Electronic Logging Device Application

Michelin Connected Fleet

JIB 4322: Christian Haynes, Ben Davidson, Alex Messersmith, and Andrew Yang

https://github.com/Haynesc3419/JIB-4322-ELD

Table of Contents

Table of Figures

Terminology

Android Studio: An integrated development environment (IDE) used for building Android applications, offering tools for coding, testing, and debugging.

Application Programming Interface (API): A set of rules and protocols that allow different software applications to communicate and interact with each other.

Electronic Logging Device (ELD): A digital system used by truck drivers to automatically record hours of service (HOS) and other work-related data to ensure compliance with FMCSA regulations.

Endpoint: A specific URL or URI in a web service where an API receives requests and sends responses, enabling communication between different systems or components.

FMCSA Regulations: A set of regulations that provides the minimum standards for the operation of commercial motor vehicles in interstate commerce

Load Management Unit (LMU): A device used in fleet management systems to monitor, track, and manage vehicle performance, location, and operational data in real time.

MongoDB: A NoSQL database that stores data in flexible, JSON-like documents, allowing for high scalability and ease of development.

NoSQL: A non-relational database design for the storage and retrieval of data.

Spring Boot: A framework for building Java-based web applications and microservices, designed to simplify setup and development with minimal configuration.

Introduction

Background

Electronic Logging Devices (ELD) are vital applications used by truck drivers to log data related to their work in compliance with FMCSA regulations. These logs include key information such as hours driven, status, and other critical metrics. Our project focuses on developing an in-house Electronic Logging Device (ELD) solution for Michelin Connected Fleet. By taking control of their own ELD system, Michelin Connected Fleet will gain full ownership and the flexibility to customize the product to meet their unique requirements. The application is designed to be used on Android devices and must also be able to connect to the vehicle via Bluetooth to collect vital information. With this in mind, the app is built with an Android Studio frontend and a Java Spring backend, with MongoDB as our database of choice.

Document Summary:

1. **System Architecture:** The System Architecture document provides a high-level overview of the ELD system's structure and its major component interactions, ensuring compliance with FMCSA regulations. It includes both static and dynamic architecture diagrams, highlighting how core components like data logging and driver interaction work together. The static architecture diagram focuses on the system's major components and their logical relationships, while the dynamic architecture highlights runtime behaviors, such as status changes.
2. **Component Design:** The Component Design document provides a more detailed, component-level perspective of the ELD system. It includes static elements, like classes, to describe the structure, attributes, and methods of components like data logging and driver status tracking. Dynamic elements focus on how these components collaborate during specific use cases through diagrams such as SSDs.
3. **Data Design:** The Data Design document provides a detailed overview of how data is stored, organized, and exchanged within the ELD system, ensuring it supports compliance with FMCSA regulations and meets the client’s needs. The diagrams will show the structure of our data entities like users within the MongoDB database. Additionally, the document addresses data transfer, file formats, and security.
4. **UI Design:** The UI and Dependencies document shows how users interact with the ELD system by presenting a visual walkthrough of major screens, including their functionality and user flow.

System Architecture

Introduction

In choosing our desired architecture, we placed heavy emphasis on ensuring easy, efficient, and scalable development, while also juggling the concerns and specific needs of our client. For this reason, we elected to employ a layered architecture that would maximize our ability to ship features while minimizing conflicts that arise from developing highly technical applications with a diverse set of frameworks and tools.

In our static architecture diagram, we will give a high-level view of the various components at work in our application and how they interact with each other. This will give an idea as to how certain parts such as our Android Studio frontend, Java Spring backend, and MongoDB database fit into the larger picture of what we are building. In the dynamic architecture section, we will deep dive into specific use cases to analyze where and when data is transferred and what exactly is at play. By approaching this from the perspective of a user we hope to offer a more realistic scenario to show the app in action.

Rationale

Our rationale for choosing this architectural design takes into account simplicity and security. We have compartmentalized our application and backend to allow them to have strictly defined roles. This not only makes debugging easier but also has maintainability and scalability advantages in the long run.

From a security perspective, this layered architecture creates natural security boundaries that limit attacks. Our Spring backend implements secure authentication and authorization mechanisms, while all transmitted data is encrypted. The isolation of our MongoDB database from direct client access protects against common vulnerabilities like injection attacks and unauthorized data access. We've also implemented input validation across all layers to provide defense-in-depth protection while maintaining system performance and scalability.

