SignON

Sign Language Translation Mobile Application and Open Communications Framework

Deliverable D2.4 -

Intermediate release of the Open SignON Framework

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No. 101017255

Project Information
Project Number: 101017255
Project Title: SignON: Sign Language Translation Mobile Application and Open Communications Framework
Funding Scheme: H2020-FT-57-2020
Project Start Date: January 1st 2021

Deliverable Information
Title: D2.4 - Intermediate release of the Open SignON Framework
Work Package: 2 - SignON Service and Mobile App
Lead Beneficiary: MAC
Due Date: 28/02/2023
Revision Number: V1.0
Author: John O'Flaherty (MAC), Marco van der Laan (INT), Marcello Paolo Scipioni (FINCONS), Marco Giovanelli (FINCONS), Riccardo Corrias (FINCONS), Vincent Vandeghinste (INT), Ed Keane (MAC)
Dissemination Level: Public
Deliverable Type: Demonstrator

Revision History

Version #	Implemented by	Revision Date	Description of changes
1.0	MAC	20/02/2023	Review, inputs and approval from all Partners
0.3	MAC	14/02/2023	Updates based on discussions and inputs from Partners, D7.6 & further research.
0.2	FIN	27/01/2023	Fincons' contribution to sections related to SignON Framework architecture, API, integration and deployment.
0.1	MAC	16/12/2022	Initial draft based on WP2 discussions, internal documents and D2.3.

The SignON project has received funding from the European Union's Horizon 2020 Programme under Grant Agreement No. 101017255. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the SignON project or the European Commission. The European Commission is not liable for any use that may be made of the information contained therein.

Approval Procedure

Version #	Deliverable Name	Approved by	Institution	Approval Date
V1.0	D2.4	Shaun O'Boyle	DCU	01/02/2023
V1.0	D2.4	Marco Giovanelli	FINCONS	06/02/2023
V1.0	D2.4	Vincent Vandeghinste	INT	13/02/2023
V1.0	D2.4	Gorka Labaka, Adrian Nuñez	UPV/EHU	10/02/2023
V1.0	D2.4	John O'Flaherty, Ed Keane	MAC	10/02/2023
V1.0	D2.4	Horacio Saggion	UPF	31/01/2023
V1.0	D2.4	Irene Murtagh	TU Dublin	20/02/2023
V1.0	D2.4	Lorraine Leeson	TCD	14/02/2023
V1.0	D2.4	Karim Dahdah	VRT	14/02/2023
V1.0	D2.4	Mathieu De Coster	UGent	30/01/2023
V1.0	D2.4	Caro Brosens	VGTC	14/02/2023
V1.0	D2.4	Henk van den Heuvel	RU	02/02/2023
V1.0	D2.4	Catia Cucchiarini	TaalUnie (NTU)	13/02/2023
V1.0	D2.4	Bram Vanroy	KU Leuven	13/02/2023
V1.0	D2.4	Davy Van Landuyt	EUD	30/01/2023
V1.0	D2.4	Mirella De Sisto	TiU	07/02/2023

Acronyms

The following table provides definitions for acronyms and terms relevant to this document.

Acronym	Definition
API	Application Programming Interface
App	SignON Communication and Translation Mobile Application
ASL	American Sign Language
ASR	Automated Speech Recognition
BSL	British Sign Language
CUDA	CUDA is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels
DHH	Deaf and Hard of hearing
DoA	Description of the Action
FTP	File Transfer Protocol
GA	Grant Agreement
GB	Gigabyte
GPU	Graphical Processor Unit
HDD	Hard-Disk Drives
HTML	Hypertext Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
ICT	Information and Communication Technologies
InterL	Interlingua
IS	International Sign
ISCSI	Internet Small Computer System Interface

ISL	Irish Sign Language
ITIL	Information Technology Infrastructure Library
LSE	Spanish Sign Language (Lengua de Signos Española)
ML	Machine Learning
MT	Machine Translation
NFS	Network File System
NGT	Sign Language of the Netherlands (Nederlandse Gebarentaal)
NLP	Natural Language Processing
RAID	Redundant Array of Independent Discs
REST	Representational state transfer
SFTP	Secure File Transfer Protocol
SignON	Both the service and this project (GA 101017255)
SL	Sign Language
SLR	Sign Language Recognition
SLTT	Sign-Language-To-Text
SSD	Solid state drives
STT	Speech-To-Text
TB	Terabyte
TTS	Text-to-Speech
TTSL	Text-to-Sign-Language
UI	User Interface
URL	Uniform Resource Locator
UX	User Experience

VGT	Flemish Sign Language (Vlaamse Gebarentaal)
VM	Virtual Machine
VPN	Virtual Private Network
WP	Work Package
WWW	World Wide Web

Table of Contents

List of Figures

List of Tables

Executive Summary

This deliverable is the intermediate release of the Open SignON Framework as a demonstrator. This report describes the progress of the shared SignON platform, which consists of two separate entities: the repository with reference data and training data, and the platform with processing space to host both developing and developed/production components of the SignON Framework service, software and data, since D2.3 “First release of the SignON Open Cloud platform” was delivered in January 2022.

1. Introduction

SignON is researching and developing the SignON Transmodal Machine Translation Mobile Application communication service that uses machine translation to translate between sign and spoken languages. This service will facilitate the exchange of information among deaf and hard of hearing (DHH), and hearing individuals. In this user-centric and community-driven project we are tightly collaborating with European DHH communities to (re)define use-cases, co-design and co-develop the SignON service and application, assess the quality and validate their acceptance. Our ultimate objective is the fair, unbiased and inclusive spread of information and digital content in European society.

SignON will be a free, open-source application and framework for conversion between video (capturing and understanding sign languages), audio and text and translation between signed and spoken languages. To facilitate these tasks, SignON uses a common representation for mapping of video, audio and text into a unified space that is used for translating into the target modality and language. To ensure wide uptake, improved sign language (SL) detection and synthesis, as well as multilingual speech processing on mobile devices for everyone, we will deploy the SignON service as a smart phone application running on standard modern devices.

The SignON App has a lightweight interface (see section 6). The SignON Framework of services, however, is distributed on a cloud platform (see section 2-5) where the computationally intensive services are executed. The project is driven by a focused set of use-cases tailored towards the sign language communities. We target signed and spoken languages from Ireland (Irish Sign Language, Irish and English), Britain (British Sign Language and English), the Netherlands (Sign Language of the Netherlands/Nederlandse Gebarentaal and Dutch), the Flanders region of Belgium (Flemish Sign Language, Flemish and Dutch) and Spain (Spanish Sign Language and Spanish). Nevertheless, SignON will eventually incorporate machine learning capabilities that will allow (i) learning new sign, written, and spoken languages; (ii) style-, domain- and user-adaptation and (iii) automatic error correction, based on user feedback.

This report describes the progress of the shared SignON Framework platform since the D2.3 “First release of the SignON Open Cloud platform” was delivered in January 2022.

2. Updated Architecture of the Cloud Platform

The internal architecture of the SignON Framework is composed of different components, namely, the SignON Orchestrator, the SignON Dispatchers, the SignON Pipeline Components (e.g., Sign Language Recognition (SLR), Natural Language Processing (NLP), Automated Speech Recognition (ASR), etc.) and the Object Storage (see Figure 1). In this section is presented a detailed description of each component and how they communicate with each other.

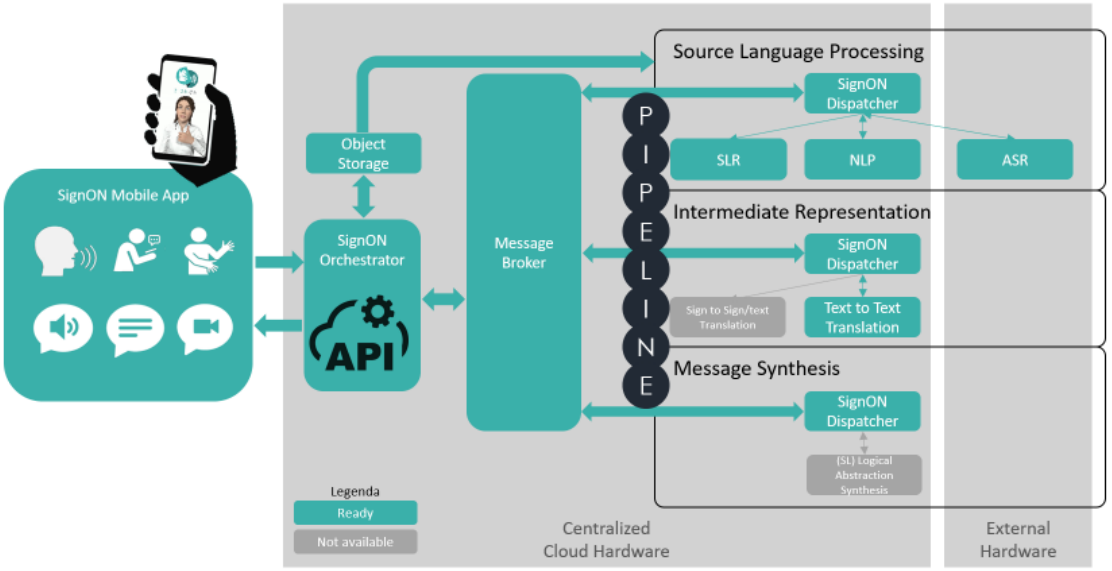


Figure 1 SignON Framework Architecture

The SignON Mobile App communicates through a REST API with the SignON Orchestrator, while the communication between SignON Orchestrator and SignON Dispatchers is supported by a message broker.

Depending on the input (text, audio or video) and the requested output (text, audio, avatar), the SignON Orchestrator invokes the SignON Dispatchers, which in turn contact the Object Storage (if needed) and the SignON Pipeline Components responsible for the relative processing.

In case the input message from the SignON Mobile App contains an audio or video file, the SignON Mobile App will first make a request to the SignON Orchestrator, which in turn contacts the Object Storage component to return a pre-signed URL. The SignON Mobile App can then use the pre-signed URL to directly upload the file to the Object Storage.

In all the other cases, when the SignON Orchestrator receives a message from the SignON Mobile App, it delivers the message to the SignON Dispatchers that in turn invokes all the other SignON Pipeline Components to elaborate the response message.

