BeAvis Car Rental Software Design Specification

Prepared by: Adriel Guerrero, Owen Morales, Patrick Dowell

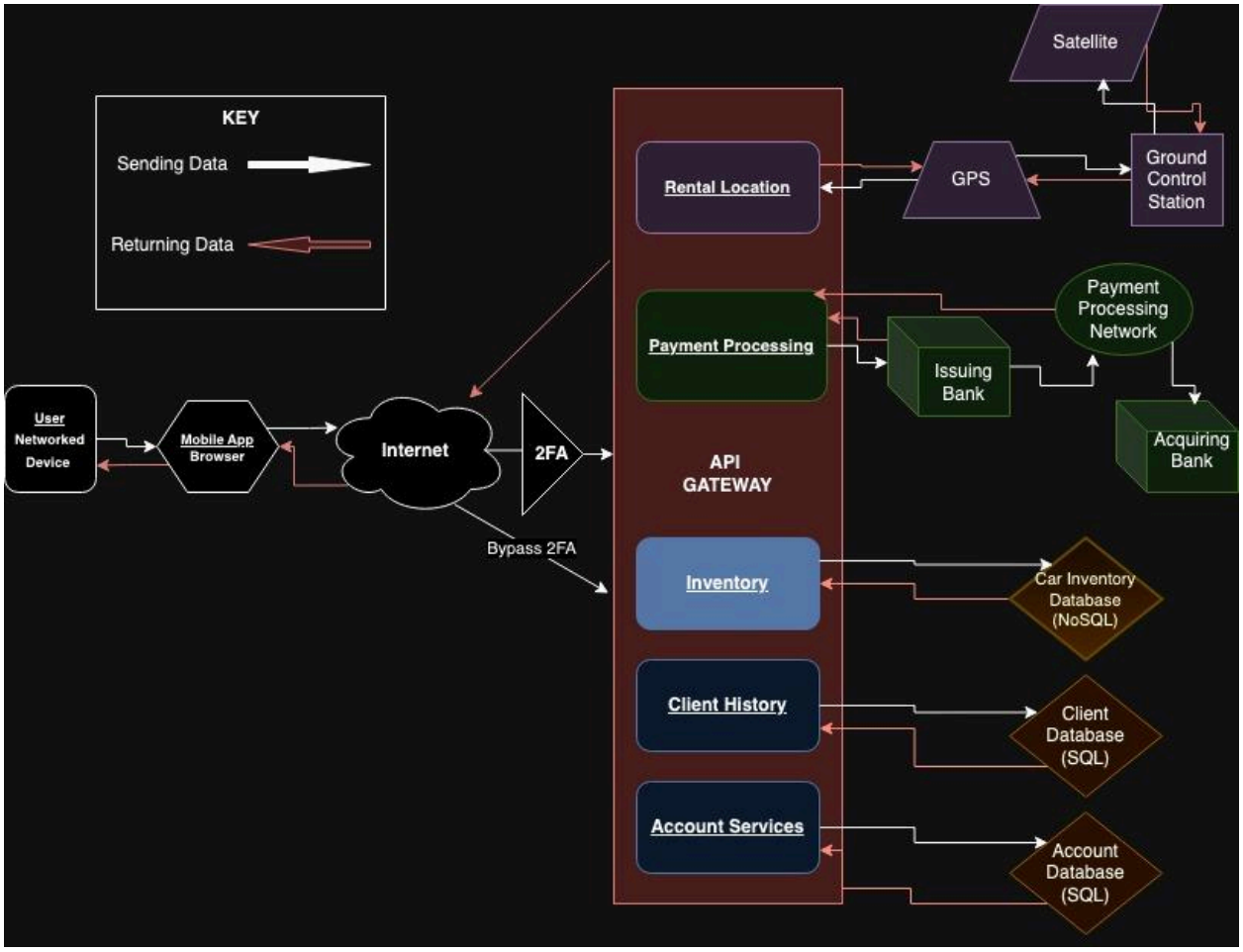
Table of contents

System Overview:

This overview explores the BeAvis Car Rental System, a contactless car rental agency designed to be completely operable via the internet. This system contains features that will grant clients, with a valid account, to perform and/or allow business transactions, GPS location, rental history access, contract e-signatures, and explore rental options. On the other end, BeAvis employees with valid user accounts will be able to access client account information, vehicle inventory and schedule routine maintenance for vehicles available.

