Swift Link

Team MAX

Team members:
Yiqiao (May) Wang (905319079)
Olivia Zhang (705377225)
Rick Yang (405346443)
Yiteng Jiang (805577865)
Mike Shi (005592011)

GitHub repository: https://github.com/zhtyolivia/UCLA-CS130-Project

1 Analysis Description

1.1 Use case diagram

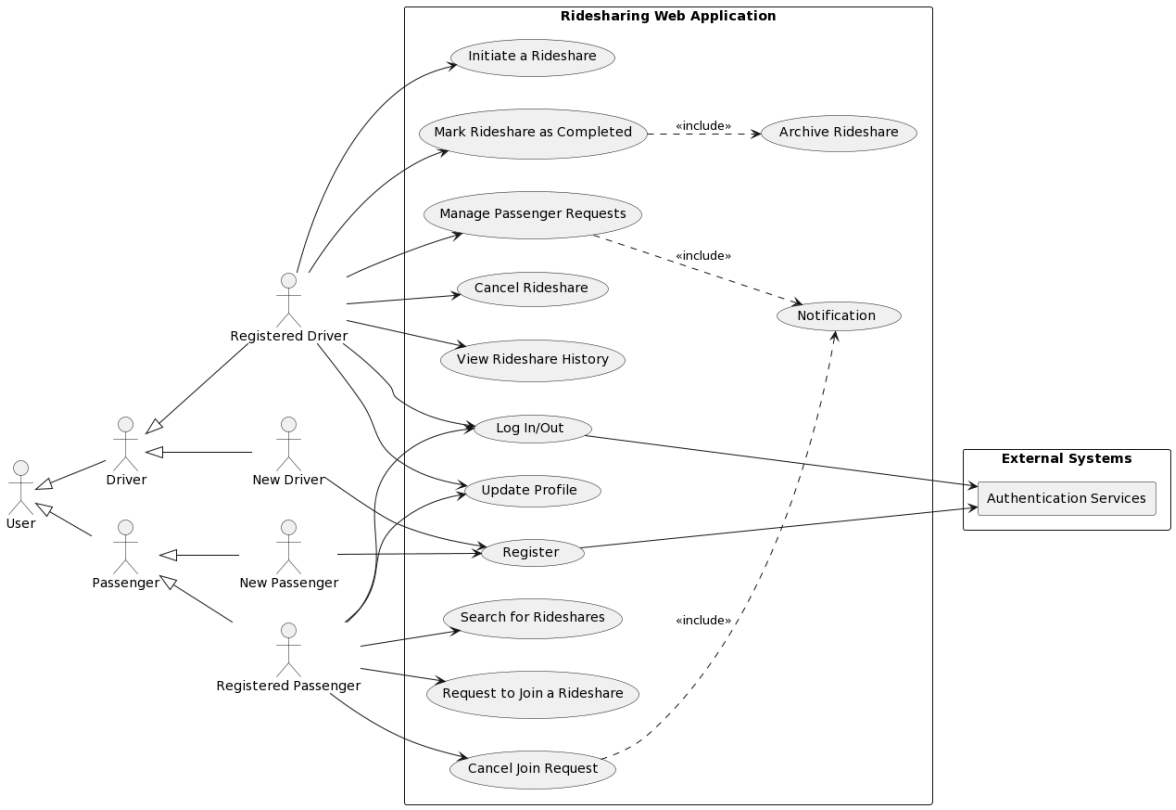


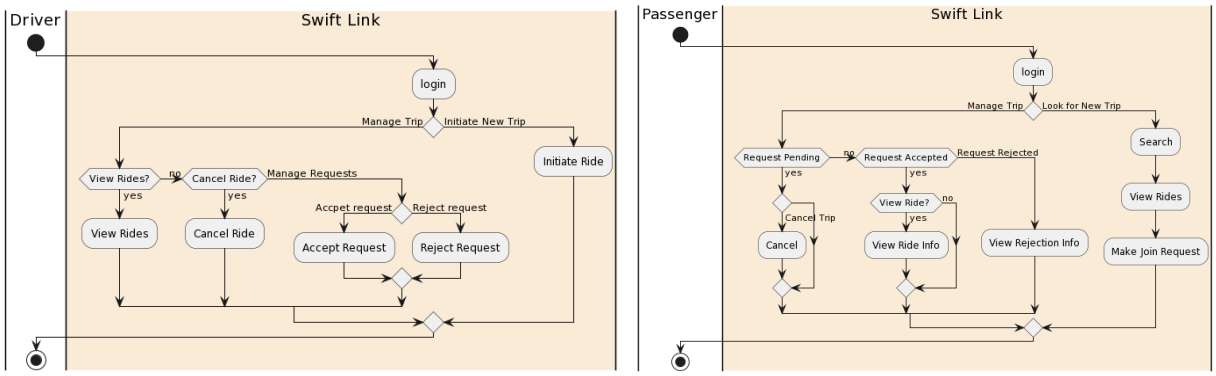
Figure 1. The use case diagram of the Swift Link application showing interactions of the subject system with the user and external system.

Figure 1 represents the use case structure of a car sharing web application. It outlines the interactions that different types of users, including "Registered Driver," "New Driver," "Registered Passenger," and "New Passenger," can have with the system. The diagram also shows "Authentication Services" as an external system interacting with the application for use cases like "Register" and "Log In/Out."

