CHALMERS
UNIVERSITY OF TECHNOLOGY

# A song voting system for Spotify

## A web application bringing democracy to the playback using Spotify's Web API

Bachelor's thesis in Computer Science and Engineering

DUANE IRVIN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY / UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Bachelor's thesis 2020

# A song voting system for Spotify

A web application bringing democracy
to the playback using Spotify's Web API

Duane Irvin


CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology / University of Gothenburg
Gothenburg, Sweden 2020

**A song voting system for Spotify**
A web application bringing democracy
to the playback using Spotify's Web API
Duane Irvin



Examiner: Jonas Duregård, Department of Computer Science and Engineering
Supervisor: Joachim von Hacht, Department of Computer Science and Engineering

Bachelor's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology / University of Gothenburg
SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

**A song voting system for Spotify**
A web application bringing democracy
to the playback using Spotify's Web API
Duane Irvin
Department of Computer Science and Engineering
Chalmers University of Technology / University of Gothenburg

# Abstract

At social gatherings we often want to play music. With digitalization we are able to access almost any music in the world using music streaming providers, such as Spotify. Problems arise when many attendees at a same gathering want to choose what music will be played. Typically, one attendee at the time gets access to the playback device, usually somebody's personal smartphone, to choose music. This is both a privacy issue and a suboptimal solution to allow many attendees to choose what songs to be played next.

This thesis aims to solve this issue by developing a software solution which allows attendees to collboratively choose music without neither getting access to the playback device, downloading apps nor creating accounts.

A design for a browser-based song voting application is presented in this thesis, which allows an attendee to search and vote for songs to be played where the song with the most number of votes is the next song to be played. The design is then implemented taking advantage of Spotify's Web API which allows software to remotely control what music is played on a device running the Spotify app.

Keywords: React, Redux, TypeScript, Go, Spotify, Music streaming, Web API, Voting, Collaborative playlists

# Acknowledgements

I would like extend my sincere thanks to Joachim von Hacht his help writing this thesis. Although not directly I involved, I would also thank my development team at Opera Software AS for helping me grow in every aspect of software engineering. And last but not least, my deepest thanks to my worldwide slackline family that makes every day of my life better.

Duane Irvin, Gothenburg, June 2020

# Contents

# 1
# Introduction

## 1.1 Background

At social gatherings we often want to play music. With digitalization we are able to access almost any music in the world without having it saved locally on the device that is playing the music. Typically, a smartphone or a computer is used to play music from the internet through a music streaming service, either by using an app or through an internet browser.

One of the largest providers of music streaming today is Spotify. They have approximately 271 million monthly users and a large selection of 50 million soundtracks to choose from [1].

Like most other music players Spotify has playlists, which simply are collections of songs. The playlist is often created by the user. When the user chooses to play a playlist, only songs in that playlist will be played. The songs will normally be played in the order they were added or in a shuffled order, depending on the user's choice. In Spotify, a playlist can be made collaborative. By sharing a link to the playlist, anyone with a Spotify account who has the link can add, reorder and remove songs in the playlist.

In addition to using playlists, the user can manually "Add song to Queue" from a song menu (see Figure 1.1a). When adding a song to the queue, this song will be played before the upcoming songs from the currently playing playlist. If there are no queued songs, Spotify will continue playing from songs in the playlist as seen in Figure 1.1b.

A common setup to play music at social gatherings is to have a smartphone running Spotify connected to the speakers. There are then two ways attendees can be part of choosing music:

(1) By using Spotify's collaborative playlists attendees can add songs to the playlist the host has decided to play ahead of time.

(2) Attendees can use the playback device running Spotify to search for songs and then add them to the playback queue.

Sometimes a combination of the two methods above is used. Still, there are some problems with the existing methods. For (1):

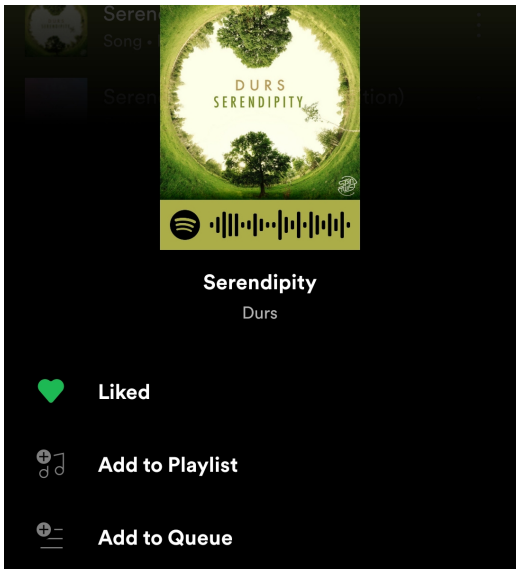
- A Spotify account is required to add songs to the collaborative playlist.
- The attendee needs to get access to the collaborative playlist.

- Attendees tend to make up their minds of what they want to hear on the fly. Playing songs from a playlist in a random order or the order they were added is therefore usually not good enough. For big playlists some songs may not even have time to be played at all.

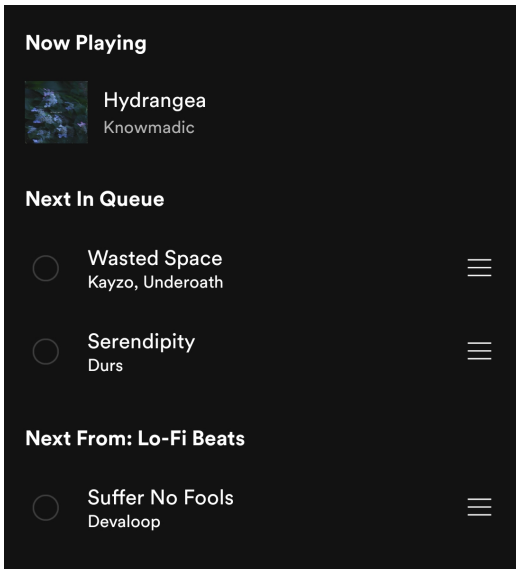
For (2):

- The attendee needs access to the playback device.
- The attendee gets full access to stop the music, immediately change song and change the order of the already queued songs which can lead to conflicts.
- The attendee gets access the playback device in an unlocked state, which is a privacy issue for the owner.

In either case there is also no way for attendees to see a history of what songs that were actually played nor when the songs were played.




**(a)** Song menu.




**(b)** Playback queue after some songs have been added to the queue. "Lo-Fi Beats" is the currently playing playlist.

**Figure 1.1:** Shows what it looks like in the Spotify app when songs are added to queue.

## 1.2 Purpose

This project aims to reduce the threshold for attendees at social gatherings to be part of choosing what music is going to be played on Spotify while removing the need for them to access the playback device or having a Spotify account.

## 1.3 Objectives

A software will be developed that will:

- Remove the need for attendees to have a Spotify account.
- Remove the inconvieniences caused by attendees having to get access to the collaborative playlists or the playback device.
- Remove the ability for attendees directly influence the playback, e.g., to stop the music, immediately change song and change the order of the playback.
- Remove the privacy issue for the owner of the playback device caused by attendees getting access to it in an unlocked state.
- Bring democracy the order of which songs are played.

The software should not require an account nor to be downloaded by the attendee.

## 1.4 Delimitations

Although the same or similar problems exist for other music providers, Spotify is the provider we will focus on in this thesis.

## 1.5 Method

The implementation of the software will go through the following phases:

**User Stories** — A set of user stories will be defined. These are descriptions of all different things the users will be able to do with the software and what will happen from the user perspective was defined. All of our objectives are directly related to the user experience, so this naturally becomes the first step of the development process.

**UX Design** — Wireframes will be used design a user interface which satisfies all the previously defined user stories. The UX design is what the attendee will be interacting with in the web application.

**Architecture** — A software architecture was designed such that all the user stories could be fulfilled.

**Development** — The software will be developed according to the software architecture and the UX design.

**Testing** — The software will manually be tested to confirm the developed sofware fulfills the previously defined user stories.

# 2

# Technical Background

## 2.1 Spotify Web API and Account Services

Spotify has an extensive Web API which allows developers to create applications that access information about all soundtracks available on Spotify, access information about a user and perform a variety of tasks on behalf of the user and more [2]. To use Spotify's Web API in an app, the app must be authorized by Spotify to do so. There are two ways to do this:

**App Authorization** — Spotify authorizes the app to use Spotify Web API. This is a server-to-server authorization which is used to gain access to *unscoped data*. Unscoped data includes things like searching for songs and albums. It does not involve any user resources.

**User Authorization** — Spotify and a user authorizes the app to access and/or modify the user's data through Spotify Web API. This method uses OAuth2[1] to allow the user to log in and agree to a list permission scopes. This is used whenever *scoped data* is requested, such as accessing information about a user, modifying the user's playlists or controlling the user's playback.

Any authorization is made through Spotify Account Services, which is a separate service from Spotify Web API.

### 2.1.1 Register a Spotify Application

For any of the authorization methods, an application that uses Spotify's Web API must first be registered on Spotify Developer Dashboard. The developer will receive a *Client ID* and a *Client Secret*, and be able to register *Redirect URIs* [4].

**Client ID** — is a unique identifier that is used to identify the application. The client ID is not a secret and can safely be used by end-user clients.

**Client Secret** — is a key used to make secure calls to the Web API. This key is secret and should not be used by end-user clients as it would risk getting leaked. Generally it is used by servers communicating with Spotify's Web API.

**Redirect URI** — is whitelisted address which the user can be redirected to after user authorization using OAuth2. Redirection to non-whitelisted URIs is forbidden.

[1] OAuth 2 is an authorization framework which enables a third-party application to obtain limited access to an HTTP service [3].

