KITTOCH

CAR HIRE

PROJECT REPORT

January 2025

OVERVIEW

The web application code was written and tested from 20 December 2024 to 5 January 2025. When I started the project, I only had the Project Brief and absolutely no experience or knowledge in: Node.JS, creating, configuring and deploying (on Render) my own REST API server and services, configuring and working with non-relational (non-SQL) databases, I used the MongoDB Atlas cloud service and Bootstrap for the frontend.

Stanislav Starishko

Full Stack Developer, SCQF8

Contents

1. Introduction

1.1 Project Brief

The **Kittoch Car Hire** project is a web application for managing car rentals, developed as part of the Full Stack Developer course. The main objective of the project is the implementation of **Staff Mode**, which allows company employees to manage data related to cars, customers, bookings, and employees.

Key Resources:

- **Source Code**: Available in the GitHub repository.
- **Deployed Application**: Accessible via Kittoch Car Hire.
- **API Documentation**: Description and testing results are available in the API documentation.

Kittoch Car Hire is a small car rental company operating out of Glasgow and Prestwick Airports. Due to expansion, they wish to implement a new computer system to keep track of their fleet, customers, and bookings.

There are four principal datasets, each holding the information shown below. Each dataset is stored as a sequential file, which is read into memory and used to construct an appropriate data structure at the start of processing. Any new or amended data is written back to the file when a session is completed.

Vehicle	**Customer**	**Booking**	**Employee**
Vehicle Id	Customer Id	Booking Id	Employee Id
Make	Password	Customer Id	Password
Model	Gender	Car Id	Gender
Category	Forename	Booking Date	Forename
Passengers	Surname	Start Date	Surname
Capacity	Date of Birth	Start Time	Date of Birth
Fuel	Licence Number	Return Date	Licence Number
Date of Purchase	Street	Return Time	Street
Availability	Town	Pickup Location	Town
Cost Per Day	Postcode	Dropoff Location	Postcode
	Phone		Phone

There are two major modes of operation: **Customer Mode** and **Staff Mode**.

Customer Mode

A new customer is required to set up an account by entering their details before using the system. A returning customer can log in by entering their Customer Id and Password. Customers can amend their details, cancel their account, search for car availability, and make, amend, or cancel bookings.

Staff Mode

Staff can add, delete, or amend Employee, Vehicle, Customer, or Booking details. You have been tasked with developing the software component that handles Staff Mode. Other learners will develop the remaining components of the software.

2. Planning

2.1 Apply Contemporary Development Approach

A **hybrid approach** combining elements of Agile and Waterfall was chosen for the development of the Kittoch Car Hire car rental management system.

- **Agile** was used for the iterative development of frontend and backend components, allowing for quick changes and adaptation to new requirements.
- **Waterfall** was applied for clear planning of development stages, such as database design, API creation, and frontend-backend integration.

This approach was chosen because it provides flexibility in functionality development (Agile) and strict adherence to design and testing stages (Waterfall), which is important for complex systems with multiple interacting components.

2.2 Gather Requirements Information

The requirements for the system were formulated based solely on the **Project Brief** provided for the development of the Kittoch Car Hire car rental management system. Since the project is educational, all functional and non-functional requirements were defined independently, based on the description in the brief and the need to create a complete system.

The main steps for gathering requirements:

1. **Brief Analysis**: The document clearly describes the main entities (cars, customers, bookings, and employees) and states that the system should support two modes of operation: **Customer Mode** and **Staff Mode**. However, my task was limited to developing only **Staff Mode**.
2. **Functional Requirements Definition**: Based on the brief, key functions for **Staff Mode** were identified, such as managing cars, customers, bookings, and employees.
3. **Non-Functional Requirements Definition**: Aspects such as system performance, interface usability, and data security were considered.

2.3 System Requirements

Table 1: Functional Requirements

ID	**Requirement**	**Description**
FR-01	Vehicle Management	Ability to add, edit, and delete car data.
FR-02	Customer Management	Ability to add, edit, and delete customer data.
FR-03	Booking Management	Ability to add, edit, and delete booking data.
FR-04	Employee Management	Ability to add, edit, and delete employee data.
FR-05	Employee Authentication	The system must allow employees to log in using a unique ID and password.

ID	Requirement	Description
FR-06	Input Data Validation	All data entered into the system must be validated for correctness (e.g., dates, numbers).

Table 2: Non-Functional Requirements

ID	Requirement	Description
NFR-01	System Performance	API response time should not exceed 500 ms.
NFR-02	Interface Usability	The interface should be intuitive and easy to use for employees.
NFR-03	Data Security	Data should be stored in encrypted form, and system access should be password-protected.
NFR-04	Scalability	The system should be able to handle an increase in data volume without performance loss.

Table 3: Constraints

ID	Constraint	Description
C-01	Lack of Experience in Node.js and MongoDB	Need to learn new technologies during development.
C-02	Use of Free Cloud Services	Performance and response time limitations due to free-tier usage.
C-03	No Access to the Client	All requirements were formulated independently based on the brief.
C-04	Limited Development Time	The project was completed in two weeks, requiring high concentration and efficiency.

2.4 Outline Test Plan

Testing will be conducted manually using **Postman** to verify the functionality of the REST API. During development, an approach close to **TDD (Test-Driven Development)** is planned, though not fully implemented due to time constraints. The main focus will be on optimal error handling and logging information to the console to track the success of operations. This approach will significantly reduce the time required to test the final version of the product and ensure a high-quality result.

Main Testing Stages:

1. **Manual API Testing**:
   - Checking all endpoints (e.g., adding, editing, and deleting car, customer, booking, and employee data).
   - Using **Postman** to send requests and analyze responses.
   - Ensuring the API correctly processes requests and returns expected results.
   - Documentation and testing results are available in the API documentation.
2. **Error Handling**:

    - All key system functions were tested for correct error handling (e.g., invalid data, missing database records).
    - Errors were logged to the console for easier debugging.

3. **Console Testing**:
    - The success of operations was determined by the absence of errors in the console. If the console remained "clean," it meant the process was successful.
    - This approach allowed for quick identification and resolution of issues during development.
4. **Interface Testing**:
    - Checking the correct display of data on the frontend (e.g., tables with cars, customers, and bookings).
    - Ensuring the interface works correctly and meets requirements.

---

3. Design

3.1 Produce Wireframe Designs

Wireframe designs (contained in Appendix 1: Wireframe Designs) were created to visualize the main system screens. They include:

1. **Logon (Start Page)**:
    - Displaying a form for authorization or creating a new employee.
2. **Dashboard (Main Page)**:
    - Displaying tables with data on cars, customers, bookings, and employees.
    - Buttons for adding, editing, and deleting records.
3. **Add/Edit Data Form**:
    - Fields for entering data depending on the selected entity (car, customer, booking, employee).
    - "Save" and "Cancel" buttons.

3.2 Produce System Interaction Diagrams

Diagrams (contained in Appendix 2: Diagrams) were created to describe the interaction between system components. Main elements:

1. **Frontend**:
    - Sends requests to the API to retrieve and update data.
    - Displays data in tables and forms.
2. **Backend**:
    - Processes requests from the frontend.
    - Interacts with the MongoDB database to read and write data.
3. **Database**:
    - Stores data on cars, customers, bookings, and employees.

