

PRODUCT DESIGN DOCUMENT

Concertify – Music and Event Tracker Application

COMP.SE.110 (Fall 2024) – Group Assignment

Document version: 1.0

Last updated: November 30th, 2024

Software Developers:

Xuan An Cao

Minh Khanh Pham

Gimantha Dissanayake

Dávid Lakos

Erfan Niketeghad

TABLE OF CONTENTS

1.	Project Overview.....	5
1.1.	Description.....	5
1.2.	Motivations	5
1.3.	Goals and Objectives	6
2.	Product Requirements and Features	6
2.1.	Functional Requirements	6
2.2.	Key Features.....	7
2.3.	Traceability Matrix	8
3.	System Architecture.....	9
3.1.	High-Level Architectural Design	9
3.2.	Components and Relationships	11
3.3.	Database and Caching System.....	14
3.4.	Project Modules and Directory Structure.....	16
4.	Design Patterns, Interfaces, and SOLID Principles.....	17
4.1.	Factory Method Design Pattern	17
4.2.	Composite Design Pattern.....	19
4.3.	SOLID Principles	21
5.	APIs, External Libraries and Dependencies	24
5.1.	APIs	24
5.2.	External Libraries and Dependencies.....	25
6.	User Interfaces.....	26

6.1. Designed Interfaces.....	26
6.2. Implemented Interfaces	27
7. Usage of Artificial Intelligence.....	30
8. Final Self-evaluation.....	31
8.1. Evaluation	31
8.2. Division of Work	34
Appendices	34

1. Project Overview

1.1. Description

Concertify is a digital platform that integrates detailed music information with live event discovery, creating a centralized destination for music enthusiasts and eventgoers. Users can access comprehensive music data, including artist discographies, album details, and track information, while also exploring related live events such as concerts and festivals. This integration allows users to follow their favorite artists, view upcoming performance dates, and gain insight into the broader music and live event ecosystem—all within a single, cohesive application.

By uniting music data with live event details, Concertify simplifies the experience for users who want to stay connected with the music they love and keep track of upcoming events in one place. Whether exploring artist information or planning event attendance, Concertify provides a seamless, user-friendly experience that enhances the connection between fans and the world of live music.

1.2. Motivations

Concertify was inspired by the growing demand for a unified platform where users can easily access music information alongside live event details. This integration addresses the challenge of navigating between multiple platforms to keep up with artists, allowing users to discover music and stay informed about concerts and festivals without additional effort. By combining these elements, Concertify simplifies the user experience and offers a streamlined approach to music and event exploration, fostering a more engaged and informed music community.

Concertify's design helps fans stay updated on both music information and live experiences, supporting artists and the live music culture by making it

easier than ever for fans to follow their favorite artists' activities on and off the stage.

1.3. Goals and Objectives

- Create a Unified Experience: Develop a platform that merges music information and event discovery, offering users an efficient, single-source solution for music and live performance exploration.
- Promote Event Engagement: Encourage users to stay informed about concerts, festivals, and other live performances by providing timely updates and reminders for events featuring their favorite artists.
- Support Artist Visibility: Make it easier for fans to track their favorite artists' music and live schedules, helping artists expand their reach and connect more meaningfully with fans.
- Simplify Music and Event Exploration: Streamline the discovery process by offering intelligent recommendations and easy access to both music data and event details, empowering users to explore the music world and live performance scene seamlessly.
- Enhance User Satisfaction: Offer a smooth, cohesive user experience that integrates music and event information, keeping users engaged and making it easier for them to enjoy and connect with the music they love.

2. Product Requirements and Features

2.1. Functional Requirements

The product is considered successfully-developed once it fulfills the following conditions:

- a. The application must retrieve data from two separate third-party API.

- b. The data from the services must be combined in a meaningful way and shown to the user:
- The data visualization should include graphs, plots, maps, or other self-developed advanced visualization. Tables can be used, but they cannot be the only way of showing information. The purpose is to increase the complexity of the design of your application, the visualizations do not have to be especially impressive.
 - The user must be able to adjust the parameters of the visualization.
 - The user must be able to select what data is shown (e.g., customize results). In practice, this also means that the application must be able to show multiple types of data.
- c. The user can save preferences for producing visualizations, and fetching those preferences will produce a visualization using the most recent data with the given parameters.
- d. The design must be such that further data sources (e.g., another third-party API), or additional data from existing sources could be easily added.

2.2. Key Features

- Authorize and save personalized user data (1).
- Display chart of top artists and tracks globally or nationally based on listeners and play counts (2).
- Search artists by name (3).
- Get artists' details (4).
- Display chart of artists' top tracks and albums based on listeners and play counts (5).
- Search events by location (6).
- Search events by artist (7).
- Get events' details (8).

- Display events' map with multiple preferences (9).
- Save/remove favorite artists and events (10).

2.3. Traceability Matrix

Requirements	Features that Satisfy
The application must retrieve data from two separate third-party API.	(3), (4), (6), (7), (8) The data for artists and events are retrieved from multiple APIs.
The data visualization should include graphs, plots, maps, or other self-developed advanced visualization.	(2) (5) (9) The data retrieved are visualized by multiple method, including bar charts, pie charts and maps.
The user must be able to adjust the parameters of the visualization.	(2) (5) User can adjust the parameters for artists' data, based on the search scope or search properties.
The user must be able to select what data is shown (e.g., customize results). In practice, this also means that the application must be able to show multiple types of data.	(2) (5) (9) The statistical data are rich in types.
The user can save preferences for producing visualizations, and fetching those preferences will produce a visualization using the	(1) (9) (10) User can have personalized database with saved preferences.

most recent data with the given parameters.	
The design must be such that further data sources (e.g., another third-party API), or additional data from existing sources could be easily added.	Refer to chapter 3. System Architecture and 4. Design Patterns.

3. System Architecture

3.1. High-Level Architectural Design

Concertify uses Passive MVC as the high-level architectural pattern. In the application, the components' roles are defined as followed:

- **Model:** The Model in Concertify is dedicated solely to managing and organizing the core data for the application, such as event details, artist information, user profiles, and track data. It does not communicate directly with external APIs; instead, it serves as the data layer for storing and structuring information once it is received from the Controller. The Model's responsibility is to retain this data in a way that allows the Controller to access or update it as needed without pushing changes to the View on its own.
- **View:** The View is responsible for the user interface, built using JavaFX to provide interactive elements like forms, buttons, and search fields. It is where users interact with the application, exploring data on artists, events, and tracks. The View's role is strictly presentational, as it relies on the Controller to gather and send it data when user actions trigger changes, keeping the View

focused on displaying information and managing input rather than on data management or business logic.

- **Controller:** The Controller is the central orchestrator, managing user input, the Model, and the View to ensure smooth communication and functionality. When a user performs an action in the View, like searching for an artist or viewing event details, the Controller handles the input, sends requests to the Service layer to retrieve the relevant data, and then processes and updates the Model with this information. Once the Model is updated, the Controller retrieves data from it and directs the View to update the interface accordingly.
- **Service:** The Service layer serves as an intermediary between the Controller and external APIs, managing communication with external resources like Ticketmaster and Last.fm. When the Controller requires data from an API, it delegates the request to the Services, which handle sending the API request and processing the response.

