POOLERZ APPLICATION FOR FAMILY CARPOOLING

Tyler Cady, Annie Vallamattam, Matthew Dworkin, Maya Narayanan, Ignacio Galindo
JID 4301

Client: Lillania Shah, Nandu Shah

Repository: https://github.com/tylerrcady/JID-4301-Poolerz

Table of Contents

Table of Figures

Terminology

Back-end: Part of the application that cannot be accessed by the user

Front-end: Part of the application that the user interacts with

HTML: Hypertext Markup Language; the standard markup language for structuring web pages and their contents

JSON: JavaScript Object Notation; a simple, readable format used for storing and transporting data

MERN Stack: MongoDB, Express, React, and Node.js; a JavaScript software stack used for building dynamic sites and web applications

MongoDB: an open-source database management system storing data in flexible formats for scalable applications

Next.js: a React framework primarily used for simplifying React web applications

Node.js: a JavaScript runtime environment primarily used to manage back-end services

OAuth: Open Authorization; an authorization framework allowing users to grant third-party access to their information without sharing their passwords

PII: Personally Identifiable Information; information connected to an individual that can be used to uncover that personal identity

RBAC: Role Based Access Control; a security method granting users access to systems, applications, and data based on their roles within an organization

React: a JavaScript framework primarily used for building user interfaces for web applications

SD: sequence diagram; a diagram used to depict the processes and objects involved and the sequence of interactions needed to carry out functionality

TypeScript: Strongly typed object-oriented programming language which is a syntactic superset of JavaScript

Introduction

Background

Our developed application is a web application for Poolerz, an organization seeking to transform carpool planning, save time, and build a community among parents while keeping children safe on their journeys. This application implements a straightforward, one-time registration for inputting scheduling information and availability, a matchmaking algorithm to sort dependents into carpooling groups, and a calendar dashboard for viewing schedules and events. The algorithm will factor in location proximity, scheduling compatibility, and personal preferences to create an optimal schedule for the user. Our project is constructed using Typescript, React, HTML, and CSS, allowing for an application that is responsive and dynamic on all platforms.

Document Summary

The **System Architecture** section provides an overview of the static and dynamic structure of our system's back and front-end and database connection, highlighting the interactions between layers and components of our application. The static diagram visualizes the levels of the program and their general relationships, whereas the dynamic diagram displays the specific links between elements as they work to perform a specific scenario task.

The **Data Storage Design** section describes our documented stored data in our database, as well as the security concerns taken into account and addressed throughout our system. The document-database diagram outlines the relationship between three major entities in the application, while the supporting JSON documents exemplify the prior entities and their attributes.

The **Component Design** section details an in-depth view of the system architecture as broken into its thorough components. The detailed static diagram depicts different

connections and interactions between the frontend, backend, and database layers, while the dynamic diagram designates the classes and methods that interact in real-time when a user goes through a certain flow.

The **UI Design** section presents the central screens of our interface that users interact with. Flows and frames are shown along with associated heuristics and reasoning to explain design choices within the application.

System Architecture

Our application is structured using a 3-layer architecture, two of which are enveloped by our web application, providing us with flexible development and easier building through the use of technologies such as React and Typescript. We also chose this structure for a clean transition during project turnover, ensuring that future teams can easily understand the design and layer separation. The static and dynamic architectures seen below explore more of the interactions within our system.

Rationale

The architecture for our project was planned in order to create an efficient way to build out our dynamic application. By keeping a clear separation between the different levels of our architecture, we were able to outline the tasks and processes each section would manage during runtime.

We utilize Google Sign-On using OAuth 2.0 as our means to authenticate users. This method relies on the robust security infrastructure of Google with multi-factor authentication. As a result, no sensitive password information needs to be stored, reducing vulnerabilities regarding password reuse and database breaches.

Static

Our static system architecture for this application was based on our goal to develop a responsive interface that provided an efficient way to create a full-stack web application. We chose to delineate between the front-end and back-end, both within the web application and database layers to clearly diagram the three main sections of system design and represent their interactions.

Figure 1.1 shows our web application, the main part of our system, which encompasses both the front-end and back-end processes of our product. Utilizing half the MERN stack,

we employed React and Node.js, as well as other web technologies like Typescript and HTML to create a seamless desktop application. Our User Interaction level makes use of the aforementioned front-end languages and libraries to develop a program easy for users to work with. On the Business Logic layer, Node.js is used to handle our server-side logic and interact with the database storing user information, as well as call on the optimization algorithm to gather information on carpool groupings, updating the front-end respectively. Outside of the web application is the final layer - the database. Our system engages MongoDB to hold and keep track of user information, including their profile information and associated carpooling organizations and groups.