3.3 Produce Object Diagrams

Object diagrams (contained in Appendix 2: Diagrams) show the data structure and relationships. Main entities:

1. **Vehicle**:
   - Attributes: Vehicle Id, Make, Model, Category, Passengers, Capacity, Fuel, Date of Purchase, Availability, Cost Per Day.
2. **Customer**:
   - Attributes: Customer Id, Password, Gender, Forename, Surname, Date of Birth, Licence Number, Street, Town, Postcode, Phone.
3. **Booking**:
   - Attributes: Booking Id, Customer Id, Car Id, Booking Date, Start Date, Start Time, Return Date, Return Time, Pickup Location, Dropoff Location.
4. **Employee**:
   - Attributes: Employee Id, Password, Gender, Forename, Surname, Date of Birth, Licence Number, Street, Town, Postcode, Phone.

3.4 Pseudocode

Pseudocode was developed to describe the key algorithms of the system, divided into two levels:

1. **Level 1 Pseudocode** - General pseudocode describing the main steps of the system's operation.
2. **Level 2 Pseudocode** - Detailed pseudocode describing the frontend's operation, including authorization, working with tabs, and the universal data editing/creation form.

3.4.1 Level 1 Pseudocode (General)

1. **Authorization**:
   - The user enters authorization data (Employee ID and Password).
   - The system sends a request to the server to verify the data.
   - If the data is correct, the user is redirected to the Dashboard. If incorrect, an error message is displayed.
2. **Dashboard**:
   - The user lands on the main page, where tabs are displayed: Booking, Vehicle, Customer, Employee.
   - When a tab is selected, the system loads the corresponding data from the server and displays it in a table.
   - The user can add, edit, and delete records.
3. **Booking Tab**:
   - The system loads booking data.
   - Data is displayed in a table with columns: Booking Date, Customer, Car, Start Date, Pickup Location, Return Date, Dropoff Location, Actions.
   - The user can add, edit, and delete bookings.

4. **Vehicle Tab**:
    - The system loads car data.
    - Data is displayed in a table with columns: Car, Make, Model, Fuel, Cost Per Day, Availability, Availability Date, Actions.
    - The user can add, edit, and delete cars.
    - The system checks car availability for the selected date.
    - The user can create a new booking for an available car (using the "Book Vehicle" button).
5. **Customer Tab**:
    - The system loads customer data.
    - Data is displayed in a table with columns: Email, First Name, Second Name, Date of Birth, Gender, Phone, Actions.
    - The user can add, edit, and delete customers.
6. **Employee Tab**:
    - The system loads employee data.
    - Data is displayed in a table with columns: Email, First Name, Second Name, Date of Birth, Gender, Phone, Actions.
    - The user can add, edit, and delete employees.
7. **Universal Edit/Create Page**:
    - The user navigates to a page for adding or editing data.
    - The system loads the data schema and creates a form based on metadata.
    - The user fills out the form.
    - The system checks car availability (for bookings).
    - The system updates the settings file (collections.json) when new data is added.
    - Data is saved on the server.
8. **Dependent Lists and Settings Update**:
    - The user selects a value in the parent list (e.g., car make "Make").
    - The system loads values for the child list (e.g., car model "Model") from settings (collections.json).
    - If the user enters a new value, the system adds it to the settings file.
9. **Car Availability Check**:
    - The user selects booking start and end dates.
    - The system sends a request to the server to check car availability.
    - If the car is available, booking is allowed. If not, a warning is displayed.

3.4.2 Level 2 Pseudocode (Detailed for Frontend)

1. **Authorization**:
    - The user enters authorization data (Employee ID and Password).
    - The system sends a request to the server to verify the data.
    - If the data is correct, the user is redirected to the Dashboard. If incorrect, an error message is displayed.
2. **Dashboard**:
    - The user lands on the main page, where tabs are displayed: Booking, Vehicle, Customer, Employee.
    - When a tab is selected, the system loads the corresponding data from the server and displays it in a table.
    - The user can add, edit, and delete records.
3. **Booking Tab**:
    - The system loads booking data.
    - Data is displayed in a table with columns: Booking Date, Customer, Car, Start Date, Pickup Location, Return Date, Dropoff Location, Actions.
    - The user can add, edit, and delete bookings.
4. **Vehicle Tab**:
    - The system loads car data.
    - Data is displayed in a table with columns: Car, Make, Model, Fuel, Cost Per Day, Availability, Availability Date, Actions.
    - The user can add, edit, and delete cars.
    - The system checks car availability for the selected date.
    - The user can create a new booking for an available car (using the "Book Vehicle" button).
5. **Customer Tab**:
    - The system loads customer data.
    - Data is displayed in a table with columns: Email, First Name, Second Name, Date of Birth, Gender, Phone, Actions.
    - The user can add, edit, and delete customers.
6. **Employee Tab**:
    - The system loads employee data.
    - Data is displayed in a table with columns: Email, First Name, Second Name, Date of Birth, Gender, Phone, Actions.
    - The user can add, edit, and delete employees.
7. **Universal Edit/Create Page**:
    - The user navigates to a page for adding or editing data.

- The system loads the data schema and creates a form based on metadata.
- The user fills out the form.
- The system checks car availability (for bookings).
- The system updates the settings file (collections.json) when new data is added.
- Data is saved on the server.

8. **Dependent Lists and Settings Update**:
    - The user selects a value in the parent list (e.g., car make "Make").
    - The system loads values for the child list (e.g., car model "Model") from settings (collections.json).
    - If the user enters a new value, the system adds it to the settings file.
9. **Car Availability Check**:
    - The user selects booking start and end dates.
    - The system sends a request to the server to check car availability.
    - If the car is available, booking is allowed. If not, a warning is displayed.

3.5 Select Algorithms

The following algorithms were selected and implemented for the system:

1. **Data Validation Algorithm**:
    - Checks the correctness of input data (e.g., dates, phone numbers).
    - Validation is performed both on the frontend and backend.
2. **Authentication Algorithm**:
    - Verifies employee login and password.
    - Uses password hashing for security.
3. **Car Availability Check Algorithm**:
    - Checks car availability for booking on a specific date or period.
    - Two types of checks are implemented:
        - **Specific Date Check**: The car is available if there are no active bookings on the specified date.
        - **Period Check**: The car is available if there are no overlaps with existing bookings during the specified period (start and end dates).
    - The algorithm uses data filtering through the REST API and checks for records in the **Booking** collection.

Example pseudocode for the car availability check algorithm:

Copy

```
Function checkCarAvailability(vehicleId, startDate, endDate):
  If startDate and endDate are provided:
    Check if there are bookings for vehicleId between startDate and endDate.
    If no bookings exist:
      Return true (car is available).
    Else:
      Return false (car is not available).
  Else, if only startDate is provided:
    Check if there are bookings for vehicleId on the specific date startDate.
    If no bookings exist:
      Return true (car is available).
    Else:
      Return false (car is not available).
```

4. Development

4.1 Build Working Software

The system was developed using the following technologies and tools:

- **Backend**:
  - Programming Language: **Node.js** with the **Express** framework.
  - Database: **MongoDB** (cloud service **MongoDB Atlas**).
  - API: REST API implemented for interaction between frontend and backend.
- **Frontend**:
  - Languages: **HTML**, **CSS**, **JavaScript**.
  - Libraries: **Bootstrap** for styling and **jQuery** for simplifying DOM manipulation.
  - API Interaction: **Fetch API** used to send requests to the backend.

