

Swift Link

Team MAX

Team members:

Yiqiao (May) Wang (905319079)

Olivia Zhang (705377225)

Rick Yang (405346443)

Yiteng (Ethan) Jiang (805577865)

Mike Shi (005592011)

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

1. Final Design

Functional Requirements

There were no notable deviations from the intended original design presented in Part B. Our application has continued to evolve, building upon the foundational features set out in earlier phases. The core functionalities, including account management, ride-sharing initiation, and join request management, have been refined and enhanced to improve user experience and application performance.

Non-functional Requirements

For the most part, our non-functional requirements remained consistent with our initial design due to the limited time frame for significant changes. However, as our testing became more rigorous and our application's scope expanded, we identified and addressed several potential security vulnerabilities:

- Enhanced security measures were implemented to safeguard user data, especially in light of our application's integration with Google services for authentication and profile management. This included securing refresh tokens and ensuring that sensitive information like our MongoDB URI was no longer stored in public repositories.
- Environment variables critical to our application's security were moved to protected locations, accessible only to our development team on secured platforms like Heroku and TravisCI. This measure prevents unauthorized access to our database and Google accounts.
- Passwords and other sensitive credentials were updated and secured to further bolster our application's defense against potential security breaches.

Epics/Stories

Since the completion of Part B, we have introduced additional stories to enrich our application's feature set and address user feedback:

- Multi-page Functionality: Added to improve navigation and user engagement with our application, enabling a more intuitive and seamless user experience.
- Landing, Error, and Confirmation Pages: Developed to provide users with clear feedback on their actions, enhancing the overall usability and reliability of our application.
- Enhanced Testing and Documentation: Focus was placed on improving our development practices, including the introduction of more comprehensive testing strategies and the generation of automatic documentation to facilitate easier maintenance and future development efforts.

In summary, our project has adhered closely to its initial design intentions while incorporating necessary refinements to security measures and user interface enhancements. The introduction of new stories post-Part B has been aimed at enriching the user experience and ensuring the application's robustness and scalability. These developments reflect our commitment to delivering a secure, user-friendly, and highly functional ride-sharing platform.

Software Architecture

The software architecture of Swift Link remains consistent with our initial vision, leveraging a monolithic architecture integrated within a layered model, and employing a user interface developed according to the Model-View-Controller (MVC) design pattern. This approach has enabled us to maintain simplicity in deployment and development, which is crucial given our project's tight timeline and the complexity of implementing a scalable, real-time ride-sharing application.

Monolithic Architecture: Our choice to continue with a monolithic architecture has facilitated ease of development and deployment, particularly beneficial given our project's scope and the learning curve associated with new technologies. The monolithic architecture ensures all components of Swift Link - from user authentication to ride management - are integrated within a single, unified codebase.

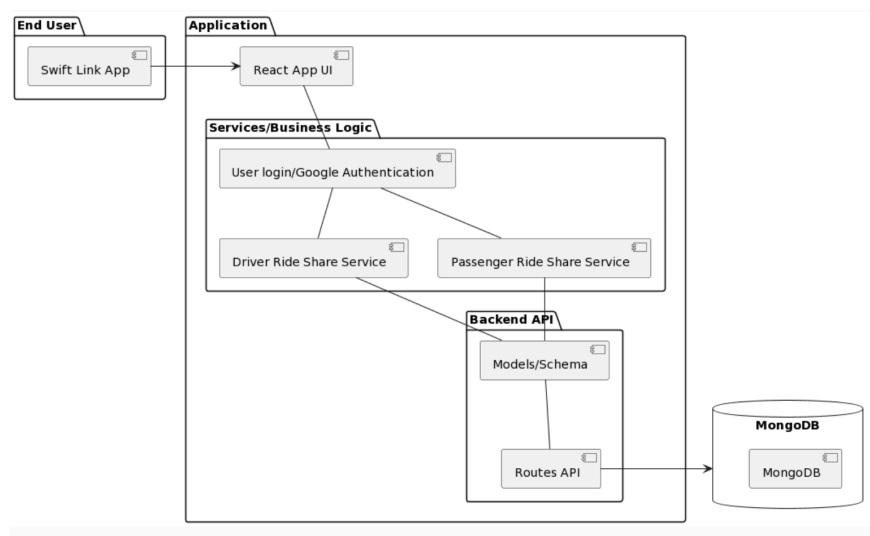


Figure 1: Illustration of the monolith application landscape.

Layered Application Structure: Swift Link is structured around a four-layer model.

- **Presentation Layer:** Built with React.js, this layer is responsible for the application's user interface and overall user experience.
- **Business Layer:** Manages the core logic of the application, including notifications, ride-sharing operations, and request handling.

