Group Music Server

Ben Basten

https://github.com/ben-basten/group-music-server

Introduction

Last spring, I headed back to Minnesota to take classes online while COVID was starting to spread in the US. I did not get to see many people other than my immediate family members, so I started looking for ways to stay connected with my friends that I could no longer see every day. I found myself frequently using services to watch Netflix or Hulu simultaneously, playing online Pictionary games with friends, or listening to album drops on Spotify while video calling someone. The common thread between all these things is that I value sharing experiences with other people.

With this in mind, I looked at my personal collection of music and the initial idea for my senior project started forming. What if I could create listening rooms where friends, family, and peers could join and listen to the same simultaneous stream of music? I have a fascination with web design, so putting this into an easy web interface is a cool and easy way to allow sharing and make the page accessible to anyone.

There were a couple iterations for how this could look, including having multiple room types like a DJ'ed room where one person chooses the music. However, I ended up settling on all rooms being the same such that anyone can add to queue and skip music. I also did not want this application to look like a public game lobby, so users must join with a room code that they were given by a friend rather than by choosing from a list of rooms. After settling on these boundaries, I felt satisfied that this would be the personal listening experience I craved when I was stuck in quarantine in Minnesota.

Design – Backend

For the server-side of the application, I decided to go with the Spring Boot framework. The framework is traditionally paired with Java, but it also supports using Kotlin or Groovy. I decided to use Kotlin to make me more familiar with the language and because of the ease of use with features such as null checking and primary constructors. I also chose the Spock framework for unit/integration testing, which uses the Groovy language. Creating a Spring project is as easy as choosing the language/build tool/dependencies and generating the bootstrapped code with "spring initializr" (https://start.spring.io/).

Spring Boot is a REST web application framework, which means that it uses HTTP requests for communication. An important characteristic of a RESTful server is that it is stateless, meaning that the server does not keep track of the identities of its clients that use the service. This is a huge driving factor in how I designed the project because this means that the client is in charge of keeping track of their identity, in this case the room ID.

Each active listening room in the application has a unique 4-digit room ID that is generated when the room is initially created. This ID is what clients use to join existing rooms. To prevent the inevitable duplicates that the probability "Birthday problem" warns against, the ID is re-generated until a unique one is found:

```
do {
    id = Room.generateRoomId()
} while(rooms.containsKey(id)) // guarantees no duplicates
```

The controllers in a Spring Boot application define what paths are available for clients to consume. Due to the stateless nature of REST applications, I split the endpoints into 2 different controllers:

1. **MusicController**: endpoints that any client can freely use. These operations include creating a room, joining a room, and getting the available music listings. In the case of displaying available music listings for instance, all rooms use this functionality but they all need the same results regardless of the room.
2. **RoomController**: endpoints that require the client to identify their room ID before any room-specific operations can be performed. The server is stateless, so a room ID can be provided as a request header, URL parameter, or request body. Operations that this controller facilitate include getting the current song file, getting the queue, adding to queue, and skipping to the next track. If an invalid room ID is provided, a "404 NOT FOUND" error is thrown.

To provide actual music to the client, the server indexes the local music library during compilation only. The code to achieve this is not necessarily elegant – it looks inside the Spring resources folder and finds all .mp3 files that are in the root directory or up to 2 directories deep (ex. folder1/folder2/song.mp3). These resources are then persisted in an indexed hash table for fast access later. These unique indexes are what will be later used to reference the track for operations such as adding to queue. Indexing the tracks as a private class variable means that they are not persisted if the server is stopped and get generated every time the server is rerun.

```
var resources: Array<Resource> = emptyArray()
try {
    resources = resourcePatternResolver.getResources("classpath:mu-
sic/*/*/*.mp3")
    resources += resourcePatternResolver.getResources("classpath:mu-
sic/*/*.mp3")
    resources += resourcePatternResolver.getResources("classpath:mu-
sic/*.mp3")
} catch (ex: IllegalStateException) {
    System.err.println("An error occurred getting the mp3 resources:
${ex.message}")
    System.err.println(ex.stackTrace)
}
val tracks: HashMap<Int, Track> = HashMap()
resources.mapIndexed { index, resource -> tracks.put(index, Track(index, re-
source))}
return tracks
```

