WP4 INTERLINK Platform

D4.2 – Reference architecture model and specification

The INTERLINK project is co-funded by the European Union.
Horizon 2020 - DT-GOVERNANCE-05-2020 - Grant Agreement No 959201

Project acronym	INTERLINK
Project full title	Innovating goverNment and ciTizen co-dEliveRy for the digitaL sINgle marKet
Call identifier	DT-GOVERNANCE-05-2020
Type of action	RIA
Start date	01/01/2021
End date	31/12/2023
Grant agreement no	959201

WP	4. INTERLINK Platform
Author(s)	Serge Sushkov Evdoshenko, Rebeca Garcia Betances (TREE), Raman Kazhamiakin (FBK), Diego López de Ipiña González de Artaza, Ana Ortiz de Guinea López de Arana (DEUSTO)
Editor(s)	Serge Sushkov Evdoshenko (TREE)
Reviewers	Pauli Misikangas, Olli Pelkonen (CNS), Quentin Fontaine (UCL), Giulia Degli Esposti, Nicola Cracchi Bianchi (DEDA)
Leading Partner	TREE
Version	v1.0
Deliverable Type	OTHER
Dissemination Level	PU
Date of Delivery	31/12/2021
Submission Date	31/12/2021

VERSION HISTORY

Version	Issue Date	Status	Changes	Contributor
v0.1	2021-11-26	Draft	Initial version by Serge Sushkov	TREE
v0.2	2021-12-07	Pre-final	Consortium comments incorporated	TREE + ALL
v1.0	2021-12-27	Final	Revisions from internal reviewers and final corrections	TREE

Glossary

ENTRY	DEFINITION
INTERLINKERs	Common building blocks, provided as software tools or in the form of knowledge offered digitally, that represent interoperable, re-usable, EU-compliant, standardized functionality for the co-production of public services
Public Service	Services that are publicly available and are provided by the government or on behalf of the government's residence in the interest of its citizens. In INTERLINK we focus not only on the software services (i.e., the services delivered digitally) but also the services that rely on digital technologies.
Software Platform	A platform is a group of technologies that are used as a base upon which other applications, processes or technologies are developed. In other words, a platform is the basic hardware (computer) and software (operating system) on which software applications can be run. This environment constitutes the basic foundation upon which any application or software is supported and/or developed. Within the context of the INTERLINK project, we define a Software Platform as a set of data storage, backend services and APIs which serve as a basis for the business logic and frontend applications to develop, integrate and function. It also includes SW deployment and operational infrastructure.
Software Backend	Is part of software services and/or applications running on server side within the client-server paradigm. It mostly dedicates to data storage, business logic, process workflow and utility functions
Software Frontend	Is part of the software services and/or applications running on the client side within the client-server paradigm. It mostly focuses on graphical user interface (GUI), workflow navigations and supporting business logic
SW API	API means Application Programming Interface, a type of software interface, offering a service to other pieces of software.

ACRONYMS

ABBREVIATED	EXTENDED
ACC	Authentication and Authorization Control (software)
API	Application Programming Interface
DB	Database
CRUD	Create Read Update Delete (basic SQL operations)
FSM	Finite State Machine (technique)
GUI	Graphical user interface
PA	Public Administration
PS	Public Service
SW	Software

Table of Contents

List of figures

List of tables

Executive summary

Public service (PS) co-production platform is one of the key parts of the INTERLINK project software (SW) implementation. This document describes the SW platform for public services co-production processes, which includes:

- the backend part of the public service co-production web portal (the frontend)
- as well as the infrastructure and operational environment for the project SW corresponding to public service co-production processes.

The platform architecture has been elaborated following the methodologies described in ***Section 1***.

It is based on the users, functional and non-functional requirements which are described in Deliverable 4.1 and in ***Section 2*** of this document.

Section 3 describes the global view on the project SW, including both the platform backend and the frontend (the web portal, mostly in the sense of its relation to the backend).

Section 4 focuses on the platform backend architecture and describes their components in detail.

Section 5 describes the SW deployment and operational environment.

Section 6 covers the communication network and security related issues.

This document has been elaborated based on results of numerous meetings and workshop discussions within the project consortium and has been produced jointly by the IT teams of the project participants.

1 Introduction and methodologies

1.1 Task T4.2 and deliverable D4.2

This deliverable D4.2 is the result of the work on the corresponding task T4.2 "Platform Architecture" of the work package 4 (WP4) of the project. The task 4.2. objective is to elaborate a complete technical design of the INTERLINK architecture which supports the technological enablers (building blocks called INTERLINKERs) provided by WP3.

The work on task 4.2 is closely related to the tasks T3.1, T3.2, T3.3 and T3.4 of WP3, which define the list of INTERLINKERs (T3.1), provide the general enablers (T3.2, T3.3) and collaborative tools (T3.4) for harmonizing the interconnections among them in the platform architecture. As described below in details, the platform being specified in this document corresponds to processes of co-production of public services (where only a few of the INTERLINKERs will run) and not covering explicitly processes of public service operations (where majority of INTERLINKERs are supposed to be used). But since these processes are very similar to each other, they also should share the same architectural approaches and design patterns for the platform level, and imply the same integration requirements onto the set of INTERLINKERs and onto the web frontend level for both public service co-production and operations.

This task T4.2 is also closely related to the work of defining and setting up the integration and platform operational framework (T4.3) which will be mentioned in Chapter 6 of this document. The infrastructure, hosting and data storage may be considered as the lowest level of the platform architecture itself.

Task 4.2 is also very much interconnected with the design and implementation of the common project web portal (the frontend, T.4.4). In fact, several INTERLINKERs will serve as a system-wide IT services, like authentication, user management, process workflow, asset management, etc. Despite all of those may have their own frontend components, their functionality and business logic is of the lower level w.r.t other INTERLINKERs, because they serve as a basis for functionality of others. Thus, such type of INTERLINKERs we call as "platform-type" and they are analysed rather within the scope of the platform architecture than within the scope of web frontend, or at least, very much together as for the case of the workflow manager, which also is the core block for elaboration of the Data Model common to the platform and the web portal.

The platform architecture also needs to meet the socio-technical requirements defined in T4.1 and D4.1, dealing with social, legal and business acceptability aspects of public services. The design of the web portal will furthermore incorporate user stories developed to ease the integration and ensure downstream usability of INTERLINK services on the use cases side. Those requirements are mentioned in more detail in Chapter 2 of this document.

1.2 Review of the project software

As described in detail in Chapter 6 of the project Deliverable 4.1, the entire project software will be grouped into the several large areas:

- a set of SW for ***co-production process of public services*** (which may be re-used in some cases of ***co-production of INTERLINKER modules*** as well), including
  - ***SW platform*** (backend);
  - ***web portal*** (frontend) representing so called Collaborative Environment for the end users;
  - a special interactive graphical ***wizard*** which is one of the key part of the web portal (frontend) and is intended to guide users on the public service co-production process;
- a set of SW for ***execution of public services*** ("implementation" or "delivery" in the sense of "operating a public service"), including
  - PS execution platform SW (backend);
  - and web portal (frontend) for PS execution;

  those service execution platforms and operation environment in general are out of scope of the current INTERLINK project – except the 3 pilot use cases, for which we still carry the responsibility including demo execution of the public services for the pilot cases;
- a set of ***SW modules called INTERLINKERS***, which in turn may be of different types:
  - co-production platform INTERLINKERS which are essential for the Collaborative Environment,
  - and service execution (service "implementation") INTERLINKERS which are essential for particular public service;

  The INTERLINKERS SW modules are specified in a separate Deliverable document D3.1.
- A set of additional SW modules and services dedicated to administration, operation and maintenance of the platform and web portal themselves, which may be connected to most for other SW components, are for example:
  - operations, startup, shutdown, etc for the entire system (platform + frontend) and INTERLINKERs;
  - logging, monitoring and dashboarding for the system operations and for the business logic;
  - data storage and cloud hosting administration and maintenance;
  - super-user level operations and technical support for the web portal functionalities (grant admin privileges, ban users, etc).

1.3 The PS co-production platform

The project SW related to the public service co-production process will consist of multiple parts developed independently but integrated and interoperating together in order to provide users with the so-called Collaborative Environment where public administrations, private companies and active citizens will co-design together a new public service.

Many functional blocks of this SW will consist of both backend (server side) and frontend (web GUI side) parts, however some of those may be considered as being rather frontend (with essential functionality of users GUI, visualization, navigation, etc.) while other as being rather backend of platform-wide (with the functionality focused on representing the process flow, the data model, auth service, etc.).

From the user perspective, the globally main function of the public service co-production software is to provide users with the Collaborative Environment and to implement and control the phases of the public service co-production process, as well as to provide users with accessibility and operations over catalogue of the available INTERLINKER building blocks (the INTERLINKERs) and catalogue of available co-designed public services.

Thus, for the PS co-production platform we have pre-selected the following components as the core platform parts:

- authentication and authorization service;
- user and team management;
- co-production process workflow manager;
- asset manager together with intermediate layers to interface to various possible data storage services;
- and logging and monitoring services.

The platform architecture is described in more details in Chapter 5 of the present document.

1.4 Partners involved in T4.2 and D4.2

The work on T4.2 and D4.2 is carried out by several project partners with the leading role of Tree Technology (Spain). The table below summarizes the list of partners and their contributions.

Table 1. Partners involved in T4.2 and D4.2

Partner	Controbution	Roles
TREE	8 PM	Leading the task & deliverable document
FBK	1 PM	Contributor
DEUSTO	1 PM	Contributor
DEDA	2 PM	Contributor

1.5 Project documents related to this one

The work on T4.2 and D4.2 is very much interconnected with other project tasks and several other deliverable documents. The table below summarizes the list of documents frequently referenced throughout this one.

Table 2. Related project documents

Doc. ID	Title	Comments
D3.1	Identification and specifications of INTERLINKERS	Described types of INTERLINKERs and functionalities for the most important of them
D4.1	List and description of the socio-technical requirements	Described in details user, functional and non-functional requirements for the project SW (both for the SW platform and INTERLINKERs)
D5.1	Use-case plans and guidelines v1	Describes deployment, operation and evaluation scenarios for the three pilot use cases

1.6 Methodologies used

The platform architecture has been elaborated based on the social and technical, functional and non-functional requirements, which were collected in the task T4.1 and are described in detail in D4.1 document and summarized in Chapter 2 of this document.

For the architecture diagrams, the multi-layer and microservices design patterns have been used, as described in details in Chapter 5.

The interfaces between the backend, the frontend, internal and external SW services have been designed with REST HTTP protocol and have been specified using the OpenApi standard.

The Data Model has been elaborated at three levels: the global one (Conceptual DM), the intermediate (Logical Level) and the detailed one at the level of DB implementations (Physical Level).

The SW development for the PS co-production platform will be carried out in close collaboration with development of the web portal frontend, because most of the SW modules have both frontend and backend parts. Agile methodology will be used for the SW development work as the most suitable in the circumstances of the INTERLINK project, where we deal with large amounts of SW components and a high level of uncertainties in functionality specification per particular SW module (among possible alternatives).

2 Socio-technical requirements

2.1 Platform architecture requirements

The general socio-technical requirements on INTERLINK software and INTERLINKER SW modules are described in D4.1 document. As mentioned in Section 5.2 of D4.1, the major functional requirements for INTERLINK public service co-production platform are:

- To host a catalogue of INTERLINKERs (as building blocks for public services)
- To use co-production INTERLINKERs in the co-production process of public services
- To allow the configuration and binding together of public service operational INTERLINKERs during the public service co-production process
- To implement and manage processes within the co-production of public services
- To store newly created public service (digital objects) into a catalogue of public services for potential reuse
- To serve as architectural template for the public service execution platform as a possible extension of the public service co-production platform

Besides that, the PS co-production platform should have an open and extendable architecture, so more platform-type SW modules (INTERLINKERs) could be added, in particular use case deployments. For example, the incentives SW module may be represented only by the activity logging and tracking component in the minimal platform version, but can be extended with use case specific reward SW module(s) when needed.

