The Resource Alignment Group

MAFES Equipment Management System

System Design Document

November 17, 2025

Project Team

Bradan Craig

Drew Marecek

McKade Wing

Theodore Morin

MAFES Customer Representative

Tyler Messerschmidt

The Resource Alignment Group

MAFES Equipment Management System

System Design Document

Table of Contents

1. Introduction

This is a capstone project for the Maine Agricultural and Forest Experimentation Station (MAFES) at the University of Maine. This project will fulfill the capstone requirement for a Computer Science Bachelor's degree for Bradan Craig, Drew Marecek, McKade Wing, and Theodore Morin. This project involves developing a web application for an equipment management and tracking system to replace MAFES's current Excel process. The new system will allow all MAFES staff and students to easily view, reserve, and manage agricultural and forestry equipment across their six research farms.

1.1. Purpose of This Document

The purpose of this document is to outline the design and architecture of the MAFES Equipment Management System. It defines how the system requirements will be implemented in the final system through models representing the system's architecture, database, functions, and files. These models will define how data flows throughout the system and how core components interact with each other. The intended audience for this document includes MAFES customer representatives, MAFES research faculty, the capstone development team, and University of Maine faculty who will review the document's completeness.

1.2. References

Atlassian. (2024). Software Design Document [Tips & Best Practices]. Atlassian. https://www.atlassian.com/work-management/knowledge-sharing/documentation/software-design-document

How to create a functional decomposition diagram? Concepts, tutorials, templates - ProcessOn. (n.d.). https://www.processon.io/blog/functional-decomposition-diagram-tutorials

SRS, The Resource Alignment Group. (2025). *The System Requirements Specification Document.*

2. System Architecture

This section outlines the system's architectural design, detailing the major components, their interactions, and the platforms used to create the product.

2.1. Architectural Design

The following diagrams illustrate the logical and technological structure of the MAFES Equipment Management System. It details the interactions between the technologies that will be present in the final product, as well as the core components prevalent within them.

Technology Architecture Diagram

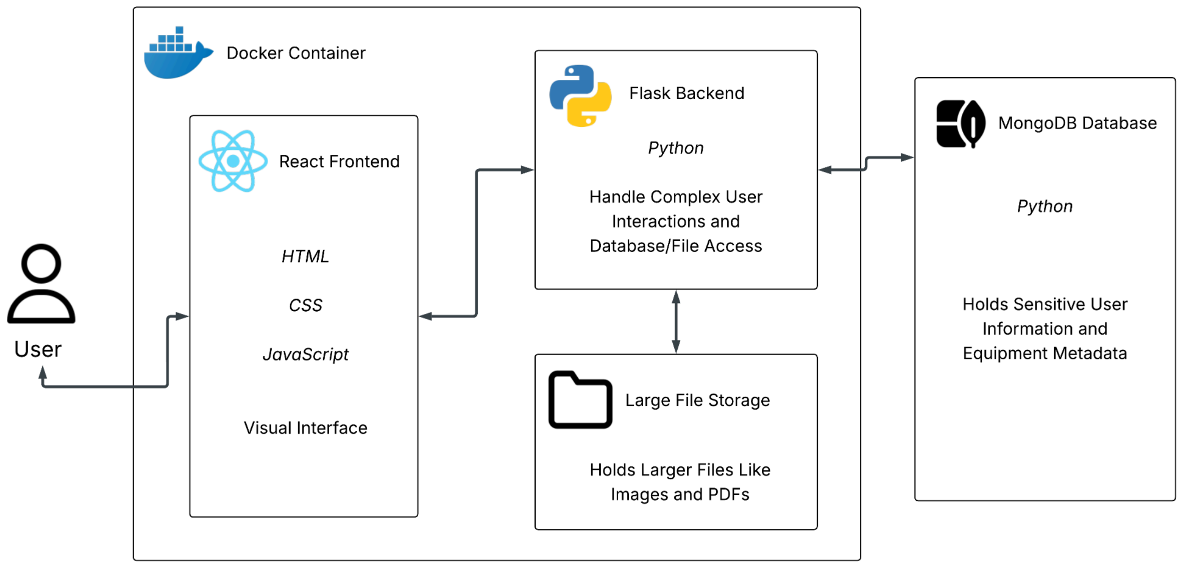
Docker Container
Flask Backend
Python
Handle Complex User Interactions and Database/File Access
React Frontend
HTML
CSS
JavaScript
Visual Interface
User
Large File Storage
Holds Larger Files Like Images and PDFs
MongoDB Database
Python
Holds Sensitive User Information and Equipment Metadata

Figure 1: Details the interactions between the top-level technologies that will be used in the MAFES Equipment Management System. This includes a description of their core purpose(s) and what languages will be used during their development.

