

# Sync-Along Final Report

Student ID	Name	Email
A0187457M	Chow Jia Ying	e0323041@u.nus.edu
A0184573W	Dinh Nho Bao	e0313562@u.nus.edu
A0183648R	Hoe Chan Wei Daniel	e0310443@u.nus.edu
A0136152N	Yeo Dong Han	e0321479@u.nus.edu

Deployed Application Link: <http://13.212.140.143:3000/>

Code Repository Link:

<https://github.com/CS3219-SE-Principles-and-Patterns/cs3219-project-ay2122-2122-s1-g6>

# Table of Contents

<b>1. Individual Contributions</b>	<b>5</b>
<b>2. Project Introduction</b>	<b>6</b>
<b>3. Project Scope</b>	<b>7</b>
3.1. Features	7
3.1.1. User Management	7
3.1.2. Room Management	7
3.1.3. Room Chat Management	7
3.1.4. Music Player Management	7
3.2. Requirements	8
3.2.1. Functional Requirements	8
3.2.2. Non-Functional Requirements	11
<b>4. Developer Documentation</b>	<b>13</b>
4.1. Project Architecture	13
4.1.1. Overview	13
4.1.2. Frontend	14
4.1.3. Backend	16
4.2. Feature Implementations	18
4.2.1. Registration	18
4.2.2. Login	20
4.2.3. Room Creation and Joining	22
4.2.3.1. Room Creation	22
4.2.3.2. Room Joining	23
4.2.4. Player Synchronisation	25
4.2.5. Playlist Synchronisation	26
<b>5. Design Decisions and Technologies</b>	<b>27</b>
5.1. General	27
5.1.1. TypeScript	27
To make it easy for developers to context switch between developing and deploying the frontend and backend as needed, both frontend and backend application code is written in Typescript.	27
5.1.2. HTTP requests and WebSockets	27
5.1.3. Validation on both ends	28
5.2. Frontend	29
5.2.1. Technologies	29
5.2.1.1. React	29
5.2.1.2. Redux and Redux Toolkit	29
5.2.2. Client-sided Music Streaming	29
5.3. Backend	30
5.3.1. Technologies	30
5.3.1.1. Express	30

5.3.1.2. Socket.IO	30
5.3.1.3. Redis Cloud	30
5.3.1.4. MongoDB and Mongoose	30
5.3.1.5. ESSerialiser	31
5.3.2. Microservices to Monolithic architecture	31
5.3.3. Backend Design Patterns	32
5.3.3.1. Singleton Pattern	32
5.3.3.2. Facade	32
<b>6. Testing</b>	<b>33</b>
6.1. Decisions	33
6.2. Stress Testing	33
6.2.1. Categorise tests	33
6.2.2. Analysis	33
6.3. Test Cases	34
<b>7. Deployment</b>	<b>35</b>
7.1. Automated code building	35
7.2. Deployment to Production	35
7.2.1. Helm chart deployment	35
7.2.1.1. Manifests	35
7.2.2. Benefits of Helm Charts	36
7.2.2.1. Automated deployment process	36
7.2.2.2. Ease of changing configuration	36
7.3. Deployment Challenges	37
7.3.1. Issues with Kubernetes	37
7.3.2. Redis and MongoDB from cluster to cloud	38
7.3.3. Application code from GKE to AWS EC2	39
7.3.3.1. Why AWS EC2	39
7.3.3.2. Our approach	39
7.3.3.3. pm2	39
7.3.4. Final application deployment	40
7.3.4.1. Scope Limitation	40
<b>8. Suggestions for Improvements</b>	<b>41</b>
8.1. Incorporation of CI/CD to application	41
8.2. Improve application security	41
8.2.1 Add more authentication to sockets and API requests	41
8.2.2 Add more checks to user management services	42
8.3. Improved User Experience	42
<b>9. Reflections and Learning Points</b>	<b>43</b>
9.1. Time Management	43
9.2. Early Testing	43
9.3. Improved documentation process	43
<b>10. Conclusion</b>	<b>43</b>

<b>11. Reference</b>	<b>45</b>
<b>11.1. API Documentation</b>	<b>45</b>
11.1.1. User Management Service	45
11.1.1.1. Registering	45
11.1.1.2. Login	46
11.1.1.3. Logout	47
11.1.2. Room Management Service	48
11.1.2.1. Client-to-Server events	48
11.1.2.1.1. Creating a new room	48
11.1.2.1.2. Joining a room	48
11.1.2.1.3. Leaving a room	49
11.1.2.2. Server-to-Client events	50
11.1.2.2.1. Updating the room status	50
11.1.3. Room Chat Service	50
11.1.3.1. Client-to-Server events	50
11.1.3.1.1. Sending Chat Messages to a room	50
11.1.3.2. Server-to-Client events	50
11.1.3.2.1. Broadcasting Chat Messages to the room	50
11.1.4. Music Playlist Service	51
11.1.4.1. Client-to-Server events	51
11.1.4.1.1. Adding a song to the playlist	51
11.1.4.1.2. Removing a song to the playlist	51
11.1.4.1.3. Selecting an active song in the playlist	51
11.1.4.1.4. Moving to the next song in the playlist	51
11.1.4.1.5. Moving to the previous song in the playlist	51
11.1.4.1.6. Playing a song in the player	51
11.1.4.1.7. Pausing a song in the player	51
11.1.4.1.8. Scrubbing the time in the player	52
11.1.4.1.9. Indicating the song is complete in the player	52
11.1.4.2. Server-to-Client events	52
11.1.4.2.1. Updating the player status	52
11.1.4.2.2. Updating the playlist status	52

# 1. Individual Contributions

Team member	Technical Contributions	Non-technical Contributions
Chow Jia Ying	<ul style="list-style-type: none"> <li>- Set up structure of frontend app</li> <li>- Implemented room chat service in the frontend</li> <li>- Researched and integrated Redux Toolkit in the app</li> <li>- Researched and integrated the use of Socket.IO in the frontend</li> <li>- Setup backend services and architecture</li> <li>- Implemented backend via TypeScript</li> </ul>	<ul style="list-style-type: none"> <li>- Final Report documentation</li> </ul>
Dinh Nho Bao	<ul style="list-style-type: none"> <li>- Implemented user management service in the backend</li> <li>- Implemented room chat service in the backend</li> <li>- Deployment of application</li> </ul>	<ul style="list-style-type: none"> <li>- Final report documentation</li> <li>- Software testing</li> </ul>
Hoe Chan Wei Daniel	<ul style="list-style-type: none"> <li>- Frontend development of player, playlist features</li> <li>- Implemented login and registration in the frontend</li> <li>- Styling and general bug fixing of frontend components</li> <li>- Integrated the music player in the frontend</li> <li>- Initial implementation of individual backend services</li> </ul>	<ul style="list-style-type: none"> <li>- UI/UX mockups using Figma</li> <li>- Final report documentation</li> </ul>
Yeo Dong Han	<ul style="list-style-type: none"> <li>- Implemented music player / playlist service</li> </ul>	<ul style="list-style-type: none"> <li>- Final report documentation</li> <li>- Generation of test cases</li> </ul>

## 2. Project Introduction

As more and more people turn to the internet for entertainment and work, there is a growing demand for applications that are able to offer such services online.

It is common for people to listen to music together in a group: at a bar, at a concert, at social gatherings, and at home with families and friends. However, we realised that there aren't many applications out there that allow users to listen to music together online, something that is easily done in person. Therefore, we came up with the idea of **Sync-Along**!

**Sync-Along** is a web application that allows users to listen to music together. Users will be able to create a room and invite their families and friends to join via room codes. Inside the room, they can select and play music from Youtube links and listen to their customised playlists together. The users can also chat with each other in the room. The application also allows users to pause, skip or adjust music playback, and these will be synchronised across all other users in the room.

## 3. Project Scope

This section explains the scope of our project, the main features of the app, and a breakdown of these features into functional and non-functional requirements.

### 3.1. Features

Sync-Along provides many features to make up a music player application that can synchronise music playback across multiple users. This section briefly covers the main features of the application.

#### 3.1.1. User Management

- User Management allows users to create accounts and use these accounts to log in to the application.
- In the room, chat messages will also display the sender's username.

#### 3.1.2. Room Management

- Room Management allows users to create rooms for other users to join by room code.
- Users in the same room will share one synchronised music player, one song playlist, and one room chat.

#### 3.1.3. Room Chat Management

- Each room has a room chat.
- Users can send messages and receive messages from all other users in the room.

#### 3.1.4. Music Player Management

- Each room has a shared music player.
- Users can add or remove songs from Youtube links in the playlist in the room.
- Users can select a song in the playlist to be played.
- Users in the room can play/pause music, jump to the next/previous song, or adjust the playback of the current song in the music player.
- All of these actions will be synchronised across all other users in the room. For example, if one user pauses the music, music will be paused across all users in the room.

## 3.2. Requirements

Based on the application description, we identify and gather our requirements. Requirements are categorised into FRs and NFRs.

We also chose to prioritise tasks using the **Must-Should-Could-Wish** scheme to identify what requirements are the most important and what requirements are the least important in terms of adding values to our end users.

We implemented a majority of functional and non-functional requirements that were listed. However, these features were not implemented:

- User Management (F1.4 and F1.5) - while these features are good to have, they do not directly solve the problem of helping users listen to music together. The username is sufficient so that the user can communicate with other users in the chat room.
- User Playlist Management features. These are **Wish** features and are not crucial to the app since the Music Player service already allows users to listen from a synchronised music player. In the interest of time, we skip these features.

### 3.2.1. Functional Requirements

The following table lists the functional requirements in the Requirements and Design Report and the implementation status for each requirement.

1. User Management			
S/N	Requirement	Priority	Implementation Status
F1.1	The application should allow users to create an account with a username and a password.	Must	Implemented
F1.2	The application should allow users to sign into their account with the created username and password in <i>F1.1</i> .	Must	Implemented
F1.3	The application should allow users to log out of their account.	Must	Implemented
F1.4	The application should allow users to associate their account with an email address.	Should	Not Implemented
F1.5	The application should allow users to change or reset their password using the email address associated with the account.	Should	Not Implemented
2. User Playlist Management (CRUD)			
S/N	Requirement	Priority	Implementation Status
F2.1	The application should allow users to create their	Wish	Not Implemented



	own personal playlists with a custom name.		
F2.2	The application should allow users to edit the name of the playlist.	Wish	Not Implemented
F2.3	The application should allow users to add YouTube links to an existing playlist.	Wish	Not Implemented
F2.4	The application should allow users to delete YouTube links for an existing playlist.	Wish	Not Implemented
F2.5	The application should allow users to delete their created playlists in <i>F2.1</i> .	Wish	Not Implemented
<b>3. Room Management</b>			
S/N	Requirement	Priority	Implementation Status
F3.1	The application should allow users to create a room with a randomly generated room code.	Must	Implemented
F3.2	The application should allow users to join a room using a valid room code.	Must	Implemented
F3.3	The application should allow the room owner (the user who created the room in <i>F3.1</i> ) to remove other users from the room.	Should	Implemented
F3.4	The application should allow that when the room owner leaves the room, all the remaining users will be automatically removed from the room and the room will be deleted from the application.	Must	Implemented
F3.5	The application should allow users (except the room owner) to leave the room that they joined	Must	Implemented
<b>4. Room Chat Management</b>			
S/N	Requirement	Priority	Implementation Status
F4.1	The application should allow users to send chat messages to the room chat.	Should	Implemented
F4.2	The application should allow users to view all chat messages in the room chat.	Should	Implemented
<b>5. Music Player Management</b>			
S/N	Requirement	Priority	Implementation Status
F5.1	The application should allow users in the room to share a common playlist and a common music player.	Must	Implemented

F5.2	The application should allow users to add new songs to the room playlist by providing YouTube links.	Must	Implemented
F5.3	The application should allow users to import their personal playlists created in <a href="#">F2.1</a> to the room playlist.	Wish	Not Implemented
F5.4	The application should allow users to start, stop, pause and set the progress of the music player.	Must	Implemented
F5.5	The application should synchronise the currently playing song to all users in the room.	Must	Implemented
F5.6	The application should synchronise the start, stop, pause, or set progress action (in <i>F5.4</i> ) from one user to all other users in the room.	Must	Implemented
F5.7	The application should automatically switch to the song at the top of the playlist once the playlist has ended.	Must	Implemented
F5.8	The application should allow users to change the ordering of the songs in the room playlist.	Should	Not Implemented
F5.9	The application should allow users to shuffle the songs in the room playlist.	Could	Not Implemented

### 3.2.2. Non-Functional Requirements

To make sure that the application is responsive and robust, we identified and revised with a set of non-functional requirements. These are the main requirements that we prioritise for our application:

1. **Performance:** Performance is critical as we need to ensure that the music player is synchronised across all other users in the room. The room chat also needs to be responsive enough to facilitate frequent chat between users.
2. **Scalability:** We want to make sure that the application can work well with many users using the application at the same time.
3. **Data Constraints:** Data constraints help us to limit the scope of our product features so that they can be implemented in a simpler way.
4. **General System Requirements:** We want our application to be fully functional on Chrome, the most popular and most supported browser worldwide. To facilitate easy testing and fast deployment of the application, we won't guarantee that the application is fully functional on browsers other than Google Chrome.

