MediaVerse

A universe of media assets
and co-creation opportunities

D6.1

Core Framework, Decentralized Communication and Content

Project Title	MediaVerse
Contract No.	957252
Instrument	Innovation Action
Thematic Priority	ICT-44-2020 Next Generation Media
Start of Project	1 October 2020
Duration	36 months

Deliverable title	Core Framework, Decentralized Communication and Content Exchange
Deliverable number	D6.1
Deliverable version	V1.0
Previous version(s)	N/A
Contractual Date of delivery	31.01.2022
Actual Date of delivery	15.02.2022
Nature of deliverable	Report
Dissemination level	Public
Partner Responsible	ATOS
Author(s)	Enrique Quirós Fernández (ATOS), Antonio Castillo (ATOS)
Reviewer(s)	Manos Schinas (CERTH), Symeon Papadopoulos (CERTH), Stratos Tzoannos (ATC), Marco Giovanelli (FINCONS)
EC Project Officer	Alberto Rabbachin

Abstract	This deliverable presents a functional and technical description of the core elements of a MediaVerse node. The Digital Asset Management and the Decentralized Framework are described in detail. Screenshots of the current status of the user interface are also provided.

Keywords	Digital Asset Management, Decentralized framework, Dashboard, User Interface, Federation Shared Dataspace, Federated Search.

Copyright

MediaVerse is an H2020 Innovation Project co-financed by the EC under Grant Agreement ID: 957252.
The content of this document is © the author(s). For further information, visit mediaverse-project.eu.

Revision History

VERSION	DATE	MODIFIED BY	COMMENTS
V0.1	21/12/2021	Enrique Quirós (ATOS), Antonio Castillo (ATOS)	First Draft Table of Content
V0.2	03/01/2022	Enrique Quirós (ATOS), Antonio Castillo (ATOS)	Final Table of Content
V0.3	19/01/2022	Enrique Quirós (ATOS), Antonio Castillo (ATOS), Andrea Cavallaro (FINCONS), Stratos Tzoannos (ATC)	First Draft
V0.4	24/01/2022	Enrique Quirós (ATOS), Antonio Castillo (ATOS)	Second Draft
V0.5	31/01/2022	Enrique Quirós (ATOS), Antonio Castillo (ATOS)	Final Second Draft
V0.6	02/02/2022	Enrique Quirós (ATOS), Antonio Castillo (ATOS)	Final Second Draft Review
V0.7	07/02/2022	Enrique Quirós (ATOS), Antonio Castillo (ATOS)	Review
V0.8	10/02/2022	Enrique Quirós (ATOS), Antonio Castillo (ATOS)	Improved draft
V0.9	14/02/2022	Symeon Papadopoulos (CERTH), Evangelia Kartsounidou (CERTH)	Final QA and refinements
V1.0	15/02/2022	Enrique Quirós (ATOS), Antonio Castillo (ATOS)	Final Document

Glossary

ABBREVIATION	MEANING
API	Application Programming Interface
DAM	Digital Archive Management
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
IPFS	InterPlanetary File System
MV	MediaVerse
NDD	Near Duplicate Detection
UI	User Interface
WCAG	Web Content Accessibility Guidelines
WP	Work Package

Table of Contents

Index of Figures

Executive Summary

This deliverable named D6.1 - Core Framework, Decentralized Communication and Content Exchange aims to provide a reference guide throughout the development of the WP6: i) with the first version of the architecture of the Core framework of a node, including the environment for creation, storage service and the user interface that will enable the users to access the different services, and ii) by describing the Decentralized framework and Federation shared dataspace.

Section 2 describes the DAM, a key element of the MediaVerse node which participates in almost all interactions between the components of the ecosystem. Subsection 2.4, regarding the Dashboard UI, describes the already developed sections of the user interface that will enable the users to access the different MediaVerse nodes in an intuitive, user-friendly way. Also, it explores the Dashboard from a technical point of view, listing the most relevant libraries used to develop it.

Section 3 describes the work done for the creation of the decentralized framework for MediaVerse inter-node communication. This framework allows peer interactions among MediaVerse nodes without the need to pass through a master control node. The auto-discovery mechanism to allow any node of the MediaVerse decentralized network to be aware of new and existing nodes is also explained. As an example of communication among nodes, it also describes the federated search service that allows expanding the search of media content beyond the boundaries of a MediaVerse node.

1 Introduction

This deliverable summarizes the activities carried out in MediaVerse WP6: "Content Management System Setup and Integration" during the first months of the WP lifecycle. The present document will be updated by D6.3 - Core Framework Decentralised Communication and Content Exchange v2, planned for month M24.

WP6 develops and integrates into MediaVerse nodes the functionality for creation, storage, editing, exchange and publication of content. Also, it is responsible for implementing the architecture for deploying and managing MediaVerse nodes and for the communication layer that makes possible the interaction between nodes. Additionally, it implements a User Interface that makes it possible for the user to access the different services available in MediaVerse in a user-friendly and intuitive way.

Figure 1, which has been extracted from D2.2 - Conceptual Design of the MediaVerse Framework[1], depicts the global architecture of the MediaVerse federated network (consisting of a pair of only two nodes) that communicate between them. From the services presented inside every node, this document will explain in detail the DAM, the Federated search, and the User Interface. Figure 1 also depicts the Federation Shared Dataspace, which is responsible for distributed storage among all the members of the federated network of MediaVerse using IPFS, a protocol for implementing decentralized file system networks. All this process is explained in detail in section 3.2.

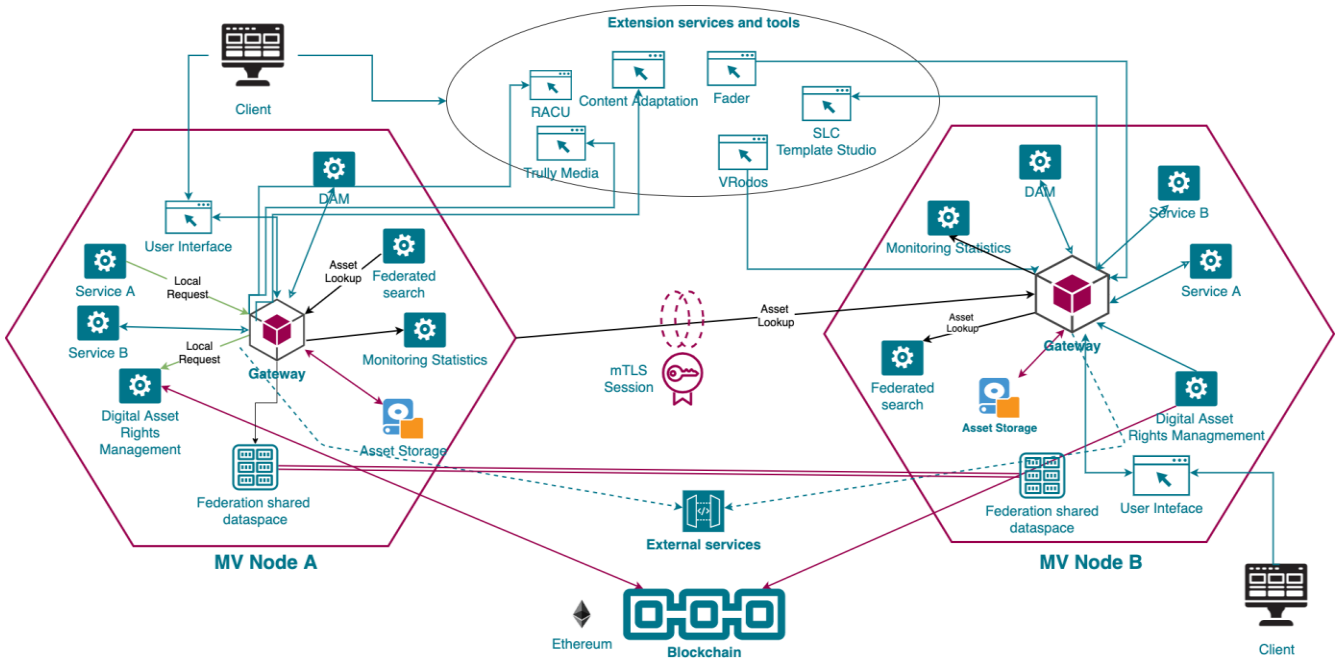

Figure 1: MediaVerse conceptual architecture

[1] https://mediaverse-project.eu/wp-content/uploads/2021/10/MediaVerse_D2.2_Conceptual-Design-of-the-MediaVerse-Framework.pdf

2 Digital Asset Management

The following section is focusing on the Data Asset Management (DAM) component of the MediaVerse node, as described in D2.2 - Conceptual Design of the MediaVerse Framework[2]. The DAM is a central element of the MediaVerse node and participates in almost all interactions between the components of the ecosystem. It is responsible for the implementation of the business logic of specific workflows, including user management, access rights and local search and retrieval, which are described in subsequent sections.

2.1 Functional description

The DAM is responsible for the management of digital multimedia objects. Moreover, it acts as the access point for all incoming calls to the MV Node by external clients (e.g., web application running in browser, distributed nodes, etc.). Except for managing the import, export, and processing of content, it also handles various workflows of the system such as user management, access rights, local search and retrieval and it provides functionality for organizing content in groups and collections. The main functionality is described in Figure 2.

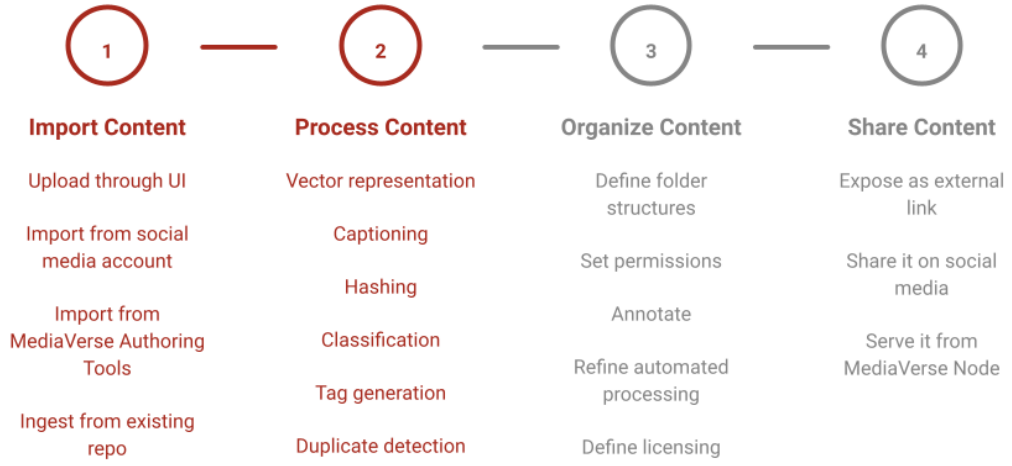

Figure 2: DAM basic functionalities

The functionalities of Figure 2 may be needed by other modules of the MediaVerse Node and are exposed by the DAM as consumable HTTP methods. This set of methods is continuously updated while the rest of the modules are progressing with development. We maintain a live version of the available methods through a Swagger interface[3]. This offers an interactive user interface where developers can try and test the available HTTP methods before integrating them with their components. A preview of the exposed API can be found in Figure 3.

