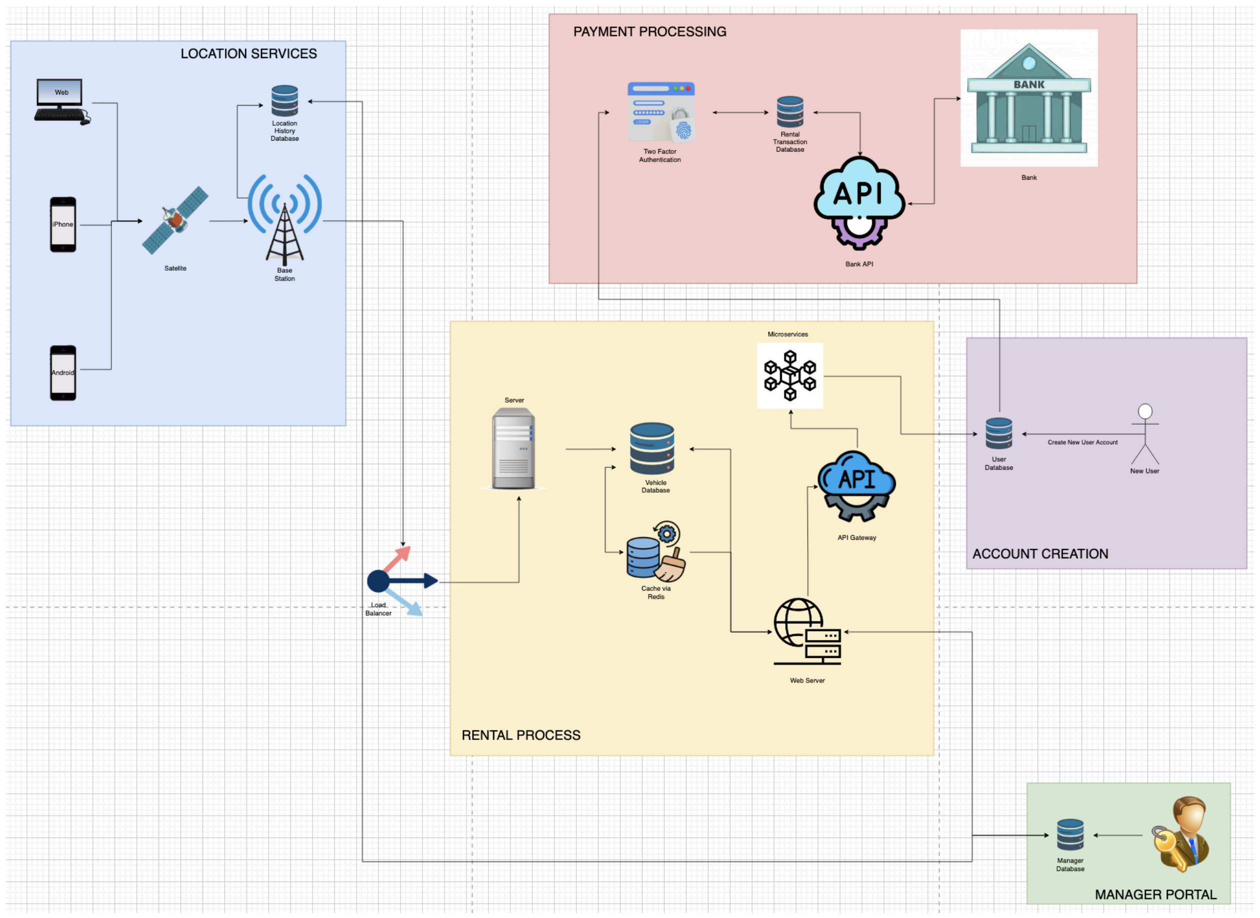
BeAvis Car Rental System Design Specifications

Team Members: Abdihakim Ahmed, Jake Foster, Alex Huang

System Description

The BeAvis Car Rental System is designed to replace the company's outdated rental process and should be accessible as an app on a mobile phone or as a website. Whether on the app or website, users must first make an account and verify their identity by correctly entering the verification code sent to them. Once the user sets up their account, they can make a rental, look at locations, and view the rental history. Users are encouraged to pay through the app or website to ensure a quick and secure transaction. The payment information is stored separately from other information to minimize the fallout of a potential privacy breach. This document will provide an optimal software architecture diagram as well as the best UML diagram so that the BeAvis software system can be correctly designed and implemented.

Software Architecture Diagram



SWA Description

Our software architecture diagram follows the flow from left to right and starts with the three possible uses whether it is on an iPhone, Android, or the web. The box labeled “Location Services” on the left is meant to satisfy the requirement of allowing the user to enable their location which would give them access to locations near them. First, satellites communicate with a device's GPS to pinpoint the location of it. This location is then sent to the base station and the user is now able to turn on their location. Before reaching the server, we have a load balancer so that the server maintains efficiency and speed.