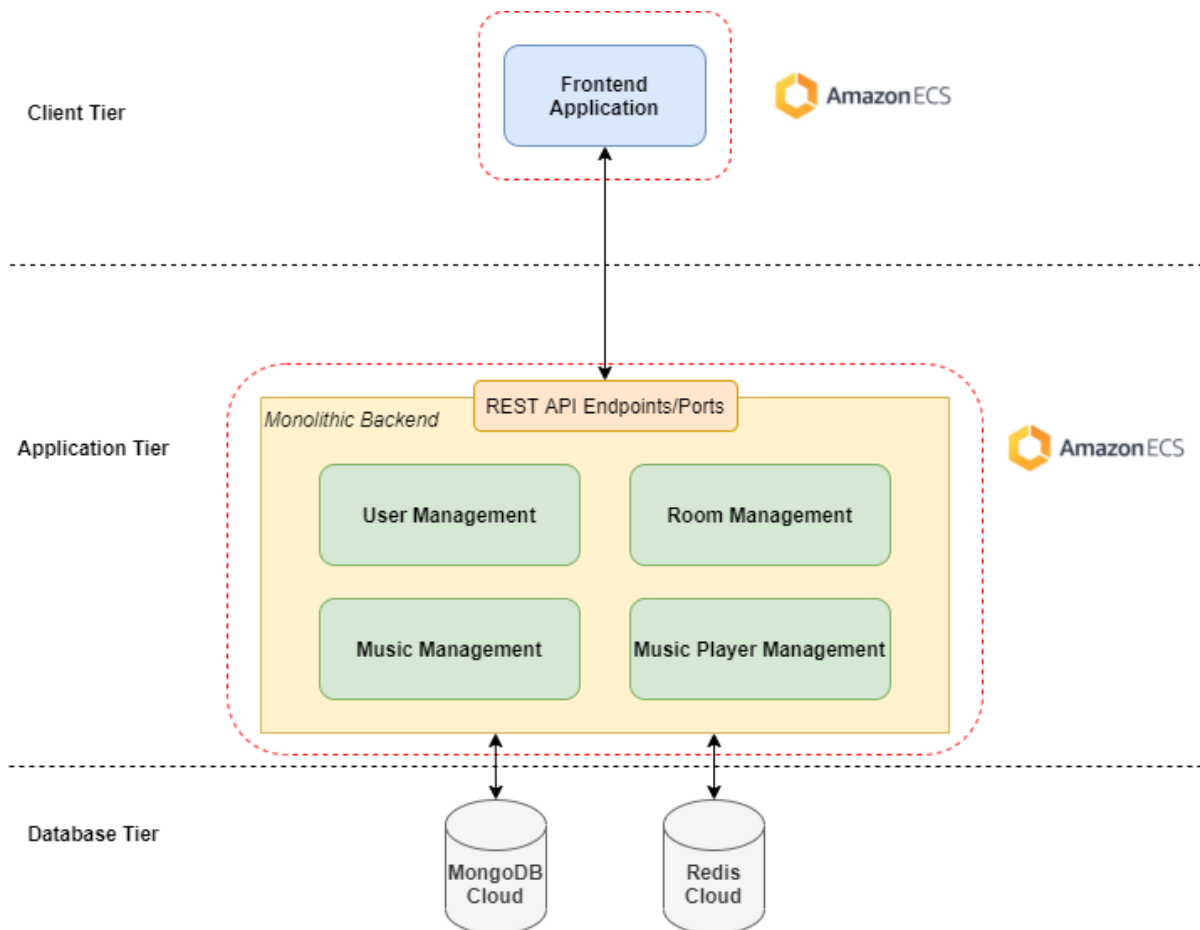
Scalability			
S/N	Requirement	Priority	Implementation Status
N1.1	The application should be able to store login details (username, hashed password in <a href="#">F1.1</a> for up to <u>1000 users</u> .	Must	Implemented
N1.2	The application should be able to support up to <u>800 concurrent users</u> .	Must	Implemented
N1.3	The application should be able to support up to <u>200 concurrent rooms</u> .	Should	Implemented
N1.4	Each room should be able to support up to <u>10 concurrent users</u> .	Must	Implemented
N1.5	Each room's playlist should be able to support up to a maximum of <u>200 songs</u> .	Must	Implemented
Performance			
S/N	Requirement	Priority	Implementation Status
N2.1	The application should allow songs imported to the room playlist (in <i>N3.1</i> ) to be available for playing in the Music Player <u>within 5 seconds</u> after the user has provided all the links.	Must	Implemented
N2.2	The application should allow Music Player commands such as start, stop, pause and set the progress of the	Must	Implemented

	music player (in <a href="#">F5.4</a> ) to be received by all users in the room in <u>less than 1 second</u> and synchronisation is maintained (in F4.5)		
<b>Data Constraints</b>			
S/N	Requirement	Priority	Implementation Status
N3.1	The application should allow imports of music from YouTube when users provide YouTube links for personal playlists (in <a href="#">F2.3</a> ) and room playlists (in <a href="#">F3.4</a> ).	Must	Implemented for F3.4, not implemented for F2.3
N3.2	The application should be able to take usernames of minimum length 4 and maximum length 16 for creating an account (in <a href="#">F1.1</a> ).	Should	Implemented
N3.3	The application should take in passwords of length no less than 6.	Should	Implemented
N3.4	The room code (in <a href="#">F3.1</a> ) should be in uppercase English letters and/or numerics of length 5.	Should	Implemented
<b>General System Requirements</b>			
S/N	Requirement	Priority	Implementation Status
N4.1	The application should be fully functional on Chrome Browser Version 93.0 on Windows 10.	Must	Implemented

## 4. Developer Documentation

### 4.1. Project Architecture

#### 4.1.1. Overview



*Fig. 4.1.1A Client-server architecture of Sync-Along.*

Sync-Along implements a 3-tier client-server architecture.

The client in React focuses on rendering the user interface as well as session management. It can communicate with the NodeJS/Express server via API requests and sockets. Sockets are used for

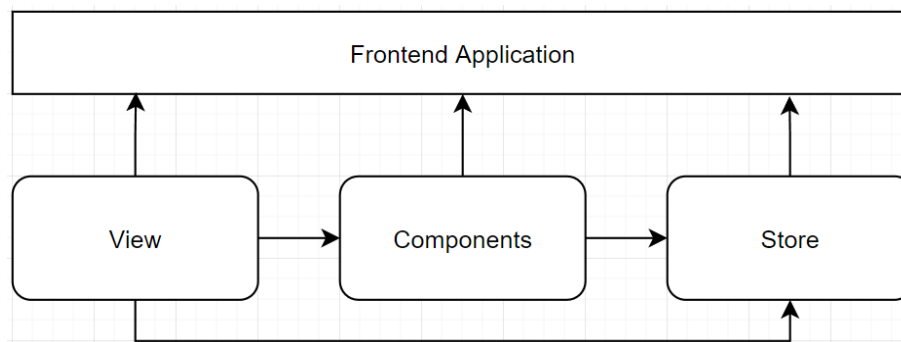
For the client tier, we decided to use a web application for the front-end that users use to interact and communicate with the application server through API calls and Socket connection (for live updating).

For the application tier, we have implemented HTTP REST API endpoints for the clients to communicate and interact with the server and socket adapters endpoints for clients to establish socket connections for live updates.

Finally, we have the database tier that provides data persistence for the application. This tier comprises a MongoDB database which stores persistent data such as user account information, and a Redis session store which keeps session data such as room and playlist information.

#### 4.1.2. Frontend

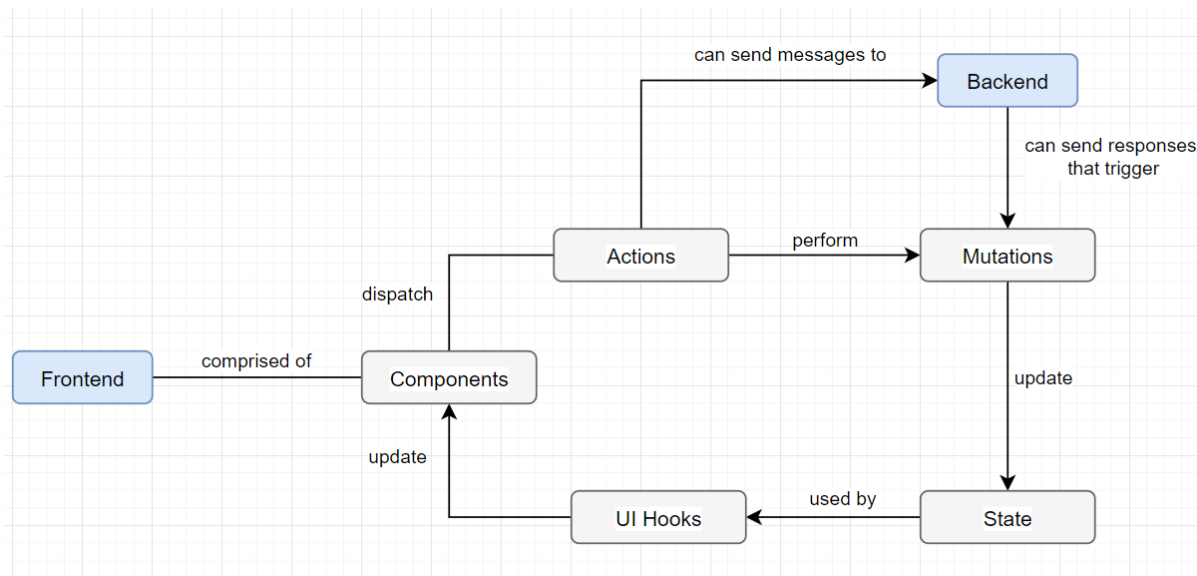
The frontend is responsible for handling the user interface and interactions. The following figure shows how the frontend is structured:



*Fig. 4.1.2A The frontend structure.*

- Frontend Application - this represents the entire application, which consists of View, Components and Store.
- View - this is responsible for displaying the relevant pages of the application (e.g. Login, Dashboard, Room)
- Components - the View comprised different Components that handle the interactions made by the user in the application.
- Store - the store handles the general state of the application, and is used by both Components and View for updating the application state

The following figure depicts the frontend architecture diagram and how data and rendering generally flow through the application:

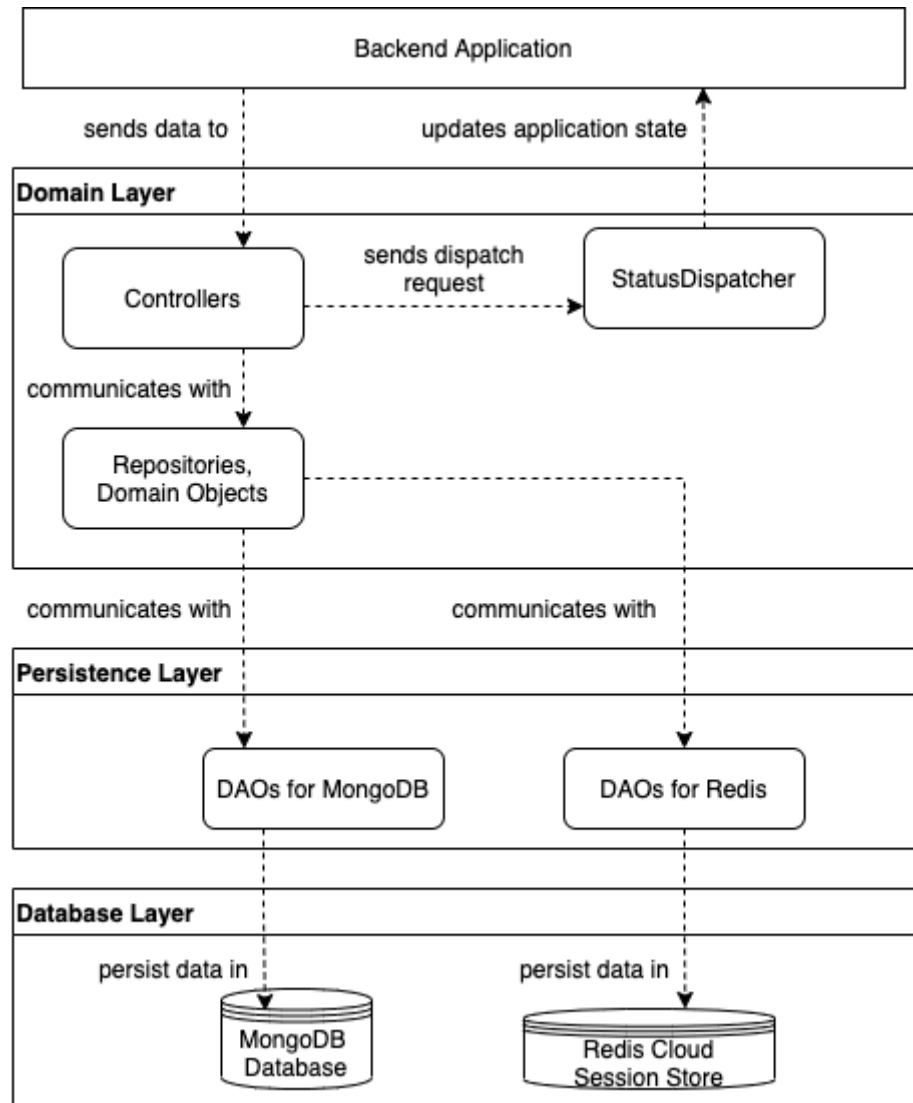


*Fig. 4.1.2A The frontend architecture diagram.*

- **Components** - these are React components that make up the application. They map user interactions to action dispatches, and also provide the look and feel of the application.
- **Actions** - actions are dispatched by components (on user interactions). Actions can either perform a mutation on the client or send messages (via WebSockets or HTTP) to the backend.
  - Actions that perform mutations directly do so only on the client itself.
  - Actions can also send messages via WebSockets (Socket.IO, for live update of information) or HTTP requests (login and registration) to the backend. Responses from the backend then trigger mutations to the state.
- **Mutations** - mutations simply perform a mutation of the state, causing it to transit from one state to another.
- **State** - the state of the application and each of the various components. This is handled by Redux and the Redux Toolkit in the form of various “stores” for different parts of the frontend
- **UI Hooks** - UI hooks depend on state transitions to provide updates to Components. This provides the responsiveness of the application to user input and actions.

### 4.1.3. Backend

The backend is responsible for handling the server-side application logic. This includes the login and registration system, room management, room chat and the room playlist and player. It also communicates with the database layer.



*Fig. 4.1.3A The backend architecture diagram.*

The backend application is a monolith, comprising three layers: Domain Layer, Persistence Layer and Database Layer. Data flows from the domain layer, to the persistence layer and to the database layer

- Domain Layer:
  - From app to domain layer: Each service (eg Room Management, Player Service) has its own controller layers. When the frontend makes a request to the backend, the request is routed to the relevant controller, which invokes the relevant repositories. The repositories then manipulate domain objects and persist data in the persistence layer.
  - From domain Layer to app: after receiving updating state in the database layers, the controllers sends a dispatch request to the status dispatcher,



which is a component that sends updates of the application state to the frontend.

- Persistence Layer: The persistence layer consists of the repositories, which uses one or more data access object layers (DAOs). Each DAO corresponds to a domain object and abstracts the persistence of that domain object to the database layer.
- Database Layer: The database layer consists of a MongoDB database and a Redis session store.

This architecture allows for limited coupling and dependencies between the business domains which allows for ease of scaling the app by adding more features in the future. With this backend architecture in place, our application follows the Single Responsibility Principle, where every module is built for a specific functionality. Thus it would be easy to maintain or add new packages for new business domains in the future.

## 4.2. Feature Implementations

### 4.2.1. Registration

Similar to most modern applications, Sync-Along contains a registration process such that each user is assigned a unique identity. The user needs to register before logging in to the application.

#### Registration

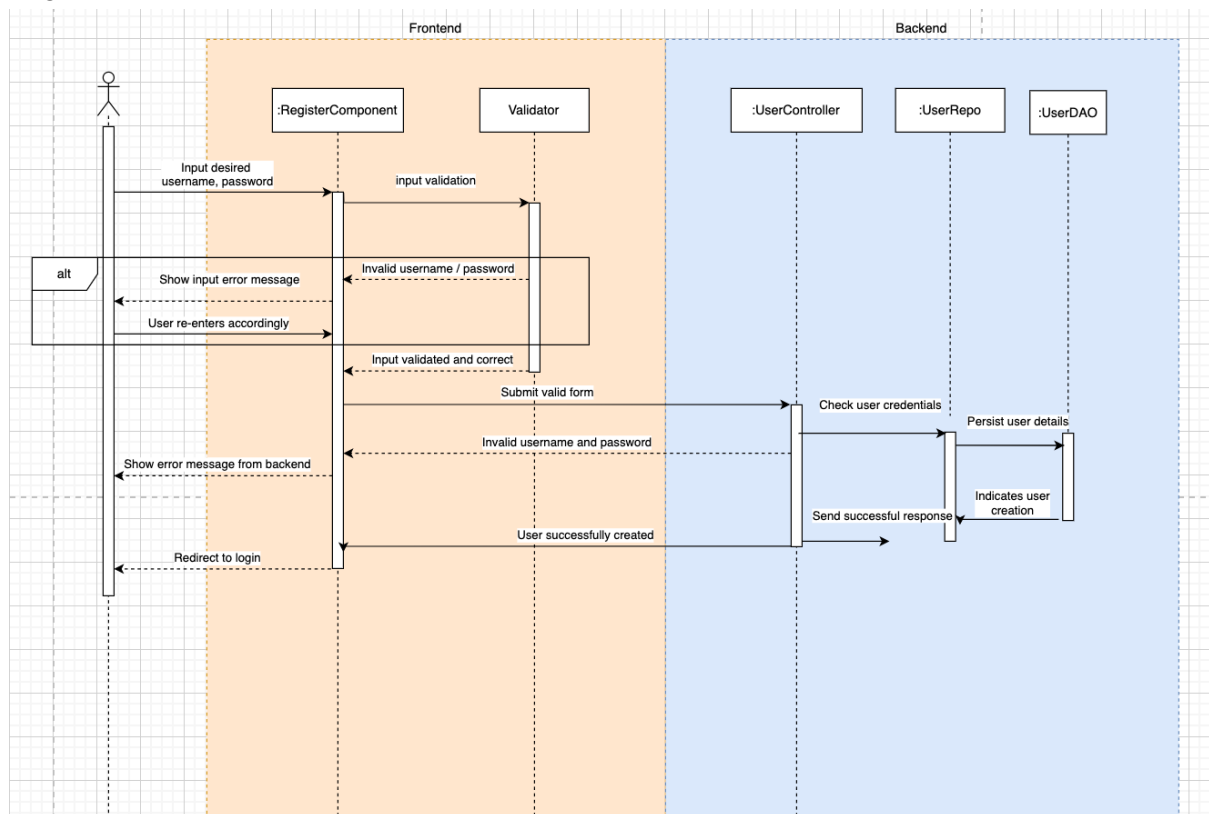


Fig. 4.2.1A General control flow of registration.

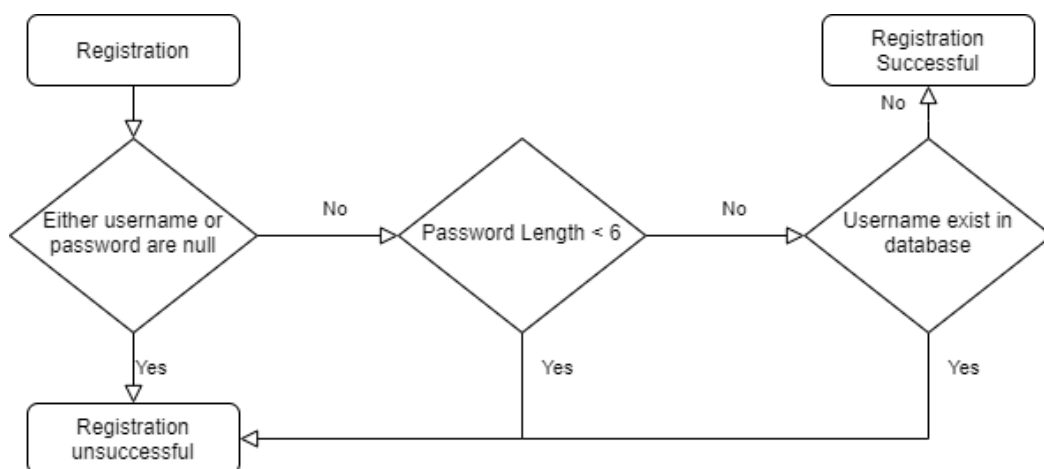


Fig. 4.2.1A Backend registration flowchart.

Registration is done inside the RegisterComponent, that requests for a username and password to be entered.

As seen on [Fig. 4.2.1A](#), validation is done twice: by the frontend (for trivial things such as empty input), and by the backend (e.g. if the username has already been taken).

- Validation on the frontend is handled by a Validator utility class, that determines whether the input is valid and also indicates any error messages if it is not valid.
- Validation logic on the backend is governed by the NFRs we have determined above, specifically [N3.2](#) and N3.3, which indicate the requirements placed on username and passwords.

Within the backend application, the HTTP request is routed to the UserController component. The UserController validates that the request has sent in valid data before adding the user to the database. This is done by sending a request to the UserRepo, which then interacts with the UserModel component.

Once the username and password have been checked to be valid, a new User is created and stored in the database. The backend informs the frontend of a successful registration, and RegisterComponent redirects the user to the login page for further action.