The important architectural decision we have reached regarding the role of the DAM inside the MediaVerse Node indicates that the DAM must control the MV Node storage layer and data-structures of all the distributed (microservice) elements of the Node. For instance, all IPR-related metadata and persistent objects are handled through the DAM. This applies to the entire MediaVerse Node where CRUD actions towards the storage must pass through the DAM.

[2] https://mediaverse-project.eu/wp-content/uploads/2021/10/MediaVerse_D2.2_Conceptual-Design-of-the-MediaVerse-Framework.pdf
[3] https://mediaverse-node.herokuapp.com/swagger-ui/#/

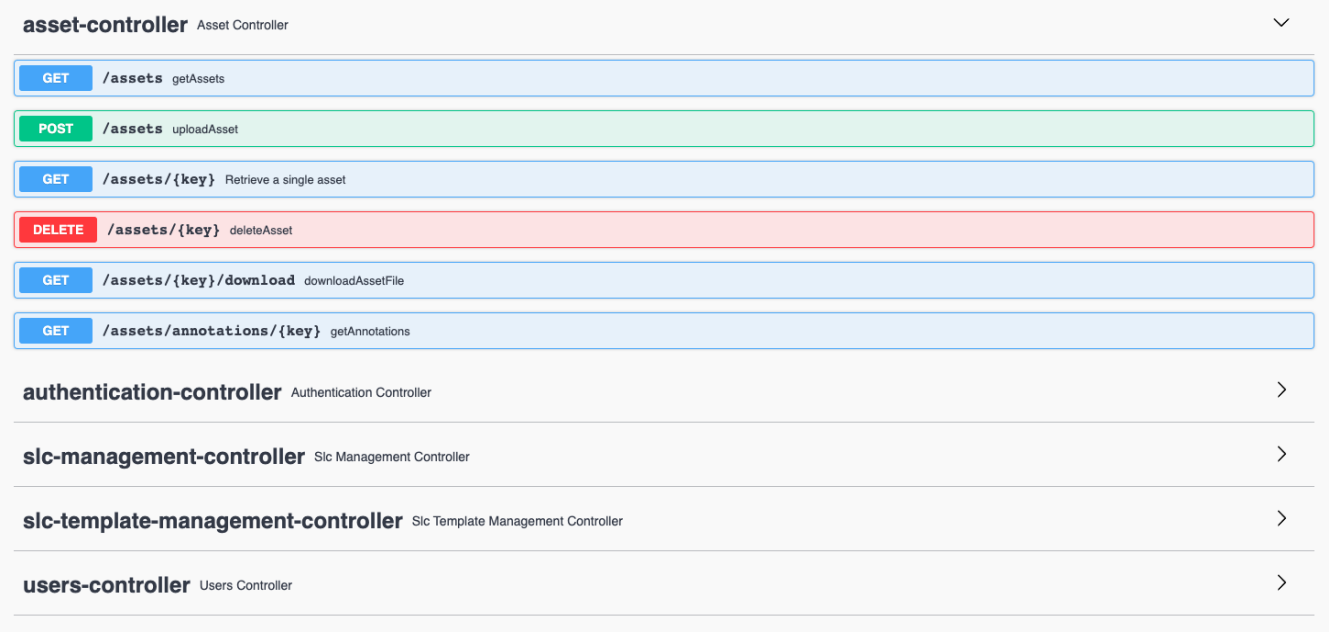

Figure 3: Swagger representation of DAM API

Except for exposing HTTP services to other MV Node components, the DAM also consumes the APIs of other MV Node elements and orchestrates several workflows as described at the beginning of this section. The way of consuming the available interfaces is explained in section 3.5.3 of the Deliverable D2.2 - The high-level diagram of this interaction is seen in chapter 2.4.2 (Figure 16) of this document.

2.2 Technical description

Conceptual Architecture

The DAM follows the microservices approach. It is itself a microservice inside the MediaVerse Node, it exposes a RESTful API and is connected with the storage layer. The storage layer consists of a binary object storage, the searchable indexes and metadata storage. Despite the fact that the storage layer can be physically deployed outside the implementation of the DAM e.g., as cloud storage, conceptually it can be considered as part of the same microservice. Moreover, since all interactions to the storage layer are directed through the DAM, we can consider it as a standalone microservice, binding the following layers, as described in Figure 4.

This grouping is based on the role of each layer: The DAM API is responsible for the implementation of the exposed RESTful HTTP methods. The Business Logic implements all the algorithms and rules that are needed for the execution of the more complex tasks inside the application. The Domain Objects contain all the structures of the objects used inside the application. Finally, Data Access and Data Connectors are responsible for the communication (read, write, and update) with the Storage Media.

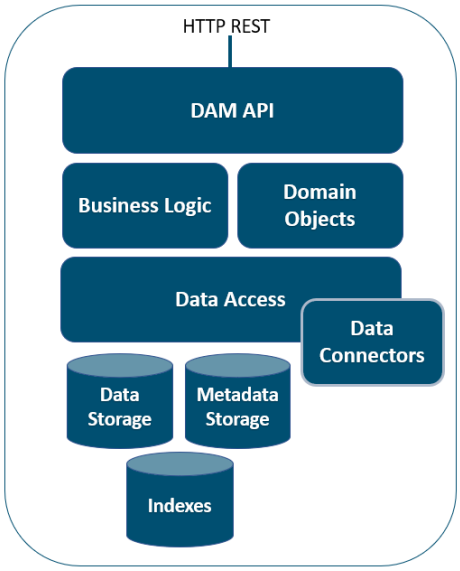

Figure 4: The DAM Microservice

Technical Implementation

From a technical aspect, all the layers of the DAM are implemented under a single software application in the Java programming language selected because of its robustness and efficiency in enterprise systems. More specifically, we have used the Spring Boot framework [4]with embedded Apache Tomcat to be used as a web server. The code structure found in the internal GitLab repository is described in Figure 5, as follows.

The project is organized in Java packages following the structure below (in parenthesis, the relevant conceptual elements previously described in the layered architecture of the module):

- **Controllers** (DAM API). This describes all the REST methods that handle external HTTP requests and messages.
- **DTO** (DAM API). This defines all the Data Transfer Objects used inside the message exchanges between the DAM and the external clients.
- **Model** (Domain Objects). This domain layer describes the structure of all the objects used in the business layer.
- **Service** (Business Logic). These services implement the business logic of the application.
- **Repository** (Data Access, Data Connectors). These are the connectors and CRUD logic with the Databases and other storage media of the ecosystem.
- **Exception**. These are the descriptions of all the exceptions that can occur inside the application.

The **Storage Systems** used for the persistence of the various objects (metadata, binary files, indexes) are currently defined as follows:

- MongoDB as NoSQL Database for the metadata
- Apache Solr for the full text search indexing
- IPFS for the inter-node communication (registry and pub keys)
- Local Volume, AWS S3, Azure or other for the binary storage

[4]https://spring.io/projects/spring-boot

Name	Last commit	Last update
..		
configurations	EUMV-68: Fix the dockerization of the application	1 month ago
controllers	EUMV-75: Update Swagger - WP6 DAM OpenAPI Issues #18 ...	1 week ago
dto	EUMV-75: Update Swagger - WP6 DAM OpenAPI Issues #23	1 week ago
exception	Fix brocken Exception Advice Handler	1 month ago
model	EUMV-75: Update Swagger - WP6 DAM OpenAPI Issues #18	1 week ago
repository	EUMV-75: Update Swagger - WP6 DAM OpenAPI Issues #21	1 week ago
service	omitting white spaces in metadata	1 day ago
sorlRepositories	EUMV-34: Complete refactor to submodules	6 months ago
JWTFilter.java	EUMV-75: Update Swagger - Updated to Swagger 3	1 week ago
MediaverseApplication.java	EUMV-68: Fix the dockerization of the application	1 month ago
MediaverseConfiguration.java	EUMV-34: Complete refactor to submodules	6 months ago
MongoConfiguration.java	EUMV-34: Complete refactor to submodules	6 months ago
SolrConfig.java	EUMV-34: Complete refactor to submodules	6 months ago

Figure 5: Code structure

The latter is agnostic by design and can be handled programmatically with the replacement of the Data Connectors implementing the same interface. Therefore, following the adapter design pattern, the DAM could switch to support any storage implementation needed.

Secure Communication

In terms of security, the DAM is using server-signed JWT tokens, which are validated on every request to the DAM by any external clients (e.g. browser or external applications). The formulation of such JWT tokens takes place during the OAuth 2.0 handshake and uses a private passphrase to protect it from replication. More specifically, once the user gets authenticated with the use of credentials, the DAM creates a JWT token, signed by itself with a passphrase. This means that only the DAM server that owns the passphrase can validate it and no other component is able to create valid tokens. Once a token is returned to the client web app it is stored to the browser's local storage to be used in all subsequent requests.

A significant advantage of the DAM application is that it is loosely coupled with the front-end interface. This means that there is no maintained open session between the triggering part and the DAM backend. As explained, every call is stateless and authenticates the actor every time with the same JWT token. This also facilitates the replication of the backend in multiple serving instances, offering load balancing, high availability, rolling upgrade functionality, etc. This functionality is described in the "Deployment" section, subsequently.

2.3 Deployment

The deployment of the DAM component is feasible via the containerization of the application. The software is using Docker in order to build the application as a standalone Docker image. This Docker image is then uploaded to a Docker registry and can be pulled by anyone in order to be deployed. A basic configuration file is used to configure the application for the custom environment.

The Docker image can run as a Docker container on any Docker runtime (cloud-based or on-premise) and can be orchestrated by Kubernetes and Istio, as specified by the MediaVerse conceptual architecture. The main advantages of containerization are that the software code is not needed for this deployment process and that the built application is environment-agnostic, therefore it can run anywhere. Moreover, the application can be packaged in deployable software containers (named pods), fully orchestrated to offer load balancing and high availability. The DAM microservice as described from a conceptual point of view can be split into different Docker images (e.g., MongoDB, Java backend, Apache Solr full-text-search Server etc.) for better management. All these modules will collaborate with each other and will be orchestrated by the same Kubernetes master node.

The build and deployment process are fully automated via a CI/CD task which runs on a Jenkins CI/CD tool. This task is responsible for running a pipeline that builds and compiles the Java code into a Docker image and uploads the Docker image on a Docker registry, ready to download and run. The extra step of the pipeline focuses on deployment. For the moment, we can deploy this instance on either a Kubernetes container on-premise or on the cloud, on a Heroku Dyno instance, which is easy for demonstration and testing purposes.

