Gayan Withana Gamage

WEB APPLICATION DEVELOPMENT PROJECT FOR VEHICLE PARKING RESERVATION SYSTEM IN STUDENT APARTMENTS

Bachelor's thesis

Bachelor of Engineering

Information and Technology

XAMK

South-Eastern Finland
University of Applied Sciences

South-Eastern Finland
University of Applied Sciences

Degree title	Bachelor of Engineering
Author	Gayan Withana Gamage
Thesis title	Web Application development project for vehicle parking reservation system in student apartments
Commissioned by	Mikkelin opiskelija-asunnot Oy (MOAS)/ Mikalo Oy
Year	2025
Pages	80 pages
Supervisor	Juutilainen Matti

ABSTRACT

This thesis presents the development, design and deployment of “ParkEz” a full stack web-based vehicle parking reservation system designed for student housing provider in Mikkeli, Finland. The objective was to create a system that would offer an automated, real-time parking spot reservation solution by addressing the inefficiencies in the commissioner’s manual parking management by automating parking bookings, enabling real-time availability updates, and promoting smooth communication between administrators and tenants.

In this study agile approaches and test-driven development were utilized to provide dependable and robust functionality, while development process placed a strong emphasis on user-centred design concepts, aiming to guarantee an intuitive experience for MOAS tenants. A completely functional online application, accompanied with comprehensive documentation, and an efficient parking management procedure for commissioner are comprise by replacing manual paperwork by a digital solution. “ParkEz” increases productivity, lowers expenses, and boosts user satisfaction. Because of its flexibility, the system can be used by other housing providers who are facing similar issues.

“ParkEz”, which was developed with the MERN stack (MongoDB, Express, React, and Node.js), simplifies parking bookings, approvals, and assignments while lowering administrative effort and human error. Tenants can easily reserve parking spots via the system's user-friendly interface, and MOAS administration can manage requests, assign roles, and keep an eye on parking allocations using the administrative dashboard. Further the system's scalability, features like payment gateways, log histories and interoperability with other housing organisations can be added in the future.

CONTENTS

1 INTRODUCTION

As a university student specializing in Information Technology with a strong foundation in full-stack web development, I actively seek opportunities to apply my skills in developing systems that simplify everyday tasks. In this regard, I explored several potential solutions relevant to my surroundings, including a parking reservation application, a sauna booking system, and an online laundry reservation system. After careful consideration, I chose to develop the parking reservation application. This decision was influenced by my own experiences as a resident of s student apartment where I have personally encountered the inconvenience and inefficiency of the current parking space reservation process.

As a result, I reached out to Mikalo Oy, the property manager of the student apartments, to discuss the idea. During our conversation, I was informed that the parking reservation process was time-consuming and handled by an inefficient manual system involving multiple employees. This discussion helped me understand the complexity of the issue and the need for a more streamlined solution. After the first meeting, I created an initial sketch of the proposed system and presented it during a second meeting. Following that discussion, it was agreed that I would proceed with developing the system. Given the real-world relevance, level of detail, usability, and potential for future development, I decided to select this development project as the topic of my thesis.

1.1 Background

The rapid advancement of digital technologies has profoundly transformed many aspects of modern life, including transportation, logistics, and urban infrastructure. In recent years, smart systems have emerged as an essential component of urban development, enabling more efficient use of resources, improving user experience, and supporting data-driven decision-making. Thesis technologies leverage automation, online connectivity, and data analytics to optimize traditionally manual processes, thereby reducing operational costs and enhancing service quality across various sectors.

One area that has seen significant innovation through smart technologies is parking management. Traditional parking systems, often based on manual processes such as paper permits, physical checks, and cash payments, have become increasingly inadequate in the face of rising urban populations, growing car ownership, and the demand for more convenient and efficient services. As a result, smart parking systems have been developed to address these challenges. Smart parking systems typically integrate technologies such as online reservations, automated approvals, real-time availability monitoring, mobile access, and data analytics. These features not only improve the user experience by offering convenience but also benefit administrators by automating repetitive tasks, optimizing space utilization, and providing valuable insight for future planning.

From a technological standpoint, the development of smart parking systems requires the integration of several key components, including a robust front-end user interface, a scalable and efficient back-end server, and a flexible database to store and manage data. For the purposes of this study as essential to choose a suitable technology stack that would ensure scalability, ease of development, and efficient performance. Several potential technology stacks were considered, including LAMP (Linux, Apache, MySQL, PHP), Django with PostgreSQL, and MERN (MongoDB, Express.js, React.js, Node.js). Each of these stacks offers distinct advantages and limitations, and it was important to assess them thoroughly before making a final selection. (MERN Stack vs Django 2024.)

The LAMP stack is one of the most widely used and time-tested combinations in web development (Jansen 2014.) It consists of Linux as the operating system, Apache as the web server, MySQL as the relational database, and PHP as the server-side scripting language. LAMP offers a mature and stable environment, with abundant documentation and community support. It performs well for small- to medium-sized applications and benefits from wide hosting availability. However, PHP is generally less suited for developing highly interactive, single-page applications (SPAs) compared to modern JavaScript frameworks. Furthermore, the integration between front-end and back-end in LAMP is not as seamless, often requiring additional effort to achieve the level of interactivity expected in today’s web applications. Vertical scaling with LAMP is usually straightforward, but horizontal scaling can pose challenges. (Jansen 2014.)

Django with PostgreSQL presents a modern alternative, leveraging Python’s readability and Django’s “batteries-included” philosophy. Django provides a wide array of built-in features, including an object-relational mapper (ORM), an admin panel, and a robust authentication system. Combined with PostgreSQL, a powerful and standards-compliant database, this stack offers high reliability and advanced data handling features, such as JSON fields and full-text search. While Django excels at rapid development and clean architecture, building highly interactive SPAs typically requires integrating a separate JavaScript front-end framework like React or Vue.js. This adds additional complexity and can increase development time if the team is not equally experienced on both sides (Holovaty & Kaplan-Moss 2009).

The MERN stack is a full JavaScript-based stack that includes MongoDB as a NoSQL database, Express.js as the back-end framework, React.js for the front-end, and Node.js as the server runtime. One of the MERN stack’s key strengths is the use of a single programming language JavaScript across the entire stack, which simplifies development, enhances integration, and reduces context switching for developers. React’s component-based architecture makes it ideal for creating dynamic, responsive user interfaces, which is crucial for features like real-time parking availability in ParkEz. MongoDB’s flexible, document-oriented schema allows for quick adaptation to evolving data requirements, making it especially suitable for systems that need to iterate rapidly. Node.js, with its non-blocking, event-driven model, provides an efficient server-side environment that can handle large numbers of concurrent connections *(Kumar, 2019).* However, developers must be cautious with MongoDB’s flexible schema to avoid potential data inconsistencies.

After evaluating these options, the MERN stack was selected for the development of the “ParkEz” system. This choice was based on several factors such as the fact that MERN stack is JavaScript based on front-end to back-end, allowing for faster development and better integration between components. React enables the creation of dynamic and responsive user interfaces, while Node.js and Express provide a lightweight and efficient server-side environment. MongoDB, as a NoSQL database, offers flexibility in handling data structures and is well-suited for the evolving requirements of a web-based application like ParkEz.

Within this context, the focus of this thesis is on the design and development of ParkEz, a smart parking system implemented for MOAS's student apartment services. The ParkEz system aims to meet both user needs and administrative challenges by following best practices in digital transformation and system development. By automating manual processes, improving access to parking resources, and integrating data analytics, ParkEz is expected to deliver higher efficiency, cost savings, and user satisfaction. The thesis will explain the technological choices made—particularly the selection of the MERN stack based on a comparative analysis of different technology stacks and will provide a theoretical foundation for the practical implementation of the system.

1.2 Research Question

This thesis's main research question is: How can a web-based system at MOAS efficiently automate the parking reservation process, cut down on administrative work, and improve the user experience for administrators and tenants? The “ParkEz” system's development and assessment are guided by this query.

2 MOAS/ MIKALO PARKING CHALLENGES AND REQUIREMENTS

At the time of developing this solution, MOAS Oy (Mikkeli Student Housing Ltd) was the primary housing provider for both domestic and international students in the city of Mikkeli. However, on January 1, 2025, MOAS Oy and Mikalo Oy underwent a consolidation process which resulted in Mikalo Oy becoming the sole provider of student housing in the city This transition centralized housing operations under Mikalo Oy, including the management of student apartment facilities and related services such as parking reservations. (Mikalo Oy 2024.)

The Mikalo Oy as a rental housing firm that provides apartments for rent in the Mikkeli region 2024 10% of Mikkeli's living this apartment.

2.1 Current Parking Management Issues

The current parking management process at Mikalo Oy that relies heavily on manual operations, which poses several challenges. Firt of all tenants must submit parking requests by the way of paper forms or emails, which leads to long processing and times. Since there is no centralized system to track the availability of parking spots in real time, confusion and overbooking frequently emerge. As a repercussion of that staff spend considerable amount of time managing parking allocations, approvals and tenant communications which could be streamlined through automation. In addition, manual data entry and record-keeping increase the risk of errors in reservations, leading to potential conflicts among tenants. Furthermore, handling parking-related issues and updates through fragmented communication channels reduces efficiency and responsiveness.

2.2 Requirements for the Parking Reservation System

In order to solve the current parking processing issues the new parking reservation system should enable tenants to easily book parking spots online, reducing the need for manual intervention, and provide up-to-date information on parking space availability to prevent double bookings and improve transparency.

In addition, the administrators should be equipped with tools to efficiently manage parking requests, approve or decline reservations, and monitor overall usage. Moreover, it must be ensured that system is intuitive and accessible for both tenants and administrators, enhancing user experience.

As for security the application encompasses measures to manage different access levels for administrators and tenants and the system design aims to support future expansions, such as integrating payment gateways, generating detailed usage reports, and adapting to other housing providers. In sum, the application should facilitate smooth interact between tenants and administrators regarding parking-related issues.

3 SYSTEM ARCHITECTURE AND DESIGN

3.1 MERN Stack Overview

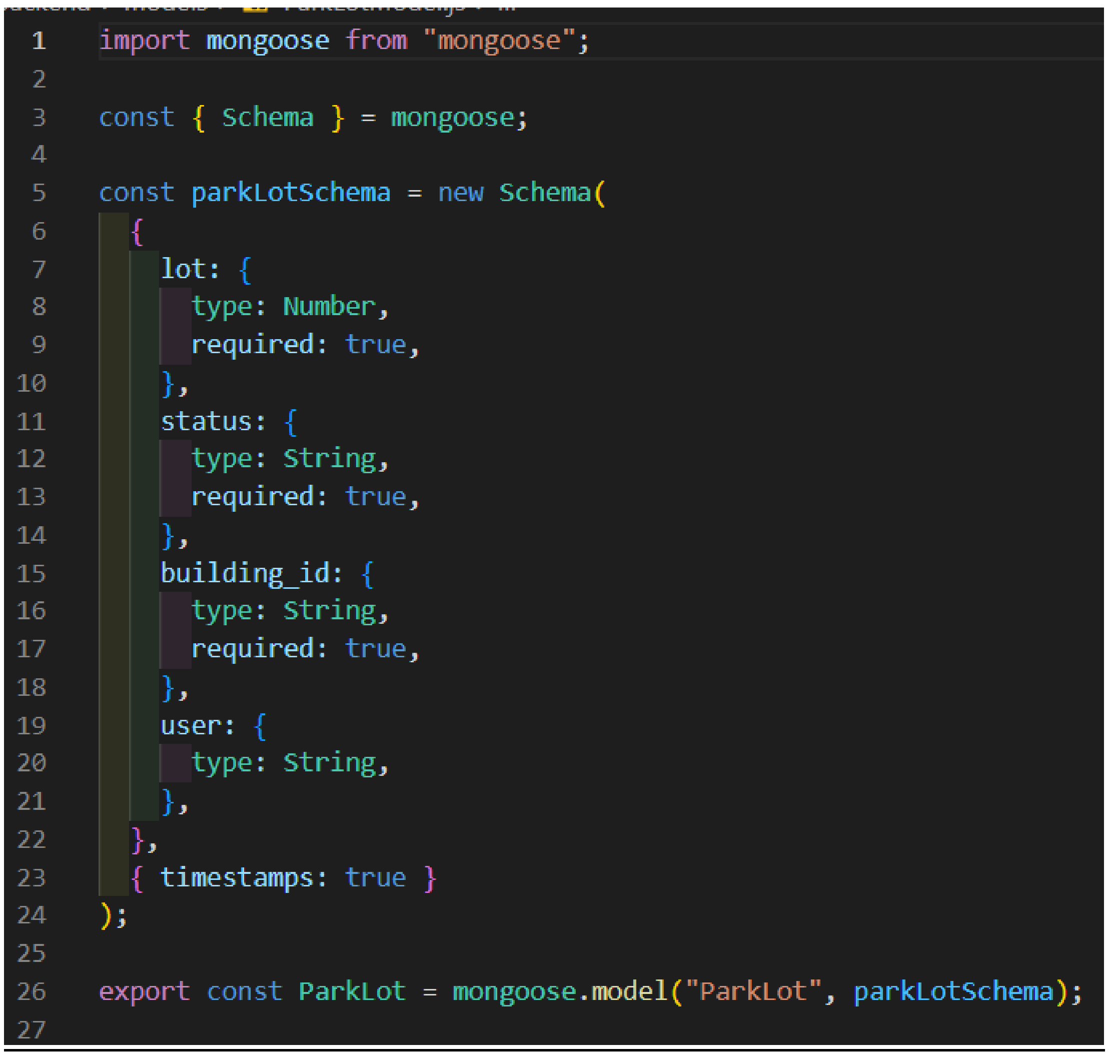
Due to its support for rapid development, easy scalability, and seamless API integration, the MERN tech stack was chosen for this study.

In the following section, MERN technologies and their role in modern web development explained.

MongoDB

MongoDB is a document-oriented NoSQL database designed for handle and store data that is dynamic, hierarchical, and flexible. MongoDB is very suitable for applications fetching dynamic data structures because, in contrast to conventional relational databases, it stores data in the JSON-like BSON (Binary JSON) format. (Mongoose. 2024.)

The MongoDB was selected for ParkEz because it has a flexible schema which facilitates future upgrades by enabling the storage of user information, parking spaces, and reservation data without a set structure. This allows for horizontal scaling and the system can effectively manage a number of users and reservations. In addition, it supports fast read/write operations designed to process large numbers of parking requests. Consequently, it reduces the number of database searches by allowing the storage of related data together (Mongoose. 2024). Figure 1 illustrates the layout of the parking lot schema, providing a visual representation of its structure and organization.



```
import mongoose from "mongoose";

const { Schema } = mongoose;

const parkLotSchema = new Schema(
  {
    lot: {
      type: Number,
      required: true,
    },
    status: {
      type: String,
      required: true,
    },
    building_id: {
      type: String,
      required: true,
    },
    user: {
      type: String,
    },
  },
  { timestamps: true }
);

export const ParkLot = mongoose.model("ParkLot", parkLotSchema);
```

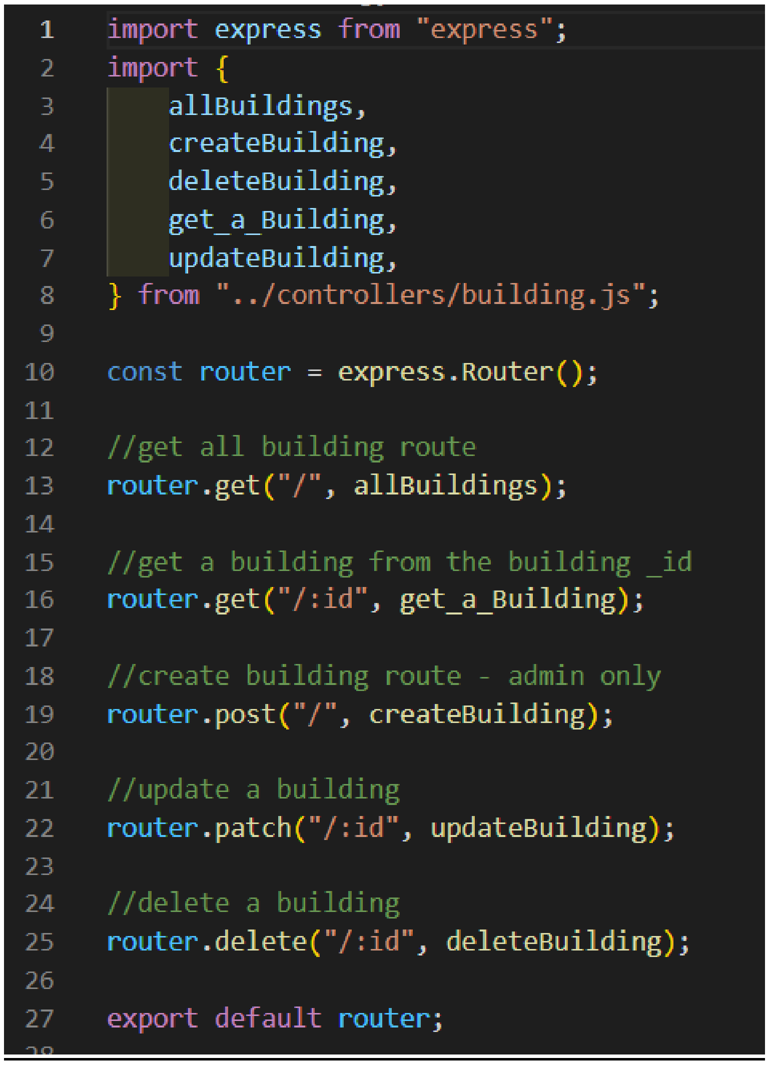
Figure 1. Parking lot schema

Express.js

Express.js is a simple web framework for Node.js that facilitates the creation of back-end functionality and RESTful APIs. By processing HTTP requests, controlling routes, and guaranteeing smooth data transmission, it serves as a link between the database (MongoDB) and front-end (React) (Express.js 2024).

Express.js is used in ParkEz primarily because it permits the definition of API endpoints for managing admin approvals, user authentication, and parking bookings. Managing this supports the use of middleware which allows to process requests, implement security features such as authentication, and perform errors.

The asynchronous design enables a simultaneous processing of several parking requests, and shorter times are guaranteed owing to the use of non-blocking I/O. *(Express.js, 2024)* Figure 2 presents a code snippet showcasing examples of all the building-related API endpoints and routes.



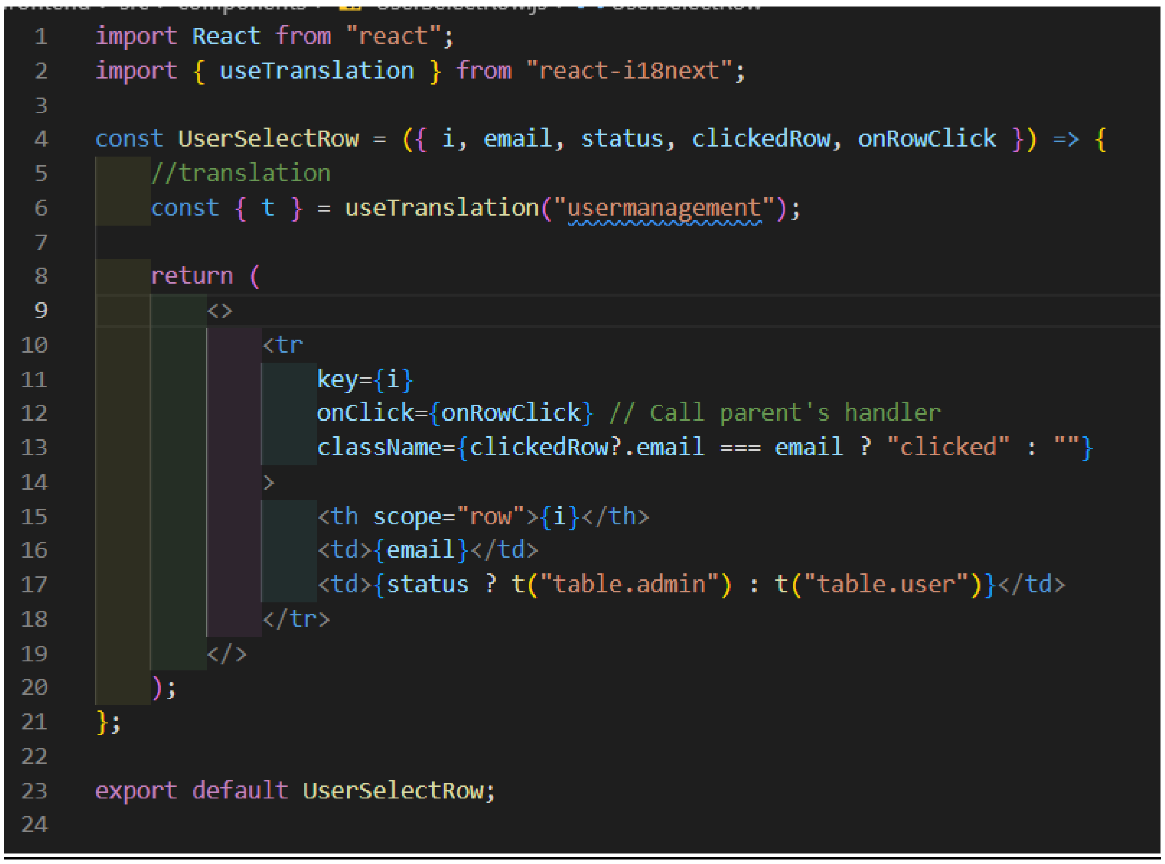
```
import express from "express";
import {
    allBuildings,
    createBuilding,
    deleteBuilding,
    get_a_Building,
    updateBuilding,
} from "../controllers/building.js";

const router = express.Router();

//get all building route
router.get("/", allBuildings);

//get a building from the building _id
router.get("/:id", get_a_Building);

//create building route - admin only
router.post("/", createBuilding);

//update a building
router.patch("/:id", updateBuilding);

//delete a building
router.delete("/:id", deleteBuilding);

export default router;
```

Figure 2. Building related API endpoints / routes

React

A JavaScript library called React is used to create responsive and dynamic user interfaces. Because of its component-driven architecture, it is extremely suitable for developing a user-friendly dashboard for parking reservations and serving the client-side component builder.

React follows a component-based architecture, allowing features such as the admin approval panel, reservation form, and parking availability list to be built as reusable components, which enhances maintainability. Furthermore, the use of React's built-in and custom hooks helps manage application context and state effectively. As a result, components render efficiently, enabling smoother interaction between tenants and administrators. Figure 3 illustrates a code snippet of the user table row component, which is rendered dynamically.

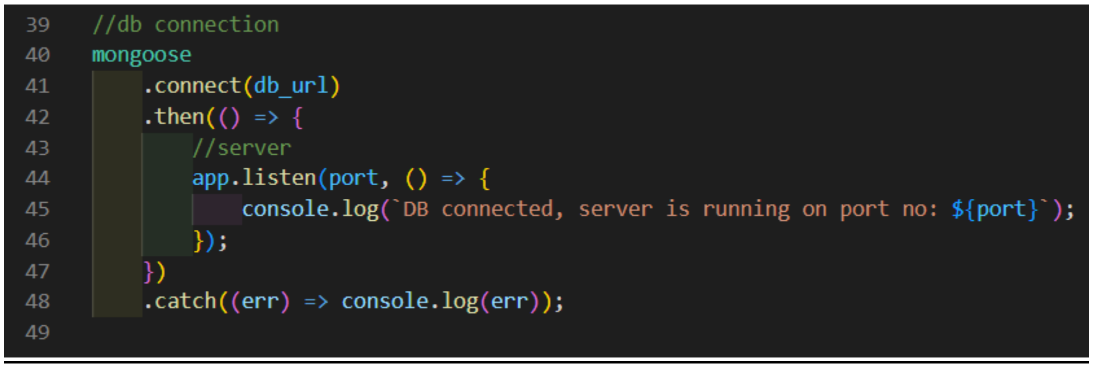


```jsx
import React from "react";
import { useTranslation } from "react-i18next";

const UserSelectRow = ({ i, email, status, clickedRow, onRowClick }) => {
  //translation
  const { t } = useTranslation("usermanagement");

  return (
    <>
      <tr
        key={i}
        onClick={onRowClick} // Call parent's handler
        className={clickedRow?.email === email ? "clicked" : ""}
      >
        <th scope="row">{i}</th>
        <td>{email}</td>
        <td>{status ? t("table.admin") : t("table.user")}</td>
      </tr>
    </>
  );
};

export default UserSelectRow;

```

Figure 3. Users table row component

Node.js

Node.js is an event-driven, high-performance JavaScript runtime for server-side programming (Kumar, S 2019). It is appropriate for real-time parking management since it enables an efficient handling of asynchronous requests. The main purpose of using node is scalability which enables the management of several parking requests at once, expediting response times. In addition, for both the front end and the back end, usage of same programming language reduces complexity reduces, and non-blocking architecture asynchronously manages API calls and database queries to guarantee a seamless user experience. Figure 4 presents a code snippet illustrating the backend server setup that runs after a successful connection to the database.



```
//db connection
mongoose
    .connect(db_url)
    .then(() => {
        //server
        app.listen(port, () => {
            console.log(`DB connected, server is running on port no: ${port}`);
        });
    })
    .catch((err) => console.log(err));

```

Figure 4. Backend server setup

3.2 System Architecture

The three-tier architecture of the ParkEz system facilitates maintainability, scalability, and modularity. Because each layer has a specific purpose, the system can effectively manage administrative tasks, user authentication, and real-time parking management. Figure 5 illustrates the three-tier architecture, depicting the client, server, and database components and their interactions.

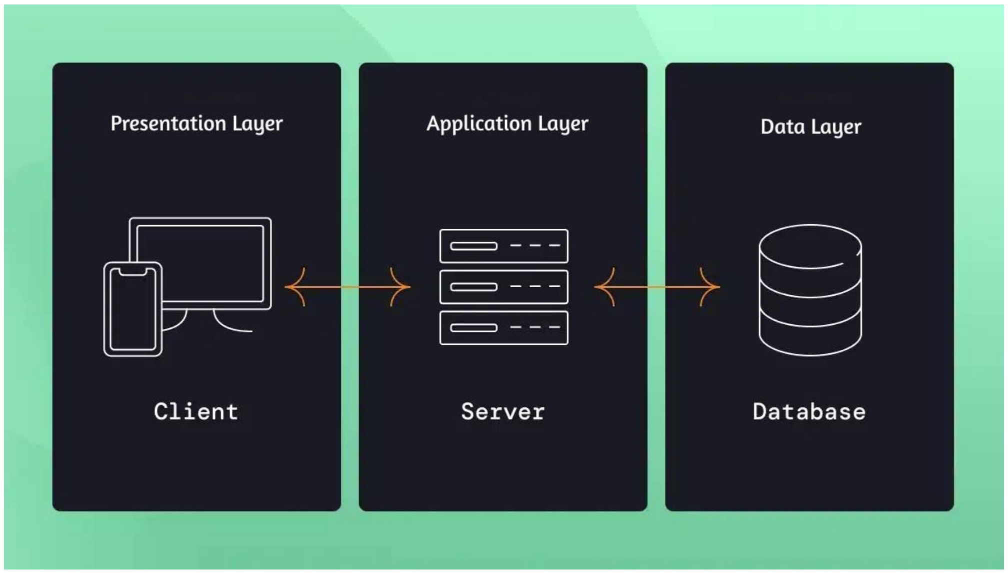


Figure 5. Three-tier architecture

Presentation Layer (Front-End - React.js)

React.js is used in the front-end development, providing administrators and tenants with an interactive and intuitive interface that allows them to check status updates, make reservations, and verify parking availability. The admin dashboard gives administrators the ability to create buildings and relevant parking spaces, monitor requests, and accept or reject bookings. The parking status is dynamically updated in real-time without requiring a page reload by using context API polling (react context hooks). The responsive design which can be accessed across a range of devices including desktop, tablet, and mobile screens. In addition, component-based architecture provides a reusable React component, improving maintainability (Stelitano, S 2025). As shown in Figure 6, the diagram depicts the UI flow, highlighting the sequence of user interactions within the application.

