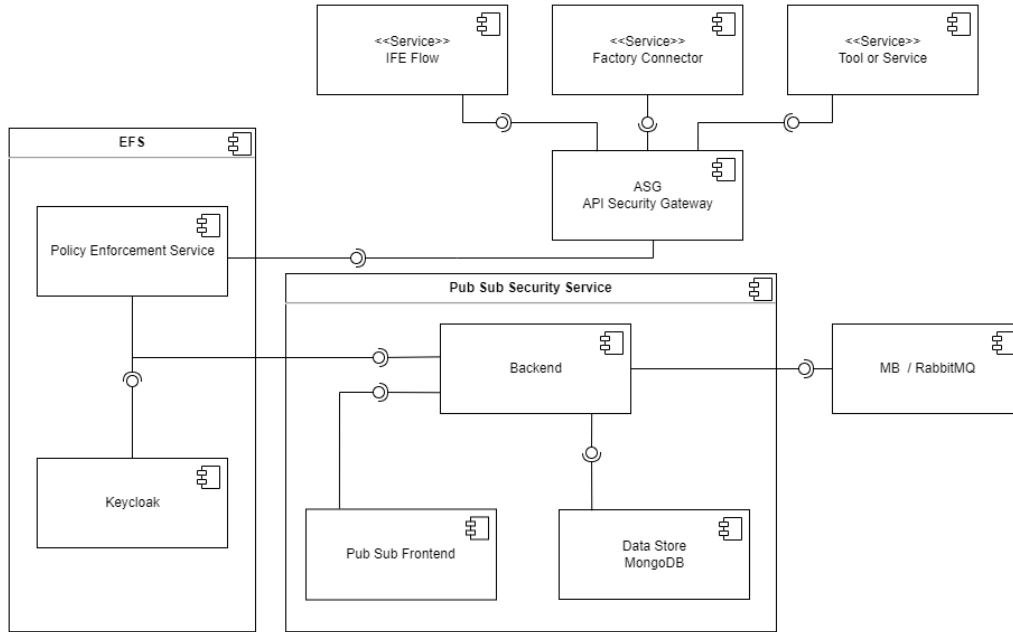


Figure 113. Pub Sub Security Service: Architecture Diagram

### Pub Sub Backend REST API:

Figure 114 has been included below to provide greater insight into the HTTP REST API provided by the Pub Sub Backend REST API.

REST Endpoint	HTTP Method	Description
/authentication	POST	Synchronise EFS and RMQ accounts
/resource	GET	Retrieves list of registered "Resources" accessible to requester.
/resource	POST	Registers a new 'Resource'
/resource/id/{resourceId}	GET	Retrieves a 'Resource' object
/resource/user-created	GET	Retrieves list of 'Resources' created by the requester.
/resource/user-created/publishing	GET	Retrieves list of 'Resources' created by the requester, and in which they have publish permission to a contained 'Topic'.
/resource/user-accessible	GET	Retrieves list of 'Resources' accessible to the user (they own or can request permission to consume)
/resource/user-accessible/publishing	GET	Retrieves list of 'Resources' accessible to the user, and in which they have publish permission
/resource/{resourceId}	PUT	Updates the existing 'Resource' object
/resource/{resourceId}	DELETE	Deletes the 'Resource' object
/pub-sub/user/vhost-permissions	GET	Retrieves list of vhost permission for user
/pub-sub/topic	POST	Creates new 'Topic' for registered 'Resource' to publish to
/pub-sub/topic	GET	Retrieves list of 'Topics' accessible to the user



/pub-sub/topic/{topicId}	GET	Retrieves 'Topic' object
/pub-sub/topic/{topicId}	DELETE	Deletes 'Topic' object
/pub-sub/topic/user/permitted/publishing	GET	Retrieves list of 'Topics' that a user has permission to publish to.
/pub-sub/topic/user/permitted/consuming	GET	Retrieves list of 'Topics' that a user has permission to consume.
/pub-sub/topic/user/managed/publishing	GET	Retrieves list of 'Topics' owned by the user in which they have publish permission
/pub-sub/topic/user/managed/consuming	GET	Retrieves list of 'Topics' owned by the user in which they have consume permission
/pub-sub/topic/user/requests	GET	Retrieves list of 'Topics' in which a new permission request is present, requiring approval or rejection from the 'Topic' owner
/pub-sub/topic/user/approved	GET	Retrieves list of 'Topics' in which the topic owner (requester) has approved other resource to consume.
/pub-sub/permission/request/{topicId}	POST	Create new permission request to consume 'Topic'
/pub-sub/permission/approveRequest	POST	Approve permission request to consume 'Topic'
/pub-sub/permission/rejectRequest	POST	Reject permission request to consume 'Topic'
/pub-sub/permission/revokeRequest	POST	Revoke previously approved permission request to consume 'Topic'
/pub-sub/credentials/consume	POST	Retrieve credentials and configuration details required to consume a given 'Topic'
/pub-sub/credentials/produce	POST	Retrieve credentials and configuration details required to publish to given 'Topic'

Figure 114: Pub Sub Security Service: HTTP REST API's

Figure 115 illustrates the data model of the Pub Sub Security Service.

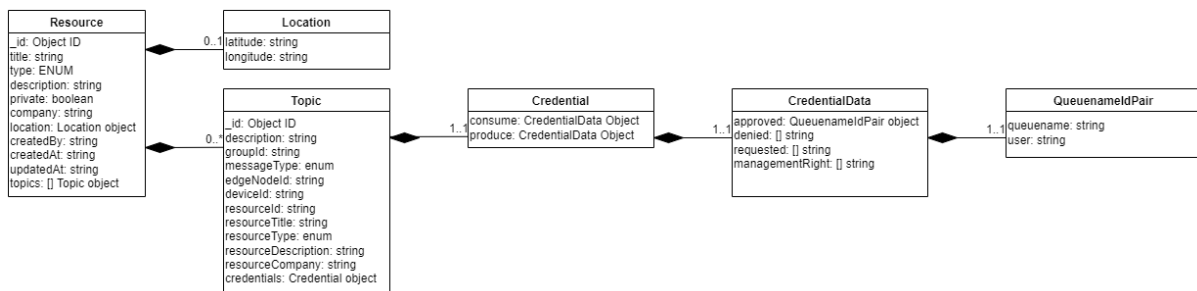


Figure 115. Pub Sub Security Service: Data Model

The attributes of the data model are described below:

Resource:

- `_id`: Unique Object ID of the Resource
- `title`: A title or name of the Resource
- `type`: The type of Resource (Tool, Service, Factory Connector, IFE Flow)

- description: A description of the Resource
- private: Indicates public (discoverable) or private (non-discoverable)
- company: Company name of Resource owner
- location: Geo Location of the Resource
- createdBy: Resource owner or creator
- createdAt: ISO 8601 timestamp of the Resource creation
- updatedAt: ISO 8601 timestamp of the last update to Resource
- topics: An array of Topic objects

Location:

- latitude: Geographic coordinate reference of Resource location
- longitude: Geographic coordinate reference of Resource location

Topic:

- \_id: Unique Object ID of the Resource
- description: A description of the Topic
- groupId: First logical grouping of the Topic name
- messageType: Type of message being sent
- edgeNodeId: Second logical grouping of the Topic name
- device\_id: Third logical grouping of the Topic name
- resourceId: Id of the Topic's associated Resource
- resourceTitle: Title of the Topic's associated Resource
- resourceType: The type of the Topic's associated Resource
- resourceDescription: The description of the Topic's associated Resource
- resourceCompany: The company name of the Topic's associated Resource
- credentials: The credentials object containing lists of user accounts which have; management rights to the Topic, been approved to consume or publish to the Topic, had permission requests rejected, and which have requested permission to consume the Topic

Credential:

- consume: CredentialData object containing lists of user accounts which have requested permission to consume the Topic, have had permission requests rejected, have been approved to consume the Topic, or have management rights to the Topic.
- produce: CredentialData object containing lists of user accounts which have requested permission to publish to the Topic, have had permission requests rejected, have been approved to publish to the Topic, or have management rights to the Topic.

CredentialData:

- approved: An array of QueuenameldPair objects listing the approved user account and their created queue name.
- denied: An array of user account id's that have been denied permission to the topic
- requested: An array of user account id's that have requested permission to the topic
- managementRight: An array of user account id's that have management rights to the topic

**QueuenameldPair**

- queue name: name of queue bound to the amq.topic exchange with the topic name as routing key
- user: User account id that has been approved to consume the topic

**Pub Sub Security Topic Naming Convention:**

Alongside the full data model listed above, the pub sub security service has also followed the Sparkplug topic naming convention for the standardised definition of topics within the EFPF platform. The Sparkplug naming convention implemented has been included in Figure 116.

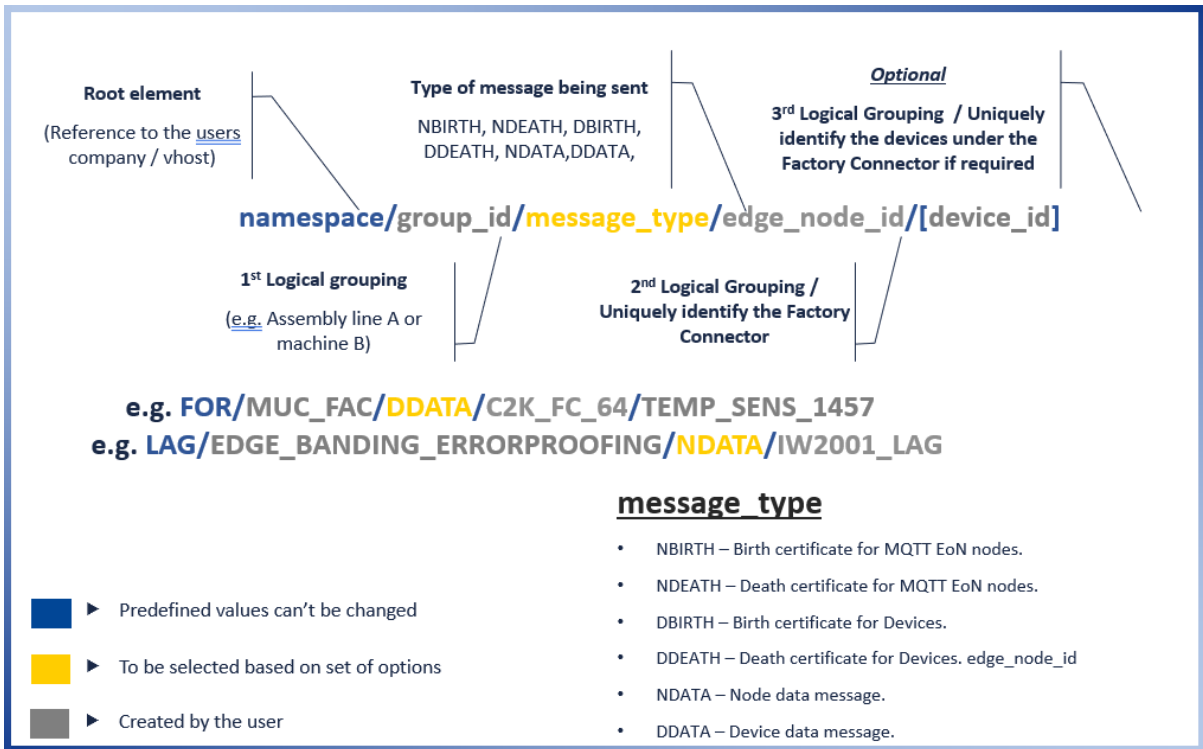


Figure 116. Pub Sub Security Service: Sparkplug Topic Naming Convention

**Pub Sub Security Service GUI for Topic and Permissions Management**

To enable the easy registration and management of resources, topics and permission, the Pub Sub Security Service provides a GUI as illustrated in Figure 117 to Figure 124.

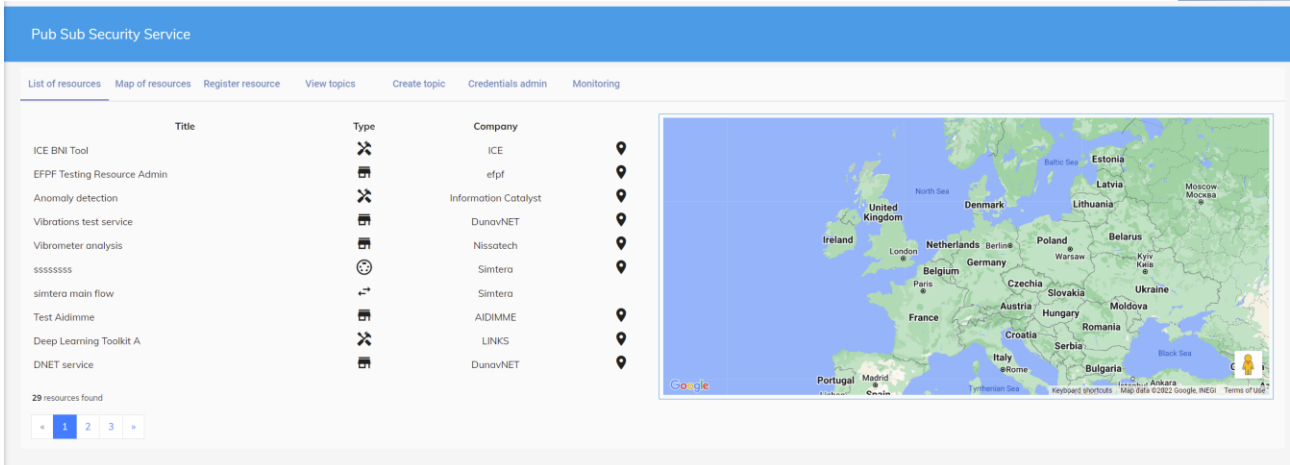


Figure 117. Pub Sub Security Service GUI: List of Resources Page

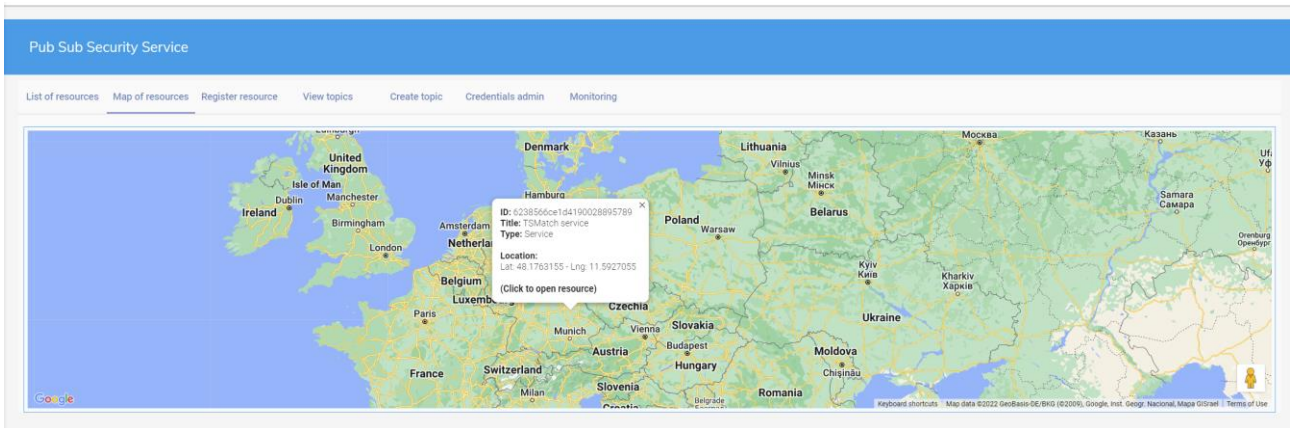


Figure 118. Pub Sub Security Service GUI: Map of Resources Page

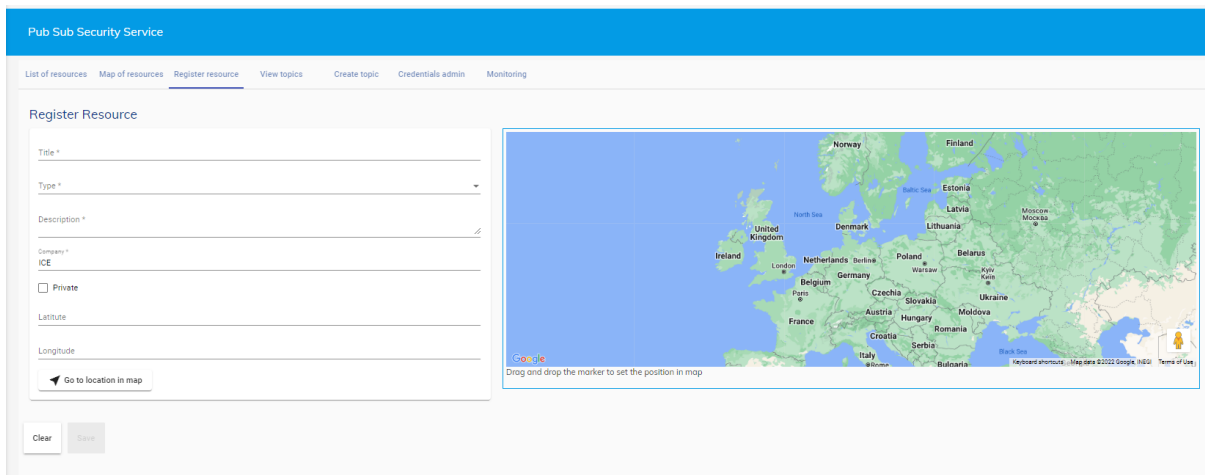


Figure 119. Pub Sub Security Service GUI: Register Resource Page

Pub Sub Security Service

List of resources | Map of resources | Register resource | **View topics** | Create topic | Credentials admin | Monitoring

### View Topics

Search By Keyword

Topic					Resource				Credentials	
GroupID	Message Type	Edge Node ID	Device ID	Description	Name	Type	Description	Company	Consume	Publish
123	NBIRTH	456	01		Label Maker 5000	Tool	1 of 2	IT Innovation Centre	<a href="#">Request</a>	⊙
765	DBIRTH	977	02		Label Maker 5000	Tool	1 of 2	IT Innovation Centre	<a href="#">Request</a>	⊙
456	NDEATH	987	01		Wood Cutter	FactoryConnector	Timber	IT Innovation Centre	<a href="#">Request</a>	⊙
787	DDEATH	676	01	Proximity Sensor	Wood Cutter	FactoryConnector	Timber	IT Innovation Centre	<a href="#">Request</a>	⊙
675	DDEATH	120	03		ICE Machine	Tool	Crushed ice	IT Innovation Centre	<a href="#">Request</a>	⊙
988	DDEATH	565	01		ICE Machine	Tool	Crushed ice	IT Innovation Centre	<a href="#">Request</a>	⊙
020399	NBIRTH	009	01		ICE Machine	Tool	Crushed ice	IT Innovation Centre	<a href="#">Request</a>	⊙
120	DDATA	049	02		Demo Resource	FactoryConnector	Demo ABC	IT Innovation Centre	<a href="#">Request</a>	⊙

Figure 120. Pub Sub Security Service GUI: View Topics Page

View Topics

Search By Key

#### Credential

```

{
  "userId": "test-user-1-efpf-admin@efpf.org",
  "password": "██████████",
  "vhost": "efpf-org",
  "exchange": "amq.topic",
  "exchangeType": "topic",
  "routingKey": "efpf-org.TEST_FACTORY_Y.DDATA.TEST_MOTION_SENSOR_1",
  "host": "efpf.smecluster.com",
  "amqpsPort": "5671",
  "mqttpsPort": "8883",
  "amqpsUri": "██████████",
  "mqttpsUri": "██████████"
}

```

Close

Figure 121. Pub Sub Security Service GUI: Credentials & Configuration Details Pop-Up

Figure 122. Pub Sub Security Service GUI: Create Topic Page

New Requests												
Search By Keyword												
Topic					Resource				Credential Request			
GroupID	Message Type	Edge Node ID	Device ID	Description	Name	Type	Description	Company	Credential	Requested By	Approve	Reject
routingefficiencies	DDATA	kmeans	analysis		Anomaly Detection Tool	Tool	ICE Instance of the Anomaly Detection Tool	ICE	Consume	ross.campbell@informationcatalyst.com	✓	✗

Approved												
Search By Keyword												
Topic					Resource				Approved Credential			
GroupID	Message Type	Edge Node ID	Device ID	Description	Name	Type	Description	Company	Credential	User	Revoke	Admin
Nothing found												

Figure 123. Pub Sub Security Service GUI: Credentials Admin Page

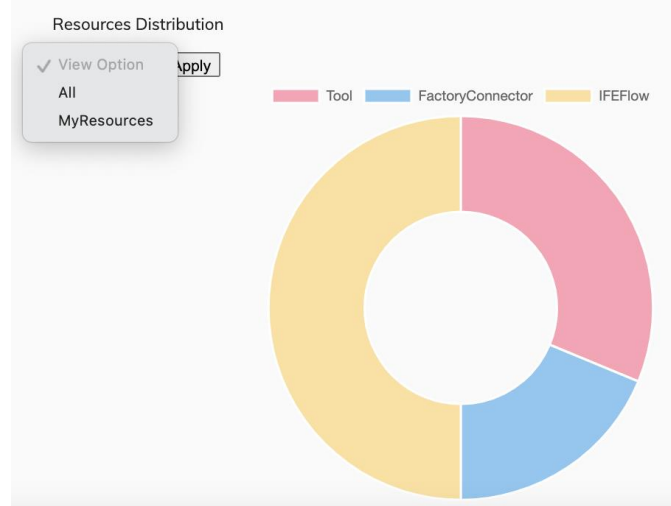


Figure 124. Pub Sub Security Service GUI: Monitoring Page

### Pub Sub Security Service: GUI Usage Steps

To illustrate the steps required to be performed in the Pub Sub Security Service GUI in order to publish or subscribe to the EFPF Message Bus, the service’s quick start guide has been included, as shown in Figure 125 and Figure 126.

**Publish Operations**

Design Time Activities to Enable Publish Operations

- To use the DS RabbitMQ for Publish operations (e.g., using MQTTs), you need to get a **vhost** for your company and a DS RabbitMQ user account under that vhost.
- Get a new vhost: Once you visit the [Pub Sub Security Service dashboard](#) in the EFPF Portal, a new vhost for your company is automatically created.
- vhost name: The name of the vhost is derived from your email Id - its the string following the @ symbol, with .’s replaced with -’s.
  - e.g. for user@companyx.com: Vhost = companyx-com
  - e.g. for user@department1.companyz.com: Vhost = department1-companyz-com
  - Exceptions:
    - user@gmail.com: Vhost = efpf-open-call-vhost
    - user@outlook.com: Vhost = efpf-open-call-vhost
- Register a resource: Next, you need to register a resource (tool/iFlow), say 'tool1', that intends to publish data. You can register tool1 using the [Register Resource](#) page.
- Create a topic: For tool1, you need to create a new topic using the [Create Topic](#) page.
- Topic name: The topic name would be of the form: <vhost name>/<Group ID>/<Message Type>/<Edge Node ID>/<Device ID, if any>. Example: companyx-com/FACTORY\_Y/DDATA/FC\_8/MOTION\_SENSOR\_1
- Obtain the credentials: Search for your topic on the page [View Topics](#), locate it, and obtain your DS RabbitMQ credentials from the [Credentials](#) > [Publish](#) tab, by clicking on the |>| button.
 

**NOTE:** The **"routingKey"** provided on the Credentials Pop Up, should be used as the **Topic Name** directly.
- Publish: Configure tool1 to use these credentials and publish to DS RabbitMQ over topic companyx-com/FACTORY\_Y/DDATA/FC\_8/MOTION\_SENSOR\_1 and start publishing data.

Example:

```
mosquitto_pub -h efpf.smecluster.com -p 8883 -u 'companyx-com:user1@companyx.com' -P 'replace_this_with_password' -t 'companyx-com/FACTORY_Y/DDATA/FC_8/MOTION_SENSOR_1' --cafile /c/Users/user1/Downloads/cacert-2020-07-22.pem -m test_message_1
```

Grant other users permission to Subscribe to your topics

- View the pending requests: Go to the "New Requests" section on the [Credentials admin](#) page to view the pending requests.
- Approve: Locate the request and approve it by clicking on ✓ in the [Credential Request](#) > [Approve](#) tab.

Figure 125. Pub Sub Security Service: GUI Usage Steps: Publish Operations

### Subscribe Operations

Design Time Activities to Enable a Subscribe Operation

- Let's say, your EFPF username is `user2@example.org`, and you want to subscribe to the `companyx-com/FACTORY_Y/DDATA/FC_8/MOTION_SENSOR_1` topic from the `companyx-com` vhost.
  - Discover the topic to subscribe to: Search for the topic that you want to subscribe to on the [View Topics](#) page.
  - Request access: Click on the (I) button in the **Credentials** > **Consume** tab next to the topic to request Subscribe access from the topic owner. The button will change to (...) pending state.
  - Wait: Wait for the topic owner to approve the request.
  - Get the credentials: Once the request is approved, the button will change to |>. Click on it to obtain the credentials.
- NOTE:** The "**routingKey**" provided on the Credentials Pop Up, should be used as the **Topic Name** directly.
- Subscribe: Configure your tool/iFlow to use these credentials and subscribe to the topic.

Example:

```
mosquitto_sub -h efpf.smecluster.com -p 8883 -u 'companyx-com:user2@example.org' -P 'replace_this_with_password' -t 'companyx-com/FACTORY_Y/DDATA/FC_8/MOTION_SENSOR_1' --cafile /c/Users/user2/Downloads/cacert-2020-07-22.pem
```

Figure 126. Pub Sub Security Service: GUI Usage Steps: Subscribe Operations

### 3.2.1.5 Factory Connectors & IoT Gateways

In order to utilise the functionality of the tools and services data from the numerous data sources available within, the manufacturing environment needs to be made available. To achieve this, it is necessary to utilise a Factory Connector or IoT Gateway that can interface with the specific devices, sensors, and systems the user wishes to gather data from. The connectors and gateways then communicate with the EFPF Data Spine using a predefined data model that relates to the nature of the data and its application.

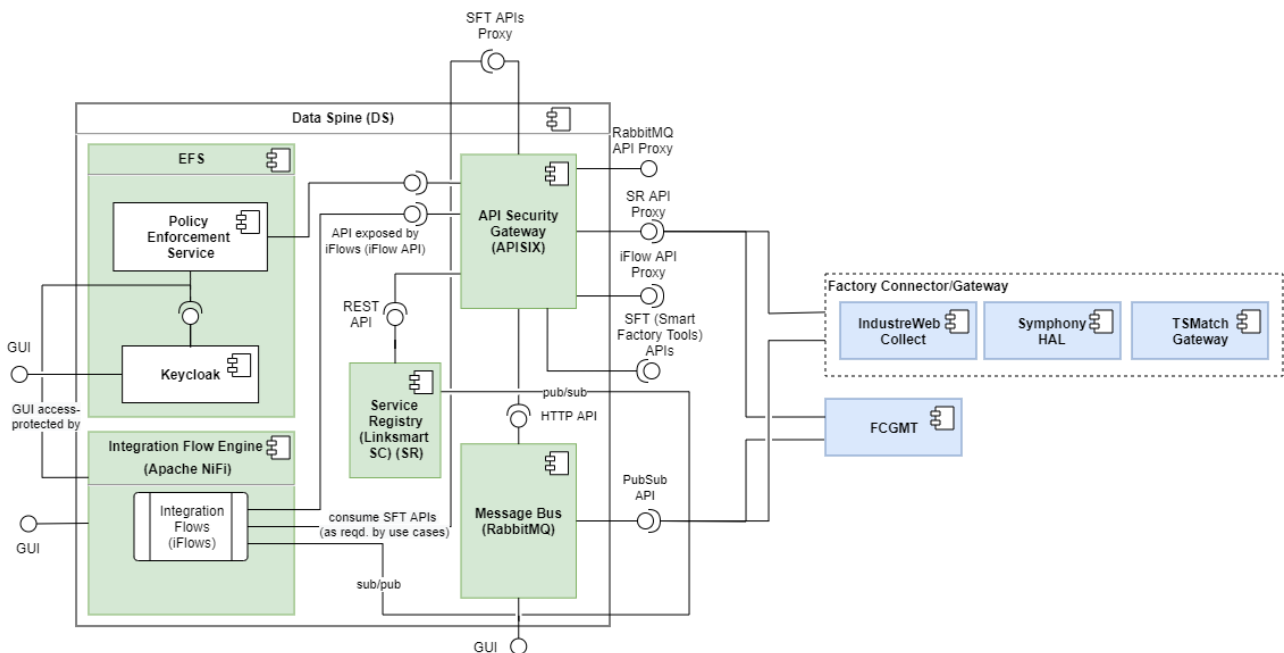


Figure 127. Interaction between the Factory Connectors and the EFPF Data Spine

In EFPF there are three implementations of Factory Connectors & Gateways, supporting between them the most widely used industry standards and systems (e.g., OPC UA, Modbus, Zigbee and propriety PLC protocols from Siemens, Rockwell, Omron, Schneider). The architecture, however, supports future compliant Connectors and Gateways to be supported.



Figure 128 shows the high-level architecture of Connectors / Gateways and their interaction with the EFPF Data Spine.

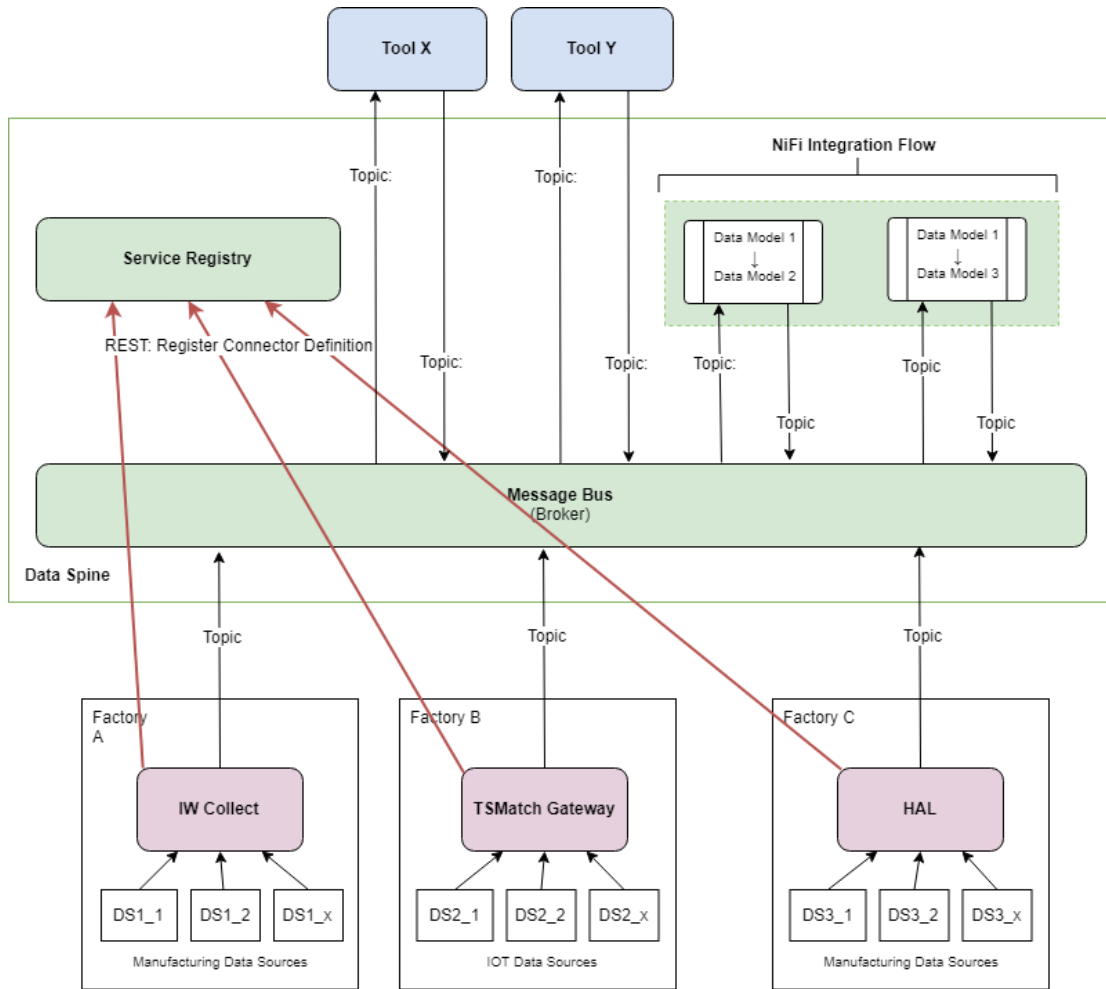


Figure 128. Factory Connectivity within EFPF

Each Connector or Gateway has its own functionality set supporting different connectivity options, protocols, and architecture, whilst offering different functions such as data thresholds the ability to make a calculation or scaling the data values. The interface with the EFPF Data Spine is a common interface, so in terms of the data presented to the tool, the versions of connector or gateway are not important.

The communication with the Data Spine is achieved via MQTT Pub/Sub protocol which allows data to be published from the factory on a specific topic name when the data changes. The Data Spine Broker component then routes the data to any Tool or Service that has been granted access to subscribe to that data. This is more efficient than polling the Connector or Gateway at the interval and presents fewer firewall security concerns than accepting inbound polling requests.

The following sections will describe each of the existing Factory Connectors and Gateways and their functionality, followed by the management tool used to configure the Pub/Sub communications.

### 3.2.1.5.1 Industweb Collect

Industweb (IW) Collect is a high-speed data engine that interfaces with industrial data sources via an architecture that uses discrete software connectors. Data sources can

include industrial control systems (e.g., PLC, CNC) both modern models with Ethernet capability and legacy equipment, wireless networks, and devices such as ZigBee, and industrial networks such as Profinet, Profibus, Modbus and AS-interface. It can also connect and interrogate databases such as MS SQL and MySQL, and flat file formats such as XML and JSON.

The architecture of the IW Collect is shown in Figure 129, the connector instances enable monitoring of a diverse range of data sources by interfacing with the production systems and sensors. IW Collect runs on an embedded industrial computer platform that can be pre-installed on machines, or retro-fitted to existing machines and is based on Microsoft .Net with an option of using Microsoft SQL Server in order capture data locally.

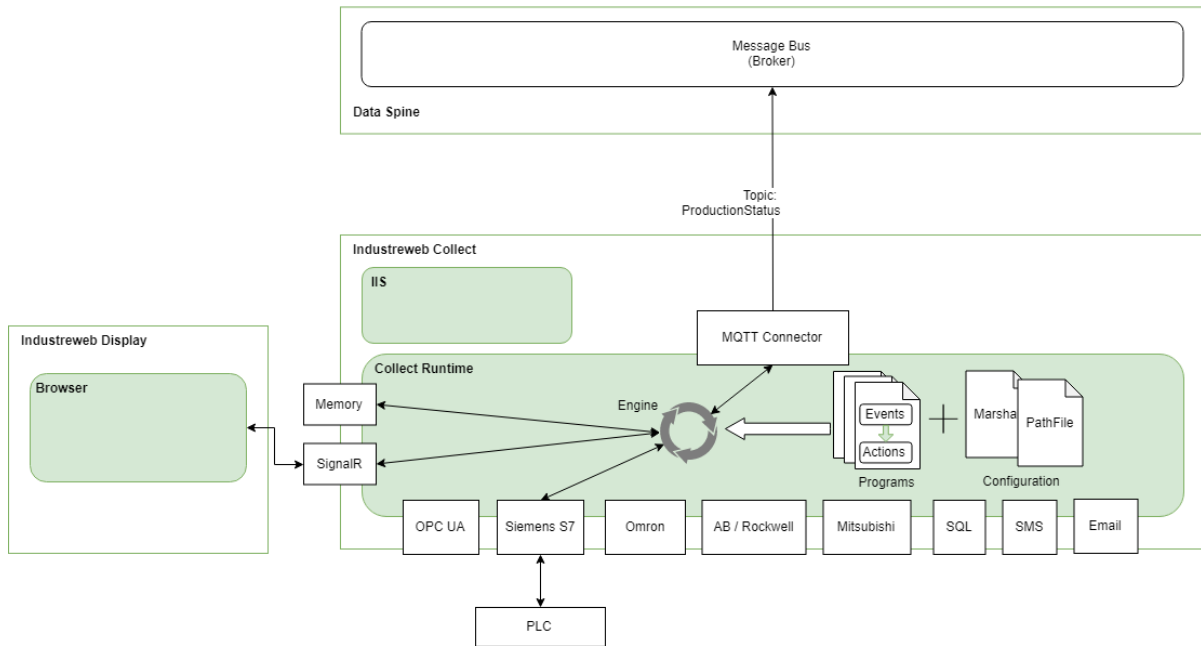


Figure 129. Industweb Collect Architecture

Industweb Collect does not expose an API however it consumes multiple API's and protocols through connectors, and also communicates with the Industweb Global API to determine if its configurations are up to date and if not to request a new set of configurations files. It does so via the following API calls which are described in section 3.2.1.18.

Data is published from the Collect Factory Connector to the Data Spine Message Bus using MQTT. The Topic format complies with Sparkplug specifications e.g.:

```
IW/MTF-Main-Demo/NDATA/OMRON_BOTTLING/OMRON_PART_COUNT
```

The message payload is in JSON format and includes timestamp data type, name of the data point and one or more associated Readings:

```
{
  "TagType": "IntTag",
  "Name": "OMRON_PART_COUNT",
  "DataType": "Int",
  "IOType": "In",
  "Units": "",
  "PublishedTimeStamp": "2022-06-11T03:03:02.7986299Z",
```

```

"Parameters": {},
"Readings": [
  {
    "TimeStamp": "2022-06-11T03:02:01.0490737Z",
    "Value": 0,
    "Quality": 1.0
  },
  {
    "TimeStamp": "2022-06-11T03:03:02.7986299Z",
    "Value": 4230,
    "Quality": 1.0
  }
],
"Min": 0,
"Max": 9321,
"Avg": 2115.0
}

```

### 3.2.1.5.2 TSMatch Gateway

TSMatch is an EFPF gateway module that supports semi-automated interconnection of IoT devices on a shopfloor to the EFPF Data Spine. For this, TSMatch relies on the following components, described next [NBN22]:

- **IoT Things.** IoT Things correspond to semantic descriptions of cyber-physical systems based on heterogeneous hardware, e.g., a sensor, a single-board computer (SBC), programmable logic controller (PLC) connected with sensors via its Input/output interface. The IoT Things have been modelled based on the OGC Sensor Thing API. The Thing description contains information such as the sensors properties and location. Advertisement is performed via Coaty, application-layer communication software. Coaty provides a way for IoT Things to advertise their semantic descriptions via multicast, notify subscribers when the IoT Thing is no longer available, by sending a deadadvertized event. When a discover event is sent by the TSMatch Engine to discover available Things, a response is sent containing the Thing description.
- **TSMatch engine.** The TSMatch engine corresponds to the server-side implementation engine that supports semantic matchmaking between IoT Things descriptions and service descriptions. The engine handles service requests transmitted via the TSMatch Client. It also handles queries to the TSMatch Thing registry and then handles the matchmaking between the properties/attributes of the Things descriptions to the semantic description of services. In TSMatch v1.0, the similarity computation is based on the Sørensen–dice coefficient and term frequency–inverse document frequency. In TSMatch v2.0, the matchmaking relies on NLP with a neural network model-based approach. Based on the matching, a set of available Things are selected, then grouped and an aggregated object representing this grouping is again stored in the database. matching result is then sent to the TSMatch Client. When a match is found, the engine triggers an event which starts a subscription to the observations of the selected sensors and calculates the mean value of the observation based on the selected location. The updates are then sent to the TSMatch Client. If a request is deleted, TSMatch

unsubscribes from the IoT Things observations (from a broker) and stops sending updates to the TSMATCH client. The TSMATCH Engine has been implemented in Node.js JavaScript runtime environment and Typescript language<sup>5</sup>. For the implementation of Sørensen–dice similarity we have relied on the Nebulous Plasma Muffin npm string-similarity package [NEB22] and on the natural package [NAT22].

- **TSMATCH client.** The TSMATCH Client handles the service semantic description, either by creating one based on specific requests from the user (e.g., monitor temperature) or by getting remote service descriptions. We have implemented the TSMATCH Client as a mobile application developed using the React Native JavaScript framework<sup>8</sup>. The client currently runs on Android 4.1 and above versions. When the TSMATCH client starts, it performs a subscription to an MQTT broker. It allows the user to see a list of available Things and their descriptions, to describe the requests, and to visualize the data updates based on the provided request.
- **Things Registry.** The registration, announcement, discovery, and communication from IoT Things to the end-user (subscriber) has been supported by Coaty v2.0. The communication is supported by an MQTT broker, based on Mosquitto v2.0.11. The IoT descriptions are stored in a local registry based on PostgreSQL 13.2 database to store data as JSONB data type for binary storage and retrieval of JSON objects. The Official PostgreSQL Docker image [POS22] has been used.

The TSMATCH sequence is illustrated in Figure 130. As illustrated, a first process relates with the IoT Things advertising their descriptions when powered on. TSMATCH Engine subscribes to Things events and registers their descriptions in the Things registry. This process is external to the TSMATCH semantic matchmaking engine and in our case is supported via Coaty.io [COA22]. The TSMATCH Engine can also request IoT Things descriptions directly from a mediating entity such as a broker. In other words, it can be configured as a subscriber. Via a separate process, a user performs a request for a service, e.g., measure temperature on a specific room using the TSMATCH client, which forwards the request to the TSMATCH Engine. Based on the request and the available IoT Things descriptions the TSMATCH selects from the registry a specific set of IoT Things to consider to answer the service, based on similarity matching. Upon success, the results of the matching are forwarded to the TSMATCH Client.

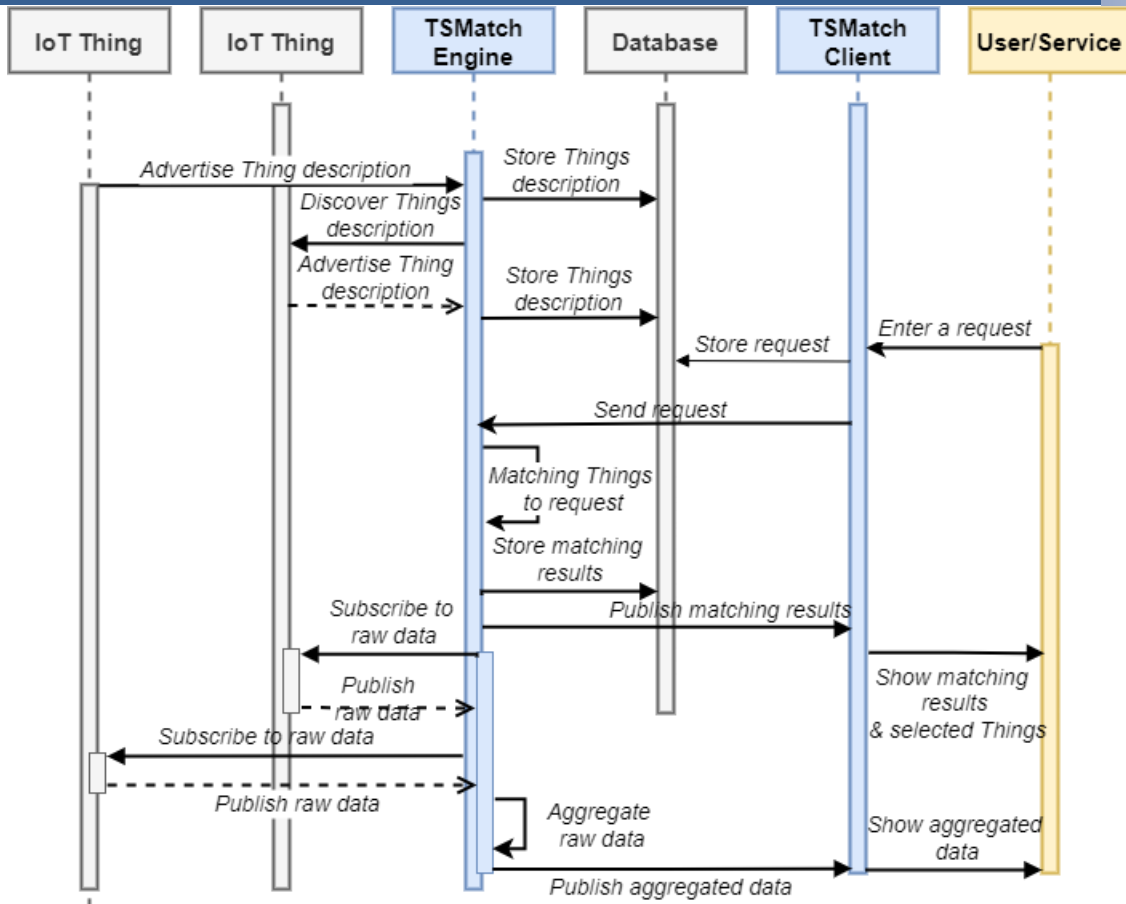


Figure 130. TSMatch Sequence Diagram

An illustration of the operation of TSMatch in EFPF is provided in Figure 131. The TSMatch Client is employed by the Symphony Factory Connector using service requests for IoT Things. A Cloud instance of the Symphony IoT automation platform ingests the information provided by TSMatch through the EFPF Data Spine which offers services with interoperable security features. The visual monitoring, sensor data and event storage, signal analysis, and alarm systems associated with the Things are provided by the Symphony internal modules, in particular: sensor data has been remotely collected through the Symphony HAL (a software module that primarily abstracts the low-level details of various heterogeneous fieldbus technologies and provides a common interface to its users) and stored by the Symphony Data Storage.

The Symphony Event Reactor handles events and alarms through combining information coming from different sources and data brokers (e.g., TSMatch) to determine actions to be taken including control actions on field-level devices, notifications (e-mails and SMS), and alarms (via stack light which provides visual and audible indications). The real-time data, along with the status of the thresholds and the alarms, can be also visualized and handled through the Symphony GUI in the Cloud instance of the platform.

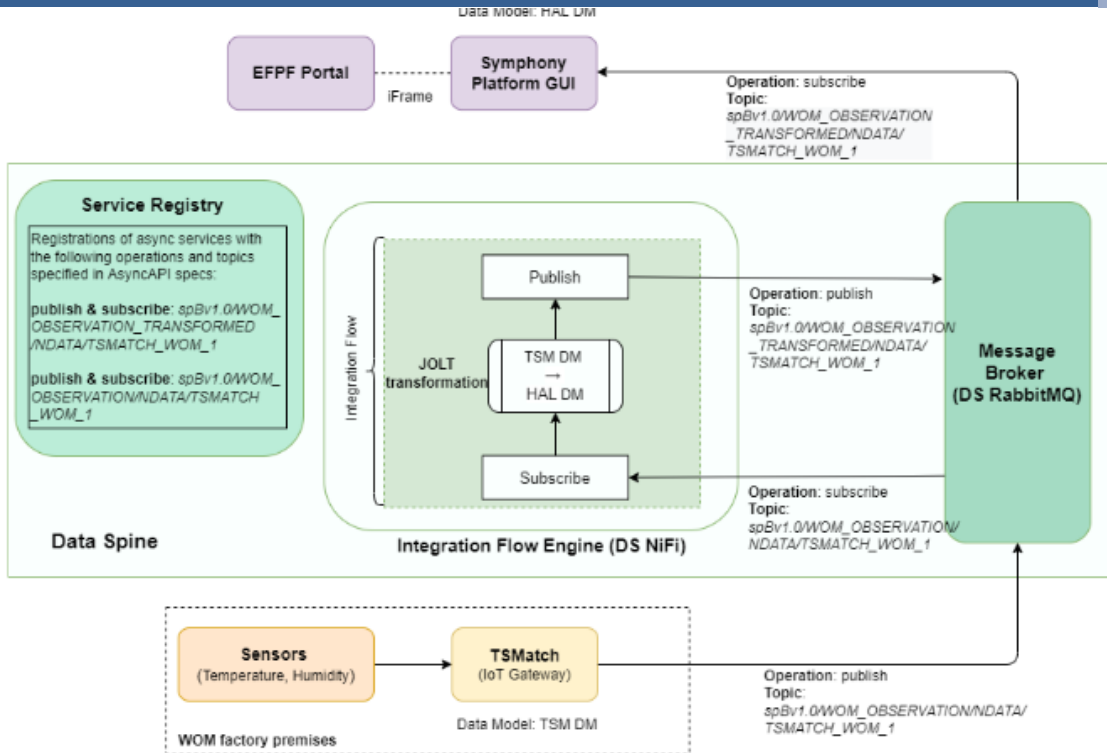


Figure 131. An illustration of the Operation of TSMatch in EFPF

To support the interconnection between TSMatch and the IoT Symphony platform, two MQTT topics were used: the first topic was used to send sensors observation from the TSMatch engine (server-side) to the EFPF Integration Flow Engine, where the data is transformed to the data model required by the IoT symphony platform using JOLT transformation. The second topic was used to share the transformed sensor observations with the Symphony platform. To ease integration, service registry is used to register the service descriptions using AsyncAPI 2.0 specs. Regarding the topic namespace, the following aspects have been integrated [BNO22]:

- edge\_node\_id has been defined as "TSMatch\_WOM\_1".
- message\_type has been set as "NDATA".
- group\_id has been set as "WOM\_OBSERVATION", assuming the interconnection between TSMatch and the EFPF NiFi Integration Flow Engine.
- group\_id is changed to "WOM\_OBSERVATION\_TRANSFORMED" for the topic used between the EFPF Ni-Fi Integration Flow Engine, and the Symphony platform.

### 3.2.1.5.3 Symphony Hardware Abstraction Layer (HAL)

The Symphony Hardware Abstraction Layer (HAL) is the software module responsible to abstract the low-level details of a set of sensors/actuators controlled by the SMBS. The HAL, as illustrated in Figure 132. interacts with the various devices using their own protocols, field buses and interfaces. On its northbound, the HAL provides a management interface, modelled as a REST API, to enable the definition of additional virtual objects and their configuration.

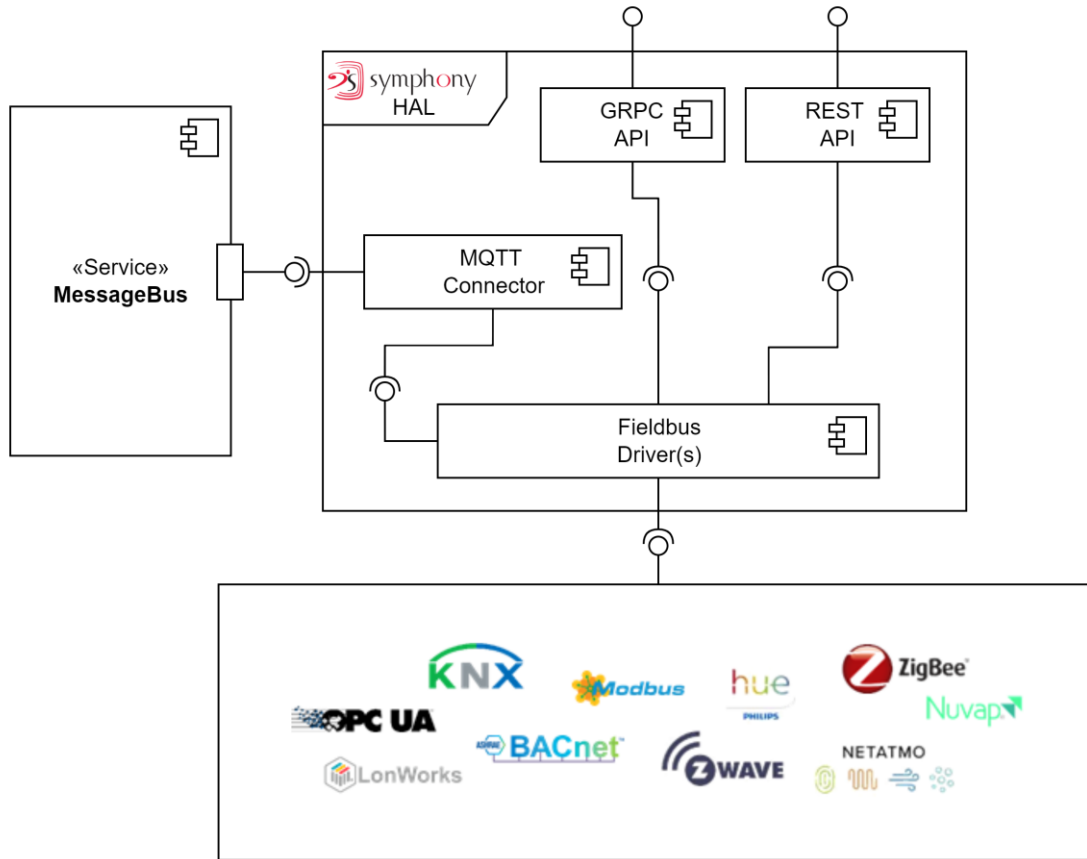


Figure 132. Symphony Hardware Abstraction Layer (HAL) Architecture

HAL supports KNX, BACnet, Modbus-TCP, Modbus-RTU as well as, several other proprietary control protocols. It can be extended by developing modules that can be plugged into its core. It can be interconnected with specific field buses either directly (via RS232/485 serial ports or GPIOs) or through the use of IP based gateways, such as KNX IP router and/or interface, Modbus/TCP gateways.

The HAL component provides access to any available resources (sensors and actuators) as datapoints. The datapoints are primitive objects with basic data type (int, float, boolean) but devoid of any semantic annotation (physical object type, measurement unit, ...) or are presented according to the OGC SensorThings data format standard. The HAL supports access via REST and gRPC and furthermore enables publish/ subscribe features via MQTT.

A detailed description of the configuration API for the Symphony HAL is described in Figure 133.

REST Endpoint	HTTP Method	Description
<b>/uhal</b>	GET	Returns the base information of the current HAL deployment (version, uptime etc)
<b>/uhal/restart</b>	POST	Restart the current HAL
<b>/uhal/wipe</b>	DELETE	Performs a HAL Factory Reset
<b>/uhal/backup</b>	GET	Dumps the current HAL configuration in JSON format
<b>/uhal/restore</b>	POST	Restores the HAL configuration from a JSON dump generated from GET /uhal/backup

/area	GET	Returns the list of the defined areas
/area	POST	Creates a new area. The configuration body is <pre>{   "aid": &lt;area_id&gt; }</pre>
/plugin	GET	Returns the configuration of all the HAL Plugin
/plugin	POST	Creates a new HAL Plugin. The configuration body is <pre>{   "area": &lt;area_id&gt;,   "name": "&lt;plugin_name&gt;",   "class": "&lt;plugin_type&gt;",   "config": {&lt;plugin_specific_configuration&gt;} }</pre>
/plugin/<name>	GET	Returns the configuration of a specific HAL Plugin
/plugin/<name>	PUT	Modifies the configuration of a specific HAL Plugin
/plugin/<name>	DELETE	Deletes a specific HAL Plugin
/datapoint	GET	Returns the configuration of all the datapoints
/datapoint	POST	Creates a new datapoint. The configuration body is <pre>{   "area": &lt;area_id&gt;,   "plugin": "&lt;plugin_name&gt;",   "config": {&lt;datapoint_specific_configuration&gt;} }</pre>
/datapoint/<id>	GET	Returns the configuration of a specific datapoint
/datapoint/<id>	PUT	Modifies the configuration of a specific datapoint
/datapoint/<id>	DELETE	Deletes a specific datapoint

Figure 133. Symphony HAL REST API

### 3.2.1.6 Secure Data Store Solution

Secure Data Storage Solution is intended to be deployable on site. As such, there are two different Docker containers provided, one that contains only the SDSS API/UI suitable for expertly configured deployments, and an extended version for a simplified standalone deployment, which is packaged with the required dependant MongoDB and InfluxDB within the Docker container. The SDSS API is accessed via REST calls. The API guards calls by utilizing the OAuth provider in the Security Portal. Internally, configuration data is stored in a document-based database MongoDB, and the collected extracted data is stored in the timeseries database InfluxDB. The architecture of Secure Data Store Solution is presented in Figure 134 below.



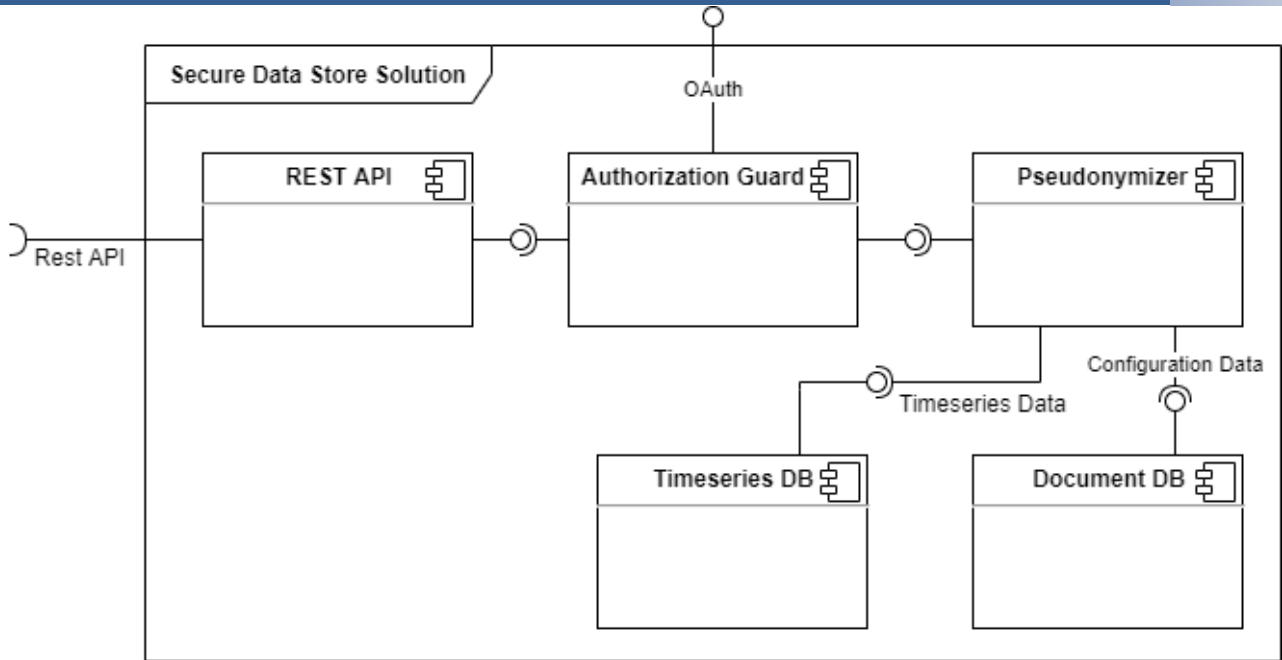


Figure 134. Secure Data Store Solution UML Diagram

For a simplified deployment, the SDSS UI is served by the API server. Users can configure how to connect to the desired channels in the message broker, and from the channel how to extract a set of datapoints as a timeseries. An example of such a mapping is shown below in Figure 135.

INOV StepNum (624d3386a11fc986b0e102a6)

Timestamp Specification: ISO\_8601 Preview: Thu Apr 21 2022 11:25:37 GMT+0200 (Central European Summer Time)

Timestamp Path: Readings.\*.TimeStamp Preview: "2022-04-21T07:25:37.1119592Z"

Timestamp Timezone: UTC +00:00 UTC

[Save](#)

### Tags

Tag Type	Use Literal Value for Field	Literal Value	JSON Path	JSON Selection Mode	InfluxDB Path	Unit of Measure	Unit Symbol	Unit Definition	Feature of Interest	Observed Property		
SENSOR	–	–	Name	Value	name	N/A	N/A	N/A	–	–	<a href="#">Edit</a>	<a href="#">Delete</a>
SENSOR	–	–	Readings.*.Value	Key (Position 1)	index	N/A	N/A	N/A	–	–	<a href="#">Edit</a>	<a href="#">Delete</a>

[New](#)

### Fields

Tag Type	Use Literal Value for Field	Literal Value	JSON Path	JSON Selection Mode	InfluxDB Path	Unit of Measure	Unit Symbol	Unit Definition	Feature of Interest	Observed Property		
–	–	–	Readings.*.Value	Value	stepNum	N/A	N/A	N/A	–	–	<a href="#">Edit</a>	<a href="#">Delete</a>
–	–	–	Min	Value	min	N/A	N/A	N/A	–	–	<a href="#">Edit</a>	<a href="#">Delete</a>

Figure 135. SDSS Data Mapping Overview

The UI enables users to specify how the SDSS should extract data from messages. Figure 136 shows a sample message that has been collected through the message broker, and how the SDSS allows for the mapping of fields within the message to be mapped into the timeseries database.

The screenshot shows the 'Field Editor' interface for a 'TAG' field. The configuration includes:

- Field Type:** TAG
- Tag Type:** SENSOR
- Timeseries ID:** INOV StepNum
- Use Literal Value for Field
- JSON Path:** Readings.\*.Value
- JSON Selection Mode:** KEY
- InfluxDB Path:** index
- Unit of Measure:** N/A
- Unit Symbol:** N/A
- Unit Definition:** (empty)

The **Preview** section shows a sample message structure:

```

{
  TagType: "IntTag",
  Name: "STEP_NUM",
  DataType: "Int",
  IOType: "InOut",
  Units: "",
  PublishedTimeStamp: "2022-04-21T07:25:37.1119592Z",
  Parameters: {
  },
  Readings: [
    {
      TimeStamp: "2022-04-21T07:25:37.1119592Z",
      Value: 0,
      Quality: 1
    }
  ],
  Min: 0,
  Max: 50,
  Avg: 25
}

```

Figure 136. SDSS Field Editor

### 3.2.1.7 Product Catalogue Service

Product Catalogue Service is a part of the NIMBLE platform and has an essential relationship and dependency to other NIMBLE services. Once catalogues of products and services are published by using the intuitive GUI, HTTP REST APIs for Catalogue Management and Product Category Management Services and the NIMBLE Indexing Service, the catalogues become available immediately to discover on the EFPF Integrated Marketplace. Since the Catalogue Management API is registered to the Data Spine Service Registry, users of the EFPF Integrated Marketplace can search for the published catalogues of products and services.

Product Catalogue Service uses YAML based default common configuration parameters defined on the NIMBLE platform itself such as the REST points of the other microservices that the Product Catalogue Service depends on, database connection parameters, etc. and all these default configurations can be overridden by setting appropriate environment variables or passing appropriate JAVA Virtual Machine arguments during the NIMBLE platform start up.

