SMART PARK

"RU Ready to Park Smart?"

Report 3

GitHub: https://github.com/swetha-5689/parking-manager

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

Individual Contribution Breakdown

After consulting the entire group, we have decided that all of the team members have contributed equally to this report.

Summary of Changes

1. Customer Problem Statement
    - ***Removed extraneous wish list items and improved clarity of the problem statement***
2. Glossary of Terms
    - ***Updated definitions of Customer Reservation System, Vacant Spots, Customer and Administrator***
    - ***Removed Weight Sensor definition***
3. System Requirements

    Customer Registration

        Updated:

            Req: C-F-3 + Req: C-F-4 (combined)
            Req: C-NF-13 + Req: C-NF-14 (combined)
            Req: C-F-10
            Req: C-F-12
            Req: C-UI-20

        Removed due to redundancy:

            Req: C-F-11
            Req: C-NF-17
            Req: C-UI-19

    Managerial / Administrative

        Updated:

            Req: M-NF-12

        Removed:

            Req: M-F-4 (*Will not be implemented.)*
            Req: M-UI-20 *(Will not be implemented.)*

    Elevator Operation

            Req: E-F-6 downgraded from 5pt to 1pt.
            Req: E-F-5: upgraded from 4pt to 5pt
            Req: E-UI-19: Moved to “future work”. Will not be implemented by final demo.

4. Functional Requirement Specification
    - ***Use cases moved to “Future Work” (will not be implemented by final demo)***
        UC-6 Make contracted reservation
        UC-8 Extend grace period
        UC-9 Create garage layout: Removed due to lack of necessity
        UC-12 Issue Rain Check (Manager): Moved to Future Work
    - ***Merged Use Case 19 - Park, with Use Case 17 as they were coinciding in functionality***
    - ***Created a new Use Case 19 View Events***
    - ***Updated Customer Subgroups’ System Sequence diagram and fully dressed cases to include the new use cases that are being incorporated***

- ***Updated and added descriptions to the system sequence diagrams for the Elevator Subgroup***
- ***Updated Traceability Matrix***

5. Effort Estimation using Use Case Points
   - ***Recalculation for all subgroups***
6. Domain Analysis

   Domain Model

   Concept Definitions

   Updated:

   DC-9: Added garage overview functionality.

   Removed mention of garage design and modularity *(will not be implemented)*

   DC-6: Removed mention of grace period *(will not be implemented)*

   Association Definitions

   Updated:

   Load Garage: Changed name from Garage Configuration

   Added real-time garage overview functionality.

   Removed garage design modularity *(will not be implemented)*

   Attribute Definitions

   Updated:

   Added separate column for Responsibility ID

   Traceability Matrix
   - ***Updated Matrix***
   - ***Updated Matrix Descriptions***

   System Operation Contracts
   - ***Updated to include Edit Account and Edit Reservation Contracts***
   - ***Removed Edit Garage Layout Contract***
7. Interaction Diagrams

   Customer Registration
   - ***Added diagrams for UC 3 and 5, labels and descriptions that mention specific design patterns***

   Managerial / Administrative
   - ***Removed diagram for UC-9 will not be implemented***
   - ***Added descriptions and image labels to reflect use of new design patterns***

   Elevator Operation
   - ***Added image labels***
   - ***Combined UC-19 and UC-17***
8. Class Diagram and Interface Specification
   - ***Overall class expanded to include three new modules***

     Dynamic Pricing

     Event Display

     Google Home Back Interface
   - ***Added OCL Contracts***

- ***Traceability Matrix***

  Added intersection of the three new classes
  Added description of new class intersections
  Updates older class descriptions

9. System Architecture and System Design

   Customer Registration

   Edited descriptions to account for recent changes in architecture

   Managerial / Administrative

   Edited package diagram to account for changes to implemented use cases
   Edited mapping hardware section to reflect deployment

10. Algorithms and Data Structures

    Managerial / Administrative

    Added format of Spots, Floors, and billings data collections

11. User Interface Design and Implementation

    Customer Registration

    - ***Added Edit Account and Reservation Pages, updated Calendar UI for Making a Reservation***

    Managerial / Administrative

    - ***Edited UI of Statistics Page and View Layout Page, implemented search in Reservation Page***

    Elevator Operation

    - ***The user no longer can park if they are a walk-in customer***
    - ***Removed the walk in terminal***

12. Design of Tests

    Customer Registration

    - ***Added Test Cases for newly implemented Use Cases 3 and 5***
    - ***Updated Test Cases 1, 2, 4 and 7 to the correct variables and postconditions***

    Managerial / Administrative

    - ***Dynamic Pricing Test Cases changed to reflect inclusion of data range elements***
    - ***Removed Edit Layout Test Cases***
    - ***View Garage Statistics Test Cases changed to reflect database connection and the inclusion of user input***
    - ***View Garage Test Cases changed to reflect lack of user input***

    Elevator Operation

    - ***Updated Test Case 17 to make sure the user is redirected to the right page***
    - ***No longer includes the walk-in customer test option***

Customer Problem Statement

Our everyday experiences continue to be dominated by inefficiencies that can be remedied by modernization. One such experience is securing parking in an urban area. The process of parking or seeking parking is responsible for overflow traffic in streets surrounding the parking garage, wasted staff-hours, and time-consuming commutes. Factors like these make the automation of parking facilities a prime target for the development of efficient digital systems. Automated parking benefits garage owners and customers by optimizing time due to assigned parking spots, allowing advance reservations, avoiding congestion in the garage itself and making a more streamlined parking experience. A computerized garage facility requires fewer employees to achieve business optimization while giving customers an improved experience through easy-to-use booking systems.

Parking garage construction and usage are rising in step with urbanization throughout the US. Despite this increase in growth, parking garage revenue is expected to be sluggish due to high-interest rates and investor uncertainty as well as consumer sentiments about parking garages. According to IBISWorld, the anticipated parking garage industry revenue will reach 11.3 billion dollars by 2024, only a 0.6% increase in revenue from 2019. The strategy to combat sluggish revenue growth is to not only increase parking garage revenue but to improve user experience by making parking garages more efficient by automating garage processes.

In the traditional parking garage paradigm, customers enter the garage and receive a ticket, which will be used once they are finished parking to pay for usage of the spot. The customer drives through the garage searching for their own spot. If there are none available, then they are forced to drive back through the garage, once again searching for a spot or exiting the garage to look for a spot elsewhere, costing their own time and effort while the garage loses a customer. If multiple customers attempt this, it is possible that not only traffic will build up inside the garage itself but into the streets of the city, which further inconveniences customers that are on a tight schedule. An automated parking solution in which customers make reservations in advance, by either manual interaction with a web application or via voice assistant eliminates much of this frustration.

As of now, parking garage staff is required to manually check the occupancy status of parking spots throughout the garage by walking through it and making note of what spots are currently being taken up. As customers can leave the garage at any time, this means that either a parking garage staff may miss a departure while still considering that spot as occupied. The garage must have staff watching and analyzing each parking spot for departures to prevent this which is not feasible as this would require long staffing hours or a significant amount of staff members per garage. We wish to automate our parking garage process to eliminate excessive employee overhead and increase efficiency of garage operations.

Additionally, our parking garage management system does not account for relationship marketing. This is because all the tickets are taken at the front and then the customers park and go on about their lives. There is no way to track recurring customers or reserving parking spots for the customers in advance. We want our new solution to include customer data tracking and usage statistics. From this data, we would like to derive garage volume on specific dates/times and be able to make seasonal or otherwise targeted marketing efforts. This would further

improve the revenue by ensuring dependent revenue from expected customers from a business standpoint.

We want our garage system to interface with an easy-to-use web app and voice assistant. The ideal solution for us is one that allows customers to access their account details and make/alter reservations on their own time, from anywhere in the world. System administrators should be able to aggregate customer usage data and make actionable changes to the garage's operations to improve customer satisfaction. This feedback can give us more insight into our shortcomings and help us improve our garage for the consumer, making a more satisfying customer experience.

In order to further improve customer interaction with the proposed system, we would like to have the ability for the customer to use voice recognition to perform most tasks of the software hands-free. This will allow customers who are driving to the garage to safely make a reservation, without needing to take their eyes off of the road in the event another passenger is not present in the vehicle.

Presently, our pricing model is static meaning we have one price for every minute of every day. As mentioned above, the long term expectation for revenue growth in our industry is flat, yet demand is rising. As parking garage operators, we must develop tactics to combat this stagnant trend. We envision our system, including a dynamic pricing model that evaluates availability based on the projected number of spots reserved when the customer checks the availability for a given day. Fewer available places mean higher demand, and the price should adjust accordingly.

We would like a solution to address as many of the above concerns as possible. We want to eliminate the need for our customer to waste time driving for a spot and also eliminate the need for employees to monitor the garage at all hours. An automated system fulfills both these requirements. We would like our solution to be efficient and low-latency. Overall, the digitalization of a traditional parking garage could drastically improve the commuting experience, the overall customer experience, reduce garage overhead and provide an insight into the most profitable business strategy for the company.

Glossary of Terms

This is a list of frequently used terms throughout this report:

Administrator - The administrator is any authorized user of the privileged portions of the system including: Customer data, billing/payments, parking garage rules (this includes all employees of the garage)

Camera - Located at the entrance of the garage, this allows the elevator terminal to begin its interaction with a customer when a customer approaches the elevator.

Central Server (Database) - The database which stores login information, customer vehicles, information regarding the parking garage and the garage layout.

Confirmed Reservations - Represent registered customers who have made a reservation.

Contracted Customers - Customers/businesses that make a long-term reservation with the garage for a certain number of spaces on a daily, weekly, monthly or annual basis.

Customer / User - The customer is any user accessing either the online reservation system or entering the parking garage in a vehicle.

Customer Reservation System - A table in the database which stores data regarding customer's reservations

Dynamic Pricing - Flexible prices for products based on current market demands. Depending on time of day, day of the week, and whether the current day is a holiday or event causes the price to change for the spot.

Elevator/Lift- A compartment used to raise and lower vehicles to different levels of the parking garage. The elevator will also be responsible for taking in user information in the event a license plate is not recognized.

Elevator Console/Terminal - Keypad in the elevator that allows the customer to edit the end time of their reservation or enter their membership number (see below).

Garage Simulation - Accessed through the management portal, the garage simulation is a mock-up of what the SmartPark system will be able to accomplish.

Grace Period - Also known as the holding period, a short amount of time before and after which the customer is allowed to come to the garage or leave the garage.

Ground Level - The bottom-most level of the garage (at street level). Cars will not be parked here, but will rather be used for walk-in customers to make reservations.

Guaranteed Reservations - Allows customers to make a monthly contract with the parking garage for a parking spot or multiple parking spots.

License Plate Reader/Recognition - A device that can read and store license plate number data to keep track of vehicles in the garage. Specifically, the device will check in arriving vehicles and check out departing vehicles.

Management Portal - The location where the garage employee can login and view garage statistics, set dynamic pricing, and change the garage layout.

Membership Number - Unique identifier for each customer that can help find their reservation in case of a misread of license plate.

Monthly Tab / Running Tab - The form of billing that accounts for all transactions (charges, refunds, etc.) over the course of a month with only one net payment is made after the completion of the month.

Occupancy - The maximum number of objects that can fit in a designated area. Max occupancy will occur when all of the parking spots are filled.
Occupied Spots - Spots that have already been reserved by customers or are currently in use.
Overbooking - Accepting more reservations than there is room to account for ‘no-show’ customers.
Overstay - Currently parked customers who wish to extend their stay beyond the time for which they made reservations, either intentionally or unintentionally.
Passenger Vehicle - A regular-use vehicle that is allowed to park in the garage [no trucks, motorbikes, trailers, etc. are allowed to park].
Premium/Handicapped Spot - Parking spots that a customer can access whether they pay a premium fee if interested in the premium spot or spots for customers with disabilities for handicapped spots.
QR Code - A machine-readable code consisting of an array of black and white squares, used for storing URLs or other information for reading by cameras. The user will be able to scan a QR code in the elevator console in order to enter the information through their phone instead.
Rain Check - A token for customers that arrive with reservations but the parking garage is full. This will allow the customer to receive the same service they purchased but at a later date.
Registration Plate Number - License plate number; this is what the user will enter when signing up for the SmartPark service. The number will then be entered into the database and will then be recognizable by the license plate sensors.
Registration Time - The times the customer inputs as a boundary to their reservation.
Reservation Confirmation Number - The number a customer receives after they have made a reservation.
Reservation ID - number given to customer upon registration for a reservation to confirm
Reserved Period - The total time a parking spot is reserved for by a customer.
Standing Reservations - Recurring reservation from known customers.
Understay - Are customers who arrive on time but decide to leave before their predicted time of departure.
Vacant/Unreserved Spots - Spots that are available to be parked in/reserved
Voice Recognition - It is a computer software program or hardware device with the ability to decode the human voice. This will be used to allow customers to enter their information hands free.
Walk-In Customer - Are registered customers who arrive at the parking garage without a contract or reservation, or unregistered customers that can have a reservation be made for them.
Walk-In Terminal - Located on the ground floor of the garage. These terminals allow customers who do not have a reservation or do not have an account in the SmartPark system to create a reservation, an account, or both.

System Requirements by Subgroup

Customer Registration Subgroup Requirements

1. *Functional Requirements*

Table 1: Functional Requirements for Customer Group

Requirement Identifier	Size	Requirement
REQ: C-F-1	5 pts.	As a user, I should be able to create an account to store my demographic information for system operation.
REQ: C-F-2	5 pts.	As a user, I should be able to access my account to edit the information provided to ensure that the system has the most up-to-date information.
REQ: C-F-3 + REQ: C-F-4 + REQ: C-F-8	5 pts.	As a customer, I should be able to make a reservation.
		As a customer, I should be able to access my previously made reservations and update them for changes in time of reservation, changing the car that will be parking in the garage, etc.
		As a user, I should be able to pay for an extended grace period while I book my reservation.
REQ: C-F-5	4 pts.	As a contracted customer, I should be able to set up a weekly, monthly or annual contract for my business' customers.
REQ: C-F-6	3 pts.	As the system owners, we should be able to charge the customer on a monthly basis in a 'running tab' format that can be paid by check or online after the month is complete.
REQ: C-F-7	4 pts.	As a user, I should be able to access this monthly payment information, to view and pay it after the completion of the month.
REQ: C-F-9	4 pts.	As the system owners, we can charge the customer if they use the half an hour extension to their reservation in their monthly charge.
REQ: C-F-10	1 pts.	As system owners, we can charge the customer for their overstay time at an increasing rate, as informed by the elevator operation subgroup.
REQ: C-F-12	5 pts.	As a user, I should be able to use voice recognition in order to set up a reservation.

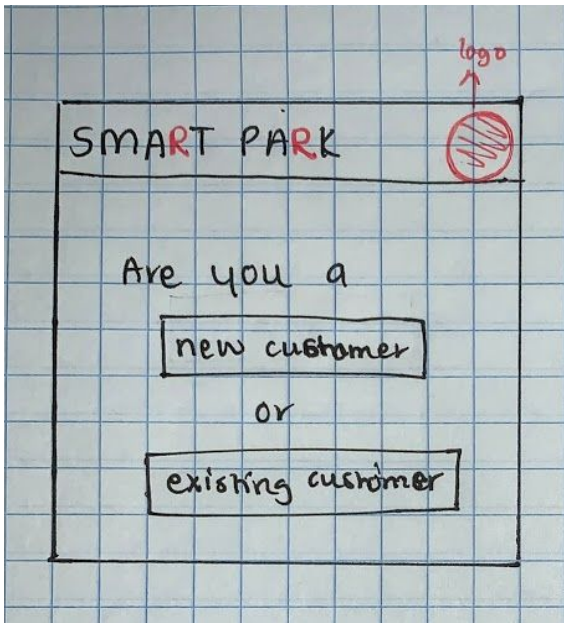
2. Non-Functional Requirements

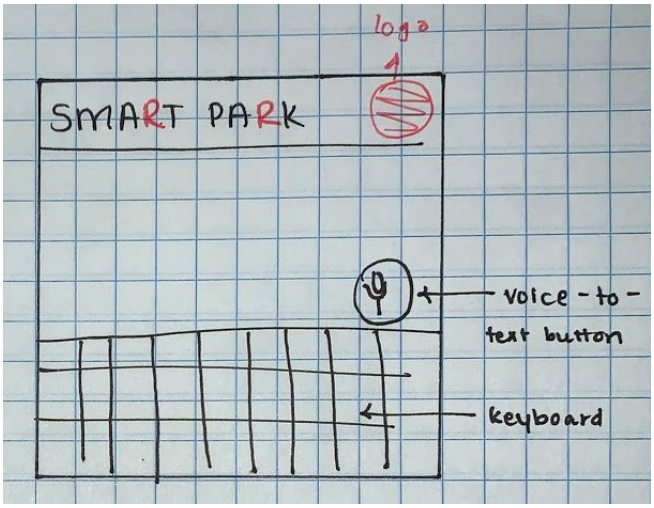
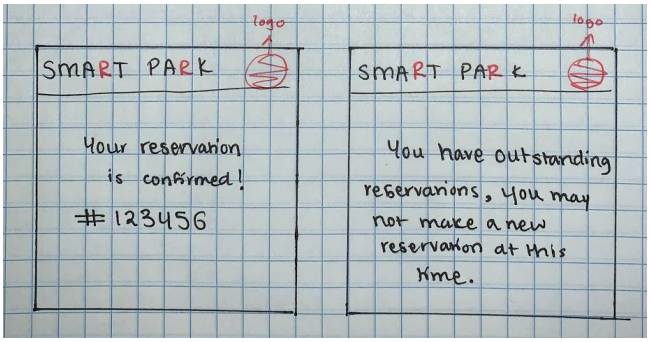
Table 2: Non-Functional Requirements for Customer Group

Requirement Identifier	Size	Requirement
REQ: C-NF-13 + REQ: C-NF-14	5 pts.	As a system, we can check the customer's reservations at the time of booking to ensure that there are no contiguous bookings under their name.
		As a system, we can check every time a user extends his/her reservation that they do not run into a continuous block of bookings after that.
		As a system, every time a new reservation is made, we check to see that the user making the reservation does not have more than 3 active reservations.
		As the system owners, we can have the customers agree to the 'terms and conditions' of our business policies.
REQ: C-NF-15	3 pts.	As system owners, we can bill the customer in the event of a 'no-show' to their reservation.
The above requirements are to ensure that there is no misuse of the system by the customers.		
REQ: C-NF-16	5 pts.	As a user, I want the rights to a 'rain-check' credit on my account, if the garage is unable to find a spot for my reservation.

3. User Interface Requirements

Table 3: UI Requirements for Customer Group

Requirement Identifier	Size	Requirement
REQ: C-UI-18	5 pts.	As a customer, I should be able to use the user interface easily to register and make reservations.
		As a customer, the number of data entries I have to enter to register and make a reservation should be minimal on my mobile device. 
REQ: C-UI-20	5 pts.	As a customer, I should be able to make reservations using my voice, while driving or in a moving vehicle.

		
REQ: C-UI-21	5 pts.	As a system operator, we should let the customer make a reservation and edit the times of their reservations on the mobile interface with the least amount of keystrokes/clicks for simplicity.
		As a system operator, we would let the customer perform sensitive tasks such as editing the account information, making a temporary registration association and accessing their ongoing tab and previous bills on the larger desktop interface. 

Managerial / Administrative Subgroup Requirements

1. *Functional Requirements*

Table 4: Functional Requirements for Manager Group

Requirement Identifier	Size	Requirement
REQ: M-F-1	5 pts.	As a parking garage administrator, I will be able to view statistics on historic garage occupancy, revenue, overbookings, overstays, and understays.
REQ: M-F-2	1 pt.	As a parking garage administrator, I will be able to view information on local events that may affect demand.
REQ: M-F-3	5 pts.	As a parking garage administrator, I will be able to change prices for reserved parking as well as fees for overstays.
REQ: M-F-5	2 pts.	As a parking garage administrator, I will be able to configure the average rates of arrivals and departures.
REQ: M-F-6	3 pts.	As a parking garage administrator, I will be able to see customer profiles and print transactions by this customer.
REQ: M-F-7	5 pts.	As a parking garage administrator, I will be able to register to login to my account.
REQ: M-F-8	5 pts.	As a parking garage administrator, I will be able to register and logout of my account.
REQ: M-F-9	4 pts.	As a parking garage administrator, I will be able to view and approve system generated rainchecks.
REQ: M-F-10	5 pts.	As a parking garage administrator, I will be able to view a monthly record of transactions.

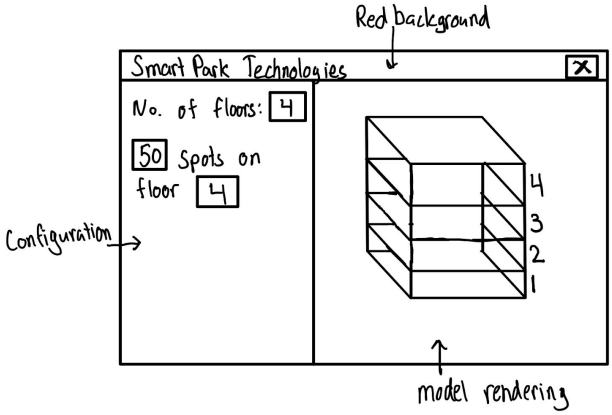
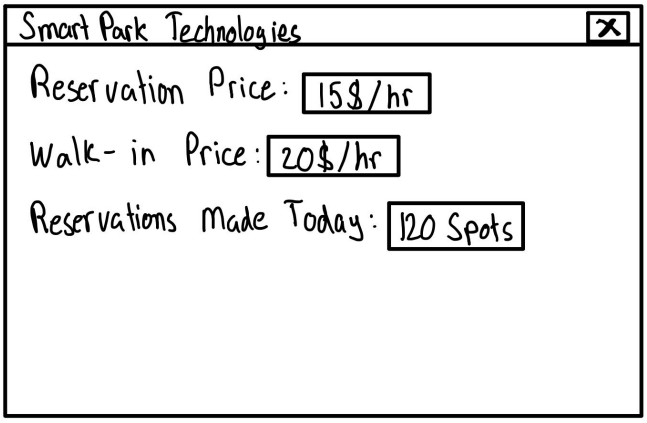
2. *Non-Functional Requirements*

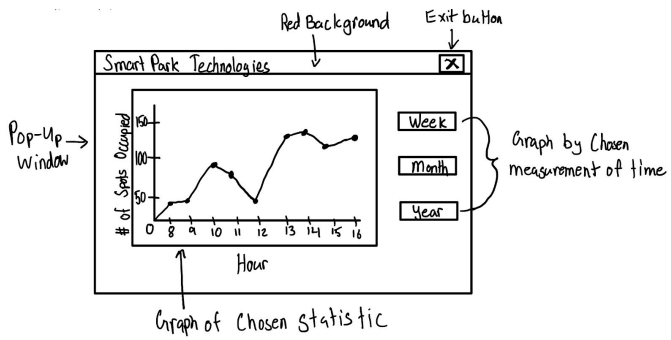
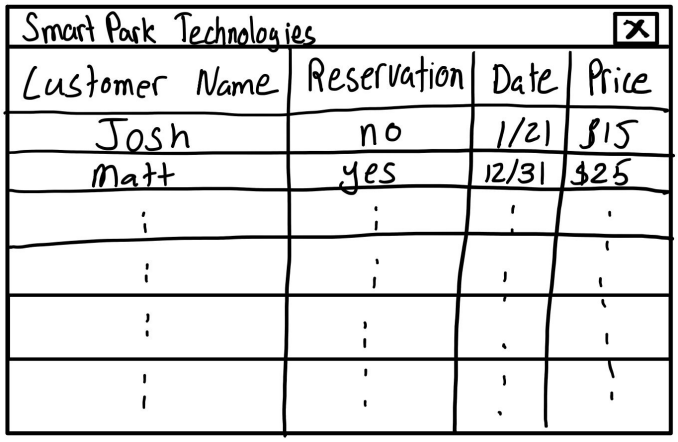
Table 5: Non-Functional Requirements for Manager Group

Requirement Identifier	Size	Requirement
REQ: M-NF-11	5 pts.	As a parking garage administrator, I will have a login page that verifies my credentials.
REQ: M-NF-12	4 pts.	As a parking garage administrator, I will be able to access separate interfaces where I can manage pricing and garage configuration.
REQ: M-NF-13	5 pts.	As a parking garage administrator, I will be able to search and select customer profiles.
REQ: M-NF-14	4 pts.	As a parking garage administrator, I will be able to select different parameters for viewing statistical information.
REQ: M-NF-15	4 pts.	As a parking garage administrator, I will be able to view a monthly summary of the garage's monetary business transactions.

3. User Interface Requirements

Table 6: UI Requirements for Manager Group

Requirement Identifier	Size	Requirement
REQ: M-UI-16	4 pts.	As a parking administrator, I will be able to see a visual representation of the garage for configuration. 
REQ: M-UI-17	4 pts.	As a parking administrator, I will have a dashboard to configure prices and manage or check on customer reservations. 
REQ: M-UI-18	5 pts.	As a parking administrator, I will be able to view a visual representation of garage occupancy statistics.

		
REQ: M-UI-19	4 pts.	As a parking administrator, I will be able to view system-generated monthly summaries of transactions. 

Elevator Operation Subgroup Requirements

1. *Functional Requirements*

Table 7: Functional Requirements for Elevator Group

Requirement Identifier	Size	Requirement
REQ: E-F-1	5 pts.	The system will keep track of occupied and available spots. This will be accomplished using weight sensors built into each of the parking spots in order to determine if a parking spot is occupied or not. The number of sensors required will change depending on the size of the garage.
REQ: E-F-2	5 pts.	The system will scan incoming license plates and reference the customer/reservation database for the next action.
REQ: E-F-3	5 pts.	The system will be able to obtain user membership info from the customer.
REQ: E-F-4	3 pts.	The system will alert customers if the garage is full and issue a rain-check notification to those who had reservations.
REQ: E-F-5	4 pts.	The system will ask users to register on the spot if they have not already done so.
REQ: E-F-6	1 pts.	The system will ask customers if they would like to extend their reserved period if they arrive after their grace period is up and there are vacant spots.
REQ: E-F-7	2 pts.	As the system operator I will allow customers to select a vacant spot of their liking in a given section of the garage as long as it is available.
REQ: E-F-8	5 pts.	Anyone will be able to access ground level walk-in terminals. No parking will occur on the ground floor.

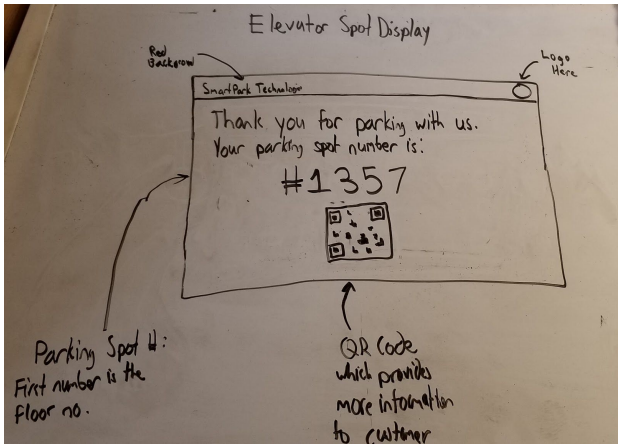
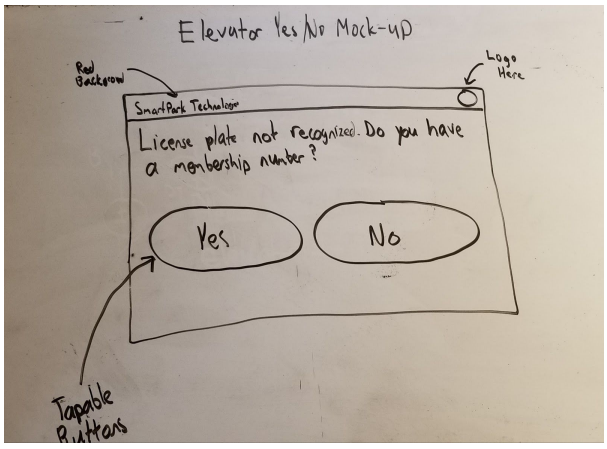
2. *Non-Functional Requirements*

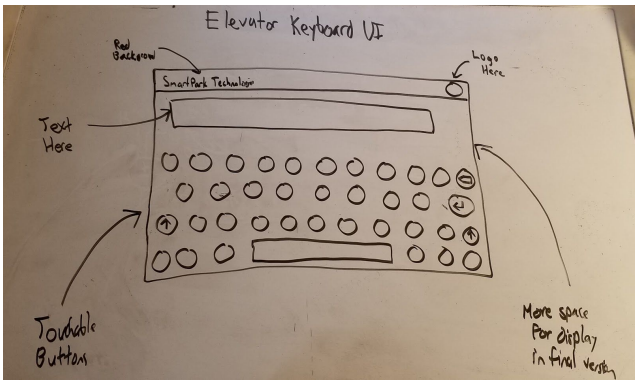
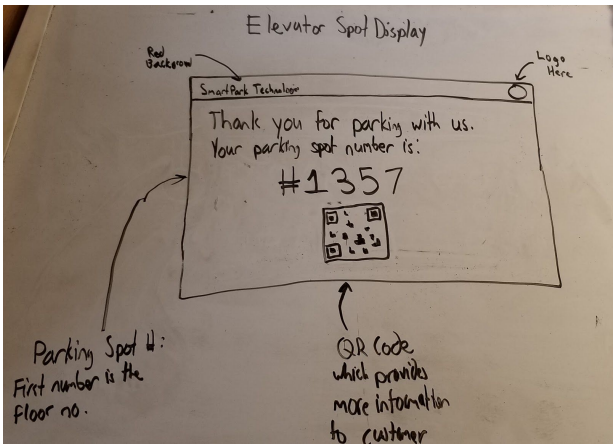
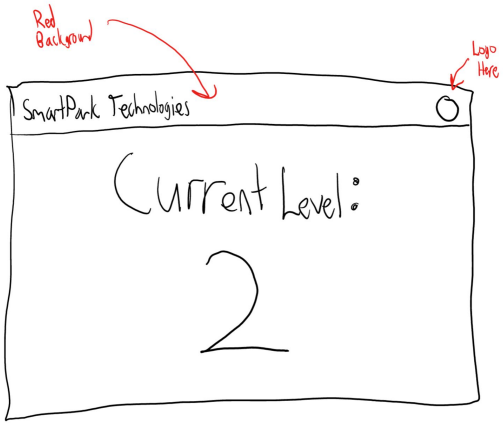
Table 8: Non-Functional Requirements for Elevator Group

Requirement Identifier	**Size**	**Requirement**
REQ: E-NF-9	5 pts.	As the system operator, I will be able to obtain user membership info from the customer.
REQ: E-NF-10	3 pts.	As the administrator, I will be able to access the database in order to account for reservation extensions.
REQ: E-NF-11	3 pts.	As the administrator, I will increase the billing rate for an overstay the longer the duration.
REQ: E-NF-dom	3 pts.	As the administrator, I will be able to overbook the parking garage.
REQ: E-NF-13	3 pts.	As the administrator I will give customers a rain check if there are no vacant spots.
REQ: E-NF-14	2 pts.	As the administrator I will be able to turn on/off dynamic pricing.

3. *User Interface Requirements*

Table 9: UI Requirements for Elevator Group

Requirement Identifier	**Size**	**Requirement**
REQ: E-UI-15	5 pts.	As a customer, I will be able to view my parking location and parking spot number by the elevator's screen. 
REQ: E-UI-16	5 pts.	As the customer, I will be able to respond yes/no when asked if I have an existing reservation. 
REQ: E-UI-17	4 pts.	As the customer, I will be able to use an on-screen keyboard in order to enter my account information or reservation number.

		
REQ: E-UI-18	3 pts.	As the customer, I will be able to scan a QR Code to get more detailed information about my parking spot. 
REQ: E-UI-19	2 pts.	As a customer, I will be able to see the current garage level I am on through the elevator's screen. This requirement was initially planned for the final demo but will not be implemented by then. 

Functional Requirement Specification

Stakeholders

The stakeholders of the project can be broken down into two different sub-groups: the subgroup that is using the parking garage and the subgroup that manages the parking garage.

Users of the Parking Garage

These stakeholders consist of anyone who uses the SmartPark service to make reservations and park in the garage. This can include:

- Customers
    - Returning customers or walk-ins, people who use the garage.
    - Can consist of only one person and/or groups of people, that share the account.
- Business Owners and Employees of that Business
    - May consist of walk-in or registered customers, but will typically be contracted.

Additionally, these stakeholders are contained in a few other sub-categories, such as:

- Walk-In Customers
    - Registered customers who arrive at the parking garage without a contract or reservation.
    - Unregistered customers that can have a reservation be made for them. This is done through the walk-in terminals in the ground floor of the parking garage.
- Contracted Customers
    - Customers and businesses that make a long-term reservation with the garage for a certain number of spaces on a daily, weekly, monthly or annual basis.
- Returning/Registered Customers
    - Customers that are registered in the SmartPark database who have either made a reservation in the past or are registered in the system.

Managers of the Parking Garage

These stakeholders consist of any staff members or other teams that work closely with the parking garage. This may include:

- Managers
    - Owners of the parking garage.
    - Will design the garage through the SmartPark graphical interface and monitor the garage's activity through an administrative portal.
    - Will be able to set prices of the garage (suggested based on a pricing model) and view trends such as garage usage, popular parking times, revenue/accounts, etc.
- Employees
    - Will be able to monitor the garage's activity to make sure everything runs smoothly and efficiently.
    - Using a modified view of the garage created by the managers and will be able to view spots that are currently occupied or not.

Actors and Goals

Table 10: Actors, Roles of Actors and Goals

Actor	Role of Actor	Actor's Goals
Customer	Initiating Actor	Register an account via the online portal. Make and cancel reservations to park. Enter the garage to park with as little system interaction as possible.
Camera	Participating Actor (supporting)	Notifies the elevator terminal when a car is approaching the elevator.
Employee of Garage/Manager	Initiating Actor	To view customer registration data, reservations, billing information. To be able to reconfigure the available parking spots with a GUI. View garage occupancy statistics and upcoming local events. Edit pricing and fees.
Parking Sensor	Participating Actor (supporting)	To send parking spot occupancy data to the parking spot management system.
License Plate Scanner	Participating Actor (supporting)	To send license plate numbers to the garage entry and exit systems.
Registration Terminal	Participating Actor	Provide an interface for the customer to interact with the registration, scheduling and billing portion of the system.
Elevator Terminal	Initiating + Participating Actor	To display which parking spot the customer should park in. Or to inform the customer they are not authorized to use the elevator by issuing a denial message.
Customer Reservation System	Participating Actor (supporting)	To accept customer login credentials and provide a GUI for creating or cancelling reservations. Also keeps track of the reservations customers have.
Central Server	Participating Actor (supporting)	To store information such as parking statistics, transaction histories, and garage layouts for administrative use.

