WePick – A dynamic web app that creates curated playlists based on multiple user's music preferences using Spotify

Edward Eldridge

Danielis Joniskis

Keith Higgins

B.Sc.(Hons) in Software Development

April 25, 2019

Final Year Project

Advised by: Martin Kenirons
Department of Computer Science and Applied Physics
Galway-Mayo Institute of Technology (GMIT)

Contents

About this project

0.0.0.0.1 Abstract For our final year project, we wanted to create an industry standard application based on something that we all loved. In our case this was music. We felt that if we were working on something that we loved it would be more enjoyable than working on something that we didn't. The basis of our application is a personalised playlist creator which also has the capability of merging your playlist with a friend to create a playlist with both of your preferred music in it. We felt this was a fantastic idea as when the three of us would go in the car together none of use could agree on music to Liston to , perhaps if we had an app like this during the journeys , all of the music arguments we had wouldn't have happened. The main goal of this project was to create a 3 tier application , we used MongoDB as our data tier, Flask and python for our logic tier and React native for our presentation tier. These three tiers gave us the architecture for our application called "WePick".

0.0.0.0.2 Authors

Eddie Eldridge
Danielis Joniskis
Keith Higgins
Three students from the Galway-Mayo Institute of Technology

Chapter 1

Introduction

When we first began to brainstorm ideas for our project, we decided to focus our project on the theme of music. This was an obvious choice for us because its something that the three of us have a keen interest in. At the start we thought of an idea to create our own music player, but soon decided after great research that they're numerous apps like this available and if we did develop this we would just be adding another music player to the app store that probably wouldn't be used.

Finally, we thought of the idea of "WePick", a personalised playlist creator with the ability to create a community playlist with your friends. This seemed like the perfect idea and something that the three of us would regularly use if there was an app like this on the market. Although it was a fantastic idea, we now needed to come up with a development plan. We had decided before the idea of "WePick" that we wanted to create a 3-tier application but were unsure what we would use for the tiers. After about a week of research into different technology we decided to go with React for the presentation tier. The main reason for this was due to it being top 5 in the most used frontend JavaScript frameworks. Our next choice was what type of database to use for our data layer. This was a tough one because we wanted to be able to easily communicate with our database and be able query our database to send and receive Json data. For these reasons, we decided to go with the popular MongoDB. For the logic tier we decided to use Python along with flask for microservices. We then decided to use Axios which is a promise based HTTP client for communication between our database and our react front end.

Once again, a quick summery of our project, A three-tier application, using react for the front-end, Mongo for the backend and python as our middleware. Using all these technologies to create a website which allows the user to create an account and add favourite artists then to create a personalised playlist with the help of the Spotify API. The user then has the option to add another user that is registered to our site to merge their personalised playlist to create one big community playlist with a mix of both tastes of music.

Chapter 2

Context

The context of this project revolves around the use case of multiple people sitting together in a car or house and not being able to design on what music to play. One of the people can open the app and generate a play list that they can all enjoy instead of spending time on deciding what to play specifically.

The goal of our project was to develop a dynamic web application that creates curated playlists based on multiple user's music preferences using Spotify.

The project was completed over a 7-month period and this meant that the project environment didn't change drastically.

2.1 Project Reference

https://github.com/WePickOrganization/WePick

2.2 Objectives

The project required several objectives to be accomplished in order to provide a solution that worked.

- Set up a MongoDB database that will be used to store the user's preferences along with the user's email and password.
- Host the application in the cloud – using AWS. This must be done so the application can be accessed from any desktop.
- Set up a user interface using React, which will serve as the frontend for the application.

- Make use of Flask and Python to set up the backend for the application.
- Use the Spotify API to generate the playlist.

2.3 Overview

Each chapter of this minor dissertation gives different details on this group project.

The Methodology outlines the different options that were considered to design the solution. Project management and development methodologies are outlined here.

The Technology Review discusses in detail, the different technologies that were used throughout the project.

The System Design gives a rundown on each part of the system, the technologies used to build each part and how they work.

The System Evaluation section evaluates the project against the objective and proves robustness and performance of the project.

The Conclusion will be a reflection on the project.

Chapter 3

Methodology

About one to two pages. Describe the way you went about your project:

- Agile / incremental and iterative approach to development. Planning, meetings.
- What about validation and testing? Junit or some other framework.
- If team based, did you use GitHub during the development process.
- Selection criteria for algorithms, languages, platforms and technologies.

Check out the nice graphs in Figure 3.2, and the nice diagram in Figure **??**.

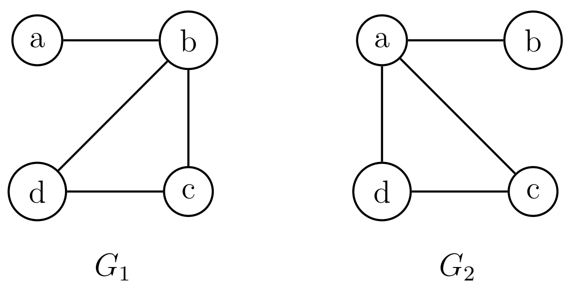


Figure 3.1: Nice pictures

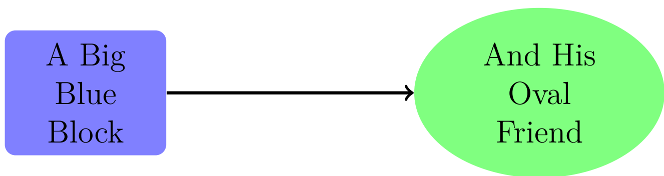


Figure 3.2: Nice pictures

Chapter 4

Technology Review

In the following section, we will discuss the technologies used in the project, how we went about implementing them and the reasoning behind our decisions.