## 4.2.2. Login

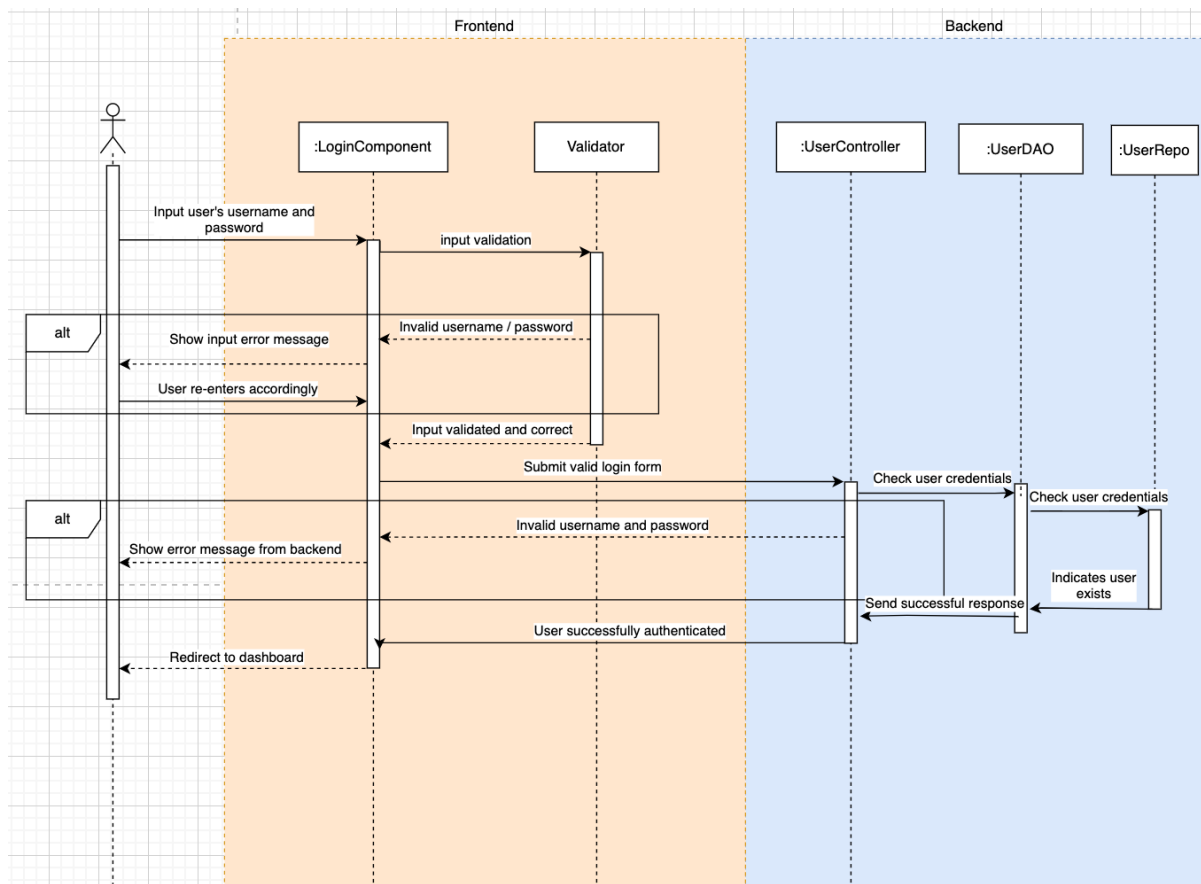


Fig. 4.2.2A General control flow of login.

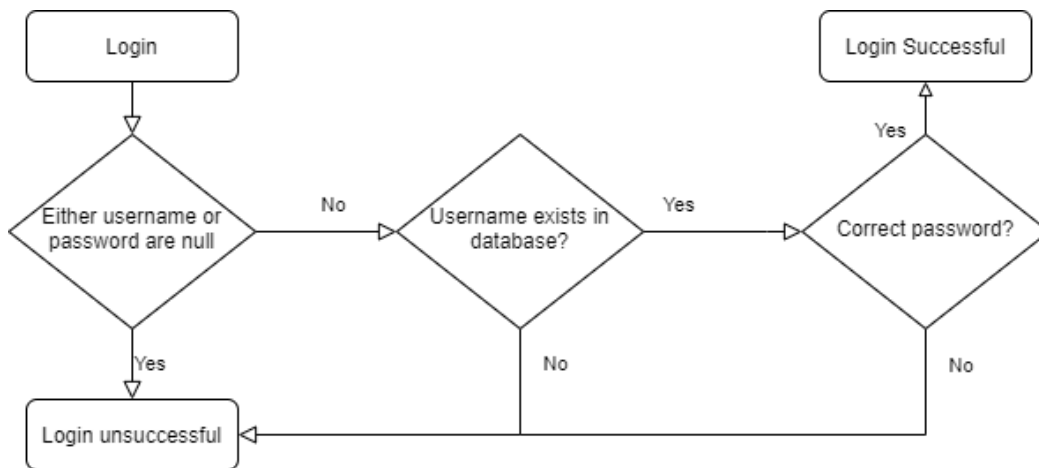


Fig. 4.2.2B Backend login flowchart.

Similar to [registration](#), the user needs to enter a username and password.

As seen from [Fig. 4.2.2A](#), validation is done twice as well here, on the frontend (for trivial empty strings) and the backend (e.g. whether the user actually exists in the database). In the backend, the request is passed from the UserController to the UserRepo. Then, within the UserRepo, the username and hashed password is checked with an entry in the database.

Once the username and password have been determined to exist, the backend informs the frontend about the approved authentication process. The LoginComponent then directs the user to the dashboard where they can create and join rooms.

### 4.2.3. Room Creation and Joining

Sync-Along provides features for users to create rooms for others to join. This feature intends to provide a common area for users to chat and listen to music, allowing them to customise what music they want to listen to.

#### 4.2.3.1. Room Creation

The room creation process starts from the user, and is depicted in this control flow diagram:

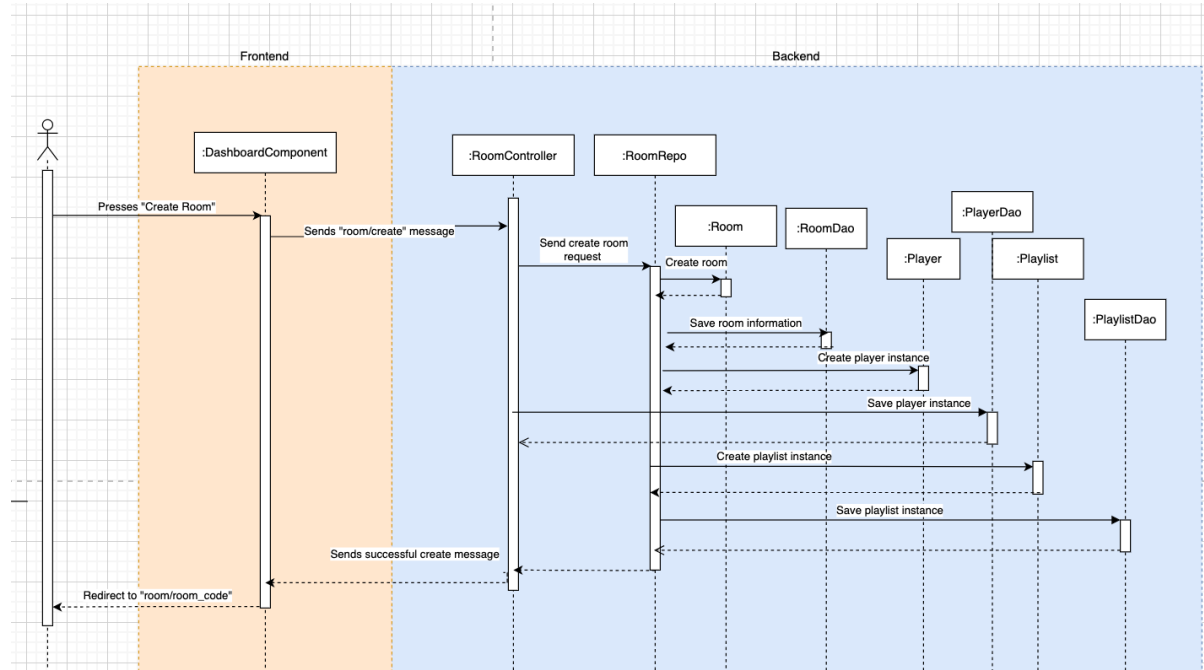


Fig. 4.2.3.1A Control flow of Room Creation

A user who has logged in will be presented with the DashboardComponent that has the "Create Room" button. Pressing this will send a WebSocket message to the backend. From there it is routed to the RoomController, then the RoomRepo which creates the domain objects such as the Room, Player and Playlist.

The creation of a room does a few things:

- Creates a new room entry in the Redis session store, with a generated room code, containing the information of the users in the room. The user who created the room will be added to the room and set as the owner of the room.
- Creates a new player entry in the Redis session store the with the room code attached, containing the state of the player for that room
- Creates a new playlist entry in the Redis session store with the room code attached, containing the state of the playlist in that room

Once this process has been completed on the backend, the backend returns a response via an acknowledgement to the WebSocket message emitted to the frontend, allowing the room host to join the newly created room. From here, the room host is able to control the various aspects of the player and playlist, as well as use the chat.

The room host will also be able to share the generated room code to other users he wishes to invite to the room.

#### 4.2.3.2. Room Joining

The room joining process starts from the user as well and is depicted in this control flow diagram:

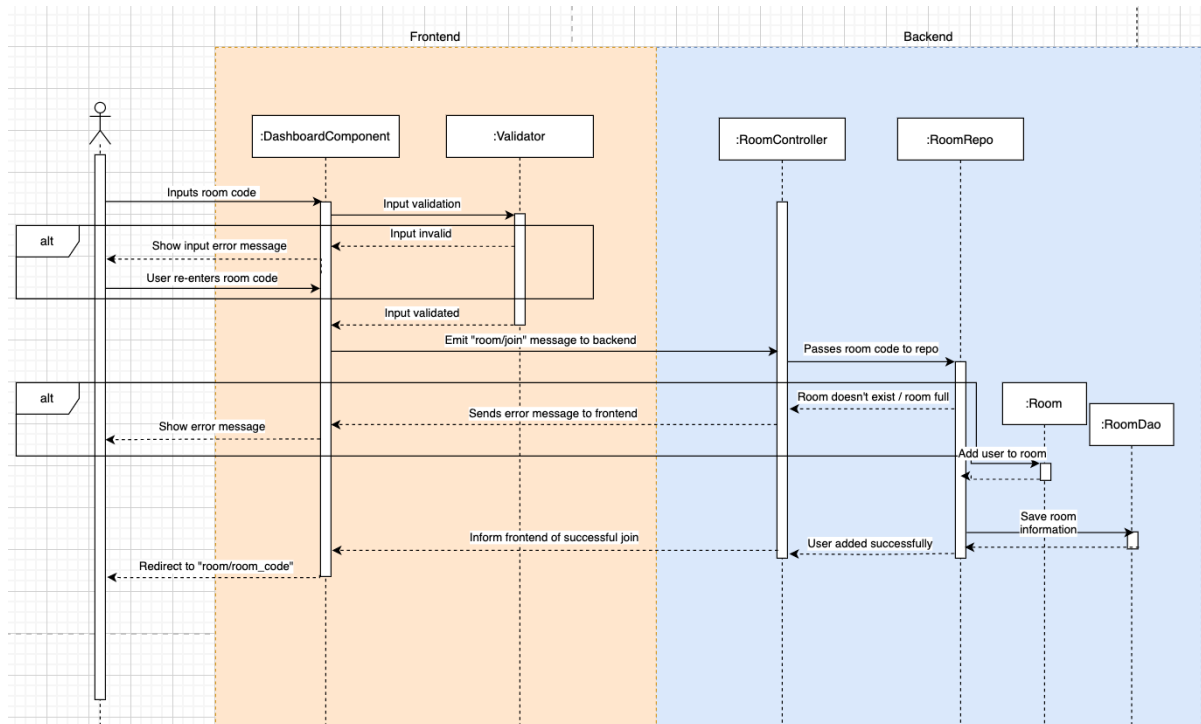


Fig. 4.2.3.2A Control flow of Room Joining

A user who has logged in will be greeted on the DashboardComponent with a “Join Room” button. This allows the user to input a room code that has been shared to them by a friend to allow them to enter that created room.

In the backend, the request for the user to join the room is handled by the RoomController, which passes on the request to the RoomRepo. A Room domain object is queried and deserialised from the RoomDao, and the user is added to the room. Then the Room object is sent to the RoomDao class to persist the information.

Entering the room code and submitting will send it through two layers of validation, similar to the login and registration process.

- Validator on the frontend ensures non-null, non-empty checks
- Backend validation ensures that the room code entered is valid (i.e. the room actually exists)

If validation does not go through, the user is prompted to try again. If it does, then the backend will inform the frontend that the user has successfully joined a room. The DashboardComponent will then redirect the user towards the room that they joined.

Moreover, the backend will emit to each of the users inside the room that a new user has joined, updating their information about who is in the room.