In the “Rental Process” of the diagram, the user finally can connect to the server. Once the connection is established, the server accesses all of the information within the central database which stores all of the rental information. Next, we must implement access to a web server so that the user can make rentals based on updated information within the system. New rentals must be taken into consideration and sent back to the central database. Since the central database and web server are communicating back and forth, we used a double-sided arrow. The web server then accesses the user database which stores important details about the users account such as their rental history or account information.

For account creation, we simply just connect to the user database to make and store the user's password, email, and other account information. Nothing else is necessary for account creation. This also doesn’t follow the traditional left-to-right flow of our diagram as creating an account is very simple which allows us to bypass mostly everything else in the diagram.

The Manager Portal communicates back and forth (double-sided arrow) with the web server as they would be able to make adjustments or updates to the available rentals as well as view customers’ rental history. The manager portal would communicate this information with the web server which would then be sent back to the central database.

In the “Payment Processing” section, there is a two-factor authentication that the user must complete before making a payment. This is only necessary for making a payment and is not required to view the information on the site. Then the system accesses a payment card industry data security standard database to successfully and safely deal with purchases. However, if the user is not verified, they will be sent back to authenticate themselves which is shown with the double-sided arrows. Next, a bank API is accessed so that the payment process can be super quick and easy for the user. Finally, the bank API sends the payment information directly to the bank so that the sender's bank can send the correct amount of money to the receiver's bank. We modeled the payment section of our software architecture diagram after that of Venmo (How To Make An App Like Venmo - IdeaUsher).

