Vaultify

"Vaulting music memories."

Interim Release Deliverables

Software Design and Documentation
1:30 PM - 3:35 PM Section
Professor John Sturman

The Band:
Matthew Bui, Michael Lam, Dillon Li, Michelle Li, Thomas Orifici

19 July 2024

VaultifyInquiries@gmail.com

Design Approach

The Vaultify project has been designed with a strong focus on object-oriented principles to make sure that our code is both modular and scalable. Maintainability is at the heart of our design philosophy, and this has informed the decisions that we've made in our codebase.

At the top level, our codebase is split into two distinct sections, the backend and the frontend. The frontend runs on the browser of the user, while the backend runs on a Node.js server. To minimize coupling between these two sections, we implemented two classes that are responsible for communication - the Transmitter and the Receiver. The Transmitter is responsible for sending post requests to the server and the Receiver is responsible for responding to these requests. These classes allow the rest of our frontend to exist independently from the rest of our backend, decoupling these sections.

In both our frontend and backend code, we've utilized multiple design patterns to solve common problems we encountered. For example, we've utilized a model view controller architecture throughout our application to separate the three layers of program control. The controller consists of both the Transmitter and Receiver mentioned above in addition to the Automator class, which is responsible for scheduling monthly generation of playlists and emails. The view section resides in our frontend and includes the Navigator class, which is responsible for displaying web pages. Our model is in the backend and includes the Model class, which is responsible for performing database and spotify operations based on requests. Additionally, our model section includes two facade design patterns, the SpotifyFacade and the DatabaseFacade classes, which separate the business logic of the Model from the external database and web API implementations. These design patterns have allowed us to clearly define class boundaries and decouple our codebase.

Another class heavily utilized in our codebase is the User class, which stores user information such as their username and playlists. This class allows for easy data transfer between classes and adds a level of readability to the codebase.

Throughout the design of our application, we've leaned heavily on SOLID principles to ensure scalability in our codebase. In particular, we've greatly benefited from the "Single Responsibility" and the "Dependency Inversion" principles of SOLID. We followed the principle of single responsibility by ensuring that the function of each of our classes can be encapsulated by a singular functionality. If a class became too big or

vague during development, we substituted it for multiple smaller, simpler classes. This allowed us to create a loosely coupled codebase that is easy to scale. We followed the principle of dependency inversion by ensuring that higher-level modules depend on abstractions rather than on concrete implementations. An example of this can be seen in our facade classes, which separate the model from the concrete implementation of database and API functionality. Additionally, our project has benefited from the use of the open-closed principle. While brainstorming the methods for our various objects, we made sure to simplify them as much as possible, allowing for new features to be added in methods without the need to change old methods.

In summary, we've built Vaultify with a structured approach that makes it easy to maintain and scale. By combining object-oriented principles and focusing on design patterns, we've built a system that's both efficient and scalable.

CRC Cards

Class: User	
Responsibilities: - **Stores user information including id, display name, email address, bio, profile picture, generated playlists, and monthly playlist opt-in status.**	Collaborators: - N/A
Coupling Rating: 1 - Data Coupling	Cohesion Rating: 7 - Informational

Class: Navigator	
Responsibilities: - **Displays react components for each page view depending on the current URL.** - **Calls the Transmitter to allow components to send requests to the server.** - **Stores the current User object to be used by components.**	Collaborators: - User, Transmitter
Coupling Rating: 1 - Data Coupling	Cohesion Rating: 6 - Functional

Class: Transmitter	
Responsibilities: - **Sends requests to the server for tasks such as logging in a user, generating a playlist, or updating a bio. Returns an updated User object sent by the server.**	Collaborators: - Receiver, User
Coupling Rating: 1 - Data Coupling	Cohesion Rating: 7 - Informational

Class: ClientMain	
Responsibilities: - **Configures and instantiates the client side application.** - **Constructs objects necessary for running the frontend including the Transmitter and Navigator.**	Collaborators: - Navigator, Transmitter
Coupling Rating: 1 - Data Coupling	Cohesion Rating: 6 - Functional

Class: Receiver	
Responsibilities: - **Parses HTTP requests from the client.** - **Calls the Model to update or fetch user data based on requests.** - **Returns a User object to the client.**	Collaborators: - Model, Transmitter, User
Coupling Rating: 1 - Data Coupling	Cohesion Rating: 7 - Informational

Class: Automator	
Responsibilities: - **Tracks the current date and time.** - **Calls the Model to generate playlists and send email reminders to opted-in users on the first of every month.**	Collaborators: - Model
Coupling Rating: 1 - Data Coupling	Cohesion Rating: 7 - Informational