The architecture of the Product Catalogue Service is presented in Figure 137 below:

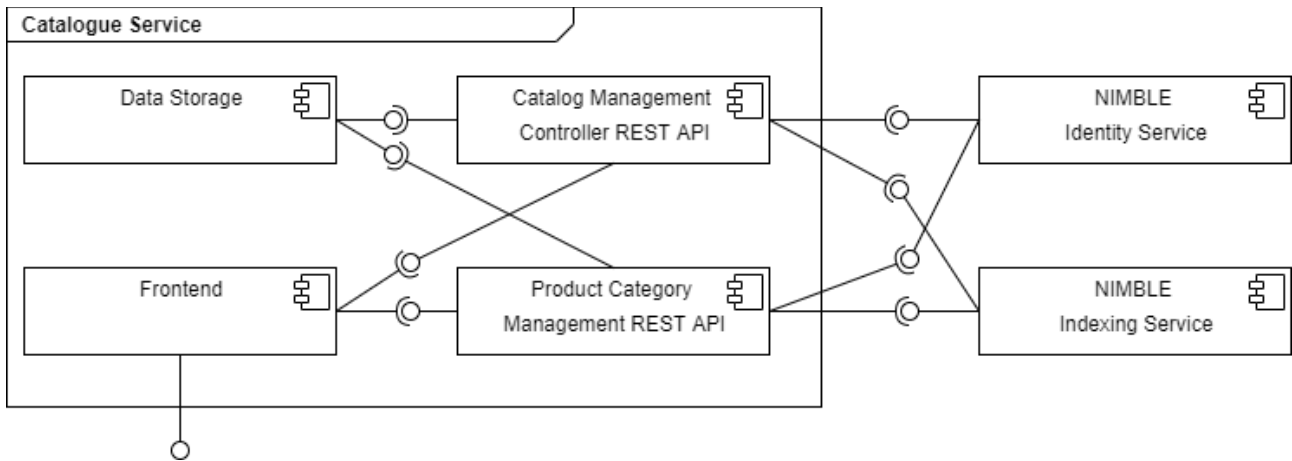


Figure 137. Product Catalogue Service UML Diagram

### Product Catalogue Service’s HTTP REST API for Product Category Management and Catalogue Management

Catalogue Service provides various REST endpoints to manage catalogues as well products and services. Basically, these endpoints are responsible for CRUD (Create, Read, Update and Delete) on both catalogues and products/services and the corresponding results are returned in JSON format which conforms to UBL 2.1 specification.

Currently, there are two main sets of REST services for *Product Category Management* and *Catalogue Management*. Details of Product Category Management endpoints can be seen in Figure 138:

REST Endpoint	HTTP Method	Description
<code>/taxonomies/{taxonomyid}/categories/tree</code>	GET	Returns the category tree for the given taxonomy and category
<code>/taxonomies/{taxonomyid}/root-categories</code>	GET	Returns the root categories for the given taxonomy
<code>/taxonomies/{taxonomyid}/categories</code>	GET	Returns a list of categories for a given keyword
<code>/taxonomies/{taxonomyid}/categories/children-categories</code>	GET	Returns the child category classes for the specified parent class

Figure 138. Product Category Management API Endpoints

Catalogue Management endpoints are described in Figure 139:

REST Endpoint	HTTP Method	Description
<code>/catalogue/{standard}/{uuid}</code>	GET	Returns the catalogue identified by the given identifier
<code>/catalogue/{standard}</code>	POST	Stores the given catalogue
<code>/catalogue/{standard}</code>	PUT	Updates a catalogue with the new data
<code>/catalogue/{standard}/{uuid}</code>	DELETE	Deletes the catalogue with the given identifier
<code>/catalogue/template</code>	GET	Generates a template for the product category specified by the given identifier
<code>/catalogue/template/upload</code>	POST	Uploads the template

Figure 139. Catalog Management API Endpoints

Products and services as well as their catalogues are persisted on a UBL-compliant relational database. UBL attributes of a product or service within a catalogue can be seen in Figure 140:

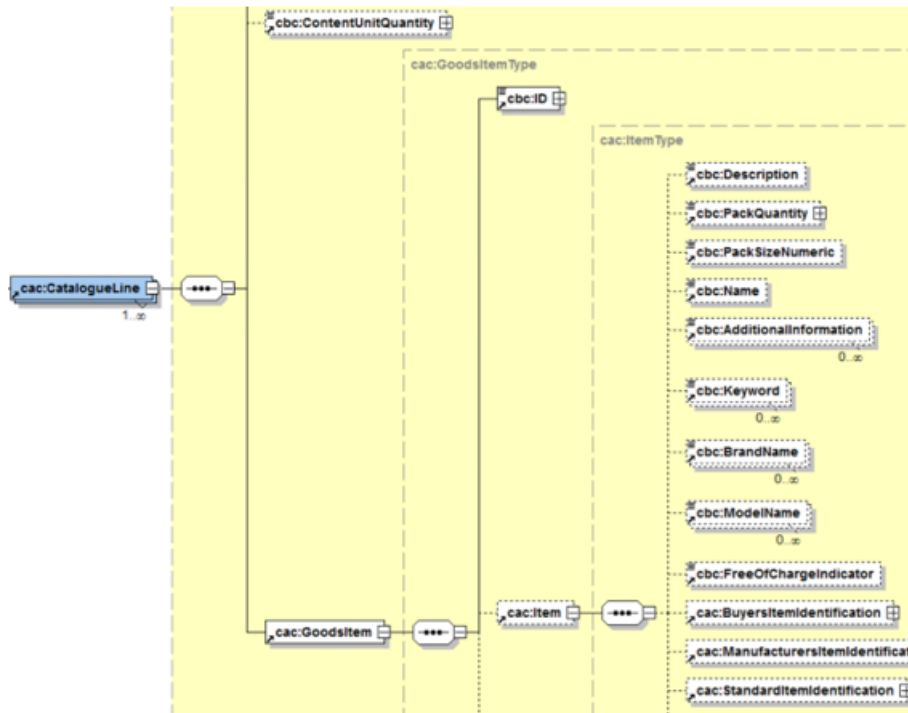


Figure 140. UBL Graphical representation for Catalogue Line schema

All the attributes of UBL Catalogue Line Schema are described below:

- UBLExtensions: A container for all extensions present in the document.
- UBLVersionID: Identifies the earliest version of the UBL 2 schema for this document type that defines all of the elements that might be encountered in the current instance.
- CustomizationID : Identifies a user-defined customization of UBL for a specific use.
- ProfileID: Identifies a user-defined profile of the customization of UBL being used.
- ProfileExecutionID : Identifies an instance of executing a profile, to associate all transactions in a collaboration.
- ID: An identifier for this document, assigned by the sender.
- UUID: A universally unique identifier for an instance of this document.
- ActionCode: A code signifying whether the transaction is a replacement or an update.
- Name: Text, assigned by the sender, that identifies this document to business users.
- IssueDate: The date, assigned by the sender, on which this document was issued.
- IssueTime: The time, assigned by the sender, at which this document was issued.
- RevisionDate: The date, assigned by the seller party, on which the information in the Catalogue was last revised.

- **RevisionTime:** The time, assigned by the Seller party, at which the information in the Catalogue was last revised.
- **Note:** Free-form text pertinent to this document, conveying information that is not contained explicitly in other structures.
- **Description:** Textual description of the document instance.
- **VersionID:** An identifier for the current version of the Catalogue.
- **PreviousVersionID:** An identifier for the previous version of the Catalogue that is superseded by this version.
- **LineCountNumeric:** The number of Catalogue Lines in the document.
- **ValidityPeriod:** A period, assigned by the seller, during which the information in the Catalogue is effective. This may be given as start and end dates or as a duration.
- **ReferencedContract:** A contract or framework agreement with which this Catalogue is associated.
- **SourceCatalogueReference :** A reference to the source catalogue.
- **DocumentReference:** A reference to another document associated with this document.
- **Signature:** A signature applied to this document.
- **ProviderParty:** The party providing the Catalogue.
- **ReceiverParty:** The party receiving the Catalogue.
- **SellerSupplierParty:** The seller.
- **ContractorCustomerParty:** The customer party responsible for the contracts with which the Catalogue is associated.
- **TradingTerms:** The trading terms associated with this Catalogue.
- **CatalogueLine:** A line in a Catalogue describing an item of sale.

### **Product Catalogue Service's GUI for Product Publishing:**

As mentioned previously, Product Catalogue Service provides an intuitive GUI for product publishing and example screenshots can be seen in Figure 141, Figure 142 and Figure 143.

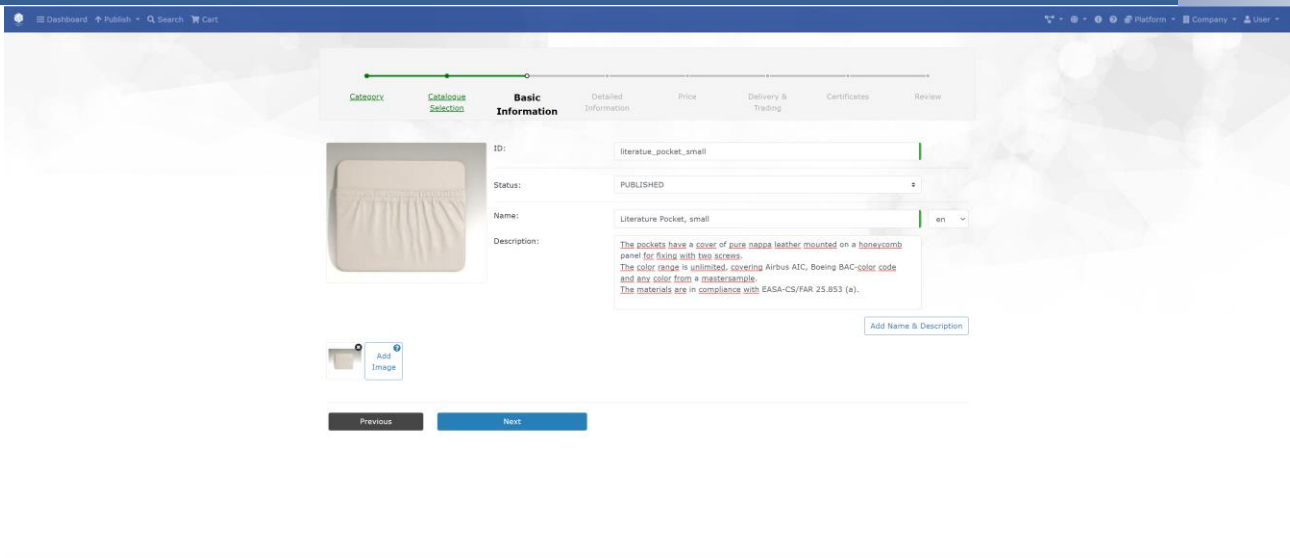


Figure 141. Filling out basic product information

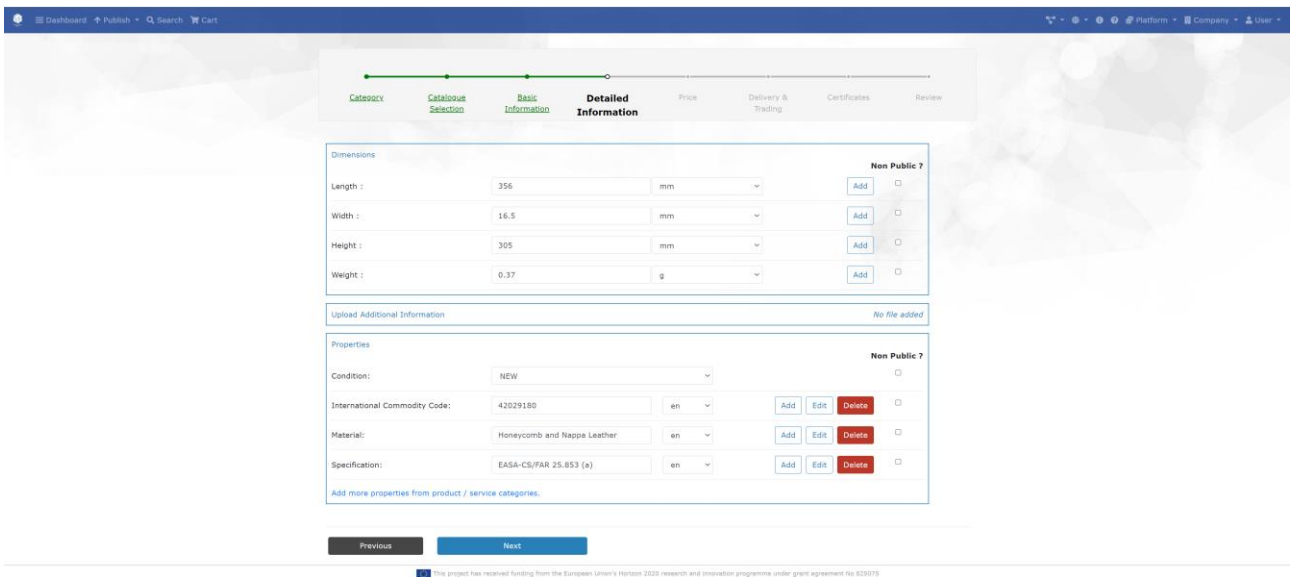


Figure 142. Filling out detailed product information

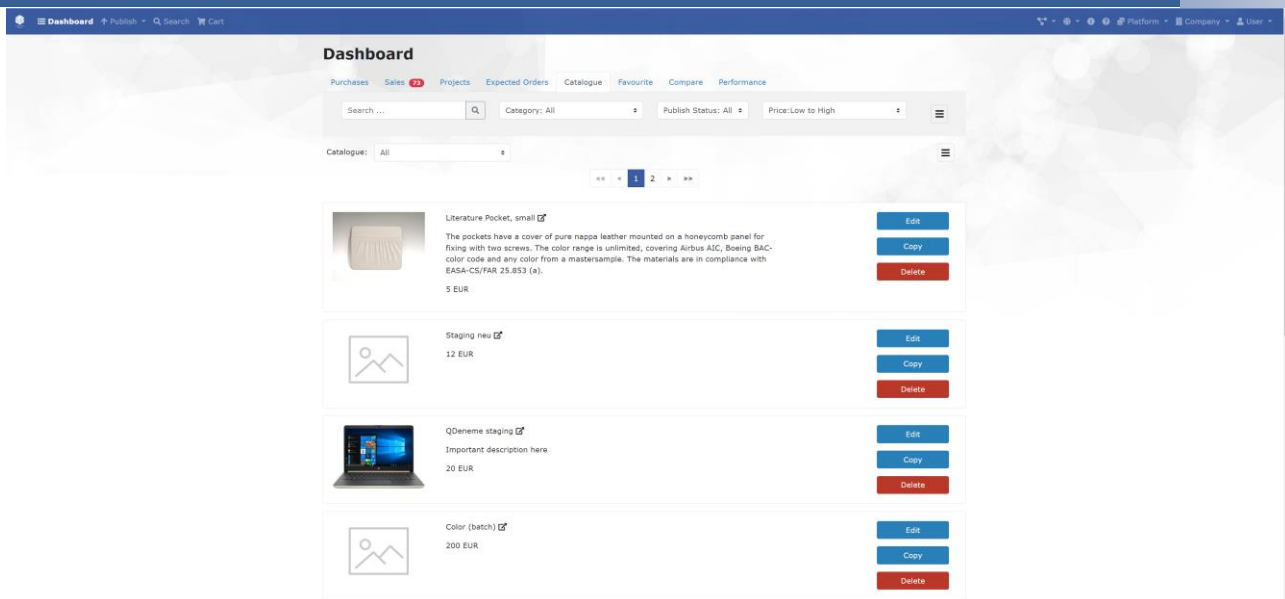


Figure 143. Product placed within a catalogue

### 3.2.1.8 Matchmaking Service

The Matchmaking Service in this context is the combination of the Data Collection and Data Transformation steps with the Data Spine, the storage of the aligned data in the Federated Search Service and finally the Team Formation Procedures provided with the participating platforms. Thus, the Matchmaking Service can be divided into the following tasks:

1. Data gathering / federation of the relevant information, data alignment for use with Federated Search Service.
2. Provision of stored information with a unified search interface
3. Agile Network Creation with selected participants

#### 3.2.1.8.1 Data Gathering & Alignment

The participating platforms need to propagate any changes of the company or product item data to the Federated Search Service. Within the EFPE ecosystem, the participants based on the NIMBLE platform may configure this setting and the company and item related data is propagated automatically. This feature cannot be enabled in each of the participating platforms; therefore, a different way of data collection and alignment is provided with the functionality of the Data Spine.

First, each of the participating platform needs to expose a web service providing the federated data. To ease the integration, the exposed webservices need not adhere to the unified data format used with the federated search service. Instead, a configurable Data Alignment is performed when processing the results of the distinct platform endpoints with JOLT Transformations. Finally, the transformed data is pushed to the Federated Search Service. Figure 144 outlines this process.

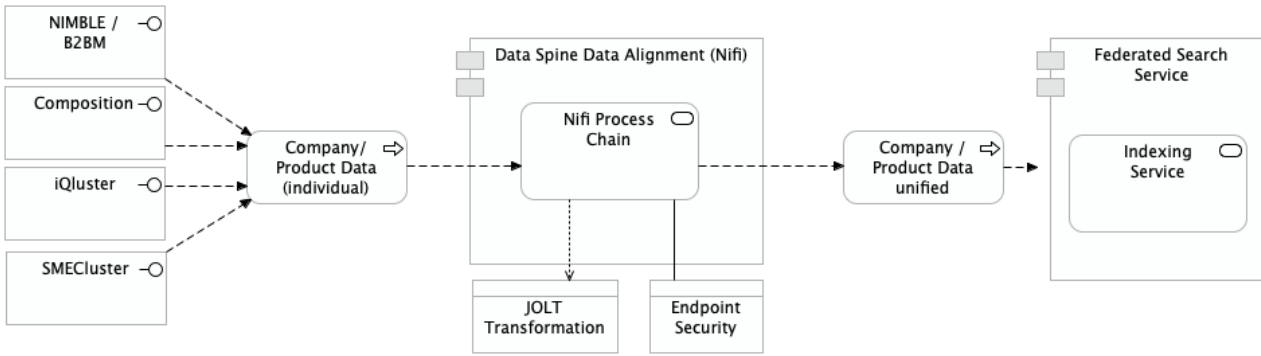


Figure 144. Data Collection and Alignment with Data Spine

### 3.2.1.8.2 Federated Search Service

The Matchmaking Service provides the federated data for search and retrieval in the EFPF ecosystem. It uses a high-performance full text index for storing arbitrary documents such as product items or company information. However, product items belong to a category or a product group. Furthermore, product items may contain custom values which are only present in a distinct product group. To support this requirement, the Matchmaking service stores all the data according to the EFPF Manufacturing Ontology as depicted in Figure 145.

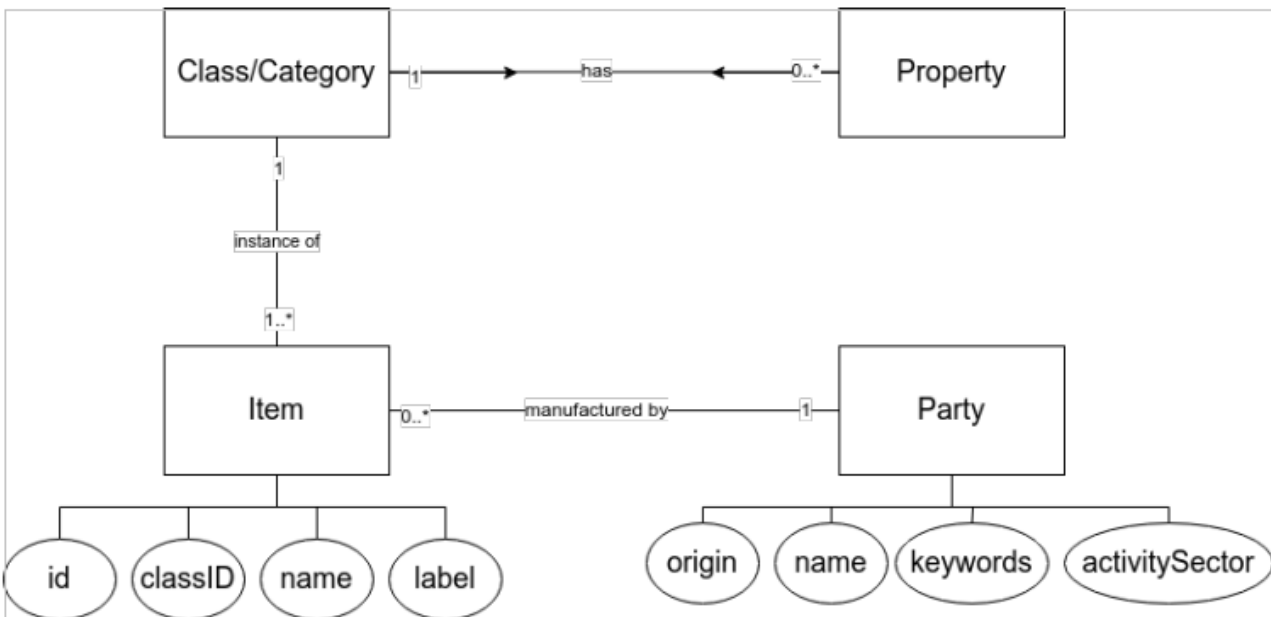


Figure 145. EFPF Manufacturing Ontology (simplified view)

This data structure is reflected in the Federated Search Service, the storage component of the Matchmaking Service. Each of the data categories (Class/Category, Property, Item, Party/Company) is stored in a data collection which is accessible with a dedicated web service interface which allows adding, updating, deleting and most important retrieval of each data category. Each of the data category exposes the same set of API methods. The currently available data categories are as follows:

- *class*: arbitrary Categories, Classifications
- *property*: Naming, Datatype of Properties
- *item*: Product Catalogue



- *party*: Company/Manufacturer Data (including Contact Persons)
- *opportunity*: Business Opportunity

Figure 146 outlines the API methods for each data category. The term {category} must be replaced with either *class*, *property*, *item*, *party*, or *opportunity*.

REST Endpoint	HTTP Method	Description
/ {category} ?uri={uri}	GET	Retrieve the stored document by its unique URI
/ {category}	POST	Create or update a “document” of the given data category
/ {category} ?uri={uri}	DELETE	Delete a single document by its unique URI
/ {category} /select?q={q}	GET	Search for documents in the given data category. Multiple search parameters such as paging field queries or faceting options are possible.
/ {category} /reset ?basePlatform={basePlatform}	POST	Replace the dataset assigned to a given Base-Platform. Provided documents are created/updated. Existing, but in the request missing documents are removed.
/ {category} /fields	GET	Documents may add custom data fields to the stored index. The full list of fields in use is therefore not predefined and may be retrieved with this request. Fields are (optionally) linked with corresponding properties providing human readable, multilingual naming for the search interfaces.

Figure 146. Federated Search Service API

### 3.2.1.8.3 Agile Network Creation

The data category named *party* is used to store manufacturer related data. During data collections and alignment, the service endpoints of the distinct platform provider are requested to provide the main contact information as well. This contact information is stored along with the company and also part of the search results. Based on this, the search interface of the Matchmaking Service allows for the selection or multiple companies and to invite them for collaboration, as illustrated in Figure 147.

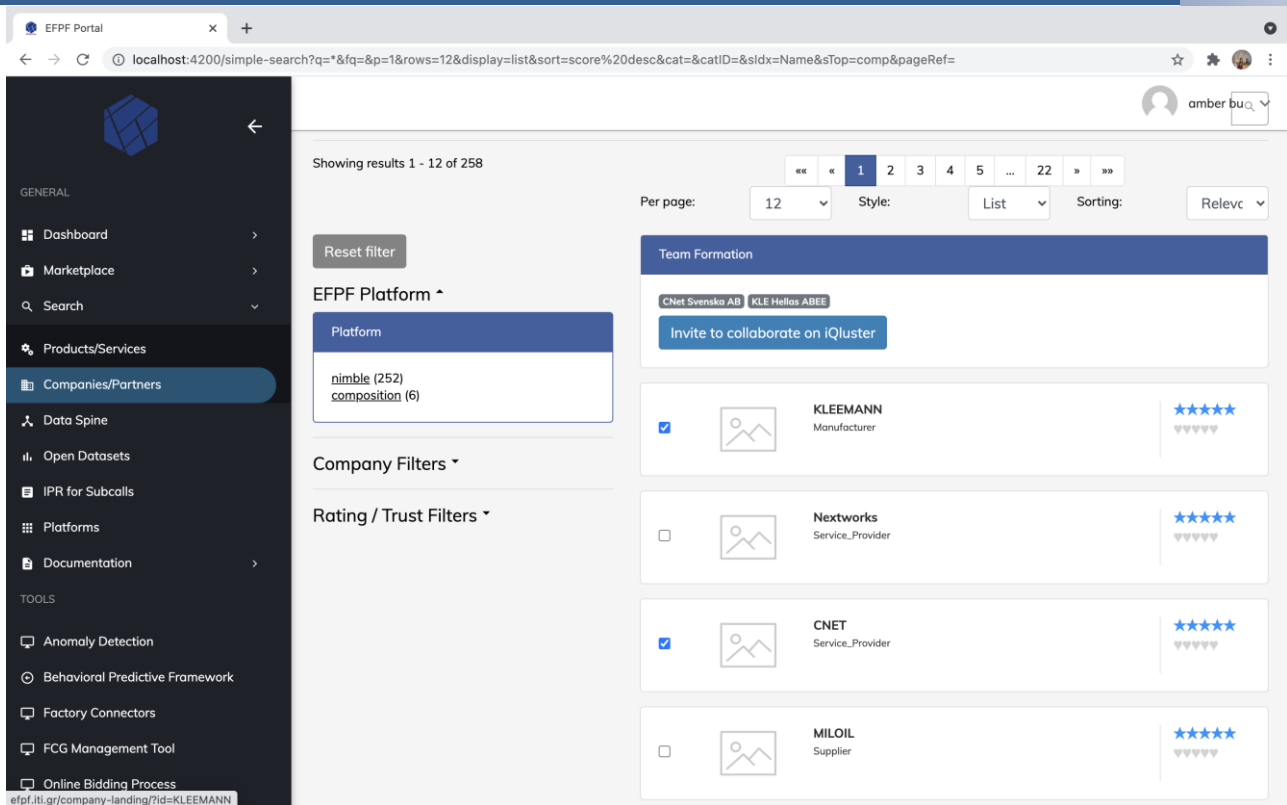


Figure 147. Invite for collaboration on VLC’s Network Portal Platform

Note: The Matchmaking Service only triggers the Team Formation process with the connected platforms. No Team/Network related data is stored with the Matchmaking Service.

### 3.2.1.9 Online Bidding Process

The Online Bidding Process platform provides an automated matchmaking mechanism for information requests from buyers to suppliers, to execute negotiations and business transactions automatically or manually via configured agents. Figure 148 shows the latest functional architecture of the components that enable such mechanism.

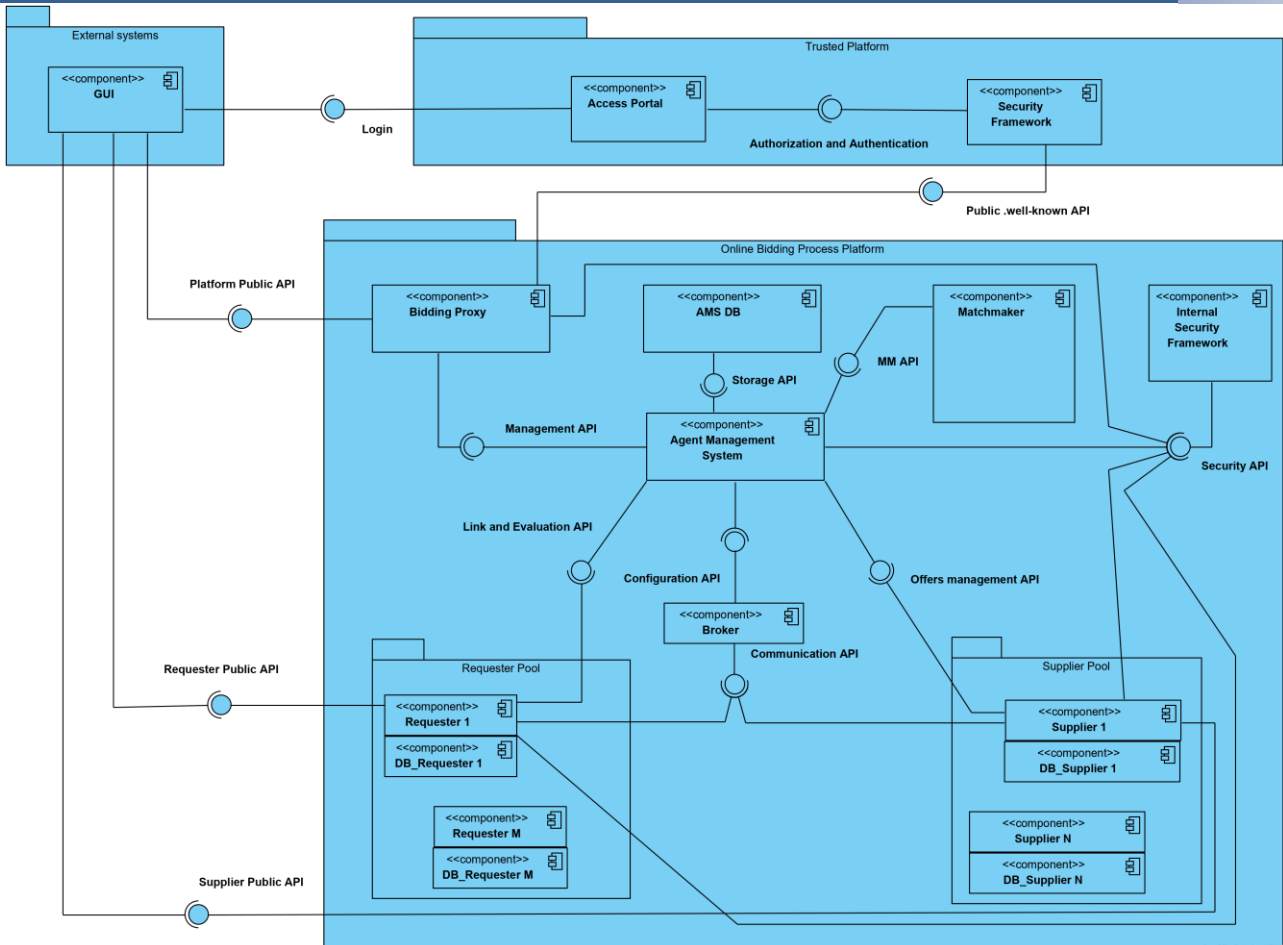


Figure 148. Online Bidding Process Architecture

The building blocks are described below:

- **Semantic Framework / Matchmaker:** the component is a complete semantic framework able to represent business entities, products etc. based on an ontology. Moreover, it supports the storing of business entities etc. as ontology instances/individual. In general, provides CRUD operations and a triple store for saving the data. The core functionality of the component is the matchmaking on 2 levels: (1) supplier/requester matching based on semantic rules/inference and (2) offers/request matching based on best score kind of algorithm. The Matchmaker has been developed in Java using Apache Jena Framework and it was deployed in a Tomcat server. Docker was used in order to create an image for the Matchmaker framework. The Matchmaker is finally deployed in a server as a Docker container. The Matchmaker services are provided as RESTful web services to both Agents and GUI. The ontology used by the Matchmaker has been developed in OWL2 using Protégé tool.
- **Graphical User Interface (GUI):** a web-based user, linked to the EFPF portal, able to interact with the APIs of the Online Bidding Process entities such as Agents APIs and Matchmaker APIs. The interface has been developed in Angular 9 and enable end-users to create an agent, to initialize a bidding process for a good, to participate in a bidding process as bidders, to manage the biddings history and to be notified for available requests etc.
- **Virtual Agents (requesters and suppliers):** these components provide all the necessary API to represent companies/business entities in the bidding processes and to trigger the

communication functionalities between them. The Agents communicate with both GUI and Matchmaker using REST interfaces. The communication channel used between the agents rely on a AMQP broker deployed within the platform. The CXL data format, an extension of the FIPA standard, is used to encapsulate the overall messages exchanged during the bidding process. The protocol used to put in place the biddings is the ContractNet. These components were developed in Python 3.9 with FastAPI, Uvicorn and Gunicorn. All the replicated workers managing REST API are dynamically configured to be reachable via nginx behind ad-hoc path, coherent with the unique ID of the agents. All the agents are dockerized and offer OAuth2-secured interfaces transparently configured during each deploy with the EFS credentials.

- **Agents Management System (AMS):** is an infrastructure component able to store and manage agent information entities, necessary to provide the interconnectivity features among the overall components of the platform. The AMS has been developed in Python using FastAPI and it was deployed in a Gunicorn server. These components were developed in Python 3.9 with FastAPI, Uvicorn and Gunicorn. The REST API offered by the AMS exploit Oauth2 credentials dynamically obtained by the virtual agents and by the bidding proxy through a Keycloak instance used within the platform itself for M2M communications.
- **Broker:** rely on RabbitMQ and dynamically configured by the AMS and by the agents to put in place direct and fanout AMQP communication among requesters and suppliers. The exploitation of the virtual host, exchanges and queues is limited to the agents of the platform.
- **Bidding Proxy:** is an infrastructure component used as main interface for the overall platform management features. It enables users authorized by the EFPF platform to create, locate, and manage virtual agents. Its API are registered within the SR and protected with the EFS features to enable the GUI to securely reach the Online Bidding Process public API. These components were developed in Python 3.9 with FastAPI, Uvicorn and Gunicorn. It is authorized to interact with the AMS to proceed with the characterization of the agents and the registration process during the deploy and removal phase. It produces containerized agents and inject their configuration within the deploy process and configures dynamically the nginx rules of the host machine to provide ways to reach the dockerized solutions.
- **Databases:** rely on Postgres and deployed within docker containers for each agent of the infrastructure to provide them all the necessary information to manage the persistent configuration applied, the states of the bidding processes, and the history of the bids. Each database is configured to communicate only with the related agent.
- **Internal Security Framework:** relies on a dedicated instance of Keycloak, deployed within the platform in a docker container. It is used to authorize the M2M communications generated among the agents of the infrastructure.

Figure 149 shows the latest functional architecture of the sub-components that enable the Matchmaker features.

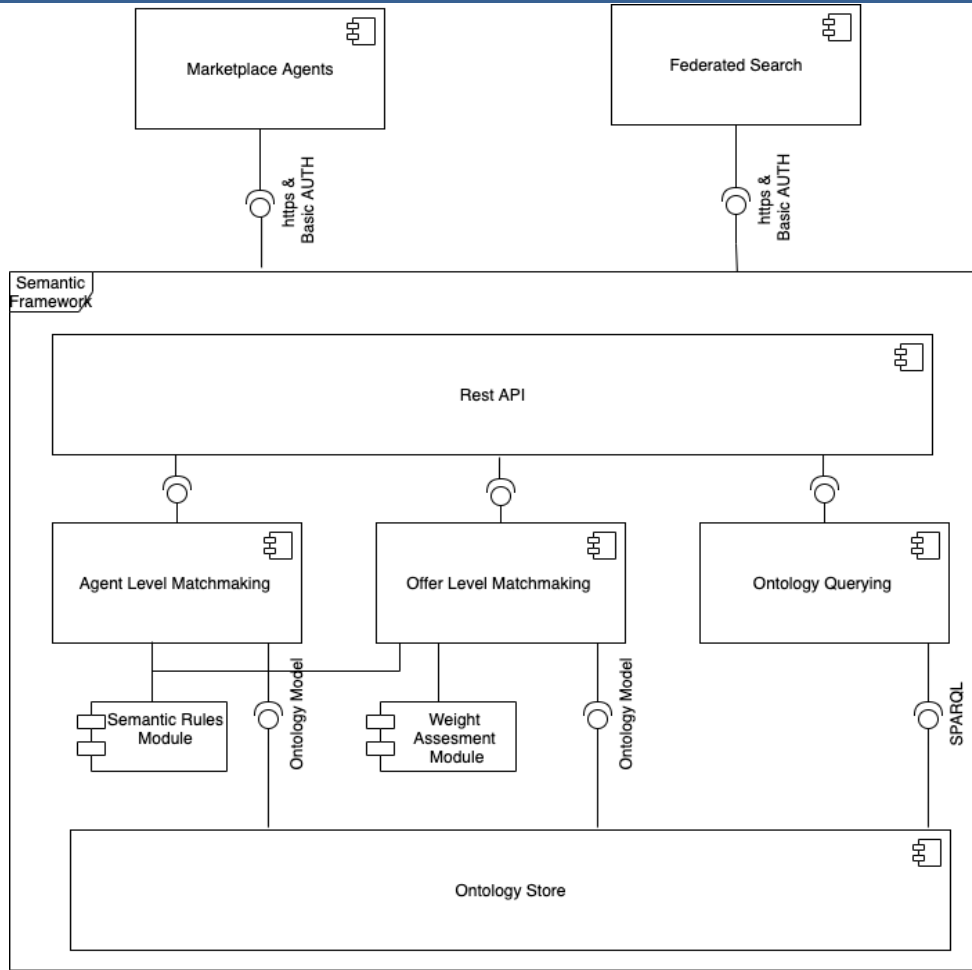


Figure 149. Matchmaker Architecture

All the agents of the Online Bidding Process Platform hosts ReDoc pages that describe the API available through the OpenAPI 3.0 specifications.

The Online Bidding Process is offered through a complete web-based to end user. The UI enables a user to create an agent and add company details, to initialize a bidding process and manage bidding processes overall, to bid for request or to evaluate and accept/reject offers. Some examples of GUIs are available below. The set of REST services of Semantic Framework’s API that are called by the agents are available in the section related to COMPOSITION interfaces.

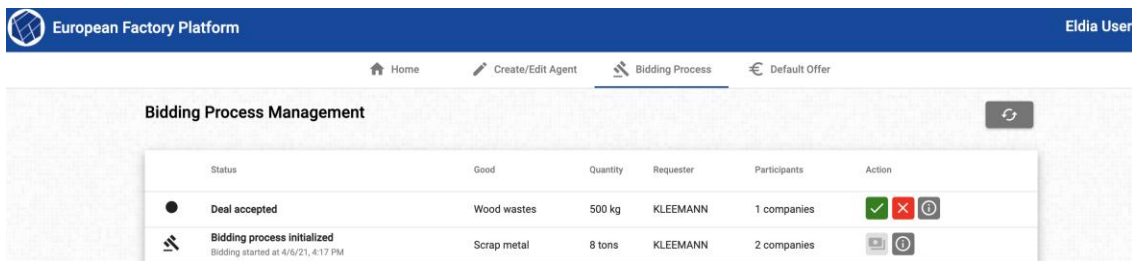


Figure 150. Management of Bidding Processes

The GUI provides the set of active and terminated biddings enabling ways to obtain real-time details from it such as the current state, the good and quantity requested, the requester/suppliers involved and their offers. Through the exploitation of a default offer configuration injected to the virtual agent supplier, based on the user preferences, the

interactions of the bid can be fully automatic. At the end of the bid, before the conclusion of the deal, the virtual requester agent waits for a final confirmation from the user of the best offer received. Figure 150 shows two bids, one bid process just initialized and another one waiting for the final confirmation.

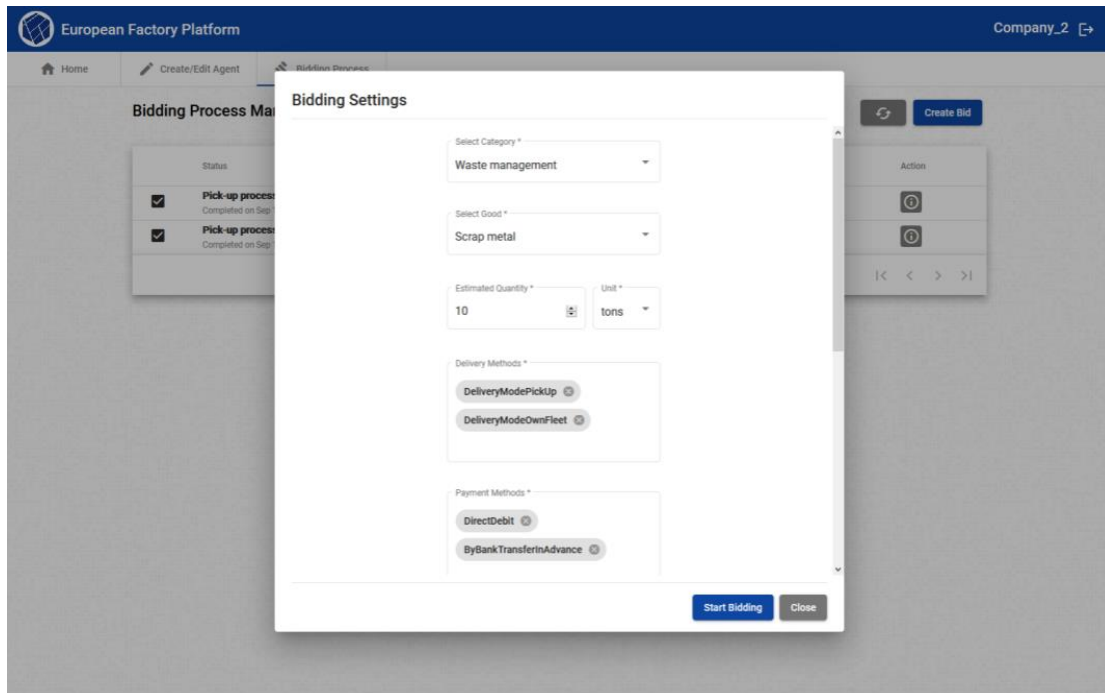


Figure 151. Initiate a new Online Bidding Process

An authorized user, through the requester agent API, can trigger a new bidding process by setting up a request for a service/good to the marketplace. It is possible to inject a set of parameters that characterize the request, such as to specify the time and date for the bid deadlines. Figure 151 shows a portion of the parameters that a user can define for the bidding request.

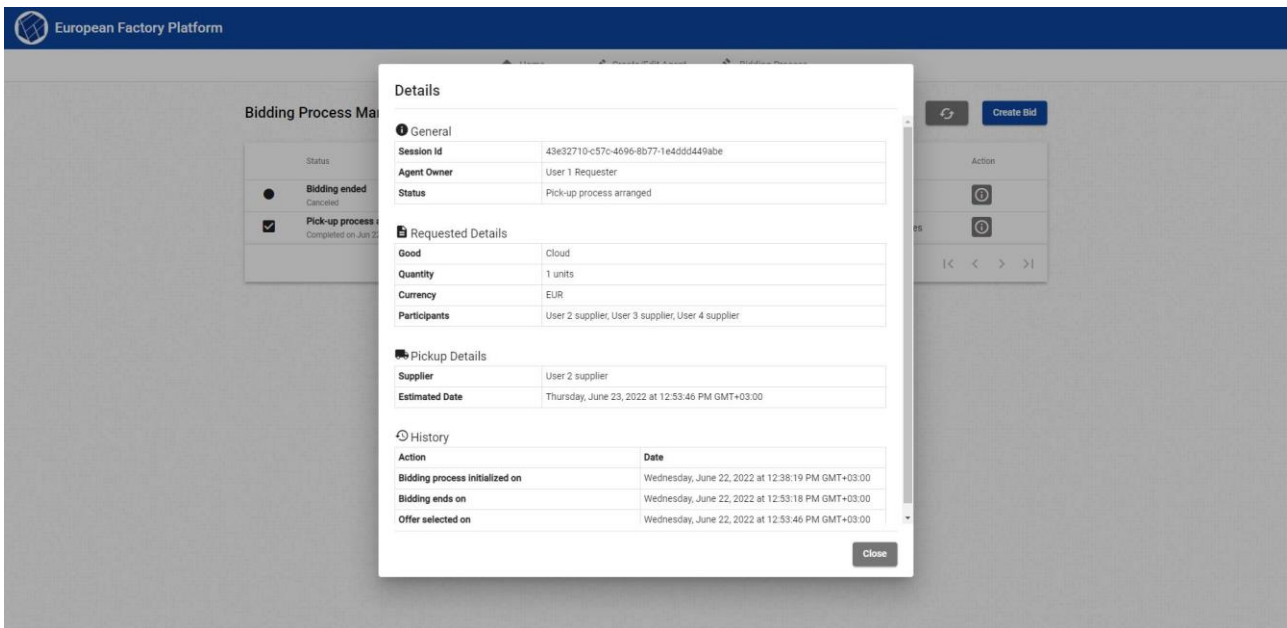


Figure 152. Details of the deal

Both the requester and supplier users, through the GUI and their virtual agents, can obtain the details of the deal generated by the cooperation of the platform components as shown in Figure 152.

### 3.2.1.10 Business Opportunity Tool

The Business Opportunity Tool allows Companies to post Business Opportunities so that they can be searched and allow application bids to be made by Supplier companies with the required capabilities and accreditations. Business Opportunities and Companies from the Tool are also aggregated to the EFPF platform through the Federated Search Component to allow EFPF members to find them based on keywords. The Tool is hosted on the SMECluster Platform that is linked to the Data Spine for authentication and the marketplace.

The Business Opportunity Tool, as illustrated in Figure 153, comprises a service-based architecture that includes Company Directory Service and Opportunity Service. The UI provided by the Tool offers several functions including Search Tools, Opportunity Creation and Application management, and a Team Formation and Collaboration.

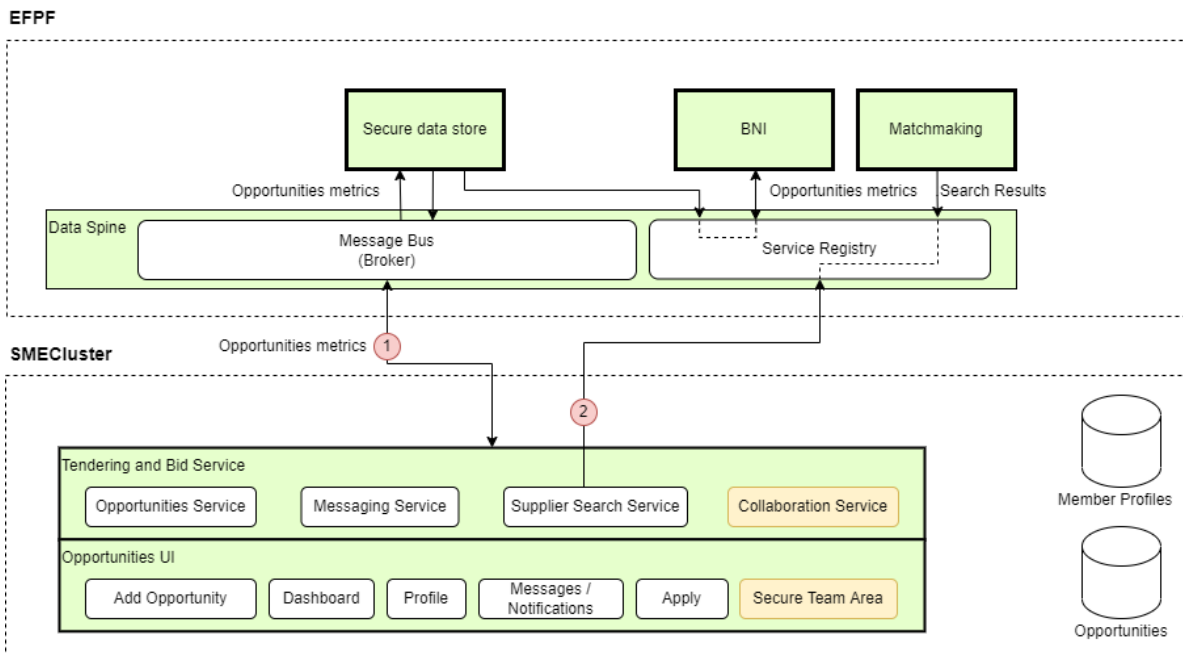


Figure 153. Business Opportunity Component View

The Business Opportunity Tool provides API access to the service framework to the Federated Search component within EFPF. This allows companies and business opportunities to be aggregated and searched by users of the EFPF platform, as shown in Figure 154.

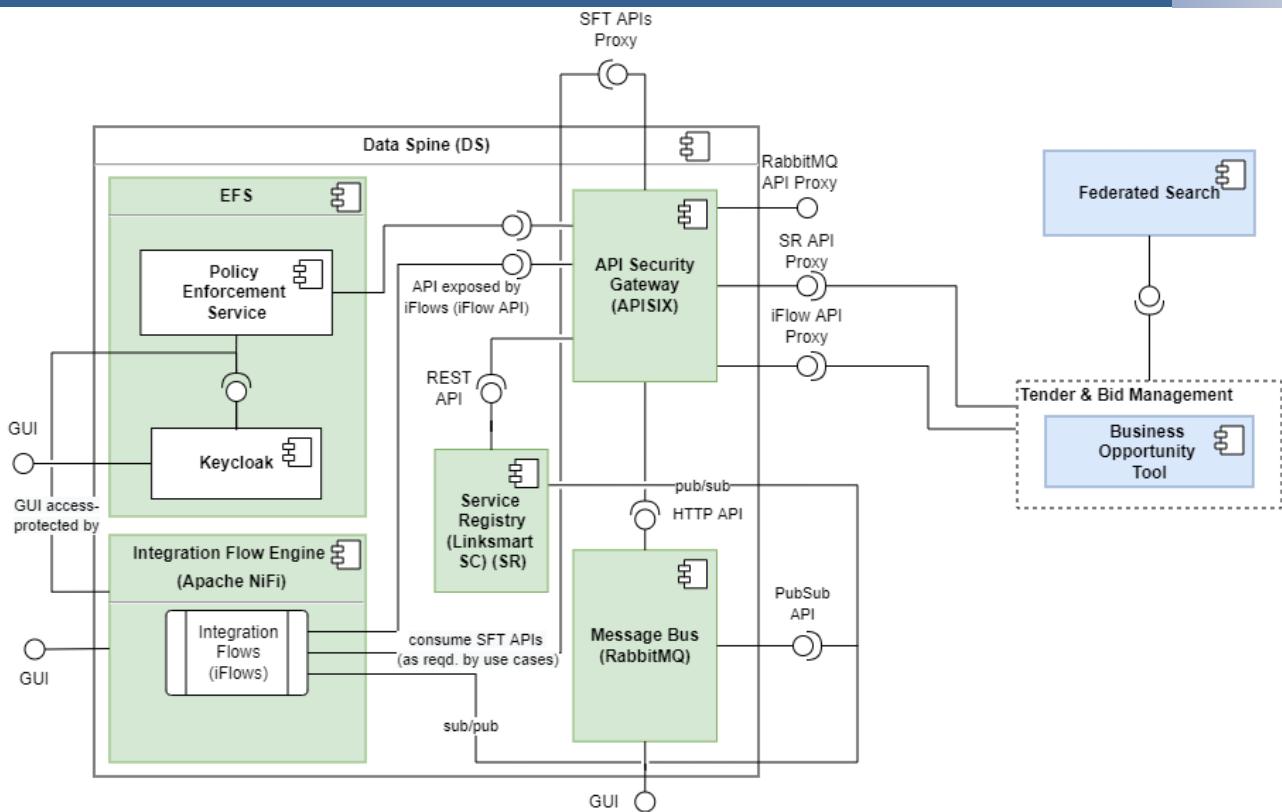


Figure 154. Business Opportunity Tool

### 3.2.1.11 Business & Network Intelligence

The insights generated from the analysis of data collected by the Accountancy Service are presented as visualisations within the Business & Network Intelligence Service. The generated visualisations are grouped into relevant dashboards to provide 3 distinct dashboards focusing on different aspects of the platform's usage.

#### Platform Trends

The platform trends dashboard has been included below in Figure 155. This provides EFPF users with insights into trends within the platform. This includes for example, usage trends of the connected platforms, tools, and services, but also search trends in the platform, both at a product and company level.



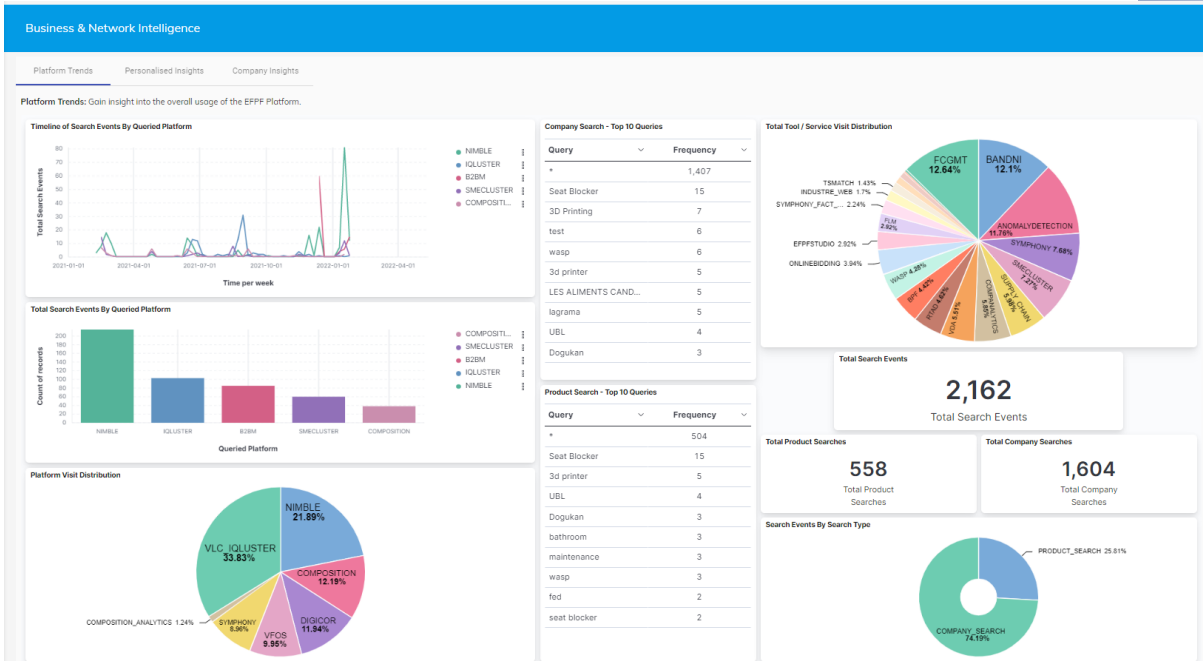


Figure 155. Business & Network Intelligence: Platform Trends Dashboard

### Personalised Insights

The personalised insights dashboard, presented in Figure 156, provides a series of visualisations that focus on providing insights into an individual user's use of the EFPF platform and is generated dynamically based on the Access Token present in the user's session. In this dashboard, users are presented with insights into their platform, tool, and service visits within the portal, their personalised search trends, but also a summary of both their purchases and expenditure within the platform.

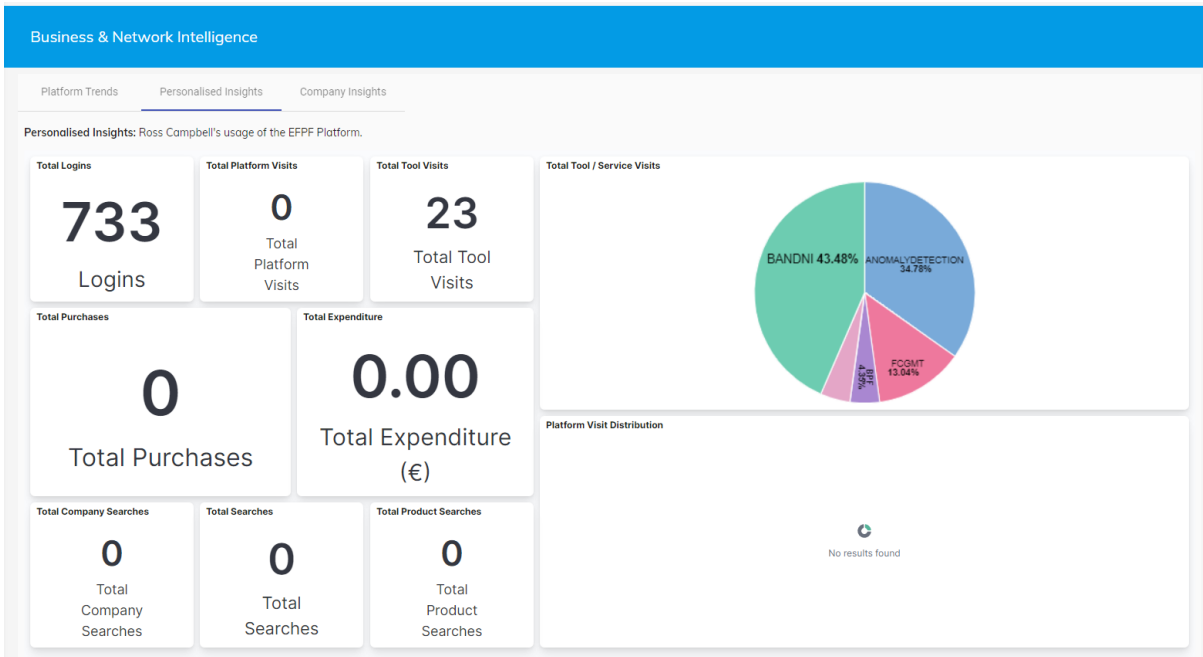


Figure 156. Business & Network Intelligence: Personalised Insights Dashboard

### Company Insights

The final dashboard provided by the Business & Network Intelligence Service is the Company Insights dashboard, illustrated in Figure 157. This provides a series of visualisations that are focused on insights into how a user's associated company has interacted with the EFPF. This is the same dashboard presented, however this time, the users token is queried against the EFS component to obtain a list of users present in the company, and the dashboard dynamically generated based on the list of user Id's returned from the EFS.

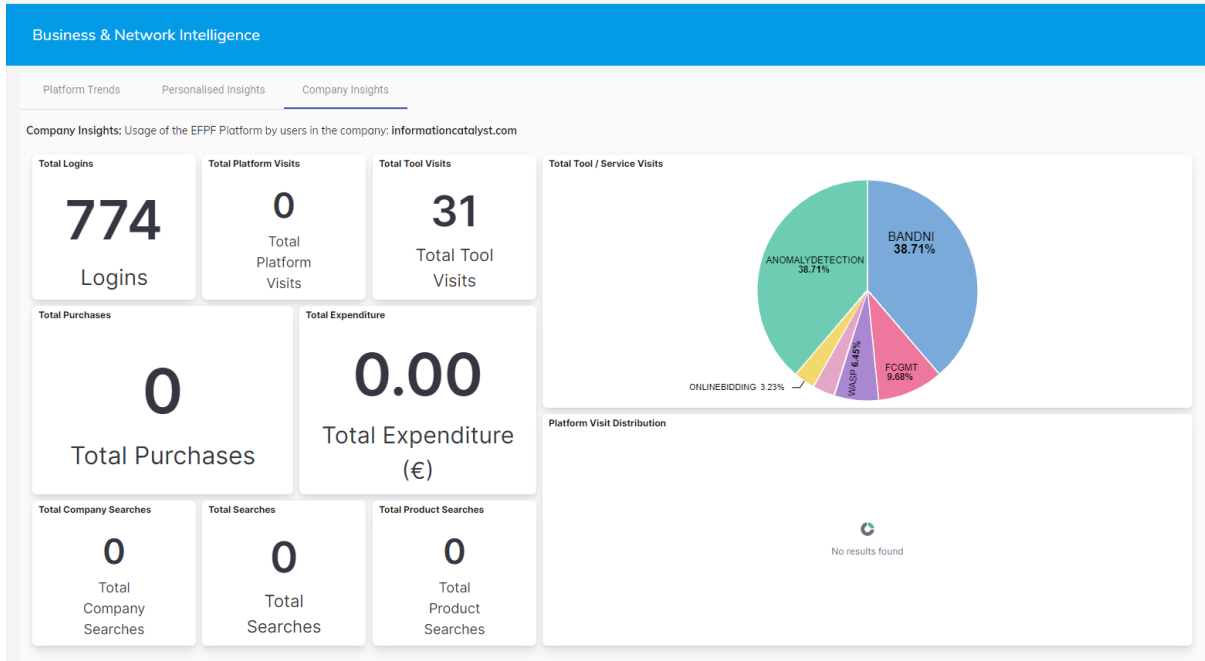


Figure 157. Business & Network Intelligence: Company Insights Dashboard

### Data Model

The data model used by the Business & Network Intelligence Service is tightly coupled with that defined by the Accountancy Service. This includes defined data models for; Login, Company Registration, User Registration, Platform Visit, Tool Visit, Search & Payment events in the EFPF Platform. However, the full index mapping implemented within the B&NI Elasticsearch component has also been included below in Figure 158 and Figure 159.

```

{
  "efpf_platform" : {
    "mappings" : {
      "properties" : {
        "@timestamp" : {
          "type" : "date"
        },
        "@version" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
              "ignore_above" : 256
            }
          }
        },
        "action" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
        "authDetails" : {
          "properties" : {
            "clientId" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "ipAddress" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "realmId" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "userId" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}}
          }
        },
        "buyerId" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
        "clientId" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
        "details" : {
          "properties" : {
            "client_auth_method" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "grant_type" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "refresh_token_id" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "refresh_token_type" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "scope" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "token_id" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "updated_refresh_token_id" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}}
          }
        },
        "facetQuery" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
        "headers" : {
          "properties" : {
            "accept_encoding" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "accept_language" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "cache_control" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "connection" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "content_length" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "content_type" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "contenttype" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "cookie" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "from" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "http_accept" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "http_host" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "http_user_agent" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "http_version" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "kbn_xsrft" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "postman_token" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "pragma" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "request_id" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "request_method" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "request_path" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "sec_ch_ua" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "sec_ch_ua_mobile" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "sec_ch_ua_platform" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "sec_fetch_dest" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "sec_fetch_mode" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "sec_fetch_site" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "sec_fetch_user" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "upgrade_insecure_requests" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "x_forwarded_for" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "x_forwarded_host" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "x_forwarded_port" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "x_forwarded_proto" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "x_real_ip" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
            "x_userinfo" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}}
          }
        }
      }
    }
  }
}

```

Figure 158. Business &amp; Network Intelligence Data Model: Part 1

```

    },
    "host" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "ipAddress" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "loginStatus" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "message" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "openCallParticipant" : {"type":"boolean"},
    "operationType" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "originPlatform" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "platform" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "products" : {
      "properties" : {
        "catalogId" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
        "productCount" : {"type":"long"},
        "productId" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
        "productName" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
        "totalPrice" : {"type":"long"},
        "unitPrice" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}}
      }
    },
    "queriedPlatforms" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "query" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "realmId" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "representation" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "resourcePath" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "resourceType" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "searchResponse" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "searchType" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "sellerId" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "sessionId" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "status" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "tags" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "time" : {"type":"long"},
    "timestamp" : {"type":"date"},
    "totalAmount" : {"type":"long"},
    "transactionId" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "type" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "userId" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "visitedPlatform" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}},
    "visitedTool" : {"type":"text","fields":{"keyword":{"type":"keyword","ignore_above":256}}}
  }
}
}
}

```

Figure 159. Business &amp; Network Intelligence Data Model: Part 2

### Business & Network Intelligence: Component Navigation

To support integration of the component in the EFPF Portal, a component module was generated in the EFPF Portal project to enable the display and navigation between the different dashboards. The service can then be accessed by opening the side navigation in the Portal and selecting the “Platform Trends” component as shown below in Figure 160. Alongside this, the three dashboards previously mentioned can be accessed through relevant tabs located at the top of the tools interface.

