D1.3 Concept and Architecture Design

2020-1-CY01-KA226-VET-082750

Remote Class System

Authors	Theodoros Giossis
Abstract	The deliverable focuses on the Concept and Architecture Design (frontend, backend), providing sufficient information for the architecture developers. The main objective is to convert the previously identified user requirements (WP1.3) into unitary technical specification. To achieve this, a deep technological analysis of each component, including user interfaces and interoperability, is performed. The technical architecture of the system is established focusing on communication requirements, interoperability protocols, topologies based on power capabilities and constraints, real-life spatial requirements and technological standards of the technologies involved in CLASSY solution.

THE CONSORTIUM

No.	Partner Name		Logo
1	GeoImaging	Geoimaging ltd	
2	AUTH	Aristotle University of Thessaloniki	
3	VS	Virtual Solutions	
4	SVR	SchooVR	

DISCLAIMER

The information in this document is subject to change without notice. Company or product names mentioned in this document may be trademarks or registered trademarks of their respective companies.

Table of Contents

THE CONSORTIUM 2

DISCLAIMER 2

Table of Contents 3

1. Introduction 4

Scope & Objectives 4

Structure of the Deliverable 4

2. Classy Server Overview 5

Server Architecture 5

Requirements 5

Modularity 5

Open Source technologies 5

Scalability & Extensibility 6

Security 6

Privacy 6

3. Classy Architecture 6

Overview 6

Content Management System 7

Database 7

Authentication 7

Roles of the system 7

Real-time features 8

Audio conferencing 8

Turn Relay Server 11

4. REST API schema Classy Platform 13

Identity and Access Control Management Module 13

Experiences Module 22

Sessions Module 25

Chat Module 30

Audio conferencing module 31

5. Conclusions 33

1. Introduction

Scope & Objectives

This deliverable is related to and depicts the output of the Task 1.4.

The task mainly focuses on system Architecture (frontend, backend) based on the initial concept and the changes in it from the research of our Consortium (user requirements analysis - WP1 task 1.3).

To create the system architecture, we carefully examined the concept idea and the various subsystems necessary to achieve the goal.

We mapped each of the subsystems, the communication between them and with the results we started implementing the systems side by side.

To achieve this, a deep technological analysis of each component, including user interfaces and interoperability, is performed, the output of which is recorded in this deliverable.

Structure of the Deliverable

The deliverable is structured as follows:

-At first, the high-level architecture. Based on the high-level architecture, the functional components that comprise the architecture are briefly discussed. In this chapter we also list the main requirements of such a platform.

-Within the next chapter, each one of the components are analyzed separately. We specify the reasons for picking the described subsystems and the special care we took to cover the platform's goals.

-After that, we document the endpoints of the REST interface and the CRUD (create, read, update, delete) operations they perform.

-The last chapter contains the conclusions.

2. Classy Server Overview

Server Architecture

The server is running on a system based on Ubuntu server 20.04.3 LTS that comes with all the security standards and features pre-configured. For the server security, we use UFW, so we can select which ports of the server are exposed, also we have configured the unattended-upgrades feature of Ubuntu server so if a vulnerability comes up the Ubuntu community will patch it automatically.

System administrators have access to the platform only through SSH with personal SSH Certifications. The administrator users don't have the option to read the database data, as the data is encrypted.

For the HTTP server, we use Nginx, which is responsible for the SSL certificates delivery and the reverse proxy that serves our services. The SSL certificate is signed with Let's Encrypt and automatically created from CertBot. Moreover, for the database we used an open source implementation of the MySQL database, the MariaDB, one of the most respectable and secure database implementations. Because the backend is developed in NodeJS, we used the Yarn package manager for the more restricted security than NPM packet manager, as the Yarn uses checksums to verify the integrity of every installed package before its code is executed. Lastly, to manage the NodeJS services, we used PM2, a daemon process manager for NodeJS.

Requirements

The Classy platform is developed around the following technological requirements:

Modularity

Utilize the micro-frontend and micro-services philosophy, every module should be a component of a larger system and operate within that system independently in its operation. Thus, the integrated system will be able to decompose an operating problem into a few, less complex sub-problems, which are usually connected by a simple structure, and independent enough to allow further work to continue separately on each item. This way the effect of an abnormal condition will remain confined to it, or at worst it will only propagate to a few neighboring ones.

Open Source technologies

A primary requirement when implementing the Classy platform concept was the minimization of costs not only for the current integration but also for every future improvement of it. Essentially, such implementations should adopt peak technologies and worldwide accepted and mature standards in order to build on online real-time platform such as Classy. Also, as long as we are based on open source projects, we rely on each project

community for backing us in the early development stage and also to utilize each communities future security updates.