### 2.1.2 Client Credentials Flow

Client credentials flow is used for app authorization [5], shown in Figure 2.1:

(1) The application uses Client ID and Client Secret to request authorization to Spotify Account Services.

(2) Spotify Account Services responds with an `access token`.

(3) The `access token` is included in any request of unscoped data from Spotify Web API.

(4) Spotify Web API responds with a JSON object containing the unscoped data.

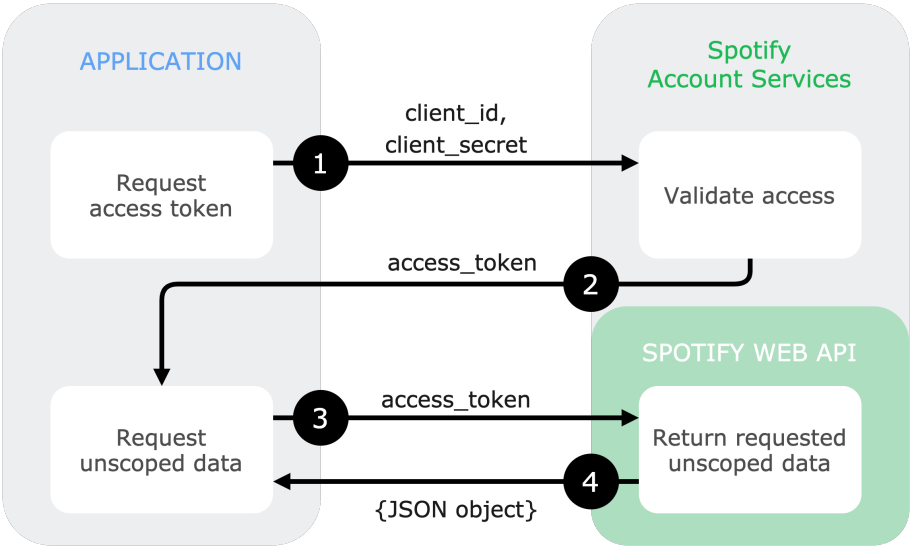



**Figure 2.1:** Visualizes the client credentials flow for Spotify Web API.

### 2.1.3 Authorization Code and Permission Scopes

Authorization code is used for user authorization [5]. The application redirects the user to Spotify Account Services, where the user will login and agree to a list of permission scopes so the application can request resources on behalf of the user. The user then gets redirected to a provided redirect URI, which as explained in subsection 2.1.1 must have been whitelisted.

**Permission scope** — is a defined scope of user resources that the application can access, but nothing more [6]. For example the permission scope `user-modify-playback-state` is described as "Write access to a user's playback state" and is required by endpoints like [6]:

- Pause a user's playback
- Set volume for user's playback
- Skip user's playback to next track
- Add an item to the end of user's current playback queue

These are the steps used by authorization code [5], shown in Figure 2.2:

1. The application redirects the user to Spotify Account Service. Client ID, permission scopes, a `state-string` and redirect URI are embedded as query parameters in the URL which the user is redirected to.
2. The user logs in (if needed) and agrees to the permission scope.
3. The user is redirected to the redirect URI. In the redirect URI, a `code` and the `state-string` are embedded as a query parameter. The `state-string` is optional, but can be useful to correlate step 1 with this step, i.e., identify which redirect from Spotify is the result of what authorization request by the application.
4. The `code`, Client ID, Client Secret and redirect URI are used by the application to request an `access token` and a `refresh token` from Spotify Account Services.
5. The `access token` is provided in any request of scoped data from Spotify Web API.
6. When the `access token` expires, the `refresh token` is sent to Spotify Account Services to retrieve a new `access token` for use in step 5.

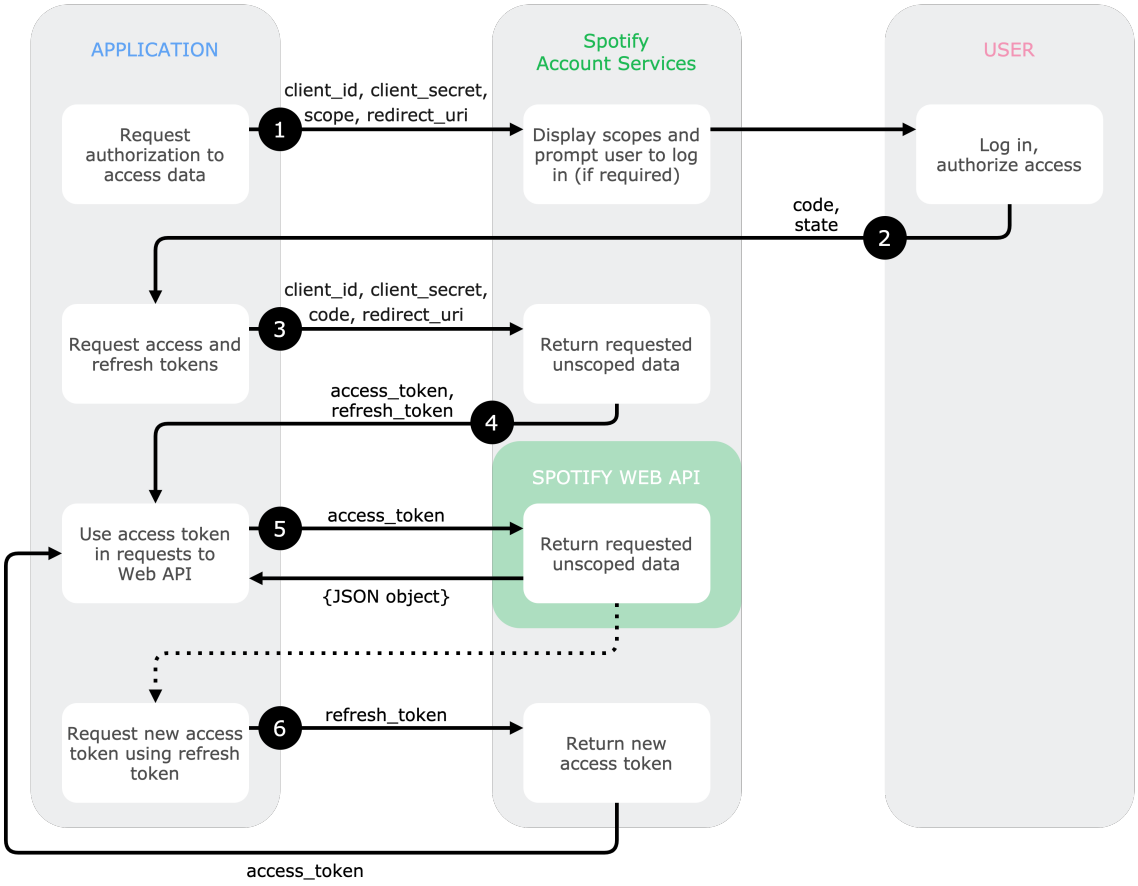



**Figure 2.2:** Visualizes the code authorization flow for Spotify Web API.

### 2.1.4 Track Objects

In Spotify's Web API a song is referred to as a `tracks`. It can have two different shapes: `full` or `simple`. A `simple track` contains a subset of the information of its eqiuvalent `full track`. Below are the parts of a `full track` that are relevant for this paper. For the `simple track`, the `album` field would be missing [2]:

```
{
  "id": "<a unique identifier>",
  "type": "track",
  "uri": "spotify:track:<the id above>",
  "name": "Wake Me Up",
  "artists": [{ "name": "Avicii" }],
  "album": {
    "name": "True",
    "images": [{ "height": 640, "width": 640, "url": "<an url to the album cover>" }]
  },
  "duration_ms": 247426
}
```

### 2.1.5 Playback context

A playback context is something from which soundtracks are played. The context can for example be an artist, a playlist or an album. If the playback context is an album, that means upcoming songs will be chosen from that album.

### 2.1.6 Spotify Web API Endpoints

The endpoints from Spotify Web API that are relevant for this paper are listed below [2]. Many parts that are irrelevant in this paper have been omitted. The host address `https://api.spotify.com` has been omitted from all endpoint URLs.

**Search — Search for an item**

Get Spotify Catalog information about albums, artists, playlists, tracks, show or episodes that match a keyword string.

**Endpoint**

`GET /v1/search`

**Permission scope**

None — can be used with app authorization.

**Query parameters**

`q` — Search query keywords and optional field filters and operators.

`type` — A comma-separated list of item types to search across. Valid types are: `album`, `artist`, `playlist`, `track`, `show` and `episode`. Search results include hits from all the specified item types.

**Response structure**

```
{
  "artists": { "items": [{ "type": "artist", ... }] },
  "playlists": { "items": [{ "type": "playlist", ... }] },
  "tracks": { "items": [{ "type": "track", ... }] }
}
```

### Player — Get the user's currently playing track

Get the object currently being played on the user's Spotify account.

**Endpoint**

`GET /v1/me/player/currently-playing`

**Permission scope**

`user-read-currently-playing` and/or `user-read-playback-state`

**Response structure**

```
{
  "is_playing": true,
  "item": { /* a full track or full episode */},
  "progress_ms": 44272
}
```

### Player — Add an item to the user's playback queue

Add an item to the end of the user's current playback queue.

**Endpoint**

`POST /v1/me/player/queue`

**Permission scope**

`user-modify-playback-state`

**Body parameters**

`uri` — The uri of the item to add to the queue. Must be a track or an episode uri.

### Player — Get current user's recently played tracks

Get tracks from the current user's recently played tracks.

