OLLSCOIL TEICNEOLAÍOCHTA
BHAILE ÁTHA CLIATH
TECHNOLOGICAL
UNIVERSITY DUBLIN

Sample Dissertation #2

(Front Matter removed)

Table of Contents

Table of Figures

1. Introduction

1.1. Project Background

Media distribution platforms such as Spotify and YouTube have drastically changed the music listening habits of people worldwide. For the first time in history, companies can see what music you are listening to and when you listen to it. What music you listen to sometimes, what music you listen to all the time and everything in between. In the past, music was spread by word of mouth or by wearing wristbands of concerts or festivals, this meant word travelled slowly and discovering new artists was a lengthy process. The radio stations knew when and what songs were being played on the radio and the record companies knew the number of CD, cassette or vinyl sales and so had they had a reasonable insight into the listening habits of people, but nothing on the scale of today's digital media giants. Due to the rapid growth of these media distributors, the music analytics industry today has an estimated worth of over two billion euro.[1] At first glance it may appear these companies are taking business away from the record companies by giving artists the means to self-distribute and promote, but the data they are generating about artists are making it easier for record companies to pick out artists who are generating the right "buzz" in the music scene and sign them to contracts. Metrics such as Twitter followers likes on Facebook, sales via Ticketmaster or Bandcamp and listens on Spotify can give record companies accurate predictions of future prospects. Spotify has a large effect on the new artists listened to by its users, the artists Spotify recommends are provided by its powerful recommendation systems.

Recommending the right music to users is a large part of what makes Spotify such a popular and successful platform. Both YouTube and SoundCloud provide users with platforms to listen to music, however, YouTubes recommendation system is not solely intended to recommend music and so is cluttered with other non-related material and SoundCloud does have a recommendation system, but neither is as popular as Spotify for listening to music. This in no small part is due to the extremely large corpus of data Spotify has gathered from its millions of users. The effective implementation of the right data analysis techniques provides Spotify with invaluable insight to its customers listening behaviours, which enables them to recommend the right songs to its users. Spotify's recommendation system combines the data from all users into one huge dataset that predictions are made against. The model examines the listening behaviour of users and if their listening behaviours are largely similar their tastes are considered to be similar. Conversely, if two songs are listened to by the same group of users, they probably sound similar. This kind of information can be exploited to make very accurate song and artists recommendations.

Spurred on by the success of companies such as Spotify and YouTube recommendation systems have become one of if not the most successful and widespread implementations of machine learning technologies in business today. They have become the backbone of some of the largest companies in the world. According to a study by McKinsey, thirty-five percent of what consumers purchase on Amazon and seventy-five percent of what they watch on Netflix come from product recommendations based on strong algorithms. [2] One of the most popular implementations of a recommendation system is a collaborative filtering model. This is due to its accuracy, efficiency and scalability some of the most successful companies, such as Amazon and Spotify implement either a purely collaborative filtering approach or hybrid approach that includes collaborative filtering and another recommendation method. The other method is generally supplemental and intended to mitigate the cold start problem associated with the approach.

The value of a well-designed recommendation system cannot be understated, in the age of huge data influx it is vital companies can efficiently analyse this data and utilize it to accurately predict the behaviours of their customers so they can continue to offer them products or services they are interested in. With the plethora of digital stores, media vendors and social platforms in the marketplace it is critical that companies differentiate themselves from their competitors. The best way for this to be achieved is to provide the user exactly what they want when they want it, failure to do so may result in loss of sales or user disinterest.

A primary use case of a robust recommendation system is to produce increased customer engagement. This is accomplished by providing users with the right recommendations. Customers can become overwhelmed when provided with too many recommendations, a robust recommendation system not only recommends the right products but the right quantity and variety of those products. This concept of increased customer engagement can be witnessed in full effect on the world's largest video streaming platform YouTube. Its algorithms are specifically designed to keep customers engaged by recommending new videos tailored to pique the customer's interest and keep them on the website generating ad revenue for the business.

1.2. Project Description

This project will implement an item-based collaborative filtering recommendation system that produces personalized live music event recommendations to users. In order to build a robust live music event recommendation system, a large corpus of historical data and data specific to each user is needed to train the model. This involves constantly gathering data from the user regarding the artists they are listening to and their location and incorporating these key pieces of information into the recommendations that are produced. The finished recommendation system should be able to gather information on the user's musical tastes and their location and produce recommendations of live music events that are of interest to the user.

The interests of each user are learned by retrieving playlist information from their Spotify account. All collaborative filtering recommendation systems require that items are rated by users, these ratings are used to gauge a personal interest in an item. Ratings can be found in many forms depending on what type of item the system is trying to recommend. For example, purchases or star ratings might be particularly useful when recommending new products to a user while likes and clicks might be indicative of what content should be rendered to a user on a social media platform. As this system will be providing live music event recommendations for artists the rating metric used is the number of tracks the user has for each artist in their playlists. The more tracks a user has for a given artist the higher the user has rated that artist.

Collaborative filtering recommendation systems use scores of data and powerful algorithms to find latent similarities between users and items and so the ability of the system to produce accurate recommendation relies heavily on the quality and volume of the data it is trained with. A large corpus of authentic data will be used in the training of this system it will be the genuine Spotify playlist contents of tens of thousands of users equating to an initial data set of nearly thirteen million data points.

The Web and mobile applications are constructed in a clean uncluttered style and ensure adherence to the six fundamental principles of design balance, proximity, alignment, repetition, contrast and space. This provides the user with an enjoyable user experience while still implementing all the desired functionality. The Web and mobile applications will be the main acquisition points for new user data, the constant acquisition of data is inherent to the success of the recommendation model. The data acquisition process will be efficient and reliable ensuring accurate up to date information is attained for all users. The Web and mobile applications will also provide up to date details on all live music events taking place in the user's location. In order to provide maximum usability, the applications will provide the user with a means to save events they are interested in for later reference and the ability to purchase tickets from Ticketmaster, listen to the artists using Spotify, or watch their music videos using YouTube. For any events the user has saved they will have access to a routing feature that will direct the user to the location of the event venue.

1.3. Project Aim and Objectives

This aim of this project is to bring the benefits of the recommendation model to the live music arena in a clean, familiar and well-designed interface. The lack of promotion and therefore awareness of live music events is prevalent throughout the industry with only the highest earning artists receiving the necessary amount of promotion to make fans aware of their shows. A common theme that has been witnessed is for fans to only become informed of a live music event after it has occurred. This leads to frustration with the effectiveness of companies to promote the artist and their shows, most of whom rely heavily on the income from their live

shows due to the current state of affairs in the digital music industry. A platform that provides up to date recommendation on live music events would benefit the fans, the artists and in turn, the businesses associated with the artists. This project attempts to address these problems by providing such a platform.

Several objectives were outlined at the beginning of this project that needed to be met for the project to be successful. The initial objective involved finding a sufficient source of historical data pertaining to the listening habits of Spotify users and their playlist contents. A method of retrieving the playlist contents of new users was also needed. After those objectives were achieved it was then necessary to prepare and combine these data sources in a format that is employable by the recommendation model. When the data acquisition had been completed and the data had been prepared it was essential that the development of the recommendation model took precedence. Several techniques and machine learning algorithms needed to be tested and researched in order to find the approach most suitable for this problem. Upon completion of the machine learning model, the final objectives were to build the Web application and the mobile application. The goal was to put the focus on the Web application with the mobile application providing supporting functionality. The construction of the Web application needed to be separated into distinct sections the back-end and front-end. The back-end would implement the logic of the application, orchestrating the various components and providing connectivity to the recommendation model. The front-end would encompass the entirety of the client-facing aspects, including any of the tools or features accessed by the users. The mobile application was intended to provide all the feature of the Web application by accessing the back-end functionality of the application via API and then rendering this content in a mobile friendly manner.

The first objective is gathering historical music listening habits and playlist contents data. The data will be used to train the model to predict artists a user may like based on the values attained for other users and artists in this dataset. As such, the dataset that will be adopted for this project is the #nowplaying dataset. This is a Twitter-based project that leverages social media for the creation of a diverse dataset which describes the music listening behaviour of users. [3] Twitter users regularly post information about the music they are currently listening to which was the basis for construction of this dataset. A subset of this massive thirty-six-gigabyte dataset, the playlist dataset will be used for this project. The playlist dataset is based on a subset of users that publish their #nowplaying tweets via Spotify. The users in this subset publish the songs they are listening to as an embedded link to Twitter the metadata from which is used to build up playlist information from the user. The playlist dataset is a CSV file that contains just shy of twelve million entries in the following format, a hashed user name followed by an artist name a track name and a playlist name. The second requirement that needed to be met is the ability to gather information on the listening habits and playlist contents of new users. This data needs to contain information on the number of artists and tracks the user has

saved in their playlists. In this project that data was attained by utilizing Spotify's APIs to gain access to the users Spotify account and retrieve their playlist data.

After the data had been attained it was necessary for it to be pre-processed and cleaned. The data in the #nowplayling dataset needed to be aggregated based on users and artists in order to get it into the required format of user, item, rating for the recommendation model. The dataset was aggregated down to the format of a hashed user name, followed by an artist name and a track count signifying the number of tracks a user had for any given artist. This process alone reduced the dataset from close to thirteen million entries to just under three and a half million entries. Further cleaning and processing were performed on the dataset to remove any malformed entries and to set the rating scale from one to ten thus limiting the maximum score a user can give to an artist. Such is another pre-requisite of a collaborative filtering model which requires a fixed scoring metric. Enforcing this type of rating system helps to deal with any outliers in the dataset and ensure there are no entries for artists that are unusually high which could impair the ability of the model to make accurate predictions. The playlist data was retrieved from the new users is in the form of an API response and so pre-processing similar to the historical dataset was required before it was utilized in the recommendation model.

The next objective of this project is to build the recommendation system. The first step to achieving this is researching the type of recommendation system to implement and then deciding which machine learning algorithm to use. The algorithm must produce an effective model that can efficiently deal with very large volumes of data and perform fast enough to operate in real time. The decision process for choosing the recommendation model and algorithm followed a CRISP-DM approach by first focusing on the data and the problem that needed to be solved before choosing the model or algorithm.

The final objective of this project was the development of the Web and mobile applications. This portion of the project needs to orchestrate the entire functionality of the project incorporating the recommendation system into an interactive front end that allows users to retrieve their recommendations. The first step in achieving this is the development of the back-end API endpoints, business logic and database connectivity. Once the back-end functionality has been implemented the next objective is constructing the front-end of the Web application and mobile applications to provide users with the live music events that are generated by the recommendation model.

1.4. Project Challenges

1.4.1. Data Acquisition and Preparation

One of the challenges faced in this project was the acquisition of a sufficiently dense dataset and the subsequent cleaning and processing of that data. A plethora of different datasets were examined before the #nowplaying was decided upon, many of the datasets that were analysed prior to the #nowplaying dataset were too small and would not have produced a diverse and realistic recommendation model. Cleaning the data was necessary to ensure the model could be accurately trained, the data preparation included several steps as mentioned in the project objectives. Firstly, removing or correcting any malformed data. Many of the artist's names were empty or populated with quotation marks or surrounded by special characters these entries were removed or edited. Choosing the scale of the rating metric to optimize it for the recommendation system was also tricky as there is no specific scale required for the algorithm the decision and came down to an educated decision based on analysing the data and sentiment analysis from peers.

1.4.2. Researching and Implementing the Recommendation System

Researching the multitude of different recommendation systems and choosing the correct one was one of the most challenging aspects of this project. There are a wide variety of implementations available when building a recommendation system each having their own advantages and disadvantages. The final decision on what approach to use was based on the analysis of industry standard approaches to similar problems, information discovered through reading research papers and the implementation that won the Netflix prize. The Netflix prize is a one-million-dollar prize awarded to the winners of a competition to improve Netflix's current recommendation system by reducing the RMSE (Root Mean Square Error Value). After the type of recommendation system had been chosen the best-suited algorithm for the problem had to be chosen. This was chosen much in the same way the type of system was. However, some comparison testing was also implemented to verify the cases made in the researched papers. The algorithm that would be chosen needed to work very well with large datasets and sparse matrices due to the nature of the data and the model chosen.

1.4.3. Programming Language and Framework Selection

The research and selection of what programming languages and frameworks that should be used in solving the problem of building a Web application, mobile application and recommendation model was challenging. This research was performed with the aim of finding a versatile language that could handle as many aspects of the project as possible to mitigate against compatibility problems but that also provided a familiar development environment. The frameworks chosen needed to be robust and scalable and provide languages for

developing back-end and front-end features that followed a model view controller (MVC) approach to their architectural design. They also needed to be able to efficiently communicate with each other and the database which will be hosted separately in the cloud and store all user-related data. The language and frameworks chosen played a pivotal role in choosing the integrated development environment best suited to this project.

1.4.4. Separation of Concerns

Another aspect of this project that posed a challenge was ensuring a separation of concerns was maintained between the different aspect of the application. It was essential the recommendation model be developed as a stand-alone application as it is operationally and computationally costly to run. Incorporating the model into the back-end of the main application would have drastically reduced the performance of the system and made the application far less scalable. This posed the problem of what aspects of data gathering and preparation should be handled by the recommendation model and what should be handled by the Web application back-end. Deliberate reasoning and design choices govern the location of all pieces of functionality throughout the codebase to ensure the MVC architecture is adhered to and the application remains as modular and efficient as possible.

1.4.5. API Request Limitations

This project relies heavily on its ability to access data via APIs provided by Spotify and Ticketmaster. It was challenging to develop an application that stayed within the minimalistic request limit provided for the free tier use of these APIs. Ticketmaster not only put a limit on the number of requests an application can make per day but also the number of requests it can make every second. This request per-second limitation has the most noticeable effect on performance on the system. Mitigating the effect of this policy meant compromises had to be made in the features and functionality of the Web application.

1.5. Thesis Roadmap

- Literature Review – Investigates the various areas of research, these include articles or papers, recommendation systems, machine learning algorithms, most of the technologies used and any existing solutions to the problem.

- Design and Architecture – Demonstrates the design and architecture of the different aspects of the project. Including the front-end, back-end, mobile application, recommender system and deployment.

- Development – The code used to create the components and functionality discussed in the Design and Architecture is demonstrated. As such it details the development of the significant features in the project.

- Testing – The various methods of testing used to evaluate the performance of the system, including ad-hoc, user experience and unit testing.

- Conclusions – The conclusions drawn from the project are outlined and the planned work for the future is detailed.

2. Literature Review

2.1. Introduction

In this chapter, some of the key areas of research that are important to this project will be presented. The development of this project involved continual research which led to ideas, plans and decisions mutating from what they were at their initial point of conception. The key areas of research that were investigated for this project are recommendation systems, machine learning algorithms, Web application technologies, mobile application technologies, deployment tools and cloud services. Any decisions made regarding the design or development of this project that is influenced by this research will be emphasized.

2.2. Research Papers and Articles

2.2.1. Basic Approaches in Recommendation Systems

In their 2014 paper *"Basic Approaches in Recommendation Systems",* Alexander Felfernig, Michael Jeran, Gerald Ninaus, Florian Reinfrank, Stefan Reiterer and Martin Stettinger detailed the basic approaches to a recommendation system including collaborative filtering, content-based filtering and knowledge-based filtering. The paper began by discussing the principles of the underlying algorithms used in recommender systems. It then provided insights into what recommendation technology to use in a certain application context. Thereafter, the paper provided an overview of hybrid recommendation approaches which combine basic variants. [4] The paper concluded with a discussion of newer algorithmic trends, especially critiquing-based and group recommendation approaches. The paper overall detailed the wide variety of possible applications of recommendation systems and provided very useful insight as to when a specific recommendation technology should be chosen and what algorithms work best with each approach.

2.2.2. Incremental Singular Value Decomposition Algorithms for Highly Scalable Recommender Systems

In their 2002 paper *"Incremental Singular Value Decomposition Algorithms for Highly Scalable Recommender Systems",* Badrul Salwar, George Karypis, Joseph Konstan and John Riedl investigated the use of dimensionality reduction to improve the performance of collaborative filtering-based recommendation systems which have historically been difficult to scale due to the sparsity of the data in their databases. In the paper, they mention testing multiple dimensionality reduction techniques before trying Latent Semantic Indexing (LSI) which implements Singular Value Decomposition (SVD) as its dimensionality reduction algorithm.

They found that SVD which is a matrix factorization technique used for producing low-rank approximations of large sparse matrices outperformed all other dimensionality reduction algorithms by producing the best low-rank linear approximation of the original data matrix. The paper further went onto describe in detail the process and mathematics behind SVD which involves the representation of each user and item by their corresponding eigenvectors and how to process of dimensionality reduction can be used the map users who have rated similar products but not necessarily the same product into the same space spanned by the same or similar eigenvectors. [5]

2.2.3. Building and Testing Recommender Systems with Surprise

In her 2018 article *"Building and Testing Recommender Systems with Surprise",* Susan Li outlined the two most popular approaches to building recommender systems, content-based filtering and collaborative filtering. The article details the development and testing of a collaborative filtering recommendation model that utilizes the Python library Surprise which was built by Nicholas Hug. In this example, she is working with a small dataset that details users rating of books. A perplexing result produced by the article was the algorithm that produced the lowest RMSE in her use case was a baseline algorithm that had no specific tailoring. The article provides a widely applicable demonstration of the effectiveness of this powerful library. [6]

2.3. Recommendation System Approaches

2.3.1. Collaborative Filtering

Collaborative filtering is based on the idea of word-of-mouth-promotion which highlights that the opinion of friends and family plays a major role in personal decision making. In a business context family members and friends are replaced by so-called nearest neighbours, who are users that have a similar preference pattern as the current user. [4] Collaborative filtering is a general technique for exploiting the preference patterns of a group of users to predict the utility of items for a particular user. Three different components need to be modelled in a collaborative filtering problem: users, items, and ratings. Most collaborative filtering methods fall into two categories: Memory-based algorithms and Model-based algorithms.

- Memory-Based Algorithms – Store the training data that has already been rated in a database and from this data the rating of a user for a specific item is predicted based on corresponding users from the training data that have similar tastes.

- Model-Based Algorithms – Statistical models are learned from the ratings of the users in the training data and predictions are made for test users against the trained model.

There are many approaches that can be taken when implementing a collaborative filtering system two of which are user-based and item-based. User-based models find users who have similar tastes based on the items they have rated, or in the case of this project their Spotify playlist behaviour chiefly, the artists and the number of songs they have stored for them. Item-based, items or artists, in this case, are recommended to the user based on the similarity of the items to other items.

LU	name	U_1	U_2	U_3	U_4	U_a
l_1	Data Structures in Java	5.0			4.0	
l_2	Object Relational Mapping	4.0				
l_3	Software Architectures		3.0	4.0	3.0	
l_4	Project Management		5.0	5.0		4.0
l_5	Agile Processes			3.0		
l_6	Object Oriented Analysis		4.5	4.0		4.0
l_7	Object Oriented Design	4.0				
l_8	UML and the UP		2.0			
l_9	Class Diagrams				3.0	
l_{10}	OO Complexity Metrics				5.0	3.0
	average rating ($\overline{r_\alpha}$)	4.33	3.625	4.0	3.75	3.67

Figure 2.1: Example Collaborative Filtering Dataset

In general, all collaborative filtering approaches assume that users with similar tastes would rate an item similarly. This means that in all approaches some form of clustering is being exploited, either explicitly or implicitly. Compared with memory-based approaches, model-based approaches provide a more principled way of performing clustering and is also often much more efficient in terms of the computation cost at the prediction time. As one of the concerns for this project was the computational power required to make predictions in real time when such a large dataset is involved, it was thought to be a judicious decision to choose a model-based approach to building the collaborative filtering recommendation system.

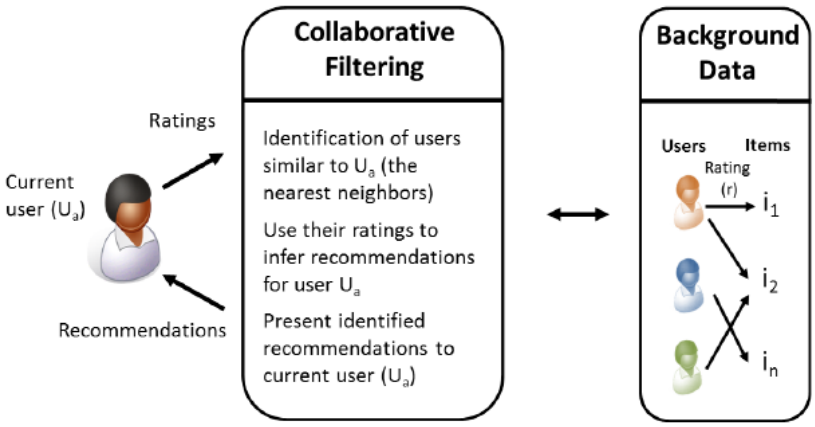


Figure 2.2: Collaborative Filtering Use Case

This would reduce the cost of deploying the model and reduce the time required to produce recommendations. There is one significant drawback to the collaborative filtering recommendation system and that is the cold-start problem.[7] The cold-start problem

describes the difficulty in making recommendations when the users or items are new or not very popular. The traditional approach to tackling this problem is to establish the user item profile via some interview process before generating any recommendations. A similar approach that has been taken with this project except the user profile is generated automatically by gathering the users Spotify playlist data before making any predictions thus mitigating against the cold start problem and eliminating the much-loathed interview process.

2.3.2. Content-Based

Content-based recommendation systems are based on the assumption of monotonic personal interests for example persons interested in Stock Exchange are typically not changing their interest profile from one day to another and will still be interested in this topic in the near future. [4] In a business context a user that reads an article or purchases a newspaper with information on the stock exchange will likely be interested in other articles or documentation about the stock exchange. A content-based recommendation system recommends items to users based upon a description of the item and a profile of the user's interests. The profile of user interests in a content-based recommender system is built from data provided by the user either explicitly, from analysing a set of documents or descriptions of objects previously rated by a user or implicitly by clicking on a link. [4]

category	U_1	U_2	U_3	U_4	U_a
Java	3 (yes)			1 (yes)	
UML	3 (yes)	4 (yes)	3 (yes)	3 (yes)	2 (yes)
Management		3 (yes)	3 (yes)		2 (yes)
Quality				1 (yes)	1 (yes)

Figure 2.3: Example Content Filtering Based Dataset

The profile is a structured representation of user interests. This profile is adapted to recommend new and interesting items. Items that can be recommended to the user are often stored in a database table. A variety of learning algorithms have been adapted to learning user profiles, and the choice of learning algorithm depends upon the representation of the content. There are two key concepts at work in all effective content-based recommendation systems

- Term Frequency – This is simply the frequency that a term or word appears in a document

- Inverse Document Frequency – This is a process that diminished the weight of a term or word that appears very frequently in a document and increases the weight of a term or word that rarely occurs.

The effective implementation of these two concepts helps to ensure the most important words are extracted from any pieces of data gathered about the user's interests rather than the most

frequent. The initial approach for this project was to implement a content-based recommendation system. The rationale behind this initial decision was mainly due to an overall lack of knowledge on recommendation systems after further research it was discovered that content-based recommendation systems typically function best when dealing with text documents. However, creating a hybrid recommendation system by combining a content-based and collaborative filtering approach has been shown to produce very good results and is the method that Spotify utilizes. The content-based recommender helps mitigate against the cold-start that collaborative filtering models can be affected by.

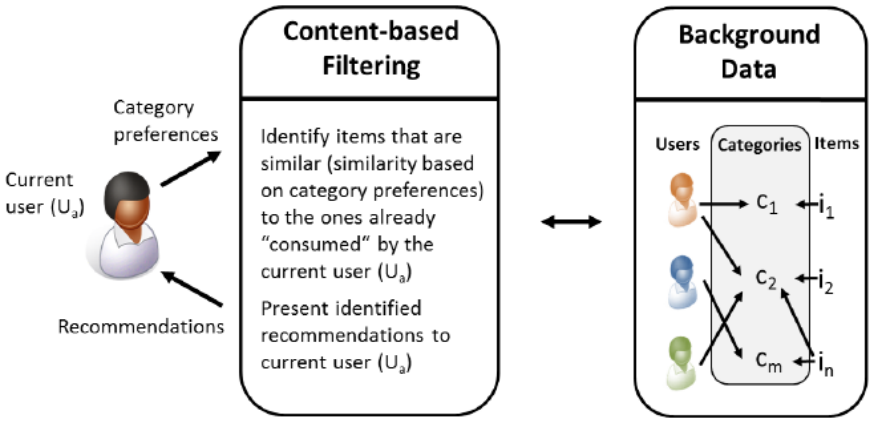


Figure 2.4: Content-Based Filtering Use Case

2.3.3. Knowledge-Based

Knowledge-based recommendation systems like the other approaches use knowledge about users and products to produce recommendations that meet specific user's requirements. However, they differ from content-based and collaborative filtering systems in that they do not recommend items based on a rating system or the textual description of an item. Rather they utilize deep semantic knowledge of the items. Such knowledge describes each item in far greater detail than the other approaches and thus allows for a different recommendation approach. [4]

LU	name	obligatory	duration	semester	complexity	topics	eval
l_1	Data Structures in Java	yes	2	2	3	Java,UML	4.5
l_2	Object Relational Mapping	yes	3	3	4	Java,UML	4.0
l_3	Software Architectures	no	3	4	3	UML	3.3
l_4	Project Management	yes	2	4	2	UML,Management	5.0
l_5	Agile Processes	no	1	3	2	Management	3.0
l_6	Object Oriented Analysis	yes	2	2	3	UML,Management	4.7
l_7	Object Oriented Design	yes	2	2	3	Java,UML	4.0
l_8	UML and the UP	no	3	3	2	UML,Management	2.0
l_9	Class Diagrams	yes	4	3	3	UML	3.0
l_{10}	OO Complexity Metrics	no	3	4	2	Quality	5.0

Figure 2.5: Example Knowledge-Based Dataset

Knowledge-based systems use this descriptive data combined with a set of constraints or rules and or similarity metrics provided by each user to produce its recommendations. The current user receives recommendations by specifying interest in particular item properties. An example of this would be topics = Java, this shows the user has a specific interest in Java-related topics and so any Java-related topics are recommended to the user. As the user's interest profile builds into a set of interests the rules outlined by the user determine what new unsolicited items will be recommended.

This recommendation approach is particularity useful for recommending items such as restaurants where the establishment, its location, the menu and its opening hours could be viewed as one large document of information. There is a variety of semantic knowledge required to accurately describe a restaurant and so using such data and a knowledge-based recommendation system a user could provide constraints such as distance and price and provide an interest in a particular food for example pasta and receive recommendations for all restaurant that satisfy the constraints and the user's interest in pasta.

This approach to solving the music recommendation problem for this project was investigated and it was thought this could be useful as Spotify's API provides vast amounts of metadata for each song. Every song has a variety of semantic knowledge such as duration, genre, artist and much more, this approach would have worked better if the problem was recommending new songs to the user but as a rating system was being used to determine a user's interest in an artist and not the description of their songs this approach was deemed unfit for purpose and so was abandoned.