1.2 Activity diagrams

Figure 2 shows two activity diagrams illustrating two main activities of our Swift Link web application, passenger interactions and driver interactions with the application, respectively. In Figure 2 (a), a driver needs to first log in. The driver can then initiate a rideshare. If the driver wishes to manage a ride, the driver can view or cancel the rideshare and manage rideshare requests. When managing the requests, the driver can either accept or reject the request. Figure 2(b) shows the activity of a passenger. After logging in, the passenger can manage a joining request that they have previously sent. If the request is still pending (i.e., not accepted or rejected by the driver), the passenger can choose to cancel the request. If the request was already accepted, the passenger can view ride information. If the request was rejected, the passenger can

view the rejection information. If the passenger does not want to manage an existing request, they can also choose to search for a desired rideshare and make a join request.



(a) (b)

Figure 2. Activity diagrams showing interactions of users with the application and important features. (a) shows Driver's interactions with the Swift Link application. (b) shows Passenger's interactions.

1.3 Class diagram

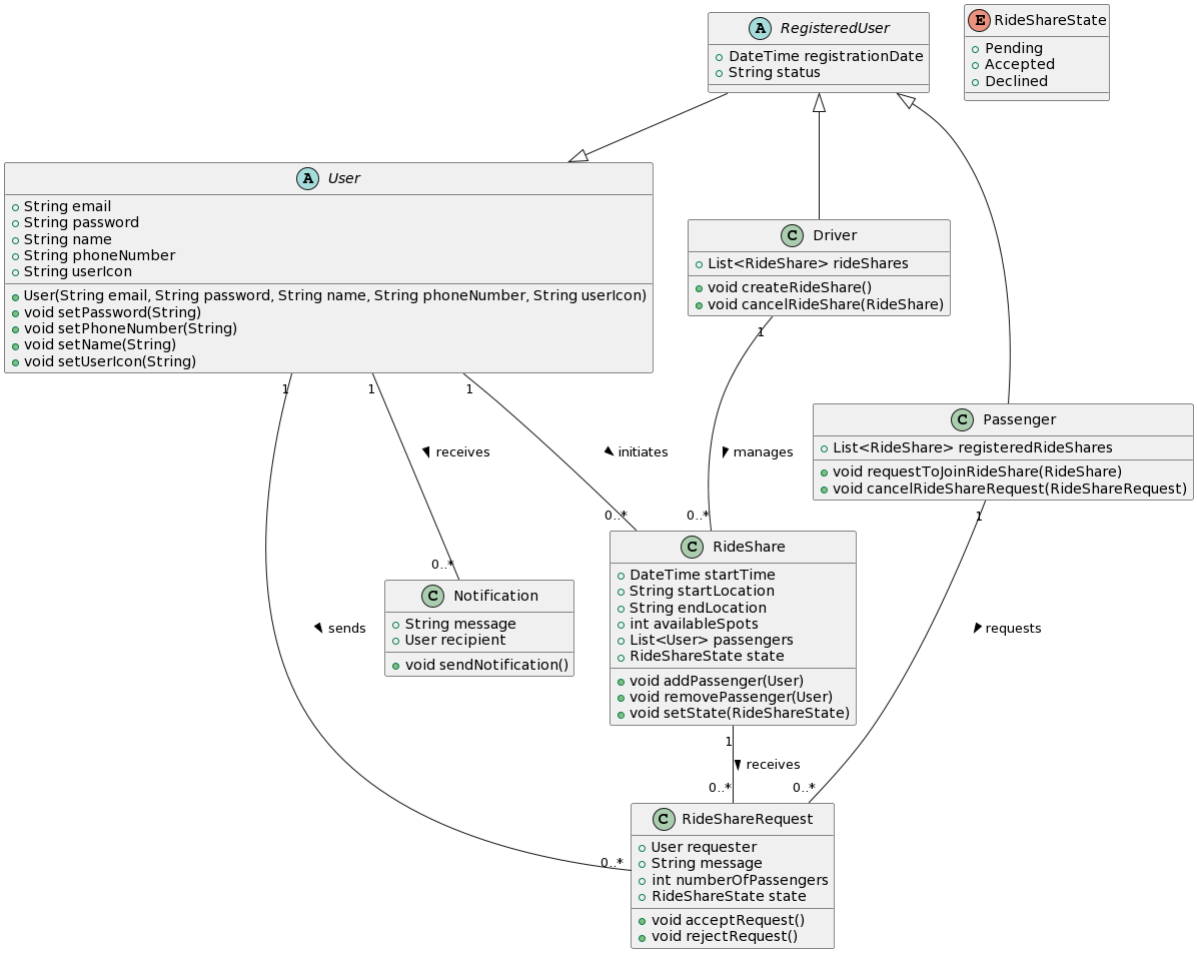


Figure 3. Class diagram of the Swift Link application.

The class diagram in Figure 3 illustrates how classes and components are defined through a class diagram.. At the core is the abstract “User” class, which has a subclass “RegisteredUser” that further subdivisions into “Driver” and “Passenger” subclasses. These subclasses represent the different user roles that interact with the application's features.

Another class that is key to the application is the “RideShare” class, which is responsible for storing and managing essential ride information, including time posted, start and end locations, and participant details. The ride's status is tracked by the “RideShareState” enumeration, indicating whether it is pending, accepted, or declined. Requests from passengers are encapsulated in the “RideShareRequest” class, which records the requester's details, their message, the passenger count, and the state of the request.

