D2.3 - Integrated FAME Data Marketplace I

Title	D2.3 - Integrated FAME Data Marketplace I
Revision Number	1.0
Task reference	T2.3; T2.4
Lead Beneficiary	GFT
Responsible	Antonio Sottosanti
Partners	ATOS, ECO, ENG, IDSA, JOT, KM, LXS, MOH, NOVA, NOVO, UNP, UPRC
Deliverable Type	DEM
Dissemination Level	PU
Due Date	2024-06-30 [Month 18]
Delivered Date	2024-08-09
Internal Reviewers	GFT UPRC
Quality Assurance	UPRC
Acceptance	Coordinator Accepted
Project Title	FAME - Federated decentralized trusted dAta Marketplace for Embedded finance
Grant Agreement No.	101092639
EC Project Officer	Stefano Bertolo
Programme	HORIZON-CL4-2022-DATA-01-04

This project has received funding from the European Union's Horizon research and innovation programme under Grant Agreement no 101092639

Revision History

Version	Date	Partners	Description
0.1	2023-03-02	GFT	Initial Table of Content
0.2	2024-05-02	GFT, UPRC	UPRC Contribution
0.3	2024-05-29	GFT	Added chapters 5+6
0.4	2024-06-09	GFT, UPRC, ENG, ATOS	Contributions
0.5	2024-07-08	GFT, UPRC, ENG	front end and user stories updates
0.6	2024-08-06	GFT, UPRC, ENG	finalizing doc
0.9	2024-08-06	GFT, UPRC, ENG	Version for peer review
1.0	2024-08-09	GFT	Version for submission

Definitions

Acronyms	Definition
AAI	authentication authorization infrastructure
AI	Artificial Intelligence
API	Application Programming Interface
APM	Assets Policy Management
CD	Continuous Development
CI	Continuous Integration
CPU	Central Processing Unit
CRUD	Create Retrieve Update Delete - Basic Operations in DBMS
CSS	Cascading Style Sheets
DCAT	Data Catalog Vocabulary
EU	European Union
FAME	Federated decentralized trusted dAta Marketplace for Embedded finance
FDAC	Federated Data Assets Catalogue
FFGA	FAME Federation Governance Application
FGB	FAME Governance Board
FMAP	FAME Marketplace Platform
GDPR	General Data Protection Regulation
GOV	Operational Governance
HTML	Hypertext Markup Language
HTTP	HyperText Transfer Protocol
HW	HardWare
ID	Identity
IP	Internet Protocol
JSON	JavaScript Object Notation
ML	Machine Learning
OS	Operating System
POD	Pay On Delivery|Point Of Distribution
RAM	Random Access Memory
REST	Representational State Transfer
SA	Supervisory Authority
SSI	Server Side Includes
SSL	Secure Sockets Layer
UI	User Interface
URL	Uniform Resource Locator

VM	Virtual Machine
XAI	Explainable Artificial Intelligence

Executive Summary

A thriving digital market is essential for Europe to operate cohesively in line with principles of self-determination, privacy, openness, security, and fair competition. The burgeoning data economy demands a robust legal framework encompassing data protection, fundamental rights, safety, and cybersecurity.

The EU has made significant strides in harmonizing digital markets, exemplified by the European Strategy for Data [1]. Building on this, the Data Governance Act [2] and Data Act [3] aim to foster data sharing and trust within the EU.

Data Spaces are identified as strategic infrastructures to drive the European data economy while minimizing environmental impact. To address the challenges of trust, legal compliance, and technical interoperability, FAME emerges as a distinct Federated Data Space.

Tailored for the financial sector, FAME offers a unified platform for diverse data assets, including AI models, analytics, and executable services. Its federated governance model ensures secure, compliant data transactions. Leveraging advanced technologies like blockchain and AI, FAME supports dynamic identity and access management for sustainable self-sovereign identity adoption.

This document outlines the FAME-enabled process and user stories, providing a comprehensive overview of the integrated architecture underpinning the FAME SA solution.

This document introduces the FAME Integrated solution, outlining how independently developed modules have been combined into a cohesive system. By adopting a microservices approach, the architecture prioritizes flexibility and scalability.

The document describes a user-friendly dashboard connected to robust backend APIs. While providing a comprehensive overview of backend module integrations, detailed specifications are available in separate documents. A CI/CD infrastructure to support development and deployment is also detailed, ensuring efficient and reliable delivery through automated build, test, and deployment processes.

Leveraging DevOps principles and modern technologies like containers and microservices, the marketplace will be built using an agile, iterative approach. Integration of technical modules from other project phases and open-source data management tools, including data cleansing and synthetic data generation capabilities, is also planned and will be accomplished in the second part of the project.

This document presents the initial findings of Tasks 2.3 and 2.4, covering activities up to project month 18. The next version of this deliverable, D2.4, is scheduled for 30 June 2025 and will encompass the remaining project activities.

Table of Contents

List of Figures

List of Tables

1 Introduction

For Europe to operate cooperatively and in accordance with European principles, such as self-determination, privacy, openness, security, and fair competition, the digital market is crucial.

The expanding data economy needs a legal framework to define data protection, fundamental rights, safety, and cybersecurity. The harmonization of digital markets is one of the fundamental policy achievements of the European Union (EU), with the European Strategy for Data [1] serving as its primary tangible product.

The Data Governance Act[2] comes next, with the goal of promoting usable data by boosting EU-wide data sharing procedures and enhancing trust in data intermediaries.

The Data Act [3] , a legislative proposal that intends to establish a framework that will promote business-to-government data sharing, is also another vision of the EU data economy.

The European Commission has identified Data Spaces as strategic infrastructures to promote the European data economy's growth while minimizing human and environmental carbon footprints.

These infrastructures ensure reliable data sharing and exchange based on agreed principles. This entails numerous challenges:

- Business/Organizational Challenges: Companies must build trust, adhere to EU rules, and keep pace with market changes.
- Legal Compliance Challenges: GDPR and other legal rules protect privacy and determine data ownership, aligning with EU-wide policies.
- Technical Challenges: Interoperability, provenance, quality assurance, and scalability issues in data sharing solutions require attention.

FAME is a Federated Data Space and distinguishes itself from other data marketplaces in several key aspects:

- Market Focus: FAME, tailored for the financial sector, provides a unified platform for customized Data Asset utilization, fostering innovation and efficient service delivery.
- Products Diversity: FAME offers diverse federated data assets, including classical datasets, value-added assets like AI/XAI models, analytics, algorithms, executable services, and educational content.
- Legal/ Governance Model: FAME implements a federated governance model to ensure trustworthy, sovereign, private, and secure data transactions in line with EU regulations, fostering transparency and consistency.
- Cutting-edge Technologies: FAME employs advanced technologies like semantic interoperability standards, AI, machine learning for enhanced analytics, and blockchain for secure, transparent transactions, and asset tokenization, adaptive authentication and authorization infrastructure (AAI) to deal with dynamic target platforms, enabling adaptability for dynamic identity and access management, dynamic policy management process enables policy adaptability for dynamic identity and access management (IAM) for sustainable adoption of SSI (self-sovereign identity).

Aligned with the presented vision, this document details the process and user stories supported by FAME and provides a comprehensive overview of the integrated architecture underpinning the FAME SA solution.

This document presents the first part of the activities carried out in Tasks 2.3 and 2.4 up to project month 18. The subsequent version and final version of the deliverable, D2.4, is planned for project month 30, scheduled for 30/06/2025, and will include the activities that will be carried out in the second part of the project.

1.1 Objective of the Deliverable

This document introduces the FAME Integration Architecture. It outlines the processes and user stories derived from the requirements analysis and details how independently developed modules have been integrated. The architecture leverages a modern microservices approach, enhancing flexibility and scalability.

The document describes a user-friendly dashboard and its connection to backend APIs. While a comprehensive overview of backend module integrations is provided, detailed specifications can be found in dedicated deliverables.

Additionally, the document presents the CI/CD pipeline, which automates build, test, and deployment processes, ensuring efficient and reliable delivery of FAME technologies and pilots.

1.2 Insights from other Tasks and Deliverables

The FAME Integration Architecture represents a critical milestone in the project. Building upon the requirements outlined in the FAME Solution Architecture (D2.6 - Technical Specifications and Platform Architecture II [5]) and the detailed requirements analysis (D2.5 - Requirements Analysis, Specifications and Co-Creation II [6]), this architecture defines the principles and methods for integrating the various components of the FAME platform.

It integrates findings from previous work packages, including the development of core functionalities (D3.2 - Federated Data Assets Catalogue I [7], D4.1 - Blockchain-based Data Provenance Infrastructure I [8], D4.2 - Pricing, Trading and Monetization Techniques I [9]), the design of the user interface (D3.3 - Mechanisms and Tools for Regulatory Compliance I [10]), and the initial development of analytical assets (D5.1 - Trusted and Explainable AI Techniques I [11], D5.2 - Energy Efficient Analytics Toolbox . I [12]).

This integration architecture provides a roadmap for future work, such as the development of the FAME Marketplace (WP2, WP3, WP4), the integration of advanced analytical capabilities (WP5), pilot implementations in real-world scenarios (WP6), and exploitation/dissemination of project results (WP7).

1.3 Structure

The deliverable is structured as follows:

- *Section 1* (current Section) includes an introduction FAME project's vision including the objectives of the current deliverable, as well as the relation of this deliverable with the existing ones and the project's technical WPs
- *Section 2.* This section details the user journeys and flows resulting from the requirements analysis. It outlines how users and external systems interact with the system and describes the principles and process of joining the federated network.

- *Section 3.* This section presents the dashboard through which users interact with the system. It details how the dashboard integrates with backend systems and how the various backend modules interact with each other.
- *Section 4.* This section describes how managed assets are integrated into the FAME Federated Catalogue and how they are managed.
- *Section 5.* This section documents the DevOps practices for the FAME project. It illustrates the application of the FAME CI/CD architecture and its role in the development and integration of the FAME marketplace components.
- *Section 6.* This section documents the guidelines and specific CI/CD processes defined for building the FAME platform, both for platform-specific components and for the development of pilot components that utilize the project infrastructure.
- *Section 7.* This section contains the conclusions drawn from the FAME integration architecture.

2 Federation process, user journeys and user flows

2.1 Introduction

To illustrate the services offered by the FAME marketplace and platform, this section details the steps users take to complete core actions such as federating, searching for assets, proposing assets for sale, purchasing assets and more. These processes are outlined in terms of **user journeys** and **user flows**.

User journeys and user flows are essential tools in understanding and improving user experiences:

- **Focus on the user:** Both methods prioritize the user's perspective, aiming to identify their needs, pain points, and goals.
- **Design-centric:** They are integral to the design process, aiding in both the initial conceptualization and refinement of products or services.
- **Goal-oriented:** User journeys and flows are structured around specific user objectives, providing a clear path to follow.

By combining these elements, FAME was designed, implemented and integrated in a more intuitive way, with the goal to provide experiences for the users.

2.2 User Journeys

2.2.1 Onboarding of Federation Member (Federation Process)

The Federation process describes the steps that members must follow to be included into the federation:

1. FAME is a regulated Marketplace for regulated access of assets (data assets and technologies)
2. Every producer / consumer must guarantee:
    a. Compliance with EU regulations (e.g GDPR, Data Act[2], Data Governance Act [1], AI Act [13], …)
    b. Ethical behaviour:
        i. Conduct Code
        ii. Adhere to SDG (Sustainable Development Goals)
    c. Legal Obligations (IP protection, Law and regulation compliance…)
3. FAME enables search, publication, of assets for EmFi applications

The FAME marketplace will distinguish different types of users with different roles as illustrated in D2.5 - Requirements Analysis, Specifications and Co-Creation II [6] . For the sake of the following user stories, the following users will be considered.

- Private User (non-affiliated users, individual, citizens, …)
- Entity (Legal entities, Organizations, Companies, Universities, profit & no-profit)
- Marketplace (for M2M interaction)

FAME has the objective to create a community of trusted users, that brings value to the marketplace and has a federation mechanism to manage users in particular business organizations.

The FAME governance onboarding structured approach ensures a systematic process for user and organization federation, covering the entire lifecycle from onboarding to active participation and continuous improvement.

The FAME user federation mechanism is a process with onboarding and consensus criteria that involves several detailed steps. Below is the FAME structured process that encompasses user and organization onboarding, with consensus criteria, and maintaining the federation.

This process happens mostly offline in an application that will be developed once the business and legal framework of FAME will be defined.

By following the following steps, FAME can create an inclusive, efficient, and transparent user federation with well-defined onboarding and consensus criteria.

There are two applications involved:

- FAME FEDERATION GOVERNANCE APPLICATION (**FFGA**)
- FAME MARKETPLACE PLATFORM (**FMAP**)

The FAME Governance Board (**FGB**) is the actual ensemble of all federated identity present at a given time into the FFGA. They are entitled to vote for application and memberships to the FAME marketplace. Once the

2.2.1.1 Step 1: Define Objectives and Scope

Objectives:

- To create a value organization/user federation that share FAME common values
- To ensure smooth onboarding of new users
- To establish clear consensus criteria for decision-making

Scope:

- Applicable to users and organization willing to join the FAME federation.
- Covers the entire lifecycle from onboarding to active participation.

2.2.1.2 Step 2: User Onboarding Process

Pre-Onboarding Preparation

- Documentation: FAME provides onboarding guides, terms of service, privacy policies, and user agreements.
- Tools: FAME provides necessary tools and platforms for user interaction (e.g., forums, communication channels) in FFGA

User Application Process

- Application Form: FFGA provides an online application form collecting necessary user information.
- Verification: FFGA provides an offline verification process to ensure the authenticity of users.

Initial Screening

- Criteria Check: Verify if applicants meet basic criteria (e.g., company legal information, relevant scope for FAME, trustworthy, provenance).
- Background Check: Conduct basic background checks if necessary.

Federation Approval

- Review Process: FFGA sends the information about the organization/user to the FAME Governance Committee to review and approve applications.

- Notification: FFGA notifies users about their application status (accepted, rejected, or need more information).

Orientation

- Welcome Session: FFGA provides a virtual orientation session.
- Resources Access: FFGA provides access to necessary resources, tools, and introductory materials.

2.2.1.3 Step 3: Establish Consensus Criteria

Define Consensus Areas

- Policy: Identify areas requiring consensus (e.g., rule changes, major decisions).
- Scope: Define the scope of decisions needing consensus (strategic, operational).

Develop Consensus Mechanism

- Voting System: The FAME governance board (FGB) uses a voting system based on majority.
- Quorum: The FAME governance board (FGB) uses a quorum requirements for decision-making.

Consensus Building Process

- Proposal Submission: FAME project will outline how proposals are submitted (e.g., proposal templates, submission guidelines).
- Discussion Period: FAME project will define a discussion period before voting (e.g., forums, meetings).
- Amendment Process: Allow FAME project will consider for amendments to proposals during discussions.

Decision Implementation

- Vote Execution: FFGA uses an electronic voting, ballots.
- Result Announcement: FFGA Announce results transparently to all members.
- Implementation Plan: FFGA implements a plan to take decisions effectively.

2.2.1.4 Step 4: Continuous Engagement and Feedback

Regular Meetings

- Schedule: FAME Governance Board has regular meetings for updates and discussions.
- Agenda: FAME Governance Board prepares and shares agendas in advance.

Feedback Mechanism

- Surveys: FGB Conduct regular surveys to gather user feedback.
- Suggestion Box: FGB Create a suggestion box for continuous input.

4.3. Training and Development

- Workshops: FGB organizes training sessions and workshops.
- Mentorship: FGB implements mentorship programs for new members.

2.2.1.5 Step 5: Evaluation and Improvement

5.1. Performance Metrics

- KPIs: FGB defines key performance indicators (e.g., user participation rates, decision implementation time).
- Regular Review: FGB conducts regular reviews of KPIs.

5.2. Process Improvement

- Feedback Analysis: Analyze feedback for recurring issues.
- Adjustments: Make necessary adjustments to onboarding and consensus processes.

5.3. Reporting

- Transparency: Maintain transparency by sharing regular reports with all members.
- Documentation: Keep detailed records of all processes and changes.

2.2.2 Casual Navigation

User Role: Generic / Gender : Unknown / Name: Anonymous /
Step-By-Step :

1. An anonymous user navigates with his/her web browser to the landing page
2. He/She browses through a series of pages with: Presentation, Search, FAQ, Other links

2.2.3 Organization FEDERATION

User Role: producer or consumer Organization/ Gender : female / Name: ACME

Step-By-Step :

1. A Business Organization (ACME) wants to federate to FAME
2. Alice on behalf of the organization lands with the browser on the FAME Platform or directly on the FAME federation governance application FFGA.
3. Alice is redirected to a new application that manages the process of FAME federation FFGA.
4. The FFGA presents the relevant information (what is FAME, what it means to federate, what is required…)
5. Alice is forced to read all documentation (Terms of Service, Legal notes, etc) and moves to the application form
6. Alice provides in the form the company details, and proves she operates as the ACME Legal Signatory. Alice supplies also a decentralized identifier (DID) and her contact information (company email alice@acme.com) to the FFGA
7. FFGA pre-screens and verifies automatically the application and the related information
8. FFGA organizes the information related to Alice and ACME and notifies the FGB
9. FGB have some time to inspect the application and are required to express their vote in the FFGA
10. FFGA at the time of expire of ACME/Alice application sums the votes and communicates to the FGB members the results
11. The FFGA communicates to the FAME Platform **FMAP** to register the DID provided by ACME/Alice (API request credentials)
12. In return the FMAP communicates to FFGA with the FMAP verified credentials (FVC) of the onboarded user (API send credentials)

13. The FFGA sends to the ACME/Alice the FVC, and Alice stores them in the wallet for future use with the platform FMAP.

This general process is simplified for EU private citizens and Institutions of EU or EU project in the step 6,7,8,9,10.

An external marketplace federates with the same process of an external business organization.

2.2.4 User Technical Onboarding

User Role: Generic / Gender : Female / Name: Alice /

- Alice, as representative of ACME organization which is a trusted member of the FAME business ecosystem, has identity managed by FFGA and is recognized by ACME/Alice DID
- FMAP has received from FFGA the ACME/Alice DID
- Alice installs a FAME-compatible Identity Wallet application on her personal device
- Alice can now operate on the **FMAP**

Step-By-Step :

- **FMAP** admin onboards the received DID. **FMAP** Admin does not know anything about Alice.
- **To be completed by GOV module**

2.2.5 Asset Preparation by Producer Application

User Role: Machine / Gender: Neutral / Name: Alice_Application /
Step-By-Step :

1. Alice prepare the asset with APIs, tokens, etc… to enable access to the asset
2. Alice can develop an Alice_Application with FMAP API check. In this case at run time, for every use a check of availability and a record of usage is written into FMAP.
3. Alice provides the asset credentials for access (API Links, tokens, Connector, etc) to FMAP (the publishing)

2.2.6 Asset Publishing

User Role: Producer / Gender : Female / Name: Alice /
Step -By Step :

1. Alice navigates with her web browser to the Asset Publishing Page
2. Alice fills in the form on the page with the asset's metadata, and submits it
3. Alice receives the ID assigned to their pending request and the input of a blockchain transaction
4. Alice, by means of her MetaMask wallet (browser plugin), signs the transaction with her trading account's private key and submits it to the P&T Tracing smart contract
5. Alice checks the request's status to get confirmation that it was processed successfully
6. Alice provides the requested metadata
7. Alice configures the access policy of the asset

2.2.7 Asset Publishing from other Marketplace (Application to FAMP - M2M)

User Role: Marketplace /Gender: Neutral / Name: Ext_MarketPlace

1. Ext_MarketPlace use FMAP API /Batch (using supplied DID) to
    a. submit or update the asset metadata on Fame catalogue
    b. provides the requested metadata

2.2.8 Define Offering

User Role: Producer / Gender : Female / Name: Alice
Step-By-Step :

1. Alice navigates the asset catalogue on the FAME Marketplace, loads the page that displays the details of a previously published asset, and opts for adding a new commercial offering for that asset: the Offering definition web page is loaded into the user's browser
2. *Optionally*, Alice asks for advice on pricing and receives a list of questions that is displayed in a dialog window; the user then submits their answers to the questions
3. Alice receives a price suggestion that is displayed on the Offering definition page
4. Alice fills in the form on the Offering definition page, and submits it
5. Alice provides the Access Credential for sale of the asset in the form of offering
6. Alice receives the ID assigned to their pending request
7. Alice checks the request's status to get confirmation that it was processed successfully

2.2.9 Asset Discovery for Trading

User Role: Casual / Gender : Male / Name: Bob /
Step-By-Step :

1. Bob is navigating to the 'home page' to locate assets of his interest
2. Bob chooses to search for an asset through the 'home page' (Alternatively, the user can search for an asset through the 'search page')
3. Bob is not able to view the existing assets without being onboarded
    a. Bob federates (Federation Process) is registered and logs-in
    b. Bob is not registered and proceeds to federate (FFGA followed by onboarding)
4. Bob finds an already indexed asset (quality assurance analytic)
5. The asset has already indexed to the FDAC
6. The asset has two (2) different offerings that have been added by producers

2.2.10 Asset Purchase

User Role: Casual / Gender : Male / Name: Bob /
Step-By-Step :

1. Bob is federated and onboarded and access the FAME Marketplace with is ID
2. Bob selects a specific offering of the chosen asset
3. Bob completes the check-out
4. A separated process with Financial Transaction (Token, Credit Cards, EU D€,) shows up (*)
5. FAME gets transaction fee for purchase
6. Bob receives the credentials (NFT) of the asset purchased (containing the token to access the asset e.g. API Links, tokens, connector, etc) provided by Alice

(*): This operation and Further Billing, Transactions, Data flows happens outside FAME Platform

2.2.11 Asset Use by Consumer Application

User Role: Machine / Gender : Neutral / Name: Bob_Application /
Step-By-Step :

1. Bob _Application is implement to:
    a. Access the provider asset (API Links, tokens, Connector, etc)
2. Bob_Application use the provided production credential to access the asset

3. Bob_Application use the asset (API Links, tokens, Connector, etc) for example:
    a. Download DataSet (API Links, Connector, etc)
    b. Download software (API Links, Connector, etc)
    c. Execute provider API (API Links, Connector, etc)

NB: All this operations happens outside FAME Platform

2.2.12 User Offboarding

Upon violation of conditions (Federation consensus agreement, Asset TOS) a user (any role) can be offboarded.

Step -By Step :

1. This is initiated by FGB, upon user violation of TOS.
2. User is marked as non-fame and frozen
3. User log-in is prevented in (FFGA and FMAP)

3 FAME Frontend & Backend Integration

3.1 Introduction

To provide an overview of FAME a conceptual view of has been developed to highlights the benefits for both Data Consumers and Producers.