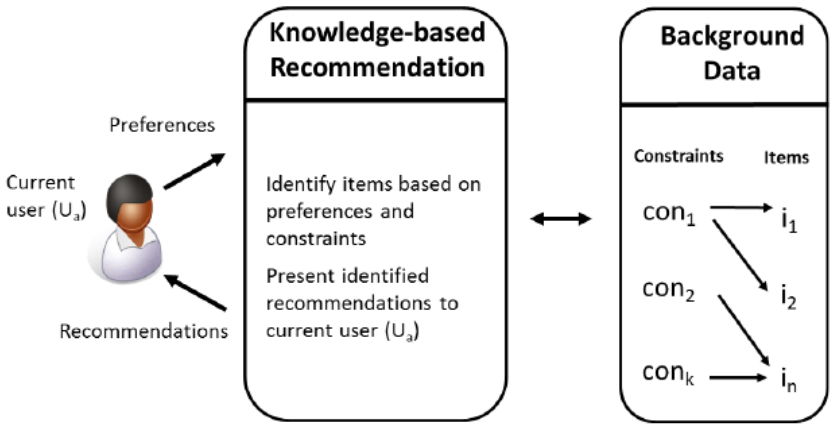


Figure 2.6: Knowledge-Based Use Case

2.4. Machine Learning Algorithms

2.4.1. K-Nearest Neighbour

The K Nearest Neighbour (KNN) algorithm is the standard approach to similarity-based learning. Similarity-based learning uses a distance metric such as Euclidian or Manhattan

Distance to determine the distance between two points in a feature space. The distance between two points determines how similar they are, the closer they are, the greater the similarity. KNN is a non-parametric lazy learning algorithm, the default distance metric used in the algorithm is Euclidian Distance.[8] This is one of the simplest classification and regression algorithms that still provides good results and is commonly utilized in various predictive analytics systems. KNN uses a database of data points that are separated into classes, these classes are used to predict the classification of a new data points based on its distance from the K nearest points within the feature space. K is chosen based on accuracy testing whether that be generic accuracy, f-score, harmonic mean etc.

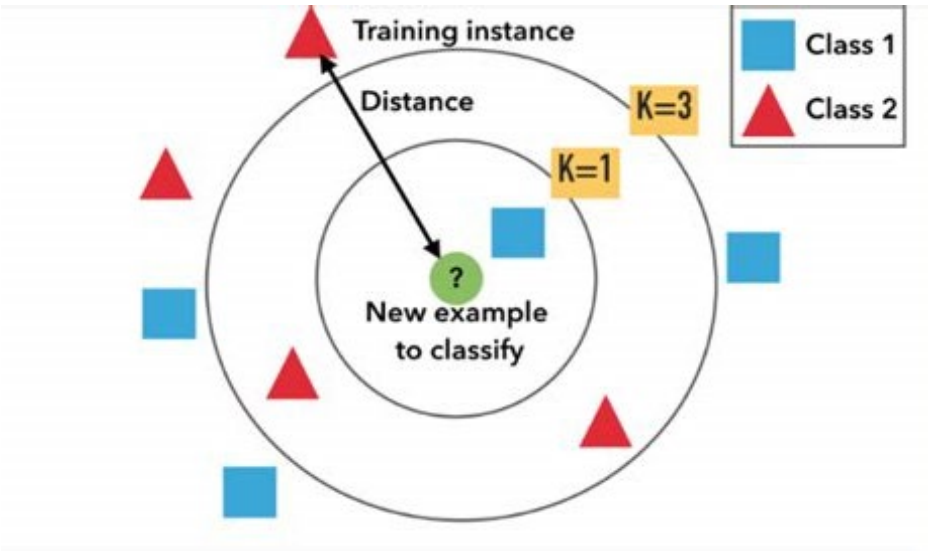


Figure 2.7: K Nearest Neighbours

2.4.2. Bayesian Clustering

Bayesian Clustering is a type of hierarchical clustering algorithm, these algorithms are one of the most commonly used approaches to an unsupervised machine learning model. These algorithms output binary trees whose leaves are data points and whose internal nodes represent the nested clusters of items. Bayesian Clustering assumes that similar users will rate the same type of item in a similar way. This allows users to be grouped together into clusters according to the items they rate. Given a user class 'z', the preferences for different items expressed as ratings are independent, and the joint probability of user class 'z' and the ratings of items can be written as the standard naïve Bayes formulation. [7]

$$P(z, r_1, r_2, \ldots, r_M) = P(z) \prod_{i=1}^{M} P(r_i \mid z)$$

Figure 2.8: Naive Bayes Formula

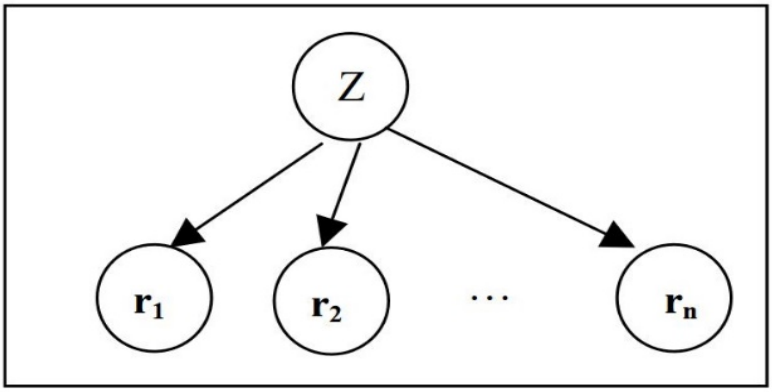


Figure 2.9: Bayesian Clustering

2.4.3. Matrix Factorization

Matrix factorization also known as Matrix Decomposition is a discipline of linear algebra that is used to factorize a matrix into a product of matrices.[9] In layman's terms, it factorizes a large complex matrix into submatrices which can be resolved to the original matrix using the dot product of two or more of the sub-matrices. There are two common methods when implementing matrix factorization *LU* and QR factorization. *LU* is used to factorize a square matrix into *L* a lower triangle matrix and *U* and upper triangle matrix. *QR* is used to factorize an *n x p* matrix into *Q* a square matrix and *R* an upper triangle matrix. Matrix factorization is extremely useful in recommendation systems when dealing with a very large sparse matrix which is the type of datasets utilized by collaborative filtering systems. The dimensionality reduction provided by matrix factorization helps to reduce computation time and reveal latent information about the relationships between the users and items in the matrix.

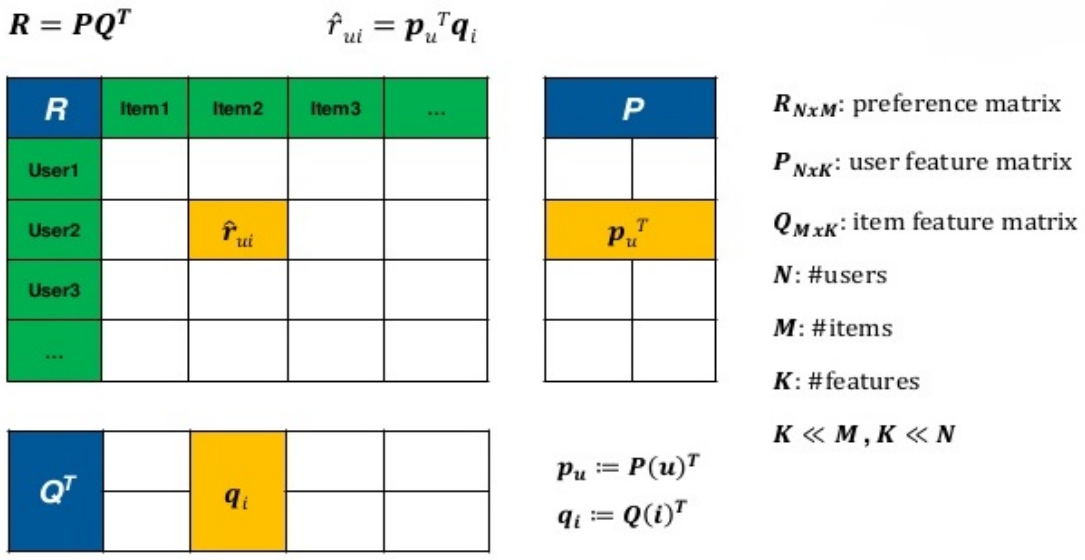


Figure 2.10: Matrix Factorization

2.4.4. Singular Value Decomposition (SVD) and SVD++

Singular value decomposition is a form of matrix factorization, it takes a rectangular matrix of gene expression data (defined as A, where A is an *n* x *p* matrix) in which the *n* rows represent the genes, and the *p* columns represent the experimental conditions. The SVD theorem states:

$$\mathbf{A}_{nxp} = \mathbf{U}_{nxn}\,\mathbf{S}_{nxp}\,\mathbf{V}^{\mathbf{T}}_{pxp} \qquad \text{Where} \qquad \mathbf{U}^{\mathbf{T}}\mathbf{U} = \mathbf{I}_{nxn}, \quad \mathbf{V}^{\mathbf{T}}\mathbf{V} = \mathbf{I}_{pxp} \text{ (i.e. U and V are orthogonal)}$$

Where the columns of U are the left singular vectors (*gene coefficient vectors*); S (the same dimensions as *A*) has singular values and is diagonal (*mode amplitudes*), and V^T has rows that are the right singular vectors (*expression level vectors*). The SVD represents an expansion of the original data in a coordinate system where the covariance matrix is diagonal.[10] In layman's terms, SVD is a form of matrix factorization that utilizes dimensionality reduction techniques to represent an *m x n* matrix as three separate matrices. One matrix represents the vectors of the users another represents the vectors of the items and a third is used to represent their relationship to each other. Performing the dot product on these matrices they can be used to resolve the original matrix

2.5. Languages and Frameworks

2.5.1. Python

Released in 1991 by Guido van Rossum, Python is one of the most popular and fastest growing programming languages today. It is an open-source, high-level, object-oriented, interpreted programming language that enforces indentation to ensure code readability. It is ideal for rapid development; it takes a high-level approach to data structures and it has an abundance of useful open-source libraries.[11] Thanks to the wide variety and abundance of libraries Python is one of the top languages of choice for data science and machine learning. It has a great built-in debugger which is also written in Python. Python also has several fantastic frameworks that provide developers well designed and tested platforms to rapidly develop applications. Because of all these benefits, Python was chosen as the programming language for this project

2.5.2. Java

Released in 1996 by Sun Microsystems which was acquired by Oracle in 2010, Java is a very popular and widely applicable computer programming language, it is concurrent, object-oriented, class-based and designed to have a minimal amount of implementation dependencies. Syntactically Java is very similar to C and C++, but it lacks some of the low-level facilities provided by these languages. Java was designed to allow developers to develop once and run anywhere because of this compiled Java code can run on all platforms that support

Java without requiring recompilation. Java is generally compiled to byte code and run on a JVM (Java Virtual Machine).[12] Java also has several great frameworks for rapid application development and was the initial choice of programming language for this project. This choice was mainly based on familiarity. However, a decision was made to invest the time in learning a new language (Python) and explore some of the benefits that it could offer in the domains of machine learning and Web application development.

2.5.3. R

Released in 1993 by Robert Gentleman and Ross Ihaka, R was written primarily in C, Fortran and R, it is an implementation of the S programming language with the addition of lexical scoping semantics. It is an open source programming language that is mainly used for statistical computing and graphics.[13] It is a very popular language among data scientists, statistician and data miners. R was initially determined to be the language of choice for the predictive analytics portion of this project but due to the accessibility of Python and the realization that time could be better spent researching solutions to the problem rather than learning another new programming language the idea was dropped.

2.5.4. Spring Boot

Released in 2013, Spring Boot is a free open source project that has been developed on top of the Java Spring framework. It enables developers to create a stand-alone MVC (Model View Controller) application with minimal configurations. It provides the developer with default code annotations and configurations to allow for rapid development of applications.[14] It provides the choice of either Maven or Gradle to manage the dependencies and libraries utilized in the project. Spring Boot generates much of the boilerplate code involved with developing Spring applications and it has an embedded HTTP Tomcat or Jetty server to allow testing of the application during development, it also has a wide variety of available plugins. Spring Boot was the initial choice of framework for this project but as it is a Java-based framework it was no longer in contention when the decision was made to use Python as the main programming language

2.5.5. Django

Released in 2013, Django is a Python-based not for profit open-source framework. Django's primary goal is to ease the creation of complex, database-driven websites. The framework emphasizes reusability and "pluggability" of components, less code, low coupling, rapid development, and the principle of don’t repeat yourself.[15] Django implements its own version of MVC (Model View Controller) called MTV (Model Template View) where the view is the controller and the template is the view, it also eliminates the development of the boilerplate code. It has a built-in web server for testing during development and an object

relational mapping system for dealing with database design that provides the developer with out of the box compatibility for some of the most popular database management systems such MySQL, PostgreSQL and MongoDB. It is very easy to configure and has a very large community. Thanks to its large community it has a wealth of fantastic libraries that can aid in the development of services such as Representational State Transfer (REST), geolocation and social authentication to name but a few that were implemented in this project. Django is the framework of choice for this project it satisfied all the requirements outlined in the initial plan, to use a framework that enabled rapid development with an MVC architecture that could be deployed and scaled with relative ease.

2.5.6. HTML, CSS and JavaScript

Hyper Text Markup Language (HTML) defines the meaning and structure of all content on the Web. Hypertext refers to links that connect Web pages to one another, every HTML page is accessible through some link. It uses markup to annotate text, images and other content for display in a Web browser.[16]

Cascading Style Sheets (CSS) is a stylesheet language that is used to describe the presentation of a document written in any markup language (HTML, XML). It uses style declarations to detail how elements should appear when rendered on the screen, in speech, on paper or in other media. [17] Style declarations are made by creating key-value pairs of styling properties.

JavaScript is a lightweight, interpreted, object-oriented scripting language for Web pages. [18] Although, it is found in many non-browser environments. It runs on the client side of the Web and it can be used to design and control how a Web page operates. It generally orchestrates the functionality of a Web page and is essential to building reactive feature-rich Web pages.

2.5.7. Angular

Angular is a JavaScript framework that uses TypeScript as its language for front-end web applications, it combines declarative templates, dependency injection end to end tooling and integrated best practises to produce high-quality applications. Angular is open-source and is developed and maintained by Google and several other organizations and individuals. It is used to build dashboard like applications that are geared towards optimizing user experience.[19] Angular is feature rich with support for almost all JavaScript based libraries, it follows a single page application type design. Angular is utilized in the mobile development framework Ionic enabling it to produce native feeling applications for mobiles while using a more familiar Web application development environment.

2.5.8. Ionic

Ionic is the world's most popular cross-platform mobile development framework. It is an open-source SDK (Software Development Kit) for developing hybrid mobile applications that are built on top of Angular and Apache Cordova.[20] It has a plethora of native feeling elements for IOS and Android such as buttons, icons, themes and much more. It has a built-in web server that allows developers to utilize the browser as a test bench for applications. The major benefit to Ionic is its cross-platform compatibility thus removing the need for platform-specific frameworks such as Android Studio (Android) or Swift (Apple). This framework was used for the development of the mobile application for this project, having Angular as its front-end framework made accessing and rendering the content served from the back-end via API relatively straight forward.

2.5.9. PostgreSQL

PostgreSQL is an open-source object-relational database system that implements the SQL language with many other features that provide safe storage and scalability. PostgreSQL enables the development of scalable database architectures and fault tolerant environments, it is highly extensible allowing developers to define their own data types making it very flexible. [21] It provides many of the good features of SQL such as data integrity without the harsh rigidity associated with it. PostgreSQL is the database management system of choice for this project, it provides fantastic compatibility with Django with the ability to create and store complex data structures such as JSON (JavaScript Object Notation) and Arrays and provides a well-designed graphical user interface called PGAdmin4. Another advantage to PostgreSQL is that it requires significantly less overhead to run than an SQL database

2.6. Libraries

2.6.1. Pandas

Pandas is an open-source, BSD-licensed library for the Python programming language. It provides high performance and easy to use data structures as well as useful data analytics tools. The flexible data structures that Pandas offers, allow developers to quickly manipulate large datasets into forms they can use to perform data analysis against.[22] Some of the most useful data structures Pandas provides are the DataFrame and Series which represent data in a matrix format. Pandas was used extensively throughout this project to process and analyse the large volumes of data, its aggregation, sorting and cleaning capabilities were essential in cleaning and preparing the data.

2.6.2. NumPy

NumPy is an open-source, BSD-licensed library for Python. It is the fundamental package required for scientific computing in Python.[23] Among many others, it provides developers with tools such as N-dimensional array objects, linear algebra operations, Fourier analysis, and random number capabilities. The N-dimensional arrays and linear algebra capabilities were particularly useful for the machine learning aspect of this project and are especially useful in the case of matrix factorization problems.

2.6.3. Scikit-Learn

Scikit-Learn is an open-source, BSD-licensed library for Python. It is built on top of NumPy, SciPy and Matplotlib and is aimed at providing tools for data mining and data analysis problems. Scikit-Learn has tools for dealing with Classification, Clustering, Dimensionality Reduction, Regression and more.[24] Scikit-Learn was used to analyse the data in this project and is a requirement for the Surprise library which uses it extensively in its recommendation system models.

2.6.4. Surprise

Surprise is an open-source, BSD-licensed library for Python. Surprise is built on top of the NumPy and Scikit-Learn libraries, it aims to give users perfect control over their experiments by providing various ready-to-use prediction algorithms such as baseline algorithms, neighbourhood methods, matrix factorization-based and the ability to create custom recommendation algorithms. It also has built-in tools for working with datasets and for evaluating, analysing and comparing the performance of the various algorithms.[25] The matrix factorization algorithms mainly Singular Value Decomposition (SVD) and evaluation tools were used throughout this project. The SVD algorithm provided by the library was developed based on the SVD algorithm used by the winning team in the Netflix prize which demonstrates both the effectiveness of this algorithm and the libraries applicability to this project.

2.6.5. Django REST Framework

Django Rest Framework is a powerful and flexible toolkit for building Web APIs. [26] Among many other useful features, it provides multiple authentication policies including packages for Token Authentication, Oauth1a and Oauth2. A serialization system that supports Object Relational Mapping (ORM) and non-ORM data sources and view sets for combining the logic of a set of related views into a single class to increase code modularity. It also provides generic API templates for views to ease the development and testing process. In this project, Django REST Framework was used to build the API endpoints that are used to serve most of the back-end functionality of the Web application. Its serialization classes were used extensively in

performing CRUD (Create Read Update Delete) operations on the database and the Token Authentication system was used to authenticate users accessing the system via a mobile device.

2.7. Cloud Service Providers and Deployment

2.7.1. Amazon Web Services

Amazon Web Services (AWS) is a subsidiary of Amazon that provides on-demand cloud computing services on a pay as you go basis. AWS is the largest cloud service provider in the world, it comprises of more than 90 services spanning a wide range including computing, storage, networking, database, analytics, application services, deployment and more.[27] The services that are of most use for this project are the Elastic Compute Cloud units (EC2) and the Relational Database Service (RDS). EC2 is a virtual machine that can be configured to the developer's required specification including the number of CPUs, operating system, storage capacity and networking capabilities.

Figure 2.11: EC2 Console

RDS is a service provided to host relational databases such as SQL, PostgreSQL and more. For this project, the EC2 service is being used to host the Django Web application and recommender system on separate instances and the PostgreSQL database is hosted using the RDS service.

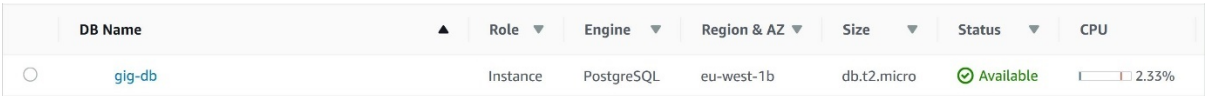


Figure 2.12: RDS Console

2.7.2. Docker

Docker is an open-source tool that is used to make it easier to create, deploy and run applications using containers. Containers are essentially packages that allow developers to wrap up their application with all its required dependencies and libraries and ship it out as one unit. This ensures that the application will work on any Linux machine so long as it has Docker installed on it. Docker works like an operating system for containers, containers work by virtualizing the operating system of a server and then run the application code on top.[28] This allows developers to produce applications without worrying about the system the application will eventually run on it also benefits operational staff by giving greater flexibility and potentially reducing the number of systems needed because of its small footprint and lower

overhead. Docker was used in this project to containerize the Web application and the recommendation system and deploy them to the EC2 instances. This was all accomplished with a Dockerfile and a handful of commands. A Dockerfile is essentially a specification file that tells the Docker kernel what packages and libraries to install in the container, in what directory the application should be placed, what scripts and commands to run and the entry point for the container when it is run.

2.8. Alternative Existing Solutions

Several products and services exist in the field of music recommendation however no products or services could be found that recommended live music events. The closest substitute that could be found were products that sell and advertise tickets to events. There are a plethora of products that provide this service so two will be examined in further detail.

2.8.1. Bandsintown

Bandsintown is a Web and mobile application that provides users with a platform to find and purchase tickets for upcoming live music events, it also provides a facility to book hotels or hostels in the city where the event is occurring. The website provides a brief description of each event including the name of the artists, venue, door time and information on what area of the venue each ticket grants the holder access to. Design inspirations were taken from this website. It has a clean and simplistic approach, the colour scheme used throughout the platform was like what had already been developed when this product was discovered and so provided a good depiction of what the front-end of the finished project might look like. https://www.bandsintown.com/

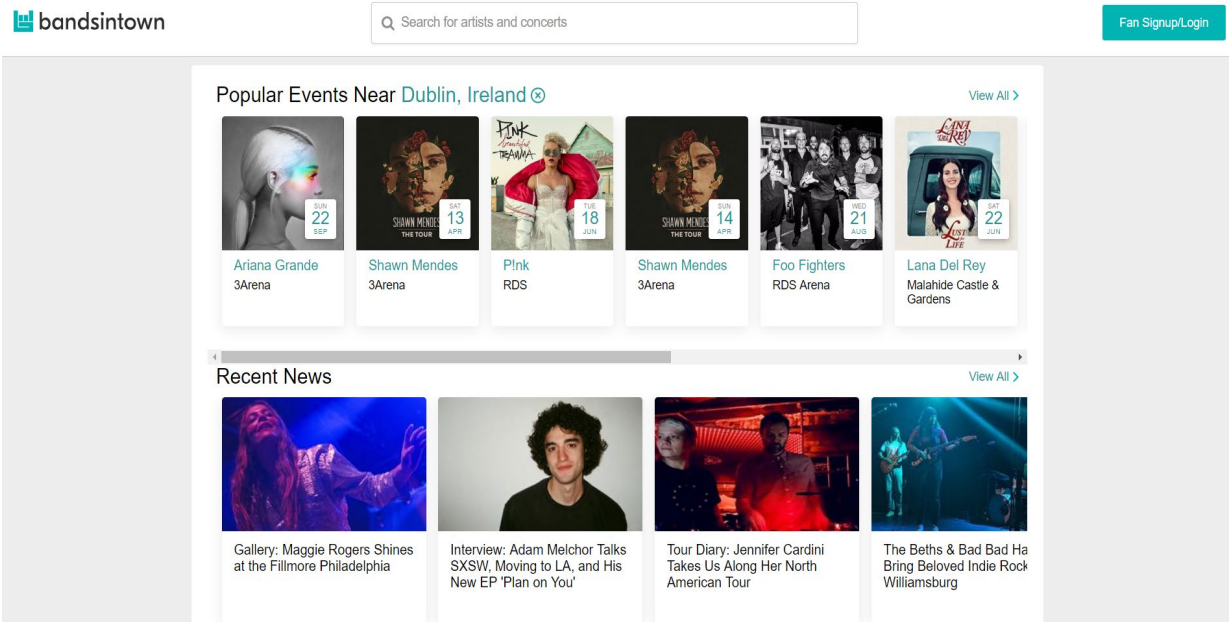


Figure 2.13: Bandsintown Website

2.8.2. Ticketmaster

Ticketmaster is the largest and most popular ticket sales and distribution company in the world. It provides a secure platform for the sale and distribution of tickets to a wide variety of events. It provides information on the type of event, its location, the different show times and dates and in some cases provides a brief description of the event and critic reviews. Ticketmaster also provides access to much of the information about these events via its APIs. Ticketmaster played a pivotal role in this project and without its APIs it would not have been possible, the project also borrowed some design queues from its website. https://www.ticketmaster.ie/

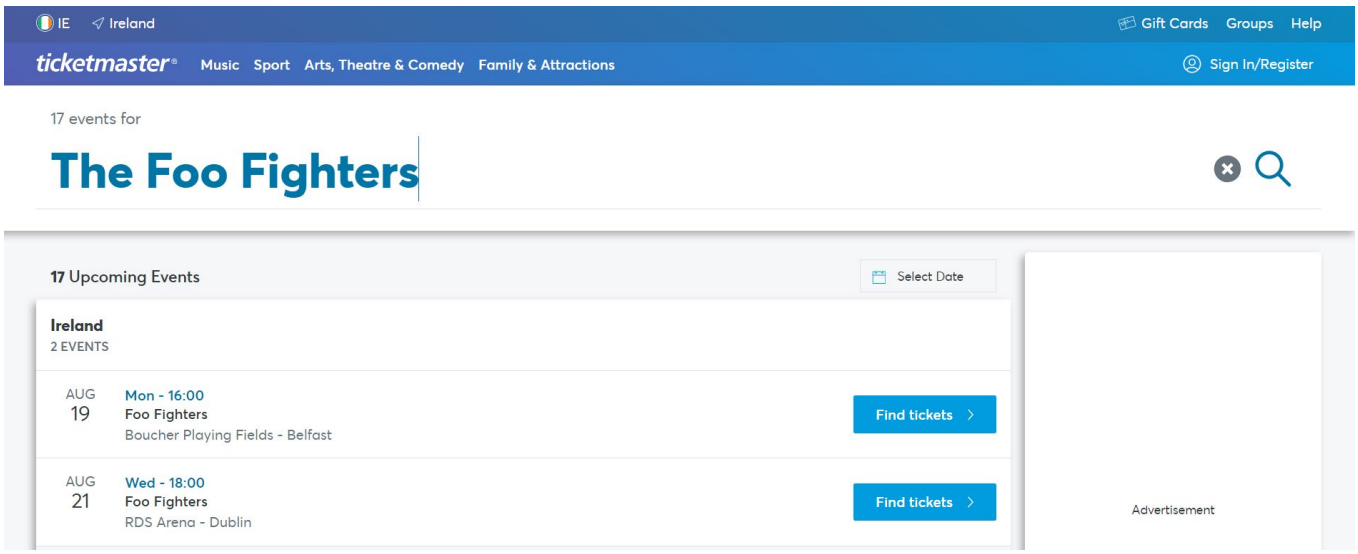


Figure 2.14: Ticketmaster Website

2.9. Conclusions

This chapter looked at some of the key background inspirations, tools and technologies used in this project. It first presented some of the relevant academic research, including basic approaches to recommendation systems, incremental Singular Value Decomposition (SVD) algorithms and building recommendation systems using Surprise. Following that, we looked at some of the approaches to recommendation systems and what fields they are most useful for. Proceeding on, we investigate several machine learning algorithms and techniques that are highly applicable to recommender systems. We then moved on to the various technologies used in the project such as the programming languages, frameworks, libraries and deployment methods. Finally, a review of some existing solutions comparable to this project.