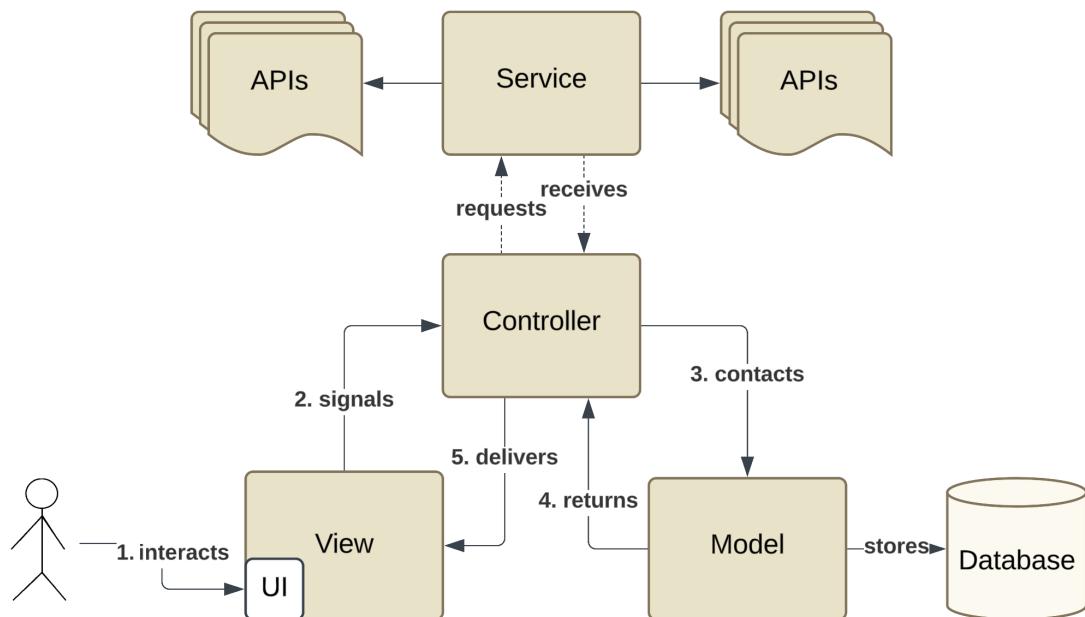


Figure 1. Concertify High-Level MVC Architectural Design

The MVC (Model-View-Controller) pattern is an ideal structure for Concertify because it ensures clear separation between the data management, user interface, and interaction logic, which is essential in an application that depends on external data and complex interactions. By organizing the app into these three distinct components, each part can function independently, making development and maintenance simpler. For example, if Concertify's external APIs change, only the Model and Services layer would need updates while the Controller and View could remain unchanged.

A passive MVC approach works best for Concertify because it keeps the Model simple and the data flow centralized. In this setup, the Controller manages all interactions between the Model and View, providing a structured flow where the Controller explicitly fetches data and then updates the View based on user actions. Since Concertify frequently needs user-driven requests—such as searching for artists or viewing event details—keeping the Controller in control ensures the application provides a responsive, predictable experience. By centralizing API interactions within the Services layer, the application maintains a clean separation of concerns, allowing Controllers to focus on application logic and user interactions.

3.2. Components and Relationships

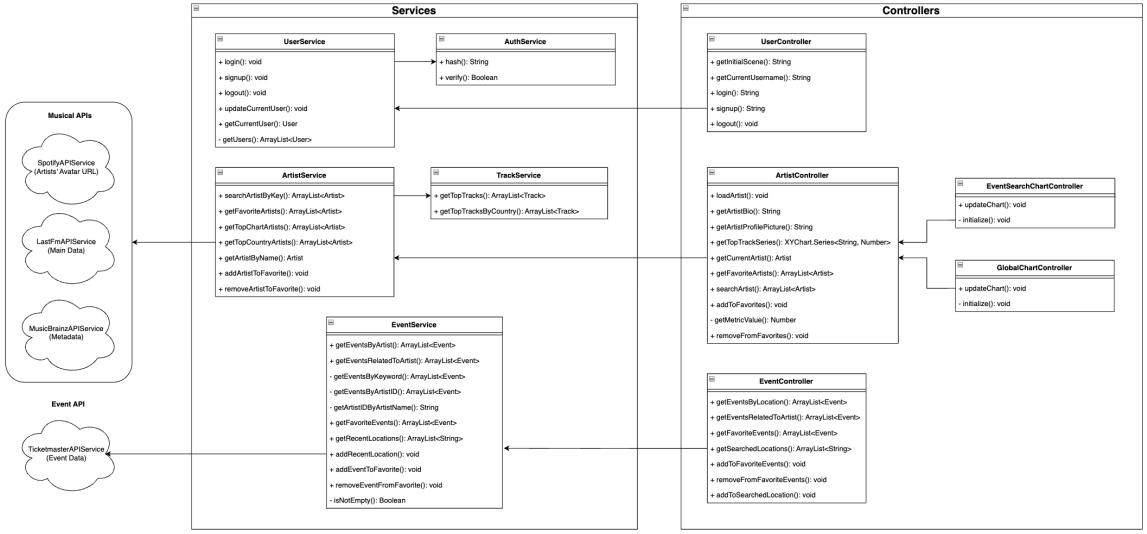


Figure 2. Components and Relationships of Controllers, Services and External APIs Layers

In this architecture, the Services Layer bridges Controllers, Models, and external APIs, managing data retrieval and processing. It includes EventService, UserService, TrackService, and ArtistService, each dedicated to a specific domain. For instance, EventService fetches event details from the Ticketmaster API, while ArtistService aggregates artist data from Spotify, LastFM, and MusicBrainz. This layer abstracts API complexities, allowing Controllers to access data seamlessly and making it easier to update data sources independently.

The Controllers Layer handles user requests by interacting with the Services. This layer includes EventController, UserController, TrackController, and ArtistController, each focused on a functional area. EventController, for example, uses EventService to handle event searches and details, while UserController handles user login, signup, and logout through UserService. Controllers only manage request logic, keeping them independent of data processing and external APIs.

