BeAvis Rental Car System

Software Design Specification by Alyssa Rivera, Sukruti Mallesh, and James Duong

October 12, 2023

1 System Overview

The BeAvis car rental company currently utilizes a traditional pen-and-paper rental process. The company would like to transition to a digital process via the BeAvis car rental system. The primary goal of this system is to provide an end-to-end digital experience for the car renting process and the management of the rental business. The updated system will utilize technology and cloud infrastructure in order to build a robust, secure, and extensible platform to digitize and streamline the car rental process for both customers and employees.

The system will provide customers with four primary functions:

I. Search for available rental cars
II. Make reservations
III. Manage their accounts
IV. Make payments online via the mobile app or the website

For employees, the system will provide a portal that will allow them to:

I. Manage inventory
II. Track vehicle status
III. Assign vehicles to reservations
IV. Process returns
V. Run rental reports

The system will integrate with various backend services for user authentication, payment processing, data analytics, caching, messaging, and search.

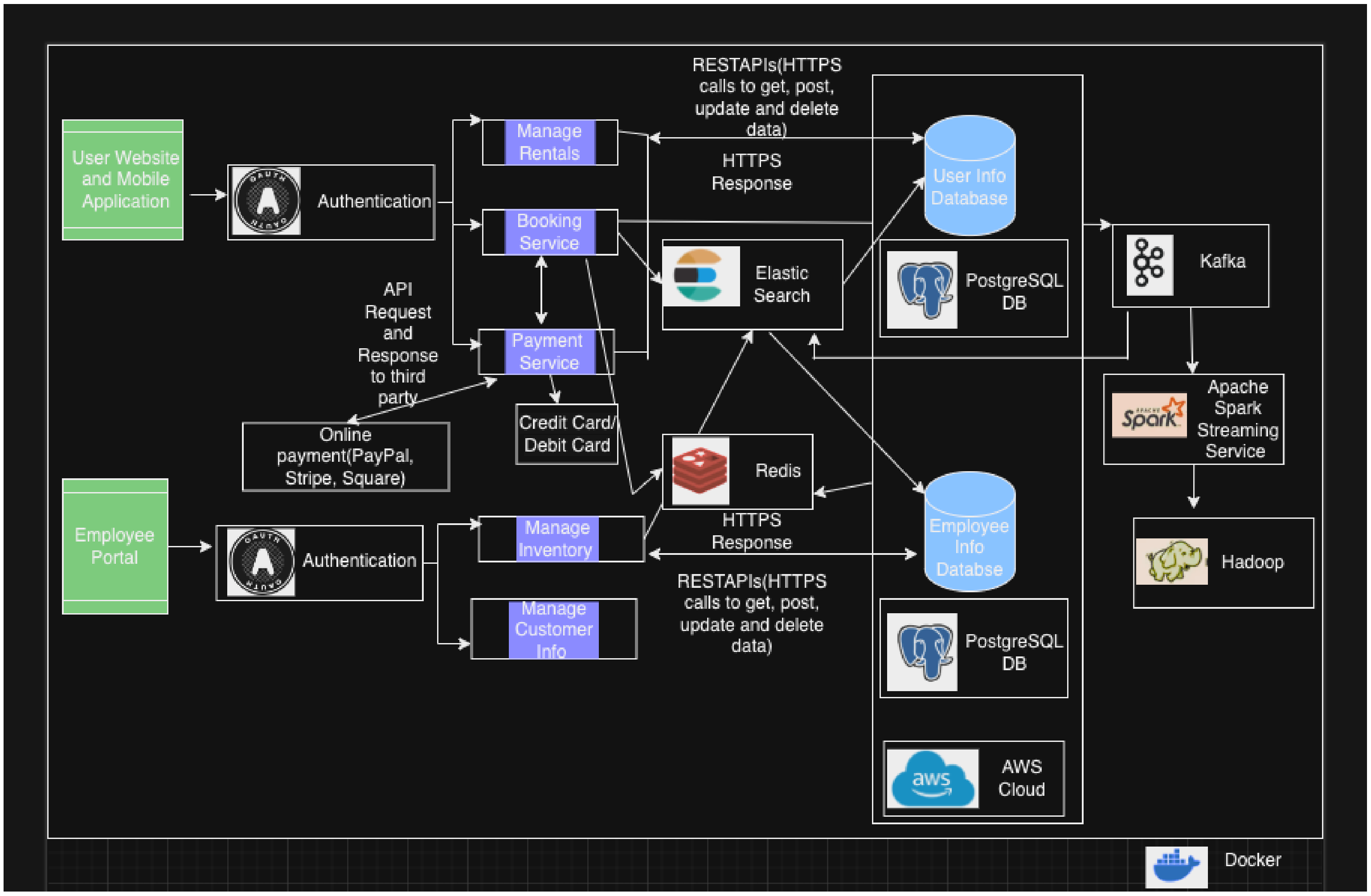
The BeAvis rental system will contain the following key features:

- Website and mobile apps for customer booking and account management
- Search and booking of rental cars across multiple locations
- Payment processing integrations
- Customer account and profile management
- Internal employee portal for managing rentals and inventory
- Integration with authentication, payment, analytics, caching, and search services
- Reporting capabilities for rental and customer data analysis
- Automated scheduling and assignment of rental cars
- Streamlined rental return processing
- Scalable architecture to handle customer traffic and bookings

Table of Contents

2 Software Architecture

2.1 Software Architectural Diagram



The main components of the system are:

- User Website and Mobile Application: This is the frontend that customers would use to view rental cars, make bookings, manage their account, etc. It would be a responsive website as well as iOS and Android apps.
- User Info Database: Stores information about users like name, contact details, login credentials, etc. This is a PostgreSQL database.
- Employee Info Database: Stores information about employees like name, roles, contact details, etc. This is also a PostgreSQL database.

- Employee Portal: An internal web portal used by employees to manage rentals, inventory, customer information, etc.
- Authentication Service: Handles user authentication and authorization. Uses Redis for caching user sessions.
- Payment Service: Handles payments from users. Integrates with payment gateways like PayPal, Stripe, Square, etc., and also has a credit/debit card option.
- Booking Service: Handles rental bookings and scheduling. Uses Kafka for streaming booking data.
- Manage Rentals: Business logic to handle rental workflow like checking in/out cars, assigning cars to bookings, calculating rental costs, etc.
- Manage Inventory: Business logic to track cars available for renting, occupied cars, cars due for maintenance, etc. Uses Elasticsearch for searching/indexing inventory.
- Manage Customer Info: Business logic to handle customer accounts, loyalty programs, promotions, etc.
- Hadoop Cluster: Used for storing and analyzing large amounts of rental and customer data to generate business reports.

The different services communicate via REST APIs. The frontend and backend are decoupled. The backend services can scale independently. AWS cloud is used for hosting the infrastructure. Docker containers are used for easy deployment and scaling of services. Key databases like PostgreSQL, Elasticsearch, and Redis provide persistence, search, and caching capabilities. The overall architecture allows BeAvis to handle user traffic, bookings, and billing in a robust and scalable manner.

The key components in the architecture diagram interact with each other via the various connectors:

- Users interact with the system via the Website and Mobile Apps. These frontend components connect to the backend services via REST APIs.
- The REST APIs connect to the Authentication Service to verify user identities and authorize access. The Authentication Service uses Redis to cache user sessions. It writes session data to Redis and reads it when needed for request authentication.

- For rental bookings, the frontend connects to the Booking Service API. The Booking Service uses Kafka to stream booking data to other services.

