# Digital Record player

Justin Hangoebl, Maurer Florian, Paul Bachinger, Kejda Domi

June 3, 2022

# 1 Digital Record Player

## 1.1 Project Idea

Vinyl Players getting back on track, and more and more people want one. But they are expensive, and it is hard to get your fingers on some records. Furthermore, everybody has a Spotify, YouTube Music or similar account already. So we thought why don't we modernize and digitalize the record player and its components. We decided on a system. This system has a RaspberryPI 4 as a Base and is connected to an NFC Reader, which enables Spotify playback on the RaspberryPIs Bluetooth Device.

## 1.2 Team

| Teammember | Task |
|---|---|
| Florian Maurer | Spotify API |
| Paul Bachinger | Frontend |
| Kejda Domi | Broker (Apache Kafka) |
| Justin Hangoebl | Hardware, Express Server |

Table 1: Team

## 1.3 Project Setup

First of all, we did some research on the Spotify API, RFID, and generally the hardware. We quickly found that Express JS and React, with a NodeJS server is the way to go, for the Spotify API and a Server running on a RaspberryPI, where we decided on the RaspberryPI because one member had a RaspberryPI and an NFC Reader. We as well found out we need some kind of broker/subscriber tool to write from a client the NFC Card, where we decided on Apache Kafka.

We decided to create a small design concept for the client-side and the hardware.

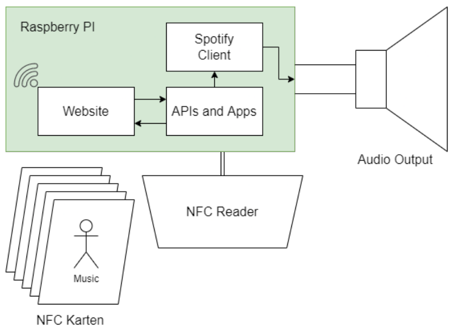


Figure 1: The Hardware sketch

Then we began to set up the project, we created an issue tracker using Trello, created a GitHub repository and began setting up a small ExpressJS/React project, which we pushed to the repository. With this Setup done each team member could begin working on their respective tasks. On the project

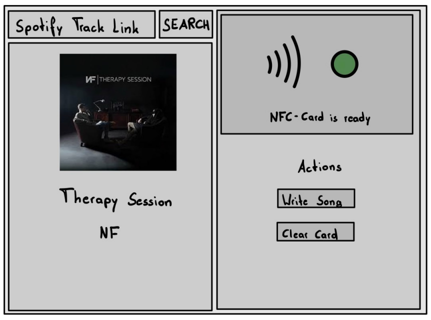


Figure 2: The Web Client Sketch

structure, we decided on a folder for each problem, so we can work easily parallel, because the server should be able to run on the RaspberryPI and localhost, and using branches was out of scope because of the integration work later on.

## 2 Hardware and Embedded Code

This section should give the necessary Insights on how the RaspberryPI was set up, the Project Structure was generated. Furthermore, it should show how the NFC Reader is assigned and the Script that reads it and request the playback of the song.

### 2.1 RaspberryPI

The RaspberryPI 4 is a small single-board computer, which has an operating system (Raspbian, a subset of a Debian OS) and has GPIO. We chose the RaspberryPI as our basis because it is easy to use and set up, it has those GPIO for the Input and Output via RFID chips, it can run the Spotify Client which is necessary, and finally, it also has a Bluetooth antenna to configure a playback device.

#### 2.1.1 Setting Up the RaspberyPI

With the base ExpressJS and React project we had to install, nodejs, npm and the necessary node-modules. On the RaspberryPI this is easy to do using the apt package manager, all the base modules were installed using the following lines of code:

```
$ sudo apt-get install nodejs npm -y
$ npm install express --save
$ npm install -g create-react-app
$ npm install --save react react-dom
```

The backend code for the hardware was written in Python, so we had to install the necessary packages on the RaspberryPI, first of all, we needed a way to send HTTP requests, for this the package *pycurl* was used and a package for the NFC Reader. The RFID Reader was the Model RFID MFRC522, this reader has its python package mfrc522. To be able to control GPIOs we needed the RPi.GPIO package as well. Those packages are installed with the following lines:

```
$ pip install pycurl ByteIO mfrc522 RPi.GPIO sys
```

With these installations, the RaspberryPI is good to go.

#### 2.1.2 Wiring the Hardware

The RaspberryPI has 40 GPIO (General Purpose Input Output), which are Pins labelled from 1 - 40, beginning at the top left with "1" and on the right top corner is "2" this scheme follows done till Pin 40. They have all a purpose, but we only need to consider those from table 2. With this the MFRC522 can be connected to the RaspberryPI using the information from table 2, it should then look like figure 3.