**Endpoint**

`GET /v1/me/player/recently-played`

**Permission scope**

`user-read-recently-played`

**Response structure**

```
{
  "items": [
    {
      "track": { /* A simple track */},
      "played_at": "2019-12-13T20:44:04.589Z"
    }
  ]
}
```

## 2.2 Go

Go is an open-source, statically typed, compiled programming language designed at Google [7]. Its syntax is similar to C but includes memory safety, stack management and garbage collection [8]. Go supports an object-oriented programming style but is not object-oriented per se [8]. It is lacking type hierarchy/inheritance, generics and many other features found in other programming languages [8]. The lack of features however makes the language itself less complex. Many features, has been suggested and designed but finally rejected because various reasons. One of which being generics [9].

## 2.3 zmb3/spotify — Spotify Web API Wrapper

The Go library zmb3/spotify is a wrapper for working with Spotify's Web API. It aims to support every task listed in the Spotify Web API endpoint reference [10]. Among other things, it includes types for all API responses and manages access tokens and other headers in API requests.

The type Authenticator from zmb3/spotify provides convenience functions for implementing the OAuth2 flow. The type Client is a client for working with the Spotify Web API and can be created from the method Authenticator.NewClient.

This is an example of a client using client credentials flow from subsection 2.1.2:

```
import (
    "context"
    "fmt"

    "github.com/zmb3/spotify"
    "golang.org/x/oauth2/clientcredentials"
)

func clientCredentialsFlow() {
    // Set configurations
    config := &clientcredentials.Config{
        ClientID:     "This is a SPOTIFY ID",
        ClientSecret: "This is a SPOTIFY SECRET",
        TokenURL:     spotify.TokenURL,
    }
    // Get token
    token, _ := config.Token(context.Background())
    // Create a client from an empty Authenticator
    client := spotify.Authenticator{}.NewClient(token)
    // Use the client to search for songs
    result, _ := client.Search("avicii wake me up", spotify.SearchTypeTrack)
    for _, track := range result.Tracks.Tracks {
        fmt.Println(track.String())
    }
}
```

This is an example of a client using authorization code from subsection 2.1.3:

```
import (
    "net/http"

    "github.com/zmb3/spotify"
)

// Create an authenticator with permission scopes and a redirect URI
var authenticator spotify.Authenticator = spotify.NewAuthenticator(
    "http://mysite.com/callback",
    spotify.ScopeUserReadPlaybackState,
)

// Variable for Authorization Code Client
var client spotify.Client

// Authentication Handler
func authenticate(w http.ResponseWriter, r *http.Request) {
    // Redirect the user, he/she will log in at Spotify Account Service,
    // accept the permission scopes and then be redirected back to the redirectURI
    url := authenticator.AuthURL("state-string")
    http.Redirect(w, r, url, http.StatusFound)
}

// Callback Handler
func callback(w http.ResponseWriter, r *http.Request) {
    // Get access token and refresh token from code
    token, _ := authenticator.Token("state-string", r)
    // Create client that is using Authorization Code
    client = authenticator.NewClient(token)
}

func queueSong(id spotify.ID) {
    // Use the client to Queue a Song
    _ = client.QueueSong(id)
}
```

## 2.4 TypeScript, Node.js and npm

**TypeScript** — is an open-source language maintained by Microsoft. It is a superset of JavaScript which adds optional static typing to the language. It compiles into clean JavaScript [11].

**Node.js** — is an open-source and cross-platform JavaScript runtime environment. It runs V8, the same JavaScript engine as Google Chrome [12].

**npm** — is a package manager for Node.js which is used to install packages from the public npm registry [13]. Its configuration is stored in the project source folder as a JSON object in a file named package.json [14]. All dependencies, run scripts and build scripts are typically defined in this file. While npm is used to install and manage dependencies, npx is used to execute command line tools from the same registry [15]. npx is included when installing npm.

## 2.5 React and create-react-app

**React** — is a JavaScript library for building user interfaces maintained by Facebook [16]. It encapsulates declarative UI components that manage their own state and renders its own HTML. When the application state changes, only the components that have changed will rerender. The developer does not need to manipulate the HTML DOM directly and the components can easily be tested with unit-tests.

**create-react-app** — is a tool used to create an initial React project with all required dependencies and a set of useful developer scripts for running developer builds, testing components and compiling optimized production builds [17]. create-react-app is available through npm. To set up a React project using TypeScript, you can run the command `npx create-react-app <app name> -template typescript`.

## 2.6 Redux with React

Redux is a state container for JavaScript apps. It allows the developer to keep the entire application state in a single object and let changes to the state happen only through defined actions. Actions can be recorded and replayed and should always produce the same state, which makes state management predictable and testable. This helps writing applications that behave consistently while improving testability and making the application easier to debug [18].

The Redux *store* holds the entire application *state*. The state can only be changed by *dispatching* an *action* using an *action creator*. The new state is computed by a *reducer*. Figure 2.3 shows an example of how React and Redux can work together, which will be described below:

The React component receives the state from the store. It uses whatever parts of the state that are relevant for its HTML to render, in this case an integer named count. count is rendered in Count View.

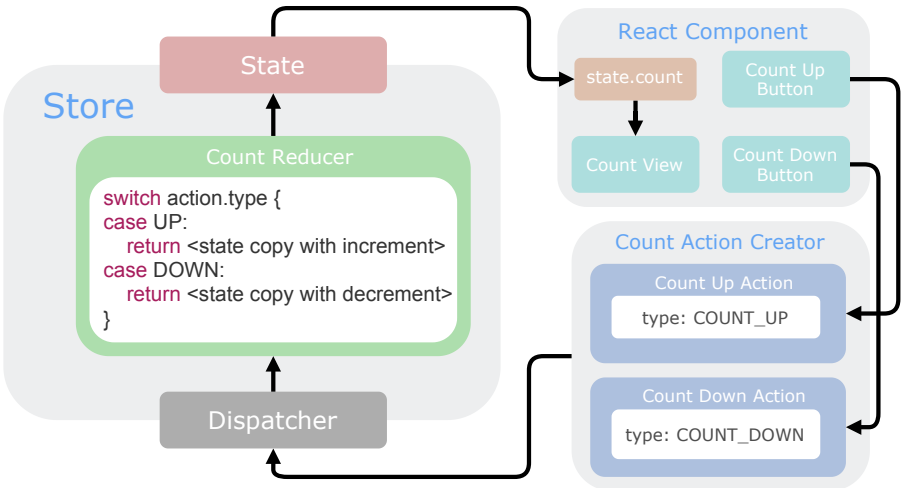



**Figure 2.3:** Illustrates a React component which gets its state from, and dispatches actions to, a Redux store.

The `React` component also renders two buttons, `Count Up Button` and `Count Down Button`. When a button is tapped, a corresponding action is created by an action creator. The action is sent to the store's dispatcher. The dispatcher sends the action to the reducer for a new state to be computed. The reducer creates a new state, where only the relevant parts are changed. In this case, `count` has been incremented or decremented, depending on the action's type.

The new state is sent to the `React` component. If needed, the `React` component will render new `HTML` code.

For more complex applications:

- The action creator can create asynchronous tasks and make multiple dispatches. This is useful for showing loading indicators when making API calls and stopping the indicator when the API call finishes.
- A *payload* can be included in the action. The payload can include any kind of information that needs to be available for the reducer to update the state, for example the result of an API call.
- The dispatcher can be wrapped with middlewares. For example providing a dispatch queue for asynchronous tasks so that the UI does not freeze when making API calls or the state does not become susceptible to race conditions when dispatching multiple actions simultaneously.

# 3

# Implementation

## 3.1 User stories and requirements

As discussed in section 1.1 one of the problems with playlists is the order in which songs are played. To emphasize the importance of the order in which the music is played, it was decided to give the attendees the possibility to vote for songs to be played, such that a song with many votes is played earlier. With this in mind, the following user stories was defined to satisfy the objective:

| User Story | Description |
|---|---|
| **US1-Search** | As an attendee, I want to type the name of a song or an artist to see a list of matching songs on Spotify. I then want to be able to select a song from the list to suggest it for playback. |
| **US2-Vote** | As an attendee, I want to see a list of songs that I or others have suggested for playback. For songs that others have suggested, I want to be able to vote for them to be played next. |
| **US3-Playback** | As an attendee, I want to see the name and artist of the song that is currently playing. I also want to see the duration and progress of the song. |
| **US4-History** | As an attendee, I want to be able to see what songs have previously been played and at what time. |
| **US5-Queue** | As a host, whenever a song ends, I want the song with the most number of votes to start playing from my device. |

From the user stories above, the following functional requirements was formed:

| Functional Requirement | Description |
|---|---|
| **FR1-Search** | Attendees can search for music available on Spotify. |
| **FR2-InitialVote** | Attendees can vote for songs from Spotify to be played. |
| **FR3-NonInitialVote** | Attendees can vote for songs that others have voted for to be played. |
| **FR4-VoteBlock** | Attendees can only vote once for each song. |
| **FR5-VoteWin** | The song with the most number of votes should be the next song to be played. |
| **FR6-CurrentPlayback** | Attendees can see the name, artist and progress of the currently playing song. |
| **FR7-History** | Attendees can see a history of songs that have been played and what time each song was played. |
| **FR8-PlaybackProtection** | Attendees can not stop the music, skip songs nor modify the order in which songs will be played. |

Given that the attendee should not have to download the software and most likely will use it from a smartphone, it becomes natural to develop a web application such that the software is available through a browser via smartphone. The following non-functional requirements was defined:

| Non-Functional Requirement | Description |
|---|---|
| **NF1-Browser** | The web application should be accessible through the web browser. |
| **NF2-Smartphone** | The web application should be adapted for usage on small screens (e.g., smartphones). |

To reduce the scope of the implementation, it was given the following delimitations:

| Delimitation | Description |
|---|---|
| **DE1-OneHost** | Will only allow one host account to be used, although we will assume that the software will be improved to allow any number of host accounts in the future. |
| **DE2-Smartphone** | Will only take web application behavior when used on smartphones into account. |
| **DE3-Visual** | Will put minimal effort into visual design. |
| **DE4-Security** | Will not protect against users intentionally trying to cheat the system by e.g., gaining extra votes by using multiple browsers. |
| **DE5-Local** | Will be hosted on a local network which all attendees are assumed to be connected to. |

## 3.2 UX Design

This section uses wireframes to provide a UX Design for how the attendee will interact with the web application. The web application consists of three different views: Main, Search and History. The user can navigate between the views as shown in Figure 3.1. The arrows indicates what components are used to navigate between the views.