2.4 Dashboard - UI

This section describes the MediaVerse node user interface that will enable users to access the different services of MediaVerse in an intuitivey way through a web application. For now, the dashboard functionality is focused on user and content management, giving support to images, audio, video and 3D content (Figure 6). This web application complies with the WCAG2.2 standard and aims to be accessible for as many users as possible.

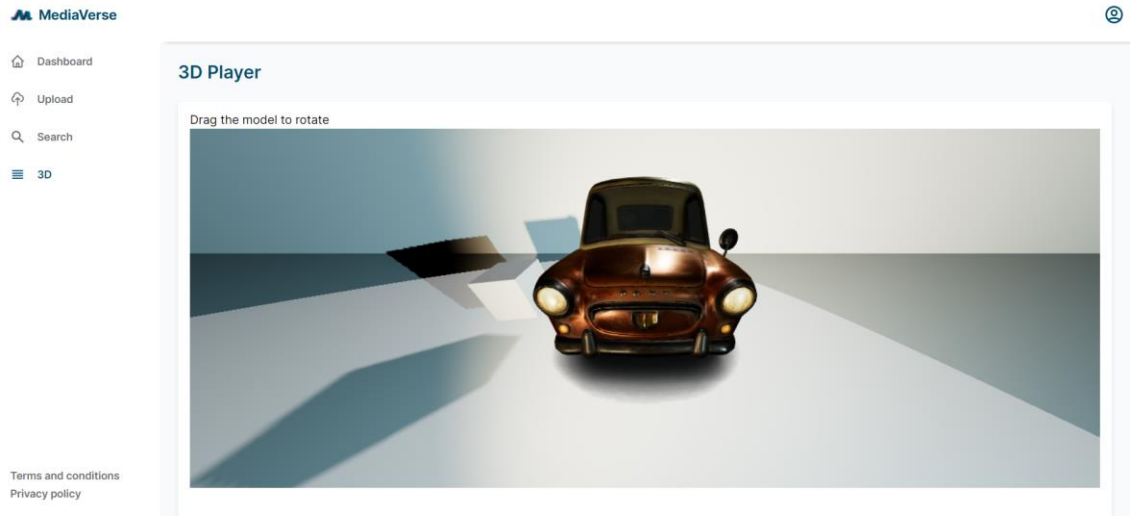

Figure 6: Detail of the Content detail section for 3D content

2.4.1 Functional description

The dashboard is integrated with the DAM backend that, at the same time, is integrated with several MediaVerse services. All the current web sections are still a work in progress and will keep being improved in the next period. Below, you can find the sections that are already working.

Register user

This section allows new users to register themselves in MediaVerse through a particular node. The minimum information needed to create the account is an e-mail address, username and password. Alternatively, it will soon be possible to log in using an already existing social media account.

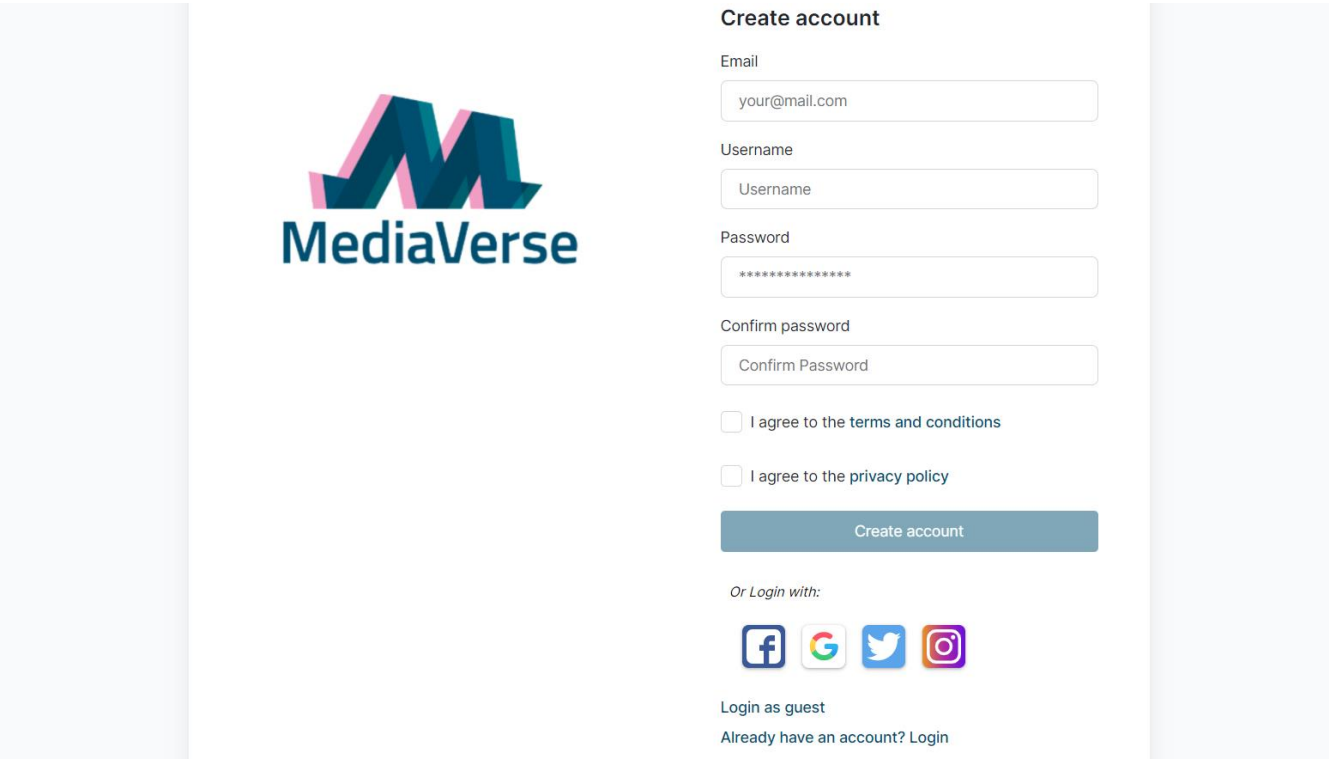

Figure 7: Register user screen

Login

Like any other web application, this section allows the users to enter their credentials and access the MV node where they have been previously registered.

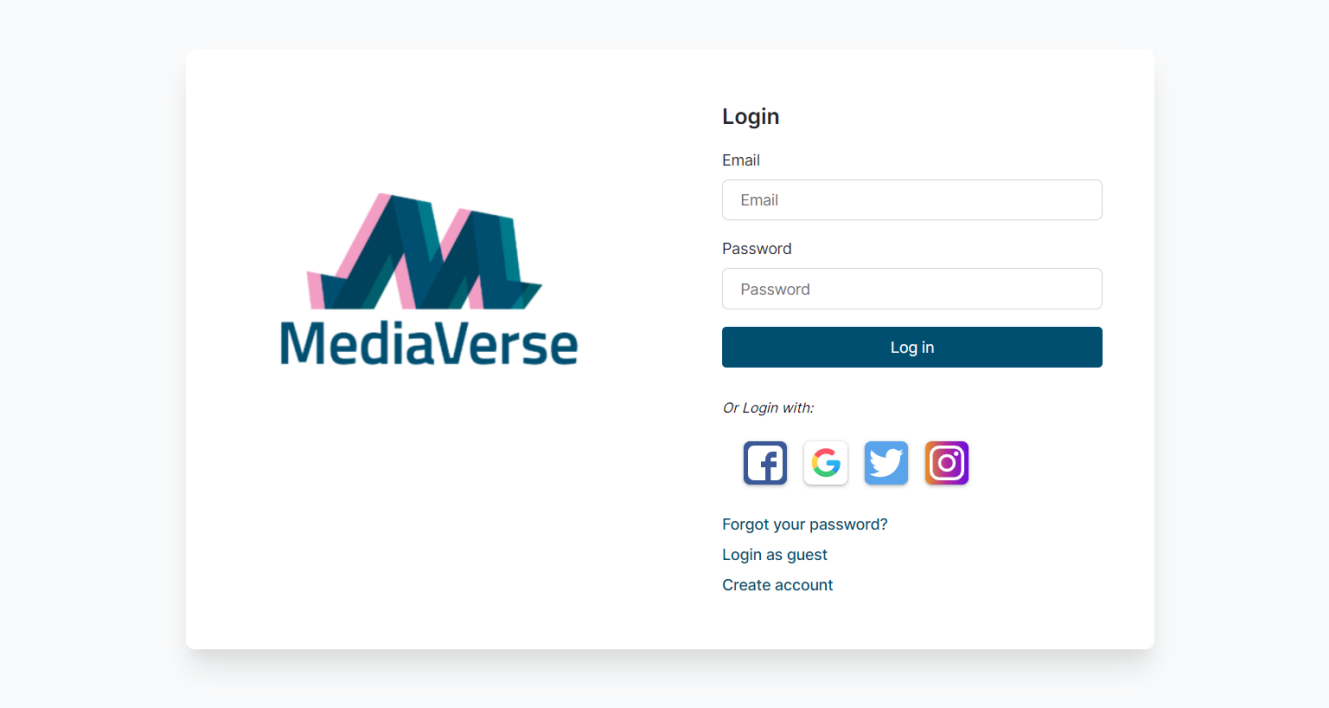

Figure 8: Login screen

Dashboard

This section shows users useful information related to their activity on the platform. The informative cards placed in this section will be developed according to the user requirements defined in D2.1 - Use Cases and User Requirements[5]. Including a list of current projects of the user, a list of content uploaded by the user, history of transactions, invitations to collaborate received, or a list of relevant open calls.