Moreover, the “Notification” class is responsible for dispatching alerts to users, informing them of actions such as ride requests or changes in ride status. Notably, drivers have an associated list of “RideShare” objects to manage their rides, whereas passengers have a list of registered rideshares. Overall, this class diagram provides a structured overview of the Swift Link application's design and will be used as a foundational guide for system development.

1.4 Sequence diagram

Figure 4 displays the sequence diagram depicting the processes and objects involved in our application. A rideshare is first initiated by a driver, who needs to provide details such as start location, destination, and estimated start time. The system records this rideshare. After some time, a passenger makes a request to join the rideshare. The system records the request and waits for driver’s acceptance/denial of the request. If the driver accepts, the system sends a notification to the passenger that the request was accepted. If the driver rejects, the system sends a notification to the passenger that the request was rejected. Moreover, after the request is made and before the driver accepts/rejects the request, the passenger may choose to cancel the request, and a notification will be sent to the driver. The Rideshare system updates the status. If the Driver decides to cancel the rideshare, the system enters a loop where it updates each passenger (who might have joined the RideShare) that the rideshare is canceled. It removes the rideshare from each passenger's records.

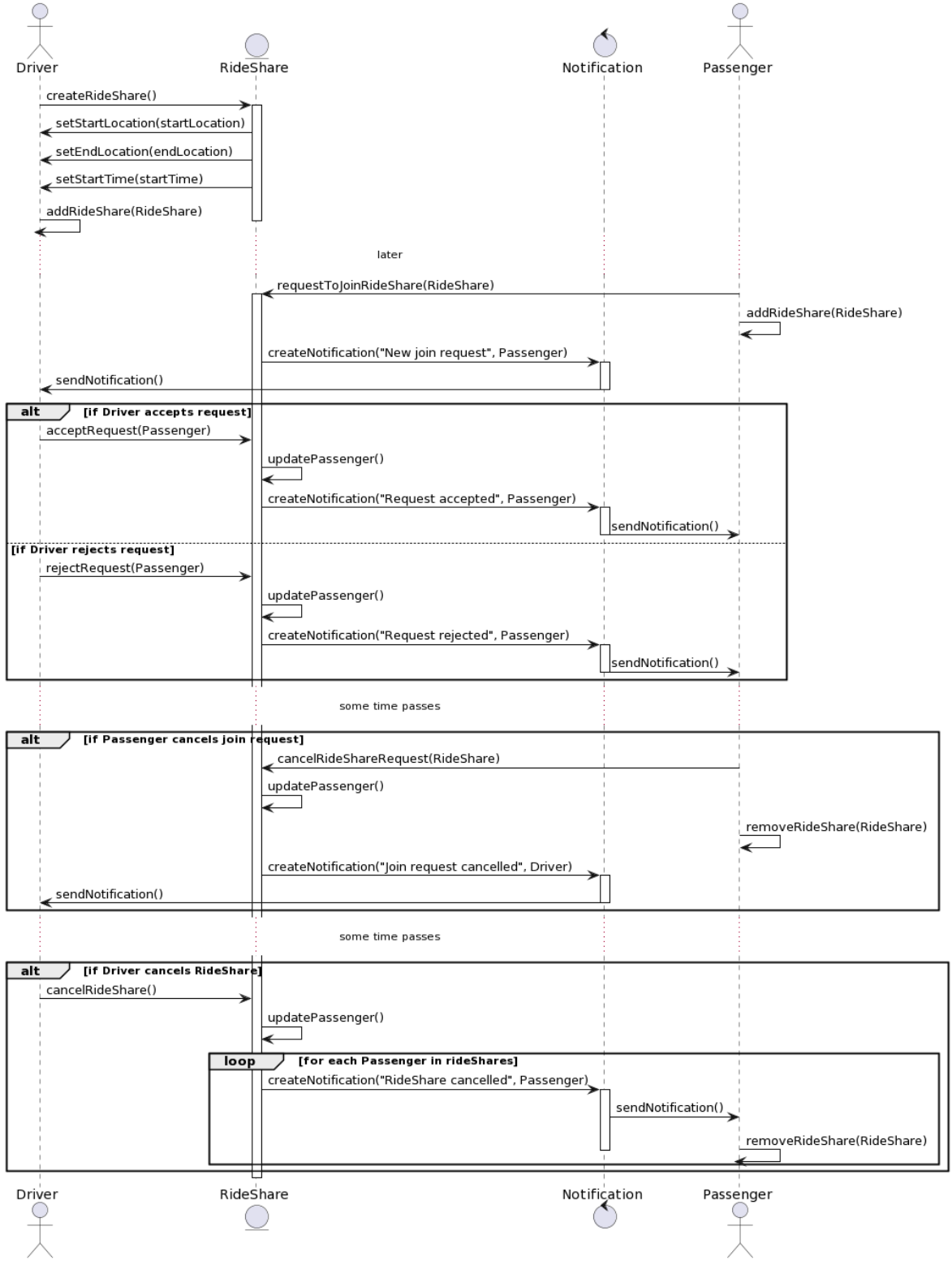


Figure 4. Sequence diagram modeling interactions between driver, passengers and, and the Swift Link application.

