SMART PARK

"RU Ready to Park Smart?"

Demo 2: Technical Documentation

Github:https://github.com/swetha-5689/SmartPark

Group 3

Disha Bailoor, Neha Nelson, Param Patel, Swetha Angara,
Nicholas Meegan, Thomas Murphy, Charles Owen,
Jeffrey Samson, Aniqa Rahim, Brian Ogbebor

Table of Contents

Customer Group Technical Documentation

Customer Frontend

1. *Login and Signup*

To create the Login page Auth0 was used. For use cases 1 and 2, logging in and signing up to check if a user has successfully created an account we can go to the Auth0 website. Under Applications click on SmartPark and click on users. There, we can see all the users who have logged onto the Smart Park website.

2. *Making a Reservation*

This is where the customer will arrive if they click on the Make a Reservation tab on the Navbar. They will see a calendar where they can select dates and 2 textboxes to type the start and end time of their reservation.

3. *Billing*

When the user arrives on this page they will see all of their outlying bills and a button to pay them all and

4. *Edit Account*

This page is fully functional and the user can now add vehicles and credit card information and change their account password. In the add new vehicle page, there will be text boxes where the user can enter their car model, make, color and license plate information. In the credit card information page, there will be 3 textboxes for credit card number, CVC and expiration date. Lastly, for the change password page, there will be two textboxes for changing username and password.

5. *Editing a Reservation*

This page will list all of the outlying reservations and give the customer the options to delete and edit those specific reservations. If they click delete, then that entry is deleted from the reservations table and if they select edit then they can edit the start and end time and the date of the reservation.

Customer Backend

1. *Billing (Model and Route)*

The model for a billing document is formed by a JSON formatted structure that contains:

```
{
        email: { type: String, required: true },
        username: { type: String, required: true },
        currentDues: { type: Number, required: true },
        dateofDues: { type: Date, required: true },
        description: { type: String, required: true }
}
```

When making a bill, the only required elements are the email, username, currentDues, dateofDues, and description. Adding a bill is only possible through doing a post request to the route:

https://secure-savannah-03864.herokuapp.com/billing/add which is accessible when a customer logs in to their account.

It is also possible to update someone's current bill by posting using the route: https://secure-savannah-03864.herokuapp.com/billing/update/:id. The id is the unique identifier that MongoDB gives any new added info when placed into the database. Adding that id to the end of the route will allow you to update the currentDues and the description.

Deleting a due (someone pays their bill) is also possible by doing a delete request to the route https://secure-savannah-03864.herokuapp.com/billing/:id. The id is the same as mentioned before and when used with url, the information associated with that bill will be deleted from the database.

In addition to all these things, you are also able to see all the bills in a table by doing a get request from the route: https://secure-savannah-03864.herokuapp.com/billing/.

2. *Reservations (Model and Route):*

The model for a reservation document is formed by a JSON formatted structure that contains:

```
{
        email: { type: String, required: true },
        date: { type: Date, required: true },
        startTime: { type: Number, required: true },
        endTime: { type: Number, required: true }
}
```

When making a reservation, the only required elements are the username, lengthreserve, date, startTime, and endTime. Adding a reservation is only possible through doing a post request to the route: https://secure-savannah-03864.herokuapp.com/reservation/add which is accessible when a customer logs in to their account.

In addition to this, you are also able to see all the reservations in a table by doing a get request from the route: https://secure-savannah-03864.herokuapp.com/reservation/.

3. *Vehicles*

The model for a vehicle document is formed by a JSON formatted structure that contains:

```
{
        make:  { type: String, required: true },
        model:  { type: String, required: true },
        color:  { type: String, required: true },
        licenseplate:  { type: String, required: true }
}
```

4. Credit Card Information

The model for a credit card document is formed by a JSON formatted structure that contains:

```
{
        creditcardnum:  { type: Number, required: true },
        cvc:  { type: Number, required: true },
        expdate:  { type: String, required: true },
}
```

Google Assistant

Smart Assistant support was implemented using Dialogflow and a webhook that sends a post request to the Reservations data collection in the SmartPark MongoDB database when a user makes a reservation. When making a reservation, the user currently only needs to specify the date of the reservation and the duration time of the reservation. More complex voice commands and interaction, as well as more required reservation information will be programmed in the future. SmartPark confirms the reservation with the user by repeating each input back as it is handled.

You can access the SmartPark assistant here: https://console.actions.google.com/.

