Festival Meetup app

Our project aimed to build the “Festival Meetup” app, designed to connect music festival goers in new and exciting ways. The primary goal was to create a platform that allows users to discover and connect with others based on shared music interests or plans to attend the same festival. The app enables users to find new festivals, interact with fellow festival enthusiasts, and showcase their own music tastes.

Key objectives included integrating an external API to facilitate the discovery of new and upcoming festivals by providing detailed information about various events. Ensuring user security was another priority, achieved through a robust authentication process that verifies users during sign-up to secure app access.

We also aimed to foster user connections by implementing messaging features and a feeds page. Personalization was a core objective, allowing users to create and customise their profiles to reflect their music preferences and festival plans, enhancing the app’s engagement. Finally, a well-designed and stylized interface was essential to provide a seamless and enjoyable user experience.

This report provides a comprehensive overview of the “Festival Meetup” application, detailing its development from initial concept to final implementation. It covers the background of the project, technical and non-technical specifications, implementation strategies including tools and libraries used, and the testing and evaluation processes to ensure functionality and usability.

Background

“Festival Meetup” was designed as a social platform specifically for music festival enthusiasts, with the core goal of facilitating connections among festival-goers before the event. The app is built around several key features to enhance user engagement and interaction.

Users start by signing up and providing basic personal information. Once registered, they are redirected to the login page, where they can access and personalise their profiles. This customization allows users to update their profiles with details such as music preferences, favourite artists, and festival plans, ensuring a tailored experience that reflects their individual interests.

A notable feature of the app is the festivals page, which provides a comprehensive list of upcoming music festivals. This section is vital for users to discover new events and

plan their festival attendance, offering detailed information about each festival to help users make informed decisions.

The feeds page serves as the social hub of the app, where users can write posts, share their festival plans, and comment on others' posts. This feature fosters community engagement by enabling users to connect with others attending the same festivals, thereby creating opportunities for real-life interactions.

Together, these features—personalised profiles, festival discovery, and social feeds—create a cohesive experience that allows users to move seamlessly from planning their festival experiences to interacting with the festival community. The app's design prioritises user-friendliness, ensuring effortless navigation between features and enhancing overall user engagement.

Specifications and design

Authorisation

The Festival Meetup app allows users to connect with other members through a secure and user-friendly registration and login process. To achieve this, the app implements an authentication system that ensures ease of use and secure data handling. The authentication process relies on a backend database for storing user credentials and profile data. Passwords are secured using bcrypt, which hashes the password to protect sensitive data. The app uses token-based authentication, where a JSON Web Token (JWT) is generated for session management. To ensure security, secret keys are managed via environment variables, while dotenv is used to securely handle sensitive API keys.

The app not only focuses on technical security and efficiency but also on providing a user-friendly experience through thoughtful UX design. The user authentication process is designed to be intuitive and seamless, ensuring that users can easily register and log in. When users sign up via the */register* endpoint, they are guided through a simple and clear interface that includes friendly error messages if their email is already in use. Passwords are securely hashed using bcrypt, but this security measure is abstracted from the user, providing a smooth onboarding process without unnecessary complexity. Additional user information, such as name, email, profile picture, and location, is also collected during registration to enhance personalization across various features of the app.

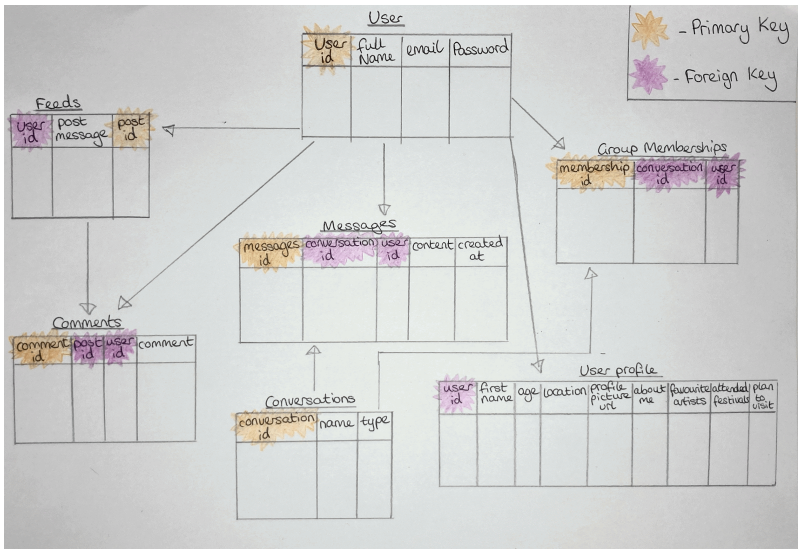

Image of how the SQL tables connect

Database and use

A key requirement for our project was the backend database used to store all the vital information needed for each page. This was created using MYSQL at the start of the project based on the initial wireframes that were designed. Throughout the project each team member has utilised and adapted the tables with the necessary information needed for each feature. The intricate design of the database had to ensure that the correct Primary keys and Foreign keys were used so that the tables linked together correctly. A sample of dummy data was inserted into each table to show that they were each working correctly.