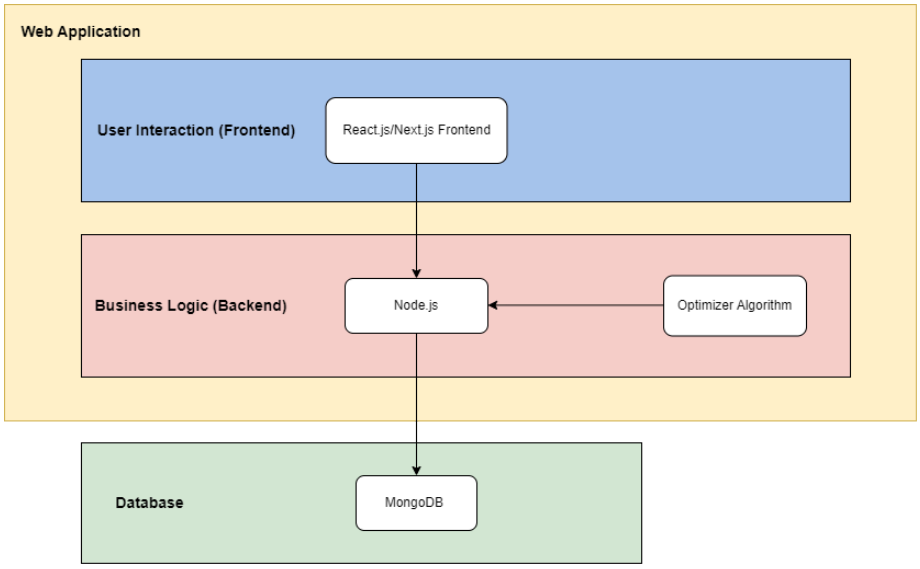

Figure 1.1 - Static System Diagram

Dynamic

Our dynamic system architecture depicts a user scenario in the case of creating and maintaining ownership of a Carpool Group. The system sequence diagram drawn out shows the key interactions of the different components of our application as they work to create the seamless experience the user encounters.

Figure 1.2, shown below, demonstrates the runtime flow when a user begins the creation of a Carpool Group. On initiation, the user is greeted by the interface, in the User Interaction layer, which is ready for front-end interaction and connection to the back-end components of the system. In the case of a user choosing to view and create a carpool, the User Interaction layer takes in the user's requests and passes the tasks of gathering the user's current information and carpool data to the back-end. From this, the Business Logic layer initiates the gathering of user information and carpool data from the database and passes it back to the interaction layer. The back-end works to query the data storage and return information about current carpools or members within the carpool and runs the optimization to sort carpool members into driving groups. After this information is sent back, the interaction layer can update the interface. The database layer is not within the web application itself, utilizing MongoDB instead, and manages the maintenance of the user's information. The database is consistently storing and updating user data, in this use case specifically fetching and returning the user's current carpools and returning data like location and availability on all members of a carpool group for the back-end optimizer to run.

