Fort Hays State University

CSCI 441 - Software Engineering

Group 10

Network of Automated Parking Garages

Report Two

October 16, 2022

Team Member:	*Email:*
Dustin Threet	dlthreet@mail.fhsu.edu
Hannah Carr	hmcarr@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	Hannah Carr	Justin Henley	Victor Munoz
Project Management	25%	25%	25%	25%
Section 1	25%	25%	25%	25%
Section 2	25%	25%	25%	25%
Section 3	25%	25%	25%	25%
Section 4	25%	25%	25%	25%
Section 5	25%	25%	25%	25%
Section 6	25%	25%	25%	25%
Section 7	25%	25%	25%	25%
Section 8	25%	25%	25%	25%

Table of Contents

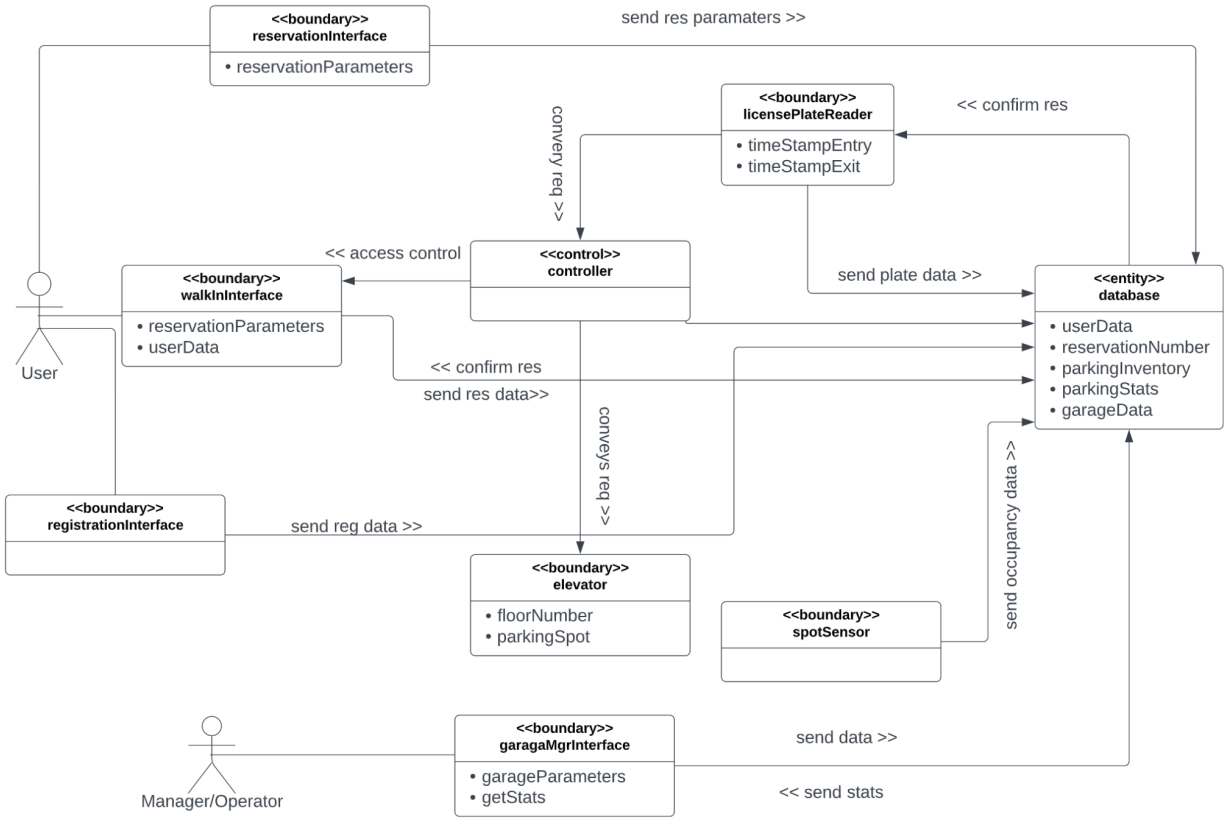
Work Assignment

Given that our group has 4 members, we have refrained from dividing our team “into 3 or 4 sub-teams.” We prefer instead to divide flexibly on a per-task basis with consideration for workload balance and individual competences.

Name	Competences	Work Assignment
Dustin Threet	Software Development, JavaScript, NodeJS, AngularJS, Bootstrap, HTML, CSS	Local Garage Locator Page, Walk-In Display, Customer Account Registration
Hannah Carr	Software Development, Database design, SQL, C#, Java, HTML, CSS, Python	Local Garage Locator Page, Database Diagram, Data Models, Operators Adding a Garage to a Network of Garages
Justin Henley	REST APIs, JavaScript, NodeJS, ReactJS, HTML, CSS	Register Page, Sign In Page, Walk-In Display, Reservations & Walk-Ins
Victor Munoz	Business Management, C++, Java, JavaScript, HTML, CSS	Register Page, Sign-In Page, Location-based Garage Finder, Pricing

1 - Analysis and Domain Modeling

1.1 - Conceptual Model



1.1.1 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.