- To check inventory availability for a booking, the Booking Service interacts with the Manage Inventory service using REST APIs.

- For making payments, the frontend connects to the Payment Service API. The Payment Service interacts with external API gateways for online payment like PayPal. For credit/debit card payments, the Payment Service integrates directly with payment processors like Stripe to charge the cards.

- To record rental transactions, the Booking Service sends data to the Manage Rentals service using REST APIs. Manage Rentals records transactions in the PostgreSQL database.

- For customer profiles and loyalty programs, the frontend uses the Manage Customer Info service APIs. Customer data is stored in PostgreSQL.

- Manage Inventory uses Elasticsearch to index and search for available inventory for rentals.

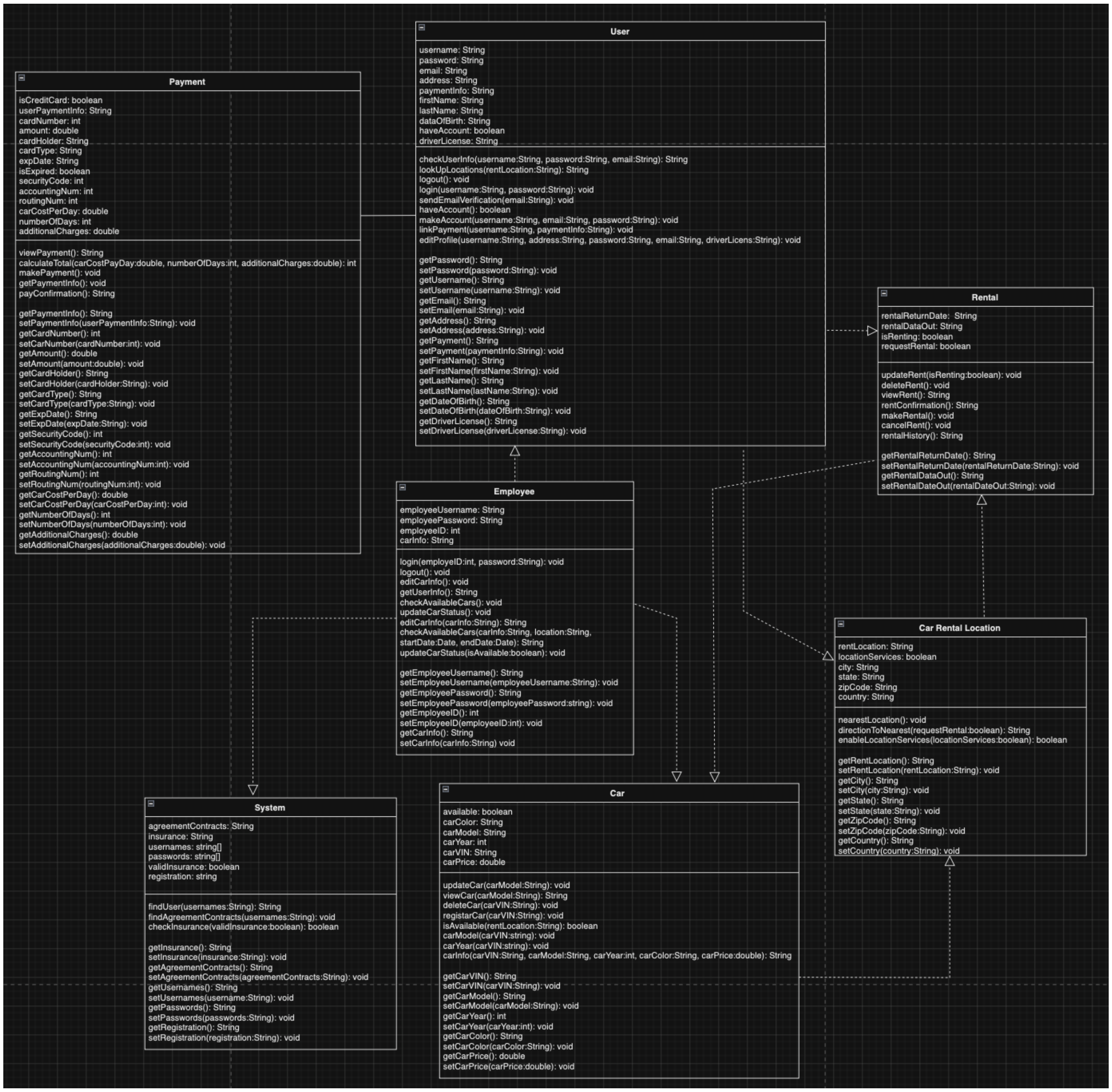
- Hadoop cluster runs analytics on historical data from PostgreSQL, Elasticsearch etc. to generate reports and insights.

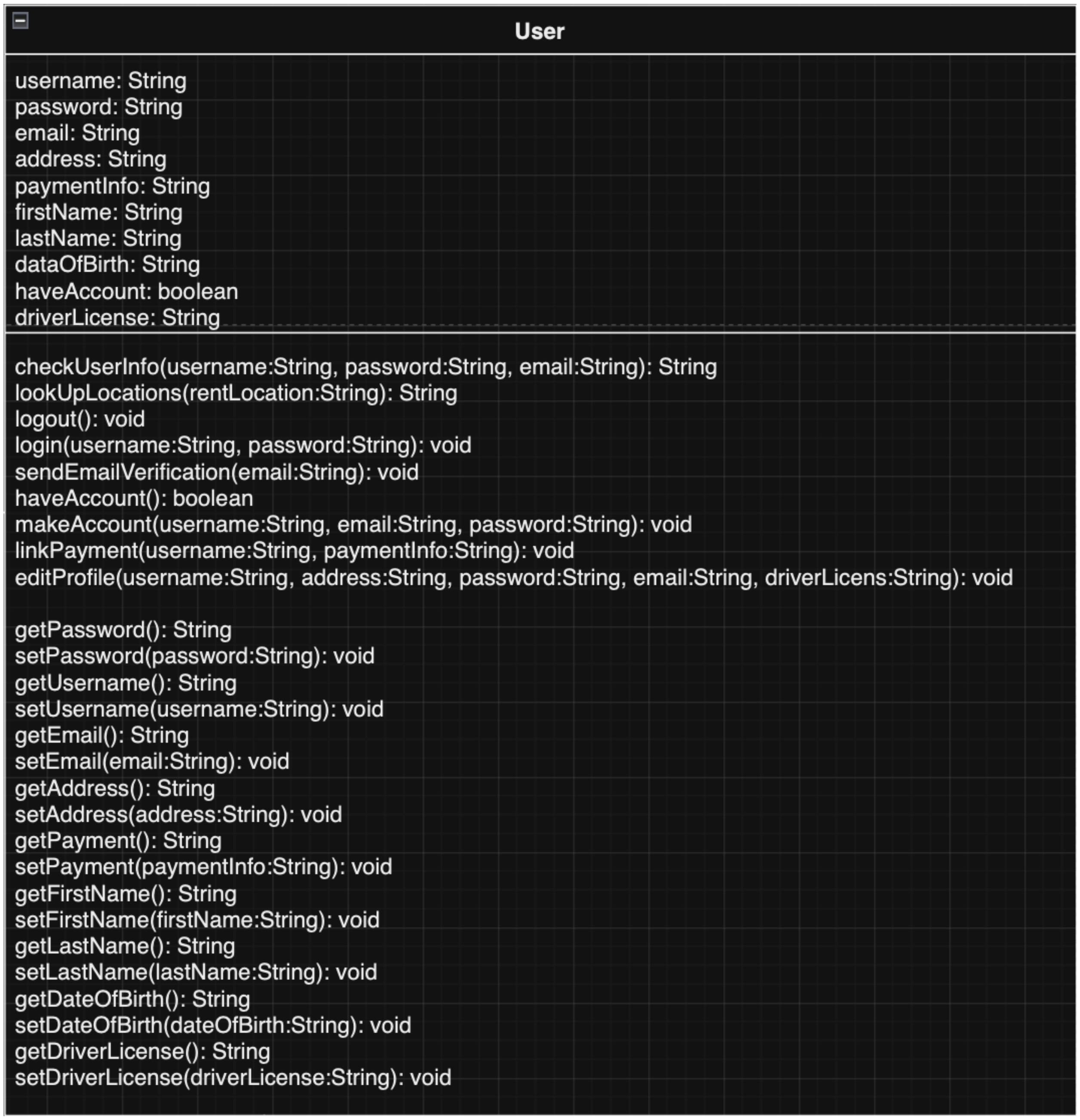
- The Employee Portal uses REST APIs to connect to services like Manage Bookings, Manage Rentals, Manage Inventory etc. to support employee workflows.