Use Cases

a) Casual Description

Table 11: Use Cases and Descriptions

Use Case Identifier	Use Case Name	Description of Use Case
UC-1	Register	The user will be able to register for an account through the SmartPark website. The user will be asked to enter in their email address, a username, a password and other demographic information.
UC-2	Login	The user will be able to access their SmartPark account using their email address and password or their username and password.
UC-3	Edit Account	After logging into their account, the user will be able to edit or update any information regarding their account. This includes changing their email address, username, password,adding a new registered vehicle or other information.
UC-4	Make Reservation	The user will be able to make a reservation once logged in.
UC-5	Update Reservation	The user will be able to change the date and time of their reservation, or cancel it.
UC-6 (Future Work)	Make Contracted Reservation	The user will be able to make a contracted reservation for a parking spot.
UC-7	Pay Bill	The user can pay their bill through the online system.
UC-8 (Future Work)	Extend Grace Period	The user can extend their grace period provided there is no conflict with another reservation.
UC-9 (Future Work)	Create Garage Layout	The administrator of the system will be able to alter the arrangement and availability of parking spots through a GUI “drag and drop” system.
UC-10	Set Pricing	The administrator of the system will be able to set pricing for daily rates and contracted rates. This person can also turn on/off dynamic pricing.
UC-11	View Garage Information	The administrator and employees will be able to see the current layout of the parking garage, which spots are open/parked, etc.
UC-12	Issue Rain	The system shall determine whether an oversold event

(Future Work)	Check	has occurred and issue a rain check to a user who cannot be accomodated.
UC-13	Bill Customer	The administrator can login to the privileged section of the online portal and bill the customer.
UC-14	View Garage Usage Statistics	The administrator of the system can view usage statistics of the garage.
UC-15	Scan License Plate	The license plate scan is an essential part of the elevator operation. When entering the elevator, the customer's license plate is scanned in order to determine if the customer has a reservation or not. This is done through the camera in the elevator of the garage, which is assumed to be an already implemented component of the elevator.
UC-16	Voice Recognition	The customer may use an Alexa/Google device to create a new reservation.
UC-17	Park	The customer has initiated the parking sequence by entering the garage. The customer has the ability to park in two different ways: through the walk-in system or through a reservation. Through the walk-in system, the customer can create an account or a reservation or both. The pricing scheme for the walk-in customer will be based on the dynamic pricing generated by the system or by a set value from the managers of the garage. The cars will wait in line on the ground floor for their chance at the terminal. Once at the terminal, the customers make their reservation and then enter the elevator like any other customer. In the elevator, the customer's license plate is scanned to find the customer's reservation. When this fails, the customer is asked to enter a reservation number.
UC-18	Update Parking Spot Status	The parking spot is updated to either be occupied when a customer is using the spot or vacant when there is no vehicle in the spot. This is done through built-in weight sensors in each spot of the parking garage. When a car is parked in the spot, the weight sensor sends a signal to the database to mark the spot as occupied. Similarly, when a car exits a spot the weight sensor will detect that the car has left the spot and will then send a signal to the database to mark the spot as unoccupied.
UC-19	View Events	The administrator can view current events that may affect current and/or future garage occupancy.

b) Use Case Diagram

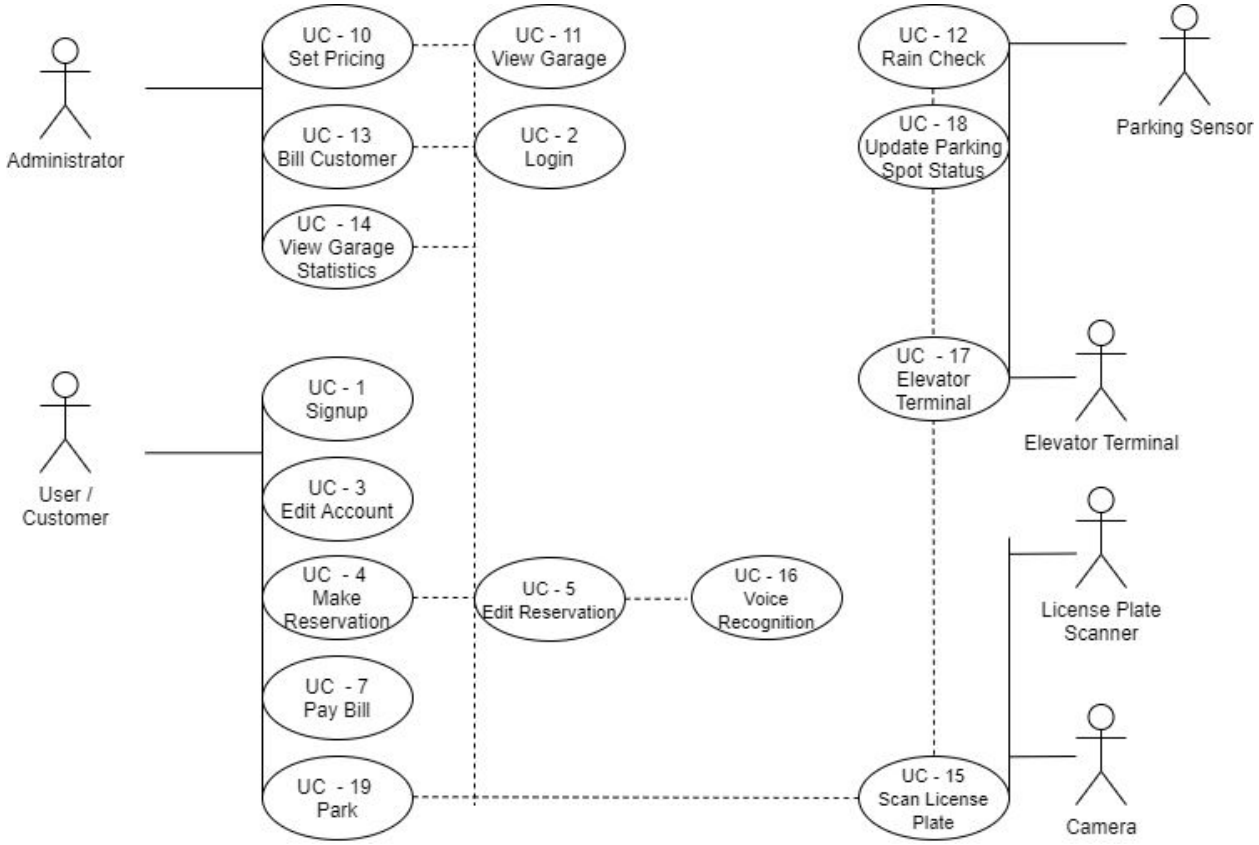


Image Label: Diagram of Use Cases and the interactions between them

c) *Traceability Matrix*

	UC ID	UC1	UC2	UC3	UC4	UC5	UC7	UC10	UC11	UC12	UC13	UC14	UC15	UC16	UC17	UC18	UC19
REQ ID	Ct.	4	4	1	4	4	4	7	4	3	7	8	4	6	3	8	3
C-F-1	1	X															
C-F-2	2		X	X													
C-F-3 C-F-4 C-F-8	3				X				X					X			
C-F-5	3				X	X								X			
C-F-6	3						X	X			X						
C-F-7	2						X	X									
C-F-9	4						X	X			X			X			
C-F-10	3						X	X			X						
C-F-12	4				X	X							X	X			
M-F-1	5							X				X	X		X	X	
M-F-2	4							X			X	X					X
M-F-3	4							X	X	X	X						
M-F-4	2								X			X					
M-F-5	2															X	X
M-F-6	1													X			
M-F-7	3	X	X									X					
M-F-8	3	X	X									X					
M-F-9	3									X	X						X
M-F-10	1										X						
E-F-1	2											X				X	
E-F-2	4											X	X		X	X	
E-F-3	1		X														
E-F-4	5								X	X		X			X	X	
E-F-5	3	X											X	X			
E-F-6	2					X										X	
E-F-7	3				X	X										X	
E-F-8	1															X	

Image Label: Matrix of Intersections between Use Cases and Functional Requirements of all subgroups

d) Fully Dressed Descriptions

Table 12: Fully Dressed Use Case 1 (Register)

Use Case UC-1:	**Register**
Related Requirements	REQ: C-F-1, REQ: M-F-6
Initiating Actors	Customer, Employee of the Garage
Participating Actors:	Customer, Employee of the Garage, Database for Customer Information
Actor's Goal	To register for the website by using a unique username, password, email address, and other information
Preconditions	The user attempting to create an account cannot use an email or username that is currently registered in the system.
Postconditions:	The user's login information will be saved into the database.
Flow of Events for Main Success Scenario	→1. The **Customer/Employee** selects the register account button on the main site page. → 2. The user is then asked to enter their email address, username, and password ← 3. The **Database for Customer Information** then saves the data regarding the user's account via Auth0
Flow of Events for Extensions (Alternate Scenarios):	2A. The user's login information (email, username) is already taken: - The system prompts the user stating that either the email address or username is already taken - The system prompts the user to select a new email address or username

Table 13: Fully Dressed Use Case 2 (Login)

Use Case UC-2:	**Login**
Related Requirements	REQ: C-F-2, REQ:M-F-7 REQ: E-F-3
Initiating Actors	Customer
Participating Actors:	Customer, Database for Customer Information
Actor's Goal	To access their SmartPark account using their email address and password or their username and password.
Preconditions	The user must have already registered to the website using a unique username, password, email address, and other information
Postconditions:	The user can now make a reservation, edit an existing reservation or edit account information
Flow of Events for Main Success Scenario	→ 1. The user is asked to enter their email address / username, and password ← 2. The Database for Customer Information verifies the login credentials via Auth0 ← 3. The user is brought to a page where they can choose one of three buttons; making a new reservation, editing an existing reservation or editing account information.
Flow of Events for Extensions (Alternate Scenarios):	3A. The user chooses to make a new reservation [proceed to **UC-4**] 3B. The user chooses to edit an existing reservation [proceed to **UC-5**] 3C. The user chooses to edit account information [proceed to **UC-3**]

Table 14: Fully Dressed Use Case 3 (Edit Account)

Use Case UC-3:	**Edit Account**
Related Requirements	REQ: C-F-2, REQ:M-F-7 REQ: E-F-3
Initiating Actors	Customer
Participating Actors:	Customer, Database for Customer Information
Actor's Goal	To edit their SmartPark account with new / changed information
Preconditions	The user must have already registered to the website using a unique username, password, email address, and other information
Postconditions:	The user's information is updated
Flow of Events for Main Success Scenario	→ 1. The user selects an attribute that they want to update and uses the textbox to type and submit it ← 2. The Database for Customer Information verifies the credential that has been changed and updated it accordingly ← 3. The user is brought to the home page, where they can choose to make/edit a reservation or edit their account again
Flow of Events for Extensions (Alternate Scenarios):	2A. The user's attribute is not valid - The system prompts the user stating that the attribute is incorrect and prompts them to try again

Table 15: Fully Dressed Use Case 4 (Make Reservation)

Use Case UC-4:	**Make Reservation**
Related Requirements	REQ: C-F-3, REQ: E-F-1, REQ: E-F-5
Initiating Actors	Customer
Participating Actors:	Customer, Employee of the Garage, Manager
Actor's Goal	To make a reservation in the garage through the online portal
Preconditions	The user must have already registered to the website using a unique username, password, email address, and other information required to login. They must also have a payment method on their account.
Postconditions:	The user will receive confirmation that they successfully made a reservation by receiving a message on the screen they made the reservation on
Flow of Events for Main Success Scenario	→ 1. The user is asked to enter their email address, username, and password if they have not done so already. ← 2. The Database for Customer Information verifies the login credentials. → 3.The user is brought to a page that requires input as to what day and time the user would like to choose for their reservation ← 4. The system checks the available times for the day and time that the user selected and makes the reservation accordingly by giving the user a unique identifier to confirm their reservation.
Flow of Events for Extensions (Alternate Scenarios):	4A. There are no available spots for the date picked - The user is automatically sent to the next available date with open slots and asked if they wish to reserve a spot then

Table 16: Fully Dressed Use Case 5 (Edit Reservation)

Use Case UC-5:	**Edit Reservation**
Related Requirements	REQ: C-F-3, REQ: E-F-1, REQ: E-F-5
Initiating Actors	Customer
Participating Actors:	Customer, Employee of the Garage, Manager
Actor's Goal	To make a reservation in the garage through the online portal
Preconditions	The user must have already registered to the website using a unique username, password, email address, and other information required to login. They must also have a payment method on their account.
Postconditions:	The user will receive confirmation that they successfully updated their reservation by receiving a message on the screen they made the reservation on.
Flow of Events for Main Success Scenario	→ 1. The user is asked to enter their email address, username, and password if they have not done so already. ← 2. The Database for Customer Information verifies the login credentials. ← 3. The system prompts the user to enter their reservationID → 4. The user is brought to a page that asks for what they want to change about the reservation and type it in ← 5. The system updates the reservation for the ID that was given
Flow of Events for Extensions (Alternate Scenarios):	3A . The reservationID is invalid and the user must try again 4A. There are no available spots for the date picked - The user is automatically sent to the next available date with open slots and asked if they wish to reserve a spot then

Table 17: Fully Dressed Use Case 7 (Pay Bill)

Use Case UC-7:	**Pay Bill**
Related Requirements	REQ: C-F-6, REQ: C-F-7, REQ: M-F-3
Initiating Actors	Customer
Participating Actors:	Customer, Manager
Actor's Goal	To pay off any existing bills incurred from reservations, overstays, etc.
Preconditions	The user must already have an account and be logged in with a payment method on their account
Postconditions:	The user will receive confirmation that they have successfully paid their bill and receive a receipt for the transaction which will be emailed to the email associated with their account
Flow of Events for Main Success Scenario	→ 1. The user is asked to enter their email address, username, and password if they have not done so already. ← 2. The Database for Customer Information verifies the login credentials ← 3. The user is brought to a page that pulls up the bills associated with their account → 4. The user is able to pay off their bill with either the payment method on file or they can type in a credit or debit card number, expiration date, and cvv. ← 5. The user is shown shown a message that they paid their bill successfully and a receipt is sent to the email address they have on their account
Flow of Events for Extensions (Alternate Scenarios):	4A. The payment information is invalid - The user is prompted to re-enter in the attribute that was incorrect 5A. The bill status does not updated (system error) - The user is asked to reload the page and try again

Table 18: Fully Dressed Use Case 10 (Set Pricing)

Use Case UC-10:	**Set Pricing**
Related Requirements	REQ: C-F-6, REQ: C-F-7, REQ: C-F-8, REQ: C-F-9, REQ: C-F-10, REQ: M-F-1, REQ: M-F-2, REQ: M-F-3
Initiating Actors	Manager
Participating Actors:	Database for Pricing, Customers
Actor's Goal	Change the pricing for hourly, overstay, and peak-time parking as well as the cancellation fee
Preconditions	The user must be logged in and have administrative privileges.
Postconditions:	The data will be updated in the pricing table in the database
Flow of Events for Main Success Scenario	→ 1. The Manager enters their desired price for hourly, peak-time, and overstay parking as well as inputs a cancellation fee. → 2. The Manager then selects their desired start date. Note that the Manager cannot select dates that are in the past. → 3. The Manager presses the submit button after they are finished making changes. ← 4. The system updates the pricing in the database for all reservations for the corresponding dates. ← 5. The system displays a notification that the change was successful.
Flow of Events for Extensions (Alternate Scenarios):	→ 3. The manager presses the cancel button before they are finished making changes. ← 4. The system makes no updates and the pricing reverts to its original value.

Table 19: Fully Dressed Use Case 11 (View Garage Information)

Use Case UC-11:	**View Garage Information**
Related Requirements	REQ: C-F-3, REQ: C-F-11, REQ: M-F-3, REQ: E-F-4
Initiating Actors	Employee of Garage/Manager
Participating Actors:	Central Server
Actor's Goal	The administrator and employees will be able to see the current layout of the parking garage, which spots are open/parked, etc.
Preconditions	The user must be logged in and have administrative privileges.
Postconditions:	Not applicable.
Flow of Events for Main Success Scenario	→ 1. The manager logs in to the Manager Access Portal and navigates to the view garage overview or reservations page. → 2. The manager enters their search parameters for viewing information and presses the search button. ← 3. The system retrieves the information from the database. ← 4. The system renders the information in the user interface.
Flow of Events for Extensions (Alternate Scenarios):	Not applicable.

Table 20: Fully Dressed Use Case 14 (View Garage Usage Statistics)

Use Case UC-14:	**View Garage Usage Statistics**
Related Requirements	REQ: C-F-3, REQ: C-F-11, REQ: M-F-3
Initiating Actors	Manager
Participating Actors:	Parking Database
Actor's Goal	To view statistics for revenue, occupancy and overstays for a specific time period.
Preconditions	The Manager must be logged in and have admin access.
Postconditions:	N/A
Flow of Events for Main Success Scenario	→ 1. The Manager clicks on the statistics tab. → 2. The Manager enters the desired time period to view statistics. ← 3. The Database retrieves the requested data for the given time slot. ← 4. The system renders the data in a graphical interface.
Flow of Events for Extensions (Alternate Scenarios):	Not applicable.

Table 21: Fully Dressed Use Case 17 (Park)

Use Case UC-17:	**Park**
Related Requirements	REQ: M-F-1
Initiating Actors	Elevator Terminal
Participating Actors:	Customer
Actor's Goal	To park in the Parking Garage
Preconditions	The consumer may or may not have a reservation. There needs to be an open spot on the ground level if the customer is a walk-in. There needs to be an open spot in general if it is a car with a reservation.
Postconditions:	If the customer is a walk-in, they will get a ticket and a parking spot on the ground floor. If the consumer already has a reservation, then they are allowed to park if there is a spot. However if they have a reservation, and there is no space they are given a raincheck. If they have walked in and there is no space, they are not given a spot in the parking garage.
Flow of Events for Main Success Scenario	→ The customer enters the parking garage → The license plate reader reads and either recognizes the plate number from an existing database of reservations or does not find it in the database. → If it does not find it in the database, it asks for a reservation number. If there is no reservation number then it is treated as a walk in customer. → If a reservation is found in the system with that number, then the customer is led onto next steps for parking. → Otherwise the car is taken to the open spot, if they have a valid reservation.
Flow of Events for Extensions (Alternate Scenarios):	→ If the license plate is recognized, then there is a check for an open spot. → If there is an open spot the customer is led to the spot, otherwise they are given a raincheck.

Table 22: Fully Dressed Use Case 18: Update Parking Spot Status

Use Case UC-18:	**Update Parking Spot Status**
Related Requirements	REQ: M-F-1, M-F-5, E-F-1, E-F-2, E-F-4, E-F-6, E-F-7, E-F-8
Initiating Actors	Elevator Terminal
Participating Actors:	Customer, Employee of the Garage/Manager, Customer Reservation System
Actor's Goal	To update a parking spot's current status as either occupied or vacant.
Preconditions	Not applicable.
Postconditions:	The parking spot's status will be updated as either vacant or occupied.
Flow of Events for Main Success Scenario	→1. The customer's vehicle enters or exits the parking spot. 2. The **Parking Sensor** will sense the vehicle either entering or leaving the spot. ← 3. The **Parking Sensor** will update the database which stores information about each parking spot.

System Sequence Diagrams

Customer Registration Subgroup

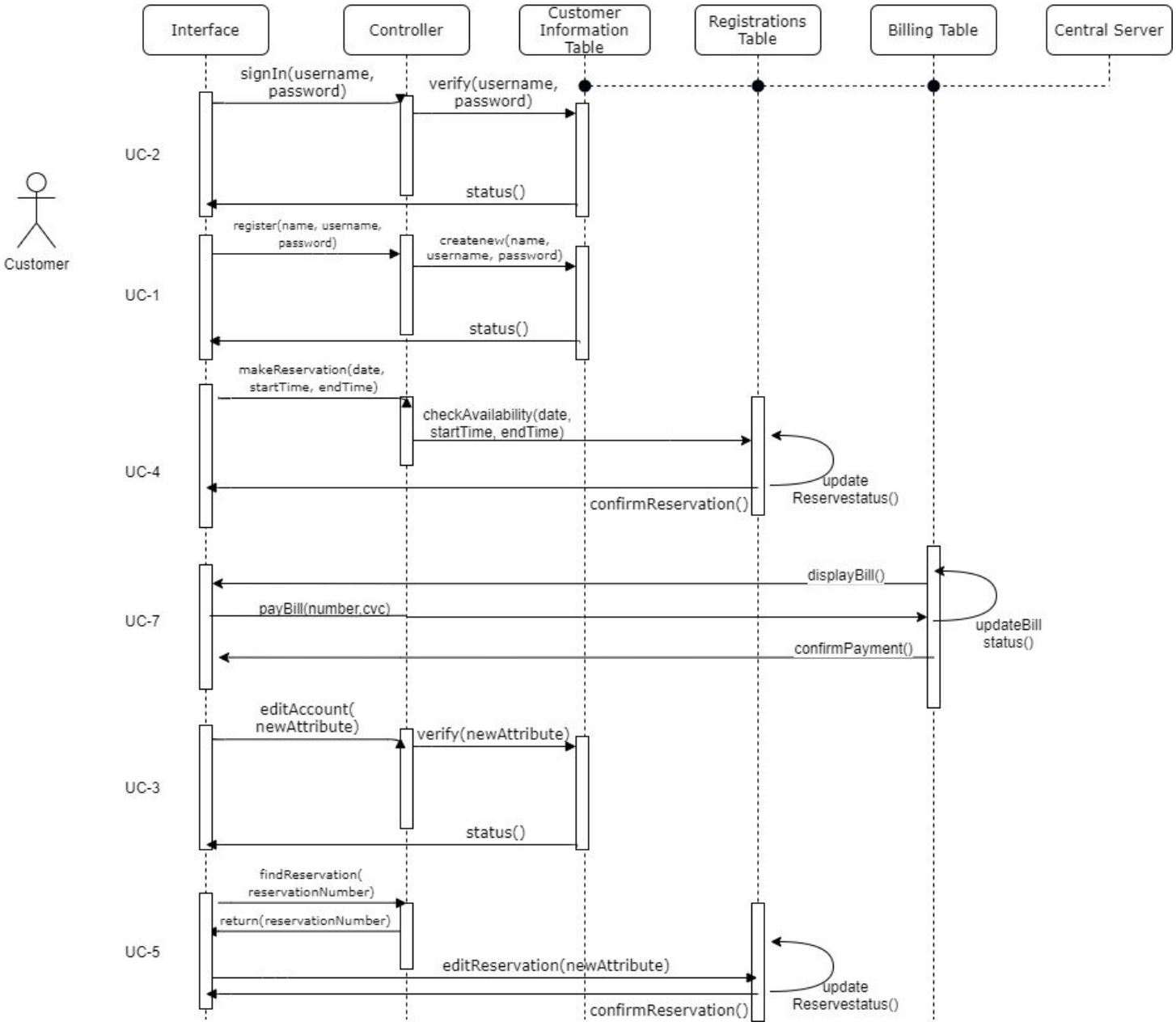


Image Label: System Sequence Diagram for Customer Interaction with SmartPark Services

The above diagram shows the system sequence diagrams for all the use cases being implemented by this subgroup. We show the successful scenarios of the complete list of use cases that we plan to have implemented by the final demo; 1,2,3,4,5 and 7.

Managerial / Administrative Subgroup

Use Case #9 - Set Prices

This use case allows the manager to manually enter prices for the garage hourly and overstay rates. Dynamic pricing is also available as an additional feature. The manager enters the prices they want into the portal, which calls the Price Changer API. The Price Changer then updates the values in the database

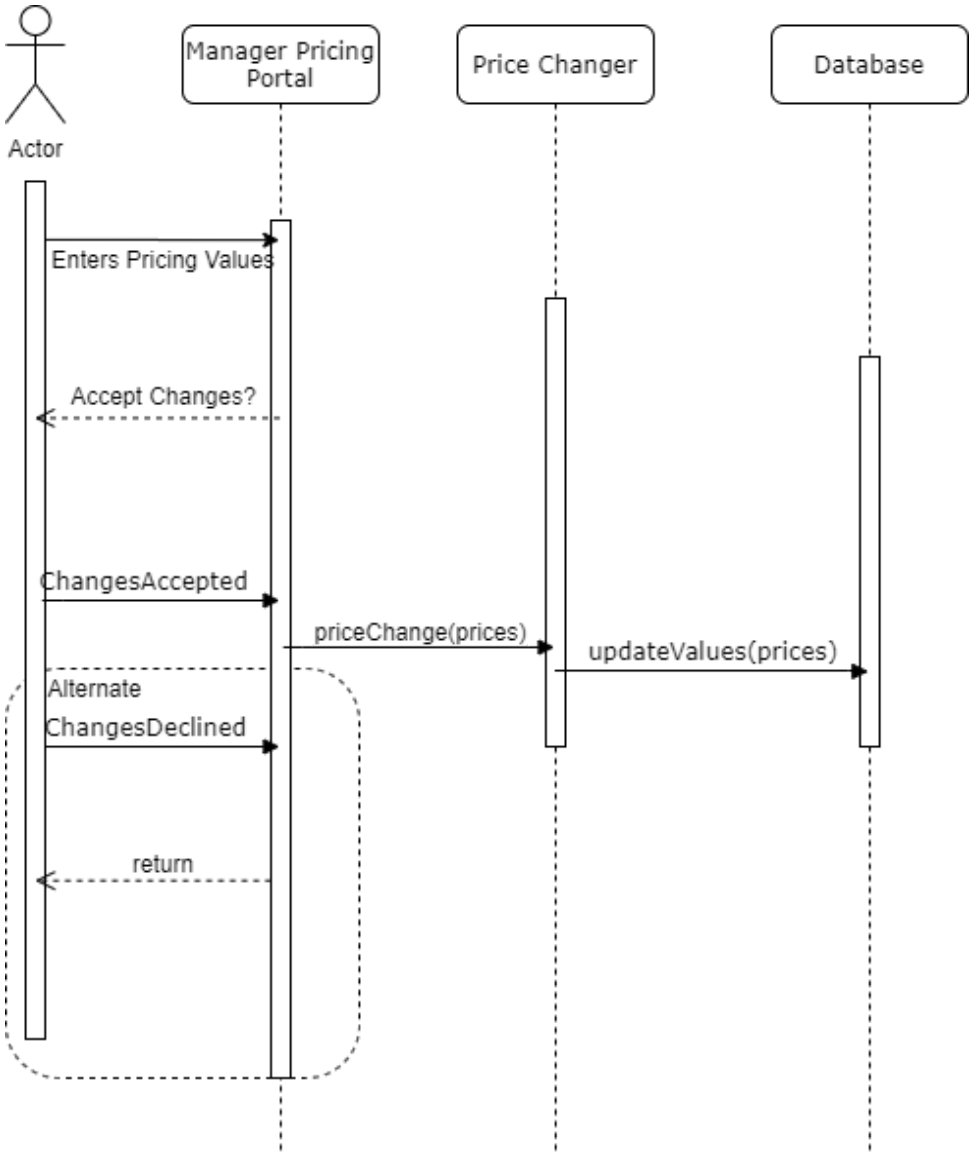


Image Label: System Sequence Diagram Manager Use Case 10

Use Case #11

This use case allows the manager to view garage information using search parameters such as vacancy status or customer reservation time. In our implementation, there are two APIs that make up the InfoRetriever, one deals with spots and one deals with reservations. When the user enters information for spot availability, the Spot API is used. On a separate page, the manager can view reservation details for all reservations in the database. In this case, the Reservation API is used.

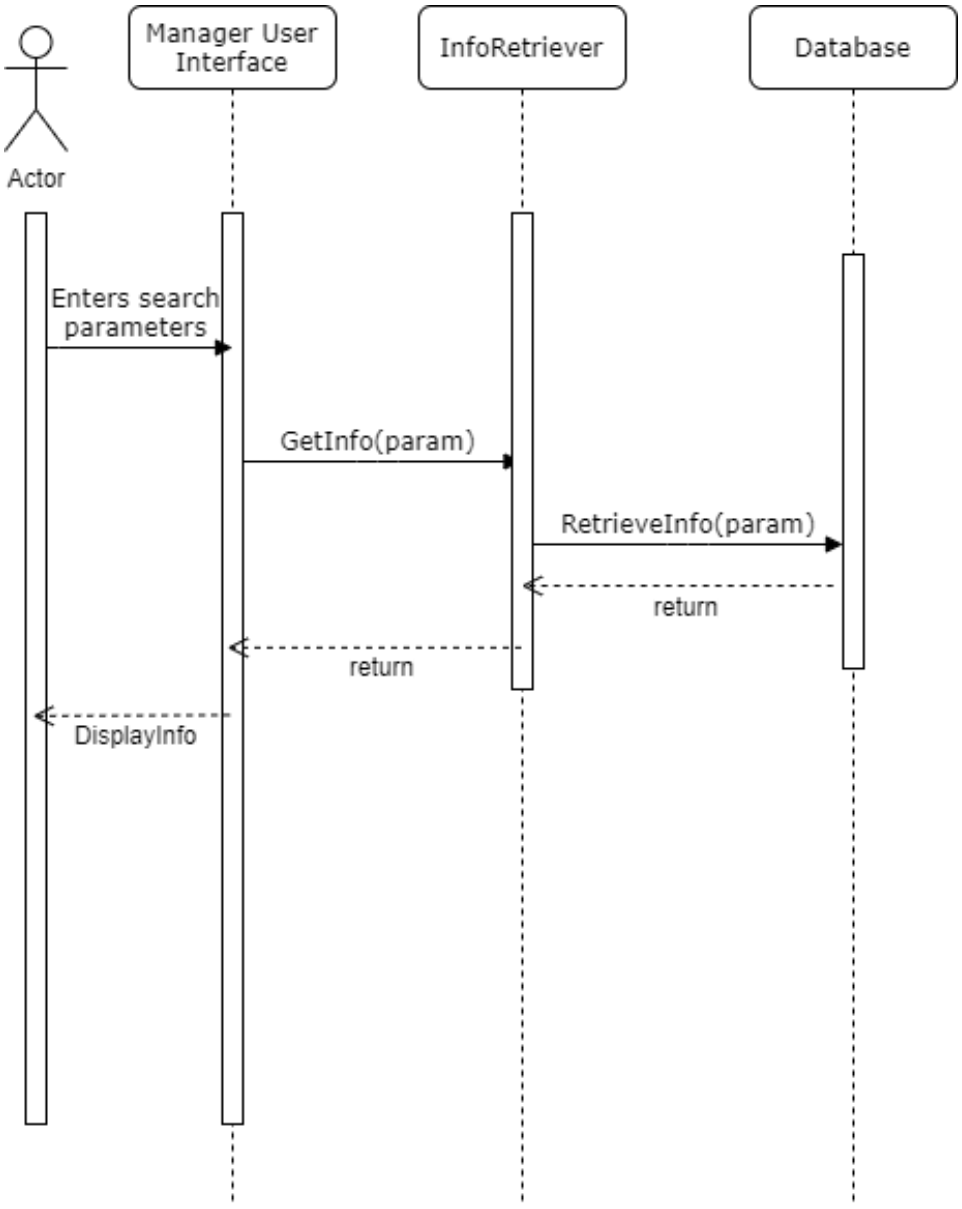


Image Label: System Sequence Diagram Manager Use Case 11

Use Case #14: View Statistics

This use case allows the manager to view occupancy statistics for the garage. The manager can enter dates and the API will query the database accordingly. The user interface will then update the graphs as desired.

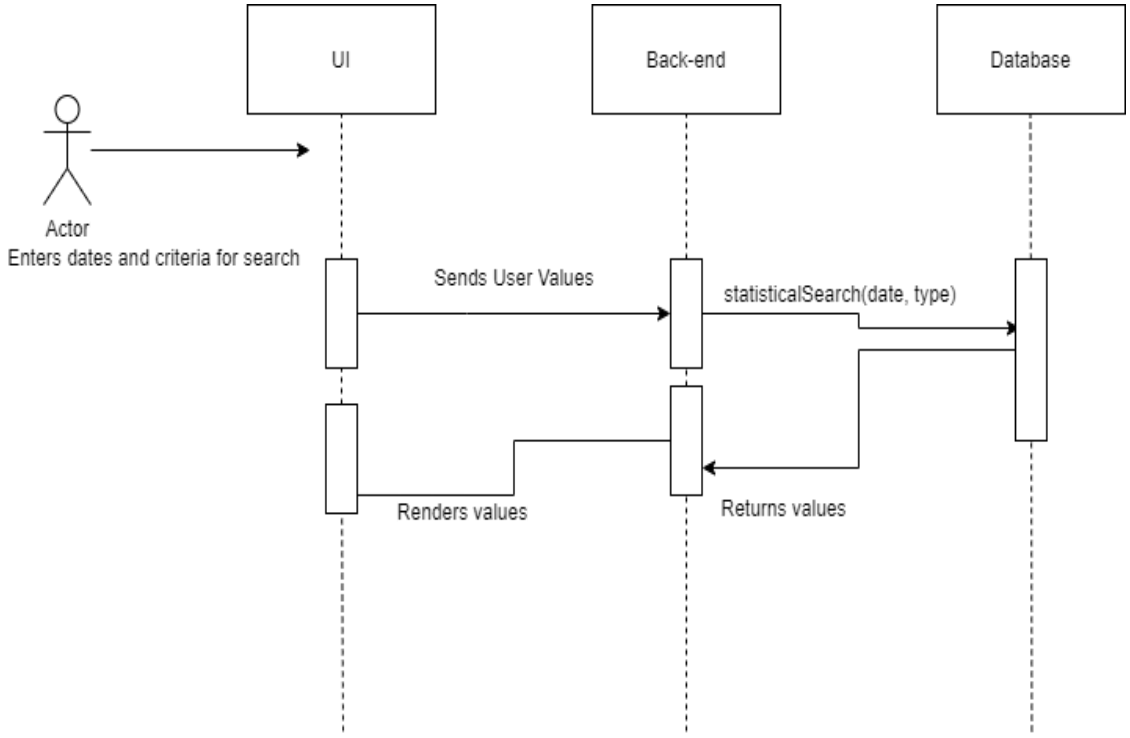


Image Label: System Sequence Diagram Manager Use Case 14

Elevator Operation Subgroup

Use Case #17: Park

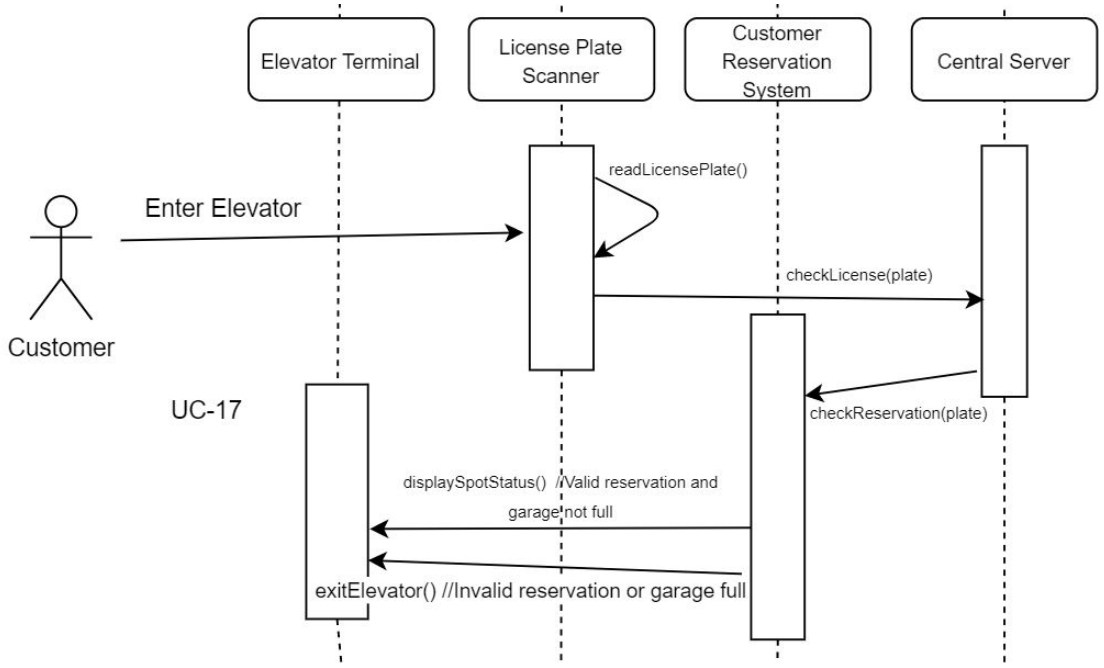


Image Label: System State Diagram for when the license plate is successfully scanned

The above system sequence diagram is for Use Case 17: Park. This is the flow of events that occur in the system when the license plate was able to be scanned successfully, and includes the cases where the customer is able to park - through displaySpotStatus()- or when the customer is unable to park - through exitElevator(). The customer begins by entering the elevator. When the customer enters the elevator, the license plate scanner reads the license plate of the user. On success, the license plate information is sent to the database and is checked to see if it is in the system. If it is in the system, then the account associated with the license plate is checked for a reservation. If the reservation is valid and the garage is full, then the spot status is displayed in the elevator terminal. Otherwise, the garage may be full, or the reservation or license plate may not be valid. In the event that a reservation was found and the garage is full, then the customer is asked to exit the elevator and a rain check is issued. Otherwise, the reservation is not valid and the customer is simply asked to exit the elevator. The customer can then return to the ground floor and create a reservation through the walk-in terminal.