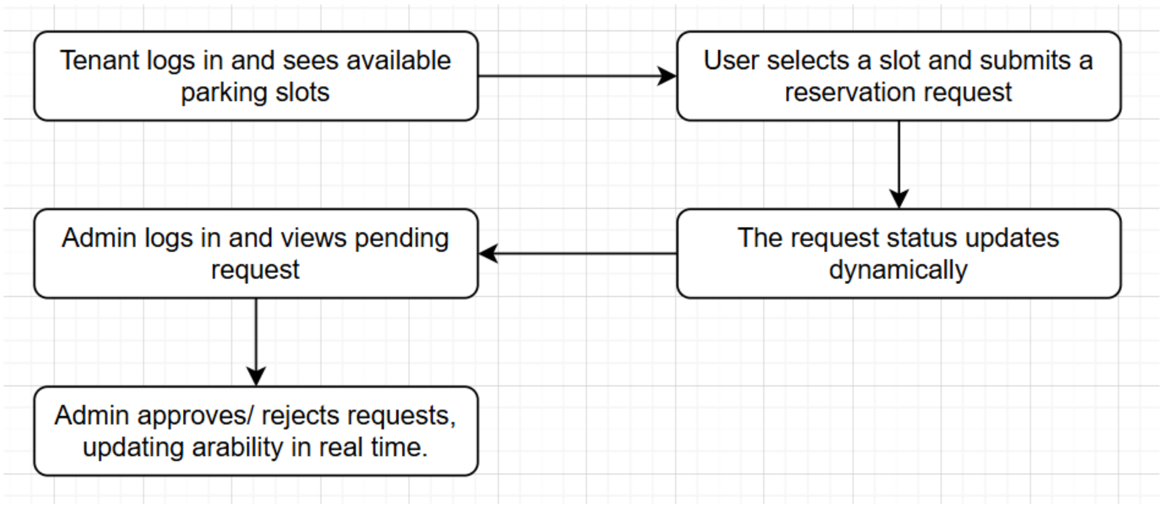

Tenant logs in and sees available parking slots
User selects a slot and submits a reservation request
The request status updates dynamically
Admin logs in and views pending request
Admin approves/ rejects requests, updating arability in real time.

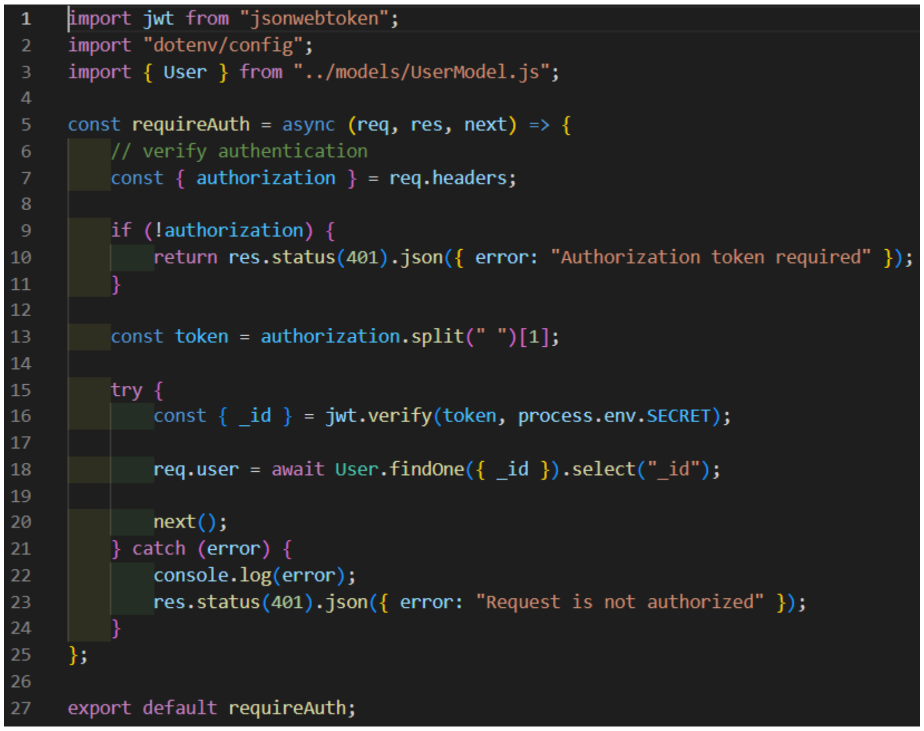
Figure 6. UI Flow

In this study React.js was used front-end UI, Context API was employed state management, bootstrap was utilized for styling and UI components, and fetch API was used to handle the API calls in frontend and backend communication.

Application Layer (Back-End - Node.js & Express.js)

The database and the front-end user interface are connected by the back end. This promotes safe and effective real-time API communication. The server the APIs were developed with Node.js and Express.js. The RESTful API endpoints have been designed to manage parking spaces, reservations, and user CRUD (Create, Read, Update, Delete) operations. In addition, the Role-Based Access Control (RBAC) distinguishes between administrators and tenants in terms of access rights.

The JSON Web Toke has been used for authentication. Once user successfully login with the right credentials, the JWT passes through the HTTP headers to verify the user via middleware implementation which manages request validation, logging, and error management for increased system stability (Kalehewatte, D 2024). Figure 7 presents a code snippet used to handle the JWT (JSON Web Token) from the front-end, illustrating its implementation in the application.

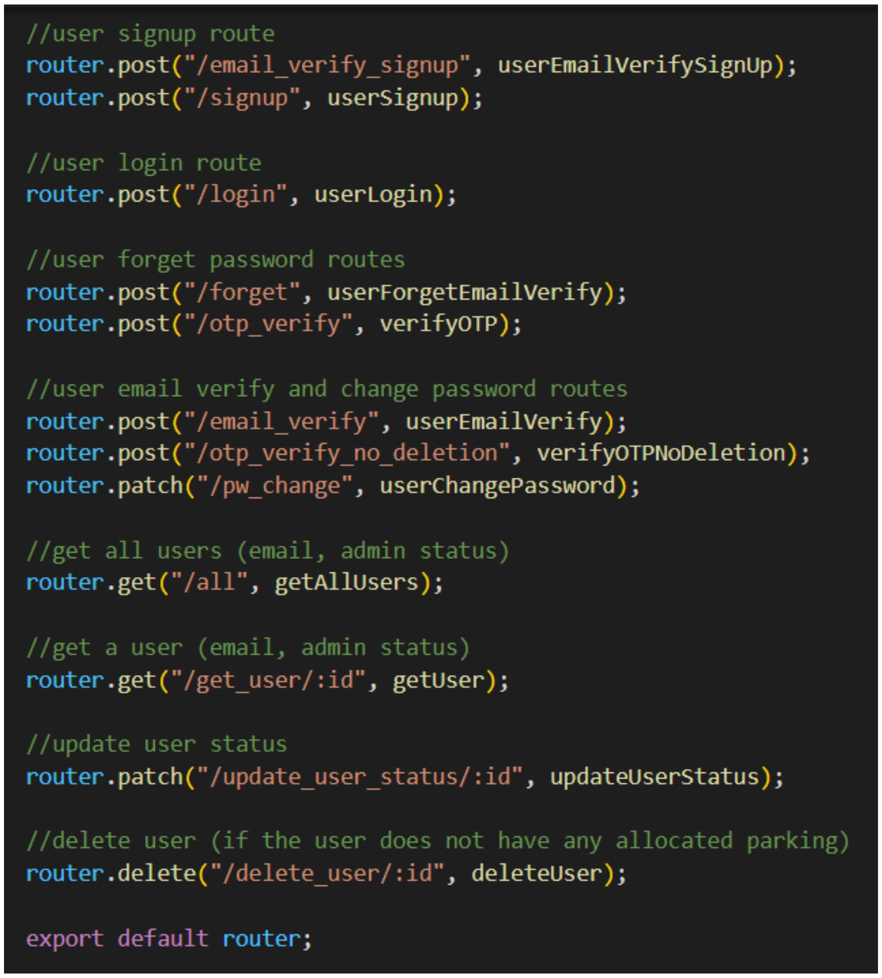


```
import jwt from "jsonwebtoken";
import "dotenv/config";
import { User } from "../models/UserModel.js";

const requireAuth = async (req, res, next) => {
    // verify authentication
    const { authorization } = req.headers;

    if (!authorization) {
        return res.status(401).json({ error: "Authorization token required" });
    }

    const token = authorization.split(" ")[1];

    try {
        const { _id } = jwt.verify(token, process.env.SECRET);

        req.user = await User.findOne({ _id }).select("_id");

        next();
    } catch (error) {
        console.log(error);
        res.status(401).json({ error: "Request is not authorized" });
    }
};

export default requireAuth;
```

Figure 7. Middleware - requireAuth.js file

As shown in Figure 7, the Bearer token which comes via headers in the front-end for verification with the backend. Once the token has been verified the front-end client has the access to the API end points.

API endpoints such as user authentication (login/signup), parking reservation requests, and admin approval are key examples that support secure access and efficient parking management. Figure 8 presents a code snippet that defines the back-end routes (API endpoints) for user sign-in and sign-up. Once the relevant endpoint is triggered with a valid token, the corresponding controller function is called.



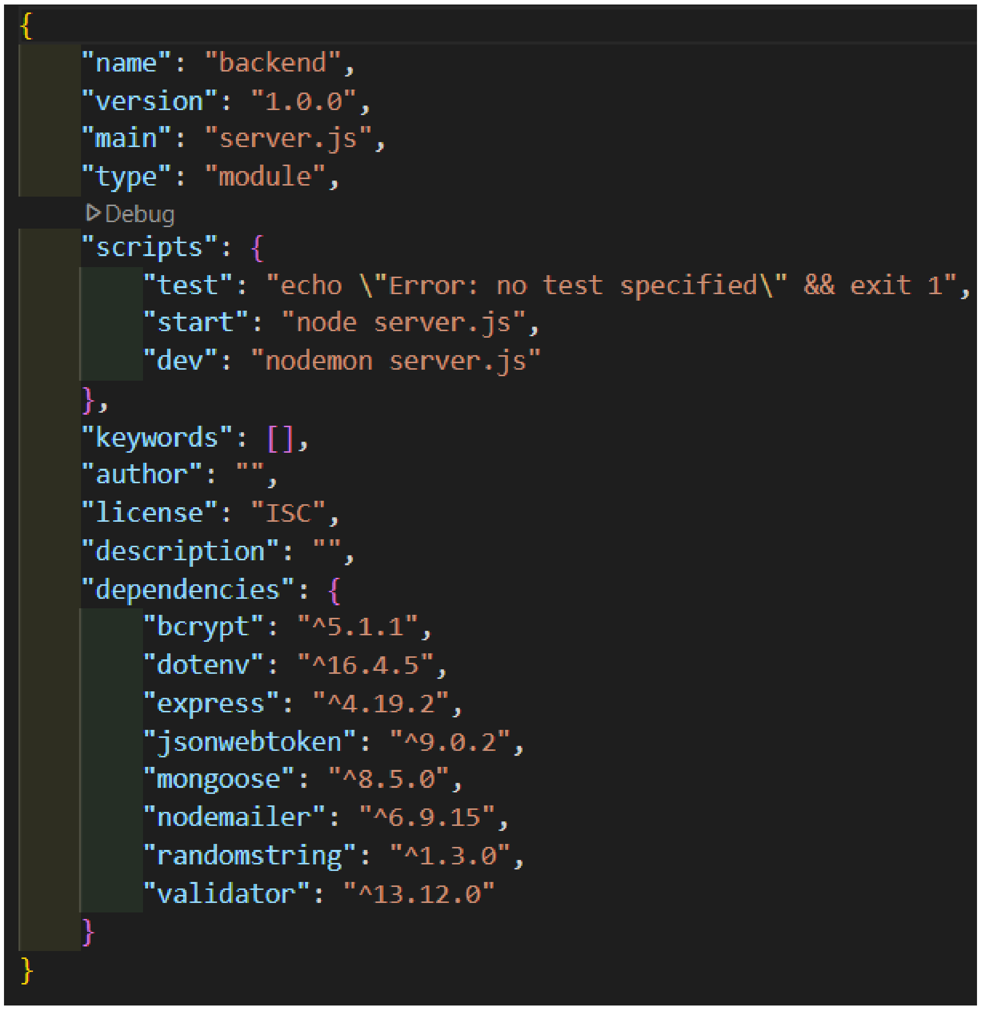
```
//user signup route
router.post("/email_verify_signup", userEmailVerifySignUp);
router.post("/signup", userSignup);

//user login route
router.post("/login", userLogin);

//user forget password routes
router.post("/forget", userForgetEmailVerify);
router.post("/otp_verify", verifyOTP);

//user email verify and change password routes
router.post("/email_verify", userEmailVerify);
router.post("/otp_verify_no_deletion", verifyOTPNoDeletion);
router.patch("/pw_change", userChangePassword);

//get all users (email, admin status)
router.get("/all", getAllUsers);

//get a user (email, admin status)
router.get("/get_user/:id", getUser);

//update user status
router.patch("/update_user_status/:id", updateUserStatus);

//delete user (if the user does not have any allocated parking)
router.delete("/delete_user/:id", deleteUser);

export default router;
```

Figure 8. User API end points

The thesis leverages a range of modern technologies and libraries to ensure secure, efficient, and scalable backend operations. These tools support tasks such as authentication, database interaction, data validation, and communication.

- Node.js – Server-side runtime
- Express.js – Web framework for handling API requests
- JWT (JSON Web Token) – Secure authentication
- Bcrypt.js – Password hashing
- Mongoose – ODM for MongoDB
- Nodemailer – Send email notifications
- Randomstring – Generate OTP for verifications
- Validator – email and password validations

Figure 9 presents the JSON file that extracts the relevant Node packages used for the backend. When the command ‘npm install’ is executed, the dependencies listed in this file are installed accordingly.



```
{
    "name": "backend",
    "version": "1.0.0",
    "main": "server.js",
    "type": "module",
    ▷Debug
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1",
        "start": "node server.js",
        "dev": "nodemon server.js"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "description": "",
    "dependencies": {
        "bcrypt": "^5.1.1",
        "dotenv": "^16.4.5",
        "express": "^4.19.2",
        "jsonwebtoken": "^9.0.2",
        "mongoose": "^8.5.0",
        "nodemailer": "^6.9.15",
        "randomstring": "^1.3.0",
        "validator": "^13.12.0"
    }
}
```

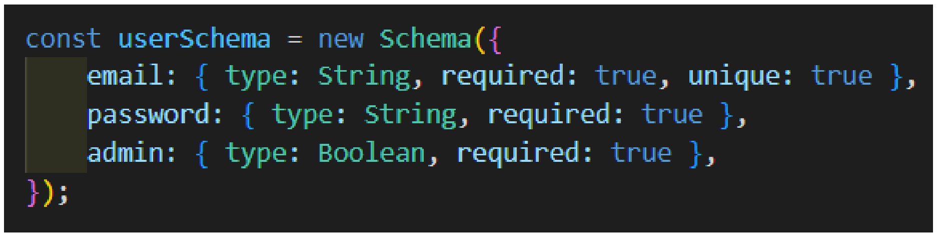
Figure 9. pakage.json file

Data Layer (Database - MongoDB)

The centralised database has parking spot information and user data. The features of this layer allow for flexible schema creation by storing data as BSON documents that resemble JSON. In addition, schema optimization utilizes indexed collections to expedite query execution, which minimises the need for multiple searches by storing related data (such as reservation history) within nested objects. In addition, scalability allows for horizontal scaling, enhancing performance even as parking requests and user data increase. (Mongoose 2024).

The application uses several MongoDB collections to store and manage data. The Users collection holds information about both tenants and admins. The Parking Slots collection stores details of available parking spaces. The

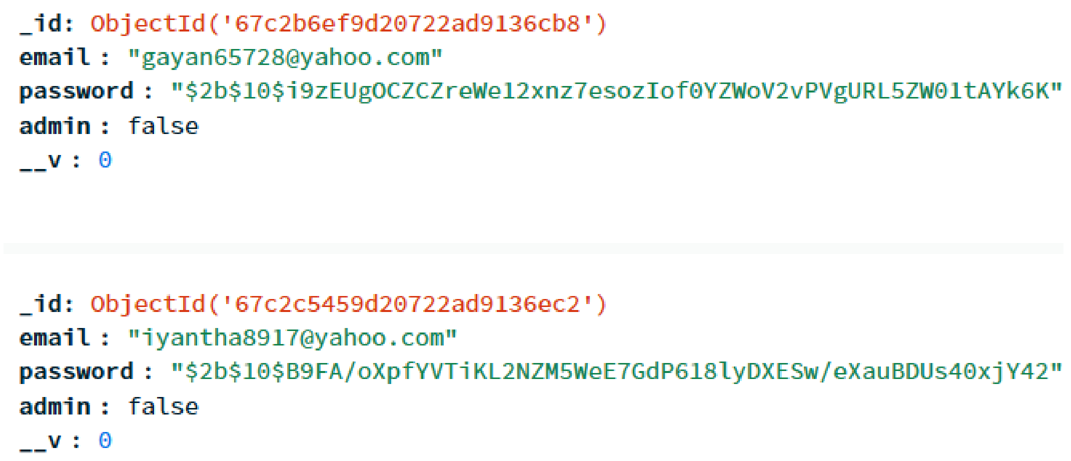
Reservations collection keeps records of all parking slot bookings made by users. Figure 10 presents a code snippet that defines the user schema, outlining the structure and fields used to store user data.



```
const userSchema = new Schema({
    email: { type: String, required: true, unique: true },
    password: { type: String, required: true },
    admin: { type: Boolean, required: true },
});
```

Figure 10. User schema

Figure 11 illustrates how user data is stored in the database, providing a visual representation of the data structure and storage process.



```
_id: ObjectId('67c2b6ef9d20722ad9136cb8')
email : "gayan65728@yahoo.com"
password : "$2b$10$i9zEUgOCZCZreWe12xnz7esozIof0YZWoV2vPVgURL5ZW01tAYk6K"
admin : false
__v : 0
```

```
_id: ObjectId('67c2c5459d20722ad9136ec2')
email : "iyantha8917@yahoo.com"
password : "$2b$10$B9FA/oXpfYVTiKL2NZM5WeE7GdP618lyDXESw/eXauBDUs40xjY42"
admin : false
__v : 0
```

Figure 11. User data in the database

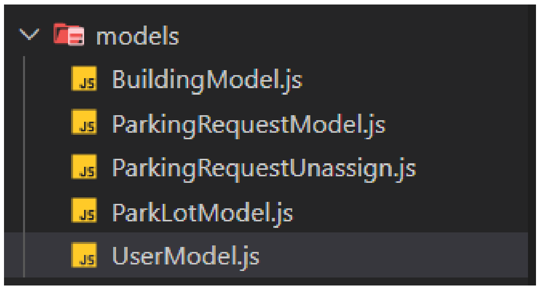
The thesis utilizes MongoDB to store and manage application data. In order to interact with MongoDB more effectively, Mongoose is used as an Object Data Modelling (ODM) library, allowing the definition of clear data schemas and relationships. Additionally, features such as indexes and aggregation pipelines are implemented to optimize query performance and enhance complex data processing, such as filtering, grouping, and summarizing. These technologies together ensure reliable data storage and fast access in a structured yet adaptable format.

3.3 Database Design and Management

MongoDB is used by the ParkEz parking reservation system to effectively store and retrieve parking-related data. The database architecture supports role-based access for administrators and tenants while maintaining data scalability, speed, and integrity.

Database Schema Design

MongoDB divides data into collections, each of which contains pertinent properties and BSON documents that resemble JSON (Mongoose 2024).
In this study schemas are implemented for user, building, parking lot, parking request, and parking unassigned request. Figure 12 illustrates the Models directory, which contains five schemas, each defining the structure and behaviour of different data entities in the application.

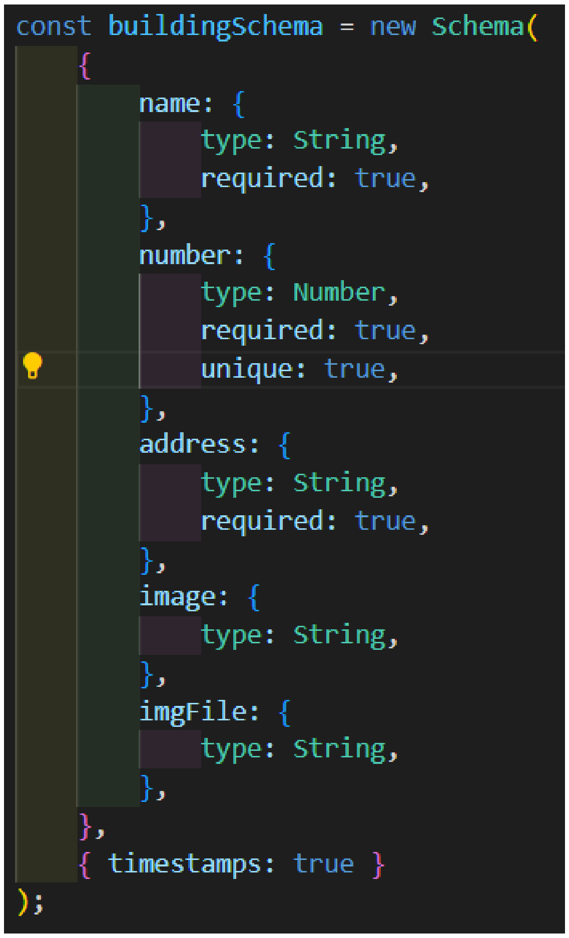


```
models
    BuildingModel.js
    ParkingRequestModel.js
    ParkingRequestUnassign.js
    ParkLotModel.js
    UserModel.js
```

Figure 12. Models directory - five schemas

BuildingModel.js

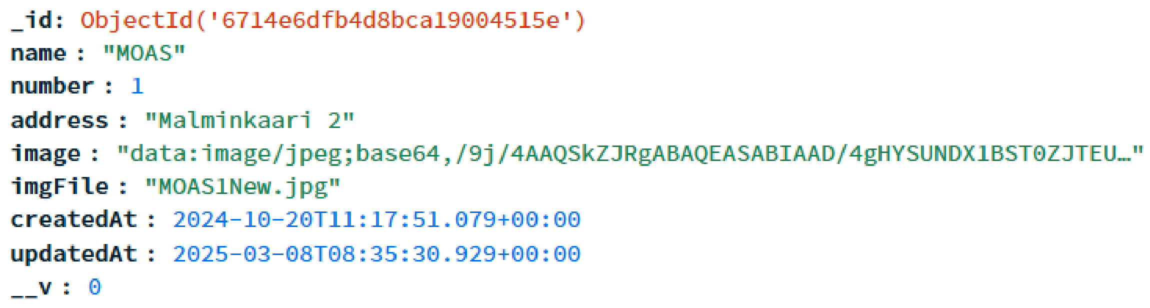
BuildingModel.js file contains the building schema including building name, building number, building address, building image file, building image file name and the date and time the data was modified. Figure 13 presents a code snippet that defines the building schema, outlining the structure and fields used to store building-related data.



```
const buildingSchema = new Schema(
    {
        name: {
            type: String,
            required: true,
        },
        number: {
            type: Number,
            required: true,
            unique: true,
        },
        address: {
            type: String,
            required: true,
        },
        image: {
            type: String,
        },
        imgFile: {
            type: String,
        },
    },
    { timestamps: true }
);
```

Figure 13. BuildingModel.js - Building schema

Figure 14 illustrates how data is saved in the database according to the above structure, detailing the format and organization of the stored building-related information.

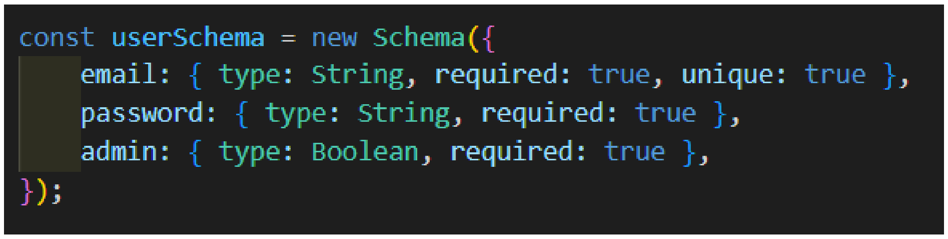


```
_id: ObjectId('6714e6dfb4d8bca19004515e')
name : "MOAS"
number : 1
address : "Malminkaari 2"
image : "data:image/jpeg;base64,/9j/4AAQSkZJRgABAQEASABIAAD/4gHYSUNDX1BST0ZJTEU…"
imgFile : "MOAS1New.jpg"
createdAt : 2024-10-20T11:17:51.079+00:00
updatedAt : 2025-03-08T08:35:30.929+00:00
__v : 0
```

Figure 14. Collection according to the building schema

UserModel.js

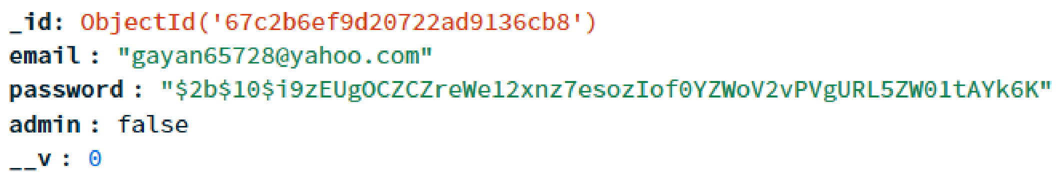
UserModel.js file contains the user schema including user email, user password and admin status. Figure 15 presents a code snippet that defines the user schema, outlining the structure and fields used to store user-related data.



```
const userSchema = new Schema({
    email: { type: String, required: true, unique: true },
    password: { type: String, required: true },
    admin: { type: Boolean, required: true },
});
```

Figure 15. UserModel.js - User schema

As per the above structure, the data is stored in the database using clearly defined collections. Figure 16 illustrates the Users collection, showcasing how user data, including tenant and admin details, is organized and stored. Similarly, Figure 17 presents the structure of the Parking Slots and Reservations collections, highlighting how booking and slot information is efficiently managed within the system.



```
_id: ObjectId('67c2b6ef9d20722ad9136cb8')
email : "gayan65728@yahoo.com"
password : "$2b$10$i9zEUgOCZCZreWe12xnz7esozIof0YZWoV2vPVgURL5ZW01tAYk6K"
admin : false
__v : 0
```

Figure 16. User collection

ParkLotModel.js

ParkLotModel.js file contains the parking space schema including the parking lot number, parking lot status (available, reserved, pending reservation, under maintenance), building id (links with the relevant building), user (which user belongs to this parking space), and time of creation and modification. Figure 16 presents a code snippet that defines the parking space schema, outlining the structure and fields used to store parking space-related data.