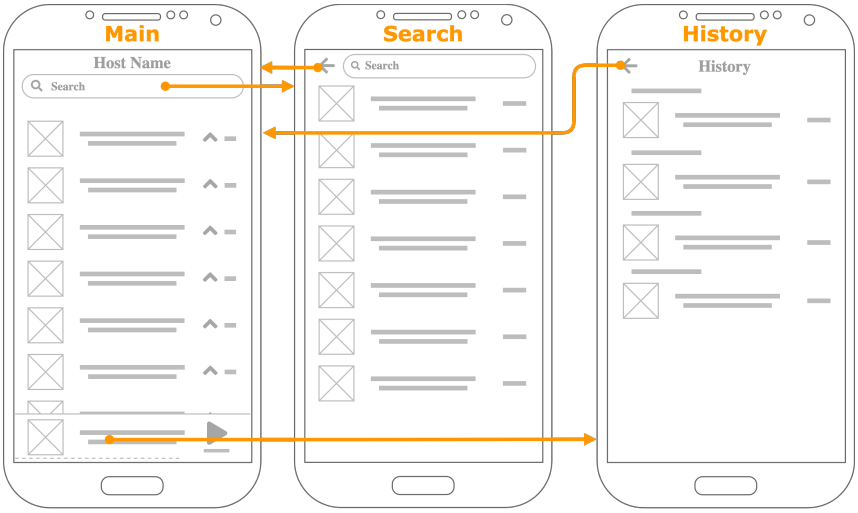



**Figure 3.1:** Shows an overview of the views that the attendee can navigate to.

The main view consists of the following components, shown in Figure 3.2:

1. Host — The name of the host. This element can be ignored, but will be important in the future to differentiate different hosts according to DE1-OneHost.
2. Search bar — Here the attendee can type a search query to search for songs on Spotify as a part of US1-Search. After submitting a query, the Search view is shown.
3. Album cover — An image showing the album cover for a song that has been voted for according to US2-Vote.
4. Song information — Song name and artists for a song that has been voted for according to US2-Vote.
5. Up-vote button — When the attendee tap this button, the song will receive an additional vote according to US2-Vote. If the attendee has already voted the song, the button will change color and become disabled to fulfill FR4-VoteBlock.
6. Number of votes — The number of votes the song has. The list of songs is ordered by this number, with highest number on the top. When a song ends, the song at the top is removed and will start playing according to US5-Queue.

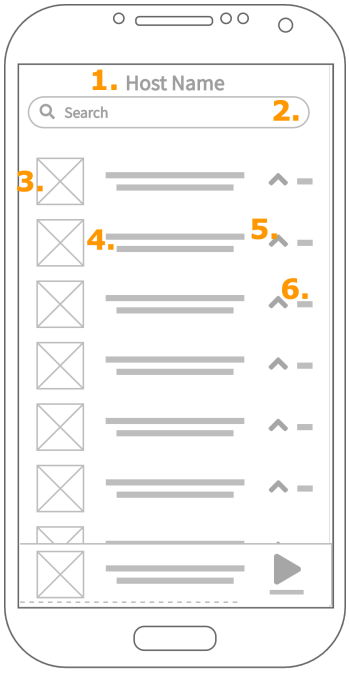



**Figure 3.2:** Shows a wireframe for the different parts of the Main view.

The playback bar in the bottom of the main view shows information about the song that is currently playing according to US3-Playback, The song information has the same format as the main view as seen in Figure 3.3 (3. and 4.). In addition to the song information, it shows an indicator for whether the music is playing or not (1.), the progress and duration of the song (2.) and an animated progress bar which visualizes the proportion of the song that has been played (5.).

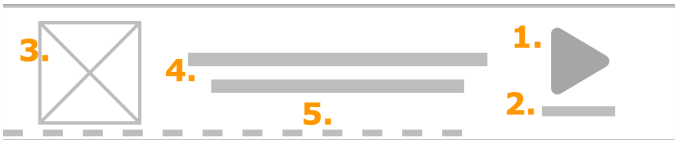
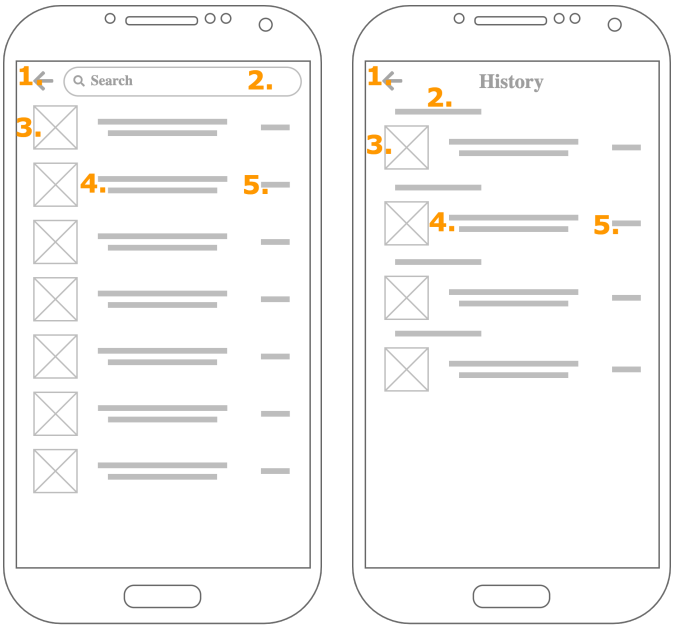



**Figure 3.3:** Shows the playback bar, which is the bottom component of the Main view in Figure 3.2.

The layout of the search and history view is similar to the main view as seen in Figure 3.4. Each row shows a song's name and artists (3. and 4.), album cover (3.) and duration (5.). In both views there is a button in the top left which takes the attendee back to the main view (1.).

In the search view, matching songs from Spotify are listed. The attendee can vote for the song by tapping on it according to US1-Search, which also takes the attendee back to the main view. The attendee can search for something else by typing another query in the search bar (2. in Figure 3.4a).

In the history view, nothing happens if the attendee taps on a song, but there is a timestamp indicating when the song was played (2. in Figure 3.4b) according to US4-History.




**(a)** Search View **(b)** History View

**Figure 3.4:** Shows wireframes for the Search View and the History View.

## 3.3 Architecture

This section presents an overview of the system and how different entities communicate with each other, without going into much technical detail. In section 3.4 we go deeper into the implementation of each entity. The developed software was named Votify, which it will be referred to as from this point.

### 3.3.1 Votify Overview

The system consists of five entities, as shown in Figure 3.5:

**Playback Device** — The device which is playing music from Spotify using the host's Spotify account. From DE1-OneHost, there is only one playback device. Our system is not concerned by any other details about this device.

**Votify Server** — A web server which manages the voting system and all interaction with Spotify, except for login by host.

**Votify Client** — A browser web client which has the UX Design from section 3.2 implemented. The client only communicates with Votify Server via a RESTful API. There can be any number of attendees, thus also any number of Votify Clients.

**Spotify Web API** — Described in section 2.1.

**Spotify Account Services** — Described in section 2.1.

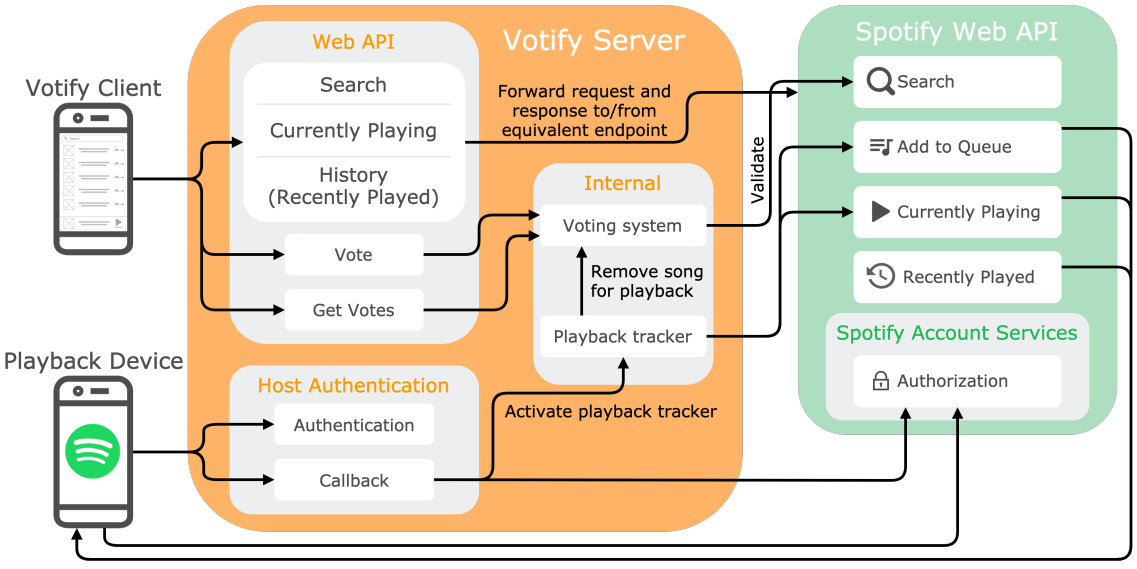



**Figure 3.5:** Visualizes the communication between the Playback Device, Votify Client, Votify Server and Spotify.

The host authenticates using Votify's Authentication endpoint. The host is redirected to Spotify to authorize Votify to perform tasks on behalf of the host as described in subsection 2.1.3. The Callback endpoint is provided as redirect URI. When the host has been redirected to the callback endpoint attendees can start voting for songs using the Votify Client.