Scalability & Extensibility

Because the Classy platform infrastructure is based on micro-services, we could scale it with many servers behind load balancers. For example, if the platform needs more resources for the Jitsi service, we can deploy various video-bridge instances on the fly to cover demand. These instances can be deployed in any part of the world, leading to reduced latencies for all users.

Security

The system is protected through the HTTP server as it encapsulates the data that is transmitting and receiving with SSL.

The SSL certification is based on the AES-256 encryption protocol, so that means that every data that the server transmits to the user or the user transmits to the server is protected.

The streams shared between users relay directly between jitsi and the users and nothing gets stored onto the server. The connection between Jitsi's subsystem and the end user is encrypted.

The database is accessible only through the API and not exposed to the internet, so there is no way for someone to remotely connect to it.

Privacy

Based on Europe's GDPR agreement, we don't keep user data in the backend such as messages or history of actions in the platform, the only data that is stored is the information that the user has given upon registration in the platform. This information is used only for authentication purposes.

3. Classy Architecture

Overview

For the platform, we have created a RESTful API (Application programming interface) that works independently of the rest of the modules of the platform (backend modules, frontend modules).
The architecture of the platform is based on the micro-services method tailored to our needs. All parts of the system run on separate containers, and we kept the functions as well-defined and small as possible.
This approach gave us the flexibility to choose different types of modules (the best for each case based on our analysis) that work together to build the platform, integrating different types of subsystems in various programming languages to reach the goal.

The following subsystem / technologies were used :

Content Management System

This API was created by using a Headless CMS (Strapi) which we tailored to the needs of the project. Strapi runs on Node.js, making it a perfect fit for our real-time | dynamic data application. The list of Rest endpoints that were created will be presented later in the document.

Database

The Database of our choice is a MariaDB SQL database. All the data gets saved to the DB through the RESTful API and Strapi middlewares.
The database is only accessible internally on the server and nothing is exposed to the internet. Root login onto the db is prohibited and the data are secured.
In any case based on our concept about Classy, the platform will hold as little information as possible for it to be operational, meaning we are not storing any type of sensitive information onto our DB.
Cron jobs were created to backup the DB in regular intervals.

Authentication

To secure the API, we have introduced the usage of JSON WEB TOKENS (JWT), which is a widely accepted and used internet standard for creating data with encryption and signature in JSON format. Through the use of JWT and Role-Based ACL (access control list) we made sure the endpoints of the API (as documented later) can only be used by users with the proper rights to do so.

Roles of the system

To manage all parts of the platform, we created 4 sets of user roles:

- Administrators. This is the role of the creators of the system that have full access to everything.
- Teachers. This is the role of people who will be using the platform to host Classy sessions. The teacher is of course a registered user in the system that got a JWT token corresponding to his/her profile. The teacher was granted the necessary permissions to create and manipulate content that belongs to him/her that fuels the application.
- Students. This is the role of the people who will be attending the sessions of Classy and have registered in the system.
- Public. This is the role of every simple visitor in the platform, including the students who are not registered.

The participants of the Classy sessions can either be registered or not (it depends on teachers wishes) and the Student role exists only to make sure that the Classy platform can log user details and sessions he/she participated in and pass the relevant information to the teachers.
For instance, if a teacher wants to keep track through the platform of the list of students and how they did in each session, then the teacher has the option to add/invite students to register onto Classy. After the procedure, the teacher can see the progress of a student for a series of Sessions.

Real-time features

Classy is a very dynamic application that needs a big amount of data to be exchanged rapidly between the users.
For this reason, we created the Websocket subsystem of our platform to handle the communication between users and our backend.
The subsystem was created using socket.io as the library of choice. In the backend we are serving it through the Strapi node process, so that each user that connects to the app, also connects to the Websocket subsystem thus he/she can receive and send data channels from/to the platform.
We introduced custom computational logic to minimize the required resources, as well as the frontend's computational needs when it comes to real time information through the Websockets. This was needed because for the duration of a Classy session, the user gets notified in real time for the 3d position and rotation angle of all the participants (the user is aware in the frontend of everyone's position in the 3d space) similar to high demand online gaming servers.

Audio conferencing

To cover the needs for audio conferencing features, we installed and configured Jitsi on our server. Jitsi is an SFU (Selective forward unit) system able to handle large amounts of streams with minimal resource usage on the server, especially in our use case where we only have audio and not video feeds.
SFU architecture features the following:

1. The Server receives incoming audio from all endpoints (users in a session)
2. The server sends several copies of uncompressed audio streams of other participants to each endpoint
3. The endpoints merge incoming audio streams

Thanks to this architecture , the user doesn't have to send the same audio feed to each participant (mesh model) , heavily reducing bandwidth needs for both the client and the server.
Jitsi comes with a powerful API to manipulate it's core engine and functionalities. We are using this API through our decoupled frontend to share the streams between the users.
During the development stage in this first phase we have runned many load tests by using special third party services , in order to monitor the system usage for many users.

