Departamento de Eletrónica, Telecomunicações e Informática
Universidade de Aveiro

Informatics Final Project

Informatics Engineering Licenciature

CrowdWire - Massive Online Meetings

by

Bruno Bastos

Daniel Gomes

Leandro Silva

Mário Silva

Pedro Tavares

Supervisor: **Prof. Diogo Gomes**

June 2021

Abstract

Our society is a complex network of people that requires cooperation and socialization between individuals to thrive. With the rise of the Covid-19 pandemic and the need to quarantine, this complex need has been truly affected not only on a personal level but also when it comes to business management and hosting large scale events. As a consequence, companies, organizations and other groups of individuals started adopting remote procedures like video conferences. Even though these videoconference platforms are becoming a more omnipresent means of communication, there are massive limitations to the platforms when it comes to an interaction's ambiance due to the lack of user immersion. This report provides a description about the technology research, the tools and the architecture used for the development of a system that allows users to communicate in a more realistic manner, while providing a scalable infrastructure able to host events for hundreds of people. Consequently we were able to come up with a solution that satisfies most objectives. Unfortunately, we were not able to fulfill the scalability goal needed to host big events. Since the project is limited, there are still a lot of improvements that can be done to achieve its maximum potential.

Keywords

Real-Life Simulation, Browser Game, Proximity Chat, Conference, File Sharing, World Editor, WebRTC, Kubernetes, React, MediaSoup, Phaser3, FastAPI, Scalability

Abbreviations

REST	Representation State Transfer
UI	User Interface
CRUD	Create, Read, Update and Delete
P2P	Peer to Peer
SFU	Selective Forwarding Unit
MCU	Multipoint Control Unit
API	Application Programming Interface
DBMS	Database Management System
WebRTC	Web Real-Time Communication

Contents

Abstract iii

Keywords v

Abbreviations vii

Contents ix

List of Figures xiii

List of Tables xvii

1 Introduction 1
- 1.1 Context . . . 2
- 1.2 Motivation . . . 2
- 1.3 Goals . . . 3

2 State of the Art 5
- 2.1 Gather Town . . . 5
- 2.2 Discord . . . 6
- 2.3 DogeHouse . . . 6

3 Conceptual Modelling 9
- 3.1 Problem . . . 9
- 3.2 Requirements Gathering . . . 9
- 3.3 Terminology . . . 10
- 3.4 Actors . . . 11
- 3.5 Use Cases . . . 11
  - 3.5.1 Guest . . . 12
  - 3.5.2 Registered User . . . 13
  - 3.5.3 World Creator . . . 15
  - 3.5.4 Admin Platform . . . 16
- 3.6 Requirements . . . 17
  - 3.6.1 Functional Requirements . . . 17
  - 3.6.2 Non-Functional Requirements . . . 18

4 Procedure and Implementation **19**
4.1 Technology Architecture . . . 19
4.1.1 React.js . . . 20
4.1.2 Phaser.js . . . 21
4.1.3 FastAPI . . . 21
4.1.4 RabbitMQ . . . 21
4.1.5 Redis . . . 22
4.1.6 PostgreSQL . . . 22
4.1.7 NGINX . . . 23
4.1.8 WebRTC and MediaServer . . . 23
4.2 Deployment Architecture . . . 25
4.3 REST API . . . 27
4.3.1 Authentication . . . 27
4.3.2 Invitation Links . . . 27
4.3.3 Data Storing . . . 27
4.3.4 WebSockets . . . 28
4.3.5 Groups Normalization . . . 28
4.4 Game Engine . . . 30
4.4.1 Architecture . . . 30
4.4.2 Object Detection . . . 30
4.4.3 World Editor . . . 31
4.5 Communications . . . 33
4.5.1 Group Call Approach . . . 33
4.5.2 Protocol . . . 34
4.5.3 Scalability . . . 35
4.5.4 File Sharing . . . 36
4.5.5 Screen Sharing . . . 36
4.6 Kubernetes Infrastructure . . . 36

5 Functionalities **39**
5.1 Guest Functionalities . . . 39
5.1.1 Login . . . 39
5.1.2 Dashboard . . . 40
5.1.3 World Details . . . 41
5.1.4 Join World . . . 42
5.1.5 Join World By Link . . . 43
5.1.6 Proximity Chat . . . 43
5.1.7 Text Chat . . . 45
5.1.8 Conferences . . . 47
5.1.9 Interactive Objects . . . 48
5.1.10 Multi-Platform Compatible . . . 51
5.2 Registered User . . . 51
5.2.1 Register . . . 51
5.2.2 Login . . . 53
5.2.3 Google Authentication . . . 53
5.2.4 Dashboard . . . 54

5.2.5 Edit Account and Profile . . . 55
5.2.6 Create a World . . . 55
5.2.7 World Details . . . 56
5.2.8 Manage World . . . 56
5.2.9 Report World . . . 57
5.2.10 Report Users . . . 58
5.2.11 Generate Invite Link . . . 58
5.3 World Creator . . . 59
5.3.1 Edit World Information . . . 59
5.3.2 Edit the World Map . . . 60
5.3.3 View World Statistics . . . 62
5.4 Admin Platform . . . 62
5.4.1 Platform Statistics and Events . . . 63
5.4.2 Users and User Reports . . . 64
5.4.3 Worlds and World Reports . . . 66

6 Results and Discussion **69**
6.1 Performance Testing . . . 71
6.2 Usability Tests . . . 72

7 Project Management **75**
7.1 Backlog . . . 75
7.2 Code Reviews . . . 77
7.3 Branch Workflows . . . 78
7.3.1 Naming pattern . . . 78
7.3.2 Branch logic . . . 79
7.4 CI/CD Pipelines . . . 79

List of Figures

2.1 Gather Town Logo 5
2.2 Gather Town functionalities 6
2.3 Discord Logo 6
2.4 DogeHouse Logo 7

3.1 Actors 11
3.2 Guest Use Cases(UML Diagram) 13
3.3 Registered User Use Cases(UML Diagram) 14
3.4 World Creator Use Cases(UML Diagram) 15
3.5 Platform Admin Use Cases(UML Diagram) 16

4.1 Technology Architecture Diagram 20
4.2 React.js Logo 20
4.3 Phaser Logo 21
4.4 FastAPI Logo 21
4.5 RabbitMQ Logo 22
4.6 Redis Logo 22
4.7 PostgreSQL Logo 23
4.8 NGINX Logo 23
4.9 MediaSoup Logo 24
4.10 Kubernetes Logo 26
4.11 Deployment Architecture Diagram 26
4.12 The JSON file of the DETI world opened on Tiled 31
4.13 The format of walls that must be followed. The red numbers represent the tiles IDs 33
4.14 Proximity Chat Worst Case Scenario 34
4.15 Communications Sequence Diagram 35

5.1 Guest Login Page 40
5.2 Guest Dashboard Page 40
5.3 Search Worlds Filters 41
5.4 Guest World Details 42
5.5 Choose Username and Avatar 42
5.6 Guest game interface when inside a world 43
5.7 Proximity Chat example where user can only communicate with 1 user 44
5.8 Proximity Chat example where user can communicate with 2 users . . . 44
5.9 Turning off camera example 44

5.10 Video and Audio Settings 45
5.11 Text Chat Example 46
5.12 Text Chat With Nearby User example 46
5.13 Conference ask for permission 47
5.14 Conference granting user permission to talk 47
5.15 Interactable object that provides access to the Whiteboard External Service 48
5.16 Whiteboard External Service example 48
5.17 Interactive object that allows file sharing 49
5.18 Sharing files with other users 49
5.19 Downloading Files from other users 49
5.20 Interactable object that allows users to share their screen 50
5.21 Choosing what screen to share 50
5.22 Seeing the screen that is being shared 50
5.23 Crowdwire running on a mobile phone 51
5.24 User register form 52
5.25 Verification Email Sent Page 52
5.26 Confirm email 52
5.27 User Login form 53
5.28 Sign In with google account 54
5.29 Registered User Dashboard 54
5.30 Registered User change credentials 55
5.31 Registered User edit account information 55
5.32 Create a World 56
5.33 World Details for the World Creator 56
5.34 Roles and permissions management 57
5.35 Adding an example Role 57
5.36 Reporting a World 58
5.37 Reporting a user 58
5.38 Generating a Invite Link 59
5.39 World Creator world details page 59
5.40 Editing World Information 60
5.41 World Editor interface when entering the editor 60
5.42 Hidden and highlighted layers with a grid displayed 61
5.43 World Editor Conference Tab and placement of conferences 61
5.44 World Statistics available to the creator 62
5.45 Admin Dashboard 63
5.46 Admin Platform Statistics 64
5.47 Admin Events Page 64
5.48 Admin User filtering page 65
5.49 User details from Admin perspective 65
5.50 User Reports filtering page 66
5.51 Admin Worlds Page 66
5.52 Admin World Details Page 66
5.53 Admin World Reports Page 67

6.1 Initial Pods on the infrastructure 69
6.2 Media Server Logs 70
6.3 Pod Scaling on the infrastructure 70
6.4 Cluster CPU and RAM Usage with no users present. 71
6.5 Cluster CPU and RAM Usage with over 10 users 72

7.1 Backlog Sprint Calendar 75
7.2 Discord Devops Text Channel Snapshot 76
7.3 Discord Backlog Text Channel Snapshot 77
7.4 Successful Tests and Lint On Pull Request 77
7.5 Example of Code Review 78
7.6 Feature branch examples 78
7.7 CD pipeline steps 79

List of Tables

4.1 User Bandwidth Requirements for a Single Call 33

Chapter 1

Introduction

Since the beginning of the COVID-19 pandemic, everyone's lives had to be adapted to reduce the risk of transmission of the disease from one person to another, this was made by making people quarantined at home not being allowed to leave, most people were even prohibited from going to the workplace. With these restrictions in people's lives and with the human's inherent social characteristics, people rapidly started craving for human interaction and since people were not allowed most types of interaction, there were a couple of solutions, either live with the lack of human interaction or find alternative forms of communication.

The main solution provided was online meetings, ranging from work meetings to a call with friends and even though the amount of companies trying to develop the best possible platform for online-user-interaction is massive, most platforms that allow for communication among the users are limited when compared to real-life communication not fulfilling entirely human's social needs.

1.1 Context

This report aims to go further in detail about the context, implementation, obtained results, and consequent discussion of the elaboration of ***CrowdWire*** - *Massive Online Meetings*, a project done under the context of the *Projecto de Informática* course.

1.2 Motivation

The motivation behind the development of this project was to, through the use of, computers and smartphones along with an internet connection to:

- **Connect people** - As previously mentioned, people are social beings and due to the pandemic scenario we are going through, most people were in need of social activities and still obey to social restrictions imposed. This can be done through text messaging, voice calls, or voice calls alongside video.
- **Immersion** - Another key point mentioned is that most interactions performed online lack life-like attributes.

  There are some relevant aspects that play a significant role in the immersion:

  - Position - Starting from the sense of position and the environment in which the interaction is being carried.
  - Distance - Another relevant aspect is the distance between users. At first, it might not be clear why this distance is an important factor to convey realism, but in a real-life interaction, communicating is different when a person interacts with someone close by or someone far away. When close, it is easier to see and hear, whereas when far away, listening starts to be difficult, or even, impossible to communicate.
  - World Interactability - Another important factor when it comes to a real-life interaction is that, when someone is in a real-life communication setting, most times, people are doing an activity, which means that a lot of interactions revolve around interacting with physical objects.
- **Events** - A big part of human socialization is made in big events such as conferences, concerts and conventions, to name a few. To achieve this type of interaction, special areas could be created where only certain users are heard, yet, those people know there are people attending the event.

1.3 Goals

With this project we were aiming to provide a solution to the inherent need for humans to socialize by providing a way to connect from our homes, this would be accomplished by creating a means to communicate through the internet using a computer or smartphone to take part in calls that allow users to communicate through text, audio and video.

Besides providing a simple form of communicating online we also intended to provide a form of communication that is as close as possible to a real interaction, this would be accomplished by letting a user choose an avatar and when the user joins a world, they would appear as the avatar and would be able to move around the world with that character, this means the user would be able to know where they stand in the map. Since every user would be capable of joining a world with an avatar to represent themselves, they would know where people were by looking for avatars and when in close range be able to talk and see each other.

Another factor mentioned was regarding events and these events can be recreated by creating specific sections that only allow some people to talk.

When it comes to the physical objects needed to perform activities that promote user connection, there would be certain items that resemble real-life objects associated with activities, to encourage engagement and, in some cases, to allow for collaboration allowing the platform to be used with a more formal perspective.

Last, but not least, communications should be able to perform as intended even when dealing with a large number of people, in the order of the hundreds of connected people at once for it to be able to host the types of events mentioned, from the concerts to the conferences.

Chapter 2

State of the Art

In this chapter, we will present our overview on the state of the art as part of the development of this project. Here, we will expose some of the technologies and projects related to our own, *CrowdWire*, and in what sense(s) are they similar to ours.

2.1 Gather Town

The biggest source of inspiration for this project was ***Gather Town***. It served as a model for our project, which means that, when it comes to functionalities and overall ideas, the *Gather Town* platform was our main source of inspiration. With that being said, our primary goal is to provide a similar solution to *Gather Town* but with the particularity of being an OSS (*Open Source Software*).

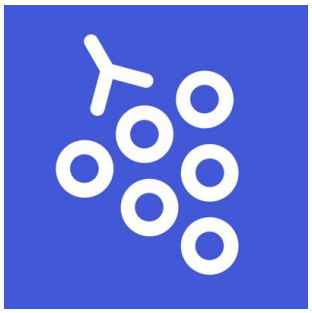

Figure 2.1: Gather Town Logo

As we can see from the image below, *Gather Town* provides a different kind of interaction between users. This kind of engagement is not provided by most of the current video conferencing applications and systems. Every time users get close to each other, a new video (and/or voice) is instantiated, in which the more distant the users are, the more difficult it will be to hear and see other people's cameras. This feature is denominated *Proximity Chat*, one of the key components of our project.

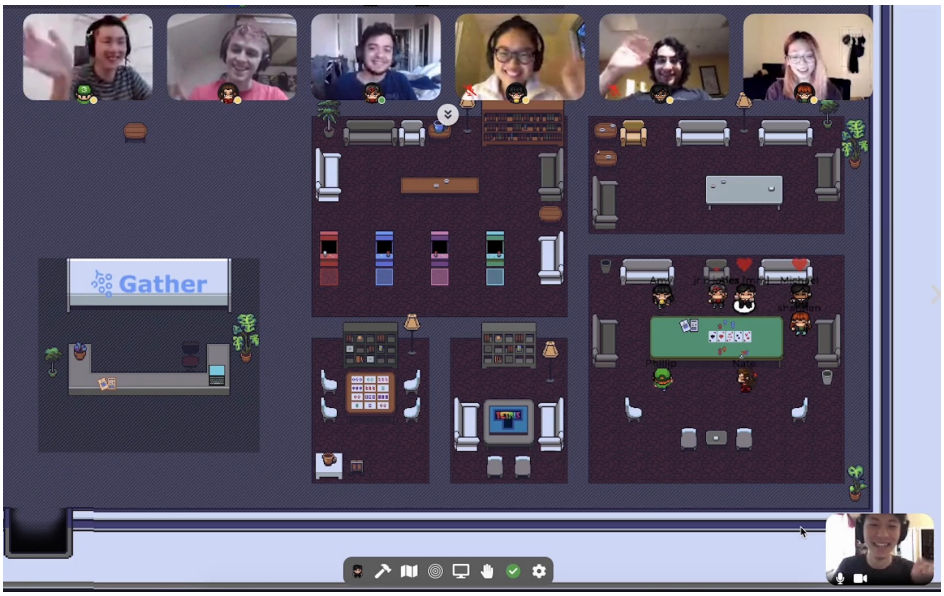

Figure 2.2: Gather Town functionalities

2.2 Discord

During our initial research, *Discord* implemented a new feature that resembles, in part, a key component of our solution, called *Discord Stages*. This feature focuses on delivering a scalable conference environment being capable of hosting a large number of people. The similarities between our solution and the newly implemented one by a product as successful as Discord, gave us both inspiration and motivation to move forward with our ideas.

Figure 2.3: Discord Logo

2.3 DogeHouse

DogeHouse, which by itself was quite a recent project, as it was under development and constantly evolving was also a massive inspiration considering that, just like *DogeHouse*, our application is highly centered on multi-party video conferences. *DogeHouse* has proven to be important guidance on how we should approach our communication

services, thanks to being an open-source project. Later, near the end of the semester, *DogeHouse* ceased development due to the high amount of competition in the area. Yet, we still consider the project to be a valuable product for the development of our project. As mentioned, *DogeHouse* is a recent project and its development was led in a fairly formal way. The developers or the system were not tied to old-fashioned technologies as many others are, which lead us to consider *DogeHouse* architectural structure in terms of communications since we share a mutual goal that being scalability.

Figure 2.4: DogeHouse Logo

Chapter 3

Conceptual Modelling

This chapter provides the reasoning behind the development of this project, the description of the problem we were motivated to solve, the terminologies used as well as some requirements and details about the system.

3.1 Problem

Interaction and communication between individuals have been key parts of our society since the beginning of its existence. Although some can coexist without much interaction, others cannot detach themselves from this necessity. The way humans built their world has communication as one of the foundations. People work and play, and in all of that, they need to interact with each other. The pandemic made it more difficult with the need to quarantine, we became less connected. Technology was an important factor during the early days of the pandemic, with people starting to opt for certain systems and platforms that allowed them to stay in touch with each other. However, the majority of these platforms do not provide a remote interactive environment, side conversations, nor visualization that mimics real-life behaviors. Our solution to this problem is an online website with features that allow users to have a more realistic feeling when interacting with others. Features such as Proximity Chat, where users that are within a close range in the game interface can interact and communicate with one another. Like in the real world, when they distance themselves, they lose the ability to hear and talk back to that person. The world environment can be personalized to match the users' preferences, increasing the overall impressiveness of the platform.

Another major goal was to create a way to host conferences in an online environment. For this particular feature, with the assist of the World Editor tool, a user can change its worlds to feature a conference room where a speaker can be heard by everyone in the room.

3.2 Requirements Gathering

To get a better idea of how to solve the problems mentioned above, we used *Gather Town* as our main source of inspiration. The research focused on understanding the main features and the services provided by this platform. We have also taken into

consideration some other platforms' behavior to achieve a solution that tries to fix some of the flaws that *Gather Town* has, mainly scalability and performance wise. Two projects stand out for different reasons, *Doge House* because it's an open-source platform that focused mainly on the scalability and performance of voice calls, and *Discord Stages* which is a recent platform made by *Discord* that tries to host big-scale conferences.

Secondly, our approach consisted of discussing and brainstorming ideas that we could implement into our system, which included ideas that we could reuse from the related work, as well as the limits and boundaries of our system.

Finally, after gathering this information, we discussed our ideas with our Advisor, Professor *Diogo Gomes*, which was an extremely important step since it gave us a wider vision of how we could tackle some adversities in the development of our system.

3.3 Terminology

During this report, some terms might not be understood without further reading. For that reason, in this subsection, we explain the most used specific terms of our solution.

- **World** — can be described as a "channel" in other platforms such as *Discord*, where users have a game interface of a world built by the one who created the world. Users that join the world can interact with the environment, whether it is with other users or static objects.
- **World Editor** — an essential feature in the application that allows users to change the aspect of their worlds by using a multi-functional drawing interface. With this tool, the creator of the world can place tiles as walls, ground, objects, or conference areas to allow a more realistic and varied environment for the users that may join.
- **Proximity Chat** — refers to a feature that allows users to communicate when inside a range threshold. Leaving that range, they will stop hearing and seeing each other. This is one of the main features that try to mimic a real-life environment. Just like in real life, when we are further away from someone, it becomes harder to talk to or hear them.
- **Role** — this term refers to a set of permissions assigned to a user in a specific world. There is always a default role in a world that will be automatically assigned to every guest and to every new user. This role has some restrictions in terms of the permissions that it can have. Roles can be edited and assigned to registered users, but users are limited to one role each.
- **Room or Conference Area** — both terms are used to refer to an area of the world where it is possible to broadcast voice and video to everyone in that area, independently of the distance. These areas are defined by the creator of the world in the world editor interface and cannot be delimited by the eye in the game interface, meaning that the users cannot see where the area ends.

3.4 Actors

An analysis to the concept of the system that we wanted to build allowed us to identify 4 **actors**:

- **Platform Admin** - Corresponds to the administrator of the platform. This user can use the platform interface to see statistics or take administrative actions in an easier and faster manner.
- **Guest** - User that does not have an account. This type of user usually only wants to get to know the platform or has been invited to participate in scenarios like conferences. This actor has access to a reduced set of functionalities when compared with a User with an account.
- **Registered User** - User that has a registered account. Besides the functionalities of **Guest**, this actor is allowed to create worlds.
- **World Creator** - A special type of **Registered User** that after creating a world, unlocks a set of functionalities in that world.

Figure 3.1: Actors

3.5 Use Cases

In this section we will take a closer look at the *Use Cases* for each *Actor*. We provide an image for every actor use case, as well as a detailed description of what the use case means. Every Use Case marked with a "*" can only be accessed if the user role in that world allows it.

3.5.1 Guest

Starting with the *Guest*, this actor is the base model for each of the other actors, meaning that the others have access to all the *Guest* functionalities.

- **Search Public Worlds** - the user can search for public worlds using filters.
- **Join Worlds** - if the user has permission it can join a world by either searching it in the world's dashboard or by using an invite link.
- **Text Chat*** - the user can send and receive messages from a text chat. Messages can be sent to everyone in that world or the nearby group of users.
- **Broadcast Video and Voice** - the user, when inside a conference area, can share its video and audio to every other user that is inside that area.
- **Interact With Objects** - there can be objects scattered across the world, each one possibly having different functionality. The user can interact with those objects to make use of that functionality. There are 3 different functionalities:
  - **Screen Share** - when interacting with an object that provides this functionality, the user can share its screen with the other users that are nearby.
  - **Share Files** - when interacting with an object that provides this functionality, the user can share a file with another user that interacts with that object. There can multiple users downloading a file from the provider.
  - **External Services** - when interacting with an object that provides this functionality, the user can access external services provided by the platform. There are different types of external services, for example, whiteboard and chess.
- **Proximity Chat** - the user can hear and talk with other users that are nearby.
- **Mute/Deafen Users** - the user can mute himself and other users.