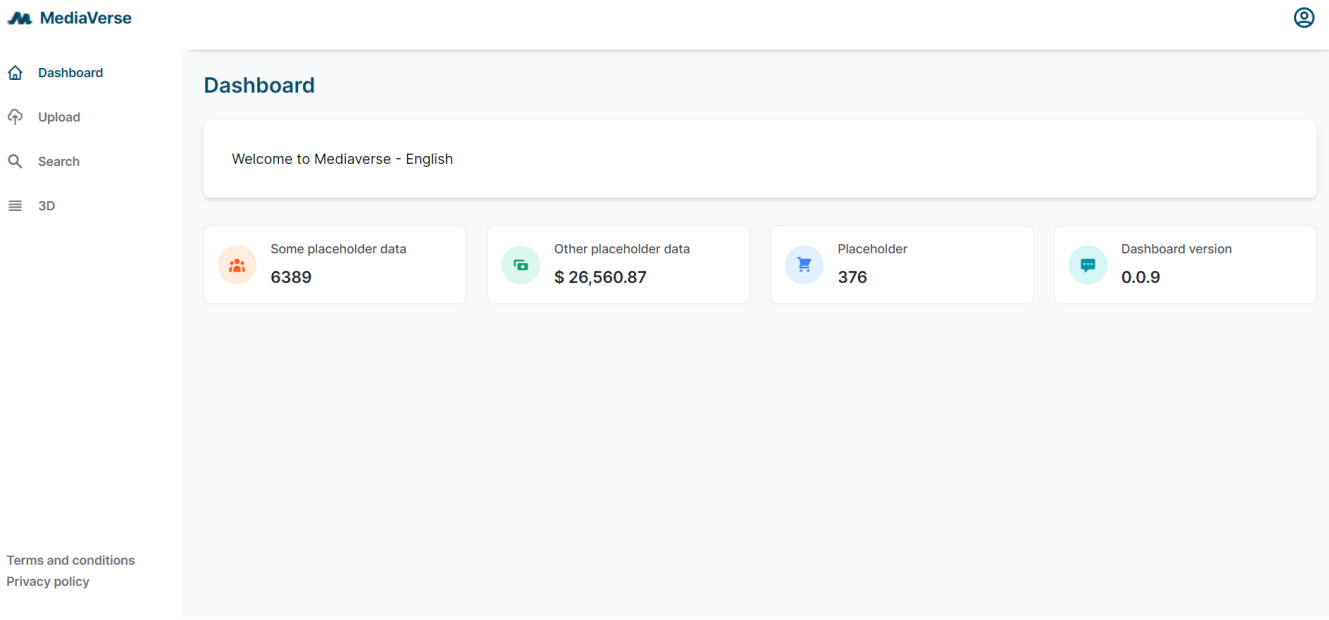

Figure 9: Dashboard screen

[5] https://mediaverse-project.eu/wp-content/uploads/2021/04/D2.1-V1.0.pdf

Profile

This section allows the users to edit their information and preferences (Integration with the DAM backend). In the future, it will also be possible to add billing information and link the account with other social platforms of the user.

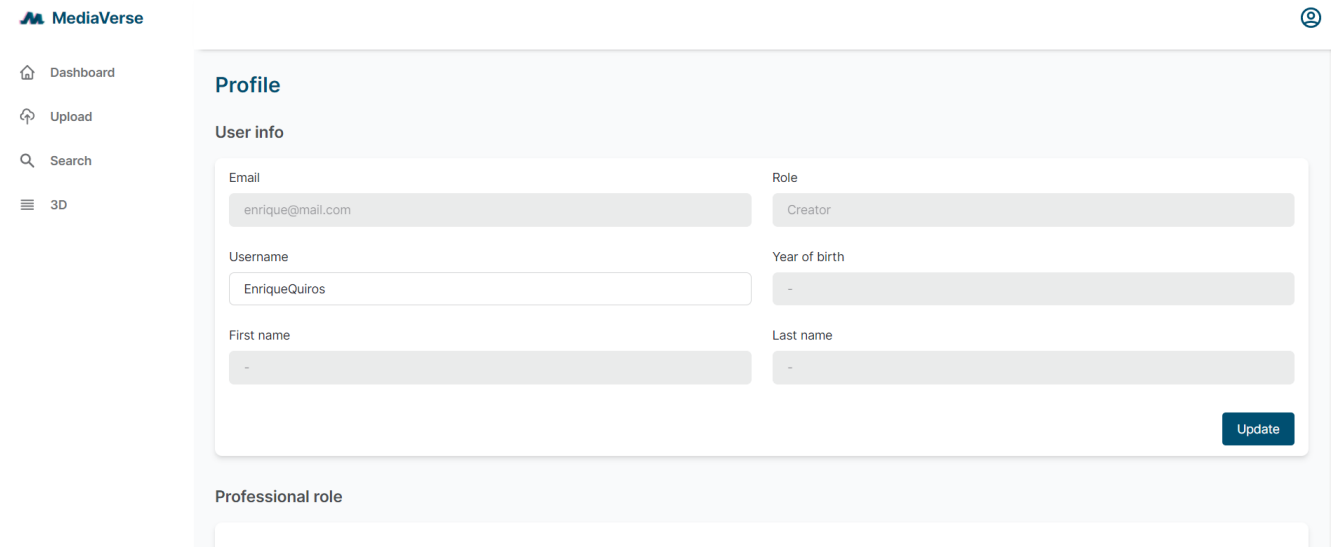

Figure 10: Profile screen

Upload

In this section, the user can upload new content (images, audio, video, 3D models) to the platform just by dragging and dropping it to the designated upload space (Figure 11). Once the content is uploaded, it will be available to its owner on the search screen. The drop area is able to distinguish the type of the content and react to it in several ways. For example, if the dropped content is a video, the screen will show a new menu with all the transcoding presets so the user can choose one of them.

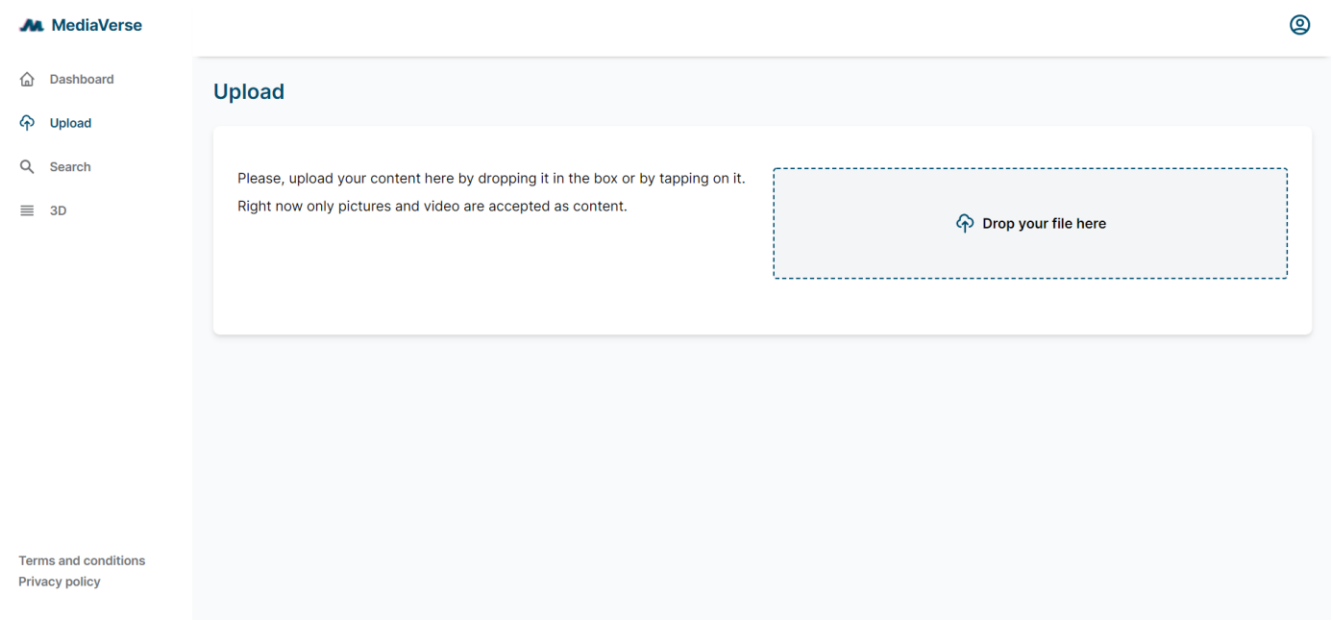

Figure 11: Upload screen

Search

In the search section, users will be shown by default their own content, and will be able to filter the results by name or other attributes (category, date, language, etc.). Here they will also be able to search the MediaVerse network or get recommended content. Once they click on a result, they will be able to preview it and see its details (Figure 12).

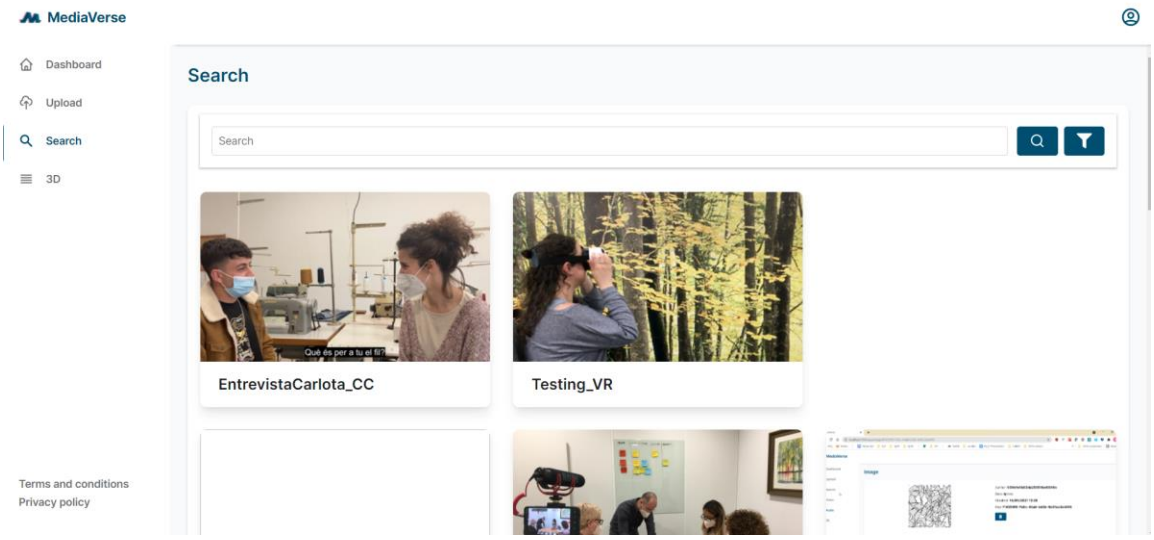

Figure 12: Search screen

Content detail - Metadata

In the content detail section (Figure 13), there is available information about the selected content, and it is possible to preview it. The content could be images, audio, video (regular or 360), or 3D models. Also, the owner can edit the metadata (integration with T3.4) of their content and publish it.