To accomplish these tests we used third party paid providers for bringing up virtual users alongside custom tests we created by using the library nightwatch.js .
For these tests we created 150 virtual users (processes on various servers around the globe) with various settings for groups of users , like internet speed, connectivity quality , distance from the main server, latency, network type, wireless or cable internet etc. The script execution brought up 150 instances each on a 2-core machine with 2 GB's of ram.
These instances were placed on various corners of the world , in order for us to get a mapping of problems related to distance from the main server.
Each of these instances corresponded to a virtual user that connected to a session in Classy, was receiving audio streams from all the other virtual users and was sending out a pre-recorded stream of audio to the Jitsi videobridge , also each one had different settings applied for connectivity, latency etc.
Each instance also connected to our websocket subsystem , and was sending and receiving dummy data through our servers.
Recording the results gave us a good understanding of the limitations of a non - scaled system that the platform will be based for the prototype phase.
We also had the chance to check bandwidth and cpu usage for the websocket subsystem, even though we knew these subsystems are capable of handling thousands of concurrent users without an issue , even on small machines.
Another outcome of this procedure was that we were able to also debug parts of the frontend and understand the issues that may come onto a dynamic application if 150 streams of audio populate the dom.
A brief report of our findings for server limits can be found in the following table:

Number of users	CPU	Bandwidth	Notes
50	<1% cpu usage	< 1% of available bandwidth on the server approx. 10Mbit	It was difficult to find the spikes of usage since the server barely reacted
100	<1% cpu usage	1% - 2% of available bandwidth, approx 15Mbit.	Still very few resource where used on the server, and the increase in bandwidth was less than 100% of the case of 50 users. This was due to jitsi handling internally the quality of the streams for a bigger number of users
150	2 - 3 % cpu	5-10% of available bandwidth	Even for a single conference with 150

			users (all talking at the same time) the responses we got where very good.

The tests were executed on a server with 1 Gbit bandwidth.

For monitoring resources we used corporate version of pm2 monitoring solution and for the WebRTC SFU load test we used loadero third party provider.

Notes:

The above tests were of course not aligned with real time conditions, since Classy will not allow more than 20 - 30 concurrent users in one session.

These tests were intended to cover edge use cases and to document server resources.

In the scenario that 150 people are talking at the same time (not viable for any conferencing system) we concluded that the issues will come from the client side and not through the server side, since a client needs to have enough bandwidth to receive 149 audio streams, each measuring between 10 - 30 KB/s totaling in about 3 - 6 MB/s so 60 Mbps.

As a side note we should add that the up going audio stream from a client to the server is not of a fixed bit-rate, because it depends on many factors (user is speaking or not, background noise etc.) and for our tests we used pre-recorded speeches of constant flow.

We also need to point out the fact that our frontend application will make use of techniques to limit the bandwidth usage by first allowing the teacher to mute / unmute students and secondly by doing background noise suppression on the client before sending the stream to the server. So to be clearer, if during a Classy session the only one that speaks is the teacher (big probability based on our research) inside a 30 people conference, the number of streams from / to the server drops from 900 (30 up-streams, 29*30 = 870 down-streams, in case of all audios on), to 30 streams (1 upstream, 29 down-streams), so 97% less bandwidth is needed.

The frontend debugging part showed us what we already anticipated based on our experience on similar systems:

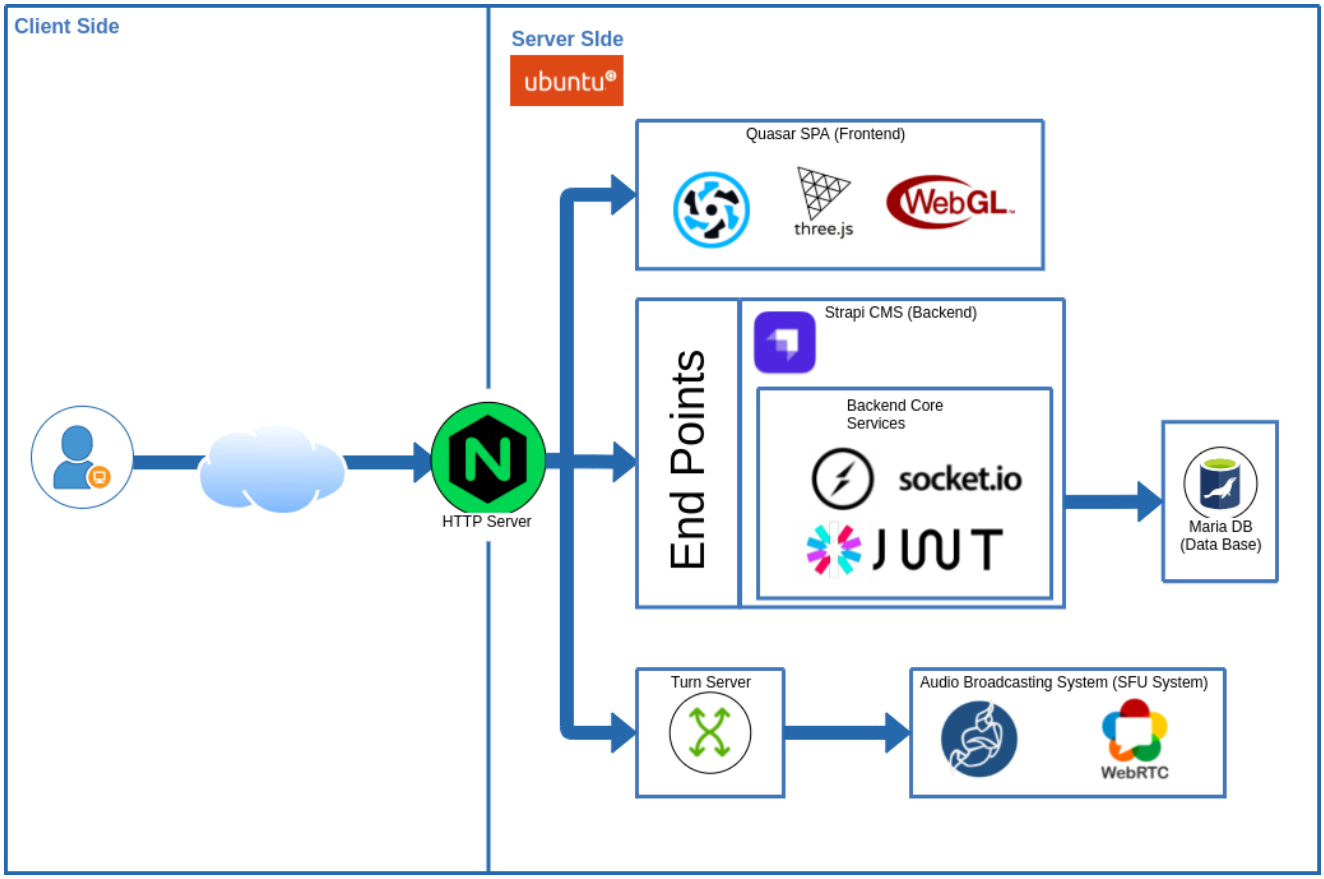
- The Vue JS application we created to test the servers had no problem handling the 150 streams of audio, and receive massive amounts of Websocket data.
- We experienced a very small amount of lost packets for the Audio transmission and only on our big test (150 users, all audios on) of about 2% on the clients with not optimal connectivity. This outcome was a surprise because we anticipated higher packet loss for these amounts of streams and because of the big diversity of client types.
- Our design approach to only have audio streaming was a good decision, since the video part would increase the load by a huge margin.
- 150 open streams transmitting audio is impossible to render since everybody is talking at the same time and the end result is pure noise, with cases of echo (due to the same stream being sent from different clients). Based on our research, anything above 10 concurrent open audio streams makes the

quality of the overall experience bad for the user, especially in the case of audio feedback from the surroundings and lack of proper noise suppression algorithms on the client side. The situation gets better if headphones are used by the participants, so there is no echo coming from the speakers to the microphone.

Turn Relay Server

A big issue in all conferencing systems is the connectivity of the clients that are behind some NAT's and firewalls. Usually these firewalls block the direct communication to ports other than the default ones (443), or they filter UDP based traffic. This situation if not dealt with will lead to these users not being able to receive audio from the platform. The percentage of users that are behind such an environment is between 5 - 15 % of all users.

To make sure we cover these edge cases, we installed a TURN (Traversal Using Relays around NAT) subsystem based on the popular library Coturn. This subsystem takes over when direct communication is not possible between Jitsi Videobridge and the user. In this case, the user sends his / her stream to the TURN subsystem, and then the subsystem relays the stream internally to the SFU.
The TURN server brings an extra overhead to the server, which is approximately +10 % of normal usage, but it is a very important part of any broadcasting solution.



Architecture diagram

4. REST API schema Classy Platform

A REST API (also known as RESTful API) is an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services.

An API is a set of definitions and protocols for building and integrating application software. It's sometimes referred to as a contract between an information provider and an information user, establishing the content required from the consumer (the call) and the content required by the producer (the response). For example, the API design for a weather service could specify that the user supply a zip code and that the producer reply with a 2-part answer, the first being the high temperature, and the second being the low.

In other words, if you want to interact with a computer or system to retrieve information or perform a function, an API helps you communicate what you want to that system, so it can understand and fulfill the request.