This is all the wiring done because the rest works wirelessly either using Bluetooth or WiFi.

| RF522 | RPi |
|---|---|
| SDA | Pin 24 / GPIO8 (CE0) |
| SCK | Pin 23 / GPIO11 (SCKL) |
| MOSI | Pin 19 / GPIO10 (MOSI) |
| MISO | Pin 21 / GPIO9 (MISO) |
| IRQ | — |
| GND | Pin6 (GND) |
| RST | Pin22 / GPIO25 |
| 3.3V | Pin 1 (3V3) |

Table 2: Connection of the MFRC522

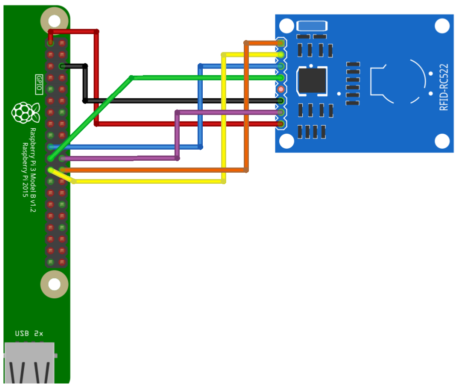


Figure 3: The finsished wiring of the RaspberryPI and the MFRC522

### 2.1.3 The Implementation of Read/Write and Requests

The Implementation is done in Python. In the main file, we first of all need to create a reference to an MFRC522 object. Another functionality includes the writing of the card using command-line arguments. If we decide to configure the card we use the function write(), where we get a command line input using the input from python and then write this input to the reader using the SimpleMFRC522().write() method. After writing the Spotify Song ID on the Tag we can get into reading mode, where we execute the read() method. The read() function begins by sending the website the information that the NFC Reader is connected to and currently reading, using a pycurl request. Then we try to read an NFC Card, the MFRC522.read() method is a blocking call, therefore the code stands until it reads a card. In this case, we read a card and send another pycurl request to the server, then append the ID to the URL as a query parameter to the server, after 10s of delay the Reader reads again

```python
if __name__ == "__main__":
    print("Configuring NFC RFID\n")
    reader = SimpleMFRC522()
    print("Successfully Configure NFC RFID\n")
    try:
        if(sys.argv[0] == 'config' or sys.argv[0] == 'Config'):
            write(reader)
        read(reader)
    finally:
        GPIO.cleanup()
```

```python
# Write the Input to the Tag
def write(reader):
    print("Place your tag to write")
    text = input('Input Song ID:')
    reader.write(text)
    print("Written")
```

```python
def read(reader):
    curl("http://localhost:5000/nfc_ready?nfc_ready=true")
    while True:
        print("Reading...")
        nId, nText = reader.read()
        print("ID: %s\nText: %s \n" % (nId,nText))
        request(nText);
        # only read all 10 seconds
        time.sleep(10)
```

```python
# another way to request from the server
def curl(url):
  c = pycurl.Curl()

  c.setopt(c.URL, url)
  c.setopt(c.CUSTOMREQUEST, "POST")

  c.perform()
  c.close()
```

## 3 Spotify API

### 3.1 General Information

The Spotify Web API can be used to gather data from the Spotify database such as information about songs, cover images and much more. It is based on REST principles where a developer can call certain endpoints to receive JSON metadata.

It is also possible to access private data via the API like user-profiles and playlists or even to control the current playback of a specific account. To gain access to these features, an application first needs to obtain permission to do so by asking for it via the Spotify Accounts service.

As for our project, we need a client that works as a middleman between the user and the Spotify API. This is where an Express.js web server comes into play. This is done and explained in 4

### 3.2 Authorization (OAuth2.0)

Authorization in general describes the process of granting any user or application the permission to specific Spotify data and features.

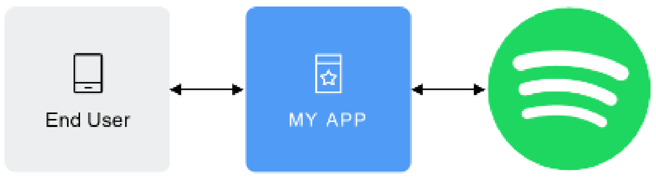


Figure 4: The Authentication Principle of the SpotfiyAPI, taken from SpotifyAPIDoc

