Jammin Design Document

MoSCoW

Mo

1. Non-negotiable
2. Minimum viable product
3. Unable to deliver the end product without this
4. Not legal with it
5. Unsafe without it
6. Without this project is not viable

S

1. Important but not vital
2. Maybe painful to leave out but the solution is still viable
3. May need some kind of workaround

Co

1. Desirable but not as important as Should Have
2. Only do if there is extra time and budget

W

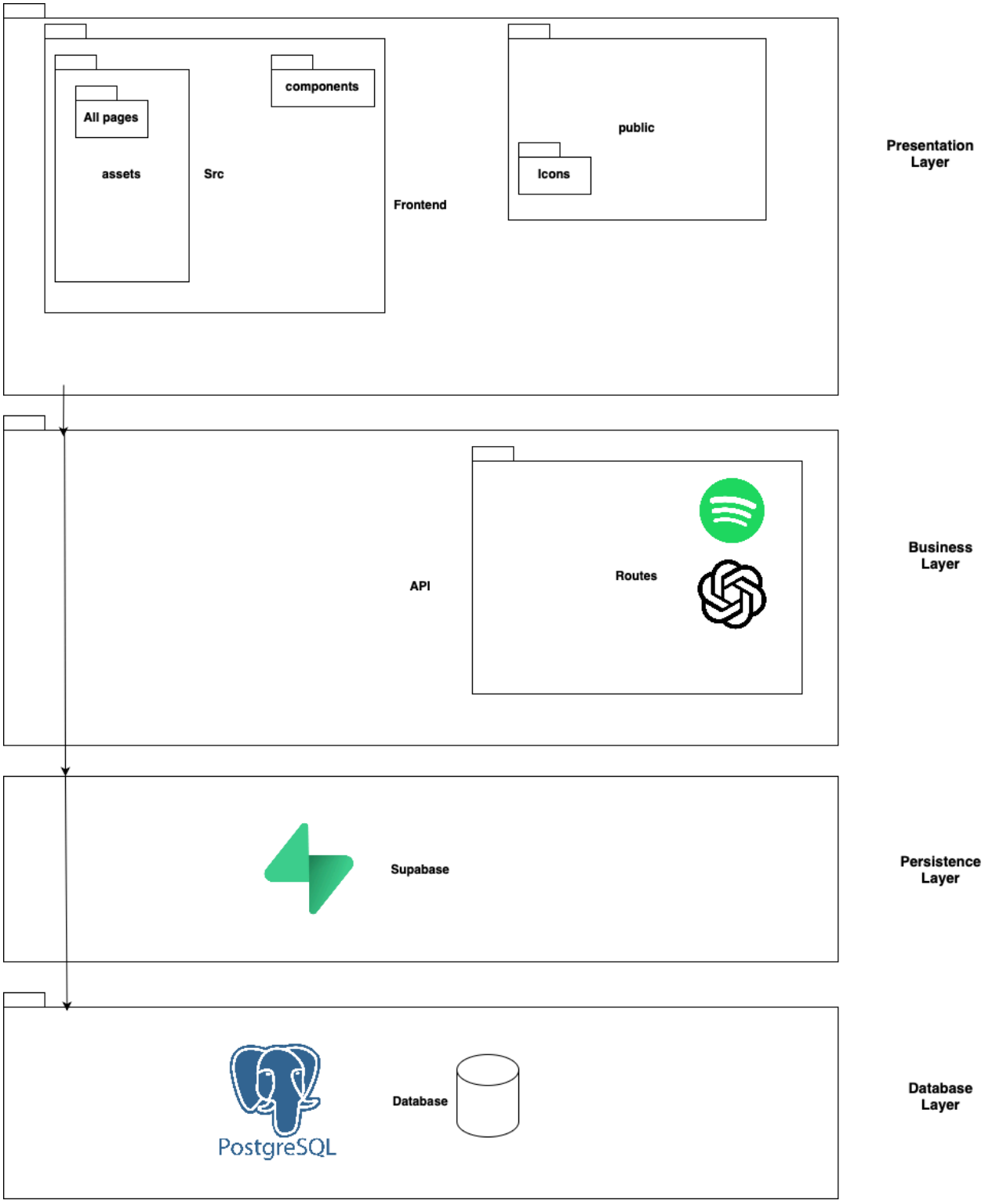
1. Won't have this time around at all
2. Out of budget
3. Nice to have but has no real impact