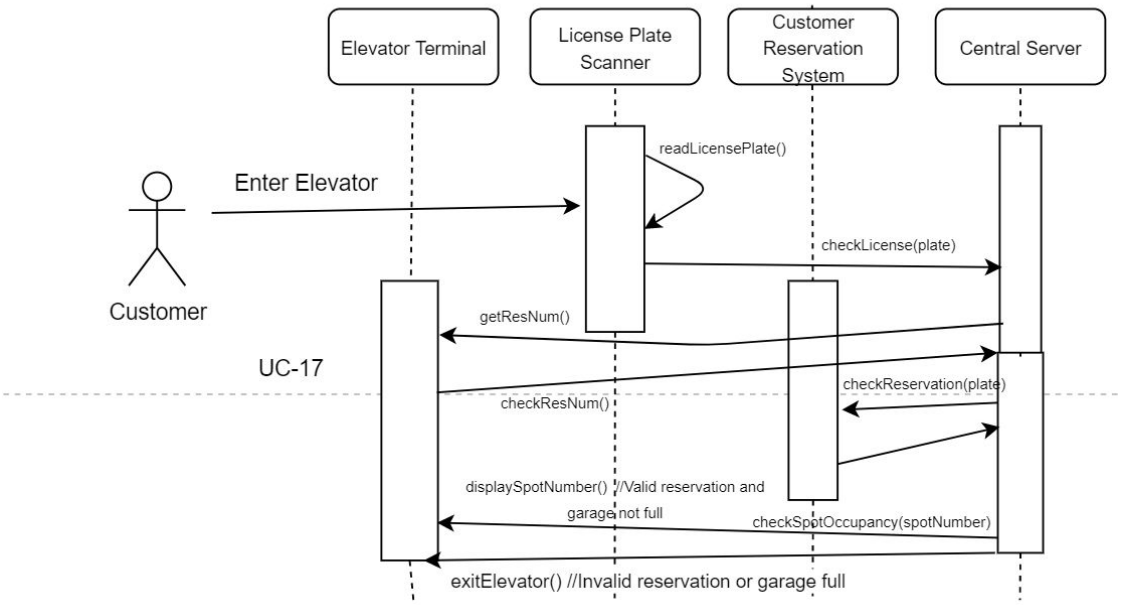


Image Label: System State Diagrams for an unsuccessful license plate scan.

If the license plate scan was unsuccessful, then more interaction with the user is necessary. The user enters the elevator and attempts to read the license plate of the customer. An attempt is made in the central server to check the license plate, however this check is unsuccessful. The elevator terminal then prompts the user for a reservation number. After receiving the reservation number, the number is checked in the system. If the reservation number is valid, then a check for the reservation is made, and the flow of events plays out the same as if the license plate was able to be scanned. If the reservation number is invalid, then the customer is asked to exit the elevator, and the customer can make a reservation at the walk-in terminal on the ground floor.

Use Case #18: Update Parking Status

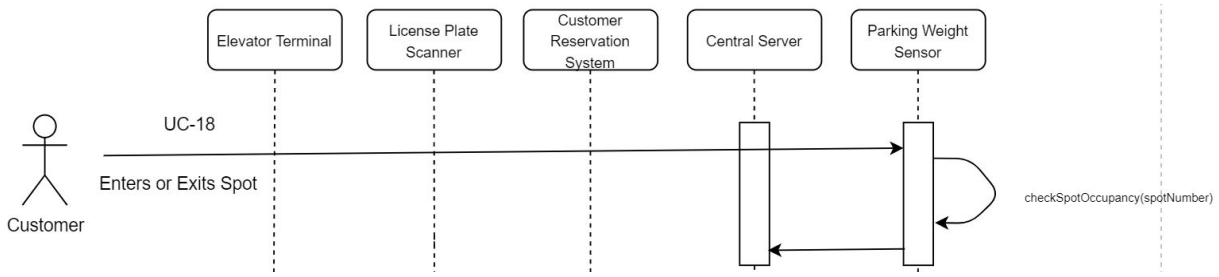


Image Label: System State Diagram showing the process of updating a parking spot.

Updating the parking status consists of a communication between the central server and the parking weight sensor. When the customer enters or exits a spot, the weight in the spot sensor changes, and sends a signal to check the spot occupancy. When this change occurs, then the new status is sent to the central server (occupied, unoccupied). This newly updated status can then be used elsewhere in the system, including as a check in the elevator for spot occupancy or for a customer creating a reservation, for example.

Effort Estimation using Use Case Points

Elevator Operation Subgroup

In order to determine the effort estimation that the elevator subgroup will have taken by the end of the project, the equation to determine duration is employed. The equation for duration is:

$$Duration = UCP * PF$$

Where PF is given to be 28 hours per use case point. The use case points are determined through the following equation:

$$UCP = UUCP * TCF * ECF$$

UUCP, or Unadjusted Use Case Points, measures the complexity of the functional requirements. This is made up of two different components, the Unadjusted Actor Weight (UAW) and the Unadjusted Use Case Weight (UUCW). The TCF, or Technical Complexity Factor, measures the complexity of the nonfunctional requirements. Lastly, the Environment Complexity Factor (ECF) assesses the development team's experience and their development environment. A breakdown of the calculations for these values for the elevator group are detailed below.

The first calculation performed is the unadjusted actor weight (UAW). The actor weights are based on a scale from simple to complex, where the weight of a simple operation is worth 1 point, up to complex which is worth 3 points. Simple operations are for system components interacting with one another through an API, average is interaction through a text based interface for users or network communication with systems, and complex is a person interacting through a user interface. The actors and their respective unadjusted actor weights for the elevator are shown below:

Table 23: Effort Estimation for the Elevator Actors

Actor	Description of Relevant Characteristics	Complexity	Weight
Customer	The customer interacts with the elevator terminal through a graphical user interface to make decisions.	Complex	3
Camera	The camera is a system which interacts with our system through an API.	Simple	1
Elevator Terminal	Same as the camera.	Simple	1
License Plate Scanner	Same as the camera.	Simple	1

Parking Sensor	Same as the camera.	Simple	1
Customer Reservation System	The Customer Reservation System is a system which interacts with our other systems through a protocol.	Average	2
Central Server	The Central Server is a system which interacts with our other systems through a protocol.	Average	2

Then, the UAW for the elevator group system is:

4*Simple + 2*Average + 1*Complex = 4*1 + 2*2 + 1*3 = 11.

The Unadjusted Use Case Weight is calculated next, also having three different categories: simple worth 5 points, average worth 10 points, and complex worth 15 points. Simple use cases consist of one participating actor and number of steps for completion less than or equal to 3. Average use cases consist of two or more participating actors, with a number of steps ranging from 4 to 7. Lastly, the complex category consists of three or more participating actors, with steps for completion ranging from 7 onwards. The use cases and their complexities, along with a description of the use cases are shown below:

Table 24: Effort Estimation for the Elevator Use Cases

Use Case	Description	Complexity	Weight
Use Case 15: Scan License Plate	The license plate is scanned using the license plate scanner and the information is then forwarded to the central server. This includes one initiating actor (the license plate scanner) and one participating actor (the central server). The number of steps for completion is 2: scan the license plate and send the information to the central server.	Simple	5
Use Case 17: Park	The use case for the entirety of the parking process. Consists of 1 initiating actor, the customer, and 6 participating actors. All of the actors in the above UAW calculation are used. The steps for completion vary depending on the user's choices in the elevator terminal. The best case scenario number of steps is four steps: receiving the license plate information in the front end, performing a check to see if the license plate has a valid reservation in the customer reservation system, receiving the reservation data in the front end, and then outputting the reservation data in the front end. Worst case scenario interaction	Complex	15

	consists of 8 steps: letting the user know that no license plate could be scanned in the front end, prompting the user to enter a reservation number in the front end, receiving the reservation number in the customer reservation system, checking the number, sending the information associated to that reservation number to the front end, determining the garage is full through a check in the central server, issuing a rain-check which requires the customer reservation system to be updated and finally giving the customer this information in the front end.		
Use Case 18: Update Parking Status	Same as use case 15, but instead the interaction is between the parking spot weight sensor and the central server.	Simple	5

Then the UUCW is calculated as the UAW was calculated before:

*2*5 + 1*15 = 25*

The UUCP is then computed by summing these two together: 11 + 25 = 36.

The TCF is calculated using the thirteen standard technical factors identified by expert developers. The list of these are shown below, along with their respective weights:

Table 25: Technical Factors and Their Descriptions

Technical factor	Description	Weight
T1	Distributed system (running on multiple machines)	2
T2	Performance objectives (are response time and throughput performance critical?)	1(*)
T3	End-user efficiency	1
T4	Complex internal processing	1
T5	Reusable design or code	1
T6	Easy to install (are automated conversion and installation included in the system?)	0.5
T7	Easy to use (including operations such as backup, startup, and recovery)	0.5
T8	Portable	2
T9	Easy to change (to add new features or modify existing ones)	1
T10	Concurrent use (by multiple users)	1
T11	Special security features	1
T12	Provides direct access for third parties (the system will be used from multiple sites in different organizations)	1
T13	Special user training facilities are required	1

The technical factor of the TCF is found by multiplying the weights given by the perceived complexity of each of the technical factors. Then, the TCF is calculated by using the following formula:

$$TCF = \text{Constant-1} + \text{Constant-2} \times \text{Technical Factor Total} = C_1 + C_2 \cdot \sum_{i=1}^{13} W_i \cdot F_i$$

Where (constant - 1) = 0.6 and (constant - 2) = 0.01. The technical factors for the elevator group is shown below:

Table 26: Calculating Technical Factors Weight for Elevator

Technical Factor	Description	Weight	Perceived Complexity	Calculated Factor (Weight * Perceived Complexity)
T1	The elevator terminal and the database/central server need to run on two different devices. This is because the central server and the customer reservation system is a shared component across all groups, while the elevator terminal is just for display in the elevator.	2	2	2*2 = 4
T2	Because the elevator system relies on real-time information, such as cars entering and exiting the spots, response time and throughput performance is critical. This is to prevent any incorrect data from entering the elevator system, which could lead to incorrect occupancy values for the garage, which in turn could lead to incorrect pricing models based on the dynamic pricing model as well as decreased revenue for garage owners.	1	5	1*5 = 5
T3	The elevator terminal needs to be efficient for the customer in order to keep the lines of customers waiting to park down.	1	3	1*3 = 3
T4	Processing information is relatively straightforward, mostly consists of reading and sending data.	1	1	1*1 = 1
T5	For the real system, the need for reusability	1	2	1*2 = 2

	is extremely necessary in order to provide the service in other garages. However, adding the system to other locations would not be as complex as it would be installing it.			
T6	Ease of install is important, as the system will be implemented in many different garages.	0.5	4	0.5*4 = 2
T7	In order to keep the customer satisfied and the garage flowing at a good pace where revenue increases, the ease of use on the customer end is extremely important.	0.5	5	0.5*5 = 2.5
T8	No major portability concerns.	2	1	2*1 = 2
T9	A simple update in order to change the system, no need to change the hardware of the elevator once implemented.	1	1	1*1 = 1
T10	No need for concurrent use as only one customer will be able to interact with the elevator terminal at a time.	1	0	1*0 = 0
T11	Since there is no way for a customer to retrieve their information directly - or others - then security is not a major concern. No billing takes place through the elevator.	1	2	1*2 = 2
T12	No access for third parties.	1	0	1*0 = 0
T13	No training necessary.	1	0	1*0 = 0

Therefore the total technical factor for the elevator group is 24.5. Then, using the formula as mentioned above, the TCF is determined to be: 0.845. This means that the UCP will be reduced by 8.45%.

Lastly, to calculate the ECF of the project, the 8 provided environmental factors are used. The table of factors and their respective weights are shown below:

Table 27: Environmental Factors and Their Descriptions

Environmental factor	Description	Weight
E1	Familiar with the development process (e.g., UML-based)	1.5
E2	Application problem experience	0.5
E3	Paradigm experience (e.g., object-oriented approach)	1
E4	Lead analyst capability	0.5
E5	Motivation	1
E6	Stable requirements	2
E7	Part-time staff	–1
E8	Difficult programming language	–1

Each factor can have a value of 0 through 5. A value of 0 has no impact on success, a value of 1 means the factor has a strong, negative impact on the project, a value of 3 means an average impact, and a value of 5 means a strong, positive impact on the project. Ultimately, the larger the number means a larger amount of experience which positively impacts the effort estimation. As with the TCF, the ECF is calculated using a formula, given below:

$$ECF = \text{Constant-1} + \text{Constant-2} \times \text{Environmental Factor Total} = C_1 + C_2 \cdot \sum_{i=1}^{8} W_i \cdot F_i$$

where,

Constant-1 (C_1) = 1.4

Constant-2 (C_2) = –0.03

The environmental complexity factors for the elevator system of the project are shown below:

Table 28: Calculating Environmental Factors Weight for Elevator

Environmental Factor	Description	Weight	Perceived Impact	Calculated Factor (Weight * Perceived Impact)
E1	The UML-based development process was new to the members of the elevator group.	1.5	1	1.5*1 = 1.5
E2	After reviewing how the parking garage process works, the elevator group felt comfortable with the application program.	0.5	2	0.5*2 = 1
E3	Due to experience with object-oriented programming in prior classes, the	1	4	1*4 = 4

	elevator group felt confident with this paradigm.			
E4	The lead analyst was a beginner.	0.5	1	0.5*1 = 0.5
E5	The elevator group for the majority of the project was extremely motivated and willing to work on the project.	1	4	1*4 = 4
E6	The requirements of the project have stayed stable.	2	4	2*4 = 8
E7	No part-time staff was used in making this project.	-1	0	-1*0 = 0
E8	Coding in React proved to be around average difficulty after understanding how React functioned.	-1	3	-1*3 = -3

Therefore the total environmental factor for the elevator group is 16. Then, using the formula as mentioned above, the TCF is determined to be: 0.92.

To reiterate, the formula used to calculate UCP is:

$$UCP = UUCP * TCF * ECF$$

The values obtained for the UUCP, TCF, and ECF for the elevator group were 36, 0.845, and 0.92, respectively. Then, by multiplying these numbers together the UCP obtained is:

$$36*0.845*0.92 = 27.9864$$

Finally, the duration can be found by taking the obtained UCP above and multiplying by the productivity factor. The productivity factor was given to be 28. Multiplying the numbers together, the result for the duration is:

$$27.9864*28 = 783.6192 \text{ hours}$$

The duration is found to be around 784 hours. Because the elevator group is composed of four people, and we assume that each person worked an equal number of hours, then each individual will need to work on the project for about 196 hours. According to the textbook, a developer will work on the project for around 30 hours a week. This means that the project for each individual will take around 6.5 weeks to complete.

Customer Registration Subgroup

Following the descriptions set by the Elevator Operation Subgroup, we will only show the final calculations in the following sections.

Unadjusted Actor Weight (UAW)

The relevant actors that interact with the customer registration are shown below:

Table 29: Effort Estimation for Customer Actors

Actor	Description of Relevant Characteristics	Complexity	Weight
Customer	The customer interacts with the elevator terminal through a graphical user interface to make decisions.	Complex	3
Customer Reservation System	The Customer Reservation System is a system which interacts with our other systems through a protocol.	Average	2
Central Server	The Central Server is a system which interacts with our other systems through a protocol.	Average	2
Registration Terminal	The Registration Terminal is a system that bridges the information translation between the Customer and the Central Server.	Complex	3

UAW = (Average*2) + (Complex*2) = (2*2) + (3*2) = 4 + 6 = 10

Unadjusted Use Case Weight (UUCW)

The relevant use cases that our subgroup deals with are:

Table 30: Effort Estimation for Customer Use Cases

Use Case	Description	Complexity	Weight
Use Case 1: Register	This is the component of our website that allows the customer to create an account on the SmartPark website that will store the information pertaining to that customer in a table in the Central Server.	Simple	5
Use Case 2: Login	This allows the customer to log into the site with their previously created account, this gives them access to all of the SmarPark services.	Simple	5
Use Case 3: Edit Account	After the account is created, if the customer wishes to change the information that SmartPark has including their email, password, vehicle information and credit card information.	Simple	5

Use Case 4: Make Reservation	This use case deals with many components including the Registration Terminal, Central Server and the Customer inputs. The customer selects the dates and times between which they want to make a reservation. The terminal then checks the Central Server and creates it and displays a confirmation.	Complex	15
Use Case 5: Edit Reservation	This is similar to the previous use case but requires the customer to input their confirmation number and the aspect of the reservation that they would like to change. Then, update the entry in the Central Server.	Average	10
Use Case 7: Pay Bill	This works the same as the previous test case except the current total of the bill is displayed and the user/customer can enter their credit card information and pay the bill online. The Terminal then needs to update the Central Server accordingly.	Average	10

UUCW = (Simple*3) + (Average*1) + (Complex*2) = (5*3) + (10*2) + (15*1)
= 15 + 10 + 15 = 40

Unadjusted Use Case Points (UUCP)

UUCP = UAW + UUCW = 10 + 40 = 50

Technical Complexity Factor (TCF)

Table 31: Calculating Technical Factors Weight for Customer

Technical Factor	Description	Weight	Perceived Complexity	Calculated Factor
T1	The Customer Registration System is accessed on multiple computers via the deployed app interface.	2	2	4
T2	The response time for the database and backend as of now are not a big cost in time of response but may be subject to change due to the size of the table	1	2	2
T3	The customer interface is set up to to be the most user-friendly and is optimized to have the least user input and minimal keystrokes. This was heavily emphasized during the development phase.	1	5	5
T4	This system only deals with known inputs to the system, there are a limited number of	1	3	3

	versions of the input that the system can receive from the user.			
T5	We used React with the Bootstrap library, this allowed us to search for ready-made components that we wanted to integrate and allowed us to focus on how to implement the most simplicity in the interface.	1	2	2
T6	It would be available to any customer who accesses the website.	0.5	1	0.5
T7	The website is set up in a straightforward and easy to use fashion.	0.5	2	1
T8	It is developed in react bootstrap which automatically configures the size of the device, this makes it easier to view on your mobile device as well as your computer.	2	1	2
T9	Any component can be swapped out as a long as we have the ability to save the current state that is manipulated by the user.	1	1	1
T10	This system can account for multiple user usage by having separate accounts for all the customers	1	1	1
T11	Securing and encrypting the data of the customers' credit card information	1	3	3
T12	No third parties garages would have access to the SmartPark system	1	0	0
T13	No special training required for users	1	1	1
TCF Total				25.5

TCF = (Constant-1) + (Constant-2) (TCF Total) = 0.6 + (0.01)(25.5)
= 0.6 + 0.255 = 0.855

Environmental Complexity Factor (ECF)

Table 32: Calculating Environmental Factors Weight for Customer

Environmental Factor	Description	Weight	Perceived Impact	Calculated Factor

E1	Not very familiar with UML based development	1.5	1	1.5
E2	No previous experience with real life problem solving	0.5	1	1
E3	Average level of paradigm experience	1	3	3
E4	Working as a separate yet cohesive group proved to be of some difficulty	0.5	2	1
E5	Strong drive to do well for the betterment of the group's combined hard work	1	4	4
E6	The requirements of the project have stayed stable.	2		
E7	No part-time staff was used	-1	0	0
E8	Although coding in React initially had a learning curve, we did manage to design most web pages and the backend before revising it all for the benefit of the connecting between the front and back end	-1	3	-3
ECF Total				7.5

ECF = (Constant-1) + (Constant-2)(ECF Total) = 1.4 + (-0.03)(7.5)
= 1.175

Use Case Points (UCP)

UCP = UUCP * TCF * ECF
= (50)*(0.855)*(1.175)
= 50.23

Total Duration

UCP * PF = 50.23 * 28
= 1406.44

The duration is found to be around 1406 hours. Because the customer group is composed of three people, and we assume that each person worked an equal number of hours, then each individual will need to work on the project for about 469 hours through the semester. We also assume that each team member works on the project for 30 hours a week. This means that every group member will finish the project in 15.6 weeks.

Managerial / Administrative Subgroup

Following the descriptions set by the Elevator Operation Subgroup, we will only show the final calculations in the following sections.

Unadjusted Actor Weight (UAW)

The relevant actors that interact with the customer registration are shown below:

Table 33: Weights for Actors for Manager Group

Actor	Description of Relevant Characteristics	Complexity	Weight
Manager	The customer interacts with the elevator terminal through a graphical user interface to make decisions.	Complex	3
Manager Access Portal	The Manager Access Portal is a system which interacts with our other systems through a protocol.	Average	2
Central Server	The Central Server is a system which interacts with our other systems through a protocol.	Average	2

UAW = (Average*2) + (Complex*1) = (2*2) + (3*1) = 4 + 3 = 7

Unadjusted Use Case Weight (UUCW)

The relevant use cases that our subgroup deals with are:

Table 34: Weights for Use Cases for Manager Group

Use Case	Description	Complexity	Weight
UC-10: Set Pricing	The use case requires interaction with text and numerical fields. This is an average actor interacting with a system that would require about 4-5 steps in the case that the manager changes the value.	Average	10
UC-11: View Garage	The use case involves an average actor with 1-2 steps of interaction with the system. The system would then retrieve and display the data as the actor requested.	Simple	5
UC-13: Bill Customer	The administrator, in this case an average actor,	Simple	5

	locates the customer for which they would like to generate a report. The system then retrieves and displays the data requested.		
UC-14: View Garage Usage Statistics	The use case is initiated by a parking administrator that interacts with basic data fields. The system then retrieves the data and displays the information in the format requested. This process would be greater than 4 steps depending on the parameters selected.	Complex	15

UUCW = (Simple*2) + (Average*1) + (Complex*1) = (5*2) + (10*1) + (15*1)
= 10 + 10 + 15 = 35

Unadjusted Use Case Points (UUCP)

UUCP = UAW + UUCW = 7 + 35 = 42

Technical Complexity Factor (TCF)

Table 35: Calculating Technical Factors Weight for Manager

Technical Factor	Description	Weight	Perceived Complexity	Calculated Factor
T1	The Manager Portal is accessed on multiple computers via the deployed app interface.	2	2	4
T2	The response time for the database and backend as of now are not a big cost in time of response but may be subject to change due to the size of the table	1	2	2
T3	The manager portal has a clean and intuitive user interface. During development, useability and aesthetics were emphasized.	1	5	5
T4	This system only deals with known inputs to the system, there are a limited number of	1	3	3

	versions of the input that the system can receive from the user.			
T5	We used React with the Bootstrap and MDBReact libraries which provide ready-made code for components that we wanted to integrate. This allowed us to focus on how to provide the best possible interface using the tools we had.	1	2	2
T6	It would be available to any manager that can access the portal with valid credentials	0.5	1	0.5
T7	The website is set up in a straightforward and easy to use fashion.	0.5	2	1
T8	It is developed in React Bootstrap/MDBReact which automatically configures the size of the device, this makes it easier to view on your mobile device as well as your computer.	2	1	2
T9	Any component can be swapped out as a long as we have the ability to save the current state that is manipulated by the user.	1	1	1
T10	This system can account for multiple user accounts by having separate accounts for all the customers	1	1	1
T11	Only managers can access the pages	1	3	3
T12	No third parties garages would have access to the SmartPark system	1	0	0
T13	No special training required for users	1	1	1
TCF Total				25.5

TCF = (Constant-1) + (Constant-2) (TCF Total) = 0.6 + (0.01)(25.5)
= 0.6 + 0.255 = 0.855

Environmental Complexity Factor (ECF)

Table 36: Calculating Environmental Factors Weight for Manager

Environmental Factor	Description	Weight	Perceived Impact	Calculated Factor

E1	Not very familiar with UML-based development	1.5	1	1.5
E2	No previous experience with real life problem solving	0.5	1	1
E3	Average level of paradigm experience	1	3	2
E4	Working as a separate yet cohesive group proved to be of some difficulty	0.5	1	0.5
E5	Strong drive to do well for the betterment of the group's combined hard work	1	4	4
E6	The requirements of the project have stayed stable.	2	4	8
E7	No part-time staff was used	-1	0	0
E8	Using the MERN stack proved to be challenging at first, but we were able to use it effectively	-1	3	-3
ECF Total				14

$$ECF = (Constant\text{-}1) + (Constant\text{-}2)(ECF\ Total) = 1.4 + (-0.03)(14)$$
$$= 0.98$$

Use Case Points (UCP)

$$UCP = UUCP * TCF * ECF$$
$$= (42)*(0.855)*(0.98)$$
$$= 35.12$$

Total Duration

$$UCP * PF = 35.12 * 28$$
$$= 985.37$$

The duration is found to be around 985.4 hours. Because the manager group is composed of three people, and we assume that each person worked an equal number of hours, then each individual will need to work on the project for about 450 hours through the semester. We also assume that each team member works on the project for 30 hours a week. This means that every group member will finish the project in about 11 weeks.

Domain Analysis

Domain Model

The domain model was created by using the use cases and requirements of the parking garage specifications. The functional requirements and actors were analyzed and then transformed into different concepts for the parking garage. With this vast amount of concepts, we ensure that no one concept has too many responsibilities.

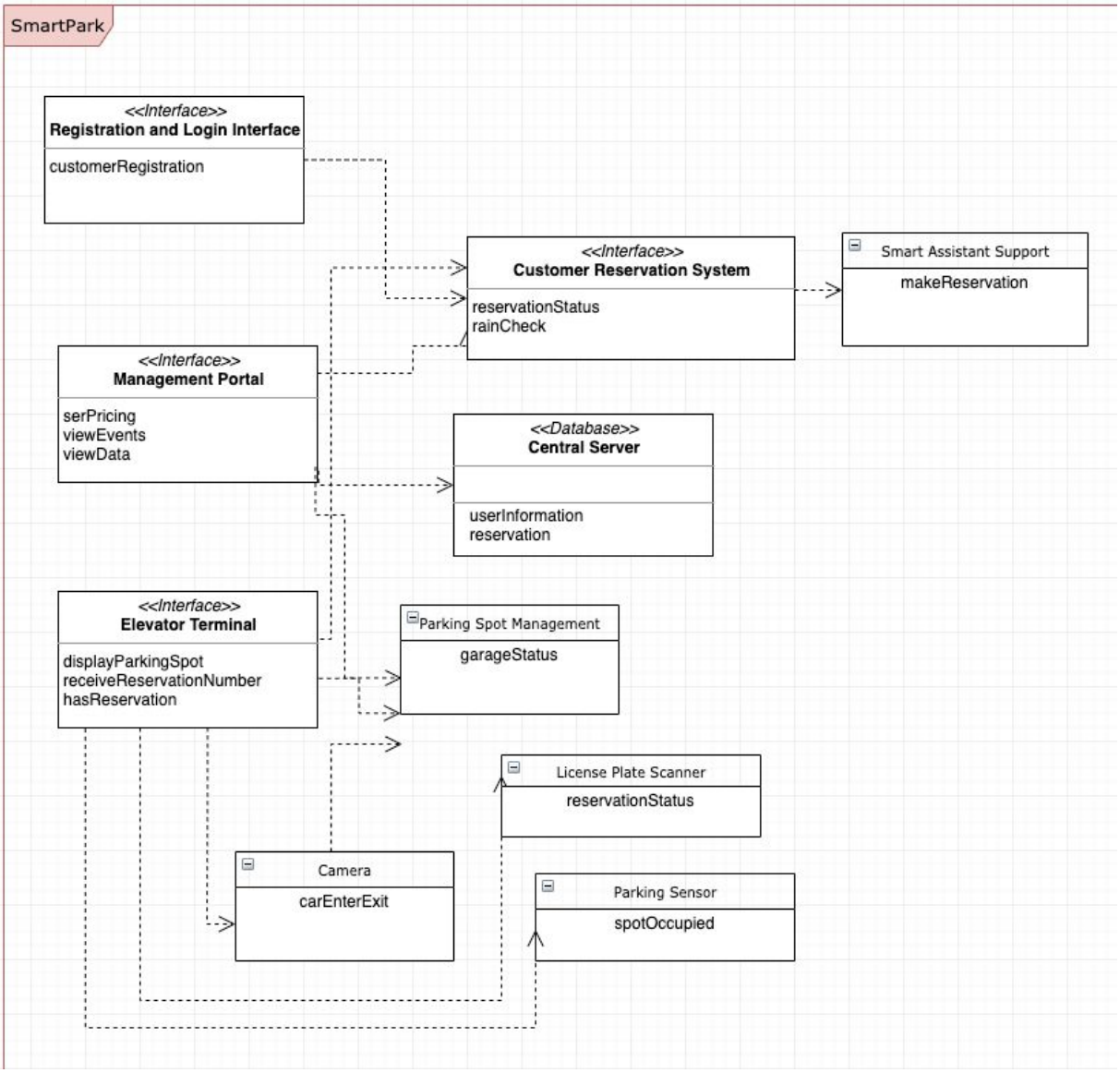


Image Label: Domain Diagram for entire project

a) Concept Definitions

Table 37: Concept Definitions

Concept Identifier	Responsibility Description	Type	Concept
DC-1	● Sends information regarding a parking spot's status to the parking management system. ● Determines when a vehicle is using a parking spot or not. ● Parking spot status is transmitted as a boolean (false for a vacant spot, true for an occupied spot).	D	Parking Sensor
DC-2	● Displays the customer's parking spot to the user if the license plate is scanned correctly or after receiving the user's reservation number through the terminal. ● Communicates with the customer reservation system to determine whether or not the customer at the elevator has a reservation or not. ● Allows the user to enter in their login information to confirm a reservation number through the registration and login interface.	D	Elevator Terminal
DC-3	● Keep track of when vehicles enter and exit the garage. ● Sends data to the central server.	D	Camera
DC-4	● Scans the license plate of the vehicle entering the elevator. ● Sends license plate information to the customer reservation system to see if there is a match and therefore a reservation.	D	License Plate Scanner
DC-5	● Stores user login information, vehicles registered, and garage information. ● Communicates between the customer reservation system, registration and login interface, parking spot management system and management portal.	D	Central Server (Database)
DC-6	● Communicates with the elevator terminal to send information regarding a customer's reservation. ● Checks the time of the reservation. ● Determines if rain checks are to be issued to the customer by communication with the management	K	Customer Reservation System

	portal and parking spot management system.		
DC-7	● Allows the customer or the employee to register for the first time to the SmartPark website or login if already registered. ● Communicates with the central server which stores all the login information. ● Allows the customer to register a new vehicle, check the garage status, display bill up to the current date and pay his or her bill.	K	Registration and Login Interface
DC-8	● Holds the status of all of the parking spots in the garage. ● Communicates with the central server and updates the customer reservation system when spots are filled. ● Spot availability is also sent to the management portal	D	Parking Management System
DC-9	● Allows the manager to set pricing, enable dynamic pricing, view garage overview, and view garage statistics and local events	D	Management Portal
DC-10	● Allows the user to enter in information without using a keyboard, such as reservation times and login information.	K	Smart Assistant Support

b) Association Definitions

Table 38: Association Definitions

Concept Pair	Association Description	Association Name
Parking Sensor ←→ Parking Management System	Parking sensor determines whether or not a parking spot is occupied or not. This information is then sent to the parking management system where it is stored until the next update.	Update Parking Status
Elevator Terminal ←→ Customer Reservation System	The customer reservation system checks to see if the vehicle in the elevator has a reservation. The information is then sent to the elevator terminal and proceeds depending on if the customer has a reservation or not.	Notify About Reservation
Elevator Terminal ←→ Registration and Login Interface	If the customer did not have a registration, then the customer will have to provide his or her account details. This is done through the registration and login interface which is used by the elevator terminal.	Elevator Login
Camera ←→ Central Server (Database)	The camera watches the vehicles entering and exiting the garage as well as a timestamp of when they arrived and left. This information is transmitted to the central server to be stored and used for the employees of the garage.	Store Vehicle Data
Central Server (Database) ←→ Management Portal	Data is extracted from the database and is displayed to the management portal as charts and graphs. Additionally, the database can be accessed by the managers to manually add, delete, or sort information.	View Statistics
License Plate Scanner ←→ Customer Reservation System	The vehicle's license plate is scanned and is then crossed-checked against the customer reservation system.	Check Reservation
Customer Login System ←→ (Database) Customer Registration Information	The login information provided by the user is checked against the database to see if the credentials provided are correct to give them access to their account.	Check Login Credentials
Customer Reservation System ←→ Managerial Database	To give the customer the option to view their current billing cycle from the first day of the month to the current day.	Check Bill
Management Price Tool ←→ Central Server (Database)	The Management Price Tool requests the database to update the prices for hourly rates and fees.	Set Pricing

Management Garage Loading Tool ←→ Central Server (Database)	The Management Garage View Tool requests the database to render an image of the garage layout to view its current occupancy in real-time.	Load Garage
Management Events Tool ←→ Central Server (Database)	The Management Events Tool retrieves information on local events from the Database.	View Local Events

c) Attribute Definition

Table 39: Attribute Definitions

Responsibility ID	Responsibility Description	Attribute	Concept
RC1	Know whether a car is parked in the spot or not.	spotOccupied	Parking Sensor
RC2	Display on screen which parking spot the customer should use.	displayParkingSpot	Elevator Terminal
RC3	Display accepts customer reservation numbers via GUI.	receiveReservationNumber	
RC4	Display walk-in or Reservation status.	hasReservation	
RC5	Takes photos of entering and exiting license plates.	carEnterExit	Camera
RC6	Determines whether or not an inbound car has a reservation.	reservationStatus	License Plate Scanner
RC7	Stores user information.	userInformation	Central Server (Database)
RC8	Stores reservation information.	reservation	
RC9	Checks time of customer reservation against current status.	reservationStatus	Customer Reservation System
RC10	Issues rain check if the garage is full.	rainCheck	
RC11	Accepts customer data for registration of an account.	customerRegistration	Registration and Login Interface
RC12	Maintains occupancy status of all spots in the garage.	garageStatus	Parking Management System
RC13	Allows administrators to set pricing policies.	setPricing	Management Portal
RC14	Allows administrators the ability to view local events.	viewEvents	
RC15	Allows administrators to view historic occupancy data.	viewData	
RC16	Allows customers to make reservations using voice interface.	makeReservation	Smart Assistant Support

d) *Traceability Matrix*

	UC ID	UC1	UC2	UC3	UC4	UC5	UC7	UC10	UC11	UC13	UC14	UC15	UC16	UC17	UC18	UC19
DC ID	Ct.	3	3	3	6	5	3	2	5	4	3	3	4	5	4	2
DC1	1														X	
DC2	2				X									X		
DC3	3								X			X			X	
DC4	3				X				X			X				
DC5	15	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
DC6	9	X	X	X	X	X	X			X			X	X		
DC7	9	X	X	X	X	X	X			X			X	X		
DC8	5					X			X		X			X	X	
DC9	5							X	X	X	X					X
DC10	3				X	X							X			

Image Label: Matrix of Intersections between Domain Concepts and Use Cases

This is the final use case implementation of our project, this matrix has been updated to show only the use cases that have been shown in the final demo and the corresponding domain concepts that they pertain to. As we can see above, some use cases overlap with many domain concepts like UC4, Make Reservation. Some domain concepts intersect with many use cases like DC5, DC6, and DC7; Central Server, Customer Registration System and Registration/Login Interface.

System Operation Contracts

Tables 40-51: System Operation Contracts

Operation	Register
Precondition	The user must have already registered to the website using a unique username, password, email address, and other information. isRegisteredUser(loginName) == false
Postcondition	The user's login information will be saved into the database. customerRegistration(loginName) == success

Operation	Login
Precondition	The user attempting to create an account cannot use an email or username that is currently registered in the system. isRegisteredUser(loginName) == true
Postcondition	The user can now make a reservation, edit an existing reservation or edit account information. hasUserAccess(loginName) == true

Operation	Edit Account
Precondition	The user attempting to change some attributes of their account isRegisteredUser(loginName) == true editAccount(newAttribute)
Postcondition	The user's information is updated and SmartPark also has a complete copy of it.

Operation	Make Reservation
Precondition	The user must have already registered to the website using a unique username, password, email address, and other information required to login. They must also have a payment method on their account. hasPaymentOnFile(loginName) == true isRegisteredUser(loginName) == true
Postcondition	The user will receive confirmation that they successfully made a reservation by receiving a message on the screen they made the reservation on. sendResConfirmation(loginName)

Operation	Edit Reservation
Precondition	The user is already logged in and has a reservation to edit. editReservation(reservationID)
Postcondition	The user will receive confirmation that they successfully updated their reservation.