In detail, as shown in Figure 2, once the message is received by the first SignON Dispatcher for “WP3 - Source Language Processing”, depending on the type of input given and the output requested by the SignON Mobile App, it will invoke the relative SignON Pipeline Components through a REST API call to process the message and add information for the translation task. Then the resulting message is given to a temporary queue that connects the first SignON Dispatcher with the second SignON Dispatcher for “WP4 - Intermediate Representation”. When the second SignON Dispatcher receives a message from the temporary queue, it invokes the Text to Text Translation component to add additional information on the translation. Once finished, like in the previous step, the resulting message is sent to a temporary queue that connects the second SignON Dispatcher with the third SignON Dispatcher for “WP5 - Message Synthesis”. The third SignON Dispatcher like the two previous dispatchers is connected through REST API to external components to generate the final version of the message that is returned to the SignON Orchestrator and from there to the SignON Mobile App.

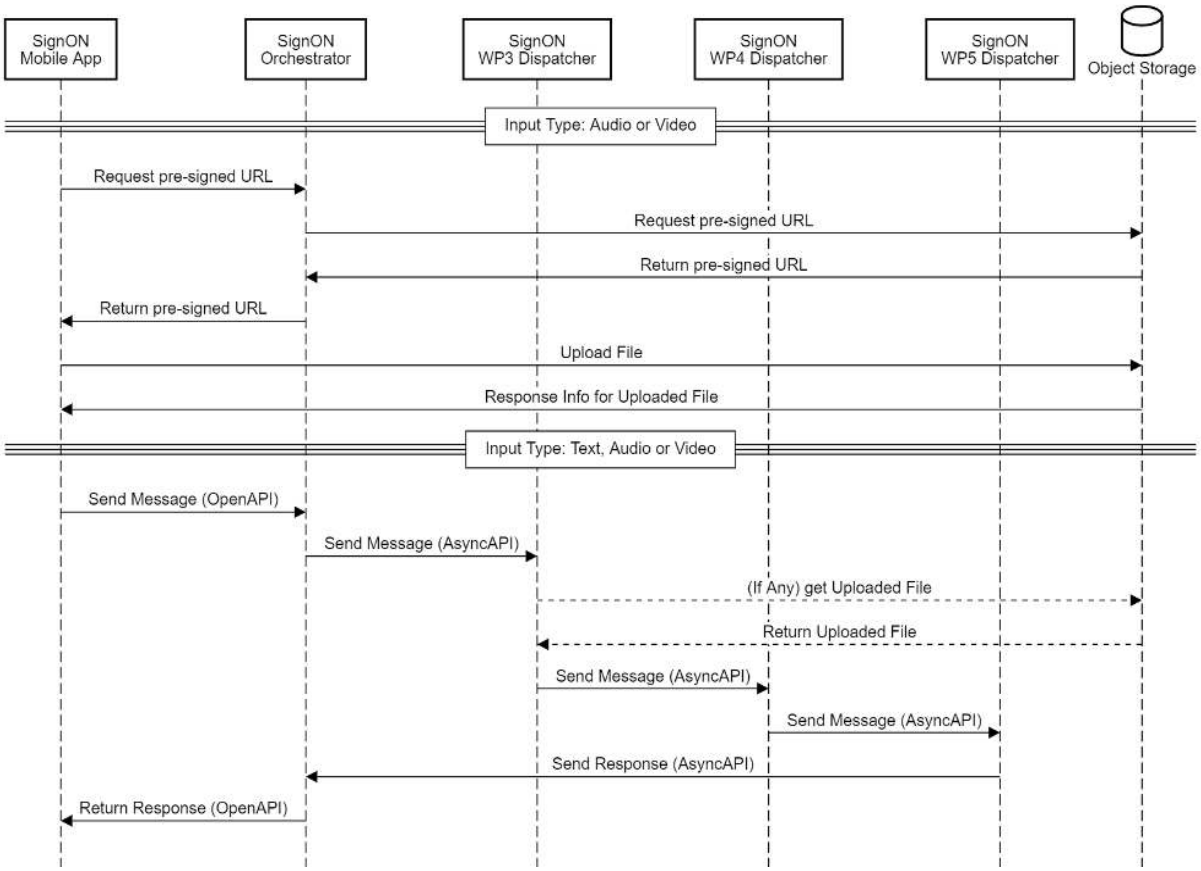


Figure 2 Sequence diagram of the SignON Framework

2.1 SignON Orchestrator

As previously explained in D2.2[1], the SignON Orchestrator component is part of the SignON Framework architecture and connects the SignON Mobile App with the SignON Pipeline Components. More precisely, the source message from the SignON Mobile App is handled through the SignON Orchestrator, which queues it towards the SignON Pipeline Components through the RPC pattern of RabbitMQ[2] message broker.

The RPC pattern is used in the SignON Framework to take advantage of a stateless architecture. In fact, the RPC pattern is a request–response communication, initiated by the client, which sends a request message to a known remote server to execute a specified procedure. The remote server sends a response to the client, and the application continues its process.

To implement the SignON Orchestrator different technologies and frameworks have been used, and the component is mainly based on the Spring Boot[3] framework in Java.
Furthermore, part of classes in the SignON Orchestrator are automatically generated starting from the AsyncAPI[4] and OpenAPI[5] YAML files through the Swagger Codegen[6] and AsyncAPI/generator[7] tools, to ensure an API-first approach and fast prototyping (more information on the topic are available in section “3 - APIs”).

More technically the SignON Orchestrator project is structured as follows:

- Controllers Packages: Contains the controllers that administer the relative API calls.
    - Inference Storage Auth Controller: Returns a pre-signed URL for the SignON Mobile App to directly upload the file on the Object Storage. More precisely, in order to return a pre-signed URL, the object name of the file is generated by the SignON Orchestrator according to the following pattern:

      ```
      appInstanceID + `/` + timestamp + `_` + UUID + `.` + fileFormat
      ```

        - The “appInstanceID” corresponds to the SignON Mobile App Instance Identifier.

[1] D2.2: SignON Services Framework Architecture
https://signon-project.eu/wp-content/uploads/2022/01/SignON_D2.2_Services_Framework_Architecture_v1.0.pdf
[2] https://www.rabbitmq.com/
[3] https://spring.io/projects/spring-boot
[4] https://www.asyncapi.com/
[5] https://www.openapis.org/
[6] https://swagger.io/tools/swagger-codegen/
[7] https://www.asyncapi.com/tools/generator

- The “timestamp” corresponds to the time when the message has been received by the SignON Orchestrator from the SignON Mobile App.
- The “UUID” corresponds to a 128-bit universally unique identifier.
- The “fileFormat” corresponds to the extension of the file that is uploaded to the Object Storage component.

  - Orchestrator Controller: For each SignON Mobile App request, a new thread is created. It prepares the message appending timestamps and queue identifiers of the RabbitMQ RPC pattern, sends it and waits for a response. Once the SignON Pipeline Components have processed and returned the message, the controller deletes the uploaded file (if any) from the Object Storage and finally returns the message to the SignON Mobile App.
  - Endpoints Controller: Returns all the available endpoints that can be called to interact with the SignON Orchestrator.
  - Status Controller: Returns a boolean that corresponds to whether the SignON Orchestrator is ready.
  - Version Controller: Returns the current version for the SignON Orchestrator, SignON OpenAPI and SignON AsyncAPI.

- Errors Reporting Packages:

  Contains the different types of errors that can be returned internally from both the SignON Orchestrator or the SignON Dispatchers and externally from the Object Storage or RabbitMQ.
- Resources:

  Contains files describing the SignON OpenAPI and SignON AsyncAPI. These structures are used to automatically generate the Java files that are needed to manage the communication between the different components in the SignON Framework.
- Target:

  Contains the Java files automatically generated from the SignON OpenAPI and SignON AsyncAPI needed to manage the communication between the different components of the SignON Framework.

As shown, a system of Error Reporting is introduced in the SignON Orchestrator, to support the SignON Mobile App to distinguish different exceptions, manage them and eventually inform the user. This has been done by dividing errors into categories and classifying them in Internal and External exceptions. This system allows also to speed up the internal testing by possibly returning the

complete stack trace of the error, with a debug mode that can be enabled only from server side through the relative configuration file.

In fact, to allow the SignON Orchestrator to be configurable, a YAML configuration file has been prepared, allowing modification of settings regarding RabbitMQ, the Object Storage and other internal parameters.

2.2 SignON Dispatchers

As previously introduced in D2.2[8], at first a SignON Pipeline Simulator was designed to enable the fast prototyping approach and integrate the SignON Mobile App even if the SignON Pipeline Components were still under development.
Then, in order to allow the SignON Pipeline Components to analyse a message without implementing the communication channels between them, the SignON Pipeline Simulator has been replaced with a solution based on SignON Dispatchers. This way, when a message is given in input from the Mobile App, it is pre-processed by the SignON Orchestrator, and then it is passed with a "piggyback" approach[9] to a series of three SignON Dispatchers: one for the "WP3 - Source Language Processing", one for the "WP4 - Intermediate Representation" and one for "WP5 - Message Synthesis". The aim of each of these SignON Dispatchers is to connect different SignON Pipelines Components and pass the message to the next SignON Dispatcher until the last SignON Dispatcher returns it to the SignON Orchestrator, which in turn forwards it to the SignON Mobile App.

The communication between the three dispatchers is done through the Python library "pika"[10]. This library allows connecting them to the RabbitMQ[11] message broker, using the publish/subscribe approach[12].
The communication between the SignON Pipeline Components and the SignON Dispatchers are then made through REST API calls.

[8] D2.2: SignON Services Framework Architecture
https://signon-project.eu/wp-content/uploads/2022/01/SignON_D2.2_Services_Framework_Architecture_v1.0.pdf
[9] i.e., the original message is augmented with some information and passed to the next receiver
[10] https://pypi.org/project/pika/
[11] https://www.rabbitmq.com/
[12] https://www.rabbitmq.com/tutorials/tutorial-three-python.html

Each message that goes through the SignON Dispatchers has the form of a JSON object, determined by the YAML file used for the AsyncAPIs creation (a more detailed explanation can be found in section “3 - APIs”).

The SignON Orchestrator is connected with the first SignON Dispatcher for “WP3 - Source Language Processing” through an RPC queue created with RabbitMQ. The same applies to the last SignON Dispatcher for “WP5 - Message Synthesis” that communicates with the SignON Orchestrator with the same RPC queue. The communication between the SignON Dispatchers is based on temporary queues instead.