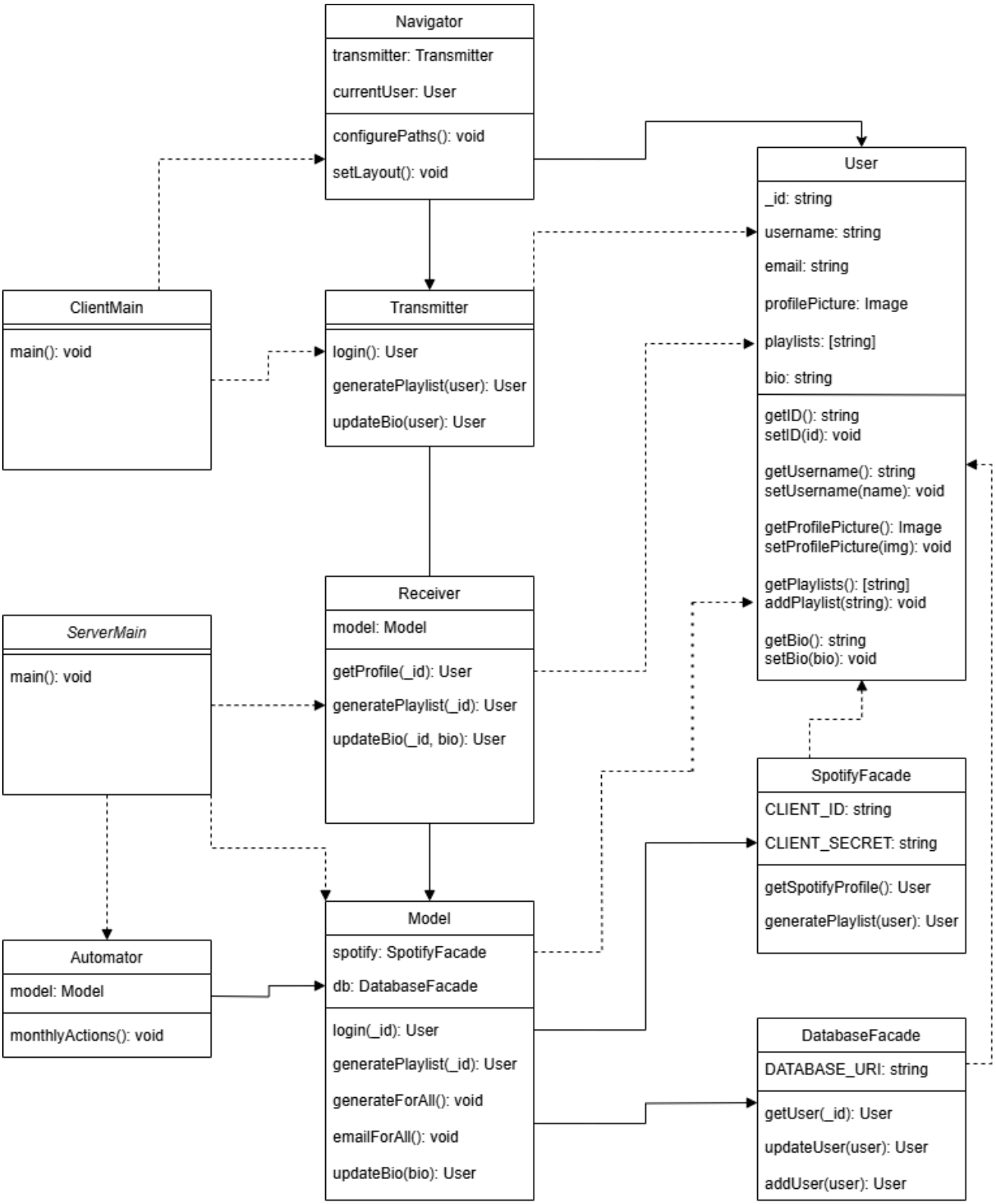
Class: Model	
Responsibilities: - **Constructs the SpotifyFacade and DatabaseFacade.** - **Calls the facades to handle data manipulation tasks such as logging in and saving a user, updating a user's bio, or generating a playlist for a user.**	Collaborators: - SpotifyFacade, Database Facade, User
Coupling Rating: 1 - Data Coupling	Cohesion Rating: 7 - Informational

Class: SpotifyFacade	
Responsibilities: - **Handles actions involving Spotify such as retrieving User profile data or generating a playlist for a User.**	Collaborators: - User
Coupling Rating: 2 - Stamp Coupling	Cohesion Rating: 7 - Informational

