

Weather Music App (Cloudtunes)

Group Number: 8

Team Members: Julie Luu, Heather Cartwright, Alexandra Simon-Lewis, Johanna Doherty and Kara Howard

Introduction

The aim of our project was to create and build a Flask app, which would allow users to listen to music returned to them based on the weather in their current location or to assign their own playlists to certain weather conditions. Our goal is to add a little more excitement to the process of checking the weather during the day: whether the user is having a bad day and needs to be cheered up, or wants to wallow in a rainy atmosphere, our app will return music that is hopefully new to the user and improves their day somehow.

This project report is laid out as follows:

Background

In this section, we discuss the thought that went into our project - what was it that we decided to do, and what problem we wanted to solve by creating it.

Specifications and Design

We lay out the requirements for running our app, including packages that need to be installed prior to execution, and command-line code that should be run in order to execute the code. Additionally, we will demonstrate our system architecture design and a wireframe prototype of our front-end design.

Implementation and Execution

We discuss our development approach, going into detail about the implementation stage of our development, and discussing additional tools we used to implement it (frameworks, libraries, and so forth).

Testing and Evaluation

We will detail actions taken to carry out functional and user testing, including which methods and approaches we decided to use that would best serve the needs of our project.

Conclusion

We will consider the success of our project and explore whether we met our goals for this project. We will reflect on team performance and review what changes we can make to improve the success of our next development cycle.

Background

The basic functionality of our weather music app, Cloudtunes, is focused around a simple process flow: the app ascertains the user's latitude and longitude using a geolocation API and using the returned data, makes a second call to a weather API. Finally, the app makes a call to a music API to return data relating to tracks, with the search query being the returned weather result. We knew we wanted to implement this in the context of a web app specifically.

To add a more dynamic feature to our app, we decided to give the user an option to select their own playlists for use within the app - for example, assigning a pre-constructed Spotify playlist associated with their user account to a particular weather condition. We also elected to include an account creation option, which would allow the user to save their data within the app. In a future iteration of the app, this would include their email address, which would allow the user to opt in to receiving a daily weather overview with playlist, as well as saving key user locations - e.g. somewhere they are planning to go on holiday, or their work location if weather is likely to be different from home. This feature would also allow us to deal with the problem that users using VPNs will not be able to accurately use the geolocation feature, an issue we considered early in the design process.

Specifications and Design

Technical Requirements

The first decision we made was which APIs we would use. As the backbone of our app's functionality, we needed to ensure that there were existing APIs that would fulfil our requirements. After conducting some research, we chose the following three APIs, due to the fact that they were free and openly available to use:

- 1) Geolocation API: Geo API (<https://getgeoapi.com/>)
- 2) Weather API: Open Weather API (<https://openweathermap.org/api>)
- 3) Music API: Spotify API (<https://developer.spotify.com/documentation/web-api/>)

The next decision we needed to make was with regards to coding our app: as we were being assessed in Python, our primary programming language had already been selected for us, but there are multiple Python web frameworks available (e.g. Django). As we had already spent some time studying Flask, this seemed the most sensible option, especially as we had a limited time to implement the project and learning an entirely new framework would take away from potentially vital implementation time.

Although we were aware that a front end was not strictly necessary and we could have opted to mock input, we decided to try to create one in any case - we felt it added flavour to our project, and since the whole aim of our project was influenced by the concept of atmosphere and mood, our project would have seemed lacking without it. It also gave us the option to use HTML, CSS, and Jinja (Flask's inbuilt templating engine which is written to have Python-like syntax). This not only allowed us to learn new skills in different areas of programming, but allowed us to see how a dynamic and highly visual web page interacts with back-end code when context is inserted.

We knew we would also need a database to enable user account creation. While we had originally assumed that the easiest option to use would be MySQL, as this is what we had studied, we discovered that the creators of Flask have a module, flask_sqlalchemy, which provides SQLAlchemy support for the app. SQLAlchemy is a Python library which allows users to describe the structure of their SQL database using Python classes, and handles the creation of relevant SQL code behind the scenes - users simply have to instantiate their class and then execute a few commands around the object in order to save new data to their database. Given that our primary focus during this project was on Python code, we knew that using SQLAlchemy would allow us to focus on Python without having to worry about correct SQL syntax, whilst giving us another chance to work with classes.

A final list of technical requirements for running the app can be found in our README.md file.