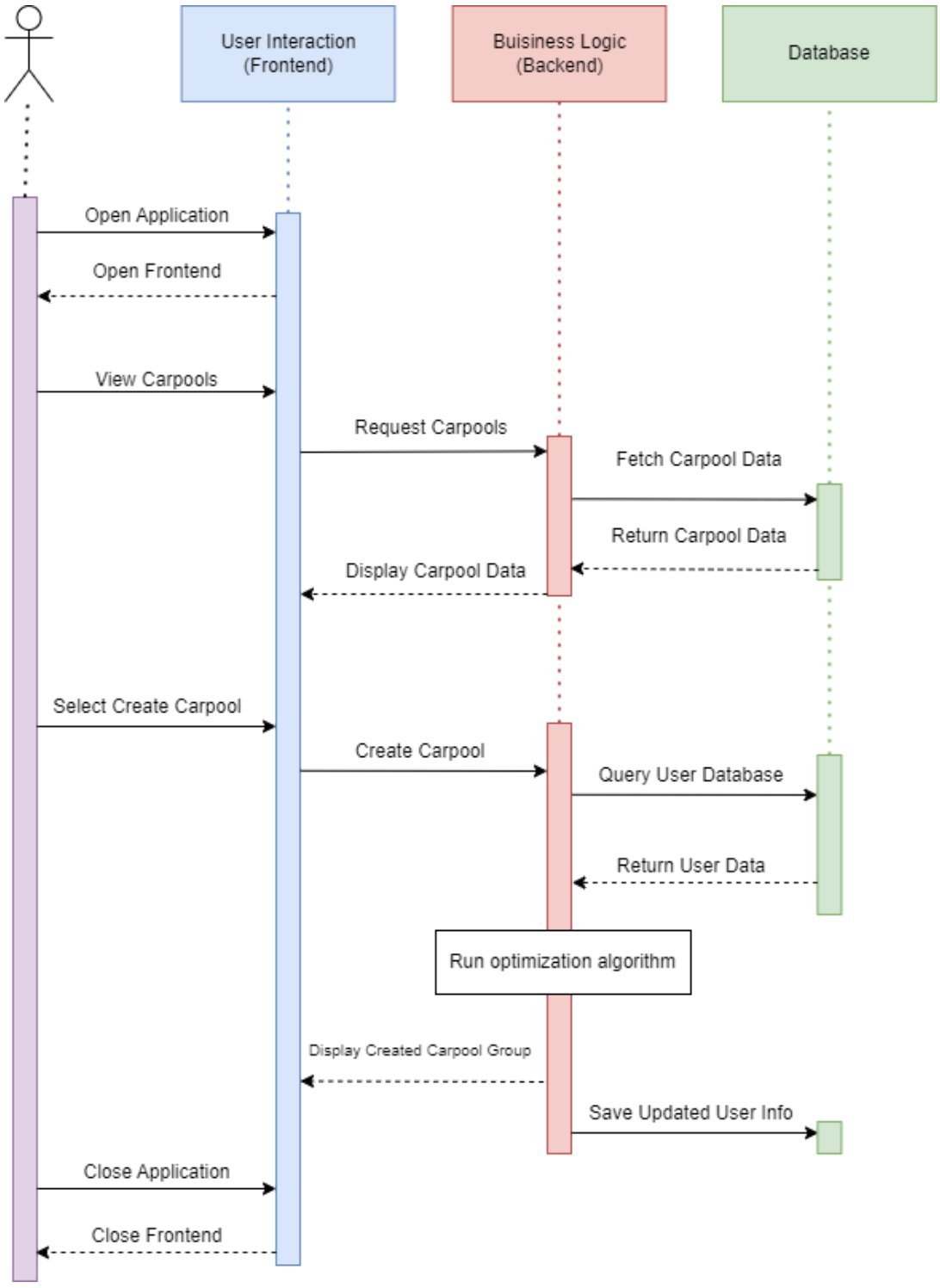

Figure 1.2 - Dynamic System Diagram

Data Storage Design

The figures below demonstrate our data storage for our two collections of users and carpools with the application. Because our system does not use SQL, but rather MongoDB, it was more logical to utilize a version of the document-database diagram in order to easily translate our database. Each box represents a document with its related attributes, with arrows connecting each document in their relationships.