UML Class Diagram

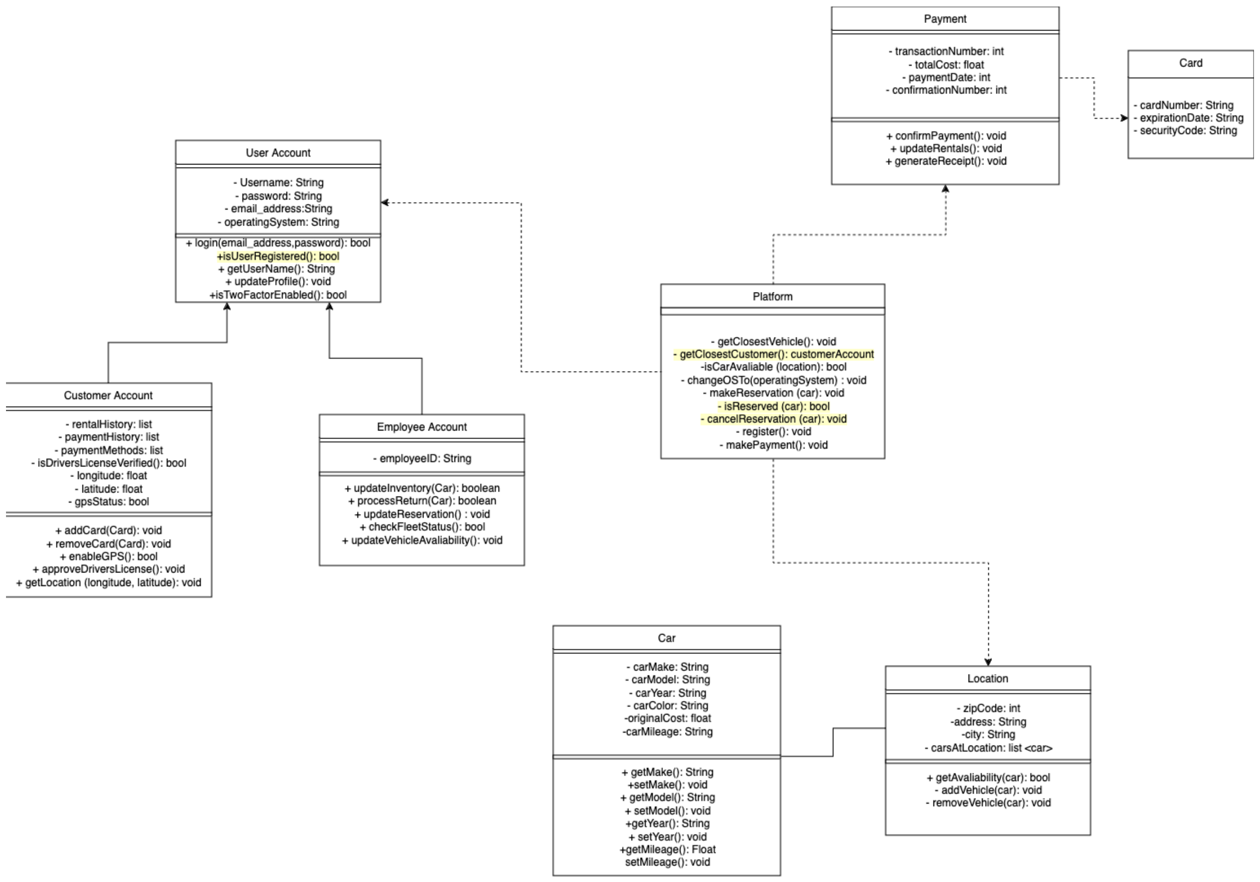


Diagram Description

Platform Class: The platform class is the center of the UML diagram as it deals with various functions with parameters from different classes. Without it, our system would not function properly. It consists of a database that uses different types of methods such as getClosestVehicle() and getClosestCustomer(). This would assist a customer with choosing a vehicle closest to the customer based on the location that is inputted. isCarAvaliable(location) is the method that inserts the availability of a car that is contingent to the location. The return type for this method is a boolean, contingent on whether the car is available or not.

changeOSTo(operatingSystem) gives the option to the customer to change what operating system that can be used whether it is an iPhone, Android, or a web browser. makeReservation(car) is a method that creates a reservation to obtain a car. Then there is the register() method which constructs and registers a user account into the platform/database. Lastly, makePayment() initiates the payment procedure when completing the reservation.

This class depends on many of our other classes to carry out these methods successfully. The Platform class must be connected to the Payment class as it would need to access the information within that class in order to use makePayment(). The platform class also depends on the Location class because methods such as isCarAvailable() require the parameter of location. Finally, the

Platform Class depends on the User Account class and this is seen where attributes from the User Account class such as operatingSystem are passed through into methods within the Platform class. Therefore, the Platform class is dependent upon the Payment class, Platform class, and User Account class.

Card Class: We designed a card class that holds the information of a customer's credit card such as the credit card number, security code, and expiration date. This information should all be stored in the form of a string. This class has no operations as we simply want to gather information to instantiate a card object.

Payment Class: There is a payment class that has the attributes of the cost of the rental, the date of payment, the transaction number, and the confirmation number. All of these should be integers except for the cost which would be a float. The class also has methods such as confirming that a payment went through and generating a receipt for a given purchase. Once a purchase is made, the operation of update rentals may be accessed to acknowledge that one of the vehicles was recently rented out. None of these operations require a return of any sort so we set them all to void. We also used a dependency relationship between payment and card as the payment class requires credit card information to be successful.

Car Class: The car class holds vital information about all of the rental cars within the system such as their color, make, model, year, cost, original cost, and mileage. However, the information is general and not relative to their location. Original cost should be in the form of a float and all other attributes should be in the form of a string. The class also has getters and setters to access and establish the most important information of the cars such as the make, model, and year. These simple operations allow for easy updates and retrieval of data.

Location Class: The location class has the attributes of zip code, address, city, and carsAtLocation. Address and city are both strings, zip code is an int, and carsAtLocation is a list that passes in information from the car's class. Essentially, the carsAtLocation attribute would hold all of the available rentals per specific location so instead of displaying every single car possible it would show only the ones at that place. For operations, we have getAvailability(car) and take in the parameter of car. This method checks to see if a specific car is at a given location. This method will return true if it is found and false if not. Next, we have addVehicle(car) where a car is passed in as well. This would be used to add a specific vehicle to a specific location and also has a return type of boolean. Finally, we have removeVehicle(car) which removes a given car from a location. We wanted to make sure that these updates were done within the location class as different locations may have different vehicles available. There is an association between this and the car class as each location has various amounts of cars.

User Account: The user account class takes in strings of the users' email, password, username, and operating system. Key operations such as login exist which take in the user's email and password and return a boolean depending on if it was successful or not. This class also has a method to check whether the user has two-factor authentication enabled and returns true or false. To add, we have simple methods such as getUsername() which simply retrieves the username or updateProfile() which would allow the user to update or change their account information. The User Account class also acts as the parent class for the Customer Account and Employee Account.

Employee Account: This class only requires the employee's ID, as it inherits the rest of the information from the user account class. We drew a generalization connection between the employee account and user account classes as they are both accounts. However, this class has additional methods that only an employee can use, such as updateInventory(car) and processReturn(car). After passing in the correct parameter, employees can update their inventory based on recent purchases or even deal with returns. These two operations return true or false depending on if they were successful. Next, we have methods to update a reservation or update the availability of a vehicle. These two have a void return type as they are simply tasks. Finally, we have an operation to check the status of all of the cars at a location that has a boolean return type. This may provide useful information through a general report of all of the cars. These are all very important features that employees are encouraged to use to do their jobs.