The internal parts of Votify consists of a voting system and a playback tracker. The voting system keeps track of what songs have been voted for and how many votes each song has. The playback tracker monitors the playback on the host account. Whenever the currently playing song is ending, the playback tracker will remove the song with the most number of votes from the voting system and add it to the playback queue via Spotify Web API. By adding the song to the queue slightly before the current song is ending the Spotify app on the playback device will make a smooth transition between songs while preserving the current playback context. The playback tracker requires authorization from the host and can therefore only be activated after the host has been redirected to the callback.

The Votify Client will use Votify's Web API to search, get the current playback, get history of played songs, vote for songs to be played and get a list of songs currently voted for. If a Votify Client would request scoped data from Spotify Web API directly it would be required to have the access token provided in the authorization step. This would be unacceptable from a security perspective because anyone with the access token is able to perform tasks on behalf of the host. Therefore, all scoped data must instead be forwarded by the Votify Server.

### 3.3.2 Votify API endpoints

All Votify's API endpoints are presented below. The endpoints are named and grouped based on their use cases the same way they are in Figure 3.5.

**Host Authentication**

**Authentication**

This endpoint is used by the host to authorize Votify to use his account. The host will be redirected to Spotify.

**Endpoint**

`GET /auth`

**Callback**

Spotify will redirect the host to this address after authorization.

**Endpoint**

`GET /callback`

**Query parameters**

`code` — The code used to request an access token and refresh token (described in subsection 2.1.3).

`state` — Allowed parameter, but currently not used. Added for future support of multiple hosts.

**Web API**

The path parameter `username` in the endpoints below is currently ignored but will be used for future support of multiple hosts according to DE1-OneHost.

**Search — Search for tracks on Spotify**

Forwards request to Search endpoint on Spotify. Unlike Spotify's Search endpoint, this endpoint does not allow the `type` parameter. When calling Spotify's Search endpoint, the `type` parameter will be set to `track` by Votify Server. It will then only return tracks (no albums, playlists nor artists).

**Endpoint**

`GET /search`

**Query parameters**

`q` — Search query keywords and optional field filters and operators.

**Currently Playing — Get the song currently playing by the host.**

Fetches and forwards the song currently playing by the host's Spotify account from Spotify Web API.

**Endpoint**

`GET /api/v1/user/{username}/playing`

**History — Get the songs most recently played by the host.**

Fetches and forwards the latest 20 tracks played by the host's Spotify account from Spotify's Web API.

**Endpoint**

`GET /api/v1/user/{username}/history`

**Vote — Vote for a song to be played**

Vote for a song to be played. Will add one vote to the song. If the song is not among the current votes, it will be added.

**Endpoint**

`POST /api/v1/user/{username}/vote`

**Body parameters**

`id` — Spotify track ID for the song to be voted for.

**Get Votes — Get the songs currently voted for**

Get the list of songs that are currently voted for. Will return a list of tracks as described in subsection 2.1.4, but with an additional field `votes` on each track which, contains the number of votes that the song has.

**Endpoint**

`GET /api/v1/user/{username}/votes`

## 3.4 Development

### 3.4.1 Votify Client

The Votify Client was implemented using `TypeScript`, `React`, `Redux` and `Node.js`. It was initiated using `create-react-app` with `TypeScript` as described in section 2.5. See Appendix A for a complete list of dependencies and devices that were used for development and testing.

We will not go into exact details of how each component of the Votify Client was implemented, such as what `HTML`-elements were used or how they were styled using `CSS`, but rather describe the behavior of each component in relation to the wireframes in section 3.2, what state from the `Redux` store they depend on and what actions they dispatch to the `Redux` store. The `Redux` store's reducers will not be discussed unless they implement something non-trivial given the action dispatched.

**The Application state** was split into five different sub-states:

**view** — Which view is currently shown: main, search or history from the wireframe in Figure 3.1 on page 17.

**search** — The result from the latest search as an array of full tracks.

**votes** — The list of songs voted for and how many votes each song has.

**playback** — The currently playing song and its progress on the same format as Spotify's endpoint 'Currently Playing'.

**history** — The most recently played songs and what time they were played on the same format as Spotify's endpoint 'Recently Played'. However, this endpoint used simple tracks and Votify Server will replace these with fulltracks as will be discussed in subsection 3.4.2.

**App component** is the root UI component which embeds all other UI components. It renders one of the three views according to the current view state as shown in Figure 3.6.

```
// Votify Client's state is divided into sub-states
export interface AppState {
  readonly history: HistoryState;
  readonly playback: PlaybackState;
  readonly search: SearchState;
  readonly view: ViewState;
  readonly votes: VotesState;
}
```

```
// The View State can have three different values
export interface ViewState {
  view: ViewTypes;
}

export type ViewTypes =
  | 'HistoryView'
  | 'MainView'
  | 'SearchView'
```

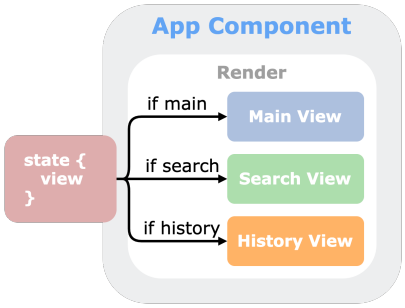



**Figure 3.6:** Illustrates how the `view` part of state is used to decide which view is rendered.

**The Main View** has three components: search bar, votes list and playback bar, shown in Figure 3.7a. After entering a query into the search bar a search action is dispatched. The search action fetches search results from Votify's Search endpoint and simultaneously switches to the search view.

The votes state is used to render the votes list. One row is rendered for each song, each row having an up-vote button according to the Main View design in Figure 3.2 on page 17. When the button is tapped, an action is dispatched which sends the song ID from that row to Votify's Vote endpoint.

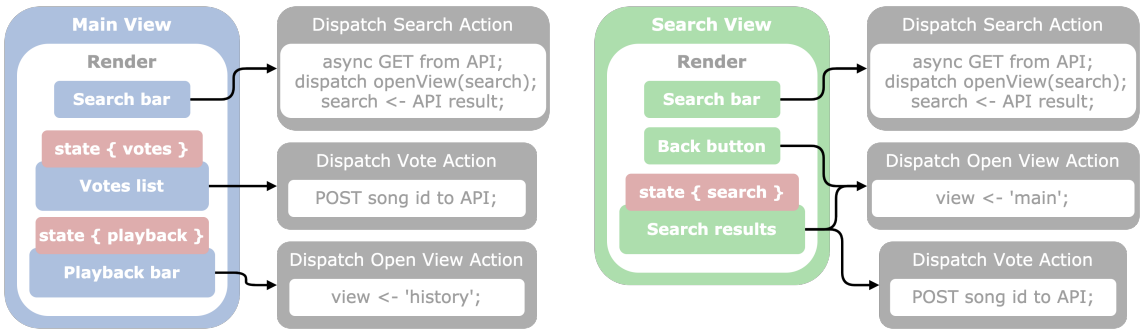
```
export interface VotesState {
  // loading can be used to show an
  // indicator while fetching
  readonly loading: boolean;
  readonly votes: VoteItem[];
}
```

```
export interface VoteItem {
  readonly votes: number;
  // votedFor is set to true by the Client when voting
  readonly votedFor: boolean;
  readonly track: Track; // FullTrack from Spotify
}
```

The playback bar renders the content of the playback state according to the playback bar design in Figure 3.3 on page 18. When tapping anywhere on the playback bar an action to open the history view is dispatched.

**The Search View** has three components: Search bar, back button and search results, shown in Figure 3.7b. The search bar behaves the same as in the main view and when tapping the back button an action to open the main view is dispatched.

The search results from the search state are according the Search View design in Figure 3.4a in on page 18, one row for each track in the search results. When tapping anywhere on a row, an action will be dispatched to upvote that song and one action to open the main view.




(a) Main View (b) Search View

**Figure 3.7:** Illustrates what part of `state` that is used to render each component of `Main View` and `Search View` and what actions they dispatch to the `Redux` store.

**The History View** has two components: Back button and history list. The back button behaves the same as in the Search View and the history list is rendered according to the design in Figure 3.4b on page 18, using the history state.

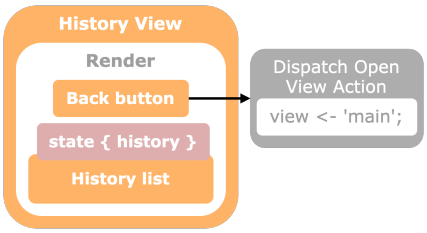



**Figure 3.8:** Illustrates what part of `state` that is used to render each component of `History View` and what actions it dispatches to the `Redux` store.