If needed, the SignON Dispatchers can retrieve audio and video files from the Object Storage through the Object ID specified in the message (see section “2.3 - Object Storage”). Once used, the files are actively deleted by the SignON Dispatcher itself in order to preserve the storage and ensure privacy.

Finally, a YAML configuration file has been prepared also for the SignON Dispatchers, allowing modification of settings regarding RabbitMQ, the Object Storage and other internal parameters.

2.3 Object Storage

To allow the SignON Mobile App to provide the previously mentioned SignON Pipeline Components with audio and video files, a component capable of uploading, storing and downloading objects is needed. For this task, MinIO[13], an open source Object Storage, has been chosen. This component can be connected with the dispatchers through the use of the “boto3”[14] Python library which is AWS S3[15] compliant and therefore enabling, if needed, the possibility of migrating to an “Infrastructure As A Service” (IAAS) approach.

By default, MinIO files can be uploaded only by an authorised account. To ease the use of the SignON Framework and avoid accounts creation, the SignON Mobile App will request a pre-signed URL to grant time-limited permission to upload an object.
In order to do so and as shown in Figure 2, these steps are followed:

1. The SignON Mobile App shall request a pre-signed URL to the SignON Orchestrator.

[13] https://min.io/
[14] https://pypi.org/project/boto3/
[15] https://aws.amazon.com/s3/

2. The SignON Orchestrator, as an authorised component, contacts the MinIO system to obtain a time-limited pre-signed URL for a specific object and a specific SignON Mobile App Instance ID.
3. The SignON Orchestrator returns the pre-signed URL to the SignON Mobile App.

The MinIO Object Storage is divided into buckets or containers to group the files. For the SignON purposes, every bucket is divided in folders related to each SignON Mobile App Instance ID, to allow a better debugging procedure in case of malfunctioning.

As previously mentioned in the SignON Orchestrator section, the MinIO objects are actively deleted once the processing is completed.

Furthermore, to configure the MinIO Object Storage, a script has been prepared to set accounts authorisation to upload and download objects, to apply different policies regarding the objects stored in the buckets and to define the endpoint on which MinIO has been mounted on.

3. APIs

This section includes a detailed explanation on how the SignON Framework communication is implemented, how the messages are composed and how the SignON Orchestrator classes, which handle the connection between the SignON mobile App, the SignON Orchestrator and the SignON Pipeline, are automatically generated.

In particular, the following OpenAPI[16] section refers to the REST API used between the SignON Mobile App and the SignON Orchestrator, while the AsyncAPI[17] section refers to the description of the communication between the SignON Orchestrator and the SignON Dispatchers.

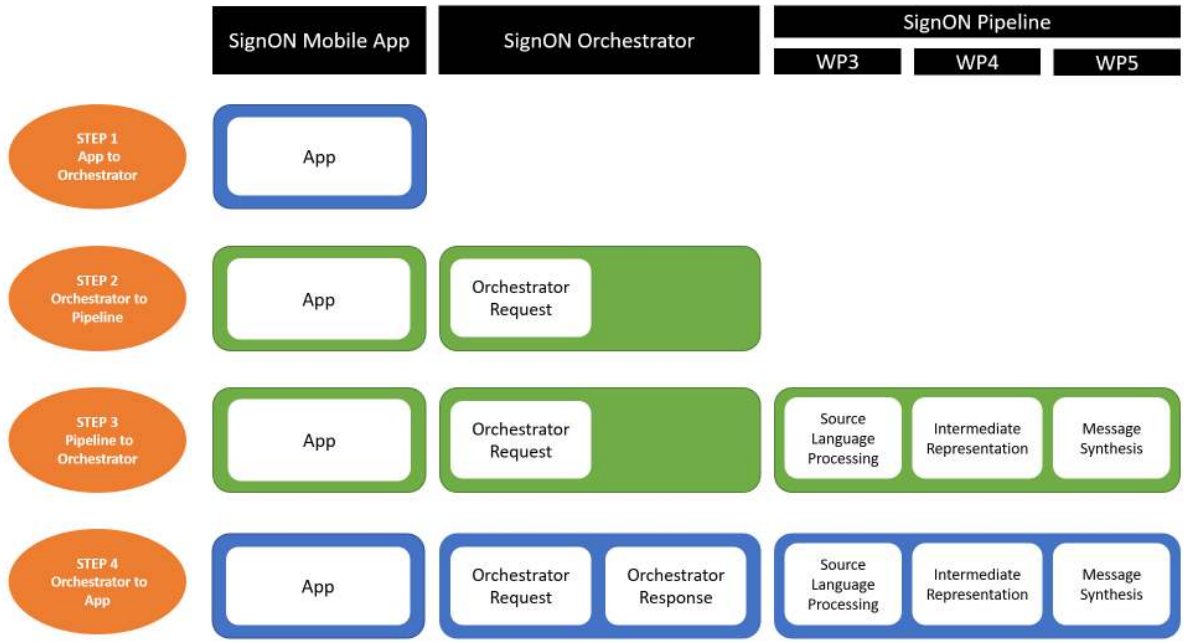


Figure 3 Message Composition (Blue Boxes: OpenAPI, Green Boxes: AsyncAPI)

In Figure 3 are shown the main fields of the message and the four steps of the "piggyback" approach mentioned in section "2 - Updated Architecture of the Cloud Platform": at first the SignON Mobile App adds the "App" data; then the SignON Orchestrator adds the "OrchestratorRequests" data; next the SignON Pipeline Components add the "SourceLanguageProcessing", the "IntermediateRepresentation" and "MessageSynthesis" data; finally the SignON Orchestrator adds the "OrchestratorResponse" data prior to send the message back to the SignON App.
In the table below (Table 1), these main fields are described in detail.

[16] https://www.openapis.org/
[17] https://www.asyncapi.com/

Field Name	Description
App	Reports the original request data (the ones sent in the cURL request)
OrchestratorRequest	Reports the data appended by the SignON Orchestrator when the request is sent to the first SignON Dispatcher (i.e., SignON Orchestrator version, timestamp of message sending and current MinIO bucket name)
SourceLanguageProcessing	Reports the data appended by the SignOn Pipeline Components called by the first SignON Dispatcher of "WP3 - Source Language Processing" (i.e. components version, timestamp of message reception and specific processing fields)
IntermediateRepresentation	Reports the data appended by the SignON Pipeline Components called by the second SignON Dispatcher of "WP4 - Intermediate Representation" (i.e. components version, timestamp of message reception and specific processing fields)
MessageSynthesis	Reports the data appended by the SignON Pipeline Components called by the third and last SignON Dispatcher of "WP5 - Message Synthesis" (i.e. components version, timestamp of message reception and specific processing fields)
OrchestratorResponse	Reports the data appended by the SignON Orchestrator, once the message is received from the last SignON Dispatcher (i.e. timestamp of message reception)

Table 1 High level view of fields in message

All these fields are shared across OpenAPI and AsyncAPI. Therefore, to avoid error prone redundancies in the documentation and generation of classes, the fields present in the previous list are described in a "shared" schema used by both OpenAPI and AsyncAPI YAML files.

3.1 SignON Mobile App and SignON Orchestrator communication (OpenAPI)

The SignON OpenAPI describes the SignON Mobile App interaction with the SignON Orchestrator, providing these endpoints:

- /message POST:

  This endpoint is called to send a message from the SignON Mobile App to the SignON Orchestrator and to translate it through the SignON Pipeline Components. The following table (Table 2) contains a description of the required fields that shall be sent.

Field Name	Description
sourceKey	Object Name saved on the Object Storage.
sourceText	Text that the user wants to be translated.
sourceLanguage	Language for the input given by the user.
sourceMode	Mode in which the message is given by the user (e.g. text, audio, video).
sourceFileFormat	Extension for the file that the user is sending.
sourceVideoCodec	Codec used to compress the video.
sourceVideoResolution	Resolution used for the video.
sourceVideoFrameRate	Frame rate used for the video.
sourceVideoPixelFormat	Pixel format used for the video.
sourceAudioCodec	Codec used to compress the audio.
sourceAudioChannels	Channels used for the audio.
sourceAudioSampleRate	Sample rate used for the audio.
translationLanguage	Language for the output that the user wants to receive.
translationMode	Mode for the output that the user wants to receive (e.g. text, audio, avatar).
appInstanceID	Identifier for instance of the SignON Mobile App.
appVersion	Version number related to the SignON Mobile App.
T0App	Timestamp relative to when the message has been sent from the SignON Mobile App to the SignON Orchestrator.

Table 2 fields composing App

The response returned by this endpoint contains all the message fields previously listed in the section “3 - API” introduction (see Table 1).

- /inference-storage-auth POST:

  This endpoint is called to request a temporary pre-signed URL to upload a file to the Object Storage. The following table (Table 3) contains the description of the fields that shall be sent from the SignON Mobile App to the SignON Orchestrator to request a temporary pre-signed URL.

Field Name	Description
appInstanceID	Identifier for instance of the SignON Mobile App.
fileFormat	Format of the file to be uploaded in the Object Storage.

Table 3 Fields required for requesting a temporary URL

The Response message sent by the SignON Orchestrator to the SignON Mobile App is composed by the fields in the following table (Table 4).

Field Name	Description
PreSignedURL	Pre-signed URL to upload an object in MinIO.
ObjectName	Name of the object once uploaded on MinIO.

Table 4 Fields in the response after requesting a temporary URL

- /version GET:

  This endpoint is called to check version numbers relative to the SignON Orchestrator, the SignON OpenAPI and the SignON AsyncAPI.
- /status GET:

  This endpoint is called to check whether the SignON Orchestrator component is ready.
- /endpoints GET:

  This endpoint is called to list the different endpoints available in the SignON Orchestrator.

As mentioned in section “2.1 - SignON Orchestrator”, through the use of Swagger Codegen[18], it is possible to generate not only the SignON Orchestrator JAVA classes from the SignON OpenAPI, but also the SignON OpenAPI documentation in HTML or Markdown format as shown in Figure 4.