3. Design and Architecture

3.1. Introduction

Following on from the previous chapter, where some of the key background research was presented, these themes will be continued and developed further in this chapter, where the design of the system will be presented. The first section will look at the software methodology employed in this project which describes the software lifecycles that were adhered to throughout design and development. The next section outlines the technical architecture of the system in which we will discuss the design of the front-end, back-end and database. Regarding the front-end some of the stylistic and operational choices will be discussed, for the back-end, we will look at the connectivity, security and performance choices and the structure of the code base and with respects to the database we will examine the structure and relationships within.

3.2. Software Methodology

The development of this project consisted of two distinct and separate processes, the development of the Web and mobile applications and the construction of the recommendation system. As such, it was prudent to take separate approaches to the development lifecycles of these aspects and implement methodologies that suited the different tasks. The following sections will outline some of the methodologies that were researched for this project, highlight the ones that were implemented and provide justification for their selection.

3.2.1. Agile

Agile software development is based on an incremental, iterative approach. It favours incremental design and development over approaches such as Waterfall which focus on in-depth planning at the begging of the project. Cross-functional teams consisting of developers, systems administrators and management work together on iterations of a project over a long time in small periods called sprints which are typically between one and two weeks. The work is organized into a backlog that prioritizes tasks based on the business requirements. Agile methodologies are open to the requirements of a project changing over time and it encourages constant feedback from the end users. [29] This allows teams working in an agile lifecycle to deliver high-quality features faster and be more flexible and adaptable to change.

3.2.2. Agile Scrum

Scrum is a subset of Agile and one of the most popular process frameworks for implementing Agile. It is an iterative software development model used to manage complex software and product development. Fixed-length iterations, called sprints lasting one to two weeks, allow the team to ship software at a regular cadence. At the end of each sprint, stakeholders and team members meet to plan the next steps. Scrum follows a set of roles, responsibilities, and meetings that never change. For example, Scrum calls for four ceremonies that provide structure to each sprint: sprint planning, daily stand-up, sprint demo, and sprint retrospective. [29] During each sprint, the team will use visual artefacts like task boards or burndown charts to show progress and receive incremental feedback. Scrums daily stand up meetings provide more transparency and visibility to projects this helps to eliminate any misunderstandings within the team. Due to the lack of a project manager the team has increased accountability, the team works as a collective to decide on the best solution to the current problem. This decreases expenditure by removing a layer of management and allows the team to be more flexible and accommodate changes with greater ease.

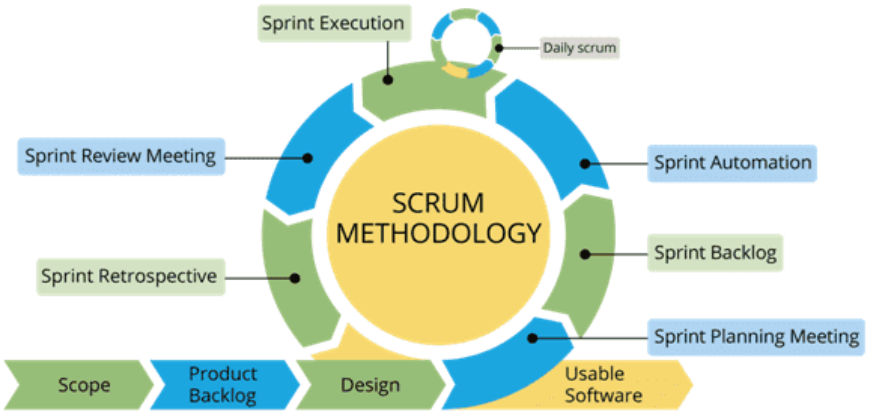


Figure 3.1: Agile Scrum Methodology

3.2.3. Agile Kanban

Kanban is Japanese for “visual sign” or “card.” It is a visual framework used to implement Agile that shows what to produce when to produce it, and how much to produce. It encourages small, incremental changes to the current system and does not require a certain setup or procedure. This means it is possible to overlay Kanban on top of other existing workflows. A Kanban board is a tool to implement the Kanban method for projects. A Kanban board, whether it is physical or online, is made up of different swim lanes or columns. The simplest boards have three columns: to do, in progress, and done. [29] The main benefits of Kanban include its ability to enable the visualization of the workflow which can provide a broader more abstracted view of the whole project. This feature allows for the project's backlog to be prioritized more effectively.

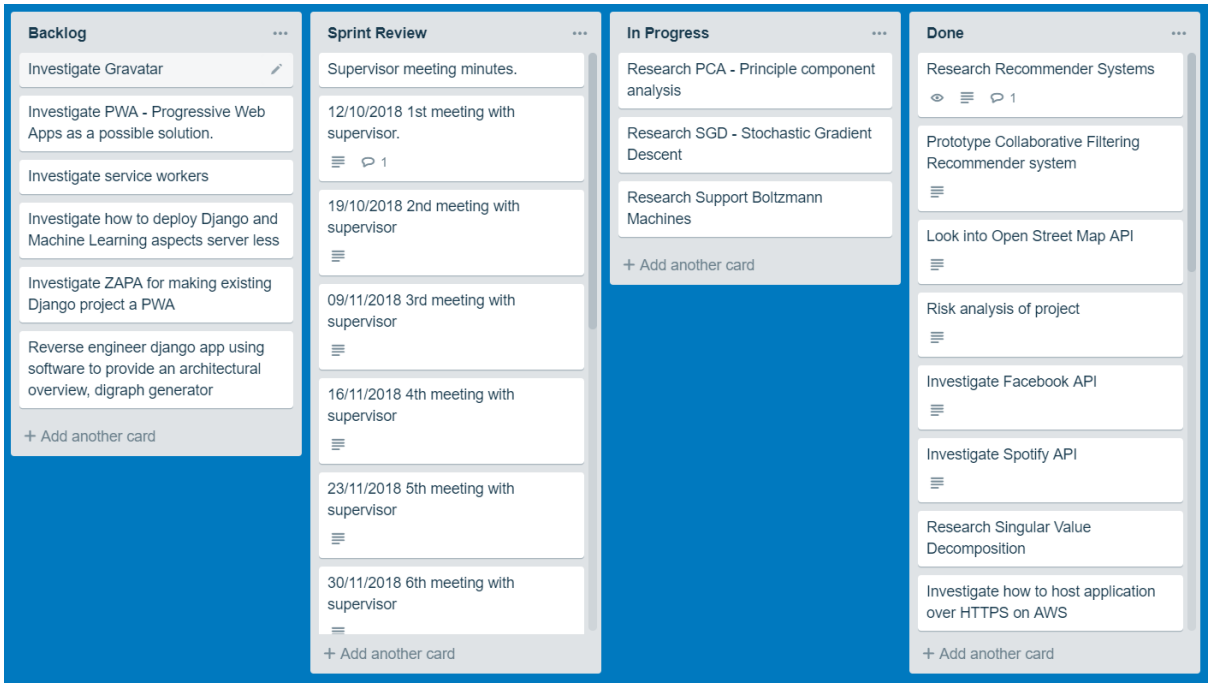


Figure 3.2: Agile Kanban Board (Trello)

3.2.4. ScrumBan

ScrumBan is a name coined for the hybrid software methodology used throughout the Web and mobile application development in this project. It is an implementation of Agile Scrums cyclic sprint development pattern that also incorporates the organizational benefits of an Agile Kanban board. The sprint cycle for this project is one week. This was decided upon so any additional features or problems could be demonstrated to the project supervisor who acted as a sudo scrum master. In order to keep track of the objectives and tasks to be completed a Kanban board was implemented using the popular free platform www.Trello.com. This effectively acted as the backlog for the project and was updated constantly throughout the project's entirety. I was used to detail the weekly meeting minutes with the project supervisor and to keep track of any ongoing work or research, potential features, and any completed work.

3.3. Predictive Data Analytics Software Lifecycle

3.3.1. CRISP-DM

Building predictive data analytics solutions for applications involves a lot more than just choosing the right machine learning algorithm. Like any other significant project, the chances of a predictive analytics project being successful are greatly increased if a standard process is being used to manage the project through the project lifecycle. One of the most commonly used processes for predictive analytics is the Cross-Industry Standard for Data Mining (CRISP-

DM). [8] While there are several other development and design methodologies for data mining and machine learning, one such example being Sample Explore Modify Model and Assess (SEMMA) developed by the SAS Institute, CRISP-DM has been chosen as the methodology that will be adhered to throughout the development of the predictive analytics portion of this project. The reasoning behind this is it outlines very clear and well-defined stages that are highly applicable to this project. The iterative cycle it proposes aligns perfectly with the development cycle for the rest of the project, it allows for constant improvements and changes as the scope of the project evolves. The recommendation model is a key component to this project and thus the adherence to a tried and tested development methodology is a prudent move in ensuring the production of a system that meets the requirements of this project.

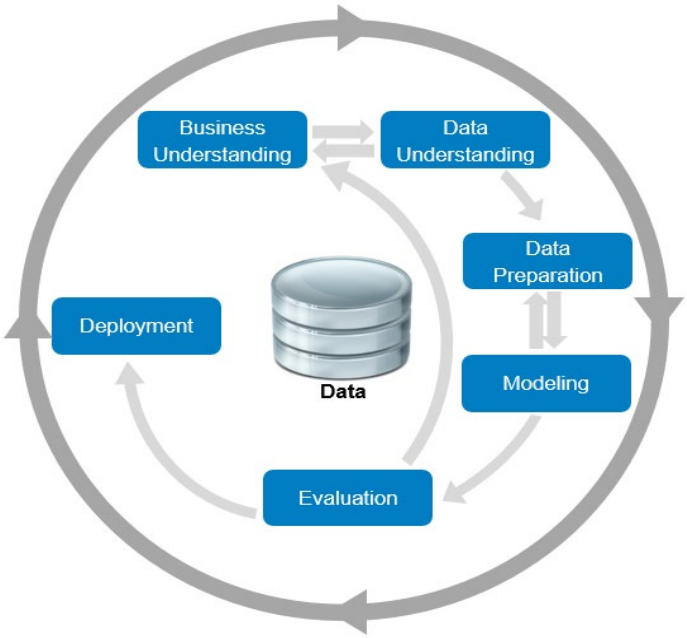


Figure 3.3: CRISP-DM Development Cycle

3.3.2. Business Understanding

The first stage in the CRISP-DM process is business understanding. Predictive analytics projects never start out with the goal of building a prediction model. Instead, their focus is to gain more customers or sell more products all while improving the efficiency of their business. So, the goal in this phase of the development cycle is to fully understand the business needs so the right model can be produced. The goal at this stage of the process is to uncover important factors that could influence the outcome of the project. This can be achieved by outlining the primary objective from a business perspective, producing a plan that details how data mining and predictive analytics can achieve that objective and clearly defining the business success criteria.

3.3.3. Data Understanding

Once the objectives have been accessed from a business perspective, it is important that the data analyst then fully understands the different data sources available within the organization and the different types of data within those sources. If the data is gathered from multiple sources, it is important at this stage to consider how and when these data sources should be integrated. There are a few key aspects to this stage, the first is exploring the data, this involves utilizing data visualization and reporting techniques such as simple statistical analysis or producing graphs based on simple aggregations. The next stage is verifying the data quality, this entails examining the data and answering questions such as, is the data complete, are there any missing values? Is it correct, does it contain any errors and how prevalent are they? The final stage is to produce a data quality report, this report should list the results of the data quality verification and suggest solutions to data quality problems.

3.3.4. Data Preparation

Building predictive data analytic models require specific kinds of data, organized in a specific kind of structure known as an Analytic Base Table (ABT). This phase of CRIPS-DM includes all the activities required to convert the disparate data sources that are available into a well-formed ABT. This involves deciding which data from which sources will be used for analysis, the criteria for which should be based on the relevance of the data to achieving the business objective. The main concepts at this point are, cleaning the data which includes raising the overall quality of the data and possibly implementing techniques such as imputation for handling missing values by substituting a plausible estimate or clamp thresholding for setting the upper and lower bounds of specific fields. The second step is constructing the data, this includes constructive data preparation operations such as the production of derived attributes or entire new records or transformed values for existing attributes. The final step is integrating the data, this involves merging the different data sources and any aggregations that are performed on the data.

3.3.5. Modelling

The modelling phase of the CRISP-DM process includes utilizing several machine learning algorithms to build a range of predictive models from which the best model will be selected. This is accomplished by following several processes the first of which is the generation of the test design. Generating a test design encompasses generating the procedure or mechanism by which the different models will be evaluated, for criteria such as speed, effectiveness and validity. The next step in building the model included the construction of the different models, adjusting the parameters to attain the best results and describing the results of the model. The

final step is assessing the model in which results of the model are noted and any revision of the parameter settings are implemented.

3.3.6. Evaluation

This phase of CRISP-DM covers all the relevant evaluation tasks required to prove the effectiveness of the tested models and determine which of the models is best suited to the business needs and to asses, any other information gathered throughout the process. This may be accomplished by assessing the degree to which these aspects meet the business objectives or possibly by testing the models on test applications. After a model has been selected at this time it is important to perform a more thorough review of the data mining engagement to ensure no details have somehow been overlooked. This process should also include quality assurance measure to determine if the model was correctly built and tuned. The final step in the evaluation process is determining the next step. Depending on the successfulness of the evaluation process it should be determined whether another iteration of the lifecycle is necessary or if the model is fit for deployment.

3.3.7. Deployment

The final section of the CRISP-DM lifecycle covers all the work that is necessary to successfully integrate the predictive data analytics model into the process within the organization. Achieving this includes determining a strategy for deploying the model, where in the business the model should be deployed or whether the system should be deployed on-site or with some other service provider. Also included in this stage is the planning of monitoring and maintenance, having a robust monitoring and maintenance plan is one of the best strategies to mitigate against unnecessarily long periods of incorrect usage of data mining results. Finally, it is necessary to produce a report that gives a detailed overview and review of the process in its entirety that also outlines any areas for future work and potential risk factors. [30]

3.4. Technical Architecture

This section discusses the technical architecture of the system, it first examines the front-end design of the system, and presents some early stylistic prototypes for the Web and mobile applications. Furthermore, it discusses the architecture and deployment of the back-end and recommendation system. The data analysis, cleaning and preparation will also be examined highlighting some of the patterns found and how the data was optimized for an item based collaborative filtering recommendation system.

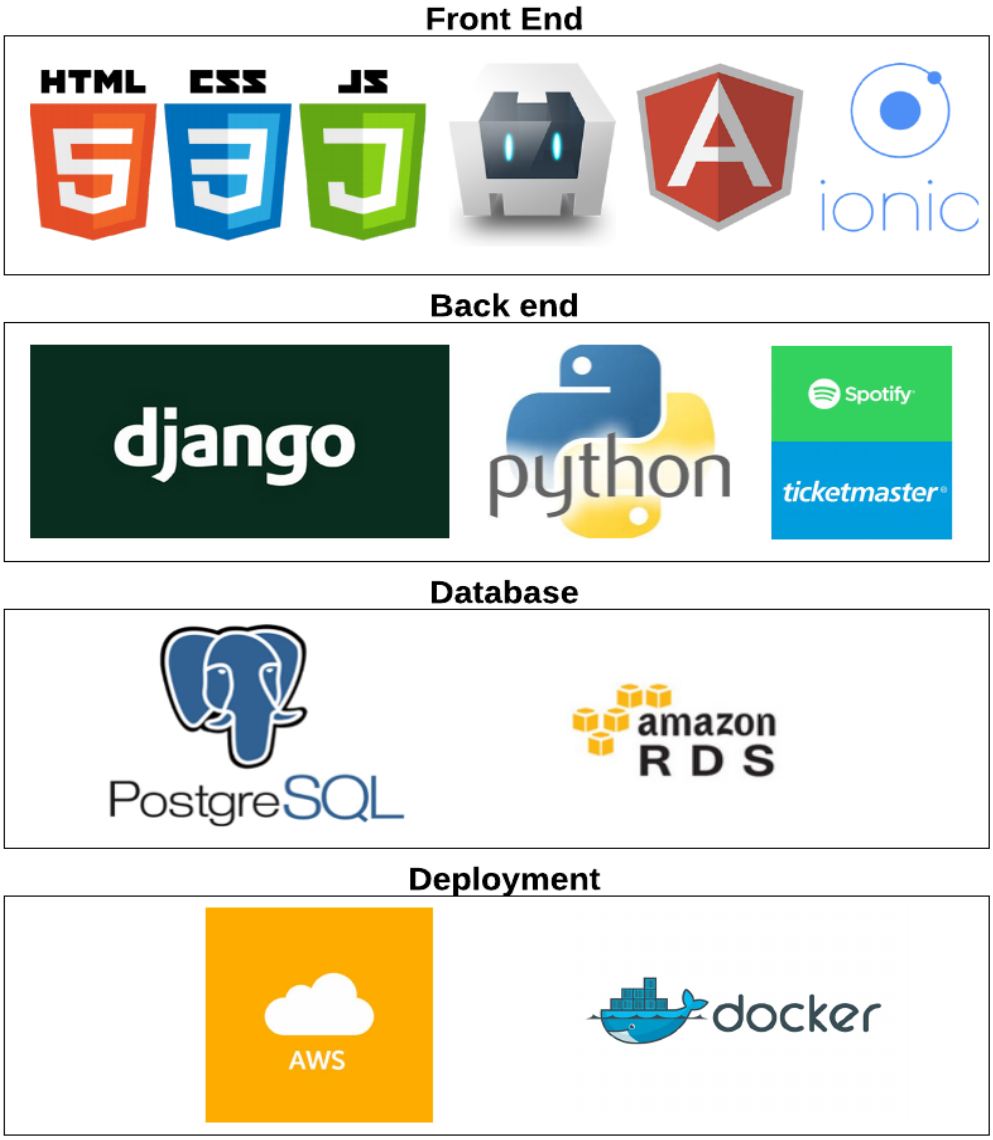


Figure 3.4: Technology Stack

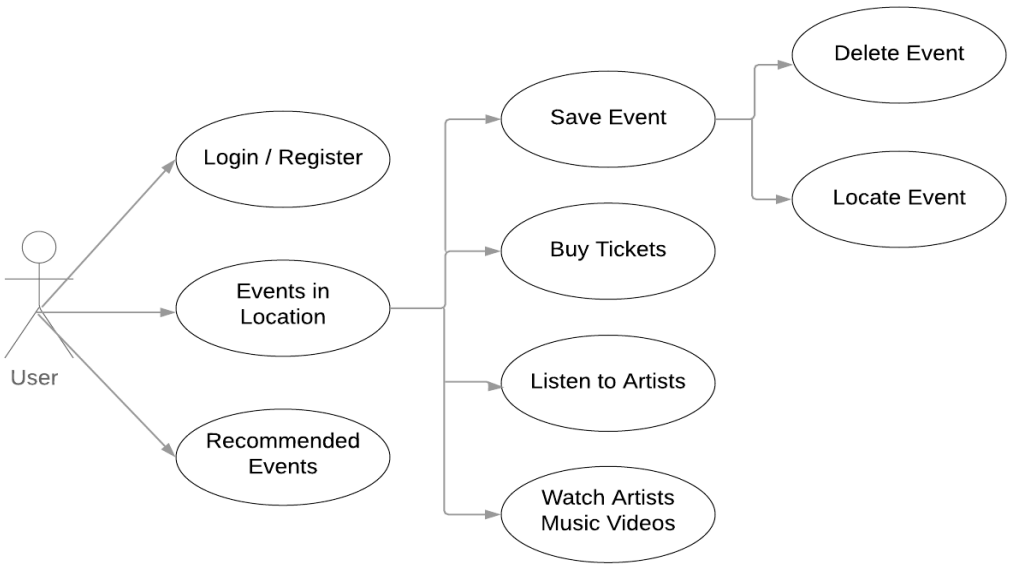


Figure 3.5: Use Case Diagram

3.4.1. Front-End Design

The front-end design of the Web and mobile applications followed the six fundamental principles of design balance, proximity, alignment, repetition, contrast and space. Balance, the components of the webpage and mobile app use an organizational structure based on their size, this provides a clear indication of the importance of each component and guides where the user focuses their attention on the screen.

Proximity, also important in where the user's attention is drawn to, it creates the relationship between the different components and involves placing related components such as the links to external services in close proximity.

Alignment, essential to creating an ordered visual design and ensures a consistent structure, the main content was aligned in the centre of the page to emphasize its importance. Repetition, vital in tying together individual elements it provides a feeling of organized movement to the content that helps it flow, this is evident in the display of the events in figure 4.6, the same repeating structure is used to display each event.

Contrast, the juxtaposition of opposing elements which helps to highlight key components and emphasis their functionality, colour places a vital role in creating contrast that is pleasing to the eye and is used throughout.

Space, the distance between elements, applying the correct amount of positive and negative space provides the display with a clean and uncluttered appearance, the edges and very centre of the display are kept clear to provide the appearance of space.

3.4.1.1. Web Application

The front-end of the web application was designed around the Django framework. Django provides a templating structure that maximizes the Don't Repeat Yourself (DRY) principle when developing Web pages, one example of this is the implementation of a base template, which acts as a parent page that all other templates (HTML Pages) in the project are derived from. In this template, all the CSS, JavaScript files and Content Delivery Network (CDN) links that are applicable to the entire website are imported. Each child page such as the landing page in figure 3.6, is then rendered in a section of the base page when required and can provide additional CSS, JavaScript and CDN links. This allowed the structure of the front-end of the project to be designed in a modularized fashion.

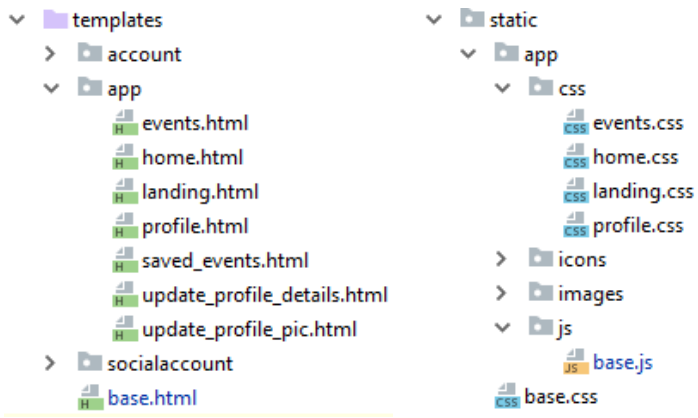


Figure 3.6: Front-End File Structure

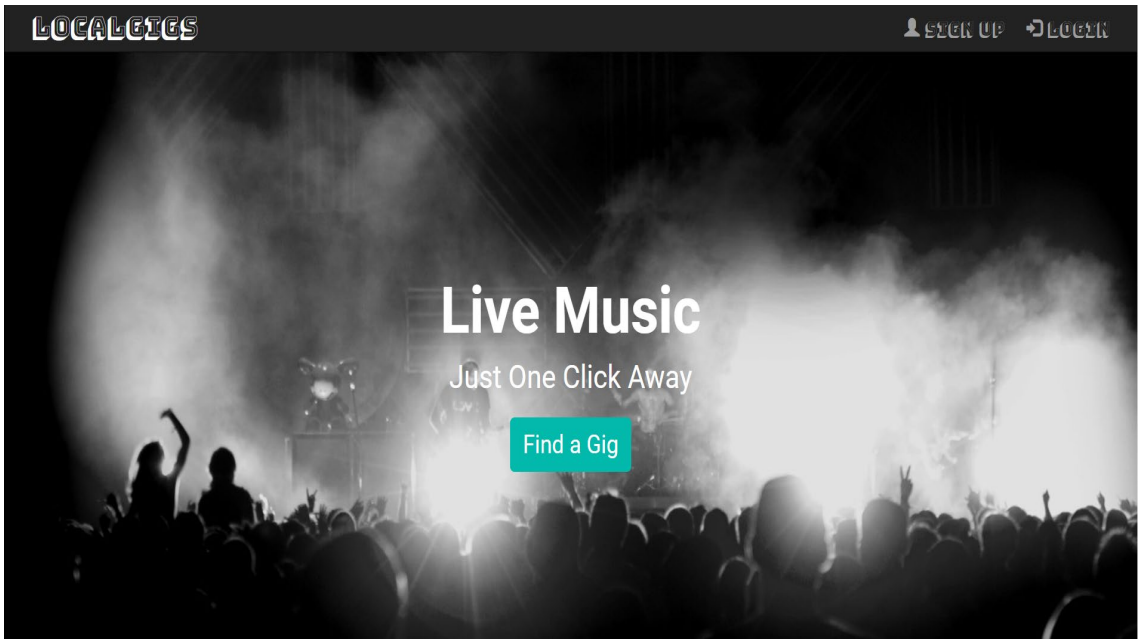


Figure 3.7: Landing Page

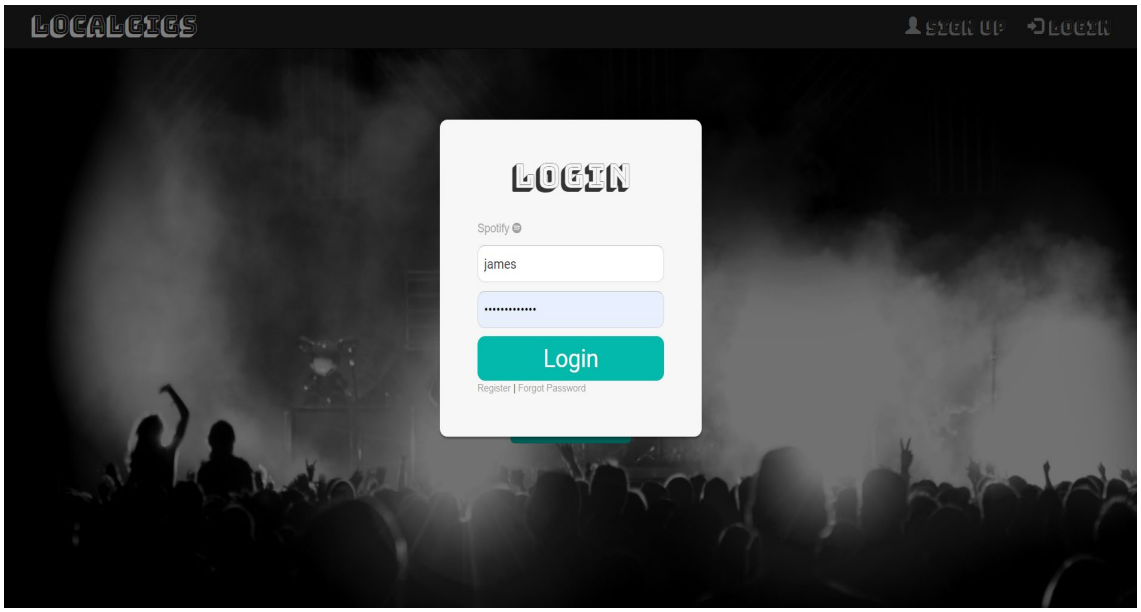


Figure 3.8: Modal Login

The landing page for the web application was designed with simplicity in mind but with the requirement that the user would know exactly what the application is intended for. The initial prototype for the profile page can be seen in figure 3.8, in the beginning, it was designed to have drag and drop functionality where the user would drag events from the search box into the interested box and they would then be saved. This approach gave a cluttered and clunky feel to the design and in later iterations was removed. At this stage the webpage appeared to be cluttered and empty at the same time, this was due to having too much happening in one portion of the screen. The decision was made to remove the section of the page that provided the recommendations and search functionality to the home page which can be seen in the finalized display format in the Development chapter (figure 4.6), a trimmed down version of the users details and the list of saved events are retained on the users profile page.

