Fort Hays State University

CSCI 441 - Software Engineering

Group 10

Network of Automated Parking Garages

Report Three

November 20, 2022

Team Member:	*Email:*
Dustin Threet	dlthreet@mail.fhsu.edu
Justin Henley	jahenley@mail.fhsu.edu
Victor Munoz	vemunoz@mail.fhsu.edu

Project Website: https://fhsu-csci-675-group-10.onrender.com

Individual Contributions Breakdown

	Dustin Threet	Justin Henley	Victor Munoz	*Hannah Carr **
Project Management	33%	33%	33%	X
Summary of Changes	33%	33%	33%	X
Section 1	25%	25%	25%	25%
Section 2	25%	25%	25%	25%
Section 3	25%	25%	25%	25%
Section 4	29%	29%	29%	13%
Section 5	29%	29%	29%	13%
Section 6	25%	25%	25%	25%
Section 7	29%	29%	29%	13%
Section 8	29%	29%	29%	13%
Section 9	25%	25%	25%	25%
Section 10	25%	25%	25%	25%
Section 11	29%	29%	29%	13%
Section 12	29%	29%	29%	13%
Section 13	29%	29%	29%	13%
Section 14	25%	25%	25%	25%

** Hannah Carr dropped out of this class and group on 10/29/2022. She remains credited for any existing work from Reports One and Two.*

Table of Contents

Summary of Changes

Section	Changes
1	● Removed language about supporting multiple garage companies with a single deployment. Later decisions oriented the project to require a single system per garage company.
2	● Update User/Garage Operator definitions. Delete Customer/Client definition from Glossary.
3	● Remove REQ-03
4	● Removed duplicate diagrams ● Modified UC-02b and UC-15 full descriptions ● Added UC-01, UC-10a, UC-11 full descriptions and sequence diagrams ● Edited diagrams: added loop to UC-2b, modified updatePrice signature in UC-16
5	● Added effort estimation calculation
6	● Added a description outlining how each concept relates to each use case ● Made a correction to the use case/concept matrix table
7	● Added new use cases
8	● Sorted classes alphabetically ● Updated class descriptions and diagram
9	● Edited all instances mentioning DB2 and IBM to read our database and website are hosted on Render ● Edited language alluding to multiple operators of multiple different garages to reflect a single company managing multiple garages ● Added implementation details to 9.6 based on feedback
10	● Modified algorithm diagram and updated diagram description
11	● Added UI designs for use cases for Demo 2
12	● Added tests for use cases for Demo 2
13	● Added deadline evolutions
14	● Imported only the references that made it into this report

1 - Customer Statement of Requirements

1.1 - Problem Diagnosis

In the USA, cars are a central part of life. They represent the freedom of the open road and the independence to travel anywhere at any time. However, wherever one drives, there must be adequate parking. American cities devote significant amounts of land to both on-and off-street parking. Hoenhe, et al. estimated in 2017 that the Phoenix, Arizona Metropolitan area offered 4.3 parking spaces per registered non-commercial vehicle. These 12.2 million parking spaces were the equivalent of 10% of the Phoenix metro's total land area at that time (Hoehne et al., 2019).

Despite all this land and all these dedicated parking spaces, finding parking in a city can still be a hassle. Garages operate on various access control systems, from older paper tickets to more modern automated systems. Pricing can be convoluted and opaque for customers, who may only discover the pricing scheme upon arrival. Finding a garage at all is only half the battle, as a driver may arrive only to find the lot already full. This inefficiency in finding parking leaves drivers wandering the city, wasting time and fuel and contributing to congestion. It also leaves garage owners with no way to advertise their available spaces and pricing to drivers in the area.

A few services have already arisen to tackle this inefficiency, however imperfectly. Parkopedia (Parkopedia) allows users to search for parking lots by location, viewing results on a map with projected pricing. However, lots do not report their availability to the site. A user is only aware of the potential to park, with no guarantee of an available parking space. Parking.com (SP Plus Corporation) likewise allows users to find garages and pricing on a map but adds the ability to reserve spaces for set periods. However, users are emailed a parking pass for their reservation, requiring them to manage their own credentials to park. Finally, SpotHero (SpotHero) builds on Parking.com's model to support garages with both manual and automatic parking control. With support for QR code-scanning automatic gates, gate attendants, and ticket-and-cashier type parking lots, SpotHero makes fewer demands on the lot owner for integration into the service. However, this increases the demands on users, who must read and understand the instructions for each different garage they use.

So far, none of the available solutions provide a truly unified parking solution for users and garage owners. Users cannot simply drive into an automated garage using their single parking website account. Garage owners cannot take advantage of the online platform’s payment system to handle all transactions at the garage. The online infrastructure for managing pricing and occupancy is separated from the onsite infrastructure, requiring operators to manage data on multiple platforms. Our project seeks to exploit this gap in the market.

1.2 - Proposed Solution

To offer value to customers and users, our solution proposes a network of garages operating our system. Users will benefit from a single service membership that allows them to find, reserve, and park in any member lot with automatic payments. Existing and prospective garage owners will benefit from a full software solution that reduces the need for on-site management of the lot. They will benefit from an expanded pool of registered customers directed to their lots, with a single system of payments. Users will have a streamlined parking experience with familiar hardware, automatic payments, and no parking passes to manage or passwords to remember.

Customers

For customers, a full software solution to garage control satisfies the needs of new projects and presents a straightforward upgrade solution for existing businesses. The software will be sold as a configurable solution based on the client's specific needs. In this case, the client has license plate readers at the entry to the garages and an automated car elevator to take drivers to their assigned floors. Each driver will be assigned a parking space upon entry, and the system will use space occupancy sensors to detect mistakes. Entry and exit events paired with occupancy sensors provide a simple, fault-tolerant way to track garage occupancy automatically. The pricing and number of available parking spaces are displayed on an electronic display outside the garage and mirrored online to inform and attract customers.

By allowing users to simply reserve *a* space in a chosen garage, instead of a specific space, we eliminate complex tracking of individual spot availability. Space assignments can be given at time of entry to the garage, allowing assignments to reflect the real-world state of the garage regardless of drivers who overstay or park in the wrong space. This also reduces demand on users, who are not required to be penalized for failing to find their assigned space, providing users with a lower-stress parking experience.

Out of the box, a garage management portal will be offered for viewing current occupancy, reservations, and historical data. It will allow operators to configure the specifications of their garages as well as add new garages to their network. Prices can also be set through this portal, giving operators a single system for managing both onsite and online garage displays. This simplification of management, combined with gate and reservation automation, requires less staff and reduced work hours for managing gates, counting spaces, generating reports, or handling reservations. Reduced labor needs and an optimized reservation system mean cost savings for garage owners while maximizing profit capabilities. Historical data and statistics provide a useful metric for our clients, allowing them to configure their reservation settings to optimize garage usage based on no-show and cancellation rates. Such data and statistics also allow clients to compare revenues and occupancy rates before and after the conversion to our system, making it easy to measure the return on investment.

The solution is scaled to a single system per client, each of which may own or operate many parking garages in many locations. The solution is flexible, allowing it to be deployed on-premises or in any cloud system. The customer may choose to buy only the software, requiring them to make all configuration changes to the software and deploy and manage upgrades themselves or through a third-party vendor. Alternatively, customers may be given an option to pay for our managed services. Our team would then continue to manage deployments, upgrades, and necessary configuration changes to match the design and structure of their garage network. This allows the client to pick a solution that matches their individual needs based on their IT capabilities and workload.

Users

For users, a single website provides streamlined access to every garage in the network. A user can reserve spaces and pay for parking, with an option to bind a license plate for automatic recognition. Users can use their device location or enter a destination address to see a map of available parking garages nearby. Each location will display it’s pricing, with an option for the user to make a reservation for an available lot. After making a reservation with their license plate number, users simply drive up to the garage gate, where the license plate reader recognizes their car, and park in any open space. As membership is required to park in the garage, and members are billed monthly, users exit the garage by simply driving to the exit gate and waiting for the license plate reader to recognize their vehicle.

The combination of easy online discovery of garages with automated garage systems saves users time and money. They do not have to drive around the city looking for open garages. They also do not have to worry about arriving at an open garage only to discover the lot charges exorbitant rates. On arrival, there is no token to hold, no QR code to scan, and no map or space number to memorize. Locate, reserve, drive in, drive out.

1.3 - Novelty

The garage system acknowledges the efforts of past teams to solve this problem, incorporating their most appealing innovations while reaching further for new features. The Fall 2018 project distinguished itself with a ‘smart’ pricing system, passcode-secured pedestrian entry, and an interface for ad-hoc users to select their own parking space on arrival. Our project incorporates their choice to reduce congestion within the garage by allowing only a single active vehicle per floor.

The Spring 2019 project introduced allowing users and administrators to search and filter data on vehicles parked in the lot. It allowed first-time users to park without an account and sign up online afterward. We are most interested in their decision to restrict drivers to only registering for an area of the garage, not a specific space. Our project expands on this idea by assigning parking spaces on arrival as available, with users only reserving the right to park somewhere in the chosen lot. This reduces the amount of input necessary to reserve a parking space and allows for tolerance of misparked users.

The Fall 2019 project took a decidedly high-tech approach to garage management. A blockchain-based payment system brought data encryption and decentralization. Docker-based containerization allowed for portability and real-time data exchange between garages, while Electronic Toll Collection (ETC) tags handled payment. A license plate blacklist alerted managers to users who should be turned away or turned in to law enforcement. However, we see limited utility to the additional layers of blockchain and Docker data containerization compared to traditional solutions. ETC tags also add hardware and software complexity to the system for limited utility, as the system already identifies users by license plate and charges them automatically.

This project innovates by zooming out and setting the focus on the wider world of city parking. We help owners of multiple garages to create a network of locations, managed through a single portal. Customers benefit from simplified management of all their locations, and automatic data collection and statistics for each. Users benefit from a single account, simple reservations, and automatic payments for the entire network of garages managed by the customer. The user experience is simplified by allowing users to make reservations by garage, with spaces assigned only on arrival. The choice to remove space selection at reservation time reduces user input and demands on attention, further reduced by introducing a fault-tolerant system that tracks the actual space the user parked in, rather than assuming that users will never make mistakes.

2 - Glossary of Terms

Confirmed Reservation: Represent registered customers who make reservations as the need arises.

Guaranteed Reservation: Represent registered customers who made a monthly *contract* with the parking garage for a parking spot, such as those going to work who need to park on a daily basis during a predetermined period.

Overstays: Currently parked customers who wish to extend their stay beyond the time of their original reservation.

Understays: Currently parked customers who arrive on time but decide to leave before their predicted time of departure.

Walk-ins: Customers who arrive at the parking garage without a contract of reservation.

No-show: Customers who have a reservation but do not arrive for that reservation within the grace period of the reservation start time.

Occupancy State: Refers to the state of parking spots which can be either “available”, “reserved”, or “occupied”.

License Plate Reader: A camera device that recognizes the characters on vehicle license plates and uses it to confirm an existing reservation.

Grace Period: A predetermined amount of time for which a reservation will be held after the beginning of a reservation period. If the customer does not arrive within this given window, the occupancy state is converted from “reserved” to “available”. The grace period is applied to the end of a walk-in user to avoid “overstay” fees.

Overbooking: Accepting more reservations for spots than are available by forecasting the number of no-show reservations, overstays, understays, and walk-ins.

Spot Occupancy Sensor: A sensor that detects whether a parking spot is available or currently occupied.

Buffer time: The time interval between reservations. Allows the current user time to leave the space before filling it with another user, and helps to account for overstays.

User Interface/Experience (UI/UX): What the user sees and interacts with when accessing and navigating the application/website.

User: A registered customer.

Garage Operator: Persons who have operator/administrative permissions to set prices, add, remove, and configure garages.

Rain Check: a discounted or free reservation at a later date

3 - System Requirements

REQ	PW	Description
User Interaction		
REQ-01	10	User Registration - A user can register for an account
REQ-02	9	User Registration - a user can reset their password
REQ-03	5	User Registration - A Garage Operator account can upgrade a user account to a garage operator
REQ-04	10	User Registration - A user must register for an account before parking in the garage
REQ-05	10	Parking Space Requests - A user can search for a parking space by location and/or date
REQ-06	10	Parking Space Requests - A user can reserve a spot in a specific garage in the customer's network
REQ-07	7	Parking Space Requests - A user can make multiple guaranteed reservations for a permanent spot
REQ-08	8	Parking Space Requests - Users can update their guaranteed reservations
REQ-09	6	Parking Space Requests - A user can choose to pay a fee for

		a larger “grace window”
REQ-10	7	Parking Space Requests - A user may miss their reservation and pay full price
REQ-11	7	Parking Space Requests - A user may show up late to their reservation and pay full price
REQ-12	7	Parking Space Requests - A user may leave their reservation early and pay full price
REQ-13	5	Parking Space Requests - A user can extend their reservation up to a half hour early, if there is availability
REQ-14	5	Parking Space Requests - A walk-in user can extend the duration of their stay
REQ-15	7	Parking Space Requests - A user can overstay their reservation and will pay a fee
REQ-16	8	Parking Space Requests - A user can have up to three reservations at a time
REQ-17	5	Parking Space Requests - A user can merge reservations that are close together
REQ-18	8	Parking Space Requests - A user may book a reservation in a full garage
REQ-19	9	Parking Space Requests - A user can modify a reservation prior to start
REQ-20	9	Parking Space Requests - A user can cancel a reservation prior to start
REQ-21	8	Parking Space Requests - A user can view all reservations on one screen
REQ-22	5	Parking Space Requests - Users will be billed monthly
REQ-23	10	Account Management - Users can add or remove Vehicles from their account
REQ-24	10	Account Management - Users can change their automatic payment method from their account
REQ-25	10	Account Management - Users can update their contact information

Garage Access Control		
REQ-26	10	Arrivals - A user must be able to trigger the elevator to access a reserved spot
REQ-27	9	Arrivals - A user’s license plate can be scanned and used to find reservation or account
REQ-29	10	Arrivals - A user will receive a spot number to park in via display board after confirming their reservation
REQ-30	10	Arrivals - A user can park on the first floor without a reservation
REQ-31	8	Arrivals - A user must provide their member number and estimated duration on arrival as a walk-in
REQ-32	9	Arrivals - A walk-in user will receive a spot number once the estimated duration is provided
REQ-33	4	Arrivals - A walk-in user will pay a fee if they do not park in the given spot
REQ-34	9	Arrivals - A user can search for a reservation by reservation number, or member number if the reservation is not automatically found by the elevator
REQ-35	7	Arrivals - A user will receive a notification if they enter an invalid reservation or member number to the console
REQ-36	8	Arrivals - The user may not park without a license plate
REQ-37	8	Arrivals - A user may show up within the grace period of their reservation with no penalties
REQ-38	7	Arrivals - A user can show up after their grace period and receive an available “reservation” spot
REQ-39	4	Arrivals - A user will receive a fee if they park in the incorrect spot for their reservation
REQ-40	6	Arrivals - A user will be asked to leave if they do not have a valid reservation, the garage is full, or they do not have a license plate
REQ-41	3	Arrivals - If a user does not leave when asked, security will be notified
REQ-42	7	Arrivals - If a user arrives to a full garage, with a reservation,

		due to overbooking, the user will be asked to leave via a display board and be given a rain check
REQ-43	3	Arrivals - If there is current availability to accommodate the reservation at another garage, the user will be provided with options between the rain check or alternate garages
REQ-44	10	Departures - A user must be able to trigger the exit
REQ-45	10	Departures - A user’s reservation will end on departure
REQ-46	5	Departures - A walk-in user can leave early and will NOT be charged for their entire expected duration
Monitoring Occupancy and Space Assignment		
REQ-47	10	Monitoring Occupancy - Current Occupancy will be counted via sensors on parking spots
REQ-48	7	Monitoring Occupancy - Occupancy will be displayed on entrance to the garage and by garage for users on the website
REQ-49	9	Monitoring Occupancy - Expected Reservation occupancy will be considered for future dates based off the current count of reservations for that date range
REQ-50	10	Space Assignment - A user will be assigned a space upon arrival
REQ-51	10	Space Assignment - Assigned Spaces will be displayed on the display board in the elevator
REQ-52	10	Space Assignment - A user with a reservation will be assigned a space on the upper floors
REQ-53	6	Space Assignment - A walk-in user will be assigned a space on the bottom floor
Simulation of Arrivals and Departures		
REQ-54	9	An Admin must be able to enter the license plate information that would be read by the elevator camera
REQ-55	9	An Admin must be able to enter the Member Number or Reservation Number that the user would enter on the console
REQ-56	9	An Admin Must be able to update the occupancy of the parking spots as read by the sensors

REQ-57	9	An Admin must be able to enter the license plate information that would be read by the exit ramp camera
REQ-58	3	An Admin will be able to view the results of the Automatic Simulation
Statistical Data Collection		
REQ-59	3	A garage operator will be able to view a dashboard for garage statistics
REQ-60	5	A garage operator will be able to view understays, overstays, overbook rating, and occupancy rating
REQ-61	2	A garage operator will be able to view data by week, by month and by year
System Administration		
REQ-62	10	A garage operator must be able to add garages to their network
REQ-63	9	A garage operator must be able to configure the number of floors and spots per floor for each garage
REQ-64	6	A garage operator must be able to deactivate a garage
REQ-65	7	A garage operator must be able to set prices for reservations, walk-ins and guaranteed reservations for each garage
REQ-66	4	A garage operator must be able to set and update the overbooking percentage for each garage
REQ-67	4	A garage operator must be able to configure fees

4 - Functional Requirements

4.1 - Stakeholders

Stakeholders include all persons who stand to gain and profit from implementing or converting their garage to support this system of management, as well as persons who enroll/sign-up to use this garage service.

1. Garage Owners: Adopt this system in pursuit of diversification in order to pursue and grow their financial interests.
2. Managers: Increase in operational efficiency and data collecting to more effectively manage operations and consume statistics.
3. Investors: Those who choose to invest in adopting this garaging system to earn on future financial gains.
4. Users: Consumers who register to use this garaging system to more efficiently locate and schedule guaranteed parking quickly with an easy-to-use interface.

4.2 - Actors and Goals

Actor	**Actor's Goals**	**Use Case(s)**
Owner/Manager	Update pricing	UC-16
Owner/Manager	View current parking status (occupancy/vacancies)	UC-14
Owner/Manager	View garage stats	UC-14
Owner/Manager	Add a garage to network of garages	UC-15
User	View garages within a given distance	UC-02
User	Reserve a space	UC-02
User	Modify reservation	UC-03, UC-07
User	Park as a walk-in customer	UC-10b
User	Exit the garage	UC-05, UC-06
User	Retrieve reservation with reservation number	UC-02, UC-03

User	Add payment information	UC-02, UC-03, UC-08
User	Receive reservation confirmation	UC-02, UC-03
User	Enter the garage	UC-04, UC-03
Database	Maintain user registry	UC-01
Database	Maintain space availability	UC-13
Database	Maintain statistical information	UC-14
Camera	Read license plate information to confirm reservation status	UC-10
Parking Sensors	Mark occupied parking spaces	UC-12, UC-13
Digital Display	Allow walk-in customers to make an on the spot reservation	UC-10,UC-13

4.3 - Use Cases

4.3.1 - Casual Descriptions

An asterisk (*) prepended to a use case number denotes a use case that may be considered for future work.

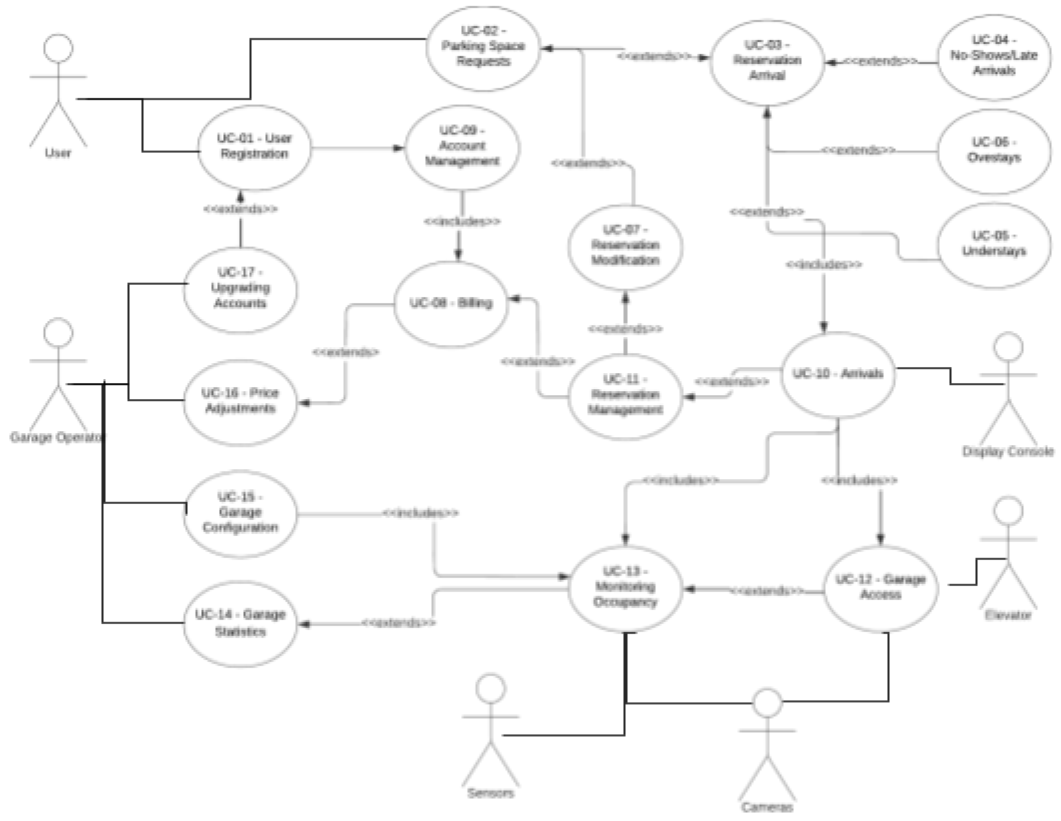
UC	REQs	Description
User Interaction		
UC-01	REQ-01, REQ-04, REQ-02, REQ-88	User Registration - A user can register for an account with their email, phone number and their password. On login, a user will use this information to access their account or reset password,
UC-02a	REQ-05, REQ-06, REQ-07, REQ-16, REQ-87, REQ-91, REQ-49	Parking Space Requests (Single) - A user can search for a parking space by location and/or date and receive a list of available spots based on their criteria. A user can request a spot and make up to 3 non-overlapping reservations if they have a card on file.