Manager Group Technical Documentation

Startup

First in your terminal run the command $yarn install; this will install all of the dependencies needed to run the application. Once the installation is completed change your directory to the “src” folder. In the src folder run $yarn start. This should be done for all folders that you wish to run locally.

Manager Introduction:

Welcome to the Manager side of our Smart Park app! The Manager subgroup is tasked with automating and reducing the work that is required to successfully run a parking garage business. As a Manager, it is extremely crucial that the administrative staff is clear and continually updated on the state of their parking garage spots while also providing an easy and pleasant experience for our customers. This is our goal: to improve the organization and efficiency a thriving business needs while also appealing to the needs of the customers by implementing new ideas and solutions. The features we have implemented that will directly benefit anyone using our app to manage a parking garage is our statistics page, the reservation page, price page, login page, events page, and the view garage page.

Manager Files

Frontend Files

Manager/src/App.js: App.js is the file that contains all of our routing that will allow managers to access the various features that can be used on each page. These pages are the calendar. statistics, overview, registration, pricing, and garage overview pages which can all be accessed from our home page.

Manager/src/Calendar.js: The calendar.js file is responsible for the front end code that will allow a manager to see local events near his parking garage and add those into a system if he would like to advertise his parking for that event and offer discounts for parking at those events as well.

Manager/src/Home.js: Home.js is the home page that welcomes a manager as soon as he or she logs in. From this page we can access all other pages that managers need to access to utilize any or all features of the SmartPark product.

Manager/src/ParkingSpot.js: This file is responsible for transferring the details of each parking spot through the backend so that managers can view information about the status of individual parking spots through a front end rendering of their garage.

Manager/src/Reservations.js: This file will do some backend work to get the status of each parking spot and then returns a front end where managers can search for a customer and view their current status within the system.

Manager/src/Statistics.js: Retrieves backend information and formats into graphs using the react bootstrap front end so that managers can see useful business analytics to view any trends that are either beneficial or detrimental and based on the information the manager can make necessary changes as he or she sees fit. These statistics are retrieved from the backend.

Manager/src/ReservationRow.js: This file will be a combination of front end and backend. It has code that will make reservations.js front end work properly and works together to make the reservation feature function as it should.

Manager/src/EditLayout.js: This file will be the front end where a manager can see real-time spot properties including its vacancy, reservation status, spot and floor number etc. Within this one page, the manager can get an overview of the whole parking garage.

Manager/src/App.css: App.css is just CSS text that's meant to style whatever is going on in App.js (fonts, headers, margins, etc.) but we're not using App.js for anything but the router.

Manager/src/Pricing.js: This will hold our dynamic pricing model and will be a page where the manager can set the price for reservations, walk-ins, premium spots, handicapped etc.

Backend Files

Manager/Manager Back-End/models/CustomerResModel.js: This is the schema for the MongoDB reservation data collection that allows us to format any relevant inputted data to the backend for the Customer Database. Establishes the various elements that a reservation is associated with including its email, lengthreserve, date, startTime, endTime etc.

Manager/Manager Back-End/models/EventModel.js: This is the schema for the MongoDB Events data collection that allows us to format any relevant inputted data to the backend. Establishes the various elements that an event is associated with including its id, title, time, time, location, description etc.

Manager/Manager Back-End/models/FloorModel.js: This is the schema for the MongoDB Floor data collection that allows us to format any relevant inputted data to the backend. Establishes the various elements that a floor is associated with including its number etc.

Manager/Manager Back-End/models/PricesModel.js: This is the schema for the MongoDB Prices data collection that allows us to format any relevant inputted data to the backend. Establishes the various elements that a price is associated with including its overstay, walkIn, noShow, hourly etc.

Manager/Manager Back-End/models/ResModel.js: This is the schema for the MongoDB Reservation data collection that allows us to format any relevant inputted data to the backend. Establishes the various elements that an event is associated with including its custFName, custLName, type, car, email etc.

Manager/Manager Back-End/models/SpotModel.js: Establishes the various elements that a spot is associated with including its parking spot number, floor number, vacancy status etc.

Manager/Manager Back-End/routes/events.js: Retrieves events from the database. Supports viewing and creating for events on the events page.

Manager/Manager Back-End/routes/floors.js: Retrieves the floors in the database.

Manager/Manager Back-End/routes/prices.js: Supports updating and viewing prices.

Manager/Manager Back-End/routes/reserve.js: Supports viewing and searching for reservations. Also supports creating a reservation through the google assistant.