- Persistence Layer: Provides an abstraction over data sources with a focus on route management and API integration.
- Database Layer: Utilizes MongoDB for data storage, including user profiles, ride details, and transaction records.

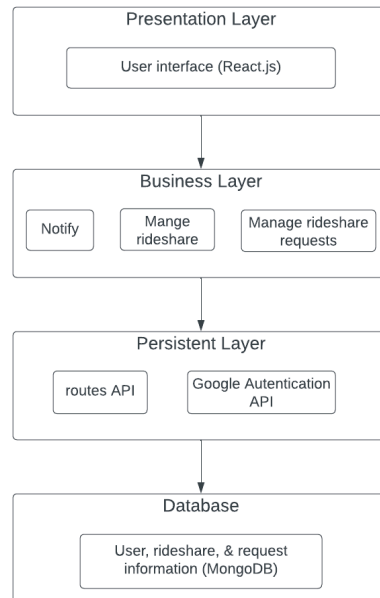


Figure 2. Illustration of the layered application structure.

Model-View-Controller (MVC) Design Pattern for UI: The MVC pattern allows for a clear separation of concerns, facilitating easier management and scalability of Swift Link's user interface components.

- Model: Handles data-related logic, including user information and rideshare details.
- View: Presents data to the user, crafted using React for dynamic and interactive UI components.
- Controller: Processes user input, managing authentication and directing the flow of information between the View and the Model.

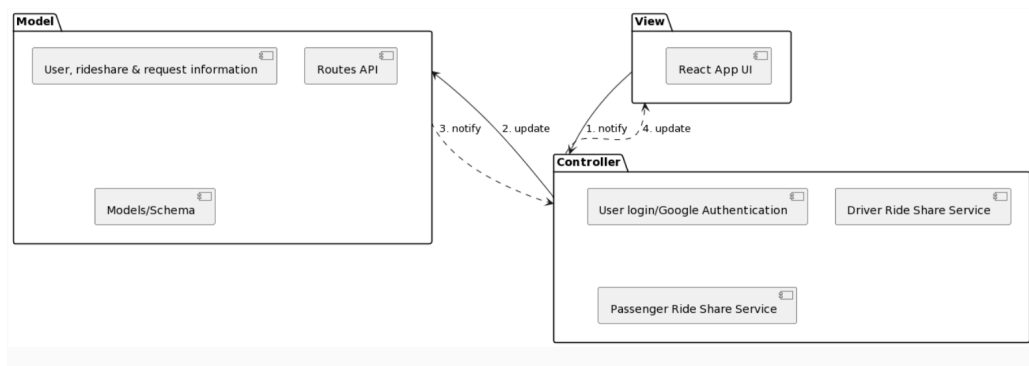


Figure 3. Illustration of MVC UI Pattern.

Software Design

Swift Link's design leverages both the Singleton and Observer patterns to enhance the application's functionality and user experience:

Swift Link's software design incorporates the Observer pattern to manage notifications, a key aspect for a responsive and scalable application. This design is represented in our class diagram, which clarifies the relationships between the core components.

- **User Class Hierarchy:** Central to our model is the User class, with subclasses RegisteredUser, Driver, and Passenger. This structure supports both shared functionalities and role-specific features efficiently.
- **RideShare Management:** The RideShare class captures essential ride details, monitored by the RideShareState for lifecycle management. Parallely, RideShareRequest handles join requests, including statuses and passenger information.
- **Notification System:** At the core of our Observer pattern implementation is the NotificationService, acting as the subject for UserObserver instances. It notifies users of events like ride updates or status changes, ensuring timely communication within the platform.

