The Food Safety Market: An SME-powered industrial data platform to boost the competitiveness of European food certification

D3.3 – Annual Report from TheFSM Software Components Integration and Testing

DELIVERABLE NUMBER	D3.3
DELIVERABLE TITLE	Annual Report from TheFSM Software Components Integration and Testing
RESPONSIBLE AUTHOR	Danai Vergeti (UBITECH)

Co-funded by the Horizon 2020
Framework Programme of the European Union

GRANT AGREEMENT N.	871703
PROJECT ACRONYM	TheFSM
PROJECT FULL NAME	The Food Safety Market: An SME-powered industrial data platform to boost the competitiveness of European food certification
STARTING DATE (DUR.)	01/01/2020 (36 months)
ENDING DATE	31/12/2022
PROJECT WEBSITE	www.foodsafetymarket.eu
COORDINATOR	Nikos Manouselis
ADDRESS	110 Pentelis Str., Marousi, GR15126, Greece
REPLY TO	nikosm@agroknow.com
PHONE	+30 210 6897 905
EU PROJECT OFFICER	Stefano Bertolo
WORKPACKAGE N. | TITLE	WP3 | Platform
WORKPACKAGE LEADER	UBITECH
DELIVERABLE N. | TITLE	D3.3 | Annual Report from TheFSM Software Components Integration and Testing
RESPONSIBLE AUTHOR	Danai Vergeti (UBITECH)
REPLY TO	vergetid@ubitech.eu
DOCUMENT URL	
DATE OF DELIVERY (CONTRACTUAL)	31 January 2022 (M24)
DATE OF DELIVERY (SUBMITTED)	31 January 2022 (M24)
VERSION | STATUS	2.0 | Final
NATURE	Report (R)
DISSEMINATION LEVEL	Public (P)
AUTHORS (PARTNER)	Danai Vergeti (UBITECH), Dimitris Ntalaperas (UBITECH), Iosif Angelidis (UBITECH)
CONTRIBUTORS	Giannis Stoitsis (AGROKNOW), Branimir Rakic (PROSPEH), Antoniy Kunchev (SAI)
REVIEWER	Simeon Petrov (SAI)

VERSION	MODIFICATION(S)	DATE	AUTHOR(S)
0.1	Table of contents	10 December 2021	Danai Vergeti (UBITECH)
0.3	Contribution to section 2 and 3	14 December 2021	Danai Vergeti (UBITECH)
0.5	Contribution to section 4	7 January 2022	Danai Vergeti (UBITECH), Iosif Angelidis (UBITECH)
0.6	Contribution to sections 2 and 3	12 January 2022	Antonyi Kunchev (SAI)
0.8	Contribution to sections 2 and 3	14 January 2022	Branimir Rakic (PROSPEH)
0.9	Contribution to sections 1 and 5 and version for review	20 January 2022	Danai Vergeti (UBITECH), Iosif Angelidis (UBITECH)
0.95	Deliverable review	22 January 2022	Antonyi Kunchev (SAI)
1.0	Final version	25 January 2021	Danai Vergeti (UBITECH)

PARTNERS		CONTACT
Agroknow IKE (Agroknow, Greece)	Agroknow	Nikos Manouselis (Agroknow) nikosm@agroknow.com
SIRMA AI EAD (SAI, Bulgaria)	ontotext	Svetla Boytcheva (SAI) svetla.boytcheva@ontotext.com
GIOUMPITEK MELETI SCHEDIASMOS YLOPOIISI KAI POLISI ERGON PLIROFORIKIS ETAIREIA PERIORISMENIS EFTHYNIS (UBITECH, Greece)	UBITECH ubiquitous solutions	Danai Vergeti (UBITECH) vergetid@ubitech.eu
AGRIVI DOO ZA PROIZVODNJU, TRGOVINU I USLUGE (Agrivi d.o.o., Croatia)	agrivi	Tanja Matosevic (Agrivi d.o.o.) tanja.matosevic@agrivi.com
PROSPEH, POSLOVNE STORITVE IN DIGITALNE RESITVE DOO (PROSPEH DOO, Slovenia)	origintrail	Ana Bevc (PROSPEH DOO) ana@origin-trail.com
UNIVERSITAT WIEN (UNIVIE, Austria)	universität wien	Elizabeth Steindl (UNIVIE) elisabeth.steindl@univie.ac.at
STICHTING WAGENINGEN RESEARCH (WFSR, Netherlands)	ALTERRA WAGENINGEN UR	Yamine Bouzembrak (WFSR) yamine.bouzembrak@wur.nl
TUV- AUSTRIA ELLAS MONOPROSOPI ETAIREIA PERIORISMENIS EUTHYNIS (TUV AU HELLAS, Greece)	TÜV AUSTRIA HELLAS	Kostas Mavropoulos (TUV AU HELLAS) konstantinos.mavropoulos@tuv.at
TUV AUSTRIA ROMANIA SRL (TUV AU ROMANIA, Romania)	TÜV AUSTRIA ROMANIA	George Gheorghiou (TUV AU Romania) george.gheorghiu@tuv.at
VALORITALIA SOCIETA PER LA CERTIFICAZIONE DELLE QUALITA'E DELLE PRODUZIONI VITIVINICOLE ITALIANE SRL (VALORITALIA, Italy)	VALORITALIA	Francesca Romero (Valoritalia) francesca.romero@valoritalia.it
TUV AUSTRIA CERT GMBH (TUV AU CERT, Austria)	TÜV AUSTRIA	Stefan Hackel (TUV AU CERT) stefan.hackel@tuv.at

ACRONYMS LIST

ABAC	Attribute Based Access Control
API	Application Programming Interface
CB	Certification Body
CI/CD	Continuous Integration and Continuous Delivery
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DoA	Description of Action
FSM	Food Safety Market
GUI	Graphical User Interface
HTTP	Hyper-Text Transfer Protocol
KPI	Key-Performance Indicator
OS	Operating System
VM	Virtual Machine
REST	Representational state transfer
STEP	Systematic Test and Evaluation Process
s/w	Software
UI	User Interface
UX	User Experience
XML	Extensible Markup Language

EXECUTIVE SUMMARY

In the current document we present the integration strategy of TheFSM Platform which includes the development, testing and deployment strategy for TheFSM technical solution. We focus on the methodology on both component and platform level. Unit testing is utilized to automate testing as much as possible, while we carefully follow well-established industry standards and methodologies in our approach. We provide examples of unit tests which test the internal functionality of components and their compliance with the specifications. Additionally, integration tests are provisioned in order to assure the interoperability between components. As more use cases are covered by the platform, these tests will evolve, adapt and carefully cover the newly added functionality. Moreover, we describe issue tracking and the dockerization process we follow during development.

The current document provides a) the approach we follow in development regarding both the overall project and within component level, b) the technical verification of our pipeline. We define how verification is guaranteed via unit and integration testing, thoroughly describing the unit testing process and c) the technical evaluation, where we define metrics and KPIs relevant to the project and we provide analytical tables which include our performance against the relevant metrics.

TABLE OF CONTENTS

EXECUTIVE SUMMARY 6

1 INTRODUCTION 10

1.1 SCOPE 10

1.2 AUDIENCE 10

1.3 STRUCTURE 10

1.4 RELATION TO OTHER DELIVERABLES 10

2 THEFSM PLATFORM INTEGRATION 12

2.1 METHODOLOGY 12

2.1.1 Integration strategy 12

2.1.2 Development process 12

2.1.3 Integration Plan for M24 16

2.1.4 Release Plan for TheFSM Platform 16

2.2 INTEGRATION TOOLS 25

2.2.1 Code Level Integration 27

2.2.2 Teams Communication – Slack 28

2.2.3 Automated Building Tools 29

2.2.4 Testing frameworks 29

2.2.5 Issue tracking 30

2.3 PLATFORM DEPLOYMENT 30

2.4 PLATFORM INTERFACES DESCRIPTION 31

3 TECHNICAL VERIFICATION 49

3.1 INTEGRATION TESTING 49

3.1.1 Unit testing 51
3.1.2 Integration testing 54
3.1.3 FSM integration points and complex flows testing 56
User registration 57
A2C engine 58
Read dynamic/streaming data 58
Write dynamic/streaming data 59
Read static data 59
Write static data 60
User Registration 61
User Authentication 61
JWT verification with DID authentication 62
4 TECHNICAL EVALUATION 66
4.1.1 Product Quality Model Categories 66
4.1.2 Technical KPIs of the FSM Platform 68
5 CONCLUSIONS 75
REFERENCES 76
ANNEX I 77

LIST OF FIGURES

Figure 2-1: CI/CD pipeline. 13
Figure 2-2: The FSM development steps following the CI/CD approach. 13
Figure 2-3: The FSM integration process lifecycle. 14
Figure 2-4: The FSM integration process. 16
Figure 2-5: GitLab repositories hierarchy 27
Figure 2-6: GitLab Resulting path 28
Figure 2-7: FSM Slack Workspace 28
Figure 3-1: Activity Timing at each level of test 50
Figure 3-2: STEP Phases 50
Figure 3-3: User registration sequence diagram. 58
Figure 3-4: A2C engine authentication and authorization 58
Figure 3-5: Read dynamic data sequence diagram. 59
Figure 3-6: Write dynamic data sequence diagram. 59
Figure 3-7: Read static data sequence diagram. 60
Figure 3-8: Write static data sequence diagram. 60
Figure 3-9: User registration sequence diagram. 61
Figure 3-10: User authentication sequence diagram. 62
Figure 3-11: User authentication #2 sequence diagram. 63
Figure 3-12: User authentication #3 sequence diagram (further simplified) 64
Figure 3-13: Trusted data brokerage mechanism (FairSwap protocol). 65

LIST OF TABLES

Table 1: TheFSM platform release plan. 17
Table 2: Features per component per milestone. 17
Table 3: ABAC API 31
Table 4: Data Services API. 40
Table 5: Data Handler API. 41
Table 6: Unit Test documentation example 52
Table 7: Identified and Planned Integration Tests 56
Table 8: Product Quality Model Categories 66
Table 9: FSM Technical Evaluation KPIs 68

1 INTRODUCTION

1.1 Scope

The scope of the current deliverable is to document the actions taken under "Task 3.6 Technical Verification & Integration Testing" and provides an annual report describing the process and outcomes of the integration and testing of all TheFSM software components. The main objectives of D3.3 are: a) to present the integration strategy followed in TheFSM project including the development and deployment process, the integration plan and the release plan of TheFSM Platform, b) to provide the description of the APIs developed in each current version of the platform and the core integration points, and c) the testing process and results as well as, the technical evaluation results of the platform.

1.2 Audience

D3.3 targets the consortium members of the TheFSM project and especially the technical partners which participate in WP2, WP3, WP4 with the scope to provide the integration strategy, and release plan of the TheFSM Platform, as well as, the description of the APIs developed for the second version of the platform and the core integration points, and the testing results as well as, the technical evaluation results of the platform. Additionally, an audience outside the consortium, with technical background (s/w engineers, developers, architects etc.) can also follow and understand the methodology, the APIs description and integration points and testing results as it is presented in the current document. For the sake of completeness and, due to the fact that this deliverable is in its second iteration, the content provided here will be in "append mode", meaning the content of the original deliverable will remain as is with updates only where needed, while the new content will be added in proper sections.

1.3 Structure

This deliverable is structured as follows.

- Section 2 documents the approach we follow in development regarding both the overall project and within component level. We discuss our proposed testing and deployment strategy to ensure a robust and stable development environment and provide the integration and automation tools we utilize in our pipeline.
- Section 3 is dedicated to the technical verification of our pipeline. We define how verification is guaranteed via unit testing, thoroughly describing the unit testing process.
- Section 4 discusses the technical evaluation. We define metrics and KPIs relevant to the project and we provide analytical tables which include our performance against the relevant metrics.
- Section 5 concludes this report.

1.4 Relation to other deliverables

The current document is strongly related to D3.1 where the functionalities and the components of the FSM platform are described at a conceptual level. The outcomes of T3.2-T3.5 (until M24),

as well as, "D2.1 - Data Models & Representations", "D2.2 – Data Services", "D2.3 - Report on Data Population" (delivered on M24) and WP4 results delivered in M24) provided input for the integration strategy, technical integration and technical tasks.

2 THEFSM PLATFORM INTEGRATION

2.1 Methodology

2.1.1 Integration strategy

The integration of the FSM platform is a living continuous activity, following an iterative agile process. Thus, each iteration produces an increment of the platform with additional features and refinements in the existing features. In order to safeguard the quality of the software solution that will be produced, it is crucial to define a well-defined and concrete integration strategy that will be followed during the entire lifetime of the project. The integration strategy followed in the FSM platform is developed under two main criteria: a) the **smooth integration and technical verification** process, as well as, b) the **on-time delivery** of the different platform versions. Regarding the first, the technical partners of the consortium defined the FSM integration process which applies the CI/CD paradigm using a set of well-established s/w tools and current architectural approaches and DevOps practices like containerization etc. Regarding the second, **prioritized implementation** of the platform features and functionalities has been applied taking under consideration the ranking of the relevant functional and technical requirements, as well as, the internal time plan of the relevant tasks of the project. Based on this, the technical partners update and follow a concrete integration plan for the second version of the FSM platform, which is further updated and extended throughout the project duration.

2.1.2 Development process

The development process of the FSM platform is following an iterative approach from which three major platform releases (v1.00 on M12, v2.00 on M24 and v3.00 on M36) will be delivered in total. Each major release provides the incremental release of the platform, in which all the issues that were identified in the previous release will have been addressed and new features and offerings will be delivered. The **development process for each incremental release** of the platform will be supported by the integration tools described in section 2.2, and following a CI/CD strategy[1], as illustrated in (Figure 2-2):

[1] https://www.redhat.com/en/topics/devops/what-is-ci-cd

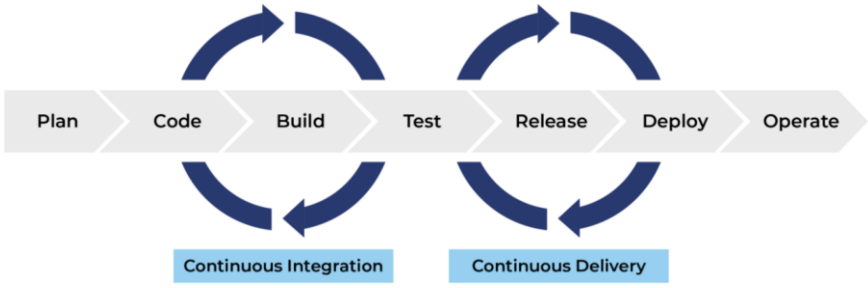

Figure 2-1: CI/CD pipeline.

The aforementioned paradigm has been adopted in TheFSM project as shown in Figure 2-2.