1. Login/Register

   To create the Login page Auth0 was used. This is currently managed by Auth0, so clicking the login button will redirect to the Auth0 login page and unlock manager pages after correct credentials are submitted. The Registration process is currently set up so that a system admin can register users and their passwords.

   Example Credentials: Email: swetha@example.com Password: SmartPark202SmartPark2020

2. Reservations

   The login page can be found at https://shielded-lake-21193.herokuapp.com/reservations. This page accesses the database through the API at https://cryptic-depths-70075.herokuapp.com/api/res/.

3. Garage Overview

   View-Only

   The garage overview page can be found at https://shielded-lake-21193.herokuapp.com/viewlayout. This page accesses the database through the API at https://cryptic-depths-70075.herokuapp.com/api/spots/ - Gets all Spots
   https://cryptic-depths-70075.herokuapp.com/api/spots/floor/:floorNumber - Gets All Spots for a Particular floor
   https://cryptic-depths-70075.herokuapp.com/api/floors/ Gets all Floors

4. Statistics

   The statistics page can be found at https://shielded-lake-21193.herokuapp.com/statistics. It is currently front-end only and shows a different graph based on which button is selected.

5. Events

   The events page can be found at https://shielded-lake-21193.herokuapp.com/calendar. It is currently connected to our MongoDB Events Data Collection and shows a daily schedule of events that could affect parking rates for the day.

6. Prices

   The pricing page can be found at https://shielded-lake-21193.herokuapp.com/pricing. It supports manual override of hourly rates and includes options for adding different rates for walk-in and overstay customers.

Back-End

Currently, GET requests to the endpoint are possible at:
https://cryptic-depths-70075.herokuapp.com/api/res/
https://cryptic-depths-70075.herokuapp.com/api/spots/
https://cryptic-depths-70075.herokuapp.com/api/floors/
https://cryptic-depths-70075.herokuapp.com/api/spots/floor/:floorNumber
https://cryptic-depths-70075.herokuapp.com/api/events/
https://cryptic-depths-70075.herokuapp.com/api/prices/
A POST Request to
https://cryptic-depths-70075.herokuapp.com/api/res/ will save a reservation from the Google Assistant to the customer and manager database for reservations.

Pricing Admin Module

The pricing admin class provides a variety of API's accessible to an outside user. The uses for these API's is as follows:

1. Generate price from two 24-hour formatted time inputs. The price is generated using the live price model as set by the admin.

   GET HTTP API Access point:

   https://pricing-admin2.herokuapp.com/api/getprice

   The JSON format should be:

```
{
        "start" : "06:00",
        "end"   : "08:00"
}
```

   The anticipated output will be a JSON formatted return, where the price shown is simply an example:

```
{
         "price": 12.50
}
```

2. Set live price model API. Through this API the currently live price model can be manipulated.

   POST HTTP API Access point:

   https://pricing-admin2.herokuapp.com/api/liveprice

   The JSON formatted input should be formatted as follows:

```
{
        "model_name" : "Live Model",
        "occupancy_percent": <array of 24 weight values>,
        "base_rate": 6,
        "min_thresh": 60,
        "max_thresh": 90,
        "base_rate_mult": 1.4,
        "total_spots": 100,
        "operation": "makenew"
}
```

   There is only one live price model at any given time. Therefore the “model_name” field must not be changed.

   The significance of weights is explained in the User Documentation.

3. View historic price model and actual parking occupancy for a date. Generates an JSON formatted output containing the following fields:

```
{
        “live_revenue”: <rev. array by hour, live model>,
        “actual_revenue”:<re. array by hour, actual results>,
        “base_rate”:<int, base rate for date>,
        “base_rate_mult”:<int, base rate mult for date>,
        “min_thresh”:<int, base rate mult for date>,
        “max_thresh”:<int, non decimal %>,
        “min_thresh”:<int, non decimal %>m
        “total_proj_rev”:<int, total projected revenue for day>,
        “occupancy_percent”:<array of occupancy weights>,
        “actual_occupancy_percent”:<array of actual occupancy %>,
        “actual_revenue_total”:<int, total actual revenue for date>
```

}

POST HTTP API Access point:

https://pricing-admin2.herokuapp.com/api/historic

The JSON formatted input should be as follows:

{

“date”: yyyy-mm-dd

}

The Back-End for

The back end is currently deployed at https://cryptic-depths-70075.herokuapp.com/api/res/.
It can be tested by sending requests through an application like Postman.
So to find reservations, one can send a GET request to
https://cryptic-depths-70075.herokuapp.com/api/res/.
Currently active:
https://cryptic-depths-70075.herokuapp.com/api/res/
https://cryptic-depths-70075.herokuapp.com/api/spots/
https://cryptic-depths-70075.herokuapp.com/api/floors/
https://cryptic-depths-70075.herokuapp.com/api/spots/floor/:floorNumber (ex. 1)

Elevator Group Technical Documentation

Elevator Front End

The elevator front end consists of and depends on five different components: connectedDevices.jsx, devMode.jsx, message.jsx, plateList.js, and navbar.jsx. Each component plays a vital role in the operation of the front end and back end combination of the elevator terminal. This documentation will walk through the inner workings of the elevator terminal's interface.

The first component, connectedDevices.jsx, is run when the elevator terminal does not have a particular important component or components connected to the front end display of the elevator. These devices include: the camera in the elevator, the license plate scanner of the elevator, the weight sensor central hub (which stores the status of each spot) and the database. In the event these devices are not connected, then the elevator terminal will stay on this screen and not proceed until the devices are properly connected. This is done for quality assurance and that the customer does not experience any unexpected results when attempting to park. Once each of the devices are properly connected, connectedDevices.jsx is not continued to be run and advances on to the main elevator terminal interface. Besides the getDevicesConnected() function, each of the other functions are for use in the DEMO MODE, discussed later in the technical documentation.

The main framework for the elevator terminal front end can be found in the message.jsx file. The flow of control for the elevator terminal is primarily controlled through the advanceScreen() function, which relies on various helper functions to complete its execution. The advanceScreen() function uses the state of the message.jsx as well as information taken from the elevator MongoDB database to control which message is displayed by the system. In order to advance through the message screens, helper functions such as wasScanned(), wasResFound(), isValidMemNumber(), and isGarageFull() are employed. Each of these functions checks the state of the current operation in progress. wasScanned() checks to see if the license plate was able to be scanned successfully or not, wasResFound() determines if a reservation is in the system for that particular user, isValidMemNumber() checks to see if a valid membership number was entered, and isGarageFull() performs a real-time check in order to see if the garage is full or not. These four functions, along with the framework that ties it all together, advanceScreen(), controls how the elevator advances based on each different situation. Moreover, to advance the screen, the button advanceButton() is used. This is to ensure that the user does not accidentally enter an incorrect value and forces the customer to pause for a moment and analyze his or her input into the system.

Two important screens to note are connected to the reservation number. This includes the screen where the customer is prompted if they have a reservation number or not. If the customer hits yes, then a screen with a number pad pops up, where the user enters his or her membership number. This value is stored in inputMemNum, a variable. This variable is reset for each time the customer enters the elevator. Additionally, in order to return flow back to the beginning of the elevator terminal, a variable known as reset is used. This variable is checked each time the advanceScreen() function is called. Every other function in the message.jsx file is used to change the state of buttons, or in order to wait for an input to be entered in order to advance the terminal.

The navbar.jsx does not have any functions associated with it and is rather for display purposes. This can be used to alter the design of the header of the SmartPark elevator terminal, as well as the background.

The last component of note is the devMode.jsx component. The dev mode, or demo mode, is used for testing purposes or to show off the product. Demo mode is controlled by one boolean variable, which enables or disables the developer mode. It can only be changed through the code, so there is no way to access this option other than by editing the source code. The demo mode allows for the screens to be advanced through buttons rather than relying on a back end or other necessary components. It activates certain buttons that are not visible when the demo mode is disabled. These buttons can be found in both the connectedDevices.jsx file and the message.jsx file. The functions associated with these buttons are also found in these respective files.

Spot UI

The Spot UI code currently has one page with multiple reactstrap components. These are the descriptions of the files that have been used in the code, components first and pages later, in the order that they have been used:

MyTable.js:

This is the ‘A’ table. The first table that is used to simulate the spots on the floor layout of the parking garage. This file includes all the styles that are used in the table layout.

MyTable2.js:

This is the ‘B’ table. The second table that is used to simulate the spots on the floor layout of the parking garage. This file includes all the styles that are used in the table layout.

MyTable3.js:

This is the ‘C’ table. The third table that is used to simulate the spots on the floor layout of the parking garage. This file includes all the styles that are used in the table layout.

MyTable4.js:

This is the ‘D’ table. The fourth table that is used to simulate the spots on the floor layout of the parking garage. This file includes all styles used in the table layout.