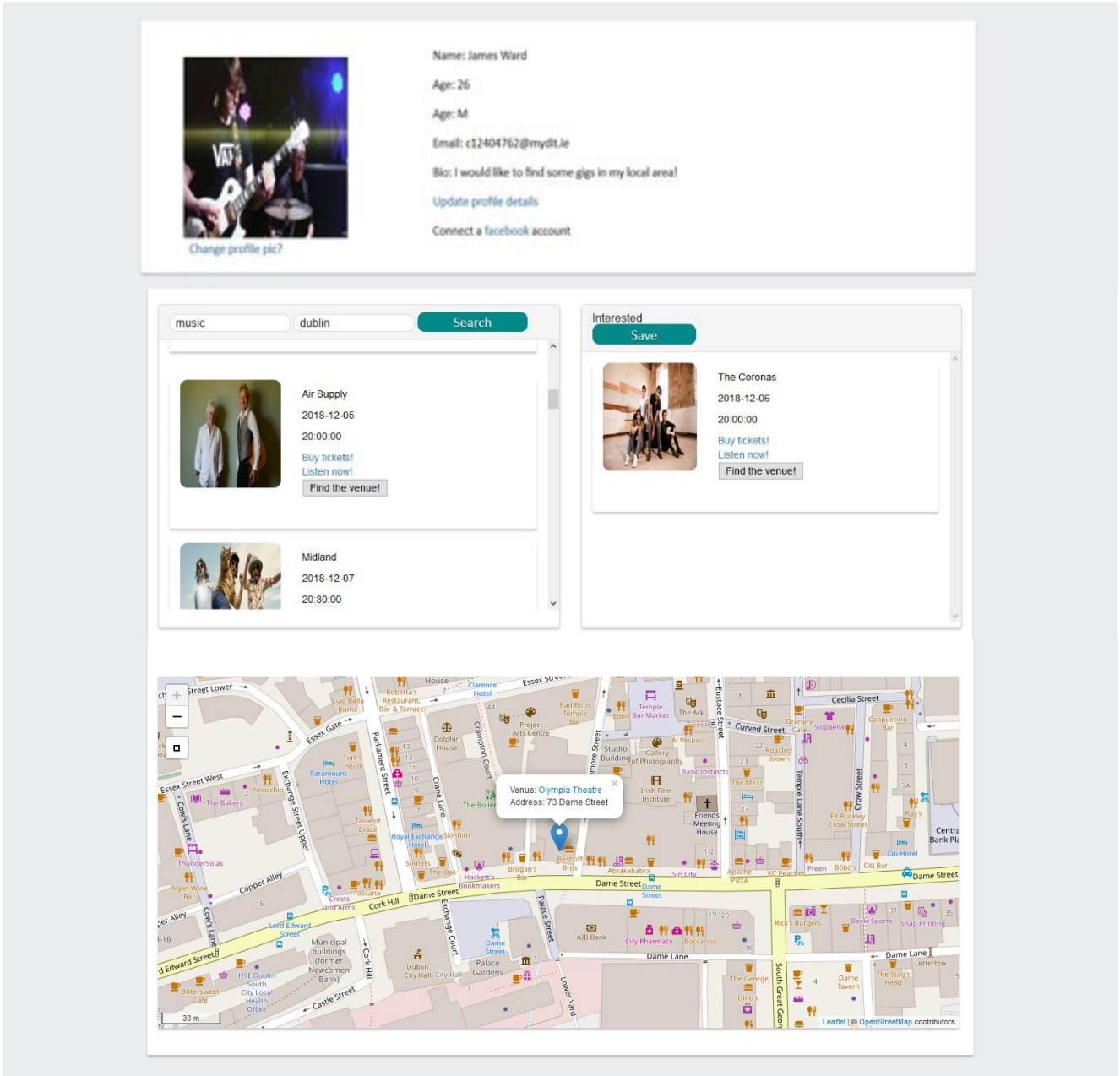


Figure 3.9: Early Profile Page Design

Each event is designed to be rendered dynamically in its own separate div. In the early prototypes shown above this was accomplished completely client side using JavaScript the code for which was ugly and not exceptionally efficient. This was changed in later iterations of the development cycle to make use of Django's page rendering capabilities server side which will be explored further in the Development chapter. In the later iterations each event contains a name and image for the artist, the venue and its location, the date and time of the event and links to purchase tickets on Ticketmaster, listen to the artist on Spotify or watch their music videos on YouTube, there is also a button to save the event to the users profile on the Home page and buttons to delete and locate events on the Profile page.

3.4.1.2. Mobile Application

The mobile application is designed on the same principles as the Web application. The appearance was kept as similar as possible, which is evident in figure 3.10, to provide continuity between the separate applications. As was mentioned in the research section the mobile application has been developed using the Ionic framework which uses Angular for its code structure and development, this kept the code structure ordered and modularized. Each page has its own HTML, CSS and TypeScript file thus compartmentalizing the code, all access to external resources such as the back-end, are handled in the rest.ts service provider class. Angular's rendering process is like Django's such that index.html acts as the base template and all other pages are rendered within, where each page fully encapsulates the display and functionality of one screen in the application.

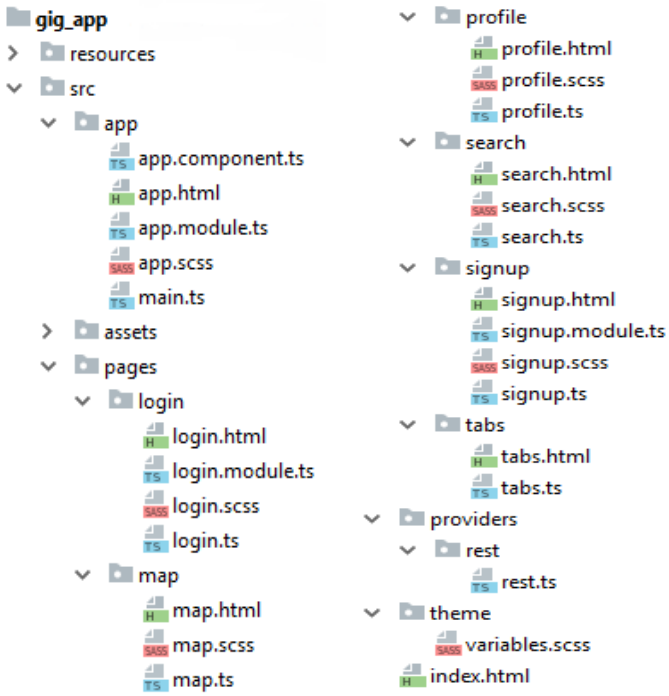


Figure 3.10: Mobile Application File Structure

The mobile application is designed to be a point of reference for the user, a place where they can access the events, they had saved in the Web application and make use of the routing feature to locate venues on the go. Users have the ability to search for new events, receive their recommended events and save events, they are also able to purchase tickets to events and listen to the artist on YouTube and in the early prototypes that can be seen in figure 3.10, they had much of their account details displayed to them that feature was thought to be unnecessary and was removed from the final design.

The mapping feature is designed using the lightweight, open source JavaScript library Leaflet, this platform was chosen over competitors such as Google Maps because it is free and highly customizable, as can be seen in figure 4.11, where custom designs have been implemented for the user and event marker and details of the venue of the event and a message to the user are dynamically rendered in the markers popup.

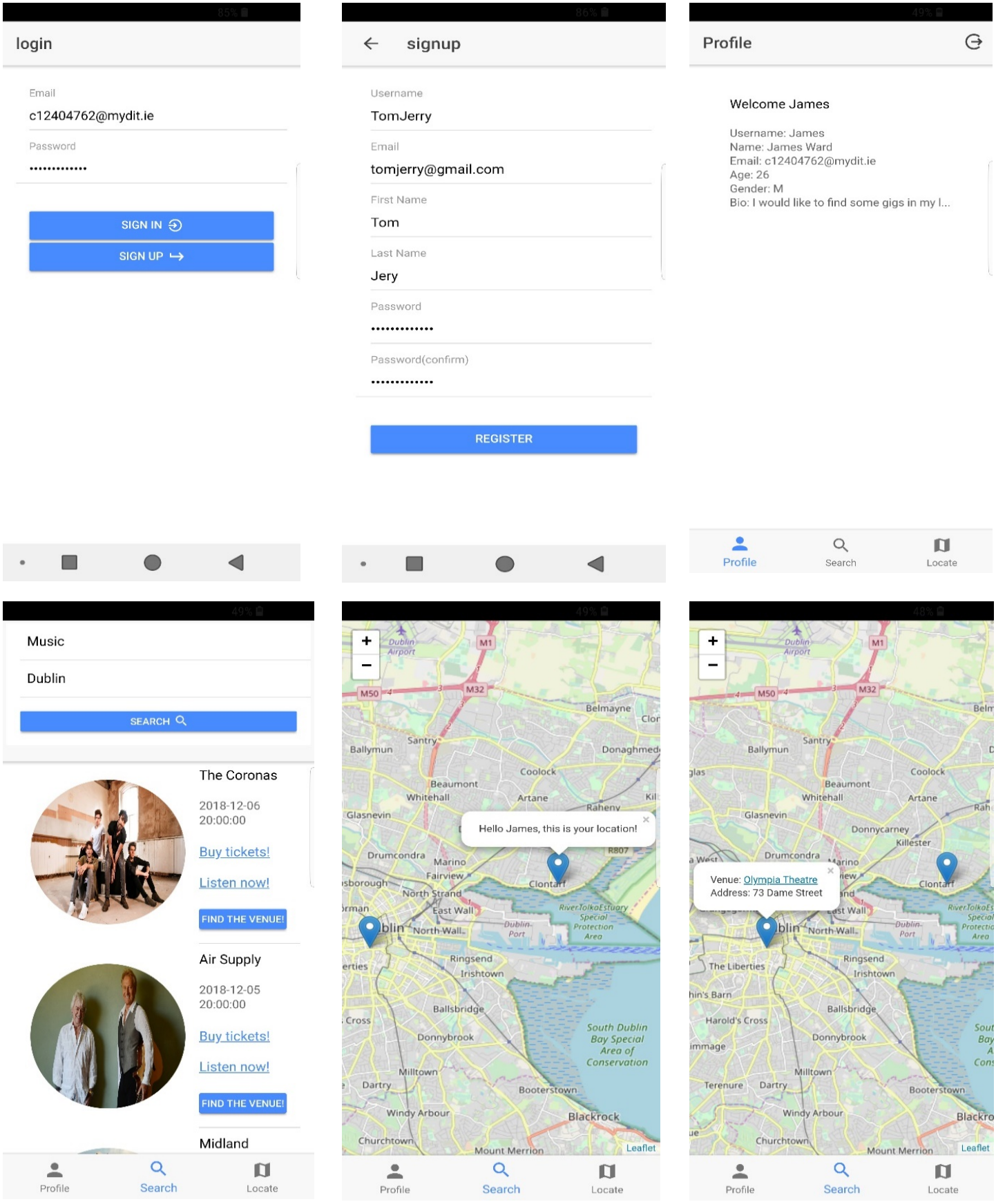


Figure 3.11: Early Screen Captures from Mobile Application

3.4.2. Back-End Design

The design of the backend follows Django's Model Template View (MTV) structure which, as explained the Research chapter is the same concept as the popular Model View Controller (MVC) approach to designing and architecting a code base. The backend is separated into two applications app and api, reference figure 3.11, both applications have very similar file structures, most of these are boilerplate file names generated by Django. However, the functionality in these applications is completely different.

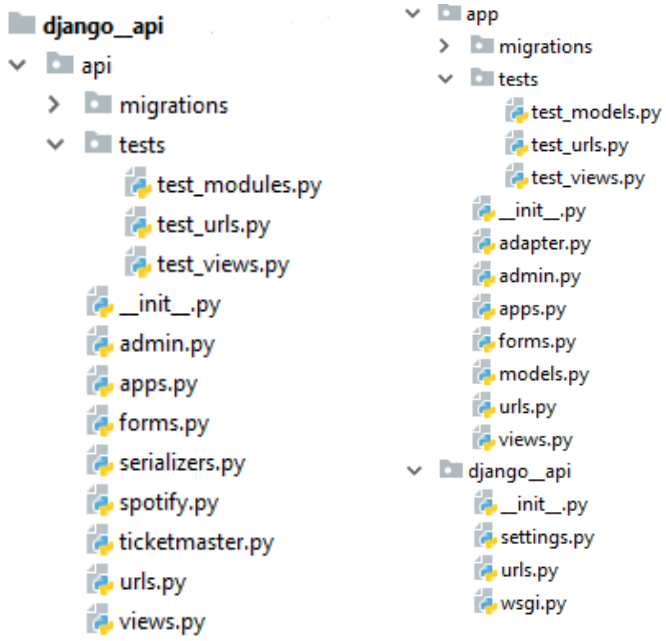


Figure 3.12: Back-End File Structure

The app application handles, all the front-end Web page rendering, this is accomplished in the urls.py and views.py files, it handles the creation of new users in models.py and the views.py. When a user profile is created, two additional profiles are required called Profile and Spotify, these contain personal information such as age and gender for each user and information retrieved from their Spotify accounts such as the playlist data required for the recommendation model. These are created on signup using signals, which are essentially triggers that are fired when a user is created, their implementation will further be detailed in the Development chapter. Also handled in the app application is the picture upload and personal details updating, both of which are accomplished using forms in forms.py.

The api application is responsible for all functionality that is internally provided or externally accessed by the application via API calls. As all the functionality of the Web application has been designed for provision via API, this application handles most of the business logic. Furthermore, all functionality is accessible via endpoints which are defined in urls.py, the logic for which can be found mostly in views.py, spotify.py and ticketmaster.py. Signals are also utilized in this application to perform several tasks, some of which are performed synchronously such as gathering user information from Spotify's APIs and retrieving the event

information from Ticketmaster's APIs. Other tasks are deployed asynchronously for efficiency including the API call to the recommendation system and updating the user's profile with recommended events which are generated using the results of the recommendation model and further calls to the Ticketmaster API.

Retrieving information from Spotify is an integral part of this project. To gain access to the Spotify APIs it is necessary to register the application in their developer console, from which a Client ID and Secret are retrieved, this is also where the redirect URL is defined for users signing up to the app using the social authentication feature which utilizes Spotify's Oauth2 service.

Figure 3.13: Spotify Developer Dashboard

For an application to retrieve data from the API it is required to provide its access tokens for identification which Spotify uses to monitor the requests being made to ensure all rules are being followed and limits are not being exceeded. To retrieve sensitive user data Spotify requires the application to provide not only its access tokens but also a token for the user which must be refreshed after every API call with a separate refresh token, this token is initially attained either when the user registers for the application using Spotify. If the user does not register through Spotify, they are prompted to connect to their Spotify account every time they log in until the token is attained.

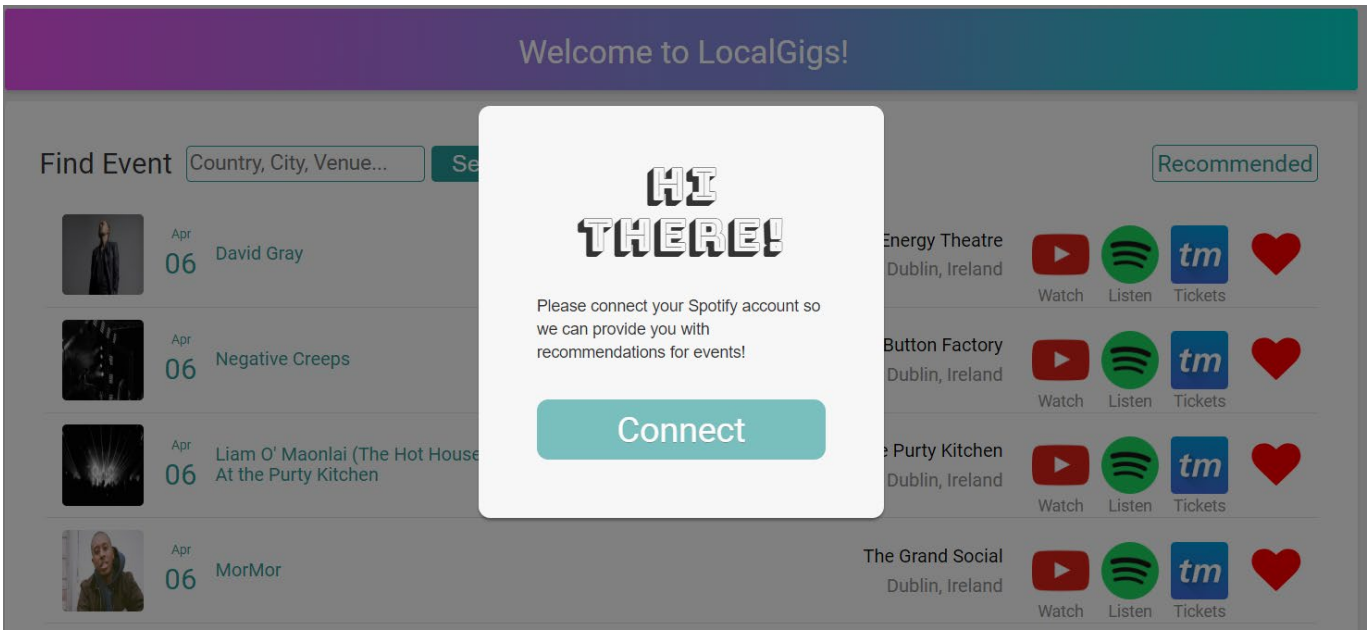


Figure 3.14: Prompt to Connect Spotify Account

Without the ability to get up to date real-time event information from the Ticketmaster Discovery API this project would not be possible. Getting access to this data required steps like those described for the Spotify API. The application was registered in Ticketmaster's developer console where a Secret Key was provided. Ticketmaster sets a hard limit of five thousand API request per day and a maximum of five requests per second. These restrictions affected the project greatly, particularly the number of requests per second, this forces the application to slow down its request making process, how this was accomplished will be detailed further in the Development chapter, this causes rendering of the home page to lag slightly (approximately three seconds) as it needs to load all of the events taking places in the users location for the next one hundred and eighty days.

The back-end also handles all user modification of the database such as saving and deleting events using the heart or trash icons. Django's model serialisers were used for adding or removing these events from the database.

The backend is designed to be packaged and deployed as a Docker container, this was very useful for rapidly deploying, maintaining and updating the application. As explained in the Literature Review, a Dockerfile was used to containerize the application with all the requirements of the systems being extracted from the local environment to a requirements.txt file, which was used to ensure all the relevant libraries and dependencies are installed in the application. The entry point for the container is the Django admin command (python manage.py runserver 0.0.0.0:8000) which maps the container to the IP address of the current machine and serves it over port eight thousand.

3.4.3. Recommender Design

Developing on what was explained in the Research chapter, the most important factors when developing a predictive analytics model are the business problem and the data. The business problem determines what the model should attempt to predict, and the data determines what approach needs to be taken to solve the business problem. Hence, the recommendation model is built following an Item-based Collaborative Filtering approach utilizing the Singular Value Decomposition (SVD) algorithm. This approach was chosen by following the CRISP-DM methodology detailed below.

Business Understanding, the business problem that needed to be solved for this project was the ability to recommend live music events to users based on a profile of musical interests that they provide. It was decided that the type of predictive analytics most suited to this problem is a recommendation system.

Data Understanding, the main source of data for this project is the #nowplaying dataset. This dataset in its unadulterated form was a CSV file comprised of nearly thirteen million entries in

the following format a hashed username (String), artist name (String), song name (String) and playlist name (String) (figure 3.14). The dataset was examined for ways it could be used with a recommendation system and it was determined that a Collaborative Filtering approach would work well with this dataset if it was aggregated based on the user and the artist in order to produce a count of the number of tracks the user had for each artist which could be used as a rating system indicating a user's interest in that artist.

```
"a474258cd4f41a4eee71f5cc645c1b3b","Phoenix","Sometimes in the Fall","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","The Doors","Soul Kitchen","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","Portishead","Sour Times","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","Brand New","Sowing Season (Yeah)","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","david bowie","Space Oddity","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","The Doors","Spanish Caravan","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","Pink Floyd","Speak To Me","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","Modest Mouse","Spitting Venom","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","Matt Pond PA","Spring Provides","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","Ben E. King","Stand by Me","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","Modest Mouse","Steam Engenius","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","The Doors","Stoned Immaculate","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","Pink Floyd","Stop","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","Billie Holiday","Stormy Blues","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","The Doors","Strange Days","IPOD"
"a474258cd4f41a4eee71f5cc645c1b3b","Coldplay","Strawberry Swing","IPOD"
```

Figure 3.15: #Nowplaying Data Structure

Data Cleaning and Preparation, after an understanding of the data had been accomplished, the task of cleaning the data of any irregularities and aggregating it needed to be accomplished, this was mostly completed in several pre-processing files in the BasicRecommender, figure3.21. The data was examined extensively for missing values, malformed entries, outliers and general rating patterns of the user.

```
# Remove all backslash occurrences in artists name
df['artist'].replace(regex=True, inplace=True, to_replace=r'\\', value=r'')

# Remove all forward slash occurrences in artists name
df['artist'].replace(regex=True, inplace=True, to_replace=r'/', value=r'')

# Remove all commas in artists names
df['artist'].replace(regex=True, inplace=True, to_replace=r',', value=r' &')

# Remove all single quotes in artists names
df['artist'].replace(regex=True, inplace=True, to_replace=r"'", value=r'')

# Remove all quotation mark occurrences in artists name
df['artist'] = df['artist'].map(lambda x: x.replace('"', ''))
df['artist'] = df['artist'].map(lambda x: x.lstrip('"').rstrip('"'))

# If the user has more than 10 songs for an artist limit it to 10
over_ten_index = df['track_count'] >= 10
df.loc[over_ten_index, 'track_count'] = 10
```

Figure 3.16: Cleaning the #Nowplaying Dataset

Because the dataset was so large any rows with missing values were removed for simplicity, the same malformed entries kept appearing in the dataset and most likely were caused during the tweet web scraping process in the #nowplaying project, as they were almost always forward and backslash characters followed by quotation marks which are fairly common to see in Web scraped textual data. In order to handle the huge outliers found, (some users have the entire back catalogue for some artist in their playlists) and satisfy the rating requirements of a collaborative filtering system which works best with a fixed rating scale the rating a user could give to an artist i.e. the maximum number of tracks per user, per artist, was set to ten and the minimum to one.

Users		Artists	
0.01	1.00	0.01	1.0
0.02	1.00	0.02	1.0
0.03	1.00	0.03	1.0
0.04	2.00	0.04	1.0
0.05	2.00	0.05	1.0
0.10	6.00	0.10	1.0
0.25	33.00	0.25	1.0
0.50	110.00	0.50	1.0
0.75	265.00	0.75	3.0
0.90	494.70	0.90	12.0
0.95	710.00	0.95	29.0
0.96	786.00	0.96	38.0
0.97	891.22	0.97	55.0
0.98	1055.74	0.98	90.0
0.99	1352.00	0.99	193.0
1.00	21967.00	1.00	4645.0

Figure 3.17: User and Artists Quantile Tables

Furthermore, as the dataset is so large it is important that it was condensed as much as possible to achieve the highest information density possible which greatly affects the accuracy and efficiency of the system. The quantile tables in figure 3.16 were produced which detail the number ratings a user has provided and the number of times an artist has been rated in percentage terms, reading from the top line it can be interpreted as one percent of users rated one item or less. To ensure the dataset had the highest information density possible any user that had rated five or artists or less and any artists that had been rated ten or fewer times were removed from the dataset this reduced the final dataset size to just under three million data points. The finalized format for the dataset can be seen in figure 3.17.

```
b0090327d7aa3267fe393414a3cff61b,Avicii,3
b0090327d7aa3267fe393414a3cff61b,Bruno Mars,3
b0090327d7aa3267fe393414a3cff61b,Carly Rae Jepsen,2
b0090327d7aa3267fe393414a3cff61b,David Guetta,2
b0090327d7aa3267fe393414a3cff61b,Enrique Iglesias,2
b0090327d7aa3267fe393414a3cff61b,Flo Rida,6
b0090327d7aa3267fe393414a3cff61b,Gym Class Heroes,2
b0090327d7aa3267fe393414a3cff61b,Jessie J,1
b0090327d7aa3267fe393414a3cff61b,Katy Perry,3
b0090327d7aa3267fe393414a3cff61b,Kelly Clarkson,2
b0090327d7aa3267fe393414a3cff61b,LMFAO,1
b0090327d7aa3267fe393414a3cff61b,Maroon 5,5
b0090327d7aa3267fe393414a3cff61b,Nicki Minaj,1
b0090327d7aa3267fe393414a3cff61b,P!nk,2
b0090327d7aa3267fe393414a3cff61b,Pitbull,3
b0090327d7aa3267fe393414a3cff61b,Rihanna,1
b0090327d7aa3267fe393414a3cff61b,Taylor Swift,4
```

Figure 3.18: Cleaned Data Structure

The graph's in figures 3.18 and 3.19 were produced to help visualize the rating patterns of users, as can be seen from the Distribution of Artist Ratings graph, nearly sixty percent of the data is made up of artists that have been rated one out of ten, in other words, the user had one song for that artists in their playlists. The Distribution of Number of Ratings Per User graph follows a descending exponential pattern and indicates that most users have provided ratings for two hundred artists or fewer.

