U.S. Army Vehicle Maintenance System

JIC-4312

Jonathan Collins, Caden Virant, Srithan Nalluri, Joel Cave, Mark Podrazhansky

Client: 3 ID CAVN BDE, Fort Stewart, U.S. Army

Repository: https://github.com/joelcave42/JIC-4312-Army-Maintenance

Table of Contents

Table of Figures

Terminology

Term	Definition
5988 Form	A form used by the United States Army to track faults on equipment and provide necessary maintenance.
API	Application Programming Interface: A set of functions and protocols allowing the creation of applications that access the features or data of an operating system, application, or other services.
Back-end	Hidden parts of the application/logic that cannot be accessed by the user.
bcryptjs	A JavaScript library for hashing passwords using the bcrypt algorithm.
Express.js	A web application framework for Node.js used to build APIs and handle server-side logic.
Front-end	The part of the application that users interact with directly.
Maintainers	U.S. Army soldiers whose jobs are to complete maintenance on faults found on vehicles.
MERN Stack	A web development stack using MongoDB, Express.js, React, and Node.js that allows for both the front and back end to be developed using JavaScript.
MongoDB	A NoSQL database platform that allows for data to be stored locally in flexible documents.
Node.js	A runtime environment that allows JavaScript to be run on the server.
NoSQL	A type of database that stores data in non-tabular formats like documents, allowing for flexibility and scalability for handling unstructured/semi structured data.
Operator	U.S. Army soldiers whose jobs are to complete 5988 forms and find faults in their vehicles.
React	A JavaScript library for building dynamic and interactive user interfaces.
Supervisors	U.S. Army soldiers whose jobs are to ensure the fault report and maintenance process proceeds smoothly.

Introduction

Background

The Army has been tracking vehicle maintenance the same way for the last 249 years - paper. This is obviously less than ideal as this causes a huge headache for tracking and maintaining records over the years in such a vulnerable form. All it takes is one misplaced binder or a strong gust of wind and the last 5 years of records are gone.

Our goal is to digitize the entire process through a web app, strengthening the record-keeping efficiency and allowing us to enforce new standards that promote responsibility and accountability through the chain of command. We utilize the popular MERN stack to allow for full stack development exclusively in JavaScript. In our application, different user types can report vehicle faults, conduct maintenance, assign maintainers to different maintenance projects, and view the high-level flow of the maintenance process at a company-level scale. When we are finished, we will be handing the project over to the Army who will continue development and hopefully deploy it across the world one day.

Document Summary

The System Architecture section provides a high-level overview of how the three major component layers of our system – the presentation layer, the business logic layer, and the database layer – interact with each other both statically and dynamically.

The Data Storage Design section outlines how we store user and vehicle specific data using MongoDB.

The Component Detailed Design section describes the components of our application and their static and dynamic relationships in more detail. Specific account types and roles that make up the components described in the System Architecture sections as well as specific methods from our application are showcased.

The UI Design section presents the primary user interface screens for each account type of our application.

System Architecture

Introduction

The goal of this System Architecture section is to explain exactly what this system aims to achieve. This section will explain the rationale behind the team's architectural choices and provide several diagrams to illustrate how the system architecture works both as individual pieces and together as a whole. The overall goal of the project is to make a simple and secure way for military personnel to report and update vehicle faults.

Static System Architecture

For our static system architecture, we have made design choices based on the client's initial requirements. The application needed to be versatile, accessible from both desktop computers and mobile devices, and intuitive enough for new users to quickly learn without being overwhelmed. Additionally, the database needed to be locally hosted due to budget constraints on cloud hosting and the need for enhanced security, as the system would store critical information about military vehicle readiness. Finally, the solution had to leverage modern technology with extensive documentation and community support, ensuring the client could easily maintain and enhance the product after handoff.

With these requirements in mind, we designed a responsive web application using the popular MERN stack. MERN was specifically chosen for its extensive community resources and its reputation for being easy to learn in a short amount of time. Once we hand the project off to the U.S. Army, they will need to make changes to suit their specific organization, so it's important that the learning curve for the application is relatively easy. The system is organized into three core layers, as illustrated in Figure 1.1 below:

1. **Presentation Layer**: The user interface, providing an accessible and user-friendly experience.
2. **Business Logic Layer**: Handles application logic, primarily facilitating interactions with the database.
3. **Database Layer**: Stores critical information, such as user details, vehicle faults, and maintenance history.

Given the sensitivity of the information handled by the application, such as login credentials and military vehicle readiness data, we implemented a local database for enhanced security and cost-effectiveness. To further protect user data, we used the bcryptjs library for password hashing, ensuring secure storage of credentials.

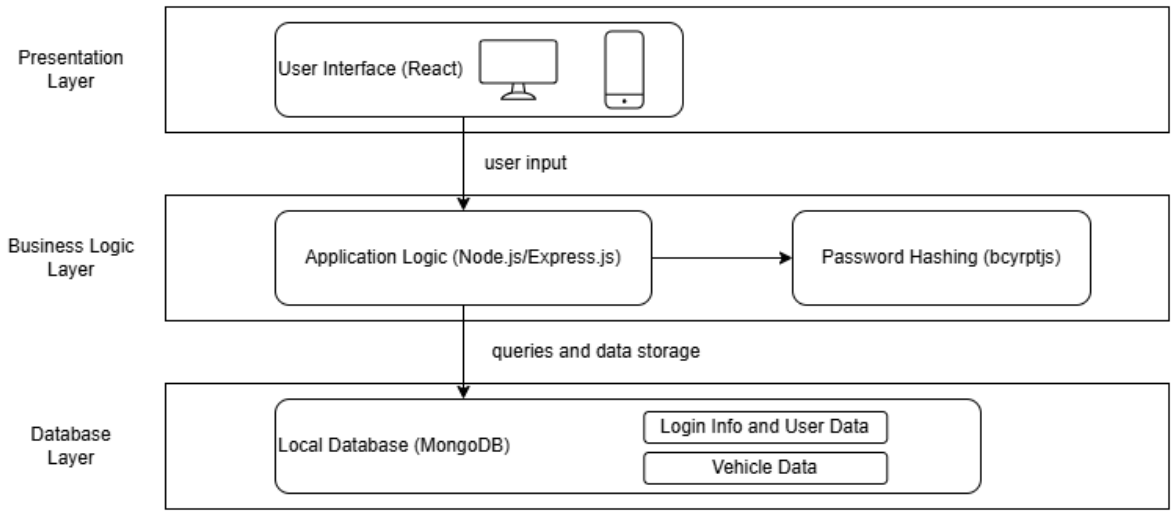

Figure 1 – Static System Architecture UML

The UML diagram above was chosen as we wanted to illustrate that the application is divided into 3 distinct layers, that communicate data in a relatively simple way. We believe that this UML diagram breaks things up into easily understandable components, without overwhelming the reader with technical jargon or large amounts of directional arrows.

Dynamic System Architecture

The maintenance intake system has several core systems tailored to the different user types. The main core functionality is centered around the ability to update information on a vehicle and retrieve that information in a secure way. The user in each case operates at the presentation layer while interfacing to the business logic layer to get any necessary information for carrying out their tasks.