Operation	Pay Bill
Precondition	The user must already have an account and be logged in with a payment method on their account. hasPaymentOnFile(loginName) == true
Postcondition	The user will receive confirmation that they have successfully paid their bill and receive a receipt for the transaction which will be emailed to the email associated with their account sendPaymentConfirmation(loginName)

Operation	Set Pricing
Precondition	The user must be logged in and have administrative privileges. hasAdminAccess(loginName) == true
Postcondition	The data will be updated in the pricing table in the database. updatePricing(pricingOptions)

Operation	View Garage Information
Precondition	The user must be logged in and have administrative privileges. hasAdminAccess(loginName) == true
Postcondition	None

Operation	View Garage Usage Statistics
Precondition	The Manager must be logged in and have admin access. hasAdminAccess(loginName) == true
Postcondition	None

Operation	Update Parking Spot Status
Precondition	None
Postcondition	The parking spot's status will be updated as either vacant or occupied. spotOccupied == true if car in spot spotOccupied == false if car not in spot

Operation	Park
Precondition	The consumer may or may not have a reservation. There needs to be an open spot on the ground level if the customer is a walk-in. There needs to be an open spot in general if it is a car with a reservation. hasReservation(licensePlate) == true Prompt appropriately in the elevator GUI. hasReservation(licensePlate) == false Prompt appropriately on ground floor GUI.
Postcondition	If the customer is a walk-in, they will get a ticket and a parking spot on the ground floor. If the consumer already has a reservation, then they are allowed to park if there is a spot. However if they have a reservation, and there is no space they are given a raincheck. If they have walked in and there is no space, they are not given a spot in the parking garage. garageFull == true Issue rain check isWalkin(licensePlate) Initiate walk-in policy. elevatorPrompt(licensePlate)

Mathematical Model

Poisson Random Process

The mathematical model used to simulate cars leaving and arriving at the garage will be based on two Poisson random processes: one to simulate arrivals and one to simulate departures. The probability of seeing n arrivals in a time interval Δt is:

$Pr(n) = \frac{e^{-\lambda\Delta t}(\lambda\cdot\Delta t)^n}{n!}$ and $E\{n\} = \lambda\cdot\Delta t$.

We can use the same model to simulate the departures.

To simulate the time between arrivals, we can generate an exponential random numbers $rx(u) = \frac{-ln(u)}{\lambda}$, where u is a unit of one hour and lambda represents the average number of arrivals per hour. Then, we can perform the arrival simulation as follows:

1. Change the status of one spot from “Available” to “Occupied.” If there are no available spots, mark as an “Overbooked” event.
2. Generate a random value rx as defined above. After a time t(rx) = 60rx minutes, perform step 1 again.

The same process can simultaneously be applied to departures.

Dynamic Pricing Model

Dynamic pricing is the practice of basing the price of a product on a non-static set of factors for the purpose of maximizing revenue. In the case of a parking garage an owner is dealing with a finite resource, parking spots, from which to extract value. In this section we explore a model to increase revenue based on traffic volume and parking statistics data from Uber.

The Uber model shows strong correlation, 93%, between the volume of traffic in a major city and the associated parking density for the given time period. Because our garage and its location are theoretical we are forced to make assumptions regarding the density in our own garage. We will assume the location of our garage is in a major metropolitan area and located in a high demand zone. Our model further assumes a 90% occupancy rate during the peak weekday times and follows the Uber model for decrease in occupancy as a function of the outflow of traffic from a metro area. Therefore, we peg a 90% garage occupancy rate with the peak traffic in Uber’s data set and allow the garage’s occupancy rate to fluctuate as the traffic data fluctuates.

A study of the price elasticity of demand for parking garages shows that “When parking demand in a specific area exceeds parking supply (resulting in a parking occupancy of close to 100%)” (Lehner, Peer 4). We use this research along with our previously stated assumption to justify no change in customer demand when using a dynamic pricing model whose maximum price threshold does not exceed a factor of 1.4 times the original static price.

To avoid deterring customers during low demand hours, our pricing model will only “go live” when the garage parking density is projected to be greater than 60% (minimum occupancy threshold of dynamic pricing) .

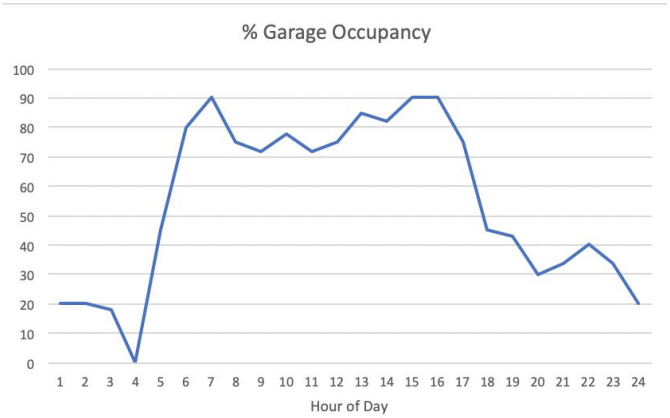


Image Label: Example of Sample Garage Occupancy in the Pricing Module

Administrator defined factors:

Base Rate: base

Minimum occupancy threshold for dynamic pricing: minThresh

Maximum occupancy threshold for dynamic pricing: maxThresh

Maximum Base Rate Multiplying Factor: baseMult

Dynamic factors:

Hours of parking desired: hrsParked

Current garage occupancy percentage: currOccPercent

$$base * hrsParked * [1 + \left[(currOccPercent - minThresh) * \frac{1 - baseMult}{maxThresh - minThresh}\right]]$$

The simple linear nature of the price model is its greatest strength. This model can be easily understood by garage owners and lends itself to simple market experiments. For example, one could take any arbitrary consecutive time period, split it in two and compare revenue performance by varying administrator defined factors to influence price/revenue/demand outcomes.

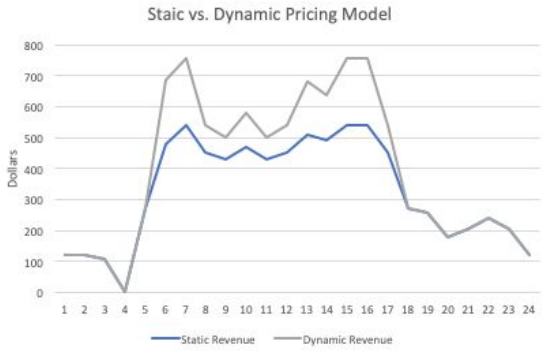
Image Label: Sample of Static and Dynamic Pricing

Static Model Revenue

Traditional flat rate: $6/hr.

Billable hours on a traditional weekday: 1313

Static Model Revenue: $7878



Dynamic Model Revenue

Minimum Charge Threshold: 60% Occupancy

Maximum Charge Threshold: 90% Occupancy

Maximum base rate multiplying factor: 1.4

Billable hours on a traditional weekday: 1313

Dynamic Model Revenue: $9564.88

Interaction Diagrams

Customer Registration Subgroup

The solid arrows show the case of a successful account creation and the dashed arrows show other possible scenarios

Use Case 1:

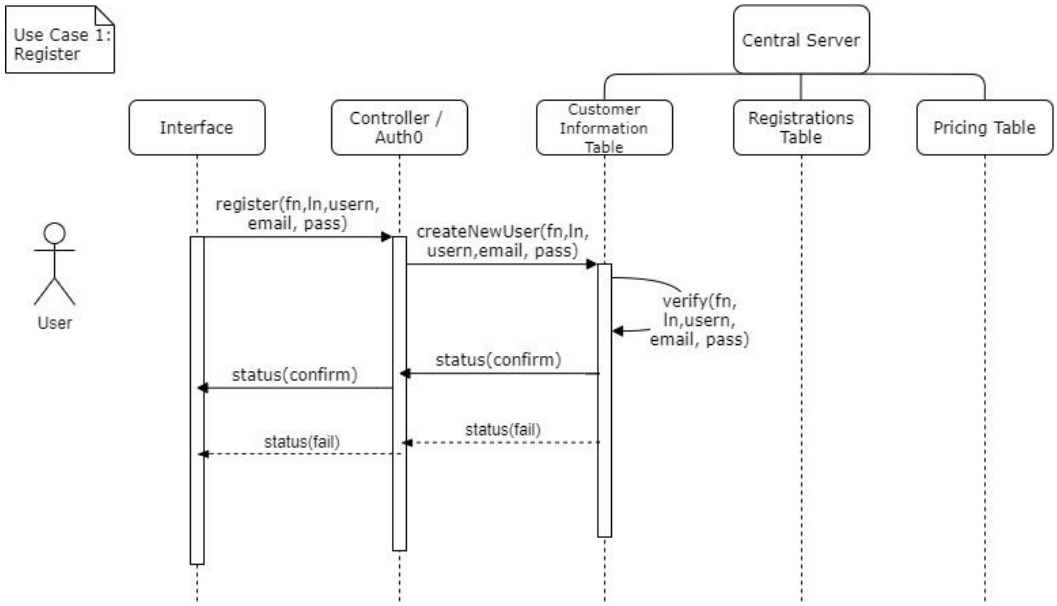


Image Label: Use Case-1 Interaction Diagram

This diagram shows the flow of control for Use Case 1, which is registering for an account. The customer provides their first and last name, preferred username, email address and password, since we used Auth0 for our login page, Auth0 now deals with these scenarios and just prompts the user to input a new attribute or lets them know that their account has been created. This use case applies the publisher-subscriber design principle because the message sent is of a fixed size and that many ‘subscribers’/accounts created are all of the same publisher type/SmartPark Account.

Use Case 2:

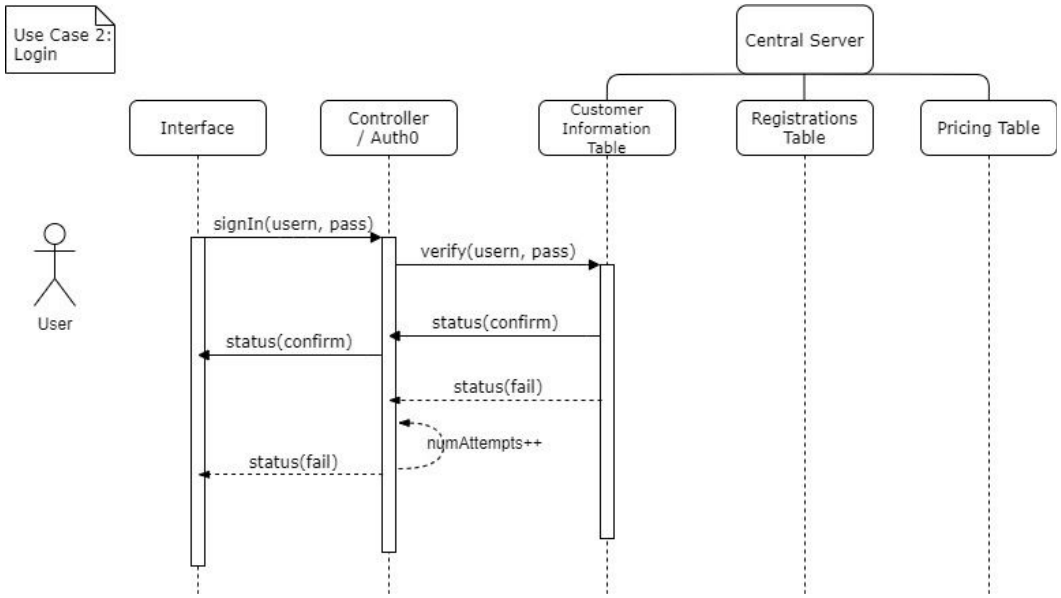


Image Label: Use Case-2 Interaction Diagram

This diagram is for Use Case 2, logging into the SmartPark System. It is also managed by Auth0 and checks the user's input for an account that matches and if no account is found then the system prompts them to try again. This is also an example of the 'publisher-subscriber' design principle, as it follows from Use Case 1.

Use Case 3:

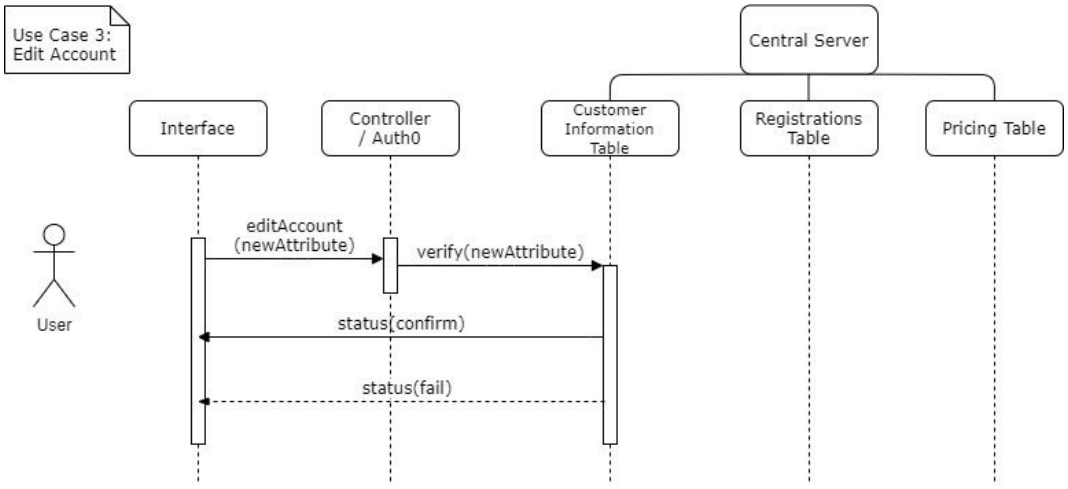


Image Label: Use Case-3 Interaction Diagram

This is the diagram for the Use Case where the user wishes to edit their account. All of the following Use Cases assume that the customer is already logged in. In case they wish to change their password or other information. This is an example of a 'decorator' design principle that the attribute that is being edited is the only one that is changed in the database table. The customer does not need to input all of their information to be re-entered just to change one detail.

Use Case 4:

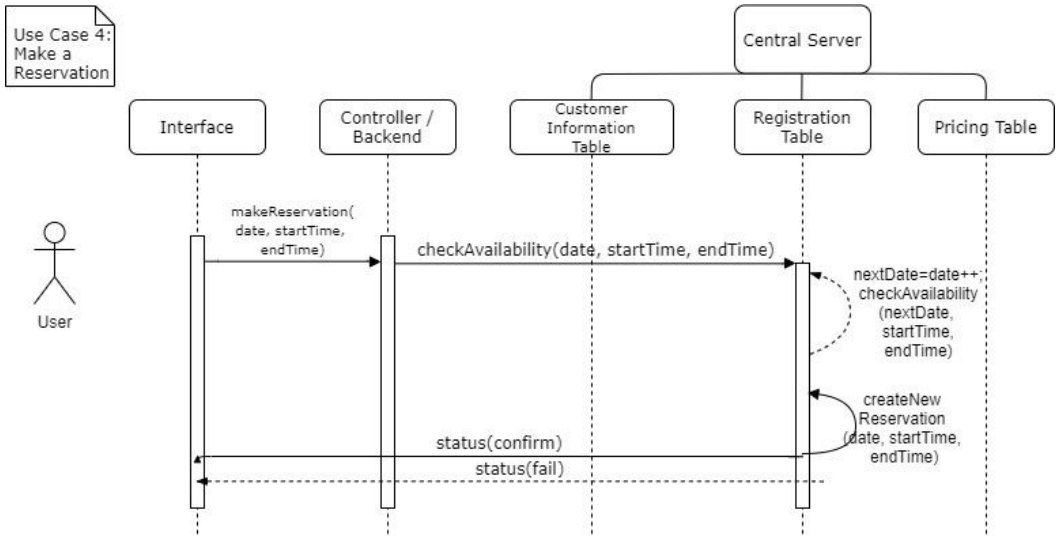


Image Label: Use Case-4 Interaction Diagram

The diagram above shows the Use Case to make a reservation. The user can select a date and input the times between which they want to park. The backend then sends the information through a post request to the database table that will check the availability and either confirms the reservation with a Reservation ID or shows the next available reservation if their desired time is unavailable. This use case applies the ‘publisher-subscriber’ design principle by having each reservation of the same type {date, lengthReserve, startTime, endTime}.

Use Case 5:

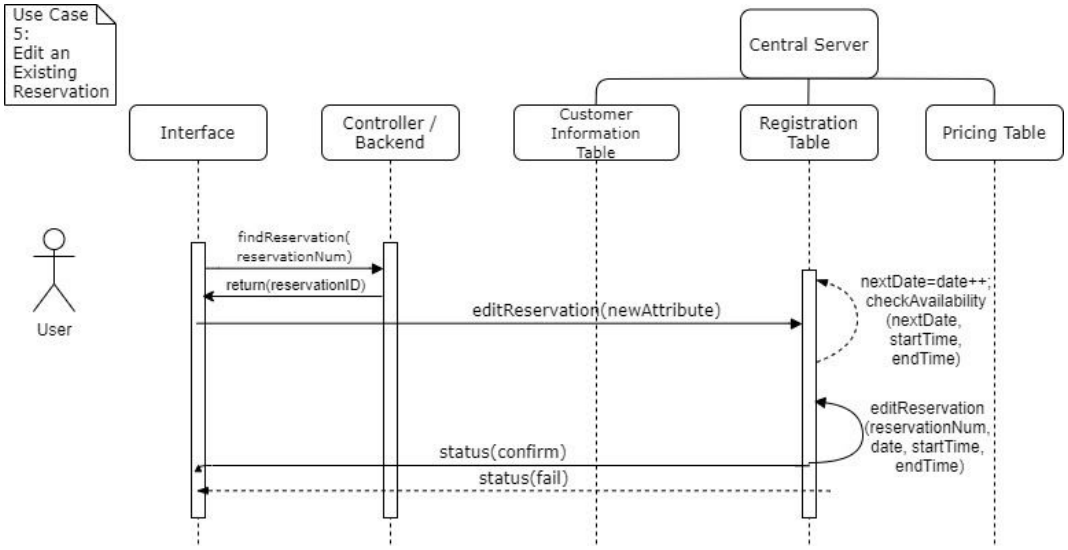


Image Label: Use Case-5 Interaction Diagram

This is the diagram for the Use Case where the user wishes to edit an existing reservation. The user can update a reservation to better fit their schedule, by changing the date, start time or end time. This is an example of a ‘decorator’ design principle that the attribute that

is being edited is the only one that is changed in the database table. The customer does not need to input all of their information to be re-entered just to change one detail.

Use Case 7:

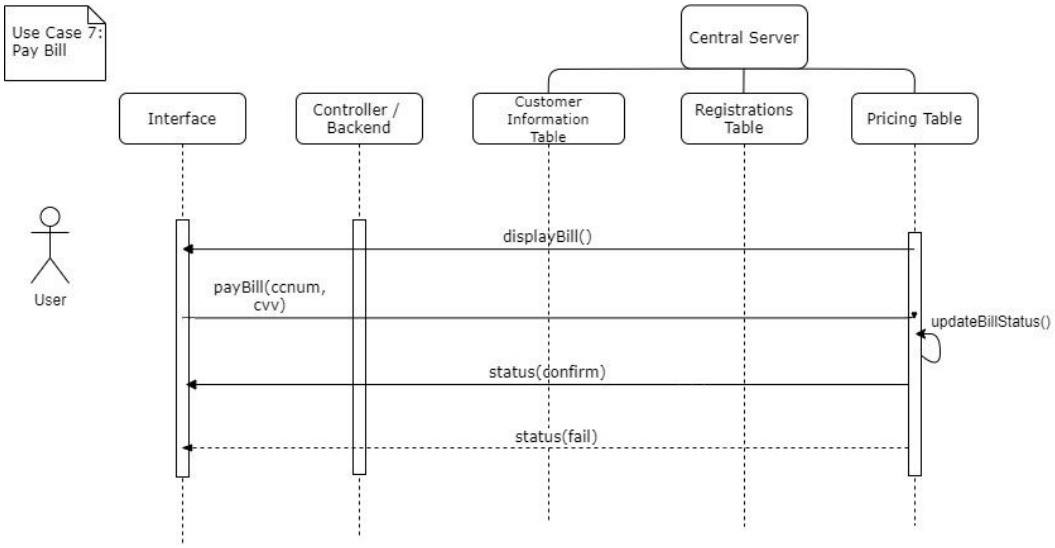


Image Label: Use Case-7 Interaction Diagram

The diagram above is for Use Case 7, where the user can pay off their bill. The user can enter in their credit card information and if processed successfully then the amountofDues value in the table of the database if updated to $0.00. This is an example of a ‘decorator’ design principle that the attribute that is being edited is the only one that is changed in the database table.

Managerial / Administrative Subgroup

Use Case 10: Set Prices

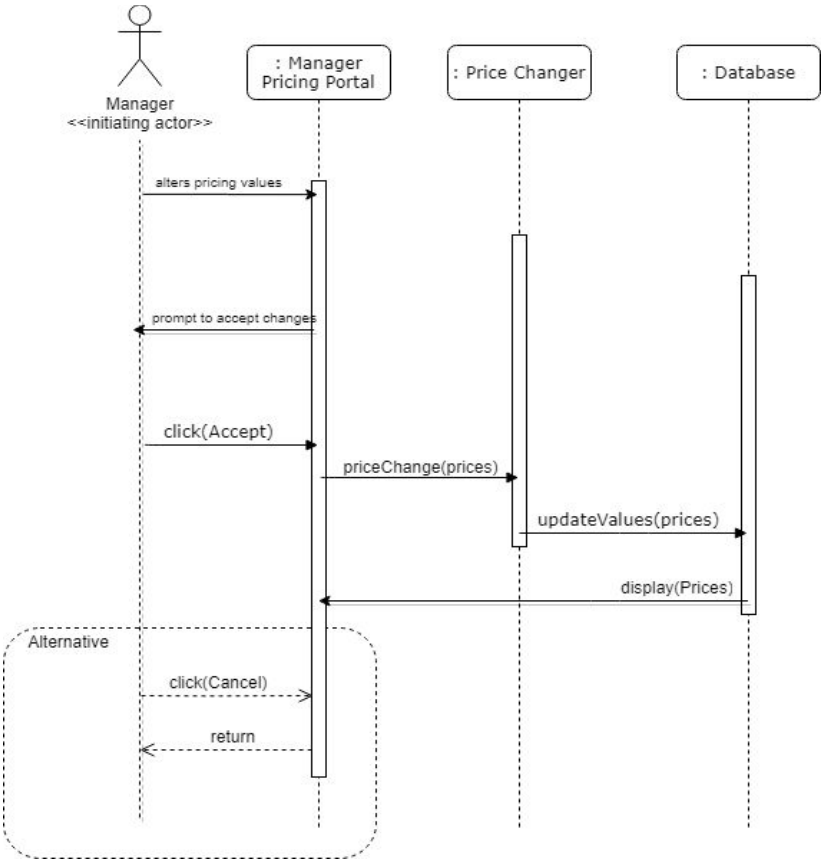


Image Label: Interaction Diagram Manager Use Case #10

Use Case 10 allows the manager to edit prices for hourly, overtime, and walk-in rates. In this case a Decorator model is used with new prices introduced through dependency injection of updateValues(prices). Only the attribute that is changed is edited in the database table and the client only sees the most updated prices, just like how a Decorator forwards a request to the next object in the chain. Any price that is edited (hourly, overtime, or walk-in) is updated in the same way.

Use Case 11: View Garage Information

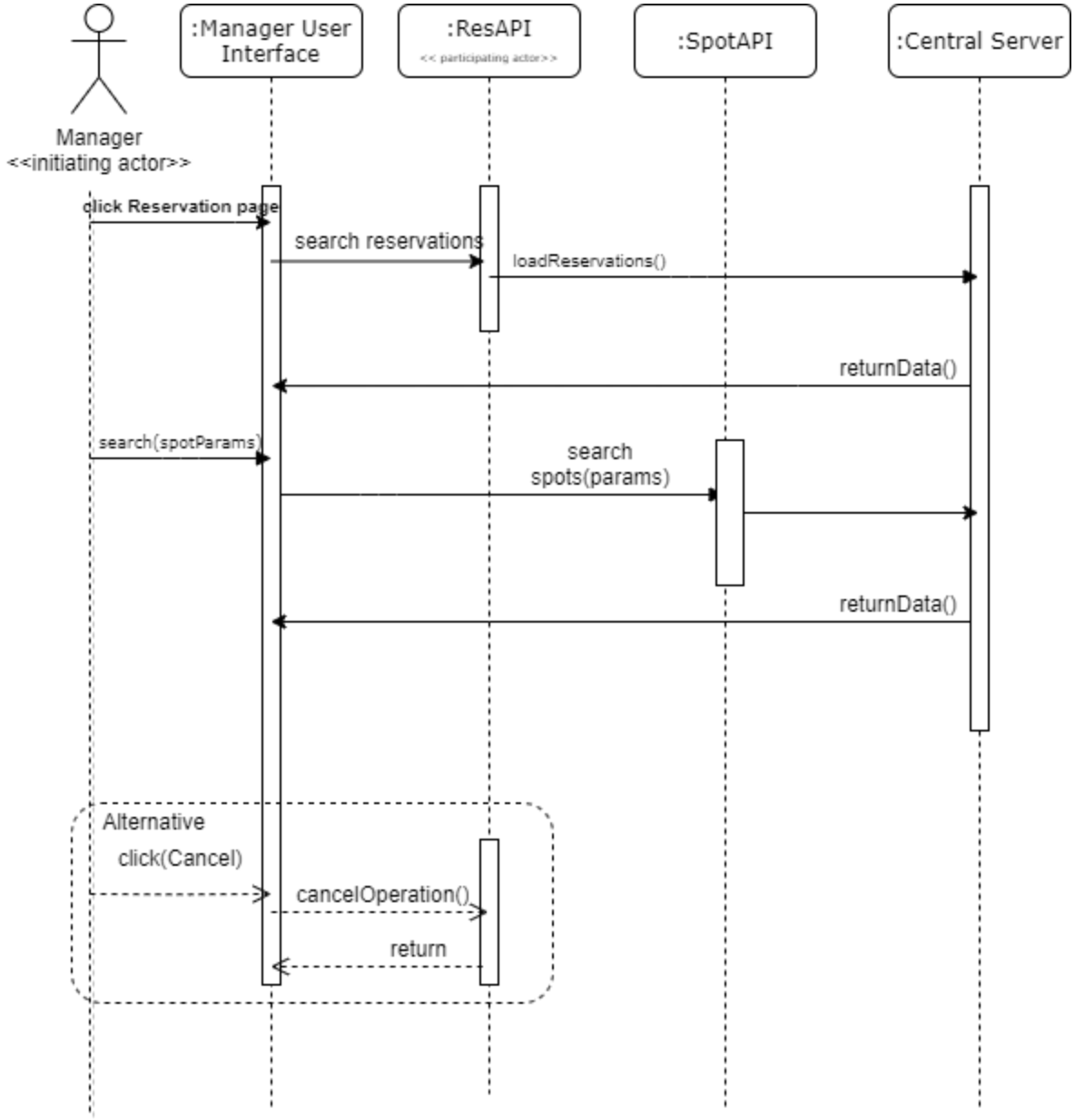


Image Label: Interaction Diagram Manager Use Case #11

Use Case 11 allows the user to view the current garage status information, including current reservations and spot occupancy status. This use case follows the Publisher/Subscriber design pattern. When a reservation is added, the message is updated in the database and subscribers like the Statistics and Reservation Pages can access this information. The data is published by Reservation API and multiple clients can read this information and update the user interface accordingly.

Use Case 14: View Garage Statistics

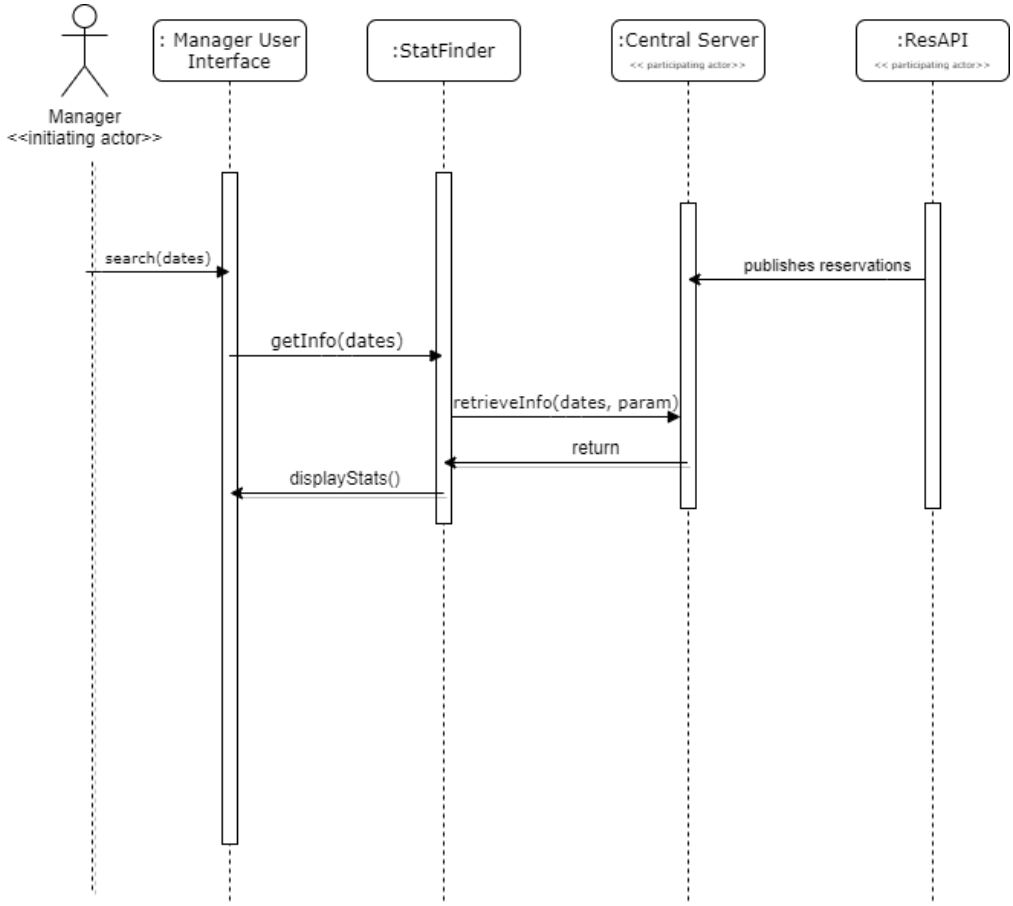


Image Label: Interaction Diagram Manager Use Case 14

This is another example of the Publisher/Subscriber Pattern. In this case, the data is published by the ResAPI and the Statistics Page acts as a subscriber of this information through retrieveInfo(dates, param). We use an aggregation of information from the Reservations table to display data for the table used to render the graphs in the Statistics Page.

Elevator Operation Subgroup

Use Case 18: Update Parking Spot Status

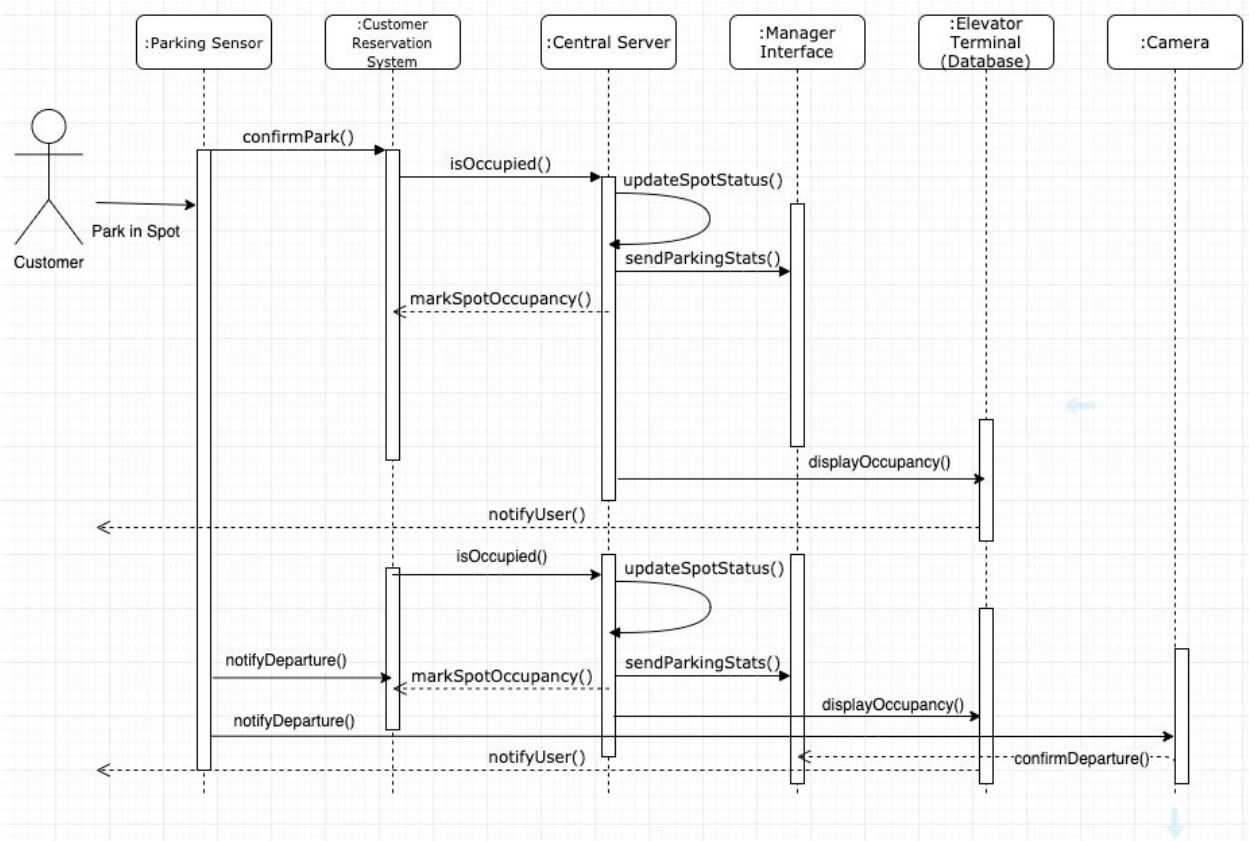


Image Label: Interaction Diagram Elevator Use Case 18

In use case 18, the customer's car interacts with the parking spot sensor. This sets off a sequence in which the parking spot sensor communicates status to the reservation system. The reservations system's job is to inform other systems about the newly occupied or unoccupied spot. Because of this it is a knower in our system. The information about the spot status is made available to the manager interface and elevator terminal via communication of this knowledge.

Use Case 19: Park, Entryway Terminal

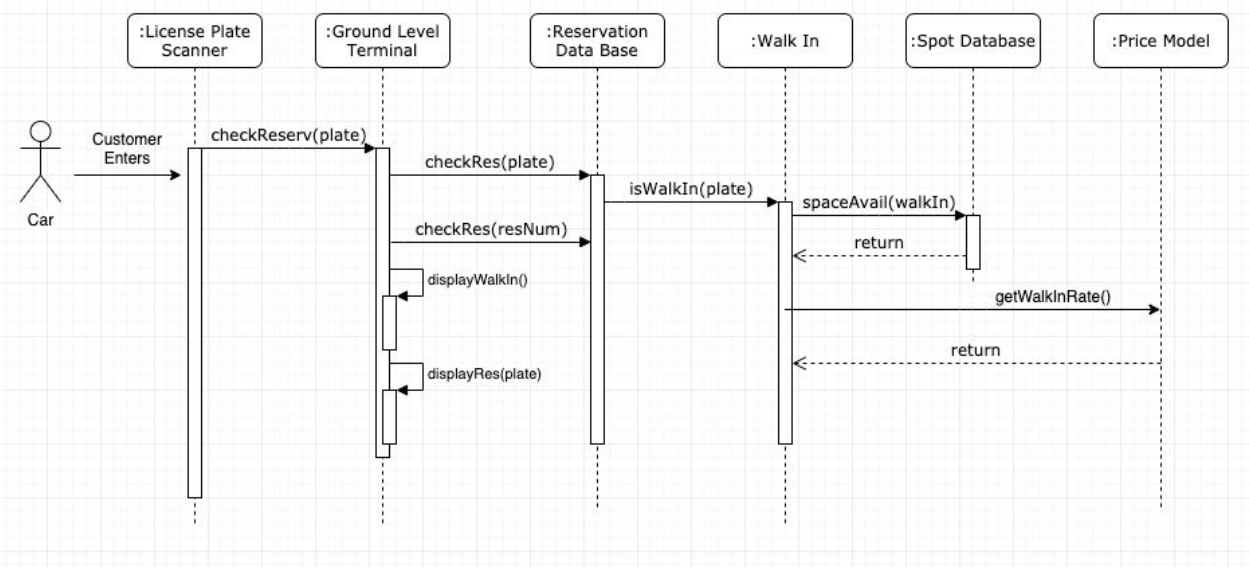


Image Label: Interaction Diagram Elevator Use Case 19

Use case 19 is initiated when the customer enters the garage. When entering the garage the license plate scanner takes a snapshot of the plate and passes this to the ground level terminal. The ground level terminal has the responsibility of *doing* in this sequence of interactions. The ground level terminal is designed using High Cohesion Principle in that it doesn't take on too many computations, it outsources requests for the information it needs and then displays this to the customer at the terminal.

