Software Design Specification 2.0
BeAvis Car Rental System

Connor Pham, Devin Widmer

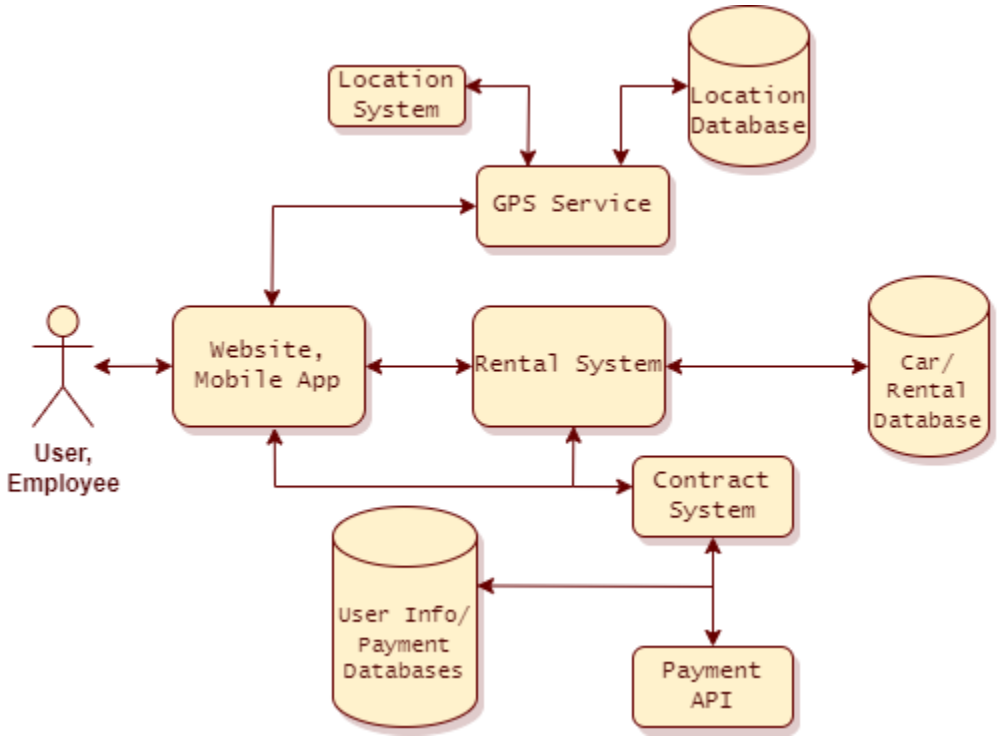
April 12, 2024

System Description:

The car rental system will enable users to create accounts, verify their identity, link payment methods, electronically sign rental agreements, and manage their rentals through an Android/iPhone mobile app as well as a dedicated website. Additionally, it will feature an administrative interface for employees and administrators to manage customer accounts, monitor car statuses, and review contracts. This document aims to provide a comprehensive specification for the development of a car rental system designed to digitize the traditional pen-and-paper process. Accessible via a mobile application and a website, the system offers an efficient, user-friendly, and secure way to rent cars. It is intended for customers seeking rental services, employees who manage rental operations, and administrators in charge of overseeing secure information.

Software Architecture Overview:

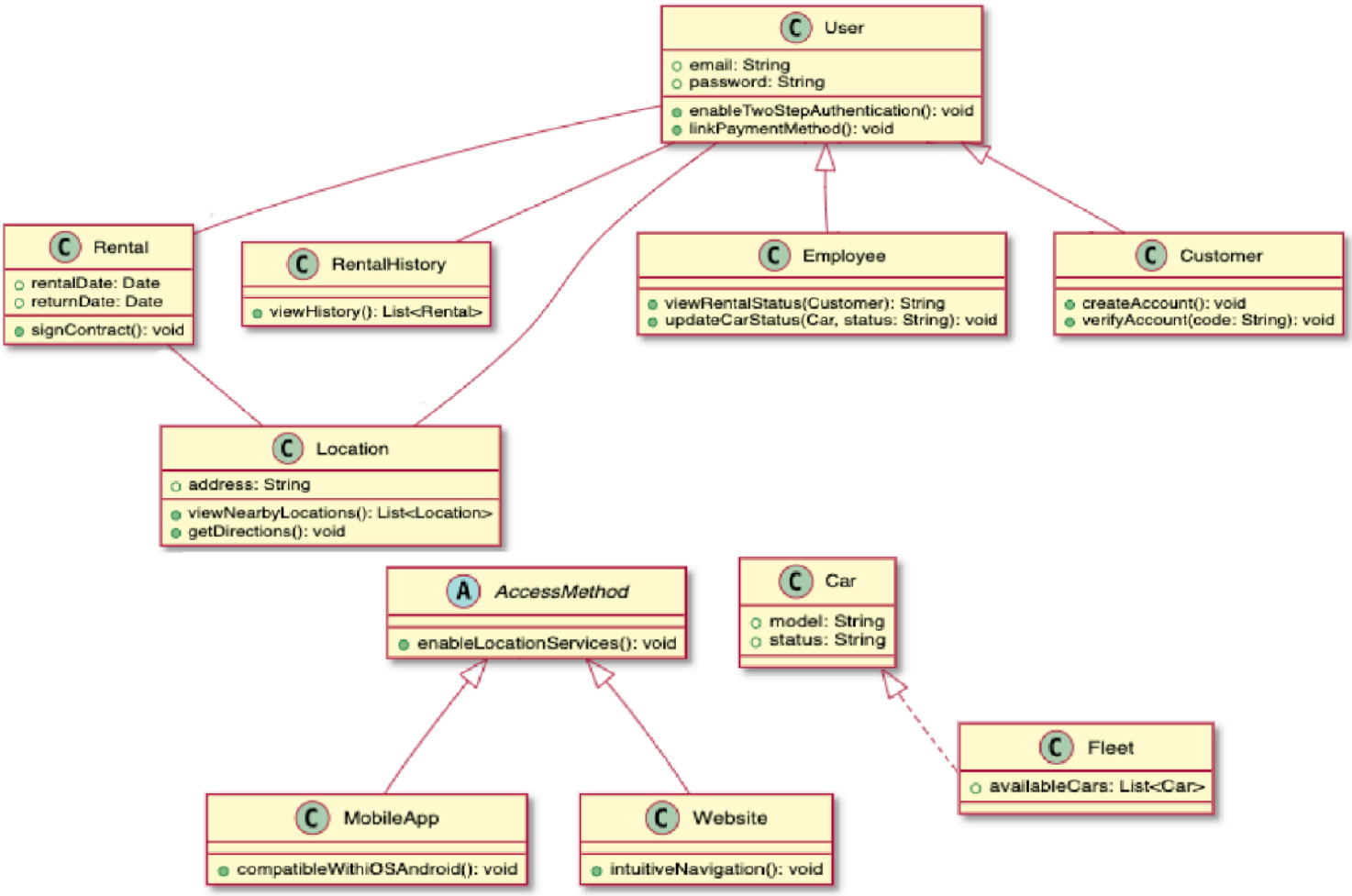
Software Architecture Diagram:



Software Architecture Description:

The software architecture diagram depicts the overall architecture of the BeAvis Car Rental System. The components, starting with the User and Employee, are the system's users. They interact with the system through a website or mobile app, which serves as the primary interface for the car rental services. The Website and Mobile App component acts as the way for users to access the rental system. It provides functionality for users to browse available cars, make car rentals, payments, sign contracts, and manage their accounts by connecting to the rental system. The central rental system component manages the core functionality of the car rental service. It processes rental requests, retrieves information from the databases, and coordinates and connects with other APIs and systems like the location and contract systems. It is connected to the Car/Rental Database, which stores information about the available cars and their rental status. The GPS service is responsible for managing geographical information, possibly used to track the cars' locations or to assist users in finding rental locations. It is connected to the location database which stores location-related data and the location system which provides tracking tools and information. The Car/Rental database contains all data related to the cars available for rent, including their models, availability, status, or rental history. The contract system manages rental agreements between the service and the users. It records terms, conditions, and duration of car rentals. This system interacts with the user info and payment databases which store personal and payment information of users, ensuring that the service has the necessary data to process transactions and maintain records for administrators, users, or employees to access. It is also connected to the payment API, that handles payment transactions. This would be used to process user payments securely when renting a car.