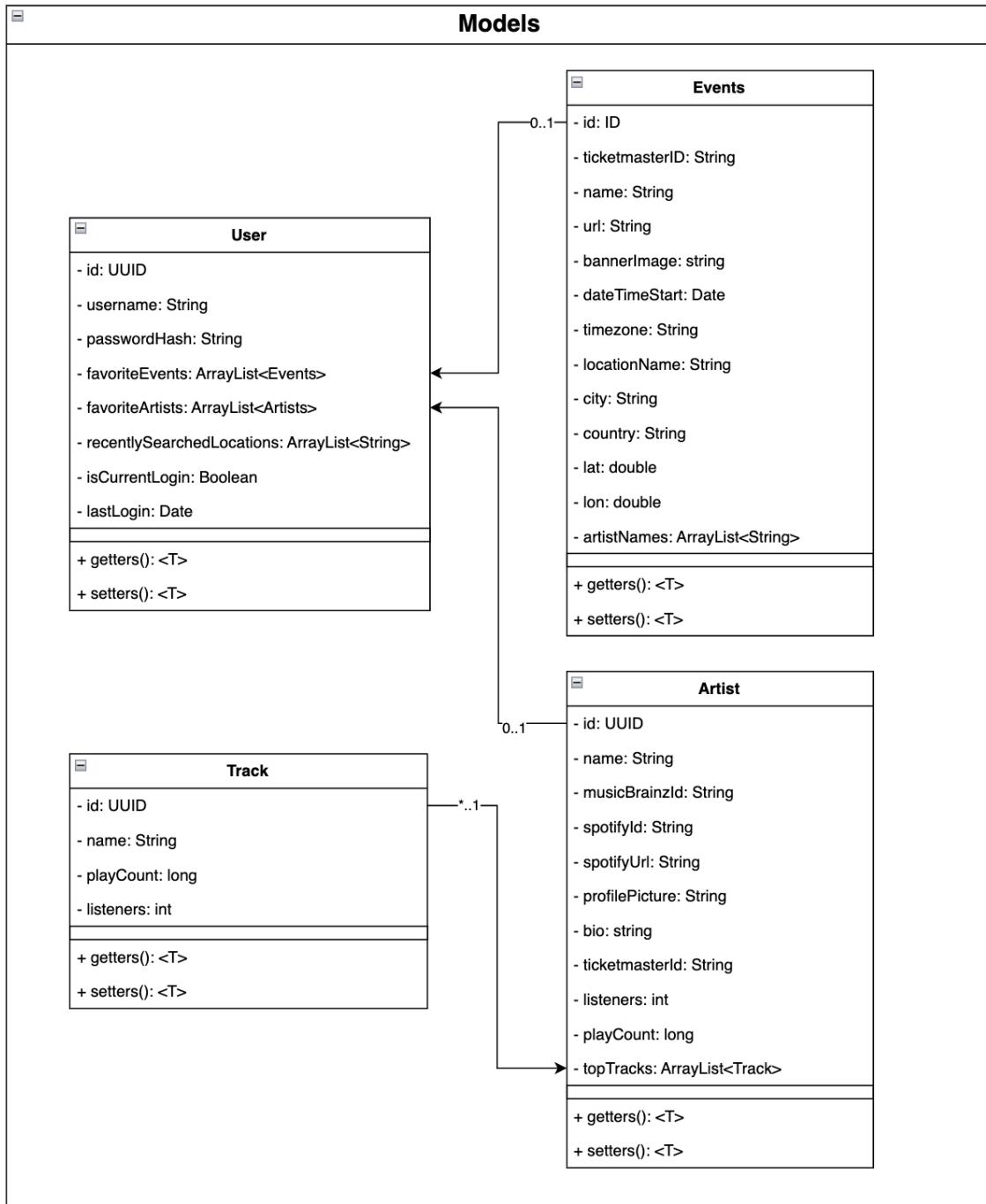


Figure 3. Components and Relationships of Data Models

The Models Layer defines core data entities and relationships, including Event, User, Track, and Artist. Each model represents a main data point, with fields specific to its role. Event includes attributes like name and

location, while User holds information such as username and favorites. Track captures song popularity, and Artist contains details like name, bio, and top tracks. This layer centralizes data structure, ensuring consistency and integrity across the application.

3.3. Database and Caching System

While the project instructions do not mandate the use of a database like PostgreSQL, the application will emulate database-like functionalities to support caching and efficient data handling. However, as the scope of the application is small, therefore it does not require a complex, traditional database system; instead, data will be stored in .json files within the codebase.

An essential feature of the application is its ability to integrate with multiple external APIs to process and utilize the provided data effectively; however, since most of these APIs enforce strict rate limits, the application's requirement for handling bulk requests could potentially result in an HTTP 429 "Too Many Requests" error, which not only risks causing significant disruptions to the user experience but could also, in more severe cases, lead to the invalidation of the provided API key. Because of this, a caching system is introduced in certain situations that can save query results and reuse up-to-date data without having to call for the API a second time. Consider this example, where the user does a search request to the music-based API, which search for a list of artists based on a search term, however, the list of artists returned contains only scarce data (for example, ID and name only), where the application require a rich set of information, such as profile picture, or artist's country of origin... The API does provide another endpoint to retrieve this information, however, it means that now the application must make $n+1$ calls to the API, with the first one being the initial search request, and for every artist returned, a

request to get the detailed information is sent, therefore creating a heavy load of requests to the API that may possibly exceed the limit rate.

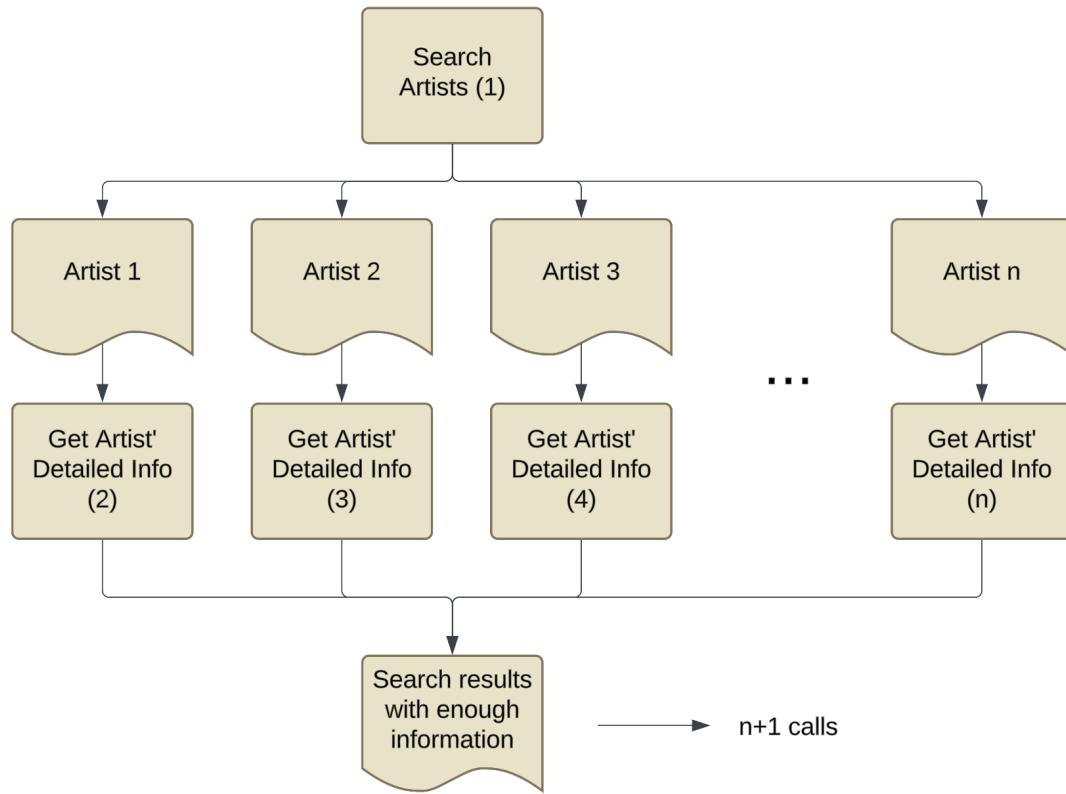


Figure 4. Search Artist functionality without caching system

However, if a caching system is introduced, it will reduce the number of calls to the API to a great extent. Whenever the application does a search query, it will check first hand if the detailed information of an artist has already been stored in the cached database; if it has, then the information can be retrieved directly from the database, otherwise it will send a GET request normally to the API, however, the information once retrieved will be instantly cached for next usage, if the search term is ever queried again. With this approach, although in the worst-case scenario, where no search results have been cached, the number of queries is still $n+1$, with n being the number of results, however as the application is used more intensively, the number of cached results in the database will also increase, thus

greatly reduce the possible calls to retrieve single instance's full information.

