Automotive Management Cloud System

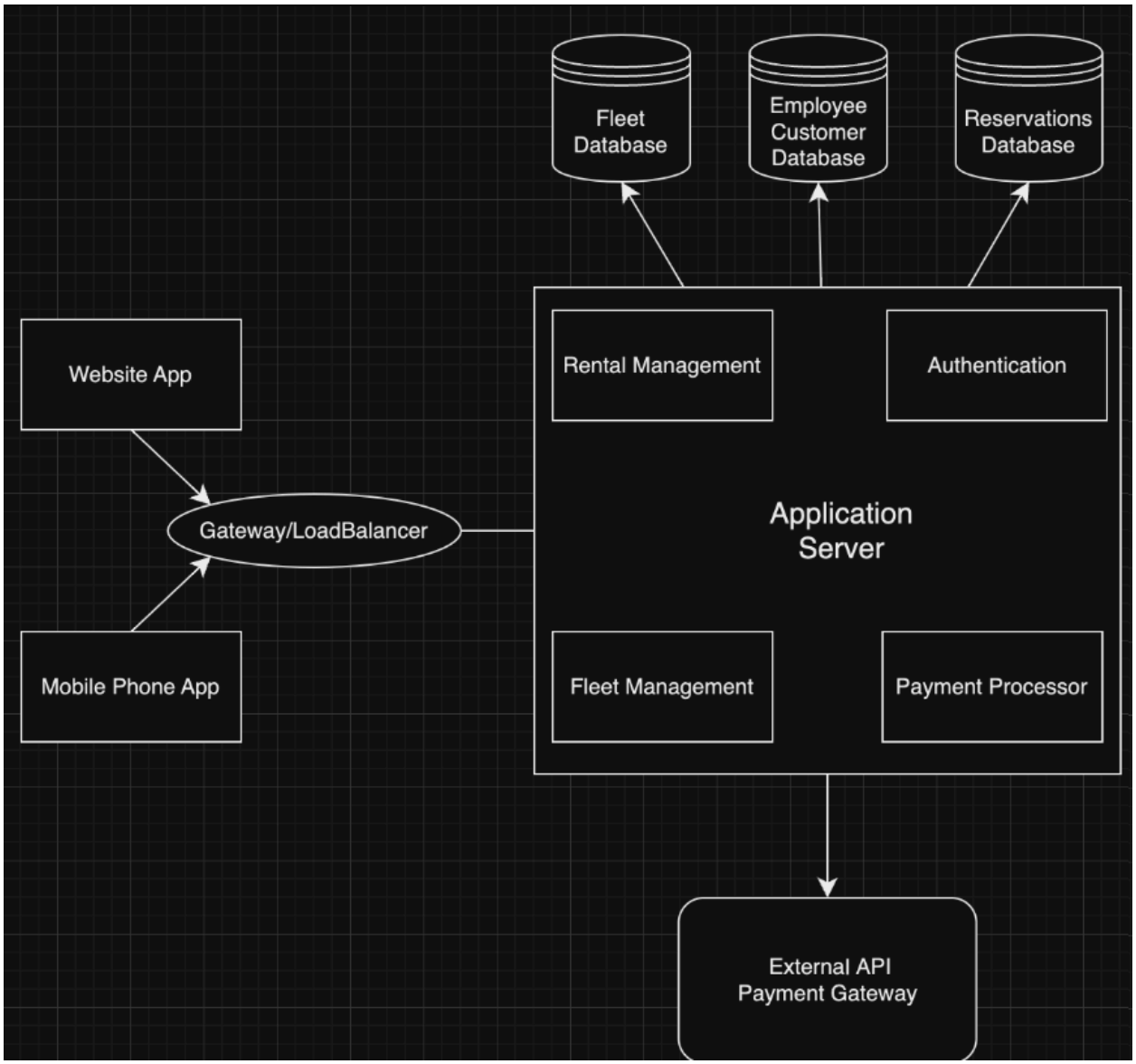
By Max Carrillo, Jeremiah Cho, and Ramil Carino

Introduction and Overview

This document outlines the functional and nonfunctional requirements for the Automotive Management Cloud System (AMCS) Software for BeAvis Car Rental Company, to modernize the car rental process and operations. Integrating the AMCS expands the capabilities of BeAvis and transitions the current manual, pen-and-paper, method to a fully digital, online platform. The AMC system aims to handle the car rental business for customer applications and reservations, maintaining records and fleet data, and business control for BeAvis Car Rental Company. Integration of the AMC system reduces operating costs, minimizes record errors and inaccuracies, and modernizes the company for customers and employees to be accessed by mobile and website applications.