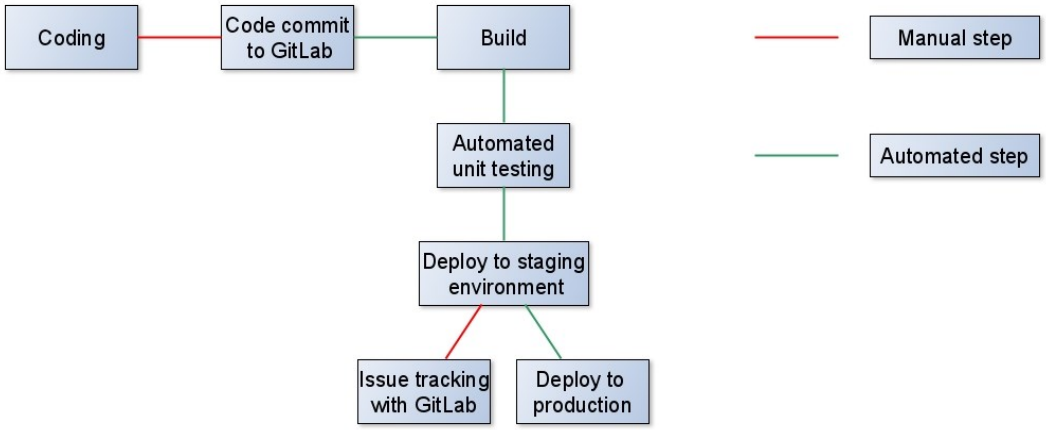

Figure 2-2: The FSM development steps following the CI/CD approach.

Figure 2-2 shows the general development steps following the CI/CD approach:

- When the code implementing a new feature, it is pushed to its respective GitLab repository.
- The code is then automatically built and it is run against unit tests to ensure CI/CD integrity in a safe, non-production environment.
- If at least one test fails, the code is further refined until all tests pass successfully. Otherwise, the built code is deployed to a staging environment for further real-time testing to detect new bugs and observe the overall behavior of the platform after the changes.
- If bugs are detected, GitLab's issue tracking is used to report the bugs and correlate them to a specific component/feature. By using well-defined naming principles in components versioning (major, minor, feature), issue tracking helps pinpoint the exact correction actions required.
- Finally, once a stable milestone is reached for the platform and staging has sufficient performance, it is deployed into production.

Each platform release constitutes an integration cycle in which the steps that will be executed are as follows:

- Each **component** that will be integrated in the platform **is identified**, on the basis of the components described in the FSM architecture.
- The **dependencies** between the various components are identified.
- The **external interfaces** that will be implemented by each component are identified. The identified external interfaces constitute the integration points of the platform and are translated into functionalities that the components will offer to the rest of the components of the platform in order to resolve the identified dependencies.
- Based on the list of identified components, the external interfaces, dependencies and third-party communication with the platform, the **overall integration plan** is defined. In this integration plan, the intermediate releases of the components are scheduled and are aligned with the upcoming releases of the FSM platform.

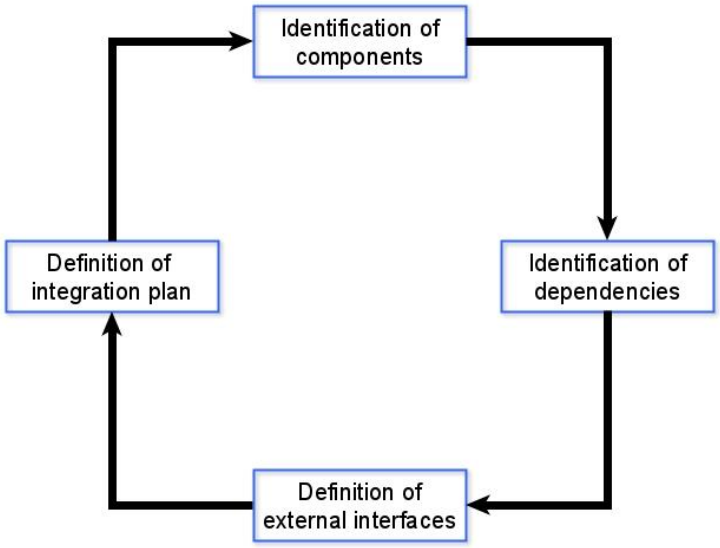

Figure 2-3: The FSM integration process lifecycle.

Following this approach, the successful integration on each integration cycle, presented in Figure 2-4, will be achieved by the following steps:

- **Component implementation**: The development team that is responsible for the implementation of the component develops the component on their local server(s) taking as input the design and specifications of the component. In the development process, the technologies and tools that are defined in the specifications of the component are exploited and the identified interfaces are also implemented.

Intermediate releases of the components are provided according to the integration plan to facilitate the development process. In order for a developed component to become available for integration, the development team has to assure the quality and functionality of the produced component by performing successful local testing on their local server(s) via unit tests.

- **Integration of the component**: During the components' integration phase, all the interactions of the components are tested via the integration tests that are executed automatically, in the highest possible degree, but also through manual testing that is performed by the involved development teams. The scope of the integration testing is to identify any possible defects in the functionalities of the components that are related to the interactions of the components. For any defect identified, a new issue is opened in Github and the responsible development team investigates the issue in order to identify and resolve the identified defect as quick as possible. If the integrations tests are successful, the component is released, labeled as stable and is available for the system integration phase.
- **System Integration**: The system integration phase is the last phase where all components that have passed successfully the integration tests and were labeled as stable are deployed on the common staging environment. In this phase, the platform is tested as a whole by all the development teams that execute thorough tests for all the available functionalities of the platform. Once the whole platform has been successfully tested, the new release is ready and is made available to the production server in order to be evaluated by the FSM demonstrators and stakeholders.

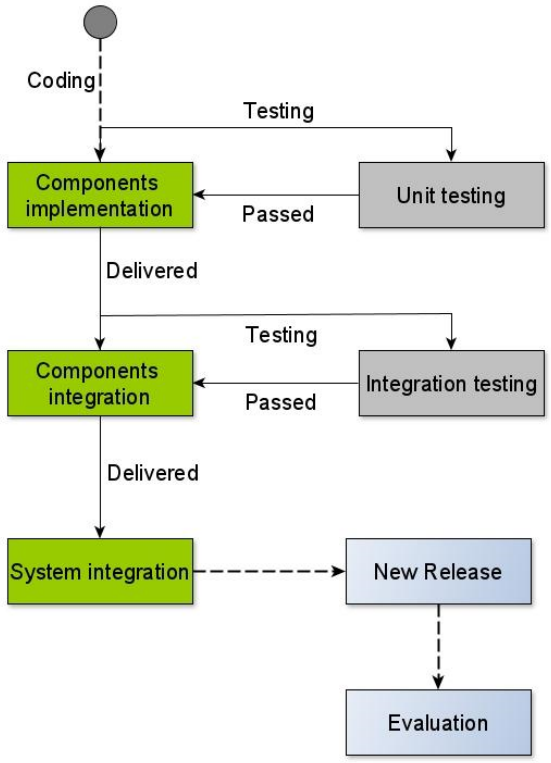

Figure 2-4: The FSM integration process.

2.1.3 Integration Plan for M24

The second version of TheFSM Platform (v2.0) delivers the prototype version of the components mentioned in Table 1: TheFSM platform release plan. The on-time delivery and smooth integration of the components was driven by the integration plan identifying the prioritization and relevant delivery dates of each component and sub-component. The integration plan for M24 can be accessed in Annex I.

2.1.4 Release Plan for TheFSM Platform

The release plan of TheFSM platform provides the delivery status of each component for each platform release. For each component we define the following delivery status: a) Prototype: the prototype of the artifact, including the core functionalities, addressing the core functionalities of the platform, as well, b) Beta version: extended/updated version of the prototype with additional functionalities addressing the core and added-value functionalities of the platform c) Final version: final version of the components, updating and fine tuning the added-value services of the platform.

The release plan of TheFSM platform is provided in the following table:

Table 1: TheFSM platform release plan.

Component	v1.0 (M12)	v2.0 (M24)	Final version (M36)
A2C Engine	Prototype	Beta version	Final version
Semantic Mapper	Prototype	Beta version	Final version
Data Handler	Prototype	Beta version	Final version
Data Staging	Prototype	Beta version	Final version
Secure Storage and Indexing	Prototype	Beta version	Final version
Anonymization Framework	Prototype	Beta version	Final version
AI models	Prototype in development	Beta version	Final version
Data sources and application catalogue	Prototype in development	Beta version	Final version
Query explorer	-	Prototype version in development	Final version
Data License and Agreement Management	-	Prototype version in development	Final version
OTNode DID services	Prototype	Beta version	Final version
OTN DLT Interfaces	Prototype	Beta version	Final version
Data Brokerage Engine	Prototype in development	Beta version	Final version
Data Brokerage Model	Prototype in development	Beta version	Final version
FOODAKAI 2.0 integration	-	Prototype	Final version
AGRIVI 2.0 integration	-	Prototype	Final version
Food Inspector integration	-	Prototype	Final version
Asset trading	-	Prototype	Final version
Advanced Query Engine	-	Prototype in development	Final version
Analytics-as-a-service	-	Prototype in development	Final version
API Gateway	Prototype	Beta version	Final version

Additionally, we provide the overall goals/required features per component for each milestone.

Table 2: Features per component per milestone.

Component	First version (M12)	Second version (M24)	Final version (M36)

A2C Engine	• Allow basic authentication • Allow basic authorization • Prototype model for policies	• Test integrity of prototype model • Refine and add policies closer to the final version	• Ensure engine has good performance under stress conditions
ABE Engine	• Allow basic encryption • Define prototype attributes model	• Test integrity of prototype model • Validate usability of the ABE engine	• Refine engine with final model • Improve performance under stress conditions if needed
Semantic Mapper	• Provide initial semantic mapping functionalities • Accommodate to data curation pipelines • Initial prototype	• Refine and augment the prototype model • Add required functionalities and ensure proper behaviour during data curation	• Finalize model • Improve performance under stress conditions if needed

	model for mapping		
Data Handler	• Provide initial implementation offering data handling operations • Initial integration with data curation pipelines	• Refine data handler by adding functionalities required by the platform	• Finalize data handler component • Ensure good performance under stress conditions
Data Staging	• Provide initial implementation offering data staging operations • Initial integration with data curation pipelines	• Refine data staging by adding functionalities required by the platform	• Finalize data handler component • Ensure good performance under stress conditions

Secure Storage and Indexing	• Provide initial implementation offering storage and indexing throughout the platform • Initial integration with data curation pipelines	• Add security to the storage • Refine secure storage and indexing by adding functionalities required by the platform	• Finalize data handler component • Ensure good performance under stress conditions
Anonymization Framework	• Initial baseline implementation of anonymization	• Refine baseline version and add features as needed • Adapt to project's needs and take extra care when anonymizing legal information in contracts	• Finalize features
AI models	• Initial development of models • Pick features the models will use in training	• Refine features chosen • Develop pipelines involving prediction and classification tasks	• Finalize models

	• Test performance and accuracy		
Data sources and application catalogue	Prototype in development	• Initial version of catalogue with basic functionality and simple filters	• Improve performance for scalability • Improve with advanced filtering and potentially integrating with other components to leverage data information for more refined search functionalities
Query explorer		• Initial version of query explorer with basic functionality and simple filters	• Improve performance for scalability • Improve with advanced filtering and potentially integrating with other components to leverage data

			information for more refined search functionalities
Data License and Agreement Management	-	• Provide initial engine handling data licensing and agreement management • Define use-cases that test the component's utility	• Finalize component by refining as needed
OTNode DID services	• Provide initial authentication services • Integrate with Security Layer's A2C engine	• Add more functionality if required • Test performance	• Finalization
OTN DLT Interfaces	• Provide initial DLT functionality via prototype interfaces	• Integrate interfaces and functionality with the platform • Introduce DLT in contracts management and integrity check	• Finalization

Data Brokerage Engine	Prototype in development	• Provide initial data brokerage use cases and implementation • Test under stress conditions	• Refine implementation • Adapt/augment initial use cases defined for data brokerage as needed to adapt to the platform's needs
Data Brokerage Model	Prototype in development	• Provide initial model to cover data brokerage use cases • Observe model's utility, strengths and shortcomings	• Refine model based on previous milestone's observations
FOODAKAI 2.0 integration	-	• Initial integration with the platform • Test use cases with interactions between FOODAKAI 2.0 and the platform	• Finalize and refine integration

AGRIVI 2.0 integration	-	• Initial integration with the platform • Test use cases with interactions between AGRIVI 2.0 and the platform	• Finalize and refine integration
Food Inspector integration	-	• Initial integration with the platform • Test use cases with interactions between Food Inspector and the platform	• Finalize and refine integration
Asset trading	-	• First draft of asset trading functionalities	• Provide asset trading functionalities final version
Advanced Query Engine	-	• Definition of requirements the query engine needs to provide • Initial version of component with basic functionality implemented	• Develop advanced query engine with complex features as part of further refining the platforms capabilities

Analytics-as-a-service	-	• Definition of requirements the analytics should cover • Initial version of analytics provision with basic functionality implemented	• Finalized version of provided analytics
API Gateway	-	• Initial version of API Gateway component • Enforcer of policies against endpoints of the API Gateway • Incorporation of credentials/authentication for endpoints of the API Gateway • Capabilities of temporarily disabling specific endpoints for maintenance	• Refinement of API Gateway • Semantic enrichment of obtained results as part of the pipeline

2.2 Integration Tools

The integration of the FSM platform is based on carefully selected tools and techniques to ensure the proper planning, design and collaboration of the involved development teams. For this reason,

the consortium decided to utilize a set of well-established tools, open-source libraries and frameworks in order to facilitate the proper integration process. Furthermore, the selection of the tools was based on their suitability and their maturity in terms of features that enable the successful execution of the integration process, as well as on their offerings for the monitoring and management of the integration process of the platform.

a) *Source code versioning and management*

For the source code versioning and management, the well-known solution of Git software is utilized. Git is considered the most widely used, modern version control system offering advanced features with regard to performance, security and flexibility. Git facilitates the development of complex software solutions with the possibility to maintain different repositories for the various components and services under development, as well as different versioning of these repositories, thus facilitating the iterative approach that is adopted for the development of the FSM platform. For the purposes of the FSM project, the consortium decided to utilize GitLab, which is an online Git repository manager. Overall, each component is hosted in a private GitLab[2] repository, where the docker images are being pushed into the registry, while an integration repository tests and deploys the entire platform.

b) *Automated building tools*

As the main development languages of the project are Java, Python and JavaScript, the need for a build automation tool only applies for the Java components. For this case, the well-established tool Maven[3] is exploited. Maven facilitates the building process of the Java projects by providing software management features and automated dependency resolution.

c) *Testing frameworks*

The software quality assurance is safeguarded with the utilization of enhanced testing frameworks that enable the testing of the software modules under development in a structured and exhaustive manner. Each component of the FSM project conducts internal unit tests, while the integration repository tests end-to-end functionality involving two or more components.

d) *Issue tracking*

For the successful integration of the software solution, issue tracking is mandatory. Issue tracking is enabling the planning and design of each process iteration, while also providing the necessary overview of the ongoing activities and progress during each iteration. Issues can be linked to specific components and versions of the components or the platform itself with

[2] As mentioned in the description, the repositories are currently private and accessible only from the consortium members. Access to the repositories can be obtained by contacting the consortium at the moment. The open source repositories will eventually become available from an official FSM project account in GitLab.