The public service co-production platform should also satisfy the following non-functional requirements:

- ***Reliability***: both platform core back-end and platform front-end parts, as well as the public service co-production INTERLINKERs, should be available in 24/7 mode with minimal manual interventions (e.g. being able to restart automatically in case of power or infrastructure interruptions);
- ***Scalability***: the Docker/Kubernetes technologies should provide easy scaling up of SW services in case of increase of use activities;
- ***Maintainability***: the platform core and the platform tools should be designed and implemented in a way which minimizes manual actions on platform operations, which should be clearly logged and monitored;
- ***Configurability***: each component (core public service co-production process flow manager, platform tools, as well as all of the integrated INTERLINKERs) should store configuration metadata in a Project common database or data warehouse; the configuration metadata should be properly versioned, archived and easily operated with a configuration manager GUI tool to allow easy re-configuration of each new instance of the INTERLINK platform being deployed in a new EU region. In order to promote **component and service reuse** such metadata will be used to recommend or suggest suitable INTERLINKERs or existing public services depending on the user context.

2.2 Platform requirements onto INTERLINKER SW modules

Socio-technical requirements on the INTERLINKER SW modules are described in detail in documents D3.1 and D4.1. Here we mention those of them which arrive from the platform point of view.

All SW modules used in the PS co-production process must use the same common Data Model and DB implementation and API for the key DM entities (of course, each SW module can extend the base DM with additional module-specific entities).

Co-production platform should not only implement functionality of a particular platform type INTERLINKERs for users, but also allow possible functional interoperability of those SW modules between each other, when needed. For example, if two platform type INTERLINKERs may need a message queue for their internal or mutual functionality (for example, in a forum or chat SW modules), then the platform should host such message queue service for them, even if it is not used by the core platform functionalities (e.g. related to the workflow of the co-production process). Such interoperability SW services should preferably be of common technology among the SW modules, for example, it would be preferable that all SW modules use Apache Kafka or Rabbit MQ, instead of having all possible message queue solutions (Redis, etc.).

Other general requirements from the platform side onto SW modules are the following:

- the same technological stack should be used for most of the backend components (e.g. message queue) of the web portal except cases when implementation can be only in given different technology,
- the same type of data storage should be preferred between components when possible (e.g MySQL or PostgreSQL for SQL-type DB, not both),
- the same data access API should be shared among different frontend components when possible (especially for the key data entities of the Data Model),
- the same SSO authentication mechanism (JWT or cookies, etc.) should be used by all frontend components when possible, including the same authentication scenario if various alternatives are possible within the authentication service SW,
- similar approaches should be shared for configurability of SW components when possible, including same type storage for configuration data (e.g. XML or JSON),
- same logging formats should be used across all custom made SW components in order to have logs easily collected.

2.3 Web to platform integration requirements

From the user perspective, the Collaborative Environment for public service co-production processes will be represented as the common co-production web portal. The web portal in turn is a client side (frontend) part of the entire SW system, so the platform will serve as an infrastructure and service basis for that web portal, including the application logic and data access levels. Thus, integration of the backend and the web frontend part is of crucial importance within the platform architecture.

Similar to the requirements onto INTERLINKERs from the platform side, there are also the following requirements onto the web portal integration from the platform point of view:

- same key DM API should be used (for the key DM entities, especially related to users & team management and co-production process workflow),
- same technological stack should be used when possible for web backend components and backend (application logic) or INTERLINKERs SW modules (DB implementation, message queues, etc),
- scalability of the frontend (web load balancers) should be aligned with scalability of data storage layer which perhaps may be shared with other SW modules (e.g. MongoDB sharding),
- same deployment infrastructure (docker-compose or kubernetes, etc) should be used for the web server containers.

3 High-level architecture of the INTERLINK SW

3.1 Crowdsourcing nature of the INTERLINK project

According to the project proposal document, INTERLINK will overcome the barriers that hinder administrations to reuse and share services with private partners (including citizens) by combining the advantages of two often opposed approaches: (1) the “top-down” approach where Government holds primary responsibility for creating these services compliant with EU directives, sometimes seeking the support of citizens for specific design or delivery tasks; and (2) the “bottom-up” approach in which citizens self-organize and deliver grassroot services where government plays no active role in day-to-day activities but may provide a facilitating framework.

Given the squeeze in public funding, PAs are under continuous pressure to perform more efficiently and deliver faster and cheaper services to meet citizens’ and businesses’ needs. Besides that, in the era of Internet and mobile communications, it is hard to organize and manage all the public services only via the centralized top-down approach. At the growing scale of electronic and social communications, the only way to scale up the manageability of public services is to include the bottom-up approach as well. For these reasons, governance is undergoing a deep transformation by developing new approaches for delivering public services, often along with two main directions.

The European Commission with its Digital Single Market Strategy is leading this transformation which recognises that digital technologies have great potential to help PAs deliver better services for less.

In the world of Information Technologies (IT) such techniques are called crowdsourcing, which according to the Oxford dictionary is defined as “*the practice of obtaining information or input into a task or project by enlisting the services of a large number of people, either paid or unpaid, typically via the internet”*.

3.1.1 Crowdsourcing in the co-production process

For the processes of co-creation of SW INTERLINKERs and co-production of public services, the crowdsourcing nature appears more in the aspects of brainstorming for the service design and in the co-production activities organization. The solution for this at the IT level is to provide a special web portal focused on teamwork in all the phases of the service co-production, providing users with a **Collaborative Environment.**

This will be specified in more details during the work on task T4.4 “INTERLINK web portal frontend” and in deliverables D4.3, D4.4 and D4.5.

The co-production process and the Collaborative Environment are defined and described in detail in the D4.1 document.

3.1.2 Crowdsourcing functionalities of INTERLINKERS

The crowdsourcing techniques will be used the most in the processes of co-execution of public services (for those services when it applies, like transportation sharing, etc.).

The SW building block modules (INTERLINKERS) which will implement such kind of public service functionalities, need to be designed making use of the crowdsourcing techniques (probably, with high-performant message queues, server load-balancers, high-speed mobile communications, etc.).

For the co-production of such types of public services, this functionality nature and types of IT solutions should be reflected in the governance models, perhaps in the form of a service template.

For the co-execution of such type of public services, not only the INTERLINKERS modules should be implemented with crowdsourcing techniques, but the service co-execution platform itself should be compatible with such type of SW functionalities of the INTERLINKER modules running there, in terms of speed and scalability.

3.2 The three collaborative processes

As mentioned earlier in this document and in D4.1, there might be the following three processes within the INTERLINKER software:

1. co-creation of INTERLINKERS SW modules,
2. co-production of public services,
3. co-execution of those co-produced public services.

Those three processes have many similarities because they all are supposed to be related with crowdsourcing functions and to use Collaborative Environment in one or another (automated or manual) form.

However, strictly speaking, for particular running instances of INTERLINK software, those three processes most probably will have different workflow phases, different SW modules being executed and different customizations.

3.2.1 Co-creation of INTERLINKER SW modules

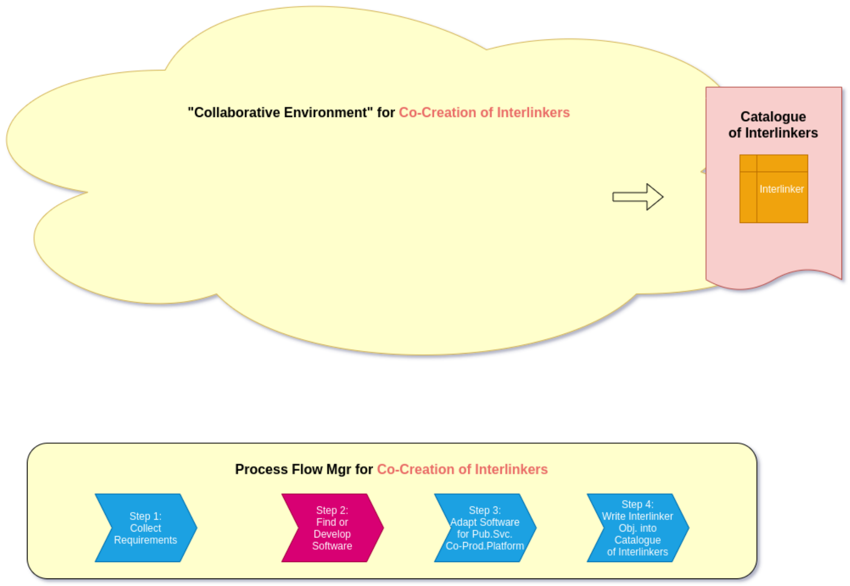

"Collaborative Environment" for Co-Creation of Interlinkers
Catalogue of Interlinkers
Interlinker
Process Flow Mgr for Co-Creation of Interlinkers
Step 1: Collect Requirements
Step 2: Find or Develop Software
Step 3: Adapt Software for Pub.Svc. Co-Prod.Platform
Step 4: Write Interlinker Obj. into Catalogue of Interlinkers

Figure 1. High-level diagram of processes of co-creation of INTERLINKERs

The process of co-creation of INTERLINKERs is shown in Figure 1. At high level, this process starts without any SW modules on the input and has an objective to create one of such SW modules (INTERLINKERs). This process in some cases may be using the same or similar Collaborative Environment as for the other processes (co-production and co-execution of public services), except cases when one of the platform-type SW modules being part of such Collaborative Environment is being created itself. In that latter case, such a SW module should be created without use of the collaborative web portal which does not yet exist. In some cases, the SW with the desired INTERLINKER functionality may already exist, so the INTERLINKER co-creation will be an insertion of metadata of this INTERLINKER into the catalogue of INTERLINKERs, without the creation (development) of INTERLINKER SW itself.

3.2.2 Co-production of public services

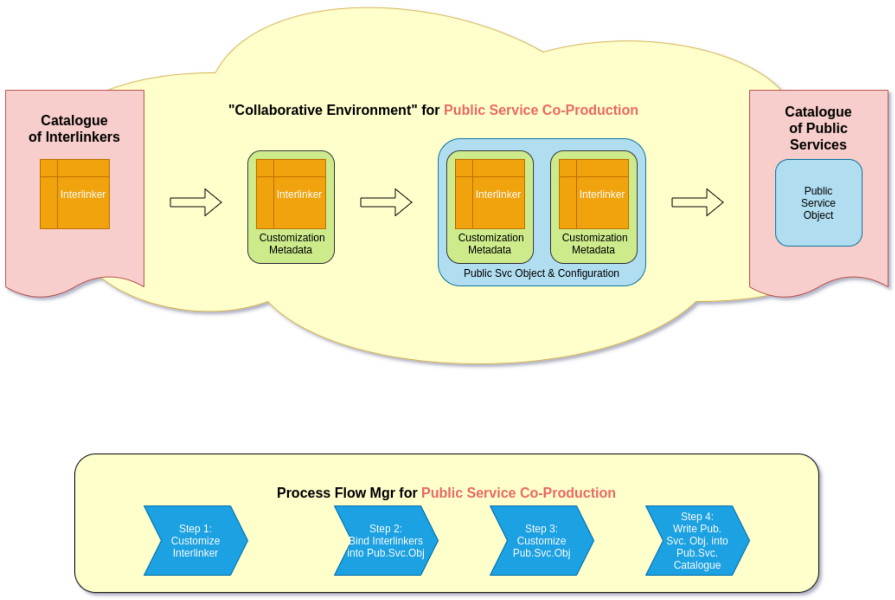

Catalogue of Interlinkers
Interlinker
"Collaborative Environment" for Public Service Co-Production
Interlinker
Customization Metadata
Interlinker
Customization Metadata
Interlinker
Customization Metadata
Public Svc Object & Configuration
Catalogue of Public Services
Public Service Object
Process Flow Mgr for Public Service Co-Production
Step 1: Customize Interlinker
Step 2: Bind Interlinkers into Pub.Svc.Obj
Step 3: Customize Pub.Svc.Obj
Step 4: Write Pub. Svc. Obj. into Pub.Svc. Catalogue

Figure 2. High-level diagram of processes of co-production of a public service

The process flow of co-production of a public service is shown in Figure 2. This process has elements of the catalogue of INTERLINKERs SW modules as an "input". Users pick up those SW (or non-SW, i.e. knowledge) INTERLINKERs from the catalogue based on suggestions provided by a special interactive Wizard GUI according to criterias and objectives of the future public service provided by users. The PS co-production process results in customization of those selected INTERLINKER SW modules, in wrapping them with additional metadata and binding them together in order to design the future public service. As the final result of this process, a new PS meta-object is being written into the catalogue of public services which is kind of an "output" of the process. This process type is the primary focus of the current project and this document.