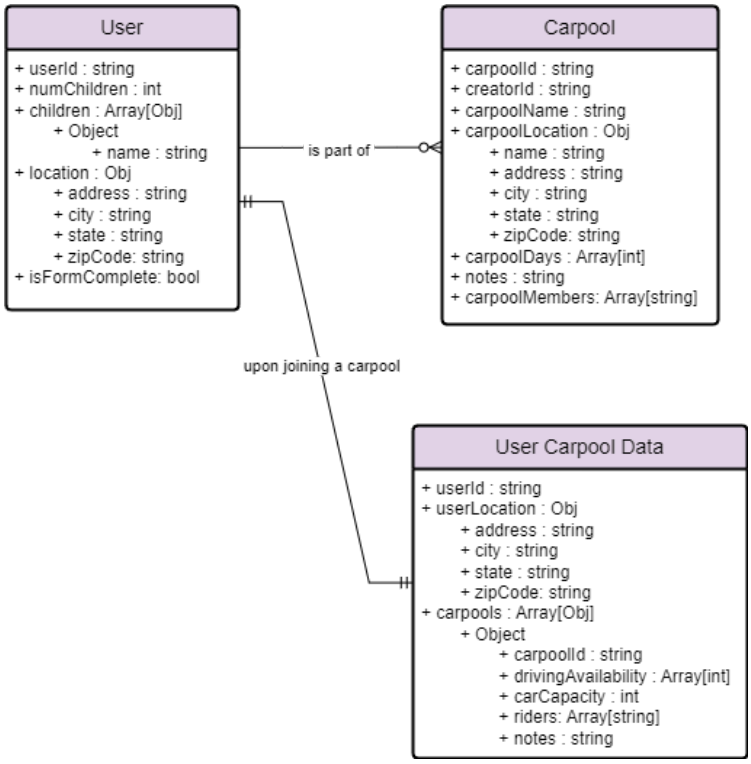

Figure 2.1 - Data Storage Diagram

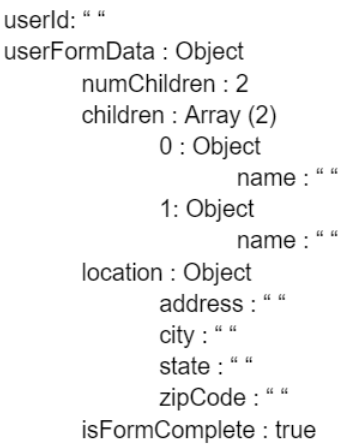

Figure 2.2 - User JSON Document

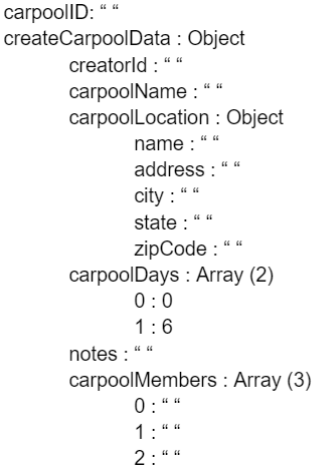

Figure 2.3 - Carpool JSON Document

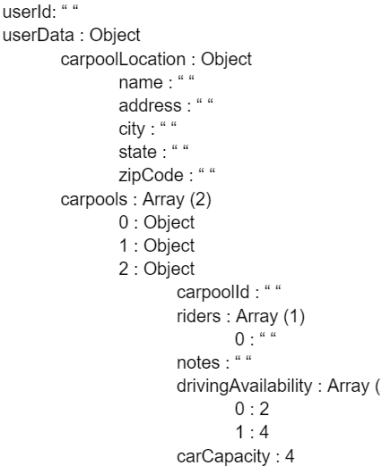

Figure 2.4 - User Carpool Data JSON Document

Users: User is a document containing information stored within the user’s profile and unchanging information. Their unique ID is taken, along with an Object containing information on their children, location, and whether they have successfully finished the registration form. Any information that needs to be edited, such as changing their address or adding a child to their profile, will be reflected in the document.

Carpools: Carpool is a document containing all information related to a carpool organization. This includes the unique carpool ID, the name and ID of the carpool org creator, and the location, time/days, and members associated with the carpool org. As more members join the organization, the document will add their user ID to the members array.

User Carpool Data: User Carpool Data is a document related to the user containing all the information connected to the carpools the user is associated with. This appears as the user’s ID and location, along with an array of carpool objects containing the user’s

information for that carpool (carpool ID, riders, driving availability, notes). Each time a user joins or creates a new carpool and adds their information, this document will be updated with the additional data.

Protocols & Security

Our application does not deal with any files; however, we make use of JSON to exchange information between the client and the server. JSON is used to organize user-input data into documents in MongoDB, enabling efficient and easy querying. Furthermore, MongoDB's in-built encryption features are employed to secure stored data.

By default, all data exchange in our web application is secured using HTTPS to ensure encrypted and safe communication. Users are required to log in to the application to restrict access to sensitive features such as carpool scheduling, location sharing, and user profiles to authorized individuals only.

The primary piece of sensitive information or Personally Identifiable Information (PII) collected is the user's profile details, specifically their email address and location. This data is protected by Google's OAuth authentication, which takes advantage of Google's high level of security. Because of this, passwords do not have to be stored in the application, and email addresses and names of users are non-editable. Location data is also accessible only to specific users in a carpool group and is not retained longer than necessary for ride coordination. Access to PII is also limited using role-based access control (RBAC) so that only legitimate users can access their respective information. Apart from PII, the app does not process or store financial information, resulting in fewer vulnerabilities.

Component Design

The diagrams depicted below show the static and dynamic architectures in more detail, highlighting relationships, dependencies, method overviews, and other components. The in-depth static diagram builds on the general architecture system (**Figure 1.1**) and breaks down the three aspects of the application: frontend, backend, and database. Within each section, interactions between each are illustrated during the creation or joining of a carpool. The microview of the dynamic diagram ties into the static diagram and demonstrates an effective representation of both previous actions. Following each path, responses between each layer and user displays are shown. The diagrams as a whole combine to provide a comprehensive view of our application's architecture.

Static

The following static component diagram details the specific aspects of our system, building on our prior static architecture. The utilization of a class diagram over a component diagram aids in clearly outlining the different elements of our application and how each class/section interacts or calls upon each other within and across certain layers.

As seen in Figure 3.1, the Frontend layer contains multiple components that a user interacts with, including Profile, Modal, JoinCarpool, and CreateCarpool. These elements take care of displaying interfaces and allowing the user options, interacting with the backend to take care of the functionality of the components. The <<uses>> and <<creates>> annotations show the aspects that specifically require or produce a certain next element. The Business Logic layer holds sections for the server and optimizer, carrying out tasks related to pulling and updating user or carpool information and grouping households into driving groups. The database is called on from the backend in order to update the information for Carpools collection and UserData collection, which hold our data for carpools and their associated members, location, days, etc., and users and their profile information.