#### 4.2.4. Player Synchronisation

For Sync-Along to work, users in the same room need to have their actions made on the player synchronised. This means that any play, pause, next, previous or seek to action made on one user's client has to carry out the same action on the other clients. This is done via WebSockets provided by socket.io. An example of the "play" functionality is carried out here, where we assume that the room has already populated:

When a user presses the play button on the PlayerComponent, the client will begin playing the currently loaded song in the player. At the same time, a WebSocket message is sent to the backend via socket.io to indicate that the user in that room has made the "player/play" action.

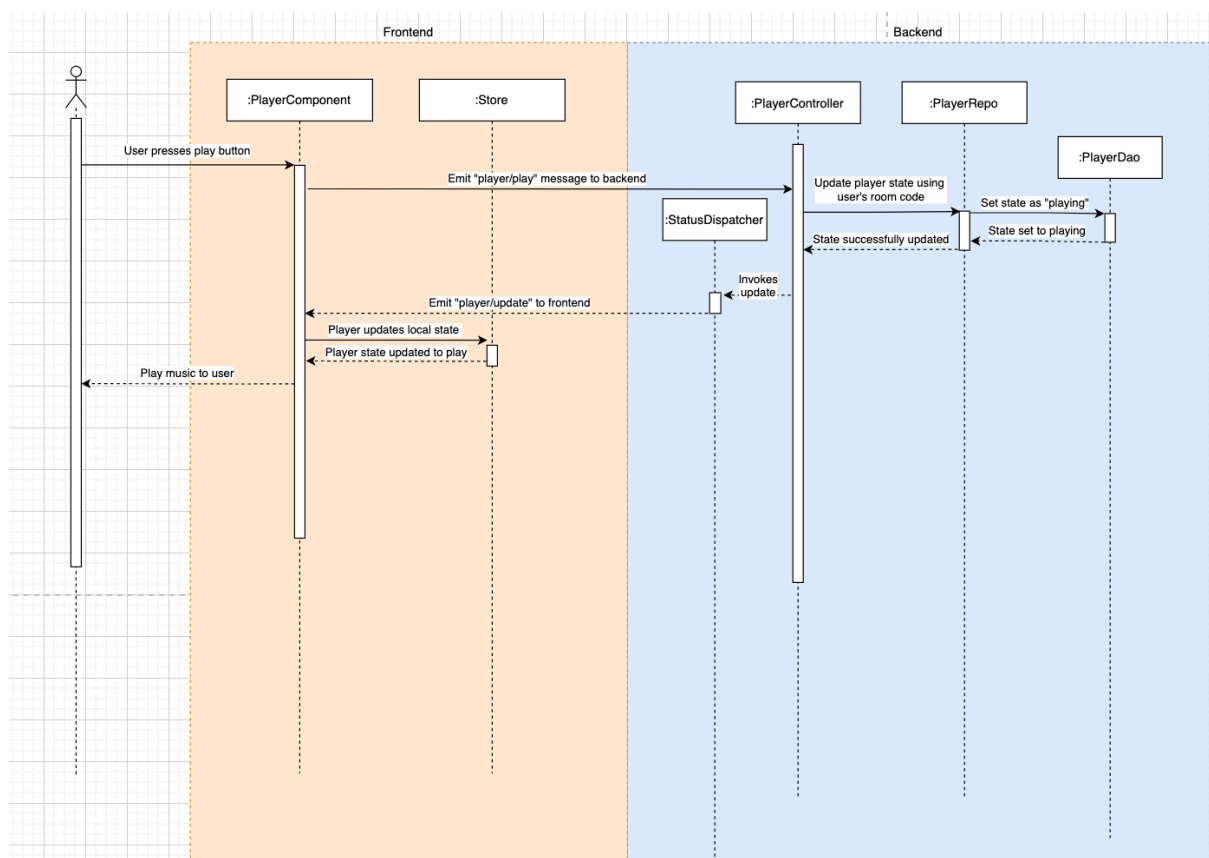


Fig. 4.2.4.1A Control flow of Player Synchronisation

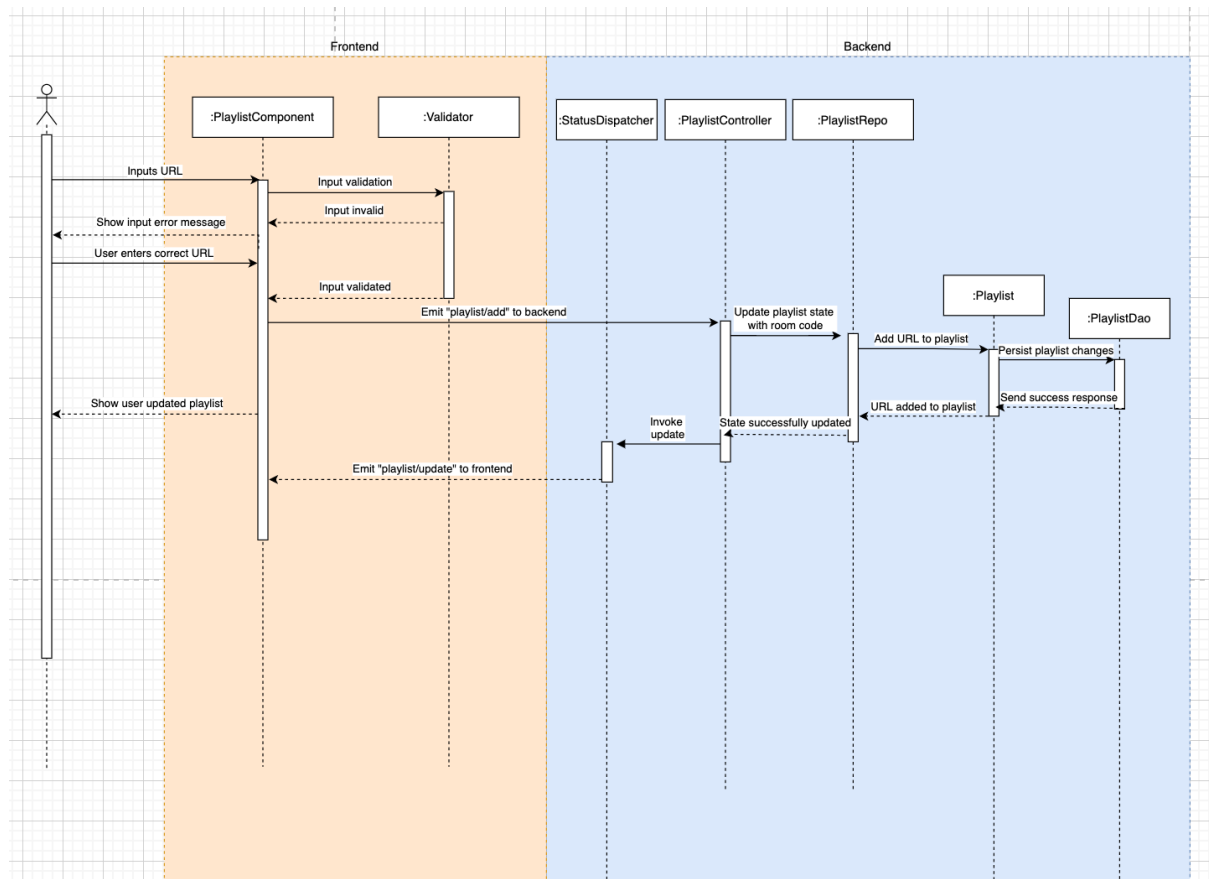
In the backend, the PlayerController will process this message, carrying out an update to the state via the PlayerRepo. The state includes the playing state of the player (whether it is playing or paused), the latest "scrubbed" time, and when that update was processed. The update is persisted through the PlayerDao class.

After which, an update is emitted to all users in the room with the new state via the StatusDispatcher. Each client has a listener to the message "player/update" on the frontend, and when this new state is emitted, they will update their state accordingly.

Note that the other actions act in a similar way, but send messages with their relevant values. Each action will cause the PlayerService on the backend to emit an update with the new state.

#### 4.2.5. Playlist Synchronisation

The playlist also has to be synchronised for each individual user in the room. Similar to the player, whenever an action is taken on the playlist that causes an update, it will send the relevant message via WebSockets to the backend for updates. A control flow for the adding of a new piece of media is shown here:



*Fig. 4.2.5.1A Control flow of Playlist Synchronisation*

When the user attempts to add a new song, after passing validation (i.e. valid YouTube URL), the frontend then emits a WebSocket message ("player/add" with the URL) to the backend via socket.io.

Upon receiving this message, the backend will pass on the message to the PlaylistController, which invokes the PlaylistRepo. The PlaylistRepo retrieves the Playlist object and persists the playlist changes in the PlaylistDao class.

After this update is processed, the PlaylistController invokes the StatusDispatcher. The StatusDispatcher dispatches a message ("playlist/update") to the frontend to update their state. This is then reflected in the various components that comprise the application.

## 5. Design Decisions and Technologies

### 5.1. General

#### 5.1.1. TypeScript

Plain JavaScript is a dynamically typed language, and type-checking is not as explicit as in TypeScript. In order to make sure that our code was working well on the frontend, we decided to use TypeScript to ensure that our code was type-safe, and this helped to reduce errors in our code, thus saving a lot of development time.

TypeScript also allows developers to define custom types. These types help to map the code to a concept or a domain, thereby reducing the code-model gap. For example, a user is represented as an object with a username and a password. We can represent that as a type in this manner:

```
type User = {  
  username: string;  
  password: string;  
}
```

To make it easy for developers to context switch between developing and deploying the frontend and backend as needed, both frontend and backend application code is written in Typescript.

#### 5.1.2. HTTP requests and WebSockets

During our discussions, we agreed that we wanted a proper registration and login system for the application and live updating for various parts of the application.

- HTTP requests -- the login process being reliant on HTTP requests is the standard way to do things. This does not require WebSockets since the updates are not “live” and the client doesn’t need to listen to any events regarding login and registration. Being “tried and tested”, it was straightforward to decide to use the same system other systems implemented for these authentication processes.
- Websockets -- the other features (rooms, chat, player, playlists) *do* require live updating of information, as we wanted the system to be responsive to the actions of all the users in the room. This involves the frontend having to listen to event emissions on the backend, which would in turn update the state on the frontend whenever an action was taken. Moreover, we needed *each client in the room* to respond to updates due to other clients’ actions as well.

We settled on using a mix of **Express** and **Socket.IO** to achieve this.

- Express helps with the login and registration process, providing various features that help to ensure that the development of the login and registration API is straightforward.

- Socket.IO helps implement real-time, bidirectional communication in a developer-friendly manner, enabling smoother and quicker development of our features.
  - It helped abstract the more intricate details of communicating via WebSockets, providing a simple API for developers.
  - It can handle rooms, allowing developers to add and remove users from rooms and only emit messages to the users in that room. This corresponded perfectly with our use case.
  - Whenever we need a request-response API similar to HTTP requests, we can make use of socket acknowledgements. This allows us to send responses similar to a HTTP response in a service that uses Socket.IO connections, and allows the frontend to standardise API calls to the backend.
  - It also has a lot of examples and relevant documentation available, making development easier.

### 5.1.3. Validation on both ends

Validation is done on the Sync-Along in both the frontend and backend. Although it would be ideal if we had one central class that controlled the validation process, we decided to split it up due to a few reasons:

- If the client-side was the only side to be delegated the validation process, there would be certain things such as rooms not existing that could not be checked.
- If the server-side was the only side to be delegated the validation process, it would mean reliance on an extra network request to check things such as empty inputs.

As a result, we decided to split it up into two parts: the client-side would check what it could before the submission of forms, which would reduce the number of network requests needed. On the server-side, we would check for the other required factors for input that the client-side would be unable to do. We felt that this was a good compromise, even though both ends would require some validation to be implemented.

## 5.2. Frontend

### 5.2.1. Technologies

The frontend of Sync-Along involves the use of the React framework with TypeScript for implementation. We used a mix of Redux (alongside Redux Toolkit) for the app's state management to handle the state mutation and updates.