- React
- Flask
- MongoDB
- Amazon Web Services
- TravisCI
- MochaJS
- Spotify API

4.1 React

For developing the front-end of our application, we used React. React is a JavaScript library for creating responsive and elegant user interfaces. It is an open source technology and this was one of the driving forces which led us to choose it over other popular choices such as Angular, Vue.js or Ember. We were more comfortable working with open source software as we knew the documentation and help online would be much more reliable than traditional proprietary software. React implements a component based architecture. This means that the user interface consists of multiple, abstracted **components** that each manage their own state and properties. This component based architecture promotes re-use and efficiency as when the page loads, each component is loaded only once and is subsequently rendered and re-rendered accordingly when the data of that component changes. This means that in practice, instead of the user having to reload commonly rendered aspects of a web page such as the footer and navigation bar every time they visit a new page, we only need to update the components that change. For example, if the user arrives at the home page of the website, they will load the navbar, footer and home components. On a traditional webpage, when visiting the register/login page they would have to reload the navbar and footer again. Using React, we were able to eliminate this shortcoming of traditional web design and only re-render what is required - the Login/Register forms.

This component based architecture also helped us to achieve one of our original goals of the webpage being fast and responsive to use.

This design approach implementing abstraction and re-use also means that it is easy and painless to share components with others and this greatly helped us rapidly develop the webpages in a clean and easy to understand manner. We were each able to work on separate components of the web page in parallel without massively conflicting with each others work. We were then able to render these components depending on which RESTful route the user was on. We accomplished this using React-Router, a React library that enables the rendering of specific components on specific RESTful routes.

An example of this can be seen below

```
<Route path="/Register"
  component={(props) =>
    <RegisterForm
      createState={this.createState}>{props.children}
    </RegisterForm>}>
</Route>
```

The above code means that when the user is on the route '/Register', render the component called 'RegisterForm'. We also create a state for this component and pass in the 'props' to this component. 'Props' refers to the parent component, which in this case is our Index page. We pass the current state of the parent component to it's child, RegisterForm.

As mentioned above, we used RESTful routes in our application. As such, we had needed a way to handle HTTP Requests. Unfortunately, unlike other popular front-end frameworks such as Angular, React does not include a HTTP Client to handle requests. To combat this shortcoming, we researched a couple of popular JavaScript HTTP Libraries but ended up settling on of the most popular, Axios.

Axios is a promise based HTTP client that works with nearly ever browser and has a host of features including automatically transforming requests into JSON, support for protecting against cross-side forgery attacks as well support for NodeJS. Axios was very simple to use and made handling HTTP Requests and Responses from our server painless and easy to handle.

```
// Perform Axios GET Request
// Sent to Flask server's route '/loginUser'
// Send our state variables captured by our handleChange function
axios.post('/loginUser', {
    params: {
      email: this.state.email,
      password: this.state.password
    }
  })
  .then(function (response) {
    console.log("Server Response: " + response.status)
    if(response.status==200)
    {
      console.log("Successful Login!")
      self.handleSuccessfulLogin();
    }
    if(response.status==201)
    {
      console.log("Wrong login details!")
      self.toggleError();
      self.handleFailedLogin();
    }
    else
    {
      console.log("Server error!")
    }
  })
  .catch(function (error) {
    self.handleFailedLogin();
  });
```

As you can see from the code above, implementation is simple, logical and easy to understand.

4.2 Flask

In order to serve our React files, we needed a web server. We spent a lot of time researching this aspect of the application as we knew it would be a critical part and we did not want to regret our decision down the line. Some of the contenders included Ruby on Rails (Ruby), Django (Python) and Grails (Java). In the end, we decided on Flask as it suited our needs the most. Flask is 'microframework' written in Python. It's lightweight and get's it's 'microframework' name from it's lack of standard support for traditional functionality such as form validation, a database abstraction layer or object-relational mapping tools. Instead, these tools and libraries are provided by third parties. As a result of this architectural design choice, your resulting server can be custom-made to suit your specific needs with the least amount of bloat possible. This tied in nicely with our original goal of the application being fast and easy to use. The nature of Flask's architecture meant that we only had to include what we needed and no unused features and services was included.

Although originally conceived in 2010 as an April Fool's Joke [1], Flask went on to be extremely popular being picked up by large corporations such as LinkedIn and Pinterest. A basic Flask application can be setup in as little as 7 lines

```python
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

One of the driving factors regarding our choice of Flask was that it is written in Python, a language none of us were hugely familiar with. We all thought it would be a good opportunity to learn a new language, especially since it is considerably different to previous languages we had used such as C# and Java.

Some of the Flask libraries we used in development of the application included

- Flask-PyMongo
- Flask-ByCrypt
- Flask-JWT

Flask-PyMongo enabled us to connect our MongoDB database with our Flask application in a simple and easy to use manner.

Flask-ByCrypt helped us with the encryption of user's passwords before they were entered in the database upon registering as well as subsequent decryption of these passwords when attempting to log in.

Flask-JWT provided support for securing our RESTFul routes with JavaScript Web Tokens

4.3 MongoDB

To satisfy our goal of an overall fast and efficient system, we wanted to make sure that access to the database was fast as well as secure. To satisfy this need, we chose MongoDB for our database. We considered using an relational-style database such as SQL but we had used this technology many times before and we wanted to try something new. We also were aware that MongoDB used a document-style model to store information. This tied in nicely with our choice of Python as MongoDB's document-style of storing information is very similar to Python's 'dictionary' variable type. It is also very similar to JSON Style documents which are passed in from our React front-end. Overall, this resulted in us being able toe easily pass the data between our React front-end, Flask back-end and our MongoDB database without having to hugely change the data at each step of the process.

As mentioned above, MongoDB uses a document-style model for storing information. This document style allows for many for many features such as

- Scalibility
- High performance
- Flexible and consistent
- Simple and powerful indexing tools

Another advantage of MongoDB is that like React, it's also open-source. This again played a large role in why we chose it as we knew that any questions or queries we might have about the technology would be easily answered online as developers of open-source software generally encourage this behaviour.

From our research, we also found that Mongo was considerably faster in general than traditional SQL databases, especially when the size of the tables/data was limited. However, I went delve into detail on this as the performance of this is discussed later in the System Evaluation chapter.

An example model of our MongoDB document we used can be seen below