In this sequence we omit the elevator portion of the use case, leaving it for below. Here we see the sequence for a walk-in customer. The modules after the Ground Terminal are all considered knowers. The spot database reports back to the walk-in module about space availability and the price module provides a current rate for the customer.

Use Case 19: Elevator Terminal

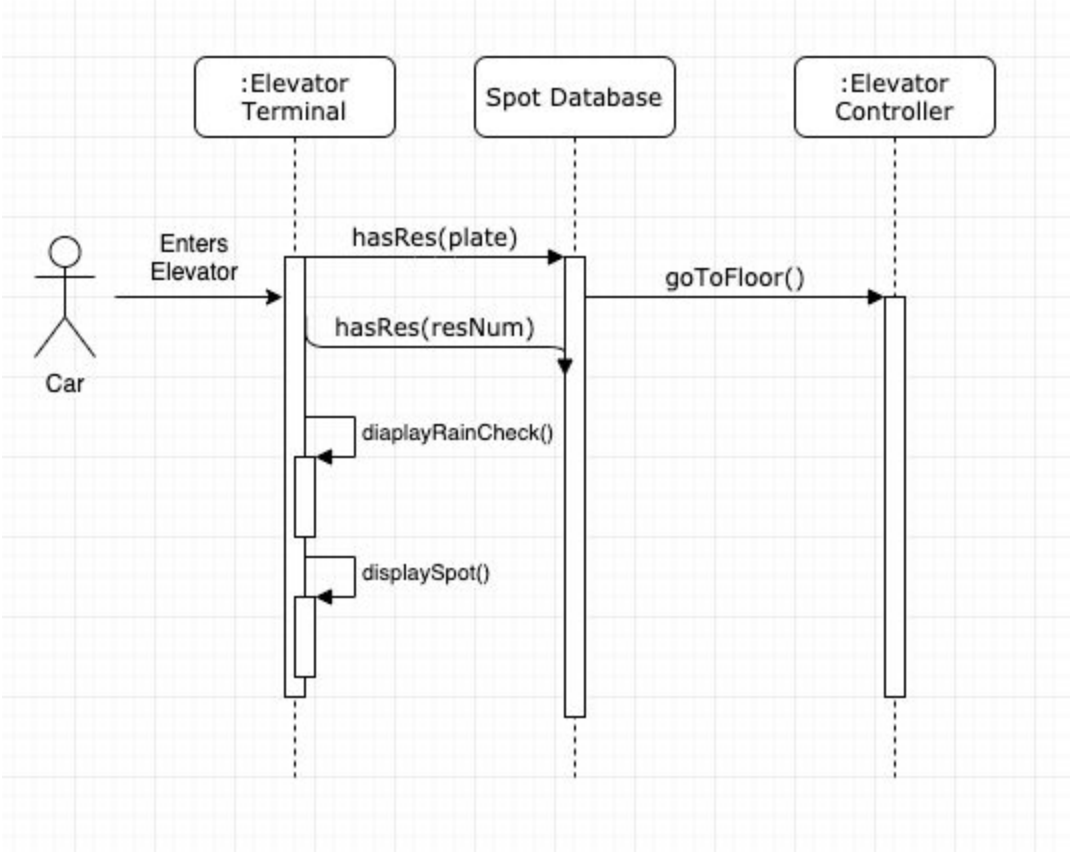


Image Label: Interaction Diagram Elevator Use Case 19

When the car advances to the elevator, the terminal within acts as an doer; requesting information from the spot database and the elevator controller. This is another use of the high cohesion principle in which the elevator terminal offloads a great deal of computation and simply references other modules for what it needs. We call on the spot database to act as the knower and provide relevant information back to the elevator terminal. Finally, the elevator controller acts as doer and takes the car to the floor corresponding to the customer's reservation.

Class Diagram and Interface Specification

Class Diagram

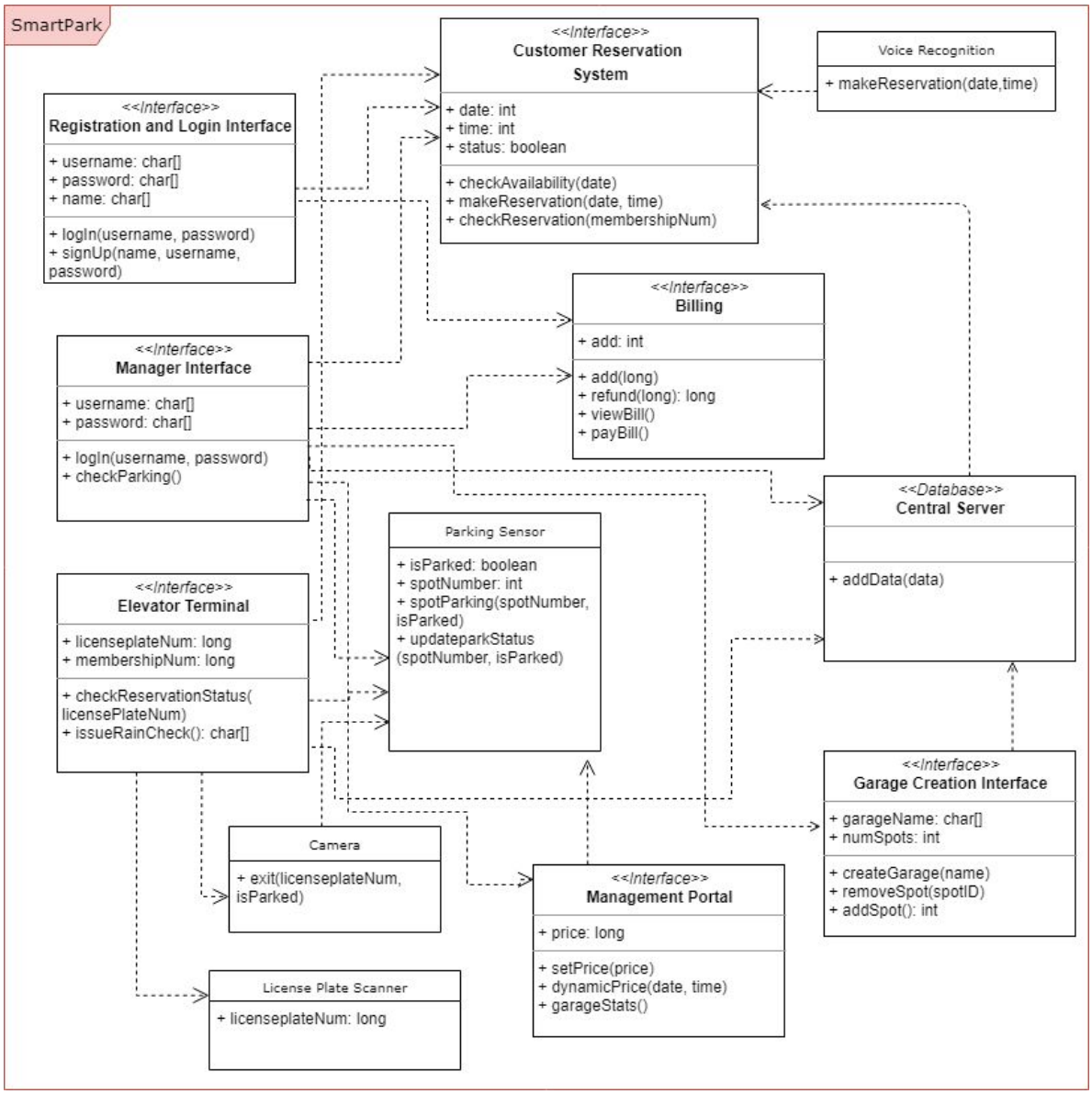
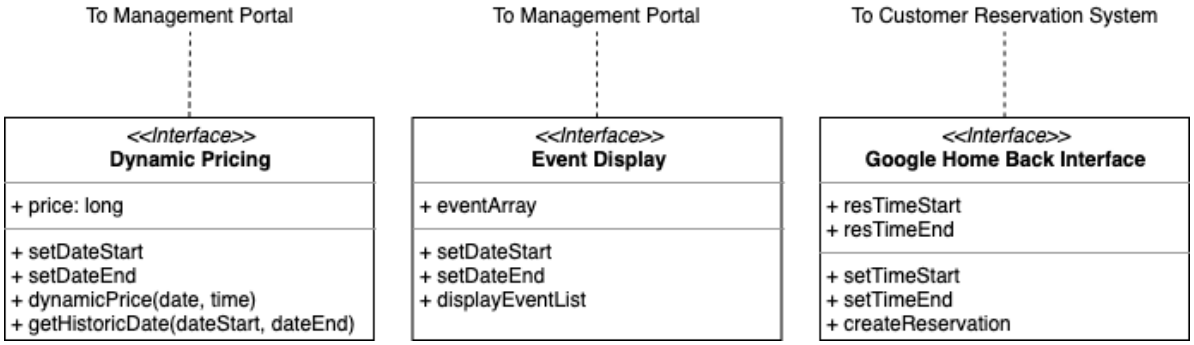


Image Label: Class Diagram for SmartPark

Added overview class diagram components here that do not fit onto the original diagram for space purposes.



Managerial/Administrative Subgroup Extension

Part 1:

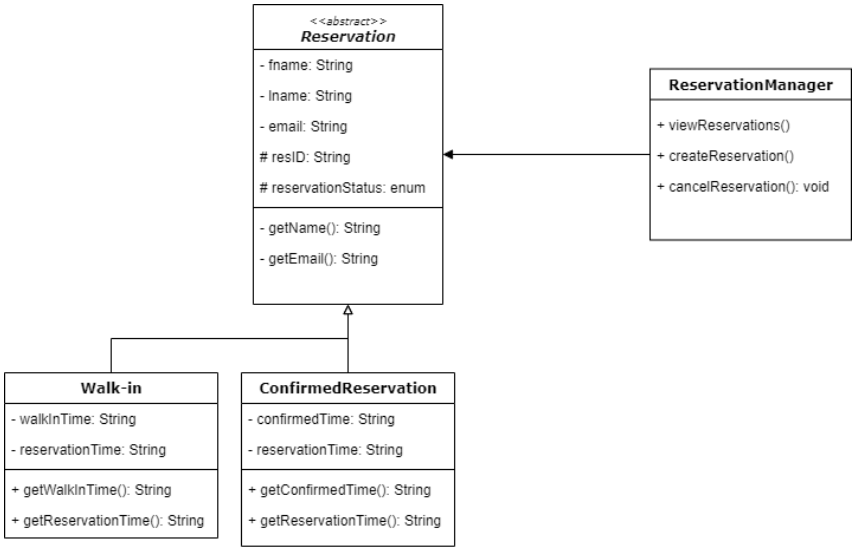


Image Label: Managerial / Administrative Subgroup Extended Class Diagram Part 1

Part 2:

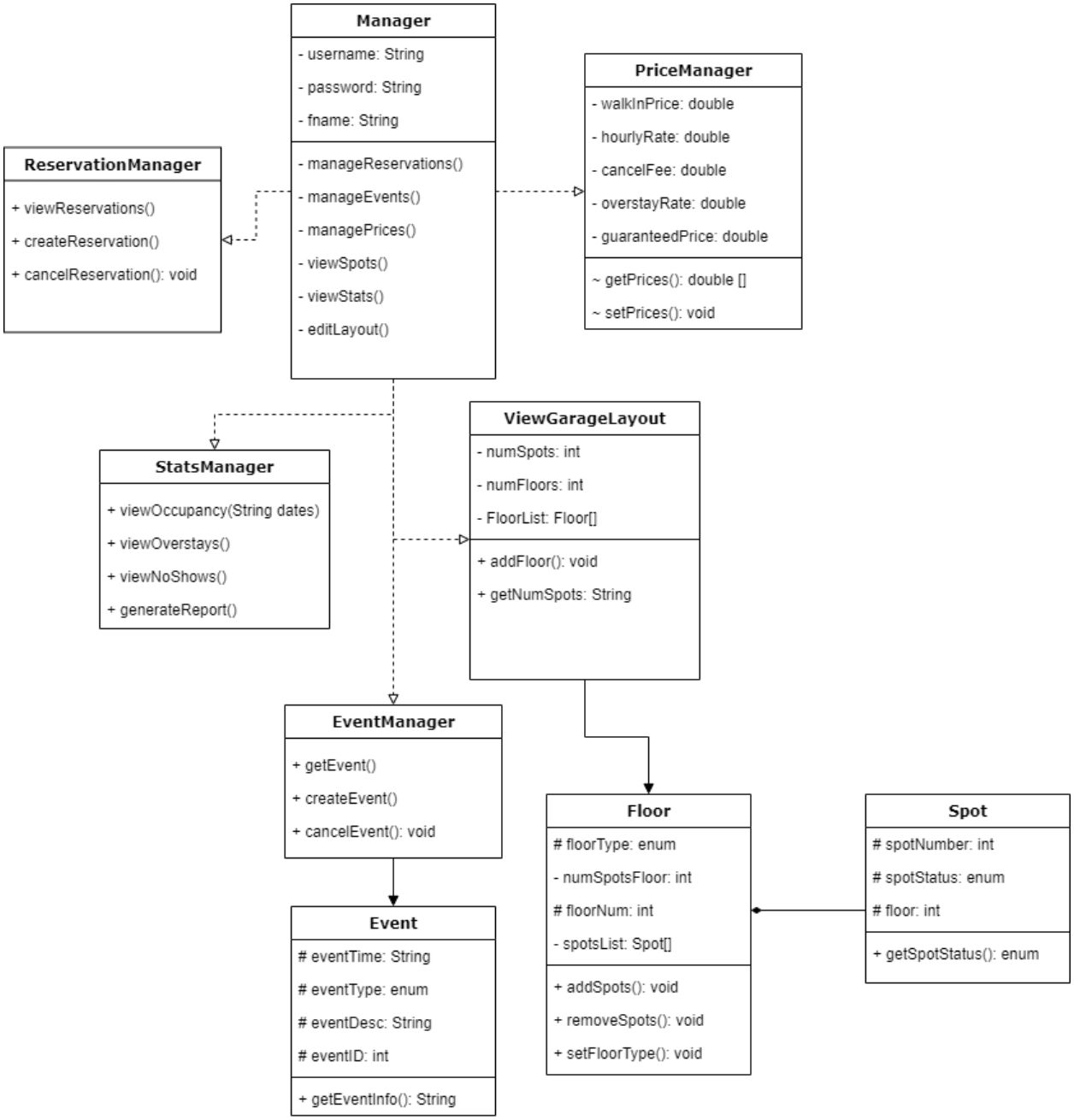


Image Label: Managerial / Administrative Subgroup Extended Class Diagram Part 1

Part 2 of the Manager portion of the class diagram shows exactly what the Management Portal has the ability to access. The Manager class handles and authorizes all administrative credentials through the Manager Interface. It then gives the Portal and therefore the user, access to the Customer Reservation System (ReservationManager) and the Central Server (PriceManager, StatsManager, and EditGarageLayout). The Dynamic Pricing System (PriceManager) affects Billing as well as Managers set the parking prices and fees for customer reservations. The Garage Overview (EditGarageLayout) is the system with the classes that run

throughout the Garage Creation Interface. The Manager class also implements the calendar system (EventManager) showing current and future “real-world” events.

Part 1 of the Manager portion of the class diagram shows that the ReservationManager has access to all parts of the Customer Reservation System, including Walk-ins and Guaranteed (Contracted) reservations. It also allows for Managers to create and issue rain checks to customers through the RainCheck class. The License Plate Scanner scans the license plates of all cars that enter the Elevator Terminal. The rain checks can be shown on the Elevator Terminal screen once the Camera and the Parking Sensor send the data to the Terminal that there are no available parking spots when the customer arrives.

Traceability Matrix

Traceability Matrix between Domain Concepts and Software Classes	Software Classes	Custom Reservation System	Billing	Central Server	Voice Assistant	Registration/Login Interface (Elevator)	Elevator Terminal	Customer Reservation Sys. (Elevator)	Parking Sensor	Camera	License Plate Scanner	Manager	Reservation	ConfirmedReservation	Walk-in	PriceManager	Spot	Floor	Event	EventManager	ViewGarageLayout	ReservationManager	StatsManager	Dynamic Pricing	Event Display	Google Home Back Interface
Domain Concepts																										
Parking Sensor				X					X																	
Elevator Terminal				X			X																			
Camera				X						X																
License Plate Scanner				X							X															
Central Server		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X			
Customer Reservation Sys.		X	X	X	X	X		X				X	X	X	X							X		X		X
Registration and Login Inter.			X	X		X									X											
Parking Management Sys.			X	X				X				X					X	X			X		X			
Management Portal			X	X								X	X	X		X	X	X	X	X	X	X	X	X	X	
Smart Assistant Support		X		X	X								X	X												

Image Label: Traceability Matrix for intersection between Domain Concepts and Software Classes

Customer Traceability Matrix Description

Registration/Login Interface: The main terminal that will be used to collect the information that the user inputs to send to the central server to cross check.

Custom Reservation System: The subsystem that deals with the reservations that need to be made by the customer, it cross checks the dates and times of the proposed reservation until confirmed by both the user and the central server.

Billing: This is where all the money balance of the customer takes place. This is where we will generate and log all the costs for every reservation made, grace period extension charge, overstay fee and raincheck credit.

Central Server: The database that will have tables for every subgroup that will be referenced for account information

Managerial / Administrative Traceability Matrix Description

The Manager Group always has to use the Management Portal to access any of the data stored in the Central Server. Thus, almost all of our classes are related to those two Domain Concepts. The Walk-in class is not connected to the Management Portal because walk-in reservations are mainly created using the Customer's User Interface. The domain concepts that we originally defined were split into the software classes based on implementation. As we implemented the concepts, we found that multiple classes would use and interact with the functionalities defined by the domain concepts.

Manager: The manager class will access the central server to access information to display for the administrator. The manager class also interacts with the customer reservation system to view customer profiles and reservations. The manager class can also create and delete reservations in the system. The manager class will also view information from the parking management system to view the status of each of the spots. It can view the daily schedule of events for the parking garage. It can also view the pricing page to control parking rates. The manager class will serve as the controller component of the management portal.

Reservation: The Reservation class will access the customer reservation system to find the customer information associated with each reservation. The Reservation class also needs to access the Parking Management System to see which spots are open and the overstay status of a reservation.

ConfirmedReservation: The managers interact with the Confirmed Reservation because the managers will be able to see how much of the garage is reserved in advance to see how well their business is doing. To see if a reservation's status the manager can use its access to the Central Server, Customer Reservation Systems, and the Garage Overview page.

Walk-in: The walk-in class interacts with the central server, registration and login interface, and customer reservation system. The class first checks the customer reservation system to see if a spot is available. If it is, then a customer can use the customer reservation system to reserve that spot as a walk in. The central server is what the classes use to communicate to the other classes.

PriceManager: The Price Manager class accesses the management portal to receive and control the amount that the manager has set as the price for a reservation.

Spot: The spot class interacts with the parking management system and the manager portal. The managers can edit a spot's characteristics like spot number and status and these changes are updated in the parking management system.

Floor: The floor class interacts with the parking management system and the manager portal. The managers can edit a floor's characteristics like floor number and status and these changes are updated in the parking management system.

Event: If there is a local event the managers can change or discount the price based on the event.

ViewGarageLayout: This class interacts with the management portal and the parking management system. Managers can view the garage layout in their management portals and these changes will be updated in the parking management system.

ReservationManager: It interacts with the Customer reservation system and the management Portal. The manager through the Management Portal can see what spots have been reserved by customers and that data is provided by the Customer Reservation System.

StatsManager: Interacts with the Management Portal and the Parking Management System. The manager can see statistics provided by the Parking Management System in the Management Portal.

Dynamic Price Module: The manager may access the current price model and edit a range of dates for it to be implemented. The manager may further view the historic performance of the price model and statistics about the occupancy in the garage.

Event Display: The manager uses this module to access a view of relevant events coming up over the year. These should be events that would impact the demand for parking. This data will be manually input by a user, and stored in the database table.

Google Home Back Interface: The user should be able to create a reservation by using google home assistant. The assistant will contact the back interface system, check reservation availability, and create the necessary database entry.

Elevator Group Traceability Matrix Description

Central Server: The database will store information for a variety of different entity sets and relationships. The specific entity sets for the elevator would include data regarding the various sensors- the parking sensor, the camera, the license plate scanner- as well the customer reservation information and information regarding the parking spots. The central server connection with the elevator will be used to search and update information on reservations and parking spots, and to make decisions based on what is returned.

Elevator Terminal: The interface for the elevator itself. The elevator terminal uses information from the customer reservation system and the central server in order to determine the spots that the customer has a reservation and the spot is available.

Registration/Login Interface for Elevator: The registration/login interface for the elevator is used in tandem with the central server and the registration and login interface used for the SmartPark system. This interface allows the customer to enter in his or her membership number, which is then cross checked with the central server. If the membership number is in the system, then the interaction is complete with the user in the elevator in the event that the license plate could not be scanned.

Customer Reservation System for Elevator: The customer reservation system is used to determine if the customer whose license plate was scanned or membership number was entered has a reservation. If the customer has a reservation, then the elevator brings the user to the floor where his or her parking space is. If, however, the customer does not have a reservation, then the customer is asked to leave the elevator terminal and wait at the walk-in terminal.

Parking Sensor: The parking sensor determines whether or not a vehicle is occupying a parking spot or not. This information is then sent to the central server which retrieves it, and sends the information to various locations such as the elevator terminal when prompted for that specific spot or to the management portal where the managers can view the garage occupancy.

Camera: The camera's use in the elevator terminal is to determine when vehicles enter or exit the parking garage or the parking elevator.

License Plate Scanner: The license plate scanner is used to scan the customer's vehicle's license plate to determine if the customer has a reservation or not. The license plate is then used in the central server and the customer reservation system to cross check the reservation, and then appropriate decisions are made by the elevator terminal.

Interface Specification

Customer Registration Subgroup

1. Design Patterns

   Publisher - Subscriber Model

   Use Cases 1, 2 and 4 pertain to this type of design principle. The format that the MongoDB Table is expecting serves as the Publisher and the 'post' and 'get' requests made by axios and the front end.

   Use Cases 1 and 2 are managed by Auth0 that takes care of checking all known accounts and the information associated with them in order to login. The Auth0 system asks for a unique email address and password to ensure no double account creation.

   Use Case 4 is to make a reservation. The format in the MongoDB table is the date of the reservation, start time and end time. This is saved along with the email attribute that is automatically pulled from Auth0 with the management token

   Decorator Model

   Use Cases 3, 5 and 7 pertain to the Decorator model, because they follow the same outline of the Publishers in the previous part but make updates to specific attributes that the customer wants / wishes to change

   Use Case 3 is for the information that the Customer provides to SmartPark such as vehicle and credit card information, once the user fills in the attribute that they want to edit and press 'save changes' this makes the post request to change that entry for the customer's account.

   Use Case 5 does the same post request except the system already knows the specific reservation to edit because they are displayed in chronological order and the customer is able to select which reservation they wish to edit.

   Use Case 7 updates the billings table by making a delete request on the press of the 'Pay Bill' button. This simulates the 'paying' of the outstanding bills by making the bills equivalent to 0. We have also now updated the interface to pay off specific bills individually.

2. Object Constraint Language (OCL) Contracts

   We use operations on types of Collections in Use Case 7, paying the bill by subtracting the total billing reservations from MongoDB.

   PayBills::deleteBills(date, amount, description): void
   deletemany: email
   PayBills.date = ;
   PayBills.amount = 0;
   PayBills.description = ;

Managerial / Administrative Subgroup

1. Design Patterns

Publisher - Subscriber Model

Use Cases 11 and 14 follow this design pattern. In View Garage Information, the Garage Overview front end page gets the information for the garage layout through an axios GET request to our MongoDB backend. It displays the reservation statuses of all of the spots in our parking garage. This makes our frontend the Subscriber to the Publisher, our database. In View Garage Statistics, the Statistics page acts as a Subscriber to our MongoDB Reservation data collection, Publisher. The Statistics page also uses axios to GET the reservation information from the database, which computes aggregations to provide or "publish" the information to the Statistics page.

Decorator Model

Use Case 10, Set Prices, follow this design principle, as the client only sees the most recently updated pricing for parking rates on the Pricing page. Any values the administrators may input into the Pricing Page to adjust the prices replace the previous one's spots in the database by updateValues(prices). This passing on of the request to update the values in the database defines this set up as a Decorator. The list of Decorators can also expand or shrink without client notice.

Dynamic Pricing Model Design Patterns

The dynamic pricing model exemplified the use of the Publisher/Subscriber design pattern. The front end components were responsible for receiving input from the admin, but the front end components were not simple static pieces. They themselves were state-maintaining elements. Each UI component was responsible for maintaining an internal state, that upon button press would become a Commander (Publisher) and communicate its needs to the Subscriber.

The back-end APIs functioned as recipients of the Publisher's requests and served appropriate data where requested. Through this separation of concerns and specific delegation of duties the subscribe/publisher design pattern emerged.
It should also be noted that the relationships among the UI front end and the API backend always followed an event based scheme. The API is constantly "listening" for an event.

2. Object Constraint Language (OCL) Contracts

context PriceManager:

inv Price.OverstayFee >= 0
inv Price.WalkIn >= 0
inv Price.Hourly >= 0
inv Price.noShow >= 0

PriceManager::SetPrices(OverstayFee, WalkIn, Hourly, noShow): void

post: Price.OverstayFee = OverstayFee
Price.WalkIn = WalkIn
Price.Hourly = Hourly
Price.noShow = noShow

ReservationManager::DeleteReservations(Reservation to Delete)

pre: Reservations DB Connection
post: Reservation to Delete is removed
EventsManager::AddEvents(Event to Add)
pre: Events DB Connection
post: Event is added to Event Database
EventsManager::RemoveEvents(Event to Delete)
pre: Events DB Connection
post: Event to Delete is removed from Event Database

Elevator Operation

1. Design Patterns

   The Use Cases for the Elevator Subgroups mostly pertain to the Publisher/Subscriber model. The customer registers their car, and makes a reservation through the front end UI built using React components and accessed through the Elevator Terminal. The front end components are parts of functions that make them react a certain way when certain actions are taken with them, which makes them get input from the customer to pass to the backend.

   The MongoDB database connection to the backend from the frontend of the elevator terminal, passes the information of these actions to build and fill the relevant information in the database. The database is constructed as a relational database with primary keys and foreign keys that connect them.

2. Object Constraint Language (OCL) Contracts

   We have no Object Constraint Language Contracts.

System Architecture and System Design

Customer Registration Subgroup

a) Architectural Styles

The customer subgroup will be utilizing the Component Based Development Style. This style allows individuals in the subgroup to develop independent components that can be linked together for a functional product. We believe that this approach will be best because the members on our team consist of those that are proficient in different building blocks that our software will be dependent on. Since our project will consist of a front end service that users interact with, those that excel in React and Javascript will be able to develop the website component of the architecture. Whereas, the members that are more comfortable in developing the backend will set up the routes and router models to ensure data is stored and retrieved correctly. Finally, the team will work together to ensure the frontend and backend work together seamlessly.

We have split Customer groups into two main components; the interface and the backend. Every time a component is created or edited, it is pushed to Github where the other members of the group can view and integrate it as necessary. This process ensures that each page works, receives and stores data from the correct locations which minimizes errors that could compromise the entire system using specific branches on Github. If mistakes are made, it is easier to undo a merge as well as find and fix problems, that is why using a component based development style is essential for this project.

b) Identifying Subsystems

The main subsystem for the Customer subgroup will be the central server. This is necessary because customer information needs to be held in a location that allows it to be tied to the registration information for a customer, their billing which also ties to their registration, and the ability to store relevant vehicle, credit card, and other information.

c) Mapping Subsystems to Hardware

The hardware for the Central Server subsystem will mainly be MongoDB and Auth0. MongoDB will allow us to store information and pull information whenever the user requests it through the User Interface and through GET/POST requests from API testing tools like Insomnia and Postman. Auth0 is used to help keep track of each customer’s account with their specific email, username and password.

d) Persistent Data Storage

The Customer Group will use Auth0 and MongoDB for data storage. Auth0 is to store customer data such as login information for accounts (name, email and password). We also use MondoDB to store other Customer information such as Credit card and vehicle information. This information needs to be dynamically updated by the user at any time but also be protected so that only the user will be able to see their own information. While the customer is making or editing information we will use GET and POST requests to access data from MongoDB to keep track of reservations.

e) Network Protocol

For communication between client and server and for the structure of the web applications, our system uses Node.JS with the Cors and Mongoose packages. Node allows users to load scripts inline with the web pages and redirect the user to different pages via routes

based on requests they send to the server. Cors provides the middleware and can secure data transfer between browser and data servers. The mongoose package is a data editing library between MongoDB and Node.JS, it translates the objects from code to their interpretations in the MongoDB.

f) Global Control Flow

i) Execution Orderliness

This system is procedure-driven and executes in a linear fashion. Customers will have to create an account, enter their name, email and password. Once logged in Customers can enter their Credit Card information and add their vehicle information(color, make, model, license plate) .Then, they can create or adjust a reservation and pay for said reservation. Every customer will have to follow this procedure in this order. All walk-in customers would also have to create an account first and follow the procedure above.

ii) Time Dependency

There is time dependency in this system. When a customer chooses to make or adjust a reservation they will be doing so in real time. The customer can scroll through the calendar and pick a date and time convenient for them. Past dates are not clickable on the calendar. The system will record the customers reservation and block off a spot for the reserved time. Additionally, after the customer checks in to their parking spot, the number of minutes the customer has left of their reservation will also be tracked by the system.

iii) Concurrency

The Customer group does not utilize concurrency.

g) Hardware Requirements

The Customer group only requires user interfaces and there are no hardware requirements.

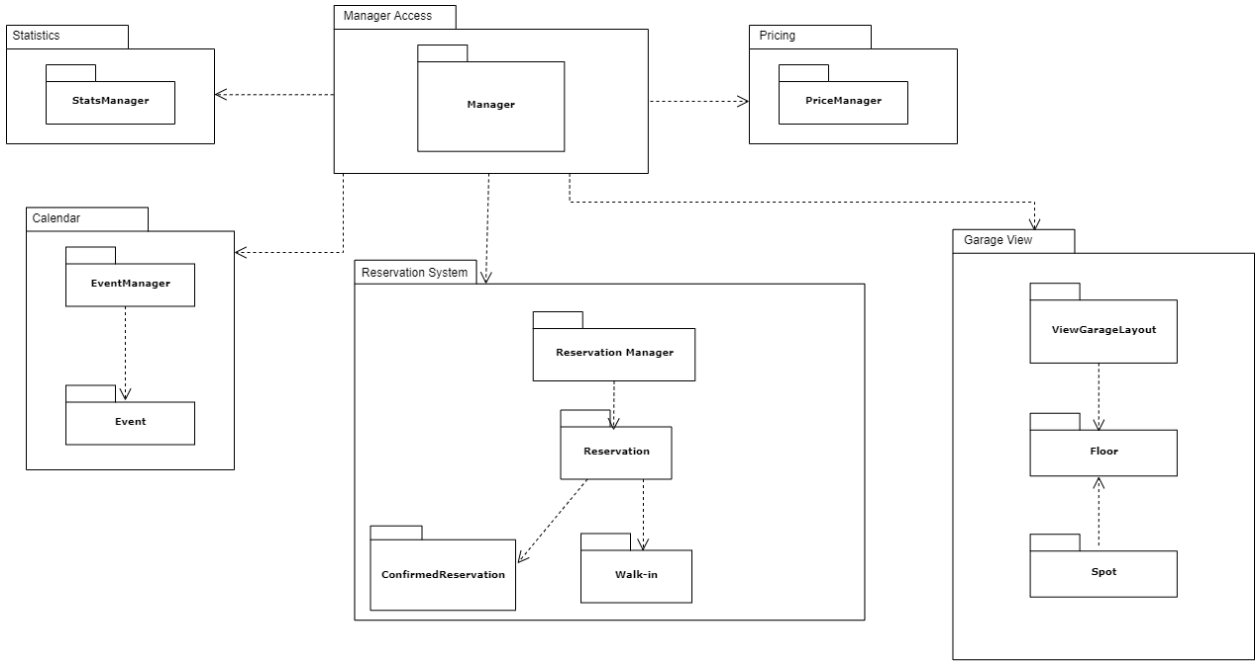
Managerial / Administrative Subgroup

a) Architectural Styles

The manager group will be creating a component-based web application as this is the architectural style best supported by the MERN stack. Each of the functions our application supports will be the responsibility of a dedicated component. For example, there is a component dedicated to displaying the current reservations in the garage.

The manager group will also be implementing a Model-View-Controller solution pattern. The user will mainly interact with the controller, which will interpret the user's actions such as button clicks. Then, the controller will edit or display the data as the user desires by editing the view or the data in the model. Crucially, the model represents only the data, and thus can be reused without modification. This will streamline our development process and allow us to easily support multiple functionalities for the same underlying data.

b) Identifying Subsystems



The Manager package diagram has five main subsystems with subpackages within each one. The Manager Access subsystem contains the *Manager* package which allows a parking garage manager access to data within the Central Server regarding all aspects of logistics. The viewStats() method within *Manager* relies on the *StatsManager* package located within the Statistics subsystem. This package manages and creates charts and graphs of all the data the parking garage has collected over a specified period of time. It is how Managers can track no-shows, overstays, reservations, and payment in general.

Manager also depends on the Calendar subsystem to keep track of upcoming and current events. The *EventManager* package accesses the *Event* package to create events in the calendar. The *EventManager* can also delete events and show them on the calendar.

Manager uses the Pricing subsystem to change the prices and fees for parking in the garage. The *PriceManager* package calculates and controls the prices of walk-ins and reservations. This can be done dynamically or manually by the manager.

The Garage View subsystem contains the controls and elements for the virtual garage layout. The *ViewGarageLayout* package allows the user to view the garage status. It directly manages the *Floor* package in order to do this. The *Floor* package shows the spots by floor. The *Spot* package contains parking spot information and statuses. Since the spots need to be on a floor and will not exist otherwise, the two packages are paired together within Garage View.

The Reservation System subsystem contains the tools for a manager to create,view, and manage reservations in the Central Server. The main package, *ReservationManager*, has the power to view, create, and manage reservations in the system, as well as issue rain checks for customers that had made a previous reservation, but could not park due to lack of spot availability in the garage. To create a rain check, this package accesses *RainCheck* which records all of the necessary information required to create a rain check such as a checkID, the time of issue, and the names of both the manager and customer involved. To create a reservation, *ReservationManager* calls the *Reservation* package to organize and send the relevant information to it. *Reservation* then imports the correct package based on the type of reservation that is being dealt with: *Walk-in or ConfirmedReservation*.

The *ConfirmedReservation* package would contain the times for the reservation and the confirmation for the reservation. The *Walk-in* package contains the reservation times for walk-in customer reservations.

c) Mapping Subsystems to Hardware

The system is hosted by a Node.js server. We have deployed our application using Heroku. Any changes we make can be deployed while development continues. The MongoDB Atlas Cloud database is hosted by Amazon AWS.

d) Persistent Data Storage

The MongoDB database will hold all data relevant to the Manager group. It is where all of the subsystems that the Manager package has to access are located. The Reservation System, Garage View, Statistics, Pricing, and Event subsystems are all implemented in the database. MongoDB uses collections to organize data; unlike SQL databases, MongoDB does not enforce a strict schema. Below is a list of collections the manager application will be able to access:

Reservations: Includes current and past reservations

Pricing: Current prices for hourly and overtime parking

Events: Information about upcoming events, includes fields like title, description, etc.