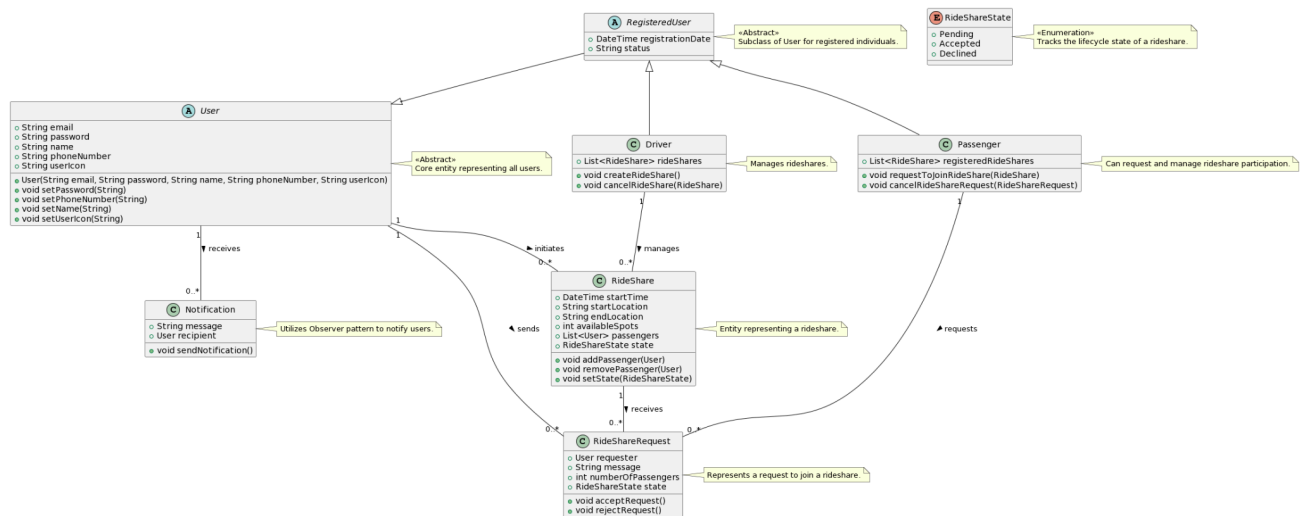


Figure 4. Class diagram of the Swift Link application.

This streamlined approach highlights Swift Link's commitment to modular, maintainable, and scalable design, ensuring our platform can adapt and grow alongside our user base and market developments.

Singleton Pattern for Database Connections: This design pattern ensures a single, consistent database connection is maintained throughout the application, optimizing resource use and data integrity.

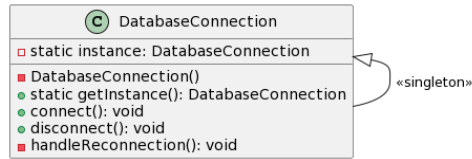


Figure 5. A class diagram illustrating the singleton pattern used for Database Connections.

Observer Pattern for Notifications: To manage and dispatch notifications efficiently, Swift Link utilizes the Observer pattern. This allows for scalable communication between the application and its users, concerning ride status updates and other important notifications.

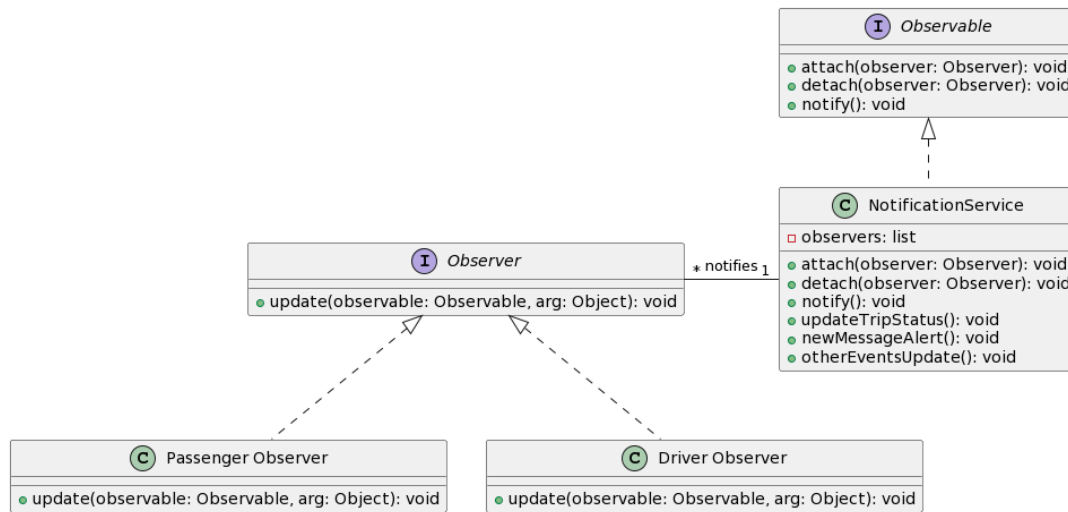


Figure 6. A class diagram illustrating the observer pattern used in Notification design.

This structure has been pivotal in developing Swift Link into a robust and user-friendly ride-sharing platform. Our adherence to proven software architecture principles and design patterns has enabled the successful implementation of complex functionalities while maintaining the project's development momentum.

GPT/Copilot Usage: In this project, we leveraged GPT and Copilot in parts of the code to ensure consistency in code and adequate testing.

2. API Documentation

Download the HTML file from the below link and open:

https://github.com/zhtyolivia/UCLA-CS130-Project/blob/main/backend/src/api/API_documentation/index.html

3. Test Cases

Frontend Testing Strategy

Our project employs Jest along with React Testing Library to execute a comprehensive frontend testing strategy. This combination enables us to simulate user interactions with our React components in a controlled testing environment, ensuring that the application behaves as expected under various conditions.