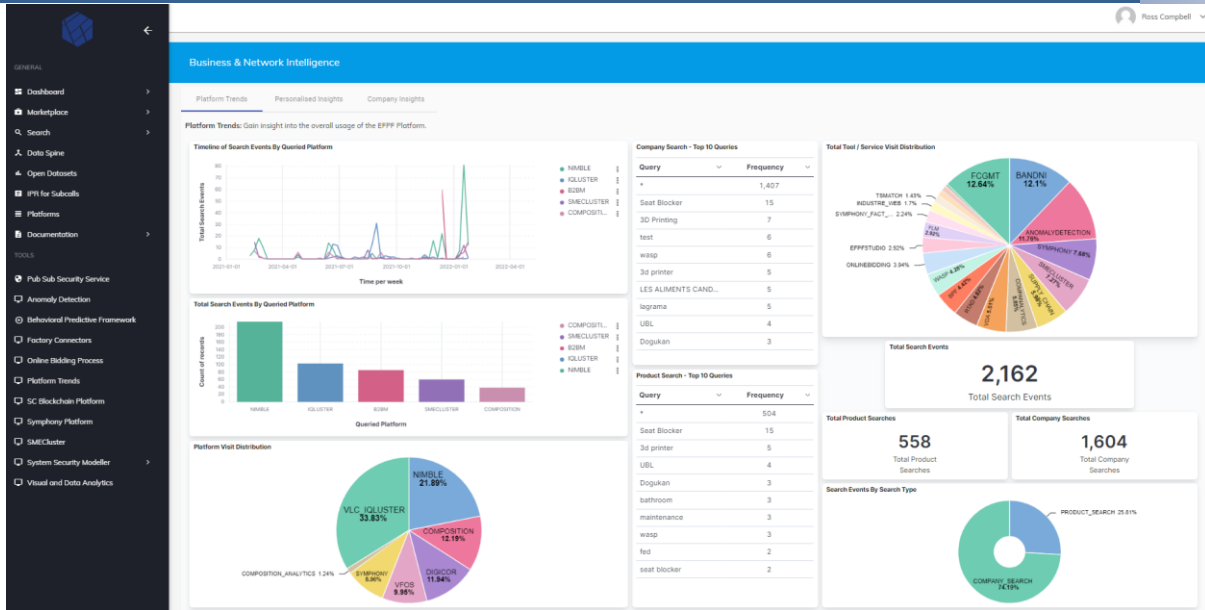


Figure 160. Business & Network Intelligence: Portal Navigation

### Business & Network Intelligence: Architecture

The Business & Network Intelligence Service is based on the Elasticsearch Stack (Elasticsearch and Kibana) with an additional middleware application to manage data importation tasks and handle any external queries or calculations that need to be made to generate insightful dashboards. The architecture of the Business & Network Intelligence Service has been illustrated and included in Figure 161.

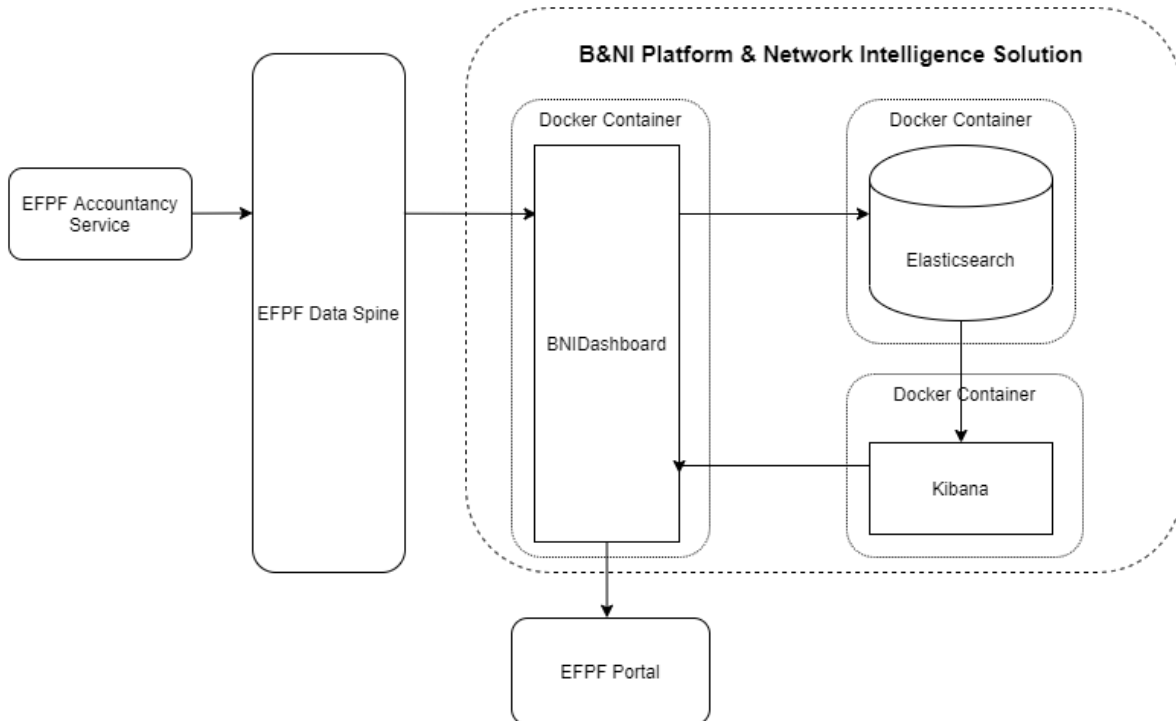


Figure 161. Business & Network Intelligence: Architecture Diagram

The B&NIDashboard application is central to the overall service and is primarily used to query and retrieve platform events from the platform’s Accountancy Service. Each event in the

EFPF platform is then indexed into to the B&NI Elasticsearch component. The Kibana component of the B&NI service then provides visualisation capabilities of the data, which is then aggregated into distinct dashboards and served to the EFPF Portal via the BNIDashboard component.

### Business & Network Intelligence: Data Spine Usage

While not necessarily needed, the B&NI Service has also taken advantage of the EFPF Data Spine to enable data importation workflows through the DS’s Integration Flow Engine. The setup of a REST endpoint in an IFE Flow has enabled the triggering of a full workflow via HTTP call, that will perform the necessary queries to the accountancy service and send each new event to the Business and Network Intelligence Service. The IFE Flow enabling this operation has been included below in Figure 162.

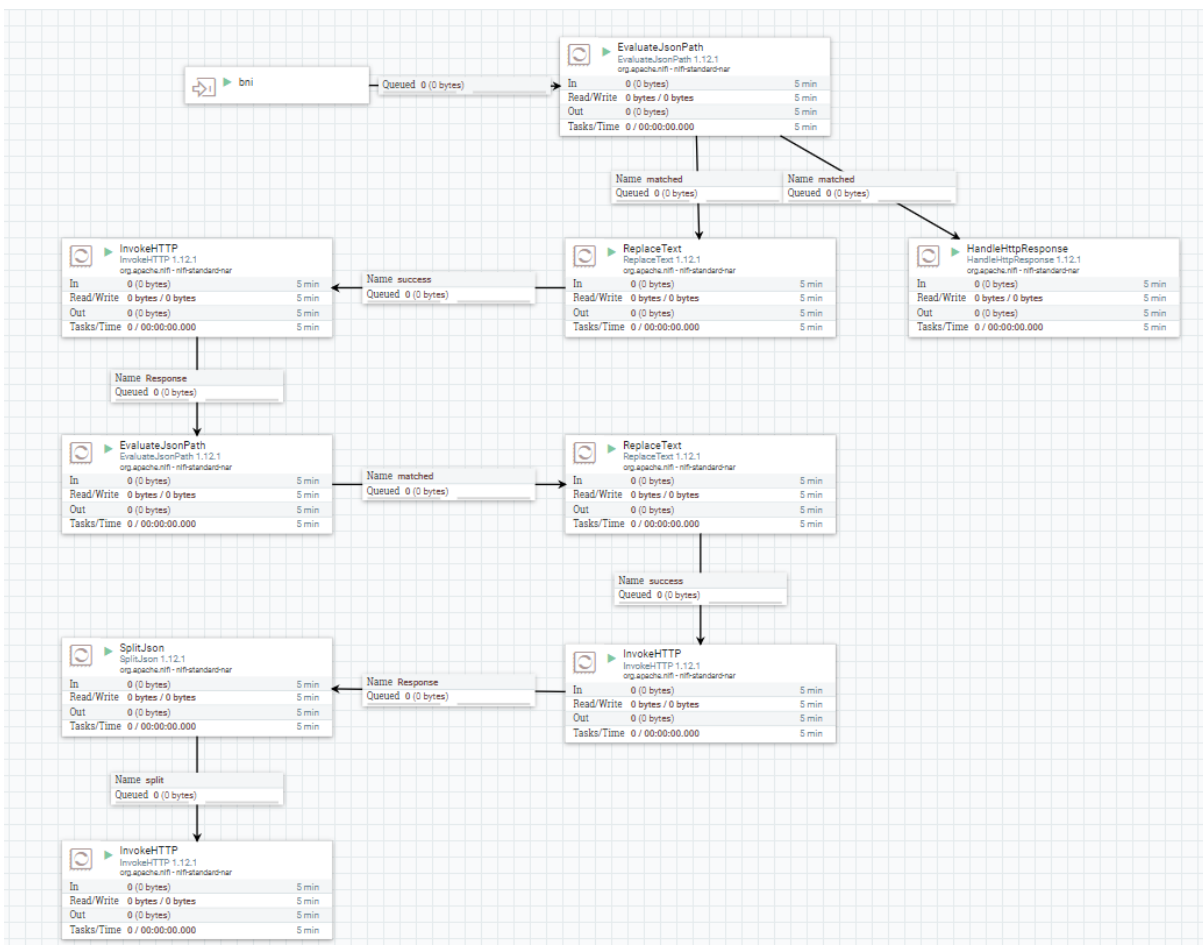


Figure 162. Business & Network Intelligence: NiFi Flow

### 3.2.1.12 Blockchain & Smart Contracting

The Blockchain and Smart Contracting services provide three high-level APIs to distributed transaction ledgers (DTL) for the EFPF Ecosystem. It is integrated with EFS for user management. The NIMBLE Blockchain is not accessed from an API but integrated into the NIMBLE Platform. The architecture is illustrated in Figure 163.

The Blockchain and Smart Contracting services hide the details of writing code for transaction families, dealing with cryptography and signing messages directly against the ledger. Hyperledger Sawtooth and Hyperledger Fabric provide the ledger implementations for the services. These are open source, private and permissioned distributed transaction ledgers [D5.2: EFPF Security and Governance] suitable for EFPF applications. Private blockchains are deployed and used by a restricted set of participants and the consensus algorithms are not based on a currency or consensus algorithms with negligible power consumption, because the stakeholders that form the network control the participation and do not need special incentives for participating apart from the business transactions they already established. Permissioned blockchain networks are suitable for enterprise solutions because with the use of permissions and access rights the participants control who can access the data stored on these networks and define the actions they can perform.

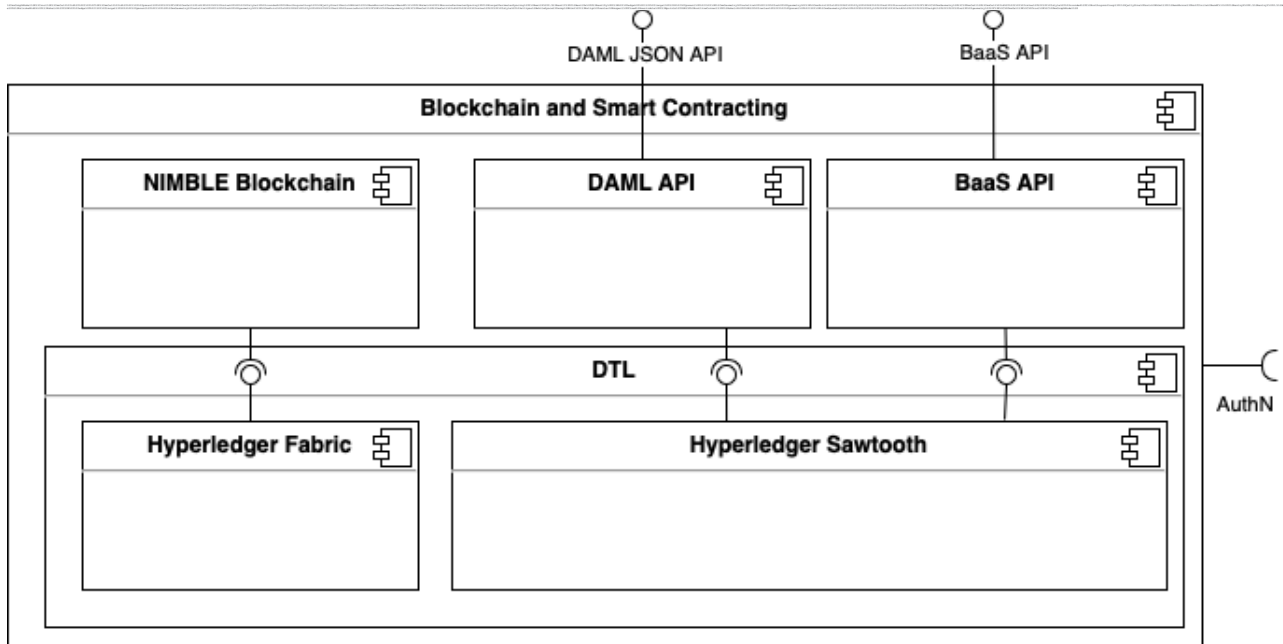


Figure 163. Architecture of Blockchain and Smart Contracting Services

### Supply Chain as a Service API

The BaaS provides three general-purpose smart contracts, or in Hyperledger Sawtooth terminology, transaction families, that allow an EFPF application developer to define customized blockchains for transaction logs, audit trails and transportation involving multiple organizations: the Schema Transaction Family, the Identity Management Transaction Family and the Track and Trace Transaction Family. The API is illustrated in Figure 164.

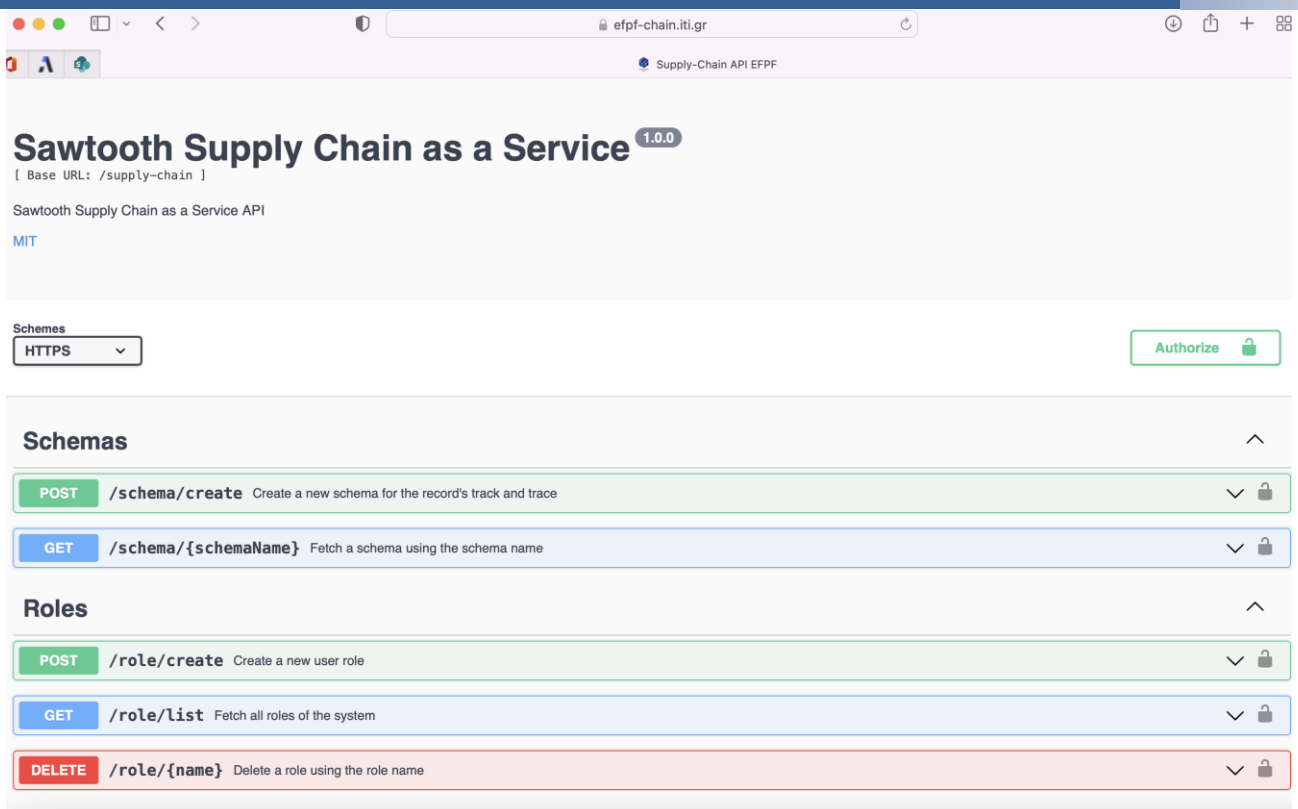


Figure 164. Supply Chain as a Service API

## Schema Transaction Family

Schema transaction family provides a reusable, standard approach to defining, storing, and consuming properties within blockchain.

Schema transaction processor has been developed in Hyperledger Sawtooth and is used to store and retrieve asset properties. To properly store and validate these properties, we need property definitions, which minimally include the property's type (integer, string, enum, etc.). In addition, the properties (for example, asset description, GPS location, or asset attributes) should always be stored and exchanged using the same format.

## State

All Schema objects are serialized using protocol buffers (protobufs) before being stored in state. These objects include Schema, PropertyDefinition and PropertyValue. Schemas are stored in a list to handle hash collisions.

A property is defined using a **PropertyDefinition**, which includes the following:

- Data type (bytes, Boolean, number, string, enum, struct, Latitude/Longitude)
- Name (The name of the Property)
- Type description (An optional description of the field)
- Optionality (whether or not the field is required)

A property value is defined using a **PropertyValue**, which includes the following:

- Data type (bytes, Boolean, number, string, enum, struct, Latitude/Longitude)



- Name (The name of the property value)
- Corresponding value of data type

Property definitions are collected into a **Schema**, which defines all the possible properties for an item that belongs to a given schema. Schemas include the following:

- a name (The name of the schema)
- a description (An optional description of the schema)
- an owner (The public key of an owner, or an organization id)
- a list of PropertyDefinitions

## Addressing

Schemas are stored under the schema namespace. For each schema, the address is formed by concatenating the namespace, and the first 64 characters of the SHA-512 hash of the schema name

## Payload

SchemaPayload contains an action enum and the associated action payload. This allows for the action payload to be dispatched to the appropriate logic. Only the defined actions are available and only one action payload should be defined in the each SchemaPayload.

The defined actions are:

- the creation of a schema and
- the updating of a schema

## Identity Management Transaction Family

The Identity Management Transaction Family (idm) is designed to track the identities of the actors involved in the supply chain. These actors could be agents, users, trucks, and the organizations they represent. The roles that the users play within their organizations are also tracked. This information can be used to determine who is allowed to interact with the supply chain, and to what extent they are allowed to interact with the supply chain.

## State

All Identity Management Transaction Family objects are serialized using Protocol Buffers before being stored in state.

User - A user consists of five fields:

- `public_key`: A user's cryptographic public key. Only one user can belong to the public key.
- `org_id`: The identifier of the organization to which the user belongs.
- `active`: Whether the user is currently considered active at the organization.
- `roles`: A list of roles the user has within the organization.

- metadata: A list of key value pairs describing organization specific data about the user.

Agent - An Agent consists of five fields:

- public\_key: An agent's cryptographic public key. Only one agent can belong to the public key.
- org\_id: The identifier of the organization to which the agent belongs.
- active: Whether the agent is currently considered active at the organization.
- roles: A list of roles the agent has with the organization.
- metadata: A list of key value pairs describing organization specific data about the agent.

Organization - An organization has four fields:

- id: A unique identifier for the organization.
- name: A user defined name for the organization.
- address: A physical address for the organization.
- metadata: A list of key value pairs describing data about the organization.

Truck - A truck has four fields:

- org\_id: The identifier of the organization to which the truck belongs.
- type: A user defined type for the truck.
- plates: The plates of the truck.
- active: Whether the truck is currently considered active at the organization.

## Addressing

Organizations are stored under the idm namespace. For all the organizations, the address is formed by concatenating the namespace, the special policy namespace "00", the first 30 characters of the SHA-512 hash of the word "orgs".

Users are stored under the idm namespace. For each organization, the address of its users is formed by concatenating the namespace, the special policy namespace "22", the first 30 characters of the SHA-512 hash of the word "users" and the first 30 characters of the SHA-512 hash of the org\_id.

Agents are stored under the idm namespace. For each organization, the address of its agents is formed by concatenating the namespace, the special policy namespace "11", the first 30 characters of the SHA-512 hash of the word "agents" and the first 30 characters of the SHA-512 hash of the org\_id.

Agents are stored under the idm namespace. For each organization, the address of its trucks is formed by concatenating the namespace, the special policy namespace "33", the first 30 characters of the SHA-512 hash of the word "trucks" and the first 30 characters of the SHA-512 hash of the org\_id.

## Payload

IdmPayload contains an action enum and the associated action payload. This allows for the action payload to be dispatched to the appropriate logic. Only the defined actions are

available and only one action payload should be defined in the each IdmPayload. The defined actions are:

- Create a User
- Update a User
- Deactivate a User
- Create an Organization
- Update an Organization
- Create an Agent
- Update an Agent
- Deactivate an Agent
- Create a Truck
- Update a Truck

### **Track and Trace Transaction Family**

The Track and Trace transaction family (tnt) allows users to track assets as they move through a supply chain. Records for assets include a history of ownership and custodianship, as well as histories for a variety of properties such as temperature and location. These properties are managed using the Schema transaction family.

### **State**

All Track and Trace transaction family objects are serialized using Protocol Buffers before being stored in state. These objects include:

1. Records/Assets,
2. Proposals, and
3. Properties

Records - Records represent the assets being tracked by Grid Track and Trace. Almost every transaction references a Record. A Record contains:

- record\_id: a unique identifier about the record.
- schema: the name of a Schema.
- owners: organization identifiers that owns the record ordered by timestamp.
- custodians: organization identifiers that has act as custodians of the record ordered by timestamp.
- final: Flag (a Boolean value) indicating whether the Record can be updated.
- modified: The timestamp of last modification of the record, as a Unix UTC timestamp.

Proposals - A Proposal is an offer from the owner or custodian of a Record to authorize another organization as an owner, custodian, or reporter for that Record. Proposals are tagged as being for transfer of ownership, transfer of custodianship, or authorization of a reporter for some Properties. Proposals are also tagged as being open, accepted, rejected, or cancelled. There cannot be more than one open Proposal for a specified role for each combination of Record, receiving organization, and issuing organization. A Proposal contains:

- record\_id: The Record that this proposal applies to.
- timestamp
- issuing\_organization: The organization identifier of the organization sending the Proposal
- receiving\_organization: The organization identifier of the organization receiving the Proposal
- role: What the Proposal is for -- transferring ownership, transferring custodianship, or authorizing a reporter
- properties: The names of properties for which the reporter is being authorized
- status: Either open, accepted, rejected or cancelled
- terms: The human-readable terms of transfer

Properties - Historical data pertaining to a particular data field of a tracked object are stored as Properties, represented as a list of values accompanied by a timestamp and a reporter identifier. The whole history of updates to Record data is stored in current state because this allows for more flexibility in writing transaction rules. A Property contains:

- name: The name of the Property, e.g., "temperature"
- record\_id: The Record that this property applies to
- property\_definition: The name of the PropertyDefinition that defines this record
- reporters: The authorized organization to send updates
- current\_page: The page to which new updates are added, maximum value  $16^4$
- wrapped: A flag indicating whether the first  $16^4$  pages have been filled.

A Property Page contains:

- name: The name of the Property, e.g., "temperature"
- record\_id: The Record that this property applies to
- reported\_values: The historical values of the record's property

## Addressing

Track and Trace objects are stored under the namespace tnt.

After its namespace prefix, the next two characters of a Track and Trace object's address are a string based on the object's type:

- Record: “aa”
- Property / PropertyPage: “bb”
- Proposal: “cc”

The remaining 62 characters of an object’s address are determined by its type:

- Record: The first 62 characters of the SHA-512 hash of the identifier of its associated Record
- Property: The first 36 characters of the hash of the identifier of its associated Record plus the first 22 characters of the hash of its Property name plus the string “0000”.
- PropertyPage: The first 36 characters of the hash of the identifier of its associated Record. The first 22 characters of the hash of its Property name and the hex representation of the current\_page of its associated Property left-padded to length 4 with 0s.
- Proposal: The first 36 characters of the hash of the identifier of its associated Record. The first 26 characters of its receiving\_organization.

## Payload

All Track and Trace transactions are wrapped in a tagged payload object to allow for the transaction to be dispatched to appropriate handling logic. The defined actions are:

- Create Record
- Finalize Record
- Update Properties
- Create Proposal
- Answer Proposal
- Revoke Reporter

## DAML JSON API

EFPF Smart Contracting platform based on DAML (Data Asset Modelling Language, an open source (Apache 2.0), high-level domain-specific language that provides an abstraction layer on top of both traditional databases such as PostgreSQL, and blockchain implementations, recently including Hyperledger Sawtooth and Hyperledger Fabric. It is rapidly gaining support on multiple platforms. DAML decouples the distributed trust models, data schemas and business logic - the smart contracts - from the implementation details of communication, cryptography, distributed data stores and synchronization. DAML is based on the functional programming language Haskell and designed to build distributed applications by describing data schemas, smart contracts, and identity management. DAML promises a business oriented, declarative way to build distributed applications using blockchains.

The DAML JSON API is used to upload DAML smart contract applications, create, fetch, and query contract instances and participants, and execute transactions on contracts.

The full specification can be found at [<https://docs.daml.com/1.17.0/json-api/index.html>], however some example uses are described below.

#### 1. HTTP Status Codes

The JSON API reports errors using standard HTTP status codes. It divides HTTP status codes into 3 groups indicating:

2. success (200)
3. failure due to a client-side problem (400, 401, 404)
4. failure due to a server-side problem (500)

The JSON API can return one of the following HTTP status codes:

- 200 - OK
- 400 - Bad Request (Client Error)
- 401 - Unauthorized, authentication required
- 404 - Not Found
- 500 - Internal Server Error

## Examples

Creating a Contract of type IncomingParts in the TrackTrace application

URL: /v1/create

Method: POST

Content-Type: application/json

```
{
  "templateId": "TrackTrace:IncomingParts",
  "payload": {
    "incomingGoodsDepartment": "Incoming_Goods_Department",
    "qualityManagementDepartment": "Quality_Management_Department",
    "incomingPartInfo": {
      "incomingPartNumber" : "DF333",
      "incomingPartDescription" : "Decor Fabric",
      "incomingPartDeliveryNoteReferenceNumber" : "incomingPartDeliveryNoteReferenceNumber",
      "incomingPartCertificateReferenceNumber" : "incomingPartCertificateReferenceNumber",
      "incomingPartSupplier" : "Decor Fabric Hamburg GmbH",
      "incomingPartBatchNumber" : "052018",
      "internalBatchNumber" : "22-033"
    }
  }
}
```

Query for Contracts of type IncomingParts in the TrackTrace application where internalBatchNumber matches 22-033

URL: /v1/query

Method: POST

Content-Type: application/json

Content:

```
{
  "templateIds": ["TrackTrace:IncomingParts"],
  "query": {
    "incomingPartInfo": { "internalBatchNumber": "22-033"}
  }
}
```

Uploading a DAR (DAML Archive) file containing a compiled project

URL: /v1/packages

Method: POST

Content-Type: application/octet-stream

Content: <DAR bytes>

The content (body) of the HTTP request contains raw DAR file bytes, without any encoding.

### **NIMBLE Logistic Contract**

The NIMBLE Logistic Contract (Figure 165) uses a Hyperledger Fabric blockchain as an integrated audit trail, data provenance and verification engine for the trading & logistics data exchanged between trade partners in NIMBLE platform. E.g., products moving through the different stages of the supply-chain process or business documents such as a certificate of origin or purchase order.

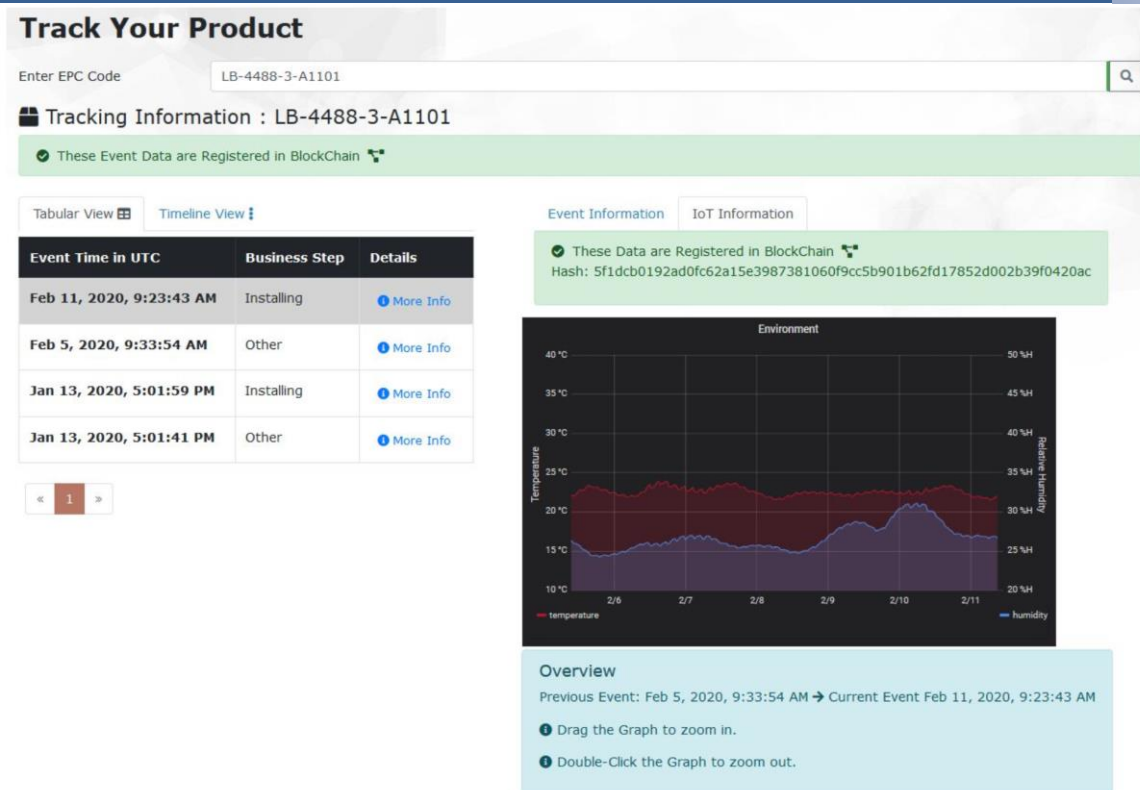


Figure 165. User interface of NIMBLE Logistic Contract

The NIMBLE Logistic Contract Chain Code includes two Smart Contracts that work in parallel:

- Logistic Contract, which handles the logistic process of a dispatched order from the business process service.
- Identity Contract, which manages identities of platform users from peer organizations (NIMBLE, EFPF).

These are called within NIMBLE, from the NIMBLE Logistics Frontend. The project can be extended for other use cases by contributions to the chaincode [<https://github.com/nimble-platform/logistic-contract>].

### 3.2.1.13 Data Analytics

#### 3.2.1.13.1 Anomaly Detection Service

The Anomaly Detection (AD) service comprises of two components, the Model builder, and the Model Manager, which are described below:

- **Model builder:** This module is a Web User Interface to allows the machine model building by workflow, each workflow is a Machine learning algorithm used to train, test and generated a model, the model can be downloaded onto the local machine, and it also can be deployed by the Model manager.
- **Model manager** This module uses the generated models to subscriber them to a Broker service and performs in real-time the Anomaly detection over a data streaming.



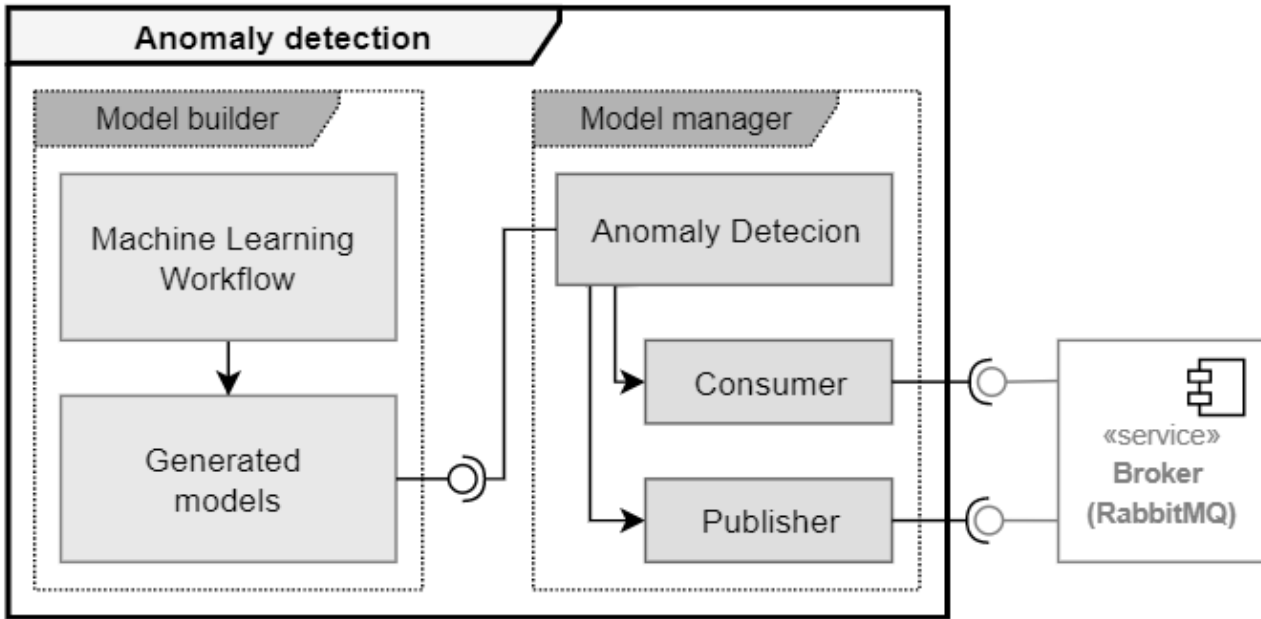


Figure 166. Anomaly Detection Architecture Diagram

**Anomaly detection Backend REST API:**

REST Endpoint	HTTP Method	Description
/	GET	Uploading the Index page, Initialization of paths-to-Datasets/Images/Models & The Model manager list
/algorithmsMenu	GET	Adding the Algorithm-menus card into the index template
/formDLEvpc	GET	Adding the DLE-algorithm-form card into the index template
/uploadDatasets	GET	Adding the Importing-datasets card into the index template
/doCorrAnalysis	POST	Making the Correlation-Matrix with the chosen columns
/anomalyDetection	POST	Building a ML Model by training and testing using the chosen columns
/downloadModel	GET	Download zip file of specified Model
/downloadTrainingData	GET	Passes information to download File to download training data CSV
/downloadFile	GET	Download a specified file
/createTrainingData	POST	Subscribe to a topic and create a csv file with messages received from the topic
/stopTrainingData	POST	Stop subscribing to a topic for creating training data
/addModel	POST	Add a model to the model manager user interface
/deleteFilesByModelName	DELETE	Delete associated files for a given model name
/deleteImagesByModelName	DELETE	Delete associated images for a given model name
/deployerDashboard	GET	Get a list of deployed/stored models
/mlModelOnStream	POST	Subscribe a model to a topic and begin processing stream data.

/stopDeployedModel	POST	Stop a deployed model
/deleteModel	DELETE	Delete a specified model
/deployerVizGrafana	GET	Display Visualisation of processed messages from stream data
/publisherForm	GET	Add the publisher form to the index template
/trainingForm	GET	Add the Training form to the index template
/publisher	POST	Begin publishing messages to the specified topic
/image	GET	Download a specific image by name
/lagrama	GET	Display the lagrama dashboard

Figure 167. Anomaly Detection Tool HTTP REST API's

### Anomaly Detection Tool GUI's:

Anomaly Detection Tool GUI's is presented in Figure 168 - Figure 180:

#### Anomaly Detection Service

Home
New Model
Model Manager
Testing

Anomaly detection is about finding uncommon conditions that can drive to errors and non-desired results. If you want to improve the quality of your products or processes, sport malfunctioning equipment or sport faulty raw material, the anomaly detection service can help you.

Anomaly Detection Service is empowered with AI algorithms that can predict in realtime sources of defects. Plug your devices and sensor in eFPF message broker, train one of our models and you are ready to go. Analyse results with our intuitive dashboard and if you want to extend your apps with our predictions, it is as simple as connecting you app to an eFPF message broker queue

Figure 168. Anomaly Detection Tool: Home Page

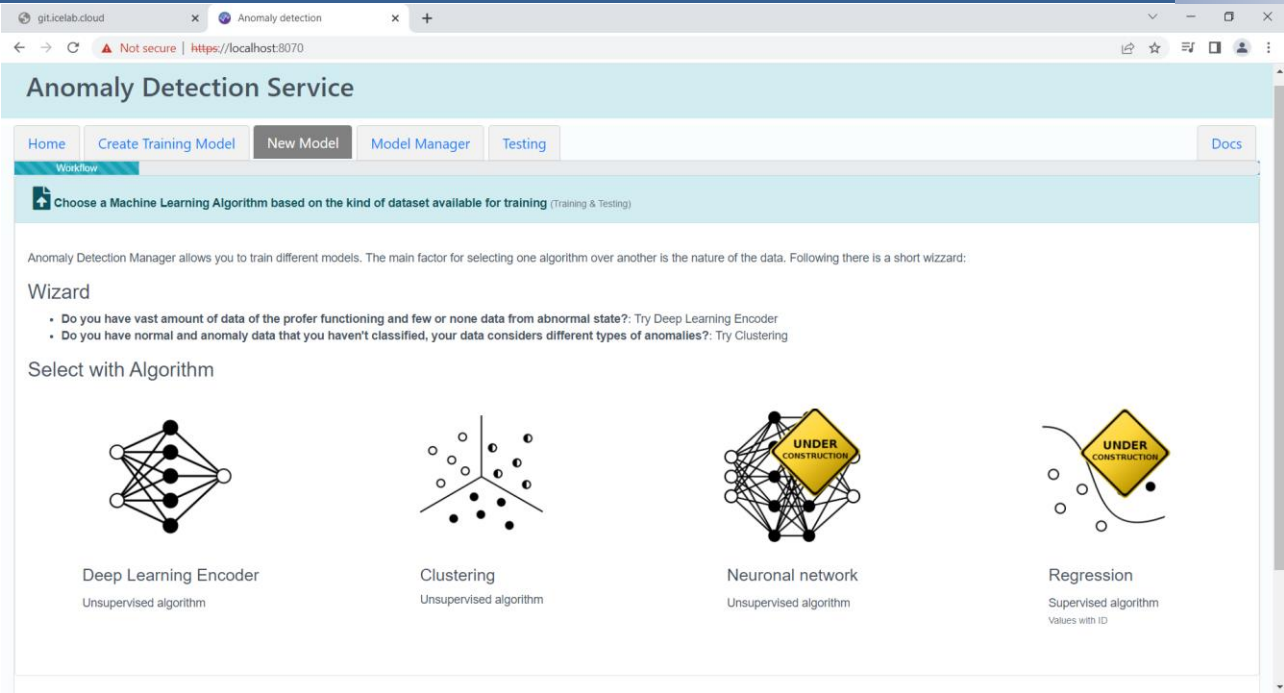


Figure 169. Anomaly Detection Tool: Select Algorithm Page

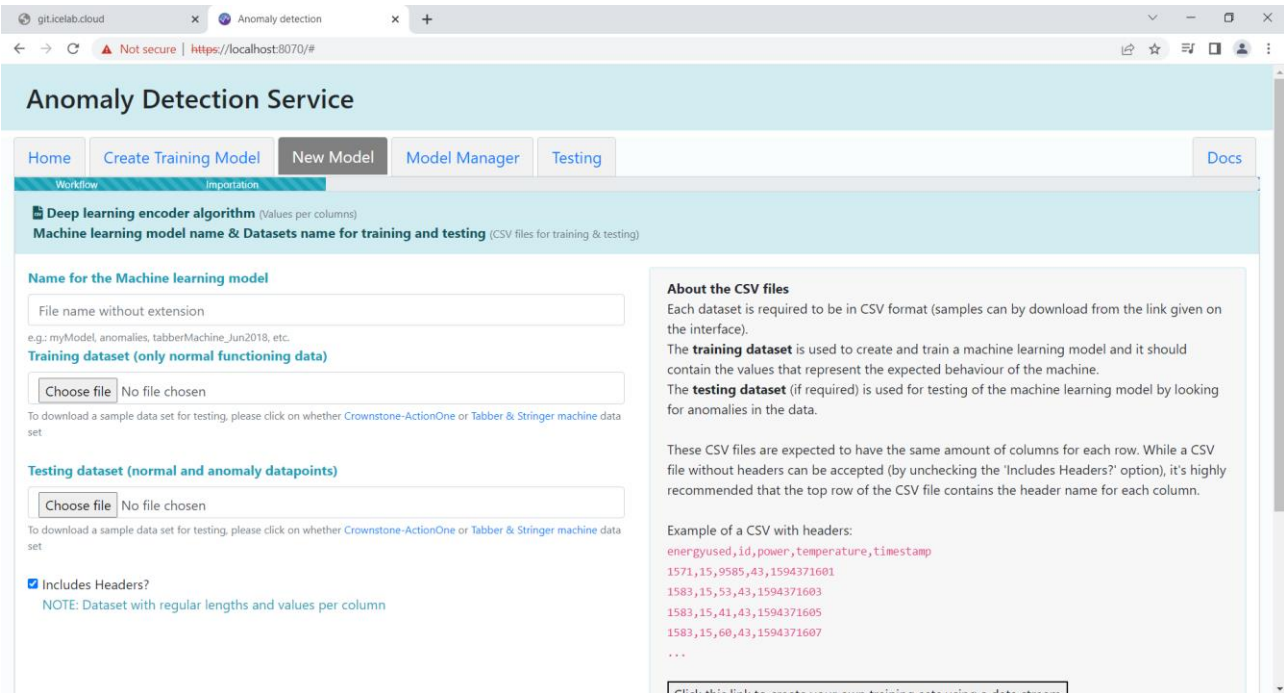


Figure 170. Anomaly Detection Tool: Create Deep Learning Encoder Upload Datasets

Dataset for training: FactoryDataTrain.csv

Select All Columns

Show 10 entries

	motor1 Numeric T1	motor2 Numeric T1	motor3 Numeric T1	motor4 Numeric T1	motor5 Numeric T1	motor6 Numeric T1	motor7 Numeric T1	motor8 Numeric T1
6819	8675	7134	9856	20000	9567	6830	10859	
6822	8675	7131	9850	20000	9565	6829	10859	
6822	8680	7132	9850	20000	9567	6833	10857	
6822	8678	7132	9853	20000	9561	6830	10853	
6829	8683	7132	9850	20000	9572	6833	10860	
6829	8666	7131	9856	20000	9556	6830	10851	
6830	8675	7127	9858	20000	9565	6852	10868	
6831	8665	7124	9850	20000	9590	6834	10856	
6832	8655	7134	9853	20000	9551	6830	10856	
6833	8686	7131	9853	20000	9568	6846	10866	

Figure 171. Anomaly Detection Tool: View Datasets Section

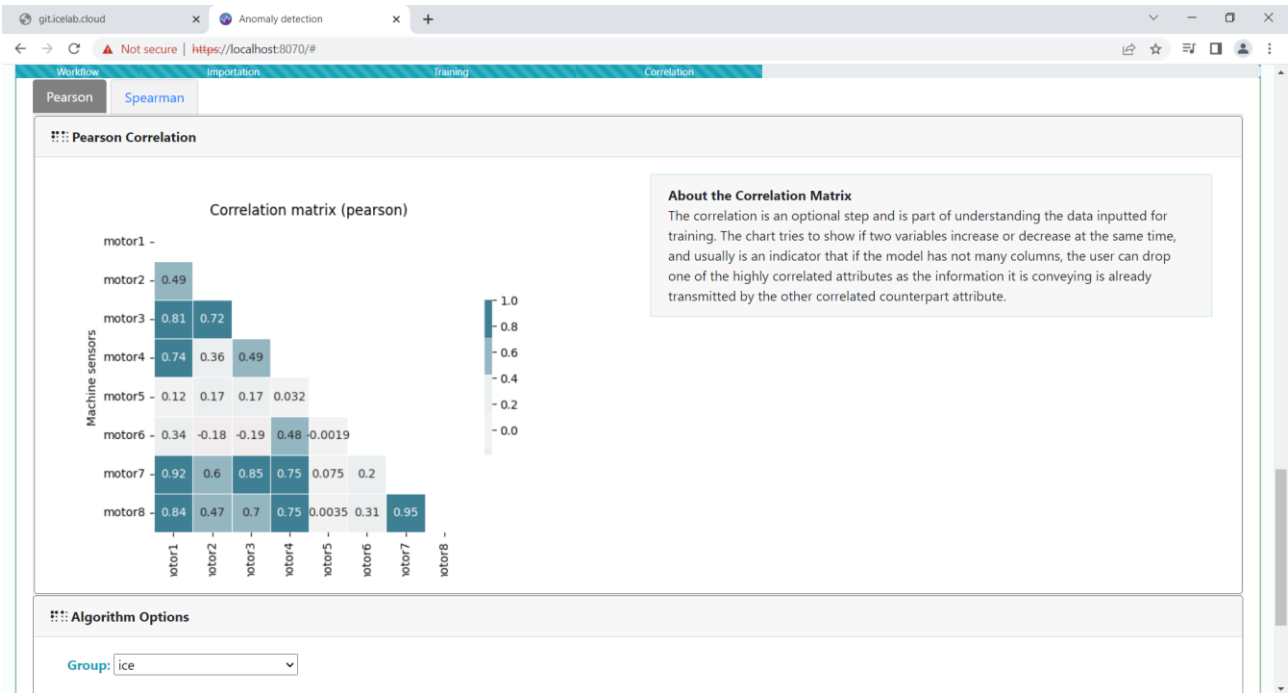


Figure 172. Anomaly Detection Tool: Correlation Index Section

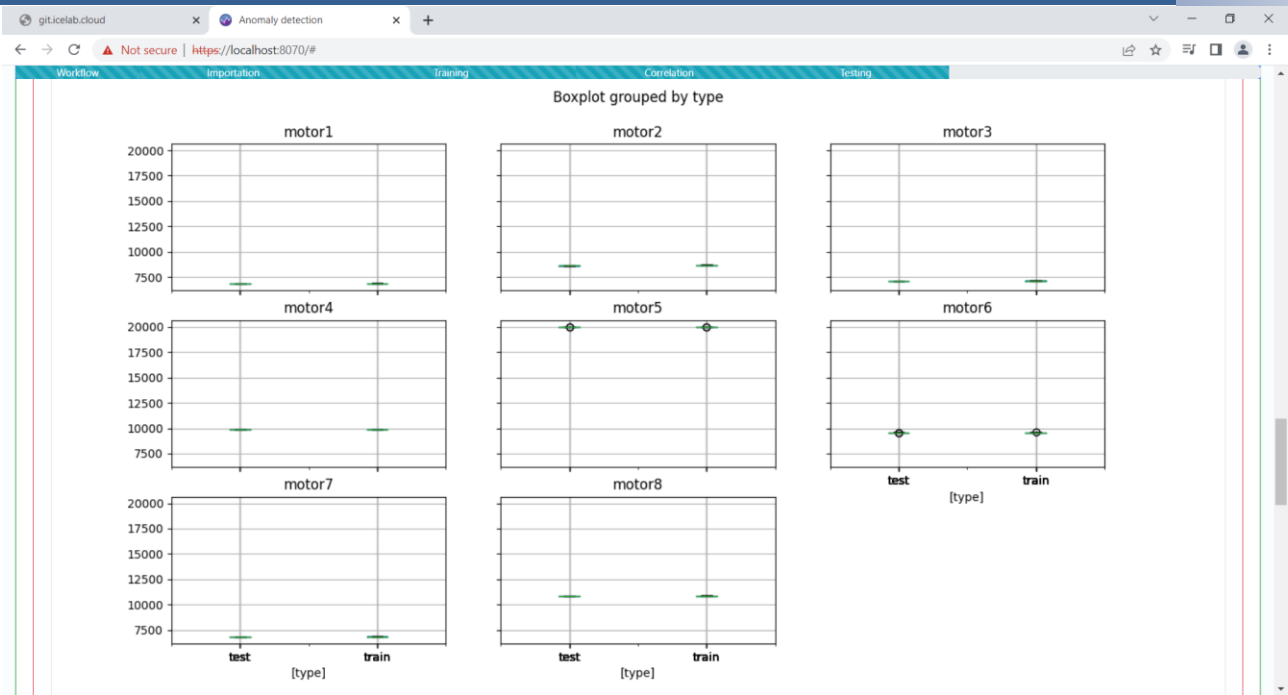


Figure 173. Anomaly Detection Tool: Visualisation of Processed Data

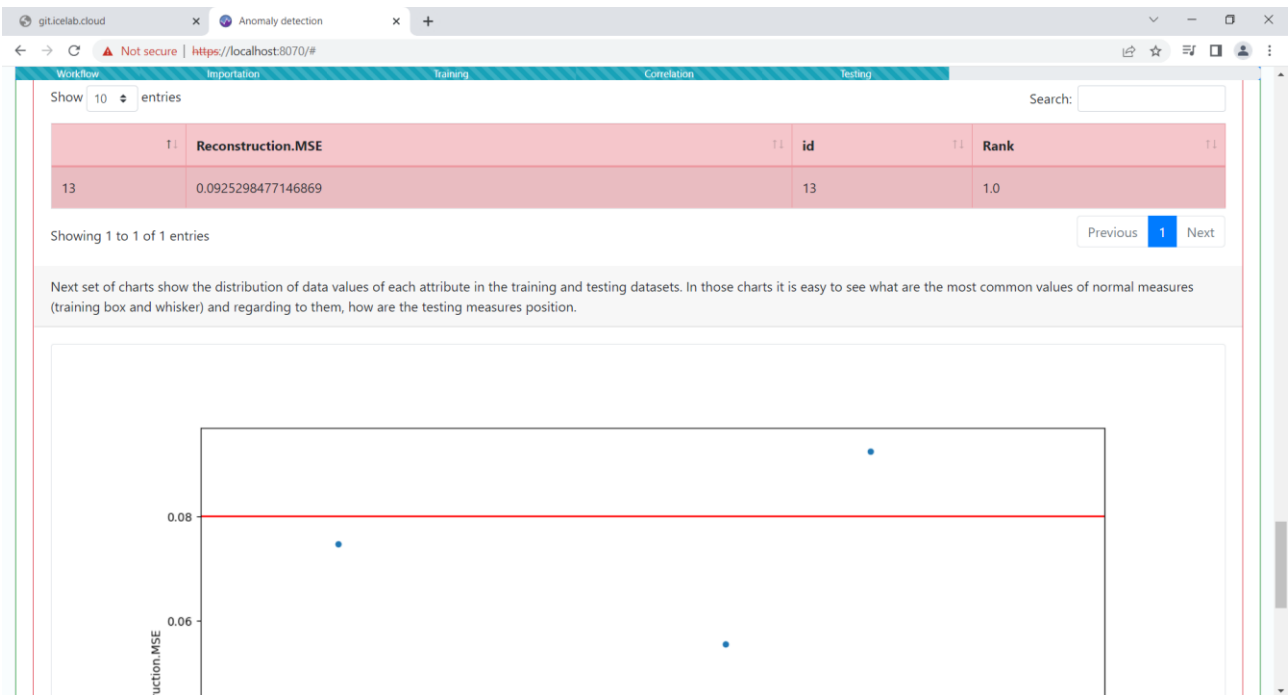


Figure 174. Anomaly Detection Tool: Detected Anomalies Visualisation

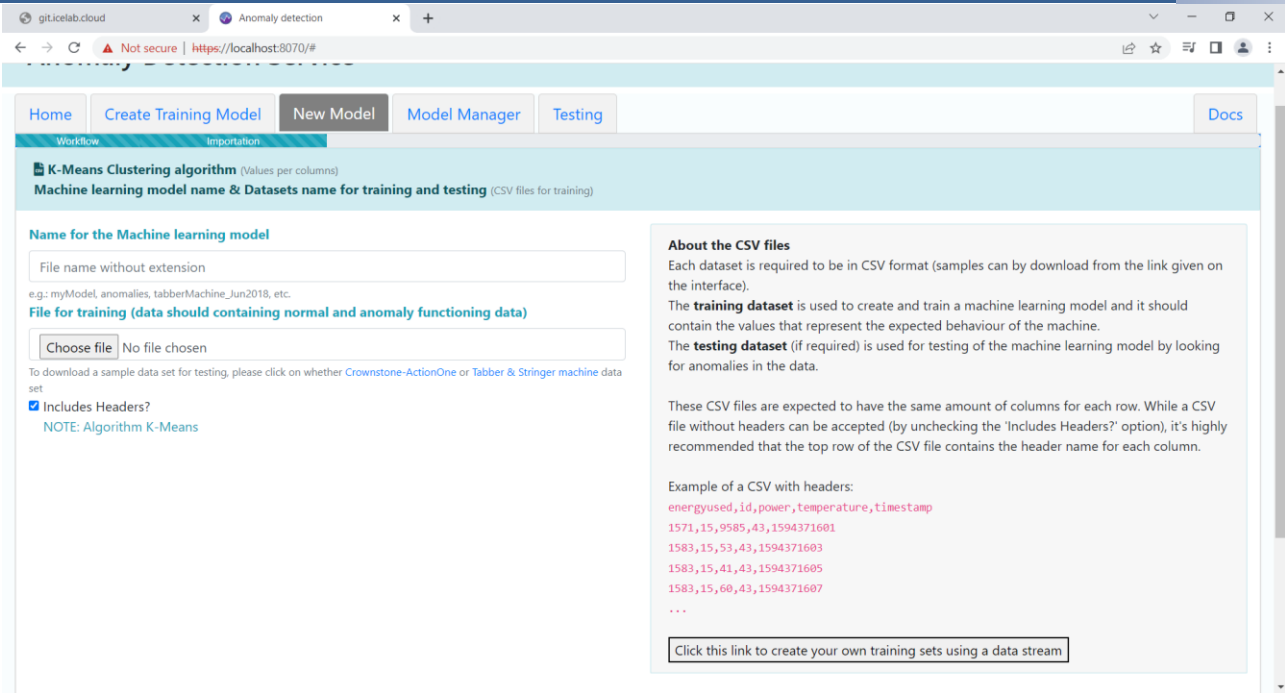


Figure 175. Anomaly Detection Tool: Clustering Algorithm Import Datasets

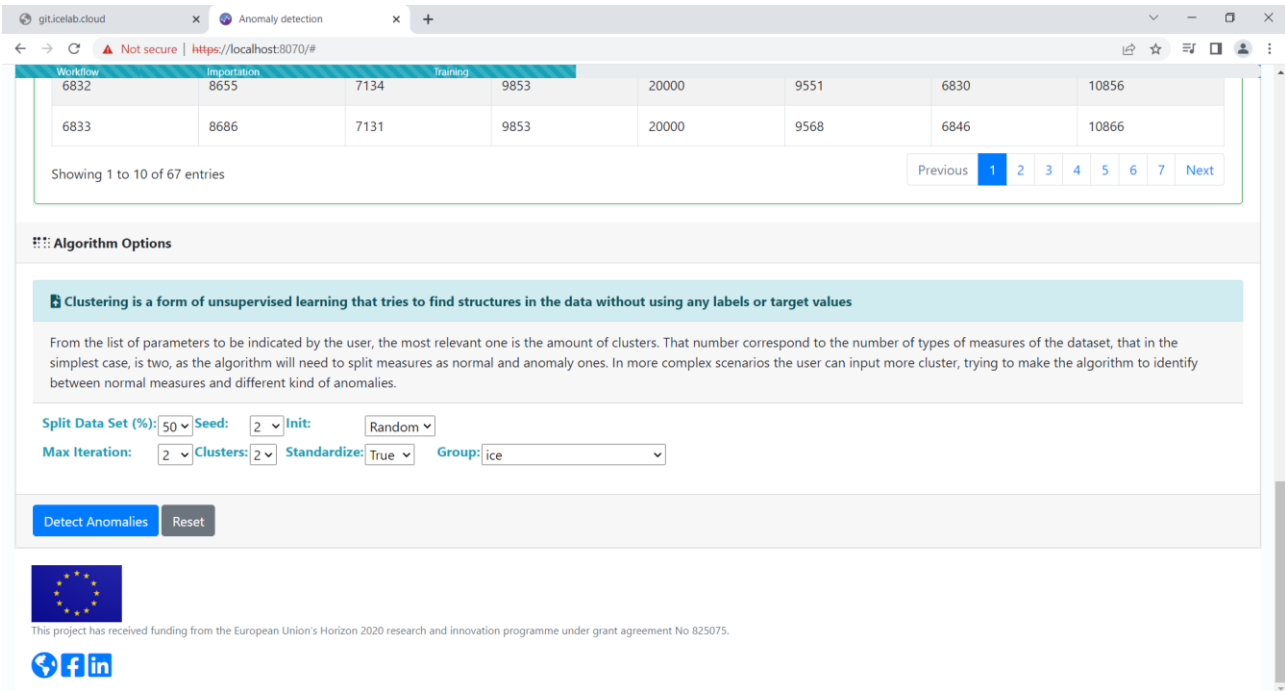


Figure 176. Anomaly Detection Tool: Clustering Algorithm Settings

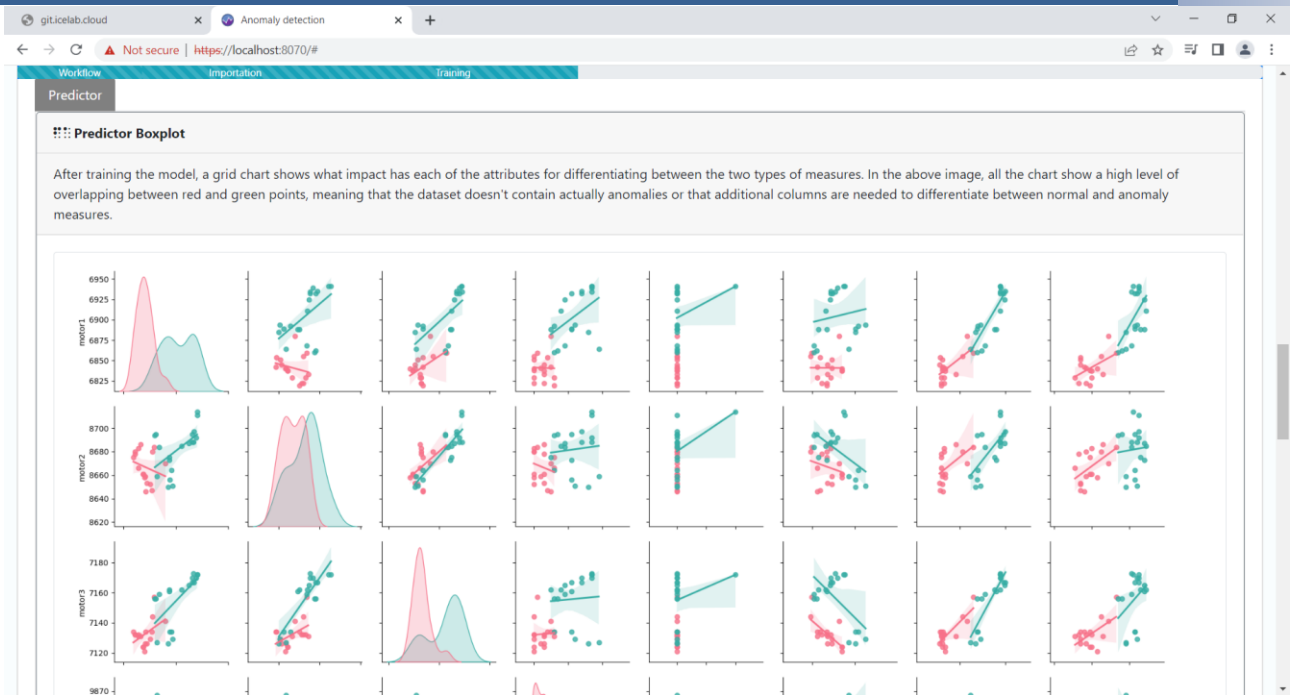


Figure 177. Anomaly Detection Tool: Clustering Visualisations

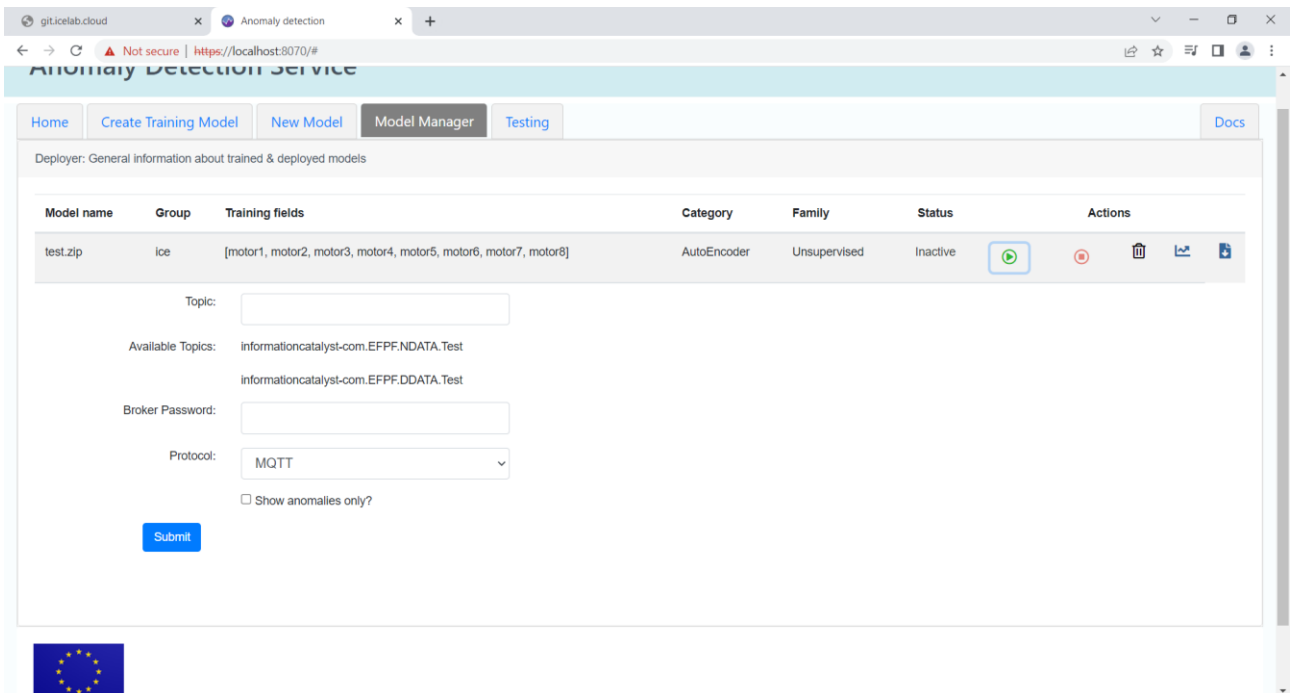


Figure 178. Anomaly Detection Tool: Model Manager With Model Deployment Options

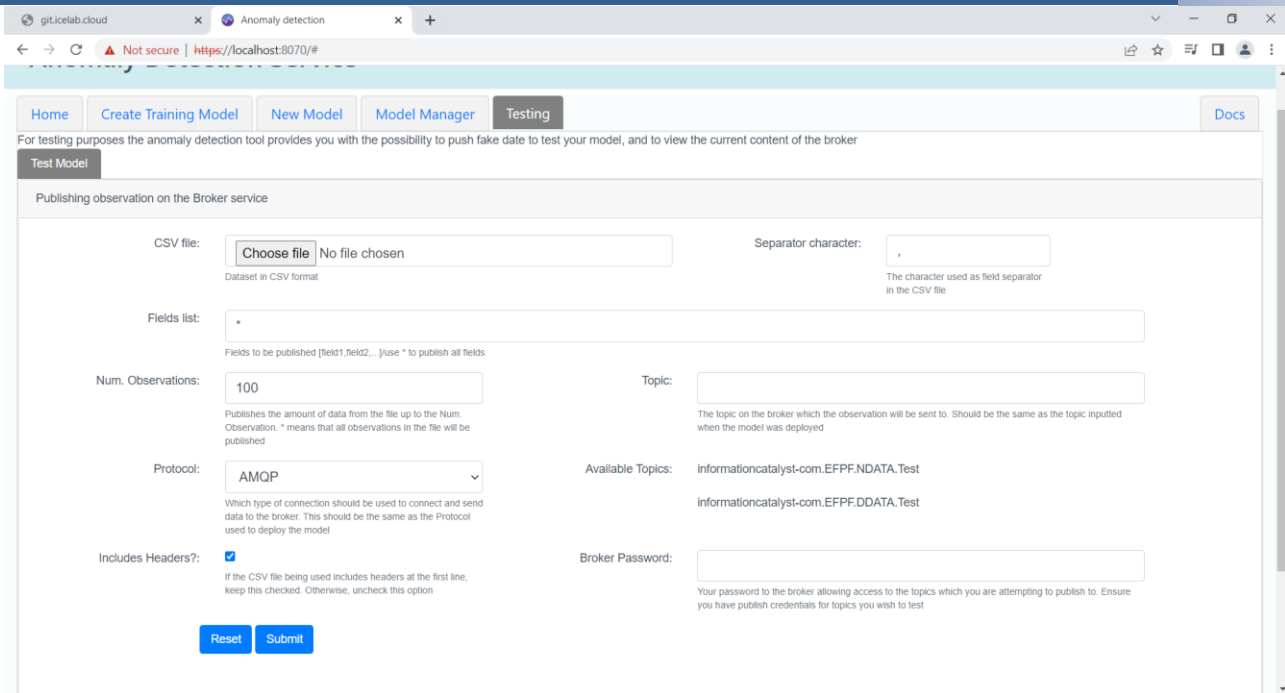


Figure 179. Anomaly Detection Tool: Model Testing Publisher Options

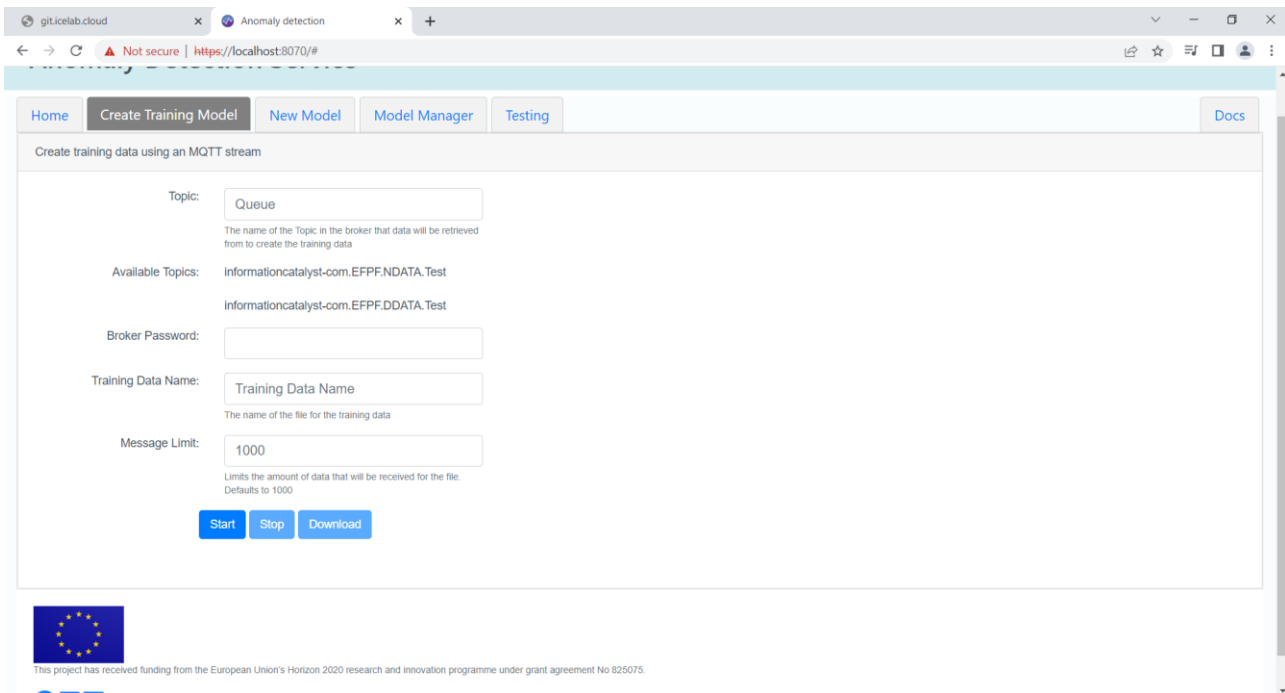


Figure 180. Anomaly Detection Tool: Training Data Subscriber

### 3.2.1.13.2 Visual & Data Analytics Service

The Visual and Data Analytics tool was developed based on web technologies such as AngularJS, JavaScript and ChartJS. The backend algorithm was developed in Python. The deployment was based on micro services logic and Docker is used as the containerization technology. Both REST/HTTP and MQTT protocols are utilized by the analytics tool to communicate with other tools and IoT devices. The Visual and Data Analytics tool's architecture and its interaction with the Data Spine is illustrated in Figure 181. It consumes



data from Data Spine Message Broker using MQTT protocol and Pub/Sub services of EFPF project. Besides this, the tool exposes only web-based graphical interfaces with the user with no further API to be provided. Some examples of the developed UIs are available below in Figure 182 and Figure 183.