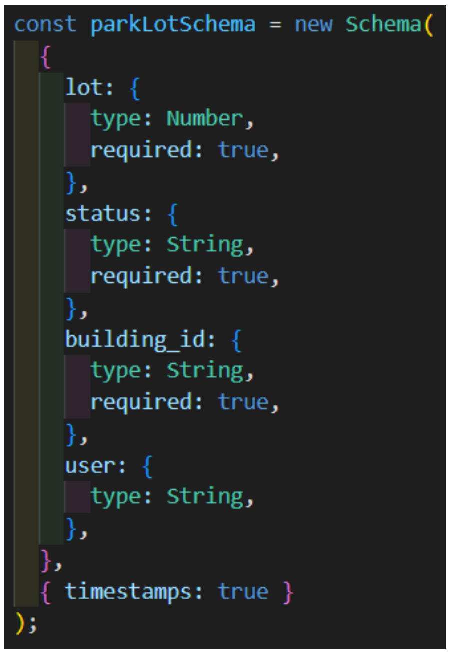

```
const parkLotSchema = new Schema(
  {
    lot: {
      type: Number,
      required: true,
    },
    status: {
      type: String,
      required: true,
    },
    building_id: {
      type: String,
      required: true,
    },
    user: {
      type: String,
    },
  },
  { timestamps: true }
);
```

Figure 17. ParkLotModel.js - Parking space schema

Figure 18 illustrates how the data is stored in the database according to the above structure, showing how parking space data is organized and saved.

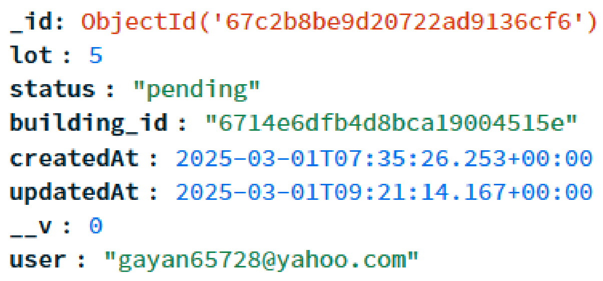
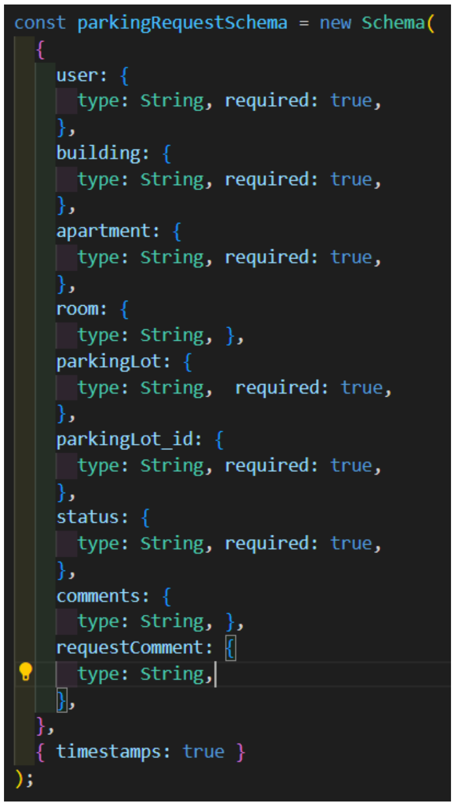

```
_id: ObjectId('67c2b8be9d20722ad9136cf6')
lot : 5
status : "pending"
building_id : "6714e6dfb4d8bca19004515e"
createdAt : 2025-03-01T07:35:26.253+00:00
updatedAt : 2025-03-01T09:21:14.167+00:00
__v : 0
user : "gayan65728@yahoo.com"
```

Figure 18 Parking space collection

ParkingRequestModel.js

ParkingRequestModel.js file contains the parking request schema which contains the user who made the request, building which relets to the parking request, apartment, room, parking lot id, parking lot number, parking lot status, comments, request comment and request creation and modification date.

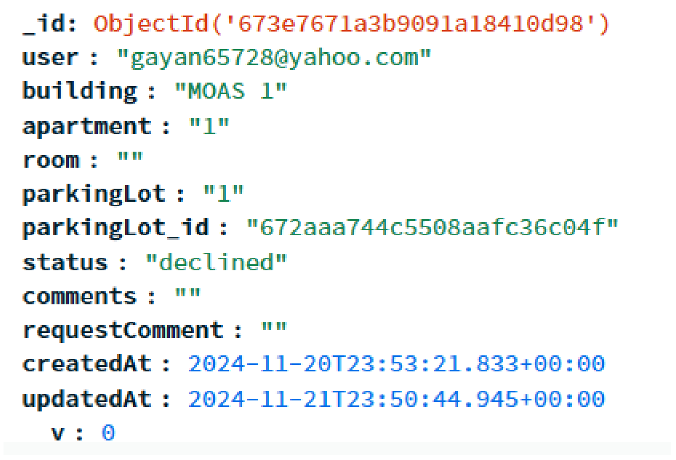
Figure 19 presents a code snippet that defines the parking request schema, outlining the structure and fields used to store parking request data.



```
const parkingRequestSchema = new Schema(
  {
    user: {
      type: String, required: true,
    },
    building: {
      type: String, required: true,
    },
    apartment: {
      type: String, required: true,
    },
    room: {
      type: String, },
    parkingLot: {
      type: String,  required: true,
    },
    parkingLot_id: {
      type: String, required: true,
    },
    status: {
      type: String, required: true,
    },
    comments: {
      type: String, },
    requestComment: {
      type: String,
    },
  },
  { timestamps: true }
);
```

Figure 19. Parking request schema

Figure 20 illustrates how the data is stored in the database according to the above structure, showing how parking request data is organized and saved.

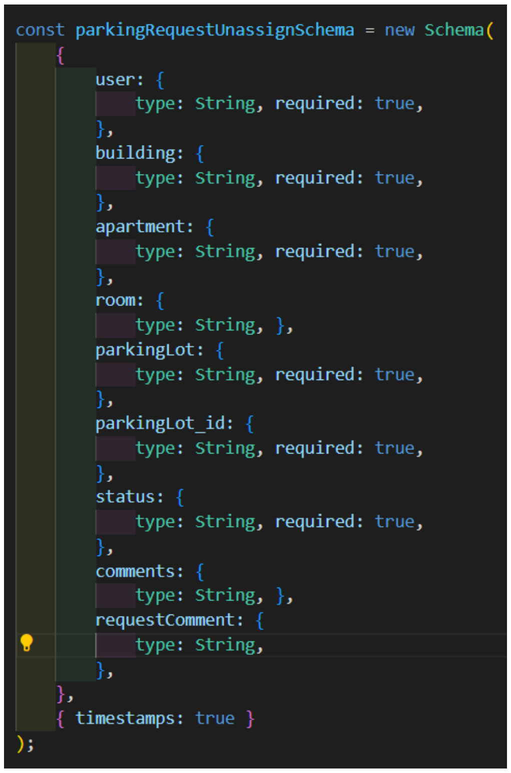


```
_id: ObjectId('673e7671a3b9091a18410d98')
user : "gayan65728@yahoo.com"
building : "MOAS 1"
apartment : "1"
room : ""
parkingLot : "1"
parkingLot_id : "672aaa744c5508aafc36c04f"
status : "declined"
comments : ""
requestComment : ""
createdAt : 2024-11-20T23:53:21.833+00:00
updatedAt : 2024-11-21T23:50:44.945+00:00
  v : 0
```

Figure 20. Parking request collection

ParkingRequestUnassignModel.js

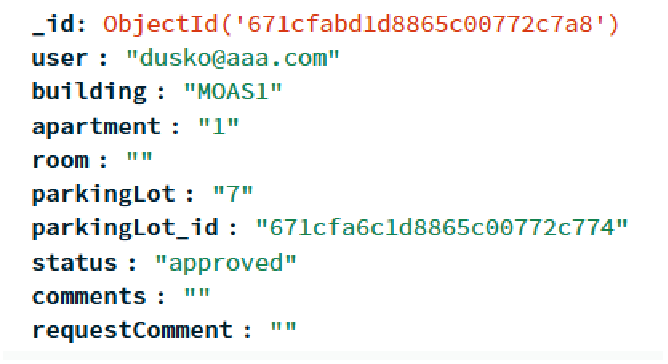
ParkingRequestUnassignModel.js file contains the parking unassign request schema including the user who made the request to unassign the current parking, building relevant to the parking unassign request, user's apartment, user's room, parking lot id, parking lot number, parking lot status, comments, request comment, creation and modification date. Figure 21 presents a code snippet that defines the parking request schema, outlining the structure and fields used to store parking request data in the database.



```
const parkingRequestUnassignSchema = new Schema(
    {
        user: {
            type: String, required: true,
        },
        building: {
            type: String, required: true,
        },
        apartment: {
            type: String, required: true,
        },
        room: {
            type: String, },
        parkingLot: {
            type: String, required: true,
        },
        parkingLot_id: {
            type: String, required: true,
        },
        status: {
            type: String, required: true,
        },
        comments: {
            type: String, },
        requestComment: {
            type: String,
        },
    },
    { timestamps: true }
);
```

Figure 21. Parking unassign request schema

Figure 22 illustrates how the data is stored in the database, showing how parking request data is organized and stored.



```
_id: ObjectId('671cfabd1d8865c00772c7a8')
user : "dusko@aaa.com"
building : "MOAS1"
apartment : "1"
room : ""
parkingLot : "7"
parkingLot_id : "671cfa6c1d8865c00772c774"
status : "approved"
comments : ""
requestComment : ""
```

Figure 22. Parking unassign request collection

Database Management

There are various database management techniques used to promote effectiveness, security, and scalability. The code snippets illustrated in Figure 22 contain techniques such as indexing and query optimization that achieve faster search operation. Unique constraints are used with the objective of preventing duplicate data. Data consistency and validation are maintained through constraints that support backend user data validation. Data security and access control follow industry norms for example, the dotenv package is used to secure sensitive data by storing it as environment variables in a .env file. The passwords are being hashed and bcrypt package has been used to hash, match, and verify the passwords.

4 DEVELOPMENT PROCESS

The agile development style used by the ParkEz parking reservation system aims to guarantee an organised and iterative development process. In order to provide smooth user experience, the process consists of planning, database implementation, front-end and back-end development, and UI/UX design phases.

4.1 Project Planning and Timeline

Table 1 illustrates the project planning and timeline, outlining key milestones and their respective deadlines.

Phase	**Tasks**	**Duration**
Sprint 1 Requirement Analysis, Planning	Identified system requirements. Wireframes structured. Database schema design. REST API structure.	6 weeks
Sprint 2: Front-end Development (React.js)	Build UI components. Authentication pages. Admin and tenant dashboards. Parking request, approving forms.	9 weeks
Sprint 3: Back-end Development (Node.js, Express.js)	Develop API endpoints for authentication, parking spot management, reservations, Approvals. Authorization and secure API endpoints.	4 weeks
Sprint 4: Database Integration (MongoDB)	Created MongoDB collections, indexing, validation, and backup strategies.	4 weeks
Sprint 5:		

UI/UX Refinements & Real-time Features	Improve responsiveness, Enhance user experience via custom css.	2 weeks
Sprint 6: Testing & Deployment	Conduct unit testing, security testing, and implemented the CI/ CD pipeline.	3 weeks

Table 1 Project planning and timeline

4.2 Frond-end Development (React.js)

The React.js front-end provides an interactive and responsive user interface for both tenants and admins. Apart from that the component-based architecture provides reusable components in the entire application dynamically.
As an example, the navigation bar has been rendered in both admin and tenant interface dynamically.

The admin user has more features in the Nav bar component as the Figure 23.

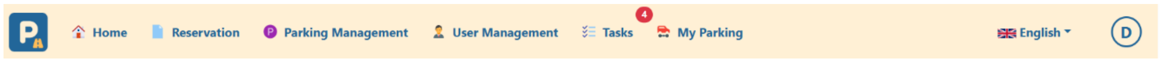


Figure 23. Admin navbar component

The tenet uses the same navigation bar with less features as the Figure 24.

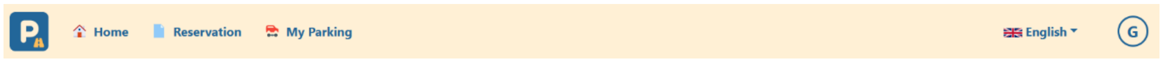
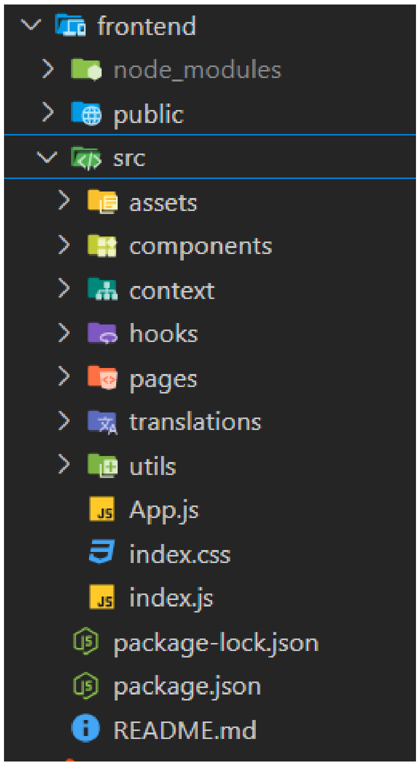


Figure 24. Tenet navbar component

According to the default react application, “node_modules”, “public” and “src” folder structure are common for the entire application. All the necessary pages and components are built inside the “src” folder. Figure 26 illustrates the front-end folder structure, showcasing the organization of files and directories within the front-end portion of the application.

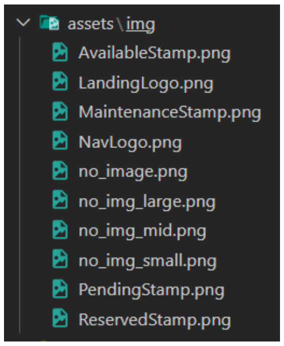


```
frontend
  node_modules
  public
  src
    assets
    components
    context
    hooks
    pages
    translations
    utils
    App.js
    index.css
    index.js
  package-lock.json
  package.json
  README.md
```

Figure 25. Front-end folder structure

Assets Directory

All the assets such as fixed images are being saved in assets directory. Figure 26 illustrates the assets directory, displaying all the images used throughout the application.

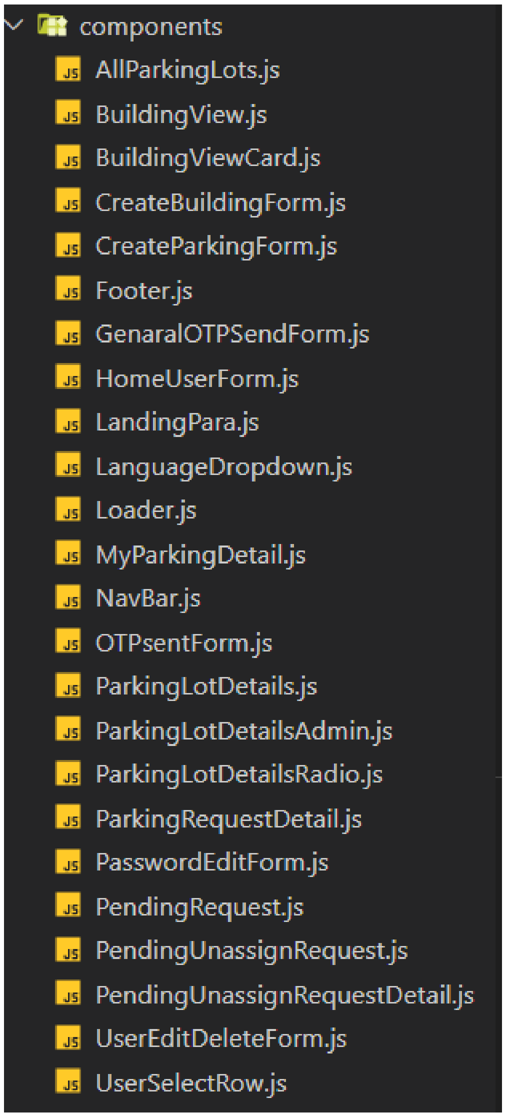


```
assets\img
  AvailableStamp.png
  LandingLogo.png
  MaintenanceStamp.png
  NavLogo.png
  no_image.png
  no_img_large.png
  no_img_mid.png
  no_img_small.png
  PendingStamp.png
  ReservedStamp.png
```

Figure 26. Assets directory

Components Directory

All reusable components have been added in the components. Figure 27 illustrates the front-end component structure, showcasing the organization of all component directories within the application.

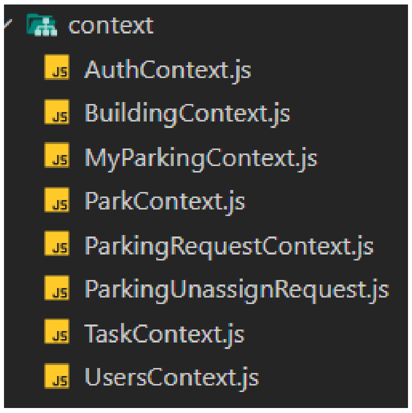


```
components
    AllParkingLots.js
    BuildingView.js
    BuildingViewCard.js
    CreateBuildingForm.js
    CreateParkingForm.js
    Footer.js
    GenaralOTPSendForm.js
    HomeUserForm.js
    LandingPara.js
    LanguageDropdown.js
    Loader.js
    MyParkingDetail.js
    NavBar.js
    OTPsentForm.js
    ParkingLotDetails.js
    ParkingLotDetailsAdmin.js
    ParkingLotDetailsRadio.js
    ParkingRequestDetail.js
    PasswordEditForm.js
    PendingRequest.js
    PendingUnassignRequest.js
    PendingUnassignRequestDetail.js
    UserEditDeleteForm.js
    UserSelectRow.js
```

Figure 27. Front-end component structure

Context Directory

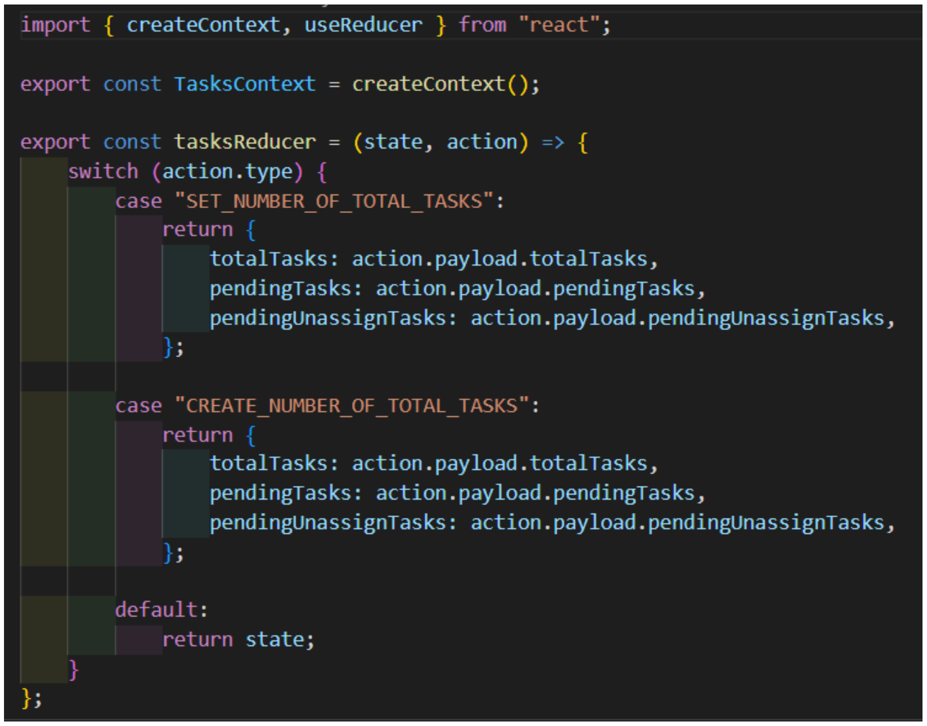
The frontend client uses react API module (Not redux), all the custom contexts are saved in context folder, and all front-end states are managed via react custom context. Therefore, all the context maintain files are stored in the context directory. Figure 28 illustrates the front-end context structure, showcasing the organization of context files within the application.



```
context
  AuthContext.js
  BuildingContext.js
  MyParkingContext.js
  ParkContext.js
  ParkingRequestContext.js
  ParkingUnassignRequest.js
  TaskContext.js
  UsersContext.js
```

Figure 28. Front-end context structure

Figure 29 presents a code snippet that defines the task context file, illustrating the structure and management of task-related data within the application.



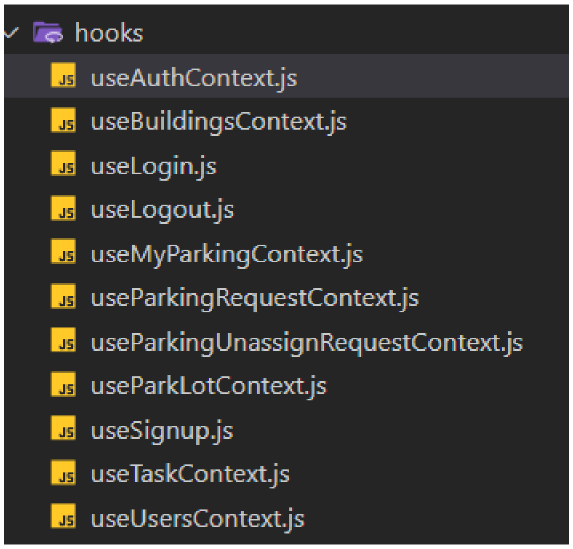
```javascript
import { createContext, useReducer } from "react";

export const TasksContext = createContext();

export const tasksReducer = (state, action) => {
    switch (action.type) {
        case "SET_NUMBER_OF_TOTAL_TASKS":
            return {
                totalTasks: action.payload.totalTasks,
                pendingTasks: action.payload.pendingTasks,
                pendingUnassignTasks: action.payload.pendingUnassignTasks,
            };

        case "CREATE_NUMBER_OF_TOTAL_TASKS":
            return {
                totalTasks: action.payload.totalTasks,
                pendingTasks: action.payload.pendingTasks,
                pendingUnassignTasks: action.payload.pendingUnassignTasks,
            };

        default:
            return state;
    }
};
```

Figure 29 Front-end taskContext.js file

Hooks Directory

All the custom react hooks are stored in the Hooks Directory. Furthermore the relevant hooks are being used in the front-end components as per the requirement.

For example, the user must be authenticated for the entire frontend session and the "useAuth" hook is used achieve the same. Figure 30 illustrates the structure of custom hook files, showcasing how hooks are organized and utilized within the application.



```
hooks
  useAuthContext.js
  useBuildingsContext.js
  useLogin.js
  useLogout.js
  useMyParkingContext.js
  useParkingRequestContext.js
  useParkingUnassignRequestContext.js
  useParkLotContext.js
  useSignup.js
  useTaskContext.js
  useUsersContext.js
```

Figure 30. Custom hooks structure

The react hook triggers the relevant context, and the code snippet illustrated in Figure 31 express's the structure of the custom-made hook called "useAuth" illustrating how authentication logic is managed and utilized within the front-end of the application.

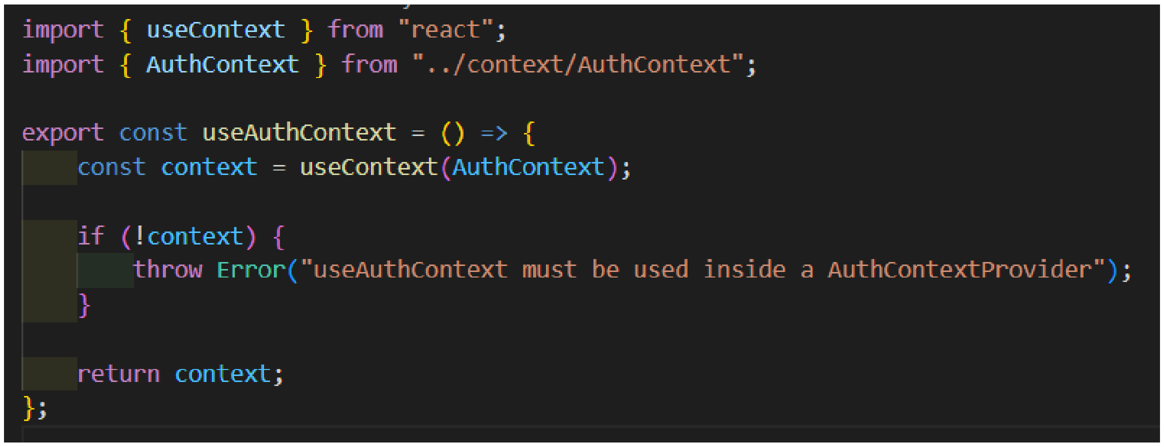

```
import { useContext } from "react";
import { AuthContext } from "../context/AuthContext";

export const useAuthContext = () => {
    const context = useContext(AuthContext);

    if (!context) {
        throw Error("useAuthContext must be used inside a AuthContextProvider");
    }

    return context;
};
```

Figure 31. Front-end useAuth custom made hook

Pages Directory

All static page components are stored in pages directory and the admin page components are separated in “admin” directory. Figure 32 illustrates the front-end pages directory, showcasing the organization and structure of all the pages within the application.

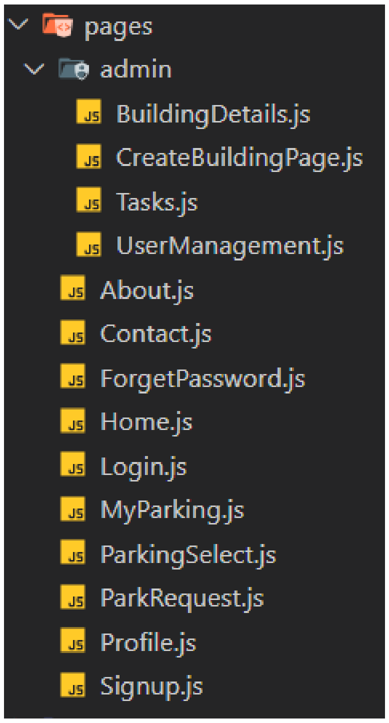
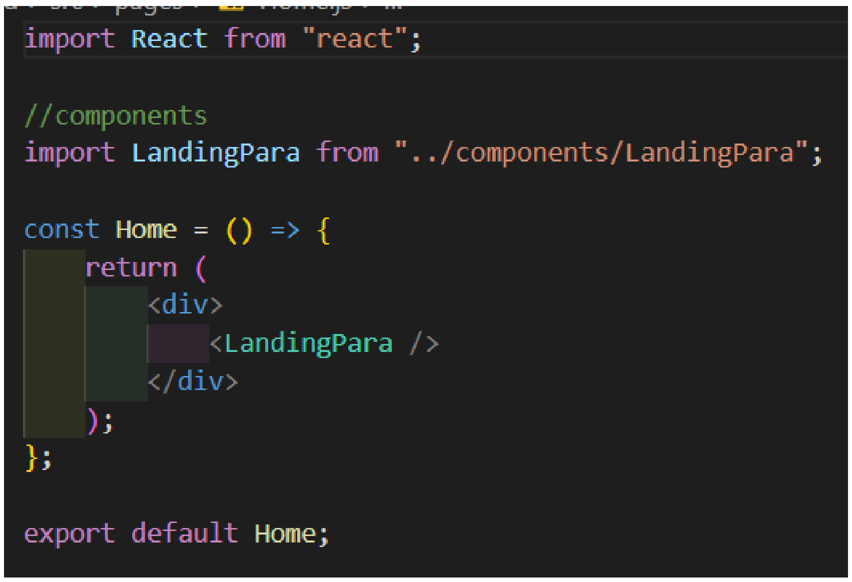

```
pages
  admin
    BuildingDetails.js
    CreateBuildingPage.js
    Tasks.js
    UserManagement.js
  About.js
  Contact.js
  ForgetPassword.js
  Home.js
  Login.js
  MyParking.js
  ParkingSelect.js
  ParkRequest.js
  Profile.js
  Signup.js
```

Figure 32. Front-end pages directory

For example, the landing page component renders the landing paragraph component after making the component connection. Figure 33 illustrates the landing/home page component, showcasing its structure and how it serves as the entry point to the application.