MyTable5.js:

This is the ‘E’ table. The fifth table that is used to simulate the spots on the floor layout of the parking garage. This file includes all the styles that are used in the table layout.

App.js:

This has the code for the page, linking and rendering all the components that need to be present on the frontend of the UI. All the tables that have been described above have been rendered as different components in order on this page.

Elevator Back End

Back End Files

Keys.js:

This is where the URL address for the MongoDB database goes. This file, as well as the URL inside as a string, is called later on in the Server.js file in order to link to MongoDB and start the server. Currently the user is set to Tom and the password for user Tom is visible in the address, but this will all be changed later on for the second demo so that all groups, Customer, Manager, and Elevator, are using the same database.

Server.js:

This file will start the server and connect to the MongoDB database that is specified in the URL in Keys.js. This imports Express and Mongoose in the file. Express is used to create the HTTP requests that are used later in the document. Mongoose acts as the MongoDB framework so that things can be added and extracted from the database. The server must be running in order for any database related activity to take place.

Customer.js:

This is the model for the customer in the database. It uses Mongoose to make the framework for the model. The customer will have a name, a license plate, and a membership number. The membership number will be changed to a registration number in order to be more consistent with the other groups.

Spot.js:

This is the model for the spot that will be accessible from the elevator terminal in the second demo. It uses Mongoose to make the framework for the model. The spot will have a number, a floor number, a vacancy status, a reserved status, a premium status, and a handicapped status associated with it. This may be changed later on in order to have more consistency with other groups.

Customers.js:

This page sets up the routes for the HTTP requests to the MongoDB database for the customer data that the elevator group is using. HTTP requests include get, post, and delete. This uses Express in order to accomplish this task. Express contains all the functions necessary to route an HTTP request. Cors is used so that the database requests can be accessed from other pages that are not associated with the database (ie the frontend). A get request set up to display all of the customers in the database. A get request is also set up to search for a license plate and see if it exists in the database. If it does exist, it is set up to display the information of the user associated with the license plate. There is also a delete request which can delete a customer in the database by entering their name. This will be changed for the second demo to be more consistent with the Manager and Customer groups. Additionally, post and delete requests will be removed from the elevator group's HTTP requests, as the user's will not be able to do these in the elevator terminal.

Spots.js:

This page sets up the routes for the HTTP requests to the MongoDB database for the spot data that the elevator group is using. HTTP requests include get, post, and delete. This uses Express in order to accomplish this task. Express contains all the functions necessary to route an HTTP request. Cors is used so that the database requests can be accessed from other pages that are not associated with the database (ie the frontend). A get request set up to display all of the spots in the database. A get request is also set up to search for all of the vacant spots in the database (having a value of "True" if vacant and "False" if occupied. There is another get request that displays a spot and the information associated with it if the spot's number is entered. Lastly, there is an ability to delete a spot in the database by its id number. Certain functions will be changed for the second demo to be more consistent with the Manager and Customer groups. Additionally, post and delete requests will be removed from the elevator group's HTTP requests, as the user's will not be able to do these in the elevator terminal.

Models

The elevator group employs models similar to the manager group. These models are modified to fit better with the elevator group's functionality. They are as follows in JSON format:

Customer
name:{
type: String
required: true
},
licensePlate:{
type:String,
required: true
},
resID:{
type: String,
required: true
},
spotNumber:{
type: string,
required: true
}

Spot
spotNumber:{
type: String
required: true
},
floorNumber:{
type:String,
required: true
},
resID:{
type: String,
required: false
},
isVacant:{
type: Boolean,
required: true
}
isReserved:{
type: Boolean
required: true

},
isPremium:{
type:Boolean,
required: true
},
startTime:{
type: String,
Default: 0
}

Technical Procedure

1. The user enters the elevator and their license plate is scanned. The license plate is input into a GET HTTP request :

   "http://localhost:5000/api/customers/searchPlate/" + license plate that was scanned”

2. If there is something found in the database, the program returns the customers name, reservation ID, and spot number associated with that ID.

3. If no license plate is found, NULL will be returned and the user will be redirected to enter their reservation ID into a numpad.

4. User either has their input found in the database in step 2 or enters their reservation into a numpad in step 3. A GET HTTP request is made:
   - If step 2: "http://localhost:5000/api/customers/searchResID/" + input
   - If step 3: http://localhost:5000/api/spots/searchResID/" + input

5. If something is found the program returns the spot the user is registered to and the user is redirected to that spot. If not, NULL will be returned and the user will be asked to leave.