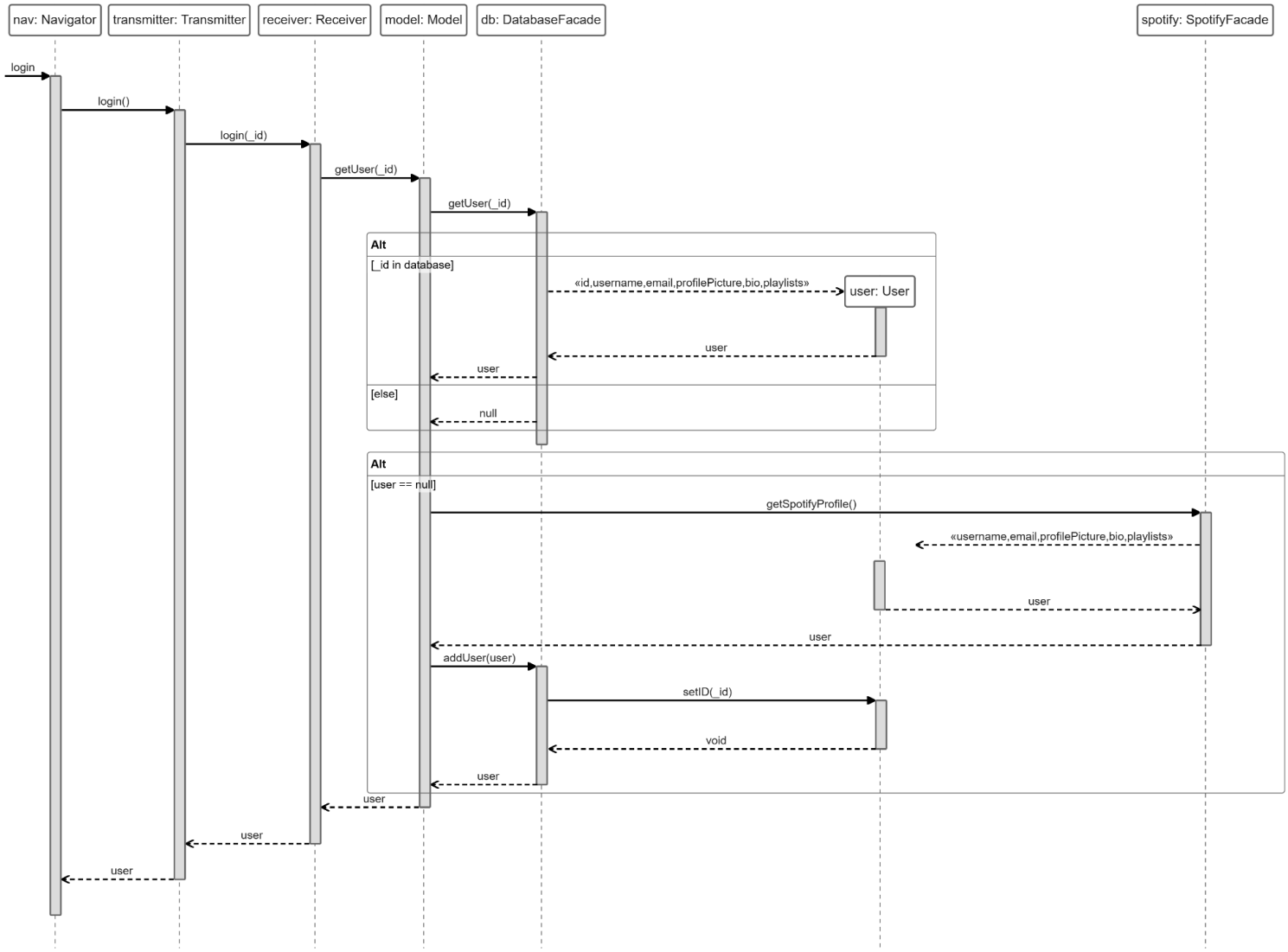
Class: DatabaseFacade	
Responsibilities: - **Handles actions involving the database such as finding, adding, or modifying a User.**	Collaborators: - User
Coupling Rating: 2 - Stamp Coupling	Cohesion Rating: 7 - Informational