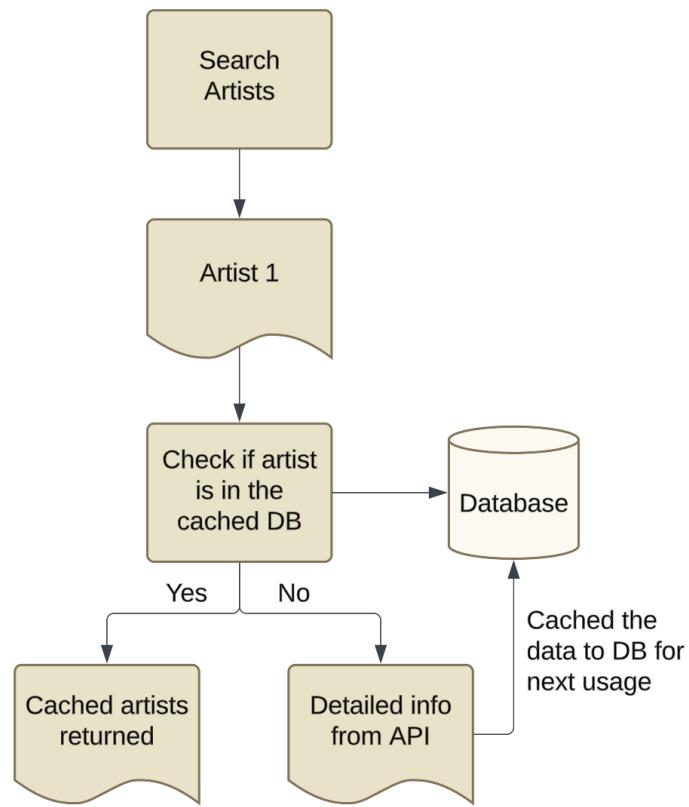


Figure 5. Search Artist functionality with caching system

3.4. Project Modules and Directory Structure

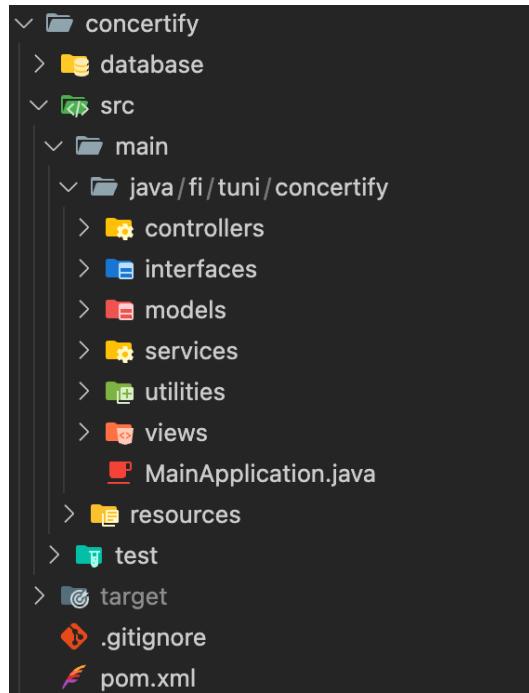


Figure 6. Directory Structure

4. Design Patterns, Interfaces, and SOLID Principles

4.1. Factory Method Design Pattern

The Factory Method is a creational design pattern that offers an interface for object creation within a superclass while enabling subclasses to modify the types of objects that can be instantiated. In the context of a frontend application, the design pattern serves as a centralized high-level manager for all the components and sub-components that comprise the main application.

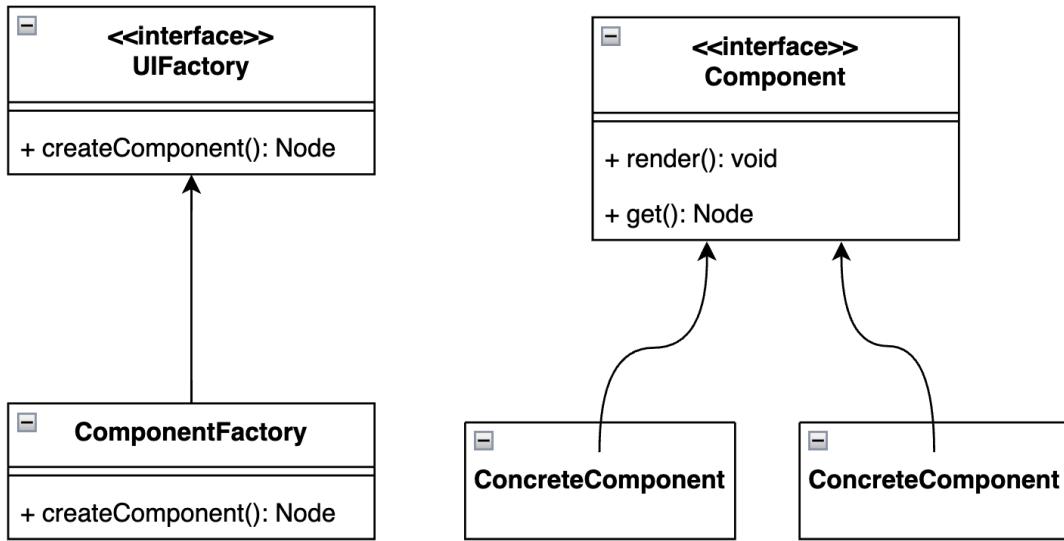


Figure 7. ComponentFactory that follows the Factory Method design pattern

In the frontend development approach, the goal is to replicate the functionality of a React application, where every element is treated as a component. This design includes a structured system for rendering, re-rendering, and data retrieval. To facilitate this, an interface called Component is established to define the essential characteristics and behaviors of individual concrete components. All components are orchestrated by a centralized ComponentFactory, which implements the UIFactory interface, specifically focusing on the createComponent method. The primary responsibility of the ComponentFactory is to initialize and manage the components necessary for the frontend application. This setup allows for efficient component lifecycle management, ensuring that each component can be rendered or updated as needed while maintaining a consistent state throughout the application.

By utilizing this architecture, the application benefits from modularity, making it easier to develop, test, and maintain individual components. Each component can encapsulate its own logic and styling, promoting

reusability and scalability within the application. The centralized factory not only streamlines component creation but also enables future enhancements, such as integrating new types of components or altering existing ones without disrupting the overall system. This approach ultimately leads to a more organized and efficient development process, similar to that found in modern React applications.



```
● ● ●
1 public interface UIFactory {
2     Component createAuthView();
3
4     Component createAuthForm(Node root);
5
6     Component createAuthImage(Node root);
7
8     Component createLogin(EventHandler<MouseEvent> switchToSignUp);
9
10    Component createSignUp(EventHandler<MouseEvent> switchToLogin);
11
12    Component createHome();
13
14    Component createContent(Node root);
15
16    Component createHeader(Node root);
17
18    Component createBody(Node root);
19
20    Component createNavBar(Node root);
21
22    Component createSearchBar();
23
24    Component createArtistsSongsChart();
25
26    Component createSingleArtistHeading(Node root);
27
28    Component createHeadingBox(Node root, String headingText);
29
30    Component createArtistBody(Node root);
31
32    Component createArtistList(Node root, String listType);
33
34    Component createEventSearch();
35
36    Component createEventList(Node root, String listType);
37
38    Component createEventSearchChart();
39
40    Component createEventMap(Node root);
41 }
```

Figure 8. Usage of UIFactory interface complies with the Factory Method design pattern

4.2. Composite Design Pattern

The Composite pattern is a structural design pattern that allows a construct of tree-like structures of objects, and treat these structures as though they were single objects.

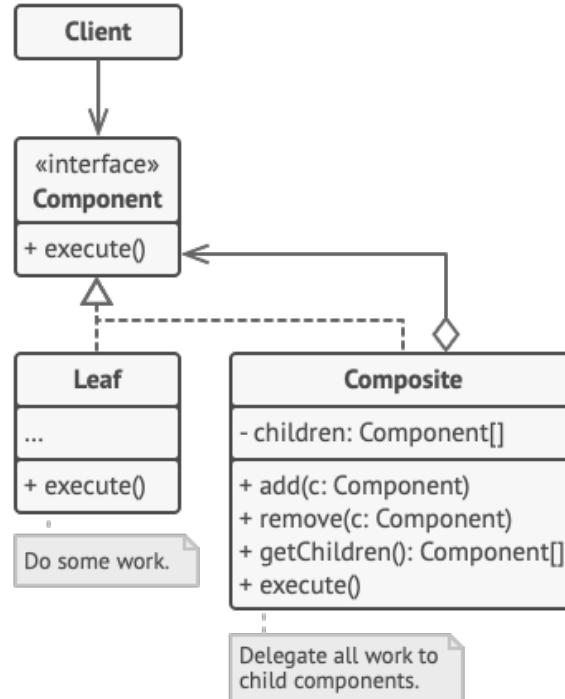


Figure 9. General structure of a Composite design pattern

The Composite pattern is well-suited here because each JavaFX component in Concertify can be treated as a node in a Document Object Model (DOM) in a typical HTML structure, where every component may consist of smaller sub-components, down to the smallest building blocks or leaf nodes.

