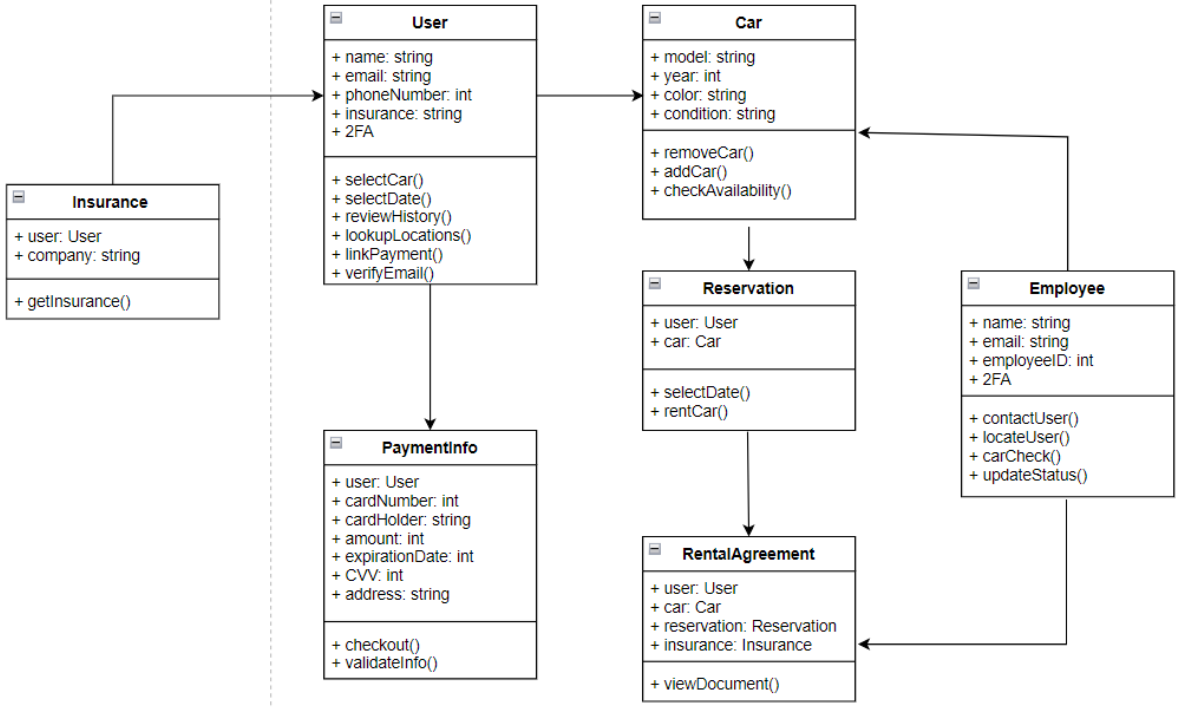
Car Rental System

Alejandro Pacheco, Kelly Aycock, Steven Trujillo

System Description

This system will make renting a car for any person an effortless task through this well-built car rental system. The system will operate through an app that can be downloaded on any mobile device or can be accessed in our locations at the front desk to make renting out cars easily. Users will simply sign in with their name, email, phone number, driver's license, and payment information and then be able to select through a large assortment of vehicles on the lot. In the app, the users after signing in will be directed to select the model, year, color, and how long they would like to reserve the selected car. To complete the process of checking out the vehicle the user will link their insurance information or be able to pay extra to go through the onsite insurance agency. After renting out the vehicle the renter or employees of the car rental business will be able to view the rental agreement at any time through the app. Employees will have access to more information through the employee login in the app which requires an additional employee id number to enter. Once logged into the administration side of the system employees can contact car renters in case of any emergency or to update them on new policies. On top of that, they will be able to locate cars that are currently rented out. Most importantly they can check the availability of cars to better assist customers and update the status of cars if for example they are damaged or out for maintenance. This system will also be made in a manner that it is able to be scaled to other car rental businesses. In total, this system will make renting cars easy for the renter and the employees of the car rental business.