Logical Architecture Diagram

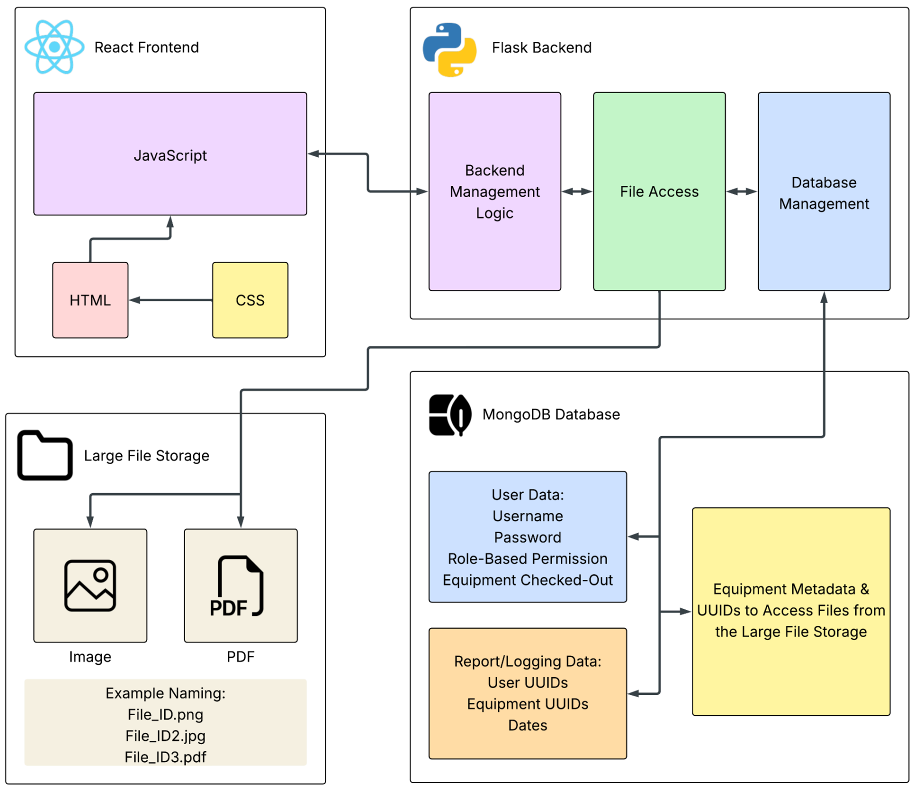

Figure 2: These are the core components within each technology that will be used in the project. The arrows help map the data flow between these components, simulating how they will fetch/receive data.

The system will follow an architecture composed of a React frontend, a Flask backend, and a MongoDB database, with an additional volume for ample file storage that will be hosted using Oracle's free block storage. The React frontend, built with HTML, CSS, and JavaScript, provides the interface that users can interact with. The frontend will communicate with the Flask backend to handle features such as user authentication, equipment checkouts, and admin-only actions. This will make React ideal for creating a stylistic user interface and making calls to the Flask backend.

The Flask backend will be written in Python and will manage the complex system logic that React can't manage. It will connect the React user interface, the MongoDB database, and the Oracle block storage volume, streamlining the system's data management. When a user interacts with the React interface, such as logging in, requesting to check out equipment, or submitting a damage report, Flask handles these actions by validating credentials, checking role-based permissions, and performing the necessary database operations. For example, when a user wants to check out a piece of equipment, Flask will verify its availability by referencing the MongoDB Equipment table to determine if it's checked out already, create a new entry in the Reports table to log the exchange, and update the User table to reflect who has current access to the equipment. Flask will also manage the retrieval of metadata and files when a user views equipment details. It will first retrieve the equipment metadata (e.g., name, model, year) from the MongoDB Equipment table and check for any references to associated images or PDFs. If UUIDs are found, Flask will then access the Oracle block storage volume to retrieve those files. Afterwards, it will serve all of the equipment data and files to the React frontend for the user to view. Using Flask in this manner ensures efficient data handling between core system components and provides a separation of concerns, thereby improving data management.

Most of the system will be containerized using Docker to simplify the deployment of the final product. Within the container, there will be the React frontend, Flask backend, and an Oracle block storage volume, while the MongoDB will remain independent. By isolating the main components within Docker and keeping the database external, the system will be easily scalable and allow for a simplified configuration between components.