Automotive Management Cloud System (AMCS) software provides a suite of online services to replace traditional paperwork car rental processes with an efficient and user-friendly application, accessible across iOS and Android devices, as well as a website.

2) Software Architecture Diagram Overview



Architecture Diagram Description

The AMCS Software Architecture diagram for BeAvis Car Rental. There are two access points for the AMCS application server. Through the website application and mobile application (across iOS and Android Devices) the user is able to access the application server to initiate rental process tasks including vehicle browsing and reservation services, user account management, and inventory and reservation management, as well as make payments. These functions are displayed as arrows or connectors to the Gateway and communicate each function between the application and application server. The application server has four different processors to manage the data and databases that allow it to continuously be up to date. The application server handles payments in particular that utilizes an external API Gateway for

payments.

*This diagram is now out of date—please refer to the revised SWA diagram found later in this document.

Gateway / Load Balancer API - This component ensures that computationally intensive workloads are equally distributed across multiple servers to avoid potential outages or slowdowns.

Application Server - This component handles business logic for each portion of the application. It contains microservices that handle each separate purpose of the application. For example, the Authentication service would handle user account creation, email verification, and logging in/out of the apps.

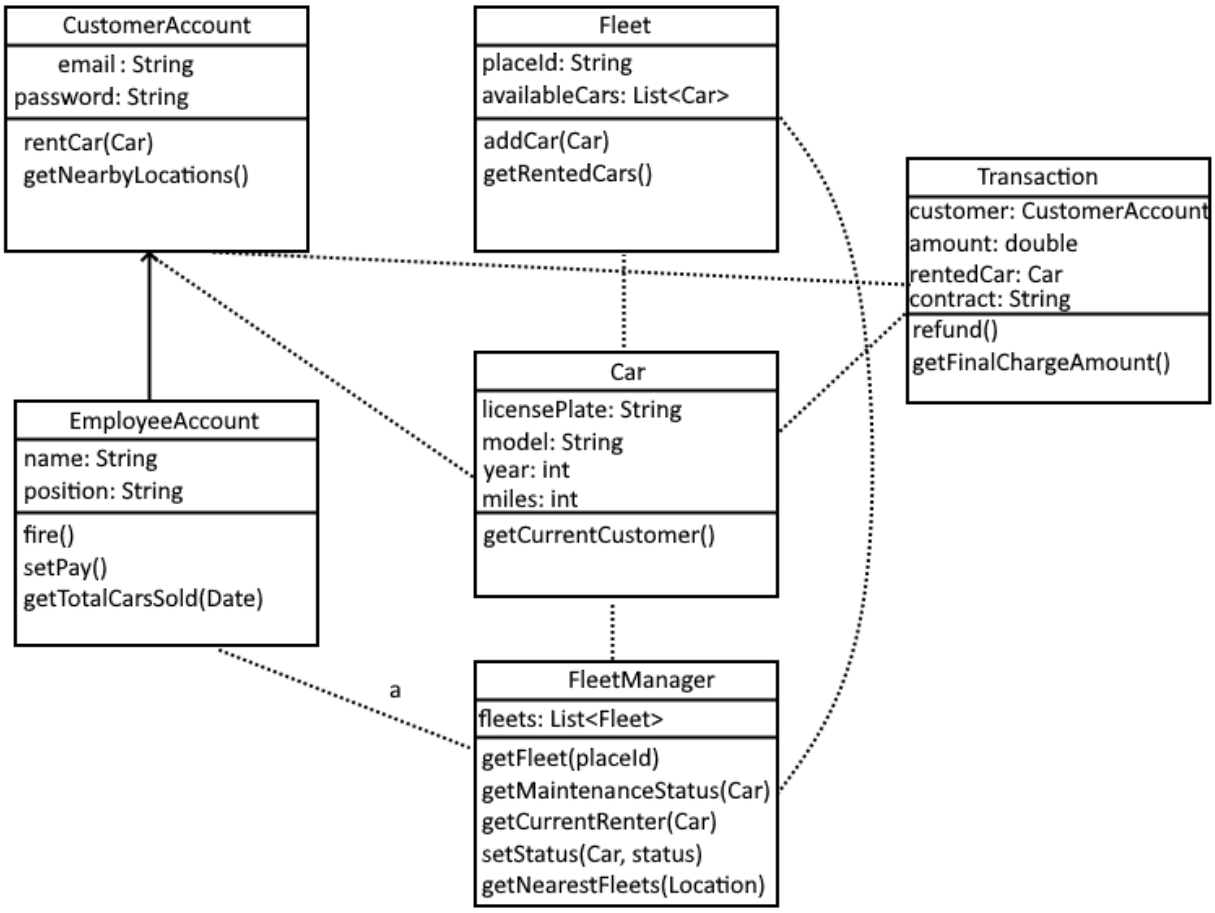
External API Payment Gateway - Interactions between the application server and a third-party service that securely manages all payments. This is necessary to ensure our application uses a reputable payment handler that users can trust with sensitive payment information.