Spots: Information about the number of spots available, status of the spot, etc.

e) Network Protocol

We are using the HTTP network protocol. We need a protocol that will send requests individually between client, proxies, and server. This is because our SmartPark website will be responding to numerous individual HTTP requests when managers type the URL into a web browser to login to their account and look at specific web pages within the Manager Portal. It is also useful for fetching images, media, and HTML files to the server and updating the website quickly with new information. HTTP can also protect important web pages with an extra layer of authentication, giving the website the ability to handle sensitive user information while also

remembering the state of the server. We plan to use HTTPS for extra security on the manager subsystem after we deploy on the web server.

f) Global Control Flow

i) Execution Orderliness

The Manager application will operate in an event-driven manner. The user can choose to perform (or not perform) any of the actions that are available to them such as change prices, edit garage layout, or view existing reservations.

ii) Time Dependency

The manager system will not have any timers. However, in a real-world situation such as a sudden influx of customers, the view reservation page would have to be dynamically updated to ensure data validity.

iii) Concurrency

The manager system does not use multithreading. However, the simulation of the parking garage may use multiple threads.

g) Hardware Requirements

Screen Display - React.js for a useable Manager Interface

Central Server - At least 100 MB of space on MongoDB for reservation, price, spot, & floor information

Elevator Operation Subgroup

a) Architectural Styles

The elevator subgroup employs various architectural styles depending on the subsystem of the elevator. For the elevator terminal that the user interacts with, an *event-driven architecture* is utilized. Each user input into the terminal can be viewed as an event, as it leads to a change of state that the elevator terminal follows. For instance, if the license plate scanner was able to successfully scan a license plate, then this state is passed on to the customer reservation system to identify if the customer has a reservation. If the license plate could not be scanned, then control is passed on to a different state- prompting the user for a membership number to determine if a reservation can be found. The elevator terminal consists primarily of branching paths depending on what the user does or enters, and the state of this event is passed on to another state to control the system.

Apart from the event driven model, the elevator subgroup also employs the *client server model* because there needs to be communication between the database with all the customer information and the elevator interface. We are allowing the customers that reserve a spot in advance to be able to park on other levels and to do that we need to ensure that they customer is not a walk in, that is that their information exists in the database, check for an empty spot in the region that they want to park (also from the database) and display the closest empty spot in on the elevator interface. In this case, the client would be able to interact through the elevator system and the server would be the database that would respond to the client's requirements.

The architecture will also employ the *data centered model*, since the client will have access to data from a central server. The exchange of data between the client and the server is mutual; the server can provide data to the client, and the client can modify data in the server. One such example of a client modifying data is in the event that they decide to register as a user or change information about their account. All clients, whether they are registered or not, have the same amount of access to the data. This will make the server integrable and the data will always remain centralized. The data inside of the server will never branch off, even though there are many different scenarios for what could happen. Each scenario, while different, will always call back to the data that is centralized.

b) Identifying Subsystems

The GUI subsystem interfaces with the customer and contains the logical grouping of the two distinct GUI elements; Elevator Terminal and Entry Way Terminal. Its functionality further depends on the Administration package and Parking package to be able to convey useful information to the customer.

The Administration subsystem encapsulates the Registration and Login class as well as the Customer Reservation System. This logical packaging of classes reflects the logical commonalities between the two classes. Both deal with the business logic of the customer interaction and dictate the states displayed by the GUI subsystem based on their interactions with database elements and the real-time dependency on the Parking package.

Finally, the parking package encapsulates the Parking Sensor, Camera and License Plate Scanner class. These three sensors comprise the "eyes" of the system and report to the database, and thereby the administrative classes the real time activity in the parking garage.

c) Mapping Subsystems to Hardware

The elevator terminal can be extended to include all of the user input devices contained in the garage facility. This would not only encompass the elevator terminal's interface but also the input devices for walk-in customers as well. Each of these separate walk-in terminals will require its own hardware, meaning that for every walk-in terminal introduced a new computer will be necessary to run the interface for user interaction. These devices would each run the walk-in customer subsystem, whereas the main elevator terminal will control the flow of events for customers in the elevator. Each of these terminals, both walk-in and the main elevator, will be able to assign rain-checks to the customer in the event that the garage is filled.

In a real world deployment of this system there would need to be a minimum of two cameras; one placed at the entrance and one at the exit. For reliability the interface would be wired. Camera data would feed directly to the associated class modules for processing in real time.

Finally there would need to be a physical parking spot sensor to determine the occupied/unoccupied state of each parking spot. This would also be hardwired into the administrative system.

d) Persistent Data Storage

The elevator terminal needs to access a database to check for already reserved spots, valid customer reservations and spot vacancies that can be filled. This information will need to be kept for a long period of time. It can not simply delete this information once the user is done using the service. There will be a relational database between all of the groups, that can be accessed by the manager, the customer, and the elevator. All of the essential information for the project will be available in this relational database; it is the foundation of this project.

The elevator terminal has access to the database which stores information such as garage occupancy status, reservations, and access to customer login information. Each customer interaction has been able to obtain information from the database to determine the flow of events the customer will experience in the elevator terminal. The customer interaction may also go in the other direction, where they modify data such as registering as a new customer or changing information about their account. We have used a remote hosted instance of MongoDB to handle database needs for the system. We have integrated MongoDB with React.js in order to appropriately link the front end and back end applications. The React.js code will be set up to update itself in accordance to any changes that happen in the MongoDB database. React.js can update in real time and will not require the user to have to reload the interface. The updates to the front end will be fast and in real time.

e) Network Protocol

To accommodate the flow of data between the elevator system (on site) and the hosted backend our system would rely on a variety of HTTP requests. The systems from the parking package will update the database to reflect real time in/out flow of cars. This is to create a network communication chain of events in which the entryway terminal prompts the customer in a manner appropriate to their reservation status.

Since we would be using a combination of a data centered model and the client server model as architectural styles in this subgroup, we should be employing an extension of the TCP/IP model to implement the socket communication structure between the information that

the client enters into the system, to the elevator system communicating with the database that has past information.

When the client enters the elevator with the relevant number plate and proper reservation, the elevator system has to make a connection with the server that is connected to the database with the repository of relevant information for reservation. To implement this, we need to form a socket connection between the elevator terminal and the server system which is connected to the database for proper flow of information. Since a simple client server model is something that our entire group is familiar with, in terms of understanding and implementing, we are considering using this for the socket connections that need to be made as a part of the project.

f) Global Control Flow

i) Execution Orderliness

The execution on the elevator side of things mostly lies on the frontend and the backend for the most part. The user accesses the different terminals, in terms of the walk-in terminal and the elevator terminal to make a reservation or check for a reservation in the MongoDB database. The elevator terminal frontend has a user-friendly GUI that helps the user find their reservation while the backend was developed to store the reservations that were made.

ii) Time Dependency

There were no timers involved in the development of the elevator side of the things. The elevator terminal is supposed to timeout, or log out after a few minutes of inactivity to make account access more secure on the front end.

iii) Concurrency

Multi-threading was not used in the making of this project.

g) Hardware Requirements

1. **Server** - To process the information in the database and convey it to the server terminal
2. **Database** - At least 1TB to store all the relevant data for the customer reservations
3. **Cameras** - 1080p Wireless Security Cameras to ensure security of the garage
4. **License Plate Reader** - 25 GB minimum to store the information in the reader as a backup
5. **Touchscreen-capable display terminal** - Visual Interface System as User Interface
6. **Parking Sensors** - Wireless Garmin 40 seems to be a good parking sensor which would serve our purposes
7. **Display Screen** - to show the vacant spots of the garage outside of the garage

Algorithms and Data Structures

Algorithms

Customer Registration Subgroup

The Customer Subgroup did not use or develop any algorithms

Managerial / Administrative Subgroup

The Managers need to be able to create parking reservations and delete or alter existing ones. In order to do that, we needed to store the reservation information somewhere. We wanted a backend that has a database that is easily searchable, even when looking for specific information, and is flexible such that relevant information can be found through the use of any number of unique identifiers. Also, it is flexible in the sense that managers can easily create, alter, and/or delete reservations stored inside the database. The backend also needed to seamlessly integrate with the frontend of the website, to ensure minimal user effort. Thus, we decided to use MongoDB as our backend since we are already using other members of the MERN stack (Node and React) for our frontend. MongoDB Compass to create a SmartPark database. Within SmartPark, we created a data collection called “Reservations”. The collection stores reservation information as .JSON files in the Documents tab. We have created a Schema in our file res.js, so that any reservation information that is inputted through the Reservation page on the SmartPark website gets stored directly into the MongoDB Reservations database as a Document. We set up the Schema so that the reservation data is matched to certain object fields. Each field in the document is formatted as “key: value” with the “value” being of data type String, Int32, Boolean, etc. Each document is also automatically assigned an ObjectID by MongoDB. For example:

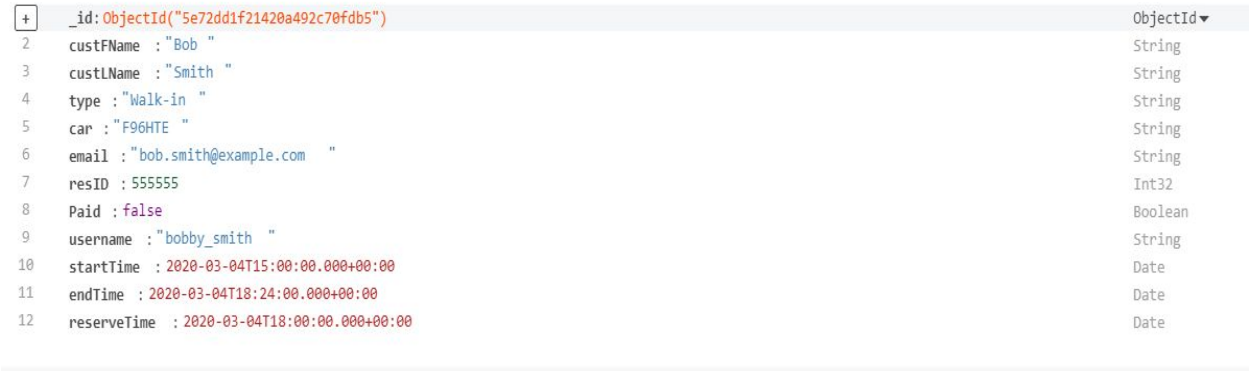


Image Label: Sample Reservations MongoDB Document

This type of formatting allows us to use the Schema tab in MongoDB to analyze the current data collection. Analyzing the schema allows our group to observe the frequency, types and ranges of the fields in Reservations, which is useful data for our parking garage Statistics page and its corresponding graphs and tables. The “key: value” format also provides managers with easy access when looking for specific information in each document. For example, normally a manager would look for a customer’s resID when searching for their reservation in the system. However, if a resID has not been issued (because a reservation was never made), a customer

can usually also be identified by their vehicle's license plate number or by the email they provide. Given a unique identifier, a manager is given the flexibility to find other data objects such as the customer's SmartPark account username or their reservation time. This system also allows for easy sorting of the reservation data, i.e. sorting by date of reservation. The data collections Spots, Floors, and billings also use .JSON files and Schemas to create MongoDB formatting.

Dynamic Pricing Algorithm

Dynamic pricing will require the use of a custom algorithm. This algorithm performs its computation phase with a formula derived from research referenced earlier in the report under the heading "Mathematical Models." The expression is represented again for reference:

$$base * hrsParked * [1 + \left[(currOccPercent - minThresh) * \frac{1 - baseMult}{maxThresh - minThresh}\right]]$$

In practice, we will need to develop a logical flow by which the administrator of the system can enter relevant data and receive a revenue projection from the model. The administrator can then choose to implement the price model. When a model is live it takes in actual parking occupancy data and uses the administrator-controlled values to compute a price for the customer.

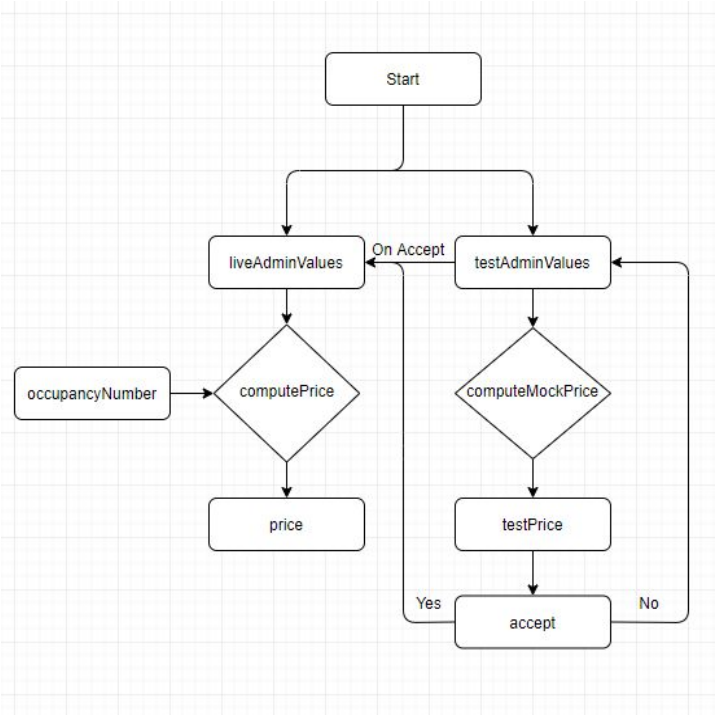


Image Label: Dynamic Pricing Algorithm Pricing Flow of Control

One can see that the algorithm will have two state options from the beginning. The admin can opt to see the price output data from the currently set model, or can enter new administrator set

values and compute recompute the model. If the admin decides to implement the new price then the algorithm recomputes a live price with the test values.

Elevator Operation Subgroup

The Elevator group used a very simple algorithm to search if a license plate was in the system. The license plate would be scanned and set to a state, referencePlate. Then, an HTTP request would be sent to find if that value existed in the database. If it existed, another state, licensePlate would be set to the value that was found. If it was not found, licensePlate would be set to NULL. referencePlate and licensePlate were then compared and if they matched, the user was recognized as a registered user.

Data Structures

Customer Registration Subgroup

The Customer Subgroup utilized a data structure in the form of JSON (Javascript Object Notation). This data structure utilizes attribute value pairs so that we can store information inside of our MongoDB document within the database. This makes it easier for people that oversee the maintenance of the database to understand what information relates to a certain attribute.

Managerial / Administrative Subgroup

The Managers need to be able to create parking reservations and delete or alter existing ones. In order to do that, we needed to store the reservation information somewhere. We wanted a backend that has a database that is easily searchable, even when looking for specific information, and is flexible such that relevant information can be found through the use of any number of unique identifiers. Also, it is flexible in the sense that managers can easily create, alter, and/or delete reservations stored inside the database. The back-end also needed to seamlessly integrate with the front-end of the website, to ensure minimal user effort. Thus, we decided to use MongoDB and Express as our back-end since we are already using JavaScript-based Node.js and React for our frontend. MongoDB Compass to create a SmartPark database. Within SmartPark, we created a data collection called “Reservations”. The collection stores reservation information as .JSON files in the Documents tab. We have created a Schema in our file res.js, so that any reservation information that is inputted through the Reservation page on the SmartPark website gets stored directly into the MongoDB Reservations database as a Document. We set up the Schema so that the reservation data is matched to certain object fields. Each field in the document is formatted as “key: value” with the “value” being of data type String, Int32, Boolean, etc. Each document is also automatically assigned an ObjectID by MongoDB. For example:

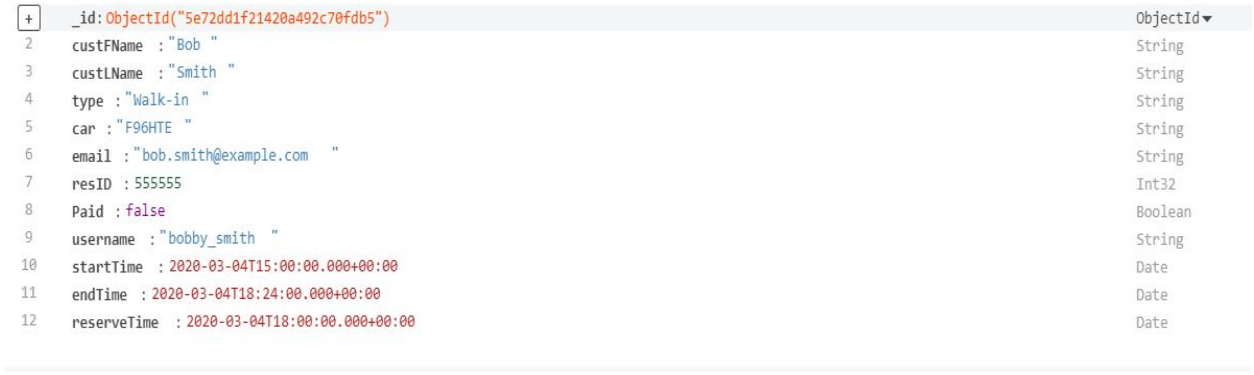


Image Label: Manager Database Example - Reservation

Elevator Operation Subgroup

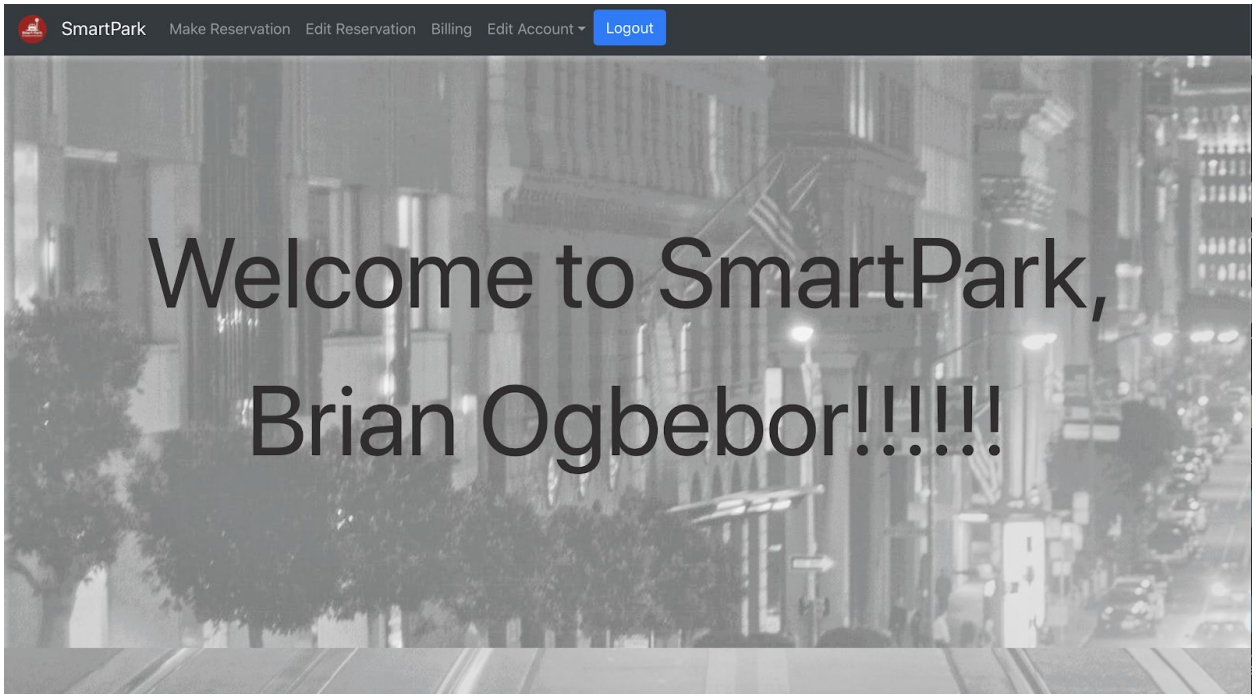
The Elevator Subgroup is using JSON packages (Javascript Object Notation) for the items in the back end. The items are the customers and the spots. The attributes within the items are organized by the type of attribute and the actual value of the attribute (i.e. spot number, floor number, customer name, license plate number etc.) The items will all be stored in MongoDB in the form of a JSON.

The elevator group searches the customer part of the database for a license plate value that is input into the program. It will see if any of the customers have a parameter that matches what is input. If this parameter exists, the state for the license plate and the reservation number will be changed to the values of those associated that are found in the database. Otherwise, they will be set to null. Likewise the same works for the reservation ID. After the license plate is found, the state for reservation ID is set to a certain value. This reservation id will then be input into the system. If a spot that is associated with this reservation ID is found, the spot number state will be set to that value. The user will be told to park at that spot.

User Interface Design and Implementation

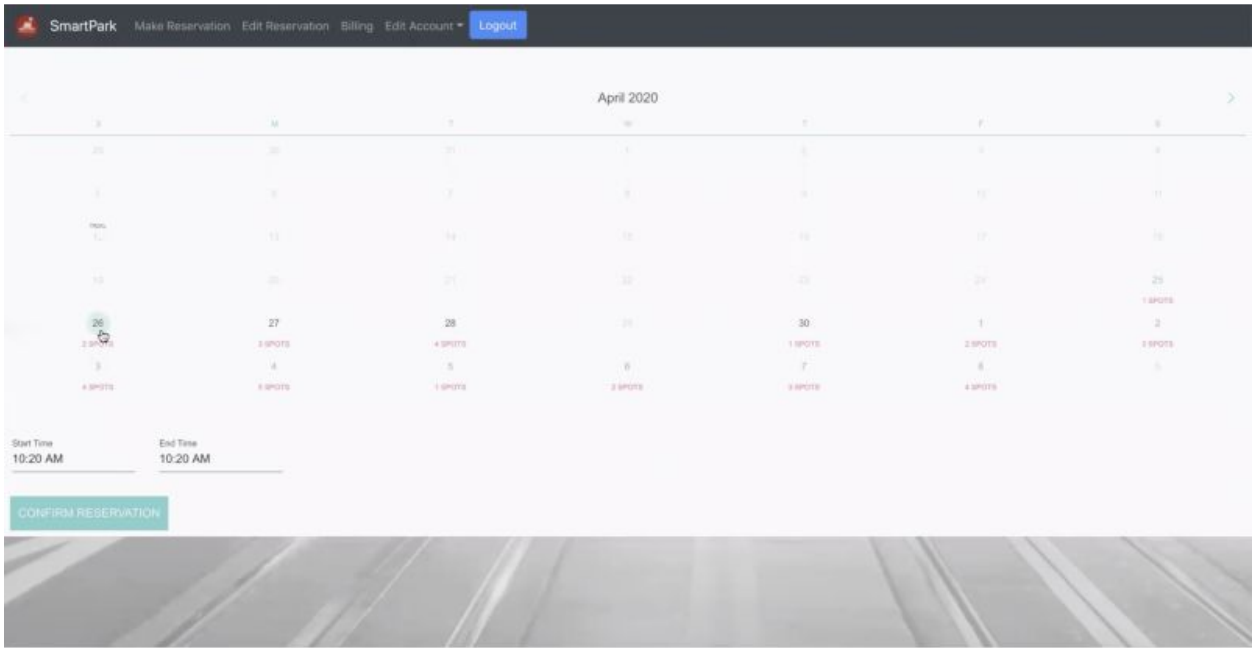
Customer Registration Subgroup

The Customer Subgroup designed the user interface to conveniently connect the customer with various aspects of their account, reservations, and billing information. We wanted the main objective when making the interface to be centered around ease of use. This means we tried not to clutter the screen with irrelevant information or ads of any sort. Starting with the Home Page, we designed the opening text to reflect the name of the user that has signed in with the login button at the top right as shown below.



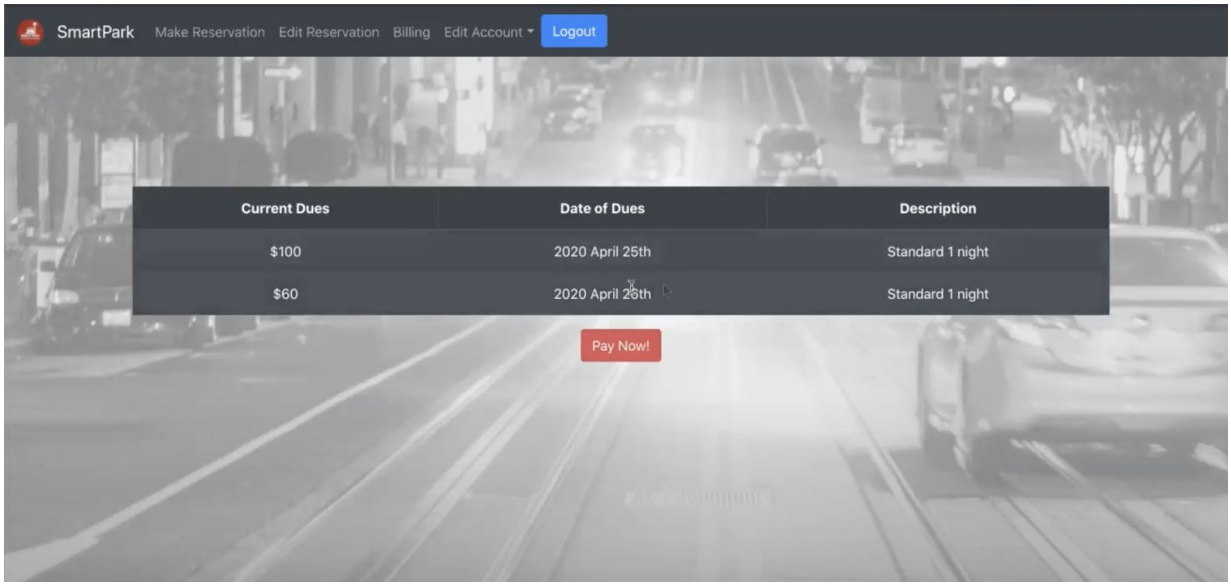
Home Page with Variable Greeting per Customer

After the user would sign in, they are greeted by a navigation bar at the top with different tabs that guide them for pages like making reservations, editing reservations, billing, and account editing (which has options within). If the user clicks “Make Reservation”, they are greeted with a page that displays a calendar which shows vacant spots for any given day. There is also a start and end time selector toward the bottom that allows the user to type in their specified starting and ending time for their reservation. We used a white as the background for the calendar so that it would stand out from the rest of the page since our UI is mostly dark themed. The color we choose to compliment the calendar is mint green which shows up very nicely against the white backdrop.



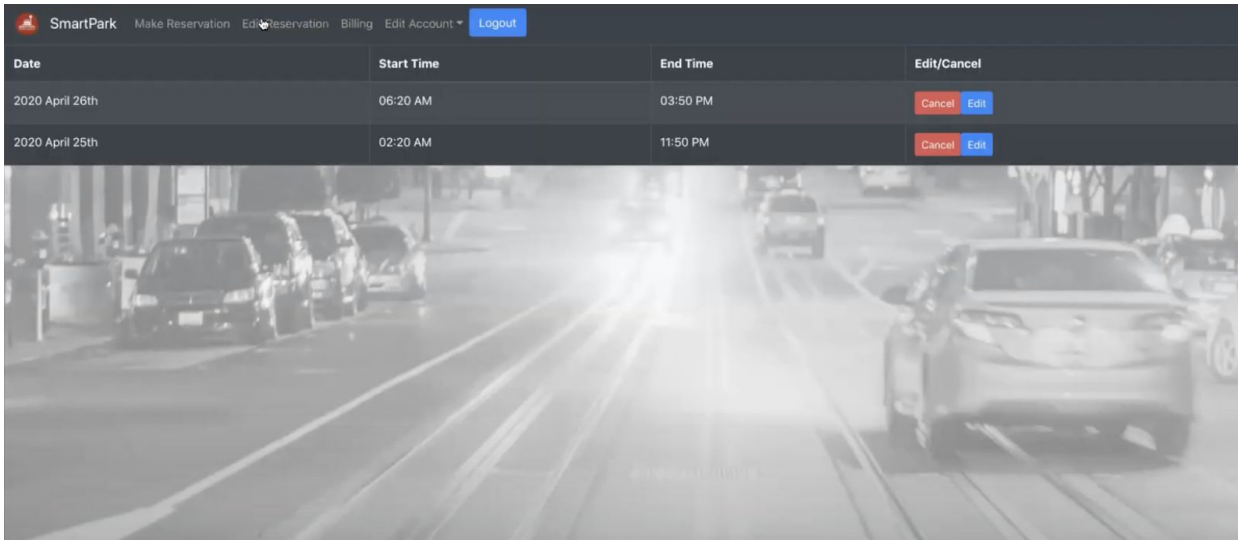
Make Reservation Page

The Billing page allows the user to pay off their bill and in order to show them this, we created a table where each entry is a different bill they have in the MongoDB database. Each column shows a different piece of information related to their respective bill such as current dues, date of those dues, and a description of the costs. Currently we only have a button in place to pay for all the bills under a person's account but in the future we will be adding buttons to pay off specific bills!



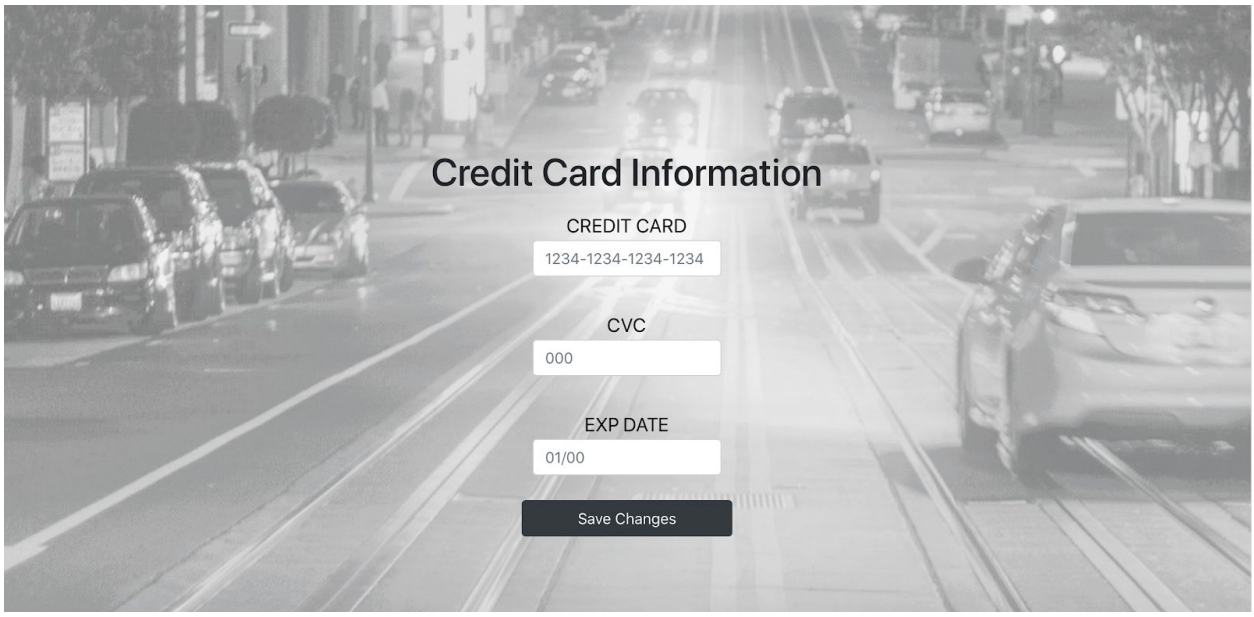
Billing Information Page

The Edit Reservation page lists all the reservations under a user's account and there are options for each reservation such as Edit and Cancel. We chose to make the Edit button green and the Cancel button red just so we can show the contrast to the user. Red is normally associated with something final so we decided to go with that for cancel. The format of the data is in a table style with each row representing a different reservation. The columns show information about a given reservation such as the date of the reservation, start time, end time, and the options discussed above. The start and end time are shown in 12 hour format.

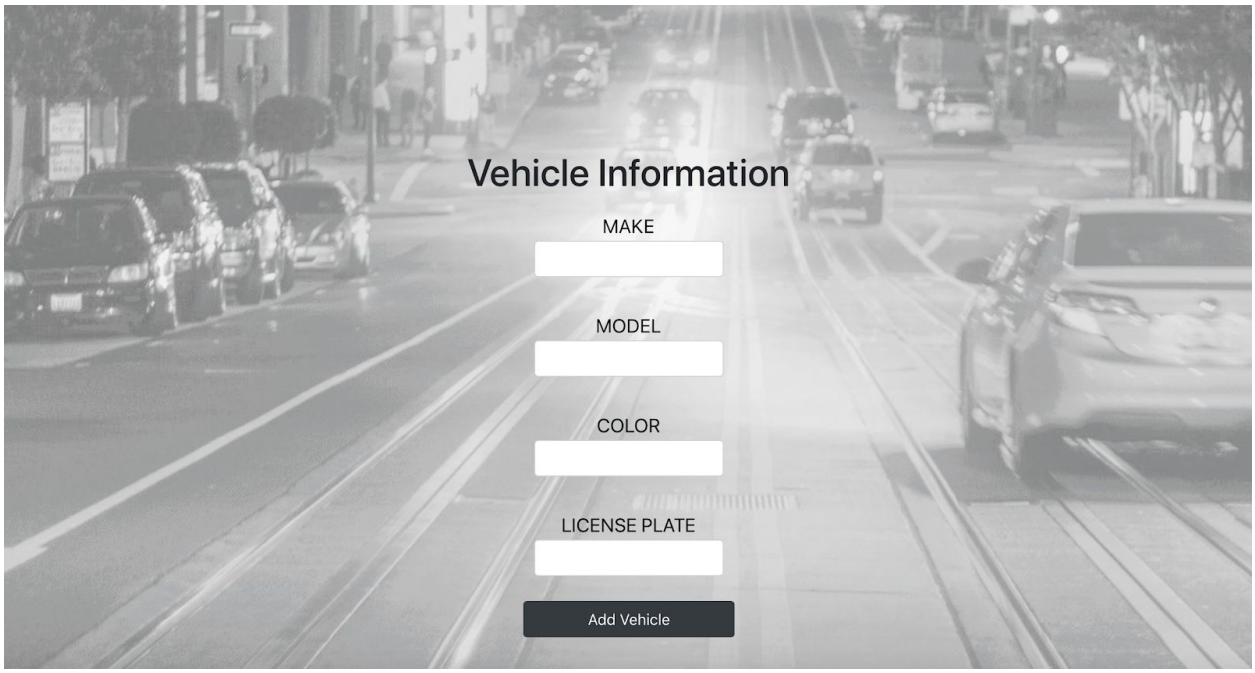


Edit Reservation Page

The last 2 pages are to Add Vehicle and Credit Card Information. The former serves as a way to input vehicle data so that when making a reservation in the future, they have an easy way to select which car they are making the reservation for (This is a feature that will be added in the future to the Make Reservation Page). The latter is a page that allows the user to input their credit card information so that when paying off a bill, the card on their account will automatically be charged. Both pages feature boxes to type in text or numbers if they are required for the box. We also placed placeholder values in each of the boxes on the Credit Card Information page to aid the user to format their data. This helps them understand what needs to be typed for the 'expected' MongoDB entry.