2.2. Decomposition Description

The section details the decomposition of the MAFES Equipment Management System into its major subsystems, components, and modules. It expands upon the architecture introduced in Section 2.1 by illustrating how the system's functionality will be divided into logical units, each responsible for specific roles within the system.

The system is organized around six main classes: User, Equipment, Reports, Main, Database Manager, and Notifications. Each one encapsulates a distinct set of behaviors that allow the system to function smoothly.

User Class
The User class manages all user related operations and authentication functions. Its primary method, authenticateUser(), verifies login credentials and establishes active sessions for valid users. This class interacts closely with the DatabaseManager to retrieve and update user information, ensuring data consistency across the system. Although it does not directly connect to the Reports class, all significant user activities are logged using Reports.logAction() to maintain traceability. This class primarily acts as an interface between frontend authentication requests and secure backend operations.

Equipment Class
The Equipment class handles the core functionality of the system related to tracking and maintaining equipment records. It provides methods for retrieving, adding, updating and deleting equipment entries as well as managing file uploads. The Equipment class

interacts heavily with the DatabaseManager for creating, reading, updating, and deleting equipment through various operations. It invokes Reports.logAction() to record significant state changes, such as when equipment is added, deleted, or marked as damaged.

Reports Class
The Reports class is responsible for tracking user and admin actions, generating reports, and exporting data summaries. It serves as a key component in analyzing the equipment being used. Through its primary function, logAction(), this class records all relevant activities to ensure full traceability. By maintaining detailed action logs, the Reports class supports non-repudiation and admin monitoring access across the system.

Main Class
The Main class acts as the central control point of the system. It initializes the system environment, manages references to the other classes and coordinates the interactions between them. This class doesn't perform direct operations on data. Instead, it calls methods from User, Equipment, Database Manager, Reports and Notifications to ensure synchronization throughout the system. This centralized structure ensures consistent communication across all system modules.

Database Manager Class
The Database Manager class encapsulates all database operations, providing a single interface for accessing and modifying MongoDB collections. Other classes such as User, Equipment and Reports rely on Database Manager to perform queries, insertions, updates, and deletions, maintaining data integrity and reducing redundancy. This class ensures consistency across the system and reduces redundant database logic.

Notifications Class
The Notification class provides a messaging mechanism that alerts users and admins when certain system events occur. Other classes such as User and Equipment interact with Notifications indirectly by triggering events that require alerts. The class encapsulates functions such as getUserNotifications() and deleteNotification(), allowing users to access messages and remove resolved notifications while maintaining system traceability.