Architectural Diagram 2.0:

The architectural diagram has been updated to replace the existing BeAvis server for the vehicle inventory system and database for client history, and account services with SQL and NoSQL database systems. The inventory vehicle system transitioned to a NoSQL database, and both client history and account services have been set up with separate SQL databases.



Architectural Overview:

The software architecture diagram for BeAvis' Car Rental starts on the left-hand side with the user devices (client or employee), which may be a cellular device, laptop, or a personal computer. Computer devices connect directly to the internet via preferred browser, while cell phones connect via an installed BeAvis application. Users have the option to utilize two-factor authentication or to connect directly to their account bypassing 2FA. Once the user has gained access to the BeAvis system API Gateway, the system can connect to different features (GPS Service, Payment Processing, Inventory, Client History, and Account Services) and then return data to the user.

The GPS feature connects to GPS, which then communicates with the Ground Control Station, which has control over satellites, and then returns the data from the satellite back to the BeAvis system, and then back to the client.

The Payment Processing feature initializes payment by connecting to the user's bank, submitting the payment from the user's bank to a Payment Processing Network, which in turn distributes the payment to BeAvis' bank as well as submitting proof of payment back to the BeAvis system.

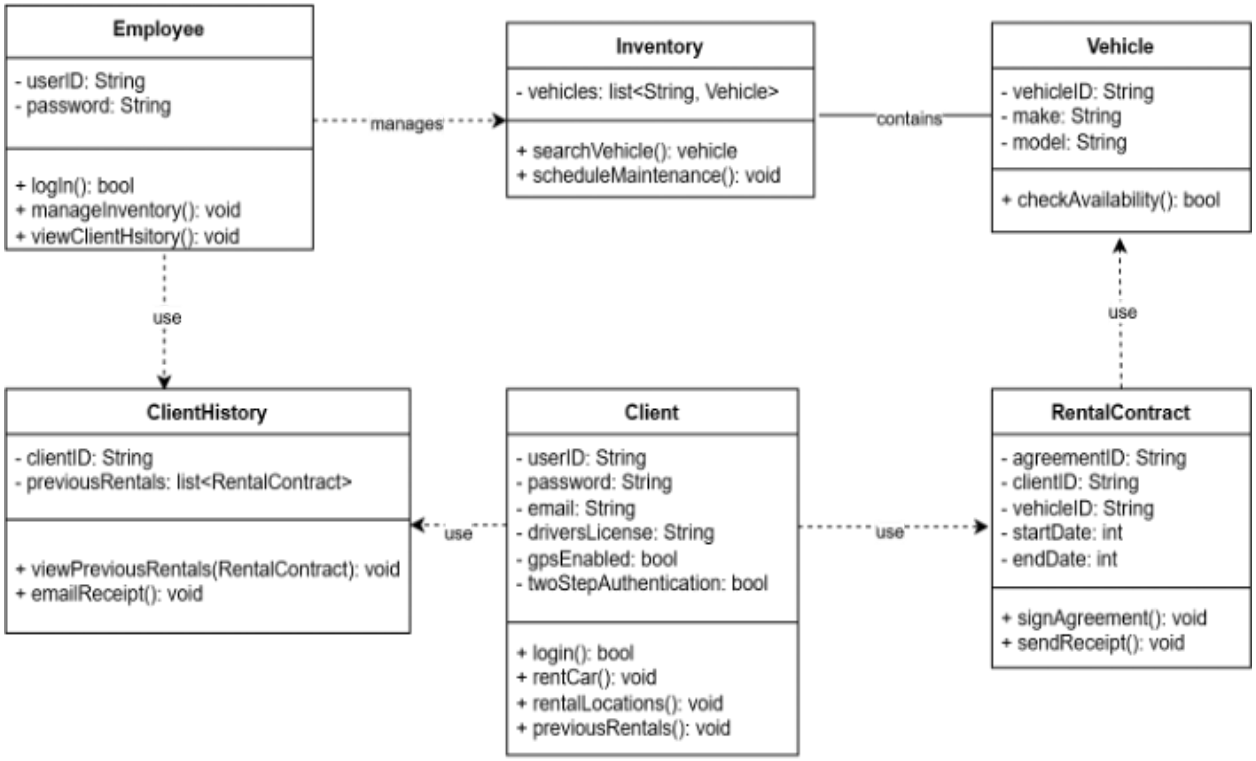
The Inventory feature connects directly to the BeAvis Server which contains all of the inventoried vehicles, allowing users to browse and select available vehicles or employees to view inventory and schedule maintenance on vehicles.