Operator Example

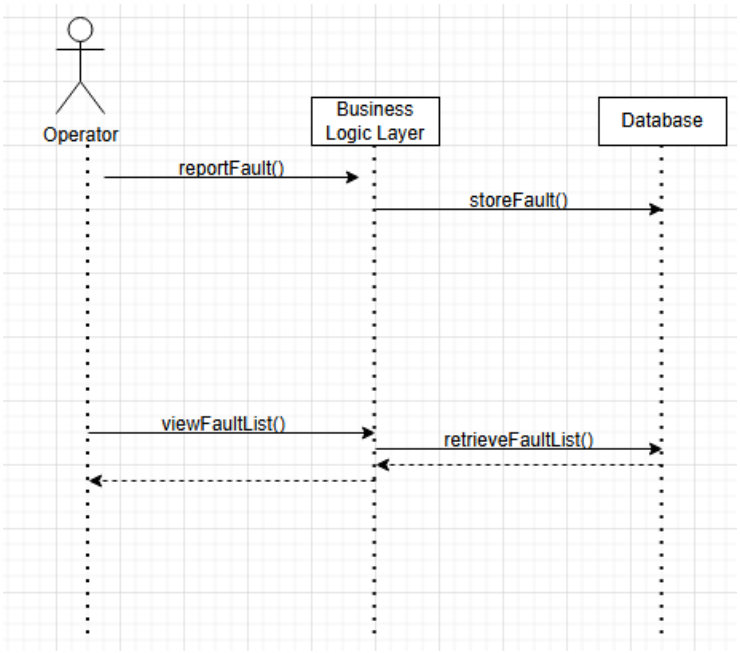

Figure 2 – Dynamic System Architecture (Operator Example)

The operator has two major tasks that they need to do which are reporting faults and view faults. The operator can report a fault with the given details necessary which will be sent to the business logic layer which will handle database entries. When viewing faults the operator can request their current faults, which the business logic layer will handle the database request to give the operator their fault list along with their statuses.

Maintainer Example

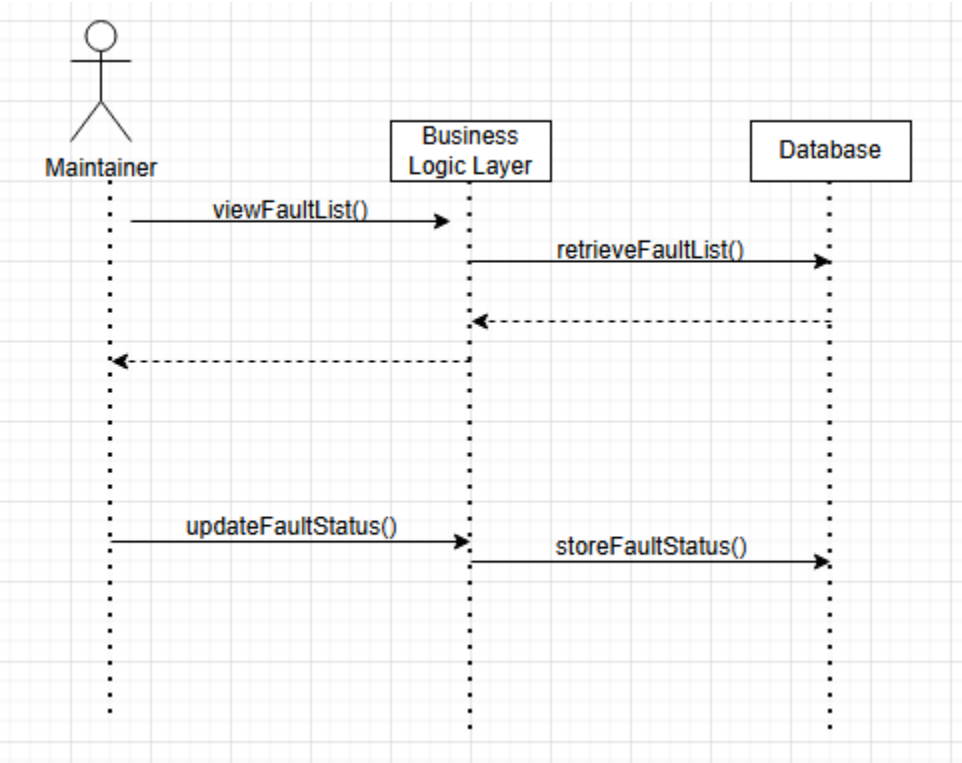

Figure 3 - Dynamic System Architecture (Maintainer Example)

The Maintainer can view faults in a similar way to the operator, however maintainers will typically have a longer fault list as they are interacting with the data from multiple operators. The maintainer can then update the status of the fault accordingly whether this is confirming the fault or requesting more information and store this status in the database by updating in the UI which will send to the business logic layer which updates the database.

Supervisor Example

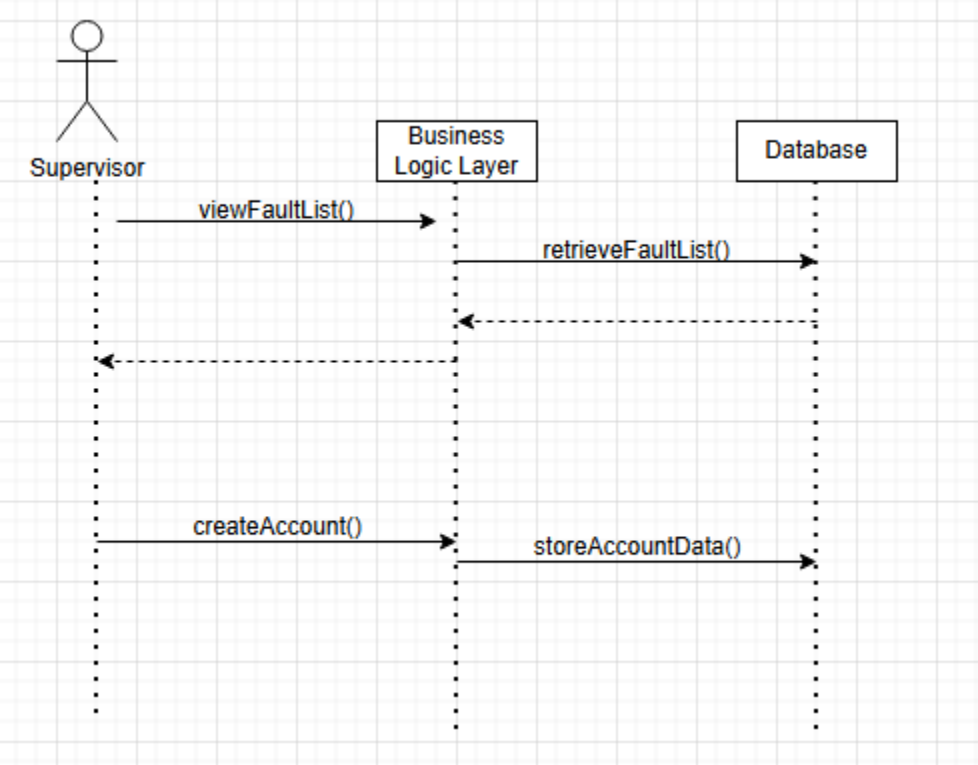

Figure 4 - Dynamic System Architecture (Supervisor Example)

The supervisor works in a similar way to the relationship between maintainer and operator, however the supervisor oversees multiple maintainers and makes sure that faults are being completed in a timely manner. The other major task for the supervisor is to create accounts for other users where the supervisor will interact with the business logic layer to handle account creation including safe password handling where this can all be stored in the database.

Through this layered approach the system can make an easy way for each role to complete their tasks while allowing efficient and secure access to updating and retrieving information.

Component Design

Introduction

This section of the Detailed Design Document provides a deeper, component-level examination of the system, expanding upon the high-level System Architecture previously outlined. It includes both a static view, illustrating the structural composition of the system, and a dynamic view, detailing the interactions between lower-level components. The intent is to offer a comprehensive understanding of how individual components function and communicate within the system as a whole.

Static Diagram

The system is structured into three primary components, each representing a distinct layer of responsibility within the overall architecture.

- **Presentation Layer**: This layer consists of the React-based frontend, which serves as the primary interface through which users interact with the system.
- **Business Logic Layer**: Responsible for handling core application logic, this backend layer manages the processing and coordination of data between the frontend and the database.
- **Data Layer**: This layer handles data persistence, storing all relevant account and fault-related information in a structured and secure manner.

