



# Parkour - Parking Assistant

**Project Supervisor**

Engr. Abdul Rahman

**Project Team**

Mahad Khalid K180187

Abdullah Raheel K180170

Ammar Nasir K181037

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science

FAST SCHOOL OF COMPUTING  
NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES  
KARACHI CAMPUS  
July 2021

Project Supervisor	<b>Engr. Abdul Rahman</b>
Project Team	Abdullah Raheel K180170 Mahad Khalid K180187 Ammar Nasir K181037
Submission Date	13 July 2022

**Engr. Abdul Rahman** \_\_\_\_\_  
Supervisor

**Dr. Zulfiqar Ali Memon** \_\_\_\_\_  
Head of Department

FAST SCHOOL OF COMPUTING  
NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES  
KARACHI CAMPUS  
July 2021

# Acknowledgements

Throughout these 8 months we have recieved tremendous support from our Supervisor **Engr. Abdul Rahman**. From the start **Engr. Abdul Rahman** helped groom and shape our idea to the best of his ability. He has also provided us with the resources to learn and to grow. We would like to thank **Engr. Abdul Rahman** for his time and effort.

We are immensely thankful to **Sir. Sayed Yousaf** for his guidance not only as a great teacher but also as a career counselor. He has been a great mentor to us, giving us the guidance we needed in many areas of life.

We would also like to thank **Sir. Danish** and **Sir. Zeeshan** for the materials they provided to us for FYP and also helping us with any FYP related queries.

We would like to thank some of the best Teachers in the University for their guidance and grooming us to people we are today. Some of the teachers are but not limited to: **Ms. Anum Qureshi**, **Ms. Farah Sadia**, **Ms. Nida Fatima**, **Dr. Zulfiqar Ali Memon** and **Dr. Jawwad Shamsi**.

I would also like to thank my Project Team for their unyielding grit in the face of difficulties. Their hard work and dedication has helped us to achieve our goals.

# Abstract

In small to large metropolitan cities alike, finding parking is a principal problem. Over population of major cities in Pakistan means high density of vehicles which the city cannot accommodate to begin with. This leads to lack of parking spots which in turn make parkers spend their valuable time to find a spot, increase their fuel consumption and contribute to traffic congestion.

To tackle this problem, we came up with the idea of an app with an inbuilt map that shows parking spots all around the city to help drivers find parking spots quickly, saving time and fuel. The app will also show open parking spots close to drivers where they can easily park and also show the route to these spots. Drivers can also see how much fuel it will take to reach said spot. Since the available parking spots are visible on the map, drivers can even pre-book the spots beforehand.

The project aims to provide parkers an application to find parking spots using their phone in real time and save them the trouble of looking for a parking spot manually. Meanwhile, the app gives the parking provider an opportunity to run a small business by putting their unused parking spot(s) up for sale. Furthermore, the project aims to provide residents who own a car but do not have a suitable parking spot near their home or residents who have trouble finding a parking spot near their workplace, a long-term parking solution by giving them the option to purchase a parking spot nearby.

# Contents

<b>1</b>	<b>Introduction</b>	<b>8</b>
1.1	Background . . . . .	8
1.2	Problem . . . . .	8
1.3	Proposed System . . . . .	9
1.4	Pricing . . . . .	9
1.5	Conclusion . . . . .	9
<b>2</b>	<b>Related Work</b>	<b>10</b>
<b>3</b>	<b>Requirements</b>	<b>11</b>
<b>4</b>	<b>Design</b>	<b>12</b>
4.1	Design Patterns . . . . .	12
4.2	Frontend . . . . .	14
4.2.1	Wireframe . . . . .	14
4.2.2	UI . . . . .	15
4.3	Backend . . . . .	17
4.4	Architecture . . . . .	19
4.4.1	Interface . . . . .	19
4.4.2	Server . . . . .	20
4.4.3	Client . . . . .	21
4.4.4	Application . . . . .	21
<b>5</b>	<b>Implementation</b>	<b>22</b>
5.1	Frontend . . . . .	22
5.1.1	API Key . . . . .	22
5.1.2	Store . . . . .	23
5.1.3	Store Slice . . . . .	23
5.1.4	Routing . . . . .	24
5.1.5	Firebase and Authentication . . . . .	25
5.1.6	Structure . . . . .	26
5.1.7	Configuring Google Map . . . . .	27
5.2	Backend . . . . .	28
5.2.1	Database Connection . . . . .	28

5.2.2	API Routes . . . . .	29
5.2.3	Sockets . . . . .	30
5.2.4	Chat Server . . . . .	30
5.2.5	Routing . . . . .	31
5.2.6	Middleware . . . . .	32
5.2.7	Controllers . . . . .	33
5.3	Database . . . . .	34
5.3.1	Models . . . . .	34
5.4	Behavior . . . . .	43
5.4.1	Parker . . . . .	43
5.4.2	Seller . . . . .	45
5.4.3	Spot . . . . .	47
5.4.4	Booking Request . . . . .	48
<b>6</b>	<b>Testing and Evaluation</b>	<b>49</b>
<b>7</b>	<b>Conclusion</b>	<b>50</b>

# List of Figures

4.1	State Pattern . . . . .	12
4.2	Memento Pattern . . . . .	13
4.3	Homepage Wireframe . . . . .	14
4.4	Parker Wireframe . . . . .	14
4.5	Seller Wireframe . . . . .	14
4.6	Color, Font, Design Schemes . . . . .	15
4.7	UI (1) . . . . .	15
4.8	UI (2) . . . . .	16
4.9	UI (3) . . . . .	16
4.10	Database Design . . . . .	17
4.11	Interface . . . . .	19
4.12	Server . . . . .	20
4.13	Client . . . . .	21
4.14	Application Architecture . . . . .	21
5.1	API Key . . . . .	22
5.2	Store Provider . . . . .	23
5.3	User Slice in Store . . . . .	23
5.4	Routes . . . . .	24
5.5	Firebase Config . . . . .	25
5.6	Folder Structure in React . . . . .	26
5.7	Google Map Configuration . . . . .	27
5.8	Database Connection . . . . .	28
5.9	API Routes . . . . .	29
5.10	Socket Connection . . . . .	30
5.11	Chat Server . . . . .	30
5.12	User Routes . . . . .	31
5.13	isAuth Middleware . . . . .	32
5.14	User Controller . . . . .	33
5.15	User Model . . . . .	34
5.16	Parker Model . . . . .	35
5.17	Seller Model . . . . .	36
5.18	Car Model . . . . .	37
5.19	Spots Model . . . . .	38

5.20 Booking Request Model . . . . .	39
5.21 Notification Model . . . . .	40
5.22 Message Model . . . . .	41
5.23 Chat Model . . . . .	42
5.24 Parker Behavior (1) . . . . .	43
5.25 Parker Behavior (2) . . . . .	44
5.26 Seller Behavior (1) . . . . .	45
5.27 Seller Behavior (2) . . . . .	46
5.28 Spot Behavior . . . . .	47
5.29 Booking Request Behavior . . . . .	48



# List of Tables

4.1	Models . . . . .	18
5.1	Technologies Used . . . . .	22

# Introduction

## 1.1 Background

Commuting is a daily part of most people's lives and for many people who own a private vehicle finding a suitable and appropriate parking spot is a daily struggle. It becomes even more difficult to find a spot when hundreds of people come out for an event or holiday festivals. The fact that no private vehicle is perpetually in motion; most private vehicles spend most of their time at rest, either during working hours or over the night, means that there should be two places for every car in the city to be parked in. The two places should be at the both ends of every trip. However the number of vehicles keeps increasing, while parking space has remained constant or reduced due to a growing population. This imbalance between parking supply and parking demand has been considered as the main reason for metropolis parking problems.

## 1.2 Problem

Lack of parking spots truly is a global issue. As global living standards rise and urbanization accelerates, especially in India and China, cities around the world are seeing huge spikes in motor vehicle ownership accompanied by demand for parking. In India, the number of private automobiles grew nearly 400% between 2001 and 2015, going from 55 million to 210 million. In China, as of 2017, there was a shortage of 50 million parking spaces, according to the central government. Cities face immense challenges from climate change and rising heat, increased urbanization and housing affordability.

Besides the lack of parking spots, there is another problem that needs to be addressed. Surprisingly, there are metropolitan cities which incorporate plenty of parking spaces. For example, one study shows that there are 2.2 million total parking spaces in Philadelphia and 1.85 million in New York City. So the problem is not just about the general lack of parking spaces but the lack of real-time data about vacant ones. Due to this, drivers are often left frustrated and spend too much time searching for a spot. In peak hours drivers in large cities have to circle around the desired destination to find a place where they can leave their vehicles. Those who run out of time might park illegally. By bouncing between parking spaces that are full, drivers become restless and may choose to park illegally or leave altogether.