- All services use Docker containers for deployment. The containers are hosted on AWS cloud infrastructure.

In summary, the components interact via REST APIs, a Kafka message bus, database connections, and calls to external payment gateways in a decoupled way to facilitate the rental system functions.

2.2 UML Class Diagram


Payment
isCreditCard: boolean
userPaymentInfo: String
cardNumber: int
amount: double
cardHolder: String
cardType: String
expDate: String
isExpired: boolean
securityCode: int
accountingNum: int
routingNum: int
carCostPerDay: double
numberOfDays: int
additionalCharges: double
viewPayment(): String
calculateTotal(carCostPayDay:double, numberOfDays:int, additionalCharges:double): int
makePayment(): void
getPaymentInfo(): void
payConfirmation(): String
getPaymentInfo(): String
setPaymentInfo(userPaymentInfo:String): void
getCardNumber(): int
setCarNumber(cardNumber:int): void
getAmount(): double
setAmount(amount:double): void
getCardHolder(): String
setCardHolder(cardHolder:String): void
getCardType(): String
setCardType(cardType:String): void
getExpDate(): String
setExpDate(expDate:String): void
getSecurityCode(): int
setSecurityCode(securityCode:int): void
getAccountingNum(): int
setAccountingNum(accountingNum:int): void
getRoutingNum(): int
setRoutingNum(routingNum:int): void
getCarCostPerDay(): double
setCarCostPerDay(carCostPerDay:int): void
getNumberOfDays(): int
setNumberOfDays(numberOfDays:int): void
getAdditionalCharges(): double
setAdditionalCharges(additionalCharges:double): void
User
username: String
password: String
email: String
address: String
paymentInfo: String
firstName: String
lastName: String
dataOfBirth: String
haveAccount: boolean
driverLicense: String
checkUserInfo(username:String, password:String, email:String): String
lookUpLocations(rentLocation:String): String
logout(): void
login(username:String, password:String): void
sendEmailVerification(email:String): void
haveAccount(): boolean
makeAccount(username:String, email:String, password:String): void
linkPayment(username:String, paymentInfo:String): void
editProfile(username:String, address:String, password:String, email:String, driverLicens:String): void
getPassword(): String
setPassword(password:String): void
getUsername(): String
setUsername(username:String): void
getEmail(): String
setEmail(email:String): void
getAddress(): String
setAddress(address:String): void
getPayment(): String
setPayment(paymentInfo:String): void
getFirstName(): String
setFirstName(firstName:String): void
getLastName(): String
setLastName(lastName:String): void
getDateOfBirth(): String
setDateOfBirth(dateOfBirth:String): void
getDriverLicense(): String
setDriverLicense(driverLicense:String): void
Rental
rentalReturnDate: String
rentalDataOut: String
isRenting: boolean
requestRental: boolean
updateRent(isRenting:boolean): void
deleteRent(): void
viewRent(): String
rentConfirmation(): String
makeRental(): void
cancelRent(): void
rentalHistory(): String
getRentalReturnDate(): String
setRentalReturnDate(rentalReturnDate:String): void
getRentalDataOut(): String
setRentalDateOut(rentalDateOut:String): void
Employee
employeeUsername: String
employeePassword: String
employeeID: int
carInfo: String
login(employeID:int, password:String): void
logout(): void
editCarInfo(): void
getUserInfo(): String
checkAvailableCars(): void
updateCarStatus(): void
editCarInfo(carInfo:String): String
checkAvailableCars(carInfo:String, location:String, startDate:Date, endDate:Date): String
updateCarStatus(isAvailable:boolean): void
getEmployeeUsername(): String
setEmployeeUsername(employeeUsername:String): void
getEmployeePassword(): String
setEmployeePassword(employeePassword:string): void
getEmployeeID(): int
setEmployeeID(employeeID:int): void
getCarInfo(): String
setCarInfo(carInfo:String) void
Car Rental Location
rentLocation: String
locationServices: boolean
city: String
state: String
zipCode: String
country: String
nearestLocation(): void
directionToNearest(requestRental:boolean): String
enableLocationServices(locationServices:boolean): boolean
getRentLocation(): String
setRentLocation(rentLocation:String): void
getCity(): String
setCity(city:String): void
getState(): String
setState(state:String): void
getZipCode(): String
setZipCode(zipCode:String): void
getCountry(): String
setCountry(country:String): void
System
agreementContracts: String
insurance: String
usernames: string[]
passwords: string[]
validInsurance: boolean
registration: string
findUser(usernames:String): String
findAgreementContracts(usernames:String): void
checkInsurance(validInsurance:boolean): boolean
getInsurance(): String
setInsurance(insurance:String): void
getAgreementContracts(): String
setAgreementContracts(agreementContracts:String): void
getUsernames(): String
setUsernames(username:String): void
getPasswords(): String
setPasswords(passwords:String): void
getRegistration(): String
setRegistration(registration:String): void
Car
available: boolean
carColor: String
carModel: String
carYear: int
carVIN: String
carPrice: double
updateCar(carModel:String): void
viewCar(carModel:String): String
deleteCar(carVIN:String): void
registarCar(carVIN:String): void
isAvailable(rentLocation:String): boolean
carModel(carVIN:string): void
carYear(carVIN:string): void
carInfo(carVIN:String, carModel:String, carYear:int, carColor:String, carPrice:double): String
getCarVIN(): String
setCarVIN(carVIN:String): void
getCarModel(): String
setCarModel(carModel:String): void
getCarYear(): int
setCarYear(carYear:int): void
getCarColor(): String
setCarColor(carColor:String): void
getCarPrice(): double
setCarPrice(carPrice:double): void