#### 5.2.1.1. React

We chose React to be used on the frontend for a few reasons:

- It was an unfamiliar framework that we wanted to try to work with
- It has good performance when it comes to rendering and processing of various actions
- It is stateful, which is necessary for the components that we needed to implement
- It is compatible with Chrome on Windows, which fulfils our General System Requirement NFR (N4.1)
- It has wide developer and community support, with many sources as to how to implement new components and many third party libraries that support integration with React.

#### 5.2.1.2. Redux and Redux Toolkit

Inside our frontend, we rely on Redux and the Redux Toolkit extension for handling the application's state. We chose this as:

- React applications built with Redux tend to have a better structure, where React components and stateful actions are loosely coupled. React components can read and write from a global store in Redux, thus simplifying data flow.
- Bundling in React tends to produce a smaller bundle size, improving the deployment process and application performance
- Redux thunks help in the dispatching asynchronous actions, which is vital for the ensuring the responsiveness of the frontend
- Redux Toolkit provides a specific structure for writing Redux code along with sensible defaults, which is vital to ensure a structured codebase. Also, Redux Toolkit comes bundled with the RTK Query library, which provides features such as data fetching and caching logic. This decouples caching state away from application state, making the codebase more modular.

### 5.2.2. Client-sided Music Streaming

To make the process of using sockets easier, we decided to rely on the client side for the retrieval of YouTube videos for the playing of music. What we initially believed was that when music was to be played, the music file itself would be streamed from the backend to the frontend. However, a few things we realised:

- Socket.IO does not play friendly with serialised data, making it difficult for us to send a music stream across it
- We would have to implement our own music processing and temporary caching that would dispatch to each person in the room, which could compromise performance if many users were using the system

As a result, the “react-player” package was ideal for our particular use case. It allowed us to only have to maintain a list of URLs for the playlist, and delegate the music playing of the application to the frontend. This meant that we could transmit messages much more quickly (since the payload size would be much smaller), and it would be much easier to synchronise the player this way.

## 5.3. Backend

### 5.3.1. Technologies

The backend of Sync-Along involves the use of the Express framework in Node.js, with the application code written in Typescript. We chose this framework as it allows the application backend APIs to be built quickly and easily.

#### 5.3.1.1. Express

We chose Express to be used on the backend for a few reasons:

- Express provides a simple routing for requests made by clients.
- Express provides a flexible middleware module for doing extra tasks on response and request such as cookie verification.
- Many commonly used features of Node.js can be readily used.

#### 5.3.1.2. Socket.IO

For real-time connection between clients and the server, we used socket.io connection to allow events received from a single client to be emitted to the other clients. Establishing socket connection between every client and the server allows the server to listen to any events triggered by any clients, update the room state and music state immediately, and then emit the updated states to all other clients. This enables Sync-Along to keep the music playback to be in sync between all the relevant clients given any interactions with the music player in the room by the users.

#### 5.3.1.3. Redis Cloud

We use Redis Cloud as our in-memory store. We chose this because:

- Its simplicity and ease-of-use allows us to use simple command structure unlike query languages of traditional databases.
- It is faster to read and write from Redis than traditional databases, making it a great choice for a session store.
- Redis employs a primary-replica architecture and supports asynchronous replication where data can be replicated to multiple replica servers. This provides improved read performance (as requests can be split among the servers) and faster recovery when the primary server experiences an outage.
- Redis Cloud uses a durable transactional log that stores data across multiple Availability Zones (AZs) thus allowing it to be durable.

#### 5.3.1.4. MongoDB and Mongoose

For our choice of database, we chose MongoDB because of a few reasons:

- It works well with the Express backend server as MongoDB's document model can be easily mapped to API payloads which are JSON by nature.
- MongoDB Atlas cluster can be connected by Express backend easily via the Node.js Driver.
- MongoDB allows ad-hoc queries, real-time integration and efficient indexing.
- MongoDB is a reliable database which can be used to store persistent data, such as user data.

We used Mongoose, an ORM library that abstracts the querying and persistence of information to the MongoDB database. By specifying a schema, we can represent the User model in the database.

#### 5.3.1.5. ESSerialiser

ESSerialiser is a third party library used to serialise and deserialise classes into JSON strings before caching the data into Redis. While we considered using a Redis Object-Relational Model library (such as ts-redis-orm), we settled for using ESSerialiser to serialise and deserialise the domain objects, and implemented DAO classes that persist the serialised information in Redis. Deserialisation returns the domain object without the developer needing to manually instantiate the object.

We chose to go with the above solution because an additional ORM layer for Redis is more complicated to set up and introduces extra overhead. We also did not need the extra features that an ORM layer offers such as querying. Since we only needed to serialise and deserialise information, a third party library that offers that is just enough for our use case.

#### 5.3.2. Microservices to Monolithic architecture

At the start of the project, the group had decided to work on the backend with the microservices architecture in mind. We initially believed microservices would be suitable for our implementation because:

- The different services could be loosely coupled, and the requests we made from the frontend could be handled by the API Gateway.
- With each service being independent, scaling would be much easier as we could deploy each service as a Kubernetes node, and add settings to scale accordingly

However, as we started on the project, we realised that the system we had in mind became more inappropriate for our use case. One huge problem we faced during development was trying to integrate Socket.IO with our API gateway. The way that sockets worked meant that after establishing a connection, the socket had to be passed on to the various services for use, which was not possible if the API gateway was implemented separately from each of the services.

As a result of this issue, we decided to change the architecture to a monolithic one, as many hours of trying to find a solution led us nowhere. A monolithic structure, although compromising the scalability of the application and introducing increased coupling, would help us to achieve our desired functionalities. The socket connection could be passed on to

the individual services, and that connection could be used as the “gateway” through which communication with the frontend could be carried out.

### 5.3.3. Backend Design Patterns

This section introduces certain design patterns that were used in the application’s backend, ensuring a cleaner codebase.

#### 5.3.3.1. Singleton Pattern

The Singleton pattern ensures that only one instance of a class is created. This is useful for managing connections to MongoDB and Redis. Because initialising a connection is expensive, we want to ensure that only one connection is created to the databases. In the codebase, the MongoDB and Redis connections are managed via the `MongodbConnection` and `RedisConnection` classes respectively. They are both singleton classes that enforce that only one connection is maintained.

These connections are then accessed by the various DAO classes such as the `RoomDao`, `PlaylistDao` and the `PlayerDao` classes.

#### 5.3.3.2. Facade

The Facade pattern is a design pattern that helps provide a simplified interface to the backend services. An example of a Facade in implementation is the `RoomRepo` class, which interacts with multiple DAO classes and provides a simplified set of APIs that the `RoomController` will interact with.



## 6. Testing

For testing, we mainly perform manual testing during development to ensure that the application works as expected, and testing after deployment to ensure that the application is working in production.

### 6.1. Decisions

We chose manual testing because of a few reasons:

- The application has to work across the frontend, backend and databases, and as a result, we needed to test everything together
- Writing test cases may help to isolate some small issues on either end, but the chunk of the testing should cover the entire application to ensure that everything is working fine
- Manual testing helps us root out any core issues within the usability of the application itself from the user's perspective

### 6.2. Stress Testing

#### 6.2.1. Categorise tests

We can categorise our tests into [HTTP requests and WebSockets](#). The tools used are also mentioned.

- API requests using [hey](#)
  - Requests that require backend to interact with the MongoDB database
  - Requests that backend does not interact with the MongoDB database
- Sockets using [Socket To Me](#)
  - Sockets that require backend to interact with the Redis cache
  - Sockets that do not require backend to interact with the Redis cache

We have also tried Python's **requests** library for sending API requests. However, it seems like the server will block repeating requests from one client after a while, whereas **hey** does not have this issue.

#### 6.2.2. Analysis

It is straightforward that requests that need to interact with [MongoDB](#) or sockets that need to interact with [Redis](#) cache will incur a longer response time compared to those without. For example, when the users press the button to log in:

- The corresponding request is sent to the backend
- The backend will process the API request, possibly interacting with the MongoDB database
- The backend returns the result to the frontend
- React re-renders the component with the new data

The main bottleneck in our request's response time is the database/cache query. Without the database/cache queries, the request's response time is negligible since it is only affected by

- Network I/O between the frontend and backend server

- Code execution time. Since we are only handling simple logic (e.g. check whether fields are valid, strings are non-empty/equal), this is very small.
- React re-rendering the affected components

We can conclude that the response time for our API requests/sockets is largely dependent on the MongoDB database/Redis cache response time.

However, our MongoDB and Redis are set up on the cloud. This means that as long as we allocate more CPU and memory to MongoDB and Redis (i.e. paying more for the resources), we can handle large amounts of API requests and sockets and have a better response time. [Non-functional requirements](#) can easily be met by spawning large instances of MongoDB and Redis cache so that it can handle the requests/sockets at a large scale.

Regardless, for our [deployment](#), we are using the free tier of MongoDB and Redis cloud. Our code works well and response time is reasonable with general use cases (login/register, add songs, play/pause music, chat functions). [Synchronising the play/pause/skip music of the music player across all users in the room within 1 second](#) [N2.2] is achieved with the use of our free-tier Redis cloud.

## 6.3. Test Cases

Based on the functional and non-functional requirements, a list of test cases have been generated. Each test case has the following format:

- Test case ID: <Service>\_<requirement ID>.<number>. For example, the first test case in the User Management Service for F1.1 would be named User\_Management\_1.1.1
- Test Scenario
- Pre-Conditions
- Test Steps: the list of steps need to be taken by the user to test this feature
- Test Data: the expected inputs (e.g a valid username or a valid password)
- Expected Results
- Postconditions: the expected behaviour at the frontend after the request is fulfilled

The full list of test cases can be found in the following link:

<https://docs.google.com/spreadsheets/d/1WT9t1S5vv2FJ8rU6CECWCvTMzRNBhZrhYs--da gLRdo/edit?usp=sharing>

## 7. Deployment

### 7.1. Automated code building

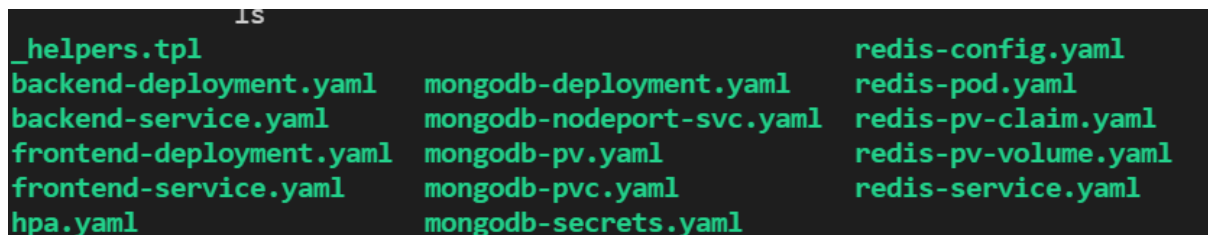
We use **GitHub Actions CI** to build the code in the master branch whenever a commit to the **master** branch is made. Building the code helps us to ensure that the code has no critical errors prior to manual deployment using [AWS EC2 and pm2](#). It also saves us a lot of time in having to re-run commands each time a new update has been made.

### 7.2. Deployment to Production

#### 7.2.1. Helm chart deployment

Helm-chart deployment uses a self-contained Kubernetes cluster with all application code inside with MongoDB database and Redis cache. **Helm chart** is a group of Kubernetes manifests that is combined into a single package (called chart). Helm chart helps us tremendously to further automate the deployment and scaling of Kubernetes resources.