3.2.3 Co-operation of public services

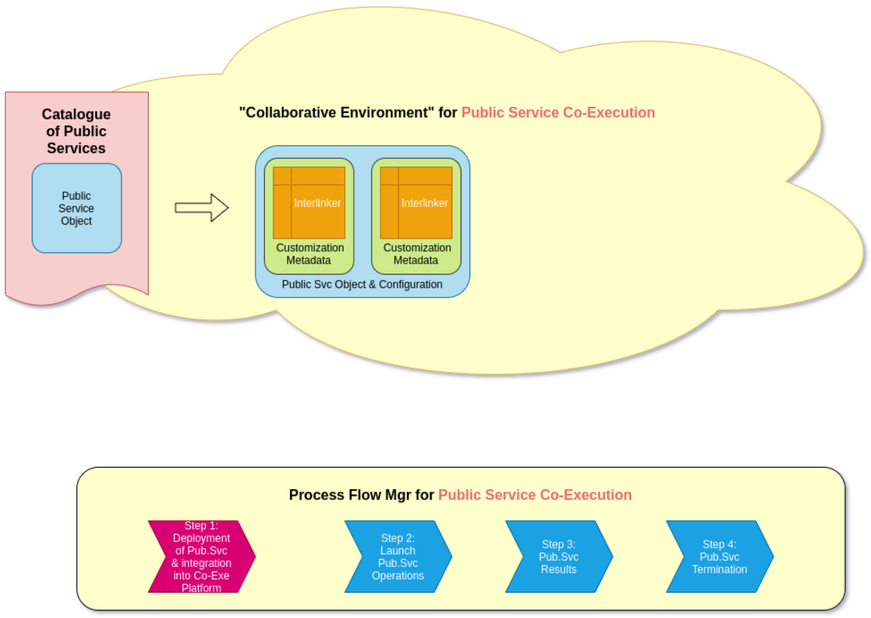

Catalogue of Public Services
Public Service Object
"Collaborative Environment" for Public Service Co-Execution
Interlinker
Customization Metadata
Interlinker
Customization Metadata
Public Svc Object & Configuration
Process Flow Mgr for Public Service Co-Execution
Step 1: Deployment of Pub.Svc & integration into Co-Exe Platform
Step 2: Launch Pub.Svc Operations
Step 3: Pub.Svc Results
Step 4: Pub.Svc Termination

Figure 3. High-level diagram of processes of co-execution of a public service

Figure 3 illustrates the process of co-execution (co-operation) of a public service. This process consists of picking up a desired PS entry from a catalogue of public services, perhaps allowing the final customizations at the moment of launching it up, and executing that PS.

In majority of cases, the execution will be carried out within a collaborative web environment very similar to the one of the PS co-production process. But in certain cases, for public services which are "deeply" crowdsourcing, with very high operation speed, data volume and scalability, both the SW INTERLINKER modules being used there and the execution platform should be much more scalable and higher performant than those of the PS co-production process.

In general, this type of process and corresponding SW platform is outside the scope of this project, except for the 3 pilot use cases where we will both design and demonstrate operation (i.e. execute) the newly co-produced public services. However, the public services in these 3 use cases have relatively modest scalability requirements, and thus, the service execution platform in those use cases will be more similar to the web portal of the co-production process.

3.3 Types of INTERLINKER SW modules

Classification of INTERLINKERs by various criteria is presented in detail in D3.1 document. Here we review types of INTERLINKERs from the point of view of their relation to the PS co-production platform.

3.3.1 Knowledge (Non-SW) INTERLINKERs

There are cases when building blocks (INTERLINKERs) of a public service are not necessarily software, for example, they may be knowledge INTERLINKERs in the form of a Wikipedia article, an Excel table or a best practice template for some processes or functions of a public service.

This case is relatively simple. Such types of INTERLINKERs do not include SW design and SW development in the module co-creation process, nor do they imply the execution of INTERLINKERs as software by service co-execution platform. For such cases, the focus of a SW system is on the Collaborative Environment and web portal functionalities, where users may further customize and re-use such INTERLINKERs.

3.3.2 INTERLINKERs as SW modules

In the modern digital era, the majority of INTERLINKERs should be SW modules which leverage mobile communications and automate most functions within a public service. Except for cases with extremely high communication rates, amount of users and volumes of data, it would be convenient to design such INTERLINKERs as SW modules pluggable into a common public service co-execution platform.

Another very similar version of INTERLINKERs being SW modules could be when an INTERLINKER is an external SW service, which is called within the workflow of the public service co-execution platform. When designing a SW system using the microservice architectural pattern, the difference between an INTERLINKER being an internal SW microservice or an external SW service is relatively little. The interaction between the platform and such a service is based on an INTERLINKER-specific API interface (which is in most cases RESTful or SOAP/WSDL).

3.3.3 Platform-type INTERLINKER modules

A special case of such SW modules are INTERLINKERs which on one side correspond to the core functionalities deeply integrated into PS co-production platform while on the other side still could be “packed” as building blocks in the catalogue of INTERLINKERs and could be re-used in other public services. Example of such type of INTERLINKERs are:

- single sign on (SSO) authentication module,
- user and team management module (which manage user and team data),
- asset management (which manage data files),
- logging and monitoring.

3.3.4 INTERLINKERs as a stand-alone SW

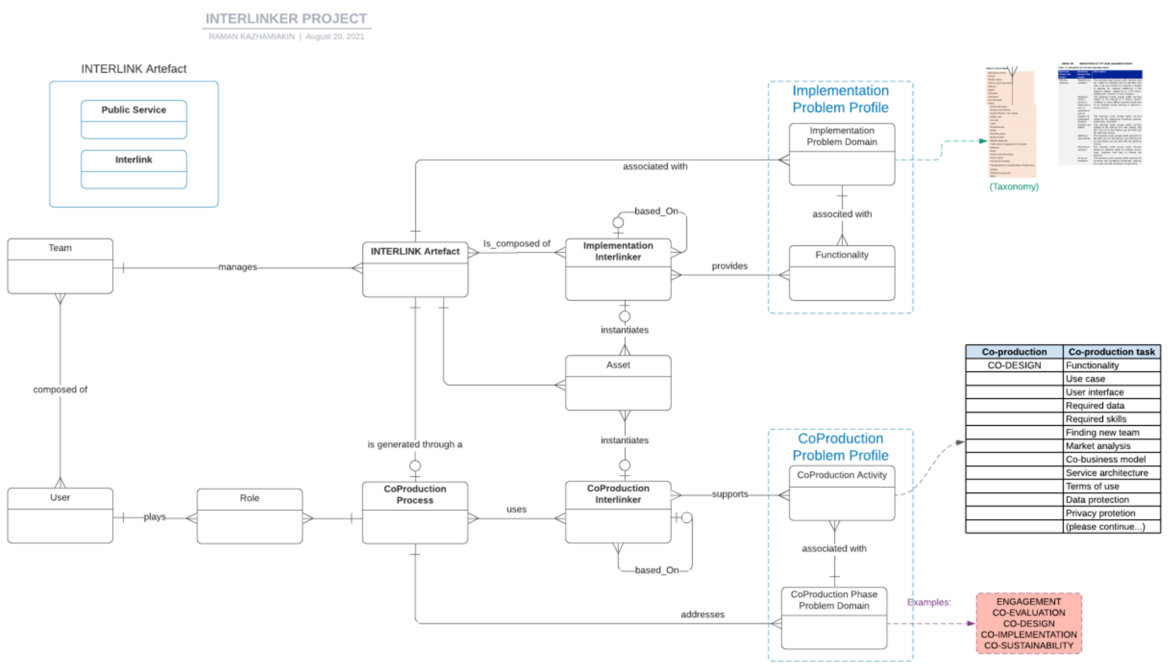
In cases of extreme complexity or extremely high scalability of a crowdsourcing type public service (e.g. car sharing), an INTERLINKER may be as big as the entire client+server software, running on its own, not as plugged into a common SW platform. In such cases, we still may include such INTERLINKER software in the catalogue of INTERLINKERs because they implement the core functionalities, which may be further wrapped and customized by a new public service. But we do not consider them to run in a common or generic public service co-execution platform. Instead, they may have their own backend servers to deploy and provide users with their own specific web or mobile client GUI (own frontend).

The same applies to SW modules which are standalone SW libraries, SDK, mobile applications, etc. Although those are not supposed to integrate into any public service co-execution platform, they still need to be in the catalogue of INTERLINKERs for possible customizations and re-use.

4 INTERLINK Data Model

4.1 Conceptual Data Model

The Conceptual DM has been widely discussed within the project partners, including the pilot user case teams, because this Conceptual DM focuses on representing the system functionality at a very high level, i.e. business level. The Conceptual DM is depicted as Figure 4 below.



Co-production	Co-production task
CO-DESIGN	Functionality
	Use case
	User interface
	Required data
	Required skills
	Finding new team
	Market analysis
	Co-business model
	Service architecture
	Terms of use
	Data protection
	Privacy protetion
	(please continue...)

Figure 4. INTERLINK Conceptual Data Model

The most important areas of the Conceptual DM are those describing the problem domains at the level of public services (top right part of the DM diagram) and of co-production process (bottom right part of the diagram).

The other key blocks of the DM are representing the core entities like

- INTERLINKERs;
- public services;
- co-production process and its phases;
- users;
- teams;
- assets.

4.2 Logical Data Model

The Logical Data Model is an intermediate level of DM, which is more detailed than the global Conceptual DM, but is still not completely detailed as Physical DM which corresponds to implementation at the data storage level. The Logical DM is depicted at Figure 5 below. It has been elaborated by the IT partners of the INTERLINK project.

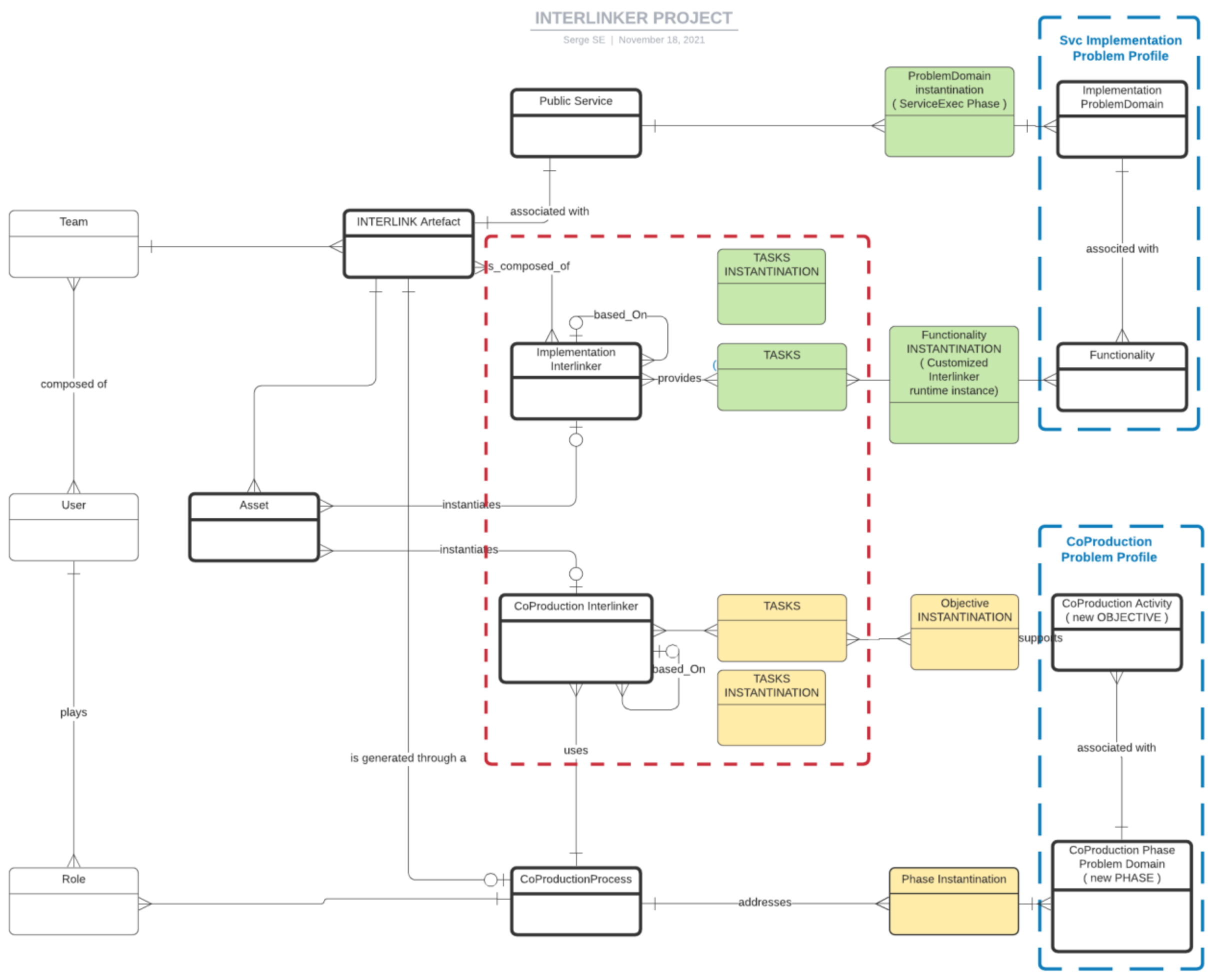


Figure 5. INTERLINK Logical Data Model

The key DM blocks of the Logical DM (the two problem domains, the INTERLINKERs, public services and co-production process) are the same as of the Conceptual DM, which is natural because the Logical DM is extending the Conceptual DM.