The Client History feature connects directly to the BeAvis Database which contains all of the user's previous rental contracts and which vehicles were rented out, allowing both user and employees to navigate prior agreements.

The Account Services feature connects directly to the BeAvis Database which contains all of the user and employee account information and log-in credentials. This feature allows users or employees to make an account, view their account, or make changes to their account, such as enabling GPS or 2 Factor-Authentication.

Updated UML Class Diagram:

UML Car Rental Diagram:



This is the updated UML diagram for our BeAvis car rental software system. Updates include:

- ClientHistory viewPreviousRentals now accepts an argument, RentalContract.
-RentalContract fields startDate and endDate now have integers.

Classes:

Employee– The Employee class is responsible for handling tasks that involve viewing customer purchase history and the ability to manage vehicle inventory.

Attributes -

- userID:string - Requires the employees unique userID.
- Password:string - Requires a password to access employee account.

Operations -

- login(): bool - Allows access to the account if employees credentials are valid, if credentials are invalid, account access will be denied.
- manageInventory():void - Allows employees to navigate vehicle inventory.
- viewClientHistory():void - Allows employees to search client rental purchase history.

Inventory - Contains vehicles that are available for customers to rent.

Attributes -

- vehicles: list<string, Vehicle> - Contains a list of vehicles.

Operations -

- searchVehicle(): Vehicle - Allows employee to search vehicle.
- ScheduleMaintenence():void - Lets employee schedule maintenance for vehicle.

Vehicle - Contains identifying info on vehicles in the system.

Attributes -

- vechicleID: String - Contains info on the vehicle's registered ID.
- make: String - Contains info on the make of a vehicle.
- model:String - Contains info on the model of a vehicle.

Operations -

- checkAvailablility(): bool - Allows employees to see if the vehicle is available.

Rental Contract - Contains information on rental contracts from clients.

Attributes -

- agreementID: String - The ID associated with the rental contract.
- clientID: String - The ID of the client who signed the contract..
- vehicleID: String - The vehicle string associated with the contract
- startDate: Integer - Information on when the contract starts.
- endDate: Integer - Information on when the contract ends.

Operations -

- signAgreement(): void - Allows the signage of a contract.
- sendReceipt(): void - Allows the receipt to be sent to the client after signage.

Client - Contains all information from each client.

Attributes -

- userID: String - The client's user ID for their account.
- password: String - The clients's password for their account.
- email: String - The client's email for their account
- driversLicense: String - The client's driver's license ID.
- gpsEnabled: bool - Shows if the client as gps enabled.
- twoStepAuthentication: bool - Checks if the account has properly authenticated.

Operations -

- login(): bool - Allows the Client to log in.
- rentCar(): void - Allows the client to rent a car.
- rentalLocations(): void - Allows the client to find rental locations in their area.
- previousRentals(): void - Shows the user a list of their previous rentals.

Client History - The Client History class is responsible for viewing previous rentals and emailing receipts. It does this by viewing the Client ID.