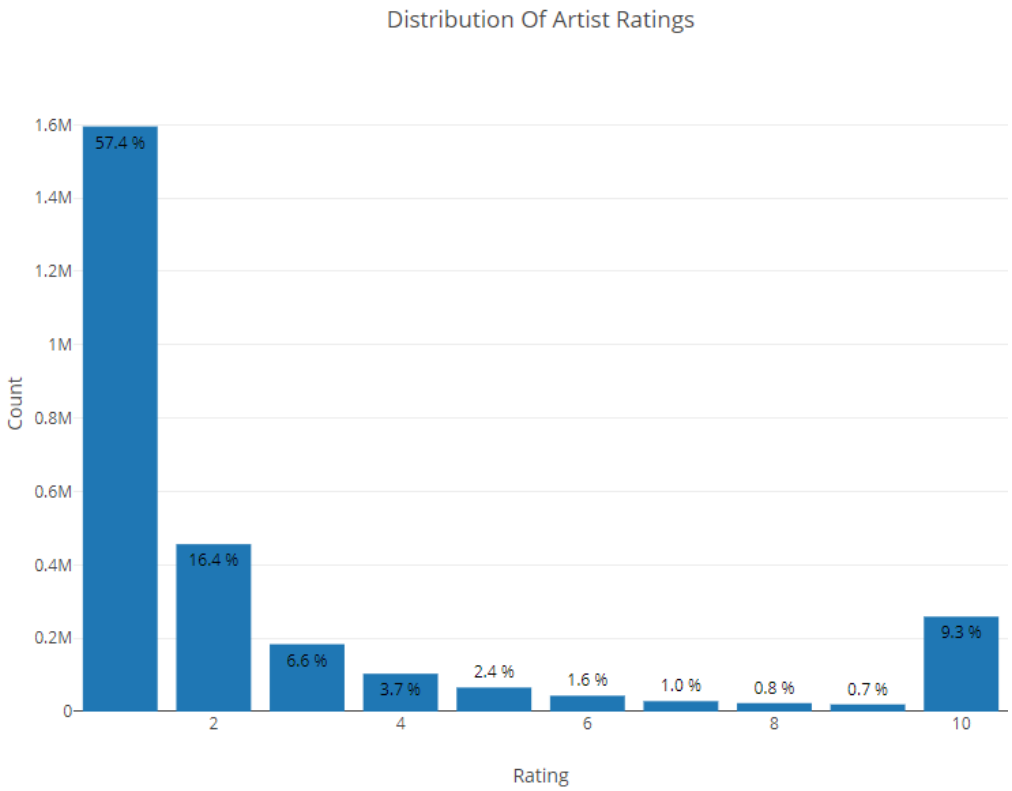


Figure 3.19: Distribution of Artist Ratings

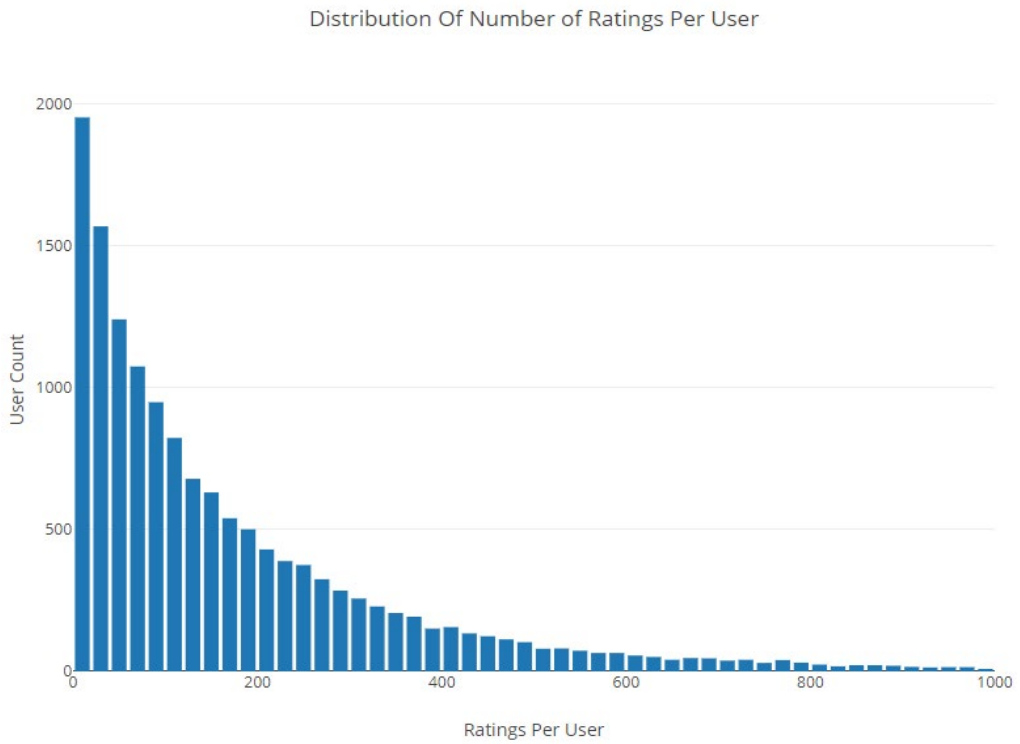


Figure 3.20: Distribution of Ratings Per User

Modelling and Evaluation, proceeding on after the data was cleaned and prepared several machine learning algorithms that are applicable for Collaborative Filtering were tested. These include SVD, SVD++, SlopeOne, numerous KNN variants and more. The performance of the different algorithms can be seen in table 3-1, these results are based on a subset of the data containing one hundred thousand data points, this smaller dataset was necessary due to long running times and memory errors.

Algorithm	RMSE	Fit Time	Test Time
SVD	2.588286	5.263005	0.381352
SVD++	2.707214	350.4747	9.1017
KNN-Baseline	2.709988	0.28833	1.436504
SlopeOne	2.74622	35.55418	6.048703
KNN Z-Score	2.76106	0.136544	1.207089
KNN-Means	2.765438	0.087179	1.034419
Co-Clustering	2.807426	4.110596	0.246487
NMF	2.882904	7.273558	0.257143
KNN	2.943408	0.06433	0.969662

Table 3-1: Recommendation Algorithm Performance Test

After testing the numerous algorithms known to be applicable to collaborative filtering recommendation systems the results showed that SVD had the lowest Root Mean Square Error (RMSE), even beating out SVD++ which was surprising. RMSE is the average error of all the algorithms incorrect predictions and is commonly regarded as the most accurate way to determine a recommendation systems accuracy, as can be seen from the results SVD is on average approximately seventy-five percent accurate as the rating scale is from one to ten.

```python
# Cross validate algorithms and retrieve their RMSE score, test and train time
def cross_validate_algos():
    benchmark = []
    # Iterate over all algorithms
    for algorithm in [SVD(), SVDpp(), SlopeOne(), NMF(), NormalPredictor(), KNNBaseline(),
                      KNNBasic(), KNNWithMeans(), KNNWithZScore(), BaselineOnly(), CoClustering()]:
        print(str(algorithm))
        # Perform cross validation
        results = cross_validate(algorithm, train_data, measures=['RMSE'], cv=3, verbose=False)
        print('Finished cross validation')
        # Get results & append algorithm name
        tmp = pd.DataFrame.from_dict(results).mean(axis=0)
        tmp = tmp.append(pd.Series([str(algorithm).split(' ')[0].split('.')[-1]], index=['Algorithm']))
        benchmark.append(tmp)

    benchmarked_algos = pd.DataFrame(benchmark).set_index('Algorithm').sort_values('test_rmse')
    print(benchmarked_algos.head())
    benchmarked_algos.to_csv('../Dataset/BenchmarkedAlgos.csv', sep=',')
```

Figure 3.21: Cross Validate Algorithms

This error value would be expected to decrease as the algorithm is provided with more data, the subset that was used to produce these results does not contain a large diversity of users or artists and such this is reflected in the results. The results achieved in this test enforced the discoveries outlined in the Literature Review that illustrated the effectiveness of matrix factorization techniques in collaborative filtering recommendation systems, as such the SVD algorithm was chosen for the recommendation system.

Deployment, as the recommendation system had been developed in Python the decided platform for deployment was the Django framework. Flask had been considered as it is geared towards providing microservices and is slightly more lightweight than Django, but it was decided the time that would have been used to learn the new framework could be put to better use developing the mobile application. The method of deployment to the server was again Docker, the recommendation system was Dockerized in the same manner as the back-end of the Web application.

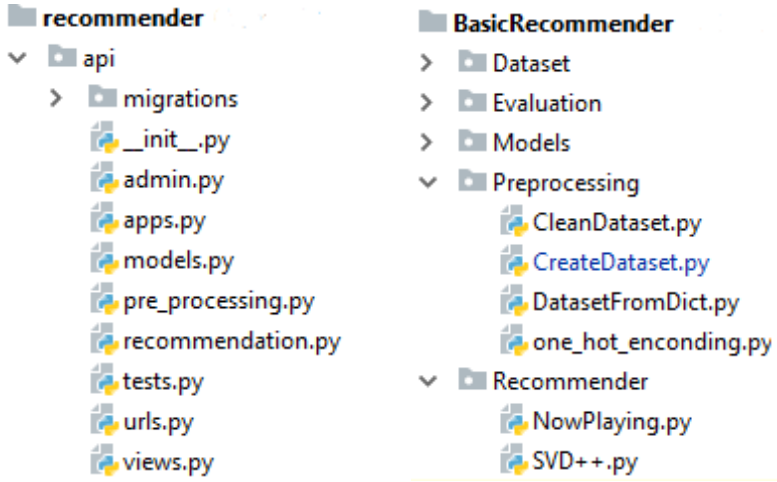


Figure 3.22: Recommender System File Structure

3.4.4. Database Design

Building on the Literature Review chapter, PostgreSQL is the database management system of choice for this project, the benefits of which have been explained in the aforementioned chapter. The database was initially constructed in AWS's RDS where it is accessible via an endpoint. The database was monitored in PgAdmin4 and connected to the Web application via this endpoint and the creation of the database structure was accomplished entirely in Django utilizing its Object Relational Mapping (ORM) system. This allows for all processes pertaining to the database to be handled entirely as objects in Python code. Django's ORM system is fantastic, other ORM implementations had been tested prior to, and during this project such as, Spring Boots Java Persistence API (JPA) and Node.js Sequelize, both of which are far less intuitive, Django's implementation seemed to just work.

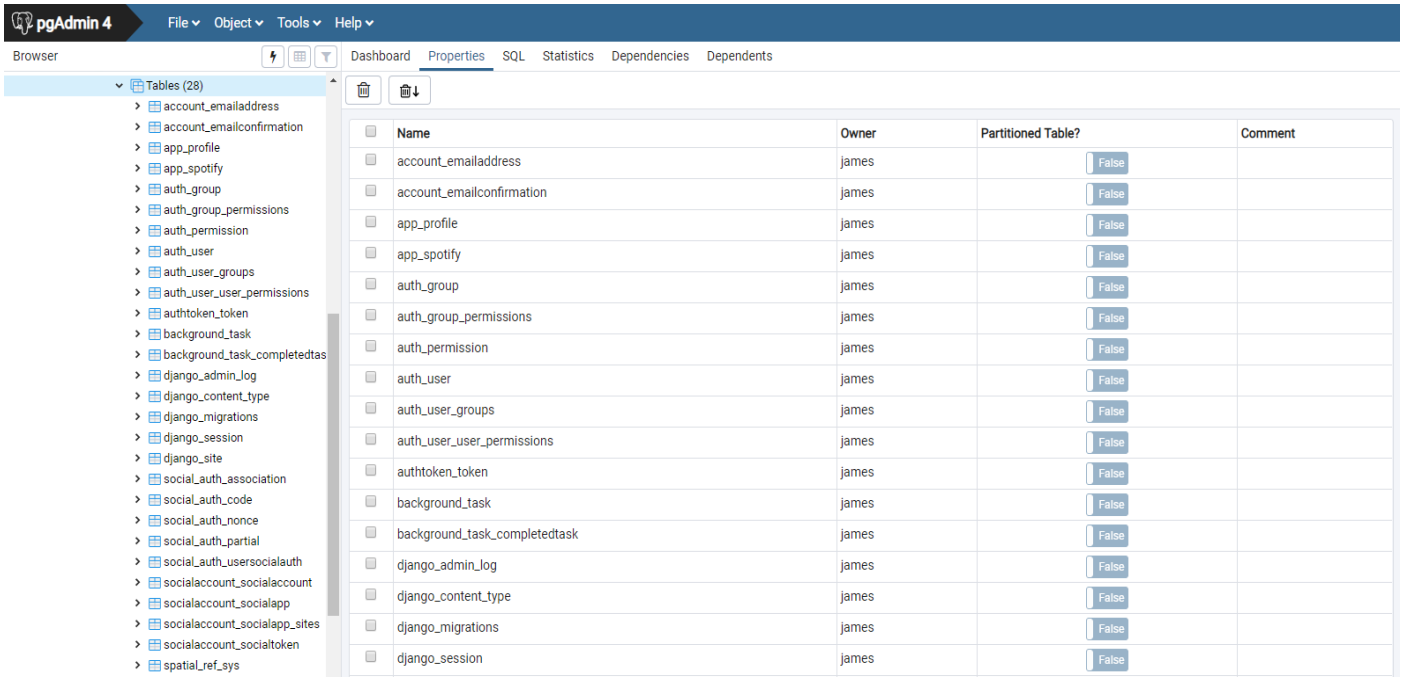


Figure 3.23: pgAdmin4 DB Monitoring Interface

Django provides a number of boilerplate table structures including the auth_user, account_emailaddress, django_admin_log, migrations and session tables that can be seen in the Entity Relationship Diagram (ERD) in figure 3.23, these help provide a basic mechanism to manage users, including creating, updating, deleting and ascertaining if a user is authenticated and if so what are their privilege levels and to keep track of the currently active sessions and any changes that are made to the database (migrations). The ERD depicts a small portion of the entire system, it highlights some of the key tables.

- auth_user – The main user table, entries are created here for every new user, privilege levels are tailored such as is_staff and is_superuser and active users are denoted by the is_active attribute

- auth_token – Stores the authentication tokens for each user, these are generated uniquely for each user upon registration using signals

- account_emailaddress – Ensures the email address is unique to each user and is partly responsible for linking the email provided upon registration and the email used to access the users Spotify account

- app_spotify – Generated uniquely upon registration via a signal, it stores all user information gathered from Spotify

- app_profile – Generated uniquely upon registration via a signal, it stores any additional details of the users such as the list of saved events and the list of recommended events

- background_task – Stores the asynchronous background tasks before they executed

- django_admin_log – Tracks all admin activity in the application

- migrations – Tracks all modifications performed on the database, can be used to easily roll back the database to previous configurations

- social_auth_usersocialauth – Generated uniquely for each user when they link their Spotify account

- session – Tracks the active sessions

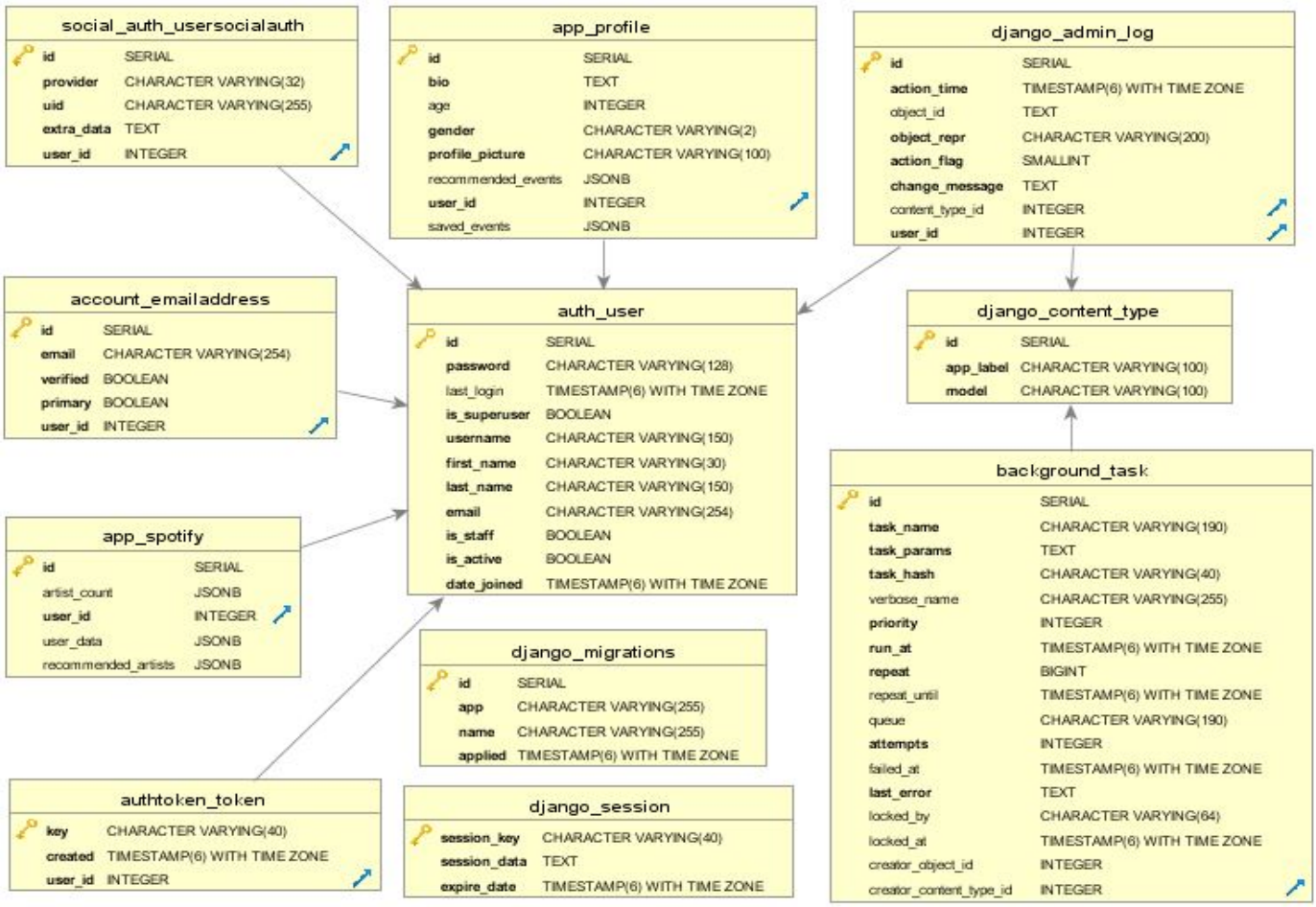


Figure 3.24: Entity Relationship Diagram (ERD)

3.5. Conclusions

In this chapter we looked at the design of the system, first exploring the methodology that is used in the development process. Next, the technical architecture was outlined, this described the design of the entire application including the layout and stylistic decisions regarding the front-end of the Web and mobile applications. Following on we looked at the architecture and deployment of the Web applications back-end, which preceded the architecture and deployment of the predictive analytics model. This contained numerous steps involving the analyzation, cleaning and preparation of the data and the design of the recommendation system. Finally, we examined the structure of the database and investigated the roles of some key tables.

Based on the key themes discussed in this chapter, the next chapter will cover the development process and will be revisiting many of the same issues covered here with a focus on the technical implementation of these areas.

4. System Development

4.1. Introduction

This chapter continues the issues explored in the previous chapter where the key design concepts were presented. These design concepts will be developed further in this chapter and their technical implementation will be detailed. This will be accomplished by highlighting the key features and explaining the code required to build them. This chapter is broken into sections of development each of which is representative of the different components of the project. The front-end will examine the development of the user interfaces, back-end will demonstrate the development of the business logic of the Web Application, recommender will outline the operation of the item based collaborative filtering recommendation system and finally, deployment will detail the deployment and hosting of the project.

4.2. Front-End Development

The following section will discuss the methods used in the final development of the Web and mobile applications. The key functionality will be outlined for each page of the applications. Developing on what was previously mentioned in the Design chapter, the front-end of the Web application was developed in the Django framework this allowed for Python to be used server-side for much of the logic and to quickly render components for the front-end when required. The mobile application was developed in the Ionic framework which utilizes Angular, which was learned for this project. The development, therefore, follows an Angular architecture, the functionality for each page is developed in separate TypeScript files and any access to external resources (APIs) is handled in the Rest service provider class.

4.2.1. Web Application

4.2.1.1. Base and Landing

Continuing from what was explained about Django's templating system in the Design chapter, it follows a single webpage approach the base.html template is the parent page that all other child webpages are rendered in. This is accomplished by using tags for the head and body in which the content is rendered.

```
{% block head %}
    {# All template heads are rendered here #}
{% endblock %}
```

```
{% block content %}
    {# All template bodies are rendered here #}
{% endblock %}
```

Figure 4.1: Template Rendering Blocks in base.html

Any components that are required throughout the entire front-end of the web app is handled in this file. This includes the navbar, which is personalized to each user upon authentication. If a user is authenticated their name and profile picture is displayed to them, if no profile picture has yet been provided it defaults to a generic head, users can upload a picture if they wish or if they have a picture in their Spotify account the profile will default to that. Furthermore, if a user is authenticated the navbar can be used to log out which renders as a modal popup confirmation.

Figure 4.2: Unauthenticated Navbar

Figure 4.3: Authenticated Navbar

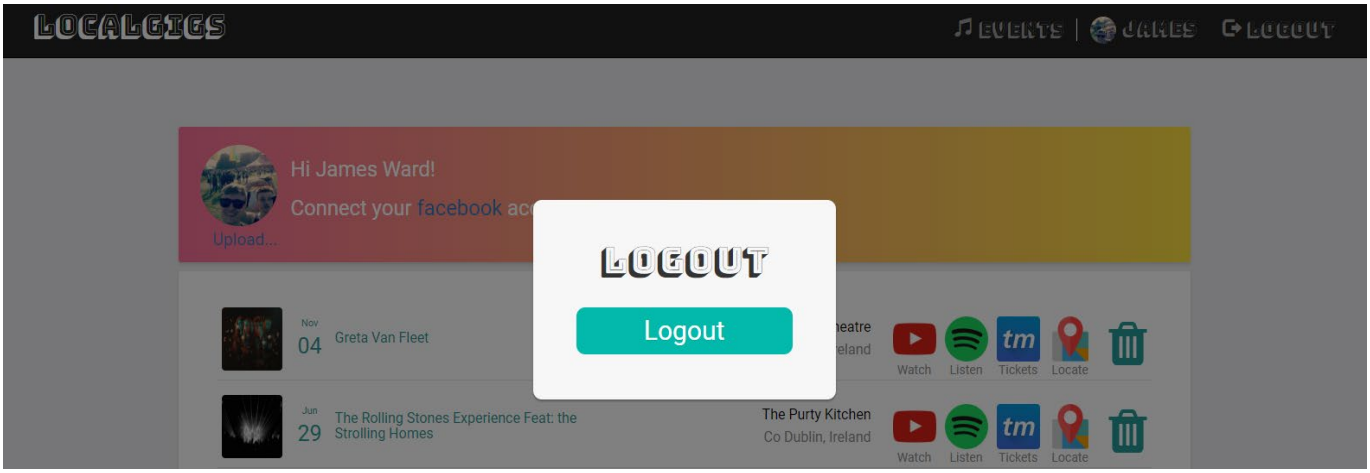


Figure 4.4: Modal Logout

The navbar is built and rendered using bootstrap and is controlled using Django. If the user is not authenticated i.e. they are not logged into their account, they will be displayed the un-authenticated navbar which provides links to login and sign up both of which are rendered as modal popups as can be seen in figure 3.7, the user has the option to log in or sign up manually both of which use standard Django forms which validate the data apply a CRSF token and then directly query the database and either return a validated user or an error.

The user can also log in and sign up using Spotify social authentication. Spotify uses Oauth2 for its authentication process which is handled by the Python library social_auth when the user logs in or if the user signs up for the Web application using Spotify. If they do not sign up using Spotify the authentication process is handled later in the app. The landing page essentially just implements the functionality developed in the base template.

```
<div id="content">
    <nav class="navbar navbar-inverse" style="...">
      <div class="container-fluid">
        <div class="navbar-header">
          <a class="navbar-brand" style="..." href="/">LocalGigs</a>
        </div>
        <ul class="nav navbar-nav navbar-right">
            {% if user.is_authenticated %}
                <li><a href="{% url 'app:home' %}"><i class="glyphicon glyphicon-music"></i>  Events</a></li></li>
                <li id="sperator" role="separator" class="divider"></li>
                <li>
                {% if user.spotify.user_data.images.0.url %}
                    <a href="{% url 'app:profile' %}"><span>
                    <img id="profile_pic" src="{{ user.spotify.user_data.images.0.url }}" style="..."/>
                {% else %}
                    <a href="{% url 'app:profile' %}"><span class="glyphicon glyphicon-user">
                {% endif %}
                </span>  {{ user.first_name }}</a></li>
                <li><a href="account/logout" data-toggle="modal" data-target="#logout-modal" style="...">
                    <span class="glyphicon glyphicon-log-out"></span>  Logout</a></li>
            {% else %}
                <li><a href="account/signup"><span class="glyphicon glyphicon-user"></span>  Sign Up</a></li>
                <li><a id="login" href="" style="..." data-toggle="modal" data-target="#login-modal">
                    <span class="glyphicon glyphicon-log-in"></span>  Login</a>
                </li>
            {% endif %}
        </ul>
      </div>
    </nav>
</div>
```

Figure 4.5: Navbar Display Control

If statements are used in the Bootstrap navbar to control what element are rendered based on user authentication and database contents.

4.2.1.2. Home

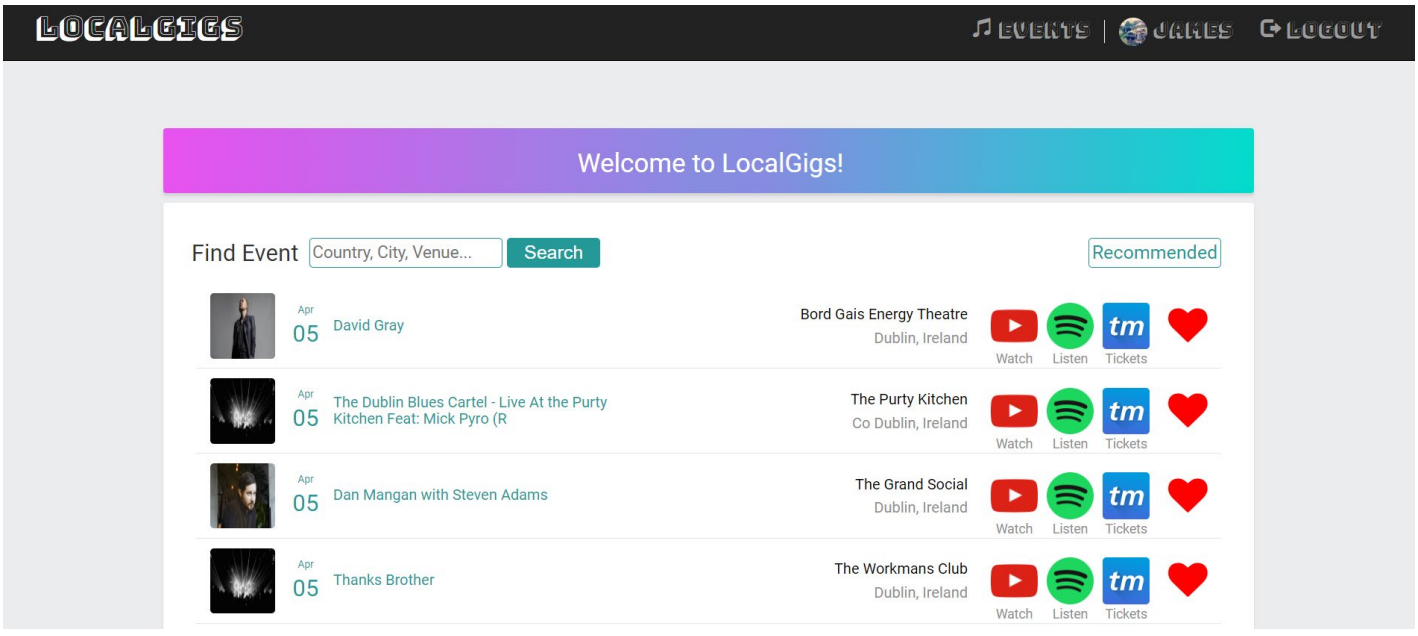


Figure 4.6: Home Page

The Home page is the point of entry for an authenticated user. When the page loads it first checks to see if the user has linked their Spotify account. If no Spotify account is found they are greeted with the prompt to connect their account, shown previously in the Design chapter

(figure 3.13). This button redirects the user to the Spotify authentication page, contrarily if the user decides not to link their account, they are provided with all the functionality of the website bar the ability to receive recommended events.