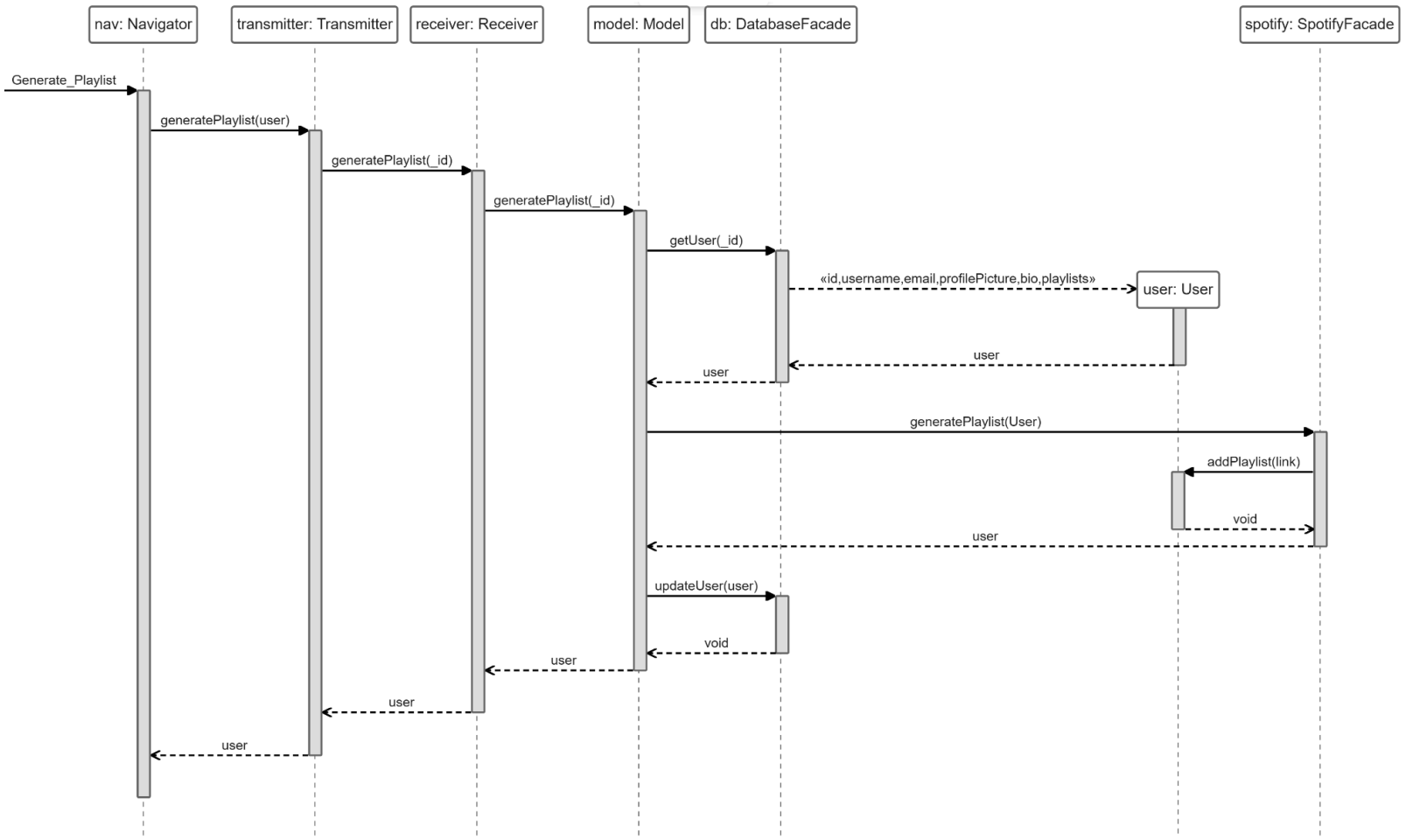
Class: ServerMain	
Responsibilities: - **Configures and instantiates the server.** - **Constructs objects necessary for backend operations including the Receiver, Model, and Automator.**	Collaborators: - Automator, Receiver, Model, SpotifyFacade, DatabaseFacade.
Coupling Rating: 1 - Data Coupling	Cohesion Rating: 6 - Functional

Static Class Diagram



Sequence Diagrams




nav: Navigator
transmitter: Transmitter
receiver: Receiver
model: Model
db: DatabaseFacade
spotify: SpotifyFacade
Generate_Playlist
generatePlaylist(user)
generatePlaylist(_id)
generatePlaylist(_id)
getUser(_id)
«id,username,email,profilePicture,bio,playlists»
user: User
user
user
generatePlaylist(User)
addPlaylist(link)
void
user
updateUser(user)
void
user
user
user

Interface Testing Summary

During our initial interface testing session, we had 10 different users interact with our platform, all of which provided very useful constructive criticism for our project. Although our users encountered problems throughout the guided tasks, each and every tester was able to accomplish the list of tasks provided, which are provided below.

