GMIT
Galway-Mayo Intitute of Technology
B.Sc.(Hons) in Software Development

[Project Dome]

Albert Rando
Pedro Henrique de Oliveira Mota

Advised by: Dr. Patrick Mannion,
Department of Computer Science and Applied Physics,
Galway-Mayo Institute of Technology (GMIT).
Submited April 2018

Abstract

Text of the Abstract.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Objectives

Motivation and Objectives here.

1.2 The Solution

Contributions here.

1.3 Scope

Statement here.

1.4 Publications

Publications here.

Chapter 2

Methodology

2.1 Planning Phase

Before a project is started, it is necessary to plan ahead and organize in order to ensure that development is carried out with as less contingencies as possible. It is highly important when working in a team, to restrain ourselves from starting coding straight away with poor planning thus it would result in members of the team overstepping each other, repeating code and hindering the overall evolution of the project.

As we started, we met in order to put our ideas in place and come up with a suitable idea for the project that would appeal both us and our supervisor, to deliver a product with the quality expected as fourth year students. To do so we conducted a few brainstorming sessions where we developed what was the germ of the product.

[display idea somehow]

The first step then was to investigate the appropriate technologies, layout the structure of the project and decide on the tools we would use in order to manage the project.

However after some investigation and given the scope that kind of project has and the amount of people needed to carry it out we decided to switch our focus from what we started with to a project that included a small scale game but shifting our resources to provide extra functionality built around it demonstrating our prowess in multiple areas. After further deliberation and more investigation, we designed a mind map to plan our project and help us focus our efforts into a productive direction.

We used coggle to create a visualization of the interaction of the different components we planned to include in the project. In it we can see how it branches into smaller components breaking down from higher level components into lower level implementations, subject to variation, defining a mind map to achieve the final product.

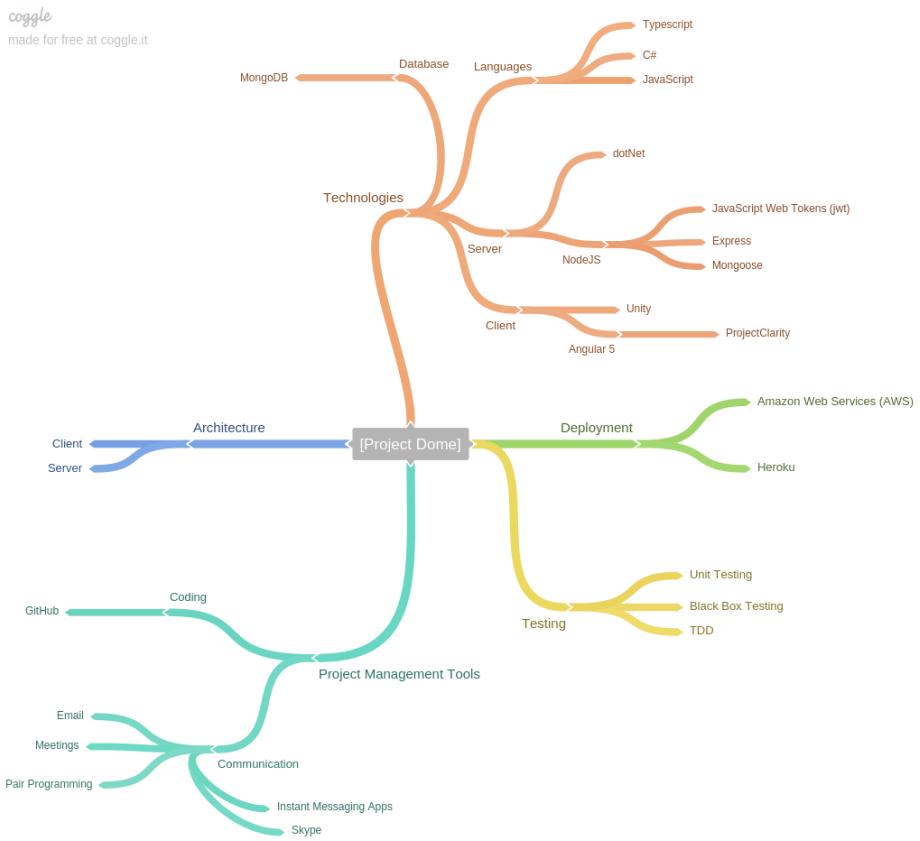


Figure 2.1: Mind Map

- Architecture: At a high level, what archetype are we going to apply in our project.
- Technologies: The different technologies available to us that will provide the necessary tools to produce the results we want. That includes the languages we are going to use, the frameworks we are going to work with and the different environments we are doing to develop in for the sake of our project.
- Project Management Tools: The management tools for unifying criteria, centralizing code and defining how the team members are going to communicate with each other.
- Testing: Testing techniques to apply to our project and the idea of TDD behind the code, to produce a better, more robust code.
- Deployment: The options we considered in order to deploy our solution/project.

2.2 Methodology

We determined to implement an amalgamation of several methodologies for the development of our project, giving us more room to experiment and play around so we could find the most suitable way to execute and

implement distinct tasks. We used mainly Agile mixed with XP Programming, for example, by using pair programming.

Our major take on Agile was the iterative approach it favours along the strong prototyping factor that allow the application to grow after every stage, starting with a working prototype and working all the way up, implementing new features while maintaining a working product. Also the looseness of it made it perfect due to the small size of our team, while not having the heavy burden of the documentation allowed us to work at a faster pace.

2.2.1 Requirements Gathering

This phase involved gathering the requirements necessary to implement our solution. Since it was not an industry project, we had to investigate, brainstorm and use our personal experience to gather a set of requirements that would help us design the solution the way we wanted.

2.2.2 Requirements Analysis

Once the requirements were gathered, it was important to analyze them and break them down to stories we could use to build our solution, as well as to resolve the priority of the tasks. This process also helped discarding the requirements that made no sense in relation to the rest of the product or correcting the ones that did not enter in the scope of the project at a first glance but offered interesting design options that could be implemented in further iterations.

2.2.3 Product Design

After we have gathered our requirements and analyzed them, we were ready to start designing the layout of the application, like how and where the decided technologies would come into play and a high specification of how they were going to interact with each other. We also draw rough sketches of how the different parts of the application might look like to create mock ups for the parts of the system that would take longer to develop. We divided the tasks into cards, and discussed who would take care of which tasks and what we were going to work on in every Sprint. Due to the many of the college projects and exams we had to organize what we called ”variable length sprints” where we organized the duration of the sprint based on the tasks and the outside factors that influenced our productivity.

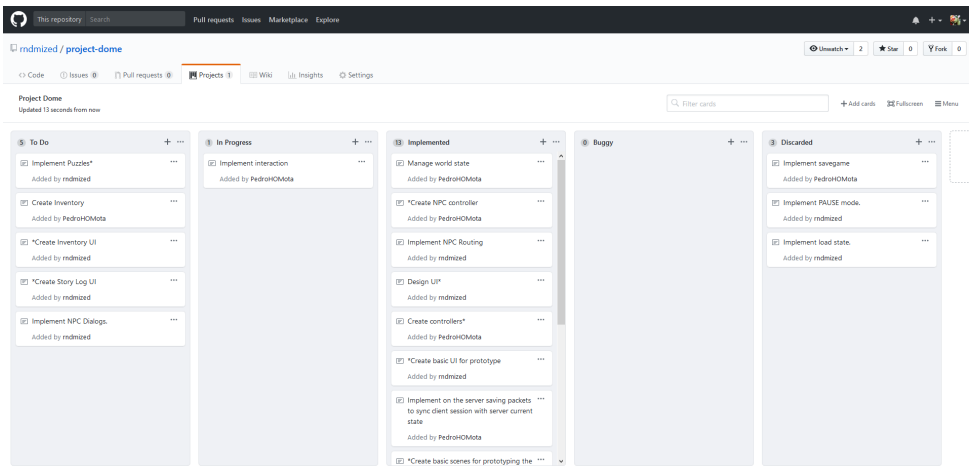


Figure 2.2: Github Cards

2.2.4 Development

Having the tasks, divided into Sprints, all that was left was to immerse ourselves into the development of the solution. Issues arisen during every sprint where either resolved by using pair programming or, if they were not critical, taken into account and pushed to a next iteration. After each sprint we tested out the solution, reviewed the accomplished tasks and prepared to take on the next iteration.

2.2.5 Testing

Intertwined with the development and implementation of new tasks, we where keen to apply TDD in order to prevent the solution from breaking. To do so we created specific test cases for the software we developed each sprint thus re-writing the bits that did no pass the test. We also had a strong focus on unit testing. However, the most common type of testing applied was without doubt black box testing.

Black Box testing is a method that focuses on examining the functionality of an application, or the part of the application we are testing, without checking the internal structures or the actual implementation. An example of that might be any of the input fields used in the client side of the application. By testing whether the text the user wrote into the input fields (input) produced the expected result, which was receiving such input somewhere else (as an output). Even though this is not an exact example, because it is testing more than one feature at once, it is useful to illustrate the point.

2.2.6 Deployment

After developing and building the solution right, the project had to be deployed. We chose.....[TODO]

2.3 Project Management

Managing a project properly is an essential components to any development effort as it shapes the structure of a project, and helps to plan efficiently, allowing us to end up with robust product, carefully developed. The sheer scope of the project required from us a carefully planned path to follow in order to deliver successfully the product we wanted, even though because of such scope we had to cut off some features, but in a manner that would allow us to implement them in further iterations if the time to do so was given. One of the keys of project management is communication. The communication not only has to be effective but it also has to be efficient. The advantages of working in a small team, and being in the same area sharing the same facilities, eliminated most of the unnecessary communication thus making us work more efficiently. By planning the project before diving into coding, and meeting to decide the features to implement in every sprint, reduced greatly the need of communication tools and by using pair programming we were able to be more effective and solve issues faster.