4.3 API to access key Data Model entities

4.3.1 Basic CRUD operations API

The key DM entities listed in the table below are common over the entire PS co-production (and could be also re-used in co-creation of INTERLINKERs and in PS co-execution processes) and are independent of particular SW components which might use them.

Table 3. Key Data Model entities according to the Logical DM diagram

DM Entity	Comments
USER	Users of the web portal
TEAM	Teams per PS co-prod. process
ASSET	Assets (resources, files) data
ROLE	User roles within teams
PublicService	Public services in svc catalogue
SvcImplementation_Phase	Phases within Svc co-execution
SvcImplementation_Functionality	Functionalities (Objectives) within Svc co-execution
SvcImplementation_Interlinker	INTERLINKERs of Svc execution type
SvcImplementation_Task	Tasks within Svc co-execution
CoProduction_Task	Tasks within PS co-production
CoProduction_Interlinker	INTERLINKERs of PS co-production
CoProduction_ProblemObjective	Objectives (Functionalities) within co-prod. problem domain
CoProduction_ProblemPhase	Phases within co-prod. problem domain
CoProduction_Process	PS co-prod. process
CoProduction_Schema	Configuration of process types & phases

Thus, the best practice would be to use the same common CRUD (create-read-update-delete) data persistency API for the key DM entities by all the SW components. So, if two or more different SW components want to execute CRUD operations over those key DM entities, they should use the same API, not re-implement it in their own way.

This can be satisfied automatically if layered architecture pattern is followed, having all the above (app logic, web presentation) layers "talk to" the data via additional data persistence layer providing single and common CRUD API.

Trying to keep the platform architecture and Data Model open for possible extensions with more platform type SW modules and thus, with more DM entities and entity relations, we provide here only the generic CRUD API specifications without listing all the DB table fields per DM entity. This allows us to use a generic CRUD API specification for an Entity_X, which can be applied to any particular DM entity out of the Table 4 above.

Table 4. Generic CRUD API for key DM entities

CRUD operation	HTTP call	API URL pattern	Parameters
Create (insert)	POST	$SRV/Entity_X	New Entity_X row as JSON data object
Read (select) one row by ID	GET	$SRV/Entity_X/$id	$id = Entity_X row id
Read (select) one row by field X value	GET	$SRV/Entity_X?FieldName=X& FieldValue=Y	FieldName is name of a field of Entity_X table, FieldValue is its value
Read (select) list of objects	GET	$SRV/Entity_X	May use additional parameters to paginate the output
Update one row by ID	PUT	$SRV/Entity_X/$id	$id = Entity_X row ID New Entity_X row as JSON data object
Delete one row by ID	DELETE	$SRV/Entity_X/$id	$id = Entity_X row ID

The Entity_X rows data flowing in and out via this REST HTTP API should be in the form of JSON objects.

4.3.2 Entity relations and other APIs

Besides the basic CRUD API for the key DM entities, there might be an API for functionality-specific operations on other DM entities and an API on entity-relations (e.g. lookups, joins, etc.).

For trivial entity relations either the basic GET by FieldName and FileValue can be used (for example, to find a team to which a user belongs to) or a lookup or filtering may be applied on a client (frontend) side once having the multi-rows data retrieved.

However, in more general cases, either for massive data or for more complex multi-field lookups, group by, loops and similar lookups, it might be more efficient to run such kind of lookups on the DB engine itself. In such cases, either additional views or complex lookup stored procedures may be created and used on the DB server. Such functionality-specific DB data should be retrieved via additional component-side API.

In some cases, when data looked up into such DB views need to be periodically updated, additional DB server-side scripts may be created and scheduled to run periodically. If such scripts are needed and should be implemented, the best way to control them is based on the common platform-wide process flow manager described in next Chapter.

5 PS co-production platform architecture

Throughout this Chapter, with “SW platform” we always refer to the platform of co-production of public services. Despite the many similarities between PS co-production process with processes of co-creation of INTERLINKERs and of co-execution of public services, strictly speaking, we do not include specifications of those SW platforms within this Chapter.

5.1 General design considerations

As mentioned in previous Chapters, some of the INTERLINKERs SW modules with key functionalities (authorization, etc.) will be deeply integrated in the PS co-production platform. In other words, in addition to SW modules of lower-layers like data storage and of general purpose services like authentication, the platform will also contain some SW modules which may be both used within the PS co-production platform and re-used later as INTERLINKERS for particular public services co-produced with the Collaborative Environment portal.

Because of possible licenses and GDPR matters, the preferred scenario project-wide is to use open-source software which could be either deployed and used locally as-is, or which could be hosted (deployed) locally and interfaced with additional custom developed adapters. We consider such types of SW as local components.

In parallel with the development and integration of the software components, in collaboration with other WPs of the project, the issue of licenses will also be addressed with a dual purpose:

- to identify solutions with compatible licenses that can be integrated into a single service,
- to apply the correct licenses to the project products (software, knowledge interlinker) balancing the need to release the project results by facilitating free reuse, but also commercial exploitation.

However there might be functionalities for which there is no open-source SW but there is a free SW service provided by some external IT company, like Google Doc, so we could use it as long as there is compliance with the GDPR requirements. In such cases, our SW platform could use those external services as “black-boxes”.

5.1.1 Layered architecture diagram

In terms of the layered SW architecture pattern, we may define the following major layers:

- data storage
- business logic (application logic) layer
- and presentation layer (web frontend)

as depicted in Figure 6 below.

Additional layers may exist for particular functionality areas between the above major layers, for example:

- data persistence API layer

- API to the presentation layer (frontend)
- and additional API or adapters for cases when external SW services (SaaS, Software as a Service) could be used, for example, google document, etc.

The layered architecture is shown in the Figure 6 below. The platform part is in the bottom left area, external SW services are illustrated in grey colour on the right.

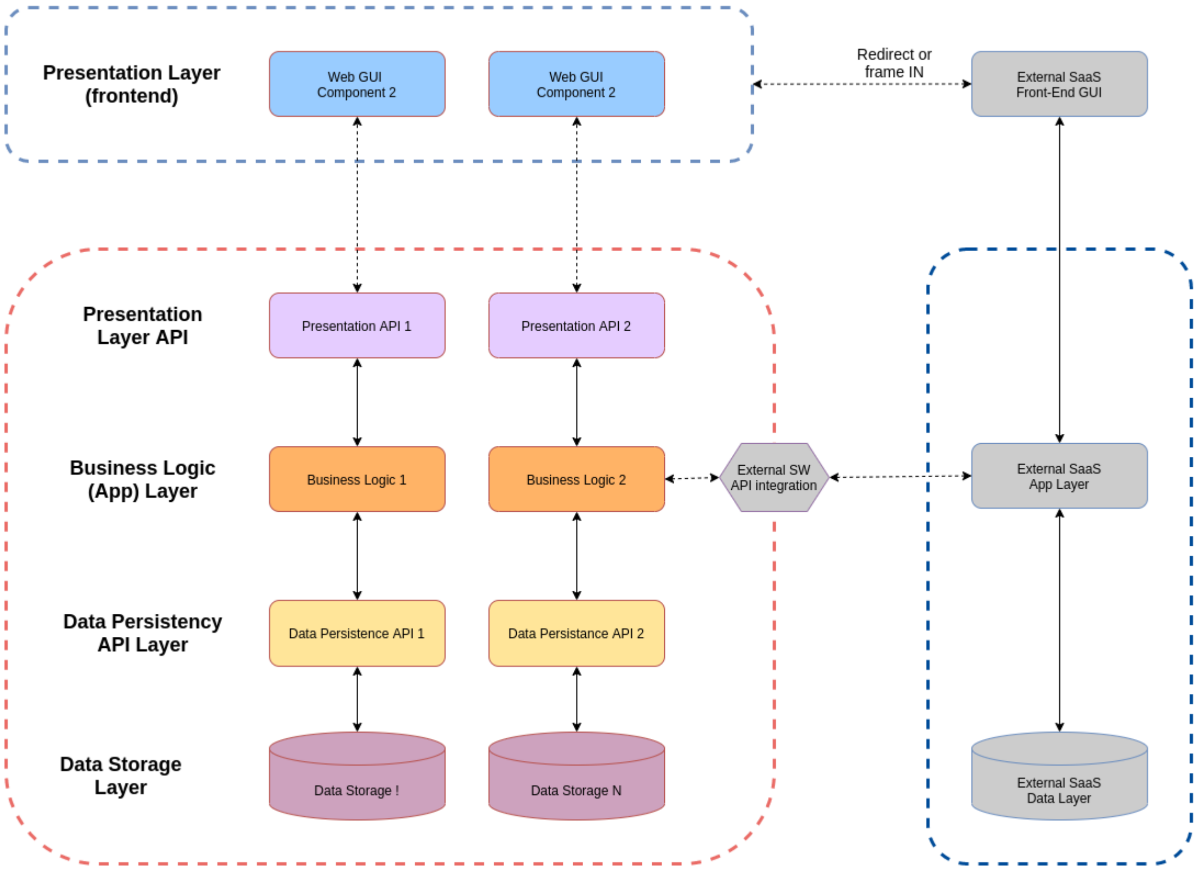


Figure 6. Layered architecture for PS co-production system

5.1.2 Data storage layer

To keep the platform architecture as open as possible, we formally allow any type of data storage, assuming the suitable ones for this type of software. For example, either or all of the following may be used:

- **MySQL (MariaDB) / PostgreSQL** (most suitable for fixed-schema data like configurational or transactional),
- **MongoDB** (with advantages of easy scalability and schema change freedom),
- **Redis** (for fast in-memory data operations),
- open-source version of **S3** from **Minio** (for file storage).

However, use of particular technologies and DB engines should be unified when possible.

For interoperability of backend SW components themselves, common file storage with low-level access within the platform could be used if needed, either via a shared disk mounted over several servers, of a Hadoop-like filesystem (when deal with very massive data, e.g. BigData). Although we do not see explicit necessity in this currently, such storage may be kept in mind when developing platform-wide logging and resource (asset) management software.

5.1.3 Business Logic (App) layer

The business logic (app level) SW components may follow:

- either micro-kernel/plugin architecture pattern (for example for the process workflow manager, user manager and perhaps for auth service),
- or microservices architecture pattern for the cases when these SW modules should be also re-used in PS co-execution processes (for example, payment or incident management, etc.)

5.1.4 Interface to the web portal frontend

The API interfaces between the backend and the frontend (the presentation layer) are mostly RESTful HTTP APIs. Besides the CRUD API for the basic Data Model entities described in the previous Chapter, they may use additional frontend specific data tables with corresponding CRUD and functionality-specific APIs.