Credit Card Information Page with Placeholder Values



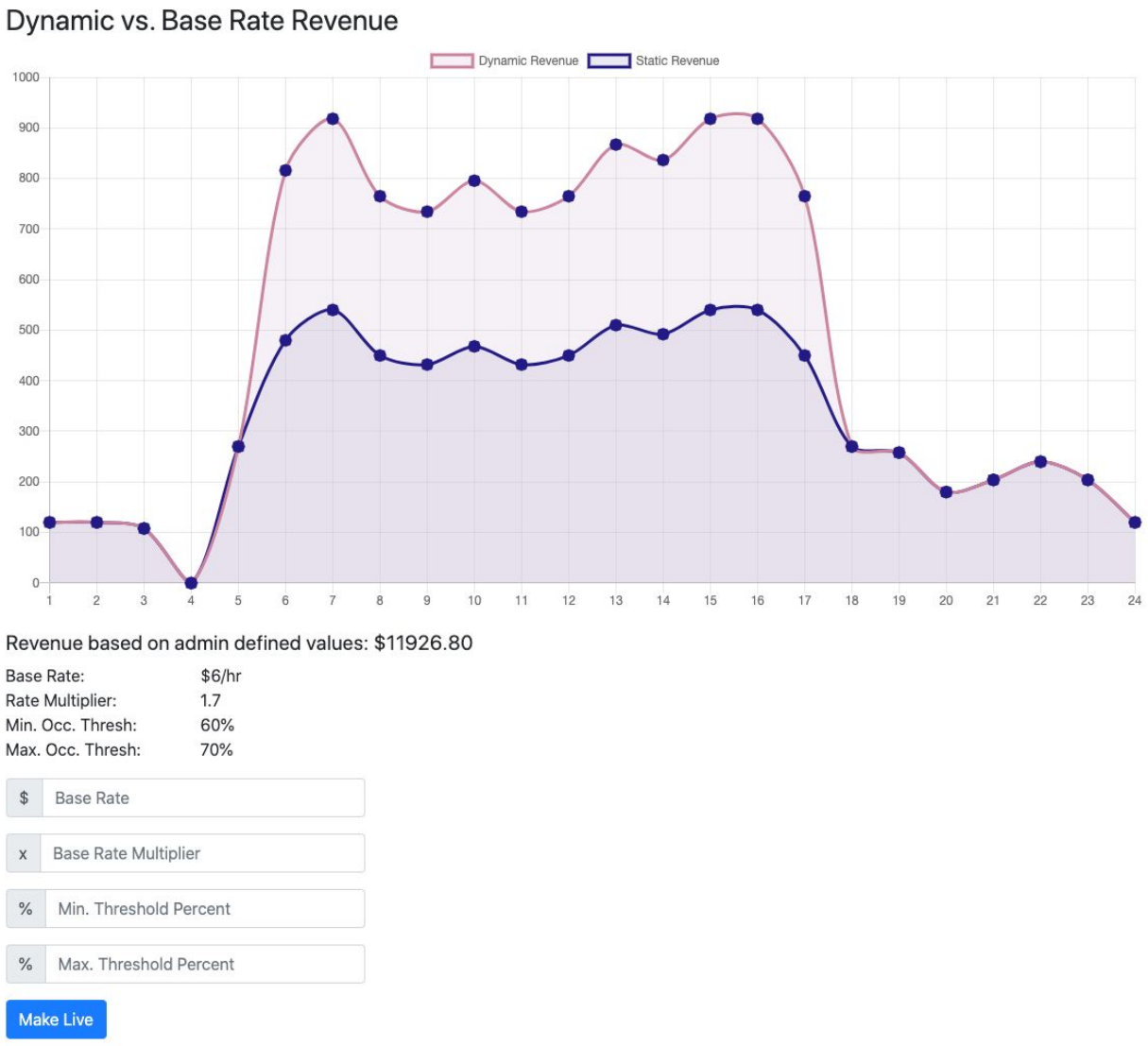
Vehicle Information Page

Managerial / Administrative Subgroup

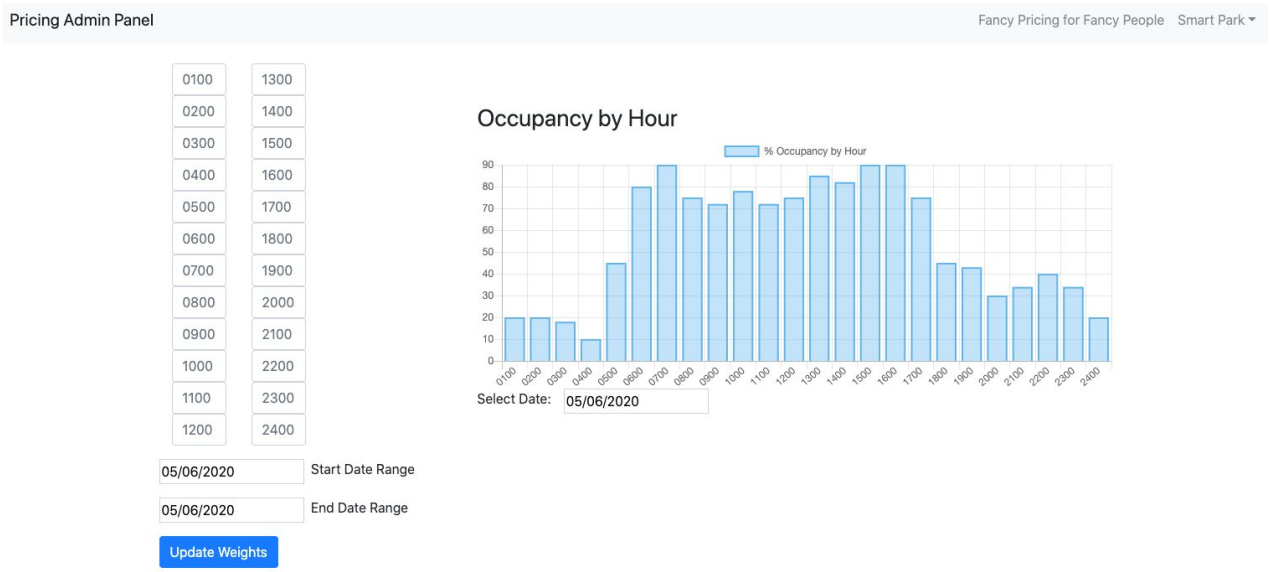
Dynamic Pricing

The dynamic pricing model focused on clean UI design for ease of use and understanding. From the front page of the UI an admin is presented with three options:
View/Edit Price Model,
Edit Weights,
Compare Historic Model with Results

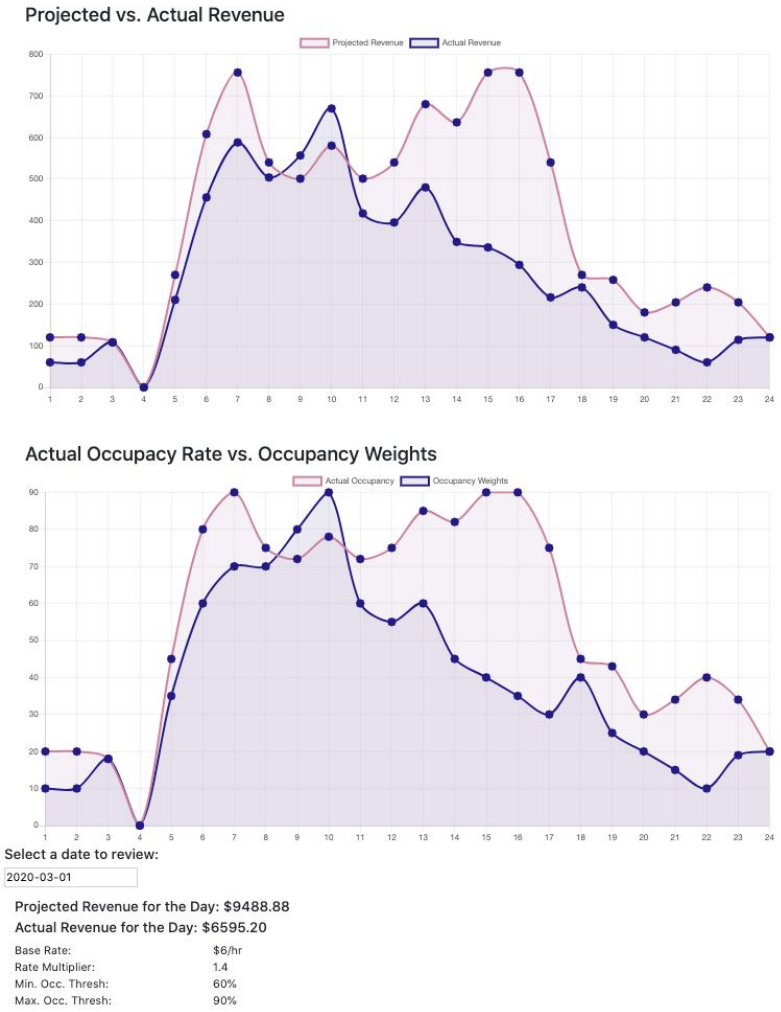
On the View/Edit price model page, the manager is presented with a single graph, below which is included model-specific data for the current day when the page loads. From here the admin can edit the price model data and select a date range for which the price model will be in effect.



The next UI element is the Edit Weights page. Here, the admin can review historic parking occupancy data. This data can be used to make assumptions about future occupancy and the adim can then set those assumptions as weights for a specific date range. On the right hand side of the screen is where the admin may edit their desired weights. Though there seem to be many input boxes, it is designed so that it does not require the admin to fill in all of them. This greatly reduces the number of inputs required of the user.



Finally, the pricing admin page includes the view historic price model page. This page is specifically for looking at historic garage statistics from the perspective of occupancy and revenue. This page also focused on a simple UI experience with minimal inputs.



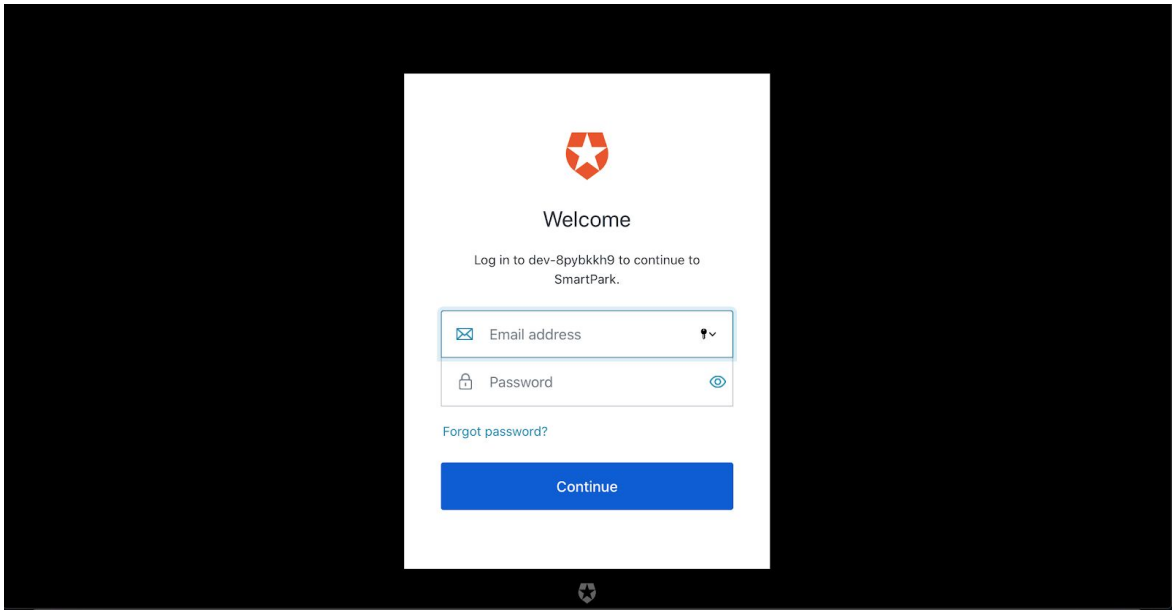
Login

First a manager is required to enter their login credentials.

Example:

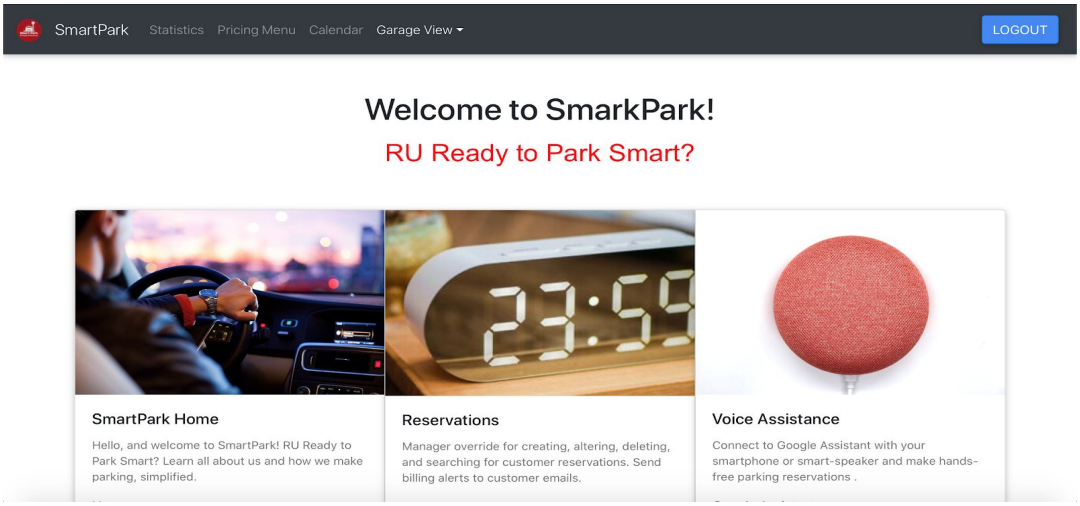
Email: jeffrey@example.com

Password: SmartPark2020



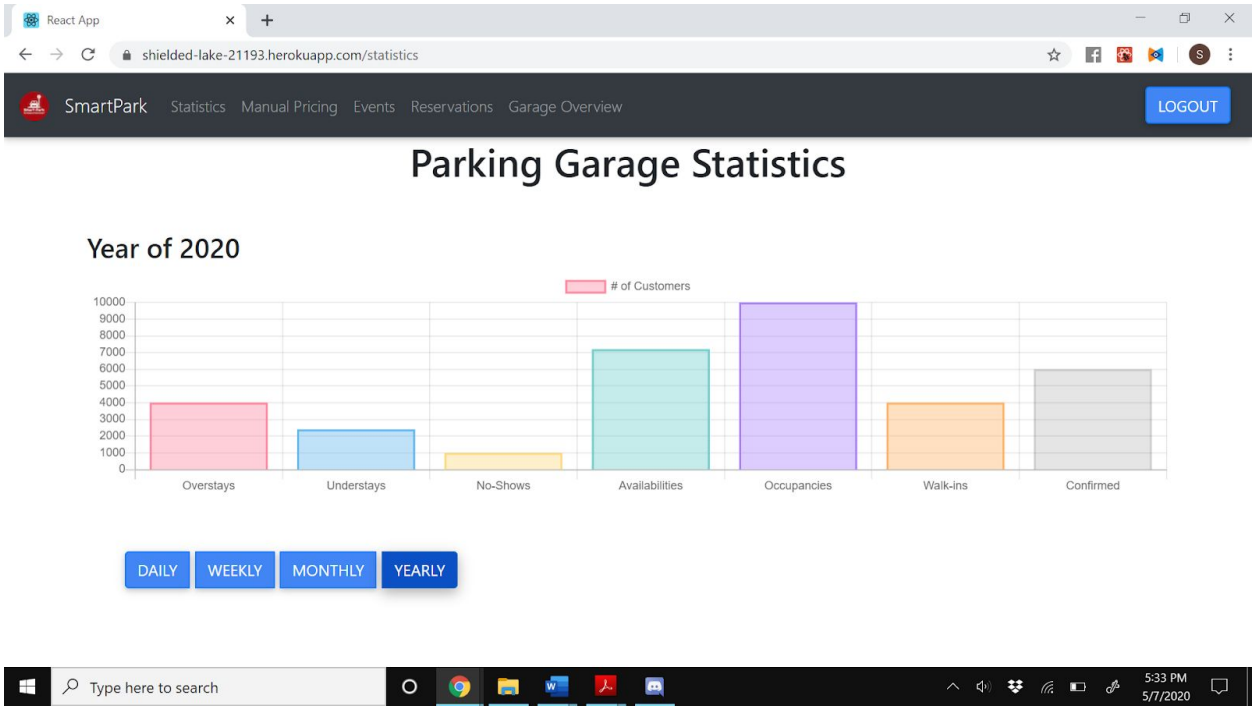
Home

Once a manager logs in they will be directed to the home page where they can go to other pages with other features.



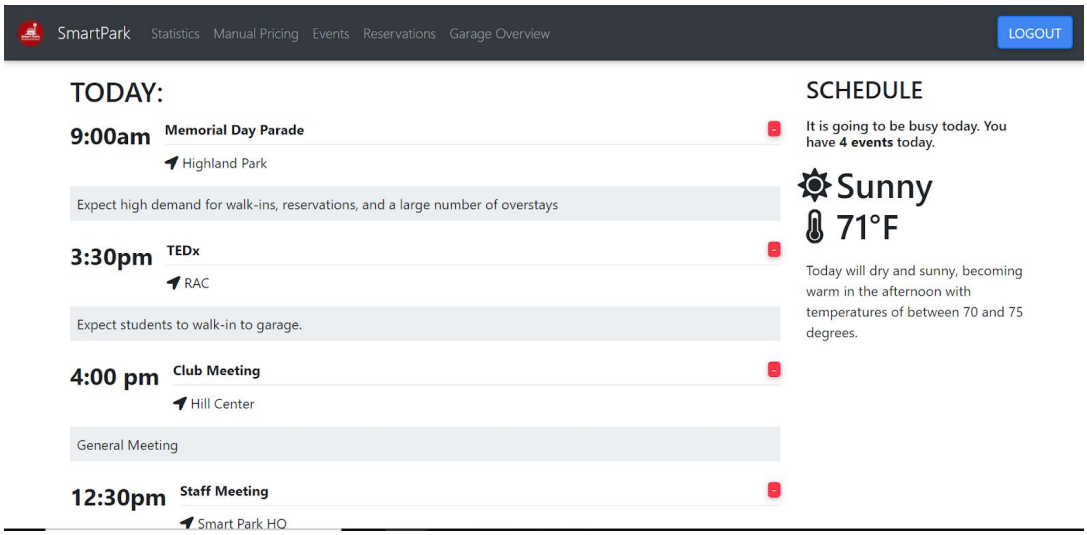
Statistics

This is our statistics page and here you can view business statistics on a daily, weekly, monthly, and yearly time frame. You can also search for a specific data range to get a more accurate and relevant business model.



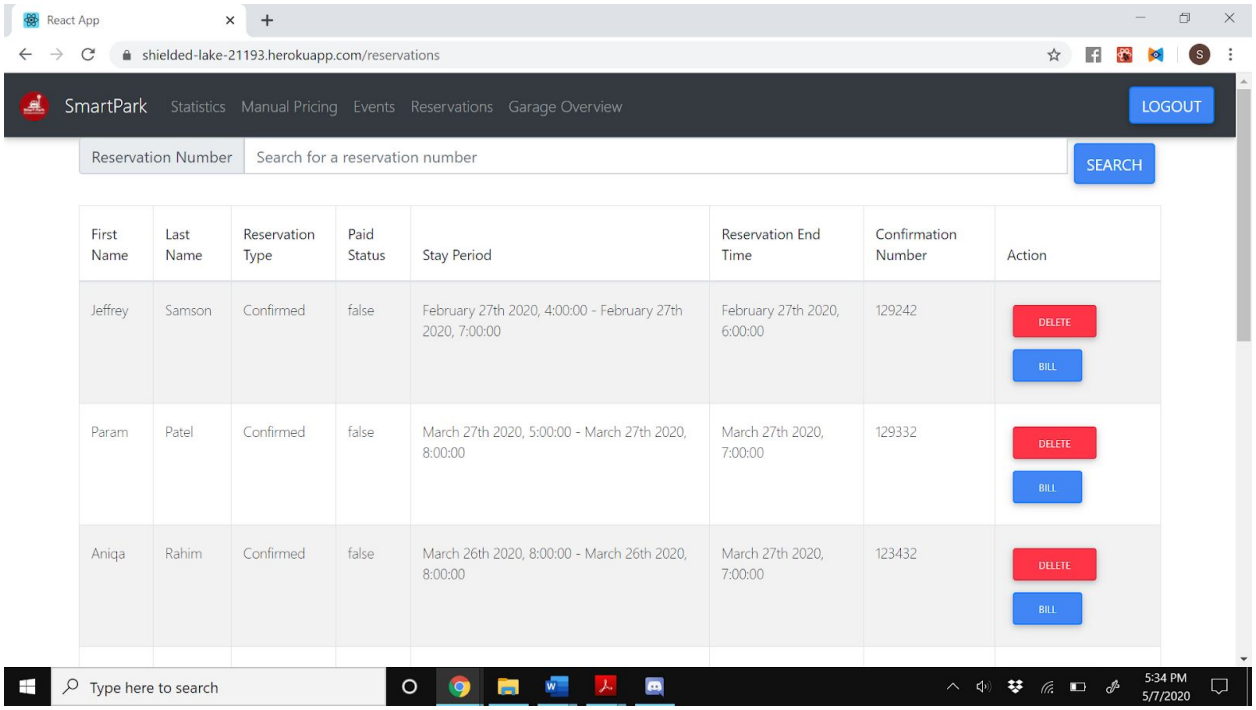
Events

Our events page can store events that would require parking and probably cause an influx of business. Also, you can have events related to your business such as staff meetings.

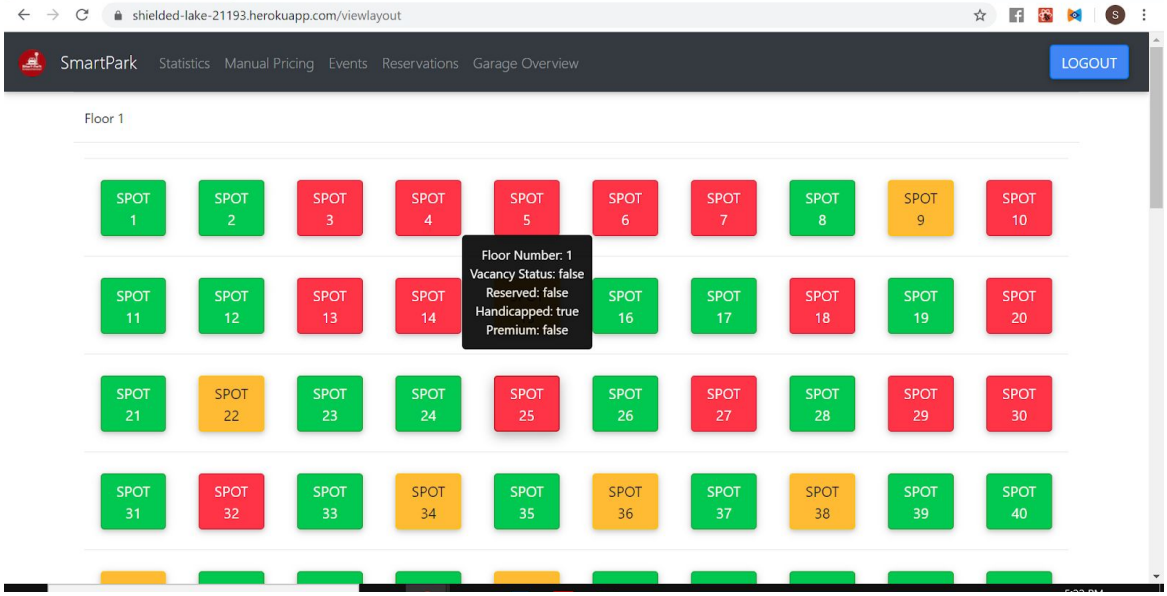


Reservations

The reservations page allows the manager to see the current scheduled reservations and customer details like name and type of reservations etc. Here we can physically bill or delete reservations.

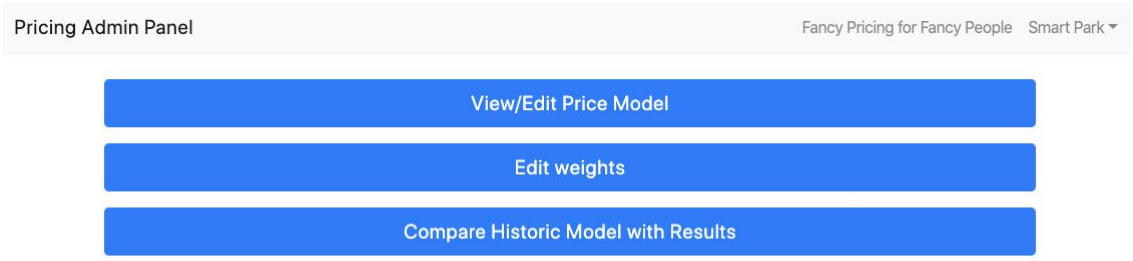


This page allows the user to see the current status of spots in the database. It includes a color scheme that allows the user to determine which spots are currently reserved, vacant, and occupied.

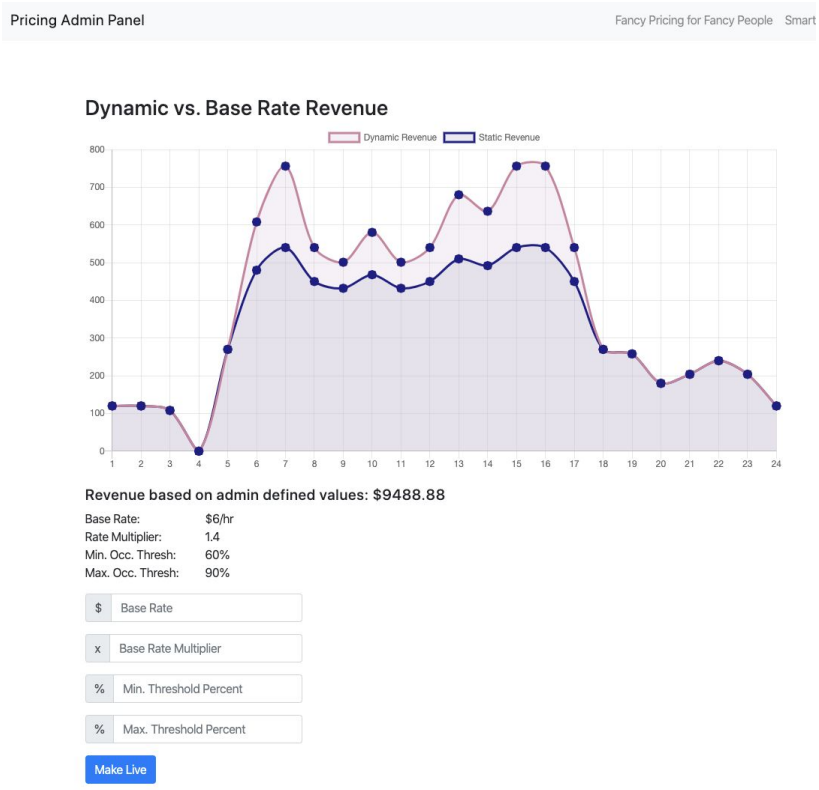


View/Edit Live Price Model

Welcome to the Pricing Administration Panel. Here, you can update the Live Price Model for your garage. The price model is intended to be an easy to understand, easy to use function to incorporate dynamic pricing principles into your parking garage rates and fees.



Under the **View/Edit Live Price Model** option, you'll find access to the user defined fields you'll use to define pricing and a simple graph that displays projected revenue.



The premise of the model is that you, the administrator of the system, may enter your standard hourly parking base rate first. This is the value from which all subsequent inputs work off of. So, if the normal hourly rate for parking in your garage is $6/hr. That's

the rate we'll start off entering in that field. You can always come back and change this value after you've experimented.

The base rate multiplier will determine the maximum rate that you would ever like to charge as an hourly rate. For instance; if your base rate is $6/hr and you would like to charge a maximum of $8.40/hr when the garage is starting to reach capacity, you would enter a multiplying factor of 1.4.

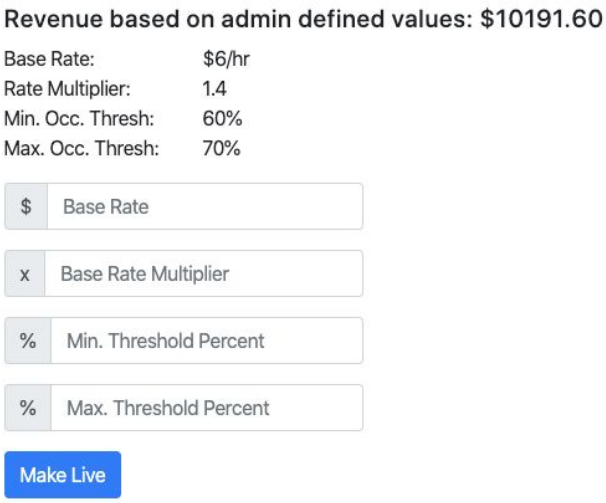
Ex: Base Rate * Multiplying Factor = Maximum Rate Charged
Ex: $6/hr. * 1.4 = $8.4/hr

Next, you can set the threshold values. The minimum threshold is the percentage of occupancy you want to wait for before the dynamic pricing "kicks in". For instance, if you don't want to charge a higher rate until the garage is at least 60% occupied. You would enter the value: 60, into this field.

The maximum occupancy threshold is the percentage of occupancy you would like the maximum rate applied to. For instance, if you want the maximum rate applied to those parking when the occupancy is 70% and above you would enter the value: 70, into this field.

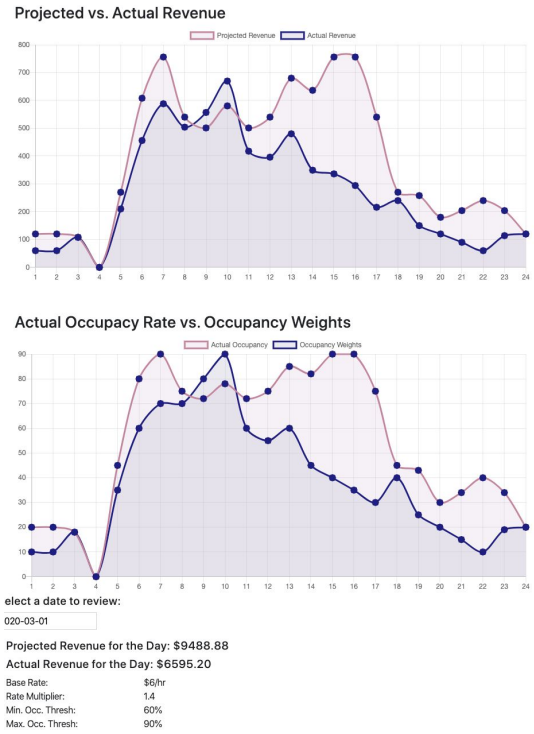
The rate will scale from base to maximum over the range of occupancy between the two values you input and stop scaling thereafter.

The described inputs are shown as an example below.



Compare Historic Model with Results

As the administrator you will want the ability to view the historic results of your price model. This feature is found by pressing the **Compare Historic Model with Results** button.



Under this portion of the Pricing Admin Panel you will find a simple “date picker”. Clicking on the date picker will bring up a calendar. On this calendar you can easily select the date of interest.

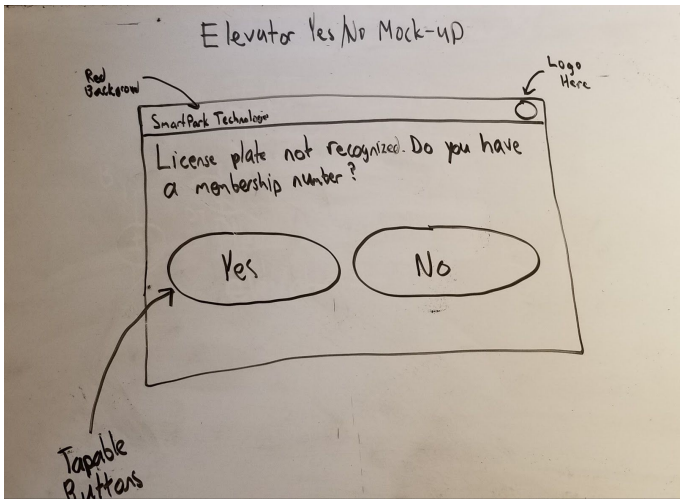
After selecting the date you’d like to see historic price model performance for the graphs and revenue fields will automatically populate. The first photo of this section is an example of what you will see.

The top graph presents you with an overlay of the projected revenue for the day in question along with the actual revenue the garage generated on that same date. This projection was created with the values you input for the price model. The graph charts revenue in dollars on the X axis and hour of the day on the Y axis.

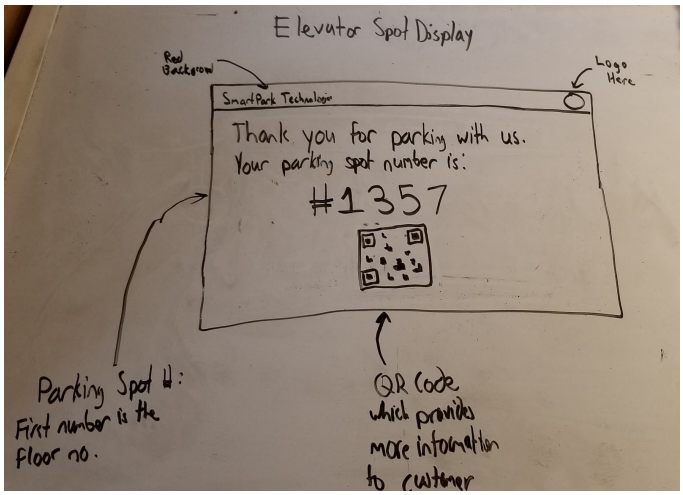
The bottom graph presents you with an overlay of the actual occupancy percentages by hour and the assumed occupancy weights on the date in questions. The occupancy of the garage in percent is charted on the X axis, and the hour of the day is charted on the Y axis.

Operation Subgroup

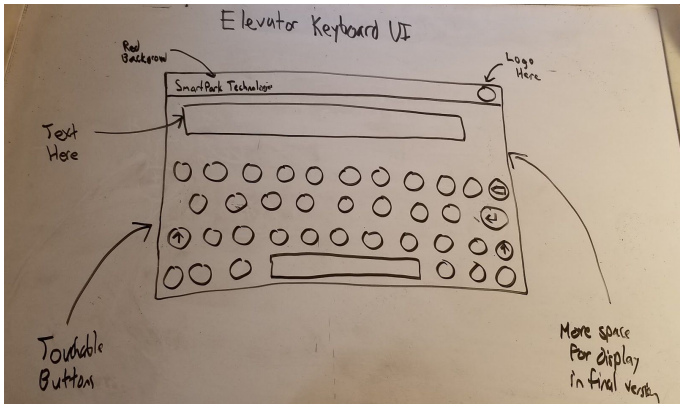
The elevator terminal user interface design went through only minor changes throughout the entirety of the project. The initial design plans for the elevator terminal interface can be seen below:



Mock-up of the prompt for a reservation number, then called membership number.