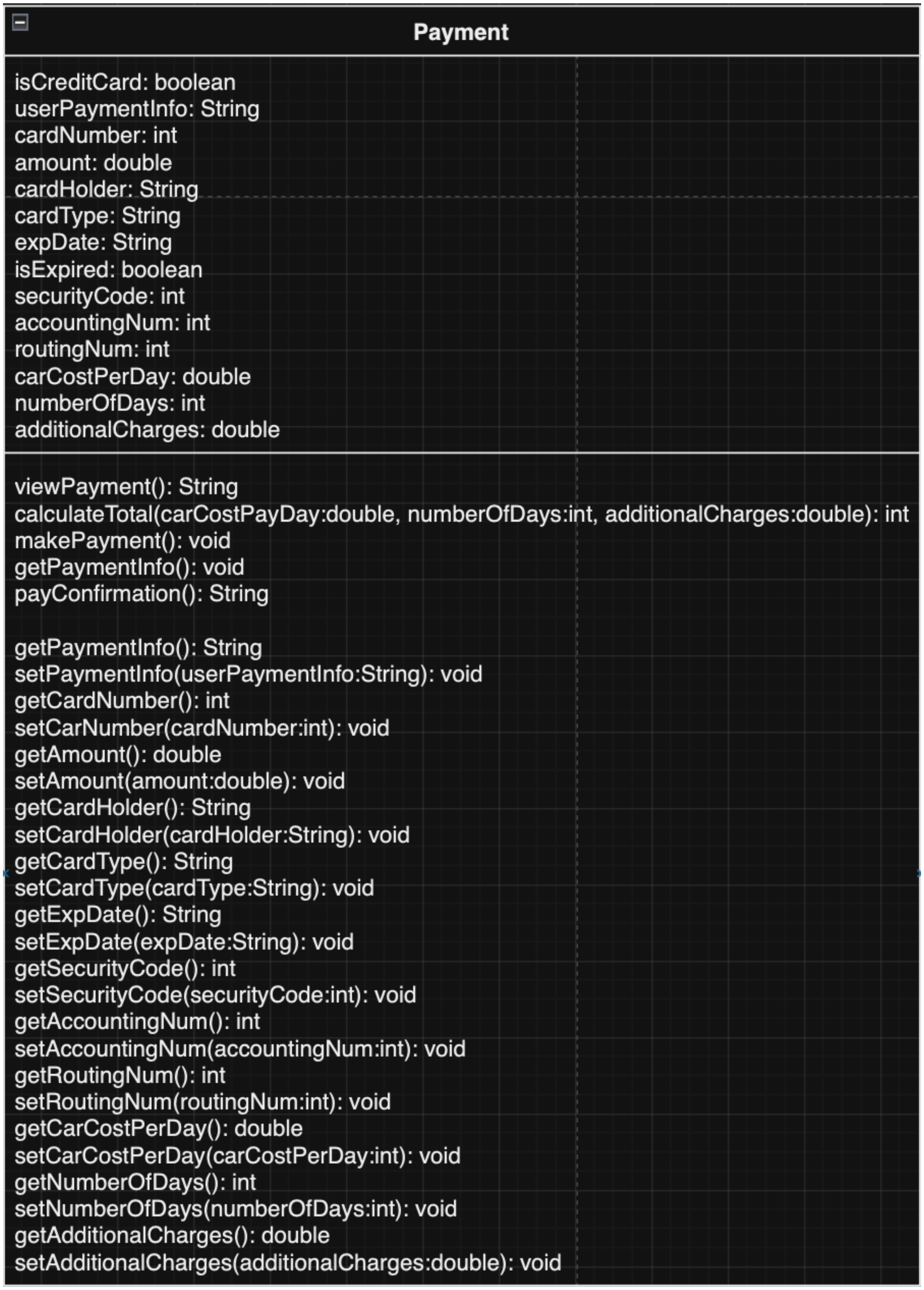


User

username: String
password: String
email: String
address: String
paymentInfo: String
firstName: String
lastName: String
dataOfBirth: String
haveAccount: boolean
driverLicense: String

checkUserInfo(username:String, password:String, email:String): String
lookUpLocations(rentLocation:String): String
logout(): void
login(username:String, password:String): void
sendEmailVerification(email:String): void
haveAccount(): boolean
makeAccount(username:String, email:String, password:String): void
linkPayment(username:String, paymentInfo:String): void
editProfile(username:String, address:String, password:String, email:String, driverLicens:String): void

getPassword(): String
setPassword(password:String): void
getUsername(): String
setUsername(username:String): void
getEmail(): String
setEmail(email:String): void
getAddress(): String
setAddress(address:String): void
getPayment(): String
setPayment(paymentInfo:String): void
getFirstName(): String
setFirstName(firstName:String): void
getLastName(): String
setLastName(lastName:String): void
getDateOfBirth(): String
setDateOfBirth(dateOfBirth:String): void
getDriverLicense(): String
setDriverLicense(driverLicense:String): void