Customer Account: Similar to the employee account class, Customer Account inherits from user account so there is a generalization arrow connecting the two. A customer account is just one form of a user account. Customer Account is where customers can view important things such as their rental history, payment history, and payment methods. This information allows customers to be able to view their past business with the rental company and should be stored in lists. Next, there is an attribute to check if the customer has a valid driver's license with a boolean return type. The final 3 attributes in this class all deal with the customer being able to enable their location to see nearby rental places. We have longitude and latitude which are given as floats, and a separate attribute that tells us whether the GPS is turned on or not. For operations, we have methods to add a credit card and remove a credit card. For these, we pass in "card" and return void. There is also an operation with a boolean return type that allows the customer to enable the GPS so that they can view nearby locations. We also added an operation that returns nothing to approve the customer's driver's license and verify it. Lastly, the getLocation method takes in the customer's latitude and longitude, or current position, and grabs their location. This doesn't need to return anything.

Development Plan and Timeline

Step 1: Planning and Brainstorming, assigning roles and responsibilities to team members

This step should take about 2-3 weeks. It is critical to our operation because we will be defining the fundamentals of the application, and will work as our framework, a.k.a. guide going forward. This will include everything from application requirements to defining core functionalities of the car rental system such as booking and payment. Here, we will also assign responsibilities to the team based on their skill sets.

Every team needs a leader, we will elect a project manager to make difficult decisions and coordinate with everyone. Since this is an application for commercial use, there will be a frontend and backend part. To save costs, we will assign one team member to work on the front end, and another team member to work on the backend. The frontend team will be designing the app interface to improve user experience, while the backend developer will be building databases, and writing logic to make the code work. Since the backend is entirely dedicated to one team member, we will hire a quality assurance team to make sure that our product is safe for commercial use. This means they will be testing for bugs and confirming that the functionalities align with our project specifications. Finally, to make sure our financials and balance sheets are cash-flowing, we will hire an economics/marketing professional to help us reach out to more customers and optimize our overall performance based on customer feedback.

Step 2: Focusing on the user interface

Having a good user interface and an overall idea of how to organize where key components are located, will help us establish a good overview on how to tackle the backend challenges. Here, we will be dedicating about 3-4 weeks to design the user interface of the application. We will be creating roadmaps of the different paths the user could take to reach a desired destination, optimizing as we get along. The UI designer will be figuring out the best color scheme, font, icons, etc.

Step 3: Backend Development

With a good user interface, the backend developer will have a clear understanding of how they should be implementing the features. Since core functionalities are developed by the backend, we will allocate at least 8 weeks to get it completed. Here, we will need a robust database to safely secure user information and meet industry standards for security. We will be implementing the UML diagram classes and functionalities, such as vehicle inventory, reservations, and payments. Furthermore, these implementations will require us to build APIs for easy access and to be able to link our backend with our front end. Backend development typically results in the most bugs due to logic errors; therefore, we will be allocated 1-2 weeks to write out unit tests to have 100% code coverage, making sure it also integrates fully with our frontend components.

Step 4: Frontend Development

In step 2, we designed the overall look and feel of the application but did not get in-depth about how we were going to implement it. In step 4 of the development process, we will be connecting the front-end interfaces to backend APIs. This means developing UI components working with backend functionalities to enhance user experience. Examples include search filters and optimized form filling. Furthermore, at this stage of the development process, we want to make sure performance is fast and responsive. This process will take about 4-6 weeks.

Step 5: Testing and Quality Assurance

We have functional requirements specified in our specification requirements, in this phase, we will be validating each feature to make sure that it is working as intended and identify bugs. Furthermore, we will be hiring anonymous users to test the app and provide feedback on its usability and experience. We will also be hiring penetration testers to perform security tests to ensure user data is properly encrypted and safely stored in our databases, especially for our payment functionalities. This process will take about 2-4 weeks.

Step 6: Launch and Maintenance

We did it! The app is ready to be released. To ensure last-minute quality control, we will be releasing the app to a small group of people to identify last-minute issues. Once that successfully passes, we will deploy the app to production and our marketing team will be in action, optimizing the app based on the feedback we get from customers. This process shouldn't take more than 1 week. However, we are mindful there are issues that we might have not thought of and maintenance will be a common thing that has no deadline and will be ongoing. In this phase, we will be monitoring user feedback, fixing bugs, and rolling out new features based on user feedback.

Testing Feature 1: Reserving a car

Unit test 1: getMake()

```
Car c
c.carMake = “Toyota”
c.carModel = "Corolla"
c.carYear = "2003"
c.carColor = "Black"
c.originalCost = 10,000
c.carMileage = "75,000"
```

```
make=c.getMake()

if(make == “Toyota”)
        Return PASS
Else
        Return FAIL
```

Output: Pass or True, car make was successfully retrieved.