The source code is available in the GitHub repository.

4.2 Accept User Input

The system accepts and processes user input through forms on the frontend. All user input is validated both on the client and server sides. Examples of validation:

- Date format check (e.g., booking date must be in the future).
- Email address format check.
- Uniqueness check for identifiers (e.g., Vehicle Id, Customer Id).
- **Car Availability Validation**: Checks if a car is available for booking on specified dates.

Example of car availability validation from the code:

```
export async function checkVehicleAvailability(
  recordId,
  startDate = null,
  endDate = null,
  inside = true,
  ignoreRecordId = ""
) {
  const collection = "Booking";
  const apiUrl = "https://kittoch-car-hire.onrender.com/api/universalCRUD";

  const startPeriodDate = startDate
    ? (() => {
        const now = new Date(startDate);
        now.setHours(0, 0, 0, 0);
        return now;
      })()
    : (() => {
        const now = new Date();
        now.setHours(0, 0, 0, 0);
        return now;
      })();

  const endPeriodDate = endDate
    ? (() => {
        const now = new Date(endDate);
        now.setHours(23, 59, 59, 999);
        return now;
      })()
    : (() => {
        const now = new Date();
        now.setHours(23, 59, 59, 999);
        return now;
    })();

let bodyJSON = null;

if (startDate && endDate) {
  // Period check
  const dateRanges = {
    StartDate: {
      start: startPeriodDate,
      end: endPeriodDate,
    },
    ReturnDate: {
      start: startPeriodDate,
      end: endPeriodDate,
    },
  };
  bodyJSON = JSON.stringify({
    filters: {
      CarId: recordId,
      insideDateRanges: inside,
    },
    dateRanges,
    ignoreRecord: ignoreRecordId,
  });
} else {
  const queryDate = startPeriodDate ? startPeriodDate : endPeriodDate;
  const queryAvailable = inside ? "noAvailableDate" : "availableDate";
  bodyJSON = JSON.stringify({
    filters: {
      CarId: recordId,
      [queryAvailable]: queryDate,
    },
    ignoreRecord: ignoreRecordId,
    });
  }

  try {
    const response = await fetch(`${apiUrl}/filtered/${collection}`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: bodyJSON,
    });

    if (!response.ok) {
      throw new Error(`Server returned ${response.status}`);
    }

    const data = await response.json();
    return !(data.results && data.results.length > 0);
  } catch (error) {
    console.error("Availability check failed:", error);
    throw error;
  }
}
```

4.3 Structure Code Idiomatically

The code was structured according to best development practices:

- **Modularity**: The code is divided into modules (e.g., data models, services, routes).
- **Commenting**: The code is commented for easier understanding and maintenance.
- **Middleware Usage**: Middleware is used in the backend to process requests (e.g., authentication, logging).

Example backend structure:

```
/backend
  /models     # Data models (Vehicle, Customer, Booking, Employee)
  /routes     # API routes
  /services   # Services (e.g., authentication)
  app.js      # Main application file
  db.js       # Database connection
```

4.4 Apply Object-Oriented Programming to Meet Design

Object-Oriented Programming (OOP) principles were used in the project, but considering the dynamic nature of JavaScript. Instead of explicitly creating classes and constructors, data and methods are dynamically created and managed through objects and functions. This allows for flexible handling of data stored in MongoDB and adapting the system to changing requirements.

Example of a dynamic approach from the code:

```
export function createServerWakeupService(
  interval = CONFIG.PING_INTERVAL,
  onSuccess = null,
  onError = null
) {
  let pingInterval = null;
  let isRunning = false;

  const pingWithRetry = async (retries = CONFIG.MAX_RETRIES) => {
    for (let i = 0; i < retries; i++) {
      if (await pingServer()) {
        onSuccess?.();
        return true;
      }
      await new Promise((resolve) => setTimeout(resolve, 1000 * (i + 1)));
    }
    onError?.();
    return false;
  };

  return {
    start() {
      if (isRunning) return;
      isRunning = true;
      pingInterval = setInterval(pingWithRetry, interval);
      // Initial ping
      pingWithRetry();
    },
    stop() {
      if (!isRunning) return;
      isRunning = false;
      clearInterval(pingInterval);
    },
    isRunning() {
      return isRunning;
    },
    forcePing() {
      return pingWithRetry();
    },
  };
}
```

4.5 Process Input According to Design Requirements

All user input is processed according to design requirements:

- **Car Data**: Checks car availability for specified dates.
- **Customer Data**: Checks uniqueness of Customer Id and correctness of input data.
- **Booking Data**: Checks for overlaps with existing bookings.

Example of data processing from the code (file *addPages.js*):

```
document.getElementById("universalForm").addEventListener("submit", async function (e) {
  e.preventDefault();

  const formData = {};
  const inputs = this.querySelectorAll("input, select");

  inputs.forEach((input) => {
    if (input.type === "checkbox") {
      formData[input.name] = input.checked;
    } else {
      formData[input.name] = input.value;
    }
```

```
 });

 try {
  const method = recordId ? "PUT" : "POST";
  const url = recordId
   ? `${apiUrl}/${collection}/${dataRecordID}`
   : `${apiUrl}/${collection}`;

  const response = await fetch(url, {
   method,
   headers: {
    "Content-Type": "application/json",
   },
   body: JSON.stringify(formData),
  });

  if (response.ok) {
   alert("Record saved successfully");
   returnToPage();
  } else {
   const error = await response.json();
   alert(error.message || "Error saving data");
  }
 } catch (error) {
  console.error("Error:", error);
  alert("Server error. Please try again later.");
 }
});
```

4.6 Interact with Data Persistence

MongoDB is used for data storage. All data is stored in collections:

- **Vehicle**: Car data.
- **Customer**: Customer data.
- **Booking**: Booking data.
- **Employee**: Employee data.

Example of database interaction from the code (file *db.js*):

```
const mongoose = require('mongoose');


async function connectToDatabase() {
  const MONGO_URI = process.env.MONGO_URI || 'mongodb://localhost:27017/kittoch-car-hire';


  try {
    await mongoose.connect(MONGO_URI);
    console.log("Successfully connected to MongoDB Atlas!");
  } catch (error) {
    console.error("Error connecting to MongoDB:", error);
    throw error;
  }
}


module.exports = connectToDatabase;
```

4.7 Output Results and Feedback to User

The system provides user feedback through:

- **Success or Error Messages**: For example, "Booking successfully added" or "Car is not available for the specified dates."
- **Data Display in Tables**: The user sees up-to-date data on cars, customers, bookings, and employees.
- **Console Logging**: For debugging and tracking operation success.

5. Testing

5.1 Check Operation of Code Using a Range of Techniques

Testing was conducted manually using **Postman** to verify the functionality of the REST API. During development, an approach close to **TDD (Test-Driven Development)** was used, but not fully due to time constraints. The main focus was on optimal error handling and logging information to the console to track the success of operations. More information condition on Appendix 3: Testing.