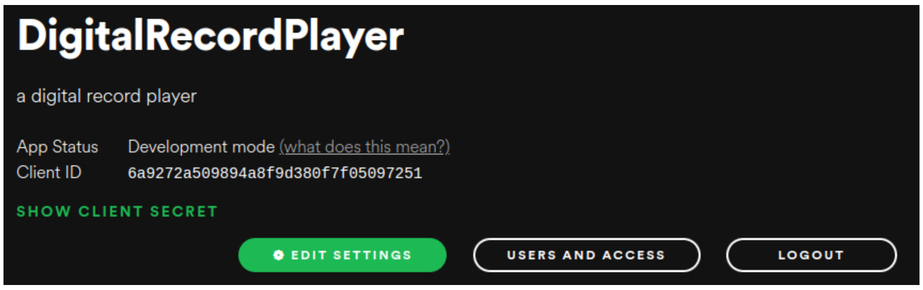
In the figure 4 the OAuth2.0 framework can be seen:

- **End User** describes a specific Spotify account that grants access to private data.
- **My App** describes a client (like a web application) that asks the **End User** for access to the private data.
- **Server** (Spotify logo) is where the private data is stored and it provides the authorization via OAuth2.0.

The procedure of authorization via OAuth2.0 works by the client sending valid client credentials (a client ID and a client secret ID) to the authorization server which then checks the validity of the credentials and returns an access token if the authorization was successful. This access token, also called OAuth2.0 token can then be used to process certain API calls on behalf of the user's account.

During authorization, a set of scopes is used to determine which features will be permitted, such as controlling the playback or editing a playlist of the user.

Spotify offers any user to create so-called integration modules based on their account. These modules create a reference point to the account which the application can use for the authorization. Such a module comes with a set of client credentials which is just what we need for authorization. Additionally, it is


DigitalRecordPlayer

a digital record player

App Status Development mode (what does this mean?)
Client ID 6a9272a509894a8f9d380f7f05097251

Figure 5: This is the Project Dashboard on the Spotify Developer Website

possible to share the access of this module to other Spotify accounts so that multiple accounts can proceed with a valid authorization.

There is a Spotify instance running on the Raspberry PI with a logged-in Spotify Account. This same account has an integration module running from where we can obtain the valid client ID and client secret ID. This means that every account that has been granted access to the module can log in to our website/webserver and use it as expected.

## 3.3 OAuth2.0 Implementation

The figure 6 describes perfectly how the whole authorization flow works step by step.

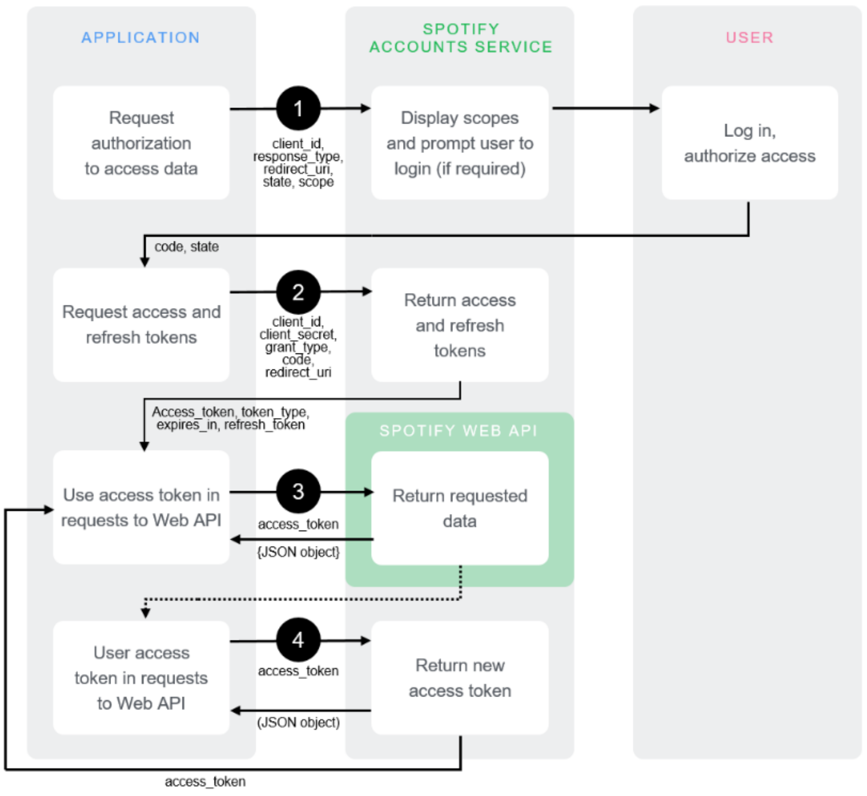

APPLICATION
SPOTIFY ACCOUNTS SERVICE
USER
Request authorization to access data
1
client_id, response_type, redirect_uri, state, scope
Display scopes and prompt user to login (if required)
Log in, authorize access
code, state
Request access and refresh tokens
2
client_id, client_secret, grant_type, code, redirect_uri
Return access and refresh tokens
Access_token, token_type, expires_in, refresh_token
SPOTIFY WEB API
Use access token in requests to Web API
3
access_token
{JSON object}
Return requested data
User access token in requests to Web API
4
access_token
(JSON object)
Return new access token
access_token