Concertify uses the Composite pattern to allow complex components to be assembled from these smaller parts, creating a hierarchical structure where each component can be composed of other components. For example, a ConcertView component might include multiple ArtistInfo, TrackList, and EventDetails components. Each of these sub-components, in turn, could be broken down further, such as ArtistInfo containing elements like ArtistImage and ArtistBio. This approach enables a recursive

structure, where each component can have its own child components, which JavaFX supports through methods like `getChildren()` for managing these nested relationships.

One significant advantage of using the Composite pattern in Concertify is the efficient propagation of updates. When a child component updates, this change can affect the parent components and ultimately the entire UI structure, maintaining consistency throughout the application. For example, if a user updates the concert date within an `EventDetails` component, this modification would automatically reflect in all related components displaying this information. Through this Composite structure, Concertify achieves modularity, allowing individual components to be developed, updated, and tested independently.

```
Component.java:
1 public interface Component {
2     void render();
3     Node get();
4 }
5

Home.java:
1 public class Home implements Component {
2     private HBox home = new HBox();
3     private ComponentFactory factory;
4
5     public Home(ComponentFactory factory) {
6         this.factory = factory;
7     }
8
9     @Override
10    public void render() {
11        home.getChildren().clear();
12        Component navBar = factory.createNavBar(home);
13        home.getChildren().add(navBar.get());
14    }
15
16    @Override
17    public HBox get() {
18        render();
19        return home;
20    }
21 }
```

Figure 10. Usage of Component interface complies with the Composite design pattern

4.3. SOLID Principles

The application aims to comply with the SOLID principles of object-oriented design. However, it should be noted that due to the simplified

nature of the projects, as well as the time constraints, these principles are not strictly followed.

Principles	Definition	Evaluation
Single Responsibility Principle (SRP)	A class should have only one reason to change, meaning it should have only one job or responsibility.	Most classes have a single responsibility, although some service classes could be further split if they grow too large. For example, UserService class handles user-related operations such as fetching users, updating the current user, signing up, logging in, and logging out. While it does a lot, these responsibilities are all related to user management, so it generally adheres to SRP.
Open/Closed Principle (OCP)	Software entities should be open for extension but closed for modification.	The design allows for extension without modifying existing code. For example, ComponentFactory class can be extended to create new components without modifying existing code. Controllers and Services can be extended to add new functionalities without modifying existing methods.

Liskov Substitution Principle (LSP)	<p>Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.</p>	<p>The use of interfaces ensures that subclasses can be used interchangeably with their superclasses. For example, classes like Login, SignUp, and AuthForm implement the Component interface, which allows them to be used interchangeably where a Component is expected. The ComponentFactory class implements the UIFactory interface, ensuring that it can be replaced with any other class that implements UIFactory.</p>
Interface Segregation Principle (ISP)	<p>Clients should not be forced to depend on interfaces they do not use.</p>	<p>Interfaces are well-segregated, ensuring that clients only depend on what they need. For example, UIFactory interface is well-segregated, providing specific methods for creating different UI components. Clients can implement only the methods they need.</p>

Dependency Inversion Principle (DIP)	High-level modules should not depend on low-level modules.	Both should depend on abstraction. There is some adherence to DIP through the use of interfaces and constructor injection, though more explicit dependency injection could improve this further. The codebase does not show explicit use of dependency injection frameworks, but it does use constructor injection in some places (e.g., ComponentFactory), which is a good practice. The use of interfaces like Component and UIFactory helps in adhering to DIP by depending on abstractions rather than concrete implementations.
--------------------------------------	--	--

5. APIs, External Libraries and Dependencies

5.1. APIs

API	Purposes	Features
Last.fm API	Primary data source for music-related statistics and information.	Get artist information and statistics for search functionality.

		Get global and country-specific top artists and tracks sorted by listeners and playcounts.
MusicBrainz API	Bridge service between Last.fm and Spotify. Ensure accurate artist matching across platforms	Identify the artist for search functionality. Extract Spotify ID of the artist to make sure of consistent artist identification.
Spotify API	Provide profile pictures for artists.	Get artist URL and avatar from Spotify.
Ticketmaster API	Provide event details such as venue information, date and time, ticket price.	Get concert details related to the artist. Get events based on location.

5.2. External Libraries and Dependencies

Concertify utilizes the following external libraries and dependencies to enhance functionality and streamline development:

- Maven: A powerful build automation tool and dependency manager, Maven simplifies the project's build process, manages library dependencies, and ensures consistency across development environments.

- JavaFX: JavaFX provides the foundation for Concertify's graphical user interface, offering tools for building rich desktop applications with a modern, responsive UI, including interactive elements and multimedia support.
- Google Gson: Gson is used for parsing and handling JSON data, allowing Concertify to efficiently convert JSON data to Java objects and vice versa. This is essential for managing external data, such as music and event information.
- Password4j: Password4j provides secure password hashing and encryption methods, supporting secure user authentication within Concertify. It ensures that user credentials are stored and handled safely.
- Leaflet: Leaflet is a lightweight open-source JavaScript library used for creating interactive maps. It supports features like zooming, panning, markers, popups, and layers, integrates easily with various map tile providers and is highly customizable through plugins and extensions.

6. User Interfaces

6.1. Designed Interfaces

The designed interfaces are described in the Figma prototyping tool, refer to the appendices. The interfaces consist of:

- Authentication Page, including Login and Sign-Up panels.
- Main Application Page, including:
 - Home Page, displaying the map of personalized events (based on favorite events, favorite artists' events and searched locations).

- Artists-related pages, with functionalities being searching for artists, displaying artists' details and statistical charts, and displaying favorite artists.
- Events-related pages, with functionalities being searching for events based on location, displaying events' details, maps and favorite events.