```
"_id": {
"$oid": "5c90d0d31fd421123063d92f"
},
"name": "Edward Eldridge",
"email": "steadyeddie101@hotmail.co.uk",
"password": {
    "$binary": "JDJiJDEyJDBlZWtUeTRMS0xQRnhFV0NGMU9vT08wb3Z5TTExOFdxTTP",
    "$type": "00"
},
"spotifyUsername": "Steadueddie",
"favArtist": [
    "Nujabes",
    "Kanye West",
    "Chief Keef",
    "Gotan Project",
    "Sia"
]
},
```


In order for the application to function in the manner we intended, we needed to host the database somewhere remotely. We researched lots of ways of doing this but ended up settling on using a an Amazon Web Services Ubuntu Virtual Machine. The database was setup on a Ubuntu 18.04 system running on a virtual machine. The provided instance had no desktop or user interface so all the database set-up was done on the command line. It was set up with security in mind, meaning a securely kept SSH key was needed to access the instance and a password was required to subsequently access the database. This meant the database and it's contents were kept secure. We had all experienced security issues with databases in previous projects, whether it was a ransom-ware attempt or deleting the contents of the database so this was a priority for us.

4.4 Amazon Web Services

Amazon Web Services, a subsidiary of the well known company Amazon, provides solutions to a vast range of cloud-computing problems. Some of these services include Elastic Beanstalk (for hosting websites), EC2 (for creating virtual machines) and they also provide their own database services if you don't fancy setting it up yourself. As mentioned above, we used Amazon Web Services to host our MongoDB instance. However, we also used Amazon

Web Services to host our actual Flask application so that it could be used by anyone, without the need to download and compile the source code.

To accomplish this task we used Amazon Web Services' *Elastic Beanstalk* service. AWS describes Elastic Beanstalk as an 'orchestration' service. Essentially what this means is that they handle the large majority of the configuration, management and co-ordination required to keep a website hosted and up and running. To enable this, AWS provides a command line tool called AWS EB CLI. It's essentially a command line program that helps simplify creating, updating and monitoring your environments/applications from a local repository. AWS provides built in support for Flask applications so the setup was relatively simple. To further compliment our deployment, AWS provides support for Travis Continous Integration for seamless transition of code from a local repository to deployment.

We ensured that all our Amazon Web Services instances were on the EU-West server as we expected the large majority of our user-base to be based in Europe and this would enable better speeds and reliability. AWS also provides a lot of free services as our Ubuntu and Elastic Beanstalk instances were able to run for roughly 2 months before reaching the free tier. This is useful for people who might be curious as to what the process of hosting an application/setting up a database is like without having to pay excessive amounts of money. Once we reached our limit, we were able to supplement our credits with free credits provided by GitHub's Student Package.

4.5 TravisCI

TravisCI or Travis Continuous Integration is a continuous integration tool that works in parallel with a GitHub repository. Continuous integration can be described as follows: Developers commit code to an online repository. Once this code has been committed, the code is then ran against automated tests to verify that it won't affect the overall stability and integrity of the system or application. Once this code has passed the outlined tests, it's often passed onto to another service to be deployed such as Amazon Web Services or Heroku. In our case, we used TravisCI as our continuous integration tool. It's free, relatively easy to setup and works with any GitHub repository. To test our code before deployment, we used Mocha. Mocha is a Javascript testing framework that will be described in more detail below. Once the code passed the tests outlined by Mocha, it was automatically deployed to our AWS Elastic Beanstalk environment. This allows new features to be **quickly** and **safely** integrated with the live deployment of the application without the developers needing to fully understand the complete configuration of the

Elastic Beanstalk environment.

It also means that if there are bugs in the code, we are immediately notified by Travis via email whether or not the code managed to pass the tests. This means that bugs can be quickly squashed and less time is spent trying to figure out where the bugs came from and more time spent on developing actual new features. This ties in nicely with a lot of the core ideas behind one of the core concepts of the Agile software methodology - 'Working software is the primary measurement of progress.'

Below we can see a snippet of the Travis configuration file used to setup Travis with MochaJS and our Elastic Beanstalk environment.

```
language: node_js

before_script:
  - cd react-frontend
  - npm install
  - npm run dev

script:
  - npm test

before_deploy:
  - npm run dev
  - cd ..