Static Architecture

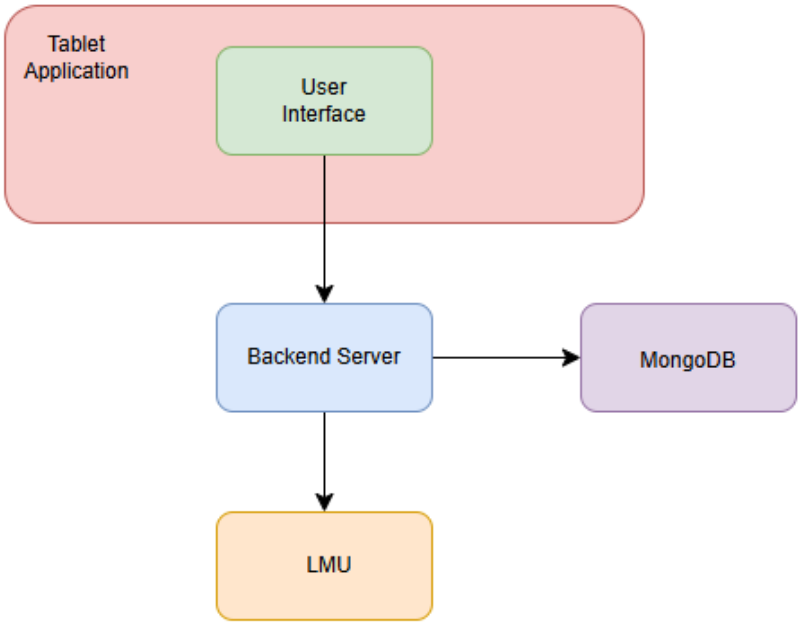


Figure 1 – Static Architecture Diagram

The Tablet Application (Android Studio) serves as the user interface, sending requests to the Backend Server (Spring Boot), which processes these requests, interacts with the MongoDB Database, and retrieves real-time data from the Load Management Unit (LMU). Spring Boot was chosen for its RESTful API support and built-in security, while MongoDB provides the scalability needed for managing large volumes of fleet and driver data. The LMU provides real-time data, such as GPS and vehicle status, for processing and display. Data transmission is encrypted, and token-based authentication makes sure that only authorized users and devices can access the system.

Dynamic Architecture

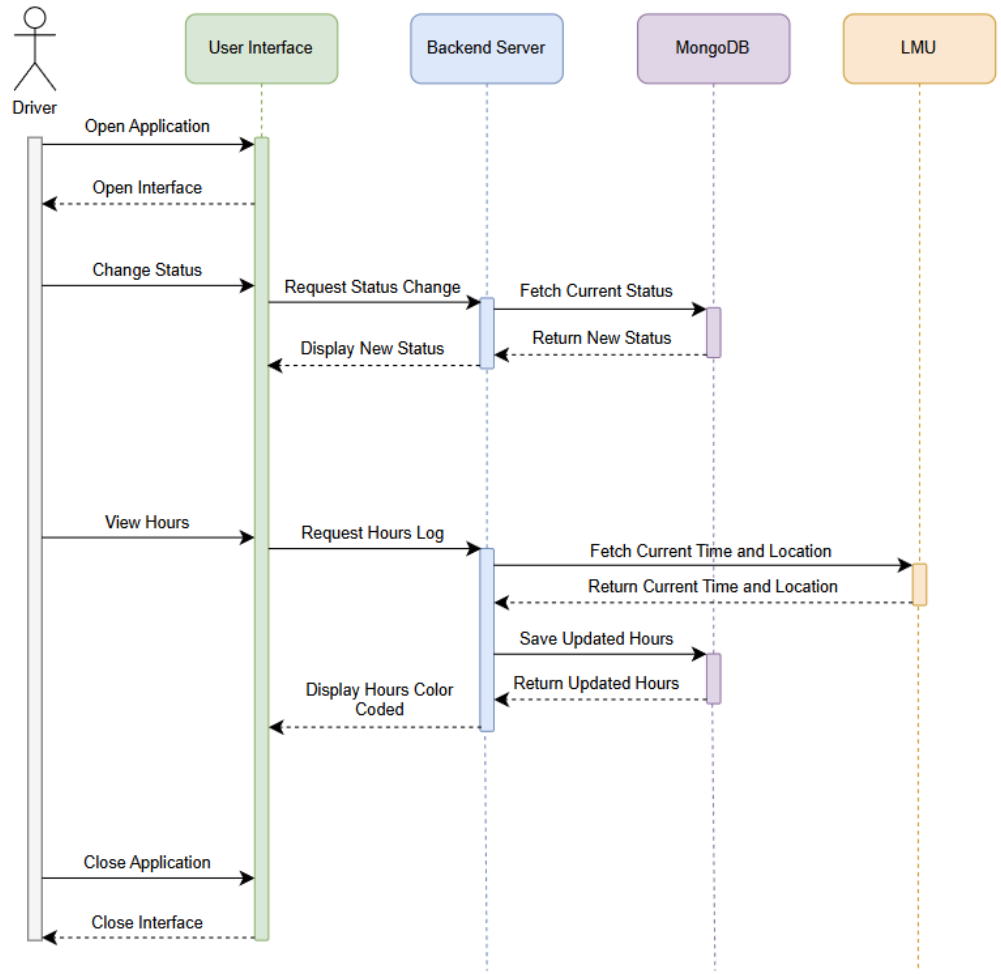


Figure 2 - Dynamic Architecture Diagram

The Dynamic Architecture Diagram shows the system's runtime behavior when a driver interacts with the app to change their status or view the remaining driving hours. When the driver changes their status, the User Interface sends a request to the Backend Server, which retrieves real-time data from the LMU, updates the MongoDB Database, and returns the new status to the UI for display. Similarly, when viewing hours, the UI requests updated hours from the server, which fetches data from both the LMU and database, processes the information, and returns it to the UI for color-coded display. This diagram highlights the interaction between components, with solid arrows for requests and dashed arrows for responses.

Component Design

Introduction

In figures 3 and 4, we are illustrating the static and dynamic components of our system architecture from a more zoomed in perspective than in figures 1 and 2. They are both color code according to the previous diagrams for consistency purposes. Green represents our user interface in the application, blue represents our backend, and purple/orange represents our data.