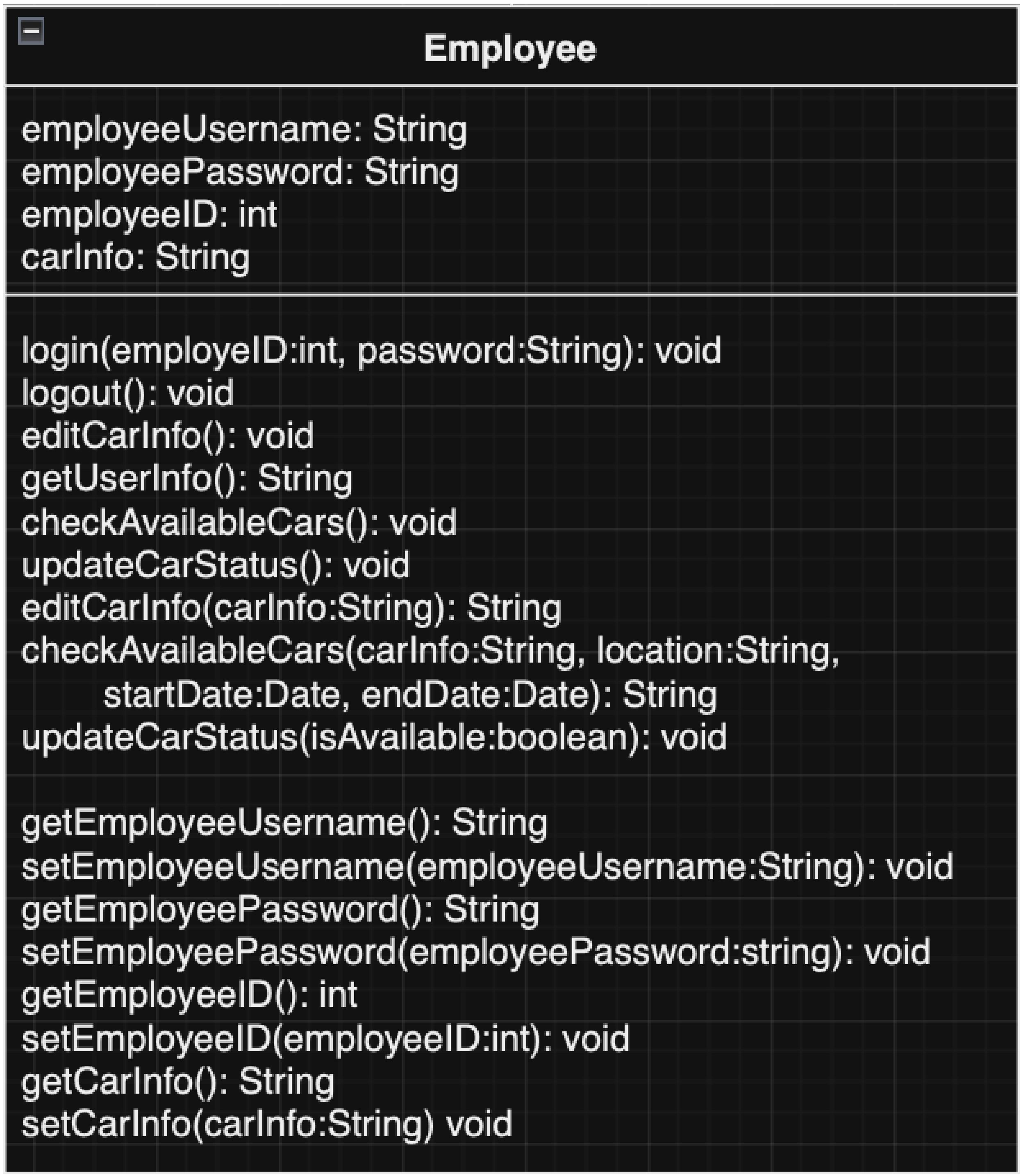


Payment

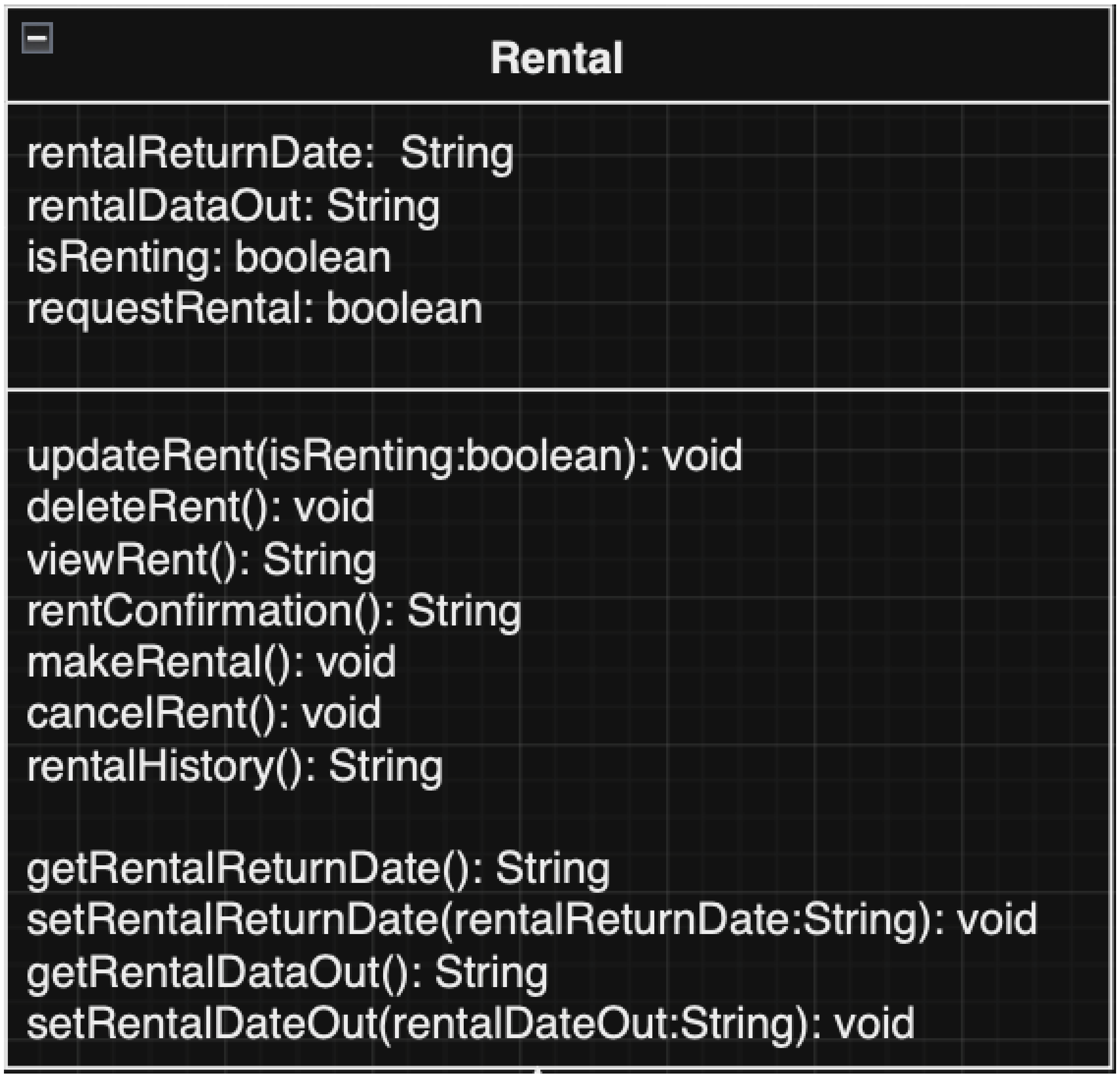
isCreditCard: boolean
userPaymentInfo: String
cardNumber: int
amount: double
cardHolder: String
cardType: String
expDate: String
isExpired: boolean
securityCode: int
accountingNum: int
routingNum: int
carCostPerDay: double
numberOfDays: int
additionalCharges: double

viewPayment(): String
calculateTotal(carCostPayDay:double, numberOfDays:int, additionalCharges:double): int
makePayment(): void
getPaymentInfo(): void
payConfirmation(): String

getPaymentInfo(): String
setPaymentInfo(userPaymentInfo:String): void
getCardNumber(): int
setCarNumber(cardNumber:int): void
getAmount(): double
setAmount(amount:double): void
getCardHolder(): String
setCardHolder(cardHolder:String): void
getCardType(): String
setCardType(cardType:String): void
getExpDate(): String
setExpDate(expDate:String): void
getSecurityCode(): int
setSecurityCode(securityCode:int): void
getAccountingNum(): int
setAccountingNum(accountingNum:int): void
getRoutingNum(): int
setRoutingNum(routingNum:int): void
getCarCostPerDay(): double
setCarCostPerDay(carCostPerDay:int): void
getNumberOfDays(): int
setNumberOfDays(numberOfDays:int): void
getAdditionalCharges(): double
setAdditionalCharges(additionalCharges:double): void



```
Employee
---------------------------------
employeeUsername: String
employeePassword: String
employeeID: int
carInfo: String
---------------------------------
login(employeID:int, password:String): void
logout(): void
editCarInfo(): void
getUserInfo(): String
checkAvailableCars(): void
updateCarStatus(): void
editCarInfo(carInfo:String): String
checkAvailableCars(carInfo:String, location:String,
        startDate:Date, endDate:Date): String
updateCarStatus(isAvailable:boolean): void

getEmployeeUsername(): String
setEmployeeUsername(employeeUsername:String): void
getEmployeePassword(): String
setEmployeePassword(employeePassword:string): void
getEmployeeID(): int
setEmployeeID(employeeID:int): void
getCarInfo(): String
setCarInfo(carInfo:String) void
```