Attributes -

- clientID: String - Requires the client-specific ID in order to find their history
- perviousRentals: Array<Rental Contract> - This checks the Array to see what the previous rental contracts are.

Operations -

- viewPreviousRentals(RentalContract):void - This allows the class to access the previous rentals in order to gather the information it needs.
- emailRecepit():void - Once the information has been gathered this is used in order to email the receipt to the client.

Development Plan and Timeline:

Partitioning of tasks –

Backend Developers - The Bulk of the work will be divided among the backend developers. Programmers will be responsible for the coding of the main functions of the system. Server architectures are responsible for making sure the clients get fast responses. A security team will be tasked with securing and encrypting user info as well as two-factor authentication. The database team will ensure that the data is stored properly.

Frontend Developers - The user interface for the web and app, as well as usability, will be handled by the frontend team. The UI will ensure that the app is visually pleasing and fits the company's themes. Additionally, they will make the app as accessible as possible in order to reach the maximum amount of potential clients.

Timeline-

With scale in mind, the expected delivery of this app should be around 6 - 9 months. Based on the amount of time that is required to be invested to make it both web and app accessible. The complexity of having to make an app with user accounts, contacts, stored car information, and GPS will cause much more effort and time to be invested. Easy beta tests and implementations can be expected as early as 5 months.

Unit Tests:

In this verification test plan, we will be conducting brute force unit testing for account login security and two-factor authentication (2FA). For the account login security, we will use 5 test cases that will test different scenarios of valid and invalid credentials to ensure comprehensive functionality. However, only correct input fields for username and password should trigger the 2FA feature. The second feature will implement 6 test cases to evaluate the three verification options that are available to account holders: call, text, and email. For the call input, clients will need to physically press the requested digit by the operator on their keypad; for text input, clients will enter a passcode provided via text; for email input clients will respond to a yes or no prompt in the email. A successful response will grant access to the client BeAvis Car Rental System account.

Unit 1 Test	Unit 2 Test
Feature: Client Account Login Security • Two-Factor Authentication Login Test 1: Login Inputs: Username = "Client01" Password = "0001" Output: Two-Factor Authentication Login Test 2: Login Inputs: Username = "Client01" Password = "2222" Output: "Incorrect Password/Username" Login Test 3: Login Inputs: Username = "Client02" Password = "0001" Output: "Incorrect Password/Username" Login Test 4: Login Inputs: Username = "Client02" Password = "2222" Output: "Does Not Exist" Login Test 5: Login Inputs: Username = " " Password = " " Output: "Does Not Exist"	**Feature:** Two-Factor Authentication • Call Factor • Text Factor • Email Factor Call 2FA Test 1: Call Factor Inputs: Digit Verification = "7" Call Output: Account Access granted Call 2FA Test 2: Call Factor Inputs: Digit Verification = "2" Call Factor Output: Account Access Denied Text 2FA Test 3: Text Factor Inputs: Text Verification = "2222" Text Factor Output: Account Access Granted Text 2FA Test 4: Text Factor Inputs: Text Verification = "1111" Text Factor Output: Account Access Denied Email 2FA Test 5: Email Factor Inputs: Email Verification = "yes" Email Factor Output: Account Access Granted Email 2FA Test 6: Email Factor Inputs: Email Verification = "no" Email Factor Output: Account Access Denied

Integration Tests:

For the integration verification tests, we use a partition of inputs to test relevant inputs and determine if the actual outputs correspond to our expected outputs. For the first integration test, we have a user enter in the criteria for a previous rental contract, and then when the contract has been successfully found, the user receives an email with the requested information. In our first test case for that scenario, the user enters the correct information and receives the emailed receipt; in the second test case the user enters invalid information and is not able to receive the receipt but instead gets a message indicating the receipt was not found and to try again. In our second integration test, we have an employee user schedule maintenance for a vehicle currently in the inventory. In test case 1 the user enters the information for a vehicle that is currently in inventory, and successfully schedules the maintenance; in the second test case the user schedules maintenance for a vehicle that is not currently in inventory and receives a message.