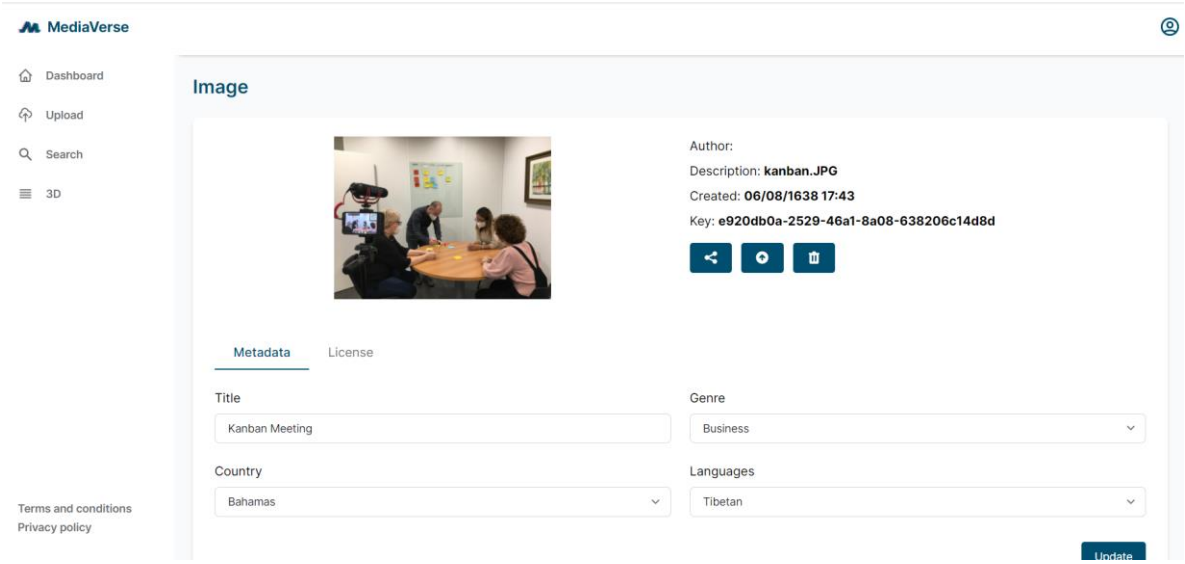

Figure 13: Content detail screen

Content detail - License advisor and Licensing

The content detail section allows their owner to choose a license for that content (Figure 14). If the user needs help choosing the license, they can use the License advisor functionality that will ask (Integration with T4.3) questions about the use that they want to allow for their content. The License Advisor will suggest the most fitting license according to the answers given by the user.
The user can freely choose the preferred License and the user interface shows the required fields to be filled (e.g., add other owners to the Ownership Deed, etc.). Then the Web UI (through the DAM) interacts with the Digital Asset Rights Management component, in order to "notarise" the License on the Blockchain (for further details, please refer to D4.1 - Copyright and Procedures for IPR Definition[6] for an overview of the main concepts and to D4.2 - Blockchain repository v1 for the technical implementation).

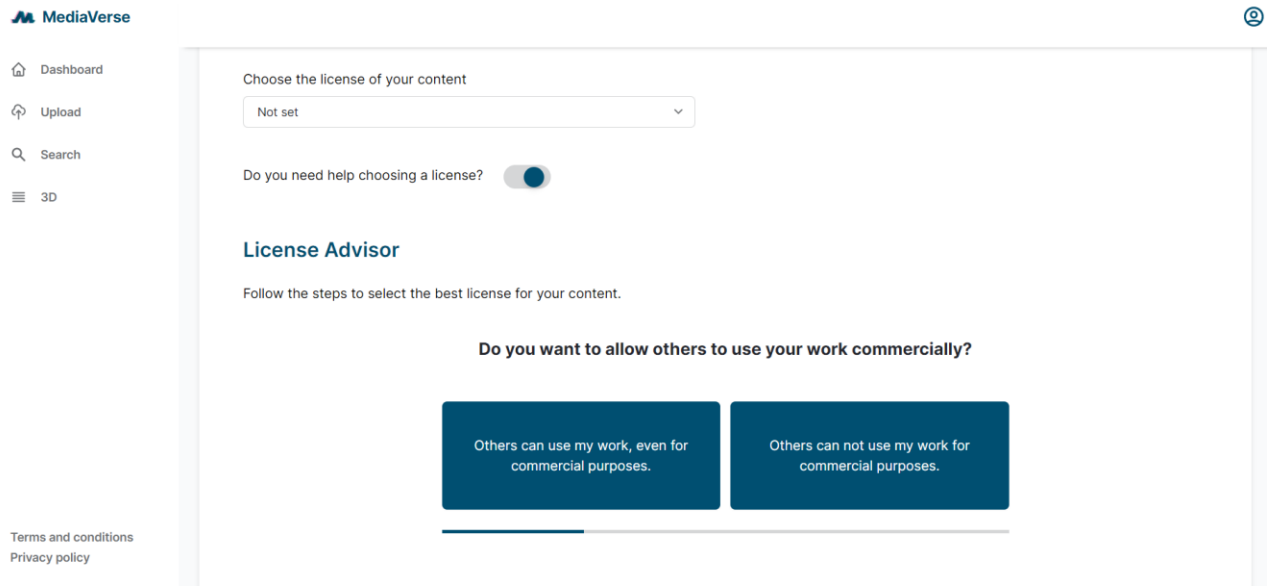

Figure 14: Detail of the License Advisor section

2.4.2 Technical description

This section explains the technical implementation aspects of the dashboard.
The Dashboard is a web application made in React. React.js is an open-source JavaScript library used for building user interfaces, specifically for single-page applications that can change the shown data without reloading the page. The Dashboard also uses the following libraries:

- Tailwind CSS, a utility-first CSS framework for rapidly building custom user interfaces.
- React Three Fiber, a React renderer for three.js. In the dashboard, it is used for previewing 3D content.
- Video.js, a web video player. It supports numerous video formats, including adaptive streaming formats, with focus on accessibility and it's compatible with 360 videos.

Figure 15 shows how the dashboard UI code is structured in folders.

- **Assets.** It contains all the assets used in the app. Most of the content is images or icons.

[6] https://mediaverse-project.eu/wp-content/uploads/2021/10/MediaVerse_D4.1_Copyright-and-Procedures-for-IPR-Definition.pdf

- **Components.** The code has been structured in components, that are independent blocks representing sections of the web app and that can be easily moved or reused in other sections.
- **Containers.** Containers are special components that add common functionality to UI components or provide them data.
- **Pages.** Contains the different sections of the web app. Every page has its own route so it can be accessed directly from a URL.
- **Routes.** Routes link specific paths with pages, making sure that when a URL is introduced the app will render the appropriate section requested by the user.
- **Store.** A store is a state container that holds the application state that will be shared between all the components that need it.
- **Styles.** This folder contains all the CSS files needed to style the application.
- **Translations.** This folder has a subfolder per language supported by the application. Every text in the app that needs to be translated will need to be declared here.

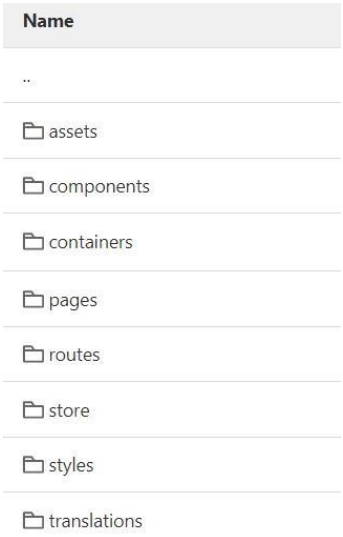

Figure 15: Dashboard code structure

Figure 16 represents how the Dashboard is related to the rest of MediaVerse services. All the communications from the dashboard with other services go through the DAM using API REST calls. The web app is running from a docker container that includes its own web Server (currently Nginx). This makes the dashboard deployment independent from the rest of the modules and easy to update. There is not a common URL to access MediaVerse nodes. Every MediaVerse node will have its own URL, and to access the web app, the end-users will need to use the URL of the node that they are affiliated with.

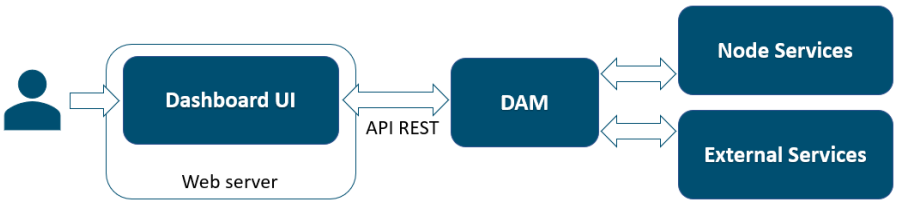

Figure 16: Dashboard communication with DAM

3 Decentralized Framework for MediaVerse Communication

The main objective of this section is to describe the design and implementation of the communication interface that manages the interactions between MediaVerse nodes for identifying, sharing, and exchanging content and fast negotiation of media rights. The decentralized framework for MediaVerse communication will operate as a federation of MediaVerse nodes implementing these functionalities:

- Auto-discovery of new MediaVerse nodes in the network, allowing awareness of their existence to the rest of the nodes.
- Search for content in the MediaVerse nodes where each node will manage their content indexes.
- Facilitation of communication for agile rights management and copyright negotiation.
- Exchange, retrieval and sharing of digital assets between MediaVerse nodes.

We have designed a group of services to implement these functionalities and taken decisions about the communication protocols to allow the communication between MediaVerse nodes. The logical architecture of this framework was provided in D2.2 - Conceptual Design of MediaVerse Framework[7] and it is shown again for convenience in Figure 17.

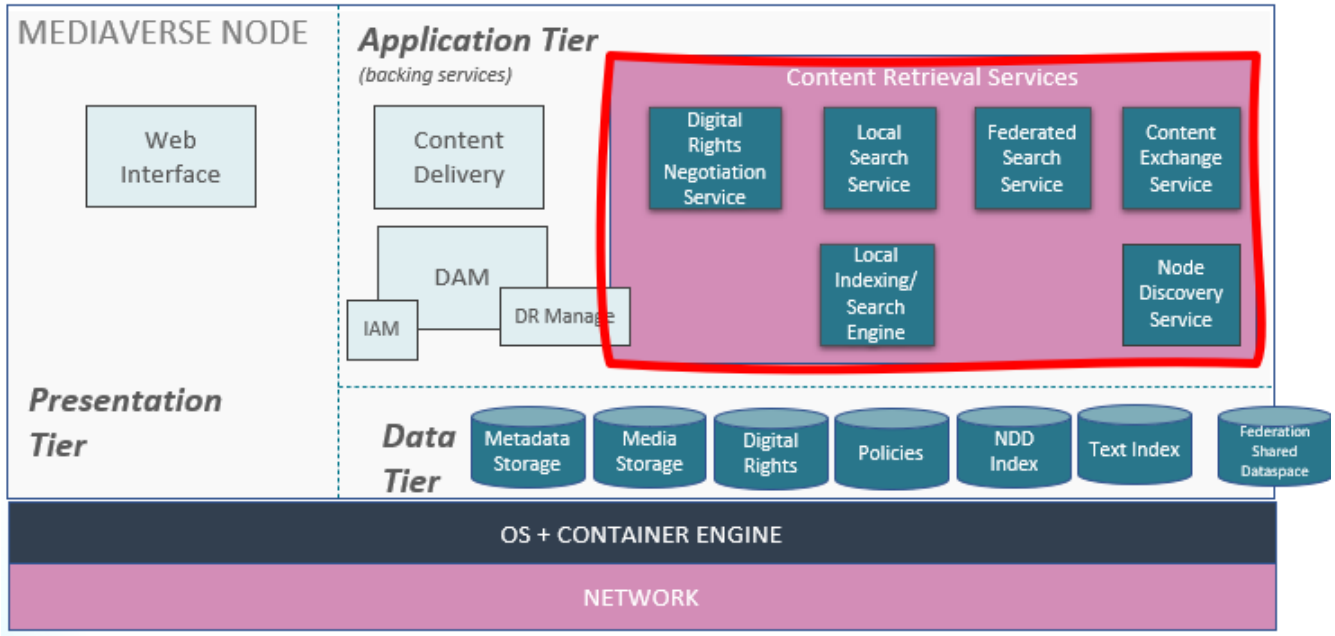

Figure 17: Decentralized framework logical services (inside the red rectangle)