Figure 6: The authorization flow of the Spotify API, taken from SpotifyAPI Documentation

To begin the implementation of the authorization process in our webserver a couple of variables and helper functions have to be declared:

```
// client ID and client secret stored in a .env file
var spotify_client_id = process.env.SPOTIFY_CLIENT_ID;
var spotify_client_secret = process.env.SPOTIFY_CLIENT_SECRET;

// the redirect uri of our integration module
```

```javascript
var spotify_redirect_uri = "http://localhost:3000/auth/callback";

// helper function to create a random string of a specific size
var generateRandomString = function (length) {
  var text = "";
  var possible =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

  for (var i = 0; i < length; i++) {
    text += possible.charAt(Math.floor(Math.random() * possible.length));
  }
  return text;
};
```

Next step is to start the authorization process to be able to give the user the ability to use the API:

```javascript
// create the url to the Spotify login pane and subsequently redirect
app.get("/auth/login", (req, res) => {
  // scope: all the permissions we are asking for
  var scope =
     "streaming user-read-email user-read-private " +
     "app-remote-control user-modify-playback-state " +
     "user-read-playback-state user-read-currently-playing";
  // protection against attacks
  var state = generateRandomString(16);

  var auth_query_parameters = new URLSearchParams({
    // Has to be "code"
    response_type: "code",
    // id of the module
    client_id: spotify_client_id,
    // scopes
    scope: scope,
    // the URI to redirect to after the user grants or denies permission
    redirect_uri: spotify_redirect_uri,
    // state
    state: state,
  });

  // redirect
  res.redirect(
    "https://accounts.spotify.com/authorize/?" +
      auth_query_parameters.toString()
  );
});
```

Now we need to request the access and update the tokens:

```javascript
// get the callback from the Spotify API
app.get("/auth/callback", (req, res) => {
  // The auth code returned from the previous request
  var code = req.query.code;

  var authOptions = {
    // where the next call will be to
    url: "https://accounts.spotify.com/api/token",
    form: {
      code: code,
      redirect_uri: spotify_redirect_uri,
      // This field must contain this value
      grant_type: "authorization_code",
    },
    headers: {
      // Field that has to contain the id and the secret in a format
      Authorization:
        "Basic " +
        Buffer.from(spotify_client_id + ":" +
             spotify_client_secret).toString("base64"),
      // field must contain this value
      "Content-Type": "application/x-www-form-urlencoded",
    },
    json: true,
  };

  // request the access_token from Spotify API and
  // then redirect to the main page
  request.post(authOptions, function (error, response, body) {
    if (!error && response.statusCode === 200) {
      access_token = body.access_token;
      res.redirect("/");
    }
  });
});
```

And lastly, we can retrieve the access token (OAuth2.0 token) which we will use for the API calls.

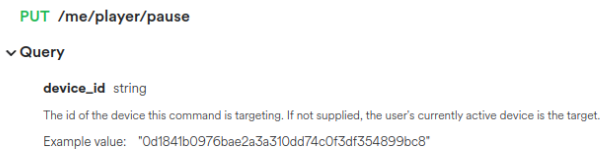
```
// Pane to get the auth token, to validate the login with the Spotify API
app.get("/auth/token", (req, res) => {
  res.json({ access_token: access_token });
});
```

## 3.4 Spotify API Endpoints

In this project, different Spotify API endpoints were used, for certain features like pausing and resuming a track.

## 3.5 Pause Playback, Skip to the Next or Previous song

These commands can be achieved using GET requests to the SpotifyAPI they work similarly with the only difference being the link.


PUT /me/player/pause

Query

device_id string

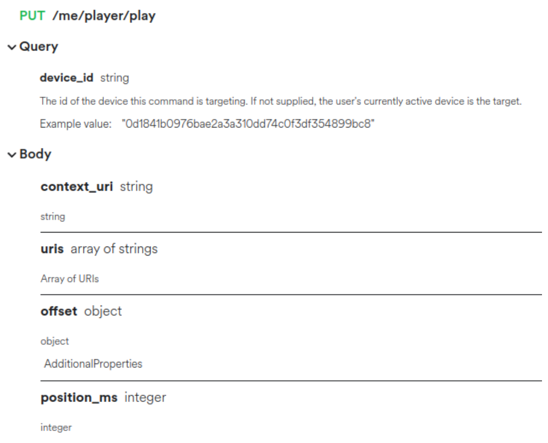
The id of the device this command is targeting. If not supplied, the user's currently active device is the target.

