Ref. Ares(2020)2524396 - 13/05/2020

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement n: 761999

EasyTV: Easing the access of Europeans with disabilities to converging media and content.

D5.7 Final report on the set up and implementation of the EasyTV multi terminal technical platform

EasyTV Project

H2020. ICT-19-2017 Media and content convergence. – IA Innovation action.

Grant Agreement n°: 761999

Start date of project: 1 Oct. 2017

Duration: 33 months

Document. ref.: D5.7

POLITÉCNICA ENGINEERING CERTH mediavoice UAB Universitat Autònoma de Barcelona Corporació Catalana de Mitjans Audiovisuals, SA arx.net CNSE Fundación

Disclaimer

This document contains material, which is the copyright of certain EasyTV contractors, and may not be reproduced or copied without permission. All EasyTV consortium partners have agreed to the full publication of this document. The commercial use of any information contained in this document may require a license from the proprietor of that information. The reproduction of this document or of parts of it requires an agreement with the proprietor of that information. The document must be referenced if is used in a publication.

The EasyTV Consortium consists of the following partners:

	Partner Name	Short name	Country
1	Universidad Politécnica de Madrid	UPM	ES
2	Engineering Ingegneria Informatica S.P.A.	ENG	IT
3	Centre for Research and Technology Hellas/Information Technologies Institute	CERTH	GR
4	Mediavoice SRL	MV	IT
5	Universitat Autònoma Barcelona	UAB	ES
6	Corporació Catalana de Mitjans Audiovisuals SA	CCMA	ES
7	ARX.NET SA	ARX	GR
8	Fundación Confederación Nacional Sordos España para la supresión de barreras de comunicación	FCNSE	ES
9	Unione Italiana dei ciechi e degli ipovedenti	UICI	IT

PROGRAMME NAME:	H2020. ICT-19-2017 Media and Content Convergence – IA Innovation Action
PROJECT NUMBER:	761999
PROJECT TITLE:	EASYTV
RESPONSIBLE UNIT:	ENG
INVOLVED UNITS:	ENG, CERTH, UPM, ARX, MV, CCMA
DOCUMENT NUMBER:	D5.7
DOCUMENT TITLE:	Final report on the set up and implementation of the EasyTV multi terminal technical platform
WORK-PACKAGE:	WP 5
DELIVERABLE TYPE:	Report
CONTRACTUAL DATE OF DELIVERY:	29-02-2020
LAST UPDATE:	29-02-2020
DISTRIBUTION LEVEL:	PU

Distribution level:

PU = *Public,*

RE = *Restricted to a group of the specified Consortium,*

PP = *Restricted to other program participants (including Commission Services),*

CO= *Confidential, only for members of the LASIE Consortium (including the Commission Services)*

Document History

VERSION	DATE	STATUS	AUTHORS, REVIEWER	DESCRIPTION
v. 0.1	03/02/2020	Draft	Giuseppe Vitolo (ENG)	Table of Contents definition and document structure
v. 0.2	07/02/2020	Draft	Giuseppe Vitolo (ENG)	Introduction and infrastructure
v. 0.3	14/02/2020	Draft	Giuseppe Vitolo (ENG)	Containerization Platform
v. 0.4	19/02/2020	Draft	CCMA, CERTH, Giuseppe Vitolo (ENG)	Content update, added Service Manager chapter
v. 0.5	26/02/2020	Draft	Stavros Skourtis (ARX), Giuseppe Vitolo (ENG)	Addex Appendix A – Docker Compose files, Added Appendix B – Service Manager API Specification
v. 0.6	28/02/2020	Review	Stavros Skourtis (ARX), Thanassis Kalvourtzis (CERTH)	Review
v. 1.0	29/02/2020	Final	Giuseppe Vitolo (ENG)	Final version

Definitions, Acronyms and Abbreviations

ACRONYMS / ABBREVIATIONS	DESCRIPTION
API	Application Programming Interface
CLI	Command Line Interface
GUI	Graphical User Interface
HTML	HyperText Markup Language
IDE	Integrated Development Environment
IP	Internet Protocol
IT	Information Technology
JSON	Javascript Object Notation
OS	Operative System
REST	Representational State Transfer
SQL	Structured Query Language
TCP	Transport Control Protocol
UDP	User Datagram Protocol
URI	Uniform Resource Indicator
URL	Uniform Resource Locator
vCPU	Virtual Central Processing Unit
VM	Virtual Machine
vRAM	Virtual Random Access Memory
YAML	YAML Ain’t a Markup Language

Table of Contents

1. Introduction **11**
2. Infrastructure **12**
2.1. Overview 12
2.2. Virtual machine 13
2.3. Storage 13
2.4. Networking 14
3. Containerization platform **15**
3.1. Overview 15
3.2. Management UI 16
3.2.1. User management 17
3.3. Module containerization 17
3.4. Networking & virtual networking 18
3.5. Data management and volume configuration 19
3.6. Container orchestration 20
4. Service Manager **21**
4.1. introduction 21
4.2. Components 21
4.3. Service integration configuration 21
4.3.1. Obtaining an API key 21
4.4. Content Owner integration configuration 22
4.5. Multiple module integration workflow 22
5. Conclusions **23**
6. References **24**
Appendix A - EasyTV Services Docker Compose definition **25**
1. Catalogue_stack **25**
2. Cp_stack **26**
3. Hp_stack **28**
4. Hplogin_stack **30**
5. Registry **32**
6. Sm_stack **33**
7. Spm_stack **36**
APPENDIX B - Service Manager API specification **38**
1. Introduction **38**
1.1. Terminology 38
1.2. JSON 38
1.3. API conventions 38

2. Integration with the EasyTV services....................40
2.1. Configuring the integration of a service....................40
2.1.1. Obtaining an API key....................40
2.2. Internal API....................40
2.2.1. Task Configuration Management....................40
2.2.2. Service Manager to EasyTV Service communication....................43
2.2.3. Get a list of all active jobs....................46
2.2.4. Get information about a certain job....................48
2.2.5. Set the status of a Job....................49
2.2.6. Cancel a Job....................50
2.2.7. Upload asset....................51
2.2.8. Get assets....................51
2.2.9. Finishing an active Job....................52
2.2.10. Get tasks for the current service....................53
3. Integration with the Content Owner....................55
3.1. Authentication....................55
3.1.1. Registration....................55
3.1.2. Login....................55
3.1.3. Ping....................56
3.1.4. Logout....................56
3.1.5. Change password....................57
3.2. Information Retrieval....................58
3.2.1. Get the list of registered services....................58
3.2.2. Get information about a Service and its tasks....................60
3.3. Job Management....................62
3.3.1. Create a new job....................62
3.3.2. Cancel job....................65
3.3.3. Get all jobs....................65
3.3.4. Get information about a specific job....................68

List of Figures

Figure 1 - VMWare virtualization solution 12
Figure 2 - Logical volume creation process 13
Figure 3 - Network setup 18
Figure 4 - Types of mount bind 19
Figure 5 - Multi-language Subtitle translation end-to-end workflow 22

List of Tables

Table 1 - Server configuration 13
Table 2- EasyTV Docker stacks 17
Table 3 - Volumes configuration 20
Table 4 – Service Manager Terminology........ 21

Executive Summary

This document is the deliverable D5.7 in WP5. It is the final report on the set up and implementation of the EasyTV multi terminal technical platform. This report focuses on the technologies used and the processes adopted in order to make the platform available and ready for use.

The first chapter describes the hardware and operative system infrastructure which lays the foundation for the entire containerization platform.

The second chapter describes the implementation of the containerization platform. In addition to the description of the enabling underlying technologies, configuration and processes used, a list of containerized EasyTV services is given.

The third chapter introduces the EasyTV Service Manager component: its purpose and main features are outlined, and its current configuration is described.

Finally, conclusions are presented in the fourth chapter.

In addition, two appendices have been added: appendix A lists the Docker Compose files used to build and deploy the EasyTV services running on the platform, while appendix B (named “Service Manager API specification”) describes in detail the EasyTV Service Manager API.

1. INTRODUCTION

The **EasyTV multi-terminal technical platform** is the platform that hosts and interconnects the available **EasyTV services** and takes care of their availability and reliability. By employing **containerization** and **container orchestration** technologies, the EasyTV multi-terminal technical platform provides service scaling and full lifecycle management both in cloud and in “on-premise” environments.

Docker, which is the leading containerization engine currently available on the market, is the main technological enabler which sits at the core of the EasyTV multi-terminal technical platform. Both Docker containerization engine and **Docker Swarm** orchestrator have been used in conjunction to allow deployment, execution and orchestration of multiple service application containers. Administrator-level fine-grained control is granted through a management user interface that gives an authorised user enough privileges to operate and manage several platform aspects, including (but not limited to) image loading, configuration of containers/services/stacks, virtual data volumes and networks configuration, user management.

All the EasyTV services that it was possible to package and containerize have been deployed and are currently in execution on the platform, readily accessible to use. The introduction of a specific software component called **EasyTV Service Manager** enables the design of potentially complex **workflows** involving multiple interacting services.

Integration with the **EasyTV Service Registry** (described in D5.8), which hosts the available service **images**, enables developers and administrators to start a new service on the platform by referencing an existing image hosted by the Service Registry itself. On the other hand, each service can be exposed and made reachable through the Internet, so it can be registered in the **EasyTV Service Catalogue** (this component is described in D5.8 too) and finally made available to any professional user who can choose the most suitable service for her/his needs.

2. INFRASTRUCTURE

2.1. Overview

The EasyTV multi-terminal technical platform is hosted at ENG premises, specifically in the main ENG data center located in Pont Saint Martin (Italy). The facility has been designed to support four levels of physical security to give access to the main data center, which is further divided in autonomous and isolated bunkers. The data center has acquired a certain number of certifications, namely:

- ISO 27001
- ISO 9001
- ISO 20000

The main technological features worth mentioning are as follows:

- Architecture based on latest generation Intel Blade servers, specifically optimised for virtualised environments.
- Vertical (in terms of single node capacity) and horizontal (in terms of number of nodes available) scalability.
- Hardware support with "Next Business Day" Service Level Agreement
- RAID 5 plus hot-spare disk configuration
- VMWare virtualization technology (vSphere virtualization platform)

VMWare vSphere manages the whole data center as a unified operating environment. The two core components of vSphere are ESXi and **vCenter Server**: ESXi is the virtualization platform where you create and run virtual machines and virtual appliances. vCenter Server is the service through which you manage multiple hosts connected in a network and pool host resources.