Unit test 2: getMileage()

```
Car c
c.carMake = “Toyota”
c.carModel = "Corolla"
c.carYear = "2003"
c.carColor = "Black"
c.originalCost = 10,000
c.carMileage = "75,000"

mileage = c.getMileage();
if (mileage == “75,000”)
        Return PASS
Else
        Return FAIL
```

Output: Pass or True, car mileage was successfully retrieved.

Integration Test 1: makeReservation(car) with 1 reservation

```
// Create a car object
Car c
c.carMake = “Toyota”
c.carModel = "Corolla"
c.carYear = "2003"
c.carColor = "Black"
c.originalCost = 10,000
c.carMileage = "75,000"
```

```
// Create a reservation object
Reservation reservation
reservation.makeReservation(car)
reservationStatus = reservation.isReserved(car)

// Check if the reservation was successfully executed
if (reservationStatus == true) {
        Return PASS
} else
        Return FAIL
```

The above is an integration test because it involves two classes, the car and the reservation class. It is an integration test as it checks that the methods can be called in sequence and function correctly. It works by first declaring an object, car, with the attributes make, model, year, color, cost, and mileage. There's a reservation class that takes in a car and makes the reservation for the particular car using the method, makeReservation. This process changes the state of the reservation object, verifying that the integration works as expected and if the state transition (a.k.a. from false to true) occurs.

Integration Test 2: makeReservation(car) with 2 or more reservation

```
// Create a car object
Car c
c.carMake = “Toyota”
c.carModel = "Corolla"
c.carYear = "2003"
c.carColor = "Black"
c.originalCost = 10,000
c.carMileage = "75,000"

// Create a reservation object
Reservation reservation_1
reservation_1.makeReservation(car)
reservation_1_status = reservation_1.isReserved(car)

Reservation reservation_2
reservation_2.makeReservation(car)
reservation_2_status = reservation_2.isReserved(car)

if (reservation_1_status) {
        if (!reservation_2_status) {
                Return PASS
        } else
                Return FAIL
} else {
        Return FAIL
}
```

In this integration test, we are checking for edge cases. Here, we set up one car object and its attributes. We create two reservation objects, namely reservation_1 and reservation_2.
We first check if the first reservation was successful with the outermost if loop, under the condition that reservation_2_status must be false. If reservation_2_status is true, we have failed the test because you can not possibly have two reservations on the same car. We are assuming that the cars are booked for the same period. On the other hand, if it's true, it means that we have not successfully executed reservation_2, meaning we only have 1 reservation for that car, which is a pass. The outermost else statement checks if reservation_1 has been successful, if it is not, there's no point in checking reservation_2.

System Test 1:
The user logs in to the rental car system with a valid email and password. The user will be prompted to enable location services and the system will verify if location services are active. The user will navigate to the reservation system where it will display an array of cars for the user to choose from. The user will then choose a car from the list of available options. After the user enters reservation details, the system will make a reservation for that particular car. No other users can now book that car, and the user is prohibited from booking the same car again. The user will then enter payment information and after the system verifies payment has been made successfully, the user will receive a confirmation of that reservation.

Explanation: This first system test explains how the user will successfully book a rental from start to finish. They are first required to sign in to the BeAvis website so that they can start browsing. Once they have found a car that they think is the right fit, they can make the reservation for the date and time of their choosing. This prohibits other people from booking the same car and then the test takes the user through the payment process. Once the payment process goes through, the user is sent confirmation that their reservation has been set.

System Test 2:
The user logins in a similar way to system test 1. When the user enters reservation details, such as start and end times. If the user enters an end date that is earlier than the start date, the system will display an error message telling the user about the error and how to fix it. If the user tries to

book the same car twice at the same period, the system will also display an error message telling the user that action is not allowed. The make reservation button will be grayed out until a valid reservation has been made. Once a valid reservation has been entered, the user can make payment to the reservation, and they will receive a confirmation once the system can verify that the payment has the correct details.

Explanation: This system test deals with the same feature, but accounts for a user who might enter in the wrong information on accident. For example, a user may accidentally put the start date as 10/25 and the end date as 10/2 when they really meant to set it as 11/2. In this instance, the system will give them an error message and ask them to try again. It also covers an additional case of two users booking the same car at the same date. In order to prevent this from happening, the system will acknowledge when a car has been reserved and disallow any other users from booking that same car for the same dates. Similarly to our last system test, the payment process is completed again.