UML Class Diagram



Description

In this UML diagram, we have the User class acting as the base. This class contains the user's information needed by the company in order to add them to the car rental system. There is an additional two factor authentication for security purposes. The PaymentInfo class that is inherited from the User class provides the system the information needed for the system to allow a rental to take place. In addition, our user class contains other methods that will allow the user to rent a car, select a booking date, review their own history, among other functions. These all work together in order to give the user the best experience that will give them all the functionalities needed.

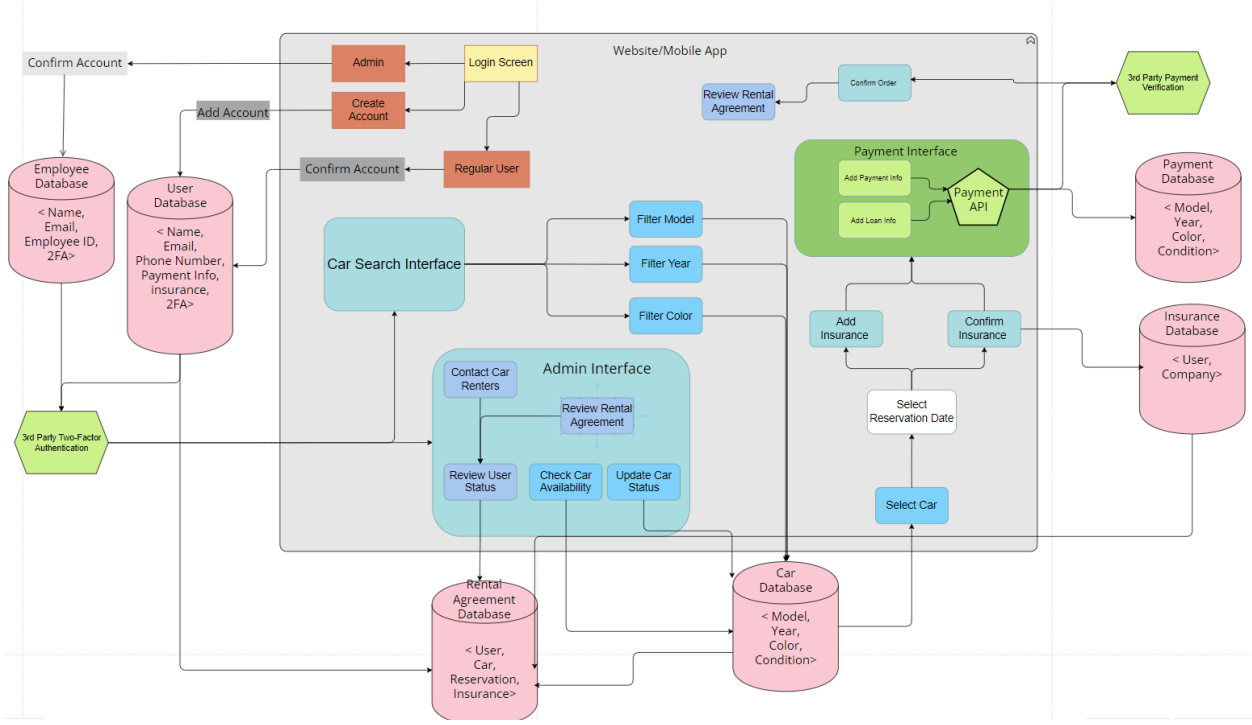
The PaymentInfo class enables the user to make payments for their car rentals. This object will store all the information about the user and credit card information. The class also has a checkout function that will let the user checkout after giving their account info.

In addition, we have a car class that contains models, year, color, condition, which are parameters that will classify our car class. If needed, we have a function

that removes the car from the inventory if it has run its service or has no more function. The car class serves as the basis of this system, as it connects to databases and serves at the main pivotal point of the program. We also connect this class with a reservation that will store the user's reservation with their car. The reservation takes in parameters of a user and car and contains functions that will let them select their reservation date, and which car they are interested in. Lastly, this class connects with the rental agreement that takes in a user, car, reservation, and insurance info. This rental agreement will have one function that is viewDocument, which lets the user or employee access the rental agreement that was previously agreed upon.