Rooms are persisted in a similar way – I determined that a database would be overkill because these rooms are not meant to be permanent. Just like the music, the rooms are kept as a hash table. The key is the unique room ID, and the value is a Room model that contains the room ID, queue, and all of the necessary member functions to change a queue and get the current song file.

```
var rooms: HashMap<Int, Room> = HashMap()
```

An issue with REST HTTP requests is that they are only a conversation between the initiating client and the server. However, since many clients can be in a room at the same time there needs to be a way to notify them that the queue has changed if they are not the initiating client. I added basic WebSocket functionality to the server to solve this issue. In this case, the WebSocket is using STOMP (Simple Text-Oriented Messaging Protocol) to send JSON to and from the server. By design, WebSockets use the publish-subscribe pattern for data transmission. When the server is notified that the queue changed, it sends the updated queue information to all subscribers of the queue topic for their room. In this case, the server gets notified that the queue has changed through the "/queue/update" endpoint and sends out the new queue to subscribers of the topic "/topic/room/[roomId]/queue".

```
// when a client adds something to the queue successfully and is connected to
the websocket, notify other subscribers of change
@MessageMapping("/queue/update") // incoming endpoint that is prefixed by
/api/gms/ws
fun notifyQueueChangeToRoom(roomId: Int) {
    // sends to all clients that are subscribed to this topic
    simpMessagingTemplate.convertAndSend("/topic/room/${roomId}/queue",
roomService.getQueue(roomId))
}
```

Clients are automatically connected to the WebSocket immediately upon joining a room in the UI, so they are almost immediately ready to receive updates from the server.

Design – Frontend

Overview

For the client-side of the application, I chose the React JavaScript library. It is a very commonly used library for web user interfaces that is maintained by Facebook. Features such as dynamically re-rendering certain components on the page when they change for quick performance, hot development changes, and the flexibility of JavaScript made it an appealing option for me. React is tightly integrated with Node.js, so creating a project is as easy as entering the command "npx create-react-app group-music-server" to generate the starter code.

I wanted to keep the navigation as simple as possible, so there are only a couple of core URL paths for the full functionality of the application.

Paths:

- /
    - Takes user to the Lobby page, where they are presented with an option to either create a new room or go to the Join room page.
- /room/:roomId
    - Take the user to the Room page where the actual fun happens. The Room page has the player, queue, and listings of available music. ":roomId" indicates that this is a free path variable. For example, if the room ID is 1234, the path would look like "/room/1234".
- /join
    - The Join page, where a user can enter a room ID to join a room.

- If none of these paths match the URL, the user is directed to the not found page.

Lobby page & Join page

The primary function of the Lobby page is to create a new room, while the primary function of the Join page is to allow users to join an existing room. On the frontend, the API calls for creating and joining a room are handled in the same way. When a room is successfully created (endpoint: /api/gms/create) or joined (endpoint: /api/gms/join), the API responds with an object containing the room ID and current queue:

```
{
    "roomId": 7040,
    "queue": [
        {
            "id": 66,
            "title": "Let the Games Begin",
            "artist": "AJR",
            "album": "Let the Games Begin"
        }
    ]
}
```

With this information about the room in hand, the user gets redirected to the appropriate room page. The object also gets put into a React state, which the Room page will be able to access and immediately display the pre-fetched queue to improve performance and reduce the number of API calls within the room.

```
props.history.push({
    pathname: `/room/${response.roomId}`,
    state: { room: response }
});
```