deploy:
  skip_cleanup: true
provider: elasticbeanstalk
```

4.6 MochaJS

Mocha describes it's self as a 'feature-rich testing framework' that runs on NodeJS. We were all relatively new to testing, having done only a small bit of unit testing with JUnit before. As such, we chose Mocha because it was easy to use and simple to integrate with Travis. Our tests were relatively simple as we expected many bugs throughout the development of the program and we wanted to ensure we didn't get bogged down on small bugs that wouldn't cause functionality problems but might stop our new features from being deployed.

4.7 Spotify API

One of the original goals set out by us when developing the application was that it was easy to use and integrated easily with existing technologies. To accomplish this goal, we chose to use Spotify to as both a music player and a music database as we suspected that a user likely to use our service would most likely already have a Spotify account. To facilitate the communication of our application and Spotify's music service, we needed an API. There are many different Spotify APIs available in a range of languages. We decided to use the Python Spotify library called 'Spotipy'. We chose to use the Python Spotify API as we thought it would integrate well with our existing Flask server, written in Python.

The 'Spotipy' module is not made by Spotify and as such is not officially supported. This meant that documentation for a lot of the functionality was lacking and was largely user-made. As a result, this hindered our progress significantly and it took an unplanned for amount of time to correctly setup the authentication that Spotify requires to make requests on behalf of the user. In hindsight, this was a mistake as we didn't realise that there was an official JavaScript API for Spotify. This would have likely been easier to use and less complicated to setup the authentication. As a result of this, we ended up re-writing a large majority of the Spotipy module to suit our needs. This resulted in us being able to achieve what we wanted but also some unexpected behaviour on the AWS deployment. Authentication was troublesome to setup on the AWS deployment as the module uses a Python's native HTTP MicroServer module in it's authentication process and it had trouble running consistently on our Elastic Beanstalk environment.

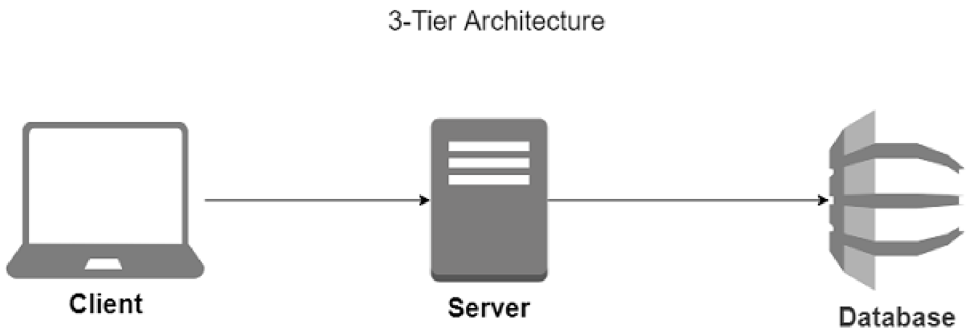
Despite these shortcomings, we were able to successfully use the Spotipy module to successfully manage and create playlists, give the user information about their own Spotify playlists as well handling any authentication required in this process.

Chapter 5

System Design

We looked at different architectures for our web application and decided to use 3-tier architecture. In doing this we were able to separate the work between the group, purposely playing to each team members strengths. Utilizing a 3-tier architecture allowed us to comfortably work on one tier without impacting the other tiers directly. It proved especially useful when a update came out for the technology at a given layer, we were able to update that layer independently without trouble.

We opted to utilise a 3-tier architecture consisting of the following components:



5.0.1 React frontend (Client)

The React front-end is what you see at the Presentation layer. This is what the user is interacting with on their own PC. On this tier input is received, and output is displayed. We used ReactJS for the front-end as there is good support for it. A great thing about React is that you can create reusable UI components. React is good for its simplicity. It makes use of JSX syntax which allows you to produce React elements more intuitively, but still also allows you to write in plain Javascript.

The React front end consists mainly of index.jsx and several separate component classes. The index page acts an entry point and is the outermost component class in the React application. The index page handles the user logging in by passing in the email from the LoginForm component. Once the details are verified, the user is redirected to the Create page which contains the ArtistEnter component. The index page handles the user logging out by clearing the logged in state and redirecting back to the Home page. The index page is where the React components are rendered visually. The Home page components displayed consist of the NavBar, Carousel, Home, GenerateButton, and StickyFooter.

There is also an app form which consists of the ArtistEnter, Logout, RegisterForm, Auth, LoginForm and Prefs.

These components are then styled with multiple CSS files.

The LoginForm component takes in the user's email and password. Once the user clicks on the submit button. An Axios GET request is performed. Sent to the Flask servers createUser route.

The Logout component simply clears any state held by its parent component and then redirects to the Home page.

The Home component acts as the component between the NavBar, Carousel and StickyFooter components. The Home component is where the GenerateButton component is rendered.

The NavBar component contains the navigation bar that is always rendered in the application. The user can follow the links on the NavBar to the Home, Create and Profile pages respectively.

The Carousel component is used to render an image carousel on the Home page. The carousel cycles through 7 different images and is a style choice which gives a more professional look to the application.

The Auth component handles authorization.

The RegisterForm component renders the form which is displayed to the user when they are registering a new account in the application. It takes in the user's name, email and password. On submit the user's information is routed to auth.

The ArtistEnter component contains the form which allows the user to generate a play list based on their preferences. Once the user has logged in successfully, they can enter four different artists of their choosing and their email address. The user can then generate a playlist by submitting the form.

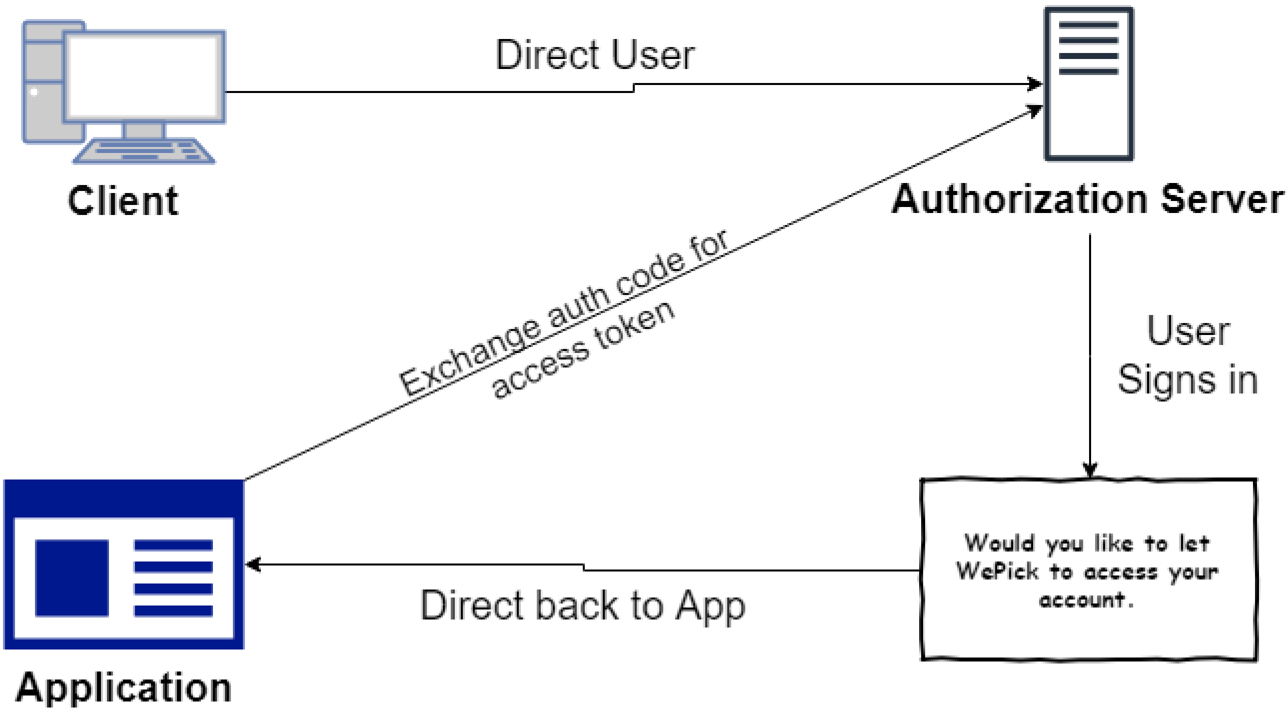
The Prefs component is what is used to handle the user preferences.

The StickyFooter component renders the footer which is present on all pages of the application. The footer acts as a second navbar and contains information about the organization.

5.0.2 Python backend (Server)

The python backend is what is present at the Application layer. This is the server which is made using flask and python. On this tier the logic is processed, and calculations are made. SpotipyAPI.py is where the playlist generation is handled behind the front-end. When an artist is searched, a lot of information about the artist is returned i.e. Hometown etc. The JSON data that is returned is where their artist ID is stored. This in-turn allows us to grab that artist ID and use it in generating the playlist.

We make use of OAuth 2.0; this is the industry standard protocol for authorization. We use OAuth2 to authenticate the user through Spotify. When the user is trying to login to their account on our application, they are in turn brought to a login dialogue through their Spotify account. Once they have logged into Spotify they are authorized and can continue with our application.



The user starts out by going to the application. The application then sends the user to the authorization server. The user logs in here and the authorization server sends the user back to the application. The application then makes a POST request to the authorization server to get an access token.

Application.py is the main backend server content file in this project. This is where an instance of the DatabaseConnector.py is created. The Flask server is created here and pointed to the static file within the frontend to serve the HTML. The Mongo object is created here along with the Flask. The getSpotifyUsernameByEmail() function gets the Spotify username from the database. The default route of the application is here, and this is where the index.html file is rendered.

The CreatePlaylist() function is where the playlist is generated. The function first gets the JSON data from the request and in turn gets the users from that JSON data. Any empty strings from the list of users is filtered out and the list of users is then converted into an array. The first user in the list is always the home user currently logged into the application.