1.1.2 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

1.1.3 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

1.1.4 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

1.1.5 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	**P W**	**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

1.2 - System Operation Contracts

UC-2a - Requesting a Single Parking Spot

Operation	SearchSpace(location, startDateTime, endDateTime)

Preconditions	• set of valid garages known to the system is not empty • startDateTime < endDateTime • startDateTime >= current datetime
Postconditions	None

Operation	ReserveSpace(garageId, startDateTime, endDateTime, customerId, vehicle)
Preconditions	• garageId is not null • startDateTime < endDateTime • startDateTime >= current datetime • customerId is not null
Postconditions	• A reservation is created in the system matching the given characteristics

---

UC-2b - Requesting a Permanent/Guaranteed Spot

Operation	SearchGuaranteedSpace(location, startDate, endDate, startTime, endTime, frequency)
Preconditions	• set of valid garages known to the system is not empty • startDate < endDate • startTime < endTime • startDate >= current date • startTime >= current time • frequency is one of Daily, Weekly, or Monthly
Postconditions	None

Operation	ReserveGuaranteedSpace(garageId, startDate, endDate, startTime, endTime, frequency, customerId, vehicle)
Preconditions	• garageId is not null • startDate < endDate • startTime < endTime • startDate >= current date • startTime >= current time

	● frequency is one of Daily, Weekly, or Monthly ● customerId is not null
Postconditions	● A reservation is created in the system matching the given characteristics

---

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

Operation	ListGarages()
Preconditions	None
Postconditions	None

Operation	addGrage(name, location, numFloors, spotsPerFloor[], overbookRate, isActive)
Preconditions	● numFloors >= 1 ● Every value of spotsPerFloor is >= 0 ● overbookRate >= 100%
Postconditions	● A new garage is created in the system with the given characteristics

Operation	UpdateGarage(garageId, name, numFloors, spotsPerFloor[], overbookRate, isActive)
Preconditions	● garageId is not null ● numFloors >= 1 ● Every value of spotsPerFloor is >= 0 ● overbookRate >= 100%
Postconditions	● The specified garage is updated in the database to reflect the given characteristics

---

UC-16 - Adjusting Prices for a Garage

Operation	GetPricing(garageId)
Preconditions	● set of valid garage IDs known to the system is not empty ● gargageId is not null
Postconditions	None

Operation	UpdatePricing(garageId, priceType, newPrice)
Preconditions	● priceType is a valid value within the system ● newPrice is a money value
Postconditions	● The specified price category of the specified garage is updated to the new price in the database

1.3 - Data Model and Persistent Data Storage

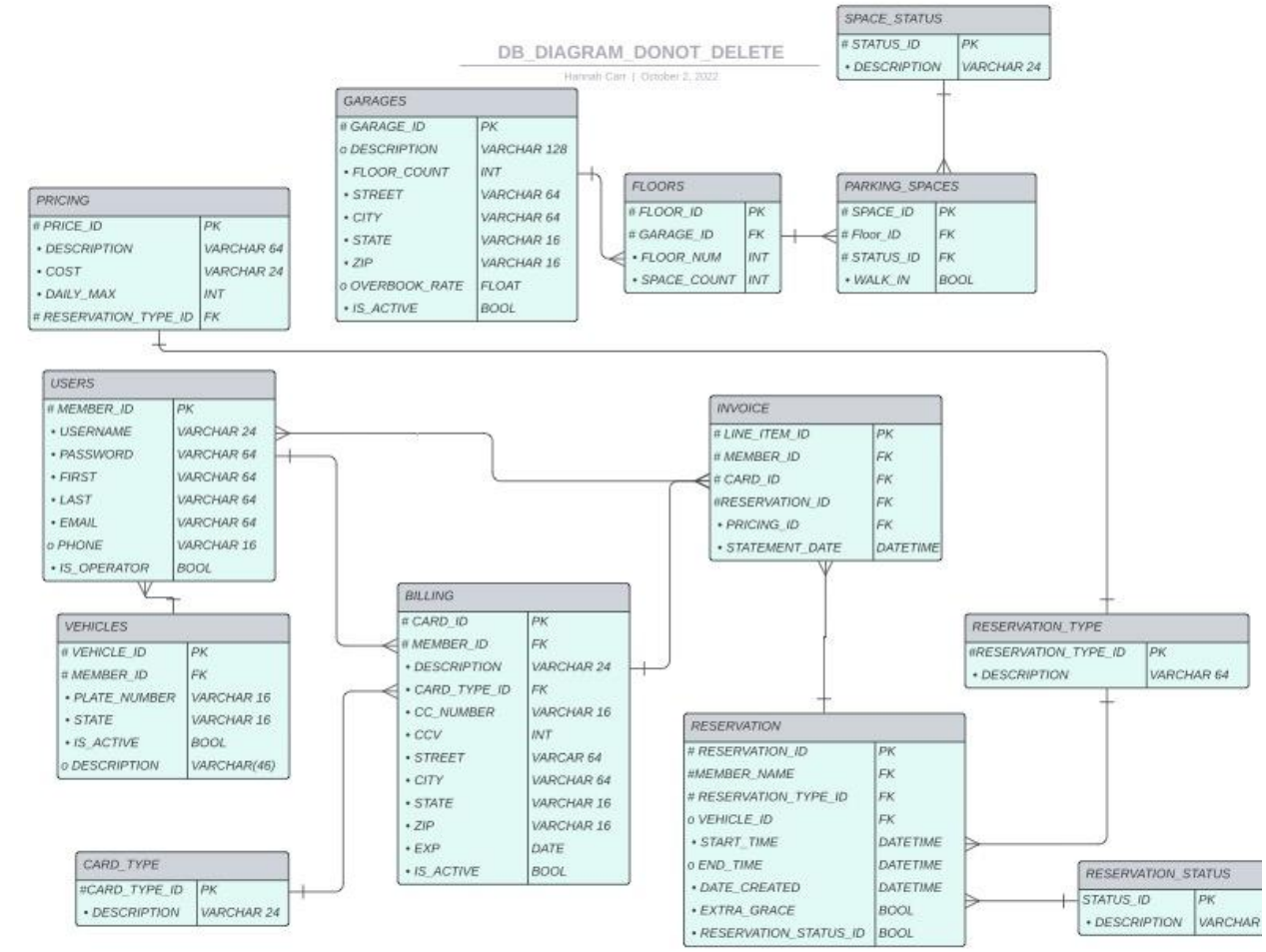
Many pieces of our application will require persistent data storage. Some examples are: User account information, Vehicle Information, Billing Information, Garage Configuration and Reservation information. A relational database using DB2 hosted on IBM Cloud will be integrated with the application. The following table displays what is outputted by the Describe command for each table:

Table Descriptions within the YHL46872 Schema					
Table	Name	Type	Is Nullable	Length	Scale
USERS	MEMBER_ID	INTEGER	N	NULL	0
	USERNAME	VARCHAR	N	24	0
	PW	VARCHAR	N	64	0
	FIRST_NAME	VARCHAR	N	64	0
	LAST_NAME	VARCHAR	N	64	0
	EMAIL	VARCHAR	N	64	0

	PHONE	VARCHAR	N	16	0
	IS_OPERATOR	BOOLEAN	N	1	0
VEHICLES	VEHICLE_ID	INTEGER	N	NULL	0
	MEMBER_ID	INTEGER	N	NULL	0
	DESCRIPTION	VARCHAR	Y	64	0
	PLATE_NUMBER	VARCHAR	N	16	0
	PLATE_STATE	VARCHAR	N	16	0
	IS_ACTIVE	BOOLEAN	N	1	0
GARAGES	GARAGE_ID	INTEGER	N	NULL	0
	MEMBER_ID	INTEGER	N	NULL	0
	DESCRIPTION	INTEGER	Y	128	0
	FLOOR COUNT	INTEGER	N	NULL	0
	STREET	VARCHAR	N	64	0
	CITY	VARCHAR	N	64	0
	STATE	VARCHAR	N	16	0
	ZIP	VARCHAR	N	8	0
	OVERBOOK_RATE	DOUBLE	Y	NULL	0
	IS_ACTIVE	BOOLEAN	N	1	0
FLOORS	FLOOR_ID	INTEGER	N	NULL	0
	GARAGE_ID	INTEGER	N	NULL	0
	FLOOR_NUM	INTEGER	N	NULL	0
	SPACE_COUNT	INTEGER	N	NULL	0
PARKING_SPACES	SPACE_ID	INTEGER	N	NULL	0
	FLOOR_ID	INTEGER	N	NULL	0
	STATUS_ID	INTEGER	N	NULL	0
	WALK_IN	BOOLEAN	Y	1	0
SPACE_STAT	STATUS_ID	INTEGER	N	NULL	0

US	DESCRIPTION	VARCHAR	N	16	0
PRICING	PRICING_ID	INTEGER	N	NULL	0
	DESCRIPTION	VARCHAR	N	64	0
	COST	VARCHAR	N	24	0
	DAILY_MAX	INTEGER	N	NULL	0
	RESERVATION_TYPE_ID	INTEGER	N	NULL	0
RESERVATION_TYPE	RESERVATION_TYPE_ID	INTEGER	N	NULL	0
	DESCRIPTION	VARCHAR	N	64	0
RESERVATION_STATUS	STATUS_ID	INTEGER	N	NULL	0
	DESCRIPTION	VARCHAR	Y	64	0
RESERVATIONS	RESERVATION_ID	INTEGER	N	NULL	0
	MEMBER_ID	INTEGER	N	NULL	0
	RESERVATION_TYPE_ID	INTEGER	N	NULL	0
	VEHICLE_ID	INTEGER	Y	NULL	0
	START_TIME	TIMESTAMP	N	10	6
	END_TIME	TIMESTAMP	Y	10	6
	DATE_CREATED	TIMESTAMP	Y	10	6
	EXTRA_GRACE	BOOLEAN	Y	1	0
	STATUS_ID	INT	N	NULL	0
INVOICE	LINE_ITEM_ID	INTEGER	N	NULL	0
	MEMBER_ID	INTEGER	N	NULL	0
	CARD_ID	INTEGER	N	NULL	0
	RESERVATION_ID	INTEGER	N	NULL	0
	STATEMENT_DATE	INTEGER	N	4	0
BILLING	CARD_ID	INTEGER	N	NULL	0
	MEMBER_ID	INTEGER	N	NULL	0
	DESCRIPTION	VARCHAR	Y	64	0

	CAR_TYPE_ID	INTEGER	N	NULL	0
	CC_NUMBER	VARCHAR	N	16	0
	CCV	INTEGER	N	NULL	0
	STREET	VARCHAR	N	64	0
	CITY	VARCHAR	N	64	0
	STATE	VARCHAR	N	16	0
	ZIP	VARCHAR	N	16	0
	IS_ACTIVE	BOOLEAN	N	1	0
CARD_TYPE	CARD_TYPE_ID	INTEGER	N	NULL	0
	DESCRIPTION	VARCHAR	Y	16	0



1.4 - Mathematical Model

This application will use multiple mathematical functions in order to display the statistics of the garage network for the garage operator. These functions include:

- Overstay Rate: Ratio of Overstays to Total Stays
    - Average Overstay Time: Mean of the time after the grace period has ended a car is in the garage
- Understay Rate: Ratio of Overstays to Total Stays
    - Average Understay Time: Mean of the time prior to the end of the reservation the car leaves
- No-show Rate: Ratio of reservations that do no show
- Cancellation Rate: Ratio of Reservations that cancel
    - Average Time of Cancellation: Mean Time prior to the reservation starting the reservation is canceled
- Current Overbooking Rate:
    - Number of reservation spots that can be reserved past full: Overbooking Rate * Number of Reservation Spots
- Average Length of Stay: Mean Length of Reservation by Walk-in and Non-Guaranteed Reservation
- Walk-in to Reservation Ratio: Ratio of Walk-ins to Reservations
- One-Time vs Guaranteed Reservation Ratio:Ratio of One-time Reservations to Guaranteed Reservations
- Occupancy Rate for Guaranteed Reservations: Ratio of time Guaranteed Reservation spots are Occupied to Empty during the time of the Guaranteed Reservations
- Income per Garage Each Month: Sum of all invoice lines per Garage per month

2 - Interaction Diagrams

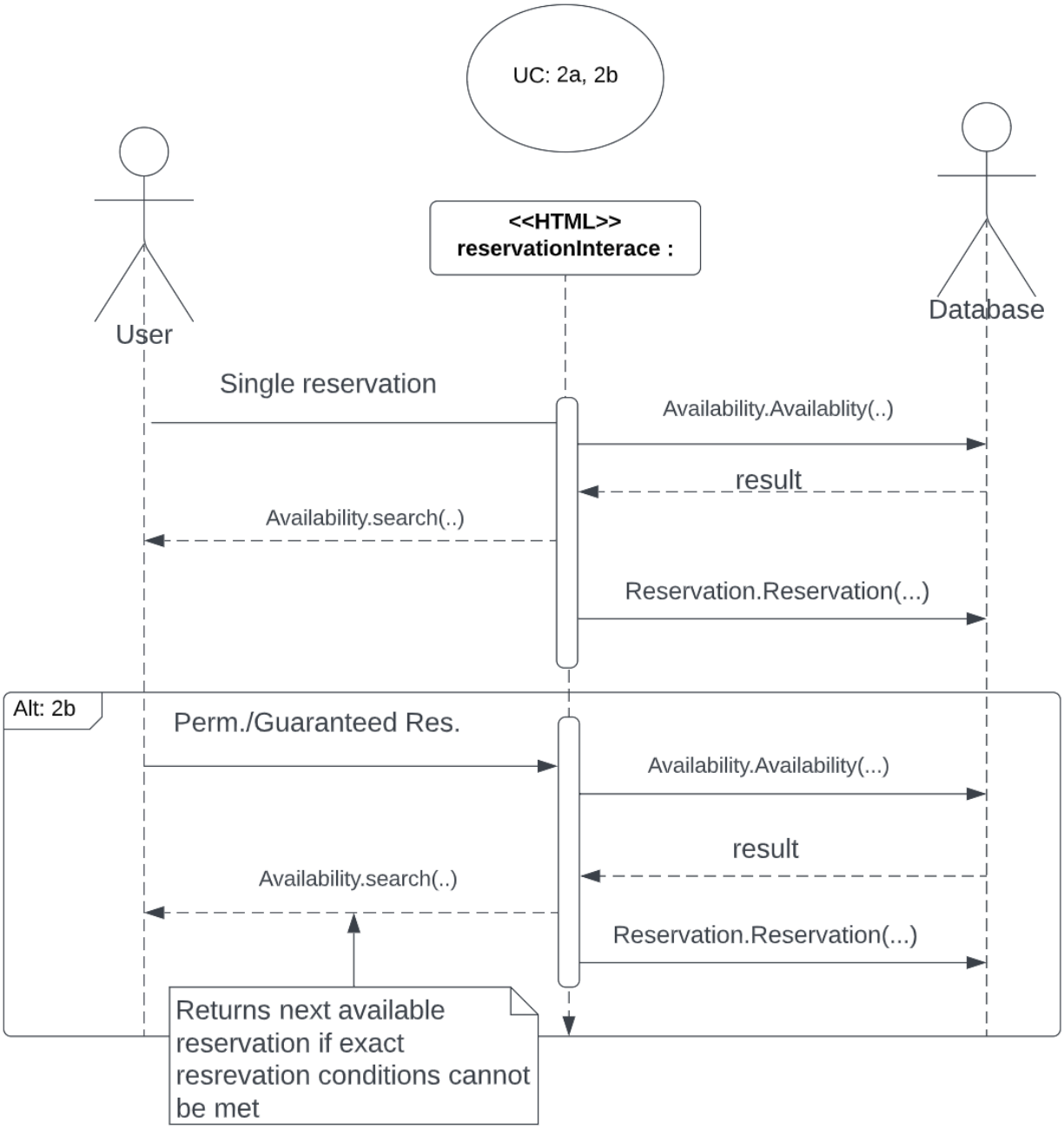
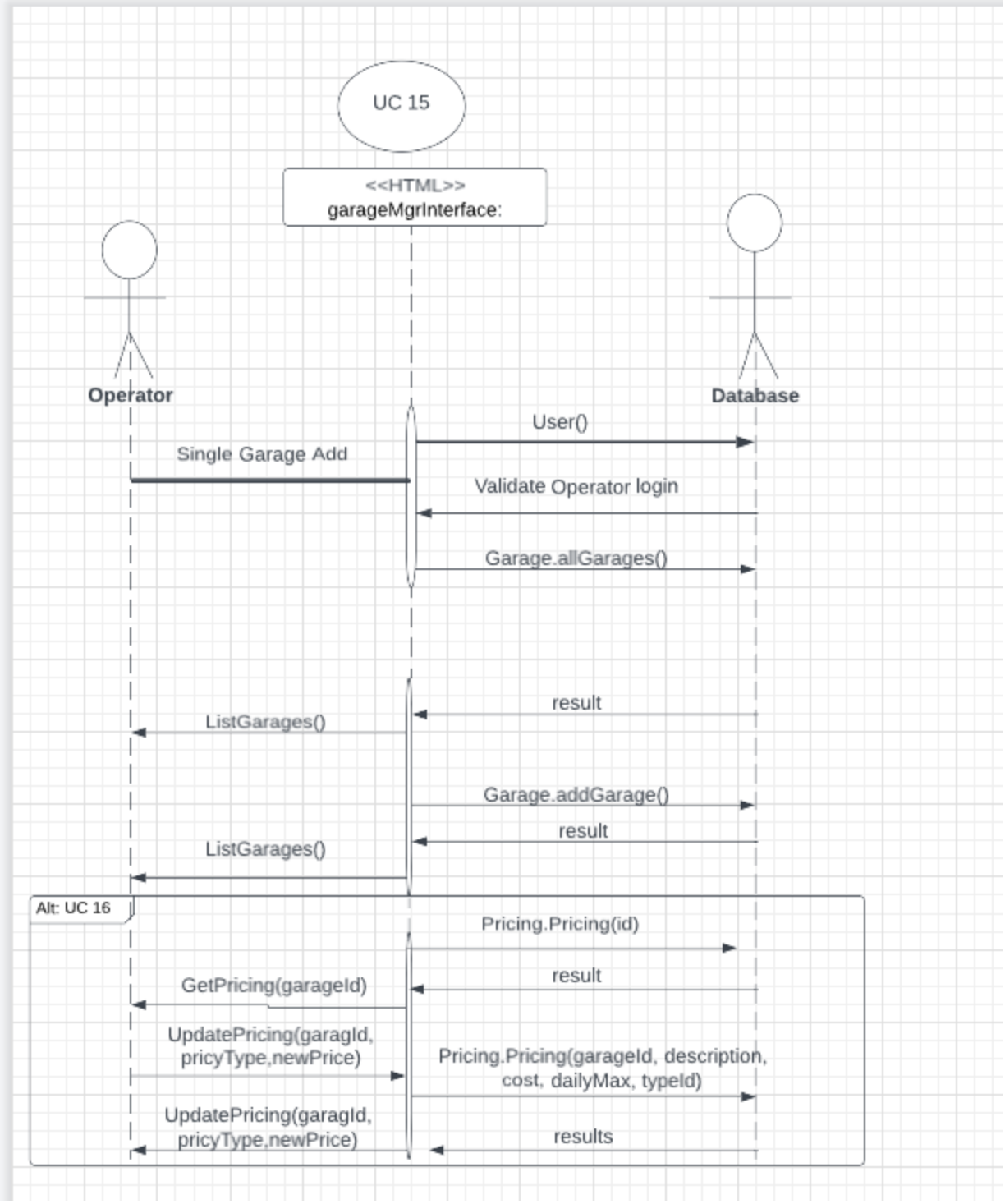