UC-02b	REQ-07, REQ-08, REQ-91	Parking Space Request (Permanent/Guaranteed) - A user can make multiple guaranteed reservations for a permanent spot. A user can select the frequency of the reservations and the timeframe these reservations are needed.
*UC-03	REQ-09, REQ-37, REQ-38	Reservation Arrival - A user with a reservation can arrive within the Grace Period with no charge, choose to pay a fee for a larger Grace Period, or show up outside of the Grace Period and only receive a spot if the garage is not full and still pay full price
*UC-04	REQ-10, REQ-11	No-Shows/Late Arrivals - A user may miss their reservation or show up late and still pay full price. If the user shows up late, they only may receive a spot if availability allows.
*UC-05	REQ-12, REQ-46	Understays - A user may leave their reservation early and pay full price. Walk-in’s do not pay full price if they leave early
*UC-06	REQ-13, REQ-14, REQ-15	Overstays - A both walk-in and reservation users can extend their reservation up to a half hour early, if there is availability. If they do not extend their reservation, a fee will apply. Walk-ins have the Grace period applied to the end of their stay.
*UC-07	REQ-17, REQ-19, REQ-20, REQ-89	Reservation Modification - A user can modify or cancel a reservation prior to start. If the reservation is within an hour of the previous reservation they can be merged.
*UC-08	REQ-22, REQ-21, REQ-89	Billing - Users will be billed monthly for all reservations in the month. They will receive a monthly statement via email and can view all historical reservations and their price.
*UC-09	REQ-23, REQ-25, REQ-24	Account Management - Users can add or remove Vehicles from their account, update contact information and add/update payment methods
Garage Access Control		
UC-10a	REQ-26, REQ-27, REQ-29, REQ-34,	Reservation Arrivals - A user’s reservation can be found via associated license plate or reservation number. If the entered reservation or member number is invalid, the user will be notified. The user will see a spot

	REQ-50, REQ-51, REQ-52 REQ-35	number on the display board and be taken to the proper floor.
*UC-10b	REQ-30, REQ-31, REQ-32, REQ-50, REQ-51, REQ-92, REQ-35	Walk-in Arrivals - A user can park on the first floor without a reservation by providing a member number and an estimated duration, then receive a spot number on the first floor. If the entered member number is invalid, the user will be notified.
UC-11	REQ-44	Reservation Management - A reservation will be marked complete when the user triggers the exit and leaves the garage.
*UC-12a	REQ-40, REQ-43, REQ-33	Garage Access - A user will not be allowed to park if the walk-in garage is full and they do not have a reservation. If the garage is full and the user does have a reservation, they will be asked to go to a different garage or receive a rain check discount.
*UC-12b	REQ-41, REQ-36, REQ-42	Request to Leave - If a spot is not available or a user does not have a license plate, they will be asked to leave. If the user does not leave, security will be notified.
Monitoring Occupancy and Space Assignment		
*UC-13	REQ-47, REQ-48, REQ-39	Monitoring Occupancy - Current Occupancy will be counted via sensors on parking spots and displayed on the entrance to the garage and on the website. A user will pay a fee if they do not park in the given spot
Statistical Data Collection		
*UC-14	REQ-59, REQ-60, REQ-61	Garage Statistics - A garage operator will be able to view a dashboard for garage statistics with understay, overstay, overbooking rate, occupancy rating, and other statistics
System Administration		
UC-15	REQ-62, REQ-63, REQ-64,	Garage Configuration - A garage operator must be able to add garages to their network and define the number of spots on each floor, activate/deactivate the garage,

	REQ-66, REQ-18, REQ-90	and set the overbooking rate.
UC-16	REQ-65, REQ-67	Price Adjustments - A garage operator must be able to set prices for reservations, walk-ins, guaranteed reservations for each garage and fees
*UC-17	REQ-03	Upgrading Accounts - A garage operator can upgrade a user account to a garage operator

4.3.2 - Use Case Diagram


User
UC-02 - Parking Space Requests
<<extends>>
UC-03 - Reservation Arrival
<<extends>>
UC-04 - No-Shows/Late Arrivals
UC-01 - User Registration
UC-09 - Account Management
<<extends>>
UC-06 - Overstays
<<extends>>
<<includes>>
<<extends>>
UC-07 - Reservation Modification
UC-05 - Understays
<<includes>>
UC-17 - Upgrading Accounts
UC-08 - Billing
<<extends>>
Garage Operator
<<extends>>
<<extends>>
UC-11 - Reservation Management
<<extends>>
UC-10 - Arrivals
UC-16 - Price Adjustments
Display Console
<<includes>>
<<includes>>
UC-15 - Garage Configuration
<<includes>>
UC-13 - Monitoring Occupancy
<<extends>>
UC-12 - Garage Access
Elevator
UC-14 - Garage Statistics
<<extends>>
Sensors
Cameras

4.3.3 - Traceability Matrix

	UC-	UC-	UC-	UC-	UC-	UC-	UC-	UC-	UC-	UC-	UC-	UC-	UC-	UC-	UC-	UC-	UC-	UC-	UC-	UC-

	01	02a	02b	03	04	05	06	07	08	09	10a	10b	11	12a	12b	13	14	15	16	17
REQ-01	10																			
REQ-02	9																			
REQ-03																				5
REQ-04	10																			
REQ-05		10																		
REQ-06		10																		
REQ-07		7	7																	
REQ-08			8																	
REQ-09				6																
REQ-10					7															
REQ-11					7															
REQ-12						7														
REQ-13							5													
REQ-14							5													
REQ-15							7													
REQ-16		8																		
REQ-17								5												
REQ-18																		8		
REQ-19								9												
REQ-20								8												
REQ-21									8											
REQ-22									5											
REQ-23										10										
REQ-24										10										
REQ-25										10										
REQ-26											10									
REQ-27											9									

REQ-29											10									
REQ-30												10								
REQ-31												8								
REQ-32												9								
REQ-33														4						
REQ-34											9									
REQ-35											7	7								
REQ-36															8					
REQ-37				8																
REQ-38				7																
REQ-39																4				
REQ-40														6						
REQ-41															3					
REQ-42															7					
REQ-43														3						
REQ-44													10							
REQ-45													10							
REQ-46						5														
REQ-47																10				
REQ-48																7				
REQ-49		9																		
REQ-50											10	10								
REQ-51											10	10								
REQ-52											10									
REQ-53												6								
REQ-59																	3			
REQ-60																	5			
REQ-61																	2			

REQ-62																		10		
REQ-63																		9		
REQ-64																		6		
REQ-65																			7	
REQ-66																		4		
REQ-67																			4	
REQ-87		9																		
REQ-88	6																			
REQ-89								6	6											
REQ-90								4	4									4		
REQ-91		9	9																	
REQ-92												9								
REQ-93										9										
TOTAL	35	62	24	21	14	12	17	32	23	39	75	69	20	13	18	21	10	41	11	5

4.3.4 - Fully-Dressed Descriptions

Use Case UC-01:	User Registration
Related Requirements:	REQ-01, REQ-02, REQ-04, REQ-88
Initiating Actor:	User
Actor's Goal:	Create a new non-operator user account in the system
Participating Actors:	Database
Preconditions:	The username is unique
Postconditions:	A new user account is created in the system
Failed End Condition:	

Flow of Events for Main Success Scenario

1. User enters their name, desired username, password, and email address, then

clicks ‘Start reserving now.’
2. System checks if new username is unique. If not, user is requested to try again.
3. System creates the new user account, and the database is updated.
4. System confirms the account was created.

Use Case UC-02a:	Requesting a Single Parking Spot
Related Requirements:	REQ-05, REQ-06, REQ-07, REQ-16, REQ-87, REQ-91, REQ-49
Initiating Actor:	User
Actor’s Goal:	Find and Request a one-time parking spot
Participating Actors:	Database
Preconditions:	The user is signed in, has a payment method associated to the account, and there are available spots
Postconditions:	A reservation is made
Failed End Condition:	
Flow of Events for Main Success Scenario	
1. User enters search conditions (location, distance, datetime range) 2. User selects “Single” spot 3. Database populates a list of garages with availability matching those conditions 4. User selects a reservation option 5. (Optional) User selects vehicle to be associated to the reservation 6. User saves reservation information and receives reservation number 7. Database is updated with new reservation	

Use Case UC-02b:	Requesting a Permanent/Guaranteed Spot
Related Requirements:	REQ-07, REQ-08, REQ-91
Initiating Actor:	User
Actor’s Goal:	Request a recurring guaranteed spot

Participating Actors:	Database
Preconditions:	The user is signed in, has a payment method associated to the account, and there are available spots for their start date
Postconditions:	The user creates a recurring reservation
Failed End Condition:	
Flow of Events for Main Success Scenario 1. User enters search conditions (location, distance, start date) 2. User selects “Monthly” spot 3. Database populates a list of garages with availability matching those conditions 4. User selects a reservation option 5. (Optional) User selects vehicle to be associated to the reservation 6. User saves reservation information and receives reservation number 7. Database is updated with new reservation	

Use Case UC-10a:	Reservation Arrivals
Related Requirements:	REQ-26, REQ-27, REQ-29, REQ-34, REQ-35, REQ-50, REQ-51, REQ-52
Initiating Actor:	User
Actor’s Goal:	Park in the garage using their existing reservation
Participating Actors:	Database, Camera, Digital Display
Preconditions:	The user has already made a reservation valid for the garage and current time.
Postconditions:	The reservation status is changed to ‘parked’
Failed End Condition:	
Flow of Events for Main Success Scenario	
	1. User drives up to the entry gate of the garage. 2. License plate camera detects the vehicle, scans the license plate, and informs the system of the vehicle attempting to enter. 3. The system searches reservations for the garage matching that vehicle and time. 4. The system informs the user of their parking space number,

allows the user to enter the garage, and updates their reservation status.

Flow of Events for Extensions (Alternate Scenario)

1. User drives up to the entry gate of the garage.
2. License plate camera detects the vehicle, scans the license plate, and informs the system of the vehicle attempting to enter.
3. The system finds no matching reservations for the vehicle, or no readable license plate, and asks the user for their reservation number.
4. The user enters their reservation number, and the system searches for a match.
5. The system informs the user of their parking space number, allows the user to enter the garage, and updates their reservation status.

Use Case UC-11:	Reservation Management
Related Requirements:	REQ-44
Initiating Actor:	User
Actor's Goal:	Leave the garage and end their reservation
Participating Actors:	Database, Camera, Digital Display
Preconditions:	The reservation status is ‘parked’. The reservation is for the specified garage.
Postconditions:	The reservation status is changed to ‘complete’ (single) or ‘valid’ (monthly)
Failed End Condition:	

Flow of Events for Main Success Scenario

1. User drives up to the entry gate of the garage.
2. License plate camera detects the vehicle, scans the license plate, and informs the system of the vehicle attempting to exit.
3. The system searches reservations for the garage matching that vehicle and time that are already marked as ‘parked’.
4. The system allows the user to exit, and updates their reservation status.

Flow of Events for Extensions (Alternate Scenario)

1. User drives up to the entry gate of the garage.
2. License plate camera detects the vehicle, scans the license plate, and informs the system of the vehicle attempting to exit.
3. The system finds no matching reservations for the vehicle, or no readable license plate, and asks the user for their reservation number.
4. The system searches reservations for the garage matching that reservation number that are already marked as ‘parked’.
5. The system allows the user to exit, and updates their reservation status.

Use Case UC-15:	Adding a Garage to a Customer’s Network
Related Requirements:	REQ-62, REQ-63, REQ-64, REQ-66, REQ-18, REQ-90
Initiating Actor:	Garage Operator
Actor’s Goal:	Add/Update a garage’s configuration
Participating Actors:	Database
Preconditions:	Garage Operator is logged in
Postconditions:	Database is updated with new garage information
Failed End Condition:	

Flow of Events for Main Success Scenario

1. Garage Operator navigates to garage management page
2. Garage Operator Selects Add or Modify Garage
    a. Add:
        i. Name is REQUIRED
        ii. Location is REQUIRED
        iii. Number of Floors is REQUIRED
        iv. Garage operator enters number of spots for each floor
        v. Overbooking rate is added
        vi. Active/Inactive toggle is switched to activate
    b. Modify:
        i. Number of Floors can be modified

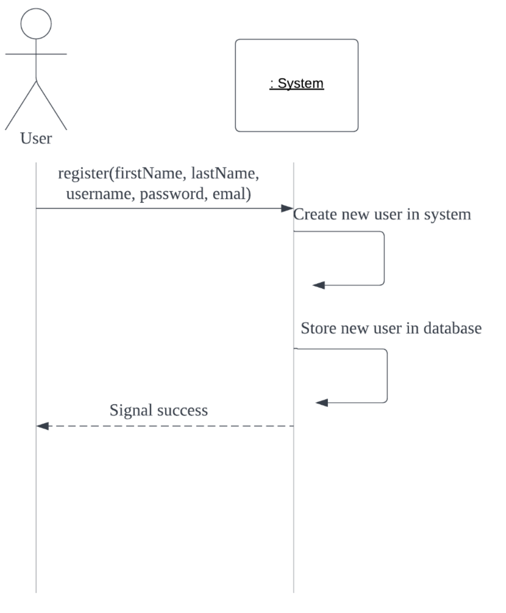
ii. Overbooking rate may be modified
iii. Active/Inactive toggle may be modified

3. Record is saved, database is updated

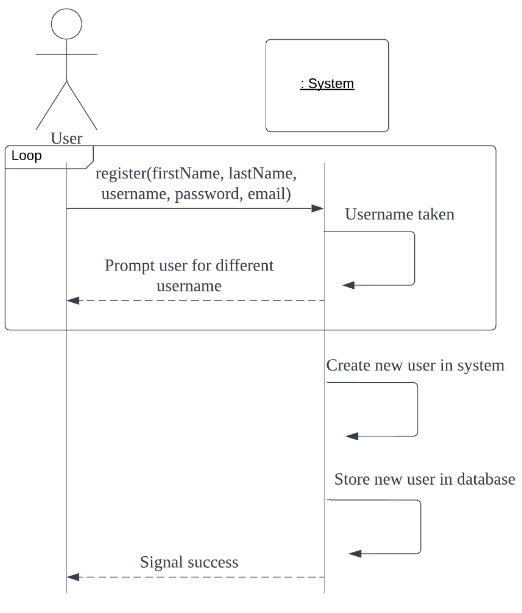
Use Case UC-16:	Adjusting Prices for a Garage
Related Requirements:	REQ-65, REQ-67
Initiating Actor:	Garage Operator
Actor's Goal:	Configure Prices and Fees
Participating Actors:	Database
Preconditions:	Garage operator is signed in
Postconditions:	Database is updated with new pricing
Failed End Condition:	
Flow of Events for Main Success Scenario	
	1. Garage Operator navigates to pricing page 2. Garage Operator Selects pricing type to modify 3. Updates Pricing 4. Saves Record and Database updates

4.4 - System Sequence Diagrams

UC-01 - User Registration

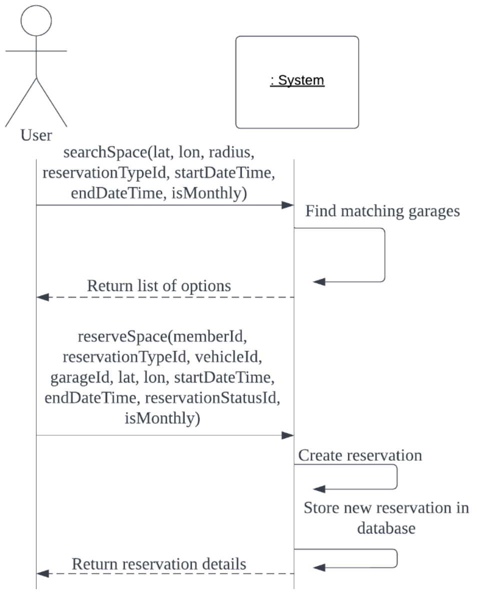

User
: System
register(firstName, lastName, username, password, emal)
Create new user in system
Store new user in database
Signal success

(a)
Main success case


User
: System
Loop
register(firstName, lastName, username, password, email)
Username taken
Prompt user for different username
Create new user in system
Store new user in database
Signal success

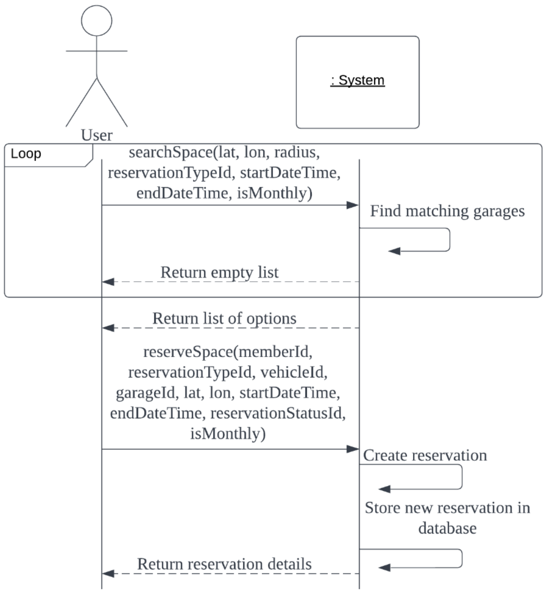
(b)
Username already in use

UC-02a - Requesting a Single Parking Spot



User
: System
searchSpace(lat, lon, radius, reservationTypeId, startDateTime, endDateTime, isMonthly)
Find matching garages
Return list of options
reserveSpace(memberId, reservationTypeId, vehicleId, garageId, lat, lon, startDateTime, endDateTime, reservationStatusId, isMonthly)
Create reservation
Store new reservation in database
Return reservation details

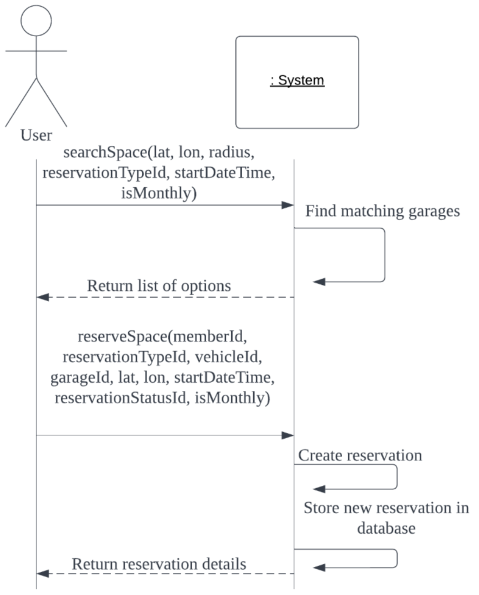
(a)
Main success case



User
: System
Loop
searchSpace(lat, lon, radius, reservationTypeId, startDateTime, endDateTime, isMonthly)
Find matching garages
Return empty list
Return list of options
reserveSpace(memberId, reservationTypeId, vehicleId, garageId, lat, lon, startDateTime, endDateTime, reservationStatusId, isMonthly)
Create reservation
Store new reservation in database
Return reservation details

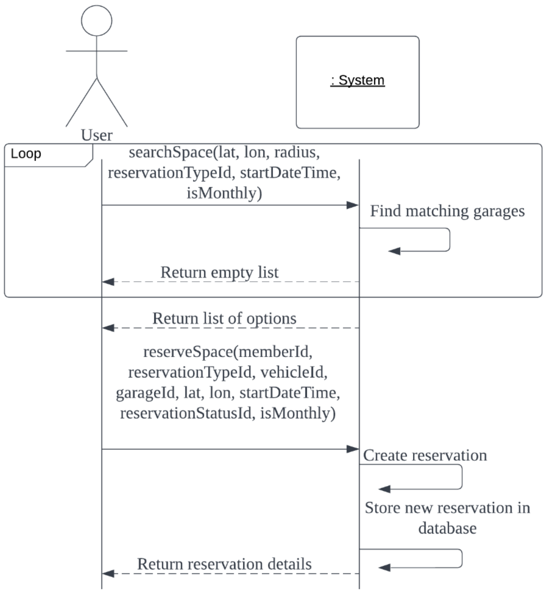
(b)
No available spaces for the giwven conditions

UC-02b - Requesting a Permanent/Guaranteed Spot



User
: System
searchSpace(lat, lon, radius, reservationTypeId, startDateTime, isMonthly)
Find matching garages
Return list of options
reserveSpace(memberId, reservationTypeId, vehicleId, garageId, lat, lon, startDateTime, reservationStatusId, isMonthly)
Create reservation
Store new reservation in database
Return reservation details

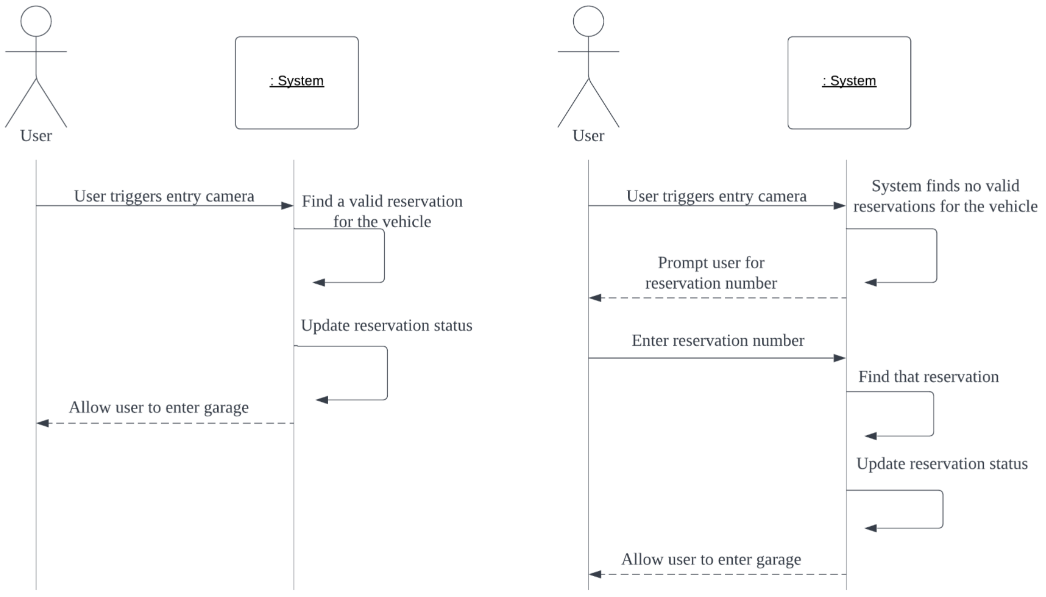
(a)
Main success case



User
: System
Loop
searchSpace(lat, lon, radius, reservationTypeId, startDateTime, isMonthly)
Find matching garages
Return empty list
Return list of options
reserveSpace(memberId, reservationTypeId, vehicleId, garageId, lat, lon, startDateTime, reservationStatusId, isMonthly)
Create reservation
Store new reservation in database
Return reservation details

(b)
No available spaces for the giwven conditions

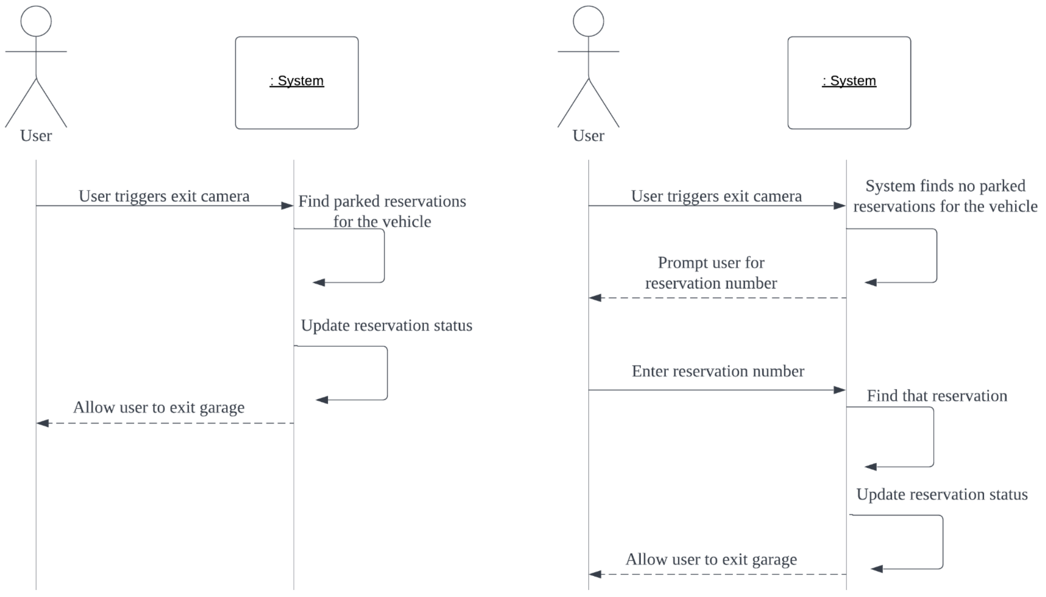
UC-10a - Reservation Arrivals


User
: System
User triggers entry camera
Find a valid reservation for the vehicle
Update reservation status
Allow user to enter garage
User
: System
User triggers entry camera
System finds no valid reservations for the vehicle
Prompt user for reservation number
Enter reservation number
Find that reservation
Update reservation status
Allow user to enter garage