The favourite artists for all the users in the array are then retrieved from the database. The collection of artists is then combined into one large list and any duplicates of artists are removed. A random sample is taken from all the artists collected from the database.

The Spotify username is taken from the currently logged in user. Their username is passed onto authentication and a token is returned. The artist IDs of the random sample are retrieved and those IDs are passed on to the GeneratePlaylist() function. A play list is then created.

The getArtistDB() function sends a GET request to return the artists from the MongoDB database based on the users email address.

The sendArtists() function sends a POST request to update the users favourite artists.

The loginUser() function handles the user logging into the application using a POST request. Once the login data comes in from the login form component the loginUser() function checks the JSON data, making sure it is in the correct format. The email is checked against the database to see if it is present. If this user's email is present, the password is then checked to see if it is correct. If the user's password checks out, that password is deleted, and the user is given an access token in exchange. If, however the user does not have a registered email in the database, the user is notified on the client end. If the user does not supply the correct login details the user is notified also.

The showUser() function uses a GET request to show specific details of a user. A query is taken from the HTTP request argument. The data is retrieved from the MongoDB database using this query and is displayed as JSON format.

The showAllUsers() function is similar to the showUser() function naturally. This function also uses a GET request and receives a query. It then queries the MongoDB database and retrieves the data. This data of all users is added to a list and is then printed out in the console.

The createUser() function handles the user creating a account using a

POST request. Once the user details come in from the form component the createUser() function validates the JSON data making sure it is in the correct format. The password is encrypted before it is inserted into the MongoDB database. Once the user details have been successfully added to the database the user on the client side is notified. If the user does not input parameters correctly, they receive an error message.

The deleteUser() function uses a DELETE request to delete a user from the database. Once the user data is in the correct format and the user details are present, the record can be deleted from the database. If there is no such user present, then an error is thrown.

The updateUser() function uses a PATCH request to update the user's details. First the data is checked that it is in the correct JSON format. The MongoDB database is then queried to update the record if it is present, otherwise an error is thrown.

The auth() function handles the Spotify authentication required , using a POST request. Once a GET request is received a token is generated in the browser. If the data is in the correct format, the token will be successfully generated and authentication will pass, otherwise it will fail.

The getSpotifyStats() function uses a POST request to display the current users statistics. A user's email is taken in, the stats are added to a list and displayed as JSON data.

5.0.3 MongoDB (database)

The MongoDB is what is at the Data layer. On this tier data is stored and managed. Once the database was set up, we wrote DatabaseConnector.py. We loaded in the IP address and password for the database from a config file found locally on the machine. Once the database was created, we could view the collection of users in the database. We made use of functions to add, show and delete users. We also wrote a function which could be used to show a user and their favourite artist from the database.

Chapter 6

System Evaluation

In this chapter, we will discuss many aspects of the software. We will break the evaluation down into 4 headings.

- Robustness - How well the software can deal with problems, it's ability to handle change etc.
- Performance - Space and time complexity of the software, responsiveness etc.
- Security - How vulnerable is the application to attacks and security risks?
- Overall evaluation - Where the project succeeded/failed, limitations in our approach and technologies etc.

6.0.1 Robustness

Robustness in regards to software can be defined as the ability of the software to cope with errors during execution and cope with erroneous input. Measuring the robustness of a system is difficult as no system can be considered completely robust. However, there are certain methodologies and practices we can implement to increase the robustness of a system. *Behdis Eslamnour and Shoukat Ali* discuss these metrics in their paper entitled *Measuring robustness of computing systems*[2] Some of these proposed metrics include

- Error Handling/Error Catching
- Time between failures and time between recovery

6.0.2 Error Handling

Error handling refers to how a system handles errors should they occur. This can be as simple as logging the error in the console to rolling back the system to a previous stable release. It is a vital aspect for any system from both the perspective of the user and developer. As a developer, implementing proper error handling and catching is an important aspect of development as it enables the developer to work more effectively and efficiently as less time is spent fixing and locating the cause of bugs if proper error handling is done pre-emptively.

For the user, it's vital that error handling is done correctly as incorrect error handling can have a significant impact on the end-users experience of the system/software. For example, if a user attempts to login to a system and their login details are incorrect, a meaningful way of handling this error would be to notify the user that their login details are incorrect. If this is not done, the end user won't understand what's happening and why they can't use the system and as a result of this will become frustrated/unhappy with the service. As such it's important to make sure any critical problems such as this are caught and handled in an appropriate manner. However, it's also important to remember that complete robustness can not be achieved and as such one must try and figure out the most important and likely errors to handle. When making these decisions, one can consider many factors when deciding what errors to handle such as the risk of not handling the error (application crashing, unexpected behaviour, potential security risk, loss of data etc.) , how much time will be required to handle the error and the likelihood of the error occuring. However, as mentioned above complete robustness can not be achieved as to even consider the above factors, one must have the foresight to see that the error will occur in the firstplace.

Below I will discuss some error handling that occured in our project and how it contributed to the overall robustness of the system.

Example 1 - Login/Register In the image below we can see how error handling and error catching can produce meaningful output and guide the user through login/register scenario of our application

—- user login diagram here —-

Example 2 - Authentication with Spotify In the image below we can see how error handling and error catching can produce meaningful output and guide the user through authenticating our applicatin with Spotify.

— authentication diagram here —-

The above error handling was done on both the server and client-side of the application. Both Python, MongoDB and Pythn provide useful tools for error handling/catching. Some examples include *Error boundaries* in React, *Try/Catch/Raise statements* and *Schema validation* in MongoDB.

- Error Boundaries - Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.
- Try/Catch/Raise statemetns - These commonly found in most programming languages and Python is no exception. We used these to catch certain errors such as HTTPErrors, null value errors, MongoDB errors and many others.
- Schema Validation - To ensure the data being passed into the Mongo database was in the correct format, we used a JSON Schema to validate our inputs against before passing it to Mongo. In the below example, we can see that for a user to be validated correctly, the minimum required properties are an email and password. This ensures that user's cant create accounts without these two required properties.