Static

In Figure 3, we have the static component design diagram, showing a higher fidelity view of our system architecture. Our app view is the key part to pay attention to here. In this diagram, you can see how the pages are linked to each other in succession, represented by the directions of the arrows. You navigate our app beginning from the login page and are brought to the home page, where you can navigate to the profile page, or use the navigation bar to maneuver around the app manually. Modals will pop up in certain interactions with the home page, and similarly, our profile page can direct you to an external webpage. The <<use>> arrows also show how each part of our static diagram is interacting with each other, specifically from front end to backend, and our backend is storing data as shown in the image as well.

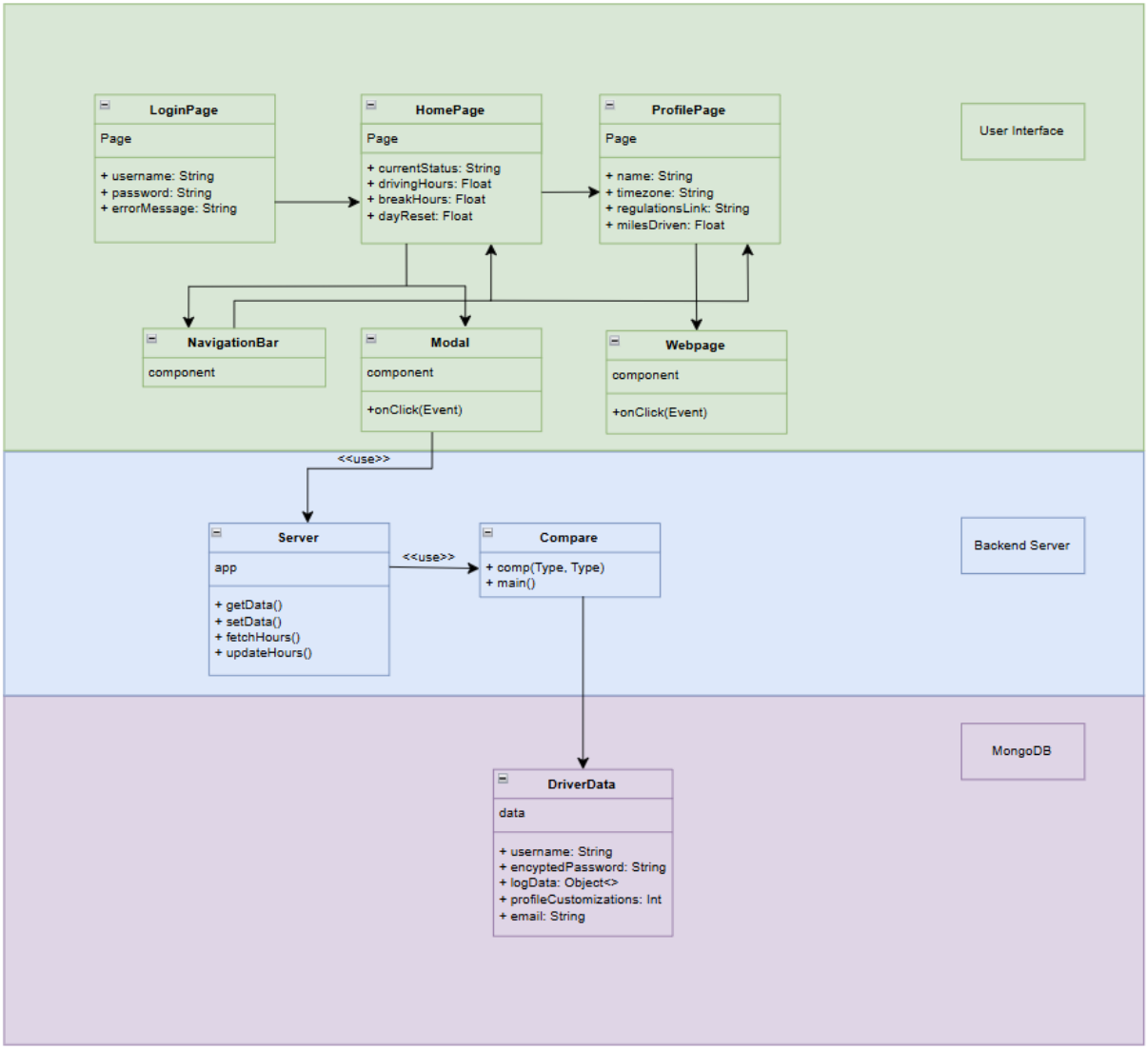


Figure 3: Static Component Design Diagram

Dynamic

Figure 4 displays an overview of the dynamic component design diagram. This shows the standard workflow our users will follow when navigating the application. Specifically, it demonstrates how a user can use the application to change their driving status, which we expect to be one of if not the most common user tasks. First the user has to log in through the login page, where he or she is then taken to the home page to view their log. From here, the user can then interact with the homepage, which changes their status in the server, and displays it back.