In our [test suite located at App.test.js](#), we focus on verifying the correct rendering and functionality of key components within our application. This suite serves as an integral part of ensuring our app's reliability from a user's perspective.

Test Suite Breakdown

App Component Rendering: The initial test confirms that the App component loads without crashing. This is a fundamental test to ensure the app's root component can start successfully.

```
it('renders App component without crashing', () => {  
  const div = document.createElement('div');  
  const root = createRoot(div);  
  root.render(<App />);  
});
```

Figure 7. A basic render test to ensure the main App component mounts without errors.

DriverSignup Component: This test checks if the DriverSignup page renders correctly within the app's routing context. It ensures that users can navigate to the signup page without issues.

```
it('renders DriverSignup component without crashing', () => {  
  const div = document.createElement('div');  
  const root = createRoot(div);  
  root.render(  
    <BrowserRouter>  
      <DriverSignup />  
    </BrowserRouter>  
  );  
});
```

Figure 8. Validates that the DriverSignup component is rendered correctly within the application's routing.

GoogleOAuthProvider Integration: A critical part of our authentication flow involves Google OAuth. We mock the OAuth provider to test if the signup process behaves as expected, focusing on rendering the "Signup as Driver" button to indicate successful integration.

```

it('renders expected text content in DriverSignup', async () => {
  render(
    <GoogleOAuthProvider clientId={mockClientId}>
      <BrowserRouter>
        <DriverSignup />
      </BrowserRouter>
    </GoogleOAuthProvider>
  );
  expect(await screen.findByText('Signup as Driver')).toBeInTheDocument();
});

```

Figure 9. Tests the GoogleOAuthProvider to verify that the "Signup as Driver" button is correctly rendered for user authentication.

Frontend Manual Testing

Due to limitations in automatic testing (e.g., when testing GET and POST requests, qualitatively checking rendering results, and Google credentials), we performed a series of manual testing to ensure the correctness in the frontend. To ensure that any change in the code does not affect the rest of the application, all developers need to go through the following steps:

1. Log in with a test passenger account.
2. Check the rendering of the home page, driver post (rideshare) page, and profile page.
3. Check if profile info can be updated.
4. Accept/cancel a join request.
5. Log out.
6. Log in with the driver account associated with the join request.
7. Check the rendering of the driver home page and profile page.
8. Check if profile info can be updated.
9. Check if the driver is able to see the join request.
10. Log out.

To ensure correct integration of Google Auth, all developers perform the following tests to guarantee the proper initialization and update of user accounts:

1. Log in with a passenger/driver account.
2. Initiate rideshare (in the case of a driver) or send a post (passenger). Go to the database/switch account type/view log to ensure the rideshare/post was successfully sent.
3. Go to the profile page and check if profile information is correct and is able to be updated.
4. Log out.

The two manual tests above should have the behavior as expected or detailed in our user manual. The combination of automatic testing and manual testing ensures the comprehensiveness of the testing process. Besides, we also ask all developers to document how to reproduce an error in the frontend to facilitate the troubleshooting process.

Backend Testing Strategy

For our backend testing, we have the automatic test combined together with the front end to ensure the connection between frontend and backend. However, we utilized more on manually testing with backend API instead of automated test since most part of development was rather incremental. We are using JWT and bcrypt for user security and authentication purposes, so most of the backend features would require a token to access. This adds a lot of difficulties to our testing. So we mainly used Postman to test every backend end-points. With the use of Postman, we can make sure that we covered all extreme test cases for each feature we added. Since our project mainly focused on certain APIs with each API containing different end-points, this manual testing approach would be the best choice for us to do. Here is an example of how we utilize Postman to test backend services.

POST driver Sign in:

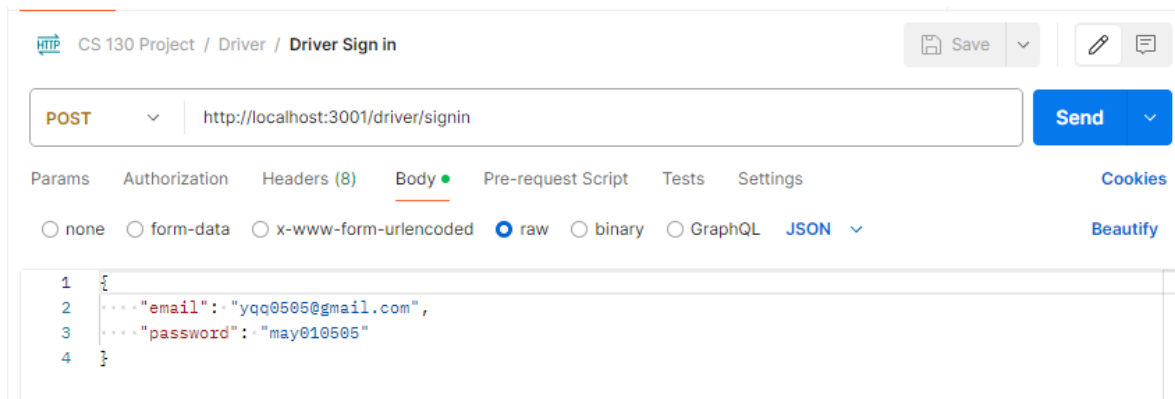


Figure 10: Driver Sign in Postman Test

By passing a correct email and password that is saved in MongoDB, accessing /driver/signin page will successfully login and generate a token:

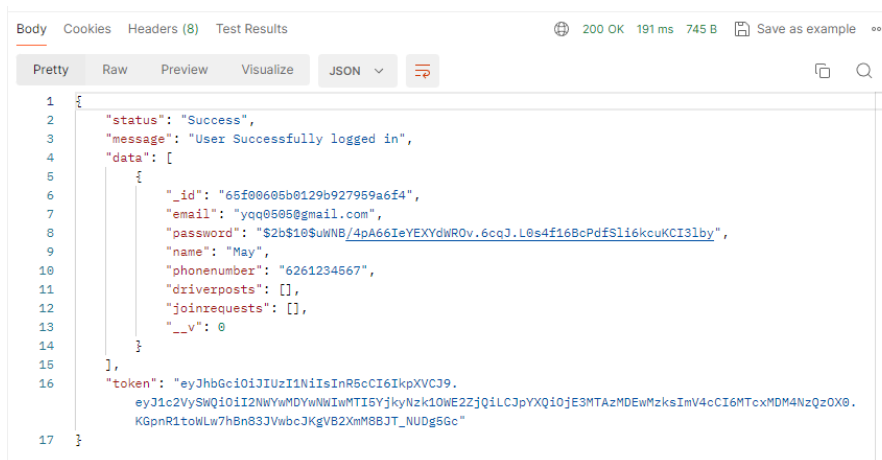


Figure 11: Driver Sign in Postman Test Result

Then if we want to test other end-points that require the user to login, we will utilize this token to pass to the authorization part.

GET driverpost Search:

Search API requires a search term, so first we would set up the search link in Postman, and set the term under the param function:

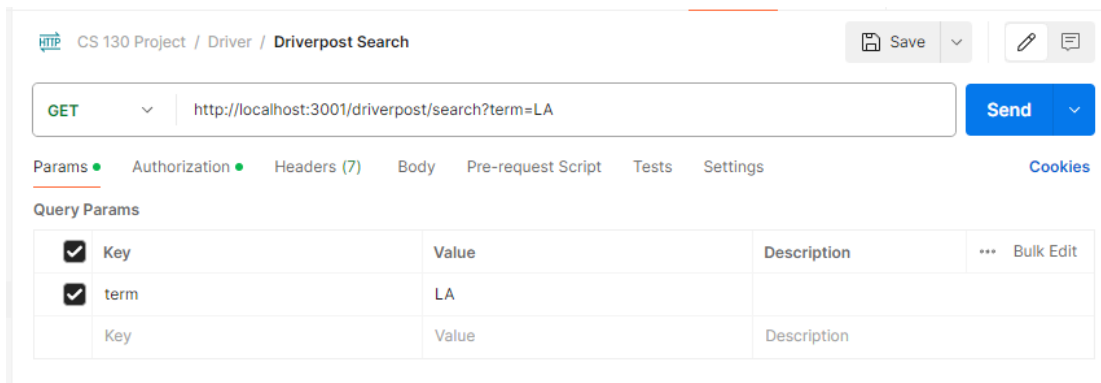


Figure 12: Set param for search link

Then we would need the token that we get from user sign in to pass to this search end-point for it to work. Else without this authorization, swiftlink app will not allow user to view any posts without sign in for security purposes:

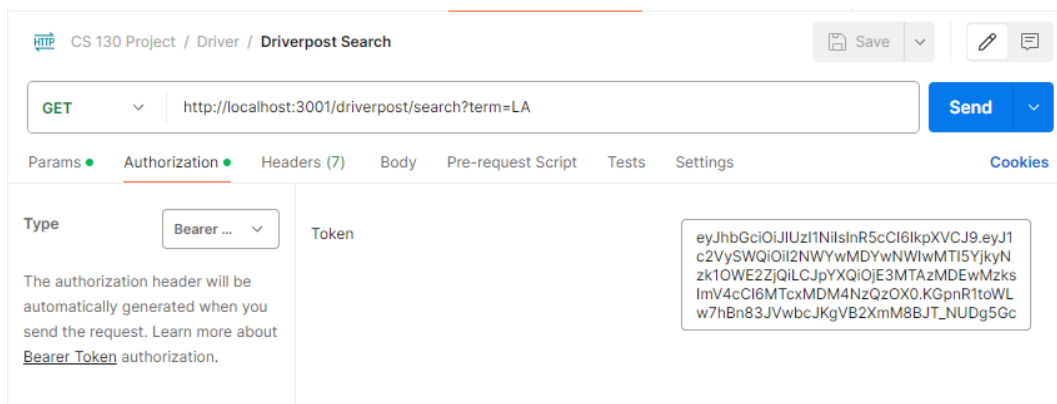
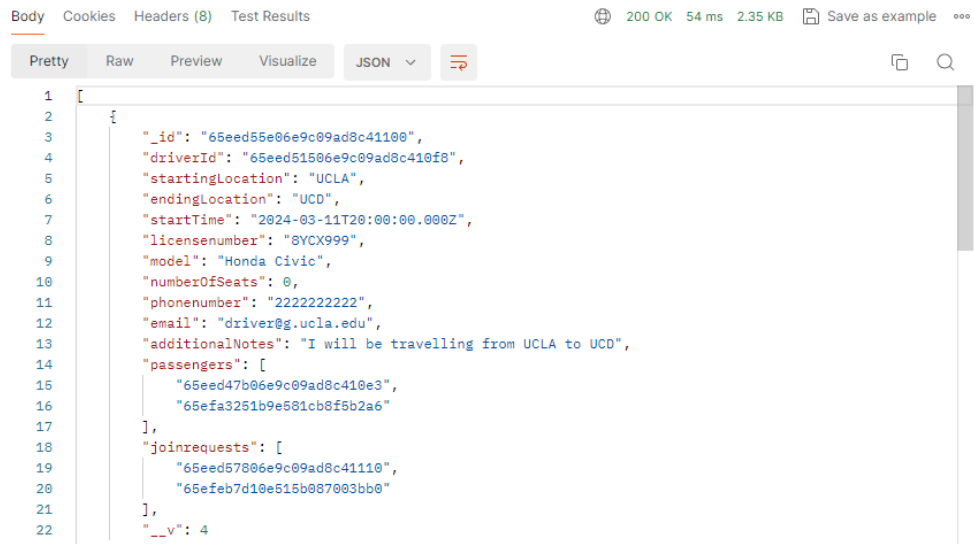


Figure 13: Set token for search link

Then after everything is set, now can click on send button and test if the result will be correct or not:



```
1 [
2   {
3     "_id": "65eed55e06e9c09ad8c41100",
4     "driverId": "65eed5106e9c09ad8c410f8",
5     "startingLocation": "UCLA",
6     "endingLocation": "UCD",
7     "startTime": "2024-03-11T20:00:00.000Z",
8     "licensenumbr": "8YCX999",
9     "model": "Honda Civic",
10    "numberOfSeats": 0,
11    "phonenumber": "222222222",
12    "email": "driver@g.ucla.edu",
13    "additionalNotes": "I will be travelling from UCLA to UCD",
14    "passengers": [
15      "65eed47b06e9c09ad8c410e3",
16      "65efa3251b9e581cb8f5b2a6"
17    ],
18    "joinrequests": [
19      "65eed57806e9c09ad8c41110",
20      "65efeb7d10e515b087003bb0"
21    ],
22    "__v": 4
23  ]
24 ]
```

Figure 14: Success output for search link

Now the testing for Search API is done.

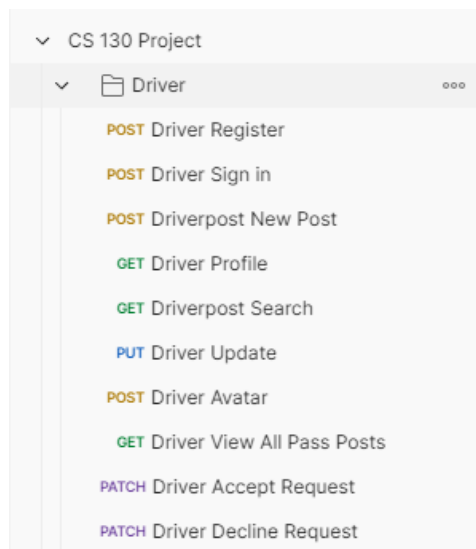


Figure 15: Driver Test Cases for this Project

We have also created a folder for this project with different test cases listed, so that we don't need to create test cases over and over again. All we need to do is to send the requests with the correct token. This is the example of how we test our backend with Postman.