5.1.5 Interface to external SW services

API interfaces for external SW services may be either REST HTTP or SOAP WSDL, depending on the protocol which the external SW service providers support.

Log collection services may be implemented either via REST HTTP protocol or at lower level, using for example Linux log daemon (syslog) communicating to other Linux hosts via UDP protocol.

5.2 Frontend-to-platform integrations

5.2.1 Common web portal frontend

The INTERLINK project will be delivered to end-users in the form of a common web portal representing all the functionalities of collaborative processes and interfacing to the users all the used INTERLINKER SW modules together.

Either iFrame technology or redirect between microservice frontend pages should be organized in such a way that users keep the overall common navigation and follow the co-production process workflow correctly, without necessity to manually jump between separate web sites.

5.2.2 Web portal integration scenarios

The Web portal of the co-production environment is designed in a way to facilitate the integration of different tools and instruments, even when provided and implemented externally (e.g., existing OSS, SaaS components, etc). For what concerns the components that constitute the core functionality of the co-production environment, the corresponding modules are integrated directly as a part of the same front-end implementation. This includes the management of co-production process instances, the corresponding assets, team management, etc.

To make the co-production open for the integration of additional functional modules and components, relevant for the co-production process activities and tasks, we adopt the micro-frontend architecture. More specifically, the following assumptions are due:

- Even when deployed and managed separately, it is necessary to allow for **seamless user experience for what concerns authentication**. Therefore, Single Sign-On functionality should be implemented between the integrated components and the core Web portal.
- It is possible to integrate the **UI of the component directly within the portal** to facilitate the user experience. It is preferable to adhere to the same UI design model, but not strictly required (otherwise it would require to significantly revise the implementation of the integrated component, which may be costly and reduce the extensibility). In particular, to facilitate and speed up the integration, such implementation should allow integrating the UI as **iframe**.
- No front-end interactions between the components are foreseen other than changing the iframe address. This drastically facilitates the implementation of the front-end integration and makes it easy to add existing components.

It is important to note that when these assumptions may not be satisfied, the integration of the Co-production Interlinkers within the Web portal may still be done by simply linking the corresponding Web applications.

To achieve this model and streamline the implementation, the following approach is used:

- Common authentication and SSO mechanisms.
- Common model of the external artefacts (assets) that are integrated within the portal.

Common authentication and SSO mechanisms. Depending on the way the integrated component is implemented or re-used the following two scenarios may be considered:

- Make the authentication of each component independent and rely on the SSO of the authentication service. This is performed as follows:

    1. The Web portal performs authentication with the SSO service.

    2. The component integrated via iframe requires its own authentication and redirects to the SSO service. Since the authentication with the SSO service has already been done, the process is completed and the component is redirected to the original endpoint in a transparent manner.

Although major flexibility in the authentication implementation is allowed, there are some requirements that should be satisfied, including the necessity to correctly manage and maintain redirects upon authentication, sharing the same user permissions, and preferring redirection to the popup-based flow.

- Use a shared cookie for all the integrated components. When deployed under the same domain, the components may have access to shared cookie credentials (SameSite cookie[1]) that encodes the authentication information. Such approach is most effective in terms of the integration, but requires the integrated component to support such an authentication mechanism (possibly, in addition to the already implemented ones) and makes it impossible to integrate the components provided externally (e.g., as a SaaS hosted on a different domain).

Common model of the external artefacts. To facilitate the management of the new components within the co-production Web portal, it is necessary for the component to adhere to the “asset” model of the co-production process. The general idea behind the asset is to represent the relevant entry point to the functionality of the Interlinker when used in the co-production process. The assets may represent, e.g., a file produced and shared between the co-production team, a shared diagram, a discussion in a forum Interlinker, the idea discussion and voting thread in the Idea Management INTERLINKER, project planning workspace in the Project Management Interlinker, etc. When used within the co-production environment, the assets will have a common approach to creation, management, and visualization.

More specifically, the integrated Interlinker should provide the API and endpoints to

- Create a new asset of a particular type. This should create the corresponding artefact within the Interlinker component and provide its reference ID to be used further.
- Get asset information given its ID.
- Update and/or delete the assets given its ID.
- Get access to the UI of the artefact given its ID. Such an endpoint then will be used within the co-production environment to integrate the UI of the artefact as an iFrame.

Implementation of such a model will make the co-production environment open for new components and functionalities in different scenarios and contexts without necessarily losing the generality of the original software implementation.

5.2.3 Web portal integration to platform

As the co-production platform focuses on the public service co-production process flow, all the frontend navigations should be transparent and compatible with the sequence of the co-production process phases and with INTERLINKER specific sub-processes. While these requirements are satisfied, further navigation within co-production phases and within local INTERLINKER sub-processes may be driven by the frontend GUI visualization and functional needs.

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite

In the sense of the backend-frontend integrations this means that frontend components should use a common workflow manager when dealing with co-production phases, as described in details in section 5.3.4 below.

5.3 Platform SW modules specification

5.3.1 Platform SW modules

Since the PS co-production process is highly collaborative and the entire project has crowdsourcing features, we would not limit the PS co-production platform with a fixed set of such deeply integrated SW modules (platform-type INTERLINKERs). We would rather define the most obvious one and leave the list of such platform components open. Thus, more platform-type INTERLINKERs could be easily added or replaced with better ones (for example, a payment SW module) in later SW releases, following the same global platform architecture defined here.

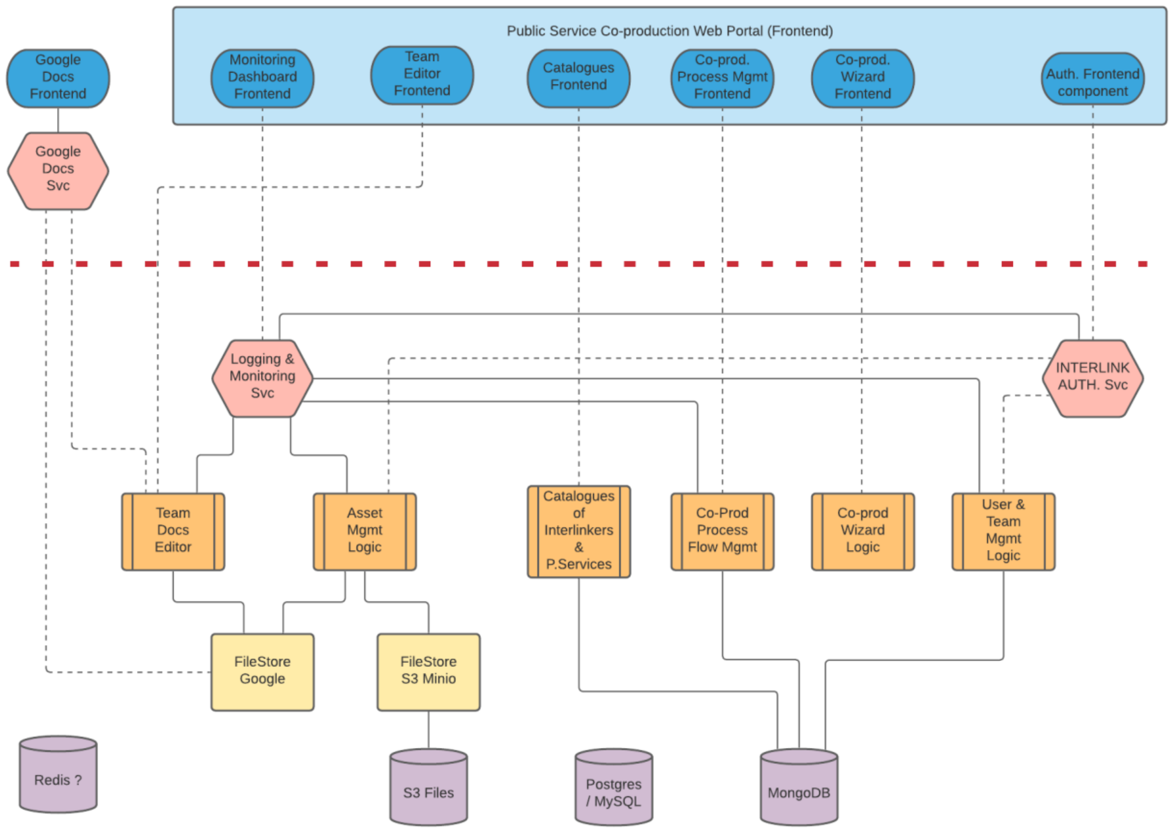


Figure 7. Initial architecture for PS co-production platform

The initial version of the SW component diagram for the PS co-production platform is shown in Figure 7. It contains the following key SW modules included:

- authentication service

- user and team management
- asset management
- workflow management
- logging and monitoring services.

More SW modules with platform-type functionality (like payment, etc.) could be added to this platform following the same architecture pattern.

5.3.2 Authentication and Authorization Control INTERLINKER

To support the functionality of the user authentication and access control across the platform components, we adopt the Authentication and Authorization Control (AAC) Interlinker. This Interlinker is a Free Open Source Software component implemented by FBK, extended and adopted in order to face the specific needs of the INTERLINK project. More specifically, the component has been extended in order to seamlessly integrate with the CEF eID[2] specification and support the corresponding cross-border authentication scenarios.

5.3.2.1 Functionality overview

The SSO Authentication service implements the Identity and Access Management functionality. That is, the core functionality of the SSO Authentication service module is twofold.

First, it is used to federate and make transparent for the users and services the involved authentication procedure with external and internal Identity Providers (e.g., social networks like Facebook or Google, direct user registration, national providers or even cross-border eIDAS authentication). The functionality allows for configuring different ways to register and manage users, such as:

- Direct user registration with email / password authentication. The user registration is enabled and open to the platform participants.
- Login with social accounts, such as Google, Facebook, or through any other OAuth2.0 / OpenID Connect compatible provider, such as Github, Azure AD, etc.
- Login with any SAML-compatible[3] SSO Identity Providers.
- Login with a pilot-specific Identity Provider available at the local or national level.

Currently AAC supports out of the box the Italian national authentication systems (SPID[4] and CIE[5]) and integration with eIDAS profile-based systems. In other cases a specific implementation may be integrated in order to integrate some custom authentication mechanism.

Second, it is used to expose the OAuth2.0[6] / OpenID[7] Connect protocols towards the components of the platform and the integrated services for the purpose of the user and service access control. In this way, the integration of the user authentication is completely externalized and is based on a standard, state-of-art security mechanism. More specifically the integrated components and services may use different authentication flows (as defined by OAuth2.0 protocol) in order to identify and authenticate the end user and obtain the information about its attributes. Furthermore, the AAC interlinker also supports machine-to-machine interaction based on the Client Credentials authentication defined by OAuth2.0 protocol.

[2] https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/eID
[3] https://www.oasis-open.org/standards/#samlv2.0
[4] https://www.spid.gov.it/
[5] https://www.cartaidentita.interno.gov.it/en/citizens/log-in-with-cie/
[6] https://oauth.net/2/
[7] https://openid.net/connect/

At high level, all other SW modules should authorize via the ACC INTERLINKER according to the standard OAuth2 flow depicted in Figure 8 below.