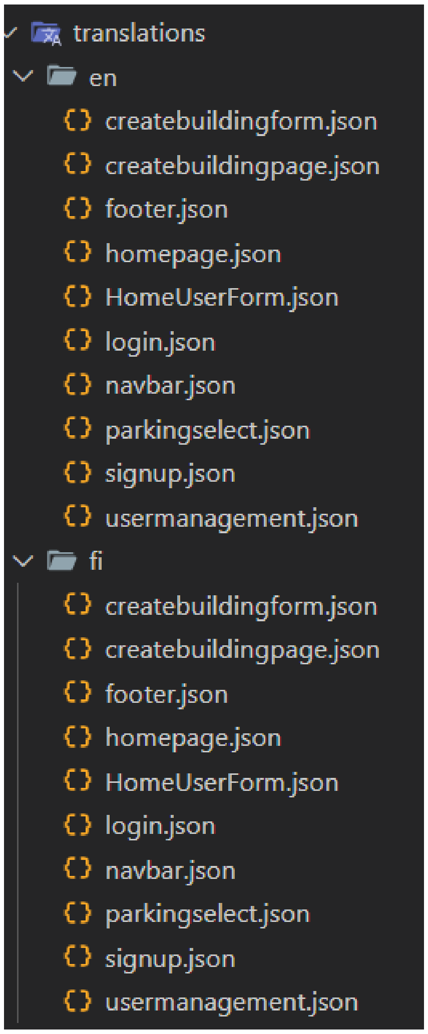


```
import React from "react";

//components
import LandingPara from "../components/LandingPara";

const Home = () => {
    return (
        <div>
            <LandingPara />
        </div>
    );
};

export default Home;
```

Figure 33 Landing/ Home page component

Translations Directory

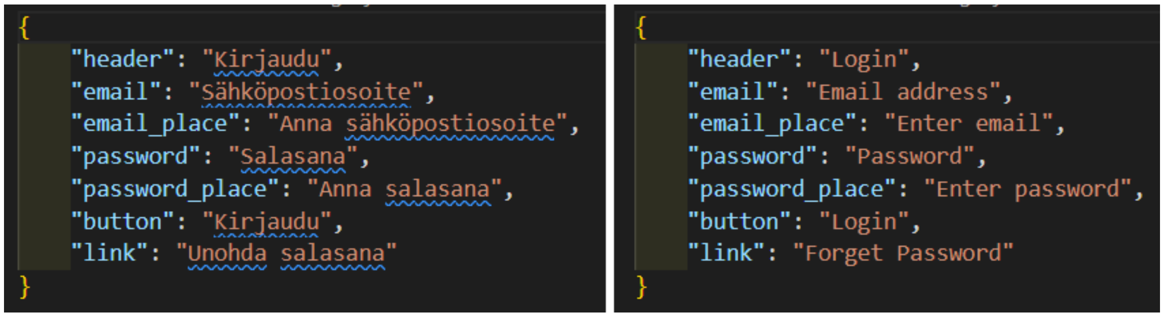
The application is compatible with both Finnish and English languages and the static language JSON file structure will assist to keep the clean code all the time. Figure 35 illustrates the translations directory structure, showcasing how localization and translation files are organized within the application.



```
translations
  en
    createbuildingform.json
    createbuildingpage.json
    footer.json
    homepage.json
    HomeUserForm.json
    login.json
    navbar.json
    parkingselect.json
    signup.json
    usermanagement.json
  fi
    createbuildingform.json
    createbuildingpage.json
    footer.json
    homepage.json
    HomeUserForm.json
    login.json
    navbar.json
    parkingselect.json
    signup.json
    usermanagement.json
```

Figure 34. Translations directory structure

Figure 35 presents the Finnish and English translation JSON files, illustrating how language-specific translations are stored and managed within the application.



```
{
    "header": "Kirjaudu",
    "email": "Sähköpostiosoite",
    "email_place": "Anna sähköpostiosoite",
    "password": "Salasana",
    "password_place": "Anna salasana",
    "button": "Kirjaudu",
    "link": "Unohda salasana"
}
```

```
{
    "header": "Login",
    "email": "Email address",
    "email_place": "Enter email",
    "password": "Password",
    "password_place": "Enter password",
    "button": "Login",
    "link": "Forget Password"
}
```

Figure 35. Finnish and English translation json files

Utils Directory

In utils directory, the utility related files are stored as illustrated in Figure 36.

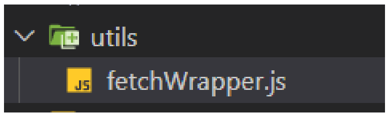
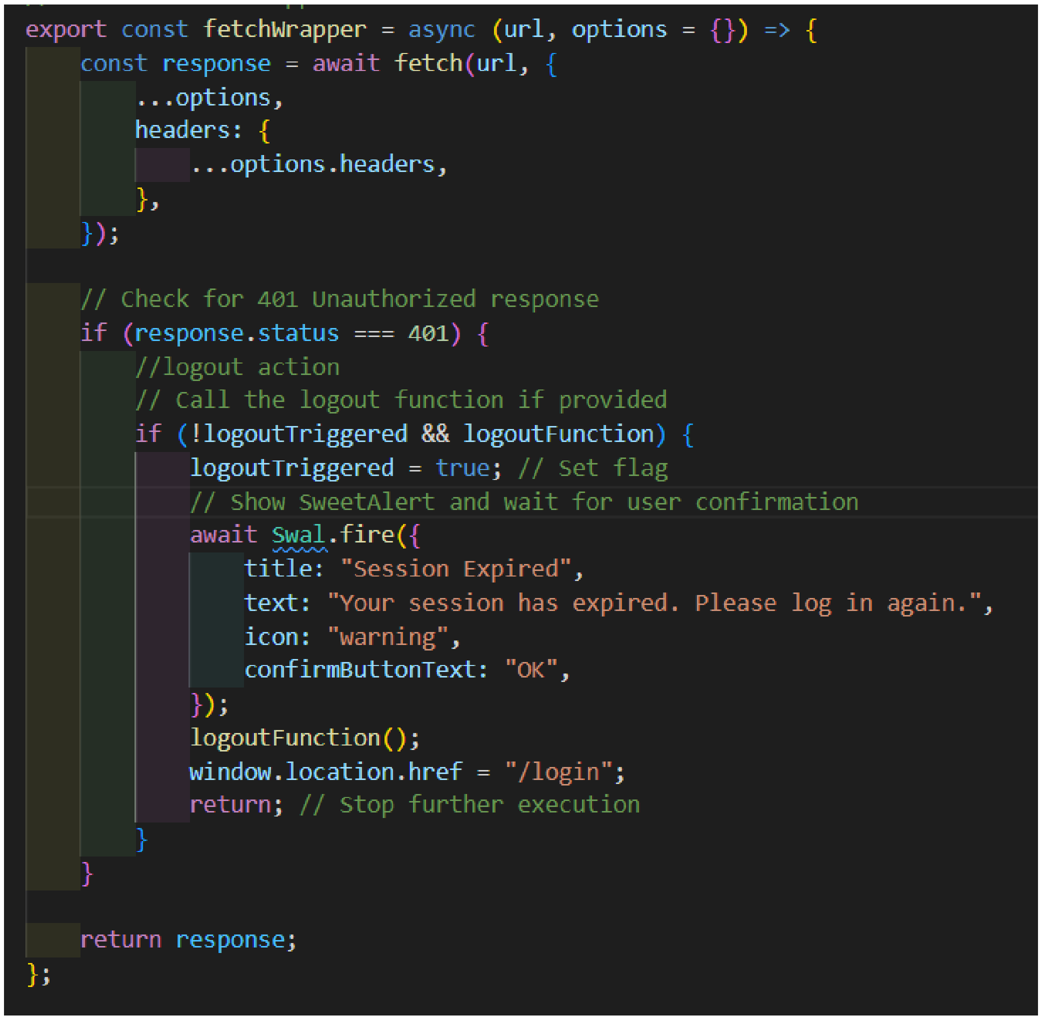


Figure 36 Utility file

In Figure 37 the http interceptor has been added with an objective of catching the status code number 401 (unauthorized) and request the user to log back in. Figure 38 illustrates the fetch interceptor, showcasing how HTTP requests are intercepted and managed before being sent or after receiving a response.

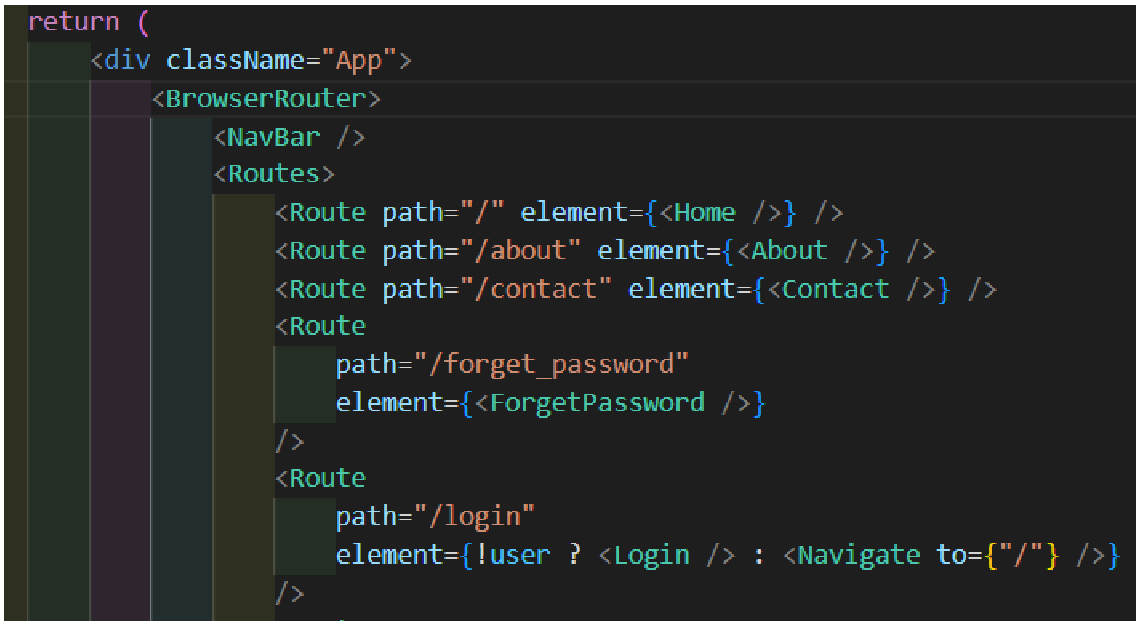


```
export const fetchWrapper = async (url, options = {}) => {
    const response = await fetch(url, {
        ...options,
        headers: {
            ...options.headers,
        },
    });

    // Check for 401 Unauthorized response
    if (response.status === 401) {
        //logout action
        // Call the logout function if provided
        if (!logoutTriggered && logoutFunction) {
            logoutTriggered = true; // Set flag
            // Show SweetAlert and wait for user confirmation
            await Swal.fire({
                title: "Session Expired",
                text: "Your session has expired. Please log in again.",
                icon: "warning",
                confirmButtonText: "OK",
            });
            logoutFunction();
            window.location.href = "/login";
            return; // Stop further execution
        }
    }

    return response;
};
```

Figure 37 Fetch interceptor

App.js file

All front-end routes logics are stored in the app.js file and the routes are mapped as shown in Figure 38. Figure 38 illustrates the front-end routes, showcasing how different application views and components are mapped to their respective URL paths.

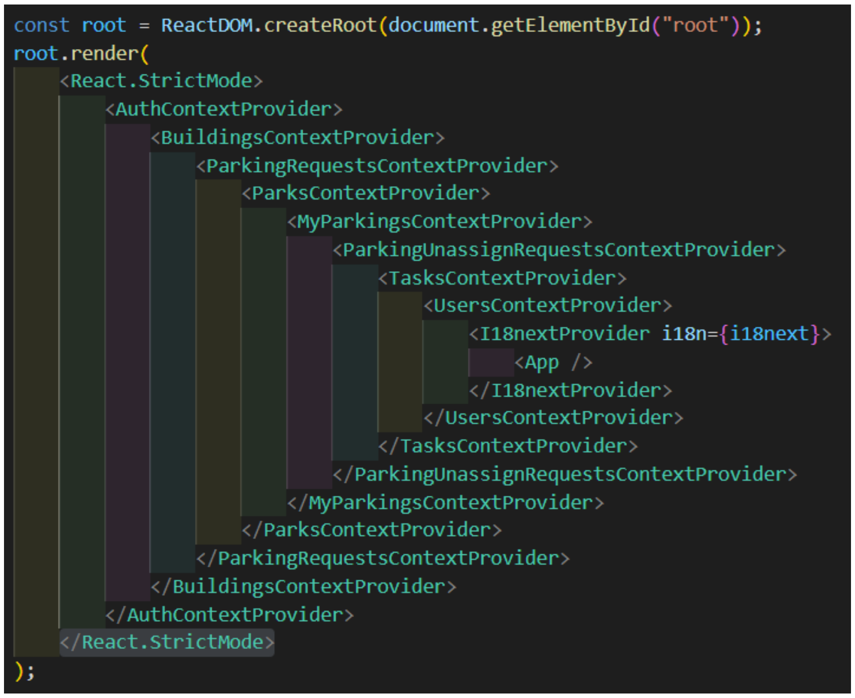


```
return (
  <div className="App">
    <BrowserRouter>
      <NavBar />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
        <Route
          path="/forget_password"
          element={<ForgetPassword />}
        />
        <Route
          path="/login"
          element={!user ? <Login /> : <Navigate to={"/"} />}
        />
```

Figure 38 Front-end routes

Index.js file

The app component is rendered inside the index.js and all custom context providers are wrapped. Figure 39 illustrates the context providers in the index.js file, showcasing how global state and context are initialized and provided to the entire application.

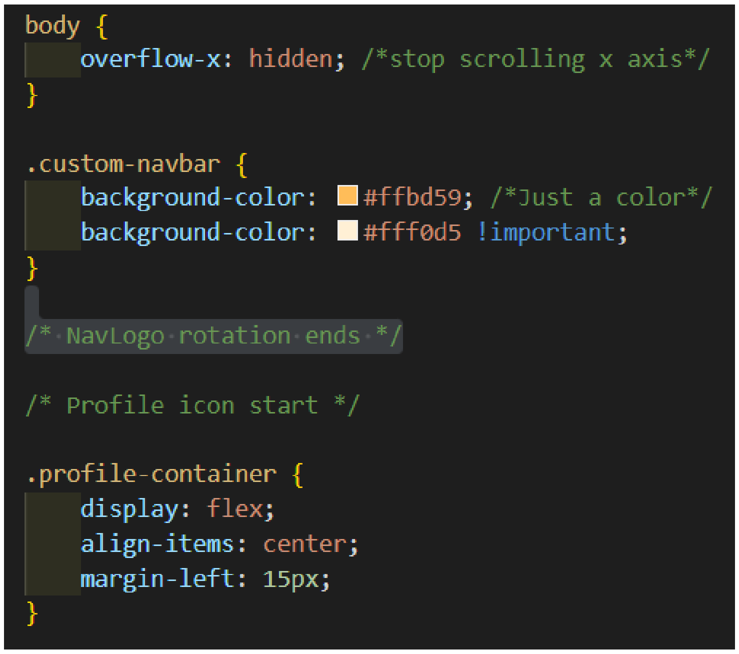


```
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
    <React.StrictMode>
        <AuthContextProvider>
            <BuildingsContextProvider>
                <ParkingRequestsContextProvider>
                    <ParksContextProvider>
                        <MyParkingsContextProvider>
                            <ParkingUnassignRequestsContextProvider>
                                <TasksContextProvider>
                                    <UsersContextProvider>
                                        <I18nextProvider i18n={i18next}>
                                            <App />
                                        </I18nextProvider>
                                    </UsersContextProvider>
                                </TasksContextProvider>
                            </ParkingUnassignRequestsContextProvider>
                        </MyParkingsContextProvider>
                    </ParksContextProvider>
                </ParkingRequestsContextProvider>
            </BuildingsContextProvider>
        </AuthContextProvider>
    </React.StrictMode>
);
```

Figure 39. Context providers in the index.js

Index.css file

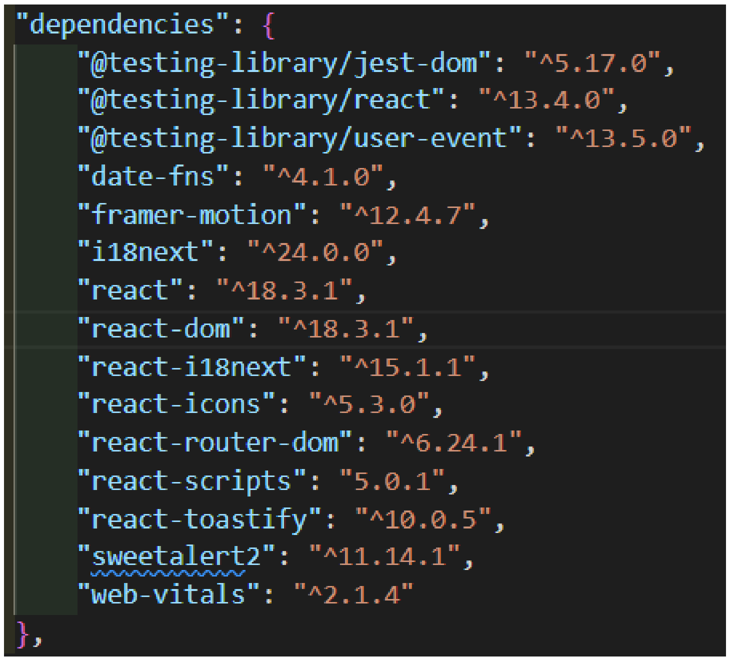
All custom css has been implemented in index.css file. Figure 40 presents the index.css file, illustrating the global styles and CSS rules that are applied throughout the application.



```
body {
    overflow-x: hidden; /*stop scrolling x axis*/
}

.custom-navbar {
    background-color: #ffbd59; /*Just a color*/
    background-color: #fff0d5 !important;
}

/* NavLogo rotation ends */

/* Profile icon start */

.profile-container {
    display: flex;
    align-items: center;
    margin-left: 15px;
}
```

Figure 40. index.css file

Apart from the main architecture the NPM packages shown in Table 2 are installed on the friend-end client side. As shown in Figure 41, package.json consists of all the dependencies. Figure 42 illustrates the front-end dependencies, showcasing the key libraries and packages required for the application's front-end functionality.



```
"dependencies": {
    "@testing-library/jest-dom": "^5.17.0",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "date-fns": "^4.1.0",
    "framer-motion": "^12.4.7",
    "i18next": "^24.0.0",
    "react": "^18.3.1",
    "react-dom": "^18.3.1",
    "react-i18next": "^15.1.1",
    "react-icons": "^5.3.0",
    "react-router-dom": "^6.24.1",
    "react-scripts": "5.0.1",
    "react-toastify": "^10.0.5",
    "sweetalert2": "^11.14.1",
    "web-vitals": "^2.1.4"
},
```

Figure 41. Front-end dependencies

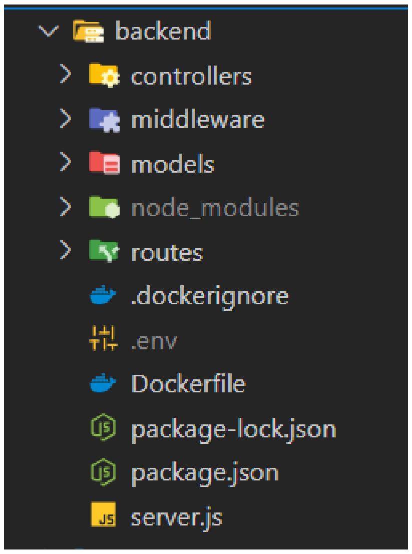
Table 2 presents the NPM package description, detailing the metadata and dependencies listed in the package.json file for the application.

NPM package name	Reason
date-fns	Formatting the front-end date and time stamps.
Framer-motion	Components animations.
React-i18next	Language Translations.
React-toastify / sweetalert2	Implementing pop ups and alerts.

Table 42. NPM pakage description

4.3 Back-end Development (Node.js, Express.js)

The Node.js back-end serves as an API provider, handling authentication, parking reservations, and admin actions. All the RESTful API are handled via express.js. Figure 42 illustrates the backend directory structure, showcasing the organization of files and folders within the back-end portion of the application.



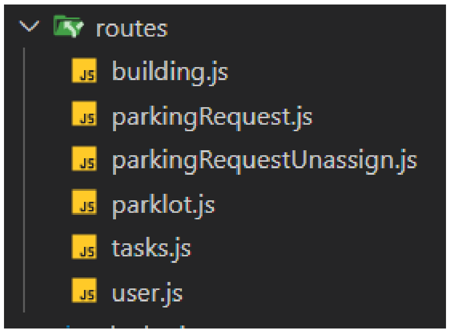
```
backend
  controllers
  middleware
  models
  node_modules
  routes
  .dockerignore
  .env
  Dockerfile
  package-lock.json
  package.json
  server.js
```

Figure 432. Back-end directory structure

The backend directory consists of controllers, middleware, models and routes directories as well as server.js and dotenv file.

Routes directory

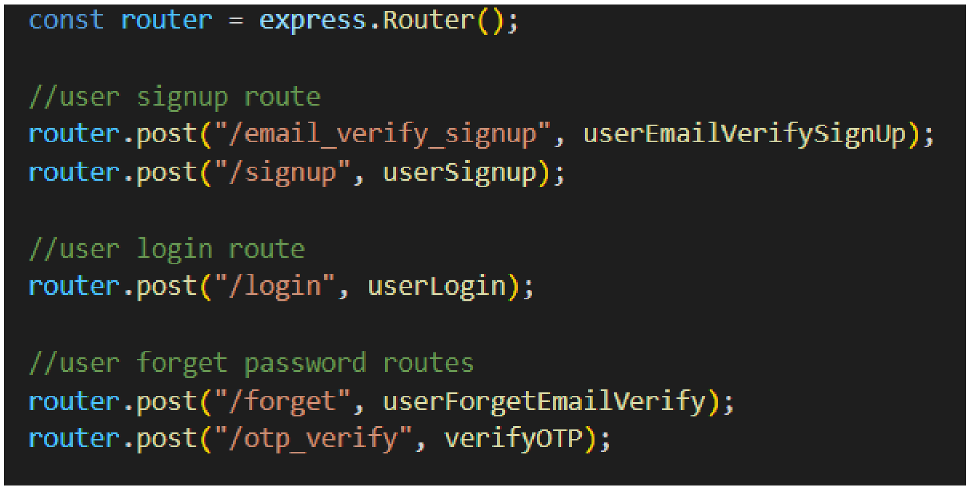
In the routes directory, all the backend routes have been saved pertaining to the relevant module.



```
routes
  building.js
  parkingRequest.js
  parkingRequestUnassign.js
  parklot.js
  tasks.js
  user.js
```

Figure 43. Back-end routes directory

For example, the user CRUD operation routes have been added in the “user.js” file. Figure 44 illustrates the back-end user routes, showcasing the API endpoints responsible for handling user-related requests in the application.



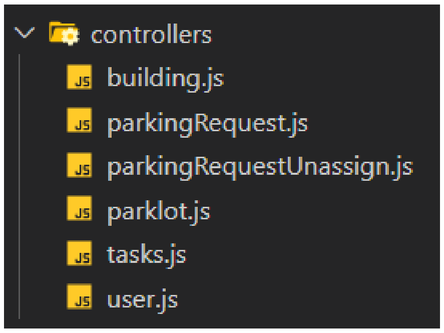
```
const router = express.Router();

//user signup route
router.post("/email_verify_signup", userEmailVerifySignUp);
router.post("/signup", userSignup);

//user login route
router.post("/login", userLogin);

//user forget password routes
router.post("/forget", userForgetEmailVerify);
router.post("/otp_verify", verifyOTP);
```

Figure 44 Back-end user routes

As illustrated by the code snippet in Figure 46, once the relevant back-end API end points are activated, the relevant controller function will be executed for retrieving the relevant data as per the request.

Controllers directory

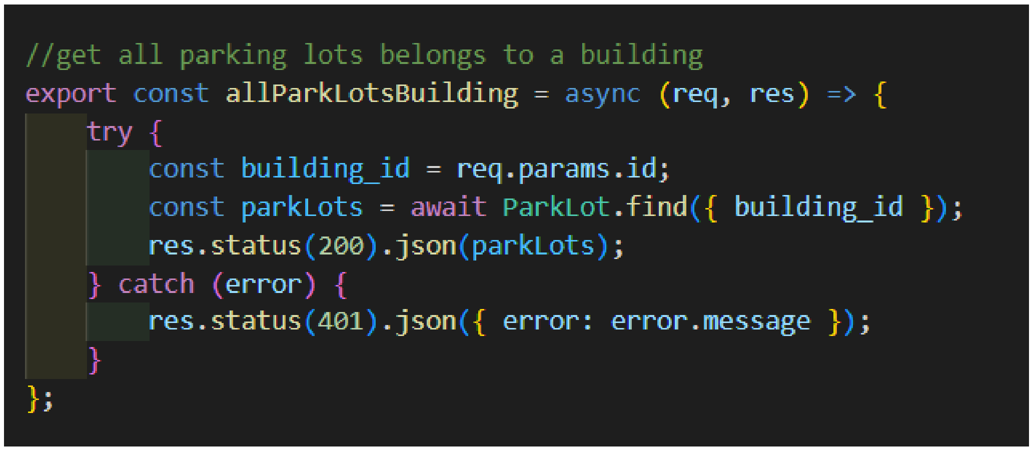
As per the API end point, the relevant controller functions are implemented in controllers directory and all controller functions are segregated as per the module. Figure 45 illustrates the back-end controller directory, showcasing the organization of controller files responsible for handling business logic and request responses in the application.



```
controllers
    building.js
    parkingRequest.js
    parkingRequestUnassign.js
    parklot.js
    tasks.js
    user.js
```

Figure 45. Back-end controller directory

Figure 46 presents the back-end controller function. For example, in order to retrieve all the parking slots that belong to a building. Figure 46 illustrates the controller function responsible for retrieving all parking slots, showcasing how the backend handles the request to fetch parking slot data from the database.



```
//get all parking lots belongs to a building
export const allParkLotsBuilding = async (req, res) => {
    try {
        const building_id = req.params.id;
        const parkLots = await ParkLot.find({ building_id });
        res.status(200).json(parkLots);
    } catch (error) {
        res.status(401).json({ error: error.message });
    }
};
```

Figure 46 Controller function for getting all the parking slots

Models directory

Models' directory consists of all database schemas as per the root separation. Figure 47 illustrates the models directory, showcasing the organization of model files that define the data schema and structure used throughout the application.

