EFPF: European Connected Factory Platform for Agile Manufacturing

European Factory Platform

WP3: EFPF Architecture

D3.2: EFPF Data Spine Realisation - I Vs: 1.0

Deliverable Lead and Editor: Rohit Deshmukh, Fraunhofer FIT

Contributing Partners: FIT, LINKS, SRFG, CNET, C2K, ICE, ASC, VLC, AID, ALM, FOR, NXW, CERTH, SRDC, CMS

Date: 2020-06

Dissemination: Public

Status: <~~Draft~~ ¦ ~~Consortium Approved~~ ¦ EU Approved>

Short Abstract

This deliverable presents an update to the architecture of EFPF ecosystem since D3.1: Design and Realisation of Interoperable Data Spine. The deliverable details the components of Data Spine and specifies how heterogeneous services can be integrated through the Data Spine to enable communication in the EFPF ecosystem.

Grant Agreement:
825075

Document Status

Deliverable Lead	Rohit Deshmukh, Fraunhofer FIT
Internal Reviewer 1	Edoardo Pristeri, LINKS
Internal Reviewer 2	Nisrine Bnouhanna, FOR
Type	Deliverable
Work Package	WP3: EFPF Architecture
ID	D3.2: EFPF Data Spine Realisation - I
Due Date	2020-06-30
Delivery Date	2020-06-30
Status	<~~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

Disclaimer

Project Partners:

ICE Information Catalyst Services Data Software

iT innovation

ASCORA Consulting - Development - Research

3D ICOM TECHNOLOGIES

BRIMATECH

AIDIMME INSTITUTO TECNOLÓGICO METALMECÁNICO, MUEBLE, MADERA, EMBALAJE Y AFINES

AM Allied Maintenance

AUSTRIAN STANDARDS

CONTROL 2K total solutions provider

elanyo ELASTIC ANALYTICS

Caixa Mágica Software

ΕΛΔΙΑ ΕΛΛΗΝΙΚΗ ΔΙΑΧΕΙΡΙΣΗ ΑΠΟΡΡΙΜΜΑΤΩΝ Α.Ε

fortiss

Innovint AIRCRAFT INTERIOR

LINKS ISMB SITI

NEXTWORKS ENGINEERING FORWARD

SIEMENS

SRDC

valuechain

KLEEMANN Your 1st Choice in Lifts

CNet SVENSKA AB

Fraunhofer FIT

AIRBUS

HANSE AEROSPACE

CERTH CENTRE FOR RESEARCH & TECHNOLOGY HELLAS

salzburgresearch

miloil

almende ORGANIZING NETWORKS

LAGRAMA

Executive Summary

This deliverable presents the current state of the design and realisation of Data Spine, the integration and interoperability layer that enables communication in the EFPF ecosystem, and explains how services can be integrated through the Data Spine.

This deliverable reports on the progress being made 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-II

T3.2: Design and Realisation of Interoperable Data Spine-I

T3.4: Interfaces for Tools, Systems and Platforms-I

T3.5: Data Model Interoperability Layer-I

In this respect, the deliverable presents an update to the architecture of the EFPF ecosystem from the baseline architecture presented in D3.1. Based on the refinements in the architecture, the deliverable describes the interfaces for tools, services, systems and platforms in the EFPF ecosystem and the data model interoperability layer that aligns the data models of different tools and platforms and specifies how data transformation can be performed to facilitate the interplay and interconnectivity between distributed technologies.

The architecture of the EFPF Data Spine and Platform has been designed with modularity and extensibility in mind to meet the need for incorporating new tools in the EFPF platform and external platforms in the EFPF ecosystem, with minimum effort, and also the needs of users and experimenters.

One important objective of this deliverable is to provide necessary information to the EFPF project participants as well as external entities who might be interested in interlinking their tools/services through the Data Spine and making them part of the EFPF ecosystem. To enable this, the deliverable highlights the steps a service provider needs to perform in order to provide his/her service through the Data Spine and the steps a service consumer needs to perform to consume services that are offered through the Data Spine.

The abstract architecture and realisation of the Data Spine presented in this deliverable will be subsequently developed and enhanced as the project progresses and the updated architectural specification and implementation details will be included in the next version of this deliverable D3.3 at M42 of the EFPF project. The updates to the interfaces for tools, services, systems and platforms and the data model interoperability layer will also be included in that deliverable.

Table of Contents

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

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, “D3.2: EFPF Data Spine Realisation - I”, is to present four different aspects of the EFPF ecosystem: the architecture, the design and realisation of Interoperable Data Spine, the interfaces for tools, systems and platforms of 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 baseline architecture that was presented in the previous architecture deliverable D3.1. 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 of service through the Data Spine i.e. the steps that a service provider needs to perform in order to provide his/her service through the Data Spine and the steps a service consumer needs to perform to consume services through the Data Spine. Third, the interfaces for tools, services, systems and platform which are the building blocks of the EFPF ecosystem, the management of their APIs and APIs contracts between them. Fourth and 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.

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, operation, etc. The scope of this deliverable also includes 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 through the Data Spine 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 platform and the Data Spine.

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
- **D2.3: Requirements of Embedded Pilot Scenarios:** Provide an overview of the pilot requirements on the federated EFPF platform
- **D3.1: EFPF Architecture-I:** Presents the baseline architecture of the EFPF ecosystem with focus on the EFPF platform and the Data Spine
- **D4.1: 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
- **D5.1: EFPF Matchmaking and Intelligence Gathering:** Presents an account of developments achieved under Matchmaking and Intelligence gathering tasks in EFPF project
- **D5.2: EFPF Security and Governance:** Provides details about the Governance Rules and Trust Mechanisms, Holistic Security, Privacy and User Management Framework in EFPF
- **D5.3: 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

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 baseline architecture that was presented in the previous architecture deliverable D3.1;
- **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: Conclusion and Outlook:** Concludes the deliverable and mentions the future work.
- **Annexes**:
  - **Annex A**: History
  - **Annex B**: References
  - **Annex C**: API Specifications

0.6 Document Status

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

0.7 Document Dependencies

This document is the second of the two deliverables that describe the architecture of the EFPF ecosystem. The first deliverable submitted at Month 9 of the EFPF project described the baseline architecture of the EFPF ecosystem and its components. This second and final architecture deliverable at Month 18 provides the final architecture.

This document is also the first of the two deliverables that describe the design and realisation of the Data Spine, the interfaces for tools, systems and platforms, and the data model interoperability layer. This first deliverable at Month 18 of the EFPF project describes the above-mentioned aspects of the EFPF ecosystem. The second and final deliverable at Month 42 provides the final report.

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".

Further information can be found at 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 platform, the Data Spine and the EFPF federated ecosystem from the baseline architecture presented in the previous deliverable D3.1. In order to preserve the context and ensure readability, some sections and information from D3.1 is also included in this section.

1.1 EFPF Architecture Context View

Figure 1 presents an overview of the high-level architecture of the EFPF platform, the Data Spine and the EFPF federated ecosystem of base and external platforms. In contrast to the high-level architecture diagrams shown in the DoA, the architecture in Figure 1 further details the composition and role of Data Spine in the EFPF ecosystem and its relationship and interaction with other components.

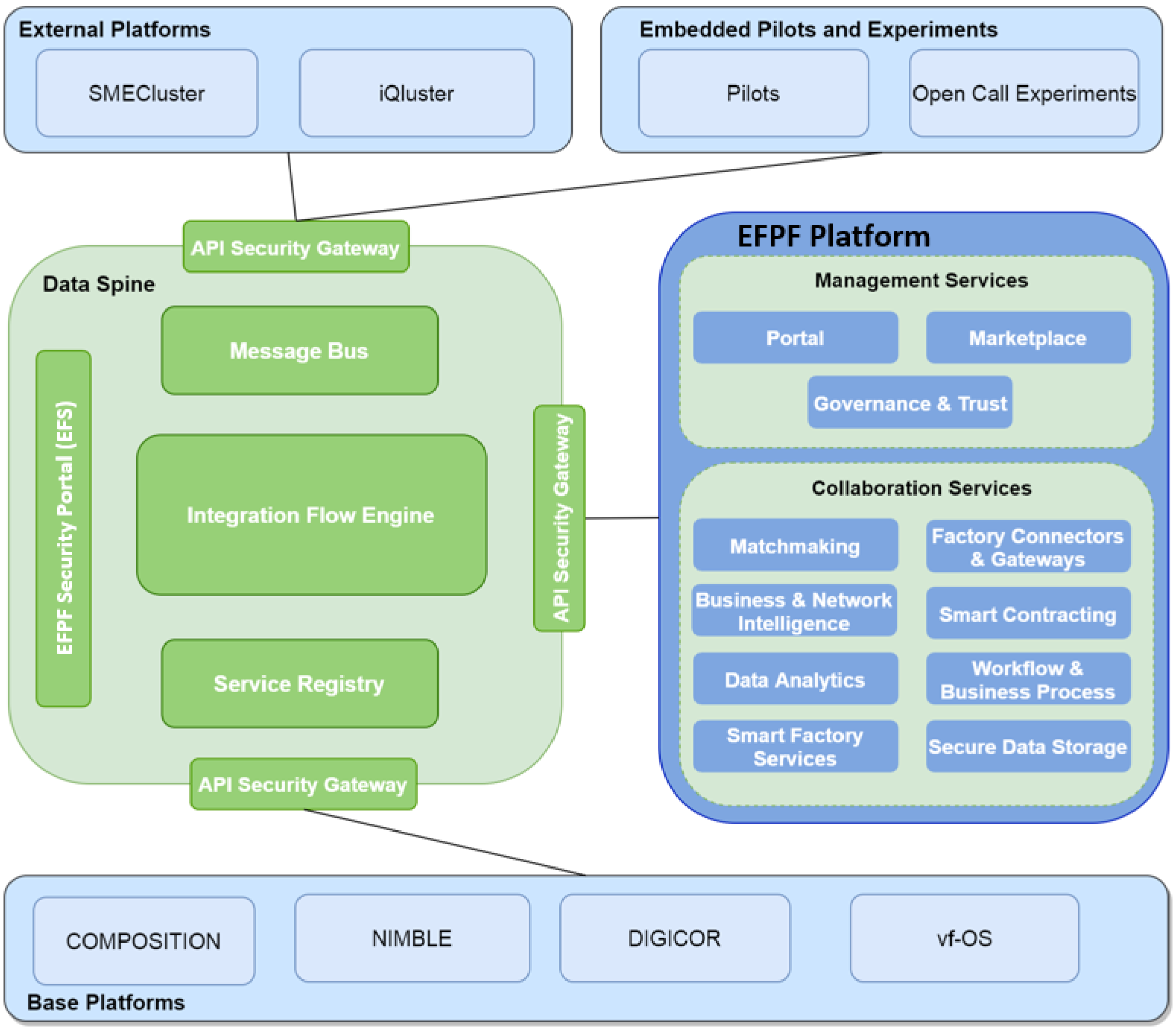


Figure 1: High-level Architecture of the EFPF Ecosystem

The EFPF platform follows the micro-service 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 and avoiding the lag-time introduced by interconnection buses.

The EFPF ecosystem is based on a federation model, which consists of distributed platforms, tools and components provided by several partners. These individual components communicate through a central entity called ‘Data Spine’. Thus, the EFPF ecosystem as a whole follows Service-oriented architecture (SOA) style. The main elements in the EFPF federation are:

- **Data Spine:** This 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 platform. Therefore, it can be easily extended beyond interconnecting the base platforms to ‘plug’ new external platforms in and interlink them with the existing platforms. Figure 1 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 platforms (external and third party). 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 a Web-based portal. 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 project partners. The collection of enhanced versions of such tools and services from the base or external platforms deployed together as micro-services 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 EFPF ecosystem
- **Base Platforms:** The four base platforms (COMPOSITION, DIGICOR, NIMBLE and vf-OS) in EFPF are funded by the European Commission's Horizon 2020 program within the Collaborative Manufacturing and Logistic Cluster (FoF-11-2016). These base platforms are interlinked through the Data Spine, which offers seamless interoperability of distributed tools and services by integrating, aligning and enhancing the open APIs of the existing platforms
- **External Platforms:** In addition to the four base platforms, the EFPF ecosystem enables interlinking of other platforms and open-source tools that address the specific needs of connected smart factories. The external platforms that joined the EFPF ecosystem at the beginning of the project are: ValueChain’s iQluster platform[1] and SMECluster’s Industreweb platform[2]
- **Pilots and Experiments:** These are the components and systems that will interact with the EFPF ecosystem (including the EFPF platform and the Data Spine) during the course of the project

[1] https://valuechain.com/supply-chain-intelligence/iqluster
[2] https://www.industreweb.co.uk/

1.2 EFPF Architecture Functional View

The EFPF ecosystem consists of two major components: The Data Spine and the EFPF platform, as illustrated in Figure 1. This section provides an architectural overview of these components and their subcomponents, responsibilities, and interactions with other (sub) components. The detailed description of the Data Spine can be found in Section 2 and the detailed description of the components of the EFPF platform can be found in the deliverables D4.1, D5.1, D5.2 and D5.3.

1.2.1 Overview of Data Spine

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

- **Protocol interoperability**: The Data Spine supports two communication patterns:
  1. Synchronous request-response pattern
  2. Asynchronous publish-subscribe pattern

  While the Data Spine supports standard protocols that are widely used in the industry, it employs an easily extensible mechanism for adding support for new protocols
- **Data Model interoperability**: The Data Spine provides a platform and mechanisms to transform between the message formats, data structures and data models of different services thereby bridging the syntactic and semantic gaps for data transfer
- **Security interoperability**: The EFPF Security Portal (EFS) component of the Data Spine facilitates the federated security and SSO capability for the EFPF ecosystem

Figure 2 depicts 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:

- **The Integration Flow Engine** component of the Data Spine provides a platform to the system integrators, allowing them to create integration flows for interconnecting the different APIs and services.
- **The Service Registry** component allows the service providers to register their services in the Data Spine. The Service Registry provides a facility for the service consumers or system integrators to discover these services and retrieve their metadata information required to create the integration flows.
- **The Message Bus** component can be used for mediating the transfer of messages or data between asynchronous services communicating through the Data Spine.
- **The EFS** component of the Data Spine is responsible for providing a SSO facility across the EFPF ecosystem (see API Security Gateway). In addition, the EFS component enables data integrity, security analytics, trust and reputation mechanisms, definition of policies and governance enforcement
- **The API Security Gateway** component of the Data Spine (and a sub-functionality of EFS) acts as the policy enforcement point (PEP) for the Data Spine and the platforms communicating through it. It intercepts all the traffic to the Data Spine and invokes the security service for authentication and authorization decisions

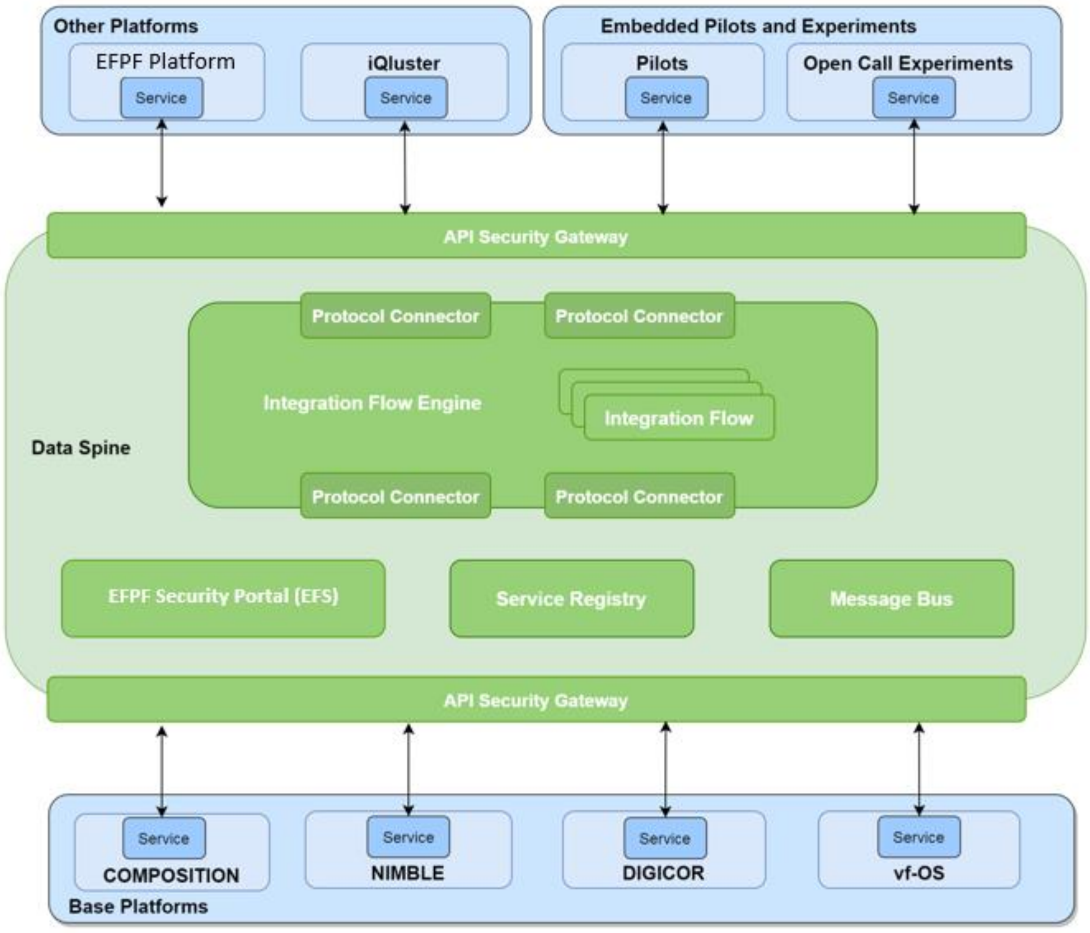


Figure 2: High-level Architecture of the Data Spine

1.2.2 Overview of EFPF Platform

The EFPF platform, shown in Figure 3, is a collection of smart tools and services designed for the EFPF ecosystem. These tools and services aim to cover the complete lifecycle of production and logistic processes that will be validated by the three cross-domain pilot scenarios brought forward by the project partners. Examples of the tools include e.g. Factory Connectors, IoT Gateways, distributed production planning and scheduling, distributed process design, monitoring, decision support, process optimisation, risk management and blockchain based trust and message exchange. In future plans (e.g. through open experimentation calls), the platform may also integrate services that play a crucial role in collaborative processes. Examples include technology services such as cloud storage, high performance computing, and value-based services, e.g. training, smart contracts, legal advice etc.

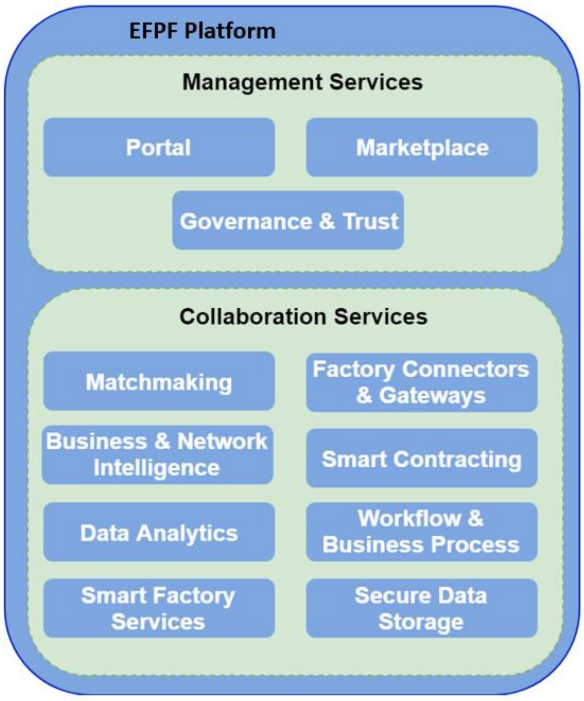


Figure 3: The EFPF Platform

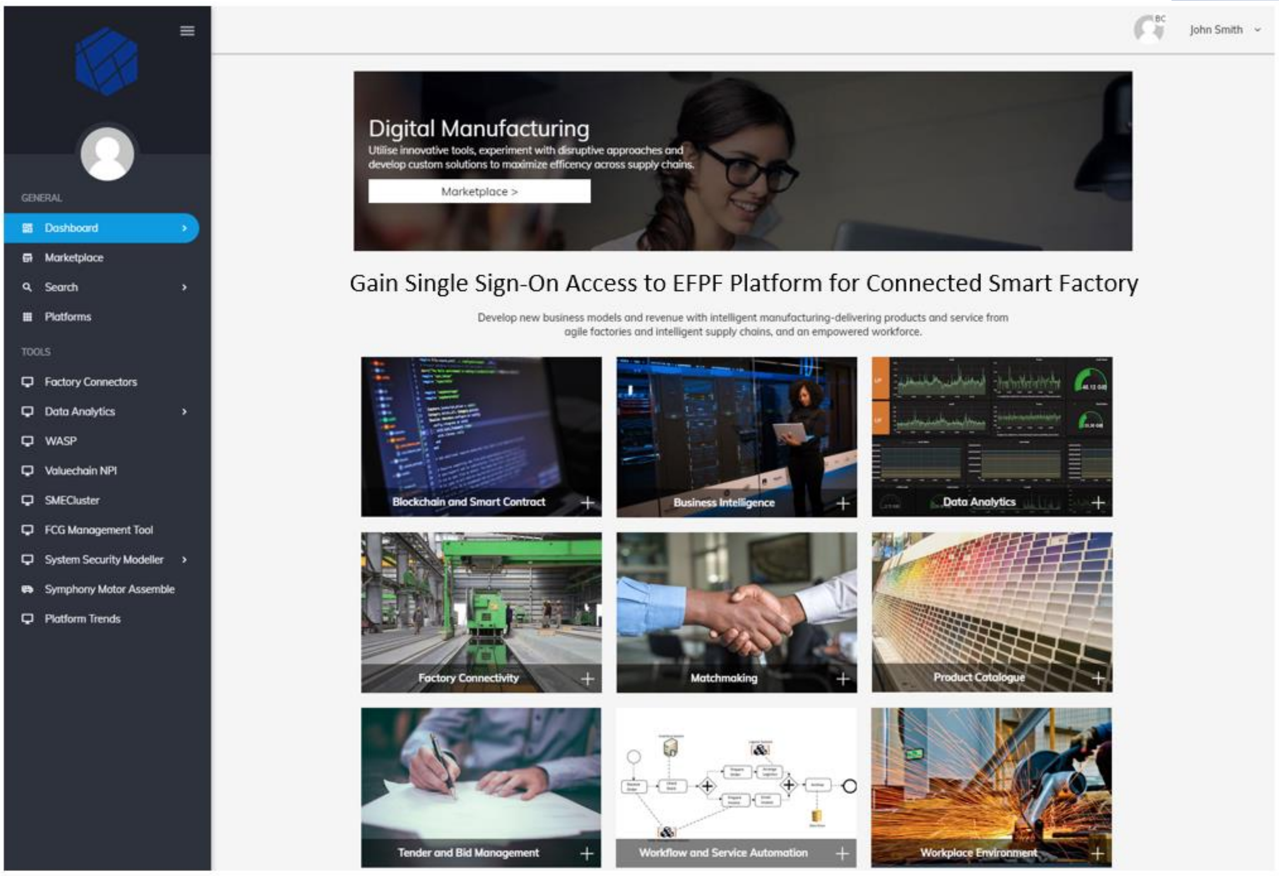
The EFPF platform aims to integrate market ready or reference implementations of the smart factory and Industry 4.0 tools from project partners. Figure 3 illustrates tools and services that are broadly categorised into two types:

- **Management Services**: These are the EFPF-specific services that provide federated or aggregate management capability of the services of different platforms and provide a coherent interface to the user, e.g. the Marketplace.
- **Collaboration Services**: These are the utility services with a concrete capability for realising collaborative processes in IoT, connected factory and automation environments, e.g. the Data Analytics services.

These tools and services of the EFPF platform are described in brief in the following sub-sections:

1.2.2.1 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://efpf-portal.ascora.eu/.



Digital Manufacturing

Utilise innovative tools, experiment with disruptive approaches and develop custom solutions to maximize efficency across supply chains.

Gain Single Sign-On Access to EFPF Platform for Connected Smart Factory

Develop new business models and revenue with intelligent manufacturing-delivering products and service from agile factories and intelligent supply chains, and an empowered workforce.

Figure 4: Snapshot of EFPF Portal Dashboard

After a login with valid credentials, the user will be forwarded to the portal dashboard (See Figure 4), which provides an overview of the value propositions of the EFPF platform. The dashboard design enables new users an easy start by selecting topics which are of interest. Each section provides additional information about a value proposition and existing solutions.

The portal consists of two parts: The GUI as shown in Figure 4 and a backend component, whose UML diagram can be seen in Figure 5. The backend component is providing communication to other EFPF platform components and services for logging events, user management and other required features.

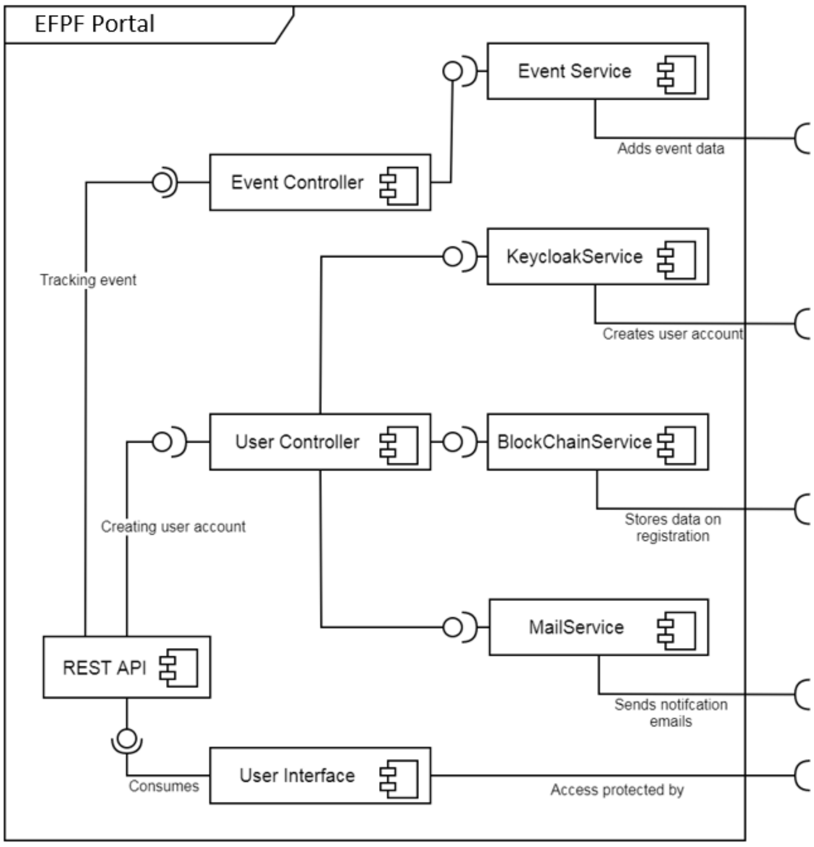


Figure 5: UML Component Diagram of EFPF Portal

1.2.2.2 Marketplace

The Internal Marketplace Framework provides access to items listed on marketplaces at different platforms provided by the EFPF partners. Additionally, its accountancy Service subcomponent provides features to track & trace and credit users of connected marketplaces.

Currently, products from the following external marketplaces are being displayed:

- NIMBLE
- SMECluster
- vf-OS
- WASP

Each product will be displayed with its name, its categories, its product image and a link, which leads to the detail product page at the external marketplace.

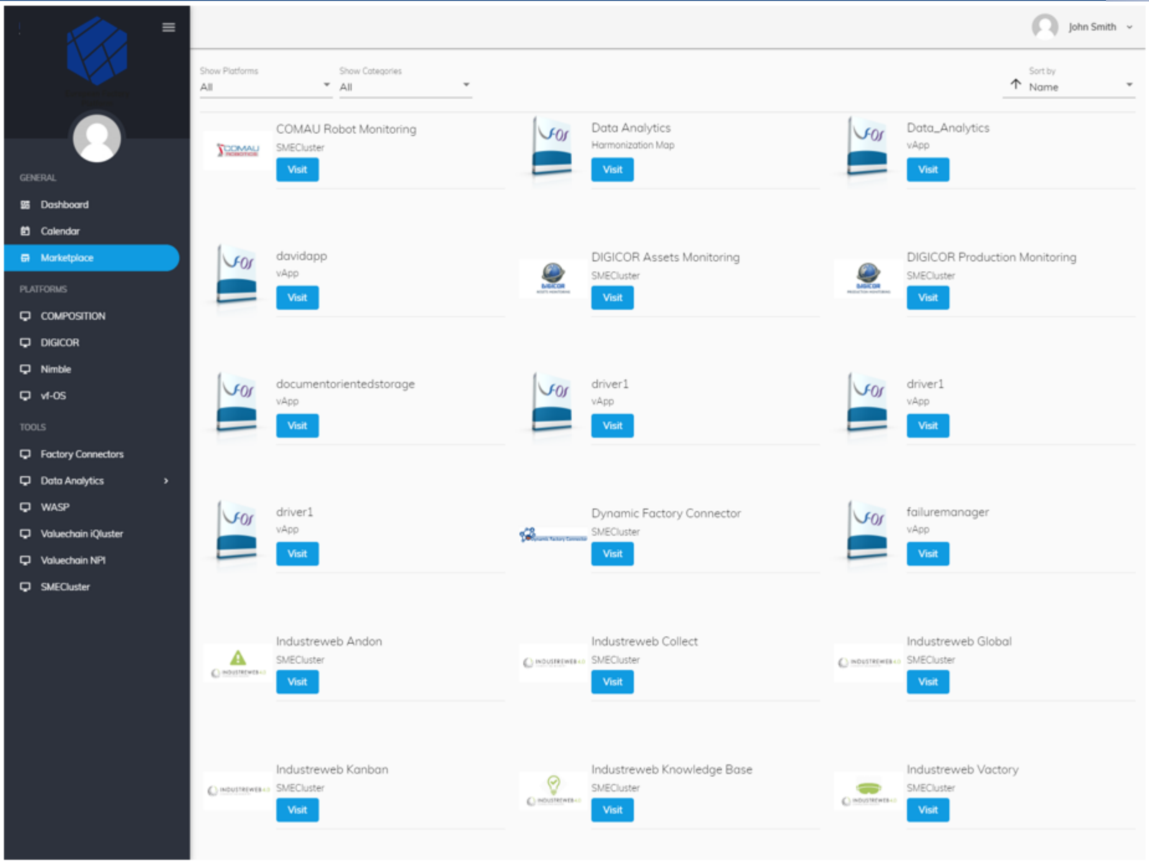


Figure 6: Snapshot of the EFPF Marketplace

The marketplace is currently existing as an Angular web component, which can be easily integrated in a website by including a JavaScript library and the Marketplace HTML element in the desired page.

Future changes will introduce a backend component, which will manage communication with other EFPF components as seen in Figure 7.

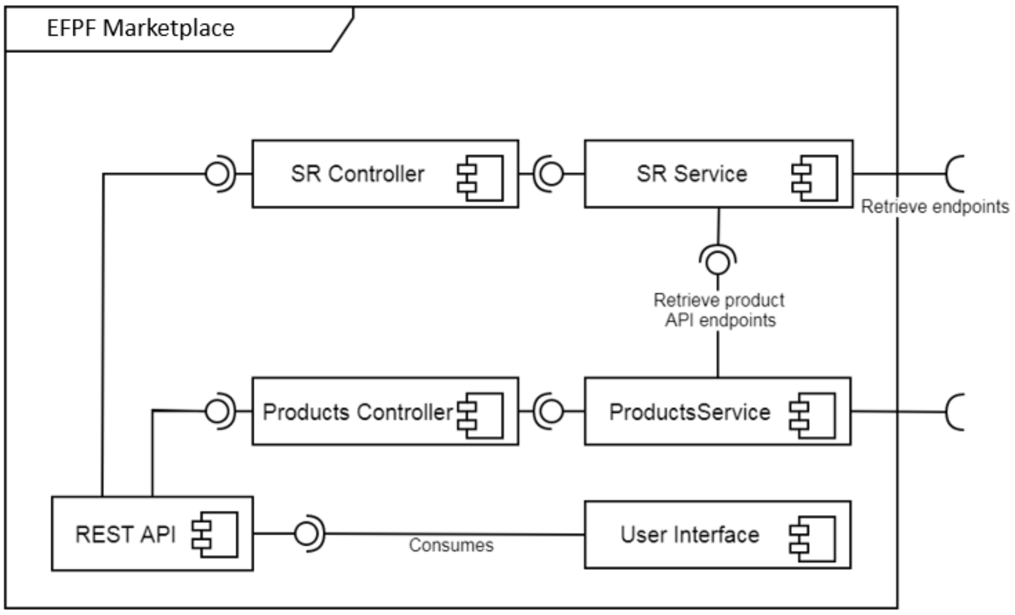
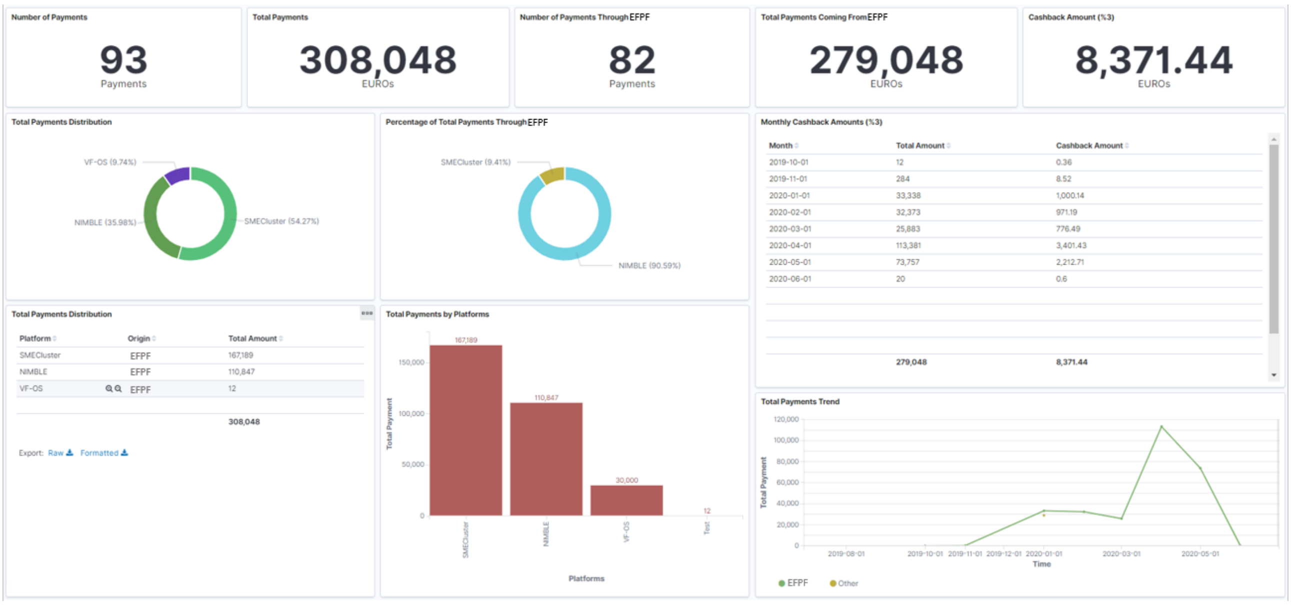


Figure 7: UML Component Diagram of Marketplace Framework

1.2.2.3 Accountancy Service



Total Payments Distribution

Platform	Origin	Total Amount
SMECluster	EFPF	167,189
NIMBLE	EFPF	110,847
VF-OS	EFPF	12
		308,048

Export: Raw Formatted

Monthly Cashback Amounts (%3)

Month	Total Amount	Cashback Amount
2019-10-01	12	0.36
2019-11-01	284	8.52
2020-01-01	33,338	1,000.14
2020-02-01	32,373	971.19
2020-03-01	25,883	776.49
2020-04-01	113,381	3,401.43
2020-05-01	73,757	2,212.71
2020-06-01	20	0.6
	279,048	8,371.44

Figure 8: Accountancy Service Dashboard

The Accountancy Service is developed as a part of the EFPF Marketplace Framework and provides insight into users' interactions with the EFPF platform, particularly any transactions that EFPF users make on different marketplaces, which are linked with the EFPF Marketplace Framework.

Tracking the user behaviour enables businesses to make productive decisions and develop effective business strategies. This is an important feature in the digital platform world, which is being used to support the long-term sustainability of the EFPF platform, beyond the span of the project. The Accountancy Service tracks and traces users' journey across the EFPF ecosystem and collects data about the transactions that EFPF users make on different marketplaces. The collected data is then used to carry out a cashback mechanism allows to charge a commission or a referral fee from the marketplace where the EFPF user carries out a business transaction (Figure 9). Note, the Accountancy Service does not collect personal and / or sensitive corporate data, instead the idea is to collect anonymised transactional data. In addition, the Accountancy Service contains a dashboard for the visualisation of the user behaviour.

A taxonomy is setup to identify the trackable user actions in which action items are listed in 'subject, verb, object' manner and these actions include users' basic interactions with the EFPF platform such as login, register, inviting other users as well as payments realized on external marketplaces if the user has initiated his/her journey from EFPF Marketplace.

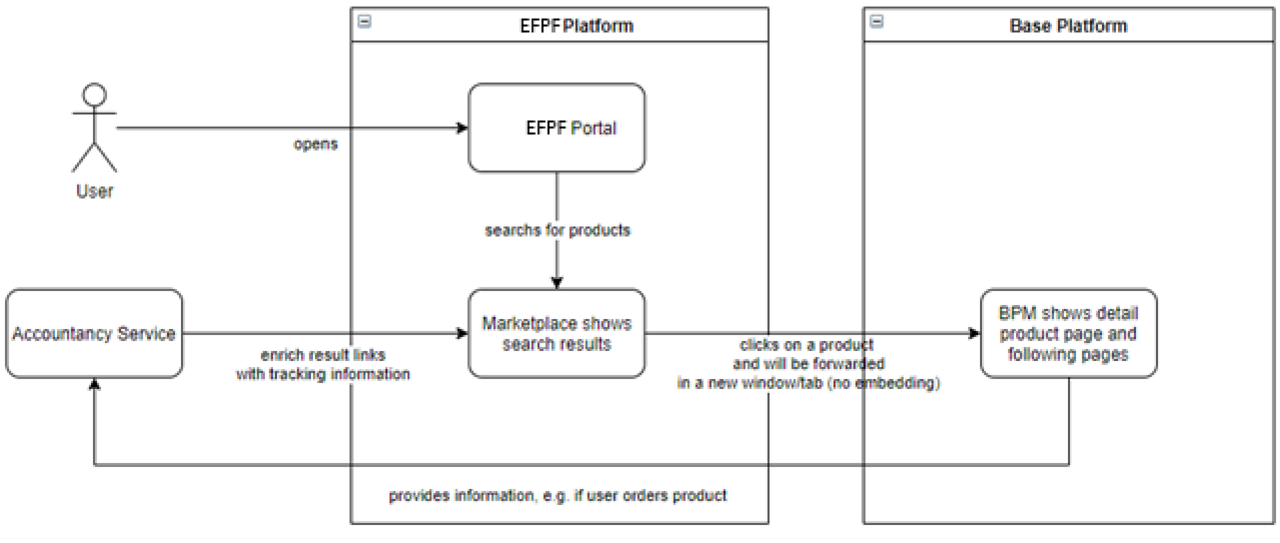

EFPF Platform
Base Platform
User
opens
EFPF Portal
searchs for products
Accountancy Service
enrich result links
with tracking information
Marketplace shows search results
clicks on a product
and will be forwarded
in a new window/tab (no embedding)
BPM shows detail product page and following pages
provides information, e.g. if user orders product

Figure 9: Cashback Process

The Accountancy Service is developed based on the Elastic Stack (Elasticsearch, Logstash, Kibana) with additional custom modules which are described below. The corresponding component diagram is illustrated in Figure 10:

- **Elasticsearch:** It 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.
- **Logstash:** It gathers user behaviour data from various components of the EFPF platform, executes different transformations and filters the content, before sending the data to the Elasticsearch component.
- **Kibana:** It enables interactive dashboards, filters and advanced data analysis and exploration of user logs.
- **Reporting Component:** This 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:** It 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

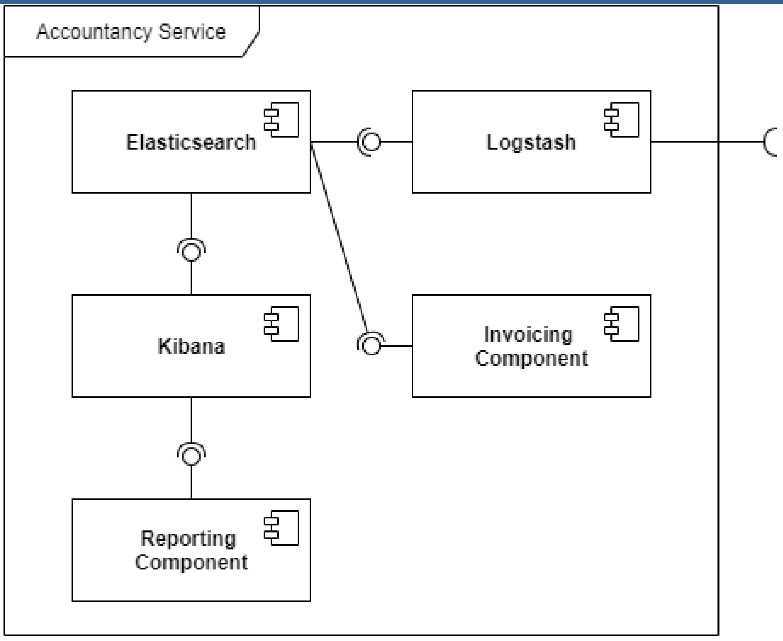


Figure 10. UML Component Diagram of the Accountancy Service

1.2.2.4 Matchmaking

In EFPF, several key manufacturing and smart factory tool platforms are interlinked as a federation of digital manufacturing platforms. Namely, they are NIMBLE, COMPOSITION, DIGICOR (represented by SMECluster) and vf-OS. In EFPF, these platforms offer different types of manufacturing and smart factory solutions. With an effective matchmaking strategy these platforms can offer their products and services to a wider client audience through a unified EFPF portal with value added features.

The goal of matchmaking in EFPF is to facilitate EFPF users to find the best suited suppliers and enable them to transact with them efficiently and effectively. This is achieved through 4 layers of matchmaking in EFPF platform;

- Federated search of participants (suppliers/service providers) & their value-units (products/services)
- Platform recommendations of suppliers/service providers & products/services
- Navigate users to perform negotiations and transactions with selected suppliers and service providers from different base platforms
- Enable users to find the best supplier to fulfil a request for a service or product in a fully automated way (via automated agents)

Figure 11 represents the component architecture of matchmaking in EFPF.

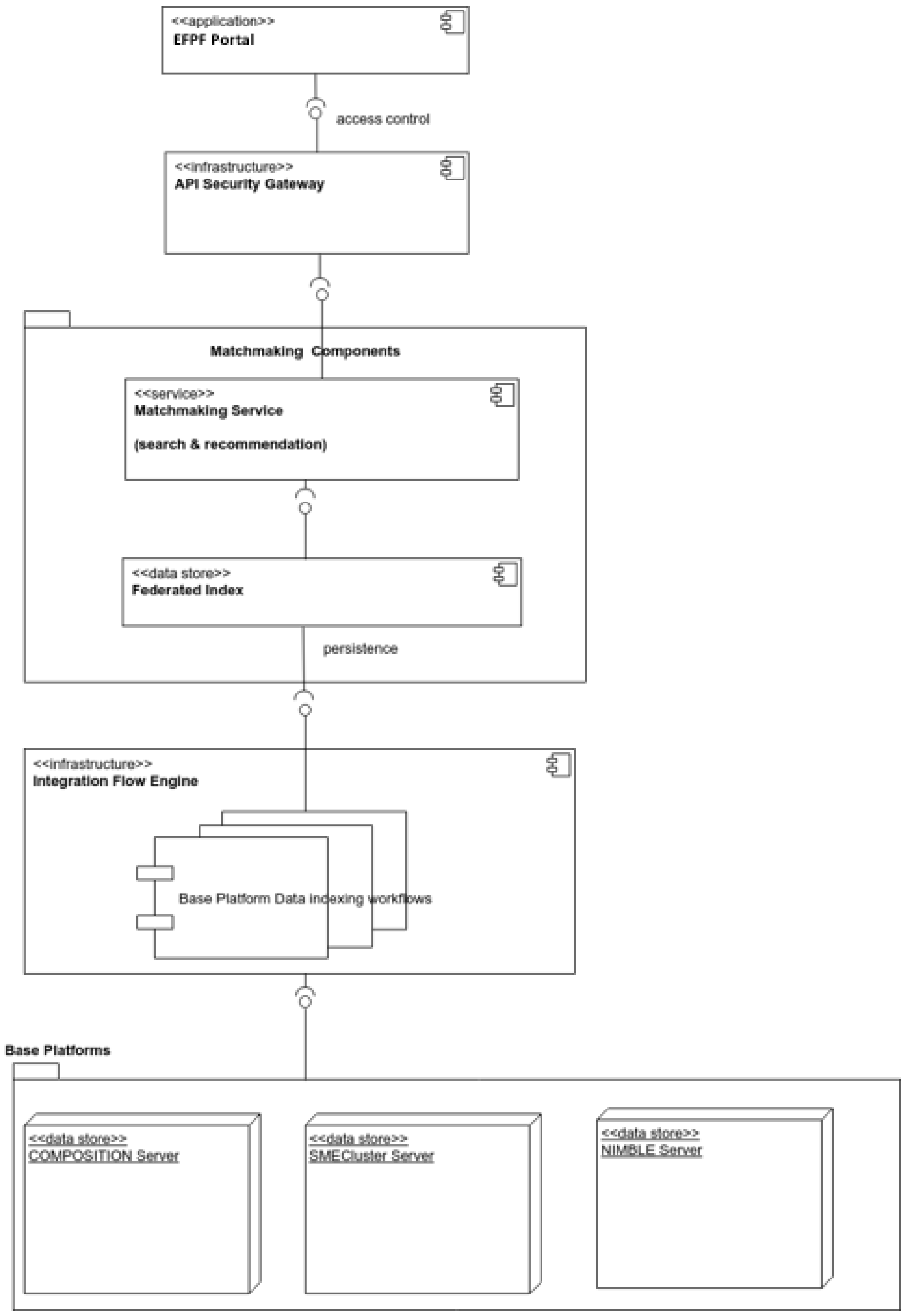


Figure 11: UML Component Diagram of Matchmaking Framework

The main components of the service can be briefly described as follows:

- **Base platform data indexing workflows**: The data stores in base platforms representing data of products/services and service providers are synced with the federated matchmaking index via these workflows. These workflows are deployed in the Data Spine's integration flow engine
- **Federated index**: The federated data index for matchmaking services such as federated search and recommendation services on EFPF.

Matchmaking service: The main service providing search & recommendation functionality to the EFPF users. This service is integrated in the EFPF portal to provide federated search functionality. The Matchmaking API specification can be found here: https://efpf-security-portal.salzburgresearch.at/api/index/swagger-ui.html

The architecture of the Matchmaking service in EFPF is further described in D5.1: EFPF Matchmaking and Intelligence Gathering.

Matchmaker for Online Bidding Process

The matchmaker component is a rule-based matchmaking engine amplified with multi-criteria algorithms for offers' evaluation. It supports semantic matching in terms of services, products and business entities' capabilities at the EFPF platform and enables automated real time bidding. The Matchmaker's functionality surpass the concept of search of services and products and realize an automated matchmaking process for negotiations and business transactions between interested stakeholders.

Figure 12 depicts the Matchmaking Semantic Framework's different components in a UML diagram. The Matchmaker base is the Ontology store which provide the other components with the Ontology information by handing over Ontology models or by receiving SPARQL queries. The Agent Level and Offer Level Matchmaking modules implement the core functionalities of the framework. They both operate using the Semantic Rules module which executes rules and queries and achieve explicit matching between Requesters and Suppliers. The Offer Level Matchmaking module additionally uses the Weight Assessment module in order to extract the best matching offer according to the request. The Ontology Querying module achieves the retrieval of the requested Ontology information. A RESTful API is exposed on top of all framework's modules and accomplishes the external communication of Marketplace Agents with the Matchmaker API.

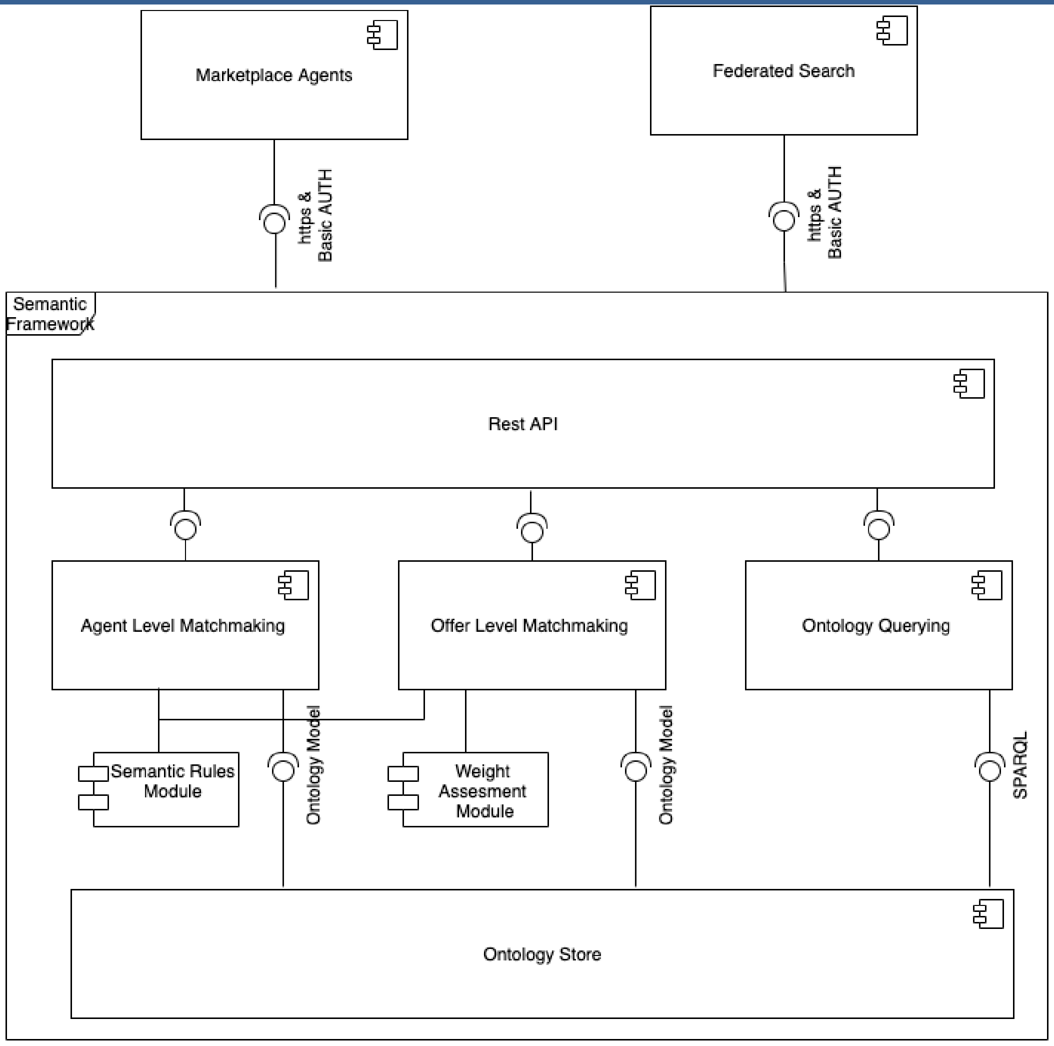


Figure 12: Matchmaker API UML Component Diagram

1.2.2.5 Governance & Trust

The EFPF is an emerging ecosystem of multi-sided digital platforms. This platform ecosystem requires effective governance mechanisms to be put in place, to reach its major goals and create sustainable outcomes. The governance mechanisms for digital platform ecosystems need to 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. The interests and legal roles require a balanced interplay and understanding of interdependencies between all stakeholders, collaborating in such an ecosystem. In other words, to stimulate positive interaction payoffs within the platform ecosystem, both platform stakeholders and platform technology enablers must be regulated and governed.

Figure 13 illustrates the design of EFPF governance framework.