The focal point of the home page is the list of events. This is generated dynamically using a template sub-rendering process by which the events.html template is included in the Home page and populated with the response from the GET request to the Ticketmaster API which returns all music events in the users location for the next 180 days, this list of event objects is passed to the events template via context variable.

```
<div id="main_events_div" class="container">
    <div id="event_list">
        {% include 'app/events.html' %}
    </div>
</div>
```

Figure 4.7: Include events.html in Div

The events template page (figure 4.10), is a structure that each event div follows. It accepts the context variable event_list, which is a list of event objects that is looped over. On each pass, a new event structure is created and given an id and the details of each object are placed within the structure. An event div consists of

- Image – An image of the artist
- Date – The date of the event
- Text – The details of the location of the event
- YouTube Icon – A link to watch the artists music videos on YouTube
- Spotify Icon - A link to listen to the artists on Spotify
- Ticketmaster Icon - A link to purchase tickets for the event
- Heart Icon – A button that fires an ajax request to save the event

When the Heart Icon fires the save event function is passes the parameter “event”, this is the event object used to generate the div. This object is sent as an ajax POST request to the backend to update the saved_events column in the profile table.

```
function saveEvent(event) {
    csrfTokenSetup();
    $.ajax({
            url: '{% url 'api:save_event' %}',
            type: 'POST',
            data: {save_event: JSON.stringify(event)},
            success: function () {
                console.log("event sent")
            }
    });
}
```

Figure 4.8: Save Event POST Request

The recommended events, accessible by clicking the recommended button, which in turn hides the currently displayed generic events, are displayed using a similar process, the slight difference being the rendering of the template. This was accomplished by using a GET request which passes a different context variable to the events template that contains the list of recommended event objects stored in the database for a given user.

```
function getRecommendedGigs() {
    $.get("{% url 'api:render_recommended_events' %}")
            .done(function(data) {
                recommended_event_list = data;
                let div = document.createElement("div");
                div.id = "recommended_event_list";
                div.style.display = "none";
                div.innerHTML = data;
                $("#main_events_div").append(div);
            });
}
```

Figure 4.9: Render Recommended Events GET Request

This method was chosen rather than just rendering both the normal events and recommended events on page load so that the GET request could be performed periodically and the div updated without the user having to refresh. This is because the recommendation model takes some time to return its recommendations and the system is designed to search Ticketmaster for events relating to the artists the user has already shown an interest in before it generates it predictions, to save time and provide a better user experience.

```
{% for event in event_list %}
    <div class="event_container">
        <div id="event{{ forloop.counter }}" class="event_div">
            <div class="image_div">
                <div class="image_container">
                    <img class="image" src="{{ event.image }}"
                        onerror="this.onerror=null;this.src='/static/app/images/default_artist.png';"/>
                </div>
            </div>
            <div class="date_div">
                <div class="date_month">{{ event.date.month }}</div>
                <div class="date_day">{{ event.date.day }}</div>
            </div>
            <div class="text_div">
                <h2 class="event_name">{{ event.name }}</h2>
                <div class="venue_location">
                    <div class="venue">{{ event.venue.name }}</div>
                    <div class="location">{{ event.venue.city }}, {{ event.venue.country }}</div>
                </div>
            </div>
            <div class="links_div">
                <div class="youtube_div">
                    <div class="youtube_img_container">
                        <a href="{{ event.youtube_url }}" class="youtube_link">
                            <img class="youtube_image" src="/static/app/images/youtube_icon.png"/>
                        </a>
                    </div>
                    <div class="youtube_text">Watch</div>
                </div>
                <div class="spotify_div">
                    <div class="spotify_img_container">
                        <a href="{{ event.spotify_url }}" class="spotify_link">
                            <img class="spotify_image" src="/static/app/images/spotify_icon.png"/>
                        </a>
                    </div>
                    <div class="spotify_text">Listen</div>
                </div>
```

```
                <div class="ticketmaster_div">
                    <div class="ticketmaster_img_container">
                        <a href="{{ event.ticketmaster_url }}" class="ticketmaster_link">
                            <img class="ticktmaster_image" src="/static/app/images/ticketmaster_icon.png"/>
                        </a>
                    </div>
                    <div class="ticketmaster_text">Tickets</div>
                </div>
            </div>
            <div class="save_div">
                <a onclick="saveEvent({{ event }})"><i class="save_heart fa fa-heart fa-3x"></i></a>
            </div>
        </div>
    </div>
    <hr class="event_seperator">
{% endfor %}
```

Figure 4.10: Event HTML Structure

Finally, the last piece of functionality in the Home page is the search bar, this is used to search Ticketmaster's API for keywords relating to an artist's name. This process is handled by the back-end for efficiency and so it will be discussed further in that section, the rendering of the returned events is handled in the same manner as the initial rendering of the page.

4.2.1.3. Profile

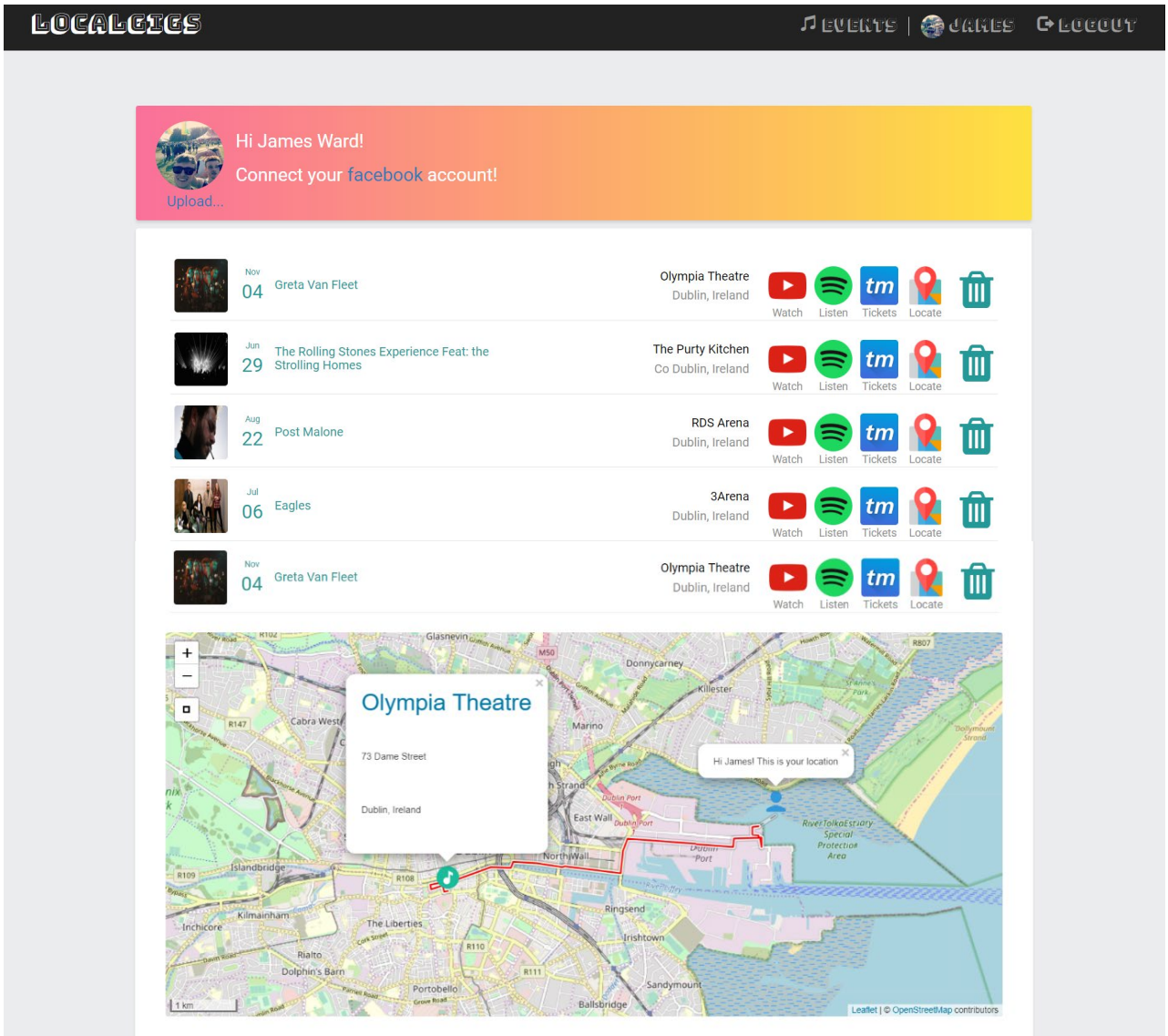


Figure 4.11: Profile Page

The main utilities of the profile page from top to bottom are photo upload, connectivity to Facebook account for alternative social authentication option, this unfortunately no longer works due to a clampdown from Facebook over the Cambridge Analytica controversy, management of saved events and locating and routing to events. By clicking on the upload link under the user's profile picture an alternative picture can be uploaded, this utilizes a Django form to upload the picture which is stored in a file structure on the system that is generated uniquely for each user, in deployment this file structure would be created on a separate file server.

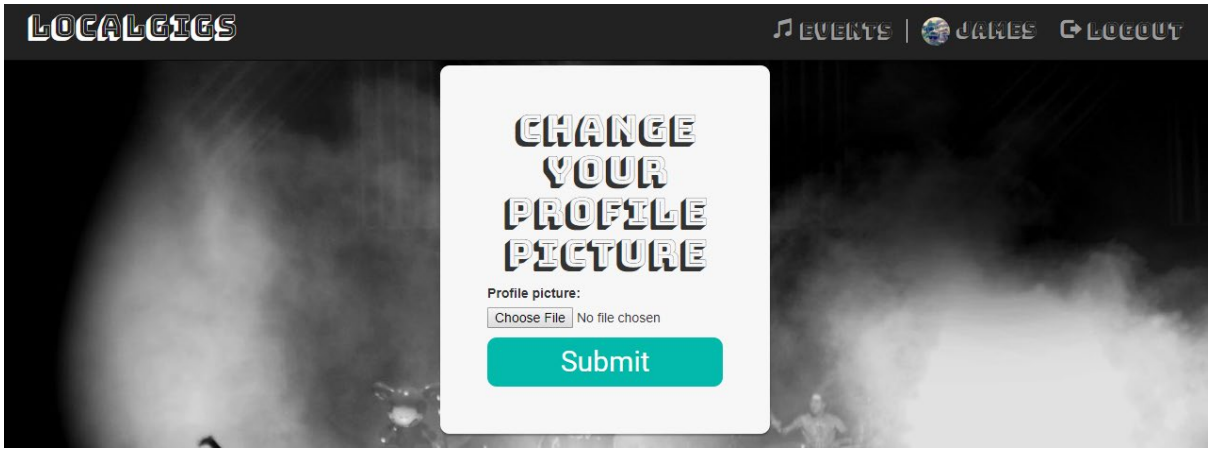


Figure 4.12: Upload Profile Picture Page

Like the home page, the main point of focus for the Profile page is the list of events, this list is rendered in the same way as the Home page with minor changes to the layout, the removal of the save button and addition of the delete and locate button. Events are deleted in much the same way they are saved, an ajax POST request is sent to the back-end with a parameter containing the event object which is then removed from the database.

```
{# Initialize map #}
function main_map_init (map, options){
    my_map = map;
    navigator.geolocation.getCurrentPosition(function (position) {
        let marker = L.marker([position.coords.latitude, position.coords.longitude],
            {icon: createUserIcon()}).addTo(map)
            .bindPopup('Hi {{ user.first_name }}! This is your location');
        centerLeafletMapOnMarker(map, marker);
        user_latlong = [position.coords.latitude, position.coords.longitude];
    });
}
```

Figure 4.13: Leaflet Map with User Marker

The map is built using the open source JavaScript library Leaflet. Leaflet was chosen because it is free to use and highly customizable. When the Profile page is loaded the getCurrentPosition method gathers the user's latitude and longitude and places a custom user marker on the map at their location. The custom markers for user and event can be seen in figure 4.14 they are created from simple PNG files.

```
navigator.geolocation.getCurrentPosition(function (position) {
    let marker = L.marker([position.coords.latitude, position.coords.longitude],
        {icon: createUserIcon()}).addTo(map)
        .bindPopup('Hi {{ user.first_name }}! This is your location');
    centerLeafletMapOnMarker(map, marker);
    user_latlong = [position.coords.latitude, position.coords.longitude];
});
```

Figure 4.14: Retrieve Current Location of User

```
function createUserIcon() {
    return L.icon({
      iconUrl: '/static/app/icons/userIcon.png',
      iconSize: [35, 35],
      iconAnchor: [17, 35],
      popupAnchor: [0, -35],
      shadowSize: [68, 95],
      shadowAnchor: [22, 94]
    });
}
```

```
function createEventIcon() {
    return L.icon({
      iconUrl: '/static/app/icons/musicIcon.png',
      iconSize: [35, 35],
      iconAnchor: [17, 35],
      popupAnchor: [0, -35],
      shadowSize: [68, 95],
      shadowAnchor: [22, 94]
    });
}
```

Figure 4.15: Custom Leaflet Markers

When the user clicks the locate button the latitude and longitude of the venue are extracted from the event object and an event marker is placed on the map at that location. Checks are performed to ensure no other event markers are on the map and the latitude and longitude of the user and venue are then passed to a function which adds the route to the venue, this function makes a GET request to the Open Source Routing Machine (OSRM), a free open source routing library for leaflet using the Routing library, this returns a number of points and lines in the form of a route object which is layered onto the map. Each time the locate button is pressed the existing route and venue layers are removed to keep the map appearing clean and uncluttered.

```
venueRoute = L.Routing.control({ waypoints: [
    L.latLng(user_latlong[0], user_latlong[1]),
    L.latLng(Number(vLat), Number(vLong))
  ],
  maxZoom: 12,
  createMarker: false,
});
my_map.addControl(venueRoute);
```

Figure 4.16: Leaflet Routing Machine

4.3.2 Mobile Application

Due to time constraints, the mobile application was set at the bottom of the pile when it came to prioritizing the workload, with the bulk of the time being invested in the recommendation system and the web application. Unfortunately, due to this the mobile application was not completely finished and the events page did not develop much visually past what is shown in the Design chapter. However much of the functionality of the application is in place and will be detailed here.

Figure 4.17: Final Login and Register Screens of Mobile Application

Authentication of the Mobile application is handled using an Authorization Token these tokens were generate uniquely for each user when they created an account. The mobile application logs in the user by performing a GET request with the user's email and password to the token_login method in the back-end and receives a response containing information about the user including their Authorization Token. The data returned from the GET request is stored in local storage on the device so it can be used throughout the application, for security reasons, this is important so the user's token is not being passed around the application which could potentially be accessed by a malicious attacker. Registration via the mobile followed a similar approach.

```
// Make GET request to the Django API to log user in and retrieve profile data and auth token
userLogIn(email, password) {

  return new Promise( executor: (resolve, reject) => {
    this.http.get(this.baseUrl + 'login/?email=' + email + '&password=' + password)
      .subscribe( options: data => {
        resolve(data);
      },
          err => { reject(err); console.log(err);
      });
  });
}
```

Figure 4.18: Log in GET Request Sent from Mobile Application

```python
# API login for mobile application that returns the users profile information
# and an authentication token
@api_view(["GET", ])
@permission_classes((permissions.AllowAny,))
def token_login(request):
    if (not request.GET["email"]) or (not request.GET["password"]):
        return Response({"detail": "Missing email and/or password"}, status=status.HTTP_400_BAD_REQUEST)

    user = authenticate(email=request.GET["email"], password=request.GET["password"])
    if user:
        if user.is_active:
            login(request, user)
            try:
                my_token = Token.objects.get(user=user)
                first_name = user.first_name
                last_name = user.last_name
                return Response({"token": "{}".format(my_token.key),
                                 "first_name": "{}".format(first_name),
                                 "last_name": "{}".format(last_name),
                                 },
                                status=status.HTTP_200_OK)
            except Exception as e:
                return Response({"detail": "Could not get token"}, status=status.HTTP_400_BAD_REQUEST)
        else:
            return Response({"detail": "Inactive account"}, status=status.HTTP_400_BAD_REQUEST)
    else:
        return Response({"detail": "Invalid User Id of Password"}, status=status.HTTP_400_BAD_REQUEST)
```

Figure 4.19: Mobile GET Request Received in token_login Method

Once the user's authorization token is retrieved the mobile application can access all the functionality of the Web application provided via API calls by providing the Token to identify the user. This was accomplished by appending an authorization header to all API request that required it.

```javascript
// Create authorization headers for necessary requests
createHeaders(){
  const token = this.getAuthToken();
   let headers = new HttpHeaders().set('Authorization', 'Token ' + token);

  return headers;
}

// Gets the users auth token from localstorage
getAuthToken() {
  const storedUser = JSON.parse(localStorage.getItem( key: 'user'));
  this.userDetails = storedUser;
  const token = this.userDetails.token;

  return token;
}
```

Figure 4.20: Authorization Header

Depicted in figure 3.10, is the list of events, this list is produced using a similar mechanism to the events list in the Web application. Angular utilizes programming functionality in its HTML files that allow data structures to be passed to them and utilized to dynamically generate display components. A GET request is made to the Django backend to retrieve events from Ticketmaster the response is passed to the template where an ngFor loop is used to iterate over the response which contains a list of event objects and produce the individual event items.

As the mapping feature is implemented entirely in HTML, CSS and JavaScript the same Leaflet mapping libraries and methods are used in the construction of the map for the mobile application as were used in the Web application with some modifications to work on a mobile.

```
<ion-list>
  <ion-item *ngFor="let gig of gigList">
    <ion-avatar item-start>
      <img id="gigImage" src="{{ gig.image}}" alt="" >
    </ion-avatar>
    <h2 id="gigName">{{ gig.name }}</h2>
    <br>
    <p id="gigDate">{{ gig.date }}</p>
    <p id="gigTime">{{ gig.time }}</p>
    <br>
    <a id="ticketLink" href="{{ gig.url }}">Buy tickets!</a>
    <br><br>
    <a *ngIf=" gig.youtube_url != undefined"  href="{{ gig.youtube_url }}">Listen now!</a>
    <br><br>
    <button block ion-button icon-end (click)="locateVenue(gig.venue_id)" >
      Find the venue!
    </button>
  </ion-item>
</ion-list>
```

Figure 4.21: Mobile Application Event Structure

```
getMusicEvents() {
  try {
    this.restProvider.searchEvents( search: "music")
      .then( onfulfilled: data => {
        this.gigList = data;
        console.log(this.gigList);
      });
  }
  catch (e) {
    console.log(e)
  }
}

getRecommendedEvents() {
  try {
    this.restProvider.getRecommendedEvents()
      .then( onfulfilled: data => {
        this.gigList = data;
        console.log(this.gigList);
      });
  }
  catch (e) {
    console.log(e)
  }
}
searchGigs() {
  // Try and search for gigs through the django api
  // If successful populate list in template
  try {
    this.restProvider.searchEvents(this.search.value)
      .then( onfulfilled: data => {
        this.gigList = data;
        console.log(this.gigList);
      });
  }
  catch (e) {
    console.log(e)
  }
}
```

Figure 4.22:Mobile Application Back-End Mostly Finished

Searching for events was one of the pieces of functionality that was finished at an early iteration of the development cycle and so the mobile application has that feature. The rest of the backend functionality of the mobile application such as retrieving the users recommended and saved events and saving and deleting events is all finished unfortunately there just was not enough time to finish off the front end of the application to fully implement these features.

```
// Make a GET request to the Django API which makes a GET request to the Ticketmaster API
// which returns a list of events.
searchEvents(search) {
  const headers = this.createHeaders();
  return new Promise( executor: resolve => {
    this.http.get( url: this.baseUrl + 'get_ticketmaster_events/?search='
      + search , options: {headers: headers} )
      .subscribe( next: data => { resolve(data); }, error: err => { console.log(err);
      });
  });
}

// Gets the users saved events from the database
getSavedEvents () {
  const headers = this.createHeaders();
  return new Promise( executor: resolve => {
  this.http.get( url: this.baseUrl + 'get_saved_events/', options: {headers: headers}
    ).subscribe( next: data => { resolve(data); }, error: err => { console.log(err);});
  });
}

// Gets the users recommended events from the database
getRecommendedEvents () {
  const headers = this.createHeaders();
  return new Promise( executor: resolve => {
  this.http.get( url: this.baseUrl + 'get_recommended_events/', options: {headers: headers}
    ).subscribe( next: data => { resolve(data); }, error: err => { console.log(err);});
  });
}
 // Saves an event to the database
 saveEvent () {
   const headers = this.createHeaders();
   return new Promise( executor: resolve => {
   this.http.get( url: this.baseUrl + 'save_event/', options: {headers: headers}
     ).subscribe( next: data => { resolve(data); }, error: err => { console.log(err);});
   });
 }

 // Gets the users interested list from db interested_gigs column
 deleteEvent () {
   const headers = this.createHeaders();
   return new Promise( executor: resolve => {
   this.http.get( url: this.baseUrl + 'delete_event/', options: {headers: headers}
     ).subscribe( next: data => { resolve(data); }, error: err => { console.log(err);});
   });
 }
```

Figure 4.23: Unimplemented Features on the Mobile

4.3. Back-End Development

This section will examine the final development of the back-end of the system involving the Web application. The back-end of the application which is developed in the Django framework is comprised of two separate applications as described in the Design chapter. As such the development of each of these applications will be outlined separately.

4.3.1. App Application

The app application (App), is responsible for the rendering of the front end of the Web application and all modifications made to the database. This section will expand on some of the features described in the front-end development and provide a detailed look at the back-end functionality behind the front-end features, it will also cover the automatic creation of new table entries for new users upon signup.

Firstly, the creation of new user-specific database tables was accomplished using Django's ORM. The table structures are defined in the models.py file and applied to the database using migrations. These tables are linked to the Django auth_user table with a one to one relationship by referencing the users id as a foreign key when creating the table, the on_delete parameter is passed and set to CASCADE so when a user is removed from the system all additional table entries related to that user are removed from the system.

```
# Spotify table model, stores the users access token and details on their playlist contents
# Defines what initial parameters and restrictions are put on it the fields
# This table is linked to the user table as a one to one relationship.
class Spotify(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    user_data = JSONField(blank=True, null=True)
    artist_count = JSONField(blank=True, null=True)
    recommended_artists = JSONField(blank=True, null=True, default=list)

    # Display the email as the identifier in django admin (Spotify)
    def __str__(self):
        return self.user.email
```

Figure 4.24: Django ORM Spotify Model

The example in figure 4.23, demonstrates the Spotify table which is used to hold all information of each user that is retrieved from the Spotify API. PostgreSQL and Django's ORM allow for complex data structures such as JavaScript Object Notation (JSON) to be stored in the columns of the table and allows for data structures to be used as default entries. This was particularly useful for storing the list of recommended artists and the list of event objects in the profile table.

```
{
  "date": {
    "day": "04",
    "month": "Nov"
  },
  "name": "Greta Van Fleet",
  "time": "19:00:00",
  "image": "https://s1.ticketm.net/dam/a/fa5/d936214a-f378-492f-a502-ba4133d20fa5_869791_TABLET_LANDSCAPE_3_2.jpg",
  "venue": {
    "city": "Dublin",
    "name": "Olympia Theatre",
    "address": {
      "line1": "73 Dame Street"
    },
    "country": "Ireland",
    "latitude": "53.344318",
    "longitude": "-6.266114",
    "venue_url": "https://www.ticketmaster.ie/Olympia-Theatre-tickets-Dublin/venue/198239"
  },
  "spotify_url": "https://open.spotify.com/search/results/Greta%20Van%20Fleet",
  "youtube_url": "https://www.youtube.com/",
  "ticketmaster_url": "https://www.ticketmaster.ie/greta-van-fleet-dublin-11-04-2019/event/1800554AECDC89F8"
}
```


Figure 4.25: Stored Event Object

As previously mentioned, these table entries were performed on sign up, this was accomplished by using Django's signal. Signals are like triggers in SQL they are fired when some other action is performed. Signals were implemented for the entries to the Profile, Spotify and Auth Token tables.

```python
# This is a trigger that is fired once a user is created,
# it creates a profile and spotify table entry for the new user
@receiver(post_save, sender=User)
def create_user_profiles(sender, instance, created, **kwargs):
    if created:
        Profile.objects.create(user=instance)
        Spotify.objects.create(user=instance)
```

Figure 4.26: Create Profile and Spotify Objects Signal

The App is responsible for rendering all the pages in the Web application with the exception of the recommended events list, as explained in the front-end design when a page is rendered it is passed a context variable which is a dictionary, the context variable that is passed is a list of events, whether this be the events the user has saved or the events returned from Ticketmaster. In order to ensure the page is never rendered to an unauthenticated user, the @login_required required decorator is used extensively throughout the backend.

```python
# Render home page
@login_required
def home(request):
    event_list = ticketmaster.search_ticketmaster(request)
    return render(request, 'app/home.html', {'event_list': event_list})
```

Figure 4.27: Render Home Page With event_list Context Variable

Furthermore, the App handles all form functionality such as user registration. The boilerplate registration form has been overridden in this project to require the user to provide their first and last name, this was implemented so the user's name could be displayed to them when they are logged into the system to provide a better user experience.

4.3.2. API Application

The api application (api app), handles all elements of the systems that involve API calls whether that be to external resources such as Ticketmaster or Spotify, internally to the recommendation system or the front end of the Web application. As such, this part of the project contains most of the business logic.