Room page

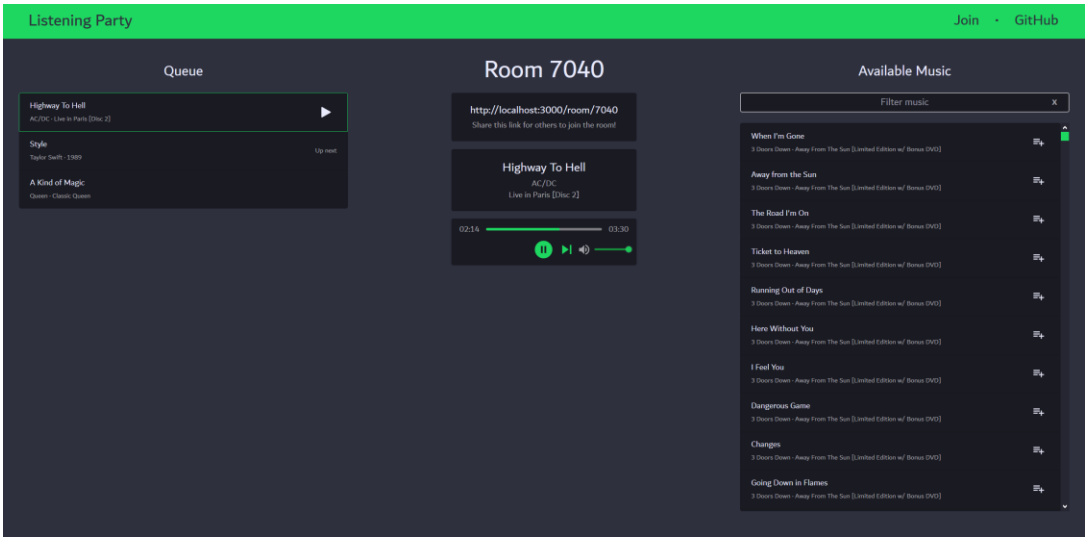


Figure 1: UI for widescreen device after successfully joining room

The Room page is where the core functionality of the project lives. Once users are in a room, this is where they can search for music, add to the queue, skip, play, and pause music.

There are a couple of steps that occur in quick succession before a room is fully loaded and ready to go. This all occurs within a “useEffect” block, which is invoked when the page is fully rendered for the first time.

```
useEffect(() => {
    // Step 1: figure out where the user came from
    if(props.location.state && props.location.state.room) {
        // the user came here from the join page
        setQueue(props.location.state.room.queue);
    } else {
        // the user navigated directly to the room page in the browser or re-
freshed
        attemptJoin();
    }

    // Step 2: asynchronously get the available music listings
    getMusicListings()
        .then(response => response.json())
        .then(response => setMusicList(response))
        .catch(() => {
            console.error("Something went wrong fetching the music list-
ings.");
        });

    // Step 3: connect to the WebSocket to make the room responsive!
    connectToSocket();
}, []);
```

Step 1: figure out where the user came from

There are 2 ways that a user made it to the useEffect page. Either from the Lobby/Join page, or by navigating to the URL directly. If the user came from the Lobby or Join page, it is already known that this is a valid room, and the queue is ready to go in the state. If the user navigated directly to the URL or refreshed the page, there is nothing in the state and no guarantee that it is a valid room. In this case, an additional API call is made to “/api/gms/join” to get the room information or kick the user back to the lobby if the room does not exist.

Step 2: get the available music

At this point, it is known that this room is valid and the queue is loaded. Now, the client needs to fetch the listings of all available music. This call is done asynchronously so that step 3 can get done sooner.

Step 3: connect to the WebSocket

To be able to responsively update the client when another user makes a change to the queue in the room, there needs to be an active and stable connection to the server WebSocket. The “connectToSocket” method handles all of this.

When I was initially designing this project, I thought that the server would need an endpoint to retrieve the information about the current track to display above the music player.

The implementation ended up being much easier than that and did not require an additional endpoint. Instead, I just use the first element in the queue array because that already has all of the information that I need.

```
<NowPlaying
    track={queue[0]}
    roomId={props.match.params.roomId}
    setQueue={setQueue}
    socketClient={client}
/>
```

With the NowPlaying component set up using the first element of the queue rather than a separate endpoint, it automatically changes the information card and .mp3 track in the media player when the queue array changes without any additional effort. It's very simple and it makes the page feel very responsive when the queue, player, and information card about the currently playing song all update simultaneously.

The WebSocket is used sparingly, but it is what truly makes the UI come alive for multiple users so that everyone can see the changes in real time. The API endpoints for skipping a track and adding to queue both return an object with the updated queue, so it's here that the WebSocket needs to be notified that the queue has changed:

```
if(socketClient.connected) {
    socketClient.publish({destination: '/api/gms/ws/queue/update', body:
roomId});
} else {
    setQueue(response);
}
```

If there is an open socket connection, the client will notify the server that the queue has changed and the clients will receive the new queue via the socket's topic that they are subscribed to. However, the socket connection can very occasionally have blips, so there is always the selfish default of just using the HTTP response and not notifying the other clients.