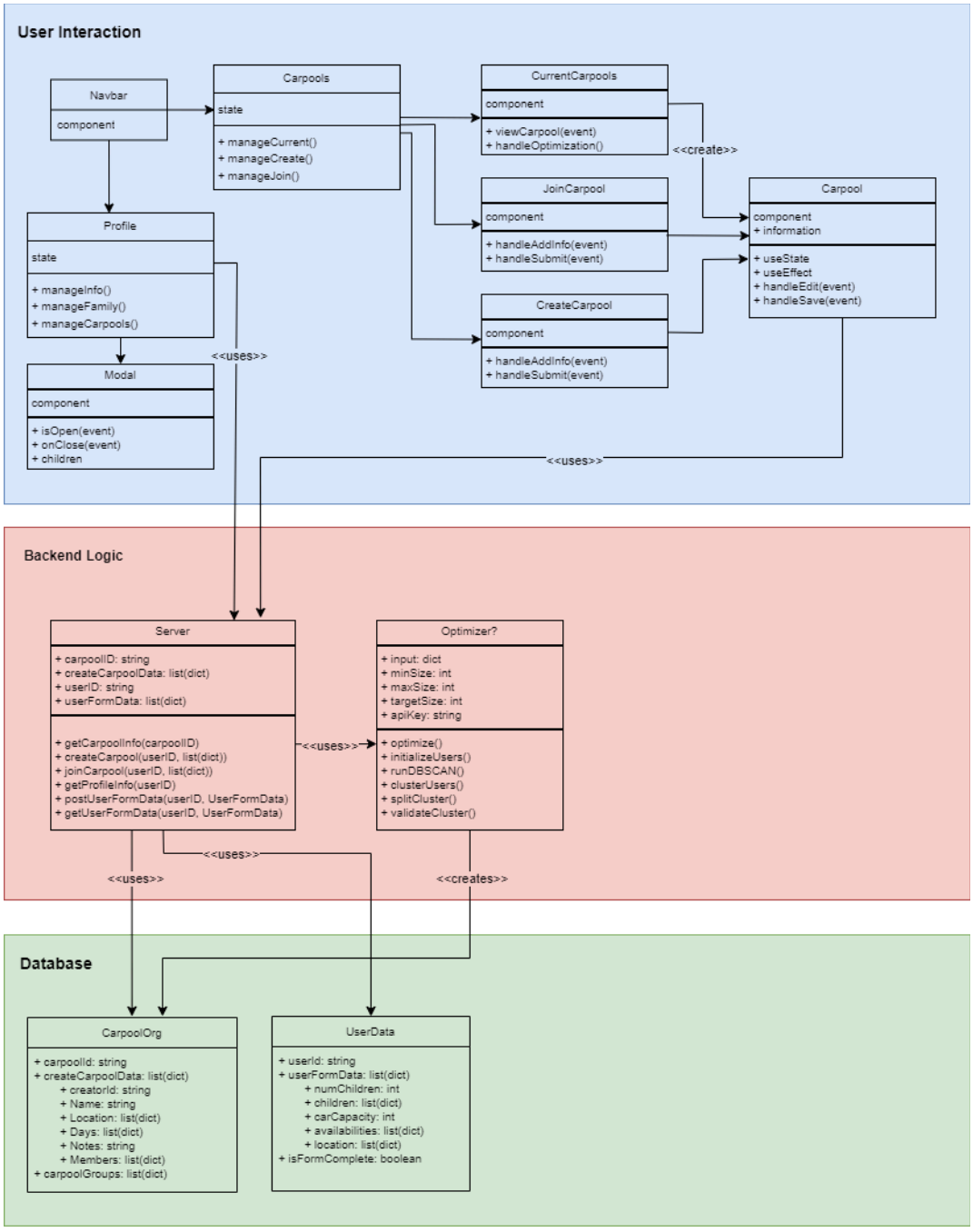

Figure 3.1 - Static Component Diagram

Dynamic

Figure 3.2 as shown below depicts the user flow of joining, creating, or viewing a carpool as shown through an interaction overview diagram. The usage of an interaction overview diagram allows us to show the main pathways of user connections in a way that is

delineated and easy to follow. It also allows for displaying multiple sequence diagrams together, nested in different flows.

The following diagram walks through multiple options including sequence diagrams of different components within the User Interaction, Backend, and Database layers. In the case of joining a carpool, the SD shows the user information being passed from the UI to the backend layer, where they are then added to the carpool database and a confirmation of joined carpool is returned. Another option follows creating a carpool, where a user can input information on the frontend, which is then transferred to the backend and works with the database to create a carpool to track. The user data is then returned and a confirmation of creation is displayed. The interaction for viewing a carpool can come from the main page, or after creating or joining a carpool. In this, a user clicks on a current carpool, where the UI layer interacts with the backend to request and query the respective information and return it to display on the frontend.

