SMART PARK

"RU Ready to Park Smart?"

Demo 1: 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 their total amount due and textboxes to enter their credit card number, cvc pin and the expiration month and year.

4. *Edit Account*

When this page is fully functional the user can add new vehicles, add their 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 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.

Customer Backend

Currently, the customer backend consists of a MongoDB cluster that houses our billing data for each customer/user as well as their reservation which is also tied to their account.

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:

```
{
        username: { type: String, required: true },
        lengthreserve: { type: Number, 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/.

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, calendar page, and the view garage page.

Manager 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 edit layout 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/ParkingsSpot.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.

Manager/src/GarageGenerator.js: This file will be the front end where a manager can see the whole garage and view the information associated with each individual spot. This page will display information like reservation status, customer name, payment status etc.

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 add or delete another spot and edit the overall dimensions of the garage. This code is the front end and will rely on manager/backendSpots/routes/spot.js for the back end process.

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.

manager.backendSpots/config/keys.js: This file is the configuration using our URI for mongo DB database.

manager.backendSpots/models/spot.js: This file will configure all of our characteristics and properties for parking spots. These characteristics are the spot number, floor number, Vacancy status, reservation status, and what type of spot it is (handicapped or premium).

manager/backendSpots/routes/spots.js This file has the back end code for creating, deleting, or getting the object of a parking spot in the backend. Any changes made using

this file will eventually be reflected in the front end once they are connected for the second demo.

manager/backendSpots/Server.js This file is what connects us to our MongoDB database and uses both express and mongoose to do that.

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. Edit Layout

Currently View-Only

The login page can be found at https://shielded-lake-21193.herokuapp.com/editlayout. 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. Calendar

The calendar page can be found at https://shielded-lake-21193.herokuapp.com/calendar. It is currently front-end only and shows hard-coded events.

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
A POST Request to
https://cryptic-depths-70075.herokuapp.com/api/res/ will save a hardcoded JSON reservation to the database.

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

In its current state, the elevator front end consists of and depends on four different components: connectedDevices.jsx, devMode.jsx, message.jsx, and navbar.jsx. Each component plays a vital role in the operation of the front end 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 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 membership number. This includes the screen where the customer is prompted if they have a membership 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.

Simulation

The simulation code currently has different components that are used in the multiple pages that are presented in the Front End. We used reactstrap to accomplish the styling of all the components used in this project. 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:

AppNavBar.js:

This has the code of the navigation bar that is included on each page. This has been used as the standard navigation bar on the elevator side of the project.

App.js :

Has all the code for the different pages that need to be linked by the react-router, including the Navigation Bar component which is set on each page before the other changes are made. Sets the path and address for which each page can be found in the project folder. The order of the linking of the pages has been mentioned in this page, so you can see that the order of pages is Home Page, Scanned Page, Unscanned Page, Found Page and Not Found Page. The links are based on the different scenarios that we are trying to simulate on the customer end.

Button.js:

This is the button that is used on the Homepage with the title ‘the car comes to the elevator terminal’. The styling of the button is decided on this page.

Home.js:

Includes the code for the styling of the home page, with the title ‘Car Enters Garage’. It’s the first page that the user can see when the user starts the project. It is linked to the Scanned Page next. This includes the Button.js component that links to the Scanned Page.

Button_scan.js:

This is a button titled ‘Enter Membership Number’ on the Scanned Page which links to the notScanned.js page. This is for the possibility that the license plate either couldn’t be found or there was a faulty scan, it gives the user to input their membership number to find the reservation.

ProceedButton.js:

This button is titled ‘Proceed’ which is for the case that the license plate was scanned and the reservation is found, so it would take you to the found.js.

Scanned.js:

This is the scanned page to outline the scanning of the license plate when the car approaches the elevator terminal. This page has two button components linking two different pages for two different scenarios. For a successful scan, we can press the proceed button which is linked to the found.js page. For an unsuccessful scan, we ask the user to enter the Membership Number to find the reservation, using the Button_scan.js which links to the notScanned.js page.

Button_aftScan.js:

This is the button titled ‘Search’ which leads to a search in the backend for the membership number entered by the user, to find the reservation if one exists. This button links to found.js for now.

FailButton.js:

This is the button titled ‘Not Found’ for the case for when the reservation is not found even when the membership number is entered in the front end UI. This links to the notFound.js page.

NotScanned.js:

This page outlines the different scenarios for when the user enters the membership number into the frontend UI. This includes the AfterScanButton class from the Button_aftScan.js which links to the success page or the found.js page. This has been coded as such for now because we haven’t established a connection to the backend for this demo. This will change once we do actual searches in the backend and query for results, that will determine whether this will be a successful search or an unsuccessful search. This also includes the Not Found Button which is

referenced from the FailButton.js file, which links to the notFound page for the scenario that the membership number is not found in the backend.

Found.js:

This page is for the success scenario, for when the reservation is found. This page has the Proceed button component which links to the home page in this scenario. The description of the Proceed Button is given above.

GoBackButton.js:

This button is present in the fail scenario. It is titled ‘Return to the Main Page’. It takes you back to the home page to loop the scenarios together from the notFound page.

NotFound.js:

This is a fail scenario page for when the reservation is not found even after the user enters the membership number, this informs the customer that there is no reservation under their membership number.

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

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.