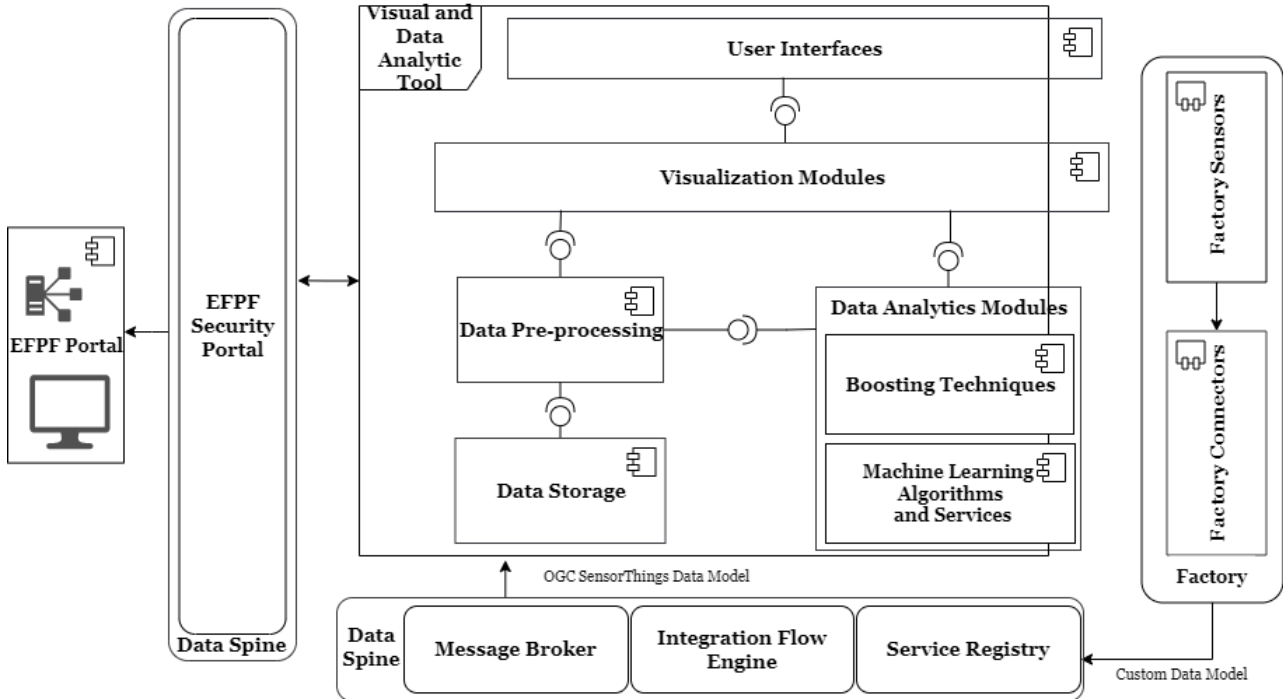


Figure 181. Visuals and Data Analytics Tool Architecture and Data Spine Interaction

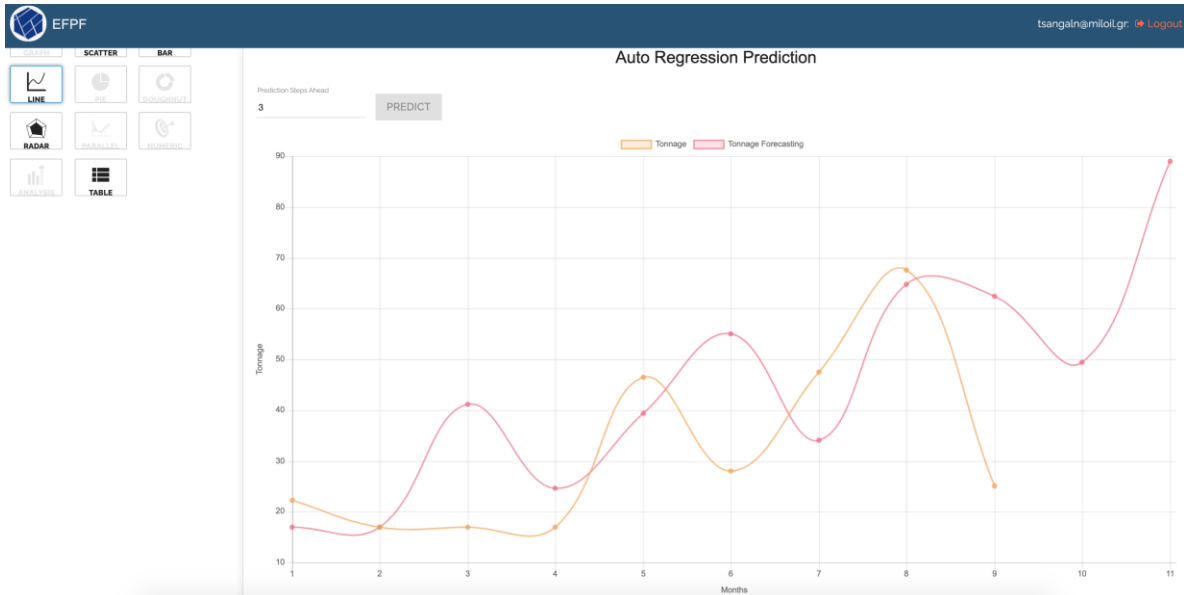


Figure 182. Example of Visual and Data Analytics Tool UI

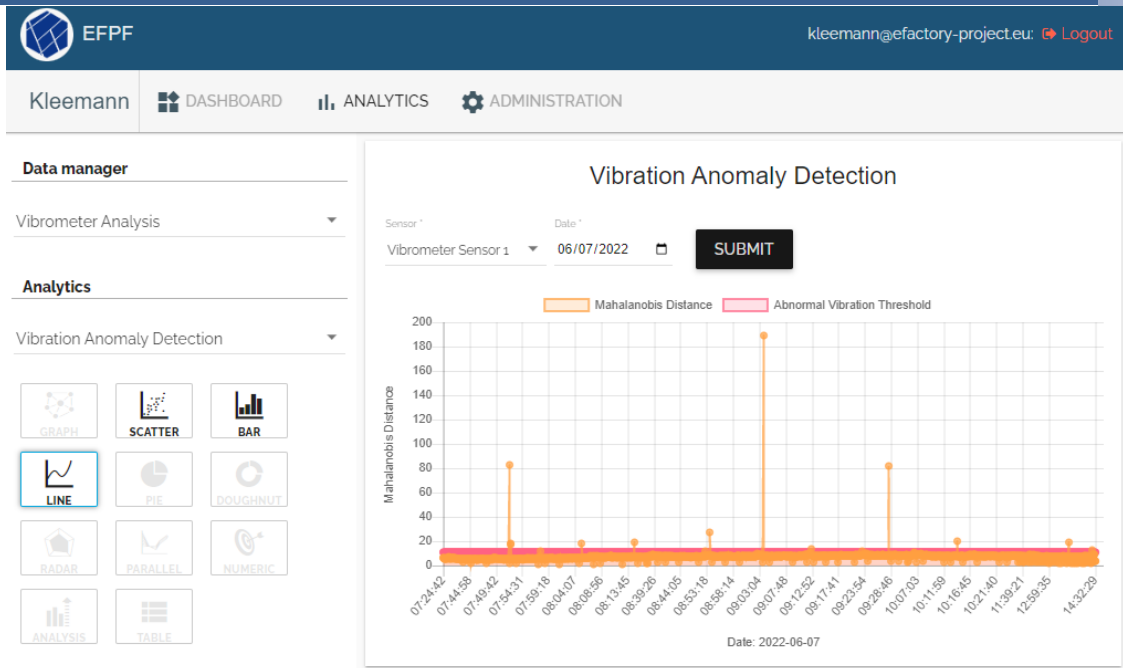


Figure 183: Example (2) of Visual and Data Analytics Tool UI

### 3.2.1.13.3 Deep Learning Toolkit

The Deep Learning Toolkit, depicted in Figure 184, provides an easy way to develop and integrate Deep Learning models with the EFPP platform. The DL models supported by the DLT consume time series data encoded in a JSON formatted OGC-Sensor Things data model. These models allow users to perform activities such as Predictive Maintenance or Price Forecasting.

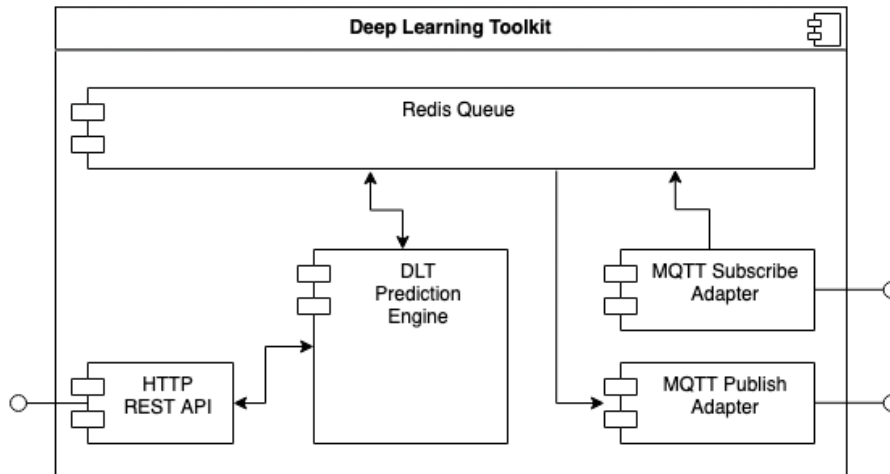


Figure 184. Deep Learning Toolkit Architecture

The Deep Learning Toolkit provides a HTTP REST API as illustrated in Figure 185 to manage both the models and the configuration of the tool. It is integrated with EFS for authentication purposes. Since each instance of the tool is separate each instance will have its separate REST API endpoint.

REST Endpoint	HTTP Method	Description
/info	GET	Retrieves status of the DLT instance
/start	GET	Starts the prediction component of the DLT
/stop	GET	Stops the prediction component of the DLT
/model?file_name={file_name}&model={model}	GET	Retrieves the {file_name} file (weights or architecture) for the {model} model
/model?file_name={file_name}&model={model}	POST	Posts the {file_name} file (weights or architecture) for the {model} model
/model?file_name={file_name}&model={model}	DELETE	Deletes the {file_name} file (weights or architecture) for the {model} model
/config?file_name={file_name}	GET	Retrieves the {file_name} config file
/config?file_name={file_name}	POST	Posts the {file_name} config file
/config?file_name={file_name}	DELETE	Deletes the {file_name} config file

Figure 185. Deep Learning Toolkit API

The DLT provides a MQTT API for input output operations. This API consumes OGC-Sensors Things data containing the information used to make the predictions which are then produced and provided to the user through this API. During the first configuration process it is possible to specify the input and output topic.

- The data pushed on the input topic is compliant with the OGC-ST data model, an example is provided below:

```
{
  "name": "string",
  "description": "string",
  "properties": {},
  "Locations": [
    {}
  ],
  "HistoricalLocations": [
    {}
  ],
  "Datastreams": [
    {
      "name": "string",
      "description": "string",
      "observationType": "string",
      "unitOfMeasurement": {},
      "observedArea": "string",
      "phenomenonTime": "string",
      "resultTime": "string",
      "Observations": [
        {
          "phenomenonTime": "string",
          "resultTime": "string",
          "result": "string",
          "resultQuality": [
            "string"
          ],
          "validTime": "string",
          "parameters": [
            {}
          ],
          "FeatureOfInterest": {
            "name": "string",
            "description": "string",
            "encodingType": "string",

```

```

    "feature": "string"
  }
}
],
"ObservedProperty": {},
"Sensor": {}
}
]
}

```

- The data the DLT publishes on the output topic has the following data model:

```

{
  "result": 0,
  "precision": 0,
  "timestamp": "string"
}

```

### 3.2.1.13.4 Customer Trend Analysis

The Elanyo EFPF Behavioral Predictive Framework (BPF) has been built in Azure, as shown in the following infrastructure diagram in Figure 186. For integration into the EFPF platform, the only used component was EFS Keycloak for security purposes.

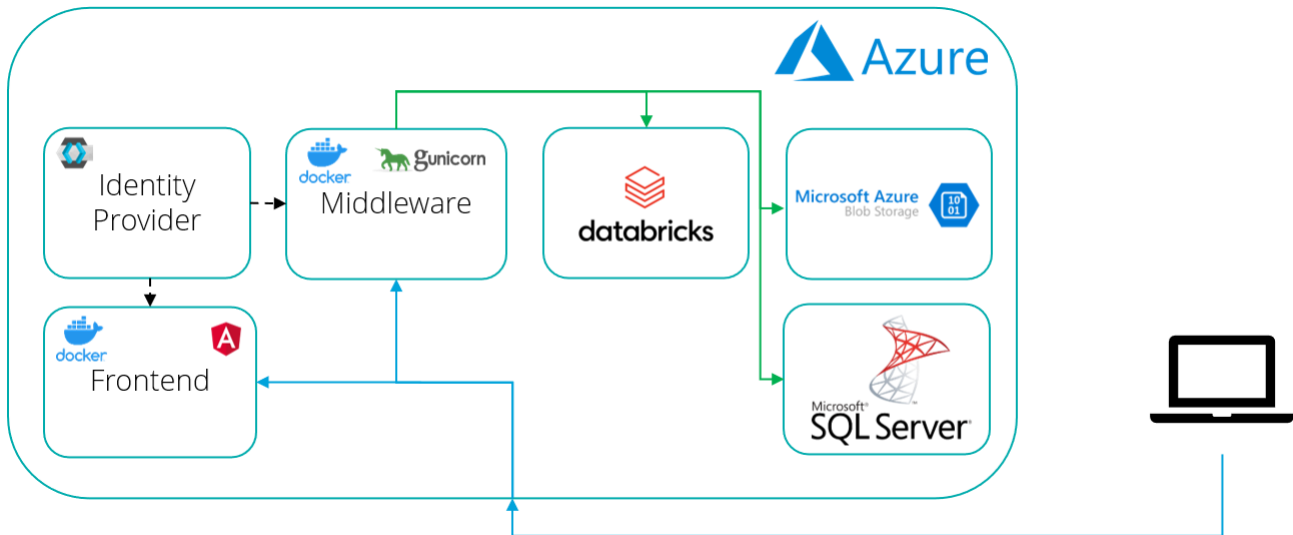


Figure 186. Behavioral Predictive Framework (BPF)

### 3.2.1.13.5 Analytics Integrator Platform (AIP)

The following diagram describes the AIP functionality in a more generic fashion. The purpose of the AIP is to provide means to define analytics flows bringing Data Spine in as a data source/sink together with analytics services. Given the analytics services are callable/scriptable one would be able to describe more complex functionality than it would be possible by having each service configurable via its own user interface. This also has the added benefit of a centralized configuration.

The AIP Architecture Overview is presented in Figure 187:

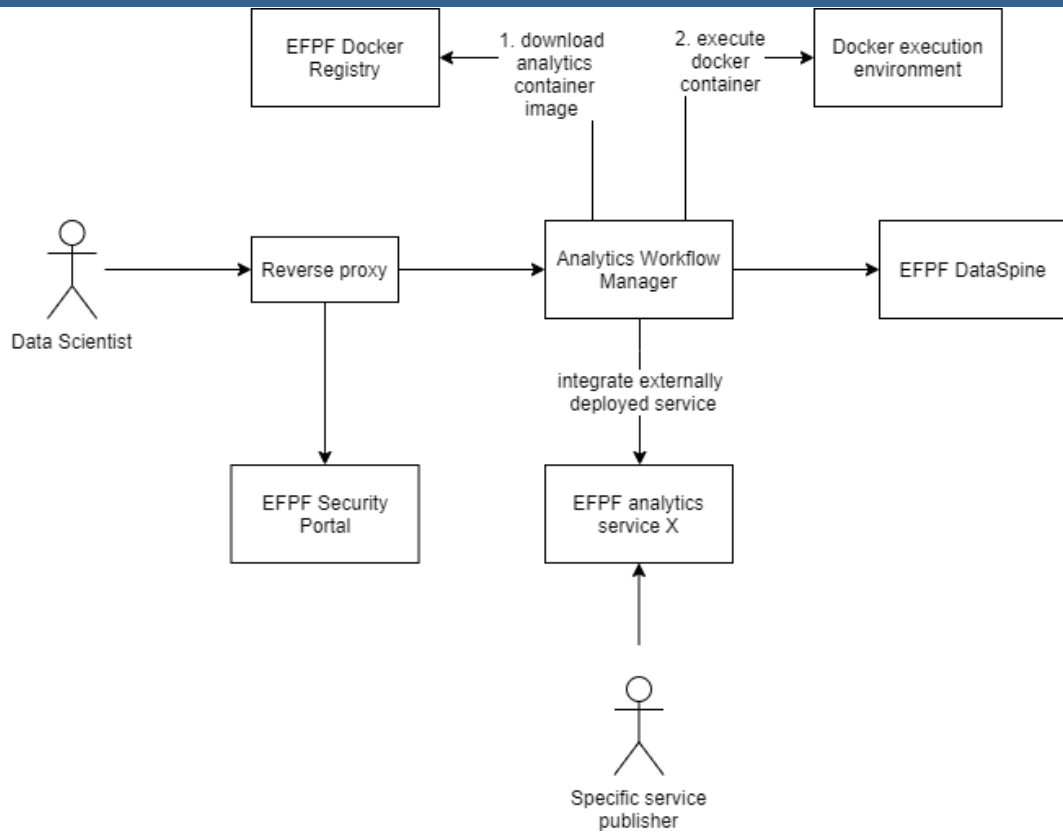


Figure 187. AIP Architecture Overview

### 3.2.1.14 Workflow and Service Automation Platform

The Workflow and Service Automation Platform (WASP) architecture is shown below in Figure 188:

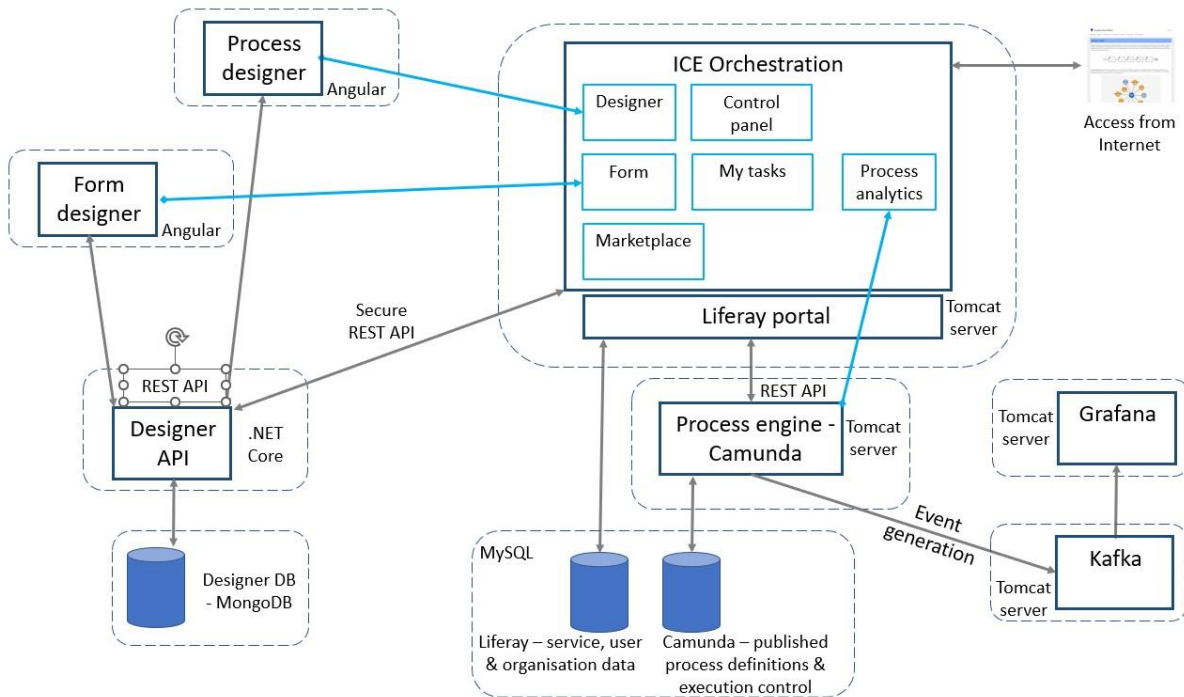


Figure 188. WASP Architecture

WASP enables fast automation of processes, from receiving purchase orders to streamlining machine operations, from handling information from heterogeneous sources to orchestrating external software services and from execution of production activities to interconnectivity of supply chain operations. Furthermore, WASP supports achieving better transparency and surveillance in low-digitized supply chains. WASP can be used through its online portal. Alternatively, WASP can be sourced for private use and inhouse deployment. Either way, users have access to a range of distributed workflow/process management functionalities that make it easy to handle the interplay of distributed services and manual (human based) activities.

The User story for creating and executing a workflow process starts with the Process Designer.

- Create a new process called 'Workflow Process'
- Add a User Task to get a 'Customer Name'
- Create a new form in the Form Designer & associate it with the User Task.

The following graphics shown in Figure 189 - Figure 214 present the steps required to do this.

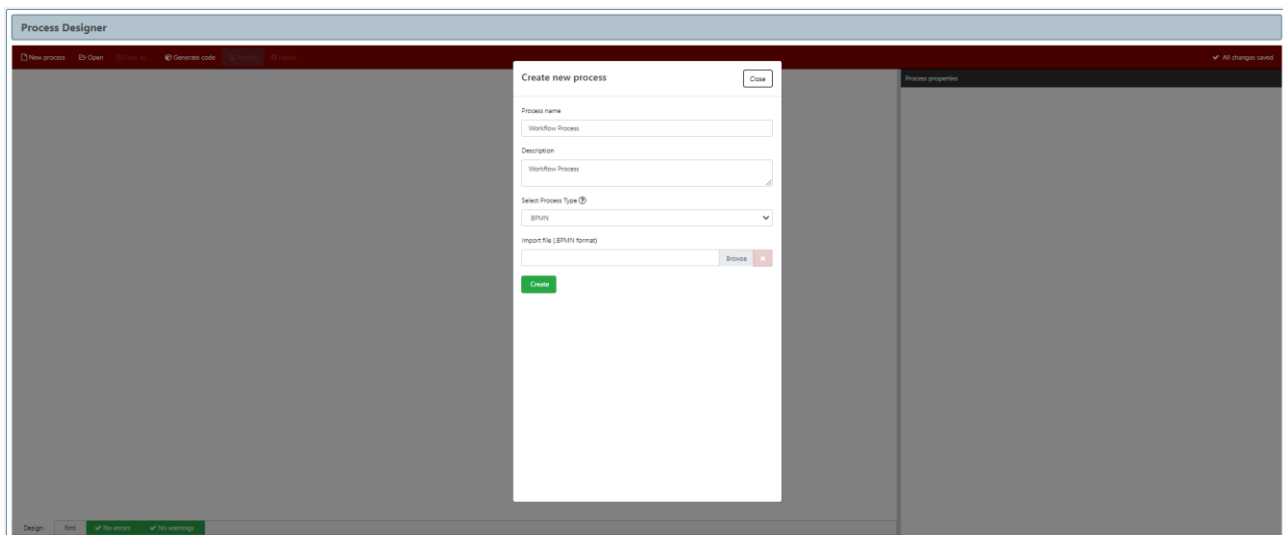


Figure 189. WASP: Create a new process called "Workflow Process"

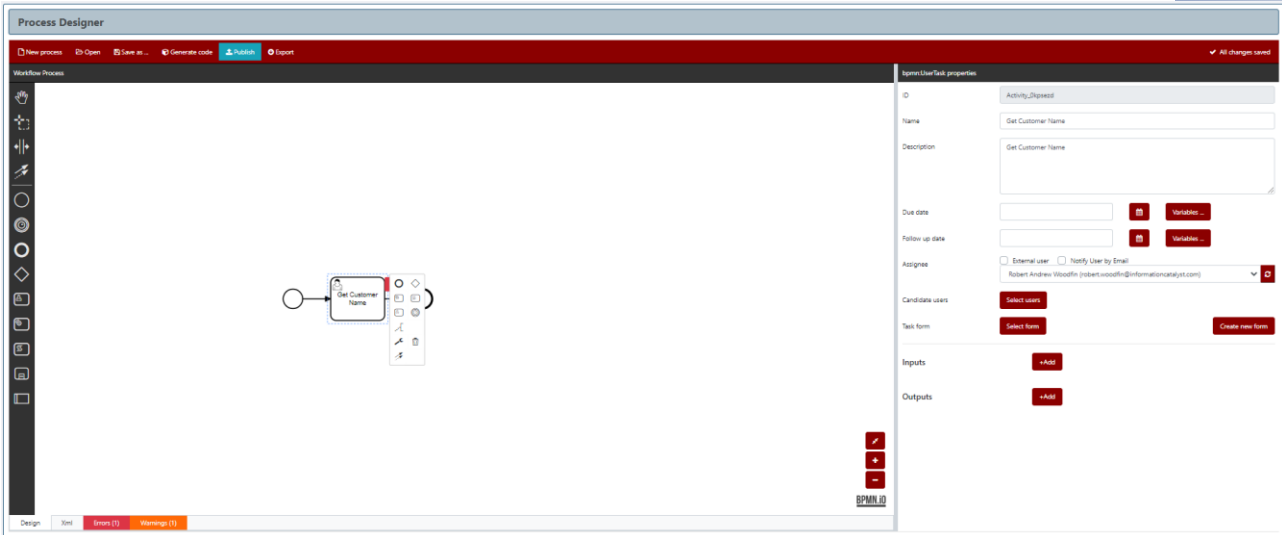


Figure 190. WASP: Add a User Task to get a "Customer Name"

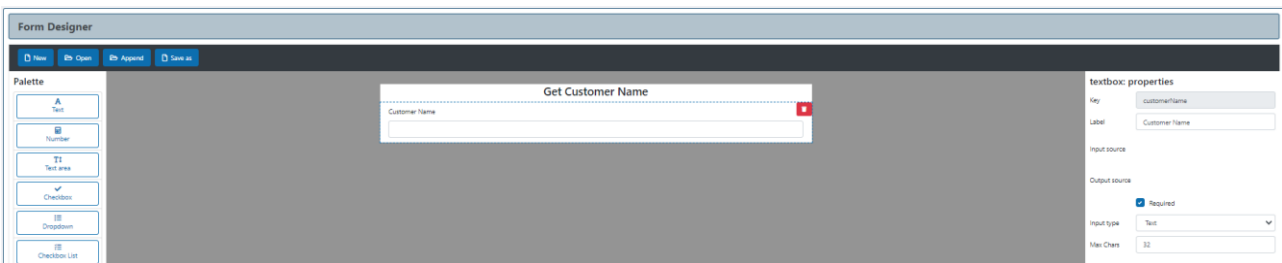


Figure 191. WASP: Create a new form in the Form Designer

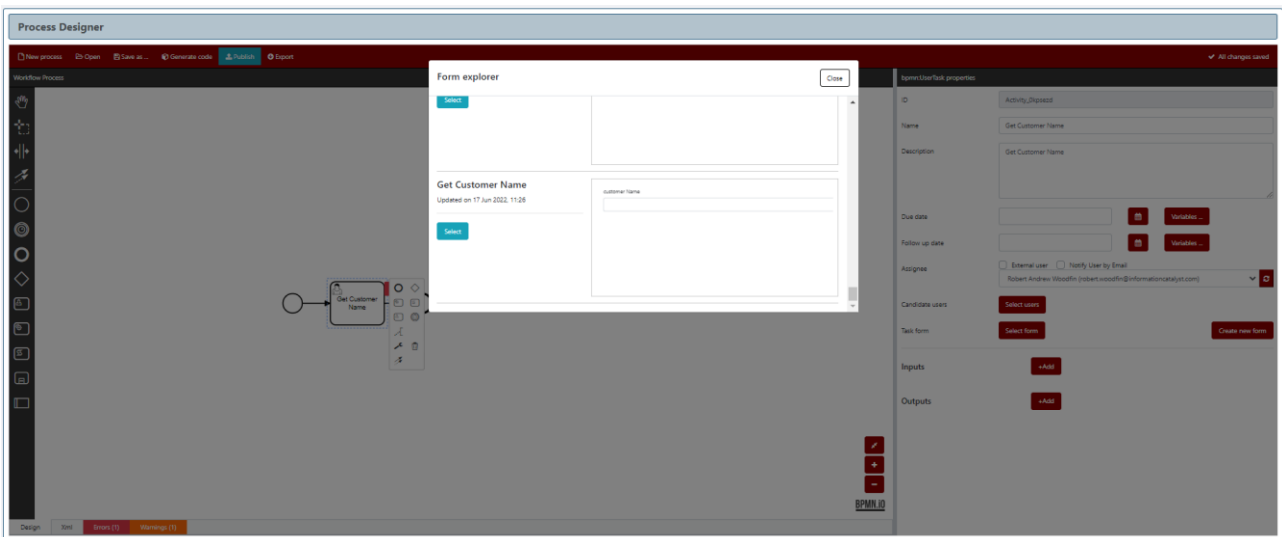


Figure 192. WASP: Assign the form to the User Task

The next step is to retrieve a list of messages for that customer. This requires access to an external API, so we use a Service Task to make a REST API call. When a Service Task is added, the User is presented with a 'Marketplace Browser' dialog that provides a list/catalog of services from which to choose.

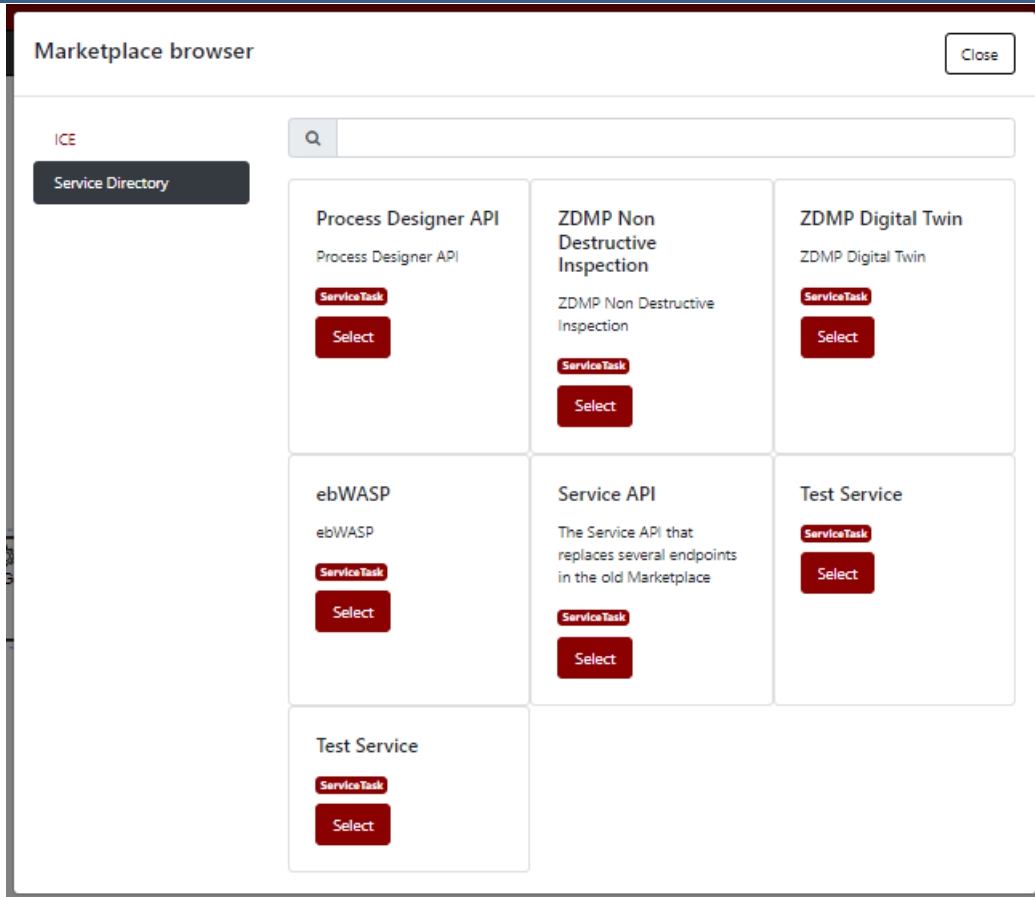


Figure 193. WASP: "Marketplace Browser" Dialogue to Populate Service Task

The 'Get Messages' REST API call requires a 'customerName' parameter, so we can select the process variable created from the initial User Task.

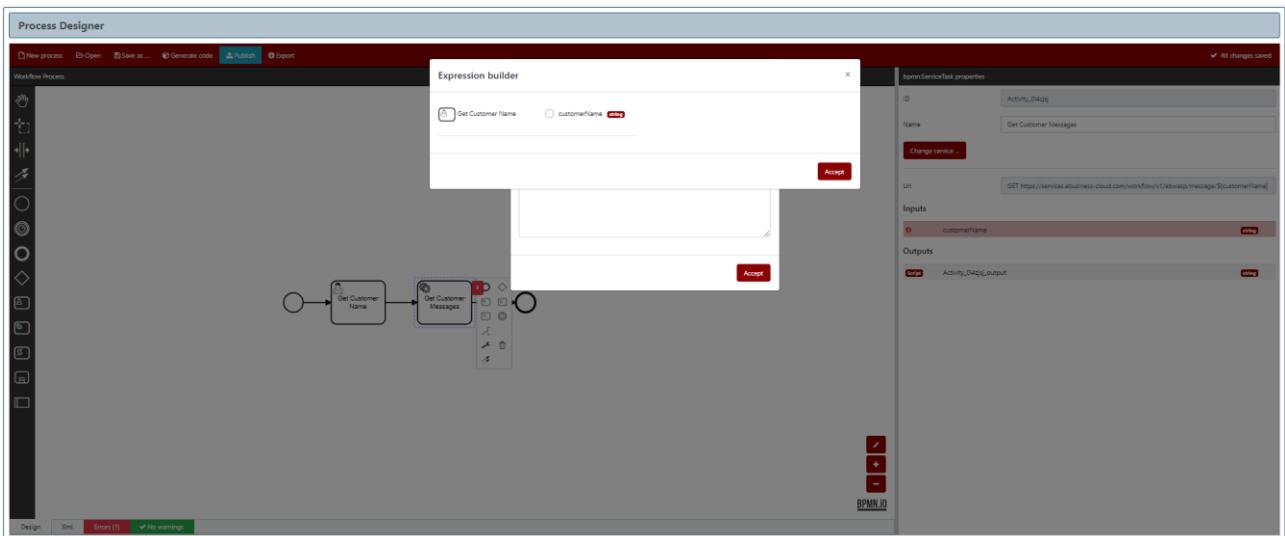


Figure 194. WASP: Assign a Process Variable to the Service Task Input Parameter

This completes the configuration of the Service Task. The last thing to do is display the response from the Service Task API call. We create another User Task called 'Show



Messages’ and add an input parameter called ‘messageList’. We can select a process variable to assign to this input parameter.

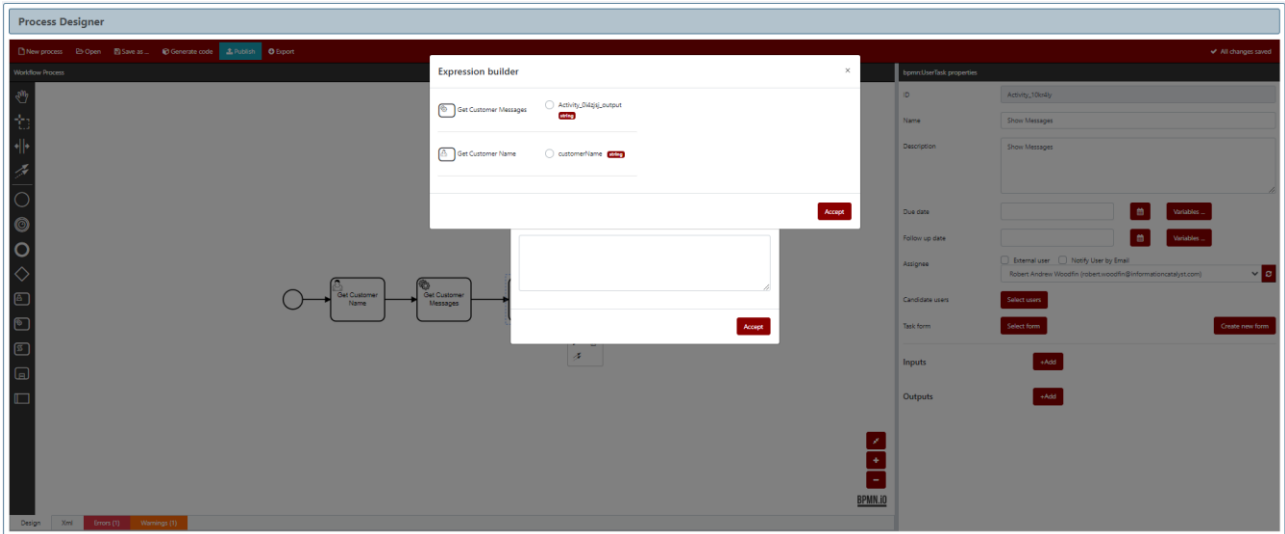


Figure 195. WASP: Assign a Process Variable to the User Task Input Parameter

This gives us the final simple process.



Figure 196. WASP: Simple Workflow Process

The next step is to publish the process to the Runtime Process Engine (Camunda).



Figure 197. WASP: Publish Menu Option

If successful, we will get a message popup.

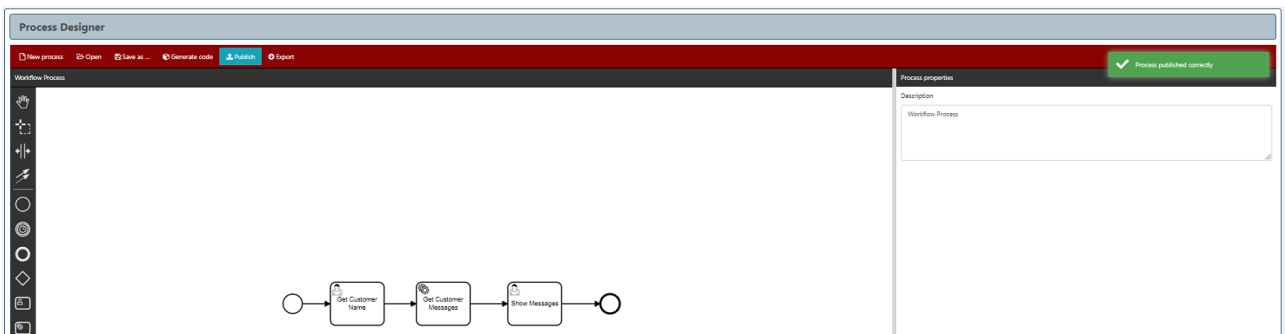


Figure 198. WASP: Successful Publish/Deployment of the Process

This is the end of the design stage. The publish/deployment of the process to the Process Engine will mean it is available to the User to start an instance of the process.

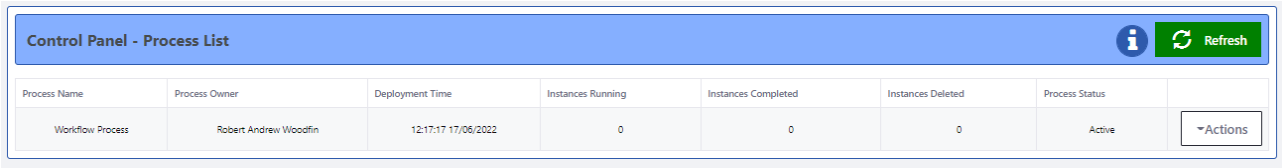


Figure 199. WASP: Control Panel Showing the Processes Owned by the Current User

The “Actions” button presents the User with a menu of options for the associated process.

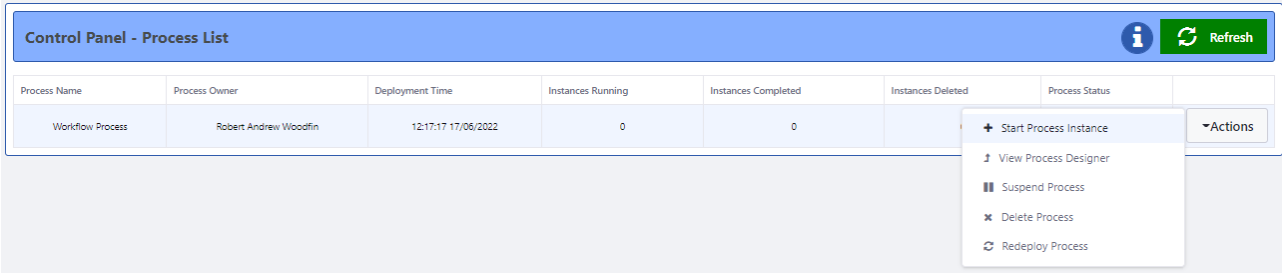


Figure 200. WASP: Menu Options Available for a Selected Process

Selecting “Start Instance” will start an instance of the process in the Process Engine. The Control Panel will be updated & show that there is now one running instance.

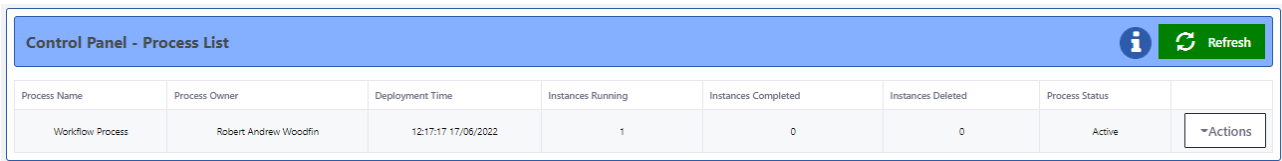


Figure 201: WASP. Updated Control Panel Showing that a New Instance is Now Running  
The User can view the current status of the running process by navigating to the ‘Instance View’ page.

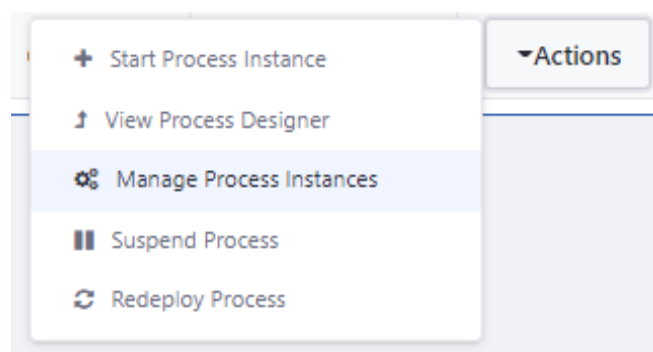


Figure 202. WASP: Navigate to the Instance List Page

This will take the User to the Instance List page.

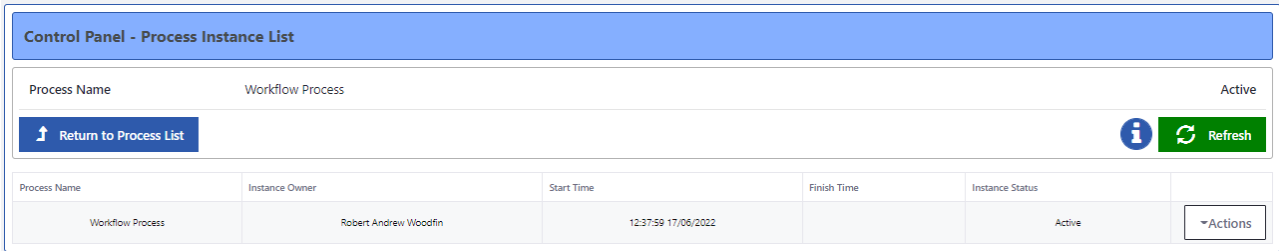


Figure 203. WASP: Instance List Page for the Selected Process

Then, select the 'View Process Instance' option from the Actions menu.

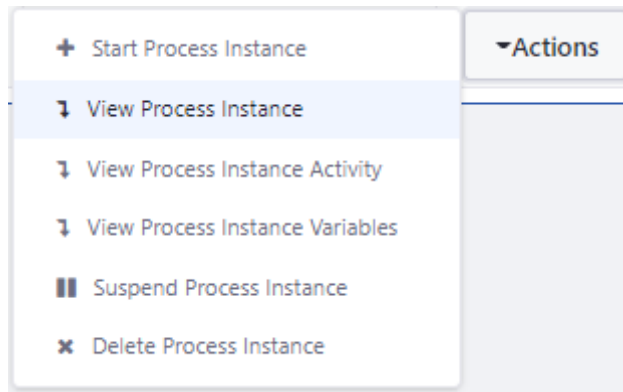


Figure 204. WASP: Navigate to the Instance View Page

The user will see the current status of the selected instance.

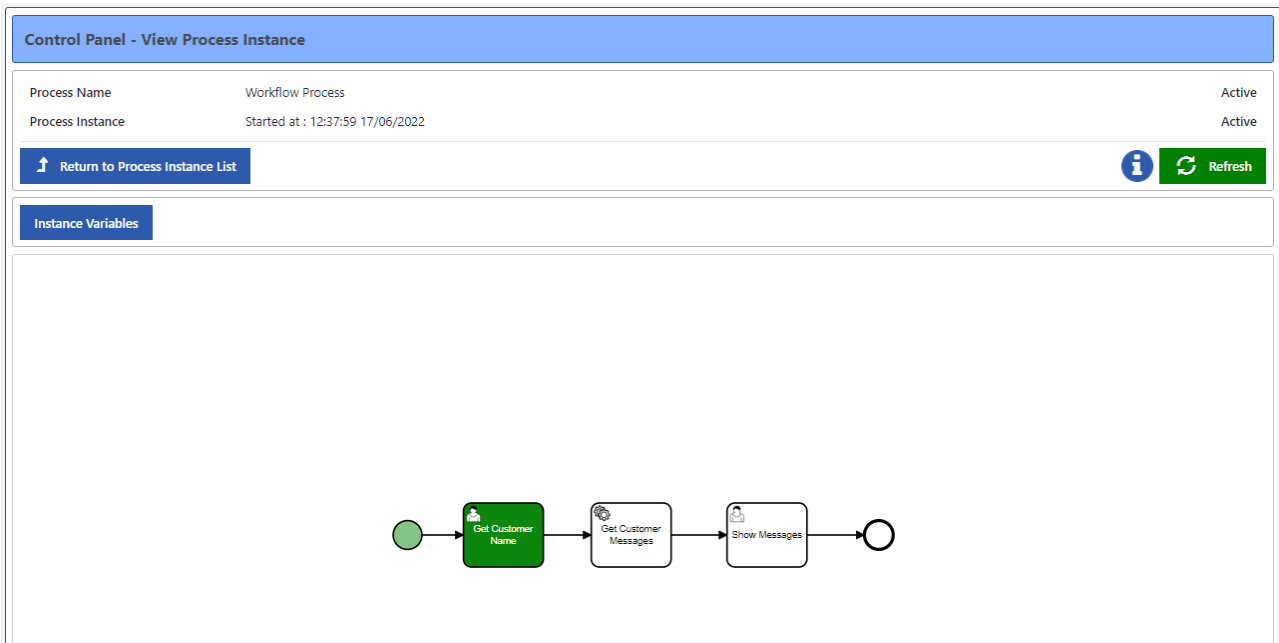


Figure 205. WASP: Instance View Page for the Selected Instance

As displayed, the process instance is in a waiting state. It is waiting for the User to complete the User Task 'Get Customer Name'. This is where the Tasks UI comes into play. The User navigates to the Tasks UI via the 'My Tasks' menu option in the primary WASP navigation menu.

The Task List displays all the pending tasks assigned to the current User.

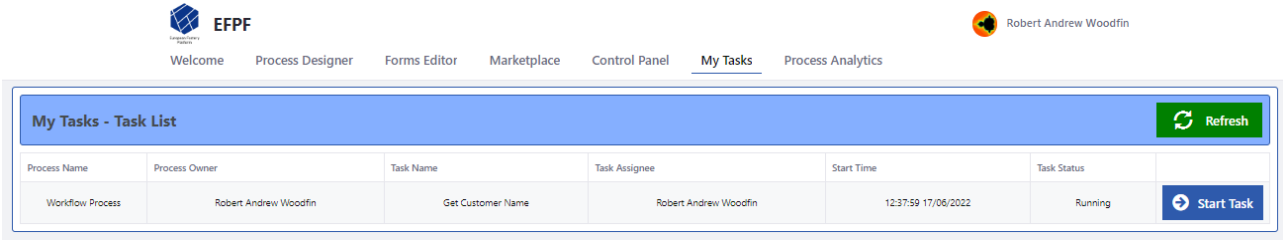


Figure 206. WASP: Task List Page Showing All the Pending Tasks Assigned to the Current User

Selecting the 'Start Task' option will navigate the User to the 'Task Form' form for the selected Task.

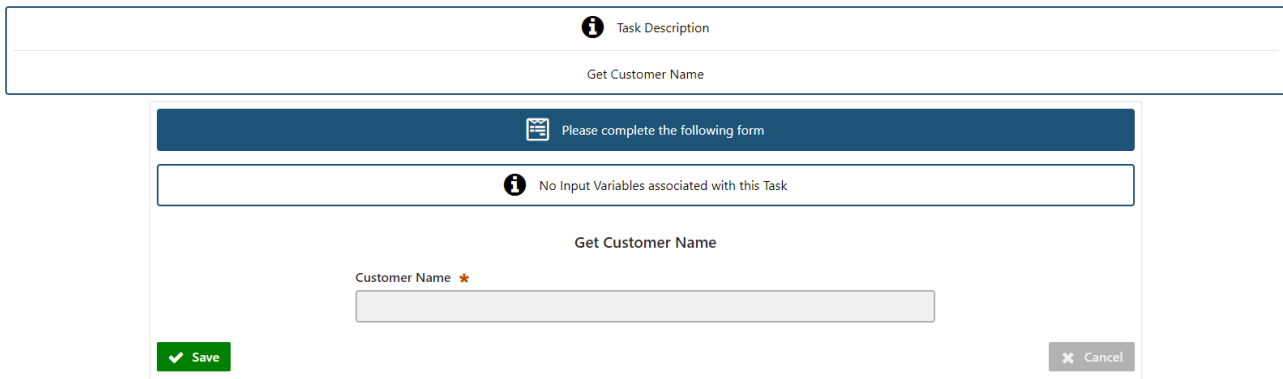


Figure 207. WASP: Task Form for the "Get Customer Name" User Task

When the User enters the value for the 'Customer Name' & presses <Save>, the Task is completed. The Process Engine receives the information for the completed task, and the process will continue to the next activity. This is the Service Task 'Get Messages'. The Process Engine will use the information entered in the 'Get Customer Name' User Task and execute the Service Task API call to retrieve the list of messages. The response from the Service Task is saved as a process variable in the Process Engine, and the process will move on to the next activity. We can see the updated status of the process via the Instance View page.

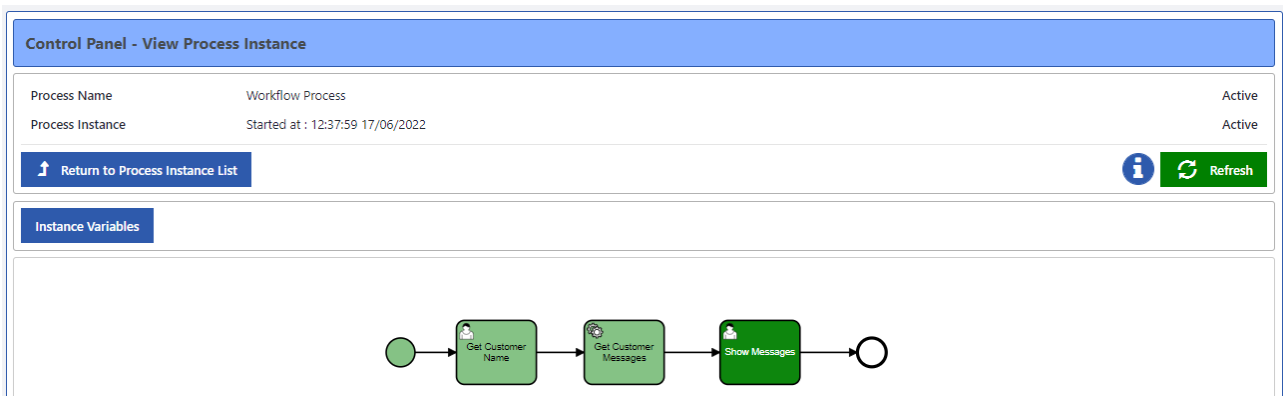


Figure 208. WASP: Updated Status of the Process

The User will be able to see the pending task in the Task List.

My Tasks - Task List <span style="float: right;">Refresh</span>						
Process Name	Process Owner	Task Name	Task Assignee	Start Time	Task Status	
Workflow Process	Robert Andrew Woodfin	Show Messages	Robert Andrew Woodfin	13:16:52 17/06/2022	Running	<span>Start Task</span>

Figure 209. WASP: Task List Showing the Pending Task for the Current User

Selecting the <Start Task> option will take the User to the Task Form associated with the selected task.