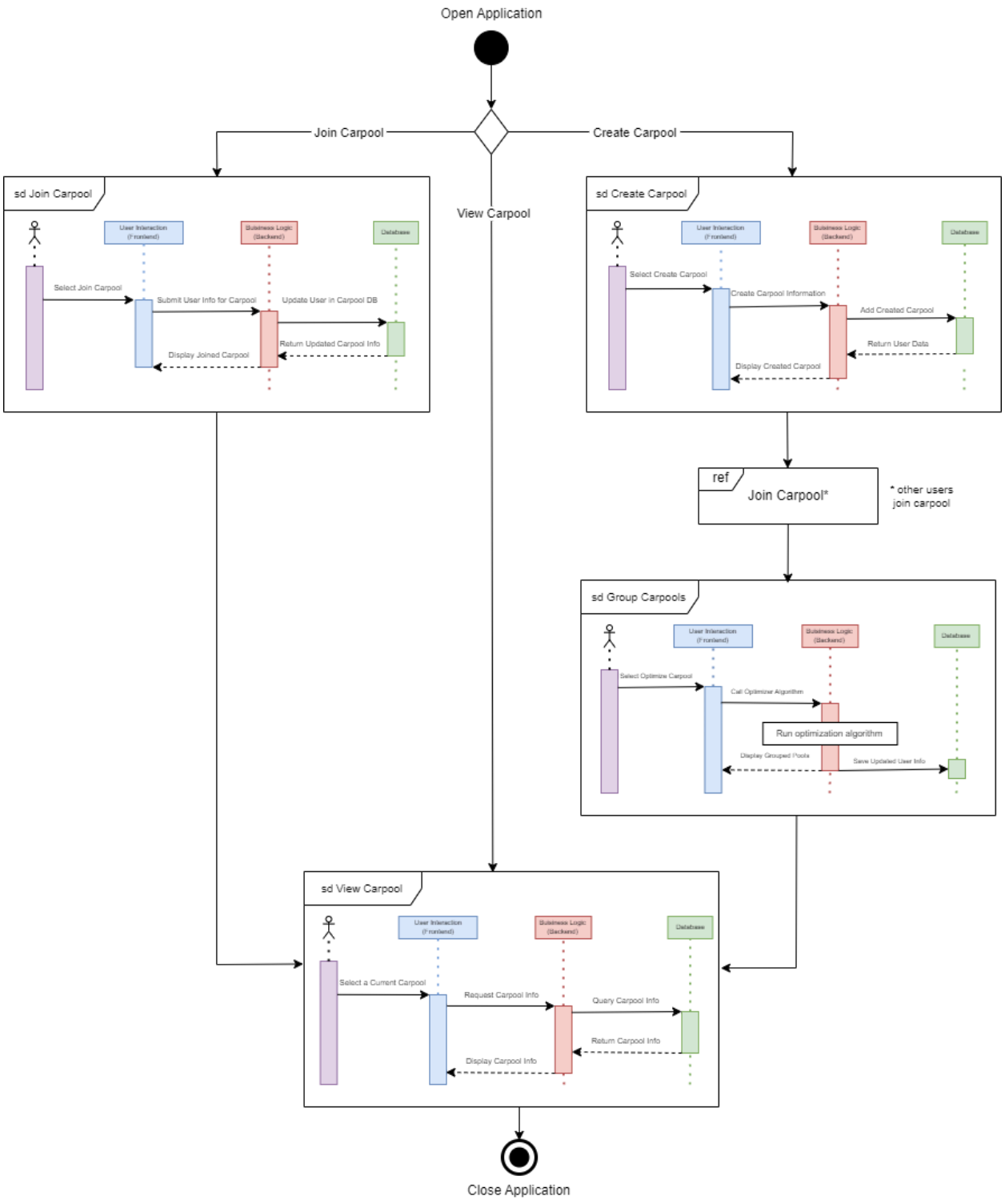

Figure 3.2 - Dynamic Component Diagram

UI Design

Our application features three main sections of interface - carpools, profile, and dashboard - which contain the primary functions of entering, optimizing, and viewing a calendar of events. Below is a detailed view of selected screens as well as an explanation of reasonings behind design choices. These interfaces and flows will focus on the main interactions on the application, the process of joining or creating a carpool, and the ability to view carpool information and optimize groups.

Figure 4.1 - Carpooling Home Page

The screen depicted above encompasses the main functions of our application - creating, joining, and viewing carpools. On this page, users can choose to add themselves to an existing carpool organization or make a new one. They can also easily view all the carpool organizations they are a part of, with a subtitle denoting whether they are the administrator of the carpool (if they created it) or simply a member.

This page utilizes many of **Nielsen's Usability Heuristics**, primarily in its minimalist design, match to real-world, and consistency.

Aesthetic and Minimalist Design: Our interface implements a clean and uncluttered design, displaying only the essential options that users would want to interact with. The limited color palette of Poolerz branded blue and yellow creates a clear visual hierarchy and ensures readability, allowing focus on primary actions. Our use of white space and tabs was used to effectively create a balanced layout without any unnecessary elements.

Match Between System and Real World: The terminology on our screen - ‘Create Carpool’, ‘Join Carpool’, ‘Current Carpools’ - is reminiscent of real-world concepts and phrases, making it intuitive for users to understand. The arrows next to each carpool within the Current Carpools section also mimic familiar navigation patterns used in other applications, which commonly signal to click for more information.

Consistency and Standards: The buttons and text styles used on this screen follow a uniform design across other parts of our application, ensuring users can predict interactions and retain their understanding of navigation within the site. The placement of the navigation menu follows a common pattern seen across many web applications. The ‘Create’ and ‘Join’ sections and buttons are visually consistent, reinforcing to users the similarity between their functionalities.