[3] https://maven.apache.org/

the use of labels, providing an additional level of management of the whole project. For the FSM project, the issue tracking is performed through GitLab.

e) *Continuous integration*

The continuous integration of the implemented software modules is a crucial development practice in which all the produced modules are integrated together and automatically deployed on the development server in a pre-scheduled short period called integration cycle in order to perform a system-wide testing with manual and automated tests. The frequent integration cycles lead to quick identification of the underlying problem and enable the risk reduction and speed in the development process, while also facilitating bug fixing. In the FSM project, a special repository integrating all components is responsible for continuous integration before each production deployment.

2.2.1 Code Level Integration

Since multiple components need to interact with each other, we have created a special repository alongside the ones for the components with the responsibility of integrating the docker images of the components and deployment. Overall, the structure and hierarchy of the repositories is as follows:

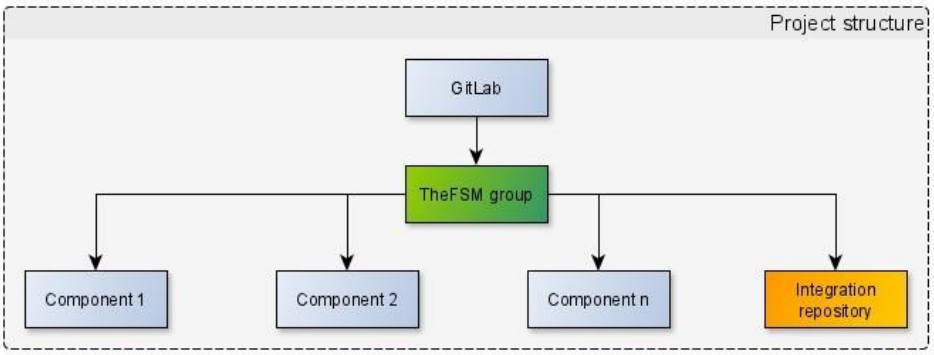

Figure 2-5: GitLab repositories hierarchy

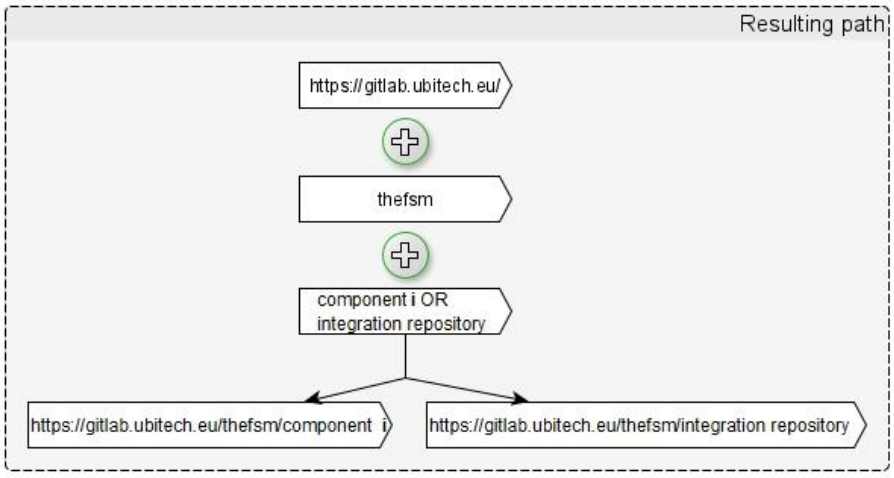

Figure 2-6: GitLab Resulting path

Furthermore, we suggest using a Continuous Integration (CI) scheme based on GitLab, which will be further documented below.

2.2.2 Teams Communication – Slack

For the easier communication during development and integration phase, we created a dedicated slack[4] group where technical partners and use case partners have joined.

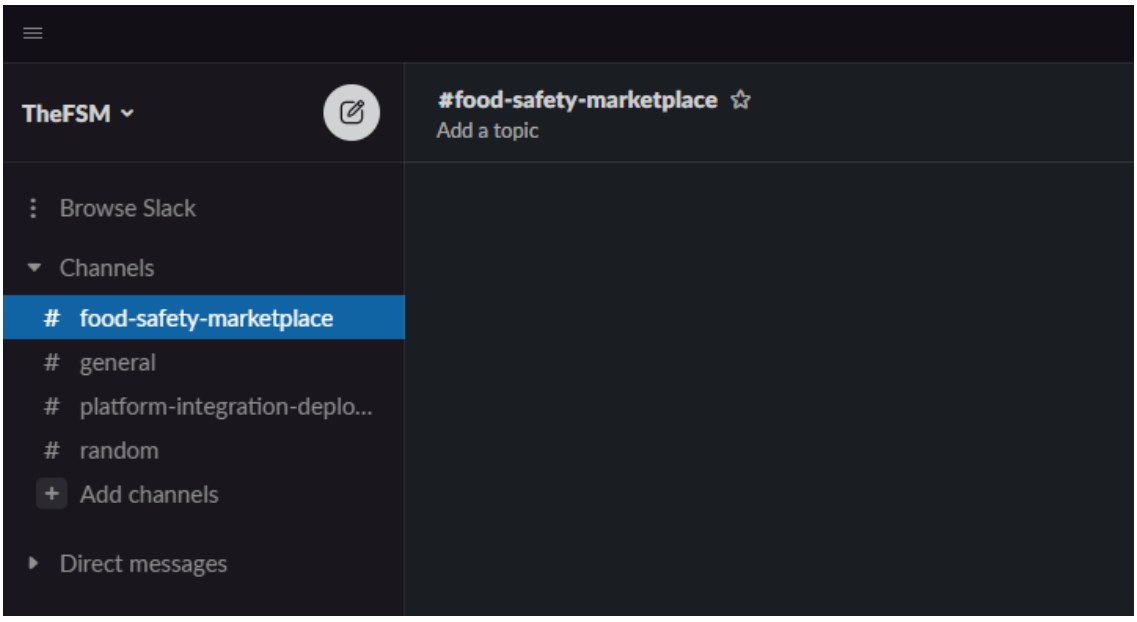

Figure 2-7: FSM Slack Workspace

[4] https://slack.com/intl/en-gr/

In the same time, (bi)weekly calls have been performed for the technical progress of FSM.

2.2.3 Automated Building Tools

As explained above, build automation is mostly required for Java, where Maven is utilized. Maven provided a number of different steps in its deployment lifecycle, called *phases*. The default lifecycle comprises of the following phases[5]:

- **validate** - Validate the project is correct and all necessary information is available.
- **compile** - Compile the source code of the project.
- **test** - Test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
- **package** - Take the compiled code and package it in its distributable format, such as a JAR.
- **verify** - Run any checks on results of integration tests to ensure quality criteria are met.
- **install** - Install the package into the local repository, for use as a dependency in other projects locally.
- **deploy** - Done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

These lifecycle phases (plus the other lifecycle phases not shown here) are executed sequentially to complete the default lifecycle. Given the lifecycle phases above, this means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g., jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository.

For special occasions, or if there is a need for specific order of components' startup for a successful deployment, custom scripts automating procedures will be written.

2.2.4 Testing frameworks

Depending on each component's needs and the programming language(s) used, different testing frameworks are utilized. For example, for Java projects, we use JUnit as our framework of choice. JUnit has been important in the development of test-driven development, while a research survey performed in 2013 across 10,000 Java projects hosted on GitHub found that JUnit (in a tie with slf4j-api), was the most commonly included external library. Each library was used by 30.7% of projects.

[5] For a complete list of the lifecycle phases, refer here: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

2.2.5 Issue tracking

For issue tracking, we use GitLab's solution of GitLab issues. Issues are supported per repository and/or per group level, greatly augmenting their usability. Additionally, they allow tagging per issue to help with archiving, while they allow follow-up discussion on the issue at hand, milestones setup etc. GitLab's issues are thoroughly described in GitLab's own documentation https://docs.gitlab.com/ee/user/project/issues/ .

2.3 Platform Deployment

We consider that each component that is developed or updated shall be created as a docker based container image. In comparison to a virtual machine that needs to include infrastructure configuration and the whole OS, a container image is a lightweight, stand-alone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and configuration files. A container is a runtime instance of an image—what the image becomes in memory when actually executed. In comparison to a Virtual Machine (VM) that is completely isolated, a container is partially isolated from the host environment, as it uses the kernel calls and commands of the host OS, but accessing host files and ports is possible only if configured to do so.

For dockerizing an application, a Dockerfile[6] is needed. The Dockerfile contains instructions on how to construct the instance of an image (called container). The Dockerfile contains directives like:

- defining a base image to be used (e.g. ubuntu 18.04). The developer can login and use the FSM registry order to re-use the images already created as base images, by using the command (FROM *registry.gitlab.com/thefsm/framework/<image_name>:<tag>*)
- adding local files to the file system of the container
- commands to be executed upon initialization of the container (e.g., packages to be installed)
- Ports to be exposed for microservices and components
- Commands which launch the applications

A docker registry has been set for the purposes of FSM development using GitLab. Developers that want to push an image to the repository should first tag it with the repository and then push it using the following commands

```
$ docker login registry.gitlab.com
$ docker build -t registry.gitlab.com/thefsm/framework/<image_name>:<tag> .
```

[6] https://www.docker.com/

```
$ docker push registry.gitlab.com/thefsm/framework/<image_name>:<tag>
```

where:

- *image_name*: The name of the image. Typically, this is the same as the local image name.
- *tag:* Optional as parameter but needed to properly support the continuous integration workflow. It is used for creating versions of the same image. For the first iteration version 0.1 will be used as a tag for all images. If the developer does not specify the tag, the docker will automatically set the tag latest.

Similarly, an image can be pulled by executing:

```
$ docker pull registry.gitlab.com/thefsm/framework/<image_name>:<tag>
```

Each component has its own repository and registry where the docker images are pushed, while the integration repository also pulls from those registries in order to always test and deploy the latest stable version of all images.

2.4 Platform Interfaces Description

In this section, we present with more details the information gathered about the interfaces required for the implementation of the integrated platform by defining the communication between the components.

The following subsections describe these interfaces by detailing the following information:

- **Description:** describes the purpose of the interface.
- **Component providing the interface:** describes the component that is offering the described interface.
- **Consumer components:** describes the components that are using the described interface.
- **Type of interface:** REST, XML-RPC, GUI, Java API etc.
- **Input data:** describes data that is required by the described interface (e.g., Methods or Endpoints, values and parameters of the interface)
- **Output data:** describes the data that is returned by the described interface (e.g., the returned data of methods or REST call)
- **Constraints:** Any other constraints (e.g., specific prerequisites, data-types, encoding, transfer rates) which apply to the interface.
- **Responsibilities:** Partner that is responsible for the implementation and usage of the interface

Table 3: ABAC API

Name: ABAC-REST			
Description	Interface of the ABAC policies editor that is used by the Data Management Pipeline		
Component providing the interface	ABAC Policies Editor		
Consumer components or External Entities	The Data Management Pipeline, Authentication		
Type of Interface	REST		
Input data / Output Data	**Methods** or endpoints of the interface	**Parameters** of the method	**Return Values** of the method
	Dataset Controller		
	DELETE /api/v1/dataset/{id}	id	String status
	GET /api/v1/dataset/{id}	id	DatasetDTO
	GET /api/v1/dataset/{id} /sample	id	DatasetDTO
	POST /api/v1/dataset	DatasetDTO	String status

	Enforcer Controller		
	POST /api/v1/enforcer /testRequest	RequestDTO	String status
	Login Controller		
	POST /api/v1/auth/login	CredentialsDTO	String status, JWT
	Marketplace Controller		
	GET /api/v1/organizations	JWT (authorization header)	List<OrganizationDTO>
	GET /api/v1 /organizationsByDID/{id}	id	List<OrganizationDTO>
	GET /api/v1/taxonomy	-	JSON Array
	GET /api/v1/user/balance	JWT (authorization header)	JSON with balance
	GET /api/v1/user/datasets	JWT (authorization header)	List<DatasetDTO>
	GET /api/v1/user/{id}/roles	id	List<RoleDTO>

	GET /api/v1/user/{id} /rolesByName	id	List<RoleDTO>
	POST /api/v1/dataset/{id} /download	id	String status
	POST /api/v1/dataset/{id} /purchase	id	String status
	POST /api/v1/gateway /exists	ServiceDTO	String status
	POST /api/v1/gateway /{serviceId}/endpoint/exists	serviceId	String status
	POST /api/v1/organization /{name}/exists	name	String status
	POST /api/v1/role/{name} /exists	name	String status
	POST /api/v1/search /datasets	JSON object with query parameters	List<DatasetDTO>
	POST /api/v1/user /{userDID}/existsDID	userDID	String status

	POST /api/v1/user /{username} /existsUsername	username	String status
	Min IO Controller		
	GET /api/v1/bucketExists	-	String status
	GET /api/v1 /downloadObjectByteArray	MinIO object id	Byte array
	GET /api/v1/listObjects	-	JSON array
	POST /api/v1/emptyBucket	-	String status
	POST /api/v1/makeBucket	-	String status
	Organization Controller		
	DELETE /api/v1 /organization/{id}	Id	String status
	GET /api/v1/organization /{id}	Id	OrganizationDTO
	POST /api/v1/organization	OrganizationDTO	String status

	PUT /api/v1/organization /{id}	Id, OrganizationDTO	String status
	Policy Management Controller		
	DELETE /api/v1/pmc /casbinRuleView/{id}	id	String status
	POST /api/v1/pmc /casbinRuleView	RuleDTO	String status
	POST /api/v1/pmc /casbinify	JSON string	String status
	PUT /api/v1/pmc /casbinRuleView/{id}	Id, RuleDTO	String status
	PUT /api/v1/pmc /casbinRuleView/{id} /activate	Id	String status
	PUT /api/v1/pmc /casbinRuleView/{id} /deactivate	id	String status
	Role Controller		
	DELETE /api/v1/role/{id}	id	String status

	GET /api/v1/role/{id}	id	RoleDTO
	POST /api/v1/role	RoleDTO	String status
	POST /api/v1/roles	Pagination input	List<RoleDTO>
	PUT /api/v1/role/{id}	id	String status
	Secure Data Transfer Controller		
	GET /api/v1/sdtc /getPublicKey	-	String
	Service Controller		
	DELETE /api/v1/gateway /{id}	id	String status
	DELETE /api/v1/gateway /{serviceId}/endpoint /{endpointId}	serviceId, endpointId	String status
	GET /api/v1/gateway	-	List<ServiceDTO>
	GET /api/v1/gateway /endpoints	-	List<EndpointDTO>

	GET /api/v1/gateway /services	-	List<ServiceDTO>
	GET /api/v1/gateway/{id}	Id	ServiceDTO
	GET /api/v1/gateway /{serviceId}/endpoint	servivceId	List<EndpointDTO>
	GET /api/v1/gateway /{serviceId}/endpoint /{endpointId}	serviceId, endpointId	EndpointDTO
	POST /api/v1/gateway	ServiceDTO	String status
	POST /api/v1/gateway/call	RequestDTO	Response obtained by calling API or String status
	POST /api/v1/gateway/{id} /endpoint	id, EndpointDTO	String status
	PUT /api/v1/gateway/{id}	id, ServiceDTO	String status
	PUT /api/v1/gateway /{serviceId}/endpoint /{endpointId}	serviceId, endpointId, EndpointDTO	String status

PUT /api/v1/gateway /{serviceId}/endpoint /{endpointId}/disable	serviceId, endpointId	String status
PUT /api/v1/gateway /{serviceId}/endpoint /{endpointId}/enable	serviceId, endpointId	String status
User Controller		
DELETE /api/v1/user/{id}	id	String status
GET /api/v1/user	JWT (authorization header)	UserDTO
POST /api/v1/user	UserDTO	String status
POST /api/v1/users	Pagination input	List<UserDTO>
PUT /api/v1/user/{id}	id, UserDTO	String status
PUT /api/v1/user/{id}/crud	id, UserDTO	String status
PUT /api/v1/user/{id} /disable	id	String status
PUT /api/v1/user/{id} /enable	id	String status

Constraints	N/A
Responsibilities	UBITECH

Table 4: Data Services API.

