Task 3 – Documentation & Evaluation

Consuming a 3rd Party API

The Spotify Web API was chosen as a third-party connection to improve the event booking system and give users a more engaging experience. Users may obtain more information and interactive media about the artists they are interested in by utilising Spotify's large artist and music data. This helps consumers make better judgments when purchasing tickets by providing information about artist popularity, genres, profile pictures, and even a list of the most popular songs.

Presenting current and dynamic material about musicians playing at coming events was the main goal of this integration. Artist data is frequently static or lacking in typical concert systems. The system's ability to retrieve real-time data straight from Spotify's servers through integration makes the user experience more engaging and up to date. The ability to provide an artist's bio, popularity rating, genres, picture, and—above all—their top tracks with 30-second preview links that viewers can listen to straight through the interface are among the main features made possible by this API.

OAuth 2.0 protects Spotify's Web API, ensuring that all queries are safe and approved. Registering the application in the Spotify Developer Dashboard was the first step in authentication. A distinct Client ID and Client Secret were supplied by this registration. The credentials were then sent in a base64-encoded manner via a POST request to Spotify's token API at /api/token. The short-lived access token that Spotify provided had to be included in the header of all subsequent API queries. It looked like this: Authorisation: Bearer {access_token}. The token was kept and used again until it expired, at which point a new one had to be requested.

The project made use of two primary Spotify endpoints. The first was the search endpoint, which enabled the system to use a GET call to https://api.spotify.com/v1/search to look for an artist by name. Two arguments were needed for this request: type, which was set to artist, and q for the artist's name. In response, Spotify sent a JSON object with comprehensive artist data, including the artist's ID, name, photos, popularity score, genres, and number of followers. A second API query was then made using the artist ID.

The artist's top tracks were obtained at the second endpoint. A GET call to https://api.spotify.com/v1/artists/{id}/top-tracks was used for this, with the necessary parameter market set to US. An array of the artist's most well-known songs, together with the title, URL for the preview, and related album art, were returned by this call. This information was very helpful for integrating interactive track previews into the concert details page's user interface.

The booking system's UI immediately made use of the data that was recovered. Alongside concert listings, the artist's name, genre, and picture were provided, and a playable selection of their best songs was displayed. This innovation made the system a

more engaging and knowledgeable way to buy tickets by enabling consumers to hear the musicians' music before deciding to make a reservation.

Spotify's developer documentation was essential to the integration process. The documentation offered thorough instructions on how to organise requests, manage answers, safely authenticate, and access the various endpoints. It included information on rate limits and recommended practices, as well as sample queries and explanations of every parameter and response field. Due to trial and error, implementation would have taken a lot longer without this legitimate advice.

Evaluation of API Testing

Throughout the API's development and testing, Postman was heavily used. It was the primary setting for testing endpoint behaviour, mimicking HTTP queries, and spotting any problems. Effective testing, quicker debugging, and a better understanding of how the API will function in practical applications were all made possible by the use of Postman.

ConcertBookingAPI is a Postman workspace specifically designed for this purpose. This made it possible to arrange the API queries and collections in an understandable manner. Users, Concerts, Bookings, Favourites, and Spotify were all given distinct collections in the system. From basic GET queries to more intricate POST and DELETE operations, including JSON request bodies and authentication headers, these sets included every request required to test functionality.

Every request was carefully made. POST and PUT requests used raw JSON in the request body, while GET requests included query parameters. Where needed, all essential headers, including Content-Type: application/json and Authorisation: Bearer {token}, were supplied. This request structure made it simpler to spot problems and mimic actual use situations.

Multiple testing scenarios were raised. These included both positive situations with legit input and negative scenarios that assessed the system's ability to handle inaccurate or absent data. To make sure they followed to standard practices, status codes and error messages were regularly observed. Status codes like 200 OK, 201 Created, 401 Unauthorised, 404 Not Found, and 500 Internal Server Error were given special consideration.

Postman's features, including environment variables and pre-request scripts, were investigated in addition to functional testing. Pre-request scripts were utilised to automate the insertion of tokens and custom headers, while environment variables facilitated the rapid change between development and production URLs. An additional degree of validation was added to the testing process by the ability to automatically check for status codes and certain response content using Postman's built-in test scripts.

The mock server functionality of Postman was tested in the early stages of development. This made it possible to simulate endpoints before the database and complete backend were prepared. By returning pre-defined JSON replies, mock servers allowed for the development and testing of frontend components before the whole API capability was available. This method helped provide the groundwork for API contracts early on, even if it was not utilised in the finished product.

All Postman collections were exported and stored in the project directory under API/Exports/Postman/ after the conclusion of the testing session. Anyone who wants to test the API on their own or carry on with development can re-import these collections. The collections are used as instruments for verification as well as documentation.

To sum up, the testing procedure made sure that every endpoint operated as planned in a range of scenarios. Postman offered the resources required for comprehensive system validation, effective debugging, and verification that the application satisfies the expected specifications. The final system's completeness, usability, and dependability were greatly enhanced by the testing procedure.