Example value: "0d1841b0976bae2a3a310dd74c0f3df354899bc8"

Figure 7: How a pause Playback request is configured in the SpotifyAPI

## 3.6 Start/Resume Playback

This PUT endpoint will either start a new playback on the user's active device or resume the currently paused playback, this will play the song/artist/album/playlist from the given body, i.e. the body states what should happen.


PUT /me/player/play

Query

device_id string

The id of the device this command is targeting. If not supplied, the user's currently active device is the target.

Example value: "0d1841b0976bae2a3a310dd74c0f3df354899bc8"

Body

context_uri string

string

uris array of strings

Array of URIs

offset object

object

AdditionalProperties

position_ms integer

integer

Figure 8: How a play Song request is configured in the SpotifyAPI

# 4 ExpressJS Server

Express is a minimal and flexible Node.js web application framework, with a myriad of HTTP utility methods and middleware, which is just perfect for us. Because the RaspberryPI is not as powerful as a full-blown server computer, and the needed HTTP functionality for requests from the outside.

## 4.1 The Basic Principle

To create an ExpressJS server is easy, initialize a new node project, using

```
$ npm init
```

and install express when using requests from other devices or scripts we need an HTTP server and setup cross-origin reference sharing with the HTTP and cors package so this can be installed as well, using

```
$ npm install express HTTP cors
```

, to create a server we need to edit the index.js file. In listing 7 firstly the ExpressJS package, HTTP package and the cors package need to be included, an app variable needs to be initialized and the cors option needs to be set. With these variables, we can create an HTTP server. The built-in methods server.listen() and server.on(), open the port and start the server respectively.

```
const express = require("express");
const http = require("http");
const cors = require("cors");

// SERVER Variables and SERVER Setup
var app = express();

// cross origin reference setup
var corsOptions = {
  origin: [
    "http://localhost:3000",
    "http://localhost:5000",
    "http://localhost",
    ‘http://${raspberrypiIP}:3000‘,
    ‘http://${raspberrypiIP}:5000‘,
  ],
};

app.use(cors(corsOptions));

var server = http.createServer(app);

// Server Pane, listen to the requests on PORT 5000
server.listen(5000, "0.0.0.0");
server.on("listening", function () {
  console.log(
    "Express server started on port %s at %s",
    server.address().port,
    server.address().address
  );
});
```

The server is started using

```
$ npm run dev
```

, this starts the server in a developer setting, which is very helpful for debugging reasons.

## 4.2 The get Requests

This server won't do anything because we did not set up the HTTP requests this is done using the app variable created earlier. Our server has two GET requests, using the app instance, which returns if the NFC-Card is ready or the id of the currently playing song. These variables are global in the server.

```
$ curl -X http://{IP address of RaspberryPI}:5000/id
=> {id: -1}
$ curl -X http://{IP address of RaspberryPI}:5000/nfc_ready
=> {nfc_ready: false}
```

```
global.id = -1;

// Pane to get the id
app.get("/id", (req, res) => {
  res.json({ id: id });
});
```

## 4.3 The post Requests

It is not enough to only get this information, we also want to write it to the server and play songs from the Spotify API. Done using POST requests to the express server, so we now need to use the function app.get(). Furthermore, we now want to send requests to the SpotifyAPI, thus the usage of the Axios package, which needs to be installed with npm install Axios. The requests for /play will get the id as a query parameter, and the usage of an asyncHandler obtains the id of the currently playing song using a get request to the Spotify API, as explained in section 3 and only if we are not currently playing this song we start the song with the id from the query parameters. When this was successful, it returns a success code. This is the same for /play, /skip and /prev, but just a different API call. And finally, the /nfc_ready function now sets the variable for the get request.

```
// To be able to pause Songs from a Client Request / from the NFC Card
app.post("/pause", (req, res) => {
  // making a axios request to the Spotify API to pause the Song with the ID
  axios({
    url: "https://api.spotify.com/v1/me/player/pause/",
    method: "put",
    headers: {
      authorization: `Bearer ${access_token}`,
    }
  });
  res.status(204);
}
);
```

