Jam TUgether - Documentation

Tobias Holper, Philipp Pirlet, Yonatan Demissie,
Andre Bonnekoh, Jarl Soerensen, Linus Segeth,
Tomer Yavor, Anne Gabler, Lennart Wenke

February 22, 2021

Contents

1 Introduction

This is Jam TUgether: A client-server application that lets you make with other people on your android phone.

2 Client

2.1 Dependency Injection

The app uses the `Dagger` library for dependency injection.
Once `Dagger` knows how to instantiate an object A, we can instantiate any object that is dependent on A without instantiating object A explicitly.
The app mostly uses dependency injection for objects that are singletons and dependent on other objects.

2.2 Logging

In order to keep logging very simple and structured, the app uses the Open-Source library `Timber`. One of the many things that make `Timber` better than the default android logger is the fact that a logged message contains the class name in which it was logged. This makes log filtering very easy. Also, `Timber` allows us to add our custom prefix to logs in order to separate our own logs from system logs.

2.3 Data storage

The app uses the interface `SharedPreferences` in order to store data locally meaning that it's still there after the user closes the app.
`SharedPreferences` uses a simple key value system to retrieve and store data. The app uses `SharedPreferences` in order to store the main instrument of the user or to keep track whether a user has finished the app tutorial.

2.4 API

The app uses the `Retrofit` library as an HTTP client. `Retrofit` allows us to implement API endpoints with their according URL and parameters in a very simple and short way through interfaces. These interfaces are then being injected (with `Dagger`) into our repository objects (`SoundtrackRepository` and `RoomRepository`) which are responsible for making the actual HTTP requests by using those interfaces. `Retrofit` is also responsible for parsing objects to and from JSON. The app uses a custom callback that extends `Retrofit`'s existing callback class. It allows us to either get the response we were looking for (in `onSuccess()`) or to get an error that was generated based on the HTTP status code (in `onError()`)

2.5 UI

The app uses the Model-View-ViewModel Pattern.
The model is being represented by `SoundtrackRepository` and `RoomRepository`. The View is being represented by fragments. The app uses fragments to represent either a part of a screen or a complete screen. Since we are using MVVM, a fragment always has a corresponding view model. While the fragment is responsible for displaying UI components and their data, a view model is responsible for providing the data that is displayed as well as handling UI events that are triggered by UI components. Most of the data in our model (i.e a list of soundtracks) is then being delegated to our view models.
The app uses the `LiveData` library in order to represent variables that contain data that can change (i.e. a list of soundtracks). `LiveData` provides methods for registering observers which are then called automatically when the value of the `LiveData` object changes. Observers can be registered together with a `LifeCycleOwner` (a component with a life cycle). If so, the observers will be unregistered automatically as soon as the `LifeCycleOwner` is destroyed.
The app uses data binding in order bind the data variables of the view model directly to the UI components of the fragment's xml layout file. If the bound variable is a `LiveData` object, the UI component is being updated automatically without having to explicitly register it as an observer. Data binding also allows us to bind click listeners directly to UI components of the xml file. This structure therefore highly reduces the need to reference UI components in the fragment making the code easier to read, shorter, and cleaner.

Important to note: The client's abstract `Soundtrack` model class does not represent the server's "Soundtrack" model but rather a soundtrack from the user's point of view (single or composite). The server's "Soundtrack" model is being represented by the `SingleSoundtrack` model class.
The `CompositeSoundtrack` model represents a collection of multiple soundtracks. In order to represent them in the same way UI-wise, they both extend from `Soundtrack`.

2.6 Audio

The app uses the `SoundPool` class for storing and playing audio files (i.e. .wav files). It allows us to play audio files without having to write a lot of code.
The app uses different `SoundPool` classes for each instrument and the metronome because they each contain different audio files.
`SoundPool` also allows us to limit the number of sounds that can be played with one `SoundPool` object.