MAFES Class Diagram

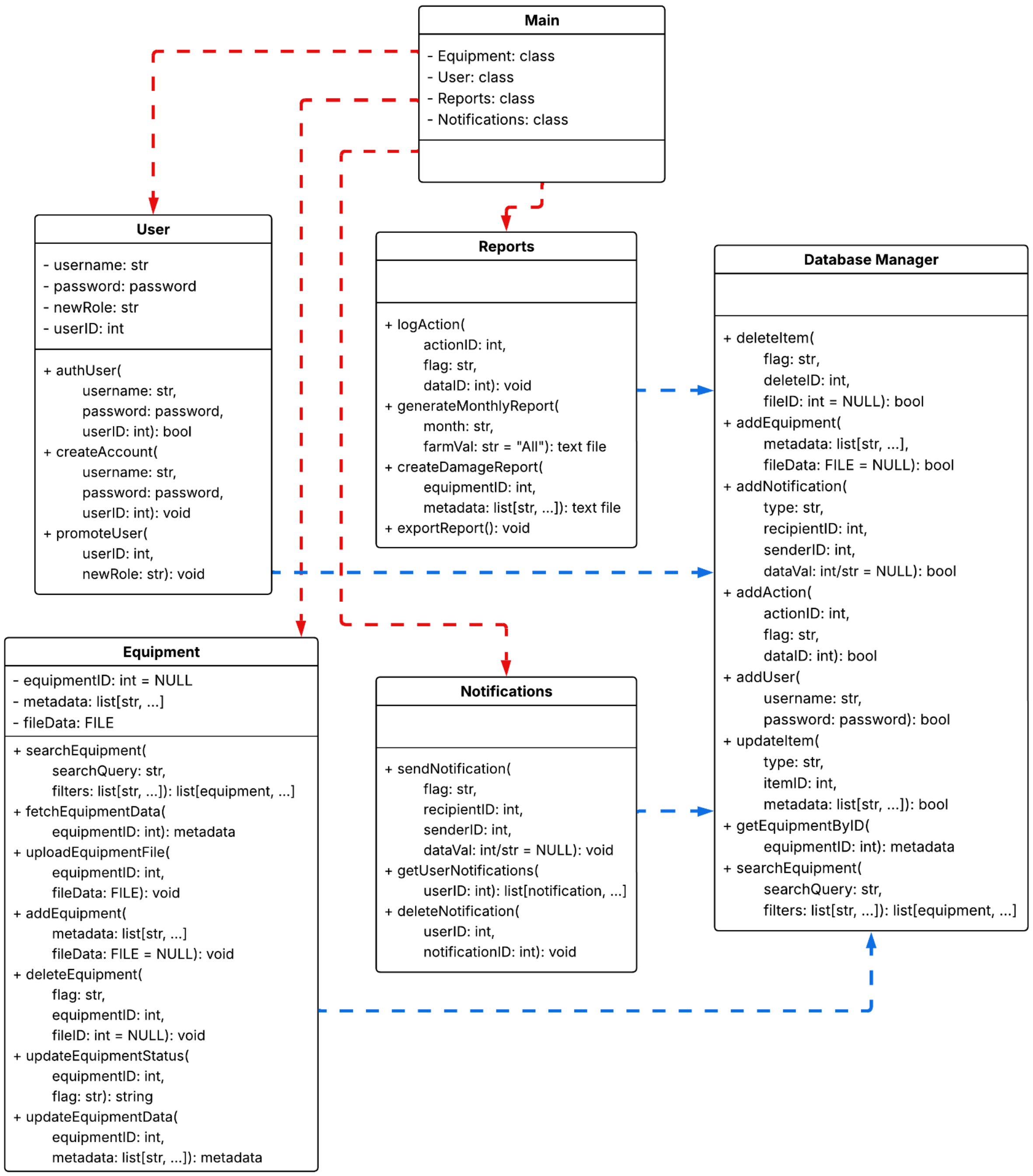

Figure 3: The class diagram illustrates the primary components and their relationships within the MAFES Equipment Management System. It shows how the User, Equipment, Reports and Notifications classes interact through the Database Manager to manage authentication, inventory, reporting, and notifications. The Main class has references to all other class objects and coordinates their interactions.

3. Persistent Data Design

This section outlines the procedures for storing, managing, and accessing data within the MAFES Equipment Management System. This will include detailed descriptions of the database tables, fields, and file formats used to support accurate records of equipment and users.

1.1. Database Descriptions

Our database will be primarily built using MongoDB, with a separate directory for storing larger files. Due to financial restraints, we will be using the free version of MongoDB, which limits us to 512MB of storage. This will not be enough to store all of the information required by the MAFES Equipment Inventory System. To combat this, we will be using 200GB of free block storage within the “Always Free” tier in Oracle’s VM. Within the VM, a Docker container will be present with an attached volume, where large files, such as images, reports, and other non-primitive data, will be stored. MongoDB will primarily be responsible for storing primitive data and containing the UUIDs of larger files for easy lookups. This should be sufficient for the permanent hosting of the system, considering the current equipment the farm has and projected future additions.