```
user_schema = {
  "type": "object",
  "properties": {
      "name": {
          "type": "string",
      },
      "email": {
          "type": "string",
          "format": "email"
      },
      "password": {
          "type": "string",
      },
      "spotifyUsername": {
          "type": "string",
      },

  },
  "required": ["email", "password"],
  "additionalProperties": False
}
```

6.0.3 Time between failures and time between recovery

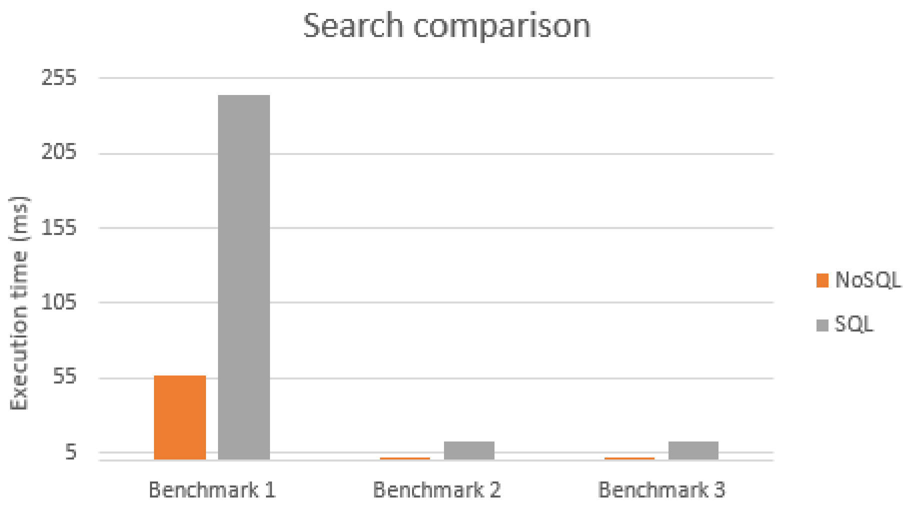
Time between failures and time between recovery are too very useful metrics in determining the robustness of a system. Systems that have a short mean time between failures and mean time between recovery either have excellent error handling or are designed in such a way that errors do not occur frequently. A good way of measuring this could be the downtime of your application over the course of a year. For example, on average, the top 50 e-commerce websites experienced 99.03% uptime. Over a year, 99.03% uptime would result in 3 days 15 hours and 39 minutes of downtime. 32 of these websites experienced 99.99% uptime. 9 out of 10 of users encountering a website that is not up will choose to use a competitors website [3]. From these statistics, we can see that time between failures anre time between recovery are extremely important aspects to any system/application. As such, they can be very helpful metrics when trying to measure the robustness of an application.

In regards to WePick's failure and recovery time, since the project was hosted on AWS' Elastic Beanstalk, there have been little to no outages as due to the continous integration solution integrated upon deployment, it was not possible for broken code to be deployed as any code proposed to be deployed had to first pass tests that were outlined by Travis CI and any problems that did arrive were dealt with quickly as Travis instantly notified us of any problems which we were able to fix quickly and promptly.

6.0.4 Performance

When planning and designing the project, one of our main aim's was that we wanted the project to perform operations and tasks quickly as well as being responsive to user input. Taking these considerations into account, we choose Python and MongoDB for our backend as they are generally considered lightweight and easier to handle performance than say something like Java. We chose React for the front-end of the application as React is well known for it's performance and responsiveness. This generally comes from the fact that React applications usually implement a single page design. React works by creating multiple components that can be re-used multiple times in the application. For example, in our application, instead of creating multiple pages with the same code (Navbar, Footer, Forms etc.), we can create seperate components for each of these items and only re-render what is required. So if the user needs a different form only that form will be re-rendered and the Navbar and Footer don't need to be re-rendered This choice of technology helped us achieve our overall goal in regards to perfromance and I think it was a good choice.

Our choice of MongoDB as a database was also important to the overall perfromance of our application as MongoDB is considered much faster than SQL and other relational databases in certain scenarios. This is because of it's non-relational, no-SQL way of storing data. Instead of storing data in rows, columns and tables, Mongo works by using a document style architecture. These documents are structured similarly to a JSON document and as such are easy to extract data from and manipulate without navigating through large sets of data like you would in a traditional SQL database. Generally, this performance increase can be seen most when you start to scale up the size of your database. For our application, it was important to be able to access user's favorite artists quickly so that they didn't have to wait a long time to generate a playlist.



From the above graph, we can see that when performing a search in the database, NoSQL databases like Mongo are considerably faster than a traditional SQL database /citeSQLvsNOSQL

6.0.5 Security

Securit was an important factor for us to consider when designing and creating the application as we had all experienced problems with group projects in the previous year. For example, many students dealt with problems where database tables were wiped and/or hijacked and held ransom to the owners. As such, we did not want to deal with such problems and thought it would be a good oppurtunity for us all to learn a bit more about how one goes about securing an application in a real world context. We also wanted user's information to be secure and not easily accessible by anyone.

To ensure our RESTful routes were not accessible to anyone, we implemented JWT Tokens into our application. A JWT Token or JSON Web Token is an open-standard that defines a way for securely transmitting and verifiying information between systems. JWT's are signed using a secret key or a public/private key using RSA encryption.

The way this ends up working in the application is as follows. When the user successfully logs in, a JSON Web Token is generated and each subsequent request made by the user will include that web token. Setting up the system in this way means that to perform a HTTP request on one of our many RESTFul routes, the user first has to verify that they are an authenicated user. This prevents from people performing actions on routes that they should not have access to.

A traditional JSON Web Token consists of a Header, Payload and Signature in the following format

`header.payload.signature`

The header details what type of token is being sent and the algorthim used to generate the token. The payload contains the information about the person sending the token and what they would like to do. The signature is essentially an your header and payload encrypted and is used to verify the payload wasn't altered during transit as well as verifying the sender is who they say they are.

An example token can be seen below