Integration 1 Test	Integration 2 Test

Feature: ClientHistory viewPreviousRentals(RentalContract)
 	
Test case 1:
Scenario: A client wants to receive the receipt from a previous rental agreement and enters the correct criteria Input:
- agreementID: 123ABC
- ClientID: BillyBob43
- vehicleID: 5DB89
- startDate: 10-31-2024
- endDate: 11-15-2024
 	
ClientHistory viewPreviousRentals(RentalContract): void
 	
RentalContract rc = new RentalContract; rc.agreementID = “123ABC”; rc.clientID = “BillyBob43”; rc.vehicleID = “5DB89”; rc.startDate = 10312023; rc.endDate = 11152023;
	Feature: Inventory scheduleMaintenance()
	
	Test case 1:
	Scenario: Employee schedules maintenance for a vehicle that is currently available in inventory.
	Input:
	- vehicleID: 2AV55
	- make: Ford
	- model: Explorer
	
	Inventory scheduleMaintenance(): void
	
	Vehicle v = new Vehicle;
	v.vehicleID = “2AV55”;
	v.make = “Ford”;
	v.model = “Explorer”;
	
	if (v.checkAvailability() == true){
	v.scheduleMaintenance(); print:
	v + “has been scheduled for maintenance”;
	} else {

if (viewPreviousRentals(RentalContract) == rc){ rc.sendReceipt();
print: rc + “receipt has been sent”;
} else {
print: “Could not locate previous rental agreement, please try again”;
}
Output:
- “123ABC BillyBob43 5DB89 10312023
11152023 receipt has been sent.”
Test result: Pass

Test case 2:
Scenario: A client wants to receive the receipt from a previous rental agreement and enters incorrect criteria Input:
- agreementID: 321CBA
- ClientID: Bob
- vehicleID: 5DB89
- startDate: 10-31-2024
- endDate: 11-15-2024

ClientHistory
viewPreviousRentals(RentalContract): void

RentalContract rc = new RentalContract; rc.agreementID = “123ABC”; rc.clientID = “BillyBob43”; rc.vehicleID = “5DB89”; rc.startDate = 10312023; rc.endDate = 11152023;

if (viewPreviousRentals(RentalContract) == rc){ rc.sendReceipt();
print: rc + “receipt has been sent”;
} else {
print: “Could not locate previous rental agreement, please try again”;
}
Output:
- “Could not locate previous rental agreement, please try again.”
Test result: Pass

print: v + “is currently in use and cannot be scheduled for maintenance”;
}
Output:
- checkAvailability() = true;
- “2AV55 Ford Explorer has been scheduled for maintenance.”
Test result: Pass

Test case 2:
Scenario: Employee schedules maintenance for a vehicle that is not currently available in the inventory.
Input: 3EF35, Dodge, Viper

Inventory scheduleMaintenance(): void

Vehicle v = new Vehicle;
v.vehicleID = “3EF35”;
v.make = “Dodge”;
v.model = “Viper”;

if (v.checkAvailability() == true){
v.scheduleMaintenance(); print:
v + “has been scheduled for maintenance”;
} else {
print: v + “is currently in use and cannot be scheduled for maintenance”;
}
Output:
- checkAvailability() = false;
- “3EF35 Dodge Viper is currently in use and cannot be scheduled for maintenance.”
Test result: Pass

System Tests:

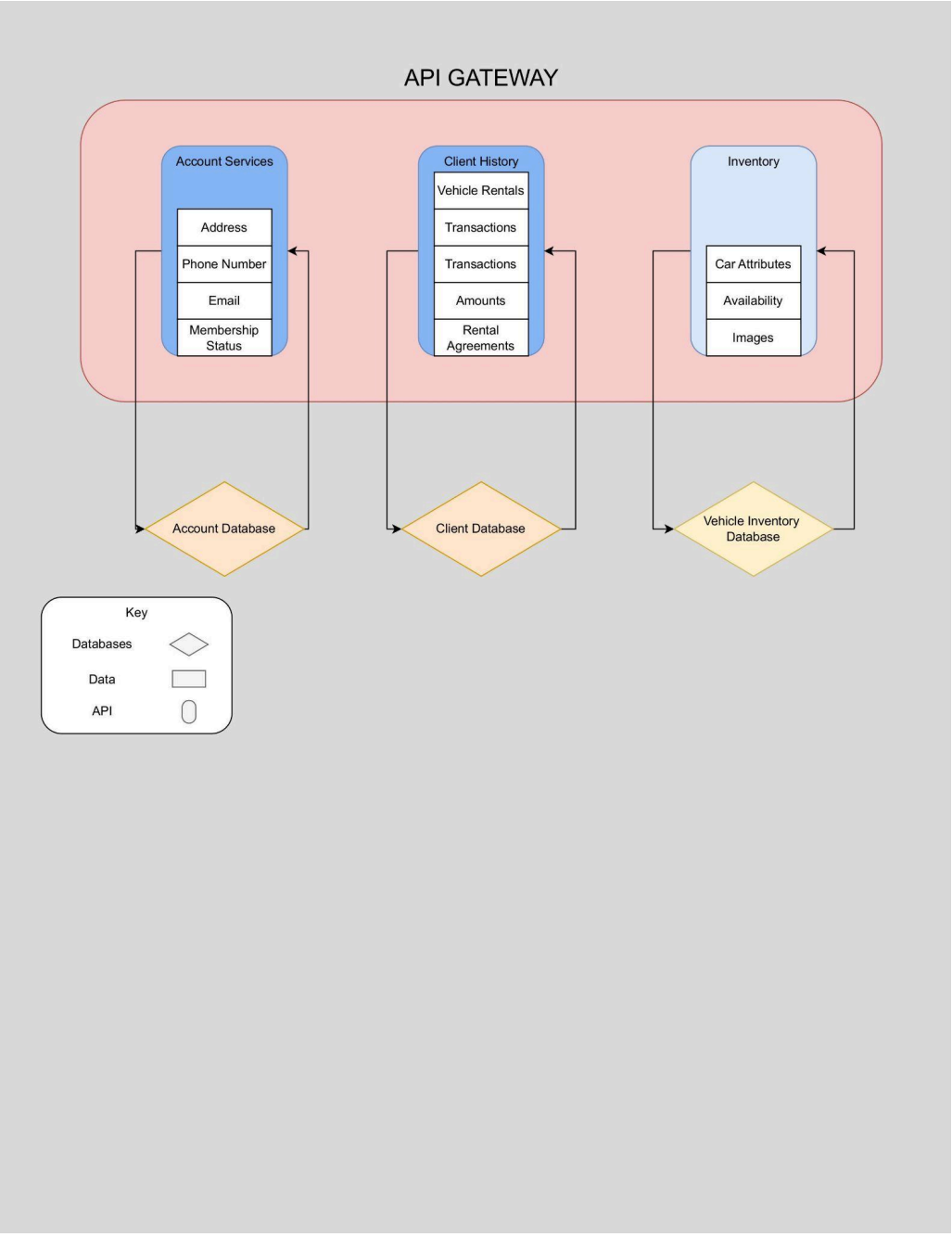
For this verification, we will use System tests to give a clear walkthrough of how the system will operate. We shall be running through two different system tests, in which we will go through various scenarios that interact with our Inventory of vehicles. Test one will feature searches for vehicles when one is in the system and another for one that is not currently in the system. When searching the inventory, if the vehicle is in the system, it should show up in the search results. If it's not in the system, then it will return nothing as the vehicle isn't found. This will demonstrate how the employees might interact with the system in opposing scenarios. The second system will be related to checking the availability of a vehicle. The user will have one scenario where the desired vehicle is available and one where it isn't. This will demonstrate the boolean technique used to determine if a vehicle is taken. The first scenario will return true as its available, and the second should return false as it is not available.