2.3.1 GitHub

We used GitHub to host our repositories containing the code, and used it as our version control manager. Due to the fact that most of college projects require the use of Github, so we had some prior experience with it, and the many possibilities it offers to track our tasks and commits made us choose it as our preferred tool.

2.3.2 Discord

Discord is node based communication tool that allows for VoIP, as well as screenshots and text sharing, and also screen sharing over a network. We have used extensively when we could not be in the same premises but we still needed to work together.

[DISCORD SCREENSHOT WITH SOME IMAGE SHARING]

2.3.3 Google Drive One Drive

For relevant files and documents that we decided to not include in our repository such as some learning resources, and some side documentation for ideas or concepts we used Google Drive or One Drive to share them over the cloud.

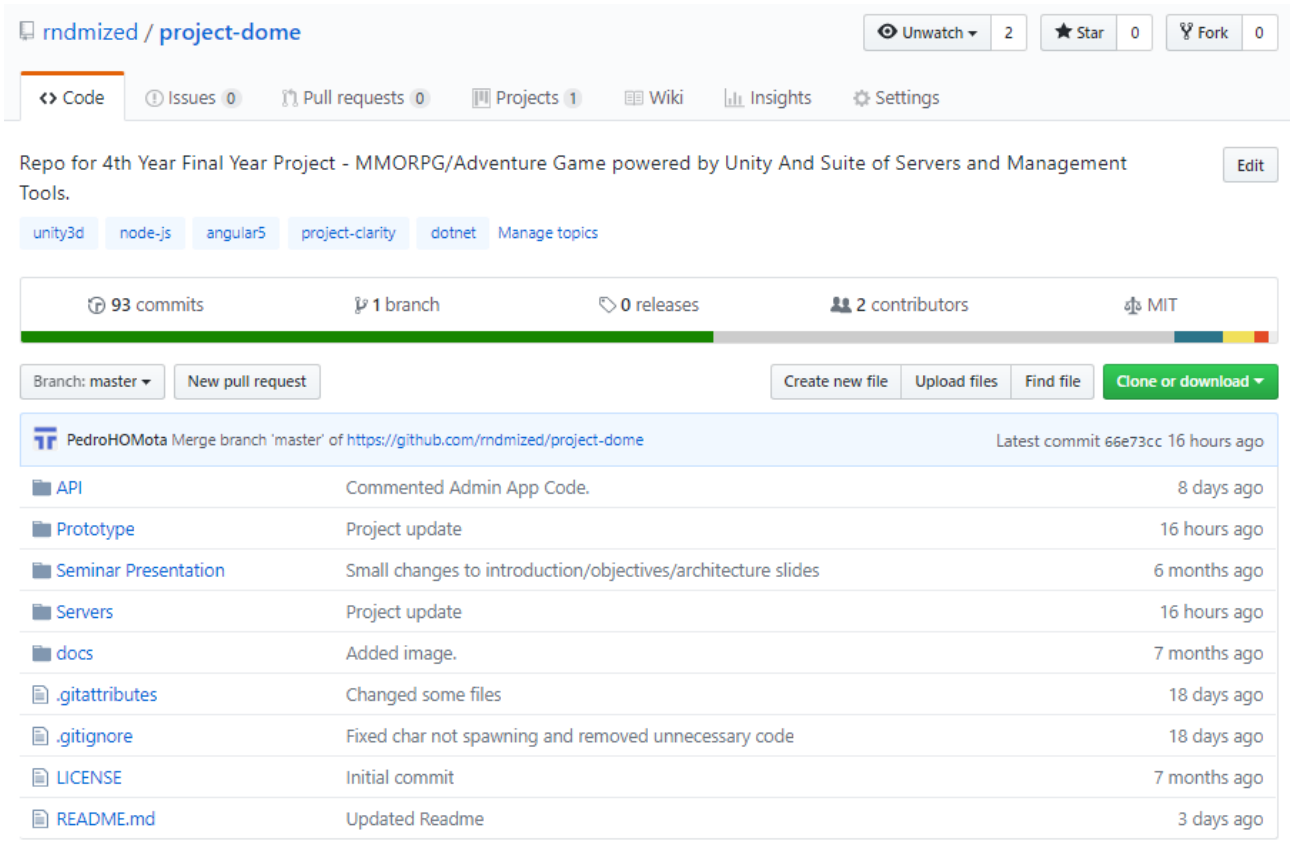


Figure 2.3: Github Screenshot

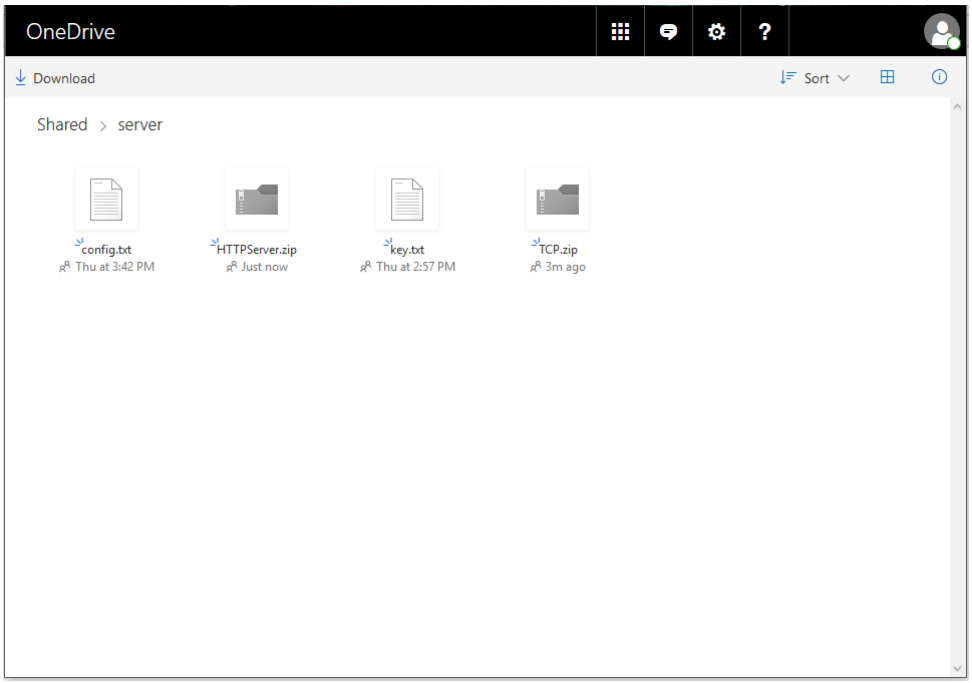


Figure 2.4: One Drive Screenshot

Chapter 3

Implementation

3.1 System Architecture

Project Dome's suite of applications have been developed from scratch, studying the requirements and carefully planning every sprint. By analyzing the requirements we determined the necessary components of our system as well as the spectrum of technologies available to us in order to carry them out. We decided to modularize the system components so the could be changed, or scaled minimizing the possible negative effects on the overall structure. Since we needed Game Clients to synchronized with each other we decided to offload the Authentication services along the Player Data to a separate server. Both servers would read and write to the database. Then a web app would manage both the Game Server and the Authentication Server. To illustrate this point, the bellow figure diagram shows a high level representation of the System Architecture. How and why we make those design decisions will be thoroughly explained throughout the rest of this chapter.

To roughly explain the relationship between the elements in the diagram above, and to give an insight on what we perceived as the flow of the application, we will use an example that will help visualize it and add some context to it.

To start of the example we divided the users into two differentiated categories, with different roles. To start off we have Players, they will be the ones interacting with the Game Client. After that we will have the Administrators, they will interact with the applications through the Management App.

So, who are the Players?

Players are the users that will play the game, enjoying it as much as possible. The tasks or challenges they will have to face in-game will depend on the game they play. In this case our focus was on creating a structure that would allow multiple players to join a persistent world server rather that developing a full game.