6.2. Implemented Interfaces

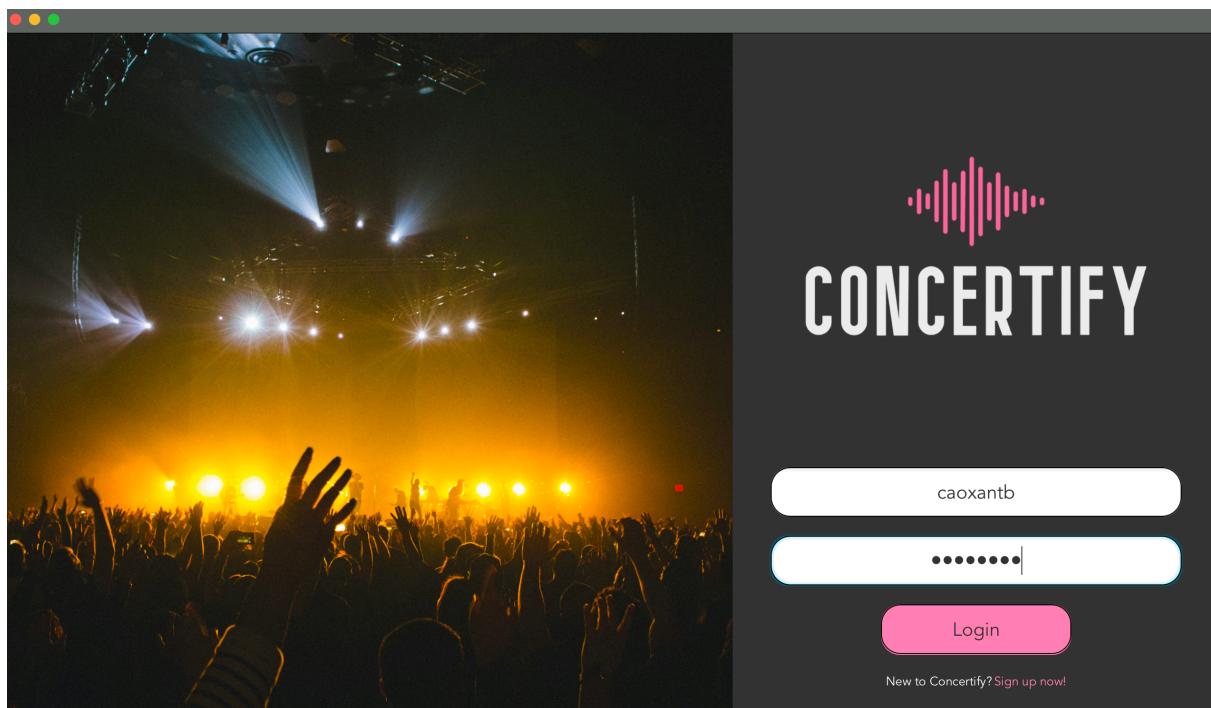


Figure 11. Implemented Authentication Page

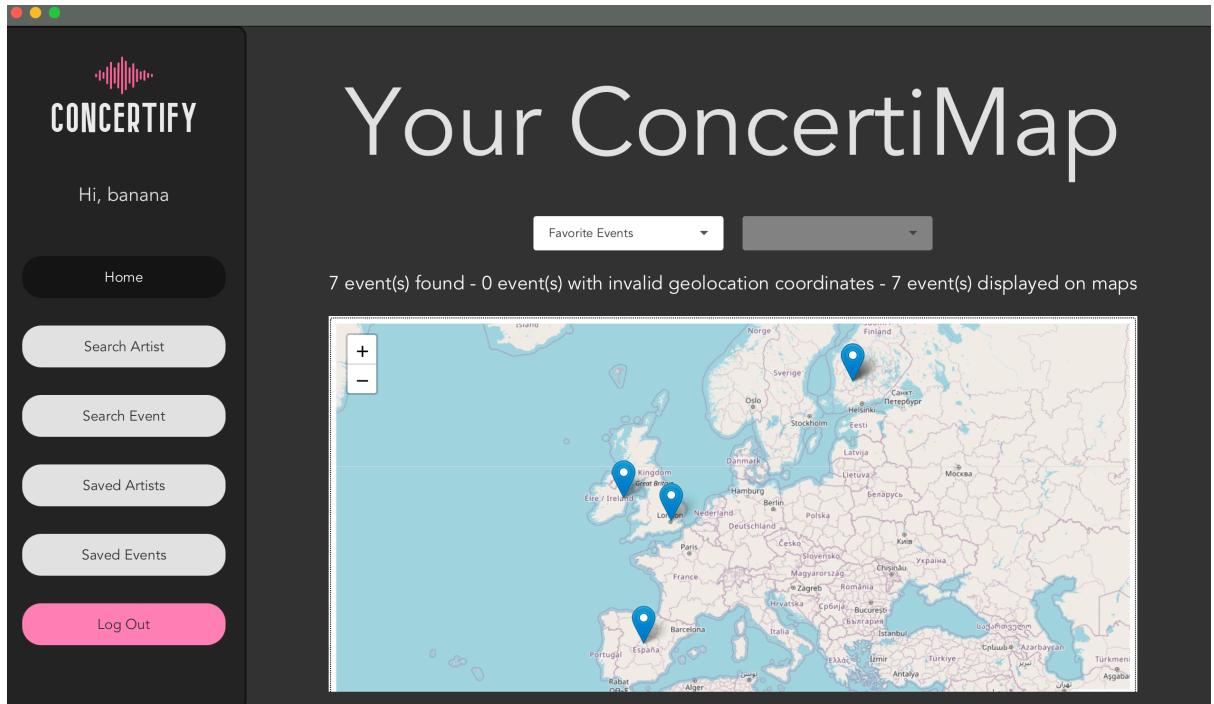


Figure 12. Implemented Home Page with Event Map Visualization

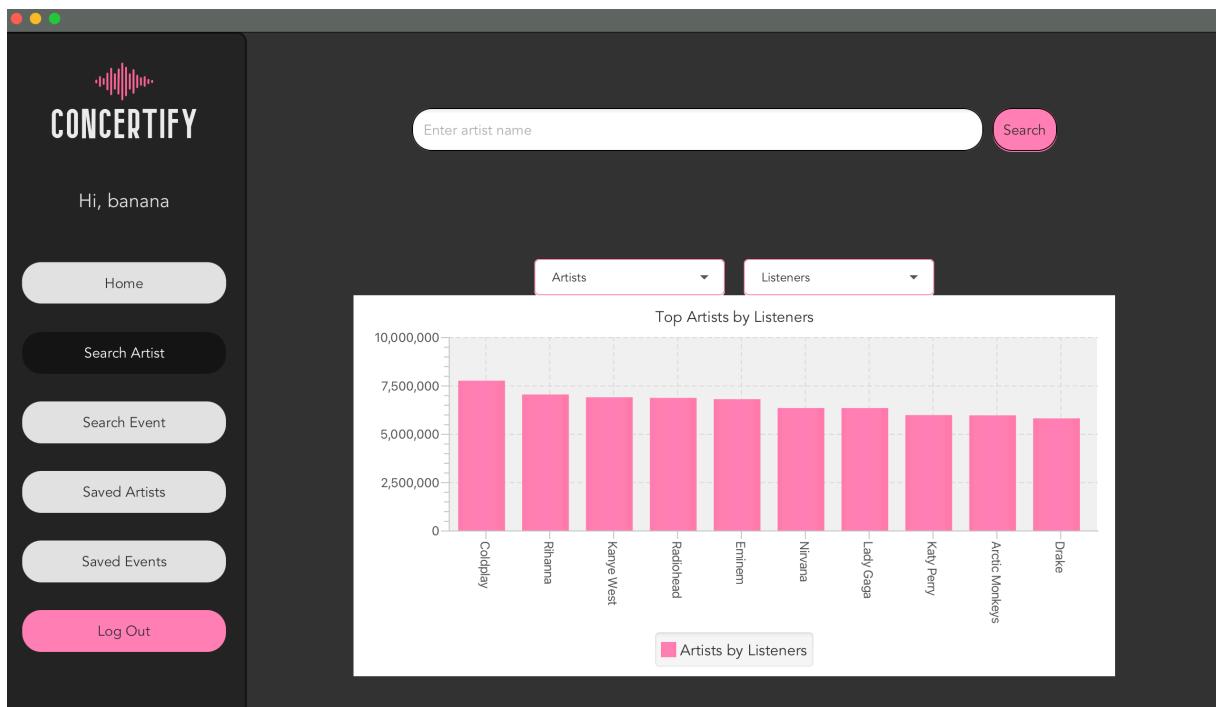


Figure 13. Implemented Search Artist Page with Bar Chart Visualization

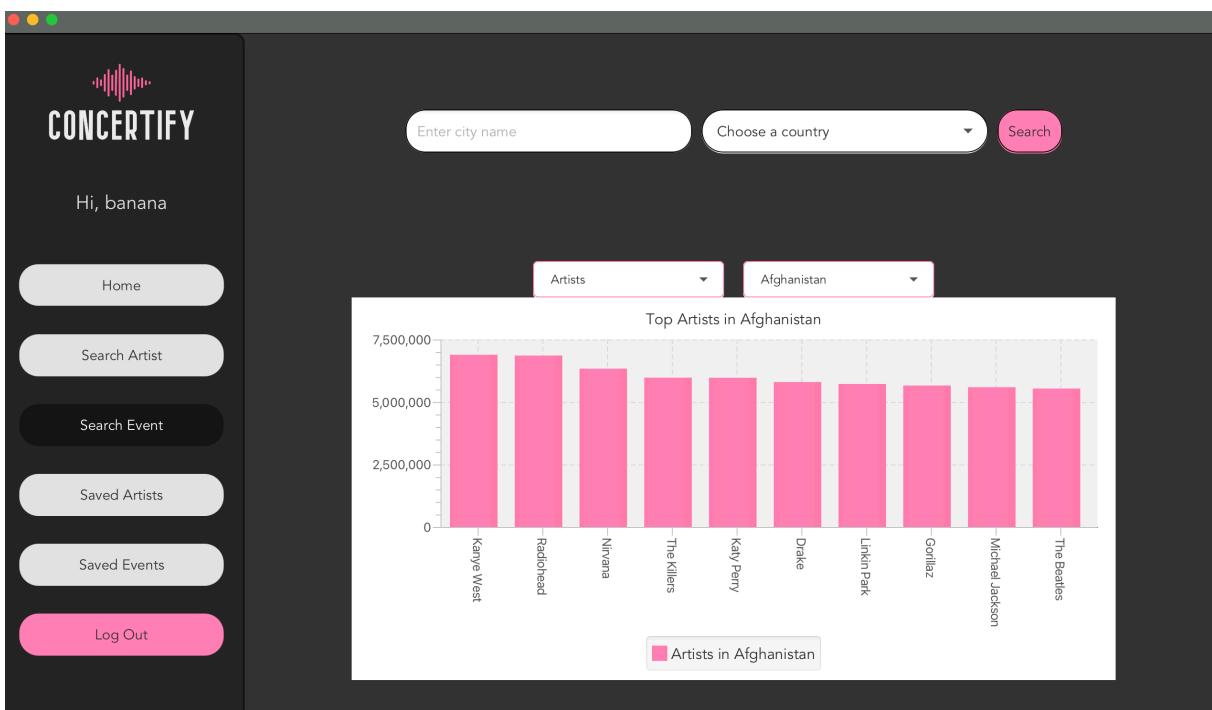


Figure 14. Implemented Search Event Page with Bar Chart Visualization

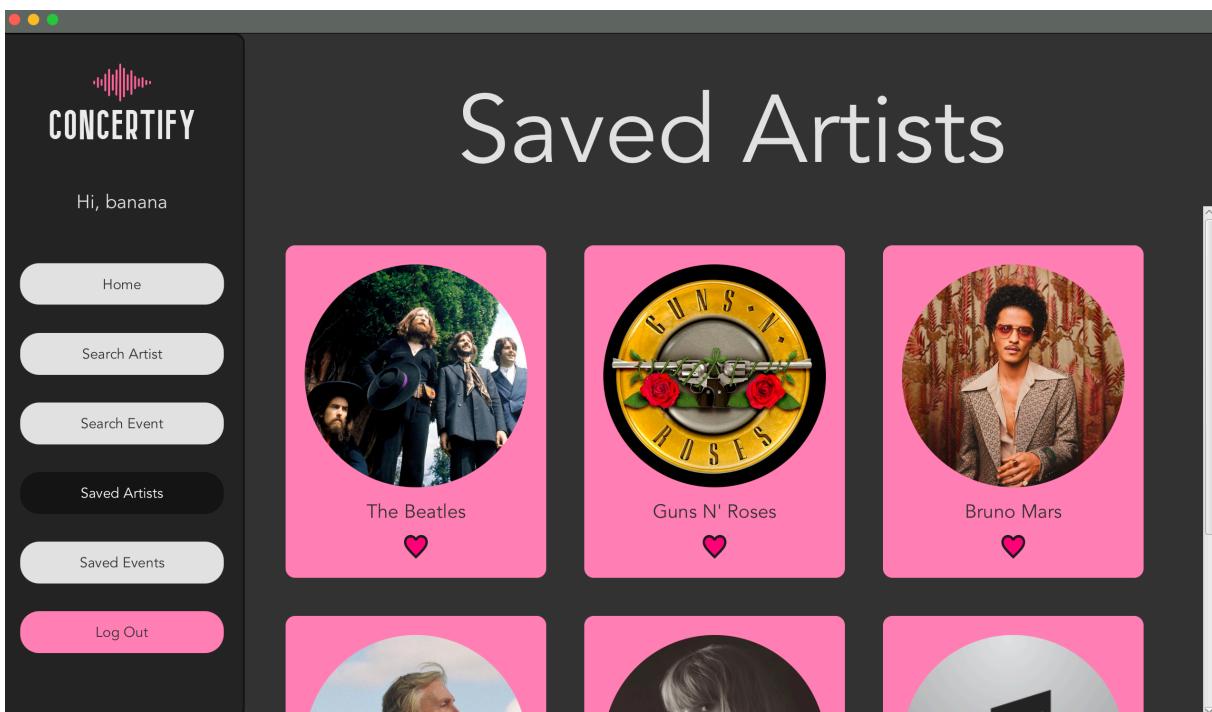


Figure 15. Implemented Saved Artists Page

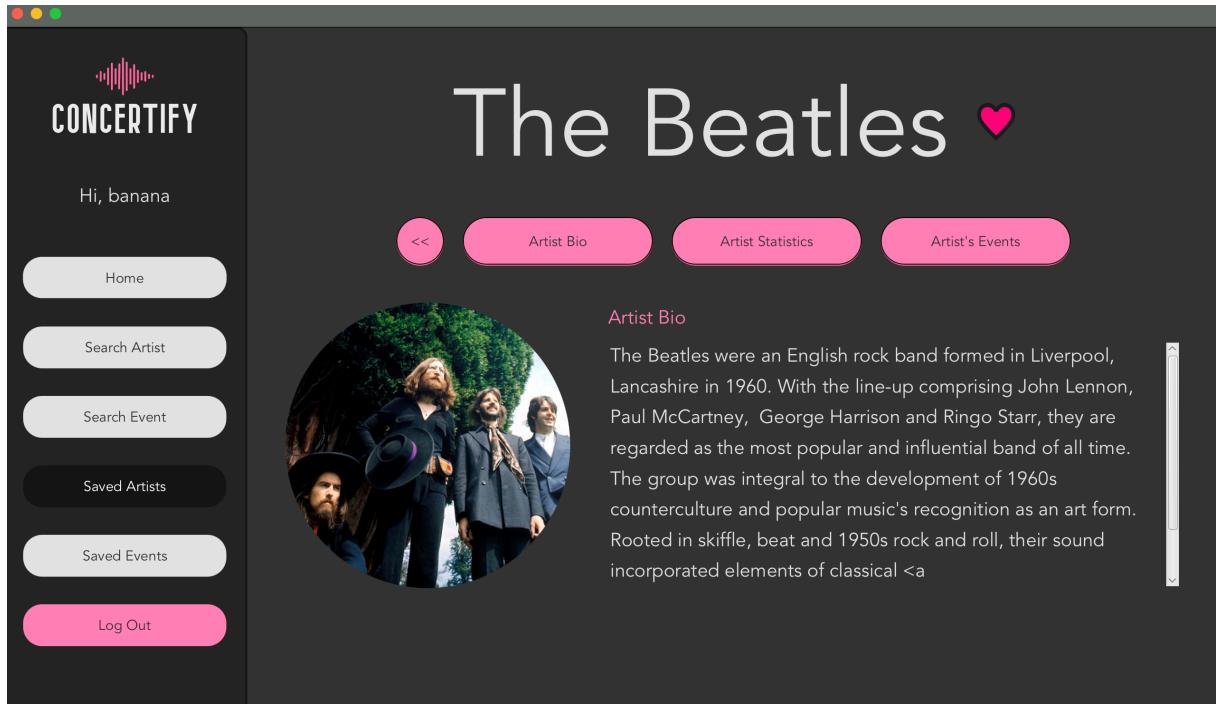


Figure 16. Implemented Single Artist Page

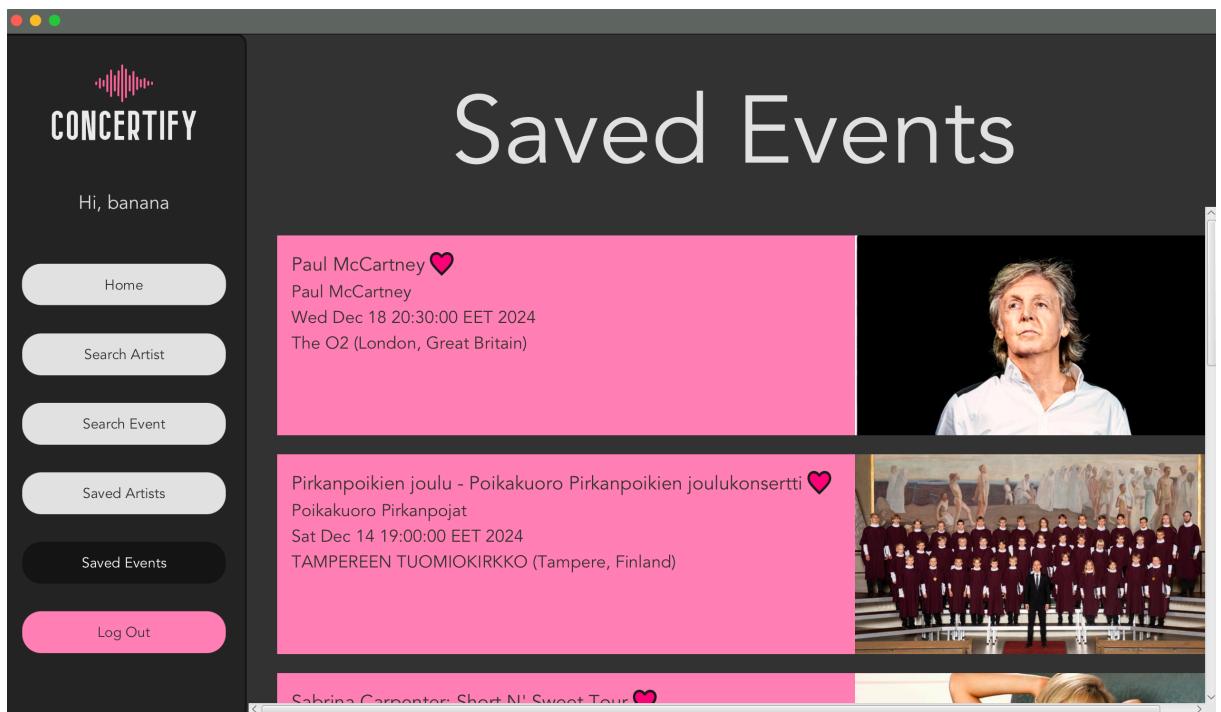


Figure 17. Partially Implemented Saved Events Page

7. Usage of Artificial Intelligence

Concertify incorporates limited reliance on LLM-based (Large Language Model) AI tools, integrating these resources in select areas to enhance specific functionalities. The exact tools used and their applications will be adapted and documented in line with the university and course guidelines on AI usage, ensuring compliance and transparency. This approach allows Concertify to leverage the capabilities of AI responsibly, enhancing aspects such as data processing, user interaction, or automated support, while adhering to academic standards and ethical considerations.

- Project idea: The project idea was generated by OpenAI's ChatGPT and developed by our team.
- Application logo: The product logo was designed using the free version of the AI-enhanced myfreelogomaker.com logo generator.
- Documentation: Enhancement on the style of writing and paraphrase of words and sentences of the documentations for this project are supported with ChatGPT and Grammarly. It is important to note that these AI tools function solely as refiners, not as the original writers and developers.