1.4 State machine diagrams showing the most important and complex components

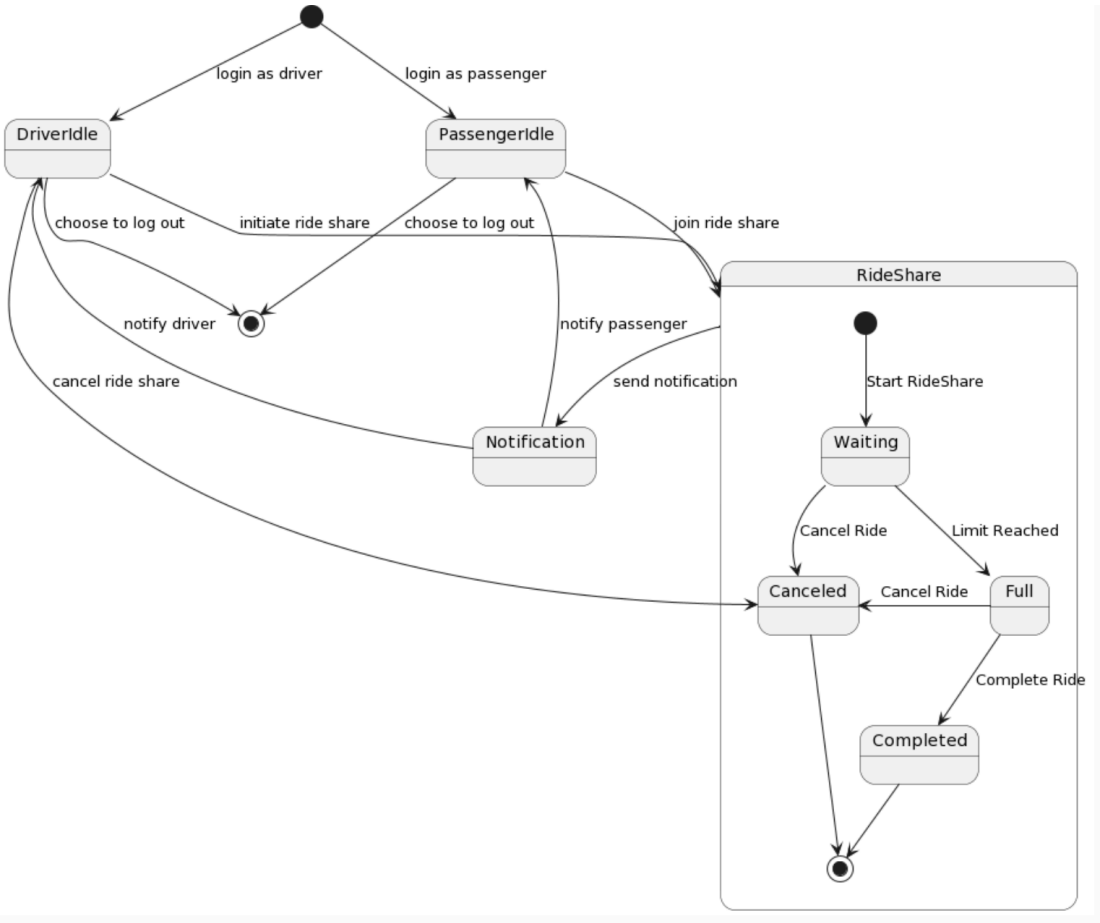


Figure 5. State machine diagram of the Swift Link application.

Figure 5 shows the state machine diagram of the RideShare class within the Swift Link application. It represents the states and transitions between states for this application. From the start state, the system transitions to the DriverIdle and PassengerIdle states, depending on the type of the user that performs the log in action. These initial idle states represent a driver and a passenger who have logged into the system, verified through third party application but have no actions so far respectively. From the “PassengerIdle” state, the passenger can choose to “join rideshare” which joins a new rideshare. The passenger can also choose to “log out,” which terminates the process. Similarly, from the “DriverrIdle " state, the driver can choose to “initiates rideshare” which handles the initiating requests. The driver can also choose to “log out.” Besides these two same functionalities, a driver will also be able to cancel a ride share which also leads to “RideShare” states.

Detailing the “RideShare” state, the state begins with a “Waiting” internal state. If a rideshare is canceled, it goes to the “Canceled” state. If the rideshare reaches its limit of the number of passengers, it transitions to the “Full” state. The system can transition from the “Full” state to the “Completed” state when the rideshare is completed.

The “Notification” state is an intermediary state between RideShare state and user idle states. This state represents the repeated notifications or ongoing notification processes when a join request is rejected/accepted and when a rideshare is canceled by the driver.