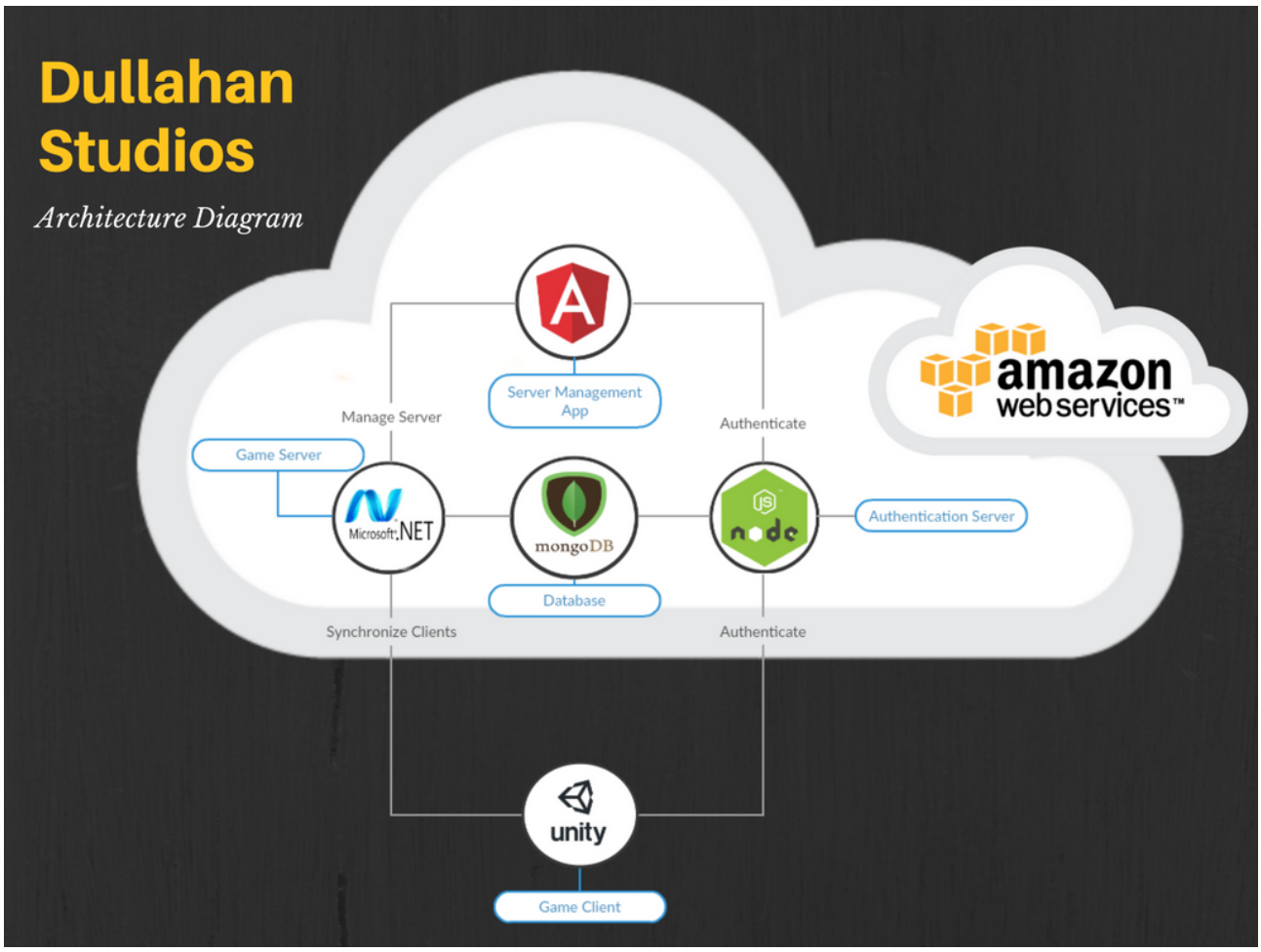


Figure 3.1: System Architecture

How do the players access such persistent world to begin with?

For the players to access the world game server they will need an avatar, or character, that will be their representation in the online world. To do so they require an account that would make them identifiable to the rest of the players as well as the Administrators. To create a character, the player will need an account, password protected that will ensure their identity while online. The game client provides the means to create a new account provided you give the required information. After that they will be able to log in into their account by using their credentials.

How will the players differentiate one from another?

When the players log in to their accounts they will be allowed to create a character by choosing between the available options. When the players connect to the game server, that information along their names and user names will be passed to the server and broadcasted to the rest of clients.

How many players will be allowed in the server at a given time?

The answer to that question is variable. The among of concurrent players at a given time will depend, of course,

of the capability of the server running the game server, but it is also an option that can be customized in the server options (Administrators can change it as we will see later).

What happens when a character connects to the game world?

As we mention before, the information about the character connecting will be broadcasted to the rest of the clients connected, besides that, the connecting player, will receive the information about the rest of the players connected as well. That way players will synchronize with each other through the server allowing them to see each other roam through the world.

What data will the system store about the player?

It could be anything! In our game, we only store the amount of played time on a session and the total amount of played time for a given character as to demonstrate some of the information that can be obtained or used.

Where is the data stored?

The data is stored in a database. The database stores both information about the users and their characters.

What about Administrators?

Administrator accounts are not so easily created. In order to change an account to be and administrator it has to be changed in the database. Since there are no available means to do that for the regular user it means new administrators cannot be created. Only the administrators of the solution will have access to create administrator users. Administrator users cannot log in the game, however, unlike player users, they can log into the Management App.

What is the Management App? What does it do?

The Management App is a web app that can be accessed from any web browser though the network. It requires the user to be an administrator in order to log in, and once inside it allows administrator users to perform some tasks. Some of those tasks are:

- Obtain information about the players.
- Ban players from entering the game.
- Pardon banned players.
- Obtain information about the players that are currently online and their characters.
- Kick players from the server.
- Change server properties such as: port number, maximum number of concurrent players.
- Restart the server.

3.2 Architecture Implementation

Breaking down the system into different levels that correspond to various parts of each procedure, we find more operations running in the background that what a user might perceive. Data is passed by the user when they log in, or when they create a character, or when the administrator kicks a player from the server. The communication occurs and it is handled on another level.

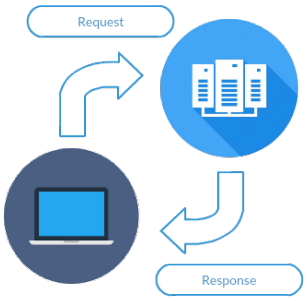


Figure 3.2: Client-Server Architecture perceived by the user

In the previous figure we can appreciate the communication between the client and the server. Since we need different types of communication for different types of procedures we will start with the most simple of the two, HTTP (Hypertext Transfer Protocol). We make use of HTTP requests and HTTP responses to authenticate the application users with the log in server, or to connect to the game server in order to obtain some live data. The request is sent from the client to the server where the request will be processed and the server then responds returning a response back to the client that will deal with it accordingly.

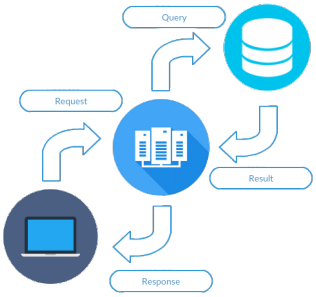


Figure 3.3: Client-Server Architecture

As we can see, when the server receive the request it processes it, for example querying the database which will return the result for that query back to the server, who will then produce an appropriate response to send back to the client.

3.2.1 HTTP

The Hypertext Transfer Protocol (HTTP), as defined by Fielding et al.[1], is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

In the RFC2616, the protocol is detailed and explain and the type of request and responses are defined. For the purpose of our solution we mainly focused on two.

GET Requests

GET Requests are used to retrieve data. That could be a new page the user is trying to access or some data that needs to be displayed in the current page. If we pass some arguments, these can be encoded in the URL since GET request do not possess a body.

POST Requests

POST request can be used to obtain some data as well, however POST requests possess a body that can be used to send data to the server. This would be for example, the credentials of a user when it is logging in, or the fields of a registration form that the user sent to the server to create an account.

3.2.2 MVC: Model, View, Controller

Modelviewcontroller (MVC) is an architectural pattern commonly used for developing user interfaces that divides an application into three interconnected parts. MVC is a very popular way of organizing your code. The big idea behind it, is that each section of your code has a purpose, and those purposes are different. Some of your code holds the data of your app, some of your code makes your app look nice, and some of your code controls how your app functions. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development. In our solution MVC can be seen implemented in the Management App, the way Angular 5 is structured follows the MVC pattern quite clearly.

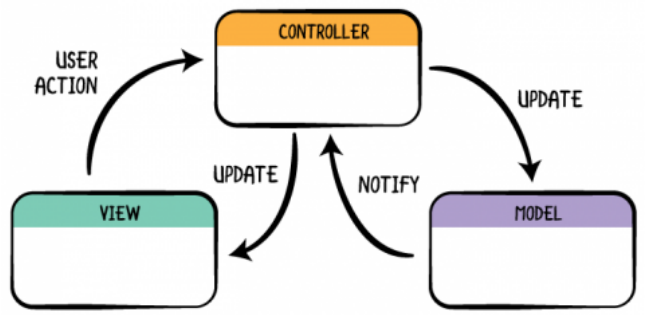


Figure 3.4: MVC diagram

The parts of MVC are the following:

Model: Model code typically reflects real-world things. This code can hold raw data, a procedure to obtain such data or define the essential components of your app. That would be the services that request data to the server.

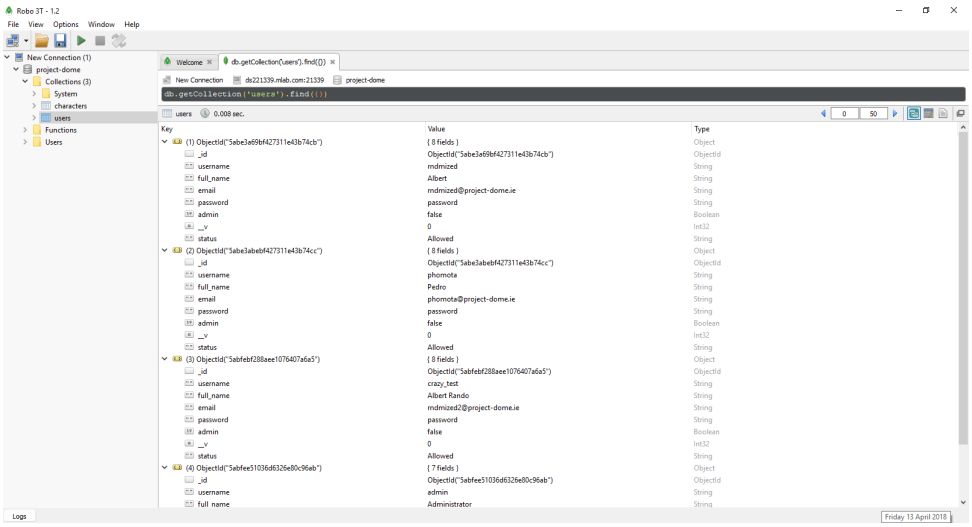


Figure 3.5: Data on MongoDB

View: View code is made up of all the functions that directly interact with the user. This is the code that makes your app look nice, and otherwise defines how your user sees and interacts with it. This would be the HTML code along the CSS of the application.