UML Diagram:



UML Class Description:

In this UML diagram the 'User' class holds most of the functionality. In the 'User' class there are 2 string variables known as 'email' and 'password.' It has two methods to enable two-step authentication and links payment method, both of these are of void return type that take in no parameters. The 'Employee' class inherits from 'User' and the operations include 'viewRentalStatus(Customer)' which returns a string and requires a parameter 'Customer'. 'UpdateCarStatus(Car, status: String): void' is a method that updates the current status of the car while taking in two strings as parameters. Similarly, the 'Customer' class inherits from the 'User' class and has two operations one to 'createAccount():' which returns void requiring no

parameters and another to ‘verifyAccount(code: String): void.’ This takes in a string as input and does not return anything to the user. The input is a code whether it is a full string or a string concatenated with numbers to compose its ID.

Moreover, in the ‘Rental’ class, it has two attributes: ‘rentalData: Date’ and ‘returnDate: Date.’ These two variables are of type Date. They do as their name describes, one tells one the rental begins and when the rental is over. The only operation it does is ‘signContract(): void,’ which only updates the software system and does not return anything. In the ‘RentalHistory’ class, the operation ‘viewHistory(): List<Rental>’ looks up if the user has previously rented any vehicles. It returns a list type whether it is a linked list, arraylist, or other. The ‘Location’ class has a string called address which stores the address of the car rental place and it has two methods ‘viewNearbyLocations(): List<Location>’ and ‘getDirections():’. Where one returns a list of ‘Location’ type wherever there may be rentals nearby, both require no parameters.

Furthermore, in the Car class some of the attributes include two String variables called ‘model’ and ‘status’ where one tells what the model of the car is and the other concerns the car’s availability, carfax reports, etc. While the ‘Fleet’ class has a list variable that contains a list of available cars within the database of the car rental establishment and can be rented by the customer.

Throughout this entire UML there is an interface called ‘AccessMethod’ which has the operation ‘enableLocationServices(): void’ that allows the user to have access to location services. This only applies if this is implemented. Likewise, the ‘MobileApp’ implements the previous Interface which has the operation ‘compatibleWithIOSAndroid(): void’ that allows for compatibility with iOS and Android devices with the car rental’s system. Lastly, the ‘Website’ class is implemented from the ‘AccessMethod’ Interface and it has a method that allows for users to surf the website, it is called ‘intuitiveNavigation(): void’ and it serves as a more general function.

Development Plan and Timeline:

2 Weeks: Implementation Planning

Begin formulating a plan for development working with the team to identify requirements, risk assessments, and resources provided. Follow software architecture and UML planning.
(Connor & Devin)

1.5 Months: Web and Mobile Application Development

Begin development of UI design, and application development to satisfy all requirements of overall application structure. Develop according to the software architecture plan. Develop based off of specifications in software and hardware requirements.
(Devin)

2 Months: Development, Database, API

Begin development of Car/Rental and User Info/Payment databases and develop system components and interfaces. Build based off of software and hardware requirements and resources given to create required software systems.
(Connor)

1.5 Months: System component implementation and testing

Integrate developed components and interfaces into application. Include concurrent testing and test cases to identify risks, failures, and to streamline implementations.
(Connor & Devin)

1 Month: Overview and Deploy

Team overviews product after identifying issues, risks, and failures to start deployment of the overall system.
(Connor & Devin)

After release Indefinitely: Maintenance and assistance

Perform needed maintenance to fix errors, failures, risks, issues, or new implementations required by the customer. Providing ongoing support and system updates.
(Connor & Devin)

Verification Test Plan:

This verification test plan for our software system includes targeting different functionalities of our system by carefully devising two test sets with methodical intention. Our test sets will cover the 3 granularities of unit, integration, and system testing.