Figure 7.1 UC-02 Sequence Diagram

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.



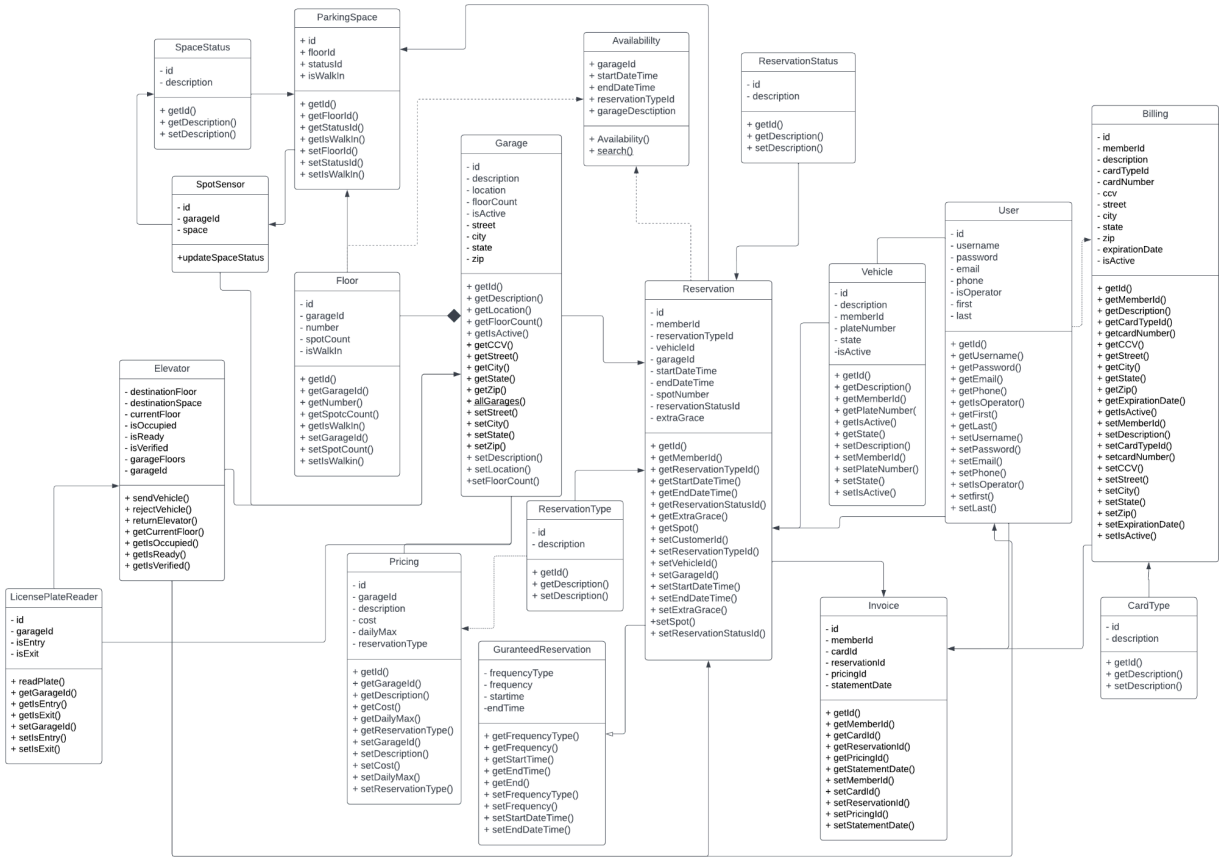
For UC 15, adding a new garage, we have assigned the responsibility to garageMgrInterface concept to receive the information, passed as arguments, to add a new garage to the Operator's network. The Operator will login; after a successful login, the Operator's garage list will be fetched by allGarages(), 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.