Non-Technical Requirements

Usability. The system should be user friendly. We will design our user interface so that it is immediately obvious what our application allows the user to do. The main site will display the location output and weather output prominently next to the grabbed playlist of songs which will be displayed centrally. This simple design makes it easier for the user to explore and discover the possibilities of the app intuitively.

The application will also display a media player to play songs with appropriate graphics that compliment the song selections. The player will have simple icons that are widely recognisable. This gives the user some possibilities of interacting with the application. There will be few controls on the player that should appeal to both a naive and experienced user, allowing them to perform actions quickly and learn from their discoveries. Although we have opted for few error messages, we believe that by reducing the range of commands and options on screen allows users to quickly repair choices.

We will give the user some control over the playlist including the capacity to stop, play and skip a song which allows some flexibility. We want to ensure basic functionality and in future development cycles the controls are something that can easily be updated with new features.

The commands and controls will be memorable so that users will not forget what they have learnt. A small range of controls and options will prevent the user from being overwhelmed when using the application. A simple and fluid first user experience will create a lasting positive impression.

We will only allow a default display setting on the website. Due to the time scale for the project we do not have enough time to develop features that allow the user any further flexibility with regard to display settings (e.g. light mode / dark mode / visual impairment accessibility settings).

The user must be able to login and logout of the application. There must be a form that the user should be able to complete that asks for username and password. The user should be able to access these options easily by clickable dynamic buttons. The user will also have the ability to return back to the home page and register using clickable buttons. These buttons will clearly be labeled to aid the user to navigate the page. We have opted for four basic navigation buttons that are widely recognized commands which are clear and explicit for the user to understand.

We have decided not to include any instructions or frequently asked questions page as we believe the simplicity of the website has been designed thus, that users will intuitively understand and discover how the site works quickly. With a longer time frame we would have time to produce detailed documentation that offers help to the user.

The application must be fully responsive on both desktop and mobile phones. The web application should function smoothly between devices showing consistency of the product.

The user interface will be designed using a colourful palette that reflects our applications' fun appeal. Using a specific palette and colour scheme will ensure consistency throughout the site in its look and feel.

We will create some wireframe examples that display how the user interface can be developed further depending on the progress we make in the allocated time scale.

Accessibility. The app should have a filter that removes all songs with explicit lyrics so content on the site will be suitable for children. Therefore, the app will be accessible to the widest range of ages.

The app will have the capacity to play audio, which allows the app to be enjoyed by those with visual disabilities or impairments. However, the app is limited in audio descriptions, which would allow the user to navigate the website with greater ease. With greater time and resources adding audio description, especially to the navigation buttons would allow further accessibility.

As the app's primary feature is its capacity to play music it will not be designed to be accessible to those with hearing disabilities or impairments.

The fonts on the website will only have a default setting. As we will prioritise simplicity over flexibility there won't be an option to enlarge font type or change colour. Unfortunately this will reduce accessibility to those with visual disabilities or impairments extending to colour blindness.

As we are using freemium APIs the application is currently free to use, which opens the app up to a wide socio-economic demographic. The app will not require a subscription or up-front payment to use at this stage. This should make the web application accessible to a wide user base from different backgrounds.

The web application requires a location grab as well as the local weather. It is very dependent on those locations being well connected with good service and heavily reliant on GPS. We anticipate that rural areas and developing countries may not have the services to support this app and is therefore not fully socially inclusive.

All navigation buttons and website text including the playlist will be presented in English. We have decided that this will reach the widest possible audience. However, we are aware that despite being the world's most spoken language it does exclude people and communities who do not understand English. With more time and resources, options for translation could be added to accommodate a wider range of communities and languages.

Availability. The web application should aim to be available 99.5% (two and a half nines) in uptime in line with the market average. However there are several factors that are likely to have an effect on this average.

The application will be reliant on several APIs and is dependent on those web servers being available. We have no control over the maintenance of these APIs and servers. Therefore this will likely reduce the availability of the application.

Due to the small budget and size of the team we anticipate there will be no dedicated maintenance team. Therefore if there is a disruption to the service there will undoubtedly be delays with regard to returning the app to fully functioning service. This will also make it difficult to structure and deliver scheduled updates promptly, which is likely to have consequences once the application has been deployed.