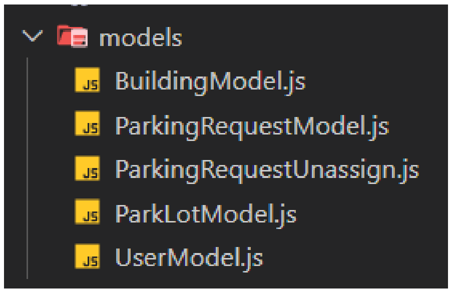
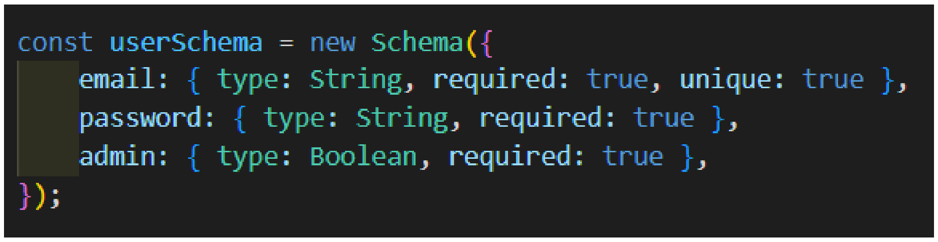


Figure 47. Models directory

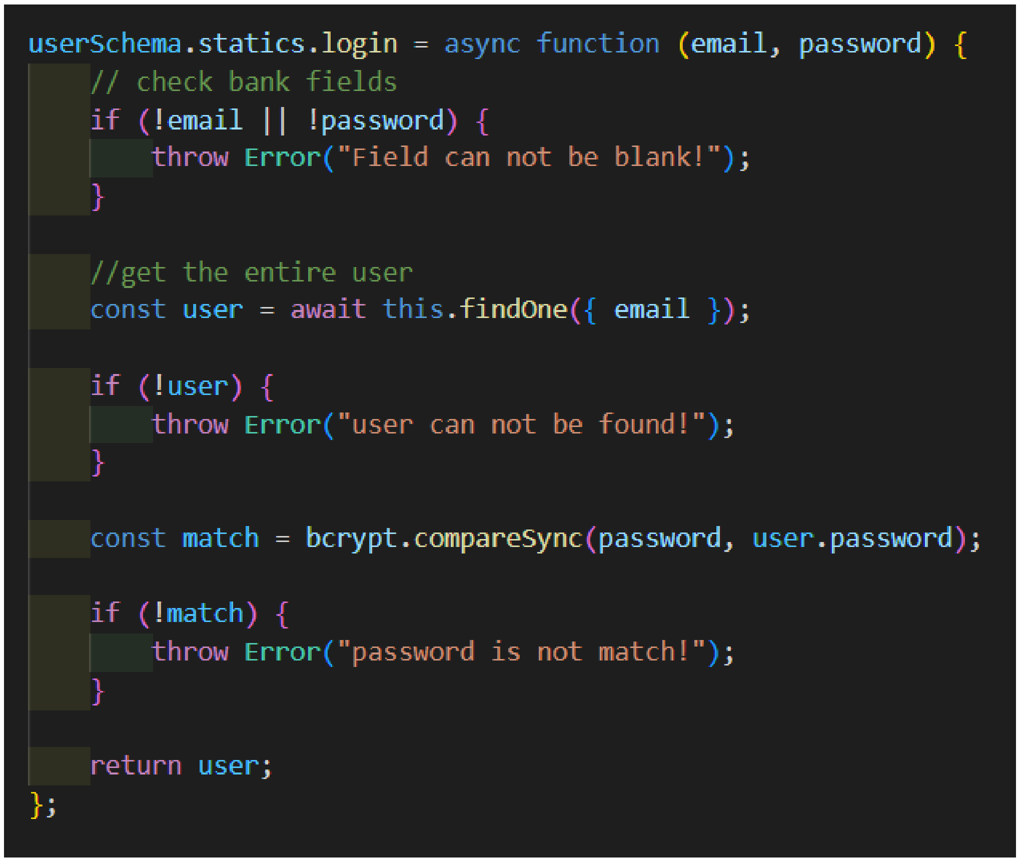
For example “userModel.js” file contains the user schema architecture. Figure 48 illustrates the user schema, showcasing the structure and fields used to define and store user-related data in the database.



```
const userSchema = new Schema({
    email: { type: String, required: true, unique: true },
    password: { type: String, required: true },
    admin: { type: Boolean, required: true },
});
```

Figure 48. User schema

Figure 49 illustrates the back-end static login function, showcasing how the application handles user authentication and login requests.



```
userSchema.statics.login = async function (email, password) {
    // check bank fields
    if (!email || !password) {
        throw Error("Field can not be blank!");
    }

    //get the entire user
    const user = await this.findOne({ email });

    if (!user) {
        throw Error("user can not be found!");
    }

    const match = bcrypt.compareSync(password, user.password);

    if (!match) {
        throw Error("password is not match!");
    }

    return user;
};
```

Figure 49. Back-end static login function

Middleware directory

The middleware directory consists of all the middleware files of the backend. (MERN Authentication Tutorial 2022).

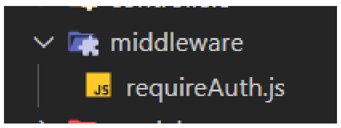
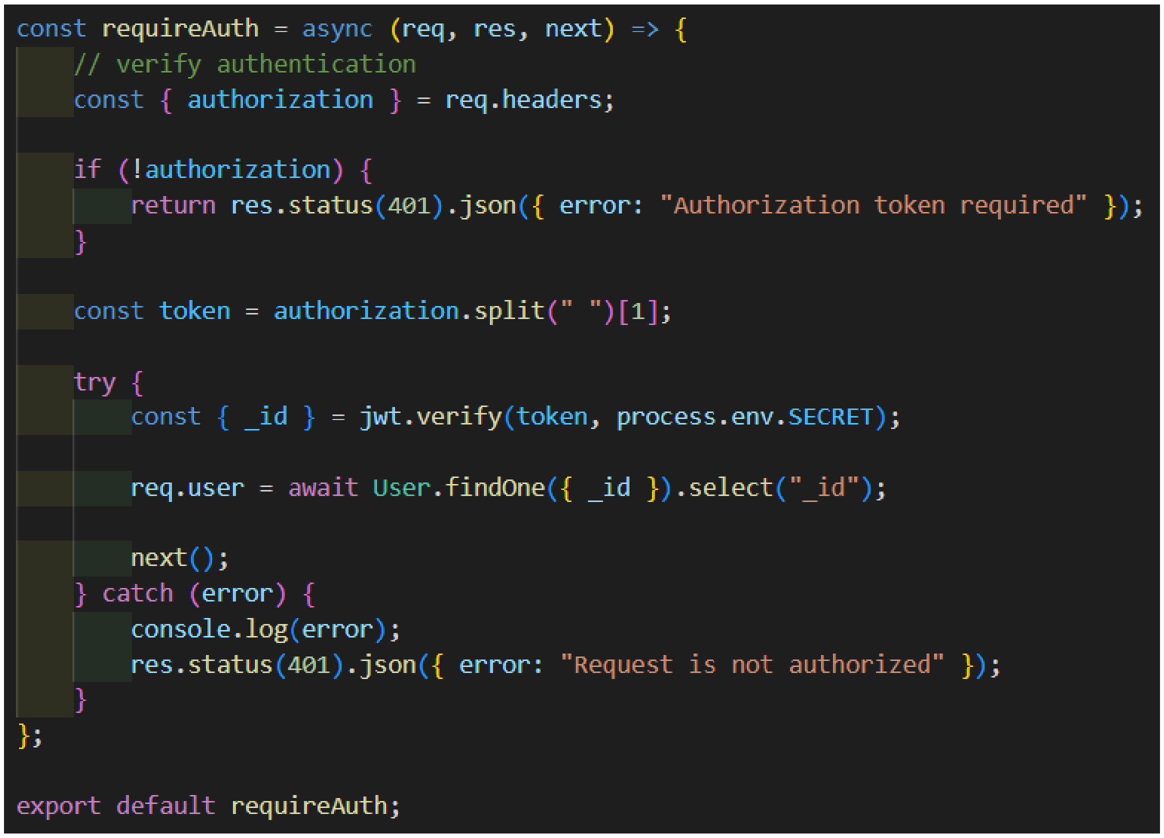


Figure 440. Middleware implementation

The token verification process is handled by the requireAuth.js file as shown in Figure 51.

Figure 51 illustrates the JSON Web Token (JWT) verification implemented as middleware, showcasing how the application validates and ensures secure access to protected routes.



```
const requireAuth = async (req, res, next) => {
    // verify authentication
    const { authorization } = req.headers;

    if (!authorization) {
        return res.status(401).json({ error: "Authorization token required" });
    }

    const token = authorization.split(" ")[1];

    try {
        const { _id } = jwt.verify(token, process.env.SECRET);

        req.user = await User.findOne({ _id }).select("_id");

        next();
    } catch (error) {
        console.log(error);
        res.status(401).json({ error: "Request is not authorized" });
    }
};

export default requireAuth;
```

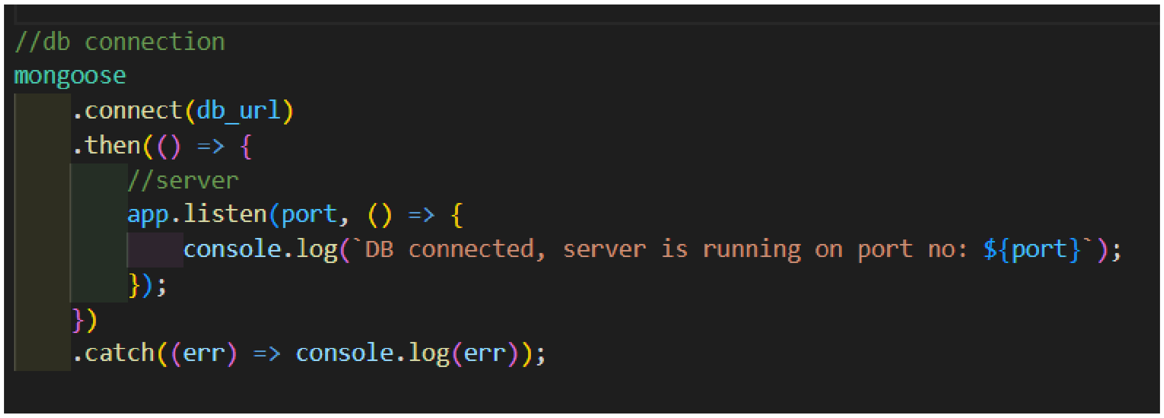
Figure 451. Json web token verification as a middleware

ENV file

the env file includes all the environment variables which are not exposed to any code repository. For example database URL, port numbers and encryption secrets are the most common use.

Server.js file

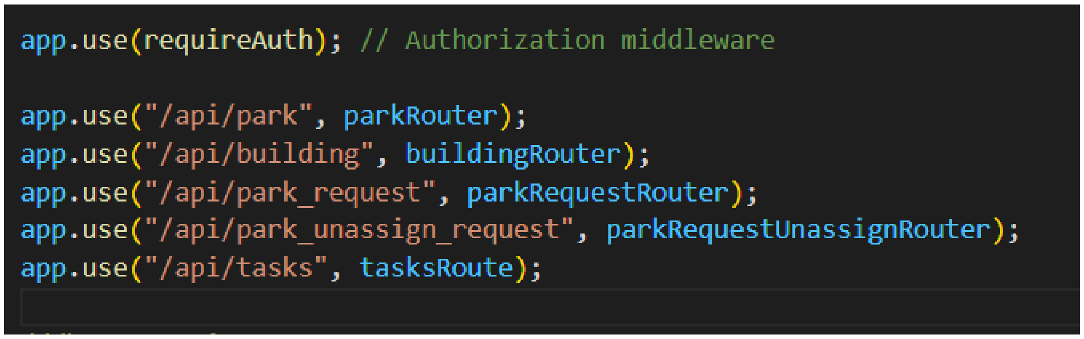
The server is handled by server.js file which consists of the database connection and port specified. Figure 52 illustrates the back-end server function, showcasing how the server handles requests, processes, and responses to the client.



```
//db connection
mongoose
    .connect(db_url)
    .then(() => {
        //server
        app.listen(port, () => {
            console.log(`DB connected, server is running on port no: ${port}`);
        });
    })
    .catch((err) => console.log(err));
```

Figure 46. Back-end server function/ logic

The auth middleware is imported before connecting the protected routes. Figure 53 illustrates the middleware implementation for protected routes, showcasing how specific routes are secured and accessible to authenticated users through token validation.

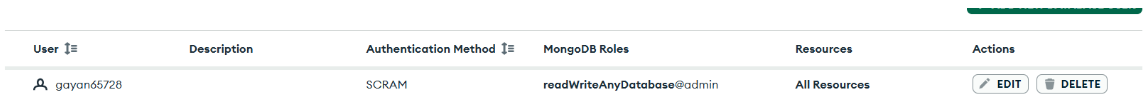


```
app.use(requireAuth); // Authorization middleware

app.use("/api/park", parkRouter);
app.use("/api/building", buildingRouter);
app.use("/api/park_request", parkRequestRouter);
app.use("/api/park_unassign_request", parkRequestUnassignRouter);
app.use("/api/tasks", tasksRoute);
```

Figure 53. Protected routes

4.4 Database Implementation (MongoDB)

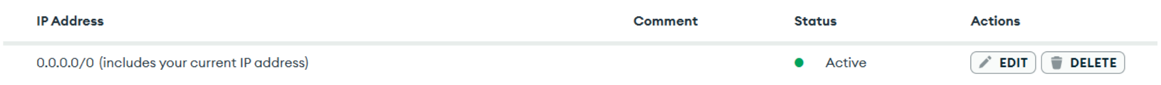
The database connection string is imported as an environment variable due to its highly sensitive nature. As shown in Figure 54 a respective user is added to access the database.



User	Description	Authentication Method	MongoDB Roles	Resources	Actions
gayan65728		SCRAM	readWriteAnyDatabase@admin	All Resources	EDIT DELETE

Figure 54. DB access users

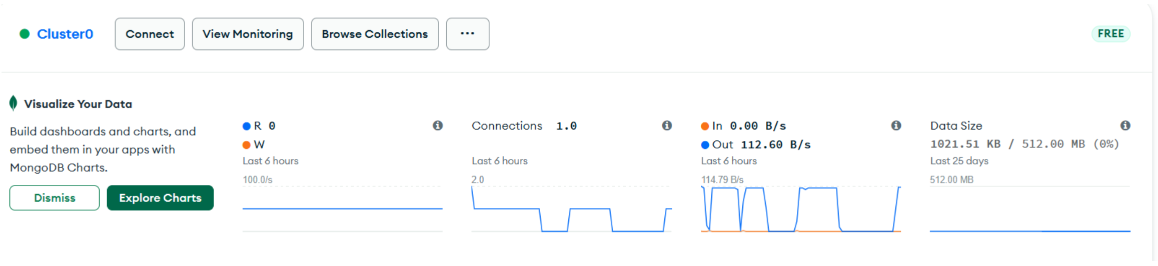
The Network access rules are added as illustrated in Figure 55 for establishing connections from anywhere.



IP Address	Comment	Status	Actions
0.0.0.0/0 (includes your current IP address)		● Active	EDIT DELETE

Figure 475. Network access settings

The default backup policies are activated which will vary with client requests. MongoDB Atlas provides an interactive user dashboard for data as illustrated in Figure 56.



Build dashboards and charts, and embed them in your apps with MongoDB Charts.

R 0
W
Last 6 hours
100.0/s

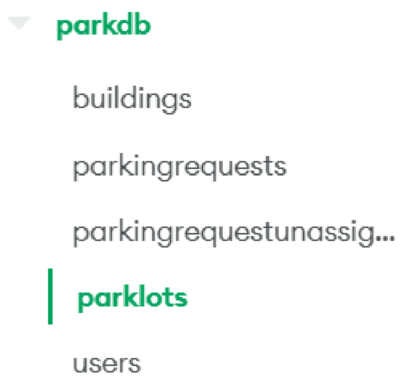
Connections 1.0
Last 6 hours
2.0

In 0.00 B/s
Out 112.60 B/s
Last 6 hours
114.79 B/s

Data Size
1021.51 KB / 512.00 MB (0%)
Last 25 days
512.00 MB

Figure 486. MongoDB data visualization

The data collections are distributed in the database as illustrated in Figure 57.

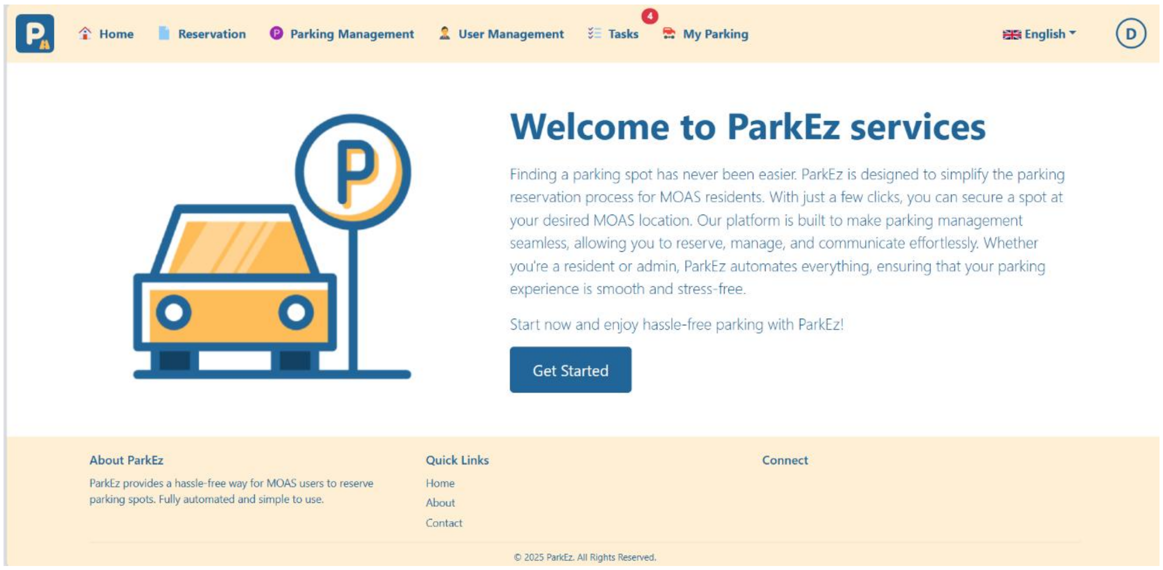


parkdb
- buildings
- parkingrequests
- parkingrequestunassig...
- parklots
- users

Figure 49. Collection distribution in the database

4.5 User Interface (UI) and User Experience (UX) Design

The proposed UI/UX design aims to ensure ease of use, accessibility, and a clean visual layout for both tenets and admin users. Figure 58 illustrates the English version of the home page, showcasing the layout and content displayed to users when the application is viewed in English.



Welcome to ParkEz services

Finding a parking spot has never been easier. ParkEz is designed to simplify the parking reservation process for MOAS residents. With just a few clicks, you can secure a spot at your desired MOAS location. Our platform is built to make parking management seamless, allowing you to reserve, manage, and communicate effortlessly. Whether you're a resident or admin, ParkEz automates everything, ensuring that your parking experience is smooth and stress-free.

Start now and enjoy hassle-free parking with ParkEz!

About ParkEz

ParkEz provides a hassle-free way for MOAS users to reserve parking spots. Fully automated and simple to use.

Quick Links

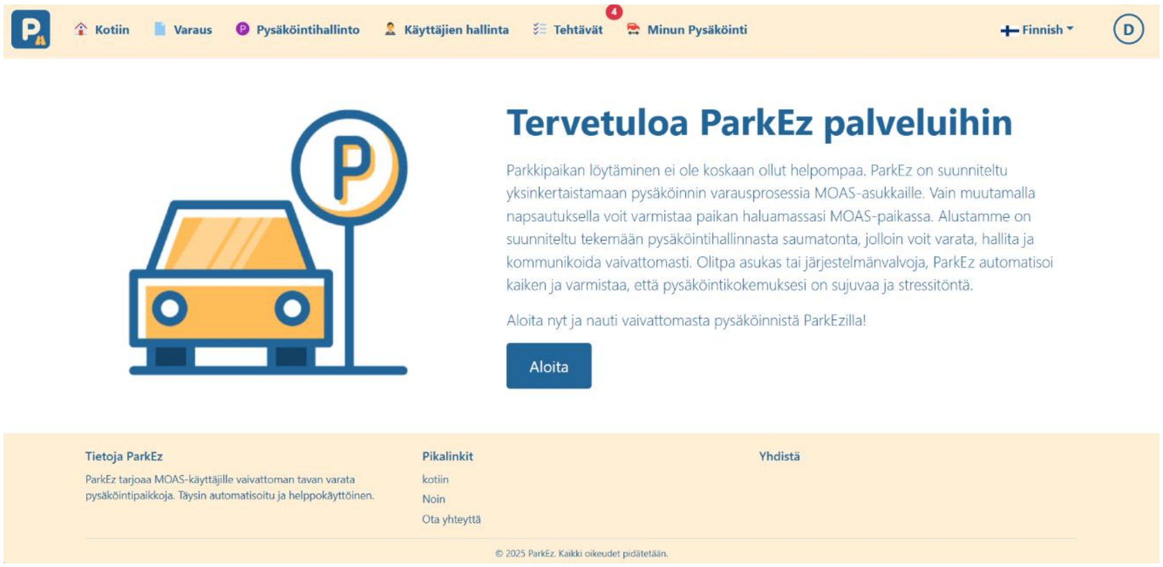
Home
About
Contact

Connect

© 2025 ParkEz. All Rights Reserved.

Figure 508. Home page English version

Figure 59 illustrates the Finnish version of the home page, showcasing the layout and content displayed to users when the application is viewed in Finnish.



Tervetuloa ParkEz palveluihin

Parkkipaikan löytäminen ei ole koskaan ollut helpompaa. ParkEz on suunniteltu yksinkertaistamaan pysäköinnin varausprosessia MOAS-asukkaille. Vain muutamalla napsautuksella voit varmistaa paikan haluamassasi MOAS-paikassa. Alustamme on suunniteltu tekemään pysäköintihallinnasta saumatonta, jolloin voit varata, hallita ja kommunikoida vaivattomasti. Olitpa asukas tai järjestelmänvalvoja, ParkEz automatisoi kaiken ja varmistaa, että pysäköintikokemuksesi on sujuvaa ja stressitöntä.

Aloita nyt ja nauti vaivattomasta pysäköinnistä ParkEzilla!

Tietoja ParkEz

ParkEz tarjoaa MOAS-käyttäjille vaivattoman tavan varata pysäköintipaikkoja. Täysin automatisoitu ja helppokäyttöinen.

Pikalinkit

kotiin
Noin
Ota yhteyttä

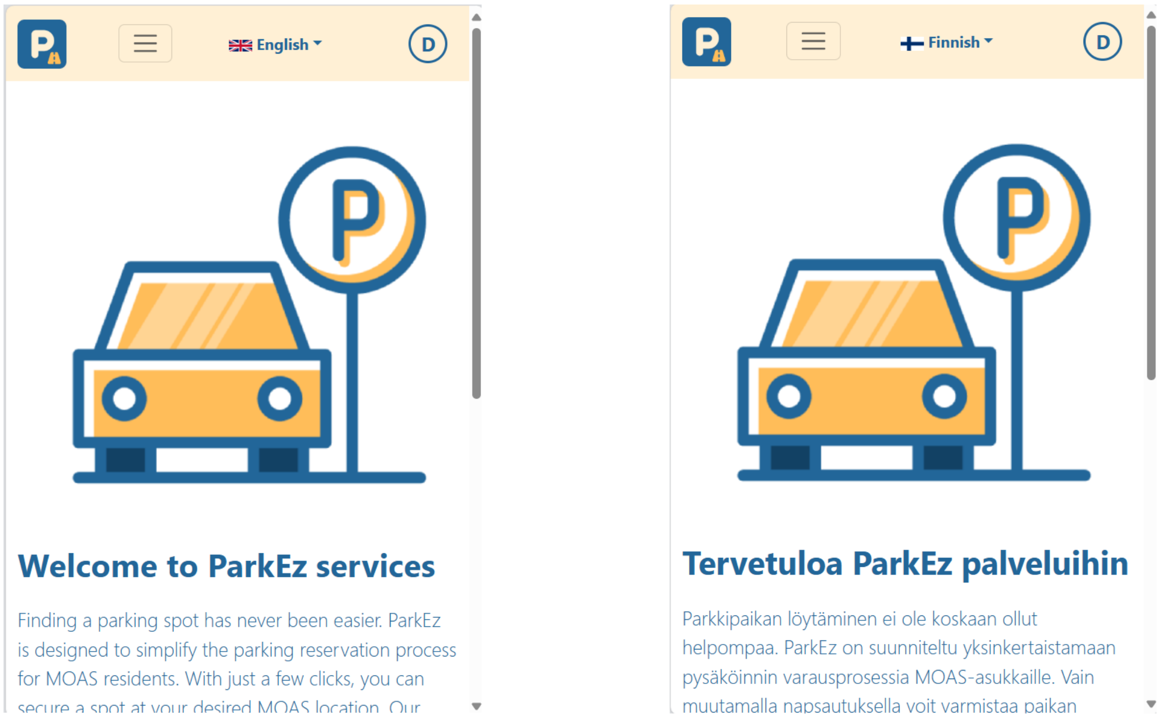
Yhdistä

© 2025 ParkEz. Kaikki oikeudet pidätetään.

Figure 59. Home page Finnish version

The default language of the system is Finnish, and alternatively English language is added for enhancing the user experience. Another key feature is the responsive design which seeks to ensure usability across mobile and desktop devices.

Figure 60 illustrates the responsive design, demonstrating how the application smoothly supports both languages, ensuring a seamless user experience across different screen sizes and languages.



Welcome to ParkEz services

Finding a parking spot has never been easier. ParkEz is designed to simplify the parking reservation process for MOAS residents. With just a few clicks, you can secure a spot at your desired MOAS location. Our

Tervetuloa ParkEz palveluihin

Parkkipaikan löytäminen ei ole koskaan ollut helpompaa. ParkEz on suunniteltu yksinkertaistamaan pysäköinnin varausprosessia MOAS-asukkaille. Vain muutamalla napsautuksella voit varmistaa paikan

Figure 60. Responsive design which supports two languages

As illustrated in Figure 61, the main theme features include #226699 (blue) as the primary colour and #ffbd59 (yellow) and #fff0d5 (beige) as secondary colours. Furthermore the entire application is organized with animations and hover effects for a better user experience.

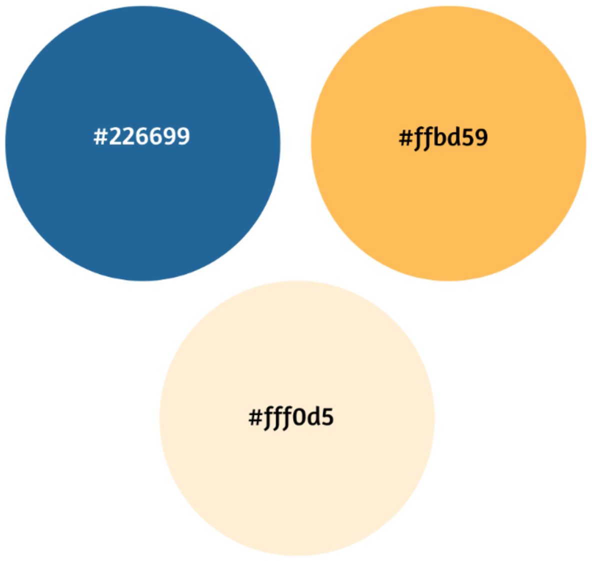


Figure 511. Selected UI colours

Similarly, the positioning of the HTML elements in the entire web application aim to enhance the user experience.