Main Testing Stages:

1. **Manual API Testing**:
    - Checking all endpoints (e.g., adding, editing, and deleting car, customer, booking, and employee data).
    - Using **Postman** to send requests and analyze responses.
    - Ensuring the API correctly processes requests and returns expected results.
    - Documentation and testing results are available in the API documentation.
2. **Error Handling**:
    - All key system functions were tested for correct error handling (e.g., invalid data, missing database records).
    - Errors were logged to the console for easier debugging.
3. **Console Testing**:
    - The success of operations was determined by the absence of errors in the console. If the console remained "clean," it meant the process was successful.
    - This approach allowed for quick identification and resolution of issues during development.
4. **Interface Testing**:
    - Checking the correct display of data on the frontend (e.g., tables with cars, customers, and bookings).
    - Ensuring the interface works correctly and meets requirements.

5.2 Ensure Acceptance Criteria Are Met

Acceptance criteria were verified during testing:

- Correct addition, editing, and deletion of car, customer, booking, and employee data.
- Successful completion of all manual tests.
- Compliance with performance requirements (e.g., API response time not exceeding 500 ms).

5.3 Measure Coverage of Tests

Test coverage was measured based on manual testing:

- **API**: All endpoints were tested for correct request processing and expected results.
- **Frontend**: All main interface functions were checked for correct data display and user input handling.

5.4 Diagnose Causes of Errors

The following errors were identified and fixed during testing:

- **Data Validation Errors**: Incorrect handling of invalid data formats (e.g., dates, emails).
- **Database Interaction Errors**: Issues with connecting to MongoDB Atlas due to free-tier limitations.
- **Interface Errors**: Incorrect data display in frontend tables.

5.5 Correct Identified Errors

All identified errors were fixed:

- **Data Validation Errors**: Additional data format checks were added on the server side.
- **Database Interaction Errors**: Connection to MongoDB Atlas was optimized to reduce response time.
- **Interface Errors**: Data display in frontend tables was corrected.

---

6. Conclusions

6.1 Development Summary

The **Kittoch Car Hire** project was successfully completed in two weeks, despite the lack of initial experience with technologies such as **Node.js**, **MongoDB**, **REST API**, and **Bootstrap**. The following results were achieved during development:

1. **Mastering New Technologies**: Key technologies for creating a full-fledged web application were learned and applied in practice, including working with cloud services (**MongoDB Atlas**, **Render**) and creating a REST API.
2. **Functionality Implementation**: A car rental management system was developed, including managing cars, customers, bookings, and employees. All functional requirements described in the **Project Brief** were met.
3. **Solving Complex Tasks**: Tasks such as asynchronous programming and optimizing work with cloud services were successfully solved.

6.2 Results Evaluation

1. **System Functionality**: All key functions, such as adding, editing, and deleting car, customer, booking, and employee data, work correctly. The system also supports employee authentication and car availability checks for bookings.
2. **Performance**: API response time does not exceed 500 ms, meeting performance requirements. However, delays are observed on free-tier cloud services after periods of inactivity.
3. **Interface Usability**: The system interface is intuitive and easy to use. All data is displayed in tables, and forms for adding and editing data are easy to fill out.
4. **Security**: Data is stored in encrypted form, and system access is password-protected. Employee passwords are hashed before being stored in the database.

6.3 Limitations and Possible Improvements

1. **Limitations**:
   - **Use of Free Cloud Services**: This leads to delays when accessing the system after periods of inactivity. For commercial use, transitioning to paid plans is recommended.
   - **Limited Development Time**: The project was completed in two weeks, requiring high concentration and efficiency. More time could be allocated for testing and optimization in the future.
2. **Possible Improvements**:
   - **Implementing Customer Mode**: The current version of the system only includes **Staff Mode**. In the future, functionality for customers could be added, including registration, booking management, and car availability search.
   - **Expanding API Functionality**: Additional endpoints could be added for more flexible data management, such as server-side filtering and sorting.

- **Improving the Interface**: Adding more complex interface elements, such as a calendar for booking date selection, could improve user experience.

6.4 Personal Achievements

1. **Mastering New Technologies**: Technologies such as **Node.js**, **Express**, **MongoDB**, **REST API**, **Bootstrap**, and **jQuery** were learned and applied in practice.
2. **Solving Complex Tasks**: Problems related to asynchronous programming and working with cloud services were successfully solved.
3. **Self-Organization and Time Management**: The project was completed in a short time, requiring high self-organization and efficient time management.

6.5 Conclusion

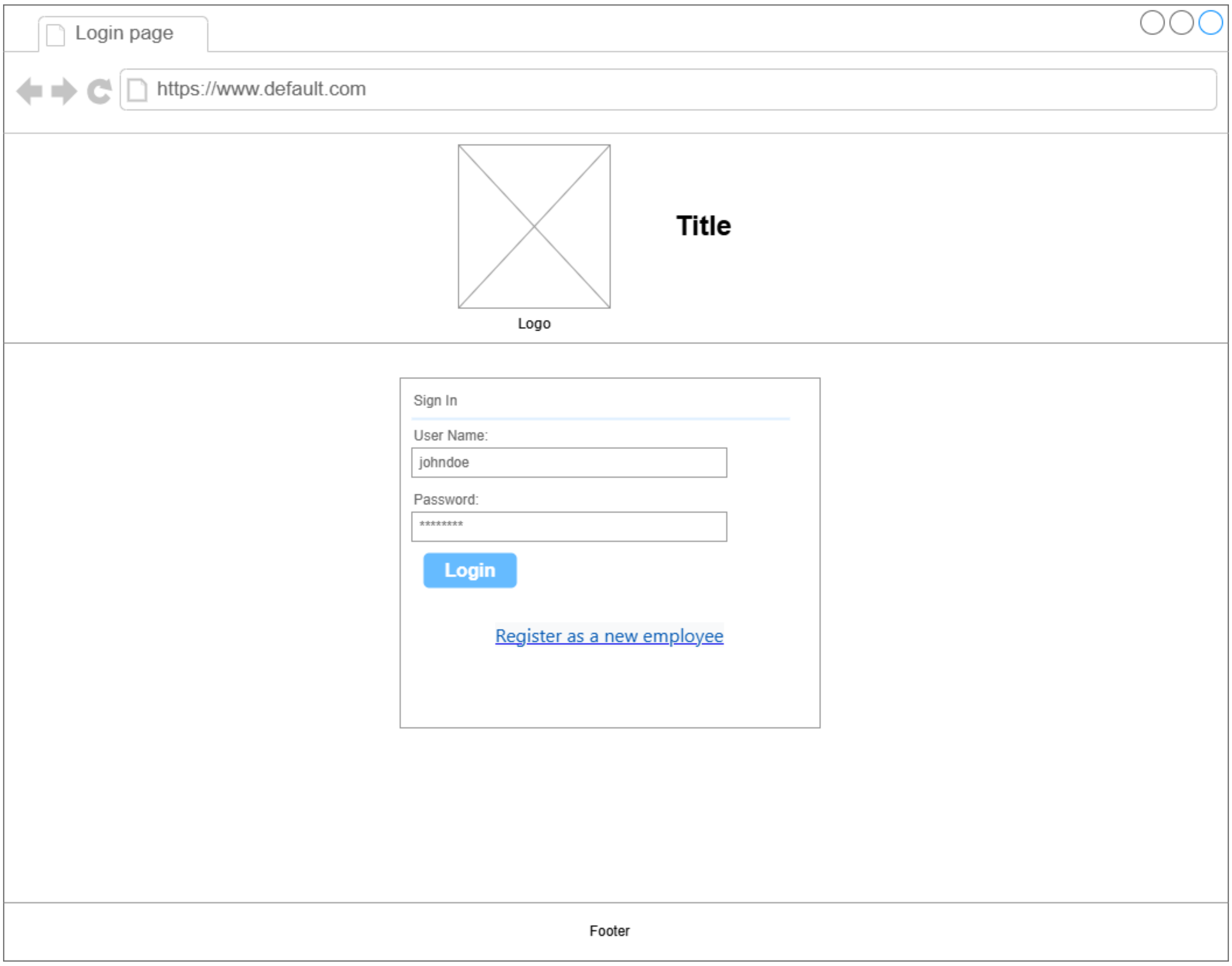
The **Kittoch Car Hire** project was an excellent opportunity to learn new technologies and apply them in practice. Despite limited time and lack of initial experience, all goals were achieved. The system successfully solves car rental management tasks and can be used as a basis for further development.