Scalability. We will consciously adopt a sustainable design to program our app, particularly with regard to the code used in our back-end services. Employing an object orientated approach and implementing SOLID principles will ensure increased sustainability of our code. The more sustainable the code the easier the code will be to scale new updates and features. The development team has varying degrees of experience delivering these concepts and therefore maximum sustainability cannot be guaranteed. This will ultimately affect the app's scalability later on.

Ideally our web application should be able to support +1500 users which is that of an average new website. We anticipate a predictable growth in our users but for this development cycle we don't believe there will be any abnormal jumps in the system that may cause a system failure. Therefore, we have decided against a ramp-up test, which would test where the saturation point is that would cause the system to malfunction. Also in the time scale we have this will not be feasible.

One of the biggest factors affecting scalability is the app's use of third-party component integration. We include two APIs that are used within the app that we have limited control over. The amount of calls that can be taken by the geolocation API currently stands at 300 calls per day, with the weather API limiting to 60 calls a minute/1,000,000 calls a month, and Spotify imposing rate limiting. As it stands we believe these two APIs offer us enough scalability to support our first deployment of the app. As the growth of our clients increases, the call rates of these APIs will have to be taken into consideration.

As we don't anticipate a bottleneck within our first deployment, we have dedicated little of our very short time frame to additional resources that would manage the recoverability of the system if it was to malfunction.

Performance. The web application should be fully compatible across all operating systems. The web application should function on both Android and iOS. The website will be designed so that it is responsive on both mobile devices and desktop.

For this deployment we will not have the resources to carry out an endurance test to determine how long the application can handle load. We anticipate that as this is a new website that has not been marketed we do not expect a large amount of users immediately. It is unlikely that there will be multiple users accessing the program concurrently.

We expect our website to load in under 3 seconds for a good user experience. However, context here is important and our application needs to download and playback music. Removing this player would improve loading speed but it is essential for this project.

The web application will use approx 390MB memory which is average for this type of web application. As we will not be testing for memory leakages it will be difficult to determine a calculation.

Reliability. The application should load the same page every time the user uploads the site. When the user selects any clickable option the web application will deliver the same expected results every time. For example; the login option should be taken to a username and password form every time. When the user selects Register they should be taken to the corresponding form.

The application will demonstrate integrity with regard to third party services used in the app. The safety of people and possessions will not be damaged in the deployment of our web application. It is vital that we protect third party interests as they are fundamental in the design and functionality of our product. It is also crucial that any people associated with our application do not have their reputations damaged by our product.

Due to the short time frame we won't designate any time to develop a disaster strategy should there be a fatal malfunction. We understand that this will need to be considered at some point in the next development cycle.

Security. We will give users the ability to create a unique encrypted password that will allow them access to the web application. These details will be stored and backed up in a private database.

As we have no budget for this project we are relying on free services. We therefore do not have a budget for anti-virus products and cannot prevent the application transporting viruses or appearing as one.

Due to the limited time frame and the design's main focus on user functionality, we decided to allocate less time to the security of client information during this development cycle. We recognise that the database could be made more secure by creating limited views and using additional resources. This could further be improved by implementing timed back-ups.

The application also uses a feature that grabs the location of the user without requesting permission. For the purpose of guaranteeing performance and functionality at this stage we will need to compromise on some features that would require more time from the development team. We felt this particular feature would take time away spent on features guaranteeing minimal functionality to the user which was our ultimate priority.