[18] https://swagger.io/tools/swagger-codegen/

Documentation for SignON Orchestrator OpenAPI Spec

Documentation for API Endpoints

All URIs are relative to *https://api.dev.signon-project.eu/orchestrator/dev*

Class	Method	HTTP request	Description
DefaultApi	**getConsentForm**	**GET** /consent-form	Return Consent Form
DefaultApi	**getDatasetStorageAuth**	**POST** /dataset-storage-auth	Request Credentials for the App to write on Minio
DefaultApi	**getEafFormat**	**POST** /eaf-format	Format data in EAF Annotation Format for ELAN (https://archive.mpi.nl/tla/elan)
DefaultApi	**getEndpoints**	**GET** /endpoints	Request to return all the possible Endpoints
DefaultApi	**getInferenceStorageAuth**	**POST** /inference-storage-auth	Request Credentials for the App to write on Minio
DefaultApi	**getStatus**	**GET** /status	Request status relative to the Orchestrator
DefaultApi	**getVersion**	**GET** /version	Request Version Number related to Orchestrator, Openapi and Asyncapi
DefaultApi	**sendMessage**	**POST** /message	JSON Message Between App and orchestrator

Documentation for Models

- Annotation
- App
- AppToOrchestrator
- EafFormatRequest
- EafFormatResponse
- Error
- Languages
- Metadata
- OrchestratorRequest
- OrchestratorResponse
- OrchestratorToApp
- consentFormRequest
- consentFormResponse
- dataset-storage-auth-request
- inference-storage-auth-request
- inference-storage-auth-response
- status
- version

Figure 4 Screenshot markdown OpenAPI documentation

3.2 SignON Orchestrator and SignON Dispatchers communication (AsyncAPI)

The SignON AsyncAPI has been created in order to define the communication between the SignON Orchestrator and the SignON Dispatchers.

The schemas are the same described in the introduction of section “3 - API” as they are shared also with the SignON OpenAPI.

The same strategy implemented for the automatic generation of Java classes for the SignON Orchestrator has been used also for the SignON AsyncAPI, to enable the aforementioned API-first and fast-prototyping approach. These JAVA classes have been generated using AsyncAPI/generator[19]

[19] https://www.asyncapi.com/tools/generator

through a template called “java-spring-template” (for further details please refer to its official documentation[20]).

Moreover, the AsyncAPI/generator has enabled the generation of the SignON AsyncAPI documentation in HTML or Markdown format as shown in Figure 5.

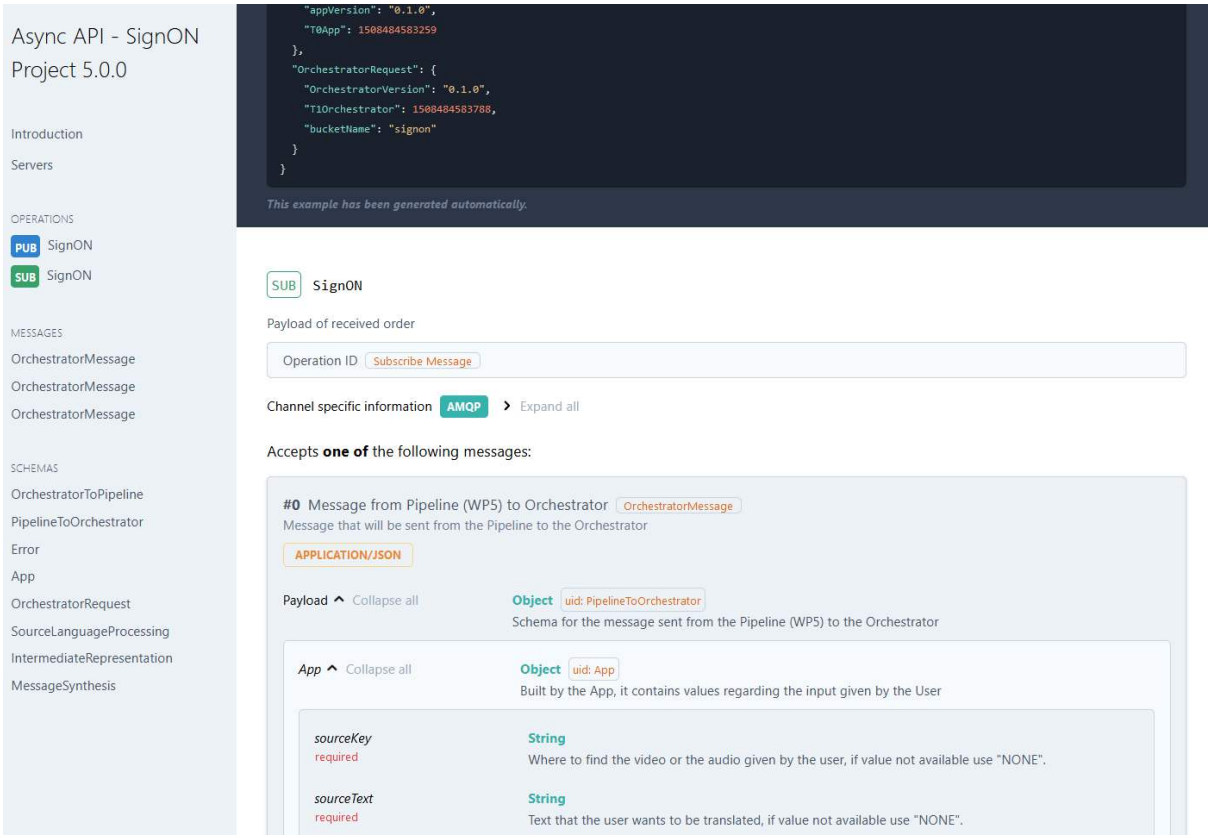


Figure 5 Screenshot HTML AsyncAPI documentation

[20] https://github.com/asyncapi/java-spring-template

4. Infrastructure

The SignON Framework infrastructure is as follows:

- Repository:
    - HP Storageworks SAN platform
    - 64TB storage in RAID 6 with extra hot standby
    - Hardened Microsoft operating system with native ssh daemon
    - NFS subsystem for sharing data
    - 2x 1 GB Network Interface Card
- Hosting platform
    - 32 core Intel Xeon
    - A40 GPU
    - 256GB DDR4 RAM
    - 4TB SSD Storage
    - 15 TB Discs Storage, RAID5
    - 4x 1GB Network Interface card

This summary is elaborated in the following subsections.

4.1 Repository

A storage system of 64TB with data is assigned for the SignON server by consortium Partner INT in order to create a private repository in which SL data can be stored for project-internal use. The server has external connections to the hosting system by implementing NFS server capabilities. It is an SFTP based system which features encrypted data transport. SFTP login information has been passed on to all consortium partners that work with the datasets. This repository and the associated data sets are described in deliverable D.3.1 Internal repository with language data resources.

The hosting platform is also located at INT and hosts the central services of the final application. A detailed description of the architecture of the SignON Mobile App can be found in public deliverable D.2.2[21], which also describes which parts are hosted on a central server, such as the SignON Orchestrator, the Message-Broker and several of the language specific analysis and generation components.

[21] D2.2: SignON Services Framework Architecture
https://signon-project.eu/wp-content/uploads/2022/01/SignON_D2.2_Services_Framework_Architecture_v1.0.pdf

4.2 Hardware

The SignON server hardware consists of 2x16 Intel Xeon 6346 cores, 2xT4 NVIDIA GPU, 256GB memory and a mixed storage with both SSD and HDD. Its goldtype Xeon processors facilitate workloads with a lot of contact switching, as experienced with virtualisation techniques like Docker or vmware-ESX type virtualisation.

Storage for the hosting platform is a 15TB storage array with built-in redundancy against hardware error (RAID). RAID storage can be relatively slow[22], though RAID 0 is faster than non-RAID configurations, and RAID1 is used here, so for better performance 4TB fast storage is provided by SSD. Our first intention to use a tiered storage system has been under discussion because a tiered system does not offer advantages in processing large amounts of unique data.

4.2.1 GPUs support

The SignON Framework platform has an A40 GPU, which has more than four times the CUDA cores of a T4 GPU. This provides the raw power and leaves room for expansion should more power ever be needed.

4.3 Operating system

The operating system and subsequent configuration of the software was done in 2 phases. After 2022 the decision was made to remove the ESX layer and continue with Docker as the only virtualisation layer. This decision was mainly prompted by the licence cost for virtualised GPU on ESX. The supporting services like VPN, docker-registry and others were not affected by the change from ESX to docker.

4.3.1 First Development phase

The hardware has been taken up into the INT ESX-based cloud. In the development phase, contributors will have their own VM with Docker stack. The repository is available to each individual VM via an NFS or ISCSI connection. In later stages of development the separate VMs can be consolidated into a single Docker host if the different contributors consider this to be beneficial.

[22] Does RAID slow down performance? - Quora

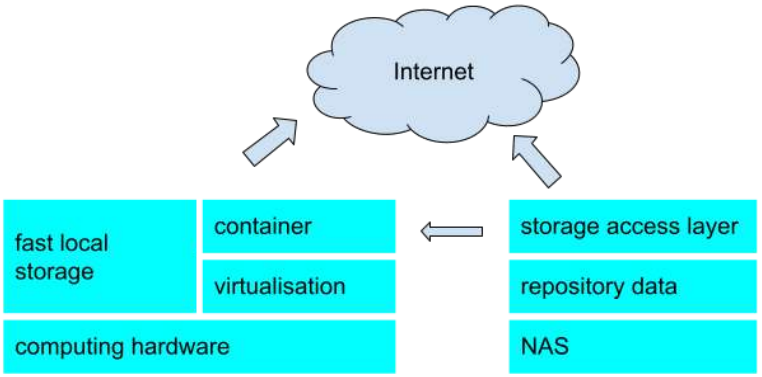


Figure 6 SignON infrastructure

4.3.2 Dev VMs, Production VM

After initial deployment, 1 VM was created as a pilot in the new environment. The first prototype of the SignON Orchestrator was deployed here. The configuration of the VM and especially the firewall and reverse proxy was tuned to the requirement of this first prototype. This machine was used as a template for 4 new VM's for each work packet. The final template was provisioned with 2 CPUs, 16GB memory and 1 TB HD, and 2 nics. 1 nic was exposed to the reverse proxy and 1 connected to the management vpn network (see section "4.4 - Security"). Each developer was provided VPN and SSH access to deploy containers on the VM intended for his work packet. This facilitated development of the separate workgroups without getting in each other's way.

One VM was deployed as a Docker registry to save and maintain the Docker images deployed by the developers. There were initial problems deploying the registry as DNS traffic was not routed to and from the management network. To facilitate safe DNS a forwarding DNS-server was implemented in the management network. The DNS server uses the INT DNS servers as forwarders.