**The client stays updated** by fetching data in intervals. The interval updater is initialized and makes the first fetch when the `Redux` store is configured. Then it will fetch the votes list, playback and history every 5 seconds. It is fetching by dispatching actions the same way that the UI components are. The redux store is as usual responsible for rerendering affect UI components when the state changes.

```
const intervalUpdater =
  (dispatch: ThunkDispatch<AppState, undefined, AnyAction>) =>
    (interval: number): NodeJS.Timeout => {
      const update = (): void => {
        dispatch(getQueue())
        dispatch(getPlayback())
        dispatch(getHistory())
      }
      update()
      return setInterval(update, interval)
    }

export function configureStore(preloadedState?: AppState): Store {
  const store = createStore(rootReducer, preloadedState)
  intervalUpdater(store.dispatch)(5000)
  return store
}
```

Because the playback state is only updated every 5 seconds by the interval updater, the playback bar makes use of an internal state that updates every 500 milliseconds. Everytime it is updated, it computes the time that has passed since the last playback state update and adds it to the progress time in the UI component. Visually, this makes the progress bar and progress time seem updated in real-time. Everytime a fetch occurs, the internal component state resets to align with the application state.

```
// Playback bar as a React Functional Component
const PlaybackBar: React.FC<Props> = (props: Props) => {
  // Declare an internal state
  const [progressSincePropsSet, setProgressSincePropsSet] = useState(0)
  // Reset internal state when props are updated, i.e., after an API fetch
  useEffect(() => { setProgressSincePropsSet(0) }, [props.progressTime])
  // Update internal state every 500 milliseconds
  useEffect(() => {
    const interval = setInterval(() => {
      setProgressSincePropsSet(progressSincePropsSet + 500)
    }, 500)
    return (): void => clearInterval(interval)
  })
  /* ... create component */
}
```

### 3.4.2 Votify Server

The Votify Server was implemented using `Go1.13` and `zmb3/spotify`. See Appendix A for a complete list of dependencies and devices that were used for development and testing. The components from Figure 3.9 on page 26 are described below:

**Spotify Searcher** is a `zmb3/spotify.Client` instance that uses the client credentials flow instance described in subsection 2.1.2. It is used only for searching for songs in Spotify. It implements an interface accordingly:

```
import "github.com/zmb3/spotify"

// *spotify.Client satisfies the interface SpotifySearcher
var _ SpotifySearcher = &spotify.Client{}

type SpotifySearcher interface {
    GetTrack(id spotify.ID) (*spotify.FullTrack, error)
    GetTracks(ids ...spotify.ID) ([]*spotify.FullTrack, error)
    Search(query string, typ spotify.SearchType) (*spotify.SearchResult, error)
}
```

**Spotify Connect** is a `zmb3/spotify.Client` instance that uses the authorization code described in subsection 2.1.3. It implements an interface accordingly:

```
import "github.com/zmb3/spotify"

// *spotify.Client satisfies the interface SpotifyConnect
var _ SpotifyConnect = &spotify.Client{}

type SpotifyConnect interface {
    PlayerCurrentlyPlaying() (*spotify.CurrentlyPlaying, error)
    PlayerRecentlyPlayed() ([]spotify.RecentlyPlayedItem, error)
    QueueSong(trackID spotify.ID) error
}
```

**Voting Service** is an internal instance that manages voting, i.e., keeps track of what songs have been voted for and how many votes each song has. It implements an interface accordingly:

```
import "github.com/zmb3/spotify"

type VoteItem struct {
    *spotify.FullTrack
    Votes int `json:"votes"`
}

//  Internally the VotingService manages a hashmap, mapping track ID to a `VoteItem`
type VotingService interface {
    AddSong(track *spotify.FullTrack) // Add song to votes, increment if exists
    GetVotes() []VoteItem             // Get list of votes, sorted with most votes first
    IsEmpty() bool                    // Return true if no songs are currently votes for
    PopSong() spotify.ID              // Remove the song with most number of votes
}
```

**Playback Tracker** is another internal instance that monitors the host's playback, so that it can remove songs from the Voting Service and add them to the playback

queue using Spotify Connect. There is no interface to present for Playback Tracker.

Because Spotify Searcher uses client credentials flow, it does not require user authorization and can be created at any time before use. On Votify Server it is created on initialization.

Spotify Connect however can only be created after user authorization. It is created directly after the host is redirected to the callback. The Authorization Code Authenticator is given the permission scopes `user-read-playback-state`, `user-modify-playback-state` and `user-read-recently-played` so that it can access the Spotify Web API endpoints 'Add to Queue', 'Currently Playing' and 'Recently Played'. The Playback Tracker depends on Spotify Connect and is therefore initialized at the same time.

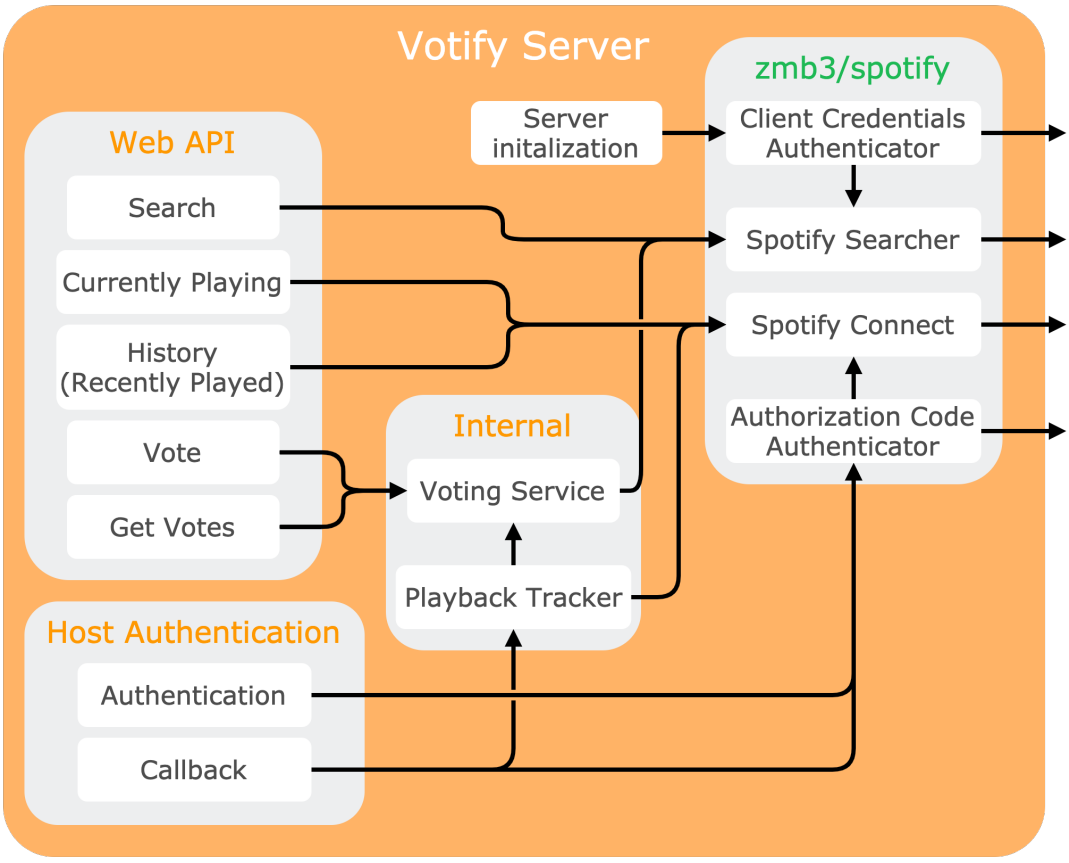



**Figure 3.9:** Visualizes the flow within Votify server.

**Web API endpoints** — As discussed in section 3.3 all Spotify resources are forwarded from Votify Server to Votify Client. The search endpoint is forwarded like this:

```
import (
    "net/http"

    "github.com/zmb3/spotify"
)

// Note that this is only a subset of the code and that error handling has been omitted.
func (server *Server) HandlerSearch() http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        // Get Search String
        searchQuery := r.URL.Query().Get("q")
        // Search on Spotify, return only type tracks.
        result, _ := server.SpotifySearcher.Search(searchQuery, spotify.SearchTypeTrack)
        // Forward the response
        w.Write(result)
    }
}
```

The handler for Currently Playing is forwarded in the same way, using Spotify Connect.

```
playbackState, _ := server.SpotifyConnect.Playing()
w.Write(playbackState)
```

Spotify Web API's history endpoint however uses a simple tracks, which does not include albums which is needed by Votify Client. To solve this issue, the recently played tracks are first fetched. Then, Spotify Searcher is used to add the missing information using the tracks' IDs:

```
// Note that this is only a subset of the code and that error handling has been omitted
func (server *Server) HandlerGetHistory() http.HandlerFunc {
    type ResponseItem struct {
        *spotify.FullTrack
        PlayedAt time.Time `json:"played_at"`
    }

    return func(w http.ResponseWriter, r *http.Request) {
        // Get tracks
        tracks, _ := server.SpotifyConnect.PlayerRecentlyPlayed()
        // Get all track IDs
        var trackIDs []spotify.ID
        for _, track := range tracks {
            trackIDs = append(trackIDs, track.Track.ID)
        }
        // Get missing track information
        fulltracks, _ := server.SpotifySearcher.GetTracks(trackIDs...)
        // Combine history information with full track information
        var response []ResponseItem
        for i := range tracks {
            response = append(response, ResponseItem{
                FullTrack: fulltracks[i],
                PlayedAt:  tracks[i].PlayedAt,
            })
        }
        w.Write(response)
    }
}
```

The endpoints 'Vote' and 'Get Votes' call the methods AddSong and GetVotes of the Voting Service.