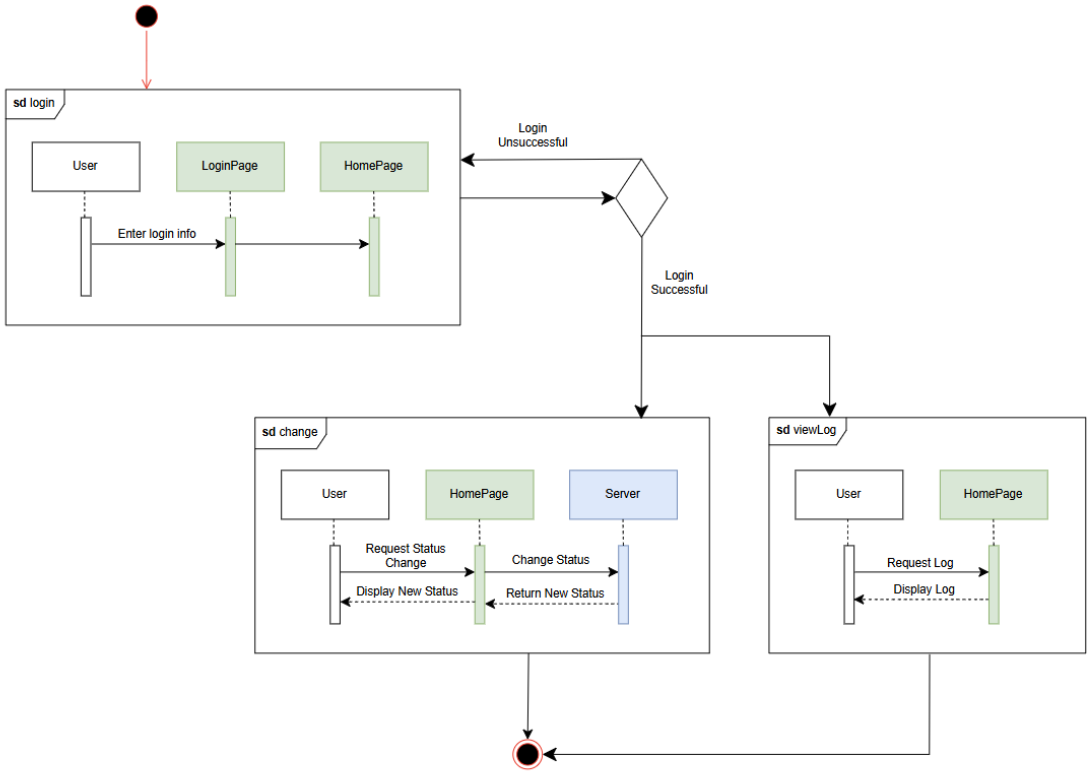


Figure 4: Dynamic Component Design Diagram

Data Design

Introduction

In Figure 5, we illustrate an in-depth perspective of the different forms of data interaction in our application. The diagram outlines the relationships between our data classes, showing how we intend for the information to flow and to be managed across each component. At the core of our data storage is MongoDB, a NoSQL database, which allows for flexible and scalable data storage. MongoDB is particularly well-suited for our needs as it allows us to manage large amounts of data efficiently while maintaining easy scalability and high performance. This structure empowers our application with the ability to grow and adapt to increasing data demands while preserving the integrity and accessibility of the data, beneficial for the long run of this product.