You can think of an API as a mediator between the users or clients and the resources or web services they want to get. It's also a way for an organization to share resources and information while maintaining security, control, and authentication determining who gets access to what.

Identity and Access Control Management Module

https://api.classy-project.eu/documentation/v1.0.0#/

The following tables describe all functions we have implemented for this platform module / component.

Short Title	Authenticate Classy users
Description	The Identity and Access Control Management module attempts to authenticate the requesting entity, validating the credentials chosen from the end-user. Basically, we have created an end point that makes a query that searches for the username or email in the database for the collection type of users. If the query finds a candidate, then checks if the hashed password is the same as the database entry, if the password matches, then it returns the message "successful authentication" and the JWT. Otherwise, if the identification isn't found in the query or the password does not match, then it returns the message "Identification or password does not match".
Pre-Conditions	The user must have been successfully registered on the Classy platform.
Post-Conditions	Upon Successful authentication, the user will be properly authorized

	on the system and a unique session identifier will be generated and sent to the user in the form of JWT
Endpoint	POST /auth/local

Request schema	{
	"foo": "string"
	}

Response schema	{
	"code": 0,
	"message": "string"
	}

Short Title	Authorize Classy users
Description	The Identity and Access Control Management module attempts to authorize the requesting entity applying a given role, based on the group in which the user belongs and following the Role-based policy supported from the Classy platform.
Pre-Conditions	The user must have been successfully authenticated within the Classy platform.
Post-Conditions	Upon successful authorization, the user receives the data from the end point of that call.
Endpoint	GET /usersme

Response schema	{
	"id": "string",
	"username": "string",
	"email": "string",
	"provider": "string",
	"confirmed": false,
	"blocked": false,
	"role": {
	"id": "string",
	"name": "string",
	"description": "string",
	"type": "string",
	"permissions": [
	"string"
	],
	"users": [
	"string"
	],
	"created_by": "string",
	"updated_by": "string"

	<pre>},
	"Avatar": {},
	"name": "string",
	"sessions": [
	{
	"id": "string",
	"sessionCode": "string",
	"experiences": {},
	"user": "string",
	"sessionTitle": "string",
	"dateTimeOfSession": "string",
	"created_by": "string",
	"updated_by": "string"
	}
	]
	}</pre>

Short Title	Registration of new Classy user as teachers
Description	The user request to create a new user and the Identity and Access Control Management module checks if the user data and credential is as the data schema and returns if the user is created or something went wrong. Also, the user upon the account registration have received a confirmation email that will expire in short time, so that the account is legit or they have filled the email address with a wrong one
Pre-Conditions	The user don't have a user account
Post-Conditions	The user is register in the Classy platform
Endpoint	POST /auth/local/register

Request schema	<pre>{
	"username": "string",
	"email": "string",
	"provider": "string",
	"password": "string",
	"resetPasswordToken": "string",
	"confirmationToken": "string",
	"confirmed": false,
	"blocked": false,
	"role": "teacher",
	"Avatar": {},
	"name": "string",
	"sessions": [
	"string"
	],
	"created_by": "string",
	"updated_by": "string"
	}</pre>

Response schema	{
	"id": "string",
	"username": "string",
	"email": "string",
	"provider": "string",
	"confirmed": false,
	"blocked": false,
	"role": {
	"id": "string",
	"name": "string",
	"description": "string",
	"type": "string",
	"permissions": [
	"string"
	],
	"users": [
	"string"
	],
	"created_by": "string",
	"updated_by": "string"
	},
	"Avatar": {},
	"name": "string",
	"sessions": [
	{
	"id": "string",
	"sessionCode": "string",
	"experiences": {},
	"user": "string",
	"sessionTitle": "string",
	"dateTimeOfSession": "string",
	"created_by": "string",
	"updated_by": "string"
	}
	]
	}

Short Title	The teachers invite users as students.
Description	From the Identity and Access Control Management module, the teacher can invite users to create an account as students.
Pre-Conditions	The teacher is register and authenticated.
Post-Conditions	The teacher has invited new students
Endpoint	POST /invite

Request schema	{
	"email": "string"
	}

Response schema	{
	"code": 0,
	"message": "string"
	}

Short Title	Users update profile
Description	The user can update their personal info and user account credentials. More specific, they can change: ● password ● full name
Pre-Conditions	The user is register and authenticated.
Post-Conditions	The user has successfully updated their personal information and credentials
Endpoint	PUT /updateme

Request schema	{
	"password": "string",
	"name": "string",
	}

Response schema	{
	"id": "string",
	"username": "string",
	"email": "string",
	"provider": "string",
	"confirmed": false,
	"blocked": false,
	"role": {
	"id": "string",
	"name": "string",
	"description": "string",
	"type": "string",
	"permissions": [
	"string"
	],
	"users": [
	"string"
	],
	"created_by": "string",
	"updated_by": "string"
	},
	"Avatar": {},
	"name": "string",
	"sessions": [
	{
	"id": "string",
	"sessionCode": "string",