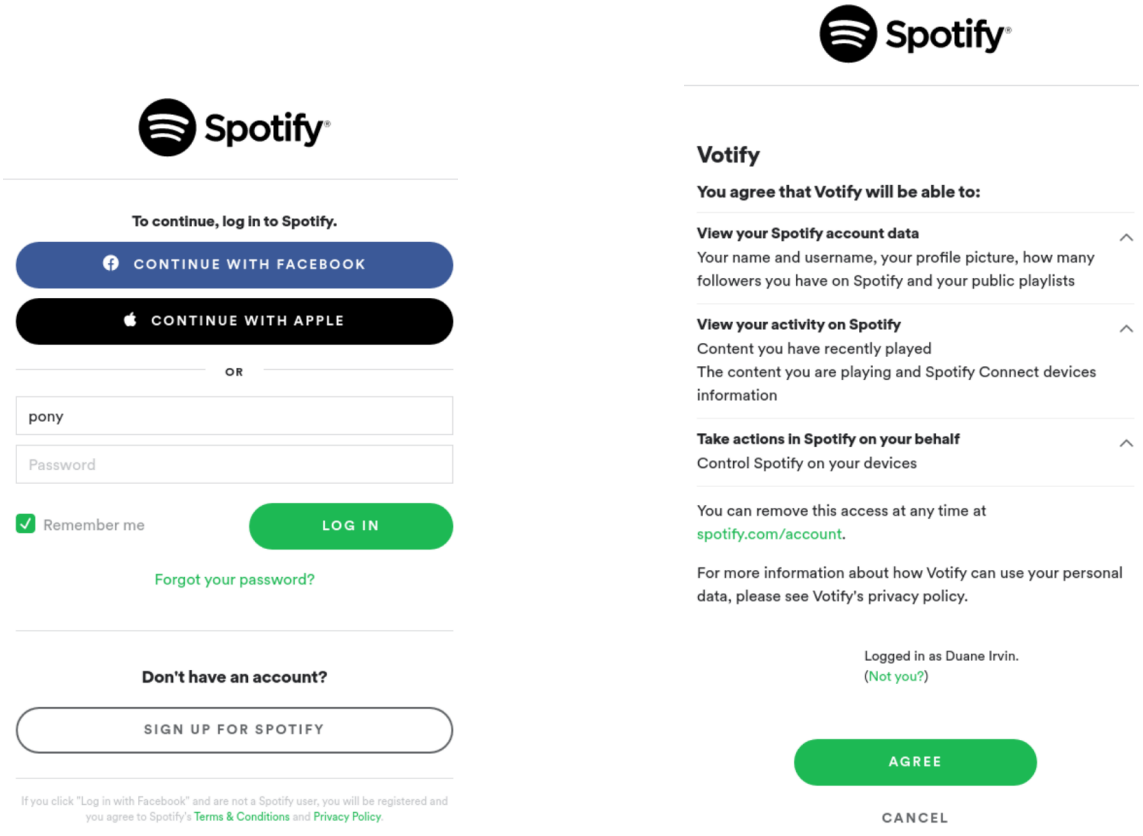
**Playback Tracking / Queueing** — The playback tracker checks the current playback on the host account every 5 seconds. If there is less than 15 seconds left until the current song ends, the top-voted song is removed for the Voting Service and added to queue via Spotify Connect. This happens only under the condition that there are songs available in the voting system and no songs are currently queued at the host account.

```
import (
    "net/http"
    "time"

    "golang.org/x/net/context"
)

// Note that this is only a subset of the code and that error handling has been omitted

// Subset of Callback Handler
func (server *Server) HandlerCallback() http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        // Get access token and refresh token from code
        token, _ := server.Authenticator.Token("state-string", r)
        // Create client that is using Authorization Code
        client := server.Authenticator.NewClient(token)
        server.SpotifyConnect = &client
        // Create PlaybackTracker from VotingService and SpotifyConnect
        server.PlaybackTracker = NewPlaybackTracker(server.VotingService, server.SpotifyConnect)
        go server.PlaybackTracker.Run(context.Background())
    }
}

const RunFrequency = 5 * time.Second

// Subset of the Run method
func (pbt *PlaybackTracker) Run(ctx context.Context) {
    // Get timer channel from internal timer
    timerC := pbt.Timer.C
    for {
        select {
        // When timer sends a signal
        case _ = <-timerC:
            // Reset timer at the end
            defer pbt.Timer.Reset(RunFrequency)
            // Reset timer and wait for signal:
            // - if VotingService is empty,
            // - if more than 15 seconds is left of the current song,
            // - if a song is already queued,
            if shouldWait() {
                continue
            }
            id := pbt.VotingService.PopSong()
            pbt.SpotifyConnect.QueueSong(id)
        }
    }
}
```

# 4

# Results

The Votify Server is hosted on port 8080 and the Votify Client on port 3000 on the developer machine. In this case, the developer machine has the IP address `192.168.1.114`. While the host is connected to the same network as the developer machine, the host can navigate to http://192.168.1.114:8080/auth for authentication. The host gets redirected to Spotify Account Services for login (see Figure 4.1a). After login, Spotify asks the host to grant access for Votify to view recently played content and currently playing content at the host's Spotify accounts and to control Spotify on the host's devices (see Figure 4.1b).




**(a)** Login page **(b)** Authorization page

**Figure 4.1:** Shows what login and authorization looks like for a host when the host authorizes Votify to access his/her account at Spotify Account Services.

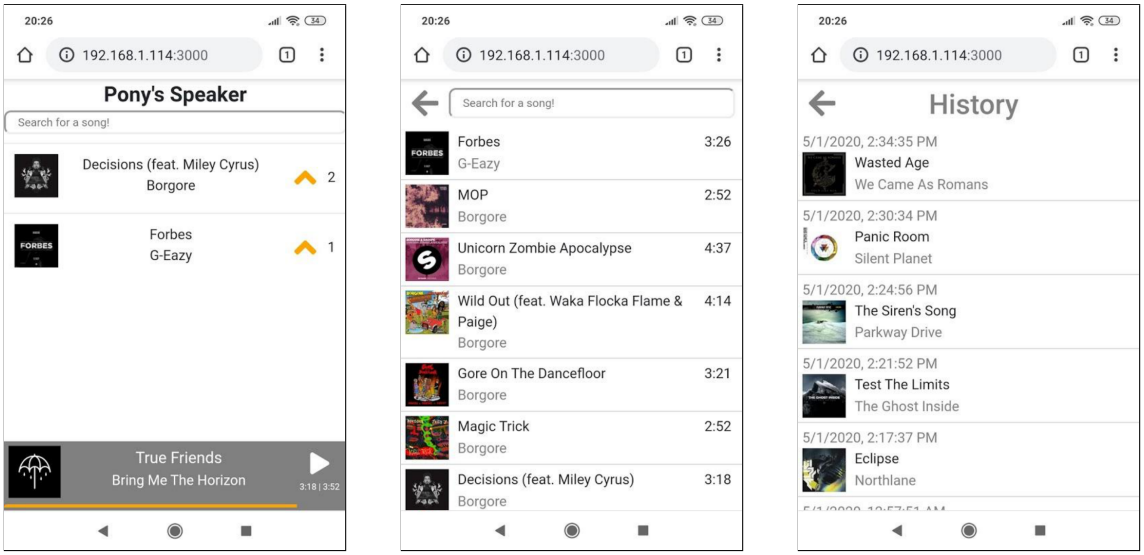
The host must now make sure music is playing Spotify using the account which was used for previous authorization before attendees can start voting for songs. Attendees can then connect to the same local network and go to http://192.168.1.114:3000 to start voting for songs.

Before we look back at the purpose and objective of his thesis, we will first make recap of the user stories that were defined in section 3.1 and see how they have been fulfilled:

**Playback** — The first thing visible in the Votify Client is the main view. There is a playback bar that stays consistent with what is playing on Spotify. Song name, artist, album cover, duration of the song and its progress is visible (see Figure 4.2a). The progress is shown both as a timestamp and a progress bar. This fulfills US3-Playback.

**Searching** — When typing something in the search bar at the main view, the search view is shown. It shows a list of matching songs from Spotify containing song name, artist, album cover, duration of the song (see Figure 4.2b). By tapping a song, it becomes voted for and will appear on the main view with one vote. This fulfills US1-Search.

**Voting** — If there are any songs that has been voted for, they are visible on the main view. If the attendee has not voted for it, he/she can do it by tapping the button to the right. The number of votes next to the button will increment and the button will become disabled. The list is ordered by the number of votes and will rearrange if needed after a vote. This fulfills US2-Vote.




**(a)** Main View **(b)** Search View **(c)** History View

**Figure 4.2:** Screenshots from Votify Client on an Android phone in Google Chrome.

**History** — When tapping the progress bar in the main view, the history view is shown. The 20 latest songs that has been played on the host's Spotify account are listed, with song name, artist, album cover and exact timestamp when the song started playing (see Figure 4.2c). By tapping the back button, the arrow in the top-right corner, the user is navigated back to the main view. This fulfills US4-History.

**Queue** — Whenever a song is about to end on the host's Spotify account, the song with the most number of votes on the main view will be removed from Votify and be the next song to start playing. The host can see it being added to the queue in the Spotify app the same way it is when it is added manually to the queue as it was shown in Figure 1.1 on page 2. This fulfills US5-Queue.

# 5

# Discussion

## 5.1 Purpose and Objectives

Inherently by the design of the system, the software does not require an account nor to be download by the attendee. This was among the objectives. Furthermore, the problems described in section 1.1 were to be solved. Votify solves the following problems:

- The attendee does not need an account for Spotify. It does introduce the need for any other accounts, such as signing up for Votify.
- Altough the attendee may not get to hear the song they want to hear immediately, the attendee does not need to add songs 'ahead of time' the way they would with collaborative playlists. Rather than playing songs in a random order or in the order they were added, the voting system should allow more attendees' song requests to be met earlier.
- The attendee does not to get access to the playback device, thus resolving the privacy issue for the host.
- The attendee gains no access to stop the music, immediately change song nor directly change the order of which songs will be played.
- Although the attendee needs an address to access Votify, the address is short enough to be easily entered by hand unlike a link to a collaborative playlist.