Task Description


---

Show Messages

Please complete the following form

Task Inputs

Activity\_014zjsj\_output

```
[
  {
    "date": "2022-06-12T19:36:16.18",
    "filename": "invoice-message-1.txt",
    "inDoctype": "INVOIC D96A",
    "messageType": "EDIFACT message",
    "receiver": "AWO",
    "sender": "ESS",
    "outDoctype": "UBL Invoice ANZ",
    "contentLength": 3245,
    "id": 1,
    "mimeType": "text/plain",
    "customer": "ESS",
    "status": "READY TO SEND"
  },
  {
    "date": "2022-06-12T19:36:16.191",
    "filename": "invoice-message-2.txt",
    "inDoctype": "INVOIC D96A",
    "messageType": "EDIFACT message",
  }
]
```

No Output Variables associated with this Task

Save
Cancel

Figure 210. WASP: Task Form for the "Show Messages" User Task

This task does not require any User input, it just displays the response from the previous Service Task API call. Pressing <Save> will complete the task & the Process Engine receives the information for the completed task and move on to the next activity, which in this case is the task completion.

The now has no pending tasks.

My Tasks - Task List <span style="float: right;">Refresh</span>						
<span style="float: right;">No Tasks Found for Robert Andrew Woodfin</span>						

Figure 211. WASP: Task List Showing there are No Pending Tasks for the Current User

The 'Instance View' page will be updated to show that the instance has been completed.

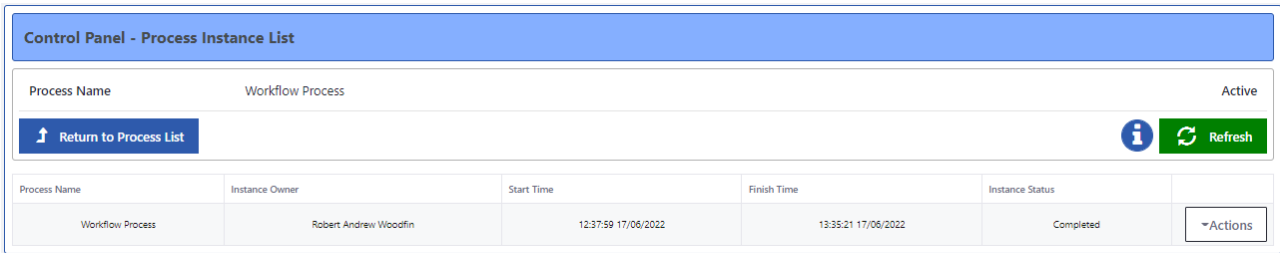


Figure 212. WASP: Instance List Showing the Updated Status of the Selected Instance  
The User can see the completed instance via the 'Instance View' page.

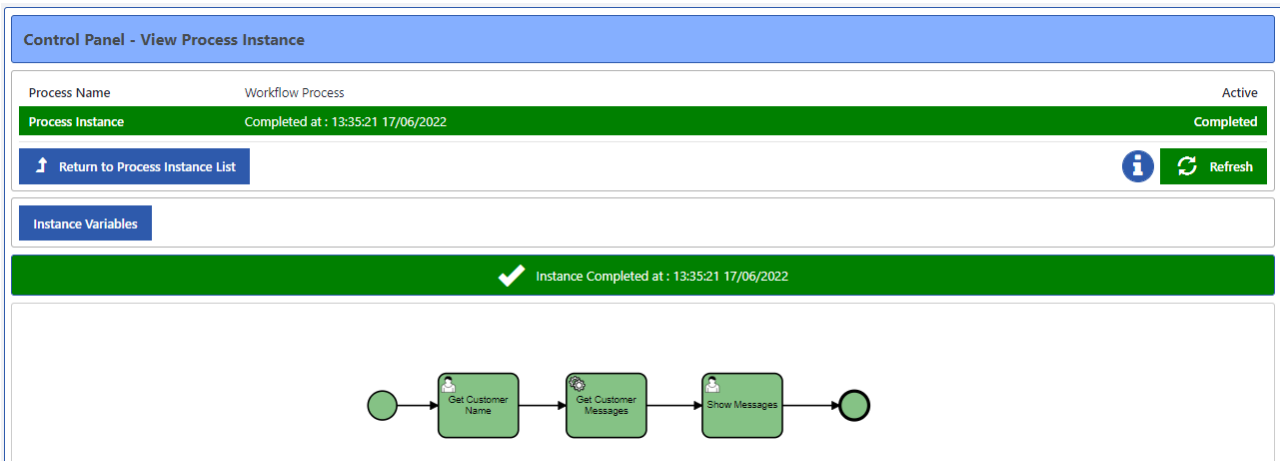


Figure 213. WASP: Instance View of the Completed Task

Finally, the 'Process List' page will be updated to show that there is now one completed instance.

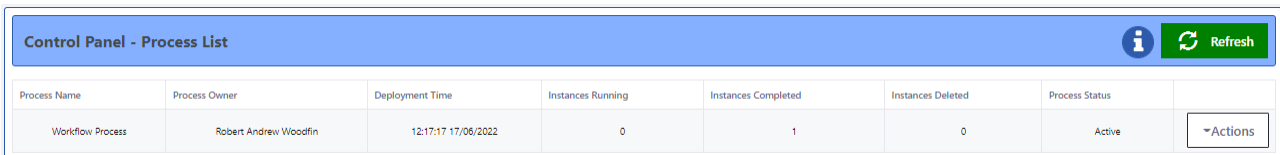


Figure 214. WASP: Process List Showing the Instance has now been Completed

Obviously, the process demonstrated here is very simple. Figure 215 is an example of a more complex process that has multiple User Tasks & multiple Service Task API calls.

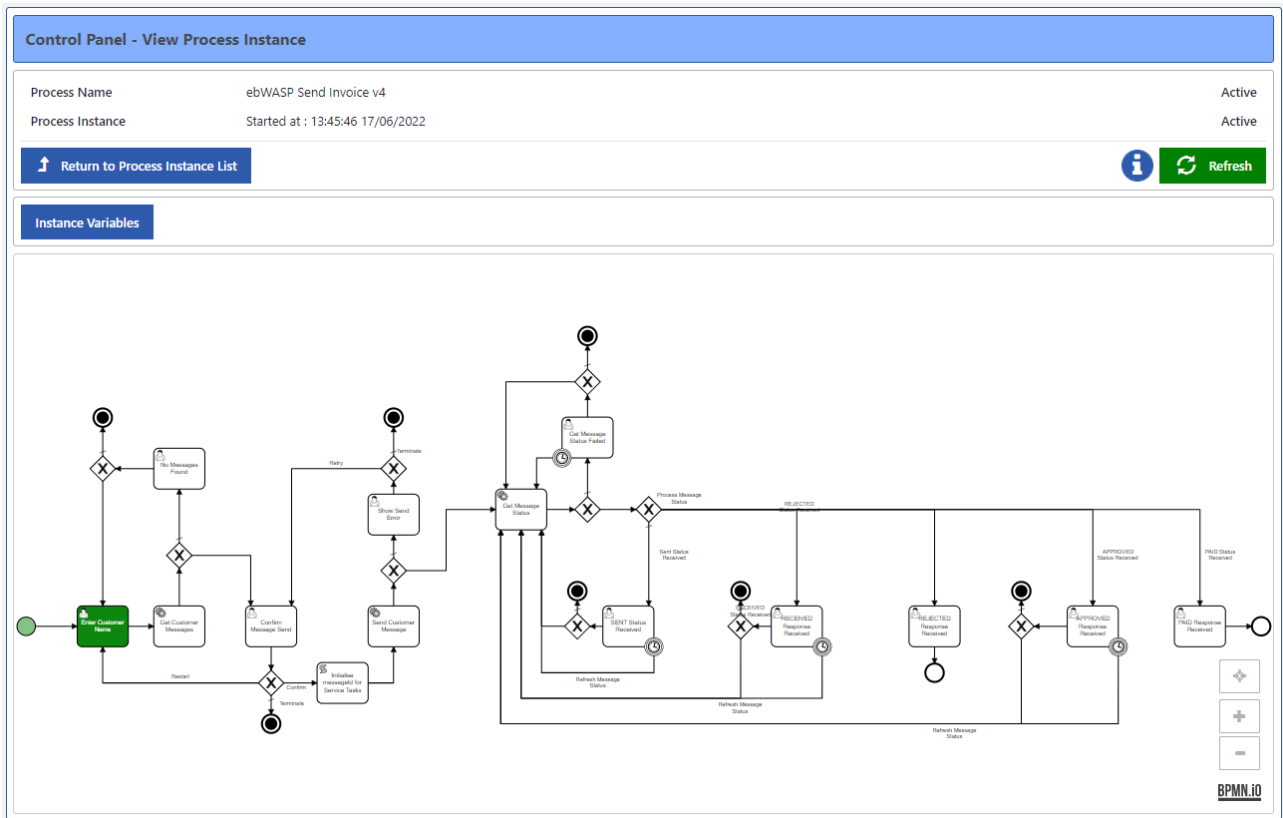


Figure 215. WASP: Instance View of a Complex Process

### The Marketplace

The WASP UI that has not been mentioned yet is the Marketplace, as shown in Figure 216. It is here where the User can add individual API endpoints or register an external marketplace that provides multiple API endpoints.

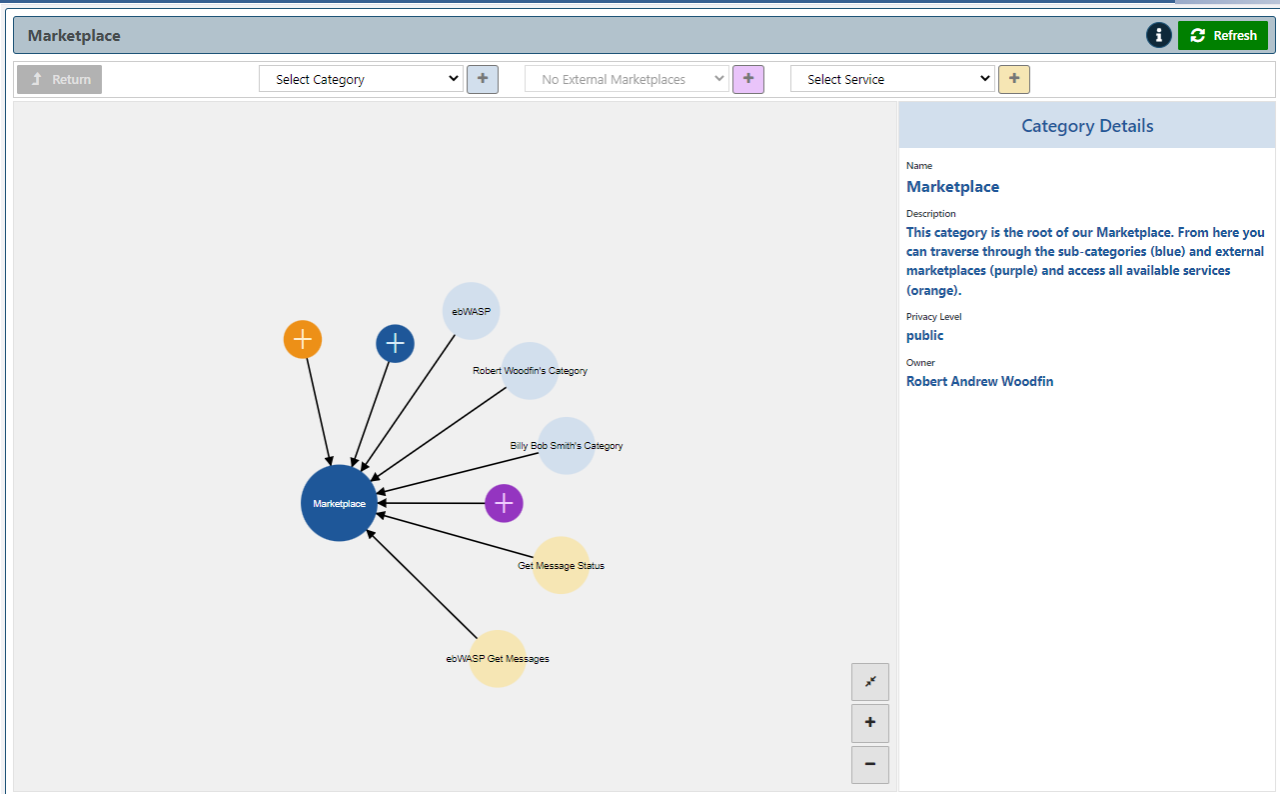


Figure 216. WASP: Marketplace View Showing the Categories/Services/External Marketplaces Available

The individual services/external marketplaces can be grouped into categories, based on functionality or ownership. Not all users can see/use all the elements in the Marketplace. For example, 'Robert Woodfin's Category' has a privacy level set to 'private' and will have services/external marketplaces owned & used specifically by that User.

Alternatively, the 'ebWASP' category is based on functionality. It has a single public external marketplace.



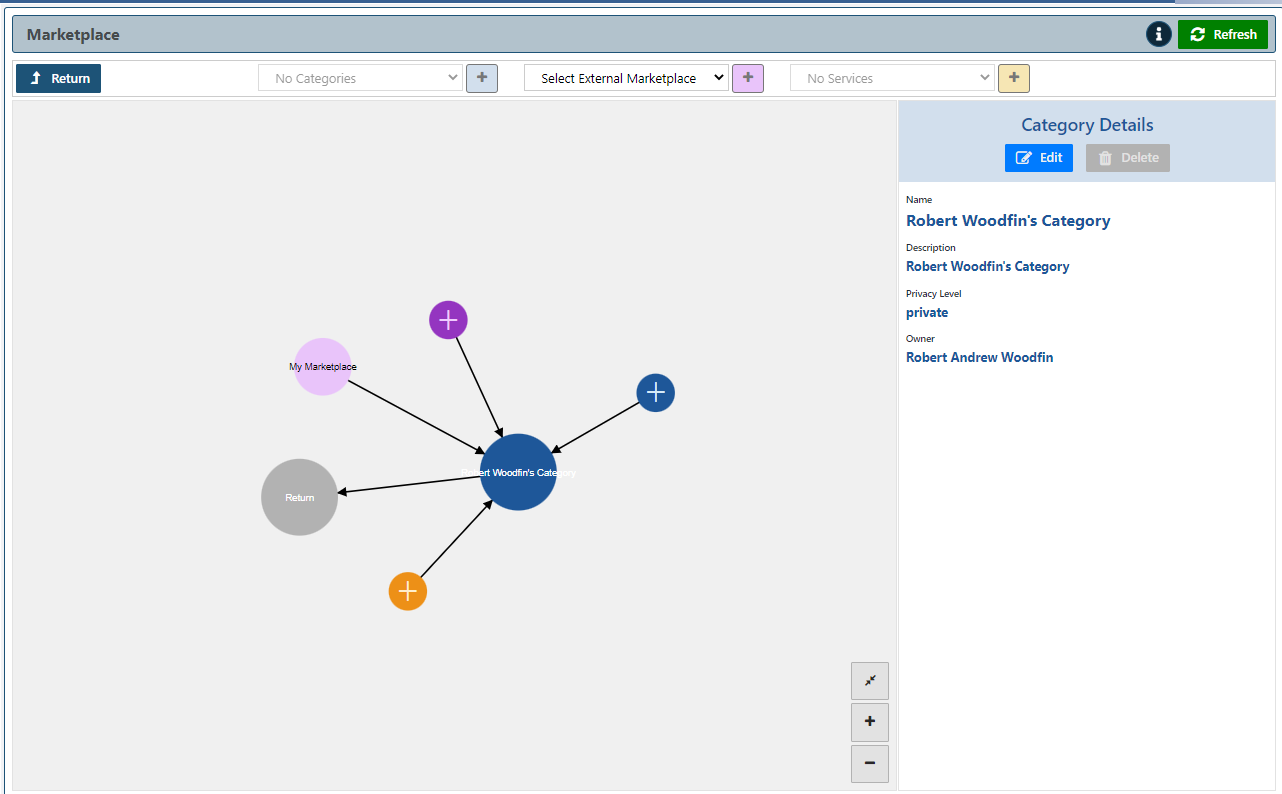


Figure 217. WASP: Private Category for User

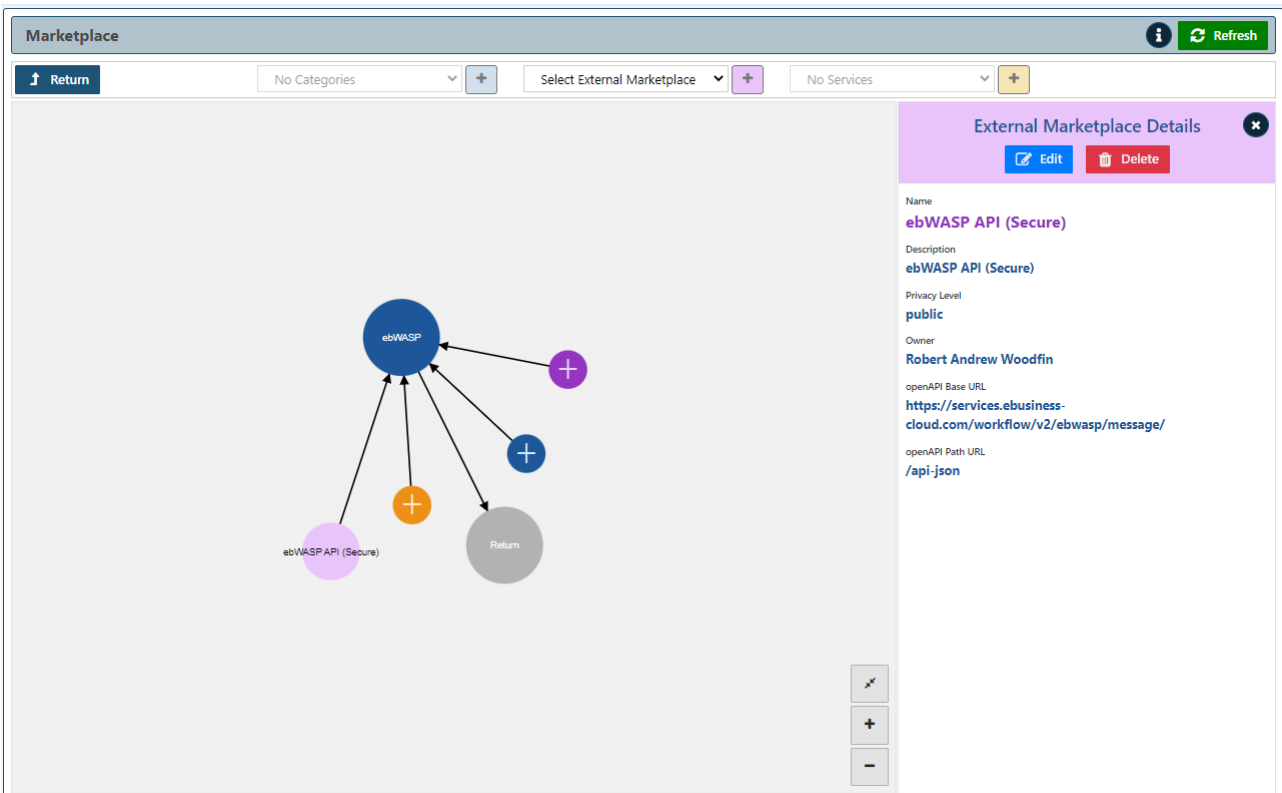


Figure 218. WASP: Public ebWASP Category with a Public External Marketplace

It is the external marketplaces that are the focus of the WASP Marketplace. It allows for the registration of external/3<sup>rd</sup> party APIs that are made available to the Process Designer and thereby providing access to a multitude of Cloud Services offering a wide range of functionality. These external marketplaces have a number of advantages:

- They must be OpenAPI compliant – This ensures they comply with an API industry standard.
- Those that have an added authentication layer (OAuth2 / Basic) can be configured via the Process Engine API Gateway.

Figure 219. WASP: Registration of an External Marketplace

Figure 220. WASP: Configuration of Authentication for an External Marketplace

The external marketplaces are integrated into the Process Designer, where the User can select a service provided by a 3<sup>rd</sup> party based on the information provided by the registered external marketplace.

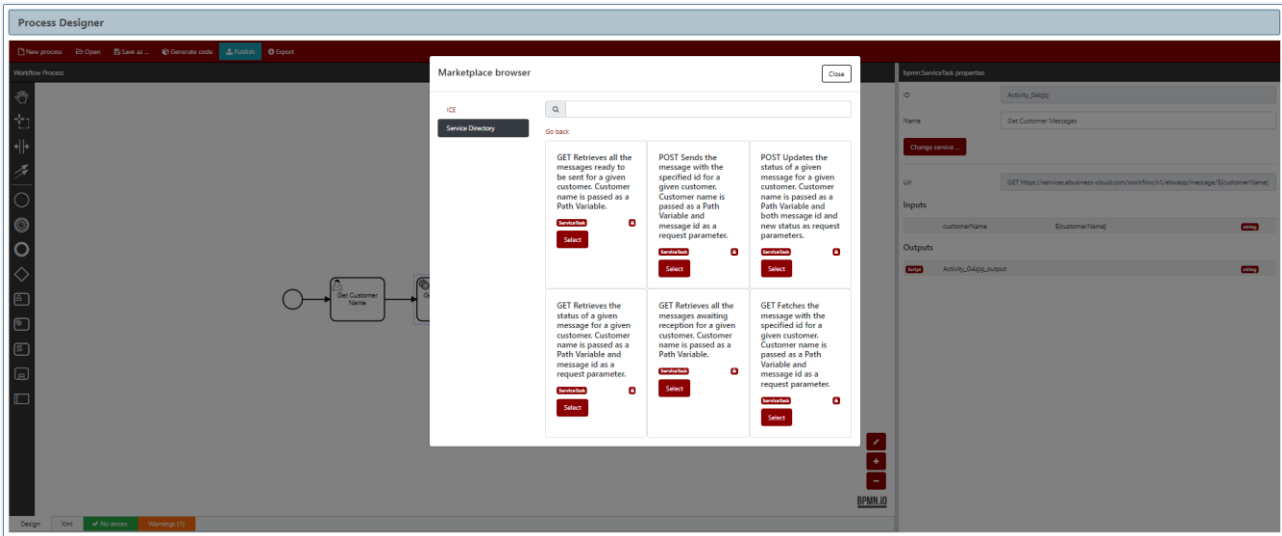


Figure 221. WASP: API Endpoints Made Available from an External Marketplace

### 3.2.1.15 EFPF SDK Studio

The EFPF SDK Studio is a web-based Integrated Development Environment (IDE) for supporting the apps development using EFPF. It was built on top of the best-of-breed Eclipse CHE project, and it allows the development of applications in a visual fashion, promoting the integration of the other SDK components and featuring services such as multiple development languages, syntax highlighting, compiling, running, and deploying the code and so on. The SDK Studio authenticates EFPF users with the security portal, integrating into the EFPF SSO environment. Some screenshots of the web development interface can be seen in Figure 222.

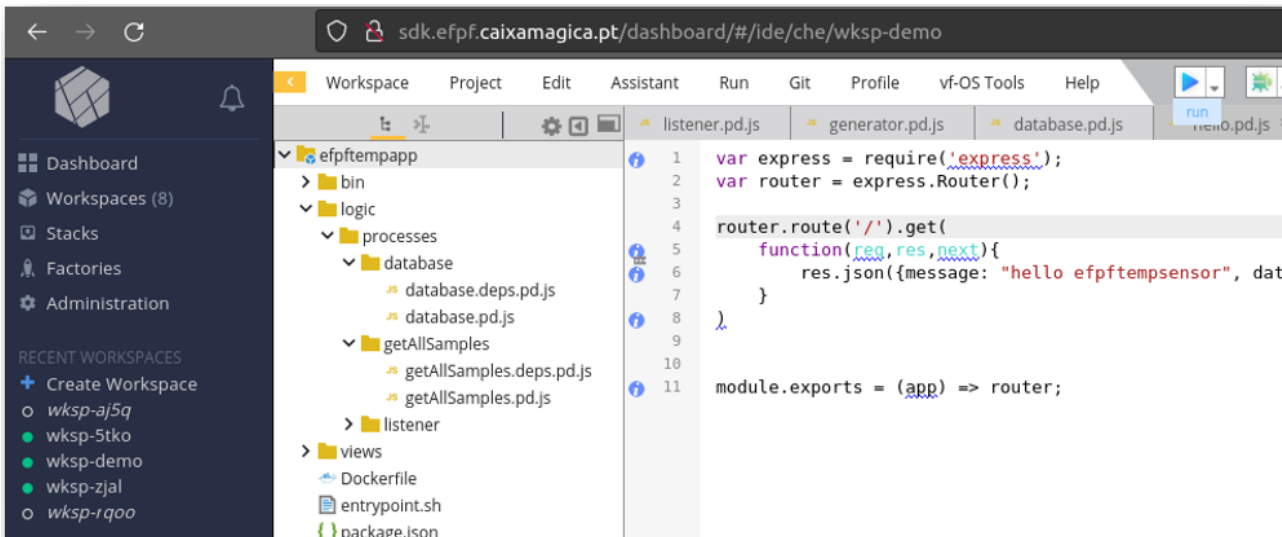


Figure 222. EFPF SDK Studio Environment

The SDK Studio has other SDK automatic integrations with other EFPF services such as the Marketplace (see Section 3.2.1.2) as shown in Figure 223, the EFPF Pub/Sub Security

Service (see Section 3.2.1.4), SDSS (see Section 3.2.1.6) or the WASP Process Designer (see Section 3.2.1.14).

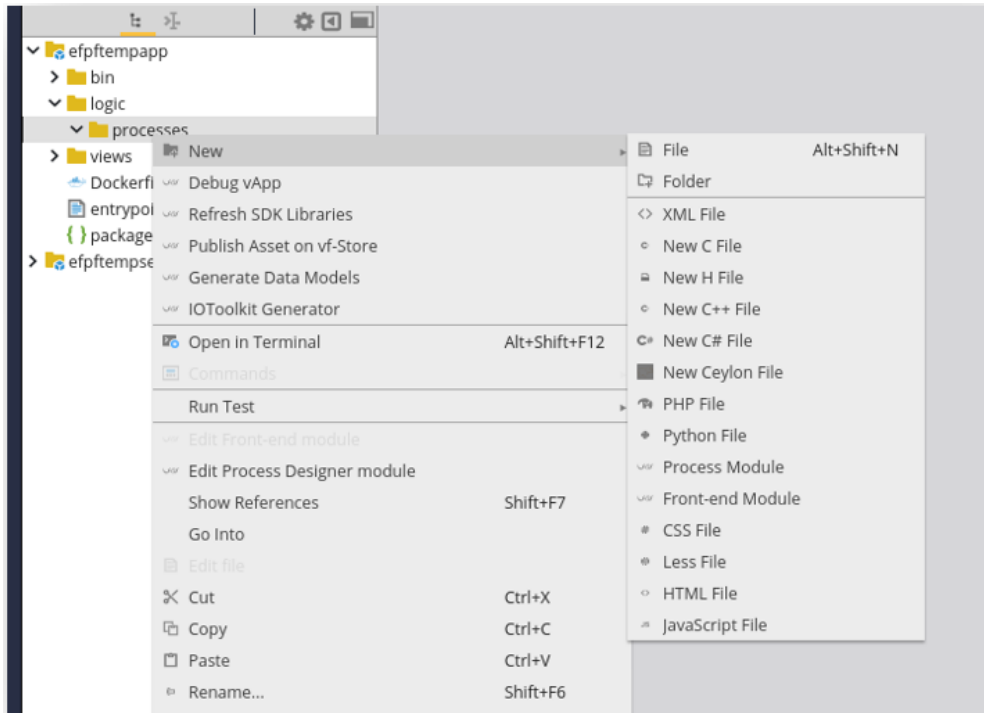


Figure 223. EFPP SDK Studio Integrations

The toolkit also includes a Frontend editor that allows the inclusion of GUI and other visual elements in the applications being developed, with the GUI actions then being connected with process steps or with other application actions. The Frontend has a NodeJS backend associated with the set of actions being developed, as seen in Figure 224.

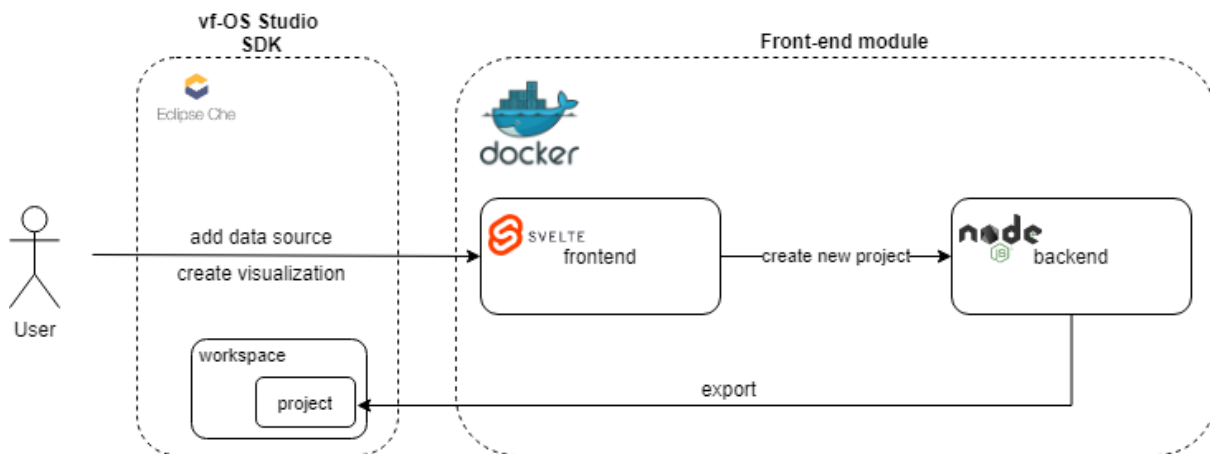


Figure 224. EFPP SDK Frontend Editor Architecture

The Frontend editor allows the creation of screens with GUI components such as buttons, labels, text boxes or tables, as seen in Figure 225. These components can then be used to create the application’s User Interface or to publish reports and graph charts to visualize project data being observed.

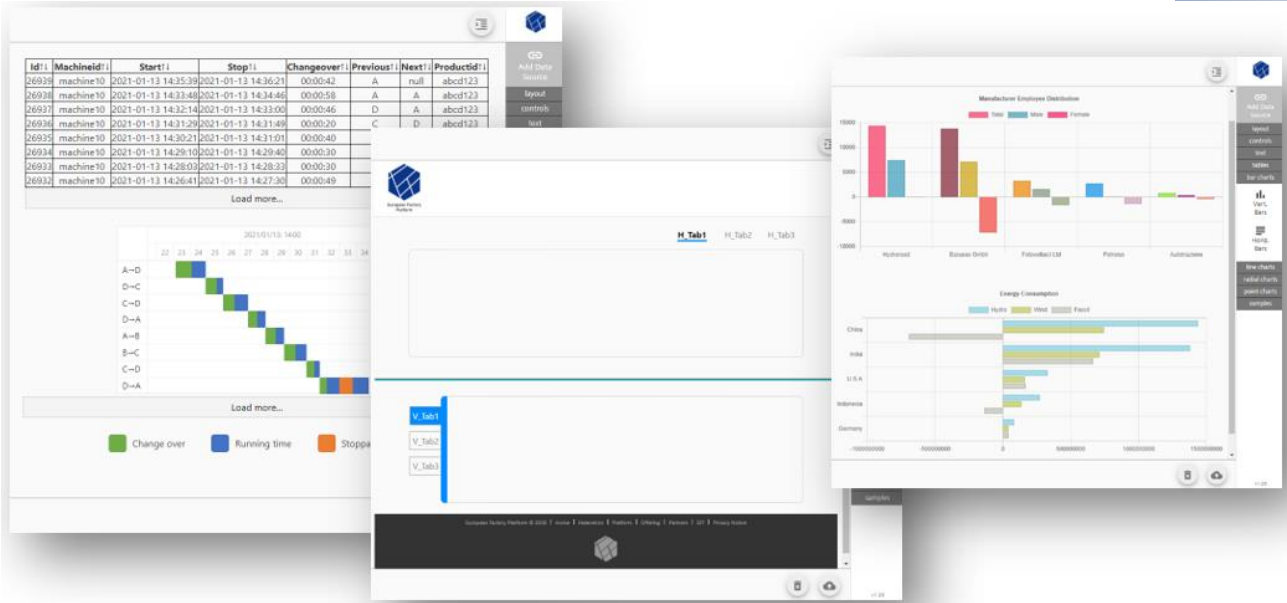


Figure 225. SDK Frontend Editor

Finally, the EFPF SDK is complete with the inclusion of the EFPF Engagement Hub, a web portal that is targeted to engage EFPF apps developers into sharing and promoting their development with the community, as seen in Figure 226.

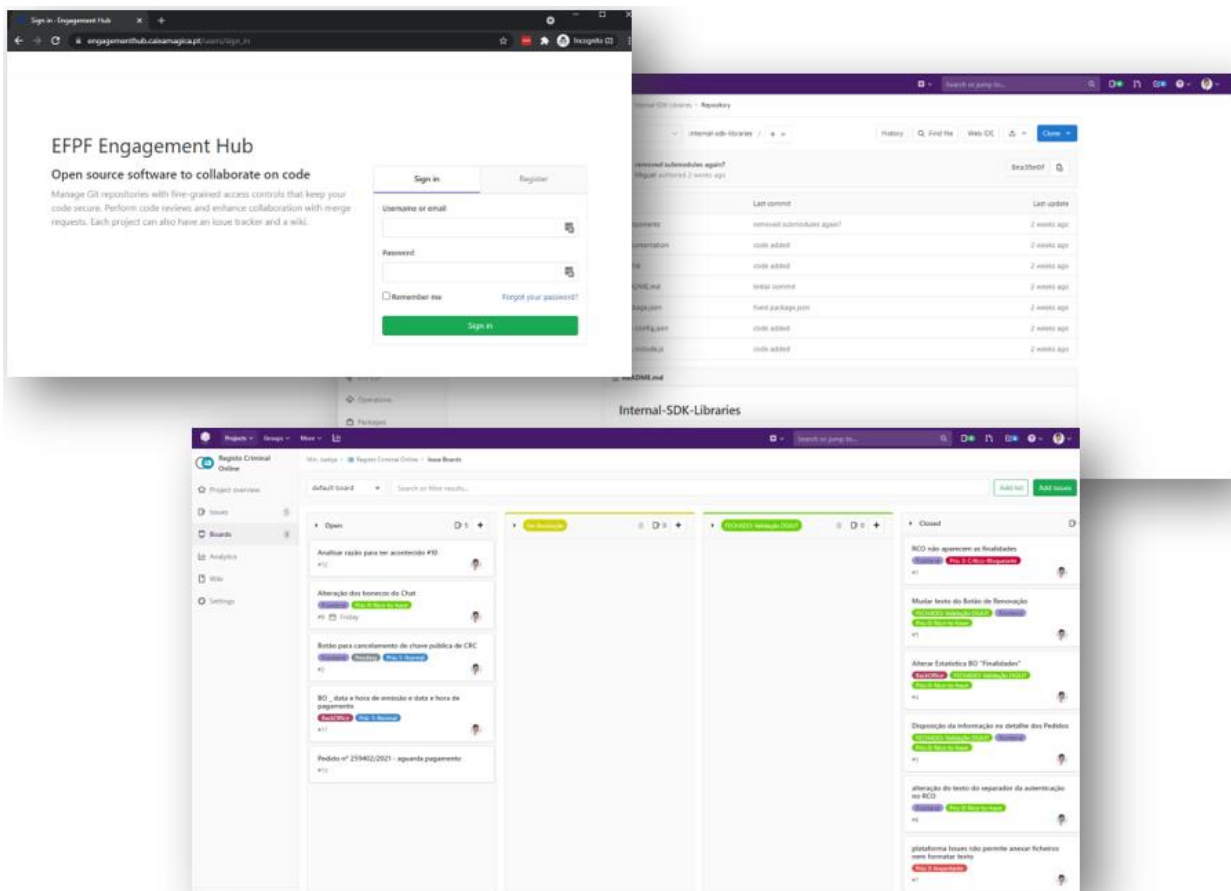


Figure 226. EFPF SDK Engagement Hub

This portal allows the hosting of source code of the apps, together with a set of other collaboration tools targeted to foster the interaction between developers and the community.

### 3.2.1.16 Risk, Opportunity, Analysis and Monitoring (ROAM) Tool

The ROAM Tool enables users to process runtime data coming in through the Data Spine Message Bus. The input data arrives on a user-defined input topic and then goes through a workflow consisting of different recipes. The output is then sent back through the Message Bus on a user-defined output topic and, if applicable, presented as an email or webpush notification. In case there is an error, that error is published to a user-defined error topic. The constituting recipes are provided by the tool and are configurable, such that the user can fill in its parameters to tailor it to their use case. The component diagram for the ROAM Tool is depicted in Figure 227. The ROAM Tool provides a REST API, that can also be interacted with through its frontend UI. This API allows for CRUD operations for workflows and configured recipes, subscribing and unsubscribing to topics in the internal MQTT client, registering webpush keys, managing ROAM Tool topics via the pub/sub security service, and testing of workflows. The ROAM Tool uses a local MongoDB database for persistent storage. Lastly, the ROAM Tool API is secured using Keycloak, where users need to supply the Access-Token header to use the API and need to login to use the frontend.

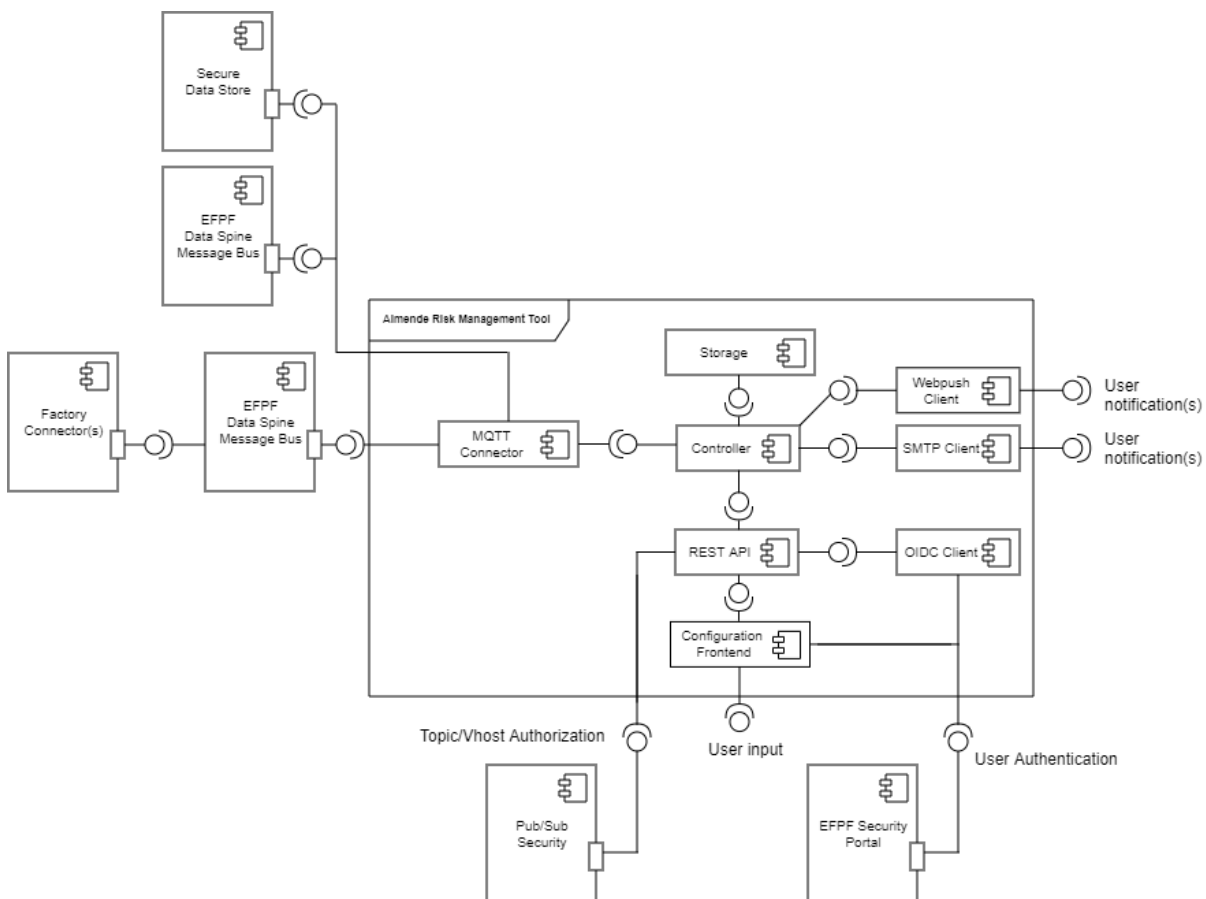


Figure 227. Architecture of the ROAM Tool

#### Roam Tool's HTTP REST API:

Figure 228 provides an overview of the REST API of the ROAM Tool, which offers configuration of workflows and recipes, and lets the user manage the MQTT functionality of their recipes.

REST Endpoint	HTTP Method	Description
/workflows/	GET	Retrieves all available workflows
/workflows/	POST	Creates new workflow with a random id
/workflows/{id}	GET	Retrieves a specific workflow by its id
/workflows/{id}	PATCH	Updates an existing workflow
/workflows/{id}	DELETE	Deletes a workflow
/workflows/{id}/subscription	PATCH	Updates a workflows subscription status, such that the MQTT client subscribes or unsubscribes to its corresponding input topic.
/workflows/{id}/run	POST	Runs a workflow with provided input data for testing purposes. If the provided id is "test", the user can provide a workflow as well.
/workflows/{id}/clear_cache	PATCH	Clears the cache of the constituting workflow recipes, in case they have caching functionality. Otherwise, this does nothing.
/workflows/push/	GET	Retrieves all workflows with some subscription key. Requires the 'Subscription-Endpoint' header.
/workflows/{id}/push	PUT	Add/removes a workflow push subscription key
/recipes/	GET	Retrieves all available recipes
/recipes/	POST	Creates new recipe with a random id
/recipes/{id}	GET	Retrieves a specific recipe by its id
/recipes/{id}	PATCH	Updates an existing recipe
/recipes/{id}	DELETE	Deletes a recipe
/recipes/schemas	GET	Retrieves all recipe template schemas
/recipes/hardcoded/	GET	Retrieves all recipe templates with default values
/pubsub/	POST	Checks pub/sub access to a certain topic for the user and for the ROAM Tool

Figure 228. ROAM Tool HTTP REST API

The attributes of workflows and recipes are described below, and can be seen in Figure 229:

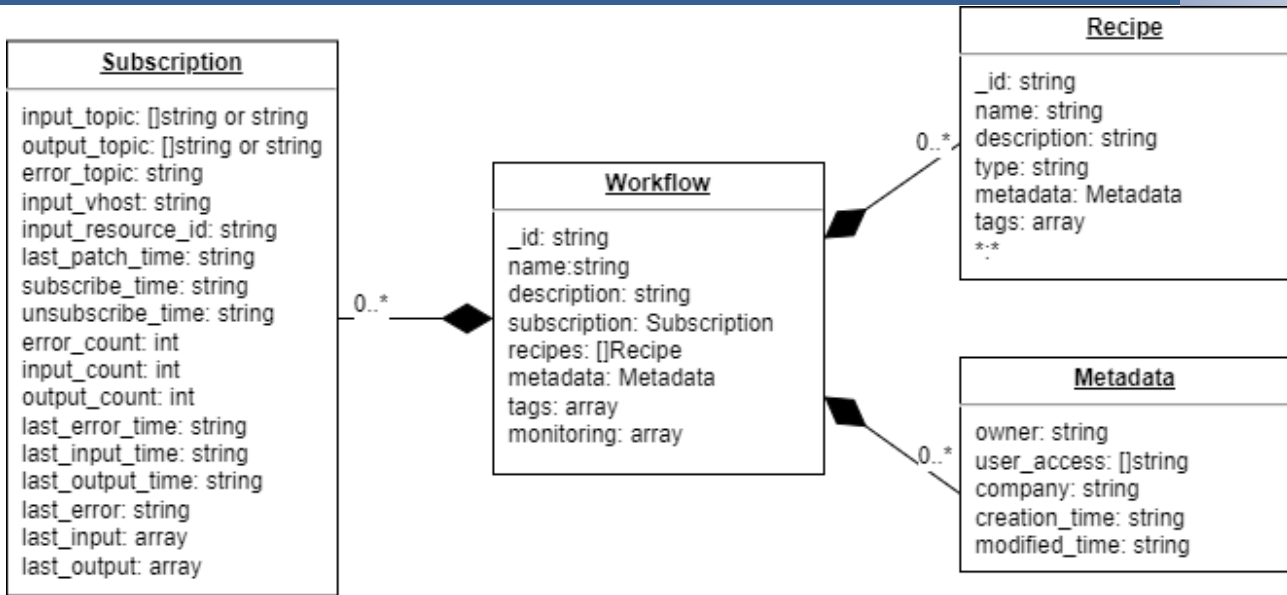


Figure 229. ROAM Data Model Diagram

A workflow consists of:

- `_id`: unique id of the workflow
- `name`: a name for the workflow
- `description`: a description of the workflow
- `subscription`: subscription configuration for the workflow
- `recipes`: constituting recipes of the workflow
- `metadata`: metadata, such as owner, creation data, etc., for the workflow
- `tags`: tags for the workflow, meant for searching and filtering in the frontend
- `monitoring`: object originally used for figure generation, that is now shelved

A recipe object consists of:

- `_id`: unique id of the recipe
- `name`: a name for the recipe
- `description`: a description of the recipe
- `type`: the name of the recipe template
- `metadata`: metadata, such as owner, creation data, etc., for the recipe
- `tags`: tags for the recipe, meant for searching and filtering in the frontend
- `*`: remaining required or optional recipe parameters

A subscription consists of:

- `running`: boolean denoting whether the MQTT client is subscribed to the input topic of the corresponding workflow, and whether it will then also do something with incoming data (in case the input topic is used by multiple workflows).



- `input_topic`: input topic(s) (without vhost)
- `output_topic`: output topic(s) (without vhost)
- `error_topic`: error topic (without vhost)
- `input_vhost`: vhost for input topic
- `input_resource_id`: id of the resource where the input topic was created
- `last_patch_time`: date on which the last subscription change was performed
- `subscribe_time`: last time on which the MQTT client subscribed to the input topic
- `unsubscribe_time`: last time on which the MQTT client unsubscribed to the input topic
- `error_count`: amount of errors occurred
- `last_error_time`: last time on which an error occurred
- `last_error`: string containing the last error
- `input_count`: amount of inputs received by the corresponding workflow
- `last_input`: last received input message; only available with the `include_last_input=true` query parameter
- `last_input_time`: last time on which a message was received
- `output_count`: amount of outputs produced by the corresponding workflow
- `last_output`: the last output the workflow produced; only available with the `include_last_output=true` query parameter
- `last_output_time`: the time on which the last output was produced

Metadata consists of:

- `owner`: owner of the recipe/workflow
- `user_access`: users that have shared access to the recipe/workflow
- `company`: company of the owner of the recipe/workflow
- `creation_time`: the time on which the recipe/workflow was created
- `modified_time`: the time on which the recipe/workflow was last modified

For recipes and workflows, only the `user_access` property of the metadata can be set and changed. When nested in a workflow, a recipe does not need a metadata property. The `_id` value is always generated by the ROAM Tool.

For subscriptions, only the `input_topic`, `output_topic`, `error_topic`, `input_vhost`, and `input_resource_id` properties can be set, and the `running` property can only be changed after it has already been created.

## ROAM Tool's Frontend UI:

To make recipe and workflow configuration a little bit easier, the ROAM Tool has a frontend UI as shown in Figure 230 - Figure 233:

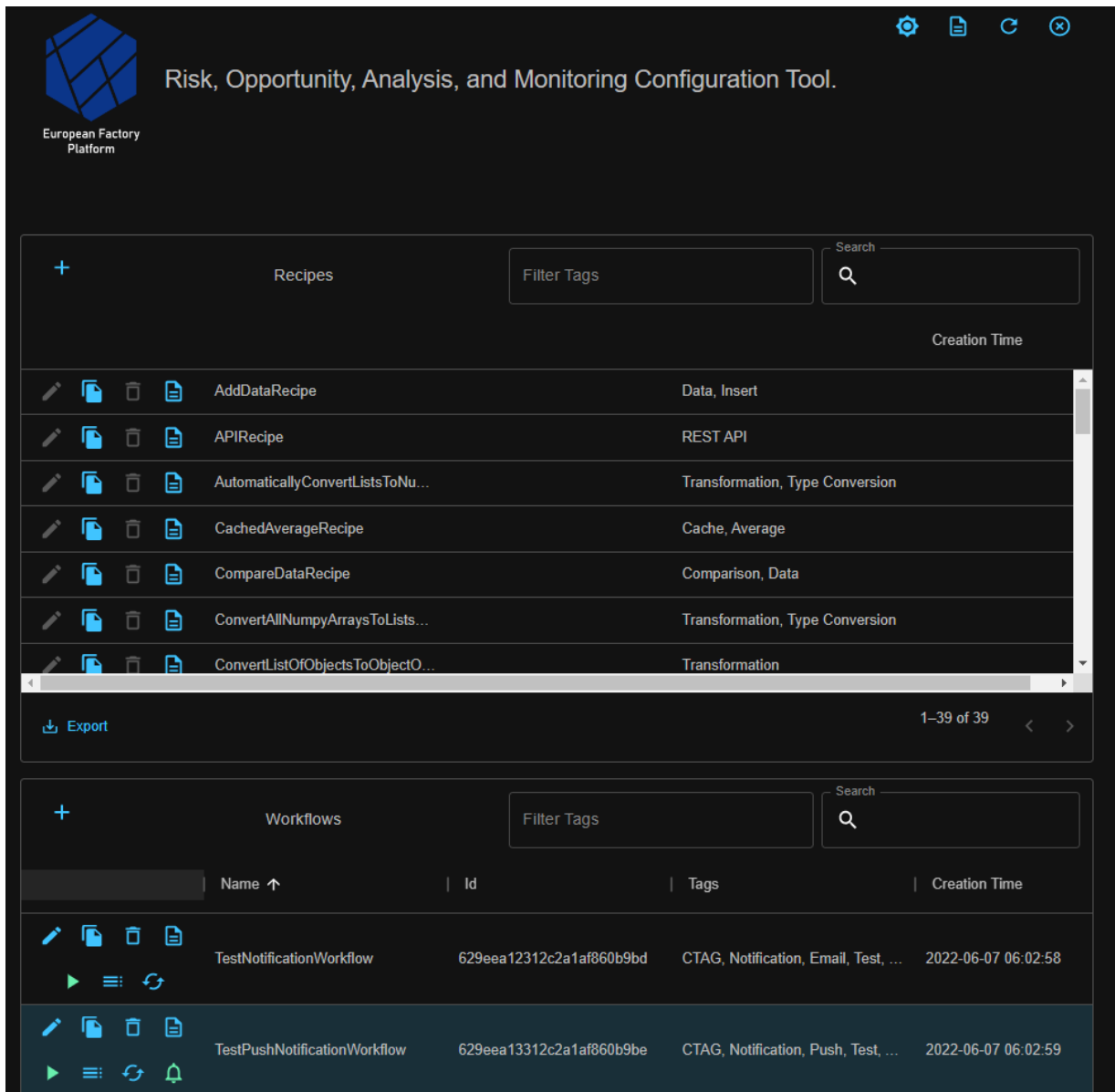


Figure 230. ROAM Tool's Frontend UI: Landing Page

The top panel shows all configurable and configured recipes (no configured recipes present here), and the bottom panel shows all created workflows. There are always “edit”, “copy-and-edit”, “delete”, and “details” buttons. Likewise, there is always a search and filter bar.

For recipe templates, the edit and delete buttons are greyed, and for workflows, there are additional buttons that let the user (un)subscribe the MQTT client to the input topic of the workflow and to show the last workflow output. When applicable, there are also buttons that clear the workflow recipe cache, and that enable/disable incoming push notifications.

When creating workflows, users need to supply all values for the MQTT configuration, the constituting recipes, and a name, description, and tags. Users can also test their workflows and recipes in the creation window.

When a workflow is created, users can check on the workflow details and monitor the incoming and outgoing data through the details button and see if the workflow is running as expected through the subscription details. Users can also read other properties of the workflow, such as the recipe properties and metadata. This functionality also extends to configured recipes themselves.

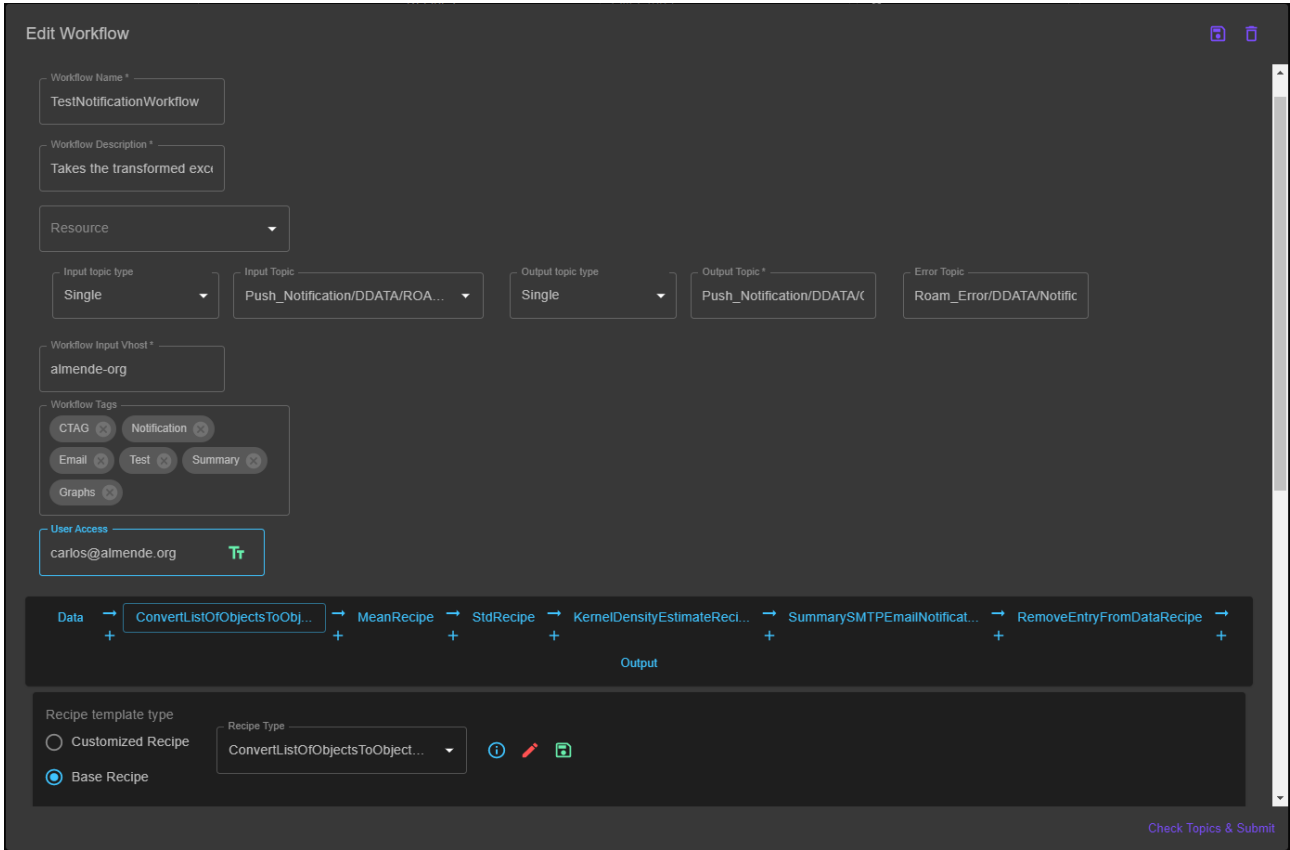


Figure 231. ROAM Tool's Frontend UI: Workflow config dialog

The screenshot displays the workflow details for 'TestNotificationWorkflow' (ID: 629eea12312c2a1af860b9bd). The interface is dark-themed and includes a description, subscription details, and a list of recipes.

These are the workflow properties of TestNotificationWorkflow with id 629eea12312c2a1af860b9bd

Description: Takes the transformed excel data and yields a daily summary notification email with figures.

**Subscription Details**

- subscribe\_time: 2022-06-07 06:03:23
- in\_topic: Push\_Notification/DDATA/ROAM/Test\_Input
- error\_count: 0
- in\_vhost: almende-org
- last\_error: None
- unsubscribe\_time: 2022-06-10 00:44:25
- last\_message: None
- error\_time: None
- last\_patch: 2022-06-10 00:44:25
- out\_topic: Push\_Notification/DDATA/CTAG/Excel\_Output
- message\_count: 0
- error\_topic: Roam\_Error/DDATA/Notification/
- output\_count: 0
- in\_resource\_id: 6254bcfcdd933d00273cb439
- last\_message\_time: None
- running: false

**Recipes**

- Recipe 1: Modified recipe of type ConvertListOfObjectsToObjectOfListsRecipe
- Recipe 2: Modified recipe of type MeanRecipe
- Recipe 3: Modified recipe of type StdRecipe
- Recipe 4: Modified recipe of type KernelDensityEstimateRecipe
- Recipe 5: Modified recipe of type SummarySMTPEmailNotificationRecipe
- Recipe 6: Modified recipe of type RemoveEntryFromDataRecipe

Close

Figure 232. ROAM Tool's Frontend UI: Workflow details

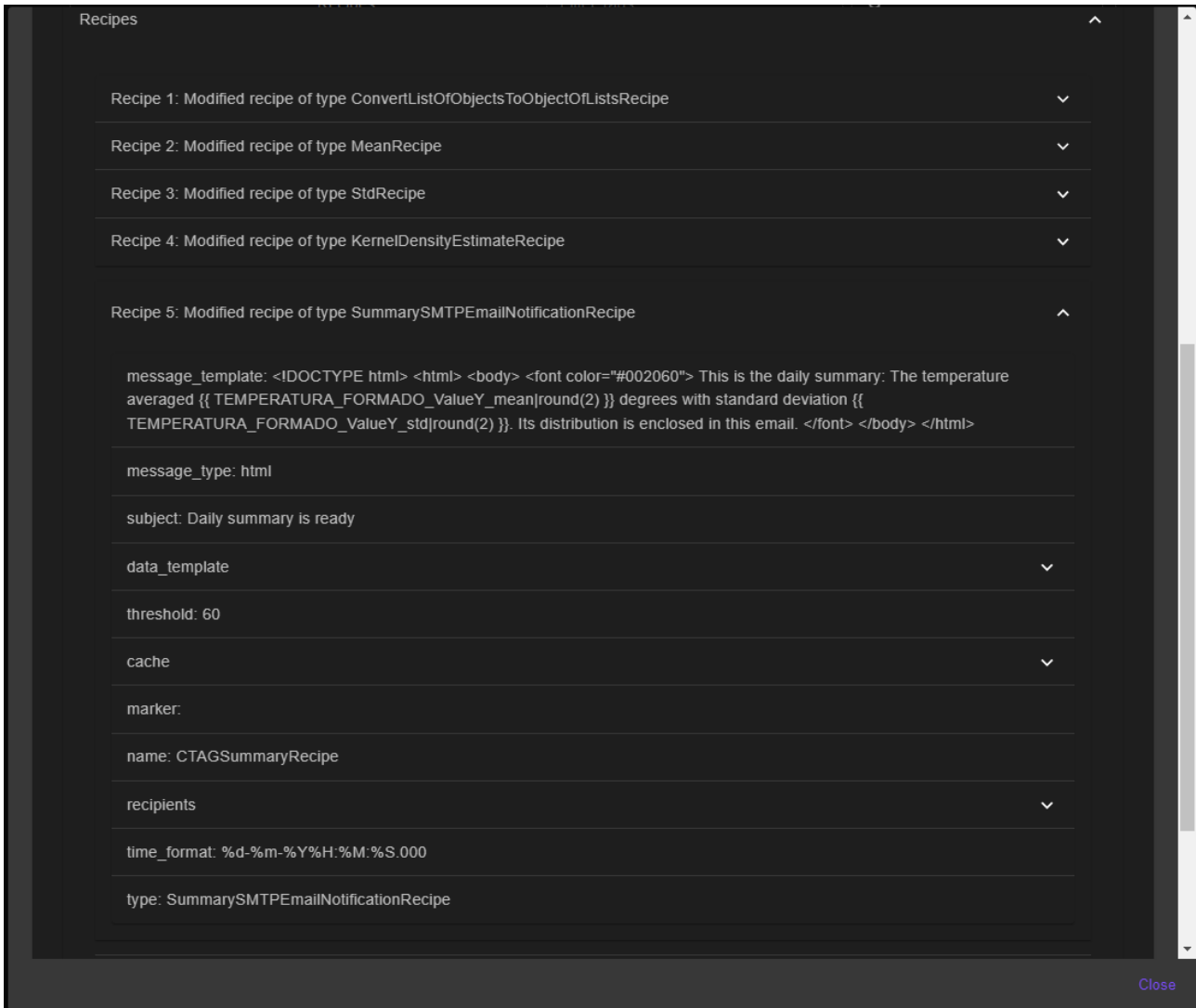


Figure 233. ROAM Tool's Frontend UI: Workflow recipe details

### 3.2.1.17 System Security Modeler (SSM)

The SSM is offered as a service and exposes two interfaces: a graphical user interface and an API interface. It allows a user to create a high-level model of your system, to detect potential security threats, to analyse the level of risk from each threat, and to find out and test what security controls could be used to reduce the level of risk in your system. The modelling process has three main stages that may be repeated several times and that can be executed through both the interfaces.

In the following paragraph the REST Interface and the interaction via GUI is shown. All these information is embedded in the tutorial that comes with all the SSM instances.