3 - Class Diagram and Interface Specification

3.1 - Class Diagram



3.2 - Data Types and Operation Signatures

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

Reservation
- id : integer - memberId : integer - reservationTypeId : integer - vehicleId : integer - garageId : integer - startDateTime : string - endDateTime : string - spotNumber : integer - reservationStatusId : integer - extraGrace : boolean

+ Reservation(id : integer)
+ Reservation(memberId : integer, reservationTypeId : integer, vehicleId : integer, garageId : integer, startDateTime : string, endDateTime : string, spotNumber : integer, reservationStatusId : integer, extraGrace : boolean)
+ Reservation(memberId : integer, vehicleId : integer, chosenGarage : Availability)
+ getId() : integer
+ getMemberId() : integer
+ getReservationTypeId() : integer
+ getVehicleId() : integer
+ getGarageId() : integer
+ getStartDateTime() : string
+ getEndDateTime() : string
+ 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) : boolean
+ setEndDateTime(endDateTime : 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
    - 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,

            startDateTime, endDateTime, spotNumber, reservationStatusId, extraGrace) - Creates a new reservation
        - Reservation(memberId, vehicleId, chosenGarage) - Creates a new reservation from the chosen Availability object
    - 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

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

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

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

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

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

Pricing
- id : integer - garageId : integer - description : string - cost : string - dailyMax : integer - reservationTypeId : integer
+ Pricing(id : integer) + Pricing(garageId : integer, description : string, cost : string, dailyMax : integer, reservationTypeId : integer) + getId() : integer + getGarageId() : integer + getDescription() : string + getCost() : string + getDailyMax() : integer + getReservationTypeId() : integer + setGarageId(garageId : integer) : boolean + setDescription(description : string) : boolean + setCost(cost : string) : boolean + setDailyMax(dailyMax : integer) : boolean

+ setReservationTypeId(reservationTypeId : integer) : boolean

- Class
    - Represents a pricing type for parking
- Attributes
    - id - The pricing type ID
    - garageId - The ID of the garage the price is for
    - 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 and setters

Garage
- id : integer - description : string - floorCount : integer - street : string - city : string - state : string - zip : string - overbookRate : float - isActive : boolean
+ Garage(id : integer) + Garage(description : string, floorCount : integer, street : string, city : string, state : string, zip : string, overbookRate : float, isActive : boolean) + allGarages() : Garage[*] + getId() : integer + getDescription() : string + getFloorCount() : integer + getStreet() : string + getCity() : string + getState() : string + getZip() : string + getOverbookRate() : float

+ getIsActive() : boolean + setDescription(description : string) : boolean + setFloorCount(floorCount : integer) : boolean + setStreet(street : string) : boolean + setCity(city : string) : boolean + setState(state : string) : boolean + setZip(zip : string) : boolean + setOverbookRate(overbookRate : float) : boolean + setIsActive(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
    - street, city, state, zip - The street address 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(description, floorCount, street, city, state, zip, 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 and setters

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

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

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

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

LicensePlateReader
- id : integer - garageId : integer - isEntry : boolean - isExit : boolean
+ LicensePlateReader(id : integer) + LicensePlateReader(garageId : integer, isEntry : boolean, isExit : boolean) + readPlate() : boolean + 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

SpotSensor

- id : integer - garageId : integer - space : ParkingSpace
+ SpotSensor(garageId : integer, space : ParkingSpace) + 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
    - space - The parking space the sensor is for
- Methods
    - Constructors
        - SpotSensor(garageId, space)
    - updateSpaceStatus() - Updates the status of the parking space

3.3 - 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.