(a)
Entering the Garage

(b)
Entering by reservation number

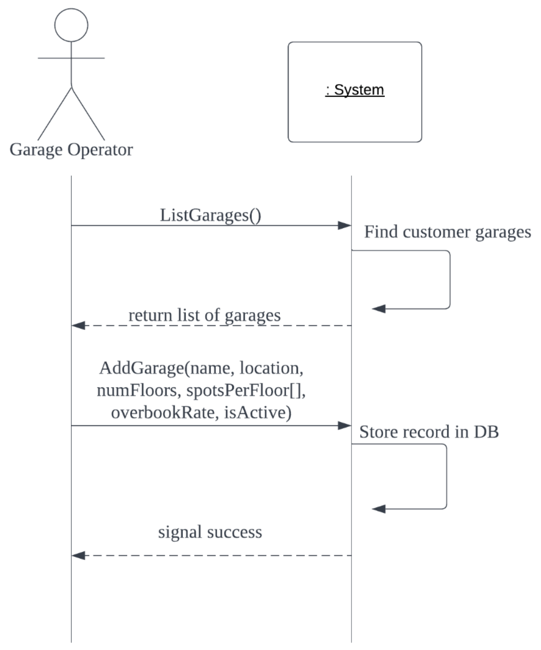
UC-11 - Reservation Management


User
: System
User triggers exit camera
Find parked reservations for the vehicle
Update reservation status
Allow user to exit garage
User
: System
User triggers exit camera
System finds no parked reservations for the vehicle
Prompt user for reservation number
Enter reservation number
Find that reservation
Update reservation status
Allow user to exit garage

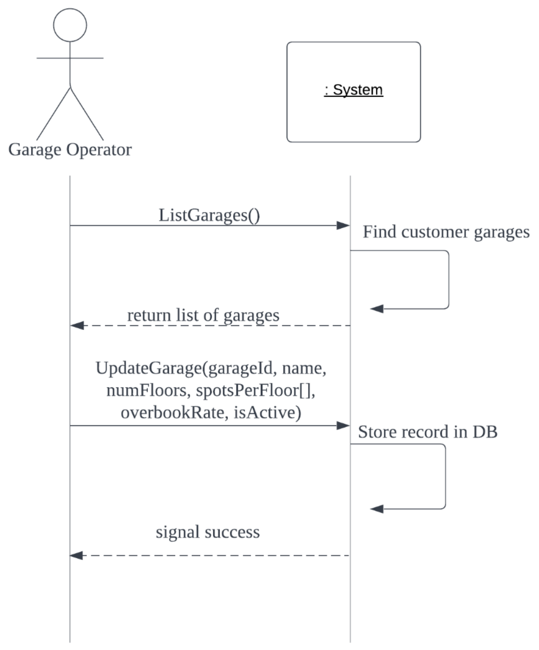
(a)
Exiting the Garage

(b)
Exiting by reservation number

UC-15 - Adding a Garage to a Customer’s Network

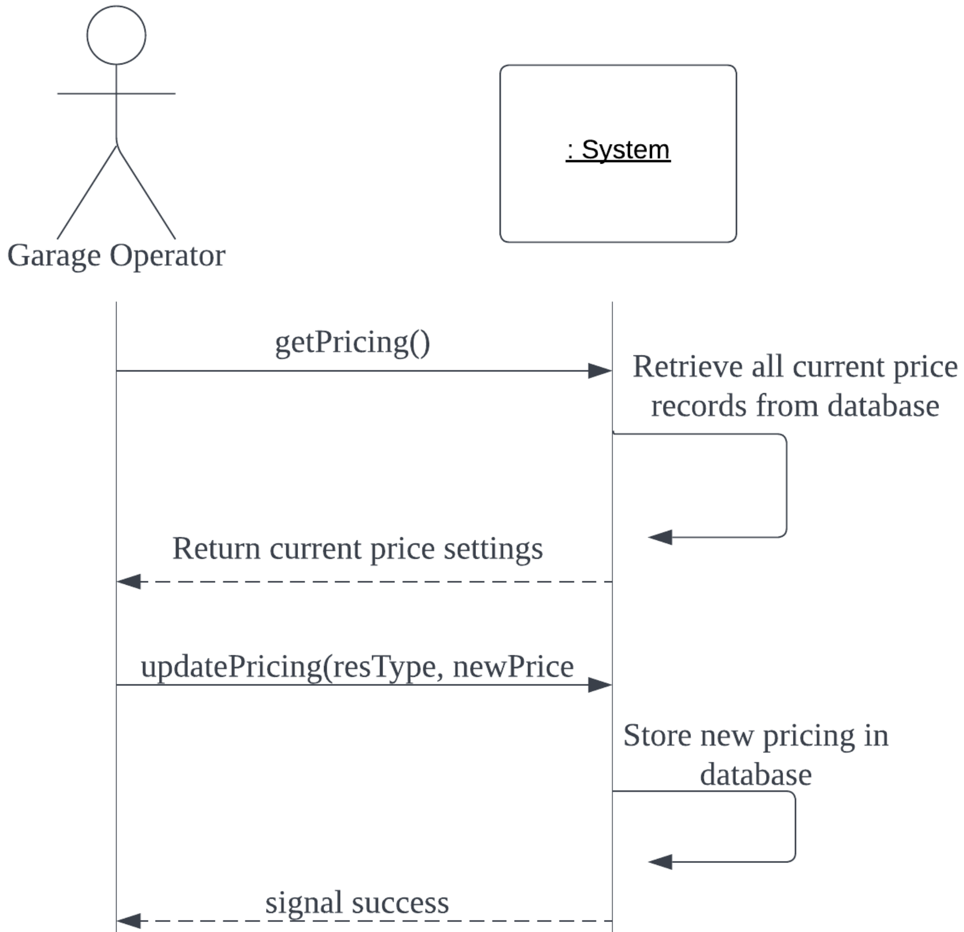


(a)
Adding a new garage



(b)
Updating an existing garge

UC-16 - Adjusting Prices for a Garage


Garage Operator
: System
getPricing()
Retrieve all current price records from database
Return current price settings
updatePricing(resType, newPrice
Store new pricing in database
signal success

(a)
Main success case

5 - Effort Estimation

5.1 - Unadjusted Actor Weight (UAW)

Actor	Description	Complexity	Weight
Garage Operator	Interacts with the system by managing pricing and garage configuration and has the capability to view garage statistics dashboard	Average	2
User	Interacts with the system by entering user information, searching for spots, making reservations, receiving and managing autopayments, triggers garage system, enters information on display console and can view garage availability	Complex	3
Database	Holds and manages Data entered into the application	Simple	1
Camera	Reads license plate and searches for memberships/reservations to then find appropriate parking spot	Average	2
Parking Sensors	Used to detect current occupancy and determine if a user parks in the correct spot	Average	2

Display Console	Used to display and receive client information. Such as reading member numbers, reservation numbers, displaying spot for user to park, notifications on current availability, suggesting other garages if no availability	Complex	3
Total	1 x Simple (1) + 3 x Average (2) + 2 x Complex (3) = 13		

5.2 - Unadjusted Use Case Weight (UUCW)

UC	Description	Category	Weight
UC-01	User Registration	Average	10
UC-02a	Parking Space Requests (Single)	Complex	15
UC-02b	Parking Space Request (Permanent/Guaranteed)	Complex	15
UC-03	Reservation Arrival (Grace Period)	Simple	5
UC-04	No-Shows/Late Arrivals	Average	10
UC-05	Understays	Average	10
UC-06	Overstays	Average	10
UC-07	Reservation Modification	Average	10
UC-08	Billing	Simple	5
UC-09	Account Management	Simple	5
	Garage Access Control		
UC-10a	Reservation Arrivals (Garage)	Complex	15

UC-10b	Walk-in Arrivals	Complex	15
UC-11	Reservation Management	Average	10
UC-12a	Garage Access	Complex	15
UC-12b	Request to Leave	Average	10
Monitoring Occupancy and Space Assignment			
UC-13	Monitoring Occupancy	Average	10
Statistical Data Collection			
UC-14	Garage Statistics	Simple	5
System Administration			
UC-15	Garage Configuration	Simple	5
UC-16	Price Adjustments	Average	10
UC-17	Upgrading Accounts	Simple	5
Total	6 x Simple (5) + 9 x Average (10) + 5 x Complex (15) = 195		

5.3 - Unadjusted Use Case Points (UUCP)

UUCP = UAW + UUCW

UUCP = UAW (13) + *UUCW* (195) = 208

5.4 - Technical Complexity Factors (TCF)

TF	Description	Weight	Perceived Complexity	Calculated Factor
T1	Distributed System	2	3	2x3=6

T2	Performance Objectives	1	3	1x3=3
T3	End-User Efficiency	1	3	1x3=3
T4	Complex Internal Processing	1	4	1x4=4
T5	Reusable Design or Code	1	1	1x1=1
T6	Easy to Install	0.5	2	0.5x2=1
T7	Easy to Use	0.5	4	0.5x4=2
T8	Portable	2	3	2x3=6
T9	Easy to Change	1	3	1x3=3
T10	Concurrent to Use	1	1	1x1=1
T11	Special Security Features	1	3	1x3=3
T12	Provides Direct Access for Third Parties	1	1	1x1=1
T13	Special User Training Facilities are Required	1	0	1x0=0
			Total:	34
TCF= 0.6 + (0.01 x 34) = 0.94				

5.5 - Use Case Points (UCP)

UCP = UUCP x TCF x ECF

UCP= 208 x 0.94 x 1 = 195.52

5.6 - Project Duration

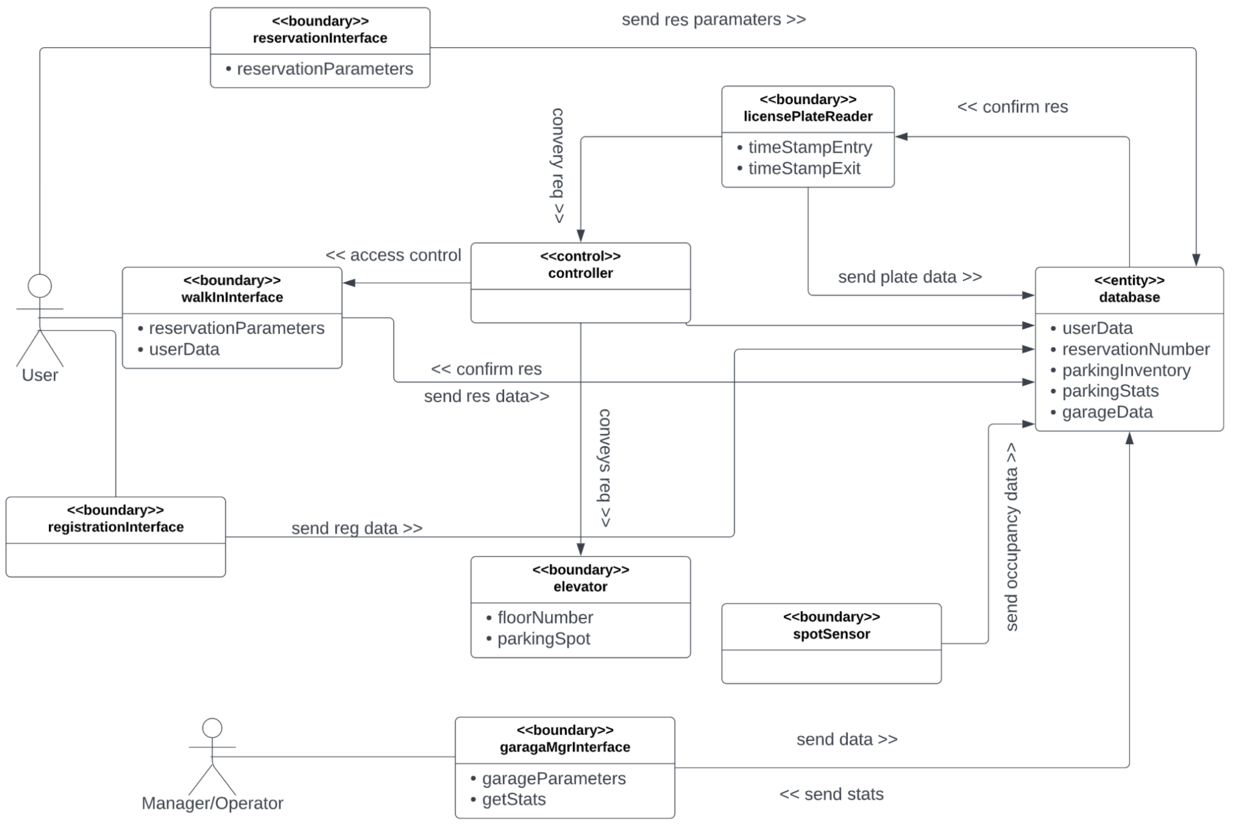
Given a Productivity Factor (PF) of 28 hours per Use Case Point (UCP):

Duration = UCP x PF = 195.52 x 28 = 5,474.56 hours

Assuming three student developers working part-time (15 hours/week), the projected project duration is approximately 122 weeks.

6 - Domain Analysis

Conceptual Model



6.1 - Model Derivation

The conceptual model is derived in keeping with the defined use cases and requirements. Use cases are handled via five concepts. The website allows users to create an account and register. Once registered they can add and edit vehicles and payment methods. From the website, they can search for and make various types of reservations in the selected garage and, if necessary, modify those reservations using the reservation interface. The walk-in display allows users to make an on-the-spot reservation using the display and entering their user ID number. The garage access concept controls entry and exit using the license plate reader. The garageManagement concept handles UC associated with adding garages, receiving stats, pricing, and user account permissions. The database concept stores all necessary information required to associate transactions with the corresponding user and garage configurations to maintain parking inventory.

A new user must register. Once registration is complete, they have the option of either making a reservation in advance or parking as a walk-in. Customers with existing reservations proceed to the elevator on arrival, where the camera system recognizes the license plate and sends that information to the database. The reservation confirmation is returned to allow access to the garage. The controller triggers the elevator for access when the reservation is confirmed. The spot sensor sends the occupancy status to the database when triggered or released. The camera picks up the license plate when exiting and tracks the exit time which is used to measure the length of the stay. These procedures are the same for walk-in users with one exception, the reservation is made on-site. Managers and operators of garages can manage a network of garages, adjust pricing, and gather statistical data to effectively estimate overbooking by requesting the data from the database.

6.2 - Concept Definitions

Responsibility Description	**Type**	**Concept Name**
Allows user to register	D	registrationInterface
Obtain user ID data and request vacancy data from database	D	walkInInterface
Receive reservation parameters	K	reservationInterface
Coordinates actions of concepts	D	controller
Takes user to designated floor	D	elevator
Recognizes license plate	K	licensePlateReader

Identify occupancy in a spot	K	spotSensor
Store all pertinent information	K	database
Allows managers to set garage parameters and request data from the database	D	garageMgrInterface

6.3 - Association Definitions

Concept Pair	**Association Description**	**Association Name**
registrationInterface ↔ database	Receives user registration information and updates the database with the new user	Send reg data
reservationInterface ↔ database	Sends reservation details for new reservation to the database	Send reg parameters
walkInInterface ↔ database	Receives reservation parameters sends to database	Send res data
Database ↔ walkInInterface	Database confirms vacancy for walkin in users	Confirm res
Controller ↔ walkInInterface	Allows access for walk in users	Access control
licensePlateReader ↔ database	Sends plate information to database for reservation confirmation	Send plate data
Database ↔ licensePlateReader	Send reservation confirmation	Confirm res
licensePlateReader ↔ controller	Request elevator access	Convey req
Controller ↔ elevator	Request elevator movement	Convey req
spotSensor ↔ database	Sends spot occupancy information to database	Send occupancy data

garageMgrInterface ↔ database	Send user account status and garage parameters to database and request statistical data from database	Send data
Database ↔ garageMgrInterface	Database sends statistical data to the manager	Send data

6.4 - Attribute Definitions

Concept	**Attribute**	**Definition**
reservationInterface	reservationParameters	Receives reservation parameters to push to database
walkInInterface	reservationParameters	Receives reservation parameters to push to database
	userData	Receives user ID to confirm account
licensePlateReader	timeStampEntry	Records time at entry
	timeStampExit	Records time at exit
elevator	floorNumber	Receives assigned floor number
	parkingSpot	Displays assigned parking spot
garageMgrInterface	garageParameters	Sends garage configuration parameters to add new garage to the owner's network, and updates overbooking configuration
	getStats	Requests statistics from the database
database	userData	User data tables to be updated and accessed
	reservationNumber	Reservation numbers to confirm advanced reservations
	parkingInventory	Maintain parking

		occupancy/vacancy inventory
	parkingStats	Parking usage statistics
	garageData	Garage configuration tables

6.5 - Traceability Matrix

C-1 : registrationInterface **C-2:** reservationInterface **C-3:** walkInInterface **C-4:** elevator **C-5:** controller **C-6:** licensePlateReader **C-7:** spotSensor **C-8:** garageMgrInterface **C-9:** database

Use Case	**PW**	**C-1**	**C-2**	**C-3**	**C-4**	**C-5**	**C-6**	**C-7**	**C-8**	**C-9**
UC-01	35	X								X
UC-02a	62		X							X
UC-02b	24		X							X
UC-03	21		X							X
UC-04	14					X	X			X
UC-05	12					X	X			X
UC-06	17					X	X			X
UC-07	32		X							X
UC-08	23					X	X			X
UC-09	39	X								X
UC-10a	75				X	X	X			X
UC-10b	69			X		X		X		X
UC-11	20	X					X			X
UC-12a	12			X		X				X
UC-12b	18			X		X	X			X

UC-13	21							X		X
UC-14	10								X	X
UC-15	41								X	X
UC-16	11								X	X
UC-17	5					X			X	X

Use Cases 01-09 are all associated with the user interacting with the system and all interactions that result in achieving the actor's goal. Each of these use cases require the use of one or more of the following concepts: registration, reservation, and walk-in interfaces. Each of these requires the controller to update and return data from the database.

Use cases 10-12 are all associated with garage access control. These use cases involve the walk-in interface, which controls walk-in access to the garage. Other concepts include the elevator, controller, license plate reader, spot sensor and database. Here, the spot sensor tells the user whether there are vacancies on the ground level walk-in users.

Use case 13 requires the spot sensor and database. This is how occupancy is monitored. Spot sensors are used to identify occupied parking spots. When one is triggered, the database is updated with the new occupancy.

Use case 14 requires the garage management interface and database. Via the garage management interface, operators and administrators can collect statistical data from the database.

Use cases 15 -17 represent the administrative actions that operators must be able to execute. All of these actions are accessed via the garage management interface concept which allows these individuals to add, remove, and edit garages that current exist. They can also update prices for different reservation types. All of these changes must occur in the database for them to take effect, and the new values ass passed through to the controller in this concept.