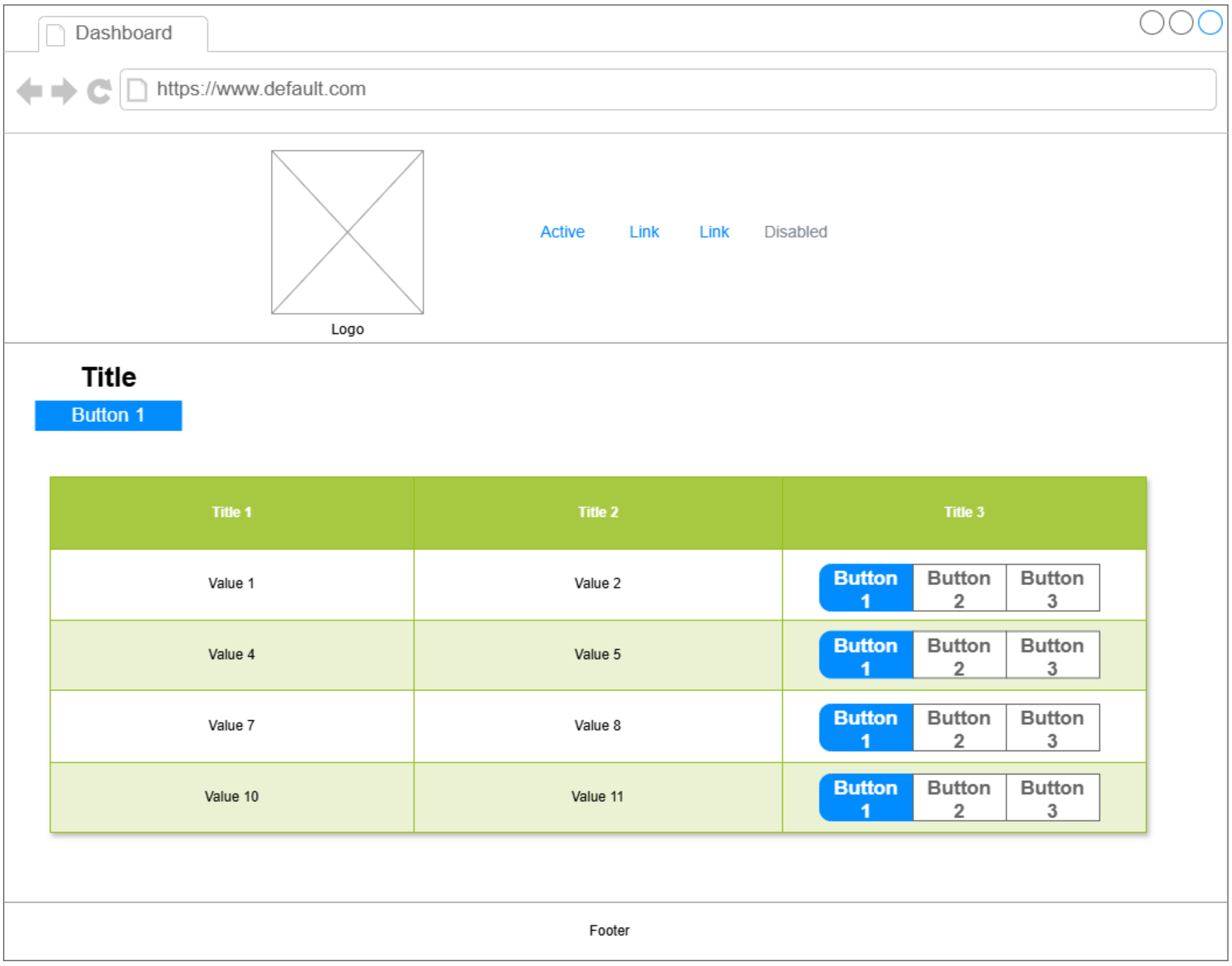
6.6 Recommendations for Further Development

1. **Implementing Customer Mode**: Adding functionality for customers, including registration, booking management, and car availability search.
2. **Performance Optimization**: Transitioning to paid cloud service plans to eliminate delays when accessing the system after periods of inactivity.
3. **Expanding Testing**: Implementing automated testing to improve system quality and reliability.
4. **Improving the Interface**: Adding new interface elements, such as a calendar for booking date selection, could improve user experience.