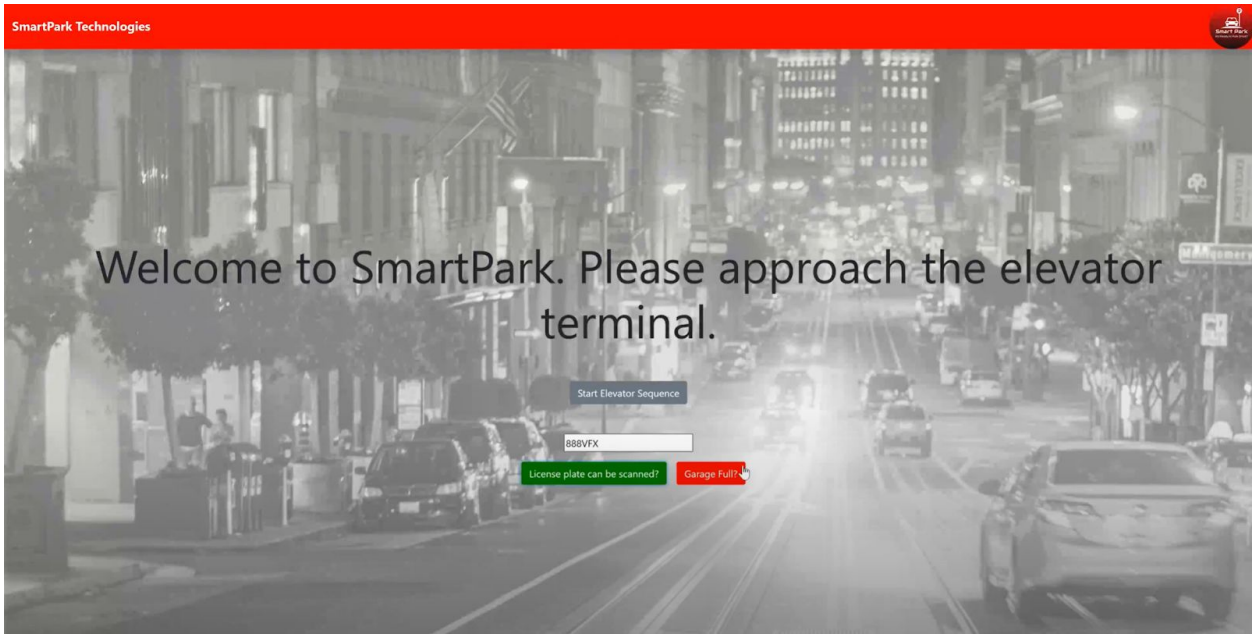
Elevator Spot Display Mock-Up



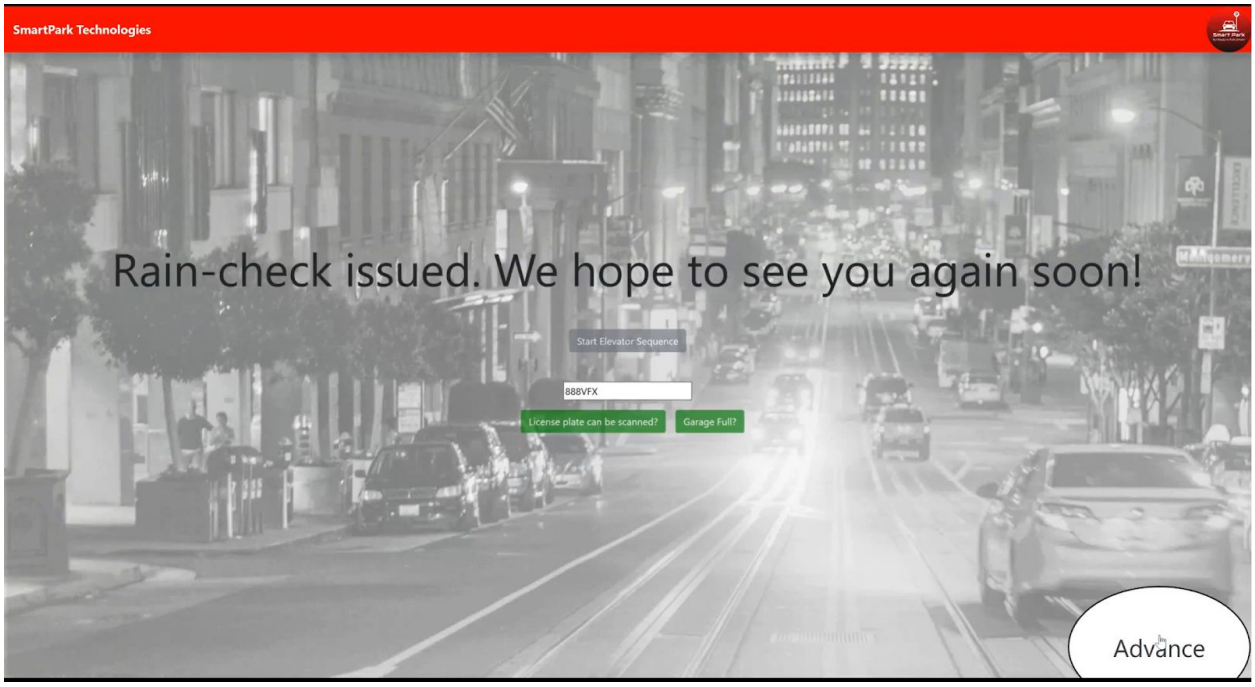
Mock-Up for Keyboard to input values to the system

The early designs for major portions of the elevator terminal screen shown above were to enable speed and efficiency in the elevator terminal to keep the number of vehicles waiting to a minimum. Additional user interface plans from the beginning included the ability to register for an account in the elevator terminal, however due to our design philosophy of speed in the elevator terminal, this feature was cut.

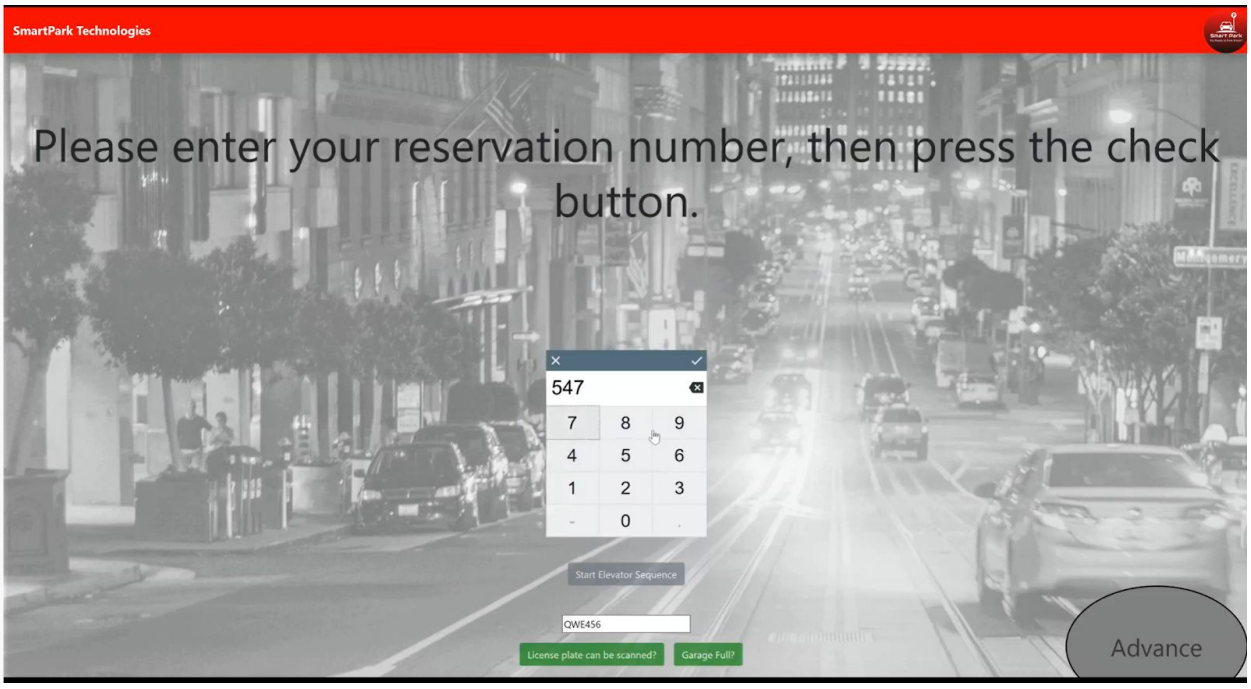
Another important design philosophy for the elevator terminal is the notion of user-friendliness. The buttons for the elevator terminal were designed to be large such that the user interacting with the terminal does not make any accidental button presses. Furthermore, when entering certain important values - such as the reservation number or checking if the user has a reservation number- a confirmation is needed in order to advance from these screens. This is to ensure that the data that the user is entering is accurate, unless entered incorrectly on purpose. With these design philosophies in mind, the final elevator terminal user interfaces are shown below:



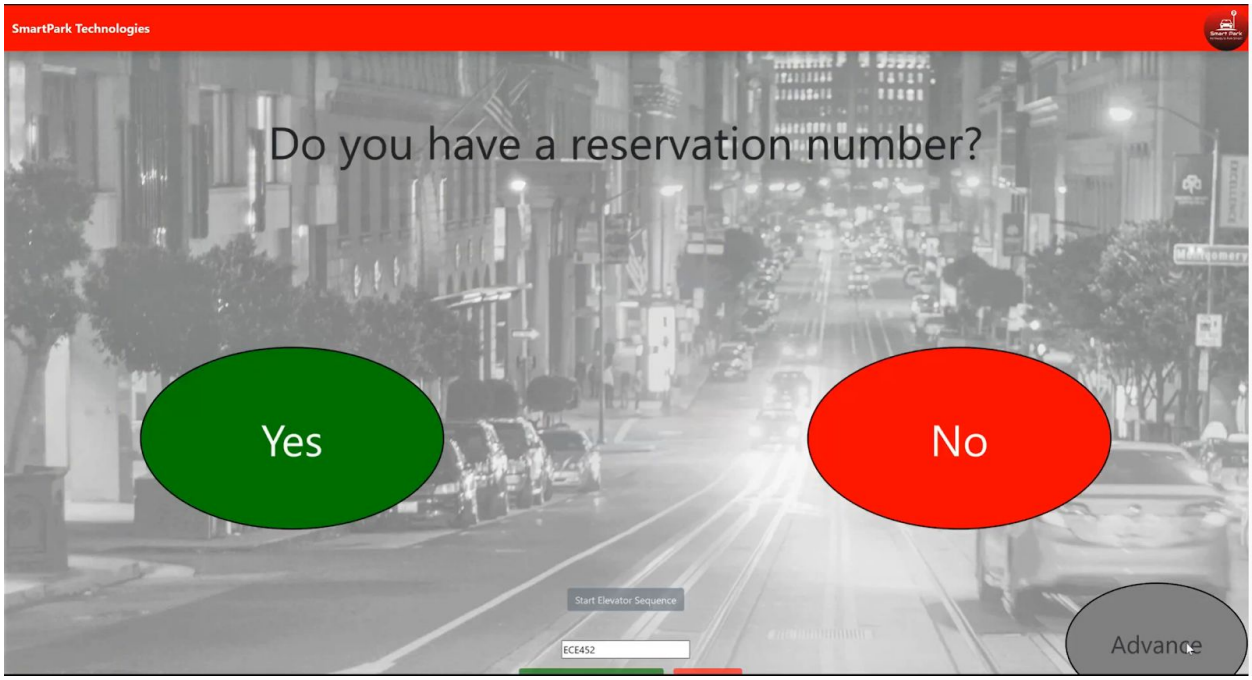
Welcome Screen for SmartPark. (Shown in Demo Mode)



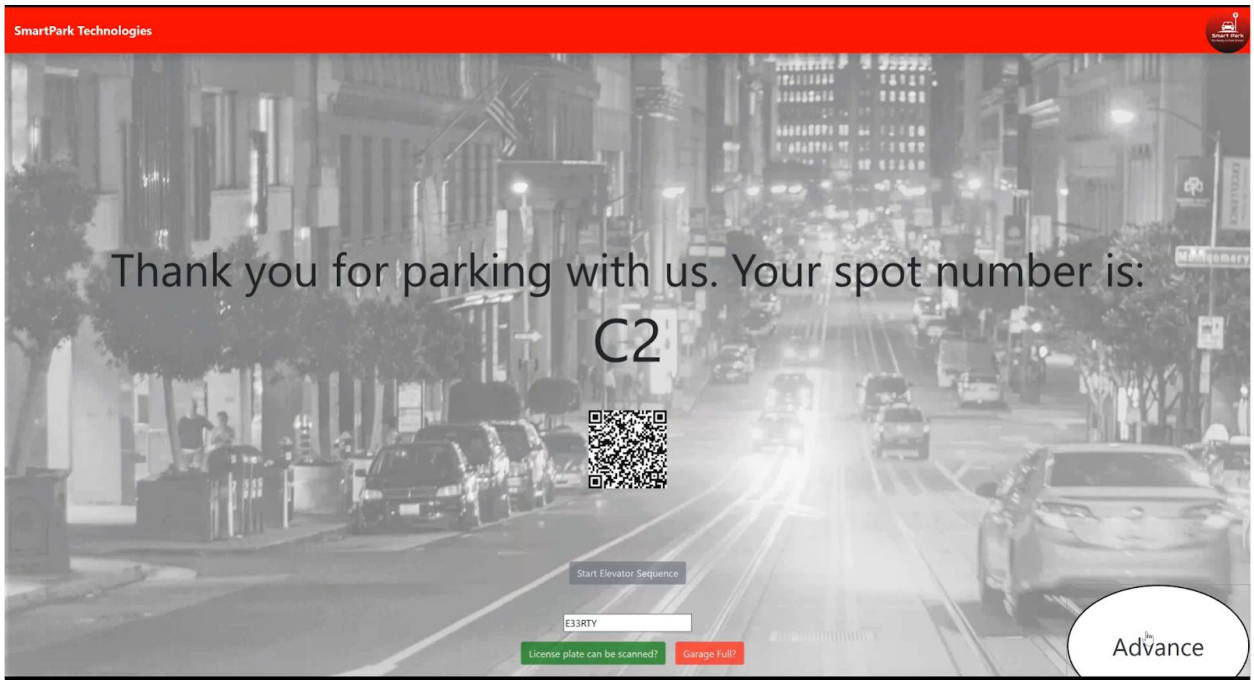
Typical "Advance Through Sequence" Screen. Large "Advance" button to avoid accidental user input.



Entering "Reservation Number", changed from "Membership Number". Also changed to a numpad instead of a keyboard. Note the check button for confirming the membership number.



Prompt for a Reservation Number screen. Kept consistent with plans from the beginning. Large buttons to avoid accidental user input.



Display of spot number including QR code. Little changes from initial design plans.

The changes made from the initial design to the final design overall did not change drastically. Most notable changes consist of the addition of the demo mode, the change from

using a keyboard to restricting access to only a numpad, and the addition of the “Advance” button.

The demo mode was added into the final build of the elevator terminal as a means to test the inputs. Because the camera and license plate scanners are physical devices that we did not gain access to, then these devices were instead modeled through the demo mode. The license plate scanner takes the form of the input field to put in the license plate as well as the “License Plate Can Be Scanned” button. In the real scenario, these buttons would instead be removed and would be given control to the license plate scanner. The camera would take the place of the “Begin Elevator Sequence” button, and instead would begin the scanning sequence once the vehicle entered the garage.

The change from a keyboard from a numpad was made for two reasons: one for efficiency sake and the other due to the removal of registering in the elevator terminal. Since the reservation number only consists of numbers, instead of having a large keyboard to enter the numbers a numpad was chosen instead to speed up the time spent in the elevator. Additionally, since no registration will occur in the elevator terminal, there is no need to have a full keyboard.

Lastly, the “Advance” button was added to meet the design philosophy of ease of use for the user. With the advance button, the user has a moment to make sure that the inputs are correct before advancing.

Elevator Simulation

The elevator simulation was implemented for the first demonstration to outline the possibilities of how the user could interact with the elevator terminal. It highlighted the different scenarios for when the elevator was to be taken and was presented in the first demo and was not reiterated in the second demo as it had served its purpose. It is built very close to the Elevator terminal UI because it served as a prototype for the eventual elevator terminal UI.

Design of Tests

Customer Registration Subgroup

Test-Case Identifier:	TC-Register
Use Case Tested:	UC-1
Pass/Fail Criteria:	The test passes if the data user enters saves to the MongodB cluster.
Input Data:	Firstname, Lastname, Email, Username, Password
Test Procedure	**Expected Result**
Step 1:Input data into the fields to register for an account. Step 2: Check MongodB	When the user inputs the data, if the data is stored successfully when we check the data cluster in MongoDB all the user input data should be stored with a unique id. If all the data saves this means the user has created an account and can proceed to Login.

Test-Case Identifier:	TC-Login
Use Case Tested:	UC- 2
Pass/Fail Criteria:	The test passes if the user enters the correct username and password and is able to access their SmartPark account
Input Data:	Username, Password
Test Procedure	**Expected Result**
Step 1: Create an account and enter the wrong username and password	The system creates a warning window stating which field has an incorrect data type entered into it. The system prompts the user to reenter the correct data.
Step 2: Put in the correct username and password	The user should be taken to their account where they can access their reservations, payment information and other account details.

Test-Case Identifier:	TC-Edit Account
Use Case Tested:	UC-3
Pass/Fail Criteria:	The test passes if the user is able to save vehicle and credit card information into that MongoDB Table.
Input Data:	Credit Card Number, CVC pin, Exp. Date or Make, Model, Color, License Plate
Test Procedure	**Expected Result**
Step 1: Enter Vehicle Information	When the Customer enters their vehicle, Make, Model, Color and License plate and clicks save changes, to check whether the data is saved we can proceed to the MongoDB database. If the customer data appears there that means their information is saved.
Step 2: Enter Credit Card Information	When the Customer enters their credit card number, CVC and expiration date and clicks save changes, to check whether the data is saved we can proceed to the MongoDB database. If the customer data appears there that means their information is saved.

Test-Case Identifier:	TC-Make Reservation
Use Case Tested:	UC-4
Pass/Fail Criteria:	The test passes if the user enters a date and time of their choosing and picks a spot to reserve. We can confirm that their spot was reserved by checking the MongoDB database. If the reservation with the unique Id shows up there then the reservation was created.
Input Data:	Date, Start Time, End Time
Test Procedure	**Expected Result**
Step 1: Pick a day, time and spot Step 2: Check for Confirmation	If the date and time chosen are in the past those dates are already predisabled. Once the dates are chosen and the customer clicks the button: confirm reservation, we can proceed to check the MongoDB database. If the reservation data shows up here then the reservation is saved.

Test-Case Identifier:	TC-Edit Reservation
Use Case Tested:	UC-5
Pass/Fail Criteria:	The test passes if the changes are updated in the MongoDB table
Input Data:	Date, Start Time, End Time
Test Procedure	**Expected Result**
Step 1: Pick the Reservation that is to be edited Step 2: Update Info / Delete the Reservation Step 3: Check MongoDB	The MongoDB table entry is updated with the new attribute in case of edit and completes deletes in case of delete.

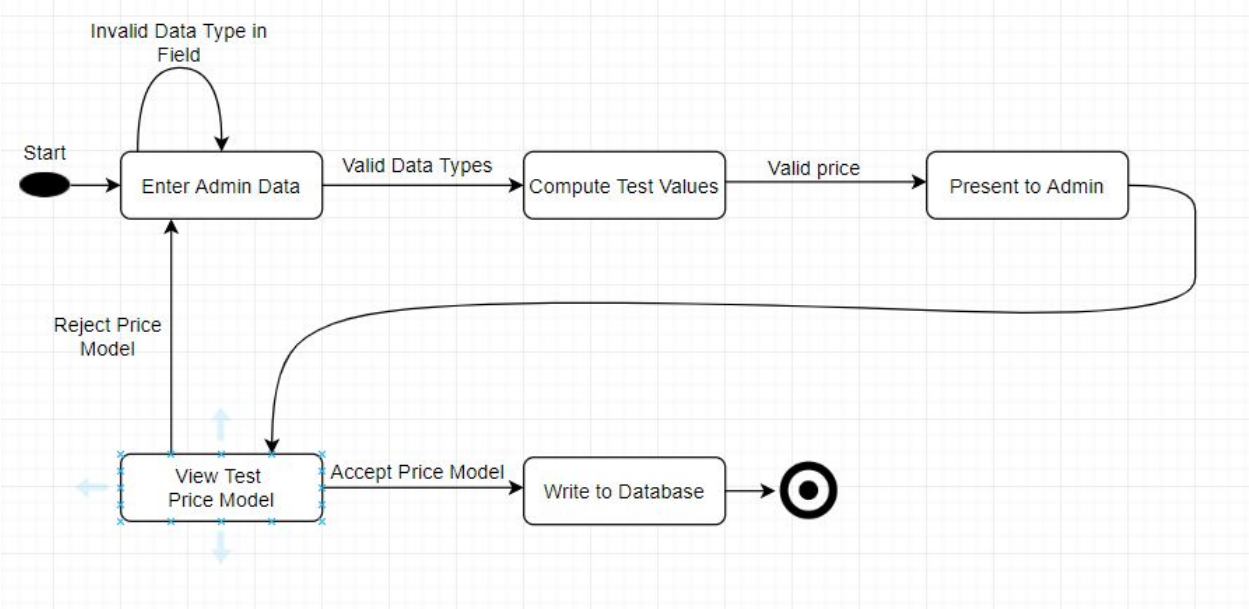
Test-Case Identifier:	TC-Pay Bill
Use Case Tested:	UC- 7
Pass/Fail Criteria:	The test passes if the customer clicks Pay Bill and the customer's current dues disappear from the account.
Input Data:	N/A
Test Procedure	**Expected Result**
Step 1: Click Pay Bill	When the Customer clicks pay bill the current dues should disappear from the screen.A way to check if the customer's Bills have been payed is by checking the MongoDB database and if the customer's dues are deleted there than the customer has successfully paid their bills.

Managerial / Administrative Subgroup

Dynamic Pricing State Diagram and Test Tables

The dynamic pricing model will need to be tested for the following cases: incorrect input from the administrator, incorrect output data output from the mathematical price model and failure to update the database to the administrator's desired price model.

The input and computation state diagram:



From this state diagram we can derive the following test case tables:

Test-Case Identifier:	TC-Dynamic Pricing
Use Case Tested:	UC-10
Pass/Fail Criteria:	The test passes if the admin enters the correct data types in the available field and receives an accurate price model based on that data.
Input Data:	Base Rate, Occupancy Percentage, Minimum Threshold, Maximum Threshold, Base Multiplier
Test Procedure	**Expected Result**
Step 1: Enter incorrect data types into the admin accessible fields. Step 2: Enter the correct data into	The system creates a warning window stating which field has an incorrect data type entered into it. The system prompts the admin to reenter the correct data. The system notifies the admin that a new model is being created with the provided values.

Set Pricing

Test-Case Identifier:	TC-Set Pricing-Front
Use Case Tested:	UC-10
Pass/Fail Criteria:	The test passes if the admin enters valid values for hourly, peak-time, and overstay rates. Boundary cases such as no values entered for a specific input will also be considered.
Input Data:	hourlyRate, walkInRate, cancelFee, overstayRate, guaranteedPrice
Test Procedure	**Expected Result**
Case 1: Administrator enters values for the pricing and presses submit Case 2: Administrator enters an invalid number	Case 1: The user interface indicates that the changes have been accepted. Case 2: System indicates that the value is invalid

Test-Case Identifier:	TC-Set Pricing-Back
Use Case Tested:	UC-10
Pass/Fail Criteria:	The test passes if valid values for hourly, peak-time, and overstay rates are successfully entered into the database.
Input Data:	hourlyRate, walkInRate, cancelFee, overstayRate, guaranteedPrice
Test Procedure	**Expected Result**
Case 1: Values for hourly, walk-in, cancellation, overstay, and guaranteed pricing are sent to the database via a post request	Case 1: The database is updated with the new prices. This change is reflected on MongoDB Compass.

View Garage

Test-Case Identifier:	TC-View Garage-Front
Use Case Tested:	UC-11
Pass/Fail Criteria:	The test passes if the reservation page can successfully display the number of available spots and current reservations
Test Procedure	**Expected Result**
Case 1: Administrator requests to view all garage spots	Case 1: The system displays all parking spots on the user interface

Test-Case Identifier:	TC-View Garage-Back
Use Case Tested:	UC-11
Pass/Fail Criteria:	The test passes if the system can successfully retrieve records from the database given fixed search criteria (no user input)
Input Data:	Available Spots, Current reservations
Test Procedure	**Expected Result**
Case 1: A fixed search query is entered (for example, all occupied spots)	Case 1: The system responds with all entries that match the fixed search criteria

View Garage Statistics

Test-Case Identifier:	TC-View Garage Stats-Front
Use Case Tested:	UC-14
Pass/Fail Criteria:	The test passes if the reservation page can successfully display statistics in a graph given fixed inputs and search criteria
Input Data:	Date Range (Button Click)
Test Procedure	**Expected Result**
Case 1: Administrator requests to garage statistics for a fixed date range	Case 1: The system displays the graph from data from aggregations made in the data range when a fixed date range is chosen

Test-Case Identifier:	TC-View Garage-Back
Use Case Tested:	UC-14
Pass/Fail Criteria:	The test passes if the system can successfully retrieve records from the database given fixed search criteria (no user input)
Test Procedure	**Expected Result**
Case 1: Admin chooses a fixed search date range	Case 1: The system responds with all entries from the database that match the fixed search criteria

Elevator Operation

Park

Test-Case Identifier:	TC-Park
Use Case Tested:	UC-17
Pass/Fail Criteria:	The test passes if the elevator terminal is able to direct the customer with the reservation to the nearest empty spot.
Input Data:	Available Spots, Current reservations, Registration Number
Test Procedure	**Expected Result**
Case 1: The license plate is scanned Case 2: The user of the scanned license plate has their reservation searched Case 3: The user asks to enter their reservation number Case 4: The user enters their reservation ID into the numpad	Case 1: The user is told the license plate was scanned successfully or unsuccessfully Case 2: The user is told that their reservation either exists or doesn't exist. Case 3; The user is redirected to the numpad Case 4: The reservation is either found in the database or not found

Update Parking Spot Status

Test-Case Identifier:	TC-Update Spot Status
Use Case Tested:	UC-18
Pass/Fail Criteria:	The test passes if the system can successfully update the recently filled spot.
Input Data:	Available Spots, Reserved Spots, Parking Sensor
Test Procedure	**Expected Result**

Case 1: Data from the parking sensor is collected and sent to the database	Case 1: The database receives this signal from the parking sensor and marks the spot as occupied or vacant depending on its previous state.

History of Work, Current Status, and Future Work

Customer Registration Subgroup

History of Work (Previous Goals + Deadline Evolution)

1. Completed all Interfaces - Login, Signup, Make a Reservation, Edit a Reservation, Bill Payment, Edit Account, Dynamic Pricing
    - This was estimated to be finished long before the demo but in reality, we only finished a few days before the demo
    - Initially, separate functions were created with each page but integration to the backend required react components instead of functions
    - This set us back quite a lot as the plan to integrate the front and back end needed to be pushed
2. Setup and Testing Backend - Reservations, Billing, Vehicle Information, Credit Card Information, Dynamic Pricing
    - Backend completion was also finished prior to the demo but after deciding to add new routes for reservation updating and accessing, bill deleting and accessing, etc.
3. Integration of Interfaces to work with Backend Routes
    - To integrate the frontend and backend we used axios which only integrates with React Components and not functions
    - This causes a problem in setting up and accessing the current values along with displaying and updating the values inputted

Current Status (Key Accomplishments + Ongoing Tasks)

1. Demo #1
    - Getting critical feedback and a second opinion on all of our work was very important to us, this is a critical accomplishment because we wanted to showcase all of our hard work and
2. Finished updating UIs
    - All the issues with the React Components were sorted out earlier this week, including the new pages for the added use cases 3 and 5
3. Finished all backend routers
    - Completed new routers for use cases 3 and 5 to preview for Demo #2
4. Preparing for Demo #2
    - Getting integration and testing done for all components (front and back) later
    - Discussing and deciding key components to re-address and newer ones to showcase (focus on showing improvement on prior design along with new developments)
    - Writing and Recording Scripts for Demo Video

Future Work (Other Directions)

1. Subgroup Assembly
    - Currently all of the subgroups operate individual of each other and have plans for complete integration such that SmartPark becomes one entity
2. Dynamic pricing adjustment to create more automated actions according to historic data.

Managerial / Administrative Subgroup

History of Work (Previous Goals + Deadline Evolution)

1. Completed all Interfaces - Login, Home Page, Statistics Page, Pricing Page, View Garage Page, Reservations Page
    - The front end was created pretty easily and ahead of schedule so we used a good amount of our time to coordinate with other groups and develop the Google Assistant and backend.
2. Setup and Testing Backend - Reservations, Billing, Vehicle Information, Credit Card Information
    - We set up a database to hold customer and reservation information such as names, spot numbers, reservation times etc. We created this database using MongoDB and wrote the backend so that our front end can interact with the corresponding data.
3. Integration of Interfaces to work with Backend Routes
    - We used Axios and Mongoose to have the backend communicate with the frontend.

Current Status (Key Accomplishments + Ongoing Tasks)

4. Demo #1
    - Getting critical feedback of the project while also deciding whether we as a team are headed in the proper direction with the project. We focused on what made our project different from previous projects by focusing on the newly implemented ideas.
5. Finished updating UIs
    - We worked out some front end issues to make sure that it was accurate with what the other subgroups have also been developing.
6. Finished all backend routers
    - We have routed our backend so that it is properly connected to the front end. And have tested the web pages for expected results and usability.
7. Preparing for Demo #2
    - Getting integration and testing done for all components (front and back) later
    - Discussing and deciding key components to re-address and newer ones to showcase (focus on showing improvement on prior design along with new developments)
    - Writing and Recording Scripts for Demo Video

Future Work (Other Directions)

8. Subgroup Assembly
    - Each subgroup has their own database and does not interact with each other. We wish to coordinate our efforts so that each subgroup's product can work together in unity.
    - Issue Rain Check capability
    - Web scraper that loads events from Facebook or Google to the database

Elevator Operation Subgroup

History of Work (Previous Goals + Deadline Evolution)

1. Completed all Interfaces - User Interface Terminal for the elevator group. We had two people develop a UI but we ultimately decided to go with one as it had more features.
2. Completed Dynamic Pricing - This was tough and took a lot of effort for Charles to get it done. There was collaboration between other groups on this as well.
3. Setup and Testing Backend - Scanning license plate and checking to see if the license plate is in the system.
    - The back end was finished before the demo, but we were not able to successfully integrate it with the front end in time.
    - We did however manage to use HTTP requests via Postman to test and see if it worked, which it did.
    - We wrote several back end functions with express to fetch the certain parameter we wanted that was associated with the user in the system.
4. Demo #1
    - The feedback was very good for our group. We were pleased with what our reviewers thought of our project. They also gave very useful input as to what else we could implement.
    - We had another meeting to decide what new features we were going to implement.
5. Finished updating UIs
    - We made sure that the Frontend not only was functional but was aligned properly and
6. Integrated front end with back end
    - Even though we went with one UI, we integrated the first UI with the back end first. This is because it was a relatively simple UI, and the people integrating them wanted to gain a better understanding of how this is done.
    - Integrated the front end with the more complex UI to the back end. This did not take as long as we had more experience doing this. It is a good thing too, because there were things that we learned from integrating the simple UI first.
7. Demo #2
    - Reviewing the different pages we created and continuing to test the frontend, backend, and the connection between the two.

- Addition of a QR code scanner to allow the customer to keep track of their parking spot.
- Review our databases and make sure that it has all the proper information needed to show the functionality of the pages.

Current Status (Key Accomplishments + Ongoing Tasks)

- Archiving all of the elevator write ups and code for the e-archive.

Future Work (Other Directions)

- Continuing to integrate the elevator group with the other groups.
- Update the status of a parking spot when somebody parks/leaves, and send this information to the manager group.
- Adding more functionality to the QR code, including the ability to open up Google Maps to find the customer's parking spot relative to their location.

Appendix

This appendix includes the question Professor Marsic asked during our Demonstration 2 Q&A and emailed to SmartPark.

1. Clarification on Specific Changes to the First Part of the Demo
    - *The Customer Group has made many significant changes compared to the first part of the demo. We can now parse information from Auth0 to get specific account information that we can search for in the MongoDB table. This is all automatic, the make a reservation now parses the data that we click in the calendar and makes its own post requests unlike like time when we still used Insomnia to make them. This is the same case for the billing, the billing page can now display all of the individual bills and not the total, and the page now makes a delete request of all of the bills at one time and can only delete the ones with that particular 'email' attribute.*

2. In Part 2 of the video, how were the statistics generated - was it random, or based on some real-world model? Did you use the Uber data mentioned in the last part?
    - *The statistics in the Statistics page were calculated through aggregations in SmartPark's MongoDB database, in the Reservations data collection. When a reservation is stored into the database, the system parses through all the relevant information and the number of customers in the relevant categories in the graph is incremented. There are currently reservations in the database and the current statistics are based on those.*

3. Example of Dynamic Pricing: If the number of overstays is higher than "usual" system suggested to increase the overstay fee.
   The number of availabilities is higher than "usual" --> decreases the hourly price.--> As a result of the dynamic pricing interventions, the number of occupancies becomes "normal".
   Question: How is "usual" and "normal" calculated? Is it some kind of an *average* and if so, over what interval (recent week, month, ..., or ever since the garage started to operate)??Did you specify these calculations in your Report #2?
    - *We have to expect that humans are humans and that at least some of the customers will overstay or not show up. So, we determined "usual" and "normal" based on the number of reservations currently stored in our Reservations database. Usually if about half of our customers that have reservations over that specific time period fit into our negative categories, (No-Shows and Overstays), we say that the number of people in the specific category is higher that usual. For instance, for the day in the demo, we had 24 reservations in the database, and 12 overstays, so we said that the number of overstays is high. And then we linked to the Dynamic Pricing page so that the Manger can make changes as they see fit. We did not specify these calculations in the Report #2 because we had decided on implementing this feature very late in the process, after we had submitted Report #2.*

4. In the Event View Page (Use Case UC-19), how is the expected information generated? such as "Expect high demand for walk-ins, reservations, and a large number of overstays"?

- *We took the feedback from the last demo, that said we didn't provide any insights with the data that we collected, and so we decided to add something that would show an example of how the data could be used but didn't have the plans to fully implement it. Right now, the manager would manually input the warnings based on their own common sense. In our example in our demo, obviously a parade would gather a lot of people, all who need parking in a parking deck as there will be no street parking available. So we can expect high demand for walk-ins and reservations. Since many people would get caught up in the festivities, or perhaps they would get lost, or get stuck in foot traffic, the number of overstays would be high as well.*

5. Are these predefined template sentences that are somehow selected by the algorithm based on the nature of the future event ...?I'm not sure I got it right from the video, but it looks like this all information is entered in advance as an advisory information, and there is no system-based inference of the expectation?
    - *For future work, perhaps some code could be implemented to parse the title and find the word, "parade" and have the system automatically make this assumption and display advisory information on the webpage. For now, no system-inference has been implemented.*

6. In Garage Overview Page (Use Case 11), are the spot occupancies updated automatically by the system based on some kind of sensor input, or manually?
    - *The Elevator Terminal is supposed to update the spot occupancy status using license plate scanners and weight sensors*

7. Also, yellow spots are "reserved", but this is a vague term because the reservation may be immediate or for the next day, or far into the future ...?
   Please clarify how the system decide to color a given spot as "yellow" ~ "reserved"
   I see that your database view only shows a Boolean value True/False for reserved, without saying when is the time for which the reservation is made.
    - *The reserved boolean refers to if there is currently a reservation for this spot for the current time but not yet occupied by a customer. As of now all of those values were randomly generated as an example, but they would be connected to the reservation system and elevator terminal which would update the spot status as customers make reservations and exit the terminal.*
        isVacant = Green
        isReserved = Yellow (isVacant == false)
        Not isVacant && not isReserved == Red (Customer is currently parked)

References

1. Draw.io - UML Diagrams, Charts and Diagram Templates
2. SOLID Design Principles
3. Composition Over Inheritance (aka Composite reuse principle)
4. Don't Repeat Yourself - DRY
5. Inversion of control - IoC (Hollywood Principles)
6. YAGNI - You Aren't Gonna Need It
7. Law of Demeter - LoD (aka Principle of Least Knowledge)
8. Principle of Least Astonishment - PoLA
9. Minimum Viable Product - MVP
10. An overview of HTTP - HTTP
11. GUI Design: Sun Microsystems, Inc. Java Look and Feel Design Guidelines. Mountain View, CA, 1999. Available at: Oracle Java Technologies

Customer Problem Statement

12. McCoy, USA Today, "Drivers spend an average of 17 hours a year searching for parking spots"
13. Alexa Skill, Who is Amazon Alexa and what is an Alexa skill?
14. Pedestrian Injuries in Parking Garages, Pedestrian Injuries in Parking Lots and Garages: Get Fair Compensation
15. IBISWorld, "https://my.ibisworld.com/us/en/industry/81293/industry-at-a-glance"
16. Reinventing Parking, reinventingparking.org/2013/10/is-30-of-traffic-actually-searching-for.html
17. Project Description, Software Engineering Course Project Parking Garage/Lot
18. Pedestrian Safety, Pedestrian Safety | Motor Vehicle Safety
19. Pedestrian Statistics, 2017 Data: Pedestrians

Mathematical Model

20. Nature.com - City-scale car traffic and parking density maps from Uber Movement travel time data
21. Four Week MBA - Dynamic Pricing: Is The Price Tag Legacy Coming To An End?
22. Price Elasticity of Parking (PDF) The price elasticity of parking: A meta-analysis

Plan of Work

23. On the map: product roadmap templates and tips - How to build a product roadmap: Tips, templates, and examples
24. Gantt Chart - Schedule your Projects | Gantt Chart History and Software
25. Gantt Chart Basics: What it is, Benefits, & Alternatives · Asana

Miscellaneous

26. Object Constraint Language (OCL): https://www.omg.org/spec/OCL/2.4/PDF
27. Microsoft Powerpoint Software Engineering Lectures 17 and 18