Lastly, our employee class takes in a name, email, employee ID, and ensures two factor authentication is enabled. Our employee has access to functions like contacting the user, locating the user, checking a car status, and updating the car status. These functions are crucial to allow our employees to have superior actions over a user, and allow them to modify the stock and availability of cars.

Software Architecture Diagram



Description

This architecture diagram illustrates the various components of the car rental system. To begin our workflow, the user will be introduced to the website or app through a login screen. They will be unable to perform any actions here until they login or create an account. There will also be an admin sign in button at the bottom right that will allow employees to log into the system. After this, their credentials will be verified through the databases, and a third party two factor authentication system will verify the login. Depending on the user that signed into the system, it will either direct them to the car search interface or the admin interface.

If our user is a customer, they will now be able to filter their car choice by model, year, or color. Once these parameters are chosen, the database will be monitored to determine which cars are available for rental. Once this is done, the system will now prompt the user to select a car, as well as a reservation date. From here the user will be prompted to confirm or add their insurance. The last vital step to this system is now having the user pay. They can either add their payment info like a credit card, or they can apply their loan information directly on the website. From here, a payment API will confirm the credentials and send the information to a secure third party payment source. Once this is done, the order will be confirmed and the user is able to review their rental agreement.

However, if our user is an employee they have several permissions to modify certain aspects of the program. Firstly, they can check car availability if needed, as well as update a car's status. Furthermore, they can review rental agreements that have previously been set, and contact renters if anything is needed.

Development Plan

The development for this system will undergo several different phases. The first phase is to build a functioning website with basic buttons and code flow. We'd like to establish a baseline for what the website should look like, and further the development from there. This step should include a login screen, a car feature selection page, and lastly a basic payment window. This step should take no longer than 1-2 weeks. To ensure the best quality, we're planning to finish this step in 2 weeks. Alejandro will be assigned to work for this part of the project.

During this time Kelly will start working on the payment API, as well as verification of our user's insurance. This part is not dependent on any other aspects of the program for now, so ensuring that a properly functioning payment system that interacts with the third parties will be crucial. It's a little challenging to be able to manage user's payments securely and safely, so this task will be given for about 3 weeks.

Now Steven will be assigned with developing and managing our database systems for our users, employees, and car inventory. He will be implementing this in SQL and other database management languages that will allow us to properly store and update our cars, users, and employees. This will likely take us about 2 weeks to complete, and will align with the initial first phase of developing the website.

Once Steven and Alejandro both complete their parts at around the same time, they can come together and start to develop the more intricate portions of the website. At this point code flows can be merged together to work simultaneously. This part will take coordination and proper usage of API's to allow for continuous development. This task is expected to be completed within 2-3 weeks.

Last part is taking Kelly's contributions with the payment system and incorporating them into the remainder of our system that we've developed. Once the payment system is fully integrated into our system, we'll incorporate a database or storage that will keep track of all of our customer transactions and insurance info. This may take 1 week at most.

Overall, the timeline for the implementation of this program will be around 4-5 weeks with most efficient production. We truly do shoot for around 6-7 weeks to get the software fully implemented and ready to go. Throughout all this time, we will be running tests to ensure that our progress is on the right track. Any different changes or modifications that are done during testing will be reported to the group and the timeline can be modified if needed.

Test Plan:

In order to effectively test our software, we have to go through several different tests to have effective coverage. These two test sets are the most crucial part of our program, which includes renting a car as a user of our company, and modifying the status of an existing car in our system as an employee. In order to effectively cover as many results as we can, we've split our two test sets with unit tests which test a smaller functionality with a class. We've extended this with integration tests that will test functionalities that depend on multiple classes, and a system test that broadly tests the entirety of our functionality.

Lastly, we've split our test sets with valid or invalid results, which can be further split into different test cases that represent various results of successful or unsuccessful tests. Through this manner of testing, we can provide the most amount of coverage of our software. We will also provide a brief description of each test set, and functionality that we are testing.