Before any business logic can be implemented involving adding data to the database via API e.g. Create or Update, data serializers are needed to get the received data into the correct format. As such, model serializers were created for the Spotify and Ticketmaster tables and registration via the mobile application. The serializers function by referencing the user via the instance and the table via the model. Once the data is into the required format the serializer automatically cleans it and the data is then saved to the database This is another feature of Django's ORM which in the background creates the corresponding SQL query required to perform the update or insert.

```
# Data serialization for the Spotify table
class SpotifySerializer(serializers.ModelSerializer):

    class Meta:
        model = Spotify
        fields = ('user_data', 'artist_count', 'recommended_artists')

    def update(self, instance, validated_data):
        instance.user_data = validated_data.get('user_data', instance.user_data)
        instance.artist_count = validated_data.get('artist_count', instance.artist_count)
        instance.recommended_artists = validated_data.get('recommended_artists', instance.recommended_artists)
        instance.save()
        return instance
```

Figure 4.28: Spotify Model Serializer

Firstly, after a user logs into the system and is authenticated but before they reach the Home page a signal is triggered which implements a chain of verification and data gathering operations. These begin with checking if the user has linked their Spotify account, if the user has not linked their Spotify account this process exits here and the message to link their account is rendered to the user. If the user has linked their account, this is confirmed by querying the social_auth_usersocialauth database table with the user id, upon confirming the user has linked their account the data retrieval process from Spotify begins.

```python
@receiver(user_logged_in)
@login_required
@permission_classes((permissions.IsAuthenticated, ))
@authentication_classes((authentication.TokenAuthentication, authentication.SessionAuthentication))
def get_spotify_playlist_artist_count(request, **kwargs):
    """
    Make a GET request to the spotify API for the users playlist IDs
    :param request: Incoming request search and city string
    :return: Result in Json
    """
    if request.user.social_auth.exists():
        user = request.user
        if user.is_authenticated:
            try:
                res = spotify.update_user_spotify_details(request, user)
                update_recommended_events(request)
                return res
            except Exception as e:
                print(e)
        else:
            print('Not a valid user')
            return Response(status=status.HTTP_401_UNAUTHORIZED)
    else:
        pass


def update_recommended_events(request):
    try:
        user_id = request.user.id
        user_ip = request.META.get('REMOTE_ADDR', None)
        ticketmaster.update_recommended_events(user_id, user_ip)
    except Exception as e:
        print('Event update failed...')
```

Figure 4.29: Standard View Layout and Asynchronous Recommender Task Initialization Method

All operations that involve the Spotify API are handled in the spotify.pf file. There are two entry points to this file, the first retrieves the currently authenticated users' personal information from Spotify, this includes their name, email, user id and profile picture.

```python
def get_user_details(user):
    """
    Make a GET request to the spotify API for the users information
    :param user: currently validated user
    :return: Result in Json
    """
    serializer = serializers.SpotifySerializer
    social = user.social_auth.get(provider='spotify')
    social.refresh_token(load_strategy())
    auth_header = {
        'Accept': 'application/json',
        'Content-Type': 'application/json',
        'Authorization': 'Bearer {}'.format(social.extra_data['access_token'])
    }
    try:
        res = requests.get(sp_base_url + 'v1/me', headers=auth_header)
        response = json.loads(res.content)
        serializer.update(serializer, user.spotify, {'user_data': response})
        return serializer

    except Exception as e:
        return Response({"detail": e}, status=status.HTTP_400_BAD_REQUEST)
```

Figure 4.30: Get User Details

In order to gain access to the users' personal data, their user id and access token need to be provided. The access token needs to be refreshed before every request, a Python library Spotipy is used here to simplify this process. Once the user details have been retrieved the method updates the database with their details using the serializer. If the user's profile picture is retrieved it is then set as the users' default profile picture.

```
def update_user_spotify_details(request, user):
    """
    Make a GET request to the spotify API for the users playlist information
    :param request: Incoming request
    :param user: currently validated user
    :return: Result in Json
    """
    serializer = serializers.SpotifySerializer
    social = user.social_auth.get(provider='spotify')
    social.refresh_token(load_strategy())
    token = social.extra_data['access_token']
    username = social.uid
    try:
        if token:
            sp = spotipy.Spotify(auth=token)
            playlists = sp.user_playlists(username)
            artist_list = build_artist_list(sp, playlists, username)

            # Create a unique dictionary of all artists and track counts
            artist_count = dict(Counter(artist_list))

            artist_count, favourite_artists = limit_artist_count_build_favourites(
                artist_count, user.spotify.recommended_artists)
            # Clean data retreive from spotify
            artist_count = remove_malformed_entries(artist_count)
            # Serializer used to update the Spotify table
            serializer.update(serializer, user.spotify,
                              {'artist_count': artist_count,
                               "recommended_artists": favourite_artists})
        else:
            print('No token found for user: {}'.format(user))
            views.get_users_spotify_details(request)

        return Response(artist_count, status=status.HTTP_200_OK)
    except Exception as e:
        return Response({"detail": e}, status=status.HTTP_400_BAD_REQUEST)
```

Figure 4.31: Retrieve Users Playlist Data from Spotify

The second entry point to this file retrieves the user playlist information. In this method the Spotipy library is used to make the call to the Spotify API for the users playlist information, the response is iterated over in the build_artists_list method to extract the artist of every song in the playlist into a list, Pythons High performance container datatype [31] Counter is used to create a dictionary of artist and occurrences which will be used later as the rating system in the recommendation model. After the artists rating data structure is created the rating scale is set from one to ten and any artists that have a rating greater than four are appended to a favourite artists list. Finally, the artists rating data structure is checked for any malformed

entries which are removed, and both the artists rating and favourite artist data is stored in the database.

```
def build_artist_list(sp, playlists, username):
    artist_list = []
    for playlist in playlists['items']:
        results = sp.user_playlist(username, playlist['id'], fields='tracks, next')
        tracks = results['tracks']
        tracks_items = tracks['items']

        if tracks['next'] is None:
            for track in tracks_items:
                artist_list.append(track['track']['artists'][0]['name'])

        else:
            while tracks['next']:
                tracks = sp.next(tracks)
                tracks_items.extend(tracks['items'])

            for track in tracks_items:
                artist_list.append(track['track']['artists'][0]['name'])
    return artist_list


def limit_artist_count_build_favourites(artist_count, favourite_artists):
    # Limit count to 10
    for k, v in artist_count.items():
        if v > 10:
            artist_count[k] = 10
        if v >= 4 and k not in favourite_artists:
            favourite_artists.append(k)
    return artist_count, favourite_artists

def remove_malformed_entries(artist_count):
    try:
        artist_count.pop('"')
    except KeyError:
        print('No quotation entries in this profile\n')
    try:
        artist_count.pop('')
    except KeyError:
        print('No blank entries in this profile\n')
    try:
        artist_count.pop(' ')
    except KeyError:
        print('No space entries, in this profile\n')
    return artist_count
```

Figure 4.32: Process the Retrieved Playlist Data

Following on, when all the relevant data is gathered from Spotify and stored in the database a background job is created and the Homepage is rendered to the user. When the Home page is being rendered it calls the search_ticketmaster method. This method checks to see if a search term has been provided, when the Home page is rendered for the first time it has not been passed a search term so it sets the classification parameter to “music” this parameter is used in the event classification request that is sent to Ticketmaster this is a specific endpoint in the Ticketmaster API which returns events based on their classification e.g. music, drama, theatre. If a search term has been provided which is the case when the search functionality of the Home page is used a different request is made to Ticketmaster which searches for events based on a

keyword. Something worth noting is the time.sleep(0.2) method call which pausing the method for 200ms each time it returns a page and the events are extracted, this is necessary because of the restrictions Ticketmaster put on the number of times their API can be accessed per second when using the free service.

```
def search_ticketmaster(request):

    try:
        classification_name = request.GET.get("search", "music")
    except:
        classification_name = "music"

    user_lat, user_long = request_latlong(request)

    return get_ticketmaster_events(user_lat, user_long, classification_name)


def get_ticketmaster_events(user_lat, user_long, classification_name):

    start_date_time, end_date_time = get_event_dates()
    try:
        if classification_name == "music":
            pages = load_events_classification(classification_name, start_date_time,
                                               end_date_time, user_lat, user_long)
        else:
            pages = search_events_keyword(classification_name, user_lat, user_long)

        single_page = []
        for page in pages:
            for event in page:
                single_page.append(event)
            time.sleep(0.2)

        event_list = event_list_builder(single_page)

        return event_list

    except Exception as e:
        print('Ticketmaster error: {}'.format(e))
        return {"error": "Couldnt connect to spotify api"}
```

Figure 4.33: Event Page Retrieval from Ticketmaster

In both cases the IP address of the user is extracted from the incoming HTTP request, this is used to ascertain the city that the current user is in and from that the latitude and longitude of the city are retrieved. This is accomplished using a geolocation library called GeoIP. When the latitude and longitude are retrieved the current date and time are retrieved and the request is made to Ticketmaster with the above information to find all events within a radius of 50km.

```python
def get_user_latlong(request):
    # Get the users location via IP in http request
    geo_ip = GeoIP2()
    ip = request.META.get('REMOTE_ADDR', None)

    try:
        if ip:
            geo_user = geo_ip.city(ip)
            user_lat = geo_user['latitude']
            user_long = geo_user['longitude']

        return user_lat, user_long
    except Exception as e:
        print('Ip could not be accessed, error: {}'.format(e))
```

Figure 4.34: Get User Latitude and Longitude

When the response is received from Ticketmaster, it is typically comprised of several pages, these pages need to be iterated through and all the events extracted. From each event specific details are extracted, some of which require pre-processing, they are then used to create an event object. Each event object is appended to a list which is then returned to the template rendering method where it is passed as a context variable to the home page and iterated over by the sub-rendered events.html template which contains the HTML structure explained in the app application.

```python
event = {
    "name": name,
    "image": image,
    "date": date,
    "time": item.local_start_time,
    "venue": {
        "name": item.json['_embedded']['venues'][0]['name'],
        "city": item.json['_embedded']['venues'][0]['city']['name'],
        "country": item.json['_embedded']['venues'][0]['country']['name'],
        "address": item.json['_embedded']['venues'][0]['address'],
        "longitude": item.json['_embedded']['venues'][0]['location']['longitude'],
        "latitude": item.json['_embedded']['venues'][0]['location']['latitude'],
        "venue_url": venue_url,
    },
    "youtube_url": yt_url,
    "ticketmaster_url": item.json['url'],
    "spotify_url": spotify_url
}
```

Figure 4.35: Event Object Created from Ticketmaster Event

The background job that was created prior to rendering the Home page is created using the Django library Background Tasks which stores tasks in the database with a time specified to wait before it is run. For this project, this task is run one second after creation and run asynchronously on a separate thread in the background of the application. This tasks firstly uses the favourite artist's entry in the database created earlier in the login process to search Ticketmaster's API for any events for those artists using the keyword search. After those events are returned, they are stored in the database and rendered to the user when they click the recommended button on the Home page, this needed to be done asynchronously as the user could potentially have hundreds of artists in this list.

Furthermore, after the recommended events are retrieved for the user based on their favourite artists the call is then made to the recommendation model which is passed the dictionary of artists and ratings stored in the database. The recommendation process will be described in the next section however, the response is a list of artists generated by the recommendation model. This list of recommended artists is used to search Ticketmaster for any events which are added to the favourite artists and recommended events columns in the database respectively.

Figure 4.36: External Links and Save Icons

When the user clicks the heart icon on the Home page, they add the selected event object to the database. This is accomplished by sending the event object from the webpage to the update_saved_events method, this method accepts the POST request containing the object, pulls in the contents of the saved events column in the database and checks to see if the object already exists if it does not it is added to the list of saved event object.

```
@api_view(['POST', ])
@login_required
@permission_classes((permissions.IsAuthenticated, ))
@authentication_classes((authentication.TokenAuthentication, authentication.SessionAuthentication))
def update_saved_events(request):

    if request.method == 'POST':
        try:
            user = request.user
            serializer = serializers.ProfileSerializer
            if user.is_authenticated:
                if user.profile.saved_events == "[]":
                    saved_events = json.loads(user.profile.saved_events)
                else:
                    saved_events = user.profile.saved_events

                new_event = json.loads(request.POST['save_event'])

                if not any(event['name'] == new_event['name']
                        and event['date'] == new_event['date'] for event in saved_events):

                    saved_events.append(new_event)
                    serializer.update(serializer, user.profile, {'saved_events': saved_events})
                    return Response(status=status.HTTP_200_OK)
                else:
                    pass

        except Exception as e:
            print(e)
            return Response({"detail": "Bad Request"}, status=status.HTTP_400_BAD_REQUEST)
```

Figure 4.37: Update Saved Events

Figure 4.38: Saved Event

When a user clicks on the trash icon from the profile page, they can delete an event object from their saved objects. This operation is performed in much the same way as the update. The event object is sent to the delete_saved_event method as a POST request the saved events database column is pulled in and the object is found and removed. The objects this time are removed based on their name and date because artists may have events on more than one possible date.

```
@api_view(['POST', ])
@login_required
@permission_classes((permissions.IsAuthenticated, ))
@authentication_classes((authentication.TokenAuthentication, authentication.SessionAuthentication))
def delete_saved_event(request):

    if request.method == 'POST':
        try:
            user = request.user
            serializer = serializers.ProfileSerializer
            if user.is_authenticated:
                saved_events = user.profile.saved_events
                delete_event = json.loads(request.POST['save_event'])
                saved_events[:] = [event for event in saved_events if not
                (event['name'] == delete_event['name'] and event['date'] == delete_event['date'])]

                serializer.update(serializer, user.profile, {'saved_events': saved_events})
                return Response(status=status.HTTP_200_OK)

        except Exception as e:
            print(e)
            return Response({"detail": "Bad Request"}, status=status.HTTP_400_BAD_REQUEST)
```

Figure 4.39: Delete Saved Event

4.4. Recommender System Development

The development of the recommender system consists of two distinct parts the data pre-processing class and the actual recommendation system, both processes will be explained. When the data is received from the Web application via POST request, the body, containing the email as the key and the dictionary of artist ratings as the value, is passed to the recommender system which then instantiates the pre-processing class with this data which extracts the key and value into instance variables, the key which is the email becomes the unique identifier for that user.

Once the instance has been created it is then used to call the process data which initializes several data cleaning and preparation operations. Firstly, the artists rating dictionary is examined and any artists the users have provided a rating of three or higher for is added to a favourite artists list, this is the same process that occurs when the user's playlist data is first gathered from Spotify. The decision was made to use a lower score requirement for the favourite artists in the recommender system than the Web application, and therefore this

process needed to be repeated, this was done in order to try and provide more diverse recommendations to the user.

```
def create_surprise_dataset(self, user_df):
    """Create Surprise Dataset from Pandas DataFrame"""
    dataset_path = os.path.join(settings.BASE_DIR, 'datasets/CleanedDataset.csv')
    df = pd.read_csv(dataset_path, sep=',', names=['user', 'artist', 'track_count'], encoding="utf8")

    # If the user is not in the dataset already add them
    if not user_df.user.values[0] in df.user.values:
        df = pd.concat([df, user_df])
        self.update_dataset(dataset_path, df)

    reader = Reader(rating_scale=(1, 10))

    return Dataset.load_from_df(df[['user', 'artist', 'track_count']], reader)
```

Figure 4.40: Create Surprise Dataset

Moreover, when the favourite artist's list has been generated the artist's rating dictionary is used to create a Pandas DataFrame. Firstly, this is used to check the #nowplaying dataset, which is also loaded into a separate DataFrame, if the users information is not contained in the dataset which is stored in a CSV file it is written to the file, this continually updates the users preferences and incorporates all users into the recommendation model continually increasing the size of the dataset and updating it with new artists. When the dataset has been updated the user and #nowplaying datasets then concatenated. This concatenated DataFrame is then used to create a Surprise Dataset which requires a Pandas DataFrame and a reader as parameters, the reader is used to inform the algorithm what the rating scale for the data is, one to ten for this a project. Finally, when the Surprise Dataset (dataset) is created, it and the favourite artists are returned to the recommendation system.

Upon receiving the dataset, the recommender system then begins the process of training and fitting the SVD algorithm. Various epoch values were tested for the training of the algorithm from one up to one hundred until it was determined that five epochs produced the best results. Epochs are the number of times the algorithm performs the Stochastic Gradient Descent operation.

```
def train_fit_model(dataset):
    """Builds and fits the model from the surprise dataset"""
    trainset = dataset.build_full_trainset()
    SVD_algo = SVD(n_epochs=5)
    SVD_algo.fit(trainset)
    return SVD_algo
```

Figure 4.41: Train and Fit SVD Algorithm

This operation selects an entry in the dataset at random and iterates over several weights which are representative of each feature in the entry until it finds the weights that represent the entry the best i.e. that reduces the RMSE error function the most, this iteratively increases

the accuracy of the algorithm on each pass. Too many passes and the model will begin to overfit to the training data, not enough passes and the model will underfit the training data. After the SVD algorithm has minimized the error function sufficiently it then generates the eigenvectors that represent each user and artist and their relationship. These are represented in the submatrices explained in the Literature Review chapter which can be can then be used to reproduce the original matrix (dataset) via the dot product.

```
def get_top_similarities(artist, model):
    """Builds a similarity table of one artists to all other artists
        using the cosine distance of each artists vector"""
    artist_vector = get_vector_by_artist(artist, model)
    similarity_table = []

    # Iterate over every possible artist and calculate similarity
    for other_artist in model.trainset._raw2inner_id_items.keys():
        other_artist_vector = get_vector_by_artist(other_artist, model)

        # Get the second artists vector, and calculate distance
        similarity_score = cosine_distance(other_artist_vector, artist_vector)
        similarity_table.append((similarity_score, other_artist))

    return build_top_n_dataframe(sorted(similarity_table))


def get_vector_by_artist(artist, trained_model):
    """Returns the latent features of an artist in the form of a numpy array"""
    artist_row_idx = trained_model.trainset._raw2inner_id_items[artist]
    return trained_model.qi[artist_row_idx]


def cosine_distance(vector_a, vector_b):
    """Returns a float indicating the similarity between two vectors"""
    return cosine(vector_a, vector_b)
```

Figure 4.42: Generate Recommendations

Furthermore, when the model is trained and fit and the eigenvectors have been generated for each artist, the list of the user's favourite artists is iterated over, and each artist's eigenvector is extracted from the model. The eigenvectors of all other artists in the dataset are then iterated over and the cosine distance from the current favourite artist's vector is computed, the ten artists with the lowest cosine distance are then used to create a Pandas DataFrame. This process is repeated for every artist and all the generated DataFrame's are concatenated together and then sorted in ascending order based on their cosine distance. The first two hundred entries are then extracted as they have the lowest overall cosine distances of all the artists, and these are returned to the Web application as the recommendations.

4.5. Deployment and Hosting

Deployment of the Web application and the recommender system is accomplished entirely using Docker. As explained in the Design chapter a Dockerfile is used to containerize the applications. The container was specified to be a base Python 3.6 container which was updated

and had all the package dependencies of the specific portion of the project installed via the requirements.txt file. Moreover, the application was then copied into the container and port 8000 of the container is exposed so it can be mapped to a port on its host machine. The CMD function is the entry point for the container and runs the server mapping it to the IP of the current machine.

```
FROM python:3.6

MAINTAINER James Ward

RUN apt-get -y update
RUN apt-get -y upgrade
RUN mkdir -p /usr/src/app

COPY requirements.txt /usr/src/app
COPY . /usr/src/app

WORKDIR /usr/src/app

RUN pip install --upgrade pip
RUN pip install --no-cache-dir -r requirements.txt
RUN pip install scikit-surprise
EXPOSE 8000
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Figure 4.43: Dockerfile

Furthermore, EC2 instances were created in AWS and Docker installed on the system. The respective portions of the project were then pulled down onto these instances and run. When these instances were up and running the domain of the website www.localgigs.org was purchased and a DNS server was set up using AWS Route53. This mapped the domain name to the EC2 instance. After which an SSL certificate was also attained for the domain, and applied by setting up a load balancer, so all content is provided via https.

4.6. Conclusions

This chapter discussed the development process involved in this project, it started by outlining the front-end development, which is broken into two sections the Web and mobile applications. For the Web application, the development of the functionality and appearance of each page was detailed. For the mobile application, the accessing of data and services from the back-end and the development of the screens was explained. Proceeding on the development of the back-end was detailed which was also separated into two sections, the back-end functionality behind the rendering of the front-end and the all the connectivity to internal and external API's which were handled in separate sub-applications. Moreover, the development of the recommender system was examined in which the pre-processing of the data from the Web application, the training and fitting of the model and the generation of predictions was investigated. Finally, the architecture of the deployment mechanisms was detailed in which the containerization and hosting of the applications and the configuring of the DNS servers were explored.

5. Testing and Evaluation

5.1. Introduction

This chapter discusses the testing of the entire systems functionality and using approaches such as ad-hoc and unit testing to ensure code reliability. User testing has also been conducted to attain some usability feedback on the performance and operation of the system, this testing was completed using a survey. This chapter will not cover the testing of the machine learning algorithms as this was completed in the Design and Architecture chapter.

5.2. Ad-Hoc Testing

The most common type of testing in any software development project is ad-hoc testing. This involves testing every feature that has been developed in every possible manner it can be used to ensure the code is working as intended without any errors or serious warnings. This type of testing can be time-consuming especially if there are many features as each one must be manually tested, and the logs examined to ensure that no errors have occurred. The benefit of this type of testing is that it only requires knowledge of how the system functions, therefore this is perfect for this project.

Ad-hoc testing was performed by visiting every Web page in the Web app and testing all features and examining the logs both in the development server locally and after the application was deployed by attaching the container performing the same tests again and monitoring the logs from there.

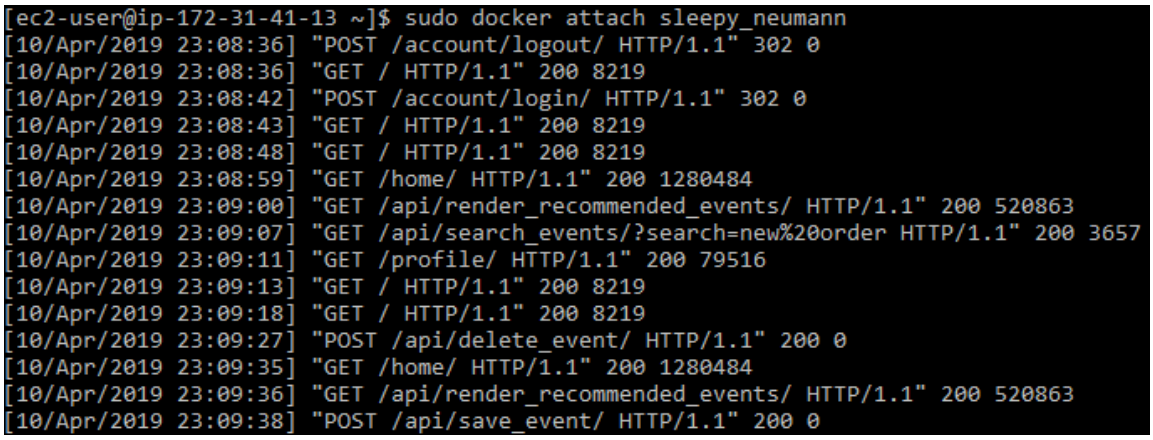


Figure 5.1: Excerpt of Testing Logs in an Attached Container

A few examples of the features that were tested were the creation of a new user ensuring they were added to the database correctly and the additional Profile and Spotify table entries had been made for the user. Also, all the API call functionality such as retrieving the user's details from Spotify ensuring that data is added to the database and that any user

information such as name and profile picture are being rendered in the correct shape and format to the correct page. On the front end ensuring that the Ticketmaster API can be searched by using the search bar that the recommended events and solely location based are being rendered and that the user can save events to their profile delete them and use the external links to purchase tickets to the events on Ticketmaster, listen to the artist on Spotify and watch their music videos on YouTube. Postman was used in the testing of all the endpoints in the Web application and the recommendation model was tested via the Django REST API template. This provides a generic template view where API requests can be made to the endpoint.

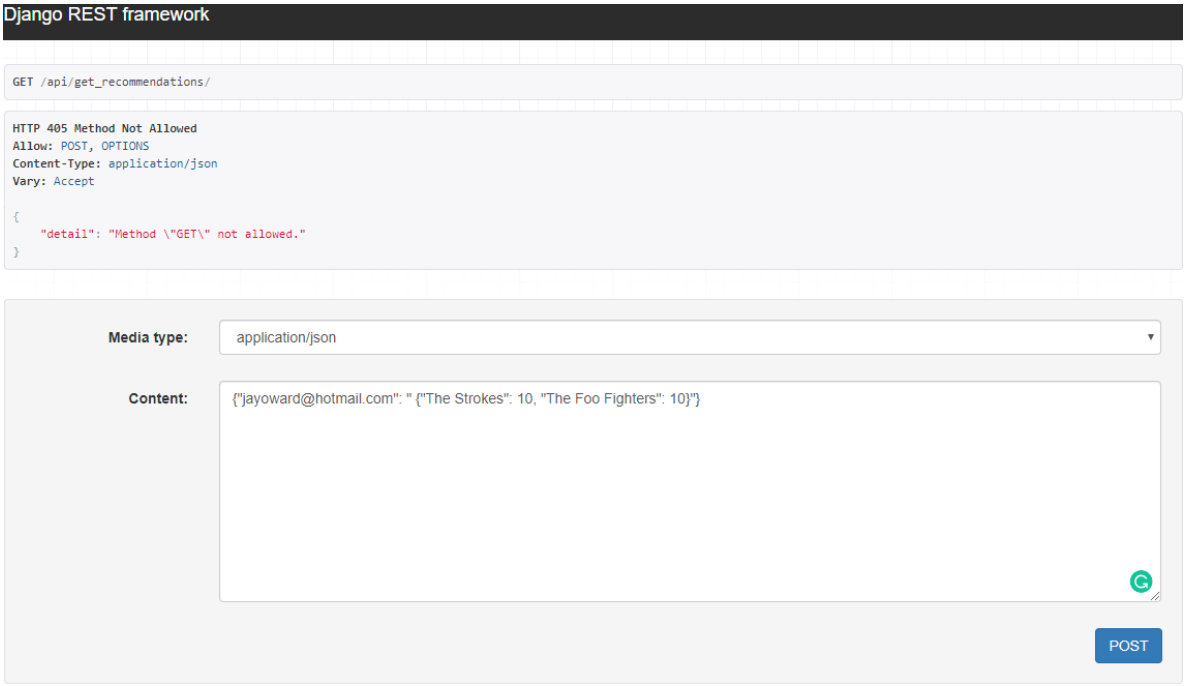


Figure 5.2: Testing Recommendation Endpoint

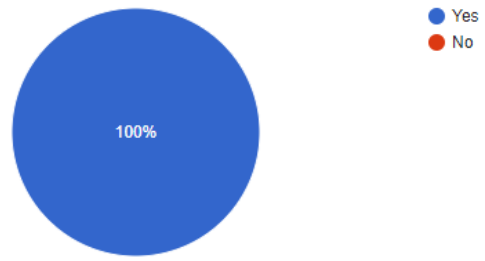
5.3. User Evaluation

This system has a heavy focus on the user and tailoring a custom experience to each user. Due to this user-centric approach to the application, it was thought to be important to get some feedback from users. A group of people were asked to use the application and fill out a simple google docs survey. The responses were all very positive with the users finding the user interface intuitive, most of the users liked the design. There was one user who was initially confused by the heart as the save button who communicated this via text message and there was a general consensus that the recommendations could be quicker, but that limitation is down to the hardware the system is running on and the limits Ticketmaster put on their API's. There were a few comments about links to YouTube not working and the routing service failing. These are known issues as not all artists have YouTube channels and so the systems defaults to youtube.com if they do not have one and the routing server is a free demo server so one hundred percent reliability cannot be expected.

5.3.1 Google Docs Survey Results

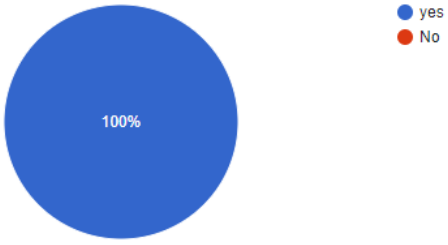
Is the colour scheme appealing?

7 responses



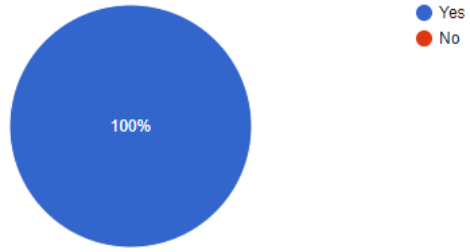
Is navigating the website intuitive?

7 responses



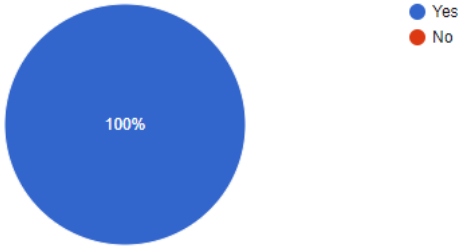
Are the events specific to your location?

7 responses



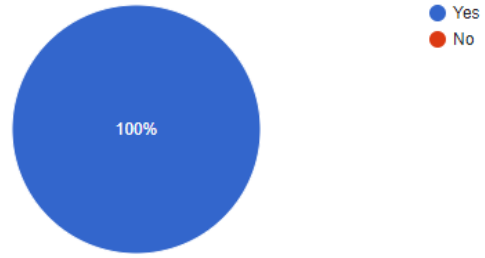
Are you able to save events?

7 responses



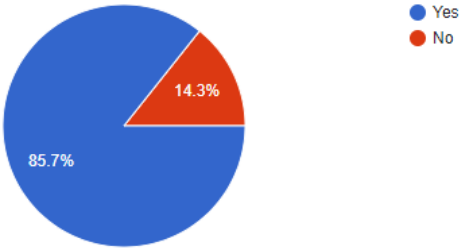
Are you able to delete events?

7 responses



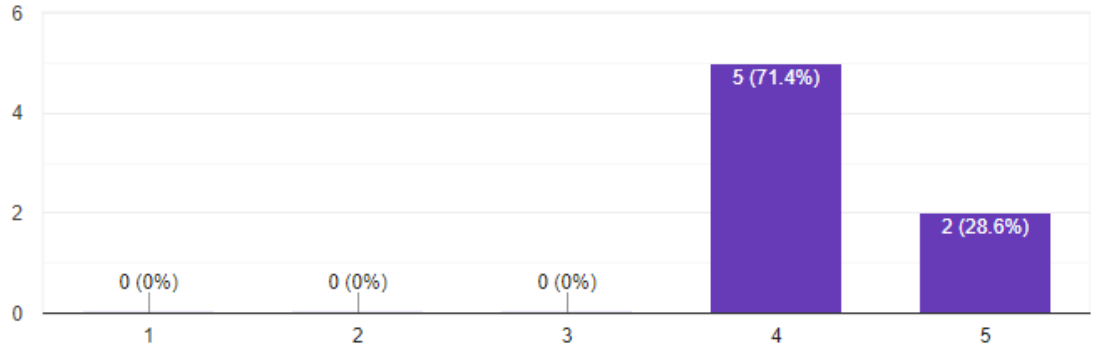
Are you able to get routing to events?

7 responses



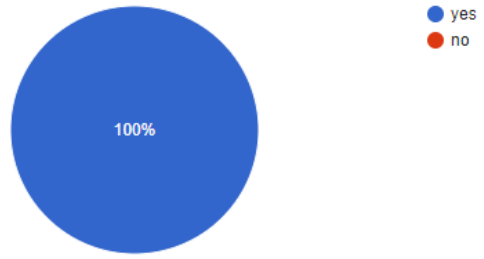
Are the recommendations accurate?

7 responses



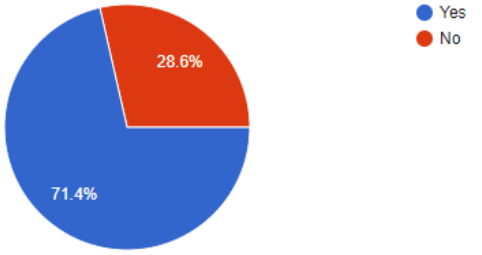
Are the links to buy tickets working?

7 responses



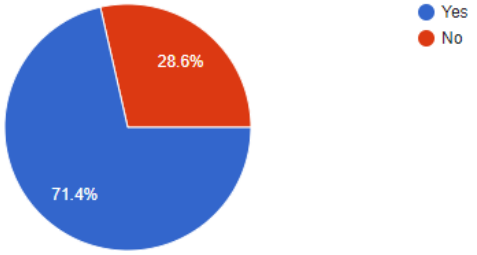
Are the links to Spotify working?

7 responses



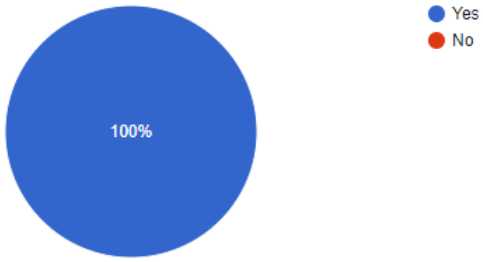
Are the links to Youtube working?

7 responses



Can you search for events?

7 responses



What would you like to see added?

7 responses

Faster recommendations
Mobile compatability
able to use on mobile
there was no routing
some way of saving tickets for gigs
recommended to load faster

Most of the users found the actual event recommendations to be accurate and they have conveyed they would use this application again. Two of the users said they would like to see a mobile compatible version, which was the intended purpose of the mobile application, one other interesting idea was a section to manage tickets purchased for events, this is a good potential idea for this project in the future.

5.4. Unit Testing

Unit testing is a method of testing code where individual units or blocks of code are tested. This is typically accomplished by performing some action on the method and expecting it to return a specific response. Unit testing is done so that each unit of code or functionality can be tested to ensure it is operating correctly, they are the smallest testable part of any software. [32] Unit testing becomes very useful when changes are being made to the existing code or new features are being added, the unit tests ensure that the functionality that is already in place is not broken in any way by the changes.

Unit testing was implemented using Django's TestCase and SimpleTest case classes for much of the back end of the application. The construction of unit tests was not high on the priority list and so took a back seat in this project. However, approximately forty percent of the overall project has unit testing implemented. The unit tests ensure that all the endpoints were resolving to the correct view functions and that the view functions were redirecting to the correct page, rendering the correct template and returning the correct response.

The URL tests were accomplished by providing the test with a URL name to reverse and then asserting that the reversed URL had been successfully resolved to the correct view function. The view tests performed either a GET or POST request to a reversed URL depending on what the view method was expecting. Several responses could be expected depending on the method but in general it was either a status code of two hundred and the template that would be rendered both of which would be asserted or it was a status code of three hundred and two (redirect) and the redirect URL was expected to be to the desired page after it had passed through authentication confirmation.