#### REST Commands:

model-controller

**PUT**/models/{modelid}

**DELETE**/models/{modelid}

**GET**/models

[POST](#)/models  
[POST](#)/models/{objid}/checkout  
[POST](#)/models/{objid}/checkin  
[POST](#)/models/{modelId}/copyModel  
[POST](#)/models/import  
[GET](#)/usermodels/{userId}  
[GET](#)/models/{objid}/palette  
[GET](#)/models/{modelWriteId}/validated  
[GET](#)/models/{modelWriteId}/calc\_risks  
[GET](#)/models/{modelWriteId}/calc\_risks\_blocking  
[GET](#)/models/{modelId}  
[GET](#)/models/{modelId}/{loadingID}/loadingprogress  
[GET](#)/models/{modelId}/validationprogress  
[GET](#)/models/{modelId}/riskvector  
[GET](#)/models/{modelId}/risks  
[GET](#)/models/{modelId}/riskcalcprogress  
[GET](#)/models/{modelId}/report  
[GET](#)/models/{modelId}/refreshcontrols  
[GET](#)/models/{modelId}/issues  
[GET](#)/models/{modelId}/export  
[GET](#)/models/{modelId}/exportAsserted

#### relation-controller

[GET](#)/models/{modelId}/relations/{relationId}  
[PUT](#)/models/{modelId}/relations/{relationId}  
[DELETE](#)/models/{modelId}/relations/{relationId}  
[GET](#)/models/{modelId}/relations  
[POST](#)/models/{modelId}/relations

#### threat-controller

[PUT](#)/models/{modelId}/misbehaviours/{misbehaviourId}/impact  
[POST](#)/models/{modelId}/threats/{threatId}/accept  
[GET](#)/models/{modelId}/threats  
[GET](#)/models/{modelId}/threats/{threatId}

#### authz-controller

[GET](#)/models/{modelId}/authz  
[PUT](#)/models/{modelId}/authz

#### asset-controller

[PUT](#)/models/{modelId}/assets/{assetId}/type  
[PUT](#)/models/{modelId}/assets/{assetId}/twas  
[GET](#)/models/{modelId}/assets/{assetId}/meta  
[PUT](#)/models/{modelId}/assets/{assetId}/meta  
[DELETE](#)/models/{modelId}/assets/{assetId}/meta

PATCH/models/{modelId}/assets/{assetId}/meta  
 PUT/models/{modelId}/assets/{assetId}/location  
 PUT/models/{modelId}/assets/{assetId}/label  
 PUT/models/{modelId}/assets/{assetId}/control  
 PUT/models/{modelId}/assets/{assetId}/cardinality  
 PUT/models/{modelId}/assets/updateLocations  
 PUT/models/{modelId}/assets/controls  
 GET/models/{modelId}/assets  
 POST/models/{modelId}/assets  
 GET/models/{modelId}/assets/{assetId}  
 DELETE/models/{modelId}/assets/{assetId}  
 GET/models/{modelId}/assets/{assetId}/controls\_and\_threats  
 GET/models/{modelId}/assets/meta

### group-controller

PUT/models/{modelId}/assetGroups/{groupId}/size  
 PUT/models/{modelId}/assetGroups/{groupId}/location  
 PUT/models/{modelId}/assetGroups/{groupId}/label  
 PUT/models/{modelId}/assetGroups/{groupId}/expanded  
 POST/models/{modelId}/assetGroups/{groupId}/removeAsset/{assetId}  
 POST/models/{modelId}/assetGroups/{groupId}/moveAsset/{assetId}/toGroup/{targetGroupId}  
 POST/models/{modelId}/assetGroups/{groupId}/addAsset/{assetId}  
 GET/models/{modelId}/assetGroups  
 POST/models/{modelId}/assetGroups  
 DELETE/models/{modelId}/assetGroups/{groupId}

### screenshot-controller

POST/models/{modelId}/screenshot  
 POST/models/{modelSecureId}/takescreenshot  
 GET/models/{modelSecureId}/screenshot

### domain-model-controller

GET/domains/{domain}/users  
 POST/domains/{domain}/users  
 POST/domains/{domain}/palette  
 POST/domains/upload  
 GET/domains/  
 GET/domains/{graphName}/export  
 GET/domains/users  
 GET/domains/ontologies

### user-controller

GET/auth/me  
 GET/administration/users  
 GET/administration/users/{userId}

json-error-controller

GET/error

PUT/error

POST/error

DELETE/error

OPTIONS/error

HEAD/error

PATCH/error

### Stage 1: Constructing your model

You construct the model of your system by placing assets (e.g., servers, data, people, buildings...etc) onto the modelling canvas and creating links (or relationships) between them. A simple, completed model can be seen below.

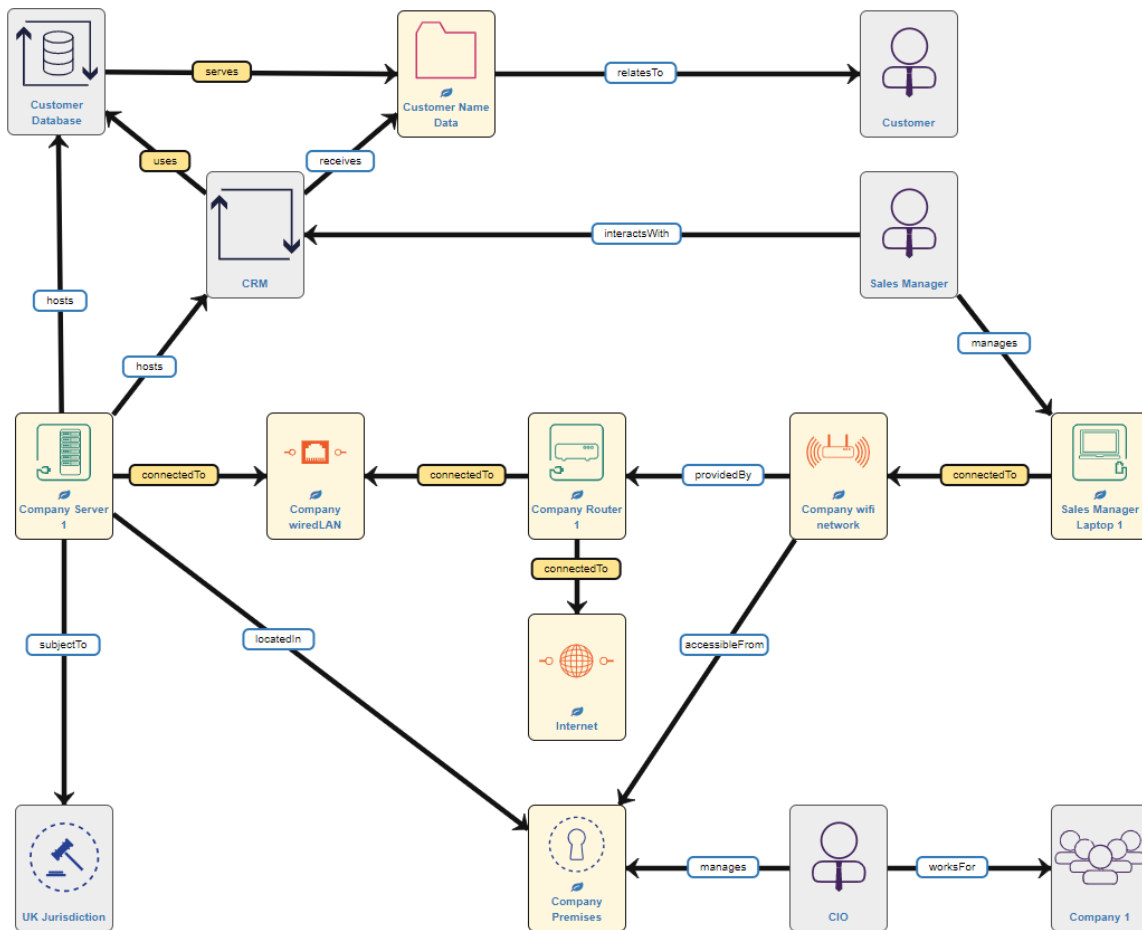


Figure 234. SSM Stage 1: Constructing your model

### Stage 2: Validating your model

When validating the model, SSM will automatically generate inferred assets/relations, will identify the security threats and will suggest security controls to counteract those threats. The validation process also checks whether the model is consistent and complete. If the



validation fails (i.e., the model gets marked as 'invalid') then we will need to go back to Stage 1 and update/correct the model – SSM will tell us where to look.

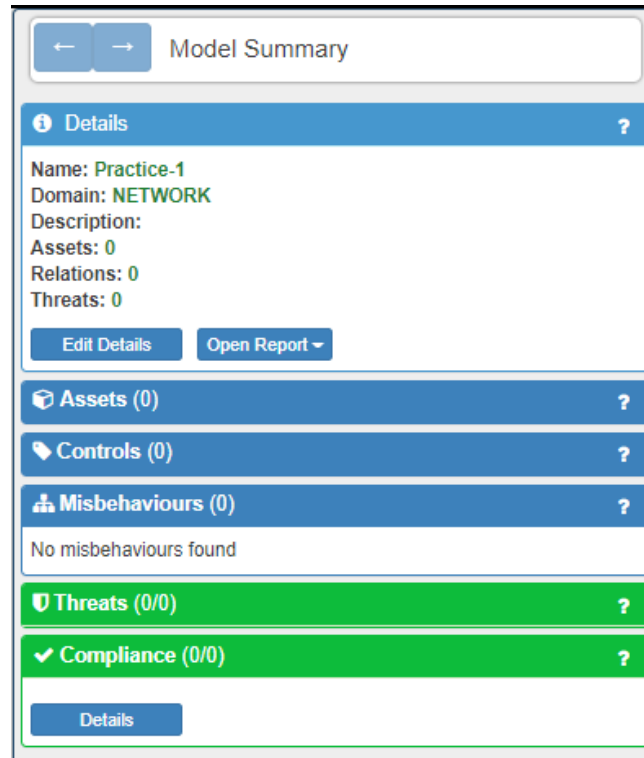


Figure 235. SSM Stage 2: Validating your model

### Stage 3: Managing your risk

You can now explore the possible threats to your system and select security controls for assets or control strategies for threats. The aim is to eliminate, or at least to mitigate, the threats by applying the suggested security Controls to specific assets in the system and then validating the system again with the new security in place. This means you will be able to test the effect of applying the security controls before implementing them in real life.

**Threat Explorer**

**Secondary Threat to Customer Name Data**

System level loss of availability for data "Customer Name Data": all stored copies of data "Customer Name Data" are unavailable.

Likelihood: N/A      Risk: N/A

**Detail**

**Cause**

Misbehaviours at assets:

	Impact	Likelihood	Risk
Loss of Availability at StoredDataSet-Customer Name Data-Company Server 1	Very Low	N/A	N/A

**Effects**

	Impact	Likelihood	Risk
Loss of Availability at Customer Name Data	Very Low	N/A	N/A

**Secondary Effects**

**Control Strategies**

No control strategies found

Accept threat?

Figure 236. SSM Stage 3: Managing your risk

**Other Features:****Creating a New Model:**

This is the SSM Dashboard. To create a new model, click on the *Create New Model* button. You will be asked to name the model (it is entirely your choice what to call it) and to select the **Domain Model** from the drop-down list provided. The domain model is important as it tells the tool what set of relationships to apply when you start modelling your system and what threats to analyse when you validate your model.

When you are happy with the name and the domain, click the *Create New Model* button and you will move to the modelling canvas and can start building your new model. When you return to the Dashboard your model will be listed in the Models list.

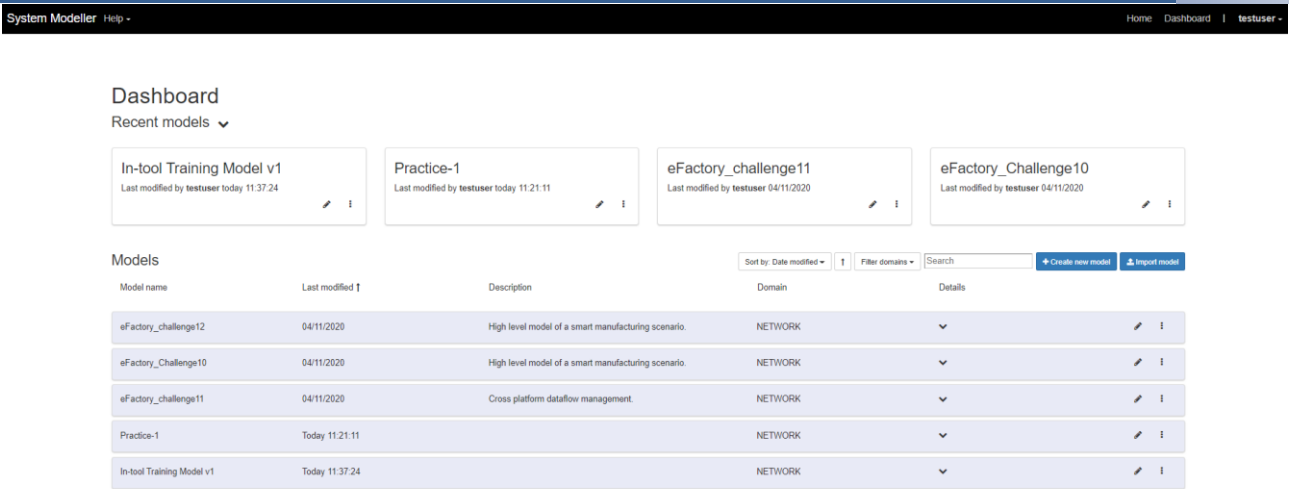


Figure 237. SSM Dashboard: Creating a New Model

### Editing an Existing Model

All models can be accessed by clicking the *pencil icon* (Edit Model) attached to that model. Models that you have recently used will be displayed in the Recent Models panels at the top of the screen.

Models are also displayed on the Models list. Any models you have previously created (and saved), or models you import from a file, will be listed here.

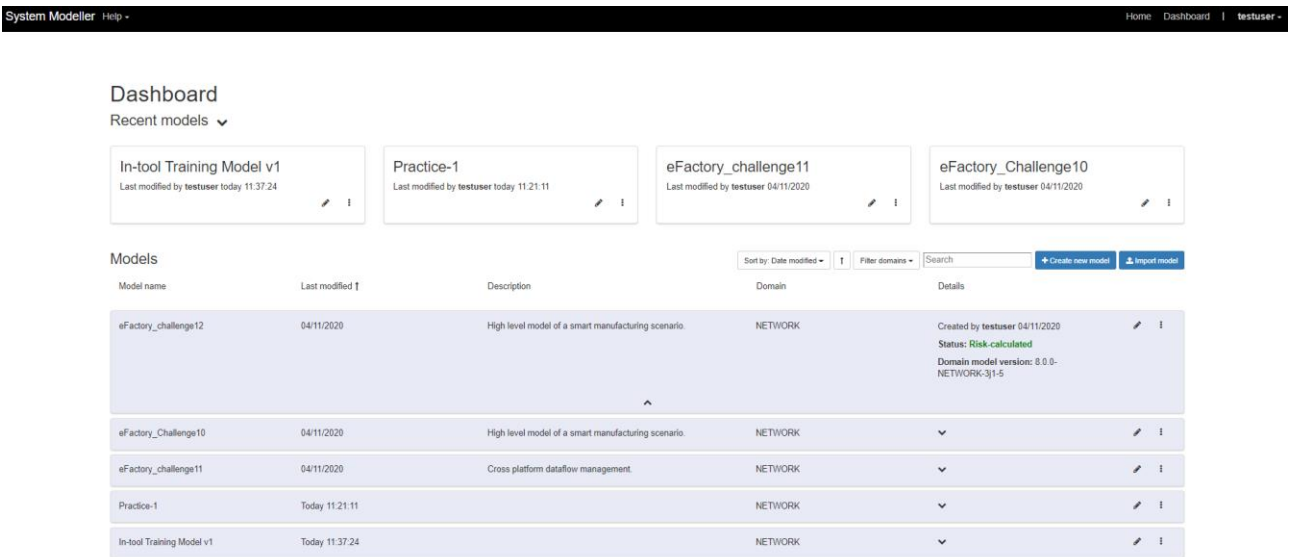


Figure 238. SSM Dashboard: Editing an Existing Model

Clicking on the Details arrow for a single model will display the following information:

- model name,
- date last modified,
- full model description,
- domain used by the model,
- date created,

- current status (not validated / validated / risk calculated).

The model list can be sorted and/or filtered by a range of variables, which makes it even easier for you to find the model you want.

### Importing an Existing Model

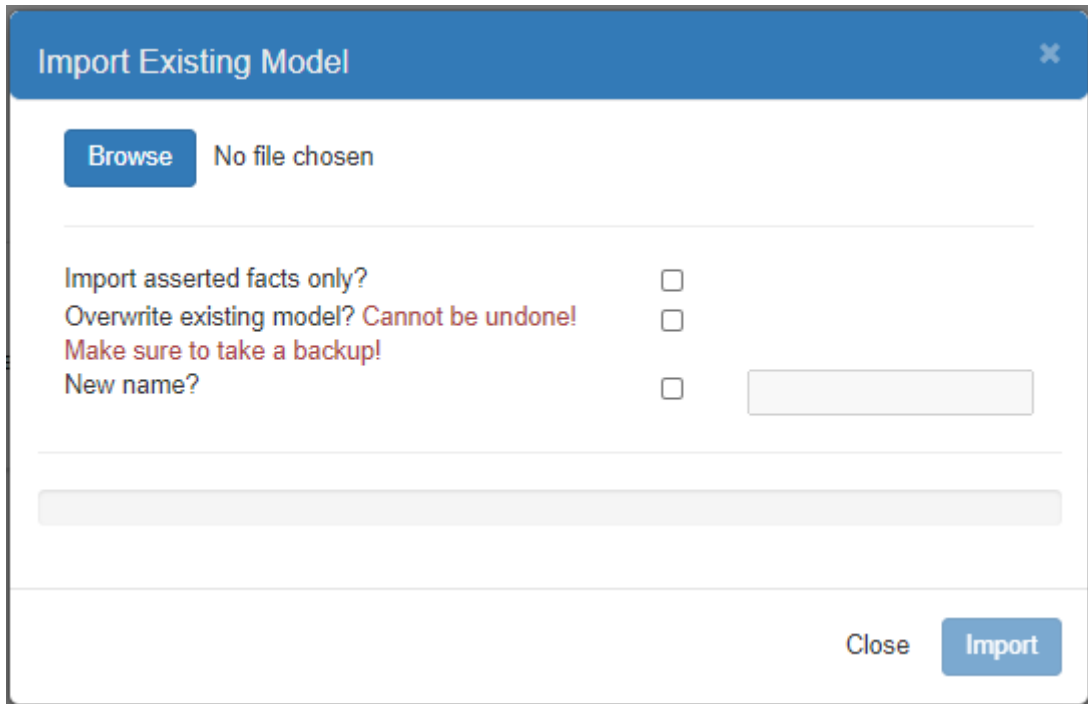


Figure 239. SSM Dashboard: Importing an Existing Model

The Import operation allows you to upload a previously saved model into SSM – just click the Import Existing Model button. A dialog box will appear, and you will be asked to Browse for and Select the file to import.

You will be given a number of import options including:

- Import asserted facts only (e.g., asserted assets, relations)
- Overwrite existing model (attempting to import the same model without this being checked will result in an error)
- New Name (to rename an existing model to avoid overwrite failures).

If none of the import options are selected the file will be imported as is and in full.

#### 3.2.1.18 Industreweb Global

Industreweb Global (IW Global) is a web framework that provides data visualisation, storage, workflow co-ordination, as well as Administration and Security Management tools for the Industreweb Ecosystem. In Figure 240 the main elements and how it interacts with Industreweb Display screens and Industreweb Collect Factory Connectors.

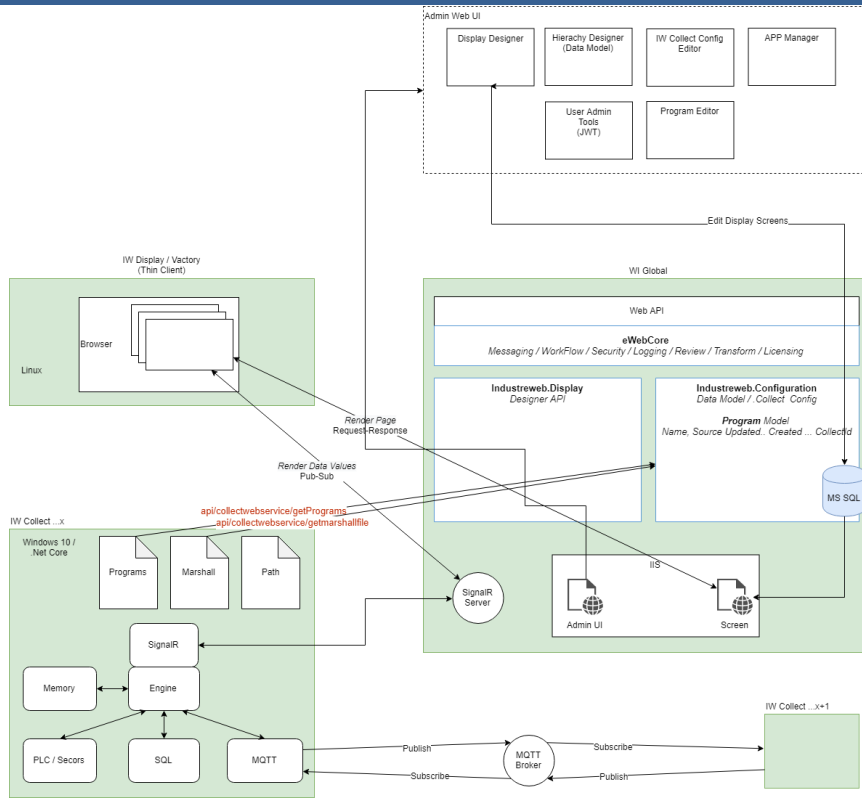


Figure 240. Industreweb Global Architecture and interfacing

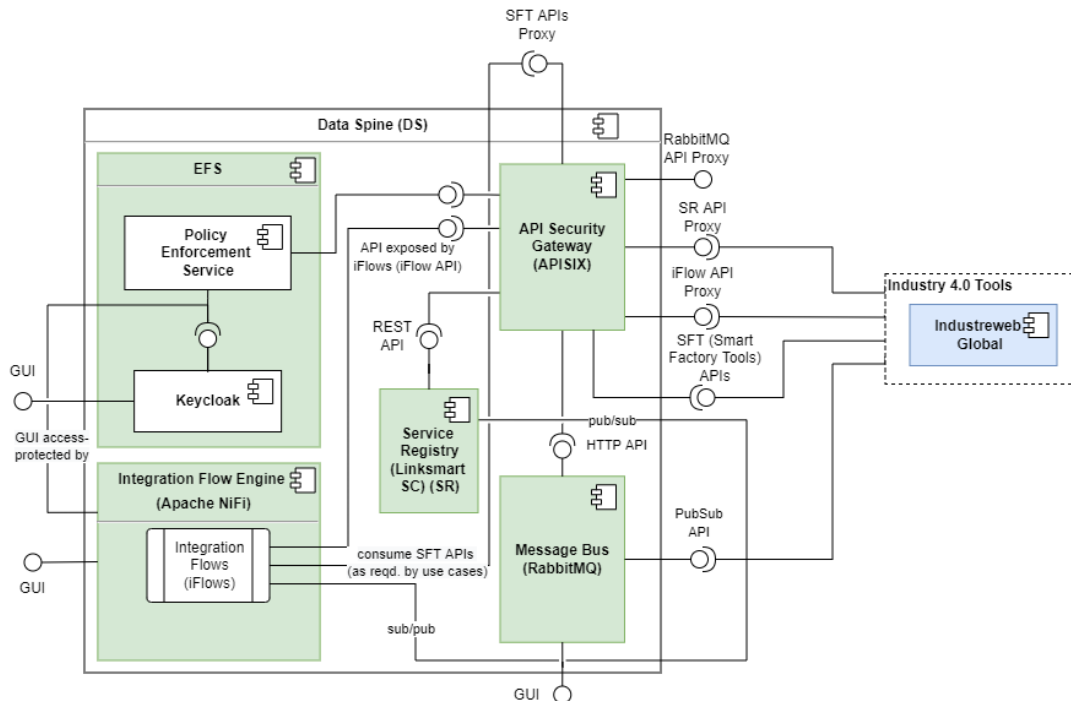


Figure 241. Interaction between Industreweb Global and the EFPF Data Spine

Industreweb Global supports a large API schema defined in its Swagger API which includes those interacted with by Industreweb Collect to determine if its configurations are up to date and if not to request a new set of configurations files. It does so via the following API calls

which are defined at <https://www.industreweb.cloud/swagger/> and described briefly in Figure 242.

REST Endpoint	HTTP Method	Description
/CollectService/PollStatus/{configGuid}&{collectGuid}	GET	Gets a Boolean response to indicate if Collect instance is up to date
/CollectService/GenerateMarshallFile/{collectId}	GET	Gets latest Marshall file
/CollectService/GeneratePathFile/{collectId}	GET	Get latest Path file

Figure 242. Brief Description of Industreweb Global API

### 3.2.1.19 Industreweb Visual Resource Monitoring Tool

The Visual Detection and Alerting system use a Monitoring Box component running in the business premises or manufacturing facility to monitor using a camera and to recognise objects within its field of vision. It uses an Intel Visual Processing AI Unit to detect objects that it recognises from a pre-learnt AI model.

IW Collect running on the monitoring box then detects these events and based on a set of rules determines what Actions to perform. This could be to notify by email or SMS, to sound a siren, to light a warning lamp, push data to the EFPF cloud or display a message on a screen or dashboard.

Subsequently using the IW Global Administration tools, the Events and Actions that should happen when an object is detected need to be created and deployed to the IW Collect node.

When the system is executed, it loads in its AI model and Actions and Events and starts its continual scanning. When an object is detected a probability of match is made, which if greater than a redefined user threshold the associated Actions can be carried out. The technology can be applied to any scenario where the system should detect a specific object or objects and trigger actions whether preventative, warning or confirmative Actions need to be undertaken.

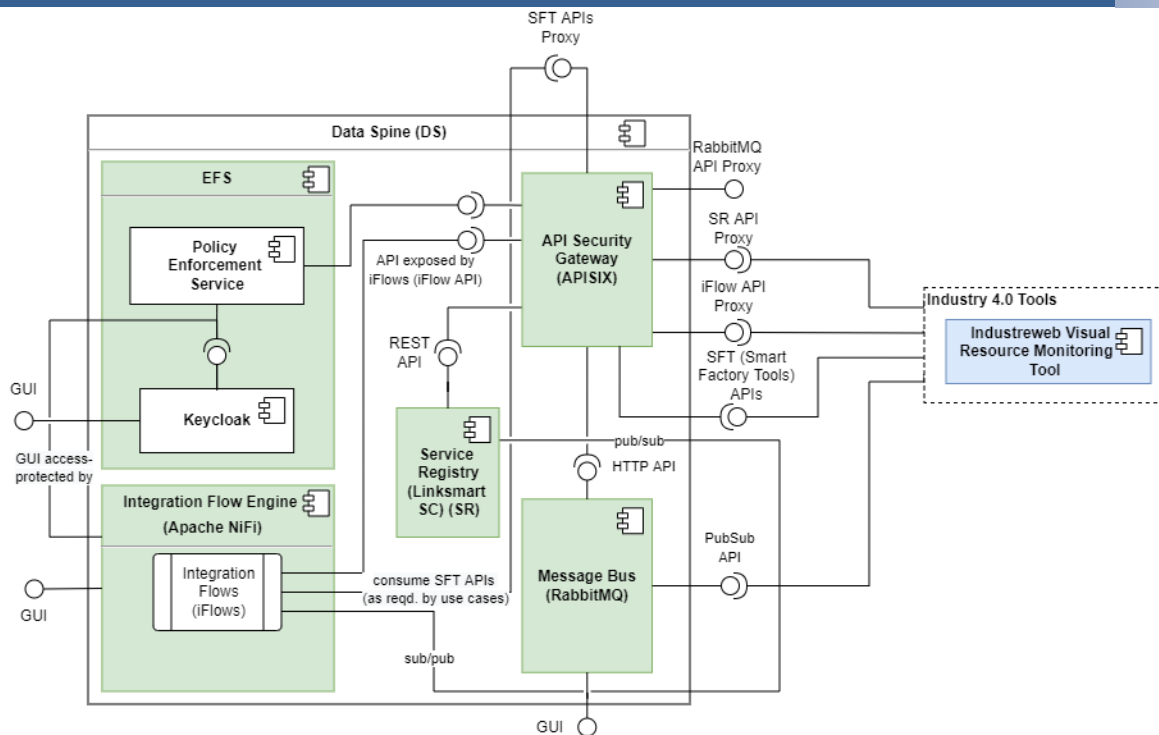


Figure 243. Interaction between Industreweb Visual Resource Monitoring Tool and the EFPF Data Spine

Since the Industreweb Visual Resource Monitoring Tool is based on the Industreweb Collect Factory Connector it has the same API support – see Section 3.2.1.5.1.

### 3.2.1.20 Symphony Event Reactor

The Symphony Event Reactor allows the integrator to define a behavior in response to specific ingress *events*, by taking actions and potentially engaging an alarm lifecycle management process. Actions include pre-configured low-level procedures, custom code provided by the integrator, or generation of new events.

A more detailed representation is illustrated in Figure 244:

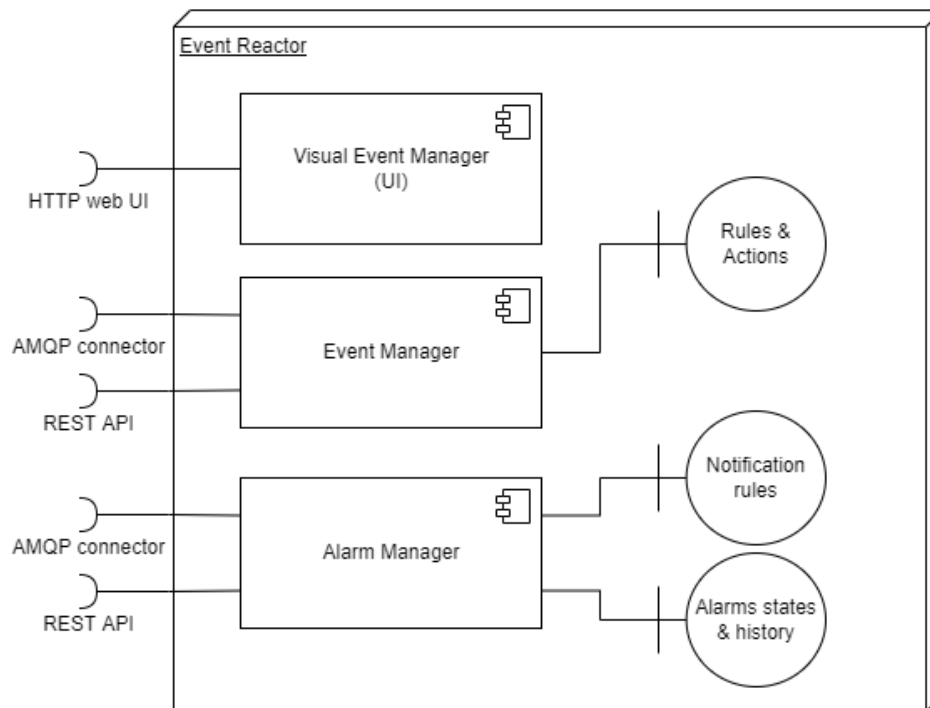


Figure 244. Symphony Event Reactor Architecture

From an architectural point of view, the Event Reactor is split into three separate independent services:

- **Event Manager (EM):** it executes custom rules, combining information coming from different sources (local sensors and device monitors, user actions, CCTV cameras, intrusion detection systems, etc.) and message brokers (e.g., AMQP, MQTT) to determine actions to be taken. Actions include actuations on field devices, activation of scenarios, generation of events, notifications, and alarms, and so on.
- **Alarm Manager (AM):** reacts to *alarm conditions* to generate new alarms and update the status of existing ones. Correspondingly, it notifies users of such changes through a configurable priority-based alert routing system to target a single, a group or mixed sets of users with SMS, emails, pop-ups, etc.
- **Visual Event Manager (VEM):** it provides the integrator with an intuitive web UI based on visual block-level rule definition interface.

## Event Manager

The Event Manager (EM) deals with the management and processing of events through the use of rules specified by the user. The EM architecture is composed of a set of specialized sub-modules that allow greater deployment flexibility by allowing the definition of clusters of EM nodes in order to activate appropriate HA and load balancing policies.

The Event Manager is composed by:

- **Rules Status Registry:** It is a content store accessible from all cluster nodes. It contains the metadata of all the rules and the execution context information existing at a given moment in the system.
- **System Events:** It is a local message flow to the cluster node, coming from the event sources through appropriate drivers that allow interfacing with the different types of



communication protocols, such as AMPQ brokers, MQTT, HTTP (WS) streams, GRPC streams and other proprietary protocols. The Event Manager can make use of these messages flow to trigger the execution of the rules.

- **Visual Event Manager:** It is a Web UI through which the user is able to define the rules using a graphic tool.

EM rules are defined through a formal structure that is represented by a JSON object. This definition is called manifest, through which the metadata describing the rule, the type of resources involved (e.g. resources observed for triggering the rule, resources on which an action is performed, etc.), controls, the actions that can be performed on a given resource according to its type, the statements and, in general, the operating logic of the rule itself and finally other useful constraints, for example to implement some control actions.

Once the manifest has been defined, it is possible to create one or more instances of rules by defining the binding between real objects (e.g., sensors, events, actuators, field devices, etc.) and the manifest template. Then one or more instances are "injected" into the EM engine that will take care of defining and managing an execution context for each instance.

EM provides a single HTTP endpoint (not REST compliant) used for all administration and management operations. The body format is derived from a proprietary protocol and is defined through a JSON schema as follow:

```
{
  "cmd_type": 21,
  "subcmd_type": 1,
  "plugin": "em",
  "command": <command>,
  "args": {
    .... // optional
  }
}
```

Where:

- “command” is one of available command supported by EM (see below table)
- “args” are command specific parameters

All APIs are called using the POST method on the single endpoint.

Command	Input parameters	Description
rule_def_new	<ul style="list-style-type: none"> <li>• id: (opt.) uuid for the rule. If not defined will be created by EM</li> <li>• name: rule name</li> <li>• inputs: dictionary for input objects</li> <li>• controls: dictionary for control objects</li> <li>• parameters: (opt.) if exists contains the value dictionary for parameters</li> <li>• triggers: dictionary for trigger objects</li> <li>• statements: list of statements</li> </ul>	Create a new rule definition.
rule_def_list		Return the rules list with related instances
rule_def_get	<ul style="list-style-type: none"> <li>• id: rule uuid</li> </ul>	Return the rule definition
rule_def_patch	<ul style="list-style-type: none"> <li>• id: rule uuid</li> <li>• (opt.) same parameters of <i>rule_def_new</i></li> </ul>	Alter the rule definition
rule_def_del	<ul style="list-style-type: none"> <li>• id: rule uuid</li> </ul>	Remove the rule definition. Used only if the rule has not instances

rule_inst_new	<ul style="list-style-type: none"> <li>def_id: uuid of rule definition</li> <li>bindings: (opt.) association between real objects and manifest objects</li> <li>parameters: (opt.) rule parameters if set in definition</li> </ul>	Create a new instance for a rule
rule_inst_get		Return the list of instances for a rule
rule_inst_start	<ul style="list-style-type: none"> <li>id: uuid of the instance</li> </ul>	Start an instance
rule_inst_stop	<ul style="list-style-type: none"> <li>id: uuid of the instance</li> </ul>	Stop an instance
rule_inst_del	<ul style="list-style-type: none"> <li>id: uuid of the instance</li> </ul>	Remove a stopped instance
rule_inst_pause	<ul style="list-style-type: none"> <li>id: uuid of the instance</li> <li>seconds: pause duration</li> </ul>	Pause an instance for x seconds
rule_inst_unpause	<ul style="list-style-type: none"> <li>id: uuid of the instance</li> </ul>	Unpause the instance
is_paused	<ul style="list-style-type: none"> <li>id: uuid of the instance</li> </ul>	Check if instance is in pause

Figure 245. Symphony Event Manager Commands

The same rules defined through the API can be composed through the Visual Event Manager web tool. This tool, in addition to the editing functions, also provides some useful interfaces for the management of the rules and their monitoring. For example, it is possible to view which rules are instantiated, their execution status, check for alarms or execution errors and possibly perform a fine monitoring for running instances using a step by step debugger.

EM provides a set of specific APIs used by the Visual Event Manager:

REST Endpoint	HTTP Method	Description
/external/v1/rule/start?instanceid={uuid}	POST	Starts a rule
/external/v1/rule/stop?instanceid={uuid}	POST	Stop a rule
/external/v1/rule/pause?instanceid={uuid}&seconds={sec}	POST	Pause the rule for x seconds
/external/v1/rule/unpause?instanceid={uuid}	POST	Unpause the rule
/external/v1/rule/list	GET	Get installed rules

Figure 246. Symphony Event Manager APIs

Figure 247 shows a simple rule that activates a monitoring on an object sensor and defines a trigger when a change occurs. The rule shows a simple if-then block. If the value is over the threshold of 1KW, then the EM activates an alarm to the AM component.

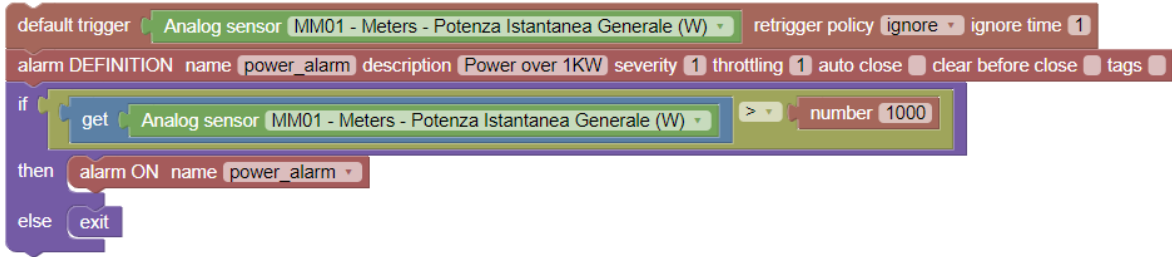


Figure 247. Symphony Event Manager Example Rule

The visual rule is compiled in the manifest and subsequently is built an injectable instance. The control flow activated during the creation of the rule (via VEM) and its activation is represented in Figure 248.

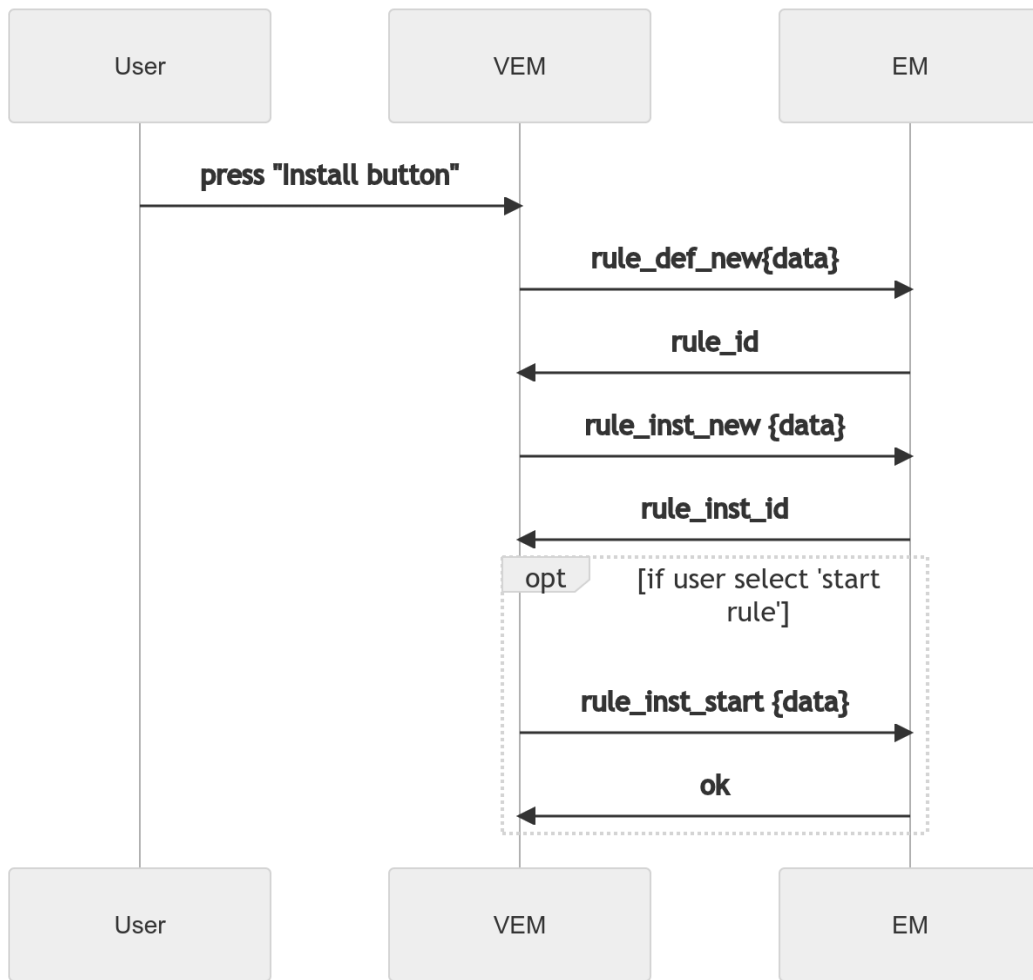


Figure 248. Control Flow Activated during the Creation of a Rule

### Alarm Manager

The AM module takes responsibility for the full management of the lifecycle of alarms. To this purpose, the following concepts and actors are involved:

- an alarm **source** (any external entity) executes one or more **rules** to determine an alarm **condition**. When a rule is triggered (e.g., a sensor’s value is above a threshold), the source notifies the condition change to AM;
- the AM reacts to a change in state of the **condition**, according to a specific **condition descriptor**, and generates or updates the state of a given **alarm**;
- the state of an alarm can be also changed by a **lifecycle event** (e.g., the alarm has been notified through one of the external channels, or it has expired having reached a deadline) or by a **user action** (e.g., it has been acknowledged or closed by an operator);
- whenever an alarm changes its state, an update is published to all the relevant **subscribers** on a specific AMQP topic. An external entity can place a subscription by defining a **filter** which identifies a class of alarms based on the fields contained in the **condition descriptor**.

The above defined relations are depicted in Figure 249.

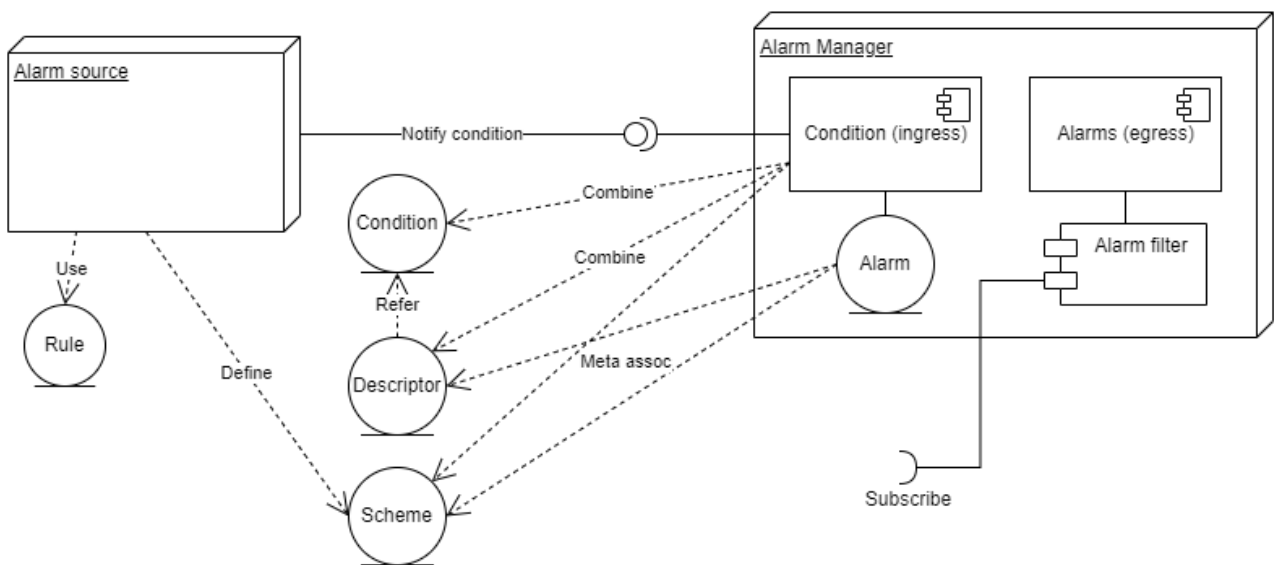


Figure 249. Alarm Manager: The Lifecycle of Alarms

The APIs provided to perform all the necessary operations are briefly described in the following table. Detailed data structures are available in the OpenAPI specification of the service.

REST Endpoint	HTTP Method	Description
/infos	GET	Get software information
/stats	GET	Provides statistics about the REST endpoint and internal DB calls
/alarms	PUT	Update the state of a list of alarms (provided in request body)
/alarms	DELETE	Delete a list of alarms (provided in request body)
/alarms/{id}/events	GET	Return the list of events (state changes) for a given alarm
/alarms/{id}	GET	Get alarm details by ID
/alarms/{id}	PUT	Update the state of an alarm
/alarms/{id}	DELETE	Delete an alarm by ID
/alarms/snapshot	POST	Return the list of all alarms matching a given filter (provided in the request body)

/alarms/count	POST	Return the number alarms matching a given filter (provided in the request body)
/filters	GET	Return the list of available alarm filters
/filters	POST	Define a new filter
/filters/{id}	GET	Get filter details by ID
/filters/{id}	PUT	Modify filter by ID
/filters/{id}	DELETE	Delete a filter by ID
/subscriptions	POST	Place a new subscription given a filter (provided in the request body)
/subscriptions/{id}	GET	Get infos about the subscription
/subscriptions/{id}	PUT	Modify the subscription
/subscriptions/{id}	DELETE	Cancel the subscription
/export/prepare	POST	Prepare an export archive
/export	GET	Get a list of available export archives
/export/{id}	GET	Download an export archive

Figure 250. Symphony Alarm Manager API

### 3.2.1.21 Symphony Data Storage

Nextworks Data Storage is a highly scalable and high-performance data storage which is designed to handle large amount of AMQP/MQTT data. Its architecture is illustrated in Figure 251. It accepts AMQP and MQTT as data source input and provides REST APIs as output to retrieve and fetch the stored timeseries. Along with the REST APIs, Symphony Data Storage provides a console to configure the internal components.

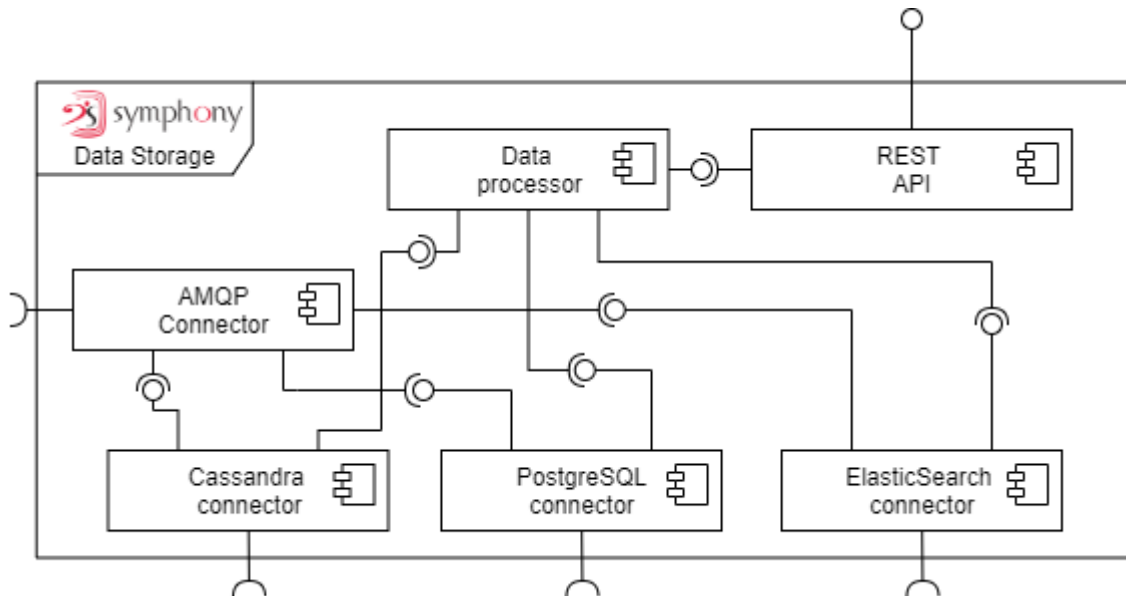


Figure 251. Symphony Data Storage Architecture

The configuration of the Symphony Data Storage can be performed through its internal interactive shell, which offers a set of command to define new local or remote backends, to define new timeseries, monitor the status of the system, etc...

These are the most important commands that can be executed from the shell to an instance of the Symphony Data Storage:

Command	Description
<b>config</b> [env=<env_ID>] [gid=<GID>] [section=datastores timeseries]	Display the current configuration of the server
<b>datastore</b>	Display/modify the configuration of a backend (datastore) Subcommands:  <b>list</b> : list all the datastores  <b>show</b> [datastore=<datastore_name>]: show the datastore configuration  <b>alter</b> [datastore=<datastore_name> followed by backend specific params]: modify the datastore configuration  <b>add</b> [name=<datastore_name> backend=<backend_name> followed by backend specific params]: add a new datastore  <b>remove</b> [name=<datastore_name>]: remove a datastore
<b>watch</b> gid=<GID> [watch specific params]	Create a new watch or update an existing one. Specific params are: [cmd_type=<cmd>] [backend=<DS name>] [active=<bool>] [watch=<bool>][direct=<bool>][rabbit=<exchange @host>][aggregation=Off <period>,<func>,<target>;...][prefilter=<fspec>][stream=<bool> [stream_interval=<int>]] [bulk=<bool> [bulk_interval=<int>] [bulk_size=<int>]][fields=<field1[,...]>][autosync=<bool>][preset=<preset>][detach_preset=<bool>][subsampling=<bool>][mirrors=<list of mirror datastore IDs>]
<b>disable_watch</b>	Disable a watch
gid=<GID>	
<b>enable_watch</b>	Enable a watch
gid=<GID>	
<b>active_watches</b>	Get a list of active watches
<b>remove</b> gid=<GID>[,<GID>,...] [force=<bool>]	Remove a timeseries. if 'force' is True, the timeseries is removed even if the back-end is not active
<b>delete</b> gid=<GID>[,<GID>,...] [start=<start_ts> [end=<end_ts>]]	Delete timeseries history, leaving the watch active

Figure 252. Symphony Data Storage Commands

As it is shown in its UML component diagram in Figure 251, it has Connector for ingesting data from message bus services and REST API for retrieving the data.

Symphony Data Storage accepts only specific data model as input. The data model schema is:

```
{
  "oid": "<id>",
  "timestamp": "<iso_ts>",
```

```
"value": {<payload_attributes>}
}
```

The attributes are:

- “oid”: the unique id of the publisher e.g., “TemperatureSensor100”
- “timestamp”: the timestamp in IOS 8601 format.
- “value”: the payload of the published message could be with single or multiple attributes.

A detailed description of the REST APIs for querying the Symphony Data Storage is described in the following table:

REST Endpoint	HTTP Method	Description
/api/v1/object/{oid}?detail=_any	GET	Gets the complete set of published data separated by timestamp including all attributes.
/api/v1/object/{oid}?detail={payload attribute}	GET	Gets specific payload attribute. It could be single or multiple (comma separated)
/api/v1/object/{oid}?limit={number}	GET	Gets limited number of data
/api/v1/object/{oid}?order={asc/desc}	GET	Gets the data with “ascending” or “descending” order
/api/v1/object/{operationId}?start={timestamp in Unix Epoch}	GET	Gets the data starting from specific time.
/api/v1/object/{operationId}?start={timestamp in Unix Epoch}&end={timestamp in Unix Epoch }	GET	Gets the data in specific time period.

Figure 253. Symphony Data Storage REST API

### 3.2.1.22 Symphony Resource Catalogue

The Symphony Resource Catalogue is made of two separate micro-services: an Object Catalogue (called **Plinsky**) and an Object Registry (called **Object & Service Registry**). The two services are described in the following.

The **Plinsky** service is a repository used to define and query a domain composed of **structures** (e.g., buildings, factories) populated by **objects** (e.g., sensors, devices). Both structures and objects are characterized by capabilities (i.e., functions) and relation properties.

The component includes an OWL2 representation language with an associated inference engine, which allows the representation to be flexible, powerful, and capable of rich queries.

More precisely, an RDF triple stores the ontology. The inference engine permits Plinsky to make use of SPARQL query language to infer the relationships among objects and building areas.

The assessment of functionality and interconnections of devices allows the system to have a detailed degree of knowledge about the internal operation of the represented system, and to know or infer which events affect which objects, and where. Along with a description of the topology of the building and the interconnection between zones, this allows for queries to identify isolated parts of the compound (e.g., to discover evacuation routes).

For example, suppose that a fire is detected in room R of a building – it is possible to query the repository in order to find whether R isolates parts of the building, how to vacate other rooms without travelling through R, finding the devices located in R, and, assuming they stop working, figure out which other devices will remain unaffected and which might stop working, including those that might prevent access to other areas (e.g., electronic locks etc.).

Even under normal conditions, Plinsky can be effectively used to assess energy load balance, network coverage, environment conditions, machines’ operating status and conditions, and a lot of other information elements in support of the maintenance of a structure.

Some of the available APIs define transitive relationships (e.g., sub/supermeter to indicate that a specific meter measures part of the quantity measured by a supermeter). These transitive relationships allow for inferential queries and, by means of relationship modelling, can be used to infer other relationships, such as actuation, which implies interconnection.

Plinsky has the ability to minimise the information required at population time; it can be configured to “know” about various types of devices, for example, meaning that addition of several new devices to the repository does not require much information besides the type and the location of the new device.

The Plinsky ontology extends SAREF [SAR22], the Smart Applications REFerence ontology, which formalises the smart applications domain by describing core concepts and relationships between them. SAREF overview is summarised in the following figure (source <https://saref.etsi.org/>):

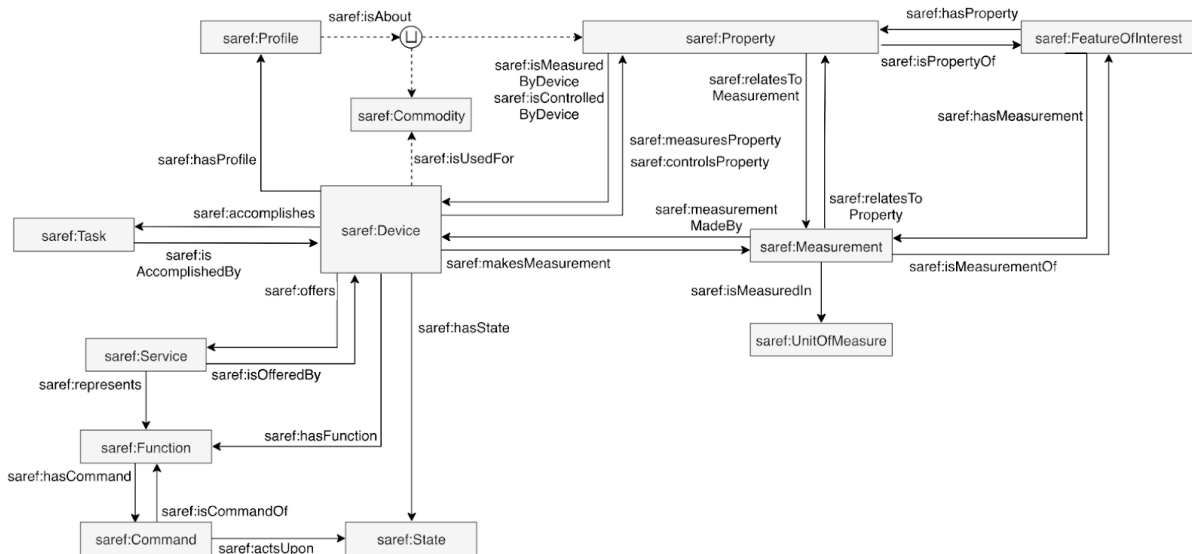


Figure 254. SAREF Ontology Overview [SAR22]

Plinsky also includes SAREF extensions, such as those geared towards building representations (SAREF4Building) and environment (SAREF4Envi).

The main APIs, formalised in protobuf and exposed via a gRPC interface, provide all the functions needed to setup, manage, and query an instance of the repository:

- CRUD (Create, Read, Update, Delete) for devices and zones, including associations between them (e.g., device X is a meter measuring property P of device Y, device W is used to actuate device Z etc.)



- specify devices and zones location, as well as the topology of zones (e.g., “light 1” is in “office 421”, which is in the “main building” and is connected to “central corridor”)
- associate an image to a zone
- define a topological map for a zone, by specifying a polyline over a zone’s image

Service	RPC	Description
<b>Ontology</b>	ZoneCreate/Get/Modify/Delete	Zones CRUD
	ObjectCreate/Get/Modify/Delete	Objects CRUD
	ObjectCreateBulk/ObjectDeleteBulk	Object creation e removal, bulk version
	ActuatorSet/Get	Set/query actuator relationships
	MeterSet/Get	Specifies that (or queries for) objects measure properties of other objects (e.g., a power meter measuring the consumption of machinery)
	MetersSupermeterSet/Get	Specifies that (o queries for) meters that are “supermeter” of other meters, (e.g., a meter for the whole floor wrt a meter for a space within the floor)
	DeviceWatchSet/Get	Specifies that a device watches an area (e.g., a camera, a volumetric sensor, etc.)
	LastModificationTime	Timestamp of the last configuration modification performed
<b>Topology</b>	GeometryCreate/CreateWithId/Get/Modify/Delete	Geometries CRUD. The “WithId” version permits to specify an ID, otherwise a fresh one is generated.
	ImageCreate/ImageWithId/Get/Modify/Delete	Images CRUD
<b>Health</b>	Check	Service health
<b>Maintenance</b>	Backup	Backup
	Restore	Restore
	Reset	Factory reset
	Sync	Synchronise local installation to the cloud

Figure 255. Symphony Resource Catalogue APIs

An operator interacts with Plinsky by using a repl (Read-Eval-Print Loop) inside a terminal.

```

$ plinskyrepl
This is Plinsky 0.4.18. Call me 'Jena'.

? or 'help' for a list of commands, 'listcmds' for a brief description.
'help COMMAND' for its documentation.

Connecting to Plinsky on localhost:50051
Using v2 stubs.

Plinsky>
    
```

In addition to traditional edit functions, such as create object, create zone, etc. it is possible to inspect the current data model:

```

Plinsky> datamodel
saref:Device
├── nxw:CompositeObject
├── saref:FunctionRelated
│   ├── saref:LightingDevice
│   │   └── nxw:Hue
│   └── nxw:TransportationDevice
│       └── nxw:Lifter
    
```



```
ObjectCreate Success for 44: 0.081s
Plinsky> createobj 66 PM10_METER "Particulate 10 meter" ""
ObjectCreate Success for 66: 0.224s
Plinsky> createobj 77 PM2_5_METER "Particulate 2.5 meter" ""
ObjectCreate Success for 77: 0.045s
```

and declare them to be parts of sensor 33:

```
Plinsky> setcomposite 44 33
CompositeSet Success for 44 -> 33: 0.133s
Plinsky> setcomposite 66 33
CompositeSet Success for 66 -> 33: 0.036s
Plinsky> setcomposite 77 33
CompositeSet Success for 77 -> 33: 0.145s
```

Asking for parts of 11 (the top level multi sensor):

```
Plinsky> getcomposites * 11
Part      Whole
-----
22        11
44        11
66        11
77        11
33        11
```

This shows that while the relation between 33 and 11, and 22 and 11 is direct, the one between 44, 66, 77 and 11 is indirect, and yet it is correctly returned.

Plinsky offers a cloud version (Plinskycloud) to host the setup of different plants, exposing REST APIs to synchronise several configurations between local installations and the cloud. This permits to easily query the status of different scenarios in a unified and accessible way.

The OSR module (**Object & Service Registry**) is a system component that provides runtime functions for the localization of system's resources, providing the capability to verify both the presence of resources (their availability) and also to find the details information regarding the interfaces exposed by the resource itself to allow the correct communication.

The OSR service is based on the etcd project (<https://etcd.io/>) from which it inherits some fundamental characteristics of distributed and reliable key-value storage. In the context of OSR the intrinsic capabilities of etcd are redefined by adding a higher level layer that introduces the functionality of register and resolver.

The basic features of OSR are briefly described in the following points:

- nodes clustering to support the implementation of distributed architecture
- dynamic registration of resources at runtime
- structured data storage to provide a complete and easily identifiable set of location and access details, such as the ability to support multiple protocols information.
- persistence and multi-node synchronisation
- independent lease time management for each entry or group of entries, with support of automatic removal of expired entries.

There are two main types of resources in OSR:

- **services:** system components that provide specific high-level functionalities and that expose a set of formally defined interfaces to be used by other entities. Each service is characterised by its own identifier that uniquely represents it in the architecture.
- **objects:** in general they refer to an heterogeneous set of elements representing different types of physical entities and that provide interfaces for their control and / or monitoring. Some examples of objects can be simple entities such as analog sensors (temperature, current, pressure, etc.) or more complex objects, composed by several sensors, actuators, registers etc. Each object is fully characterised through a unique identifier and formal type.

The registration of a new entity within OSR depends on the context of use. In general, system service components perform automatic registration at bootstrap or when a modification is requested in the deployment setup (for example when the service is migrated from one node to another within a cluster). The registration of objects is usually carried out by a higher-level component that manages communication to the physical object via proper drivers, such as the HAL (Symphony Hardware Abstraction Layer) or other gateway and protocol adapter entities. As the services, the object resources can also be represented through some predefined types that allow the categorization of controlled objects and their interfaces.

Each entry on OSR is defined by a JSON structure similar to the following:

```
{
  "<TYPE>": {
    "server": "<HOST>",
    "port": <PORT>,
    "interfaces": {
      <IFACE>: {
        "host": <HOST>,
        "port": <PORT>
      }
      ...
    },
    ...
  }
}
```

Where:

- **<TYPE>**: endpoint protocol type, for example:
  - CORBA
  - REST
  - gRPC
  - other
- **server**: default host or IP address to be used to contact the endpoint.
- **Interfaces**: dictionary of all supported interfaces and exposed through the endpoint. Each interface can be identified with the <IFACE> parameter whose structure depends on the endpoint type. For example, gRPC APIs might be represented through a fully qualified gRPC service name (<package>.<service name>) while REST endpoints might be represented using the url path.