Data Consumers, such as SMEs and Data Scientists, authenticate to access FAME's features, enabling them to search for and select data assets. Meanwhile, Data Providers, like Data Spaces and Dataset owners, connect to FAME to index their assets, allowing Data Consumers to further utilize them. This dual approach reflects FAME's bidirectional nature, fostering interaction between data providers and consumers for a mutually beneficial ecosystem.

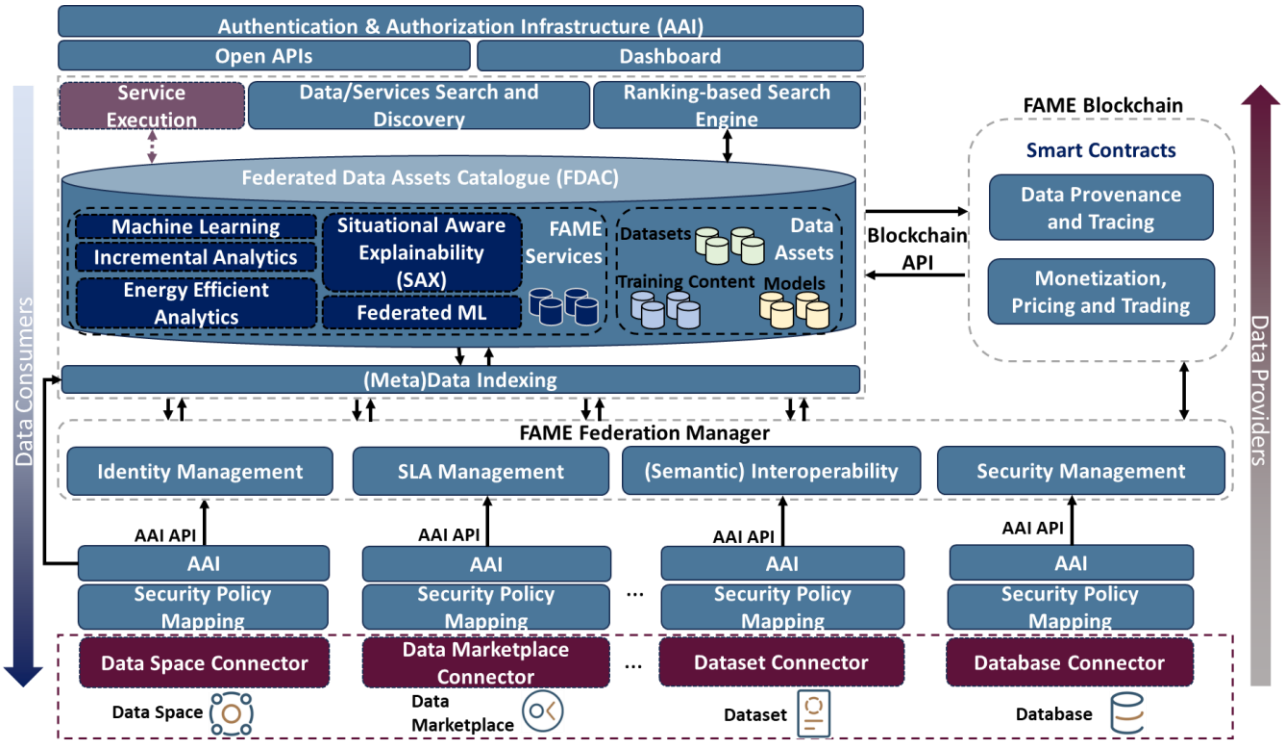

Figure 1 : Conceptual view of FAME

This conceptual view has been was then exploded in the C4 FAME architecture in the following containers: dashboards, open APIs, federation manager, transactional operations, and energy-efficient analytics services.

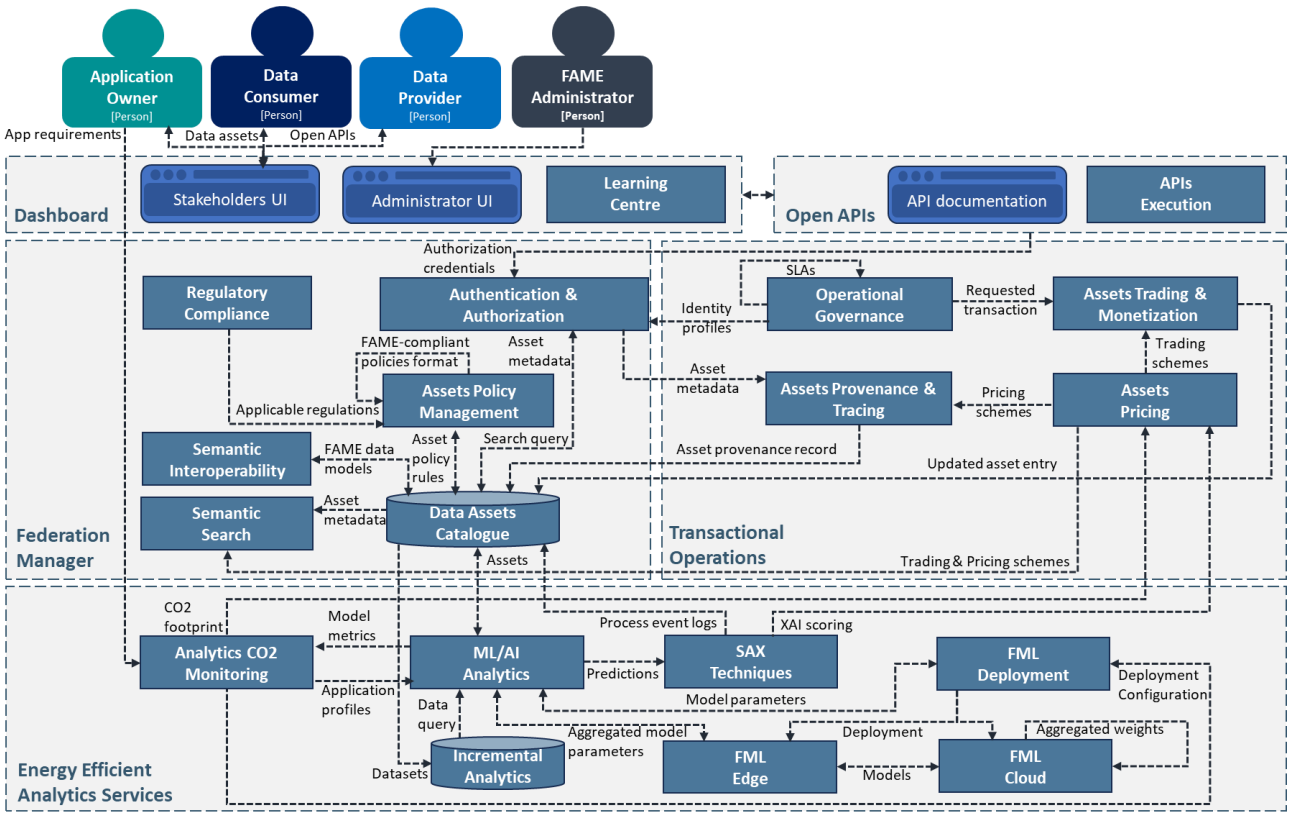

Figure 2 : FAME C4 architecture

Dashboard

The *Dashboard* layer contains all the components that are required for providing access to the FAME end-users

Open APIs layer

The interconnection between these UIs and the FAME functionalities occurs with the corresponding open APIs through the *APIs execution* service (lying into the ***Open APIs*** layer), all of which provide access to the selected operation

Federation Manager

This layer contains all the components that are needed for realizing the **federated functionalities** of FAME, ranging from the support of registration, authentication, and authorization of federated users and data sources, to the sharing and indexing of data assets deriving from such federated parties (i.e., external users and data sources).

Transactional Operations

This layer contains enabling the components that are mostly related to the **daily operation** of a Data Space that supports **trading** – i.e., the online process through which the provider and the consumer of a given data asset can stipulate a legally-binding agreement that determines the **terms and conditions of use**, and possibly includes the **contextual execution of a payment** by means of a digital currency

Energy Efficient Analytics Services

This layer brings to FAME the necessary tools for implementing **analytical functionalities** for EmFi applications. This allows end-users to apply AI/ML models to their own data (or data coming from an external source) to obtain **desired outputs** (i.e., forecasting values, ranking systems, sentiment analysis), and to receive an **explanation of the given results**, thus adding business value to client's companies subscribed to the FAME Federated Data Space

The FAME SA has been designed using the concept of the decoupled approach separating the front end from the backend that is a cornerstone of modern development, promoting flexibility, maintainability, and efficient development workflows.

The decoupled approach is a way of separating the frontend (what users see and interact with or external application interact with as already explained using a microservices architecture) from the backend (the data and logic that powers the application).

The advantages of the decoupled approach are:

- **Independent development:** Frontend and backend teams can work more independently. Frontend developers can choose the most suitable technologies for the user interface (UI) framework, while backend developers can focus on business logic without worrying about the specific UI implementation. This allows for parallel development and faster release cycles.
- **Flexibility and Scalability:** Each tier, frontend and backend, can be scaled independently based on its needs. This is especially useful for applications that experience traffic spikes. Resources can be allocated efficiently by scaling up the backend during high traffic periods and the frontend to handle increased user interactions.
- **Improved Maintainability:** With a clear separation of concerns, the codebase becomes easier to understand and maintain. It's simpler to identify issues and implement new features because the frontend and backend logic are not intertwined.
- **Teamwork and Communication:** Decoupled development fosters better communication and collaboration between frontend and backend teams. Since the codebases are separate, teams can work more autonomously while still needing to collaborate on APIs (Application Programming Interfaces) that define how the frontend and backend communicate.

3.2 FAME Dashboard

Within its complete implemented environment, FAME will provide an integrated user-friendly end-to-end User Interface (UI) facilitating the interaction of the FAME stakeholders with all the involved FAME backend services, components, and processes. Such UI, namely the FAME Dashboard, has been already modelled and depicted within the overall FAME SA as described in D2.6 - Technical Specifications and Platform Architecture II [5], and also illustrated in the figure below. As it can be observed, various types of stakeholders can have access to the FAME Dashboard, where in the cases that an external stakeholder may be an end-user representing either a data provider, or a data consumer, or an application owner intending to connect and interact with FAME, he/she has access to the Stakeholders UI. Otherwise, in the cases that the external stakeholder is a FAME administrator needing to access FAME for administration purposes or for any system parameterization/maintenance, he/she is navigated through the Administrator UI that offers a related UI for such purposes.

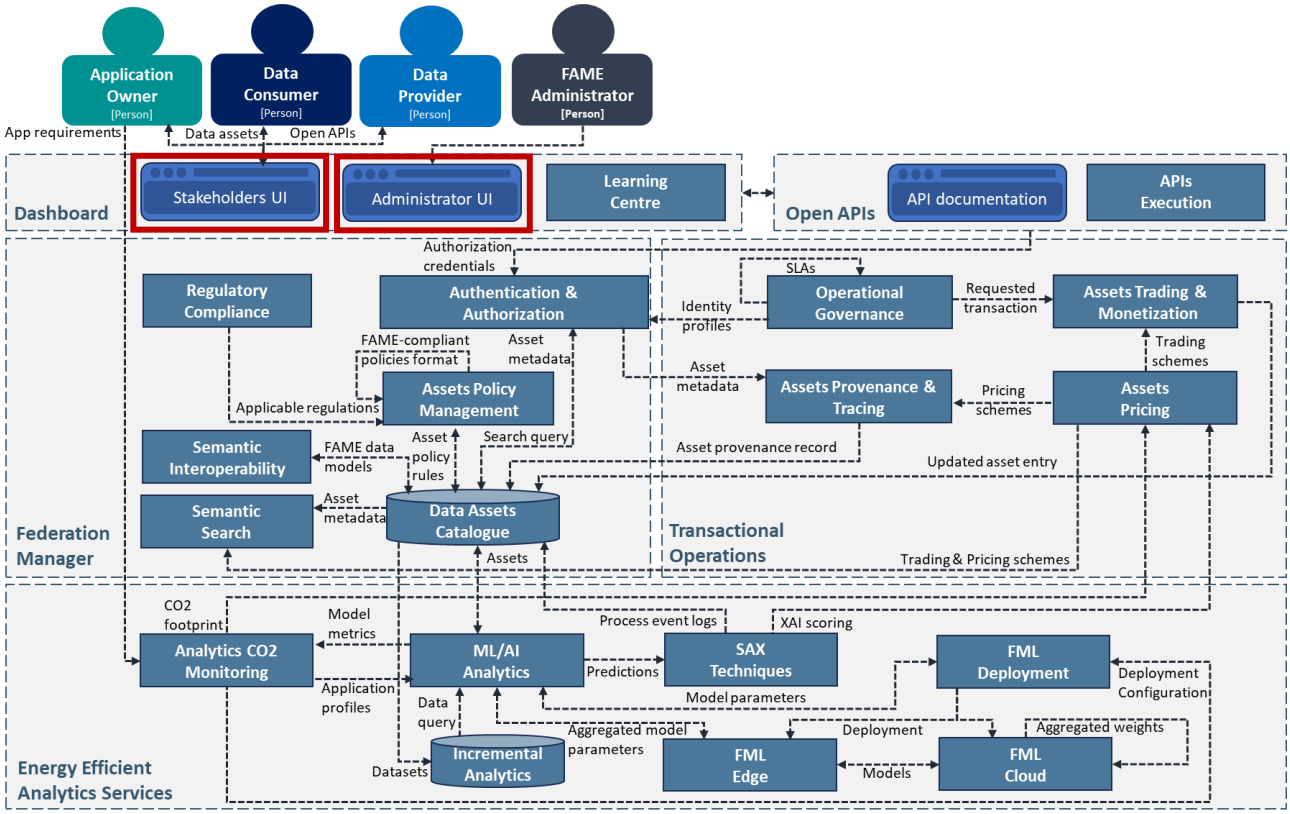

Figure 3 : Positioning of FAME Dashboard within FAME SA

Hence, the main reason behind the FAME Dashboard is to provide to all of its stakeholders (both technical and non-technical ones), a user-friendly end-to-end UI that will facilitate their interaction with all the involved FAME services, components, and processes. Towards this direction, the FAME Dashboard has been designed and implemented for providing an interface that will be aesthetically pleasing and simple to use, inviting all the FAME stakeholders (both end-users and administrators) to further explore all the associated capabilities and components of the overall platform. As a result, User Experience (UX) will be enhanced with an emphasis on streamlining complicated FAME-related activities (such as searching, monetization, and trading), guaranteeing fluid navigation, and ensuring that end-users can perform their tasks with ease. Thus, a strong brand identity will be built, helping towards transforming FAME to a widely known single-entry point of data assets and functionalities related with EmFi applications.

3.2.1 Dashboard design process

Prior to initiating the overall design process of the FAME Dashboard, a specific methodology was followed in order to fulfill the different FAME platform requirements and facilitate the final integration. Such methodology is traditionally followed when designing applications, including a variety of steps for capturing the different needs and point of views. This methodology considers both the User Interface (UI) and User Experience (UX) aspects, considered in general critical when designing digital products. While they are often used interchangeably, they have distinct roles and responsibilities. The key difference between UI and UX is that the UI considers the visual and interactive elements of a digital product that facilitate user interaction, whereas the UX deals with the overall experience that users have while interacting with a digital product, including their emotions, perceptions, and responses. Considering the latter, the steps followed for the FAME Dashboard design process are depicted below:

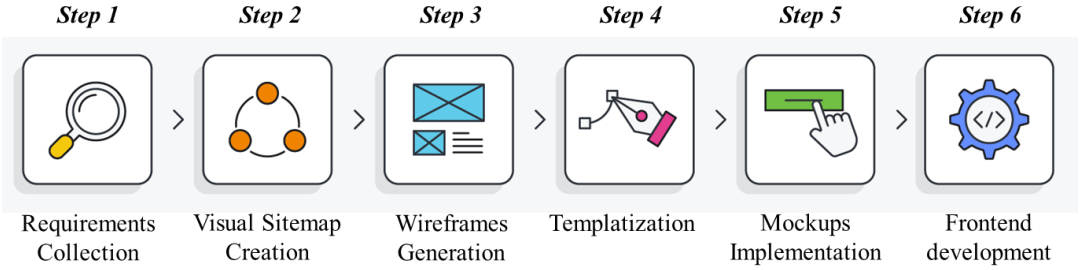

Figure 4: FAME Dashboard design process

3.2.1.1 Step 1: Requirements Collection

To make the FAME Dashboard interface more engaging to the user, it has been considered that the UI design should be a top priority for the end-users. That is the reason why deep research was followed on the potential FAME users by the different designer and development teams, focusing primarily on the set requirements. Since the beginning of FAME, a plethora of Business and Technical requirements have been identified for capturing the different needs of the FAME pilots, being available in D2.5 - Requirements Analysis, Specifications and Co-Creation II [6]. These requirements have guided and facilitated the implementation process of the FAME components, whereas also boosting the design, specification, and implementation of the FAME Dashboard. Apart from the consideration of all these requirements, a questionnaire has been circulated (see the figures below) to the different FAME technical components' leaders, to better identify their needs in terms of UI design requirements, facilitations, or potential adaptations (e.g., requirement for a specific number of input and output fields or specific actions' triggering buttons). The questionnaire included several questions covering the topics of the: (i) personal information of the different technical components' leader, (ii) name/description of the technical components, (iii) requirements of the UI, (iv) details of the interacting technical components, (v) way of interaction of the technical components, (vi) totally required input/output UI fields, (vii) preferred way of communication with the FAME Dashboard, and (viii) technologies/programming languages that are used for the technical component. The figures below illustrate some indicative responses from the overall components' leader feedback.

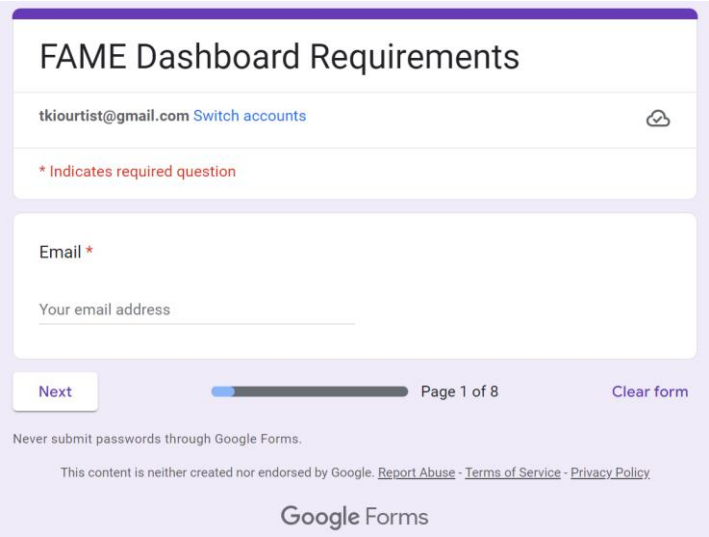

Figure 5: FAME Dashboard components' requirements collection

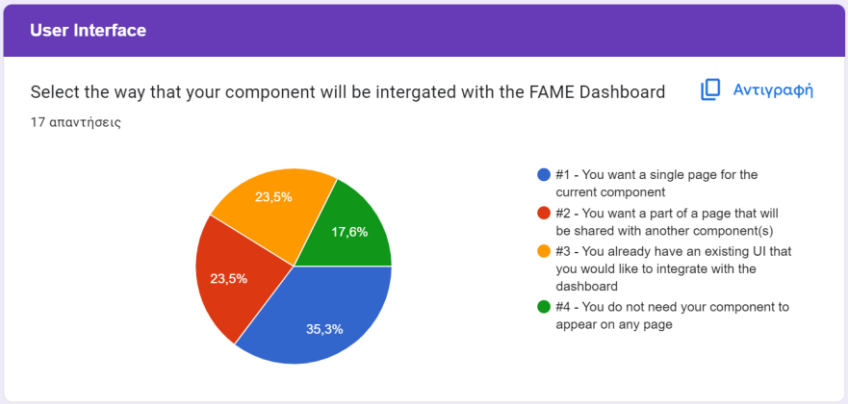

Figure 6:FAME Dashboard components' needed UIs feedback

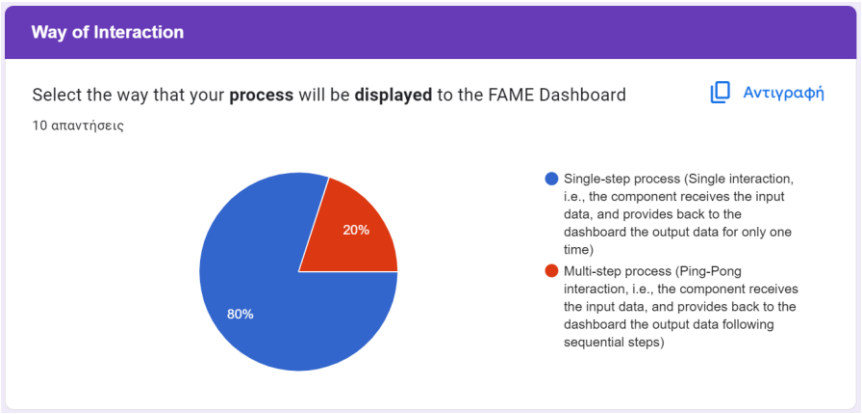

Figure 7:FAME Dashboard components' UIs interactions feedback

It should be highlighted that through this questionnaire the interactions among the different backend services were identified through the proper questions, as well as the specific UIs that were needed to be: (i) initially designed (i.e., in the case that not any prior UI was already implemented), (ii) re-designed (i.e., in the case that the already implemented UI was not fulfilling the design or integration concepts of the FAME Dashboard), or (iii) integrated with the FAME Dashboard (i.e., in the case that the already implemented UI was already in place and aligned with the design principles of the FAME Dashboard). The collected results can be seen in Table 1 that includes the list of all the available backend services of the FAME platform, and their related requirements for a specific UI or not.

It should be mentioned that *Step 3 - Step 6* of the design process described below was followed only for the backend services that did not have any implemented UI (i.e., to-be-implemented - *TBI*). For the rest of the backend services (i.e., the ones that already had a UI (*Implemented*)), a dedicated UI design methodology was followed as described in the corresponding backend services' deliverables (*Deliverable* column in Table 1).