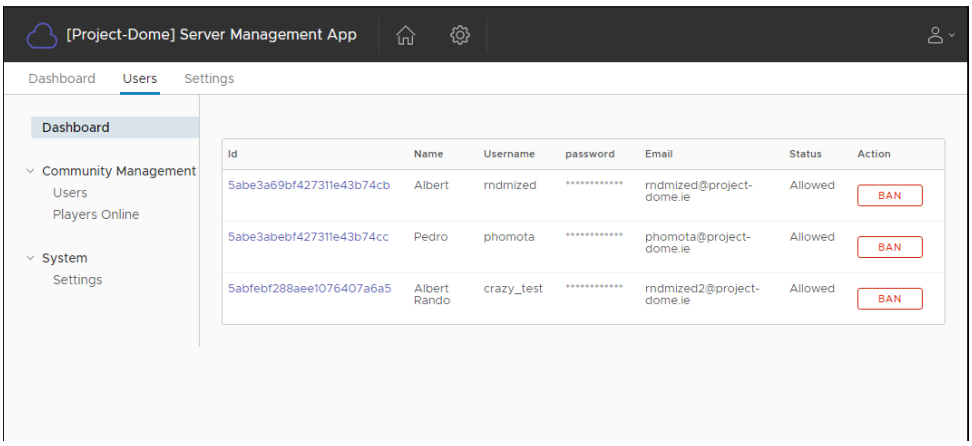


Figure 3.6: Management App View

Controller: Controller code acts as a liaison between the Model and the View, receiving user input and deciding what to do with it. Its the brains of the application, and ties together the model and the view. The Typescript code associated to every component would fulfill this role acting as an intermediate between the view and the appropriate service.

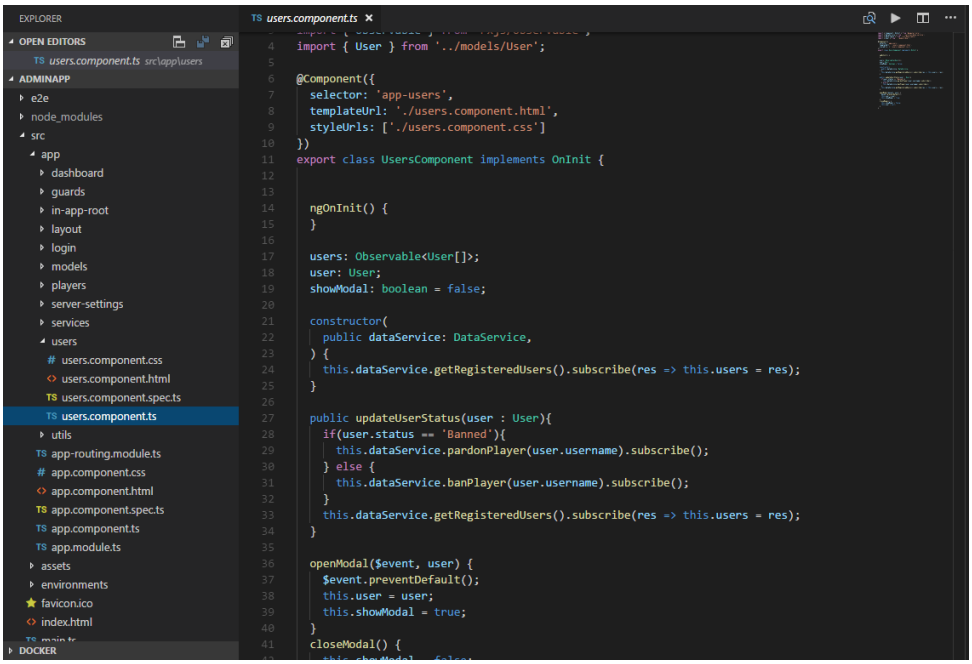


Figure 3.7: MVC Controller

3.2.3 The Unity game client

The game client is written in C# and takes advantage of the Unity Engine. Unlike more traditional approaches, and even though unity itself do not forbids them, it shies away from the MVC / MVVC models, the

common treatment of classes, inheritances and so on towards a different type of approach. It mixes Component-based software engineering (CBSE), also called as component-based development (CBD), which is a branch of software engineering that emphasizes the separation of concerns with respect to the wide-ranging functionality available throughout a given software system. It regards components as part of the starting platform for service-orientation. Components play this role like in service-oriented architectures (SOA). However it emphasizes the individuality of the components. This is accomplished by treating every component as a separate game object, attaching scripts to it, and reusing the engine capabilities that can be inherit from Mono Behaviour to perform any necessary task. The diagram bellow illustrates at a very high level, how some of the game components interact with each other, yet the diagram itself does not follow the UML conventions because the classes do not usually interact with each other like classes (composition, aggregation, inheritance,..) but through referencing other components or game objects.

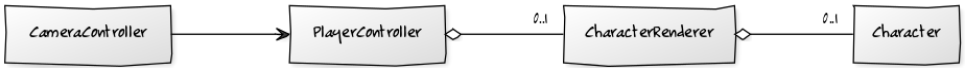


Figure 3.8: Character Game Object Components

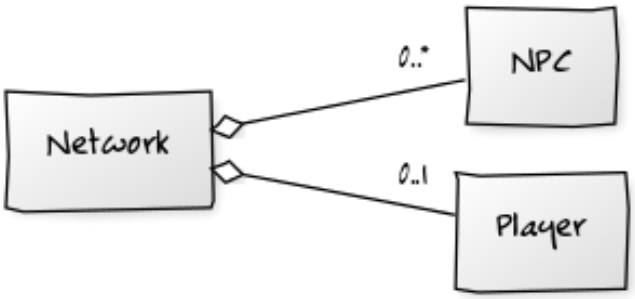


Figure 3.9: Network Game Object Components

3.2.4 RESTful APIs

For the log-in server we applied the RESTful API principles. REpresentational State Transfer (REST) is an architectural style that defines a set of constraints and properties based on HTTP. Web Services that conform to the REST architectural style, or RESTful web services, provide interoperability between computer systems on the Internet. REST-compliant web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations. Other kinds of web services, such as SOAP web services, expose their own arbitrary sets of operations.The term was coined by Fielding in his doctoral dissertation in 2000[2].

The RESTful architecture presents ourselves with a set of constrains, that must be accomplished in order to achieve RESTfulness: Client-server architecture, statelessness, cacheability, layered system and a uniform interface. Is this last concept, the most fundamental one to the design of REST services. It has four specific constrains:

- Resource identification: Resources are identified in the requests, an example of it would be URIs in web-based systems.
- Resource Manipulation through representations: Representation of resources hold enough information about it to modify or delete it.
- Self-descriptive messages: Each message includes enough information to process it, disregarding any state.
- Hypermedia as the engine of application state: Having accessed an initial URI for the REST application a REST client should then be able to use server-provided links dynamically to discover all the available actions and resources it needs.

Additionally we implemented JSON Web Tokens (or jwt) to provide a layer of security on top of the application, by requiring a valid token in order to access certain routes in our API. JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties.[3]

3.2.5 Databases - NoSQL

As part of the solution we required a form of information / data storage organized in a way that could be easily accessed, managed, and updated. It had to perform the basic CRUD operations and it had to be malleable enough o embrace changes during the process of development, while maintaining the capacity to scale as it were needed. When compared to traditional relational databases, NoSQL databases are more scalable and provide superior performance, and their data model addresses several issues that the relational model is not designed to address. With that in mind we decide to use MongoDB as our database. As a document storage it allows to store documents, that can vary from one another thus providing a great deal of adaptability, letting the contents of the database to change. Mongo is not as constrained as a more classic SQL databases, and does not require the same upfront design making it perfect for rapid development and prototyping.

3.3 System Design

Upon decided on the architecture, there are more aspects that must be reflected on. The components, data and interfaces need to be defined in order to build the system. This process is what it is called System Design. We have determined our requirements and built the stories so it comes the time to implement the functionality required. We are going to break down the system in its different high-level components and we will then proceed to break them down in detail and explain their implementation along the reason why we decided to implement them in such manner.

3.3.1 Log in Server

The log in Server APIs main goal is to authenticate users and provide them with the means to access their application data. The server uses Node.js and express. Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications providing a thin layer of fundamental web application features, without obscuring Node.js.

API Entry point

The API entry point is **LoginServer.js**. It define a set of constants (middleware) and creates a listener waiting for incoming requests. The following figure is a partial depiction of the API entry point.

```
const cors = require('cors'),
http = require('http'),
jwt = require('jsonwebtoken'),
express = require('express'),
mongoose = require('mongoose'),
config = require('./config.json');
/* config.json contains database address */
const app = express();
/* Database driver connector */
mongoose.Promise = global.Promise;
mongoose.connect(config.database);
/* This are the route files */
app.use(require('./GameClient/ClientRoutes'));
app.use(require('./AdminApp/AdminAppRoutes'));
 /* Create the listener */
http.createServer(app).listen(port, function (err) {
  console.log('listening in http://localhost:' + port);
});
```

Figure 3.10: API entry point (partial)

As we can see the routes have been split into two differentiated files. **ClientRoutes** will handle the requests coming from the game client while **AdminAppRoutes** will manage the requests from the Management App. The routes have been separated for maintainability and better organization within the API. API file structure is the following:

```
./
├── AdminApp
│   └── AdminAppRoutes.js
├── GameClient
│   └── ClientRoutes.js
├── Utils
│   └── TokenValidator.js
└── models
    ├── Character.js
    └── User.js
```

Models

Before we start with the routes, we will talk about the models. Character.js and User.js are predefined schema, used by mongoose to store and retrieve data from the database. They are the definitions of the documents contained in mongo and are effectively what defines the document collections.

```
/** Database Driver */
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
/** Defining User Schema */
var UserSchema = new Schema({
    username: { type: String, required: true},
    full_name: { type: String, required: true},
    email: { type: String, required: true},
    password: { type: String, required: true},
    status:{ type: String },
    admin:{ type: Boolean}});
/** On user creation initialize some values */
UserSchema.pre('save', function (next) {
    var user = this;
    user.admin = false;
    if (!user.status) { user.status = 'Allowed'; }
    next();
})
/** Export Schema */
module.exports = mongoose.model('User', UserSchema);
```

Figure 3.11: User schema definition

The fields where required is true will return an exception if they are not provided on trying to save a user in the database. Obviating the most common properties of a user, this is the reason behind them:

- **status**: This field determines whether the user is Allowed, Banned or other future status yet to define. Default is 'Allowed'.
- **admin**: This field determines whether the user is and Administrator or not. Default is false.