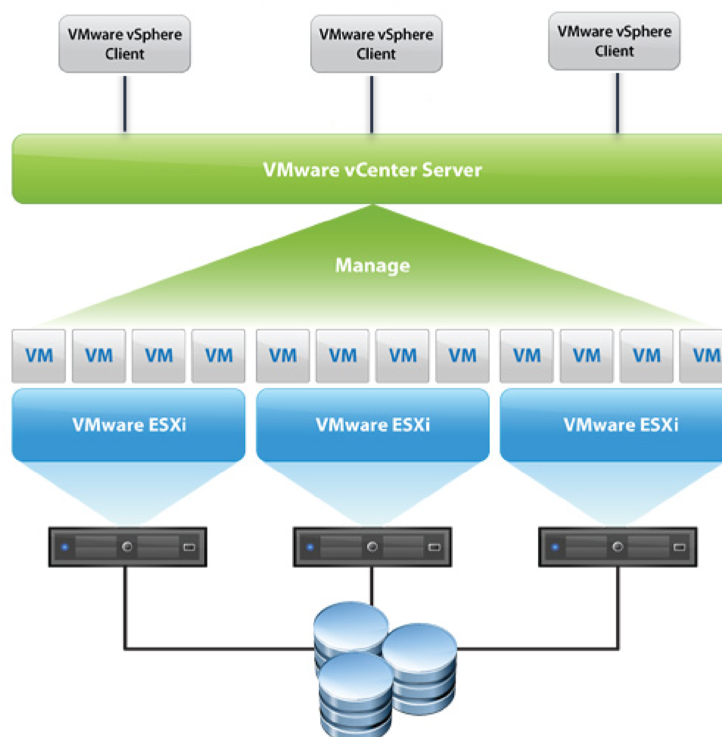


Figure 1 - VMWare virtualization solution[1]

[1] https://docs.vmware.com/en/VMware-vSphere/index.html

2.2. Virtual machine

A **virtual machine**, which hosts the whole **containerization stack solution** described in the next chapter, has been set up and activated. The following table, which has also been presented in D5.1, explicitly shows the main VM characteristics:

No. of vCPU	vRAM (GB)	Storage (GB)	OS
4	8	100	CentOS Linux 7.6

Table 1 - Server configuration

The VM has a set of virtual devices which offer the same capabilities as their physical counterpart, but with added benefits such as portability, safety and easiness of management. In addition to the common operations that is possible to perform on a physical machine, the VM supports operations in the virtual infrastructure: it is possible, for example, to generate a **snapshot** of the VM state at any given point in time and to restore it later as needed.

The running OS is CentOS Linux 7.6, which is a rolling-release (updates are released continuously) Linux distribution that "tracks just ahead of Red Hat Enterprise Linux (RHEL) development, positioned as a midstream between Fedora Linux and RHEL" [1].

2.3. Storage

A **Logical Volume Manager**, which employs the Linux kernel's **device mapper** to provide a set of partitions independent of underlying disk layout, has been used to abstract storage and have "virtual" partitions instead of physical ones. An additional **logical volume** named "lv_dati", specifically targeted to be used by the containerized EasyTV services, has been added to the VM, thus easing data management operations like dynamic resizing, data migration between physical devices within the pool on the running system and data export.

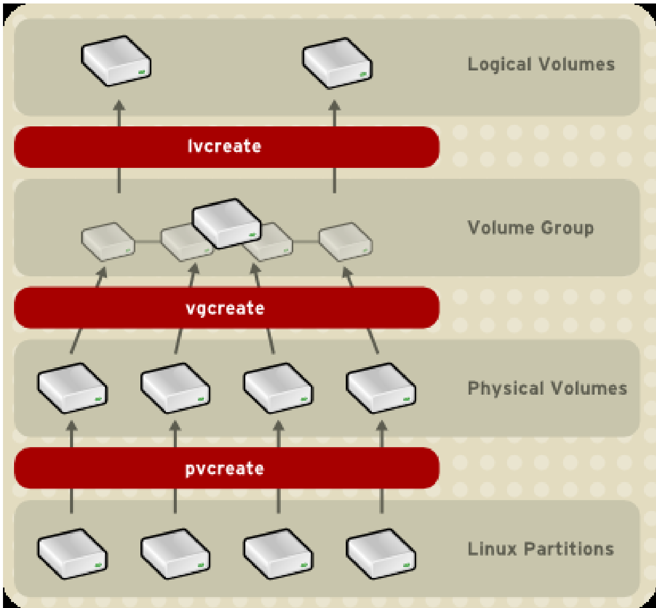


Figure 2 - Logical volume creation process[2]

[2] https://access.redhat.com/documentation/it-it/red_hat_enterprise_linux/5/html/cluster_suite_overview/s1-clvm-overview-cso

2.4. Networking

In order to enhance platform security, remote VM access through SSH is only allowed by connecting to a secure ENG **virtual private network**: an administrator who wishes to perform some kind of task shall be authorised, first of all, to connect to the ENG VPN. Once connected, the administrator can use the "ssh" command with the appropriate parameters (namely username/password and remote IP address) and start working on the remote console. It is worth noting that aside from the "root" user added during the VM configuration phase, another user named "easytv" has been added to the system, who has been given enough permissions to fully manage the containerization engine and the container orchestrator involved in the execution of the available EasyTV services.

The VM can be reached from the Internet through a **reverse proxy** (based on Apache *httpd* server) which has been configured to allow access to the individual EasyTV services: thus, each service may be used via a registered third-level domain (**.easytv.eng.it*). Data integrity and confidentiality during network communication is enforced by means of the HTTPS protocol. SSL/TLS certificates are supplied by a CA, in this case "Let's Encrypt", which offers domain validated certificates that are verified through the associated domain name.

In addition to the abovementioned reverse proxy, another security layer consists of the Linux system firewall named **firewalld**, which supports **zones** that define the trust level of network connections or interfaces. The firewalld daemon has been configured through the "firewall-cmd" tool from the command line to make a set of TCP ports available to the outside world ("public" zone), specifically all the ports ranging from 8080 to 8089, plus two other ports (9000 and 5000). These ports are used by the containerized EasyTV services in order to communicate with external service consumers.

An SMTP **mail relay** server is also available inside the virtual LAN and can be readily used by any service that needs to send emails, for example confirmation emails to successfully complete a registration process.

3. CONTAINERIZATION PLATFORM

3.1. Overview

The containerization platform which hosts the EasyTV services packaged in the form of software images is based on **Docker Engine** (container execution) and **Docker Swarm** (container orchestration). The **swarm manager** runs on the VM described in the previous chapter and the whole platform supports scaling over several nodes in order to balance the swarm's load.

A brief explanation of the most common terms related to Docker is given here to facilitate the understanding of the next paragraphs and sections. Specifically, it is useful to have a basic grasp of the following terms (the major part of the following definitions have been gathered from [2]):

- **Image** – an Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. An image typically contains a union of layered filesystems stacked on top of each other. An image does not have state and it never changes.
- **Dockerfile** - A Dockerfile is a text document that contains all the commands normally executed manually in order to build a Docker image.
- **Layer** – in an image, a layer is modification to the image, represented by an instruction in the Dockerfile. Layers are applied in sequence to the base image to create the final image. When an image is updated or rebuilt, only layers that change need to be updated, and unchanged layers are cached locally.
- **Container** – a container is a runtime instance of a Docker image. It consists of a docker image, a running environment and a standard set of instructions.
- **Service** - A service is the definition of how you want to run your application containers in a swarm. At the most basic level a service defines which container image to run in the swarm and which commands to run in the container. For orchestration purposes, the service defines the “desired state”, meaning how many containers to run as tasks and constraints for deploying the containers.
- **Stack** – A stack is a group of interrelated services that share dependencies, and can be orchestrated and scaled together. A single stack is capable of defining and coordinating the functionality of an entire application, though very complex applications may want to use multiple stacks.
- **Swarm** - A swarm is a cluster of one or more Docker Engines running in “swarm mode”.

The containerization technique employed provides an additional layer of abstraction, effectively isolating the running services from the underlying environment. This, in turn, allows an efficient use of the available resources and leads to an increase in security. As explained in D5.1, system containers are fundamentally different from virtual machines: system containers actually offer an environment as close as possible to an environment provided by a VM, but without the overhead that comes with running a separate kernel and simulating all the hardware.

The software packages required have been installed through the standard “yum” **package manager** available in the CentOS Linux system distribution. The steps followed to install Docker can be found in [3]. After the installation phase has been completed, some tweaks have been made in order to improve platform operation.

First of all, *dockerd* daemon has been configured to start on boot: this is useful in a certain number of situations, for example in case of sudden system crash and machine restart. CentOS uses *system* to manage which services start when the system boots. The following command enables Docker auto-start at system boot:

```
$ sudo systemctl enable docker
```

In order to ease the management of the Docker-based platform, it has been useful to grant a particular user sufficient permissions to use the *docker* command as a non-root user. So, instead of

forcing the user to use the *sudo* command to perform an admin-level operation, a new group called "docker" has been created and the user "easytv" has been added to it.

```
$ sudo groupadd docker
$ sudo usermod –aG docker easytv
```

It has been convenient to also change the default Docker data root directory and take advantage of a logical volume specifically created to host Docker images, container data, virtual storage and configuration files. To change the default location, the configuration file located in /etc/docker/daemon.json has been modified by adding the following line:

```
{
[...]
  "data-root": "/dati/docker",
[...]
}
```

Finally, a service restart had to take place for the changes to take effect:

```
$ sudo systemctl daemon-reload
$ sudo systemctl restart docker
```

After these steps the software stack has been properly configured and it is ready to manage the lifecycle of the available EasyTV services, which have to be packaged prior to their upload and execution on the containerization platform.

3.2. Management UI

A management user interface has been deployed, thus allowing an administrator, a developer, a group of administrators or a group of developers to easily build, manage and maintain the deployed Docker environment. Portainer, specifically Portainer Community Edition [4], has been chosen as it represents one of the best products of its kind, with over 2 billion downloads throughout his history [5].

Since Portainer itself runs as a containerized application, its deployment has been straightforward by using the following commands:

```
$ docker volume create portainer_data
$ docker run -d -p 8000:8000 -p 9000:9000 --name=portainer --restart=always -v
/var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer
```

In addition, the reverse proxy that shields the virtual machine has been configured to allow connections to the Portainer application by using the following URL: https://portainer.easytv.eng.it. Further details regarding reverse proxy configuration are given in a later section.

An authorized Portainer user can:

- See a list of all the endpoints defined within Portainer and search by tag/keyword/ip/name and select a specific endpoint;
- View information about a particular endpoint;
- Have an overview of images, containers, services, stacks, networks and volumes;
- Upload and remove a particular docker image;
- Create, launch, stop or remove a container or a group of containers;
- Create, launch, stop or remove a service or a group of services;
- Create, launch stop or remove a stack or a group of stacks;

- Manage virtual networks
- Manage virtual volumes

3.2.1. User management

Portainer user management features allow an administrator to manage users and groups (called “teams”) in order to assign specific permissions and capabilities to the registered users.

First of all, a team named “development” has been created and any user that has been subsequently created belongs to this team. Later, a user for each partner has been added so that each of them could upload or delete their software images and could start, change or stop their own EasyTV service stacks. The list of users together with their role is as follows:

- admin (role “administrator”)
- arx (role “team leader”)
- ccma (role “user”)
- certh (role “user”)
- eng (role “team leader”)
- mv (role “user”)
- upm (role “user”)

Each user is in complete control of its own software stacks, but **restriction** on stack operations can also be changed by allowing other users in the same group (namely the “development” group) to perform modifications.

3.3. Module containerization

Each EasyTV service which runs on the containerization platform has been converted into a package (a Docker image). Each image has been uploaded to the platform and then used in the appropriate Docker stacks, which are a group of interrelated services that share dependencies (common services or common resources).

The currently deployed EasyTV service stacks are as follows:

Stack name	Description
catalogue_stack	Service Catalogue
cp_stack	Crowdsourcing Platform
hp_stack	Hyper-personalisation module
hplogin_stack	Hyper-personalisation login module
Registry	Service Registry
sm_stack	Service Manager
spm_stack	Subtitle Production Module

Table 2- EasyTV Docker stacks

Each stack has been deployed by creating an appropriate Docker **compose** file, which is a particular file that defines services, networks and volumes for each stack, along with specific configuration options such as number of container replicas and container restarting/rescheduling policy. Compose files are created and edited by using the Portainer Compose Editor, which is a user-friendly web editor available in Portainer. Docker Compose files are listed in Appendix A.