The list of services that make up this framework are the following:

- The **Federation Shared Dataspace** (Content Exchange service in the Figure 17). This service is responsible for an additional distributed storage among all the members of the federated network to be used as Public Keys storage. This storage layer will be complementary to the main storage layer of each MediaVerse node managed by the Asset Storage service described in D2.2 - Conceptual Design of the MediaVerse Framework.[7]

[7] https://mediaverse-project.eu/wp-content/uploads/2021/10/MediaVerse_D2.2_Conceptual-Design-of-the-MediaVerse-Framework.pdf

- The **Node Discovery** service**.** This service allows access to a distributed Node Registry of nodes connected to the federated network. It is part of the Federation Shared Dataspace.
- The **Federated Search** service will be in charge of searching and retrieving content within the MediaVerse network. It will be responsible for sending the search queries and collecting search results across the network, but it delegates the indexing, search and retrieval of content to the DAM (Digital Assets Management) of each node (Local Search service in the Figure 17).
- The **Digital Rights Negotiation** service will be responsible for facilitating the communication among the different Digital Asset Rights Management components of MediaVerse nodes in the federated network for digital asset rights negotiation.

All the nodes of MediaVerse include an instance of each of these services. Every service of this framework connects with its counterpart in every node of the distributed network following an established protocol developed in this project to allow this communication.

3.1 Federation Shared Dataspace

3.1.1 Functional description

The **Federation Shared Dataspace** service is responsible for the distributed storage among all the members of the federated network with the purpose of storing and retrieving Public Keys. This will allow any node to sign or validate the signature of any message/content exchanged between nodes using asymmetric key algorithms.

This service will also have a distributed registry of nodes connected to the federated network (**Node Discovery** service). Any node will be able to auto-discover the address of current active nodes in the federated network. This distributed registry of nodes will be shared among all the nodes using redundancy and partitioning mechanisms.

3.1.2 Technical description

For the implementation of this framework, we followed the microservice approach and the exposition of functionalities of every microservice by means of REST API interfaces. We have created for this first release two microservices (Federated Search and Federation Shared DataSpace) following the API first contract approach. These services will be part of the core services of every MediaVerse node. For the specification of the REST API interfaces we are using OpenAPI 3.0 to document (HTML format) and generate code of the exposed APIs. All the services in this decentralized framework follow the same approach.

The Federation Shared Dataspace is based on the IPFS (**InterPlanetary File System**)[8] framework. IPFS is a protocol and a peer-to-peer network for implementing decentralized file system networks. This distributed file system will be based on content-addressing to access content in a global namespace inside the decentralized peer-to-peer network.

IPFS allows us to create a public or a **private** (permissioned) **distributed network**. The main difference is the way how the nodes join the network. In a public network, we only need one address of a node already connected to the network to be part of this network but in case we choose a private network we also need a shared secret key

[8]IPFS. InterPlanetary File System. https://ipfs.io

(**swarm key)** to be part of the network. In a private network, the nodes do not interact with other nodes outside of the network. For MediaVerse nodes, we have decided to use a private/permissioned network to facilitate the administration of common functionalities of the network like global policies, security access, digital rights management, etc. A convenient installation script will be available to allow anyone to join the MediaVerse network without worrying about the technical details that take place under the hood because MediaVerse is an open network, and our aim is to lower the barriers to entry as much as possible. At the same time, we need to separate the MediaVerse network from other IPFS networks.

The inter-node awareness discovery will be based on a DHT (**Distributed Hash Table**) supplied by IPFS. This distributed table is split across all the peers in the distributed network and consists of key-value pairs. This table stores the location address (IP addresses) and UID (node ID) of each node of the federated network in the distributed file system.

For the implementation of our service (Federation Shared Dataspace), we will wrap calls to the IPFS API[9] to communicate with an official container image of the IPFS node implemented in Golang running in the local node.[10] This official IPFS node container will be part of the services of every MediaVerse node. This IPFS node will be responsible for connecting the MediaVerse node to the MediaVerse decentralized network of IPFS nodes.

IPFS uses three main ports to be opened in the network. These ports correspond to the official container image of the IPFS node:

- Internode communication: port 4001. Protocols TLS and QUIC.
- Management API: port 5001. Protocol HTTP/S.
- HTTP Gateway: port 8080. Protocol HTTP/S. This interface allows us to get content from IPFS without implementing any IPFS protocol (useful for web browsers). This is not used in MediaVerse at the moment.

The Management API port (5001) will be opened only for local connections of the MediaVerse node or the backend network. It will not allow remote connections by configuration. The same case applies to the HTTP gateway (8080). We must allow the remote connection to port 4001 to inter-node communication to establish the private decentralized network of IPFS nodes. This last connection will use IPFS proprietary network protocols (based on libp2p[11] modular network stack) with mTLS connections for encryption in transit.

Besides, our Federation Shared DataSpace service will use the port 4040 and the protocol HTTP/S. This service will be only available inside the node or the backend network. We do not need to expose this service to the network. Every service of a MediaVerse node that needs access to any functionality of IFPS will be handled by our service (e.g. Federated Search service for pub/sub functionalities).Figure 18 illustrates the different components and services of this decentralized framework.

[9] https://docs.ipfs.io/reference/http/api/#getting-started
[10] https://hub.docker.com/r/ipfs/go-ipfs
[11] https://libp2p.io/

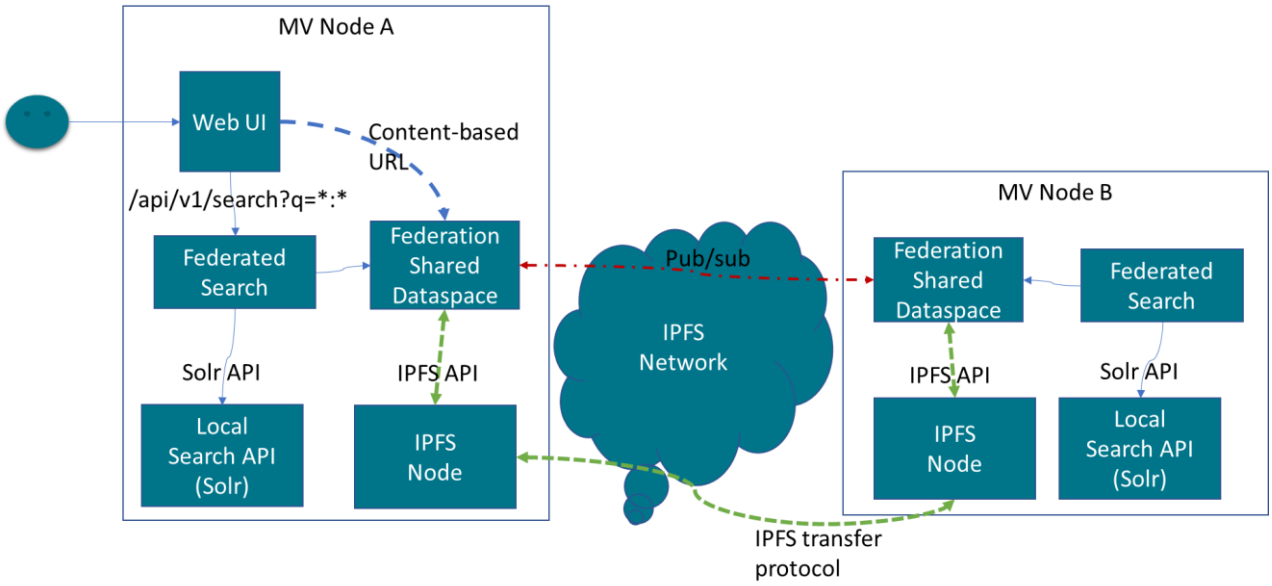

Figure 18: Deployment of the Decentralized framework for communication and content exchange

Figure 19 shows the web interface (swagger UI) of the implemented Federation Shared Dataspace service API.

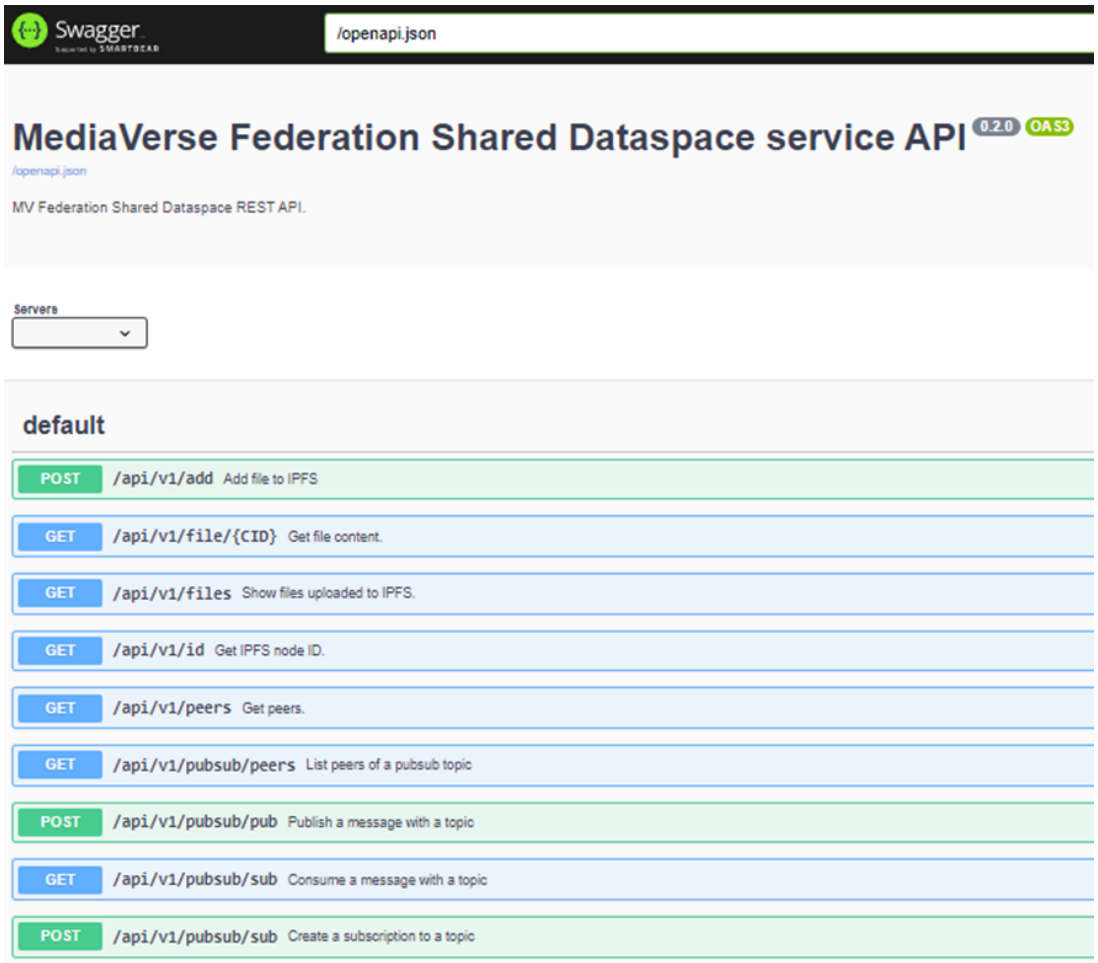

Figure 19: Federation Shared Dataspace service OpenAPI web doc interface

We have implemented a few additional operations in this API to support the publish/subscribe functionalities of IPFS needed by the Federated Search service and the Digital Right Negotiation service (Figure 20).