Assignment	Type	Status	Priority Level	Finish By	Notes
Design Document	Other	Finished	Must	Mar 21, 2025	
Decide on TechStack	Archit...	Finished	Must	Jan 24, 2025	
Initialize Project (Github repo)	Archit...	Finished	Must	Feb 1, 2025	
Create Jira Project Board	Archit...	Finished	Must	Feb 1, 2025	+Share with everyone
Design Visual Drafts (pages, layout, theme)	Fronte...	Finished	Sho...	Feb 4, 2025	Non-committal, but this also needs to consider, how do I possibly implement this. I.e. If you want to implement swiping, find a video tutorial.
Decide on	Datab...	Finished	Sho...	Feb 4, 2025	Hard to describe but this is

Assignment	Type	Status	Priority Level	Finish By	Notes
architectural and conceptual approach to database connection/maintainability					effectively brainstorming that should reach a conclusion
Early research on LLM implementation	Backe...	Finished	Must	Feb 4, 2025	
Early research on Spotify API integration	Backe...	Finished	Must	Feb 4, 2025	
Early research on hosting/server updates	Functi...	Finished	Must	Feb 4, 2025	
Give consideration to and decide on flow	Fronte...	Finished	Sho...	Feb 4, 2025	
Homepage	Fronte...	Finished	Must	Feb 18, 2025	
Login/Signup page	Fronte...	Finished	Must	Feb 18, 2025	
Dashboard	Fronte...	Finished	Sho...	Feb 25, 2025	
Matching UI	Fronte...	Finished	Must	Feb 25, 2025	
Matches UI	Fronte...	Finished	Sho...	Feb 25, 2025	

Assignment	Type	Status	Priority Level	Finish By	Notes
Messages UI	Fronte...	Not start...	Could	Feb 25, 2025	
Settings/Profile Page	Fronte...	Finished	Sho...	Feb 25, 2025	
Instantiate MySQL database	Datab...	Finished	Must	Feb 18, 2025	
Database retrieval	Datab...	Finished	Must	Feb 18, 2025	This includes functions to retrieve from the database and update the database
Parse database receipt	Datab...	Finished	Must	Feb 18, 2025	
Database insertion	Datab...	Finished	Must	Feb 18, 2025	
Retrieval from Spotify API	Backe...	Finished	Must	Feb 25, 2025	
Parse receipt from Spotify API call(s)	Backe...	Finished	Must	Feb 25, 2025	
Spotify API authentication	Backe...	Finished	Must	Feb 25, 2025	
LLM implementation + entry function(s)	Backe...	Finished	Must	Feb 25, 2025	
Matching	Functi...	Finished	Sho...	Mar 4, 2025	

Assignment	Type	Status	Priority Level	Finish By	Notes
algorithm/method implementation					
Process LLM results and configure display accordingly	Fronte…	Finished	Sho…	Mar 4, 2025	
Play potential matches song on the page as you view them	Functi…	Not start…	Could	Mar 4, 2025	
Understand Deployment	Datab…	Finished	Must	Mar 4, 2025	
Deploy (V1)	Datab…	Finished	Must	Mar 14, 2025	
SUBMIT	Other	Finished	Must	Mar 21, 2025	Should already be deployed, at this point it has been polished to the max


All pages
components
assets
Src
Frontend
public
Icons
Presentation Layer
API
Routes
Business Layer
Supabase
Persistence Layer
PostgreSQL
Database
Database Layer

Overall Structure

- **Presentation layer**
    - This layer holds all of the user functionality and user interface such as the pages, user directions, buttons and the core matching interface. This layer was organized by having a frontend folder which has two main sections, a public section where all globally accessible icons are available, and a src section for holder components and pages. The components and pages are organized by page, with unique related components being stored in the same folder as the page. Whereas more generic components like buttons and input fields were stored elsewhere for more general access. The structure of each front end page has one component that encapsulates the entire page formed of smaller components customized with Tailwind CSS, and organized using HTML. Each of these pages has all functionality it needs for interacting with the database and Spotify handled through connecting functions that pass data to external layers. Responses are then received and the necessary information that should be displayed to the user is formatted into the UI. Finally, each

- **Business layer:**
    - The business layer holds the functionality for creating requests to be sent to the database and controlling the overall functionality of the application. These requests can be formatted by accessing routes specifically designed for purposes such as retrieving a user's Spotify data, posting a new user to the database, verifying users, storing / updating user information and more. This layer provides all the functionality that the presentation layer depends on, allowing users to login and match with others.

- **Persistence Layer:**
    - The persistence layer acts as a bridge between the requests made by the business layer and database itself. Using Supabase, these requests can be parsed and the proper actions requested can be undertaken, with data being sent where it needs to go.