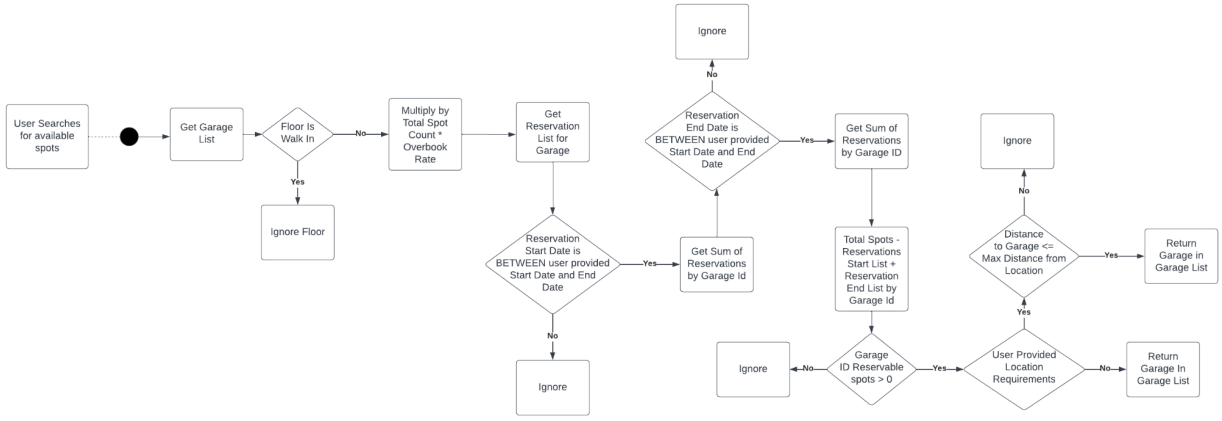
The spotSensor concept led to: ParkingSpace, SpaceStatus, and SpotSensor. ParkingSpace represents a physical parking space in a garage, with an associated SpaceStatus updated by the associated SpotSensor for the parking space.

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.

4 - Algorithms and Data Structures

4.1 - Algorithms



When a user searches for a spot, a list of garages with available spots will need to be displayed. Two required search fields (Start and End Date and Times) will be entered by the user. Additionally, there are optional fields for location: maximum distance and

location. From this information, paired with the saved reservations, the list of garages with available parking will be displayed to the user.

4.2 - Data Structures

As a result of the algorithm mentioned above, an array of garage Ids 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.

4.3 - Concurrency

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

5 - 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.

6 - Design of Tests

6.1 - 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 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.

6.2 - 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.

6.3 - 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.

6.4 - 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	Poll all cameras and sensors to detect whether	All cameras and sensors are successfully polled.

	operational at all times	they are operational.	
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 updates that need to be reflected on the UI immediately	Measure DB update time versus complete request time.	DB update time >= 0.5 * request 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.

6.5 - 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 reserve their spot online.	Attempt to reserve a space online.	Reservation is successful.
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.

7 - Plan of Work

7.1 - Merging the Contributions of Individual Team Members

The report document is maintained and shared on Google Drive. As such, all team members have read/write access to the master report document. Google Docs has a lot of functionality to assist in working with teams over a single document. All formatting is handled by defined text and heading types, with headings automatically populating the table of contents. Comments, change proposals, and section discussions all happen within the document, and each sends automatic email updates to all team members. As sections are completed, all team members are asked to review the completed work and make any necessary suggestions or fixes.

Before report submission, a final pass is made by the team. The report specifications are checked against the completed report to ensure all sections exist and meet the established requirements. Grammar and spelling are checked, and formatting is checked and updated for consistency. Grammarly's (grammarly.com) automatic tools are used here to ensure the entire document has correct grammar, clear and concise

language, and a consistent tone. Once the final pass is complete and all team members consent, the final report version is named in the document history, saved to Google Drive as a PDF, and submitted.

With this workflow and the functionality provided by Google Drive, there have been no serious issues merging the contributions of the team into this report.

7.2 - Project Coordination and Progress Report

Database: Currently, the initial Database Structure has been built. The database has been connected to the system, and the first test queries have been successfully executed. There will need to be views built for garage availability to be pulled into the system. Additionally, Insert and update statements will need to be written in accordance with Node.JS syntax to begin testing integration.

Use Cases: Use cases 2a and 2b are in their initial stages. Using MVC architecture with Express API routing, the controller and route for UC-2a and UC-2b have been defined with function stubs and unit tests. These two use cases may be called on and return temporary responses. The stub functions are in the process of being filled out completely. A blueprint for UC-15 and UC-16 has been layed out with stub functions. User registration and login is functional.

User Interface: The 4 major components of the UI, Register, Login, Locator, and Walk-In, have had their first initial page finished. A re-routing routine was added in case a user mistyped a URL to take them back to the Locator page. Another feature added was a custom 404 error page if the URL was completely wrong or a page loaded incorrectly. Functionality for the Locator reservation search page is in progress, with support for maps and geolocation.

Server: The Express server that runs the project has been created and hosted. The skeletonized app is functional. New functionality, including new models, views, controllers, and Express routes, is integrated into the Express structure and automatically deployed to the live server through Github integration.

Project Management:

- **Meetings:** The team meets through WebEx every Monday and Wednesday morning.
- **Deadlines:** Deadlines are tracked through a spreadsheet as well as the project plan of work and Gantt chart.