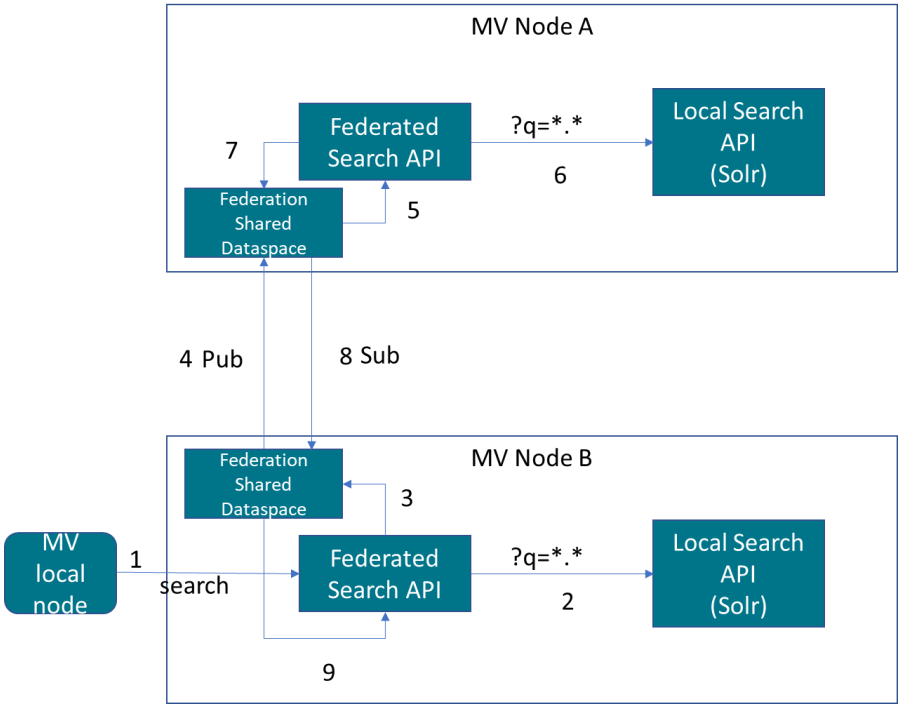

Figure 20: Interdependency among Federated Search service and Federation Shared Dataspace

3.2 Federated description

3.2.1 Functional description

The **Federated Search** service will allow users to search for content in the entire MediaVerse network without the need to know about where and how many nodes are in the network. This service forwards the search queries and aggregates the search results across the network.

Every MediaVerse node implements the indexing, search and retrieval of its local or managed content. This functionality is part of the DAM subsystem and its local search service. The Federated Search service running in each MediaVerse node will query its corresponding local search service or DAM to obtain local/managed partial results of the search from its node and will send the results to the Federated Search service running in another node that sent the original search request.

3.2.2 Technical description

As said before, every node of the MediaVerse network implements an indexing and search service for their locally managed content (content that they manage and/or store in their own infrastructure). This service will allow us to search for local content without asking other nodes in the network. This indexing and search service will be based on Apache Solr[12] engine, NDD index and Metadata Storage.

[12]Apache Solr. https://solr.apache.org

The Federated Search service will allow us to search for content beyond the boundary of a node and search in every node of a distributed network. This service implements the communication protocol between the nodes of the network to query the local search service of each node and collect the search results of each node.

For this first release, we implemented direct Apache Solr API calls[13] from the Federated Search service to query the Apache Solr engine (local search engine) running in the same MediaVerse node. These calls will be executed by the Federated Search service of each node to collect and relay results from its local indexes when other nodes send a global search request. The sequence of actions depicted in figure 21 is the following:

1. Every MediaVerse node has a Federated Search service running and at startup this service starts to subscribe to a generic topic "fsearch" in the pub/sub system of IPFS. This topic, which is common for all the MediaVerse nodes, is the topic to accept search requests from the network.
2. When a MediaVerse node needs to launch a global search, it publishes a message in the private IFPS network of MediaVerse with the query parameters to the topic "fsearch".
3. After publishing the message, the Federated Search service starts to listen to the topic with the unique ID of the node (this ID is sent as part of the search request message, and it is generated by IFPS). This unique topic for each node is the topic to accept search results/responses from the network.
4. The rest of the nodes start to receive search request messages listening the topic "fsearch". They process the messages and resend the requests (after some adaptation) to their local search service or DAM.
5. When the local search service responds to the Federated Search service of its node with the partial result, a message is published in the pub/sub system with the partial results in JSON format and the unique ID of the origin request node as the topic.
6. The Federated Search service of the origin node aggregates all the partial results in an asynchronous process (iterative and incremental) to show the result in the Web UI of the MediaVerse node.

The query syntax for this Federated Search service (exposed by the API) will be the same used by Apache Solr in this fist release. The Federated Search service will act as a proxy to propagate the remote query (using an easy internal translation logic) to its local search service (based on Apache Solr) and collect the local partial results to send back to the node origin of the global query.

This Federated Search service is a core service of any MediaVerse node, and it will be exposed to other services within the MediaVerse node (local or backend network) via an HTTP REST API. This service must send a query to every node active in the network when the query is created. To know which nodes are active in the network we could use the Node Discovery service based on the DHT of IFPS and send the query to every active node in the network, but this approach is very inefficient (we would need to establish a connection with every node of the network, manage timeouts, retries, etc.). To implement this communication fan-out paradigm in an efficient way we are going to use a publish/subscribe implementation available in IPFS network protocols (libp2p). The pub/sub pattern decouples producers (query node) and consumers (local search services) and has a better scalability and performance. Besides, peers can join and leave the network anytime without affecting other subscribers.

[13] https://solr.apache.org/guide/8_11/json-request-api.html

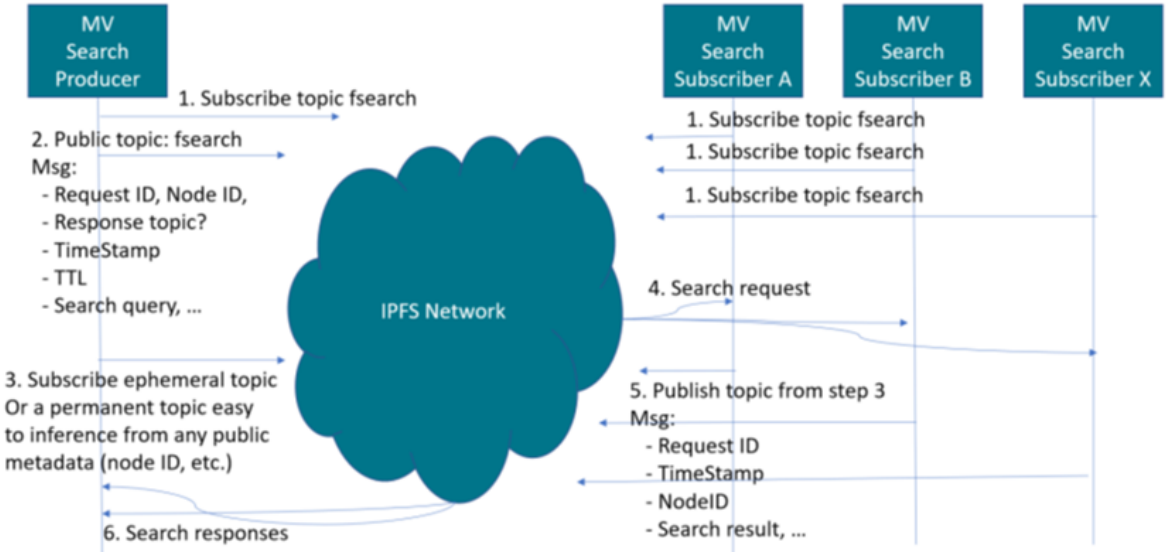

Figure 21: Federated Search pub/sub inter-node communication

The implementation of pub/sub pattern in IPFS creates a virtual network of peers inside the federated network to connect publishers and subscribers based on topics or subjects.[14] We have to define a couple of topics or subjects of messages in order to address the publishing of messages in the publish/subscribe virtual network. The main topic will be "fsearch". Every node of the network will create a subscriber or listener on this topic to receive any query search from any node of the network to process the query request in its own local search service. We also need a different topic to send back the search result of every node of the network to the original publisher of the query. This topic will be the peer or node ID of the publisher node. Besides, we need a request ID to separate the search requests/responses among users of a node (we have multitenant MediaVerse nodes).

Both the query request and the results aggregation will be processed asynchronously. From the publisher point of view, the Web UI of the MediaVerse node has to send the request via a REST API call to the Federated Search service of the DAM and receives a request ID (Claim-Check pattern[15]) with the promise of results. Following this call, the UI will start a polling loop calling another REST API operation of the Federated Search service to update the result of the search query. The collection of results of any query will be incremental and iterative because the response time of all the nodes of the federated network will depend on many external factors like network latency, bandwidth, processing capacity, current load of the node, local storage size and IOs, concurrent users, etc. From the point of view of the subscriber, concurrent queries will be received from any node of the network and some queries could come from the same node because they were sent by different users of the node (multitenant services). We implemented an internal queue system to store pending queries to process in each subscriber node. A thread in the subscriber node will loop to check new requests in its internal queue.