An unregulated tariff structure is another issue which leads to a scarcity of parking spaces. For example, in Indian and Pakistani metros, parking is either free or minimally priced, the fees being unregulated for many years now. Because parking prices stop increasing after a certain period of time, the longer one stays in a parking space, the less one has to pay. This is a problem because parking space is a scarce commodity today and should come with a price. A low parking price encourages more vehicles on the road, contributing to air and noise pollution.

### 1.3 Proposed System

The project aims to solve all the problems addressed above by attempting to create a self-managing peer to peer parking system where people who have unused parking spots can both earn and contribute by making their spots available for others to park and charge a fare for the spot. The system will embody a map that provides reliable, real-time data that allows drivers to choose from the available spots or even pre-book a parking spot beforehand.

### 1.4 Pricing

The system will further implement a regulated tariff system. One could say the best way to manage the parking is by charging the right price for it. This can be done by using demand to price parking and optimize occupancy. If the price is too high and spaces remain vacant, operators lose revenue, nearby shops lose customers, employees lose jobs, and governments lose tax revenue. If the price is too low and no spaces are available, it leads to traffic congestion and chaos. Pricing can thus be a very effective tool for the management of travel demand as a whole. Deploying a cost-effective parking management and guidance solution ultimately generates more revenue for a city, as existing parking spaces are properly monetized. Drivers are more motivated to pay for a spot when they know they'll be able to find it quickly, without having to circle around in vain. The awareness by drivers that all spaces are monitored by a modern system further increases the understanding that it is fair to pay for the valuable public space and service.

### 1.5 Conclusion

The growth of private vehicles is not going to halt and in turn the demand for parking will only increase. Solutions for parking need to be developed to address this growing pain. Parkour is a step in this direction, where we connect people looking for parking, and people who have vacant spots. This report details the design of a system that will connect people looking for parking with people who have vacant spots.

# Related Work

To provide parking spots it is absolutely necessary to rely on either people or sensors. If the application is dependent on people it must either implement a peer to peer system like **Parkour** or use a smart phone to provide the parking spots. If the application is dependent on sensors, it must be able to provide the parking spots without the need of people.

The other method of determined the parking spots is to use a combination of sensors and people. This is called **Sensor-based Parking** [1] This type of methodology uses both the sensor and data from people to accurately predict of the spots are empty.

Parking Spots can also be determined using data from the general public. This is called **Crowd-Sourced Parking** [2] This type of methodology uses people to verify if the parking spots are indeed empty.

One unique method of determining if parking spots were empty or occupied was through the use of ultrasonic sensors [3]. This method was found to be the most accurate and efficient method of determining if parking spots were empty or occupied without entailing the heavy cost of image recognition based parking.



The most effective method of determining the parking without relying on people is by using some sort of image recognition. This is called **Image-Detection-based Parking** [4] This is not only very expensive to implement but also comes with it's challenges of accurate image detection.

**Parkour** uses peer to peer communication to determine the parking spots. This is a peer to peer system that is implemented using a smart phone. This is a system that is designed to be used by people to communicate with each other.

# Requirements

**Parkour - Parking Assistant** is an Android / Web based application that requires an active internet connection. The application also requires a GPS to be able to track the user's location. The location must be enabled on the device for the application to work properly.

Services required on the phone:

-  The application requires an active internet connection, To connect to google maps and google services. The application requires an active internet connection to download the map data. without an active internet connection, the application will not be able to talk to the server.
-  Gps / Location services must be enabled on the device to detect the current location of the user and show them parking spots near them in a certain radius.

# Design

A significant amount of time was spent on the design of the system. A careful consideration was given to working of the frontend and the backend. The system was designed to be as modular as possible. The system was designed to be able to be used in multiple ways. Special attention to the design of the frontend was given including contrast ratios, colors and rhythm was given. User experience was also given a high priority.

## 4.1 Design Patterns

- State

The useState hook in react exposes a way to change the state of the component, this allows us to re-render the ui whenever change the state of the component. The functional component might behave differently depending on the state of the component. This is how a the state pattern is used here.

```
const [socket, setSocket] = useState<Socket>(null);  
const [receiver, setReceiver] = useState(null);  
const [messages, setMessages] = useState([]);
```

Figure 4.1: State Pattern

- Memento

The useMemo hook allows us to memoize the output of the component. This is how the memento pattern is used here. Here we are memoizing the slotList. This is done to avoid re-calculating and re-sorting the slotList when the component re-renders.

```
// Sorting the slotList
const sortedSpotList = useMemo(() => {
  const list = [...slotList];

  if (list.length === 0) {
    return [];
  }

  // Sort by Date
  list.sort((availability1, availability2) =>
    availability1.slotDate > availability2.slotDate ? 1 : -1
  );

  // Sort time slots of each Date
  list.forEach((availability) => {
    availability.slots.sort((ts1, ts2) => {
      // First compare by start time
      if (ts1.startTime > ts2.startTime) {
        return 1;
      } else if (ts1.startTime < ts2.startTime) {
        return -1;
      }

      // Else compare by endTime
      if (ts1.endTime < ts2.endTime) {
        return -1;
      } else if (ts1.endTime > ts2.endTime) {
        return 1;
      } else {
        // nothing to split them
        return 0;
      }
    });
  });
});
```

Figure 4.2: Memento Pattern

## 4.2 Frontend

### 4.2.1 Wireframe

Before the development of the UI the design of the wireframe was considered. A wireframe of the UI was created that gave the general understanding of how the UI will be broken into components.

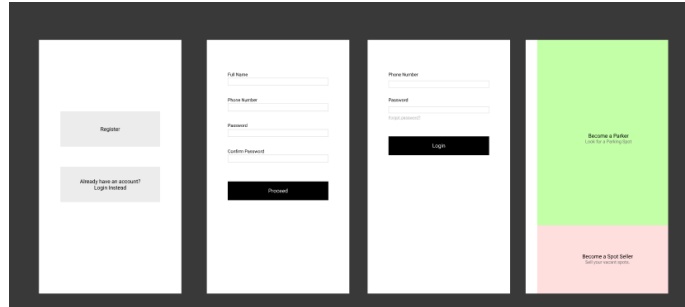


Figure 4.3: Homepage Wireframe

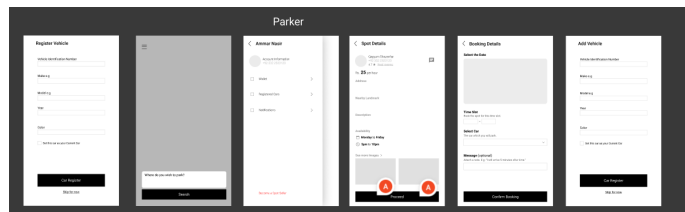


Figure 4.4: Parker Wireframe

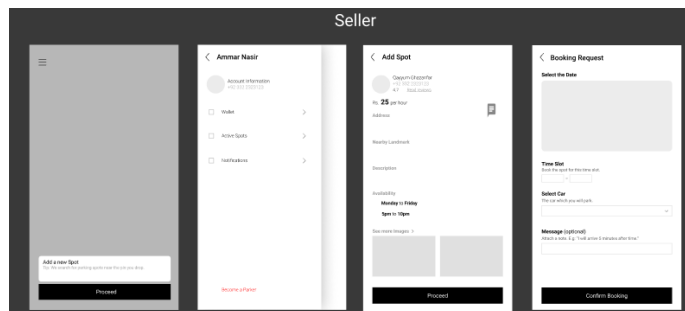


Figure 4.5: Seller Wireframe



## 4.2.2 UI

At the start of UI design the Font, Colors and Design schemes were solidified to ease the process and keep the consistency throughout the application. The aim was to make the UI look and feel like one cohesive application.