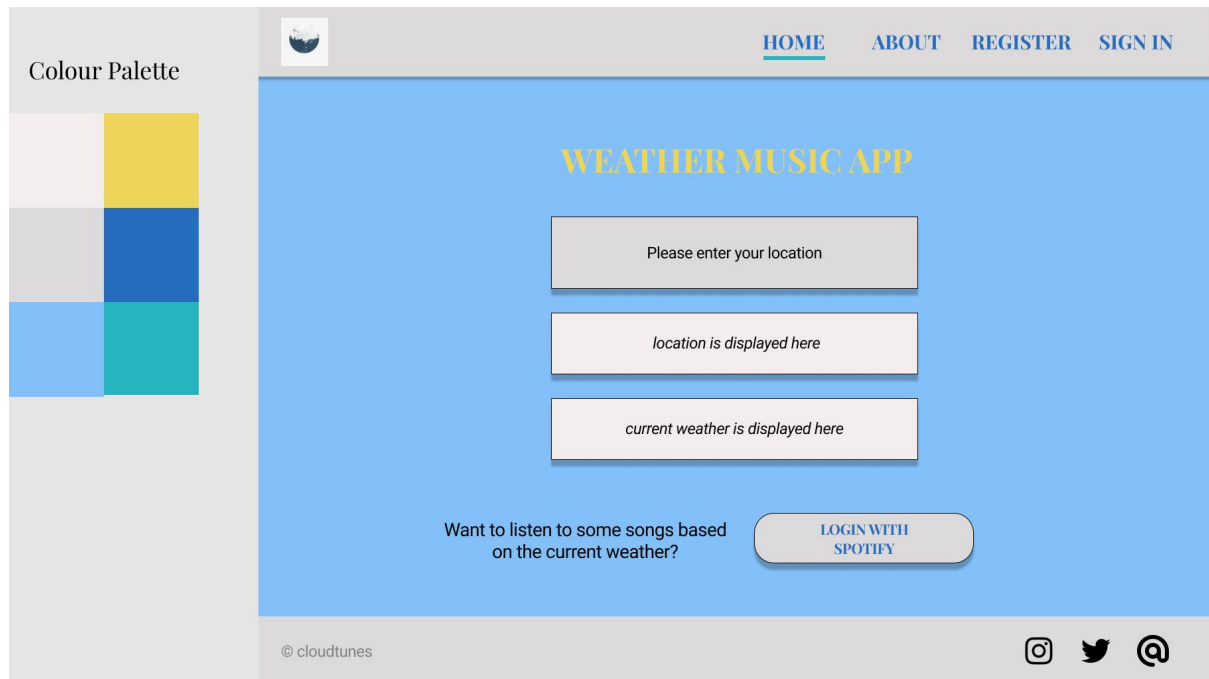
The data gathered by registered users will be stored in a database. The priority for this project will be that the application delivers functionality to the user. The short time frame will not give us enough time to make a completely secure database that will not have any vulnerabilities.

We will not display any information of the contributors to the application on the website or any user information including email address or phone number. We will not have an email address or alerts as currently we cannot prevent our project being susceptible to social engineering attacks.

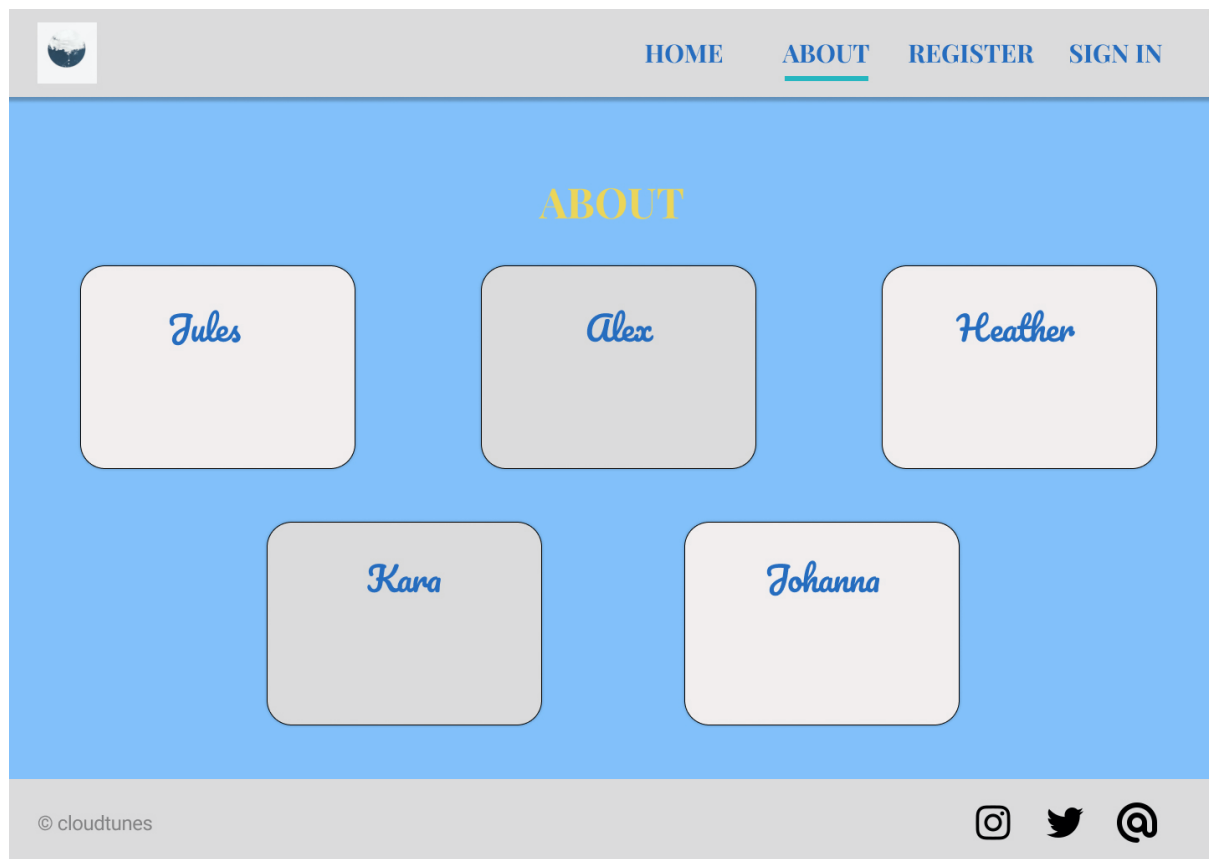
We will try to maintain the secrecy of our authentication however we have discovered a vulnerability within the underlying system that exposes a username of a contributor. To allow us access to the Spotify api this username will remain hard-coded into the system. For the next development cycle it will be essential that this username is replaced by that of the user.