Database Schema Design

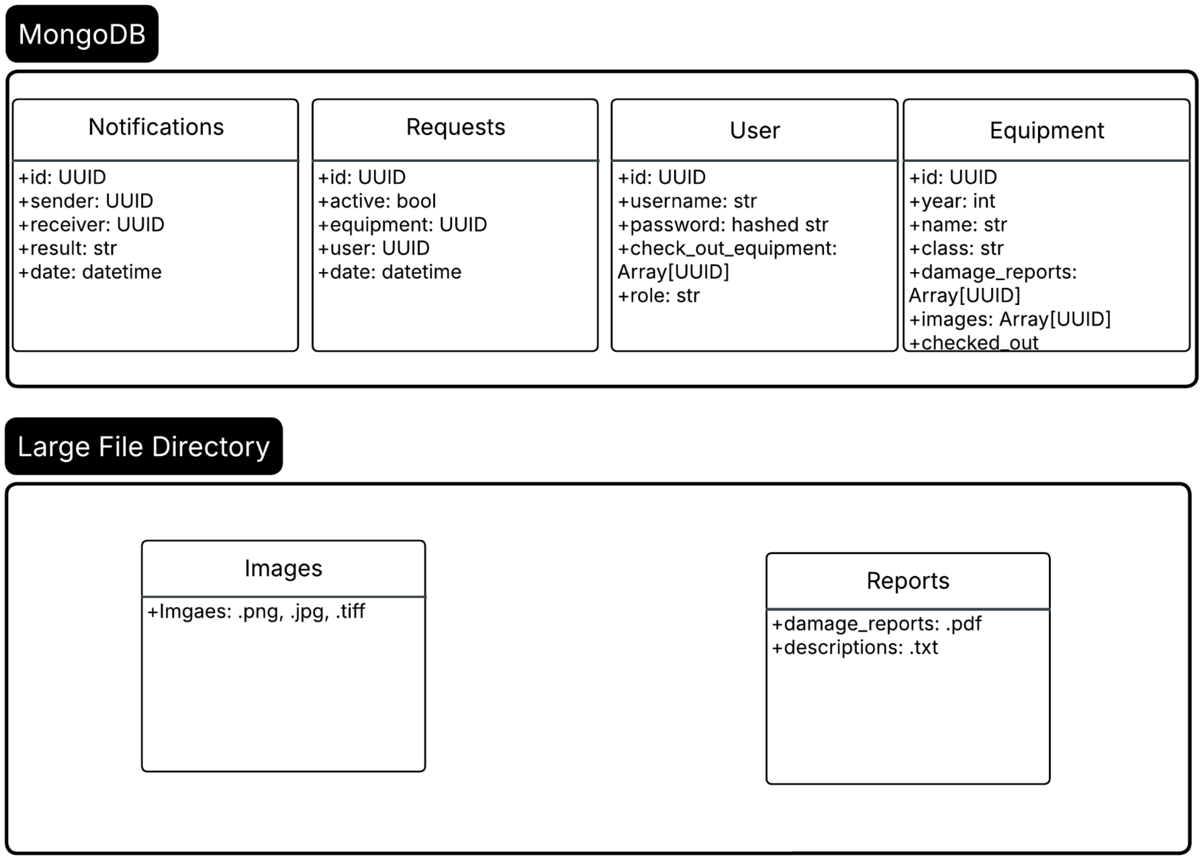

Figure 4: Outlines the schema of the MongoDB database and what it will store. Data will be accessed via Flask endpoints, either using pymongo or traversing the direct path using the os library in Python. Randomly generated UUIDs will be used to maintain data integrity and individuality.

1.1.1. MongoDB

Information on each item stored in the database is provided below. This includes the type of each item to be stored, its approximate size, and a general description of its purpose.

Requests:
This table will include all data regarding requests for checking out items. The system will utilize this information to store all item movements for monthly reports and logging purposes. This will be stored within the MongoDB and will be a whole separate collection.

Number	Name	Type	Size	Description
1	id	UUID	16B	This will be the primary key for each request

2	active	bool	1b	This shows whether the request is still pending or not
3	equipment	UUID	32B	This will point to the primary key of the equipment being requested
4	user	UUID	32B	This will point to the primary key of the user who is requesting the equipment
5	date	datetime	8B	This will log the date and time of the request

Users:
This MongoDB collection will contain all the users' specific information, including their profiles. This will include their authorization information, as well as details about their particular equipment and the permissions required for the system to access it.

Number	Name	Type	Size	Description
1	id	UUID	16B	This will be the primary key for each user
2	username	str	8-32B	This will be the username for the user
3	password	str (hashed)	64B	This will be the password of the user, hashed so it is not easily readable
4	checked_out equipment	Array[UUID]	0-128B	This stores all of the equipment that the user currently has checked out
5	role	char	1B	This will keep track of the role that gives the user certain permissions

Equipment:
Each piece of equipment will be stored in this collection hosted with MongoDB. This will be accessed whenever a user is on the home page or needs more information about a specific piece of equipment. Any new equipment will also be added here and will require all of the fields below to be completed.

Number	Name	Type	Size	Description
1	id	UUID	16B	This is the primary key of the specific equipment
2	year	short	2B	This is the year of the equipment creation

3	name	str	16-64B	This is what the user wants the equipment to be called when it is added to the database
4	class	str	16-64B	This is the type of equipment that it belongs to
5	damage reports	Array[UUID]	0-1MB	This will store all the UUIDs of the damage reports in the “Large Files Database.”
6	images	Array[UUID]	0-1MB	This will hold all of the UUIDs of the images held in the “Large Files Database”
7	checked_out	bool	1b	This will be “True” when the equipment is checked out and “False” when it is not

Notifications:
Any notifications sent throughout the system will be stored in this MongoDB collection, making them easily accessible. The body of all notifications should be template-like, so that the specific primitive information is the only thing stored.