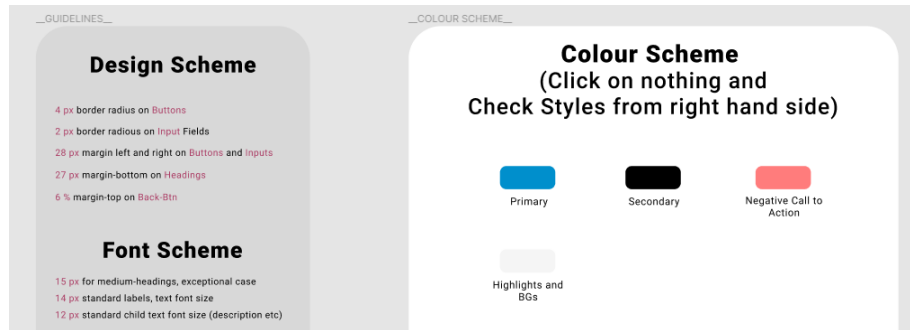


Figure 4.6: Color, Font, Design Schemes

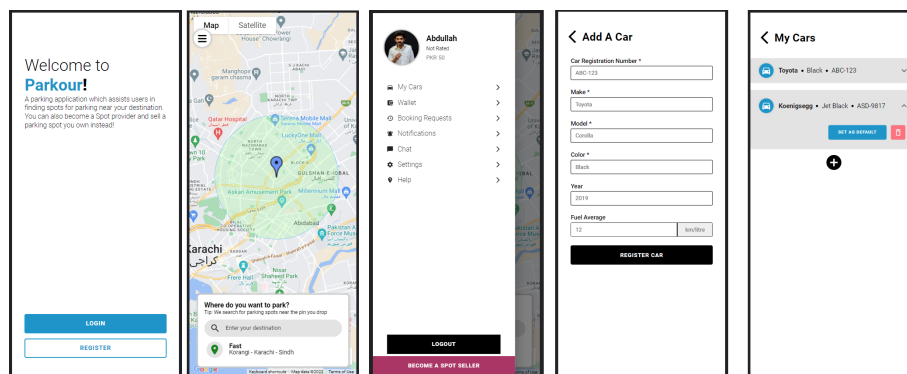


Figure 4.7: UI (1)

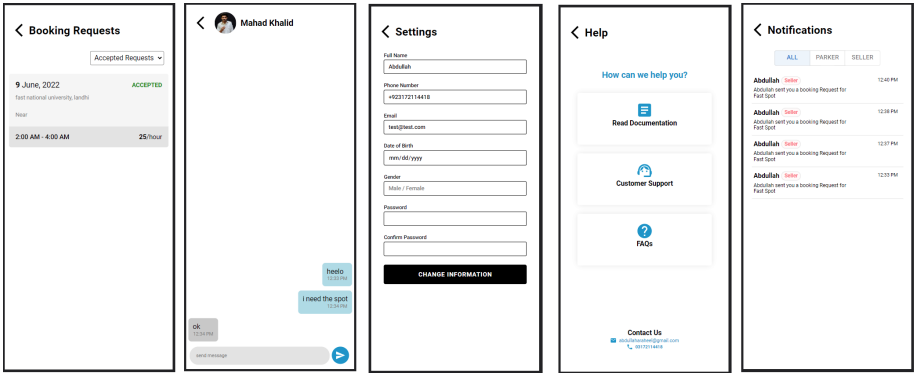


Figure 4.8: UI (2)

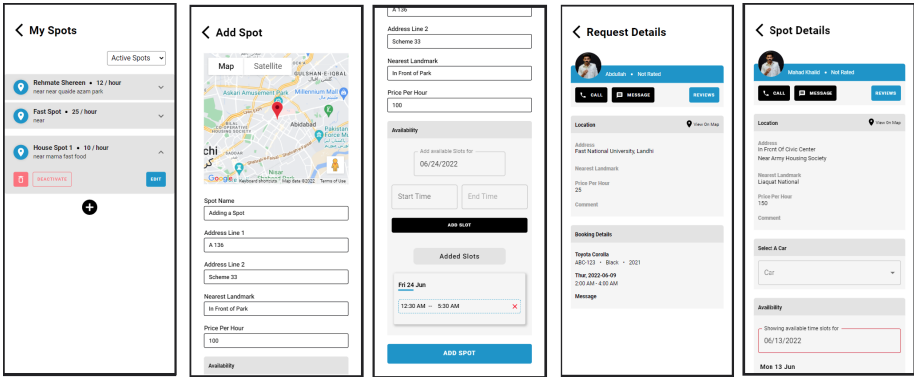


Figure 4.9: UI (3)

### 4.3 Backend

The design was planned considering the User Experience. Every effort is made to create a flexible data management method. An Example includes, maintaining history of data entities, time stamps, quality of life data.

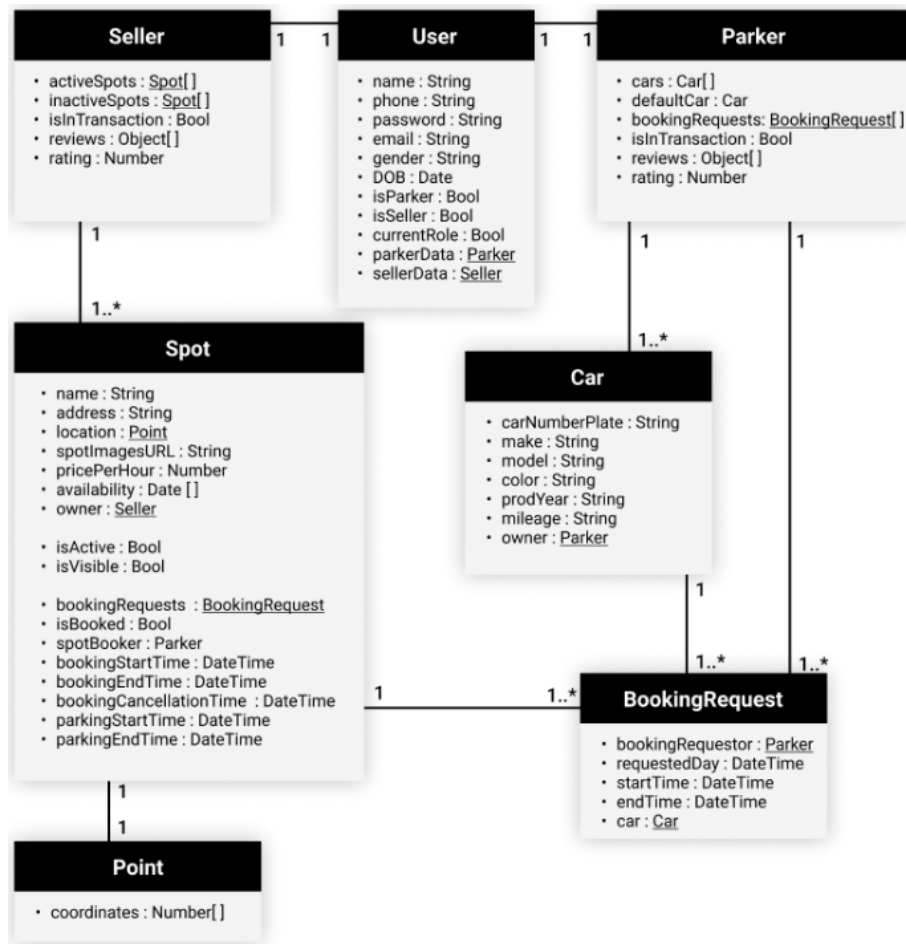


Figure 4.10: Database Design

Described the models in the following table.

Model	Description
User	Schema containing personal information. Through sub documenting, the information of the user's "seller" and "parker" accounts is also stored here.
Parker	All user attachments related to their "parker" role, along with transactional parker states.
Seller	All user attachments related to their "parker" role, along with transactional parker states.
Car	Parker's car data to be shown in transactions.
Spot	Seller's spot data, its location, images, status, ratings etc.
Booking Request	Purely transactional data relating to parker's request and the spot requested.
Notification	Used to cache notifications
Chat	The model that contains all the information about chat between parkers and sellers.

Table 4.1: Models

## 4.4 Architecture

System handles Parker and Seller Interfaces separately. Sellers and Parkers will be able to indulge in transactions which are described as ‘a Parker finding a parking spot and finding one, and a seller listing their vacant parking spot for sale.’

Both Parker and Seller will have digital in-app wallets to track their earnings/credits. Sellers will be able to cash in their credit after a certain number of conditions have been met.

### 4.4.1 Interface

The Interface was designed to be as user friendly as possible. The interface was designed to deliver the best user experience possible.

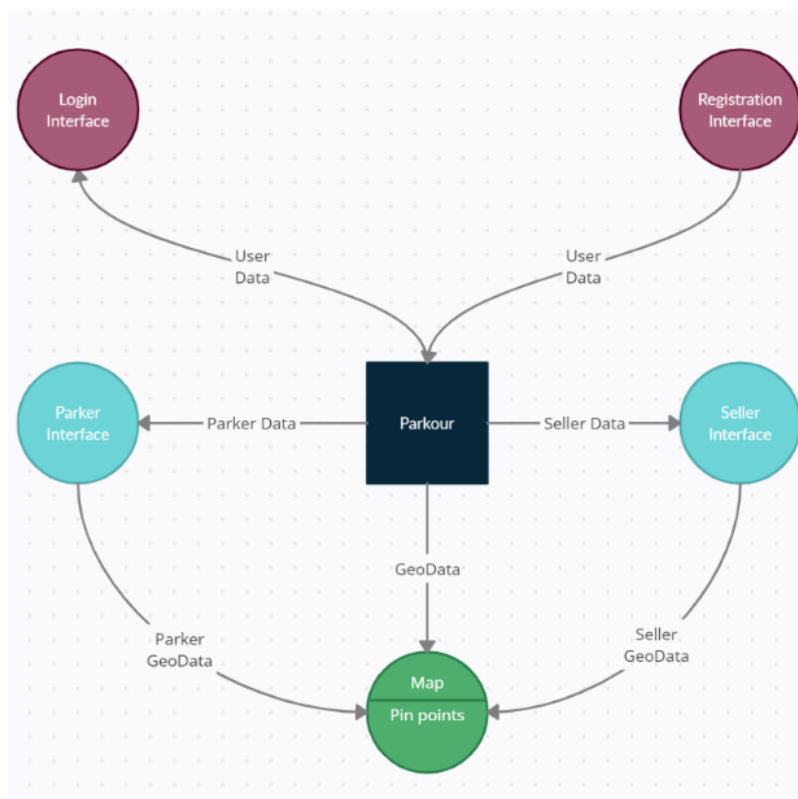


Figure 4.11: Interface

### 4.4.2 Server

The server processes all the request form the frontend and makes changes to database accordingly. The server also authenticates user and makes sure that the user is logged in. The server also handles the booking request and the notification. The server also handles the chat. The server sends a response on each request.