2.7 Soundtrack Player

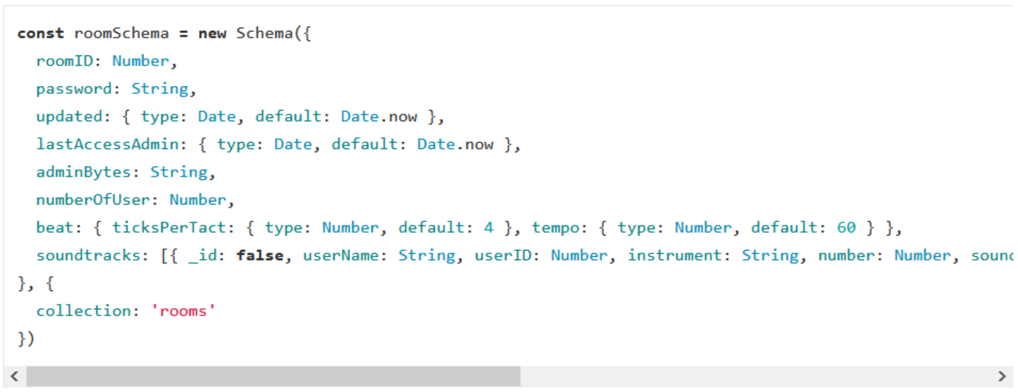
The app uses a custom soundtrack player in order to support our soundtrack data structure. While the `SingleSoundtrackPlayer` is responsible for playing

one soundtrack, the `CompositeSoundtrackPlayer` is responsible for playing a composite soundtrack.
These players use a thread to play a soundtrack because it allows them to implement a while loop that executes functions every millisecond. This thread was implemented through the custom class `SoundtrackPlayingThread` which extends from `Thread`.
This object is responsible for playing the soundtrack it was assigned to and therefore has to keep track of the soundtrack's current progress (in milliseconds). While the while loop is still running, the `SoundtrackPlayingThread` object checks every millisecond if its soundtrack contains a sound that needs to be started, resumed or stopped according to the current progress and acts accordingly. Both the `SingleSoundtrackPlayer` and the `CompositeSoundtrackPlayer` keep track of which thread is assigned to which soundtrack. This is necessary because each thread is only used for one specific soundtrack in order to keep track of each soundtrack's progress (in milliseconds) separately.