- Implementation: Partial usage of Copilot AI integrated in Visual Studio Code for commenting, refactoring and completing code snippets are traceable. The AI tools are not used whatsoever to generate fully functioning codes for any parts or subparts of the application.

8. Final Self-evaluation

8.1. Evaluation

Evaluation Criteria	Justification for Evaluation
Changes to the original design	<ul style="list-style-type: none">• During the implementation phase, we found a way to connect last.fm API and Spotify API to retrieve artists' profile

	<p>pictures using another API called MusicBrainz. On the prototype phase, we were planning to use only last.fm API despite the drawback that it does not provide any pictures.</p> <ul style="list-style-type: none"> • We also changed the two graphs in artist search view showing global music statistics into one graph with dropdown boxes to adjust the graph based on different identity types and metrics. • We decided not to implement a separate page for single event, since most of the data is already displayed in the event list.
Implementation progress based on the original plan	<ul style="list-style-type: none"> • By the mid-term submission, we have completely designed our application called Concertify. We have implemented the user sign-up/log-in's user interface and functionalities, along with the user interface of artist search and artist profile view. The integration of 3 APIs providing artist information has also been implemented. • By the final submission, all the required functionalities of the application have been implemented. 4 APIs (3 music-based and 1 event-based) have been used for data retrieval. Visualizations are also done with multiple bar charts and maps.

	<ul style="list-style-type: none"> • We use JavaFX for user interface implementation and Model-View-Controller (MVC) architecture as indicated in the original plan.