An example POST Request on the ExpressJS Server

## 4.4 React and Login

Now, we need to create a login page, this was done in react on port 3000, to be able to log in to the app using your Spotify account. When starting the server the ReactApp (App.js) will start as well. On this react page, we just try to get the /auth/token and if the NFC Card is ready, using the server GET requests, if this token is empty, we send the user to the login page, otherwise to the final page, with information. The GET requests are handled in React useEffect and useState functions explained in section 5. The login page is, till now, a simple HTML Page with a button that redirects to the login explained in section 3. When a user is logged in we have a token and thus are able to back-direct to the App.js and now redirect to the Setup.jsx, where the user can get information on the system.

```
//Base React App
function App() {
  // State as token, changes when Login was success
  const [token, setToken] = useState("");
  // State as id, because we want to change it on run time, from NFC
  const [nfc_ready, setNFCState] = useState("")

  // changes the state on Login and the state of the id in case we try to play using NFC
  useEffect(() => {
    async function getToken() {
      const response = await fetch("/auth/token");
      const json = await response.json();
      setToken(json.access_token);
    }

    async function getId(){
      const response = await fetch("/nfc_ready");
      const json = await response.json();
      setNFCState(json.nfc_ready);
    }

    getToken();
    getId()
  }, []);

  // Return the Login Component when not logged-in
  // else return the Setup Component with the Token
  return <>{token === "" ? <Login /> : <Setup token={token} nfc_ready={nfc_ready}/>}</>;
}
```

### 4.5 Putting everything together

Now that everything is working, we can start the server, using

```
$ npm run dev
```

and run the python script in a second shell

```
$ python main.py
```

The user can now use the project, by logging in and placing the NFC tag on the NFC Reader.

## 5 Front End

### 5.1 React (JavaScript-Library)

React is a library for building frontend user interfaces. It was released in 2013 on Facebook. The concept is that you build components that are reusable for different parts of the user interface.

#### 5.1.1 Components

In a nutshell, components are just JavaScript functions that return HTML. This is very useful because in combination with JSX you can make responsive HTML code that can react with the user.

At first, it is just a search element, figure 9 and after insertion of a song link a component with text and image pops up, figure 10.

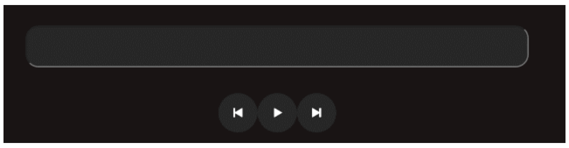
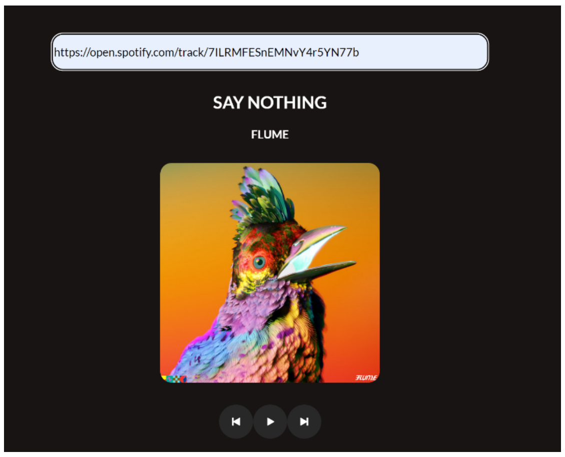


Figure 9: The Saerchbar



https://open.spotify.com/track/7ILRMFESnEMNvY4r5YN77b

Figure 10: The Web Page with a Song Link

### 5.1.2 Functions

JavaScript uses functions to return something when it is called somewhere. In a function, there can be operations or other instructions to perform and after going through them a Component gets returned.

```
function scream() {
    return <h1>Hello World!</h1>
}
```

### 5.1.3 JSX (JavaScript XML)

It is a mixture of HTML with JavaScript. This has the advantage that you can reverence a component multiple times which reduces code repetition. It can also add properties to those elements (e.g. `<Title/>`) which can be a fixed value or a variable. This gives you the possibility to dynamically change parts of a component, as an example you can do something like the following `Action onClick={ () => sendABanana } />`.

### 5.1.4 React Hooks

Hooks are functions that return a value and a function to change the value. As you can see in the code below there is a constant with a [key, value/function] pair. The use state initializes the hook and in the return statement after that, the function gets defined and we see that when you would click the button the parts of the component would dynamically change.

```
function Counter() {

    const [count, setCount] = useState(0);

    return (
        <div>
        <p>You clicked {count} times</p>
        <button onClick={() => setCount(count + 1)}>
            Click me
        </button>
        </div>
    );
}
```

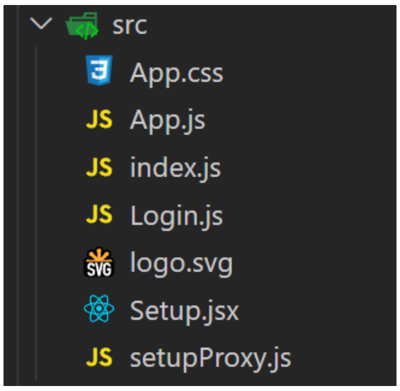
### 5.1.5 How the project is managed with React

React manages the components that should be dynamically displayed through the interaction on the webpage and Raspberry's NFC interface.