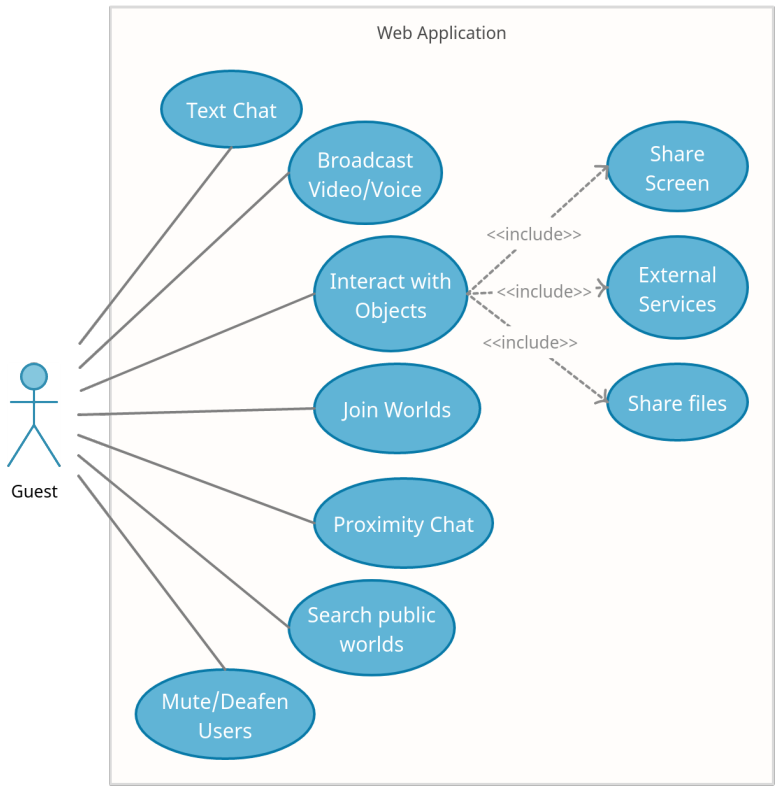

Figure 3.2: Guest Use Cases(UML Diagram)

3.5.2 Registered User

This actor has access to all the functionalities from the previous one. Inside a world there are multiple functionalities that are associated with the role of the user in that world.

- **Edit Account** - since this user as an account, it can be edited through the web interface.
- **View Joined Worlds** - the user history is saved and the worlds that the user has joined can be visualized.
- **Create Worlds** - it can create a world providing its information.
- **Report Users/Worlds** - some users or worlds might have an inappropriate behavior and so the user can report them so that either the platform admins or, in case the report is made to a user, the "moderators" of the world the user was reported in.
- **Invitation Link*** - it can generate a link that can be used by others to join the world.
- **View User Reports*** - the user can see the reports made to users in that world.

- **Ban Users from World*** - the user can ban other users from the world.
- **Assign User Roles*** - the user can create, change permissions or delete the roles for that world.

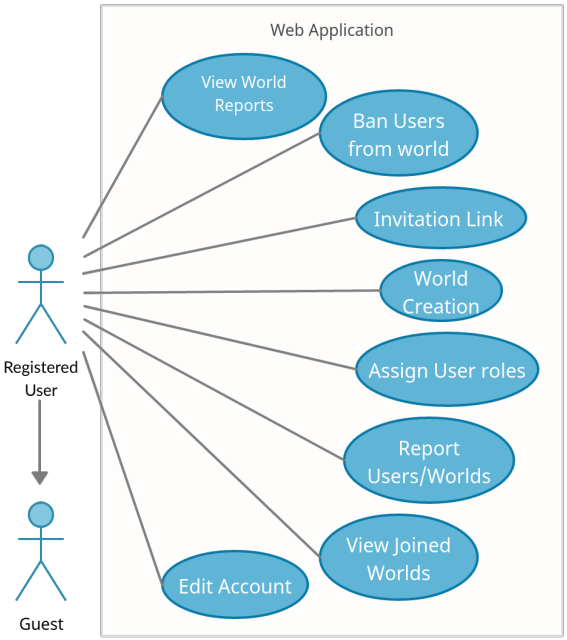

Figure 3.3: Registered User Use Cases(UML Diagram)

3.5.3 World Creator

After the previous actor creates a world, it receives access to extra features related to that world.

- **View World Statistics** - the world creator can view statistics about its world, providing useful information for it to administrate the world.
- **Edit/Delete World** - only the creator of a world can edit or delete it. Using the *World Editor* interface the creator can design the how its world will look like. This interface also allows the creator to set up *Conference Rooms* or *Interactive Objects*.

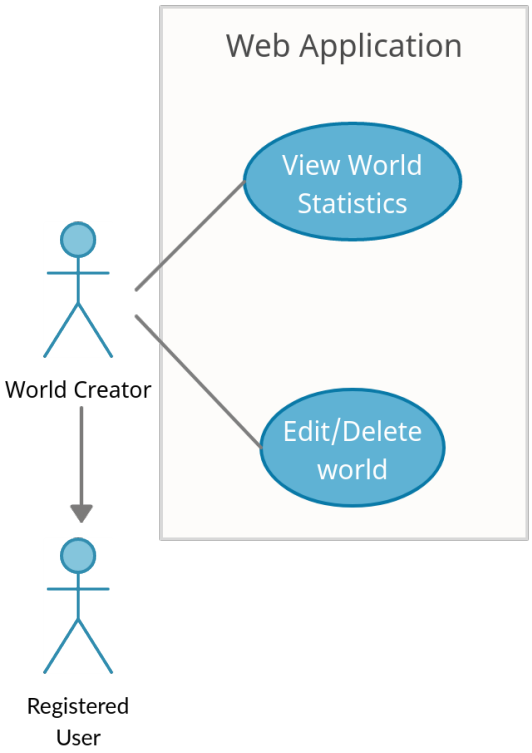

Figure 3.4: World Creator Use Cases(UML Diagram)

3.5.4 Admin Platform

Although this actor has access to many administrative functionalities, we decided that it should also be able to use the rest of the platform normally, so this actor has the same set of features as the *Registered User* and the *World Creator* when inside its world.

- **View Reports** - read access to the user reports and world reports.
- **Ban Users from Platform** - the admins can ban users from the platform and they cannot log in again unless they are unbanned.
- **Ban Worlds** - some world activity might not follow intended behavior, so the admin has permission to ban that world.
- **View Statistics** - it has access to various statistics about the platform and each world and user to help better administrate the platform.

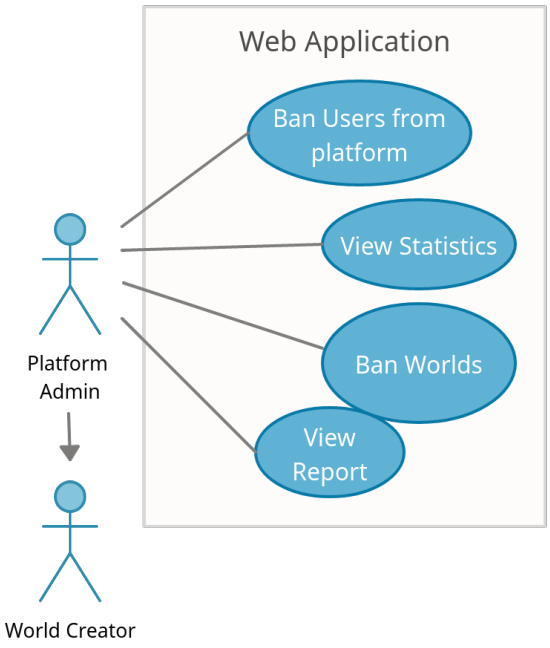

Figure 3.5: Platform Admin Use Cases(UML Diagram)

3.6 Requirements

3.6.1 Functional Requirements

- **Business Rules**
  - Distinct Use Cases for Guests, Registered Users, World Creators and Platform Admins.
  - In each world, every user, registered or not, has a role associated that dictates its permissions in that world. The roles should be editable, however, there must be a default role in every world. The default role cannot have some permissions and Guests are always assigned to that one role. Every world also has an *Administrator* role that belongs to the creator of the world and has access to every permission. This role can also be assigned to other registered users, but those users cannot do everything the creator can do.
- **Authentication and Authorization Level**
  - Users can be logged in or join as guest.
  - Implementation of external sign-in services(Google Sign In).
  - Guests have limited access to functionalities.
  - Any world created can be public or private. Public worlds can be visible and accessed by anyone, while private can only be accessed by an invitation link.
  - Worlds can be configured to not allow guests to enter.
- **Administrative Functions**
  - The Platform Administrators should be able to access the admin dashboard, see the platform statistics and have access to multiple administrative functionalities.
  - Every registered user that creates a world has an administrative role, in terms of business logic, towards it.
  - Registered Users that have a role that provides administrative functionalities within a world can administrate that world.
- **Reporting**
  - Users can report errors in the open-source github project by making an issue with the description of the problem.
- **In-World**
  - Users can communicate using Video, Voice and Text Chat.
  - Users can move freely and interact with the world's objects.
  - Users may have the option to speak to the entire room.
  - The users are associated with roles to be able to perform certain actions dictated by that role.

3.6.2 Non-Functional Requirements

In terms of Non-functional requirements we have to take into account the objectives of the project and what it takes to build the kind of system we want to. These are the following non functional requirements that we gathered:

- **Usability**, as we want to simulate the interactions from real life, our platform needs to be usable and simple to use.
- **Performance**, in order to provide a better experience for end user without any delayed actions.
- **Scalability**, since we need to host conferences with a considerable number of people present in the same call.
- **Availability**, as the system should be always available to answer user's requests on the platform.
- **Recoverability**, which comes along with the previous requirement, as if the system crashes, it needs to have recoverability in order to continue being available for the users.
- **Compatibility**, the platform needs to be compatible with multiple devices.

Chapter 4

Procedure and Implementation

In this chapter, we will explore the details of the implementation built for our system as well as the procedure that led us to it. We will start by discussing our architectural choices and the deck of the technologies used, as well as the importance associated with each. Following this, all of the segments that are present in our architecture will be deeply analyzed, as a way of understanding the mechanisms and inner workflows of CrowdWire.

4.1 Technology Architecture

The figure 4.1 represents the architecture that was defined for our system. The architecture can be separated into three main modules: *frontend*, *backend* and storage. In the *frontend* module, the technologies elected were *React.js* and *Phaser.js*, being this last one embedded onto React. The frontend communicates with the *backend* module through our **REST API**, whose framework is FastAPI. Consequently, to enable the existence of video and voice calls in our system. The API exchanges information with our media server, *MediaSoup*, through a Message Broker, *RabbitMQ*. To store information, we looked forward to using *PostgreSQL* to store persistent data, and also *Redis* to allow caching and save volatile data. Finally, another component that participates in our architecture is *NGINX*, which is mainly used to route HTTP Traffic into our *frontend* and *backend* modules, as well as storing static files.

In the next subsections, each of the technologies mentioned before will be explored in greater detail.

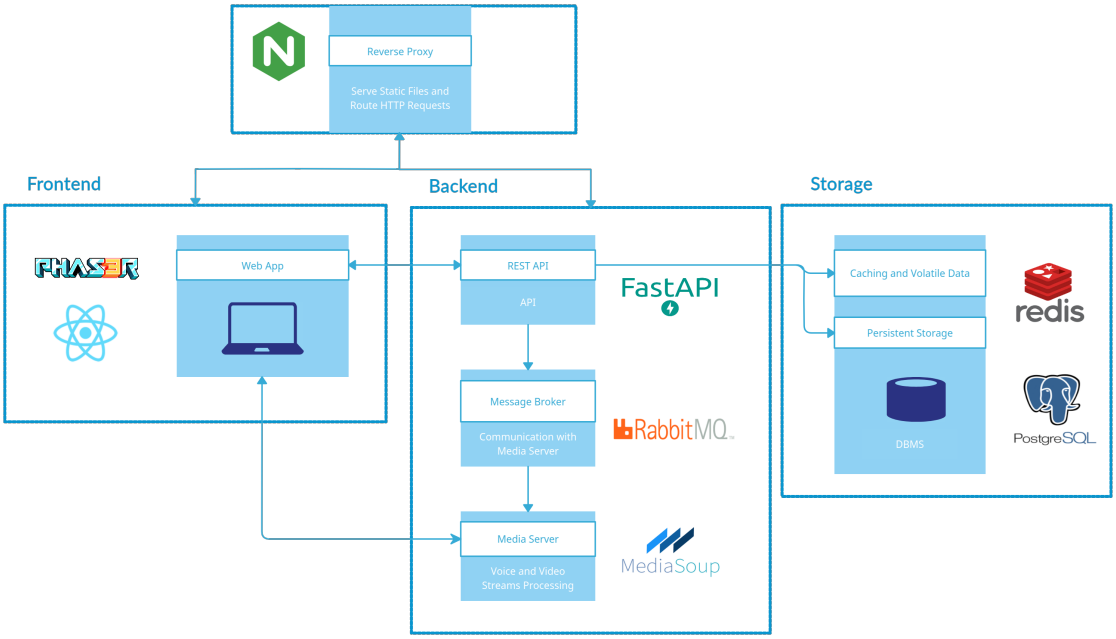

Figure 4.1: Technology Architecture Diagram

4.1.1 React.js

Beginning with the *frontend*, as it was already referred to, this module was built using *React.js*. React is a JavaScript framework that in the last few years has increased its popularity and usage being the most adopted choice on today's market [1] as it allows for the usage of other modules through **npm** (*Node Package Manager*). *React.js* makes creating complex pages a facilitated process since some components needed for it were already developed and are possible to use through these modules. Besides this, another advantage of React is the re-usage of the components written speeding up the process of developing Web Applications and makes the code more maintainable and easier to understand. The previous factors allied to the experience of some of the elements of our team, made us realize that React would be the most suitable technology to develop our web pages.

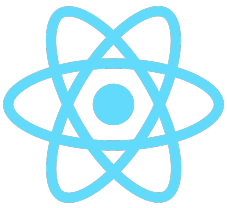

Figure 4.2: React.js Logo

4.1.2 Phaser.js

The use of a gaming framework was essential since an important objective of this project consisted of providing the simulation of real-world behaviors. Therefore, another key part to build our *frontend*, was the 2D gaming framework *Phaser* that allows the development of games on Web Pages, which makes it possible to play through any modern browser, with support for both mobile and desktop [2]. *Phaser* is one of the most used gaming frameworks for the web, supported by extensive documentation and an active community, that is mainly used within JavaScript and TypeScript programming languages. In addition to this, it can be integrated with React.js, an important factor that we took into consideration to choose this technology.

Figure 4.3: Phaser Logo

4.1.3 FastAPI

On the *backend* side of our system, the REST API represents a core section of Crowdwire's Business Logic. It communicates with the frontend through the API, as well as with the Media Server through the RabbitMQ broker. It also has access to storage which was mentioned previously. To do so, we recurred to FastAPI which is a recent Web Framework used to build APIs on top of the Python Programming Language [3]. This framework has earned great popularity in the last year due to its lightweight and high performance when compared to other frameworks like Django or Flask [5], and, as it uses Python, it is an advantage for us since we are familiar with the language and it provides simplicity on both code writing and reading. For those reasons, we ended up picking FastAPI.

Figure 4.4: FastAPI Logo

4.1.4 RabbitMQ

Another component of the system's *backend* is *RabbitMQ*.*RabbitMQ* is a Message Broker that makes use of a diversity of protocols, being the common one, the Advanced Messaging Queuing Protocol, or **AMQP** [6]. *RabbitMQ* makes it possible to send messages from a service, the *producer*, to other ones, commonly known as *consumers* ensuring message ordering and even persistence, in a way that the consumers are not forced to consume the messages immediately. The use of the *RabbitMQ* Broker was

fundamental to exchange information between the REST API and the Media Server, in which both were producers and consumers at the same time.

Figure 4.5: RabbitMQ Logo

4.1.5 Redis

Redis is a Non-Relational in-memory database that can be used for different purposes besides database, such as cache or message broker[7]. Redis was an important component in our architecture's production environment, as we shall see in the next sections of this report, related to the deployment and infrastructure of the System. Besides this, Redis was used to store information related to Guest Users, which is not persisted in any case, and to cache some queries of our relational database, which speeds up considerably the time to do some requests.

Figure 4.6: Redis Logo

4.1.6 PostgreSQL

Persisting Data is fundamental in a great part of today's Software Systems. In this case, we decided to use the relational database *PostgreSQL*, since relational databases ensure the ACID properties, have a high query processing speed and keep data transactions more secure. Besides this, *PostgreSQL* has simple integration mechanisms with our REST API, due to the popularity of this database and the Python drivers available, which contributed to our decision.

Figure 4.7: PostgreSQL Logo

4.1.7 NGINX

Our frontend uses a great number of assets, images, textures, and many other kinds of static files, which increases considerably the loading time of each page on the Web Application. In addition to this, as we may have to deal with heavy amounts of network traffic, the use of a Load Balancer would be important to avoid congestions on the network traffic. This is where *NGINX* comes in since it is a WebServer that can be put on top of our services to receive network traffic from the outside system, and can be used for caching, load balancing purposes or even work out as a Reverse Proxy [8]. *NGINX* was designed to deal with most extreme performance situations such as our case.

Figure 4.8: NGINX Logo

4.1.8 WebRTC and MediaServer

4.1.8.1 WebRTC

In order to simulate a **real environment**, **real-time communication** is **essential**. To achieve this, WebRTC is ideal, since it is an open-source project that supports video, voice, and generic data to be sent between peers. This technology is also available on all modern browsers, as well as on native clients for all major platforms.

WebRTC approaches security from a variety of angles starting from the protocol level to the browser level. For a start, every stream sent between participants is

encrypted with Secure Real-Time Protocol (SRTP). This protocol ensures that the session is **encrypted** and only the two participants can view the contents. To generate the keys to encrypt the session WebRTC utilizes DTLS-SRTP which exchanges keys directly between peers on the media plane.

Additionally, WebRTC requires a secure signaling server to establish peer connections. To meet this requirement, we used the **HTTPS** and **WSS** protocols, which encrypt the contents sent during the signaling process.

At the browser level, there are a variety of factors that ensure **security**. The first feature that browsers implement is requiring the user to give consent to the use of the microphone and webcam before the browser can access the hardware. Secondly, browsers require that a website uses HTTPS to grant access to WebRTC features. Lastly, browsers anonymize the device information until a user has directly granted access.

Once the computer gets the data from the webcam and microphone and it is transcoded, it must then packetize this information to send it to the other people in the call. The fundamental protocol that powers WebRTC is **UDP**. While UDP does not ensure packet delivery or that the packets will arrive in order, it is the best delivery protocol for live streaming because it ensures speedy delivery of packets.

4.1.8.2 MediaSoup

To facilitate the implementation of the WebRTC protocol in our application, we decided to use an open-source project called **Mediasoup**, a multi-party video component with a client and server-side library.

Figure 4.9: MediaSoup Logo

Mediasoup has a **SFU** (Selective Forwarding Unit) topology, where the server routes media around between participants while balancing its limitations with the media inputs it receives. With this approach, participants send their media to the server and receive others' in separate streams, one each.

Pros:

- Server simply routes the streams. Low processing, low latency, high throughput.
- Low CPU usage on the server-side.

- Client decides what streams to receive.
- Client/Server chooses quality for each stream.

Cons:

- Higher download link required.
- No transcoding.

4.1.8.2.1 Basic Concepts:

To better understand how mediasoup works and our approach to it, here are some definitions used from its library:

- A **worker** represents a C++ subprocess that runs in a single CPU core and handles Router instances.
- A **router** enables injection, selection, and forwarding of media streams through Transport instances created on it.
- A **Transport** represents the channel for ICE, DTLS, SRTP. It connects an endpoint with a mediasoup router and enables transmission of media in both directions utilizing Producer, Consumer, DataProducer, and DataConsumer instances created on it.
- A **Producer** represents an audio or video source that will be transmitted to the mediasoup router through a WebRTC transport.
- A **Consumer** represents an audio or video remote source being transmitted from the mediasoup router to the client application through a WebRTC transport.
- A **Device** represents an endpoint that connects to a mediasoup Router to send and/or receive media.

4.2 Deployment Architecture

In this subsection will be explained and discussed the architecture behind the infrastructure implemented and prepared for the deployment on the production environment settled. Once again, scalability was extremely important to take into consideration in the context of this project, especially when it comes to our media server module since one single entity handles all data streams it would be impossible to have the same number of people on the same call as a usual virtual conference. Therefore, we aimed into the horizontal scalability that the *Kubernetes* technology provides. Kubernetes, or usually referred to as *K8S*, is an open-source container orchestration tool that allows simple handling of multiple replicas of the same services and automation on the deployment process.

Each Kubernetes system deployed results in a cluster. A Kubernetes cluster is made of a group of worker machines, nodes, that run containerized applications, isolated

Figure 4.10: Kubernetes Logo

and virtualized at the Operating System level [9]. Even though it runs containerized applications, when we configure our microservices on K8s, we do not work directly with containers but with *Pods*, the lowest abstraction layer on Kubernetes, that represents one or more containers. However, to simplify the management of each Pod, we can make use of *workloads* resources that, in a simple definition, can be referred to as a set of Pods [10]. As it was mentioned, Kubernetes clusters and consequently each resource are isolated, which makes communication from within the cluster and outside the cluster impossible. Therefore, there is also another kind of resource denominated *Services*, which can be configured to expose ports, allowing networking traffic to flow through those ports. It is important to refer that both services and workloads have different types with specific goals and characteristics.