Design and Architecture


Below are the wireframe designs for the user interface for the application:






Home Page




About Page


[HOME](#)
[ABOUT](#)
[REGISTER](#)
[SIGN IN](#)

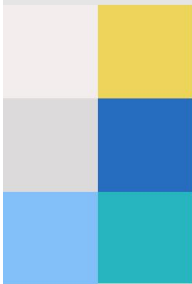
CREATE ACCOUNT

© cloudtunes
 






Create Account Page


[HOME](#)
[ABOUT](#)
[REGISTER](#)
[SIGN IN](#)

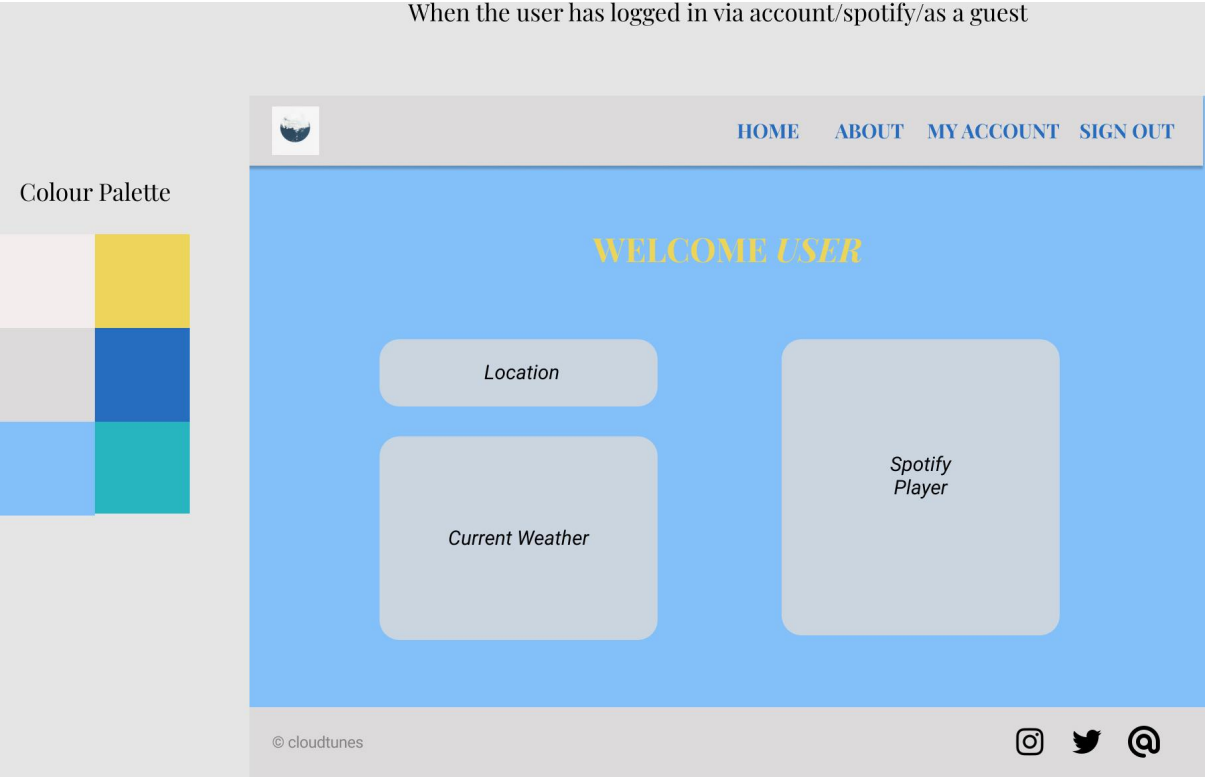
Colour Palette



LOGIN

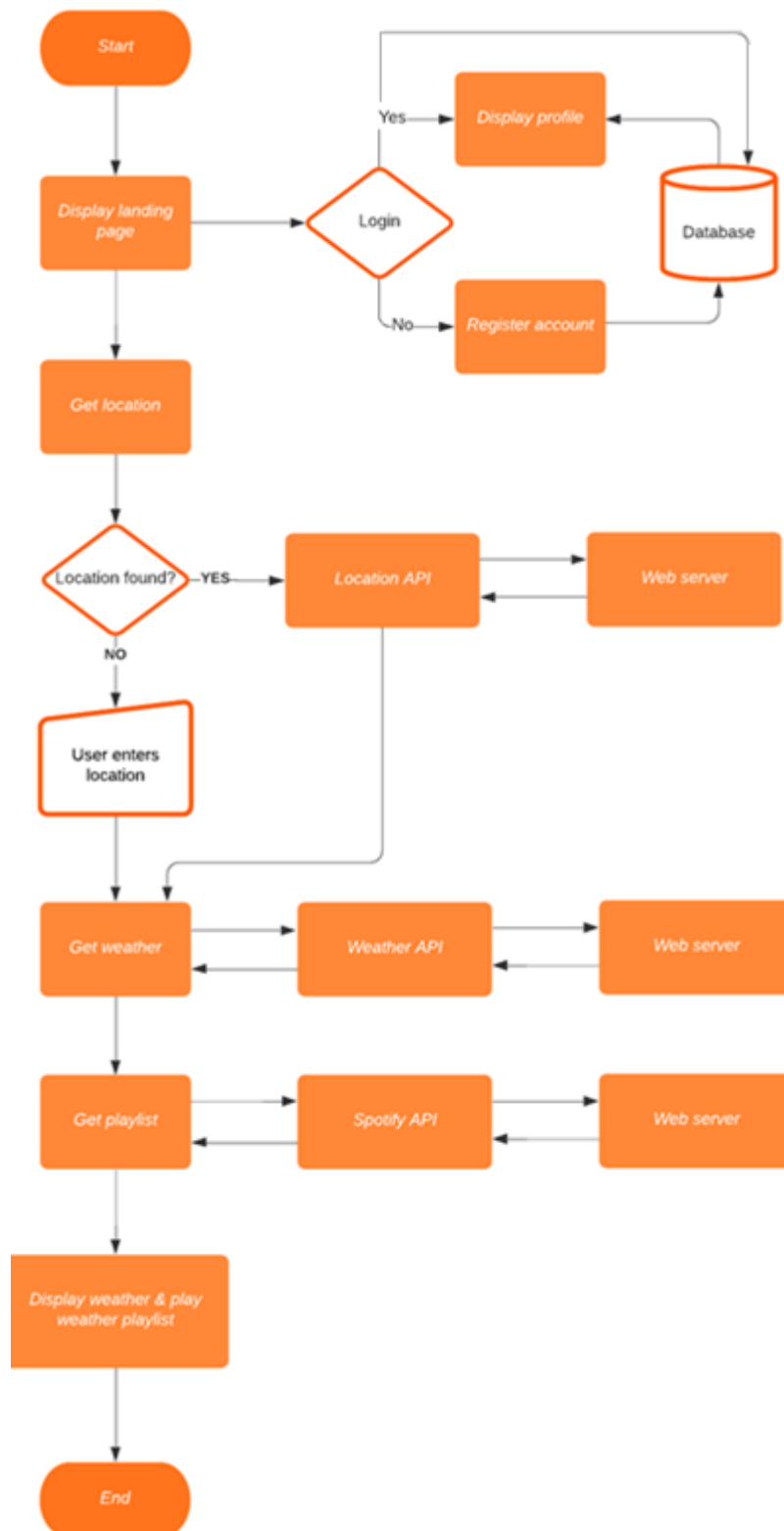
© cloudtunes
 



Login Page



Weather Music App Page

Below is the architecture design for this application:



Implementation and Execution

Development Approach

Our initial plan was to take an agile approach to the development of our code, with a daily scrum and iteration on our code. However, due to the nature of the project, we ran into difficulty with this - for example, some group members were balancing the course with other full-time commitments, which meant that a daily scrum was unrealistic. Additionally, the project timeline was only four weeks, so we were unable to divide our work into individual sprints - and, frustratingly, one team member was absent for the first week due to a prior commitment, which meant that we were slower to get started than would have been strictly ideal. As a result, the system we ended up using was much more of a waterfall approach, but we took inspiration from agile in some areas.

Once we were able to meet properly, we broke the project up into its component pieces, including testing, calling the APIs, creating a Flask app, and designing classes. From this pool, team members were able to choose which areas they were interested in working on. We ultimately met twice weekly (more frequently within the latter stages of the project) to update each other on what we were working on, what we had achieved, whether we had run into difficulties, what we hoped to work on next, and outstanding components on the project - this we modeled on the daily scrum. We also set up a Slack channel, which we used to communicate, send resources, and send short code snippets. The majority of our coding was shared on GitHub, although at one point we ended up with two different repositories (one which was handling API calls, another which was handling the majority of our code base), so a third was created to merge together the code (this is the repository in which our project was submitted).

Tools and Libraries

In addition to our own modules, the following libraries were used, listed along with their applications:

- flask - as we used the Flask framework to build our site, the flask library formed the backbone of our project. This library comes pre-bundled with other tools, including the Jinja templating engine, and Werkzeug, which handles communication between the user-created app and remote servers.
- flask_sqlalchemy - added Flask support for SQLAlchemy to our application.
- flask-bcrypt - used to encrypt user passwords securely within our database.
- Flask_session - Flask_session.Session() was used to handle storage of session variables - namely, a Spotify session that would store a Spotify user access code.
- spotipy - developed as a more concise way of interacting with Spotify's API, we used spotipy to assist in our API calls by handling the passing of client credentials to Spotify and parsing of returned information.
- wtforms - provides forms rendering and validation for Python web development, allowing us to easily display and implement login/registration forms.
- flask_wtf - allows for simple integration of wtforms with Flask.
- requests - requests.get() used for making requests to external APIs.
- json - json.loads() used to parse incoming data from APIs into a usable Python dictionary
- random - random.shuffle() used to shuffle a playlist
- flask-login - provides user session management. Handles tasks such as logging in, logging out and remembering user's sessions.

As well as importing libraries, we used other tools: Bootstrap, for example, was used to format our front end, and granted us the ability to create a mobile responsive user interface. In combination with Flask's Jinja, this allowed us to create a dynamic design that matched our original wireframe. We also used DBFopener (<https://www.dbfopener.com>) to view the database and ensure it was behaving as expected.

Implementation Process

The biggest challenge we faced along the way was definitely communication. At the point of starting the project, many of the team members were still getting to know each other, and so there was a general unwillingness to take charge and risk appearing domineering or dictatorial - this in turn led to difficulties in decision-making. Reflecting back, if one person had been willing to take charge of discussions and ensure that decisions were made, we might have had success earlier. We struggled to balance project work alongside everyday commitments, as some of us had full-time jobs and childcare responsibilities - between these, which were priorities over coursework, and our communication difficulties, we found it difficult to keep up with who was working on which parts of the project at times. There was at least one point where this resulted in two people working on the same thing without the other's knowledge, making the time and effort of the other person redundant when the first person had achieved what they had hoped to.

In terms of actual coding, one of the most difficult things we encountered was implementing the Spotify user authentication flow. Spotify's API has several different authorisation flows, and while we initially had success with the Client Credentials Flow (which enabled us to request and receive basic information about music from their API), we struggled with the Authorization Code Flow (which was required in order to return data about user playlists). This is possibly due to the fact that we were using the Spotipy library, which simplified our code by abstracted some of the process away from us, but ultimately meant that we had to seek assistance from a programmer more experienced in OAuth2 protocols to understand why this code might not be producing the desired outcome.