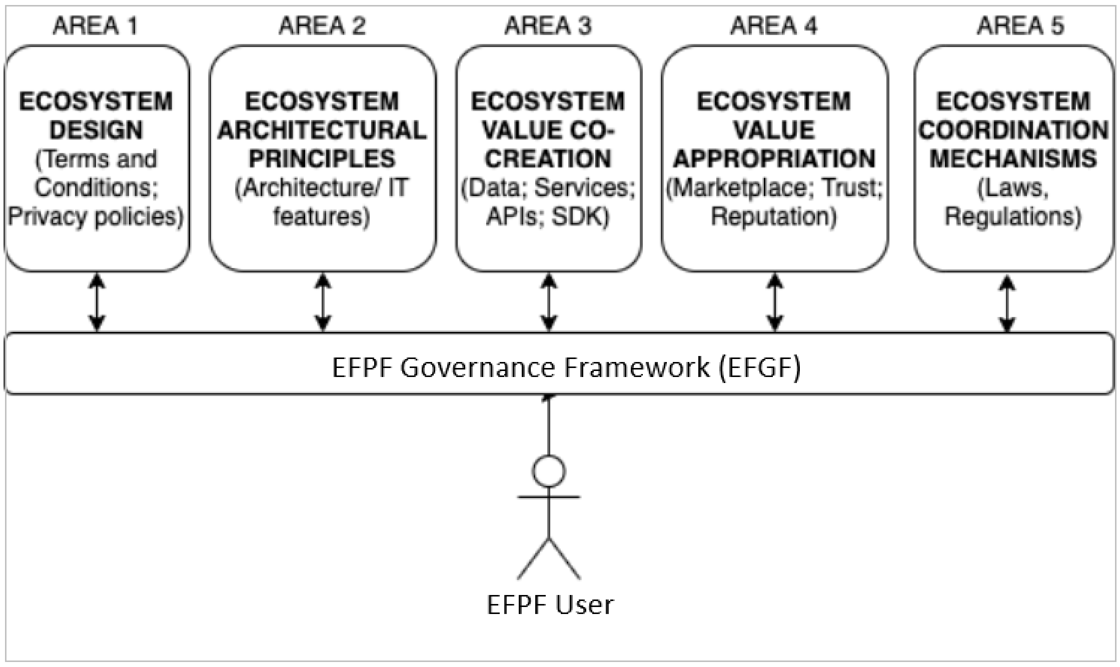

AREA 1
ECOSYSTEM DESIGN (Terms and Conditions; Privacy policies)
AREA 2
ECOSYSTEM ARCHITECTURAL PRINCIPLES (Architecture/ IT features)
AREA 3
ECOSYSTEM VALUE CO-CREATION (Data; Services; APIs; SDK)
AREA 4
ECOSYSTEM VALUE APPROPRIATION (Marketplace; Trust; Reputation)
AREA 5
ECOSYSTEM COORDINATION MECHANISMS (Laws, Regulations)
EFPF Governance Framework (EFGF)
EFPF User

Figure 13: UML Component Diagram of Marketplace Framework

More details about the EFPF governance framework can be found on the Deliverable D5.2: EFPF Security and Governance.

1.2.2.6 Business & Network Intelligence

The business and network intelligence task focused on identifying and catering to the applications of this service for different levels of engagement of a company. To simplify it further, the partners have decided to compartmentalise the tools and services under this task into three distinct levels:

Intracompany Level

This level aims to generate actionable intelligence from activities and processes within a company. This category is to do with generating and sharing intelligence for work done within the organisation.

The tools/services introduced under this category are focused on helping manufacturers use digital tools that can be accessed through EFPF platform, are simple to use and provide key insights towards a certain goal. A few examples are given below:

- **Production KPIs**

  A must-have metric of any manufacturing operation are Key Performance Indicator. One of the most recognised is Overall Equipment Effectiveness (OEE) which measures the overall performance of a given machine, product line or work centre.

  OEE is calculated using the formula of Availability * Performance * Quality and is considered the most reliable metric for tracking production performance to the machine and plant floor level.

  This can be used to share Overall Equipment Effectiveness between industry members. This will offer the opportunity to not only track specific work centre performance but can also track average industry performance and see whether there are any short comings or where you excel.

- **Waste Monitoring Intelligence**

  Waste monitoring gives businesses the opportunity to automatically track bin fill levels and compare local suppliers to find the best price.

  This will offer benefits for both customer and supplier. Suppliers will be able to track businesses within a perimeter that all require the same waste disposal. Whereas, customers will have the opportunity to monitor waste levels year-round and identify parts of the calendar year where waste levels may be higher.

- **Factory Environment Monitoring**

  Manufacturing effectiveness can be dependent on many factors. To assist with managing any extenuating circumstances that would affect production, the factory environment can be tracked.

  A great example of this in practice would be within aerospace industry where temperature effects metal with thermal expansion, to reduce any issue with incorrect parts live data could be married with predictive weather and forecasted temperature data.

Platform/Network level

Aims to generate intelligence from the existing networks that a company is already involved in as part of their supply chains and other business activities. This category is to do with analysing all work that goes on in the existing networks of a company.

The solutions introduced under this category are briefly explained below:

- **Platform & Network Intelligence Solution**

  The Platform & Network Intelligence solution collects data around the usage of and traffic flow within the EFPF platform. Through the collection and analysis of both the search events and login events that take place within the platform, the insights gained from the analysis are visualised and presented within a dashboard in the EFPF portal. This enables users to gain intelligence not only about the trends within the EFPF platform, but also within its ecosystem of connected platforms through the Federated Search feature.

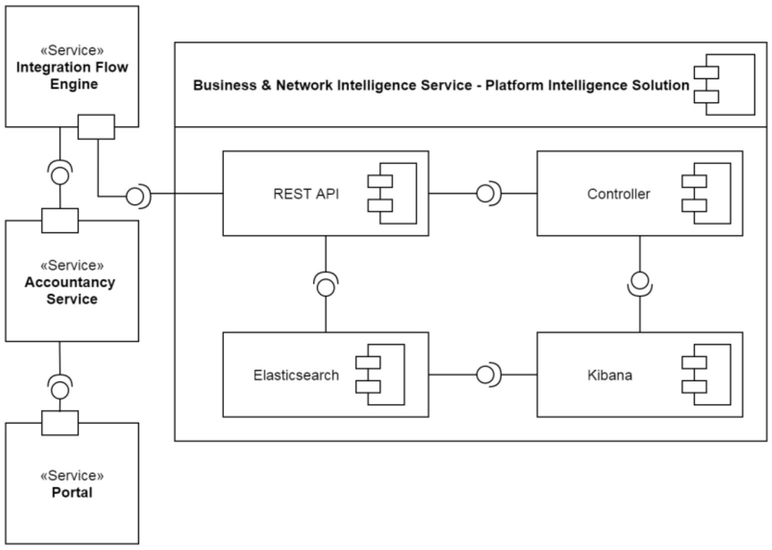

«Service» Integration Flow Engine
Business & Network Intelligence Service - Platform Intelligence Solution
REST API
Controller
«Service» Accountancy Service
Elasticsearch
Kibana
«Service» Portal

Figure 14: Architecture of Platform & Network Intelligence Solution

- **iQluster Platform**

  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, iQluster 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. This creates a direct benefit to for each user and the benefit compounds as we move up the tiers of supply chain as this data aggregates and reflects unparalleled intelligence that is otherwise fragmented or unavailable.

  iQluster combines the conventional 'top down' supply chain mapping with an innovative 'bottom up' approach to address the gaps. iQluster's data science expertise creates a great take off platform by big data scrapping to identify key intelligence about companies in supply chain. Then, the suppliers are invited to join the platform and a unique incentivisation model is used to boost adoption rates

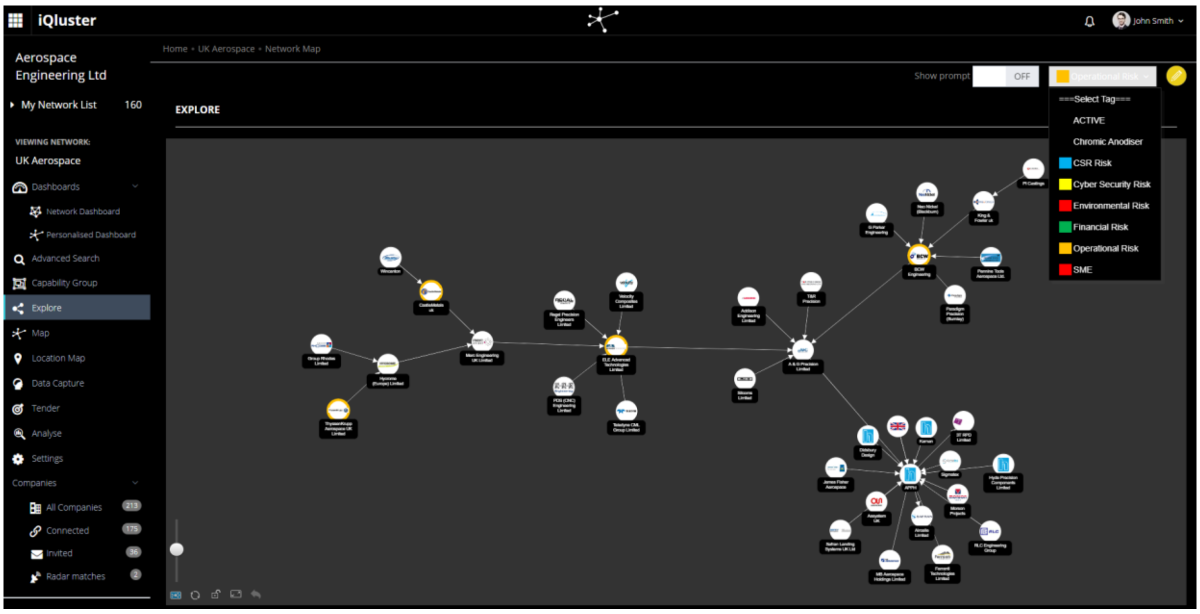


Figure 15: iQluster supply chain intelligence platform's Explore Map

- **Tendering and Bid Management**

  Tendering and Bid Management or Business Opportunity is a platform that will give SMEs a chance to offer their services for income generating opportunities. This will be achieved through three main elements: a company directory profile; a business opportunity board – where you can apply for tenders or search for relevant business opportunities; along with a messaging system. This will allow procurers to find prospective supplies, post opportunities and contact suppliers all within one portal.

  Within the functionality to search both the directory and business opportunities, we will offer insight for businesses on the current marketplace. Data will be gathered and represented, so that supplies can see the requirements of procurers. For example, if procurers are regularly looking for suppliers that hold specific accreditations (i.e. QS 9000) suppliers will be able to make an informed decision on what is necessary for their business development. This will also offer the opportunity to observe the market and give the opportunity to win business better.

Market Intelligence

Finally, for the residual factors that are not covered in internal workings and that of other connected companies through their network, a market intelligence tool (Figure 16) is being introduced. This service aims to utilise advance data engineering and ML (machine learning) over openly available public data streams such as national company registers, websites, financial and credit rating databases etc. to provide an insight in to the movements in market conditions.

The task partner VLC have proposed to introduce this standalone application directly onto the EFPF platform. While the development of this service is still in progress, this section will share the specifications and scope of this application.

The intended application for the platform users could be to use this service to find potential customers and suppliers that they did not know of and to explore working with them to expand their business. Some other methods are being explored where a user could benefit from getting a demographic summary of the companies in a region for example, to understand "how many competitors or potential customers are there in this region?" and "how should I strategize to enter a strong growth sector with big presence in my region?"

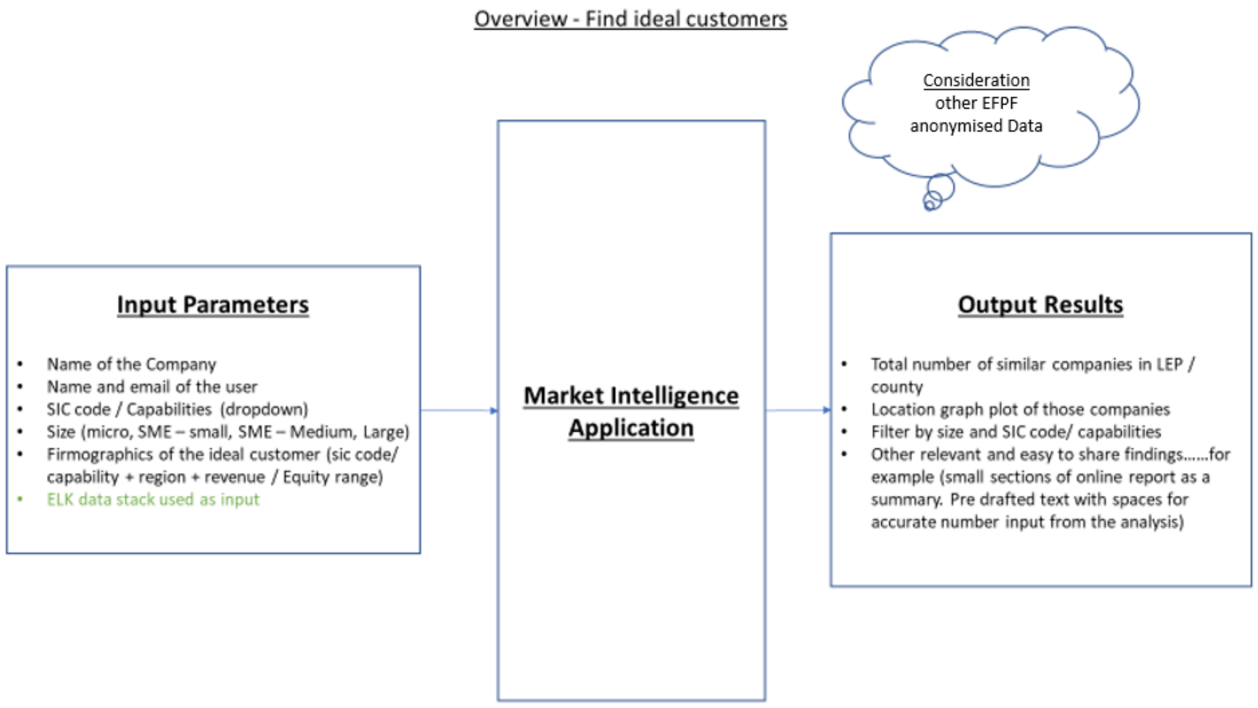

Overview - Find ideal customers
Consideration other EFPF anonymised Data
Input Parameters
Name of the Company
Name and email of the user
SIC code / Capabilities (dropdown)
Size (micro, SME – small, SME – Medium, Large)
Firmographics of the ideal customer (sic code/ capability + region + revenue / Equity range)
ELK data stack used as input
Market Intelligence Application
Output Results
Total number of similar companies in LEP / county
Location graph plot of those companies
Filter by size and SIC code/ capabilities
Other relevant and easy to share findings……for example (small sections of online report as a summary. Pre drafted text with spaces for accurate number input from the analysis)

Figure 16: Sample SIPOC diagram from spec document – Market Intelligence Application

1.2.2.7 Smart Contracting

The EFPF project aims to provide a smart factory ecosystem as an open platform to manufacturing and logistics companies, while incorporating innovative approaches for solving industrial problems. The Blockchain and Smart Contracting component provides a trusted system for smart contracting agile networks in the EFPF ecosystem. This component is based on the distributed ledger technology systems provided by the base platforms.

Distributed ledger technology (DLT) is a suitable design mechanism [KRU04] for persistence when the system is situated in an environment where three main conditions exist: 1) a distributed, immutable log of transactions is needed, 2) there is a need for distributed trust and 3) there is an incentive to manipulate data. Major design concerns for the architecture of the Blockchain and Smart Contracting service were:

- The Blockchain Smart Contracting service and the chosen implementation mechanism (blockchain framework) should be a private[3], not a public distributed ledger technology system.
- The Blockchain Smart Contracting service and the chosen implementation mechanism (blockchain framework) should have the capability to be a permissioned distributed ledger system, requiring authorization to perform a particular activity or activities.
- Due to the nature of EFPF business networks, the Blockchain Smart Contracting service and chosen implementation mechanism should have the capability to be private, i.e. accessible for use only to a limited group of DLT users, a consortium.
- Sector Agnostic – the solutions should be usable by cross-sectorial stakeholders e.g. in production, distribution and by customers.
- The Blockchain Smart Contracting service should be a federation level solution, no single entity will own the process of Blockchain.
- All stakeholders will be able to access and use the Blockchain Smart Contracting service.

Use cases

The design process towards the blockchain and smart contracting service is bottom-up, solidifying and merging the existing base platform approaches into EFPF, while taking into account the core project objectives and user requirements. Use cases have been selected and explored to build a sectoral agnostic, distributed service accessible by all EFPF stakeholders.

From exploration of a selected pilot applications, technology and solutions will be tested and evaluated. Functionality and design patterns in the different scenarios will be generalized to support arbitrary business processes and provide re-usable blockchain application building blocks and services. The goal is to provide a blockchain platform that can support value co-creation in the EFPF ecosystem, with stakeholders extending the existing functionality in a modular way.

After a thorough analysis of the EFPF digital manufacturing platform ecosystem, we identified the following areas that could benefit from the use of blockchain technologies [VIDR18]:

- **Identity Management**: Offering unique identities to authenticate and authorize participants of the network
- **Track & Trace**: Enabling historical records with information including provenance data and an audit trail of the different assets, chronologically ordered
- **Permission Handling**: Supporting different access rights, depending on each user's role and organization
- **Contract Management and Procurement**: Supporting smart contracts to arrange the required logistics such as payment, shipping, etc.

[3] https://www.iso.org/obp/ui/#iso:std:iso:22739:dis:ed-1:v1:en

- **Quality Control and Accountability**: Adding a unique tag to every transaction that takes place in the ecosystem to achieve new levels of quality assurance and accountability
- **Scalability**: Facilitating the acquisition of new partnerships among the manufacturers by utilizing cryptocurrencies
- **Authenticating IoT**: Enabling identities for remote devices
- **Tamper Proofing**: Giving each record a verifiable date and time
- **Encryption**: Ensuring the data transmitted over a public network
- **Protecting intellectual property management and control**

Identity Management and a Supply Chain Mobile app using sensor data to provide provenance and tamper proofing have been developed and are described in the deliverable D3.1. At the time of writing, further pilot scenarios to apply the Blockchain and Smart Contracting platform have been selected and are under development together with pilot partners:

- Circular Economy
- Track and Trace
- Shipping DApp

These pilot scenarios are described in more detail in D5.2.

Design of Blockchain-based Applications

In EFPF, the use cases are concentrated around businesses with private data, thus excluding the use of public blockchain solutions. By following market best practices, we had to select among well-established blockchain frameworks. After a thorough research, we concluded that private and consortium-based frameworks, e.g. Hyperledger Fabric and Hyperledger Sawtooth with Hyperledger Grid support already implement Proof of Concepts and provide out of the box solutions for value chain and supply-chain use cases in EFPF. The design decision was taken to use Hyperledger Sawtooth as the primary blockchain implementation mechanism. On this implementation mechanism, we can build private, permissioned blockchain applications that are not limited in domain and accessible to all stakeholders. Consensus nodes can be hosted by a subset of the partners in the business network. No single entity needs to own the blockchain.

The decision was influenced by the following characteristics of the Hyperledger Sawtooth:

- Separation Between the Application Level and the Core System
- Private Networks with the Sawtooth Permissioning Feature
- Parallel Transaction Execution
- Event System
- Ethereum Contract Compatibility with Seth
- Dynamic Consensus
- Sample Transaction Families
- Real-world Application Examples.

By further exploration of blockchain and DLT state-of-the-art tools, we have noticed a framework with rapidly growing support, which could be of interest to our applications in EFPF. This framework is called DAML (Data Asset Modelling Language)[4] and represents a 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[5] and Hyperledger Fabric. 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. It will be further evaluated and considered for use in EFPF, as it is open source (Apache 2.0) and compatible with the design decision to use Hyperledger Sawtooth. In addition, DAML promises a business oriented, declarative way to build distributed applications using blockchains.

1.2.2.8 Data Analytics

During the manufacturing activities – from production to supply chain enactment – a huge amount of data is generated; some of this is a well organised and stored for utilisation, but much more is not captured or not utilised and thus the potential value is not extracted. With advanced big data analytical techniques, the insights locked in the data captured from manufacturing activities can be unleashed and used to not only improve the development, production and supply processes, but also transform the business aspects. In this respect, the data analytic solutions gathered in the EFPF platform are representative of the needs of manufacturing companies for better visibility into distributed processes, better understanding of complex problems and better investigation of optimisation potential.

The data analytic solutions in EFPF are either the enhancements of the features that are already validated in the base platforms or are developed to satisfy the specific needs and requirements of manufacturing users in the EFPF ecosystem. Overall, the data analytic solutions in EFPF include Machine Learning based anomaly detection solutions, deep learning toolkit to analyse for price forecasting of materials, shop-floor monitoring solutions and customer behaviour analysis solutions. The overview of these solutions is available in the EFPF deliverable D4.1: Smart Factory Solutions in the EFPF Ecosystem.

Visual and Data Analytics Tool

The visual and analytics tool provides different type of analytic solutions for both predictive maintenance and supply chain optimization activities alongside with advanced visualizations. It is a web-based tool that is accessible through EFPF portal only to authorized users. The tool is able to analyse and visualize historical data that are stored in its internal data store. Furthermore, the tool is able to analyse or visualize data coming from sensors on real time by using a factory connector based on MQTT protocol. Besides this, the tool supports HTTP connectivity as well. The next figure highlights the main internal components of the tool and its interactions. The analytic functionalities and the user interfaces of the tool are described in D4.1.

[4] https://daml.com/
[5] https://github.com/blockchaintp/daml-on-sawtooth

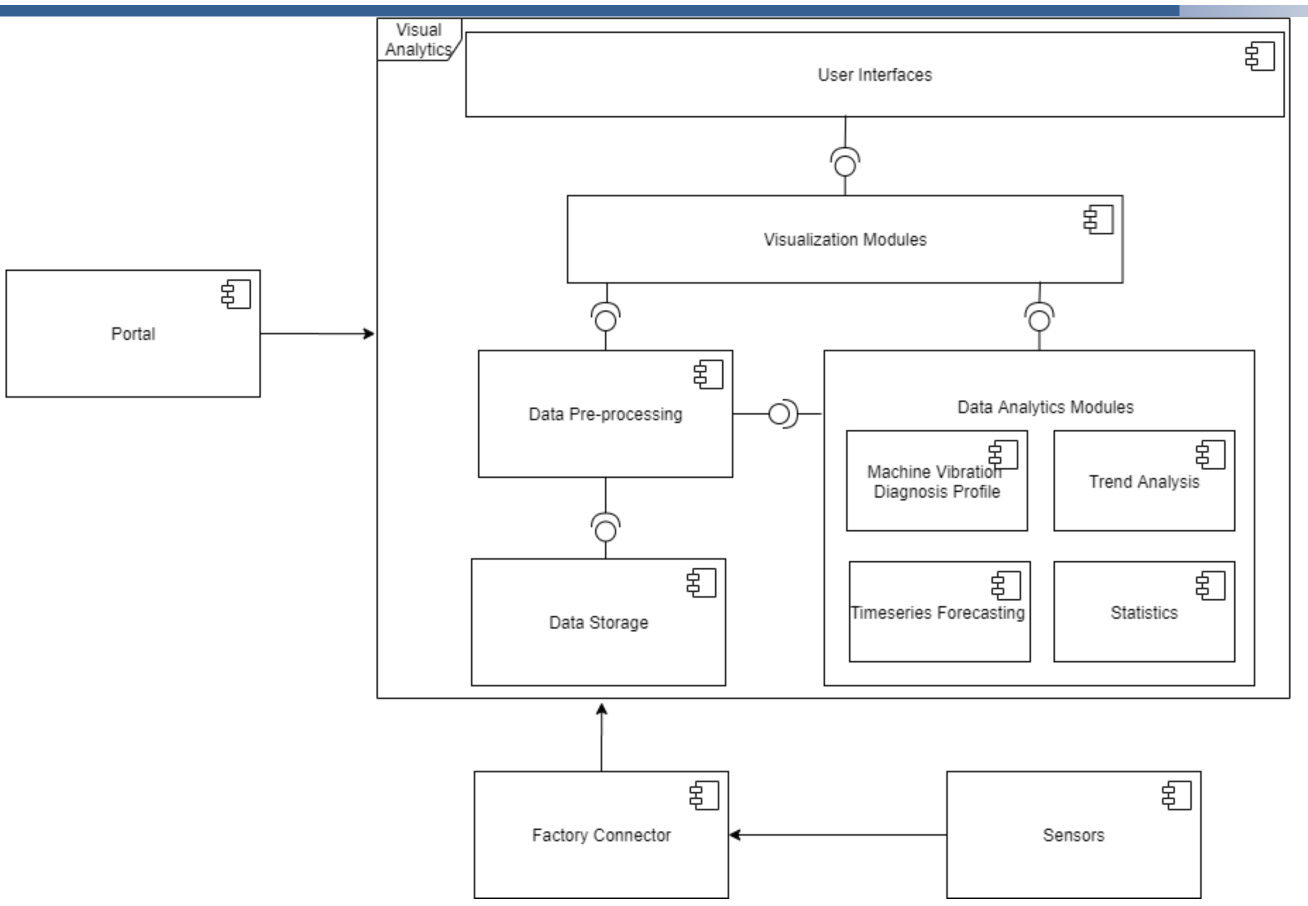


Figure 17 Visual and Data Analytics Tool UML component diagram

1.2.2.9 Workflow & Business Process

The Workflow and Business Processes automation services in the EFPF platform is provided by WASP (Workflow and Service Automation Platform). WASP offers a solution for fast automation of internal or distributed processes regardless of their nature e.g. whether it is an internal process that interlinks several internal activities; or a distributed process with multiple tasks performed by different stakeholders. WASP is able to interlink and orchestrate the automated services, manual services or the combination of both.

Offered as a Cloud based service with an intuitive GUI, WASP allows users to design, execute and monitor multiple processes where each process may be composed of several activities that are either performed manually (and status updates are provided through human interfaces) or are executed automatically through web-services. In the EFPF platform, WASP is accessible through the EFPF portal using the EFPF Single-Sign-On security features.

1.2.2.10 Secure Data Store

Secure Data Storage facilitates connecting sensor data with analysis tools, by collecting data over time for latter access by analysis tools, while allowing data owners the explicit control of how to share data with the analysis tools. Fine grained data management controls are used to authorize data access, with pseudonymization techniques to protect data at rest. Secure Data Storage consists of a collection of Docker images, which are intended to be deployed by data owners on their own premises. While the exact configuration is flexible, parallel deployments of the API server and dependent databases is assumed.

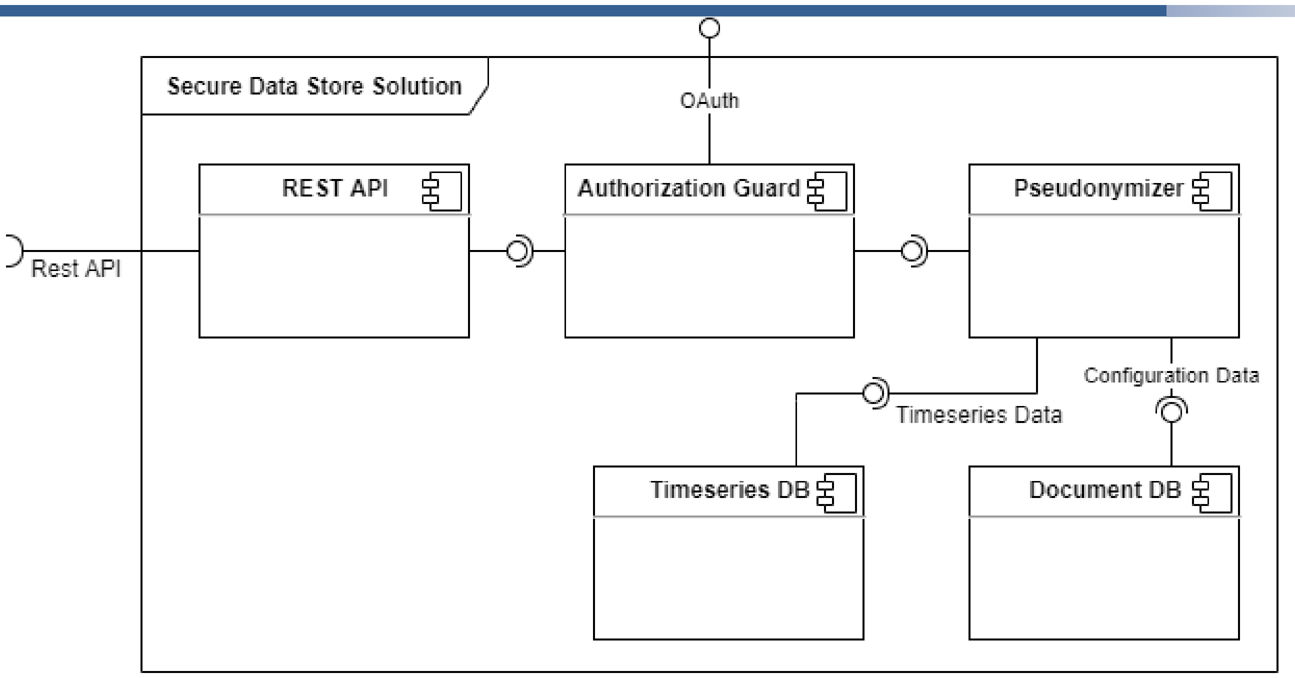


Figure 18: Secure Data Store Architecture

1.2.2.11 Smart Factory Tools & Services

The EFPF ecosystem makes available a diverse catalogue of Tools and Services from the base platforms that can be used by end users to deal with a range of factory applications. Tools can be used in isolation or combined by interfacing them using the Data Spine functionality in order to create smart factory composite solutions.

1.2.2.11.1 Data Model Transformation Tool Suite

In state-of-the-art open source enterprise integration platforms, performing data model/schema transformation is still a manual task. Therefore, it makes the service integration process slow, tedious and expensive. Also, the semantics of the data to be transformed and other service metadata required for writing the schema mapping rules are often not well-documented and hence this process becomes error-prone too.

The objective of the Data Model Transformation Tool Suite is to address the problems listed above. As shown in Figure 19, the tool suite contains semantic repositories to facilitate the lifecycle management of functional as well as technical metadata of services based on standard vocabularies and the lifecycle management of these standard vocabularies. In addition, it provides a Data Model Transformation (DMX) Tool that guides the system integrator for writing the mapping rules in a semi-automated way, thereby reducing the human involvement to make it faster, easier and more efficient. The tool suite is currently in conceptualization phase and a detailed description of it would be included in the deliverable D4.2.

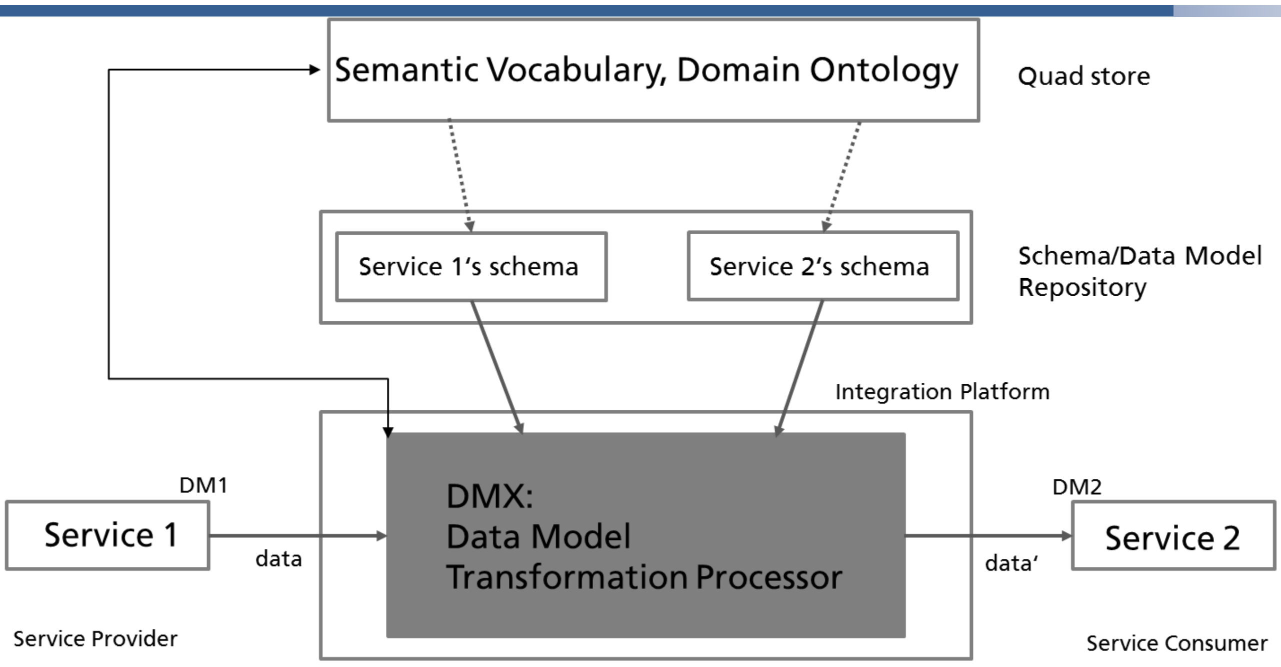


Figure 19: Conceptual Components of Data Model Transformation Tool Suite

1.2.2.11.2 Blockchain Framework

This blockchain framework from base platform COMPOSITION has been redesigned and modified to use the blockchain implementation chosen for EFPF, Hyperledger Sawtooth. The mobile DApp has been made multi-platform and is being adapted to EFPF pilot needs. To further strengthen the integration with EFPF the shipping process is being reengineered to work with a BPMN process engine, which will allow the user to configure the process in the WASP tool.

To deploy the shipping DApp, a peer-to-peer network of blockchain nodes needs to be set up. There are already nodes set up that is currently used by the DApp, new nodes can attach to this network or build a new private consortium blockchain. The first version used Multichain 2.0 nodes, while the next update one will use Hyperledger Sawtooth as the other EFPF blockchain applications. The DApp needs to be deployed to the mobile unit (iOS or Android) through the development IDE. At the time of writing, it is not published to App Store or Play.

The key feature of the mobile app is to use physical interaction to address weak points or loopholes where to manipulate the supply chain. We use added sensor data and biometric identification to corroborate the transaction data (e.g. a receipt of received goods). As the data is a representation of a transaction taking place in the physical world and entered into the blockchain, it cannot be validated by the blockchain transaction rules or smart contracts directly. To circumvent this reliability problem of physical interfaces we add other metadata such as NFC tags, weight, images and sign in with face or fingerprint id to provide other evidence that the event took place as described.

Figure 20 shows the architecture of the blockchain framework.

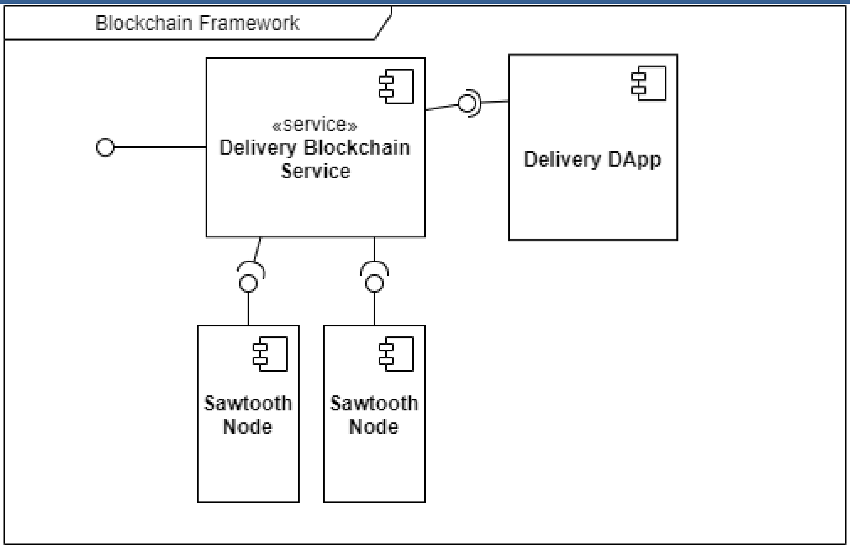


Figure 20: Architecture of Blockchain Framework

1.2.2.11.3 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. Industreweb Global contains Administration Tools that can be used for a variety of tasks, these include:

- Edit Industreweb Collect node configurations
- Manage, create, and edit Industreweb Display programs
    - Authoring Collect Programs is done using a drag and drop interface based upon Google Blockly.
- Manage, create, and edit Industreweb Display screens
    - Authoring and editing Display screens is performed using a responsive bootstrap drag and drop UI which includes a palette of screen components, which can be dragged onto the page
- Manage user logins for Administrators and Users to view screens
- Configure settings on back end services to affect systems functionality

IW Global also offers solution specific Industreweb Applications to be used, these tools include:

- IW Display
    - Visualisation tool designed to display product information such as production faults, production performance, waste and historical reports
- IW Vactory
    - Makes data available in a WebGL 3D UI that maps data points to a digital twin of the production assets.
    - Can be used to monitor production processes in a more intuitive way

- Work Instructions
  - Allow screens to be displayed to users based on the operation that they need to perform
- Error Proofing
  - Allows for operator error scenarios to be modelled so that when they arise, the machine can either display a corrective message or the process can be inhibited
- Fault Escalation
  - Allows alerts to maintenance staff when an error arises, for corrective action to be taken

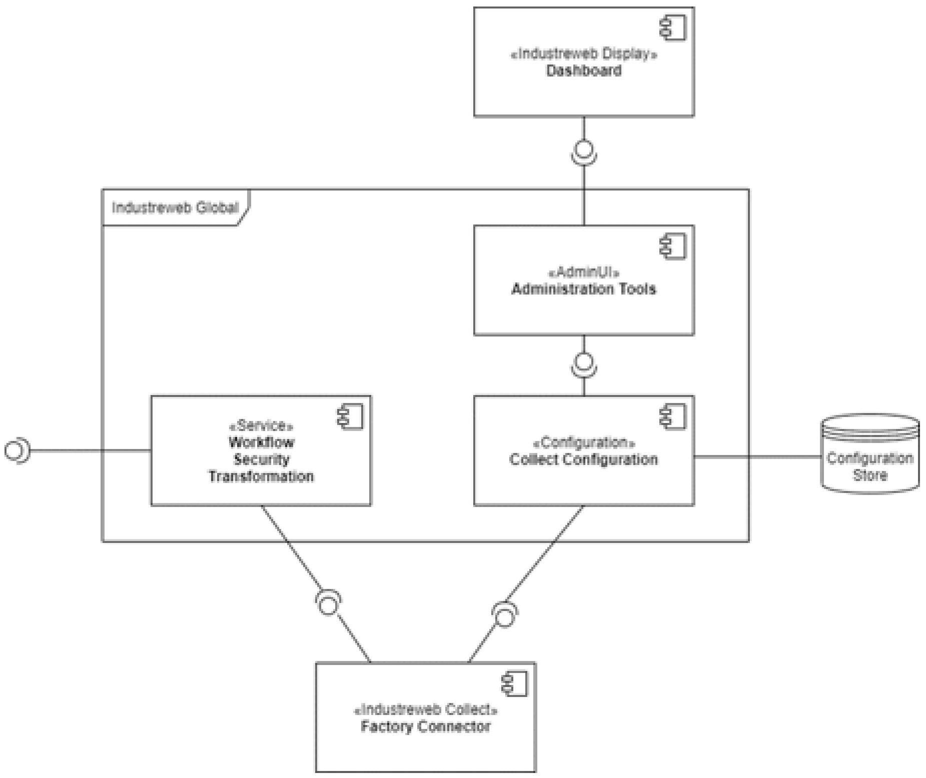

«Industreweb Display» Dashboard
Industreweb Global
«AdminUI» Administration Tools
«Service» Workflow Security Transformation
«Configuration» Collect Configuration
Configuration Store
«Industreweb Collect» Factory Connector

Figure 21: Architecture of Industreweb Global

1.2.2.11.4 Risk Tool

The Risk Tool enables users to create and customize risk recipes, capable of transforming input data into a new shape. These risk recipes can be chained into risk workflows that receive data through MQTT, and at the end produce risk metrics and statistics, which are also published via MQTT. It will also support notification functionality that notifies the user whenever a risk exceeds some threshold. These functionalities are written in a Python backend.

Additionally, the Risk Tool contains a configuration frontend that lets the user create and edit recipes and workflows visually. It also allows the user to test the recipes and workflows and start/stop the MQTT subscriptions of workflows. Lastly, users can see the last output of a running workflow.

There is also going to be a monitoring frontend, more focused on the visualisation of the workflow outputs. It will display figures of the data and risks but will not allow for any configuration (except (un)subscription). The frontends are React webapps, written in JavaScript. The webapps interface with the backend through its API.

The relation between the different components of the tool are displayed in Figure 22.

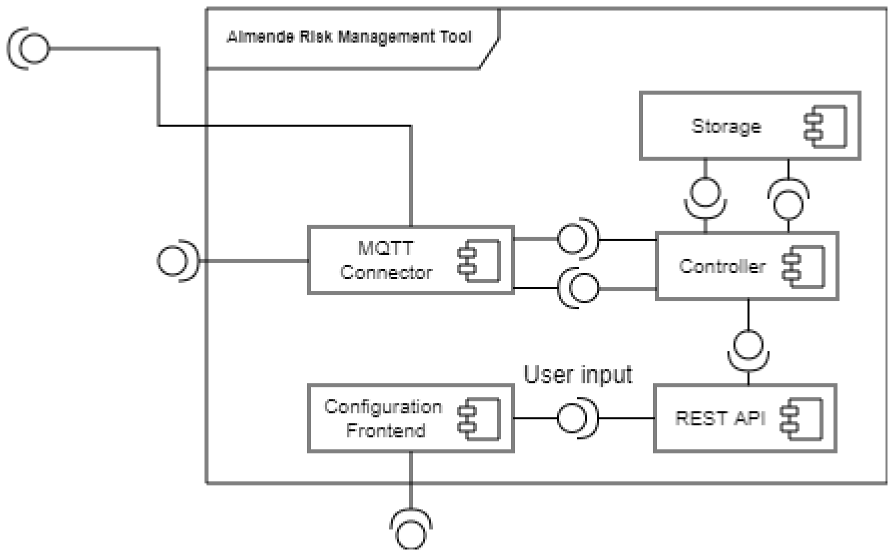


Figure 22: Risk Tool UML Diagram

1.2.2.11.5 Catalogue Service

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 enable users to find what they are looking for quickly, 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 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 and Textile Taxonomy as well.

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.

The relevant UML component diagram of Catalogue Service can be seen in Figure 23.

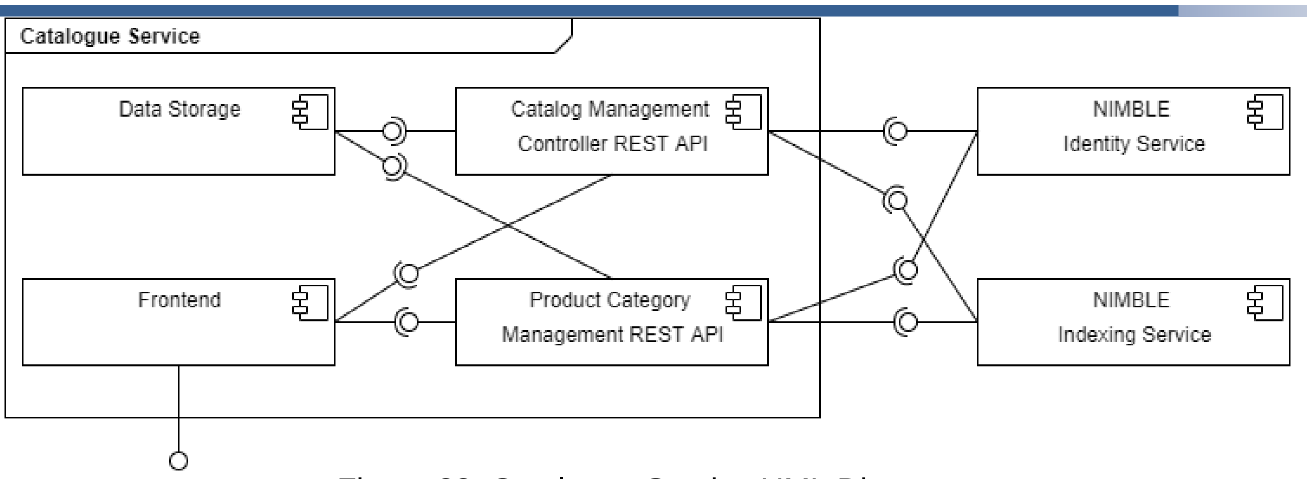


Figure 23. Catalogue Service UML Diagram

1.2.2.11.6 Symphony Event Reactor

The Symphony Event Reactor gives the ability to trigger actions and alarms through its Event Manager/ Alarm Manager in response to different kinds of event types. Moreover, it offers a logging and lifecycle system for alarms. The data model of events is based on a JSON Schema which is human-and-machine readable and dynamically updatable. The event reactor is written in Python and has a GUI written in Blockly (Google), a free and open source client-side JavaScript library for creating block-based visual programming languages (VPLs) and editors.

The Symphony Event Reactor leverages on a highly customisable logging system that allows to handle events locally and synchronise them remotely, together with user activity and alarm history.

The Symphony Event Reactor is composed of two separate software modules:

- **Event Manager (EM)**: The EM executes custom rules that combine information coming from different sources (local sensors and device monitors, user actions, video-cameras, intrusion detection systems, etc.) and data 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)**: Alerts can be raised in order to present the situation to specific users or user-groups. The system provides a configurable priority based alert routing system that allows to target a single, a group or mixed sets of users with SMS, emails, pop-ups, etc. The AM has an internal state machine to track each alarm’s status (Open, Close, Acknowledged, Resolved, Delivered). Also, it logs and keeps alarms history in a log database which is accessible through a REST interface. Different notification channels for the AM are being developed.

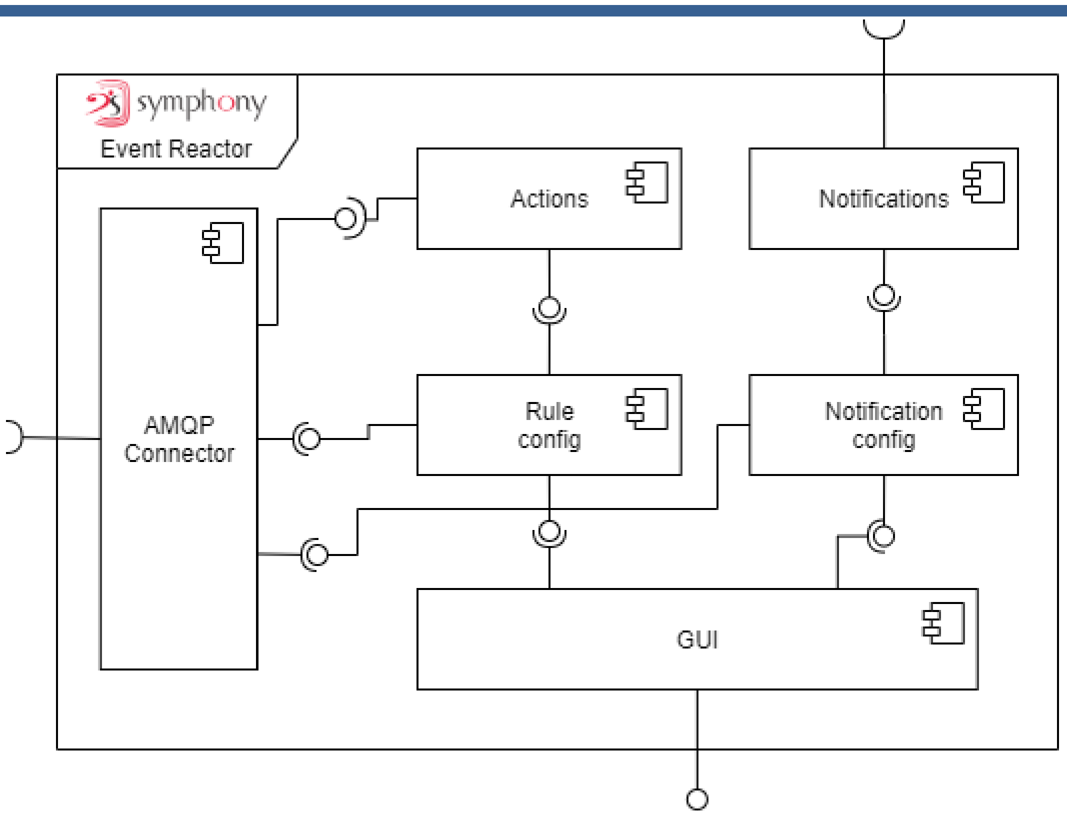


Figure 24 Symphony Event Reactor's UML Diagram

1.2.2.11.7 Symphony Data Storage

Symphony Data Storage is a highly scalable and high-performance data storage which is designed to handle large amount of AMQP/MQTT data. It offers aggregation, rate limiting and sub-sampling, configurable data retention policies and synchronization across multiple instances. It accepts AMQP and MQTT as data source input and provides REST as output. Symphony Data Storage is designed to handle large amounts of data and providing high availability with no single point of failure. Also, it supports PostgreSQL, Cassandra and ElasticSearch as backends.

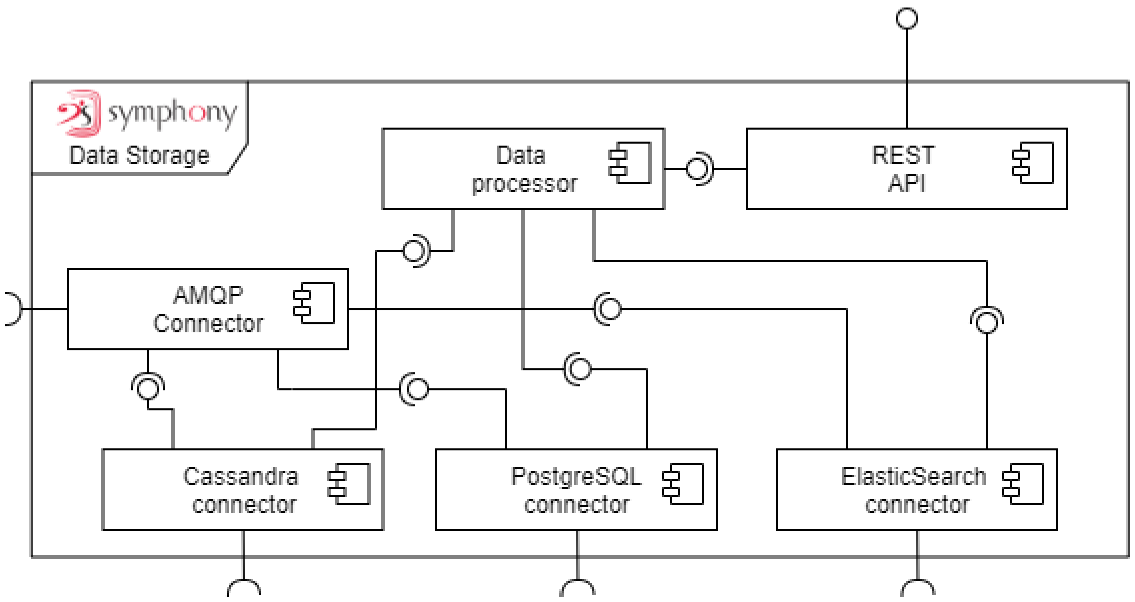


Figure 25 Symphony Data Storage's UML Diagram

1.2.2.11.8 Factory Connectors & Gateways

In order to access data from the numerous data sources available within the manufacturing facilities of members of the four base platforms, it is necessary to utilise a Connector or IoT Gateway that can interface with the devices, sensors and systems. This can be made available through a user defined data model for Smart Factory Tools to be able to operate. The purpose of data provided by these connectors varies from production status, alerting, Kanban / stock level monitoring, to energy consumption, machine/ process efficiency, and more.

In EFPF there are multiple implementations of Factory Connectors & IoT Gateways, supporting between them the most widely used industrial standards and systems (e.g. OPC UA, Siemens, Rockwell, Omron, Schneider etc.).

1.2.2.11.9 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. All data sources are then transformed to a common data model to allow processing and event triggering.

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 UML diagram of IW Collect is shown in Figure 26 which is based around the concept of connectors to enable it to monitor a diverse range of data sources. Data acquired from connectors interfaced with the production systems and sensors. To commission the system, the industrial PC must firstly be interfaced with the production data sources, which typically involves physically connecting the required networks, and the installation of any intermediate 3rd party hardware such as network switches or wireless transmitter/ receivers. Once this has been carried out the interface settings are defined by editing the connector´s configuration file, which defines each data source connector and its properties necessary to function.

Following this stage, the rules to orchestrate the collection and manipulation of data are created which are based on logic events and subsequent actions. The system is then run in the background as a Windows Service, constantly monitoring the manufacturing process.

Actions that IW Collect can trigger may include changing data in a connector, displaying an alert on a screen, sending an SMS or email, or writing a value to a database.

For interfacing with the MQTT broker in EFPF the data is collected from the range of production systems and sensors via the appropriate connector instance and mapped to the MQTT connector instance data model. Upon data changing within the process or at a set time interval the MQTT broker publishes the data. This is then subscribed to by the Smart Factory Tools and Services.