Number	**Name**	**Type**	**Size**	**Description**
1	id	UUID	16B	This is the ID for each of the notifications
2	sender	UUID	32B	This corresponds to the UUID of the person sending the notification
3	receiver	UUID	32B	This corresponds to the UUID of the person receiving the notification
4	result	str	0-16MB	This will either be the result of the notification of what class it responds to if it is not a request notification
5	date	datetime	8B	This will be when the notification is sent

1.1.2. Large Files Directory

This will be a Docker volume that is attached via the Docker Image. This will simply be a directory with two sub-directories, both named “Reports” and “Images.” Each will hold the larger non-primitive data required for the system and will be accessed based on the UUIDs stored in MongoDB.

Reports:
Within this directory, you will find all the large files required for the system. These will include damage reports and any extended descriptions. Each file will have a unique UUID as its name, allowing it to be easily accessed by the system.

Number	**Name**	**Type**	**Size**	**Description**
1	damage report	.pdf	0-20MB	This will be a damage report that corresponds with a specific piece of equipment
2	description	.txt	0-20MB	This description will correspond to a specific piece of equipment and will be created when the equipment is added to the database.

Images:
The system requires images for easy identification of the equipment. They will also be stored in a directory, with each image assigned a unique UUID as its name. These images will be pulled whenever it is needed by the s

Number	**Name**	**Type**	**Size**	**Description**
1	images	.png, .jpg	0-20MB	This will be an image that corresponds with a specific piece of equipment

3.1. File Descriptions

This section describes the key source code files and their interactions, which comprise the MAFES Equipment Management System. It provides a map for developers to understand the project's structure, critical data fields, and the organization of functions.

File Interaction Diagram

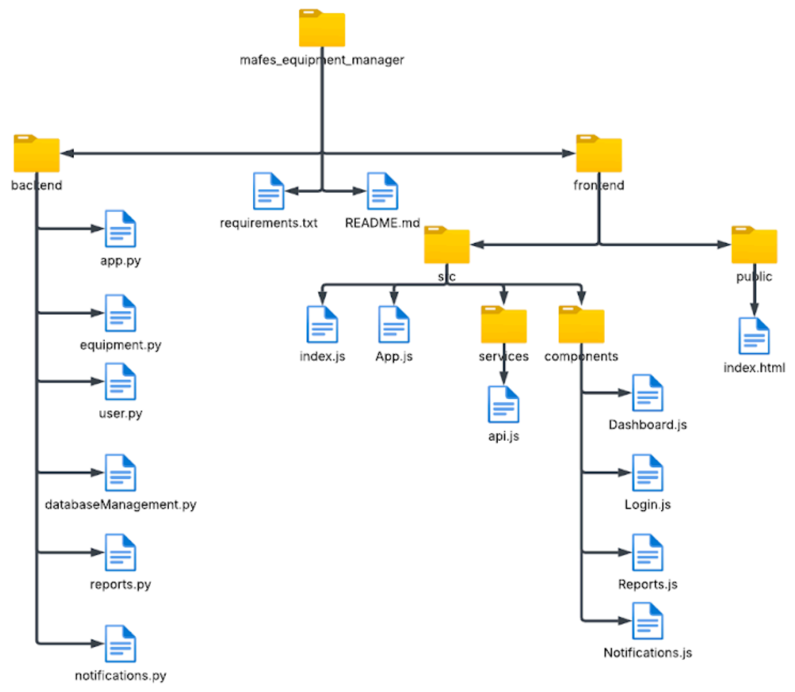

Figure 5: This diagram illustrates the folder and file structure of the MAFES Equipment Management System, showing the organization of both the frontend (client) and backend (src) source code.

Frontend File Descriptions (React)

File: api.js

Description: Centralizes all API communication with the Flask backend.

Key Variables/Fields:

- API_URL (string): Stores the base URL of the backend

File: Dashboard.js

Description: Loads the main equipment dashboard. Manages the state for the equipment list and user filters.