Table 1: Status of backend services UIs

FAME Stakeholder	Backend Service/ Functionality	UI Status	Deliverable
End-user	Dashboard home	TBI	D2.3 (current deliverable)
	FDAC	Implemented	D3.2[7]
	Asset details	Implemented	D3.2[7]
	Asset provenance	Implemented	D4.1[8]
	Asset publishment	Implemented	D4.1[8]
	Asset policy editor	Implemented	D3.1
	Offering publishment	Implemented	D4.1[8]
	Offering details	Implemented	D3.2[7]
	Pricing advisor	Implemented	D4.2[9]
	Trades	TBI	D2.3 (current deliverable)
	Trade details	TBI	D2.3 (current deliverable)
	Search	Implemented	D4.2[9]
	About us	TBI	D2.3 (current deliverable)
	Helpdesk	TBI	D2.3 (current deliverable)
	Profile	TBI	D2.3 (current deliverable)
	Profile details	TBI	D2.3 (current deliverable)
	Account details	TBI	D2.3 (current deliverable)
	Tickets details	TBI	D2.3 (current deliverable)
	Member login	Implemented	D3.1
	Learning centre	TBI	D2.3 (current deliverable)
	Regulatory compliance	Implemented	D3.3[10]
	Analytics	Implemented	D5.1[11], D5.2[12]
Administrator	Admin home	TBI	D2.3 (current deliverable)
	Admins	TBI	D2.3 (current deliverable)
	Admin onboarding	TBI	D2.3 (current deliverable)
	Admin details	TBI	D2.3 (current deliverable)
	Members	TBI	D2.3 (current deliverable)
	Member onboarding	TBI	D2.3 (current deliverable)
	Member details	TBI	D2.3 (current deliverable)
	Trusted sources	TBI	D2.3 (current deliverable)
	Trusted source onboarding	TBI	D2.3 (current deliverable)
	Trusted source details	TBI	D2.3 (current deliverable)
	Traders	TBI	D2.3 (current deliverable)
	Trader details	TBI	D2.3 (current deliverable)
	Trading account details	TBI	D2.3 (current deliverable)
	Trading tokens	TBI	D2.3 (current deliverable)
	Tickets	TBI	D2.3 (current deliverable)

This collected knowledge led to the next step of the FAME Dashboard design process, considering all the existing backend services of the FAME platform, namely the creation of the Visual Sitemap.

3.2.1.2 Step 2: Visual Sitemap Creation

A Visual Sitemap is an essential information architecture step that can visually expose difficult-to-crawl content and help refine a website's navigation and overall structure before and after it is built. Hence, its purpose is focusing on illustrating the relationships among all the pages of a website, and providing a clear picture of the website's content, organization, and navigation. In essence, a Visual Sitemap represents the structure of a website in an easy-to-understand visual diagram. In the context of the FAME Dashboard, such concept is quite crucial for plotting out the whole platform's frontend and backend functionalities, capturing the current state of the platform visualized outcomes and results, planning in parallel for future needs and updates. Through the FAME Dashboard Visual Sitemap, it has been easily ascertained how many pages are needed in place to cover all the underlying services/functionalities, along with the predefined requirements from the previous step, whereas it was also identified whether there are missing any crucial elements or not, how fast the FAME Dashboard could deliver the desired content to its users, as well as how efficiently its users could navigate themselves across the entire environment. The Visual Sitemap design was performed through the Octopus Visual Sitemap tool [14], keeping in mind both the end-users and the FAME platform administrators, leading to the creation of two (2) different Visual Sitemaps.

As for the end-users Visual Sitemap, illustrated in the figure below, it can be seen that the landing page of the end-user is the Dashboard Home from which the user can navigate to the different 1st level backend services/functionalities of the FAME platform, referring to the FDAC, the Search Engine, the Asset Publishment, the Learning Centre, the Regulatory Compliance Tool, the Login (Authentication/Authorization) and the Profile ones, which are directly accessible either from the Dashboard Home menu bar or the provided call-for-action buttons residing within the Home page. The rest of the backend services (i.e., Asset Policy Editor, Asset Provenance, Asset Trade, Asset Pricing) are also accessible through the FAME Dashboard, with the difference that they can be reached through the respective internal pages of the 1st level services, since their functionalities are implemented as part of those 1st level services' procedures. Also, through the Dashboard Home, the end-users can access the FAME platform general information pages (i.e., About Us, Helpdesk) for locating details upon the platform, its overall scope, supported functionalities, etc. To this end, it should be noted that in the figure below, are also depicted the actions that can be performed to each page from the end-user side, as well as the connections among the different pages that interact with the underlying backend services. Further information on the detailed content of each page depicted in the end-users' Visual Sitemap are provided in *Step 6*, where the mockups of all these pages are provided.

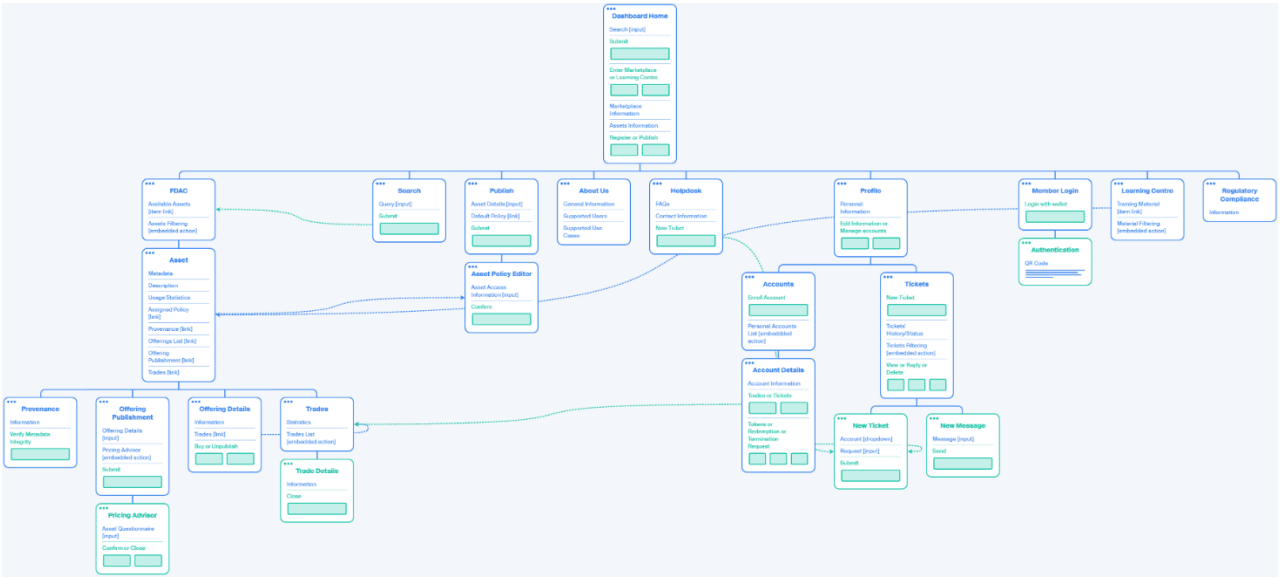

Figure 8:FAME Dashboard end-users' Visual Sitemap

As for the administrators' Visual Sitemap, illustrated in the figure below, it can be seen that the landing page of the administrator is the Admin Home from which the administrator can monitor the platform general statistics (e.g., federation members, trusted sources, catalogue entries) and navigate to the different backend services/functionalities of the administrative part of the FAME platform, referring to the Administrators', the Members', the Trusted Sources', and the Traders' management pages, as well as the related Tickets' overview page. To this end, it should be also stated that in the figure below, there are additionally depicted the actions that can be performed to each page from the administrator side, as well as the interactions among the different pages. Further information on the detailed content of each page illustrated in the administrators' Visual Sitemap are provided in *Step 6*, where the actual mockups of all these pages are provided.

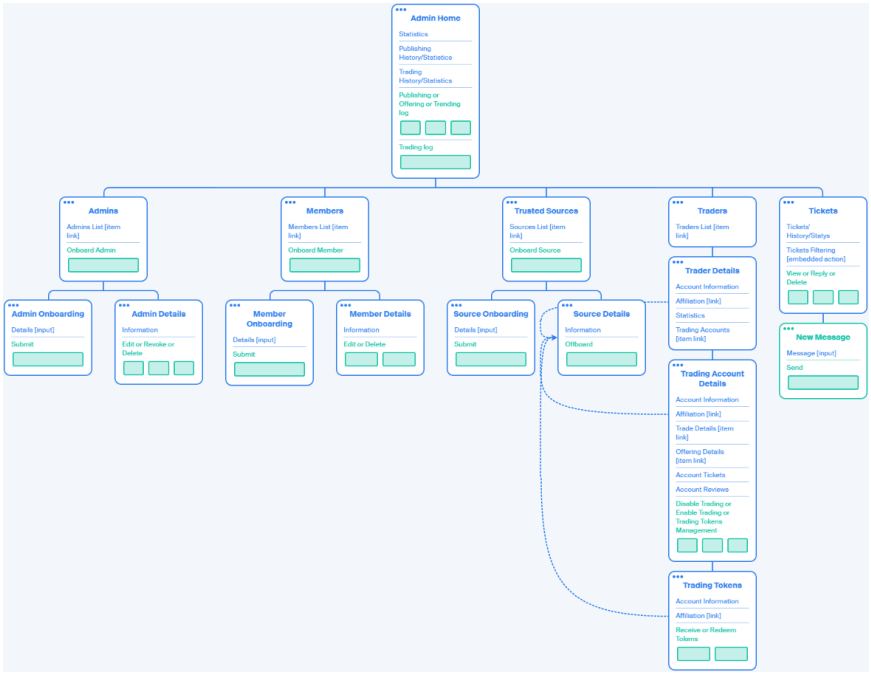

Figure 9: FAME Dashboard administrators' Visual Sitemap

The fulfilment of this part led to the initiation of the next step, regarding the Wireframes.

3.2.1.3 Step 3: Wireframes Generation

After the creation of the Visual Sitemaps, a plethora of ideas have started generated around the overall solution, and that is where the concept of Wireframes was developed. Wireframes are created before moving to the final UI design. This step of the process focuses on giving shapes to the rough ideas that were collected from the previous steps. While wireframing, text is being added, as well as sample relatable images, and dummy content (i.e., lorem ipsum text). In the context of the FAME Dashboard, the wireframes developed through Microsoft PowerPoint, involving the generation of low-fidelity outlines of the layout and structure of the complete UI. These Wireframes served as a blueprint for the design, helping into clearly identifying the placement of elements, information hierarchy, and overall flow before diving into the visual design. Moreover, Wireframes also facilitated into validating the design with all the FAME platform key stakeholders, allowing everyone to comment about what element and text goes where before anything gets finalized, and to achieve high quality results. Since multiple Wireframes were generated, being considered as primary steps of the Mockups that were generated afterwards (explained in detail in Section 3.3.1.5), only an example of a Wireframe designed for the administrator Home page of the FAME platform is being depicted in this deliverable (see the figure below) to minimize its content and total number of pages. It should be

mentioned that a similar Wireframe design concept has been followed for all of the pages that can be identified in the Visual Sitemaps, resulting into a plethora of Wireframes.

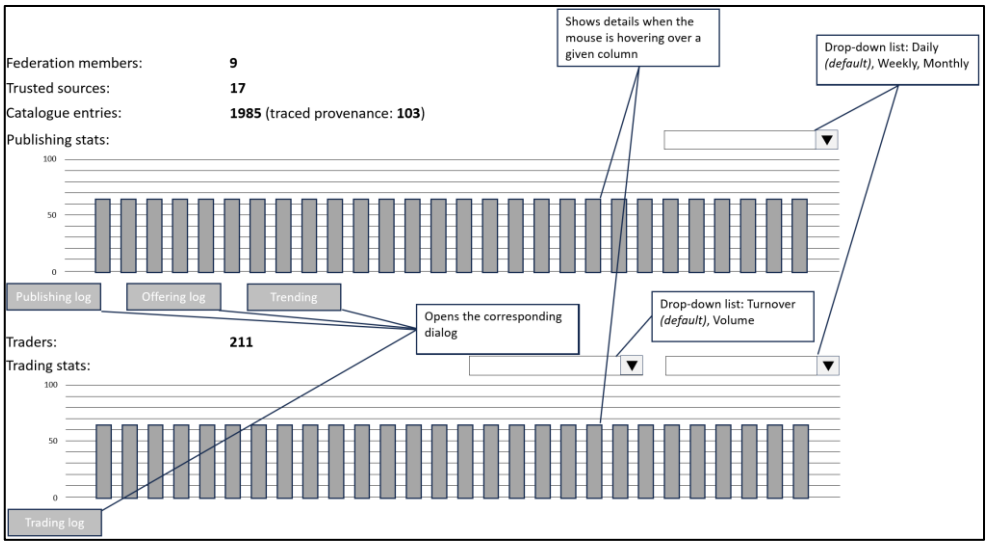

Figure 10: FAME Dashboard administrators' home page Wireframe

The finalization of the Wireframes gave the green light to the next step related with the concept of templatization.

3.2.1.4 Step 4: Templatization

The process of templatization includes the creation of reusable components across all the platform's UIs, for multiple atomic elements like buttons, input fields, or texts, among others. In general, templatization and component creation in the context of the FAME Dashboard was performed to create consistency among elements, and for all the design and implementation process to follow a common design pattern and paradigm. All these components were created by adding images, graphics text, and related placeholders, leading into a homogeneous format. The figure below illustrates a snapshot of the templatization results, showcasing the way that the navigation bar should be designed, and the related colorings of the texts and buttons that each UI designer should comply with.

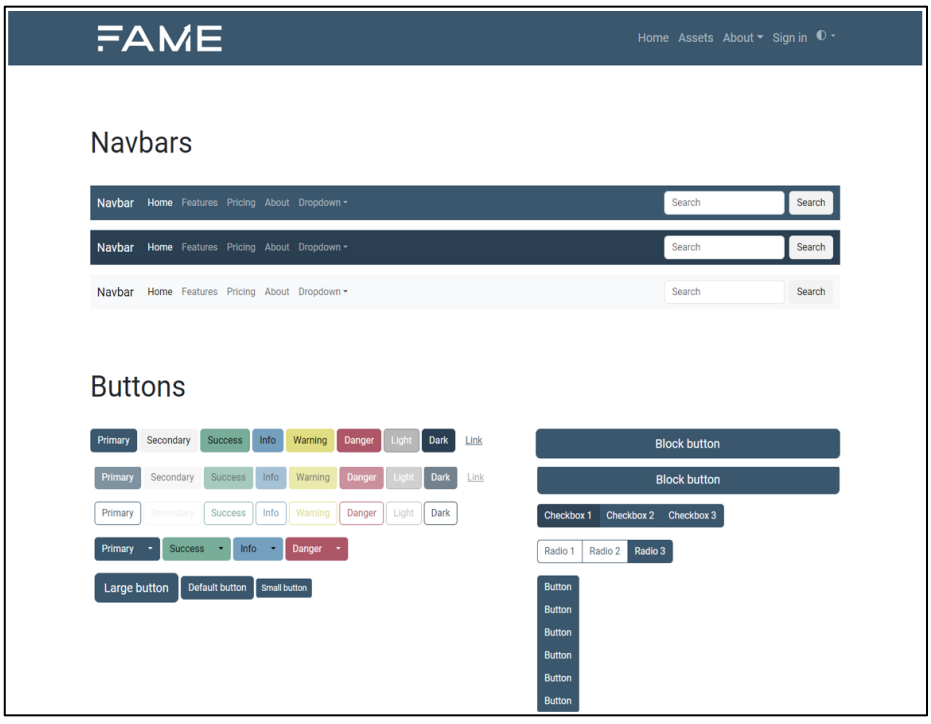

Figure 11: FAME Dashboard templatized visual elements

As soon as all the aforementioned steps have been completed, the Mockups creation phase initiated.

3.2.1.5 Step 5: Mockups Implementation

This step is to exactly design the screens as they would appear in the final product. Close to the pixel-perfect design, using all the work and data collected from previous steps, adding all the text copies, graphics, icons, and images to the design. It should be mentioned that the included texts in some cases may not be considered as final since during the frontend development phase there may be some minor textual modifications. Nevertheless, the mockups implementation phase is the step where all the colors, fonts, and aesthetics are added to the screens. For the FAME Dashboard, the latter process was conducted through FIGMA[1], whereas once the design was finished, the related screens were shown to the stakeholders to gather related feedback. A set of changes and a/b testing variants were decided based on this review, whereas as the designs were finalized, assets like images, text or translations, icons, and illustrations, were shared with the developers.

3.2.1.5.1 End-user Mockups

The Mockups that have been designed from the side of the end-users are provided in this Section. It should be noted that the purpose of this Section is not to provide an end-user manual, since this is to be depicted in the upcoming version of the deliverable, where the FAME Dashboard implementation and overall backend services' integration will have been finalized.

In this context, it should be also noted that all the different Mockups are based on a specific skeleton page (see figure below) that includes all the necessary information that should always exist in each different page of the end-users FAME Dashboard, whereas its main content is the one that is being adapted based on the purpose of each page and its related backend service. This necessary information includes the navigation menu bar from which the end-users can login and access to the different core pages of the FAME platform, the widgets through which the end-users can have immediate access to the FDAC, the Learning Centre, the Regulatory Compliance tool, and the FAME Chatbot, as well as the footer of the page containing information about FAME in general, related Legal Issues, and Communication/Social Media information. As a result, all the pages that are depicted below are built based on this skeleton page, equally adapting their content to their purpose of use.

[1] https://www.figma.com/ a collaborative interface design tool, commonly used in application development

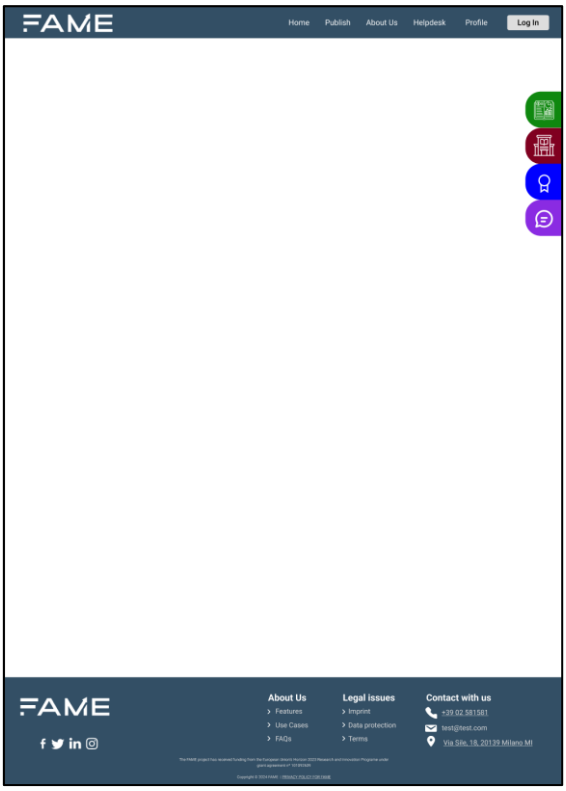

Figure 12:FAME Dashboard end-users' skeleton page

With that in mind, the rest of the Mockups follow. The Home page of the end-users can be seen in the figure below, showcasing the exact way that the FAME Dashboard Home page will be finally implemented.

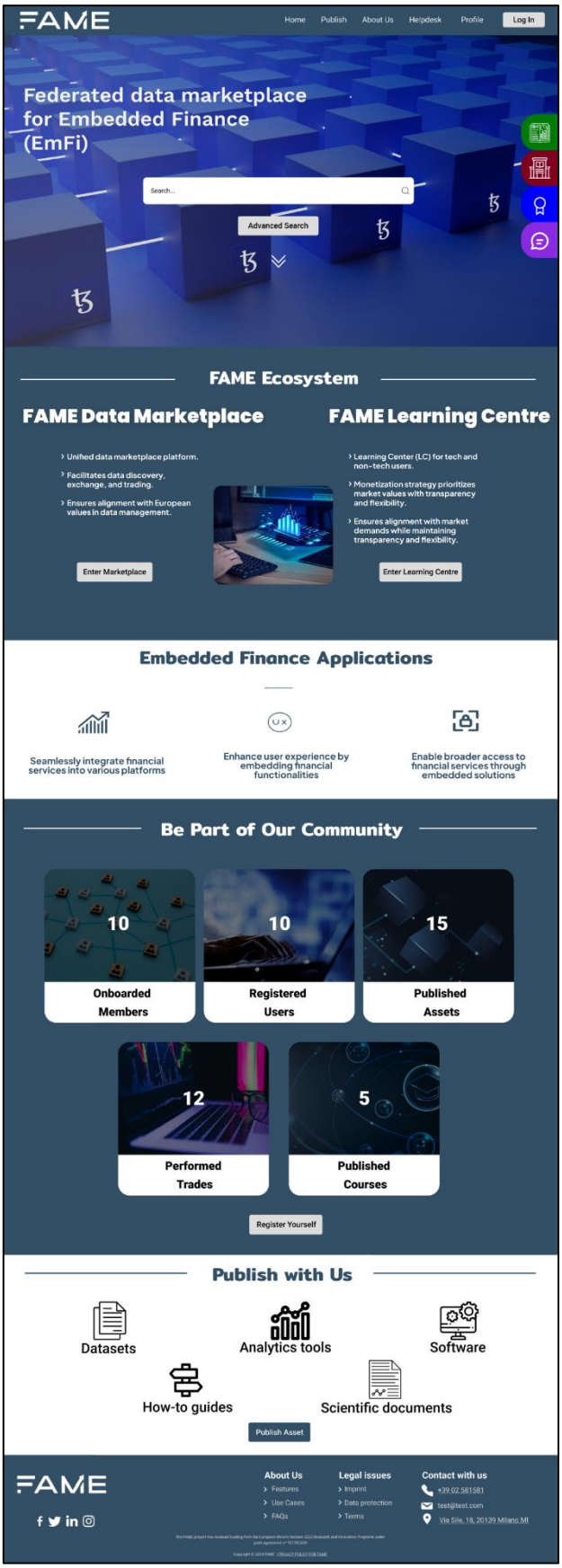

Figure 13: End-users' home page Mockup

Respectively, the About Us page of the end-users can be seen in the figure below.

Figure 14: – End-users' about us page Mockup

The Helpdesk page of the end-users can be seen in the figure below.

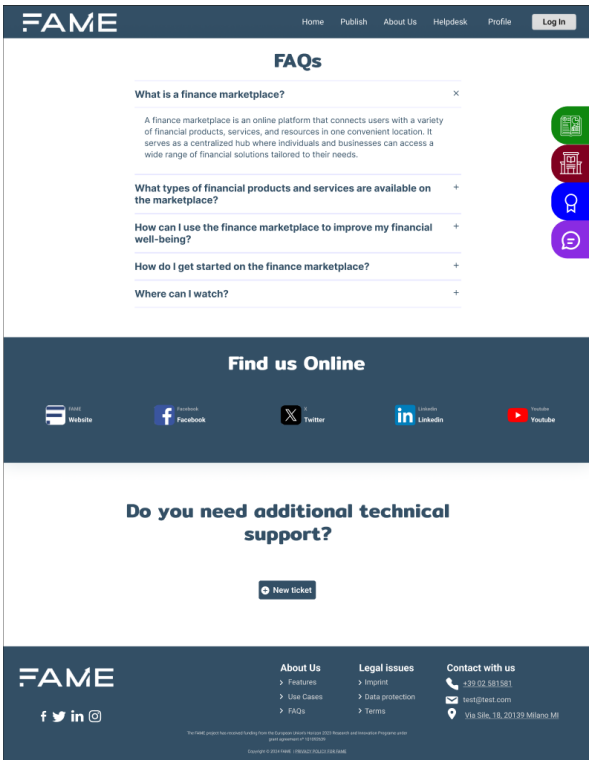

Figure 15: End-users' helpdesk page Mockup

The FDAC page of the end-users can be seen in the figure below.