With the initial idea of a self-contained Kubernetes cluster, here is how the Kubernetes manifests look like:



```
_helpers.tpl
backend-deployment.yaml  mongodb-deployment.yaml  redis-config.yaml
backend-service.yaml     mongodb-nodeport-svc.yaml redis-pod.yaml
frontend-deployment.yaml mongodb-pv.yaml           redis-pv-claim.yaml
frontend-service.yaml    mongodb-pvc.yaml         redis-pv-volume.yaml
hpa.yaml                 mongodb-secrets.yaml     redis-service.yaml
```

*Fig. 7.2.1A The Kubernetes manifests used for deployment*

##### 7.2.1.1. Manifests

- **backend-deployment.yaml**, **backend-service.yaml** are self-explanatory -- they are individual .yaml manifests that define a Deployment and a Service for the backend.
- **frontend-deployment.yaml**, **frontend-service.yaml** -- these define a Deployment and Service for the frontend.
- **hpa.yaml** -- defines a Kubernetes Horizontal Pod Autoscaler (HPA) that allows us to scale the number of pods for the **backend** or **frontend** based on the application's CPU and/or memory utilization
- **redis-pv.yaml** -- defines a PersistentVolume that allows Redis to be allocated storage in the cluster to store its data.
  - **redis-pvc.yaml** defines a PersistentVolumeClaim that allows the user to claim the storage space above. PVC has a one-to-one mapping to PV.
  - Deployments that want to use persistent storage will only interact with the PVC and not PV. **redis-config.yaml**, **redis-pod.yaml**, and **redis-service.yaml** are self-explanatory.
- MongoDB has a similar configuration. The only difference is **mongodb-secrets.yaml** which contains authentication credentials for the database.

## 7.2.2. Benefits of Helm Charts

### 7.2.2.1. Automated deployment process

Helm Chart automates our deployment further because it will deploy resources in the correct order of dependencies. For example, the Redis manifests have the following dependencies:

**redis-pv** ← **redis-pvc** ← **redis-pod**  
**redis-config** ← **redis-pod**

where **A** ← **B** denotes that resource B is dependent on resource A. With the manual running of Kubernetes commands, the users would need to deploy the Resources one by one such that it follows this order.

With helm chart, **helm install** will deploy all the Resources at once; **helm uninstall** will also remove all the Resources at once.

### 7.2.2.2. Ease of changing configuration

Helm chart's yaml files are heavily parameterised:



```
! backend-deployment.yaml X
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: {{ index .Values "backend-component" "deployment" "name" }}
5    labels:
6      {{- include "sync-along.labels" . | nindent 4 }}
7  spec:
8    {{- if not .Values.autoscaling.enabled }}
9    replicas: {{ .Values.replicaCount }}
10   {{- end }}
11  selector:
12    matchLabels:
13      {{- include "sync-along.selectorLabels" . | nindent 6 }}
14  template:
15    metadata:
16      {{- with .Values.podAnnotations }}
```

7.2.2.2A The backend-deployment.yaml, with heavily parameterised values

The values of the above fields will be populated through a **values.yaml** file.

```
# values.yaml
replicaCount: 1

backend-component:
  deployment:
    name: backend-deployment
  image:
    repository: backend
    pullPolicy: IfNotPresent
    # Overrides the image tag whose default is the latest tag
    tag: latest
  service:
    name: backend-svc
    type: LoadBalancer
    port: 4001 # external port
    targetPort: 4001 # internal port
```

7.2.2.2A The `values.yaml`, used to populate the `backend-deployment.yaml`

For example, `{{ index .Values "backend-component" "deployment" "name" }}` will be populated as `backend-deployment` when we deploy the Helm charts.

Thus, any configuration changes in the Helm chart can be done through the `values.yaml` file. This makes it easy for us to change the configuration of the Helm chart deployments compared to traditional Kubernetes.

## 7.3. Deployment Challenges

### 7.3.1. Issues with Kubernetes

We initially used Google Kubernetes Engine with the free credits being offered. However, there were major challenges when deploying to GKE/Kubernetes that prompted us to switch to a simpler alternative:

- Buggy deployment of MongoDB in Kubernetes. We have found many resources on deploying MongoDB to Kubernetes; however, most of the instructions given are OS/distribution-dependent and were not working in GKE. There are multiple error messages and frequent crashes in Deployment (CrashLoopBackOff in Kubernetes).
- Unable to connect to Redis from the backend server. Redis was not able to receive any of our requests. We encountered something similar to [this](#), but there was no working solution.
- Unable to connect the frontend server to our backend server. This is because of two main reasons:
  - CORS errors - since our frontend and backend server is on different cluster IP addresses. This requires changes in the CORS configuration on our backend NodeJS server and adding CORS-related headers to our API requests from the backend. However, testing out these changes in GKE is very time-consuming, because with every change of the code, we need to rebuild the code using Docker. We were not able to come up with a combination of

backend CORS configuration and frontend API requests configuration that work in GKE. Our socket connections between frontend and backend encounter a similar issue; and it also took a lot of time debugging to make it work for our use cases

- Unable to connect to the backend server using Cluster IP addresses or service host name.

Within each Pod in the cluster, Kubernetes provides the Cluster IP of all the services in the cluster with the `process.env` variable. Hence, each Pod can connect to all Services in the Cluster using the Cluster IP. However, we were unable to do so, even though the connection is working on our local Kubernetes cluster.

Our guess was the incompatibility between the API/socket requests sent out by the frontend and the receiving of those requests in the NodeJS/Express backend server. This can be related to our CORS issue above where our frontend requests are not properly constructed to be received correctly at the backend.

As a result, we were not able to construct a working frontend request because it was very hard to test with each change of the code - we needed to rebuild the Docker image and re-deploy the Kubernetes Resources and this took at least 5 minutes each time.

### 7.3.2. [Redis](#) and [MongoDB](#) from cluster to cloud

With the setting up of Redis and MongoDB challenging in our GKE cluster, we decided to host these two resources on the cloud instance. There are many advantages of this:

- Redis and MongoDB on the cloud are much easier to set up. We were able to set up Redis Cloud and MongoDB Atlas in less than 10 minutes.
- CPU and memory allocated to Redis Cloud and MongoDB Atlas can be easily managed and configured with the cloud platforms.
  - If our cloud storage is under load, we can easily scale up our CPU and memory in less than a minute.
  - This is opposed to configuring storage manually in GKE/Kubernetes with the use of PersistentVolumeClaim and PersistentVolume.
  - The configuration of memory storage in Kubernetes is also limited by the memory cluster itself which might be troublesome.
- Passwords/secret keys can also be easily managed on the cloud platforms. In GKE/Kubernetes, this corresponds to a ConfigMap or Secrets resource which took time to set up.
- Performance monitoring is provided as built-in in cloud platforms. In GKE/Kubernetes, this corresponds to setting up a resource called Prometheus which makes our cluster even more complex.
- Easier access and debugging from the public internet. In development, we can run the code locally but use the Redis/MongoDB as cloud storage to aid coding and developing. Redis and MongoDB resources could also be accessed publicly from the cluster, but this is harder to set up.
- Easy collaboration between different team members, since we can all use the cloud database

- Enjoy built-in Disaster Recovery features of the platform. This makes our storage more reliable and error-resilient.

The disadvantage of using cloud storage would be:

- Increased security risk. This is mostly applicable for the MongoDB database where we store users' emails and (hashed) passwords and other sensitive data. MongoDB provides a password which helps to alleviate this.
- Our database would be down if the cloud provider is down. However, this is unlikely.

The benefits of using cloud storage for Redis and MongoDB far outweigh the disadvantages; it's clear that deploying storage to the cloud would be the best option for our project.

### 7.3.3. Application code from GKE to AWS EC2

With the difficulty of containerizing the application code in Docker and deploying to GKE/Kubernetes due to networking configurations, we decided to deploy the application on AWS EC2 instead.

#### 7.3.3.1. Why AWS EC2

One of the main reasons that we choose EC2 is because EC2 provides the Ubuntu Server 20.04 production environment, which resembles our development environment. Minimal code changes are required when code is transferred from development code to production/remote code.

The connection from the user to our AWS instances can be facilitated using AWS built-in network firewalls.

#### 7.3.3.2. Our approach

We deploy the frontend server on one AWS EC2 instance; and the backend server on another AWS EC2 instance. This approach achieves **scalability**, because:

- EC2 allows us to provision CPU and memory for its instances. With this design, we can allocate more CPU and memory for the backend because it needs to handle a high load of API requests and sockets, improving the response time of the requests from the user.
- Ideally, AWS EC2 Auto Scaling can be set up so that it can scale up when the application has more load and scale down when the application is idle.

The obvious drawback is we need to connect the two servers. However, we can simply connect them using their public IP address, with network access restricted using AWS' network firewall. CORS does not create any issues here.

#### 7.3.3.3. pm2

pm2 helps us to run our application persistently in the AWS instance. pm2 is great for our project because

- Easier deployment compared to Kubernetes
- No code change required, simply "wrap" the pm2 around npm commands
- Highly compatible with npm, our application's package manager

- Provides CPU and memory resources monitoring which compliments well with AWS' CPU and memory allocation
- Allows us to spawn multiple subprocesses for the backend server, thus improving response time compared to only having a single-process single-thread NodeJS/Express server.

### 7.3.4. Final application deployment

#### 7.3.4.1. Scope Limitation

We limit our GitHub Actions CI build and the production build to **Node v12** because we have been using Node v12 for our development environment. Moreover, using only Node v12 in production will facilitate easier package management in **npm**. It also simplifies the testing process.

Finally, the deployed app consists of the following technologies:

- MongoDB Atlas
- Redis Cloud
- Frontend on AWS EC2 instance using pm2
- Backend on AWS EC2 instance using pm2

All resources are available on public IP addresses/hostname which facilitates the seamless connection between them.



## 8. Suggestions for Improvements

The following section lists down some areas in which we think the application or our development process can be improved.

### 8.1. Incorporation of CI/CD to application

As we learned in CS3219, continuous integration and deployment have a number of useful benefits. In an ideal situation, we would be implementing CI/CD in our application, which would ensure that new changes made would not compromise the currently available functionalities.

A possible way of implementing continuous integration would be to write unit tests in frameworks like Mocha and Chai. While more difficult to figure out, frontend unit tests can be written in frameworks like Jest. Dependencies, such as the MongoDB database, can be stubbed out using third party libraries. With unit tests in place, Continuous Integration can then be fully set up in GitHub Actions.

Continuous deployment can be set up. The application build's image from GitHub Actions CI can be automatically pushed to AWS ECS. AWS EC2 can then pull the build from ECS and then run the containers. With this, we would complete the full cycle of Continuous Deployment.

### 8.2. Improve application security

Given more time, we would have improved the general security of the web app. We decided to sacrifice a bit of security because of:

1. Faster application development times: the need to implement and bypass application security measures will slow down development, thus we prioritised feature development as they are not completed yet.
2. Improvement in user experience: we wanted to make the registration and user login process as seamless as possible, without the need for multiple checks such as 2-Factor Authentication or email verification.
3. No storage of personally identifiable information: our app does not require users to store personally identifiable information before using our app. Therefore, we do not need to implement many protocols to safeguard user data.

However, as our app matures, it would be better to implement more protocols and checks to ensure app security.

#### 8.2.1 Add more authentication to sockets and API requests

Our current implementation of the application requires the username and password to be entered before the user is able to use the features of the application. However, this is the only form of authentication that we have implemented in the application.

Currently, API requests or socket messages are still able to be sent to the backend from the frontend and will be registered, even without any authentication. This means that a malicious

user could potentially send messages to the backend to change the state of existing rooms, disrupting the use of the application by proper users.

An improvement thus could be made in this aspect. For this, we would take the following steps:

- Actions made in rooms (e.g. control of the player and playlist, sending of chat messages) would require the backend to verify that the user's session is valid before the action is registered and dispatched to other clients.
- Navigation to the deeper parts of the application (e.g. dashboard, room) would be governed not by the login status of the frontend, but rather whether the user has a valid session established on the backend.
- Invalid actions or navigation would be rejected and prompt the user with a login if they wished to proceed.

### 8.2.2 Add more checks to user management services

We could improve our user management service by implementing the following features:

- Implementation of basic HTTP authentication or JWT authentication into our app
- Implementing email verification services, such as requiring users to verify account creation with their email addresses to complete the registration process

## 8.3. Improved User Experience

While our application is functional and has a UI created with Bootstrap, we could have let our friends test the app and provide feedback on how to improve the features and user experience of the app. This would also help us improve the user experience of the app, which is an important, but often overlooked area in software development.

Another big area that would help improve the user experience is to make the application responsive to different screen sizes because users can access the app via multiple devices (PCs, laptops, mobile phones).