Name: Data Services-REST			
Description	Interfaces of the Semantic Data Services and components for general data retrieval and enrichment		
Component providing the interface	Semantic Mapper		
Consumer components or External Entities	All data consumers		
Type of Interface	REST		
Input data / Output Data	**Methods or endpoints** of the interface	**Parameters** of the method	**Return Values** of the method
	POST /semantic-services/graphql/	{User authorization token (jwt)} + the text of query-type GraphQL query	JSON object with the result on success or error message on fail
	GET /semantic-services/soml	{User authorization token (jwt)} + the text of query-type GraphQL query	The current semantic object schema in JSON format
	POST /semantic-services/soml	{User authorization token (jwt)} + the text of a new schema in yaml format	Status code
	GET /semantic-services/__health	User authorization token (jwt)}	JSON array with different component services current conditions

	GET /taxonomies/food	-	JSON array with food taxonomies (reconciled against Langual food taxonomy)
	POST /refine	CSV file (dataset) and JSON file (transformation script)	CSV dataset which is the semantically enriched dataset after applying the operations defined in the transformation script.
	POST /refine/rdf	JSON file (dataset) and JSON file (transformation script)	CSV dataset which is the semantically enriched dataset after applying the operations defined in the transformation script.
Constraints	Only objects on which the user has read permissions are included in results		
Responsibilities	SAI		

Table 5: Data Handler API.

Name: Data Handler-REST	
Description	Interface of the Semantic Data Services and components for data mutations
Component providing the interface	Data Handler
Consumer components or External Entities	Predefined data ingestion pipelines
Type of Interface	REST

Input data / Output Data	**Methods or endpoints** of the interface	**Parameters** of the method	**Return Values** of the method
	POST /semantic-services/graphql/	{User authorization token (jwt)} + the text of mutation-type GraphQL query	Status code and error message (on fail)
Constraints	The types (classes) of ingested data must be already defined and included in semantic objects schema. The incoming data must be incorporated in the query text		
Responsibilities	SAI		

Name: OT-DID-AUTH			
Description	Interface of the OT-DID-Authenticator utilized for provisioning, resolving and verifying Decentralized Identifiers controlled by the users		
Component providing the interface	OT-DID-Authenticator		
Consumer components or External Entities	TheFSM services requiring user authentication, and possibly service authentication		
Type of Interface	REST		
Input data / Output Data	**Methods or endpoints** of the interface	**Parameters** of the method	**Return Values** of the method
	POST /{version}/did/create **Description:** Endpoint for user registration in the service ecosystem. The user may or may not have a key management application such as a wallet - if not, a custodian DID key is generated for them, stored in the secure storage, and returned as result.	{ did_method } *or* { did_url } *or* { username, password, did_method } *or*	{ did_document, private_key (optional) }

	Option 1: User already has DID. Payload example:
	
	`{ "did_method": "ethr" }`
	
	Option 2: User already has DID. Payload example:
	
	`{ "did_url": "did:ethr:0x2f2697b2a7BB4555687EF76f8fb4C3DFB3028E57" }`
	
	Option 3: User registers through an interface and provides user name and password. Payload example:
	
	`{ "username": "test", "password": "36f028580bb02cc8272a9a020f4200e346e276ae664e45ee80745574e2f5ab80", "did_method": "ethr" }`
	
	Option 4: User is already authenticated in another system, which is trusted by the OT-Authenticator. Payload example:
	
	`{ "user_id": "test", "metadata": {...}, "did_method": "ethr",`
		{ user_id, metadata, did_method, seed, entity_did_url, signature }	

	<pre>{
	"seed": "timestamp",
	"entity_did_url":
	"did:ethr:development:0x2f2697b2a7BB4555687EF76f8fb4C3DFB3028E57",
	"signature":
	"0x11fbf0696d5e0aa2ef41a2b4ffb623bcaf070461d61cf7251c74161f82fec3a4370854bc0a34b3ab487c1bc021cd318c734c51ae29374f2beb0e6f2dd49b4bf41c"
	}</pre>		

	POST /{version}/did/authenticate
	Description: Endpoint for authentication in the service ecosystem. The user provides authentication material and OT-Authenticator checks if the user has an existing DID registered with the platform in the secure storage. If this is the case, DID service resolves the DID document and OT-Authenticator generates the JWT token signed with the private key of the entity's DID document (OT-Authenticator).
	Later, trusted entities can validate JWT token according to the JWT Token usage sequence diagram.
	
	Option 1: User signed seed is sent accompanied with DID. Payload example:
	<pre>{
	"seed": "timestamp",
	"did_url":
	"did:ethr:development:0x2f2697b2a7BB4555687EF76f8fb4C3DFB3028E57",</pre>
		{ did_url,
		seed,
		signature }
		or
		{ username,
		password }
		or
		{ user_id,
		seed,
		entity_did_url,
		signature }
			jwt_token

```
  "signature":
"0x11fbf0696d5e0aa2ef41a2b4ffb6
23bcaf070461d61cf7251c74161f82f
ec3a4370854bc0a34b3ab487c1bc0
21cd318c734c51ae29374f2beb0e6f
2dd49b4bf41c"
}
```


Option 2: User sends username/password. Payload example:


```
{
  "username": "test",
  "password":
"36f028580bb02cc8272a9a020f420
0e346e276ae664e45ee80745574e2f
5ab80"
}
```


Option 3: User is already authenticated in another system, which is trusted by the OT-Authenticator service. Payload example:


```
{
  "user_id": "test",
  "seed": "timestamp",
  "entity_did_url":
"did:ethr:development:0x2f2697b2a
7BB4555687EF76f8fb4C3DFB3028E
57",
  "signature":
"0x11fbf0696d5e0aa2ef41a2b4ffb6
23bcaf070461d61cf7251c74161f82f
ec3a4370854bc0a34b3ab487c1bc0
21cd318c734c51ae29374f2beb0e6f
2dd49b4bf41c"
}
```

	PUT /{version}/did/provision **Description:** Endpoint for DID provisioning in the service ecosystem. Entities can request a DID provision by providing a valid JWT token.	{ policy data, jwt_token}	{did_document, private_key }
	GET /{version}/did/resolve/{did_url} **Description:** Endpoint for resolving DID documents in the service ecosystem.	{ did_url }	{ did_document }
	POST /{version}/auth/fetch **Description:** Endpoint for fetching DID document details including private key by providing authentication material.	{ seed, did_url, signature }	did_document, private_key
	PUT /{version}/auth/update **Description:** Endpoint for ERC725 key management by trusted authorities.	{ blockchain_transaction }	{ status }
	DEL /{version}/auth/remove **Description:** Endpoint for ERC725 key management by trusted authorities.	{ blockchain_transaction, seed, did_url, signature }	{ status }
	POST /{version}/auth/get_master_key **Description:** Endpoint for fetching OT-Authenticator public key for JWT token validation by providing authentication material.	{ did_url, seed, signature }	ENC(master_public_key)
	POST /{version}/did/sign **Description:** Endpoint for signing transactions using provided did.	{ did, value, data, method, contract }	signed(tx) }

Constraints	N/A
Responsibilities	Prospeh

Name: OT-MARKETPLACE-REST			
Description	Interface of the data brokerage mechanism that is used by the data monetization service		
Component providing the interface	OT-Marketplace		
Consumer components or External Entities	Data brokerage mechanism		
Type of Interface	REST		
Input data / Output Data	**Methods or endpoints** of the interface	**Parameters** of the method	**Return Values** of the method
	POST /{version}/initiate **Description:** Endpoint for data purchase initiation. A user has to provide dataset identifier, seller and buyer identifiers, and price	{ original_dataset_did, dataset_did, seller_did, buyer_did, price }	{ status }

	POST /{version}/withdraw **Description:** Endpoint for token withdrawal if buyer didn't complain. Call should transfer tokens from the smart contract to the seller wallet.	{ purchase_id }	{ status }
	POST /{version}/revert **Description:** If the admin user decides to revert purchase, tokens should be reverted to the buyer's wallet.	{ purchase_id }	{ status }
	GET /{version}/balance **Description:** Returns the balance of the JWT's user in tokens.	JWT in Authorization header	{ status , data -> balance}
Constraints	N/A		
Responsibilities	Prospeh		

3 TECHNICAL VERIFICATION

3.1 Integration Testing

This section presents the platform testing as part of the overall evaluation strategy in the context of the FSM. Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. A number of software projects rely solely on manual acceptance testing to verify that a piece of software conforms to its functional and non-functional requirements. Even where automated tests exist, they are often poorly maintained and out-of-date and thus require extensive manual testing. Testing is a cross-functional activity that involves the whole team, and should be done continuously from the beginning of the project.

Building qualitative software implies the creation of automated tests at multiple levels (e.g. unit, integration and acceptance) and running them as part of the deployment pipeline which is triggered every time a change is made to the application, its configuration or the environment and software stack that it runs on. Manual testing is also an essential part of building quality in showcases, usability testing, and exploratory testing that need to be done continuously throughout the project. For the testing of the FSM platform both automated and manual testing has been adopted in order to combine testing reliability of automated testing tools and also allow for human observation, which may be more useful if the goal is user-friendliness.

It has to be mentioned that the technical testing and evaluation will be based on STEP (Systematic Test and Evaluation Process), a well-established industry methodology for testing and evaluation activities in information technology and software projects. The testing will be performed to verify the proper functioning and performance of the integrated FSM platform.

STEP assumes that the total testing job is divided into levels during planning. A level represents a particular testing environment (e.g., unit testing usually refers to the level associated with program testing in a programmer's personal development library). Simple projects, such as minor enhancements, may consist of just one or two levels of testing (e.g., unit and acceptance), while for complex projects, more levels might be needed (e.g., unit, function, subsystem, system, acceptance testing, alpha, beta, etc.) [1].

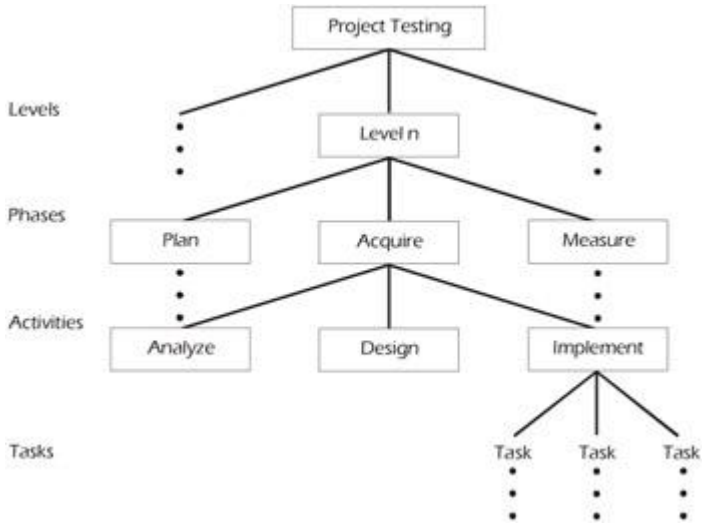

Figure 3-1: Activity Timing at each level of test

STEP provides a model that can be used as a starting point in establishing a detailed test plan, but it is intended to be tailored and revised, or extended to fit each particular test situation.

The three major phases in STEP that are employed at every level include: **planning** the strategy (selecting strategy and specifying levels and approach), **acquiring** the testware (specifying detailed test objectives, designing and implementing test sets), and **measuring** the behavior (executing the tests and evaluating the software and the process). The phases are further broken down into eight major activities, as shown in Figure 3-1.

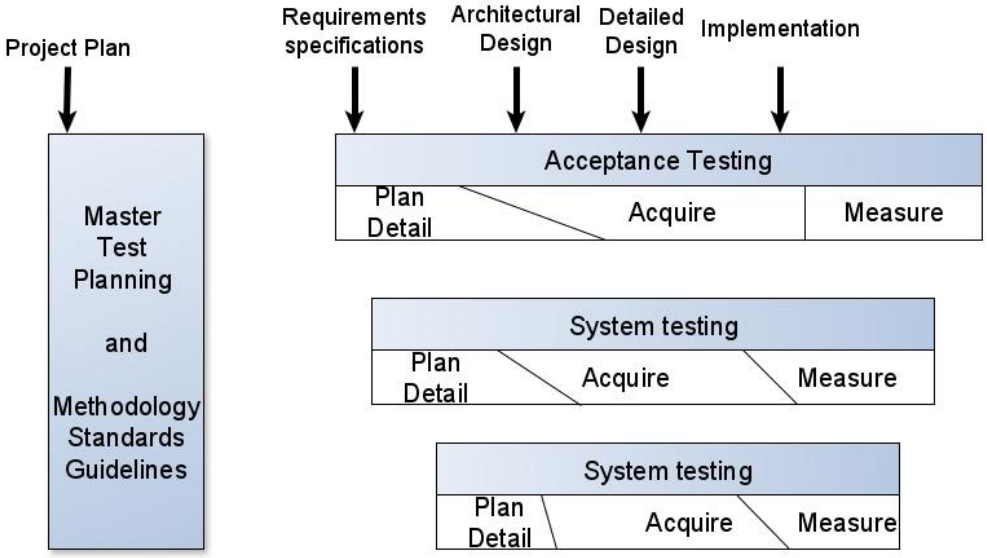

Figure 3-2: STEP Phases

STEP specifies when the testing activities and tasks are to be performed, as well as what the tasks should be and their sequence, as shown in the figure above. The timing emphasis is based on getting most of the test design work completed before the detailed design of the software. The trigger for beginning the test design work is an external, functional, or black box specification of the software component to be tested. For this reason, it is important to plan early for the tests to be performed.

In FSM, we define the following facets of testing:

- Unit testing that can be performed by the separate development teams when new functionalities are developed.
- Integration testing performed by the development teams in order to test the smooth co-operation between the various layers and components. The integration tests and also any unit tests that will be created for the project validation will be continuously executed based on continuous integration (CI) scheme
- Testing of a set of advanced scenarios based on demonstrators' needs.

These testing facets are presented in the following sections.

3.1.1 Unit testing

An important part of both the integration and the validation process is the execution of unit tests. Unit tests are the tool to test the functional modules of software. A suitable unit test is applied to the piece of code without any dependencies on other code parts. Therefore, the developer of the particular layers will test their components by means of unit tests before integrating them into the full application. In the case of the FSM, development is based on the development of standalone components but also on the adaptation and integration of existing components. Therefore, unit testing at the lowest level is not a primordial part of the general testing methodology of FSM, as the main focus is the integration testing.