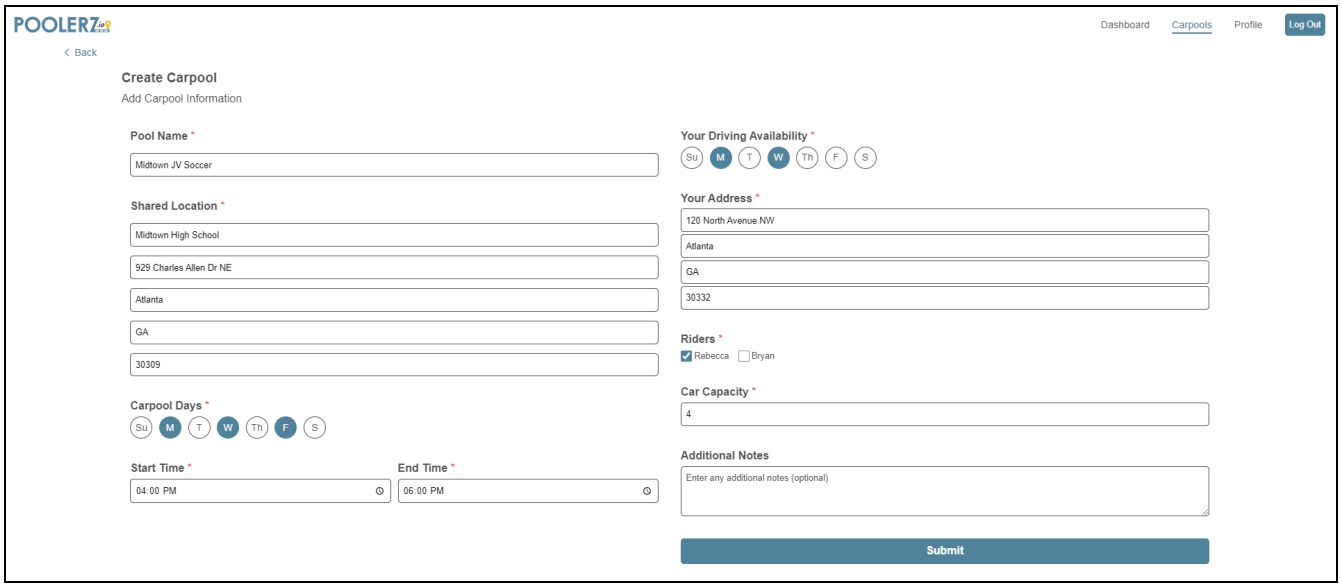

Figure 4.2 - Create Carpool Page

The screen displayed above visualizes the page for creating a carpool organization. On this page, users can input new information for an organization - name, address, time, etc. - as well as add their own information as the first member of the org. This information includes their driving availability, riders, car capacity, and any additional comments or information they would like other people in their carpool to know. The Join Carpool follows a very similar format, with the exception of inputting carpool information, and therefore we did not need to display it. Following the creation of a carpool, the user will receive a join code they can share with others, and the carpool will be added to the Current Carpools section on the Carpools page.

This carpool information page also highlights many of Nielsen's Usability Heuristics, focusing on error prevention, user control and freedom, and system status.

Error Prevention: This page utilizes required fields which are marked with red asterisks, reducing the chance of users submitting incomplete information. The 'Continue' button is also disabled until all required fields are filled, being shown in a translucent blue color as opposed to the brighter color, preventing mistakes prior to submission. The interface also groups related inputs together logically,

with carpool information coming before personal information, and pool time directly above driving availability, which minimizes input errors.

User Control and Freedom: Editable fields for all inputs ensure that users can adjust information before finalizing their pool setup or their information. They can also easily toggle selections for days, riders, and availability, allowing for quick and intuitive corrections and customization. This page also features a ‘Back’ button at the top left, providing an escape for users to return to the previous page without committing to changes. This button also pops up a confirmation modal asking users to confirm they want to return without saving information (also a feature of error prevention).

Visibility of System Status: The layout of this form helps users understand their progress, utilizing clear sectioning for essential details like location, schedule, and car capacity. The selected days for organization times and driving availability are also highlighted, providing immediate visual confirmation and feedback for users to understand their selections and progress.

Figure 4.3 - Carpool Information Page

The screens and flow shown above demonstrate the process of a carpool creator running the optimizer to push out carpool groups to members within the organization. On this page, all users can view the organization information, their personal information within the org, and their carpool, however, organization owners can edit the organization information, run the optimizer, and view all the carpools within the group. The top two screens show the before and after of running the optimizer on the admin side and the resulting pools and unassigned members, while the bottom two screens show the view of general members who can view their carpool once the optimizer has been run.

This carpool information page features Nielsen's Usability Heuristics, specifically in its status visibility, flexibility and efficiency, and recognition ability.

Visibility of System Status: The screens on this page clearly indicate a user's status within the carpooling process. The first screen has clear messaging that no carpools have been assigned, while the second screens provide updates on assigned groups and depict the necessary pool information. The 'Run Optimizer' button also suggests users a way to move forward when no current pools exist, helping the administrators to understand their next steps in the flow.