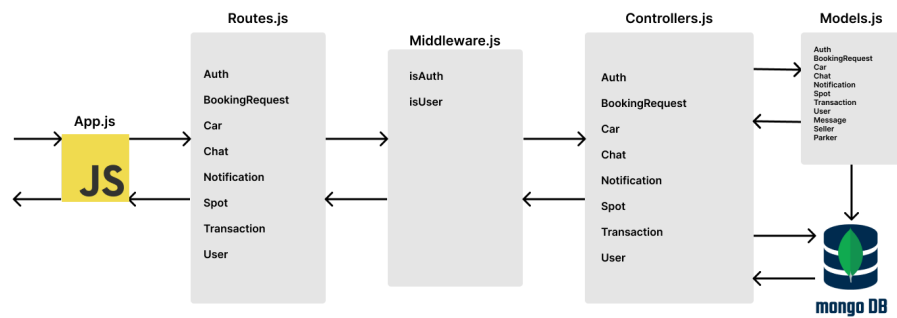


Figure 4.12: Server

### 4.4.3 Client

The client makes API calls to the server as well as the Google Maps server. The client makes use of redux to store information that can be used throughout the application.

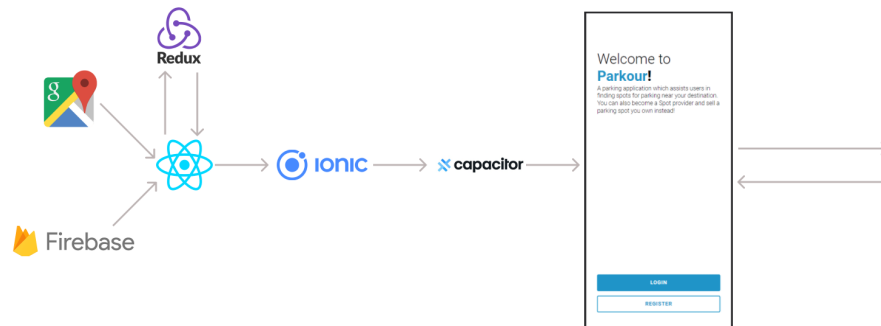


Figure 4.13: Client

### 4.4.4 Application

The interaction between client and the server.

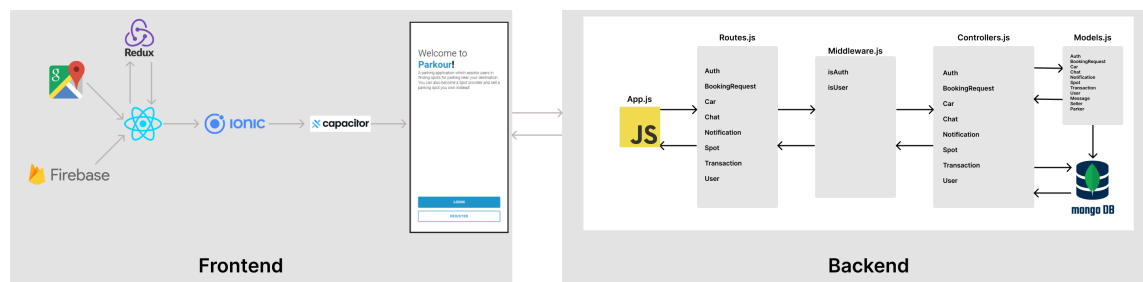


Figure 4.14: Application Architecture

# Implementation

The Application was mainly implemented in TypeScript. Implementation was done using React and Ionic.

	Technologies used
Frontend	React
Backend	Node.js
API	REST
Database	MongoDb
OTP	Firebase
Chat	Socket.io

Table 5.1: Technologies Used

## 5.1 Frontend

React with TypeScript was used for the frontend. Redux was used to store the state of the application and use it throughout frontend.

### 5.1.1 API Key

The API key to communicate with the Google Maps APIs is in the environment variable. This is done in the .env file.

```
import App from './App';
import store from './store/store';

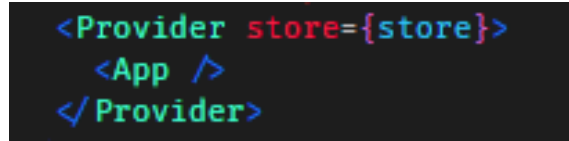
ReactDOM.render(
  <React.StrictMode>
    <LoadScript
      googleMapsApiKey={`https://maps.googleapis.com/maps/api/js?v=3.exp&libraries=geometry,drawing,places&key=${process.env.REACT_APP_MAPS_API_KEY}`} //
      libraries={['places']}
    />
  />
```

Figure 5.1: API Key



### 5.1.2 Store

The store is the main data storage for the application. It uses redux which in turn uses different slices to store information.



```
<Provider store={store}>
  <App />
</Provider>
```

Figure 5.2: Store Provider

### 5.1.3 Store Slice

The store slice is used through the reducers defined in the slice. Each Slice is given a name and a initial value. In this case the initial value is an empty object and the name of the slice is User.



```
const userSlice = createSlice({
  name: 'user',
  initialState,
  reducers: {
    createUser(state, action) {
      // Profile Info
      state.name = action.payload.name;
      state.phone = action.payload.phone;
      state.email = action.payload.email;
      state.gender = action.payload.gender;
      state.dob = action.payload.dob;
      state.password = action.payload.password;
      state.credit = action.payload.credit;

      // State
      state.isParker = action.payload.isParker;
      state.isSeller = action.payload.isSeller;
      state.currentRoleParker = action.payload.currentRoleParker;

      // Parker & Seller Details
      state.parker = action.payload.parker;
      state.seller = action.payload.seller;
    },
    updateUserInfo(state, action) {
      state.name = action.payload.name;
      state.phone = action.payload.phone;
      state.email = action.payload.email;
      state.gender = action.payload.gender;
      state.dob = action.payload.dob;
    },
    switchRole(state) {
      state.currentRoleParker = !state.currentRoleParker;
    },
    set_isSeller(state, action) {
      state.isSeller = action.payload.isSeller;
    },
    set_parker(state, action) {
      state.parker = action.payload.parker;
    },
    set_cars(state, action) {
      console.log(action.payload.cars);
      console.log(action.payload);

      state.parker.cars = action.payload.cars;
    },
    set_default_car(state, action) {
      state.parker.defaultCar = action.payload.defaultCar;
    },
    set_credit(state, action) {
      state.credit = action.payload.credit;
    }
  }
})

export const userActions = userSlice.actions;
export default userSlice.reducer;
```

Figure 5.3: User Slice in Store

### 5.1.4 Routing

Routing on the Frontend is done using React Router 6. All the routes are defined in the App.tsx

```

return (
  <IonApp className="app">
    <Router>
      <Routes>
        <Route path="/" element={
          <Fragment>
            { (isAuth && currentRoleParker === true) && <ParkerHome /> }
            { (isAuth && currentRoleParker === false) && <SellerHome /> }
            { !isAuth && <Home /> }
          </Fragment>
        } />

        <Route path="/login" element={
          <Fragment>
            {isAuth && <Navigate to="/" /> }
            {!isAuth && <Login /> }
          </Fragment>
        } />

        <Route path="/signup" element={<Signup/> } />

        <Route path="/otp" element={<OTP/> } />
        <Route path="/search" element={<Search/> } />
        <Route path="/bookingRequest" element={<BookingRequest/> } />
        <Route path="/requestDetails" element={<RequestDetails/> } />
        <Route path="/reviews" element={<Reviews/> } />
        <Route path="/submitReview" element={<SubmitReview/> } />
        <Route path="/notifications" element={<Notifications/> } />

        <Route path="/parker/mycars" element={<MyCars/> } />
        <Route path="/parker/registerCar" element={<AddCar/> } />
        <Route path="/parker/bookspot" element={<BookSpot/> } />
        <Route path="/parker/intransit" element={<Transit/> } />

        <Route path="/seller/mySpots" element={<MySpots /> } />
        <Route path="/seller/addSpot" element={<AddSpot /> } />
        <Route path="/seller/spotdetails" element={<SpotDetails /> } />

        <Route path="/allChats" element={<AllChats /> } />
        <Route path="/chat/:chatId" element={<Chat /> } />

        <Route path="/help" element={<Help /> } />
        <Route path="/setting" element={<Setting /> } />

        <Route path="/wallet" element={<Wallet /> } />
      </Routes>
    </Router>
  </IonApp>
);

export default App;

```

Figure 5.4: Routes

### 5.1.5 Firebase and Authentication

After a user registers on the application a OTP is sent to the user's number to verify their identity. The OTP is sent using Firebase's Authentication API.

```
import { initializeApp } from "firebase/app";

// Your web app's Firebase configuration
// For Firebase JS SDK v7.20.0 and later, measurementId is optional
const firebaseConfig = {
  apiKey: "AIzaSyB...",
  authDomain: "parkour-336414.firebaseapp.com",
  projectId: "parkour-336414",
  storageBucket: "parkour-336414.appspot.com",
  messagingSenderId: "144640928276",
  appId: "1:144640928276:web:7d5e8f8179336e7d589501",
  measurementId: "G-PC8DE8HHEJ",
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
export default app;
```

Figure 5.5: Firebase Config

### 5.1.6 Structure

Folder Structure used in the application.