Fleet Database - Stores records about each fleet of cars owned by BeAvis. Access to this information is guarded by the Server Application.

Employee Customer Database - Stores records about employees and customers, including all rental, contract, and payment histories.

Reservations Database - Stores current information about which cars are currently being rented and which cars are available for rent.

UML Class Diagram



CustomerAccount - Stores information about a customer that can rent cars.

attributes: email and password used to sign into the account. Email must be verified through the application server.

operations:

- rentCar(Car): Rents a car. Creates a new Transaction capturing information such as who is renting a car, the amount the rental costs before local taxes, payment processing, and other unpredictable fees, the car that was rented, and a String containing the location of a Contract PDF on the Application server.
- getNearbyLocations(): Returns a list of nearby Fleets in the vicinity of the user based on their GPS location.

EmployeeAccount - Stores information about an employee. Is a subclass of CustomerAccount, as employees also have an email and password.
attributes:

- name: the first and last name of the employee
- position: the specific job position or role the employee has within BeAvis.

operations:

- fire(): Fires the employee from BeeAvis. Can only be performed by employees with management roles.
- setPay(): Sets the hourly pay rate for this employee. Can only be performed by employees with management roles.
- getTotalCarsSold(Date): Gets the number of cars an employee sold on a given day so that commissions may be calculated.

Car - Stores information about a car owned by the BeAvis company.
attributes:

- licensePlate: The license plate of the given car.
- model: The model of the given. For example, Honda Civic.
- year: The year of the car's model. For example, 2005.
- miles: The total miles driven in this car.

operations:

- getCurrentCustomer(): Returns the CustomerAccount of the customer who is currently renting this car, or none if this car is not being rented.

Fleet - Represents a fleet of cars, and is associated with a physical location where rentals would usually be made on pen and paper before introduction of this service.
attributes:

- placeId: The ID of the physical location this fleet is kept at. May correspond to a place ID from Google Places API, for example.
- availableCars: A list of all cars that may or may not be rented from this fleet.

operations:

- addCar: Adds a car to this fleet. Used when a new fleet is created or a car is purchased by a certain location that wants to expand this fleet.
- getRentedCars: Returns a list of cars that are already being rented by a customer at the current moment.

FleetManager - Class that stores information about all BeAvis fleets across all physical locations. Can be used by management to control fleets of cars or manage cars across different fleets.
attributes:

- fleets: The list of all fleets owned by BeAvis.

operations:

- getFleet(placeId): Gets all fleets associated with the physical location referenced by the given placeId.
- getMaintenanceStatus(Car): Gets the maintenance status of the given car, such as if the car is in working condition or not, and how long a car is estimated to be out of service before being safe to reintroduce into the fleets.
- getCurrentRender(Car): Determines the current customer renting the given car, or returns none if nobody is renting the car.
- setStatus(Car): Setter that changes the maintenance status of a given car.
- getNearestFleets(Location): Given a GPS location, returns a list of fleets in order of increasing physical distance from the location.