Test Set 1:

Unit:

verifyAccount(): void

```
private String verificationCode = "123456";
public void verifyAccount(String code)
   if(this.verificationCode.equals(code))
      “pass”
    else
     “fail”
```

Test Case: Verify Account with Correct Code
Input: Verification code "123456" provided by the system to the user.
Expected Output: Verification process should pass, indicating the verification code is correct.
Verify: Check if the output of verifyAccount() method is "pass" when the correct code is provided.

Integration:

```
public void testCustomerVerificationAndRentalContractSigning()
   String correctCode = "123456”
   customer.verifyAccount(correctCode);
   if (customer.isVerified())
      rental.signContract();
   If customer.isVerified() && rental.isContractSigned() == true
      “Pass”
    Else
        “Fail”
```

Test Case: Customer Verification and Rental Contract Signing
Input: Correct verification code "123456" provided by the customer.
Expected Output: The customer's account is verified, and the rental contract is signed, indicated by the customer being marked as verified and the rental contract's status being marked as signed.
Verify: Assert that customer.isVerified() returns true and rental.isContractSigned() returns true, indicating success.

System:

The client will arrive on site of the rental place and choose a vehicle where the system will ask for their account information where they will either create a new account or verify and use a previously existing account, or they could browse online. Then, they will go through the selection process where they will see details for the vehicle they like and will then proceed to rental duration, information, and all the paperwork that entails such a transaction either via mobile app or website. After selecting the service and vehicle that is desired the next prompt for the user is payment methods where he or she will then be able to pay with their previous information on their account or new. After all is said and done they will receive both confirmation and a full-length receipt detailing their transaction through to their email.

Test Set 2:

Unit:

```
EmployeeUpdateCarStatus(): void
private Car car = “Scat Pack”;
private String status = “rented”;
Employee.updateCarStatus(Car car, String status);
if (car.getStatus().equals(“rented”))
    “Pass”
else
    “Fail”
```

Test Case: Update the status of a car to "rented"
Input: Car object with initial status "available", String "available"
Expected Output: Car's status is updated to "rented"
Verify: Assert that car.getStatus().equals("rented")

Integration:

```
EmployeeUpdatesCarStatusAndFleetUpdates()
 private Fleet fleet = new Fleet();
 fleet.avaialableCars.add("Scat Pack");
 fleet.availableCars.add("Car2");
 fleet.availableCars.add("Car3");
 private Car car = "Scat Pack";
 private String status = "rented";
 Employee.updateCarStatus(Car car, String status);
if ("Scat Pack" is NOT in fleet.availableCars())
   "Pass"
else
    "Fail"
```

Test Case: Update the status of a car to "rented"

Input: Car object with initial status "available", String "available"

Expected Output: Car's status is updated to "rented" and car is no longer in availableCars in the fleet

Verify: Assert that the "Scat Pack" Car is not in the Fleet's availableCars

System:

A customer goes on the system app or website and selects a vehicle that is listed as "Available". The customer proceeds through the process of renting the vehicle, which includes providing their account information (either by creating a new account or using an existing one), selecting the vehicle and rental duration, choosing a payment method, and finalizing the transaction, and signing the rental agreement contract. Upon successful completion of this process, an employee updates the vehicle's status in the system from "Available" to "Rented" to reflect the new rental agreement. The system then processes this update and ensures that the vehicle is no longer listed as available for other customers and it is updated in the system that it is no longer available in the fleet's available cars list.

Test Set 1 Explanation:

Within Test Set 1, the unit test presented is the function to verify if the client has a preexisting account. To do so, the system sends a code to the email given by the user and if there is an existing account under that email then the code they receive will be valid. Then that verification code is input to see if it is valid or not and the if condition dictates if it passes or not. The scope of this test is mostly singular.