After the authentication, once the user has logged into the app they are redirected to the default profile page. The user can edit this page to enter their own details and the profile page uses a put request to send and update this information in the ‘User_profile’ table in the database. The profile page also uses a get request to get all of the information from the same table for the profile that the user has previously provided. This is linked through the user id when the user logs in.

The feeds page uses the database by storing posts and comments that have been added by users. When the feeds page is loaded it initiates a get request to gather the information to be displayed on the page from the ‘Feeds’ table within the database, this includes the user’s name, profile image alongside their post. When a user writes a post on the feeds page, a post request is run to send their message and add it into the database. The page also offers an option to add a comment to an existing post. When the comment button is clicked it sends a post request to the database to insert the new

data into the ‘Comments’ table. A get request is performed to display comments that have previously been added to existing posts.

The messages page uses the database by managing user conversations and storing the messages that are sent within them. When the messages page is loaded, it initiates a GET request from the ‘conversations’ and ‘group_memberships’ table to display all the messages associated with the logged-in user. The ‘conversation’ table includes conversation ID, name of the chat, and the type of chat (either private or group), and the ‘group_memberships’ table links users to conversations through user ID. When the user selects a conversation, a GET request is made to fetch all messages within that conversation, querying the ‘messages’ table and filtering the results by the conversation ID. The ‘messages’ table stores each message's content and the time it was created, making sure the messages are displayed in chronological order. These messages are retrieved and displayed in real-time by using polling, where the frontend continuously sends GET requests every 5 seconds to the backend to fetch new messages in the selected conversation. Sending messages results in a POST request, targeting the ‘messages’ table, which includes the conversation ID, user ID, and message content. The backend then processes this request, inserting a new record into the ‘messages’ table and the frontend triggers another GET request to refresh the displayed messages. Finally, when a new group conversation is created or an existing conversation is updated to add/remove participants, INSERT and DELETE requests are made respectively using the ‘group_memberships’ table.

Festival API

The "Festival Meetup" app includes an API integration with Ticketmaster, designed to enhance the user experience by retrieving and displaying festival data in the frontend React application. This API was developed with both technical and non-technical requirements to ensure optimal functionality and a user-friendly interface.

The backend server was built using Express.js, with Node.js as the runtime environment. Axios served as the HTTP client for making requests to the Ticketmaster API. The primary API endpoint supports pagination, allowing users to navigate through various festival events. The route configuration permits optional query parameters, such as page numbers, and constructs the necessary URL for querying the Ticketmaster API. This includes parameters for classification (Festival), page size, and the API key.

To ensure security and performance, the API key is stored in environment variables, preventing exposure in client-side code or logs. The API was designed to handle high traffic and large datasets efficiently, maintaining responsiveness. It also features

error-handling mechanisms that return clear, meaningful error messages when requests to Ticketmaster fail.

The system architecture relies on Express.js for server configuration, route definition, and port management, all controlled by environment settings. The key route processes GET requests to fetch festival data, which is returned in a structured format including festival names and IDs. If errors occur, the API responds with a 500 status code and a descriptive failure message, ensuring issues are handled gracefully.

The UX design of the "Festival Meetup" app ensures that, while the backend focuses on security, performance, and data management, the user experience remains smooth and intuitive. The app's design balances robust security with ease of use, providing users with a seamless experience as they interact with various features.

Implementation and execution

Our team decided to use an agile development approach to be able to adapt our tasks and targets throughout the project. We decided initially to divide up the tasks and the workload evenly by allocating each member with a different page of the app to work on. Communication was crucial throughout the process to move forward with tasks and piece all the different parts of the code together.

At the start of the project we each produced an individual SWOT analysis in which we stated our strengths and weaknesses and how they might affect our work. This made it clear how some members of the team felt their strengths lied in certain areas such as API's or CSS and highlighted any concerns that team members had about their abilities. By assessing our SWOTs we decided who should start with each area. However, we have found throughout the project that we have been able to help each other in different areas, often sharing the workload between us to achieve our tasks.

Pair programming has also been invaluable to help us achieve our goals for this project, a great way to learn from each other and most importantly to solve the challenges we came across whilst coding our different parts.

Tools and libraries

Tools:

Node Package Manager (node.js and npm): the runtime environment for developing our React application.

Visual Studio Code: code editor for writing and editing the code for our project that supports React applications.