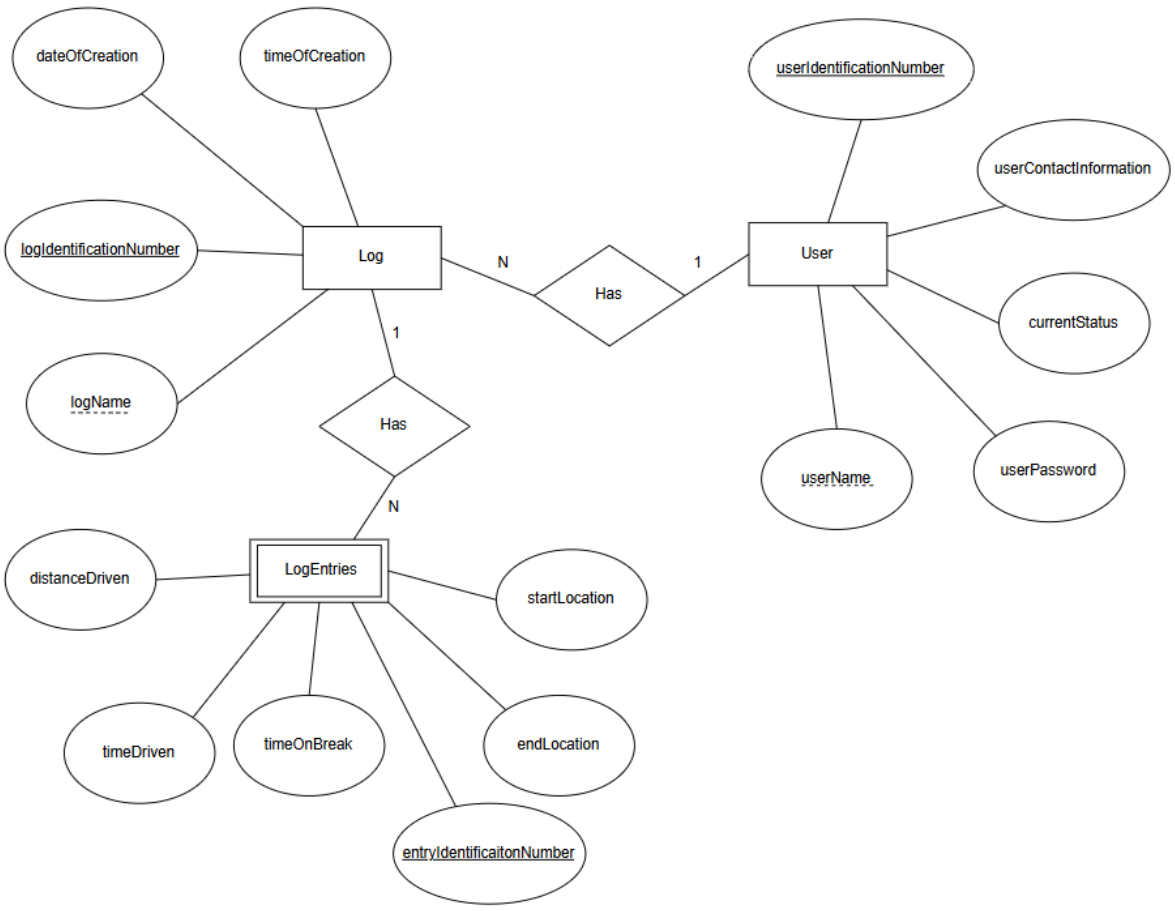


Figure 5: Entity Relationship Diagram

Data Classes

1. User: Represents the driver, which contains information such as his or her ID number, contact information (email), current driving status, username and password.
2. Log: Represents the driver's weekly log, which contains information such as the date and time of creation, ID number, and name. Each log consists of multiple log entries.
3. LogEntry: Represents the actual data for the weekly logs. This includes information such as the distance driven, time driven, time on break, starting location, and ending location.

File Use

No files used; we are simply storing data in our MongoDB database. This is shown in our appendix under "Example Database Entries".

Data Exchange

Format and Protocols

For data transfer between physical devices, we utilize many formats and protocols including JSON, MongoDB, RESTful API, and HTTPS.

- JSON will be our primary format for our data exchange between the frontend and backend due to its versatility.
- MongoDB was chosen because it provides the scalability needed for managing large volumes of fleet and driver data.
- Our backend also follows the RESTful API standards for efficient communication between the app and backend.

Security Considerations

To ensure security and confidentiality of our data, we will be taking many precautions. Firstly, we will have user authentication every time the app is opened by requiring a unique username and password. We also implemented data encryption of sensitive information such as using a hash for all the passwords in MongoDB. We chose to use HTTPS over HTTP due to its encryption of transmitted data.

UI Design

Introduction

In this section, we will present our user interface design and explain why each decision was made. This is what the drivers see and interact with while using our application. Each screen and interaction have been thoughtfully crafted with the user experience in mind, ensuring that the interface is intuitive, accessible, and efficient for users navigating it in real-world driving scenarios. This includes our Home Page, Driver Profile, Log Page, and entries in the logs.