Unit testing will be used as an additional mechanism of validating the developed code, as a task that each component developer can use in order to verify proper functionality before the integration of the component in FSM. Usually, all unit-tests are executed during the build-process, unless they are defined to be ignored (e.g., marked as @Ignored for Java applications). This practically means that each release of a component is guaranteed to have good stability and at the same time allows developers to better control the level of test coverage on their components.

The exercise of creating unit tests is still evolving as the components become more involved with each other. For the sake of completeness, we provide unit tests for the ABAC part of the Security Layer component. The tests consist of a set of predefined policies, against which test requests are being made. Depending on the policies set, the requests should be either accepted or rejected.

Table 6: Unit Test documentation example

Unit Test Case Documentation Form	
Unit Test Reference Code	#UT1
Component	Security Layer
Tester	Junit
Short Description	
Test requests against a set of policies. Some should be accepted and some should be rejected.	
Input Data (Configuration and Policies)	

```
[request_definition]
r = sub, obj, act

[policy_definition]
p = sub, obj, act, eft

[role_definition]
g = _, _

[policy_effect]
e = some(where (p.eft == allow)) && !some(where (p.eft == deny))

[matchers]
m = (eval(p.sub) ) && (keyMatch(r.obj, p.obj) || keyMatch2(r.obj, p.obj)) && keyMatch(r.act, p.act)

p, r.sub.role.equals("role::ownerOfDataset1"), /foodakai_user_1/dataset1, *, allow
p, r.sub.role.equals("role::defaultUser"), /foodakai_user_1/dataset1, buy, allow
p, r.sub.role.equals("role::defaultUser"), /foodakai_user_1/dataset1, preview, allow

p, r.sub.role.equals("role::foodakai"), /foodakai_user_1/dataset1, review, allow

p, r.sub.accessDate > 1606380768 && r.sub.accessDate < 1606382989 &&
r.sub.role.equals("role::foodInspector"), /foodakai_user_1/dataset1, read, allow
p, r.sub.accessDate > 1606380768 && r.sub.accessDate < 1606382989 &&
r.sub.user.equals("food_inspector_1"), /foodakai_user_1/dataset1, read, allow

p, r.sub.user.equals("agrivi_user_1"), /agrivi_user_1/dataset2, *, allow
p, r.sub.role.equals("role::defaultUser"), /agrivi_user_1/dataset2, buy, allow
p, r.sub.role.equals("role::defaultUser"), /agrivi_user_1/dataset2, preview, allow

p, r.sub.role.equals("role::MachineLearningModel"), *, runExperiment, allow
p, r.sub.role.equals("role::banned"), "*", *, deny

g, r.sub.User == "foodakai_user_1", "role::foodakai"
```

```
g, r.sub.User == "foodakai_user_2", "role::foodakai"
g, r.sub.User == "agrivi_user_1", "role::agrivi"
g, r.sub.User == "food_inspector_1", "role::foodInspector"
g, r.sub.User == "food_inspector_2", "role::foodInspector"
g, r.sub.User == "ml_user", "role::MachineLearningModel"

g, r.sub.User == "foodakai_user_1", "role::defaultUser"
g, r.sub.User == "foodakai_user_2", "role::defaultUser"
g, r.sub.User == "agrivi_user_1", "role::defaultUser"
g, r.sub.User == "food_inspector_1", "role::defaultUser"
g, r.sub.User =="food_inspector_2", "role::defaultUser"
g, r.sub.User == "ml_user", "role::defaultUser"

g, r.sub.User == "normal_user", "role::defaultUser"
g, r.sub.User == "JackTheRipper", "role::defaultUser"

g, r.sub.User == "JackTheRipper", "role::banned"

g, r.sub.User == "foodakai_user_1", "role::ownerOfDataset1"
```

Output Data (test requests)

```
// user accesses their own dataset
request = {"User":"foodakai_user_1", "Role":"role::ownerOfDataset1", "AccessDate": 1606381254L},
object = "/foodakai_user_1/dataset1", action = "read" --> True

// normal user unauthorized access (read)
request = {"User":"normal_user", "Role":"role::defaultUser", "AccessDate": 1606381254L}, object =
"/foodakai_user_1/dataset1", action = "read" --> False


// normal user authorized access (preview)
request = {"User":"normal_user", "Role":"role::defaultUser", "AccessDate": 1606381254L}, object =
"/foodakai_user_1/dataset1", action = "preview" --> True

// normal user authorized access (buy)
request = {"User":"normal_user", "Role":"role::defaultUser", "AccessDate": 1606381254L}, object =
"/foodakai_user_1/dataset1", action = "buy" --> True

// foodakai user authorized to access other foodakai user's data (as a group, review action)
request = {"User":"foodakai_user_2", "Role":"role::foodakai", "AccessDate": 1606381254L}, object =
"/foodakai_user_1/dataset1", action = "review" --> True

// agrivi user unauthorized to access foodakai user's data
request = {"User":"agrivi_user_1", "Role":"role::agrivi", "AccessDate": 1606381254L}, object =
"/foodakai_user_1/dataset1", action = "read" --> False

// agrivi user accessing their own data
```

```
request = {"User":"agrivi_user_1", "Role":"role::agrivi", "AccessDate": 1606381254L}, object =
"/agrivi_user_1/dataset2", action = "read" --> True

// ml user accessing resource to run experiment
request = {"User":"ml_user", "Role":"role::MachineLearningModel", "AccessDate": 1606381254L},
object = "/foodakai_user_1/dataset1", action = "runExperiment" --> True

// food inspector reading foodakai user's data for auditing (authorized)
request = {"User":"food_inspector_1", "Role":"role::foodInspector", 1606381254L}, object =
"/foodakai_user_1/dataset1", action = "read" --> True

// food inspector reading foodakai user's data for auditing (unauthorized)
request = {"User":"food_inspector_1", "Role":"role::foodInspector", 1606388254L}, object =
"/foodakai_user_1/dataset1", action = "read" --> False

// banned user denied access to a resource
request = {"User":"JackTheRipper", "Role":"role::banned", "AccessDate": 1606381254L}, object =
"/foodakai_user_1/dataset1", action = "preview" --> False

// super admin has full access to everything
request = {"User":"SuperAdminUser", "Role":"role::Super Admin", "AccessDate": 1606381254L}, object
= "/foodakai_user_1/dataset1", action = "write" --> True
```

Apart from the tests that guarantee the functional correctness of the components, it is important to make tests at the integration level for a complete testing and validation process. This means that integration tests shall be created and used for all identified interfaces and to some major platform functionalities. This can be done using unit testing on the methods that are implementing the integration, in order to make them part of continuous integration and continuous testing process.

3.1.2 Integration testing

Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. Integration testing in FSM can also be seen as an extension of unit testing. The main idea of integration testing is to start from two components to test the interface between them. In some cases, more than two components can participate in a common test. Eventually, this process will be expanded in order to test all the integrated components of the platform.

The goal of integration testing is to identify problems that occur when components are combined. By using a test plan that suggests the usage of unit tests before combining components, the errors discovered when in integration tests are most probably related to the interface between them. This method reduces the number of possibilities of errors to a far simpler level of analysis.

A system expert can perform integration testing in a variety of ways but the following are three most common strategies:

- The top-down approach of integration testing requires the highest-level modules to be tested and integrated first. This allows high-level logic and data flow to be tested early in the process and it tends to minimize the need for drivers. However, the need for stubs complicates test management and low-level utilities are tested relatively late in the development cycle. Another disadvantage of top-down integration testing is its poor support for early release of limited functionality.
- The bottom-up approach requires the lowest-level units to be tested and integrated first. These units are frequently referred to as utility modules. By using this approach, utility modules are tested early in the development process and the need for stubs is minimized. The downside, however, is that the need for drivers complicates test management and high-level logic and data flow are tested late. Like the top-down approach, the bottom-up approach also provides poor support for early release of limited functionality.

- The third approach, sometimes referred to as the "umbrella" approach[2], requires testing along functional data and control-flow paths. First, the inputs for functions are integrated in the bottom-up pattern discussed above. The outputs for each function are then integrated in the top-down manner. The primary advantage of this approach is the degree of support for early release of limited functionality. It also helps minimize the need for stubs and drivers. The potential weaknesses of this approach are significant, however, in that it can be less systematic than the other two approaches, leading to the need for more regression testing.

The FSM consortium chose the last option, being the best, as it combines the best of both worlds. It allows all participating entities to simultaneously execute multiple testing in several components. Integration testing plays a significant role for the proper functionality of the platform. As indicated by the proposed system architecture, the platform consists of a number of components implemented by different partners with different technologies addressing different functionalities. The valid integration of these components is a fundamental need and integration testing should be a priority throughout the development process.

For testing the network communication of web services that were undergoing development we also employed service virtualization techniques. In software engineering, service virtualization is a method to emulate the behavior of specific components in component-based applications. It is used to provide software development and testing teams access to dependent system components that are needed to exercise an application under test, but are unavailable or difficult-to-access for development and testing purposes. With the behavior of the dependent components "virtualized" testing and development can proceed without accessing the actual live components, until they become available.

Service virtualization involves creation and deployment of a "virtual asset" that simulates the behavior of a real component which is required to exercise the application under test, but is difficult or impossible to access for development and testing purposes.

A virtual asset stands in for a dependent component by listening for requests and returning an appropriate response—with the appropriate performance. For example, in a web service, this involves listening for a RESTful message over HTTP and then returning another message. The virtual asset's functionality and performance tries to reflect the actual functionality of the component, also tries to simulate exceptional conditions (such as extreme loads or error conditions) to determine how the application under test responds under those circumstances.

Virtual assets are typically created either by providing logs representing historical communication among components, analyzing service interface specifications (such as RESTful) or defining the behavior manually with various interface controls and data source values. They are then further configured to represent specific data, functionality, and response times. After the release of a web service prototype, it substitutes the virtualized one, in order for the integrated testing of the real one to be initiated.

3.1.3 FSM integration points and complex flows testing

As it is important for FSM to ensure the proper integration of the components, tests that are based on functions that cover different components will be used. In these tests methods from different components are combined in order to achieve the needed functionality, so the focus is given to the combination of pieces which create a basic integrated functionality. It should be noted that the table contains representative examples of such integration points.

Table 7: Identified and Planned Integration Tests

Test ID	Test	Interface(s) Tested	Components Used	Short Description
IT1	End to end encryption when data transferring basic test	Security Layer end to end encryption	Security Layer, all other components requiring data transfer from/to the end user	Ensure hybrid encryption works correctly and data are encrypted between transfers.
IT2	ABAC policy editing	ABAC dashboard and Security Layer Backend	ABAC dashboard (UI, frontend), Security Layer backend	Testing policy CRUD operations work as intended.
IT3	ABAC enforcement test	ABAC dashboard and Security Layer backend	ABAC dashboard (UI, frontend), Security Layer backend,	Testing granted status on requests by filtering through authentication and authorization

			Authentication component	
IT4	Dataset upload	Marketplace interaction with user in order to upload dataset with simple steps	Security Layer, Data Curation Layer	Testing that users with correct attributes can upload datasets. Test each individual step of the dataset upload process (column filtering, sampling, curation, column semantic mapping etc.) and ensure the dataset arrives at the platform encrypted.
IT5	Dataset monetization and purchase	Marketplace interaction with user in order to purchase dataset	Security Layer, Data Brokerage model, ABAC	Testing that a proper transaction between a buyer and a seller can take place. Make sure policies enforced by the original owner are maintained. Test corner cases (e.g. buyer and seller are the same person, dataset costs 0 tokens, buyer does not have enough tokens to buy, exception occurring during the FairSwap protocol and triggering a revert).

Next, for the sake of completeness, we provide a few sequence diagrams where the authentication component of PROSPEH and the ABAC authorization component of UBITECH interact with each other for specific pipelines. Those have been integration tested.

User registration

This process involves the steps required for user registration in FSM.

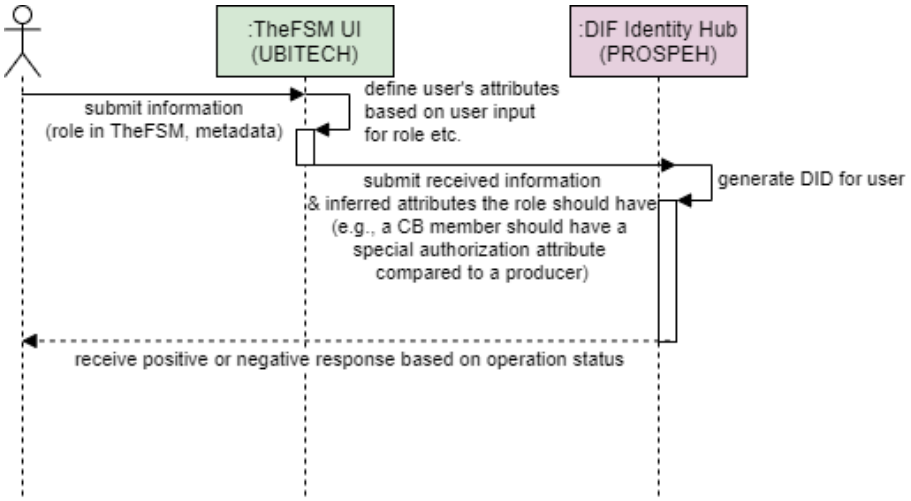

Figure 3-3: User registration sequence diagram.

A2C engine

This process involves the Security Layer and, more specifically, UBITECH's ABAC component and PROSPEH's authentication component. A request is granted iff the user is authenticated and authorized. Each user has well-defined attributed with both user and environmental parameters.

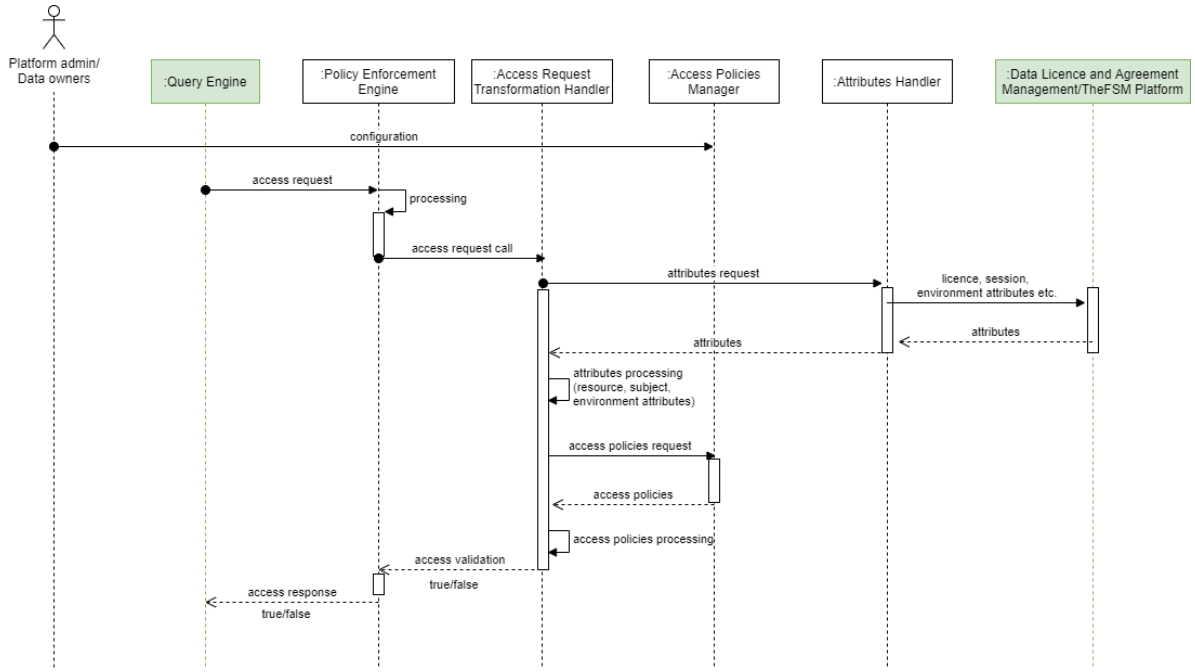

Figure 3-4: A2C engine authentication and authorization

Read dynamic/streaming data

This process involves the steps required to read dynamic data (normally, data curation is another integration point with SAI, but is skipped in the diagram for simplicity).