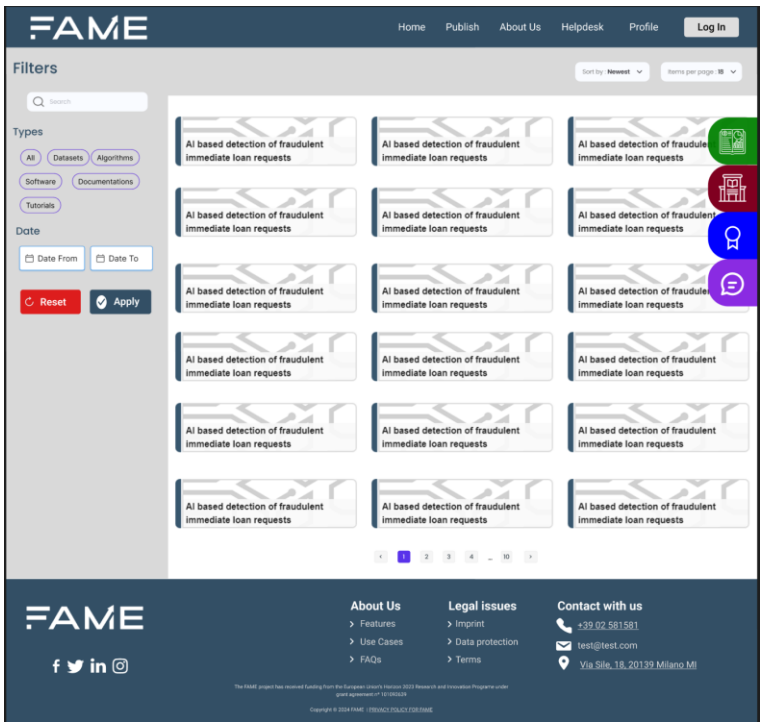

Figure 16: End-users' FDAC page Mockup

Accordingly, the Learning Centre page of the end-users is provided in the figure below.

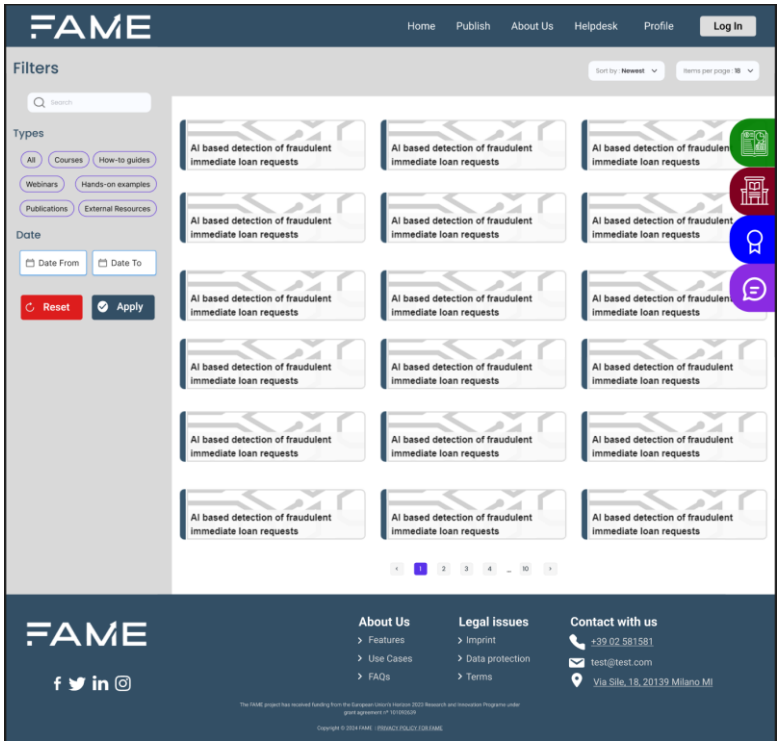

Figure 17: End-users' learning centre page Mockup

The Profile page of the end-users is provided in the figure below.

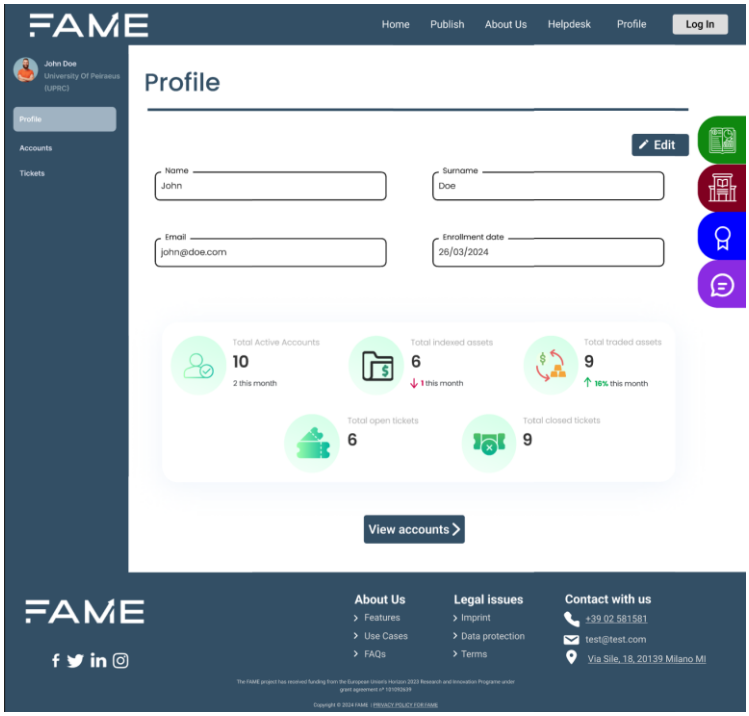

Figure 18: End-users' profile page Mockup

Respectively, the Accounts page of the end-users can be seen in the figure below.

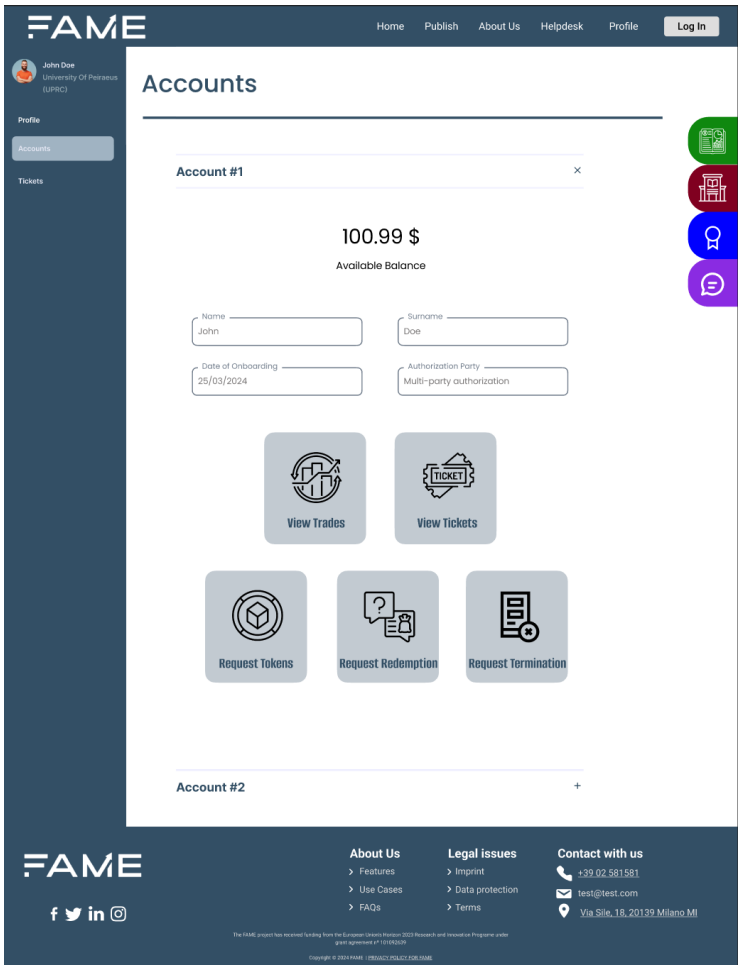

Figure 19: End-users' accounts page Mockup

The Tickets page of the end-users is depicted in the figure below.

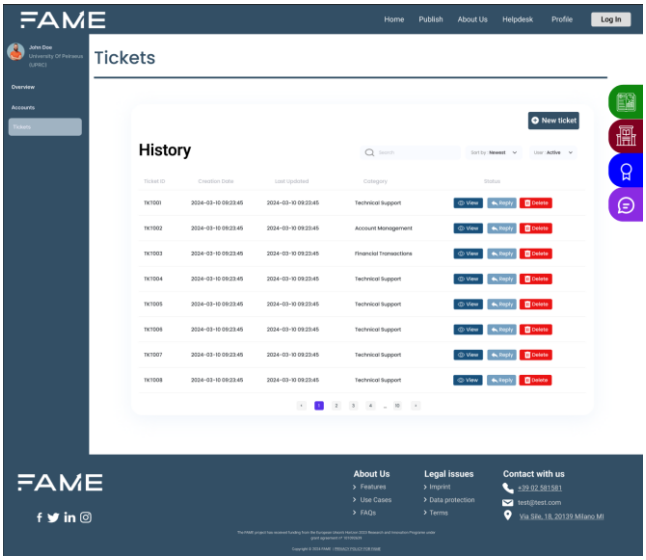

Figure 20: End-users' tickets page Mockup

3.2.1.5.2 Administrator Mockups

The Mockups that have been designed from the side of the administrator are provided accordingly in this Section. It should be noted that, as in the case of the end-users' side, the purpose of this Section is not to provide an administrator manual, since this is to be depicted in the upcoming version of the deliverable, where the FAME Dashboard implementation and overall backend services' integration will have been finalized.

In this stage, it should be also noted that as in the case of the end-users, all the different Mockups of the administrator side are based on a specific skeleton page (see figure below) that includes all the necessary information that should always exist in each different page of the administrator FAME Dashboard, whereas its main content is the one that is adapted based on the purpose of each page and its related backend service. This necessary information includes the horizontal navigation menu bar from which the administrator can logout, as well as the vertical navigation menu bar from which the administrator can access the different pages of the administrator FAME Dashboard for managing the administrators, the members, the trusted sources, the traders, and the platform tickets. The skeleton page also includes the footer of the page containing information about FAME in general, and Communication/Social Media information. As a result, all the pages that are depicted below are built based on this skeleton page, equally adapting their content to their purpose of use.

Figure 21: FAME Dashboard administrators skeleton page

Following the latter, the Home page of the administrator is offered through the figure below.

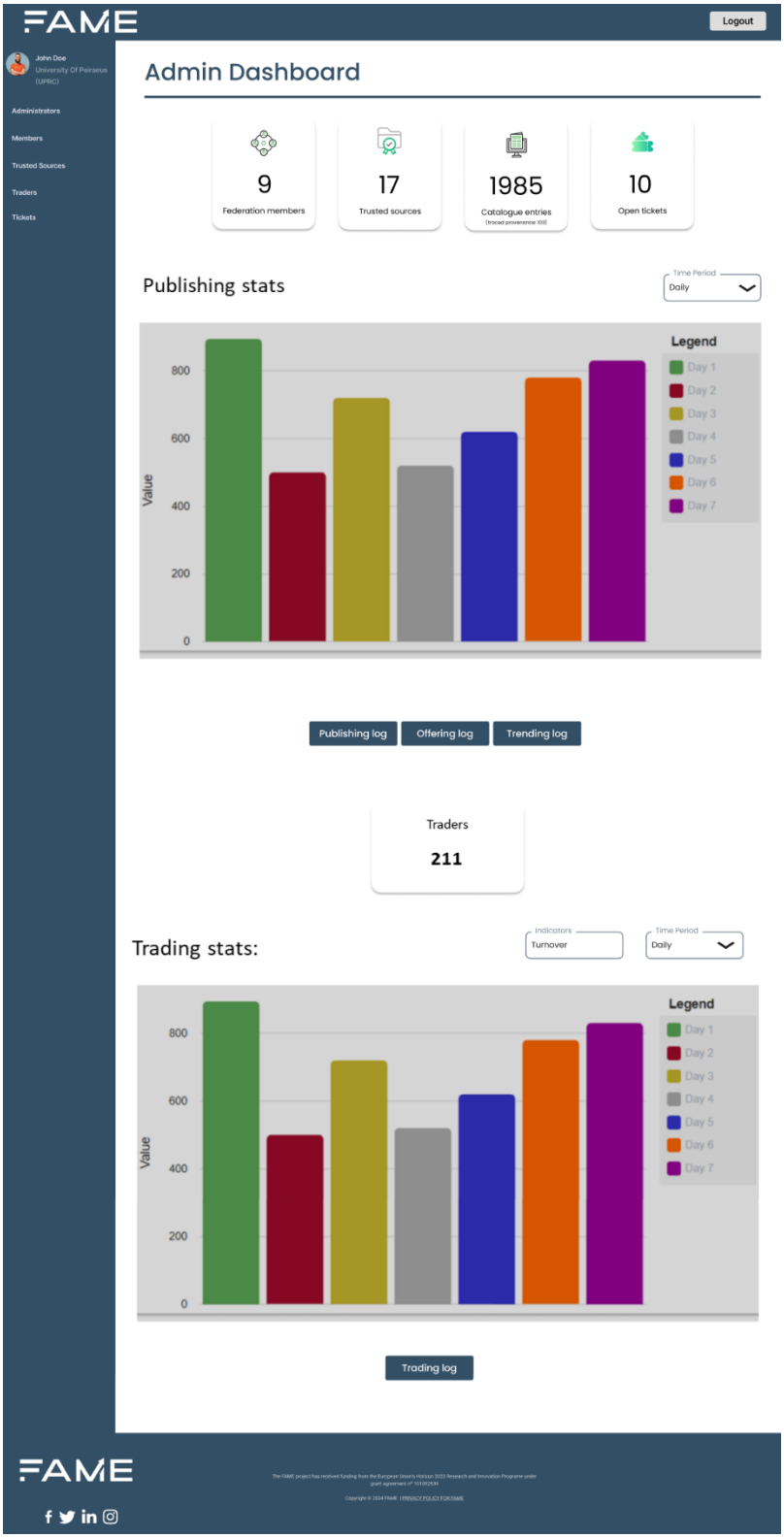

Figure 22: Administrators' home page Mockup

The Administrators management page can be seen in the figure below.

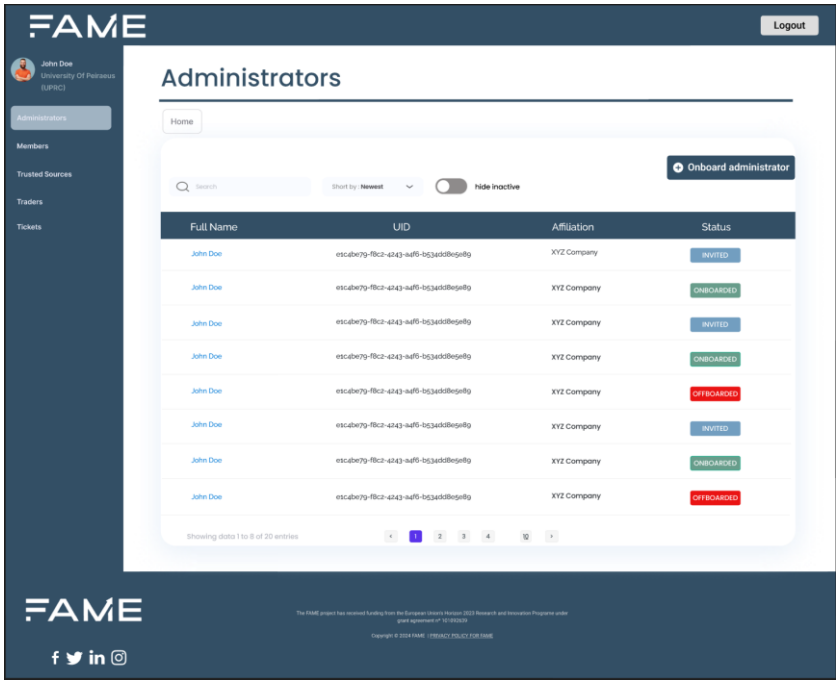

Figure 23: Administrators' management page Mockup

The Administrator onboarding page is visualized in the figure below.

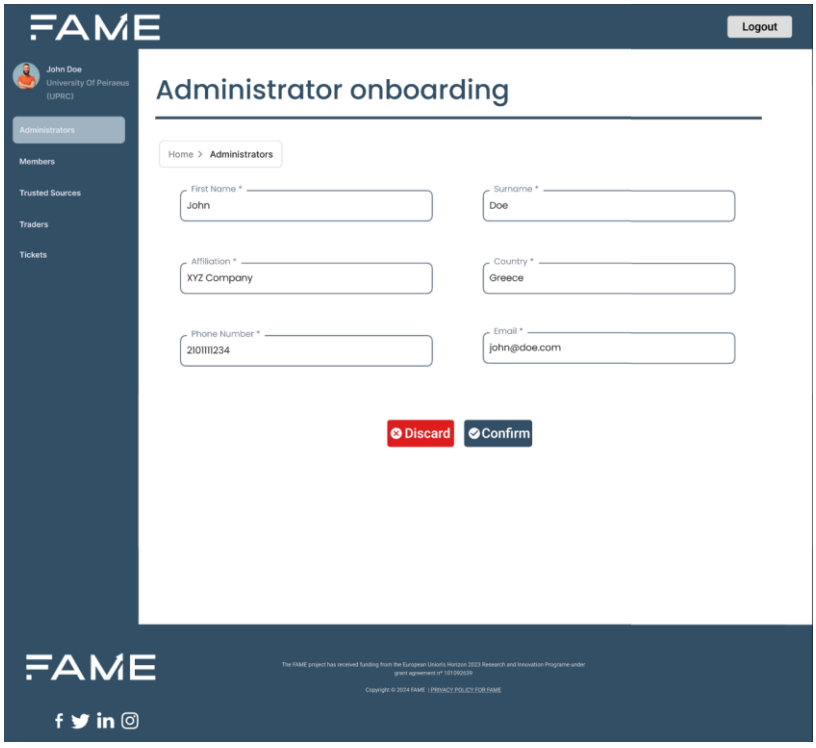

Figure 24: Administrators' onboarding page Mockup

The Administrator details page can be seen in the figure below.

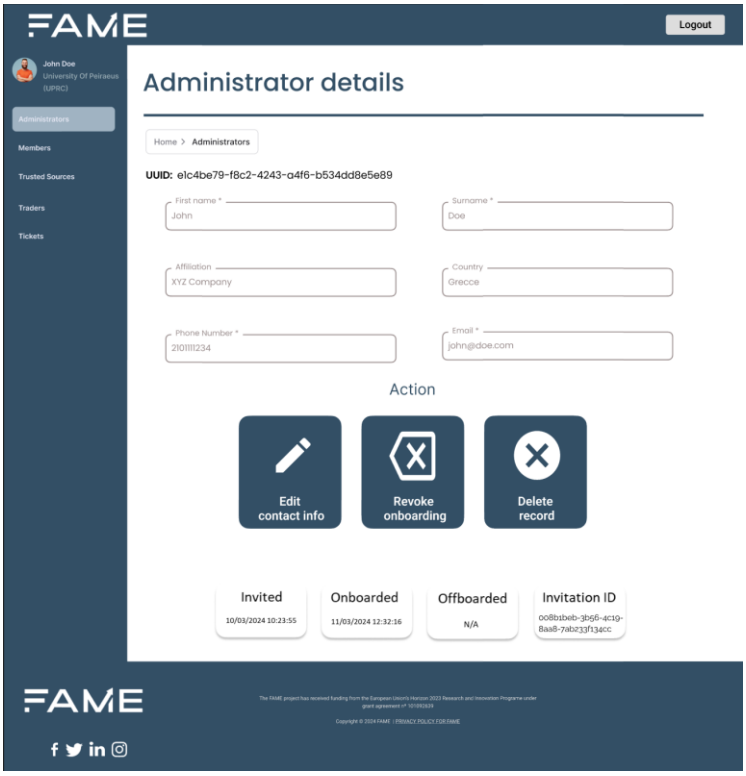

Figure 25: Administrators' details page Mockup

The Members page is provided in the figure below.

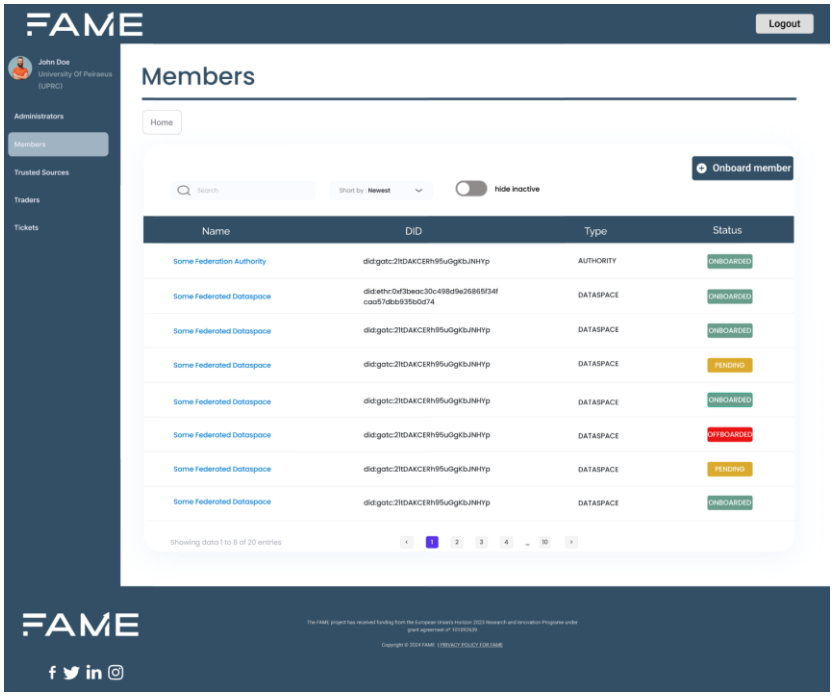

Figure 26: Administrators' members page Mockup

The Member onboarding page can be depicted in the figure below.