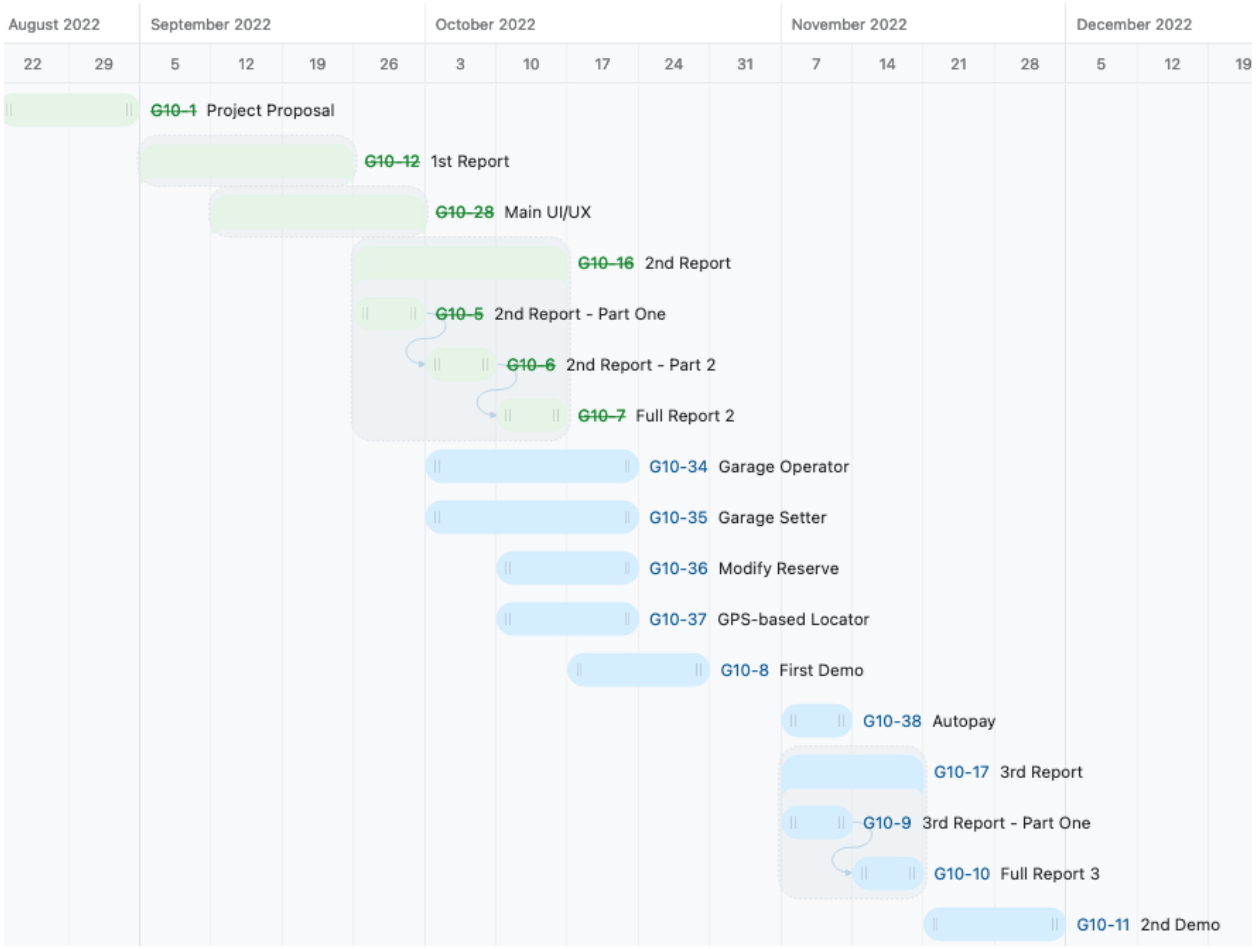
- **Shared Resources:**
  - The project and project website repositories are hosted on Github, and all new functionality is introduced through feature branches with pull requests, which are reviewed by other members of the team and merged into the dev branch.
  - All project documents are hosted on Google Drive in a single shared folder to which all members have access.

7.3 - Plan of Work

Project Target	Tasks	Description	Contributors	End Date	Technologies
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	***UI/UX*** HTML, CSS, Bootstrap4, Javascript, AngularJs ***Backend*** C#,Express, NodeJs, DB2
	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 available spots for them.	Dustin Threet, Justin Henley	9/12/22 - 10/02/22	
	Database Diagram	Basic structure for entities that will be stored in the DB	Hannah Carr	9/12/22 - 10/02/22	DB2
1st Report : Part 2	Functional Requirements		Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	9/18/22	N/A

Full Report 1			Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	9/25/22	N/A
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	***UI/UX*** HTML, CSS, Bootstrap4, Javascript, AngularJs
2nd Report: Part 1	Use cases/ User interactions		Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	10/2/22	N/A
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	***Backend*** C#,Express, NodeJs, DB2
2nd Report: Part 2	System architecture		Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	10/9/22	N/A
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	***UI/UX*** HTML, CSS, Bootstrap4, Javascript, AngularJs ***Backend*** C#,Express, NodeJs, DB2
Full Report 2			Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	10/16/22	N/A
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, DB2

First Demo		Combine project to present		10/30/22	Project
Autopay	Alter autopay	See user's payment history, change payment options/ occurrence	Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	11/12/22	***Backend*** C#,Express, NodeJs, DB2
3rd Report: Part 1			Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	11/13/22	N/A
Full Report 3			Justin Henley, Victor Munoz, Hannah Carr, Dustin Threet	11/20/22	N/A
Second Demo		Combine project to present		12/24/22	Project



7.4 - Breakdown of Responsibilities

- **Dustin:** Accounts
    - Registration, Login
    - Classes: User, Vehicle
- **Justin:** UC-2a & UC-2b - Reservations
    - Single reservations, Guaranteed reservations
    - Classes: Availability, Reservation, ReservationType, ReservationStatus
- **Victor:** UC-15 & UC-16
    - Garage Configuration, Pricing
    - Classes: Garage, Floor, ParkingSpace, SpaceStatus, Pricing
- **Hannah:** DB Administration:
    - Data Models

    - Classes: DatabaseConnection
- **Hannah:** Integration Coordination and Integration Testing

8 - 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).

DB2 On Cloud (no date) *IBM*. Available at: https://www.ibm.com/cloud/db2-on-cloud (Accessed: October 15, 2022).

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

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

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

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