After this brief introduction to Kubernetes, we may now check on the figure 4.11 the Architecture Diagram for our Deployment:

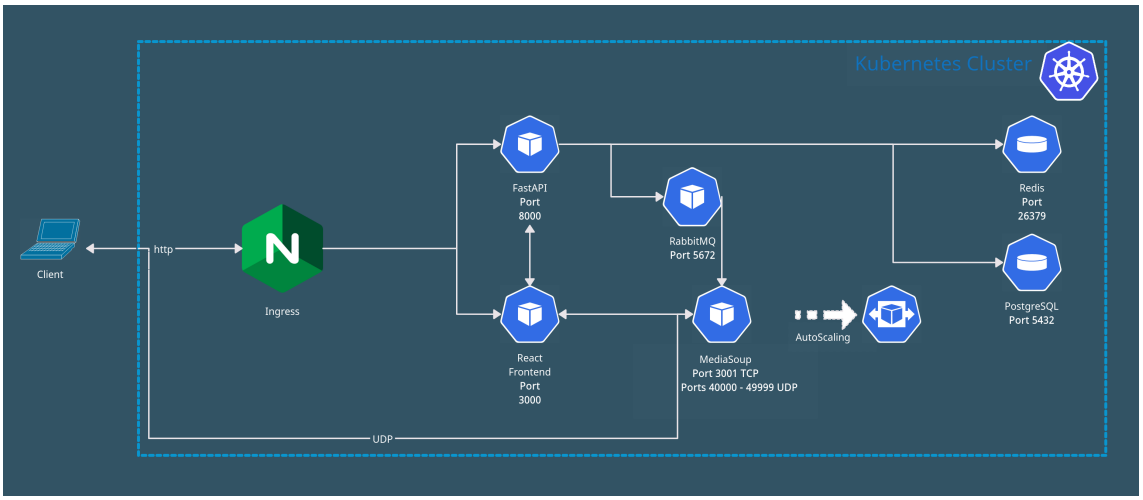

Figure 4.11: Deployment Architecture Diagram

As we may see from the figure above, each of the microservices of our system is packaged as Kubernetes resources inside the cluster, where the HTTP access point is an NGINX Ingress. This means that every HTTP request made to the cluster will pass through NGINX, and proxied to the REST API or the *frontend* Web Application, which improves security since the number of opened HTTP ports are reduced, and uses

of one of the NGINX advantages, that is the Load Balancing mechanisms. Therefore, an Ingress in Kubernetes can be referred to as an entry point to the Cluster. On the other hand, the Media Server needs to communicate directly with the client to handle the data streams using the UDP Protocol, which makes also the Ports in the range 40000-40999 opened for UDP Streams. Our Kubernetes Cluster has been configured using two nodes (machines) that have been provided kindly by our advisor, Prof. *Diogo Gomes*. The first node, having 8 CPU cores and 7GB of RAM, and the second one with 2 CPU cores and 7GB of RAM.

4.3 REST API

As it was already mentioned, to build our REST API, we recurred to FastAPI. However, as FastAPI does not support by itself some functionalities that would be useful to facilitate the development process of the business logic, we resorted to some external libraries, recommended by the official documentation of FastAPI, which were:

- ***Pydantic***, used to parse data incoming in each request or as a response to those requests, by recurring to Python classes [16]. Besides this, Pydantic helps with the restriction on the data posted, by putting validators on it.
- ***SQLAlchemy***, in order to create entities on the PostgreSQL database, by using of **ORM**(Object Relational Mapping)[17].

4.3.1 Authentication

Security mechanisms and policies have to make part of any System nowadays, therefore we were no exception. To do so, each request made to the REST API is secured with a JWT Token. This kind of token is generated and signed by the REST API and provided to the clients to make future requests. Besides this, it identifies each subject and is valid for a limited period. For Guest users, a token was also created, where the subject identification is generated through a unique UUID.

4.3.2 Invitation Links

One of the functionalities of our system consisted in the generation of invitation links to user's Worlds, which will be referred to again with greater detail in Chapter 5. To do this, a different type of token was also created which takes into account who generated the link and for what world is the token valid. This type of token is retrieved on the *frontend* and appended to a valid URL path. Once this URL is introduced on the *backend*, the token is decoded, and the user who has made the request joins the world successfully.

4.3.3 Data Storing

To store information on the *backend* side of the system, we used the libraries ***aioredis*** and ***sqlalchemy***, as it was mentioned, where the first one works out as an asynchronous Redis driver [18], and the second one allows the connection to a PostgreSQL driver.

Every operation on the relational database is done through Python classes on the files on the folder *crud* containing all the business logic involved, each representing an entity of the database model. However, every operation related to guest users is instead stored on Redis, since a Guest User token expires after a few hours when inactive, it would not make sense to store persistent information about Guest Users.

Besides this, as every operation regarding the information of the users inside a world needs to be efficient, the data involved is stored in Redis. Important to refer that the case of registered users, as a way to persist the information, most of this data is also stored on PostgreSQL, which comes along with another use case of Redis on our System, caching database queries. Our caching mechanisms were implemented through a decorator that wraps the function whose result we want to cache, and, consequently, stores as a key the combination between the function name, the arguments passed, and also the database model involved.

The code snippet below shows the example of the use of our cache decorator.

```
@cache(model="World")
    async def get(self, db: Session, world_id: int) -> Tuple[Optional[World], str]:
        world_obj = db.query(World).filter(
            World.world_id == world_id,
            World.status != consts.WORLD_DELETED_STATUS
        ).first()
        if not world_obj:
            return None, strings.WORLD_NOT_FOUND
        return world_obj, ""
```

4.3.4 WebSockets

A WebSocket is a persistent connection between a client and server. They allow the sending of message-based data, similar to UDP, but with the reliability of TCP, meaning that there is no packet loss and the packets arrive in the order they were dispatched. They are usually used when building applications with "real-time" features. [19]

For all these reasons, WebSockets were fundamental whenever the user was inside the world. They handled the protocols behind the user movement, the text messages, the proximity chat, and the communications.

4.3.5 Groups Normalization

The logic behind the proximity chat is done in the server with the REST API when it receives a message with topic WIRE or UNWIRE. According to the topic, one of the algorithms below will be executed, yet this is not necessarily always the case for the WIRE topic, as we will explain.

```
WireUsers(user, users)
1 Normalize groups before addition
  1.1 Get all groups from the users detected
  1.2 From those groups, select all of them where the user can join
  (groups exclusively with users that the user is already connected
  to or that the user wants to connect to)
    1.2.1 If there is none, create a new group
    1.2.2 Else and if there are many, merge all into one
2 Add the users detected and the user to the previous group, if they
are not there yet
```

WireUsers is the algorithm responsible for joining one or more users to the same group. It receives the user that initiated the connection and the newly detected users. To avoid cases where users send erroneous messages claiming to be nearby some other user, this algorithm is only triggered once both users allege to be nearby each other through WIRE messages.

```
UnwireUsers(user, users)
1 Remove the users no longer detected from the user's groups
2 Normalize groups after removal
  2.1 Remove the groups with only one user
  2.2 Remove the groups that are included in other groups
3 Call WireUsers to reconnect the user to the users that should
still be connected
```

UnwireUsers is the algorithm responsible for separating the users. It receives the user that initiated the disconnection and the no longer detected users. This algorithm is triggered for every user, meaning every UNWIRE message, without the consent of the others, as we thought the will to disconnect from anyone should be individual.

On one hand, an important aspect to notice is that these algorithms assume that the state of the world does not change while they are being executed. For that matter, it is necessary to remove the concurrency on those blocks of code by using a Redis key for the world in question as a lock. The code snippet below shows an example of this.

```python
async def wire_players(world_id: str, user_id: str, payload: dict):
    while not await redis_connector.sadd(f"world:{world_id}:lock", 1):
        asyncio.sleep(0)

    # rest of the code

    await redis_connector.srem(f"world:{world_id}:lock", 1)
```

On the other hand, as these algorithms are not restricted to a relationship of one to one users, they are more advantageous when executing for many users at one time, which was also factor considered on the making of object detection (see section 4.4.2).

4.4 Game Engine

4.4.1 Architecture

Phaser 3 is structured with Scenes, and can not run without them. When the Phaser game instance is created, it generates a Scene Manager to control the Scenes, switch between them, and run them in parallel, if required. We can view a Scene as a container specialized in some aspect of the game.

In the making of the Game Engine, we separated the game aspects in 3 Scenes:

- **BootScene** - the first scene that runs whenever the game is instantiated, being responsible for loading all assets, including the tile sets and collection of images that are needed for the world requested, and launching the right scene afterwards.

- **GameScene** - the scene launched by the BootScene when entering a world. It handles everything inside the game, since updating the remote users' position and velocity and sending the own user's movement, initializing the proximity chat protocol whenever a new user gets in range or out of range, interacting with objects with services, etc.

- **WorldEditorScene** - the scene launched by the BootScene when editing a world. It is connected by a Zustand store to the interface in React used to edit the world, so that we can inform the Scene how the user wants to manipulate the world.

4.4.2 Object Detection

We considered some other aspects worth mentioning on the developing with the Phaser framework, and one of those aspects is the object detection. In the game loop, there is a method that searches all physics bodies in a given circular area around the user. With the fps settings we are using, this is repeated 60 times per second, which seems pretty heavy to such a method. However, searching for bodies is rather fast, since Phaser uses R-trees to locate spatially every object, otherwise the search would be O(N).

This search is done to detect the bodies of close ranged interactive objects or remote users. We could perfectly use this in the game loop, but since the interactive objects are static, there is only need to perform this search whenever the user moves, which is the case. For the remote user detection, this search should be performed every time. However, since the group normalization takes more advantage when merging multiple users at the same time, rather than one at the time, and since flooding the REST API with WIRE and UNWIRE messages was a concern (see section 4.3.5 for contextualization), we thought that it was better to delay the remote user detection according to the density of users around, more specifically, delay the detection 300ms per user already in range.

4.4.3 World Editor

4.4.3.1 Loading and Saving

To display 2D maps with tiles on Phaser, there is a method on the Phaser API that allows to convert a map made on Tiled, a 2D level editor that helps developing the content of games, into Phaser objects - `scene.load.tilemapTiledJSON(...)`. Firstly, we have to load the texture images used apart, then we have to pass the JSON file containing the map into this method, and lastly we have to match the map layers with the images they are using. As an example, the figure 4.12 shows the DETI map opened on Tiled, where it is possible to see the tile sets and collection of images on the lower right corner and the map layers on the upper right corner.

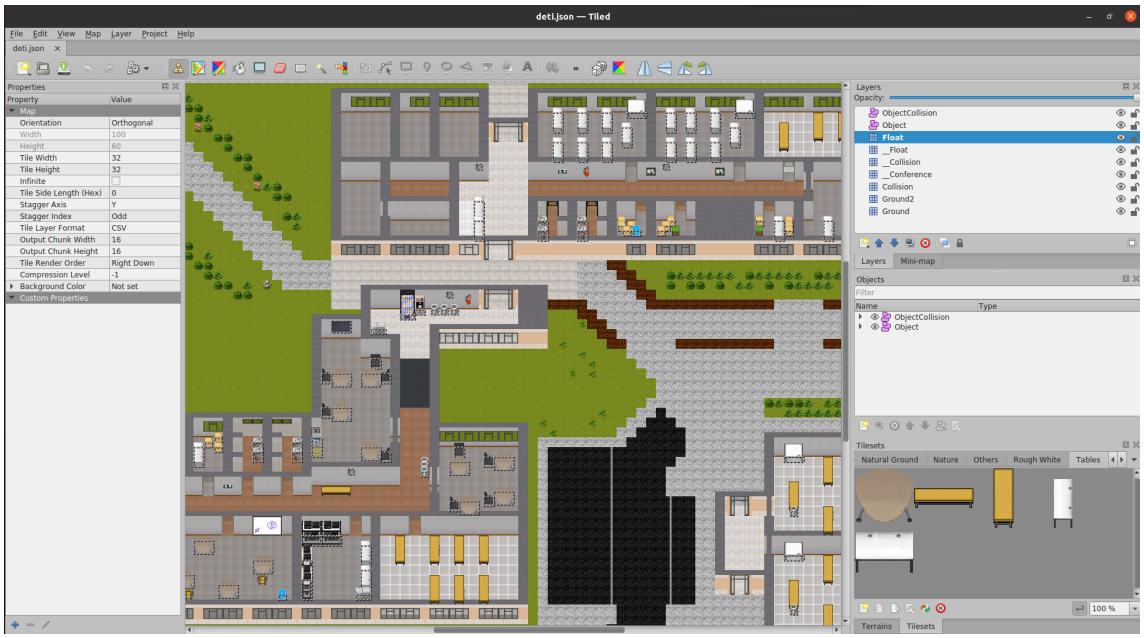

Figure 4.12: The JSON file of the DETI world opened on Tiled

This is very practical when making static maps, yet what we wanted was to load different maps dynamically, meaning that we would have different texture images for different maps that we were incapable of telling in advance. The work around was to read the JSON file alongside and load the images by the paths existent on it. These images are loaded from the FastAPI's static folder, and their paths are all relative to the folder that contains the files of the default maps (this makes it possible for Tiled to find the images when opening the map with it).

Saving the map was also a challenge, as the current version of Phaser does not have the means to revert the conversion of maps on the Tiled format, hence also having to be achieved manually, which required some study on the Tiled JSON structure beforehand.

All these additional methods that we needed from the Phaser API are encapsulated on the **MapManager** class.

4.4.3.2 Including Properties

Having to read the map's JSON file became advantageous for overcoming other incapabilities of Phaser, such as injecting the Tiled properties into the Phaser objects. The exclusive properties which are translated into Phaser are the individual tile properties, all the remaining properties, such as the object, layer and tile sets properties, are ignored. Thus, when reading the file these remaining properties are injected manually or kept aside in the **MapManager** class, where they can be retrieved anytime.

4.4.3.3 Specifying Layers and Objects

Every world is restricted to have the same number of layers with the same names. This is purposely done to specify the behaviour of each layer according to its name and it implies that the user is not able to edit the layers' state (this is not a concern, as there is no interest in letting the user create more layers, nor in redefining them). The structure of the layers can be seen in figure 4.12.

4.4.3.4 Generating Conferences

To create new conferences in the World Editor, some hacks had to be done to permit this. In *Gather Town* the equivalent of conference tiles, which are broadcast tiles, are placed on the map as pink squares with the ID of the conference labelling it. However, Phaser does not support anything alike. We could create 100 images for making the squares with one number as the label and identify each conference, but this would be very restrictive for the user and not very elegant. Therefore, to achieve a similar feature, we used colors rather than number.

Phaser has a tint property on its Tile objects to tinge the tile's texture. Thus, by using a single white tile that could be tinged with any color, and by creating new tile sets that use this single tile and that contain the ID of the conference as a property, we were able to offer the end user a simple way of distinguish conference tiles.

4.4.3.5 Managing the Walls

Besides tiles, objects and conferences, the World Editor has a different type of material than can be placed on the world — walls. This is by far the most complex material, since it must follow a set of rules in order to make it correctly placed and easy to use. To do so, the tile sets that contain the walls have a property that labels them as walls, and are disposed obligatorily as shown in the figure 4.13, so that each type of wall has one window and one door.

With that established, the **WallManager**, the class that encapsulates the management of walls, is capable of checking if it is allowed to place or remove a wall anywhere and handle the placement or removal.

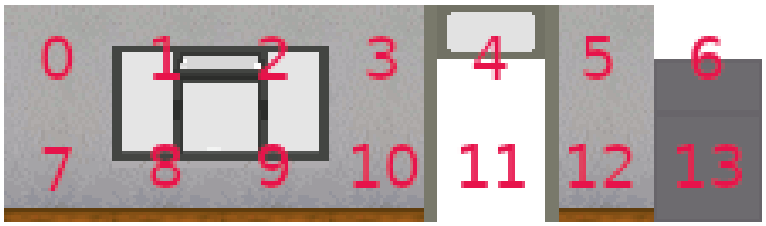

Figure 4.13: The format of walls that must be followed. The red numbers represent the tiles IDs

4.5 Communications

4.5.1 Group Call Approach

When facing a problem like making a real-world alike and interactive environment that can scale, the approach on how group calls are created and destroyed is crucial.

The approach we decided to go for allows the users to join several group calls at the same time, however, it will have to create more WebRTC Transports and upload its stream to all the calls. This has an higher bandwidth cost if a user is in several calls at the same time, but allows to scale better the media server by splitting into several calls, in opposition to having the users in the same group call (even though they wouldn't know that since they would only consume the streams of the ones close to them).

The Figure 4.14 illustrates the worst-case scenario for a user, where he is in the center of 5 other users and in a different group call for each of them due to the proximity chat feature. After realizing this, we tested how much bandwidth is required for a single call.

By analyzing the results from the tests made, displayed in the Table 4.1, if we apply the total bandwidth required to the worst-case scenario with a 10% margin of error, the user requires 2.61 Megabytes per second, which nowadays isn't a hard requirement to accomplish.

Bandwidth Requirements			
Upload	Audio	Video	Total
Maximum Bandwidth (Mb/s)	0.375	0.1	0.475

Table 4.1: User Bandwidth Requirements for a Single Call

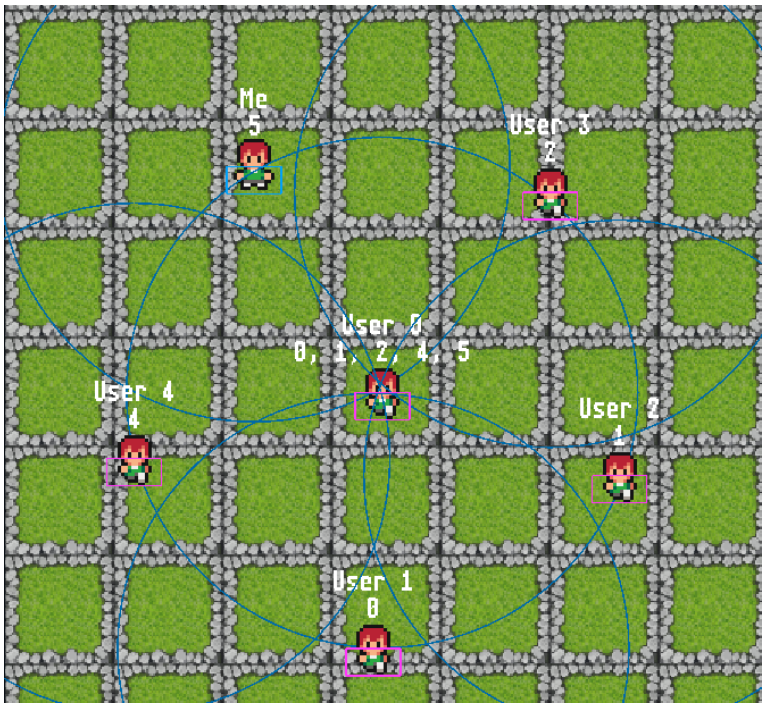

Figure 4.14: Proximity Chat Worst Case Scenario

4.5.2 Protocol

To establish the video-audio connection between users whenever they get close to each other, a sequence of messages is required to create a call successfully. The Figure 4.15 on page 35 illustrates this process.

First, a user needs to be inside a world with a web socket connection established with the API, which will serve as a signaling entity between the user and the media server, then send a message to the API with the users that they detected close to them, that will then, do the group normalization explained in section 4.3.5.

After the normalization process, the API sends to the media server actions made to the users and the groups, for example, close a group call or add a user to a group call.

The media server then sends the Real-Time protocol parameters for the user to be able to connect with the media server by creating a Transport for receiving and another for sending.

Once the user's Transports have been successfully created and connected, it must request for the streams of other users that are already inside the group call. If there are users producing data inside the group call, the media server must create a new Consumer for each Producer and send those Consumer parameters for the user to be able to consume their streams.

Finally, if the user wants to also send data, he must create a Producer, send his Device RTP parameters (see mediasoup documentation for more information), and finally start producing data. The media server on the other end will have to create a consumer for the new Producer and send the Consumer parameters to everyone else in the group call so they can start consuming the new stream.

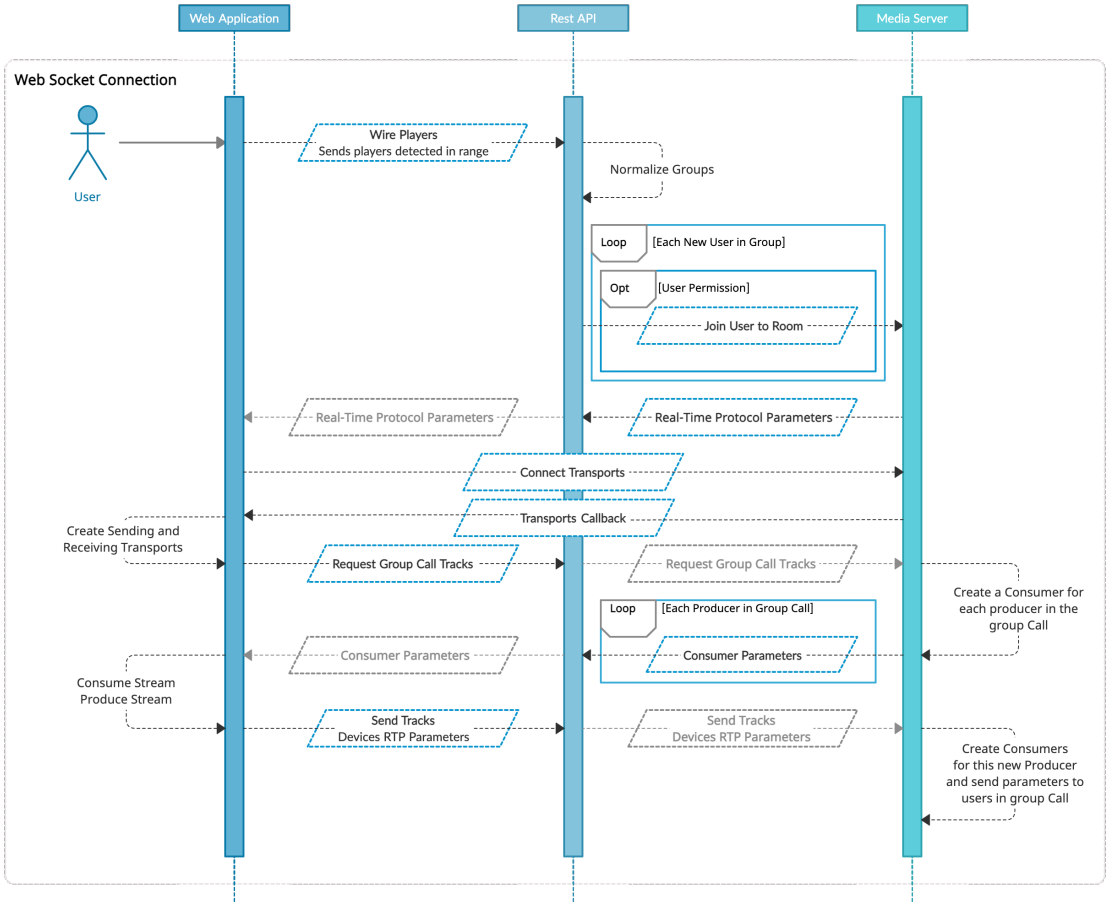

Figure 4.15: Communications Sequence Diagram

4.5.3 Scalability

In terms of scaling the communications, a mediasoup Router can hold up to a maximum number of 500 producers. For each data stream, audio, and video, it is created a new producer, so if in a group call with 16 users are sending both audio and video, when applying an equation (Eq. 4.1) that calculates the number of producers, we get 480. This means that it's no longer advisable to add more users to the call. One way we solved this issue was by starting to cut some video streams when it reaches a number close to 500. Also, since a Router runs on a Worker instance and there can only be several Workers equal to the number of CPU cores of the host machine, it is easy to calculate the maximum number of producers (Eq. 4.2) that the media server can hold on that machine, this way, whenever it reaches 80% of its full capacity, a message is sent to the API to scale the number of media server replicas. This also requires that the API ensures the signaling is done correctly to the right media server queue since they don't have shared memory access so they can run smoother.

$$NOPIAC = 2 * (NOU^2 - 2) \tag{4.1}$$

NOPIAC: Number Of Producers In A Call
NOU: Number Of Users producing both audio and video

$$MNOPIAMS = NOHMCC * 500 \tag{4.2}$$

MNOPIAMS: Maximum Number Of Producers In A Media Server
NOHMCC: Number Of Host Machine CPU Cores

4.5.4 File Sharing

For users to be able to share files, we decided to use WebRTC and mediasoup as well, since we were already using it and we didn't want to add another technology dependency. In addition to that, WebRTC file sharing is known to be very secure since it is passed through encrypted channels and also supports any file size because it divides the files into chunks. By doing this, it also allows us to show a percentage of the amount of data transferred, estimate the download time and how much time is left while downloading, which enhances the user experience. Also, it allows to detect incomplete file transfers and handle these situations differently.