2 Architecture Description

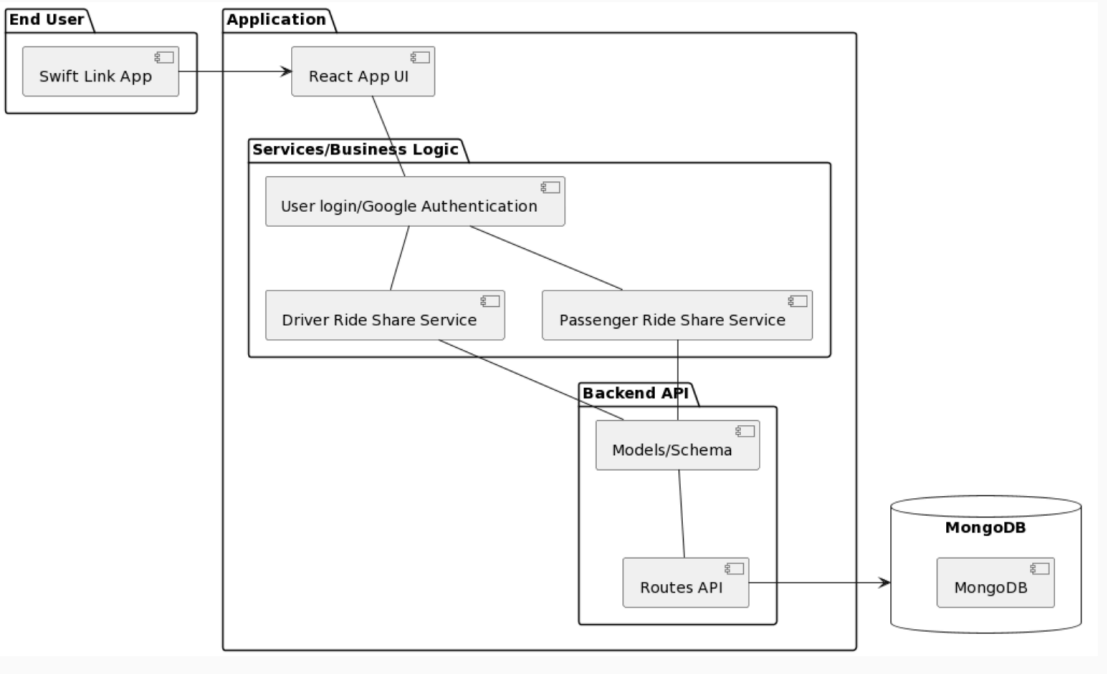


Figure 6: Illustration of the monolith application landscape.

Figure 6 shows the **monolith application landscape** that we adopt in Swift Link. Due to our tight project development schedule and small number of deployments, we believe that the monolithic paradigm is very appropriate. It provides simplicity and convenience of deployment, which are beneficial when the team is on a steep learning curve with new programming languages and frameworks over a short period of time. Within the application block, we can further separate the application into UI, services and backend API where all components of the application are tightly integrated into a single codebase.

We used the **layered application structure** in our development. As shown in Figure 7, there are four layers in our structure: the presentation layer, the business layer, the persistence layer, and the database layer. The presentation layer is responsible for the user interface and user experience built with React.js. The business layer handles the core logic of the application. It is responsible for handling the application's operations and includes functionalities such as notification, rideshare management, and rideshare request management. The persistence layer provides an abstraction over the data sources using Routes API. Finally, the database is the data storage layer of the application. It utilizes MongoDB to store user information, rideshare details, and request data.

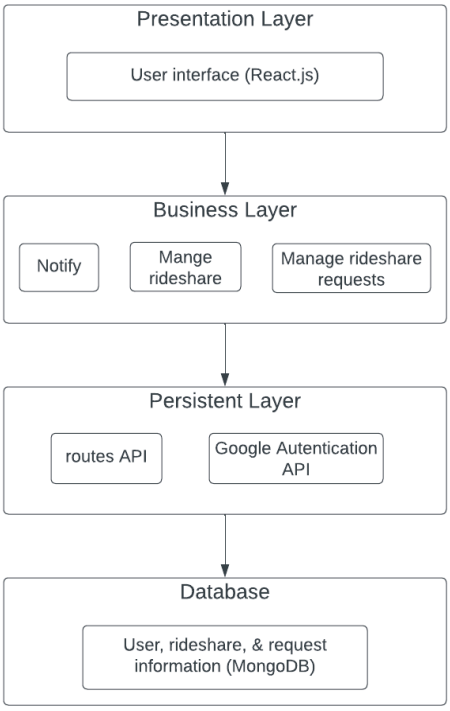


Figure 7. Illustration of the layered application structure.

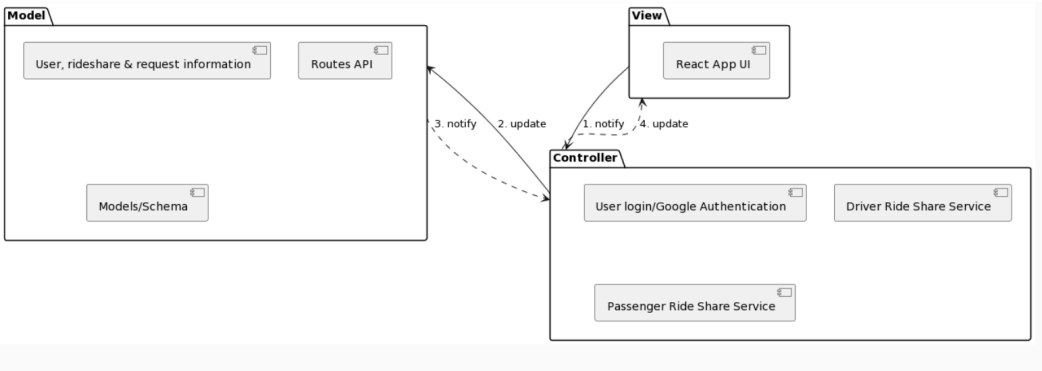


Figure 8. Illustration of MVC UI Pattern.