Transaction - Stores information about a specific time when a car was rented.
attributes:

- customer: The account of the customer who was involved in this transaction.
- amount: The cost of the transaction to the customer.
- rentedCar: The Car that was rented by the customer in this transaction.
- contract: A String pointing to a PDF for the contract of this transaction on the Application Server's file storage.

operations:

- refund(): Returns all the money paid by the customer to the customer. Can only be performed by management in the case where they would like to reimburse the customer for a faulty rental.
- getFinalChargeAmount(): Calculates the actual cost to the customer of this transaction based on local fees and taxes as well as payment processing fees, if credit card was used.

Development Plan

The development of the AMCS Software involved the collaborative minds of three talented individuals, each contributing to every part of the project. From the Introduction and Overview to the Software Architecture and UML Diagrams, everyone’s ideas, opinions and work were valued. This method ensured each team member stayed

engaged throughout the project. Of course, there were days where priorities did not align, however, that was easily mediated through the trust and mutual understanding our contract was founded on. Each day, the team tackled every segment of the project from the Introduction and Overview to the Software Architecture design and descriptions, and lastly the UML design and description to give each person a responsibility and a voice.

Five Key Phases

1. Requirements Gathering (3 Weeks)

   This first phase involves the members of BeAvis Car Rental Company to create a guideline for functional and functional requirements for the system.

2. System Design (3 Weeks)

   This second phase is to create a detailed system architecture diagram and UML diagram that include key components and their interactions,

3. Development (5 Weeks)

   The third phase focuses on building the software for AMCS with the thought of the car rental industry at the core for front end mobile app and website design, as well as the controls of management for the fleet to customer information.

4. Testing (3 Weeks)

   The fourth phase is the testing phase for the key components and ensures the communication and result meets the clients requirements and the expected outputs. By using structural and functional testing, the software undergoes rigorous testing for the validation and verification this system strives for.

5. Launching and Maintaining (1 Week)

   The final phase is a confident live launch of the AMCS as well as training for BeAvis staff and management, as well as establishing maintenance for stability and software updates.

The Automotive Management Cloud System (AMCS) software is the transformative gateway designed to modernize the car rental industry for BeAvis Car Rental Company. By transforming from a traditional manual paperwork system to a fully digital platform, AMCS provides a suite of online services focused on a seamless and efficient experience for both customers and employees aimed at enhancing business capabilities, reducing operational costs, and minimizing errors. AMCS offers customers a user-friendly interface across mobile and web platforms for browsing vehicles, creating reservations, and managing accounts, while giving comprehensive control to BeAvis for inventory management, reservation handling, and payment processing. AMCS is capable of growing with BeAvis business with the flexibility to add and alter

any features, added security to reduce the risk of fraudulent transactions, scalability for large-scale reporting and system loads, and many more.

We have decided that the existing SWA and UML diagrams need no further changes. In our SWA diagram, there are clear access points to the server, which initiates the user to the rental process task, three different databases that allow for continuous updates, and a gateway for payments. The UML class diagram includes all the classes, attributes, and operations we need for the Car Rental System. Our descriptions specify the necessary data types, function interfaces, and parameters. The UML diagram includes arrows for all key relationships among the classes.

Test Plan 1: FleetManager Class

Targeted Features:

1. Fleet Manager (to remove or add cars from fleet inventory)
2. Car maintenance tracking (to get or set a maintenance status of a car)
3. findNearestFleet() (to get the nearest fleet according to a GPS location

Unit Test Set:

Test Case 1: Add new vehicle to the fleet

- This test is to successfully add a new vehicle and the information to the fleet inventory
- The expected output is the car and the information will be stored to the fleet's availableCars list

Code:

```
void addNewCar() {
     Car car("GasWhere", "Tesla Model Y", 2021, 30000);
     fleet.addCar(car);
     assert(fleet.getAvailableCars().size() == 1);
}
```

Explanation:

This unit test is to test the addCar() function in the Fleet class. This function is to add a car to the availableCars list of a Fleet. In the test, fleet(fleet) is called with placeId and a test car (car) with license plate, model, year and miles as attributes. Then fleet.addCar(car) is called to add the car to the availableCars list. This is verified by increasing availableCar integer size and the license plate matches to ensure addCar() is registered in the fleet.