Test Set 1: Renting a Car

Unit Tests:

Our unit tests for this test set will test the selectDate and verifyEmail functionality. Both of these tests will be testing the functionality within one class, and are both an important part of being able to rent a car in our system as a user.

The first unit test, verifyEmail() will test if our verifyEmail function works properly. If we are able to locate the email, we will send the email and provide a message sent notification to the user. If the email does not exist in our system, we will start over and inform the user to try again. Lastly, if our email is not a valid address, we will inform the user the email they provided is invalid.

For the second unit test, the selectDate() will incorporate the functionality of being able to successfully reserve a date for a car within the Reservation class. In order to test this, we must be able to receive a valid output given two strings which represent the reservation time in the form of (MMDDYYYY). If we have a valid reservation provided, our system will return a message validating the success of the date provided. If we were to provide a date that has an invalid range, such as the first string being before the second one in terms of our date, it will return an error message. If our date is already occupied, it will return an error message.

- verifyEmail()
    - verifyEmail(customer@gmail.com)
        - Valid: If the email exists in our system, we will send the email and a successful sent message will appear for the user.
        - Invalid: Email Not verified
            - Inform user the email was not verified

        - Take user back to verifyEmail for another entry attempt
    - verifyEmail(949-379-4490)
        - Invalid: Return message email is not valid
        - Take user back to verifyEmail for another entry attempt
- selectDate()
    - selectDate(07202023,07282023)
        - Valid: Returns a message saying the date has been reserved successfully if the date is available
        - Will allow the user to confirm their reservation
    - selectDate(07202023,07102023)
        - Invalid: Return message stating that the date is invalid
        - Will take user back to selectDate interface to try again
    - selectDate(07202023,07282023)
        - Invalid: Return message stating date is already occupied
        - Take user back to selectDate interface to try again
    -

Integration Tests:

Our integration tests will cover the selectCar and linkPayment functionality. The selectCar function will test the User, Employee, and Car class. We do this by going through the User class in the interface, and then checking the car's availability in the car's class. Secondly, our link payment will connect the functions between our User Class and PaymentInfo class.

For selectCar, we provide a pre-existing car object. With our system, our employee has the ability to determine if a car is occupied or not, which is why we must have the employee go through the rental process with our user. In the circumstance where the employee knows the car has been taken recently, we will update the function with updateStatus. Therefore, this will return false immediately. Otherwise, if our car passes this update status and returns true, we can verify the availability within the Car class using checkAvailability. From here, if our tests are valid, we can successfully rent out the car, or if it is invalid, we will need to restart the selection process

Lastly, for linkPayment, we will verify whether or not the user's payment info has been validated. We will do this by providing the user with the PaymentInfo class to enter their information and store it on their database. We'll be moving code flow from the User class into the PaymentInfo class. If we have a valid paymentInfo object, we can store the information on the database, and otherwise we will prompt the user of invalid information.

For the following unit tests, we will be using the following Car object
Car carOne(Ford, 2010, Blue, Good)
PaymentInfo TestUserInfo(User test, 919, Test LastName, 9000, 042025, 999, 123 Address)

- selectCar(x):
    - Employee updateStatus(x, occupied)
        - User rentCar(x) -> False

- Employee updateStatus(y, available)
    - User rentCar(x) -> True
        - checkAvailability(carOne)
            - Valid: If our car is available and verified within the car class, we can successfully choose the car
            - Invalid: If our car is not available within the car class, we cannot choose the car
                - The user will be taken back to the car selection menu.
- linkPayment(TestUserInfo)
    - Valid: If our info is valid through our initial screening in our user class, we can continue onward with the functions in PaymentInfo
    - Invalid: Our info is tested here for any blatant incorrect information that is presented initially. This can include incorrect expiration dates, names that aren't under our insurance policy, among others. If flagged here, the user will need to re-input their data.
        - PaymentInfo validateInfo(TestUserInfo)
            - Valid: Our third-party verification will be called through the ValidateInfo API, and if successful, we have officially linked our payment
            - Invalid: If our third-party verification flags the information to be incomplete or the card does not exist, we will be taken back to our User Class to re-input the information.