Note, however, that due to technical limitations it was not possible to containerize all the EasyTV services developed. For example, a service developed by MV is based on specific .NET framework libraries that it was not possible to port on a Linux VM and thus it is running on an external server, and services developed by UPM that need heavy GPU-based computation are installed on an external server where it is possible to take advantage of GPUs in computation. Nonetheless, these services are conceptually and architecturally part of the platform, and they could be containerized on a commercial-ready infrastructure by satisfying the specific services requirements.

3.4. Networking & virtual networking

Docker containers can mutually communicate over a “virtual network”. Docker allows to create and configure several virtual networks based on a pluggable driver architecture. Several drivers exist by default and provide core network functionalities: bridge, host, overlay, macvlan, none. Further information can be found at [6].

A virtual network named “easytv_net” has been created on the platform to enable communication among different EasyTV service containers. The “overlay” driver has been used: overlay networks are distributed network that can work among multiple Docker daemon hosts by sitting on top of the host-specific networks, so that containers can communicate securely inside a swarm.

The VM sits inside a VLAN and it is not accessible from the outside world. To expose the services which are running on it (and thus on the Docker Swarm platform) it is mandatory to configure a **reverse proxy** which filters and routes traffic from the Internet to the internal VLAN nodes. The reverse proxy is based on Apache.

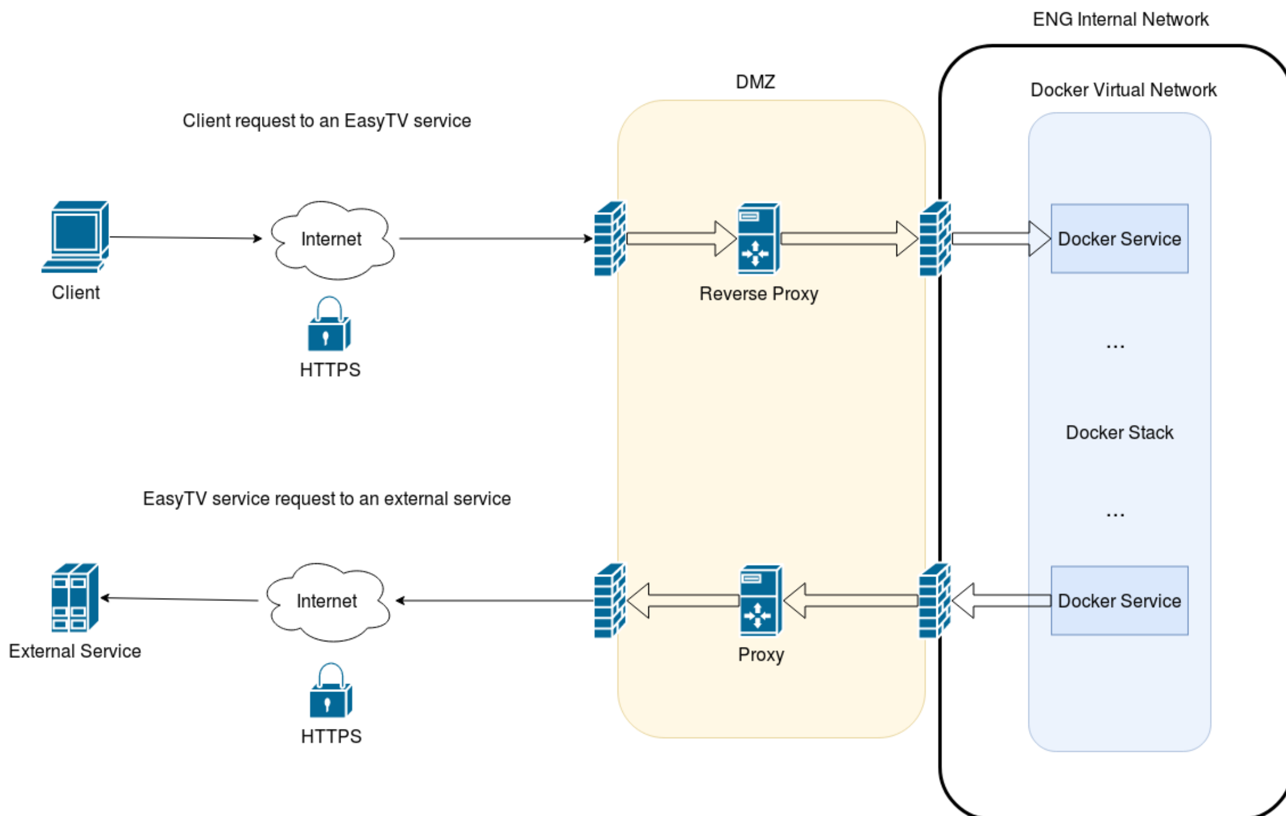


Figure 3 - Network setup

Some EasyTV services may also need to start communication with an external service provider rather than responding to a request coming from the outside. A **proxy** server has been configured to handle this kind of situation. Both the proxy and the reverse proxy employ an **SSL termination** mechanism, so that traffic is encrypted between an external client/service and the proxy/reverse

proxy but it gets decrypted for communication inside the virtual private network. This kind of mechanism is useful to reduce the load on the main server by shifting the cryptographic computation towards the proxy/reverse proxy.

Docker Compose files, which belong to each running EasyTV service and are editable through Portainer Compose Editor, include an option to configure networks for each service stack. For each service in a single Compose file a “networks” option can be used to explicitly state which networks have to be used:

```
networks:
  easytv_net:
    external: true
```

The previous parameters have been entered in all the Compose files related to the available services.

3.5. Data management and volume configuration

Each available EasyTV service running as a container owns a portion of disk space on the VM virtual hard drive. **Bind mounts** have been configured in order for each service to have access to a directory that resides on the host machine. When a bind mount is used, a file or directory on the host machine is mounted inside a container and the file or directory is referenced by its full or relative path on the host machine.

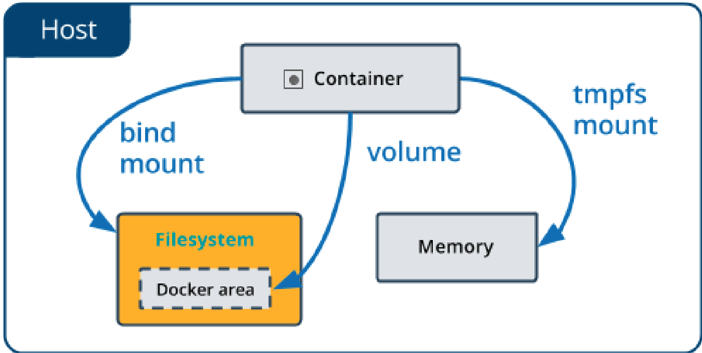


Figure 4 - Types of mount bind

The directories associated with each service reside inside the host directory named “/dati”, which itself resides on a **logical volume** named “lv_dati”. Bind mounts simplify access to service data from outside a container, which in turn facilitate backup and restore operations.

Docker Compose files, which belong to each running EasyTV service and are editable through Portainer Compose Editor, include an option to configure volumes for each service stack. For each service in a single Compose file a “volumes” option can be used to explicitly state which volumes have to be mounted inside the service containers.

The following is a table of stacks and associated Docker services which have their own volume configuration (stacks which do not employ any form of volume storage have been omitted):

Stack name	Service name	Volumes configuration
catalogue_stack	catalogue-mongo	- /dati/catalogue/db:/data/db
cp_stack	crowdsourcing_app	- /dati/docker/volumes/cp-vol/_data - /dati/cp/assets/mocap-submits:/path/inside/container/mocap-submits

		- /dati/cp/assets/video-submits:/path/inside/container/video-submits
cp_stack	crowdsourcing_mongo	- /dati/cp/db:/data/db
hplogin_stack	personalization_db	- /dati/hp/db:/var/lib/mysql
Registry	registry_registry	- /dati/registry:/var/lib/registry
sm_stack	service_manager_cron	- /dati/sm/logs/cron:/var/log/sm
sm_stack	service_manager_api	- /dati/sm/assets:/asset/ - /dati/sm/logs/api:/var/log/sm
sm_stack	service_manager_db	- /dati/sm/dbdata:/var/lib/postgresql/data
spm_stack	spm_api	- /dati/spm/db:/local-data

Table 3 - Volumes configuration

3.6. Container orchestration

Quoting what has been reported in D5.1, "container orchestration defines both the initial deployment of the containers and the management of multiple containers as a single entity. Specifically, container orchestration encompasses a certain number of features, including container instantiation, container execution rescheduling, container linking through interfaces, cluster scaling".

Docker "**swarm mode**" is a specific Docker mode which has been used to enable module orchestration features on the EasyTV multi-terminal technical platform by natively managing a cluster of Docker engines called a **swarm**. A single node swarm has been deployed and it could be expanded by means of horizontal scaling by adding more nodes to it.

A given swarm node can act as a **manager** or as a **worker**, or even perform both roles. Multiple managers and workers can operate simultaneously inside a swarm. To enable a single-node swarm on the VM the following command has been used:

```
$ docker swarm init
```

The engine has then switched the current node into swarm mode, designating the current node as a leader manager node and creating a swarm called "default". In order to add worker nodes to this particular swarm it is sufficient to run the following command on another machine that is able to communicate with the existing swarm pool:

```
$ docker swarm join –token <SWARM-TOKEN>
```

where "<SWARM-TOKEN>" is a special unique token printed by the previous "docker swarm init" command that was executed on the manager node.

The "replicas" option inside each stack Compose configuration file controls the number of **replica tasks** for the swarm manager to schedule onto available nodes. In this way, the swarm manager can distribute the given number of tasks on the available nodes based on the desired scale number.

4. SERVICE MANAGER

4.1. introduction

The EasyTV Service Manager, introduced in D5.9, is an asset which provides a way to manage tasks and jobs related to each of the available EasyTV services. A Content Owner can use the Service Manager to interact with a **service** of her/his interest by creating a **job** which is comprised of multiple **tasks** carried out by a specific service.

Word	Refers To
Service	Each EasyTV service/module that is exposed to the Content Owner through the Service Manager
Task	A specification of work that is offered by a **service** in the EasyTV platform
Job	It is created by the Content Owner to complete one or more **tasks** offered by the **services**.

Table 4 – Service Manager Terminology

It provides an **API management mechanism** which replaces Kong [7] API management solution, previously described in D5.1.

Two set of APIs are provided: one for the EasyTV platform and one for the Content Owner. The data interchange format used for communications is JSON. The EasyTV Service Manager management UI is reachable at https://sm.easytv.eng.it, while its REST API is reachable at https://sm-api.easytv.eng.it.

4.2. Components

The Service Manager is running on the EasyTV multi-terminal technical platform as a Docker **stack** consisting of the following Docker services:

- service_manager_cron – jobs scheduling service
- service_manager_api – REST api backend
- service_manager_ui – UI frontend
- service_manager_db – PostgreSQL database
- service_manager_cache – memcached cache service

4.3. Service integration configuration

To achieve a smooth integration with every service of the EasyTV platform as well as new services that might come in the future, the Service Manager will provide a generic way of handling the different services. For this reason, before the communication can start, a service must be first registered to the Service Manager. Further details on the internal API can be found in Appendix B.