Every VM was connected to the internet by a reverse proxy that proxied an instance of the storage and orchestrator component. The reverse proxy was based on Apache webserver but requirements for the storage component of the orchestrator prompted a change to an NGINX based reverse proxy. This engine could better solve URL discrepancies in the outer and inner situation of the proxied requests using the rewrite module.

4.3.3 Second development phase

In the second development phase, steps were taken to reconfigure the OS layer into a single OS (Ubuntu LTS) with Docker. The single virtualisation layer means that all containers share the same

hardware but Docker was configured with multiple virtual networks to separate development- and production-containers on network level. The development and production container ports were remapped on the reverse proxy accordingly.

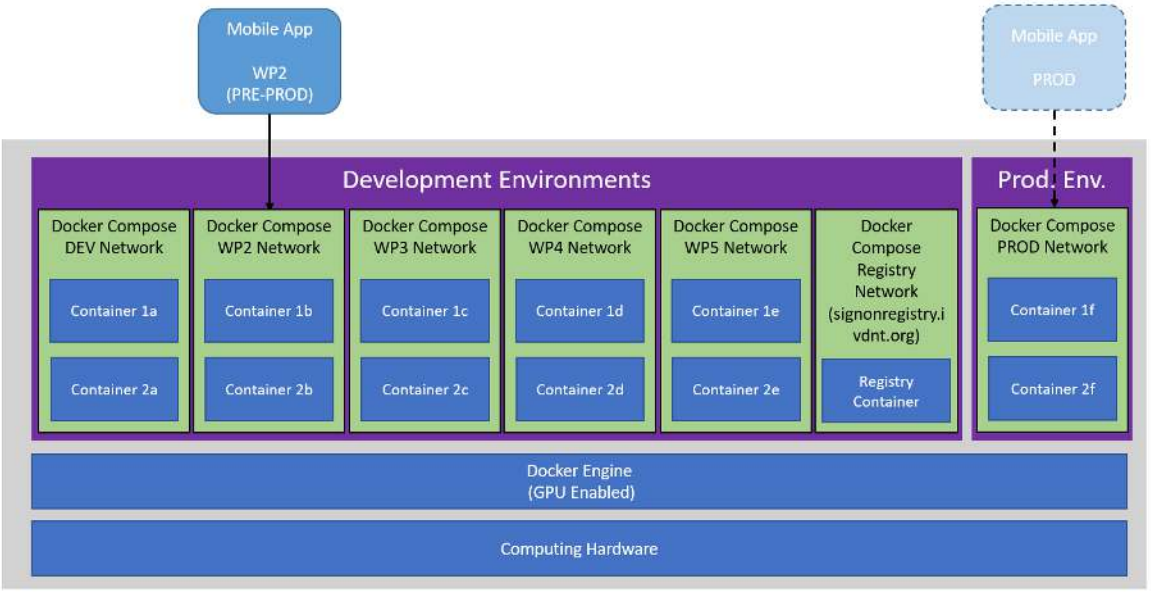


Figure 7 Consolidation of all containers in 1 docker host

4.3.4 T2.4 facilitate data capturing storage

An NFS connection from the docker host to the SignON data repository was mounted in the filesystem to facilitate storage for captured data as described in project requirements described under T2.4.

4.4 Developer access

To facilitate developer access, a VPN access point was created. Using open source software a secure point-to-point connection can be established over the internet, routing traffic from the developers workstation to the management network used to access the VM.

The VPN software used is OpenVPN[23], which has been used successfully for the work-at-home environment of the INT. A central access point to a management network was created instead of a VPN on every individual container, as this is easier to manage.

[23] https://openvpn.net/

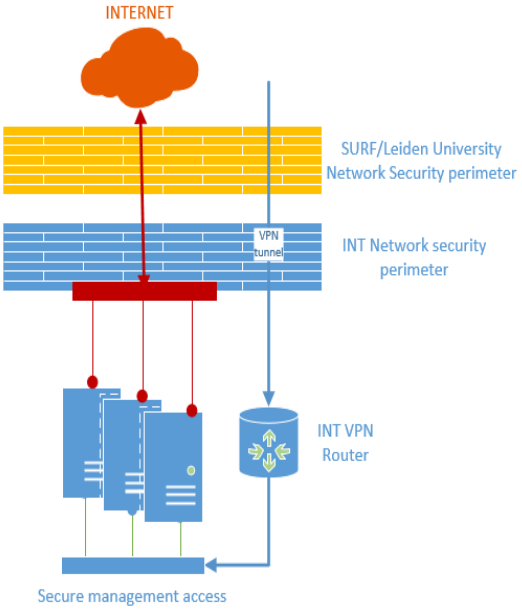


Figure 8 VPN access point in the SignON infrastructure

4.5 Security

Network security on the exposed systems is established by multiple layers of network filtering using firewalls, and exposing as little as possible to the internet to minimise the attack surface. The platforms' operating systems are secured according to best practices, as recommended by Leiden University and SURF, the Dutch academic ICT cooperation organization. Brute force attacks are mitigated against by using the fail2ban[24] system that blocks network addresses after too many failed logins.

Security patches are evaluated on release to determine severity. Important patches are implemented as fast as possible, regular patches are implemented once a month. Regular external scans are done by security teams from Leiden University and SURF, to expose vulnerabilities and check compliance with the security policy of these organisations.

Physical access to the hardware is restricted to INT support personnel in a secured data centre. System backups are made daily. Emergency recovery backups are kept for 2 weeks, long term data backups are kept for 7 years in a daily/monthly/yearly classic tiered backup schedule.

[24] https://www.fail2ban.org/wiki/index.php/Main_Page

INT provides a team of system administrators to support SignON tenants. INT uses the ITIL process library[25] and uses email as their primary means of communication. Service level is best effort and during office hours.

[25] https://www.servicenow.com/lpebk/itil4-guide.html

5. Integration and Deployment

As mentioned in section "2 - Updated Architecture of the Cloud Platform", a number of components developed by different partners are involved in the pipeline that processes a message. To allow a simplified deployment, Docker[26] was chosen as the virtualisation system. For this reason, each component shall be firstly developed, then packed in a container, next pushed to the SignON Container Registry and finally deployed in the SignON Server.

To allow each partner to develop and test its components without interfering with the others, the process has been divided in two different phases:

- **Local integration and testing**: the entire SignON Framework is run on each partner's local development machine, and the relative SignON Dispatcher code can be edited in order to accommodate the calls to the relative dockerised version of the SignON Pipeline Components of the partner. The SignON App requests can be simulated with cURL[27].
- **Deployment and testing**: once the local integration and testing succeed, a new Docker image with the updated version of the SignON Dispatcher is created. Then all the components are deployed to the SignON Server (for further details, please check section "4 - Infrastructure") and then the system is ready to be tested.

Those two phases are described in detail in the following sections.

5.1 Local integration

To start the integration process with the SignON Framework on a local machine, different repositories have been prepared and the steps below shall be followed:

- *Docker Compose Repository*
    1. Clone Docker Compose Repository
    2. Open the Docker Compose[28] file and comment the section of code relative to the relevant SignON Dispatcher (as example, in Figure 9, the section relative to the "signon-wp3-dispatcher" is commented).

[26] https://www.docker.com/
[27] https://curl.se/
[28] https://docs.docker.com/compose/

```
# signon-wp3-dispatcher:
#   image: signonregistry.ivdnt.org/signon-wp3-dispatcher:2.5.0
#   container_name: signon-wp3-dispatcher
#   volumes:
#    - "./signon-wp3-dispatcher/config.yml:/config.yml"
#    - "./minioDownload:/minioDownload"
#   # depends_on:
#   #  - signon-orchestrator
#   restart: always

signon-wp4-dispatcher:
  image: signonregistry.ivdnt.org/signon-wp4-dispatcher:3.2.0
  container_name: signon-wp4-dispatcher
  volumes:
   - "./signon-wp4-dispatcher/config.yml:/config.yml"
  # depends_on:
  #  - signon-orchestrator
  restart: always

signon-wp5-dispatcher:
  image: signonregistry.ivdnt.org/signon-wp5-dispatcher:2.1.0
  container_name: signon-wp5-dispatcher
  volumes:
   - "./signon-wp5-dispatcher/config.yml:/config.yml"
  # depends_on:
  #  - signon-orchestrator
  restart: always
```

Figure 9 Example of Docker Compose File with commented SignON Dispatcher

3. Before Launching all the components it is necessary to login into the SignON Container Registry in order to download the currently working images of the various components.
4. Next, the Docker Compose shall be launched with:

```
docker-compose up
```

5. In the root of the project shall be created a folder called “minioUpload”, where can be placed all the files to be used for the upload during the testing.

- *Dispatcher Repository*
  1. Now that the Docker Compose has been launched, the repository of the SignON Dispatcher, which shall be integrated with the SignON Pipeline Components of the partner, shall be cloned. As previously mentioned, three types of SignON Dispatchers are available:

i. **WP3-dispatcher**: This is the first SignON Dispatcher that receives the message in the "piggyback" approach. It will be connected to the SignON Pipeline Components of the partner related to the "WP3 - Source Language Processing".

ii. **WP4-dispatcher**: This is the second SignON Dispatcher that receives the message in the "piggyback" approach. It will be connected to the SignON Pipeline Components of the partner related to the "WP4 - Intermediate Representation".

iii. **WP5-dispatcher**: This is the third and last SignON Dispatcher that receives the message in the "piggyback" approach. It will be connected to the SignON Pipeline Components of the partner related to the "WP5 - Message Synthesis".

2. Once the corresponding repository is cloned and the code adjusted to accommodate the SignON Pipeline Components of the partner, the "run_dockerised.sh" script will be launched from a terminal inside the SignON Dispatcher cloned repository (otherwise the files are not recognised). This script takes care of running the SignON Dispatcher code in a Docker container, to allow simulating the communication with the other deployed components.

Now the system is ready to be tested locally, as explained in the next section.

5.2 Local Testing

For the local testing, the SignON App requests can be simulated with cURL commands. Those commands will be run from within the SignON Orchestrator container with:

```
docker exec -it signon-orchestrator bash \
        -c "cd minioUpload && bash"
```

N.B. From now on, all the commands are run inside the "minioUpload" folder of the SignON Orchestrator container. If correctly configured, all the test files previously placed in the "minioUpload" folder of the Docker Compose Repository now should be visible (see previous section "5.1 - Local integration").