However, we realize this approach had some limitations, since our architecture isn't peer-to-peer the media server has to relay all data channel traffic to the right data consumer, which can be a burden to the server if a lot of people start sharing their files. To decrease this, we added a file size limit and also reduced a little the upload and download ratio.

4.5.5 Screen Sharing

Screen sharing is simply another video track that is stored in the mediasoup data structures and assigned to the right peer. Then the process is the same as the video and audio tracks of the calls, but it is identified by the "kind" property set on mediasoup's Producer.

4.6 Kubernetes Infrastructure

To create resources on our Kubernetes cluster, in continuation of what was already explained, we have created YAML files for each of the resources that we needed. However, as all files need to be applied one by one to achieve those objectives, it would not be practical to manage and use. Therefore, we recurred to ***Helm***, a Kubernetes package manager. With Helm, we can pack the resources needed to deploy a service, in a unit called *charts*. Each of our services is packed into a different chart, and, consequently, those charts are packed into a parent chart, an *Umbrella Chart*.

4.6.0.1 CertManager

Great part of today's browsers only accept video and audio stream transmissions on websites with secure connections, in other words, using the HTTPS protocol. To do this, we needed to get issued a valid certificate from a Certificate Authority and, at the same time, use that certificate in the Kubernetes Infrastructure. Kubernetes allows the management of certificates through ***CertManager***, which can be used to request the emission of a certificate automatically to our hostname by the *Let's Encrypt* Certificate Authority [20]. Once the certificate is issued it is stored inside a specific type of Kubernetes resource, a *secret*.

4.6.0.2 Persistent Volumes and Persistent Volume Claims

Every pod running on Kubernetes, by default, does not persist any data, therefore if, for example, a PostgreSQL pod dies, every data once stored on the pod's database will be eventually lost. To fight this back, Kubernetes provides a specific type of resource, denominated *Persistent Volumes*. Persistent Volumes allow the data to be persisted independently from the pods' lifecycle. However, as many pods may access at the same time these volumes, data inconsistencies may occur, another resource should be used - a *Persistent Volume Claim* - which is, in its simplest form, a request for storage by a pod. In our system, we used these kinds of resources on RabbitMQ, Redis, and PostgreSQL.

4.6.0.3 Autoscaling

The capacity of creating multiple replicas of the same services, dynamically, was a functionality that we wanted to make use, to increase our horizontal scalability. To do so, an addon has been installed on the cluster, called *Metrics Server*[21]. This Metrics Server collects CPU and RAM usage by each service, and, by defining on the YAML configuration files the maximum CPU and RAM usage by pod of our services, Kubernetes will dynamically scale the number of replicas. Therefore, if we define that the maximum CPU of service is 80% of the resources available for it, once that value is reached, a new replica will be created, and so on. This approach was used on the *frontend* Web App and the REST API.

However, as we needed to control the scaling of the Media Server based on occurring events instead of the metrics mentioned before, we decided to use the Kubernetes driver for Python. With this driver, every time the REST API receives a request from the Media Server to upgrade the number of replicas, consequently, a request to the Kubernetes API is made to do so. The figure below, shows the python code needed to increase the number of replicas:

```
def get_all_pods(self):
    return self.client.list_pod_for_all_namespaces(watch=False).items

def get_mediaserver_pods_names(self):
    pods_list = self.get_all_pods()
    return [pod for pod in pods_list if 'mediaserver' in pod.metadata.name]

def scale_mediaserver_replicas(self):
    payload = {'spec': {}}
    payload['spec']['replicas'] = len(self.get_mediaserver_pods_names()) + 1
    p = JSON.dumps(payload)
    try:
        self.apps.patch_namespaced_deployment(
            name='crowdwire-mediaserver', namespace='default', body=JSON.loads(p))
        logger.info(strings.SCALE_UP_SUCCESS)
    except ApiException as e:
        logger.warning(e)
```

Chapter 5

Functionalities

As it was presented before on chapter 3.4, throughout our work we have defined 4 actors. In this section, to explain the functionalities implemented in our platform, we will be dividing them across all the actors. It is also important to remember that some of the actors share functionalities. For more information see chapter 3.4.

5.1 Guest Functionalities

The *Guest* actor has access to the least amount of functionalities and serves as a base actor to the other actors, meaning that every other actor inherits, to some degree, most of its functionalities.

5.1.1 Login

Guest users do not have to register in the platform, however, once the user leaves the platform all its data is lost. Figure 5.1 shows how a user can join as a guest in the platform. By clicking in "JOIN AS GUEST" (signalized with a red square), the user does not need any further steps to be able to access the rest of the available functionalities.

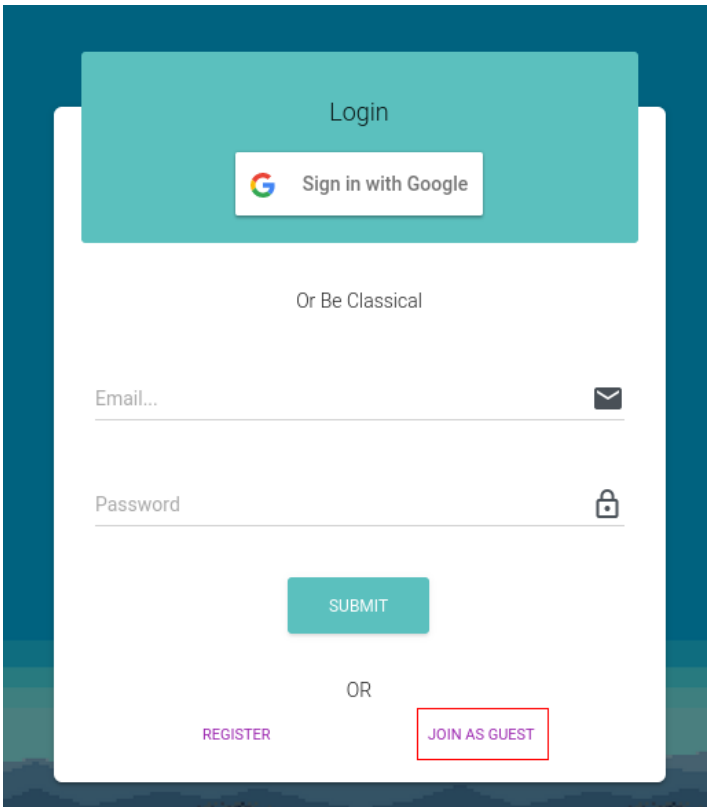

Figure 5.1: Guest Login Page

5.1.2 Dashboard

Once the user is logged in, it is redirected to the *Dashboard* Page where it has access to multiple functionalities. In Figure 5.2 we can see a sidebar(expanded in the figure) with the possible actions that the user can perform.

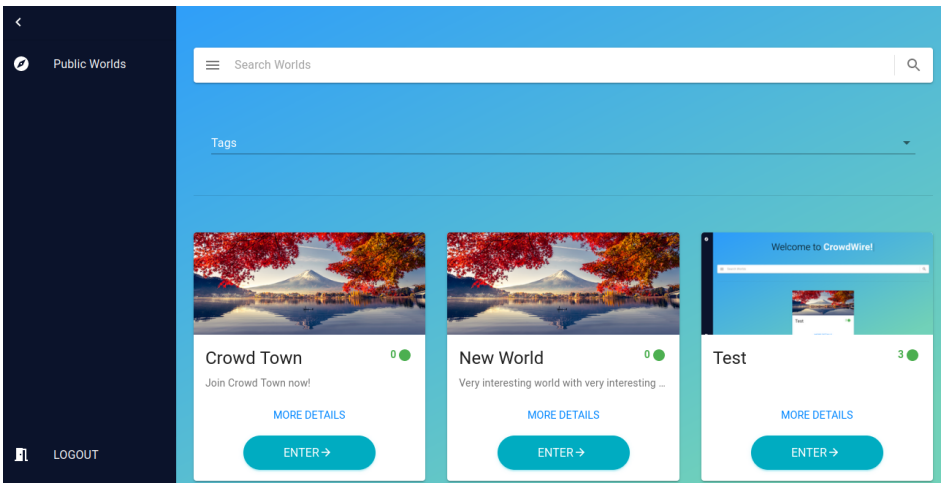

Figure 5.2: Guest Dashboard Page

On the bottom, the user can log out, which in this case, since it is a guest user, it will not be possible to join with the same guest "account" again.

Considering that guests have a reduced amount of functionalities, the only page available for them is the current page, where it is possible to search for public worlds.

On this page, the guest can see the worlds available to it. Worlds can be private or may not allow guests to join, so the guest user cannot see any world that matches those requirements. For users to search for words based on their preferences, they have at their disposal a filter that allows them to search by either the name or description of the world as well as tags that can be used to find worlds that better serve their interests. Figure 5.3 shows some of the possible filters that the user can use when searching for worlds.

The available worlds are displayed as a *Card* as we can see in Figure 5.2. The Card contains information about the world such as name, description, and the number of online users.

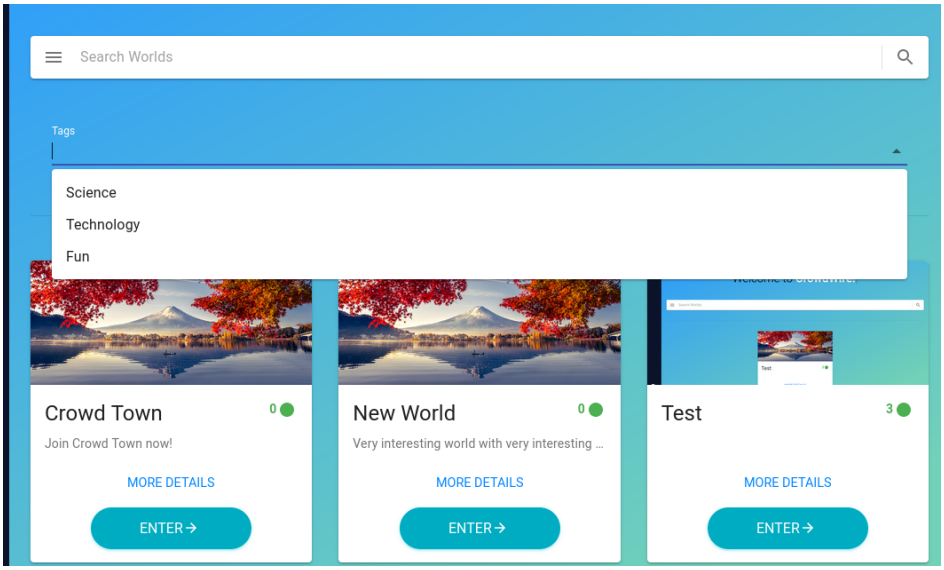

Figure 5.3: Search Worlds Filters

5.1.3 World Details

The World Cards from the *Dashboard* Page allows the user to see the details of a world by pressing "MORE DETAILS". It is redirected to a page that contains more information about the world such as tags and the visibility of the world, as it can be seen in figure 5.4.

The user can join the world from this page or by pressing "ENTER" in the world Card from the previous page.

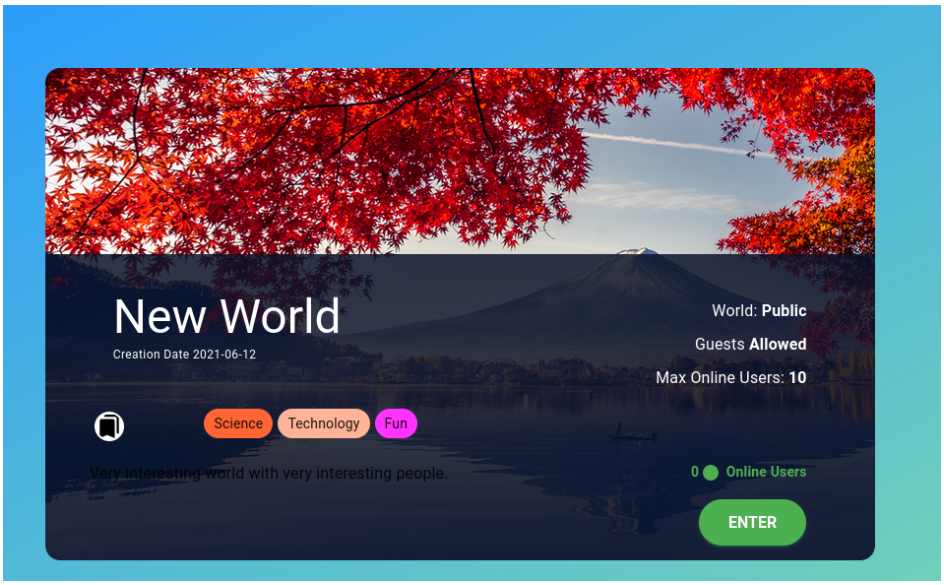

Figure 5.4: Guest World Details

5.1.4 Join World

When entering the world, the user will be asked for the **Username** and **Avatar** that should be associated in the world in cause, pictured in figure 5.5. It is possible to have different Usernames and Avatars in different worlds and they can be changed whenever the user joins the world again. Besides this, in the same figure, figure 5.5, it can be observed the key bindings on the screen that show the user how they can move and interact inside a world. This kind of feedback was added after the Usability Tests, that are mentioned later on chapter 6. After, choosing its preferences, it will be redirected to the page containing the world interface. When inside a world, multiple features are available to the user. Figure 5.6 illustrates a user inside an example world.

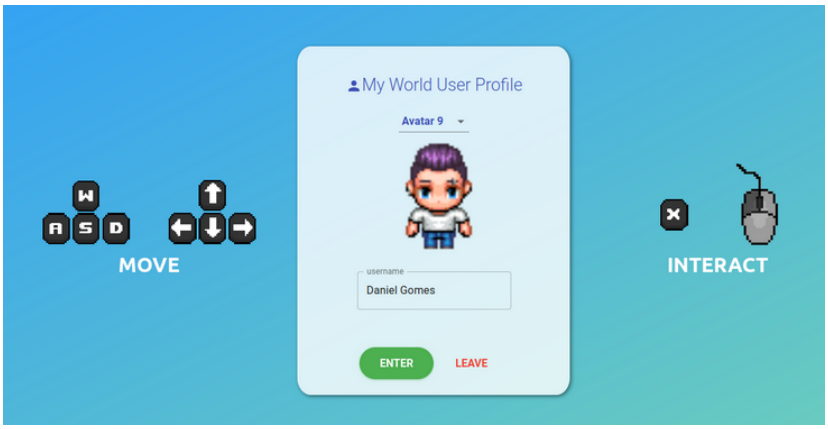

Figure 5.5: Choose Username and Avatar

If the user wants to leave the world, it can do so by pressing the exit button in the sidebar.

Figure 5.6: Guest game interface when inside a world

5.1.5 Join World By Link

It was mentioned previously that some worlds might not be public, making them impossible to access when searching. However, these worlds can be accessed through an invitation link. Anyone with that link can enter the world whether or not it is private. There is an exception, guests can never join a world that does not allow them, even when using the invitation link.

When the user accesses the link, if it's not logged in already, or its token has expired, it will have to log in firstly, and, once this process is successful, is redirected to the world.

This feature combined with login as a guest user makes it very efficient and user-friendly to gather around users and create on-the-fly meet-ups inside a world, while being able to keep it private or not.

5.1.6 Proximity Chat

Although there might be some restrictions to this functionality when taking into account the role of the user, usually guests are allowed to walk and talk with nearby users. The concept of proximity chat was explained in section 3.3 as well as what is a role and how it affects the users when inside a world.

The next two figures, 5.7 and 5.8, present an example of how the proximity chat works. The user in the middle can communicate with both users(figure 5.8), while the users on each side can only communicate with the one in the central position (figure 5.7).

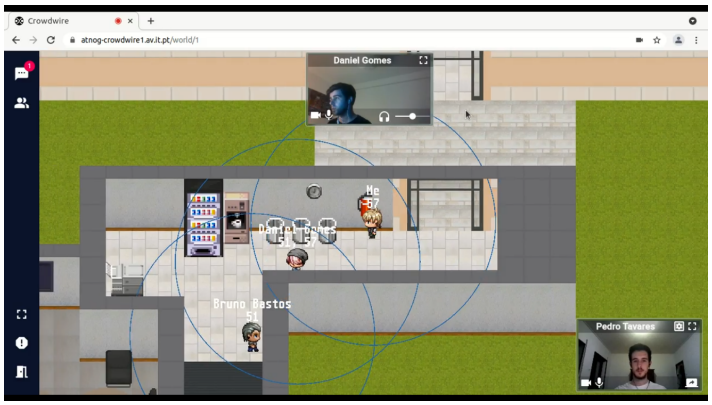

Figure 5.7: Proximity Chat example where user can only communicate with 1 user

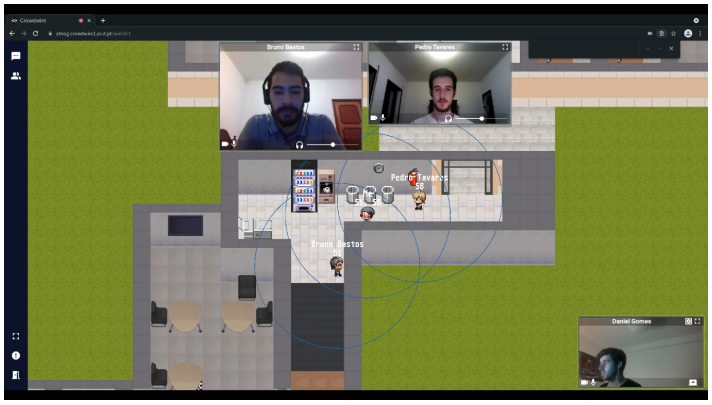

Figure 5.8: Proximity Chat example where user can communicate with 2 users

Related to this functionality, there is also the ability to mute other users or even itself. The same can be done with video cameras, as users can choose to see others or show themselves to others. Figure 5.9 shows how the user can turn off its camera and microphone by pressing the *camera* and *microphone* buttons.

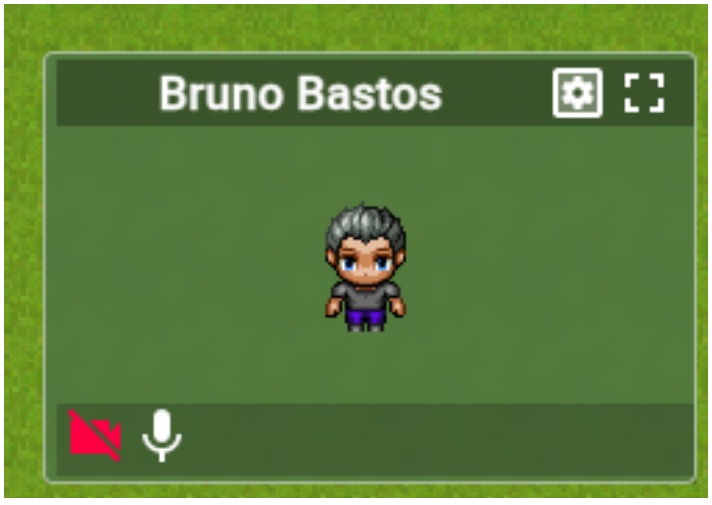

Figure 5.9: Turning off camera example

It is also possible to configure and change the current camera and microphone, allowing every kind of user to choose their audio and video input device, as pictured in the figure 5.10. This interface is reached after pressing the definitions button shown in figure 5.9.

Figure 5.10: Video and Audio Settings

5.1.7 Text Chat

Sometimes it is useful to share something by text because there might be some difficulties when using the voice. For that reason, we provide a chat where users can communicate using text. To do so, the user must click on the text chat button in the sidebar, where it will have access to the recent messages available. It is important to note that messages are not stored, so if the user re-enters the world, all of the past messages will not be accessible.

There are two channels where messages can be sent to, which can be selected at the bottom. The "All" channel, which will send the message to every user in the world, and the "Nearby" channel which sends the message to everyone in the same voice channel.

Figures 5.11 and 5.12 show an example on how the text chat works. The first user does not receive a message from others because these are not close to each other. On the other hand, because both users are in each other's range in the second image, there is a message sent to the nearby chat that the user may see. As we can notice from both examples, when sending a message to the *All* chat, everyone can see it.