4.3.1. Obtaining an API key

The creators of the new service must contact the administrator of the Service Manager in order for the service to be registered and also to get an API key. This key must be passed to the Service Manager in every HTTP request that is made. For this purpose a custom HTTP header will be used as seen in the example bellow.

```
X-EasyTV-Key: <API_KEY>
```

The <API_KEY> is the actual API key that was given to the service. For the internal API the rest of the document assumes that all requests contain this API key in their headers.

4.4. Content Owner integration configuration

Integration process with the Content Owner involves the following sub-processes:

- **Authentication**
- **Information retrival**
- **Job management**

The Service Manager administrator manages and authorizes the Content Owners, which in turn can obtain information on the registered services and the available tasks. A Content Owner can manage its own jobs by creating new jobs, cancelling or stopping them and obtaining information on a specific job.

Further details can be found in Appendix B.

4.5. Multiple module integration workflow

In order to better understand how the EasyTV Service Manager handles the interaction with a module and how a module can communicate with other services, a workflow involving more than one module is presented in this section. Specifically, the workflow presented is the one involving the Service Manager, The Crowdsourcing Platform and the Subtitle Production Module.

Figure 5 shows the information flow in a success scenario (error handling has been omitted for brevity and clarity). The figure shows how a broadcaster (Content Owner) can initiate a request by contacting the Service Manager, which in turn communicates with the Subtitle Production Module in order to start a new translation job. The Subtitle Production Module communicates with the Crowdsourcing Platform in order to complete the job.

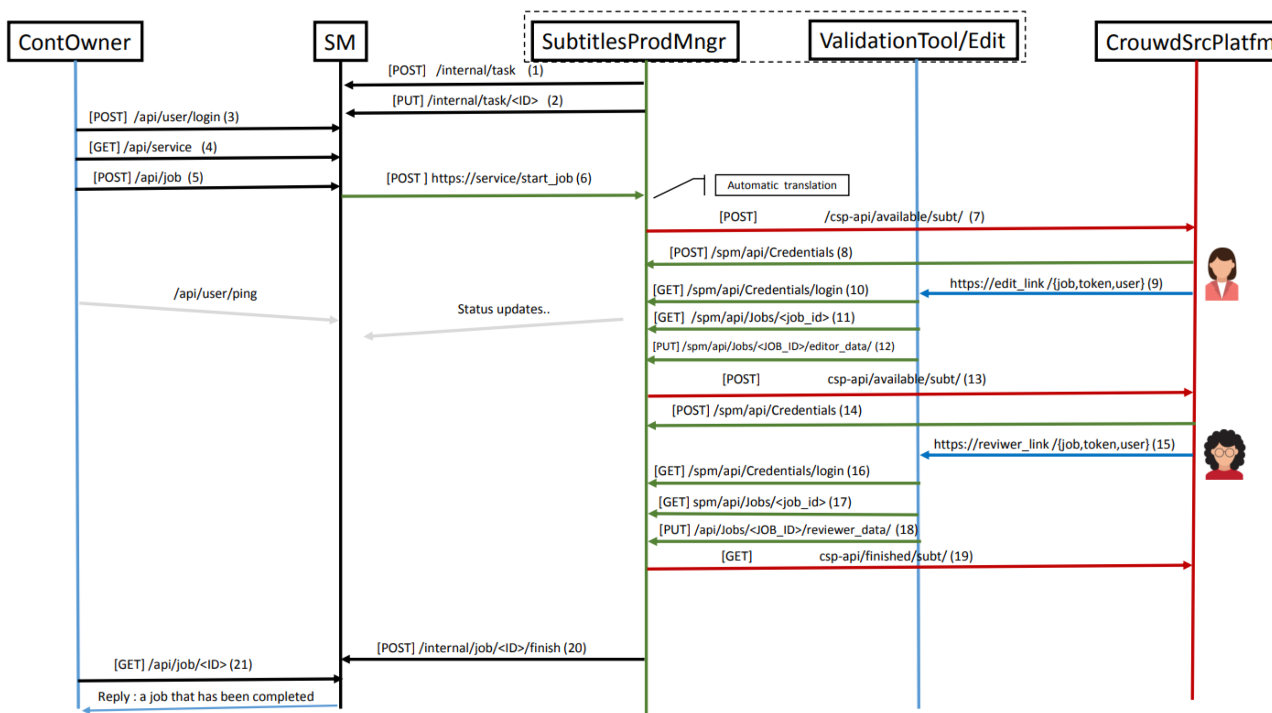


Figure 5 - Multi-language Subtitle translation end-to-end workflow

5. CONCLUSIONS

This report presented the work done in order to setup and implement the EasyTV multi-terminal technical platform. The description followed a bottom-up approach by starting from the platform foundation: data center overview, virtual machine choice, storage and networking have been described. Then, moving up to the description of the platform software solution and the related EasyTV services, an overview of the core technologies has been given, followed by details on how the platform was developed and configured, together with the list of containerized services available.

Given its importance, a separate chapter has been devoted to the EasyTV Service Manager module and a workflow involving more than one EasyTV service has been also analyzed. An appendix (Appendix B) has been added to the present document in order to describe the EasyTV Service Manager API. Two set of APIs are described: one set regarding integration with the EasyTV services and one set regarding the integration with the Content Owner.

The EasyTV multi-terminal technical platform can potentially host a large number of services thanks to the container scaling and orchestration capabilities provided. As an alternative container orchestrator, Kubernetes [8] could fit the role in a possible platform evolution scenario. Of course, additional nodes could be added too to provide more resources in terms of computational power and storage space, and also to support specific service requirements that need to be satisfied.

6. REFERENCES

[1] The CentOS Project. CentOS. [Online]. https://www.centos.org/

[2] Docker. Docker Glossary. [Online]. https://docs.docker.com/glossary/

[3] Docker. Install Docker on CentOS. [Online]. https://docs.docker.com/install/linux/docker-ce/centos/

[4] Portainer. Portainer CE. [Online]. https://www.portainer.io/products-services/portainer-community-edition/

[5] Portainer. Portainer. [Online]. https://www.portainer.io/

[6] Docker. Docker network overview. [Online]. https://docs.docker.com/network/

[7] Kong Inc. Kong HQ. [Online]. https://konghq.com/

[8] Kubernetes. Kubernetes. [Online]. https://kubernetes.io

APPENDIX A - EASYTV SERVICES DOCKER COMPOSE DEFINITION

This appendix lists the Docker Compose files used to build and deploy the EasyTV stacks running on the platform.

1. CATALOGUE_STACK

```
version: '3.1'

services:
  catalogue-mongo:
    image: mongo:3.5
    networks:
      - easytv_net
    volumes:
      - /dati/catalogue/db:/data/db
    ports:
      - "20000:27017"

  catalogue-cms:
    image: catalogue-cms:1.0
   networks:
     easytv_net
   ports:
      - "8088:1337"

networks:
  easytv_net:
    external: true
```

2. CP_STACK

```
version: '3.1'
services:
 crowdsourcing_app:
  image: crowdsourcing-app:latest
  networks:
   - easytv_net
  ports:
   - "8083:1337"
  environment:
   CP_MONGO_HOST: "mongodb://crowdsourcing_mongo/cpdb"
   CP_REDIS_HOST: "redis://crowdsourcing_redis"
   BASE_URL: "https://cp.easytv.eng.it"
   ADMIN_KEY: "admin-access-12345"
   IS_DOCKER: "yes"
   MANAGE_LEVEL: "YES"
   RESTORE_ORGS: "NO"
   FORCE_CONFIRM: "NO"
   http_proxy: "http://161.27.206.250:8080"
   https_proxy: "http://161.27.206.250:8080"
   no_proxy: ".easytv.eng.it"
  volumes:
   - /dati/docker/volumes/cp-vol/_data
   - /dati/cp/assets/mocap-submits:/path/inside/container/mocap-submits
   - /dati/cp/assets/video-submits:/path/inside/container/video-submits

 crowdsourcing_mongo:
  image: mongo:3.5
  networks:
   - easytv_net
  ports:
   - "27017:27017"
  volumes:
   - /dati/cp/db:/data/db

 crowdsourcing_redis:
  image: redis:5.0-alpine
```

```
    networks:
      - easytv_net

networks:
  easytv_net:
    external: true
```

3. HP_STACK

```
version: '3.1'

services:

  stmm:
    image: stmm:v1
    networks:
      - easytv_net
    depends_on:
      - stmm_rt
    environment:
      STMM_HOST: "stmm_rt"
      STMM_PORT: "8077"
      DB_HOST: "personalization_db"
      DB_PORT: "3306"
      DB_NAME: "easytv"
      DB_USER: "easytv"
      DB_PASSWORD_FILE: "/run/secrets/hpapidb_pass"
    stdin_open: true
    tty: true
    secrets:
      - hpapidb_pass
      - hpapidbroot_password

  stmm_rt:
    image: stmm_rt:v1
    networks:
      - easytv_net
    environment:
      PORT: "8077"

  rbmm:
    image: rbmm:v1
    networks:
      - easytv_net
    environment:
```

```
      PORT: "8080"

  hbmm:
    image: hbmm:v1
    networks:
      - easytv_net
    depends_on:
      - stmm_rt
      - rbmm
    environment:
      STMM_HOST: "stmm_rt"
      RBMM_HOST: "rbmm"
      STMM_PORT: "8077"
      RBMM_PORT: "8080"
      PORT: "8087"
      http_proxy: "http://161.27.206.250:8080"
      https_proxy: "http://161.27.206.250:8080"
      no_proxy: ".easytv.eng.it, rbmm, stmm_rt"
    ports:
      - "8087:8087"

secrets:
  hpapidb_pass:
    external: true
  hpapidbroot_password:
    external: true

networks:
  easytv_net:
    external: true
```

4. HPLOGIN_STACK

```
version: '3.1'

services:

  personalization_db:
    image: mysql:5.7
    environment:
      MYSQL_DATABASE: "easytv"
      MYSQL_USER: "easytv"
      MYSQL_PASSWORD_FILE: "/run/secrets/hpapidb_pass"
      MYSQL_ROOT_PASSWORD: "/run/secrets/hpapidbroot_password"
    volumes:
      - /dati/hp/db:/var/lib/mysql
    networks:
      - easytv_net
    command: --default-authentication-plugin=mysql_native_password
    secrets:
      - hpapidb_pass
      - hpapidbroot_password

  personalization_api:
    image: easytv-hp-api:v6
    networks:
      - easytv_net
    environment:
      DB_NAME: "easytv"
      DB_USER: "easytv"
      DB_PASS_FILE: "/run/secrets/hpapidb_pass"
      JWT_SECRET: "./run/secret/jwt_secret"
      DB_HOST: "personalization_db"
      PORT: "80"
    ports:
      - "8086:80"
    secrets:
      - hpapidb_pass
```

```
secrets:
  hpapidb_pass:
    external: true
  hpapidbroot_password:
    external: true

networks:
  easytv_net:
    external: true
```