For our application, we choose to use **Model-View-Controller (MVC) UI design pattern** since it would serve to separate concerns and make the project easier to manage and scale. The Model would handle all data-related logic such as user, rideshare and requests information, routes API and all the models and schemas. The View, built with React App UI, would be responsible for presenting this data to the user using interactive interface. The Controller handles input from the React App UI, processing user login and authentication via Google, and directing the flow of data (driver and passenger ride share services) between the View and the Model. It would handle the business logic for both driver and passenger ride-sharing services, updating the Model with new data and notifying the View to reflect changes. This model best suits our application's capabilities of separating users and controlling between the services.

3 Design Description

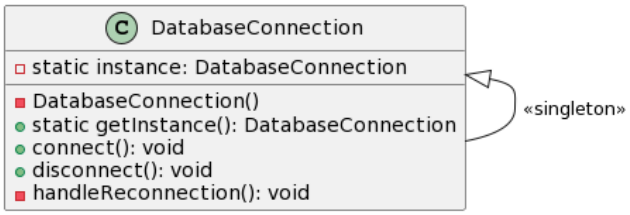


Figure 9. A class diagram illustrating the singleton pattern used for Database Connections.

Within the ride-sharing application, the **Singleton pattern** is utilized for the Database Connection with MongoDB to ensure that there is only one instance of the database connection throughout the application's lifecycle. This design pattern provides a single point of access to the database connection, preventing the overhead of multiple instances. It encapsulates all the necessary functionality to connect, disconnect, and manage the database interactions. This ensures that the application uses a single, shared connection to the database, conserving resources and maintaining a consistent state across different parts of the application.

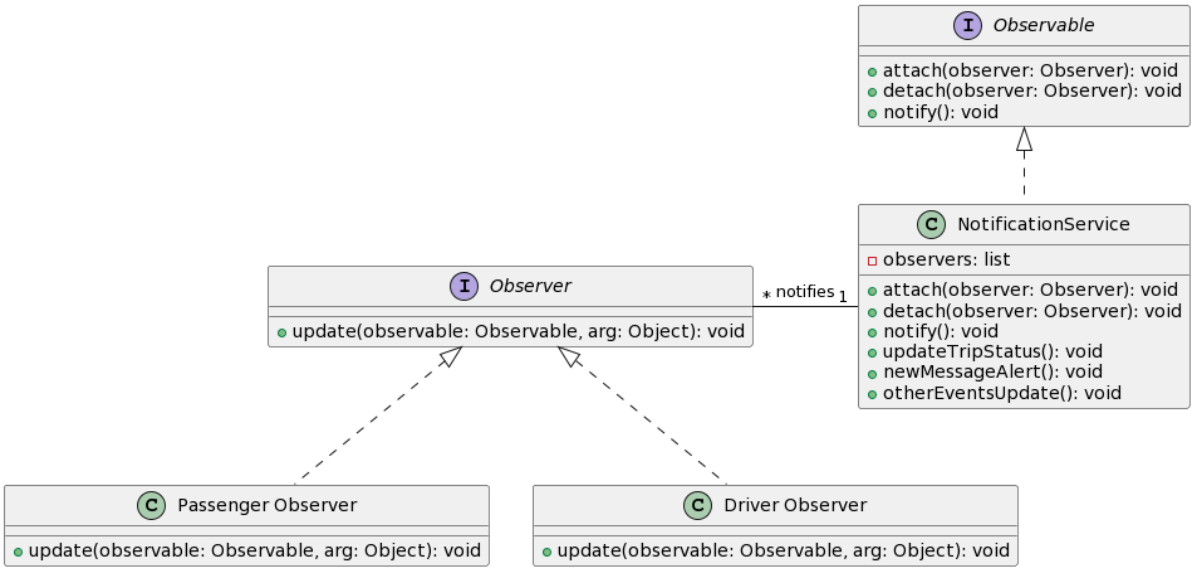


Figure 10. A class diagram illustrating the observer pattern used in Notification design.

We choose the **observer pattern** for the Notification designs. This best suits our needs to manage and dispatch notifications regarding various user-related events, such as trip status updates, new messages, or other notifications. The important part of the diagram is the "NotificationService" class which is a concrete subject class. It implements the "Observable" interface which is subject interface and maintains a registry of different Observers – in this case, "UserObserver" and "DriverObserver". When an event occurs, such as a change in trip status, the "NotificationService" class is responsible to notify all current observers whenever state changes. The two concrete observers are used to update the changes. This design allows the application to scale effectively, as it can support a growing number of users and notifications

with minimal coupling between the service responsible for managing notifications and the users who receive them.

4 Implementation Status

4.1 Progress by epics and user stories

At this point of development, we have completed most of our sprint 1 goals and are migrating to work on the sprint 2 goals. Sprint 1 goals mostly focused on the initial set up of infrastructure and planning of the software design. Feature-wise, we also successfully delivered registration and login functionalities for both driver and passenger clients. With available backend APIs and front end designs, we are 95% done on epic 1 except for client-server connection for the account modification functionality. The completed list of epics and user stories are listed below:

https://github.com/zhtyolivia/UCLA-CS130-Project/issues/1
https://github.com/zhtyolivia/UCLA-CS130-Project/issues/1
https://github.com/zhtyolivia/UCLA-CS130-Project/issues/14
https://github.com/zhtyolivia/UCLA-CS130-Project/issues/7
https://github.com/zhtyolivia/UCLA-CS130-Project/issues/8
https://github.com/zhtyolivia/UCLA-CS130-Project/issues/6
https://github.com/zhtyolivia/UCLA-CS130-Project/issues/5
https://github.com/zhtyolivia/UCLA-CS130-Project/issues/48

From a bigger scope, we are approximately 30% done for the entire project as we have successfully tackled almost the entirety of Epic1 and have done about 20% of work for Sprint2 where we will be focusing on the Driver page and rideshare post management from the driver's end.