Test Case 2: Verify License Plate

- Tests licensePlate assignment in the Car class
- The expected output is licensePlate should match the assigned value

Code:

```
void_licensePlateAssignment() {
     Car car("SDSU24", "BMW M3", 2024, 12000);
     assert(car.licensePlate == "SDSU24");
}
```

Explanation:

This unit test verifies that the licensePlate in the Car class is assigned correctly. The car(car) is assigned a specific license plate and is checked if licensePlate matches the assigned value. This test confirms that the Car sets basic properties like license plate, model, or miles.

Function Test Set:

Test Case 1: Fleet and Car integration

- Verifies the cars added to the fleet can be rented and tracked.
- The expected output is the fleet should contain rented and available cars.

Code:

```
void fleetCarIntegration() {
     Fleet fleet(4234);
     Car car("SDSU24". "BMW M3", 2024, 283000);
     CustomerAccount customer("customer@sdsu.edu", "password");

     fleet.addCar(car)
     car.rentCar(customer);
     assert(fleet.getRentedCars().size() == 1);
     assert(fleet.getAvailableCars().size() == 1);
}
```

Explanation:

This integration test checks that Fleet and Car classes interact correctly with the car rentals. A fleet(fleet), a car(car), and a customer(customer) is created and is placed accordingly. The car is added to the fleet and is rented to a customer using car.rentCar(Customer). This test checks the rented car returned by fleet.getRentedCars() and the car is included in the availableCars list in fleet to confirm that both Fleet and Car class contain records correctly when cars are being rented.

System Test Set:

Test Case 1: Fleet Management System

- This test validates a car can be added, rented by a customer, and return maintenance status for FleetManager
- The expected output for this full system test is adding a car, renting, checking status, and verifying attributes.

Code:

```
void fleetSystem() {
      FleetManager manager;
      Fleet fleet(8249);
      Car car("SDSU24", "Tesla Model 3", 2023, 3000);
      CustomerAccount customer("elonmusk@sdsu.com", "nvidia29$");

      fleet.addCar(car);
      manager,setStatus(car, "Available");

      assert(fleet.getAvailableCars().size() == 1);
      assert(manager.getMaintenanceStatus(car) == "Available);
}
```

Explanation:

This system test verifies adding a car to a fleet, setting its maintenance status, and confirming availability to rent. The FleetManager manages fleets and maintenance statuses according to the Fleet location. A Car and CustomerAccount represent the available cars for rent and a customer account. The car is added to the fleet and the status is set to "Available" using the FleetManager. The availableCar size increases and sets the added car as available. This test ensures the FleetManager and Fleet communicate correctly in adding cars and tracking their maintenance status.

Test Plan 2: EmployeeAccount Class

Targeted Features:

1. Login functionality (email and password verification) + position check
2. fire() operation (fires the employee)
3. setPay() operation (set hourly pay rate for the employee)
4. getTotalCarsSold(Date) operation (gets numbers of cars an employee sold on given day for commissions)

***Unit Test Set*:**

Test Case 1: Verify Email Format + Valid Position

- Tests if the system correctly validates an email format + Must be one of the listed positions (e.g. manager, cashier, retail)
- Expected Output: System should return false for invalid email formats and true for valid format + false for N/A or invalid job roles and true for existing positions

Code:

```
void emailValidation() {
     EmployeeAccount e;
     assert(e.isEmailValid("invalidemail.com") == false);
     assert(e.isEmailValid("validemail@domain.com") == true);
     assert(e.isPositionValid("N/A") == false);
     assert(e.isPositionValid("manager") == true);
}
```

Explanation:

This unit test checks that isEmailValid validates the email format, rejects the invalid format and accepts the valid one. The test also verifies isPositionValid in identifying positions by rejecting "N/A" and accepting "manager" to confirm that both the email and employee role work.