- **Database Layer:**
    - Lastly, the database layer is a hosted SQL database of user information, and user music data. Any users that create an account with our program have their information stored here for future use such as displaying it to other users attempting to match with them. The database alongside the rest of the application is hosted using Vercel such that it remains active even when users are not actively using the program, awaiting requests to be sent to it from the persistence layer.

Use of Design Patterns

- **Singleton:** In our application, we implemented the Singleton design pattern for the database connection. By using a singleton we ensure that there is only one instance of

the database connection throughout the entire application lifecycle. This guarantees that all queries are handled through this single instance, which reduces the overhead that would have occurred by creating multiple connections. This also creates maintainability as all requests must pass through a consistent point in the program, meaning any changes to requests will be realized across the entire program.

- **Facade:** Both our front end and back end present themselves as facades for more complex systems. The front end, or presentation layer, takes simplistic user inputs such as button presses and input fields, and converts them into complex LLM requests, API calls and more, to finally match our users together. Furthermore, the access to the backend, or business layer, is simplified greatly within the code to simplistic route calls with some information passed alongside it. This way any code on the frontend can easily integrate with the functionality of the backend, and the users can have a good useability throughout the program.

- **Observer:** Because the active (logged-in) user requires access to their personal information, as well as other user's information on different pages in a single session, we have use 'context' which is a React hook that can pass information and update information between multiple pages without excessive prop-drilling or database queries. While this does not implement interfaces or communicate between 'classes' as a react project doesn't utilize these, however, the functionality of React Context to notify other pages and components of state changes makes it operate just like an observer interface between classes. An instance of how this is used in our app is when you log-in/sign-up, the user data associated with you in the database is temporarily stored in the frontend using context, so that throughout pages, the information you perceive is specific to you.

Use of SOLID principles:

Single responsibility:

In our **gpt.py** we use single responsibility by having two methods: **run_chatquery** and **insert_response**. The first handles sending the query to ChatGPT and retrieving the response. The second manages storing the query and response in the database. By separating these concerns, the code becomes more maintainable. Any changes interacting with ChatGPT can be made in run_chatquery, and any changes having to do with inserting into the database can be made in insert_response.

In our **spotify.py** we use single responsibility by having separate functions for getting data from the spotify api and verifying the call back link after returning. Similar to above, this improves maintainability by separating concerns.

In our **users.py** and **user_data.py** files, we organize functionality by using separate functions for updating, deleting, adding, and retrieving data. Each file is dedicated to a specific aspect of

user data management, ensuring that each function has a clear, distinct purpose. By following this structure, we improve **maintainability**, making it easier to modify, debug, and extend the code without affecting unrelated components.

Moreover, on the frontend, few jsx files exceed 30 lines because we aimed to make our code modular to separate concerns and allow individually modifiable components. With the exception of some files with responsibilities that are difficult to separate or do redundant lengthy tasks, like creating the Jammin profile form, our frontend code base heavily relies on the component based nature of React.

Open-Closed:

The open-closed principle was exercised in our project by using react to create user interface components. This allowed us to easily extend our pages by adding or rearranging components without modifying existing code. By designing the components to be easily slotted in we ensured flexibility, and scalability which in turn will make our application more maintainable and adaptable to future changes that might happen. For instance, the navigation bar that appears on all pages is not re-written for every instance, it is a component that is reused so that it can extend functionality without modifying every individual page.

Liskov Substitution:

No inheritance was used in our application

Interface Segregation:

Not possible as our projects backend was made using python, and interfaces are not possible in python

In the frontend, when passing props or using context, we would only pass props or components that are used on the pages to prevent unnecessary and confusing dependencies. It might be effective to divide UserContext.jsx into multiple context providers in the future, however, at the moment everything in UserContext is denoting user data so it adheres to Dependency Inversion Principle. That being said Contexts are not synonymous with interfaces so this is mostly analogous.

Dependency Inversion Principle:

Not possible as our projects backend was made using python, and interfaces are not possible in python

The UserContext also contributes to adhering to this principle as it allows components to depend on abstractions like hooks and contexts rather than concrete implementations such as doing fetches in every page.

Diagrams:

- Use-Case Diagram, Sequence Diagram(s),Layered Architecture Diagram and Presentation Layer Diagram can be found in repository under the Diagrams folder

Testing

Testing document for both unit and exploratory testing can be found in the testing folder on our github repository. Robust unit testing was conducted on all API routes that did not require a specific API key. All remaining main functionalities were tested throughout exploratory testing. Minimal testing was developed for user_settings.py as this functionality was not deployed in the final product.