## 9. Reflections and Learning Points

### 9.1. Time Management

Firstly, our team could have been better at time management. While we set out a developer schedule at the beginning of the project, we did not adhere to the schedule.

One major blocker was that we tried to develop the app in a microservices architecture and pivoted to a monolithic architecture too late. We could have set a deadline for exploring the microservices architecture, then moved to the monolithic architecture somewhere around Week 10 or Week 11, where there would have been more time to pivot and properly structure a monolithic architecture.

Another blocker was in the exploration of new libraries, such as the Redux Toolkit. While we managed to implement Redux Toolkit in the frontend application, this greatly delayed our schedule and resulted in rushed work towards the end of development.

### 9.2. Early Testing

Secondly, to improve the code quality and robustness of the app, our team could have worked on early testing. For starters, we could have set up a list of test cases earlier. Ideally, these test cases would be generated before developing the app. By setting up the test cases earlier, it allows us to verify the correctness of the app early on in the development process, be it through [manual testing](#) or automated testing.

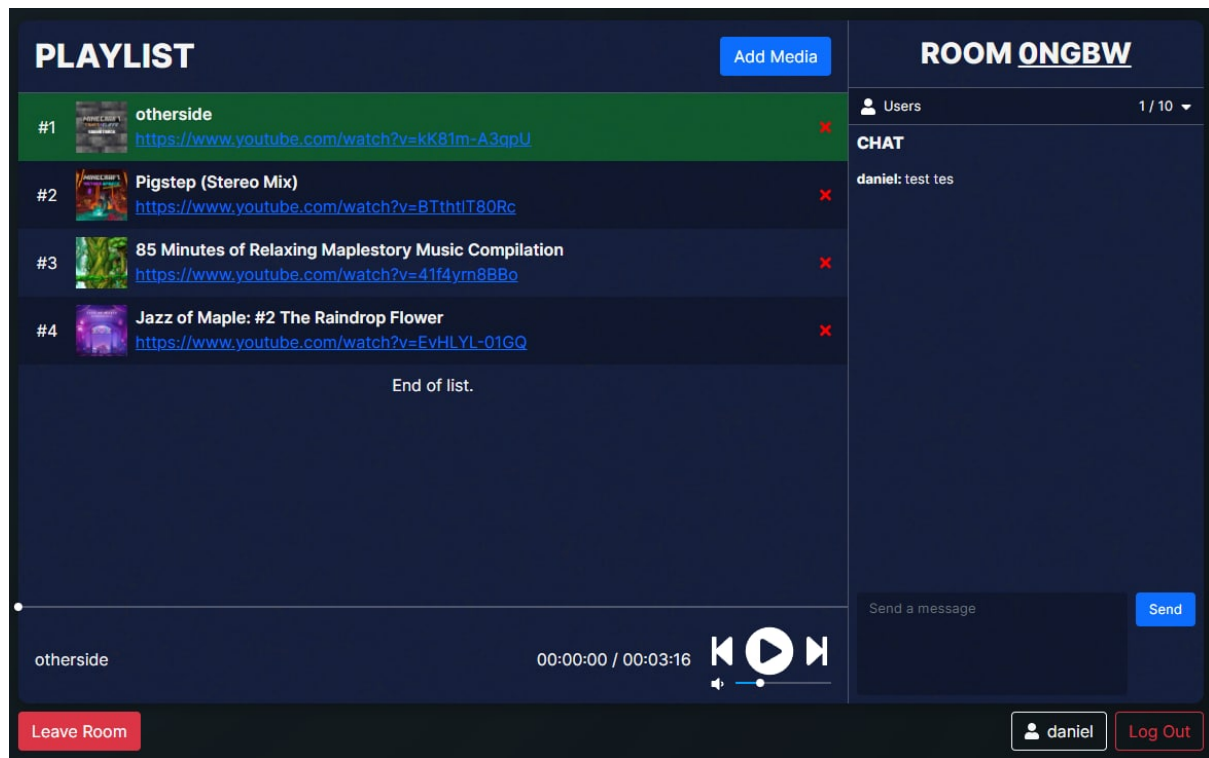
Additionally, setting up testing early would enable us extra time to perform [load testing](#), which is important to ensure that our app scales up even before we acquire that amount of user traffic. For example, we can perform load tests on Socket.IO using Artillery.IO.

### 9.3. Improved documentation process

To improve documentation of the app, we could make use of tools such as Swagger to automatically generate API documentation. Besides automating documentation, Swagger is also widely known within the developer community. Documentation generated with this tool thus complies to standards well known by developers, which helps to ramp up new developers if any.

## 10. Conclusion

Lastly, we greatly enjoyed creating the app and learnt a lot along the way, such as the usage of RTK Query and Socket.IO to deliver a real-time web application. This was a great project where we also tried to practice the implementation of design patterns and software architecture.



*Figure 10 Screenshot of the app*

The end result was an app that our team members enjoyed using with our friends. In general, we are satisfied with how this project turned out.

# 11. Reference

## 11.1. API Documentation

The following section lists the APIs used to facilitate communication between the frontend and the backend.

### 11.1.1. User Management Service

The user management service communicates with the backend via HTTP Requests.

#### 11.1.1.1. Registering

URL: [http://{BACKEND\\_URL}/register](http://{BACKEND_URL}/register)

Method: POST

Body of the request is a JSON object with the following fields:

```
{
    username: string,
    password: string,
}
```

A successful registration is indicated by a HTTP response of status code 200 and the following body:

```
{
    isSuccessful: true,
    message: "User Registration is successful.",
}
```

If the request does not contain the required parameters, such as a missing username or password, a status code of 401 is sent along with the following body:

```
{
    isSuccessful: false,
    message: "Some fields are missing. Please fill up all the fields",
}
```

#### 11.1.1.2. Login

URL: [http://{BACKEND\\_URL}/login](http://{BACKEND_URL}/login)

Method: POST

Body of the request is a JSON object with the following fields:

```
{
    username: string,
    password: string,
}
```

A successful registration is indicated by a HTTP response of status code 200 and the following headers:

```
{
    Access-Control-Allow-Credentials: true,
    Access-Control-Allow-Origin: // backend app URL,
    Set-Cookie: // a cookie indicating the session
}
```

And the following body:

```
{
    isSuccessful: true,
    username: string,
}
```

If the request does not contain the required parameters, such as a missing username or password, a status code of 401 is sent along with the following body:

```
{
    isSuccessful: false,
    message: "Some fields are missing. Please fill up all the fields",
}
```

If the password fields are present but invalid, a status code of 401 is sent along with the following body:

```
{
    isSuccessful: false,
    message: "Passwords have to be at least 6 characters.",
}
```

If the user is already registered, a status code of 401 is sent along with the following body:

```
{
  isSuccessful: false,
  message:"This user is already registered. Please log in.",
}
```

#### 11.1.1.3. Logout

URL: [http://{BACKEND\\_URL}/logout](http://{BACKEND_URL}/logout)

Method: POST

A successful logout is indicated by a HTTP response of status code 200 and the following body:

```
{
  isSuccessful: true,
}
```

A successful request also clears the cookies that were set in the user's browser. The next time the user accesses the application, they would have to log in again.

If the request does not contain the required parameters, such as a missing username or password, a status code of 400 is sent along with the following body:

```
{
  isSuccessful: false,
}
```

## 11.1.2. Room Management Service

The room management service communicates with the frontend by emitting events via Socket.IO. There are two types of events: Client-to-Server events and Server-to-Client events.

### 11.1.2.1. Client-to-Server events

#### 11.1.2.1.1. Creating a new room

Event: room/create

Data: An object of the following format:

```
{
  username: // string,
}
```

If the room creation is successful, the server sends an acknowledgement to the client with the following message body:

```
{
  status: 200,
  code: // the room code
}
```

If the event is passed in without a username, the server also sends an acknowledgement with the following message body:

```
{
  status: 400,
  isSuccessful: false,
  message: "Please provide a username to create a room."
}
```

#### 11.1.2.1.2. Joining a room

Event: room/join

Data: An object consisting of the following fields:

```
{
  username: string,
  room: // the room code of the application
}
```

If the room exists and the user successfully joins the room, the server sends an acknowledgement to the client with the following message body:

```
{
```



```
    status: 200,  
    isSuccessful: true,  
  }
```

If the requested room cannot be found in the Redis session store, the server sends an acknowledgement to the client with the following message body:

```
{  
  status: 400,  
  isSuccessful: false,  
  message: "The requested room cannot be found.",  
}
```

If the requested room is full, an acknowledgement is sent with the following message body:

```
{  
  status: 400,  
  isSuccessful: false,  
  message: "The room you are trying to join is full  
currently.",  
}
```

#### 11.1.2.1.3. Leaving a room

Event: room/leave

Data: no data needs to be sent when this event is emitted.

If the user successfully leaves the room, the server sends an acknowledgement with the following message body:

```
{  
  status: 200,  
  isSuccessful: true,  
  message: "You have successfully left the room.",  
}
```

If there is an error while leaving the room, the server sends a callback with the following message body:

```
{  
  status: 400,  
  isSuccessful: false,  
  message: "An error occurred when leaving the room.",  
}
```

### 11.1.2.2. Server-to-Client events

#### 11.1.2.2.1. Updating the room status

Event: room/update

Data: the server sends the client a list of data consisting of the following:

```
{
  isValidRoom: // boolean,
  users: // an array of users consisting of users, where each
user is of the format: { username: string; isOwner: boolean;
},
  userCount: // a number indicating the number of users in the room
}
```

### 11.1.3. Room Chat Service

#### 11.1.3.1. Client-to-Server events

##### 11.1.3.1.1. Sending Chat Messages to a room

Event: chat/message

Data: an object of the following format:

```
{
  message: // a non-empty string
}
```

#### 11.1.3.2. Server-to-Client events

##### 11.1.3.2.1. Broadcasting Chat Messages to the room

Event: message

Data: an object with the following format:

```
{
  username: // username of the app,
  text: // chat message of the app,
  time: a date with the format H:MM a
}
```

## 11.1.4. Music Playlist Service

### 11.1.4.1. Client-to-Server events

#### 11.1.4.1.1. Adding a song to the playlist

Event: playlist/add

Data: an object consisting of the following fields:

```
{  
  url: // a valid Youtube URL  
}
```

#### 11.1.4.1.2. Removing a song to the playlist

Event: playlist/remove

Data: an object consisting of the following fields:

```
{  
  id: // the ID of the song to be removed  
}
```

#### 11.1.4.1.3. Selecting an active song in the playlist

Event: playlist/select

Data: an object consisting of the following fields:

```
{  
  id: // the ID of the song to be selected  
}
```

#### 11.1.4.1.4. Moving to the next song in the playlist

Event: playlist/next

Data: no data needs to be sent when emitting this action

#### 11.1.4.1.5. Moving to the previous song in the playlist

Event: playlist/prev

Data: no data needs to be sent when emitting this action

#### 11.1.4.1.6. Playing a song in the player

Event: player/play

Data: the timestamp of the current player in UNIX format.

#### 11.1.4.1.7. Pausing a song in the player

Event: player/pause

Data: the timestamp of the current player in UNIX format.

#### 11.1.4.1.8. Scrubbing the time in the player

Event: player/scrub

Data: the timestamp of the current player in UNIX format.

#### 11.1.4.1.9. Indicating the song is complete in the player

Event: player/complete

Data: no data is sent

### 11.1.4.2. Server-to-Client events

#### 11.1.4.2.1. Updating the player status

Event: player/update

Data: an object is sent with the following format:

```
{
  playlist: an array of songs, where each song has the format of
  {
    id: number,
    url: // a youtube URL
  },
  current: the current active song being played.
}
```

#### 11.1.4.2.2. Updating the playlist status

Event: playlist/update

Data: an object is sent with the following format:

```
{
  roomCode: // room code of the app,
  lastScrubTime: // the last time of the application,
  lastUpdateTime: // the last updated time of the music player,
  isPlaying: // boolean,
  waiting: // a number indicating the number of users that is waiting
for the next song to play,
}
```