We eventually determined, after reading through the Spotipy documentation and source code, that Spotipy was likely not intended to be used with Flask, due to the fact that the authorisation callback URL was being routed back separately, likely to the server rather than to the front end (which was difficult to determine as, in initial testing, both the front end and the server were being operated on the same machine by the same person). However, the end result was that the authentication flow wasn't being completed. The Spotipy documentation did not help in this regard, as the authorisation flow was not discussed in detail, but we eventually resolved the issue by examining which methods returned which desired effects in the source code. Unfortunately, attempting to get this key functionality working took a significant amount of time out of the implementation period, especially since, at one point, this was a task on which everybody was working due to how vital this functionality was to our eventual goal. This did mean, however, that overcoming this difficulty was one of our greatest achievements.

We also struggled to accurately imagine what we would be capable of achieving within a given timeframe, and so early on, we suffered from scope creep. One of the major benefits of working as a team was that, if we were getting over-enthusiastic in proposing new features, there were other people present to be able to draw our attention to this. This meant that, for example, when a proposed new feature was not likely to be implemented in time for the deadline, we could put some of the more basic structures in place that would allow us to implement it in the future and showcase our intent, but not include the feature itself in our original minimum viable product.

One major problem worth pointing out in our implementation is that in our code, one of the developers' Spotify username is hard-coded into the Spotify user authentication route. We recognise that hard-coding user data into code is a bad decision, but opted to include it as a username is required in order for the function to work. In a release, we would of course remove it in favour of asking for the user's Spotify username, but this would have required implementation of front-end features that we decided were less important than our Python code.

Testing and Evaluation

Testing strategy was inspired by the waterfall approach, completing unit testing after bulk of coding had already been completed. We used unit tests to check that our API functions were working, ensuring that these functions would be able to catch exceptions where expected, and testing how they would behave when input was not given as expected. We also implemented some unit tests to demonstrate the capabilities of one of our underused classes, Playlist, which we ultimately hope will be used to handle movement between tracks and other music player functionalities - as this has not been implemented for this release, we have included a few methods that are unused but will be useful in the future.

The user testing was carried out as each individual component was completed by its contributor. Although this is not in accordance with the waterfall approach we realised quickly that in the short time scale it was a quicker alternative to leaving the user testing until the end. As we were a fairly inexperienced team and the majority of the APIs were new to us, as well as using them in combination with Flask. This made it difficult to accurately schedule how long each feature would take to build and how long realistically we should allocate for user testing. There would have been a likelihood that had we taken a waterfall approach to user testing here, with an inexperienced team, the amount of errors and bugs would have resulted in missing the project deadline and not being able to guarantee the minimal functionality we had prioritised.

Due to only having four weeks to implement the website, learning new tools as we went, and setbacks due to absence and illness, we were unable to implement the app to quite the scope we had originally hoped - for example, a full app would also allow for much greater integration of the user's own Spotify playlists. While the functions to get that data from Spotify exist and function as expected (having been user tested with the terminal), we elected not to use the data on the front end, as this would have required dedicating significantly more time to the front end at the expense of more important areas. However, even though we would have completed more features in a perfect scenario, we can happily say that our product functions as we expected it to, and the vast majority of the features that we had hoped to implement have indeed been implemented.

Conclusion

While we would perhaps have liked to have achieved more of a “complete” product, we were successful in delivering the key functionality of our project as part of a minimum viable product, and despite not needing to create a front end, we were able to implement one in any case, giving a sense of the fun user experience that we had intended to achieve.

There were several key factors that ultimately impacted on our success (limited time, lack of experience, limited resources), and we are confident that there is room for improvement in our project. We would like to add more features, for example, and give a greater consideration to aspects such as security - it would be advisable in a public release to allow the user to agree to their location being shared, and as users have the option to create an account, we would add the ability for users to alter their account details (such as resetting a forgotten password).

Additionally, we would probably benefit from having a different methodology in the future - while waterfall did not pose too many problems in a short-term project, implementing an agile methodology such as scrum would perhaps enable us to manage our time better by dividing our work into limited sprints, encourage better communication with a daily standup, and encourage more thoughtful prioritisation of tasks. We would go into the next development cycle with a clearer idea of what we wanted to accomplish, and with a better sense of the timeframe in which we would be able to do it.