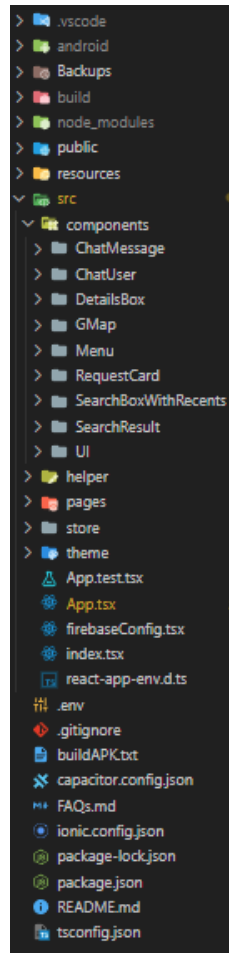


Figure 5.6: Folder Structure in React

### 5.1.7 Configuring Google Map

Google Map was configured in the Map.tsx file using the Google Maps API that was defined in Fig. 5.1.

```
// Creating parker map
const Map = useMemo( () => {
  return (
    <GoogleMap
      zoom={mapZoom || 13.5}
      center={currentLocation || KHI}
      mapContainerStyle={{ width: '100%', height: '100%' }}
      options={{
        fullscreenControl: false,
        zoomControl: false
      }}
    >
      {console.log(`${role} MAP RUNNING`)}
      <Marker
        position={currentLocation || KHI}
        options={{
          icon: {
            url: "http://maps.google.com/mapfiles/ms/icons/blue-dot.png",
            scaledSize: new google.maps.Size(50, 50),
          }
        }}
      />
      {spots.map((spot) => {
        const lng = spot.location.coordinates[0]
        const lat = spot.location.coordinates[1]

        return (
          <Marker
            key={spot._id}
            position={{ lat, lng }}
            onClick={() => markerClickHandler(spot)}
          />
        )
      })}
      {isParker &&
        <Circle
          center={currentLocation}
          radius={5000}
          options = {{
            strokeColor: '#6bbaff',
            strokeOpacity: 0.8,
            strokeWeight: 1,
            fillColor: '#9efa9b',
            fillOpacity: 0.3,
            zIndex: 1
          }}
        />
      }
    </GoogleMap>
  )
}, [currentLocation, mapZoom, spots, isParker, role, markerClickHandler])
```

Figure 5.7: Google Map Configuration

## 5.2 Backend

Node.js and Express was used to communicate with the frontend and Express provides Express Router which was used to create proper routing for APIs.

### 5.2.1 Database Connection

The connection was done using the Mongoose library. The server starts when the connection is established.

```
// Connecting to Database and Starting the server
mongoose
  .connect(MONGODB_URI, {
    useNewUrlParser: true,
    useUnifiedTopology: true
  })
  .then(() => {
    app.listen(SERVER_PORT, () => {
      console.log(`Server running on Port ${SERVER_PORT}`);
    });
  })
  .catch((err) => {
    console.log(err);
  });
```

Figure 5.8: Database Connection

### 5.2.2 API Routes

All the Routes are established in the App.js which is also the entry point in our application.

```
// ← Incoming requests flow from top to bottom. →
//Routes

app.use('/auth', authRoutes);
app.use('/user', userRoutes);
app.use('/spot', spotRoutes);
app.use('/car', carRoutes);
app.use('/bookingrequest', bookingRequestRoutes);
app.use('/chat', chatRoutes);
app.use('/transaction', transactionRoutes);
app.use('/notification', notificationRoutes);

// response for any unknown api request
app.use((error, req, res, next) => {
  console.log(error);
  const statusCode = error.statusCode || 500;
  const message = error.message;
  const data = error.data;
  res.status(statusCode).json({ message: message, data: data });
});
```

Figure 5.9: API Routes

### 5.2.3 Sockets

The sockets were used to communicate with the frontend. The sockets were used to enable bi-directional conversation to the frontend.

```
// Connect socket for client to client communication
io.on('connection', async (socket) => {
  console.log('=====→ CONNECTION INITIATED ←=====');
  socketHandler(socket);
});
```

Figure 5.10: Socket Connection

### 5.2.4 Chat Server

Chat Server uses sockets and is run differently from the main server. Chat Server emits and receives events through sockets.

```
import { Server } from 'socket.io';

const io = new Server(5001, {
  cors: {
    origin: 'http://localhost:3000',
    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE'
  }
});

export default io;
```

Figure 5.11: Chat Server



### 5.2.5 Routing

Routing on the Backend is done using Express Router. Here is an example of routing that is done for each module.

```
import { Router } from 'express';
import isAuth from '../middleware/isAuth.js';
import {
  getAllUsers,
  switchRole,
  getUserByRole,
  rateUser,
  getAllReviews,
  updateInfo,
  getUser // dev
} from '../controllers/users.js';

const router = Router();

// Posts

// Puts
router.put('/switchRole', isAuth, switchRole);

// Patch
router.patch('/', isAuth, updateInfo);
router.patch('/review', isAuth, rateUser);

// Gets
router.get('/getUserByRole/:roleId', isAuth, getUserByRole);
router.get('/review/:specialUserId', isAuth, getAllReviews);

// Dev APIs
router.get('/all', getAllUsers);
router.get('/getUser/:userId', isAuth, getUser);
router.get('/getAllusers', getAllUsers);

export default router;
```

Figure 5.12: User Routes

### 5.2.6 Middleware

Middleware are used to reduce duplication in code and to make the code more readable. This also helps with filtering out the invalid API calls.

```
import jwt from 'jsonwebtoken';
const verify = jwt.verify;

export default (req, res, next) => {
  const authHeader = req.get('Authorization');

  if (!authHeader) {
    const error = new Error('Not authenticated. ');
    error.statusCode = 401;
    throw error;
  }

  // token: header.payload.signature ← usually
  // here our token is: header.signature.payload, so...
  const signature = authHeader.split(' ')[1];
  let decodedToken;

  try {
    decodedToken = verify(signature, 'secretkey');
  } catch (err) {
    err.statusCode = 500;
    throw err;
  }

  if (!decodedToken) {
    const error = new Error('Not Authorized. ');
    error.statusCode = 401;
    throw error;
  }

  req.userId = decodedToken.userId;
  next();
};
```

Figure 5.13: isAuth Middleware

### 5.2.7 Controllers

Controllers are used to save and retrieve data from the database. Controllers are used to handle the requests from the frontend and after the request is handled with appropriate calculation, the response is sent to the frontend.

```
export const getUser = async (req, res, next) => {
  const userId = req.params.userId;

  try {
    const user = await User.findById(userId);
    if (!user) throwError('User not found', 404);

    let modifiedUser;

    if (user.currentRoleParker) {
      user.currentRoleParker = true;
      user.isParker = true;
      modifiedUser = await user.populate({
        path: 'parker',
        populate: {
          path: 'defaultCar cars'
        }
      });
      delete modifiedUser.seller;
    } else {
      user.currentRoleParker = false;
      user.isSeller = true;
      modifiedUser = await user.populate({
        path: 'seller',
        populate: {
          path: 'activeSpots inactiveSpots'
        }
      });
      delete modifiedUser.parker;
    }

    res.status(200).json({
      message: 'User fetched successfully!',
      user: modifiedUser
    });
  } catch (error) {
    next(error);
  }
};
```