Access to the OSR service is made available through multi-language libraries (Python, Go, C++) that abstract the internal complexity and ensure a uniform and logically consistent use of the service, regardless of the technology chosen for the implementation of the client counterpart. The following snippet shows a simple registration flow for a hypothetical REST endpoint using the Python library.

Library initialization: in this phase the client starts the communication with the OSR. In the init function can be passed a TTL parameter (time to leave) used to define the default duration of the registered keys.

```
osrclient = OSR.GetOSRClient()
osrclient.Init(host=<osr server host>, port=<osr server port>, TTL=10)
```

Definition of the REST endpoint: when creating the endpoint is specified the type and coordinates (host and port) used for reachability.

```
ep = Endpoint(eptype=EndpointType.REST, server=<endp host>, port=<endp port>)
```

Interfaces definition: for each endpoint one or more interfaces can be defined, for example in case of is necessary to differentiate the pointing to different addresses. In the case of REST API, the interface is generally identified through the base path or more specific url portion.

```
ep.addInterface('/api/v1/')
ep.addInterface('/api/v2/sensors', host=<host 2>, port=<port>)
```

Entity registration: each resource is uniquely represented by a key (such as a service name). The library will take care of defining a new context status for each entity in order to ensure correct management and monitoring, for example it is the library itself that transparently manages the registration lease time.

```
osrclient.RegisterService('service.name', ep)
```

### 3.2.1.23 EFPF Dev-Portal

The EFPF-Dev Portal serves as the documentation hub for hosting the documentation, tutorials, and guides for the components in the EFPF ecosystem. It provides a facility to the tool/service providers in EFPF to submit their documentation in the form of simple markdown files to get it published onto the portal. A process that makes use of pull requests (PR) has been defined to make the documentation and review process easy. The portal internally converts the markdown files and publishes them as HTML Web pages. The documentation is made accessible to everyone over the Web. The EFPF Dev-Portal is integrated with the EFPF Portal, as illustrated in Figure 256. The EFPF Dev-Portal also provides the functionality to visualise the APIs registered into the Service Registry, which is further described in Section 3.3.

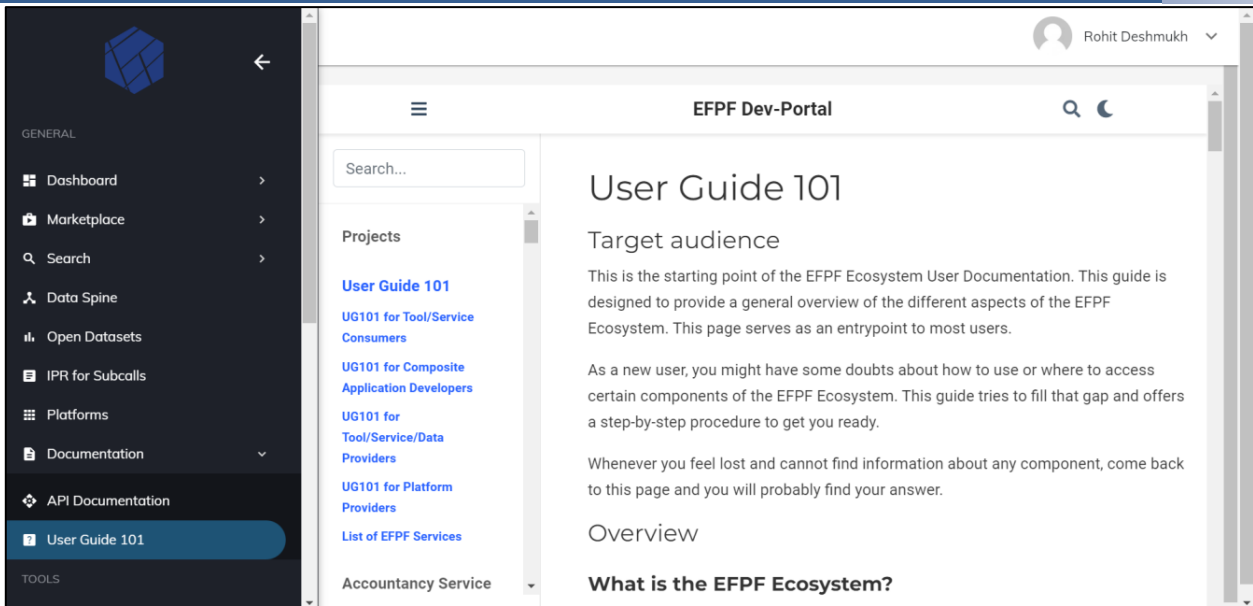


Figure 256. EFPF Dev-Portal Available through the EFPF Portal

### 3.2.1.24 Tikki Ticketing System

Tikki is a web-based ticketing system for support activities regarding EFPF. Incoming emails sent to the configured support contact addresses are processed by first line support and assigned to relevant EFPF staff to be handled, allowing multiple people to handle tickets while maintaining a well-defined external points of contact. Tikki authenticates EFPF users with the security portal, integrating into the EFPF SSO environment. A redacted screenshot of the UI showing how Tikki can manage support emails can be found below in Figure 257.

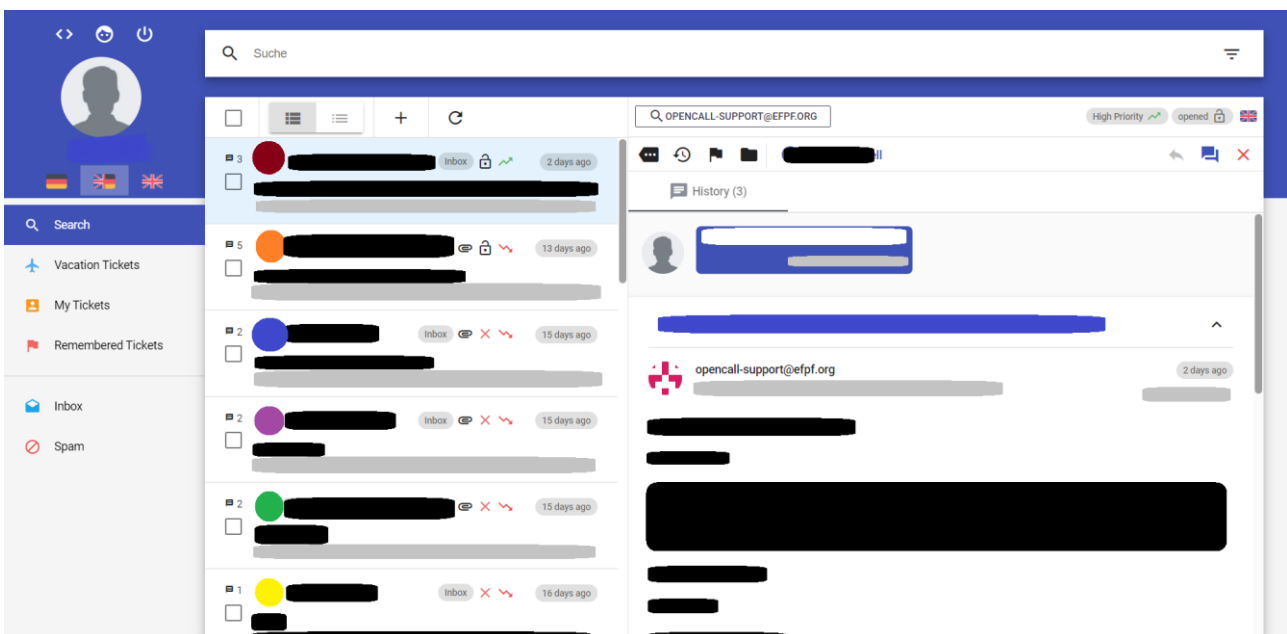


Figure 257. Tikki UI

### 3.2.1.25 Monitoring & Alerting Service

With the growth of micro-service-based architectures, the points of failure have distributed across multiple applications and servers. This raises a need for an active monitoring solution

that helps the administrators and the application developers to know the failures before even the users of the systems notice them. The Monitoring and Alerting service addresses these challenges. It provides the following functionalities:

- **Metrics Monitoring:** Metrics collected from the containers running on different hosts includes CPU, memory usage, and network transactions. This is achieved using Prometheus.
- **Log Management:** Logging gives an insight into the running applications and their behaviour. This can be used by the service providers to deduce the causes for malfunctioning and to get an overview of the target’s behaviour. This is realized using Loki.
- **Visualization and Alerting:** Monitored metrics and logs are visualized and explored using different Grafana panels. Alertmanager is used to manage the alerts generated by Prometheus.

Figure 258 illustrates a monitoring infrastructure deployment where a Monitoring server monitors different target hosts running in local or remote premises. Prometheus, Loki, Grafana, and Alertmanager are installed on the Monitoring server. Vector and cAdvisor are installed on the target hosts to extract the logs and metrics respectively from the target hosts. Vector collects the docker logs and cAdvisor collects different docker metrics. Prometheus scrapes the metrics from cAdvisor. Loki, on the other hand, does not have a scraping mechanism and hence the logs are pushed by Vector to Loki. Both the metrics (collected by Prometheus) and the logs (collected by Loki) are visualized using Grafana. Generated alerts are forwarded to Alertmanager which then further groups, deduplicates and routes to the relevant applications such as email servers.

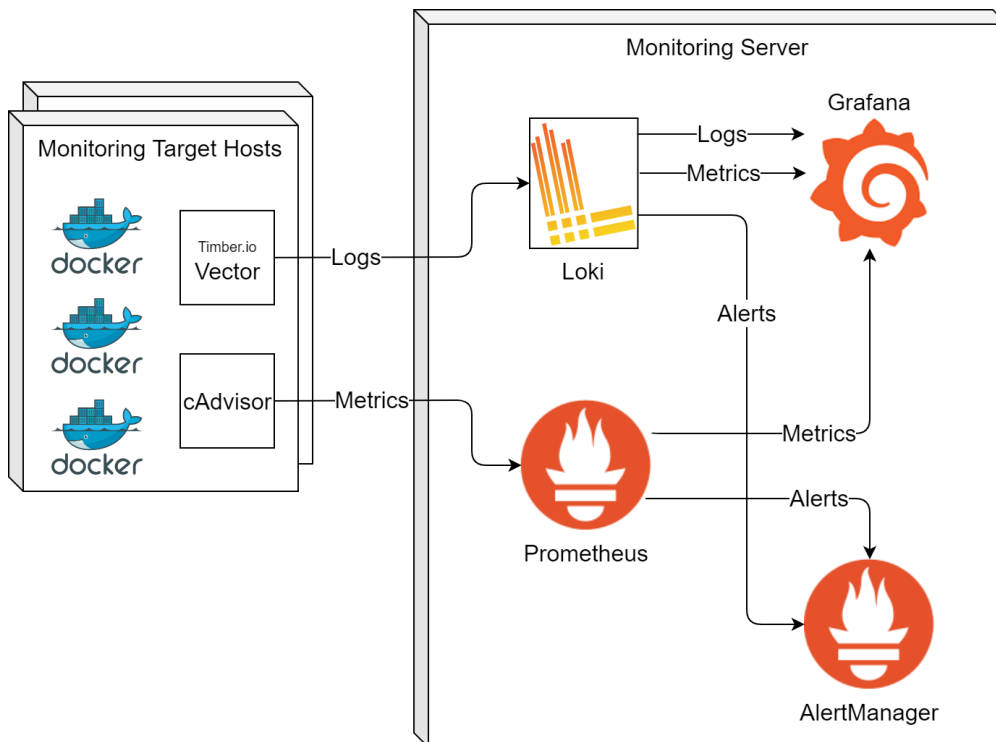


Figure 258. High-level Architecture of the Monitoring & Alerting Service

Grafana supports the creation of dashboards for monitoring resource consumption, as shown in Figure 259, and availability of services, as shown in Figure 260.



Figure 259. Monitoring of Resource Consumption using the Monitoring & Alerting Service

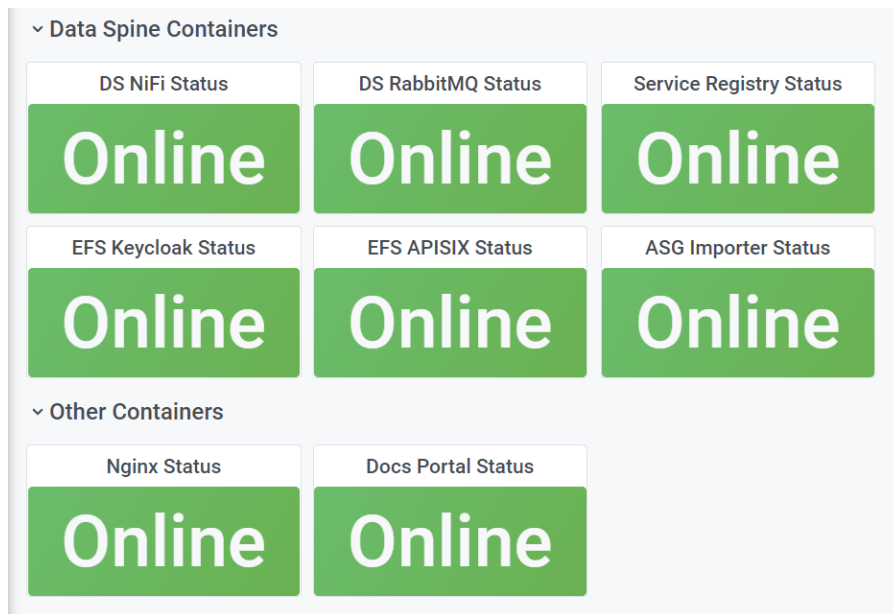


Figure 260. Monitoring of Service Availability using the Monitoring & Alerting Service

### 3.2.2 Interfaces for Platforms in the EFPF Ecosystem

#### 3.2.2.1 COMPOSITION

This Matchmaking API has migrated from COMPOSITION project and been integrated to the EFPF common platform. The API provides two main functionalities to the EFPF platform:



1. Services that provide all the information about companies/services coming from COMPOSITION project in order to be used for indexing and federated search mechanism in EFPF level
2. Services that enable the matchmaking of companies' agents and offers' evaluation in online bidding process of EFPF which is a core service especially for the Circular Economy pilot

The integration has been performed by indexing all information with Apache NiFi and Solr. The COMPOSITION ontology instances has been indexed by the Apache Solr, in order to ensure that all required information about services and products can be available to EFPF platform. Furthermore, the Semantic Matchmaker's services are accessible through Apache NiFi connecting to the RESTful API for multi-level matchmaking and online bidding.

The Matchmaker API is an application for automated online bidding through agent-level and offer-level matchmaking. It is an Ontology based framework which applies semantic rules and SPARQL queries to the dedicated Ontology for requesters and suppliers straightforward matching and implements weighted criteria assessment for offer evaluation and best offer suggestion. The Matchmaker is connected with the Marketplace agents and stakeholders through RESTful web services and HTTP protocol.

The basic concepts of the Matchmaker Ontology are: Business Entity (Company), Service/Product, Operations (Generic Activity Sectors) and provided Goods. Figure 261 contains the web services catalog of the Matchmaker API. GET services are available for retrieving the Ontology information, whereas POST web services are used in order to insert new information and start negotiations towards matchmaking and online bidding. The services specification is presented in Annex C of D3.11: EFPF Data Spine Realisation - I.

REST Endpoint	HTTP Method	Description
/getInfoFromOntology	GET	Retrieves all Marketplace Companies, Services and Products with the corresponding information
/getMarketplaceCompanies	GET	Retrieves Marketplace Companies with the corresponding information
/getMarketplaceServices	GET	Retrieves Marketplace Services and Products with the corresponding information
/getServicesFromCompany	GET	Retrieves Marketplace Services and Products of a specific Company with the corresponding information
/getCompanyDetails	GET	Retrieves specific Company's information
/getGoodsByCategory	GET	Retrieves the Marketplace Goods for each Service Category
/deleteCompany	GET	Deletes a specific Company from Marketplace
/performMatchmaking	POST	Performs agent level matchmaking
/offersEvaluation	POST	Performs offer level matchmaking
/setMarketplaceCompany	POST	Inserts new Company in Marketplace
/setMarketplaceService	POST	Inserts new Service in Marketplace

Figure 261. Matchmaker RESTful services catalog

Detailed descriptions of the matchmaking mechanisms are available on D5.11: EFPF Matchmaking and Intelligence Gathering. The bidding process interfaces are documented on D5.13: EFPF Interfacing, Evolution and Extension.

### 3.2.2.2 NIMBLE

NIMBLE is a federated, multi-sided, and cloud services-based business ecosystem that supports:

- B2B collaboration for industry, manufacturers, business, and logistics
- Technology-based innovation of products and evolution of traditional business models
- Federated, competitive yet interoperable instances of the platform

NIMBLE platform aims to achieve the following objectives:

- Create a platform ecosystem to attract early adopters: providers, vendors, buyers, collaborating using federated platform instances
- Ensure ease of entry and initial ease of use with quick rewards
- Grow platform usage by showing the benefits and by adding services where the need arises (release early, release often)
- Master the usage of the platform step-by-step to evolve business cooperation
- From the earliest steps to master-level, ensure trust, security, and privacy

NIMBLE platform was developed on Microservices architecture on Java Spring Boot framework. The following table presented in Figure 262 encompasses the main services, their descriptions and the API documentation links.

Service registry: <https://efpf-nimble.salzburgresearch.at/api/routes>

Service Name	Description	Swagger Documentation URL
<b>Identity Service</b>	Service for managing identities on the NIMBLE platform. Working as a middleware for Keycloak IS Server. Providing SSO capability to the whole platform. With the support of OAuth 2 + OpenID Connect protocols. All the Users and Companies that are registered to the platform are persisted in the Identity Service Persistence Layer. Inbuilt support to add and manage employees or members of a company with appropriate roles.	<a href="https://efpf-nimble.salzburgresearch.at/api/identity/swagger-ui.html">https://efpf-nimble.salzburgresearch.at/api/identity/swagger-ui.html</a>
<b>Catalog Service</b>	Manage Catalogues and catalogue-lines (products) for companies on the platform. All the related binary-content are persisted in the catalogue-persistence layer as well.	<a href="https://efpf-nimble.salzburgresearch.at/api/catalog/swagger-ui.html/">https://efpf-nimble.salzburgresearch.at/api/catalog/swagger-ui.html/</a>
<b>Business Process Service</b>	Handles the business workflow and persists related information of the platform. Using Camunda as the workflow designing tool and have native business workflow defined for NIMBLE-platform.	<a href="https://efpf-nimble.salzburgresearch.at/api/business-process/swagger-ui.html/">https://efpf-nimble.salzburgresearch.at/api/business-process/swagger-ui.html/</a>

	Manage and persist all the digital agreements between parties in the platform. Provide the backend functionality for NIMBLE-shopping carts as well as expose the ability to group negotiations as projects.	
<b>Indexing Service</b>	Service works as the middle layer to index products, companies also properties and classes of the used ontologies in Apache Solr.	<a href="https://efpf-nimble.salzburgresearch.at/api/index/swagger-ui.html/">https://efpf-nimble.salzburgresearch.at/api/index/swagger-ui.html/</a>
<b>Trust Service</b>	Microservice which calculates and manages the trust profiles of companies. Have a predefined set of trust attributes which can be extended according to requirement.	<a href="https://efpf-nimble.salzburgresearch.at/api/trust/swagger-ui.html/">https://efpf-nimble.salzburgresearch.at/api/trust/swagger-ui.html/</a>
<b>Data-Aggregation Service</b>	Working as the layer to aggregate data. Have the ability to aggregate data on company as well as platform levels which gives the ability to expose data based on the role that clients try to retrieve data.	<a href="https://efpf-nimble.salzburgresearch.at/api/data-aggregation/swagger-ui.html/">https://efpf-nimble.salzburgresearch.at/api/data-aggregation/swagger-ui.html/</a>

Figure 262. NIMBLE Services

### 3.2.2.3 DIGICOR (SMECluster)

This section describes the SMECluster platform, which is an instance of the DIGICOR platform utilising and hosting many of the SMECluster services. SMECluster provides Tools and Services via a marketplace available to its members and is enabled for interoperability with DIGICOR tools and services, demonstrating how DIGICOR tools and services can gain additional market exposure. The business model of SMECluster is to offer opportunities to collaborate between members and to support this goal through readily available technology that will provide productivity and quality improvements.

SMECluster provides two different API types, a CompanyDirectoryAPI (Figure 263) and a MarketplaceAPI (Figure 264).

REST Endpoint ( <a href="http://smecluster.com/api/DirectoryWebService">smecluster.com/api/DirectoryWebService</a> )	HTTP Method	Description
<b>/GetCompany?companyID={id}</b>	GET	Gets a company based upon the ID provided in the URL
<b>/GetCategory?categoryID={id}</b>	GET	Gets a category based upon the ID provided in the URL
<b>/GetAllCompanies</b>	GET	Gets a simplified version of all Companies contained in the database
<b>/GetAllCompaniesFull</b>	GET	Gets a fully detailed version of all Companies contained in the database
<b>/GetAllCategories</b>	GET	Gets all Categories contained in the database
<b>/GetAllCapabilities</b>	Get	Gets all Capabilities contained in the database

Figure 263. DIGICOR CompanyDirectoryAPI Endpoints

GetCompany returns a Company object, this consists of:

Addresses: A list of addresses associated with the company

Capabilities: A list of capabilities associated with the company

Categories: A list of categories associated with the company

**CompanyGuid:** A unique Guid assigned to the company  
**CompanyID:** A unique ID assigned to the company  
**Contacts:** A list of contacts associated with the company  
**CreatedBy:** A string showing who created the Company in the database  
**CreatedOn:** A date-time string showing when the Company was created in the database  
**Description:** A description of the Company  
**InterestedInBusiness:** A Boolean value to indicate whether the Company is interested in business or not  
**IsApproved:** A Boolean value to indicate whether the Company has been approved or not  
**Logo:** A URL to the company logo  
**ModifiedBy:** A string showing who most recently modified the Company (if modified from original insert)  
**ModifiedOn:** A date-time string showing when the Company was most recently modified in the database (if modified from original insert)  
**Name:** Name of the Company  
**NumberOfEmployees:** An Integer to show how many employees are in the Company  
**Website:** A URL to the Company website  
**YearsTrading:** An Integer to show how long the Company has been trading

**GetCategory** returns a Category object, this consists of:

**CategoryGuid:** A unique Guid assigned to the Category  
**CategoryID:** A unique ID assigned to the Category  
**Description:** A description of the Category  
**Name:** Name of the Category

**GetAllCompanies** returns a list of Company objects, this consists of:

The same as **GetCompany**, except the **Attributes**, **Capabilities** and **Categories** are not returned.

**GetAllCompaniesFull** returns a list of Company objects, this consists of:

The same as **GetCompany**, this is returned for every company in the database.

**GetAllCapabilities** returns a list of Capability objects, this consists of:

**CapabilityGuid:** A unique Guid assigned to the Capability  
**CapabilityID:** A unique ID assigned to the Capability  
**Description:** A description of the Capability  
**Name:** Name of the Capability

REST Endpoint ( <a href="http://smecluster.com/api/CatalogUtilsWebService">smecluster.com/api/CatalogUtilsWebService</a> )	HTTP Method	Description
<b>/GetAllProducts</b>	GET	Gets all Products contained in the database
<b>/Search?searchTerm={searchTerm}</b>	GET	Gets a Product/Products based upon the search term provided; the search term must contain a Product name e.g., Industreweb

Figure 264. DIGICOR MarketplaceAPI Endpoints

GetAllProducts returns a list of SearchModel objects, this consists of:  
 CustomValues: Dictionary of KeyValue pairs defined in the web.config e.g., publisher  
 Description: A description of the product  
 ImageURL: A URL to the product image on SMECluster  
 Name: Name of the product  
 Price: String representation of the product price  
 URL: A URL to the product page on SMECluster

Search returns either a singular or a list of SearchModel objects, dependent upon the search term provided. It contains the same fields as the GetAllProducts api.

The SMECluster platform architecture is based on a federation of service libraries orchestrated via calls from the integrated workflow engine and the client web UI, as illustrated in Figure 265. Whilst technology agnostic, the main stack runs on Microsoft .Net infrastructure under IIS, currently hosted on the SMECluster dedicated server but can equally be hosted on a cloud infrastructure. Data storage is provided by Microsoft SQL Server.

Figure 265 shows the interaction between services within the SMECluster platform, and those from the DIGICOR platform.

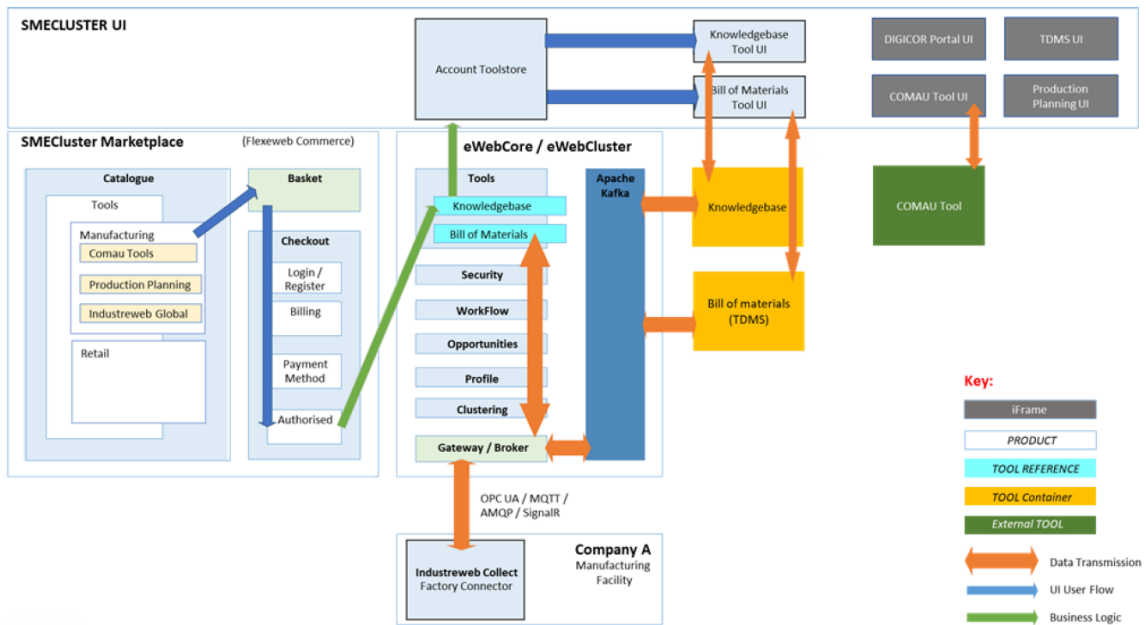


Figure 265. SMECluster Component Interaction

### 3.2.2.4 vf-OS

vf-OS was conceived as a platform to develop an Open Operating System for Virtual Factories. The platform comprises various components and elements as described in Figure 266 which have many purposes, from creating a virtualisation of the factory to providing enabler services for accessing the factory physical and digital assets (done via device drivers also developed in the project), to developing applications that take advantage of all the richness of assets made available to create value to the manufacturing business.

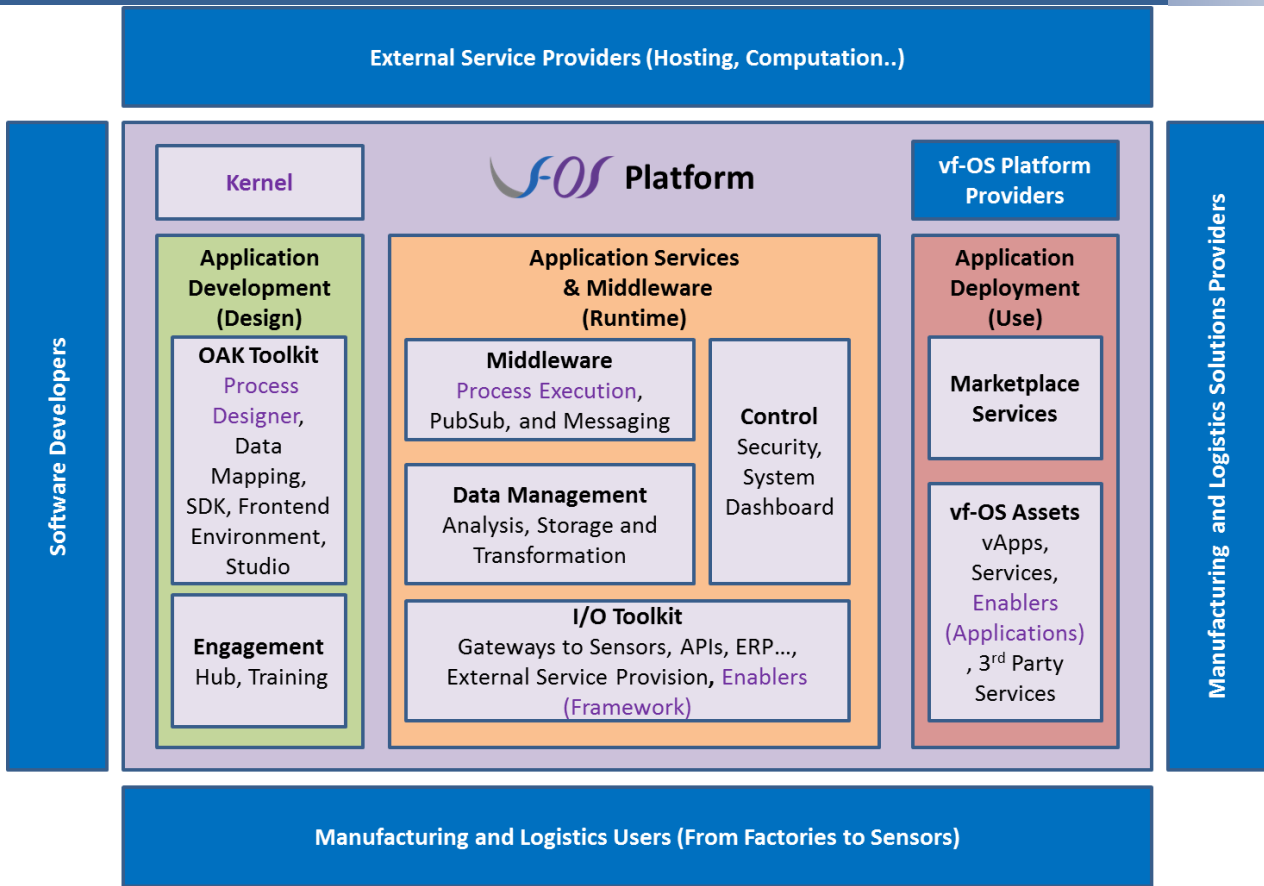


Figure 266. The vf-OS Architecture

The purpose of the project was to mimic an actual operating system but targeted to the virtual factories and its business, therefore a large number of components were developed, each with their documentation and APIs, which are not being adapted and made available in the EFPF ecosystem - as can be seen in Figure 267.

Service Name	Description	Documentation URL
<b>vf-OS Platform</b>	Overall environment, platform, and kernel	<ul style="list-style-type: none"> <li>vf-Platform: <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-platform-environment-vf-p">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-platform-environment-vf-p</a></li> </ul>
<b>Application Development (vf-OAK Toolkit)</b>	Tools for developers to develop vf-OS assets, such as vApps, Enablers, etc.	<ul style="list-style-type: none"> <li>vf-OAK Toolkit: <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/open-applications-development-toolkit-vf-oak">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/open-applications-development-toolkit-vf-oak</a></li> </ul>
<b>Application Services &amp; Middleware</b>	Set of enablers that are able to receive stimulus and actuate on the factory elements or that virtualise the factory	<ul style="list-style-type: none"> <li>Process Designer: <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/middleware/process-enabler---runtime">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/middleware/process-enabler---runtime</a></li> <li>Messaging and Publish/Subscribe: <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/middleware/publish-subscribe">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/middleware/publish-subscribe</a></li> <li>Data Storage: <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/data-management/data-storage">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/data-management/data-storage</a></li> <li>Data Analytics: <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/data-management/data-analytics">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/data-management/data-analytics</a></li> </ul>

		<ul style="list-style-type: none"> <li>• Enablers Framework (EF): <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/io-toolkit/enablers-framework">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/io-toolkit/enablers-framework</a></li> <li>• API Connectors: <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/io-toolkit/api-connectors">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/io-toolkit/api-connectors</a></li> <li>• External Service Provision (ESP): <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-platform-environment-vf-p/-/tree/master/esp">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-platform-environment-vf-p/-/tree/master/esp</a></li> <li>• Identity Management (IDM): <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/control/security/identity-management">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/control/security/identity-management</a></li> <li>• Authorisation Policy Decision Point (PDP): <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/control/security/authorisation-policy-decision-point">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/control/security/authorisation-policy-decision-point</a></li> <li>• System Dashboard: <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-platform-environment-vf-p/-/tree/master/systemDashboard">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-platform-environment-vf-p/-/tree/master/systemDashboard</a></li> </ul>
<p><b>Application Deployment Services</b></p>	<p>Set of components that will be taken into consideration when the vf-OS environment is going to be in use</p>	<ul style="list-style-type: none"> <li>• Marketplace Services: vf-Store: <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/marketplace-vf-store">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/marketplace-vf-store</a></li> <li>• FIWARE Generic Enablers: <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-assets/fiware-generic-enablers">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-assets/fiware-generic-enablers</a></li> <li>• Manufacturing Enablers: <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-assets/fiware-manufacturing-enablers">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-assets/fiware-manufacturing-enablers</a></li> <li>• vf-OS Enablers: <a href="https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-assets/vf-os-enablers">https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-assets/vf-os-enablers</a></li> </ul>

Figure 267. vf-OS services

### 3.2.2.5 Network Portal

Valuechain’s Network Portal [VLC22] (previously known as ‘iQluster’) is a supply chain intelligence platform that is designed to facilitate supply chain visibility, easily share data, and engage the lower tier supplier by creating a digital community. Unlike most platforms which are ‘one to many’ in design, Network Portal leverages its ‘many to many’ platform model to engage every stakeholder, big and small. The idea is to return some benefit to every company that joins and share data/information with the network.

Valuechain’s Network Portal is integrated with EFPF matchmaking service, via federated search and team formation. As a data API provider, Network Portal provides APIs to allow its data to be indexed into EFPF’s federated search making its data to be available to users accessing EFPF. On top of that, as an intelligent networking platform that offers easy communication and cluster management, Network Portal offers the ability allowing EFPF users to form a team (also known as ‘network’ or ‘cluster’)

To access the APIs provided by Network Portal, the user needs to retrieve the appropriate API token from Network Portal environment. This can be found under 'My Account' section of the user profile page as shown in Figure 268. Get API Token from Network Portal.

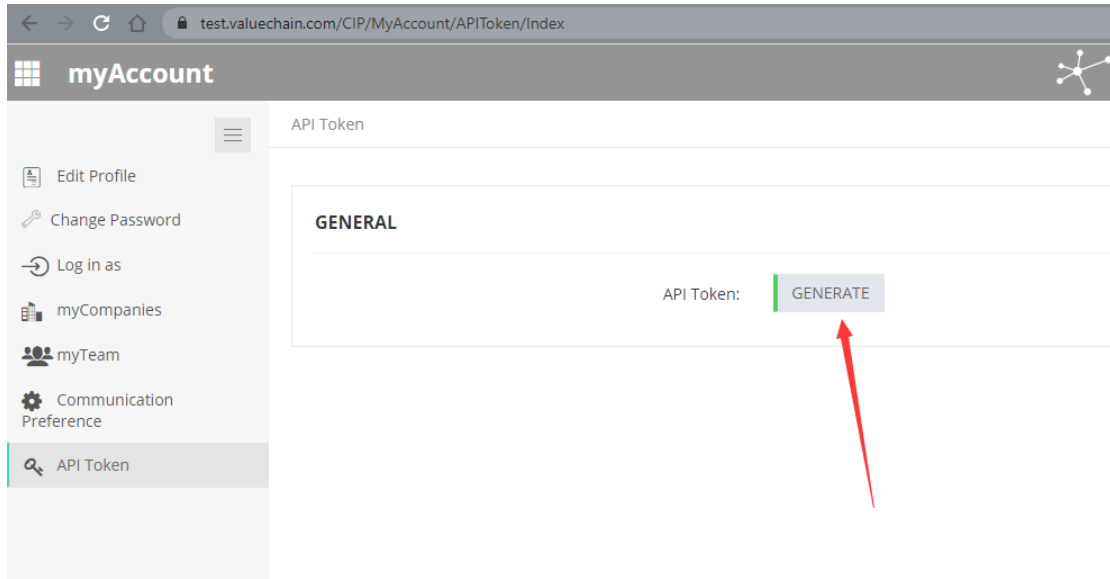


Figure 268. Get API Token from Network Portal

Figure 269 lists the APIs that have been developed to enable data sharing between Network platform and the EFPF matchmaking indexing workflow. The API structure is made up of multiple API calls that return specific information.

REST Endpoint (baseUrl: <a href="https://testapi.valuechain.com/api">https://testapi.valuechain.com/api</a> )	HTTP Method	Description
<b>/company/AllCompanies</b>	get	This is an API that allows you to retrieve all companies you have permission to retrieve. Then CompanyID can be used to retrieve company specific details.
<b>/company/generalinfo</b>	get	Get a company's general information
<b>/company/addresses</b>	get	Get a company's address
<b>/company/capabilities</b>	get	Get a company's capabilities
<b>/company/AllCompaniesFull</b>	get	This API allows you to retrieve all company information that Network Portal holds for the company. This covers additional information about a company such as contact person, which is used to facilitate the team formation feature (inviting a company to join a team). Some general fields are mentioned below as examples: <ul style="list-style-type: none"> <li>a. Company description (from company profile)</li> <li>b. Company status (active, dormant etc.)</li> <li>c. Incorporation date</li> <li>d. Website</li> <li>e. Social media handles...</li> </ul>
<b>/network/GetNetworks</b>	get	This is an API to get all networks that the company of current user is in, whether the company's role in the network is network leader or network member.  Note: Network Portal also provides API to allow EFPF user to initiate the team formation process from EFPF UI. To support this, in the backend, /download/uploadcompanies endpoint



		(post method) was added to allow EFPF to pass details of selected companies, once the data is posted successfully user will be navigated to the team formation process on Network Portal via SSO with required parameter to indicate it's a network/team creation operation (/Common/SSO?ls=efpf&dataid=data id&ps=create_network) . These networks then can be retrieved from EFP side to be displayed on the EFPF UI.
--	--	---

Figure 269. Network Portal API that Enables Data Sharing with the EFPF Matchmaking Indexing Workflow

### 3.2.2.6 Symphony

Symphony is a comprehensive cloud-based software suite of building management tools to create a complete Building Management System (BMS). The Symphony M2M platform can communicate with any automation controls, both standard protocols and proprietary systems. It allows the monitoring and control of diverse building automation systems, by integrating different protocols under a coordinated, unified management level with an open and modular approach.

Symphony control logics and system behaviors can be defined both at a local level (single system) as well as at a global level (multiple systems controlled by cloud-based services). Incorporation into the DVL allows communication with other M2M platforms and provides means of information transfer for use in the upper cross DLT layer for a secure exchange of information.

The generic development framework of Symphony can define reactions to events which are fed to potentially complex processing rules ranging from simple algorithms to more complex decision systems, which result in actions performed by the system.

Events are generated by objects (e.g., motion/presence detectors, open/close contacts), simple threshold comparators (e.g., lux sensor, temperature sensor) or processing rules themselves. Actions include specific operations on (groups of) objects, notifications, activation of scenarios, etc.

A simplified graphical interface can be used as an alternative to low-level programming languages (LUA, Python) which are also available.

An action scheduler allows programming functions with fine-grained timings or simply in a daily, weekly, monthly, or seasonal fashion.

The basic building blocks of the Symphony BMS are depicted in Figure 270. They include:

- Low level fieldbus controller (Hardware Abstraction Layer)
- Specialized functional modules:
  - Comfort automation system
  - Technical monitoring system
  - Energy manager
  - access control / anti-intrusion system
  - video surveillance system
  - communications

- network controller
- Event Reactor
- Multi-tier storage system
- Analytics & reporting engine
- Authentication and authorization manager
- Event reactor & logic engine to develop processes and workflows
- Cluster manager (high availability)
- Visualization App
- Cloud gateway
- Cloud reflector

These building blocks provide a complete functional stack covering an automation chain from the field bus level up to the user interface level.

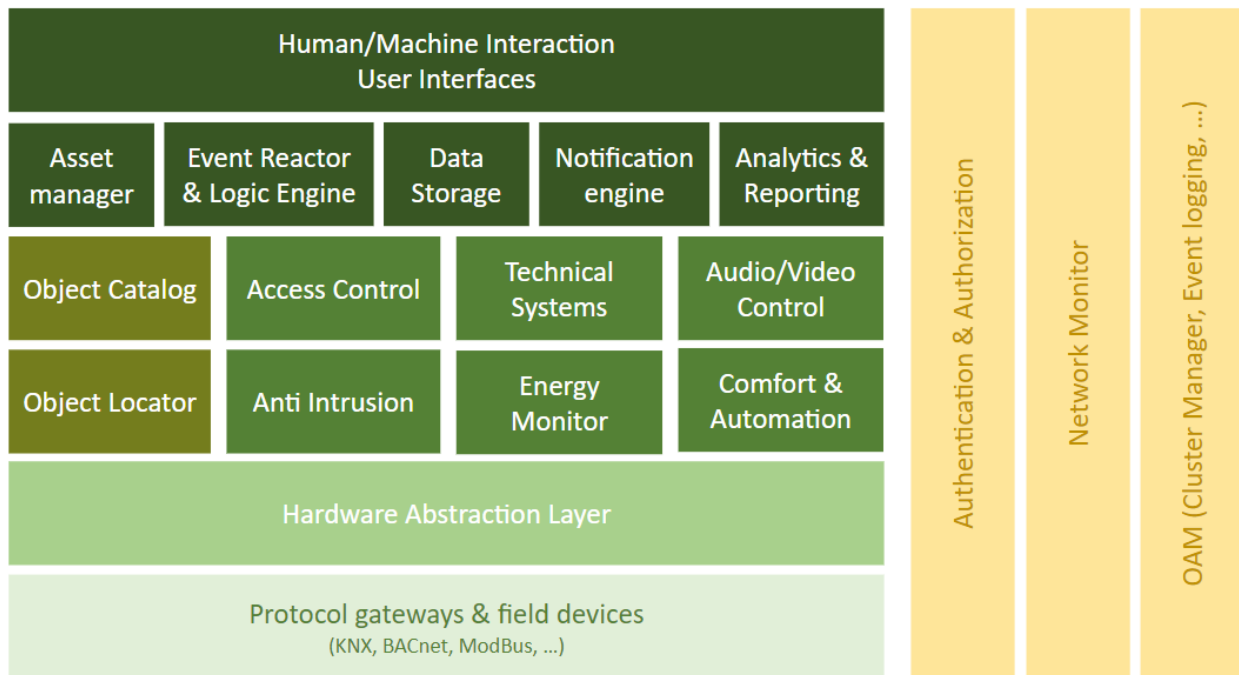


Figure 270. Basic Building Blocks of the Symphony BMS

Symphony exposes a GUI, as presented in Figure 271, to configure and monitor the status of the system, manage the deployments (including backups/restores and upgrades of the system).

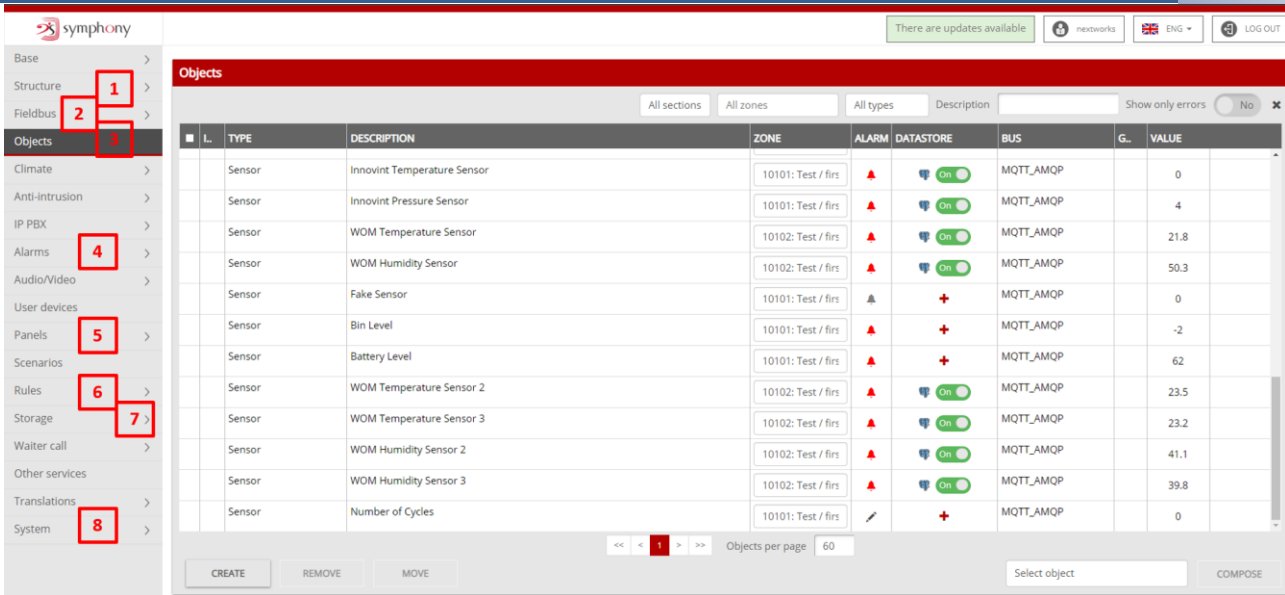


Figure 271. Symphony Configuration GUI

The Symphony configuration GUI is the main tool to configure and monitor the Symphony Building Management System (SBMS). It wraps most of the configuration APIs exposed by each of the software component to create an easy-to-use graphical system to customize the setup.

The GUI is composed by a left menu bar that is the main entrypoint to each Symphony component’s configurator, in detail:

- **1, Structure:** It enables the possibility to define logical building structures based on the real (or virtual) topology of the monitored building
- **2+3, Fieldbus & Objects:** Is the Symphony HAL’s configurator, Section 3.2.1.5.3 shows a detailed description of the wrapped APIs
- **4+6, Alarms & Rules:** Is the Symphony Event Reactor’s configurator, Section 3.2.1.20 shows a detailed description of the wrapped APIs
- **5, Panels:** It allows the user to configure the Symphony Visualization App, to graphically display the status of the object defined in the Building Management System
- **7, Storage:** Is the Symphony Data Storage’s configurator, Section 3.2.1.21 shows a detailed description of the wrapped APIs
- **8, System:** It offers a set of tools to manage the current Symphony deployment (Backup/Restore, Factory Reset, Upgrade etc.)

### 3.3 API Management

API management is the process that is concerned with creating, publishing, monitoring, and securing, and controlling access to the APIs. The EFPF ecosystem offers various tools for effectively managing the APIs of services.

#### Lifecycle Management of APIs

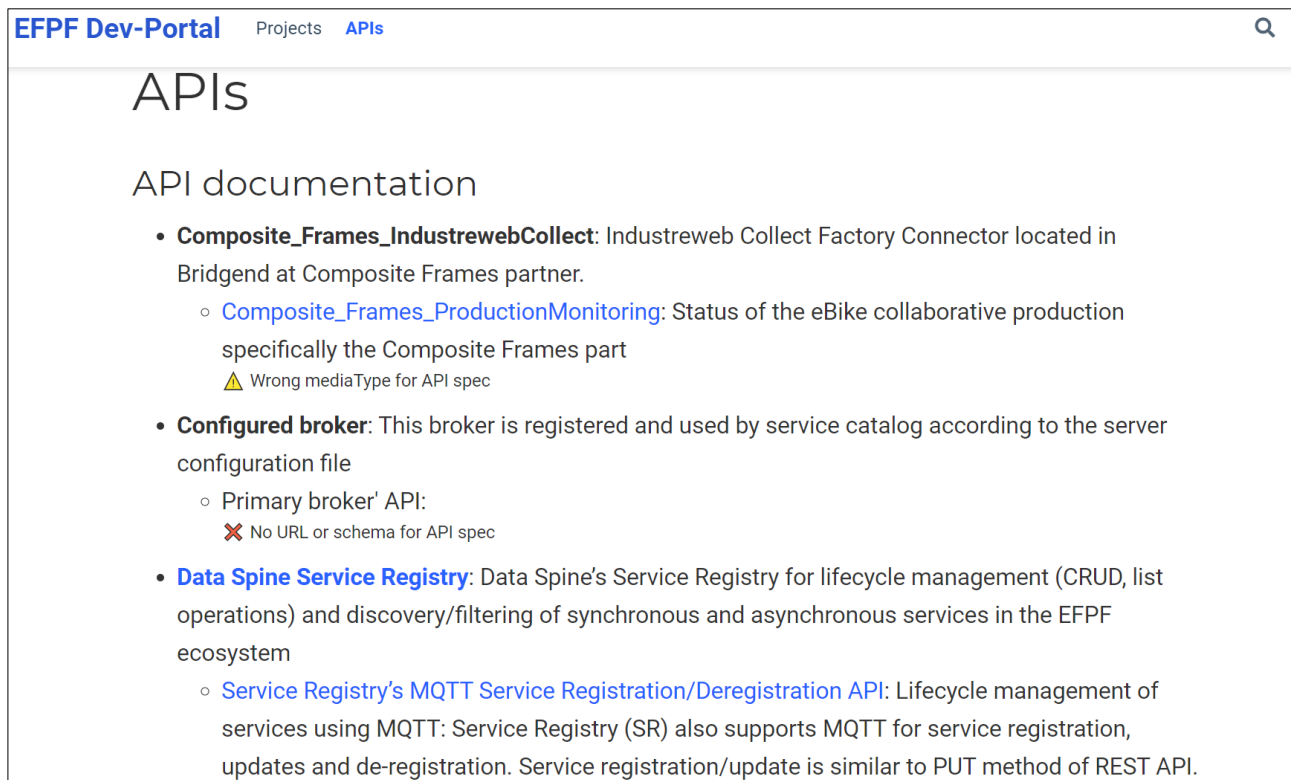
As discussed in Section 3.1, API specification standards are used to create API specification documents for services. These API specification documents need to be stored in a repository and should be discoverable and retrievable. The Service Registry (Sections

2.4.1.4 and 2.5.4) component of the Data Spine provides this functionality. It provides an API for lifecycle management of services. A service object can have many APIs and each API has an API spec that can be stored in the Service Registry. The Service Registry provides a functionality to browse through the services and their API specs and to retrieve metadata and the API spec for a particular service. It also provides a service filtering API that can be used to search for services and their APIs based on functional or technical metadata.

The Service Registration Tool provides a simple, easy to use GUI for the Service Registry. In addition, it also offers a functionality to verify the correctness of API specifications before registering them to the Service Registry.

### Visualization of APIs

A new microservice called ‘api-spec-retriever’ was developed that retrieves the APIs registered to the Service Registry and displays their summary onto the EFPF Dev-Portal as illustrated in Figure 272. The summary includes error or warning messages regarding incompleteness of the service registrations and also links to the visual representation of APIs.



The screenshot shows the EFPF Dev-Portal interface. At the top, there is a navigation bar with 'EFPF Dev-Portal' on the left, 'Projects' and 'APIs' in the center, and a search icon on the right. Below the navigation bar, the main content area is titled 'APIs' and 'API documentation'. It contains a list of APIs with their descriptions and associated error or warning messages:

- **Composite\_Frames\_IndustwebCollect:** Industweb Collect Factory Connector located in Bridgend at Composite Frames partner.
  - [Composite\\_Frames\\_ProductionMonitoring](#): Status of the eBike collaborative production specifically the Composite Frames part
    - ⚠ Wrong mediaType for API spec
- **Configured broker:** This broker is registered and used by service catalog according to the server configuration file
  - Primary broker' API:
    - ✖ No URL or schema for API spec
- **Data Spine Service Registry:** Data Spine's Service Registry for lifecycle management (CRUD, list operations) and discovery/filtering of synchronous and asynchronous services in the EFPF ecosystem
  - [Service Registry's MQTT Service Registration/Deregistration API](#): Lifecycle management of services using MQTT: Service Registry (SR) also supports MQTT for service registration, updates and de-registration. Service registration/update is similar to PUT method of REST API.

Figure 272. Visualization of APIs Registered in the Service Registry

The API specifications of synchronous request-response type of services can be visualised using an instance of Swagger UI that is embedded into the EFPF Dev-Portal, as illustrated in Figure 273, while the API specifications of asynchronous Pub/Sub type of services can be visualised using the AsyncAPI Playground, as illustrated in Figure 274.

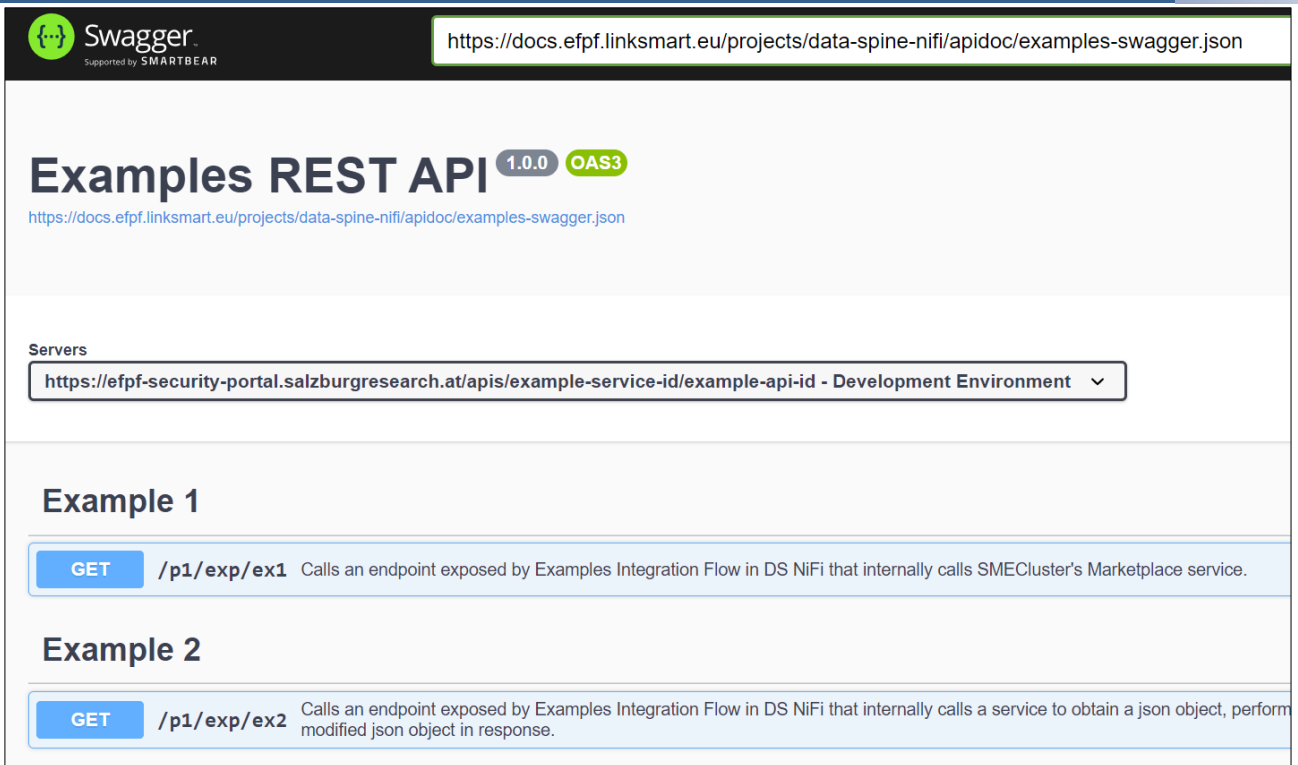


Figure 273. Visualization of Sync APIs using Swagger UI

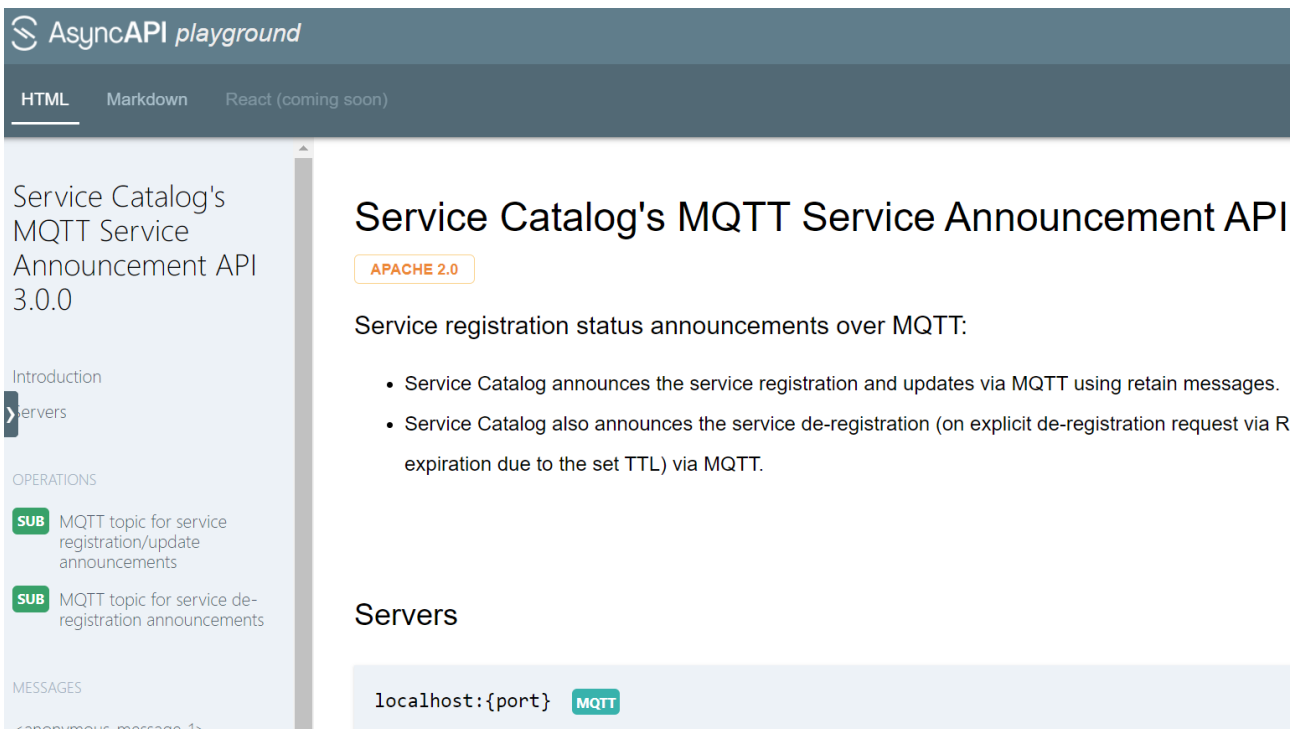


Figure 274. Visualization of Async APIs using AsyncAPI Playground

### Monitoring of APIs

The monitoring of APIs can be for multiple reasons: for detecting failures/unavailability of API endpoints, for collecting various usage statistics for APIs and analysing those to examine the performance, to generate reports, etc.

The Service Registry makes use of ‘heartbeat’ like mechanism to ensure availability and reachability of services. Every service has an associated ‘ttl’ (time-to-live) and the registered services are obliged to update themselves within the ttl timeframe. If a particular service is not updated within the set ttl, it is presumed dead and removed from the Service Registry. The minimum value for ttl can be 1 second and the maximum value is set to 2147483647, which same as the max value for a DNS packet ttl. The reasoning behind having the ttl field and a max value for it is twofold:

1. **Keepalive:** ttl serves as a keepalive mechanism to detect failures/unavailability of registered services. If a service fails to update itself within the ttl timeframe, it is concluded to be unavailable.
2. **Provider-consumer contract:** In a system if there is no tool in place to detect interface updates and to enforce interface-contracts, ttl also serves as the de facto contract mechanism between the service provider and consumers. So, the responsibility of checking updates to a service's interface is offloaded to the service consumers, who should retrieve the service from the Service Registry after every ttl time period to detect changes/updates, if any. This has been replaced in EFPF with the Interface Contracts Management Tool presented in Section 3.4.

### **Securing Access to API Endpoints**

The access to HTTP-based API endpoints in EFPF ecosystem is secured with the access policies defined in the EFS. To enforce the policies for the integration flow APIs, the API Security Gateway component of the Data Spine is used.

### **Access Consent Delegation for APIs**

In an ecosystem as large as EFPF, there are multiple data providers and consumers. It is desirable that the data providers have direct control over who is authorized to access their APIs. The Pub/Sub Security Service (Section 3.2.1.4) provides the access consent delegation functionality in the EFPF ecosystem. It automates the process of obtaining a Message Bus user account and getting the necessary permissions to publish or subscribe to the topics through an intuitive GUI. It removes the administrator from the loop and enables the topic publishers to approve or deny subscription requests.

## **3.4 Interface Contracts and Their Management**

This section introduces API contracts, the governance policies for API contracts and a new tool that track changes to APIs of services to ensure conformance with the defined policies. The interface contracts between EFPF and base platforms were specified in D3.1: EFPF Architecture-I.

### **3.4.1 Introduction**

APIs are the contracts between service providers and service consumers. APIs allow the service consumers to know the technical capabilities of a service and how to interface with it without needing access to source code. Therefore, in a federated platform ecosystem such as the EFPF ecosystem, the involved parties, i.e., the service providers and the service consumers rely on the agreed API Contracts or Interface Contracts for communicating with each other. However, as the participant services evolve, the upgradation of APIs becomes necessary and inevitable. Therefore, the federated platform ecosystem should define Interface Contract policies that allow the Service Providers to convey plans to deprecate/upgrade their APIs to the service consumers in advance allowing a smooth

transition/collaboration. The definition of such Interface Contract policies for EFPF can be found in D7.1: Planning, Operational Management and Technical Support for EFPF Platform-I. Furthermore, the EFPF ecosystem offers a tool called Interface Contracts Management Tool that helps the Service Consumers monitor the APIs they consume and helps them identify and any breaking changes to them.