As seen in figure 11, the structure of the webpage is as it is typically for React projects. The main code is in the src folder and some general instructions are in the public folder.

All components and the stylesheet are in src. In the public folder are the instructions for the startup of the server and loading the components into the index file of the HTML.

App.css contains all the styling attributes which are explained further down in the chapter CSS. The components for the project are mainly in Setup.jsx and the rest of them are in App.js, index.js and Login.js.



```
src
    App.css
    App.js
    index.js
    Login.js
    logo.svg
    Setup.jsx
    setupProxy.js
```

Figure 11: The filetree of our React Application

## 5.2 HTML (Hypertext Markup Language)

HTML is for structuring content on a webpage. It uses tags to give the unorganized text a meaning.

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Digital Record Player</title>
    </head>
    <body>
        <h1>Hello World!</h1>
        <p>Carpe Diem</p>
    </body>
</html>
```

## 5.3 How it is used with react

HTML is used in the return of a component. As seen below it is not normal HTML syntax it is combined with JavaScript properties.

```jsx
return (
    <div className="grid">
        <div className="search">
            <form>
                <label htmlFor="searchbar"></label>
                <input  className="corner-round-1"
                    type="text" name="search"
                    id="searchbar"
                    onChange={(value) => this.getSearchResults(value.target.value)}>
                </input>
            </form>
        </div>
    </div>
);
```

### 5.3.1 How HTML is used in the project

It is used in the React components to structure the elements of the individual components. The main part of the HTML of this project is in the Setup.jsx where the player interface gets structured. The sections of the HTML are bundled up with ¡div¿ elements which have no semantical meaning except for putting other HTML elements into a block.

## 5.4 CSS (Cascading Style Sheets)

CSS is the styling language for making websites look pretty. An HTML only webpage uses some default styling of the web browser which gives an unstructured and complex user experience. To make this more visible there is one example below that shows a webpage with default styling and then with CSS styling.

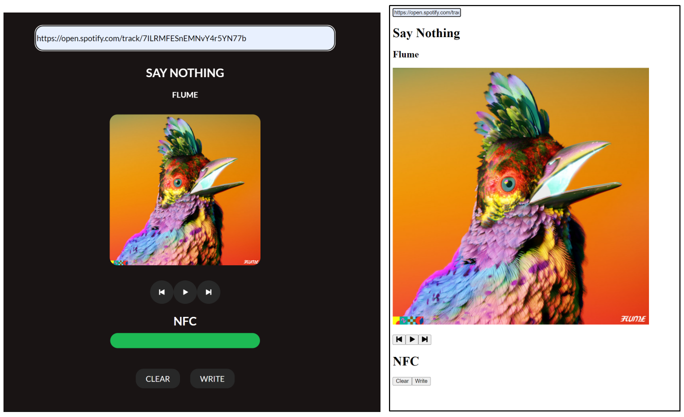

https://open.spotify.com/track/7ILRMFESnEMNvY4r5YN77b
SAY NOTHING
FLUME
NFC
CLEAR
WRITE
https://open.spotify.com/trac
Say Nothing
Flume
NFC
Clear
Write

Figure 12: Comparison between webpages with and without CSS

On the unstructured webpage the buttons and search bar are small, so users would have a hard time clicking them. The status of the NFC interface is also not visible which reduces the functionality of the webpage.

It is best practice to use a stylesheet for the CSS code, to separate the styling from the HTML code. Class

The class tag which is referenced in the HTML elements is a styling component which can be referenced multiple times throughout the HTML code to reduce code repetition. This is the reference in HTML

```html
<button class="button-style" type="button"></button>
```

This is the reference in the stylesheet

```css
.button-style {
    padding: 1vh 2vw 1vh 2vw;
}
```

### 5.4.1 ID

The id tag is used for only one specific part of an HTML page. It is to say that HTML and CSS do not enforce this but it is best practice. How it is used in the project

React automatically recognizes the stylesheet in the src folder and uses it to style the HTML. The CSS code is structured in different types of tags. So native HTML elements, classes and IDs are all in separate blocks to make the code more readable and maintainable.