5.2.1 Test#01: Check Orchestrator and API's Version

This is the first and most basic call that can be invoked, and it works as a sanity check to verify that the SignON Orchestrator is up and running.

Request:

```
curl -X 'GET' 'http://localhost:8080/version'
```

Response example:

```
{
"Orchestrator":"10.0.0",
"OpenAPI":"9.0.0",
"AsyncAPI":"7.0.0"
}
```

5.2.2 Test#02: Request URL to Upload an Object to the storage

As previously mentioned, objects (e.g. *.mp4 files) can be uploaded to the Object Storage by the SignON Mobile App without the need to have a MinIO account, because the SignON Orchestrator can return a pre-signed URL. The generated pre-signed URL is set to expire after 300s (5 minutes) and allows the SignON Mobile App to upload the file within this time frame. The following cURL simulates a SignON Mobile App request of a pre-signed URL that can later be used to upload an object .

Request pattern:

```
curl -X 'POST' \
'http://localhost:8080/inference-storage-auth' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
      "appInstanceID": <APP_INSTANCE_ID>,
      "fileFormat": <FILE_EXTENSION>
}'
```

Request example:

```
curl -X 'POST' \  'http://localhost:8080/inference-storage-auth'\
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
      "appInstanceID": "WP3MODULE",
```

```
    "fileFormat": "mp4"
}'
```

In the above cURL two fields needs to be filled:

- “appInstanceID”, which is a string containing the unique identifier of each instance of the SignON Mobile App; for the tests it can be set to a constant value, without spaces (e.g. “WP3MODULE”), so that in case of troubleshooting is it possible to know who issued the requests;
- “fileFormat”, which indicates the extension of the file to be uploaded (e.g. “mp4”).

Response example:

```
{
"PreSignedURL":"http://minio:9000/signon/WP3Module/2022-11-18_13-33-28_450_e3dc35dd-2326-47c2-8ac0-4fd63155f455.mp4?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=minioadmin%2F20221118%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20221118T133328Z&X-Amz-Expires=300&X-Amz-SignedHeaders=host&X-Amz-Signature=dad439217ccd7eb31a5cd9c2bbc0ff6d9a475f4c7ac8c3892c705eead98fd9fe",
"ObjectName":"WP3Module/2022-11-18_13-31-21_754_7f08e986-5002-42f5-b881-105f35b77a9e.mp4"
}
```

Notice that in the Response above two fields are returned:

- “PreSignedURL”, which is the actual pre-signed URL that is used later to upload the object;
- “ObjectName”, a string uniquely identifying the uploaded object, needed to retrieve the object later on from the Object Storage.

5.2.3 Test#03: Upload File to the Object Storage (Minio)

The previously obtained pre-signed URL can be used to upload our desired object (e.g. “*.wav” file). In this test, two fields need to be filled with specific information:

- the local path of the file to be uploaded.
- the pre-signed URL returned from the previous cURL.

Request template:

```
curl -v -X PUT -T "<LOCAL_PATH_OF_THE_FILE_TO_BE_UPLOADED>" \
-H "Content-Type: application/octet-stream" \
"<PRESIGNED_URL>"
```

Request example:

```
curl -X PUT -T
"/home/corrir/repositories/signon-project/example-files/test_file.mp4" \
-H "Content-Type: application/octet-stream" \
"http://minio:9000/signon/WP3Module/2022-11-18_13-33-28_450_e3dc35dd-2326-47c
2-8ac0-4fd63155f455.mp4?X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=min
ioadmin%2F20221118%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20221118T133328
Z&X-Amz-Expires=300&X-Amz-SignedHeaders=host&X-Amz-Signature=dad439217ccd7eb3
1a5cd9c2bbc0ff6d9a475f4c7ac8c3892c705eead98fd9fe"
```

Response example:

```
HTTP 200 (OK)
```

To check whether the file is actually uploaded, the MinIO web interface can be used :

1. Open a web-browser on the local machine and go to the following link: http://localhost:9001
2. Login in the MinIO web interface with the credential specified in the configuration (see section “2.3 - Object Storage”).
3. Click on “Buckets” in the left panel to have a view of the buckets (see Figure 10).

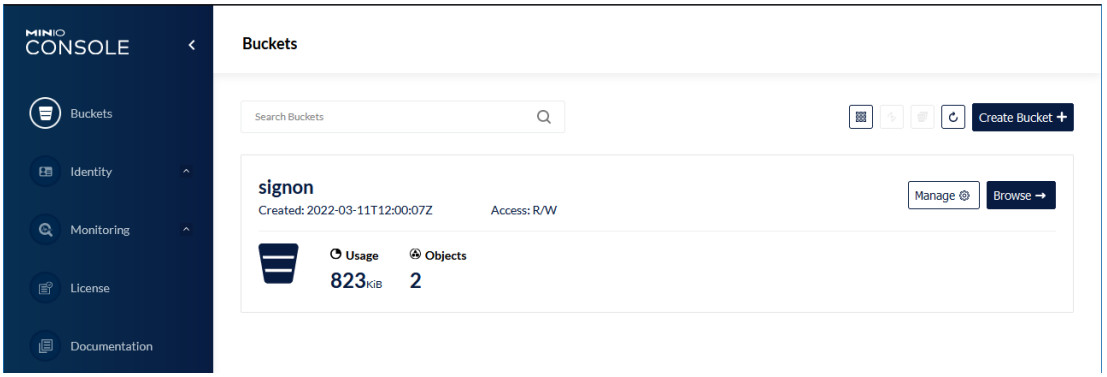


Figure 10 MinIO buckets list in the web interface

- Click on the blue button “Browse” to show all the objects in the bucket (see Figure 11). Please notice the file Object Names, which can be employed to retrieve the files from the storage as shown in the next section.

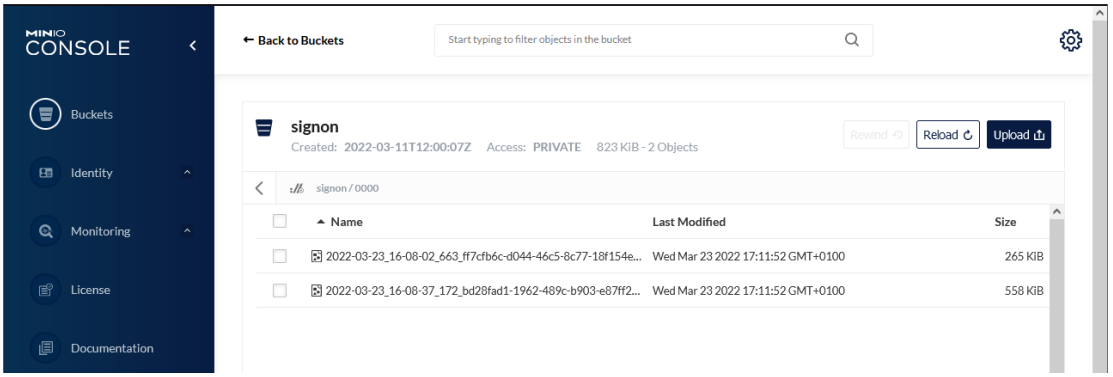


Figure 11 MinIO objects list in the web interface

5.2.4 Test#04: Simulate Message from App through cURL

In this test is simulated the SignON Mobile App request to translate a message encoded in a given language (e.g. English) and mode (e.g. audio) to another language (e.g. Spanish) and mode (e.g. text). This message goes in sequence through the SignON Orchestrator to the WP3, WP4 and WP5 SignON Dispatchers, where it is processed. Then, the processed message is returned back through the SignON Orchestrator to the SignON Mobile App (or, in this case, to the client that issued the cURL request simulating the behaviour of the SignON Mobile App).

Request template:

```
curl -X 'POST' 'http://localhost:8080/message' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
    "App": {
      "sourceKey": "<OBJECT_NAME>",
      "sourceText": "<SOURCE_TEXT>",
      "sourceLanguage": "<SOURCE_LANGUAGE>",
      "sourceMode": "<SOURCE_MODE>",
      "sourceFileFormat": "<SOURCE_FILE_FORMAT>",
      "sourceVideoCodec": "<SOURCE_VIDEO_CODEC>",
      "sourceVideoResolution": "<SOURCE_VIDEO_RESOLUTION>",
      "sourceVideoFrameRate": <SOURCE_VIDEO_FRAME_RATE>,
      "sourceVideoPixelFormat": "<SOURCE_VIDEO_PIXEL_FORMAT>",
      "sourceAudioCodec": "<SOURCE_AUDIO_CODEC>",
      "sourceAudioChannels": "<SOURCE_AUDIO_CHANNELS>",
      "sourceAudioSampleRate": <SOURCE_AUDIO_SAMPLE_RATE>,
      "translationLanguage": "<TRANSLATION_LANGUAGE>",
      "translationMode": "<TRANSLATION_MODE>",
```

```
      "appInstanceID": "<APP_INSTANCE_ID>",
      "appVersion": "<APP_INSTANCE>",
      "T0App": <T0_APP>
   }
}'
```

It's important to keep aligned the name for both the "folder" in which the file is uploaded in MinIO and the folder in which the file is downloaded on our local machine. As shown in the following example, the first part of the "sourceKey" field indicates that the file is uploaded on MinIO in the "folder" named "WP3Module" and then the field "appInstanceID" indicates that the file will be downloaded in the folder "WP3Module" on our local machine.