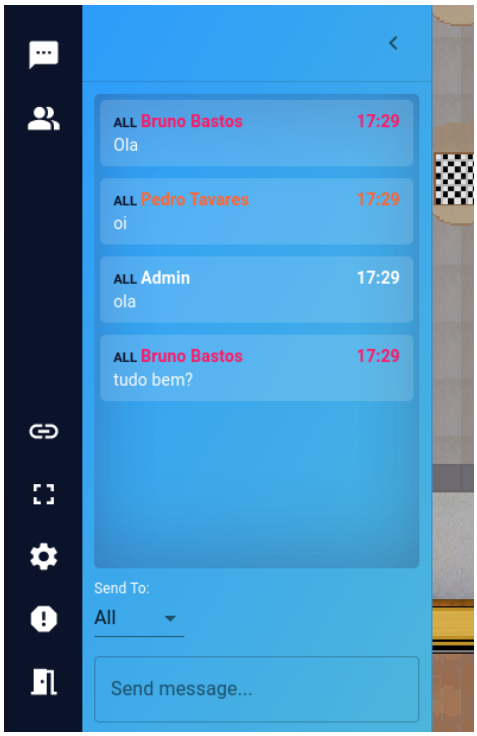

Figure 5.11: Text Chat Example

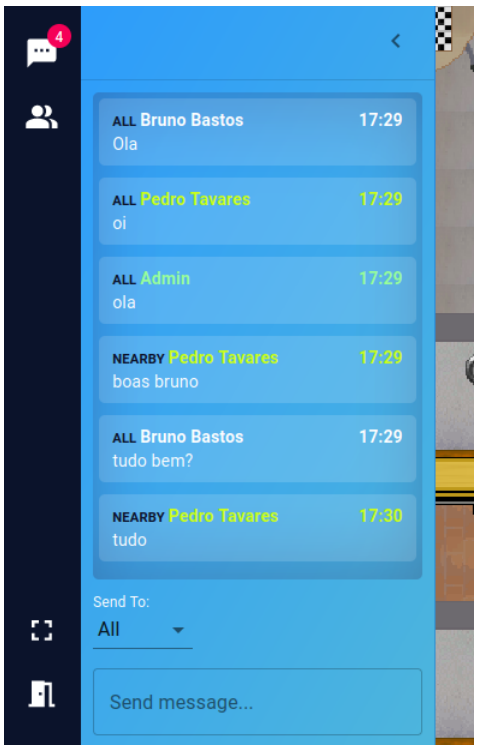

Figure 5.12: Text Chat With Nearby User example

5.1.8 Conferences

Conference Areas, as it was explained in section 3.3, are invisible areas where users can communicate with everyone inside independently of their distances. However, the ability to talk in these areas is limited by the role assigned to the user.

If the role does not allow, it must be the user to request for permission to talk. This is done by clicking on the "hand" icon in the voice controls, as shown in figure 5.13. The permission to talk request is sent to a user with the conference managing permission, that must be inside that conference area and that user can decide whether or not the person requesting can talk. Figure 5.14 shows how the conference manager can give permission to the requester. When allowed, the guest user can talk freely as if it had the permission, however, leaving the conference area will revoke that permission.

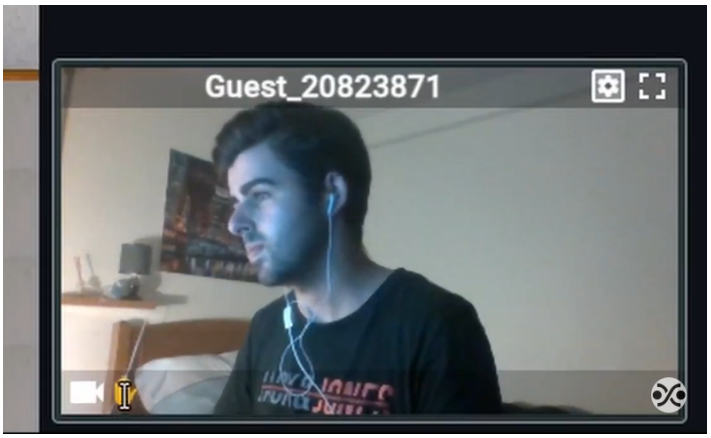

Figure 5.13: Conference ask for permission

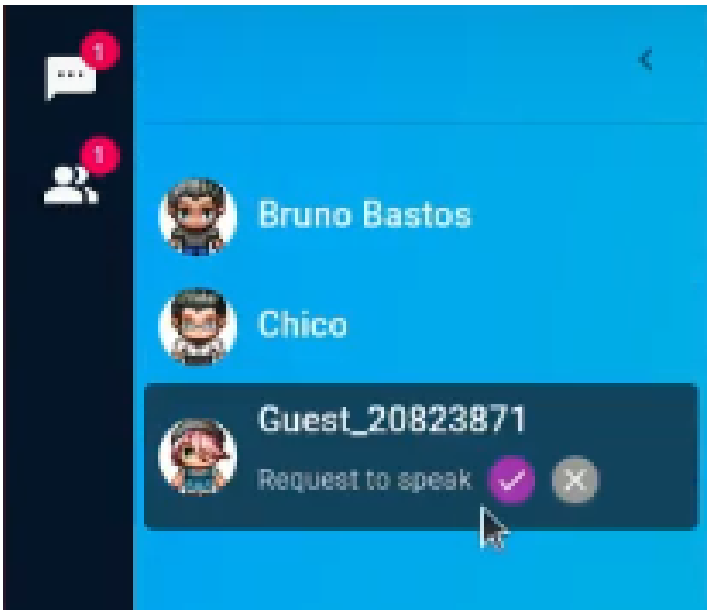

Figure 5.14: Conference granting user permission to talk

5.1.9 Interactive Objects

Scattered around the world there can be multiple interactive objects. The existence and position of those objects are defined by the world creator when editing the world map.

The functionality that allows the user to interact with objects is, again, gated by its role. To interact with the object, the user has to get closer, until the object is highlighted in yellow. Then, by clicking on top of the object, or the X key, the user will be able to use the respective functionality. Different types of objects provide different functionalities. We decided to group the objects into 3 different functionalities: Screen Share, File Share and External Services. For more information the section 3.5.1 in *Interact With Objects* should be seen.

Figure 5.15 shows an interactive object that gives the user access to the whiteboard external service. Figure 5.16 shows what the user sees when using this service.

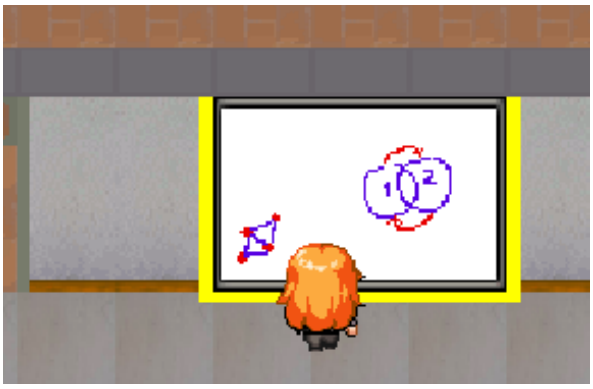

Figure 5.15: Interactable object that provides access to the Whiteboard External Service

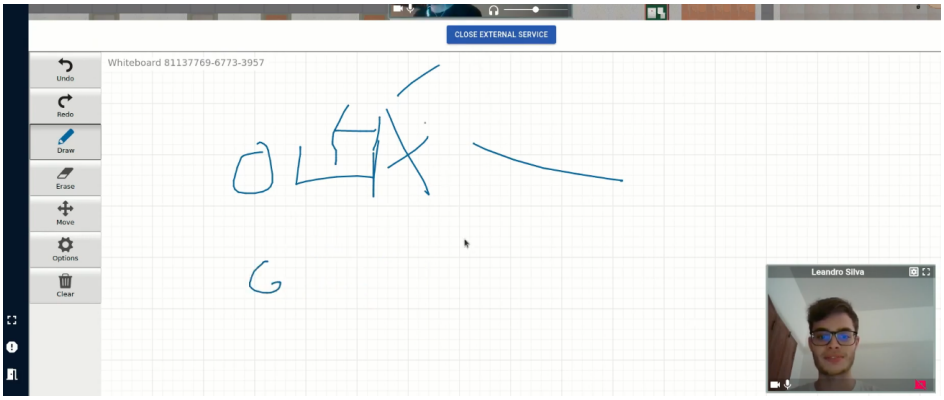

Figure 5.16: Whiteboard External Service example

Another type of interactive objects is presented in figure 5.17. By using this kind of object, the user can provide files for other nearby users to download as can be seen in figure 5.18. Any other user nearby can download the files by clicking on them. In figure 5.19 it is possible to see an example on how downloading a file might look like.

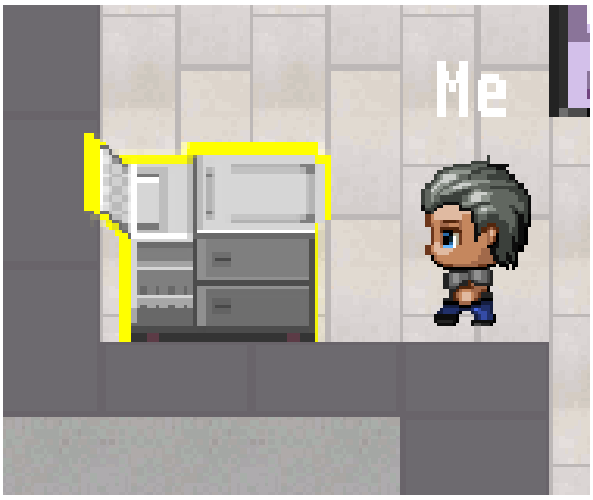

Figure 5.17: Interactive object that allows file sharing

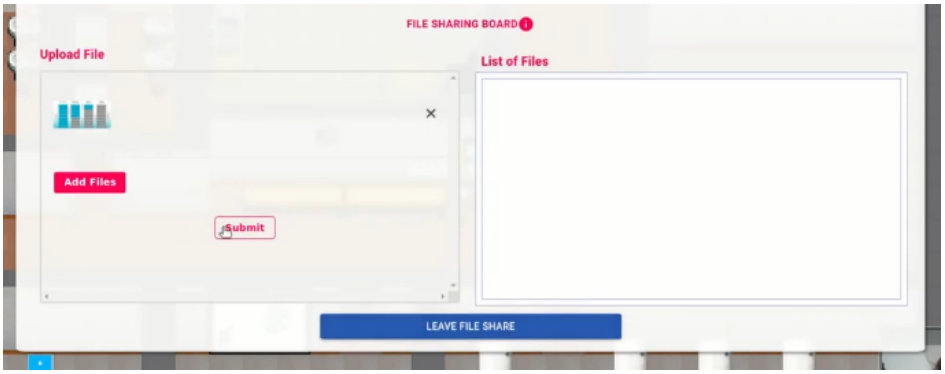

Figure 5.18: Sharing files with other users

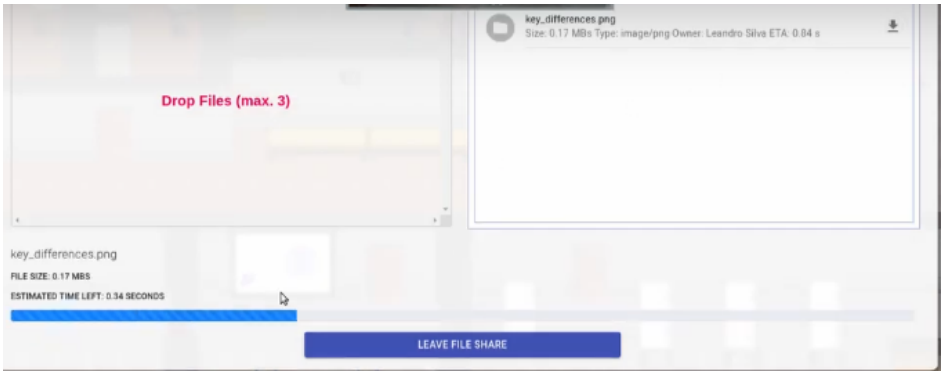

Figure 5.19: Downloading Files from other users

Finally the last type of interactive objects is the one that provides a user with the ability to share its screen with others that are nearby. Figure 5.20 gives an example of an object with that capability. Only the user that wants to share its screen needs to interact with an object, which will then ask for the window to share, as can be seen in figure 5.21. For the rest of the users that are nearby, their perspective will look something like the example in figure 5.22.

Figure 5.20: Interactable object that allows users to share their screen

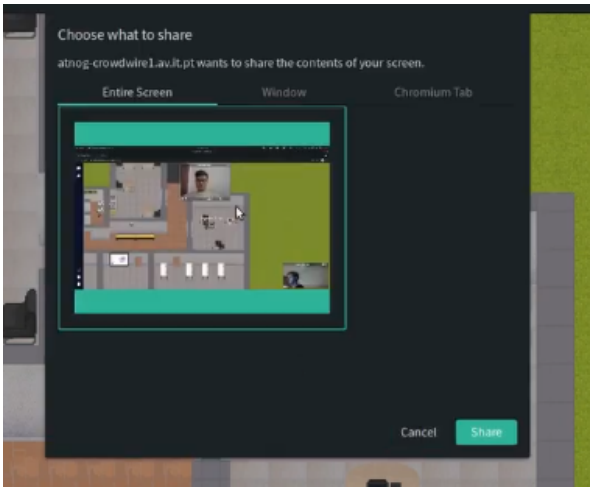

Figure 5.21: Choosing what screen to share

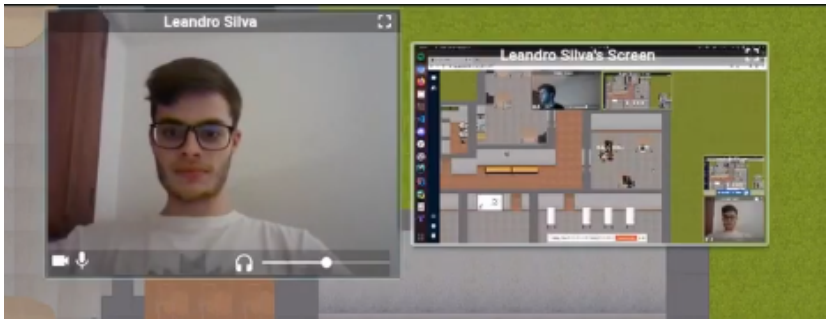

Figure 5.22: Seeing the screen that is being shared

5.1.10 Multi-Platform Compatible

Any user can access our platform through any device and will be able to operate everything normally since all the guests' functionalities are compatible with any screen size and even the communications as mentioned before are compatible with any devices.

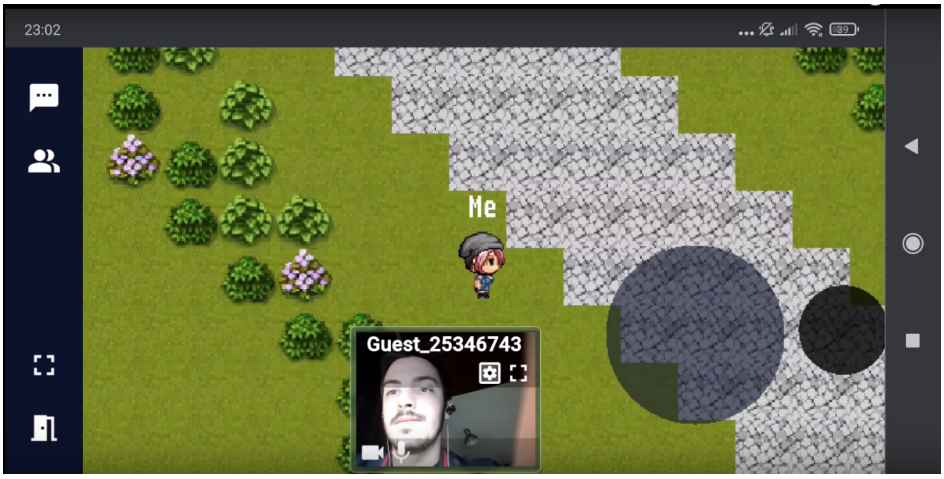

Figure 5.23: Crowdwire running on a mobile phone

5.2 Registered User

In the previous section, the functionalities of guest users have been exhibited. In this one, we will present the additional features that a registered user has access to.

5.2.1 Register

The user needs an account created in the platform so that it can enjoy a wider set of features. In the login form, there is a button that redirects it to the registration form. Here, there are some fields of information that need to be provided like the email and password. When finished, if the form is valid, the user will be sent an email. The email contains a link to activate its account, which will also make the login to the platform. The user can now access the features of a registered user.

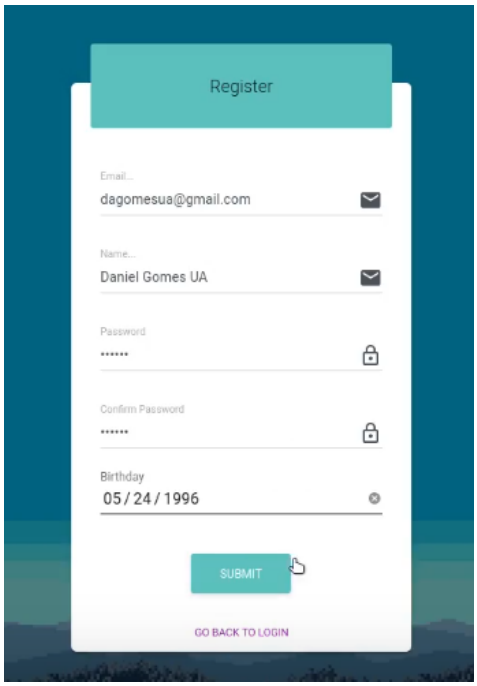

Figure 5.24: User register form

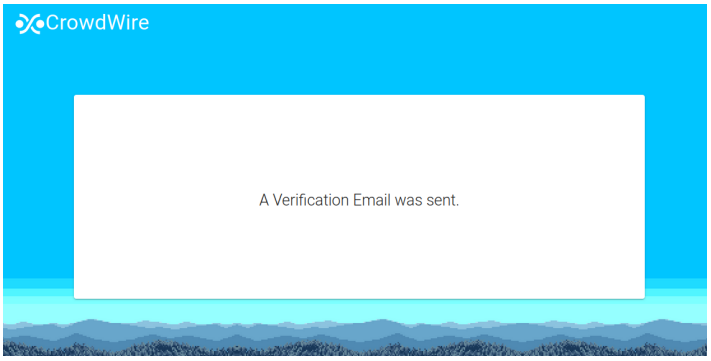

Figure 5.25: Verification Email Sent Page

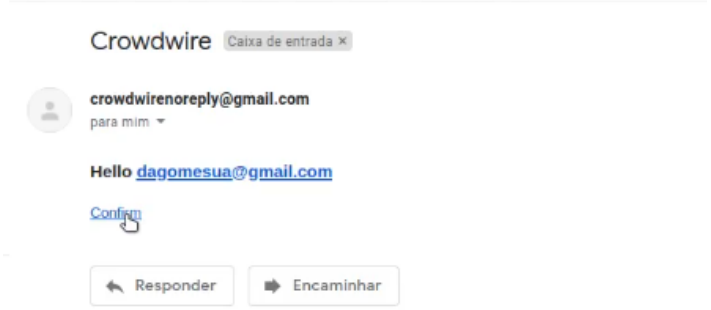

Figure 5.26: Confirm email

5.2.2 Login

If the user already has an account it is only necessary to provide its credentials in the login form and it can use the platform normally.

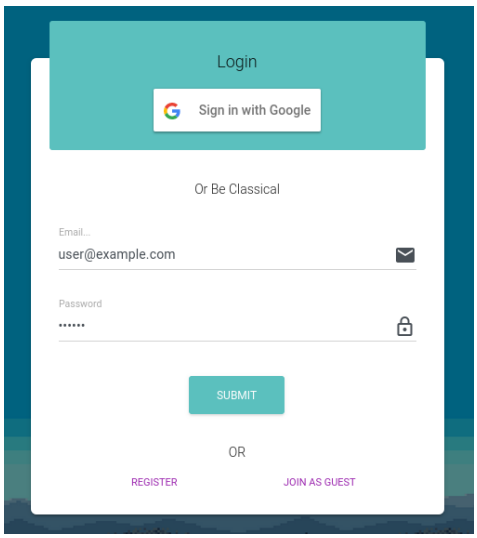

Figure 5.27: User Login form

5.2.3 Google Authentication

Filling the registration form and confirming the email can take some time, so we provide another method of authentication using Google Sign In. Using a Google account makes it is possible for a user to register and login into the application in a much easier and faster way. The user can choose which of its google accounts it wants to use in the platform via a pop-up window. Choosing one will register or login and redirect it to the dashboard.

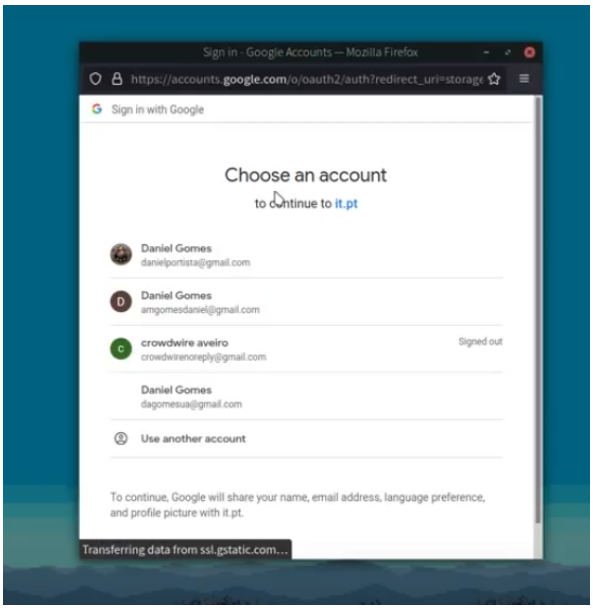

Figure 5.28: Sign In with google account

5.2.4 Dashboard

The dashboard differs from the guest user's one in the number of functionalities available. The registered user still has access to the public worlds but now it can also see the joined worlds and the created worlds, being able to perform the same search queries in both pages as it can in the public worlds.