Full name, user name, email and password store the information that they imply. Character.js structure is very similar, however the fields change to store the relevant data to characters belonging to a player (which is a non-admin user). The properties of a Character as defined in the schema are:

- **userID**: The id of the user that owns this character.
- **char_name**: The character's name.
- **char_hairId**: The id of the hair asset for this character.
- **char_bodyId**: The id of the body asset for this character.

- **char_clothesId**:The id of the clothes asset for this character.
- **score**: The only unrequired field. It is a dummy value used to illustrate other properties that can be added to a character. Depending on the game they could be just a score like this or they could be statistics or any relevant information for a character in that game.

Token Validation

```
module.exports = function validateToken(req,res, next){
    const bearerHeader = req.headers['authorization'];
    if(typeof bearerHeader !== 'undefined') {
        const bearer = bearerHeader.split(" ");
        const bearerToken = bearer[1];
        req.token = bearerToken;
        next();
    } else {
        res.sendStatus(403);
    }
}
```

Figure 3.12: Token Validation

On the folder utils we can find the JavaScript file TokenValidator.js. In order to prevent unauthorized access to sensible routes, we used JSON web tokens to implement a layer of security. In order to determine whether a request is allowed to access certain routes there is a prior validation. If the request contains a token in the header the request is allowed to continue to verify the token, if not, it will be responded with a *forbidden* status code.

Routes

Every route JavaScript file imports the necessary components in order to respond to the appropriate requests.

```
/** Routing */
const express = require('express');
const app = module.exports = express.Router();
/** Utils */
const jwt = require('jsonwebtoken');
const tokenValidator = require('../Utils/TokenValidator');
/** Models */
const User = require('../Models/User');
const Character = require('../Models/Character');
/** For demonstration purposes: Key */
const tempSecretKey = '********************';
```

Figure 3.13: Routes files header

We will proceed to explain briefly how the routes work before detailing each route separately:

```
app.post('/login', function (req, res) { /* ... */ }
app.get('/getCharacterList', function (req, res) { /* ... */ }
app.post('/createCharacter', tokenValidator, function (req, res) { /* ... */ }
```

Figure 3.14: Routes examples

The first route is **'/login'**, following the route there is the callback function with the request and response arguments. Since it is a post, the body of the request is accessible through the 'req' parameter. Inside the function we can process the request, access the database, compare the password and return a response, along a status code or JSON as we see fit.

The second route, **'/getCharacterList'**, very much like the first one allows you to access the request as well as the response, however due to it being a GET request, the 'req' parameter does not contain a body.

The third route, **'/createCharacter'**, is slightly difference since it is a 'protected route'. The request has to pass the tokenValidator function (previously described) in order to be processed. If the request does not contain a token the API will respond with a *403* code, if it contains a token it will be allowed in the route.

Client Routes

As we previously mentioned, **Client Routes** contains those routes that are related to the client's requests.

- **'/login'**: Takes in user name and password from the Request body. Looks for user in the database, If the result of such query returns a null or an error, return JSON error message. Else, compare the password stored against password provided and determine whether the user is validated. If it is return token.
- **'/register'**: A low level validation is performed in the client. Checks whether all field are complete, as well as if there is an existing account with a given name or if the email address is already on use. Returns JSON containing the success of the request, a message, and a code. On failure the returning message will be displayed on the client.
- **'/createCharacter'**[Protected]: Checks whether the request header contains an authorization token or if such token is valid. Once the token has been checked, check the number of characters for a given player. If such count is, or exceeds, 4, the maximum number of slots for characters for the given user has been reached thus not allowing the creation of a new character. (There is a low level validation implementation of this in the client, but it has been added to the server as a fail safe). If the count is lower than 4, then the creation is valid, check if the appropriate information for character creation has been sent and proceed with storing character in the database for future use.
- **'/getCharacterList'**[Protected]: Checks whether the request header contains an authorization token or

if such token is valid. Once the token has been checked,query the database to retrieve the character's data for a given player. If successful returns JSON containing the character's data.

AdminApp Routes

AdminApp Routes contain the routes related to the Management App such as:

- **'/loginAdmin'**: Takes in user name and password from the Request body. Looks for user in the database where the user is also an admin, If the result of such query returns a null or an error, return JSON error message. Else, compare the password stored against password provided and determine whether the user is validated. If it is return token.
- **'/getRegisteredUsers'**[Protected]: Checks whether the request header contains an authorization token or if such token is valid. Once the token has been checked, return a list of non-admin users.
- **'/banUser'**[Protected]: Bans a given user of entering the game. Effectively sets user's status to Banned. Returns a success message.
- **'/pardonUser'**[Protected]: Changes user status to Allowed. Returns a success message.

3.3.2 Game Client

The Game Client provides one of the front end layers of our solution. It is the game that users will register for, log in, create their characters and connect to the game server to interact with the rest of character connected. In order to do so we used the Unity Engine, as for the language of preference, even though Unity can use JavaScript as well, we decided to use C#. As mentioned before, Unity favours a less Object Oriented approach which makes it difficult to document using traditional tools and diagrams like UML. Even though Unity is pretty modular and has an asset store where different components can be downloaded, we refrained from doing so, only using graphic assets in the client, writing all the scripts used in order to obtain a more customized project. We will start describing every scene and what components contains, as well as the service it provides.

The Log In Scene

The log in scene requires the user to input a user name and a password in order to proceed into the game. Where the user not created an account yet it provides a clickable button (text in blue). If ran on full screen the user can exit the game by clicking on the button in the top right corner.

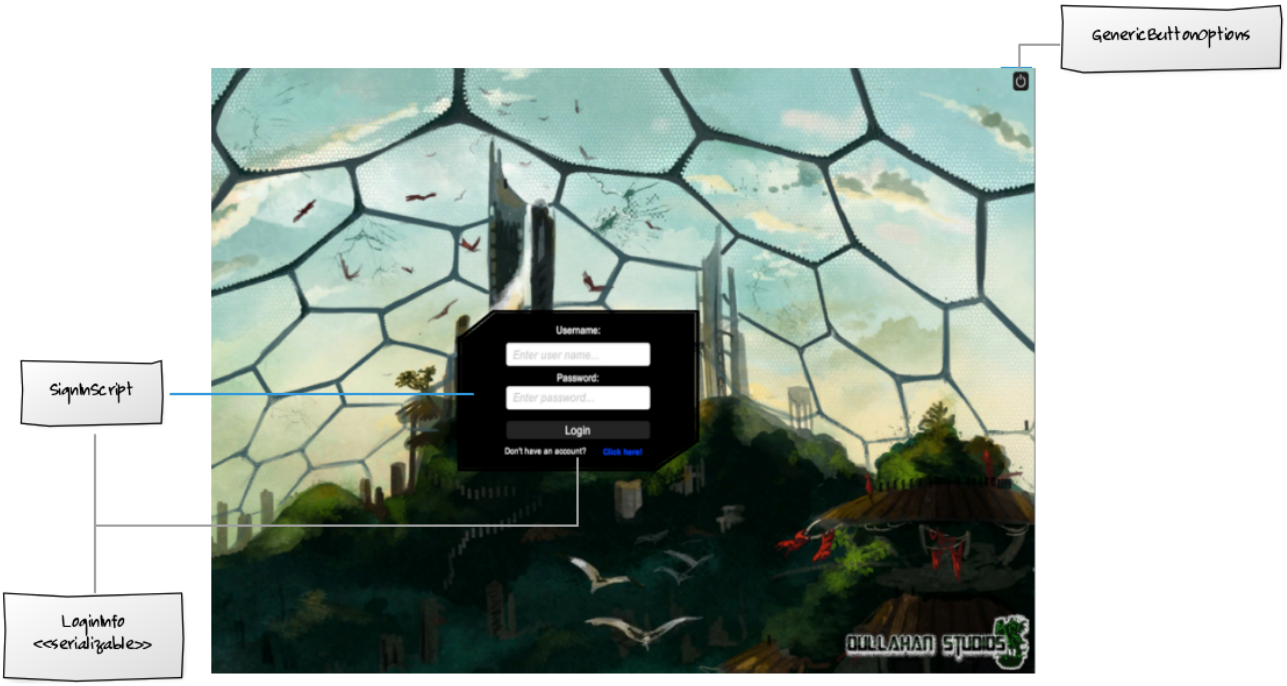


Figure 3.15: Log In Scene

This Scene contains a SignIn Script that provides a low level validation for the input fields, and sends a http request to the log in server (our API) in order to authenticate.

```
 private bool Validation()
{
    if (usernameInputField.text.Length < 5 || usernameInputField.text.Length > 16) { return false; }
    if (passwordInputField.text.Length < 8 || passwordInputField.text.Length > 17) { return false; }
    return true;
}
```

Figure 3.16: Login Validation

The rest of the validation occurs on the server.

```
 private IEnumerator Upload()
{
    UnityWebRequest http =
        UnityWebRequest.Post(PlayerProfile.GetLoginServerAddress().ToString()+"/login", formFields);
    yield return http.SendWebRequest();
    if (success)
    {
        PlayerProfile.uID = username;
        PlayerProfile.token = token;
        SceneManager.LoadScene("PlayerAccountScene");
    }
}
```

Figure 3.17: HTTP Request

User name and password are added to a dictionary of strings called formFields and sent in the body of the post

request in order to be authenticated. On server's response, if successful, there is a token that will be added to the Player Profile class, and will be use it from now onward to verify the clients access to certain routes. The client is then forwarded to the next scene, PlayerAccountScene.

Register Scene

If the user does not has an account already, they can create one from the Register Scene.