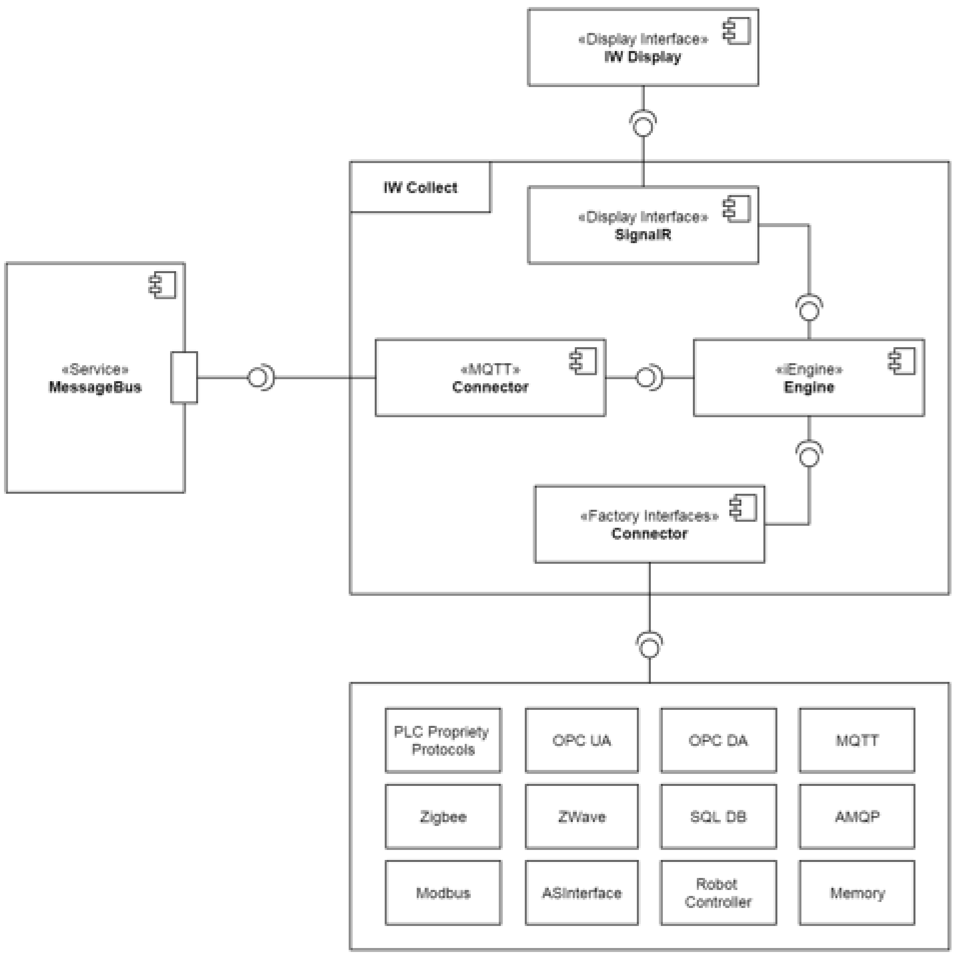


Figure 26: Industreweb Collect UML Diagram

1.2.2.11.10 TSMatch Gateway

TSMatch (Thing to Service Matching) solution provides matching between Things and IoT services via a data matching based on service requirements description. TSMatch solution is in the design phase and is composed of two main components:

- **TSMatch Application**: is an Android application that provides an interface for users (end-users and IoT services) to specify requirements (data inputs, area of interest, observation rate, threshold, etc.) and notify users about updates (which are responses to the user's requests).
- **TSMatch Gateway**: is a software component that receives requests from the TSMatch application and performs data matching to see if the request can be fulfilled. The TSMatch gateway works as follows: first, it selects the optimum set of Things to respond to the request, then decides if additional data processing is required to transform the raw IoT data to match the information requested by the user. This is achieved through 5 main subcomponents:
  - **Thing Discovery**: discovers the IoT devices descriptions and data streams
  - **Context Model**: builds a semantic model of the environment context based on the descriptions of the IoT devices. The goal of the model is to identify if the user request can be answered based on the current context.

- **Negotiator**: interfaces between the TSMatch gateway and App. It receives the user's requirements descriptions through the TSMatch App and informs the user about the outcome of the request (meaning if the request can be fulfilled or not)
- **Service Mapping**: provide a match between the service requirements and the IoT devices by selecting an optimum set of IoT devices capable of answering the request and deciding if additional data processing is required
- **Data Stream Transformation**: based on the selected IoT devices and the service requirements (i.e. threshold, update rate), data stream transformation component subscribes to the raw data and sends updated to the TSMatch App

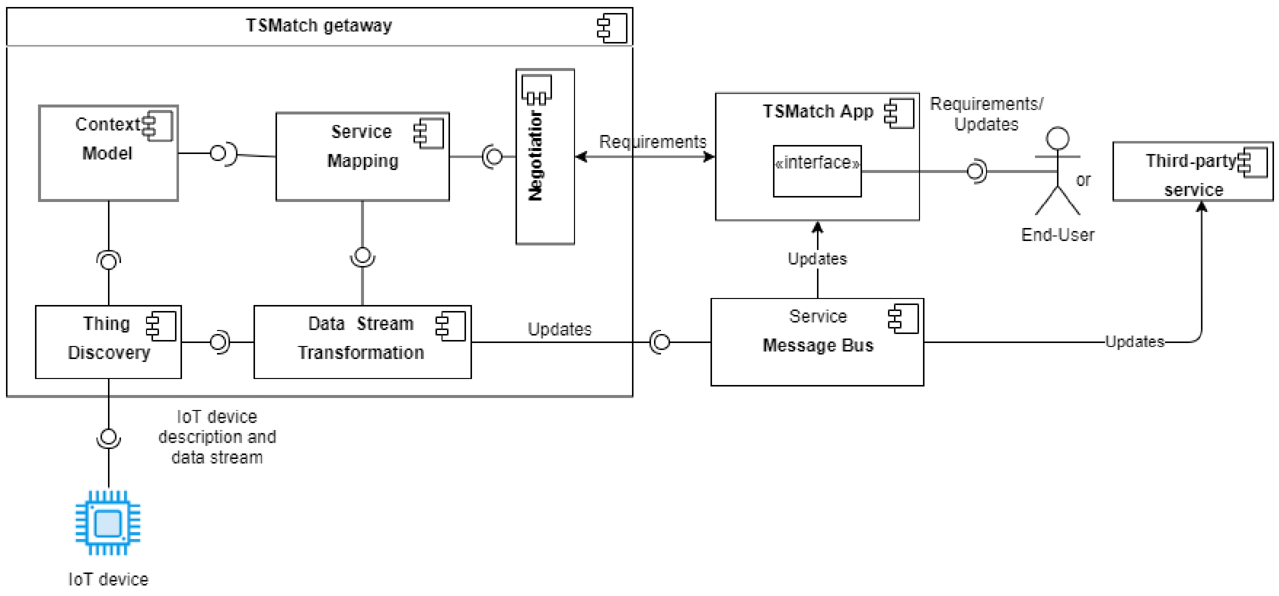


Figure 27: TSMatch Gateway UML Diagram

1.2.2.11.11 Symphony Hardware Abstraction Layer (HAL)

Symphony Hardware Abstraction Layer (HAL) primarily abstracts the low-level details of various heterogeneous fieldbus technologies and provides a common interface to its users. It adapts the device protocols and provides the necessary logic to manage them accordingly to their respective constraints (e.g. timing constraints). It also implements optimizations, e.g. avoid spamming the KNX bus[6] with too many messages, pack contiguous Modbus reads into a single multi-register read.

[6] https://www2.knx.org/no/knx/association/what-is-knx/index.php

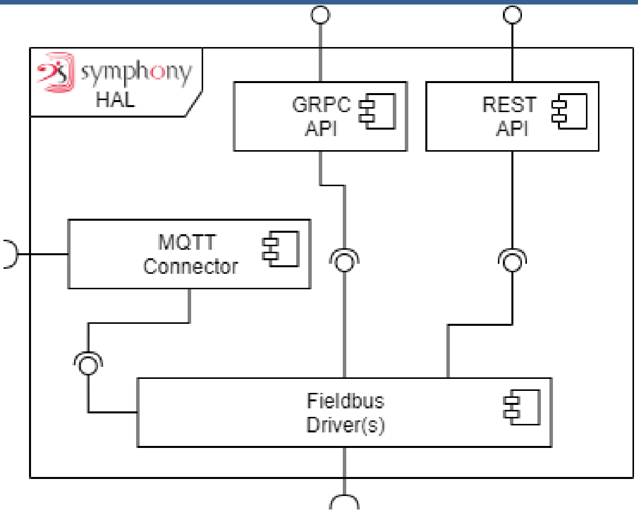


Figure 28: Symphony HAL's UML Diagram

The 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 dynamically 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.

1.2.2.11.12 Factory Connector Gateway Management Tool

The Factory Connector Gateway Management Tool (FCGMT) is an administration tool for Factory Connectors and IoT Gateways in the EFPF platform. This consists of a frontend application (FC frontend) - which is directly accessible from the EFPF portal or as a standalone application – and an API to interact with the device data managed through the frontend (FC API).

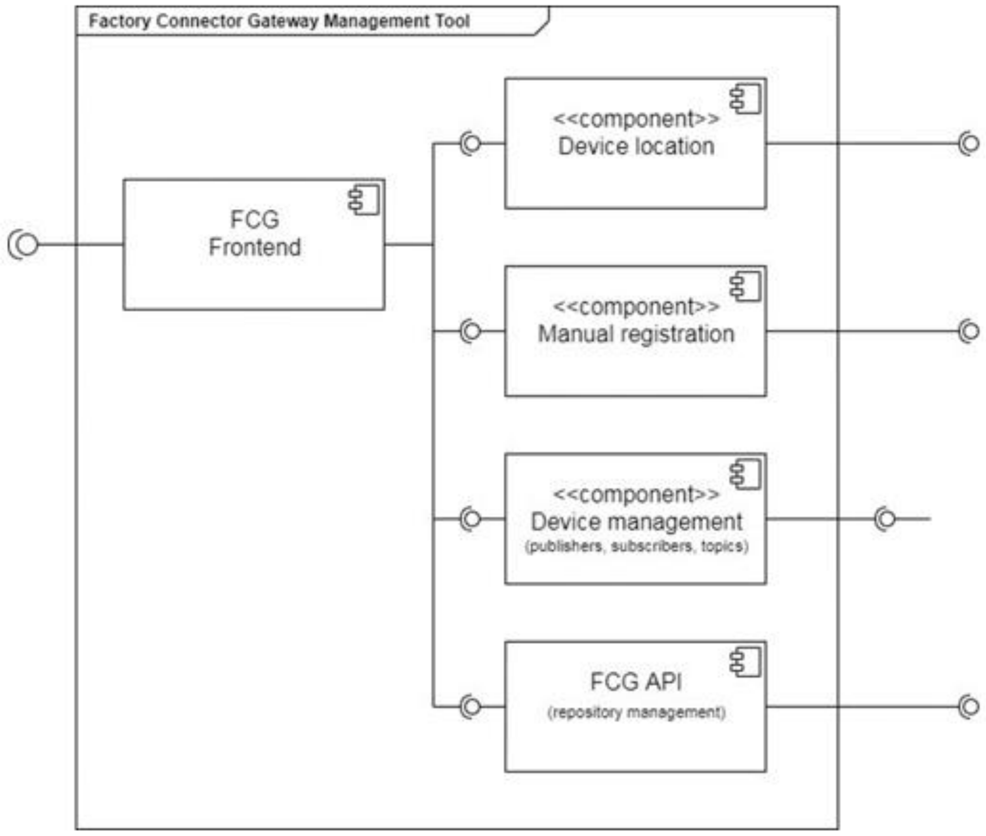

Factory Connector Gateway Management Tool
FCG Frontend
<<component>> Device location
<<component>> Manual registration
<<component>> Device management (publishers, subscribers, topics)
FCG API (repository management)

Figure 29: FCGMT's UML Diagram

The Service Registry API is initially consumed to get the main relevant information about Factory Connectors and IoT Gateways registered in the EFPF platform.

Furthermore, the FCGMT provides an API to register particular IoT devices which may play the role of producers (data publishers) or consumers (data subscribers). The producer devices are able to create and publish data on specific topics (parent or subtopics) depending on the authorization. In the same way, subscriber devices are able to listen to allowed topics. In order to manage the authorization access, the API provided by the API Security Gateway is also consumed by this tool.

In addition, the operations to directly interact with the Message Bus have been considered, and therefore will be integrated in the FCG API using a preliminary development of a RabbitMQ API that manages the creation of resources in the Message Broker, such as exchanges, queues and bindings between queues and routing keys, which are in the end equivalent to the topics concept in MQTT.

1.3 EFPF Architecture Information View

This section describes dataflow in the EFPF ecosystem at different levels of detail. First, a very high-level overview of service-to-service communication in the EFPF ecosystem through the Data Spine, with the Data Spine as a 'black-box' is presented. Second, the dataflow between the services that follow synchronous (request-response) communication pattern at run-time through the Data Spine is presented with the help of an example. Finally, the dataflow between the services that follow asynchronous (publish-subscribe) communication pattern at run-time through the Data Spine is presented with the help of an example.

1.3.1 High-level Dataflow in the EFPF Ecosystem

The EFPF ecosystem is based on a federation model. The services belonging to different platforms are heterogeneous and interoperability gaps exist between them at the levels of protocols, data models, data formats, data semantics, and also authentication providers. Data Spine is the gluing mechanism that is capable of bridging these interoperability gaps and enabling communication between them, thereby enabling communication in the EFPF ecosystem. In order for a pair of heterogeneous services in the EFPF ecosystem to communicate with each other, they are integrated through the Data Spine at first. Once the integration is done, communication can happen. Figure 30 shows a very high-level overview of dataflow between such heterogeneous services in the EFPF ecosystem through the Data Spine, with the Data Spine as a 'black-box'. The subsequent sections illustrate the dataflow in the EFPF ecosystem at greater levels of detail.

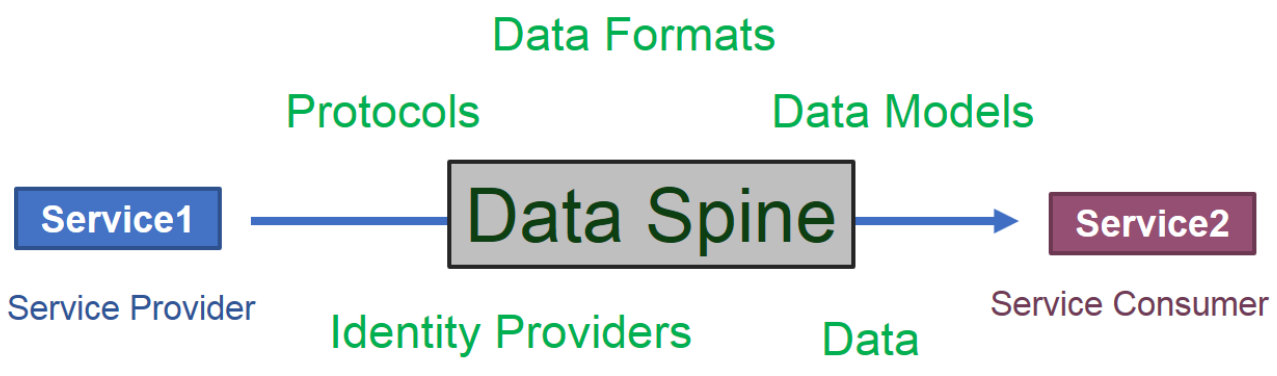


Figure 30: High-level Dataflow through Data Spine

1.3.2 Synchronous Dataflow in the EFPF Ecosystem

Figure 31 illustrates an example of dataflow at run-time between synchronous services in the EFPF Ecosystem. It shows an example of a request-response workflow for EFPF the Marketplace service where the Marketplace services fetches a list of products and services from the marketplaces of the base platforms and displays them onto a GUI. The Marketplace GUI initiates the call to its backend. The Marketplace Backend searches for services of type 'marketplace' in the Service Registry through a proxy endpoint in API Security Gateway and discovers the marketplace services of COMPOSITION and NIMBLE platforms. It then invokes these services through the Integration Flows created in the Integration Flow Engine of the Data Spine and gets a response. The data models of responses from these marketplace services are transformed to the EFPF Marketplace's data model through the Integration Flows. Finally, the Marketplace backend aggregates the responses, hands over the data to the GUI which then displays it onto a Webpage. The exact detailed steps for this kind of synchronous communication workflow are explained in Sections 2.3 and 2.4.

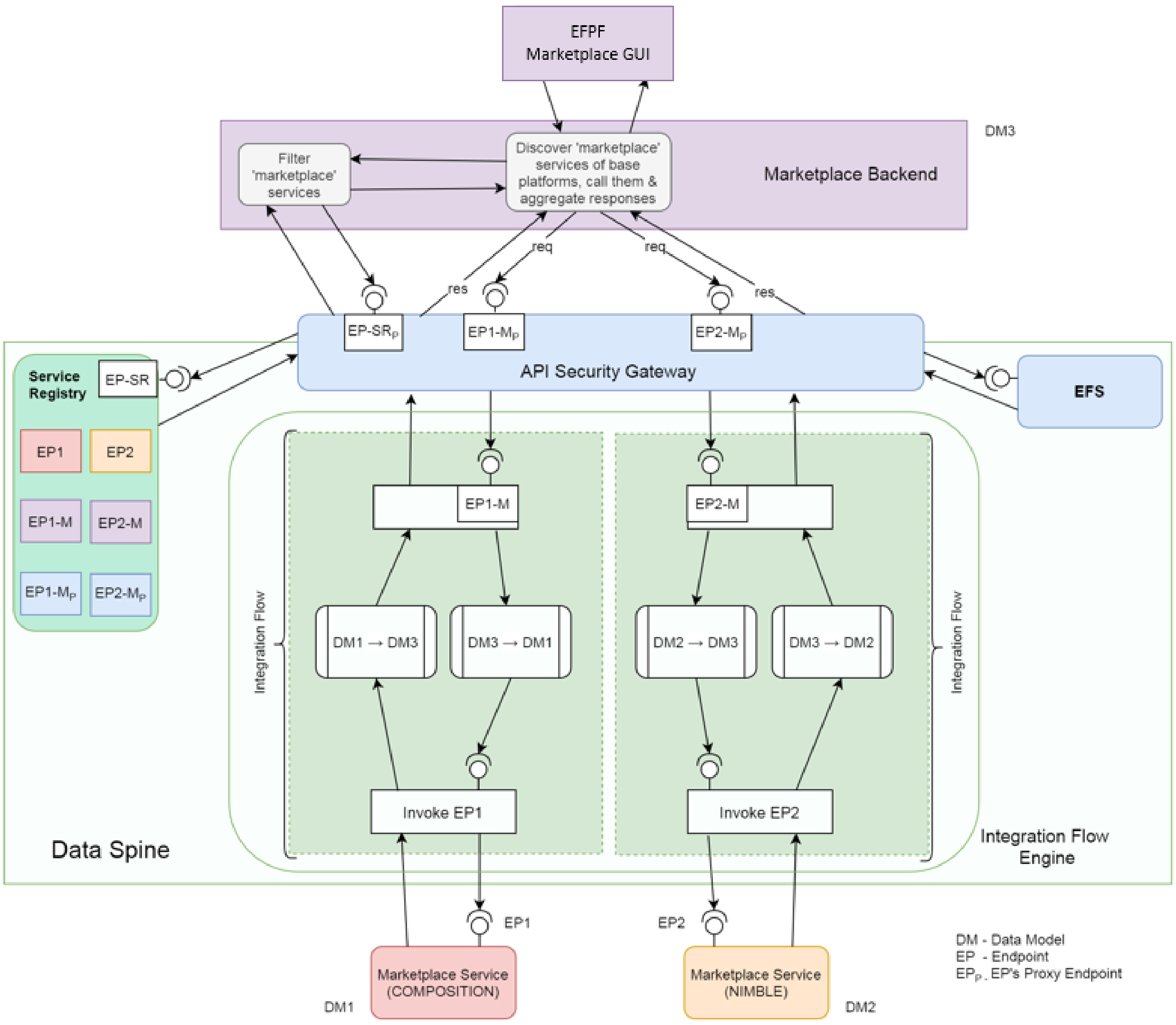


Figure 31: Example of Synchronous Dataflow in the EFPF Ecosystem

1.3.3 Asynchronous Dataflow in the EFPF Ecosystem

Figure 32 illustrates an example of dataflow at run-time between asynchronous services that follow publish-subscribe communication pattern in the EFPF Ecosystem. This example presents a scenario where the Risk Tool receives data from the Factory Connectors through the Data Spine, analyses it, calculates risk and the Event Reactor based on this risk generates an alert. In this example, the Factory Connectors/Gateways ‘IW Collect’ installed at ‘Factory 1’ and ‘HAL’ installed at ‘Factory 2’ publish data to the Data Spine Message Bus. Through the pre-configured Integration Flows, this data is transformed to align with the Risk Tool’s data model and is published again to the Message Bus over new topics. The Risk Tool subscribes to these new topics and gets the data. It then analyses the data and identifies the risk associated, if any. The data could be the status of production with a delivery date for example and delayed delivery date could be a risk that the Risk Tool identifies. Once the Risk Tool identifies a risk, it publishes the risk data to the Message Bus. Again, through another pre-configured Integration Flow, this data is transformed to align with the Event Reactor’s data model and is published again to the Message Bus over a new topic. The Event Reactor subscribes to this new topic, gets the risk data and generates an alert. The exact detailed steps for this kind of asynchronous communication workflow are explained in Sections 2.3 and 2.4.

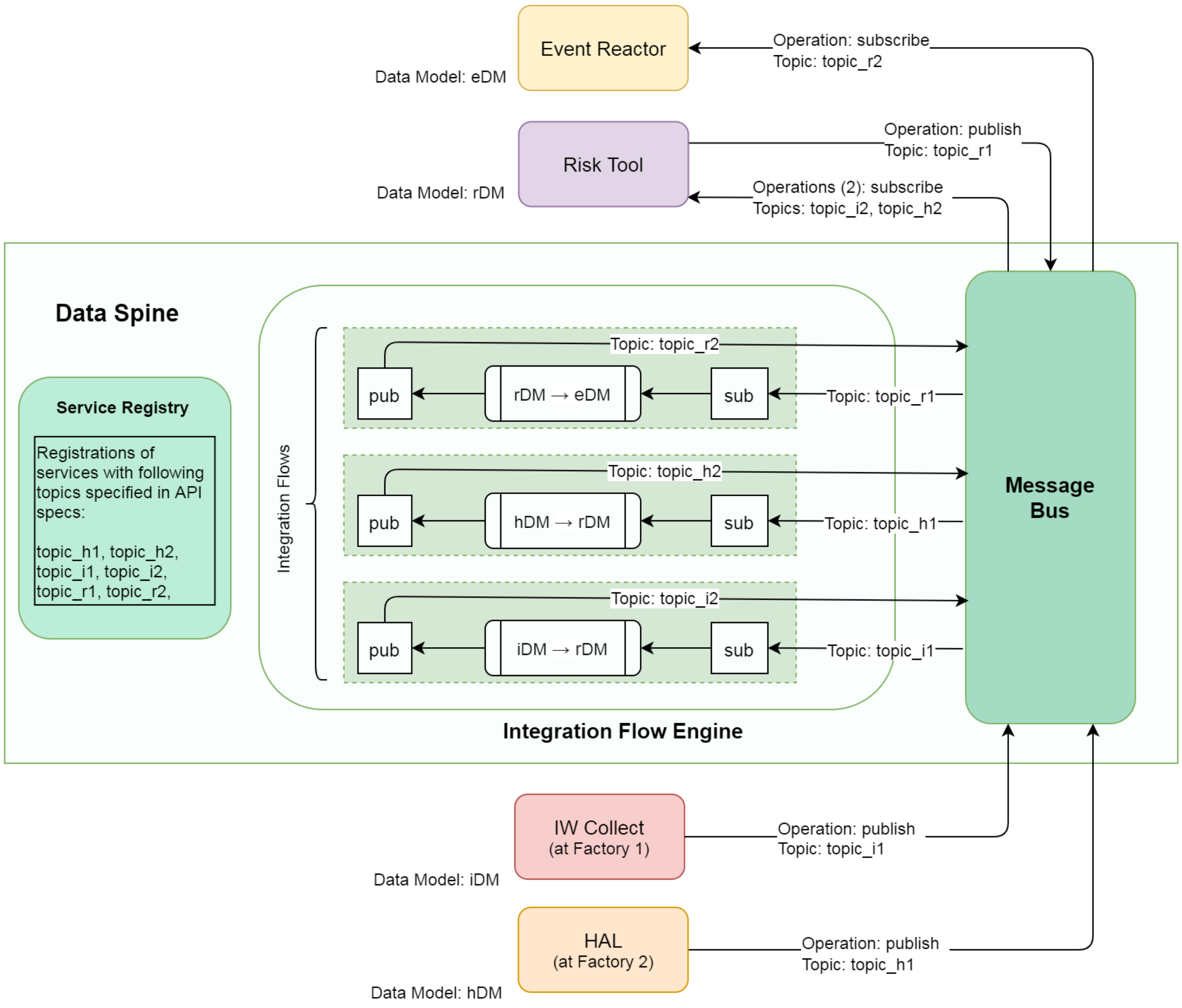


Figure 32: Example of Asynchronous Dataflow in the EFPF Ecosystem

1.4 EFPF Architecture Development and Deployment View

1.4.1 Deployment Process

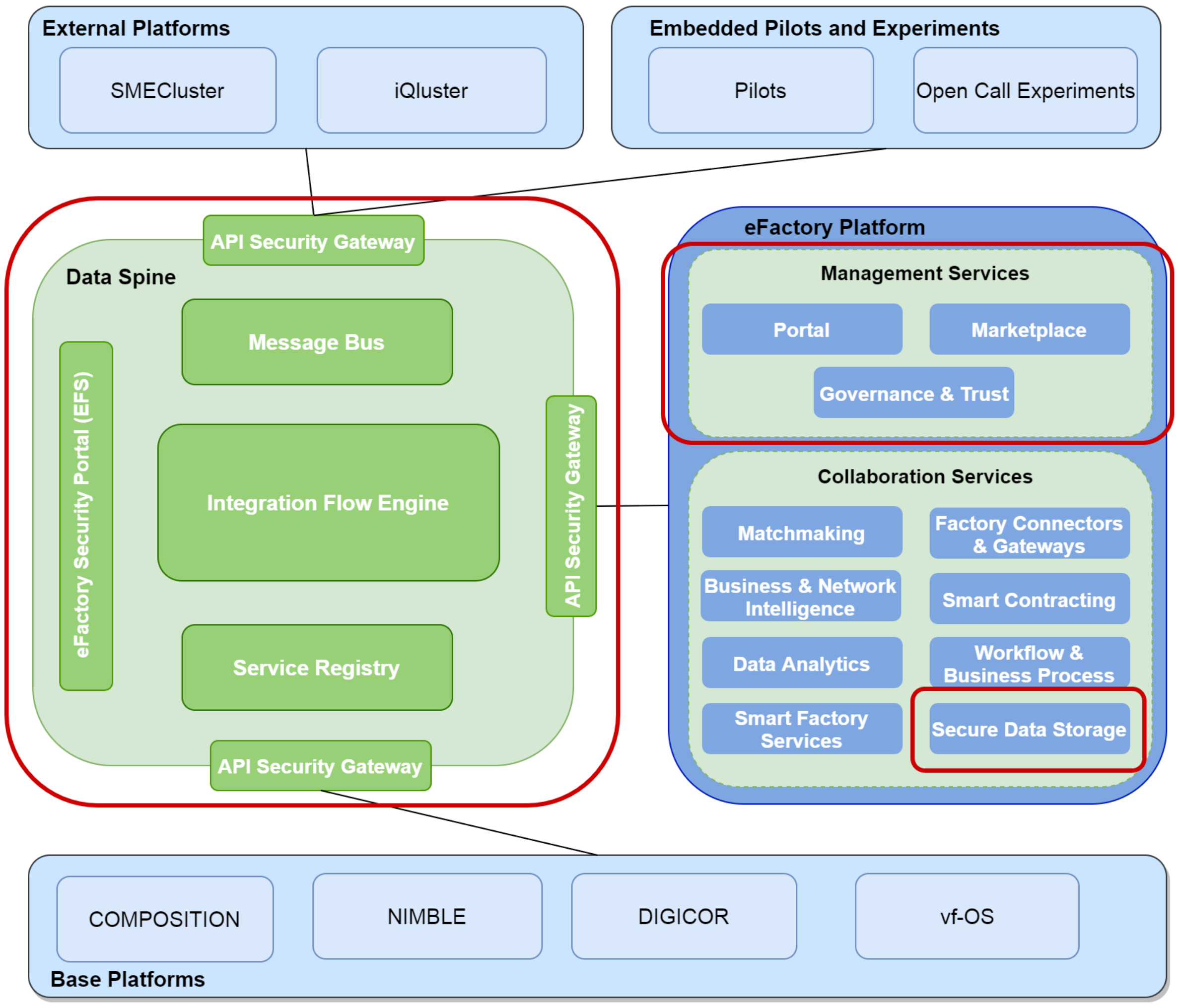


Figure 33: The EFPF Ecosystem and its Components

For the components framed in red colour in Figure 33 - the Management Services, the Portal and Marketplace components, the Secure Data Storage and the Data Spine - the hosting, deployment and operational requirements would be specified. The Collaboration Services and Base platforms are individually installed and configured by the responsible partner (defined in D6.1 and D4.1). A number of components do not allow access to source code or runtime environment (see D10.1) and cannot be deployed and operated by other partners. However, the recommendation for all components in EFPF is to use Docker images as unit of deployment and comply with operational management requirements.

A GitLab repository is currently used for project planning, source code management as well as continuous integration (CI), continuous delivery (CD) and monitoring. The core components are migrating towards this setup. The continuous integration and testing environment are available to all EFPF base platform and service providers but mainly used for the core EFPF infrastructure. The repository will be successively populated and adapted for integration of additional external base platforms and/or tools.

The deployment process in EFPF is based on containerization using Docker. GitLab CI/CD can be used to build new version and push versioned docker images to the EFPF registry (Gitlab Artifact Repository). This allows for an automated deployment and rollback process that is not dependent on component developers.

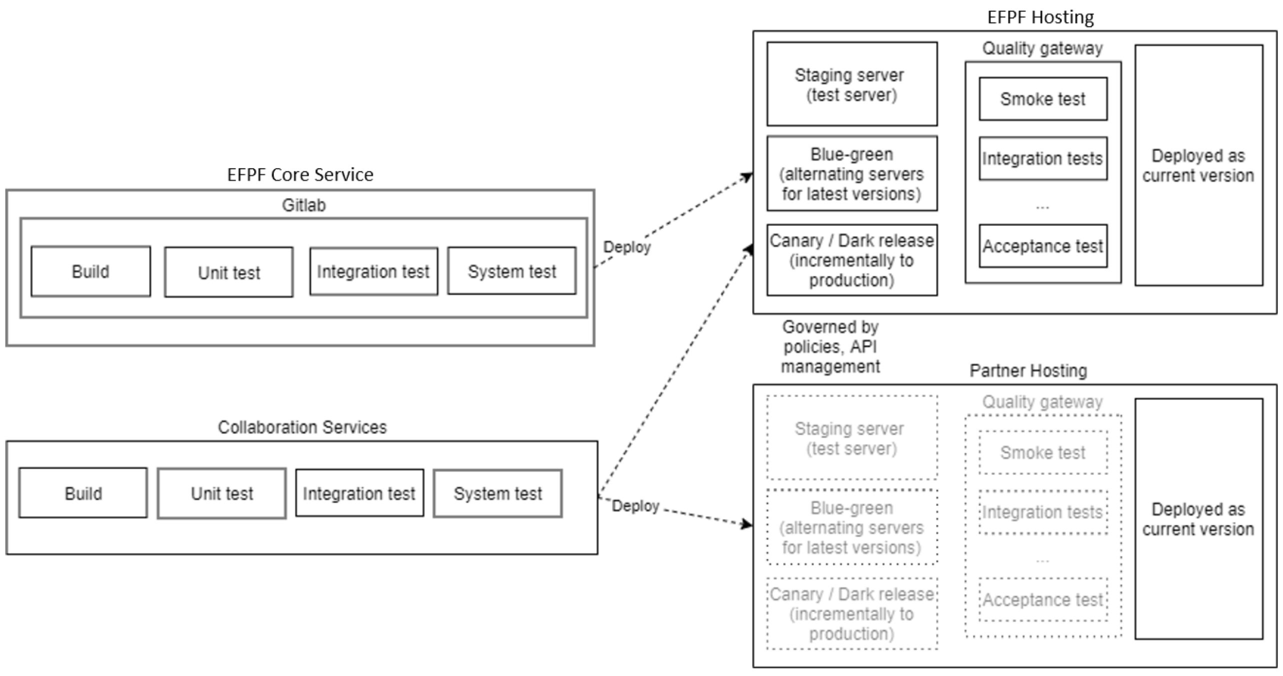

EFPF Hosting
Staging server (test server)
Blue-green (alternating servers for latest versions)
Canary / Dark release (incrementally to production)
Quality gateway
Smoke test
Integration tests
...
Acceptance test
Deployed as current version
EFPF Core Service
Gitlab
Build
Unit test
Integration test
System test
Deploy
Governed by policies, API management
Partner Hosting
Collaboration Services
Build
Unit test
Integration test
System test
Deploy
Staging server (test server)
Blue-green (alternating servers for latest versions)
Canary / Dark release (incrementally to production)
Quality gateway
Smoke test
Integration tests
...
Acceptance test
Deployed as current version

Figure 34: EFPF Deployment Process

Component deployment

The core components that will have a defined deployment pipeline. Although Docker images in the recommended unit of deployment, other components may be deployed in various ways and on different platforms.

1.4.2 Environments

Starting with the Data Spine, a 3-tier deployment model is being used, composed of a development, testing and production environment.

Environment	Content	Provider
Development	Development version where new updates will be made.	FIT / SRFG
Testing	Test version to verify the integrated Data Spine. Elements passing quality gateway may be brought into Production.	C2K
Production	Stable live version used by all, including the open calls	C2K

Figure 35: Data Spine Deployment Environments

The environments will have different rate and type of change.

Development will be a less stable environment where e.g. component connectors, security mechanisms and network topology may change. There may be in-place changes by developers in the development environment and no quality gateways – apart from unit tests – are needed to deploy anything there. Malfunctions and conflicting versions are solved by developer-to-developer communication on instant messaging channels.

The testing environment serves as a more stable environment for integration testing. (Although this is not the intended use, some performance testing can also be done there.)

Components are deployed to the test environment by script, using docker images as unit of deployment for EFPF hosted components. Configuration should be complete and no in-place changes needed. If a new version does not pass the quality gateway, it is rolled back, corrected, and re-deployed, not edited in place.

Once the component passes the quality gateway, this version may be deployed to the production server.

The policies for API management apply and a quality gateway is applied.

The environments should be isolated. No connections are set up between development, test and production. An exception could be factory connectors feeding data to the test environment.

An open issue is how to manage data in the test environment. Both streaming and static data is often needed for integration tests. Streaming data could be mirrored from the production environment (not recommended for security reasons), provided in a “playback loop” that provides a fixed sequence of observations and events that each component can use for test purposes, or each component can inject its own set of test data into relevant streams. Static data, e.g. in the Secure Data Store, should not be considered stable. Integration tests that change static data should set up and restore the data sets needed. Some global data may have to be versioned in sync with component versions, or a policy of restoring all data on a set interval may be used. This will be selected depending on emerging needs and practices.

2 Design and Realisation of Interoperable Data Spine

This section presents in detail the interoperability backbone of the EFPF ecosystem – the Interoperable Data Spine. First, it describes in brief the need for having such a component in the EFPF ecosystem. Second, it elaborates the design of the Data Spine in the form of identification of the conceptual components that constitute the Data Spine. It also presents the reference architectures of these conceptual components which are independent of the implementation aspects. Finally, it presents the open source platforms, services and tools chosen to realise these conceptual components of the Data Spine.

2.1 Design of Interoperable Data Spine

The EFPF ecosystem is based on a federated model which consists of distributed, heterogeneous digital platforms, tools and other components developed, provided and, in some cases, hosted by independent entities. Therefore, their technical aspects such as interfaces, protocols, data formats, data models, identity and access management mechanisms, etc., differ significantly from each other and direct communication between their services isn't possible. 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 Service Providers and Service Consumers to implement connectors/plugins so that the former can align their proprietary APIs to these standard APIs and the latter can consume these standard APIs to 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. A solution that provides a communications layer that acts as a translator/adapter between these heterogeneous tools, services, systems and platforms providing data handling, routing capabilities and API adaptation functionalities and enabling communication without needing modifications to the existing services is needed. The EFPF Data Spine is designed to provide such a solution that address these problems.

Data Spine is a collection of components that work together to form an integration, interoperability and communications layer for the EFPF ecosystem. Sections 1.1 and 1.2.1 provide an overview of the Data Spine and its conceptual components. Figure 2 depicts these conceptual core components that provide the expected functionality of the Data Spine:

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

The process followed for the identification and design of these conceptual components included gathering of interoperability requirements from the base platforms and some of the external platforms to be integrated into the EFPF ecosystem. The technical profiles of these platforms were documented, which included the specific components from these platforms, 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 and Figure 36 presents a summary of the platform profiles. Based on these technical profiles of the base and external platforms, their interoperability requirements and the other factors mentioned before, the conceptual components of the Data Spine were defined. The subsequent sections describe the design and architectural aspects of these individual conceptual components and Section 2.1.6 illustrates the reference architecture of the Data Spine highlighting the relationships between these conceptual components that constitute the Data Spine.

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 36: Summary of Platform Profiles

2.1.1 Integration Flow Engine

Integration Flow Engine (IFE) is the component of the Data Spine that provides service integration and interoperability capabilities such as connectivity, data routing, data transformation and system mediation functionalities. These capabilities could be used to bridge the interoperability gaps at protocol level and data model level between the heterogeneous services communicating through the Data Spine.

The Integration Flow Engine, in order to provide these capabilities, borrows concepts and functionalities from Enterprise Integration Patterns, Enterprise Service Buses, Big Data Processing Frameworks and, Message-oriented Middlewares. It can be said that the Integration Flow Engine of the Data Spine is analogous to a dataflow management system based on the concepts from flow-based programming [Mor10] that makes use of workflows/dataflows to interlink and interoperate between a particular pair of services. In the context of the Integration Flow Engine, such workflows/dataflows are termed as ‘Integration Flows’.

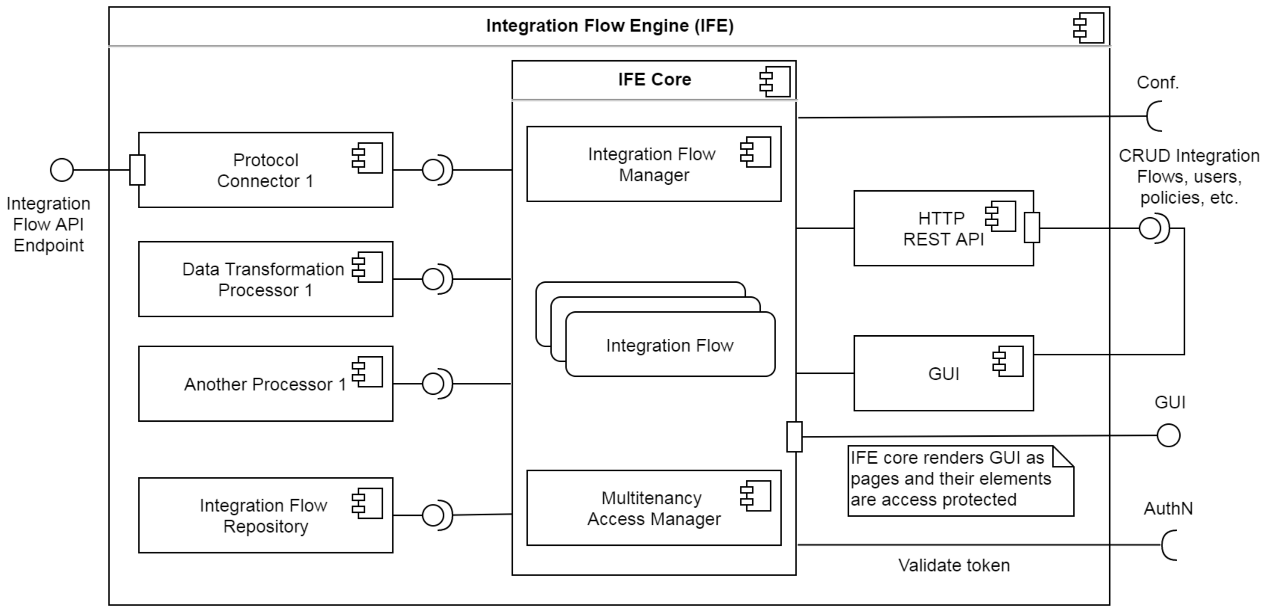

Integration Flow Engine (IFE)
IFE Core
Integration Flow Manager
Integration Flow
Multitenancy Access Manager
Protocol Connector 1
Data Transformation Processor 1
Another Processor 1
Integration Flow Repository
Integration Flow API Endpoint
HTTP REST API
GUI
Conf.
CRUD Integration Flows, users, policies, etc.
GUI
AuthN
IFE core renders GUI as pages and their elements are access protected
Validate token

Figure 37: Architecture of the Integration Flow Engine

Figure 37 illustrates the conceptual components to the Integration Flow Engine. The 'IFE Core' component consists of an 'Integration Flow Manager' component that manages the lifecycle (Create, Read, Update, and Delete operations) of Integration Flows and the Integration Flows are persisted into an inbuilt 'Integration Flow Repository'. 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 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. Figure 31 and Figure 32 show the examples of such Integration Flows that involve synchronous as well as asynchronous communication.

The Processors are the extension points of the Integration Flow Engine. An instance of the Integration Flow Engine should have in-built Protocol Connectors for standard communication protocols that are widely used in the industry. Support for a new protocol could be added by writing a new Protocol Connector and adding it to the Integration Flow Engine.

Figure 37 also shows the interfaces of the Integration Flow Engine. The HTTP REST API offers endpoints for lifecycle management of Integration Flows, processors, process groups, users, user groups, access policies, etc. The Integration Flow Engine offers an intuitive, drag-and-drop style Web-based Graphical User Interface (GUI) to the system integrators to create the integration flows which is based on the concepts from visual programming [Shu86] paradigm. The Integration Flow Engine and its GUI have support for multitenancy. The GUI can be configured based on the defined access policies to allow or restrict visibility of and/or access to certain GUI elements. In addition, access policies such as 'a user or a user group should only be able to view and manipulate only the Integration Flows created by him/her/them' can also be defined and enforced. Thus, the GUI of the Integration Flow Engine is truly multitenant and enables collaboration among system integrator users who create the Integration Flows. Moreover, the Integration Flow Engine supports standard authentication protocols such as OpenID Connect (OIDC) to secure access to its GUI using a pluggable authentication provider such as Keycloak. Therefore, the same user-base from the EFS, the identity provider for EFPF, can be used for authentication.

Finally, to ensure high availability, throughput and low latency, the Integration Flow Engine is scalable as it is capable of operating in a clustered fashion.

2.1.2 API Security Gateway

The API service in EFPF is handled by the API Security Gateway (ASG). The ASG acts as a border gateway for all API calls targeting the Data Spine. In addition to proxying service calls in the Data Spine, the ASG should also be able to enforce granular policies for each API call. The ASG exposes the services available in the EFPF ecosystem by consuming the service registry. The following image shows the UML diagram for the API Gateway.

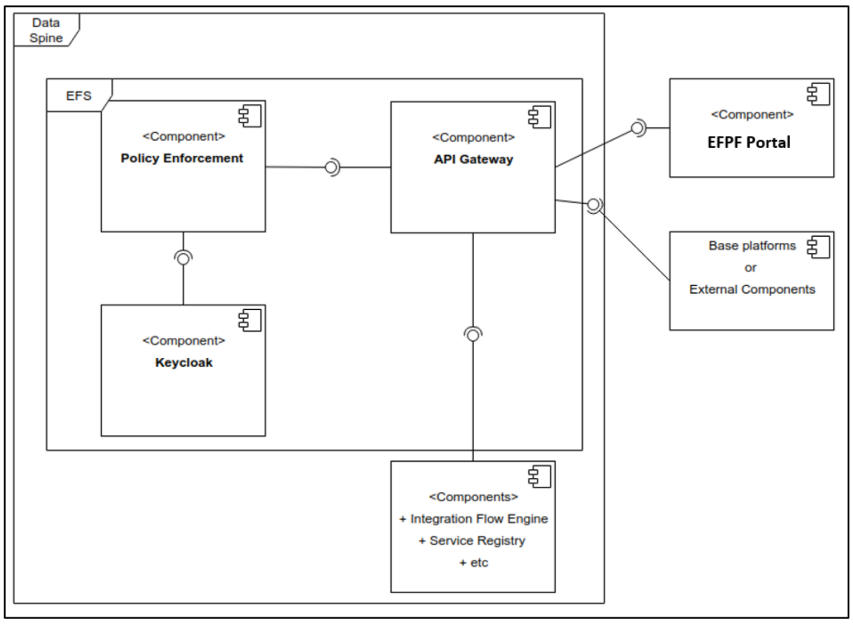

Data Spine
EFS
<Component> Policy Enforcement
<Component> API Gateway
<Component> EFPF Portal
Base platforms or External Components
<Component> Keycloak
<Components> + Integration Flow Engine + Service Registry + etc

Figure 38: API Security Gateway

2.1.3 Service Registry

In an interconnected EFPF ecosystem, services of different platforms need to be first integrated with each other through the Data Spine to enable communication and then they can be orchestrated together to achieve common objectives. In order for the services of one platform to discover the services of other platforms, the service providers should be able to advertise their services along with the associated metadata and make those discoverable for the potential service consumers and for the system integrators. As explained in Section 2.1.1, a pair of services can be integrated with each other through the Data Spine with the help of Integration Flows.

The rationale for having a Service Registry component comes from these requirements that the service consumers need a way to search for services based on their functional metadata such as ‘service type’ whereas in order to write the integration flows, the system integrators need a means to retrieve the technical metadata of the services, e.g. protocols, endpoints, data formats, data models, etc. The Service Registry component of the Data Spine fulfils these purposes. The Data Spine Service Registry is capable of managing such heterogeneous metadata.