Function Test Set:

Test Case 1: Email, Role, and Pay Setup

- This test allows management with an EmployeeAccount to validate the employee's email, verify manager role and set their hourly pay rate.
- The expected output for the system to recognize valid emails and manager roles and getPay() to return the correct pay rate after setting.

Code:

```
void emailPositionandPayFunction() {
     EmployeeAccount ("elonmusk@sdsu.edu", "elonRulz", "CEO");
     assert(e.isEmailValid("elonmusk@sdsu.edu") == true);
     assert(e.isPositionValid("CEO") == true);
     e.setPay(420000.0);
     assert(e.getPay() == 420000.0);
}
```

Explanation:

This test verifies the functionality of EmployeeAccount by checking that the data stored in the object is valid and matches the expected values. It does this by first calling EmployeeAccount() with "elonmusk@sdsu.edu" as the email, "elonRulz" as the username, and "CEO" as the position. Then we check the isEmailValid function by calling it with the same email and ensuring that the email is indeed valid. We also

check if the position is valid by calling isPositionValid with the “CEO” position we initialized the EmployeeAccount with. Finally, we check that management can adjust pay of employees with the setPay function. This is done by using the accessors and mutators, getPay and setPay.

System Test Set:

Test Case 1: Employee Account Management

- This system test verifies the functionality of EmployeeAccount with login and validation to set pay and track sales.
- The expected output for the full management test should produce and confirm functionality for each management function (email, role validation, setting pay, and recording sales.)

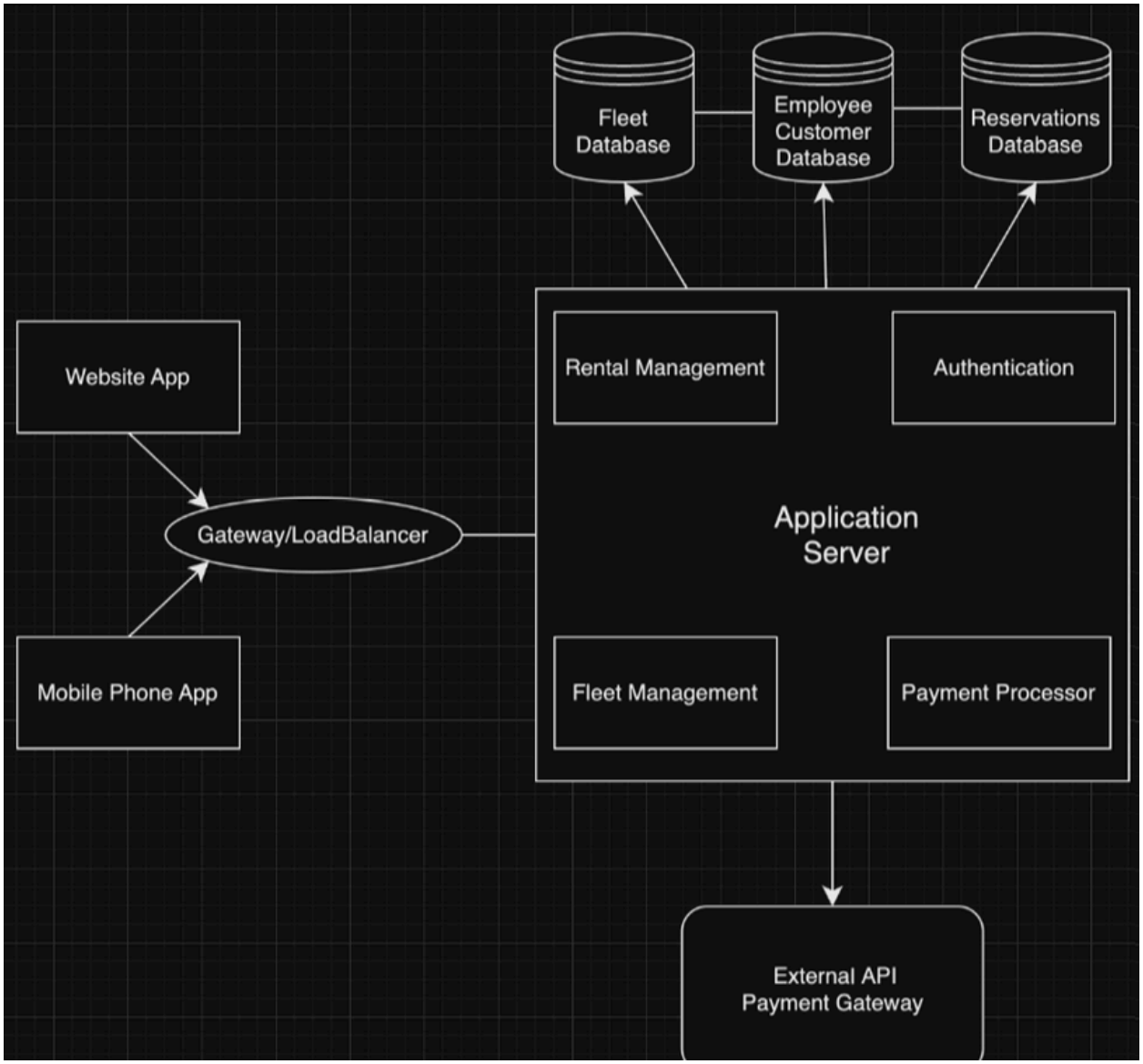
Code:

```
void employeeAccountManagement() {
     EmployeeAccount e(“elonmusk@sdsu.edu”, “elonRulz”, “CEO”);
     assert(e.isEmailValid (“elonmusl@sdsu.edu”) == true);
     asset(e.isPositionValid (“CEO”) == true);
     e.setPay(20.0);
     assert (e.getPay() == 20.0);

     e.recordSale(“2001-09-12”);
     e.recordSale(“2008-04-20”);
     e.recordSale(“2008-04-20”);
     e.recordSale(“2020-03-05);
     assert(e.getTotalCarsSold(2008-04-20) == 2);

     e.fire();
     assert(e.isActive() == true);
}
```