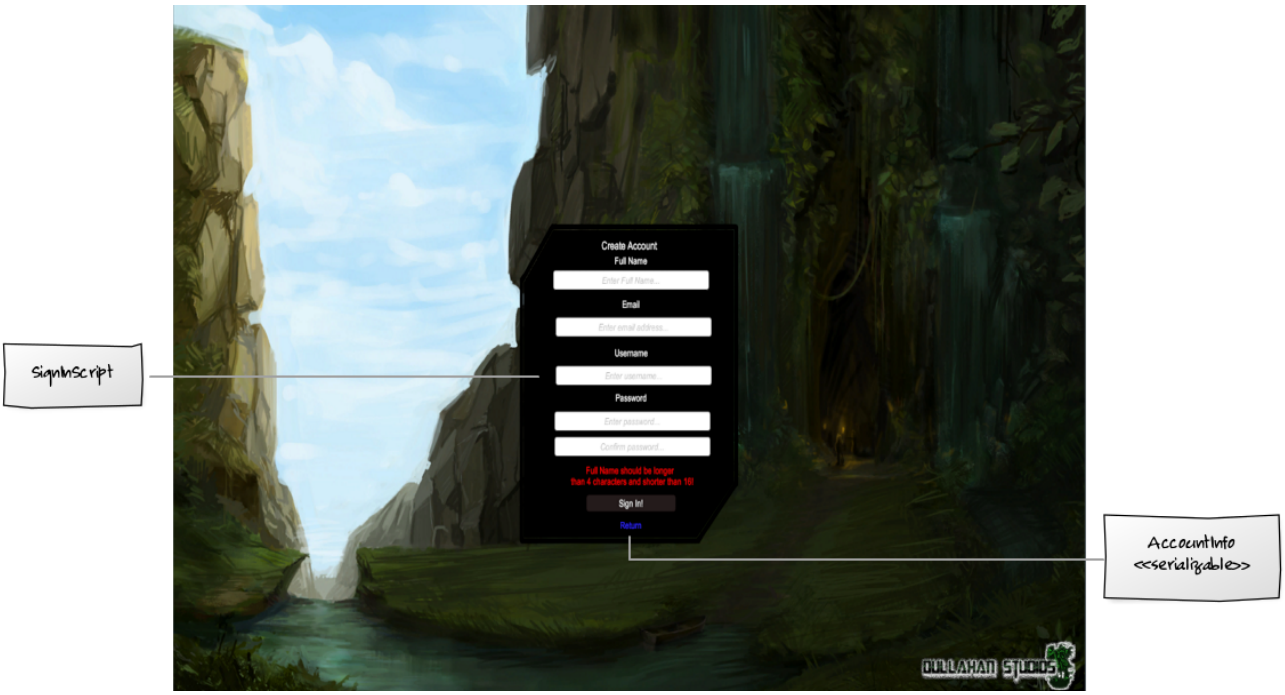


Figure 3.18: Register Scene

The register form contains 6 fields:

- **Full Name**: User's full name.
- **Email**: A valid email address. If the address is already in the database it will return an error.
- **Username**: A username. If the username is already in the database it will return an error.
- **Password** and **Confirm password**: A password to access the account.

Very much like in the Login Script, SignIn Script provides a low level validation and uses the same method for posting the date to the server.

Errors in the form will be displayed in red on the scene so they can be corrected. Upon receiving response from the server, if the the registration is successful the user will be forwarded to the login scene, otherwise it will display the error on the current scene. The only errors returned from server, other than network error, are either username already in use or email already in use.

```
private bool Validate()
{
    if (fullnameInputField.text.Length < 5 || fullnameInputField.text.Length > 16) {
        errorDisplay.text = "Full Name should be longer \n than 4 characters and shorter than 16!";
        return false; }
    if (!emailInputField.text.Contains("@") && (!emailInputField.text.EndsWith(".com") ||
        !emailInputField.text.EndsWith(".ie"))) { errorDisplay.text = "Invalid email address!";
        return false; }
    if (usernameInputField.text.Length < 5 || usernameInputField.text.Length > 12) {
        errorDisplay.text = "Username should be longer \n than 4 characters and shorter than 16!";
        return false; }
    if (passwordInputField.text.Length < 8 || passwordInputField.text.Length > 16) {
        errorDisplay.text = "Password should be longer \n than 7 characters and shorter than 17!";
        return false; }
    if (passwordInputField.text != passwordInputFieldValidation.text) { errorDisplay.text = "Password
        does not match!"; return false; }
    return true;
}
```

Figure 3.19: Sign In form Validation

Player Account Scene

Once the user has an account, and logs in successfully, the user is forwarded to his account. In their account players will have access to their characters, where they can select them to play in the game server or delete them. Players will also be able to create new character's, provided they own four characters at most at the same time.

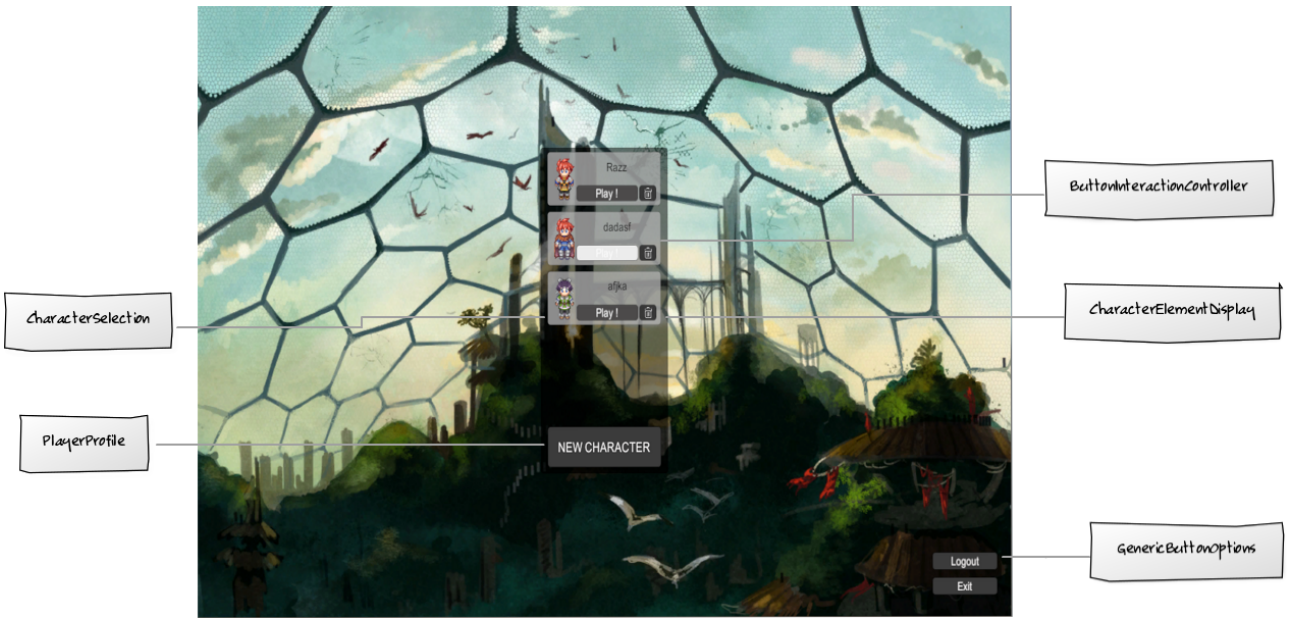


Figure 3.20: Account Scene

The scene contains two buttons on the bottom right corner implementing the Generic Button Options script that allow the user either to log out or exit the application. The Character Selection scripts queries the server to retrieve user's character list. The server returns a JSON array containing character data. Character's data

is parsed and rendered into Character Element Display elements, that are added into their list and displayed on the scene. Button Interact Controller script then, gets the id of the display element and uses it to trigger the correspondent event. If the amount of characters is lower than four, Character creation is allow, enabling the 'NEW CHARACTER' button, that on click will forward the character to the Character Creation Scene.

Character Creation Scene

The character creation scene allows the user to create a character assigning it a name, a hair style and a clothing from a set of predefined assets.

Figure 3.21: Log In Scene

The Character Creation Script manages the creation menu, determining which asset to display based on the asset iteration. On clicking on an arrow, the script will iterate over the assets list rendering the selected asset. Upon deciding a name and an appearance, the user can proceed to create a character. The character details will be submitted to the server through an HTTP POST request and stored in the database. Character's stored in the database will then be retrieved on the Account Scene, on success, users will be forwarded to the Account Scene where they can select the newly created character, or any other previous character and enter the game scene.

Game Scene

The game scene is where everything about the client comes together. The data of the player such as its ID, the character selected and the authentication token are pulled from the Player Profile Class and sent to the game

server through the network component. The client connects with the server using a TPC socket and request for authorization, once authorized the client spawns an instance of a player, along an instance for every other player in the scene (other clients).