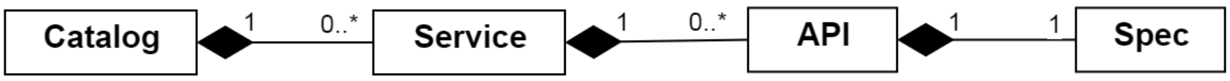


Figure 39: Abstract Class Diagram for the Service Registry

```
{
        "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 40: Abstract Service Description Schema of the Service Registry

Figure 39 shows the abstract class diagram for the Service Registry that shows composition relationship between its classes. 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 Spec. Figure 40 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[7]
- For asynchronous (publish-subscribe) services: AsyncAPI Spec[8]

Thus, this design makes the schema capable of managing metadata for synchronous (request-response) as well as asynchronous (publish-subscribe) type of services. All the technical metadata for the APIs of services that is needed for creating the Integration Flows 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.

[7] OpenAPI/Swagger Spec: https://swagger.io/docs/specification/about/
[8] AsyncAPI Spec: https://www.asyncapi.com/

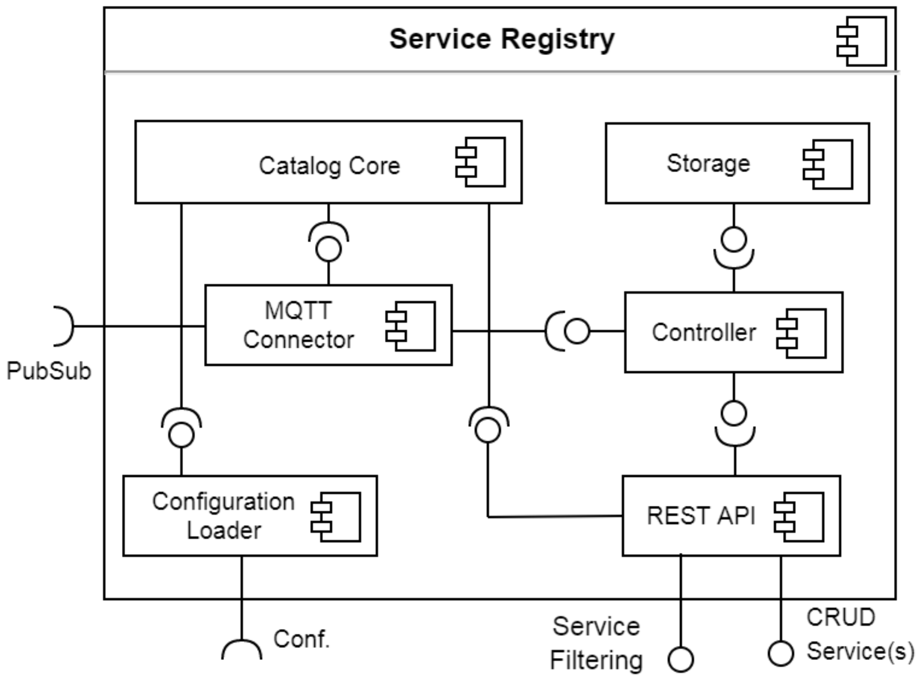


Figure 41: Architecture of the Service Registry

Figure 41 shows the conceptual components of the Service Registry and a REST API. The RESTful API provides an endpoint for the lifecycle management of services and another 'Service Filtering' endpoint for supporting service discovery. The Service Registry also has a Pub Sub API for publishing service status announcements over predefined topics to the Data Spine's Message Bus. The topic names and URL of the Message Bus can be configured using the Configuration Loader component.

In this way, the design of the Service Registry has been kept simple yet effective, so that it could either be realised with any suitable open source service registries or developed entirely from scratch easily while also providing the intended core functionality.

2.1.4 Message Bus

Some of the platforms in the EFPF ecosystem, interconnected through the Data Spine, offer their shop floor data as data streams through their factory connectors/gateways and some tools in the EFPF platform also make use of the publish-subscribe communication pattern. The Integration Flow Engine of the Data Spine supports such asynchronous communication as well. In addition, the Data Spine offers the Message Bus component to the publishers and/or the subscribers in these platforms. The Message Bus supports standard publish-subscribe based messaging protocols such as MQTT, AMQP, etc. that are widely used in the industry. The Message Bus in EFPF could be extended to add support for new protocols via plugin mechanism.

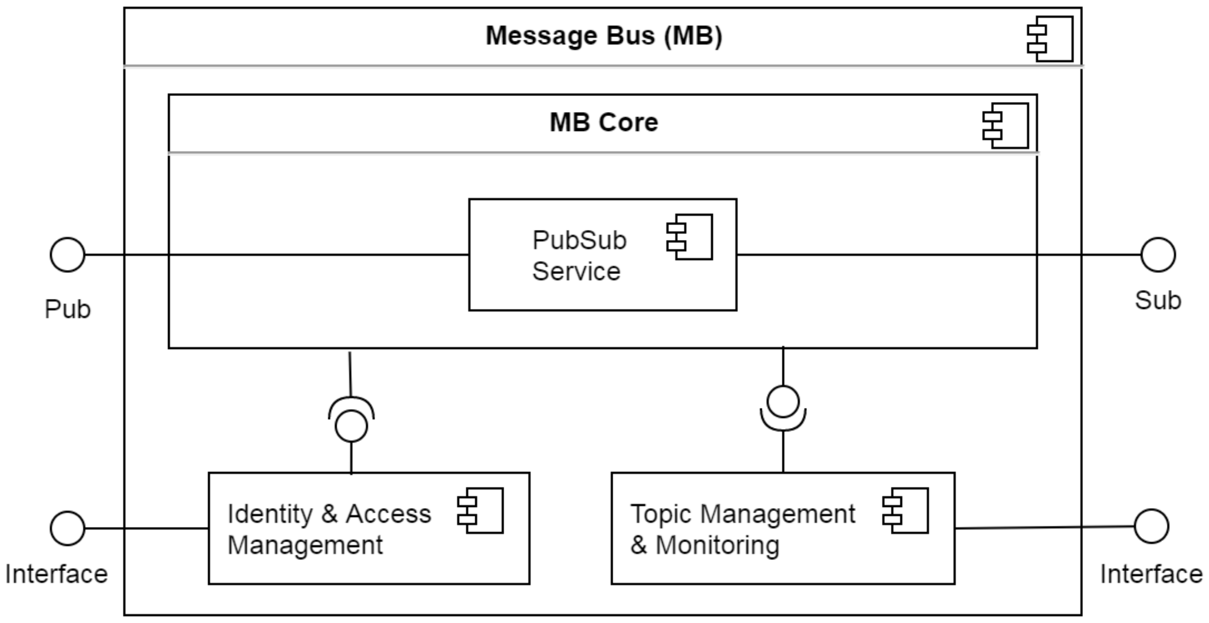


Figure 42: Architecture of the Message Bus

Figure 42 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 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 use of username-password based as well as key 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.

2.1.5 EFPF Security Portal

One of the core requirements of the EFPF platform is to establish a federation of digital manufacturing platforms and enable interoperation between them, using a federated identity management model. To implement federated identity mechanisms, the EFPF platform requires the EFPF Security (EFS) portal in order to govern the security management controls of different platforms in the platform ecosystem. The EFS should be designed and implemented as a distributed single point of trust that enables a class of Super Administrator whose role is to provide secure authentication of any tenant platform in the ecosystem (e.g. multi identities to be managed across company's accounts).

With respect to access control mechanisms for the IoT and the cloud, traditional Role Based Access Control (RBAC) has shown serious weaknesses, e.g. confused deputy attacks through inherited permissions (the user with higher permissions grants access to a specific resource to a user with lower permissions). As an alternative to RBAC, the Attribute Based Access Control (ABAC) model provides fine-grained access mechanisms, in which the authorization decisions are based on attributes that need to be proven by the user (location, roles, etc.) and on other properties (e.g. resource properties).

Figure 43 illustrates a cross-platform user identity and access control management through the EFS [SJDB19].

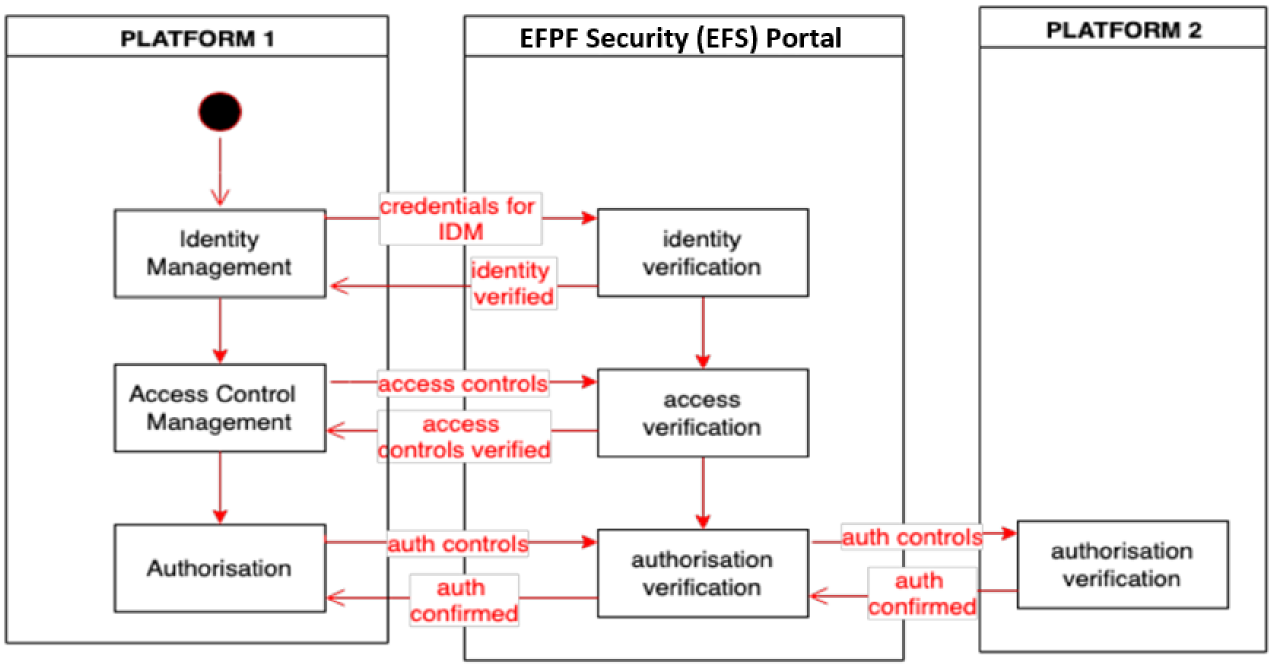


Figure 43: Cross-platform user identity and access management via EFS

The EFPF platform and its Data Spine integration middleware, should act as the federation provider that governs identity federation and user provisioning workflows across the EFPF ecosystem. The Data Spine integration middleware includes the EFS as a component (see Figure 1). Apart from the federated IDM, EFS implements other security controls.

Figure 44 illustrates a high level EFS perspective of the integration of four base platforms within EFPF. Each base platform contains its own IDP that maintains the platform's users, their roles and access policies. Practically the core requirement is that the Data Spine, through EFS, enables federation of the users of base platforms across the platform ecosystem in EFPF.

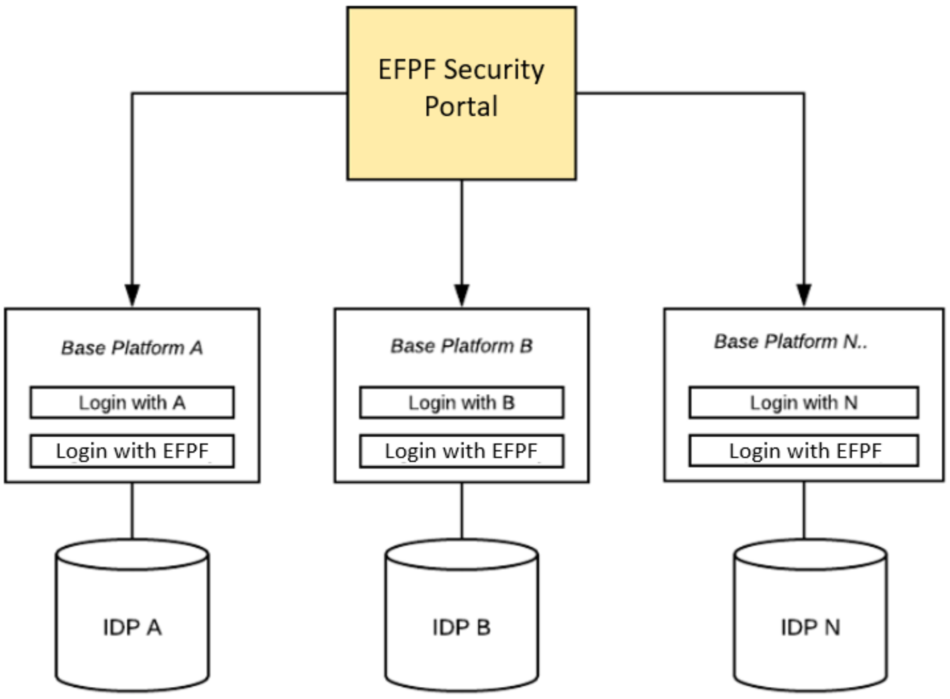


Figure 44: High-level Design of the EFPF Federation

Another design goal in EFPF is to accommodate not only four base platforms, but external platforms that will provide further collaborative manufacturing services. With such a design requirement, one-to-one user federation mappings between platforms will result in a high number of login options for individual platforms (e.g. login to platform A, B, C, ..., N), which will require continuous updates of the authentication and authorization workflows for each platform in the ecosystem that is rather, not feasible.

Therefore, the Data Spine and EFS take on the role of the central federation provider to manage all federation workflows across platforms participating in the ecosystem. The EFS also acts as the gateway enabling the access to the collaborative manufacturing resources provided by the platforms in the ecosystem.

2.1.6 The Data Spine

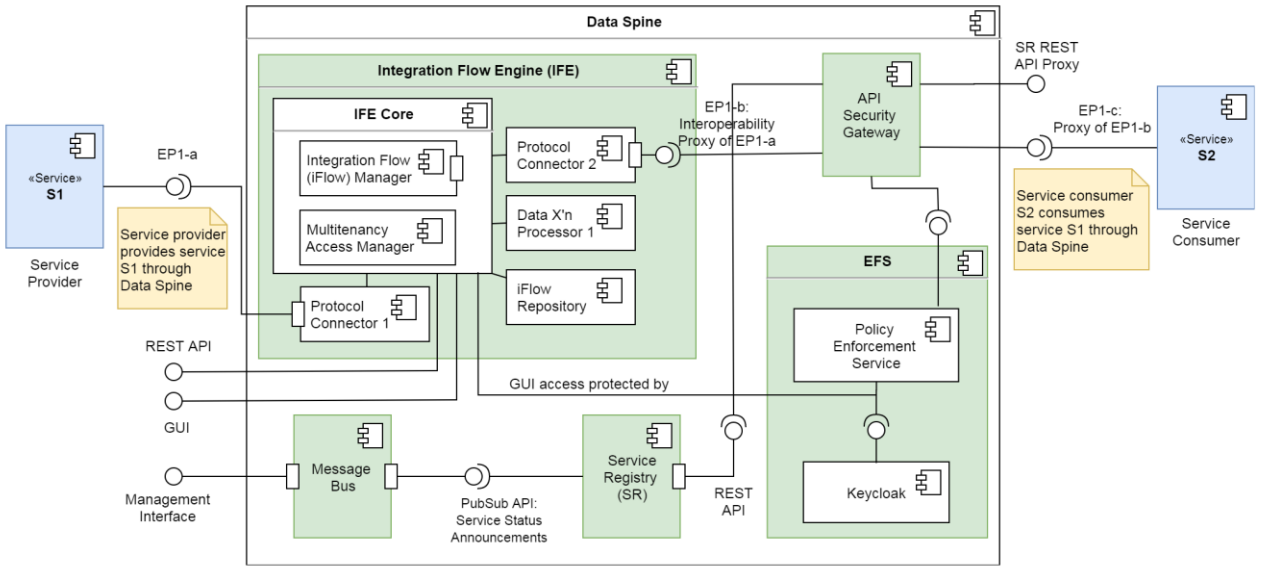


Figure 45: Architecture of the Data Spine

Figure 45 shows the conceptual components of the Data Spine discussed in the sections above as the core components of the Data Spine. The relationships and interactions between these core components are also illustrated. The access to the GUI of the Integration Flow Engine and its elements is protected by the 'Keycloak' component of the EFS. The API Security Gateway acts as the Policy Enforcement Point for the Data Spine. API Security Gateway relies on the Policy Enforcement Service of the EFS to make the access control related decisions. The API Security Gateway is configured to check the Service Registry for new service registrations and service updates periodically to automatically create proxy endpoints or routes for protecting access to them in the API Security Gateway. The access to the REST API of the Service Registry is secured through the proxy endpoints in API Security Gateway. The Service Registry publishes service status announcement related messages to the Message Bus. The design-time administration and management related endpoints of the Integration Flow Engine and the Message Bus are secured through the Identity and Access Management services internal to these respective components. The run-time access to the endpoints exposed by the Integration Flows in the Integration Flow Engine is protected through the corresponding proxy endpoints in the API Security Gateway, once they are registered to the Service Registry.

Moreover, Figure 45 also shows the run-time view of the communication between two services S1 and S2 happening through the Data Spine. As design-time prerequisites, service S1 is registered in the Service Registry, the Integration Flow to consume S1 and perform data transformation has already been created and activated and, finally, service S2 has acquired access rights for invoking S1 through the Data Spine. The operation at run-time:

1. S2 makes a call to the proxy endpoint EP1-c in the API Security Gateway with an EFPF token.
2. The API Security Gateway delegates the authentication and authorization responsibility to the EFS.
3. The EFS verifies whether the token is valid and has necessary authorization to invoke the EP1-c endpoint and perform the specified operation and finally, replies to the API Security Gateway.
4. Upon receiving a positive reply from the EFS, the API Security Gateway invokes the corresponding endpoint EP1-b exposed by the respective Integration Flow in the Integration Flow Engine.
5. The Integration Flow transforms the request, if specified and invokes the original EP1-a endpoint of S1.
6. Upon receiving response from S1, the Integration Flow transforms the response payload, if specified and returns the response to the API Security Gateway.
7. The API Security Gateway returns the response to S2.

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

2.2 Realisation of Interoperable Data Spine

This section details the process of implementation of the Data Spine based on the design aspects and requirements identified in Section 2.1 and, presents and elaborates the open source platforms, services and tools chosen to realise these conceptual components of the Data Spine.

In order to realise the conceptual components of the Data Spine, available permissive open source technological platforms and frameworks were surveyed. The surveyed platforms included Apache NiFi, WSO2 Integration Agile Platform, FIWARE, Symphony BMS, Apache Camel along with the platforms based on it such as Talend ESB and Red Hat Fuse, LinkSmart Platform, Apache Flink, Confluent Platform, etc. The factors considered for the survey were Platform, License, Language, Plugin/Extension Mechanism, Supported Languages for Plugins, Hot Plugin Deployment, REST/API Management, Reverse Proxy Support, Identity and Access Management, Type, Message Bus and Relation to Data Spine Conceptual Component. The results of the survey are included in Annex D of D3.1.

The surveyed platforms could be broadly classified into three different categories:

- Lean Frameworks or Code Libraries such as Apache Camel, Apache Synapse, etc. that offer no GUI to create Integration Flows graphically.
- Lightweight Dataflow Centric Platforms such as Apache NiFi that offer service integration capabilities such as connectivity, routing, data transformation, mediation, etc. and a Web-based intuitive GUI to collaboratively create Integration Flows.
- Heavy-weight Integration Suites or Enterprise Service Buses such as WSO2, Talend ESB, etc. that, along with the integration capabilities mentioned above, also provide other functionalities such as Business Process Management Rule Engine, Activity Monitoring, Security Compliance and Repositories, etc. built it into a single solution.

Based on the design requirements identified in Section 2.1, the platforms in categories 2 and 3 above were found to be the most suitable for realising the components of the Data Spine. From these 2 categories, as stated in D3.1, the survey resulted into the identification of two possible solutions:

- WSO2 Carbon Infrastructure Stack to realise all the components of the Data Spine and,
- Apache NiFi to realise the Integration Flow Engine of Data Spine with other components such as LinkSmart Service Catalog or Consul, Keycloak, RabbitMQ, etc. to realise the other components of Data Spine.

Finally, based on the experimental evaluation for quality assessment of Apache NiFi and WSO2 Platform with focus on WSO2 EI, the solution 2 above was selected to realise the components of the Data Spine. The details of this experimental evaluation can be found in D3.1. The subsequent sections introduce the technological platform, tool or service chosen to realise that particular component of the Data Spine and elaborate its design and technological aspects such as architecture, interfaces, configuration, operation, etc.

2.2.1 Integration Flow Engine

As stated in Section 2.2.6, Apache NiFi was selected to be the Integration Flow Engine of the Data Spine. Apache NiFi 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.

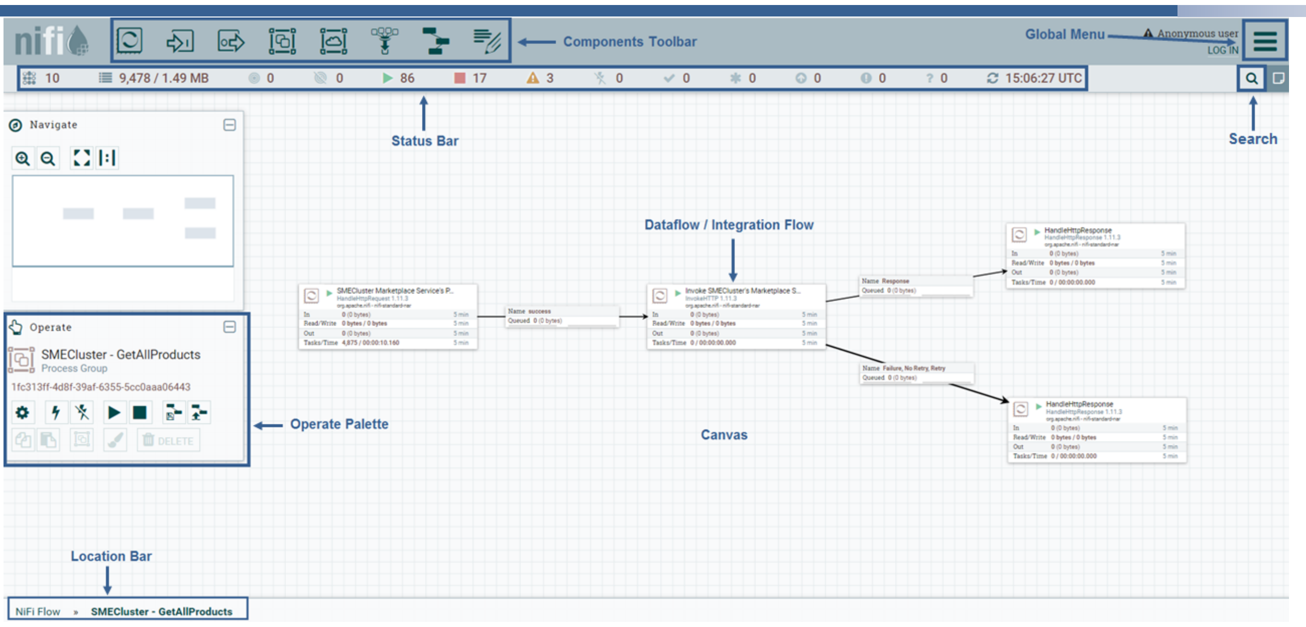
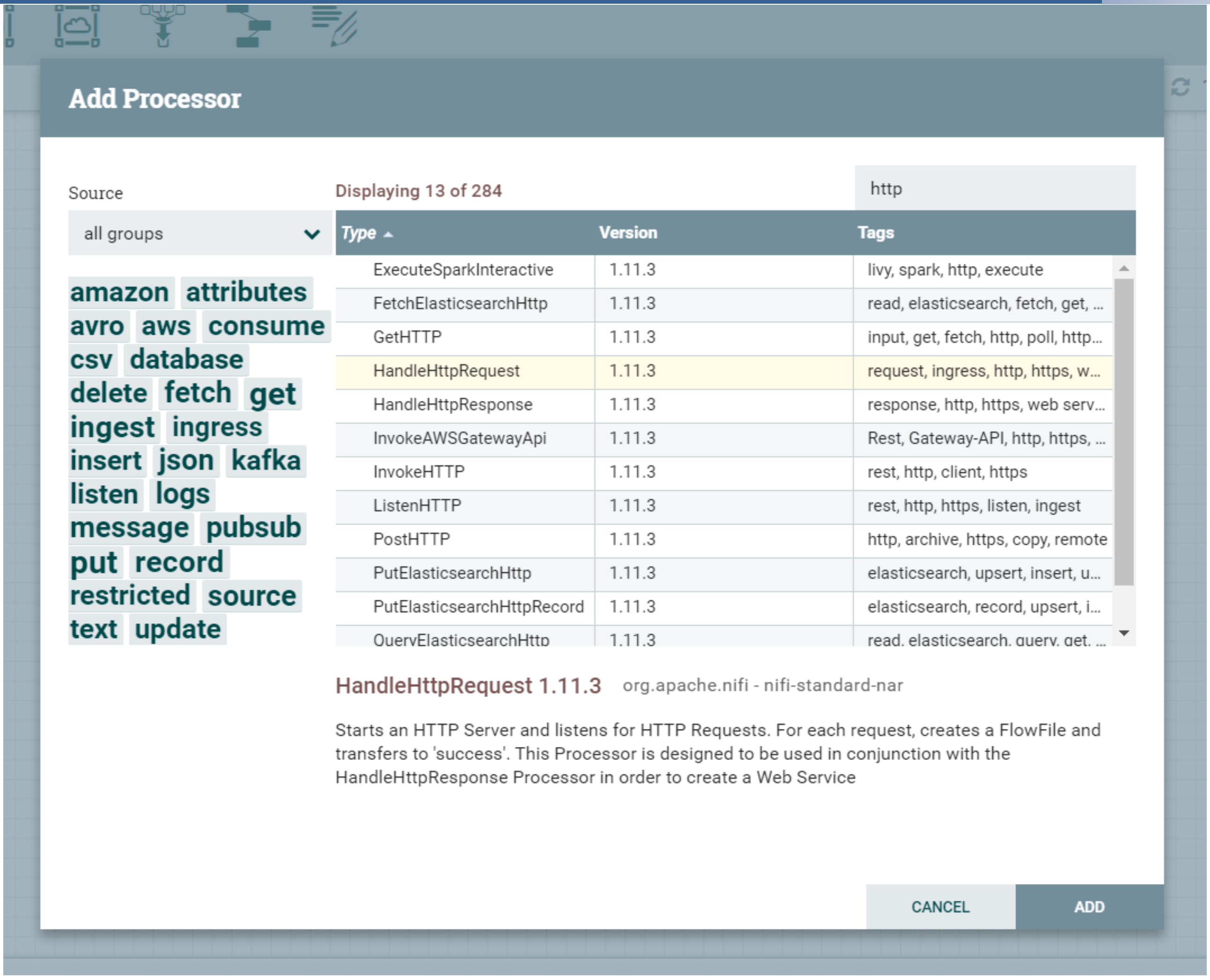


Figure 46: Apache NiFi GUI Elements

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 46 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 47. 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.


Add Processor
Source
all groups
amazon attributes avro aws consume csv database delete fetch get ingest ingress insert json kafka listen logs message pubsub put record restricted source text update
CANCEL
ADD

Displaying 13 of 284

http

Type	Version	Tags
ExecuteSparkInteractive	1.11.3	livy, spark, http, execute
FetchElasticsearchHttp	1.11.3	read, elasticsearch, fetch, get, ...
GetHTTP	1.11.3	input, get, fetch, http, poll, http...
HandleHttpRequest	1.11.3	request, ingress, http, https, w...
HandleHttpResponse	1.11.3	response, http, https, web serv...
InvokeAWSGatewayApi	1.11.3	Rest, Gateway-API, http, https, ...
InvokeHTTP	1.11.3	rest, http, client, https
ListenHTTP	1.11.3	rest, http, https, listen, ingest
PostHTTP	1.11.3	http, archive, https, copy, remote
PutElasticsearchHttp	1.11.3	elasticsearch, upsert, insert, u...
PutElasticsearchHttpRecord	1.11.3	elasticsearch, record, upsert, i...
QueryElasticsearchHttp	1.11.3	read, elasticsearch, query, get, ...

HandleHttpRequest 1.11.3 org.apache.nifi - nifi-standard-nar

Starts an HTTP Server and listens for HTTP Requests. For each request, creates a FlowFile and transfers to 'success'. This Processor is designed to be used in conjunction with the HandleHttpResponse Processor in order to create a Web Service

Figure 47: Apache NiFi Processors

Apache NiFi’s conformity to the fundamental design requirements for Integration Flow Engine identified in Section 2.1.1 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 primarily provides three data transformation processors: JoltTransformJSON, TransformXml and ExecuteScript. 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. More information on the Data Transformation Processors can be found in Section 4.3.

- **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 though 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 found to be a suitable candidate to realise the Integration Flow Engine of the Data Spine.

2.2.1.1 Architecture and Interfaces

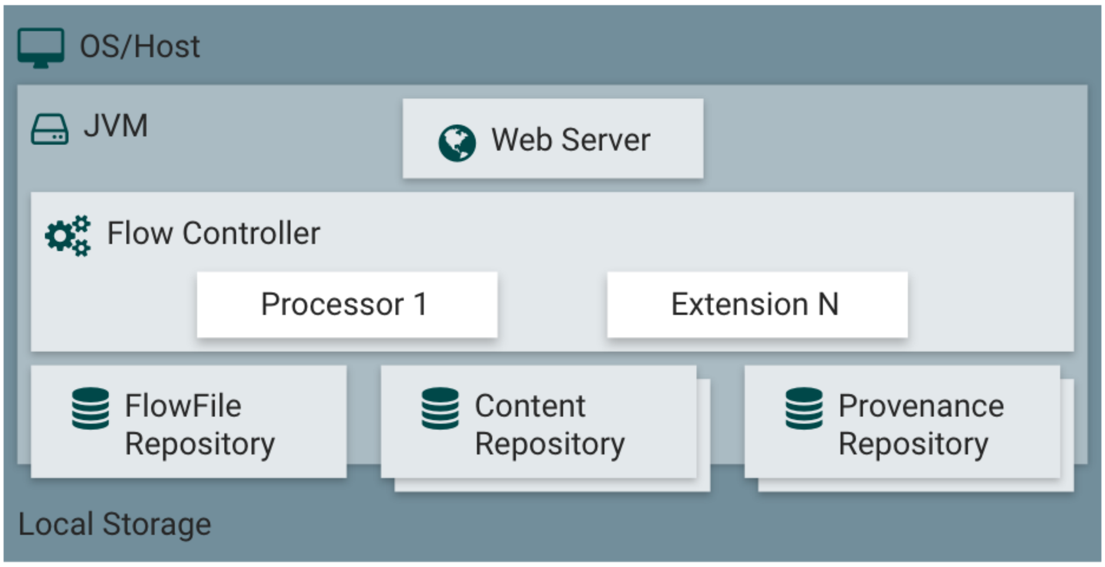


Figure 48: Architecture of Apache NiFi [NOG20]

Figure 48 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.

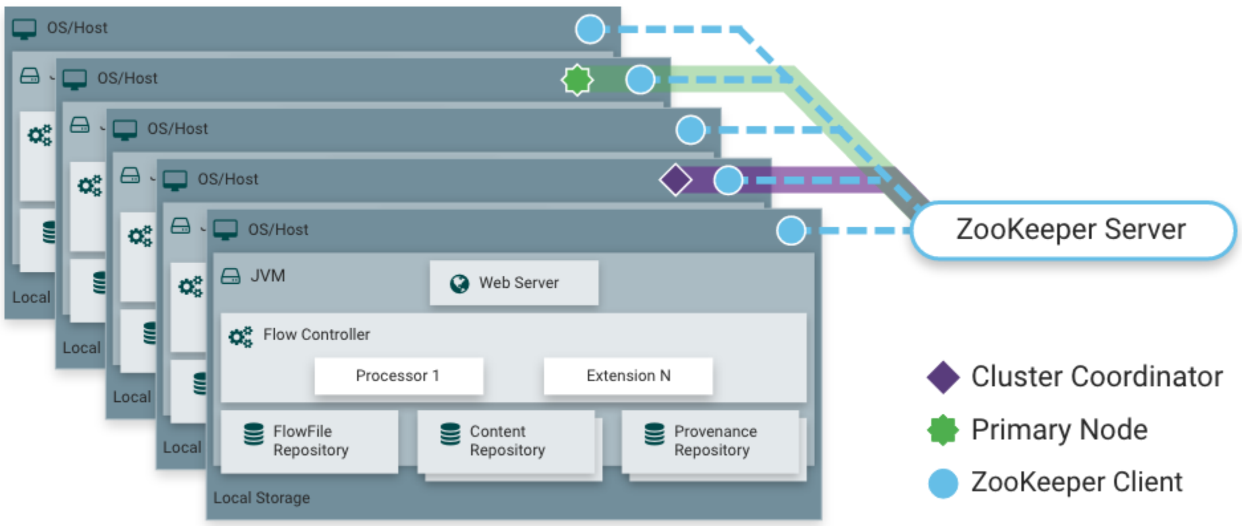


Figure 49: Apache NiFi Cluster

NiFi also supports scaling-out though the use of clustering. Figure 49 shows the operation of NiFi within a cluster. NiFi employs a Zero-Master Clustering Paradigm. Every node in a particular cluster of NiFi perform the same operation on data but operate on different sets of data. Apache ZooKeeper is used for leader election. It elects a NiFi node as a Cluster Coordinator. All other nodes report heartbeat and status information to this Cluster Coordinator. Thus, Cluster Coordinator handles joining and leaving of nodes. When a Cluster Coordinator fails, ZooKeeper automatically elects another node as a new Cluster Coordinator. The GUI of any node can be used to manage the cluster and any changes made to a particular node are automatically replicated across the cluster.

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.

2.2.1.2 Configuration

In order for two heterogeneous services to be able to communicate, they must be integrated and interoperated though the Data Spine first 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 EFPF, NiFi's GUI is secured using EFS's Identity Provider via OIDC. To enable user authenticated via OIDC, the properties as shown in Figure 50 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 50: NiFi OIDC Properties Configuration

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 it '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 hardcoded 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 51 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 are 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	TBD	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	TBD	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	TBD	TBD	access counters	Allows users to view/modify Counters

Figure 51: 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 52 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 52: 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 uses that are applicable to that particular component (Process Group, Template, etc.). Figure 53 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 groups 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	TBD	receive data via site-to-site	Allows a port to receive data from NiFi instances
Y	TBD	send data via site-to-site	Allows a port to send data from NiFi instances

Figure 53: Component-level Access Policies for a particular component (Process Group) PG_X

Figure 54 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.

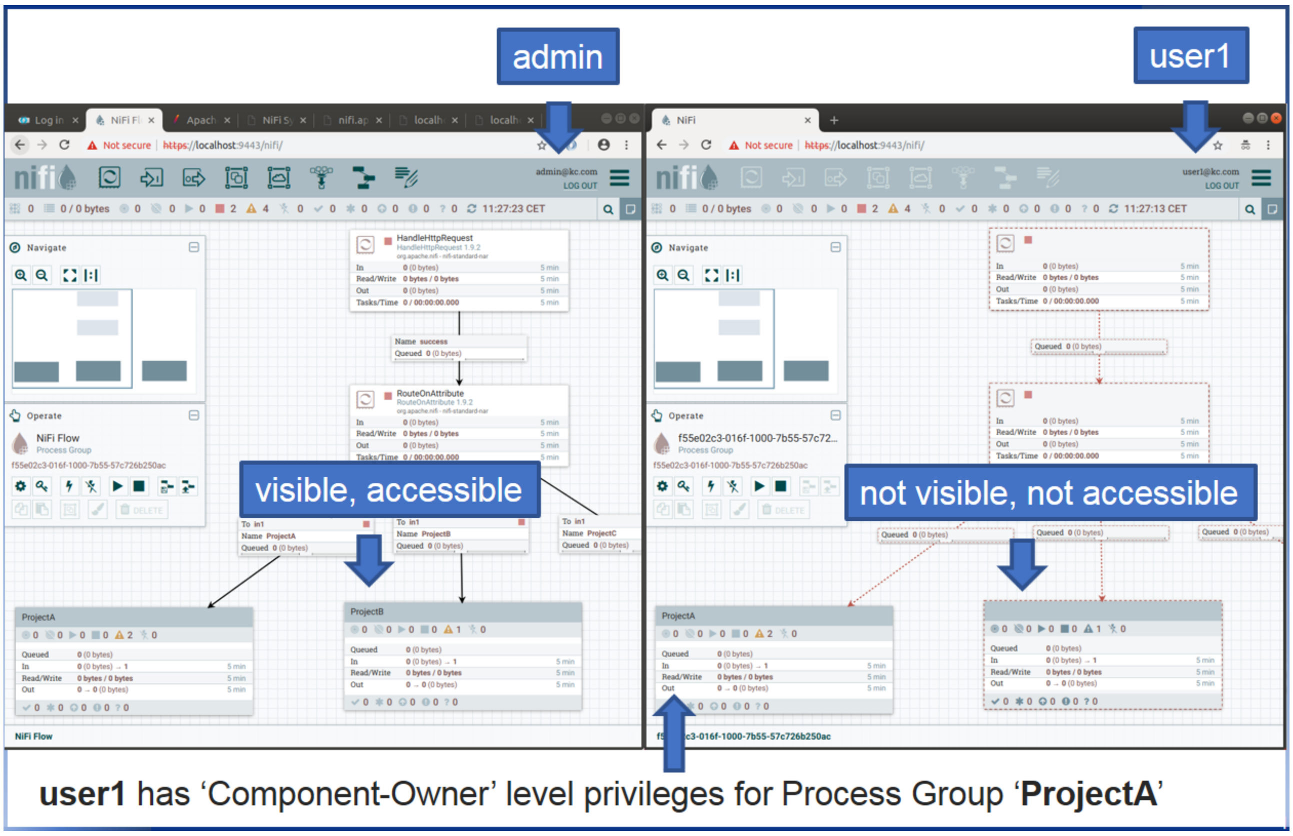

admin
user1
visible, accessible
not visible, not accessible
user1 has 'Component-Owner' level privileges for Process Group '**ProjectA**'

Figure 54: 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[9] endpoints for external services and access to these endpoints needs to be secured. As shown in Figure 55, 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

[9] 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

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.

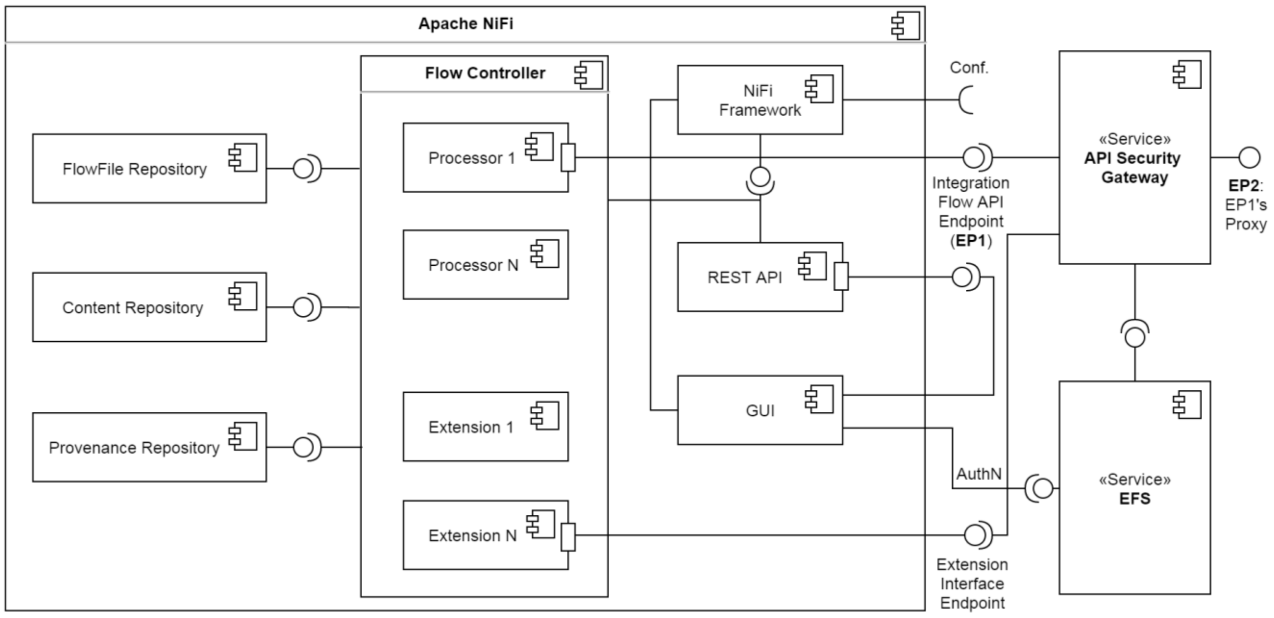
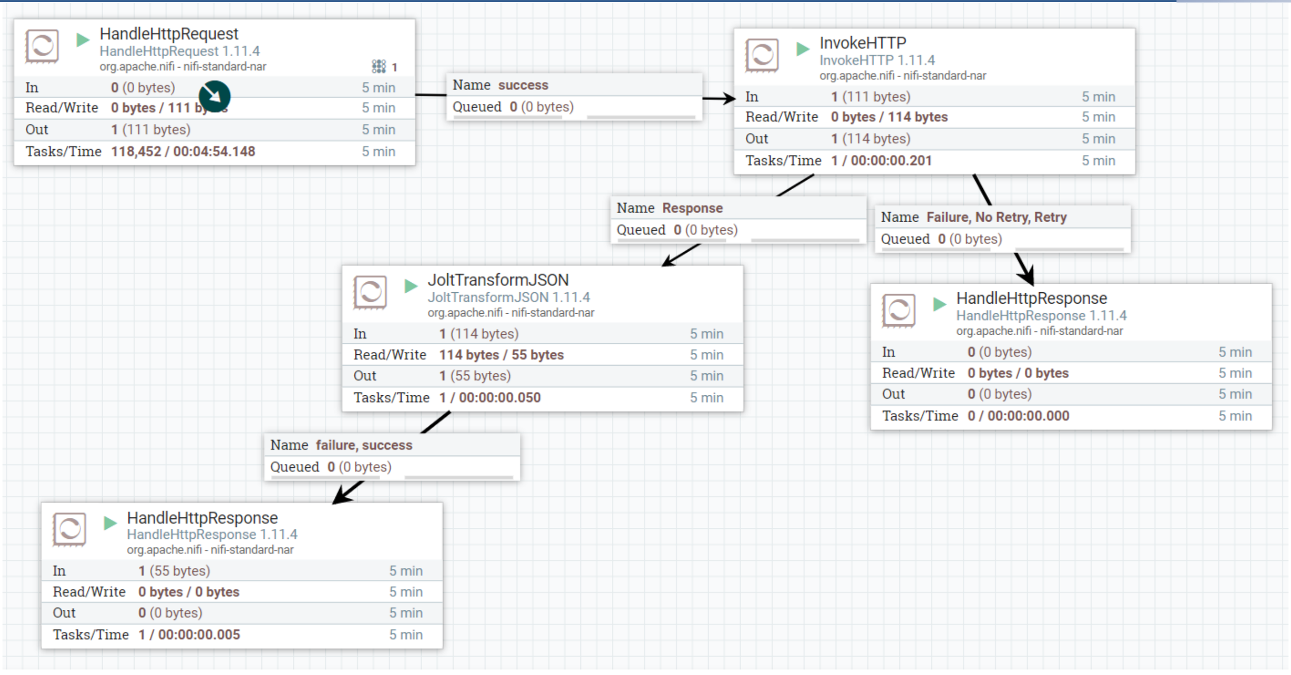


Figure 55: Data Spine NiFi Security Integration

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

2.2.1.3 Operation

Figure 56 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.



HandleHttpRequest
HandleHttpRequest 1.11.4
org.apache.nifi - nifi-standard-nar
1

In	0 (0 bytes)	5 min
Read/Write	0 bytes / 111 bytes	5 min
Out	1 (111 bytes)	5 min
Tasks/Time	118,452 / 00:04:54.148	5 min

Name success
Queued 0 (0 bytes)

InvokeHTTP
InvokeHTTP 1.11.4
org.apache.nifi - nifi-standard-nar

In	1 (111 bytes)	5 min
Read/Write	0 bytes / 114 bytes	5 min
Out	1 (114 bytes)	5 min
Tasks/Time	1 / 00:00:00.201	5 min

Name Response
Queued 0 (0 bytes)

Name Failure, No Retry, Retry
Queued 0 (0 bytes)

JoltTransformJSON
JoltTransformJSON 1.11.4
org.apache.nifi - nifi-standard-nar

In	1 (114 bytes)	5 min
Read/Write	114 bytes / 55 bytes	5 min
Out	1 (55 bytes)	5 min
Tasks/Time	1 / 00:00:00.050	5 min

HandleHttpResponse
HandleHttpResponse 1.11.4
org.apache.nifi - nifi-standard-nar

In	0 (0 bytes)	5 min
Read/Write	0 bytes / 0 bytes	5 min
Out	0 (0 bytes)	5 min
Tasks/Time	0 / 00:00:00.000	5 min

Name failure, success
Queued 0 (0 bytes)

HandleHttpResponse
HandleHttpResponse 1.11.4
org.apache.nifi - nifi-standard-nar

In	1 (55 bytes)	5 min
Read/Write	0 bytes / 0 bytes	5 min
Out	0 (0 bytes)	5 min
Tasks/Time	1 / 00:00:00.005	5 min

Figure 56: Integration Flow Example

2.2.2 API Security Gateway

The API service security is handled by the API Security Gateway (ASG). The ASG acts as a border gateway ahead of all API calls to the Data Spine. Its role is to enforce security policies on the service calls. In EFPF, ASG is implemented using Apache APISIX, which is a technology solution selected due to its 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 57 compares core features of Apache APISIX and Kong 2.0 Open Source API Gateway [KON20].

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

Figure 57: Comparison of Apache APISIX and Kong 2.0 API Gateways

Due to the above advantages the ASG in EFPF is implemented using Apache APISIX, which is an effort currently undergoing incubation at the Apache Software Foundation (ASF) (for more details: https://github.com/apache/incubator-apisix#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.

2.2.2.1 Architecture and Interfaces

Figure 58 shows the workflow diagram of the API Security Gateway.

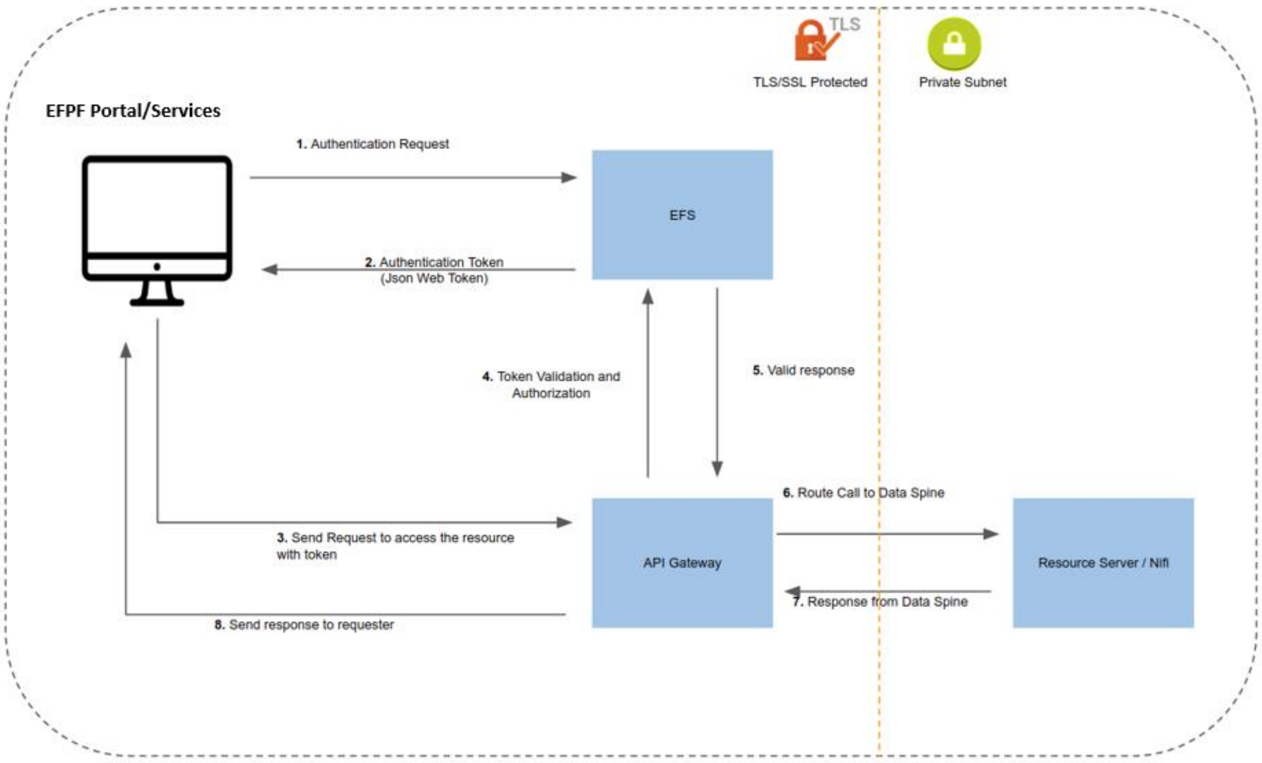

TLS
TLS/SSL Protected
Private Subnet
EFPF Portal/Services
1. Authentication Request
EFS
2. Authentication Token (Json Web Token)
4. Token Validation and Authorization
5. Valid response
6. Route Call to Data Spine
3. Send Request to access the resource with token
API Gateway
Resource Server / Nifi
7. Response from Data Spine
8. Send response to requester

Figure 58: General Communication Workflow involving the API Security Gateway

Figure 58 shows the basic communication workflow around the ASG (note, ASG is represented by the API Gateway (blue 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 (blue 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 EFPF identity server and does basic authorization via JSON web token 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.

2.2.2.2 Configuration

The configuration of the API Security Gateway involves three main steps.

1. **Configure connecting with Service Registry**: Service registry contains the services registered in the EFPF platform. The API Gateway performs a periodic scanning of the services to create routes to the services.
2. **Configure Open ID connect Plugin**: Open ID connect plugins perform introspection of the EFPF token. The ASG should be configured to communicate with the EFPF identity server 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 API Gateway. This plugin should be configured by stating the upstream resource of the route and the scope of the operation.

2.2.2.3 Operation

The ASG is available via Docker 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 to monitor the ASG.

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

2.2.3 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. Its functionality mainly covers the lifecycle management of services i.e. the registration, viewing, updating and deregistration of services' metadata. In addition, it supports browsing of the service entries in the Catalog and provides a service filtering functionality that can be used by service consumers to search services by known capabilities.

Figure 59 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 2.1.3.

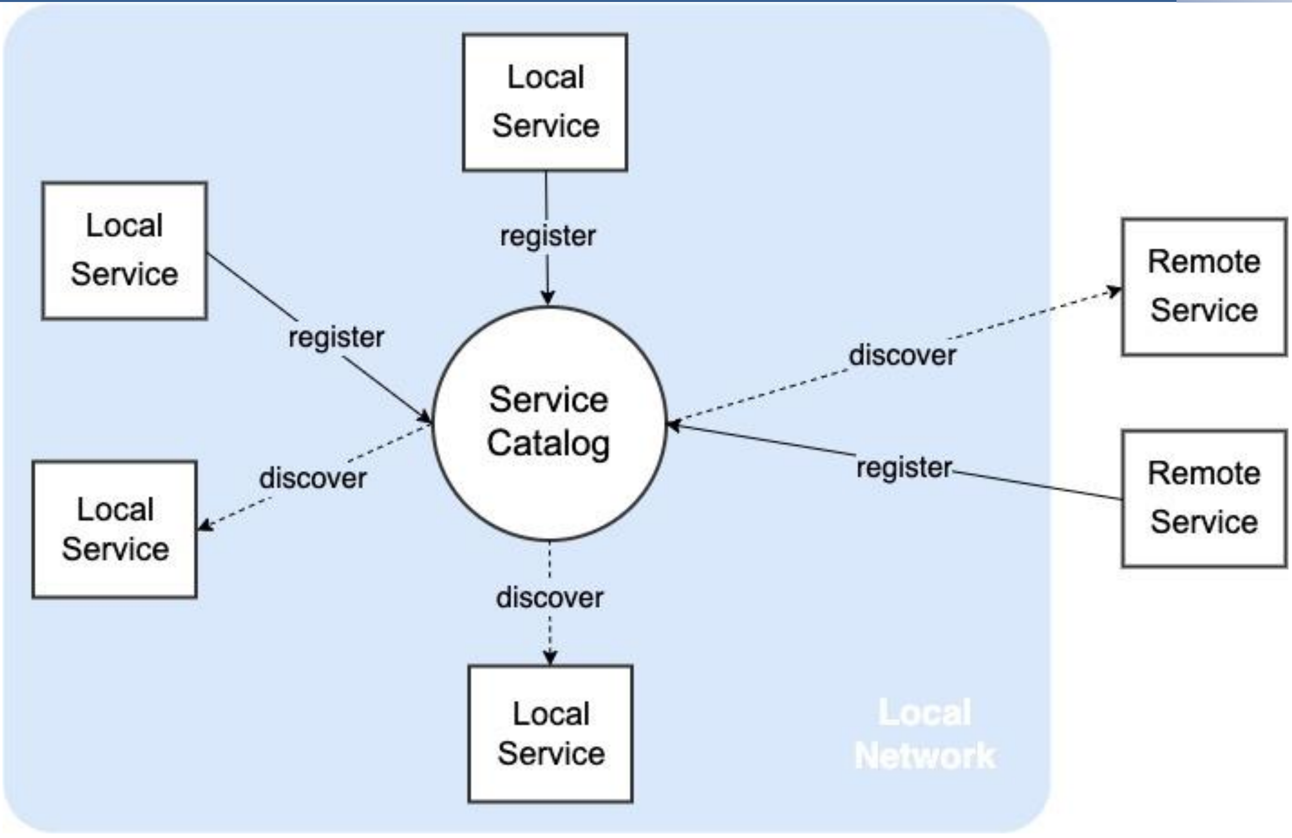


Figure 59: LinkSmart Service Catalog

2.2.3.1 Architecture and Interfaces

The schema of LinkSmart Service Catalog was updated as per the design requirements identified in Section 2.1.3. The new schema (see Figure 60) is capable of managing metadata for synchronous (request-response) as well as asynchronous (publish-subscribe) type of services. The schema can be extended to include additional metadata for the entire service or for a specific API. E.g., Figure 61 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 60: 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 61: Extended Service Description Schema of the Service Registry

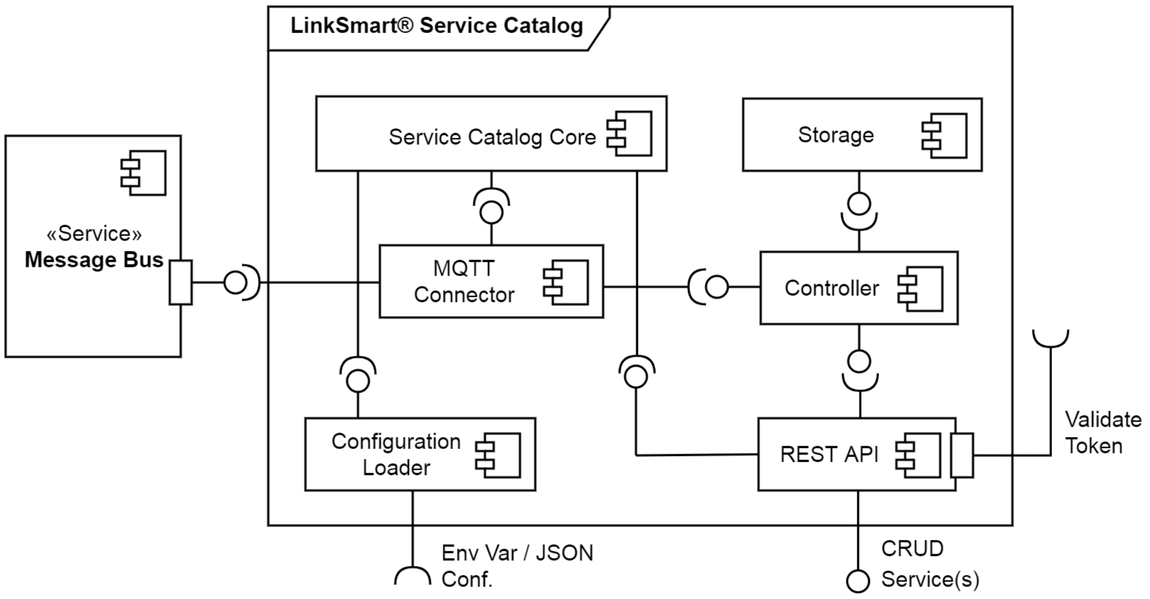


Figure 62: Architecture of LinkSmart Service Catalog

The Service Catalog, as illustrated in its UML component diagram in Figure 62, 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 Catalog consumes a JSON configuration file which contains the MQTT configuration as described in Section 2.2.3.2. The Service Catalog uses LevelDB on-disk key-value store to persist data. Finally, the Service Catalog can optionally consume an authentication provider's interface to secure access to its APIs; however, this wouldn't be used as access to Service Registry's APIs would be secured through the API Security Gateway and EFS. These APIs of the Service Registry are described below.

HTTP REST API for Lifecycle Management and Discovery of Services:

Figure 63 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 63: Service Registry HTTP REST API

Figure 64 shows the data model of the Service Registry.

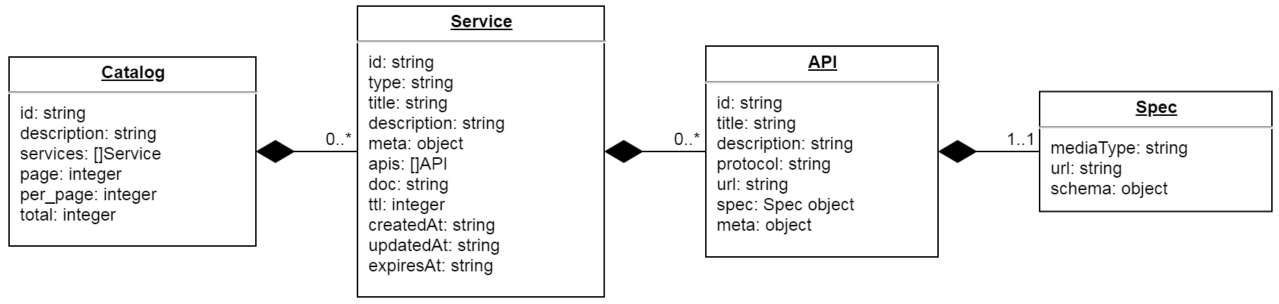


Figure 64: Data Model of the Service Registry

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

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'
```

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'
```

2.2.3.2 Configuration

The Service Catalog (SC) consumes a JSON configuration file which is shown in Figure 65.

```
{
  "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 65: 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.

2.2.3.3 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.