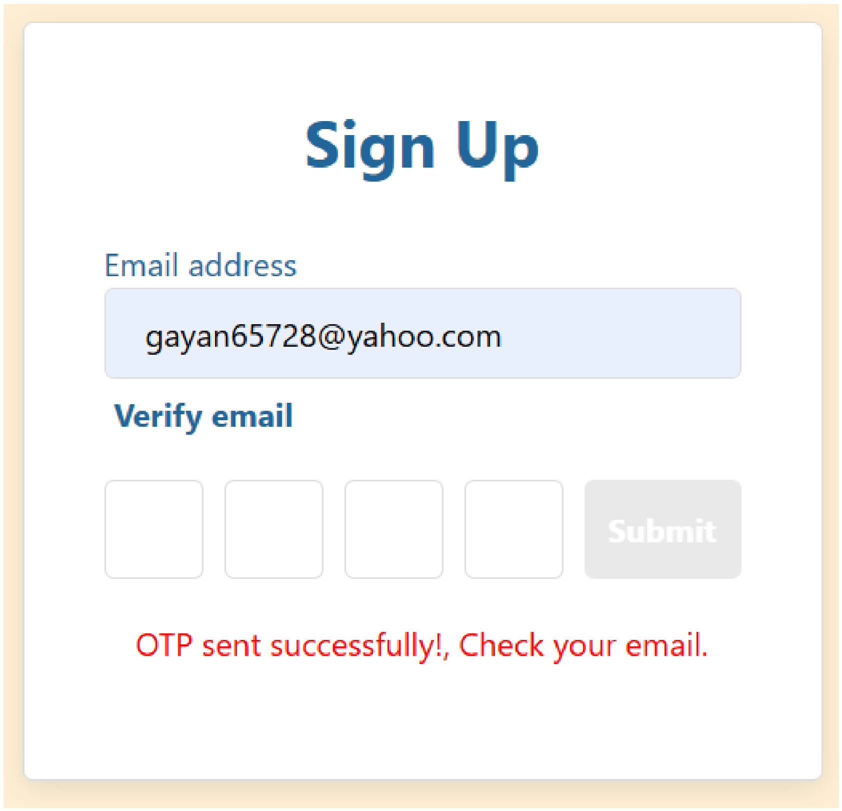
5 SYSTEM FEATURES

The web application workflow and some of the features are described in this section and the relevant technologies are introduced.

5.1 Tenant/ Administration Registration and Login

Registration form

The registration form is used to onboard the user to the system. User's email should be stored in the MOAS/ MIKKALO Oy's database and password and OTP verification should be required for user registration. Both frontend and backend data validation are for better user experience.

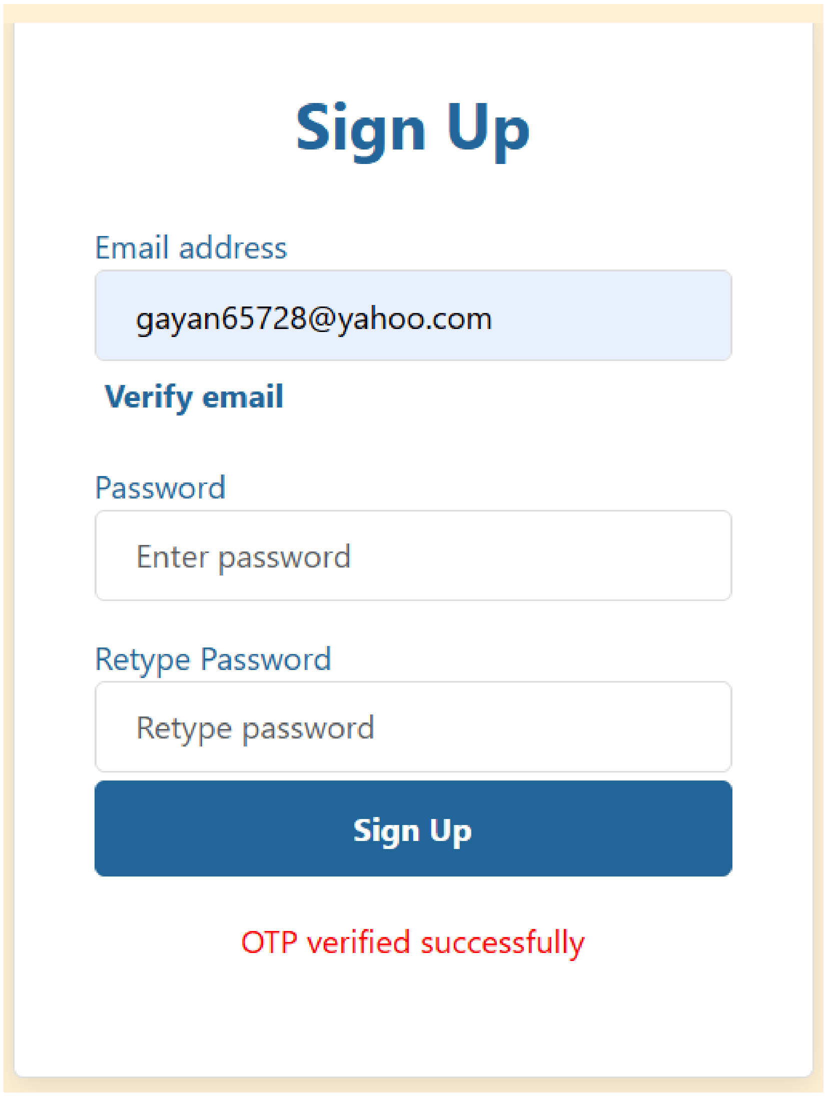


gayan65728@yahoo.com

OTP sent successfully!, Check your email.

Figure 522. Email verification

After the OTP field is successfully addressed, the password fields become visible. The default password rules require at least eight characters, including one uppercase letter, one number, and one special character.


Sign Up

Email address

gayan65728@yahoo.com

Verify email

Password

Enter password

Retype Password

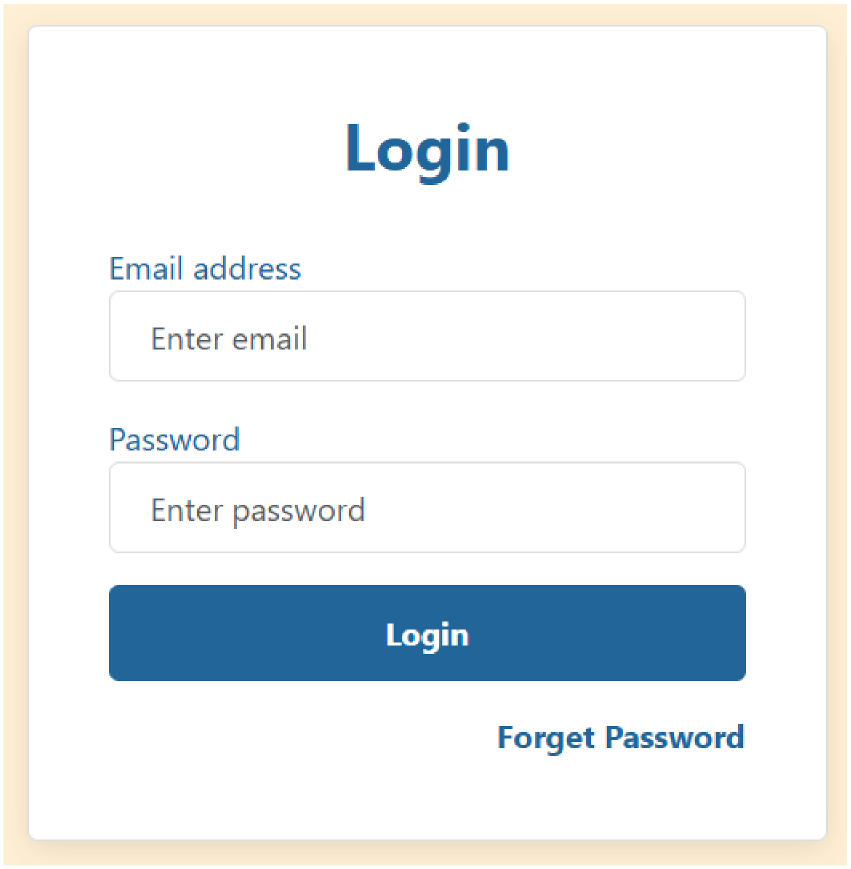
Retype password

OTP verified successfully

Figure 533. Password fields visible

Login form

With the registered credentials, the user can access the login form as illustrated in Figure 64.


Login

Email address

Enter email

Password

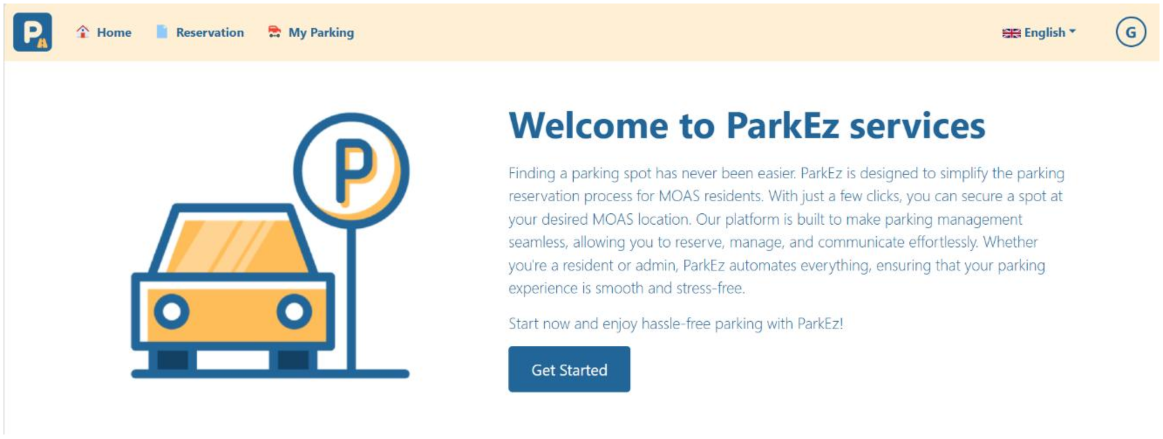
Enter password

Login

Forget Password

Figure 54. Login form

Once the user has successfully registered or logged in, they are directed to the home page, as shown in Figure 65.



Welcome to ParkEz services

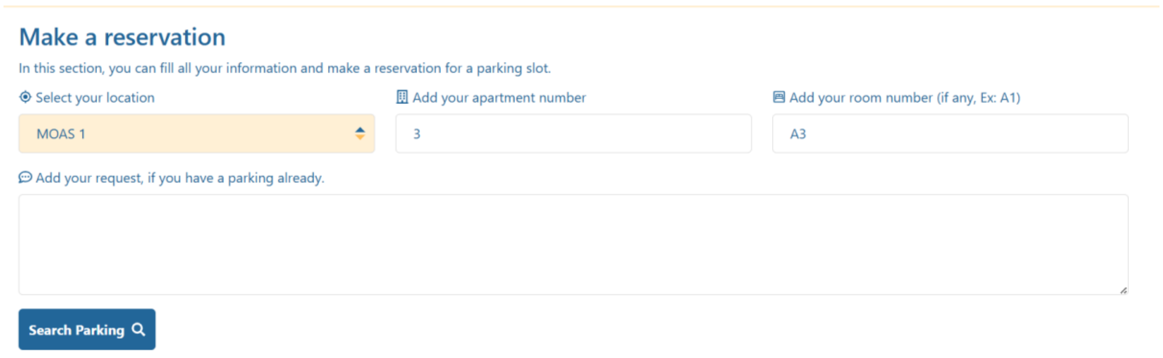
Finding a parking spot has never been easier. ParkEz is designed to simplify the parking reservation process for MOAS residents. With just a few clicks, you can secure a spot at your desired MOAS location. Our platform is built to make parking management seamless, allowing you to reserve, manage, and communicate effortlessly. Whether you're a resident or admin, ParkEz automates everything, ensuring that your parking experience is smooth and stress-free.

Start now and enjoy hassle-free parking with ParkEz!

Figure 555 Home page

5.2 Parking Request System

As illustrated in Figure 68 the user can access the reservation system either via the "Reservation" link in the navigation bar or by clicking the "Get Started" button on the home page interface.



Make a reservation

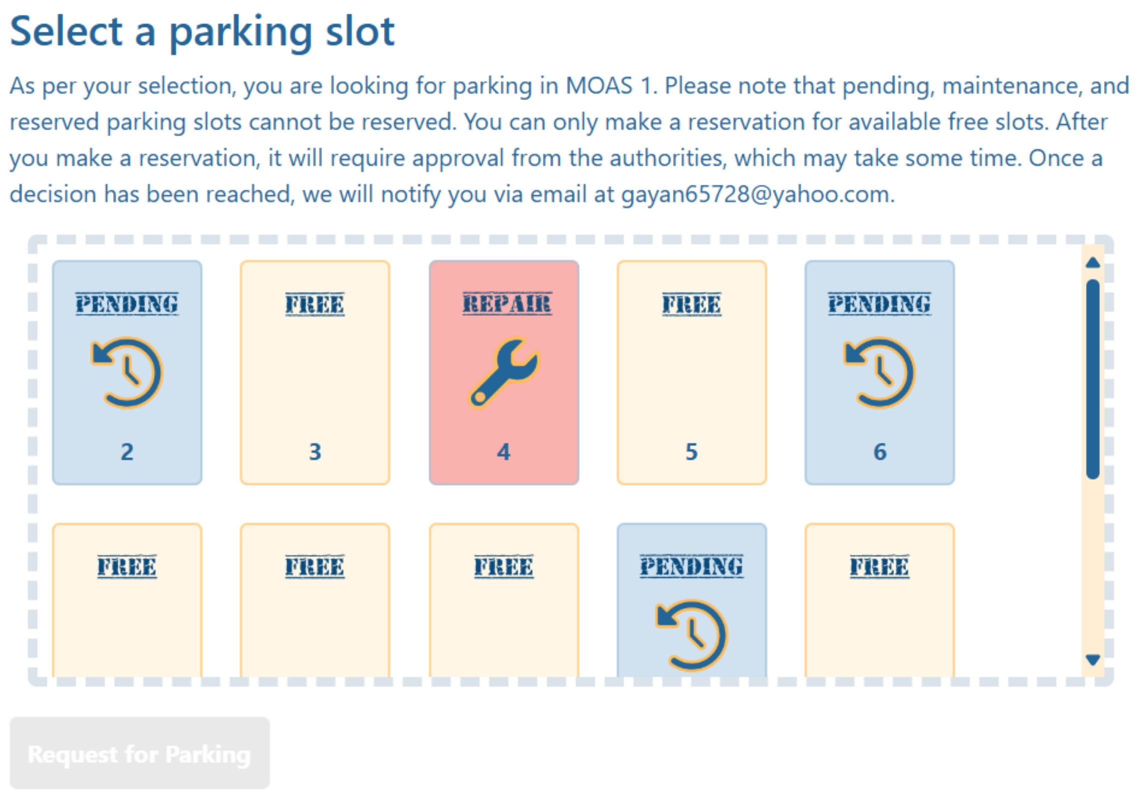
In this section, you can fill all your information and make a reservation for a parking slot.

Select your location	Add your apartment number	Add your room number (if any, Ex: A1)
MOAS 1	3	A3

Add your request, if you have a parking already.

Figure 566. Reservation section

Once the tenant fills in all the necessary information, parking slots can be searched using the "Search Parking" button. This then directs the user to the parking selection phase. Figure 67 illustrates the parking selection layout, showcasing how users can view and choose available parking spots within the application.



Select a parking slot

As per your selection, you are looking for parking in MOAS 1. Please note that pending, maintenance, and reserved parking slots cannot be reserved. You can only make a reservation for available free slots. After you make a reservation, it will require approval from the authorities, which may take some time. Once a decision has been reached, we will notify you via email at gayan65728@yahoo.com.

Figure 577. Parking selection

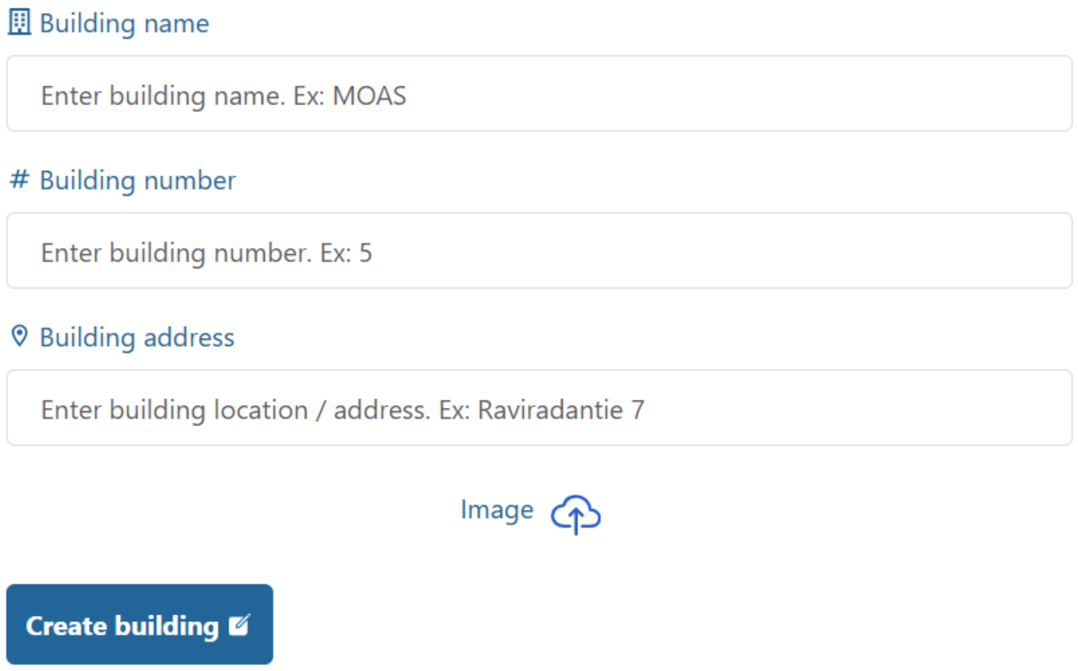
Once a free parking slot is selected, the request is automatically sent, and a notification is forwarded to the authorized officer for verification. An email is also sent to both the tenant and the admin.

5.3 Admin Management Interface

The admin features include creating buildings and their associated parking spots, as well as manually updating the status of parking spots. For example, parking spaces can be marked as available, under maintenance, or allocated. User management is also part of the admin’s role, including deleting users and assigning privileges in accordance with organizational policies. Additionally, the admin is responsible for reviewing and either approving or declining client requests.

Create building

The admin can add a building and its features, such as the building name, building number, building image, and address, which can be uploaded via the form illustrated in Figure 68.


Building name

Enter building name. Ex: MOAS

Building number

Enter building number. Ex: 5

Building address

Enter building location / address. Ex: Raviradantie 7

Figure 588. Building creation form

Once the building is created the relevant building card will be displayed on the right side of the interface as illustrated in Figure 69.

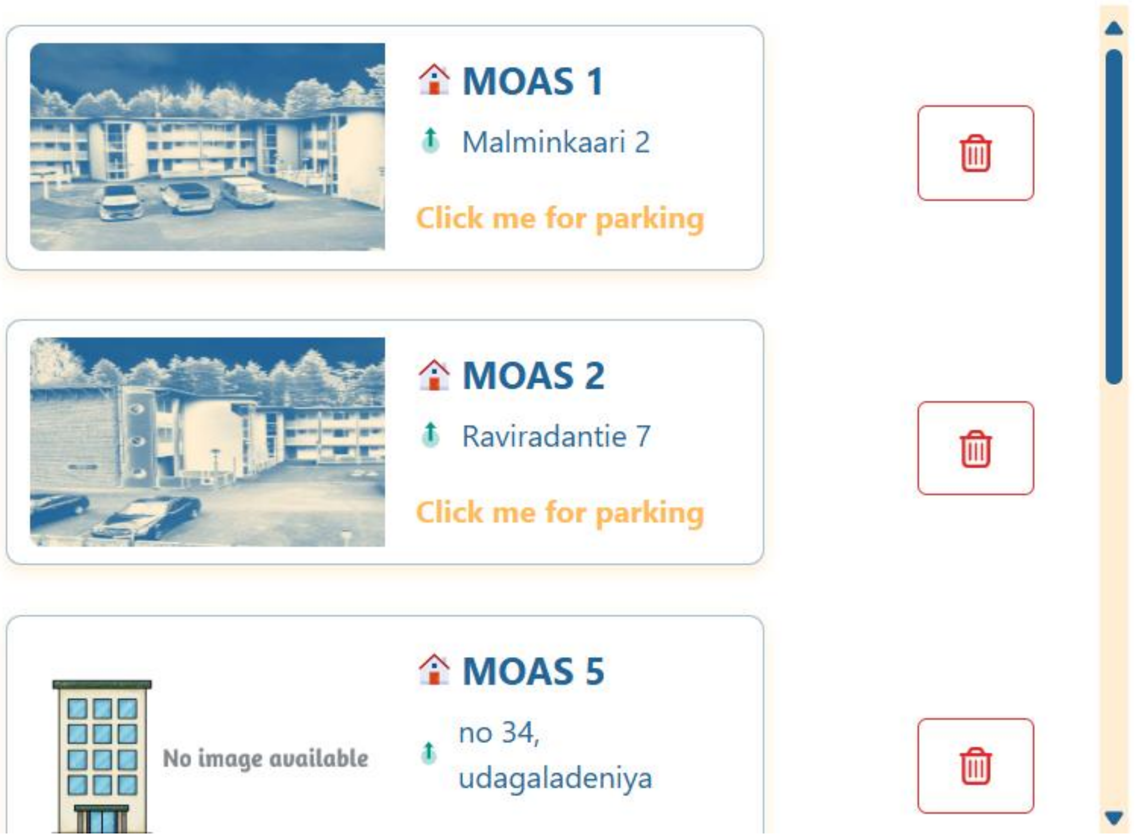
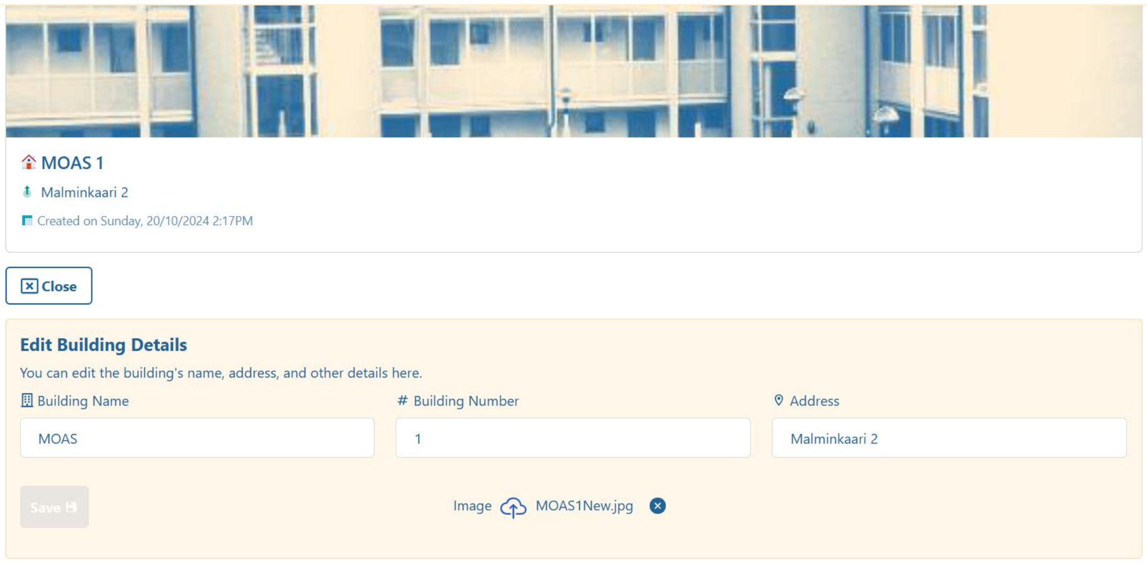


Figure 69. Building card view

After successfully creating a building, the admin can click the relevant building card, which directs them to the building edit section and the parking spot allocation page.


MOAS 1
Malminkaari 2
Created on Sunday, 20/10/2024 2:17PM

Close

Edit Building Details

You can edit the building's name, address, and other details here.

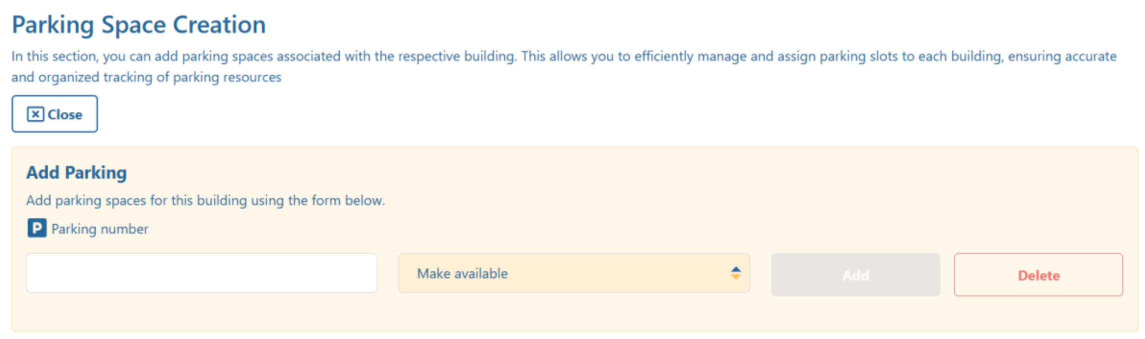
Building Name	# Building Number	Address
MOAS	1	Malminkaari 2

Figure 70. Building edit section

In this view, the building data entered in the initial phase can be edited through the form illustrate in Figure 70.

Create Parking

Scrolling through the building edit page, the parking spot creation form can be found as illustrate in Figure 71.



Parking Space Creation

In this section, you can add parking spaces associated with the respective building. This allows you to efficiently manage and assign parking slots to each building, ensuring accurate and organized tracking of parking resources

Add Parking

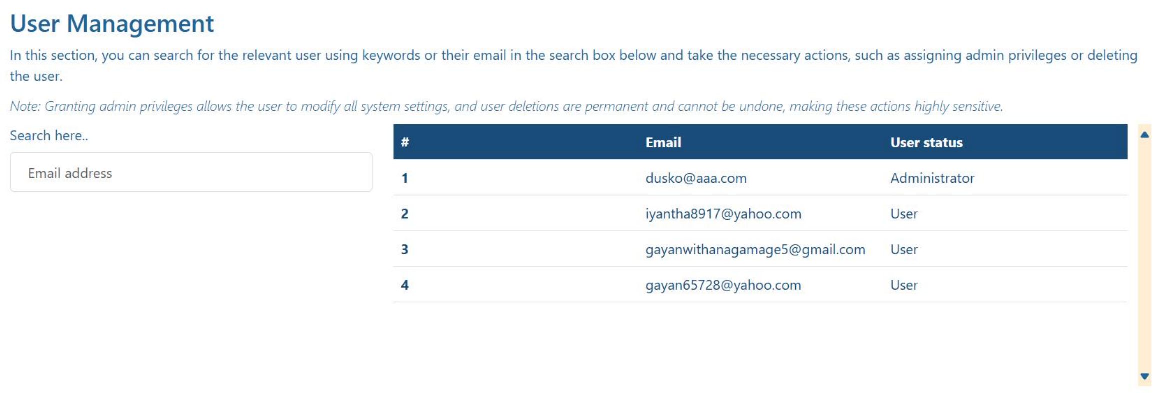
Add parking spaces for this building using the form below.