7 - Interaction Diagrams

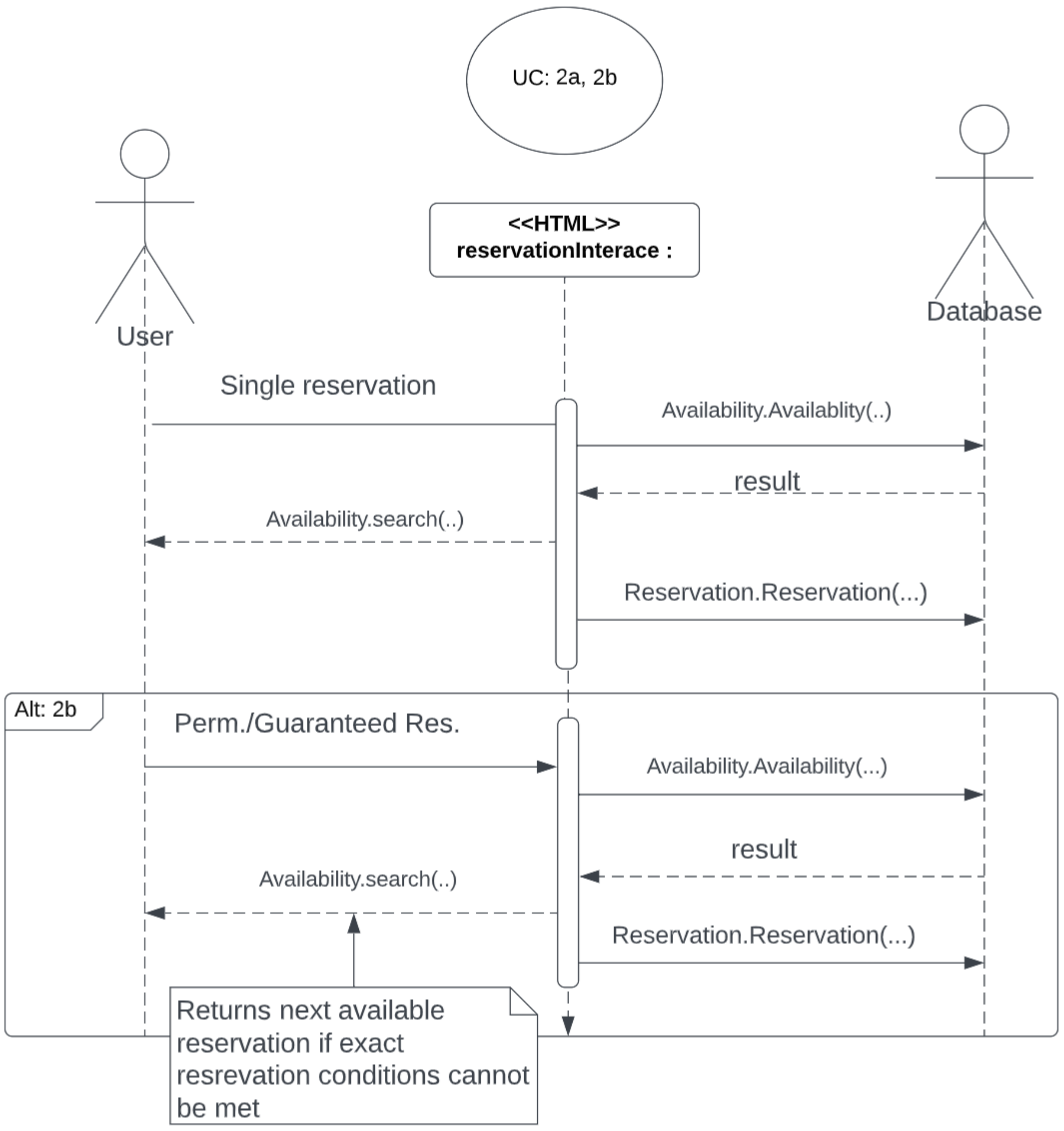
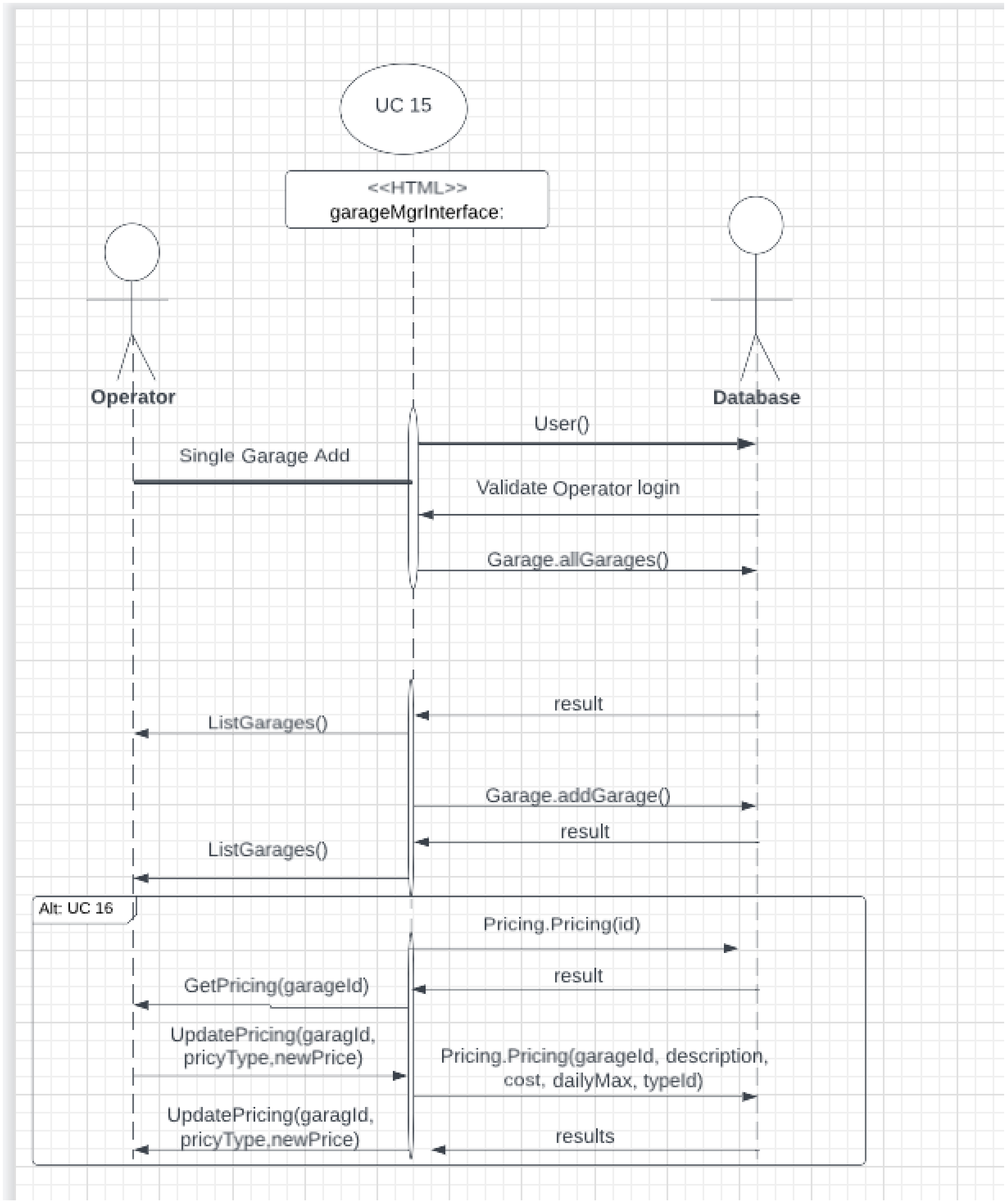

UC: 2a, 2b
<<HTML>> reservationInterace :
User
Database
Single reservation
Availability.Availablity(..)
result
Availability.search(..)
Reservation.Reservation(...)
Alt: 2b
Perm./Guaranteed Res.
Availability.Availability(...)
result
Availability.search(..)
Reservation.Reservation(...)
Returns next available reservation if exact resrevation conditions cannot be met

Figure 7.1 UC-02 Sequence Diagram

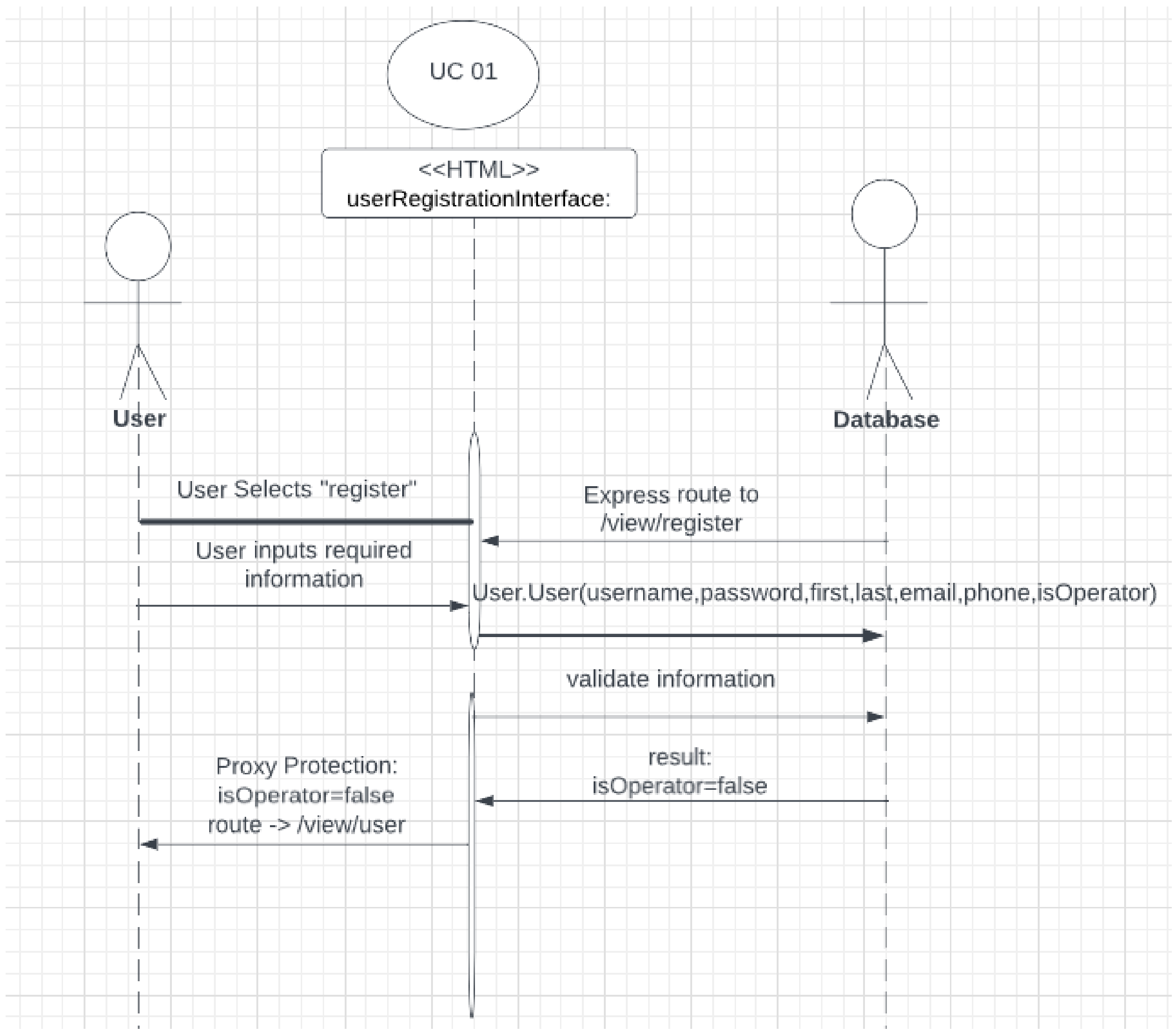
For UC-02, making reservations, we have assigned the responsibility to the reservationInterface concept to receive the information, passed as arguments, to create a reservation. The reservationInterface calls the availableGarage() method to retrieve the data on garages that meet the user's parameters from the database. Therefore, the database is assigned the responsibility of storing and returning results to the interface. The results are returned in the form of an Array of objects in which each element is a garage that meets the user’s requirements. When the user selects the garage they wish to use, the reservationInterface calls the Reservation() method. This method receives the attributes of the selected garage as arguments and creates a new reservation in the database.

An alternate sequence is making a guaranteed or permanent reservation. This follows a similar series of events as the previous sequence, except for when the requested reservation parameters cannot be met. In that event, the next available date is returned to the user and they are able to submit the new reservation parameters.



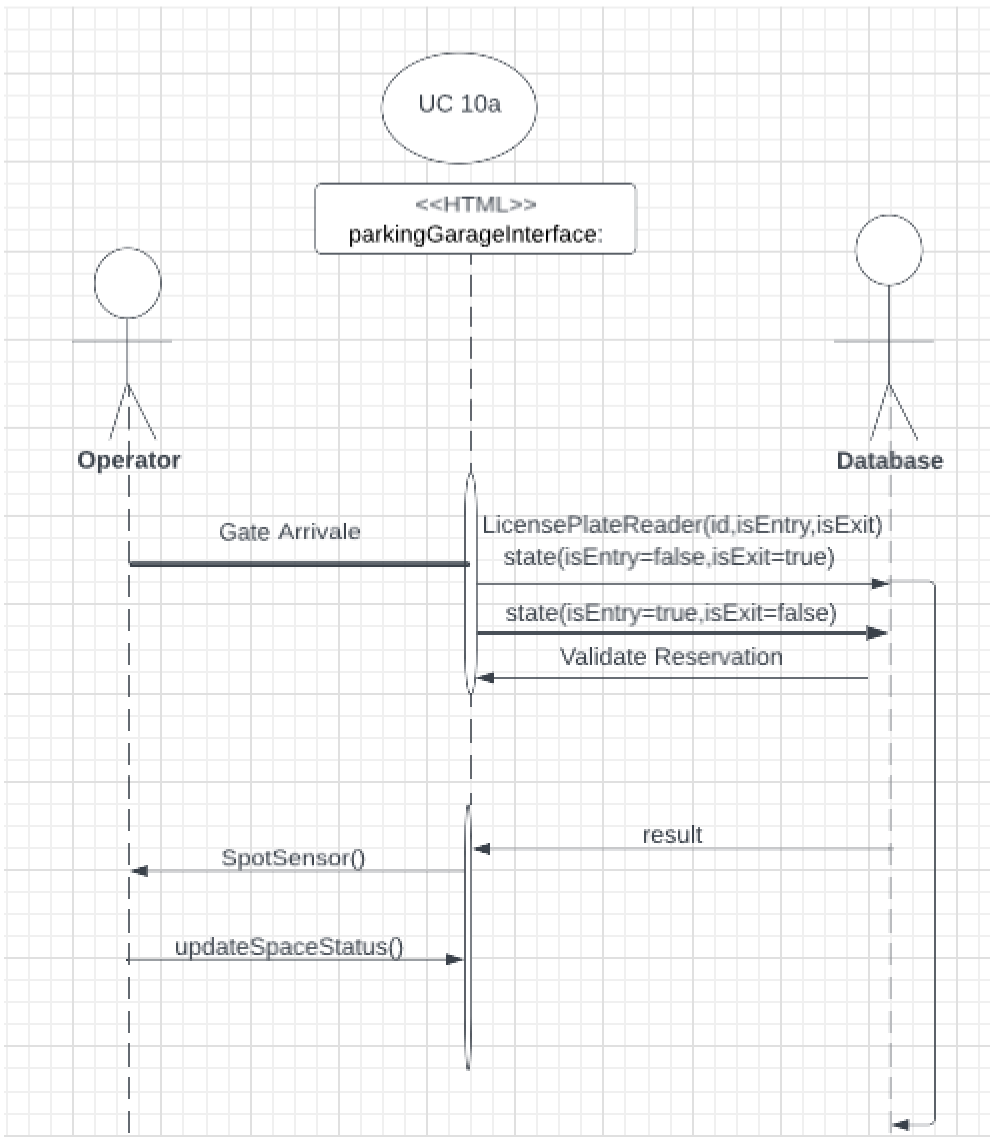
For UC 15, adding a new garage, we have assigned the responsibility to garageMgrInterface concept to receive the information, passed as arguments, to add a new garage to the Operator's network. The Operator will log in; after a successful login, the Operator's garage list will be fetched by allGarages(), which retrieves all garages for that Operator as an Array of Objects. Next ListGarages() will be called to display all of the garages for the Operator to interact with. Then, the Operator will select the ability to add a garage, and from there the garageMgrInterface with the appropriate attributes needed to add a new garage. This will create a new Garage for the Operator.

Alternatively, after a successful login, the Operator will have the opportunity to update existing garages and their pricing. The garageMgrInterface calls the Pricing.Pricing() function to get all of the pricing information for the Operator's garages, which will be an array of objects. The GetPricing() function is then called to display the pricing information for each. After displaying, the Operator will select the garage to update, make changes, then UpdatePricing() is called to get the updated pricing and the garage. The garageMgrInterface then calls on Pricing.Pricing() with more information to update the appropriate garage. The results are returned once again, then UpdatePricing() will be called to reflect the new pricing information.


UC 01
<<HTML>>
userRegistrationInterface:
User
Database
User Selects "register"
Express route to /view/register
User inputs required information
User.User(username,password,first,last,email,phone,isOperator)
validate information
result: isOperator=false
Proxy Protection: isOperator=false route -> /view/user

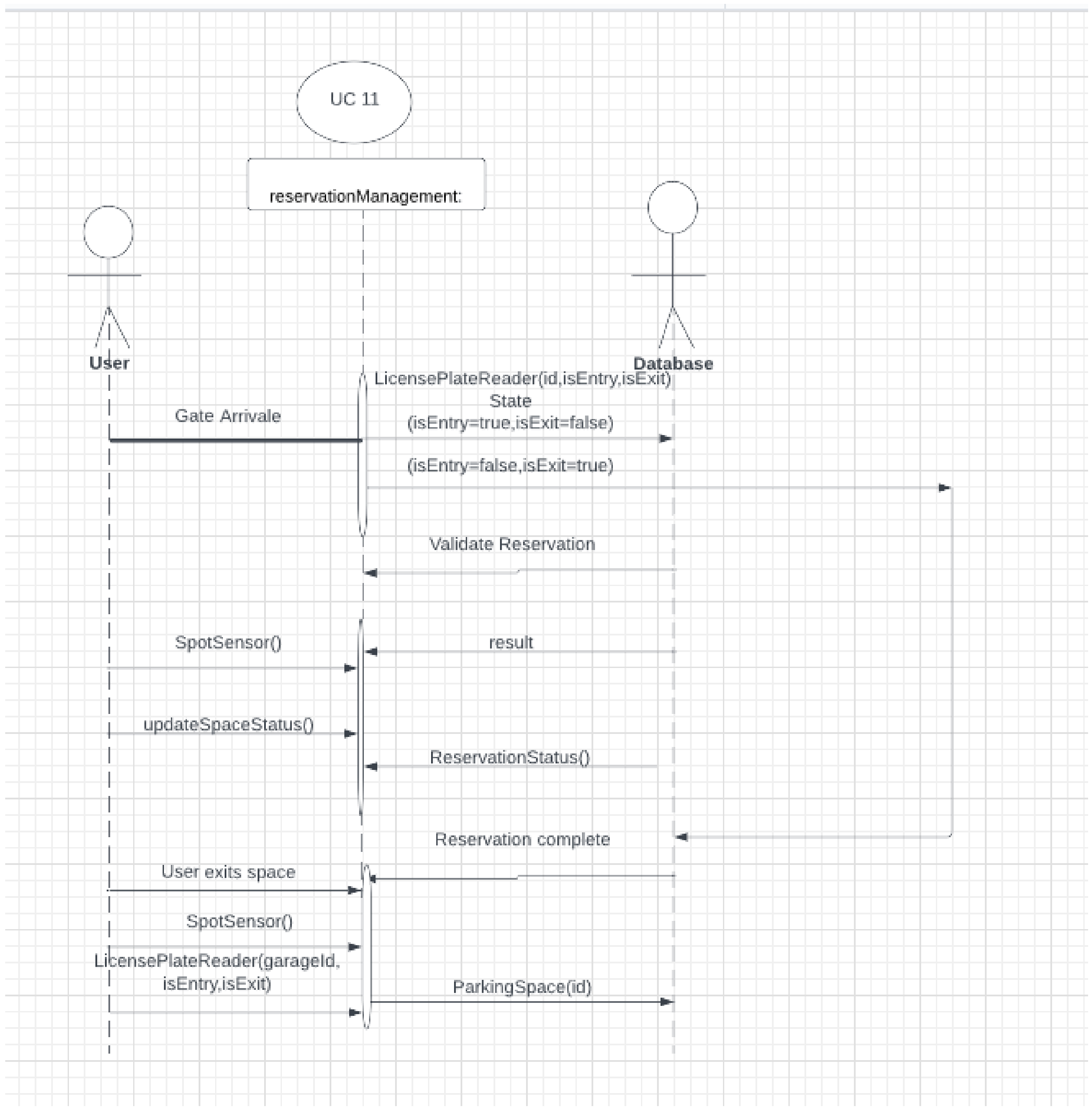
For UC-01, we have designated responsibility to the registrationInterface. The user will select the “Register” option from the Home (Locator) page. From there the user will be taken to the register page. The user will need to put in a unique username, their password, first and last name, email address, and phone number. This will be then sent to the database, alongside a secret isOperator, which will be set to false. There the userController will validate their information, with the correct validated information the result of the successful sign up will be sent to the view/register.html, where the user will see a sign-up successful message.

The Protection Proxy design pattern is used in this model as we have two types of logins: user and operators. Normal users do not need to have access to any operator commands, so the DB model reflects by triggering events based on the isOperator boolean value. In UC-01 this a User signing up for an account, so their isOperator variable will be automatically set to false. Setting this to false will trigger the server to lead them back to their User account page. Additionally, a future feature is to add more protection with the Proxy Pattern to add server-side protection to the Operator functions to check against the user attempting to trigger those functions.



For UC-10a , a user arriving for their reservation at the garage, we have assigned that responsibility to the parkingGarageInterface. The interface will be at the entrance of the garage when the user arrives. The user will arrive at the gate with their information for their reservation, with the appropriate information, combined with the LicensePlateReader() function to ensure this is their registered vehicle. The database will confirm the reservation with this information from the user and LicensePlateReader(), the result will be sent to the gate, and if accurate will display their parking spot number and let them through. At that point, the parking spot sensor will detect their vehicle, which will then call updateSpaceStatus() to send the occupancy information to the database.

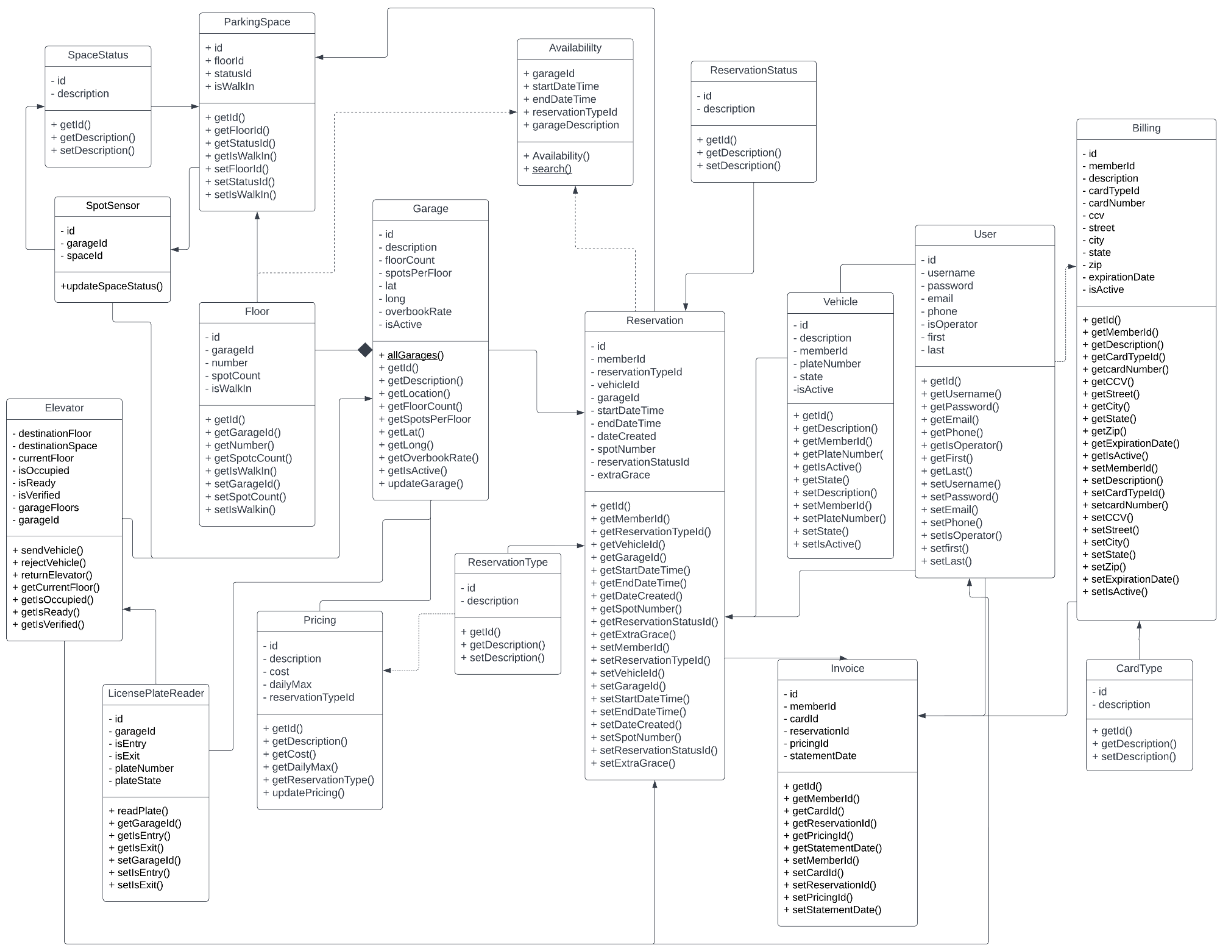
The state pattern is used here to determine if the user at the gate isEntry (entering), isExit(leaving), both as boolean values. These were implemented as a part of the State Pattern due to them both triggering different events for the Reservation, so isEntry=true, the Reservation is just beginning, opposite isExit=true, the reservation is ending; thus would be 2 different states, triggering different code server side.