Request example:

```
curl -X 'POST'   'http://localhost:8080/message' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
    "App": {
      "sourceKey":
"WP3Module/2022-11-18_13-31-21_754_7f08e986-5002-42f5-b881-105f35b77a9e.mp4",
      "sourceText": "NONE",
      "sourceLanguage": "ENG",
      "sourceMode": "VIDEO",
      "sourceFileFormat": "mp4",
      "sourceVideoCodec": "NONE",
      "sourceVideoResolution": "NONE",
      "sourceVideoFrameRate": -1,
      "sourceVideoPixelFormat": "NONE",
      "sourceAudioCodec": "NONE",
      "sourceAudioChannels": "NONE",
      "sourceAudioSampleRate": -1,
      "translationLanguage": "NLD",
      "translationMode": "TEXT",
      "appInstanceID": "WP3Module",
      "appVersion": "0.1.0",
      "T0App": 1508484583259
    }
}'
```

Response example:

```
{
"App": {
  "sourceKey": "
WP3Module/2022-11-18_13-31-21_754_7f08e986-5002-42f5-b881-105f35b77a9e.mp4",
  "sourceText": "NONE",
  "sourceLanguage": "ENG",
  "sourceMode": "VIDEO",
  "sourceFileFormat": "mp4",
  "sourceVideoCodec": "NONE",
  "sourceVideoResolution": "NONE",
  "sourceVideoFrameRate": -1.0,
  "sourceVideoPixelFormat": "NONE",
  "sourceAudioCodec": "NONE",
  "sourceAudioChannels": "NONE",
  "sourceAudioSampleRate": -1.0,
  "translationLanguage": "NLD",
  "translationMode": "TEXT",
  "appInstanceID": "0000",
  "appVersion": "0.1.0",
  "T0App": 1508484583259
},
"OrchestratorRequest": {
  "OrchestratorVersion": "8.2.1",
  "T1Orchestrator": 1668779398904,
  "bucketName": "signon"
},
"SourceLanguageProcessing": {
  "T2WP3": 1668779409795,
  "WP3ComponentsVersions": "NLP:V1.12, SL:V1.16, ASR:V3.4",
  ...
},
"IntermediateRepresentation": {
  "T3WP4": 1668779409796,
          "WP4ComponentsVersions":     "ComponentA:V1.5,     ComponentB:V1.1,
ComponentC:V6.4",
  ...
},
"MessageSynthesis": {
  "T4WP5": 1668779412639,
          "WP5ComponentsVersions":     "ComponentX:V1.2,     ComponentY:V1.6,
ComponentZ:V1.114",
```

```
  ...
},
"OrchestratorResponse": {
  "T5Orchestrator": 1668779412661
}
}
```

The response is structured in different fields as described in section “3 - API” (see Table 1).

5.3 Deployment and Testing

Once the local development and testing has been completed, it’s possible to proceed with the deployment. Firstly, the Docker images of the SignON Dispatchers themself and of the SignON Pipeline Components shall be created and pushed to the SignON Registry. Next, the Docker Compose YAML file shall be updated and launched.

Finally, the testing procedure shown in section “5.2 - Local testing” can be repeated, with the foresight of replacing the local endpoint - i.e. “http://localhost:8080/”, with the endpoint defined by the reverse proxies - e.g. “https://signon.api/orchestrator/” (for further details please refer to section “4 - Infrastructure”).

6. SignON Mobile Apps

In addition to the SignON Dev engineering development test App that is described in D2.3[29], the following 2 user Apps have now been developed and are running on the SignON Framework platform

- SignON Mobile Communications App
- SignON ML MT Training App

6.1 SignON Mobile Communications App

The SignON Mobile App V1.0 is published and available for both Android and iOS mobile devices on the Google Play Store and Apple App Store, as “SignONMobile”. It is described in D2.6 “First release of the SignON Communication Mobile”.

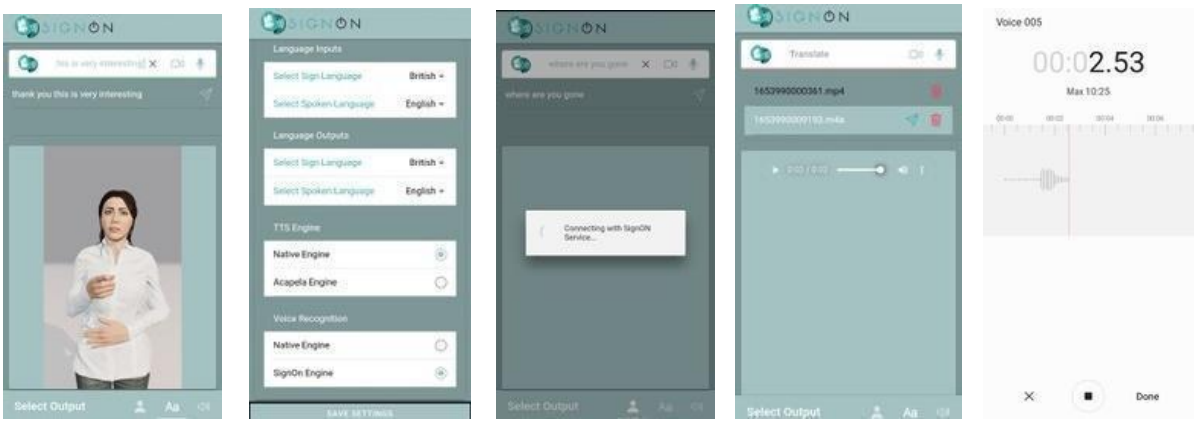


Figure 12 SignON Mobile App V1.0 screens

Use of the App has been designed to be very simple and intuitive, as illustrated in Figure 5 and explained in the “User Guide” in Figure 13.

[29] D2.3 “First release of the SignON Open Cloud platform”, Public Deliverables | SignON Project (signon-project.eu)

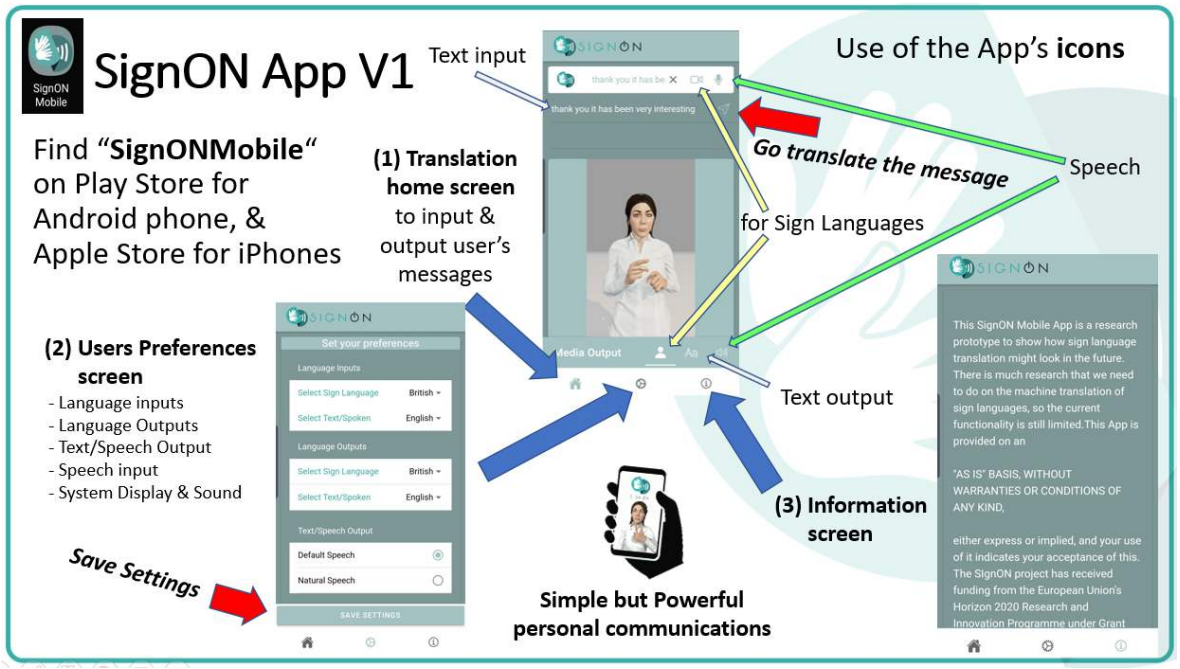


Figure 13 How to use the SignON App V1.0

Currently, the SignON Mobile provides the SignON SL, ASR and MT Framework Services, that are now available from the ongoing WP2, WP3, WP4 and WP5 R&D work,[30] as summarised in the next Figure:

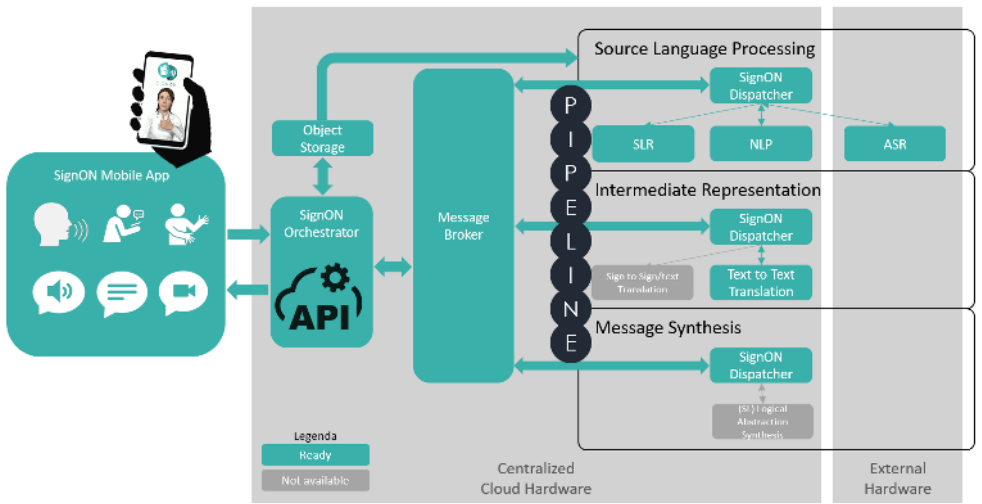


Figure 14 SignON App V1.0 Functionality

Given the available SignON Framework services, the partners agreed that the App V1.0 would demonstrate initial pre-recorded Avatar messages in all 5 SLs, and include the Acapela TTS[31] and the

[30] As described in the most recent deliverables of each of the WPs.
[31] Acapela Group: Text To Speech (TTS) solutions, personalized voices based on neural technology. (acapela-group.com)

initial version of the SignON ASR[32] for "atypical speech" as described in the DoA of the GA. The SLR functionality will be included in the App V2.0 when available later in the project.[33]

6.2 SignON ML Training App

To train and improve the SignON SL and atypical ASR Machine Translation Learning systems, the SignON ML (Machine Learning) mobile App enables SignON Authorised Users to (a) Record Task Messages in SL video or speech audio inputs from predefined use case storyline tasks, (b) Review and Edit their Messages, (c) Tag their Messages with Text translation/ identification, and (d) Upload them to the SignON Server.