Styling

Sass is what is used to generate the CSS for the project. It is a CSS compiler that makes writing CSS easier by allowing easy creation of variables, reusable functions, and providing shorthand for nested selectors. For instance, the color scheme of the project is all setup in a single file by creating variables:

```
$primary-green: #1ed760;
$secondary-green: #16883a;
$slate: #2e303e;
$dark-slate: #1a1a22;
$hover: #262632;
$off-white: #e9ebfc;
$mute-grey: #a3a3a9;
$error: #ef5552;
```

The UI is responsive for all screen sizes. It looks best on a full widescreen monitor, but there is full functionality for mobile

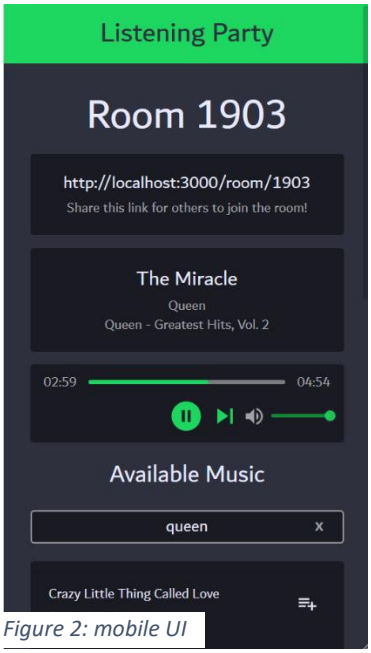


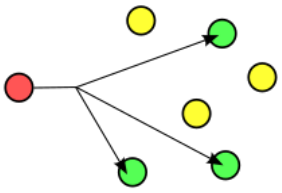
Figure 2: mobile UI

users with narrow displays or adding scrolling for tablet users with wide but skinny displays. I want all users to be able to enjoy this application, regardless of what device they are using.

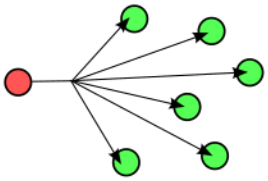
Challenges

A large portion of my time that I spent working on this project was spent stuck in analysis paralysis and trying to get my proposal off the ground. The thing that had me stuck: how is streaming done in modern web development? And to add another layer of complexity, how can this be done simultaneously with multiple clients that want to consume the same stream?

After doing some research into the issue, I realized that what I was looking for is some sort of multicasting technology. Multicasting is considered a "one-to-many" type communication, where it is sending the same data to select groups of clients at the same time. This is different than broadcasting, where simultaneous data is sent to all clients. Finding a solution for multicasting would fit perfectly with my idea of listening rooms. See the below diagram to illustrate this distinction.



Multicasting (source: Wikipedia)



Broadcasting (source: Wikipedia)

It helped to have a better idea of what I was looking for, but I still did not quite know how to achieve this. With some more looking, I found that Java has a class called MulticastSocket. This class uses UDP for the data transmission, which sounded like it would do the trick for audio transmission. However, Java out of the box does not have much in the way of audio encoding to optimize it for streaming transmission, so there are not a ton of resources or tutorials available showing a MulticastSocket applied as a streaming tool. I did some searching on GitHub and found a project where someone created a basic audio stream with Java sockets, but it was extremely choppy when I ran it and it convinced me to start looking for other options.

My next approach was to do some research on what the tech stack looks like for companies that already use this technology such as Spotify and Google. Spotify's engineering page and various scholarly papers studying Spotify's network spoke at length about their P2P (peer to peer) system, which is done over TCP. This sent me spinning back to square one since I had just done a deep dive into GitHub and StackOverflow in search of resources for UDP streaming. I also ran into an old Google patent outlining a real-time audio and video streaming system architecture. My conclusion from these findings was that the type of streaming I was looking for exists all over the place, but the companies that apply it at scale often have large custom systems that fit their specific needs.