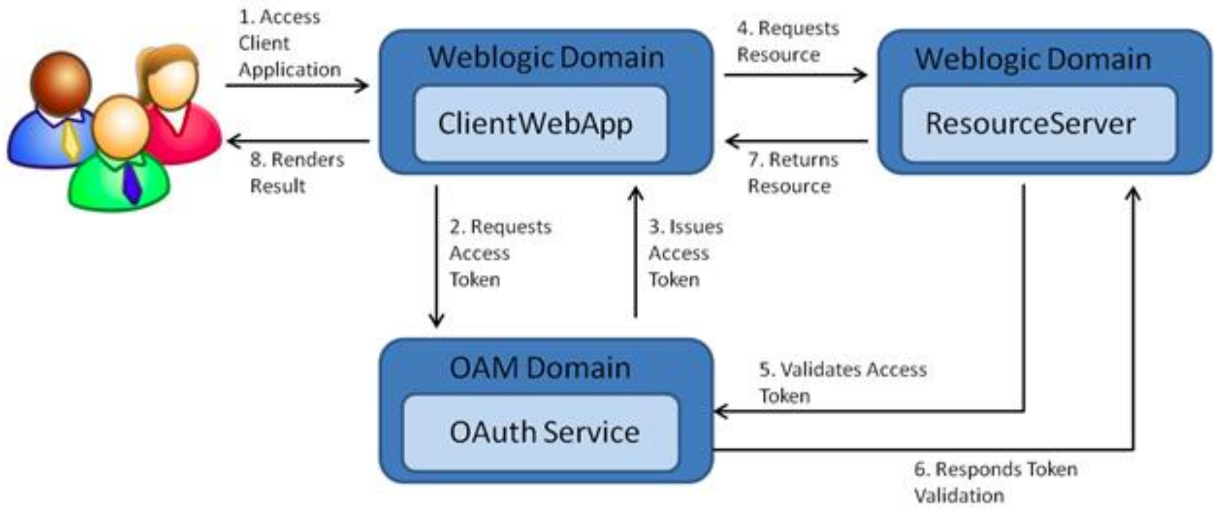


Figure 8. Standard authorization flow with OAuth2

In order to identify and authenticate the user, the functional flow with AAC may be defined as follow:

- Configure: the Client Application in AAC to represent the integrated party. This amounts to defining the used flows, the required permissions and user attributes, the type of authentication token, etc.
- Implement: the appropriate OAuth2.0 flow using the standard OpenID Connect / OAuth2.0 libraries (e.g., Authorization Code Flow for Web applications, PKCE for SPAs or mobile apps, etc).
- Authenticate: the users and obtain the necessary user attributes (e.g., name, surname, email) directly from the token (if JWT token is used) or from the corresponding user information API as required by the protocol.

Besides this core functionality, AAC exposes further customization capabilities that allow for

- Define custom attribute sets and extract them from the information provided by the identity providers or managed and stored explicitly by AAC itself. This is necessary to implement advanced authorization policies, such as Identity-Based Access Control (IBAC), Role-Based Access Control (RBAC), or Attribute-Based Access Control.
- Define custom permissions and access control rules associated with them. Custom permissions (scopes) are necessary to isolate roles and capabilities of different services / components of the platform.

In order to guarantee the seamless user experience when multiple components require user authentication, the AAC Interlinker implements the Single Sign-On (SSO) functionality. The same user is authenticated only once if the open session for this user exists. When the user is logged out, it is possible to implement also Single Logout (SLO).

It is important to note that the AAC is designed to guarantee GDPR compliance. Specifically, it is a multi-tenant platform so that different organizations sharing the same instance do not necessarily share the users; may treat them differently and in a completely isolated manner. Second, thanks to the flexible and extended implementation of the authorization policies of the OAuth2.0 flows, it is possible to define custom permissions and require explicit user consent when the application tries to access such information.

More details about the AAC implementation and functionality may be found here: https://scc-digitalhub.github.io/AAC/

5.3.2.2 Technologies used

The implementation of the AAC Interlinker relies on a set of standards including:

- OAuth2.0 protocol: exposed for the authentication and authorization flows as well as for the federation with external identity providers.
- OpenID Connect protocol: exposed for the user identity information on top of the OAuth2.0 protocol as well as for the federation with external identity providers.
- SAML protocol: used for the federation with external identity providers.
- JWT: used for token representation and validation.
- SPID and CIE: SAML dialects and extensions to integrate with Italian identity providers.
- eIDAS: SAML dialects and extensions to integrate with the eID CEF building block.

The implementation stack relies on a set of Java-based technologies that enable Cloud-Native Application model and deployment. Specifically:

- Java 11+
- Java Spring Framework modules (Spring Core, Spring JPA, Spring Security, Spring Boot, …).

For what concerns data persistence, the component uses any JPA-compatible RDBMS, such as MySQL or PostgreSQL.

To support Cloud deployment, the Docker container specification is provided together with the recipes for the deployment in orchestrated environments, in particular Kubernetes.

5.3.2.3 Design and platform Integration specifications

The architecture of the AAC INTERLINKER module is shown in Figure 9. It devised in a set of modules that include:

- Inbound Authentication module: the implementation of OpenID Connect and OAuth2.0 protocols exposed for integration.
- Authentication Framework: the core part of the component responsible for the various functionality related to the protocol, user management, authorization, etc.

- Local Authenticator: used to manage the authentication of internally registered users (through username/password authentication).
- Federated Authenticator: extensions to deal with external Identity Providers via a set of protocol connectors (OAuth2.0, OpenID connect, SAML, etc.).
- Management environment and UI: the set of components and the UI console to configure and manage the different integrations of the component (external identity providers, user and role management, authorization policies, attribute management, etc).
- Protocol API: the standard OpenID Connect and OAuth2.0 APIs exposed to support the authentication and user identity flows.
- Management API: the set of APIs that allow for performing the management operations programmatically, via REST API calls.

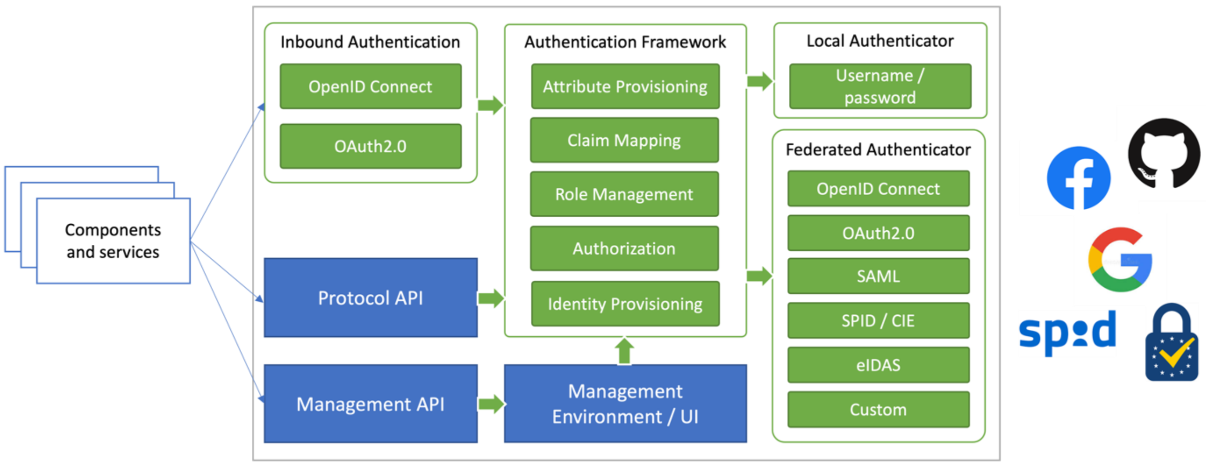


Figure 9. AAC module architecture

The integration of the platform components with AAC Interlinker is achieved therefore via the following scenarios:

- User Authentication: following OAuth2.0 specification the components trigger the corresponding authentication flow in order to authenticate and identify the users interacting with the components and to obtain the corresponding tokens.
- Token Validation: use the OAuth2.0 and OpenID Connect endpoints to validate the access and identity tokens and to extract necessary information.
- User and Role Management: use management APIs for the purpose of the user data operations, such as user attribute management, role management, etc.

5.3.3 User & team management

The goal of the User & Team Management component is to organize the members of the co-production teams within the collaboration environment according to their roles. To accomplish this, the component relies on the functionality of the SSO Authentication service implemented as AAC Interlinker.

5.3.3.1 Functionality overview

The key functionality of this component amounts to

- Creating and managing the co-production process teams. Each co-production process instance may deal with different participants having different roles in the project – from managing all the aspects and artefacts of the instance to merely commenting or suggesting something without access to all the data.
- Inviting and managing the members of the team with their corresponding roles. Such functionality may be explicit (i.e., performed by the project owner through the corresponding UI section in the collaboration environment) or implicit (i.e., when a user adheres to some functionality, such as following the project).

With respect to the operations the team members may perform within the project, we therefore can distinguish

- **Project Owners**. These members have full access to the instance data, activities, and co-production process operations.
- **Project Members**. The stakeholders making the active part of the process, such as decisions and voting, commenting, managing assets of other Interlinkers. Instead, they cannot manage the project members.
- **Project Community Members**. These are the users of a wider project network, which are able to comment project updates, receive notifications about the project evolution, etc.

Please note that the User & Team Management component will not do the user provisioning and user data management. For this purpose, the AAC Interlinker will be engaged, thus separating the problem of user registration, federated authentication, and user data management from the participation of the user in the collaboration environment.

5.3.3.2 Technologies used

The User & Team Management component makes part of the core co-production set of microservices and is implemented with the same technological stack:

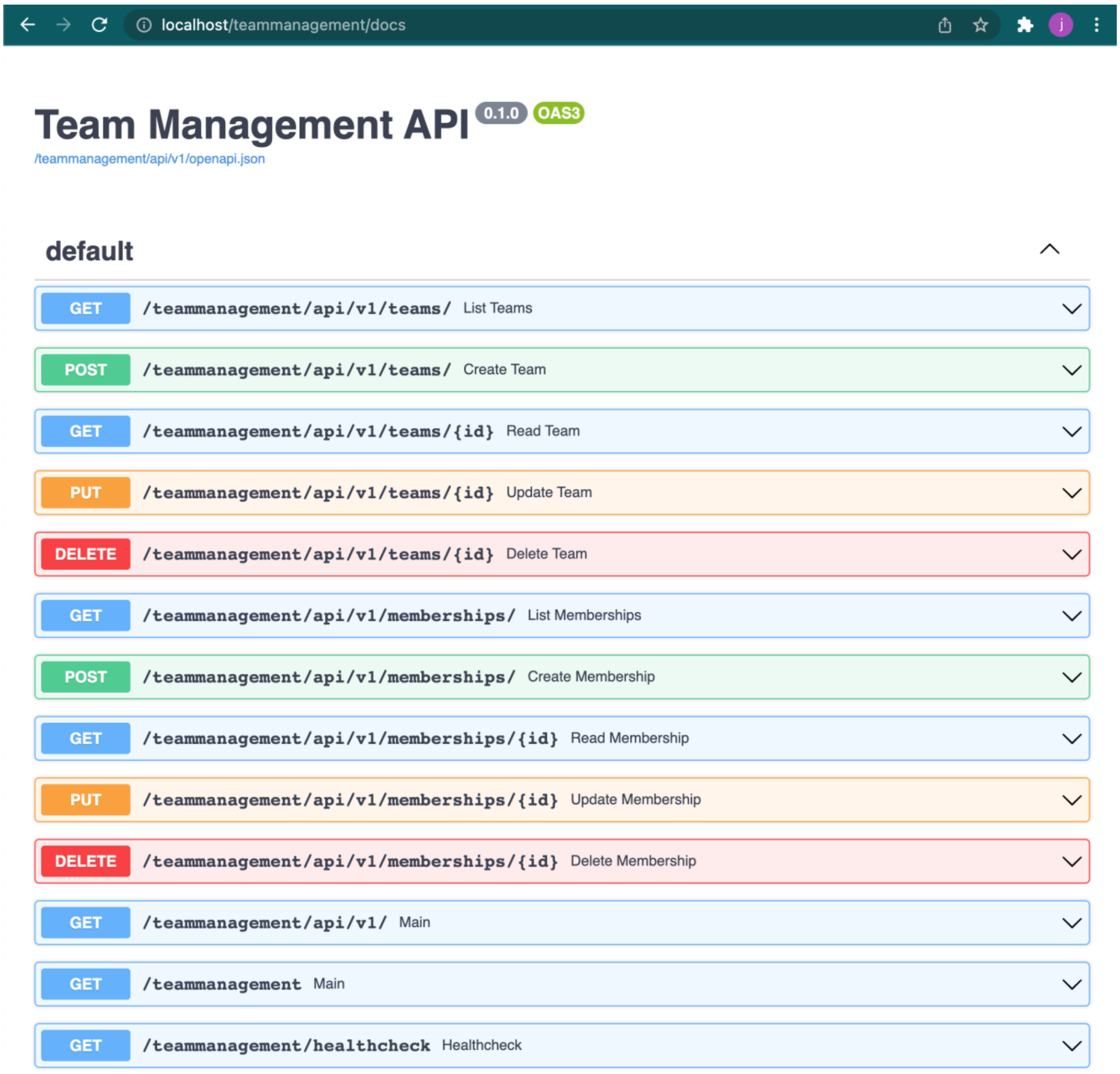
- Python (3.8) and Gunicorn
- SQL Alchemy framework for the data management
- PostgreSQL for persistence.