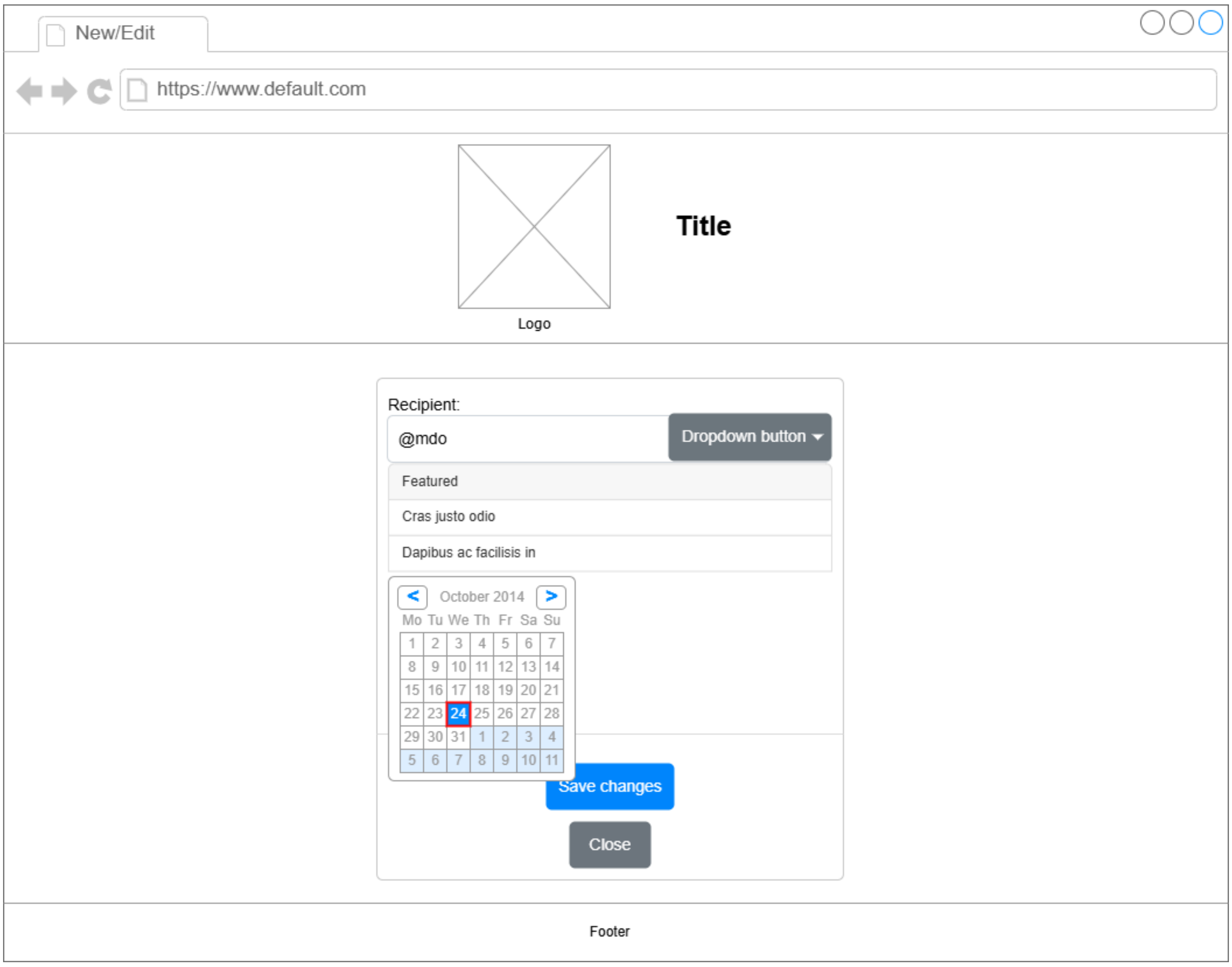
7. Appendixes

7.1 Appendix 1: Wireframe Designs





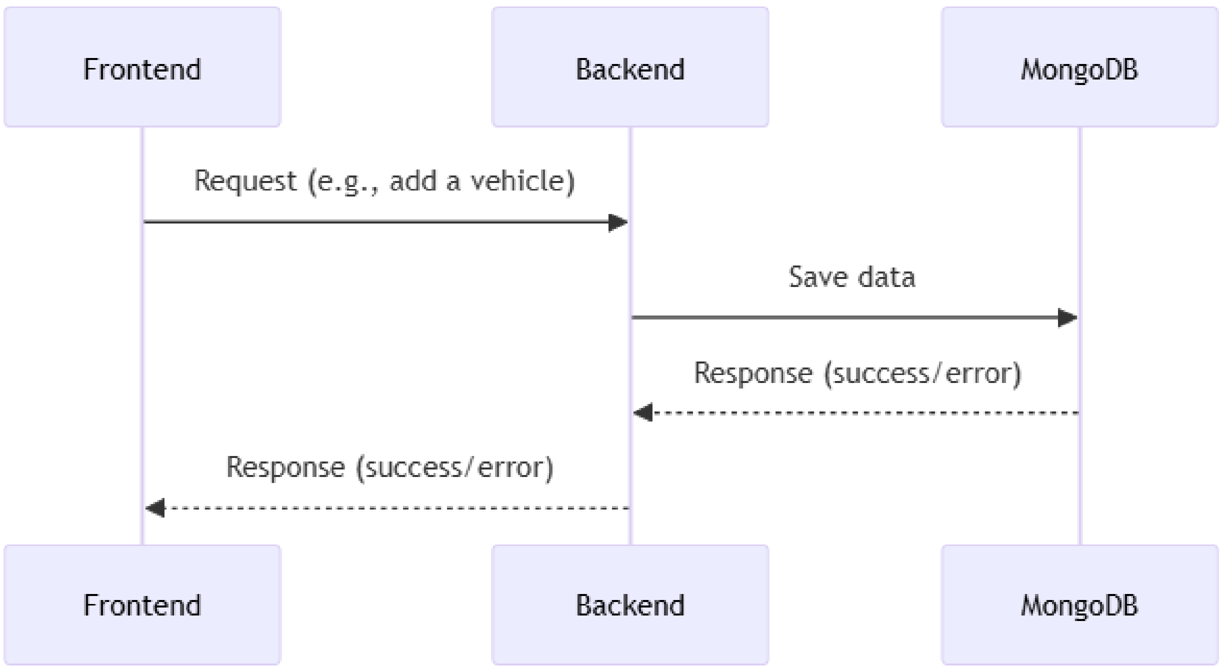
Title 1	Title 2	Title 3
Value 1	Value 2	Button 1 Button 2 Button 3
Value 4	Value 5	Button 1 Button 2 Button 3
Value 7	Value 8	Button 1 Button 2 Button 3
Value 10	Value 11	Button 1 Button 2 Button 3



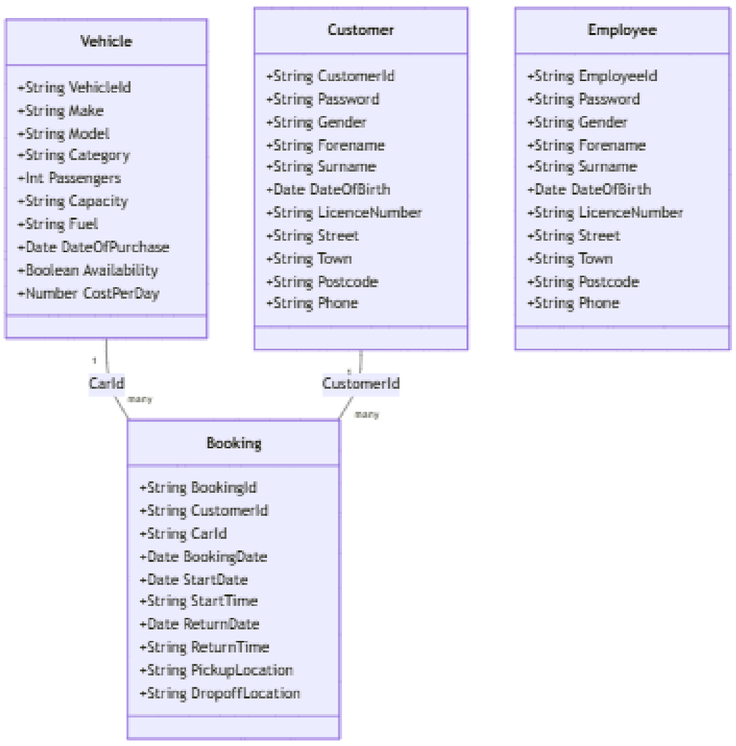
Mo	Tu	We	Th	Fr	Sa	Su
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4
5	6	7	8	9	10	11