5. REGISTRY

```
version: '3.1'
services:
  registry:
    image: registry:2.6.2
    volumes:
      - /dati/registry:/var/lib/registry
    networks:
      - registry_net

  ui:
    image: docker-registry-ui:custom
    volumes:
      - ./registry-config/htpasswd:/etc/nginx/conf.d/nginx.htpasswd
    ports:
      - 5000:80
    environment:
      - REGISTRY_TITLE=EasyTV Service Registry
      - REGISTRY_URL=http://registry:5000
    depends_on:
      - registry
    networks:
      - registry_net

networks:
  registry_net:
      external: true
```

6. SM_STACK

```
version: '3.1'
services:
  service_manager_cron:
    image: easytv-sm:1.0.11
    networks:
      - easytv_net
    environment:
      DB_SCHEMA: "sm"
      DB_PORT: "5432"
      DB_HOST: "service_manager_db"
      DB_USER_FILE: "/run/secrets/smdb_user"
      DB_PASSWORD_FILE: "/run/secrets/smdb_password"
      http_proxy: "http://161.27.206.250:8080"
      https_proxy: "http://161.27.206.250:8080"
      no_proxy: ".easytv.eng.it, service_manager_api, crowdsourcing_app"
    command: ["crond", "-f", "-L", "/dev/stdout"]
    volumes:
      - /dati/sm/logs/cron:/var/log/sm
    secrets:
      - smdb_user
      - smdb_password

  service_manager_api:
    image: easytv-sm:1.0.11
    ports:
      - "8082:3000"
    networks:
      - easytv_net
    volumes:
      - /dati/sm/assets:/asset/
      - /dati/sm/logs/api:/var/log/sm
    depends_on:
      - service_manager_db
      - service_manager_cache
    environment:
      DB_SCHEMA: "sm"
```

```
      DB_PORT: "5432"
      DB_HOST: "service_manager_db"
      DB_USER_FILE: "/run/secrets/smdb_user"
      DB_PASSWORD_FILE: "/run/secrets/smdb_password"
      MAX_CONNECTIONS: "100"
      IDLE_CONNECTIONS: "10"
      MAX_CONN_LIFETIME: "10"
      SRT_CMD: "./srt"
      http_proxy: "http://161.27.206.250:8080"
      https_proxy: "http://161.27.206.250:8080"
      no_proxy: ".easytv.eng.it, service_manager_api, crowdsourcing_app"
    secrets:
      - smdb_user
      - smdb_password

  service_manager_ui:
    image: easytv-sm-ui:1.0.7
    ports:
      - "8081:80"
    networks:
      - easytv_net
    environment:
      NGINX_PORT: "80"
      SM_API_HOST: "service_manager_api"
      SM_API_PORT: "3000"

  service_manager_db:
    image: postgres:11.1-alpine
    environment:
      POSTGRES_USER_FILE: "/run/secrets/smdb_user"
      POSTGRES_PASSWORD_FILE: "/run/secrets/smdb_password"
      POSTGRES_DB: "sm"
    command: ["-c", "max_connections=100"]
    volumes:
      - /dati/sm/dbdata:/var/lib/postgresql/data
    networks:
      - easytv_net
```

```
    secrets:
      - smdb_user
      - smdb_password

  service_manager_cache:
    image: memcached:1.5.12-alpine
    networks:
      - easytv_net

secrets:
  smdb_user:
    external: true
  smdb_password:
    external: true

networks:
  easytv_net:
    external: true
```

7. SPM_STACK

```
version: '3.1'
volumes:
  spm-vol:
services:
  spm_api:
    image: easytvspmapi:latest
    volumes:
      - /dati/spm/db:/local-data
    networks:
      - easytv_net
    ports:
      - "8085:80"
    environment:
      http_proxy: "http://161.27.206.250:8080"
      https_proxy: "http://161.27.206.250:8080"
      no_proxy: ".easytv.eng.it, service_manager_api, crowdsourcing_app"
  spm_engine:
    image: easytvspmengine:latest
    networks:
      - easytv_net
    environment:
      http_proxy: "http://161.27.206.250:8080"
      https_proxy: "http://161.27.206.250:8080"
      no_proxy: ".easytv.eng.it, service_manager_api, crowdsourcing_app"
  spm_frontend:
    image: spm-frontend:0.23.6
    networks:
      - easytv_net
    ports:
      - "8084:80"
    environment:
      http_proxy: "http://161.27.206.250:8080"
      https_proxy: "http://161.27.206.250:8080"
      no_proxy: ".easytv.eng.it, spm_frontend, service_manager_api, crowdsourcing_app"

networks:
```

```
  easytv_net:
    external: true
```

APPENDIX B - SERVICE MANAGER API SPECIFICATION

1. INTRODUCTION

The Service Manager is a module of the EasyTV platform through which the Content Owner can access the EasyTV services. Apart from being the gateway of the EasyTV platform, the Service Manager will manage the status of jobs and also will be able to combine tasks that are offered by the EasyTV services into a single job.

In this document it will be defined how the EasyTV services, as well as the Content Owner will be able to communicate with the Service Manager. In order for it to be achieved, the Service Manager will provide two APIs, one for the EasyTV platform and the other for the Content Owner.

1.1. Terminology

Word	Refers To
Service	Is each EasyTV service/module that is exposed to the Content Owner through the Service Manager
Task	Is a specification of work that is offered by a **service** in the EasyTV platform
Job	Is created by the Content Owner to complete one or more **tasks** offered by the **services**.

1.2. JSON

One of the most popular data formating languages at the time of writing this document is JSON. It is a lightweight language that is easy to read and write both for machines and humans. There are libraries that help encode and decode JSON data for every programming language. The Service Manager APIs will use JSON for the input and the output of its API endpoints.

1.3. API conventions

The API uses the bellow HTTP status codes as listed in the table.

Code	Description
200	The action was completed sucessfuly
202	The requested action was accepted for processing
400	There was an error with the data or action request by the client
401	Unauthorized access. It is returned when a valid auth key/token is missing from the HTTP headers
404	The requested resource was not found
500	There was an internal error in the API

Every response of the API will also always return the following fields in their json response.

Name	Value	Type
"code"	A code that specifies whether the action was completed successfuly. The possible codes for every endpoint are documented later in this document	Integer
"description"	A description of the result of the action in English	String

For example a JSON response could be the following

```
{
        "code": 200,
        "description": "The job with id <ID> was canceled successfully"
}
```

Please note that the HTTP status code is different from the application code in the JSON response.

2. INTEGRATION WITH THE EASYTV SERVICES

2.1. Configuring the integration of a service

To achieve the easy integration with every service of the EasyTV platform as well as new services that might come in the future, the Service Manager will provide a generic way of handling the different services. For this reason before the communication can start, a service must be first registered to the Service Manager.

2.1.1. Obtaining an API key

The creators of the new service must contact the administrator of the Service Manager in order for the service to be registered and also to get an API key. This key must be passed to the Service Manager in every HTTP request that is made. For this purpose a custom HTTP header will be used as seen in the example bellow.

```
X-EasyTV-Key: <API_KEY>
```

The <API_KEY> is the actual API key that was given to the service. For the internal API the rest of the document assumes that all requests contain this API key in their headers.

2.2. Internal API

2.2.1. Task Configuration Management

2.2.1.1. Register a new task

Location: /internal/task

HTTP Method: POST

Description: Registers a new task for the current service

The request should have the following data as input in the JSON payload.

Name	Type	Value
"name"	String	A short name of this task
"description"	String	A description about the purpose of this task
"start_url"	String	The API url of the service that will be used by the Service Manager to create this task.
"cancel_url"	String	The API url of the service that will be used by the Service Manager to cancel this task.
"cancel_rest_url"	String	The REST API endpoint of the service that will be used by the Service Manager to cancel this task. By using this field instead of the "cancel_url", the Service Manager will send a DELETE request instead of a POST request.
"input"	Object	An object that contains the input parameters of the task. The key:value items in the object should be formatted as name:type. The type can be "string", "int" or "double"

"output"	Object	An object that contains the output of the task. The key:value items in the object should be formatted as name:type. The type can be "string", "int" or "double"

Example input JSON that creates a service named "Auto Translation" that has three string input parameters and one string output parameter:

```
{
        "name": "Auto translation",
        "description": "Automatically translates text to another language",
        "start_url": "https://service/start_job",
        "cancel_url": "https://service/cancel_job",
        "input": {
                "language_source": "string",
                "language_target": "string",
                "original_text": "string"
        },
        "output": {
                "translated_text": "string"
        }
}
```

Example input JSON that registers using a REST cancel url.

```
{
        "name": "Auto translation",
        "description": "Automatically translates text to another language",
        "start_url": "https://service/start_job",
        "cancel_rest_url": "https://service/job",
        "input": {
                "language_source": "string",
                "language_target": "string",
                "original_text": "string"
        },
        "output": {
                "translated_text": "string"
        }
}
```

The output will contain the following data in the JSON payload

Name	**Type**	**Value**
"code"	Integer	• 200: The task was registered successfuly • -8: A task with that name already exists • -9: The task should have at least one input parameter • -10: The task should have at least one output parameter • -11: The "start_url" does not look like a valid URL

		• -12: The “end_url” does not look like a valid URL • -13: The given “input” has invalid types • -18: The given “output” has invalid types • -400: There are missing required data in the JSON input. The “description” field will contain more information about the error. • -401: Unauthorized access. A valid api key is missing in the HTTP headers • -500: Internal server error
“description”	String	A short description of the outcome of the task
“task_id”	Integer	The id of the registered task. Only present if “code” is 200.

Example JSON:

```
{
        "code": 200,
        "description": "The task “Auto translation” was successful registered",
        "task_id": 16
}
```

2.2.1.2. Set availability of a Task

Location: /internal/task/<**ID**>

HTTP Method: PUT

Description: Enables/disables the task with id=<**ID**>. When a task is disabled new jobs **can't** be created from it.

The request should have the following data as input in the JSON payload.

Name	Type	Value
“disabled”	Boolean	• true: disable the task • false: enable the task

Example JSON:

```
{
        "disabled": true
}
```

The output will contain the following data in the JSON payload

Name	Type	Value
“code”	Integer	• 200: The availability was changed successfuly • -400: There are missing required data in the JSON input. The “description” field will contain more information about the error. • -401: Unauthorized access. A valid api key is missing in the HTTP headers

		• -404: The task with id=<ID> doesn’t exist • -500: Internal server error
“description”	String	A short description of the outcome of the task

Example response JSON:

```
{
        "code": 200,
        "description": "The task <ID> was successfully set to disabled"
}
```

2.2.1.3. Delete a Task

Location: /internal/task/<**ID**>

HTTP Method: DELETE

Description: Deletes the task with id=<**ID**>. A task can only be deleted when it is disabled and there are no active jobs based on it.

The output will contain the following data in the JSON payload

Name	**Type**	**Value**
“code”	Integer	• 200: The task was deleted successfuly • -1: The task can’t be deleted because it is not disabled • -2: The task can’t be deleted because there are active job depending on it. • -401: Unauthorized access. A valid api key is missing in the HTTP headers • -404: A task with id=<ID> doesn’t exist • -500: Internal server error
“description”	String	A short description of the outcome of the task

Example JSON:

```
{
        "code": 200
        "description": "The task <ID> was successful set to disabled"
}
```

2.2.2. Service Manager to EasyTV Service communication

As described when registering a new task in 2.2.1.1, two URLs are provided to the Service Manager in order to create or cancel a job on the service.

2.2.2.1. Posting a new job

The Service Manager in order to post a new job, will send a POST HTTP request to “start_url” that was given when registering the task.