This layered architecture promotes modularity, enabling the system to be scalable, secure, and easier to maintain over time.

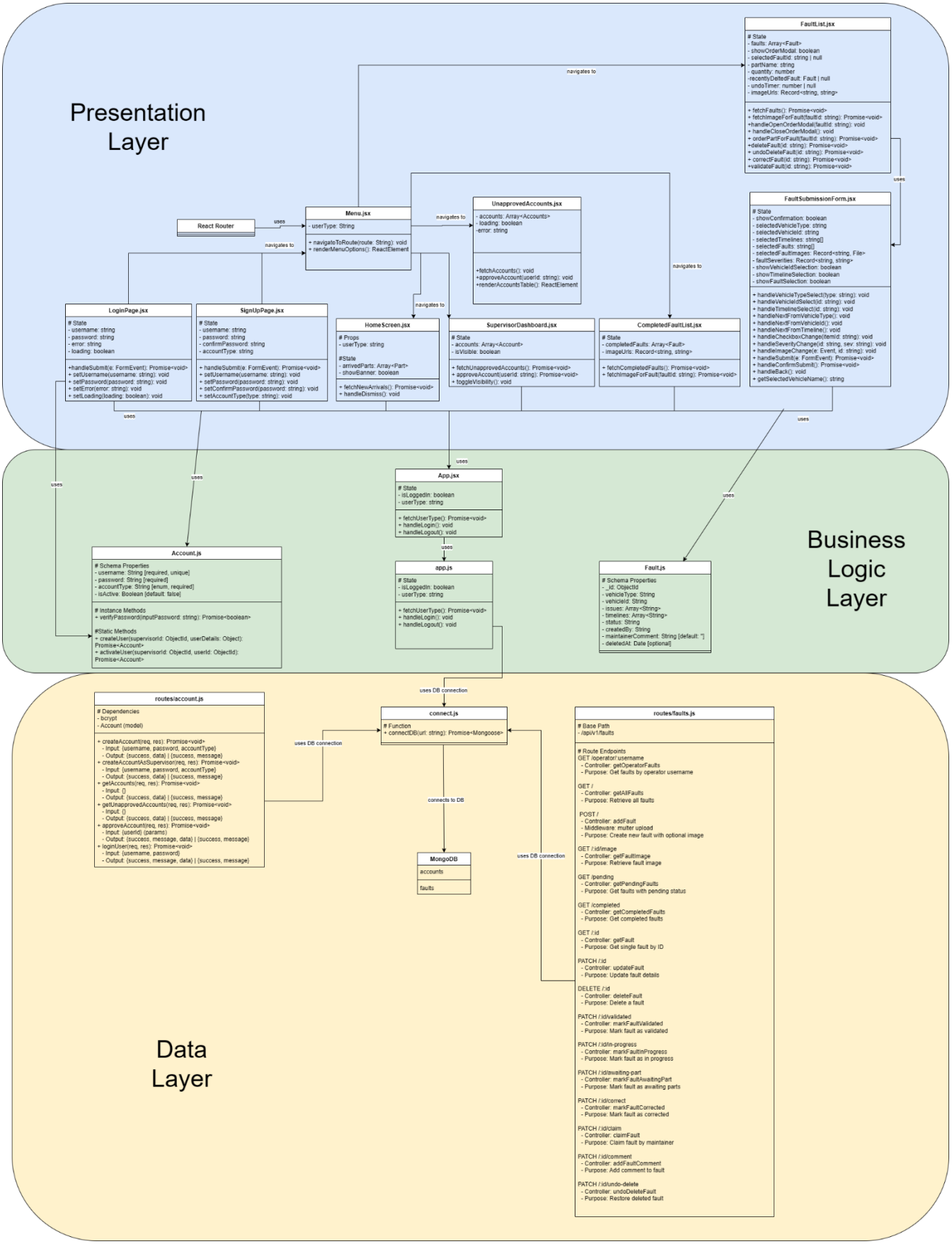
Presentation Layer
Business Logic Layer
Data Layer
FaultList.jsx
State
- faults: Array<Fault>
- showOrderModal: boolean
- selectedFaultId: string | null
- partName: string
- quantity: number
-recentlyDeltedFault: Fault | null
- undoTimer: number | null
- imageUrls: Record<string, string>
+ fetchFaults(): Promise<void>
+ fetchImageForFault(faultId: string): Promise<void>
+handleOpenOrderModal(faultId: string): void
+ handleCloseOrderModal(): void
+ orderPartForFault(faultId: string): Promise<void>
+deleteFault(id: string): Promise<void>
+ undoDeleteFault(id: string): Promise<void>
+ correctFault(id: string): Promise<void>
+validateFault(id: string): Promise<void>
navigates to
uses
Menu.jsx
- userType: String
+ navigateToRoute(route: String): void
+ renderMenuOptions(): ReactElement
React Router
uses
navigates to
navigates to
UnapprovedAccounts.jsx
- accounts: Array<Accounts>
- loading: boolean
-error: string
+fetchAccounts(): void
+approveAccount(userId: string): void
+renderAccountsTable(): ReactElement
navigates to
FaultSubmissionForm.jsx
State
- showConfirmation: boolean
- selectedVehicleType: string
- selectedVehicleId: string
- selectedTimelines: string[]
- selectedFaults: string[]
- selectedFaultImages: Record<string, File>
- faultSeverities: Record<string, string>
- showVehicleIdSelection: boolean
- showTimelineSelection: boolean
- showFaultSelection: boolean
+ handleVehicleTypeSelect(type: string): void
+ handleVehicleIdSelect(id: string): void
+ handleTimelineSelect(id: string): void
+ handleNextFromVehicleType(): void
+ handleNextFromVehicleId(): void
+ handleNextFromTimeline(): void
+ handleCheckboxChange(itemId: string): void
+ handleSeverityChange(id: string, sev: string): void
+ handleImageChange(e: Event, id: string): void
+ handleSubmit(e: FormEvent): Promise<void>
+ handleConfirmSubmit(): Promise<void>
+ handleBack(): void
+ getSelectedVehicleName(): string
uses
LoginPage.jsx
State
- username: string
- password: string
- error: string
- loading: boolean
+handleSubmit(e: FormEvent): Promise<void>
+ setUsername(username: string): void
+ setPassword(password: string): void
+ setError(error: string): void
+ setLoading(loading: boolean): void
SignUpPage.jsx
State
- username: string
- password: string
- confirmPassword: string
- accountType: string
+ handleSubmit(e: FormEvent): Promise<void>
+ setUsername(username: string): void
+ setPassword(password: string): void
+ setConfirmPassword(password: string): void
+ setAccountType(type: string): void
HomeScreen.jsx
Props
- userType: string
#State
- arrivedParts: Array<Part>
- showBanner: boolean
+ fetchNewArrivals(): Promise<void>
+ handleDismiss(): void
SupervisorDashboard.jsx
State
- accounts: Array<Account>
- isVisible: boolean
+ fetchUnapprovedAccounts(): Promise<void>
+ approveAccount(userId: string): Promise<void>
+ toggleVisibility(): void
CompletedFaultList.jsx
State
- completedFaults: Array<Fault>
- imageUrls: Record<string, string>
+ fetchCompletedFaults(): Promise<void>
+ fetchImageForFault(faultId: string): Promise<void>
uses
uses
uses
uses
uses
App.jsx
State
- isLoggedIn: boolean
- userType: string
+ fetchUserType(): Promise<void>
+ handleLogin(): void
+ handleLogout(): void
uses
app.js
State
- isLoggedIn: boolean
- userType: string
+ fetchUserType(): Promise<void>
+ handleLogin(): void
+ handleLogout(): void
Account.js
Schema Properties
- username: String [required, unique]
- password: String [required]
- accountType: String [enum, required]
- isActive: Boolean [default: false]
Instance Methods
+ verifyPassword(inputPassword: string): Promise<boolean>
#Static Methods
+ createUser(supervisorId: ObjectId, userDetails: Object): Promise<Account>
+ activateUser(supervisorId: ObjectId, userId: ObjectId): Promise<Account>
Fault.js
Schema Properties
- _id: ObjectId
- vehicleType: String
- vehicleId: String
- issues: Array<String>
- timelines: Array<String>
- status: String
- createdBy: String
- maintainerComment: String [default: '']
- deletedAt: Date [optional]
uses DB connection
connect.js
Function
+ connectDB(url: string): Promise<Mongoose>
uses DB connection
connects to DB
MongoDB
accounts
faults
uses DB connection
routes/account.js
Dependencies
- bcrypt
- Account (model)
+ createAccount(req, res): Promise<void>
- Input: {username, password, accountType}
- Output: {success, data} | {success, message}
+ createAccountAsSupervisor(req, res): Promise<void>
- Input: {username, password, accountType}
- Output: {success, data} | {success, message}
+ getAccounts(req, res): Promise<void>
- Input: {}
- Output: {success, data} | {success, message}
+ getUnapprovedAccounts(req, res): Promise<void>
- Input: {}
- Output: {success, data} | {success, message}
+ approveAccount(req, res): Promise<void>
- Input: {userId} (params)
- Output: {success, message, data} | {success, message}
+ loginUser(req, res): Promise<void>
- Input: {username, password}
- Output: {success, message, data} | {success, message}
routes/faults.js
Base Path
- /apiV1/faults
Route Endpoints
GET /operator/:username
- Controller: getOperatorFaults
- Purpose: Get faults by operator username
GET /
- Controller: getAllFaults
- Purpose: Retrieve all faults
POST /
- Controller: addFault
- Middleware: multer upload
- Purpose: Create new fault with optional image
GET /:id/image
- Controller: getFaultImage
- Purpose: Retrieve fault image
GET /pending
- Controller: getPendingFaults
- Purpose: Get faults with pending status
GET /completed
- Controller: getCompletedFaults
- Purpose: Get completed faults
GET /:id
- Controller: getFault
- Purpose: Get single fault by ID
PATCH /:id
- Controller: updateFault
- Purpose: Update fault details
DELETE /:id
- Controller: deleteFault
- Purpose: Delete a fault
PATCH /:id/validated
- Controller: markFaultValidated
- Purpose: Mark fault as validated
PATCH /:id/in-progress
- Controller: markFaultInProgress
- Purpose: Mark fault as in progress
PATCH /:id/awaiting-part
- Controller: markFaultAwaitingPart
- Purpose: Mark fault as awaiting parts
PATCH /:id/correct
- Controller: markFaultCorrected
- Purpose: Mark fault as corrected
PATCH /:id/claim
- Controller: claimFault
- Purpose: Claim fault by maintainer
PATCH /:id/comment
- Controller: addFaultComment
- Purpose: Add comment to fault
PATCH /:id/undo-delete
- Controller: undoDeleteFault
- Purpose: Restore deleted fault