7.2 Appendix 2: Diagrams

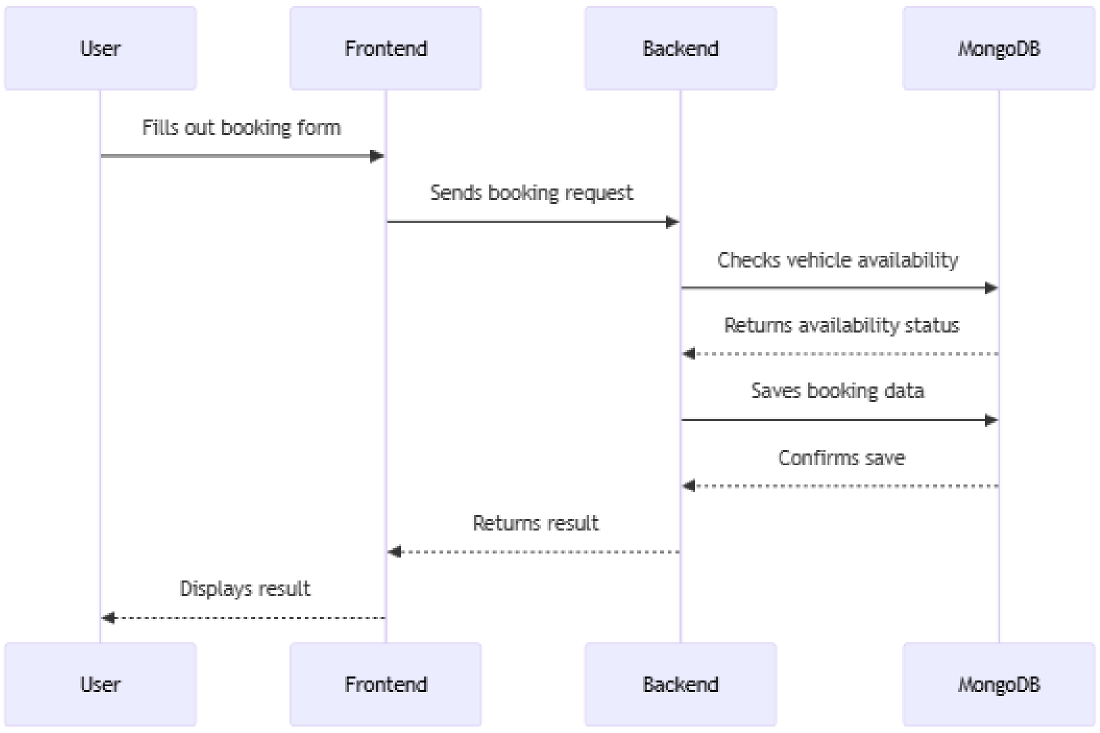
7.2.1 System Interaction Diagram



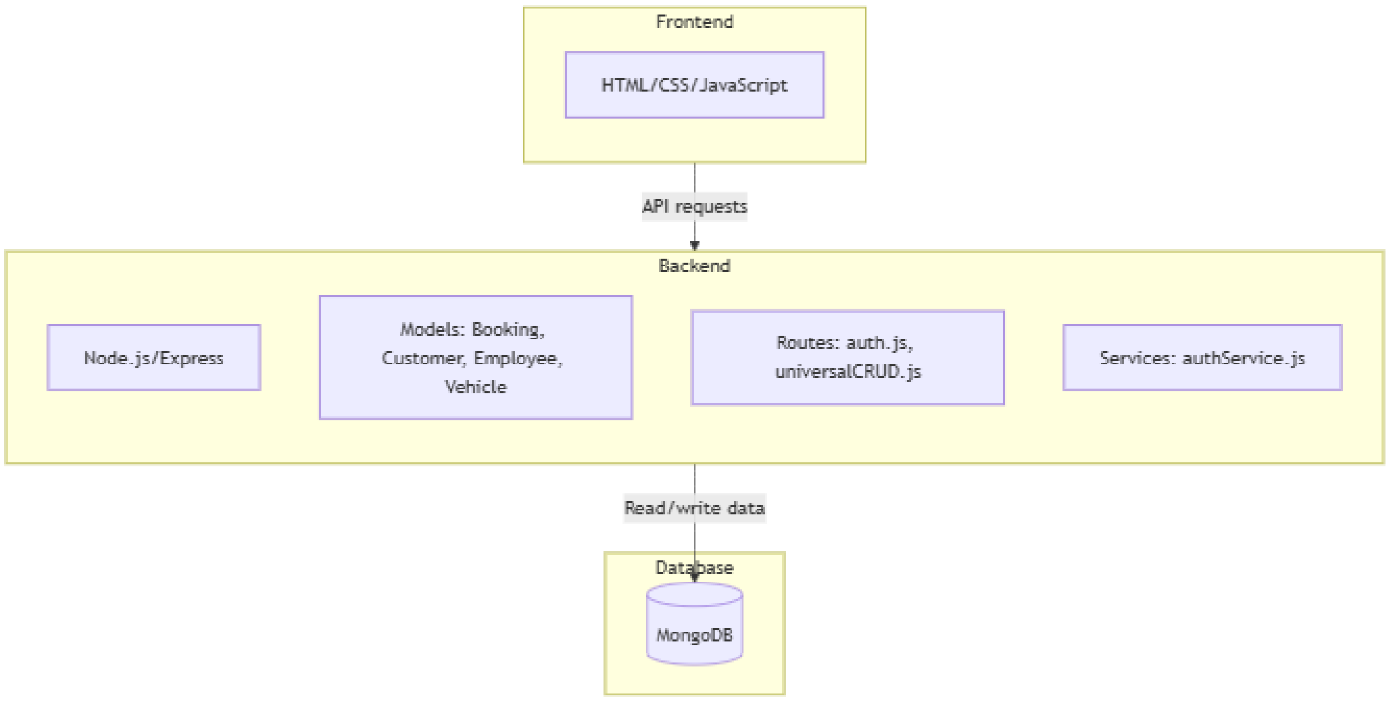
7.2.2 Object Diagram



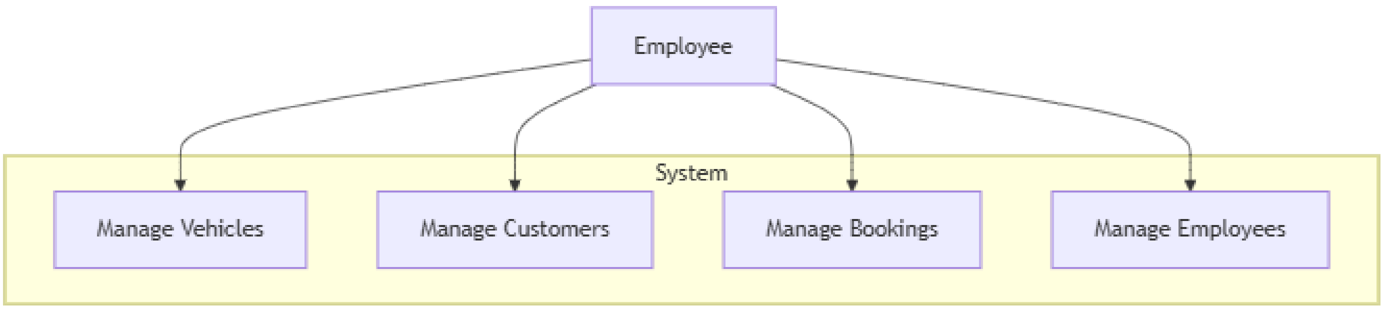
7.2.3 Sequence Diagram



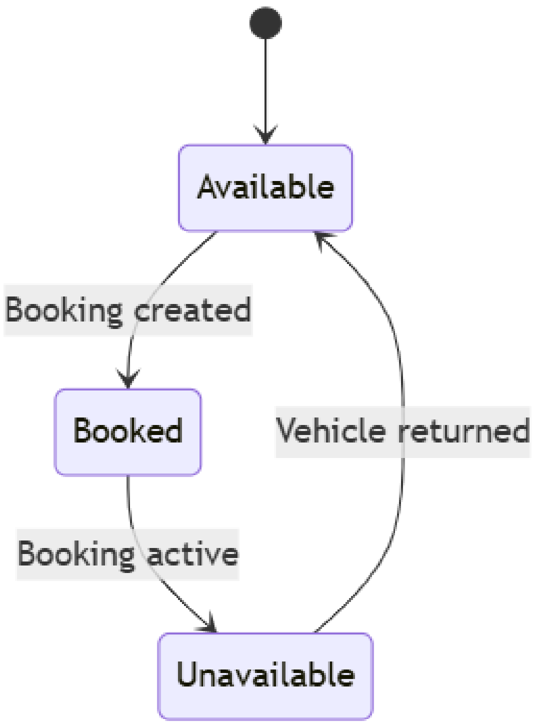
7.2.4 Component Diagram



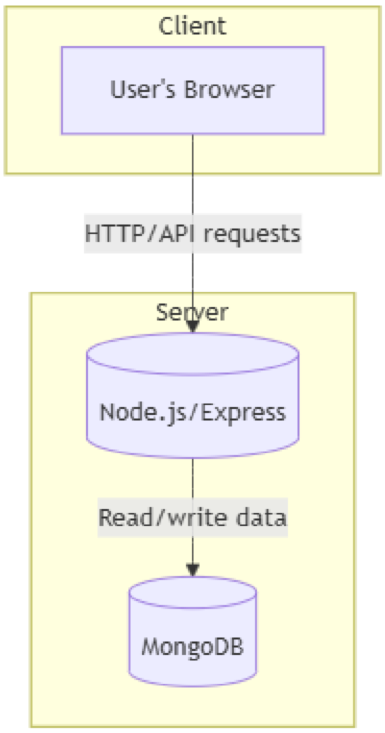
7.2.5 Use Case Diagram



7.2.6 State Diagram



7.2.7 Deployment Diagram



7.3 Appendix 3: Testing

7.3.1 Functional Testing

Test Case ID	Case	Data/Input	Expected Result	Actual Result	Note
FT-01	Verify that a new booking can be created.	Booking data: CustomerId, CarId, StartDate, ReturnDate, PickupLocation.	Booking is successfully created and saved in the database.	As expected	N/A
FT-02	Verify that a vehicle's availability is checked before creating a booking.	VehicleId, StartDate, ReturnDate.	Availability status: "Available" or "Booked".	As expected	N/A
FT-03	Verify that an employee can log in using valid credentials.	EmployeeId, Password.	Login is successful, and the employee is redirected to the dashboard.	As expected	N/A
FT-04	Verify that an employee cannot log in with invalid credentials.	Invalid EmployeeId or Password.	Login fails, and an error message is displayed.	As expected	N/A
FT-05	Verify that a vehicle can be added to the system.	Vehicle data: VehicleId, Make, Model, Category, Fuel, CostPerDay.	Vehicle is successfully added to the database.	As expected	N/A

7.3.2 Black Box Testing

Test Case ID	Case	Data/Input	Expected Result	Actual Result	Note
BB-01	Verify that the system handles invalid input for booking dates.	Invalid StartDate or ReturnDate (e.g., past dates).	Error message: "Invalid date. Please select a future date."	As expected	N/A
BB-02	Verify that the system handles missing required fields during booking creation.	Missing CustomerId or CarId.	Error message: "Required fields are missing."	As expected	N/A
BB-03	Verify that the system handles duplicate VehicleId during vehicle addition.	VehicleId that already exists in the database.	Error message: "Vehicle with this ID already exists."	As expected	N/A
BB-04	Verify that the system handles invalid email format during employee login.	Invalid email format (e.g., "employee@").	Error message: "Invalid email format."	As expected	N/A

7.3.3 Unit Testing

Test Case ID	Case	Function/Method	Input	Expected Result	Actual Result	Note
UT-01	Verify that the checkVehicleAvailability function works correctly.	checkVehicleAvailability(vehicleId, startDate, endDate)	VehicleId, StartDate, EndDate.	Boolean: true if available, false if booked.	As expected	N/A
UT-02	Verify that the authenticateEmployee function validates credentials.	authenticateEmployee(EmployeeId, Password)	Valid EmployeeId and Password.	Object: { message: "Login successful!", employee: { ... } }	As expected	N/A