For the integration testing in the first test set, we are analyzing how the customer and the rental class interact with each other, or more broadly how the customer goes through the rental process to complete their transaction. To do so the client is verified similar to the previous test and they complete the paperwork to finalize the process. If both conditions are met on the customer side and the system's side where everything is a good ot go then it passes otherwise this test fails. The scope of this test involves two classes or two parties in a real life situation. The fail coverage would entail if one or both conditions fail, leads to a failed transaction and does not go through.

The system test of this test set is mostly self explanatory and covers nearly a global scope of the car rental system along with user interaction. Any fault at any point would lead to an overall failure to complete the objective. So fail coverage is global but not a pinpointed aspect.

Test Set 2 Explanation:

Test Set 2 focuses on the functionality and integration within the vehicle management aspect of the car rental system, particularly around updating the status of vehicles as they are rented out to customers.

In the unit test scenario, it is testing the ability of an employee to update the status of a car within the system. This is a critical function as it directly impacts the availability of cars for rental, affecting both customer experience and inventory management. The test verifies that when an employee updates a car's status to "rented," the system accurately reflects this change. This test is narrowly focused on the method Employee.updateCarStatus(), ensuring that it functions correctly in isolation. The pass/fail condition is: if the car's status is successfully updated to "rented," the test passes. Otherwise, it fails. This test's scope is limited to the interaction between an employee and a single car object, making it a unit test.

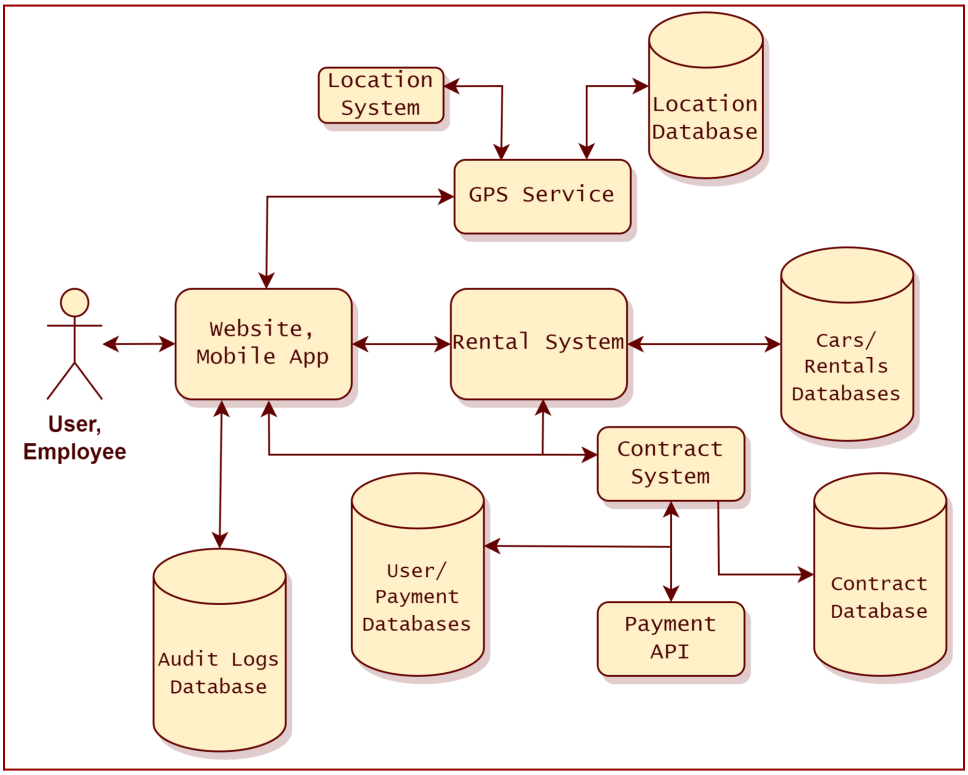
The integration test, EmployeeUpdatesCarStatusAndFleetUpdates, examines how changes to a car's status affect the fleet management. When a car's status is updated to "rented," it should not only change the car's status but also remove it from the list of available cars in the

fleet. This test simulates the process of updating a car's status and then checking the fleet's available cars to ensure the update was properly integrated across the system. The expected outcome is that once a car is rented and its status updated, it no longer appears as available in the fleet, signifying a successful integration between the Employee and Fleet classes. This test bridges multiple components of the system such as individual cars and the fleet management system which makes it an integration test.