Figure 5.14: User Controller

## 5.3 Database

### 5.3.1 Models

It is important to highlight the models that were designed and used. Each model had been given a careful consideration to make sure that the model is as modular as possible. The models were designed to be able to be used in multiple ways. The models were also designed in such a way that it will reduce redundancy wherever possible.

```
import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const Role = {
  PARKER: 'parker',
  SELLER: 'seller'
};

const userSchema = new Schema(
  {
    // Identification
    name: {
      type: String,
      required: true
    },
    phone: {
      type: String,
      required: true
    },
    password: {
      type: String,
      required: true
    },
    credit: { type: Number, default: 0 },
    email: { type: String, default: null },
    gender: { type: String, default: null },
    DOB: { type: String, default: null },
    socketId: String,

    // Attachments
    parker: {
      type: Schema.Types.ObjectId,
      ref: 'Parker'
    },
    seller: {
      type: Schema.Types.ObjectId,
      ref: 'Seller'
    },

    // State
    isParker: { type: Boolean, default: true },
    isSeller: { type: Boolean, default: false }, // to change the text on the button from either "Become a Seller" or "Switch to Seller Panel"
    currentRoleParker: { type: Boolean, default: true }
  },
  {
    timestamps: true
  }
);
```

Figure 5.15: User Model

```

import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const parkerSchema = new Schema({
  // ===== Attachments =====
  cars: [
    {
      type: Schema.Types.ObjectId,
      ref: 'Car'
    }
  ],

  defaultCar: {
    // default car to be shown on a fresh "Book Spot" page
    ref: 'Car',
    type: Schema.Types.ObjectId
  },

  bookingRequests: [
    {
      ref: 'BookingRequests',
      type: Schema.Types.ObjectId
    }
  ],

  // ===== State =====
  isInTransaction: {
    type: Boolean,
    default: false
  },

  // ===== Rating and Reviews =====
  cumulativeRating: {
    type: Number,
    default: -1.0
  },

  numberOfRatings: {
    type: Number,
    default: 0
  },

  reviews: [
    {
      author: {
        ref: 'User',
        type: Schema.Types.ObjectId
      },
      text: String,
      providedRating: Number
    }
  ]
});

```

Figure 5.16: Parker Model

```

import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const sellerSchema = new Schema({
  // ===== Attachments =====

  activeSpots: [
    {
      type: Schema.Types.ObjectId,
      ref: 'Spot'
    }
  ],

  inactiveSpots: [
    {
      type: Schema.Types.ObjectId,
      ref: 'Spot'
    }
  ],

  // ===== State =====

  isInTransaction: {
    type: Boolean,
    default: false
  },

  // ===== Rating and Reviews =====

  cumulativeRating: {
    type: Number,
    default: -1.0
  },

  numberOfRatings: {
    type: Number,
    default: 0
  },

  reviews: [
    {
      author: {
        ref: 'User',
        type: Schema.Types.ObjectId
      },
      text: String,
      providedRating: Number
    }
  ]
});

export default mongoose.model('Seller', sellerSchema);

```

Figure 5.17: Seller Model

```

import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const CarSchema = new Schema(
  {
    // ===== Identification =====>

    numberPlate: {
      type: String,
      required: true
    },
    make: {
      type: String,
      required: true
    },
    model: {
      type: String,
      required: true
    },
    color: {
      type: String,
      required: true
    },
    prodYear: {
      type: String
    },
    mileage: {
      type: Number
    },

    // ===== Owner Reference =====>
    owner: {
      type: Schema.Types.ObjectId,
      ref: 'Parker'
    }
  },
  {
    timestamps: true
  }
);

export default mongoose.model('Car', CarSchema);

```

Figure 5.18: Car Model

```

import mongoose from 'mongoose';
import PointData from './point.js';
const Schema = mongoose.Schema;

const SpotSchema = new Schema(
  {
    // Identification
    spotName: {
      type: String,
      required: true
    },
    addressLine1: {
      type: String,
      required: true
    },
    addressLine2: String,
    nearestLandmark: String,
    comment: String,
    location: {
      type: PointData.PointSchema,
      index: '2dsphere',
      ref: 'Point',
      required: true
    },
    imagesURI: [
      {
        type: String
        // required: true
      }
    ],
    pricePerHour: {
      type: Number,
      required: true
    },
    // References
    owner: {
      type: Schema.Types.ObjectId,
      ref: 'Seller',
      required: true
    },
    booker: {
      type: Schema.Types.ObjectId,
      ref: 'Parker'
    },
    bookingRequests: [
      {
        ref: 'BookingRequests',
        type: Schema.Types.ObjectId
      }
    ],
    // State
    availability: [
      {
        slotDate: Date,
        slots: [
          {
            startTime: { type: Date, required: true },
            endTime: { type: Date, required: true }
          }
        ]
      }
    ],
    isActive: { type: Boolean, default: true },
    isVisible: { type: Boolean, default: true }, // used to hide spot from map when spot is confirmed as 'booked'
    isBooked: { type: Boolean, default: false },
    bookingStartTime: Date, // seller accepted a parking request at this time
    bookingEndTime: Date, // listed start time of the spot
    bookingCancellationTime: Date, // booking cancelled - who cancelled will be checked in API
    parkingStartTime: Date, // The startTime of the slot for which the parker booked the spot.
    parkingExpireTime: Date, // user should un-park at this time
    actualParkingEndTime: Date // actual time the car was removed by user
  },
  {
    timestamps: true // provides createdAt and updatedAt fields
  }
);

export default mongoose.model('Spot', SpotSchema);

```

Figure 5.19: Spots Model



```
import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const bookingRequestsSchema = new Schema(
  {
    bookingRequestor: {
      ref: 'Parker',
      type: Schema.Types.ObjectId
    },

    spotOwner: {
      ref: 'Seller',
      type: Schema.Types.ObjectId
    },

    spot: {
      ref: 'Spot',
      type: Schema.Types.ObjectId
    },

    car: {
      ref: 'Car',
      type: Schema.Types.ObjectId
    },

    day: String,

    slots: [
      {
        startTime: { type: Date, required: true },
        endTime: { type: Date, required: true }
      }
    ],

    message: String,

    status: {
      type: String,
      enum: ['pending', 'accepted', 'rejected', 'all', 'past'],
      default: 'pending'
    }
  },
  {
    timestamps: true
  }
);

export default mongoose.model('BookingRequests', bookingRequestsSchema);
```

Figure 5.20: Booking Request Model

```
import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const NotificationSchema = new Schema({
  user: {
    type: Schema.Types.ObjectId,
    ref: 'User'
  },
  notifications: [
    {
      from: {
        type: String,
        required: true
      },
      text: {
        type: String,
        required: true
      },
      target: {
        type: String,
        required: true
      },
      time: {
        type: Date,
        required: true
      }
    }
  ]
});

export default mongoose.model('Notification', NotificationSchema);
```

Figure 5.21: Notification Model

```
import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const messageSchema = new Schema({
  sender: {
    ref: 'User',
    type: Schema.Types.ObjectId
  },
  message: String,
  time: String
});

export default mongoose.model('Message', messageSchema);
```

Figure 5.22: Message Model

```
import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const chatSchema = new Schema({
  userA: {
    ref: 'User',
    type: Schema.Types.ObjectId
  },
  userB: {
    ref: 'User',
    type: Schema.Types.ObjectId
  },
  messages: [
    {
      ref: 'Message',
      type: Schema.Types.ObjectId
    }
  ]
});

export default mongoose.model('Chat', chatSchema);
```

Figure 5.23: Chat Model

## 5.4 Behavior

A list of the behavior that was implemented in the Application. This also includes the flow of the application.

### 5.4.1 Parker

The Behavior of the Parker is as follows

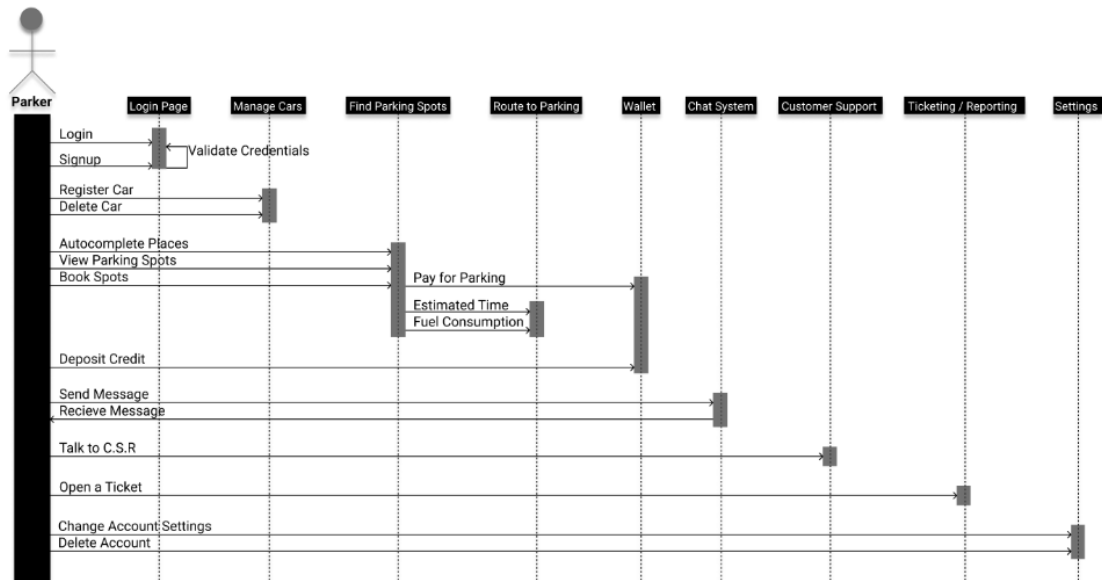


Figure 5.24: Parker Behavior (1)