4. CI/CD

We utilized Github Action, Heroku, and netlify to automate our CI/CD process, with details explained below.

Scripts

To compile our project, one should 'cd frontend' and run 'npm run build', where build is not applicable for backend based on our implementation. To thoroughly test our project with existing test cases we provided, one should 'cd frontend' and run 'npm ci' and 'npm test' which would install clean dependencies and run tests for front end implementation. Similarly, to test our packend, one should 'cd backend' and run 'npm ci' and 'npm test' which would install clean dependencies and run tests for backend end implementation. Both tests invoke the Jest framework. For package and deployment of our code, we connected our project to heroku(backend) and netlify(frontend) with Github Actions, which will be described below.

CI Pipeline

To facilitate continuous integration, we used Github Action as our CI. In the github repository, this is reflected by the document (<https://github.com/zhtyolivia/UCLA-CS130-Project/blob/main/.github/workflows/CI.yml>). In this CI script, the workflow is mainly divided into two steps. Step one prompts a backend test after installing all backend dependencies. Step two prompts a frontend test after installing all frontend dependencies. With this construction, we ensure that for each time there is a push or pull-request to the main branch, we run the new version of code through the test package to ensure that it meets the testing requirements. The new version of code, if it includes modification in backend code, after merge to the main branch, will trigger Heroku to automatically deploy the newer version of backend after passing the CI test we have. Similarly, we also set up netlify to automatically redeploy any change made to frontend code that has passed our CI tests and merged to the main branch. The entire setup ensures a seamless continuous integration and deployment flow with assurance that false changes to the original code will not contaminate the deploy version.

Related Links

Backend package.json:

<https://github.com/zhtyolivia/UCLA-CS130-Project/blob/main/backend/package.json>

Frontend package.json:

<https://github.com/zhtyolivia/UCLA-CS130-Project/blob/main/frontend/package.json>

5. User Manual

The user manual of this application can be found in our Github wiki page:

<https://github.com/zhtyolivia/UCLA-CS130-Project/wiki>.

6. Scrum Summary

We hold stand-up meetings three times a week: on Monday, Wednesday, and Friday. We have more frequent meetings than we planned in Sprint 1, which helps us stay on track and ensure everyone is on the same page. In addition, we kept meeting notes for each meeting so that we could review them if any of us were absent or needed to refresh on what was discussed.

We conducted a Scrum Retrospective after each sprint to discuss what went well and what could be improved, which helped us refine our processes and make a more efficient plan for the next sprint. Additionally, we schedule tasks from the product backlog every day, keeping the team focused and productive. Moreover, we have more organized team documentation to make the whole process efficient and have smooth teamwork.

Sprint 2 Summary (2/23/2024)

During the second sprint, our team achieved significant milestones. We successfully developed the API for creating posts and initiating ride-share requests. Additionally, we completed the feature that allows users to view their profiles and implemented the functionality to search for posts. These achievements marked substantial progress in our project, enhancing user interaction and engagement.

Sprint 3 Summary (3/5/2024)

In the third sprint, we introduced a notification feature, enabling users to send and receive requests to join a ride share. We also developed the logic for managing available seats, including reserving and subtracting them, along with related CRUD operations for the backend. This sprint focused on improving the application's functionality and user experience, making it more dynamic and responsive to user needs.

Stories and Epics

The completed list of epics and user stories of Sprint 2 and 3 are listed below:

Sprint 2:
As a passenger, I would like to view and edit my information
As a passenger, I would like to browse and view rideshares
As a passenger, I would like to send and manage join requests.
As an user, I would like to register to join a ride share

[As an user, I would like to initiate a ride share](#)

[As a driver, I would like to initiate new rides](#)

[As a driver, I would like to initiate new ride share](#)

[As a passenger, I would like to be able to search through all the posts so that I could found the posts that I wanted.](#)

Sprint 3:

[As a passenger, I would like to tell drivers about a ride I'd like to see.](#)

[As an user, I would like to receive notification when another user send me an join request.](#)

[As an user, I would like to receive email notifications once there is an update on my ride status](#)

[As a new user, I want to be automatically logged in after I sign up using Google, so I can start using the platform without unnecessary delays.](#)

[As a user, I want to sign up using my Google account, so that I can avoid filling out lengthy registration forms.](#)

[As a driver, I want to initiate rides and log in to my account.](#)

[As a driver, I want to receive notifications and handle join requests](#)