Design and quality correspondence	<ul style="list-style-type: none"> • The application uses Factory Method and Composite design patterns to ensure modularity, maintainability, and scalability. The Factory Method centralizes component creation, allowing new components to be easily integrated without disrupting existing structure, while the Composite pattern organizes UI elements hierarchically, enabling independent development and testing of each component. • The application has a React-like architecture, which promotes a consistent, high-quality user experience and prepares the application for future expansion.
Potential bugs and possible improvements	<ul style="list-style-type: none"> • The implementation can be considered a finish application; however, there are known bugs that have not been fixed, due to either time constraints, or the extreme rare occurrences of these bugs. • One problem may occur is the 429 Too Many Requests error which may happen when querying from Ticketmaster API.

	<p>This is because the Ticketmaster API has a strict limit rate (5 requests per second only). The error has been fixed partially so it does not crash the application.</p> <ul style="list-style-type: none"> • The application can be further improved with more extensive visualizations, using more data retrieved from MusicBrainz API.
--	--

8.2. Division of Work

- Xuan An Cao: Structural concept and design patterns implementation (Component Factory and Composite), API Services, User Service, Auth Service, Map Visualization and Documentation.
- Minh Khanh Pham: Artists View, Artists Controllers, Bar Chart Visualization, and Styling.
- Gimantha Dissanayake: Event Service, Event Controller, and Unit Testing.
- Dávid Lakos: Event List, Artist List and Documentation.
- Erfan Niketeghad: Event Model and Unit Testing.

Appendices

Appendix A. Figma prototype of the wireframing of the application:
<https://www.figma.com/design/iRqrjmGgMKLah1r4cwkG8N/SW-Design-Project?node-id=103-638&node-type=frame&t=qnVyKFjQtpvIDoZb-0>.

Appendix B. Traceable ChatGPT conversation on the development of the product idea: <https://chatgpt.com/share/671e670f-e770-8000-9932-dceb1e2a5898>.