	"experiences": {}, "user": "string", "sessionTitle": "string", "dateTimeOfSession": "string", "created_by": "string", "updated_by": "string" }] }

Short Title	Users update avatar
Description	The user can update their avatar based on personal preference. More specific, they can change the following attributes of their character: ● eyes size ● eyes color ● eyes placement ● skin color ● mouth shape ● mouth color ● hair color ● hairstyle ● and they can add some accessories such as hats or glasses
Pre-Conditions	The user is register and authenticated
Post-Conditions	The user is successfully updated their avatars characteristics
Endpoint	PUT /updateme

Request schema	{
	"Avatar": {
	"eyesColor": "string",
	"mouthSize": "string",
	"mouthPosition": "string",
	"mouthRotation": "string",
	"textureUrl":
	},
	}

Response schema	{
	"id": "string",
	"username": "string",
	"email": "string",
	"provider": "string",
	"confirmed": false,
	"blocked": false,
	"role": {
	"id": "string",
	"name": "string",
	"description": "string",

	"type": "string",
	"permissions": [
	"string"
	],
	"users": [
	"string"
	],
	"created_by": "string",
	"updated_by": "string"
	},
	"Avatar": {
	"eyesColor": "string",
	"mouthSize": "string",
	"mouthPosition": "string",
	"mouthRotation": "string",
	"textureUrl": "string",
	},
	"name": "string",
	"sessions": [
	{
	"id": "string",
	"sessionCode": "string",
	"experiences": {},
	"user": "string",
	"sessionTitle": "string",
	"dateTimeOfSession": "string",
	"created_by": "string",
	"updated_by": "string"
	}
	]
	}

Short Title	Student registers from invitation link
Description	The users that they have receive an invitation email from teacher they can register with their credentials such as: ● full name ● email ● username ● password
Pre-Conditions	The user has received an invitation email from teacher
Post-Conditions	The user now is registered as student.
Endpoint	POST /auth/local/register
Request schema	{ "username": "string", "email": "string", "provider": "string",

	"password": "string", "resetPasswordToken": "string", "confirmationToken": "string", "confirmed": false, "blocked": false, "role": "student", "Avatar": {}, "name": "string", "sessions": ["string"], "created_by": "string", "updated_by": "string" }

Response schema	{
	"id": "string",
	"username": "string",
	"email": "string",
	"provider": "string",
	"confirmed": false,
	"blocked": false,
	"role": {
	"id": "string",
	"name": "string",
	"description": "string",
	"type": "string",
	"permissions": [
	"string"
	],
	"users": [
	"string"
	],
	"created_by": "string",
	"updated_by": "string"
	},
	"Avatar": {},
	"name": "string",
	"sessions": [
	{
	"id": "string",
	"sessionCode": "string",
	"experiences": {},
	"user": "string",
	"sessionTitle": "string",
	"dateTimeOfSession": "string",
	"created_by": "string",
	"updated_by": "string"
	}
	]
	}

Short Title	Request Forgot password
Description	If a user has forgotten their credentials, they can request to send an email with steps that specifies how to recreate a new password.
Pre-Conditions	The user has forgotten their credentials.
Post-Conditions	The user now has a new password and can authenticate in the platform.
Endpoint	POST /auth/forgot-password

Request schema	{
	id: "user.id",
	}

Response schema	{
	"code": 0,
	"message": "string"
	}

Experiences Module

The experiences module addresses the demand for listing, editing, create and use experiences from the register teachers. The experiences will contain media like photos, videos, presentations, mini quizzes and texts. There will be experiences that all teachers will have access to, and each one teacher has only the authority to create new experiences, edit and delete them for his own experiences. The actions will be protected from the Identity and Access Control Management Module.

The following tables describe all functions we have implemented for this platform module / component.

Short Title	Lists public Experiences
Description	This endpoint returns all the public Experiences
Pre-Conditions	-
Post-Conditions	The experiences have listed for display and sampling
Endpoint	GET /experiences

Response schema	<pre>[
	  {
	  "id": "string",
	  "Title": "string",
	  "Discription": "string",
	  "Media": [
	  {
	  "id": "string",
	  "name": "string",
	  "alternativeText": "string",
	  "caption": "string",
	  "width": 0,
	  "height": 0,
	  "formats": {},
	  "hash": "string",
	  "ext": "string",
	  "mime": "string",
	  "size": 0,
	  "url": "string",
	  "previewUrl": "string",
	  "provider": "string",
	  "provider_metadata": {},
	  "related": "string",
	  "created_by": "string",
	  "updated_by": "string"
	  }
	  ]
	  }
	]</pre>