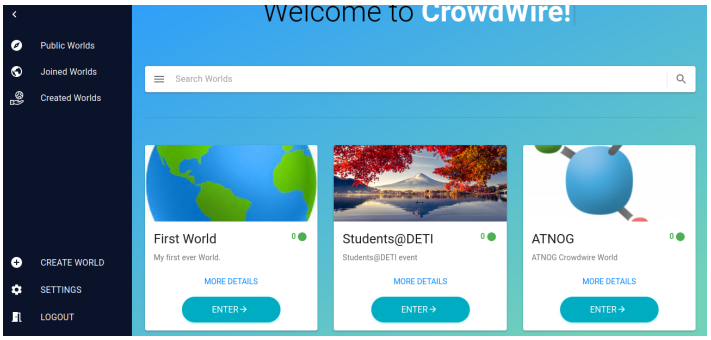

Figure 5.29: Registered User Dashboard

An important note is that worlds that are banned can appear on both of these new pages but not on the public worlds page. The reason for this is to give the user some feedback as to why it will not be able to join a specific world.

5.2.5 Edit Account and Profile

We allow users to change their credentials through the *Settings* page. It is also possible to change other information like the *name* and *birth*. The picture 5.30 shows the form to update a user's password, while the figure 5.31 represents the one that allows the update of other account's information.

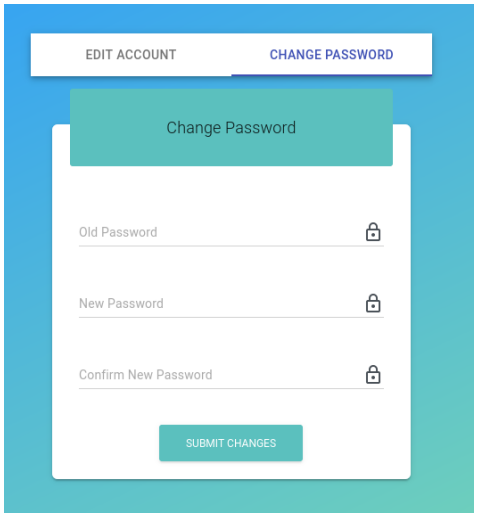

Figure 5.30: Registered User change credentials

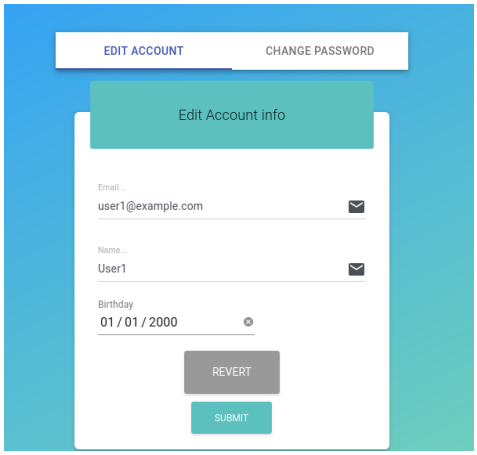

Figure 5.31: Registered User edit account information

5.2.6 Create a World

There is a button in the sidebar that allows a registered user to access the world creation form. By providing valid information for the fields shown in the figure 5.32, the user can have its world where it will have access to more functionalities.

After the world creation has been confirmed, the user is redirected to the details of that world.

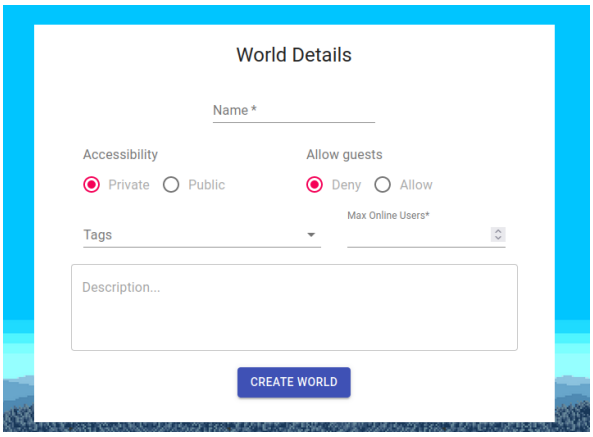

Figure 5.32: Create a World

5.2.7 World Details

This page is much more complete when compared with the guest's one. There a few more actions that the registered user can perform, some of which are only available, in this example, because the user is the owner of the world. World Creator functionalities will be explained in the next section.

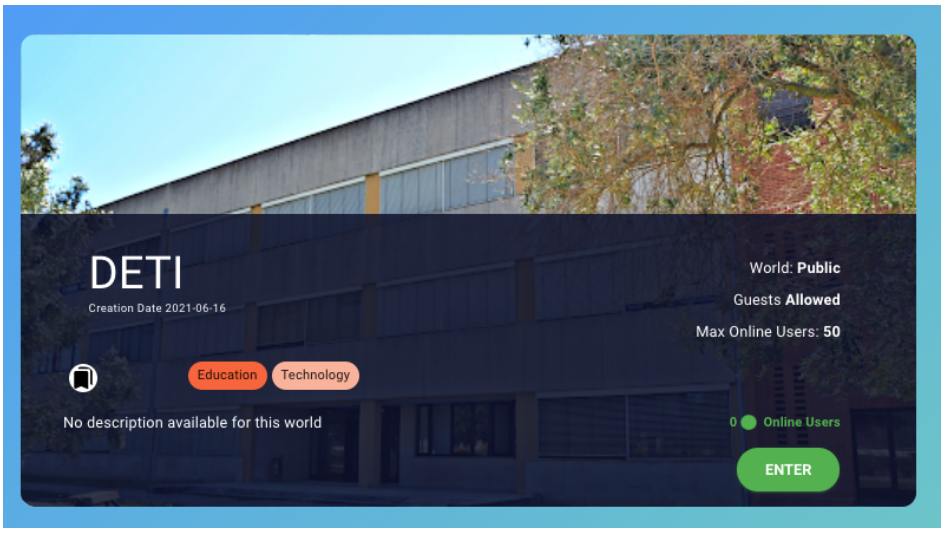

Figure 5.33: World Details for the World Creator

5.2.8 Manage World

This functionality can be accessed if the user has a role with role managing permissions. Pressing the "MANAGE WORLD" button redirects the user to a page where the roles and users can be managed.

Here, roles can be added, permissions changed and it is possible to see the reports and ban users from the world.

Figure 5.34 shows the role management interface. In order to change a user role, it must be moved its new role. The figure also shows the permissions associated with a role. Disabling some permission and clicking on the *Change* button, the role permissions are updated.

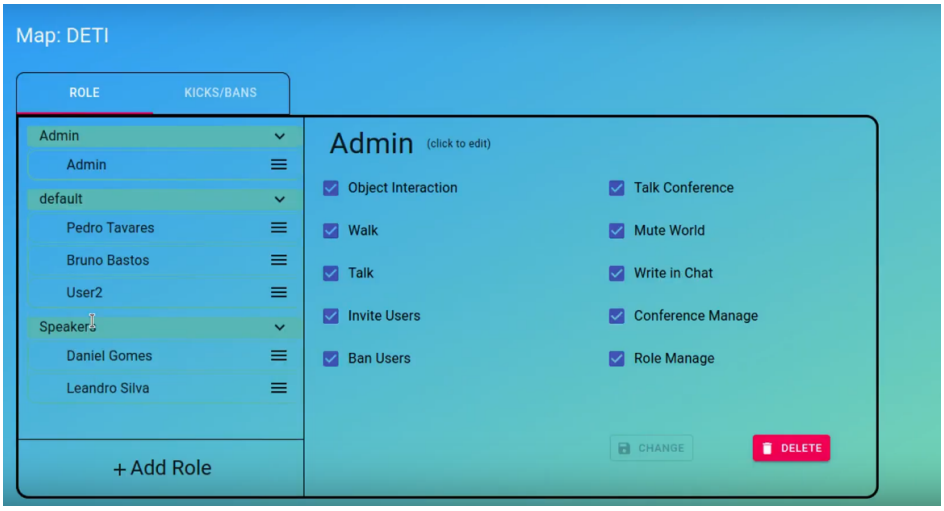

Figure 5.34: Roles and permissions management

Creating a new role is as easy as pressing the *Add Role* button and provide the role name, as can be seen in figure 5.35.

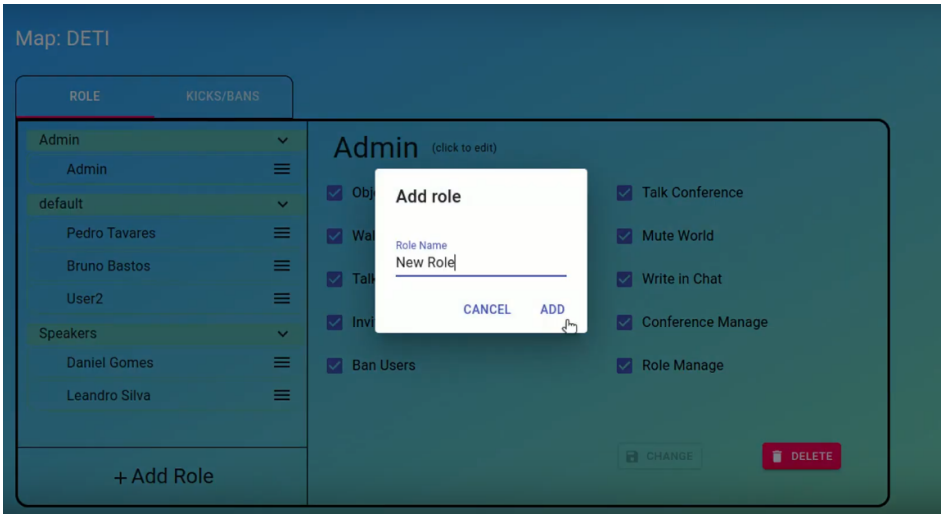

Figure 5.35: Adding an example Role

5.2.9 Report World

It is possible for a registered user to report a world if it feels that the world is doing something wrong. The same user can only report a world once and it is not possible to remove the report. Users cannot report a world that they haven't joined. Taking a look at figure 5.36 we can see an example of how to report a world through its details.

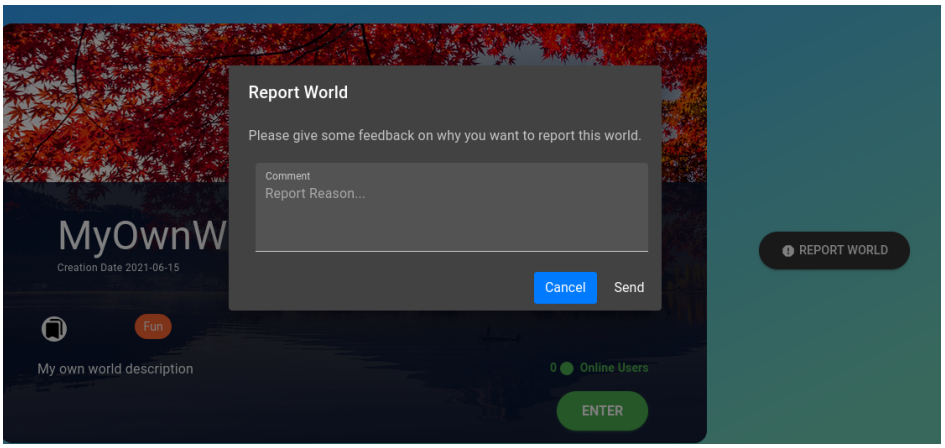

Figure 5.36: Reporting a World

5.2.10 Report Users

When inside the world, the user can report any other user that is currently online in that world. There can only be a report from a user to another in a given world, meaning that a user can report another user multiple times as long as they met in different worlds.

In figure 5.37 it is possible to see an example of how to report a user.

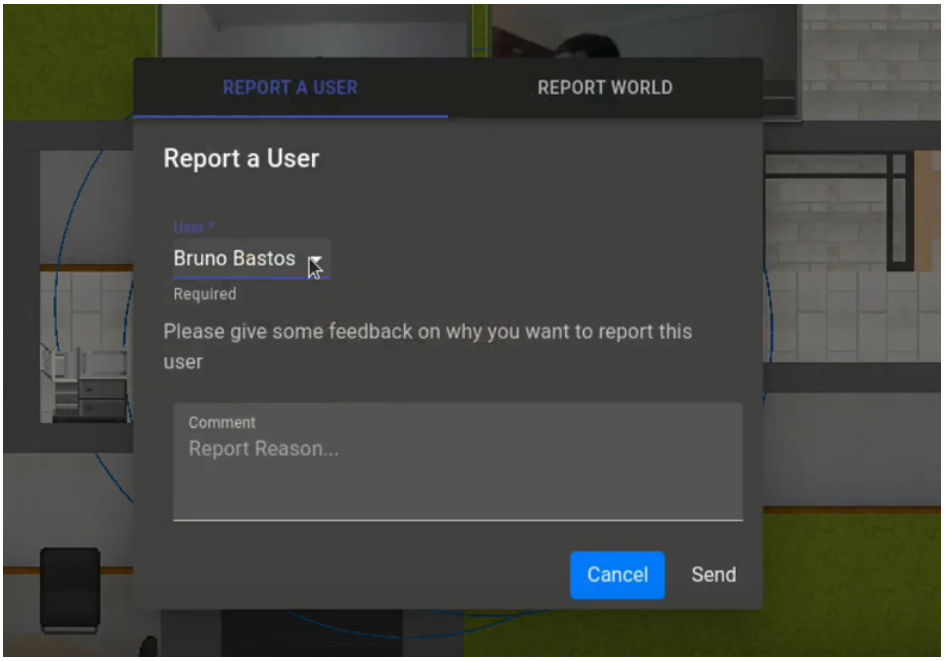

Figure 5.37: Reporting a user

5.2.11 Generate Invite Link

There is a role permission that allows users to generate a valid invitation link. Anyone using that link can join the world by following the steps explained in subsection 5.1.5.

To generate the link and have access to it, a user must click on the button in the sidebar when inside a world and generate the link in the new window, as can be seen in figure 5.38.

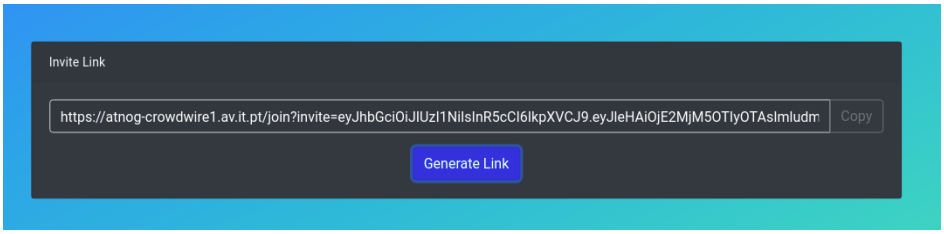

Figure 5.38: Generating a Invite Link

5.3 World Creator

As previously explained, in section 3.4, World Creator is a Registered User with extra functionalities to a world created by it.

5.3.1 Edit World Information

When inside the world details page of an owned world, the user can edit that world information by clicking in the *Edit* button. Here it is possible to change the logo, name, tags, description, and name of the world. In terms of the privacy of the world, the world creator can opt to change the visibility, making the world public or not, and can decide if it wants to allow guests or not to join the world.

Figure 5.39: World Creator world details page

Worlds can also be deleted by their owners in this page.

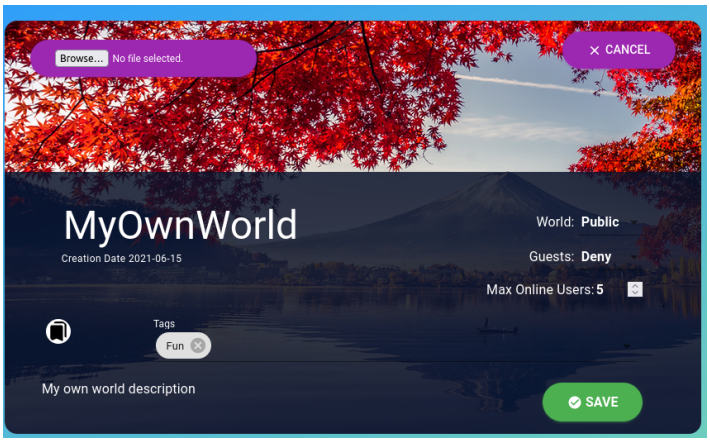

Figure 5.40: Editing World Information

5.3.2 Edit the World Map

CrowdWire provides a multi-functional map editor that allows the creators of worlds to customize and build their worlds according to their preferences. This functionality can be accessed through the world details page and by clicking on the "Edit World" button. The user is shown a complex interface where it is possible to place and delete tiles in different layers by selecting them in the menus (view figure 5.41). These layers can be blocked or hidden, as in most layer based editor, to facilitate the process of editing. Besides, it is possible to view a grid by activating the option "Show Grid" in the headers menu, and if the option "Highlight Selected Layer" is active, then the layers which are not selected will have less opacity (view figure 5.42). On the header options, it is also possible to configure the map's size.

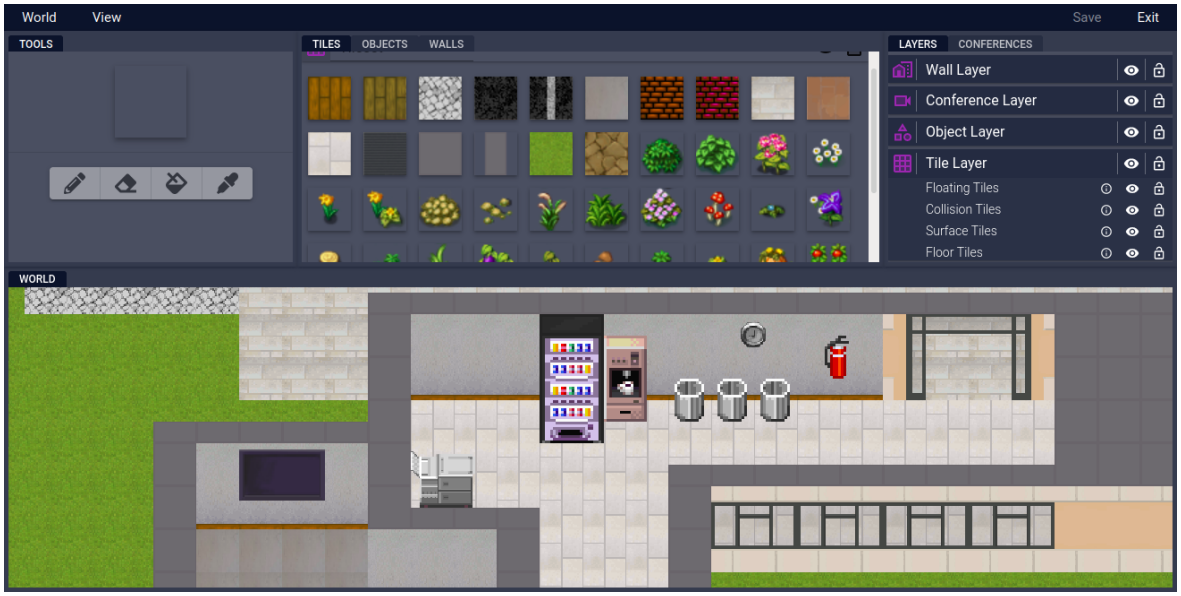

Figure 5.41: World Editor interface when entering the editor

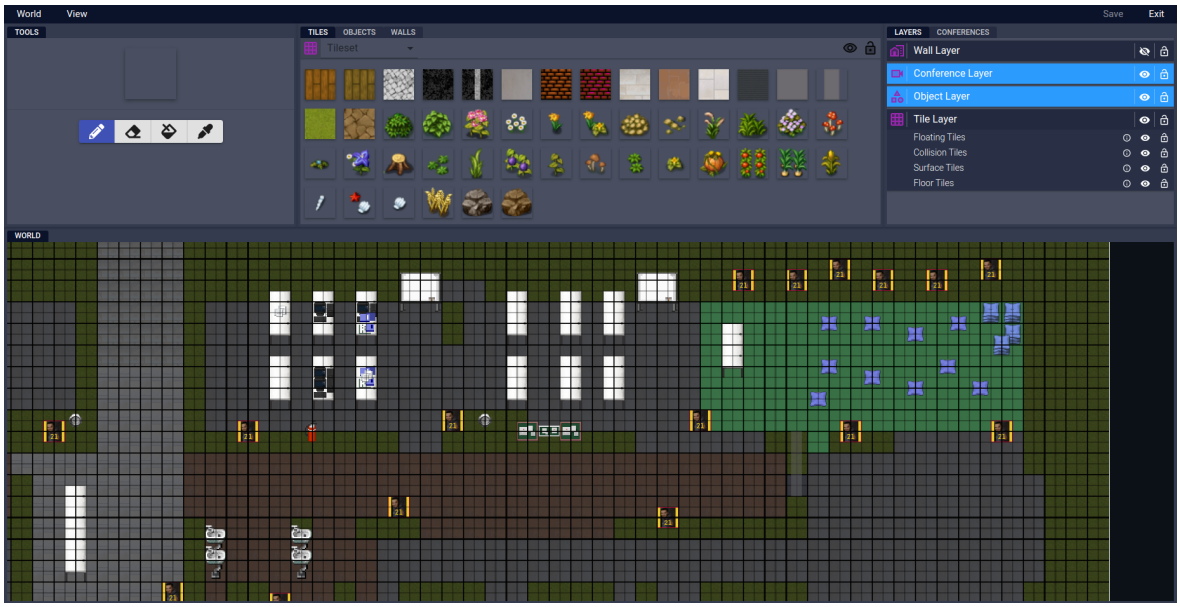

Figure 5.42: Hidden and highlighted layers with a grid displayed

Moving the camera follows the same principle as moving the player, which is done using either the arrows or *WASD*. To zoom, the keys *QE* are used, being *Q* the zoom in key, and *E* the zoom out.

It is also possible to mark areas as conference areas by utilizing the *Conference* tab, as shown in figure 5.43. Here the user can create and edit conference areas. It can choose the color, the name, and place them in the map where it would like the conference section to be.

Figure 5.43: World Editor Conference Tab and placement of conferences

When the results pleases the user, the world can be saved by pressing the "Save" button on the top right corner. To discard changes and ignore the warning message, the user has to press "Exit" twice.