System Test:

Our system test for the car rental system will cover the overarching system from the renter's perspective and from the employee's perspective. The system test on the renter will cover them logging into the system, selecting a vehicle, entering payment information, selecting insurance, and then reserving the car/receiving the rental agreement.

For the system the renter once opening the system will be prompted to create an account. This will require the user to input a name, email, phone number, and insurance. On top of that, they will be asked to verify their account through either email or phone number to activate two-factor authentication. If they are unable to create an account or verify it the expected behavior is they will not be able to proceed. After this, they will be able to look through all available cars which display the model, year, color, and condition. After selecting a vehicle they will be required to select dates if the vehicle is already booked for overlapping dates the user will be prompted to choose different dates or another vehicle that works for their dates. After this, the user will need to link a payment method to their account. This will require their card information and address, without linking payment the system will not allow users to continue. After linking payment information they will proceed to the checkout where they finalize their reservation and

sign the rental agreement. At any time during the active rental period, the user will be able to log in to the system and view their active rental agreement.

- User(Steven, steven@gmail.com, 4087684002, AAA)
- PaymentInfo TestUserInfo(User test, 919, Test LastName, 9000, 042025, 999, 123 Address)
- Car x(Ford, 2010, Blue, Good)
- 2FA, verifyEmail(steven@gmail.com):
    - Valid: If the user's account is verified by two-factor authentication the system will allow the user to proceed to car selection.
    - Invalid: If the user's account is unable to be verified the user will be redirected to try again and will not be able to proceed to car selection.

- rentCar(x), selectDate(10-15)
    - Valid: If the vehicle the user selected is available for the given dates and in good condition, the user will be able to select vehicle x and proceed to link payment.
    - Invalid: If the user's selected vehicle is not available the user will be redirected to select a different vehicle before proceeding to link payment.

- linkPayment(TestUserInfo)
    - Valid: If information passes initial screening we continue to validate the information.
    - Invalid: If the address is clearly incorrect or for example, there aren't enough numbers to make up credit card information the user will be directed to try again.
        - validateInfo(TestUserInfo)
            - Valid: Checks via third party whether credit card information is correct and if valid continues user to checkout
            - Invalid: If the third-party API flags information as being invalid user will be notified and redirected to reattempt linking a payment method.

- checkout()
    - Valid: The user will have to accept the terms and conditions of the rental agreement and pay then will be able to retrieve their rental car.
    - Invalid: If the users don't accept the terms and conditions of the rental agreement they will be unable to complete the process.

- viewDocument()
    - Use: Anytime during the active rental period the user will be able to log in through the app and view the rental agreement.

Test Set 2: Modifying an Existing Car In the System

Unit Tests:

To effectively test the modification of a car in our system, we must go through the process of adding a car and removing a car. These two functions are an integral part of being able to modify the existing cars in our system, but are the basis of much more of our software system.

To test this, we must add a car into the system, and if this process is validated, we will return a message indicating that we've successfully added a car into the system. If this process is invalidated, we must restart the add car process, which may include creating a new car object, but is not the basis for this test.

Similar to adding a car, removing a car will go through a similar process mentioned previously. We will have a mock car, in which the validated removal process will send us a message saying the car has been removed. Otherwise, it will flag us and inform us that the car has not been removed, as likely due to the car not existing in our system.

For the unit tests, we will be using the following Car object

- Car carOne(Ford, 2010, Blue, Good)
- addCar()
    - addCar(carOne)
        - Valid: Returns a message saying car was returned and added back to the system
    - addCar(carOne)
        - Invalid: Returns a message saying car couldn't be successfully returned and asks to try again
- removeCar()
    - removeCar(CarOne)
        - Valid: Returns a message saying car was successfully rented and removed from the list of available cars
        - Invalid: Returns a message saying car rental failed and asks to re input information

Integration Tests:

For our integration tests, we will be testing the CarCheck and the UpdateStatus function. These two functions will allow us to review if a car is in stock and update the status of the car in our system. These functions are different from our overall test set, as these functions are not adding or removing the current cars in our system.