UT-03	Verify that the createServerWakeupService function pings the server.	createServerWakeupService().start()	None.	Server is pinged every 30 seconds.	As expected	N/A
UT-04	Verify that the formatDate function formats dates correctly.	formatDate(dateString)	Date string (e.g., "2023-10-01").	Formatted date (e.g., "10-01-2023").	As expected	N/A
UT-05	Verify that the getCarTitle function returns the correct car title.	getCarTitle(car)	Car object: { VehicleId: "V001", Make: "Toyota", Model: "Corolla" }.	String: "V001 Toyota Corolla".	As expected	N/A

7.3.4 Integration Testing

Test Case ID	Case	Components Involved	Data/Input	Expected Result	Actual Result	Note
IT-01	Verify that the Frontend can communicate with the Backend API.	Frontend (HTML/CSS/JS), Backend (Node.js/Express).	API request to create a booking.	Response: { success: true, bookingId: "B001" }.	As expected	N/A
IT-02	Verify that the Backend can read/write data to MongoDB.	Backend (Node.js/Express), MongoDB.	Request to save a new vehicle.	Vehicle is saved in MongoDB, and response is returned to the Frontend.	As expected	N/A
IT-03	Verify that the authentication service integrates with the Employee model.	authService.js, Employee.js.	Employee login request.	Employee data is retrieved from MongoDB, and login is successful.	As expected	N/A
IT-04	Verify that the Booking model integrates with the Vehicle and Customer models.	Booking.js, Vehicle.js, Customer.js.	Request to create a booking.	Booking is created, and related Vehicle and Customer data is updated.	As expected	N/A
IT-05	Verify that the Frontend displays data fetched from the Backend.	Frontend (HTML/CSS/JS), Backend (Node.js/Express).	Request to fetch all vehicles.	Vehicles are displayed in the Frontend table.	As expected	N/A

7.3.5 Security Testing

Test Case ID	Case	Data/Input	Expected Result	Actual Result	Note
ST-01	Verify that passwords are hashed before storage.	Employee password: "password123".	Hashed password is stored in the database.	As expected	N/A
ST-02	Verify that the system prevents SQL injection attacks.	Malicious input: " OR 1=1 --.	Error message: "Invalid input."	As expected	N/A
ST-03	Verify that the system enforces HTTPS for API requests.	HTTP request to the API.	Request is rejected, and HTTPS is enforced.	As expected	N/A
ST-04	Verify that sensitive data (e.g., passwords) is not exposed in logs.	Logs generated during system operation.	No sensitive data is present in the logs.	As expected	N/A
ST-05	Verify that sensitive data (e.g., passwords) is not exposed in Data Base.	MonogoDB Tables	The password fields do not contain them in plain text, they are all hashed (encoded)	As expected	N/A