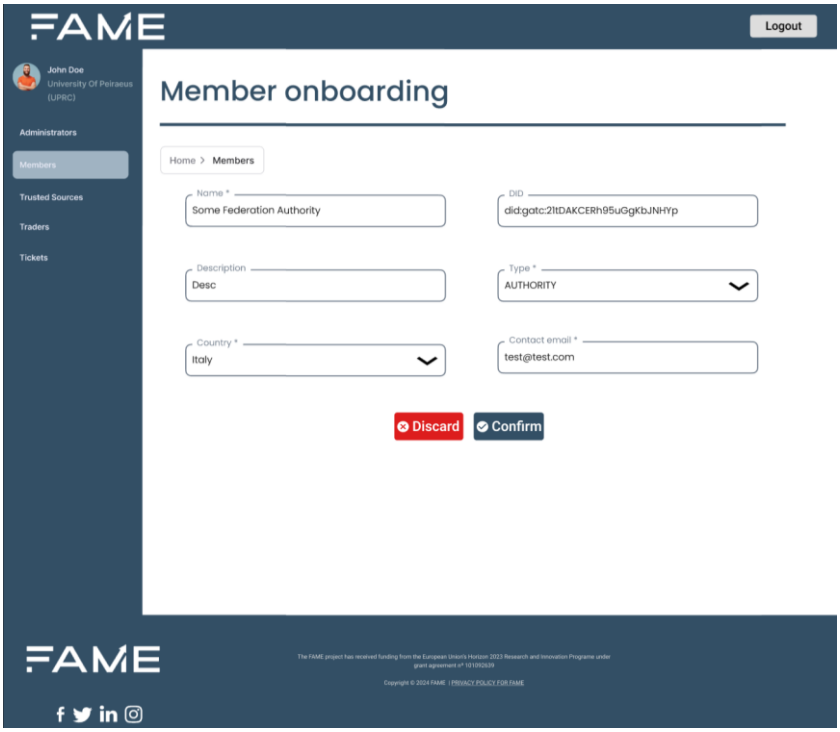

Figure 27: Administrators' member onboarding page Mockup

The Member details page can be seen in the figure below.

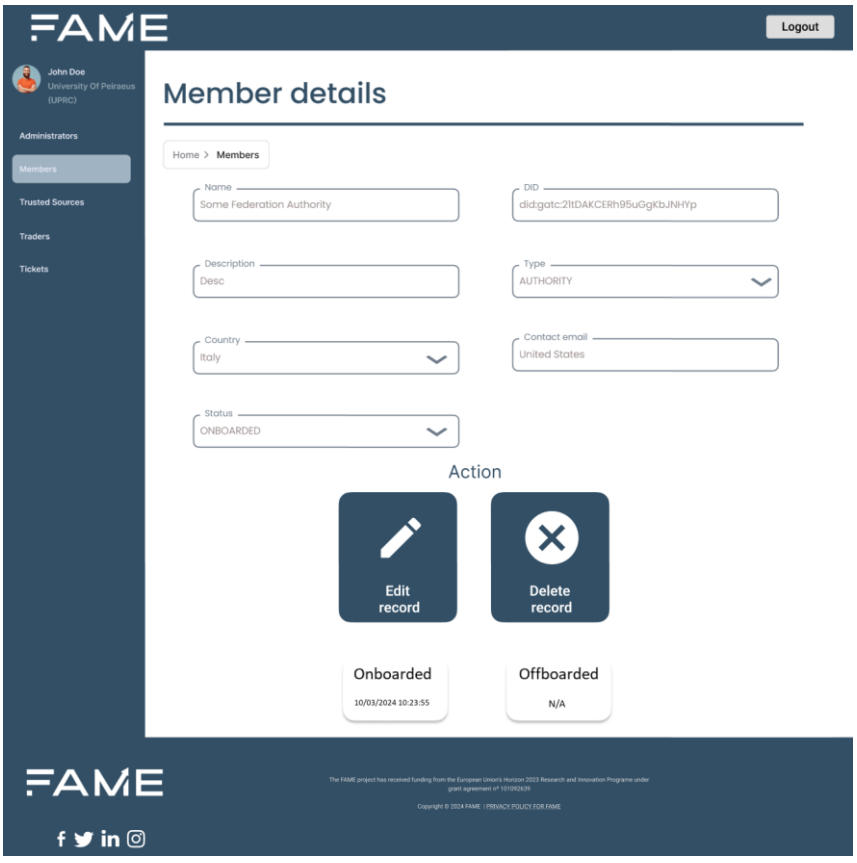

Figure 28: Administrators' member details page Mockup

The Trusted sources page is provided in The figure below.

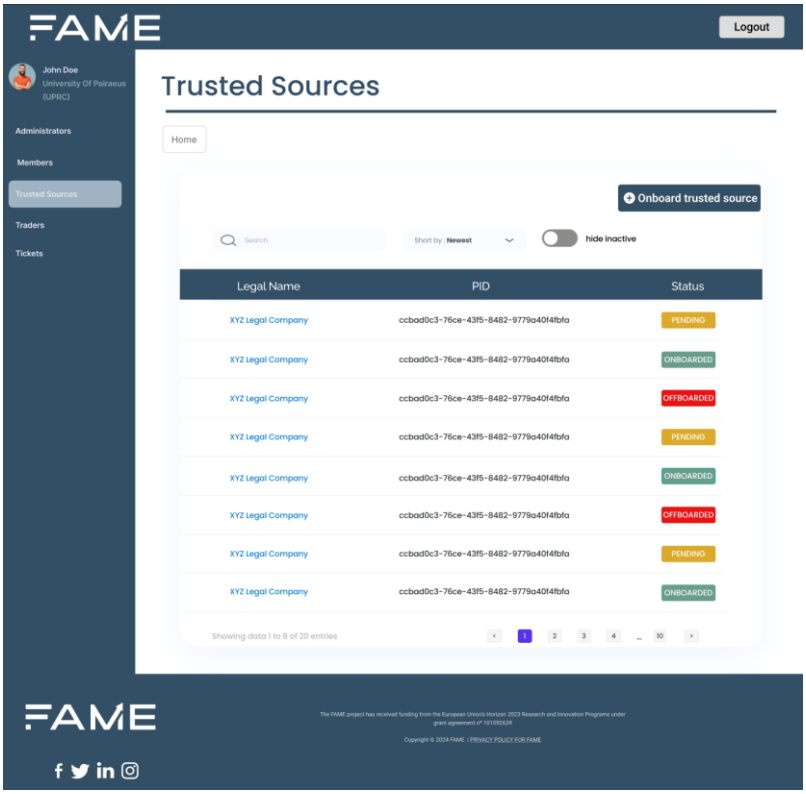

Figure 29: Administrators' trusted sources page Mockup

The Trusted source onboarding page can be depicted in The figure below.

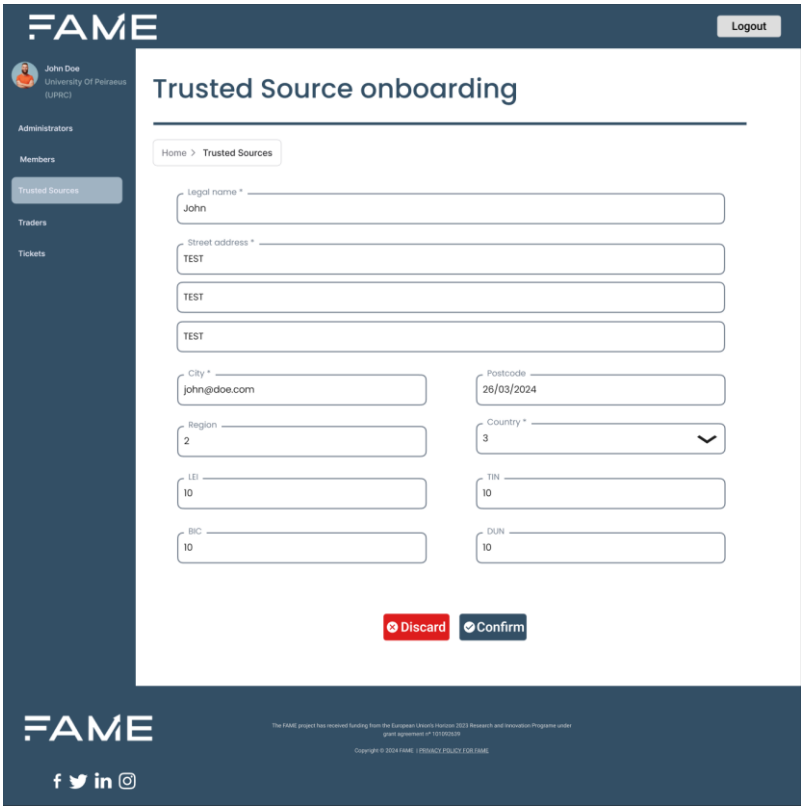

Figure 30: Administrators' trusted source onboarding page Mockup

The Traders page can be visualized in The figure below.

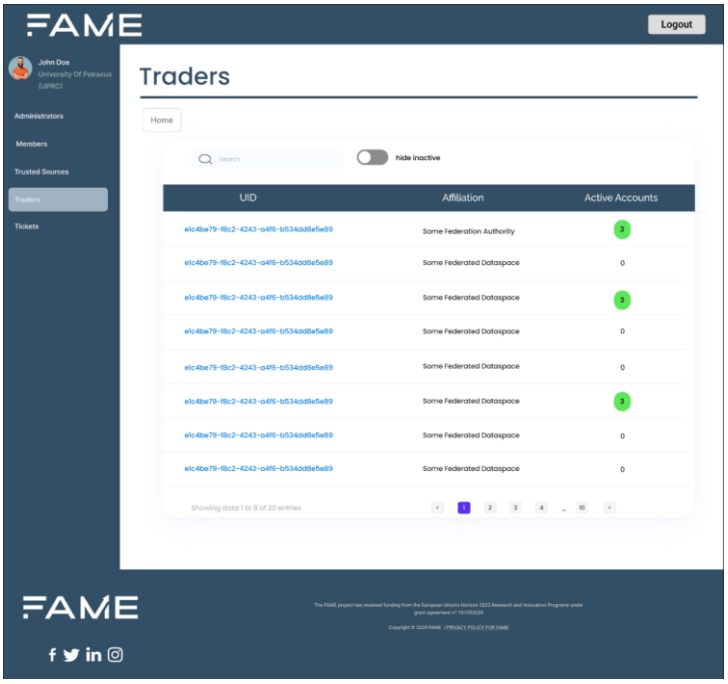

Figure 31: Administrators' traders page Mockup

The Trading accounts details page can be seen in The figure below.

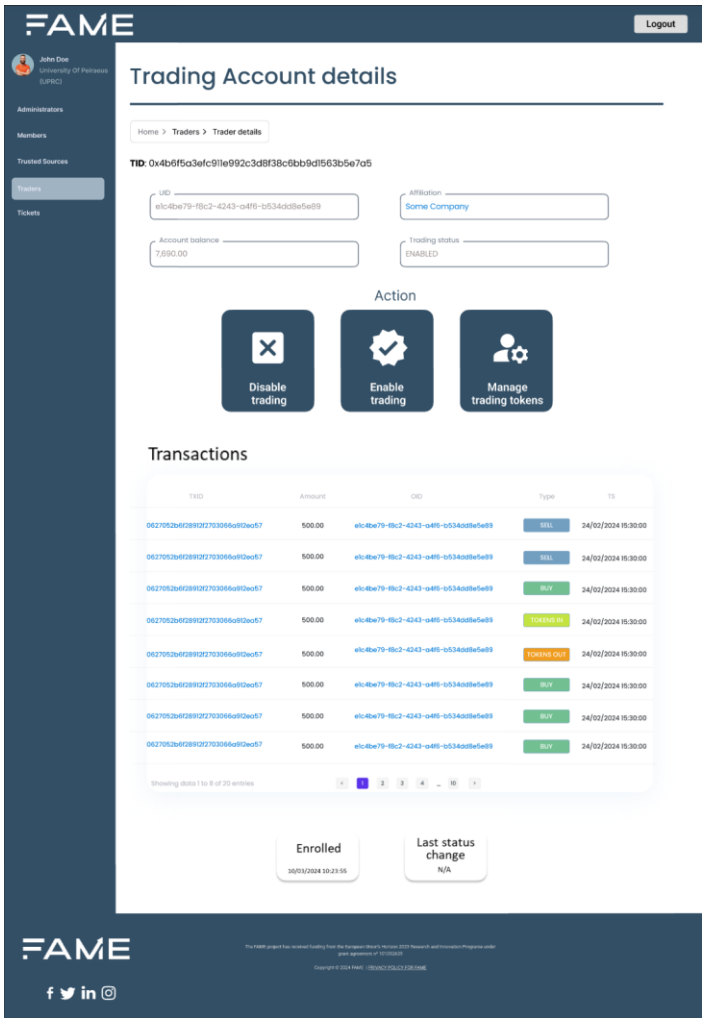

Figure 32: Administrators' trading account details page Mockup

The Trader details page can be depicted in The figure below.

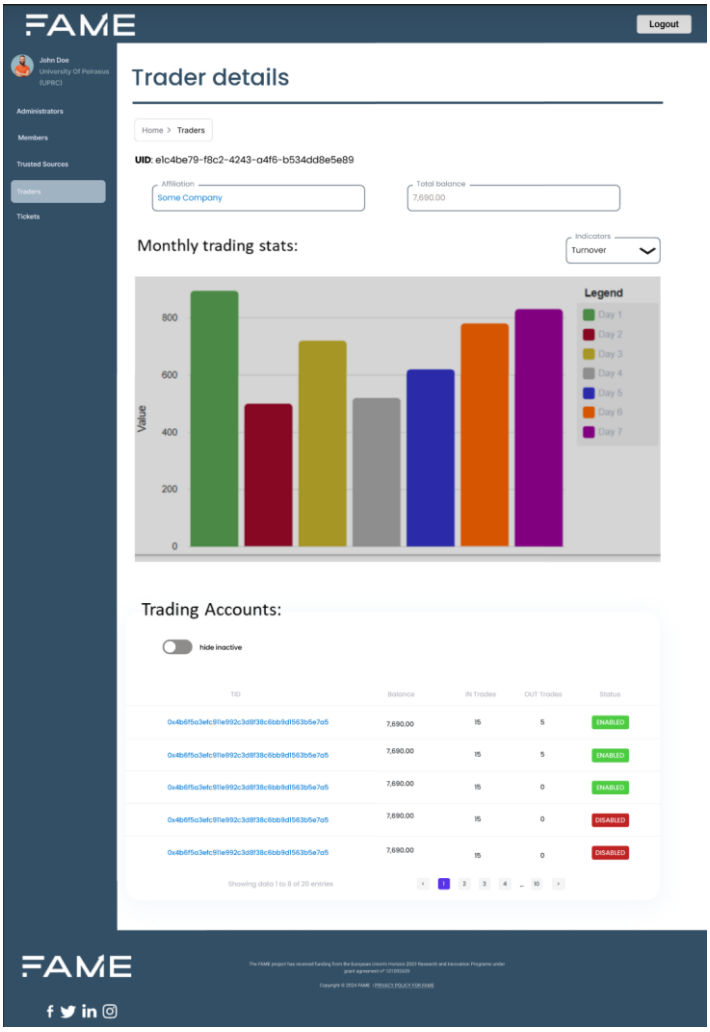

Figure 33: Administrators' trader details page Mockup

The Trading tokens page can be seen in The figure below.

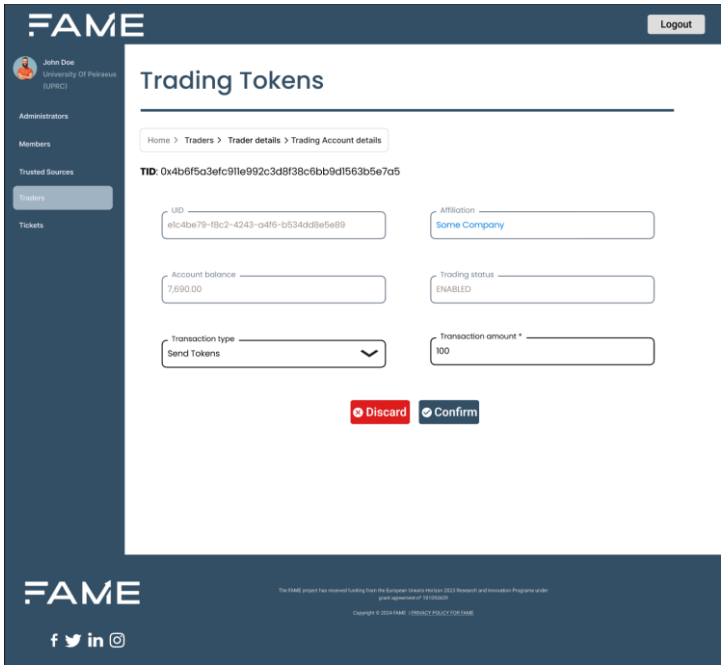

Figure 34: Administrators' trading tokens page Mockup

The Tickets page is provided in The figure below.

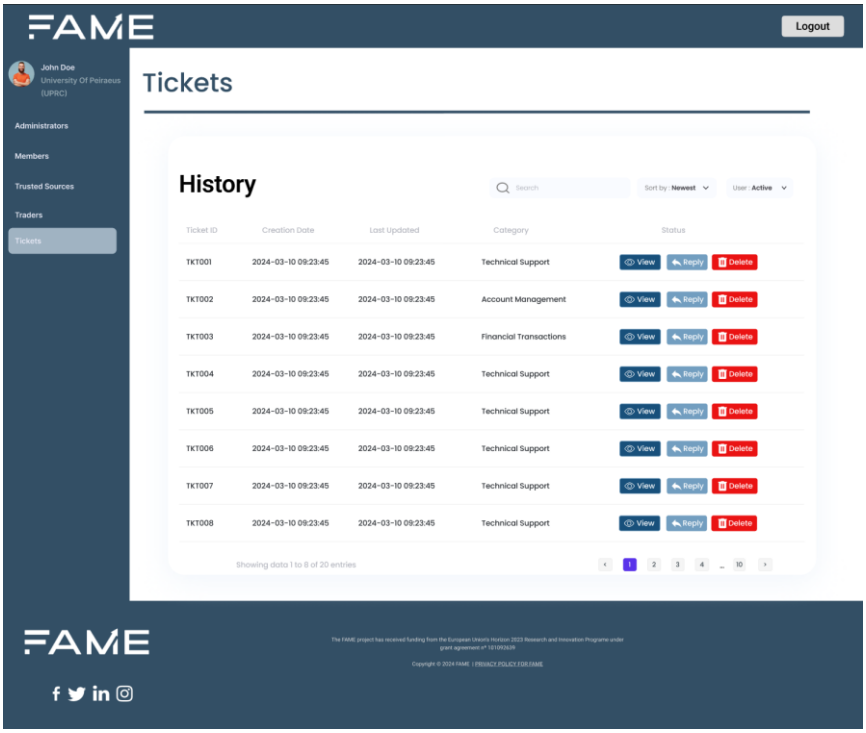

Figure 35: Administrators' tickets page Mockup

By the time that the Mockups were approved and finetuned based on all the stakeholders and FAME technical components' developers needs, the last phase of the FAME Dashboard initiated, namely the frontend development.

3.2.1.6 Step 6: Frontend Development

By the time that the Mockups design is finalized, the actual frontend development phase starts that deals with the preparation of the detailed design specifications and assets for the developers, so they can ensure that the final product closely matches the design vision. For the FAME Dashboard, since the complexity of the design was high, specific interaction guidelines were additionally provided that explain how interactions like hover effects, transitions, or animations should behave. Moreover, related style guides were generated, which serve as comprehensive documentation of the visual and interaction design guidelines, branding elements, and UI components used in the design. All in all, these style guides also helped into establishing and maintaining a consistent visual language across the FAME Dashboard. Finally, it should be noted that during the development phase some further updates were performed in the already provided Mockups (*Step 5*), to offer a fully compliant UI based on the relevant feedback. At this stage, all the different Mockups are already implemented, whereas their integration with the different backend services is taking place. Related information on how the FAME Dashboard UIs can be executed and experimented by any stakeholder are being provided in Section 3.2.2.3.

At this point, it should be noted that the step related with the "Testing and Iteration" phase was not yet performed as a final step, since multiple internal testing activities have taken place in the context of the different FAME Dashboard design phases (as explained in the previous Sections), and for the moment it would be considered as redundant and time consuming. Nevertheless, such step is considered of high importance since well-organized and targeted user testing plays a crucial role in the UI design process. The design might look ideal from where the developers' side, but it is important to remember that the developer is not the end-user. Testing the prototypes with real users helps into identifying any usability issues or pain-points that might have been missed along the way.

Furthermore, it helps into getting a sense of how the visual design strategy is resonating with real users and how it meets their expectations. For that reason, the consideration of the "Testing and Iteration" step as an independent phase in the overall FAME Dashboard design phase will be considered in the next and final version of this deliverable, to correctly manage and perform this step, gather related feedback, and analogously adapt and finalize the FAME Dashboard design.

3.2.2 Dashboard development process

3.2.2.1 Dashboard technologies

The FAME Dashboard will be released as a responsive web application, whose frontend part will primarily utilize **Angular** [16] (version – Angular v17.0.0) along with the following technologies:

- **Docker** [17]: Docker will be used to containerize the Angular application, facilitating easy deployment and scalability. By containerizing the frontend, we will ensure consistency across different environments and simplify the deployment process.
- **HTML** [18]: HTML will be used for structuring the content of the web application, providing a solid foundation for the presentation layer (version – HTML5).
- **SCSS** [19]: SCSS (Sass) will be employed for styling the web application. Its powerful features such as variables, nesting, and mixings will aid in maintaining scalable and maintainable CSS code (version – SCSS (Sass) v1.70.0).
- **TypeScript** [20]: TypeScript will serve as the primary programming language for building the frontend of the web application. Its static typing and modern syntax will help in writing cleaner and more structured code (version – TypeScript v5.4.2).
- **Angular Material** [21]: Angular Material will be utilized for styling and theming the web application. Angular Material provides a set of pre-built UI (User Interface) components following the Material Design principles, enabling us to create a visually appealing and consistent user interface (version – Material v16).
- **Npm Packages**: Additional libraries and frameworks will be exploited depending on the arising web application's requirements, ensuring the efficient development and delivery of the application. Such libraries or frameworks may include:
    - HttpClient: Angular's HttpClientModule for making HTTP (Hypertext Transfer Protocol) requests to a server. This package provides a simplified API (Application Programming Interface) for interacting with RESTful services.
    - RxJS: Reactive Extensions for JavaScript (RxJS) for handling asynchronous operations and managing streams of data.
    - Angular Router: While Angular's router comes built-in, additional features or enhancements may be achieved by utilizing npm packages related to routing and navigation.
    - Angular Forms Modules: Alongside the core forms' functionality, additional npm packages might be considered for form validation, custom form controls, or form layout management.
    - Angular CLI: Although not strictly a package, Angular CLI (Command Line Interface) provides various commands and utilities for scaffolding, building, and managing Angular applications. It is essential for efficient development workflows.
    - Testing Frameworks: Depending on the web application's requirements, npm packages for testing frameworks like Jasmine, Karma, or Protractor might be utilized for writing and running tests for Angular applications.