Throughout our development process, we have strictly followed the SCRUM process. This significantly accelerated our development as we have most of the features thoroughly defined and designed in the beginning. All team members were also able to equally distribute the work during our sprint planning session. During standups, most blockers are effectively communicated and prompted to resolve rapidly. We were also able to leverage the sprint retrospective to review some of the areas we can improve on. The biggest key point that we want to improve on is the documentation of progress. Initially, we were relying heavily on figma as we were going through the software design process. As a result, a lot of progress was tracked using figma rather than github. To ensure a more robust software development process in the future, we have already migrated most of the records back to github. From now on, we will also record all development activities using github as the only source of truth.

4.2 Libraries, frameworks, and special tools

For the overall framework, we utilized the MERN techstack which consists Mongodb, Express.js, React, and Node.JS.

On the backend, we include JSON Web Token as a middleware to track user instances. Upon sign in, a token will be returned to the frontend to signify which user was signing in in this session. The token will be later decrypted to retrieve specific user data when prompted. We also used bcrypt when storing user passwords to the Mongodb database. Both of these implementations while slightly complicated backend development ensures the data safety of our users, making sure that their private data won't be easily accessible by others.

The construction of our front-end architecture has employed the library, "Font Awesome", suitably able to combine icons for Google and icons for password protection features without disconnects. While the utilization of this library enhances the user interface with aesthetically pleasing and recognizable icons, it is important to consider the trade-offs involved. The reliance on external libraries can bring additional dependencies and latent costs to the project.

Throughout the development process, we have leveraged GPT4 when constructing some of the APIs and front end designs. For the login and registration API, we leveraged the tutorial on Youtube: How to Create a NodeJS Login System Backend with Express and MongoDB… made by ToThePointCode.

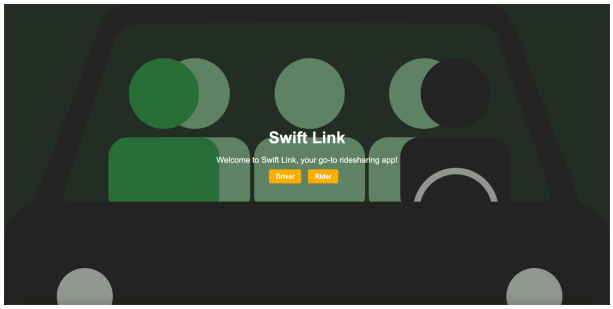
4.3 Special Techniques

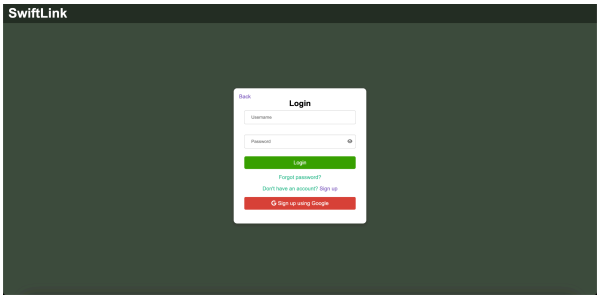
Since we divided our development into frontend and backend. It is not always easy to test under client-server connections for every single new API created. To ensure the capabilities of the APIs without frontend connection, we utilized Postman as our API testing tool. For each API, before connecting with the frontend, we thoroughly test it with Postman using several test cases (including edge cases).

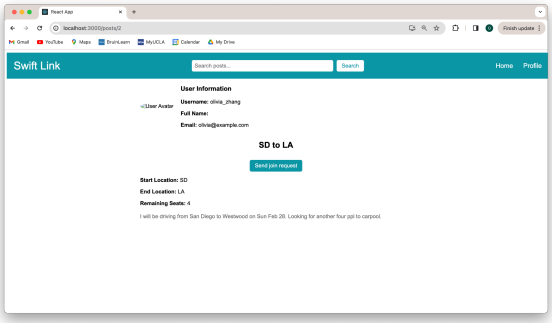
In addition, to ensure our application's capability to manage concurrent user sessions effectively, we employed Locust for conducting scalability and performance assessments. This tool facilitates the simulation of millions of users accessing the application simultaneously, allowing us to meticulously evaluate and enhance its performance under heavy load conditions

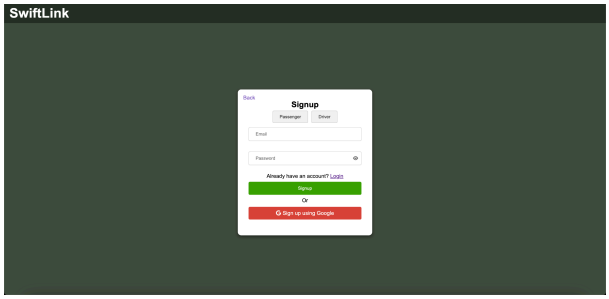
What is more, we implemented Prettier as our code formatting tool, which helped maintain high coding standards and styles. This approach led to a more organized and standardized code, contributing to the overall cleanliness and coherence of our project's codebase.