Testing Feature 2: Finding the closest customer

Unit test:

```
login(email_address, password): bool
```

Input

```
UserAccount account
account.Username = "DenverNuggetsFan"
account.email_address = "testingtesting123@gmail.com"
account.password = "JokicForMVP123!"
account.operatingSystem = "mac"

if (account.isUserRegistered()) {
        // allow for login (returns true)
} else {
        // throw an error, telling the user their account is not registered with the database
}
```

Output

True (Successful Login)

Input

```
UserAccount account
account.Username = "DenverNuggetsFan"
account.email_address = "testingtesting123@gmail.com"
account.password = ""
account.operatingSystem = "mac"

if(account.password.isEmpty()){
        Return FAIL //blank password is not valid
}else {
        Return PASS
}
if (account.isUserRegistered()) {
        // allow for login (returns true)
} else {
        // throw an error, telling the user their account is not registered with the database
}
```

OUTPUT
False (Login Failure)

First, we wanted to ensure that our login feature worked properly so that customers will be able to access our system. We then formulated possible inputs for the necessary account information such as username, email address, password, and what operating system the user was using. Our first test case accounts for a user who passes in correct account information. Once the user passes in their information the system simply checks with the database to make sure that they are already registered. If they do not already have an existing account, they will be prompted with an error message.
In our second test case, we accounted for somebody who put in invalid account information such as a blank password. After establishing the blank password, the system should check if the password is empty. In this case, we only tested it for the password, but it should check to see if the email address is empty as well since they will need that to log in too. Since the password was empty, the expected output would be false since the login was unsuccessful.

Integration test:

```
getClosestCustomer(): void

CustomerAccount account
account.Username = “DenverNuggetsFan”
account.operatingSystem = “mac” // Different OS uses different programming languages to track
user location
account.getLocation (127.01, 89.19)
```

```
if (account.getLocation() == NULL) {
        throw turnOnGPSExecption("Please turn on location tracking!");
        Return FAIL
}
```

For this method, we are checking to see if the customer has turned on their GPS tracking. If they haven't, the method "getLocation" should return NULL. Therefore, it should not continue executing the getClosestCustomer() because the software doesn't know the position of the customer. It will throw an exception to the customer telling them to turn on location tracking. It returns FAIL because we were expecting a longitude and latitude.

```
if (account.getUsername != "") {
        Return PASS
} else {
        Return FAIL
}
```

For this method, we are checking to see if the username is an empty string. If it is an empty string, it means that this is an invalid account and should not be allowed to go any further with the booking process. If on the other hand, it is not equal to an empty string, it means that it is a valid account and can move on with the booking process.

```
//Tests for closestCustomer
// Hypothetical
Set account location to (127.01, 89.19)
if (platform.getClosestCustomer() != account.Username) {
        Return FAIL
} else {
        Return PASS
}
```

We assume the account location to be that of the customer. We are testing to see if the customer is the closest to the account location, and it should be obvious that this is true because the account location and the customer location are the same. Therefore, if the closest customer is not account.Username, this test will fail; otherwise, it will pass.

```
// Multiple Customers
Set Customers [] = {
    -   Customer 1's location is (100, 90)
    -   Customer 2's location is (200, 80)
    -   Customer 3's location is (500, 1000)
```

Current Location is (99, 89)

```
if (platform.getClosestCustomer(account.getLocation) == Customer 1) {
        Return PASS
} else {
        Return FAIL
}
```

To conclude this test, we came up with 3 possible locations for different customers as well as the current location. To ensure that the getClosestCustomer feature works properly, we pass the locations of the different customers into it. The expected outcome should be that the closest customer is customer 1 since they are very close to the current location. If it is customer 1, then the system will pass the test.

System test:

Test Case 1:

If a user is registered, they can proceed to login; otherwise, they will be prompted to register for an account. A user signs in to the BeAvis Car Rental System with a valid email address and password. The system will authenticate with the database to make sure that it is a valid user and that all the fields are properly filled. When they login, their location services will prompt them to turn it on. When the user goes to book a car, the system will check for the closest car determined by the user's location. If it's not on, the system will throw an error message telling the user to turn on their location services. Once location is verified, the customer can make the reservation by making payment. A confirmation/receipt will be sent to their email address.

Explanation: The first system test walks a user through signing in. If they already have an existing account then they may proceed. If not, they must register. Once the user successfully registers or logs into their existing account, the system will urge them to turn their GPS on so that the app or website may track their location. Once the user turns this setting on, the system will conveniently check for the closest available car to the customer. Once the system key's in on the closest car, the user may book a reservation and go through the payment process for that vehicle.

Test Case 2:

A user signs in to the BeAvis Car Rental System with a valid email address and password. The system ensures that they are registered and allows them to login. They have their GPS enabled so their location is taken in by the program. Unfortunately for the user, the closest car to their location is over 100 miles away which just isn't feasible. They are prompted with a message

saying that there are no rentals available nearby. They are then returned to the home screen on the website or app where it displays all available rentals, but none near them.
Explanation: This system tests the same feature, but makes sure that there are nearby vehicles that can be rented. In this case, the system checked for nearby rentals based off of the users location, but the closest one was over 100 miles away. Since this isn't realistic, the system app or website simply tells the user that there are no rentals available as they are all too far away. They can continue browsing the rentals screen, but the location feature for that customer is useless.