Figure 5 – Static Component Diagram (Class Diagram)

Component Breakdown -

- Presentation Layer: The presentation layer is implemented using React and consists of the primary user interface components. Each screen and sub-component is modularly designed, allowing for clear separation of concerns and ease of maintenance. This structure supports a responsive and consistent user experience across the application.
- Business Logic Layer: This layer contains the backend services responsible for processing application logic. It manages operations related to account data and fault records, serving as the intermediary between the frontend and the database. It ensures data validation, request handling, and secure communication between components.
- Data Layer: The data layer utilizes MongoDB to persistently store all account and fault-related information. This schema-less NoSQL database structure allows for flexibility in storing diverse data formats, while maintaining performance and scalability.

Dynamic Diagram

The primary functionality of the system is to support the creation, retrieval, updating, and deletion (CRUD) of vehicle maintenance records. The dynamic component diagram below (Figure 6) illustrates the typical flow a user follows: logging in or creating an account, then navigating to a dashboard where they can either add new maintenance records or view and manage existing ones.

The color scheme used in this diagram is consistent with that of the static architecture diagram, maintaining visual continuity. Each color-coded component corresponds to its counterpart in the static view, with additional emphasis placed on real-world interactions and responsibilities of each module within the application.

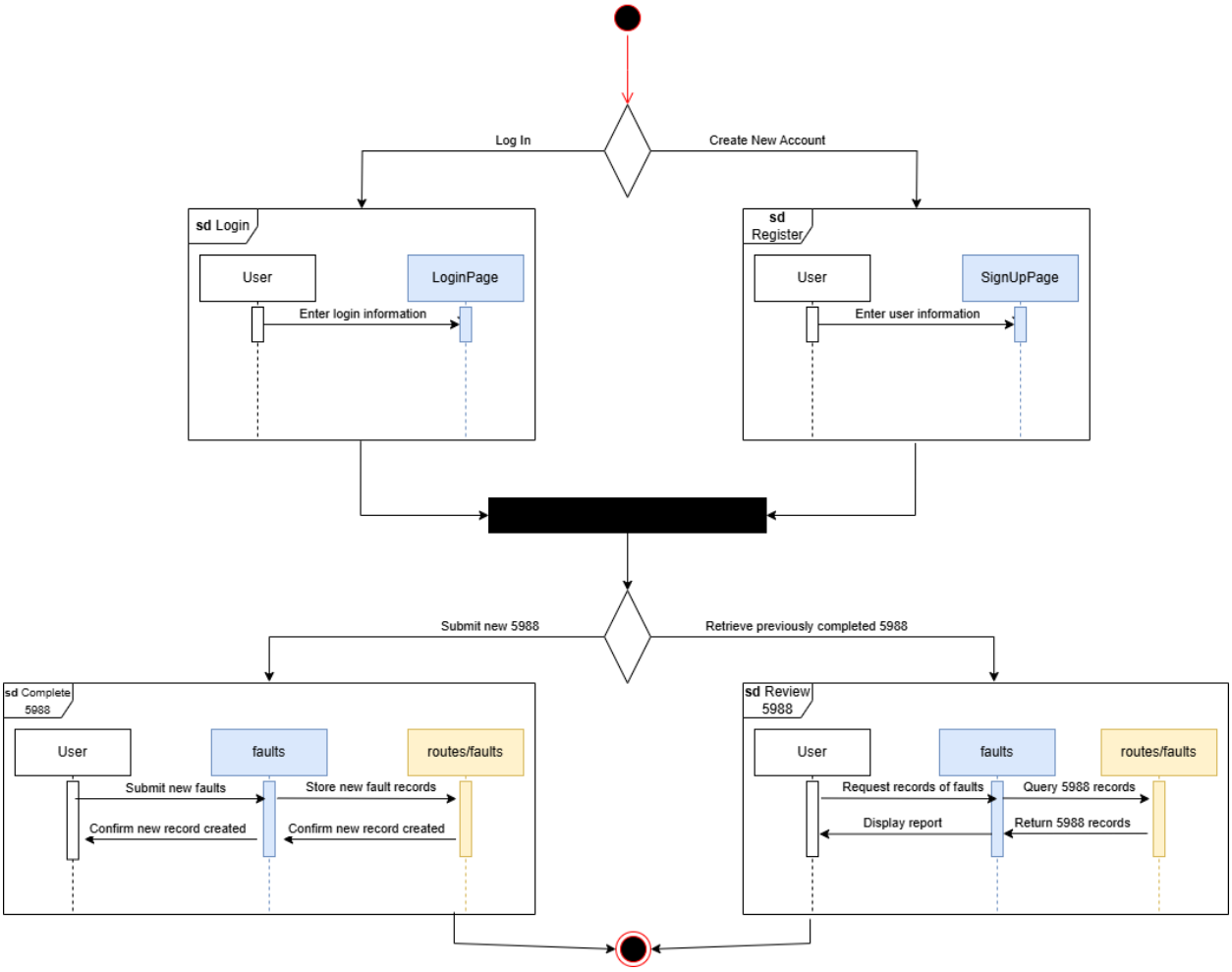

Figure 6 – Dynamic Component Diagram (Interaction Overview Diagram)

Data Design

Introduction

The example and diagram below (Figure 7, 8) represent how we use MongoDB to save and retrieve information relating to users, parts, and vehicle fault information. MongoDB supports our apps CRUD operations by storing data as BSON (Binary JSON), making a JSON document the best way to illustrate the data fields and relationships logically.