```css

/**********************ELEMENTS*****************************/
html, body {
    background-color: var(--spotify-black);
    margin: 0;
}
/**********************CLASSES*****************************/
.corner-round-1{
    border-radius: 1rem;
}
```

# 6 Broker (Apache Kafka)

## 6.1 What is a Broker and what is Apache Kafka

Apache Kafka is a microservice used for distributed streaming as a platform. A stream is just an infinite stream of data. Distributed means that Kafka works in a cluster, each node in the cluster is called Broker. Those brokers are just servers executing a copy of Apache Kafka. This distributed architecture is one of the reasons that made Kafka so popular. Each broker in a cluster is identified by an ID and contains at least one partition of a topic. The Brokers is what makes it so resilient, reliable, scalable, and fault-tolerant. Figure 13 explains the principle as very simple but effective.

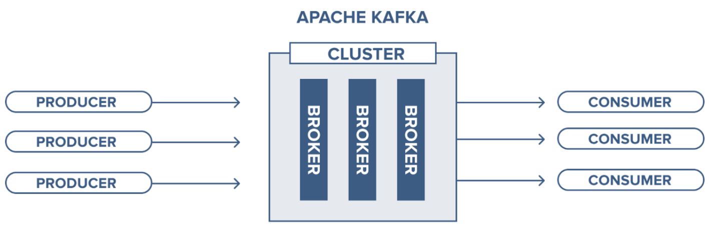


Figure 13: The principle of Kafka

## 6.2 Why we need Apache Kafka

To be able to write to the card, from the web page, we need some kind of system that allows the python script to also handle, some kind of requests or something like that, so we decided we use Apache Kafka as a Broker and an extra Thread in the python Script subscribes onto the Kafka Topic and if there is a change it will stop the Reader.read() process and further it uses the Writer.write() method to write the song id from the Kafka topic. The Producer is the client-side web page, which then tells the ExpressJS server to push onto a broker, and python is our consumer.

## 6.3 The Implementation

First of all, Kafka relies on ZooKeeper, which we have to start using the command.

```
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

Afterwards, the Kafka Broker server can be started, with the following line of code:

```
$ bin/kafka-server-start.sh config/server.properties
```

The IDs are pushed into a topic, which we have to initialize as well, the following line of code does this.

```
$ kafka_2.11-1.1.0 bin/kafka-topics.sh --create
--zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic ID
=> Created topic "ID".
```

### 6.3.1 Accessing Kafka in python

For this we are using the package kafka-python, this has to be installed `$ pip install kafka-python` We now can use kafka-python in a python script, in python we only want to listen to the topic, so we, first of all, need a Consumer which is created as follows:

```python
consumer = KafkaConsumer(
    "my-topic",
    bootstrap_servers="localhost:2181"),
    value_deserializer=lambda v: json.dumps(v).encode("utf-8"),
    auto_offset_reset='earliest')
```

With this consumer we subscribe to this topic and read using a while loop, this is done in the next code block.

```python
while True:
    try:
        records = consumer.poll(60 * 1000) # timeout in millis , here set to 1 min
        record_list = []
        for tp, consumer_records in records.items():
            for consumer_record in consumer_records:
                record_list.append(consumer_record.value)
        print(record_list)
```

### 6.3.2 Publishing messages in JavaScript

Installing kafkaJS using `$ npm install kafka-client`, after the installation we can create a Producer in JAvascript and send messages to the topic, like the following code examplifies:

```javascript
const Producer = kafka.Producer;
const client = new kafka.KafkaClient();
const producer = new Producer(client);

const payloads = [
    { topic: 'ID', messages: id },
];

producer.send(payloads, function(error, data) {
    if (error) {
        console.error(error);
    } else {
        console.log(data);
    }
```

#### 6.3.3 Problems with Kafka

We were not able to implement Kafka on the server for the RaspberryPI it only works in a separate Production Code.

# 7 Improvements

## 7.1 Kafka

As already mentioned, we were not able to include Kafka into the build with the RaspberryPI due to this fact we cannot write the NFC Cards using the client-side of the web page. So this would be the next task to tackle.

## 7.2 Adding more playable Capabilities

We also want to add functionality to save playlists, albums and artists to the NFC cards and play them as well because now the functionality only works for tracks.

## 7.3 Adding Bluetooth Device Changeability

Write a script that lets you connect to new Bluetooth devices on run-time and swap Bluetooth devices, and further add this to the web page. Now you have to beforehand connect to a Bluetooth device in the command line, which is a tedious task.

# 8 References

- Web API — Spotify for Developers
- Expressjs Homepage
- Fireship.io
- Web Dev Simplified
- What is Apache Kafka
- Getting Started with Apache Kafka