Design

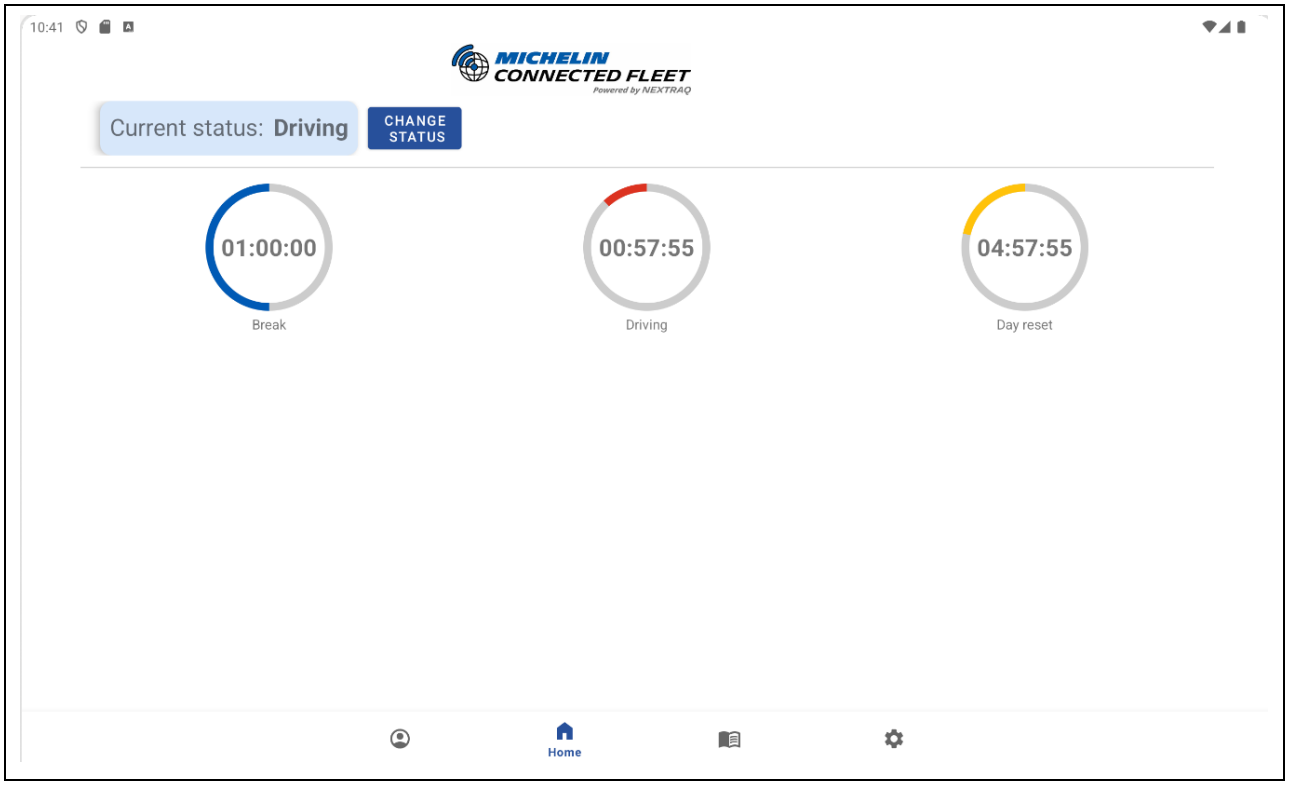


Figure 6: Home Page

When the user opens our application, they are greeted with the home page shown in figure 6. This page displays the driver's most relevant information, including their current status, hours remaining in their shift, hours of break remaining, and their day reset. We chose to display the status at the top to make it one of the first things the driver sees. This meets the heuristic of visibility of system status. We also color coded the rings that display the

remaining hours to be blue when there is a lot of time remaining, yellow when there is a moderate amount, and red when the limit is approaching. This meets the heuristic of aesthetic and minimalist design as the use of colors eliminated the need for a symbol/sentence to display these messages. We also chose to display the seconds remaining so that the driver can easily see which hours they are using. This meets the heuristic of recognition over recall.