### 3.4.2 Interface Contract Management Tool

The Interface Contract Management Tool (ICMT) is a custom component which is useful to track changes of the interfaces of the tools and services in the EFPF platform. This tool is independent from the Service Registry (SR), and it serves as an extension of it, helping developers keeping track of the software interfaces they make use of.

This tool communicates with the SR through the Message Bus using the MQTT protocol. When a service publishes an update to its interfaces to the SR, the SR publishes a message to the Message Bus. This message is read from the ICMT which then checks inside its internal storage whether it already has any information about that service. If not, it is stored, on the other hand if some information is found then the ICMT proceeds to compare the newly received information about its interfaces with the one stored. If changes to the version number of the tool or service are detected, as well as changes to the interfaces, the tool is marked as “updated”. The updates to the tools and services are displayed on the user interface.

On the main page (Figure 275) of the tool the services are displayed on a list and are sorted from most recently updated or added, to the least. Here the tools are marked with a dot which represents how much the latest update of the tool can impact the developers using its interfaces. The updates of the tool can be “patch”, “minor” or “major” in agreement with the Semver specifications. Tools and services can be “New” in case they have been not updated since they have been added or “unknown” if the developer of the tool did not provide enough information for the tool to understand the version number or the type of update to the tool or service.

#	Tool	Version	Update type	Last Updated ↓
1	ExtraCash API [ExtraCash API]	None	● Unknown	2022/4/13
2	EFPF Accountancy Service [Logstash endpoint for data ingestion and pipeline]	None	● Unknown	2022/4/7
3	Nimble products in the marketplace [Select Item API]	None	● Unknown	2022/4/5
4	SME Cluster APIs [SMECluster GetAllCompaniesFull]	None	● Unknown	2022/4/5
5	SME Cluster APIs [SMECluster GetAllCompanies]	None	● Unknown	2022/4/5
6	SME Cluster APIs [SMECluster GetAllProducts]	None	● Unknown	2022/4/5
7	vf-OS Services [vf-OS products]	None	● Unknown	2022/4/5
8	WASP Services [Marketplace services]	None	● Unknown	2022/4/5

Figure 275. Interface Contract Management Tool GUI: Services’ List View

If a developer wants to have more detail about a tool or service and about how its latest update affects the API, he/she is consuming then a detailed view is available, as shown in Figure 276. In this view the developer can see details about the tool or service collected from the data fetched from the SR. The services and tools providing the OpenAPI 3.0 and above compatible specifications for their API also benefit from this by having a detailed changelog of the latest update displayed in this view.

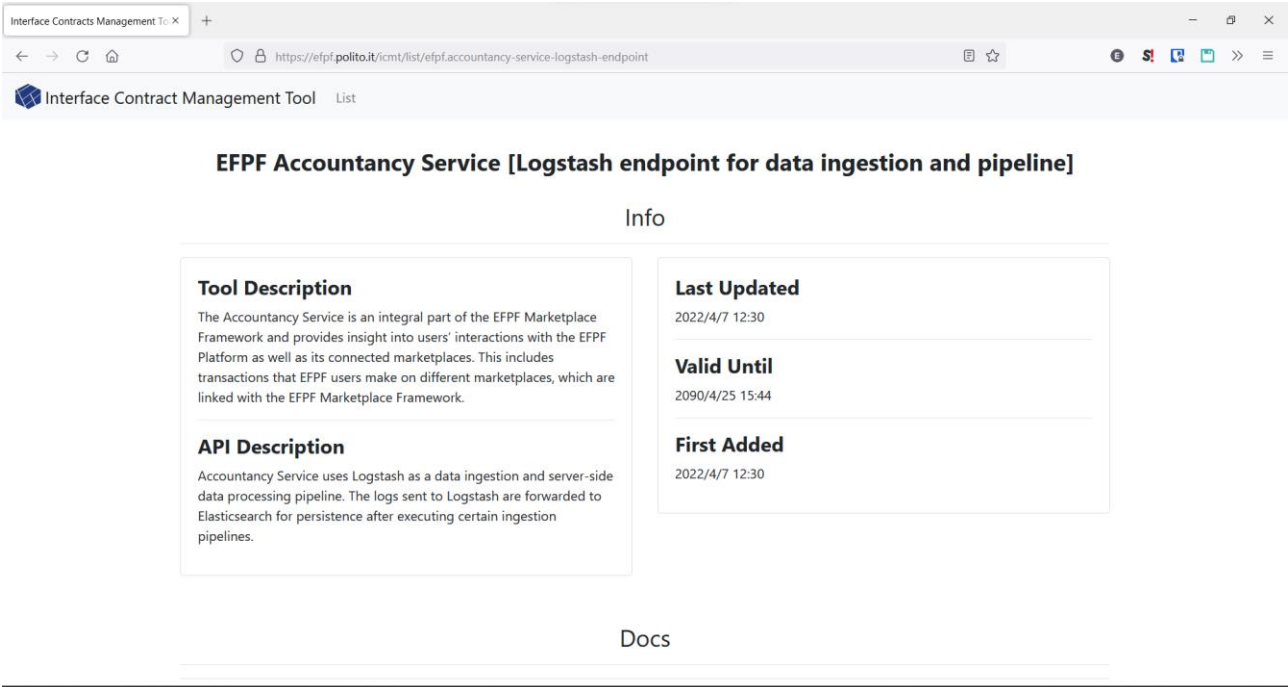


Figure 276. Interface Contract Management Tool GUI: Service Details View



## 4 Data Model Interoperability Layer

### 4.1 Introduction

It has already been stated that the objective of the data model interoperability layer is supporting information exchange and business processes that spread across two or more EFPF platforms and tools. This happens through:

1. A standardization effort in the data modelling topic which enables different tools and services to be able to exchange information.
2. Use of policies to promote standards adoption to reduce interoperability issues.
3. Development of tools to solve data model incompatibilities.

In this section we will address the updates that have been implemented since the previous version of this document. This work has been carried out by analysing the implementation of proposed interoperability workflows and how these workflows were implemented in terms of tools and of which data models have been made interoperable by the users of the EFPF platform. Pilots and open call projects have been chosen for analysis since those represent better, differently from NiFi workflows developed for integrating the base platform components, whether developers which have not been directly involved with the selection of the technologies and best practices for the Data Spine can use those technologies effectively or if there are some gaps.

### 4.2 Final Pilot scenarios analysis

In this section an analysis is carried out about pilot projects data model interoperability workflows. The pilots are introduced by a brief description to bring context. Their architectural diagrams if unchanged are available in the previous version of this document, Deliverable 3.11: EFPF Data Spine Realisation - I. These follow-up analyses have been performed to see if and how the proposed integration workflows described in D3.11 and the data model transformations (if present) have evolved.

#### 4.2.1 Working Environment Monitoring

The purpose of the working environment monitoring scenario is to monitor in (soft) real-time a working environment using IoT devices.

The working environment could be an office building composed of offices, a manufacturing shop floor, or a warehouse where materials and parts are stored. Various aspect can be monitored including air quality, energy consumption, space utilization, safety, and comfort. In the use case described above, HAL connector, TSMATCH gateway/application, and the event reactor service are using different models, which makes integration process difficult and time consuming.

To make the data coming from TSMATCH gateway interoperable the following JOLT spec has been developed. The purpose of this JOLT spec is to make this data interoperable with the data model required by the Nextworks Event Reactor model which provides notifications to the users according to values received from the monitoring sensors.

```

Jolt Spec
[
  {
    "operation": "shift",
    "spec": {
      "results": "result",
      "sensorIds": { "*" : "oid" }
    }
  }
]

```

Figure 277. Jolt Spec for Working Environment Monitoring Scenario

#### 4.2.2 Production Optimisation Pilot

In the Production Optimisation Pilot factory connectivity solutions are used to improve the efficiency of an edge banding machine by displaying clear instructions to the operator to avoid mistakes being made and to predict when preventative maintenance will be required.

For the analog measurements provided by the various (temperature, current, etc.) sensors mounted on the actual machine, the Industweb Factory connection offers a MQTT digital interface. These results can be thought of as machine KPI information. Then, the tools that employ those from the MQTT broker are used to fetch this data.

The messages received on NiFi contain the readings of the analog measurements. In order to make them human readable those have been processed using the JOLT spec below. The steps which are performed on all the readings collected from the factory connector are to first convert the string values to integers, then to appropriately scale those integers to the values of the standard units of measurement and then to cast back these integers to strings in order to maintain compatibility with the proprietary data model defined by the Industweb Factory Connector.

```

Jolt Spec
[
  // current    conversion: cast to int, subtract 4000, divide by 800, cast to string
  // temperature conversion: cast to int, subtract 4000, divide by 160, cast to string
  // pressure   conversion: cast to int, subtract 4000, divide by 1600, cast to string
  {
    "operation": "modify-overwrite-beta",
    "spec": {
      "payload": {
        "current": {
          "*": {
            "values": {
              "*": {
                "value": "=toInteger"
              }
            }
          }
        }
      }
    }
  }
]

```

```

    }
  }
},
"temperature": {
  "*": {
    "value": "=toInteger"
  }
},
"pressure": {
  "*": {
    "values": {
      "*": {
        "value": "=toInteger"
      }
    }
  }
}
}
},
{
  "operation": "modify-overwrite-beta",
  "spec": {
    "payload": {
      "current": {
        "*": {
          "values": {
            "*": {
              "value": "=intSubtract(@(1,value), 4000)"
            }
          }
        }
      }
    },
    "temperature": {
      "*": {
        "value": "=intSubtract(@(1,value), 4000)"
      }
    },
    "pressure": {
      "*": {
        "values": {
          "*": {

```

```

        "value": "=intSubtract(@(1,value), 4000)"
    }
}
}
}
}
},
{
    "operation": "modify-overwrite-beta",
    "spec": {
        "payload": {
            "current": {
                "*": {
                    "values": {
                        "*": {
                            "value": "=divide(@(1,value), 800)"
                        }
                    }
                }
            },
            "temperature": {
                "*": {
                    "value": "=divide(@(1,value), 160)"
                }
            },
            "pressure": {
                "*": {
                    "values": {
                        "*": {
                            "value": "=divide(@(1,value), 1600)"
                        }
                    }
                }
            }
        }
    },
    {
        "operation": "modify-overwrite-beta",
        "spec": {
            "payload": {

```

```

    "current": {
      "*": {
        "values": {
          "*": {
            "value": "=toString"
          }
        }
      }
    },
    "temperature": {
      "*": {
        "value": "=toString"
      }
    },
    "pressure": {
      "*": {
        "values": {
          "*": {
            "value": "=toString"
          }
        }
      }
    }
  },
  {
    "operation": "modify-overwrite-beta",
    "spec": {
      "payload": {
        "temperature": {
          "motor5": {
            "value": "20"
          }
        }
      }
    }
  },
  {
    "operation": "modify-overwrite-beta",
    "spec": {
      "payload": {

```

```
    "temperature": {  
      "motor6": {  
        "value": "20"  
      }  
    }  
  }  
}  
]  
]
```

Figure 278. Jolt Spec for Production Optimisation Pilot Scenario

Once transformed the data is pushed again on a different MQTT topic where is consumed by different tools such as the Visual Analytics Solution, the Risk Tool, and the Anomaly Detection Solution. This part of the workflow is described in more detail in Section 1.4.4.

Another transformation of the data already scaled on NiFi happens on a dedicated microservice outside of the Data Spine. Once transformed to the OGC-Sensor Things data model the data is pushed again to the MQTT broker from where is then consumed by the Deep Learning Toolkit.

### 4.2.3 Bins' Fill Level Monitoring

This use case focuses on the detection of bin and container fill levels and the calculation of the optimal route for collecting shop-floor bins.

Sensors provide early (real-time) notification of the recyclable and scrap bins fill levels and suggest optimal routes for collecting bins within the factory. An IR distance sensor should be chosen for the indoor bins, mostly because weight sensing would require a complex mechanical solution. Low power friendliness of sensors has also to be taken into account, therefore LoRa LPWAN is the most suitable solution due to its "lightweightness" and low power needs.

Sensors measure distance measurements from their position to the surface of the waste over a fixed time period. This data is then processed and translated to fill percentage and sent to the platform. Fill percentage is displayed in real time to scrap collectors or waste management staff in order to plan the collection process.

To implement this process a NiFi workflow has been developed. The main challenge was to separate the information about the bin's fill level from the information relative to the board with the Lora communication module status. To do so the aggregated data coming from the board and transmitted over MQTT has been collected on NiFi. The second step has been to implement two different JOLT transformations to the data received.

Both JOLT specs extract different information from the LORA boards payloads, the first one extracts the battery level of the board, the second one extracts the bin's fill level percentage. An ID is added to the newly created messages.

These messages are eventually sent over MQTT to be consumed by the end-user application frontend.

Jolt Spec
<pre>[   {     "operation": "shift",     "spec": {       "request": {         "measurements": {           "battery": "result"         }       }     }   },   {     "operation": "default",     "spec": {       "oid": "7efac212-fa26-4319-83b1-6c9ebcb5378d"     }   } ]</pre>
<pre>[   {     "operation": "shift",     "spec": {       "request": {         "measurements": {           "FillPercentage": "result"         }       }     }   },   {     "operation": "default",     "spec": {       "oid": "aaf7b26e-c6eb-4fab-922c-15537b13c2ce"     }   } ]</pre>

Figure 279. Jolt Spec for Bins' Fill Level Monitoring Scenario

#### 4.2.4 Matchmaking

The EFPF platform's matchmaker has been developed specifically from the platform and implements the EFPF Manufacturing Ontology (EFONT) standard interoperable data model to define the metadata related to the suppliers and their products and services from different base platforms in EFPF.

The Matchmaker interacts directly with other components such as the Marketplace and its agents and it bridges those to the other base platforms (Nimble, Network Portal (previously known as 'iQluster'), SMECluster & Composition) NiFi has then been used for implementing routing workflows that transformed the data transmitted through the platform to make it interoperable with the EFPF standard.

Matchmaking integration flows contains the data indexing/re-indexing workflows from connected base platforms. Different base platforms have different data sources and their data models in their respective base platforms are different. These heterogeneous data models need to be transformed and indexed into the common data model that is used in EFPF federated search index (Apache Solr based). There are 2 types of data that are typically retrieved from a base platform: company data and products data. These 2 types of data are indexed into 2 Solr collections in EFPF federated index (party, item).

In summary, the matchmaking iFlows contain the extract-transform-load (ETL) workflows for each individual base platform data to the EFPF federated search index.

An example JOLT transformation for the integration of the SMECluster platform with the EFPF Matchmaker is provided below.

Jolt Spec
<pre>[   {     "operation": "modify-default-beta",     "spec": {       "*": {         "ID": "=concat('smecluster_opp', @(1,OpportunityID))"       }     }   },   {     "operation": "shift",     "spec": {       "*": {         "Name": ["@(1,ID).label.en"],         "Description": "@(1,ID).description.en",         "CompanyID": "@(1,ID).basePlatformCompanyId",         "Website": "@(1,ID).website",         "ID": "@(1,ID).id",         "Value": "@(1,ID).customDoubleValues.value[]",         "Quantity": "@(1,ID).customStringValue.quantity[]",         "QuantityShow": "@(1,ID).customBooleanValue.quantityShow",</pre>



```

    "Currency": "@(1,ID).customStringValue.currency[]",
    "Location": "@(1,ID).customStringValue.location[]",
    "LocationShow": "@(1,ID).customBooleanValue.locationShow",
    "DurationDate": "@(1,ID).customStringValue.durationDate[]",
    "LeadDate": "@(1,ID).customStringValue.leadDate[]",
    "OpeningDate": "@(1,ID).customStringValue.openingDate[]",
    "ClosingDate": "@(1,ID).customStringValue.closingDate[]",
    "OpportunityCategory": {
      "CategoryGuid": ["@(2,ID).classifications[]",
"@(2,ID).classificationMap.@(1,CategoryGuid).uri"],
      "Name": ["@(2,ID).classificationMap.@(1,CategoryGuid).label.en"],
      "Description": "@(2,ID).classificationMap.@(1,CategoryGuid).description.en"
    }
  }
},
{
  "operation": "shift",
  "spec": {
    "*": {
      "@": ""
    }
  }
},
{
  "operation": "default",
  "spec": {
    "*": {
      "basePlatform": "smecluster"
    }
  }
}
]

```

Figure 280. Jolt Spec for the Integration of the SMECluster platform with the EFPF Matchmaker

Details of the EFONT data model are provided in the previous version of this document, Deliverable D3.11: EFPF Data Spine Realisation - I.

#### 4.2.5 Projects which did not implement interoperability workflows

Some projects such as Supply Chain Transparency and Blockchain and Smart Contracting didn't use NiFi to perform any data conversion or data model transformation. This has

happened for different reasons such as newly developed tools or the adoption of a particular standard across the whole project.

#### **4.2.5.1 Tendering and Bid Management**

Tender and Bid Management portal has been separated into three main components: a company directory, where businesses can choose to share their profile with a breakdown of what they offer and more; a business opportunity board, that includes tenders, small work opportunities and allows suppliers to submit applications; along with a messaging system. A range of different data is collected through this portal, from the business opportunities themselves to the personal data and company information from both suppliers and procurers registered.

The Tender and Bid Management platform does not follow any standards due to the bespoke nature of the information needed, and a lack of suitable standards that could be applied to these needs.

#### **4.2.5.2 Supply Chain Transparency**

The supply chain transparency pilot scenario aims to provide end-to end transparency for production process monitoring in a supply chain. The objective of the pilot has been to provide live visualization at a glance and end to end visibility of the production process. The system can also send status updates, e.g., orders and warnings if a problem is detected at some step.

The information needed to model the processes can be found in the Enterprise Resource Planning of the companies using the service and it needs to be published to the message broker to be used from Workflow and Service Automation Platform (WASP) tool. Translators on the Integration Flow Engine can ensure interoperability between different data models; however, no interoperability has been developed since such need has not arisen during the course of the pilot.

#### **4.2.5.3 Blockchain and Smart Contracting**

The Blockchain and smart contracting pilot deals with order management, transportation, and circular economy. The blockchain infrastructure itself it is not expected to expose any standards to the Data Spine, it will use domain standards to represent data and adopt the blockchain frameworks to store transactions using this data. The BiTAS data model and format has been used as expected in the blockchain shipping applications, and EPCIS in the order processing. There has been no need to make use of the Integration Flow Engine data model translators for Enterprise Resource Planning data in this case as well, to be used from WASP.

#### **4.2.6 Conclusion from Pilot scenarios analysis**

It has been interesting to compare the evolution of these projects with the analysis done in the first version of this document. Most of the pilots have respected the plans detailed in Deliverable 3.11 and the deviations from the plans have been of minor importance.

The interoperability workflows developed have tackled different issues, from data conversion to data model transformations. The workflows developed have different degrees of complexity and this proves that the platform has been capable of supporting even demanding workflows.

## 4.3 Open-Call Project Analysis

Although open call projects are still work in progress at the time of writing this document, some have already implemented some interesting integration workflows worth of being described on NiFi. In the sections below these projects have briefly described and their workflows documented.

### 4.3.1 Open Call LORTEK

LORTEK open call project aims to create an Intelligent Welding Quality Monitoring and Assessment System implemented as a shared service of the EFPF platform. The data-driven intelligent system will be focused on steel arc welding processes, specifically in the well established MAG (i.e., metal-inert gas) processes, which are used in several industrial sectors. Automated system will enable advanced welding process data visualization, main process data analysis and process modelling.

In order to implement this system LORTEK had to develop a solution to acquire data from different sources: external sensors, welding robot and welding power source and to push this data directly to data spine through the message bus from EFPF platform

Since the data acquisition platform acquired more data than effectively required LORTEK implemented a filter, as earlier projects did, using a JOLT processor. This allowed them to use only the data required and to reduce the burden on their infrastructure where the data has been processed and used for implementing their ideas and project.

```

Jolt Spec
[
  {
    "operation": "shift",
    "spec": {
      "Temperature": "Temperature",
      "GasFlux": "GasFlux",
      "SeamTrack": "SeamTrack",
      "Voltage": "Voltage",
      "WireFeedSpeed": "WireFeedSpeed",
      "Current_s": "Current_s",
      "WireSpeed_s": "WireSpeed_s",
      "Current": "Current",
      "datetime": "datetime"
    }
  }
]

```

Figure 281. Jolt Spec for the LORTEK Open Call Project

### 4.3.2 Open Call DNET

The purpose of DNET's project is to make heterogeneous data available for data analytics, predictive maintenance, and 3<sup>rd</sup> party use, we will combine data from several company processes using EFPF into a single product passport component.

The Metalac corporation, an end user in their project, is the owner of the data used. Vibration and other sensors have been integrated with EFPF (EFPF Data Spine, EFPF Blockchain

Network), followed by the related data from SCADA and controlling platform (Siemens WinCC, EMS and OEE): energy consumption, quantity/amount of produced parts, working and non-working regimes of machines and production lines.

DNET’s project intends to use EFPF predictive maintenance to link with the current vibration sensor data and produce assessment results to be used for malfunction prediction and anomaly identification. The predictive maintenance functionality will happen through the integration of the Deep Learning Toolkit for Predictive Maintenance.

The workflow built by DNET is described in detail in Section 1.4.5.

## 4.4 Data Model Interoperability Surveys

In order to understand how and if the tools to address the data model interoperability issues satisfied the developers, two different surveys have been written and launched to both the developers involved in the EFPF Pilot project and the EFPF Open-Call projects.

In the data model interoperability section of the surveys different questions have been asked in both surveys but the question on “how interoperability has been achieved” stayed the same to capture the degree of adoption of the tools in the two different scenarios.

### 4.4.1 Pilot Survey Analysis

The Pilot survey has been responded by 18 respondents, 15 of which identified themselves as “Tool or service provider”.

The majority of the respondents have used the provided tools to develop their interoperability workflows while a minority made changes to their tool. The adoption of the interoperability workflows during the Pilot Projects’ development phase has been increased thanks to the presence of many already mature solutions in the pilot projects. The developers have then felt more convenient to use these tools than to making changes to their solutions for the implementations of the described projects.

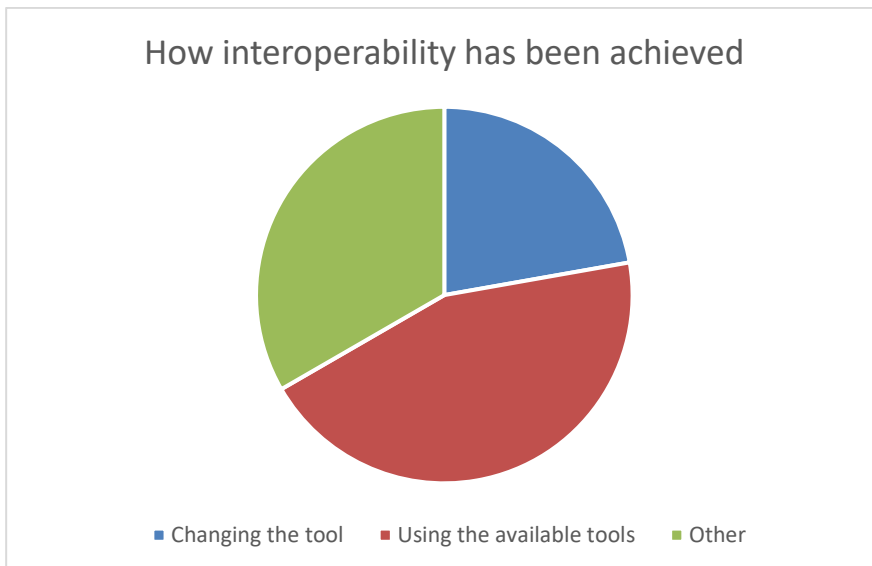


Figure 282. Data Model Interoperability Pilot Survey: Followed Interoperability Approach

It is interesting to note as well how the respondents who did not use the provided tools did not do that because they felt that those tools were too complex or because they lacked discoverability. The analysis of the open answer questions explains that the developers

resorted to making changes to their tools which were still under development (Other) or that no development of interoperability workflows was deemed as necessary (Not applicable).

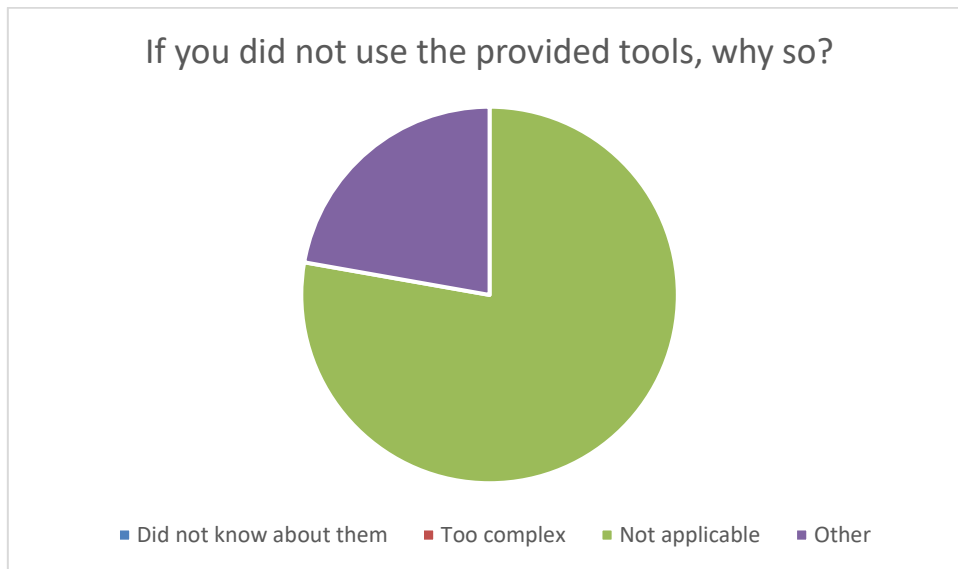


Figure 283. Data Model Interoperability Pilot Survey: Reason for Not Using the Provided Tools

#### 4.4.2 Open Call Survey Analysis

Although only 12 responses were collected at the time this document was written some interesting results can be found.

Users generally preferred changing their tool than using the available data model interoperability tools on the platform. This is possibly mostly since the respondent of the survey were developers actively working and bringing improvements to their tools and services. In this scenario it makes sense to be in a similar situation since the developers would probably more akin to make changes to their tools than to implement a separate workflow to make their tools interoperable. As with the pilot projects the respondents who marked the Other options did not require any interoperability workflow to be developed at all.

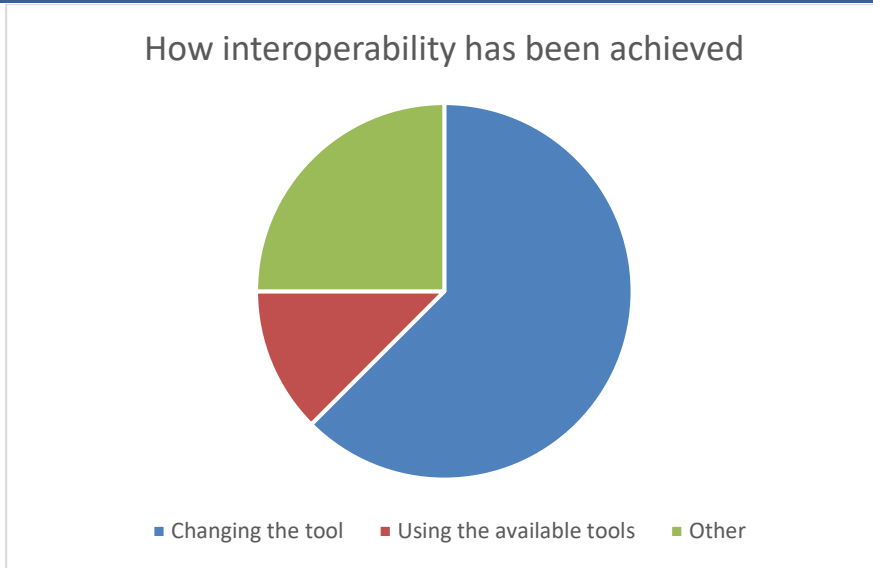


Figure 284. Data Model Interoperability Open Call Survey: Followed Interoperability Approach

Between the developers who used the tools available on the Data Spine to implement their interoperability workflows, one performed just operations on data, while the others performed both operations on data and data model transformations.

It is also worth noting that all the developers who used these tools found them to be “easy” to use. It is also interesting to note that no transformation towards a “standard” data model has been implemented between the open call projects.

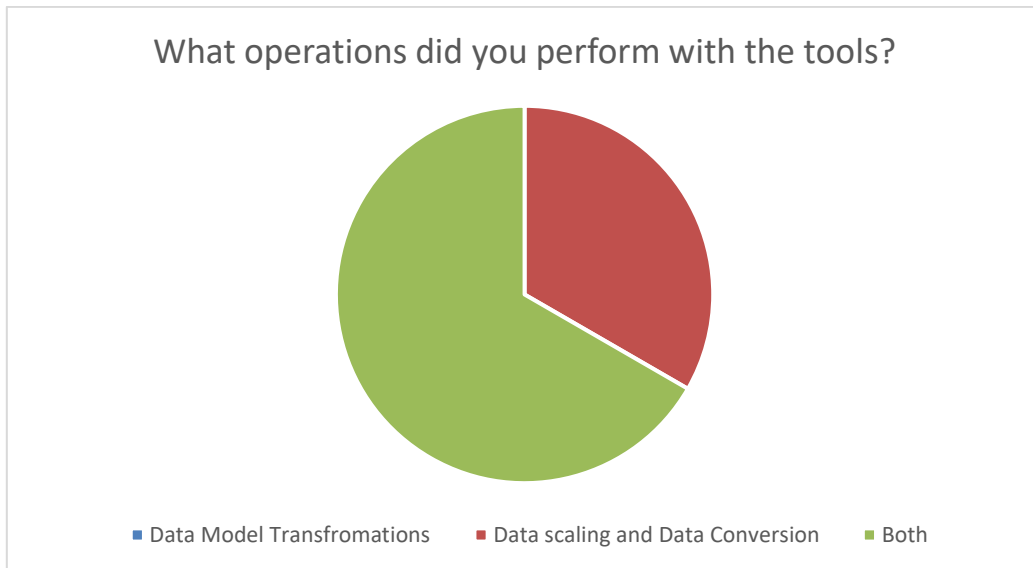


Figure 285. Data Model Interoperability Open Call Survey: Data Manipulation Operations Performed

#### 4.4.3 Data Model Interoperability Tools Adoption

Counting the active NiFi processors on the Test (Pilot projects) and Production (Open-call projects) environment gives an insight into which tools have been used the most.

	JOLT	XSLT	Replace Text	Microservice	Exec Script
# of active instances	19	3	14	2	0

Figure 286. Data Model Interoperability Tools Adoption Statistics

JOLT clearly is the tool which has been used the most thanks to its simplicity for quick operations on data. It is also important to note that most of the data exchanges happens using the JSON format which JOLT is usable on. For unstructured text on the other hand the Replace Text has been used. XSLT has been used only in the Matchmaking workflows to convert the data from the XML format to JSON format. Microservices have been used to make more complex data transformations (e.g., from proprietary data models to OGC Sensors Things). The Exec Script processors have never been used, probably due to the fact that are marked as “Experimental” on NiFi itself.

#### 4.4.3.1 SIM Tool

The Semantic Information Management (SIM) tool aims to make use of the formal semantic models represented in RDF/OWL to aid the process of data transformation. In addition, it aims to provide a drag-and-drop style GUI for the users to create mappings between the input and output data models. This tool is currently under development and more details will be included in the deliverable D4.14: Smart Factory Solutions in the EFPF Ecosystem - Final Report due at M48.

## 5 Summary of Architectural Considerations & Implications

This section presents a summary of architectural considerations and implications based on the requirements and from the perspectives of the providers and consumers of smart factory platforms, tools, services, and system integrator users concerning factors such as usability, multitenancy, platform integration efforts, etc.

- **Architectural design:** The EFPF ecosystem architecture classifies the EFPF components into two high-level categories:
  1. **Ecosystem Enablers:** The components that provide the core central functionality and tooling support to enable the creation and functioning of the ecosystem and operation of and communication among the other components. The Ecosystem Administrators are responsible for maintaining these components
  2. **Smart Factory Tools and Services:** These are the components from the connected platforms that provide use case- and domain-specific functionalities. The individual providers of these tools are responsible for their deployments and provisioning.

This classification makes it easy for the Ecosystem Administrators to focus on maintaining the core central infrastructure, while the System Integrator users can make use of the Ecosystem Enablers in order to integrate the Smart Factory Tools and Services with the ecosystem and compose them together to realise use case scenarios from the manufacturing domain.

- **Federated interoperability approach:** The EFPF ecosystem architecture follows a federation approach, where the interoperability between different tools/services is established “on-demand” i.e., when required by a use case through an integration flow. There is no common data model or API imposed at the ecosystem-level. Therefore, there is no overhead for the system administrators of maintaining such a complex canonical model and on the services to understand it and adhere to it. On the downside, the services are required to create integration flows to interoperate and communicate with each other, in contrast to the interoperability approaches where the services adhere to a common API that prescribes a common data model at the ecosystem-level and hence, there is no additional overhead of data transformation.
- **Communication patterns:** The EFPF ecosystem supports two communication patterns that are widely used in the manufacturing domain:
  1. Synchronous (request-response) pattern
  2. Asynchronous (Pub/Sub) pattern
- **Interoperability:** The Data Spine bridges the interoperability gaps between services mainly at the following levels:
  - **Protocol interoperability:** For the Synchronous (request-response) and the Asynchronous (Pub/Sub) pattern, the Data Spine supports standard application layer protocols that are widely used in the industry and employs an easily extensible mechanism for the supporting new protocols.
  - **Data model interoperability:** The Data Spine provides the necessary digital infrastructure and tooling support to transform between the message formats, data structures and data models of different services thereby bridging the interoperability gaps for data transfer.



- **Security interoperability:** The Data Spine facilitates the federated security and SSO capability in the EFPF ecosystem.
- **Interaction approach interoperability:** The mismatch in the interaction approaches followed by different services can hinder communication among them. For example, the services of one platform might expect separate steps for discovery and accessing data, while the services of another platform might provide a querying mechanism that combines the data discovery and retrieval operations. Another example would be, one service offers data over an HTTP GET endpoint, while the other expects it over an HTTP POST endpoint instead. Such interoperability gaps between the services at the levels of “interaction approaches” can be bridged by using one/more components of the Data Spine.
- **Dependence on IoT Gateways:** To support lower layer protocols and other IoT networking technologies (e.g., ZigBee, ZWave, BLE, etc.), the Data Spine relies on Factory Connectors and IoT Gateways deployed at the edge.
- **Agility and flexibility:** The use of the Data Spine to establish interoperability allows the tools/services to be loosely coupled. This allows the tools/services to have neutral APIs that are not strongly tied to any specific implementation and provides the flexibility to different tools/services to evolve independently. The reliance on APIs as contracts between service providers and service consumers is a standard practice; however, successful collaboration depends upon the former adhering to the semantic versioning [Sem22] standard recommended by the EFPF ecosystem to version their APIs and conveying plans to deprecate/upgrade their APIs to the latter in advance.
- **Usability, multitenancy, and collaboration:** The Data Spine provides an intuitive, drag-and-drop style GUI to the system integrator users to create integration flows with minimal effort. The collaboration of work concerning a particular integration flow among different users is easy to manage as the Data Spine provides a Web-based GUI for creating integration flows. In addition, it provides a multi-tenant authorization capability that enables different groups of users to command, control, and observe different parts of the integration flows, with different levels of authorization. In addition, the Federated Search, Matchmaking and Agile Networks Creation services enable users to search for collaborators, negotiate, form teams, and work collaboratively for realizing various use case scenarios.
- **Built-in functionality and tool/service integration effort:** The Data Spine provides connectors for standard communication protocols such as HTTP, MQTT, AMQP, etc., that are widely used in the industry. In order to enable transformations among different data models, it provides data transformation processors. For instance, the technology ‘Apache NiFi’ used to realise the IFE provides processors such as JoltTransformJSON, TransformXml, ExecuteScript, ReplaceText, ConvertRecord, etc., for performing data transformation [NDoc22]. Thus, the Data Spine takes care of the boilerplate code and facilitates the system integrator users for integrating their services by configuring only the service-specific parts of the integration flows with minimal coding effort.
- **Platform integration effort:** The Ecosystem Enablers are cloud-native solutions, and therefore, no additional local deployments are needed to integrate platforms through it. Integration of a platform with the EFPF ecosystem needs federating its Identity Provider with the EFS, registration of its services and integration with the unified ecosystem services such as the Integrated Marketplace.

- **API management:** The Data Spine provides a Service Registry component to store and retrieve service metadata including the API specifications. The Service Registry ensures uniformity across and completeness of the API specifications, Thus, this implies that the service providers need to register their services to the Service Registry and follow the proposed standards to specify the APIs of their services. The Pub/Sub Security Service provides mechanisms that enables the data providers to have direct control over who accesses their data published to the Message Bus. The ICMT tool notifies users in the case of any breaking changes to the APIs they are consuming.
- **Modularity and extensibility:** The architecture has been designed with modularity and extensibility in mind to meet the need for incorporating new tools, services, and platforms in the EFPF ecosystem with minimal effort. The Ecosystem Enablers are modular in nature and communicate with each other through standard interfaces and protocols. Support for new functionality such as protocols can be added by developing new processors/plugins. The Ecosystem Enablers adhere to common industry standards and follow a modular approach to enable the creation of a modular, flexible, and extensible ecosystem.
- **Performance, scalability, and availability:** The EFPF ecosystem makes it easy to integrate new tools/services and promotes reusability. To ensure high performance, high throughput and high availability, the performance critical Ecosystem Enablers such as the Data Spine have CD/CD pipelines configured for automated deployment and are deployed within a cluster using the Docker Swarm container orchestration technology.
- **Maintainability:** The loosely coupled and modular nature of the EFPF ecosystem helps significantly towards its maintainability. A high-quality user documentation of the Ecosystem Enablers and the smart factory services and tools in the EFPF ecosystem has been published on the EFPF Dev-Portal.
- **Use of standards:** The EFPF ecosystem uses or recommends the use of the following standards:
  - **Interoperability approach:** The Data Spine’s interoperability approach closely aligns with the Federated Interoperability Approach specified by the CEN/ISO 11354 Enterprise interoperability Framework.
  - **Communication patterns and protocols:** Synchronous (Request-Response) and Asynchronous (Pub/Sub) patterns and standard application protocols such as HTTP, MQTT, AMQP, etc.
  - **API specification:** OpenAPI and AsyncAPI specification standards
  - **Pub/Sub topic naming convention:** Eclipse Sparkplug™
  - **API versioning:** Semantic Versioning Standard - Semver (MAJOR.MINOR.PATCH)
  - **Data models:** No common, canonical data model is enforced/prescribed at the ecosystem-level; however, the use of the following data models is recommended:
    - Industrial IoT, Industry 4.0: OGC SensorThings, W3C WoT TD, OPC UA Part 100
    - Supply Chain and Logistics: BITAS, GS1 – EPCIS
    - Platform Marketplace: UBL

## 6 Conclusion and Outlook

The increasing digitalisation and servitization in the manufacturing companies opens new opportunities for them to collaborate, share data, reuse each other's resources such as products, tools, and services, etc., to enable a variety of new business models and revenue streams. However, such opportunities are lost because of the interoperability gaps among today's digital manufacturing platforms and the current absence of technology enablers for an easy creation of cross-platform applications. To fill this gap, the EFPF ecosystem provides the core 'Ecosystem Enablers' that enable an easy integration of tools, services, and platforms regardless of who provides them, or where they are deployed at, in order to create innovative composite applications using cross-platform capabilities. This enables the reuse and application of the existing proven solutions in other companies, factories contexts, or domains.

This deliverable presents the final report of the EFPF ecosystem architecture, the design and realisation of the Data Spine, the interfaces for tools, services, systems, and platforms from the EFPF ecosystem and the data model interoperability layer.

The new categorization of components into two types, namely Ecosystem Enablers and Smart Factory Tools, Services, and Platforms, makes the administration of the ecosystem easier and more efficient. The architectural design enables the ecosystem to be modular, scalable, and extensible.

The Data Spine provides the core functionalities and tooling support for identity federation, cross-platform interoperability, and an easy composition of services. It supports synchronous request-response as well as asynchronous Pub/Sub communication patterns that are widely used in the industry. It bridges the interoperability gaps between services at the levels of security, communication protocol, data model, and interaction patterns, and provides built-in protocol connectors and data transformation tools. For an easier and efficient administration, the DevOps process is fully automated and monitoring and alerting tools are used. On the other hand, to make its usage easier, comprehensive user documentation with templates and examples is published onto the EFPF Dev-Portal which is publicly accessible. Moreover, to ensure scalability and high availability, the Data Spine components are deployed in a clustered fashion.

This deliverable describes the interfaces of the tools, services, systems, and platforms that constitute the EFPF ecosystem, along with the API management and Interface Contracts management mechanisms.

The data model interoperability layer section provides an analysis of the data models and dataflows in the pilot scenarios as well as two Open Call experimentation scenarios. The interoperability workflows developed have tackled different issues, from data conversion to data model transformations. The analysis shows that these interoperability workflows tackle different issues, from data conversion to data model transformations, with different degrees of complexity and this proves that the ecosystem has been capable of supporting even demanding workflows.

Finally, the implementation of EFPF pilot use cases and validation and the implementation of the Open Call experimentation scenarios has proved that the EFPF components are capable of realising the real-world applications to provide state-of-the-art solutions. Further results will be disseminated through publications and the EFPF Interop CWA (CEN-CENELEC Workshop Agreement) with the objective of presenting a standardized interoperability framework enabling an open market infrastructure for European

manufacturing, interlinking digital manufacturing platforms, smart factory tools and Industry 4.0 concepts to realise and support a connected and smart factory ecosystem of the future.

## Annex A: History

Document History	
Versions	V1.3 <ul style="list-style-type: none"> <li>Internal review 2</li> </ul>
	V1.2 <ul style="list-style-type: none"> <li>Internal review 1</li> </ul>
	V1.1 <ul style="list-style-type: none"> <li>Consolidated the contributions from SIE, CMS, FOR</li> </ul>
	V0.4-V1.0 <ul style="list-style-type: none"> <li>Consolidated the contributions from partners</li> </ul>
	V0.3 <ul style="list-style-type: none"> <li>Integrated the contributions from FIT on the architecture, Ecosystem Enablers, Data Spine, and the interfaces for tools, services, and platforms</li> </ul>
	V0.2 <ul style="list-style-type: none"> <li>Creation of sample sections that describe the level of expected in partners' contributions</li> </ul>
	V0.1 <ul style="list-style-type: none"> <li>Creation of ToC</li> </ul>

**Contributions**

- FIT:
  - Rohit Deshmukh
  - Alexander Schneider
- LINKS:
  - Edoardo Pristeri
- ICE:
  - Usman Wajid
  - Ross Campbell
  - Robert Woodfin
  - Jacob Griffiths
- SRFG:
  - Violeta Damjanovic-Behrendt
  - Dietmar Glachs
- ASC:
  - Norman Wessel
  - Brian Clark
  - Anouar Mabrouk
- VLC:
  - Amber Bu
- CNET:
  - Mathias Axling
  - Matts Ahlsen
- CERTH:
  - Alexandros Nizamis
  - Dimosthenis Ioannidis
  - Georgios Michailidis
- C2K:
  - Simon Osborne
- NXW:
  - Gabriele Scivoletto
  - Gianluca Insolubile
- ALM:
  - Carolyn Langen
  - Carlos Hermans
- SRDC:
  - Senan Postaci
- UOS-ITI:
  - Stefano Modafferi
- ELN:
  - Maximilian Norz
- SIE:
  - Raluca Repanovici
  - Alexandru Plesa
  - Vladut Dinu
  - Cosmin Grigoras
- CMS:
  - Carlos Coutinho
  - Miguel Tavares
- FOR:
  - Rute Sofia

## Annex B: References

- [Mor10] Morrison, J. Paul. Flow-Based Programming: A new approach to application development. CreateSpace, 2010.
- [Shu86] Shu, Nan C. "Visual programming languages: A perspective and a dimensional analysis." Visual Languages. Springer, Boston, MA, 1986.
- [JGH20] Jolt source code repository and readme documentation on Github. <https://github.com/bazaarvoice/jolt> (accessed on 1 June 2022).
- [JPG20] Jolt playground. <http://jolt-demo.appspot.com/> (accessed on 1 June 2022).
- [JNF20] NiFi Processor: JoltTransformJSON. <https://nifi.apache.org/docs/nifi-docs/components/org.apache.nifi/nifi-standard-nar/1.5.0/org.apache.nifi.processors.standard.JoltTransformJSON/> (accessed on 1 June 2022).
- [TX20] NiFi Processor: TransformXml. <https://nifi.apache.org/docs/nifi-docs/components/org.apache.nifi/nifi-standard-nar/1.11.4/org.apache.nifi.processors.standard.TransformXml/index.html> (accessed on 1 June 2022).
- [ES20] NiFi Processor: ExecuteScript. <https://nifi.apache.org/docs/nifi-docs/components/org.apache.nifi/nifi-scripting-nar/1.11.4/org.apache.nifi.processors.script.ExecuteScript/index.html> (accessed on 1 June 2022).
- [NOG] Apache NiFi Overview Guide. <https://nifi.apache.org/docs/nifi-docs/html/overview.html> (accessed on 1 June 2022).
- [NAR20] Apache NiFi REST API Documentation. <https://nifi.apache.org/docs/nifi-docs/rest-api/index.html> (accessed on 1 June 2022).
- [NAG20] Apache NiFi Administrator's Guide: [https://nifi.apache.org/docs/nifi-docs/html/administration-guide.html#tls\\_generation\\_toolkit](https://nifi.apache.org/docs/nifi-docs/html/administration-guide.html#tls_generation_toolkit) (accessed on 1 June 2022).
- [RMQ20] RabbitMQ Documentation. <https://www.rabbitmq.com/documentation.html> (accessed on 1 June 2022).
- [CAQ20] RabbitMQ for beginners. <https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html> (accessed on 1 June 2022).
- [OAS20] OpenAPI Specification 3.0.3. <https://swagger.io/specification/> (accessed on 1 June 2022).
- [AAS20] AsyncAPI Specification 2.0.0. <https://www.asyncapi.com/docs/specifications/2.0.0/> (accessed on 1 June 2022).
- [VIDR18] M. Vidrih, 2018. How will Blockchain Work in Industry 4.0? Online available: <https://medium.com/datadriveninvestor/how-will-blockchain-work-in-industry-4-0-efdb5446e40c>. (accessed on 1 June 2022).
- [KON20] Kong 2.0 Open Source API Gateway. <https://konghq.com/kong/>. (accessed on 1 June 2022).
- [OAM20] OGC SWE: Observation Measurement (O&M). <https://www.opengeospatial.org/standards/om> (accessed on 1 June 2022).
- [SML20] OGC SWE: SensorML <https://www.opengeospatial.org/standards/sensorml> (accessed on 1 June 2022).
- [SOS20] OGC SWE: Sensor Observation Service (SOS). <https://www.opengeospatial.org/standards/sos> (accessed on 1 June 2022).

- [SPS20] OGC SWE: Sensor Planning Service (SPS). <https://www.opengeospatial.org/standards/sps> (accessed on 1 June 2022).
- [STA20] OGC SWE: 5. SensorThings API. <https://www.opengeospatial.org/standards/sensorthings> (accessed on 1 June 2022).
- [WTD20] Web of Thing - Thing Description (TD). <https://www.w3.org/TR/wot-thing-description/> (accessed on 1 June 2022).
- [OUD20] OPC UA Part 100: Devices. <https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-100-device-information-model> (accessed on 1 June 2022).
- [BIT19] BiTAS Std 120-2019: LOCATION COMPONENT SPECIFICATION. <https://www.bitas.studio/s/BiTAS-Location-Component-Specification-v4-rf7s.pdf> (accessed on 1 June 2022).
- [BTD20] BiTAS Tracking Data Framework Profile. [https://static1.squarespace.com/static/5aa97ac8372b96325bb9ad66/t/5c7e88397817f73e6c60a967/1551796284047/BiTAS+Tracking+Data+Framework+Profile+v9\\_ISTO.pdf](https://static1.squarespace.com/static/5aa97ac8372b96325bb9ad66/t/5c7e88397817f73e6c60a967/1551796284047/BiTAS+Tracking+Data+Framework+Profile+v9_ISTO.pdf) (accessed on 1 June 2022).
- [EPC20] EPCIS. <https://www.gs1.org/epcis/epcis/1-1> (accessed on 1 June 2022).
- [GS120] GS1. <https://www.gs1.org/> (accessed on 1 June 2022).
- [RFC34] RFC3444. <https://tools.ietf.org/html/rfc3444> (accessed on 1 June 2022).
- [RMP20] RabbitMQ Messaging Patterns. <https://www.rabbitmq.com/getstarted.html> (accessed on 1 June 2022).
- [Sim17] European Commission, Directorate-General for Internal Market, Industry, Entrepreneurship and SMEs, Simons, M., Van de Velde, E., Copani, G., et al., An analysis of drivers, barriers and readiness factors of EU companies for adopting advanced manufacturing products and technologies, Publications Office, 2017, <https://data.europa.eu/doi/10.2873/715340>
- [Des21] Deshmukh, R.A.; Jayakody, D.; Schneider, A.; Damjanovic-Behrendt, V. Data Spine: A Federated Interoperability Enabler for Heterogeneous IoT Platform Ecosystems. Sensors 2021, 21, 4010. <https://doi.org/10.3390/s21124010>
- [EMC22] European Collaborative Manufacturing and Logistics Cluster. Available online: <https://ec.europa.eu/info/funding-tenders/opportunities/portal/screen/opportunities/topic-details/fof-11-2016> (accessed on 1 June 2022).
- [NIM22] Nimble (Collaboration Network for Industry, Manufacturing, Business and Logistics in Europe) Project. Available online: <https://www.nimble-project.org/> (accessed on 1 June 2022).
- [COM22] COMPOSITION (Ecosystem for Collaborative Manufacturing Processes) Project. Available online: <https://www.composition-project.eu/> (accessed on 1 June 2022).
- [DIG22] DIGICOR (Decentralised Agile Coordination Across Supply Chains) Project. Available online: <https://www.digicor-project.eu> (accessed on 1 June 2022).
- [VFO22] vf-OS (Virtual Factory Operating System) Project. Available online: <https://www.vf-os.eu> (accessed on 1 June 2022).



- [VLC22] ValueChain's Network Portal platform. Available online: <https://valuechain.com/products/network-portal/> (accessed 1 June 2022).
- [NXW22] Nextworks' Symphony platform. Available online: <https://www.nextworks.it/en/products/symphony> (accessed 1 June 2022).
- [C2K22] SMECluster's IndustrieWeb platform. Available online: <https://www.industreweb.co.uk/> (accessed 1 June 2022).
- [Hil00] Hilliard, Rich. "Ieee-std-1471-2000 recommended practice for architectural description of software-intensive systems." IEEE, <http://standards.ieee.org> 12.16-20 (2000): 2000.
- [IEEE11] May, I. S. O. Systems and software engineering—architecture description. Technical Report. ISO/IEC/IEEE 42010, 2011.
- [RW05] Rozanski, Nick, and Eoin Woods. "Software Systems Architecture: Viewpoint Oriented System Development." (2005).
- [SUL20] Swagger UI. Available online: <https://github.com/swagger-api/swagger-ui>. (accessed on 15 June 2022).
- [Sem22] Preston-Werner, T. Semantic Versioning 2.0.0. Available online: <https://semver.org/spec/v2.0.0.html> (accessed on 15 June 2022).
- [NDoc22] Apache NiFi Documentation. Available online: <http://nifi.apache.org/docs.html> (accessed on 15 June 2022).
- [EFS22] EFPF Security Portal User Guide. Available online: <https://docs.efpf.linksmart.eu/projects/efs/efs-user-guide/> (accessed on 15 June 2022).
- [ISO11354] ISO 11354-1:2011 Advanced Automation Technologies and Their Applications—Requirements for Establishing Manufacturing Enterprise Process Interoperability—Part 1: Framework for Enterprise Interoperability. Available online: <https://www.iso.org/standard/50417.html> (accessed on 6 January 2021).
- [ASGI22] asg-importer micro-service. Available online: <https://github.com/nimble-platform/asg-importer> (accessed on 15 June 2022).
- [SAR22] SAREF: The Smart Applications REference ontology. Available online: <https://saref.etsi.org/>. (accessed on 15 June 2022).
- [TSM22] TSMATCH v1.0. Available online: [https://git.fortiss.org/iiot\\_external/tsmatch](https://git.fortiss.org/iiot_external/tsmatch) (accessed on 1 June 2022).
- [BNO22] Applying MQTT Sparkplug in the EFPF Platform. [https://www.researchgate.net/publication/358618553\\_White\\_Paper\\_Applying\\_MQTT\\_Sparkplug\\_in\\_the\\_EFPF\\_Platform](https://www.researchgate.net/publication/358618553_White_Paper_Applying_MQTT_Sparkplug_in_the_EFPF_Platform) (accessed on 1 June 2022).
- [NBN22] N. Bnouhanna, E. Karabulut, Rute C. Sofia, E. E. Seder, G. Scivoletto, G. Insolubile, An Evaluation of a Semantic Thing To Service Matching Approach in Industrial IoT Environments. In Proc. IEEE Percom 2022 IoT-Prod workshop. June 2022, Pisa, Italy.
- [NEB22] NPM Nebulous Plasma Muffin npm string-similarity package. Available online: <https://www.npmjs.com/package/string-similarity> (accessed on 1 June 2022).
- [NAT22] NPM natural package. Available online: <https://www.npmjs.com/package/natural> (accessed on 1 June 2022).

[POS22] PostgreSQL Docker image. Available online: [https://hub.docker.com/\\_/postgres](https://hub.docker.com/_/postgres) (accessed on 1 June 2022).

[COA22] The lightweight open-source framework for Collaborative IoT. Available online: <https://coaty.io/> (accessed on 1 June 2022).

[EFF22] European Factory Foundation - EFF. Available online: <https://efactoryfoundation.org/> (accessed on 1 June 2022).

[AID22] AIDIMME B2BMarket. Available online: <https://b2bmarket.aidimme.es/> (accessed on 1 June 2022).

[EX122] DS NiFi Integration Flow Examples. Available online: <https://docs.efpf.linksmart.eu/projects/data-spine-nifi/ds-nifi-iflow-examples/> (accessed on 1 June 2022).

[EX222] NiFi Data Transformation Examples. Available online: <https://docs.efpf.linksmart.eu/projects/data-spine-nifi/nifi-dmx-examples/> (accessed on 1 June 2022).

## Annex C: Data Spine Testing Scenarios

### Integration and System Test Scenarios:

#### 1. EFS (Keycloak)

ID	Test Objective / Functionality / Feature	Action	Expected Result
1	Availability and Keycloak's GUI	Open <a href="https://&lt;Keycloak base URL&gt;/auth/">https://&lt;Keycloak base URL&gt;/auth/</a> E.g.: <a href="https://efpf-security-portal.salzburgresearch.at/auth/">https://efpf-security-portal.salzburgresearch.at/auth/</a>	Keycloak's Welcome Page should be displayed
2	User authentication through GUI	On 'Keycloak's Welcome Page', click on 'Administration Console' or go to <a href="https://&lt;Keycloak base URL&gt;/auth/admin">https://&lt;Keycloak base URL&gt;/auth/admin</a> and upon redirection to login page, enter valid login credentials	Login should be successful
3	Admin functionality	Check creation of users, roles, clients, assignment of roles to users, etc.	All the operations should be successful
4	OIDC discovery	Open OIDC discovery URL: <a href="https://&lt;Keycloak base URL&gt;/auth/realms/master/.well-known/openid-configuration">https://&lt;Keycloak base URL&gt;/auth/realms/master/.well-known/openid-configuration</a>	OIDC discovery URL should be accessible without login and should return OIDC metadata in the form of a JSON listing the OpenID/OAuth endpoints, supported scopes and claims, public keys used to sign the tokens, and other details.
5	Get EFS token	Make a POST request to OIDC token endpoint ' <a href="https://&lt;Keycloak base URL&gt;/auth/realms/master/protocol/openid-connect/token">https://&lt;Keycloak base URL&gt;/auth/realms/master/protocol/openid-connect/token</a> ' with required parameters	EFS should return 'access_token' in the response
6	Check other functionality such as token introspection	Make a POST request to OIDC token endpoint ' <a href="https://&lt;Keycloak base URL&gt;/auth/realms/master/protocol/openid-connect/token/introspect">https://&lt;Keycloak base URL&gt;/auth/realms/master/protocol/openid-connect/token/introspect</a> ' with required parameters	EFS should return the active state and information about the token, including the permissions or roles granted by Keycloak

#### 2. Message Bus (RabbitMQ)

ID	Test Objective / Functionality / Feature	Action	Expected Result
1	Availability and RabbitMQ's GUI	Open RabbitMQ base URL in a browser E.g.: <a href="https://dataspine.efpf.linksmart.eu/rabbitmq">https://dataspine.efpf.linksmart.eu/rabbitmq</a>	RabbitMQ's login page should be displayed
2	Login	Enter admin's credentials to on RabbitMQ's login page	Login should be successful
3	Lifecycle management of users and vhosts	CRUD users and vhosts through RabbitMQ's Management GUI	All operations should be successful

4	Management of access permissions	Using RabbitMQ's Management GUI, create a vhost 'vhost1' and two users 'user1' and 'user2' with their email Ids linked with EFS as username in RabbitMQ. For vhost1, configure permissions and topic permissions for user1 and user2 so that user1 can only publish over topic 'topic1' and user2 can only subscribe to topic1.	All operations should be successful
5	Pub/Sub with MQTTS/AMQPS and access control	Using MQTT/AMQP clients, publish over topic1 under vhost1 to RabbitMQ with user1's credentials and subscribe to it with user2's credentials.	Only user1 should be able to publish to topic1 under vhost1 and only user2 should be able to subscribe to it.  user1 and user2 should not be able to pub/sub to any other topic under any vhost.

### 3. Integration Flow Engine (Apache NiFi)

ID	Test Objective / Functionality / Feature	Action	Expected Result
1	Availability and NiFi's GUI	Open <a href="https://&lt;NiFi base URL&gt;/nifi/">https://&lt;NiFi base URL&gt;/nifi/</a> E.g.: <a href="https://dataspine.efpf.linksmart.eu/nifi/">https://dataspine.efpf.linksmart.eu/nifi/</a>	NiFi should redirect to EFS Keycloak login page
2	Login	Login to NiFi with 'First User' ('INITIAL_ADMIN_IDENTITY') credentials	Login should be successful and NiFi's canvas page should be displayed
3	Drag-and-drop functionality and lifecycle management of process groups, processors, users, user groups, templates, etc.	Use NiFi's drag-and-drop functionality to CRUD process groups and integration flows using processors.  CRUD users (with their email Id linked with EFS as username), user groups, templates, etc.	All operations should be successful
4	Lifecycle management of controller services	CRUD, enable, disable controller services such as 'StandardHttpContextMap' and 'StandardRestrictedSSLContextService'	All operations should be successful
5	Access control and multi-tenancy	Use NiFi's GUI to create users 'user1' (username: user1@example.com) and 'user2' (username: <a href="mailto:user2@example.com">user2@example.com</a> ), process groups 'pg1' and 'pg2' and define access policies for these process groups so that only user1 can view and access pg1 and only user2 can view and access pg2.	Users should be able to view and access the process groups as per the defined access permissions. In this case, user1 should be able to view and access pg1 and user2 should be able to view and access pg2.
6	Operation	Synchronous communication: Create a workflow 'workflow1' to call a REST API endpoint using InvokeHTTP processor, perform data transformation with a processor such as Jolt or XSLT and expose an endpoint using HandleHttpRequest processor. Make a call to this endpoint.	Both operations should be successful

	Asynchronous communication: Create a workflow to publish to the Data Spine RabbitMQ over a topic using MQTTS and another workflow to subscribe to that topic.	
--	---	--

#### 4. Service Registry (LinkSmart Service Catalog)

ID	Test Objective / Functionality / Feature	Action	Expected Result
1	Access to SR API	Try to access SR API from the VM where it is deployed and from outside	SR's API should be accessible only from the VM where it is deployed (alongside API Security Gateway)
2	SR REST API	CRUD, list, and filter services using SR's REST API	All operations should be successful
3	SR MQTT Service Status Announcement API	Subscribe to 'alive' and 'dead' Service Status Announcement topics defined in SR's configuration file.	Messages over 'alive' topic should be received whenever a service is created/updated and over 'dead' topic when a service is deleted (by a user or when its 'ttl' expires).

#### 5. API Security Gateway (Apache APISIX and ASG Importer)

ID	Test Objective / Functionality / Feature	Action	Expected Result
1	Startup	Start APISIX and then ASG Importer	ASG Importer should create routes in APISIX for all the service APIs registered in SR
2	Routes	Open <a href="https://&lt;EFS base URL&gt;/apisix/admin/routes">https://&lt;EFS base URL&gt;/apisix/admin/routes</a>	All routes in APISIX should be displayed
3	SR routes	Through SR proxy route in SR, CRUD services	Only the users with role 'efpf_basic' should be able to read and users with role 'efpf_admin' should be able to write to SR.
4	Policy enforcement	Configure policies in EFS for a route and invoke it.	Only authorized users should be able to perform authorized operation(s) on that route.

#### 6. Data Spine

ID	Test Objective / Functionality / Feature	Action	Expected Result
1	Operation – Part 1	As a user with role 'efpf_admin', register the API exposed by workflow1 in NiFi to SR as 'service1' with service id 's_id' and API Id 'api_id' through SR's proxy route in APISIX.	ASG Importer should create route route1 in APISIX for service1
2	Operation – Part 2	Configure access permissions for route1 in EFS, giving read/write access permissions to user1	The operation should be successful

3	Operation – Part 3	Using the EFS authorization token granted to user1, invoke route1: https://<NiFi root URL>/<the two letter identifier of pg1's parent PG directly inside the 'NiFi Flow' PG>/<identifier for pg1>/<the rest of the URL path specific to the Integration Flows inside pg1>	The expected response should be returned
---	--------------------	---	--



European Factory  
Platform

[www.efpf.org](http://www.efpf.org)