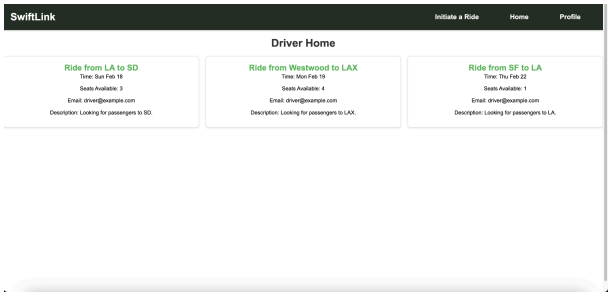
4.4 UI Mock-up

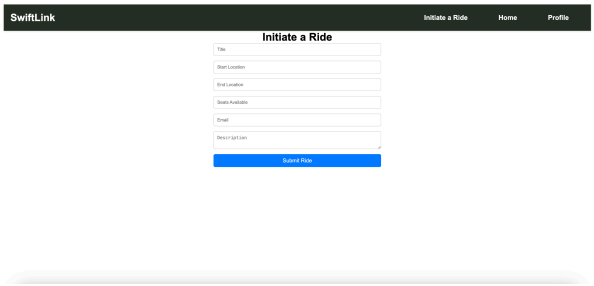

Swift Link
Welcome to Swift Link, your go-to ridesharing app!
Driver
Rider

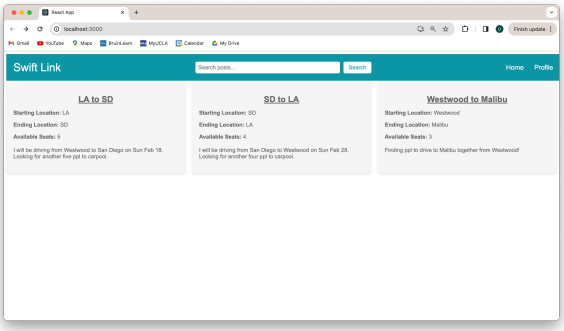

SwiftLink
Back
Login
Username
Password
Login
Forgot password?
Don't have an account? Sign up
Sign up using Google

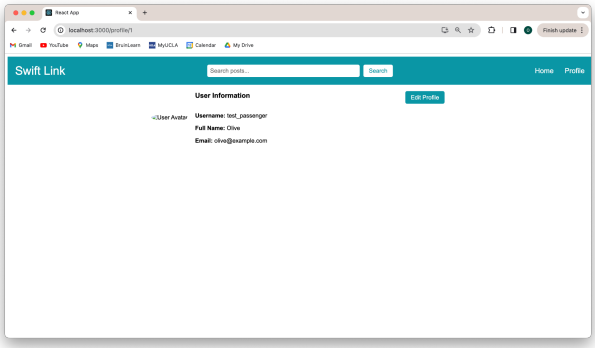

React App
localhost:3000/posts/2
Finish update
Gmail
YouTube
Maps
BrainLearn
MyUCLA
Calendar
My Drive
Swift Link
Search posts...
Search
Home
Profile
User Information
User Avatar
Username: olivia_zhang
Full Name:
Email: olivia@example.com
SD to LA
Send join request
Start Location: SD
End Location: LA
Remaining Seats: 4
I will be driving from San Diego to Westwood on Sun Feb 28. Looking for another four ppl to carpool.


SwiftLink
Back
Signup
Passenger
Driver
Email
Password
Already have an account? Login
Signup
Or
Sign up using Google


SwiftLink
Initiate a Ride
Home
Profile
Driver Home
Ride from LA to SD
Time: Sun Feb 18
Seats Available: 3
Email: driver@example.com
Description: Looking for passengers to SD.
Ride from Westwood to LAX
Time: Mon Feb 19
Seats Available: 4
Email: driver@example.com
Description: Looking for passengers to LAX.
Ride from SF to LA
Time: Thu Feb 22
Seats Available: 1
Email: driver@example.com
Description: Looking for passengers to LA.


SwiftLink
Initiate a Ride
Home
Profile
Initiate a Ride
Title
Start Location
End Location
Seats Available
Email
Description
Submit Ride


React App
localhost:3000
Finish update
Gmail
YouTube
Maps
BruinLearn
MyUCLA
Calendar
My Drive
Swift Link
Search posts...
Search
Home
Profile
LA to SD
Starting Location: LA
Ending Location: SD
Available Seats: 5
I will be driving from Westwood to San Diego on Sun Feb 18. Looking for another five ppl to carpool.
SD to LA
Starting Location: SD
Ending Location: LA
Available Seats: 4
I will be driving from San Diego to Westwood on Sun Feb 28. Looking for another four ppl to carpool.
Westwood to Malibu
Starting Location: Westwood
Ending Location: Malibu
Available Seats: 3
Finding ppl to drive to Malibu together from Westwood!


React App
localhost:3000/profile/1
Finish update
Gmail
YouTube
Maps
BruinLearn
MyUCLA
Calendar
My Drive
Swift Link
Search posts...
Search
Home
Profile
User Information
Edit Profile
User Avatar
Username: test_passenger
Full Name: Olive
Email: olive@example.com