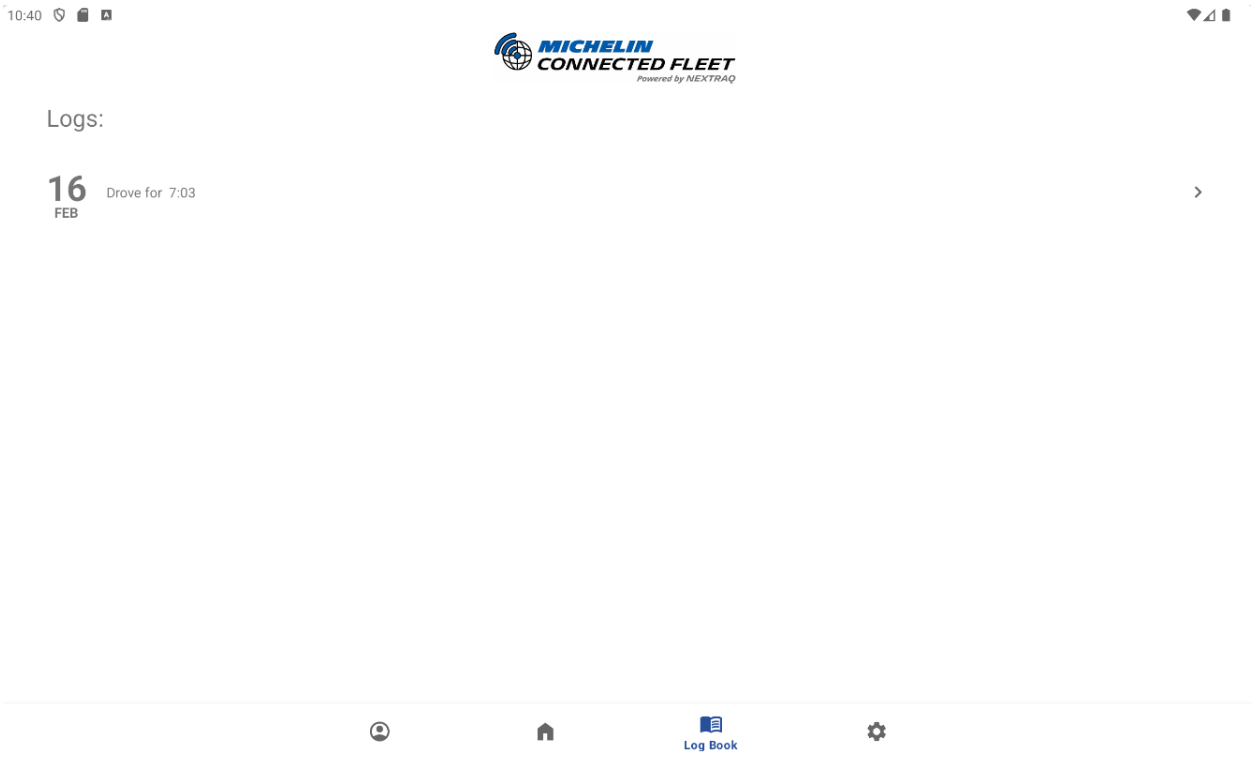


Figure 7: Log Page

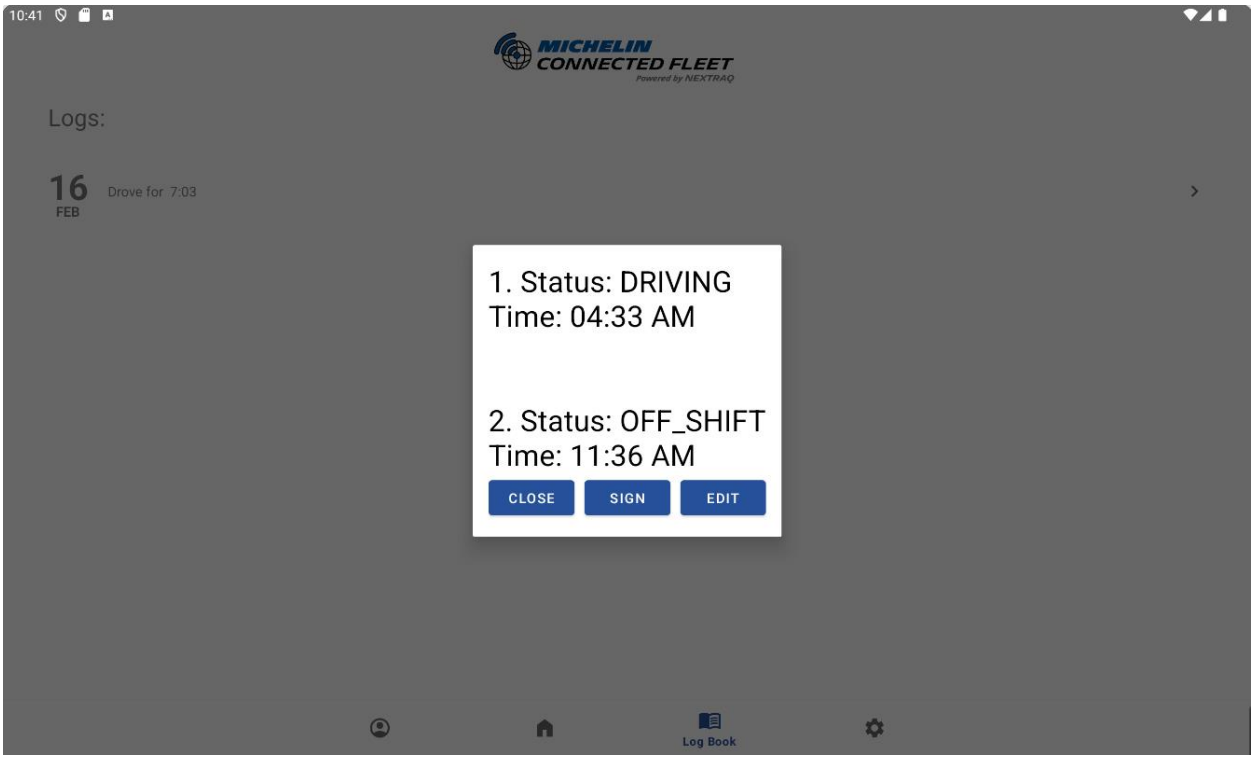


Figure 8: Log Entries

Next, we have our driver logs, which show a list of the recent logs of the drivers with the date listed on the left, as seen in figure 7. The most recent logs will be at the top, as they are likely the most relevant to the driver. This is how other ELD's format their logs, meeting the heuristic of match between the system and real world. As seen in figure 8, in each log, we show important information about the day of driving such as the time each status is started and changed. In each log, the driver also has buttons at the bottom to allow them to either sign or edit their log, meeting the heuristic of helping users recognize, diagnose, and recover from errors. We chose this design as it is very intuitive and simple, and we chose to make the buttons blue to maintain color consistency to meet the heuristic of aesthetic and minimalist design.