Each request from the service manager to the service (using the start_url or cancel_url) will have the “X-EasyTV-Key” HTTP header set to the sha256 **hash** of the service’s API key. The service should use this value and compare it with the sha256 hash of their own copy of their API key and not accept any actions unless it is correct.

The JSON payload will have the following data:

Name	Type	Value
“job_id”	Integer	The id of the job created
“publication_date”	Integer	The date that the job should be completed. The value is a unix timestamp.
“expiration_date”	Integer	The date that every asset stored regarding this job should be deleted. The value is a unix timestamp.
“content_owner”	String	The name of the content owner that started this job
“input”	Object	An object that contains the input parameters formatted as name:value

Example input json for this request:


```
{
        "job_id": 5012,
        "publication_date": 1544448784,
        "expiration_date": 1544449890,
        "content_owner": "Owner Name",
        "input": {
                "language_source": "en",
                "language_target": "es",
                "original_text": "Hello world"
        }
}
```


When the service manager sends this request, it expects the following JSON data to be returned.

Name	Type	Value
“code”	Integer	• 200, The job was completed synchronously • 202, The job was accepted. It will be processed asynchronously • -13: Input parameter count, name or type is incorrect • -14: Invalid publication date, it shouldn’t be in the past • -15: Invalid expiration date, it should be greater than the publication_date • -400: There are missing required data in the JSON input. The “description” field will contain more information about the error. • -401: The service rejected the request because a wrong API key was sent. • -500: There was an internal error while processing the request
“description”	String	A small description of the outcome of this request. The purpose of this argument is to be logged in case of an

		error
“output”	Object	It is only present if the task was completed synchronously (“code”: 200). It will contain the output of the task formatted as name:value.

Example JSON of synchronous task:

```
{
	"code": 200,
	"description": "Job completed synchronously",
	"output": {
		"translated_text": "Hola Mundo"
	}
}
```

Example JSON of asynchronous task:

```
{
	"code": 202,
	"description": "Job accepted"
}
```

2.2.2.2. Canceling a job

The Service Manager in order to cancel a job, will send a POST HTTP request to the “cancel_url” that was given when registering the task.

If during the registration of the task, “cancel_rest_url” was used instead of “cancel_url” then the Service Manager will append the job id to the url and send a DELETE HTTP request to it. For example if “cancel_rest_url” was set to “https://example/job” then the service manager in order to cancel the job with id 54 will send a DELETE request to “https://example/job/54”.

Each request from the service manager to the service (using the start_url or cancel_url) will have the “X-EasyTV-Key” HTTP header set to the sha256 **hash** of the service’s API key. The service should use this value and compare it with the sha256 hash of their own copy of their API key and not accept any actions unless it is correct.

The JSON payload will contain the following.

Name	**Type**	**Value**
“job_id”	integer	The id of the job that should be canceled
“action”	String	Will always have the value “cancel”

Example input json:

```
{
	"job_id": 5012,
	"action": "cancel"
}
```

When the service manager sends this requests, it expects the following data to be returned encoded in JSON.

Name	Type	Value
"code"	Integer	• 200, The job was canceled • -3: The job is already canceled • -4: You can't cancel a completed job • -400: There are missing required data in the JSON input. The "description" field will contain more information about the error. • -401, The service rejected the request because a wrong API key was sent. • -404, there is no job with that id in the service • -500, The was an internal error while processing the request
"description"	String	A small description of the outcome of this request. The purpose of this argument is to be logged in case of an error

Example response JSON:

```
{
        "code": 200,
        "description": "Job canceled"
}
```

2.2.3. Get a list of all active jobs

Location:

- /internal/job
- /internal/job/limit/<**LIMIT**>
- /internal/job/limit/<**LIMIT**>/before/<**JOB_ID**>

HTTP method: GET

Description: Returns a list of active Jobs for this service and their status. The jobs are ordered by the date created in descending order, meaning the latest jobs will be shown first.

Pagination:

- **<LIMIT>** : is the maximum number of items to be returned
- **<JOB_ID>** : will return the jobs before the job with id=**<JOB_ID>** (not including that job)

The JSON response will contain the following values

Name	Type	Value
"code"	Integer	• 200: Success • -401: Unauthorized access. It is returned when a valid api key is missing from the HTTP headers • -500: Internal server error
"description"	String	A short description of the action's outcome
"next"	String	The endpoint to call to return the next page of data. If the current page is the last then the value will be

		null
“jobs”	Array	The array of jobs, only present if code is 200

On sucess the API will return an array of objects “jobs” inside the request that contain the information of the jobs. Each job will have the bellow information.

Name	**Type**	**Value**
“is_completed”	Boolean	Whether the job has been completed
“is_canceled”	Boolean	Whether the job has been canceled
“status”	String	Describes the current status of the job
“creation_date”	Integer	The timestamp that the job was created. Returned as a unix timestamp.
“publication_date”	Integer	The time limit by which the job should be ready. Returned as a unix timestamp.
“expiration_date”	Integer	Is the date that every resources liked to this job should be deleted. Returned as a unix timestamp.
“completion_date”	Integer	The timestamp that the job has been completed. If the job is in progress, the value will be null.
“content_owner”	String	The name of the content owner that started this job.
“id”	Integer	The id of the job.

Example JSON response:


```
{
	"code": 200,
	"description": "The following jobs exist for this service",
	"next": "/internal/job/limit/2/before/231"
	"jobs ": [
		{
			"id": 432,
			"is_completed": false,
			"is_canceled": false,
			"status": "A string that describes the current status of the job",
			"creation_date": 1544188422,
			"completion_date": null,
			"publication_date": 1544188500,
			"expiration_date": 1544188550,
			"content_owner": "content owner name"
		},
		{
			"id": 231
			"is_completed": false,
```

```
                        "is_canceled": false,
                        "status": "A string that describes the current status of the job",
                        "creation_date": 1544188422,
                        "completion_date": null,
                        "publication_date": 1544188500,
                        "expiration_date": 1544188550,
                        "content_owner": "content owner name"
                }
        ]
}
```

2.2.4. Get information about a certain job

Location: /internal/job/<**ID**>

HTTP method: GET

Description: Returns information about the job with the specified id in the URL

The JSON response will contain the following values

Name	**Type**	**Value**
“code”	Integer	• 200: Success • -401: Unauthorized access. It is returned when a valid api key is missing from the HTTP headers • -404: A job with the given id doesn’t exist • -500: Internal server error
“description”	String	A short description of the action’s outcome
“job”	Object	The information about the job. It is present only if the code is 200

On sucess the API will return the object “job” inside the request that contains the information of the speficied job.

Name	**Type**	**Value**
“is_completed”	Boolean	Whether the job has been completed
“is_canceled”	Boolean	Whether the job has been canceled
“status”	String	Describes the current status of the job
“creation_date”	Integer	The timestamp that the job was created. Returned as a unix timestamp.
“publication_date”	Integer	The time limit by which the job should be ready.

		Returned as a unix timestamp.
“expiration_date”	Integer	Is the date that every resources liked to this job should be deleted. Returned as a unix timestamp.
“completion_date”	Integer	The timestamp that the job has been completed. If the job is in progress, the value will be null.
“content_owner”	String	The name of the content owner that started this job
“id”	Integer	The id of the job

Example JSON response bellow:

```
{
        "code": 200,
        "description": "The job with id :id was found",
        "job ": {
                "id": 243,
                "is_completed": false,
                "is_canceled": false,
                "status": "A string that describes the current status of the job",
                "creation_date": 1544188422,
                "completion_date": null,
                "publication_date": 1544188500,
                "expiration_date": 1544188550,
                "content_owner": "content owner name"
        }
}
```

2.2.5. Set the status of a Job

Location: /internal/job/<**ID**>

HTTP method: PUT

Description: Update the status data field of a Job

The input should contain the following data in the JSON payload.

Name	**Type**	**Value**
“status”	String	A new description of the status that the job is currently in.

Example input JSON:

```
{
        "status": "Awaiting translation in crowd-sourcing platform"
}
```

The JSON response will contain the following values

Name	**Type**	**Value**
“code”	Integer	• 200: Success • -16: The Service can't update the job status because it has currently no pending tasks for this job • -400: There are missing required data in the JSON input. The “description” field will contain more information about the error. • -401: Unauthorized access. A valid api key is missing in the HTTP headers • -404: A job with the given id doesn’t exist • -500: Internal server error
“description”	String	A short description of the action’s outcome

Example JSON response:

```
{
	"code": 200,
	"description": "The job status was updated"
}
```

2.2.6. Cancel a Job

Location: /internal/job/<**ID**>

HTTP method: DELETE

Description: Cancel the job with id=<**ID**>

The JSON response will contain the following values

Name	**Type**	**Value**
“code”	Integer	• 200: Success • -3: The job has already been canceled • -4: A completed job can’t be canceled • -401: Unauthorized access. A valid api key is missing in the HTTP headers • -404: A job with the given id doesn’t exist • -500: Internal server error
“description”	String	A short description of the action’s outcome

Example JSON response:

```
{
	"code": 200,
	"description": "The job has been deleted"
```

```
}
```

2.2.7. Upload asset

Location: /internal/job/<**ID**>/asset

HTTP method: POST

Description: Upload an asset for a job. The asset will be stored and linked with this job. When the job expires the asset will expire as well.

This is the only endpoint of the Service Manager that doesn't accept JSON payload. Instead the HTTP body should be the file encoded as "multipart/form-data" with the name "asset".

The output of the request is JSON and will contain the following data

Name	Type	Value
"code"	Integer	• 200: The asset was uploaded successfuly • -5: Empty asset payload • -17: It is forbidden to upload asset for a completed or canceled job. • -401: Unauthorized access. A valid api key is missing in the HTTP headers • -404: A job with id=<ID> doesn't exist • -500: Internal server error
"description"	String	A short description of the action's outcome
"asset_id"	Integer	The id of the newly uploaded asset. Only returned if "code" is 200.
"asset_url"	String	A url to download the asset.

Example JSON response:

```
{
	"status": 200,
	"description": "Asset was uploaded",
	"asset_url": "/asset/file.txt",
	"asset_id": 54
}
```

2.2.8. Get assets

Location: /internal/job/<**ID**>/asset

HTTP method: GET

Description: Get a list of assets for the job with id=<ID>

The JSON response will contain the following values

Name	Type	Value
"code"	Integer	• 200: Successfully returned the asset list

		• -401: Unauthorized access. A valid api key is missing in the HTTP headers • -404: A job with id=<ID> doesn't exist • -500: Internal server error
"description"	String	A short description of the action's outcome
"assets"	Array	An array that contains information about each asset. Only present if the code is 200.

The "assets" array contains one objects about each asset. They have the following values.

Name	**Type**	**Value**
"asset_id"	Integer	The unique id of this asset
"asset_url"	String	The url to access this asset

Example JSON response:


```
{
        "status": 200,
        "description": "A list of assets has been returned for job with id=1",
        "assets": [
                {
                    "asset_id": 12,
                    "asset_url": "/asset/file.txt"
                },
                {
                    "asset_id": 18,
                    "asset_url": "/asset/file.docx"
                }
        ]
}
```


2.2.9. Finishing an active Job.

Location: /internal/job/<**ID**>/finish

HTTP Method: POST

Description: If a job has been processed asynchronously by the service, then in order to finish it, this endpoint must be called.