```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1Ni
J9
.
eyJpc3MiOiJodHRwOi8vdHJ1c3R5YX
BwLmNvbS8iLCJleHAiOjEzMDA4M
TkzODAsInN1YiI6InVzZXJzLzg5ODM0
NjIiLCJzY29wZSI6InNlbGYgYXBpL2J1e
SJ9
.
43DXvhrwMGeLLlP4P4izjgsBB2yrpo8
2oiUPhADakLs
```


To implement this technology in our application, we used Flask-JWT, a simple Python library that provides JWT Support for Flask applications. A simple example of how this technology was implemented can be seen below.

```
# Authentication Stuff
@application.route('/refresh', methods=['POST'])
@jwt_refresh_token_required
def refresh():
    current_user = get_jwt_identity()
    ret = {
            'token': create_access_token(identity=current
    }
    return jsonify({'ok': True, 'data': ret}), 200

@jwt.unauthorized_loader
def unauthorized_response(callback):
    return jsonify({
        'ok': False,
        'message': 'Missing Authorization Header'
    }), 401
```

As well as ensuring API calls were authenicated, we also made sure that our database and user information was secure. A secure admin account was created on the database, requring a password to perform any actions in the databse. This way, no operations could be performed on the database without knowing both the IP address of the server the database was hosted on and the password required to gain access. These two important pieces of information were known only to us and weren't available anywhere else.

We also made sure that any important user information such as passwords were encrypted before being entered into the databse. For this, we used another Python library for Flask entitiled Flask-Bycrypt. Using this library, we can generate a hash of the user's password and store this in the database instead of storing their password in plaintext. Then, when checking if the user's login details are correct, we compare the generated hash of the passwords instead of the actual passwords themselves. This enables a secure login scenario and prevents passwords being revelead to anyone.

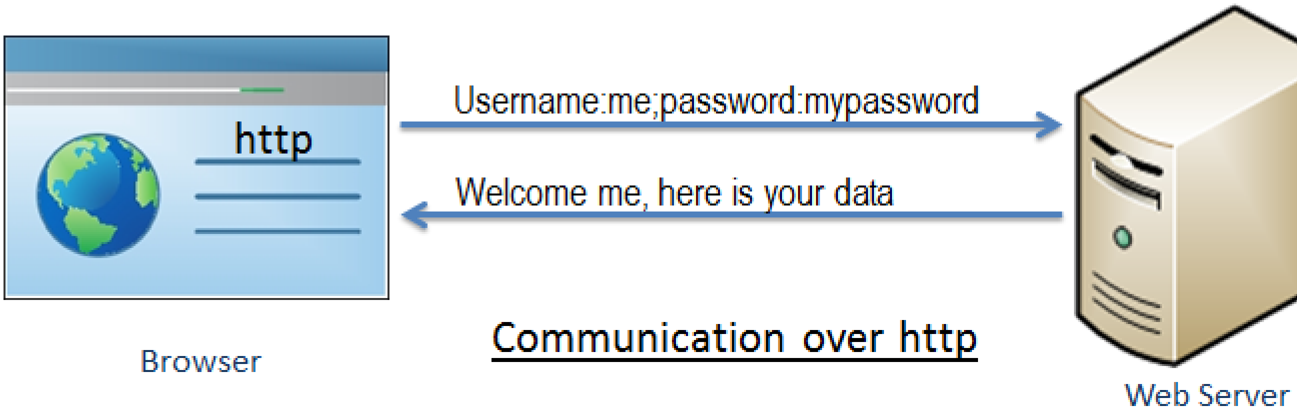
To improve overall security, we also chose to host our application on a different instance than our database. This meant that if one were to be compromised in any way, it would prevent the other in turn being compromised.

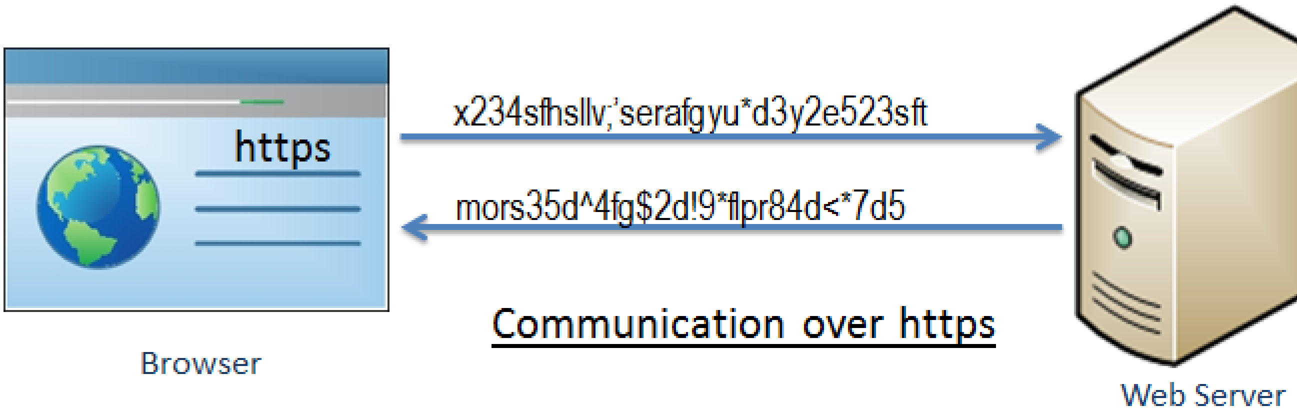
6.0.6 Improving Security

6.0.6.1 HTTP/HTTPS

One shortcoming of our project is the lack of a secure Hyper Text Transfer Protocol (HTTPS). HTTP can be described by Mozilla's standards /cite-HTTPviaMozilla as an application-layer protocol for transmitting data and documents. It was originally designed for communication between web browsers and web servers but it has since exceeded it's original purpose. It follows a client/server model. The client makes a request to establish a connection and waits for a response from the server. Despite this however, it is a /bfstateless protocol. This means that once communication between server and client, the server forgets all information regarding the communication. It can be used on any reliable transfer layer such as TCP/IP or UDP.

The difference between HTTPS and HTTP is that HTTPS is considered 'secure' where HTTP is not. Essentially, the difference between the two can be illustrated cleary in the following diagram.





In standard HTTP, the information is transmitted in hypertext format whereas with HTTPS, the information is first encrypted before sending the information. The issue with standard HTTP is that the data is transmitted in plaintext. This means that anyone who can 'see' the data or is monitoring the network can potentially intercept this data and/or modify it before being passed on to the server. This can lead to serious security issues. This could be potentially devestating to an application/system that relies on sensitive data such as passwords or bank/card details.

HTTPS guarantees that even if the message is intercepted, it would be of no value as it is encrypted. HTTPS provides these guarantees using a security standard known as SSL or Secure-Socket Layer. SSL is a standard for establishing an encryped link between a client and server. To first use SSL, one must aquire an SSL certificate. These certificates are small files that bind a domain name/IP address/hostname to an organizational identity and location.

6.0.6.2 But who exactly has the authority to issue these certificates and how can we trust them?