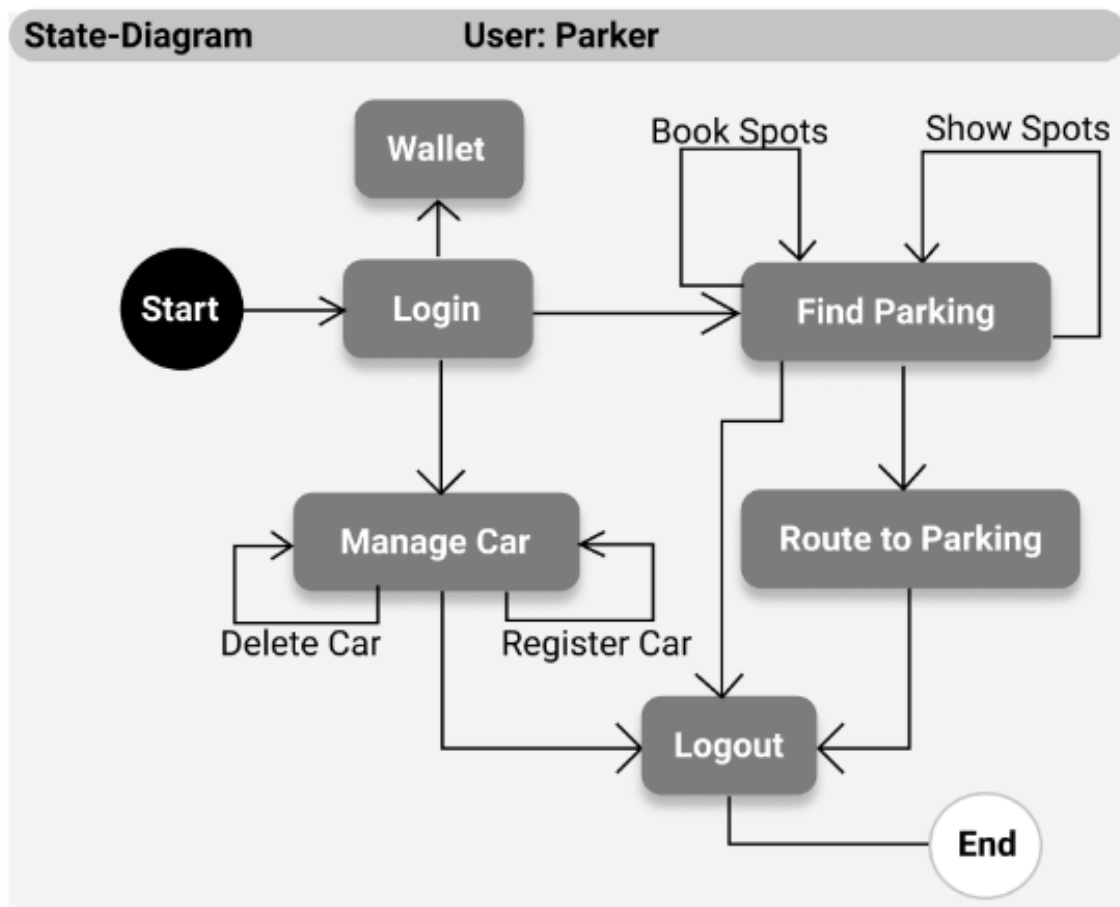


Figure 5.25: Parker Behavior (2)

### 5.4.2 Seller

The Behavior of the Seller is as follows

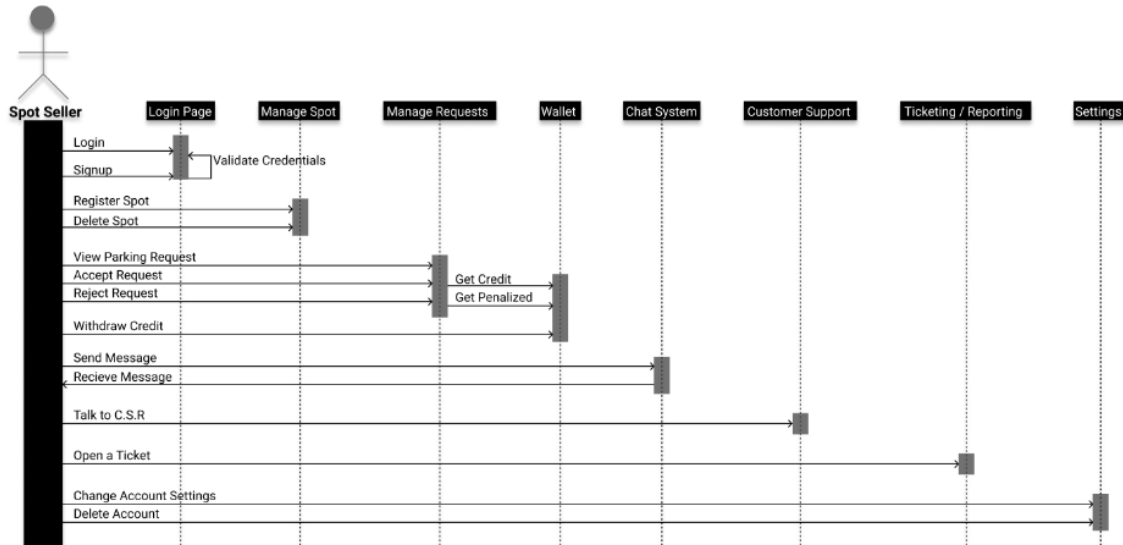


Figure 5.26: Seller Behavior (1)

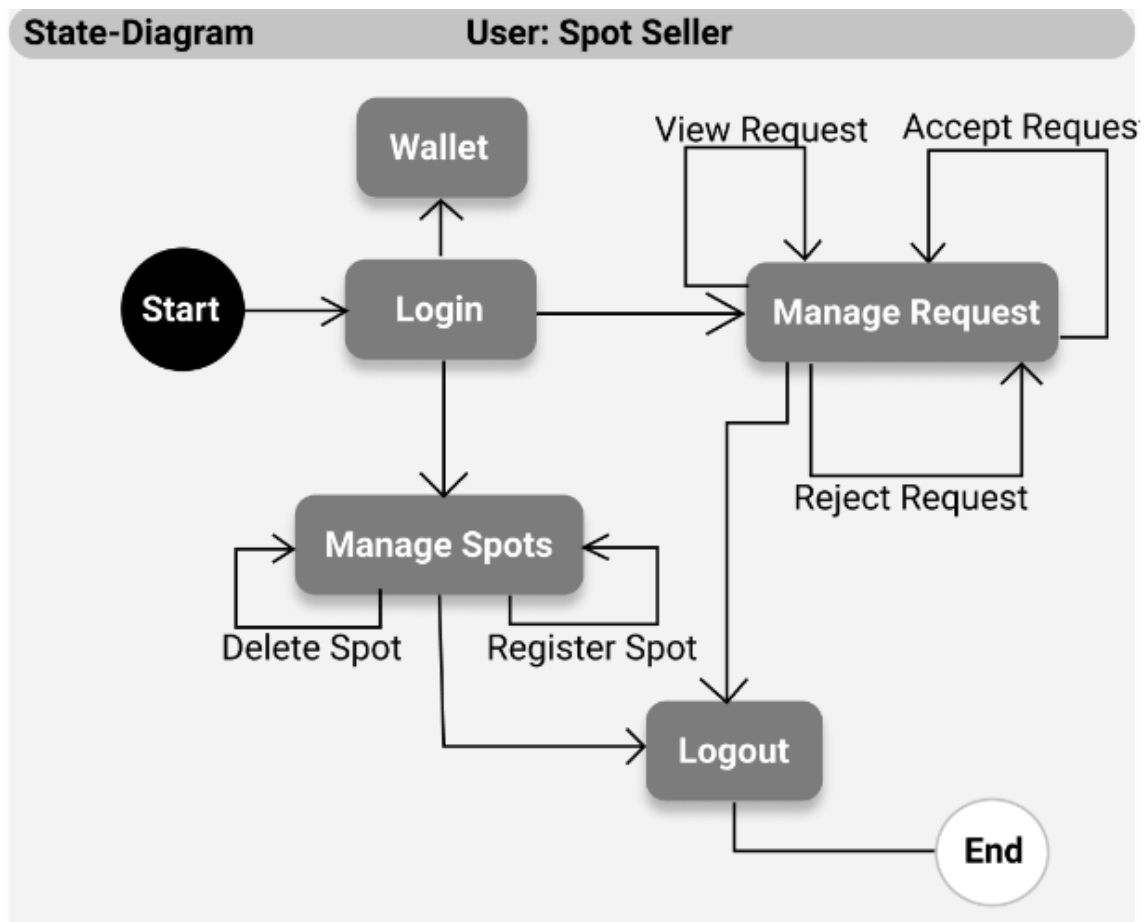


Figure 5.27: Seller Behavior (2)