The input should contain the following data in the JSON payload.

Name	**Type**	**Value**
"output"	Object	It will contain the output of the task formatted as name:value. This is based on the information given at the task registration.

Example input JSON:

```
{
        "output": {
                "translated_text": "Hola Mundo"
        }
}
```

The JSON response will contain the following values

Name	Type	Value
“code”	Integer	• 200: Successfully finished the job • -18: Output parameter count, name or type is incorrect • -19: This is job is not completable by this service • -400: There are missing required data in the JSON input. The “description” field will contain more information about the error. • -401: Unauthorized access. A valid api key is missing in the HTTP headers • -404: A job with id=<ID> doesn’t exist • -500: Internal server error
“description”	String	A short description of the action’s outcome

```
{
        "status": 200,
        "description": "The job <ID> was finished"
}
```

2.2.10. Get tasks for the current service

Location: /internal/task

HTTP Method: GET

Description: Returns the tasks registered for this service

The JSON response will contain the following values

Name	Type	Value
“code”	Integer	• 200: Successfully finished the job • -401: Unauthorized access. A valid api key is missing in the HTTP headers • -500: Internal server error
“description”	String	A short description of the action’s outcome
“tasks”	Array	An array of object with information about each task

Example json response:

```
{
```

```
        "status": 200,
        "description": "Success",
        "task": [
                {
                        "id": 1,
                        "name": "Task A",
                        "description": "Description of task A",
                        "start_url": "http://task_a/start",
                        "cancel_url": "http://task_a/cancel",
                        "enabled": false,
                        "input": {
                                "var1": "string"
                        },
                        "output": {
                                "var2": "string"
                        }
                },
                {
                        "id": 2,
                        "name": "Task B",
                        "description": "Description of task B",
                        "start_url": "http://task_b/start",
                        "cancel_url": "http://task_b/cancel",
                        "enabled": false,
                        "input": {
                                "in1": "string"
                        },
                        "output": {
                                "out2": "string"
                        }
                }

        ]
}
```

3. INTEGRATION WITH THE CONTENT OWNER

3.1. Authentication

3.1.1. Registration

The Content Owner will be register to the Service Manager by the administrator of the Service Manager.

3.1.2. Login

Location: /api/user/login

HTTP method: POST

Description: Authenticates an user and creates a session token

The input should contain the following data in the JSON payload.

Name	Type	Value
“username”	String	The user’s username
“password”	String	The user’s password

Example input JSON:

```
{
        "username": "test-content-owner",
        "password": "123456789"
}
```

The JSON response will contain the following values.

Name	Type	Value
“code”	Integer	• 200: Login was successful • -400: There are missing required data in the JSON input. The “description” field will contain more information about the error. • -401: Username and/or password were incorrect • -500: Internal server error
“description”	String	A short description of the action’s outcome
“session_token”	String	The session token that needs to be passed in other requests. This token will expire after 20 minutes of being inactive.

Example response JSON:

```
{
        "code": 200,
```

```
        "description": "Login was successful",
        "session_token": "<TOKEN>"
}
```


After the user of the API gets the token, they need to include it in every further HTTP request that is made to the public API. This must be done by including it in the “X-EesyTV-Session” custom HTTP Header as shown in the example bellow.

```
X-EasyTV-Session: <TOKEN>
```

Where <TOKEN> is the actual value that is returned by the */api/login* endpoint.

All the rest of the HTTP requests that described in section 3 are assumed to have this header present in their requests.

3.1.3. Ping

Location: /api/user/ping

HTTP Method: GET

Description: Checks that the API is reachable and also extends the expiration of the token.

The JSON respone will contain the following data

Name	Type	Value
“code”	200	• 200: API is reachable and session is valid • -401: Unauthorized access. A valid session token is missing in the HTTP headers • -500: Internal server error
“description”	String	A short description of the action’s outcome

Example JSON response:


```
{
        "code": 200,
        "description": "pong"
}
```


Example JSON response where the session token has expired:


```
{
        "code": -401,
        "description": "No session"
}
```


3.1.4. Logout

Location: /api/user/logout

HTTP Method: DELETE

Description: Invalidates current user session.

The JSON response will contain the following data

Name	Type	Value

“code”	200	• 200: logout was successful • -401: Unauthorized access. A valid session token is missing in the HTTP headers • -500: Internal server error
“description”	String	A short description of the action’s outcome

Example JSON response:


```
{
        "code": 200,
        "description": "Session was destroyed"
}
```


3.1.5. Change password

Location: /api/user/change_password

HTTP method: POST

Description: Changes the password for the current user. After a successful change, the current session will be lost.

The input should contain the following data in the JSON payload.

Name	Type	Value
“old_password”	String	The current password
“new_password”	String	The new password
“new_password_verification”	String	Repeating the new password for verification

Example input JSON:


```
{
        "old_password": "password1234",
        "new_password": "password4321",
        "new_password_verification": "password4321"
}
```


The JSON response will contain the following values.

Name	Type	Value
“code”	Integer	• 200: Login was successful • -27: The new password is too short (it should be at least 8 characters) • -29: The new password doesnt match the verification • -400: There are missing required data in the JSON input. The “description” field will contain more information about the error. • -401: Old password is incorrect

		• -500: Internal server error
“description”	String	A short description of the action’s outcome

Example response JSON:

```
{
        "code": 200,
        "description": "Password change was successful",
}
```

3.2. Information Retrieval

3.2.1. Get the list of registered services

Location: /api/service

HTTP Method: GET

Description: Get a list of the available services

The JSON response will contain the following data

Name	Type	Value
“code”	Integer	• 200: Successfully returned the list of services • -401: Unauthorized access. A valid session token is missing in the HTTP headers • -500: Internal server error
“description”	String	A short description of the action’s outcome
“services”	Array	An array of services objects

The “services” array will contain an object for each service. These objects will have the following values:

Name	Type	Value
“name”	String	The name of the service
“id”	Integer	The id of the service
“description”	String	A description of the tasks that the service offers
“enabled”	Boolean	Whether this service is enable to receive new jobs
“tasks”	Array	Is an array that contains a object for each task

Each “task” object will have the following values”:

Name	Type	Value

“name”	String	The name of the task
“id”	Integer	The id of the task
“description”	String	A description of the task’s action
“enabled”	Boolean	Whether the task can accept new jobs
“input”	Object	An object that contains the input parameters of the task. The key:value items in the object should be formatted as name:type. The type can be “string”, “int” or “double”
“output	Object	An object that contains the output of the task. The key:value items in the object should be formatted as name:type. The type can be “string”, “int” or “double”

Example JSON response:


```
{
        "code": 200,
        "description": "Returning the list of services",
        "services": [
                {
                        "name": "Service A",
                        "id": 1,
                        "descriptions": "Offers tasks X",
                        "enabled": true,
                        "tasks": [
                                {
                                        "name": "X",
                                        "id": 144,
                                        "description": "Does X",
                                        "enabled": true,
                                        "input": {
                                                "in1": "string",
                                                "in2": "int"
                                        },
                                        "output": {
                                                "out1 ": "string"
                                        }
                                }
                        ]
                },
                {
                        "name": "Service B",
                        "id": 2,
                        "descriptions": "Offers tasks Z",
                        "enabled": true,
                        "tasks": [
                                {
                                        "name": "Z",
                                        "id": 154,
```

```
                                "description": "Does Z",
                                "enabled": true,
                                "input": {
                                        "in1": "string",
                                        "in2": "int"
                                },
                                "output": {
                                        "out1 ": "string"
                                }
                        }
                ]
        },
        {
                "name": "Service C",
                "id": 3,
                "descriptions": "Offers tasks N",
                "enabled": true,
                "tasks": [
                        {
                                "name": "N",
                                "id": 175,
                                "description": "Does N",
                                "enabled": true,
                                "input": {
                                        "in1": "string",
                                        "in2": "int"
                                },
                                "output": {
                                        "out1 ": "double"
                                }
                        }
                ]

        }
    ]
}
```


3.2.2. Get information about a Service and its tasks

Location: /api/service/<**ID**>

HTTP Method: GET

Description: Get information about a service with id=<**ID**> as well as the tasks that it supports

The JSON response will contain the following data

Name	Type	Value
“code”	Integer	• 200, Successfully returned the list of services • -401: Unauthorized access. A valid session token is missing in the HTTP headers • -404: A service with id=<ID> doen’t exist • -500: Internal server error

“description”	String	A short description of the action’s outcome
“service”	Object	The information about the service with id=<ID>. Is only present when the code is 200

The “service” object will have the following values:

Name	**Type**	**Value**
“name”	String	The name of the service
“id”	Integer	The id of the service
“description”	String	A description of the tasks that the service offers
“enabled”	Boolean	Whether this service is enable to receive new jobs
“tasks”	Array	Is an array that contains a object for each task

Each “task” object will have the following values”:

Name	**Type**	**Value**
“name”	String	The name of the task
“id”	Integer	The id of the task
“description”	String	A description of the task’s action
“enabled”	Boolean	Whether the task can accept new jobs
“input”	Object	An object that contains the input parameters of the task. The key:value items in the object should be formatted as name:type. The type can be “string”, “int” or “double”
“output	Object	An object that contains the output of the task. The key:value items in the object should be formatted as name:type. The type can be “string”, “int” or “double”

Example JSON response:


```
{
        "code": 200,
        "description": "Returning the list of services",
        "service": {
                "name": "Service A",
                "id": 1,
                "descriptions": "Offers tasks X and Y",
                "enabled": true,
                "tasks": [
                        {
                                "name": "X",
                                "id": 143,
                                "description": "Does X",
```

```
                                "enabled": true,
                                "input": {
                                        "in1": "string",
                                        "in2": "int"
                                },
                                "output": {
                                        "out1 ": "string"
                                }
                        },
                        {
                                "name": "Y",
                                "id": 154,
                                "description": "Does Y",
                                "enabled": true,
                                "input": {
                                        "in1": "double",
                                        "in2": "string"
                                },
                                "output": {
                                        "out1": "string",
                                        "out2": "string"
                                }
                        }

                ]
        }
}
```

3.3. Job Management

3.3.1. Create a new job

Location: /api/job

HTTP Method: POST

Description: Creates a new job on the Service Manager

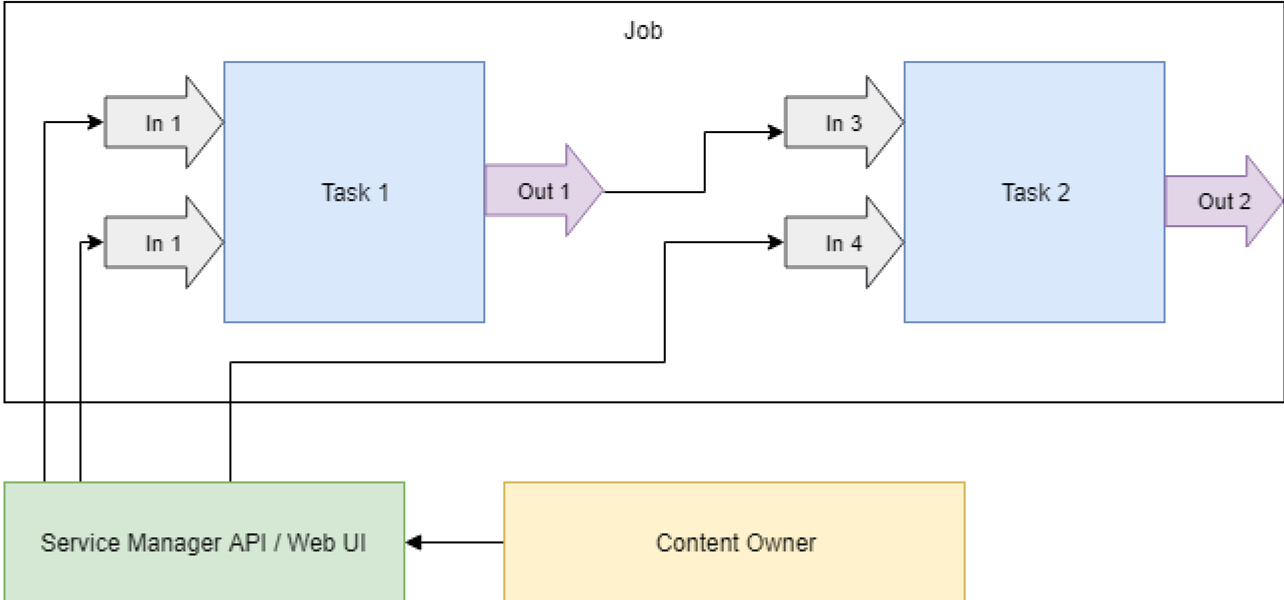
The input JSON should contain the following data:

Name	Type	Value
“publication_date”	Integer	The date that the job should be ready (unix timestamp in seconds)
“expiration_date”	Integer	The date that every asset regarding this job will be deleted (unix timestamp in seconds)
“tasks”	Array	The tasks that should be completed in this job. They will be completed in the order that they are defined in the array. The output of the job is the output of the last task.