Git and GitHub: a version control tool to manage the codebase, track the changes, collaborate between team members and host our project repository.

Postman: used for testing API endpoints during the development of our app to ensure the features needed on the backend server work as expected.

Libraries:

React: used to build the user interface of the app through components.

React Router: manages the routes throughout our application to navigate between different pages and handle the URLs.

Express.js: used to build the backend API for the React application. Cors and body.parser have also been used to communicate between the frontend and backend more seamlessly and extract the data needed.

React Bootstrap: this was used to integrate the modals for the login page and the navigation bar.

Axios and Fetch API: makes requests from the frontend to the backend server.

Ticketmaster API: a third party API to integrate the data about different festivals within the application for the user to browse.

MySQL: database used to store user credentials and profiles which connects the frontend and backend of the React app through API calls. The `pool` makes these connections efficient.

JSON WEB TOKEN (JWT): used to create tokens for a secure authentication process when signing up and registering the user.

bcrypt.js: hashes passwords on the backend so passwords can be stored securely in the database.

The project has been a significant achievement for our team, marking our ability to collaboratively develop a web app from scratch. Successfully implementing an API, integrating all pages through the backend, and designing a responsive web interface were notable milestones. Our most substantial challenge was merging our individual contributions into a cohesive whole, which required extensive teamwork and problem-solving to achieve our goals.

Throughout the project, we faced numerous challenges and learned valuable lessons. One of the major hurdles emerged towards the end when we attempted to combine our separate pieces of code. This process revealed compatibility issues, resulting in a broken app and access problems just days before the deadline. This situation forced us to regroup, troubleshoot, and methodically address the issues. The phrase "one step at a time" became our mantra as we worked together to resolve the problems, highlighting the importance of patience and collaborative effort.

Another challenge was integrating the live chat feature into the app. Although it functioned independently, we struggled to fully connect it with the rest of the application, limiting its effectiveness despite successful styling. For future projects, focusing on one feature at a time could improve integration efficiency.

The experience taught us the importance of frequent code updates and communication within the team. Regularly pushing updates to our development branch and ensuring our local repositories are synchronised with the remote repository will help maintain code functionality and coherence. This project has underscored the value of teamwork, persistence, and continuous learning in software development.

Testing and Evaluation

Testing Strategy

The unit testing strategy for the API focuses on validating the core functionalities of the API endpoints. It aims to ensure that the API behaves as expected under various conditions, including normal operation and error scenarios.

Functional Testing

Functional testing ensures that each API endpoint within the Festival Meetup app performs its intended functions accurately. For the `/api/festivals` endpoint, testing involves verifying that festival data is correctly returned when the Ticketmaster API responds with expected data. This process includes confirming that the request is properly handled and results in a status code of 200 OK, accompanied by the correct festival information. In scenarios where the external Ticketmaster service fails, functional testing also assesses error handling, ensuring that the API responds with appropriate error messages and status codes, such as a 500 Internal Server Error. This is critical for maintaining a reliable and user-friendly experience, even in the event of external system failures.

User Testing

While functional testing is essential for validating the backend logic, user testing plays a complementary role, focusing on how the API behaves from the user's perspective. Although user testing is less applicable to unit tests—given their primary focus on backend functionality—it still includes checks for the validity of API responses. This involves ensuring that the data returned to the user is in the correct format and contains the necessary fields for display. Furthermore, user testing verifies that error messages are not only accurate but also informative and user-friendly, allowing users to understand and address any issues they may encounter during their interaction with the app.

System Limitations

Despite the thoroughness of functional and user testing, some system limitations remain. For instance, unit tests mock external services like the Ticketmaster API, meaning they do not account for real-world issues such as network connectivity problems, service outages, or unexpected changes to the API. Additionally, unit tests may not fully capture environment-specific issues that could arise between different deployment settings, such as variations between production and development environments. Finally, while unit tests are valuable for isolating individual functions, they may not adequately simulate more complex user interactions or workflows that span multiple API calls or integrate with other systems. These limitations highlight the importance of complementary testing strategies to cover edge cases and integration challenges in the system's overall design.

Conclusion

Within the given timeframe, our team successfully developed a web app that aligns with our initial vision. We implemented several key features aimed at enhancing the user experience, demonstrating solid practices as beginner software developers.

Throughout the development process, we encountered and overcame various challenges, which provided valuable learning opportunities. Some issues, such as the disappearing user profile picture and incomplete functionality for password reset modals, would require additional time to resolve fully.

Looking ahead, we had planned to incorporate additional features to further improve the user experience. These include implementing a light and dark mode using Redux, completing the messaging functionality to facilitate user connections, and enhancing the app's styling with CSS or Bootstrap. With more time, these enhancements would help create an even more seamless and polished user experience.