Database Use


```
{
  "accounts": [
    {
      "id": "1",
      "username": "johndoe",
      "password": "hashedpassword123",
      "accountType": "admin",
      "isActive": true
    }
  ],
  "faults": [
    {
      "id": "101",
      "vehicleId": "VH001",
      "issues": ["Engine failure"],
      "status": "pending",
      "createdBy": "1",
      "isClaimed": false,
      "claimedBy": null,
      "maintainerComment": "Needs further inspection",
      "deletedAt": null,
      "deletedBy": null,
      "createdAt": "2025-03-10T08:30:00Z"
    }
  ],
  "partorders": [
    {
      "id": "201",
      "partName": "Brake Pads",
      "quantity": 4,
      "status": "shipped",
      "orderedBy": "1",
      "fault": "101",
      "orderedAt": "2025-03-10T12:30:00Z",
      "arrivedAt": "2025-03-11T10:00:00Z"
    }
  ]
}
```


Figure 7 – Database Schema Diagram

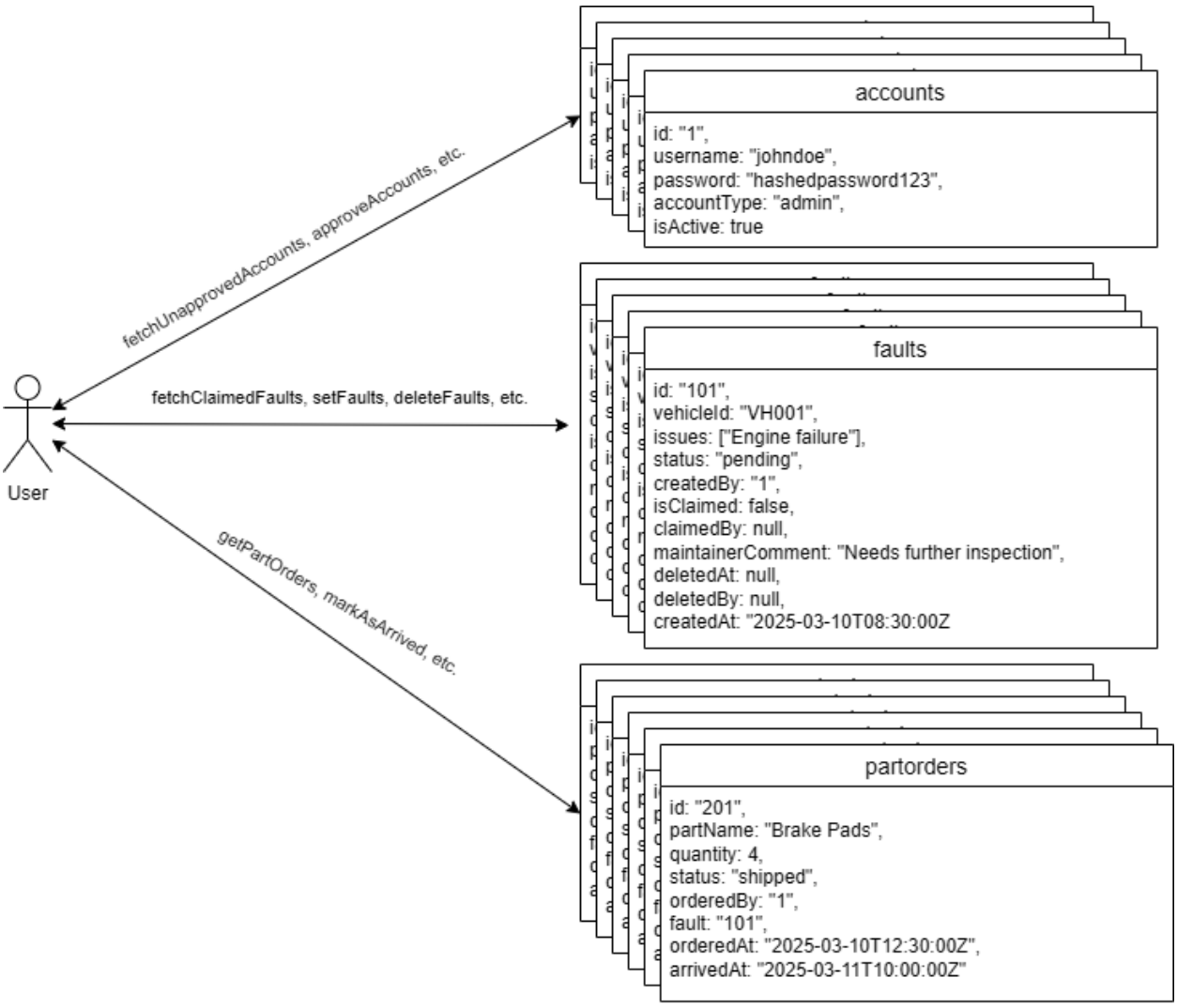

Figure 8 – Database Interactions Diagram

For our database, we have 3 tables – accounts, faults, and partorders. All the information needed for the app is contained within these 3 tables, and each of the tables has the typical set of CRUD operations defined for it. So, we can addAccounts, deleteAccounts, updateAccounts, and so on for all tables.

File Use

The only files we use in our application are images that are part of the 5988 processes. They are stored in 'client/public/images', and they serve as simple visual aids for the repair and troubleshooting process.

Data Exchange

Format and Protocols

We are not exchanging data between any devices

Security Considerations

Being an app that deals with military vehicles, security is a concern. We address these concerns in several ways-

1. We use bcrypt for password hashing, which is implemented to make brute-force attacks computationally expensive. It is widely used in security applications due to its adaptive nature and is therefore well-documented.
2. User login is required, and to make it even more secure we implemented a process where only administrators can verify accounts. If a user registers for an account, the account is inert, and the user can't login at all until the supervisor manually goes through the system and verifies the account eligibility. This ensures that every person using the app has been personally verified, and this was a specific request from the client.

UI Design

Introduction

In this section we present the User Interface (UI) of our application. Users of our application will navigate through our UI to manage the maintenance process of military equipment. We will discuss the design decisions and the major screens the different users will be interacting with. Under each textual description, we will list 3 usability heuristics and how they relate to the screenshot.

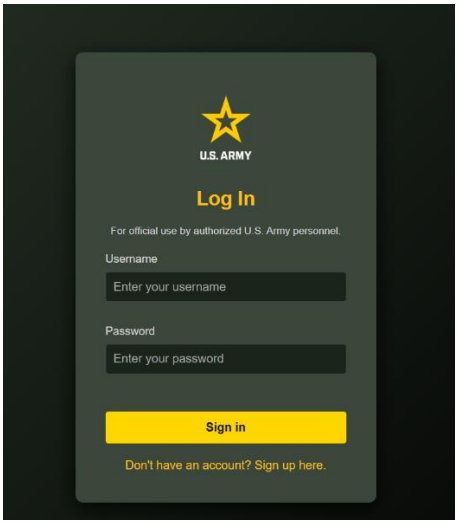

Figure 9 – Application Landing/Login Page

Figure 9 depicts the login screen, which is where users of all types will access our application.