Figure 591. Parking spot creation form

In the parking spot creation view, parking spots can be created and edited using the same form. Additionally, the status of each parking spot can be manually updated.

User Management

The user management view can be found by clicking the “User Management” link in the top navigation bar in the administrator interface.



User Management

In this section, you can search for the relevant user using keywords or their email in the search box below and take the necessary actions, such as assigning admin privileges or deleting the user.

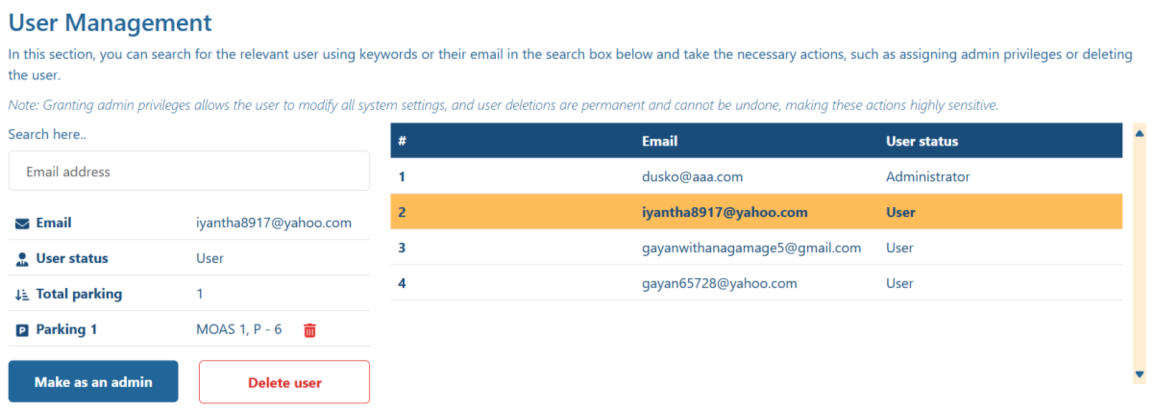
Note: Granting admin privileges allows the user to modify all system settings, and user deletions are permanent and cannot be undone, making these actions highly sensitive.

#	Email	User status
1	dusko@aaa.com	Administrator
2	iyantha8917@yahoo.com	User
3	gayanwithanagamage5@gmail.com	User
4	gayan65728@yahoo.com	User

Figure 602. User management interface

As illustrate in Figure 72, a search bar is provided to search users based on email keyword and the same is sorted in real time with the back-end API.

Once the user is selected, the relevant parking spot details, request information, and the user's building details are displayed simultaneously, as shown in Figure 73.



User Management

In this section, you can search for the relevant user using keywords or their email in the search box below and take the necessary actions, such as assigning admin privileges or deleting the user.

Note: Granting admin privileges allows the user to modify all system settings, and user deletions are permanent and cannot be undone, making these actions highly sensitive.

Search here..

Email address

Email	iyantha8917@yahoo.com
User status	User
Total parking	1
Parking 1	MOAS 1, P - 6

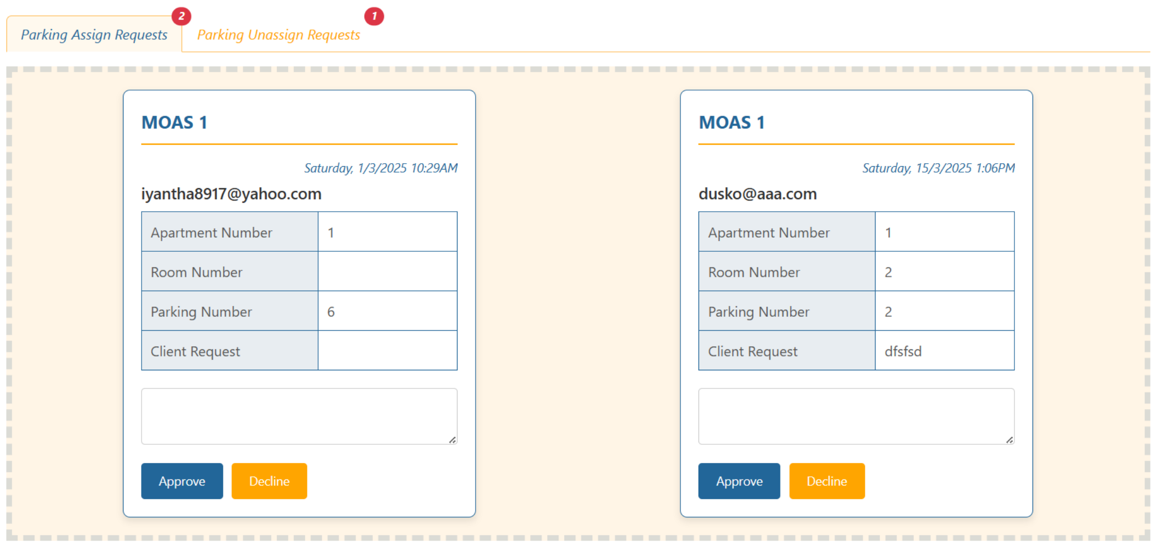
#	Email	User status
1	dusko@aaa.com	Administrator
2	iyantha8917@yahoo.com	User
3	gayanwithanagamage5@gmail.com	User
4	gayan65728@yahoo.com	User

Figure 613 User details

Furthermore, the relevant parking spot can be unallocated, and users can be deleted. Additionally, other users can be granted admin privileges in the absence of the current admin.

Task Management

Task management section can be found by clicking the "Task" link in the top navigation bar in the administrator's interface and the admin can approve or decline clients' parking requests as illustrates in Figure 74.



MOAS 1

Saturday, 1/3/2025 10:29AM

iyantha8917@yahoo.com

Apartment Number	1
Room Number	
Parking Number	6
Client Request	

MOAS 1

Saturday, 15/3/2025 1:06PM

dusko@aaa.com

Apartment Number	1
Room Number	2
Parking Number	2
Client Request	dfsfsd

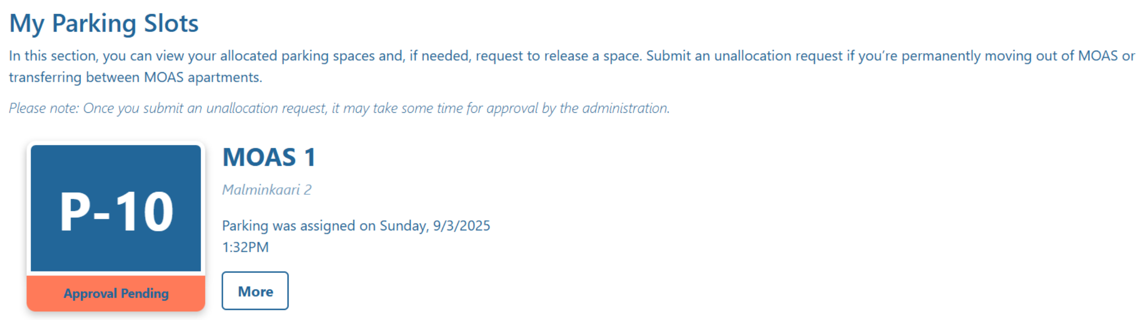
Figure 62. Task management interface

5.4 Real-Time Parking Availability / Other features

All the relevant states such as parking spots, buildings and users are maintained in a separate state and therefore all the parking spots are available in real time.

Tenant’s parking details

The status of the requested parking spot can be found via the link “My Parking” in the navigation bar as illustrates in Figure 75.



My Parking Slots

In this section, you can view your allocated parking spaces and, if needed, request to release a space. Submit an unallocation request if you're permanently moving out of MOAS or transferring between MOAS apartments.

Please note: Once you submit an unallocation request, it may take some time for approval by the administration.

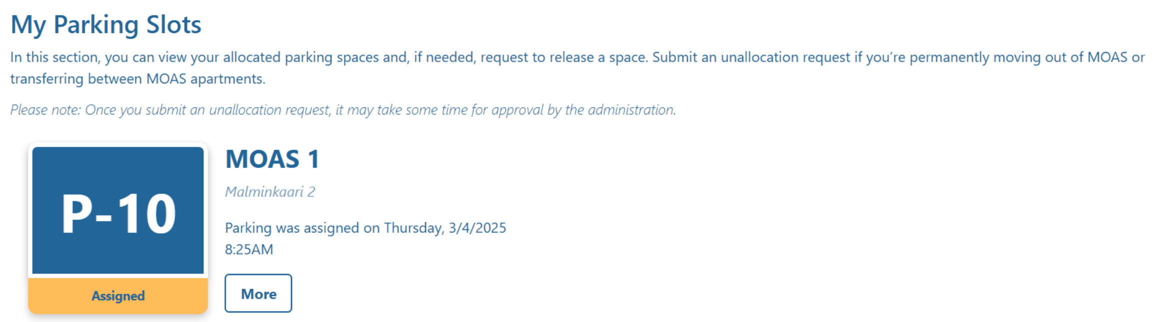
MOAS 1

Malminkaari 2

Parking was assigned on Sunday, 9/3/2025 1:32PM

Figure 635. Tenant's parking status

Once the request get approved the status changed to assigned status as illustrated iin Figure 76.

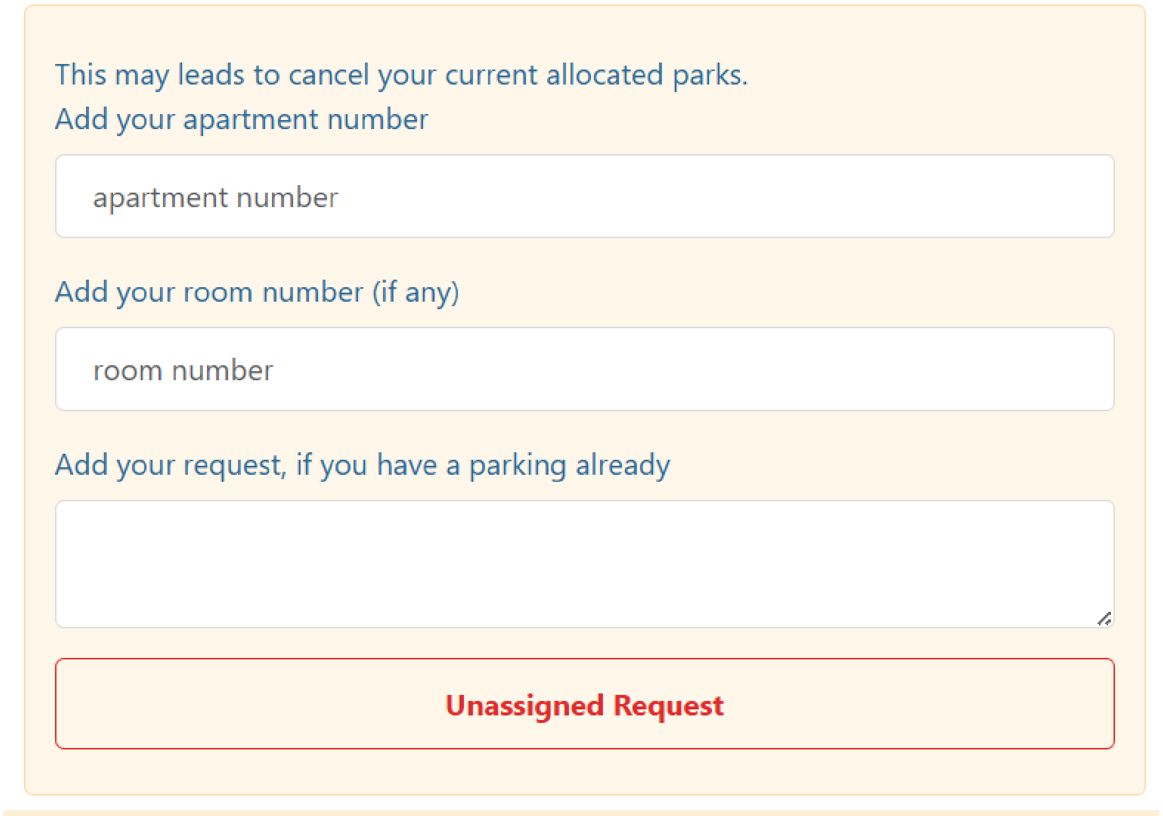

My Parking Slots

In this section, you can view your allocated parking spaces and, if needed, request to release a space. Submit an unallocation request if you're permanently moving out of MOAS or transferring between MOAS apartments.

Please note: Once you submit an unallocation request, it may take some time for approval by the administration.

Figure 646. Tenant's parking request get approved

In addition, users can request cancellation by clicking the “More” button, as shown in Figure 77.


This may leads to cancel your current allocated parks.

Add your apartment number

apartment number

Add your room number (if any)

room number

Add your request, if you have a parking already

Figure 657. Parking cancelation steps

Password changed

As shown in Figure 78, the email link directs the user to the change password page.

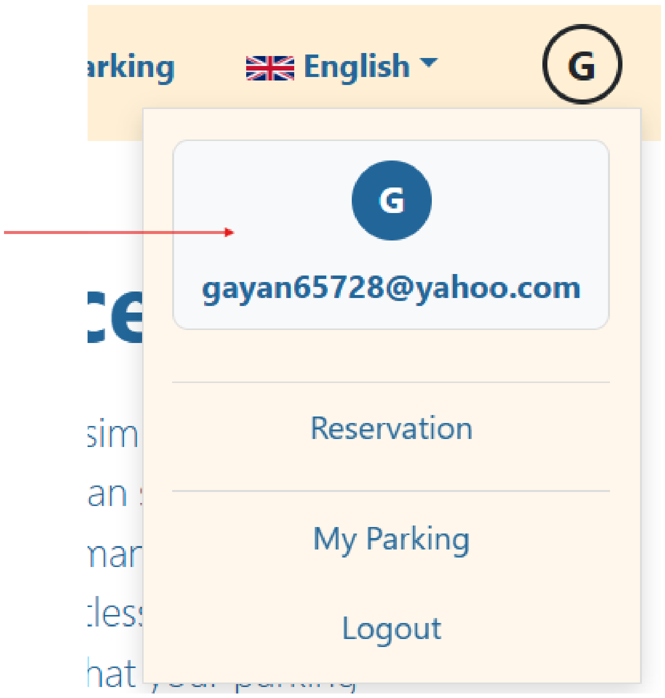
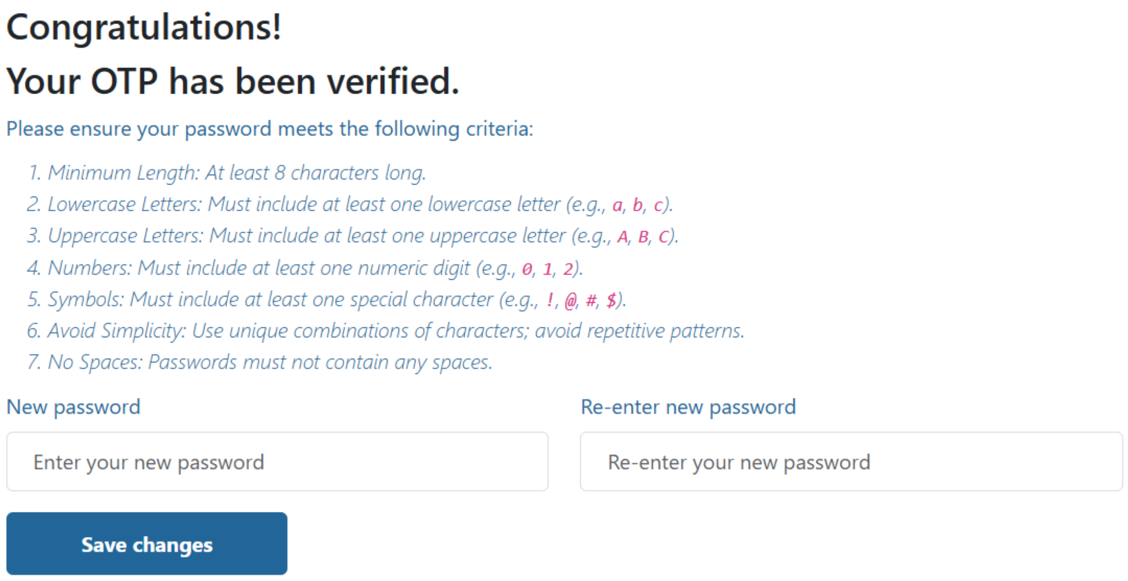


Figure 668. Email link

The user can change the password of the application via an OTP. After a successfully OTP verification, the user can change the password as shown in Figure 79.



Congratulations!
Your OTP has been verified.

Please ensure your password meets the following criteria:

1. *Minimum Length: At least 8 characters long.*
2. *Lowercase Letters: Must include at least one lowercase letter (e.g., a, b, c).*
3. *Uppercase Letters: Must include at least one uppercase letter (e.g., A, B, C).*
4. *Numbers: Must include at least one numeric digit (e.g., 0, 1, 2).*
5. *Symbols: Must include at least one special character (e.g., !, @, #, $).*
6. *Avoid Simplicity: Use unique combinations of characters; avoid repetitive patterns.*
7. *No Spaces: Passwords must not contain any spaces.*

New password: Enter your new password
Re-enter new password: Re-enter your new password

Figure 79 Password reset

Foregate Password

The “forget password” link can be found at the bottom of the login form as shown in Figure 80.

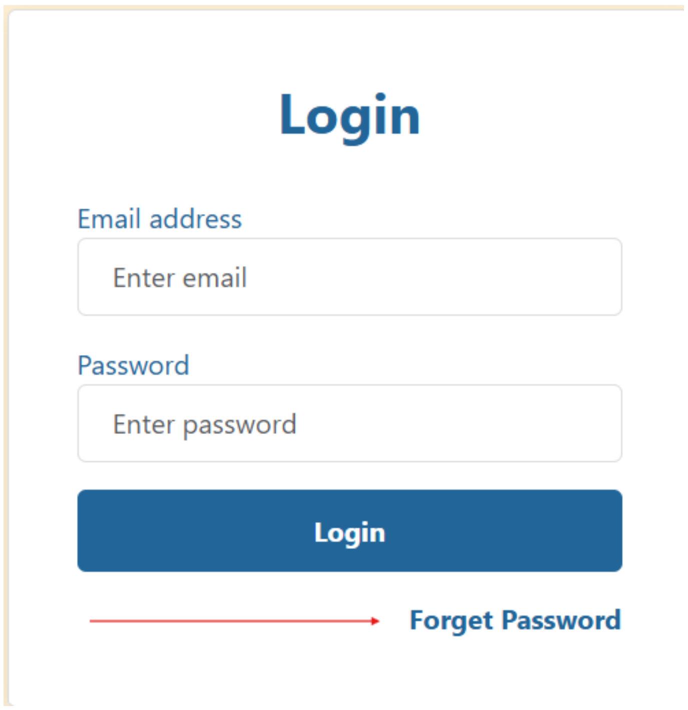
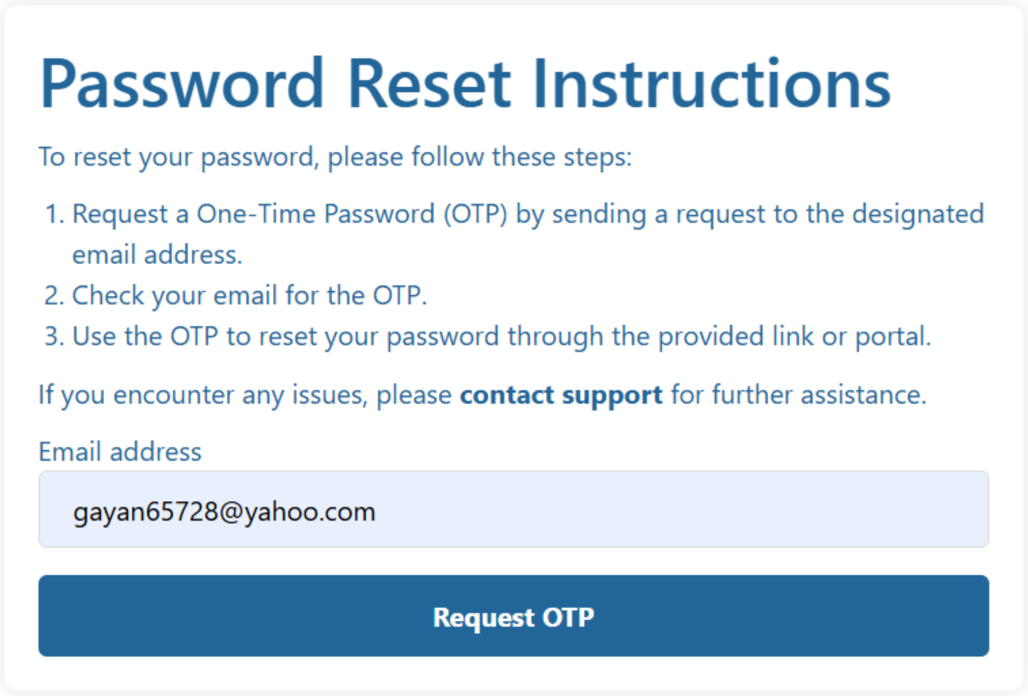


Figure 80. Forget password link

The ink directs the user password reset form and the password can be reset via requesting an OTP as shown in Figure 81.



Password Reset Instructions

To reset your password, please follow these steps:

1. Request a One-Time Password (OTP) by sending a request to the designated email address.
2. Check your email for the OTP.
3. Use the OTP to reset your password through the provided link or portal.

If you encounter any issues, please **contact support** for further assistance.

Email address

gayan65728@yahoo.com

Figure 671. Password reset form

6 DEPLOYMENT AND TESTING

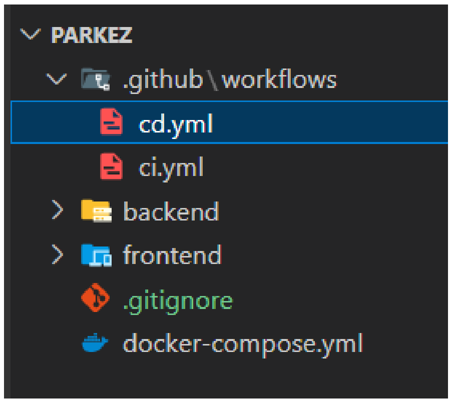
This section describes how development environment is set up, and how continuous integration (CI) and continuous deployment (CD) pipelines are implemented. Additionally, the testing methods used throughout the project are discussed.

6.1 Deployment environment

The ParkEz application is deployed using a cloud-based environment to promote accessibility and scalability. The deployment process utilizes various tools and platforms.

GitHub Actions

The GitHub Actions is used for automating the CI/CD pipeline. Upon each commit to the main code repository, the GitHub Actions build the application, run test, and create a Docker image (Patel, R 2024). Figure 82 illustrates the workflow folder structure, showcasing the organization of files and directories that define the flow and logic of the application's processes.

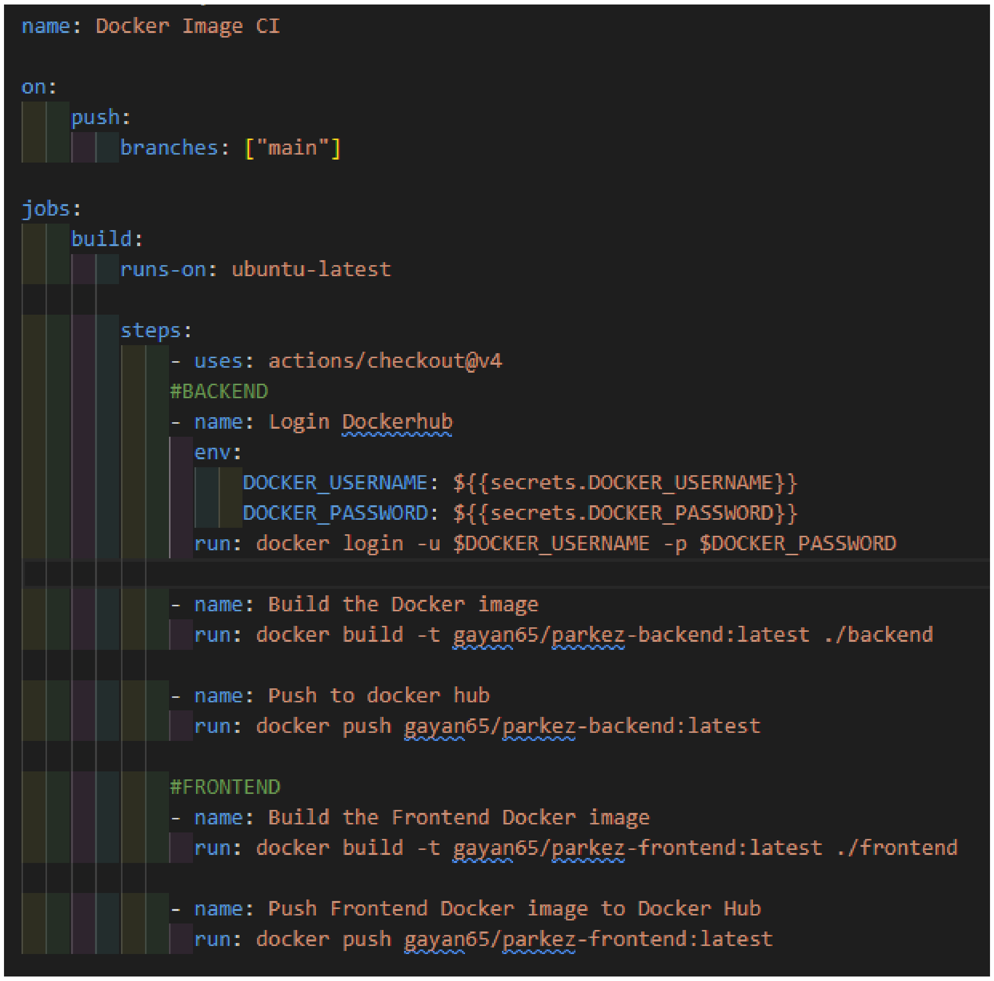


```
PARKEZ
  .github\workflows
    cd.yml
    ci.yml
  backend
  frontend
  .gitignore
  docker-compose.yml
```

Figure 682. Workflow folder structure

Two "yml" files have been used to automate the process and the continuous integration (CI) file as shown in Figure 83.