Short Title	Teacher creates new Experiences
Description	Teachers can create new personal experiences. The user has to provide the following data: ● title ● description ● data of the experience
Pre-Conditions	The user is registered as teacher and authenticated
Post-Conditions	The user has created a new experience
Endpoint	POST /experiences

Request schema	{
	"Title": "string",
	"Discription": "string",
	"created_by": "string",
	"updated_by": "string"
	}

Response schema	{
	"id": "string",
	"Title": "string",
	"Discription": "string",
	"Media": [
	{
	"id": "string",
	"name": "string",
	"alternativeText": "string",
	"caption": "string",
	"width": 0,
	"height": 0,
	"formats": {},
	"hash": "string",
	"ext": "string",
	"mime": "string",
	"size": 0,
	"url": "string",
	"previewUrl": "string",
	"provider": "string",
	"provider_metadata": {},
	"related": "string",
	"created_by": "string",
	"updated_by": "string"
	}
	]
	}

Short Title	Teacher edits their own experiences.
Description	The teacher update edits the experience and can update: ● title ● description ● data of the experience
Pre-Conditions	The user has to be authenticated as teacher and must have created their own experiences
Post-Conditions	The teacher has successfully edits the experience
Endpoint	PUT /experiences/{id}

Request schema	{
	"Title": "string",
	"Discription": "string",
	"created_by": "string",
	"updated_by": "string"
	}

Response schema	{
	"id": "string",
	"Title": "string",
	"Discription": "string",
	"Media": [
	{
	"id": "string",
	"name": "string",
	"alternativeText": "string",
	"caption": "string",
	"width": 0,
	"height": 0,
	"formats": {},
	"hash": "string",
	"ext": "string",
	"mime": "string",
	"size": 0,
	"url": "string",
	"previewUrl": "string",
	"provider": "string",
	"provider_metadata": {},
	"related": "string",
	"created_by": "string",
	"updated_by": "string"
	}
	]
	}

Sessions Module

The sessions module manages the sessions that teachers create and students attends. The teacher can create a new session based on an experience of the same or public available experiences. The teacher can define the date and the time of that the sessions will be conducted. The teacher can also list their personal sessions, edit, delete or initiate the sessions. In the process of initiation of a specific session, the system will create a new socket room and join automatically the students that waiting in a waiting room. During the session, the teacher can terminate or remove students from the sessions.

The following tables describe all functions we have implemented for this platform module / component.

Short Title	Teacher creates new sessions
Description	Teachers can create new sessions based on a public experience or on a personal one. The user has to provide the following data: ● title ● description ● date and time ● the experience that is based of
Pre-Conditions	The user is registered as teacher and authenticated
Post-Conditions	The user has created the new session.
Endpoint	POST /sessions

Request schema	{
	"sessionCode": "string",
	"experiences": {},
	"user": "string",
	"sessionTitle": "string",
	"dateTimeOfSession": "string",
	"created_by": "string",
	"updated_by": "string"
	}

Response schema	{
	"id": "string",
	"sessionCode": "string",
	"experiences": {},
	"user": {
	"id": "string",
	"username": "string",
	"email": "string",
	"provider": "string",
	"password": "string",
	"resetPasswordToken": "string",

	"confirmationToken": "string", "confirmed": true, "blocked": true, "role": "string", "Avatar": {}, "name": "string", "sessions": ["string"], "created_by": "string", "updated_by": "string" }, "sessionTitle": "string", "dateTimeOfSession": "string" }

Short Title	Teacher edits their own sessions.
Description	The teacher has the ability to edit their own session and can update the following: ● title ● description ● date and time ● the experience that is based of
Pre-Conditions	The user has to be authenticated as teacher and must have created their own session
Post-Conditions	The teacher has successfully updated their session
Endpoint	PUT /sessions/{id}

Request schema	{
	"sessionCode": "string",
	"experiences": {},
	"user": "string",
	"sessionTitle": "string",
	"dateTimeOfSession": "string",
	"created_by": "string",
	"updated_by": "string"
	}

Response schema	{
	"id": "string",
	"sessionCode": "string",
	"experiences": {},
	"user": {
	"id": "string",
	"username": "string",

	"email": "string", "provider": "string", "password": "string", "resetPasswordToken": "string", "confirmationToken": "string", "confirmed": true, "blocked": true, "role": "string", "Avatar": {}, "name": "string", "sessions": ["string"], "created_by": "string", "updated_by": "string" }, "sessionTitle": "string", "dateTimeOfSession": "string" }

Short Title	Teacher start their own session
Description	Teacher can start the session, and in the background a new room socket created and auto joins the users that have joined before and is connected in a waiting room and the users that will connect afterword.
Pre-Conditions	The user has to be authenticated as teacher.
Post-Conditions	The justifiable sessions have now started
Socket.io event	on.sessionCreate