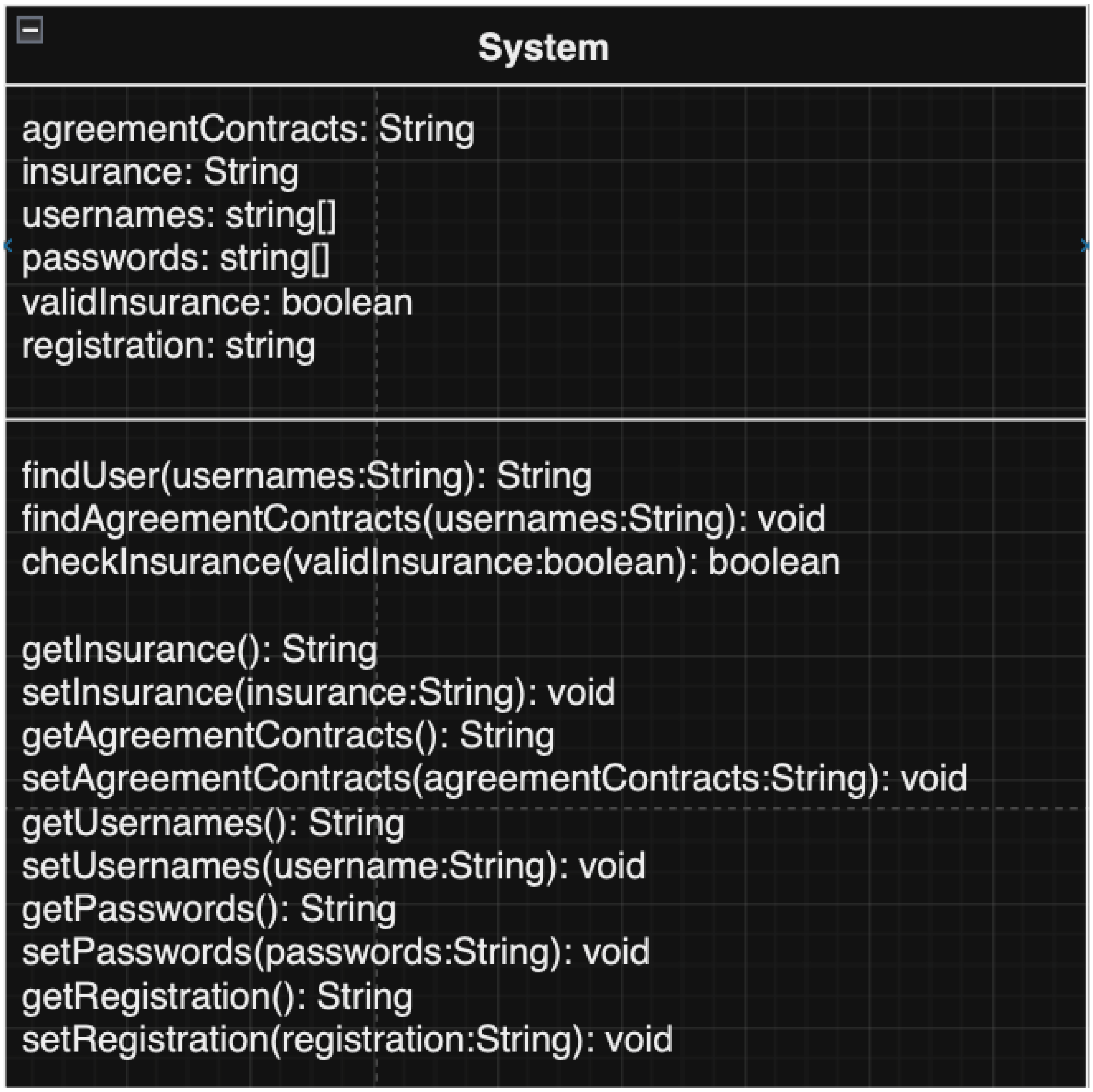


Rental

rentalReturnDate: String
rentalDataOut: String
isRenting: boolean
requestRental: boolean

updateRent(isRenting:boolean): void
deleteRent(): void
viewRent(): String
rentConfirmation(): String
makeRental(): void
cancelRent(): void
rentalHistory(): String

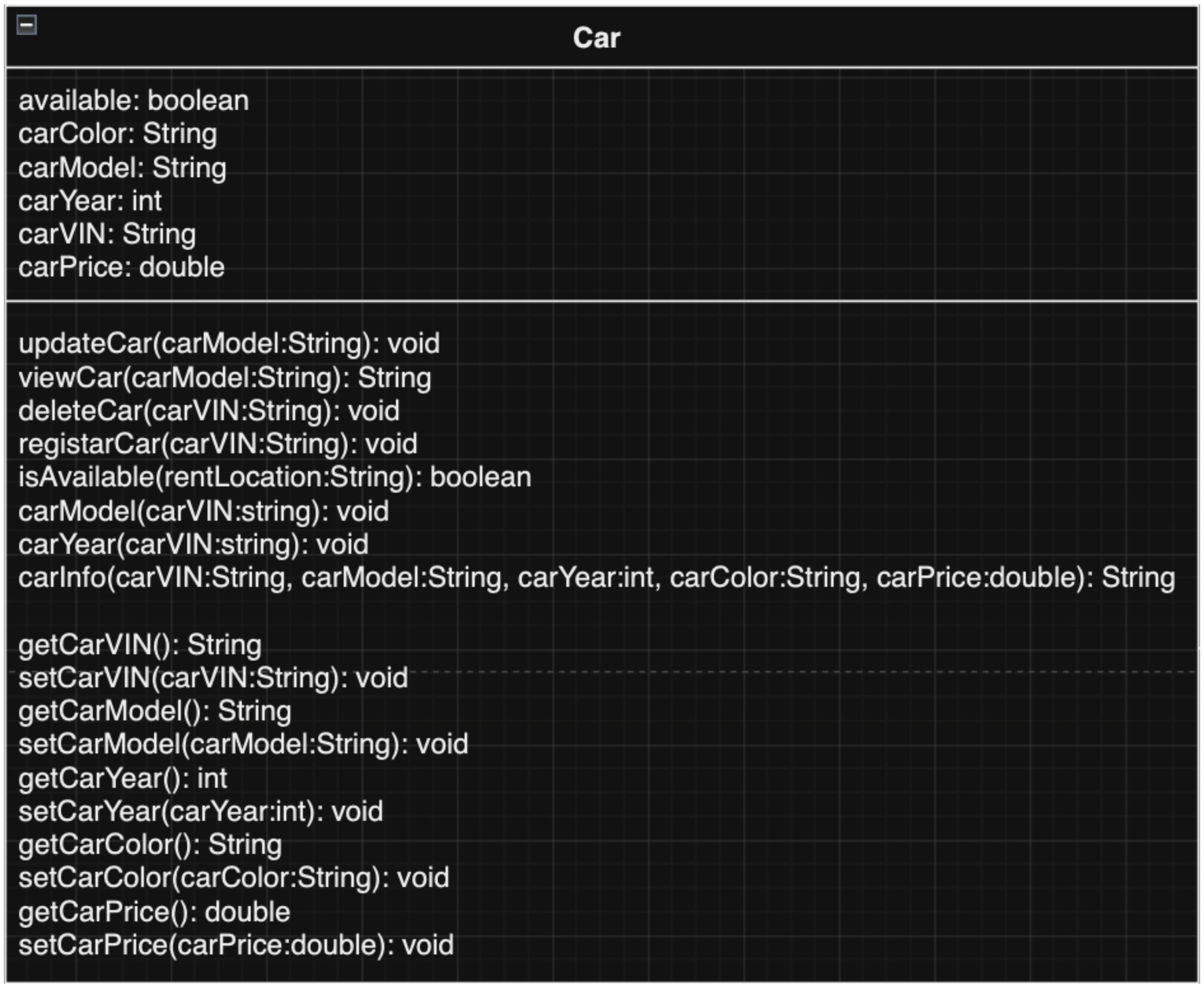
getRentalReturnDate(): String
setRentalReturnDate(rentalReturnDate:String): void
getRentalDataOut(): String
setRentalDateOut(rentalDateOut:String): void


System

agreementContracts: String
insurance: String
usernames: string[]
passwords: string[]
validInsurance: boolean
registration: string

findUser(usernames:String): String
findAgreementContracts(usernames:String): void
checkInsurance(validInsurance:boolean): boolean

getInsurance(): String
setInsurance(insurance:String): void
getAgreementContracts(): String
setAgreementContracts(agreementContracts:String): void
getUsernames(): String
setUsernames(username:String): void
getPasswords(): String
setPasswords(passwords:String): void
getRegistration(): String
setRegistration(registration:String): void