To issue SSL certificates and guarantee their authenticity, one must become a Certificate Authority. This title is usually limited to private companies and governments that have proven that the certificates they issue are secure. The more secure certificates they authorize, the more certificates they are able to distribute. This means that when aquiring an SSL certificate, you can guarantee it's authenticity. A certified certificate should contain your domain name, company name, address, city, state, country. It also contains an expiration date upon which the certificate must be renewed as well as the Certificate Authority that issued the certificate in the first place. All these factors provide some kind of accountability in regards to the certificate if something were to go wrong. Expiring certificates is a common occurence, even for large companies. According to GlobalSign /citeGlobalSign, between just October 2017 and February 2018, large organizations such as LinkedIn, PokemonGO the British Conservative Party and astonglishly The White House have all let their SSL certificates expire. Any users arriving on these websites would have been instantly greeted with a warning that these websites are not secure and sensitive information could have possibly been leaked in the process. For something like a goverment website, this kind of behaviour is a perfect example of why we have SSL and HTTPS in the first place and why it's vital that these systems are maintained and updated.

6.0.6.3 How does SSL work?

Let's use a traditional client/server architecture as our example. The browser/client connects to the server secured with SSL. The server will then prompt the client to identify itself. The server then sends a copy of it's SSL certificate. These certificate is then verified by the client. If verification is successful and the client is comfortable sending messages, it sends a response to the server which consists of a digitally signed acknowledgement which initiates an SSL encryped session. Any data exchanged between these two parties is now sent over the Secure-Socket Layer established by the client/server.

6.0.6.4 Benefits of SSL/HTTPS

We can describe the overall benefits of SSL/HTTPS as the following

- Utilize HTTPs, which elicits a stronger Google ranking.
- Create safer experiences for your customers.

- Build customer trust and improve conversions.
- Protect both customer and internal data.
- Encrypt browser-to-server and server-to-server communication.
- Increase security of your mobile and cloud apps.

6.0.6.5 Why didn't we implement SSL/HTTPS?

Due to the nature of aquiring an SSL certificate, one usually has to purchase a certificate and pay a yearly fee for using it. We made a decision based on the information stored by our application and by factoring in the cost, decided that we didn't think it would be critical if HTTPS wasn't implemented. We understand it makes our application insecure, but we also understand why it makes it insecure and how this could be dealt with appropriately in future projects.

Chapter 7

Conclusion

About three pages.

7.0.1 Original Idea vs. Actual Implementation

Our original idea/plan for this project was to create an application that created curated playlists based off multiple user's music preferences. This was our original idea before we had chosen any technologies or done any kind of design. Limiting the scope to this simple statement, I feel like we achieved our original goal that we set out in late September. Up to 6 user's can go on our website, create an account, enter their music preferences and create a curated playlist of music based on their and their friends preferred music. We also wanted the website to be simple and intuitive to use without the user feeling overwhelmed or confused as the original purpose of the program was simple and as such the website should be simple to use. We focused heavily on this as we all felt like that websites can be extremely overbearing and complicacted when sometimes the user just wants to perfrom a simple task and having to go through hoops to perform this simple task can be extremely frustrating.

7.0.2 Learning Outcomes

7.0.3 Shortcomings/Oppurtunities

7.0.4 Closing Statement

7.0.5 Acknowledgements

- Briefly summarise your context and ob-jectives (a few lines).

- Highlight your findings from the evalua-tion section / chapter and any opportuni-ties identified.

Bibliography

[1] A. Ronacher, “Opening the flask.”

[2] B. Eslamnour, “Measuring robustness of computing systems,”

[3] Pingdom, “Downtime for the world’s top-50 e-commerce websites.”