For UC-11, there will not be a direct user interface, rather relying on sensors and cameras. After the user arrives for their reservation, they will be there for their allotted time. Upon leaving, the SponSensor() will update the database with the information that the user is leaving that spot. updateSpaceStatus() will be called updating that space to empty. With the user exiting the LicensePlateReader() sends the needed information to the garage, the user is exiting, then will update that ParkingSpace() with its id, and complete the reservation. This portion of the code utilizes the State Pattern as what happens when someone is arriving at the gate changes the state of the reservation. For example, if someone is arriving at the gate and isEntry state is true, then the reservation will then begin. On the flip side, if isEntry is false and isExit state is true, then the reservation is ending. The states of these two variables determine what server code will be triggered.

8 - Class Diagram and Interface Specification

8.1 - Class Diagram



8.2 - Data Types and Operation Signatures

Availability

+ garageId : integer
+ garageDescription : string
+ startDateTime : string
+ endDateTime : string
+ reservationTypeId : integer
+ garageDescription

+ Availability(garageId : integer, garageDescription : string, startDateTime : string, endDateTime : string, reservationTypeId : string)
+ search(street : string, city : string, state : string, zip : string, startDateTime : string, endDateTime : string, reservationTypeId : integer) : Availability[*]

- Class
  - Represents an available parking space in a garage matching the parameters passed to the static search function.
- Attributes
  - garageId - The garage for this space
  - garageDescription - The description of the garage
  - startDateTime - When the reservation would start
  - endDateTime - When the reservation would end
  - reservationTypeId - The type of reservation
- Methods
  - Availability() - Constructor
  - search() - A static method that takes the search criteria and returns an array of Availability results

Billing

- id : integer
- memberId : integer
- description : string
- cardTypeId : integer
- cardNumber : string
- ccv : integer
- street : string
- city : string
- state : string
- zip : string
- expirationDate : string
- isActive : boolean

+ Billing(id : integer)
+ Billing(memberId : integer, description : string, cardTypeId : integer, cardNumber - string, ccv : integer, street : string, city : string, state : string, zip : string,

```
expirationDate : string, isActive : boolean)
+ get() :
+ getId() : integer
+ getMemberId() : integer
+ getDescription() : string
+ getCardTypeId() : integer
+ getCardNumber() : string
+ getCcv() : integer
+ getStreet() : string
+ getCity() : string
+ getState() : string
+ getZip() : string
+ getExpirationDate() : string
+ getIsActive() : boolean
+ setMemberId(memberId : integer) : boolean
+ setDescription(description : string) : boolean
+ setCardTypeId(cardTypeId : string) : boolean
+ setCardNumber(cardNumber : string) : boolean
+ setCcv(ccv : integer) : boolean
+ setStreet(street : string) : boolean
+ setCity(city : string) : boolean
+ setState(state : string) : boolean
+ setZip(zip : string) : boolean
+ setExpirationDate(expirationDate : string) : boolean
+ setIsActive(isActive : boolean) : boolean
```

- Class
  - Represents a billing account for a user
- Attributes
  - id - The ID of the billing object
  - memberId - The ID of the member the billing data is for
  - description - A description of the billing account
  - cardTypeId - The card type
  - cardNumber - The card number
  - ccv - The card ccv number
  - street, city, state, zip - The billing address
  - expirationDate - The card expiration date
  - isActive - Whether the billing account is active
- Methods
  - Constructors
    - Billing(id) - Populates the object with an existing billing account from the database using the billing ID
    - Billing(memberId, description, cardTypeId, cardNumber, ccv, street, city, state, zip, expirationDate, isActive) -Creates a new billing account

    - A full set of getters and setters

CardType
- id : integer - description : string
+ CardType(id : integer) + CardType(description : string) + getId() : integer + getDescription() : string + setDescription(description : string) : boolean

- Class
    - Represents a payment card type
- Attributes
    - id - The card type ID
    - description - A description of the card type
- Methods
    - Constructors
        - CardType(id) - Populates the object with an existing card type from the database using the card type ID
        - CardType(description) - Creates a new card type
    - A full set of getters and setters

DatabaseConnection
- connectionString : string
+ DatabaseConnection(connectionString : string)

- Class
    - Forms the connection between the system and the database
- Attributes
    - connectionString - The connection string to the database
- Methods
    - Constructors
        - DatabaseConnection(connectionString) - Creates a new database connection
    - A full set of getters and setters

Elevator
- destinationFloor : integer - destinationSpace : integer - currentFloor : integer - isOccupied : boolean - isReady : boolean - isVerified : boolean - garageFloors : integer - garageId : integer
+ Elevator(floors : integer, garageId : integer) + sendVehicle() : boolean + rejectVehicle() : boolean + returnElevator() : void + getCurrentFloor() : integer + getIsOccupied() : boolean + getIsReady() : boolean + getIsVerified() : boolean

- Class
  - Represents a vehicle elevator in a garage
- Attributes
  - destinationFloor - The floor the current vehicle will park on
  - destinationSpace - The space the current vehicle will park in
  - currentFloor - The floor the elevator is currently at
  - isOccupied - Whether there is a vehicle in the elevator
  - isReady - Whether the elevator is ready to move
  - isVerified - Whether the vehicle currently in the elevator has a verified reservation
  - garageFloors - The number of floors in the garage
  - garageId - The garage the elevator is in
- Methods
  - Constructors
    - Elevator(floors, garageId) - Populates the object with an existing elevator from the database using the elevator ID
  - sendVehicle() - Sends the vehicle in the elevator to a floor
  - rejectVehicle() - Refuses the current vehicle entry to the garage
  - returnElevator() - Returns the elevator to the ground floor
  - A full set of getters for currentFloor, isOccupied, isReady, and isVerified

Floor

- id : integer
- garageId : integer
- floorNumber : integer
- spaceCount : integer
- isWalkIn : boolean

+ Floor(id : integer)
+ Floor(garageId : integer, floorNumber : integer, spaceCount : integer)
+ getId() : integer
+ getGarageId() : integer
+ getFloorNumber() : integer
+ getSpaceCount() : integer
+ getIsWalkIn() : boolean
+ setGarageId(garageId : integer) : boolean
+ setFloorNumber(floorNumber : integer) : boolean
+ setSpaceCount(spaceCount : integer) : boolean
+ setIsWalkIn(isWalkIn : boolean) : boolean

- Class
    - Represents a floor in a garage
- Attributes
    - id - The floor ID
    - garageId - The garage the floor is in
    - floorNumber - The number of the floor in the garge
    - spaceCount - The number of spaces on the floor
    - isWalkIn - Whether the floor is for walk-ins
- Methods
    - Constructors
        - Floor(id) - Populates the object with an existing floor from the database using the floor ID
        - Floor(garageId, floorNumber, spaceCount) - Creates a new floor
    - A full set of getters and setters

Garage

- id : integer
- description : string
- floorCount : integer
- spotsPerFloor: integer[*]
- lat: string
- long - string
- overbookRate : float
- isActive : boolean

+ Garage(id : integer) + Garage(garageName : string, numFloors : integer, spotsPerFloor: integer[*], location: string[2], overbookRate : float, isActive : boolean) + allGarages() : Garage[*] + getId() : integer + getDescription() : string + getFloorCount() : integer + getSpotsPerFloor(): float[*] + getLat() : string + getLong() : string + getOverbookRate() : float + getIsActive() : boolean + updateGarage(garageName : string, overbookRate : float, floors : integer, spotsPerFloor : integer[*], location : string[2], isActive : boolean) : boolean

- Class
  - Represents a parking garage
- Attributes
  - id - The parking garage ID
  - description - A description of the garage
  - floorCount - The number of floors in the garage
  - lat, long - The coordinates of the garage
  - overbookRate - The allowed rate of overbooking for the garage
  - isActive - Whether the garage is active in the system
- Methods
  - Constructors
    - Garage(id) - Populates the object with an existing garage from the database using the garage ID
    - Garage(garageName, numFloors, location, overbookRate, isActive) - Creates a new garage
  - allGarages() - A static method to return an array of all garages in the system
  - A full set of getters
  - updateGarage(garageName, overbookRate, floors, spotsPerFloor, location, isActive) - updates the garage record

Invoice
- id : integer - memberId : integer - cardId : integer - reservationId : integer

- pricingId : integer - statementDate : string
+ Invoice(id : integer) + Invoice(memberId : integer, cardId : integer, reservationId : integer, pricingId : integer, statementDate : string) + getId() : integer + getMemberId() : integer + getCardId() : integer + getReservationId() : integer + getPricingId() : integer + getStatementDate() : string + setMemberId(memberId : integer) : boolean + setCardId(cardId : integer) : boolean + setReservationId(reservationId : integer) : boolean + setPricingId(privingId : integer) : boolean + setStatementDate(statementDate : string) : boolean

- Class
  - Represents an invoice for an individual member
- Attributes
  - id - The invoice id
  - memberId - The ID of the member the invoice is for
  - cardId - The card to be charged
  - reservationId - The reservation the invoice is for
  - pricingId - The price to be charged
  - statementDate - The date of the invoice
- Methods
  - Constructors
    - Invoice(id) - Populates the object with an existing invoice from the database using the invoice ID
    - Invoice(memberId, cardId, reservationId, pricingId, statementDate) - Creates a new invoice
  - A full set of getters and setters

LicensePlateReader
- id : integer - garageId : integer - isEntry : boolean - isExit : boolean - plateNumber : string

- plateState: string
+ LicensePlateReader(id : integer) + LicensePlateReader(garageId : integer, isEntry : boolean, isExit : boolean) + readPlate() : string[2] + getGarageId() : integer + getIsEntry() : boolean + getIsExit() : boolean + setGarageId(garageId : integer) : boolean + setIsEntry(isEntry : boolean) : boolean + setIsExit(isExit : boolean) : boolean

- Class
    - Represents a license plate reader in the garage
- Attributes
    - id - The ID of the reader
    - garageId - The garage the reader is in
    - isEntry - Whether the reader is for entry control
    - isExit - Whether the reader is for exit control
- Methods
    - Constructors
        - LicensePlateReader(id) - Populates the object with an existing license plate reader from the database using the reader ID
        - LicensePlateReader(garageId, isEntry, isExit) - Creates a new license plate reader
    - readPlate() - Reads a license plate
    - A full set of getters and setters

ParkingSpace
- id : integer - floorId : integer - statusId : integer - isWalkIn : boolean
+ ParkingSpace(id : integer) + ParkingSpace(floorId : integer, statusId : integer, isWalkIn : boolean) + getId() : integer + getFloorId() : integer + getStatusId() : integer + getIsWalkIn() : boolean + setFloorId(floorId : integer) : boolean + setStatusId(statusId : integer) : boolean

+ setIsWalkIn(isWalkIn : boolean) : boolean

- Class
  - Represents a single parking space in a garage
- Attributes
  - id - The parking space ID
  - floorId - The floor the parking space is on
  - statusId - The status of the space
  - isWalkIn - Whether the space is for walk-ins
- Methods
  - Constructors
    - ParkingSpace(id) - Populates the object with an existing space from the database using the space ID
    - ParkingSpace(floorId, statusId, isWalkIn) - Creates a new space
  - A full set of getters and setters

Pricing
- id : integer - description : string - cost : string - dailyMax : integer - reservationTypeId : integer
+ Pricing(id : integer) + Pricing(description : string, cost : string, dailyMax : integer, reservationTypeId : integer) + getId() : integer + getDescription() : string + getCost() : string + getDailyMax() : integer + getReservationTypeId() : integer + updatePrice(cost : string, dailyMax : integer)

- Class
  - Represents a pricing type for parking
- Attributes
  - id - The pricing type ID
  - description - The description of the price
  - cost - The cost per time period
  - dailyMax - The maximum daily charge for this price type
  - reservationTypeId - The type of reservation

- Methods
    - Constructors
        - Pricing(id) - Populates the object with an existing price type from the database using the price type ID
        - Pricing(garageId, description, cost, dailyMax, reservationTypeId) - Creates a new price
    - A full set of getters
    - updatePrice(cost : string, dailyMax : integer) - updates the pricing record

Reservation
- id : integer - memberId : integer - reservationTypeId : integer - vehicleId : integer - garageId : integer - startDateTime : Date - endDateTime : Date - dateCreated: Date - spotNumber : integer - reservationStatusId : integer - extraGrace : boolean
+ Reservation(id : integer) + Reservation(memberId : integer, reservationTypeId : integer, vehicleId : integer, garageId : integer, lat: float, lon: float, startDateTime : string, endDateTime : string, reservationStatusId : integer) + getId() : integer + getMemberId() : integer + getReservationTypeId() : integer + getVehicleId() : integer + getGarageId() : integer + getStartDateTime() : Date + getEndDateTime() : Date + getDateCreated() : Date + getSpotNumber() : integer + getReservationStatusId() : integer + getExtraGrace() : boolean + setMemberId(memberId : integer) : boolean + setReservationTypeId(reservationTypeId : integer) : boolean + setVehicleId(vehicleId : integer) : boolean + setGarageId(garageId : integer) : boolean + setStartDateTime(startDateTime : string, timezone: string) : boolean + setEndDateTime(endDateTime : string, timezone: string) : boolean

+ setSpotNumber(spotNumber : integer) : boolean + setReservationStatusId(reservationStatusId : integer) : boolean + setExtraGrace(extraGrace : boolean) : boolean

- Class
    - Represents a reservation for a parking space
- Attributes
    - id - Reservation ID
    - memberId - ID number of the member the reservation is for
    - reservationTypeId - The type of reservation
    - vehicleId - The vehicle the reservation is for
    - garageId - The garage the reservation is for
    - startDateTime - When the reservation starts
    - endDateTime - When the reservation ends
    - dateCreated - The creation timestamp of the reservation
    - spotNumber - The parking spot assigned to the reservation
    - reservationStatusId - The status of the reservation
    - extraGrace - Whether the user is given an extra grace period after the reservation end time has passed
- Methods
    - Constructors
        - Reservation(id) - Populates the object with an existing reservation from the database using the reservation ID
        - Reservation(memberId, reservationTypeId, vehicleId, garageId, lat, lon, startDateTime, endDateTime, reservationStatusId) - Creates a new reservation with the time understood as being in the [lat, lon] timezone
    - A full set of getters and setters

ReservationStatus
- id : integer - description : string
+ ReservationStatus(id : integer) + ReservationStatus(description : string) + getId() : integer + getDescription() : string + setDescription(description : string) : boolean

- Class
    - Represents a reservation status

- Attributes
    - id - The reservation status ID
    - description - Describes the reservation status
- Methods
    - Constructors
        - ReservationStatus(id) - Populates the object with an existing reservation status from the database using the reservation status ID
        - ReservationStatus(description) - Creates a new reservation status
    - A full set of getters and setters

ReservationType
- id : integer - description : string
+ ReservationType(id : integer) + ReservationType(description : string) + getId() : integer + getDescription() : string + setDescription(description : string) : boolean

- Class
    - Represents a type of parking space reservation
- Attributes
    - id - The reservation type ID
    - description - Describes the reservation type
- Methods
    - Constructors
        - ReservationType(id) - Populates the object with an existing reservation type from the database using the reservation type ID
        - ReservationType(description) - Creates a new reservation type
    - A full set of getters and setters

SpaceStatus
- id : integer - description : string

+ SpaceStatus(id : integer)
+ SpaceStatus(description : string)

+ getId() : integer + getDescription() : string + setDescription(description : string) : boolean

- Class
    - Represents a possible status for a parking space
- Attributes
    - id - The status ID
    - description - A description of the space status
- Methods
    - Constructors
        - SpaceStatus(id) - Populates the object with an existing space status from the database using the space status ID
        - SpaceStatus(description)
    - A full set of getters and setters

SpotSensor
- id : integer - garageId : integer - spaceId : integer

+ SpotSensor(garageId : integer, spaceId : integer)
+ updateSpaceStatus(statusId : integer)

- Class
    - Represents a sensor for detecting occupancy of a single parking space
- Attributes
    - id - The ID of the spot sensor
    - garageId - The garage the sensor is in
    - spaceId - The parking space the sensor is for
- Methods
    - Constructors
        - SpotSensor(garageId, spaceId)
    - updateSpaceStatus() - Updates the status of the parking space

User
- id : integer - username : string

- password : string - first : string - last : string - email : string - phone : string - isOperator : boolean
+ User(id : integer) + User(username : string, password : string, first : string, last : string, email : string, phone : string, isOperator : boolean) + User(username : string, password : string) + getId() : integer + getUsername() : string + getPassword() : string + getFirst() : string + getLast() : string + getEmail() : string + getPhone() : string + getIsOperator() : boolean + setUsername(username : string) : boolean + setPassword(password : string) : boolean + setFirst(first : string) : boolean + setLast(last : string) : boolean + setEmail(email : string) : boolean + setPhone(phone : string) : boolean + setIsOperator(isOperator : boolean) : boolean

- Class
    - Represents a user of the system
- Attributes
    - id - The user ID
    - username - The username
    - password - The user’s password
    - first - The user’s legal first name
    - last - The user’s legal last name
    - email - The user’s email address
    - phone - The user’s contact phone number
    - isOperator - Whether the user is a garage operator
- Methods
    - Constructors
        - User(id) - Populates the object with an existing user account from the database using the user ID
        - User(username, password, first, last, email, phone, isOperator) - Creates a new user
        - User(username, password) - Returns a user object if the username

        and password match an existing user
    - A full set of getters and setters

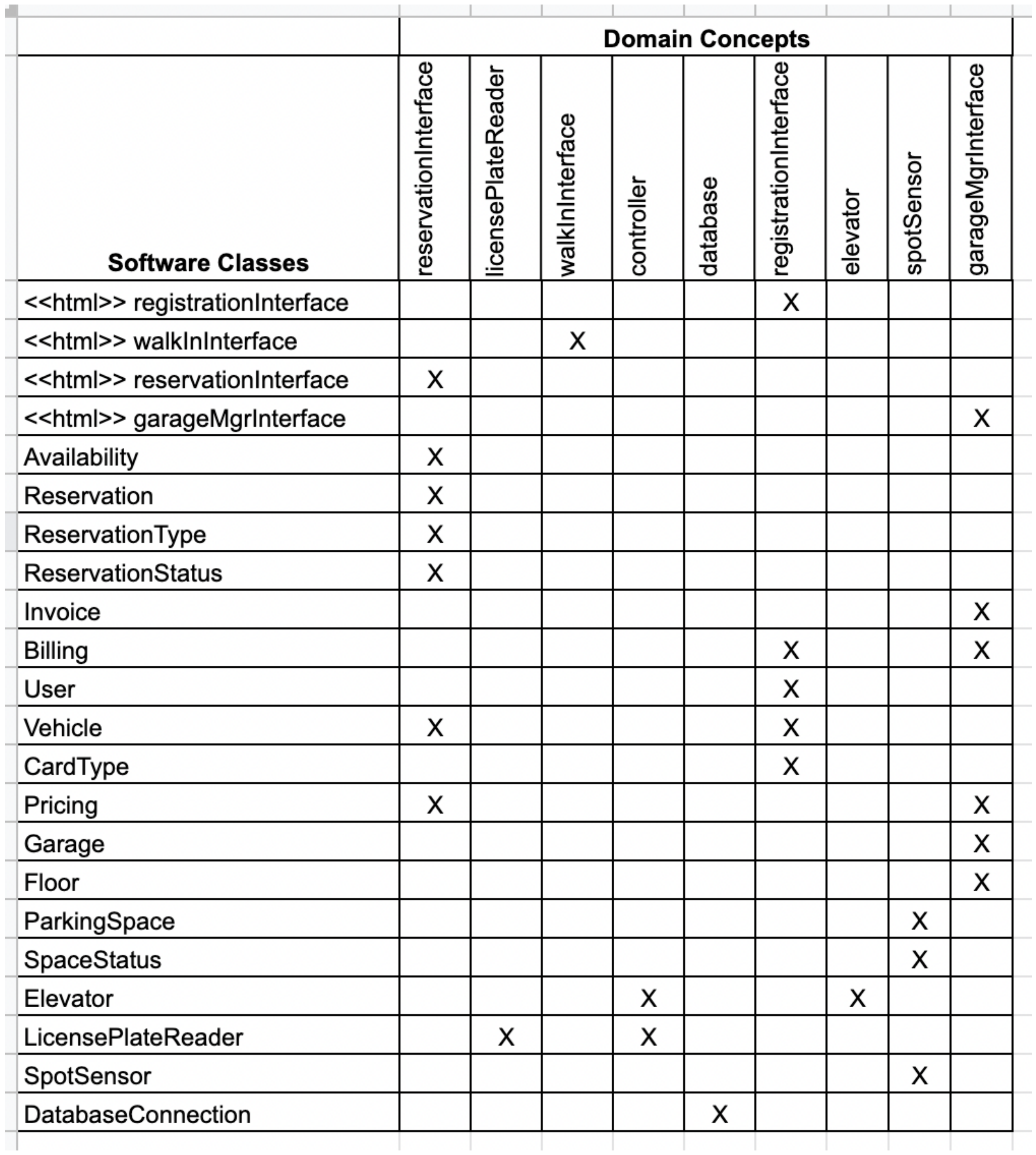
Vehicle
- id : integer - memberId : integer - plateNumber : string - state : string - isActive : boolean - description : string
+ Vehicle(id : integer) + Vehicle(plateNumer : string, state : string) + Vehicle(memberId : integer, plateNumber : string, state : string, isActive : boolean, description : string) + getId() : integer + getMemberId() : integer + getPlateNumber() : string + getState() : string + getIsActive() : boolean + getDescription() : string + setMemberId(memberId : integer) : boolean + setPlateNumber(plateNumber : string) : boolean + setState(state : string) : boolean + setIsActive(isActive : boolean) : boolean + setDescription(description : string) : boolean

- Class
    - Represents a vehicle registered to a member
- Attributes
    - id - The vehicle ID
    - memberId - The ID of the member the vehicle is for
    - plateNumber - The license plate number of the vehicle
    - state - The license plate state of the vehicle
    - isActive - Whether the vehicle is active for the user
    - description - A description of the vehicle
- Methods
    - Constructors
        - Vehicle(id) - Populates the object with an existing vehicle from the database using the vehicle ID
        - Vehicle(plateNumer, state) - Populates remaining attributes if a matching vehicle is found
        - Vehicle(memberId, plateNumber, state, isActive, description) -

Creates a new vehicle

- A full set of getters and setters

8.3 - Traceability Matrix



	Domain Concepts								
Software Classes	reservationInterface	licensePlateReader	walkInInterface	controller	database	registrationInterface	elevator	spotSensor	garageMgrInterface
<<html>> registrationInterface						X			
<<html>> walkInInterface			X						
<<html>> reservationInterface	X								
<<html>> garageMgrInterface									X
Availability	X								
Reservation	X								
ReservationType	X								
ReservationStatus	X								
Invoice									X
Billing						X			X
User						X			
Vehicle	X					X			
CardType						X			
Pricing	X								X
Garage									X
Floor									X
ParkingSpace								X	
SpaceStatus								X	
Elevator				X			X		
LicensePlateReader		X		X					
SpotSensor								X	
DatabaseConnection					X				

The reservationInterface concept has led to the creation of several classes: Availability, Reservation, ReservationType, ReservationStatus, Vehicle, and Pricing, as well as the interface HTML page. Availability is for search results when making the reservation. Reservation represents a parking reservation in the system with a Reservation Type and Status. Vehicle is the vehicle the reservation is made for, and Pricing is the price per time period of the reservation.

The registrationInterface concept has led to several classes: Billing, User, Vehicle, and CardType, as well as the interface HTML page. The User class is for users who register through the interface. Vehicle is associated with a user as the vehicle they wish to park. Billing represents a billing account, which must have a payment card and associated CardType.