### 5.4.3 Spot

The Behavior of the Spot is as follows

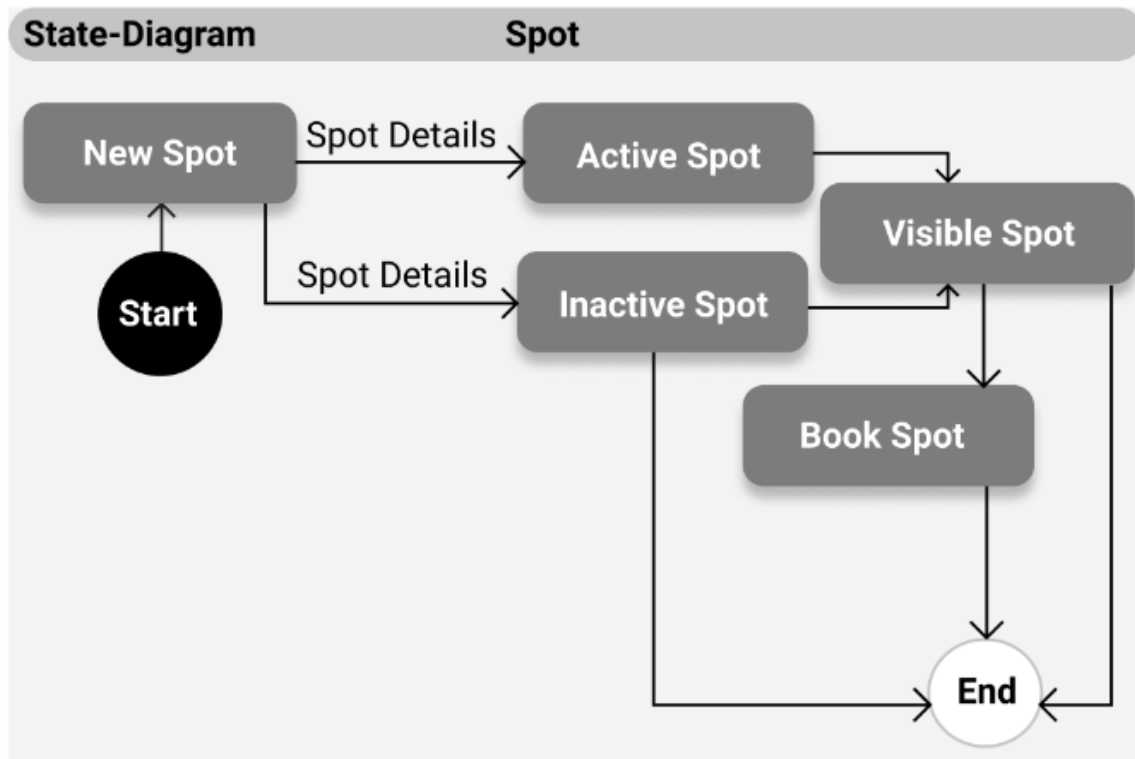


Figure 5.28: Spot Behavior

#### 5.4.4 Booking Request

The Behavior of the Booking Request is as follows

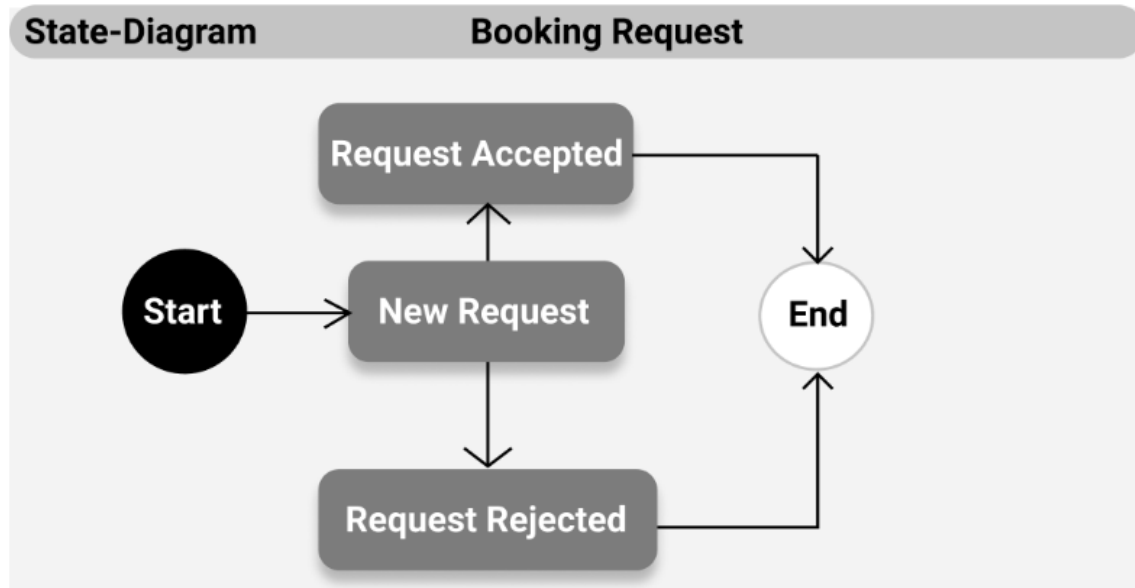


Figure 5.29: Booking Request Behavior

# Testing and Evaluation

Parkour will be a lightweight, reactive and responsive Application that will perform optimally on low-tier to high-tier mobile devices, including smartphones and tablets. The application will be tested on a wide range of smartphones with throttling to ensure a smooth and reactive experience. The User Experience in terms accuracy and responsiveness will be ensured because of the Application being a Single-Page App (SPA). Information on the application will be displayed and updated seamlessly without hindering User actions.

Parkour ensures safe and secure User Authentication using Session Tokens and Password Encryption. Each User will be awarded a Session Token through which their session will remain logged in. All sensitive information of a User is encapsulated and hidden from other users, and such information is only displayed on the Application's Interface by prior approval of the User. User Authentication will be done on the basis of Username, Phone Number and Password provided by the User.

The Parkers may find the exact parking spot location of the spot shown on the map and end up parking the car there without paying. The app does its best to prevent this and show only some details of the spot and the rest of the information is available after booking the spot. Parkour assumes the seller lists the spot that is in front of their property boundary or within their property. It also assumes that the spot is: i) at no parking zone, ii) any area that will cause inconvenience to residents and other drivers, iii) illegal spots.

Parkour provides a platform that provides a way for drivers to find parking spots listed by sellers. However, these spots must be self-managed by the seller. In case of any loss, or harmful activity, Parkour will not be responsible. However, Parkour provides customer support service, ticketing/reporting system, a penalty system, and rate review system, to help the users of our app.

# Conclusion

Due to the growing population, the growth of private vehicles is not going to halt and in turn the demand for parking. And increasing the parking supply by creating more and more parking spaces is not a feasible solution. Because land resources are limited, it isn't possible to conveniently plan parking spaces according to demand. Instead of increasing available parking spaces, an effective technology-based solution must be employed to optimize the use of available spaces and to provide drivers with a real-time map of available spaces and allow drivers to reserve spots based on their convenience. Not only does this save drivers time spent searching for a spot, but it also reduces environmental degradation resulting from congestion caused by parking. Finally, the parking system plays a key role in the metropolitan traffic system, and lacking it shows closed relations with traffic congestion, traffic accidents, and environmental pollution. And even with all of these factors, the parking problem is an often-overlooked aspect of urban planning and transportation. Therefore an efficient parking system can improve urban transportation and city environment besides raising the quality of life for citizens.

# References

- [1] Wenyu Cai, Dong Zhang, and Yongjie Pan. Implementation of smart parking guidance system based on parking lots sensors networks. In *2015 IEEE 16th International Conference on Communication Technology (ICCT)*, pages 419–424. IEEE, 2015.
- [2] Ellen Mitsopoulou and Vana Kalogeraki. Efficient parking allocation for smartcities. In *Proceedings of the 10th International Conference on PErvasive Technologies Related to Assistive Environments*, pages 265–268, 2017.
- [3] Wan-Joo Park, Byung-Sung Kim, Dong-Eun Seo, Dong-Suk Kim, and Kwae-Hi Lee. Parking space detection using ultrasonic sensor in parking assistance system. In *2008 IEEE intelligent vehicles symposium*, pages 1039–1044. IEEE, 2008.
- [4] Rahayu Yusnita, Fariza Norbaya, and Norazwinawati Basharrudin. Intelligent parking space detection system based on image processing. *International Journal of Innovation, Management and Technology*, 3(3):232–235, 2012.