All of the problems mentioned in Background have been solved and the Objective has been achieved. The implementation of Votify did however introduce one new problem: For the attendee to vote for songs on Votify, the attendee has to be connected to the same local network as the machine hosting Votify.

## 5.2 Multiple Hosts and Public Deployment

The implementation from this thesis only supports use within a local network limited to one host at the time. Obviously it introduces a new obstacle for each attendee to be connected to the network to use the application, which does not align with the purpose "reduce the threshold for attendees to be part of choosing music. . . ". It also requires each host to run their own server. For many situations, such as in a public park, no local network is even available.

The most natural next step for Votify would be to break delimitations DE1-OneHost and DE5-Local by implementing support for multiple hosts and then deploy it on a public server. A few additional features would then make sense to implement:

- An admin dashboard where the host can activate and deactivate Votify. Otherwise Votify could keep controlling the host's Spotify account even when it is used privately.
- Password controlled access to Votify, such that a host can control who is able to vote for songs in their voting session. Otherwise anyone in the world would be able vote for songs on the host's account.

## 5.3 Cheat Protection

The implementation from this thesis uses client side prevention for the attendee to vote for the same song multiple times. An easy way to get around this is to clear the browser, use incognito mode and go the Votify multiple times or using multiple browsers. For native smartphone applications, methods can be used to uniquely identify different devices. For Android `ANDROID_ID` [19] is an option and for iOS there is `identifierForVendor` [20], both of which should be good enough for an application like this. So called fingerprints, defined as "A set of information elements that identifies a device or application instance" by IETF [21], does not come out of box by browsers. In fact some browsers actively try to restrict fingerprinting as a privacy measure, one of which being Google Chrome [22]. Further more, user data collection must follow the regulations of GDPR which raises the question: Is this user data collection? If yes, what measures must be taken by Votify? Breaking delimitation DE4-Security by moving this specification to the server may therefore prove to be a quite difficult problem to solve. Relying on the previously mentioned password controlled access to Votify may be a more viable option.

## 5.4 Additional Improvements and Features

The client-side has many potential small features that can be used to improve the user experience:

- By implementation delimitation DE3-Visual minimal effort was put into the visual design of Votify. There is plenty of room for improving the visual design to improve the user experience.
- The URI for Spotify tracks is embedded in the track object. The URI could be added as a link to each element in the `History View`, such that the attendee can with one tap show the song in the Spotify application to save it in his/her playlists.

## 5.5 Ethical considerations

There was no intention to collect any information about users in the development of Votify. The software does not collect any user information and it will be maintained that way. It should therefore not have any ethical impact in that sense. No other relevant ethical aspects has been found.

## 5.6 Sustainability

No substantial enironmental impacts related to this thesis has been identified.

# Bibliography


[1] Spotify. Company info. Accessed: June 2020. [Online]. Available: https://newsroom.spotify.com/company-info/

[2] Spotify. Web api. Accessed: June 2020. [Online]. Available: https://developer.spotify.com/documentation/web-api/

[3] IETF. Rfc 6749 — the oauth 2.0 authorization framework. Accessed: June 2020. [Online]. Available: https://tools.ietf.org/html/rfc6749

[4] Spotify. App settings. Accessed: June 2020. [Online]. Available: https://developer.spotify.com/documentation/general/guides/app-settings/

[5] Spotify. Authorization guide. Accessed: June 2020. [Online]. Available: https://developer.spotify.com/documentation/general/guides/authorization-guide/

[6] Spotify. Authorization scopes. Accessed: June 2020. [Online]. Available: https://developer.spotify.com/documentation/general/guides/scopes/

[7] Google. The go project. Accessed: June 2020. [Online]. Available: https://golang.org/project/

[8] Google. Frequently asked questions (faq). Accessed: June 2020. [Online]. Available: https://golang.org/doc/faq

[9] Google. Generics. Accessed: June 2020. [Online]. Available: https://go.googlesource.com/proposal/+/master/design/go2draft-generics-overview.md

[10] zmb3. A go wrapper for the spotify web api. Accessed: June 2020. [Online]. Available: https://github.com/zmb3/spotify

[11] Microsoft. microsoft/typescript. Accessed: June 2020. [Online]. Available: https://github.com/microsoft/TypeScript

[12] Google. V8 javascript engine. Accessed: June 2020. [Online]. Available: https://v8.dev/

[13] O. Foundation. What is npm? Accessed: June 2020. [Online]. Available: https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/

[14] O. Foundation. npm-package.json. Accessed: June 2020. [Online]. Available: https://docs.npmjs.com/files/package.json

[15] O. Foundation. npx. Accessed: June 2020. [Online]. Available: https://www.npmjs.com/package/npx

[16] Facebook. facebook/react. Accessed: June 2020. [Online]. Available: https://github.com/facebook/react

[17] Facebook. Create a new react app. Accessed: June 2020. [Online]. Available: https://reactjs.org/docs/create-a-new-react-app.html

[18] D. Abramov. Redux. Accessed: June 2020. [Online]. Available: https://redux.js.org/

[19] Google. Settings.secure. Android Developers. Accessed: June 2020. [Online]. Available: https://developer.android.com/reference/android/provider/Settings.Secure

[20] Apple. identifierforvendor — uidevice. Apple Developer Documentation. Accessed: June 2020. [Online]. Available: https://developer.apple.com/documentation/uikit/uidevice/1620059-identifierforvendor

[21] IETF. Rfc 6973 — privacy considerations for internet protocols. Accessed: June 2020. [Online]. Available: https://tools.ietf.org/html/rfc6973

[22] Google. Improving privacy and security on the web. Accessed: June 2020. [Online]. Available: https://blog.chromium.org/2019/05/improving-privacy-and-security-on-web.html

# A

# Development Environment

## A.1 Developer and Hosting Machine

For development and server/client hosting, the following machine was used:

| **Macbook Pro 13** | |
|---|---|
| OS | `macOS Catalina 10.15.4` |
| `Go` | `go1.13.4 darwin/amd64` |
| `Node.js` | `v14.3.0` |
| `npm` | `6.14.4` |
| Google Chrome | `83.0.4103.61` (Official Build) (64-bit) |

## A.2 Testing Environments

For testing the Votify Client, two devices were used — The developer machine from section A.1 using Google Chrome's Device Simulation Mode and the device below:

| **Xiaomi Redmi Note 7** | |
|---|---|
| OS | Android 9 (Pie) `PKG1.180904.001` |
| Google Chrome | `83.0.4103.83` |

## A.3 Votify Server Dependencies

This is the `go.mod` containing the dependencies that was used for Votify Server.

```
go 1.13

require (
    github.com/golang/mock v1.4.3
    github.com/gorilla/mux v1.7.4
    github.com/stretchr/testify v1.6.0
    github.com/zmb3/spotify v0.0.0-20200525010707-bc712583571e
)
```

## A.4 Votify Client Dependencies

This is the `package.json` that was used for Votify Client:

```
{
    "name": "votify",
    "version": "0.1.0",
    "private": true,
    "dependencies": {
        "@fortawesome/fontawesome-svg-core": "^1.2.26",
        "@fortawesome/free-solid-svg-icons": "^5.12.0",
        "@fortawesome/react-fontawesome": "^0.1.8",
        "@testing-library/jest-dom": "^4.2.4",
        "@testing-library/react": "^9.4.0",
        "@testing-library/user-event": "^7.2.1",
        "@types/jest": "^25.2.1",
        "@types/node": "^13.11.1",
        "@types/react-bootstrap": "^0.32.20",
        "@types/styled-components": "^4.4.2",
        "axios": "^0.19.2",
        "bootstrap": "^4.4.1",
        "react": "^16.12.0",
        "react-bootstrap": "^1.0.0-beta.16",
        "react-dom": "^16.12.0",
        "react-redux": "^7.1.3",
        "react-scripts": "3.3.0",
        "redux": "^4.0.5",
        "redux-thunk": "^2.3.0",
        "styled-components": "^5.0.0",
        "typescript": "^3.7.5"
    },
    "scripts": {
        "start": "export REACT_APP_APIURL=http://192.168.1.114:8080 && react-scripts start",
        "build": "react-scripts build",
        "test": "react-scripts test",
        "lint": "eslint -ignore-path .gitignore **/*.tsx",
        "eject": "react-scripts eject"
    },
    "eslintConfig": {
        "extends": "react-app"
    },
    "browserslist": {
        "production": [
            ">0.2%",
            "not dead",
            "not op_mini all"
        ],
        "development": [
            "last 1 chrome version",
            "last 1 firefox version",
            "last 1 safari version"
        ]
    },
    "devDependencies": {
        "@types/enzyme": "^3.10.5",
        "@types/enzyme-adapter-react-16": "^1.0.6",
        "@types/react": "^16.9.19",
        "@types/react-dom": "^16.9.5",
        "@types/react-redux": "^7.1.7",
        "@typescript-eslint/eslint-plugin": "^2.17.0",
        "@typescript-eslint/parser": "^2.17.0",
        "editorconfig": "^0.15.3",
        "enzyme": "^3.11.0",
        "enzyme-adapter-react-16": "^1.15.2",
        "eslint": "^6.8.0",
        "eslint-plugin-react": "^7.18.0"
    }
}
```

This is the `TypeScript` compiler configuration file, `tsconfig.json`, that was used for Votify Client:

```
{
  "compilerOptions": {
    "target": "es5",
    "lib": [
      "dom",
      "dom.iterable",
      "esnext"
    ],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react",
    "types": [
      "jest"
    ]
  },
  "include": [
    "src"
  ]
}
```