Furthermore, for providing a fully functional Dashboard, the latter will be also able to be integrated with technologies from external sources seamlessly. The following key features of Angular will play a significant role in achieving this integration:

- **Two-way Data Binding**: Angular's two-way data binding feature facilitates the automatic synchronization of data between the model and the view. This reduces the need for manual DOM manipulation, enhancing the efficiency of our application.
- **Component-Based Architecture**: Angular promotes the use of reusable and encapsulated components. This architecture makes it easier to manage and maintain complex UIs by breaking them down into smaller, manageable pieces.
- **Dependency Injection**: Angular boasts a powerful dependency injection system. It helps managing component dependencies, making our application more modular and testable, while enhancing code maintainability.
- **Directives**: Angular provides both built-in and custom directives. These directives enable developers to create custom behaviors and manipulate the DOM effectively, enhancing the flexibility of our application.
- **Routing**: Angular offers a robust routing module to handle navigation between views and pages in a single-page application. This feature ensures smooth navigation and enhances the user experience.
- **Forms**: Angular provides robust support for building both template-driven and reactive forms. This makes it easy to handle user input and validation, ensuring data integrity and enhancing user interaction.
- **Services**: Angular allows the creation of services to encapsulate and share functionality across components. This feature facilitates better management of data, API calls, and other common tasks, enhancing the overall efficiency of our application.

3.2.2.2 Dashboard design principles

Adhering to ***fundamental design principles*** is also quite crucial for the platform's success and user satisfaction. The essential core principles that are followed by all the FAME Dashboard pages are:

- **Simplicity**: Aim for a straightforward design, reducing complexity of the platform's content.
- **Consistency**: Maintain both visual and operational uniformity across all the platform pages to aid in user interaction.
- **Efficiency**: Structure the pages' information thoughtfully to facilitate effective navigation and use.
- **Accessibility**: Embrace user diversity by designing inclusive pages accessible to individuals of varying abilities.
- **Scalability**: Design with an eye toward future expansion and adaptability, streamlining future adjustments and maintaining platform coherence.
- **Memorable Design**: Create an engaging and memorable experience, trying to foster an emotional connection with the user.
- **Aesthetic and Minimalist Design**: Favor a design that is both visually appealing and minimalist, evoking a positive reaction and enhancing the perceived effectiveness of the platform.

The FAME Dashboard also adopts key principles of ***SEO (Search Engine Optimization)*** to enhance its digital footprint and ensure unparalleled visibility across search engines, prioritizing:

- **Proper Response Codes**: Ensure web pages return a 200 HTTP status code for successful rendering and update or remove unnecessary redirects to optimize user and crawler access.
- **URL Structure**: Craft URLs that are concise, descriptive, lowercase, and free of unnecessary characters, reflecting the platform's content hierarchy.
- **Mobile-Friendliness**: Optimize the pages' content for mobile viewing and Google's mobile-first indexing, ensuring the mobile site contains all the critical content.
- **Site Speed Optimization**: Focus on reducing page load times through efficient code, optimized images, and minimized use of heavy plugins.

In FAME's Dashboard development, adhering to ***green principles*** not only supports environmental sustainability but also enhances user experience. To incorporate these values, all of its pages support:

- **Minimalist Design**: Streamline interfaces to include only essential elements, reducing cognitive load and energy consumption.
- **Dark Colors**: Utilize darker color palettes to reduce energy usage on devices with OLED screens, promoting longer battery life.
- **Compressed Media Files**: Ensure media files are optimized for quick loading, conserving bandwidth, and improving access times.
- **Full-responsive Website**: Craft a design that adapts seamlessly to any screen size, minimizing the need for redundant resources across devices.
- **Clear and Concise Code**: Write efficient code that performs tasks without unnecessary complexity, reducing processing time and energy usage.

3.2.2.3 Dashboard source code

As stated in the previous Sections, the FAME Dashboard is a crucial component, providing users with an intuitive interface to interact with the platform's functionalities. This Section details the source code and deployment procedures for exploiting the FAME Dashboard, catering to both non-Docker and Docker environments. To this context, the users should follow the below instructions carefully to set up the FAME Dashboard correctly.

Initially, the following programs and libraries are required to be installed into the user's environment to be able to perform the rest of the subsequent instructions:

- WSL [22]
- Git [23]
- Node.js [24]
- Docker (non-required for section ***Non-Docker Environment***) [25]

Moreover, in order for someone to be able access and install the FAME Dashboard, related access credentials should be granted to the following repository: https://gitlab.gftinnovation.eu/

The versions of the required programs and libraries currently installed on the machine where the development takes place are the following:

```
C:\Users\user_fame> git --version
git version 2.32.0.windows.2
C:\Users\user_fame> node -v
v18.14.2
C:\Users\user_fame> npm -v
9.5.0
C:\Users\user_fame> nvm list
  * 18.14.2 (Currently using 64-bit executable)
    16.14.0
    16.13.1
```

```
C:\Users\ user_fame > docker info
Client:
 Context:    default
 Debug Mode: false
 Plugins:
  buildx: Build with BuildKit (Docker Inc., v0.6.1-docker)
  compose: Docker Compose (Docker Inc., v2.0.0-rc.1)
  scan: Docker Scan (Docker Inc., v0.8.0)

Server:
 Containers: 17
  Running: 5
  Paused: 5
  Stopped: 7
 Images: 19
 Server Version: 20.10.8
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
  userxattr: false
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Cgroup Version: 1
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
 Swarm: inactive
 Runtimes: io.containerd.runc.v2 io.containerd.runtime.v1.linux runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version: e25210fe30a0a703442421b0f60afac609f950a3
 runc version: v1.0.1-0-g4144b63
 init version: de40ad0
 Security Options:
  seccomp
   Profile: default
 Kernel Version: 5.10.16.3-microsoft-standard-WSL2
 Operating System: Docker Desktop
 OSType: linux
 Architecture: x86_64
 CPUs: 12
 Total Memory: 12.36GiB
 Name: docker-desktop
```

Non-Docker Environment

To execute the FAME Dashboard without using Docker, the user should ensure that Node.js version 18 is installed. If not, the NVM (Node Version Manager) [26] packager can be used to download and manage Node.js versions on the preferred machine. The related steps follow:

Step 1: Install Node.js

- In the case that Node.js 18 is not installed, NVM should be used to install it. The installation guide for Node.js and NPM should be followed on the desired operating system [27].
    - To install NVM on Windows, the instructions are as follows:
        - Download the NVM Windows installer from the NVM for Windows GitHub repository [26].
        - Execute the installer and adhere to the on-screen prompts.
        - After the installation finishes, open a new command prompt or PowerShell window.
        - Confirm the installation by running the following command: `nvm --version`

- If everything is set up correctly, you should see the NVM version number.

Step 2: Clone the repository

- The terminal should be opened to run the command: `git clone https://gitlab.gftinnovation.eu/fame/fame-dashboard.git`
- This command will clone the FAME Dashboard code into the local machine.

Step 3: Navigate to the project folder

- The directory must be changed to the project folder: `cd fame-app`

Step 4: Install the required dependencies

- The following command should be run to install all the necessary Node modules: `npm install`

Step 5: Serve the application

- The application can start with the command: `ng serve --open`
- This command will automatically open the default browser and it will accordingly be navigated to ***localhost:4200***, where the Dashboard will be running.
- The `--open` flag is optional and will automatically open the default browser to ***localhost:4200***. If it is preferred for the browser to open manually, the following command can be executed: `ng serve`

Docker Environment

In the case that Docker is preferred to be used, the following steps outline how to set up and execute the FAME Dashboard using Docker containers.

Step 1: Clone the repository

- As in the case of the non-Docker setup, the repository must be cloned onto the local machine: `git clone https://gitlab.gftinnovation.eu/fame/fame-dashboard.git`

Step 2: Add docker configuration

- The `docker-compose.yml` file must be added to the root level of the cloned repository. The structure should look like this:

  ```
  ├── fame-app
  ├── docker-compose.yml
  ```

Step 3: Run docker compose

- The user should navigate to the root level of the repository where the docker-compose.yml file is located, and execute the command: `docker-compose up -d`
- This command will build and start the Docker containers in detached mode.

Step 4: Access the Dashboard

- The browser must be opened, and the user should navigate to ***localhost:4200***. The FAME Dashboard will be running, being fully accessible.

By following these steps, a user can set up and run the FAME Dashboard either with or without exploiting Docker, depending on the environment preferences. This flexibility ensures that anyone can work with the FAME Dashboard in a way that best fits the overall development setup.

3.3 Integration of Dashboard & Backend Services

3.3.1 Integration methodology

The FAME Dashboard will put in place a RESTful API communication among the FAME platform's frontend and backend parts, which will leverage the HTTP protocol to facilitate communication between those two (2) layers. Exploiting RESTful APIs that are characterized by their stateless operation and reliance on standard HTTP methods (GET, POST, PUT, DELETE), will offer a flexible way to fetch, create, update, and delete data from the platform. This will allow the frontend part of the FAME platform to dynamically display, and update information based on the user's interactions and the backend data changes.

A concise guide aligned with the Angular framework and the integration of the two (2) layers (frontend, backend) through RESTful APIs is provided below:

- **API**: Defines rules and protocols for the software applications/services/components to interact.
- **HTTP**: Transfers data over the web, enabling communication between clients (e.g., browsers) and servers.
- **Endpoints**: Represent specific resources on the server, used in RESTful APIs to perform actions like data retrieval, creation, update, and deletion.
- **HTTP Methods**: Various methods used in RESTful APIs for different operations:
    - GET: Retrieve data from the server.
    - POST: Send data to create new records.
    - PUT: Update existing records.
    - DELETE: Remove records from the server.

The complete process for accomplishing integration with RESTful APIs is outlined below:

- *Step 1*: Setting up backend API
    - Choose a backend technology stack such as Node.js with Express or Django for Python, ensuring compatibility with Angular.
    - Define routes and endpoints for the API, following RESTful principles.
    - Implement business logic and database operations within the API endpoints.
- *Step 2*: Sending requests from Angular frontend
    - Utilize Angular's built-in HttpClient module or libraries like Axios to send HTTP requests from the frontend.
    - Construct the appropriate request method (GET, POST, PUT, DELETE) and headers.
    - Include necessary data in the request body or as query parameters.
- *Step 3*: Handling responses in Angular frontend
    - Subscribe to the observable returned by HttpClient methods to receive responses from the backend.
    - Parse and process the response data received from the backend.
    - Update the Angular components and UI based on the response data.
    - Implement error handling to manage any potential errors or exceptions.

As a final note for the integration approach, the FAME Dashboard will embrace an API-First approach [28] to foster collaboration and streamline development with the various underlying backend parts of the platform. This means that developers will have to prioritize the design and development of the APIs at the beginning of the project. This strategy will ensure that the APIs are robust, well-documented, and ready to support various frontend applications (web, mobile, etc.) from the beginning. By focusing on the API-First approach, FAME will encourage a more modular design, allowing teams to work in parallel and making the platform more adaptable to future requirements.

Thus, a more collaborative approach will be put into practice, where the frontend and the backend teams will work together to design the APIs. Thus, implementations will take place simultaneously as the frontend teams can use any mock backend for their development work.

3.3.2 Frontend & Backend Integration

For efficiently achieving the integration between all the frontend and the backend parts (Table 1) of the FAME platform, all those parts will follow the abovementioned integration methodology for being organized following the RESTful APIs and the API-First integration principles. More specifically, the steps that are followed are depicted below:

- *Step 1*: Start with API design
  - Before diving into the frontend development, developers should define their API requirements based on the functionalities they aim to support. This includes identifying the resources, actions, and data formats needed.
  - In this phase, the developers must provide the postman exported APIs' collection [29], and provide a related readme file with a concrete (local) installation guide for each API, following the below example:
    - Description of service endpoint: Count Items in a Specific Time Range
    - API Documentation: https://example.com/api-docs/#/Items/getItemsCount
    - HTTP Method: GET
    - API Endpoint: https://api.example.com/items/count
    - Query Parameters:
      (i) time_from: Required. Start of the time interval (in seconds)
      (ii) time_to: Required. End of the time interval (in seconds)
    - Example Request:
      https://api.example.com/items/count?time_from=10&time_to=16
    - Example Response: {

```
"status_code": 200,
"status": "success",
"count": 150
}
```

- *Step 2*: Ensure proper documentation and versioning
  - It must be ensured that all APIs are thoroughly documented and versioned. This aids in maintenance and scalability, as new features can be added with minimal impact on existing functionalities.
  - In this step, the developers must document and version their APIs exploiting one of the following (or similar) tools:
    - SwaggerHub [30]: An API editor compliant with Swagger, also supporting AsyncAPI.
    - Online Swagger editor [31]: A browser-based editor for creating and testing Swagger/OpenAPI specifications.

- *Step 3*: Develop in parallel
  - With a clear documentation API (based on *Step 1* and *Step 2*), frontend and backend teams can work simultaneously, speeding up the development process. The frontend teams can use mock APIs or API stubs to start the development even before the backend is fully implemented.

- *Step 4*: Perform iterative testing

  - Regular tests must occur to validate the integration between the frontend and backend using the RESTful APIs to catch and fix issues early in the development cycle.
  - Establish a feedback loop between the frontend and backend teams to continuously refine the APIs based on real-world usage and frontend requirements.

- *Step 5*: Communicate with the integration team
  - It is recommended that developers subscribe to the Slack dedicated channel for directly communicating with the rest of the integration development teams, so as to solve any raised issues/technical points.

3.4 Integration of Backend Services

3.4.1 Integration methodology

In the first place, we should clarify what the integration of backend services is about, and in what it differs from the Dashboard / Backend integration that was described in section 3.3 of this document. In a nutshell, the former is also called *internal integration*: its goal is to enable cross-module communication for those workflows that require the *orchestrated cooperation* of multiple platform modules.

The most elaborate example of such workflow is "Asset Publishment" (see deliverable "D4.1Blockchain-based Data Provenance Infrastructure" [8]): although the starting point is a call to an Open API service endpoint provided by the Provenance and Tracing (P&T) module, the workflow then proceeds in the background with the involvement of the Federated Data Asset Catalogue (FDAC), Asset Policy Management (APM) and Operational Governance (GOV) modules.

In this scenario, P&T acts as the *initiator and orchestrator*, while FDAC and GOV are *contributors*. All cooperative workflows supported by the FAME platform follow this pattern, with one initiator/orchestrator and one or more contributors. Internal integration is thus the development, testing and deployment of *non-public* service endpoints (i.e., deployed on the common hosting environment of the FAME platform's backend but not accessible from the public Internet) that serve the needs of cooperative workflows.

A much simpler but more generally applicable example is user authentication / authorization: every Open API service endpoint, regardless of the module, needs to ensure that their callers are actually authorized to consume the provided service. In order to do that, the caller of a service endpoint must provide a valid Verifiable Onboarding Credential (VOC) token issued by a trusted Onboarding Authority.

While the details of the onboarding and authentication process are not in the scope of the present document (see WP3 deliverables for more information on that), suffice to say that all Open API service endpoints need to integrate with the Authentication and Authorization Infrastructure (AAI) platform module.

3.4.2 Backend Integration

All non-public service endpoints follow the same approach than public ones: REST[2] architectural style on top of the HTTP communication protocol.

A significant difference between the two groups is that non-public service endpoints can only be called by other modules running in the same hosting environment. Access control is thus implemented by means of networking rules rather than through the integration with the AAI module. Moreover, as all communications travel internally to the hosting environment, the HTTP channel used by non-public service endpoints is *not* encrypted.

A lesser difference - mostly cosmetic - is the pattern used in HTTP *paths*. As seen below, the path of Open API operations starts with the "api" element, which in non-public ones is replaced by the acronym of the module.

- Public service:
  https://<server-address>/api/<api-version>/<operation-specific-path>
  Example: [POST] https://api.fame-horizon.eu/**api**/1.0/members
  (Open API: register a new Federation Member - Access control: ADMIN profile required)

[2] REpresentational State Transfer - see https://www.codecademy.com/article/what-is-rest

- Non-public service:
  http://localhost:<port-number>/<module>/<api-version>/<operation-specific-path>
  Example: [GET] http://localhost:4333/**tm**/1.0/check-clearance
  (Trading and Monetization module: check if the user has data access tokens)

The following interaction diagrams show the two major workflows where cross-module communication is required. The internal integration points are presented in the context of the overall process and are highlighted by a red box.

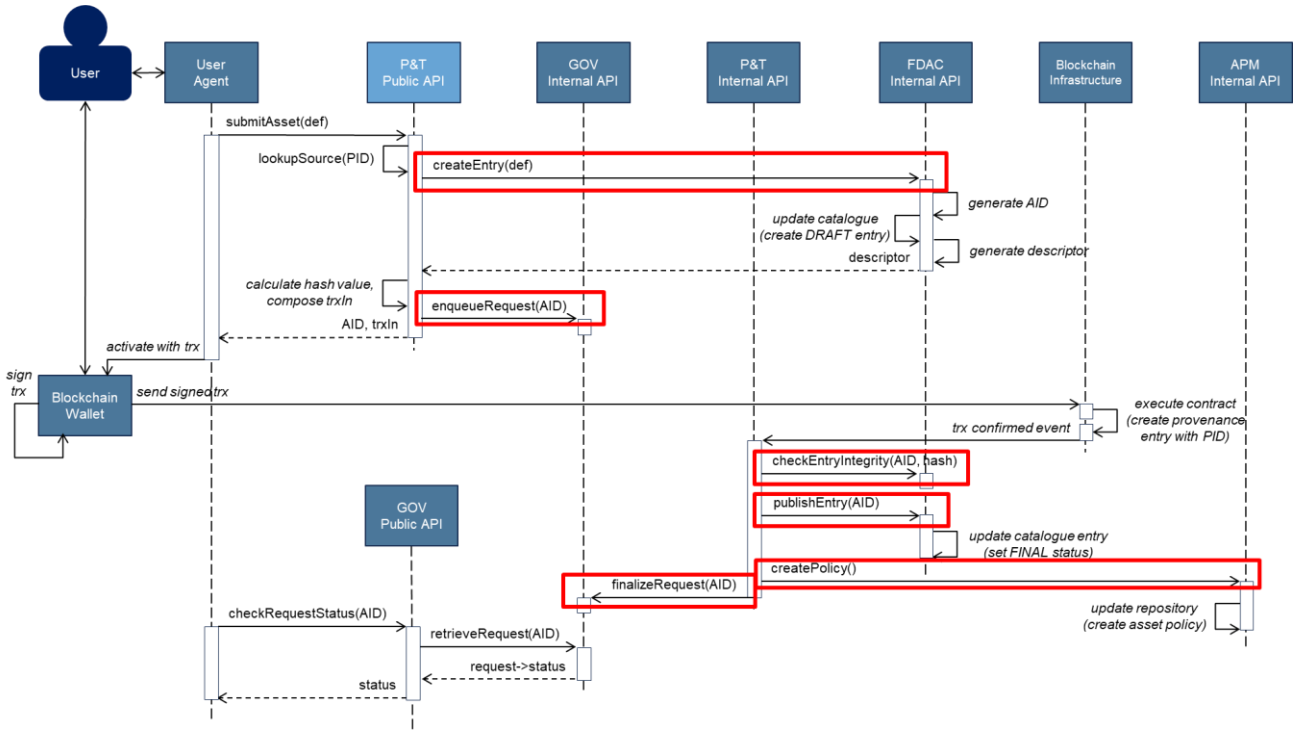

Figure 36: Asset Publishment

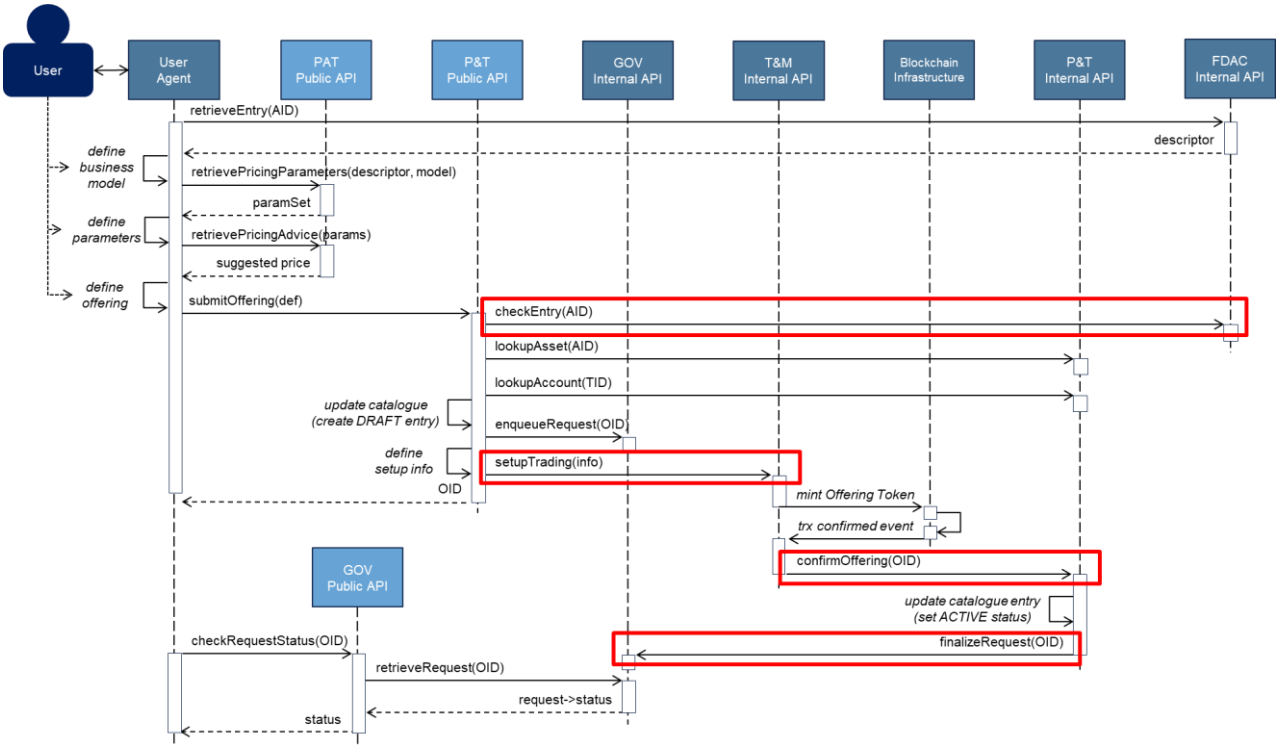

Figure 37: Offering Publishment

With regards to the integration of the FDAC module, it should be noted that the above diagrams are representing the final platform design (*to-be*) rather than the current implementation (*as-is*). This is due to the fact that the current implementation is a minimum viable product that had to cut some corners to meet its deadline, and that FDAC is a pre-existing independent system that runs in a separate runtime environment, so that the standard approach for the deployment of non-public services (e.g., the `createEntry` operation) could not be adopted. Instead, FDAC's integration points are deployed as public services that are only accessibly with a system-owned API token.