Since I was starting to realize the limitations of stock interfaces and classes for languages like Java and Python, I changed directions and started looking for pre-made libraries that could help me out. Two common streaming file formats that are compatible across many browsers are MPEG-DASH and HLS (HTTP Live Streaming). What if libraries existed that could take my

audio files, encode them, and distribute them for me? An open-source library called “libdash” popped out to me and seemed like it could fit the bill. I quickly ran into barriers due to the minimal amount of documentation, no code support for the last couple of years, and an old Visual Studio compiler. After some more digging, it turned out that the company that maintained the library, Bitmovin, saw a business opportunity and shifted over to a paid model for audio encoding. The pricing is steep - $5499/year.

At this point, I was feeling very frustrated and did not have much to show for my research other than a long list of mildly helpful bookmarks in my browser. I made the decision to shift my time away from endless research and start tackling the project requirements that I did have a better grasp of.

Instead of streaming the audio, I decided to just send the whole audio file over HTTP because it is quick, simple, and easy to get an implementation going fast. Sending the data to the client in this fashion is similar to how files get downloaded over the web. Adding the “Content-Disposition: inline” header to the response with the audio file indicates that the file is meant to be displayed within the webpage rather than downloaded.

```
@GetMapping("/now-playing")
fun getRoomNowPlaying(@RequestParam("roomId") roomId: Int):
ResponseEntity<StreamingResponseBody> {
    val file: File = roomService.getCurrentTrack(roomId) ?: return
ResponseEntity.noContent().build()
    val responseBody = StreamingResponseBody { outputStream: OutputStream ->
Files.copy(file.toPath(), outputStream) }
    return ResponseEntity.ok()
        .header(HttpHeaders.CONTENT_DISPOSITION, "inline")
        .contentType(MediaType.asMediaType(MimeType("audio", "mpeg")))
        .body(responseBody)
}
```

This definitely was not exactly what I was imagining when I planned the project and does not entirely meet my requirements for multicasting, but I believe that it still fits the spirit of the project. Clients within a listening room are all still listening to the same music, despite there being no guarantee of being at the same point of the song with no synchronized play/pause functionality. Figuring out how to send real-time streams of data still fascinates me, but it ended up being outside of the scope of the project for me. Working around this technical problem was a huge challenge, and I am very relieved that the solution did not compromise the initial goal of collectively sharing the joy of music while being apart.

Conclusion

I am incredibly happy and proud about how this project turned out. Lately, I have been pulling it up on my phone and listening to music while I am cooking or cleaning dishes in the kitchen. I had some prior experience in Spring Boot and React prior to this project, but spending this much time and energy working closely with these technologies has given me a detailed familiarity with how they work that will be benefitting me in my full-time web development job after graduation. I believe that I should receive an A on this project. Streaming was not quite achieved and there’s an endless list of things that I could do to improve the stability/performance of the application, but the effort that I put into it stretched me as a developer, taught me how to manage my time when facing a large scary project, and created a product that I am excited to have on my portfolio.

Rubric		
Requirement	**Points Earned**	**Points Available**
Backend		
Unit Testing on controllers using Spock	5	5
File organization consistent with Spring Boot standards	2	2
Backend can index .mp3 and .ogg files that are available in a root directory	10	10
Backend can recursively index .mp3 and .ogg files that are nested (ex. [artist_name] > [album] > song.mp3)	3	3
Endpoint to retrieve listing of available song titles	8	8
Endpoint to provide current song information for a room	5	5
Users can create a room and generate a room code	10	10
Users can join a room by room code	5	5
A room can play a single predetermined song to multiple clients (multicasting)	10	15
A room can play a predetermined playlist of music	5	5
Clients within the room can play and pause	10	10
Clients within the room can skip to the next song	5	5
Clients within the room can add to queue for that specific room	5	5
Count the number of clients within the room	0	2
Frontend		
Create/join room landing page	5	5
Successfully can create or join a room with API calls	15	15
Consistent look/feel	5	5
User can play/pause	10	10
User can skip	5	5
User can see a listing of available music to add to queue	3	3
User can add to queue	2	2
Audio plays on the client-side	10	10
Audio stream is synchronized for multiple clients	0	15
Display current track information	5	5
Queue visible in UI	5	5
TOTAL	148	170

Grading Scale	
Grade	**Range**
A	130-160
B	110-129
C	95-109
D	80-94
F	< 90