2.2.4 Message Bus

RabbitMQ [RMQ20] Message Broker satisfies the design requirements the Data Spine Message Bus enlisted in Section 2.1.4. RabbitMQ is a message broker or message-oriented middleware that implements AMQP (Advanced Message Queuing Protocol). 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 inherently and MQTT/MQTTS via a plugin
  - 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 a management command line tool 'rabbitmqadmin' that can be used in performance critical environments such as production as opposed to the GUI.
- **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.

2.2.4.1 Architecture and Interfaces

RabbitMQ is an implementation of the AMQP protocol. The AMQP 0-9-1 model followed by RabbitMQ is shown in Figure 66. 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 subscribes. 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 that messages from a queue.

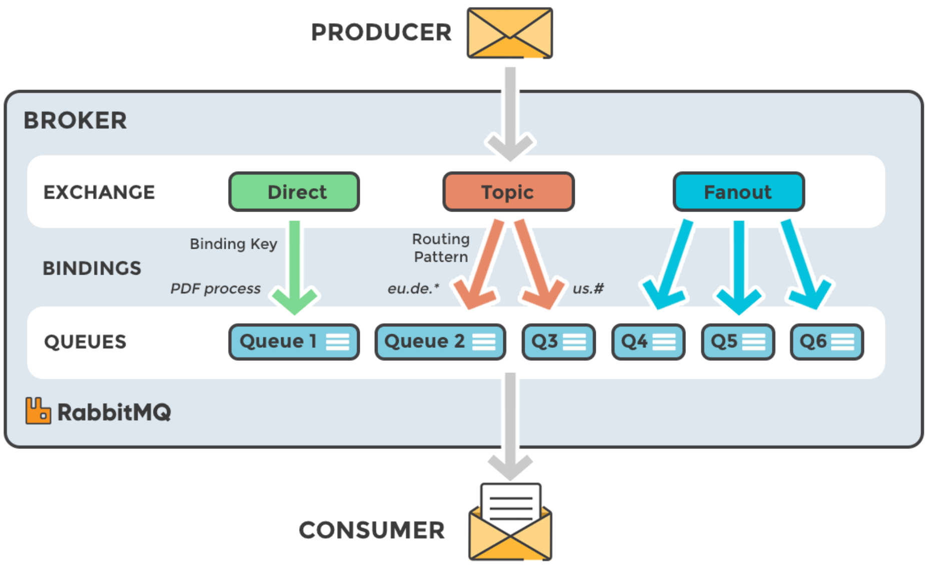


Figure 66: AMQP 0-9-1 Model Followed by RabbitMQ [CAQ20]

The exchanges are of different types. 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. The type of an exchange is specified when it is created.

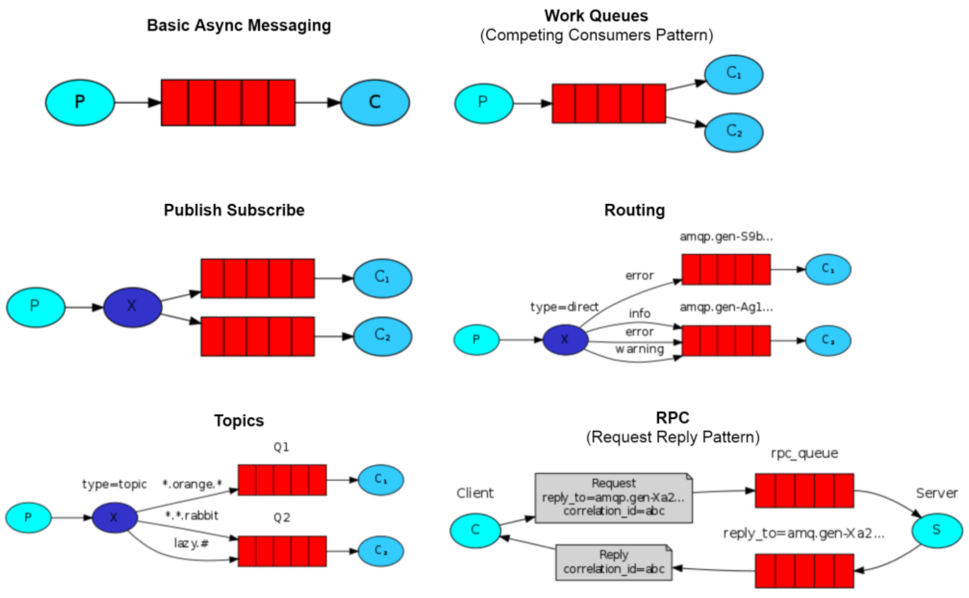
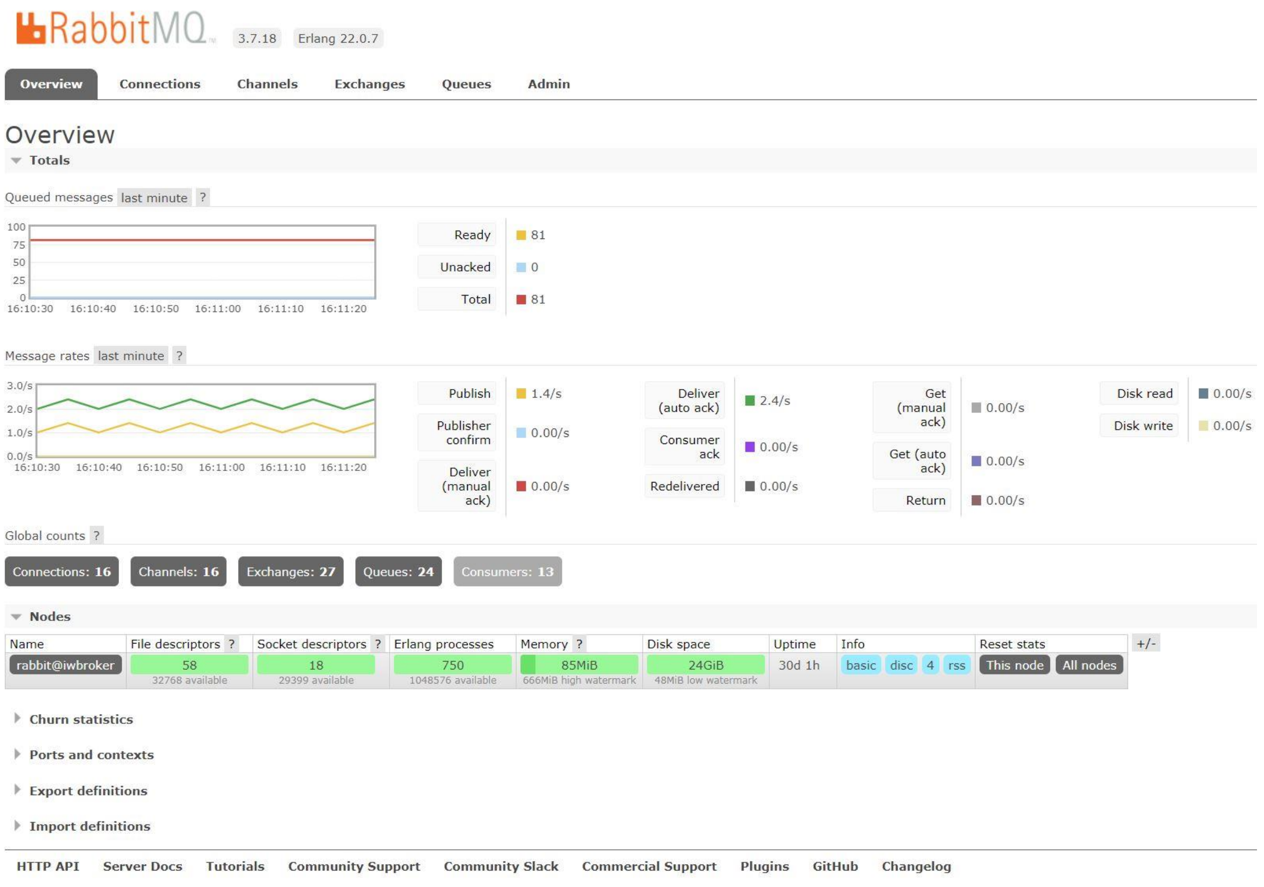


Figure 67: RabbitMQ Messaging Patterns [RMP20]

RabbitMQ supports several messaging patterns as shown in Figure 67. 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.



Ready 81
Unacked 0
Total 81

Publish 1.4/s
Publisher confirm 0.00/s
Deliver (manual ack) 0.00/s
Deliver (auto ack) 2.4/s
Consumer ack 0.00/s
Redelivered 0.00/s
Get (manual ack) 0.00/s
Get (auto ack) 0.00/s
Return 0.00/s
Disk read 0.00/s
Disk write 0.00/s

Connections: 16 Channels: 16 Exchanges: 27 Queues: 24 Consumers: 13

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info	Reset stats	+/-
rabbit@iwbroker	58 32768 available	18 29399 available	750 1048576 available	85MiB 666MiB high watermark	24GiB 48MiB low watermark	30d 1h	basic disc 4 rss	This node All nodes	

Figure 68: RabbitMQ Management GUI

RabbitMQ's Management plugin provides an HTTP API, a Web-based GUI (Figure 68) and a CLI for management and monitoring of RabbitMQ nodes and clusters. The CLI is provided via a command line tool 'rabitmqadmin'. rabitmqadmin 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 in development and testing environments.

2.2.4.2 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-diagnotics, 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.

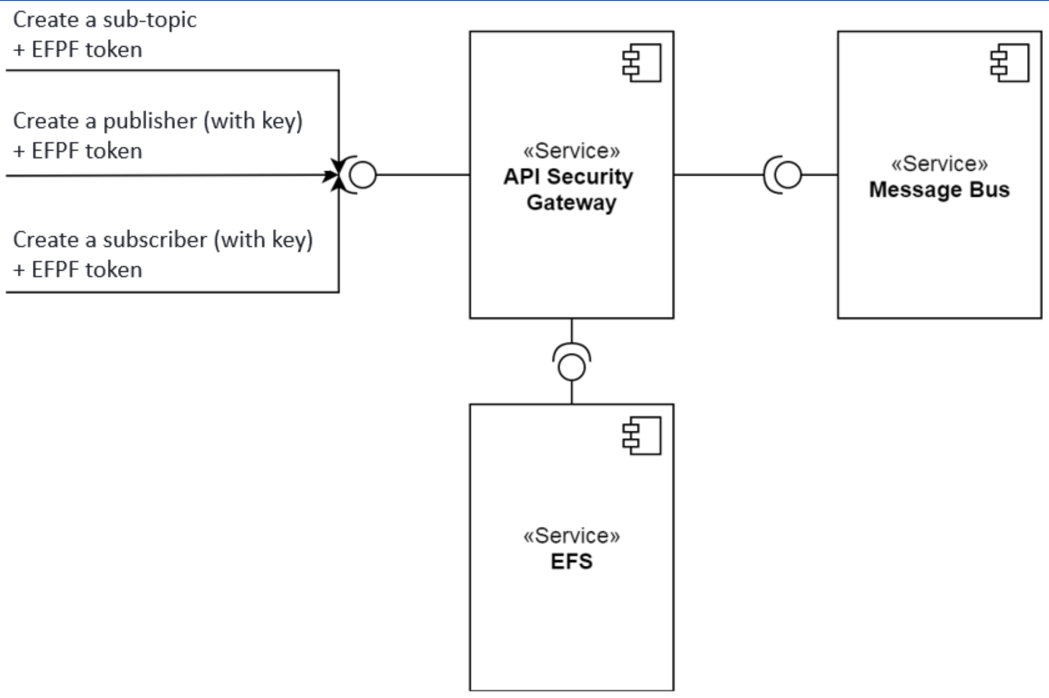


Figure 69: Design-time Security Configuration for PubSub Workflows

Figure 69 shows the design-time security configuration that needs to be done in order for enabling secure communication and dataflow between the Message Bus and the clients. The first step is to obtain a root topic from the EFS. The users with ‘EFPF Connector’ role can invoke EFS’s API to get such a root topic for their company and the clients belonging to this company need to create sub-topics under this root topic and then create publishers and subscribers for such sub-topics. The API Security Gateway provides a proxy API to offer these functionalities. Upon receiving such a request, the API Security Gateway and EFS check whether the root topic assigned to his/her company matches with the one decoded from his/her token. If so, the request is authorised and the requested action is performed. The clients this get the keys to publish/subscribe to the topics on the Message Bus. The exact API calls and/or the mechanism that will enable this kind of security configuration in RabbitMQ is currently being investigated and discussed.

2.2.4.3 Operation

Once the configuration on both the RabbitMQ side and the clients’ side is done, the communication can start. In the case of MQTT, 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.

2.2.5 EFPF Security Portal

In order for the realization of EFS, EFPF uses a combination of micro services and a Keycloak identity server. The following lists the main functionalities performed by the EFS.

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

2.2.5.1 Architecture and Interfaces

User Management

The user management and authentication is handled by the Keycloak Identity Server. Keycloak is an OpenId connect and UMA compliant identity provider. Keycloak also provides an Admin API to perform user user management and a fully extensible plugin based ecosystem.

User Federation Service

To enable user's login to any of the four base platforms, the user can select one of the two procedures for the authorization:

- Login through a base 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, shown in Figure 70 and Figure 71 respectively. Note that both workflows require the user to be registered on the EFPF platform.

Figure 70 illustrates the workflow that follows the bottom up approach for federation. Here, the user logins to the base platform using his/her EFPF credentials. The EFPF credentials are issued by the EFS and are not propagated to the base platforms. The initial representation of the user will be created when the user opts to login with EFPF credentials in a base platform. If a user is already present in the base platform then a linked user will be created with the existing roles of the base platform.

The second workflow (Figure 71) enables the user to login to the EFPF platform (via EFS) and then visit any base platform in the same browser session, e.g. PLATFORM 1, PLATFORM 2, etc. In this approach, the user logins to the base platforms using his/her previously provided EFPF credentials. By keeping the common browser session, the EFPF user can achieve SSO capability when login to other base platforms. The current implementation of the EFPF EFS (with IDS) follows the OpenID based identity method. Here, the user who tries to login to the base platform using EFPF credentials, is redirected to the EFPF platform (and its EFS) for the validation of his/her credentials. After verifying the credentials, the user is redirected to the relevant base platform. Furthermore, the base platform identifies the user and provides required roles based on predefined policies.

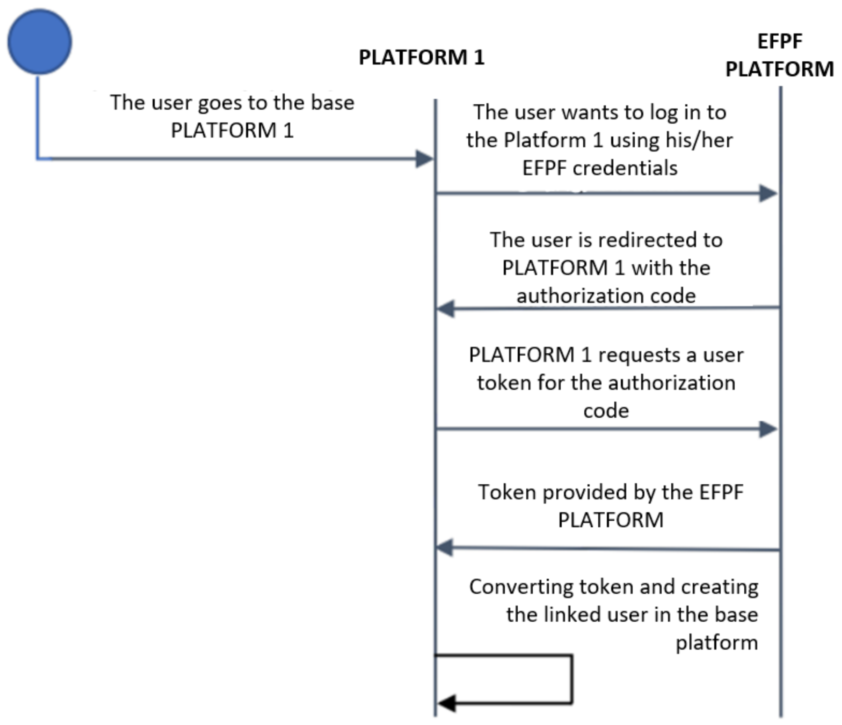

PLATFORM 1
EFPF PLATFORM
The user goes to the base PLATFORM 1
The user wants to log in to the Platform 1 using his/her EFPF credentials
The user is redirected to PLATFORM 1 with the authorization code
PLATFORM 1 requests a user token for the authorization code
Token provided by the EFPF PLATFORM
Converting token and creating the linked user in the base platform

Figure 70: Workflow 1: Login to PLATFORM 1 using EFPF platform credentials

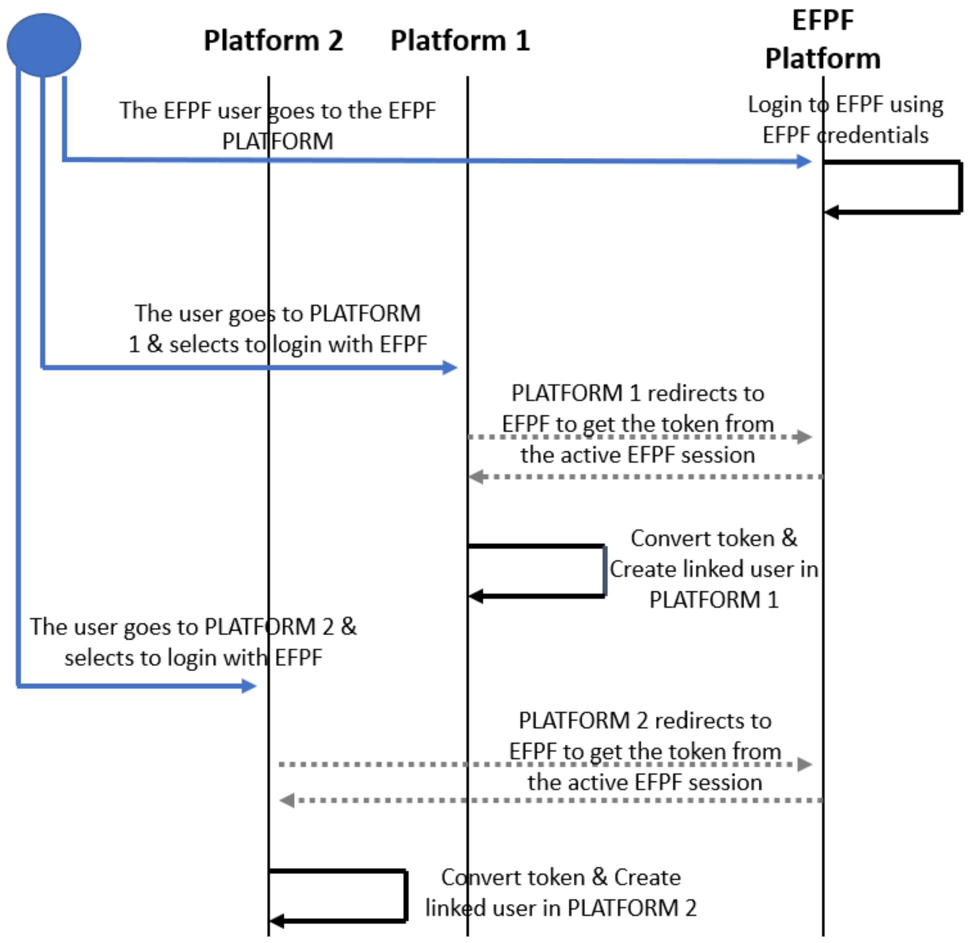

Platform 2
Platform 1
EFPF Platform
The EFPF user goes to the EFPF PLATFORM
Login to EFPF using EFPF credentials
The user goes to PLATFORM 1 & selects to login with EFPF
PLATFORM 1 redirects to EFPF to get the token from the active EFPF session
Convert token & Create linked user in PLATFORM 1
The user goes to PLATFORM 2 & selects to login with EFPF
PLATFORM 2 redirects to EFPF to get the token from the active EFPF session
Convert token & Create linked user in PLATFORM 2

Figure 71: Workflow 2: Login to EFPF using EFPF 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 us to enforce policies to services exposed via the Data Spine. EFS allows to create 2 types of permissions:

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

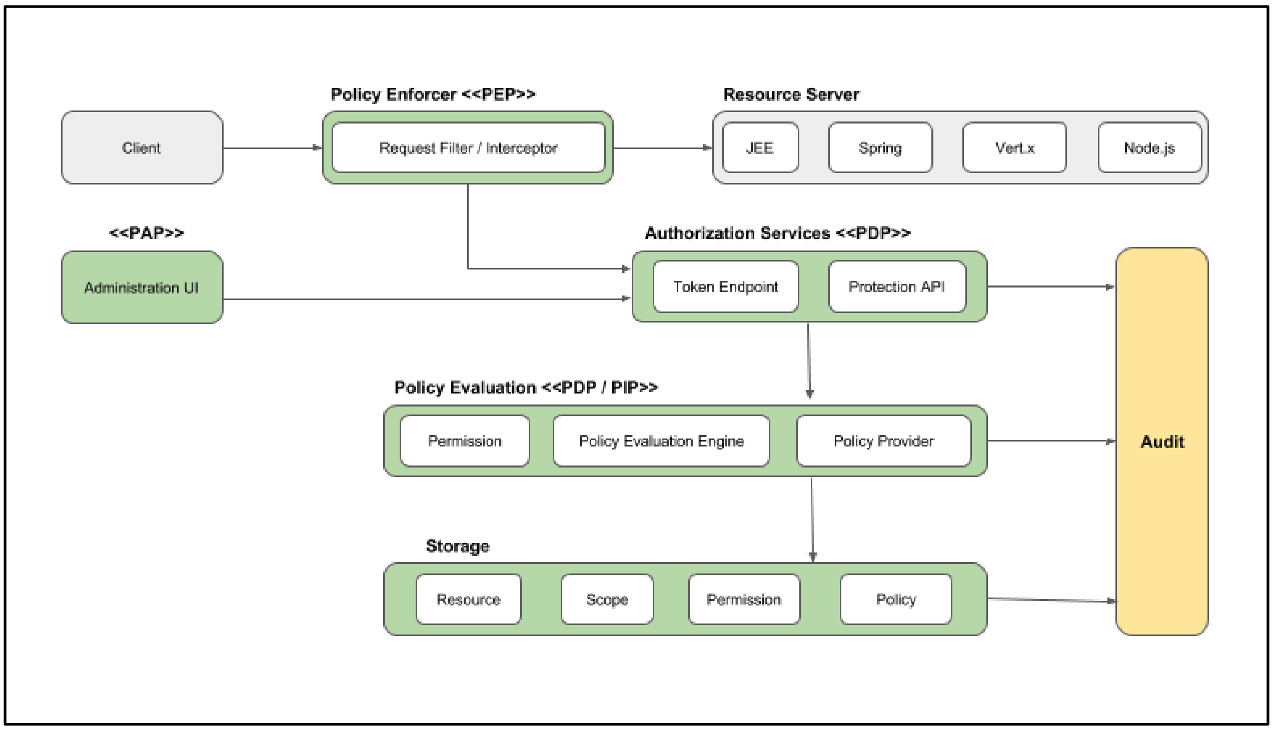


Figure 72: Policy Enforcement Architecture

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

Token Translation Service

Base platforms have their internal authentication and authorization mechanisms, and use the token generated by their authentication system to allow access. Therefore EFPF users cannot access the connected base platforms with the EFPF token generated by EFS. Token translation service acts as an intermediary service to obtain a valid access token to external API calls.

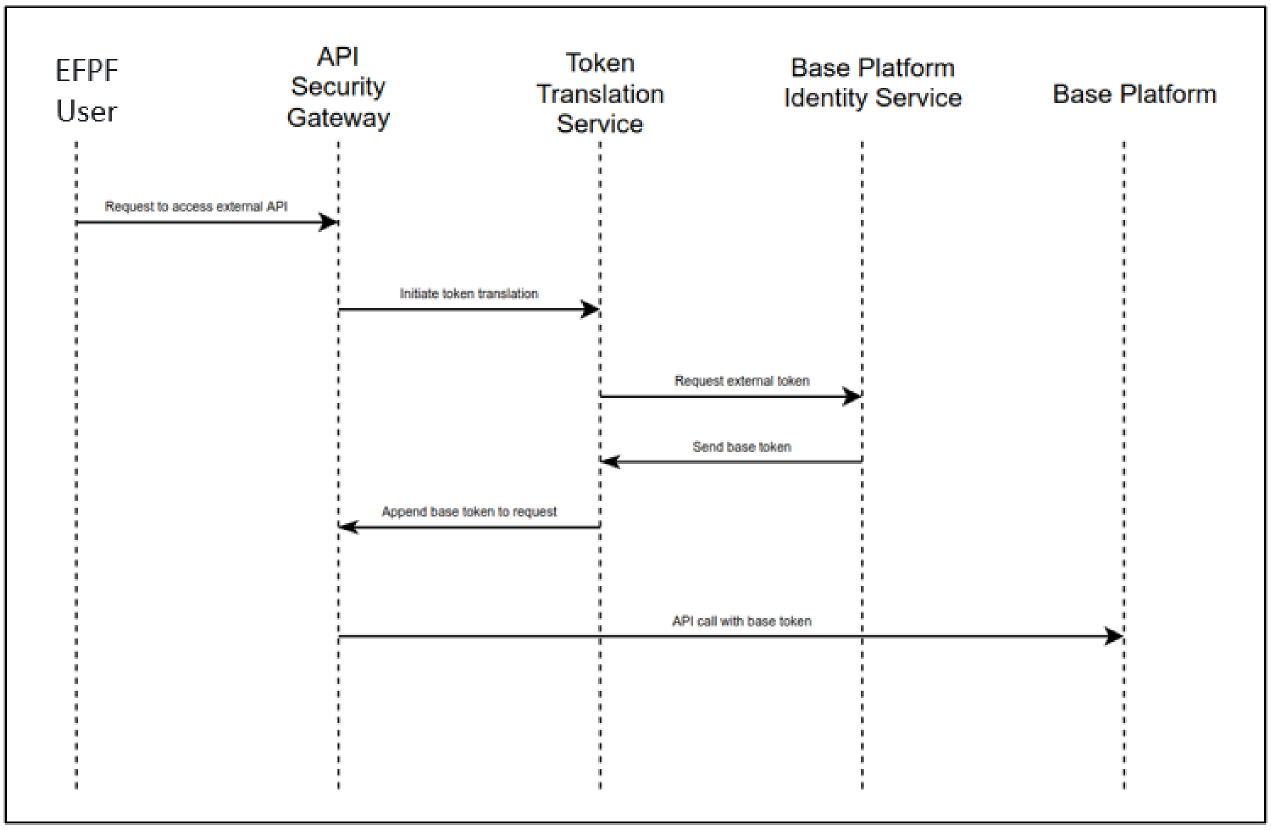

EFPF User
API Security Gateway
Token Translation Service
Base Platform Identity Service
Base Platform
Request to access external API
Initiate token translation
Request external token
Send base token
Append base token to request
API call with base token

Figure 73: Sequence Diagram of the Token Translation Service

2.2.5.2 Configuration

Identity Server Configuration

The identity provider is configured to use the Open ID connect based authentication to authentication and authorize the users. Trusted clients should be created in the identity server for the application to authenticate users and micro services in the ecosystem. In addition to authentication the authorization aspect of the identity server.

SSO Configurations

A dedicated client should be created in the identity server for each base platform’s SSO configurations. The SSO client will have the redirect URL and other Open ID configurations such as the Oauth flows supported for the client. By default the identity server supports the standard flows. However if the base platform does not have a back end, then the implicit flow can be enabled to exchange SSO tokens.

2.2.5.3 Operation

The components of the EFS are packaged as 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 and on a periodic basis exported to the Kibana log management platform.

The identity server also provides several functionalities to ensure smooth operation such as detection of brute forcing or any suspicious activities.

2.2.6 The Data Spine

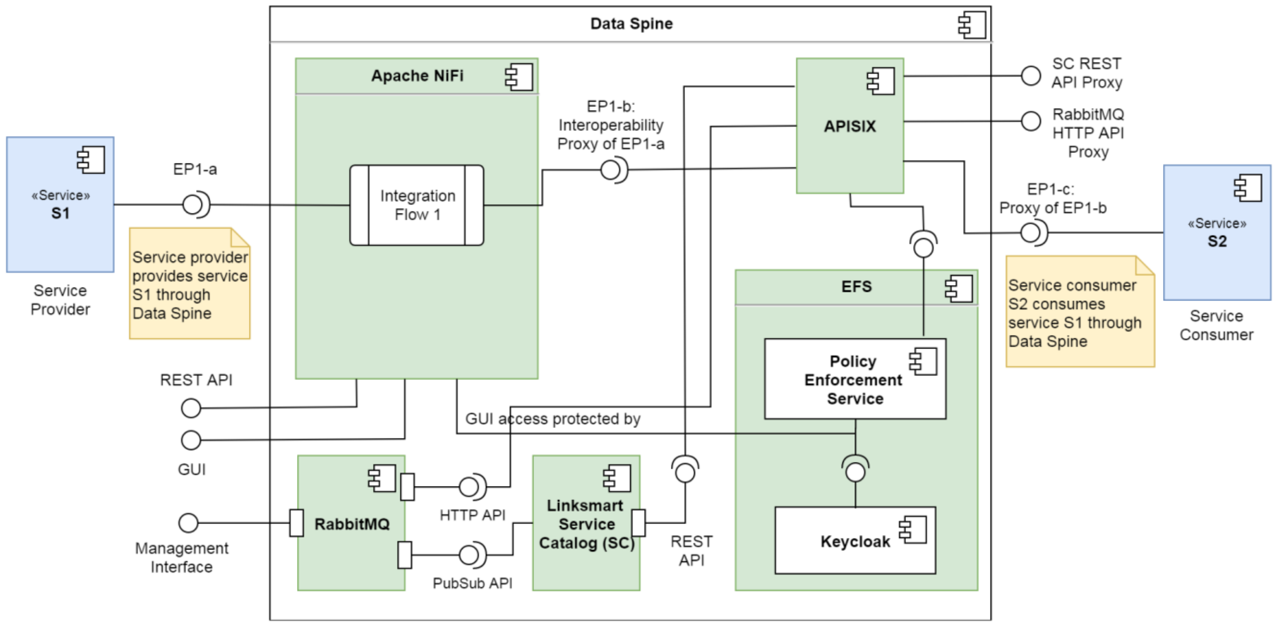


Figure 74: Data Spine Realised

Figure 74 illustrates the technological platforms, tools and services selected to realise different components the Data Spine and the interactions between them. The interactions are similar to the ones between their respective conceptual components illustrated in Figure 45 and explained in Section 2.1.6. The figure also shows Service Provider providing service S1 through the Data Spine and Service Consumer's service S2 consuming S1 through the Data Spine with the help of 'Integration Flow 1'. Finally, Figure 75 summarises the technologies selected to realise the conceptual components of the Data Spine.
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.

Conceptual Component	Technology
Integration Flow Engine	Apache NiFi
API Security Gateway	APISIX
Service Registry	LinkSmart Service Catalog
Message Bus	RabbitMQ
EFS	Keycloak (& Policy Enforcement Service)

Figure 75: Technologies Selected to Realise the Data Spine

2.3 Service Integration through Data Spine

This section enlists the activities that the service providers need to do in order to provide their services through the Data Spine and the activities the service consumers need to do in order to consume the services provided through the Data Spine with the help of examples. These design-time aspects become the prerequisites to enabling communication through the Data Spine. This section is further divided into 'Synchronous Communication' and 'Asynchronous Communication' as the activities differ for both.

- **Synchronous Communication:**

Prerequisites:

The service provider ‘provider1’ and service consumer ‘consumer1’ are both EFPF users. provider1 and consumer1 have the necessary access permissions required to register services to the Service Registry.

Design-time activities:

1. provider1 registers his/her service ‘service1’ to the Data Spine Service Registry with an appropriate service ‘type’. For simplicity, let us assume that service1 has only one API endpoint ‘EP1’.
2. An EFPF administrator user ‘admin1’ defines/configures the access permissions for accessing EP1 in the EFS.
3. The API Security Gateway (ASG), which checks for new service registrations/updates to services in the Service Registry periodically, creates a proxy endpoint/route ‘$EP1_P$’ for EP1 in ASG – this could be used to invoke the endpoint directly without creating an integration flow in case if protocol translation and/or data transformation isn’t needed (here it is assumed that data transformation is needed).
4. consumer1 discovers service1 and decides to consume it and gets the technical metadata for service1 including its API spec from the Service Registry.
5. consumer1 requests for and acquires the necessary access permissions to invoke EP1.
6. consumer1 creates an integration flow using the GUI of the Integration Flow Engine to invoke EP1, perform data transformation and finally create an ‘interoperability-proxy’ endpoint EP1-C for EP1 in the integration flow.
7. consumer1 registers this new EP1-C endpoint to the Service Registry.
8. ASG creates a proxy endpoint EP1-C_P for EP1-C.
9. consumer1 requests for and acquires the necessary access permissions to invoke EP1-C_P.
10. provider1’s service and consumer1’s service are now integrated through the Data Spine and can start communicating with each other.

- **Asynchronous Communication:**

Prerequisites:

The publisher ‘publisher1’ and subscriber ‘subscriber1’ are both EFPF users. publisher1 and subscriber1 have the necessary access permissions required to register services to the Service Registry.

Let us assume that publisher1’s entity that publishes/intends to publish to the Data Spine Message Bus is ‘fc1’.

Design-time activities:

1. If fc1 is a Factory Connector/Gateway (FCG), publisher1 logs in to the EFPF Marketplace and purchases this Factory Connector fc1 there; else, step 1 is skipped.
2. publisher1 gets a root topic name (for publisher1’s company) ‘p1’.
3. publisher1 logs in to the Factory Connector/Gateway Management Tool (FCGMT) and creates a sub-topic ‘topic1’ under p1 to publish to and gets a key ‘pk’ for publishing to that topic ‘p1/topic1’.

4. publisher1 configures fc1 to publish to Data Spine Message Bus over the topic ‘p1/topic1’ using the key pk.
5. publisher1 registers its service ‘service1’ that consists of this API containing its publication information to the Service Registry.
6. subscriber1 discovers service1’s topic ‘p1/topic1’, decides to subscribe to it and gets the technical metadata for service1 including its API spec from the Service Registry.
7. subscriber1 requests for access permissions to subscribe to p1/topic and gets the key pk for the same.
8. subscriber1 creates an integration flow using the GUI of the Integration Flow Engine to subscribe to p1/topic, perform data transformation and finally to publish the resulting data to Message Bus over the topic ‘s1/topic1’ using the key sk (‘s1’ is subscriber1’s root topic and he/she obtains the same along with the key ‘sk’ following the same procedure as publisher1).
9. subscriber1 registers his/her service with the APIs containing its subscription and publication information to the Service Registry.
10. publisher1’s service and consumer1’s service are now integrated through the Data Spine and can start communicating with each other.

2.4 Dataflow through Data Spine

This section describes the service-to-service communication that happens at run-time through the Data Spine in the form of workflows. In order to enable such a communication, the participant services need to be integrated through the Data Spine first by following the design-time activities enlisted in Section 2.3. Therefore, in a way, the workflows described in this section are a continuation of the activities mentioned in the previous section. This section is further divided into ‘Synchronous Communication Workflow’ and ‘Asynchronous Communication Workflow’ as the workflows differ for both.

Synchronous Communication Workflow (Figure 76):

1. consumer1’s service invokes EP1-C$_P$ endpoint/route of ASG.
2. ASG, acting as a reverse proxy, checks with EFS to ensure that consumer1 has the necessary permissions to access EP1-C$_P$ and to perform the requested operation.
3. ASG, upon receiving a positive reply from EFS, invokes EP1-C exposed by the integration flow.
4. Integration Flow Engine does the necessary request (payload, query parameter, path parameter and/or header) transformation as defined by the integration flow and finally, invokes the external endpoint EP1.
5. Upon receiving the response, the Integration Flow Engine performs payload data transformation as defined by the integration flow and finally, sends the resulting payload as response to ASG.
6. ASG sends the received response to consumer1’s service.

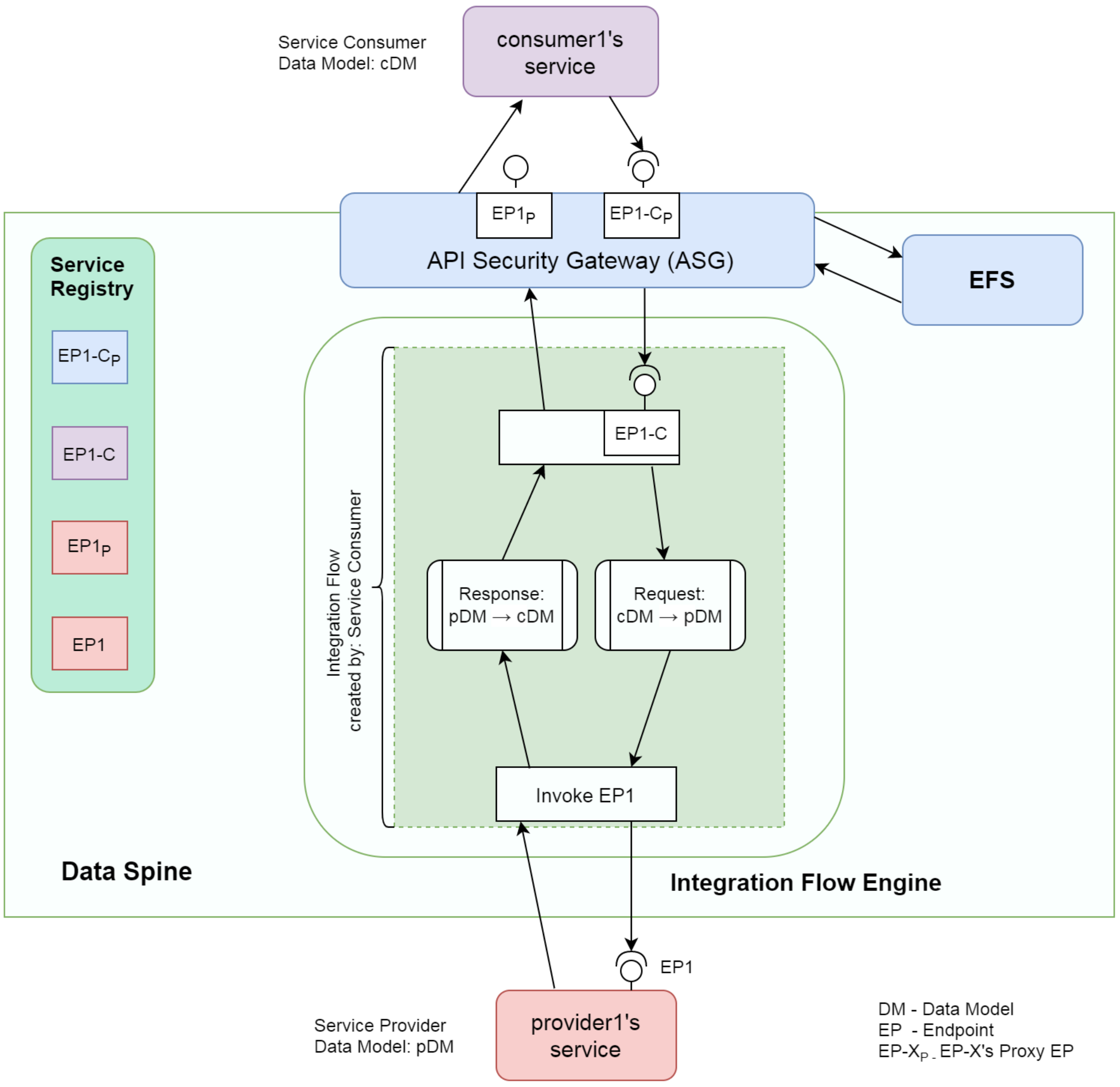


Figure 76: Example of Synchronous Communication Workflow

Asynchronous Communication Workflow (Figure 77):

1. publisher1's service publishes data to the Data Spine Message Bus over topic p1/topic1 with the key pk.
2. The integration flow created by subscriber1 subscribes to this topic p1/topic1 using the key pk, transforms the payload data from publisher1's data model pDM to its own data model sDM and finally, publishes this transformed payload data to the Message Bus over topic s1/topic1 with the key sk.
3. subscriber1's service subscribes to the topic s1/topic offered through the Message Bus using key sk and starts receiving the data.

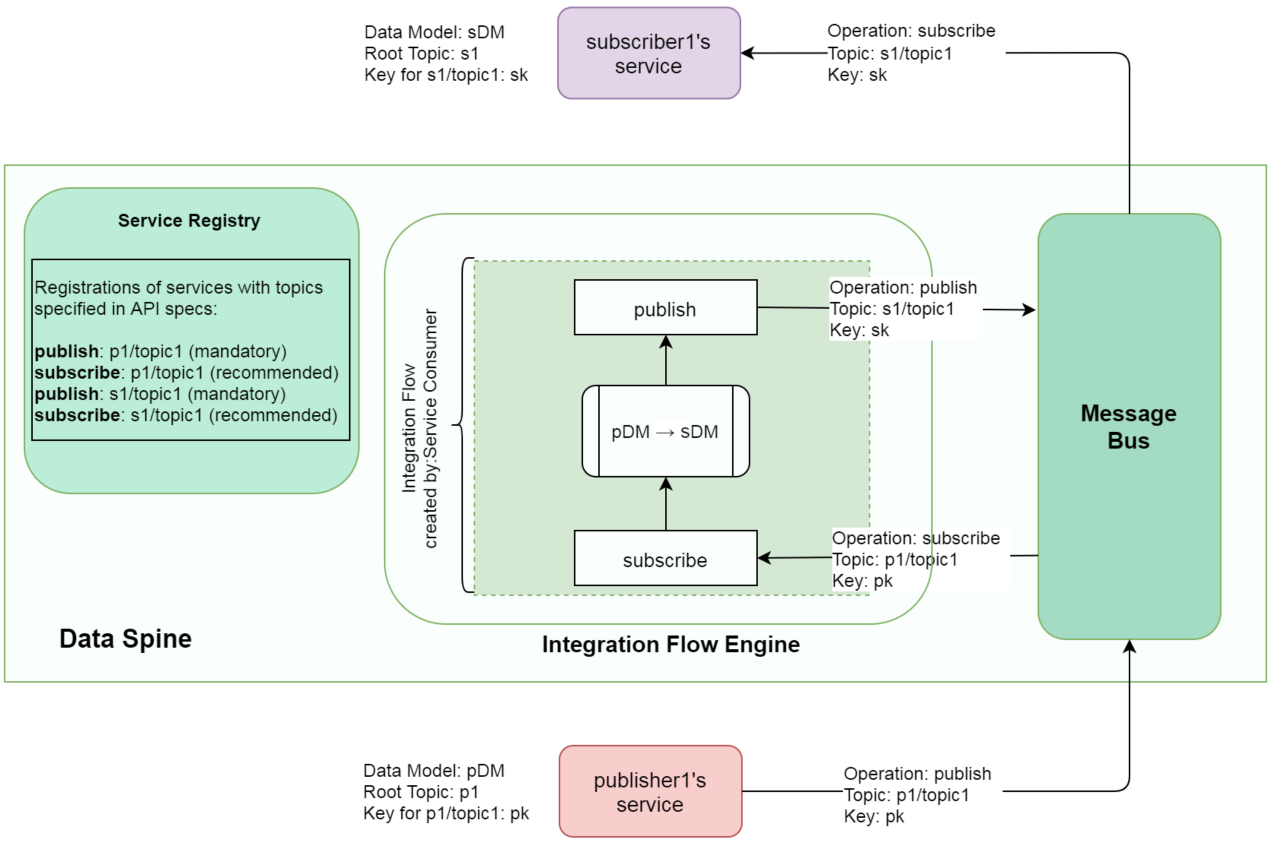


Figure 77: Example of Asynchronous Communication Workflow

3 Interfaces for Tools, Systems and Platforms

This section presents the interfaces for tools, services, systems and platforms in the EFPF ecosystem. It focuses on the APIs of these entities and other interfaces such as GUIs are presented in deliverable D4.1 and other relevant deliverables. This section 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 external entities) to the EFPF project are the building blocks of the EFPF ecosystem. These tools, services, systems and platforms must be able to communicate 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 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 [OAS20] standard is used whereas for specifying APIs of services that follow asynchronous publish-subscribe communication pattern, AsyncAPI Specification [AAS20] standard is used. Both of these industrial-grade standards provide standard vocabulary, structure and formats for specifying the APIs. Moreover, they provide tooling such as editors for writing 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 APIs of tools, services, systems and platforms that together constitute the EFPF ecosystem. The exact API specification documents are included in Annex C. As the exact API specifications for some tools depend upon the definition of concrete pilot scenarios, the preparation of the exact API specifications is work in progress. 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, etc. are also presented in the subsequent sections.

3.2 APIs for Tools, Systems and Platforms

This section presents a brief description of the APIs of tools, services, systems and platforms that together constitute the EFPF ecosystem. The exact API specification documents are included in Annex C.

3.2.1 EFPF Platform

3.2.1.1 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 78 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 78: 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 (Section 2.2.5), creating a new user is using a third-party mail service and the Smart Contracting (3.2.1.6).

Figure 79 shows the data model for creating a user and Figure 80 shows the data model for updating a user.

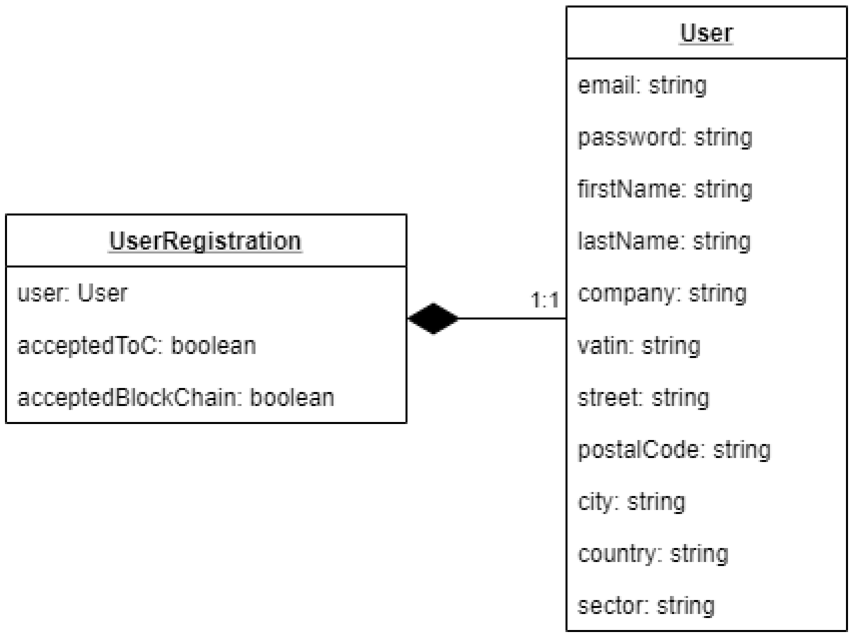


Figure 79: EFPF Portal - User Registration Data Model

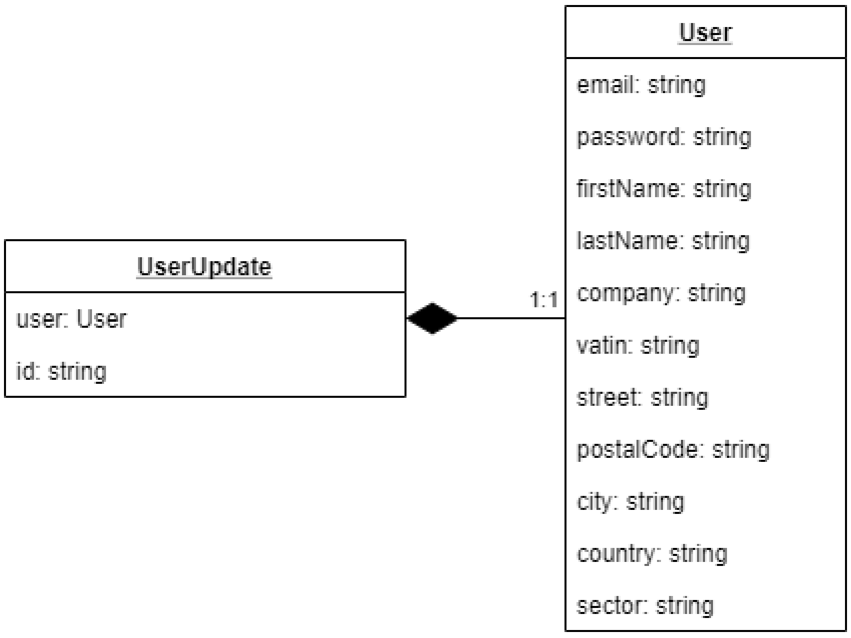


Figure 80: EFPF Portal - User Update Data Model

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[10]

3.2.1.1.2 HTTP REST API for Event Logging

The backend provides an API endpoint for logging events to the Accountancy Service (Section 3.2.1.2).

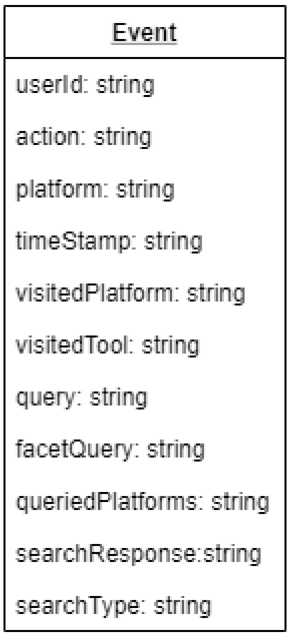


Figure 81: EFPF Portal – Event Data Model

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

10 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 Marketplace

The Marketplace framework consists of an Angular web component, which is communicating with external marketplaces. Future changes will introduce a backend component, which will manage communication with other EFPF components as seen in Figure 11. This component will handle the communication with the Data Spine and external marketplaces, which is currently handled in the web component itself.

As the marketplace component does not provide any REST API endpoints to other components but only consumes endpoints, no API description is being provided at this point. This section will be updated when the backend component has been implemented.

Accountancy Service

Accountacy Service uses Logstash as a data ingestion and server-side data processing pipeline. The data sent to Logstash are forwarded to Elasticsearch for persistence after executing certain ingestion pipelines; and then Kibana dashboards are automatically updated based on the certain fields stored on Elasticsearch. In other words, Accountancy Service uses a basic data model for visualization.
Currently, there is a running instance of Logstash component that is publicly available through a public endpoint so that events from EFPF Portal and external marketplaces can be sent using HTTP POST method. Events are modeled as a JSON message related with the action conforming to the data model described below. This will be enough for the Accountancy Service to capture the data and update the dashboards. In the future releases of the Accountancy Service, its integration with the Service Registry will be realised. In this way, Logstash endpoint of the Accountancy Service will discoverable by external marketplaces through the Service Registry and new marketplaces will be integrated easily. 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.

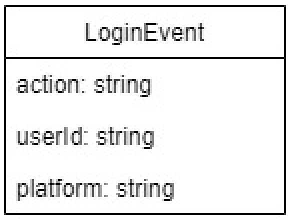


Figure 82. Accountancy Service Login Event Data Model

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.

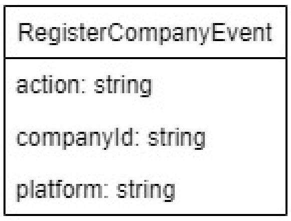


Figure 83. Accountancy Service Company Registration Event Data Model

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 “EFPF” value as the name of the platform

REGISTER_USER

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

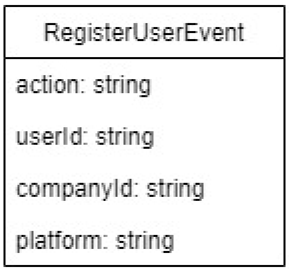


Figure 84. Accountancy Service User Registration Event Data Model

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 “EFPF” value as the name of the platform

PLATFORM_VISIT

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

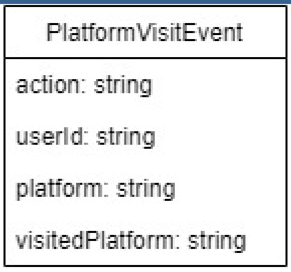


Figure 85. Accountancy Service Platform Visit Event Data Model

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 “EFPF” 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.

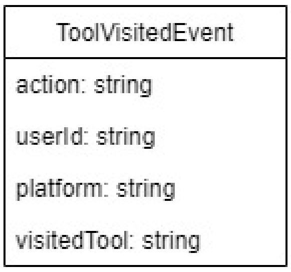


Figure 86. Accountancy Service Tool/Service Visit Data Model

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.

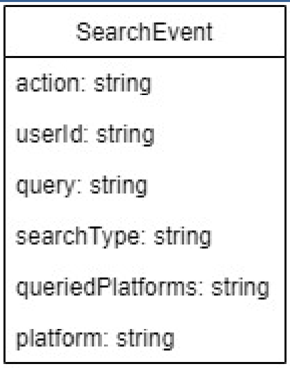


Figure 87. Accountancy Service Search Event Data Model

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.