To test our CarCheck() we will create a test car class that will be checking if the car exists from the Employee Class. From here, we will integrate a call from our Car class to check the availability of our car in the database. If our car exists in our database we will confirm that our car was found by the employee. If we cannot confirm this, we will return a message implying that we could not find the car.

Similar to the updateStatus(), we will move from the Employee class into the Car class. The main difference is that this is just updating the overall condition of our car. If we can update

the status, we will inform the employee of the success of the update. Otherwise, we will inform the user that this is an invalid modification or the car does not exist in the database.

For the following integration tests, we will use the following car Object
Car carOne(Ford, 2010, Blue, Good)

- CarCheck(carOne)
    - Car checkAvailability(carOne)
        - Valid: Returns a message saying car was successfully checked, available, and in good condition
        - Invalid: Returns a message saying the car check failed and that they need to re-input the parameters.
- UpdateStatus(carOne, poor)
    - Valid: If our input is valid, we can continue to checking the car in the Car class
        - Valid: If the car exists in the database, we can update the car value directly here.
        - Invalid: If our car is not a part of the car database, we will inform
- UpdateStatus(carOne, invalidInput)
    - Invalid: If we input an invalid value, we will be informed that the condition is not available

System Test:

Our system test for the car rental system will cover the overarching system from the renter's perspective and from the employee's perspective. The system test for the employee will follow the flow of an employee logging into the system and being able to then view the rental agreement and make modifications to cars in the system.

For the system the employee once opening the system will be prompted to log into their employee account. This will require the user to input a name, email, and employee ID number. On top of that, they will be required to verify themselves through the two-factor authentication process. If the user is unable to successfully provide an employee id number and verify through two-factor authentication the expected behavior is they will not be able to proceed. After logging into the system they will have a couple of options for what they are able to do. They can view any active rental agreement or previous ones. Through the rental agreements, they will be able to locate active users and contact them. They are also able to check on vehicles and update their status. If a vehicle is damaged for example they can update the status to not available. However, they should not be able to make a vehicle available for people to rent if there is an active rental agreement for it.

- Employee(Steven, steven@gmail.com, 8250)
- User x(John, john@gmail.com, 4087684002, AAA)
- Car carOne(Ford, 2010, Blue, Good)
- 2FA, verifyEmail(steven@gmail.com):

        - Valid: If the employee's account is verified by two-factor authentication and has the correct employee id the system will allow the employee to proceed to the management system.
        - Invalid: If the employee's account is unable to be verified the employee will be redirected to try again and will not be able to proceed to the management system.

- viewDocument()
    - contactUser(x)
        - Valid: The system should always allow the employee to be able to contact a renter at any given time.
    - locateUser(x)
        - Valid: The system should allow employees to locate where the vehicle is at for any rental during active rental agreements
        - Invalid: The system will not allow employees to track users after the rental agreement has expired.

- carCheck(carOne)
    - checkAvailabillity(carOne)
        - Valid: Informs the employee that the vehicle is in good condition and currently available for rent.
        - Invalid: Car check failed or the car is currently not available for rent due to multiple potential circumstances.

- updateStatus(carOne, occupied)
    - Valid: Updates carOne's status to occupied in the system
    - Invalid: attempting to set this car to available for rent while currently occupied.