The spotSensor concept led to: ParkingSpace, SpaceStatus, and SpotSensor. ParkingSpace represents a physical parking space in a garage, with an associated SpaceStatus updated by the associated SpotSensor for the parking space.

The garageMgrInterface concept gave rise to the interface HTML page, as well as: Invoice, Billing, Pricing, Garage, and Floor. Billing is required for each user to pay for parking, which is charged with an Invoice from the garage based on Pricing set for each garage. The Garage is the garage which is being managed with a specified number of Floors.

The Elevator and LicensePlateReader classes have subsumed the Controller concept, taking on the control functions relevant to themselves.

8.4 - Design Patterns

Given that the system interfaces are web pages communicating with the system through HTTP requests, the Remote Proxy pattern is used throughout the design. Each controller in the system corresponds to a route, with request and response objects passed between the front-end UI page across the network to the controllers. This allows for simple requests and predictable responses across the network barrie between the front and back ends.

Relatedly, the Protection Proxy will be used to protect the Garage and Pricing interfaces from unauthorized access by regular users. The system will automatically route users away from these interfaces based on their access privileges, even if they attempt to access the resources directly. Similarly, any access to the API endpoints for

Garages and Pricing will eventually be handled with a Protection Proxy to prevent unauthorized attempts to directly access the system functions.

The implementation of the DatabaseConnection class follows the Singleton Pattern. On system start-up, the DatabaseConnection is first instantiated with the connection details. Any subsequent calls to the DatabaseConnection do not recreate the connection, and instead return the same single connection created at program startup. Such an arrangement ensures that the system and database do not waste resources on managing multiple instances of the connection.

8.5 - OCL Contracts

UC-01 - User Registration

contract User **inv UniqueUsernames:** `User.allInstances()->forAll(u | u = self or (u.username != self.username and u.id != seld.id))`

```
contract User::User(username : string, password : string, first
     : string, last : string, email : string, phone : string,
     isOperator : boolean)
pre: User.allInstances()->forAll(u | u.username != username)
post: User.allInstances()->one(u | u.oclIsNew() and
          u.username=username and u.password=password and
          u.first = first and u.last = last and u.email=email
               and
          u.phone=phone and u.isOperator=isOperator
     )
```

UC-02a & UC-02b - Requesting a Single or Guaranteed Parking Spot

```
contract Availability::search(street : string, city : string,
     state : string, zip : string, startDateTime : string,
     endDateTime : string, reservationTypeId : integer)
pre: Garage.allInstances()->size() > 0 and startDateTime <
     endDateTime and startDateTime >= DateTime.Now
```

```
contract Reservation inv ValidFKs:
     User.allInstances()->exists(u | u.id=self.memberId) and
     Garage.allInstances()->exists(g | g.id=self.garageId) and
     ReservationType.allInstances()->exists(rt | rt.id=
     self.reservationTypeId) and
     Vehicle.allInstances()->exists(v | v.id=self.vehicleId) and
     ReservationStatus.allInstances()->exists(rs | rs.id=
     self.reservationStatusId)

contract Reservation inv ValidDates:
     self.startDateTime > self.endDateTime and
     self.startDateTime >= self.dateCreated

contract Reservation inv ValidCoords:
     self.lat <= 180 and self.lat >=-180 and
     self.lon <= 180 and self.lon >= -180

contract Reservation::Reservation(memberId : integer,
     reservationTypeId : integer, vehicleId : integer, garageId
     : integer, lat: float, lon: float, startDateTime : string,
     endDateTime : string,reservationStatusId : integer)
pre: User.allInstances()->exists(u | u.id=memberId) and
     Garage.allInstances()->exists(g | g.id=garageId) and
     ReservationType.allInstances()->exists(rt | rt.id=
     reservationTypeId) and
     Vehicle.allInstances()->exists(v | v.id=vehicleId) and
     ReservationStatus.allInstances()->exists(rs | rs.id=
     reservationStatusId) and
     startDateTime > endDateTime and
     startDateTime >= dateCreated and
     lat <= 180 and lat >=-180 and
     lon <= 180 and lon >= -180
post: Reservation.allInstances()->one(r | r.oclIsNew() and
          r.memberId=memberId and
          r.reservationTypeId=reservationTypeId and
          r.vehicleId=vehicleId and r.garageId=garageId and
          r.startDateTime=startDateTime and
          r.endDateTime=endDateTime and
          r.dateCreated=DateTime.Now and r.spotNumber=null and
          r.reservationStatusId=1 and r.extraGrace=false
     )
```

UC-10a - Reservation Arrivals

```
contract LicensePlateReader::readPlate()
pre: self.isEntry=true and self.isExit=false
post: Reservation.allInstances()->one(r |
          r.garageId=self.garageId and
          r.reservationStatusId=3 and
          Vehicle.allInstances()->exists(v |
               v.id=r.vehicleId and
               v.plateNumber=self.plateNumber and
               v.state=self.plateState
          )
     )

contract SpaceSensor::updateSpaceStatus(statusId : integer)
post: ParkingSpace.allInstances()->exists(ps |
     ps.id=self.spaceId and ((ps.status@pre=0 and
     ps.status@post=1) or (ps.status@pre=1 and
     ps.status@post=0))
```

UC-11 - Reservation Management

```
contract LicensePlateReader::readPlate()
pre: self.isExit=true and self.isEntry=false
post: Reservation.allInstances()->one(r |
          r.garageId=self.garageId and
          (r.reservationTypeId=2 or r.reservationStatusId=3) and
          Vehicle.allInstances()->exists(v |
               v.id=r.vehicleId and
               v.plateNumber=self.plateNumber and
               v.state=self.plateState
          )
     )
```

UC-15 - Adding a Garage to a Customer’s Network

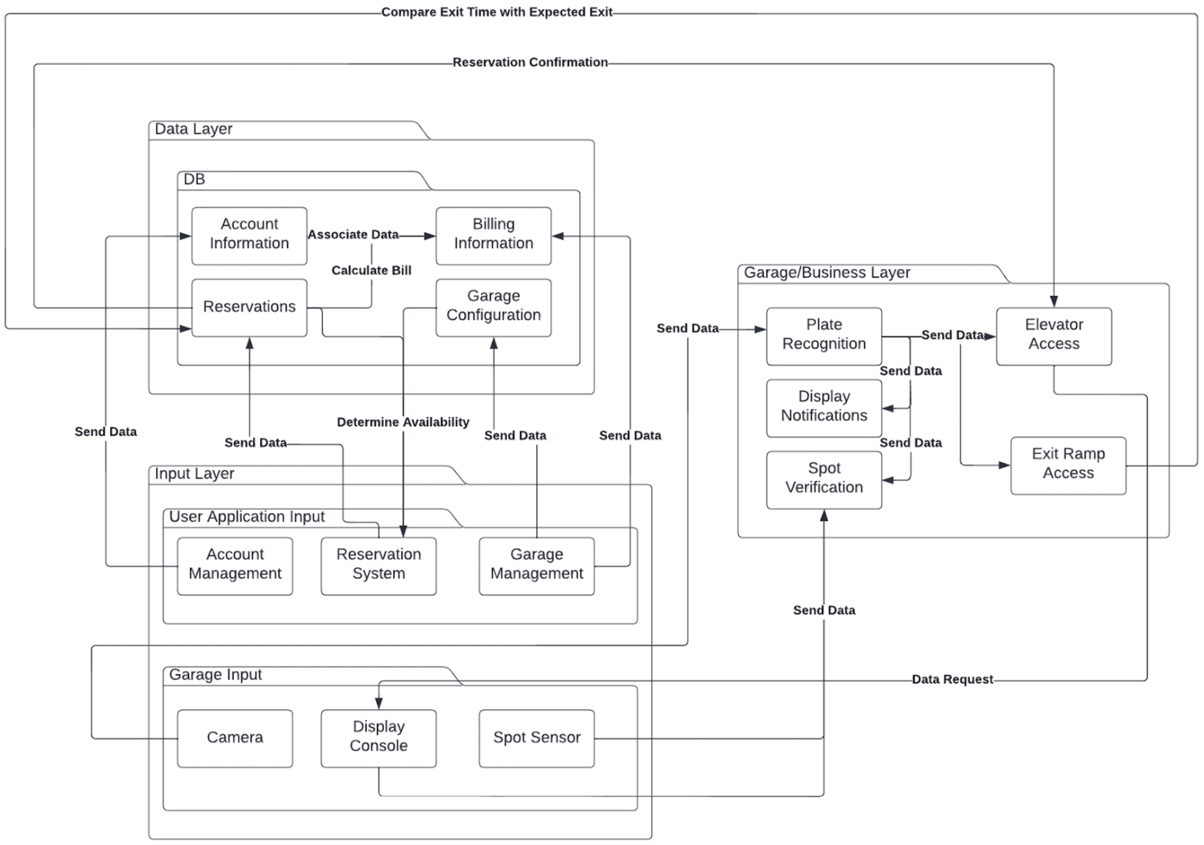
```
contract Garage::Garage(garageName : string, numFloors :
     integer, location: string[2], overbookRate : float,
     isActive : boolean)
pre: numFloors >= 1 and spotsPerFloor->forAll(spf | spf > 0) and
     overbookRate >= 1.0 and spotsPerFloor->size() = numFloors
post: Garage.allInstances()->one(g | g.oclIsNew() and
          g.description=garageName and
          g.floorCount=numFloors and
          g.spotsPerFloor=spotsPerFloor and
          g.lat=location[0] and g.long=location[1] and
          g.overbookRate=overbookRate and g.isActive=isActive
     )

contract Garage::updateGarage(garageName : string, overbookRate
     : float, floors : integer, spotsPerFloor : integer[*],
     location : string[2], isActive : boolean)
pre: floors >= 1 and spotsPerFloor->forAll(spf | spf > 0) and
     overbookRate >= 1.0 and spotsPerFloor->size() = floors
post: self.description=garageName and
     self.overbookRate=overbookRate and self.floorCount=floors
     and self.spotsPerFloor=spotsPerFloor and
     self.lat=location[0] and self.long=location[1] and
     self.isActive=isActive
```

UC-16 - Adjusting Prices for a Garage

contract Pricing::updatePrice(cost : string, dailyMax : integer)
pre: cost != “” and dailyMax > 0
post: self.cost=cost and self.dailyMax=dailyMax

9 - System Architecture and System Design

9.1 - Identifying Subsystems



As shown in the diagram, our system consists of 3 subsystems: the input layer, the data layer, and the Business (Garage) layer. Within the input layer, there are inputs done on the application such as Account Management information, Reservation Information and Garage configuration information. Additional inputs are done in the garage itself upon arrival or departure. These include inputs from automated systems such as the sensors, the hardware, and inputs done on the display console.

In the data layer lies the database. The database and website are cloud hosted on Render. The database will maintain all records for account information, reservations, garage configuration, and more. The input layer and the business layer will send and receive data from the database as needed throughout the usage of the application.

The Garage/Business Layer consists of the actions taken within the garage when a user arrives. This includes considering inputs by the ‘Garage Input’ piece of the Input Layer, and data from the database that was initially sent via the ‘User Application Input’ piece of the input later. This is where many business rules will exist for processing the parking of the users.

9.2 - Architecture Styles

9.2.1 - Software Architecture

Our Software will follow Model-View-Controller Architecture. The Model will consist of our business logic and all the data. Thus, including the database and constraints on the database. This will hold user data, garage configuration, reservations, and more. This data from the Model is reflected in the View, also known as the UI. The view handles all visual components and the layout of the application. The user will input data into the view to be handled by the Controller and checked for requirements and other business logic before updating the database. In some cases, the controller will update the view immediately to reflect errors in the entered data based on the defined business logic and requirements (MDN).

9.2.2 - Network Architecture

The network architecture is going to be a Client-Server architecture. In this scenario, we have two separate clients: Both the user’s device (computer, phone, tablet) and the console on-site at each garage. Both of these clients will communicate with the server the software is hosted on.

9.3 - Mapping Subsystems to Hardware

Our system runs on multiple machines accessing a database concurrently. Users and managers must be able to access the service via individual computers (clients).

The system (server) is hosted either by the customer or through a managed service. The database is cloud-based and hosted offsite.

9.3.1 Garage Input Hardware

Garage input hardware includes the camera and spot sensor. The camera facilitates license plate recognition as a form of reservation verification against reservation records in the database. The spot sensor identifies occupied spaces. This is also communicated to the database through the garage/business layer. Occupancy is then updated and stored in the database based on spot sensor data.

9.3.2 Garage/Business Hardware

The garage hardware includes the elevator and exit control arm. The elevator provides garage access when the reservation information is confirmed, either by plate recognition or user reservation number. The plate reader at the exit reads the vehicle's license plate and updates the exit time, the exit arm then lifts allowing the vehicle to exit the lot. This information is processed in the database to establish billing details.

9.4 - Connectors and Network Protocols

For this project, we have chosen to use the HTTP protocol combo of NodeJs with Express routing. NodeJs was chosen specifically for its familiarity with a number of individuals in the group, as well as its low barrier to entry, zero cost, and extensive documentation. Express routing will be combined with our Node environment. Express (ExpressJS) was chosen to pair with NodeJs due to its vast choices of HTTP utility methods, as well as the very easy-to-use and well-documented API (https://expressjs.com/en/4x/api.html). Another reason ExpressJs was chosen as a partner with NodeJs is that Express routing is much faster and more robust than the majority of NodeJs' built-in server protocols. In a great writeup by Hiren Dhaduk, he goes into great detail as to why Express is so widely used with NodeJs. During Dhaduk's explanation he points out ease of use, and response time making Express superior to NodeJs, especially if your application has more than one page, which this project does. (Dhaduk, 2022).

Other web frameworks do exist that rival ExpressJs, such as Fastify. Fastify touts itself as even faster than ExpressJs. However, Fastify is much newer than Express, and it is far less adopted. Checking the npm website, Fastify has 710,000 downloads a week (Fastify.io, 2022), while Express pushes up over 25 million downloads per week (ExpressJS, 2022). This larger adoption of Express means it is much more well-documented than Fastify. While Fastify claims to be faster than Express, tests are usually done by websites or blogs with a bias toward Fastify. This makes the results somewhat unclear or unreliable.

All the factors above made the combo of NodeJs and Express the ideal choice for our group and project.

9.5 - Global Control Flow

9.5.1 Execution Orderness

Execution is both procedural and event-driven. Reservations are procedural in that the user always creates or modifies a reservation following a sequence of steps. Reservation confirmation and occupancy data are event-driven. Reservation confirmation is initiated when the license plate reader recognizes and validates the plate information. This triggers the elevator. When the spot sensor is triggered, garage occupancy is updated in the database. When a vehicle leaves, the spot sensor is deactivated releasing the spot and making it available again. Billing is event-driven because the billable time is locked in once the vehicle exits the garage. If a vehicle leaves after their original reservation time, the exit time is adjusted to correctly bill the user.

9.5.2 Time Dependency

We have a real-time system. Reservations are based on a start and end time. Billing is calculated based on start and time. Real-time is also necessary to calculate overstays to correctly bill users. There are no limits to reservation windows.

9.6 - Hardware Requirements

Hardware requirements may be broken into several categories:

- **User hardware**
  - The device used by each user to access the service
- **Garage management terminal**
  - The device used by the operator to administer their garage network
- **Garage external informational display**
  - The display outside the garage for displaying available spaces
- **Garage elevator informational display**
  - The display inside the car elevator for communication with users

- **Entrance and exit terminal displays**
    - The terminals used to display and enter reservation details to enter or exit the garage if the system fails to match the license plate to a reservation
- **System server**
    - The server running the garage software system

Other key hardware, including the car elevator, spot occupancy sensors, exit gate arm, and license plate cameras will be sourced from Commercial Off-The-Shelf (COTS) parking-garage hardware vendors.

The system hardware for our project may be broken into two categories: those we will write software for and those we will simulate.

We will write software to run on user hardware, the garage management terminal, and the system server. The user interface and garage management interface consist of interactive web pages accessible from a modern web browser, communicating with the server through a REST API that we will write. The server software will be written by us using NodeJS and Express.

All physical garage hardware will be simulated by us in a garage simulator. This includes the license plate cameras, entrance and exit terminals, car elevator controls, garage elevator informational display, spot occupancy sensors, exit gate arm, and garage external informational display. The simulator will allow us to demonstrate the functionality of the garage without relying on hardware unavailable to us.

Implementation

To deliver a functioning piece of software before the end of the semester, the project will be divided into two phases matching the two system demonstrations required in this course.

- Phase 1:
    - User Device Interface webpages
        - Account Login
        - Single Online Reservation
        - Guaranteed (Monthly) Online Reservation
    - Garage Management Terminal webpages
        - Garage Configuration
        - Garage Pricing

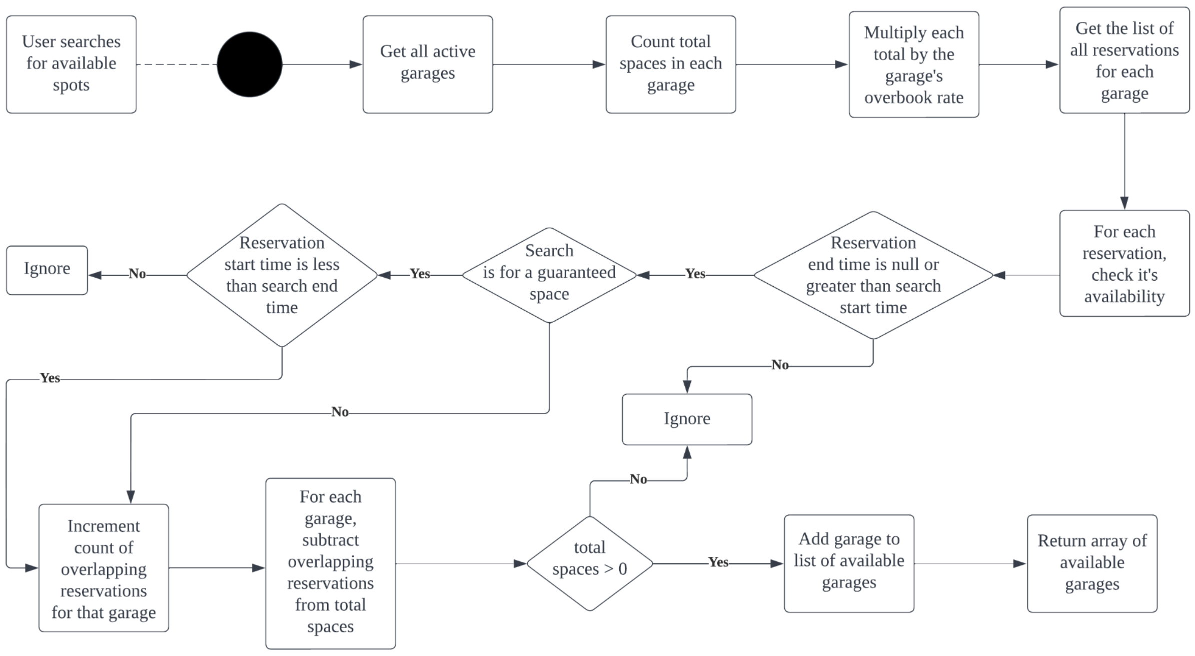
    - System Server
        - Functionality to support the frontend web pages
        - REST API routes to support data exchange for the frontend web pages
- Phase 2:
    - User Device Interface webpages
        - Account Registration
    - Garage Simulator
        - A software simulator allowing reservations to “enter” and “exit” garages.
    - System Server
        - Functionality to support the garage simulator
        - REST API routes to support data exchange for the garage simulator
- Future phases:
    - All remaining use cases
    - Hardware integration

Minimum Requirements	
User Device	
Display	Color; 360px * 640px viewport size
Network	4G mobile data or 10 Mbps
Software	An up-to-date install of one of Google Chrome, Mozilla Firefox, Microsoft Edge, or Apple Safari
Garage Management Terminal	
Display	Color; 768px * 1024px viewport size
Network	4G mobile data or 10 Mbps
Software	An up-to-date install of one of Google Chrome, Mozilla Firefox, Microsoft Edge, or Apple Safari
Garage External Informational Display	
Brightness	5,000 nits

Pixel Pitch	To be determined per installation based on expected viewing distance
Size	
Garage Elevator Informational Display	
Brightness	2,500 nits
Pixel Pitch	To be determined per installation based on expected viewing distance
Size	
Entrance and Exit Terminal Displays	
Input	Touchscreen
Size	20 inches
Resolution	Full HD
System Server	
Storage:	500 GB SSD
RAM:	4GB
Network:	100 Mbps

10 - Algorithms and Data Structures

10.1 - Algorithms


User searches for available spots
Get all active garages
Count total spaces in each garage
Multiply each total by the garage's overbook rate
Get the list of all reservations for each garage
For each reservation, check it's availability
Reservation end time is null or greater than search start time
Yes
No
Ignore
Search is for a guaranteed space
Yes
No
Reservation start time is less than search end time
No
Ignore
Yes
Increment count of overlapping reservations for that garage
For each garage, subtract overlapping reservations from total spaces
total spaces > 0
No
Yes
Add garage to list of available garages
Return array of available garages

When a user searches for a spot, a list of garages with available spots will need to be displayed. Four required search fields will be entered by the user: location, search radius, reservation type, and start date. Additionally, if the reservation is not for a guaranteed space, a start time and end datetime is required. From this information, paired with the saved reservations, the list of garages with available parking will be displayed to the user.

10.2 - Data Structures

As a result of the algorithm mentioned above, an array of garage result objects will be returned for the user to select a garage to create a reservation. There will be multiple single-dimension arrays used throughout this application to pass a list of objects such as reservations and garages.

10.3 - Concurrency

This does not apply as our system is single-threaded.

11 - User Interface Design and Implementation

During the course of building our UI pages have evolved with the idea of making them as easy to use as possible.