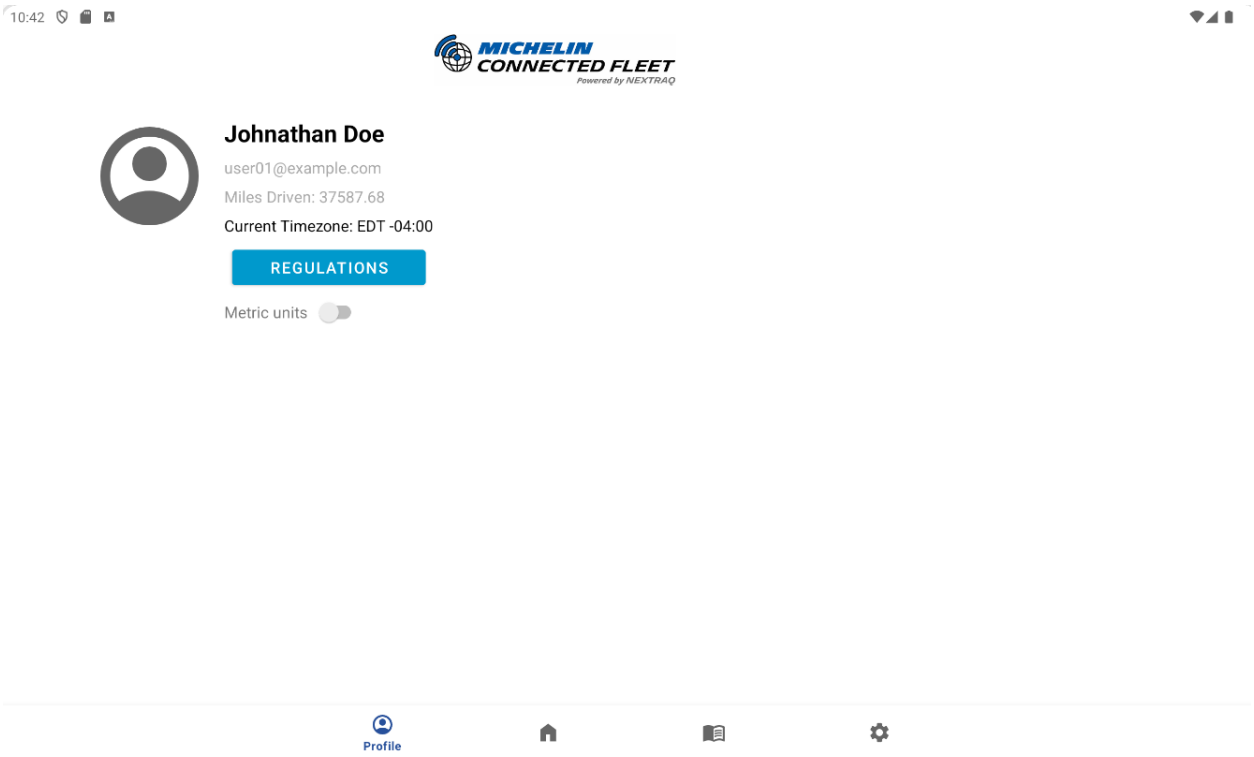


Figure 9: Driver Profile

Lastly, we have the driver profile page, shown in figure 9. This page shows the driver's name, contact information, miles driven, and current time zone. This information is listed to be readily accessible to the driver per our design decision. This meets the heuristic of visibility of match between the system and real world because it is how other ELD's implement their user profiles as well. We also included a switch to convert between imperial and metric units. This meets the heuristic of user control and freedom. We chose a switch because it is very intuitive for binary options such as units. We also implemented a button that takes the user to the FMCSA driver regulations and made it large and blue to be easily seen and color coded with the Michelin Connected Fleet company colors. This meets the heuristic of help and documentation.

Design Considerations

Throughout the process of designing the user interface, we prioritized intuitive, appropriate layouts while maintaining full functionality. By applying established UI design principles such as consistency, visibility, and ease of navigation, we attempted to create the most user-friendly experience possible to mitigate user error and confusion. One key decision was to display our pages in landscape mode as drivers are using the application on a horizontally oriented tablet, per our client's request.

Appendix

RESTful API

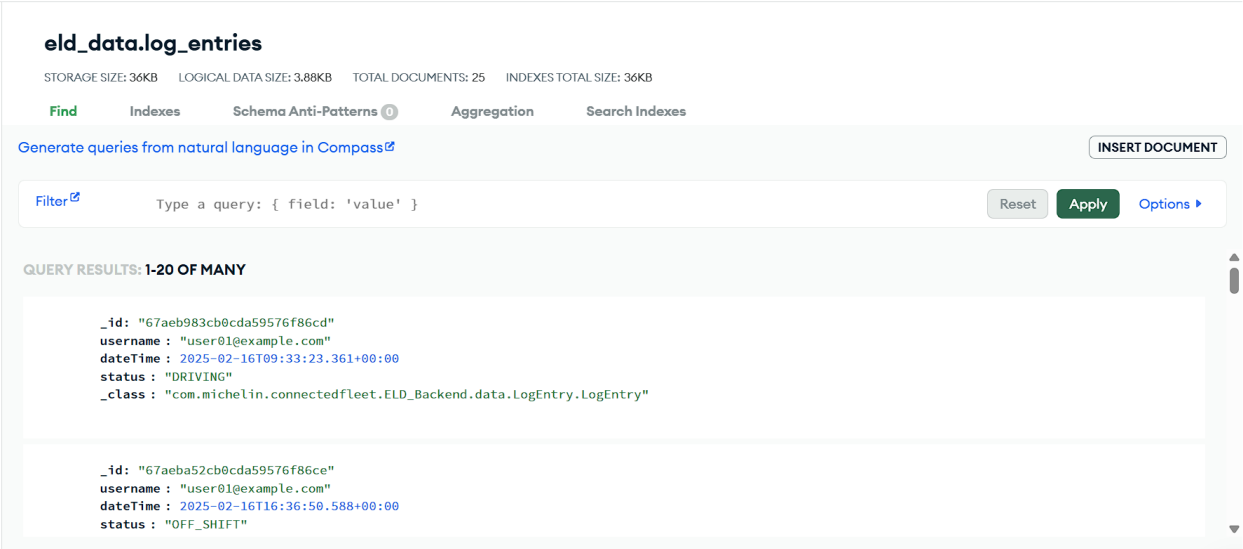
URL: https://aws.amazon.com/what-is/restful-api/

Resource Information

Response Format: Java

Requires Authentication: Yes

Example Database Entries



Team Members

Andrew Yang
Contributions: Design Documents, Weekly Meeting Agendas, Incremental Release Notes, Minor Contributions to Codebase.
Contact: ayang341@gatech.edu

Alex Messersmith
Contributions: Setup of Codebase and Database, Worked Heavily on Making Code Functional.
Contact: amessersmith@gatech.edu

Ben Davidson
Contributions: Team Leader, Hosted Group Meetings, Scheduled TA Meetings, Primary Contact for Clients, Contributed to Codebase.
Contact: bdavidson38@gatech.edu

Christian Haynes
Contributions: Jira Tracking, Worked Heavily on Codebase to Meet User Stories, Ensured Smooth Workflow of Team, and Resolved In-Group Conflicts.
Contact: chaynes38@gatech.edu