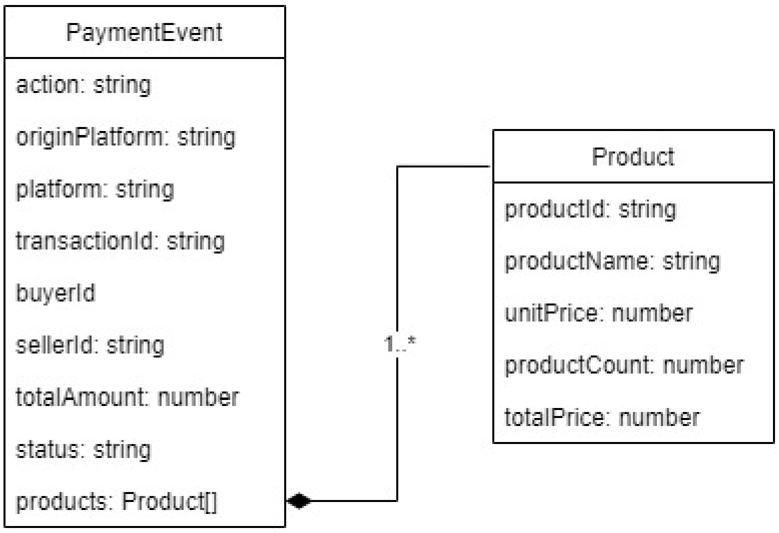


Figure 88. Accountancy Service Payment Event Data Model

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)

3.2.1.3 Matchmaking

Matchmaking Service

Currently the matchmaking service only supports federated search API and ontology indexing APIs.

- Federated Search API: Provides API endpoints to search items, parties and categories (Swagger API Spec: https://efpf-security-portal.salzburgresearch.at/api/index/swagger-ui.html#/index-controller)
- Ontology Controller API: Provides API endpoints to add/delete ontologies (e.g.: eclass, furniture ontology etc) in the federated index. (Swagger API Spec: https://efpf-security-portal.salzburgresearch.at/api/index/swagger-ui.html#/ontology-controller)

Matchmaker for Online Bidding Process

This Matchmaking API, has migrated from COMPOSITION project and been integrated to the EFPF common platform. The details of this API have been included in Section 3.2.2.1.

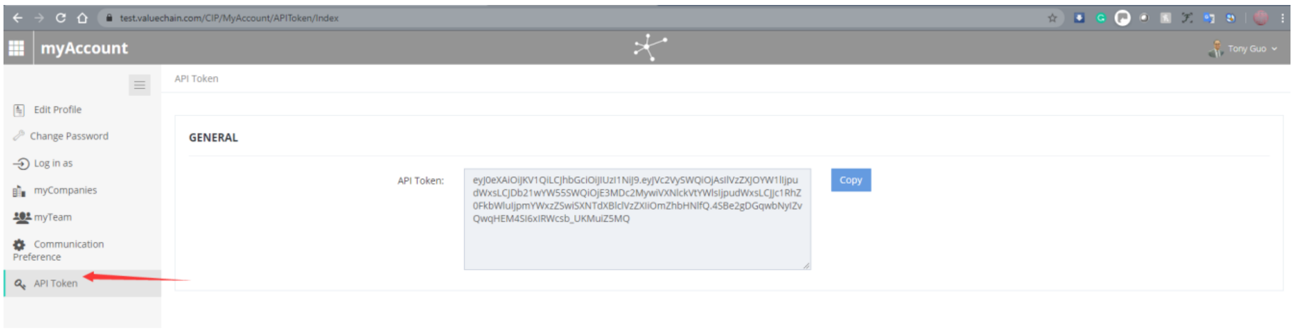
3.2.1.4 Governance & Trust

In Governance and Trust component, the APIs to add/delete/modify EFPF policies are currently under development.

3.2.1.5 Business & Network Intelligence

At present, the only API developed and being used under this section is for iQluster platform. This API has been developed to enable data share between iQluster platform and the EFPF matchmaking indexing workflow.

The API structure is made up of multiple API calls that return specific information. Firstly, the user needs to retrieve the appropriate API token for iQluster's production environment. This can be found under 'My Account' section of the user profile as shown in Figure 89.



API Token: eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJVc2VySWQiOjAsIlVzZXJOYW1lIjpudWxsLCJDb21wYW55SWQiOjE3MDc2MywiVXNlckVtYWlsIjpudWxsLCJJc1RhZ0FkbWluIjpmYWxzZSwiSXNTdXBlclVzZXIiOmZhbHNlfQ.4SBe2gDGqwbNyIZvQwqHEM4SI6xIRWcsb_UKMuiZ5MQ

Figure 89: API Token Generator for iQluster Platform

The various API calls with their description are provided below:

1. **Get all companies**: This is an API that allows you to retrieve all companies you have permission to retrieve. Using the company number returned in the API, you can retrieve more information about this company using other API calls.
2. **Get basic company data**: This API allows you to retrieve all company information that iQluster holds for the company ids (retrieved from the first API above) that are called with this API. Some general fields are mentioned below as examples:
   a. Company description (from company profile)
   b. Company status (active, dormant etc.)
   c. Incorporation date
   d. Website
   e. Social media handles…
3. **Get company address**: This API allows you to retrieve all the addresses stored in iQluster platform for the company ids used in this call. iQluster can hold multiple international locations for companies.
4. **Get company capability**: iQluster builds capability structures to facilitate classification of member companies. This API allows you to retrieve the capabilities attributed to the company as well as the parent and child node for that capability field so that it provides more context.

Following successful testing of this implementation. The partners have agreed to expand on these API calls to include even more information including fields such as (supply chain connections, financial information etc.)

3.2.1.6 Smart Contracting

The objective of the EFPF Blockchain and Smart Contracting Service is to allow applications built on EFPF to leverage distributed ledger technology to provide distributed trust, transparency and decentralized business models. A set of services will allow EFPF stakeholders to verify the authenticity, origin and standards of data and services. The chosen distributed ledger implementation is Hyperledger Sawtooth, which has a number of desirable characteristics that address the design concerns for the EFPF Blockchain and Smart Contracting Service. It can support multiple domains due to separation between the application level with specific transaction families, and the core system managing the distributed ledger cryptography, event system, parallel transactions and distributed data synchronization. A transaction family is a set of operations allowed on the ledger, from simple transaction rules to programmable smart contracts. It can provide a private blockchain for a business network with consensus algorithms that require less resources, and has permissioning features that enable a federation level solution where no single entity will own the distributed ledger but all can have varying levels of access, ensuring data security. Hyperledger Sawtooth allows development of smart contracts in a number of languages and has compatibility with Ethereum contracts through the Seth transaction family.

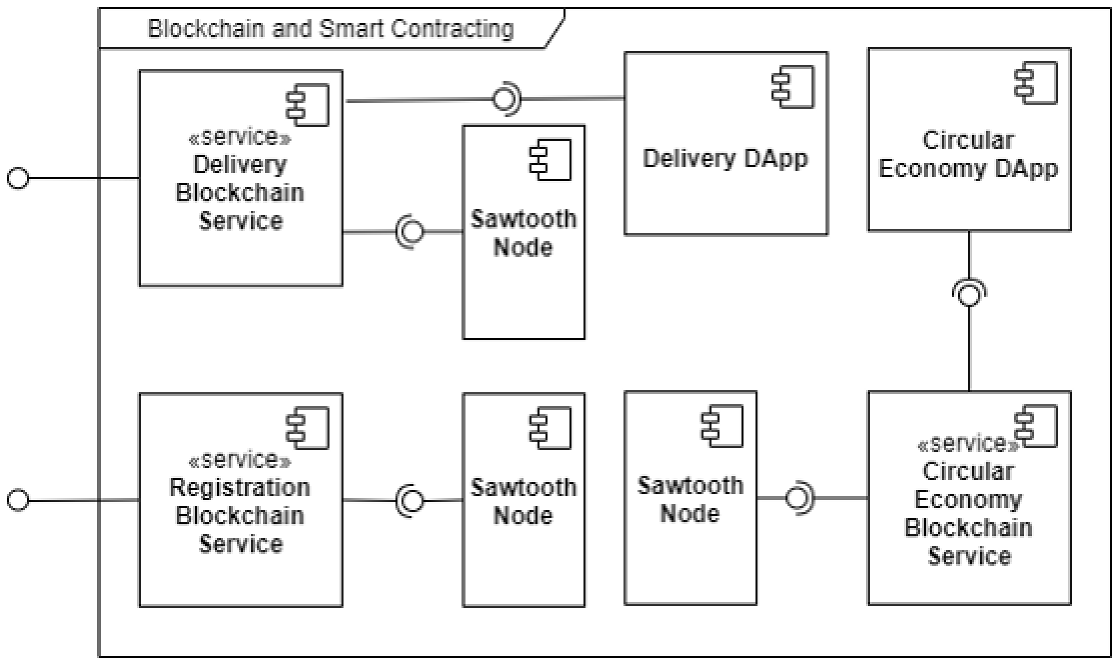

Blockchain and Smart Contracting
«service» Delivery Blockchain Service
Sawtooth Node
Delivery DApp
Circular Economy DApp
«service» Registration Blockchain Service
Sawtooth Node
Sawtooth Node
«service» Circular Economy Blockchain Service

Figure 90: Blockchain and Smart Contracting

On this foundation, EFPF has built applications for delivery, supply chain visibility, circular economy and supporting DApps using real-world evidence together with distributed ledgers as proof-of-concept and re-usable examples.

The sawtooth network is the foundation of the Blockchain and Smart Contracting Service, with different application running on top. It will be distributed among EFPF stakeholders, with no central authority, but with the possibility to control permissions and ownership of proprietary data.

3.2.1.7 Data Analytics

In the EFPF platform the Data Analytic solutions are offered as standalone Cloud-based services, which make use of the EFPF Data Spine functionalities for data exchange, data interoperability and security features.

The integration of the analytic solutions only concerns the UI level where the interfaces of the analytic solutions are made available through the EFPF Portal.

3.2.1.8 Workflow & Business Process

In the EFPF platform the WASP solution offered as standalone Cloud-based service, which make use of the EFPF Data Spine functionalities for data exchange, data interoperability and security features.

The integration of the analytic solutions only concerns the UI level where the interfaces of the WASP solution is made available through the EFPF Portal.

3.2.1.9 Secure Data Storage

The primary interaction with Secure Data Store Solution is the resource access APIs, which allows specific time ranges to be addressed from a full resource range, enforcing a minimization of data exposure.

To support the use of the User Managed Access (UMA) standard for fine-grained management of authorization access, each resource needs to be available as a URL. Secure Data Store Solution needs to address the primary level of resource management as the overall set of timeseries data collected from sensors. As such, this is used to form the base URL, which is used by data owners to manage the resource collection. Resource owners can use a REST API to manage data sensor configuration:

- Human readable description
- Selection criteria within broker environment (i.e. path)
- Data semantics, i.e. unit specification
- Pseudonymization policies

Secure Data Store Solution is in early stages of development, and the specific data model API access is still changing. Configuration of the sensor specifications occurs via a REST interface supporting common (Create, Read, Update, Delete) CRUD operations with a JSON version of the data model. A detailed description of the stable data model will be provided in the next deliverable.

The primary resource is broken down into time regions by specifying a start and end time for the range, under the URL of the base resource. This selection of the primary resource can then be addressed externally, and authorization managed. The contents of the time range selected is expected to be retrieved by analysis tools, looking to apply computer learning techniques on past knowledge. As such, the API provides analysis tools the ability to read sensor data. Basic timeseries queries are supported, but the primary mechanism to retrieve data is paging.

Data access is guarded with authorization according to the User Managed Access protocol standard, which build on top of authentication and identity management OAuth and OpenID Connect. Data is further guarded by pseudonymization techniques. The exact configuration of these controls occurs within the Secure Data Store Solution.

3.2.1.10 Smart Factory Tools & Services

3.2.1.10.1 Industreweb Global

Industreweb Global is essentially a web application where Industreweb admin tools are provided, along with a few management services (workflow, security etc.).
Industreweb Global contains an API Library

REST Endpoint	HTTP Method	Description

/GetAllCollects	GET	Gets all collect configurations from the database
/GetAllConnectors	GET	Gets all the connector configurations from the database
/GetAllDisplays	GET	Gets all display configurations from the database. These are shop floor nodes that display data.
/GetAllConnectorTags	GET	Gets all the connector tags from the database

Figure 91: Industreweb Global API Endpoints

GetAllCollects returns a list of Collect objects, this consists of:

CollectID: A unique ID assigned to the Collect configuration
CollectGuid: A unique Guid assigned to the Collect configuration
Name: Name of the Collect configuration
Description: Description of the Collect configuration
CreatedOn: A string showing who created the Collect in the database
CreatedBy: A date-time string showing when the Collect was created in the database
ModifiedBy: A string showing who most recently modified the Collect (if modified from original insert)
ModifiedOn: A date-time string showing when the Collect was most recently modified in the database (if modified from original insert)

GetAllConnectors returns a list of Connector objects, this consists of:

ConnectorID: A unique ID assigned to the Connector configuration
ConnectorGuid: A unique Guid assigned to the Connector configuration
ConnectorType: ConnectorType object the Connector belongs to
Collect: Collect object the Connector belongs to
CreatedOn: A string showing who created the Connector in the database
CreatedBy: A date-time string showing when the Connector was created in the database
ModifiedBy: A string showing who most recently modified the Connector (if modified from original insert)
ModifiedOn: A date-time string showing when the Connector was most recently modified in the database (if modified from original insert)
DeployedBy: A string showing who most recently deployed the Connector (if deployed)
DeployedOn: A date-time string showing when the Connector was most recently deployed (if deployed)

GetAllDisplays returns a list of Display objects, this consists of:

DisplayID: A unique ID assigned to the Display configuration
DisplayGuid: A unique Guid assigned to the Display configuration
Name: Name of the Display configuration
Description: Description of the Display configuration
CreatedOn: A string showing who created the Display in the database
CreatedBy: A date-time string showing when the Display was created in the database

ModifiedBy: A string showing who most recently modified the Display (if modified from original insert)
ModifiedOn: A date-time string showing when the Display was most recently modified in the database (if modified from original insert)
Screen: Screen object that the Display belongs to

GetAllConnectorTags returns a list of ConnectorTag objects, this consists of:
DisplayID: A unique ID assigned to the ConnectorTag configuration
DisplayGuid: A unique Guid assigned to the ConnectorTag configuration
Connector: Connector object which the ConnectorTag belongs to
Name: Name of the ConnectorTag configuration
Description: Description of the ConnectorTag configuration
CreatedOn: A string showing who created the ConnectorTag in the database
CreatedBy: A date-time string showing when the ConnectorTag was created in the database
ModifiedBy: A string showing who most recently modified the ConnectorTag (if modified from original insert)
ModifiedOn: A date-time string showing when the ConnectorTag was most recently modified in the database (if modified from original insert)

3.2.1.10.2 Risk Tool

The Risk Tool provides an interface for users to design and use risk analysis "recipes". *Recipes* transform an input JSON object to another format. This can, for example, be used to compute a specific type of risk. Recipes can be used by configuring *workflows*, which can either be run a single time for testing purposes, or can be run continuously by subscribing to specific topics on the Data Spine.

The current version of the REST API is v1. Its root is found at "`/api/v1`". The following REST API endpoints are currently available:

REST Endpoint	HTTP Method	Description
/recipes/	POST	Add a recipe
/recipes/	GET	List all recipes
/recipes/{rid}	GET	Retrieve single recipe
/recipes/{rid}	DELETE	Delete a recipe
/recipes/{rid}	PATCH	Update a recipe
/workflows/	POST	Add a workflow
/workflows/	GET	List all workflows
/workflows/{wid}	GET	Retrieve single workflow
/workflows/{wid}	DELETE	Delete a workflow
/workflows/{wid}	PATCH	Update a workflow

Figure 92: Risk Tool REST API Endpoints

The REST API will be expanded in future versions to include endpoints to run workflows either once or via subscriptions.

3.2.1.10.3 Catalogue Service

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 the below table:

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

Figure 93: Product Category Management API Endpoints

Catalog Management endpoints are described in the following table:

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

Figure 94: Catalog Management API Endpoints

3.2.1.10.4 Symphony Event Reactor

The Symphony Event Reactor gives the ability to trigger actions and alarms through its Event Manager/ Alarm Manager in response to different kinds of event types. As it is shown in its UML component diagram it has Connector for receiving the triggering message for alarms and events from message bus services. Symphony Event Reactor accepts only specific data model as input. The data model schema is:

```
{
  "message_type": <type>,
  "timestamp": <epoch_ts>,
  "payload": {
    "state": "<state>",
    "channel": <channel_id>,
    "description": "<risk_description",
    "severity": "<severity_level>",
    "timestamp": "<epoch_ts>"
  }
}
```

The attributes are:

- message type: type of the message, it is fixed on “alarm”
- timestamp: time stamp in EPOCH format
- state: state of the alarm or event
- channel: channel
- description: description of the risk
- severity: severity level of the alarm
- timestamp: time stamp in EPOCH format

3.2.1.10.5 Symphony Data Storage

Symphony Data Storage is a highly scalable and high-performance data storage which is designed to handle large amount of AMQP/MQTT data. As it is shown in its UML component diagram, it has Connector for ingesting data from message bus services and REST API for retrieving the data.

- AMQP Connector (Input): 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.
- REST API (Output): for querying the Symphony Data Storage:

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 95: Symphony Data Storage REST API Endpoints

3.2.1.11 Factory Connectivity

As described in Section 1 EFPF supports a numbers of factory connectivity solutions. This section will describe the API used by all factory connectivity solution in EFPF. However, the same API can be used by external factory connectivity solutions to integrate and start publishing messages to the EFPF platform.

The factory connectivity solutions API is described as a specification of the MQTT topic namespace. Since Pub/Sub communication is used in EFPF to exchange factory data with the EFPF services, the API of the factory connectivity solutions is based on Pub/Sub communication and describes the topic namespace and payload. Currently MQTT is main standard used in EFPF to exchange factory data; therefore, the factory connectivity API specification is based on MQTT. Future work is to include other protocols such as AMQP and analyse if the same namespace can be applied

3.2.1.11.1 MQTT Topic Namespace and Payload

The MQTT topic namespace is using the Sparkplug™ Specification of the MQTT topic (not the payload definition). Thus, all the factory connectivity solutions will use the following structure:

```
namespace/group_id/message_type/edge_node_id/[device_id]
```

Where the ‘namespace’ is the root element and is created when an EFPF user purchase an IoT gateway or a factory connector. The ‘group_id’ is a logical grouping of the factory connectors and IoT gateway, referred to in Sparkplug Specification as Edge of Network (EoN) nodes, e.g. factory connectors and IoT gateways that can be grouped as Assembly line A or Factory B. The ‘message_type’ elements specifies the type of message being sent and are follow:

- NBIRTH: Birth certificate for the factory connector/IoT gateway (MQTT EoN nodes)
- NDEATH: Death certificate the factory connector/IoT gateway (MQTT EoN nodes)
- DBIRTH: Birth certificate for devices connected to the factory connector/IoT gateway
- DDEATH: Death certificate for devices connected to the factory connector/IoT gateway
- NDATA: Factory connector/IoT gateway data message

- DDATA: Devices data message connected to the factory connector/IoT gateway
- NCMD: Factory connector/IoT gateway command message.
- DCMD: Devices command message connected to the factory connector/IoT gateway
- STATE: Critical application state message.

The 'edge_node_id' element uniquely identify the MQTT Edge of Network node which is the EFPF factory connector/IoT gateway. If there are devices behind the factory connector/IoT gateway, then would be identified using the 'device_id'.

The payload the MQTT message follows the OGC Web Enablement standard suite mainly focusing on OGC SensorThings API for Industrial IoT data. When the data shared is focused on machine and production data OPC UA Part 100: Device Information Model. More details about the data models used please refer to section 4.

3.2.1.11.2 Factory Connector Gateway Management Tool

The FCG API is part of the FCGMT and it provides a collection of services to interact with a repository of producers/consumers IoT devices that publish/subscribe to data from the Message Broker, as well as associated topics.

Figure 96 includes the services provided by the HTTP REST API of the FCGMT. This is mainly a collection of CRUD operations (create, read, update and delete) of *Device* objects provided by RESTful services.

REST Endpoint	HTTP Method	Description
/	GET	Retrieves all devices
/{device}	POST	Creates new 'Device' object with an ID as primary key
/{device}	GET	Retrieves a 'Device' object
/{id}	PUT	Updates the existing 'Device' based on the submitted ID
/{id}	DELETE	Deletes the 'Device'

Figure 96: FCG HTTP REST API

Internally, the API manages the models *Device* and *Topic*, which are related in order to represent what topics are produced or consumed by the devices. The relationships between devices and topics are defined directly updating (PUT) the information of a given device. This information includes to what topics the device publishes data as well to what topics the device is subscribed to.

Besides this, an API for administration tasks in the Message Broker has been also implemented. This API enables to perform some operations such as creating exchanges, queues and routing key bindings on the RabbitMQ instance, as well as purge and delete operations.

Figure 97 lists the services provided by this Message Broker API which can be also consumed by any other component in a RESTful manner.

REST Endpoint	HTTP Method	Description

`/createBinding?exchangeName=`
`?queueName=?binding`
	POST	Creates a new exchange (or uses default if not indicated), and a new queue, and performs a binding between the queue and the binding parameter

`/sender?routingkey=`
`?exchangeName=?queueName=`
`?binding`
`Body message`
	PUT	Published data (message) to the specified exchange, queue and routing key (topic). If the exchange is not indicated, the default one is used. The specified resources are created in the RabbitMQ instance if they are have not been created yet.

`/receiver?queueName`	GET	Retrieves one-by-one messages from a queue
`/admin/purgeQueue?queueName`	PUT	Removes all the messages of a queue
`/admin/deleteQueue?queueName`	DELETE	Deletes a queue
`/admin/deleteExchange` `?exchangeName`	DELETE	Deletes an exchange

Figure 97: Message Broker (RabbitMQ) HTTP REST API

As of the dat.
e of this document, the definition is still open to additions to improve the overall functionality of the FCGMT component, so some subtle changes may occur in the managed models as well as the endpoints provided by the included APIs.

3.2.2 Base Platforms

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. Table 1 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.

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 98: Matchmaker RESTful services catalog

Detailed descriptions of the matchmaking mechanisms are available on D5.1. The bidding process interfaces are documented on D5.3.

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
- ICT-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. Following table 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
Identity Service	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.	https://efpf-nimble.salzburgresearch.at/api/identity/swagger-ui.html
Catalog Service	Manage Catalogs and catalogue-lines (products)	https://efpf-nimble.salzburgresearch.at/api/catalog/swagger-ui.html#/

	for companies on the platform. All the related binary-content are persisted in the catalog-persistence layer as well.	
Business Process Service	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. 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.	https://efpf-nimble.salzburgresearch.at/api/business-process/swagger-ui.html#/
Indexing Service	Service works as the middle layer to index products, companies also properties and classes of the used ontologies in Apache Solr.	https://efpf-nimble.salzburgresearch.at/api/index/swagger-ui.html#/
Trust Service	Microservice which calculates and manages the trust profiles of companies. Have a predefined set of trust attributes which can be extended according to requirement.	https://efpf-nimble.salzburgresearch.at/api/trust/swagger-ui.html#/
Data-Aggregation Service	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	https://efpf-nimble.salzburgresearch.at/api/data-aggregation/swagger-ui.html#/

	based on the role that clients try to retrieve data.	

Figure 99: NIMBLE Services

3.2.2.3 DIGICOR

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 and a MarketplaceAPI.

REST Endpoint (smecluster.com/api/DirectoryWebService)	**HTTP Method**	**Description**
/GetCompany?companyID={id}	GET	Gets a company based upon the ID provided in the URL
/GetCategory?categoryID={id}	GET	Gets a category based upon the ID provided in the URL
/GetAllCompanies	GET	Gets a simplified version of all Companies contained in the database
/GetAllCompaniesFull	GET	Gets a fully detailed version of all Companies contained in the database
/GetAllCategories	GET	Gets all Categories contained in the database
/GetAllCapabilities	Get	Gets all Capabilities contained in the database

Figure 100: 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 (smecluster.com/api/CatalogUtilsWebService)	**HTTP Method**	**Description**
/GetAllProducts	GET	Gets all Products contained in the database
/Search?searchTerm={searchTerm}	GET	Gets a Product/Products based upon the search term provided; the search term must contain a Product name e.g. Industreweb

Figure 101: 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 102. 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 102 shows the interaction between services within the SMECluster platform, and those from the DIGICOR platform.

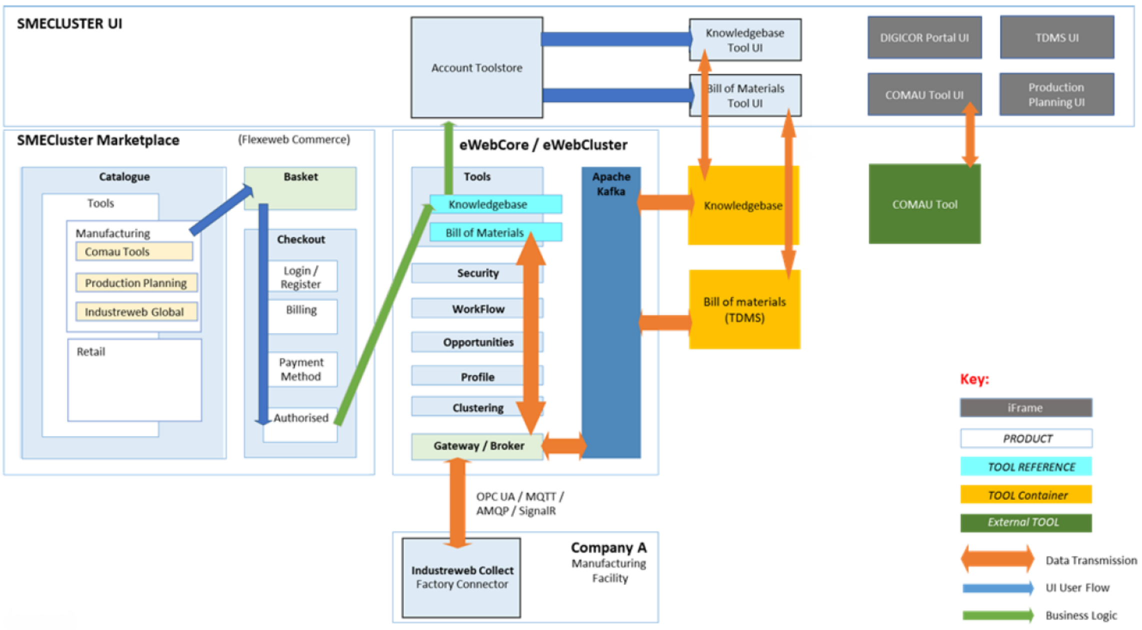


Figure 102: SMECluster Component Interaction

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 103 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.

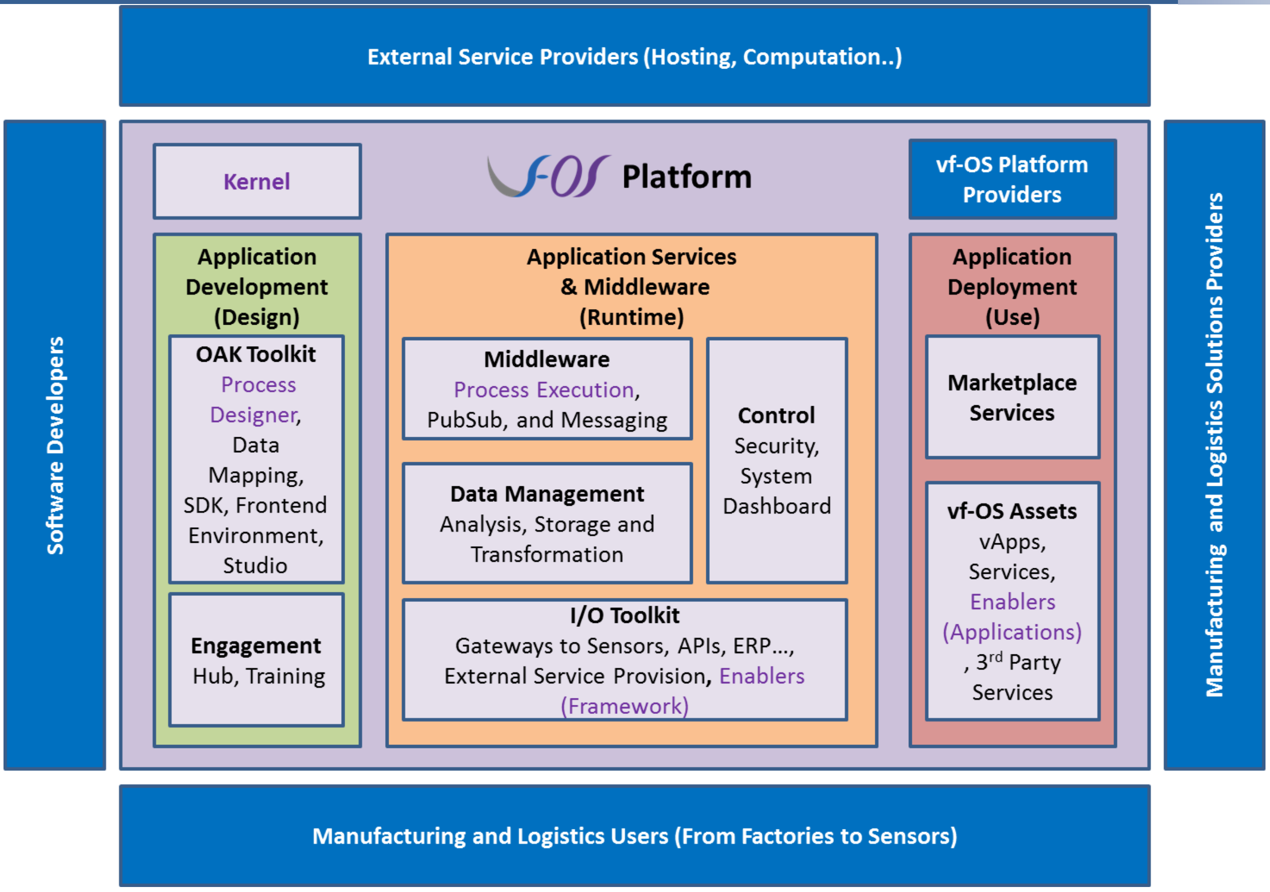

External Service Providers (Hosting, Computation..)
Software Developers
Manufacturing and Logistics Solutions Providers
Kernel
vf-OS Platform
vf-OS Platform Providers
Application Development (Design)
OAK Toolkit
Process Designer, Data Mapping, SDK, Frontend Environment, Studio
Engagement Hub, Training
Application Services & Middleware (Runtime)
Middleware
Process Execution, PubSub, and Messaging
Data Management
Analysis, Storage and Transformation
Control
Security, System Dashboard
I/O Toolkit
Gateways to Sensors, APIs, ERP..., External Service Provision, Enablers (Framework)
Application Deployment (Use)
Marketplace Services
vf-OS Assets
vApps, Services, Enablers (Applications), 3rd Party Services
Manufacturing and Logistics Users (From Factories to Sensors)

Figure 103: 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 - s can be seen in Figure 104.

Service Name	Description	Documentation URL
vf-OS Platform	Overall environment, platform and kernel	• vf-Platform: https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-platform-environment-vf-p
Application Development (vf-OAK Toolkit)	Tools for developers to develop vf-OS assets, such as vApps, Enablers, etc.	• vf-OAK Toolkit: https://engagementhub.caixamagica.pt/virtual-factory-operating-system/open-applications-development-toolkit-vf-oak
Application Services & Middleware	Set of enablers that are able to receive stimulus and actuate on the factory elements or that virtualise the factory	• Process Designer: https://engagementhub.caixamagica.pt/virtual-factory-operating-system/middleware/process-enabler---runtime • Messaging and Publish/Subscribe: https://engagementhub.caixamagica.pt/virtual-factory-operating-system/middleware/publish-subscribe

		• Data Storage: https://engagementhub.caixamagica.pt/virtual-factory-operating-system/data-management/data-storage • Data Analytics: https://engagementhub.caixamagica.pt/virtual-factory-operating-system/data-management/data-analytics • Enablers Framework (EF): https://engagementhub.caixamagica.pt/virtual-factory-operating-system/io-toolkit/enablers-framework • API Connectors: https://engagementhub.caixamagica.pt/virtual-factory-operating-system/io-toolkit/api-connectors • External Service Provision (ESP): https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-platform-environment-vf-p/-/tree/master/esp • Identity Management (IDM): https://engagementhub.caixamagica.pt/virtual-factory-operating-system/control/security/identity-management • Authorisation Policy Decision Point (PDP): https://engagementhub.caixamagica.pt/virtual-factory-operating-system/control/security/authorisation-policy-decision-point • System Dashboard: https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-platform-environment-vf-p/-/tree/master/systemDashboard
Application Deployment Services	Set of components that will be taken into consideration when the vf-OS environment is going to be in use	• Marketplace Services: vf-Store: https://engagementhub.caixamagica.pt/virtual-factory-operating-system/marketplace-vf-store • FIWARE Generic Enablers: https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-assets/fiware-generic-enablers

		• Manufacturing Enablers: https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-assets/fiware-manufacturing-enablers • vf-OS Enablers: https://engagementhub.caixamagica.pt/virtual-factory-operating-system/vf-os-assets/vf-os-enablers

Figure 104: vf-OS services

3.3 API Management

API management is the process that is concerned with creating, publishing, monitoring and securing the APIs in an ecosystem. EFPF project offers various tools for effectively managing the APIs of services in the EFPF ecosystem.

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 made retrievable. The Service Registry (Sections 2.1.3 and 2.2.3) 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 creation of a simple GUI for easily browsing through, viewing and searching for API specs of services is in progress. It is planned to have this GUI integrated into the EFPF Portal.

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 isn't updated within the set ttl, it is removed from the Service Registry. The minimum value for ttl can be 1 second and the maximum value will be decided soon. 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 3.4.2.

A new monitoring tool is currently being developed in EFPF that will have the capabilities to monitor APIs for ensuring their availability, for collecting usage stats such as API calls per hour, for example, and performance factors such as latency, uptime, request processing time, etc. and for visualising these statistics and for generating alerts. More details about this tool can be found in deliverable D7.1.

Securing Access to API Endpoints

The access to API endpoints in EFPF ecosystem is secured with the access policies defined in the EFS and it is enforced with the help of the API Security Gateway component of the Data Spine.

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.

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 access to source code, documentation, etc. Thus, in a federated 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 upgrades of APIs becomes necessary and inevitable. Therefore, the federated 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 is in progress and can be found in D7.1.

3.4.2 Interface Contracts Management Tool

The Interface Contract Management Tool (ICMT) is a custom component under development which will be used 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 interfaces are detected than the tool proceeds to warn the users about them to allow developers making use of those interface to take appropriate actions.

The updates are delivered through the Message Bus, however developers can check for the latest updates for a known service through a set of HTTP API.

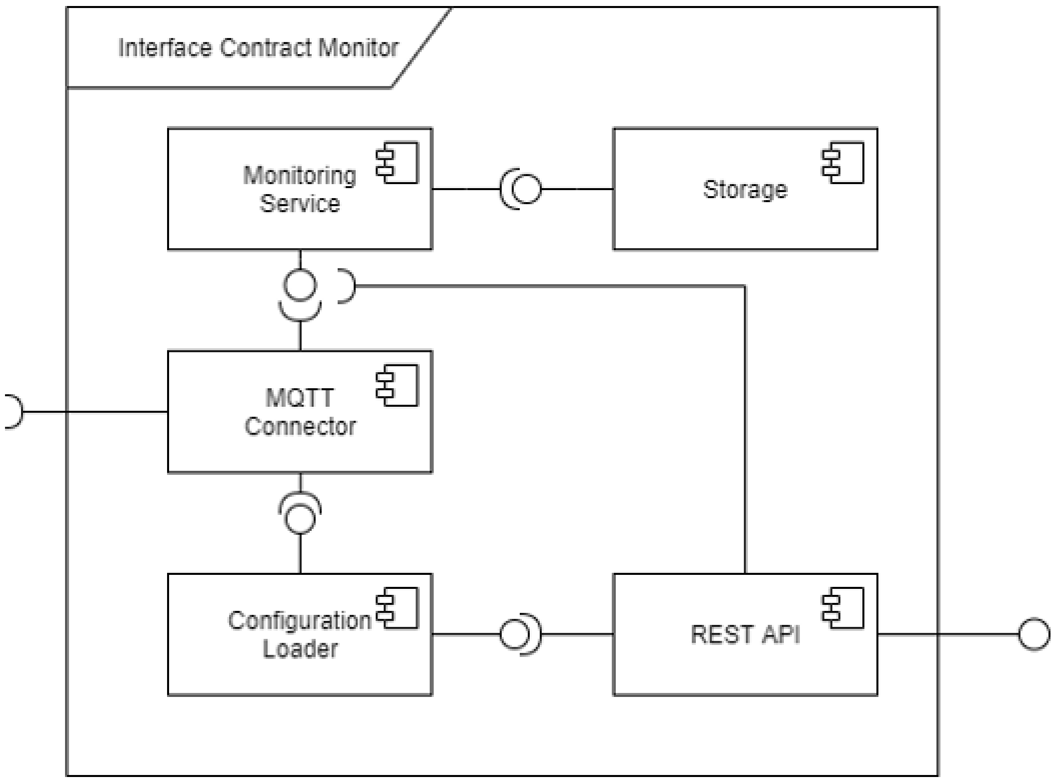


Figure 105: Architecture of Interface Contracts Management Tool

4 Data Model Interoperability Layer

4.1 Introduction

The objective of the data model interoperability layer is supporting information exchange and business processes that spread across two or more of the existing EFPF platforms. 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.

4.1.1 Methodology for the Data Model Interoperability

Two approaches have been considered for dealing with the data model interoperability issues, bottom-up and top-down.

The top-down method starts from the general and moves to the specific. Using the top-down method requires having a detailed understanding of the system. That, in this case, would mean finding, right at the beginning of the project, a set of standard data models which would then be used for the project according to their specifications. Then, while the project develops, according to the feedback from the pilots, the data models can be maintained or swapped if they are deemed as unfit for the task. In some cases, top-down design can lead to unsatisfactory results because important and necessary features for the system can be missed.

A first approach with a top-down method has been attempted by performing a general survey about all the data models used from all the tools and services that are part of the EFPF platform. More than the data models used this survey also collected information about data rate, size and format used to encapsulate the data. By looking at the results of this survey alone it has not been possible to make a decision since the data models discovered were too many and the majority were proprietary. The huge number of proprietary data models can be explained since most of the tools and services come from different previous projects which had not interoperability in mind since they were not designed as open platforms. The unsatisfactory results of the survey made necessary to attempt a bottom-up approach to have a clearer picture about the collected data.

The bottom-up approach begins with the specific details and moves up to the general. To begin a bottom-up design, the pilots representing the use cases will be investigated and the data models used in the tools included in the pilots will be checked. Then, the data models which will find the most applications or the more widespread adoption will be kept and picked as the standards for the platform. The disadvantage of this kind of approach is that if the pilots do not represent well enough all the use cases, one may end up with choosing data models which may exclude some scenarios.

After examining both the pros and cons of both the approaches it has been chosen to proceed with a bottom-up approach due to the complexity of the data collected while attempting a top-down approach. This collected data however did not go to waste since it proved very useful while analysing the pilot scenarios as explained in the following sections.

The steps which have been taken in the aforementioned bottom-up approach are:

- Investigating the standard data models which are suited to the different EFPF use cases.
- Exploring the tools and services included in the EFPF pilots to see whether the tools used the same data models found in the previous step.
- Picking the data models used in the pilot projects as well as found in the prior investigation as the standard data models.

These are instead the steps ahead:

- Creation of policies to promote the adoption of the standard data models in the EFPF ecosystem.
- Development of tools for integrating the standard data models with any proprietary data model used.
- Validation of work through an analysis of the adoption of the standard data models during the open call phase.

Choosing to use a set of standard data models on the Data Spine can be interpreted as using a one-to-one approach when dealing with data model incompatibilities, rather than using a one-to-many approach in which each tool / service has to directly deal with all the data models used from the other tools / services.

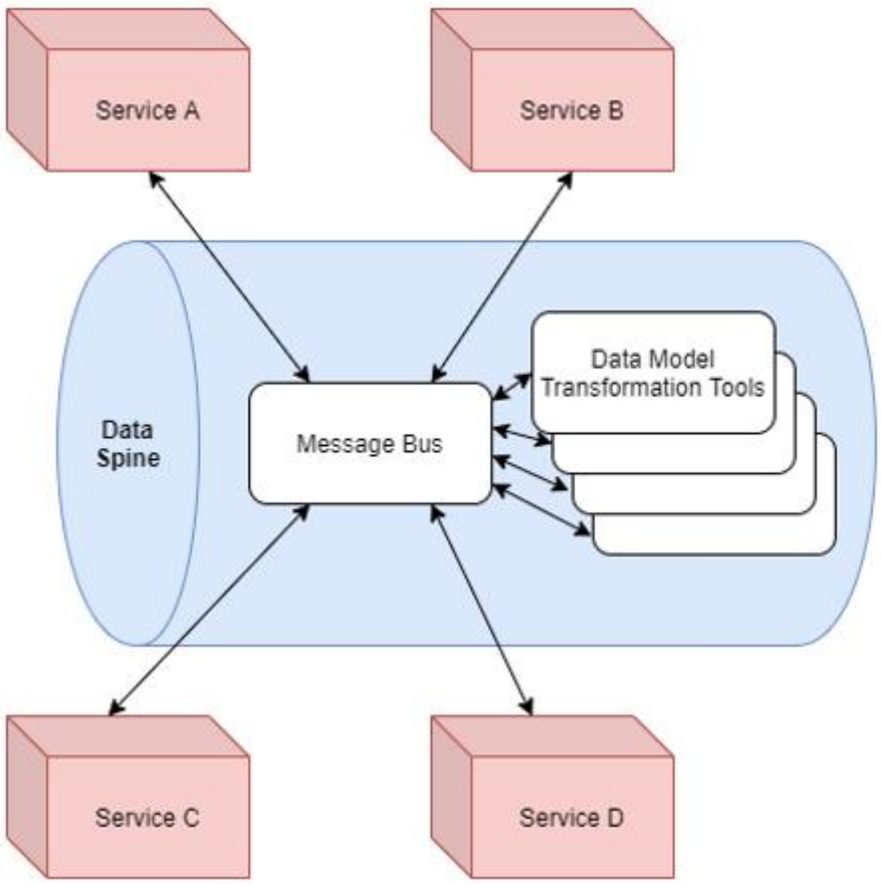


Figure 106: Use of Standard Data Models through a Data Model Transformation Tool

An example is provided in the scenario shown in Figure 106. In this scenario there are 4 linked services all using different custom data models. The interoperability for these services is provided from the Data Model Transformation Tools which receive the data encoded with the respective custom data models from the Message Bus, translate it to the EFPF chosen standard data model and then push it back to the Message Bus. Then this data is captured from another Data Model Transformation Tool which encodes it to the destination service’s custom data model, pushed back to the Message Bus and eventually consumed from the service. The advantage of this approach is that if a change is made to one of the custom data models of one of the services, only the Data Model Transformation Tool in charge of dealing with translating that data model from / to the EFPF standard data model will need to be updated. The change will be transparent for all the other Data Model Transformation Tools which will still be consuming data encoded with the unchanged standard data model. Otherwise, if the Data Model Transformation Tool provided custom data model to custom data model translation rather than using a central standard data model, all the Data Model Transformation Tools of the other services would need to be updated according to the new specifications of the changed data model.

4.1.2 Reference Models

In this subsection, the main domains relevant to EFPF are introduced along with candidate data models and standards that are relevant for each domain.

The following overview will help in selecting the data models that satisfy majority of the EFPF ecosystem needs in term of data exchanged between its components and platforms.

4.1.2.1 Industrial IoT and Industry 4.0

Industrial Internet of Things (IIoT) involves interconnected things networked together with applications, services, and people. The connectivity permits data collection, analysis, improved efficiency, and tapping into the potential of data value. Things refers to the semantic representation of a cyber-physical system, i.e., IoT devices embedded with electronics, software, sensors/actuators, and connectivity to enable objects to exchange data with the services, applications, people and/or other connected devices. Usually IIoT scenarios are linked to manufacturing; however, IIoT is brother field that comprises multiple vertical markets such as energy, finance, healthcare, industrial, residential, retail and transportation according to oneM2M and ETSI standard bodies. In EFPF, some pilot scenarios fall into the IIoT domain mainly in the industrial vertical market such as the “bins’ fill level monitoring” or the “working environment monitoring” scenarios. Those scenarios are reflecting the pilot topics and will be described more in depth in a following deliverable. The goal is to enable interoperability during the data exchange process; therefore, potential candidate standards relevant in the IIoT domain are studied and analysed to see their suitability for EFPF pilot scenarios. The candidate standards are:

- **OGC Sensor Web Enablement (SWE) initiative:** OGC SWE goal is to establish interfaces and protocols that enable "sensor web" so that applications can access sensors and their observations over the web. The SWE standards include:
    1. Observation Measurement (O&M) [OAM20]: provides standard models and XML Schema for encoding observations and measurements from a sensor (user-centric view)
    2. SensorML [SML20]: provides standard models XML Schema for describing sensors and measurement processes (provider-centric view)

3. Sensor Observation Service (SOS) [SOS20]: defines a web service interface that allows querying observations, sensor metadata, and representation of observed features
4. Sensor Planning Service (SPS) [SPS20]: defines a web service interface for queries that provide information about sensor capabilities.
5. SensorThings API [STA20]: defines an open OGC API that allows managing, storing, sharing, and analysing IoT-based sensor observation addressing the syntactic and semantic interoperability between IoT devices and services.

- **Web of Thing - Thing Description (TD)** [WTD20]**:** allows semantic description of things and provides a set of interactions based on a small vocabulary to interface with things. TDs are encoded in a JSON format that allows JSON-LD processing and defines the following aspects:
  1. Semantic metadata: provides a generic and context enriched information about a Thing
  2. Thing's interaction resources describes properties, actions, and events
  3. Security: describes the prerequisite to access Things
  4. Communication: defines the protocols, data exchanged formats, and bindings to an interaction resource supported by a Thing.
- **OPC UA Part 100: Devices** [OUD20]**:** is a companion specification that features an information model for devices and is based on the OPC UA information model framework. The device model provides a model to identify device type through properties such as serial number, model, and device type. It also allows the organization of parameters and commands of a device, the grouping of devices, usage of device interface type (i.e. VendorNameplate Interface and DeviceHealth Interface), alarm handling, initial device set up (i.e. device security), and physical and logical network representation among others.

4.1.2.2 Supply Chain and Logistics

In transportation, and also part of the circular use case, the "BiTAS Std 120-2019: LOCATION COMPONENT SPECIFICATION" [BIT19] and "BiTAS Tracking Data Framework Profile" [BTD20] from Blockchain in Transport Alliance Standards Council (BiTAS) are the main candidates. It is a recently developed data model and format for events and data regarding shipment and transport.

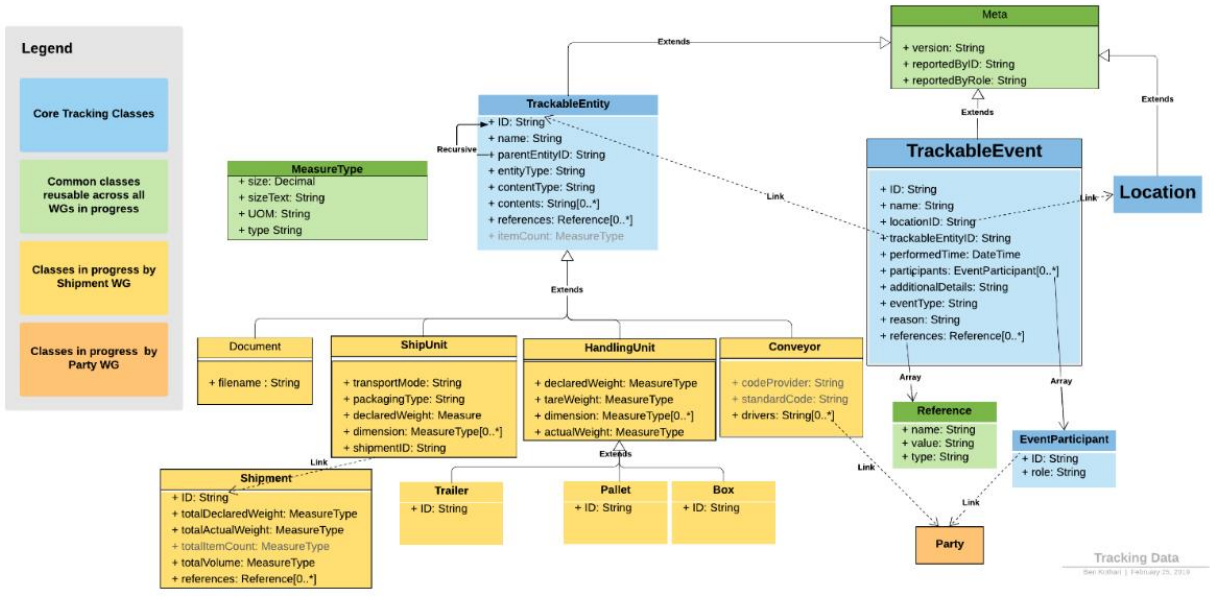


Figure 107: Tracking Entity Model (Blockchain in Transport Alliance Standards Council (BiTAS))

EPCIS [EPC20] is a GS1 [GS120] standard using Electronic Product Code (EPC) data, which is used for the track & trace feature in the NIMBLE base platform to share product movement and business process events status. It will also be used in the blockchain pilot.

UBL has been mentioned by pilot partners as a candidate to externally represent proprietary ERP system schemas.

4.1.2.3 Data Models - Ongoing Activities

The goal of data model interoperability is to enable an information exchange between applications that use different models to encode data. According to RFC 3444 [RFC34], Data Models define data objects at a lower level of abstraction and include implementation- and protocol-specific details. Application developers use data model specification as a reference how to encode data structures before they can be exchanged with other applications. Those specifications can include recommendations for data serialization formats, i.e. XML, JSON, or CSV, as well as descriptions to convert internal data structures to the chosen serialization format.

Different application developers use different models to encode their data due to several reasons, e.g. different implementation frameworks or data exchange protocols. This is very often the case when the interacting applications exist in different domains. Every domain has its unique requirements and therefore its own tools, standards, and concepts to support the execution and operation of applications. Figure 108 gives an overview of the previously introduced data models that are common for the presented domains.

Besides the two scenarios presented more in depth, it is worth mentioning the Platform Marketplace scenario. This scenario has not been investigated in depth since the pilots have not made significant progress yet. However discussions have been performed and an early agreement has been reached on UBL as data model worth investigating for this scenario.

Domain	Common data model
Industrial IoT, Industry 4.0	OGC SensorThings API

	W3C WoT TD
	OPC UA Part 100
Supply Chain & Logistics	BITAS
	GS1 - EPCIS
Platform Marketplace	UBL

Figure 108: Overview of previously introduced data models from different domains

The task of data model interoperability is to ease a translation between different models by selecting standards that would be supported by the EFPF platform. On technical level, this means to convert data from services of external or base platforms to the EFPF standards and models. Therefore, we investigated the different domains that are relevant for EFPF and pre-selected common standard models as presented previously. Furthermore, an analysis of the single pilot scenarios was conducted (Section 4.2) in order to check the models for applicability. However, this remains an on-going activity with some results already becoming available for use in the project.

4.2 Pilot Scenarios Analysis

The goal of this section is to present an overview of different pilot scenarios with a focus on data model used in each of them. This is done to put the basis for a subsequent analysis of overlaps, conflicts or compatibility between different data models. This work has been done by combining the surveys collected about the data models used from tools and services with the pilot applications. This study will result in a final evaluation that will aim at solving the issues coming from the heterogeneity of data models used by the different pilots, selecting a subgroup of them to be adopted as reference in the EFPF platform. All the EFPF internal communications will be aligned with the reference data models by using translation modules.

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 and meetings rooms, a manufacturing shop floor where production process occurs, or a warehouse where materials and parts are stored. Various aspect can be monitored in a working environment depending on the working environment and the needs of users. Some of the aspects that can be monitored are air quality, energy consumption, space utilization, safety, and comfort.

In the working environment monitoring scenario, an example use case is presented to identify and illustrate the type of data exchanged between EFPF solutions and Data Spine. The diagram bellow shows the EFPF tools and services involved in the “temperature, humidity and air quality monitoring in the shop floor” use case as well as the data exchanged between them. In this use case, the actor is the production line manager who wants to monitor the temperature, humidity and air quality values of the production area where milling machines are located so that the machine parameters can be adjusted when the values are beyond the threshold. Thus, the production manager specifies the threshold values for each area then informs the actor via SMS, email, or notifications when the threshold is reached.