Updated Design Specifications

For our UML Diagram, we added the following methods:

- isUserRegistered(): bool
- isReserved(car): bool
- getClosestCustomer: customerAccount
- cancelReservation(car): void

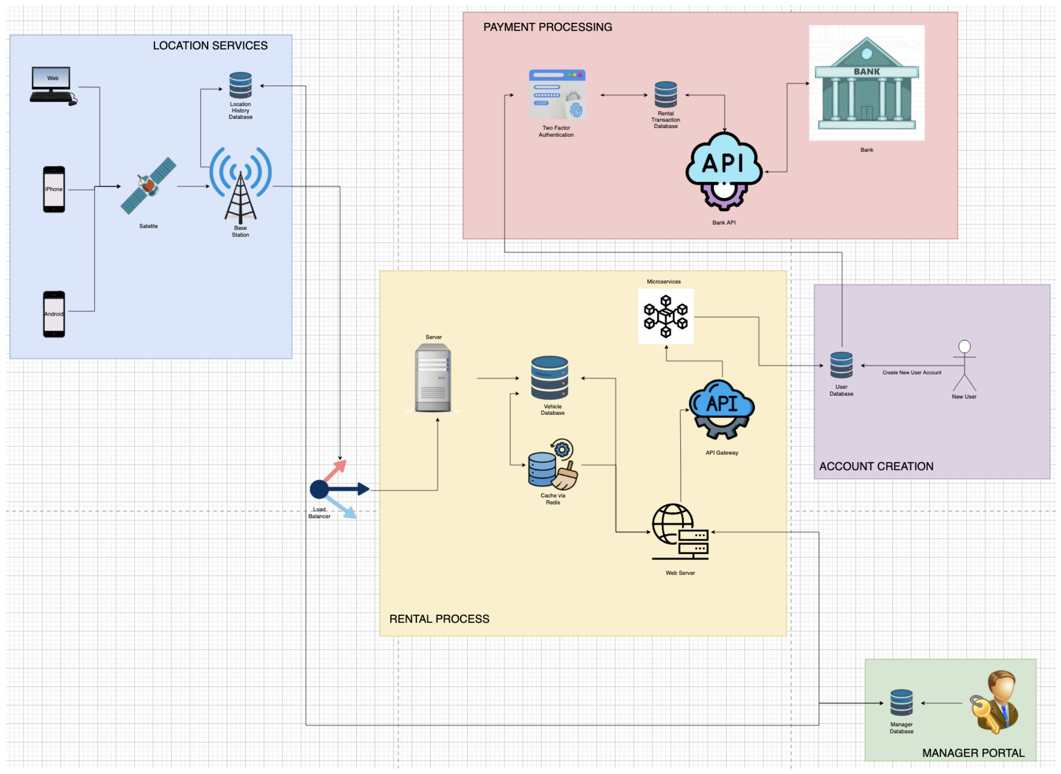
We added isUserRegistered() to check if a user is already registered in the system. This method is important because before allowing a user to log in, the system can call this method to determine if the user needs to register. After receiving a user's information, the system must cross-reference with the database to make sure that an account exists with that information.

For isReserved(car), the method checks to see if a specified car is reserved or not. This method helps our test cases because if a customer books the same car twice for the same time, it is an invalid reservation and the system should not allow it. This is necessary to ensure that the booking system operates properly.

Next, we changed the return type of getClosestCustomer from bool to customerAccount. The system pinpoints the location of all of its customers who have their GPS/location enabled and grabs the closest customer to that specific location, since that would be the most convenient. Rather than returning a boolean, getClosestCustomer should return the information and account of the user who was the closest, so the system acknowledges it.

Finally, for cancelReservation(car), we initially had a makeReservation(car), we quickly figured that if you can make a reservation, there's got to be a way to cancel it. This is important for the customers because mistakes happen and it is not fair for the customer to not be able to manage their reservation effectively.

Updated Software Architecture Diagram



Updated Software Architecture Diagram Description

In databases, compartmentalizing different groups of data is instrumental in keeping everything organized, easy to access, and clear to tell apart. Furthermore, separating databases allows a business to enhance scalability. Say, for example, our rental car software system that handles user accounts daily. Putting all the data into one centralized database and having to read everything from user information to vehicle details can slow down the software. However, if these functionalities were separated into different databases, each specific database could be optimized for its task. In this case, we can optimize the user accounts database for read-heavy operations to ensure the quickest time complexity for accessing a customer's profile and authenticating their account.