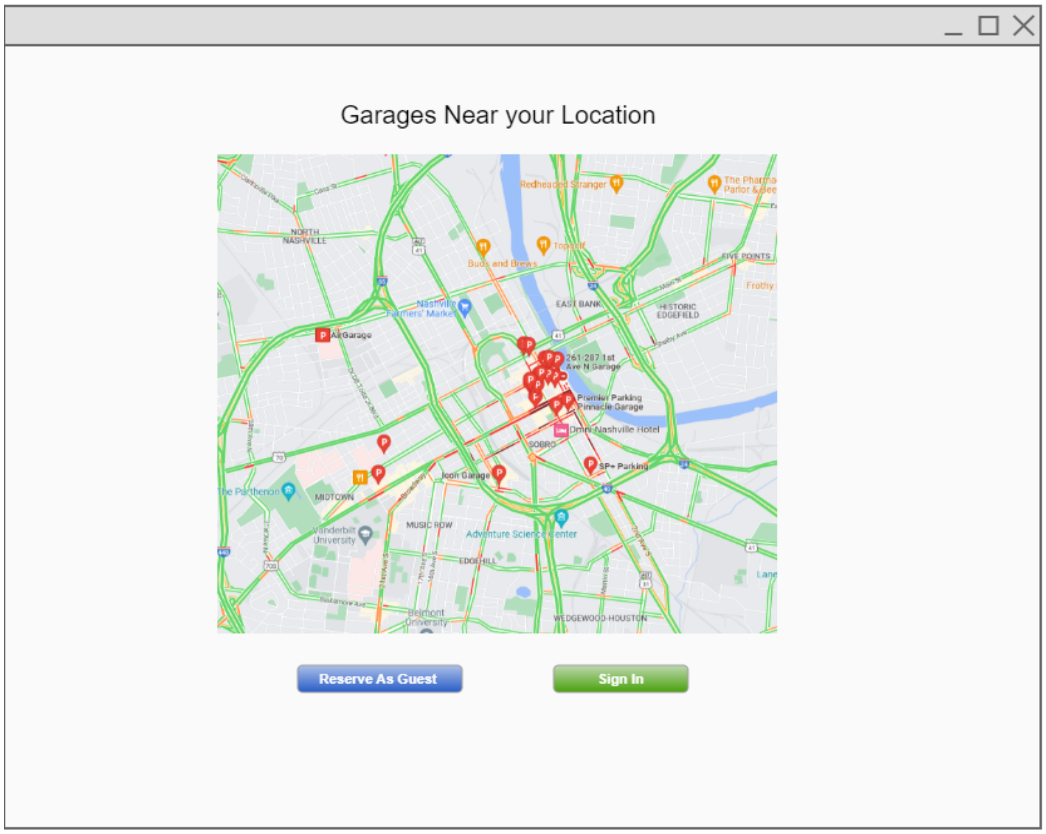
UC-02a — Requesting a Single Parking Spot had some tweaks as to how we had the user selecting a parking spot to reserve. Initially, on the design, we would have a user selecting a Start-End date, and a time frame; then after the user had garages in that location they would have to select a garage and then be taken to a modal to either Sign in or to reserve the spot. With our new design the user still puts in Timeframe but as 1 single input field, and then instead of forcing the user to automatically use their location we allow them to select a location and select how far from that location they are willing to go; or use the new "Use My Location" button to grab their current location. On top of that, once a user makes their search, we display each garage with that garage's price for reserving. This makes the page easier and faster to navigate by keeping the user on a single page, as well as giving them more control over what garages they can choose from by allowing them to select a specific location. We implemented this Locator page as our website's Home page. The reason for this is the most common use case is that someone is trying to reserve a parking spot ahead of time, so it made sense for the garage locator page to be front-facing since it will be the most commonly used page.

UC-02a — Requesting a Permanent/ Guaranteed Parking Spot has been affected the same as the changes mentioned in **UC-02a — Requesting a Single Parking Spot** above. We worked on making the first user interaction with the site easier and required less initial navigation.

Designs for **UC-15 — Adding a Garage to a Customer's Network** and **UC-16 — Adjusting Prices for a Garage** designs have remained the same through our coding and testing. Right now these are the most efficient ways to navigate these Use Cases. The implementation for these pages started with creating and implementing a Login and Registration page, as a user has to be able to sign up as an Operator and must be able to login as said Operator to create a garage or network of garages. Then we are currently in the phase of implementing the ability for an Operator to add and update their set garages.

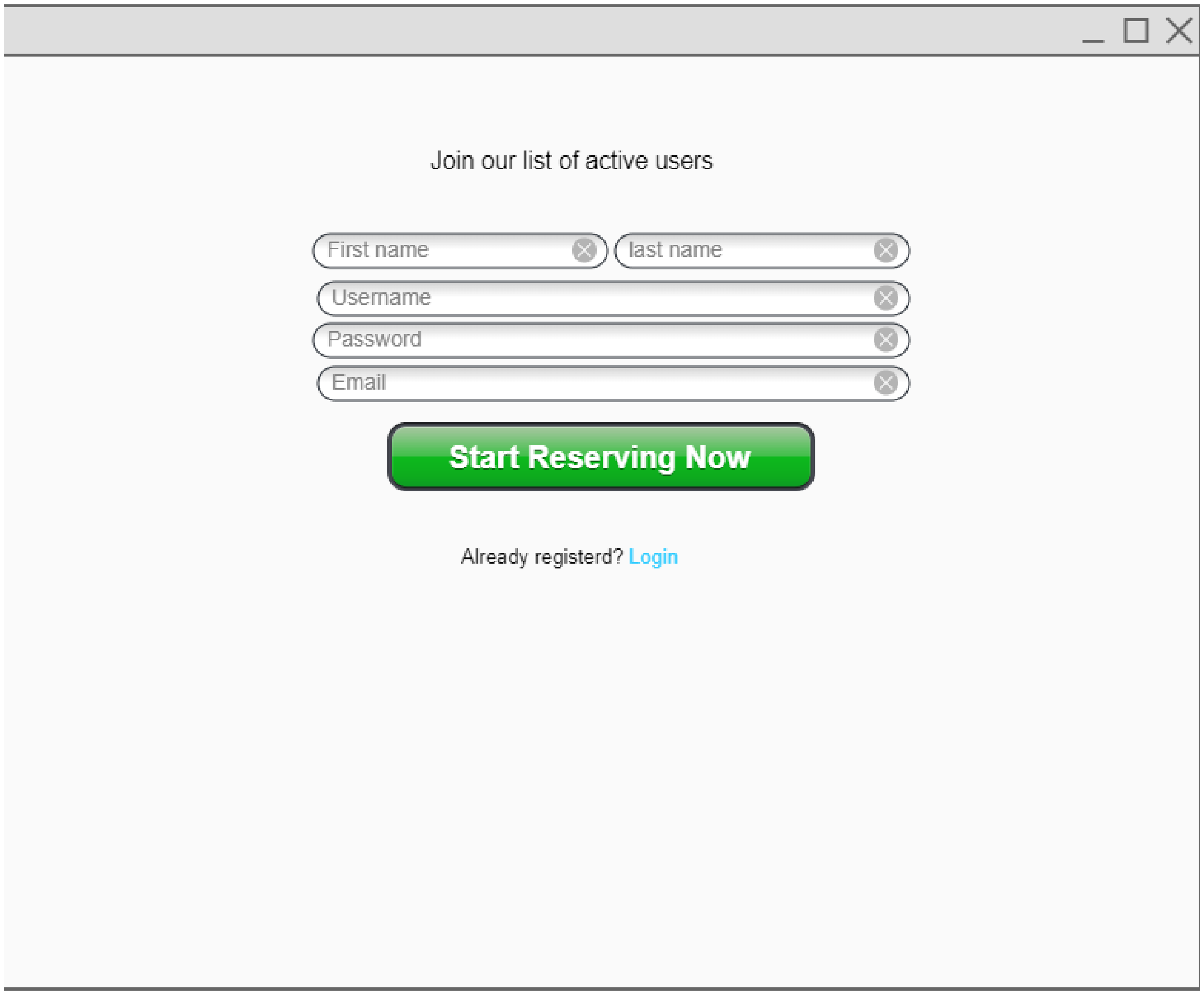
Near Garage Locator Page (Main):

User will be able to locate a garage near them, and select to reserve is a guest, or sign in to reserve.

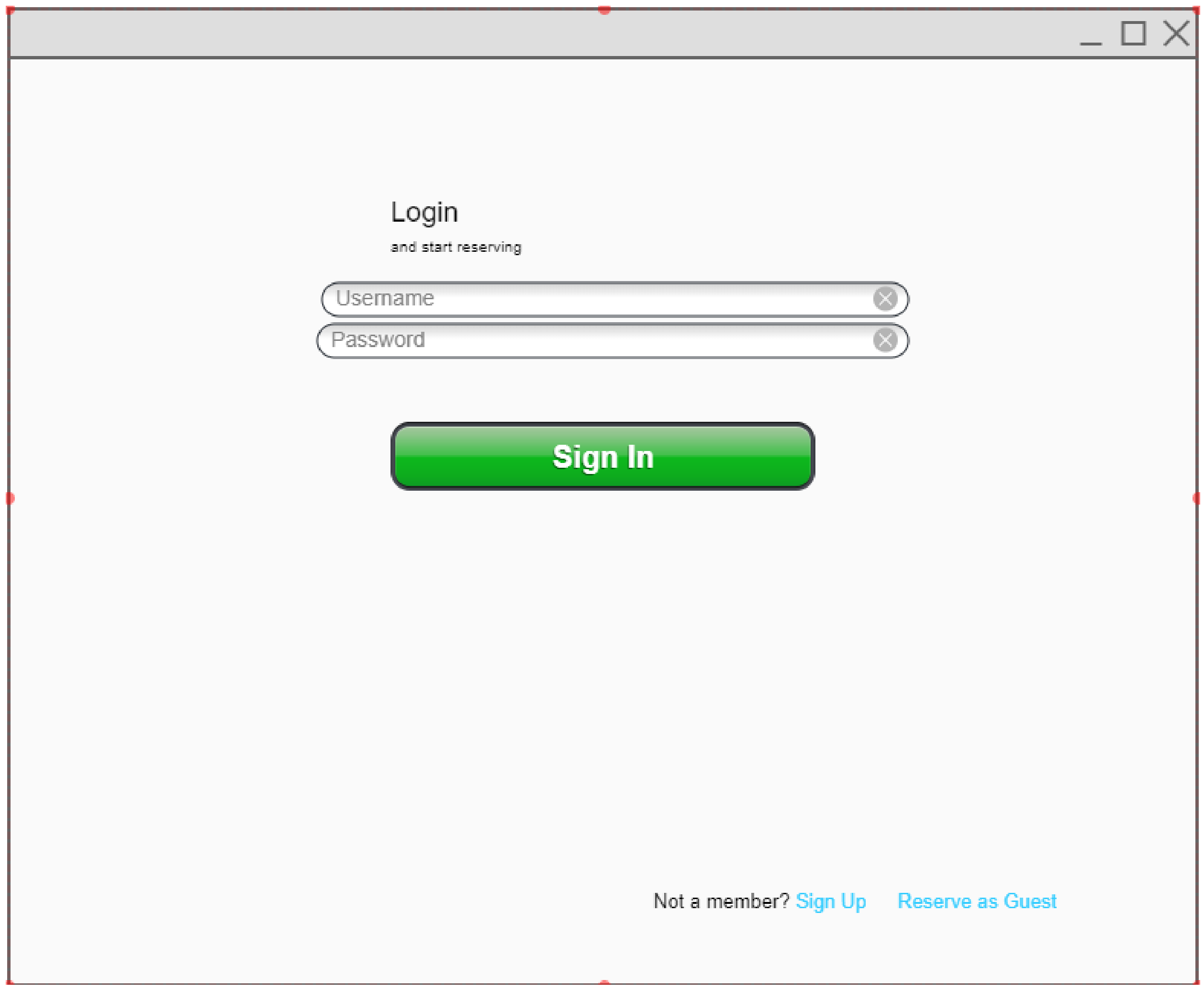


Registration Page:

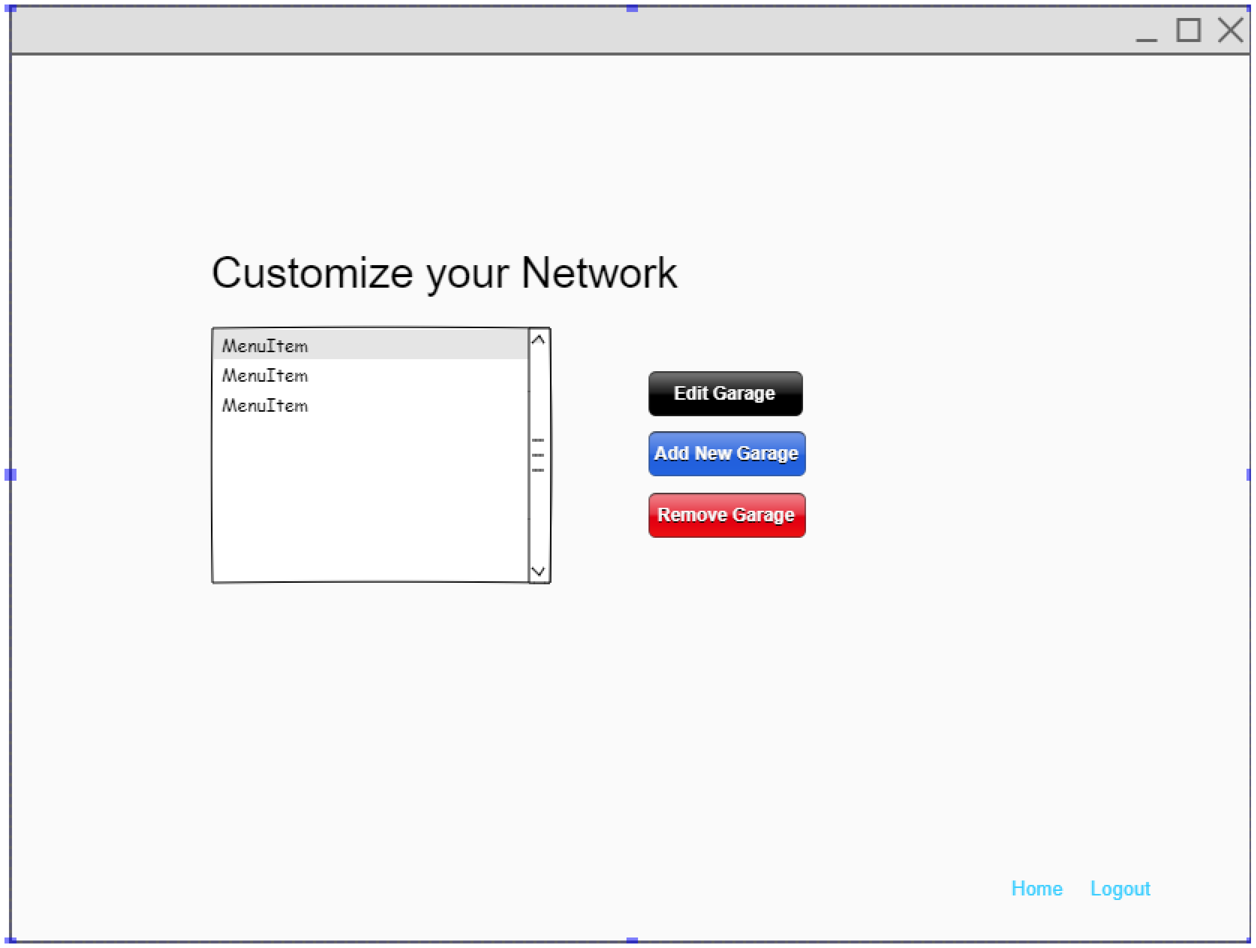
Registration page gives a new user the ability to create a name with the required information.



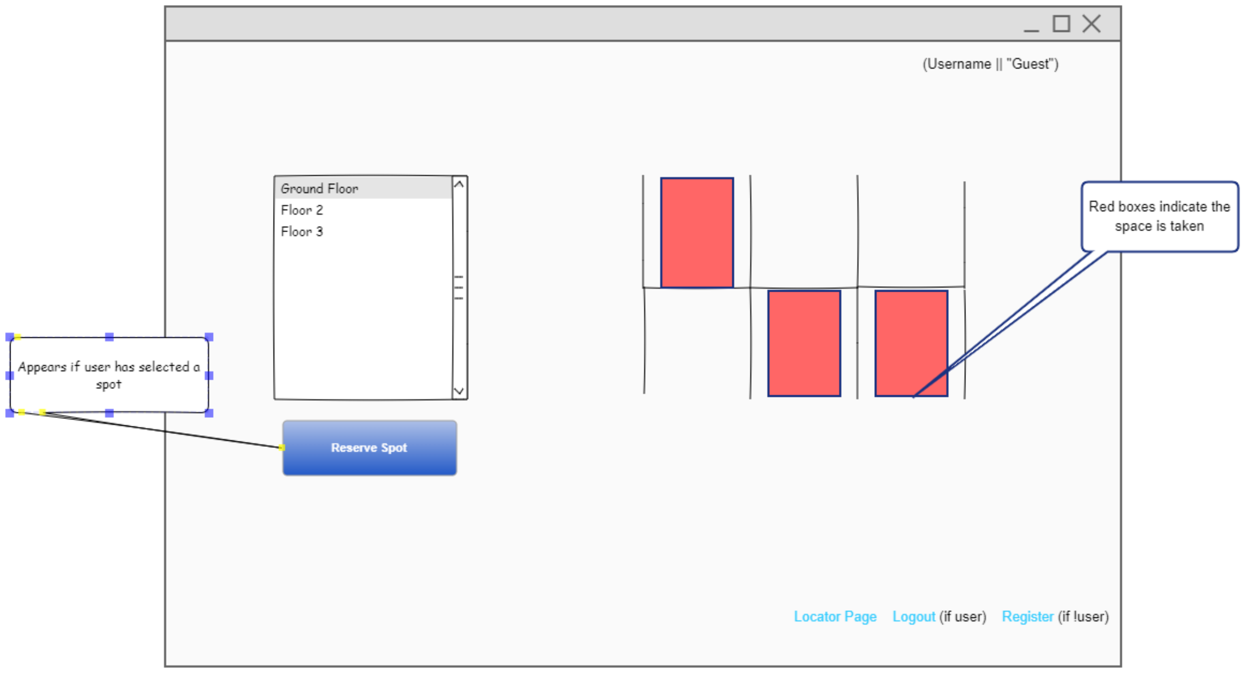
Login Page:


Login
and start reserving
Username
Password
Sign In
Not a member? Sign Up
Reserve as Guest

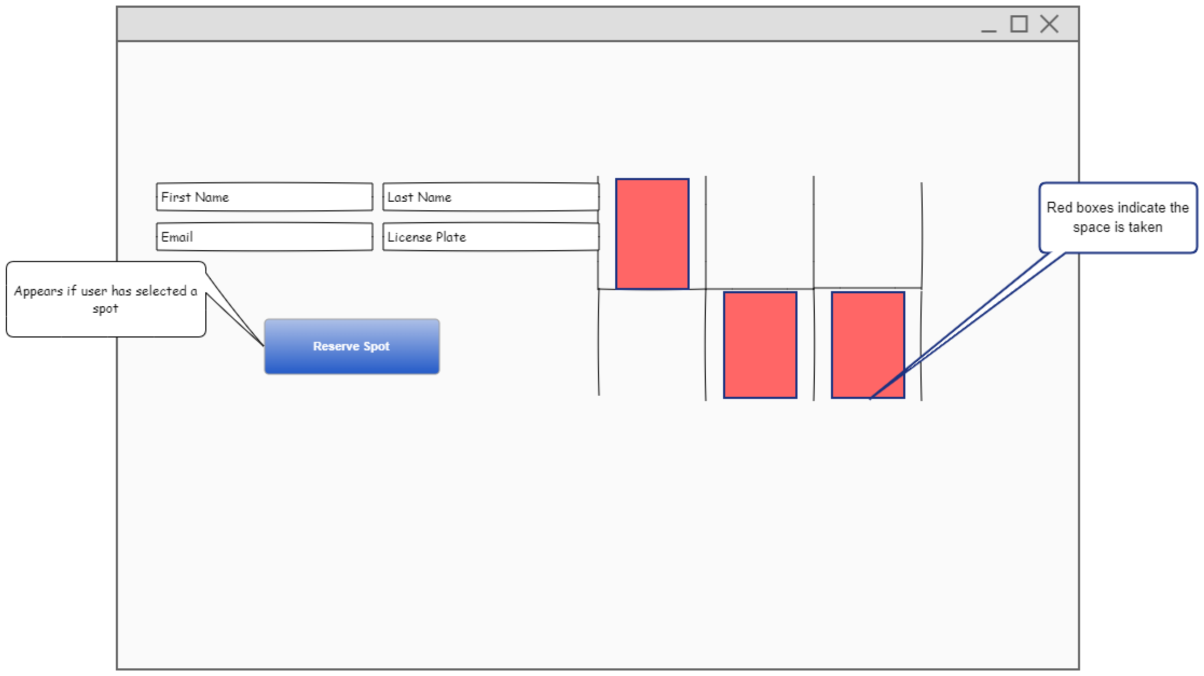
Garage Network Page



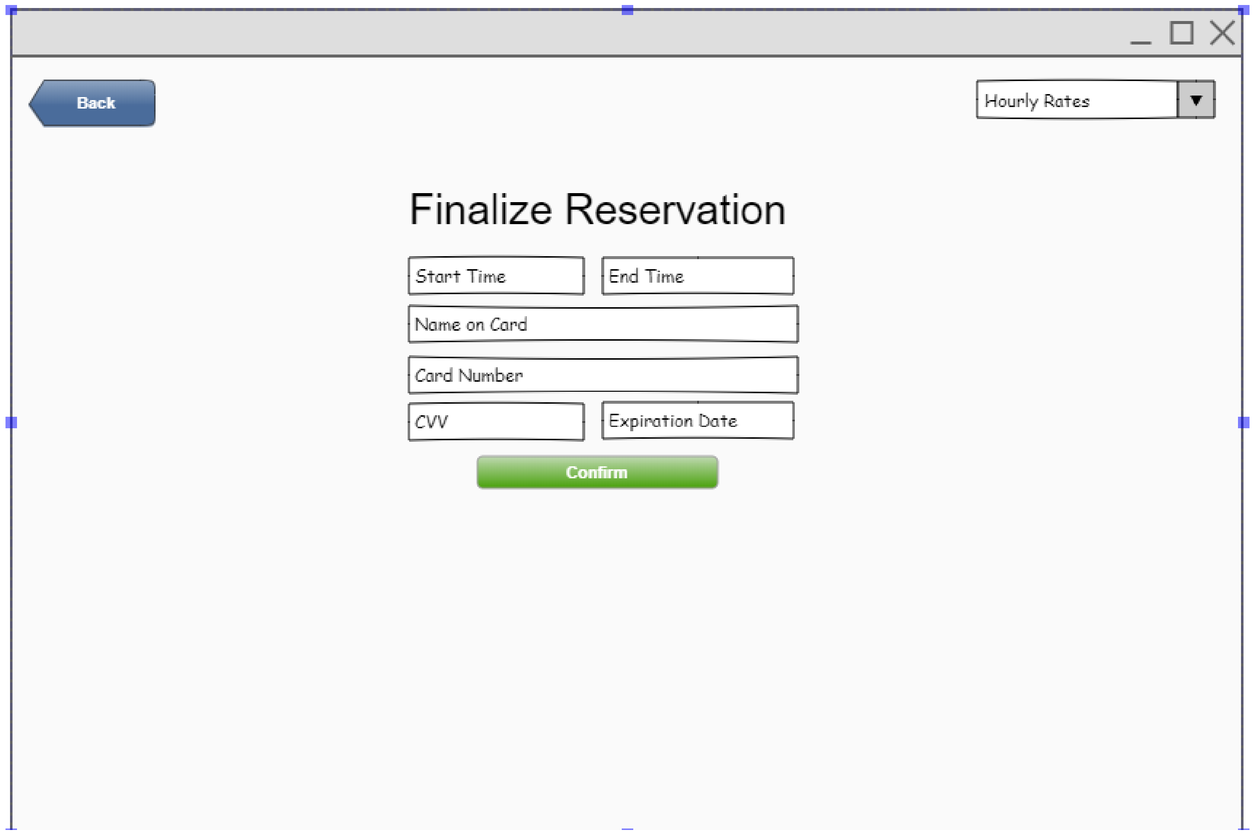
Online Registration Page



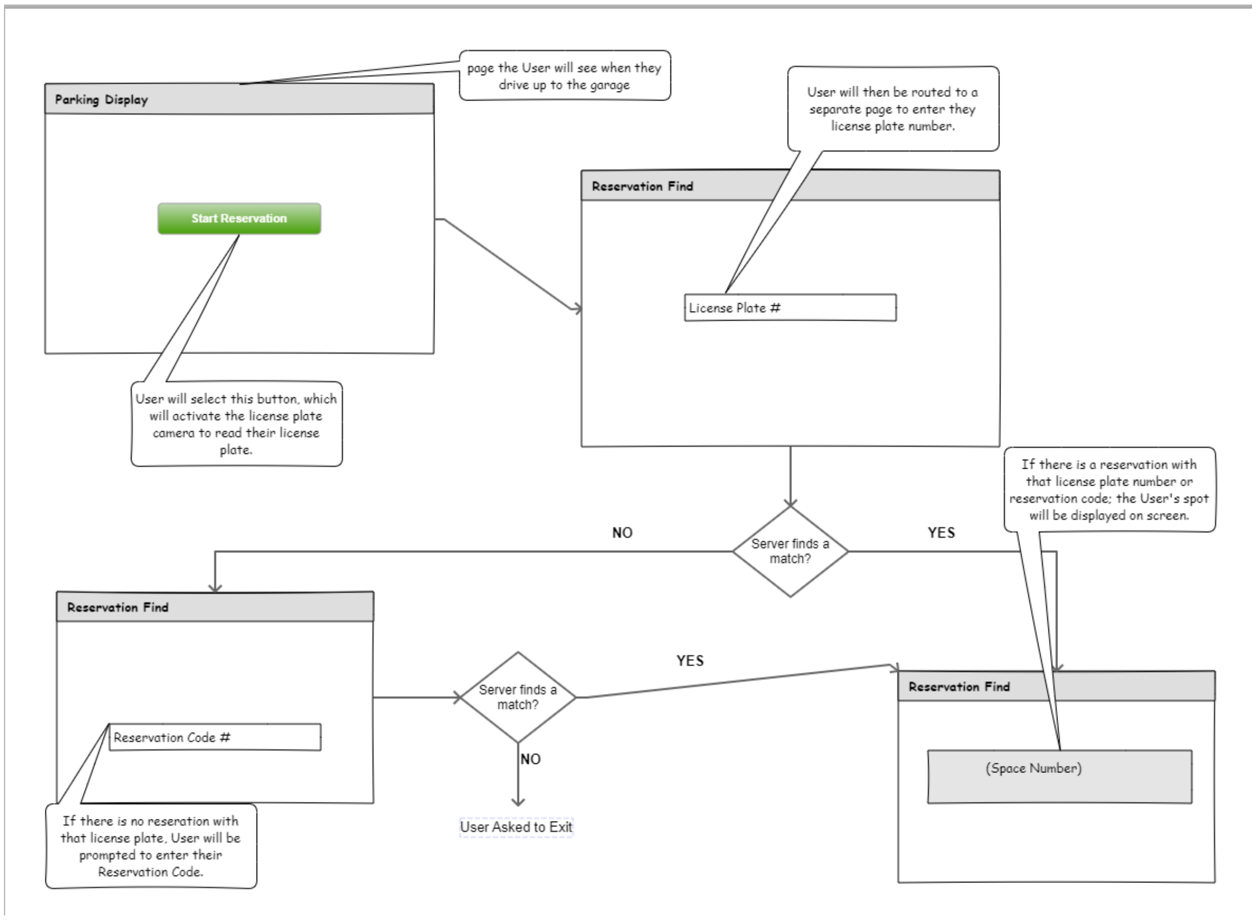
Walk-In Page


First Name
Last Name
Email
License Plate
Red boxes indicate the space is taken
Appears if user has selected a spot
Reserve Spot