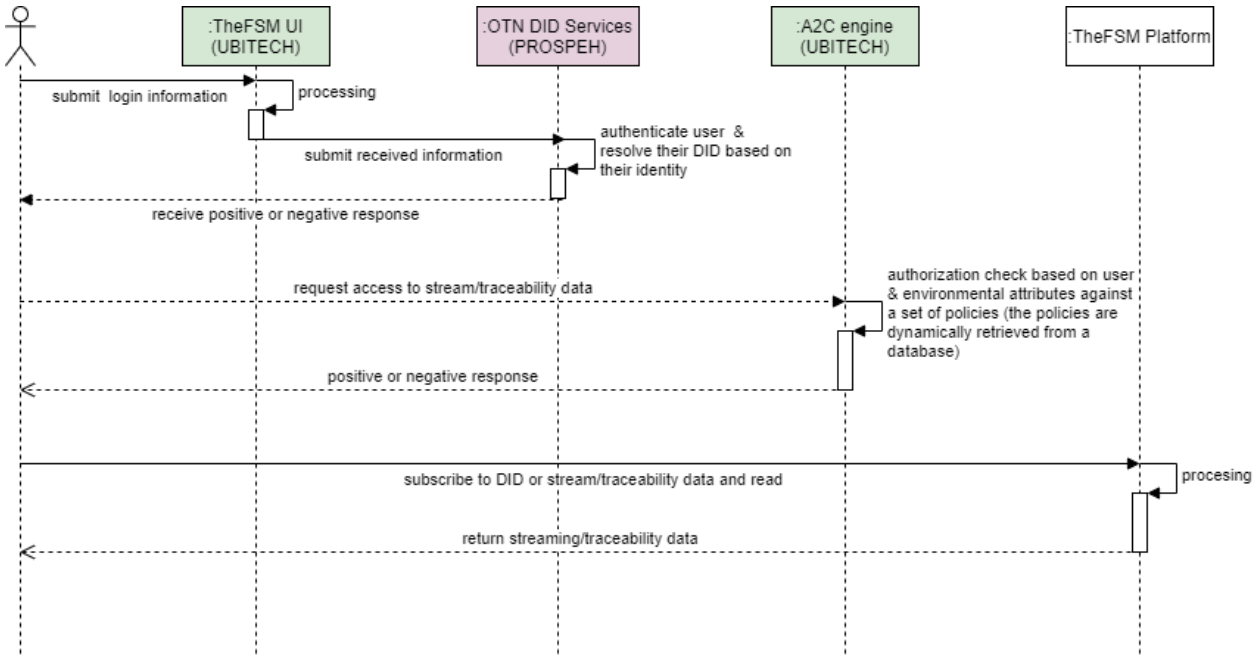

Figure 3-5: Read dynamic data sequence diagram.

Write dynamic/streaming data

This process involves the steps required to write dynamic data (normally, data curation is another integration point with SAI, but is skipped in the diagram for simplicity).

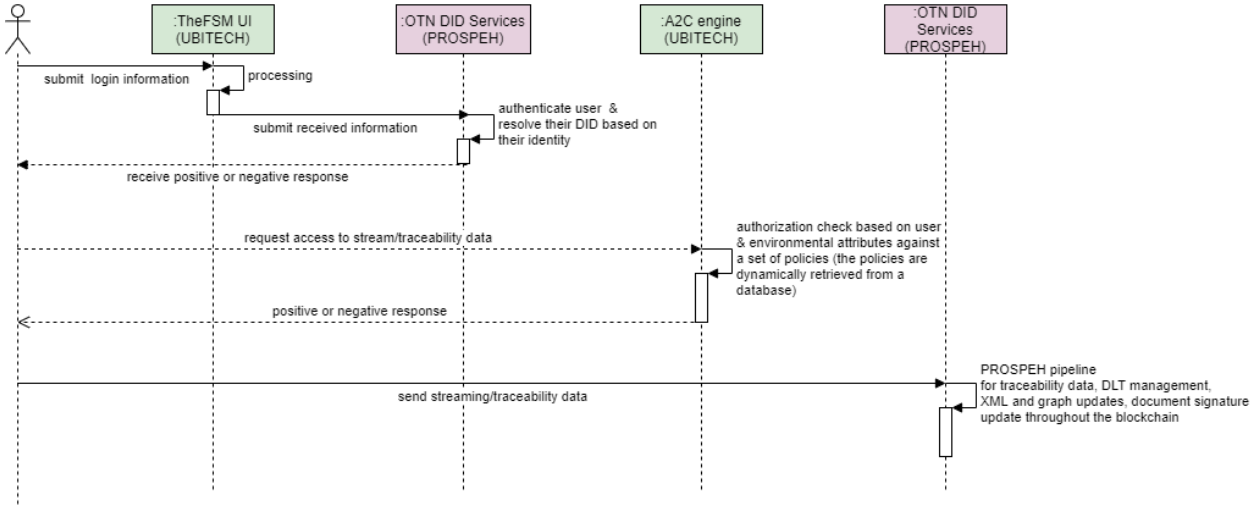

Figure 3-6: Write dynamic data sequence diagram.

Read static data

This process involves the steps required to read static data (normally, data curation is another integration point with SAI, but is skipped in the diagram for simplicity).

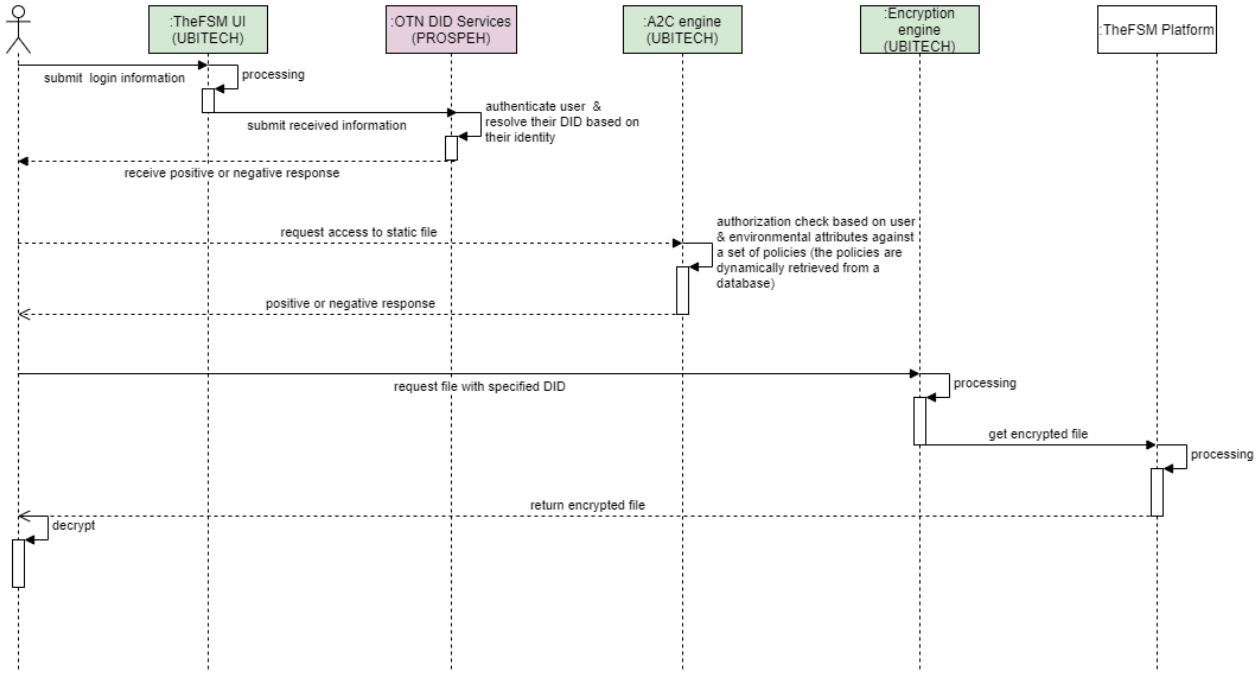

Figure 3-7: Read static data sequence diagram.

Write static data

This process involves the steps required to write static data (normally, data curation is another integration point with SAI, but is skipped in the diagram for simplicity).

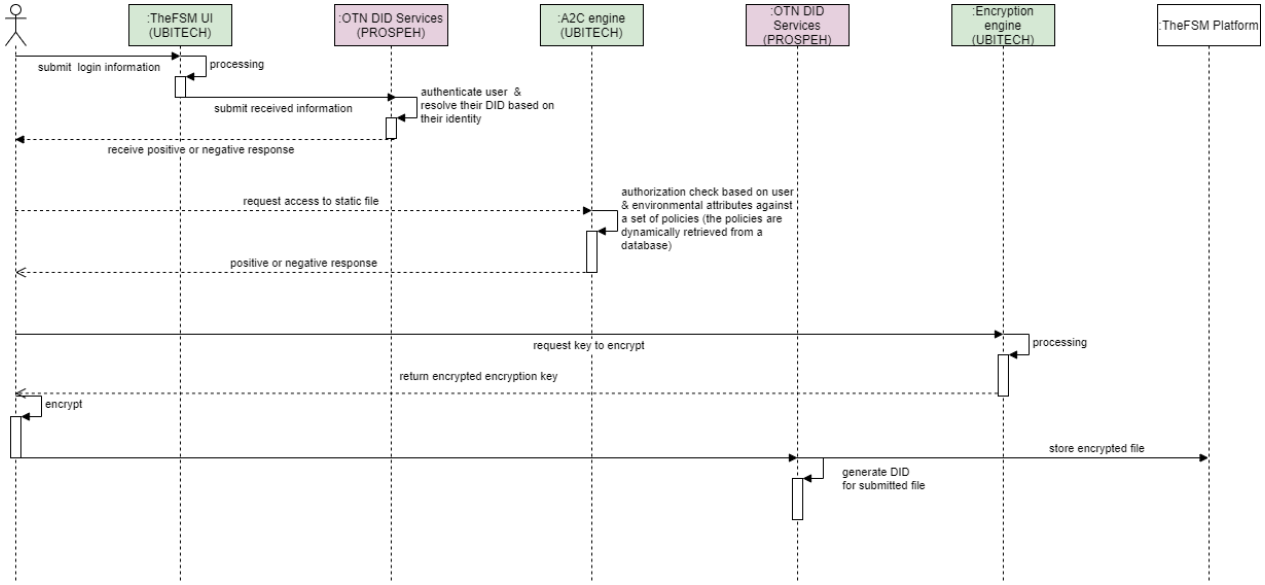

Figure 3-8: Write static data sequence diagram.

User Registration

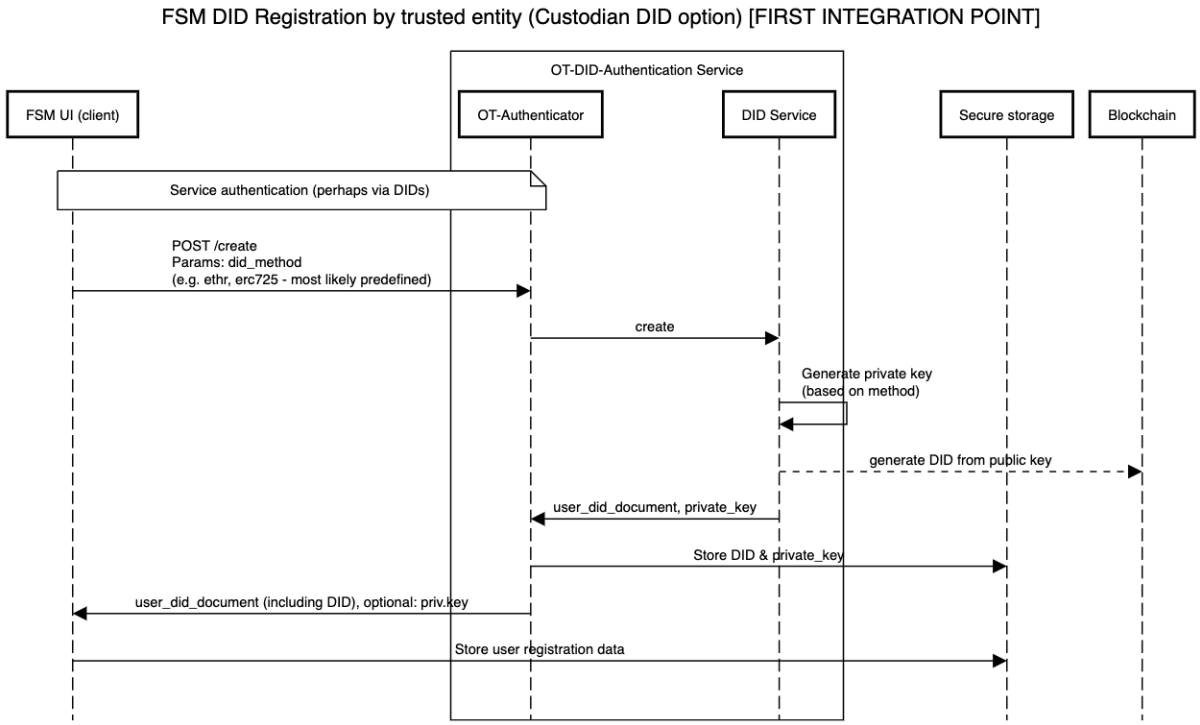

Figure 3-9: User registration sequence diagram.

User Authentication

User sends a username/password to check if the user has an existing DID registered with the platform in the secure storage.