Payload	{
	"sessionId": "string"
	}

Short Title	Teacher ends their own sessions
Description	The teacher has the ability to end the session that is started
Pre-Conditions	The user has to be authenticated as teacher.
Post-Conditions	The justifiable session has now terminated
Socket.io event	on.sessionTerminate
Payload	{

	“sessionId”: “string” }

Short Title	Public join Session
Description	The users that they don't have account, they can attend the session via a 5 digit alphanumeric code, but they have to choose a username and a random generated avatar.
Pre-Conditions	user has not registered yet
Post-Conditions	the user can connect in a session
Socket.io event	on.sessionJoin

Payload	{
	“sessionId”: “string”
	}

Short Title	Registered user join Session
Description	Registered users can join sessions with the 5-digit code, but they don't have to fill information such as username or avatar
Pre-Conditions	user registered and authenticated
Post-Conditions	the registered user can connect in a session
Socket.io event	on.sessionJoin

Payload	{
	“sessionId”: “string”
	}

Short Title	Teacher deletes their own sessions
Description	The teacher can delete the sessions that they have created.
Pre-Conditions	The user has to be authenticated as teacher.
Post-Conditions	The appropriate session has now deleted
Endpoint	DELETE /sessions/{id}
Response schema	{ "code": 0,

	"message": "string" }

Chat Module

Through this operation, users will be able to chat in the session. The history of the conversations, among users, maintain in the system and are available only during session. The user is allowed to read the conversation and send a message. This module works through the Websocket service.

The following tables describe all functions we have implemented for this platform module / component.

Short Title	Users send messages
Description	Users can send messages in the chat messaging system of the session that they have participate
Pre-Conditions	The user is participating in one session
Post-Conditions	Users have sent messages
Socket.io event	on.message

Payload	{
	“message”: “string”
	}

Short Title	User receives messages
Description	Users receive send messages in the chat messaging system of the session that they have participate
Pre-Conditions	The user is participating in one session
Post-Conditions	Users have received messages
Socket.io event	on.message
Payload	[{ “messageId”: “string” “message”: “string” },]

Short Title	User edits messages

Description	Users can edit messages in the chat messaging system of the session that they have participate
Pre-Conditions	The user is participating in one session
Post-Conditions	Users have edit a message
Socket.io event	on.editMessage

Payload	{
	“messageId”: “string”
	“message”: “message”
	}

Audio conferencing module

For communications purposes, the users connect to the Jitsi backend server, so they can send and receive audio streams. Under the hood the Jitsi system consists of many other services like Prosody XMPP server, Jicofo conferences, Videobridges and SFU media server. More specifically :

- XMPP: an open-source alternative to commercial messaging and chat providers that provides signaling systems and real-time multimedia exchange.
- Prosody: an open-source and modern XMPP communication server. It aims to be easy to set up and configure.
- Jicofo: Jitsi Conference Focus is the server that manages the connection between the participants and the videobridges.
- Videobridge: an SFU server that manages all conference media streams.

Although we don't use video transmitting in our platform, basically the system is the same for audio streaming, but we don't use the video attribute.

The following tables describe all functions we have implemented for this platform module / component.

Short Title	User produces audio stream
Description	User connects to the Jitsi to a specific conferencing room
Pre-Conditions	The user is participating in one session
Post-Conditions	User produces audio stream

Short Title	User consumes audio streams

Description	User connects to the Jitsi to a specific conferencing room
Pre-Conditions	The user is participating in one session
Post-Conditions	User consumes audio streams

5. Conclusions

-This deliverable mainly focuses on system core features development and the output of it will provide a ready system for developing the back end side of the platform. Its main objective is to convert the previously identified platform requirements into functional concepts.

To achieve this, a deep technological analysis of each component.

Several fundamental technological requirements are considered in order to build synchronous, integrated solutions and services. To this end, Classy platform include fundamental architectural principles, the main benefit of which is the consistent usage of standards-based architectures, that provide easily modified and expanded functionality and re-engineered integrated services, ensuring that end-users perceive the quality of the services provided, and trust of these that being delivered.

Targeting towards this direction and considering the aforementioned fundamental principles / requirements, components within the Classy layered architecture pattern are organized into modules/components, each module performing a variety of functions based on the user case scenarios within the application.

To these modules, specific functions are added in order to fulfil the requirements of the project. Specifically, these core modules have as follows:

- Identity and Access Control Management Module
- Experiences Module
- Sessions Module
- Chat Module
- Audio conferencing Module

Each module of the architecture has a specific role and responsibility within the Classy platform. Each module in the Classy architecture forms an abstraction around the work that needs to be done to satisfy a particular business request.