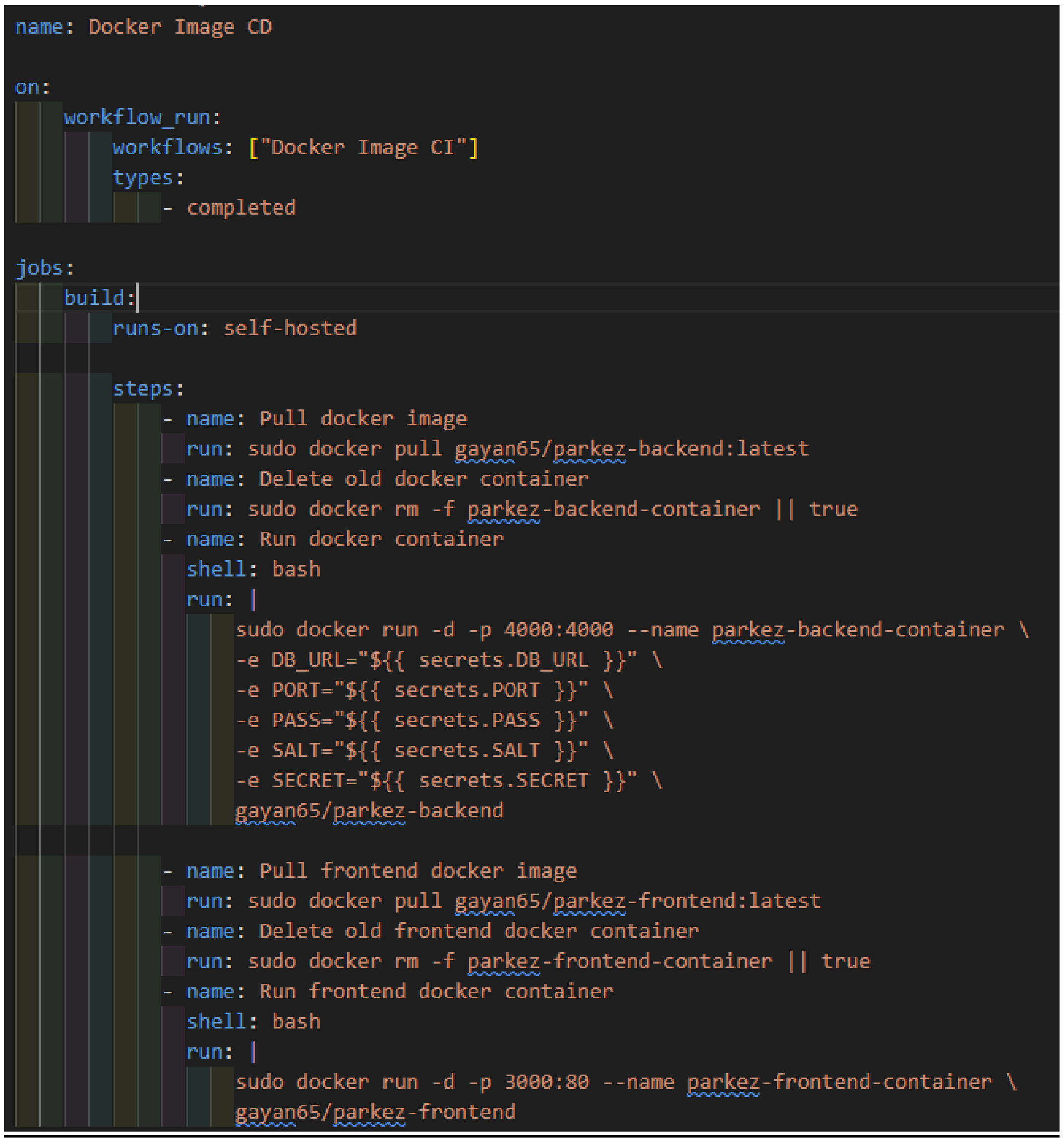
Figure 83 illustrates the CI (Continuous Integration) file, showcasing the configuration and setup for automated build and deployment processes within the application.



```yaml
name: Docker Image CI

on:
  push:
    branches: ["main"]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4
      #BACKEND
      - name: Login Dockerhub
        env:
          DOCKER_USERNAME: ${{secrets.DOCKER_USERNAME}}
          DOCKER_PASSWORD: ${{secrets.DOCKER_PASSWORD}}
        run: docker login -u $DOCKER_USERNAME -p $DOCKER_PASSWORD

      - name: Build the Docker image
        run: docker build -t gayan65/parkez-backend:latest ./backend

      - name: Push to docker hub
        run: docker push gayan65/parkez-backend:latest

      #FRONTEND
      - name: Build the Frontend Docker image
        run: docker build -t gayan65/parkez-frontend:latest ./frontend

      - name: Push Frontend Docker image to Docker Hub
        run: docker push gayan65/parkez-frontend:latest
```

Figure 693. CI file

Figure 84 illustrates the CD (Continuous Deployment) file, showcasing the configuration and setup for automating the deployment process in the application.

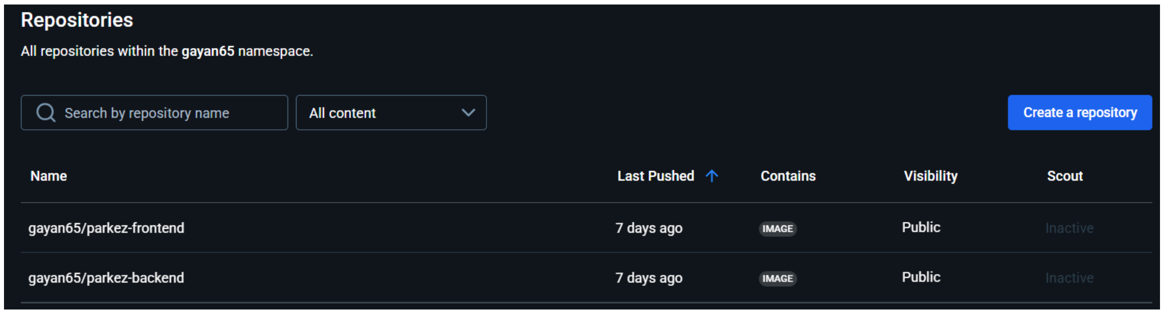


```yaml
name: Docker Image CD

on:
  workflow_run:
    workflows: ["Docker Image CI"]
    types:
      - completed

jobs:
  build:
    runs-on: self-hosted

    steps:
      - name: Pull docker image
        run: sudo docker pull gayan65/parkez-backend:latest
      - name: Delete old docker container
        run: sudo docker rm -f parkez-backend-container || true
      - name: Run docker container
        shell: bash
        run: |
          sudo docker run -d -p 4000:4000 --name parkez-backend-container \
          -e DB_URL="${{ secrets.DB_URL }}" \
          -e PORT="${{ secrets.PORT }}" \
          -e PASS="${{ secrets.PASS }}" \
          -e SALT="${{ secrets.SALT }}" \
          -e SECRET="${{ secrets.SECRET }}" \
          gayan65/parkez-backend

      - name: Pull frontend docker image
        run: sudo docker pull gayan65/parkez-frontend:latest
      - name: Delete old frontend docker container
        run: sudo docker rm -f parkez-frontend-container || true
      - name: Run frontend docker container
        shell: bash
        run: |
          sudo docker run -d -p 3000:80 --name parkez-frontend-container \
          gayan65/parkez-frontend
```

Figure 704. CD file

Docker Hub

The Docker Hub acts as the container registry where the Docker image is stored version-controlled after a successful build process, allowing it to be easily shared, deployed, and managed across different environments. (CI/CD Pipeline sinhalen with Docker Jenkins and GitHub | DevOps Projects Sinhala 2024).


Repositories

All repositories within the **gayan65** namespace.

Name	Last Pushed	Contains	Visibility	Scout
gayan65/parkez-frontend	7 days ago	IMAGE	Public	Inactive
gayan65/parkez-backend	7 days ago	IMAGE	Public	Inactive

Figure 715. Docker Images in Docker hub

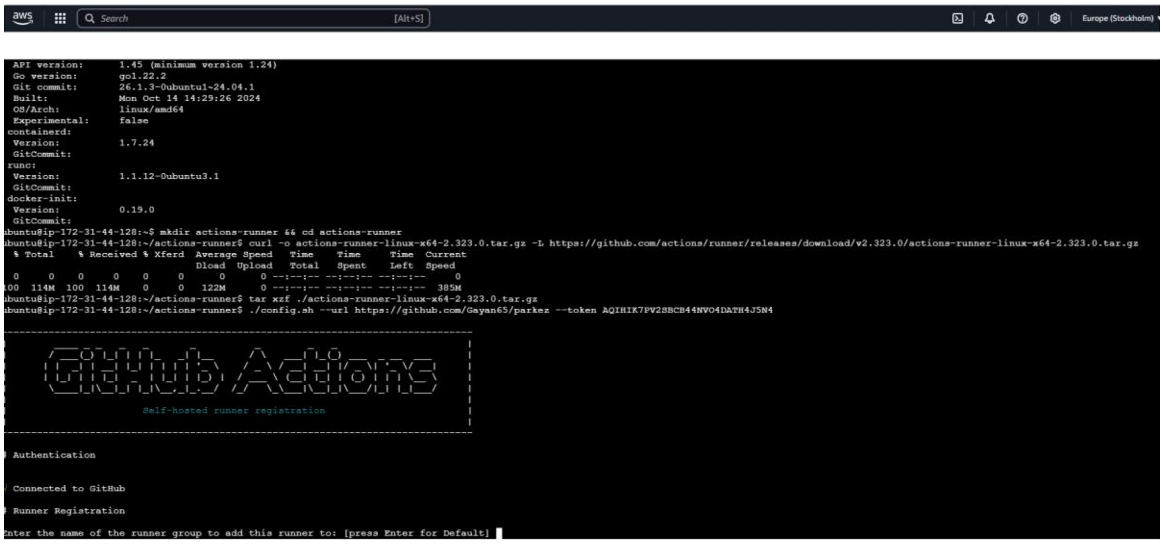
Both the frontend and backend Docker images are automatically built and pushed to Docker Hub whenever the CI workflow is triggered via GitHub Actions.

AWS EC2 Instance

The live application is hosted on an EC2 instance which pulls the latest image from Docker Hub and runs the container to make the ParkEz app available online. This setup ensures a smooth and automated deployment workflow, minimizing manual steps and reducing potential errors.

The EC2 instance was set up as an Ubuntu-based server, with Docker and Nginx installed to manage the deployment process efficiently. Docker enables the containerized execution of both the frontend and backend applications, while Nginx acts as a reverse proxy to route traffic between them. A custom Nginx configuration file was implemented to ensure seamless communication between the frontend and backend, optimizing performance and maintaining a stable connection (Vinay, N 2024).

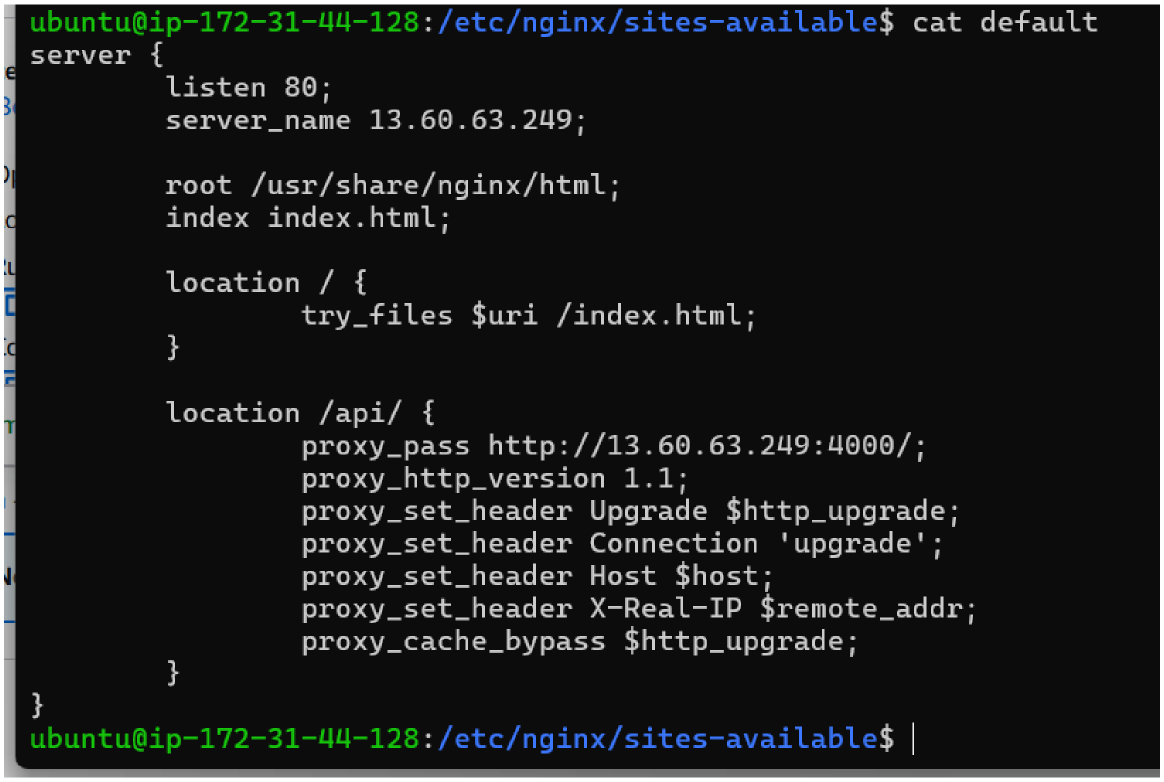
In order to automate deployment, GitHub Actions was used to trigger a CI/CD pipeline whenever changes were pushed to the repository. The workflow file builds both frontend and backend docker images, pushes them to Docker Hub, and remotely connects to the EC2 instance to pull and deploy the latest images. This should ensure that updates are automatically reflected in the live environment without manual intervention (Deploy MERN Stack App on AWS EC2 Using GitHub Actions 2024).



```
 API version:      1.45 (minimum version 1.24)
 Go version:       go1.22.2
 Git commit:       26.1.3-0ubuntu1~24.04.1
 Built:            Mon Oct 14 14:29:26 2024
 OS/Arch:          linux/amd64
 Experimental:     false
containerd:
 Version:          1.7.24
 GitCommit:
runc:
 Version:          1.1.12-0ubuntu3.1
 GitCommit:
docker-init:
 Version:          0.19.0
 GitCommit:
ubuntu@ip-172-31-44-128:~$ mkdir actions-runner && cd actions-runner
ubuntu@ip-172-31-44-128:~/actions-runner$ curl -o actions-runner-linux-x64-2.323.0.tar.gz -L https://github.com/actions/runner/releases/download/v2.323.0/actions-runner-linux-x64-2.323.0.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100  114M  100  114M    0     0   122M      0 --:--:-- --:--:-- --:--:--  385M
ubuntu@ip-172-31-44-128:~/actions-runner$ tar xzf ./actions-runner-linux-x64-2.323.0.tar.gz
ubuntu@ip-172-31-44-128:~/actions-runner$ ./config.sh --url https://github.com/Gayan65/parkez --token AQIHIK7PV2SBCB44NVO4DATH4J5N4

--------------------------------------------------------------------------------
|                                GitHub Actions                                |
|                                                                              |
|                       Self-hosted runner registration                        |
|                                                                              |
--------------------------------------------------------------------------------

# Authentication


√ Connected to GitHub

# Runner Registration

Enter the name of the runner group to add this runner to: [press Enter for Default]
```

Figure 726. Git hub actions setting up

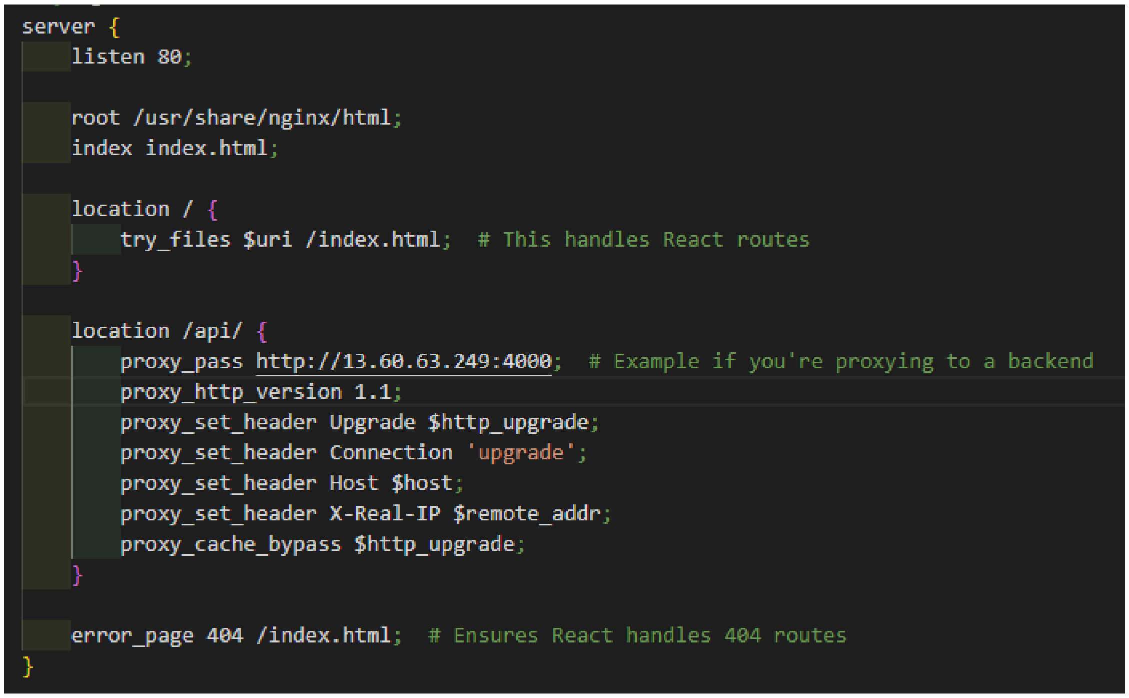
Figure 87 illustrates the default Nginx configuration, showcasing the setup for managing web server settings, including routing and handling of requests.



```
ubuntu@ip-172-31-44-128:/etc/nginx/sites-available$ cat default
server {
        listen 80;
        server_name 13.60.63.249;

        root /usr/share/nginx/html;
        index index.html;

        location / {
                try_files $uri /index.html;
        }

        location /api/ {
                proxy_pass http://13.60.63.249:4000/;
                proxy_http_version 1.1;
                proxy_set_header Upgrade $http_upgrade;
                proxy_set_header Connection 'upgrade';
                proxy_set_header Host $host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_cache_bypass $http_upgrade;
        }
}
ubuntu@ip-172-31-44-128:/etc/nginx/sites-available$
```

Figure 737. Default nginx configuration

In addition, a separate nginx configuration file has been uploaded to the frontend container. Figure 90 illustrates the custom Nginx configuration for the frontend, showcasing the tailored settings for serving the front-end application and managing its specific routes.



```
server {
    listen 80;

    root /usr/share/nginx/html;
    index index.html;

    location / {
        try_files $uri /index.html;  # This handles React routes
    }

    location /api/ {
        proxy_pass http://13.60.63.249:4000;  # Example if you're proxying to a backend
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_cache_bypass $http_upgrade;
    }

    error_page 404 /index.html;  # Ensures React handles 404 routes
}
```

Figure 748. Custom Nginx configuration for the frontend

6.2 Testing Methods and Results

Throughout the development lifecycle, a variety of testing techniques were used to guarantee the ParkEz application's functionality, performance, and dependability. These included unit testing for individual components, integration testing to ensure smooth interactions between modules, and end-to-end testing to validate real-world user workflows.

Unit testing

In order to verify specific features and modules, unit tests were created for both frontend and backend components. Jest and Supertest were used to test backend API endpoints, aiming to ensure correct request handling and data integrity. UI components for the frontend were verified using the React Testing Library.

Integrated testing

In order to ensure smooth communication between various elements, including the frontend, backend, and database, integration tests were conducted. Postman and Jest were used to verify API request-response processes, aiming to guarantee correct data retrieval and storage.

End-to-End testing

Cypress was used for E2E tests in order to replicate actual user interactions. In order to promote a seamless user experience, the tests addressed essential features such as admin approval workflows, parking space bookings, and user identification.

Security testing

In order to find vulnerabilities, basic security evaluations were carried out, such as input validation and authentication checks. In order to reduce typical vulnerabilities such as SQL injection and cross-site scripting (XSS), OWASP security principles were followed.

6.3 User Feedback

After the ParkEz application was deployed, user input was gathered to assess its usability and efficacy. Surveys, issue reports, and direct user interviews were used to collect feedback.

Positive feedback

Users valued the user-friendly UI/UX which made parking reservations simple and quick. The automatic approval mechanism expedited the booking process for both parties.

Areas for improvement

Some users recommended adding more filters and sorting options to improve the process of choosing a parking space. Performance issues were observed on low-bandwidth connections, indicating a need for optimization. While the app generally responded well on mobile devices, slight adjustments are necessary for smaller screens to enhance usability.

Data driven insights

The recommended updates include a number of improvements based on the user feedback, such as improved user interface, performance advancements, and more filtering choices. Based on continuous user feedback, the system will continue to be improved with regular upgrades and issue corrections.

7 RESULTS AND EVALUATION

The ParkEz application has positively impacted MOAS Oy’s management and end customers by enhancing the parking management process and user experience. For management, it has streamlined monitoring and allocation of parking spaces, while end users benefit from a more intuitive interface and faster, more reliable access to parking services.

7.1 Benefits to MOAS Oy

ParkEz has simplified parking operations by replacing manual processes with an automated system, saving time and reducing paperwork. The web application enables real-time monitoring and management of parking spaces, allowing staff to promptly address issues and maintain smooth operations. The automation of routine tasks has lowered operational costs and improved accuracy, while data collection on parking usage supports informed decisions about facility improvements and expansion.

7.2 User Experience Improvements

For tenants and administrators, ParkEz offers a user-friendly online reservation system that reduces wait times and eliminates the need for physical requests. Real-time availability indicators help users quickly find and book parking spots. The responsive design ensures easy access across devices such as smartphones and tablets. Additionally, automated notifications keep users informed about booking statuses and reminders, increasing transparency and satisfaction.

7.3 Administrative Efficiency Gains

The system enhances administrative workflows with a centralized dashboard that provides quick access to tenants’ parking history and requests. This allows administrators to efficiently approve or decline bookings, reducing processing time. Centralized data management improves organization and traceability while automation minimizes human errors in record-keeping and slot assignments.

Designed for scalability, ParkEz can support a growing user base and number of parking areas without increasing administrative workload.

8 FUTURE ENHANCEMENTS

Several improvements are planned to expand ParkEz’s functionality, enhance user convenience, and leverage emerging technologies for smarter parking management.

8.1 Payment Integration

A secure online payment system will be added, allowing users to pay for reservations directly within the app. This will increase convenience by supporting multiple payment methods such as credit/debit cards, mobile wallets, and bank transfers. Automated receipts and payment confirmations will provide transparency and assurance for every transaction.

8.2 Scalability for Other Housing Firms

ParkEz is designed for scalability beyond MOAS Oy. Future upgrades will introduce a multi-tenancy architecture to support multiple organizations with secure data isolation. Customizable branding and policies will allow each housing firm to tailor the platform to their needs. Additionally, flexible administrative roles and permissions will enable efficient management across diverse organizational structures.

8.3 Smart Parking Solutions

In order to align with smart city initiatives, ParkEz will integrate advanced technologies including IoT sensors for real-time parking spot detection and license plate recognition (LPR) for automatic vehicle access control. AI-driven analytics will also be implemented to forecast parking demand, optimize space usage, and detect anomalies, enabling more intelligent, data-driven management.

9 CONCLUSION

In this section, the main outcomes of the thesis are summarized, the key contributions are highlighted, and recommendations for future enhancements are outlined.

9.1 Summary of Findings

The ParkEz application effectively resolved MOAS Oy’s parking management challenges by automating reservation and approval workflows, improving communication between users and administrators, and providing real-time visibility of parking availability. The system reduced manual workload and errors while delivering a user-friendly experience that met all functional requirements. Feedback and testing confirmed increased operational efficiency for the organization.

9.2 Contributions of the Thesis

The thesis contributed both technically and practically through the development of a full-stack web application featuring a responsive frontend and scalable backend. The design emphasized usability, performance, and maintainability.

A significant technical achievement was the implementation of a continuous integration and deployment (CI/CD) pipeline using GitHub Actions, Docker, and AWS EC2. This automated deployment process ensures efficient and reliable delivery of updates.

Comprehensive documentation was also produced, offering valuable guidance for future developers or researchers interested in similar systems or further development.

9.3 Recommendations for Future Work

While ParkEz meets its primary objective, several enhancements are planned to expand its capabilities. These include integrating online payment processing to automate fee collection, adapting the platform for use by other housing organizations with customizable features, and incorporating smart technologies such as IoT sensors and AI-driven analytics to advance toward a more intelligent, automated parking management system.

REFERENCES

bcrypt, 2023. bcrypt – npm. Web page. Available at: https://www.npmjs.com/package/bcrypt [Accessed 16 October 2024].

CI/CD Pipeline sinhalen with Docker Jenkins and GitHub | DevOps Projects Sinhala. 2024. Adomic Arts. Video clip. Youtube. Available at: https://www.youtube.com/watch?v=XgbVYGX0Pls [26 October 2024].

Deploy MERN Stack App On AWS EC2 Using GitHub Actions. 2024. Dhanian Tech industry. Video clip. Youtube. Available at: https://www.youtube.com/watch?v=_SG9xQkKiug&t=1405s [20 December 2024].

Express.js, 2024. Express - Node.js web application framework. Web page. Available at: https://expressjs.com/ [Accessed 26 October 2024].

Holovaty, A. & Kaplan-Moss, J. 2009. The definitive guide to Django: Web development done right. 2nd ed. New York: Apress.

Jansen, S. 2014. Building scalable web sites with the LAMP stack. Sebastopol: O'Reilly Media.

Kalehewatte, D. 2024. Implementing an Application with a Secure Authentication System in the MERN Stack. Web page. Available at: https://medium.com/@dushyanthak/implementing-an-application-with-a-secure-authentication-system-in-the-mern-stack-a7e033d7439f [Accessed 20 February 2025].

Kumar, S. 2019. Full-stack React projects: Modern web development using React 16, Node, Express, and MongoDB. Birmingham: Packt Publishing.

MERN Authentication Tutorial. 2022. Net Ninja. Video clip. Youtube. Available at: https://www.youtube.com/watch?v=WsRBmwNkv3Q&list=PL4cUxeGkcC9g8OhpOZxNdhXggFz2lOuCT [27 September 2024].

MERN Stack vs Django: Which is Better for Web Development 2024?. 2024. NxtWave. Web page. Available at: https://www.ccbp.in/blog/articles/mern-stack-vs-django [Accessed 06 May 2025].

Meta. 2025. React documentation. Menlo Park: Meta. Available at: https://react.dev. [Accessed 25 October 2024].

Mikalo Oy. 2024. About Us. Web page. Available at: https://mikalo.fi/en/about-us/ [Accessed 25 October 2024].

Mongoose, 2024. Mongoose elegant mongodb object modeling for Node.js. Web page Available at: https://mongoosejs.com/ [Accessed 26 October 2024].

Muneer, F. 2025. Building a Full-Stack Authentication System with MERN. Web page. Available at: https://medium.com/@fizamuneer0101/building-a-full-stack-authentication-system-with-mern-6e1b1158031e [Accessed 6 April 2025].

Nodemailer. n.d. Nodemailer. Web page. Available at: https://nodemailer.com/ [Accessed 22 October 2024].

Patel, R. 2024. Implementing CI/CD Pipeline for a MERN Stack Application Using GitHub Actions with Automatic Deployment to a Local Server(VM with Public IP). Web page. Available at: https://medium.com/@ravipatel.it/implementing-ci-cd-pipeline-for-a-mern-stack-application-using-github-actions-with-automatic-9080459173bf [Accessed 16 February 2025].

Stojchevska, F. 2025. 8 Benefits of ReactJS. Available at: https://www.designrush.com/agency/web-development-companies/reactjs/trends/benefits-of-reactjs [Accessed 21 February 2025].

Vinay, N 2024. Streamlining CI/CD for a MERN Application with Docker-Compose on EC2 via GitHub Actions: Achieving Zero Downtime Deployment. Web page. Available at: https://medium.com/@techievinay01/streamlining-ci-cd-for-a-mern-application-with-docker-compose-on-ec2-via-github-actions-achieving-75671d5df23c [Accessed 18 February 2025].

jsonwebtoken, 2023. jsonwebtoken - npm. Web page Available at: https://www.npmjs.com/package/jsonwebtoken [Accessed 20 October 2024].