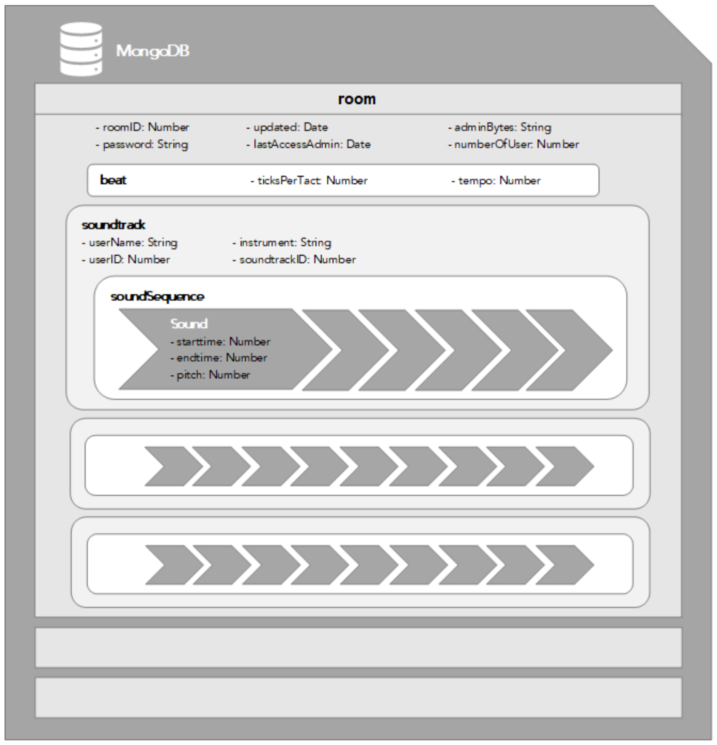
3 Server

3.1 Data structure

Jam TUgether has its own data structure for the transmission and storage of individual tones as well as entire compositions.
The client sends a ”sound”-object to the server for every note played. This includes the start time and end time of the sound, as well as the corresponding pitch.
From these individual sounds, a ”soundSequence”-object is built together on the server, which is located in a ”soundtrack”-object with further information about the instrument and the user.
Several soundtracks then result in a ”composition” which summarizes all the soundtracks of a room, contains information about the room (roomID, password, information about the admin, ...) and is sent from the server to the client on request.
Every room contains also a ”beat”-object, in which tempo and ticks per tact for the metronome are stored.



```
const roomSchema = new Schema({
  roomID: Number,
  password: String,
  updated: { type: Date, default: Date.now },
  lastAccessAdmin: { type: Date, default: Date.now },
  adminBytes: String,
  numberOfUser: Number,
  beat: { ticksPerTact: { type: Number, default: 4 }, tempo: { type: Number, default: 60 } },
  soundtracks: [{ _id: false, userName: String, userID: Number, instrument: String, number: Number, sounc
}, {
  collection: 'rooms'
})
```



3.2 Database

Jam TUgether uses MongoDB for saving rooms and their data. Every room password is hashed with bcrypt before it is stored in the data base.
After an admin inactivity of more than 1 minute the server choose a new admin from the active users of the room. In case the last activity in the room was before more than 30 minutes the server will close the room. All containing data is deleted.

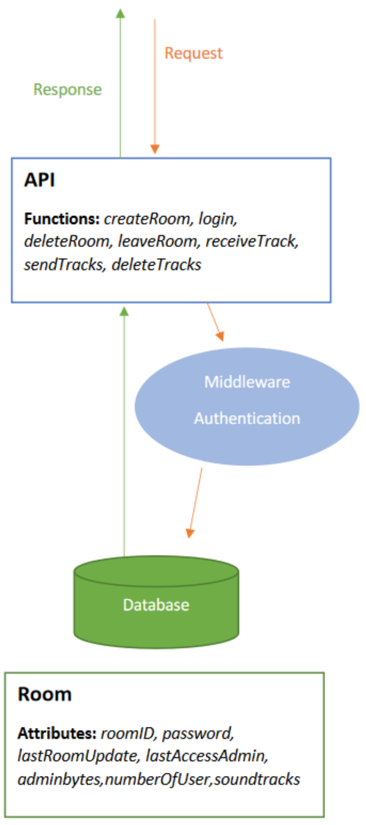
3.3 API

Users interact with the server by http requests to certain api-endpoints. Jam TUgether realizes this endpoints with express. On our swaggerUI you can see all api endpoints and how to send request to them. The client-server communication bases on https.

Important to note: *The client sends its password in plain text to the server. This is a known security issue which is not yet addressed by Jam TUgether.*

3.4 Security

Every request to operate with a room's data needs to be authorized by tokens. For this purposes Jam TUgether uses JWT.
The following figure is visualizing a client request of an existing room.


Request
Response
API
Functions: createRoom, login, deleteRoom, leaveRoom, receiveTrack, sendTracks, deleteTracks
Middleware
Authentication
Database
Room
Attributes: roomID, password, lastRoomUpdate, lastAccessAdmin, adminbytes,numberOfUser,soundtracks

The token's payload contains user roles e.g. "admin" and random unique bytes. The in use encryption scheme is by default "SHA-256" and can be changed. The secret key is contained in the ".env" - file which is created by "configure-script.js". The secret key has a size of 16 random bytes and will be changed each time the user starts to configure the server. Via the token the server verifies if a user is an admin or not.

4 Further reading

If you are interested to read our user guide please check our wiki.