[14] https://docs.libp2p.io/concepts/publish-subscribe/
[15] https://docs.microsoft.com/en-us/azure/architecture/patterns/claim-check

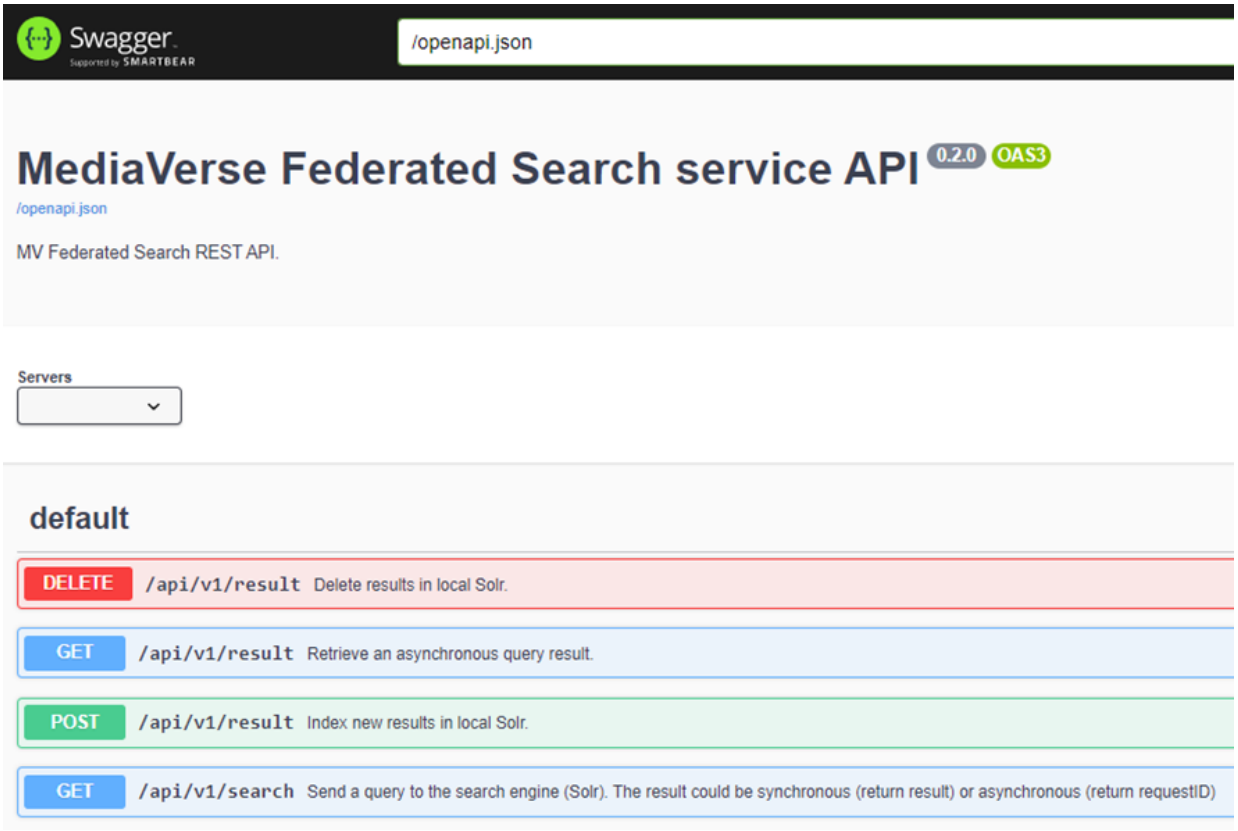

Figure 22: Federated Search service OpenAPI web doc interface

Advanced features for Federated Search service

We plan to incorporate new features like facets, filtering, paging, and sorting to process the search results. To this end, we assess Apache Solr near real-time indexing capabilities to store and index partial results collected from all the nodes and query to the Apache Solr engine[16] of the local node for faceted search and filtering.

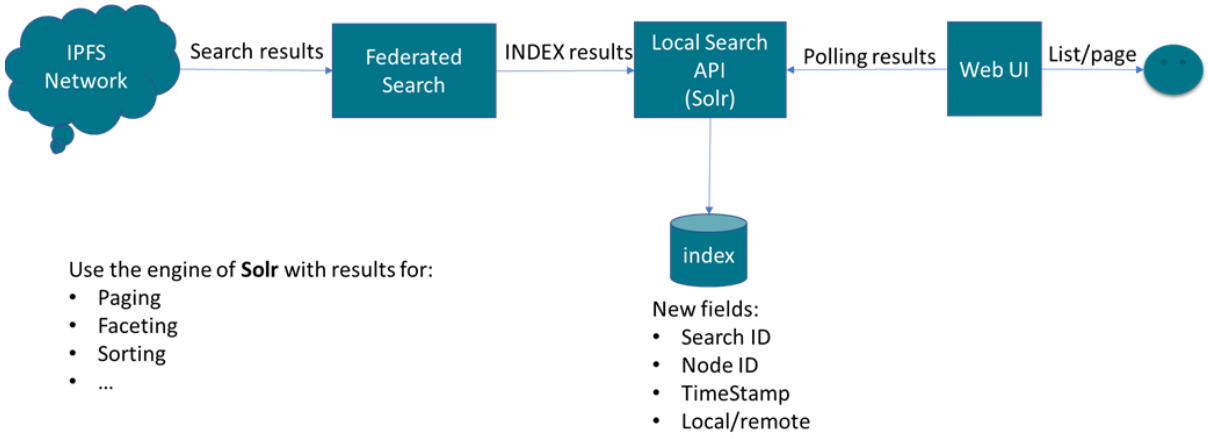

Figure 23: Search results post-processing advanced features proposal

[16] https://solr.apache.org/features.html

Source Code, packing and distribution

The code developed for this framework in this project will be released under open-source licenses (Apache-2.0). The official source code repository is currently maintained in a private repository and is available upon request.

The distribution and deployment of this first release of this framework will be based on Docker container images and compose files but we plan to upgrade to Kubernetes deployments in future releases. We made docker images for the two microservices named above and provide instructions in the source code repository of the project to download and configure an IPFS golang node (packed in an official docker image in Docker Hub[17]) in a private network. Every node of the MediaVerse network will need to install and run this IPFS golang node container and configure the private network to allow the inter-node communication of MediaVerse nodes in a federated network. We also supply a script to do this automatically (deploy.sh).

3.2.3 Installation and usage

Pre-Requisites

- Ubuntu 18.04.4 LTS
- Docker 20.10.9
- Docker-Compose 1.29.2
- Apache Maven 3.6.0
- Jq 1.5-1
- Dos2Unix 7.3.4
- User with 'sudoer' permissions

Deployment

1. The first time we are going to deploy the containers we will use the 'deploy.sh' script. We will need to give execution permissions: chmod u+x deploy.sh
2. To ensure all scripts use UNIX format we will run on the 'deployment' folder: sudo find . -type f -exec dos2unix {} \;
3. Execute the script: ./deploy.sh
    a. The 'deploy.sh' script will deploy an IPFS node container and add it to the private network, as well as generate the python-flask based APIs source code and generate its respective containers.
    b. If you need to change the IPFS HOST to your own, edit the ./deployment/mv_ipfs_api/runapi.sh docker run environment variable before the deployment: IPFS_HOST=YOUR_OWN_IP
    c. Temporarily, the SOLR Search API uses an environment definition file (./deployment/mv_search_api/openapi_server/openapi_server/controllers/.env) in which are defined the SOLR and IPFS hosts and used ports. You can edit this file to add your own hosts.
4. After the first deployment, one can redeploy the containers using the docker-compose definition file: docker-compose up -d

[17] https://hub.docker.com/r/ipfs/go-ipfs/

3.2.4 Usage

Federated Search service API

Once deployed, the Federated Search service API will be published on the selected IP using port 5050. The search service of the API will accept HTTP GET requests (used to request data from a specified source). The "q" parameter is used by the Local Search service (direct SOLR API calls for this first prototype) to filter search requests based on the 'key:value' model (E.g. q=username:admin). The two most common ways of sending an HTTP request to the API are by browsing the URL in a web browser, by using the appropriate language specific library, or by using the CURL tool as shown below:

curl http://:5050/api/v1/search?q=*:*

http://MY-IP-ADDRESS:5050/api/v1/search?q=*:*

- Accepted HTTP request: GET
- q=*:* is the SOLR standard and mandatory query parameter to filter search requests.
- Try it (CLI): curl http://<my-ip-address>:5050/api/v1/search?q=*:*

Federation Shared Dataspace API

Similar to the SOLR Search API, the Federation Shared Dataspace API will be published on the selected IP but using port 4040. The Federation Shared Dataspace API is also configured to be communicated via HTTP requests. When accessing the '/ui/' endpoint any user will be able to check and try every endpoint with specific HTTP methods indicated in each endpoint.

http://MY-IP-ADDRESS:4040/ui/

- Access to the Federation Shared Dataspace API definition. Here you can check and try every endpoint with accepted HTTP methods.
- Try it (CLI): curl http://<my-ip-address>:4040/ui/

3.3 Digital Rights Negotiation

The **Digital Rights Negotiation** service will be responsible for facilitating the communication among the different Digital Asset Rights Management components of MediaVerse nodes in the federated network for digital asset rights negotiation. The NDD (Near Duplicate Detection) service, responsible for the identification of duplicates each time a new asset is ingested in the federated network, will use this communication framework for the synchronization among NDDs of the nodes using a different topic (following pub/sub integration pattern).

This will be implemented in the next release of the MediaVerse decentralized framework for communication.

4 Conclusion and Next Steps

At the end of this first period of WP6, progress can be observed in the implementation of the different modules contained in this work package, including the node main services and components.

The future steps will be:

1. Design and implementation of the Digital Rights Negotiation service to allow the communication among the Digital Asset Rights Management services of MediaVerse nodes inside the federated network. To this end, also extend the Federated Search service to allow searches with the NDD service.
2. Federated Search service: Implement facets, filters and sorting of search results based on Apache Solr engine. More testing of federated search in a multi-node environment.
3. Improve the integration with the rest of MediaVerse core components.
4. Configuration of the CI/CD pipeline for the services of this decentralized framework. Moving the deployment distribution to a Kubernetes configuration.
5. Keep improving and adding new functionality to the Dashboard UI by
    a. Increasing the number of supported content types, adding audio and 3D content support.
    b. Keep developing new functionality described in the User Requirements in D2.1 - Use Cases and User Requirements[18] and marked to be in the version 1.0, adding it to the dashboard new requirements, like a new section for administrators, a more complete profile section, the possibility of creating projects, more user settings, etc.
    c. Keep focusing on the accessibility to comply with every aspect of the WCAG2.2, level AA.

[18] https://mediaverse-project.eu/wp-content/uploads/2021/04/D2.1-V1.0.pdf

MediaVerse is an H2020 Innovation Project co-financed by the EC under Grant Agreement ID: 957252.