[As a Driver, I would like to view my own profile and see detailed information about my ridesharing posts](#)

[As a Driver, I would like to view my own profile and see detailed information about my ridesharing posts](#)

Contributions of Each Team Member

Ethan Jiang focused on frontend development, aiming to enhance driver user experience. He was responsible for creating the welcome page to introduce users to the application, and he aided in developing the login page. His key achievements include three parts: implementing the driver's home page, the main portal for drivers to engage with the app's features, and crafting the driver profile page for personal information management. Furthermore, he established connections between the driver pages and backend endpoints, managing API requests for optimal app responsiveness. He also designed the 'initiate ride' page, streamlining the process for drivers

to start and manage rides. Ethan led the front-end testing, employing Jest and React Testing Library to build comprehensive test suites, ensuring accurate component functionality and seamless backend integration.

Olivia Zhang focused on frontend development, backend deployment, and user manual documentation. Her contribution to the frontend development is focused on the passenger's user experience. This includes two aspects: UI design and handling API requests. She oversees the UI design of all pages and components on the passenger side. While Rick helped with handling fetch requests for the passenger home page, which involves fetching all rideshares, Olivia handled the API requests for the rest of the pages herself. These pages include passenger log-in, passenger sending post, passenger profile (including edit profile), passenger viewing rideshare, passenger managing join requests, etc. pages/features. She is also responsible for deploying the backend of this project using Heroku. After deployment, she worked with Mike to ensure the CI was integrated with the deployment process by making sure that automatic deployment of backend services waits for the CI to complete. Finally, Olivia cleaned up the source code on the frontend side, constructed a README for users and developers, and established a driver user manual in the repository wikis page.

Mike Shi focused on backend development, his contributions to the project were vital, particularly in managing the database and developing the backend schema and APIs with a focus on driver functionalities like join requests and posts. These APIs include user signup/signin, join-request initialization/management, driverpost initialization/management, object-communication logic, etc. He established a continuous integration pipeline to maintain the codebase's integrity and spearheaded the unit testing of APIs, ensuring robustness through regression testing after significant updates. His efforts extended to assisting with the deployment, helping launch the frontend on Netlify and the backend on Heroku, and streamlining the process for users to access the service effortlessly.

In the project, **May Wang** focused on the backend of the project. Her primary focus was on the passenger side, where she put a lot of thought into creating a smooth user experience. She built a search API that enables passengers to find rides quickly and effectively, considering various search parameters to return the most relevant results. The passenger side of the application required attention to detail, and she worked a lot to handle profile management, join rideshare, and communication processes. This required a deep dive into the backend logic to ensure that all the information would be sent successfully to MongoDB. She developed the avatar feature for both drivers and passengers, providing a unique visual identity for users. She also contributed to the updates of passengers, enhancing the app's functionality to enable drivers and passengers to view each other's posts and increase transparency. May also did the whole notification part of the application, where she added the notification for passengers to receive an email notification when the driver accepted/rejected the notification, and the driver received an email notification

when a passenger requested to join or cancel a ride share. Lastly, May contributed to the deployment of the project, where she used Netlify to deploy the frontend of the project for people to use a link to access the project instead of having to download it from GitHub and run it locally.

Rick Yang focuses on improving team efficiency and project quality. He fostered a learning environment and efficient teamwork space by introducing Notion for recording meeting notes and organizing team documents, which provided easier access to information. Besides, he implemented the ApiDoc module for auto API documentation, enhancing collaboration between the frontend and backend teams. For the coding part, Rick serves as a fullstack engineer, aiming to bridge the gap between frontend and backend development. He set up the local development environment to connect both sides of the application, which ensured seamless integration and functionality. In particular, he implemented the 'view all posts' feature, which included developing the necessary endpoint APIs as well as the frontend logic to fetch and display data from the endpoints. Rick also led the integration of Google OAuth 2.0 Sign-in/up, improving user experience and authentication. Moreover, Rick undertook the code optimization effort, recognizing the need due to our codebase's separation into driver and passenger modules by different developers. A significant overlap happens in code structure due to the separation, especially in authentication processes like login and signup. By refactoring these processes, Rick made the modules more maintainable, directly addressing the redundancy and enhancing the scalability.

7. Conclusion and future work

In conclusion, throughout the development of this project, we successfully built an MVP of the SwiftLink app. We strictly adhered to the Scrum process and closely collaborated. In our sprint reflections, we have distilled valuable lessons and identified exciting directions for future work. Moving forward to the next sprint, we aim to enable in-app chat functionality to facilitate better user interactions, integrate the Google Maps API for enhanced navigational features, and explore in-app transaction capabilities.