1. **Match Between System and the Real World** – The use of the U.S. Army logo, military colors and clear mention of "authorized U.S. Army personnel" aligns with the language and branding expected by military users. It communicates purpose and audience effectively
2. **Visibility of System Status** - The fields for username and password have placeholder text (e.g., "Enter your username"), which guides users on what input is expected. The bright yellow Sign in button gives a clear indication of the primary action to take, providing visual feedback for where the user is in the process.
3. **Aesthetic and Minimalist Design** - The design is clean, focused, and free of unnecessary elements. Only the essential components - logo, instructions, login fields, and buttons - are presented. This supports user focus and reduces cognitive load.

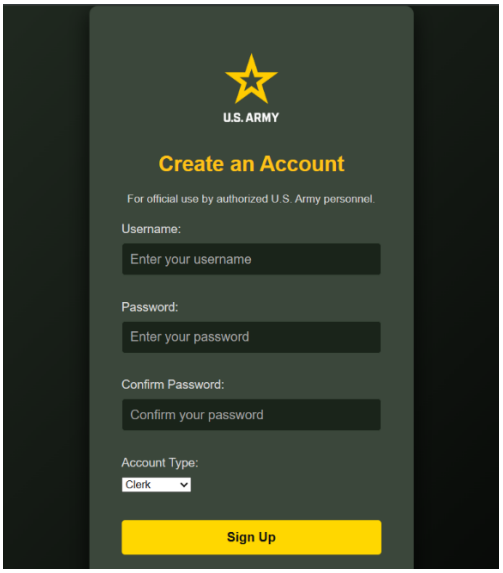

Figure 10 – Sign Up Page

Figure 10 depicts the user registration screen, where all new users will register. The user will create a username and password of their choice according to the rules of their organization, along with selecting their account type.

1. **Consistency and Standards** - The design language (colors, fonts, button styles) is consistent with the login screen, reinforcing branding and helping users feel oriented. Field labels like “Username,” “Password,” and “Confirm Password” follow standard naming conventions for account creation.
2. **User Control and Freedom** - The “Back to Login” button allows users to easily switch back if they ended up on this page by mistake -supporting easy exit and recovery.
3. **Help Users Recognize, Diagnose, and Recover from Errors** - The inclusion of “Confirm Password” helps reduce user errors related to typos during password creation. Although not shown here, this element anticipates common mistakes and encourages accuracy before submission.

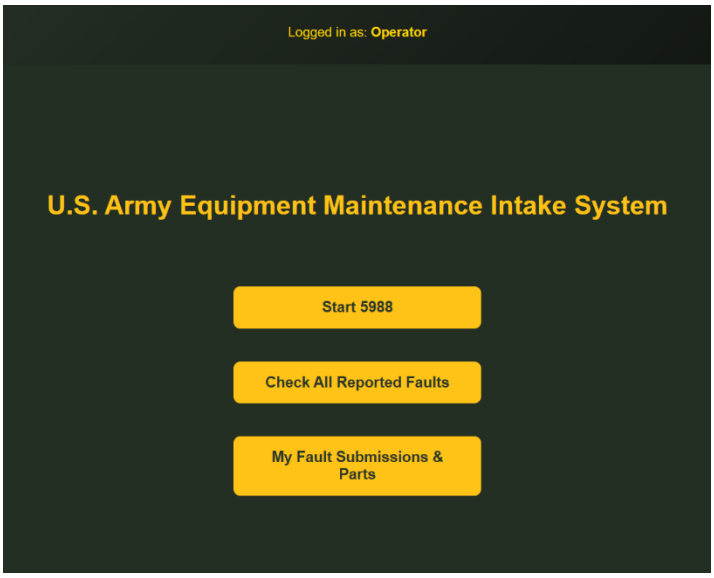

Figure 11 – Main Menu (Operator)

Figure 11 depicts the home screen, which is different for every account type. The provided screenshot is of an Operator account, which has access to all the important tools the operator might need in their job duties. We will show the major screens of all the different user account types.

1. **Recognition Rather Than Recall** - The large, clearly labeled buttons - “Start 5988,” “Check All Reported Faults,” and “My Fault Submissions & Parts” - allow users to immediately recognize available actions without needing to remember commands or navigate complex menus.
2. **Visibility of System Status** - The header displays “Logged in as: Operator”, which keeps the user informed of their current session status.
3. **Aesthetic and Minimalist Design** - The layout is clean and uncluttered, with plenty of spacing and only the most essential features shown. This reduces cognitive load and makes navigation intuitive, especially in a task-focused military context.

Figures 12, 13, 14 and 15 show the sequential steps that the user will follow once they navigate to the “Start 5988” form found on the home screen. This is the most important part of the entire application, as this is where all the data is collected from users and presented and shared with others. Every account type has access to this functionality, so the UI design was very carefully planned.

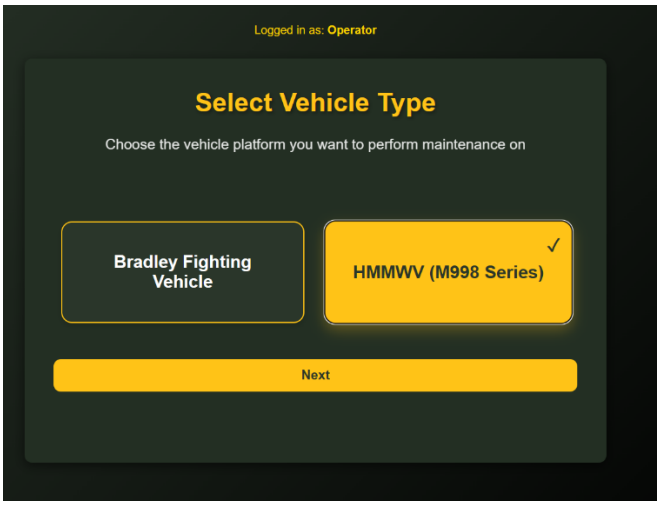

Figure 12 – 5988 Vehicle Type Selection Form

First, the user selects which vehicle type they are performing maintenance on (Figure 12).

1. **Recognition Rather Than Recall** - The vehicle types (in this example, "Bradley Fighting Vehicle" and "HMMWV (M998 Series)") are presented as large, clearly labeled buttons. Users don't need to remember names or codes - they simply select from visible options.
2. **Visibility of System Status** - The selected vehicle (HMMWV) is highlighted in yellow with a checkmark, providing immediate and visible confirmation of the user's choice.
3. **Aesthetic and Minimalist Design** - The layout is clean, well-spaced, and focused on one task: selecting a vehicle type. The prominent "Next" button ensures that progression is intuitive once a selection is made.

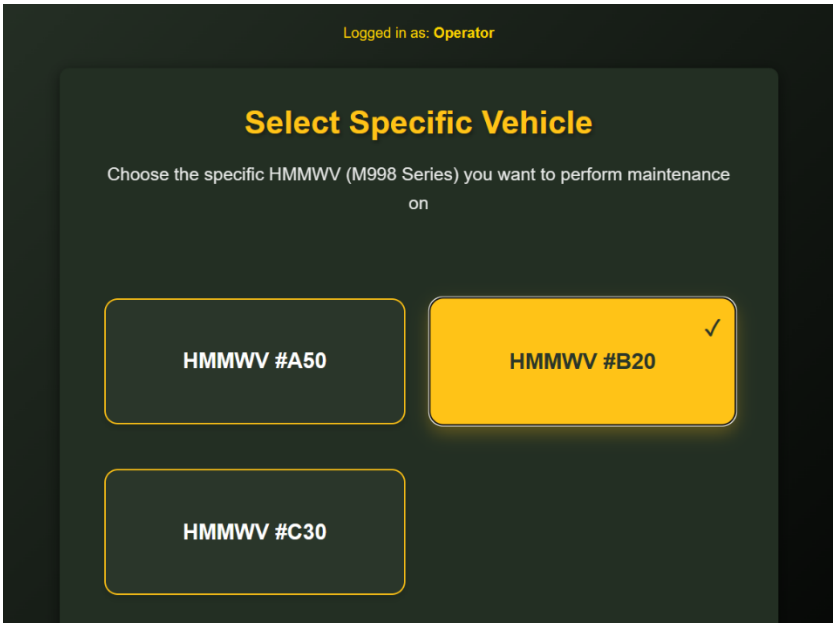

Figure 13 – 5988 Vehicle Selection Form

Then the user selects the specific vehicle (Figure 13).