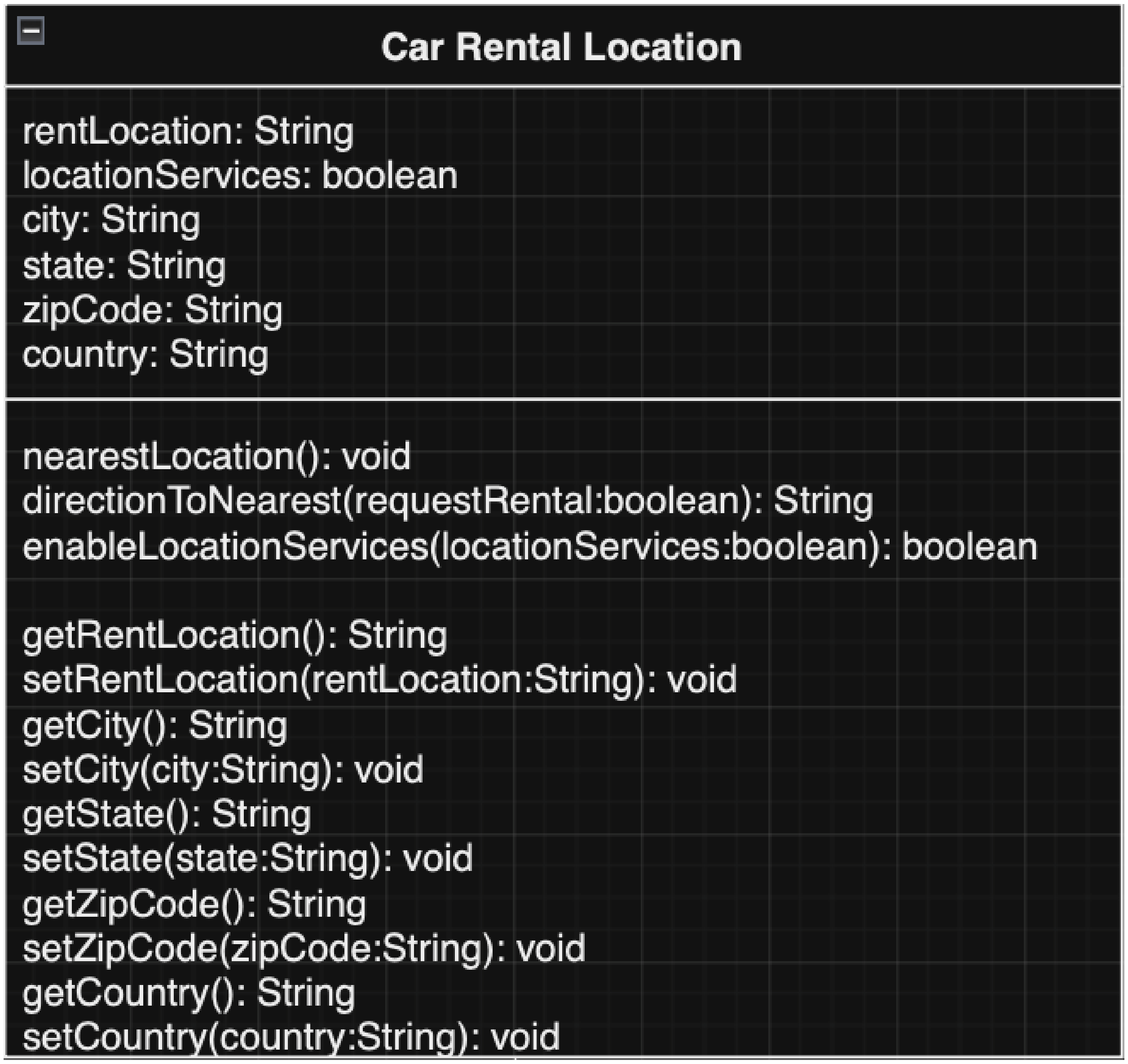


Car

available: boolean
carColor: String
carModel: String
carYear: int
carVIN: String
carPrice: double

updateCar(carModel:String): void
viewCar(carModel:String): String
deleteCar(carVIN:String): void
registarCar(carVIN:String): void
isAvailable(rentLocation:String): boolean
carModel(carVIN:string): void
carYear(carVIN:string): void
carInfo(carVIN:String, carModel:String, carYear:int, carColor:String, carPrice:double): String

getCarVIN(): String
setCarVIN(carVIN:String): void
getCarModel(): String
setCarModel(carModel:String): void
getCarYear(): int
setCarYear(carYear:int): void
getCarColor(): String
setCarColor(carColor:String): void
getCarPrice(): double
setCarPrice(carPrice:double): void



Car Rental Location

rentLocation: String
locationServices: boolean
city: String
state: String
zipCode: String
country: String

nearestLocation(): void
directionToNearest(requestRental:boolean): String
enableLocationServices(locationServices:boolean): boolean

getRentLocation(): String
setRentLocation(rentLocation:String): void
getCity(): String
setCity(city:String): void
getState(): String
setState(state:String): void
getZipCode(): String
setZipCode(zipCode:String): void
getCountry(): String
setCountry(country:String): void

2.3 Description of Classes

- `User:` The class contains attributes and operations regarding the user. The user class uses the Rental class and Car Rental Location class. The user class has options to make a rental, view rental history, lookup location, edit their profile information, etc. The class also has the payment class.

- `Payment:` The class contains attributes and operations regarding the payment process. The payment class securely stores the user's payment details for future purchases. This data is stored separately from the system class to minimize the chances of privacy breaches. The class also has the user class.

- `Employee:` The class uses other classes such as the System class, the Car class, and the User class. The employee class is able to carry out functions like reviewing rentals status/contracts, check available cars, update car status, etc.

- `Rental:` The rental class uses the Car class. The rental class contains all the functions regarding the rental process. The class is able to delete/make rent, view rent, update rent, and view rental history.

- `System:` The purpose of the system class is to store important company information such as user contracts, insurance, car registration, and user's username and password. Only employees can access this information using their employee login.

- `Car:` The car class uses the Car Rental Location Class to acquire information specific to that rent location . The car class maintains the company's vehicle information and can be modified by the employees. The class uses the unique pre-existing identification of the car, the VIN, to obtain the vehicle year and model.

- `Car Rental Location:` This class uses the Rental class to give the users the nearest rent location if the user has requested for a rental. The class also stores information regarding the company locations via city, state, zip code, and country.

2.4 Description of Attributes

User Class:

- `username` : a String that contains the username people using the system will input in order to log in to the system.

- `password` : a String that contains the password people using the system will input along with the username in order to log in to the system.

- `email` : a String that contains the email address attached to the user's account. This will be used to contact the user in case of notifications, updates, and rental charges.

- `address` : a String that contains the street address of the user.

- `paymentInfo` : a String that contains the payment information saved to the user's account. paymentInfo will not contain all of the information but instead will serve as a way to view your current payment method. This will work in conjunction with the Payment class, the Payment class includes attributes with the full payment information. The string will contain payment type (direct deposit, Visa, American Express, Mastercard, etc.), if it is a credit card it will also contain the expiration date and last four digits of the number, meanwhile if it is an eft it will contain the routing number and the last four digits of the accounting number.

- `firstName` : a String with the user's first name.

- `lastName` : a String with the user's last name.