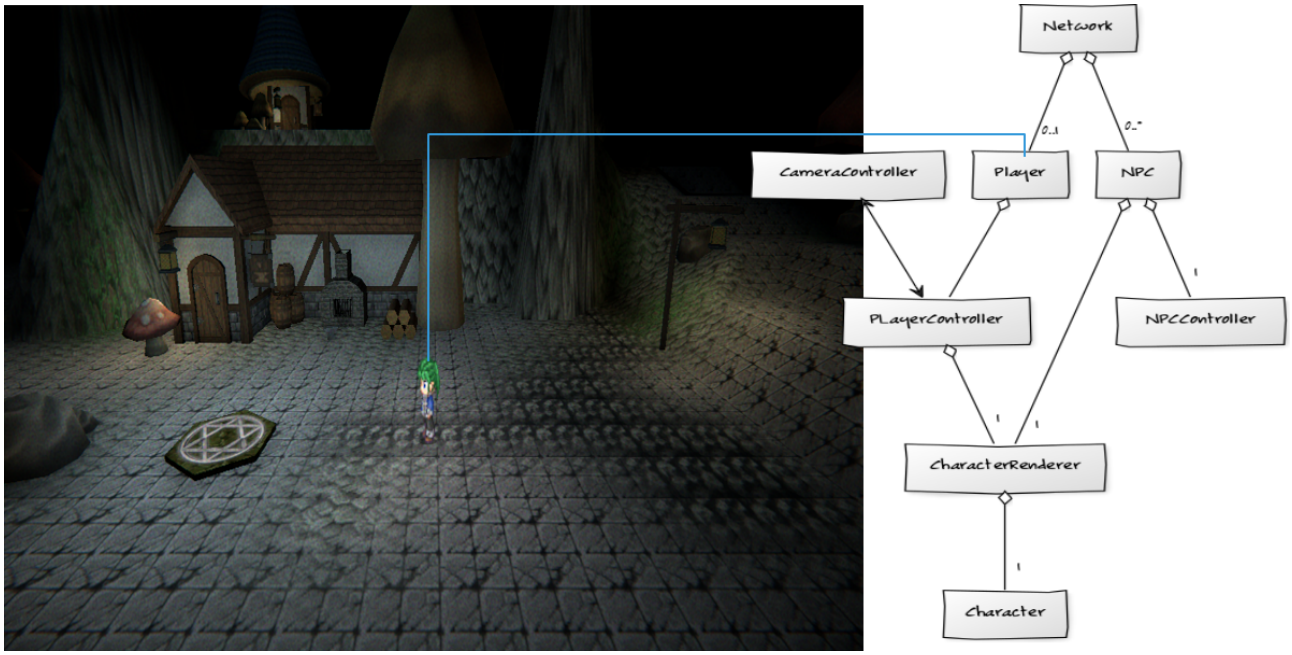


Figure 3.22: Log In Scene

The users, now a players, can control their character using either WASD pattern of movement or using a game controller. The movement and animation is handled by the PlayerController script that takes such input and passes the values to an animation controller to update the character's model.

```
private void GetInput()
{
    input.x = Input.GetAxis("Horizontal");
    input.y = Input.GetAxis("Vertical");
}
private void Move()
{
    Vector3 movement = new Vector3(0, verticalVelocity * Time.deltaTime, 0);
    if (!(Mathf.Abs(input.x) < 0.1 && Mathf.Abs(input.y) < 0.1))
    {
        movement += new Vector3(input.x, verticalVelocity, input.y );
    }
    player.Move((movement*speed) * Time.deltaTime);
}
```

Figure 3.23: Character Controller Snippet

The other clients, NPCs for the player, use the Unity component NavMeshAgent that makes use of the UnityEngine AI package, allowing agents to be sent to a valid set of coordinates in a navigable mesh. When a character moves it is broadcasted through the server to every other client, while the movement of self is handled in the client by the Player Controller script, such movement in the other clients is handled by the agent. On

receiving the binary data on the network component, the id of the character along their coordinates are used to set the agents destination to such coordinates and smooth the transition.

```
public void SendMovement(string id, float x, float y, float z) {
   ByteBuffer buffer = new ByteBuffer();
   buffer.WriteInt((int)Assets.Scripts.Enums.AllEnums.SSyncingPlayerMovement);
   buffer.WriteString(id);
   buffer.WriteFloat(x);
   buffer.WriteFloat(y);
   buffer.WriteFloat(z);
   myStream.Write(buffer.ToArray(), 0, buffer.ToArray().Length);
   myStream.Flush();
}
public void HandleSSyncingPlayerMovement(byte[] data) {
   ByteBuffer buffer = new ByteBuffer();
   buffer.WriteBytes(data);
   buffer.ReadInt();
   string playerID = buffer.ReadString();
   float x, y, z = 0;
   x = buffer.ReadFloat();
   y = buffer.ReadFloat();
   z = buffer.ReadFloat();

   npcLst[playerID].GetComponent<NavMeshAgent>().SetDestination(new Vector3(x, y, z));
   }
```

Figure 3.24: Network Controller Snippet

The client maintains a list of other players connected and their instances, and removes them from the list when they disconnect.

3.3.3 Management App

The second part of the solution proposed is the Management App. It is developed using Angular5 and its function is to manage both the game server and the community of players registered and using the game. The management app provides an interface where administrators can perform the before mention functions, among those are the following:

- Obtain information about the players.
- Ban players from entering the game.
- Pardon banned players.
- Obtain information about the players that are currently online and their characters.
- Kick players from the server.
- Change server properties such as: port number, maximum number of concurrent players.

- Restart the server.

Angular5 implements the model view controller architecture, so in order to convey its design we will start by explaining the Model in the form of angular services, proceeded by the view and controller for every route of the web application.

Services

The breakdown for the services is as follows:

```
./services
├── server-settings.service.ts
├── data.service.ts
└── authentication.service.ts
```

Authentication Service

Authentication service handles authentication in the management app. It also manages the token received and the log out function.

```
import { Injectable } from '@angular/core';
import { AppSettingsService } from './app-settings.service';
import { Http, Headers, Response } from '@angular/http';
import { Observable } from 'rxjs';
import 'rxjs/add/operator/map'

@Injectable()
export class AuthenticationService {
    public token: string;

    private apiURL = this.appSettings.getApiURL();

    constructor(private http: Http, public appSettings: AppSettingsService) {
        var currentUser = JSON.parse(localStorage.getItem('currentUser'));
        if(currentUser){
            this.token = currentUser.token;
        } else {
            this.logout;
        }
    }

    login(username: string, password: string): Observable<boolean> {
        return this.http.post(this.apiURL + 'loginAdmin', { username: username, password: password })
            .map((response: Response) => {
                let token = response.json();
                if (token.success) {
                    this.token = token.token;
                    localStorage.setItem('currentUser', JSON.stringify({ username: username, token: token.token }));
                    return true;
                } else {
                    return false;
                }
            });
    }

    logout(): void {
        this.token = null;
        localStorage.removeItem('currentUser');
    }
}
```

Figure 3.25: Authentication Service

It uses observables to wait for the promise, and process it once the response is returned form the server.

Data Service

Data Service handles all data related with the players and their characters. It is also tasked with the 'banning' and 'pardoning' of players or 'kicking' them from the game server. In order to access the required routes in the server, all request pass in an authorization token in the header.

```
import { Injectable } from '@angular/core';
import { Http, Headers, RequestOptions, Response } from '@angular/http';
import { AppSettingsService } from './app-settings.service';
import { Observable } from 'rxjs/Observable';
import { User } from '../models/User';
import { AuthenticationService } from './authentication.service';

@Injectable()
export class DataService {

  private apiURL = this.appSettings.getApiURL();
  private gameServerURL = this.appSettings.getGameServerURL();

  constructor(private http: Http, public appSettings: AppSettingsService, private authenticationService: AuthenticationService) {}

  getRegisteredUsers(): Observable < any > {···
  }
  listPlayers(): Observable < any > {···
  }

  kickPlayer(player_ID: string, char_ID: string): Observable < boolean > {···
  }

  banPlayer(username: string): Observable < boolean > {···
  }

  pardonPlayer(username: string): Observable < boolean > {···
  }
}
```

Figure 3.26: Data Service

Server Settings Service

Server Settings Service retrieves all data related to the server configuration.

```
import { Injectable } from '@angular/core';
import { AppSettingsService } from './app-settings.service';
import { AuthenticationService } from './authentication.service';
import { Http, Headers, RequestOptions, Response, RequestMethod } from '@angular/http';
import { Observable } from 'rxjs';
import { Settings } from '../models/settings';

@Injectable()
export class ServerSettingsService {

  private apiURL = this.appSettings.getApiURL();
  private gameServerURL = this.appSettings.getGameServerURL();

  constructor(private http: Http, public appSettings: AppSettingsService, private authenticationService: AuthenticationService) { }

restartServer(): Observable<boolean> {···
}
changeSettings( port : number , concurrent_players : number, restart : boolean): Observable<boolean> {···
}
getCurrentSettings(): Observable<any> {···
}
```

Figure 3.27: Server Settings Service

Routes

For every different route handled by the routing module, there is a controller and an associated view, in angular this is called component. In this case the pages are rendered inside a template contain a nav bar and a side bar that are intended as menus to access the different pages of the application, except the log in screen that is rendered in the root component of the application.

The routes can be accessed from the address bar, however the application will prevent any access if the user

```
const routes: Routes = [
  {
    path: '',
    children: [
      { path: '', redirectTo: '/dash-menu', pathMatch: 'full' },
      { path: 'dash-menu', component: InAppRootComponent, children:[
        { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
        { path: 'dashboard', component: DashboardComponent },
        { path: 'users', component: UsersComponent },
        { path: 'settings', component: ServerSettingsComponent },
        { path: 'players', component: PlayersComponent },
      ] }
    ], canActivate: [AuthGuard]
  },
  { path: 'login', component: LoginComponent },
  { path: '**', redirectTo: '' }];
```

Figure 3.28: App Routing Module

tries to access any of them without being authenticated due to the AuthGuard component that the 'children' routes can activate.

```
@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private router: Router) { }
  canActivate() {
    if (localStorage.getItem('currentUser')) {
        // logged in so return true
        return true;
    }
    // not logged in so redirect to login page
    this.router.navigate(['/login']);
    return false;
}
```

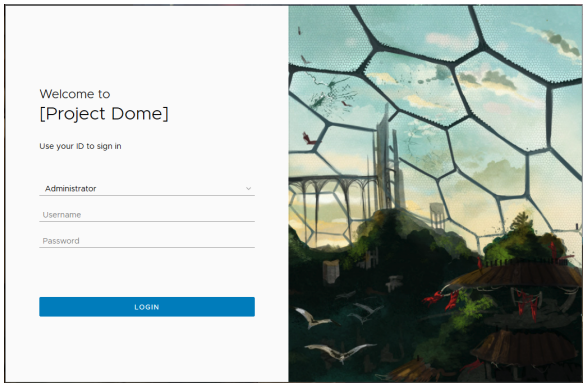
Figure 3.29: AuthGuard Component

Summarizing, if the user is not logged in and tries to access any route other than '/login' it will be automatically redirected to the login page, however if it has been logged in before in that browser and has not logged out, its session will still be active and it will be allowed into the application. Navigating then to any of the allowed routes will be granted, by accessing any route that is not defined the user will be redirected to the root route '/dash-menu'.