1. **Recognition Rather Than Recall** - All available vehicles (e.g., HMMWV #A50, #B20, #C30) are presented visually in large buttons.
2. **Visibility of System Status** - The selected vehicle (HMMWV #B20) is visually distinguished with a yellow highlight, a checkmark and a shadow/glow effect for added contrast
3. **Aesthetic and Minimalist Design** - The interface is clean and focused solely on the task: selecting a vehicle.

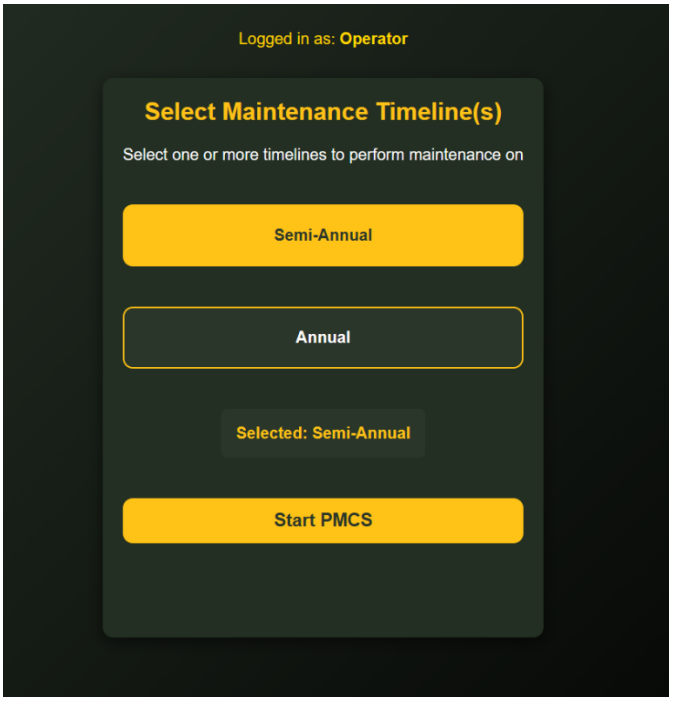

Figure 14 – 5988 Maintenance Timeline Selection Form

Then the user selects the desired maintenance interval timeline (Figure 14).

1. **Visibility of System Status** - When “Semi-Annual” is selected, it's highlighted in yellow, and a label appears below that says “Selected: Semi-Annual”.
2. **Recognition Rather Than Recall** - Maintenance options are shown directly as buttons labeled "Semi-Annual" and "Annual", so there is no need to remember what timelines exist - the system presents choices clearly.
3. **Aesthetic and Minimalist Design** - The design is clean and focused on a single decision: choosing a timeline. The Start PMCS button is clearly separated from the selection process and only becomes the next logical step once a selection is made.

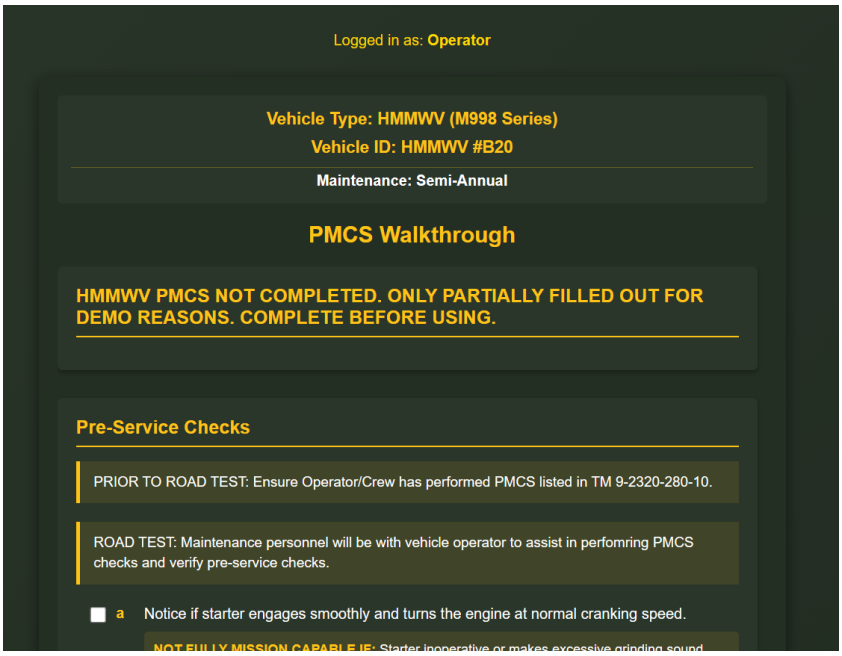

Figure 15 – 5988 Form

Then the user arrives at the digitized 5988 form (Figure 15), where they begin the data entry portion of the maintenance process.

1. **Visibility of System Status** – The header confirms the role that the logged-in user is acting in. It also shows the selected vehicle type, vehicle ID and the maintenance timeline being performed
2. **Match Between System and the Real World** - Terminology like “Pre-Service Checks,” “Road Test,” and references to TM 9-2320-280-10 mirrors language used in real military maintenance manuals. PMCS (Preventive Maintenance Checks and Services) is a term users in this context will already understand.
3. **Consistency and Standards** - Styling and terminology match the rest of the system (yellow highlight for emphasis, same font hierarchy, button styling).

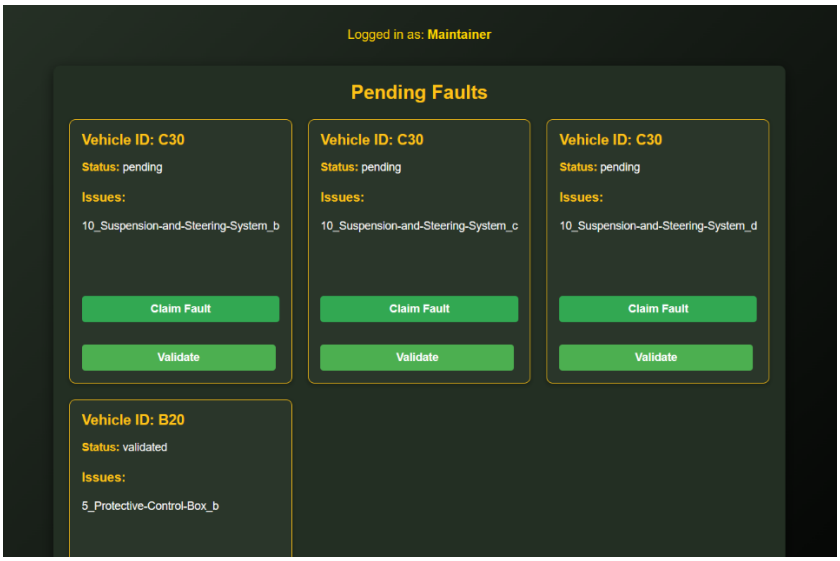

Figure 16 – Pending Fault Screen (Maintainer)

Figure 16 depicts the "Pending Faults" screen, which can be accessed from the Maintainer account type homepage. This is another crucial feature of the application, as this is where maintainers validate the faults that need attention and then "claim" faults to be responsible for fixing them. This is a very important step in the process as this ensures that all faults are given proper attention.