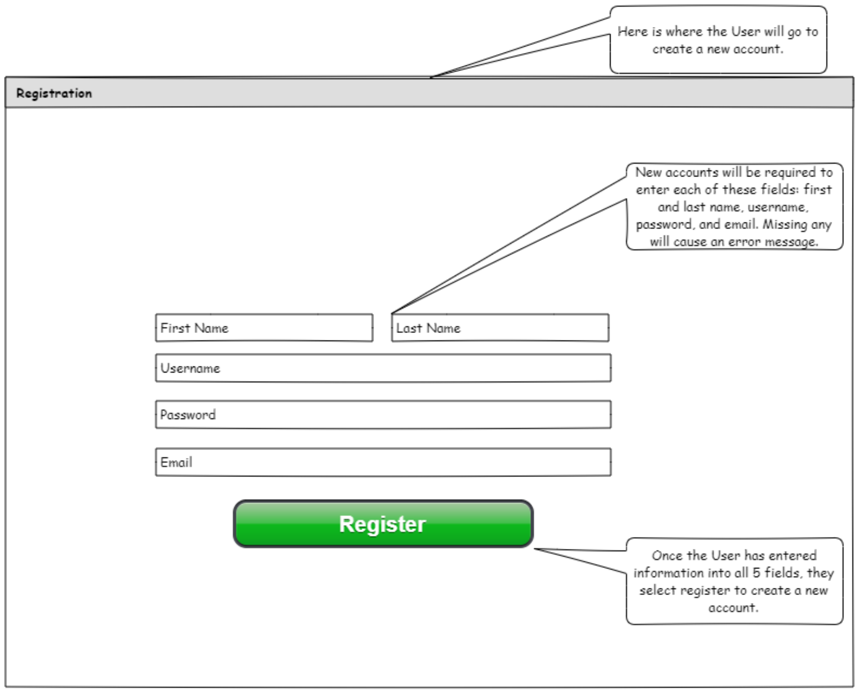
Payment Page



UC-10a:
This UI section will be the main display for the garage when a User drives up to start their reservation.


page the User will see when they drive up to the garage
Parking Display
Start Reservation
User will select this button, which will activate the license plate camera to read their license plate.
User will then be routed to a separate page to enter they license plate number.
Reservation Find
License Plate #
NO
Server finds a match?
YES
If there is a reservation with that license plate number or reservation code; the User's spot will be displayed on screen.
Reservation Find
Reservation Code #
If there is no reseration with that license plate, User will be prompted to enter their Reservation Code.
Server finds a match?
YES
NO
User Asked to Exit
Reservation Find
(Space Number)

UC-01
This webpage will be implemented by adding it to part of the main website at view/register. This will be accessible by any potential new User to create an account.


Here is where the User will go to create a new account.
Registration
New accounts will be required to enter each of these fields: first and last name, username, password, and email. Missing any will cause an error message.
First Name
Last Name
Username
Password
Email
Register
Once the User has entered information into all 5 fields, they select register to create a new account.

---

12 - Design of Tests

12.1 - Test Cases

UC-01 – Registering

- A user must be able to register with the service to make reservations. Registration requires a unique username, unique email address, and password. Once registered, a user can add vehicle and payment information.

- Testing:
    - Edge case: Input fields will be required, therefore, attempts to submit a blank form or an incorrect email address should trigger a notification to the user.

    - We will attempt to register using an existing username. We will repeat this test with a unique username and existing email address. Both should reject the registration and prompt correction of the existing detail.
    - Next, we test with a unique username and email address, this should allow a successful registration. A User constructor generates a new user object passed to the database. We can confirm registration with a database query using the User class Constructor (username, password) which returns an object with the user's information.

UC-02a/b – Reservations (single/guaranteed)

- A user can search for a parking space by location and/or date and receive a list of available spots based on their criteria. A user can request a spot and make up to 3 non-overlapping reservations if they have a card on file. A user can also make a guaranteed reservation for a permanent spot or over a user-specified period.

- Testing:
    - Edge case: Invalid dates and times should prompt correction. Enter a start date or time before the current.
    - Enter a valid reservation start and end time and reservation type. This is passed parameters to the Available class which invokes the search() function and returns an array of available garages given the arguments passed in. We select a garage, the garage object is passed as a parameter to the Reservation class constructor, creating a reservation in the database. We verify the reservation by making a database query constructing a ReservationStatus object containing the details of that reservation.

UC 10a - Reservation Arrivals

- A user with a valid reservation arrives at the garage, the reservation is retrieved either by one of two options. First, the system scans the vehicle license plate and attempts to match an existing reservation to that plate number. If a match is not found, the user will be prompted to enter their reservation number on the display, and the reservation is retrieved.

- Testing:
    - Edge case: We pass a non-existing license plate and receive a message that no such plate number could be located. We pass a non-existing reservation number and receive a message that the reservation does not exist

    - Enter a valid license plate number with an existing reservation. When the reservation details are located in the DB, the elevator activates taking the user to the assigned floor. We also test against a reservation number in the event that the license plate reader fails. Once a valid reservation number is passed, the corresponding reservation is retrieved from DB and the elevator activates to allow access to the assigned floor.

UC 11 - Reservation Management

- A reservation will be marked complete when the user triggers the exit and leaves the garage.
- Testing:
    - Use function stubs to simulate a vehicle exiting the garage. This records the exit time and marks the reservation as complete. We verify the changes by querying the DB for this reservation.

UC 15- Add garage to network

- Garage Configuration - A garage operator must be able to add garages to their network and define the number of spots on each floor, activate/deactivate the garage, and set the overbooking rate. Duplicates cannot be added.

- Testing:
    - Edge case: Pass an existing garage ID or address, should return an error message stating that the garage already exists.
    - We test that a new garage is added to the database. We authenticate the user as a garage when the getIsOperator() method in the User class returns true. Next, we pass the set of attributes to a Garage constructor to pass a new garage object to the database. We verify by calling the allGarges() method in the Garage class which returns an array of all garages in the database. The newly added garage should be in the returned array.

UC 16 - Price Adjustment

- A garage operator must be able to set prices and fees for reservations, walk-ins, and guaranteed reservations.

- Testing:
    - Edge case: Price cannot be set to a negative value. A negative value should return an error message. We test this by passing a negative integer value to the Pricing constructor in the Pricing class.
    - Next, we pass an acceptable price value to the constructor to generate a new price. We verify by invoking the getCost() method which should return the price we just set.

12.2 - Test Coverage

We will use equivalence testing for our test coverage. We divide our equivalence classes into one valid and one invalid equivalence class for each set of values passed to each method or constructor. The invalid class being what we expect to be edge cases, and the valid class being the acceptable set of values. This black box method of testing allows us to pass parameters and test for the outcome that we expect.

Because we expect each class method to behave the same when acceptable values are passed, there is no need to test an infinite number of potential parameters values. We conduct a few tests by varying the expected parameters and verifying the results via database queries. We can vary the inputs in our driver program to ensure the code is reusable and reliable.

Our approach will be to test each edge case and confirm the expected outcome. The edge cases are those that should return an error message or prevent the user from continuing due to unacceptable values being passed to a method. We can test this behavior by running only a few tests since we expect that every unacceptable value is treated the same.

12.3 - Integration Testing Strategy

We will use the “bottom-up” integration testing strategy for our system. The leaves in the hierarchy are those classes which have no dependencies and the user interface pages, which will be tested with unit tests. The next level above the base classes are the composite classes which depend on them, which will then be tested with the base classes. Above the base and composite classes are the controllers which provide functions to the routes and which will be tested using the classes below them. Finally, above the controllers and the user interface webpages are the routes, which define the REST API that coordinates communication between the front and back end of the system. These routes will be tested using the full available set of tested classes and pages, completing the integration testing.

The system will be built with NodeJS [About NodeJs] and ExpressJS [ExpressJS FAQ]. NodeJS is a JavaScript runtime environment, allowing JavaScript to run outside a browser. ExpressJS is a web framework for NodeJS, allowing for the creation of web servers in NodeJS. The database will be a cloud-hosted instance of IBM's DB2 [DB2 On Cloud], which will be managed through Sequelize [Sequelize] models connecting the database tables to corresponding system classes. On the front end, AngularJS [AngularJS] will be used to develop responsive user interface pages. Testing will be completed using the Jest Javascript Testing Framework [Jest] in combination with the

Supertest package [Supertest]. For more information, the web page for each tool may be consulted by following the citation to the references section of this report.

12.4 - Non-Functional Requirement Tests

Non-Functional Requirement Tests			
REQ	**Description**	**Test Case**	**Expected Result**
Functional			
REQ-68	The elevator entrance and exit will have operational cameras to detect license plates.	Poll the cameras to detect whether they are operational.	Camera is polled successfully.
REQ-69	The elevator will rotate floors when possible to ensure the user with the reservation has enough time to get to their parking spot	Send a series of mock vehicles through the garage simulator and track which floor each is sent to.	Each vehicle is sent to the lowest non-busy floor or the lowest least-busy floor.
REQ-70	An external image processing system local to the garage will process the images captured into a payload to be sent to the system via API	Load an image of a license plate into the processing system and wait for the payload to be sent to the garage system.	License plate data is transmitted to the system.
REQ-71	Each Spot will have a light reactive sensor to detect occupancy of that spot	Physically examine the placement of spot sensors in the garage.	Every parking space has a sensor installed.
REQ-72	Spot sensors and cameras will be assumed to be operational at all times	Poll all cameras and sensors to detect whether they are operational.	All cameras and sensors are successfully polled.
REQ-73	Displays will be connected to the software via internet connection	Attempt to connect to the system from a display.	Connection successful.
REQ-74	Consoles will be touch screen so users can select multiple options and type in their reservation/member numbers	Physically examine all consoles.	All consoles have working touchscreens.

Usability			
REQ-75	The interface must be clean and easy to use without instruction	Observe a sample of individuals using the interface.	Most individuals can use the system with no assistance or instructions.
REQ-76	Required fields will be marked and error messages will be clearly displayed so mistakes can be corrected by the user	Examine the UI for markers and errors.	All required fields are marked and display error messages.
REQ-77	In order to use the system, an automatic payment method must be set up by the user, and contact methods must be provided	Attempt to use the system without setting a payment method, without setting contact information, and without both.	All attempts are refused.
REQ-78	The spots will be assigned when the user arrives, therefore in order to avoid congestion the algorithm must prioritize efficiency	Send a series of mock vehicles through the garage simulator and track the spaces assigned to each vehicle.	All assigned spaces prioritze closeness to elevator, while sending active vehicles to non-busy floors.
Reliability			
REQ-79	The system must be implemented on a stable server	Monitor up- and downtime of the system.	System exhibits >= 99.9% availability.
REQ-80	Managed Services (if chosen by client) will provide credits for any downtimes	Compare records of downtime and credits provided.	Each downtime record has a matching credit record.
REQ-81	5G connectivity is suggested for mobile users	Sample connection data by connection speed.	A majority of connections have a speed >= 5G speed.
Performance			
REQ-82	Algorithms must be efficient in order to limit wait time for users	Observe a sample of individuals using the system and measure the rate of users who quit or exhibit impatience.	A majority of individuals complete their actions without displaying impatience.
REQ-83	Database updates will be efficient and prioritize	Measure DB update time versus complete request	DB update time >= 0.5 * request time.

	updates that need to be reflected on the UI immediately	time.	
Supportability			
REQ-84	The system must be usable on mobile device, computer and tablet via web browser	Physically test the system from a variety of physical devices.	The system is functional on all modern hardware.
REQ-85	Major web browsers such as Chrome, Firefox, and Safari must be supported	Physically test the system from a variety of browsers.	The system is functional in all modern browsers.
REQ-86	Disability Accommodations must be made in the web design	Physically check the system for appropriate font sizes, tap targets, color contrast, and aria labels.	All font sizes, tap target sizes, color contrasts, and labels meet accessibility guidelines.

12.5 - User Interface Requirement Tests

User Interface Requirement Tests			
REQ	**Description**	**Test Case**	**Expected Result**
REQ-87	Users will be able to find a garage to book near them.	Attempt a reservation in the demonstration region.	Results are displayed.
REQ-88	New user account creation.	Attempt to create a new user account.	Account creation successful.
REQ-89	Users will access their account to register / alter registrations/ view payment history.	Attempt to add a payment method, change it, and view payment history.	Payment method added and modified successfully. Payment history is available.
REQ-90	Owners will be able to add new garages to their network(s).	Attempt to add a new garage to the network.	Garage is added successfully.
REQ-91	Users will be able to	Attempt to reserve a	Reservation is successful.

	reserve their spot online.	space online.	
REQ-92	Walk-in user will be able to reserve a ground floor spot	Attempt to make a walk-in reservation.	Reservation is successful.
REQ-93	Users enter their payment option securely.	Check that payment information is transmitted over a secure connection.	Connection is secure.

13 - History of Work, Current Status, and Future Work

Project Target	Tasks	Description	Contributors	End Date	Technologies	Deadline Evolution
Main UI/UX	Register Page	User's will be able to register with designated information.	Justin Henley, Victor Munoz	9/12/22 - 10/02/22 -old 12/2/22	***UI/UX*** HTML, CSS, Bootstrap4, Javascript, AngularJs ***Backend*** C#,Express, NodeJs, PGSQL	All of the deadlines for this section were met, except the Register page. The fully functional Register page was opted to be completed by the 2nd Demo.
	Local Garage Locator Page	Starting page, user able to locate nearby garages and select one for booking	Dustin Threet, Hannah Carr	9/12/22 - 10/02/22		
	Sign In Page	Login Page	Justin Henley, Victor Munoz	9/12/22 - 10/02/22		
	Walk-In Display	Display for walk-ins with	Dustin Threet,	9/12/22 - 10/02/22		

		available spots for them.	Justin Henley			
	Database Diagram	Basic structure for entities that will be stored in the DB	Hannah Carr	9/12/22 - 10/02/22	PGSQL	Met Deadline
1st Report : Part 2	Functional Requirements		Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	9/18/22	N/A	Met Deadline
Full Report 1			Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	9/25/22	N/A	Met Deadline
Garage Operator	Garage Operator After-Sign In	Owner user will be directed here after sign-in	Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	10/23/22 - old 11/1/22 - actual deadline	***UI/UX*** HTML, CSS, Bootstrap4, Javascript, AngularJs	This deadline was altered and pushed back due to there being some bugs in the garageController.
2nd Report: Part 1	Use cases/ User interactions		Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	10/2/22	N/A	Met Deadline
Garage Setter	Add new garage	Owner will be able to add a garage to network	Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	10/23/22 - old 10/29/22 - actual deadline	***Backend*** C#,Express, NodeJs, PGSQL	This deadline was pushed back for further debugging. Code was done before the deadline, but after further inspection realized it had issues such as

						fetching the list of garages.
2nd Report: Part 2	System architecture		Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	10/9/22	N/A	Met Deadline
Modify Reserve	Alter reservations	User will be able to change, cancel, and extend reservations	Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	10/23/22 - old 10/28/22 - new	***UI/UX*** HTML, CSS, Bootstrap4, Javascript, AngularJs ***Backend*** C#,Express, NodeJs, PGSQL	Altering reservations deadline was also pushed back due to there being some bugs with garageControlle r, as well as losing some database functionality.
Full Report 2			Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	10/16/22	N/A	Met Deadline
GPS-based Locator	Find garage	Locate garage based on user input	Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	10/23/22	***UI/UX*** HTML, CSS, Bootstrap4, Javascript, AngularJs ***Backend*** C#,Express, NodeJs, PGSQL	Met Deadline
First Demo		Combine project to present		10/30/22	Project	Met Deadline
Autopay	Alter autopay	See user's payment history, change payment options/ occurrence	Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	11/12/22 - old TBD - Future Feature	***Backend*** C#,Express, NodeJs, PGSQL	Altering the autopay has been put on a hold due to the developers working on more usable features for Demo 2, such

						as handling reservations and a user registering a new account from the UI
Handling Reservation (Start -> Exit) UC-10a & UC-11	Handle finding reservations at the garage. Handle users leaving after reservation time.	Users will have ability to claim their reservation at the gate. Display spot number for user. Handle user exiting, updating the spot status	Justin Henley, Victor Munoz, Dustin Threet	12/?/22 - Demo 2 Date	***UI/UX*** HTML, CSS, Bootstrap4, Javascript, AngularJs ***Backend*** C#,Express, NodeJs, PGSQL	Handling reservations from user arrival to their exit was added after the 1st Demo, as it was not properly documented clearly in our plan of work originally.
3rd Report: Part 1			Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	11/13/22	N/A	Met Deadline
Full Report 3			Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	11/20/22	N/A	TBD
Second Demo		Combine project to present		12/24/22	Project	TBD

Key Accomplishments

- Full Operator and User sign in
  - User can fully sign in
  - Operator(s) can sign in and will be redirected to their Operator page.
- Fully operational Operator page to manage and adjust pricing and garages.
  - Operators can change pricing based on reservation type.
  - Operators can edit or add garages
    - This includes altering number of floors, spots, active (true/false)
    - Includes adding a whole new garage
- Fully operational Locator page

    - User can change location to find a garage
    - Timezone conversion so User knows what the real time the reservation will be in.
    - Reserve their spot with the locator page
- All relevant models for each component complete in the database
- Database migration from local machine to a cloud-based PGSQL database
- Registering a new account (database side only)
    - UI portion of this is still incomplete, but can happen with direct HTTP request

Project Moving Forward

Going forward with this project the first direction we are going to take leading into Demo 2 is going to be: fully flesh out the reservation arrival (customer arriving -> parking -> exiting garage) and a user registering for a new account from the UI. The reason we chose this direction is this will cover all of the smaller bases. In short, this means an Operator will be able to log in and add/ edit their garages, and Users will be able to register, log in or out, locate a garage nearby, reserve a spot, claim their reservation, and leave the garage. This particular direction was chosen because it would give the full experience of parking in or operating the garage.

Another direction that could have been chosen would be refining the user login and accounting editing and setting up the payment backend. This would be another great option as this would be how the garage would make its money. However, this idea was pushed off as a future feature due to the team wanting to have the full start-to-finish product ready before adding the payment options.

14 - References

About NodeJS (no date) *NodeJS*. Available at: https://nodejs.org/en/about/ (Accessed: October 15, 2022).

AngularJS (no date) *AngularJS*. Available at: https://angularjs.org/ (Accessed: October 15, 2022).

Dhaduk, H., 2022. Node.js vs. express: Determining the best backend technology. Simform. Available at: https://www.simform.com/blog/node-js-vs-express/ [Accessed September 22, 2022].

ExpressJS, 2022. Express. npm. Available at: https://www.npmjs.com/package/express [Accessed September 23, 2022].

ExpressJS FAQ (no date) *ExpressJS*. Available at: https://expressjs.com/en/starter/faq.html (Accessed: October 15, 2022).

Fastify.io, 2022. Fastify. npm. Available at: https://www.npmjs.com/package/fastify [Accessed September 23, 2022].

Hoehne, C.G. et al., 2019. Valley of the sun-drenched parking space: The growth, extent, and implications of parking infrastructure in Phoenix. *Cities*, 89, pp.186–198. Available at: https://www.sciencedirect.com/science/article/abs/pii/S0264275118311636 [Accessed September 6, 2022].

Jest (no date) *JestJS*. Available at: https://jestjs.io/ (Accessed: October 15, 2022).

MVC, MDN - Web Docs. Available at: https://developer.mozilla.org/en-US/docs/Glossary/MVC [Accessed September 24, 2022]

Parkopedia, *Parkopedia.com* Available at: https://en.parkopedia.com/ [Accessed September 3, 2022].

Sequelize (no date) *Sequelize*. Available at: https://sequelize.org/ (Accessed: October 15, 2022).

SP Plus Corporation, *Parking.com*. Available at: https://parking.com/ [Accessed September 3, 2022].

SpotHero, *SpotHero.com*. Available at: https://spothero.com/ [Accessed September 3, 2022].

Supertest (no date) *npm*. Available at: https://www.npmjs.com/package/supertest#about (Accessed: October 15, 2022).