5.3.3 View World Statistics

Creators might feel the necessity to check their world's statistics in order to know if something should be changed. The world statistics page provides information about the world which might be helpful for the creator if its objectives are, for example, to increase the number of users that visit the world.

Figure 5.44: World Statistics available to the creator

5.4 Admin Platform

Admins have access to all the functionalities described previously. There are some changes however to the sidebar of the platform. Now as can be seen in figure 5.45, there are different possible actions that were not possible when logged in as other users.

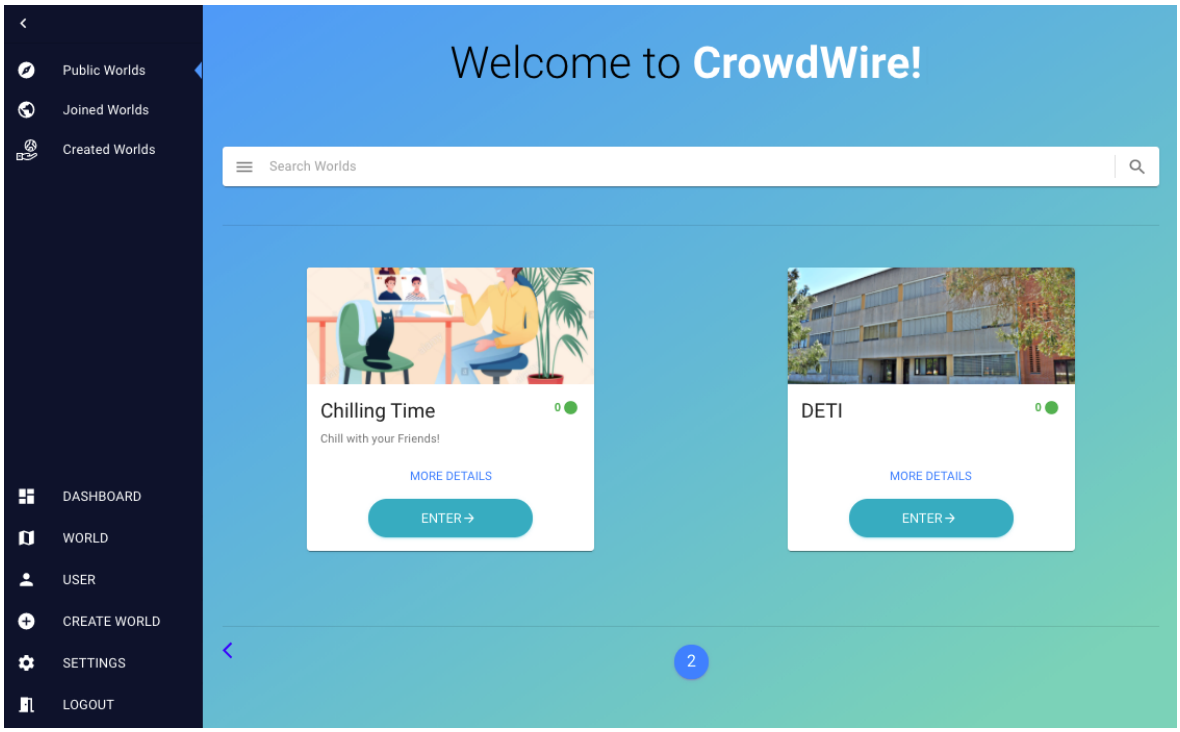

Figure 5.45: Admin Dashboard

5.4.1 Platform Statistics and Events

During a registered user session, there are many events that are being stored in the database, these are used to show statistics to the Admins. Choosing the option in sidebar, seen in figure, provides two options to the Admin, where it can choose between the statistics of the platform or the events.

Statistics page presents some statistics about the platform, that can be observed in figure 5.46.

Events page is a bit more interactive, allowing the Admin to filter events by date, world and type of event.

Figure 5.46: Admin Platform Statistics

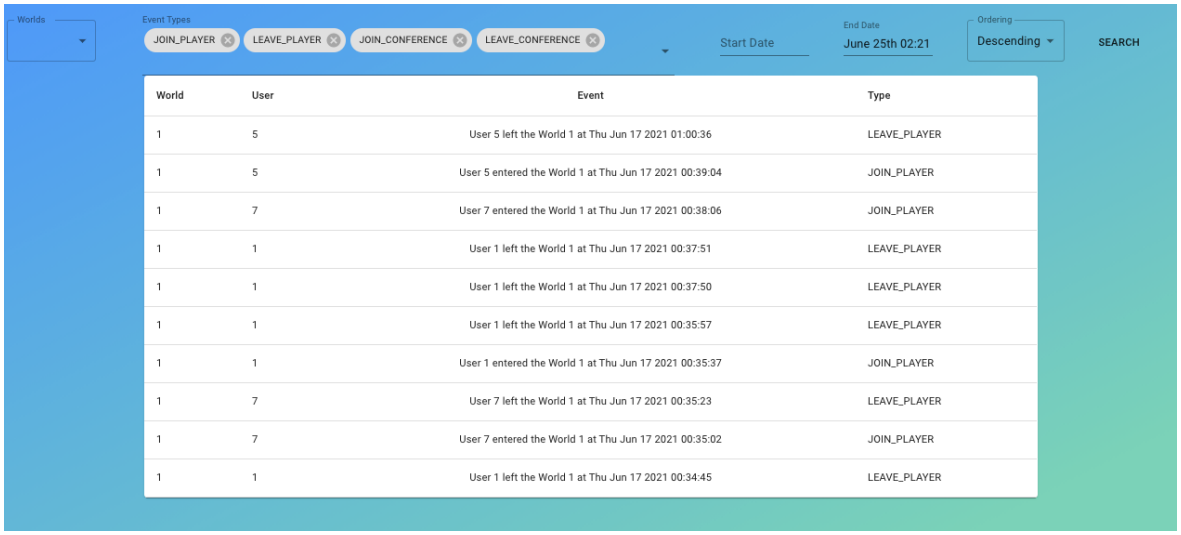

Figure 5.47: Admin Events Page

5.4.2 Users and User Reports

Selecting the *User* icon in the sidebar provides two options to access different pages, the users page and the users reports page.

The users page provides different filters for the Admin to search for the users of the platform. Picking a user will show the details of that user. It is also possible to ban the user if the Admin desires. In this details page there is important information about the user, not only the account details but the recent worlds created and the recent reports received.

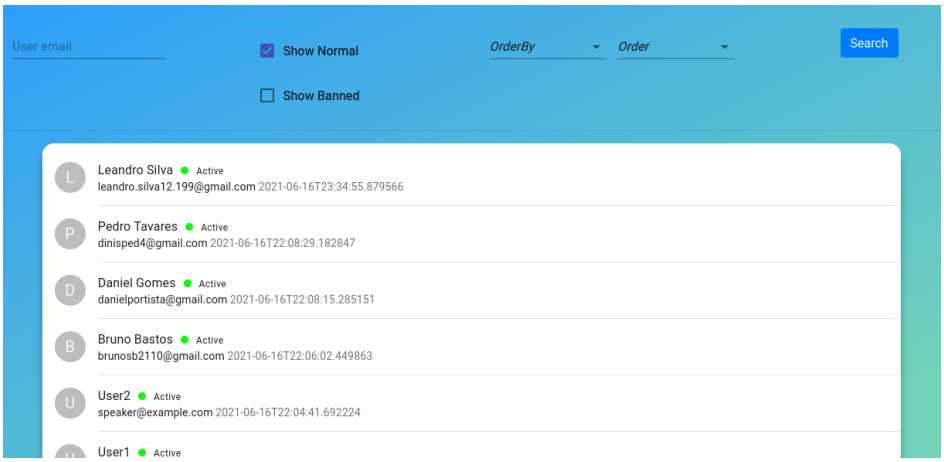

Figure 5.48: Admin User filtering page

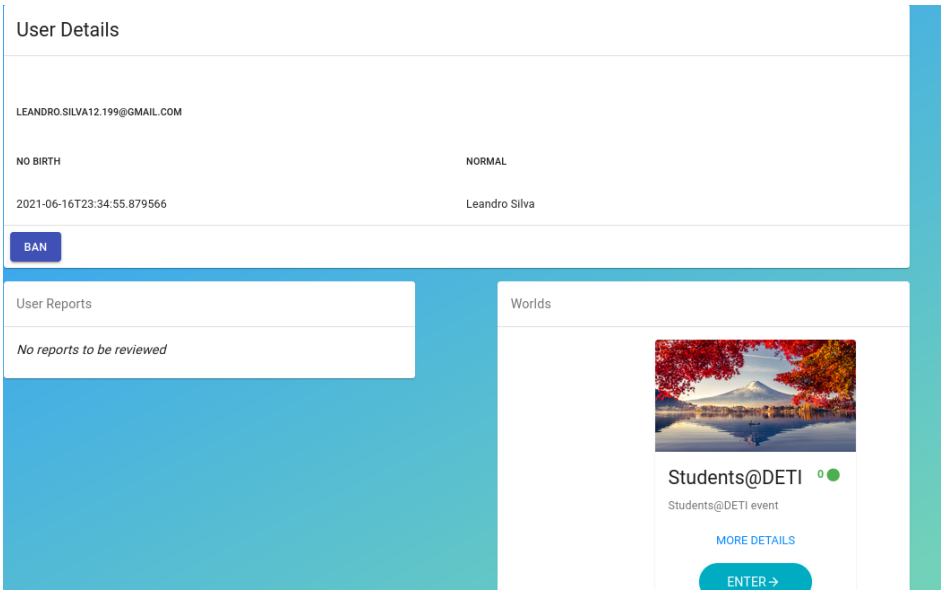

Figure 5.49: User details from Admin perspective

Reports page comes up with easy way for the Admin to search and filter reports made to users. After analysing a report, the Admin can mark them as reviewed, in order to not waste time on the same report again. Reviewed reports can still be seen by utilizing the filters.

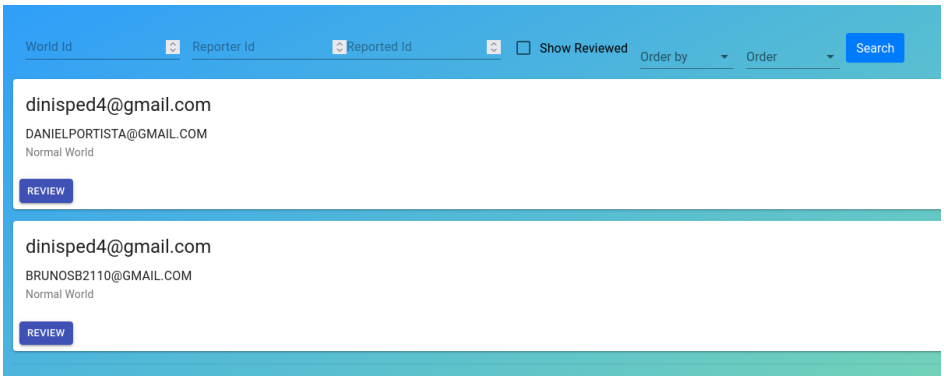

Figure 5.50: User Reports filtering page

5.4.3 Worlds and World Reports

Similar to the last subsection there is a way to filter both worlds and world reports.

Admins can find the worlds created in the application, and by selecting one of them its details are shown and here it is possible to ban the world and see its statistics.

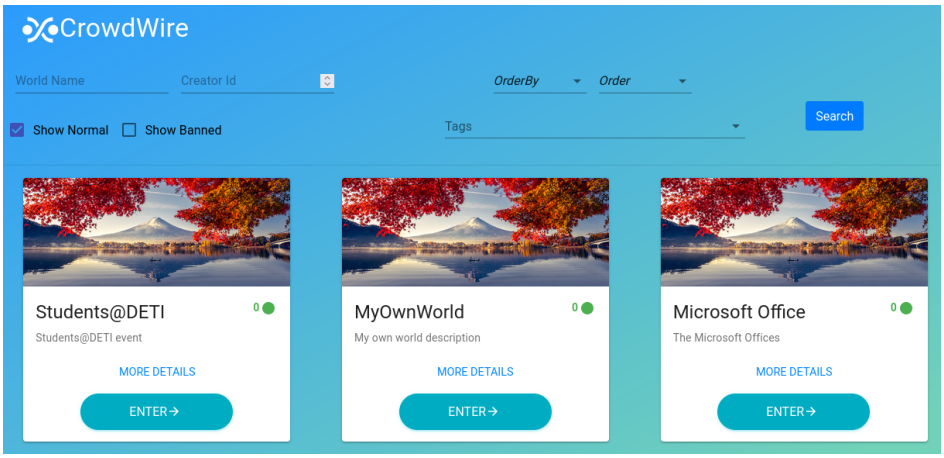

Figure 5.51: Admin Worlds Page

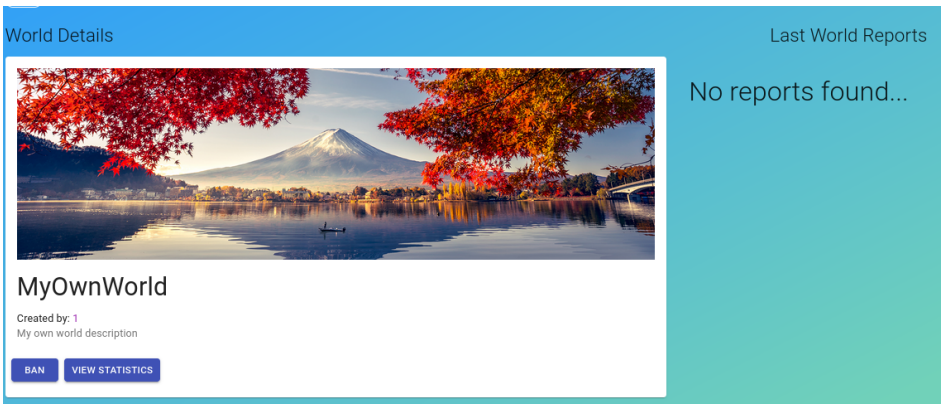

Figure 5.52: Admin World Details Page

In respect to the world reports, the functionalities are identical to the user reports, but this time dealing with worlds.

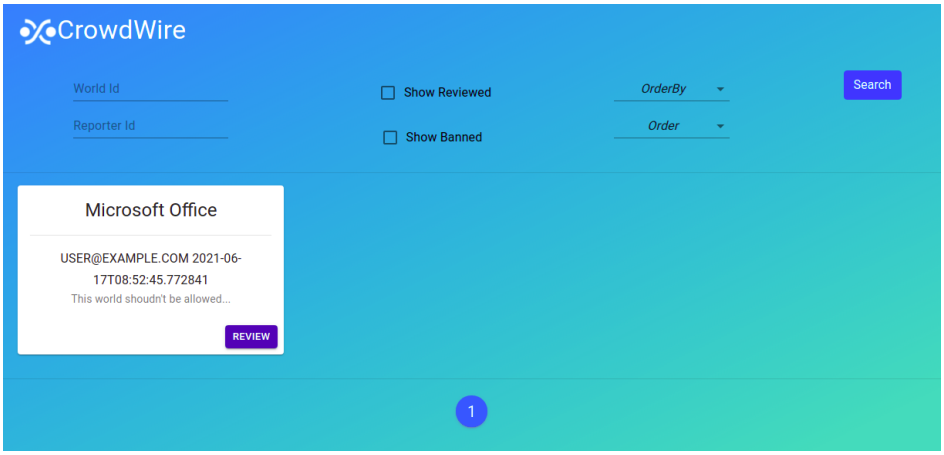

Figure 5.53: Admin World Reports Page

Chapter 6

Results and Discussion

Coming along with what was discussed in Chapter2, in terms of objectives of our project, we believe that most have been accomplished. However, when it comes to scalability, as our system has been deployed on machines belonging to *Instituto de Telecomunicações*, many network restrictions are applied for a user outside the network in cause. Therefore, the capability to test the scalability of our video and voice calls is affected since the number of users that can use this network is limited. To test the ability to dynamically increase the number of replicas we have changed, temporarily, the maximum threshold condition for a single replica handling data streams to a lower value that did not require a large amount of connected users to create a replica. The figure 6.1 shows the initial pods on the Kubernetes Infrastructure.

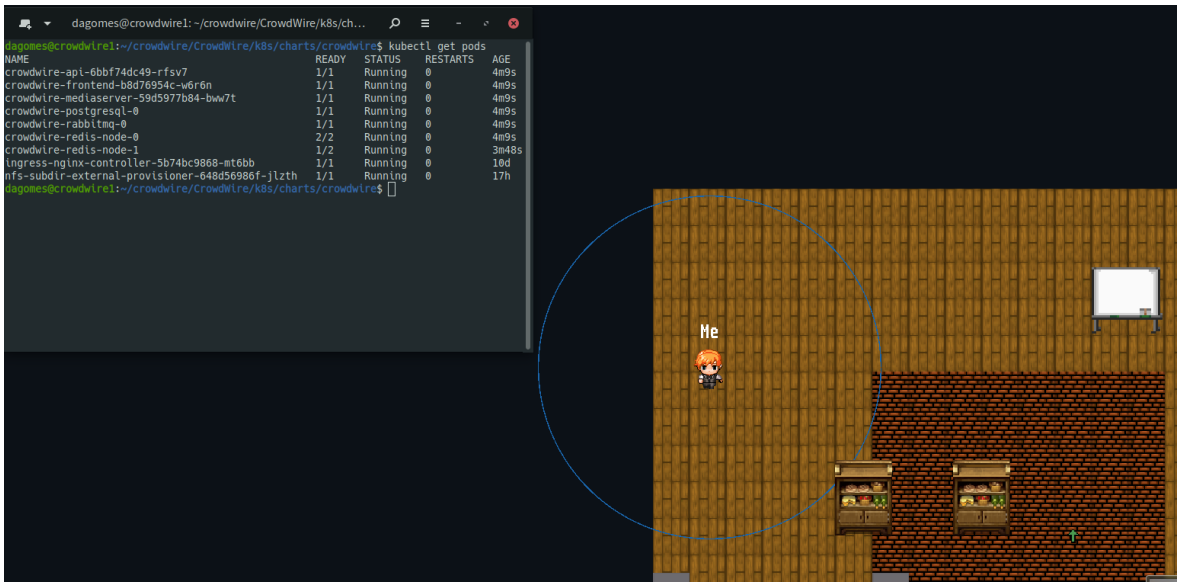

Figure 6.1: Initial Pods on the infrastructure

Once the maximum capacity is reached, the Media Server sends a request to the REST API to upgrade the number of replicas, which can be checked on the figure 6.2.

```
| INFO     | app.rabbitmq.RabbitHandler:publish:107 - Published message to Queue b'media_server_1'
| WARNING  | app.websockets.websocket_handler:wire_players:100 - unlock from wire
| INFO     | app.rabbitmq.RabbitHandler:on_message:44 - Received Request to scale replicas..
| INFO     | app.rabbitmq.RabbitHandler:on_message:45 - Scaling Replicas
| INFO     | app.k8s.KubernetesHandler:scale_mediaserver_replicas:38 - Success updating number of replicas
| INFO     | app.api.api_v1.endpoints.websocket:world_websocket:52 - Received message with topic WIRE_PLAYER

| WARNING  | app.websockets.websocket_handler:wire_players:71 - lock from wire
```

Figure 6.2: Media Server Logs

Consequently, a new pod of the Media Server is initialized with success, as it may be seen on image 6.3, with the presence of two pods with the prefix *crowdwire-mediaserver*.

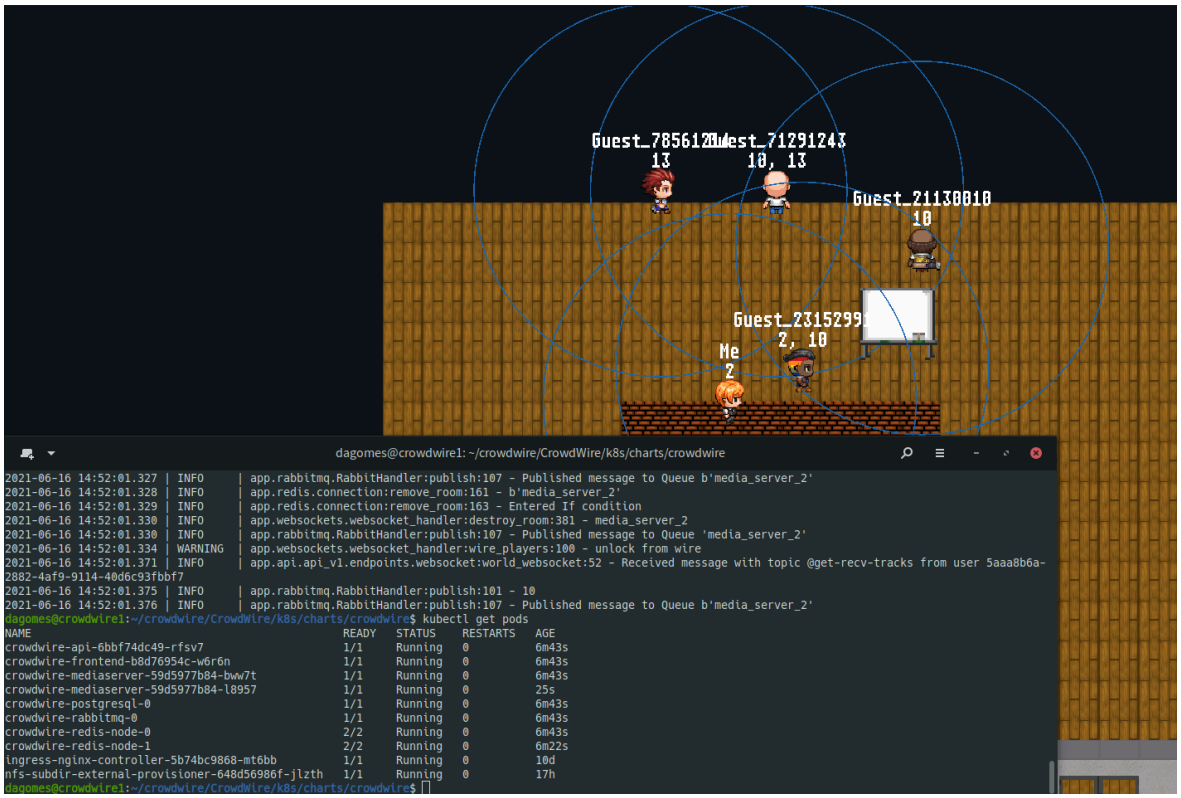

Figure 6.3: Pod Scaling on the infrastructure

6.1 Performance Testing

Taking into account the drawbacks mentioned previously, we went forward to test the performance of our application with over 10 users inside the same world. We have noticed that, with this amount of users, some issues started appearing. Firstly, as users enter and leave calls, since new WebRTC transports are initialized to allow video and voice calls, the movement of the players inside the world started losing fluidity. Besides this, the previous issue led to a lot of delay on the calls and made it impossible to talk with some users.