Each object in the “tasks” array can have the following data.

Type	Name	Value
“task_id”	Integer	The id of the task that should be completed
“input”	Object	An object that contains the input parameters of the tasks, formatted as name:value
“linked_input”	Object	An object that contains the input parameters that are linked with the output of the last executed task in this job (if there is any). They are formated as “this-task-input-name”:“last-task-output-name”

A task input parameter can only be defined in the “input” or “linked_input” object, not both. Also the type of the two parameters must be same. A parameter that is of type string can only be linked with output parameters that are also of type string. If an input parameter of a task is not present in the JSON data, then an error occurs and the job will not be accepted.



In the above figure it can be seen more clearly how the input of a task will be evaluated in these two cases.

Example input JSON:

```
{
    "publication_date": 156000000,
    "expiration_date": 156000500,
    "tasks": [
        {
            "task_id": 5,
            "input": {
                "language_source": "el",
                "language_target": "es",
                "text": "Hello world"
            }
        },
        {
            "task_id": 6,
```

```
            "input": {
                "language_source": "it",
                "language_target": "es"
            },
            "linked_input: {
                "text": "translated_text"
            }
        }
    ]
}
```

In the above example a job will be created that consists of 2 tasks.

The first task will be the one with id=5 and three input parameters.

- 1st parameter has name “language_source” and value “el”
- 2nd parameter has name “language_target” and value “es”
- 3rd parameter has name “text” and value “Hello world”

The second task is the one with id=6 and has two regular input parameters and one linked parameter

- 1st parameter has name “language_source” and value “it”
- 2nd parameter has name ”language_target” and value “es”
- The 3rd parameter has name “text” and will take the value of the output parameter of the previous task (in this case the task with id=5) with name “translated_text”

The JSON response will contain the following information:

Name	**Type**	**Value**
“code”	Integer	• 200: Successfully created the job • -14: Publication date is not valid • -15: Expiration date is not valid (it should be later than the publication date) • -13: Input parameter count, name or type is incorrect • -20: An input parameter can only be linked with an output parameter of the same type • -21: The output parameter to be linked can't be found on the previous job • -400: There are missing required data in the JSON input. The “description” field will contain more information about the error. • -401: Unauthorized access. A valid session token is missing in the HTTP headers • -500: Internal server error
“description”	String	A short description of the action’s outcome
“job_id”	Integer	The id of the create job. Only present if the code is 200

Example response json:

```
{
    "code": 200,
```

```
    "description": "Job created",
    "job_id": 123
}
```

3.3.2. Cancel job

Location: /api/job/<**ID**>

HTTP Method: DELETE

Description: Cancels the job with id=<**ID**>

The JSON response will contain the following information:

Name	**Type**	**Value**
“code”	Integer	• 200: Successfully canceled the job • -3: The job is already canceled. • -4: You can't cancel a completed job. • -401: Unauthorized access. A valid session token is missing in the HTTP headers • -404: The job with id=<ID> doesn’t exist • -500: Internal server error
“description”	String	A short description of the action’s outcome

Example response json:

```
{
    "code": 200,
    "description": "Job <ID> was canceled"
}
```

3.3.3. Get all jobs

Locations:

- /api/job
- /api/job/limit/<**LIMIT**>
- /api/job/limit/<**LIMIT**>/before/<**JOB_ID**>

HTTP Method: GET

Description: Returns information about the job of the current user. The items are always returned sorted by date created in descending order (latest first)

Pagination:

- **<LIMIT>** : is the maximum number of items to be returned
- **<JOB_ID>** : will return the jobs before the job with id=**<JOB_ID>** (not including that job)

The JSON response will contain the following information:

Name	**Type**	**Value**
“code”	Integer	• 200: Success • -401: Unauthorized access. A valid session token is missing in the HTTP headers

		• -500: Internal server error
“description”	String	A short description of the action’s outcome
“next”	String	The endpoint to call to return the next page of data. If the current page is the last then the value will be null.
“jobs”	Array	Contains information about the job. Only exists when the code is 200.

The “jobs” array will contain one object about each job that will have the following data:

Name	**Type**	**Value**
“id”	Integer	The id of the job
“is_completed”	Boolean	Whether the job has been completed
“is_canceled”	Boolean	Whether the job has been canceled
“status”	String	The current status of the job
“creation_date”	Integer	The timestamp that the job was created. Returned as a unix timestamp in seconds
“publication_date”	Integer	The time limit by which the job should be completed. Returned as a unix timestamp in seconds
“expiration_date”	Integer	The date that every resources of the job will be deleted. Returned as a unix timestamp in seconds
“completion_date”	Integer	The timestamp that the job was actually completed. Returned as a unix timestamp in seconds
“tasks”	Array	An array of task objects that contain the task id and take name. The array is ordered the same as the execution order of the tasks.
“current_task”	Integer	It is the index of the current task in the “tasks” array. If the job is not in progress then it will be null.
“output”	Object	The output data of the job. Contains data formatted as name:value. It will be null if the job has not been completed, has been canceled or it has exceeded the publication date

Example JSON response:

```
{
        "code": 200,
        "description": "Job <ID> was found",
        "next": "/api/job/limit/3/before/123"
```

```
	"jobs": [
		{
			"id": 125,
			"is_completed": false,
			"is_canceled": false,
			"status": "A string that describes the current status of the job",
			"creation_date": 1544188422,
			"completion_date": null,
			"publication_date": 1544188500,
			"expiration_date": 1544188550,
			"tasks": [
				{
					"id": 153,
					"name": "Task A"
				},
				{
					"id": 123,
					"name": "Task B"
				},
				{
					"id": 452,
					"name": "Task C"
				}
			],
			"current_task": 1,
			"output": null
		},
		{
			"id": 124,
			"is_completed": true,
			"is_canceled": false,
			"status": "A string that describes the current status of the job",
			"creation_date": 1544188422,
			"completion_date": 1544188500,
			"publication_date": 1544188500,
			"expiration_date": 1544188550,
			"tasks": [
				{
					"id": 153,
					"name": "Task A"
				},
				{
					"id": 123,
					"name": "Task B"
				},
				{
					"id": 452,
					"name": "Task C"
				}
			],
			"current_task": null,
			"output": {
				"translated_subtitle": "http://example.com/s.srt"
```

```
                        }
                },
                {
                        "id": 123,
                        "is_completed": false,
                        "is_canceled": false,
                        "status":  "A string that describes the current status of the job",
                        "creation_date": 1544188422,
                        "completion_date": null,
                        "publication_date": 1544188500,
                        "expiration_date": 1544188550,
                        "tasks": [
                                {
                                        "id": 153,
                                        "name": "Task A"
                                },
                                {
                                        "id": 123,
                                        "name": "Task B"
                                },
                                {
                                        "id": 452,
                                        "name": "Task C"
                                }
                        ],
                        "current_task": 0,
                        "output": null
                }

        ]
}
```


3.3.4. Get information about a specific job

Location: /api/job/<**ID**>

HTTP Method: GET

Description: Returns information about the status of job with id=<**ID**>

The JSON response will contain the following information:

Name	**Type**	**Value**
“code”	Integer	• 200: Successfully created the job • -401: Unauthorized access. A valid session token is missing in the HTTP headers • -404: The job with id=<ID> doesn’t exist • -500: Internal server error
“description”	String	A short description of the action’s outcome
“job”	Object	Contains information about the job. Only exists when

		the code is 200.

The “job” object will contain the following data:

Name	**Type**	**Value**
“id”	Integer	The id of the job
“is_completed”	Boolean	Whether the job has been completed
“is_canceled”	Boolean	Whether the job has been canceled
“status”	String	The current status of the job
“creation_date”	Integer	The timestamp that the job was created. Returned as a unix timestamp.
“publication_date”	Integer	The time limit by which the job should be completed. Returned as a unix timestamp
“expiration_date”	Integer	The date that every resources of the job will be deleted. Returned as a unix timestamp
“completion_date”	Integer	The timestamp that the job was actually completed. Returned as a unix timestamp.
“tasks”	Array	An array of the task IDs of the job. The array is ordered the same as the execution order of the tasks.
“current_task”	Integer	It is the index of the current task in the “tasks” array. If the job is not in progress then it will be null.
“output”	Object	The output data of the job. Contains data formatted as name:value. It will be null if the job has not been completed, has been canceled or it has exceeded the publication date.

Example JSON response of a job in progress:


```
{
        "code": 200,
        "description": "Job <ID> was found",
        "job": {
                "id": 125,
                "is_completed": false,
                "is_canceled": false,
                "status": "A string that describes the current status of the job",
                "creation_date": 1544188422,
                "completion_date": null,
                "publication_date": 1544188500,
                "expiration_date": 1544188550,
                "tasks": [
```

```
                        {
                                "id": 153,
                                "name": "Task A"
                        },
                        {
                                "id": 123,
                                "name": "Task B"
                        },
                        {
                                "id": 452,
                                "name": "Task C"
                        }
                ],
                "current_task": 0,
                "output": null
        }
}
```


Example JSON response of a job that has been completed


```
{
        "code": 200,
        "description": "Job <ID> was found",
        "job": {
                "id": 125,
                "is_completed": true,
                "is_canceled": false,
                "status": "A string that describes the current status of the job",
                "creation_date": 1544188422,
                "completion_date": 1544188500,
                "publication_date": 1544188500,
                "expiration_date": 1544188550,
                "tasks": [
                        {
                                "id": 153,
                                "name": "Task A"
                        },
                        {
                                "id": 123,
                                "name": "Task B"
                        },
                        {
                                "id": 452,
                                "name": "Task C"
                        }
                ],
                "current_task": null,
                "output": {
                        "translated_subtitle": "http://example.com/download.srt"
                }
        }
}
```