```python
class TestViews(TestCase):

    def test_landing_view(self):
        response = self.client.get(reverse('app:landing'))
        self.assertEquals(response.status_code, 200)
        self.assertTemplateUsed(response, 'app/landing.html')

    def test_home_view(self):
        response = self.client.get(reverse('app:home'))
        self.assertRedirects(
            response,
            '/accounts/login/?next=/home/',
            status_code=302,
            fetch_redirect_response=False)

    def test_profile_view(self):
        response = self.client.get(reverse('app:profile'))
        self.assertRedirects(
            response,
            '/accounts/login/?next=/profile/',
            status_code=302,
            fetch_redirect_response=False)

    def test_update_profile_view(self):
        response = self.client.get(reverse('app:update_profile'))
        self.assertRedirects(
            response,
            '/accounts/login/?next=/update_profile_details/',
            status_code=302,
            fetch_redirect_response=False)

    def test_update_profile_pic_view_no_POST(self):
        response = self.client.get(reverse('app:update_profile_pic'))
        self.assertRedirects(
            response,
            '/accounts/login/?next=/profile/update_profile_pic/',
            status_code=302,
            fetch_redirect_response=False)
```

Figure 5.3: Unit Tests Class for App views.py

5.5. Conclusions

This chapter reviewed the testing and evaluation of the system. The testing included several approaches. Ad-hoc testing was implemented to test the functionality of all features of the website and ensure that the code was not failing, and no serious errors or warnings were occurring. We then looked at user experience testing in which a survey of the small test group was examined and their overall rating of the system was detailed. Finally, we investigated the unit tests that were constructed to ensure that any updates that were being made to the system's code were not breaking any existing features or functionality.

6. Conclusions and Future Work

6.1. Introduction

In this chapter, the key lessons learned from this project will be discussed. First, a discussion of the initial project proposal and how the final product diverged from the initial proposal. Following this, the conclusions for the different aspect of the project including a personal conclusion. Finally, a description of the future work planned for the project.

6.2. Project Plan

6.2.1. Initial Proposal

Throughout the development process, this project has undergone many changes from the initial proposal, while still staying on track to the overall objective. The overall objective was to build a web application that would recommend live music events to users based on the artists they listen to on Spotify, and to that extent, the project stayed true to the initial plan and was a success. An excerpt from the initial proposal can be seen below.

"The objective of this project is to enable people to find upcoming live music events that are taking place in their geographical location. The person will be able to get directions to and book tickets for these live music events. The application will make recommendations to the person based on their musical interests and the musical interests of other persons with similar taste. The person's musical taste will be determined by the artists they have in their Spotify playlists."

The initial plan involved the construction of a fully-fledged Web application followed by the construction of a mobile application when that was completed. This was to allow users to make use of the recommendation system and mapping features on the go, in their home city or overseas where they may not be as familiar with the locale. While the Web application was fully completed there was not enough time to complete the mobile application. With that in mind, the core deliverables were achieved and not much development is required to finish the mobile application.

6.2.2. Changes from Proposal

During the development lifecycle of this project, there were several notable changes from the initial proposal. Most of the changes are to frameworks or approaches to the problems. The proposal detailed a requirement of constructing the front-end of the application using the Angular framework and while this was used in the construction of the mobile application this

was intended for the front-end end of the Web application also. It was decided during the development process that some of the tasks such as rendering the large lists of HTML elements and searching Ticketmaster's API from the Home page would work more efficiently if they were performed server-side with as few API calls in between as possible, therefore it was prudent to utilize Django's templating structure which proved to be just as competent as Angular for the requirements of this project.

Furthermore, another notable change was the approach to the recommendation system. The system was initially being designed as a user-based collaborative filtering approach where the artist's recommendations would be generated based on the similarity of one user to another and their rating patterns. The recommendation model had to be changed due to constraints on the Surprise library. The approach was changed to an item-based collaborative filtering system where the artist's recommendations are generated based on the artist's similarity to other artists based on the ratings provided by the users. Both approaches were tested, they provided very similar recommendation results as they implement the same algorithm and method of training and fitting the algorithm, the method of prediction is the only part that changed in the recommendation system.

6.3. Conclusions

6.3.1. Literature Review

6.3.1.1. Recommendation Systems

After substantial research into recommendation systems, including what types of recommendation systems work best for what problems, what methods are implemented by a variety of companies and different possible ways to implement the same type of system a great deal was learned about recommendation systems and their functionality. I believe the correct decision was made for the approach to the system and the algorithm. The item-based collaborative filtering recommendation system is very common in the movie and music recommendation industry utilized by companies such as Netflix and Spotify, and the Singular Value Decomposition algorithm has proven to be excellent for collaborative filtering systems. This was demonstrated by Bellkor, the winners of the Netflix prize who improved a multimillion-dollar companies movie recommendation system dramatically, reducing their overall prediction error by ten percent through implementing this algorithm, among other techniques.

6.3.1.2. Technologies

The initial research into what technology stack was going to be used on this project drastically changed from the project's initial conception. Almost all the programming languages and frameworks that were planned to be used changed. This was beneficial to me as a student and

a developer as it compelled me to learn new languages and frameworks and explore avenues of development I had not encountered or considered before. Such as, implementing Python to construct the Web application, which was always intended to be a Java Spring Boot application. Ionic for the Android application which was always intended to be developed using Android Studio.

6.3.1.3. Existing Solutions

At the beginning of this project, it was believed there were no companies providing a service even remotely similar to this project. However, Bandsintown was discovered during the development process of this project. Bandsintown does not offer any recommended events to its users but it does provide the user with the events on in their location and a map showing where the event is, finding this was somewhat disheartening as I believed the idea was entirely original. However, some inspiration was taken from this website, such as its user interface. It is well designed and structured, and a similar implementation was applied in this project.

6.3.2. Design and Development

6.3.2.1. Methodology

The agile methodology had become familiar over the course of the third-year internship and so, some implementation of it was decided upon for the project. ScrumBan was created for this project. The implementation of the Kanban board to document the meeting minutes with the project supervisor and all tasks that needed to be completed in the project was invaluable. With such a large workload it was easy to forget topics discussed in the weekly meetings and to forget about ideas for new features. This was all kept track of on the LocalGigs Kanban board on www.trello.com. The scrum aspect of the methodology was also useful as every week the progress of the project was demonstrated to the supervisor and feedback was received.

6.3.2.2. Front-End

The design of the front-end of the Web and mobile applications changed substantially over the course of the project. Initial design concepts such as drag and drop features with automatic dataset updates were dropped and a clean uncluttered design was implemented. As one of the test users mentioned it has a social media appearance which is attributable to the organization and colour scheme used throughout. Small nuances of the front-end design achieved through following the design principles helped to make the appearance visual striking yet not overbearing.

The decision to move the development of the front-end to Django was a good idea for this project. It allowed for the easy use of some of the great features of the framework such as

confirming user authentication and direct access to database contents in Web pages allowing the rendering of user's information securely and with great accessibility, removing the need for performing unnecessary API calls to the back-end. This and Django's templating structure really helped trim sown a lot of the fluff typically associated with front-end development.

6.3.2.3. Back-End

The back-end of the application was designed based on its functionality. The functionality was separated out into separate Django applications each one responsible for a portion of the project. The code is designed in a modular fashion all aspect of template rendering and database design handled in one application and the other dealing with all internal and external API calls. This approach worked in modularising the code, it helped increase the separation of concerns of the project and makes navigating the file structure intuitive. This could have been taken one step further and separate applications implemented for all external API's, however, there is always the possibility of bloating the application with that approach.

The development of the back-end involved constructing most of the functionality of the system. Overall, I am happy with the outcome, there are a few issues that I would like to sort out such as the asynchronous tasks which run in the background and talks to the recommendation system this uses a Django library to store tasks for later execution and it is not one hundred percent reliable, the process that runs the tasks can fail. I believe all the required functionality of the system was implemented. A huge amount of information was learned about many areas such as user authentication, database management, dealing with external resources, working within their limitations and finding methods to exploit them to their fullest. Setting up and configuring a domain using cloud-based services to host and deploy applications and much more.

6.3.2.4. Recommendation System

The design of the recommendation system took the correct approach from the start thanks to the extensive research undertaken into recommendation systems. There was an emphasis put on finding the data first, this set the recommendation model up for success as its development was able to follow a tried and tested methodology CRISP-DM. The data was the focus when deciding what type of recommendation system to use. The item-based collaborative filtering system works well and produces accurate recommendations. There were some roadblocks along the way which led to many avenues being explored including the AWS service Sagemaker to try and deploy the recommendation model as and AWS provided endpoint however this was quickly decided against and the initial system was implemented.

The development of the recommendation system took a long time. I learned a lot about how to effectively process large volumes of data, how to analyse them for patterns and how to exploit those patterns using the correct algorithms and techniques. Much of the development process was accomplished using Pandas and Surprise, two fantastic libraries I would highly recommend. The recommendations generated by this project are nowhere near on the scale of Spotify and so are never going to be as diverse however from user feedback the systems seem to be accurate in what it recommends.

6.3.3. Personal Conclusion

The idea behind this project was spawned of my own desire to reduce the number of live music events of artists that I like that I missed. I had never worked with many of the technologies or concepts implemented throughout this project such as data science, machine learning, predictive analytics, recommendation systems, Python, Django and much more. It was very enjoyable to be able to learn about all these concepts while working towards solving a problem that occurs in my day to day life. The research undertaken for this project was very interesting and I learned a lot about how companies such as Spotify, YouTube and Netflix generate the recommended songs or videos to their users.

The end goal for any recommendation system is to recommend something new to a user or something they need or want. In the realm of music, this used to be accomplished through friends. Artists were spread by word of mouth from person to person which requires an inherent understanding of the person, their tastes and the domain. The question for a long time has been 'can a machine really be as smart and intuitive as a human?' The answer in this use case is leaning towards yes. The computer can analyse so many different artists and the interests of so many users that is able to attain a very good understanding of a person's musical taste, or any other tastes for that matter, through similarity comparisons. This is evident in the success of companies such as Spotify and Netflix who attribute more than seventy-five percent of the content viewed on their platform to have been recommended to the user by a recommendation system. In other words, the content users are watching or listening to and therefore their behaviours are becoming more and more influenced by the intelligence and intuitiveness of machines.

As for my own personal sentiment. I am happy with the outcome of the project. I believe it was a complex project with several elements that had to come together in the correct manner for it all to work and this was achieved. It was a fantastic learning curve; it was the first time I had worked on a large project from start to finish having been involved in all aspects. I learned a great deal about time and project management and meeting hard deadlines. I learned how to effectively communicate the technologies and development that went into the project. I started the project with no knowledge of data science machine learning, predictive analytics,

recommendation systems, Python, deployment and much more and have developed several invaluable skills along the way.

6.2. Future Work

The first area where I plan to continue the development of this project is the mobile application. I plan to finish the development of the mobile application as I intend to use it myself in the future and several users have also expressed an interest in a mobile application. I do plan to keep the website live on the free tier of AWS for the foreseeable future.

The recommendation system now is utilizing a CSV file as its data storage and it does not perform any user authentication. In the future, I would like to implement another database for the recommendation system and user authentication and verification.

A long-term goal of my own is to continually improve my skills in the domain of machine learning and predictive analytics. I believe a good way to do this and improve the functionality of this project would be to implement the SVD algorithm myself. The Surprise library is fantastic but certain features such as the learning rate cannot be controlled without modifying the source code.

One of the major elements that have a noticeable effect on performance in this project is the requests made to the Ticketmaster API. I want to contact Ticketmaster and see if these restrictions can be made less severe, so performance is not affected.

I would like to implement a different method for initializing the request to the recommendation system. At the moment a Django library is being used and it is not one hundred percent reliable. I would like to implement a solution that is entirely reliable and functions in a manner like Angular's request and promise mechanism which can keep a method running in the background asynchronously until the response is returned.

A feature that was suggested by one of the users will also be implemented. The ability to store tickets that have been purchased from Ticketmaster, this could possibly be implemented as storing a QR or code or a link to one and would be a great addition to the project.

References

[1] K. Shubber, “Music analytics is helping the music industry see into the future,” *The Guardian*, 09-Apr-2014.

[2] A. Asokan, “Solving Information Overload With Recommendation Engine,” *Analytics India Magazine*, 26-Mar-2019. .

[3] E. Zangerle, M. Pichl, W. Gassler, and G. Specht, “#Nowplaying Music Dataset: Extracting Listening Behavior from Twitter,” in *Proceedings of the First International Workshop on Internet-Scale Multimedia Management*, New York, NY, USA, 2014, pp. 21–26.

[4] “Basic Approaches in Recommendation Systems,” *ResearchGate*. [Online]. Available: https://www.researchgate.net/publication/256458336_Basic_Approaches_in_Recommendation_Systems. [Accessed: 02-Apr-2019].

[5] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, “Incremental Singular Value Decomposition Algorithms for Highly Scalable Recommender Systems,” in *Fifth International Conference on Computer and Information Science*, 2002, pp. 27–28.

[6] S. Li, “Building and Testing Recommender Systems With Surprise, Step-By-Step,” *Towards Data Science*, 26-Dec-2018. [Online]. Available: https://towardsdatascience.com/building-and-testing-recommender-systems-with-surprise-step-by-step-d4ba702ef80b. [Accessed: 03-Apr-2019].

[7] R. Jin, “A Study of Mixture Models for Collaborative Filtering,” p. 33.

[8] J. D. Kellher, B. Mac Namee, and A. D’Arcy, *Fundamentals of Machine Learning for Predictive Analytics*. The MIT Press, Cambridge Massachusetts, London, England.

[9] “Linear Algebra and Matrix Decompositions — Computational Statistics in Python 0.1 documentation.” [Online]. Available: https://people.duke.edu/~ccc14/sta-663/LinearAlgebraMatrixDecompWithSolutions.html. [Accessed: 12-Apr-2019].

[10] “Singular Value Decomposition (SVD).” [Online]. Available: https://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm. [Accessed: 01-Dec-2018].

[11] “3.7.3 Documentation.” [Online]. Available: https://docs.python.org/3/. [Accessed: 12-Apr-2019].

[12] “What is Java?” [Online]. Available: https://www.java.com/en/download/whatis_java.jsp. [Accessed: 12-Apr-2019].

[13] “R: What is R?” [Online]. Available: https://www.r-project.org/about.html. [Accessed: 12-Apr-2019].

[14] “Spring Projects.” [Online]. Available: https://spring.io/projects/spring-boot. [Accessed: 12-Apr-2019].

[15] “The Web framework for perfectionists with deadlines | Django.” [Online]. Available: https://www.djangoproject.com/. [Accessed: 12-Apr-2019].

[16] “HTML: Hypertext Markup Language,” *MDN Web Docs*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML. [Accessed: 04-Apr-2019].

[17] “CSS: Cascading Style Sheets,” *MDN Web Docs*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/CSS. [Accessed: 04-Apr-2019].

[18] “JavaScript,” *MDN Web Docs*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript. [Accessed: 04-Apr-2019].

[19] “Angular.” [Online]. Available: https://angular.io/. [Accessed: 12-Apr-2019].

[20] Ionic, “Ionic Documentation.” [Online]. Available: https://ionicframework.com/docs/. [Accessed: 12-Apr-2019].

[21] “PostgreSQL: The world’s most advanced open source database.” [Online]. Available: https://www.postgresql.org/. [Accessed: 06-Nov-2018].

[22] “Python Data Analysis Library — pandas: Python Data Analysis Library.” [Online]. Available: https://pandas.pydata.org/. [Accessed: 12-Apr-2019].

[23] “NumPy — NumPy.” [Online]. Available: http://www.numpy.org/. [Accessed: 04-Apr-2019].

[24] “scikit-learn: machine learning in Python — scikit-learn 0.20.3 documentation.” [Online]. Available: https://scikit-learn.org/stable/. [Accessed: 12-Apr-2019].

[25] N. Hug, “Home,” *Surprise*. [Online]. Available: http://surpriselib.com/. [Accessed: 12-Apr-2019].

[26] “Home - Django REST framework.” [Online]. Available: https://www.django-rest-framework.org/. [Accessed: 04-Apr-2019].

[27] “AWS Documentation.” [Online]. Available: https://docs.aws.amazon.com/index.html?nc2=h_ql_doc. [Accessed: 12-Apr-2019].

[28] “What is Docker? | Opensource.com.” [Online]. Available: https://opensource.com/resources/what-docker. [Accessed: 04-Apr-2019].

[29] “What’s the Difference? Agile vs Scrum vs Waterfall vs Kanban,” *Smartsheet*, 27-Apr-2016. [Online]. Available: https://www.smartsheet.com/agile-vs-scrum-vs-waterfall-vs-kanban. [Accessed: 24-Nov-2018].

[30] “What is the CRISP-DM methodology?,” *Smart Vision - Europe*. .

[31] “8.3. collections — High-performance container datatypes — Python 2.7.16 documentation.” [Online]. Available: https://docs.python.org/2/library/collections.html. [Accessed: 08-Apr-2019].

[32] “Unit Testing,” *Software Testing Fundamentals*, 04-Jan-2011. .

Appendix

Project URL: www.localgigs.org

Docker commands for building and deploying the applications

Docker build -t gig0me/local_gigs .
Docker push gig0me/local_gigs

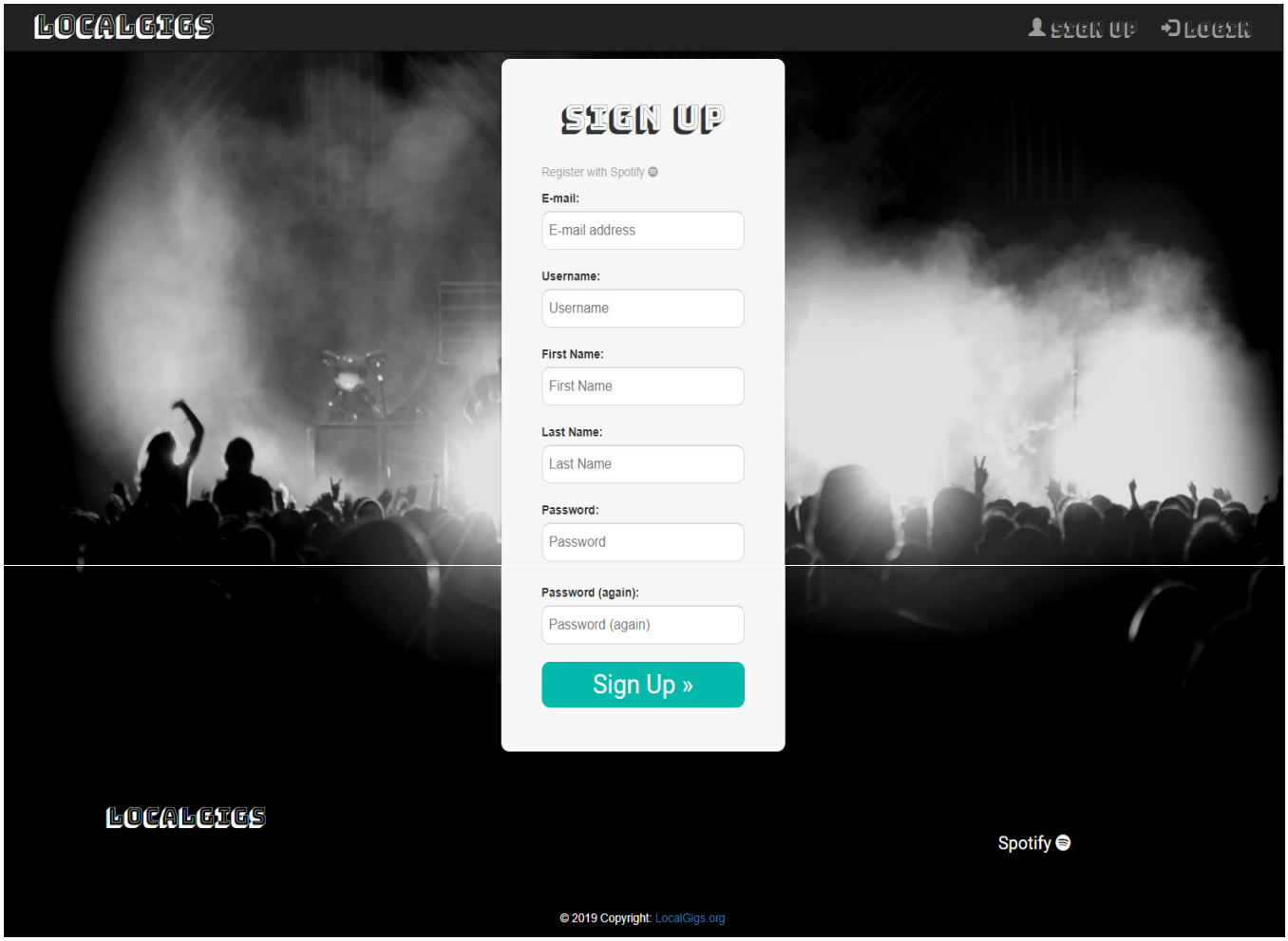
Docker pull gig0me/local_gigs
Docker run -p 80:8000 gig0me/local_gigs

Docker build -t gig0me/recommender .
Docker push gig0me/ recommender

Docker pull gig0me/ recommender
Docker run -p 80:8000 gig0me/ recommender

Docker attach <container>
Docker top <container>
Docker exec -it <container> /bin/bash

Sign up Page


LOCALGIGS
SIGN UP
LOGIN
SIGN UP
Register with Spotify
E-mail:
E-mail address
Username:
Username
First Name:
First Name
Last Name:
Last Name
Password:
Password
Password (again):
Password (again)
Sign Up »
LOCALGIGS
Spotify
© 2019 Copyright: LocalGigs.org

An early draft of the Profile page design