To support Cloud deployment, the Docker container specification is provided.

5.3.3.3 Design and platform Integration specifications

Following the approach used within the core co-production set of components, the User & Team management component is implemented as a backend microservice exposing the APIs for the team and membership management presented in Figure 10 below.

The API is made available directly to the co-production environment frontend and other components, where the corresponding operations are triggered upon explicit user actions or in relation with the process operations (e.g., follow/unfollow project, create new project, etc).


localhost/teammanagement/docs

Team Management API 0.1.0 OAS3

/teammanagement/api/v1/openapi.json

default

GET /teammanagement/api/v1/teams/ List Teams

POST /teammanagement/api/v1/teams/ Create Team

GET /teammanagement/api/v1/teams/{id} Read Team

PUT /teammanagement/api/v1/teams/{id} Update Team

DELETE /teammanagement/api/v1/teams/{id} Delete Team

GET /teammanagement/api/v1/memberships/ List Memberships

POST /teammanagement/api/v1/memberships/ Create Membership

GET /teammanagement/api/v1/memberships/{id} Read Membership

PUT /teammanagement/api/v1/memberships/{id} Update Membership

DELETE /teammanagement/api/v1/memberships/{id} Delete Membership

GET /teammanagement/api/v1/ Main

GET /teammanagement Main

GET /teammanagement/healthcheck Healthcheck

Figure 10. Team management API

5.3.4 Process workflow manager

5.3.4.1 Functionality overview

Despite this document is dedicated mostly to the SW platform for one collaborative process (co-production of public services) among the three similar processes possible within the scope of the INTERLINK project, we aim to elaborate an architecture well extendible for the two other types of collaborative processes (namely, to co-creation of INTERLINKER SW modules and to co-execution of public services).

As mentioned in the previous Chapter, each of these three collaborative processes are split into phases which are supposed to run sequentially one after another (i.e. neither in parallel, nor randomly). Obviously, this does not mean to restrict the web portal users from going back to a previous phase for example to add a new team member, this only means that the set of phases itself is an ordered one, so there is a “normal” sequence, and if a user needs to add something to a previous phase, it should be done in a proper way. For example, a new team member added at a later phase of a PS co-production process, for example at SW design stage, should not break the decision e.g. on the service feasibility and set of stakeholders, which had been discussed in previous co-production phases long ago and are already decided and took into account in the ongoing SW design works.

In order to build the SW platform on a solid while flexible foundation, the phases and key processes of a PS co-production should be controlled in a clear and centralized way, not just as free as a web page could let users to navigate from one to another. A good approach here is to establish a set of process-flow related policies (rules) and to have the SW code related to the process flow in a single backend SW module, which we call a workflow (process) manager.

Remark that we obviously do not need to design the SW platform too complex w.r.t. how it could be, neither to waste the project manpower onto SW code which would be more complex than just enough. For example, within the scope of the PS co-production process there is no need for automation of data batch processing, in contrast to some possible public service co-execution cases where it may be really needed (e.g. for a car sharing service). So, for the PS co-production, we do not need a workflow engine like Apache Airflow or similar. Still, we need to use best practices for the process management functionalities within the PS co-production scope.

5.3.4.2 Technologies used

One of the best approaches here is the methodology of Finite State Machinery (FSM). Allowing the agility in particular SW implementation, here we specify the key FSM methodology principles which we would want the SW implementation to follow. Here they are:

- all the code dealing with start, end, identification of current phase of PS co-production process, etc., should be gathered in a single SW module or set of SW functions, which is easy to find and clear to separate from other type functionalities of the web portal frontend (for example, visualization of numerous GUI elements, etc.);

- the SW code dealing with processes and workflow should be "blind", which means, all the phases, stages, states, etc should be used not as a hardcoded string values, but as elements of a configuration list;
- besides the configurable list of states (phases, etc.), the possible transitions between them (from phase A to phase C and e.g. not to phase D) should be also configurable by a list linked with the list of phases (states, etc.);
- there might be only one currently ongoing process (phase) within a PS co-production, not several at the same moment (thus, for example to add a new team member on a later phase, one need first to "pause" the ongoing phase, for example, to persistify its data, roll back the phases back to the one where a new team member could be added, add him or her to a team, and then "resume" the latest ongoing phase again);
- such SW code ("process management engine") dealing with processes should be generic and extensible to other objects/processes, not be restricted only to the PS co-production case;
- all the other (app logic, frontend, etc.) SW components should make use of such process manager to get or update process states via a common API.

Such FSM methodology has several benefits if used, for example:

- the SW code implementation becomes very robust with minimal probability of SW bugs;
- improper use of navigation within the web portal will be automatically blocked by such a process flow engine;
- it allows good flexibility because list of states (phases) and transactions between them are fully configurable, and can be re-defined per each server launch without touching the SW code;
- it allows to map transaction list to list of functions to call which correspond to these transactions, and have extra flexibility in such binding;
- it still does not imply any restrictions on the web navigation when it is not related to the process management, for example, within the same PS co-production phase, there may be multiple lower-level processes running according to different policies (for example, in parallel).

5.3.4.3 Design and platform Integration specifications

An example functionality of such an FSM mechanism can be illustrated in a simplified sequence diagram which shows possible API calls between frontend, backend app logic and data storage layers within the layered architecture concept. It is depicted in Figure 11.

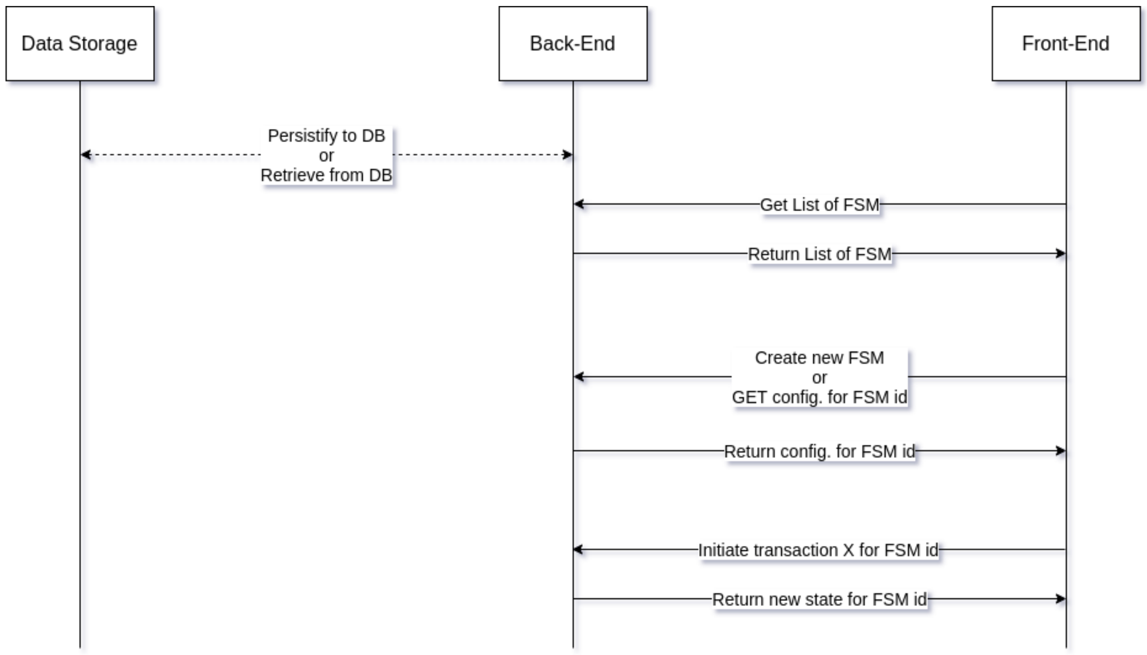
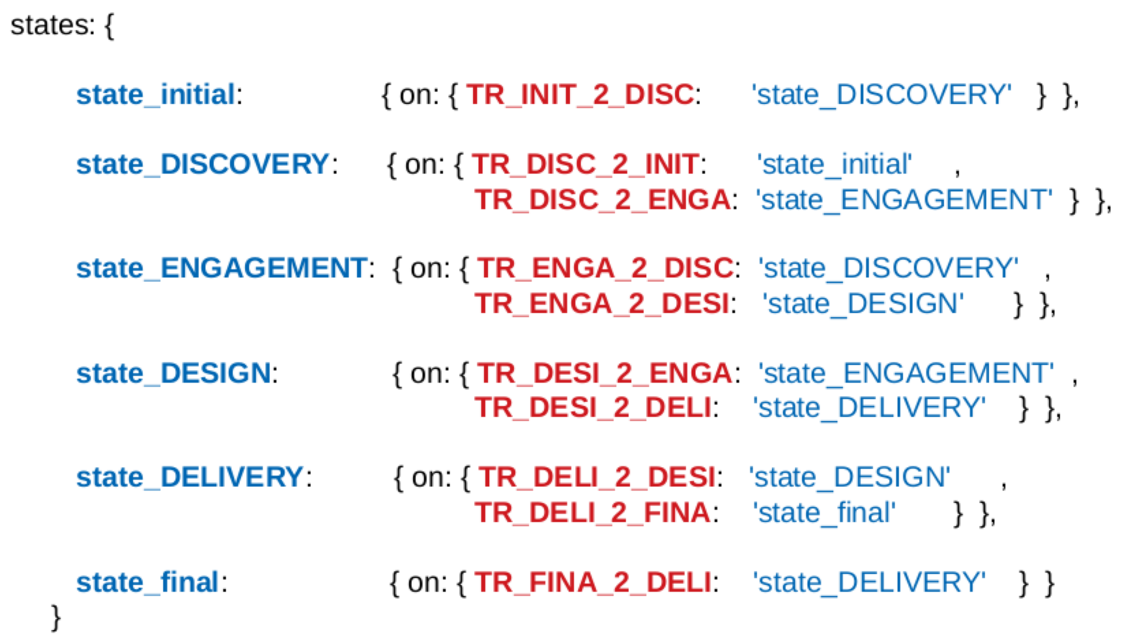

Data Storage
Back-End
Front-End
Persistify to DB or Retrieve from DB
Get List of FSM
Return List of FSM
Create new FSM or GET config. for FSM id
Return config. for FSM id
Initiate transaction X for FSM id
Return new state for FSM id

Figure 11. Sequence diagram for FSM API calls

Configuration of process states (phases) and of transactions allowed between them can be illustrated in the code sketch shown in Figure 12..



```
states: {

        state_initial:          { on: { TR_INIT_2_DISC:    'state_DISCOVERY'   } },

        state_DISCOVERY:        { on: { TR_DISC_2_INIT:    'state_initial'     ,
                                        TR_DISC_2_ENGA:    'state_ENGAGEMENT'  } },

        state_ENGAGEMENT:       { on: { TR_ENGA_2_DISC:    'state_DISCOVERY'   ,
                                        TR_ENGA_2_DESI:    'state_DESIGN'      } },

        state_DESIGN:           { on: { TR_DESI_2_ENGA:    'state_ENGAGEMENT'  ,
                                        TR_DESI_2_DELI:    'state_DELIVERY'    } },

        state_DELIVERY:         { on: { TR_DELI_2_DESI:    'state_DESIGN'      ,
                                        TR_DELI_2_FINA:    'state_final'       } },

        state_final:            { on: { TR_FINA_2_DELI:    'state_DELIVERY'    } }
    }
```

Figure 12. Example of infrastructure logging format specification

Within such FSM approach, the same FSM process management engine may act for any of the Data Model entities (USER, TEAM, CoProdProcess, PubService, etc.), by creating and manipulating a virtual FSM object per each DM entity (thus, managing states of USER, of

TEAM, of CoProdProcess, etc.). An example of such FSM API can be presented in the following table.