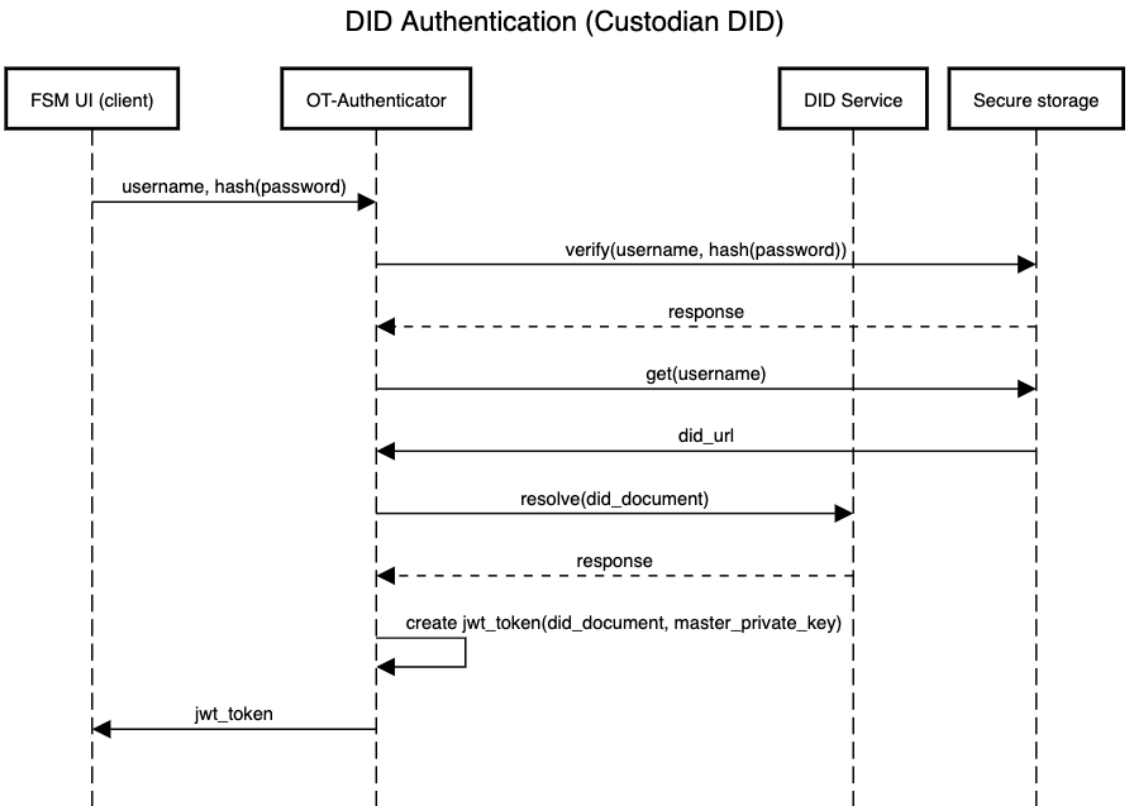

Figure 3-10: User authentication sequence diagram.

JWT verification with DID authentication

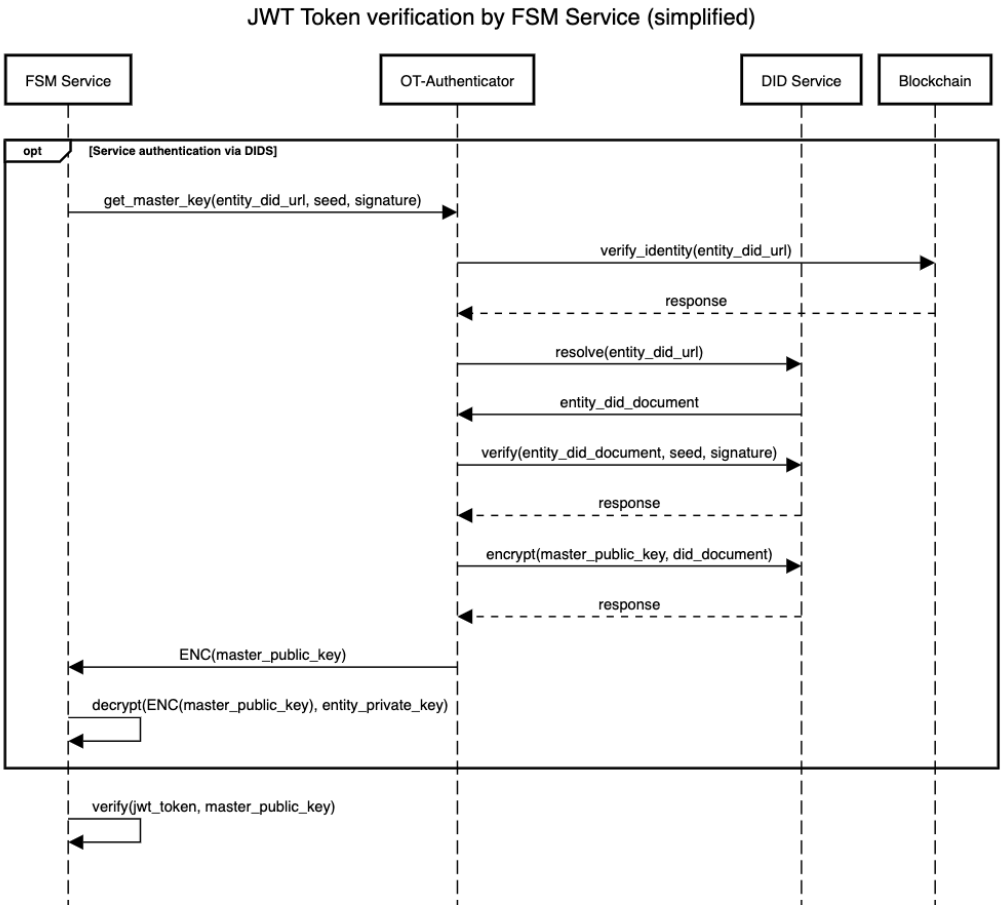

Figure 3-11: User authentication #2 sequence diagram.

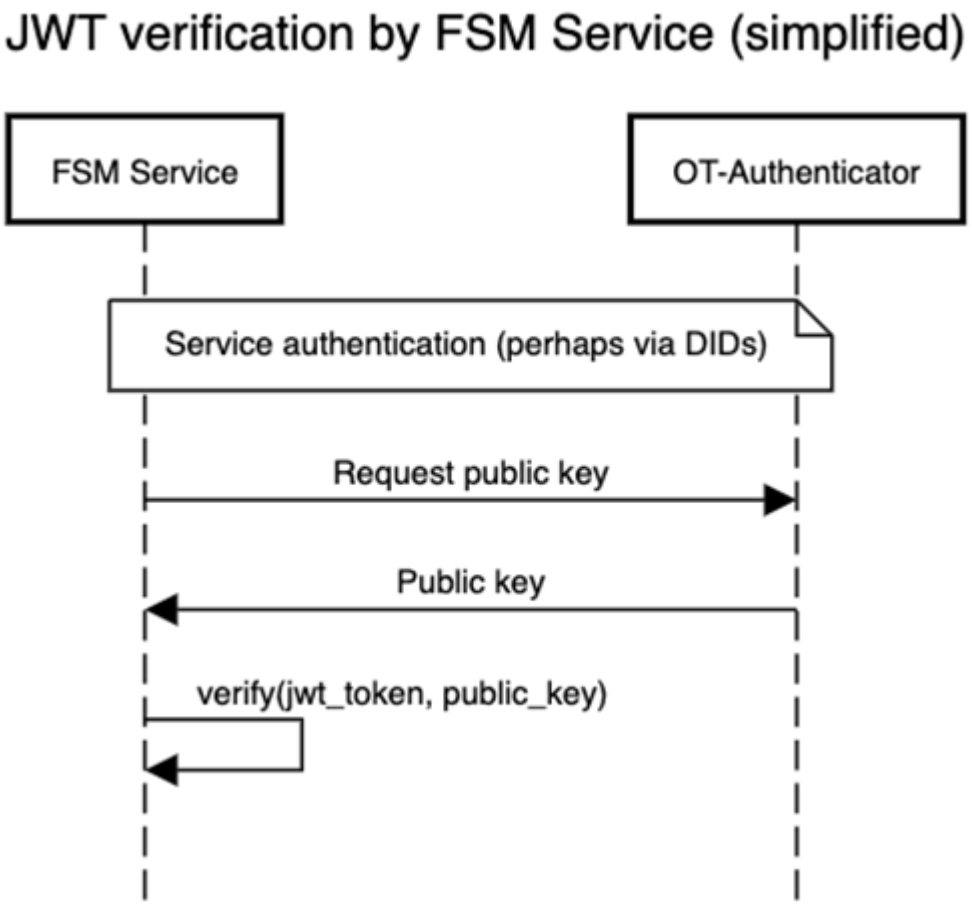

Figure 3-12: User authentication #3 sequence diagram (further simplified).

Finally, we provide the sequence diagram illustrating the trusted purchase mechanism (i.e. the FairSwap Protocol), which is part of IT5.

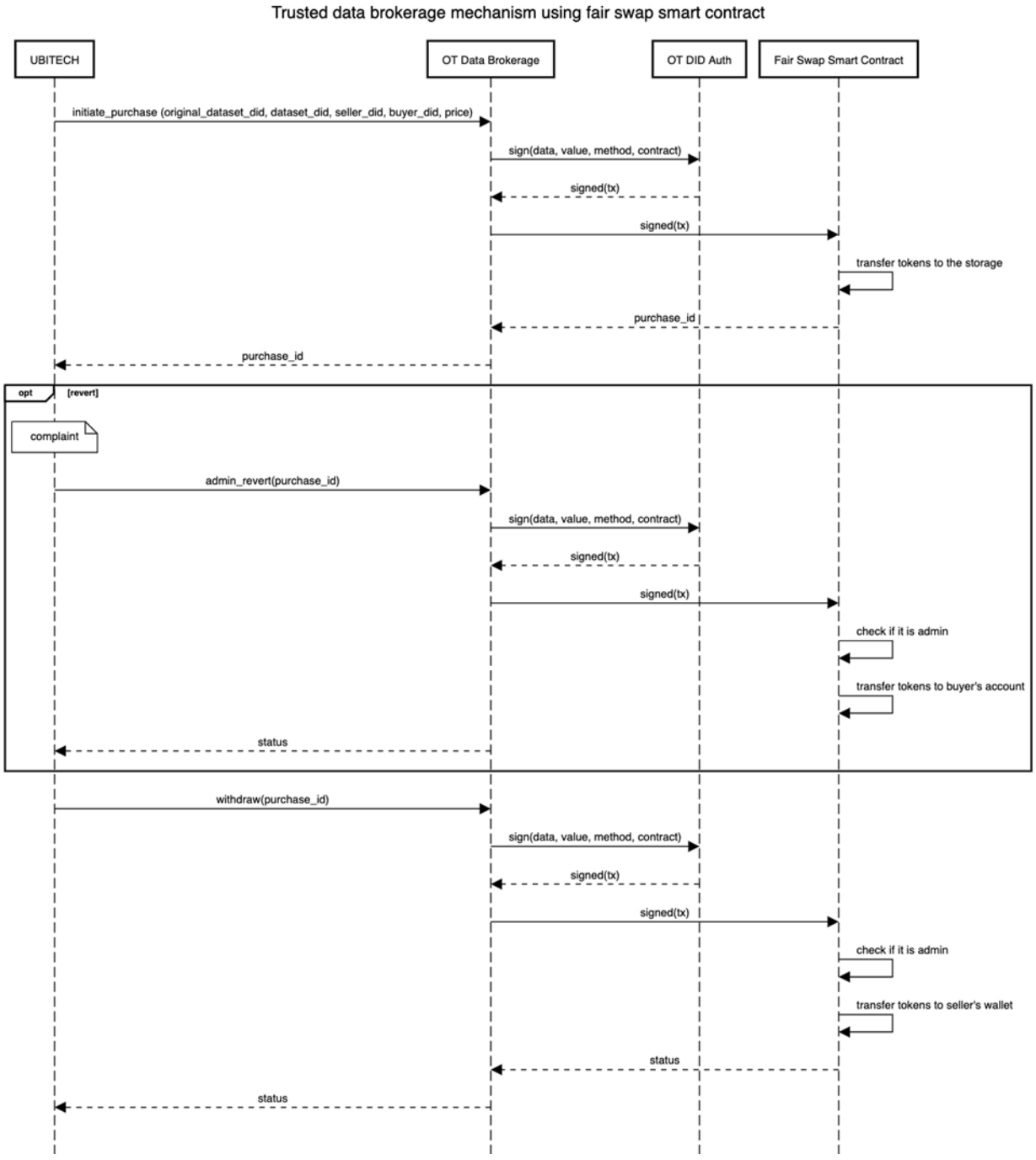

Figure 3-13: Trusted data brokerage mechanism (FairSwap protocol).

4 TECHNICAL EVALUATION

For the technical evaluation of the FSM platform, the "Product Quality Model" from ISO/IEC 25010:2011[7] was used as guidance for the generation of the technical KPIs of the Platform. Each aspect of the model was examined whether and how it covers the requirements of the project. Based on this analysis, a number of relevant KPIs were identified. The next section examines which of the Product Quality Model categories are relative to the project, while the section after describes the specific, project-related KPIs.

4.1.1 Product Quality Model Categories

Table 8 showcases the sub-characteristics of each category of the Product Quality Model which are relevant to the FSM platform.

Table 8: Product Quality Model Categories

Sub-characteristics	Definition	Relation to TheFSM
Functional suitability		
Completeness	Degree to which the set of functions covers all the specified tasks and user objectives.	YES
Correctness	System provides the correct results with the needed degree of precision.	YES
Appropriateness	The functions facilitate the accomplishment of specified tasks and objectives.	YES
Performance efficiency		
Time behavior	Response, processing times and throughput rates of a system, when performing its functions, meet requirements.	YES
Resource utilisation	The amounts and types of resources used by a system, when performing its functions, meet requirements.	YES
Compatibility		
Co-existence	Product can perform its functions efficiently while sharing environment and resources with other products.	YES
Interoperability	A system can exchange information with other systems and use the information that has been exchanged.	YES
Usability		
Ease of Use	System has attributes that make it easy to operate and control.	YES
User error protection	System protects users against making errors.	YES

[7] https://www.iso.org/standard/35733.html

User interface aesthetics	User interface enables pleasing and satisfying interaction for the user.	YES
Technical Accessibility	System can be used by people with the widest range of characteristics and capabilities.	YES
Reliability		
Maturity	System meets needs for reliability under normal operation.	YES
Availability	System is operational and accessible when required for use.	YES
Fault tolerance	System operates as intended despite the presence of hardware or software faults.	YES
Security		
Confidentiality	System ensures that data is accessible only to those authorized to have access.	YES
Integrity	System prevents unauthorised access to, or modification of, computer programs or data.	YES
Non-repudiation	Actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later.	YES
Authenticity	The identity of a subject or resource can be proved to be the one claimed.	YES
Maintainability		
Modularity	System is composed of components such that a change to one component has minimal impact on other components.	YES
Reusability	An asset can be used in more than one system, or in building other assets.	YES
Testability	Effectiveness and efficiency with which test criteria can be established for a system.	YES
Portability		
Installability	Effectiveness and efficiency with which a system can be successfully installed and/or uninstalled.	YES (for third-party integration)

To validate that the platform fulfills adequately all technical requirements, a set of technical methodologies and tools were used, and these include:

- W3C Mark-up Validation Service for HTML/XTHML and CSS validation in the context of technical quality assurance.
- Sonar for the integrated platform Quality Assurance

- Unit Testing on discrete platform components
- Integration testing on the integrated platform

4.1.2 Technical KPIs of the FSM Platform

Based on the tables presented previously, which reveal which criteria could be measured during the FSM operation and, based on the fact that the ISO/IEC 25010:2011 standard does not define specific attributes (measuring ways) for each one of the sub-characteristics, the following list of KPIs has been defined to allow the technical evaluation of the FSM platform.