Key Variables/Fields:

- equipmentList (state, array, 1MB): Stores the whole list of equipment objects fetched from the API
- filteredList (state, array, 1MB): Stores the equipment list after applying user filters

- searchQuery (state, string, 128B): Stores the current text from the search bar
- isLoading (state, bool, 1b): Tracks the data being fetched with a loading bar

File: Login.js

Description: Loads the login page and manages the user input for authentication

Key Variables/Fields:

- username (state, string, 64B): Stores the text in the username field
- password(state, string, 64B): Stores the text in the password field
- errorMsg(state, string, 256B): Stores login error message

File: Notifications.js

Description: Fetches and displays user notifications

Key Variables/Fields:

- Notifications(state, array): Stores the list of notification objects for the logged-in user
- unreadCount(state, int): Stores the number of unread notifications

File: Reports.js

Description: Admin page that generates, views, and exports reports

Key Variables/Fields:

- reportData(state, object): Stores data returned from report generations
- startDate(state, string): Stores report start date
- endDate(state, string): Stores report end date

Backend File Descriptions (Flask)

File: app.py

Description: Main entry point for Flask app. Initializes the app, connects to MongoDB, and registers the API route blueprints.

Key Variables/Fields:

- app (Flask_instance): Main Flask application instance
- mongo (PyMongo_instance): object for interacting with the MongoDB database
- app.config[“MONGO_URI”] (string, 128B): Connection string for the database

File: equipment.py

Description: Defines all API routes related to equipment

Key Variables/Fields:

- Equipment_blueprint (Blueprint): The Flask blueprint object that organizes all routes

File: databaseManagement.py

Description: Abstracts all database logic. This keeps the route files clean and separates concerns.

Key Variables/Fields:

- deleteItem(flag, deleteID, fileID)
- addEquipment(metadata, fileData)
- addNotification(type, recipientID, senderID, dataVal)
- addAction(actionID, flag, dataID)
- addUser(username, password, type)
- updateItem(type, itemID, metadata)
- getEquipmentByID(equipmentID)
- searchEquipment(query, filters)

File: user.py

Description: Defines API routes related to the User class

Key Variables/Fields:

- User_blueprint: flask blueprint object

File: reports.py

Description: Defines API related routes to the Reports class

Key Variables/Fields:

- Reports_blueprint

File: notifications.py

Description: Defines API related routes to the Notifications class

Key Variables/Fields:

- Notifications_blueprint

4. Requirements Matrix

This section maps between the functional requirements defined in the SRS and the system components that will be implemented to satisfy them. This will help verify that each requirement is fully integrated into the system.

FR - Name	**Use Case #**	**Function Name(s) & Description**
Submit Request	1	validateRequestData(requestData) ● This will be a behind-the-scenes function that validates the form input when a user attempts to

		check out equipment. It will check to see if the equipment is already checked out, permissions, etc. sendNotification("request", adminId, userId, equipmentId) ● Notifies the admin that a user is requesting to check out equipment logAction(userId, "submit_request", adminId) ● This will record the equipment checkout, if approved by the admin, in the monthly report table and update the user table to note who currently has access to the equipment. Takes in adminID for logging.

Add Equipment	2	addEquipment(metadata)
		● Adds new equipment to the system
		validateEquipmentData(equipmentData)
		● This will be a behind-the-scenes function that will validate the form input when new equipment is added to the database

Equipment Filtering	3	searchEquipment(searchQuery, filters) ● Searches the equipment table by name and keywords. It will also take into account filters such as farm, model, and status, among others.
Logging In	4	authUser(credentials, userId) ● Validates the user to allow access to the system. This also includes ensuring they have the proper permissions to view certain aspects of the app.

Delete Equipment	5	deleteEquipment(type, equipmentId, NULL)
		● Deletes the equipment along with its metadata and attached files from the database/large file storage. The type will specify that this is an equipment deletion, NULL is there because a file isn't being deleted.
		logAction(adminId, "delete_equipment", equipmentId)
		● Record the deletion for monthly reports

User Account Creation	6	createUserAccount(username, password, userId)
		● Creates a new user account
		validateUserData(username, password, userId)
		● This will be a behind-the-scenes function that will validate the form input when a new account is created
		sendNotification("new_user", adminId, userId, NULL)
		● Notifies the admin of a new account creation for review. NULL value because no additional data needed in notification.

Viewing Equipment Details	7	fetchEquipmentData(equipmentId) ● This will be used to retrieve all assigned equipment metadata and display it to the user. When metadata is accessed from the database, the UUIDs of any attached files will be gathered and fetched from the large file storage.