The system-level test outlines a scenario where a customer rents a vehicle through the app or website, which triggers a series of actions culminating in an employee updating the car's status to "rented." This reflects the end-to-end process from customer interaction through internal system updates to ensure the vehicle is no longer listed as available. Any failure in this chain from account verification, vehicle selection, payment processing, to the final status update could prevent the transaction from completing successfully. The system test's scope is the broadest, encompassing customer interactions, back-end processes, and the synchronization of data across the platform to achieve the desired outcome which is a vehicle successfully rented and correctly reflected in the system's inventory.

Software Architecture Overview 2.0:

Updated Software Architecture Diagram:



Updated Software Architecture Description:

The software architecture diagram depicts the overall architecture of the BeAvis Car Rental System. The components, starting with the User and Employee, are the system's users. They interact with the system through a website or mobile app, which serves as the primary interface for the car rental services. The Website and Mobile App component acts as the way for users to access the rental system. It provides functionality for users to browse available cars, make car rentals, payments, sign contracts, and manage their accounts by connecting to the rental system. The central rental system component manages the core functionality of the car rental service. It processes rental requests, retrieves information from the databases, and coordinates and connects with other APIs and systems like the location and contract systems. It is connected to the Cars/Rentals Database, which stores information about the available cars and their rental status. The GPS service is responsible for managing geographical information, used to track the cars' locations or to assist users in finding rental locations. It is connected to the Location database which stores location-related data and the location system which provides tracking tools and information. The Cars/Rentals database contains all data related to the cars available for rent, including their models, availability, and status. The contract system manages rental agreements between the service and the users. It records IDs, terms, conditions, and duration of car rentals. This system interacts with the user info and Payment databases which stores transaction and payment information of users, ensuring that the service has the necessary data to process transactions and maintain records for administrators, users, or employees to access. It is also connected to the payment API, that handles payment transactions. This would be used to process user payments securely when renting a car. The user database contains user information, employee information, passwords, and information. The Contract database is used to store information about contracts information, length, ID, cost, and signature. The Audit Log database is for the tracking and security of the system. This database is made to capture interactions

within the system, including user activities through the website or mobile app and administrative actions. It records a wide array of actions such as making reservations, processing payments, signing contracts, and managing accounts, alongside the administrative tasks performed by employees. Each entry in the AuditLogs database is identified by a LogID and includes the UserID of the individual who performed the action, the ActionType to describe the action, detailed descriptions of the action taken, and a timestamp marking the exact moment of the action.

Data Management Strategy:

Users(S)

UserID	Username	UserRole	Password
1	Jenny	Customer	YXT5qsQgqvsw
2	James	Customer	sznnTSDwBn7Z
3	JoeM	Employee	LeH5JuAghD2q
4	Jane123	Employee	rn4Q5puhYrmr

Rentals (R)

Rental ID	User ID	Car ID	Make	Model	Price	Color	Status
5500	1	110	Mercedes	C63 AMG	600	White	Available
5501	2	111	BMW	M4 CS	700	Blue	Unavailable
5502	3	112	Volkswagen	Jetta	200	Black	Unavailable
5503	4	113	Honda	SI Civic	100	Gray	Available

Rental Contracts (RC)

Contract ID	Rental ID	Start Date	End Date	Payment Date	Amount	eSignature
4100	5501	2024-01-20	2024-02-20	2024-01-20	300.32	John Doe
4101	5502	2024-02-15	2024-02-18	2024-02-15	421.12	Jane Doe
4102	5503	2024-03-01	2024-03-05	2024-03-01	150.43	Jenny Doe
4103	5504	2024-03-15	2024-04-01	2024-03-15	170	James Doe

AuditLogs(A)