After some discussing and debugging, we realized that the WebRTC transports were not being closed once a call was over, which increases overhead and decreases the performance, which was fixed afterwards. Another problem identified was the message flooding that our REST API received via WebSockets once users join or leave a call. The solution found to this issue was the addendum of delays on the message sending, once a user is in the range defined for starting a call. These delays are defined according to the number of other users within the range.

Even though we have had these issues on this process, there are some positive notes that should be taken into consideration. Recurring to the Kubernetes Metrics Server, we realized that the CPU usage on each Kubernetes node would only increase slightly. As we may see on figure 6.4, the initial CPU usage on the cluster, before testing, is over 17% on the node *crowdwire1*, and over 11% on the node *crowdwire2*.

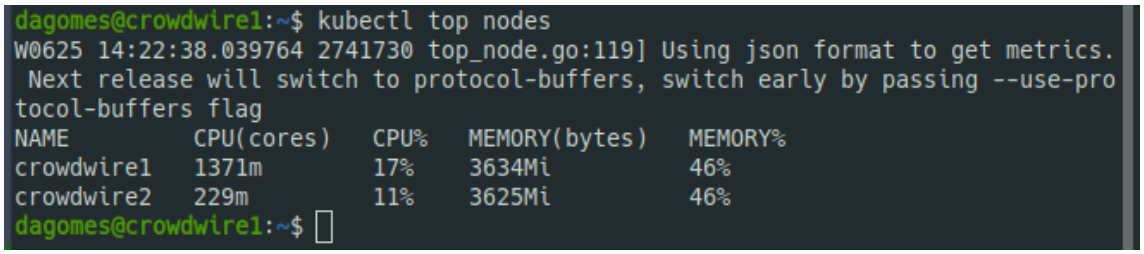

Figure 6.4: Cluster CPU and RAM Usage with no users present.

During the testing, the CPU usage has only increase slightly on each node, represented by the figure 6.5.

```
dagomes@crowdwire1:~$ kubectl top nodes
W0625 14:06:24.401770 2700854 top_node.go:119] Using json format to get metrics.
 Next release will switch to protocol-buffers, switch early by passing --use-pro
tocol-buffers flag
NAME         CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
crowdwire1   1878m        23%    3564Mi          45%
crowdwire2   254m         12%    3626Mi          46%
dagomes@crowdwire1:~$
```

Figure 6.5: Cluster CPU and RAM Usage with over 10 users

With the previous observations, we can conclude that the previously obtained results are an indicator that the infrastructure is not using an enormous usage of CPU resources, which is always a positive take.

6.2 Usability Tests

One of the non-functional requirements that were defined for our project was usability which led us to the decision of doing Usability Tests. To do so, we have proceeded to the creation of a *Google Forms* form with some guidance notes, and, most importantly, which tasks should the users do on the platform. For more context in each of these functionalities, Chapter 7 should be followed. The tasks that users were asked to do were the following:

- Login with the User's Google Account
- Search a World with any online player and joining it.
- Getting close to a user and start video and voice communication.
- Sending a text message using the available chat feature
- Leaving the World and Creating a new World
- Editing the World Created by adding a chair into the world map.

To be able to cover all topics, the following questions were present on this questionnaire:

- "Were you able to complete the task?"
- "Did you need help to complete the task?"
- "How difficult was the task on a scale of 0 to 5? (0 - Very Easy; 5 - Very Difficult)"

After the tasks section, there was another one aimed to collect some data for statistical purposes such as the user's age, sex and academic qualifications, as well as suggestions and problems identified on the platform.

From the form, we managed to do 9 usability tests, from which all users managed to complete all the tasks. The tasks where the user showed more difficulty were the World Search, and also editing a world, since these were tasks rated with that were considered, by one of the users surveyed to have a difficulty of "4" and "3" respectively, the highest among all usability tests. The problems identified by the users were that the platform should have instructions on how to move and interact inside a World, a better pagination system on the world search, and a "drag and drop" functionality on the World Editor. From these problems identified, we have added the suggested tutorials on the platform, while the rest of those, we ended up leaving them a future work into the platform. Overall, the results obtained in this usability test were positive, since every user managed to complete the tasks purposed.

Chapter 7

Project Management

When talking about our project management process, we can divide the process in four major aspects. With these being, the backlog management and the associated processes and tools used to simplify this area, code reviews and how they were processed, the branch workflow and, finally the CI/CD pipelines.

7.1 Backlog

Our progress in the project progress was all managed through Jira and its multiple tools and hooks available. To keep track of the progress made in the project we created a project relative to CrowdWire and as the project was being developed and features implemented, other issues would replace the ones deemed as closed.

When it comes to the strategy adopted, we took on a scrum methodology, so for that reason we opted for the Scrum board, since it was the one that met our needs.

To split the project into smaller, easier to digest, parts, to encourage us to present regular increments to the project, we divided it into sprints. The sprints we created correspond to about two weeks as shown in Figure 7.1.

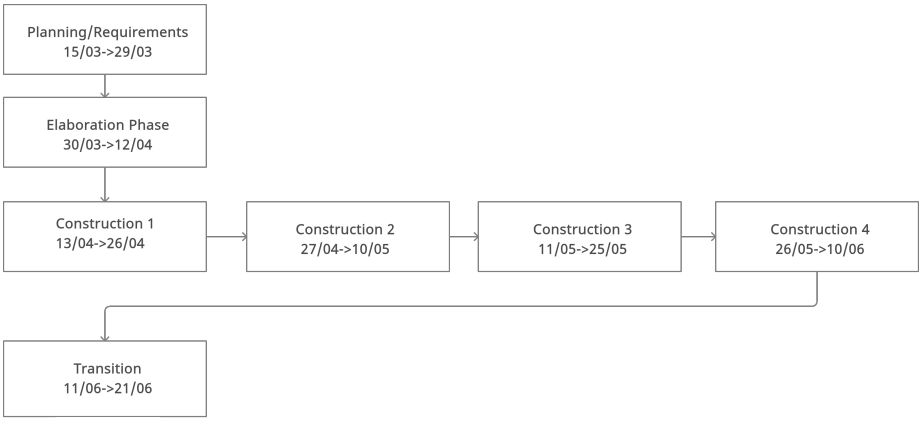

Figure 7.1: Backlog Sprint Calendar

For each sprint, we would create a series of issues that may represent features, issues, improvements needed on the project. At the beginning of each sprint, we would

gather to understand what had been done and what issues could be tackled overtime set for that sprint. With this in mind, we were not always able to address all issues relative to the sprint at hand. That does not mean we disregarded those issues. It means that when the sprint finished, the next one started with the issues relative to the last sprint already assigned to the newly created sprint. Even though we did not always address all issues, there were also sprints whose associated issues were developed faster than we anticipated therefore, we added some more issues to be completed in that sprint.

Most issues were added to sprints right after the sprint transition, but not all issues follow that principle. Some issues added mid-sprint were going to be developed later into the project, which means they were already in the backlog ready to be assigned. Other issues were unexpected and required immediate attention from the development team, which meant the issue should be directly moved to the current sprint and dealt with as soon as possible. These issues were mostly bugs or unexpected necessities that as we were developing.

To complement each issue, we have set up a system that for each issue Jira creates a sub-task, and the user assigned to sub-task should be the one reviewing the pull or pull requests associated with it. This was done through a round robin mechanism that excludes the developer assigned for the issue. For the team to stay up to date with all the the progresses being made, hooks were created between github and discord and also another one between Jira and discord.

The github-discord hook allows the team to know when a push or a pull request took place and if the associated code passed the lint and tests 7.2.

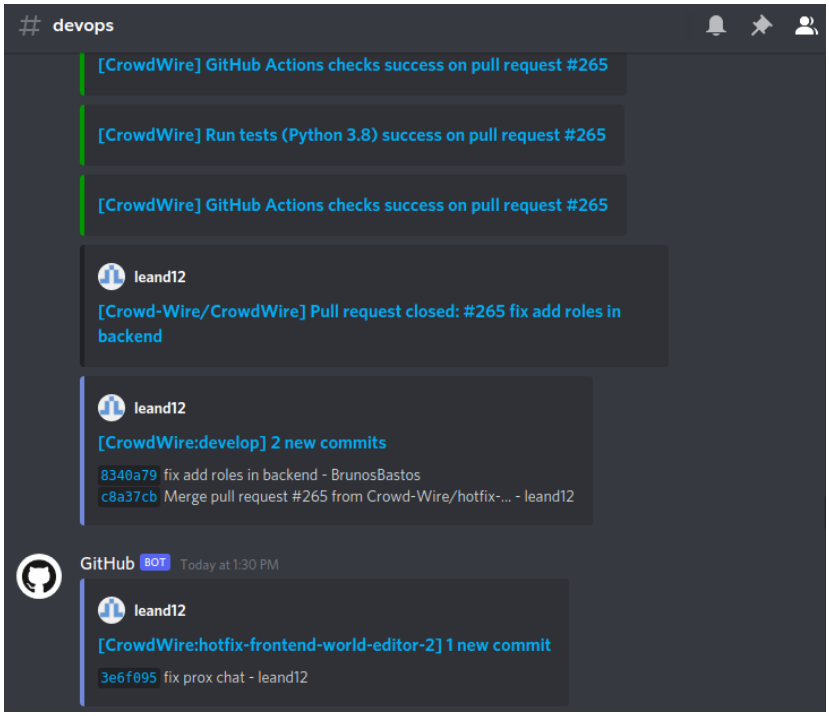

Figure 7.2: Discord Devops Text Channel Snapshot

The Jira-discord hook allows the team to see what changes are being made when it comes to the active sprint. Besides these messages, relative to every update, if the issue is updated to the *Done* section the developer assigned to the sub-task through the round robin is notified on the discord's *backlog* text channel 7.3.

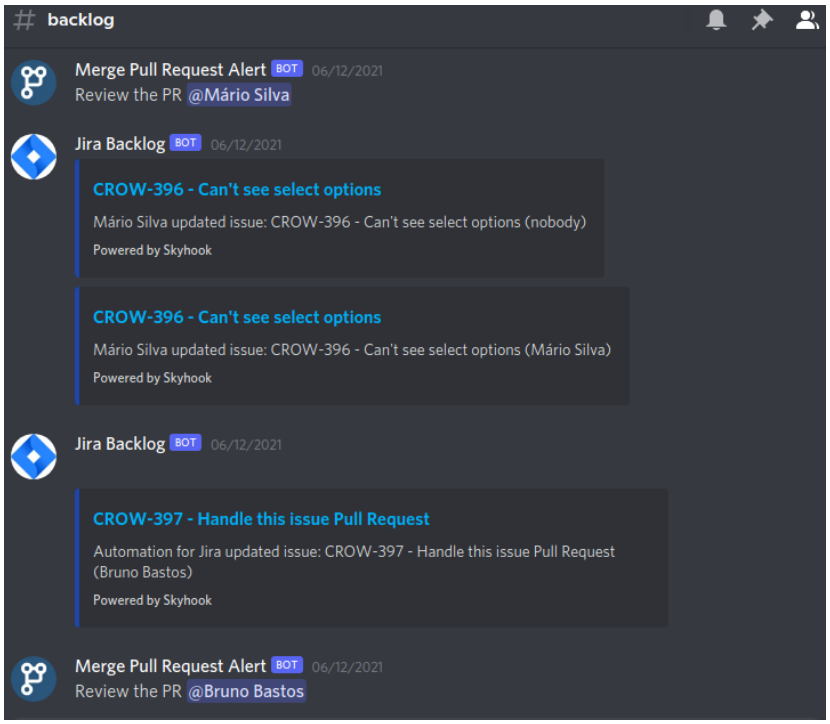

Figure 7.3: Discord Backlog Text Channel Snapshot

7.2 Code Reviews

When reviewing code, first of all, all pushes and pull requests would be subject to the CI pipeline, which will be discussed later on, to check whether or not the projects passed the tests and lints. If both the tests and lint results were positive 7.4, the code would then be evaluated by a group member to add an additional level of assurance to the newly added code as presented in figure 7.5.

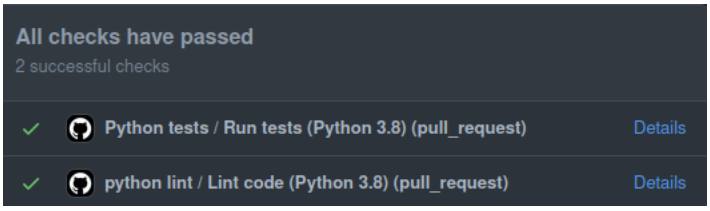

Figure 7.4: Successful Tests and Lint On Pull Request

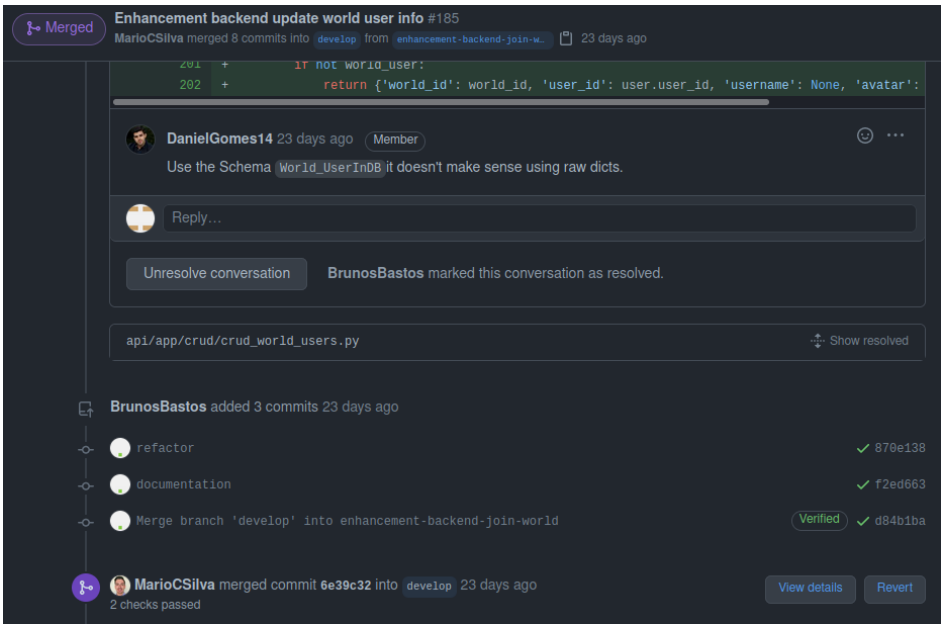

Figure 7.5: Example of Code Review

7.3 Branch Workflows

When it comes to the branch workflow, most branches follow a naming pattern and a logic. This normalization helps with the comprehension of the matter being addressed 7.6.

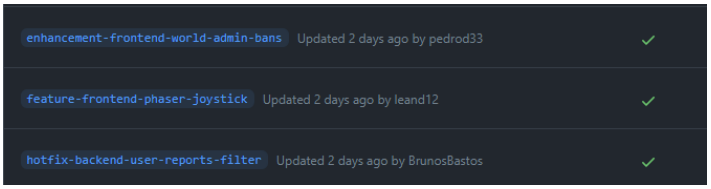

Figure 7.6: Feature branch examples

7.3.1 Naming pattern

For code reviewers to understand what issue is being changed, most branches follow the pattern below:

Issue_type-project_section-issue_name

- *Issue_type*: The *Issue_type* represents the type of issue this branch addresses and it may range from *feature* to *enhancement* or *hotfix*.
- *Project_section*: *Project_section* refers to what section is being addressed, if it is the *frontend* or *backend*.
- *Issuename*: *Issue_name* refers to the issue being addressed by the branch.

7.3.2 Branch logic

When it comes to branches, we are using feature branches in association with the *Git Workflow* strategy. These feature branches may not address an entire feature, yet they solve at least part of the problem. For example, a feature that, in order to be fully completed, needs a connection from the persistence section to the user interface, we would divide the feature branch into a *frontend* part that would address the section closer to the user interface and *backend* that would address all the business logic and connection to persistence.

7.4 CI/CD Pipelines

Concerning CI and CD pipelines, these have been implemented through GitHub Actions, as showed earlier on figure 7.2. We have set up a CI pipeline that runs, firstly, a Python Linter, to verify the python coding standards and style that we defined, and, after this step, it runs Python tests that were written for the REST API module. This pipeline runs on every Pull request created. For our pipeline of Continuous Delivery, as we use Docker to pack our services in containers, we decided to add two important steps in it. The first step consists in the build of the Docker Images for the *frontend*, REST API and the Media Server modules. Once all builds have been done with success, the following step defined was the push of each Docker Image to a Docker Image Registry, *DockerHub*. The use of Image Registries is really important as it allows the use of services in any environment without having to do the build step every time we want to use our services "containerized". As it would not make sense, to run this pipeline on every push to the Code Repository or Pull Request, since there could be any issues within the development branches, we decided to only run this pipeline, on every pull request to the *master* branch. Consequently, for our Kubernetes Infrastructure, by just pulling the latest version of the Docker Images, our services would up and running on the production environment. The figure 7.7 shows up the example of all consequent steps on a pull request to the *master* branch.

Figure 7.7: CD pipeline steps

Conclusion

The principal objective of this work was the development of a clone of *Gather.town*, an online videoconferencing platform that aims to simulate real-world behaviors, able to host events, supported by a scalable infrastructure, allowing for a large number of users to communicate at once.

When we look at the objectives set for this project, it is safe to say that most goals were met, we created a platform that allows users to communicate at a distance simulating to a certain extent reality by providing a sense of being inside the platform, that, with the help of a text, audio and video chat, for nearby-only users increases the realism of the interactions taking place throughout the worlds. Users can also interact with an object placed throughout the world. Each object allows users to perform a range of actions, ranging from games to whiteboards and even file and screen-sharing, to ease the collaboration among users. This project also provides a solution for hosting events. Even though all these goals were accomplished, our solution was not as scalable as we intended it to be. To create such a complete system that approaches so many software areas, starting from the distributed computing, going over Web Development and Design, Multimedia Communications, and many more, only with a good dynamic, communication and dedication from every member of the team, is possible to do so. With this being said, the development of this work was a valuable experience in all levels for each and every member of the group, since we got to be in touch with so many technologies new to us, making the development process a challenge to all of us.

As future work, we would improve our functionalities inside a World by providing more interaction features, greater feedback and tutorials for newbie users (as suggested on the Usability Testing) and support more tools on the Map Editor functionality. Besides this, a greater improvement on the proximity chat algorithm and protocol could be done, to improve significantly the performance of the communications, supported by the media server.

For the communications, there are still a lot of improvements that could be done to increase the performance of massive meetings, however, massive video and audio conferences are a very complex branch of communications and require a lot of time to implement any of them. For example, a TURN Server, that relays all the traffic between peers even if they cannot connect directly, which is useful in our case because even though peers can always connect through our signaling server, some firewalls block UDP packets and some may even block TCP as well, in contradiction it adds latency which is another factor that made us not implement it.

Another improvement to the project would be to have a broader range of tests that allows developers to find bugs and unintended behavior more easily.

When it comes to statistics related to worlds and the platform itself, there could be a wider set of data available, such as object interactions and conference room entries.

Bibliography

[1] *React.js Documentation*
https://reactjs.org/docs/getting-started.html

[2] *Phaser Documentation*
https://photonstorm.github.io/phaser3-docs/

[3] *FastAPI*
https://fastapi.tiangolo.com/

[4] *Mediasoup*
https://mediasoup.org/

[5] *Sandy, J. (2021, January 4). Choosing between Django, Flask, and FastAPI*
https://www.section.io/engineering-education/choosing-between-django-flask-and-fastapi/

[6] *RabbitMQ. Which protocols does RabbitMQ support?*
https://www.rabbitmq.com/protocols.html

[7] *Redis*
https://redis.io/

[8] *NGINX Inc. What is NGINX?*
https://www.nginx.com/resources/glossary/nginx/

[9] *Kubernetes*
https://kubernetes.io/

[10] *Kubernetes Documentation*
https://kubernetes.io/docs

[11] *Getting started with.* (n.d.). WebRTC.
https://webrtc.org/getting-started/overview

[12] Castillo, I. B., & Millán, J. L. (n.d.). *Mediasoup :: Scalability.* Mediasoup. (2021, June 24) .
https://mediasoup.org/documentation/v3/scalability/

[13] André, E., & Breton, N. L., & Lemesle, A. & Roux, L., & Gouaillard, A. (2018, October) *Comparative Study of WebRTC Open Source SFUs for Video Conferencing.*

[14] Castillo, I. B., & Millán, J. L. (October 2018). OpenSIPS Summit 2018. *Building Multi-Party Video Apps with Mediasoup*

[15] Iyengar, M. (2021, March 10). *WebRTC Architecture Basics: P2P, SFU, MCU, and Hybrid Approaches*. Medium.
https://medium.com/securemeeting/webrtc-architecture-basics-p2p-sfu-mcu-and-hybrid-approaches-6e7d77a46a66

[16] *Pydantic*
https://pydantic-docs.helpmanual.io/

[17] *SQLAlchemy Documentation.*
https://docs.sqlalchemy.org/en/14/

[18] *AioRedis Documentation.*
https://aioredis.readthedocs.io/en/latest/

[19] Sookocheff, K. (2019, April 4). *How Do Websockets Work?* Kevin Sookocheff.
https://sookocheff.com/post/networking/how-do-websockets-work/

[20] *CertManager*
https://cert-manager.io/docs/

[21] *Kubernetes Metrics Server*
https://github.com/kubernetes-sigs/metrics-server