Table 5. Example API for process flow management

CRUD operation	HTTP call	API URL pattern	Comments
Create (insert)	POST	$SRV/fsm/Entity_X	Create new FSM for Entity_X; returns ID of the new FSM obj
Read (select) list of FSM	GET	$SRV/fsm	Returns list of existing FSM objects for data entities
Read (select) FSM for row ID	GET	$SRV/fsm/$id	Returns FSM configuration (list of states & transitions) and current state for FSM object ID
Update FSM state for ID	PUT	$SRV/fsm/$id/$transition	Execute state $transition for FSM with given ID
Delete FSM by ID	DELETE	$SRV/fsm/$id	Delete FSM object by ID

5.3.5 Resource / asset management

5.3.5.1 Functionality overview

Within the scope of the public service co-production process, we assume resources or assets to be standalone files of any type stored / used by any of the platform type INTERLINKER in the process of work within the Collaborative Environment. Such files can be for example:

- module-specific binary data of any kind, packed/unpacked by particular SW module;
- video, audio and image files;
- office documents, pdf files or excel tables;
- just text, JSON, YAML or XML files containing module-specific or general textual data.

The main function of the resource / asset management module is to provide other SW modules with basic CRUD-like operations over such data at file level, for example:

- create / receive data file of desired type,
- store data file in desired data storage engine,
- search / read / retrieve desired data file from given storage engine by some criteria (by ID, date, user, etc.) and pass it to a frontend module for visualization;
- update desired data file with new data file data;
- delete desired data file in given data storage engine;
- copy (or move) desired data file within storage engine(s).

The access control will be based on the common SSO Authentication module described in section 5.3.2 of this document, so only authorized (logged in) users will be able to read and write data. Besides that, the read/write should be mapped to the user/team management and roles metadata, so only users belonging to a given team / project should have access to read/write to team's / project's data area.

5.3.5.2 Technologies used

There might be several alternative data storage engines used for these purposes. Particular choice is based on types of data, amount of data and typical frequency and kind of use. Within the scope of the co-production platform we consider to use the following engines:

- a MongoDB cluster: this engine is best to use for JSON data and binary files (images, audio, video, etc); the most suitable use cases are when a SW module runtime functionality is based on frequent operations over JSON and/or binary data, for example, in chats, forums, discussion boards, voting module, etc;
- an S3 storage cluster based on Minio SW: this is an open-source version of the SSS file storage similar to the one provided by Amazon Web Services; the most suitable use cases for this engine are operation of document and media files, for example, collaborative document editor, automatic sending / receiving of data files, e.g. as email attachments, etc.

5.3.5.3 Design and platform Integration specifications

The resource management module can be composed of two subcomponents:

- implementing interface to data storage engine(s) to read/write data files from/to data storage;
- implementing API to frontend and other SW components to pass commands and data files.

As all the other SW modules, the asset management module should authorize each operation with the common SSO Auth module, according to the authorization pattern used (pass JWT token per each request or redirecting HTTP calls for JWT validation, etc). For the SW development phase, a simplified local JWT validation can be used, and later integration with the real SSO Auth module should be made.

Usually the data itself is accompanied with additional metadata about the data file like for example:

- date/time of data creation,
- date/time of last modification,
- user_id of user who has created or modified data,
- type of data,
- name of SW module doing API call for this data,
- and any additional module specific details which a SW module wants to associate with the data file passed to/from the storage layer.

To pass both the file data itself and the metadata about that data file, a HTTP call can be used with metadata passed as API call parameters and the data file attached as a binary payload.

In case of a MongoDB storage engine, the metadata can be stored as a JSON document entry within the same DB as the data file. For the case of S3 data storage, the data file should be placed in an S3 bulk directory pre-configured for that type of data, for that SW module, etc., and the metadata could be stored either as a JSON file in the same S3 directory next to the primary data file, or in a separate SQL or JSON-based DB (e.g. MongoDB) with reference to the full path of the data file in the S3 storage.

The table below summarizes example API references for the basic CRUD-like HTTP calls to operate the data files.

Table 6. Example resource management API specifications

CRUD operation	HTTP call	API URL pattern	API call parameters
Create (insert)	POST	$SRV/files	user_id, team_id, role_id, date/time, file metadata
Read (select) list of files	GET	$SRV/files	user_id, team_id, role_id
Read (select) file for ID	GET	$SRV/files/$id	user_id, team_id, role_id, file reference ID or file path
Update file for ID	PUT	$SRV/files/$id	user_id, team_id, role_id, file reference ID or file path
Delete file by ID	DELETE	$SRV/files/$id	user_id, team_id, role_id, file reference ID or file path

5.3.6 Logging & monitoring service

5.3.6.1 Functionality overview

Within the scope of the PS co-production platform, logging and monitoring functionality will be used for the following two goals:

- to log and to monitor software operations and platform infrastructure at runtime in order to ensure correct operations of the SW system; in this case we speak about infrastructure logging & monitoring;
- to collect KPIs from application level components of the Collaborative Environment in order to monitor user activities within the PS co-production process; in this case we speak about application/business level monitoring.

It is supposed to use the same logging and monitoring SW module for both levels of monitoring. The infrastructure logging and monitoring will be based on collecting logs from all Docker containers deployed with the SW system, then those logs will be stored in a common logging area of the platform data storage, where it could be post-processed and visualized with a basic dashboard.

In order to log and monitor the application level KPIs, each of the SW modules (both platform core and platform-type INTERLINKERs), should implement additional logging functionality in their code where they will log their functionality specific data to the common platform logging service, which will collect all that application level data and save in the logging area of the data storage. Additional algorithms or scripts or configurations could be provided by those SW modules to the logging & monitoring service in order to do app-specific post-processing of their KPIs and specific visualizations.

5.3.6.2 Technologies used

The golden standard for logging and monitoring within the docker and kubernetes runtime environment is the so-called ELK stack: Elasticsearch indexing DB + Logstash log collection + Kibana dashboarding.

There might be several alternative tools to collect, process and visualize logs with the logging SW module. For example, fluentD can be used instead of Logstash, Prometheus instead of Elasticsearch, Grafana instead of Kibana. But any of them should provide the same overall log collection, processing and visualization workflow. The decision to choose among them depends on the mechanism of how SW components send their data to logging. The prefered way is that all SW modules send both standard output and standard error I/O streams to syslog service. In this case, the ELK services may benefit from using the log driver of the docker software itself, so all docker containers are configured in docker-compose or kubernetes to send logs from all docker containers to ELK. If a particular SW module has very different software stack and does not send logging data to syslog service as standard I/O streams, then sending logs from that docker container may be configured with use of additional log-collector daemon set up to run there, which will transmit its logs to the ELK master host.

Once the log data is being received by Logstash (or another similar log collector, for example FluentD), this data is parsed and streamed to Elasticsearch engine, which will store and index that data according to pre-configured patterns.

Finally, the indexed data will be streamed to Kibana service which will visualize them in a dashboard according to corresponding configurations.

Additional 3-rd party open-source or custom-made SW modules may be created and used to add more functionalities to the application / business logging level. For example, logging and monitoring of users navigation at the web frontend may be tracked with additional mechanisms like use of cookies, GUI events logging, etc.

5.3.6.3 Design and platform Integration specifications

Since the log parsing and routing by Logstash component is based on tags, format specification and tag naming are very important throughout all the SW components of the co-production platform and of the Collaborative Environment web portal. Each SW component should write every logging entry according to the following (or very similar if little adjustment apply) logging format:

{DateTime} {LogLevel} {SwComponentName} {Hostname} {TAG} {LogMessage}

The fields should be separated by the same character for all SW components, for example by ";".

The details and examples of the log fields are specified in the table below:

Table 7. Example of infrastructure logging format specification

Log field	Description	Examples
DateTime	Date and time string of the same format for all SW components, including timezone	2021-12-31 09:45:23 UTC
LogLevel	Level of log severity	DEBUG, INFO, WARNING, ERROR
SwComponent Name	All SW components should be committed to the project's common repository with a fixed name, which should be used both for GitHub directory name and in this logging field. List of names for all the existing SW components should be formally documented.	CoProdWorkflowMgr, AuthService, UserTeamMgmt, AssetMgmt, IncidentMgmt, CommForum, CollaborativeEditor, etc.
Hostname	Host IP or name where this SW component runs (or docker container ID)	
TAG	This may be single (or several keywords separated with different separator), having meaning of application functionality or business logic, which has meaning for streaming and visualisation of log data. SW modules has to define those tags, but should follow a common naming logic here.	UserRegistration, UserLogin, UserEmail, UserAccount, etc.
LogMessage	Meaningful logging message string in a free format defined by each SW module (but having no the field separator character inside)	User id 01s820jf04 registered successfully.

6 Development, deployment and operational environments

6.1 Common source code repository

All the project SW code, both own development as well as re-use and customization (dockerization) of open-source SW when available, should be stored in the common project code repository. The repository is chosen to be stored on the private area of the GitHub web servers here:

https://github.com/interlink-project

The administration of the project area at GitHub is carried out by FBK team and the repository can be accessed only by project members explicitly granted with corresponding permissions by the administration team.

The following top-level structure is proposed in the project github area to organize source code within the project development:

- PLATFORM-CORE = platform backend + platform-type (non-reusable) interlinkers (user mgmt ? auth ? logging with ELK, asset/file mgmt),
- WEB-PORTAL = frontend + web related backend (web server, wizard),
- INTERLINKERs/${name} or INTERLINKER-${name} = customizable and reusable INTERLINKERs (not platform-type): forum, incident mgmt (redmine), collaborative editor, etc.

6.2 Development environments

The software development will be carried out by several project member organizations.

All the SW development, deployment and operations tasks have been split among project members according to their manpower contribution per project work packages (WP). Thus, each project team contributing in SW development and maintenance (DevOps) activities should have its own sub-areas within the common GitHub repository.

For the SW development and local testing of SW modules (unit/integration testing), organizations are supposed to use their own local or cloud servers, so the source code and DevOps scripts/tools are checked out and tests get executed on organisations own servers.

Standard SW development methodologies (Agile, iterative/spiral and continuous integration) are expected to be used by each project team locally within their area of responsibility, however, each team has flexibility to organize their own SW development work according to the amount of work and available manpower (e.g. strict following of 2-week SCRUM cycles is not required). The typical phases of SW development cycle are supposed to pass: prototyping, development, unit testing, integration testing, release build and testing, SW release deployment and operations.

Integration of SW components and corresponding integration tests should be carried out by the corresponding SW module teams together on local servers of one of them and only after having per-module unit tests passed.

Once corresponding SW modules are released, the overall project SW integration tests and demo can be carried out by all IT teams of the project.

6.3 Deployment environments

Docker containerization should be used for deployment of each SW component, so each SW module development should end-up with a script building a docker image out of the source code, and starting that docker image up for running.

Administration and run-time management of the containers should be carried out with one of the implementations of Kubernetes clustering technology (Rancher or similar). Docker-compose or docker swarm could be used in local SW module development and unit testing.

Globally within the project, it is supposed to have the following four deployment environments:

- one project-wide SW release demonstration, irrelevant or coinciding with one of the 3 pilot cases; this will include functionalities of the public service co-production process and corresponding Collaborative Environment;
- separate deployments and (demo) operations for each of the 3 pilot cases, which correspond to co-execution of public services:
  - one public service for Zaragoza / Spain,
  - one public service for MEF / Italy,
  - one public service for VARAM / Latvia.

6.4 Co-production platform operations & monitoring

Once deployed (installed) at the corresponding environment, the project SW system (co-production platform or pilot co-execution platform) should be operated (pre-configured, started up) and continuously monitored by corresponding project teams involved in that co-production platform demo or pilot case demo.

The logging and monitoring SW module described above will be used at this stage.

6.5 User support incident management

While operating at runtime (mostly for the pilot cases or post-project co-execution of newly co-produced public services), various types of incidents, either at infrastructure, software or user levels, may occur. To deal with them all in a coherent manner, corresponding user support and incident resolution policies have been established within the project. They are described in more detail in document D5.1.

The open-source Redmine incident management SW has been adapted to automate such incident management. This SW will run as a docker-based microservice at SW platform level.

In brief, the entire support process will be split into the following 2 levels:

- L1: a supporting team which communicate to users, collect incident report from them, properly log it into the incident management system, classify type of incident and resolving it within L1 if it is usability or knowledge related issue, or pass it to the next L2 if it is a technical problem,

- L2: attend technical type incidents logged by L1 team in the incident management system.