1. **Visibility of System Status** - The user's role is clearly shown at the top: "Logged in as: Maintainer". Each fault card displays the Vehicle ID, Status, and Issue description, making the current state of each report clear.
2. **User Control and Freedom** - Each card offers direct action: Claim Fault and Validate. Users can interact with faults individually, suggesting a level of granular control without being forced into a specific workflow.
3. **Consistency and Standards** - All cards follow the same structure (Vehicle ID → Status → Issue → Action Buttons)

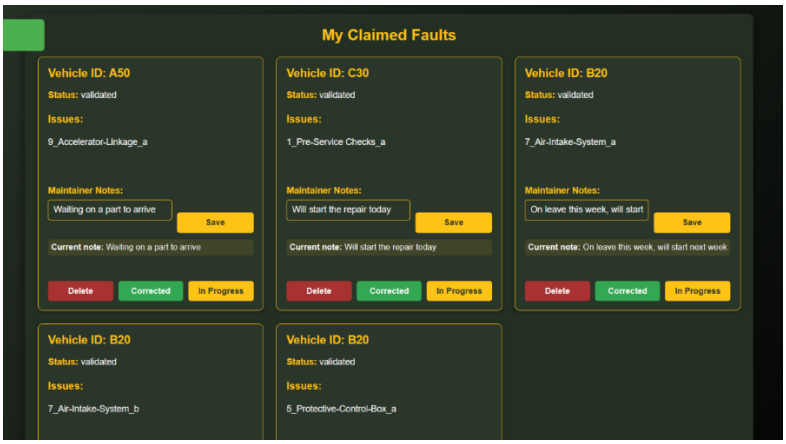

Figure 17 – My Claimed Faults Screen (Maintainer)

Figure 17 depicts the "My Claimed Faults" screen of the application, which can be accessed from the Maintainer home screen. From this page the user can update the status of a fault using the different indicators at the bottom of the fault card, or they can leave more detailed messages that get sent to the user who submitted the fault report.

1. **Visibility of System Status** - Each card shows the vehicle ID, status, issue, and the current maintainer note. Status options like "Corrected", "In Progress", and "Delete" are actionable and visually distinguished.
2. **User Control and Freedom** - Users can update notes, change the status of the fault, or delete it entirely.
3. **Match Between System and the Real World** - Labels like "Maintainer Notes," "Status," and "Issue," mirror real-world terminology used by mechanics and NCOs.

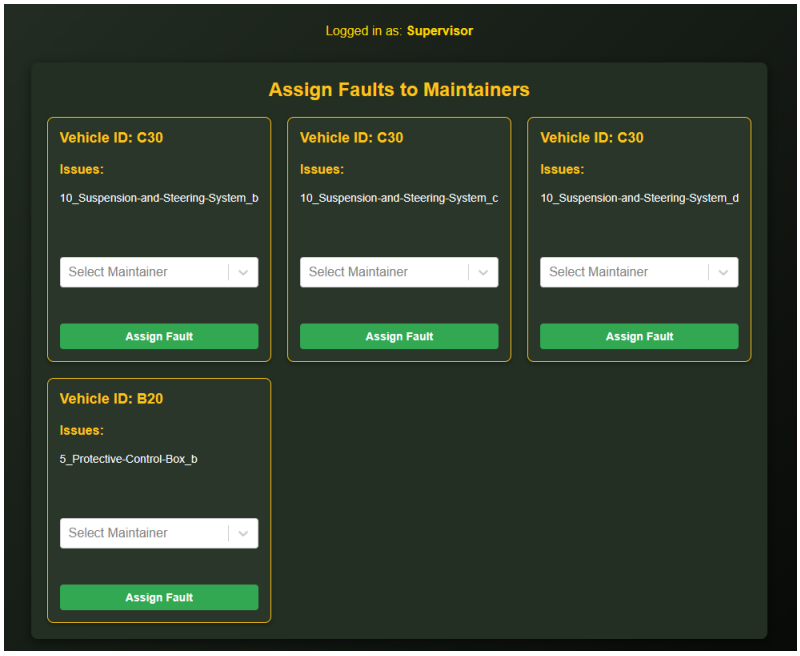

Figure 18 – Assign Faults to Maintainers Screen (Supervisor)

Figure 18 depicts the Supervisor tool that allows them to delegate the tasks of repairing faults. This is an especially important tool as it is up to the Supervisor to ensure that faults are repaired in a timely manner, and this also allows Supervisors the ability to specially assign people who might be best suited for a particular job.

1. **Recognition Rather Than Recall** - Vehicle IDs, fault descriptions, and dropdowns are shown explicitly -supervisors don't have to remember which faults exist or which maintainers are available. Also, the label "Select Maintainer" cues the user on exactly what to do.
2. **User Control and Freedom** – Each fault is independently assigned, meaning the user can control assignment on a per-task basis. No bulk actions are forced - the supervisor can move at their own pace and assign faults selectively
3. **Consistency and Standards** - Each fault card is uniformly styled: Vehicle ID → Issues → Dropdown → Assign Button.

Design Considerations

We designed our app so that most of the necessary navigation takes place immediately after logging in, starting with the main landing page. For most user roles, this landing page consists of three to four primary buttons, as shown in Figure 11. The most common actions

- reporting faults, viewing reported faults, and checking faults assigned to the user - are centrally located as buttons, accessible to all roles. Since these actions are frequently used, their central placement makes them easy to find and use.

For maintainers, a fourth button is added to allow them to claim faults and view their own claimed faults. This button remains aligned with their role-specific tasks and is placed alongside the other actions for consistency. This layout ensures that navigation is simple and that users can access key screens with minimal clicks.

Outside of the central area, navigation remains intuitive. A back button is always located in the top-left corner, making it easy to return to the previous screen and correct any mistakes. The logout button is consistently placed in the top-right corner. For supervisors, an additional button appears in the top-right corner that leads to their dashboard. This separate button allows the main layout to remain consistent with other roles while still providing access to supervisor-specific functions. The supervisor dashboard also follows the same design principle, displaying only the most essential actions relevant to that role.

Several screens display lists of faults, and we ensure all of them follow a consistent format. These lists are sorted from oldest to newest, as older faults are typically more urgent. As illustrated in Figure 16, the layout of each list item is consistent in color, size, and shape, so users can easily recognize that these elements represent database entries.

Finally, we prioritize user feedback at every step. Whenever an action is completed, a confirmation message appears to inform the user of its success. Additionally, we've implemented safeguards against mistakes - for example, when a fault is deleted, an "Undo" button temporarily appears in case the deletion was accidental.

Appendix

We didn't use any APIs for this project; however, we currently have a placeholder for the inventory system that is used which will later be used to interface with the military SAP API. However, that will be for when the project is passed to the military software engineers as it is beyond the scope of the current project.

Each member acted as full stack developers and helped contribute to all areas of the project as needed. The contribution section includes some of the larger sections each member contributed to the project.

Member Name	**Contribution**	**Contact Email**
Joel Cave	UI lead, part orders and notifications, fault status tracking	jcave31@gatech.edu
Jonathan Collins	PMCS process, Large UI portions, Role-based access	joncollins@gatech.edu
Srithan Nalluri	Login Page, General Home screen, Fault organization	snalluri9@gatech.edu
Mark Podrazhansky	Authentication, Fault Claiming, Account Approval	mpodrazhansky6@gatech.edu
Caden Virant	Database Design, Operator Screens, Maintainer Functions	cvirant6@gatech.edu