Flexibility and Efficiency of Use: The structured layout of the page helps both new and experienced users efficiently find relevant information without taking unnecessary steps. Users can quickly edit their information, and owners can edit organization information, ensuring flexible adaptability and customization without navigating to other pages. The 'Run Optimizer' button also allows users to automate their carpool grouping instead of manually processing members into groups, a much more efficient procedure.

Recognition Rather than Recall: Information is displayed on the screen clearly rather than requiring users to remember all information. Users are able to see their assigned groups, driving days, and whether members were assigned or not, in a direct and structured way at all times. Different sections on the page, such as 'Organization Information', 'My Information', and 'My Pool', help users quickly

recognize where relevant details are without having to search or recall previous actions and information.

Appendix

Team Members

Tyler Cady (tcady3@gatech.edu)

- Engineering Manager, engineering task delegations, system architecture, optimization algorithm development, integration, and documentation

Annie Vallamattam (avallamattam6@gatech.edu)

- Project Manager, general task delegation, client communication, front-end development, user interface design

Matthew Dworkin (mdworkin3@gatech.edu)

- Back-end development, database development, test infrastructure, data integration, optimization algorithm output handling

Natasha Narayanan (nnarayanan34@gatech.edu)

- Database management, front-end development, back-end development

Ignacio Galindo (igalindo3@gatech.edu)

- Back-end development, optimization algorithm input handling, database management, front-end development

API Documentation

Code	Text	Description
200	Success	The request was completed successfully.
400	Bad Request	The request is invalid or missing the required data.
401	Unauthorized	The request requires a user to be authenticated or authorized.
403	Forbidden	The server recognized the request but user access was denied.
404	Not Found	The request could not be completed because

		the URL was not recognized.
500	Internal Server Error	An unexpected error occurred on the server.

GET /api/name

Retrieves user names based on user IDs.

Parameters

Parameter	Description
userId	one or more user IDs used to retrieve names

Response Fields

Response	Description
message	JSON object mapping user IDs to names
error	message returned if invalid or missing

POST /api/create-carpool-data

Generates a unique carpool ID for the created pool, saves the inputted data, and commits carpool data to the database.

Parameters

Parameter	Description
createCarpoolData	object containing data for the carpool to be created

Response Fields

Response	Description
message	success message
joinCode	unique carpoolID generated for carpool
error	message returned if invalid or missing

GET /api/create-carpool-data

Retrieves carpool data after accepting the user and its carpoolID to identify the corresponding carpool.

Parameters

Parameter	Description
carpoolId	unique ID of carpool to retrieve
creatorId	ID of user who created carpool

Response Fields

Response	Description
createCarpoolData	carpool data retrieved from database
error	message returned if invalid or missing

POST /api/join-carpool-data

Validates incoming joining data, and commits the carpool and user data to the database.

Parameters

Parameter	Description
joinCarpoolData	object containing data for the carpool to be joined (includes createCarpoolData, joinData, and userId)

Response Fields

Response	Description
message	success message
joinCode	unique carpoolID of carpool that user joined
error	message returned if invalid or missing

GET /api/join-carpool-data

Retrieves carpool data after accepting the user.

Parameters

Parameter	Description
userId	ID of user whose carpool data is being retrieved

Response Fields

Response	Description
createCarpoolData	carpool data retrieved from database
error	message returned if invalid or missing

GET /api/optimization-results

Fetches optimization results from the database after authenticating a user and accepting the carpool ID.

Parameters

Parameter	Description
carpoolId	unique ID of carpool whose optimization results are being retrieved

Response Fields

Response	Description
results	optimization results retrieved from database
updatedAt	timestamp of when results were last updated
error	message returned if invalid or missing

POST /api/save-optimization

Confirms whether the user is the owner of the carpool, then sanitizes optimization results and saves incoming data in the database.

Parameters

Parameter	Description
carpoolId	unique ID of carpool whose optimization results are being saved
results	optimization results to be saved

Response Fields

Response	Description
success	boolean indicating successful save
error	message returned if invalid, missing, or unauthorized

POST /api/update-carpool

Validates incoming carpool ID and data, then updates the carpool database with new data.

Parameters

Parameter	Description
carpoolId	unique ID of carpool whose optimization results are being saved
carpoolData	new data to update the carpool with

Response Fields

Response	Description
success	boolean indicating successful save
error	message returned if invalid, missing, or unauthorized

POST /api/update-user-carpool

Validates the incoming user ID, carpool ID, and carpool data, then updates the user-carpool-data database with new data.

Parameters

Parameter	Description
userId	unique ID of user whose carpool data is being updated
carpoolId	unique ID of carpool whose optimization results are being saved
userLocation	updated user location
carpoolData	new data to update the carpool with

Response Fields

Response	Description
success	boolean indicating successful save
message	success message
error	message returned if invalid, missing, or unauthorized