System 1 Test	System 2 Test
Feature: searchVehicle: Vehicle Search Test 1: The employee will use the car rental system to look for a vehicle that is in stock. They will search for the vehicle using the vehicle name. As the vehicle is in the system, it will show up in the search results. Search Test 2: The employee will be searching for a vehicle that is not in the system. The employee will enter the desired vehicle using the search. Due to the vehicle not being in the system, there will be nothing in the search results.	Feature: checkAvailibility(): bool Availability Test 1: The employee will be checking the availability of a desired vehicle. They will search for the desired vehicle in the system, and it will indicate that it has indeed not been taken with a true statement. Availability Test 2: The employee searches the system for information on whether or not a vehicle is available. The vehicle is being used already, so the check availability will return with a false message to indicate that it is not available to be used.

Data Management Strategy:

We chose 3 separate databases, an SQL database for account services and client history, and an NoSQL database for Vehicle Inventory. Splitting the data into 3 databases would help organize information based on specific services that are accessing it, which simplifies the overall maintenance. We chose SQL for both account services and client history because of SQL's ACID properties that ensure data consistency with logging transactions and accurate client information data. We chose NoSQL for vehicle inventory because of its practical browsing use and scalability. As the vehicle inventory must exceed the number of clients, NoSQL's ability to take on large amounts of data is crucial for both employee and client. NoSQL also handles dynamic schema and unstructured data making it ideal for managing numerous car attributes and vehicle images for clients to view. SQL for the vehicle inventory system may have issues for long-term inventory growth. As the inventory increases, SQL vertical scaling would lead to costly hardware upgrades. This makes it less efficient for managing an expanding inventory system. The logic for client history is divided into multiple fields such as vehicle rentals, transactions, amounts, and rental agreements to track rental history. Account Services is divided into information that corresponds with the client details, like address, phone number, email and current status of membership. Car inventory is split by car attributes, availability, and images.

Data-Management Diagram:



Trade Off Discussion:

We decided to implement 3 databases for our BeAvis car rental software system, two of which utilize SQL (client history and account services) and the other NoSQL (car inventory). This arrangement allows BeAvis to efficiently manage their inventory, client information, and account services in a secure yet scalable manner. There is an abundance of options available for a company of this magnitude to manage their day to day business needs, however, this combined implementation of SQL and NoSQL have marked advantages.

One option that could have been a bit more straightforward would have been to use a single database in which the car inventory, client information and account services were all located, saving costs (initially) and simplifying the overall architectural structure of the software system. While this may seem enticing, the trade off of ease of structural construction for potential security risks or long term overspending is to great to overlook: with one database for a company of this significance, the choice of NoSQL would clearly allow for scalability and increasing revenue, however, the vulnerabilities associated with a lack of encrypted data and insecure authentication create too much of a potential security risk for users personal information. Additionally, the lack of A.C.I.D. properties means monetary transactions are at risk for both the client and the company. Conversely, if BeAvis were to utilize a single database composed solely of SQL, the business would ultimately take a hit due to the costly maintenance and limited flexibility. Due to the complex nature of SQL, having a massive single database will require consistent maintenance by skilled and expensive administrators who ensure performance, which in turn leads to increased operational expenditures. Using both SQL and NoSQL spread across multiple databases is the clear choice for efficiency and cost effectiveness, allowing BeAvis to maximize revenue in a safe and practical manner.

Another setup that could potentially serve BeAvis would be to use two databases: one NoSQL database for the car inventory and one SQL database combining the client information and account services. Once again, this allows for a simplified structural arrangement, but alas with a tradeoff as well. Combining the client information with the account services leads to an unnecessary amount of information being stored in a single database; when dealing with large datasets, SQL has been known to experience a noticeable decrease in query performance, which will be directly experienced by clients interacting with the software system. Employee work productivity will also be affected due to delays in system processing. While it is possible to vertically scale the SQL database server by adding in additional supportive hardware to assist with query speed, the cost of installing and updating the system to accommodate such traffic would significantly increase company overhead. Splitting up the SQL databases will ensure reliability, efficiency and overall user experience for clients and BeAvis employees.