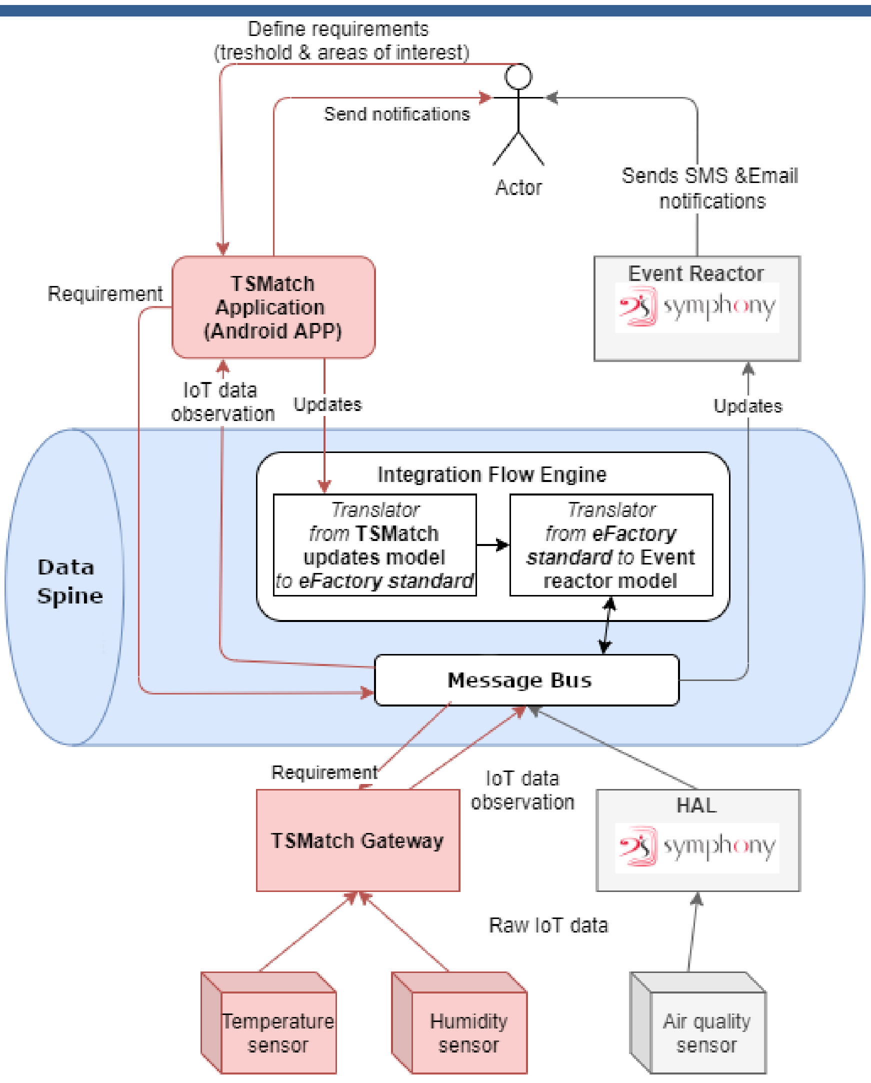


Figure 109: Illustration of the EFPF services involved in the temperature, humidity and air quality monitoring in the shop floor use case

In the use case described above, HAL connector, TSMatch gateway/application, and the event reactor service are using different models, which makes the integration process difficult and time consuming; therefore, the objective is to build model transformation “translator” to ease and improve interoperability between the services. The translators are responsible for transforming the models used by the services to the specified EFPF standard for the domain.

4.2.2 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. The sensors provide early (real-time) notification of the recyclable and scrap bins fill levels and suggest optimal routes for collecting bins within the factory. Overall, minimization of the total distance from bins to container and improvements in containers’ fill level management are expected.

The Scrap and Recyclable Waste Transportation is triggered by a full bin in the shop-floor. For the Prediction engine to be able to estimate and propose the optimal path to follow for the transportation of waste to a central bin outside the production line, a fill level sensor for the internal bins has to be designed and developed.

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.

A board with a Lora communication module is needed to push information towards the data collection framework. The sensors take raw 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. If this value exceeds a certain threshold, an event is automatically triggered (i.e. a bidding process for the scrap or finding optimal routes for waste collection).

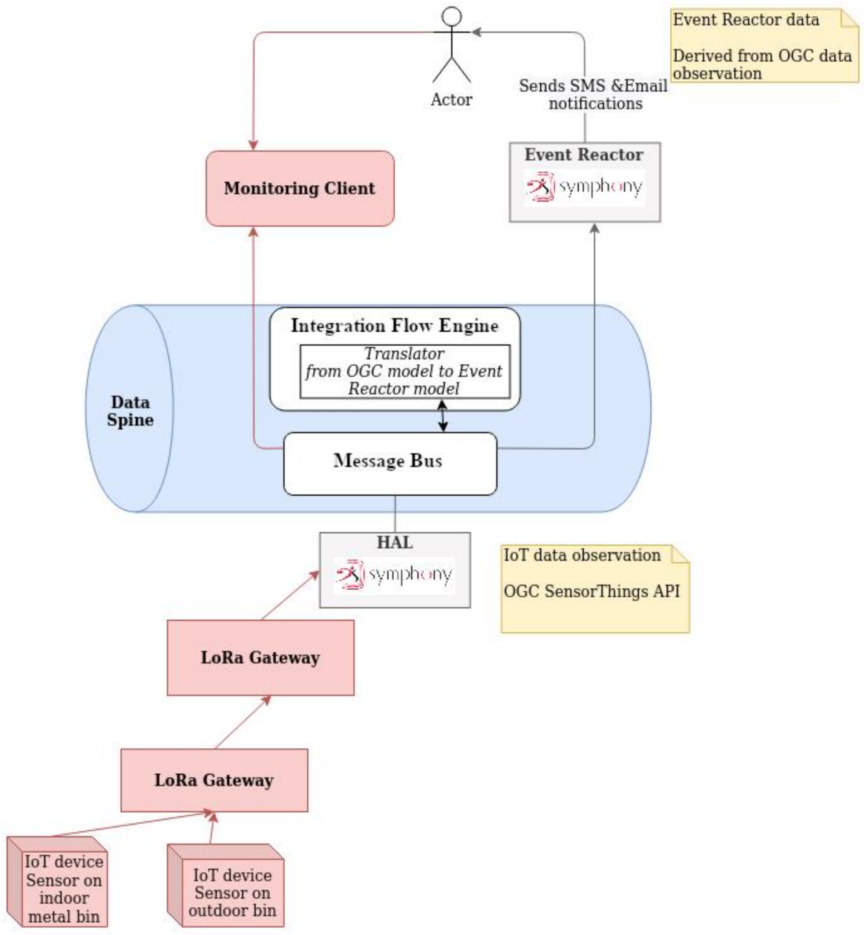

Actor
Event Reactor data
Derived from OGC data observation
Sends SMS &Email notifications
Monitoring Client
Event Reactor
symphony
Data Spine
Integration Flow Engine
Translator from OGC model to Event Reactor model
Message Bus
HAL
symphony
IoT data observation
OGC SensorThings API
LoRa Gateway
LoRa Gateway
IoT device Sensor on indoor metal bin
IoT device Sensor on outdoor bin

Figure 110: EFPF services involved in bins fill level monitoring

As in the previous use case, HAL connector and the Event Reactor service are using different models, which makes the integration process difficult and time consuming; therefore the HAL connector is using OGC-ST standard, one of the data models that could possibly become reference in the EFPF platform.

Event Reactor uses a data model derived from OGC-ST data instead. To overcome the data model difference a translation will be applied exploiting NiFi translator modules at Data

Spine level. By doing so, the data model used into the Data Spine will be the reference one, that will be translated once a different data model needs to be output or input.

4.2.3 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.

The first topic of the pilot, provisioning of instructions to the operator, the EFPF Data Spine is not used since all the data flows happen outside of it, thus it will not be considered in the scope of this deliverable.

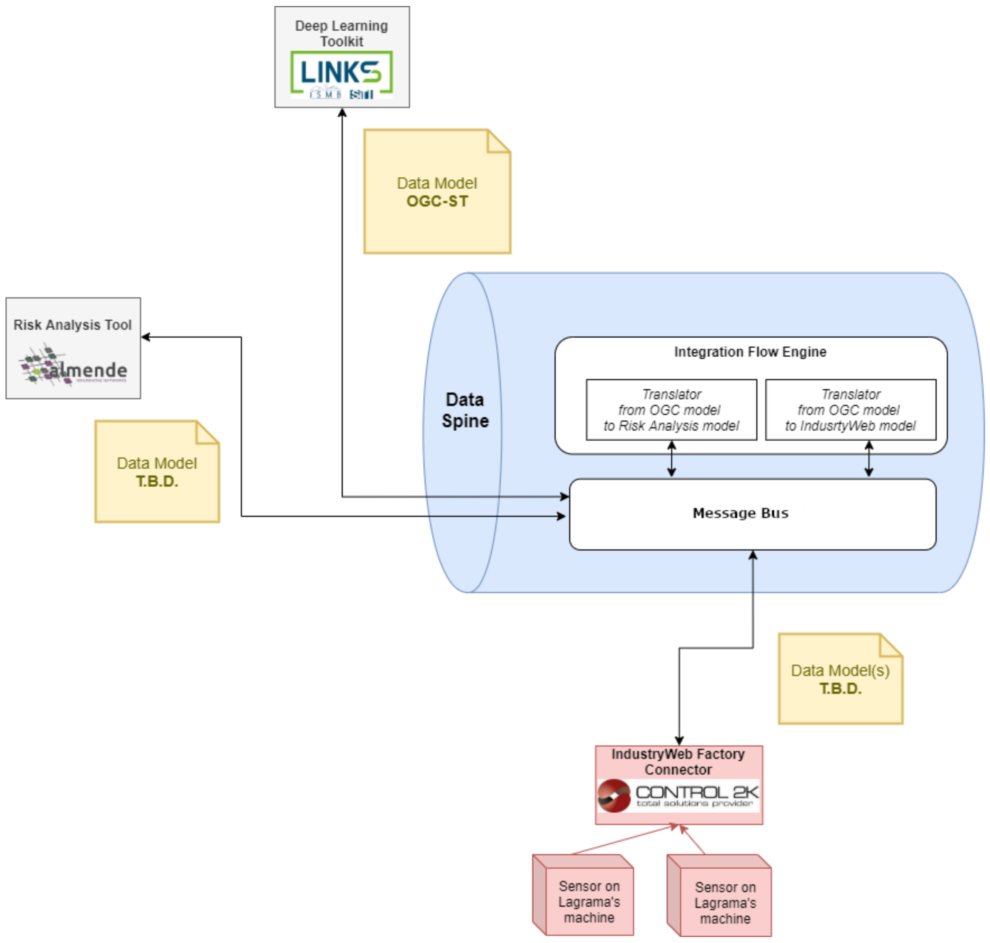

Deep Learning Toolkit
LINKS
Data Model OGC-ST
Risk Analysis Tool
almende
Data Model T.B.D.
Data Spine
Integration Flow Engine
Translator from OGC model to Risk Analysis model
Translator from OGC model to IndusrtyWeb model
Message Bus
Data Model(s) T.B.D.
IndustryWeb Factory Connector
CONTROL 2K total solutions provider
Sensor on Lagrama's machine
Sensor on Lagrama's machine

Figure 111: Use of EFPF Data Spine from Production Optimization pilot

In the second topic, providing risk forecasting and predictive maintenance solutions, the pilot makes use of the Data Spine by sending readings from sensors located in the bending machine to the analytics tools which will output information about risk and fault forecasting.

The Industryweb Factory connector provides a MQTT digital interface for the analogical readings provided from the different (temperature, current, etc.) sensors installed on the physical machine. These readings can be referred to as machine KPI data. This data is then fetched from the tools which make use of those from the MQTT broker. In order to provide interoperability between the Industryweb Factory connector and the tools making use of its data it is necessary to analyse the data models used from both the Industryweb Factory connector and the tools. In case those are found to be different, multiple translators to/from a given standard data model will be developed. These translators will also be useful in case, in the future, different tools will be added to the pilot to provide access to the machine KPI data without breaking the existing flows.

At the moment, due to the pilot still being under development the data model used from the Industryweb Factory connector is not yet finalized so it will be considered as custom and not standard.

For what concerns the Risk Analysis component, it uses a proprietary custom data model. To fetch the data needed to perform the risk assessment from the MQTT broker a translator to its custom data model will need to be developed, due to the aim of keeping the data models used in the Data Spine standard.

About the predictive maintenance service, provided form the Deep Learning Toolkit component, a translator will be developed only in case the standard data model used in the Data Spine for machine KPI data is different from the OGC-SensorThings standard, which is used from the aforementioned component.

4.2.4 Tendering and Bid Management

The opportunities portal will give SMEs a chance to offer their services for income generating opportunities. This service will be aimed at industrial/manufacturing SMEs specifically, as an example an automotive SME selling brake drums.
Tender and Bid Management or opportunities portal will be 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 will include tenders, small work opportunities and allow suppliers to submit applications; along with a messaging system. This will assist procurers and suppliers to find business better.
There will be a range of different data 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 will 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 Supply Chain Transparency (WASP)

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 is to provide live visualization at a glance and end to end visibility of the production process. The system can also send status updates for e.g. orders and warnings if a problem is detected at some step. The scenario is based on the JIRA user stories EF-19 and EF-303. The main component in the pilot is the Workflow and Service Automation Platform (WASP) tool. WASP allows uses to design, execute and monitor distributed workflows and service orchestration using Business Process Model and Notation (BPMN) graphical representation.

The information needed to model the processes can be found in the Enterprise Resource Planning and it needs to be published to the message broker to be used from WASP. Translators on the Integration Flow Engine ensure interoperability between different data models. The processes are then executed in the workflow engine and status and information from distributed (supply chain) activities taking place across multiple facilities is displayed on a dashboard.

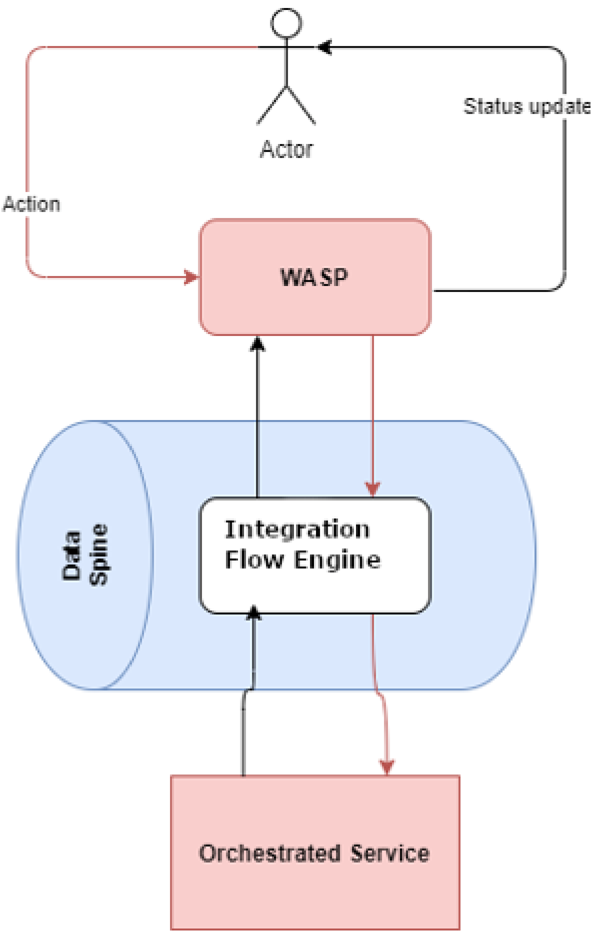


Figure 112: Supply Chain Transparency

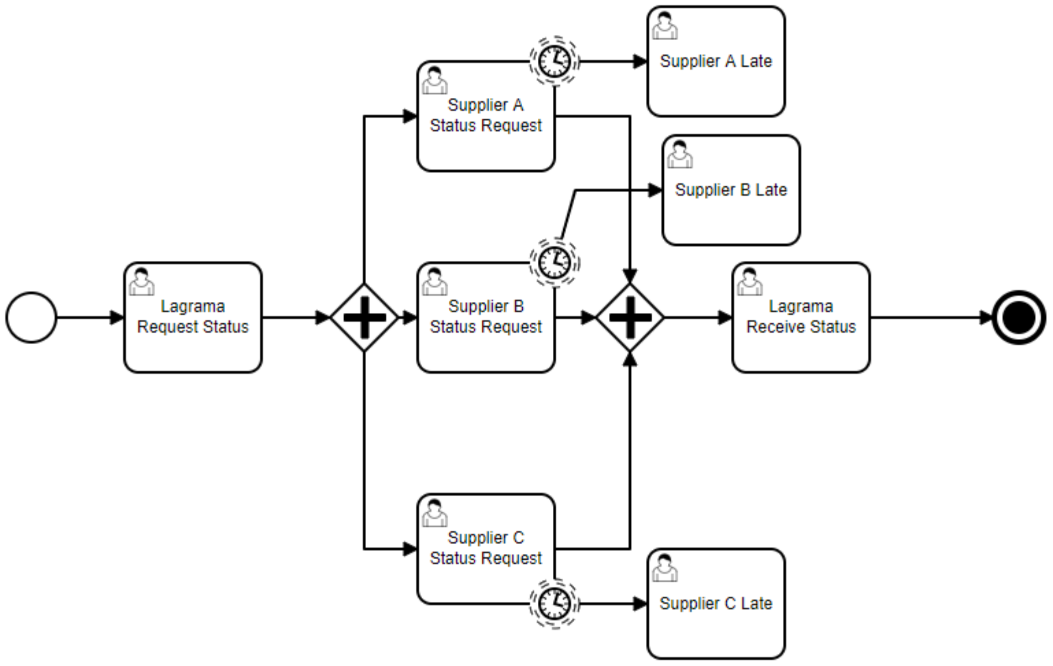


Figure 113: BPMN Process Visualization.

4.2.6 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 will be used in the blockchain shipping applications, and EPCIS in the order processing. The only information expected to make use of the Integration Flow Engine data model translators at this stage is the Enterprise Resource Planning data which needs to be published to the message broker to be used from WASP.

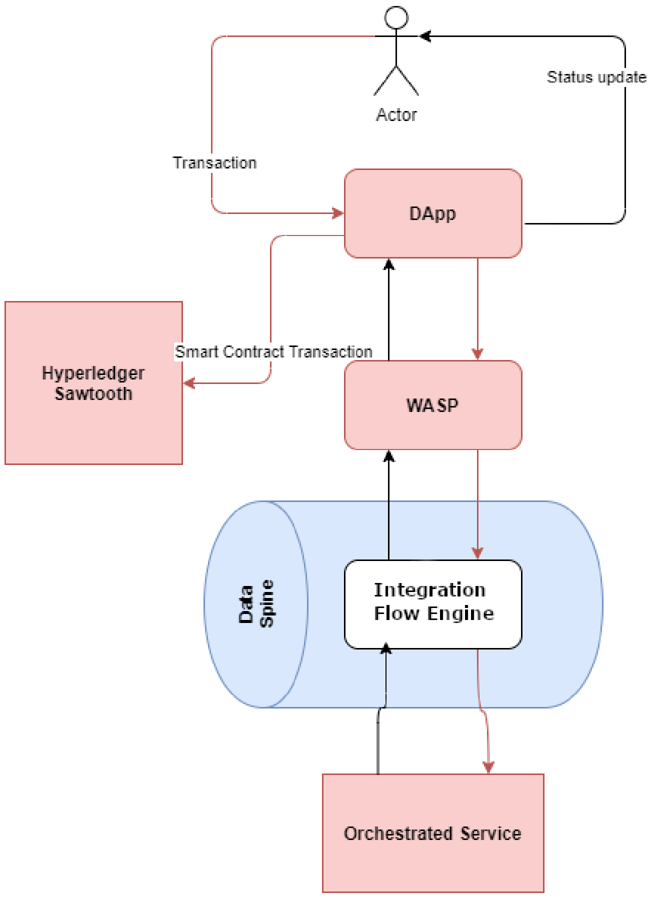


Figure 114: Blockchain DApp Data Spine use.

4.2.7 Matchmaking

EFPF platform federated search and associated matchmaking functionality requires a standard interoperable data model to define the metadata related to the suppliers and their products and services from different base platforms in EFPF. For this purpose, an EFPF matchmaking ontology is defined namely EFPF Manufacturing Ontology (EFONT). COMPOSITION, SMECluster, NIMBLE and other platforms connecting to EFPF uses their own metadata structure to define their value-units. These value units can be identified as service providers, products and services offered by the base platforms.

The metadata of these value units can be structured with the following concepts of EFONT ontology at a high-level. The concepts and their attributes defined are inspired by the

Universal Business Language (UBL)[11] specifications on Supplier PartyType and CatalogueType.

- A Class / Category of a product/ service/ partner's capability has 0 or more properties,
- A Property describes the product/service class in detail, e.g. length, height, certificates,
- An Item is an instance of a Class / Category. Each Class / Category has 1 or more item instances representing the actual product/service or partner's capability that will be manufactured/ provided by a party/ company,
- A Party has attributes such as a legal-name, keywords and activity sector that extend a variety of attributes for matchmaking processes.

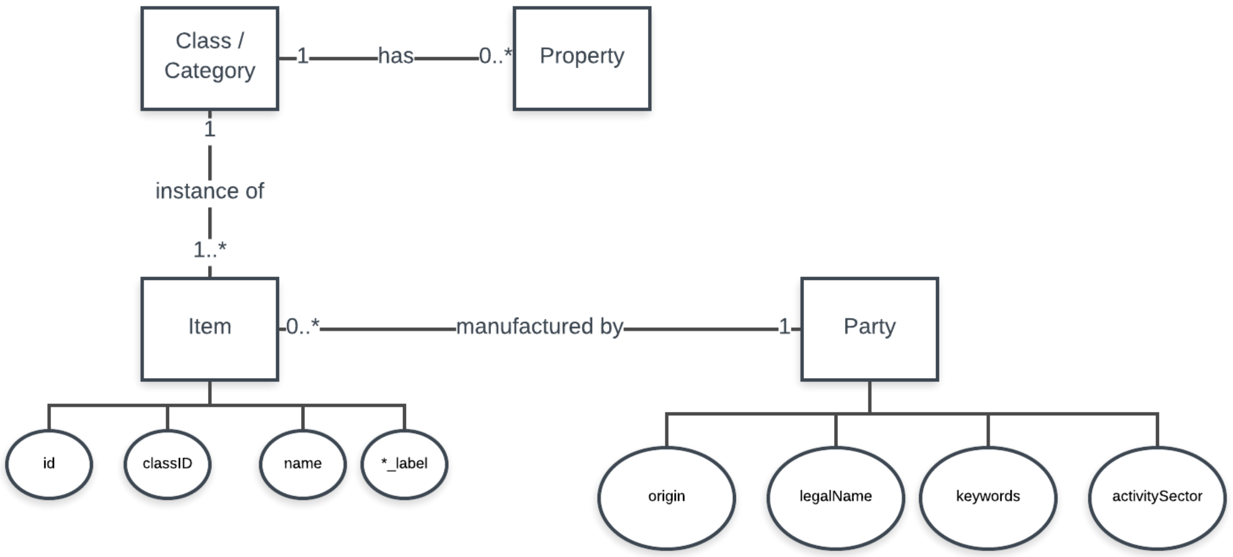


Figure 115: A High-Level View of the EFPF Manufacturing Ontology

Above figure depicts how these concepts are related in the ontology. We extended the above ontology with more attributes which will be useful for matchmaking transactions. The additional relations/attributes were added mainly to the Item and Party concepts by analysing the different schemas used across the base platforms. More details regarding matchmaking data models and data flow are discussed in D5.1.

4.2.8 Conclusions from Pilot Scenario Analysis

The result from pilot analysis is that use cases are clearly distributed in quite different domains, as previously identified in Section 4.1.2.

The first two pilot scenarios "Working environment monitoring" and "Bins' fill level monitoring" have been included in "Industrial IoT and industry 4.0" and both pilots are focused on gathering data from the physical environment and send it to servers that will perform operations on this data.

Looking at these two pilots it is clear that the common data model is represented by OGC and other two custom data models used by the Event Reactor and TSMatch application. In this case the best solution will be to use the OGC data model as reference one and create

[11] http://www.datypic.com/sc/ubl21/ss.html

translation modules to enable the components that use custom data models communicate in the Data Spine context.

To be mentioned in this category there is also the “Production Optimisation Pilot” that make use of OGS standard for the Deep Learning Toolkit component and custom data models for the others.

For “Supply Chain and Logistics” domain we can find the “Supply chain transparency” and “Blockchain and smart contracting” pilots. In these cases there is a wide selection of standard models specifically oriented to the use cases like BiTAS and GS1- EPICS. In this context it is difficult to identify a reference data model that will cover general needs of the platforms.

Regarding “Platform Marketplace” domain, there are different customs and evolving data models used to exchange information. UBL probably will be used as reference in business domain and in future developments will be clarified if can be useful to cover all EFPF platform needs.

4.3 Data Model Interoperability Tools

As stated in the previous subsections, the EFPF platform is a collection of smart tools and services that aim to cover the complete lifecycle of production and logistic processes. Also, the EFPF ecosystem aims at standardizing the data models used to describe the data exchanged between these tools and services. However, it is to be expected, even in the best possible scenario, that some tool can come with proprietary data models that make their integration with other tools and services a hard job.

To solve this problem, two routes can be followed, upgrade the tools to use one between the proposed EFPF data models, or make the tool interoperable by translating the data produced or consumed from that tool on the fly.

The first option is to be preferred since standardization right at the source would bring the most advantages, however this is not always applicable. An example is a tool already integrated in a process, its upgrade to a standard data model could lead to a service disruption. In this case it is better to proceed with a data model transformation outside from the tool. These data model transformations are completely transparent from the tools’ point of view and can happen both on the Integration Flow Engine and outside from it on dedicated infrastructure.

4.3.1 JOLT

Jolt (JsOn Language for Transform) is a JSON to JSON transformation library written in Java [JGH20]. The Jolt specification (spec) for performing the transformation is also written in the form of a JSON document. Jolt provides a set of transforms as illustrated in Figure 116 and each transform has its own Domain Specific Language (DSL) to specify the transform concern. These transforms can be used to transform the structure of JSON data. The inclusion of several ‘wildcards’ into the DSL of the shift transform make it the most powerful among these enabling it to perform about 80% of the transformation work.

Transform	Description/Purpose
shift	To copy data from input to desired place(s) in the output JSON tree
default	To add default values to the output JSON tree

remove	To remove data from the tree
sort	To sort the Map key values alphabetically for debugging purposes
cardinality	To "fix" the cardinality of input data. E.g., the "urls" element is usually a List, but if there is only one, then it is a String
java	To perform value transformation through Java code

Figure 116: Jolt Stock Transforms

These transforms can be chained together to form the complete transformation spec. Figure 117 shows an example of Jolt transformation with input and output JSON documents, and the Jolt spec that chains together shift and default operations together to perform the transformation.

Input	Output	Jolt spec

`{`
`  "temperature": 25,`
`  "humidity": 81,`
`  "wind": 14,`
`  "precipitation": 51,`
`  "location": "Bonn",`
`  "timestamp": "2020-06-04T13:49:00.4096331Z"`
`}`
	`{`
	`  "weather-data" : {`
	`    "temperature" : 25,`
	`    "humidity" : 81,`
	`    "wind" : 14,`
	`    "precipitation" : 51,`
	`    "location" : "Bonn",`
	`    "timestamp" : "2020-06-04T13:49:00.4096331Z"`
	`  },`
	`  "attributes" : [ "temperature", "humidity", "wind", "precipitation", "location", "timestamp" ],`
	`  "metric-system" : "custom"`
	`}`
		`[`
		`  {`
		`    "operation": "`**`shift`**`",`
		`    "spec": {`
		`      "@": "weather-data",`
		`      "*": {`
		`        "$(0)": "attributes"`
		`      }`
		`    }`
		`  },`
		`  {`
		`    "operation": "`**`default`**`",`
		`    "spec": {`
		`      "metric-system":`
		`          "custom"`
		`    }`
		`  }`
		`]`

Figure 117: Jolt Transform Example

The downside of Jolt is that it is not Turing complete i.e., not every aspect of data transformation can be achieved with Jolt. Jolt can be used to perform structural transformations on data but not manipulate values. Java code needs to be written to perform value transformations. Jolt is certainly not intuitive, but easier to learn as compared to some other data transformation tools/languages such as XSLT. Having lesser options helps to keep it simple. Thus, there is a trade-off between 'ease of use' and transformation 'coverage'.

The documentation for Jolt is comprehensive enough for the functionality it offers. In addition, in order to learn Jolt, a Jolt playground [JPG20] is also available. Apache NiFi, the dataflow management platform that is used to realise the Integration Flow Engine of the Data Spine contains Jolt as a processor [JNF20]. Therefore, it can be easily added to an integration flow to perform data transformation.

4.3.2 XSL Transformations (XSLT)

XSLT12, a WC3 standard, is a language for transforming XML documents into other XML documents. It is used in conjunction with XPath 2.0 to write rules to match parts of a parsed XML document and compile a new XML document. However, since a JSON to XML conversion is part of the specification and the rules can define functions and use regular expressions, it is applicable for other formats as well. E.g. Avro can be converted from binary to JSON and then to XML for transformation by XSLT. (Any text representation can be generated, as long as it is included between two XML root elements.) CNet have experience building integration software for the AdsML13 standard using libraries of XSLT 2.0 transformations. XSLT transformations can use imports, so libraries of rules can be imported and re-used. Partial support for transformation of a set of standards can be built and extended in a modular fashion, gradually covering more and more of the schema in a value co-creation process. The XSLT transformation documents can be version managed using e.g. Git or other tools. There are scripting capabilities and regular expression support in some environments.

Factors that speak against XSLT and XML are that today, it is not as widespread as e.g. JSON is and there may not be a large base of developers familiar with it. That all transformations must start and end in valid XML is also a limitation that needs to be circumvented using the above methods.

The TransformXml processor in NiFi transforms the XML payload using the provided XSLT file.

4.3.3 ExecuteScript

ExecuteScript [ES20] is a processor provided by Apache NiFi that facilitates users in writing a script for performing data transformation. It takes data input in the form of a FlowFile – the data serialization format used by NiFi, processes it as specified in the script and finally generates another FlowFile as output that contains the transformed data. The languages supported for writing the data transformation script are Clojure, ECMAScript, Groovy, Lua, Python and Ruby.

Figure 118 illustrates an example where a Python script is used to perform the data transformation.

ExecuteScript: Data transformation with a Python script

Input

```
{
  "userId": 1,
  "id": 18,
  "title": "lorem ipsum",
  "completed": false
}
```

Output

```
{
  "completed": true,
  "title": "lorem ipsum"
  "userId": 19,
  "id": 18
}
```

Python Script

```
import json
from org.apache.commons.io import IOUtils
from java.nio.charset import StandardCharsets
from org.apache.nifi.processor.io import StreamCallback
```

[12] https://www.w3.org/TR/xslt/all/
[13] https://www.adsml.biz/

```
class PyStreamCallback(StreamCallback):
def __init__(self, flowfile):
        self.ff = flowfile
        pass
def process(self, inputStream, outputStream):
        text = IOUtils.toString(inputStream, StandardCharsets.UTF_8)
        tmp = json.loads(text)
        tmp['userId'] = tmp['userId'] + int(self.ff.getAttribute('value'))
        tmp['completed'] = not tmp['completed']
        text = json.dumps(tmp)
        outputStream.write(bytearray(text.encode('utf-8')))

flowFile = session.get()
if (flowFile != None):
        flowFile = session.write(flowFile, PyStreamCallback(flowFile))
        session.transfer(flowFile, REL_SUCCESS)
```

Figure 118: ExecuteScript Example

The ExecuteScript processor has the following shortcomings:

1. The main concern is with the engine of Python is that the processor uses Jython, the Java implementation of Python, as opposed to pure Python. Therefore, the packages that can be installed and used need to be written in pure python. Also, the path of these modules needs to be set in the configurations of the Processor.
2. For NiFi to reload the modified version of a script, the ExecuteScript processor that runs it needs to be restarted.
3. It is extremely difficult to test and debug.
4. As per the official documentation of NiFi, this processor is experimental and therefore, the impact of sustained usage not yet verified.

4.3.4 Ad-Hoc Microservices

A custom microservice is an option which can be used in case all the previously listed options are not applicable because of any reason.

When using this solution, the data is fetched from the Integration Flow Engine of the Data Spine from a custom developed and maintained component which is hosted outside from the Data Spine. Once there the data can be transformed from a custom to a standard data model. This process can be performed using a framework of choice from the developers without the need for it to be supported from the Integration Flow engine. Once transformed the data can be then pushed back to the Data Spine to be used from other services.

This option has the obvious advantage of not being tied to the Integration Flow Engine so, in case of a replacement of the framework behind it. There will not be the need to replace the data model translation tool with it. Moreover, as stated before, it can use any framework of choice of the developer in charge of maintaining it.

The main disadvantages of this solution are:

1. The increased complexity in development for getting and pushing data from / to the Integration Flow Engine.
2. Increased delay added to the data flow from the source to the destination due to the added possible network delays.

4.3.5 Summary and Future Work

Three different data model transformation tools that are integrated in the Integration Flow Engine of the Data Spine tools facilitate the system integrator user in writing the data model/schema mapping rules or source code for data transformation that is specific to the interaction between a particular pair of services. Each tool has its own advantages and limitations - for some tools, there is a steeper learning curve (e.g. XSLT); for others, they accomplish most of the required transformation, but are not Turing complete (e.g. Jolt).

On the downside, the data transformation process remains largely manual. In addition, in order to write rules or source code for data transformation, the system integrators need to know the semantics for the data. However, there is no system available to capture the semantics of the data to be transformed and guide the user while writing the data transformation rules. A new ‘Data Model Transformation Tool Suite’ is being designed to address these problems. Section 3.2.3 in the deliverable D4.1 describes this tool in detail.

In conclusion, the system integrator users are provided with tools that offer a diverse range of options for performing data transformation and efforts are being made to ensure that the data transformation process becomes easier, faster and more accurate in the future

4.4 Future Work on Data Model Interoperbility

Pilot analysis has highlighted different issues: some easily solvable with the adoption of a reference data model and a set of custom data models derived or related from the reference one, making the job of translating easy; others with a wider set of possibilities hardly solvable like in the previous case.

After this first phase of pilots and data models evaluation, the system integrator users in EFPF project will be provided with a set of data models to be used as reference and a set of tools provided by the Integration Flow Engine of the Data Spine to perform data model transformation.

As next steps of task T3.5 will be developed a set of translators to integrate each pilot with the Data Spine according to guidelines given by pilot scenario analysis and data model selection. After this phase the translators will be tested in pilot contexts to validate them.

5 Conclusion and Outlook

In this deliverable, the current state of design and realisation of the Data Spine, the update to the architecture of the EFPF ecosystem, the APIs for tools, services, systems and platforms of the EFPF ecosystem and data model interoperability layer were presented.

The Data Spine is already capable of integrating synchronous as well as asynchronous services through it and enable communication in the EFPF ecosystem. It bridges the interoperability gaps between services at the levels of security, communication protocol and data model and already provides the protocol connectors and data transformation tools required at this stage. The next steps are creating more documentation and tutorials to explain service integration through the Data Spine from the perspectives of service providers and service consumers, enhancing the levels of automation for design-time configuration, enhancing security integration, ensuring high availability and high throughput, etc. The abstract architecture and realisation of the Data Spine presented in this deliverable will be subsequently developed and enhanced based on the identification of new requirements.

The APIs of the tools, services, systems and platforms that constitute the EFPF ecosystem, the API management and Interface Contracts management mechanisms would be developed and enhanced further as well.

The data model interoperability layer, after the first phase of pilots and data models evaluation, will provide the system integrator users in EFPF project with a set of standard data models to be used as reference. The system integrator users are already provided with tools that offer a diverse range of options for performing data transformation and efforts are being made to ensure that the data transformation process becomes easier, faster and more accurate in the future.

Finally, the implementation of EFPF pilots’ use cases followed by the Open Call Experimentation would provide a great opportunity for the evaluation of the architecture and the implemented infrastructure, for discovering new requirements and, enhancing and strengthening the EFPF ecosystem even further.

Annex A: History

Document History	
Versions	V1.0 • Final version of the document V0.8 • Address internal review comments V0.7 • Sent for internal reviews V0.4 – V0.6 • Collection of contributions from partners • Preparation of a consolidated document, formatting V0.3 • Restructured Sections 2.2 and 3.2 V0.2 • Creation of internal structure V0.1 • Creation of main Sections

Contributions	FIT: • Rohit Deshmukh • Alexander Schneider • Vinoth Pandian ICE: • Usman Wajid SRFG: • Violeta Damjanovic-Behrendt • Nirojan Selvanathan • Dileepa Jayakody ASC: • Norman Wessel • Brian Clark VLC: • Happy Dudee CNET: • Mathias Axling • Matts Ahlsen CERTH: • Alexandros Nizamis • Dimosthenis Ioannidis • Ioannis Iakovidis • Terzi Sofia C2K: • Simon Osborne NXW: • Alì Nejabati ALM: • Carolyn Langen • Carlos Hermans AID: • Fernando Gigante FOR: • Nisrine Bnouhanna • Hendrik Walzel SRDC: • Senan Postaci LINKS: • Edoardo Pristeri • Jure Rosso CMS: • Carlos Coutinho

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 June, 2020. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. Accessed Sept 2019

[JPG20] Jolt playground. http://jolt-demo.appspot.com/. Accessed June, 2020.

[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 June, 2020.

[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 June, 2020.

[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 June 2020.

[NOG] Apache NiFi Overview Guide. https://nifi.apache.org/docs/nifi-docs/html/overview.html. Accessed June, 2020.

[NAR20] Apache NiFi REST API Documentation. https://nifi.apache.org/docs/nifi-docs/rest-api/index.html Accessed June, 2020.

[NAG20] Apache NiFi Administrator's Guide: https://nifi.apache.org/docs/nifi-docs/html/administration-guide.html#tls_generation_toolkit Accessed June, 2020.

[RMQ20] RabbitMQ Documentation. https://www.rabbitmq.com/documentation.html Accessed June, 2020.

[CAQ20] RabbitMQ for beginners. https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html Accessed June, 2020.

[OAS20] OpenAPI Specification 3.0.3. https://swagger.io/specification/ Accessed June, 2020.

[AAS20] AsyncAPI Specification 2.0.0. https://www.asyncapi.com/docs/specifications/2.0.0/ Accessed June, 2020.

[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 June, 2020.

[KON20] Kong 2.0 Open Source API Gateway. https://konghq.com/kong/. Accessed June, 2020.

[OAM20] OGC SWE: Observation Measurement (O&M). https://www.opengeospatial.org/standards/om. Accessed June, 2020

[SML20] OGC SWE: SensorML https://www.opengeospatial.org/standards/sensorml. Accessed June, 2020

[SOS20] OGC SWE: Sensor Observation Service (SOS). https://www.opengeospatial.org/standards/sos. Accessed June, 2020

[SPS20] OGC SWE: Sensor Planning Service (SPS). https://www.opengeospatial.org/standards/sps. Accessed June, 2020

[STA20] OGC SWE: 5. SensorThings API. https://www.opengeospatial.org/standards/sensorthings. Accessed June, 2020

[WTD20] Web of Thing - Thing Description (TD). https://www.w3.org/TR/wot-thing-description/. Accessed June, 2020

[OUD20] OPC UA Part 100: Devices. https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-100-device-information-model. Accessed June, 2020

[BIT19] BiTAS Std 120-2019: LOCATION COMPONENT SPECIFICATION. https://www.bita.studio/s/BiTAS-Location-Component-Specification-v4-rf7s.pdf. Accessed June, 2020

[BTD20] BiTAS Tracking Data Framework Profile. https://static1.squarespace.com/static/5aa97ac8372b96325bb9ad66/t/5c7e88397817f73e6c60a967/1551796284047/BiTAS+Tracking+Data+Frameowork+Profile+v9_ISTO.pdf. Accessed June, 2020

[EPC20] EPCIS. https://www.gs1.org/epcis/epcis/1-1. Accessed June, 2020

[GS120] GS1. https://www.gs1.org/. Accessed June, 2020

[RFC34] RFC3444. https://tools.ietf.org/html/rfc3444. Accessed June, 2020

[RMP20] RabbitMQ Messaging Patterns. https://www.rabbitmq.com/getstarted.html. Accessed June, 2020.

Annex C: API Specifications

1. Data Spine Service Registry

1.1. REST API for Lifecycle Management of Services

```
{
  "openapi" : "3.0.0",
  "info" : {
    "version" : "3.0.0",
    "title" : "LinkSmart Service Catalog REST API"
  },
  "tags" : [ {
    "name" : "sc",
    "description" : "Service Catalog"
  } ],
  "paths" : {
    "/" : {
      "get" : {
        "tags" : [ "sc" ],
        "summary" : "Retrieves API index.",
        "parameters" : [ {
          "$ref" : "#/components/parameters/ParamPage"
        }, {
          "$ref" : "#/components/parameters/ParamPerPage"
        } ],
        "responses" : {
          "200" : {
            "description" : "Successful response",
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/APIIndex"
                }
              }
            }
          },
          "401" : {
            "$ref" : "#/components/responses/RespUnauthorized"
          },
          "403" : {
            "$ref" : "#/components/responses/RespForbidden"
          },
          "500" : {
            "$ref" : "#/components/responses/RespInternalServerError"
          }
        }
      },
      "post" : {
        "tags" : [ "sc" ],
        "summary" : "Creates new `Service` object with a random UUID",
        "requestBody" : {
          "$ref" : "#/components/requestBodies/Service"
        },
        "responses" : {
          "201" : {
            "description" : "Created successfully",
            "headers" : {
              "Location" : {
                "description" : "URL of the newly created Service",
                "schema" : {
                  "type" : "string"
                }
              }
            },
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Service"
                }
              }
            }
          },
          "400" : {
            "$ref" : "#/components/responses/RespBadRequest"
          },
          "401" : {
            "$ref" : "#/components/responses/RespUnauthorized"
          },
          "403" : {
            "$ref" : "#/components/responses/RespForbidden"
          },
          "500" : {
            "$ref" : "#/components/responses/RespInternalServerError"
          }
        }
      }
    },
    "/{id}" : {
      "get" : {
        "tags" : [ "sc" ],
        "summary" : "Retrieves a `Service` object",
        "parameters" : [ {
          "name" : "id",
          "in" : "path",
          "description" : "ID of the `Service`",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "200" : {
            "description" : "Successful response",
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Service"
                }
              }
            }
          },
          "400" : {
            "$ref" : "#/components/responses/RespBadRequest"
          },
          "401" : {
            "$ref" : "#/components/responses/RespUnauthorized"
          },
          "403" : {
            "$ref" : "#/components/responses/RespForbidden"
          },
          "404" : {
            "$ref" : "#/components/responses/RespNotfound"
          },
          "500" : {
            "$ref" : "#/components/responses/RespInternalServerError"
          }
        }
      },
      "put" : {
        "tags" : [ "sc" ],
        "summary" : "Updates the existing `Service` or creates a new one (with the
provided ID)",
        "parameters" : [ {
          "name" : "id",
          "in" : "path",
          "description" : "ID of the `Service`",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "requestBody" : {
          "$ref" : "#/components/requestBodies/Service"
        },
        "responses" : {
          "200" : {
            "description" : "Service updated successfully",
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Service"
                }
              }
            }
          },
          "201" : {
            "description" : "A new service is created",
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/Service"
                }
              }
            }
          },
          "400" : {
            "$ref" : "#/components/responses/RespBadRequest"
          },
          "401" : {
            "$ref" : "#/components/responses/RespUnauthorized"
          },
          "403" : {
            "$ref" : "#/components/responses/RespForbidden"
          },
          "409" : {
            "$ref" : "#/components/responses/RespConflict"
          },
          "500" : {
            "$ref" : "#/components/responses/RespInternalServerError"
          }
        }
      },
      "delete" : {
        "tags" : [ "sc" ],
        "summary" : "Deletes the `Service`",
        "parameters" : [ {
          "name" : "id",
          "in" : "path",
          "description" : "ID of the `Service`",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        } ],
        "responses" : {
          "200" : {
            "description" : "Successful response"
          },
          "401" : {
            "$ref" : "#/components/responses/RespUnauthorized"
          },
          "403" : {
            "$ref" : "#/components/responses/RespForbidden"
          },
          "404" : {
            "$ref" : "#/components/responses/RespNotfound"
          },
          "500" : {
            "$ref" : "#/components/responses/RespInternalServerError"
          }
        }
      }
    },
    "/{jsonpath}/{operator}/{value}" : {
      "get" : {
        "tags" : [ "sc" ],
        "summary" : "Service filtering API",
        "description" : "The filtering API enables service filtering based on a given
path, operator, and value. Below are few examples:\n* Filter all services belonging to
PlatformX (convention for 'type' followed: <platform-name>.<service-type>):\n
`/type/prefix/PlatformX`\n* Filter all services that have MQTT API(s):\n
`/apis.protocol/equals/MQTT`\n* Filter all services based on address meta field:\n
`/meta.address/contains/Bonn`\n",
        "parameters" : [ {
          "name" : "jsonpath",
          "in" : "path",
          "description" : "The dot notation path to search for in service objects",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "operator",
          "in" : "path",
          "description" : "One of (equals, prefix, suffix, contains) string comparison
operators",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "name" : "value",
          "in" : "path",
          "description" : "The intended value, prefix, suffix, or substring identified
by the jsonpath",
          "required" : true,
          "schema" : {
            "type" : "string"
          }
        }, {
          "$ref" : "#/components/parameters/ParamPage"
        }, {
          "$ref" : "#/components/parameters/ParamPerPage"
        } ],
        "responses" : {
          "200" : {
            "description" : "Succcessful response",
            "content" : {
              "application/json" : {
                "schema" : {
                  "$ref" : "#/components/schemas/APIIndex"
                }
              }
            }
          },
          "401" : {
            "$ref" : "#/components/responses/RespUnauthorized"
          },
          "403" : {
            "$ref" : "#/components/responses/RespForbidden"
          },
          "500" : {
            "$ref" : "#/components/responses/RespInternalServerError"
          }
        }
      }
    }
  },
  "servers" : [ {
    "url" : "/"
  } ],
  "components" : {
    "parameters" : {
      "ParamPage" : {
        "name" : "page",
        "in" : "query",
        "description" : "Page number in the pagination",
        "required" : false,
        "schema" : {
          "type" : "number",
          "format" : "integer"
        }
      },
      "ParamPerPage" : {
        "name" : "per_page",
        "in" : "query",
        "description" : "Number of entries per page",
        "required" : false,
        "schema" : {
          "type" : "number",
          "format" : "integer"
        }
      }
    },
    "responses" : {
      "RespBadRequest" : {
        "description" : "Bad Request",
        "content" : {
          "application/json" : {
            "schema" : {
              "$ref" : "#/components/schemas/ErrorResponse"
            }
          }
        }
      },
      "RespUnauthorized" : {
        "description" : "Unauthorized",
        "content" : {
          "application/json" : {
            "schema" : {
              "$ref" : "#/components/schemas/ErrorResponse"
            }
          }
        }
      },
      "RespForbidden" : {
        "description" : "Forbidden",
        "content" : {
          "application/json" : {
            "schema" : {
              "$ref" : "#/components/schemas/ErrorResponse"
            }
          }
        }
      },
      "RespNotfound" : {
        "description" : "Not Found",
        "content" : {
          "application/json" : {
            "schema" : {
              "$ref" : "#/components/schemas/ErrorResponse"
            }
          }
        }
      },
      "RespConflict" : {
        "description" : "Conflict",
        "content" : {
          "application/json" : {
            "schema" : {
              "$ref" : "#/components/schemas/ErrorResponse"
            }
          }
        }
      },
      "RespInternalServerError" : {
        "description" : "Internal Server Error",
        "content" : {
          "application/json" : {
            "schema" : {
              "$ref" : "#/components/schemas/ErrorResponse"
            }
          }
        }
      }
    },
    "requestBodies" : {
      "Service" : {
        "content" : {
          "application/json" : {
            "schema" : {
              "$ref" : "#/components/schemas/Service"
            }
          }
        },
        "description" : "Service to be created",
        "required" : true
      }
    },
    "schemas" : {
      "Service" : {
        "title" : "Service",
        "type" : "object",
        "required" : [ "type" ],
        "properties" : {
          "id" : {
            "type" : "string",
            "readOnly" : true
          },
          "type" : {
            "type" : "string"
          },
          "title" : {
            "type" : "string"
          },
          "description" : {
            "type" : "string"
          },
          "meta" : {
            "type" : "object"
          },
          "apis" : {
            "type" : "array",
            "items" : {
              "type" : "object",
              "properties" : {
```

```
            "id" : {
              "type" : "string"
            },
            "title" : {
              "type" : "string"
            },
            "description" : {
              "type" : "string"
            },
            "protocol" : {
              "type" : "string"
            },
            "url" : {
              "type" : "string"
            },
            "spec" : {
              "type" : "object",
              "properties" : {
                "mediaType" : {
                  "type" : "string"
                },
                "url" : {
                  "type" : "string"
                },
                "schema" : {
                  "type" : "object"
                }
              }
            },
            "meta" : {
              "type" : "object"
            }
          }
        }
      },
      "doc" : {
        "type" : "string"
      },
      "ttl" : {
        "type" : "integer",
        "format" : "int64",
        "minimum": 1,
        "maximum": 2147483647
      },
      "createdAt" : {
        "type" : "string",
        "format" : "date-time",
        "readOnly" : true
      },
      "updatedAt" : {
        "type" : "string",
        "format" : "date-time",
        "readOnly" : true
      },
      "expiresAt" : {
        "type" : "string",
        "format" : "date-time",
        "readOnly" : true
```

```
          }
        }
      },
      "APIIndex" : {
        "type" : "object",
        "properties" : {
          "id" : {
            "type" : "string"
          },
          "description" : {
            "type" : "string"
          },
          "services" : {
            "type" : "array",
            "items" : {
              "$ref" : "#/components/schemas/Service"
            }
          },
          "page" : {
            "type" : "integer",
            "format" : "int64"
          },
          "per_page" : {
            "type" : "integer"
          },
          "total" : {
            "type" : "integer"
          }
        }
      },
      "ErrorResponse" : {
        "type" : "object",
        "properties" : {
          "code" : {
            "type" : "integer"
          },
          "message" : {
            "type" : "string"
          }
        }
      }
    }
  }
}
```

1.2. MQTT Service Registration/De-registration API

```
{
        "asyncapi": "2.0.0",
        "info": {
                "title": "Data Spine Service Registry's MQTT Service
Registration/Deregistration API",
                "version": "3.0.0",
                "description": "### Lifecycle management of services using MQTT: \n\n
* Service Registry (SR) also supports MQTT for service registration, updates and de-
registration. \n\n * Service registration/update 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. \n\n * The will message
```

```
of the registered service is used to de-register it from the SR. The will topic(s) are
also defined in the config file (see `commonWillTopics` and `willTopics`).",
                "license": {
                        "name": "Apache 2.0",
                        "url": "https://www.apache.org/licenses/LICENSE-2.0"
                }
        },
        "servers": {
                "ds-message-broker-dev": {
                        "url": "broker.smecluster.com:{port}",
                        "description": "This RabbitMQ broker at SMECluster's servers
is Data Spine's Message Broker for the development environment.",
                        "protocol": "mqtt",
                        "variables": {
                                "port": {
                                        "description": "Secure connection (TLS) is
available through port 8883. Currently MQTTs is not implemented",
                                        "default": "1883",
                                        "enum": [
                                                "1883",
                                                "8883"
                                        ]
                                }
                        }
                }
        },
        "channels": {
                "sr/v3/cud/reg/{serviceId}": {
                        "parameters": {
                                "serviceId": {
                                        "$ref": "#/components/parameters/serviceId"
                                }
                        },
                        "publish": {
                                "summary": "MQTT topic for service
registration/updates",
                                "description": "The Service Registry subscribes to
this topic in `commonRegTopics` for service registrations and updates with the default
qos of 1 as defined in the config file. \n\n Users can publish the service registration
object as payload to this topic with a custom service `{serviceId}` for registring a
service with that id. \n\n Example: \n\n `mosquitto_pub -h localhost -p 1883 -t
'sr/v3/cud/reg/id1' -f ./service_object.json`",
                                "message": {
                                        "payload": {
                                                "type": "object",
                                                "$ref":
"#/components/schemas/Service"
                                        }
                                }
                        }
                },
                "sr/v3/cud/dereg/{serviceId}": {
                        "parameters": {
                                "serviceId": {
                                        "$ref": "#/components/parameters/serviceId"
                                }
                        },
                        "publish": {
                                "summary": "MQTT topic for service de-registration",
                                "description": "The Service Registry subscribes to
this topic in `commonWillTopics` for service de-registrations with the default qos of 1
as defined in the config file. \n\n Users can publish any random message as payload to
this topic with the `{serviceId}` of the service to be de-registered. \n\n Example:
\n\n `mosquitto_pub -h localhost -p 1883 -t 'sr/v3/cud/dereg/id1' –m 'foobar'`",
                                "message": {
                                        "payload": {
                                                "type": "string"
                                        }
                                }
                        }
                }
        },
        "components": {
                "parameters": {
                        "serviceId": {
                                "description": "The ID of the service.",
                                "schema": {
                                        "type": "string"
                                }
                        }
                },
                "schemas": {
                        "Service": {
                                "title": "Service",
                                "type": "object",
                                "required": ["type"],
                                "properties": {
                                        "id": {
                                                "type": "string",
                                                "readOnly": true
                                        },
                                        "type": {
                                                "type": "string"
                                        },
                                        "title": {
                                                "type": "string"
                                        },
                                        "description": {
                                                "type": "string"
                                        },
                                        "meta": {
                                                "type": "object"
                                        },
                                        "apis": {
                                                "type": "array",
                                                "items": {
                                                        "type": "object",
                                                        "properties": {
                                                                "id": {
                                                                        "type":
"string"
                                                                },
                                                                "title": {
                                                                        "type":
"string"
                                                                },
                                                                        "description": {
                                                                                "type":
"string"
                                                                        },
                                                                        "protocol": {
                                                                                "type":
"string"
                                                                        },
                                                                        "url": {
                                                                                "type":
"string"
                                                                        },
                                                                        "spec": {
                                                                                "type":
"object",

        "properties": {

        "mediaType": {

        "type": "string"
                                                                                        },

        "url": {

        "type": "string"
                                                                                        },

        "schema": {

        "type": "object"
                                                                                        }
                                                                                }
                                                                        },
                                                                        "meta": {
                                                                                "type":
"object"
                                                                        }
                                                                }
                                                        }
                                                },
                                                "doc": {
                                                        "type": "string"
                                                },
                                                "ttl": {
                                                        "type": "integer",
                                                        "format": "int64",
                                                        "minimum": 1,
                                                        "maximum": 2147483647
                                                },
                                                "createdAt": {
                                                        "type": "string",
                                                        "format": "date-time",
                                                        "readOnly": true
                                                },
                                                "updatedAt": {
                                                        "type": "string",
                                                        "format": "date-time",
                                                        "readOnly": true
                                                },
                                                "expiresAt": {
                                                        "type": "string",
                                                        "format": "date-time",
                                                        "readOnly": true
                                                }
                                        }
                                }
                        }
        }
}
```