Data Management Strategy

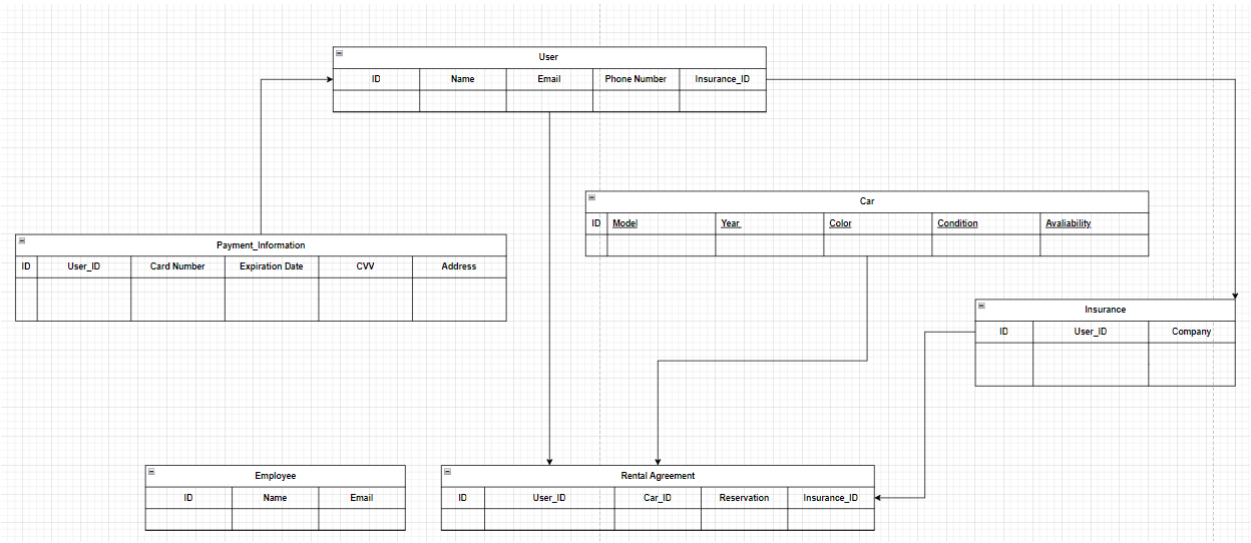
Description:

In order to effectively store our data, we made slight modifications to our Software Architecture Diagram. We previously did not have all the databases to store the essential information for our software to run properly. In our previous plan, the software was expected to just store all the information in objects in our program, as opposed to storing them more efficiently and securely in our database. Our code should solely contain functions that will allow our program to properly run, instead of needing to store our car inventory and information. The final modification that was made, was an updated test plan. This test plan will cover the new SQL queries to our databases, a no-SQL data table, as well as proper integration of our functionality of the system.

Having said that, we have decided to develop our database management mostly through SQL. We believe that being able to properly use and manage our relational database is a crucial aspect to having a system that successfully operates, but also one that is efficient. Being able to connect our database entries like cars, status, and other useful information will be done more efficiently through the usage of shared ID's with relational databases. SQL allows us to do this process eloquently and efficiently, without needing to accommodate other API calls to non-relational databases. We do have a no-SQL feature, which is our employee database, which will store the information about each employee. Their information has no relation to our other databases, so we believed the best course of action would be to implement it with a no-SQL approach. Our database will also benefit from a predefined schema as there is no plans to add new categories into our storing of the data

There were lots of design choices we had to make with respect to storing our car, user, and other important information. For starters, we had to consider what data we should include or remove from our codebase. This information included car objects, user information, and our rental agreements. We ultimately decided that it would be best to transfer this information to a database, as we want our classes to solely contain functions and not data. This required some modifications to our code, but will be more effective having this information stored in relational databases. Having the main principle be SQL over NoSQL is important because we store rental agreements that take a user field, car field, and insurance field. We need to store it this way to be able to efficiently track and save information as needed. This process is essential to having our software run in parallel with the user information and inputs.

For our data management diagram, we elected to have 6 different tables, with 5 of them maintaining relationships with each other. We decided to have an employee table, which will contain simple information and will be verified each time our worker logs in. This was the only portion of our database that included no-SQL because of the nature of its verification and creation in our codebase. Next, we have a user that contains information about all our users. This user has an insurance relation, which will be inputted near the beginning of the creation of their account. Next, we also have a car, which contains all our data on the car we have on our system. Lastly, we have a rental agreement which is the combination of a user, car, and insurance. The rental agreement is the main reason we elected for a SQL approach for these tables, as it contains relations to most of the data tables that we have.


User
ID
Name
Email
Phone Number
Insurance_ID
Car
ID
Model
Year
Color
Condition
Availability
Payment_Information
ID
User_ID
Card Number
Expiration Date
CVV
Address
Insurance
ID
User_ID
Company
Employee
ID
Name
Email
Rental Agreement
ID
User_ID
Car_ID
Reservation
Insurance_ID