Table 9: FSM Technical Evaluation KPIs

Sub-characteristics	KPIs	Calculation Type	Recommended Limit	Value	Mandatory/ Optional	Comments
Functional suitability						
Functional completeness	Percentage of completed Use Cases / Usage Scenarios	(Completed Use Cases / Defined Use Cases) * 100 %	100%	100%	M	For v2.0 a set of use cases are executed. Final version will cover all use cases.
Functional correctness	Percentage of Use Cases without reported bugs, after tests	(Completed Use Cases without bugs / Defined Use Cases) * 100 %	>90%	95%	M	No critical issues for v2.0. To be further analyzed during development.
Functional appropriateness	Percentage of tasks accomplished / Tasks Defined	(Accomplished Tasks / Tasks Defined) * 100%	>90%	100%	M	No critical issues for v2.0. To be further analyzed during development.
	Performance efficiency					
Time behaviour	Average Latency	(Total Response Time)/(No. of Requests)	<= 1 sec	<1 sec.	M	All requests are executed in 0,5 -1 sec. Pagination and DTOs utilized everywhere to improve UI/UX as well.
Resource utilisation	Mean % CPU Utilisation	(Σ (% CPU utilisation	<60%	<30 %	M	No critical issues for v2.0. To be

		probes))/(No. of probes)				further analyzed in final version. CPU utilization temporarily gets to ~80% when processing datasets for semantic enrichment, but this is normal and expected behavior. Average CPU consumption still remains as recorded here.
	Mean Memory Usage	(Σ (RAM Megabytes used in each probe))/(No. of probes)	<60%	<24 %	M	No critical issues for v2.0. To be further analyzed in final version. Memory usage temporarily gets to ~60% when processing datasets for semantic enrichment, but this is normal and expected behavior. Average memory consumption still remains as recorded here.
	Compatibility					
Co-existence	Ability to host in a single environment	Can the FSM components operate in a shared environment? YES/NO	YES	YES	O	Virtualization and dockerization techniques are applied. All components are deployed via a single docker-compose file (UI, API, swagger

						documentation, databases).
	Usability					
Ease of Use[8]	Number of clicks required to reach requested information	Number of clicks required to reach required information	<3	~2-3 clicks	M	Nielsen's principles kept in mind when designing the UI/UX.
User error protection	System crash on user errors	Does the whole crash on user errors? YES/NO	NO	NO	M	Code handling is enforced in all provided backend services, as well as detailed error messages which are properly handled and displayed to the user through the UI, in simple non-technical terms. All user input is filtered at all times to prevent errors of input and/or background operations which should proceed uninterrupted for their entirety.
User interface aesthetics	User Interface Accessibility standardisation	No. of User Interface Accessibility standards followed	1	1	M	The UI follows Nielsen's principles in terms of clicks required for all actions, as well as avoiding color polluting. Furthermore, the UI aims to be intuitive

[8] Behavioral characteristics are part of the quality in use model, here only metrics that can be automated extracted are mentioned

						and easy to use, with clear buttons and/or icons.
Technical Accessibility	Cross-Platform Accessibility	The FSM platform is accessible and operational through different platforms (e.g. Windows / MacOS / Linux)	YES	YES	M	Due to using a docker deployment, the entire infrastructure is completely OS agnostic.
	Cross-Browser Accessibility	The FSM platform is accessible and operational through different browsers (e.g. Chrome / Firefox/EDGE)	YES	YES	M	The platform has been thoroughly tested for proper support in Microsoft Edge, Google Chrome, Safari, Mozilla Firefox, Brave, Vivaldi.
	Cross-Device Accessibility	The FSM platform accessible and operational through different devices (e.g. PC / Laptop/ Tablet / Smartphone)	YES	YES	M	No critical issues for v2.0. UI is designed to function properly in smaller screens as well (mobiles, tablets).
	Reliability					
Maturity	Max. Concurrent Users Supported	No. of Max. Concurrent Users Recorded	>100 users	>120 users	M	Achieved in simulation through automated stress testing for v2.0
	Simultaneous Requests	No. of Simultaneous Requests (e.g. page open)	>75	>110	M	Achieved in simulation through automated stress testing for v2.0
Availability	% Monthly Availability	1- ((Downtown Time	>90%	>99 %	M	No critical downtime observed for v2.0.

		Minutes)/(Month Days*24*60))				
	Error Rate	(No. of Problematic Requests)/(Total Number of Requests)	<10%	~2%	M	No critical issues for v2.0. Most error cases are handled in a reversible and non-intrusive way for the user. Additionally, user input is preserved to avoid forcing them to re-insert everything again, when possible.
Fault tolerance	Number of Software problems identified without affecting the platform	No. of Non-Critical Software Errors	<10	4	M	No critical issues for v2.0.
	Security					
Confidentiality	Incidents of ownership changes and accessing prohibited information	No. of incidents recorded	0 (zero)	0	M	No critical issues for v2.0. In fact, the access policies provide fine-grained control for all APIs, datasets, ownership privileges etc. Datasets are always encrypted at transit, maximizing confidentiality.
Integrity	Incidents of authentication mechanism breaches	No. of incidents recorded	0 (zero)	0	M	No critical issues for v2.0.
Authenticity	Level of User Authenticity	Can you identify whether a	YES	YES	M	Strict authentication and

		subject is the one it claims to be? YES/NO/Partially				authorization pipeline per request via the Security Layer and ABAC policies. All requests are forced to pass through these two layers, ensuring that all requests are properly authenticated and authorized.
	Level of Data Resource Authenticity	Can you identify whether a resource is the one it claims to be? (or e.g. is it created by robots) YES/NO/Partially	YES	YES	O	Integration and exchange of information is encrypted and token protected.
	Maintainability					
Modifiability	% of Update Effectiveness	(No. of updates preformed without noticing operational problems)/(No. of updates performed)	>75%	100%	M	All updates were performed smoothly for v2.0.
Testability	Level of Testing	Are tests able to probe the FSM platform behavior? YES/NO/Partially	YES	Partially	O	End-to-end unit tests and per component tests exist for v2.0. Data exchange scenarios, monetization pipelines and relevant features exist in a mock testing environment. As

						more and more use cases are covered, more tests will be written and existing tests will be adapted to probe the entire platform's behavior.
	Portability					
Adaptability	Mean No. of Errors per Installation	(No. of Total Errors recorded during Installations)/(Total No. of Installations)	<1	0	O	No installation errors were recorded for v2.0.

5 CONCLUSIONS

In the current document we presented the integration strategy of TheFSM Platform which includes the development, testing and deployment strategy for TheFSM technical solution. The iterative process of the project yielded an agile approach which is well adopted by the integration strategy. Meeting this paradigm required the CI/CD pipeline for code development and integration. Containerization and state-of-the-art tools are used to support the aforementioned process. The integration points (along with their APIs description) implemented and tested in the current version cover the core and fundamental functionalities of the platform. The testing and technical evaluation of the platform also, provided with positive feedback regarding the technical readiness of the released version of TheFSM Platform. As more use cases are covered by the platform, these tests will evolve, adapt and carefully cover the newly added functionality. Moreover, we describe issue tracking and the dockerization process we follow during development.

REFERENCES

[1] An Overview of the Testing Process | Preface.
https://flylib.com/books/en/2.174.1/an_overview_of_the_testing_process.html

[2] http://www.technofunc.com/index.php/erp/178-what-is-integration-testing

ANNEX I

Annex I provides the integration plan until M24.

Relevant task in DoA	Component	Subcomponent/Action	Responsible partner	Output (API, Prototype, UI, algorithm etc.)	Technology/Framework	version	Planned date of delivery	Contractual date of delivery	Project month
T2.1	TheFSM Semantic Model		SAI	Data model		v2.0	31/1/2022	31/1/2022	M24
T2.2	Semantic Mapper	OntoRefine tool	SAI	Demonstrator		v2.0	15/12/2021	31/1/2022	M24
		Apollo Federation	SAI	API		v2.0	31/1/2022	31/1/2022	M24
T2.2	Data Handler	ETL pipeline	SAI	Demonstrator		v2.0	15/12/2021		
		Automated ETL pipeline	SAI	API	Springboot application and REST API	v1.0	15/12/2021		M24
		Reconciliation service	SAI	API	Springboot application and REST API	v1.0	15/12/2021		M24
		WoT, EPCIS parsing	SAI	Demonstrator		v2.0	15/06/2022		
T2.2	Data Staging	File Storage	SAI			v2.0			
		RDBMS	SAI	API		v2.0	15/06/2022		
		Distributed MinIO	UBITECH	API	MinIO containers deployed behind nginx load balancer	v1.0	20/11/2021		M24
T2.2		GraphDB	SAI	API		v2.0	15/12/2021		

	Secure Storage and Indexing	MongoDB	SAI	API		v2.0			
		Elastic search	SAI	API		v2.0	15/12/2021		
T2.2	Data sources and application catalogue	Extensions to SOML	SAI	Data model		v2.0	15/06/2022		
		Wrappers of external data sources (API)	SAI	API		v2.0	15/06/2022		
T2.2, T3.4, T5.3	Data Licence and Agreement Management	Backend access policies for purchase	UBITECH	API	Springboot application and REST API	v1.5	15/01/2022	31/1/2022	M24
		UI implementation	UBITECH	UI		v2.0	30/3/2022		
T2.3	AI Models	Supplier & Product Risk Assessment Models	AGROKNOW	Part of the Agroknow Data Platform. It is provided through API	Django. Python	v2.0	30/10/2020	31/1/2022	M24
		Incident Prediction Models		Part of the Agroknow Data Platform. It is provided through API	Django. Python	v2.0	15/12/2020	31/1/2022	M24
		Risk prediction models		Part of the Agroknow Data Platform. It is provided through API	Django. Python	v2.0	15/12/2020	31/1/2022	M24
		Supplier risk prediction models		Part of the Agroknow Data Platform.	Django. Python	v2.0	30/11/2021	31/1/2022	M24

				It will be provided through API					
T2.3	Query Explorer	Ontotext Platform GraphQL Playground	SAI	API	Springboot application and REST API	v2.0	30/06/2022		
		Ontotext Platform Semantic Object Service		API	Springboot application and REST API	v2.0	30/06/2022		
		Ontotext Platform Search Service		API	Springboot application and REST API		31/06/2022		
T3.4	OTNode DLT Services	Identity Hub/Identifiers Registry	PROSPEH			v2.0	04/01/2021		
		DID Resolver				v2.0	04/01/2021		
		DID Provisioning Service				v2.0	04/01/2021		
	User authentication API	API				v2.0	15/01/2021		
	OTNode DLT Interfaces	DLT data management interface				v2.0	04/01/2021		
						v2.0			
						v2.0			
T3.4	Data Brokerage Model	Monetization backend service and data exchange service	PROSPEH	Data model		v2.0	01/10/2021		M24
	Data Brokerage	Blockchain FairSwap protocol, tokens management,	PROSPEH	API	Test Ethereum network, smart contracts,	v2.0	01/10/2021	31/1/2022	M24

	Engine (FairSwap)	monetization backend service and data exchange service			Springboot application and REST API				
T3.3	Attribute Based Encryption	Data encryption to ensure secure data at transit	UBITECH	Service	Springboot backend REST service	v2.0	01/09/2021	31/1/2022	M24
T3.2	A2C Engine	Definition of access policies per application	UBITECH	Table/template	Springboot application and REST API	v2.0	01/09/2021	31/1/2020	M24
		Access Policy Management		API service & prototype with UI for policies definition per data source/application	Springboot application and REST API	v2.0	01/09/2021	31/1/2020	M24
		Policy Enforcement Business Logic				v2.0	01/09/2021	31/1/2020	M24
		Access Request Transformation Handler				v2.0	01/09/2021	31/1/2020	M24
		Attributes Handler				v2.0	01/09/2021	31/1/2020	M24
		Access policies UI				v2.0	01/09/2021	31/1/2020	M24
T2.3, T3.4, T5.3	Anonymization Framework	Anonymization policies API	UBITECH	Microservice based on the ARX framework	Springboot api	v2.0	01/10/2021	31/1/2020	M24
		Anonymization API	UBITECH	REST API for integration	REST API & Swagger installation	v2.0	01/10/2021	31/1/2020	M24
		PIIs and Policies Administrator	UBITECH	Table/template		v2.0	01/10/2021		M24
T3.5	TheFSM Data Platform UI		UBITECH	UI		v2.0	11/30/2020	31/1/2020	M24
	TheFSM Data		AGROKNOW						

	Marketplace UI								
	Analytics tools					v2.0			
	REST API		UBITECH	API	Springboot application and REST API	v2.0	01/10/2021		M24
	API Gateway		UBITECH	API	Springboot application and REST API	v1.0	01/10/2021		M24
T4.2.2	FOODAKAI 2.0	FSI Data Platform extensions	AGROKNOW	API	python, Django and elasticsearch	v2.0	1/2/2021		M24
		UI extensions		UI	Ruby on Rails, React JS, Redux	v2.0	26/2/2021		M24
T4.2.3	Agrivi 2.0	Auditor/certifier/consultant interface	Agrivi	UI	.NET, .NET MVC, Vue, Angular	v2.0	2/12/2021	M24	M24
		New weather partner integration	Agrivi	API, UI	.NET api, Vue, Angular	v3.0	31/06/2022	M36	M24
		Traceability report and tracking improvements	Agrivi	API, UI	.NET api, Vue, Angular	v2.5	31/04/2022	M37	M24
		Documents feature improvements - API access policies	Agrivi	API, UI	.NET api, Vue, Angular	v2.5	31/03/2022	M38	M24
		Product database development	Agrivi		Database/MongoDB	v2.5	31/05/2022	M39	M24
		Integration with API gateway	UBITECH	API	.NET api	v2.0	M24	M24	M24
T4.2.1	Food Inspector	Company Dashboard	AGROKNOW	UI	Ruby on Rails, React JS, Redux	v2.0	M16	M24	
		Inspector Dashboard	AGROKNOW	UI	Ruby on Rails, React JS, Redux	v2.0	M16	M24	
		Dialy Alerts	AGROKNOW	UI	Ruby on Rails, React JS, Redux	v2.0	M17	M24	

		Hazards Dashboard	AGROKNOW	UI	Ruby on Rails, React JS, Redux	v2.0	M19	M24	
		Risk Dashboard	AGROKNOW	UI	Ruby on Rails, React JS, Redux	v2.0	M22	M24	
		Agrivi 2.0 Integration	AGROKNOW	API	python, Django and eslasticsearch	v2.0	M22	M24	
		GLOBALG.A.P. Integration PoC	AGROKNOW	API	python, Django and eslasticsearch	v2.0	M24	M24	
		Integration with API gateway	UBITECH			v2.0	M24	M24	M24
T4.2.2	FOODAKAI 2.0	FSI Data Platform extensions	AGROKNOW	API	python, Django and elasticsearch	v2.0	M22	M24	M24
		Sourced Ingredients Risk	AGROKNOW	UI	Ruby on Rails, React JS, Redux	v2.0	M16	M24	
		Weekly Insights	AGROKNOW	UI	Ruby on Rails, React JS, Redux	v2.0	M16	M24	
		My Suppliers Dashboard	AGROKNOW	UI	Ruby on Rails, React JS, Redux	v2.0	M18	M24	
		Supplier Reports	AGROKNOW	UI	Ruby on Rails, React JS, Redux	v2.0	M21	M24	

Figure Annex I – 1: Integration Plan until M24