Management App log in

Starting from the first view the user encounter, the login page is the first page to come up if the user is not logged in, either because it is the first visit to the page or because it previously logged out. The user will require a username and a password to access the application. The type of user required is an admin user, which means that not every user that has registered to the application, typically from the client will have access to

the Management App.



(a) Log in view

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { User } from '../models/User';
import { AuthenticationService } from '../services/authentication.service'

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {

  user: any = {};
  loading = false;
  error = '';

  constructor(private router: Router,
    private authenticationService: AuthenticationService) { }

  /** Log out on navigation this page. */
  ngOnInit() {
    this.authenticationService.logout();
  }

  /** Login call Authentication service to verify whether the provided
   * user data matches the one in the database.
   */
  login(): void {
    this.loading = true;
    this.authenticationService.login(this.user.username, this.user.password)
        .subscribe(result => {
            if (result === true) {
                // On login successful navigate to default route.
                this.router.navigate(['/']);
            } else {
                // On login failed display error message.
                this.error = 'Username or password is incorrect';
                this.loading = false;
            }
        });
  }
}
```

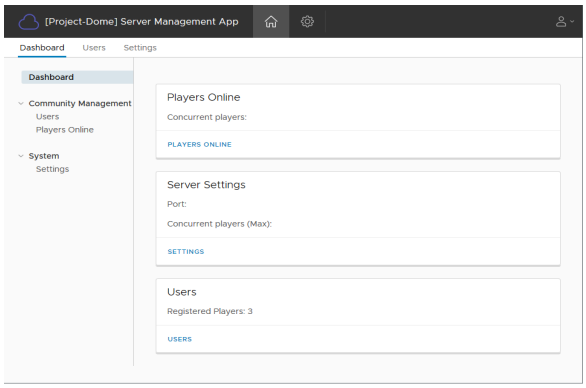
(b) Log in Controller

Figure 3.30: Log In View/Controller

The controller will call the authentication service previously discussed passing in the input from the view that user entered for authentication. The service will return a response either allowing or rejecting the request. If the username and password do not match the user will see an error message at the bottom of the form. There is a drop down box with 'Administrator', it is a feature to be implemented in a future iteration to allow non-administrator users to manage their characters or services associated to them.

Dashboard

The dashboard page allow the user to briefly overview some of the details of the server.



(a) Dashboard view

```
import { Component, OnInit } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { ServerSettingsService } from '../services/server-settings.service';
import { DataService } from '../services/data.service';
import { OnlinePlayer } from '../models/OnlinePlayer';
import { User } from '../models/User';
import { Settings } from '../models/settings';

@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
  styleUrls: ['./dashboard.component.css']
})
export class DashboardComponent implements OnInit {

  /**  */
  public playersOnline = {};
  public users = {};
  public settings = new Settings();

  /** Query Serber for Relevant information */
  constructor(private serverSettingsService : ServerSettingsService, private dataService : DataService) {
    this.dataService.listPlayers().subscribe(players => this.playersOnline = players);
    this.serverSettingsService.getCurrentSettings().subscribe(settings => this.settings = settings);
    this.dataService.getRegisteredUsers().subscribe(users => this.users = users);
  }

  ngOnInit() {
  }
}
```

(b) Dashboard Controller

Figure 3.31: Dashboard View/Controller

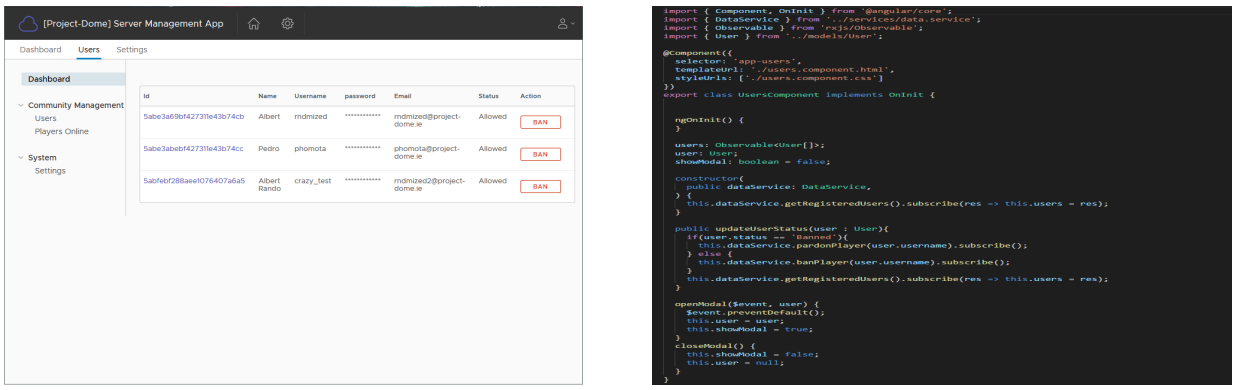
On loading the page the controller calls the services to present the user with a minimal view of server's data.

- Number of Online Players
- Server Port
- Maximum number of concurrent users allowed

- Number of total registered users.

Users

This page display the total number of registered users on the database that are not administrators (therefore players). The controllers calls the data service to retrieve such data and prints it to the table, generating a row for each user. The last field of the table, 'Action', allows the manager to either ban or, in case the player has been banned already, to pardon it changing the text and the action of the button accordingly.



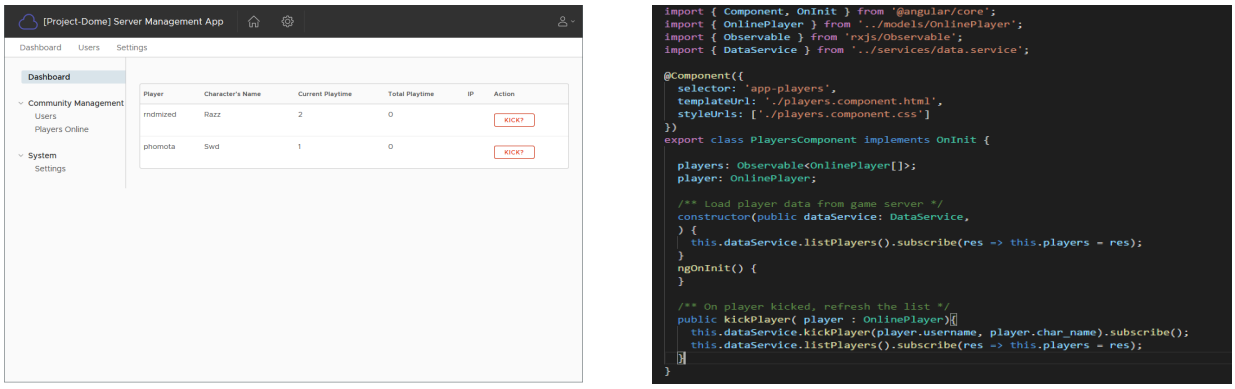
(a) Users view (b) Users Controller

Figure 3.32: Users View/Controller

On clicking in the id a modal will open displaying player's data in JSON format.

Online Players

Online players provides the manager with the list of players that are currently connected to the server.



(a) Players Online view (b) Players Online Controller

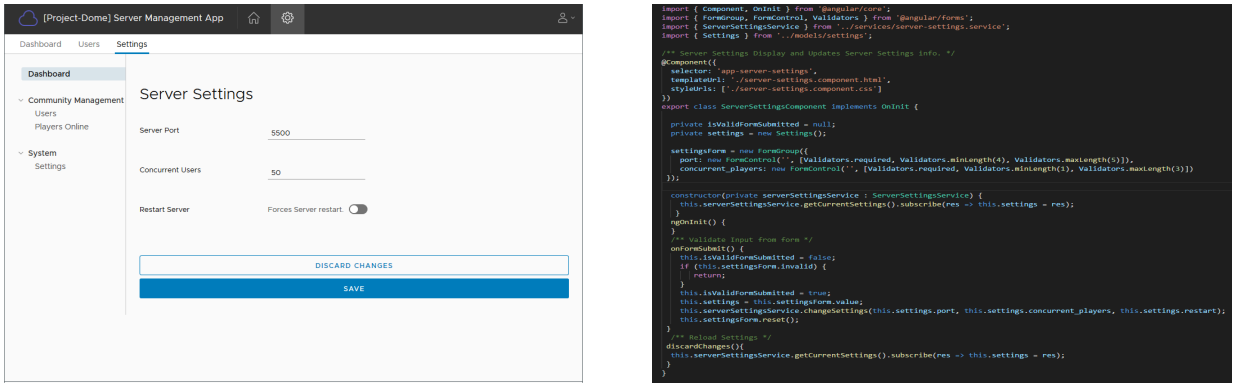
Figure 3.33: Players Online View/Controller

Among the information provided there is the character's name, the player's name, their playtime and IP address, and the option to kick them out of the server. The controller calls the data service in order to retrieve such

information and display it to the administrator.

Server Settings

Server Settings allow administrators to change some of the settings on the server such as the port it is running on and the maximum number of concurrent players allowed. It also allows the administrator to force a server restart. Changes on server do not apply until restart.



(a) Server Settings view

(b) Server Settings Controller

Figure 3.34: Server Settings View/Controller

The controller calls the Server Settings Service in order to server the configuration options to the server as well as to retrieve the current settings in order to display them on the view. It also provides with some validation in order to avoid mistakes that might affect the game service. It is the only view, apart from login, that allows user input.

3.3.4 ByteBufferDLL

3.3.5 Game Server (TCP)

3.3.6 Game Sever (HTTP)

Chapter 4

Conclusion

4.1 Summary of Thesis Achievements

Summary.

4.2 Applications

Applications.

4.3 Future Work

Future Work.

Bibliography

[1] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, 1999.

[2] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[3] M. Jones, J. Bradley, and N. Sakimura. JSON Web Token (JWT). Internet-Draft draft-ietf-oauth-json-web-token-18, Internet Engineering Task Force. Work in Progress.