Monthly Reports	8	generateMonthlyReport(month, farmId = “All”)
		● Will output a document containing the data from equipment checkouts, return, role changes, damage reports, and deletions
		exportReport()
		● Export the report as a PDF

Damage Reports	9	createDamageReport(equipmentId, metadata)
		● Logs the information provided by the user about the damaged equipment
		updateEquipmentStatus(equipmentId, "damaged")
		● Updates the equipment availability in the equipment table
		sendNotification("damage", adminId, userId, equipmentId)
		● Notify an admin of the damage
		logAction(userId, "report_damage", equipmentId)
		● Record the damage log for monthly reports

Editing Equipment Information	10	updateEquipmentData(equipmentId, metadata)
		● Updates any changed information about a piece of equipment
		validateEquipmentData(changedData)
		● This will be a behind-the-scenes function that will validate the form input when editing/adding equipment

Interactive Notifications	11	sendNotification(flag, recipientId, senderId, dataVal)
		● The definition of the function that sends notifications throughout the system. This is called by other functions. The type of notification sent will vary based on the flag. dataId param will hold extra information for the notification, like the equipment being checked out and/or damaged.
		getUserNotifications(userId)
		● Gets all of the notifications for a certain user to display in their notifications page
		deleteNotification(userId, notificationId)
		● Once a notification is resolved, it should be deleted from the database

Return Equipment	12	updateEquipmentStatus(equipmentId, "available")
		● Updates the equipment availability in the equipment table
		logAction(userId, "return_equipment", equipmentId)
		● Record the equipment return action for monthly reports

Attaching Files to Equipment	13	uploadEquipmentFile(equipmentId, fileData)
		● Uploads a file (adding it to the Oracle volume) and attaches its ID to the equipment record in the equipment table
		deleteEquipment(type, equipmentId, fileId)
		● Removes the file from storage and removes its reference to the equipment. The type will specify that this is a file deletion, and fileId is the reference to the file to delete.

Promoting a User	14	promoteUser(userId, newRole)
		● Updates the user's role to admin
		sendNotification("promoted", adminId, userId, newRole)
		● Notifies the user of the promotion
		logAction(adminId, "promote_user", userId)
		● Records the role change in the monthly report database

Appendix A - Agreement Between Customer and Contractor

This document outlines the agreed-upon system requirements between the customer and our project team. By signing below, both the customer and all team members confirm that they've read and understood the content of this document, and that it accurately reflects the expectations of the system. The project team agrees to build the system based on the descriptions provided here, and the customer confirms that these requirements meet their needs.

If any changes need to be made to this document in the future, the customer or project team can submit a request for the change. This process involves notifying the other party of the suggested change(s) and then conducting a joint review to understand how the change(s) may impact the project's scope, timeline, or deliverables. No changes will be made until both sides agree and sign off on the updated document.

Tyler Messerschmidt	Tyler Messerschmidt	11/14/25
Customer Name Printed	Customer Signature	Date
Bradan Craig	Bradan Craig	11/14/25
Team Member Name Printed	Team Member Signature	Date
Drew Marecek	Drew Marecek	11/14/25
Team Member Name Printed	Team Member Signature	Date
McKade Wing	McKade Wing	11/14/25
Team Member Name Printed	Team Member Signature	Date
Theodore Morin	Theodore Morin	11/14/25
Team Member Name Printed	Team Member Signature	Date

Customer Comments:

Appendix B - Team Review Sign-off

All team members have read through this document and agree with its content and structure. Each team member has had the opportunity to provide feedback and propose revisions during the creation of this document. By signing below, team members confirm their approval of the outlined system requirements. Any minor comments or points of clarification can be noted in the space provided.

Bradan Craig	Bradan Craig	11/14/25
Team Member Name Printed	Team Member Signature	Date

Comments:

Drew Marecek	Drew Marecek	11/14/25
Team Member Name Printed	Team Member Signature	Date

Comments:

McKade Wing	McKade Wing	11/14/25
Team Member Name Printed	Team Member Signature	Date

Comments:

Theodore Morin	Theodore Morin	11/14/25
Team Member Name Printed	Team Member Signature	Date

Comments:

Appendix C - Document Contributions

Each team member contributed to the development of this document to ensure the final version is accurate, precise, and complete. This included researching SRS structure, writing project overviews, creating UML diagrams, and creating functional and non-functional requirements. The table below shows each team member’s contributions, along with an estimated percentage of their work on the document.

Team Member	Contributions	Estimated %
Bradan Craig	Section 3.1	25%
Drew Marecek	Section 2.2	25%
McKade Wing	Section 2.1 + Section 4	25%
Theodore Morin	Section 3.2	25%