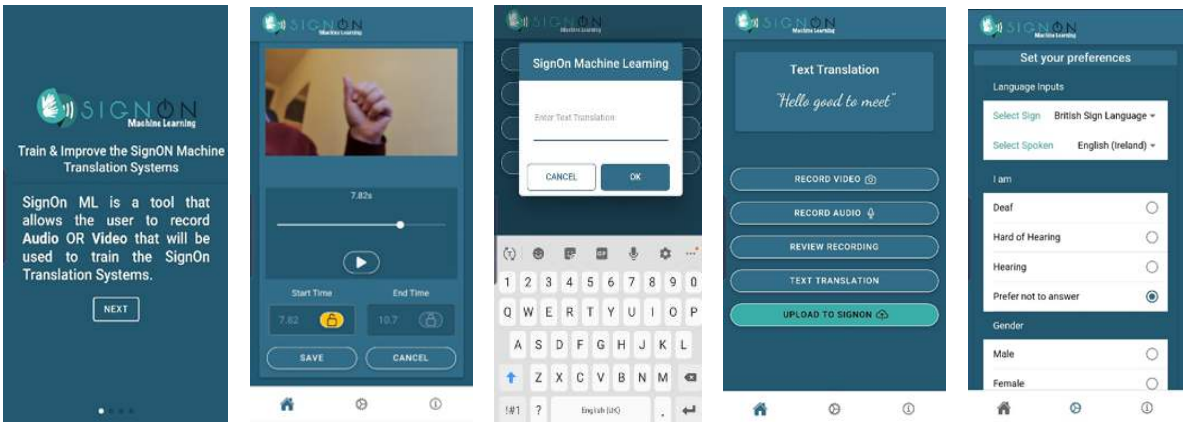


Figure 15 SigON ML Training App

The SignON ML App will be easy and intuitive to use as illustrated in Figure 15 and explained in the Annex "SignOn ML App user Guide". It will be published on the Google Play Store and Apple App Store, for Android and IOS phones, respectively. It will operate in all of languages specified in the SignON DoW (i.e. Text: Dutch, English, Irish and Spanish, SL: British, Dutch, Flemish, Irish and Spanish, Spoken languages: Dutch Northern, Dutch Southern, English (Ireland), Irish, and Spanish), and adheres to all of the SignON ethical and GDPR requirements.

Operation of the SignON ML App and the backend SignON Framework Server will be described in detail in the forthcoming "D2.9 - Final Machine Learning Interface" due by M30 (June 2023).

[32] As described in D3.4 "Automatic speech recognition component and models".
[33] As described in D3.2 "Sign language recognition component and models"

7. Conclusions and Recommendations

The development and production infrastructure (software and hardware) for the SignON Framework service is in place and operational. This deliverable describes the progress of the shared SignON platform, which has developed significantly since D2.3 “First release of the SignON Open Cloud platform” was delivered in January 2022. The platform consists of two separate entities: the repository with reference data and training data, and the platform with processing space to host both developing and developed/production components of the SignON Framework service, software and data.

The internal architecture of the SignON Framework that has been developed is presented with a detailed description of each component and how they communicate with each other. The Framework is composed of different components, namely, the SignON Orchestrator, the SignON Dispatchers, the SignON Pipeline Components (e.g. SLR, NLP, ASR, etc.) and the Object Storage.

The current SignON Framework infrastructure that has now been put in place, including the Repository and Hosting platform, are described. They have been designed and implemented in a way that enables SignON to be a free and open-source MT platform of services, with an open API, between sign language, speech and text in different languages that will go beyond current partial applications. With SignON each user will be unrestricted by the source and target modalities and languages and can choose their preference via the mobile App’s UI, a lightweight interface that features an intuitive responsive easy-to-use UI, personalised to provide each user with their typical translation languages and modalities, allowing the user to simply modify these as they require, and allowing to train to improve its performance to better meet their needs.

To facilitate Integration and Deployment, the SignON Architecture of the Cloud Platform has been evolved and developed to allow a simplified deployment of the components developed by different partners involved in the pipeline that processes a message. To allow each partner to develop and test its components without interfering with the others, the process has been divided in two different phases: (a) Local integration and testing, and (b) Deployment and testing, which are described.

Finally, two SignON Mobile Apps have been developed in addition to the original SignON Dev engineering development test App that was described in D2.3, to run on the SignON Framework platform. These are the SignON Mobile Communications App and the SignON ML MT Training App.

Their use is described. Operation of the SignON ML App and the backend SignON Framework Server will be described in detail in the forthcoming “D2.9 - Final Machine Learning Interface” due in M30 (June 2023).

Annex - SignON ML App User Guide

A. How to get the SignON ML App

a. To ensure the security and quality of SignON MT training, the SignON ML App can only be used by **Users authorised by SignON**

b. If you are not already a SignON Authorised User, please contact John@mac.ie to become one.

c. Once you are a SignON Authorised User, you can find and install the "SignON ML" App:

i. On the Google Play Store – if you are using an Android phone.

ii. On the Apple App Store – if you are using an Apple phone.

B. How to use the SignON ML App

To make SignON recordings to train its Machine Translation (MT)

a. SignON ML App opening page

Briefly describes app, its purpose and use, and advises you to read the **SignON information Tab**, which is the only active tab until You click "Accept"

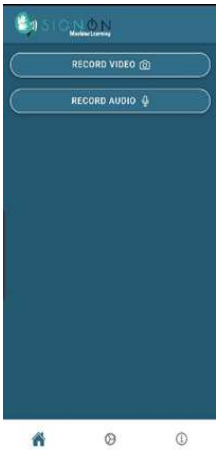
i. To use the App & meet GDPR/Ethical requirements you must

a. choose your text **language** - <Dutch, English, Irish, Spanish> (default is English)

b. **enter your phone number** &

c. **agree** for your Session of Sign Language (SL) or Spoken Language (SpL) Messages & associated metadata to be stored on the SignON server as per the Consent Form. [34]

ii. When you click the "**Accept**" button, you will be allowed to run the SignON ML App by activating its Tabs.



b. SignON ML App Screen Tabs

The SignON ML App screen is organised with 3 tabs at the bottom of screen for App navigation, as follows:

1) Information Tab

a. **How** to use the SignON ML App to record & tag SL or SpL Messages
- This User Guide, with links to a choice of SL translations.

b. **Use** of your session of SL or SpL Messages
- Consent Form, GDPR & Ethics requirements, with links to more extensive information.
- To delete your data & Messages at any time, email your User Token (encoded/anonymised phone number) to signon-rec@adaptcentre.ie

[34] The phone number is requested to re-identify your data on the SignOn server, in case you request to delete files uploaded to the SignOn system. The phone number will not leave your mobile phone and will not be known to us.

c. Information on **SignON**
- Link to [SignON Project - Sign Language Translation Mobile Application (signon-project.eu)]

2) Your Settings Tab

To set up your settings for a Session of SL or SpL Messages to be recorded & uploaded to the SignON server

i. **Sign or Spoken Messages** to be **recorded -** <SL, SpL>
   1. If SL**: SL** - <British, Dutch, Flemish, Irish, Spanish>
   2. If SpL: **SpL**- < Nederlands (Dutch Nth), Vlaams (Dutch Sth), English (Ireland), Irish, Spanish>

ii. **Gender** - <Female, Male, Other, Prefer Not to answer>

iii. **Age** - <18-30, 31-45, 46-60, 60+, Prefer Not to answer>

iv. **I am** - <Deaf, Hard of Hearing, Hearing, Prefer Not to answer>

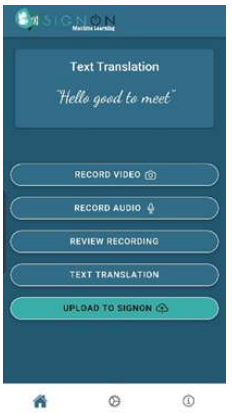
3) Home or Main Screen Tab

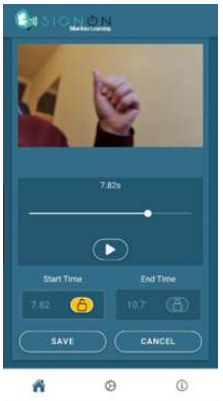
1) Record and review your SL or SpL Message
2) Provide its Text translation
3) Upload the Message to the SignON server
4) Move on to your next SL or SpL message

C. How to use the SignON ML App to make a SignON recording for MT training

1) Record and Review a SL or SpL Message

a) Record a Task Message (SL or SpL) from predefined storyline tasks.
b) Review the Recording of your SL or SpL Message contribution.
c) If you are not happy with the Recording,
   i) Press “Cancel” to delete the Recording
   ii) Go back to step a) & record your Message again.
d) If you wish to take out non-relevant parts of your Recording before & after your Message,
   i) Press “Start Time” & move the slider to position. Press again to lock the start ,
   ii) Press “End Time” and do the same .
e) Review your trimmed Message & if OK, save it by selecting “Save”





2) Add Text of SL or SpL Message

a) Tag your recorded SL or SpL Message with its Task identifier, e.g. H1 for the first task in the Hotel storyline.
b) You may also add a text transcription to your recording. You are free to do this or not

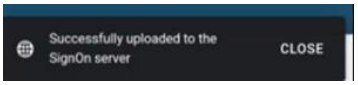
3) Upload the SL or SpL Message

Use the “Upload” button to then upload your SL or SpL Message, its Text translation/ identification & your Session settings metadata, as a **Message Data Package** to the SIgnON Server.

4) <u>Next</u> Message or <u>End</u> the Session

When you get an Acknowledgement from the SignON Server, you can then proceed to

i. Record your next SL or SpL Message by going back to step C.1, or
ii. End your Session by exiting the App.



D. How to ensure the Quality of each SignON recording

- Make the recordings on the basis of the agreed predefined storylines
- Always use a quiet and well lit location with a plain/smooth background.
- When making an SL recording hold the phone steady, or place it on a solid base, especially if it is a two-handed recording.
- Hold the phone at a comfortable distance if you make a SpL recording.
- Keep each SL and SpL message short - no more than a minute.
- Always review the SL or SpL message to ensure that it is clear and correct.
- The SL or SpL message should be immediately tagged with the task identifier by the user. Adding a transcription is optional.
- Check and edit the tagged text before uploading the Message Data Package to the SignON server.

E. Post Hoc Quality Checking & Processing

Each SignON Message Data Package will be stored in a standard database that will be accessed for **Post Hoc Quality Checking & Processing** (editing, transcription, deleting, tagging, etc) only by SignON researchers without revealing any of your personal information.