Internal integration points are documented in the same way as FAME's Open API: online by means of the Swagger UI[3] and offline in the relevant project deliverables. The only exception to this rule is for closed-source modules - namely, FDAC and APM - that only provide offline documentation. Here below we provide a screenshot of the Swagger UI, documenting the internal integration points provided by the P&T, T&M and GOV modules.

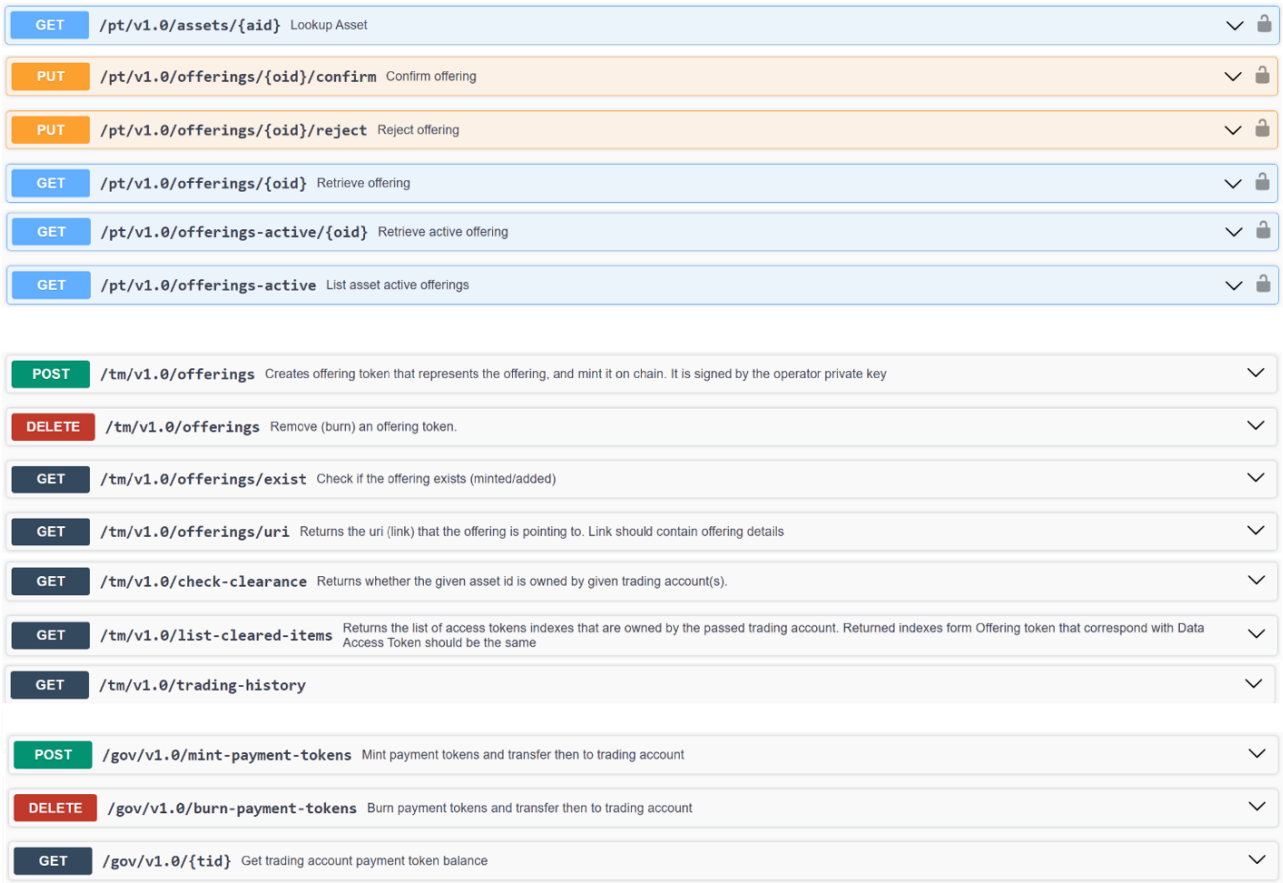

Figure 38: Service endpoints for internal integration

[3] See https://swagger.io/

4 FAME Data Assets Integration

4.1 Introduction

This chapter illustrates the process of integrating Data Assets within the FAME platform. Data Assets can take various forms, such as Data Sets, Databases, Marketplaces, and DataSpaces, including both static and dynamic components. Integration into the platform refers to the process of acquiring, indexing descriptive metadata, and the necessary integrations for their valorization and commercialization.

In other words, integration involves incorporating Data Assets into the FAME ecosystem, ensuring they are correctly identified, organized, and made available for use by users. This process involves gathering descriptive information about Data Assets, such as title, author, creation date, and other relevant information. These metadata are then indexed and used to facilitate searching and discovery of assets within the platform.

Additionally, integration may also require implementing additional features to valorise Data Assets, such as the ability to add tags, categories, or other annotations to enhance their visibility and accessibility. Furthermore, technical integrations may be necessary to ensure compatibility and interoperability between different types of assets and the functionalities offered by the FAME platform.

Finally, once integrated into the platform, Data Assets can be commercialized through the FAME marketplace, allowing users to buy, sell, or exchange data securely and reliably. This commercialisation process may include defining pricing policies, managing transactions, and providing support to users interested in purchasing or using Data Assets available on the platform.

The component of the FAME solution that manages data assets is the Federated Data Asset Catalogue or FDAC. It has been integrated into the FAME backend. The following of this section provides the high level description and the API integrated to the marketplace.

4.2 Federated Data Asset Catalogue (FDAC)

The FDAC functions as a registry within FAME, tasked with systematically cataloging and archiving various assets. These assets include datasets, AI models, services, and essential documentation, ensuring a structured approach to data management. The registry can represent assets originating from FAME activities and index assets from external sources such as relevant Data Spaces or Data Marketplaces. All asset information is accessible to any FAME component requiring it through well-defined interfaces.

4.2.1 Technical Specification

4.2.1.1 Component-level C4 Architecture

This section outlines FDAC’s main functionalities, which include storing and indexing asset metadata and providing interfaces to support CRUD (Create, Read, Update, Delete) actions on these assets. These functionalities are supported by the "Asset Management" and "Database" components shown in Figure 43. Additionally, the "Search" component enhances user discovery functionalities. Other FDAC components (External Source Manager, External Source Resolver, and Policy Extractor) will be detailed in the next document version.

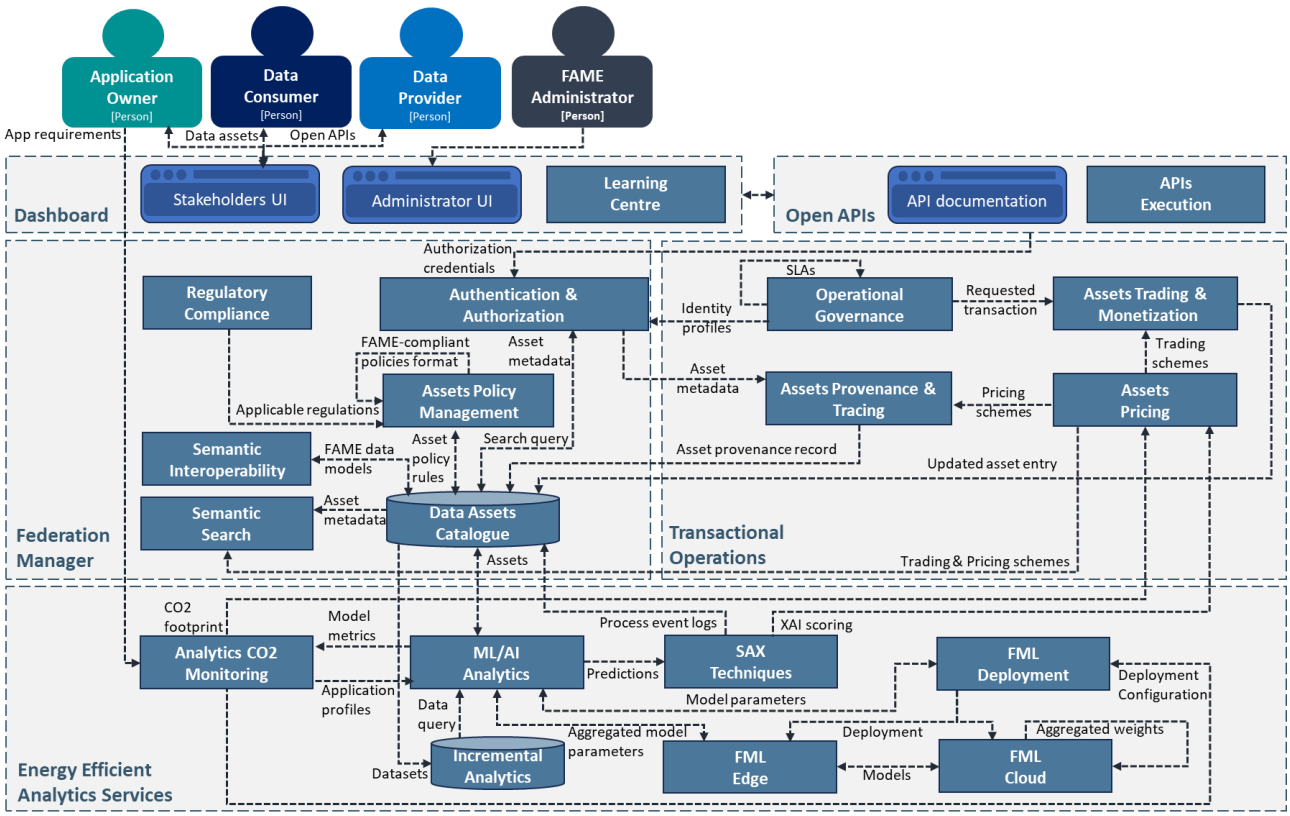

Figure 39: FDAC in C4 Architecture

4.2.1.2 Asset Manager

The Asset Manager handles the creation, deletion, and modification of assets. It is based on FDAC and has been extended to support CRUD actions via REST API, allowing the addition of information produced by external tools.

REST API The REST API facilitates data exchange with the FDAC provided by partner UNPARALLEL. The APIs use REST (Representational State Transfer) for better integration with heterogeneous components. The FDAC REST API allows users to manipulate all FDAC assets via GET, POST, PUT, or DELETE requests. Authentication is required to use the API.

The search endpoint supports free text search with optional parameters, allowing results to be filtered by manufacturer, developer, owner, tags, or components. Additional parameters like 'expand' and 'output filter' refine search results further.

Interfaces FDAC's REST API Swagger interface displays available endpoints, required and optional parameters, and example responses. A sample of the interface is shown in the figure below, enabling users to test the API and explore its functionalities.

Figure 40: FDAC REST API Swagger

4.3 Data Asset Management

The FDAC endpoints, which allows users to query FDAC for specific components are presented below. Through the Dashboard the users can describe a query by defining terms and filters to refine results.

The following API are actually integrated into the Dashboard to provide the functionalities provided to end-users:

Creating an Asset FDAC provides two POST REST endpoints for adding new assets, depending on the data model (FDAC or DCAT).

Component Endpoint This endpoint is used to add a new component using the internal FDAC data model. A successful addition returns the asset ID; otherwise, an error code and message are provided.

Interoperability Component Endpoint This endpoint allows adding a new asset by defining a data model in JSON, such as DCAT. The FDAC instance converts the asset from the DCAT structure to the internal FDAC data model using the Semantic Interoperability Middleware.

Searching (via REST API) This endpoint allows users to search for assets on the FDAC instance. The response includes the number of results and an array with search results, each featuring a "weight" property indicating the relevance of the search term.

5 FAME CI/CD Infrastructure for Integration

5.1 Introduction

This section explores the complexities of combining various software components, or packages, created from different project tasks within the FAME Solution Architecture. Several key challenges emerge:

- **Technology diversity:** Integrating modules built using disparate technologies necessitates a flexible solution that can accommodate different systems.
- **Data interoperability:** Seamless data transfer between these diverse modules is crucial, requiring a non-centralized architecture that promotes information sharing.
- **Modular design:** A modular platform structure is essential for adaptability, scalability, and efficient development.
- **Governance and DevOps:** A robust governance framework, coupled with DevOps practices like Continuous Integration/Continuous Deployment (CI/CD), is vital for effective implementation, automation, integration and quality assurance.

The FAME Solution Architecture highlights the significant challenges posed by integrating modules developed using varying technologies. To overcome these obstacles, a multifaceted approach is required. This includes a solution capable of handling diverse components, ensuring smooth data flow through a non-monolithic structure, and adopting a modular platform design. By implementing a comprehensive governance framework and leveraging DevOps methodologies, FAME development and integration was streamlined, with enhance quality, and has delivered a flexible, scalable solution.

5.2 Microservices Approach

The microservices approach is a software development methodology where applications are built as a collection of small, independent, and loosely coupled services. Each service is responsible for a specific business function and can be developed, deployed, and scaled independently.

The microservices approach promotes agility, scalability, and resilience in software development, making it well-suited for building complex and rapidly evolving applications.

This methodological change has both advantages and disadvantages. The advantages can be summarized as:

- **Simple to develop**: Each microservice is independent and small.
- **Simple to upgrade**: Since each microservice is independent it’s possible to upgrade each component independently.
- **Simple interaction**: Each microservice communicates with each other through well-defined and standard interfaces (API).
- **Simple to scale**: Each microservice can be scaled independently.
- **Scale on demand**: It is possible to run multiple microservices behind a load balancer to scale on demand request.
- **Technology Diversity**: Microservices can be built using different programming languages, frameworks, and databases, depending on the specific requirements of each service.

The disadvantages of the methodological change can be summarized as:

- **Complexity**: Splitting an application into multiple independent microservices increases the complexity of the deployment process.
- **Monitoring**: Monitoring each microservice requires having many metrics and logs to manage it.
- **Performance**: All communications occur on the network so these are slower than memory communications.
- **Debugging**: a Monolithic application is much easier to debug and test due to the fact it is composed by single indivisible units.

5.2.1 Microservices with containers

The spread of containers led, as already mentioned, to the necessity to change the approach of the developers in the creation of an application, moving from design of monolithic applications, where the various components (generally UI, Business logic and Data-layer) were strongly coupled among them, to microservices applications where the various components are decoupled from each other.

Figure 41 : Monolithic vs Microservice

Our approach for a rapid iterative development of a microservice based software infrastructure builds upon a widely used methodology like DevOps, as will be described in Chapter 5 .

In the last years there has been a strong transformation in conceptualizing benefits of containers, similar to what happened in the early 2000s with the advent of virtualization, due to containers' spread which led to rethinking both how to manage the infrastructure and how to design and build the applications.

The introduction and the wide spread of containers concept and technologies have made it possible to improve the computational management of infrastructures, thanks to the possibility of removing the overhead generated by the use of the hypervisor (integrated software that allows to virtualize the HW resources of a server and make them available among several applications) and through the usage of the functionalities already available within the Linux OS kernel, as in Figure 2.

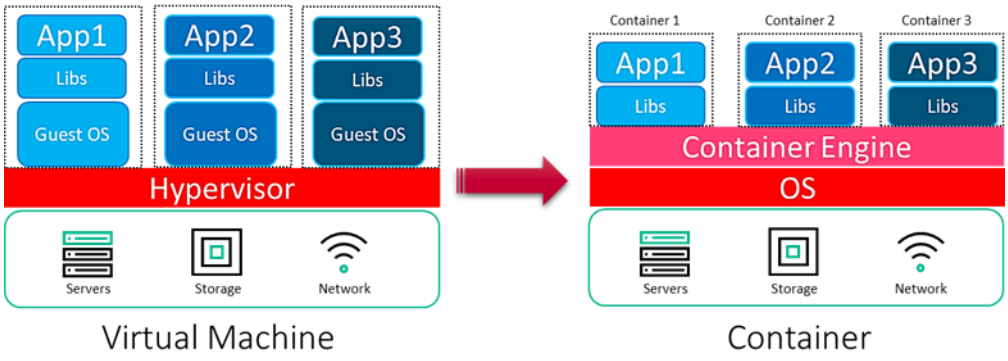

Figure 42 : VM vs Container

Docker, a widely used container engine, leverages Linux Control Groups and Linux Namespaces to create isolated container environments similar to virtual machines (VMs).

- **Linux Control Groups** allocate specific amounts of system resources—such as CPU, memory, disk I/O, and network—to each container, preventing resource exhaustion and ensuring fair sharing.
- **Linux Namespaces** isolate containers from each other and the host system by providing distinct views of system resources like process IDs, network stacks, and file systems.

This combination of technologies enables Docker to efficiently run multiple containers on a single host without compromising security or performance. An example is given in the figure below.

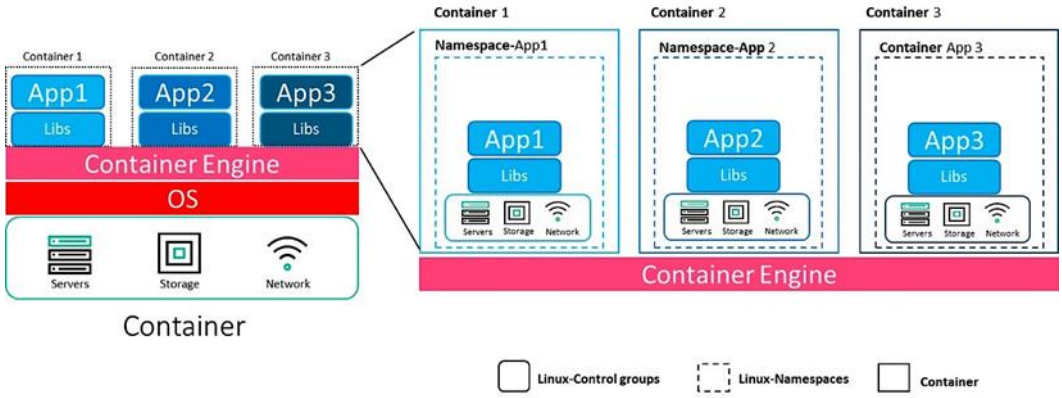

Figure 43: Container kernel properties

The main advantages to using containers to respect VMs can be summarized in the following macro points:

- Size: A container is small.
- Overhead: No full OS is required.
- Speed: Boot time is faster.
- Scaling: Real time provisioning.

Overall, container architecture offers many benefits, including improved portability, scalability, efficiency, and consistency for deploying and managing modern applications.

At the same time, to take advantage of containers it is necessary to rethink and redesign the currently monolithic applications as microservices based applications.

5.2.2 Kubernetes containers orchestration

The arrival on the market of containers and related microservices based applications, on the one hand enabled applications that quickly scale according to the requirements and that could be easily updated, on the other hand meant that software previously managed as a single indivisible piece was split into several dozen microservices (containers), making it more difficult to manage them.

In this context, the necessity to develop a tool that was able to manage the life-cycle of the microservices (deployment, scaling, and management) arose: such a tool was developed by Google with the name of "Project Seven of Nine "and released as open-source software in 2014. Today that tool is widely known as Kubernetes. For an outline of the entire Kubernetes solution, see the documentation on kubernetes.io.

The following bullet points are provided with the sole scope of highlighting the features Kubernetes provides to developers and integrators of the FAME platform using the FAME CI/CD infrastructure.

- **Service discovery and load balancing**: Kubernetes can expose a container using the DNS name or using its own IP address. If traffic to a container is high, Kubernetes is able to load-balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration**: Kubernetes allows to automatically mount a storage system of different types, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks**: a developer can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, a developer can automate Kubernetes to create new containers for deployment, remove existing containers and adopt all their resources to the new containers.
- **Automatic bin packing**: Kubernetes works with a cluster of nodes that are used to run containerized tasks. Kubernetes is configured with how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing**: Kubernetes restarts containers that fail, replaces containers, kills containers that do not respond to your user-defined health check, and does not advertise them to clients until they are ready to serve.
- **Secret and configuration management**: Kubernetes stores and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. Secrets and application configuration can be deployed and updated without rebuilding container images, and without exposing secrets in the stack configuration.

5.2.2.1 Kubernetes architecture

A Kubernetes (aka **K8s**) cluster is made up of two macro blocks, the first one called Control Plane (Master) and the second one called Data Plane (Worker).

The Control Plane constitutes the brain of the cluster and internally it is made up of the following components:

- **Kube-APIserver** is the component that exposes cluster API and truly is the main component, since Kubernetes has been designed and built to base all the operations on the use of the API.
- **Etcd** is a key value database that maintains all information relating to the status of the cluster.
- **Kube-scheduler** schedules which nodes of the Data Plane runs the containers (in Kubernetes named POD, the smallest deployable units of computing that can be created and managed in Kubernetes) based on the resources required, cluster status and affinity and anti-affinity rules.

- **Kube-controller manager** consists of a set of control processes which:
    - Check if cluster nodes are active.
    - Check if number and status of running POD is aligned with the required one, in case some POD became unavailable for example.
    - Control and create tokens to access on the K8s resource.
    - Populates the Endpoints object (that is, joins Services & PODs).

The Data Plane is the part where the workload is carried out, i.e. where the PODs are put into execution and it is characterized by the following components:

- **Kube-proxy** is a proxy that allows communication to the PODs from within and outside the cluster.
- **Kubelet** checks that PODs are running.
- **Container runtime (engine)** is software responsible for running containers; Kubernetes can use any CRI-O implementation like: Docker, CRI-O and containerd.

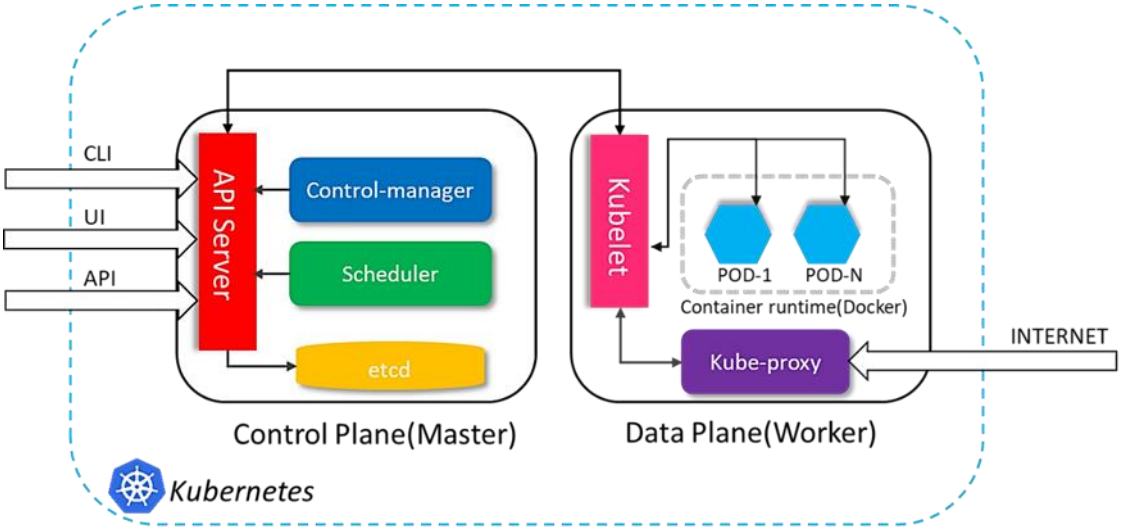

Figure 44 : Kubernetes

For the purpose of this chapter it is sufficient to outline two main concepts available in Kubernetes that we will use later to implement the Sandbox concept in the FAME integration process.