1. *Login into the website with Spotify and go through the OAuth process*
2. *Read the tutorial on the Home page*
3. *Navigate to the Profile page and change your bio three times to the following:*
    - *An empty bio*
    - *“Hello World!”*
    - *Any phrase of your choosing*
4. *Navigate to the Playlist Generation page and generate a playlist with:*
    - *Time of 12 hours*
    - *Monthly email notifications*
5. *Navigate to the Vault page and find your playlist from May 2023*
6. *Navigate to the About Us page and read through the developer team’s biography*

The feedback we received from our users was generally positive, as they were able to easily navigate through most sections of the website without needing detailed instruction. Our UI was also praised by users for its general aesthetic. However, users did have several criticisms for the application. For example, a few of our users encountered trouble with accessing the vault page. When the “Vault” page is initially presented to users, a large vault is displayed on the screen. Upon clicking the vault, an animation plays that opens the vault and displays the user’s playlists. However, to several users it was unclear that the vault graphic had to be clicked. Additionally, some users thought that the vault opening animation was too long. Users also noted that it would make more sense for playlists to scroll vertically instead of horizontally in the vault. In addition to the “Vault” page, users also encountered some issues on the “Playlist Generation” page. When users generate a playlist they are given several parameters to customize their playlists. The function of some of these parameters were unclear to users, such as a date selector that didn’t seem to serve a purpose or a “New Only” option that was overly vague. On top of this, in terms of technical errors, our users found that the profile picture was specifically difficult to locate on Windows operating systems.

For the next steps for our user interface, we will be making several changes to address the concerns of users. For starters, we will change the vault animation to play automatically and speed it up. We will also implement vertical scrolling. On our playlist generation page, we will remove the option to select a date as it doesn't actually do anything and is not a feasible functionality. Additionally, we will change the "New Only" option to a "New to Me" option, as that better conveys that the option only includes songs that are new to the user. Finally, we will perform more testing on different development environments to ensure that our site is working on both Windows and Mac.

Overall, we found this exercise to be very useful in gathering feedback, and through it, we were able to identify several areas for improvement in our user interface. Moving forward, we will get as many eyes on our interface as we can as changes are made to ensure that our site provides an elegant and fluid experience for users.

Project Status Report

Since Sprint 4, the Band has progressed tremendously in the development of Vaultify. Our team has completed the tasks that were outlined in our updated project schedule from Sprint 4, including implementing our Figma page designs, configuring playlist generation and database functionality, and setting up monthly automation for our site. We have also streamlined our development process through the deployment of a peer review system and the implementation of an automatic linter and code formatter.

Our current risks primarily involve scaling our system to implement new features. While we've built a solid and scaleable foundation with our current codebase, the integration of new systems and features does present a potential risk, primarily in regards to storing new playlist information and adding more options for automation. These features will involve extending both the Automator and User classes of our codebase. We will have to ensure that these extensions follow SOLID principles such as the open-closed principle to maintain the stability of our platform.

Moving forward, the team's next steps are to resolve the bugs discovered during the Interface testing and finish the implementations of Vaultify's core features to prepare for the beta release of our platform. Each team member will continue working on their assigned tasks to refine Vaultify for our Beta Release.

Contribution Summary

Matthew Bui:
Matthew developed and implemented the “Playlist Generation” page and linked it to the “Playlist Successfully Generated” page, which will redirect the user to their playlist in Spotify. In terms of documentation, Matthew completed the status report section.

Michael Lam:
Michael was responsible for development of the “Vault” page, including the vault’s animation and the timeline’s styling. He also helped brainstorm the team’s set of CRC cards and sequence diagrams and helped in writing up the interface testing summary section.

Dillon Li:
Dillon was responsible for developing the Automator class, allowing for automatic monthly playlist generation and email notifications. Dillon also helped to implement part of the model and contributed to the design approach.

Michelle Li:
Michelle was responsible for developing the interfaces for the “Home,” “Profile,” “Welcome,” and “About Us” pages. She also developed the navigation bar and served as the UI overseer, making sure that the team’s work looked standardized and followed our mock ups. For the deliverable, Michelle contributed to the design approach, interface testing summary, and status report sections.

Thomas Orifici:
Thomas finalized the CRC cards for the group and constructed both the static and sequence class diagrams. Additionally, he was responsible for implementing the Model, SpotifyFacade, DatabaseFacade, Receiver, and Transmitter classes.