Databases:

In our software architecture diagram, we added a user database for the account creation module. The functionality is to store and retrieve user account details. By keeping the user’s database in a specialized database, we minimize the loss of sensitive PII if a bad actor manages to compromise one part of the company’s system. Whereas, if we kept all the information in one database, a single compromise would lead to the loss of all information stored in the database. We added a vehicle database that connects the rental process to provide vehicle availability and other information regarding the vehicle. The rental transaction database ensures payment and transaction data are associated. The location history database connects to the location services to store vehicle location and customer location information. The manager has a database that provides them with access to all relevant databases necessary for analytics and providing future guidance.

Additional Components:

We added the API Gateway because it is a connection between clients and backend services to improve security and organize workflow. The microservices architecture is necessary for modularizing components like user, vehicle, and payment management for scalability and independent development. The cache will use Redis, a NoSQL database, to store frequently accessed information to reduce the amount of work the database does, in doing so, improving response time.

Data Management Strategy

In a car rental software system, where we want to get the customer in and out, as efficiently as possible, we want a database to have well-organized data. In an SQL database, tables keep our data structured, making it easy for customers to navigate through our platform. Furthermore, the amount of queries you can make with a SQL database is more than that of a NoSQL database, meaning customers will have access to more advanced functionality to improve their user experience. In addition, this is useful for generating detailed analytics for the manager, helping them make informed business decisions. SQL is also widely used by professionals around the world, not only is it reliable, but it is also a standard language many developers use, making it easy to find support if required. For our system especially, security is of the utmost importance, since we are handling with Sensitive PII, SQL databases come with features that help meet regulatory compliance requirements as well and SQL has a strong backup and recovery mechanism, making sure customer data is safe and protected.

Using SQL does not mean we can't use NoSQL. There might be additional features we want to implement in our car rental system that might not be as important, good to have, but not of a concern if it went missing. For example, say we want to keep track of how many times a particular car was rented out. We can use a NoSQL database to store these nonessential, cluttered data. Furthermore, sometimes, based on the user's activity, our car rental system algorithm may detect that a user likes a certain brand/type of car. To store these temporary insights, we can use a NoSQL database to store cache, which is also fast to access.

While we will use SQL to store the bulk of our data, NoSQL comes into play when we want to add miscellaneous features.

Below are some possible SQL tables for the various databases that exist within our Software Architecture Diagram.

Customer SQL Table:

CustomerID	CustomerName	LastName	City	Phone	Address	ZipCode	Email	Password
1	Ben	Shen	San Francisco	XXX XXX XXXX	123 Main Street	12345	bob@gmail.com	1234
2	Joe	Smith	Los Angeles	XXX XXX XXXX	345 Broadway Street	12346	james@gmail.com	1111
3	Princess	Peach	San Diego	XXX XXX XXXX	9872 El Cajon Blvd	12347	william@outlook.com	9999

Vehicle SQL Table:

VehicleID	Make	Model	Year	LicensePlate	Color	VIN	Mileage	Value
1A	Toyota	Corolla	2024	XXX XXX XXXX	Blue	5483270964	10000	$25,000
2A	Honda	Acord	2023	XXX XXX XXXX	White	9380149773	100000	$30,000
3A	Lamborgini	Urus	2020	XXX XXX XXXX	Black	2039959882	250000	$250,000

Rental Transaction Database:

TransactionID	CustomerID	VehicleID	Price	Time	CardType	Bank	Vendor	isSuccess
OxABCD	1	1A	$3.50	0:12	Credit	Wells Fargo	MasterCard	TRUE
OxABCDE	2	2A	$12.50	12:57	Debit	Chase	Visa	TRUE
OxABCDEF	3	3A	$2,300.50	2:34	Credit	SoFi	MasterCard	FALSE

Location Services Database:

LocationID	CustomerID	VehicleID	Latitude	Longitude	Time
1	24	1A	34	-118	1:36
2	35	2A	-23	-46	8:12
3	76	3A	100	110	3:56

Alternatives

Cloud Databases:
Cloud databases are more secure, scalable, and cost-efficient. If the business is small, a cloud database allows it to scale to meet the needs of its growing applications. Furthermore, cloud databases are managed by industry professionals with enhanced security, such as Azure by Microsoft, Google Cloud, and AWS by Amazon. For a growing business, managing a balance sheet with positive cash flow is critical to help the business stay afloat. A cloud database can reduce costs by eliminating additional hardware, software, and employees. Most cloud databases are pay-per-use subscriptions, meaning they are tailored to the needs of each business individually. However, by choosing a cloud database, you are sacrificing on-premise servers to allow for better control of backup schedules and data security. Furthermore, there will be no instance refresh in real time, meaning data may take longer to get transferred from one place to another.

Sources: Grammarly was used to proofread my work