Explanation:

This system test is for the functionality of EmployeeAccount class by validating email format and security role access before proceeding. Management is now able to set the pay rate and confirm the rate is stored correctly. Then management is able to view sales activity by recording three sales on specific dates and verifying the total sale count is correct. EmployeeAccount can also deactivate an employee by calling fire() and checking isActive returns true that Elon Musk’s employee status is updated to fired.

Updated SWA Diagram



We have updated the Software Architecture Diagram to incorporate more detail in our database section. Specifically, we have added connections between each database to reflect foreign key relationships between objects in each database. For example, the reservations table references the employee-customer database to know which customer has placed a certain reservation. There's also a connection between the fleet and employee-customer database, as fleets are connected to the employees who are responsible for managing them.

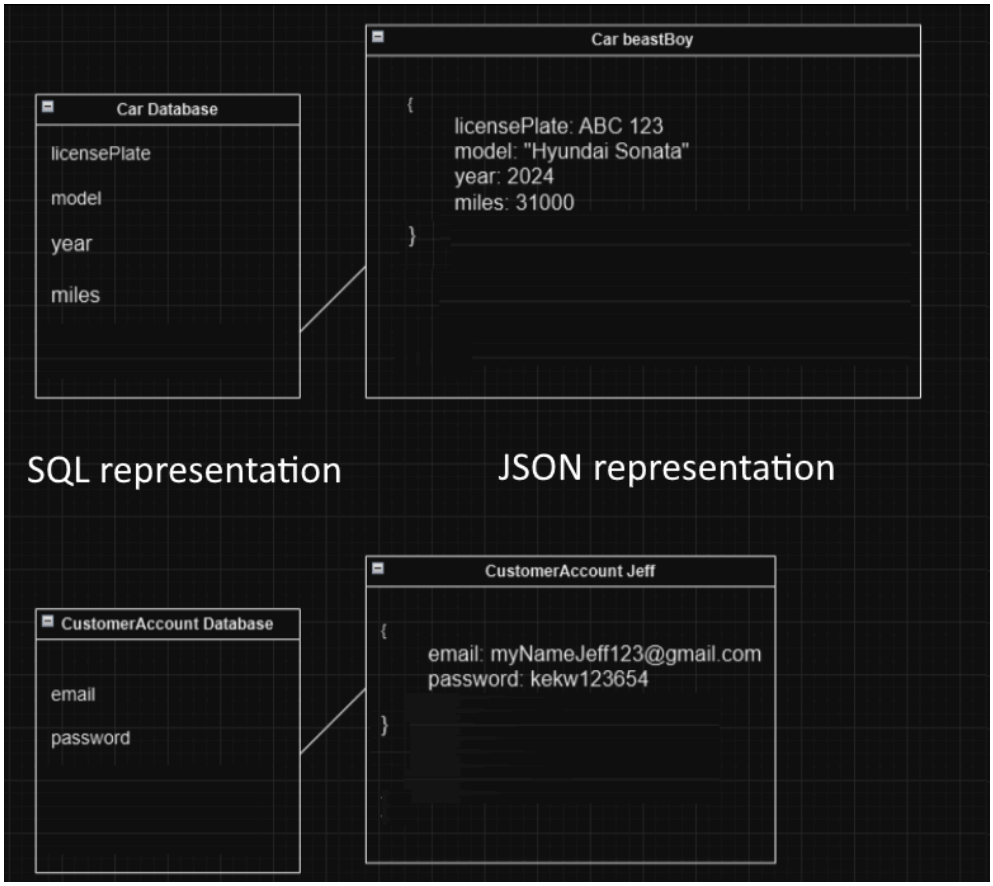
Outside of the database changes, we felt that our existing SWA diagram was sufficient