1.3. MQTT Announcement API

```
{
        "asyncapi": "2.0.0",
        "info": {
                "title": "Data Spine Service Registry's MQTT Service Announcement
API",
                "version": "3.0.0",
                "description": "### Service registration status announcements over
MQTT: \n\n * Service Registry announces the service registration and updates via MQTT
using retain messages. \n\n * Service Registry also announces the service de-
registration (on explicit de-registration request via REST/MQTT API or on expiration
due to the set TTL) via MQTT.",
                "license": {
                        "name": "Apache 2.0",
                        "url": "https://www.apache.org/licenses/LICENSE-2.0"
                }
        },
        "servers": {
                "ds-message-broker-dev": {
                        "url": "broker.smecluster.com:{port}",
                        "description": "This RabbitMQ broker at SMECluster's servers
is Data Spine's Message Broker for the development environment.",
                        "protocol": "mqtt",
                        "variables": {
                                "port": {
                                        "description": "Secure connection (TLS) is
available through port 8883. Currently MQTTs is not implemented",
                                        "default": "1883",
                                        "enum": [
                                                "1883",
                                                "8883"
                                        ]
                                }
                        }
                }
        },
        "channels": {
                "sr/v3/announcement/{serviceType}/{serviceId}/alive": {
                        "parameters": {
                                "serviceId": {
                                        "$ref": "#/components/parameters/serviceId"
                                },
                                "serviceType": {
                                        "$ref": "#/components/parameters/serviceType"
                                }
                        },
                        "subscribe": {
                                "summary": "MQTT topic for service registration/update
announcements",
                                "description": "The Service Registry publishes to this
topic when the service with ID `{serviceId}` and type {serviceType} is registered or
whenever it is updated. The service registration/update messages are retained. Default
qos used for the publish operation is 1. \n\n Users can subscribe to this topic to
monitor service registration/updates. \n\n Examples: \n\n * `mosquitto_sub -h localhost
-p 1883 -t 'sr/v3/announcement/efpf.marketplace-service/eb647488-a53b-4223-89ef-
63ae2ce826ae/alive'` \n\n * `mosquitto_sub -h localhost -p 1883 -t
'sr/v3/announcement/efpf.marketplace-service/+/alive'` \n\n * `mosquitto_sub -h
localhost -p 1883 -t 'sr/v3/announcement/+/+/alive'`",
                                "message": {
                                        "payload": {
                                                "type": "object",
                                                "$ref":
"#/components/schemas/Service"
                                        }
                                }
                        }
                },
                "sr/v3/announcement/{serviceType}/{serviceId}/dead": {
                        "parameters": {
                                "serviceId": {
                                        "$ref": "#/components/parameters/serviceId"
                                },
                                "serviceType": {
                                        "$ref": "#/components/parameters/serviceType"
                                }
                        },
                        "subscribe": {
                                "summary": "MQTT topic for service de-registration
announcements",
                                "description": "The Service Registry publishes to this
topic when the service with ID `{serviceId}` and type {serviceType} is de-registered.
The service de-registration messages are not retained. Default qos used for the publish
operation is 1. Upon de-registration of services, the associated retained messages of
service registration/updates (topic:
`sr/v3/announcement/{serviceType}/{serviceId}/alive`) are also removed. \n\n Users can
subscribe to this topic to get notified when services become unavailable i.e. when they
are removed from the Service Registry upon explicit de-registration or expiration. \n\n
Examples: \n\n * `mosquitto_sub -h localhost -p 1883 -t
'sr/v3/announcement/efpf.marketplace-service/eb647488-a53b-4223-89ef-
63ae2ce826ae/dead'` \n\n * `mosquitto_sub -h localhost -p 1883 -t
'sr/v3/announcement/efpf.marketplace-service/+/dead'` \n\n * `mosquitto_sub -h
localhost -p 1883 -t 'sr/v3/announcement/+/+/dead'`",
                                "message": {
                                        "payload": {
                                                "type": "string"
                                        }
                                }
                        }
                }
        },
        "components": {
                "parameters": {
                        "serviceId": {
```

```
                "description": "The ID of the service.",
                "schema": {
                    "type": "string"
                }
            },
            "serviceType": {
                "description": "The `type` of the service.",
                "schema": {
                    "type": "string"
                }
            }
        },
        "schemas": {
            "Service": {
                "title": "Service",
                "type": "object",
                "required": ["type"],
                "properties": {
                    "id": {
                        "type": "string",
                        "readOnly": true
                    },
                    "type": {
                        "type": "string"
                    },
                    "title": {
                        "type": "string"
                    },
                    "description": {
                        "type": "string"
                    },
                    "meta": {
                        "type": "object"
                    },
                    "apis": {
                        "type": "array",
                        "items": {
                            "type": "object",
                            "properties": {
                                "id": {
                                    "type":
"string"
                                },
                                "title": {
                                    "type":
"string"
                                },
                                "description": {
                                    "type":
"string"
                                },
                                "protocol": {
                                    "type":
"string"
                                },
                                "url": {
                                    "type":
"string"
```

```
                                                                },
                                                                "spec": {
                                                                        "type":
"object",

        "properties": {

        "mediaType": {

        "type": "string"
                                                                                },

        "url": {

        "type": "string"
                                                                                },

        "schema": {

        "type": "object"
                                                                                }
                                                                        }
                                                                },
                                                                "meta": {
                                                                        "type":
"object"
                                                                }
                                                        }
                                                }
                                        },
                                        "doc": {
                                                "type": "string"
                                        },
                                        "ttl": {
                                                "type": "integer",
                                                "format": "int64",
                                                "minimum": 1,
                                                "maximum": 2147483647
                                        },
                                        "createdAt": {
                                                "type": "string",
                                                "format": "date-time",
                                                "readOnly": true
                                        },
                                        "updatedAt": {
                                                "type": "string",
                                                "format": "date-time",
                                                "readOnly": true
                                        },
                                        "expiresAt": {
                                                "type": "string",
                                                "format": "date-time",
                                                "readOnly": true
                                        }
                                }
                        }
                }
        }
```

```
}
```

2. Matchmaker for Bidding Process - REST API

```
 {
  "openapi" : "3.0.1",
  "info" : {
    "title" : "Matchmaker-API services",
    "description" : "Matchmaker API for Automated Online Bidding",
    "version" : "0.1"
  },
  "servers" : [ {
    "url" : "https://inter.composition-ecosystem.eu"
  } ],
  "paths" : {
    "/matchmaker/COMPOSITION_RBMM_Restful_WS/RBMM/setMarketplaceService" : {
      "post" : {
        "description" : "Insert new services or products in marketplace",
        "requestBody" : {
          "content" : {
            "application/json" : {
              "schema" : {
                "type" : "object",
                "properties" : {
                  "insert" : {
                    "type" : "object",
                    "properties" : {
                      "details" : {
                        "type" : "object",
                        "properties" : {
                          "service" : {
                            "type" : "string"
                          },
                          "id" : {
                            "type" : "string"
                          },
                          "category" : {
                            "type" : "string"
                          },
                          "good" : {
                            "type" : "string"
                          }
                        }
                      },
                      "vars" : {
                        "type" : "array",
                        "items" : {
                          "type" : "string"
                        }
                      }
                    }
                  }
                }
              },
              "examples" : {
                "0" : {
                  "value" : "{\r\n\t\"insert\" :{\r\n\t\t\"vars\": [\"id\",
\"service\"] ,\r\n\t\t\"details\": {\r\n\t\t\t\"id\":
```

```
\"agentCompA@composition\",\r\n\t\t\t\"service\": \"Company_22_Service\",
\r\n\t\t\t\"good\": \"chair\", \r\n\t\t\t\t\"category\":
\"Chair_Manufacturing\"\r\n\t\t}\r\n\t\r\n\t}\r\n}\r\n"
                }
              }
            }
          }
        },
        "responses" : {
          "200" : {
            "description" : "Successful insert of new services or products in
marketplace",
            "content" : {
              "*/*" : {
                "schema" : {
                  "type" : "string"
                },
                "examples" : { }
              }
            }
          }
        },
        "servers" : [ {
          "url" : "https://inter.composition-ecosystem.eu"
        } ]
      },
      "servers" : [ {
        "url" : "https://inter.composition-ecosystem.eu"
      } ]
    },
    "/matchmaker/COMPOSITION_RBMM_Restful_WS/RBMM/getMarketplaceCompanies" : {
      "get" : {
        "description" : "Get Marketplace Companies and their information",
        "responses" : {
          "200" : {
            "description" : "Successful retrieval of Marketplace Companies"
          }
        },
        "servers" : [ {
          "url" : "https://inter.composition-ecosystem.eu"
        } ]
      },
      "servers" : [ {
        "url" : "https://inter.composition-ecosystem.eu"
      } ]
    },
    "/matchmaker/COMPOSITION_RBMM_Restful_WS/RBMM/getGoodsByCategory" : {
      "get" : {
        "description" : "Get Marketplace Goods by the Service Type they belong",
        "responses" : {
          "200" : {
            "description" : "Successful retrieval of Marketplace goods",
            "content" : {
              "application/json" : {
                "schema" : {
                  "type" : "object",
                  "properties" : { }
                },
                "examples" : {
                  "0" : {
                    "value" : "[\n  {\n    \"category\": \"Energy_supply\",\n
\"goods\": \"Biodiesel_fuel\"\n  },\n  {\n    \"category\": \"Waste_management\",\n
\"goods\": [\n      \"Wood_wastes\",\n      \" Plastic_wastes\",\n      \" Paper\",\n
\" Organic_wastes\",\n      \" Scrap_metal\",\n      \" Glass\"\n    ]\n  },\n  {\n
\"category\": \"Support_operation\",\n    \"goods\": [\n      \"Cabin\",\n      \"
Lifts\",\n      \" Elevators\",\n      \" Escalators\",\n      \" Car_Lifting\"\n
]\n  },\n  {\n    \"category\": \"Software_solutions\",\n    \"goods\": [\n
\"Visualization\",\n      \" Object_Locator\",\n      \" Object_Catalog\",\n      \"
Cloud\",\n      \" IoT\",\n      \" Analytics\",\n      \" Data_Storage\",\n      \"
Alarm_Manager\",\n      \" Event_Manager\",\n      \" Hardware\"\n    ]\n  },\n  {\n
\"category\": \"Lift_manufacturing\",\n    \"goods\": \"Lifts\"\n  }\n]"
                  }
                }
              }
            }
          }
        },
        "servers" : [ {
          "url" : "https://inter.composition-ecosystem.eu"
        } ]
      },
      "servers" : [ {
        "url" : "https://inter.composition-ecosystem.eu"
      } ]
    },
    "/matchmaker/COMPOSITION_RBMM_Restful_WS/RBMM/performMatchmaking" : {
      "post" : {
        "description" : "Rule based match of requester with suppliers",
        "requestBody" : {
          "content" : {
            "application/json" : {
              "schema" : {
                "type" : "object",
                "properties" : {
                  "offers" : {
                    "type" : "array",
                    "items" : {
                      "type" : "string"
                    }
                  },
                  "agent_owner" : {
                    "type" : "string"
                  },
                  "conversation_id" : {
                    "type" : "string"
                  },
                  "service" : {
                    "type" : "string"
                  },
                  "type" : {
                    "type" : "string"
                  },
                  "offer_details" : {
                    "type" : "object",
                    "properties" : {
                      "delivery_methods" : {
                        "type" : "array",
                        "items" : {
                          "type" : "string"
                        }
                      },
                      "payment_methods" : {
                        "type" : "array",
                        "items" : {
                          "type" : "string"
                        }
                      },
                      "quantity" : {
                        "type" : "number"
                      },
                      "quantity_uom" : {
                        "type" : "string"
                      },
                      "expiration" : {
                        "type" : "string"
                      },
                      "currency" : {
                        "type" : "string"
                      },
                      "good" : {
                        "type" : "string"
                      }
                    }
                  },
                  "sender_id" : {
                    "type" : "string"
                  }
                }
              },
              "examples" : {
                "0" : {
                  "value" : "{\r\n
\"conversation_id\":\"kjhfewKJDGWHJGWH7856186GBFWE\",\r\n
\"sender_id\":\"agent_req_1\",\r\n  \"agent_owner\": \"ELDIA\",\r\n
\"type\":\"CFP\",\r\n  \"service\":\"Software_solutions\",\r\n  \"offer_details\":{\r\n
\"good\":\"IoT\",\r\n    \"expiration\":\"2017-06-07T24:00:00+01:00\",\r\n
\"currency\":\"USD\",\r\n    \"quantity\":100.0,\r\n    \"quantity_uom\":\"q\",\r\n
\"delivery_methods\":  [\"DeliveryModeFreight\", \"DeliveryModePickup\"], \r\n
\"payment_methods\": [\"PayPal\", \"DirectDebit\", \"Discover\", \"Cash\"]\r\n  },\r\n
\"offers\":[]\r\n  \r\n}"
                }
              }
            }
          }
        },
        "responses" : {
          "200" : {
            "description" : "Successful retrieval of the match making results",
            "content" : {
              "*/*" : {
                "schema" : {
                  "type" : "string"
                },
                "examples" : { }
              }
            }
          }
        },
        "servers" : [ {
          "url" : "https://inter.composition-ecosystem.eu"
        } ]
      },
      "servers" : [ {
        "url" : "https://inter.composition-ecosystem.eu"
      } ]
    },
    "/matchmaker/COMPOSITION_RBMM_Restful_WS/RBMM/offersEvaluation" : {
      "post" : {
        "description" : "Rule-based and weighted criteria assessment of offers",
        "requestBody" : {
          "content" : {
            "application/json" : {
              "schema" : {
                "type" : "object",
                "properties" : {
                  "offers" : {
                    "type" : "array",
                    "items" : {
                      "type" : "object",
                      "properties" : {
                        "offer_details" : {
                          "type" : "object",
                          "properties" : {
                            "agent_owner" : {
                              "type" : "string"
                            },
                            "delivery" : {
                              "type" : "object",
                              "properties" : {
                                "methods" : {
                                  "type" : "array",
                                  "items" : {
                                    "type" : "string"
                                  }
                                },
                                "time" : {
                                  "type" : "integer"
                                }
                              }
                            },
                            "price" : {
                              "type" : "object",
                              "properties" : {
                                "insurance" : {
                                  "type" : "number"
                                },
                                "total" : {
                                  "type" : "number"
                                },
                                "service" : {
                                  "type" : "number"
                                },
              "transportation" : {
                "type" : "number"
              }
            }
          },
          "payment" : {
            "type" : "object",
            "properties" : {
              "terms" : {
                "type" : "integer"
              },
              "methods" : {
                "type" : "array",
                "items" : {
                  "type" : "string"
                }
              },
              "currency" : {
                "type" : "string"
              }
            }
          },
          "good" : {
            "type" : "string"
          },
          "sender_id" : {
            "type" : "string"
          }
        }
      }
    }
  }
},
"agent_owner" : {
  "type" : "string"
},
"conversation_id" : {
  "type" : "string"
},
"service" : {
  "type" : "string"
},
"criteria" : {
  "type" : "array",
  "items" : {
    "type" : "string"
  }
},
"type" : {
  "type" : "string"
},
"offer_details" : {
  "type" : "object",
  "properties" : {
    "delivery_methods" : {
      "type" : "array",
      "items" : {
        "type" : "string"
```

```
                        }
                      },
                      "payment_methods" : {
                        "type" : "array",
                        "items" : {
                          "type" : "string"
                        }
                      },
                      "quantity" : {
                        "type" : "number"
                      },
                      "quantity_uom" : {
                        "type" : "string"
                      },
                      "expiration" : {
                        "type" : "string"
                      },
                      "currency" : {
                        "type" : "string"
                      },
                      "good" : {
                        "type" : "string"
                      }
                    }
                  },
                  "sender_id" : {
                    "type" : "string"
                  }
                }
              },
              "examples" : {
                "0" : {
                  "value" : "
{\r\n\"conversation_id\":\"kjhfewKJDGWHJGWH7856186GBFWE\",\r\n
\"sender_id\":\"agent_req_1\",\r\n  \"agent_owner\": \"KLEEMANN\",\r\n
\"type\":\"OFFER\",\r\n  \"service\":\"Software_solutions\",\r\n\r\n
\"offer_details\":{\r\n    \"good\":\"IoT\",\r\n    \"expiration\":\"2017-06-
07T24:00:00+01:00\",\r\n    \"currency\":\"USD\",\r\n    \"quantity\":120.0,\r\n
\"quantity_uom\":\"q\",\r\n    \"delivery_methods\":  [\"DeliveyModeDirectDownload\",
\"DeliveryModeMail\"], \r\n    \"payment_methods\": [\"PayPal\", \"DirectDebit\",
\"Discover\", \"Cash\"]\r\n  },\r\n  \"criteria\": [\"rating\", \"payment_terms\",
\"delivery_time\", \"certification\", \"price\"],\r\n   \"offers\": [\r\n   {\r\n
\t\"offer_details\":{\r\n\t  \t\"sender_id\":\"agent_supplier_1\",\r\n\t
\t\"agent_owner\": \"Nextworks\",\r\n\t    \"good\":\"IoT\",\r\n\t    \"delivery\":
\r\n\t    \t{\r\n\t    \t\"time\":2,\r\n\t    \t\"methods\":
[\"DeliveyModeDirectDownload\", \"DeliveryModeMail\"]\r\n\t    \t},\r\n\t
\"payment\":\r\n\t    \t{\r\n\t    \t\"methods\": [\"PayPal\", \"DirectDebit\",
\"Discover\", \"Cash\"],\r\n\t    \t\"terms\": 60,\r\n\t    \t\"currency\": \"EUR\"
\r\n\t    \t},   \t\r\n\t    \"price\":\r\n\t    \t{\r\n\t    \t\"transportation\": 0.0
,\r\n\t    \t\"insurance\": 80.0, \r\n\t    \t\"service\": 142.0, \r\n\t
\t\"total\": 222.0\r\n    \t\t}\r\n    \t}\r\n    },\r\n    \r\n    {\r\n
\t\"offer_details\":{\r\n\t  \t\"sender_id\":\"agent_supplier_2\",\r\n\t
\t\"agent_owner\": \"CNET\",\r\n\t    \"good\":\"IoT\",\r\n\t    \"delivery\": \r\n\t
\t{\r\n\t    \t\"time\":1,\r\n\t    \t\"methods\": [\"DeliveyModeDirectDownload\",
\"DeliveryModeMail\"]\r\n\t    \t},\r\n\t    \"payment\":\r\n\t    \t{\r\n\t
\t\"methods\": [\"DirectDebit\", \"CheckInAdvance\", \"Cash\"],\r\n\t    \t\"terms\":
50,\r\n\t    \t\"currency\": \"EUR\" \r\n\t    \t},   \t\r\n\t    \"price\":\r\n\t
\t{\r\n\t    \t\"transportation\": 0.0 ,\r\n\t    \t\"insurance\": 60.0, \r\n\t
\t\"service\": 150.0, \r\n\t    \t\"total\": 210.0\r\n\t    \t\r\n\t    \t}\r\n    \r\n
\t}\r\n   }\r\n   ]\r\n}\r\n"
                }
              }
            }
          }
        },
        "responses" : {
          "200" : {
            "description" : "Return of best available offer",
            "content" : {
              "*/*" : {
                "schema" : {
                  "type" : "string"
                },
                "examples" : { }
              }
            }
          }
        },
        "servers" : [ {
          "url" : "https://inter.composition-ecosystem.eu"
        } ]
      },
      "servers" : [ {
        "url" : "https://inter.composition-ecosystem.eu"
      } ]
    },
    "/matchmaker/COMPOSITION_RBMM_Restful_WS/RBMM/getInfoFromOntology" : {
      "get" : {
        "description" : "Get available marketplace Companies and Services/Products",
        "responses" : {
          "200" : {
            "description" : "Successfully return of information"
          }
        },
        "servers" : [ {
          "url" : "https://inter.composition-ecosystem.eu"
        } ]
      },
      "servers" : [ {
        "url" : "https://inter.composition-ecosystem.eu"
      } ]
    },
    "/matchmaker/COMPOSITION_RBMM_Restful_WS/RBMM/getMarketplaceServices" : {
      "get" : {
        "description" : "Get available marketplace Services/Products",
        "responses" : {
          "200" : {
            "description" : "Successfully return of information"
          }
        },
        "servers" : [ {
          "url" : "https://inter.composition-ecosystem.eu"
        } ]
      },
      "servers" : [ {
        "url" : "https://inter.composition-ecosystem.eu"
      } ]
    },
    "/matchmaker/COMPOSITION_RBMM_Restful_WS/RBMM/deleteCompany" : {
      "get" : {
        "description" : "Delete company from marketplace by its id",
        "parameters" : [ {
          "name" : "id",
          "in" : "query",
          "schema" : {
            "type" : "string"
          },
          "example" : "agentCompA@composition"
        } ],
        "responses" : {
          "200" : {
            "description" : "Successful delete",
            "content" : {
              "application/json" : {
                "schema" : {
                  "type" : "object",
                  "properties" : {
                    "operation_info" : {
                      "type" : "string"
                    },
                    "description" : {
                      "type" : "string"
                    },
                    "status" : {
                      "type" : "integer"
                    }
                  }
                },
                "examples" : {
                  "0" : {
                    "value" : "{\n  \"status\": 1,\n  \"operation_info\":
\"successful\",\n  \"description\": \"Delete Company\"\n}"
                  }
                }
              }
            }
          }
        },
        "servers" : [ {
          "url" : "https://inter.composition-ecosystem.eu"
        } ]
      },
      "servers" : [ {
        "url" : "https://inter.composition-ecosystem.eu"
      } ]
    },
    "/matchmaker/COMPOSITION_RBMM_Restful_WS/RBMM/setMarketplaceCompany" : {
      "post" : {
        "description" : "Insert new company to the Marketplace",
        "requestBody" : {
          "content" : {
            "application/json" : {
              "schema" : {
                "type" : "object",
                "properties" : {
                  "insert" : {
                    "type" : "object",
                    "properties" : {
                      "details" : {
                        "type" : "object",
                        "properties" : {
                          "agent_owner" : {
                            "type" : "string"
                          },
                          "service" : {
                            "type" : "object",
                            "properties" : {
                              "name" : {
                                "type" : "string"
                              },
                              "category" : {
                                "type" : "string"
                              },
                              "good" : {
                                "type" : "string"
                              }
                            }
                          },
                          "business_type" : {
                            "type" : "string"
                          },
                          "rating" : {
                            "type" : "string"
                          },
                          "description" : {
                            "type" : "string"
                          },
                          "location" : {
                            "type" : "object",
                            "properties" : {
                              "number" : {
                                "type" : "string"
                              },
                              "country" : {
                                "type" : "string"
                              },
                              "code" : {
                                "type" : "integer"
                              },
                              "city" : {
                                "type" : "string"
                              },
                              "street" : {
                                "type" : "string"
                              },
                              "state" : {
                                "type" : "string"
                              }
                            }
                          },
                          "legal_name" : {
                            "type" : "string"
                          },
                          "sender_id" : {
                            "type" : "string"
                          }
                        }
                      },
                      "vars" : {
                        "type" : "array",
                        "items" : {
                          "type" : "string"
                        }
                      }
                    }
                  }
                }
              },
              "examples" : {
                "0" : {
                  "value" : "{\r\n\t\"insert\" :{\r\n\t\t\"vars\": [\"agent_owner\",
\"sender_id\", \"rating\", \"location\", \"legal_name\", \"description\",
\"business_type\", \"service\"] ,\r\n\t\t\"details\": {\r\n\t\t\t\"agent_owner\":
\"Company_A\",\r\n\t\t\t\"sender_id\": \"agentCompA@composition\",\r\n\t\t\t\"rating\":
\"3\",\r\n\t\t\t\"location\": {\r\n\t\t\t\t\"street\":\"Street\",
\r\n\t\t\t\t\"number\": \"1\", \r\n\t\t\t\t\"city\": \"City\",\r\n\t\t\t\t\"state\":
\"State\", \r\n\t\t\t\t\"code\": 90000,
\r\n\t\t\t\t\"country\":\"Greece\"\r\n\t\t\t\t\t}, \r\n\t\t\t\"legal_name\": \"CompA
SA\", \r\n\t\t\t\"description\": \"My description\", \r\n\t\t\t\"business_type\":
\"Manufacturer\",\r\n\t\t\t\"service\":
{\r\n\t\t\t\t\"name\":\"CompanyA_chairService\",
\r\n\t\t\t\t\"category\":\"Chair_manufacturing\",\r\n\t\t\t\t\"good\": \"Chair\"
\r\n\t\t\t}\t\t\r\n\t\t\t\r\n\t\t}\r\n\t\r\n\t}\r\n}\r\n"
                }
              }
            }
          }
        },
        "responses" : {
          "200" : {
            "description" : "Successfully added new company",
            "content" : {
              "*/*" : {
                "schema" : {
                  "type" : "string"
                },
                "examples" : { }
              }
            }
          }
        },
        "servers" : [ {
          "url" : "https://inter.composition-ecosystem.eu"
        } ]
      },
      "servers" : [ {
        "url" : "https://inter.composition-ecosystem.eu"
      } ]
    },
    "/matchmaker/COMPOSITION_RBMM_Restful_WS/RBMM/getServicesFromCompany" : {
      "get" : {
        "description" : "Get Services and Products of a Company by the company's id",
        "parameters" : [ {
          "name" : "id",
          "in" : "query",
          "schema" : {
            "type" : "string"
          },
          "example" : "agentKLE@composition"
        } ],
        "responses" : {
          "200" : {
            "description" : "Successful return of Company's services and products",
            "content" : {
              "application/json" : {
                "schema" : {
                  "type" : "object",
                  "properties" : { }
                },
                "examples" : {
                  "0" : {
                    "value" : "[\n  {\n    \"service\": \"Lift_Manufacture\",\n
\"description\": \"KLEEMANN offers complete traction and hydraulic lift systems, both
with and without machine room, from initial design through to development and
production.\",\n    \"service id\": \"15-40 days\",\n    \"warranty period\": \"2
months\",\n    \"transportation mode\": \"DeliveryModeFreight\"\n  },\n  {\n
\"service\": \"Lift_Manufacture\",\n    \"description\": \"KLEEMANN offers complete
traction and hydraulic lift systems, both with and without machine room, from initial
design through to development and production.\",\n    \"service id\":
\"lift_manufacturing@KLE\",\n    \"warranty period\": \"2 months\",\n
\"transportation mode\": \"DeliveryModeFreight\"\n  },\n  {\n    \"service\":
\"24/7_service/support\",\n    \"description\": \"To ensure that designing, ordering,
receiving and maintaining these products is as stress-free and effortless as possible,
KLEEMANN presents dedicated service teams that offer additional support and guidance at
every stage\",\n    \"service id\": \"support@KLE\",\n    \"warranty period\": \"1
month\"\n  },\n  {\n    \"description\": \"KLEEMANN offers pre-engineered or fully-
customised elevators, to perfectly fit in each and every project. KLEEMANN responds
reliably, quickly, precisely, and cost effectively to any specifications, as result of
an acquired experience on demanding projects.\",\n    \"service id\": \"fully-custom-
elevators@KLE\",\n    \"warranty period\": \"2 months\",\n    \"delivery time\": \"15-
30 days\",\n    \"transportation mode\": [\n      \"DeliveryModeFreight\",\n      \"
DeliveryModeOwnFleet\"\n    ],\n    \"product\": \"Fully-customized_Elevators\"\n  },\n
{\n    \"description\": \"Pick from futuristic, modern, classic or panoramic lifts, and
create your ideal design with doors and cabin parts including ceiling, floor, walls,
handrail, mirrors and operation panels\",\n    \"service id\": \"Cabins\",\n
\"warranty period\": \"1 month\",\n    \"delivery time\": \"10-25 days\",\n
\"transportation mode\": [\n      \"DeliveryModeFreight\",\n      \"
DeliveryModeOwnFleet\"\n    ],\n    \"product\": \"Cabins\"\n  },\n  {\n
\"description\": \"KLEEMANN Parking Systems offer a diverse range of vertical stacking
and sliding platform solutions. Building on technological advancements, our intelligent
multi-storey Parking Systems guarantee exceptional performance, passenger comfort and
safety.\",\n    \"service id\": \"carLifting@KLE\",\n    \"warranty period\": \"2
months\",\n    \"delivery time\": \"20-30 days\",\n    \"transportation mode\": [\n
\"DeliveryModeFreight\",\n      \" DeliveryModeOwnFleet\"\n    ],\n    \"product\":
\"Car_Lifting_Systems\"\n  },\n  {\n    \"description\": \"KLEEMANN Escalators and
Moving Walks are versatile and suitable for use in all kinds of buildings, with tailor-
made design alternatives that offer high rise, outdoor, heavy duty, high capacity and
eco-friendly options for special projects. \",\n    \"service id\": \"escalators-
```

```
movingWalks@KLE\",\n    \"warranty period\": \"1 month\",\n    \"delivery time\": \"15-
30 days\",\n    \"transportation mode\": [\n      \"DeliveryModeFreight\",\n      \"
DeliveryModeOwnFleet\"\n    ],\n    \"product\": \"Escalators_and_Moving_Walks\"\n
},\n  {\n    \"description\": \"KLEEMANN lifts are designed to enhance the quality of
commercial domain and the functionality of all types of industrial establishment.Our
wealth of technical know-how ensures high technological performance, reliability and
safety.\",\n    \"service id\": \"elevators-lifts@KLE\",\n    \"warranty period\": \"2
months\",\n    \"delivery time\": \"15-30 days\",\n    \"transportation mode\": [\n
\"DeliveryModeFreight\",\n      \" DeliveryModeOwnFleet\"\n    ],\n    \"product\":
\"Elevators_and_Lifts\"\n  },\n  {\n    \"description\": \"Pick from futuristic,
modern, classic or panoramic lifts, and create your ideal design with doors and cabin
parts including ceiling, floor, walls, handrail, mirrors and operation panels\",\n
\"service id\": \"cabins@KLE\",\n    \"warranty period\": \"1 month\",\n    \"delivery
time\": \"10-25 days\",\n    \"transportation mode\": [\n
\"DeliveryModeFreight\",\n      \" DeliveryModeOwnFleet\"\n    ],\n    \"product\":
\"Cabins\"\n  }\n]"
                  }
                }
              }
            }
          }
        },
        "servers" : [ {
          "url" : "https://inter.composition-ecosystem.eu"
        } ]
      },
      "servers" : [ {
        "url" : "https://inter.composition-ecosystem.eu"
      } ]
    },
    "/matchmaker/COMPOSITION_RBMM_Restful_WS/RBMM/getCompanyDetails" : {
      "get" : {
        "description" : "Get information of marketplace company by giving its id",
        "parameters" : [ {
          "name" : "id",
          "in" : "query",
          "schema" : {
            "type" : "string"
          },
          "example" : "agentCNET@composition"
        } ],
        "responses" : {
          "200" : {
            "description" : "Successful return of company's information",
            "content" : {
              "application/json" : {
                "schema" : {
                  "type" : "object",
                  "properties" : { }
                },
                "examples" : {
                  "0" : {
                    "value" : "[\n  {\n    \"company brand name\": \"CNET\",\n
\"trust-score\": \"5\",\n    \"company legal name\": \"CNet Svenska AB\",\n
\"description\": \"CNet is an Internet of Things company developing innovative products
and services for a wide range of business and consumer applications utilizing the
latest IoT, Big Data and Cloud technologies to create business improvements. \",\n
\"location\": {\n      \"street\": \"Svärdvägen\",\n      \"number\": \"3A\",\n
\"state\": \"Danderyd\",\n      \"code\": \"18233\",\n      \"city\": \"Danderyd\"\n
},\n    \"origin country\": \"Sweden\",\n    \"business type\": \"Service_Provider\",\n
\"website\": \"https://www.cnet.se/\",\n    \"logo\":
\"http://efpf.iti.gr/public/logos/cnet.png\",\n    \"services\": [\n
\"Cloud_Services\",\n      \" IoT_Connectivity_Services\",\n      \" IoT_Analytics\"\n
],\n    \"activity sector\": \"Software_solutions\"\n  }\n]"
                  }
                }
              }
            }
          }
        },
        "servers" : [ {
          "url" : "https://inter.composition-ecosystem.eu"
        } ]
      },
      "servers" : [ {
        "url" : "https://inter.composition-ecosystem.eu"
      } ]
    }
  },
  "components" : {
    "securitySchemes" : {
      "basic" : {
        "type" : "http",
        "scheme" : "basic"
      }
    }
  }
}
```

3. Portal: Backend API

```
{
        "openapi": "3.0.1",
        "info": {
                "title": "PortalBackend",
                "description": "Portal Backend service as part of T5.2 EFPF Portal",
                "version": "v1"
        },
        "paths": {
                "/v1/Admin": {
                        "get": {
                                "tags": [
                                        "Admin"
                                ],
                                "responses": {
                                        "200": {
                                                "description": "Success"
                                        },
                                        "401": {
                                                "description": "Unauthorized",
                                                "content": {
                                                        "text/plain": {
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/ProblemDetails"
                                                                }
                                                        },
                                                        "application/json": {
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/ProblemDetails"
                                                                }
                                                        },
                                                        "text/json": {
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/ProblemDetails"
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
                },
                "/v1/Event": {
                        "post": {
                                "tags": [
                                        "Event"
                                ],
                                "requestBody": {
                                        "content": {
                                                "application/json-patch+json": {
                                                        "schema": {
                                                                "$ref":
"#/components/schemas/Event"
                                                        }
                                                },
                                                "application/json": {
```

```
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/Event"
                                                                }
                                                        },
                                                        "text/json": {
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/Event"
                                                                }
                                                        },
                                                        "application/*+json": {
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/Event"
                                                                }
                                                        }
                                                }
                                        },
                                        "responses": {
                                                "200": {
                                                        "description": "Success"
                                                },
                                                "400": {
                                                        "description": "Bad Request",
                                                        "content": {
                                                                "text/plain": {
                                                                        "schema": {
                                                                                "$ref":
"#/components/schemas/ProblemDetails"
                                                                        }
                                                                },
                                                                "application/json": {
                                                                        "schema": {
                                                                                "$ref":
"#/components/schemas/ProblemDetails"
                                                                        }
                                                                },
                                                                "text/json": {
                                                                        "schema": {
                                                                                "$ref":
"#/components/schemas/ProblemDetails"
                                                                        }
                                                                }
                                                        }
                                                },
                                                "500": {
                                                        "description": "Server Error"
                                                }
                                        }
                                }
                        },
                        "/v1/User": {
                                "post": {
                                        "tags": [
                                                "User"
                                        ],
                                        "requestBody": {
```

```
					"content": {
						"application/json-patch+json": {
							"schema": {
								"$ref": 
"#/components/schemas/UserRegistration"
							}
						},
						"application/json": {
							"schema": {
								"$ref": 
"#/components/schemas/UserRegistration"
							}
						},
						"text/json": {
							"schema": {
								"$ref": 
"#/components/schemas/UserRegistration"
							}
						},
						"application/*+json": {
							"schema": {
								"$ref": 
"#/components/schemas/UserRegistration"
							}
						}
					}
				},
				"responses": {
					"201": {
						"description": "Success"
					},
					"400": {
						"description": "Bad Request",
						"content": {
							"text/plain": {
								"schema": {
									"$ref": 
"#/components/schemas/ProblemDetails"
								}
							},
							"application/json": {
								"schema": {
									"$ref": 
"#/components/schemas/ProblemDetails"
								}
							},
							"text/json": {
								"schema": {
									"$ref": 
"#/components/schemas/ProblemDetails"
								}
							}
						}
					},
					"403": {
						"description": "Forbidden",
						"content": {
							"text/plain": {
```

```
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/ProblemDetails"
                                                                }
                                                        },
                                                        "application/json": {
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/ProblemDetails"
                                                                }
                                                        },
                                                        "text/json": {
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/ProblemDetails"
                                                                }
                                                        }
                                                }
                                        },
                                        "409": {
                                                "description": "Conflict",
                                                "content": {
                                                        "text/plain": {
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/ProblemDetails"
                                                                }
                                                        },
                                                        "application/json": {
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/ProblemDetails"
                                                                }
                                                        },
                                                        "text/json": {
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/ProblemDetails"
                                                                }
                                                        }
                                                }
                                        },
                                        "412": {
                                                "description": "Client Error",
                                                "content": {
                                                        "text/plain": {
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/ProblemDetails"
                                                                }
                                                        },
                                                        "application/json": {
                                                                "schema": {
                                                                        "$ref":
"#/components/schemas/ProblemDetails"
                                                                }
                                                        },
                                                        "text/json": {
```

```
                                                "schema": {
                                                        "$ref":
"#/components/schemas/ProblemDetails"
                                                }
                                        }
                                }
                        },
                        "500": {
                                "description": "Server Error"
                        }
                }
        },
        "put": {
                "tags": [
                        "User"
                ],
                "requestBody": {
                        "content": {
                                "application/json-patch+json": {
                                        "schema": {
                                                "$ref":
"#/components/schemas/UserUpdate"
                                        }
                                },
                                "application/json": {
                                        "schema": {
                                                "$ref":
"#/components/schemas/UserUpdate"
                                        }
                                },
                                "text/json": {
                                        "schema": {
                                                "$ref":
"#/components/schemas/UserUpdate"
                                        }
                                },
                                "application/*+json": {
                                        "schema": {
                                                "$ref":
"#/components/schemas/UserUpdate"
                                        }
                                }
                        }
                },
                "responses": {
                        "200": {
                                "description": "Success"
                        },
                        "400": {
                                "description": "Bad Request",
                                "content": {
                                        "text/plain": {
                                                "schema": {
                                                        "$ref":
"#/components/schemas/ProblemDetails"
                                                }
                                        },
                                        "application/json": {
```

```
                                                                        "schema": {
                                                                                "$ref":
"#/components/schemas/ProblemDetails"
                                                                        }
                                                                },
                                                                "text/json": {
                                                                        "schema": {
                                                                                "$ref":
"#/components/schemas/ProblemDetails"
                                                                        }
                                                                }
                                                        }
                                                },
                                                "403": {
                                                        "description": "Forbidden",
                                                        "content": {
                                                                "text/plain": {
                                                                        "schema": {
                                                                                "$ref":
"#/components/schemas/ProblemDetails"
                                                                        }
                                                                },
                                                                "application/json": {
                                                                        "schema": {
                                                                                "$ref":
"#/components/schemas/ProblemDetails"
                                                                        }
                                                                },
                                                                "text/json": {
                                                                        "schema": {
                                                                                "$ref":
"#/components/schemas/ProblemDetails"
                                                                        }
                                                                }
                                                        }
                                                },
                                                "409": {
                                                        "description": "Conflict",
                                                        "content": {
                                                                "text/plain": {
                                                                        "schema": {
                                                                                "$ref":
"#/components/schemas/ProblemDetails"
                                                                        }
                                                                },
                                                                "application/json": {
                                                                        "schema": {
                                                                                "$ref":
"#/components/schemas/ProblemDetails"
                                                                        }
                                                                },
                                                                "text/json": {
                                                                        "schema": {
                                                                                "$ref":
"#/components/schemas/ProblemDetails"
                                                                        }
                                                                }
                                                        }
```

```
                                        },
                                        "500": {
                                                "description": "Server Error"
                                        }
                                }
                        },
                        "get": {
                                "tags": [
                                        "User"
                                ],
                                "parameters": [
                                        {
                                                "name": "email",
                                                "in": "query",
                                                "schema": {
                                                        "type": "string"
                                                }
                                        }
                                ],
                                "responses": {
                                        "201": {
                                                "description": "Success"
                                        }
                                }
                        }
                }
        },
        "components": {
                "schemas": {
                        "ProblemDetails": {
                                "type": "object",
                                "properties": {
                                        "Type": {
                                                "type": "string",
                                                "nullable": true
                                        },
                                        "Title": {
                                                "type": "string",
                                                "nullable": true
                                        },
                                        "Status": {
                                                "type": "integer",
                                                "format": "int32",
                                                "nullable": true
                                        },
                                        "Detail": {
                                                "type": "string",
                                                "nullable": true
                                        },
                                        "Instance": {
                                                "type": "string",
                                                "nullable": true
                                        },
                                        "Extensions": {
                                                "type": "object",
                                                "additionalProperties": {
                                                        "type": "object",
                                                        "additionalProperties": false
                        },
                        "nullable": true,
                        "readOnly": true
                }
        },
        "additionalProperties": false
},
"Event": {
        "type": "object",
        "properties": {
                "UserId": {
                        "type": "string",
                        "nullable": true
                },
                "Action": {
                        "type": "string",
                        "nullable": true
                },
                "Platform": {
                        "type": "string",
                        "nullable": true
                },
                "Timestamp": {
                        "type": "string",
                        "nullable": true
                },
                "VisitedPlatform": {
                        "type": "string",
                        "nullable": true
                },
                "VisitedTool": {
                        "type": "string",
                        "nullable": true
                },
                "Query": {
                        "type": "string",
                        "nullable": true
                },
                "FacetQuery": {
                        "type": "array",
                        "items": {
                                "type": "string"
                        },
                        "nullable": true
                },
                "QueriedPlatforms": {
                        "type": "string",
                        "nullable": true
                },
                "SearchType": {
                        "type": "string",
                        "nullable": true
                },
                "SearchResponse": {
                        "type": "array",
                        "items": {
                                "type": "string"
                        },
                                "nullable": true
                        }
                },
                "additionalProperties": false
        },
        "User": {
                "type": "object",
                "properties": {
                        "Password": {
                                "type": "string",
                                "nullable": true
                        },
                        "FirstName": {
                                "type": "string",
                                "nullable": true
                        },
                        "LastName": {
                                "type": "string",
                                "nullable": true
                        },
                        "Company": {
                                "type": "string",
                                "nullable": true
                        },
                        "Vatin": {
                                "type": "string",
                                "nullable": true
                        },
                        "Street": {
                                "type": "string",
                                "nullable": true
                        },
                        "PostalCode": {
                                "type": "string",
                                "nullable": true
                        },
                        "City": {
                                "type": "string",
                                "nullable": true
                        },
                        "Country": {
                                "type": "string",
                                "nullable": true
                        },
                        "Sector": {
                                "type": "string",
                                "nullable": true
                        },
                        "Email": {
                                "type": "string",
                                "nullable": true
                        }
                },
                "additionalProperties": false
        },
        "UserRegistration": {
                "type": "object",
                "properties": {
```

```
                                        "User": {
                                                "allOf": [
                                                        {
                                                                "$ref": 
"#/components/schemas/User"
                                                        }
                                                ],
                                                "nullable": true
                                        },
                                        "AcceptedToC": {
                                                "type": "boolean"
                                        },
                                        "AcceptedBlockChain": {
                                                "type": "boolean"
                                        }
                                },
                                "additionalProperties": false
                        },
                        "UserUpdate": {
                                "type": "object",
                                "properties": {
                                        "User": {
                                                "allOf": [
                                                        {
                                                                "$ref": 
"#/components/schemas/User"
                                                        }
                                                ],
                                                "nullable": true
                                        },
                                        "ID": {
                                                "type": "string",
                                                "nullable": true
                                        }
                                },
                                "additionalProperties": false
                        }
                }
        }
}
```

4. Risk Tool [ALM]

4.1. REST API for Recipe and Workflow management and usage

```
 {
  "swagger": "2.0",
  "basePath": "/api/v1",
  "paths": {
        "/recipes/": {
        "post": {
        "responses": {
        "200": {
        "description": "Success",
        "schema": {
                "$ref": "#/definitions/Recipe"
        }
        }
        },
"summary": "Add a recipe",
"operationId": "post_recipe",
"parameters": [
{
"name": "payload",
"required": true,
"in": "body",
"schema": {
        "$ref": "#/definitions/Recipe"
}
}
],
"tags": [
"recipes"
]
},
"get": {
"responses": {
"200": {
"description": "Success",
"schema": {
        "$ref": "#/definitions/Recipe"
}
}
},
"summary": "List all recipes",
"operationId": "list_recipes",
"tags": [
"recipes"
]
}
},
"/recipes/{rid}": {
"parameters": [
{
"name": "rid",
"in": "path",
"required": true,
"type": "string"
}
],
"get": {
"responses": {
"200": {
"description": "Success",
"schema": {
        "$ref": "#/definitions/Recipe"
}
}
},
"summary": "Retrieve single recipe",
"operationId": "list_recipes",
"tags": [
"recipes"
]
},
"delete": {
"responses": {
```

```
"204": {
"description": "Recipe deleted"
},
"404": {
"description": "Not found"
}
},
"summary": "Delete a recipe",
"operationId": "delete_recipe",
"tags": [
"recipes"
]
},
"patch": {
"responses": {
"200": {
"description": "Success",
"schema": {
        "$ref": "#/definitions/Recipe"
}
}
},
"summary": "Update a recipe",
"operationId": "patch_recipe",
"parameters": [
{
"name": "payload",
"required": true,
"in": "body",
"schema": {
        "$ref": "#/definitions/Recipe"
}
}
],
"tags": [
"recipes"
]
}
},
"/workflows/": {
"post": {
"responses": {
"200": {
"description": "Success",
"schema": {
        "$ref": "#/definitions/Workflow"
}
}
},
"summary": "Add a workflow",
"operationId": "post_workflow",
"parameters": [
{
"name": "payload",
"required": true,
"in": "body",
"schema": {
        "$ref": "#/definitions/Workflow"
}
}
],
"tags": [
"workflows"
]
},
"get": {
"responses": {
"200": {
"description": "Success",
"schema": {
        "$ref": "#/definitions/Workflow"
}
}
},
"summary": "List all workflows",
"operationId": "list_workflows",
"tags": [
"workflows"
]
}
},
"/workflows/{rid}": {
"parameters": [
{
"name": "rid",
"in": "path",
"required": true,
"type": "string"
}
],
"get": {
"responses": {
"200": {
"description": "Success",
"schema": {
        "$ref": "#/definitions/Workflow"
}
}
},
"summary": "Retrieve single workflow",
"operationId": "list_workflows",
"tags": [
"workflows"
]
},
"delete": {
"responses": {
"204": {
"description": "Workflow deleted"
},
"404": {
"description": "Not found"
}
},
"summary": "Delete a workflow",
"operationId": "delete_workflow",
        "tags": [
        "workflows"
        ]
        },
        "patch": {
        "responses": {
        "200": {
        "description": "Success",
        "schema": {
                "$ref": "#/definitions/Workflow"
        }
        }
        },
        "summary": "Update a workflow",
        "operationId": "patch_workflow",
        "parameters": [
        {
        "name": "payload",
        "required": true,
        "in": "body",
        "schema": {
                "$ref": "#/definitions/Workflow"
        }
        }
        ],
        "tags": [
        "workflows"
        ]
        }
        }
  },
  "info": {
        "title": "Risk Analysis Tool",
        "version": "1.0",
        "description": "A tool for risk analysis in the manufacturing domain."
  },
  "produces": [
        "application/json"
  ],
  "consumes": [
        "application/json"
  ],
  "tags": [
        {
        "name": "recipes",
        "description": "Recipe related operations"
        },
        {
        "name": "workflows",
        "description": "Workflow related operations"
        }
  ],
  "definitions": {
        "Recipe": {
        "description": "Module which computes output given a specific input",
        "properties": {
        "_id": {
        "type": "string"
```

```
         },
         "name": {
         "type": "string"
         },
         "description": {
         "type": "string"
         }
         },
         "type": "object"
         },
         "Workflow": {
         "description": "TODO",
         "properties": {
         "_id": {
         "type": "string"
         }
         },
         "type": "object"
         }
  },
  "responses": {
         "ParseError": {
         "description": "When a mask can't be parsed"
         },
         "MaskError": {
         "description": "When any error occurs on mask"
         }
  }
}
```

4.2. MQTT Subscription, Unsubscription and Publishing API

```
 {
  "asyncapi": "2.0.0",
  "info": {"title": "Endpoints of MQTT functionality for the ALM Risk Management Tool",
      "version": "0.1"
  },
  "servers": {
      "development": {
          "url": "broker.smecluster.com",
          "protocol": "mqtt",
          "description": "This is a placeholder for the final EFPF Dataspine message
bus.",
          "security": []
    }
  },
  "channels": {
      "efpf/{topic}": {
          "publish": {
              "message": {
                  "payload": {
                      "type": "array",
                      "examples": [
                          [
                              {"timestamp": 1591093430, "machineId": 0,
"averageCycleTime": 30, "currentQuantity": 50, "targetQuantity": 100},
                              {"timestamp": 1591093430, "machineId": 1,
"averageCycleTime": 27, "currentQuantity": 42, "targetQuantity": 100}
                          ]
                      ],
                      "description": "Raw input data, which is an array that may
consist of multiple elements, on which the recipe is to be run simultaneously."
                  }
              },
              "description": "A user or factory connector can stream data to this
endpoint and the tool will read it. {topic} assumes the workflow input topic."
          }
      },
      "efpf/risk/{user}/{id}/{output}": {
          "subscribe": {
              "message": {
                  "payload": {
                      "type": "array",
                      "examples": [
                          [
                              {"timestamp": 1591093430, "machineId": 0,
"averageCycleTime": 30, "currentQuantity": 50, "targetQuantity": 100,
"extrapolatedEndTime": 1591094930},
                              {"timestamp": 1591093430, "machineId": 1,
"averageCycleTime": 33, "currentQuantity": 42, "targetQuantity": 100,
"extrapolatedEndTime": 1591095344}
                          ]
                      ],
                      "description": "Returns the input data, optionally with
additional output data. For instance, calculates extrapolated end time of production
with current production rates and amount of required products."
                  }
              }
          },
          "description": "The risk tool transforms the input data into output and
streams it to this endpoint. {id} assumes the workflow id and {output} assumes the
workflow output topic. {user} should be a string specific to the user receiving this
data."
      },
      "efpf/risk/{user}/{id}/notification": {
          "subscribe": {
              "message": {
                  "payload": {
                      "type": "array",
                      "examples": [
                          "The extrapolated finish time for machine 1 is more than 6
minutes too late."
                      ],
                      "description": "For instance, if the threshold for 'much too
late' is 6 minutes, a notification can be sent if the machine is expected to be 6
minutes too late."
                  }
              },
              "description": "The risk tool eventually outputs a notification too if
there's a risk. {id} assumes the workflow id. {user} should be a string specific to the
user receiving this data."
          }
      },
      "efpf/risk/{user}/{id}/error": {
          "subscribe": {
              "message": {
                  "payload": {
                      "type": "array",
                      "examples": [
                          "In update: index 100 is out of bounds for axis 1 with size
100"
                      ],
                      "description": "Returns the error received in the backend."
                  }
              },
              "description": "If there is an error for any reason, the risk tool will
publish it to this topic. {id} assumes the workflow id. {user} should be a string
specific to the user receiving this data."
          }
      }
  }
}
```

European Factory Platform

www.efpf.org