1. **Namespaces**: They are a logical grouping of a set of Kubernetes objects to whom it's possible to apply some policies, in particular:
    - **Quote** sets the limits on how many HW resources can be consumed by all objects.
    - **Network** defines if the namespace can be accessed or can access to other Namespaces, in other word if the Namespace is isolated or accessible.

    Different policies can be given to different namespaces.

2. **POD**: This is the simplest unit in the Kubernetes object. A POD encapsulates one container, but in some cases (when the application is complex) a POD can encapsulate more than one container. Each POD has its own storage resources, a unique network IP, access port and options related to how the container/s should run.

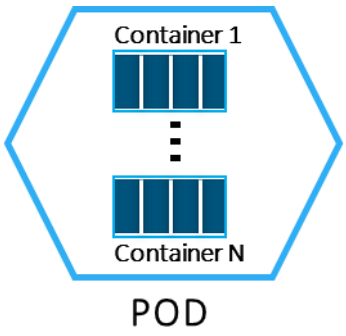

Figure 45: Graphical view of a POD in Kubernetes

5.3 Development view

The FAME CI/CD infrastructure with its components and relationships is depicted in the following figure.

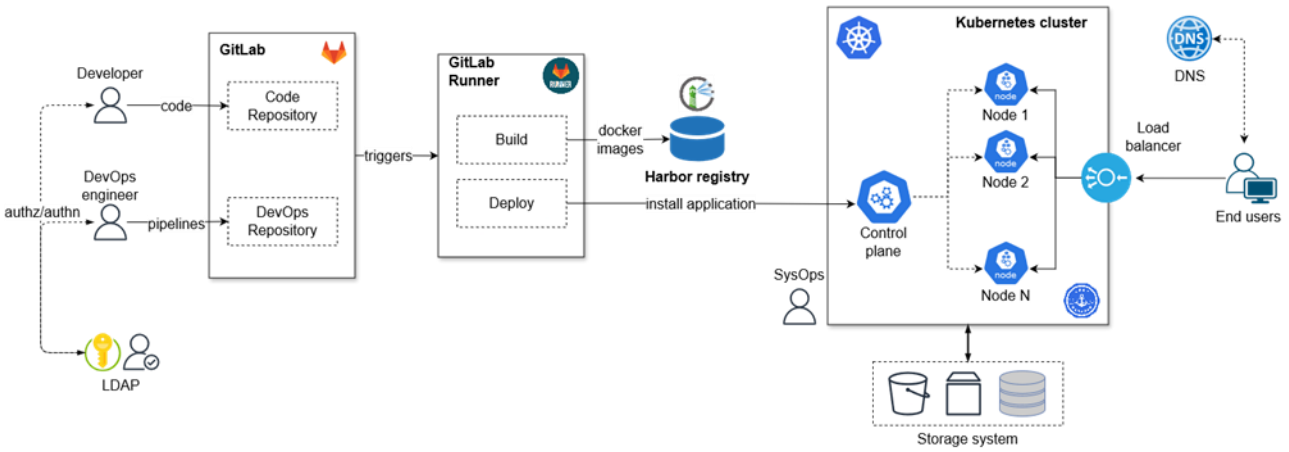

Figure 46: FAME CI/CD Infrastructure - Logical View

In the current implementation, the Kubernetes cluster is composed of 1 control plane node and 6 worker nodes. The cluster uses a storage system for the data persistence of the applications. This system can use different storage types like: object stores, block devices, network drives, file systems, etc.; the storage can be assigned to a specific node or shared among the cluster nodes.

The FAME CI/CD Solution uses the following application/tools:

- **GitLab**, the Source Code Version application, used to store and version the application source code. The CI/CD pipelines are stored in GitLab as well; pipelines are executed in GitLab using specific *runners*.
  A runner is basically a process which is launched to run the pipeline jobs, and is terminated when the jobs are completed.
- **Harbor**, the docker-compatible image registry, used to push and pull application images, maintaining version history.
- **LDAP server**, used to store user credentials and for SSO access to the other applications like GitLab and Harbor

- **Kubernetes Dashboard**, to view resources in the cluster and useful for troubleshooting and debugging, providing access to the logs of the application Pods
- **Cert-Manager**, a certificate management tool used to manage, generate and automatically renew the SSL certificates provided to secure the applications exposed to Internet

To develop and maintain the applications in the FAME CI/CD platform, three different roles are required:

- **Developer**, which is responsible for the application code development, testing, and versioning in the proper repositories. The developer also has the task of writing the Dockerfile of the application to be implemented and deployed

- **DevOps Engineer**, which is responsible for writing the CI/CD pipelines in GitLab, as well as the Kubernetes manifest files or Helm charts.
- **SysOps Engineer**, which is in charge of the Kubernetes cluster and DevOps tools maintenance, as well as other administrative and operational tasks like resources configuration, namespaces quota and limits settings, etc.

In questa sezione vengono . They must be registered in the platform with email and password, and these credentials are stored in a LDAP server. The registration process of the users in the platform is called *onboarding*.

The user access to the other DevOps applications, like GitLab, is guaranteed by the same credentials stored in the LDAP server.

5.4 Deployment view

In the FAME CI/CD Architecture, each application is deployed in a Kubernetes cluster in its own namespace, in order to achieve a good degree of isolation from other applications. Deploying applications in different namespaces allows a better control of resources in terms of CPU, memory and storage assigned to each namespace. In fact, it's possible to set limits and assign resources quotas for each namespace.

This organization model for the applications also facilitates monitoring and troubleshooting. If necessary, complex applications could be deployed in more than one namespace, but the isolation principle between namespaces remains valid.

The following picture illustrates the isolation principle between namespaces.

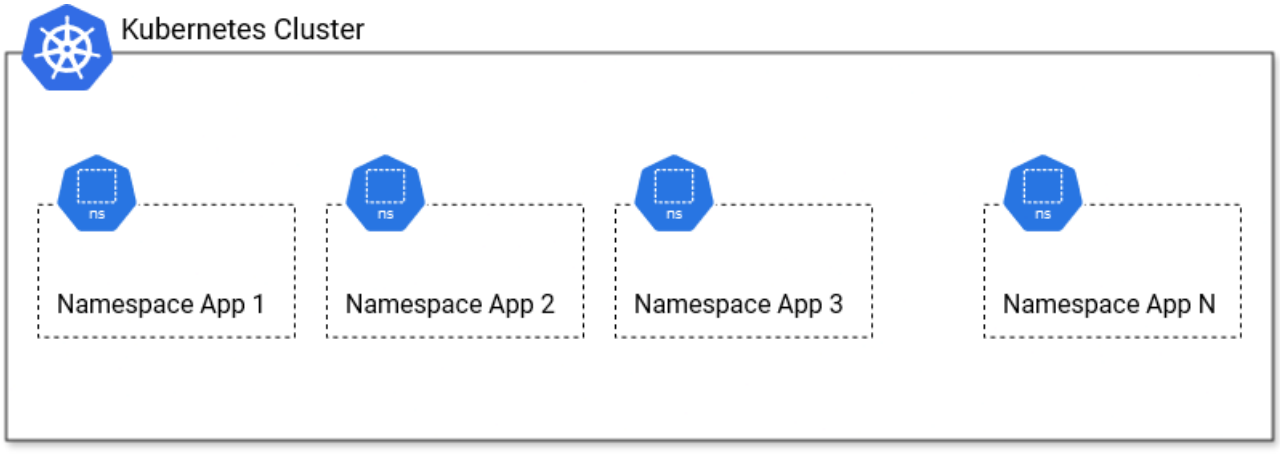

Figure 47: Kubernetes Cluster

A specific stage in a GitLab pipeline is in charge of the continuous deployment phase (CD). Once the application is properly built and the application images are pushed in the registry, the deployment stage of the pipeline is executed and, as a result, the application is deployed in the cluster.

The deployment stage of the pipeline is automatically executed in a GitLab runner after the previous build stage is successfully completed. In this way, each change in the code or configuration, which is committed in the source code repository in GitLab, produces the deployment of a new version of the application in the cluster. No manual tasks are needed in this phase; however, when necessary, GitLab also allows to manually rerun pipelines, either completely from the beginning or specific stages only.

Basically, the deployment can be performed in two different ways: either by applying Kubernetes manifest files, or by installing helm charts. The instructions to perform these actions must be written in the specific deployment stage of the pipeline.

5.4.1 Exposing the application

To expose applications and services to Internet, several components are involved in the platform.

A specific Kubernetes Ingress resource must be created and configured for each application or service. The Ingress resource uses a specific and well-known Ingress Class for all the applications (pilots), independently of the namespace in which they are allocated.

A dedicated Load Balancer is used to manage traffic among different applications and services, through the Ingress component installed in the cluster .

SSL certificates must be created and installed for each application or services to allow secure communications. The component involved in the creation and management of the certificates is the Cert-Manager application above mentioned.

Finally, a DNS record must be created and configured for the application; this record points to the cluster load balancer. The load balancer recognizes the host requested and route the traffic to the proper application or service.

5.4.2 Storage and Volumes management

A given amount of storage can be set for each namespace, by configuring specific resources in the Kubernetes cluster, called PersistentVolumes. This kind of resource is bound to a physical storage asset, and at the same time provides access to this storage into the namespace. This approach allows the deployment of stateful applications like databases.

If an application requires a given amount of storage, the following steps must be performed:

- first, a PersistentVolume resource must be created in the cluster, referencing a physical storage asset having the same amount of space.
- a PersistentVolumeClaim resource must be defined in a manifest file to be deployed, referencing the volume created above. This resource will be created during the deployment of the application
- a Pod which needs to access the storage must specify the claim resource created above. In this way, the Pod can access the required storage.

Developers must provide any storage requirements for their applications.. Volumes resources are created in the cluster by SysOps engineers, while DevOps engineers are in charge of declaring the claim and proper references for Pod resources in the manifest files.

The following picture illustrates how the above resources are organized in the cluster

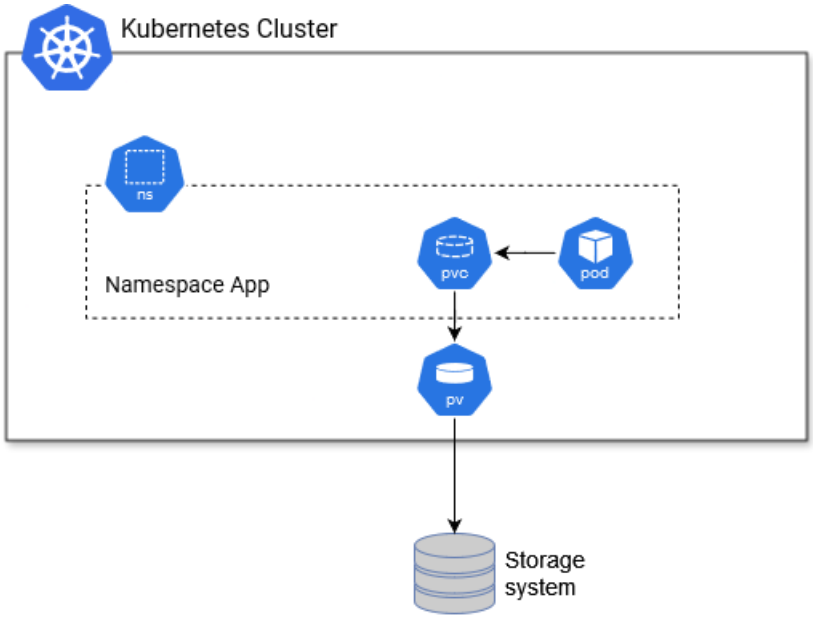

Figure 48: Kubernetes Persistent Volumes

6 Blueprint guidelines for FAME deployments of project pilots and application technologies

6.1 Guidelines overview

This chapter describes the process and guidelines for the development and the deployment of an application using the FAME CI/CD infrastructure with the DevOps principles. It is provided with the main purpose to help the developers of FAME components, such as the dashboard, the backend, analytics, and technology components, the know-how to use the infrastructure to integrate the applications in a common namespace.

The following picture depicts the typical CI/CD workflow with the main actors (developer, devops engineer, end-user) involved.

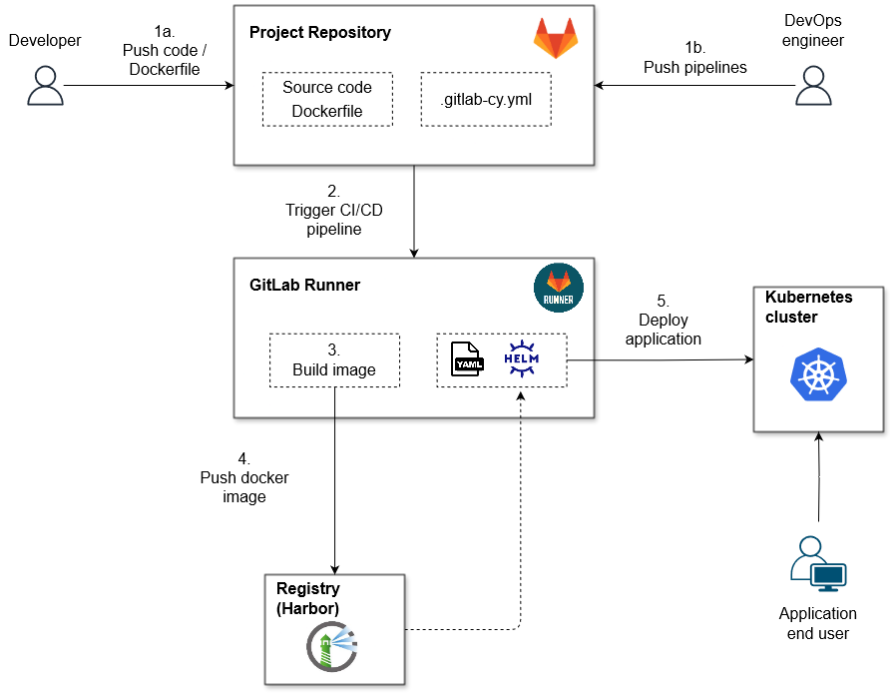

Figure 49: FAME CI/CD Workflow

6.1.1 Development cycle

With respect to the diagram above, the development cycle consists of the following steps:

(1a.) A Project Repository is created on GitLab for each application or service to be developed. The developer uploads the application code on the repository and prepare the Dockerfile, which will be used to produce the docker image in a later step. The code, including Dockerfile and all material necessary to build the application, is committed to the GitLab repository.

(1b.) A DevOps engineer writes the CI/CD pipeline code in a specific file located in the repository, (*.gitlab-cy.yml)*. The pipeline code is organized in several stages, executed in the order specified, and contains all the instructions necessary to build and later deploy the application in the Kubernetes cluster.

(2.) As soon as the code is committed to the GitLab repository, the pipeline is automatically triggered, and a process (called GitLab runner) is instantiated and start the execution of the steps indicated in the pipeline.

(3.) Typically the first stage of the CI/CD pipeline is the *build* stage (the CI part of the pipeline). In this stage, the Dockerfile is used to produce the docker image of the application or service. Each docker image produced in this stage should be versioned, typically using a tag or a hash (sha256).

(4.) If the build stage is successful, the docker image produced in pushed into the docker image registry (manage by the application Harbor), to be properly stored and versioned.

(5.) The execution of the pipeline proceeds with the CD part of the pipeline, the *deploy* stage. The application docker image is pulled from the Harbor registry and deployed to the Kubernetes cluster, either by using the *kubectl* command to apply a set of manifest files describing resources (e.g. Deployment, Service, ConfigMap, Ingress, etc.), or by installing an application from a Helm chart.

The manifest files or the Helm chart are stored in a specific folder in the same GitLab project repository which contains the application code, and versioned along with the application code. Another approach could be to use a different repository than the application code; both these approaches have their pros and cons.

The development cycle can therefore be summarized as follows:

- *Development*: developers write code locally using their preferred IDE or text editor. Developers are responsible for the local application code tests.
- *Version Control*: Application code is pushed to GitLab for version control and collaboration, both between developers on the same team and between different teams.
- *Continuous Integration* (CI): each time a change in the code is committed in the Gitlab repository, a CI pipeline is automatically triggered.
- *Artifact generation* (aka *build*): successful CI pipeline execution produces deployable Docker images
- *Artifact repository storage*: Artifacts are stored in a repository in Harbor, and made available for deployment

6.2 Building a FAME component

A FAME component, like an application or a service, is built using a CI/CD pipeline in GitLab. The pipeline scripts use so-called CI/CD variables, stored and configured in GitLab.

The use of variables has two main purposes:

- avoid hardcoding and allows reuse of configurations in different scripts pertaining to the same project, even if the code is located in multiple repositories
- store secret information that should not be shown in plain text

Currently, several CI/CD variables are defined and used both in build and in deploy stages, as follows.

Build stage

- **DOCKER_REGISTRY**, points to the Harbor URL and used to publish container images
- **DOCKER_USER**, the username for Docker login to Harbor.
- **DOCKER_PASSWORD**, the password for Docker login to Harbor

Deploy stage

- **KUBECONFIG**, containing the configuration file for Kubernetes cluster access and operation

The DOCKER_USER and DOCKER_PASSWORD credentials are specific for CI/CD pipelines and different from user credentials, which are instead used for logging in to GitLab. The use of specific credentials for the pipeline allows a better control of permissions, which can be established at project level and not at user level.

Moreover, because a user can belong to different groups and associated to multiple projects, establishing permissions at user level does not allow to specify different user permissions for different projects. Therefore, the permission model for the pipeline is a group/project based access control instead of a user based access control

6.3 Building an image using pre-compiled libraries/binaries

All necessary libraries should be imported in the application code, typically from external repositories, so that will be part of the built application image.

It’s also possible build an application image by referencing multiple images in the Dockerfile using specific features like Docker multi-stage build

In this cases, the CI/CD pipelines can be still used to build and deploy the application as described before, using same variables as necessary.

7 Conclusions

As previously mentioned, FAME aims to be a realized Data Space, specifically a Finance Data Space, one of the types of Data Space defined by the EU among the Common European Data Spaces[4].

Compared to traditional market players, FAME, as a Data Space, aims for a lighter version of implementation constraints for the valuation, sharing, and commercialization of digital assets (Assets). In fact, no specific implementation constraints are imposed on the exchange of data.

FAME's federated model aims to guarantee the rules, principles, and control of participation, safeguarding the values of commercial exchange, rather than imposing the creation of a mere interchange platform through connectors. This is to allow asset producers the freedom to decide how to provide/exchange assets with consumers, as long as they are acceptable within the federation.

The motivation is to enable even SMEs to participate in and benefit from the Data Economy in the Finance sector through FAME.

FAME focuses on the regulation of economic exchanges of assets.

FAME aims to enrich the reference architectures of Data Spaces with value-added components for data assets, such as trading and monetization modules, which are currently missing.

To achieve this objective, FAME aims to define three frameworks: a Business Framework, a Legal/Organizational Framework, and a Technological Framework for the realization of Data Spaces in the Finance sector.

This document describes the integration architecture of the technical framework of the FAME Marketplace for the valuation and trading of assets. This integration architecture allows the various technical modules of the solution to integrate, enabling interaction with users and integration with other systems/marketplaces/Data Spaces. The goal of this integration is to allow the inclusion of assets (digital assets of various types) in the FAME "showcase," enhancing their value through their integration, commercialization, and exchange. In this document, the term "asset" refers to digital assets of various types such as Data Assets, AI models, and static analytics or their runtime made available through services accessible to consumers.

In addition to the integration of the various components, the document describes the infrastructure on which the FAME Data Space will be built during the project phase. This DevOps infrastructure includes applications/tools for managing code or pre-existing work-in-progress (Docker images) and all deployment processes according to best CD/CD practices, as well as an environment for releasing the solution at runtime.

In this version, the document describes the outputs of the work carried out in the first 18 months of the project. Subsequent evolutions, implementations, and integrations will be documented in the next version of the document. The next version will be the document D2.6, scheduled for June 2025.

References

1. European data strategy, https://digital-strategy.ec.europa.eu/en/policies/strategy-data
2. European Data Governance Act, https://digital-strategy.ec.europa.eu/en/policies/datagovernance-act
3. Data Act, https://digital-strategy.ec.europa.eu/en/policies/data-act
4. Common European Data Spaces, https://digital-strategy.ec.europa.eu/en/policies/data-spaces
5. FAME, D2.6 - Technical Specifications and Platform Architecture II (2024)
6. FAME, D2.5 - Requirements Analysis, Specifications and Co-Creation II (2024)
7. FAME, D3.2 - Federated Data Assets Catalogue I (2024)
8. FAME, D4.1 - Blockchain-based Data Provenance Infrastructure I (2024)
9. FAME, D4.2 - Pricing, Trading and Monetization Techniques I (2024)
10. FAME, D3.3 - Mechanisms and Tools for Regulatory Compliance I (2024)
11. FAME, D5.1 - Trusted and Explainable AI Techniques I (2024)
12. FAME, D5.2 - Energy Efficient Analytics Toolbox . I (2024)
13. European AI Act, AI Act | Shaping Europe's digital future (europa.eu)
14. Octopus Visual Sitemap tool, https://octopus.do/
15. FIGMA, https://www.figma.com/
16. Angular, https://github.com/angular/angular/releases?page=5
17. Docker, https://www.docker.com/
18. HTML, https://html.com/html5/
19. SCSS, https://www.npmjs.com/package/sass/v/1.70.0
20. TypeScript, https://www.npmjs.com/package/typescript
21. Angular Material, https://v16.material.angular.io/
22. WSL, https://learn.microsoft.com/en-us/windows/wsl/install
23. Git, https://www.simplilearn.com/tutorials/git-tutorial/git-installation-on-windows
24. Node.js, https://nodejs.org/en/download/package-manager
25. Docker installation, https://www.docker.com/products/docker-desktop/
26. Node Version Manager, https://github.com/coreybutler/nvm-windows/releases
27. NPM, https://phoenixnap.com/kb/install-node-js-npm-on-windows
28. API-First approach, https://www.postman.com/api-first/
29. postman, https://www.postman.com/
30. SwaggerHub, https://swagger.io/tools/swaggerhub/
31. Online Swagger editor, https://editor.swagger.io/