for our business needs, as the only recent changes to our design specification relate to the database management strategy, which is covered below.

Data Management Strategy

SQL

The Automotive Management Cloud System is better equipped to utilize SQL due to its structured and interrelated data. SQL's capabilities are an ideal choice for managing the system's diverse and sensitive data. This includes fleet inventory, employee accounts, reservations, customer data, and payment processings that are critical to a car rental operation. SQL's structured and relational data model ensures that these relationships are managed efficiently and reliably .As BeAvis continues to grow, so does the AMCS with SQL databases as they are designed for scalability. Database servers can easily be added to continue handling large data sets efficiently and share the load balances of distributed data without compromising reliability and efficiency. A key strength to SQL in this software system is the guarantee of reliable and consistent transaction processing through its ACID compliance. This is crucial when dealing with sensitive operations such as payment processing and managing customer information. SQL's predefined schemas and enforcing strong relationships between entities, SQL databases ensure data integrity, especially in constant updating and changing data.



Here, we show that we can take advantage of SQL's stability and robustness without complicating our data layout. For example, the car database represents each rental car as a row in the database, which means table lookups are quick and allow for transactional database modifications to improve stability. The CustomerAccount database is even simpler, and benefits from the same stability.

Trade Offs and Alternative

Automotive Management Cloud System is designed to prioritize structure, consistency, and scalability for data management through SQL. SQL is very structured with its predefined schema and ACID compliance to provide reliability in critical operations like payment processing and managing interrelated data such as customer with the car, with the reservation. Data is organized in multiple structured tables such as,
Cars table includes tracking vehicle availability and maintenance, a Customer table manages user profile and payment information, and a Reservations table would link the customer and the car database, providing clear relationships and efficiency querying. This structured approach ensures consistent and secure updates across the systems, making SQL ideal for processing sensitive and interrelated data.

Alternatively, NoSQL databases like MongoDB offer greater flexibility in schema evolution and scalability. NoSQL is efficient in handling real-time changes to data, such as simultaneously adding or removing cars and customers, ideal for fast paced business operations. NoSQL allows for efficient scalability without the constraints of SQL's robust structures. However, this flexibility comes with the tradeoff of guaranteed consistency provided by SQL. NoSQL can lead to data discrepancies, which is critical for the reliability and accuracy required in the AMCS when handling highly structured and sensitive data. SQL ensures consistency when updating multiple databases and documents, achieving a balance in relation ability, performance, and maintainability, making it a reliable and scalable solution for the company's long-term needs.
SQL ensures consistency when updating multiple databases and documents in normalized tables, achieving a balance between relationship management, performance, and maintainability, making it a reliable and scalable solution for BeAvis' long-term needs.