LogID	UserID	ActionType	Details	TimeStamp
1001	1000	Login	User Logged In	2024-04-01 08:00:00
1002	999	Modify Status	Admin Modified Car Status	2024-04-02 10:00:00
1003	998	Rent	Car Rental started	2024-04-01 09:00:00
1004	1000	Void	Contract Redacted	2024-04-03 07:30:00

Location (L)

CarID	Current Location	Nearest Car Rental
100	Point Loma	SDSU
101	Chula Vista	SDSU
102	San Ysidro	SDSU
103	La Jolla	SDSU

Payment (P)

Payment ID	Rental ID	UserID	Amount	Payment Method	Date & Time	Status	Receipt
1	5501	9	400.20	Debit Card	2024-04-01 08:00:00	Processed	Email
2	5502	8	140.70	Apple Pay	2024-04-10 03:00:00	Failed	Text
3	5503	7	700.30	Credit Card	2024-04-01 09:00:00	Processed	Email
4	5504	6	323.00	Credit Card	2024-04-03 07:30:00	Processed	No receipt

Description:

The BeAvis Car Rental System employs a comprehensive database schema to manage its operations effectively ,comprising key tables that resemble data management. The Users Table holds user information such as ‘UserID’, ‘Username’, ‘UserRole’, and ‘Password’, detailing both customer and employee data. The Rentals Table catalogs car rentals, with fields like ‘Rental ID’, ‘User ID’,’ Car ID’, ‘Make’, ‘Model’, ‘Price’, ‘Color’, and ‘Status’, tracking each vehicle's details and availability. The Rental Contracts Table stores contract-related data, including ‘Contract ID’, ‘Rental ID’, ‘Start Date’, ‘End Date’, ‘Payment Date’, ‘Amount’, and ‘eSignature’, essential for managing rental agreements. The AuditLogs Table logs significant actions within the system, containing ‘LogID’, ‘UserID’, ‘ActionType’, ‘Details’, and ‘TimeStamp’ to record activities such as logins and modifications to car statuses. The Location Table manages geographic data of vehicles, with ‘CarID’, ‘Current Location’, and ‘Nearest Car Rental’, indicating where cars are located and their proximity to rental facilities. Finally, the Payment Table documents payment transactions, including ‘Payment ID’, ‘Rental ID’, ‘UserID’, ‘Amount’, ‘Payment Method’, ‘Date & Time’, ‘Status’, and ‘Receipt’, providing comprehensive tracking of financial transactions. This detailed and structured data management framework supports the car rental system's functionality for both user interactions and administrative operations, ensuring accuracy and efficiency.

Trade Off Discussion:

As our software system has a lot of sensitive information, it is important that it is stored securely in a fashion that protects the confidentiality of the clients and the car rental establishment. After looking at both how the queries were handled along with the visual representation of NoSQL and SQL databases, we came to the conclusion that SQL databases are far more easy to work with and provide secure storage. In SQL, it is a simple and straightforward table representation of the database and its attributes as opposed to NoSQL which can vary from cryptic key-value, graphs, document trees, and column family grids, while our SQL database is a simple relational table. With this representation there is structured data with clear relationships and it follows the ACID compliance for atomicity, consistency, isolation, and durability.

We chose to do 6 essential databases because these accomplish the original outline of the software system and our implemented improvements, while hitting the major benchmarks for necessity and security with detail. All of the data in these databases are split up to be sound and logical when they are relationally mapped to one another and provide a basis of overall completion and modularity to accomplish the primary objectives of the software system.

The alternative to organization of data in our database could be singular databases that were distinct from each other. For example the Rental Contract and Payment database has overlapping attributes. We could have created unique tables, but with the way we have done it it is far more secure but does require more sophistication to be in sync if we were to implement it on a technical level. Furthermore, if we were to use NoSQL the tradeoff would offer more scalability and flexibility at the expense of reducing efficiency of the SQL transactions. With our SQL databases, we have a lesser chance of bottlenecks since we have multiple databases. Our technology choice is the most optimal for us at the current iteration and requirements of the software system. If there are foreseeable changes or discrepancies in what the objectives are, then revisiting what technologies should be used can be up for discussion.