- `dateOfBirth` : a String that contains the user's date of birth

- `haveAccount` : a boolean that determines whether a user already exists or needs to create a new account. This will return true if the username is found in the database and false otherwise, prompting the user to create an account.

- `driversLicense` : a String that contains the drivers license number of the user

Payment Class

- `isCreditCard` : a Boolean that contains whether the user is using a credit card to pay for the rental or not. If true, the system will prompt the user to enter what card type will be used, the credit card number, expiration date, and security code. If false, the user will be paying using EFT (electronic funds transfer) and will be asked to provide an accounting number and routing number.

- `amount` : a double that contains the amount the user must pay for a rental.
- `nameOnPayment` : a String that contains the name of the cardholder or the holder of the account funds are being withdrawn from.
- `isValid` : a Boolean that returns true if the entered information is correct.
- `cardType` : a String that contains the type of card the user is using, such as Visa, MasterCard, American Express, etc.
- `cardNumber` : an int that contains the saved credit card number on the user's account.
- `expDate` : an int that contains the saved credit card's expiration date on the user's account
- `isExpired` : a Boolean that contains if the card has expired or not. If it has (Boolean returns true) then the user will be unable to use the payment method saved to their account.
- `securityCode` : an int that contains the saved credit card's security code on the user's account.
- `accountingNum` : an int that contains the saved accounting number of the account the funds will be withdrawn from.
- `routingNum` : an int that contains the saved routing number of the bank the user's account is linked with.
- `carCostPerDay:` a double that contains the cost of renting a car per day.
- `numberOfDays:` an int that contains the number of days that the car is being rented for.
- `additionalCharges:` a double that contains the cost of additional charges on top of the rent fee.

<u>Employee Class</u>

- `employeeUsername` : a String that contains the username employees will input in order to log in to the system.

- `employeePassword`: a String that contains the password employees will input along with the username to log in to the system.

- `employeeID`: an int that contains a unique identification of the employe.e

- `carInfo:` a string that contains the information regarding the car. The string will store information such as the VIN, model, year, color, and price.

Rental Class

- `rentalReturnDate[]` : a String that contains what date a user's rental will be returned

- `rentalDateOut[]` : a String that contains what date a user's rental will be available

- `isRenting`: a Boolean that returns true if the user has picked up the rental and has yet to return it and false if the user has not picked up the rental yet or has already returned it

- `requestRental`: a Boolean that returns true if the user is looking to rent a vehicle

System Class

- `agreementContracts`: a String that contains the agreement contracts between the company and the users.

- `registration` : a String that contains the registration information for the cars.

- `usernames[]` : a String array that contains the usernames of the people using the system.

- `password[]` : a String array that contains the passwords of the users of the system.

- `validInsurance`: a Boolean that returns true if the insurance is valid.

Car Class

- `available`: a Boolean that returns true if the car is available for rental at the selected location and false otherwise

- `carColor` : a String that contains what color the rental car is

- `carMode`: a String that contains what model the rental car is

- `carYear` : an int that contains what year the car model is

- `carVIN` : a String that contains the VIN number of the car to identify which car is being rented

- `carPrice` : a double that contains the price of renting the car per day

<u>Car Rental Location Class</u>

- `rentLocation` : a String that contains which location a rental car belongs to

- `locationServices` : a Boolean that returns true if the user has enabled location services on their device and false if they have not. This is used so that the system can locate the nearest rental location.

- `city` : a String that contains what city the user would like to rent a car in

- `state` : a String that contains what state the user would like to rent a car in

- `zipCode` : a String that contains the zip code of the car rental location

- `country` : a String that contains which country the user would like to rent a car in

2.5 Description of Operations

<u>User Class</u>

- `checkUserInfo(username: String, password: String, email: String): String` - The checkUserInfo method is used to validate user information during the login process. It takes three parameters: username, password, and email. It checks the provided information against stored user data, and if the provided information is valid, it returns a string indicating success or appropriate error messages.

- `lookUpLocations(rentLocation:String): String` - The lookUpLocations method takes a rentLocation parameter, which is a string representing the rental location the user is interested in. This method is expected to look up detailed information about the specified rental location and return it as a string. This information may include the address, available vehicles, and rental rates for that location.

- `logout(): void` - The logout method is responsible for logging the user out of their account. This operation typically involves clearing the user's session or authentication tokens to ensure that the user is no longer authenticated and can't access restricted functionalities.

- `login(username:String, password:String): void` - The login method is used for user login. It takes two parameters: username and password. This method is responsible for authenticating the user by verifying the provided username and password. If the login is successful, it initiates a user session. This operation does not return a value.

- `sendEmailVerification(email:String): void` - Given an `email` parameter, this method sends an email verification to the provided email address. The email verification typically contains a link or code to confirm the user's email address. This is a security measure to ensure that the email provided is valid.

- `haveAccount(): boolean` - The haveAccount method is used to check whether a user has an existing account. It does not take any parameters and returns a boolean value, where true indicates that the user has an account, and false indicates that they do not.

- `makeAccount(username: String, email: String, password: String): void` – The makeAccount method is used to create a new user account. It takes three parameters: username, email, and password. This method does not return a value and is responsible for creating a new user account with the provided information.

- `linkPayment(username: String, paymentInfo: String): void` – The linkPayment method is used to associate payment information with a user account. This method does not return a value and is responsible for linking payment details to the user's account.

- `editProfile(username: String, address: String, password: String, email: String, driverLicense: String): void` – The editProfile method allows users to update their profile information. It takes multiple parameters including username, address, password, email, and driverLicense. This method does not return a value and is used to update user profile details, including contact information and driver's license details.

- `getPassword(): String` – The getPassword method retrieves the user's password and returns it as a string. This operation is typically used for user authentication and password reset purposes.

- `setPassword(password: String): void` – Given a new password as a parameter, this method updates the user's password. It does not return a value and involves securely storing the new password.

- `getUsername(): String` – The getUsername method retrieves the user's username and returns it as a string. This username is used for authentication and user identification.

- `setUsername(username: String): void` – Given a new username as a parameter, this method updates the user's username. It does not return a value and ensures the uniqueness of the new username.

- `getEmail(): String:` – The getEmail method retrieves the user's email address and returns it as a string. The email is an essential piece of user contact information.

- `setEmail(email: String): void` – Given a new email as a parameter, this method updates the user's email address. It does not return a value and typically involves email address validation.

- `getAddress(): String` – The getAddress method retrieves the user's address and returns it as a string. This can include the user's physical address or mailing address.
- `setAddress(address: String): void` – Given a new address as a parameter, this method updates the user's address. It does not return a value and ensures that the address format is valid.
- `getPayment(): String` – The getPayment method retrieves the user's payment information, such as credit card details or payment method names, and returns it as a string. This information is necessary for processing payments for rental services.
- `setPayment(paymentInfo: String): void` – Given new payment information as a parameter, this method updates the user's payment information. It does not return a value and ensures the security of the payment data.
- `getFirstName(): String:` – The getFirstName method retrieves the user's first name and returns it as a string. This is part of the user's profile information.
- `setFirstName(firstName: String): void` – Given a new firstName as a parameter, this method updates the user's first name. It does not return a value and ensures that the first name is within valid constraints.
- `getLastName(): String` – The getLastName method retrieves the user's last name and returns it as a string. This is another part of the user's profile information.
- `setLastName(lastName: String): void` – Given a new lastName as a parameter, this method updates the user's last name. It does not return a value and ensures that the last name is within valid constraints.
- `getDateOfBirth(): String` – The getDateOfBirth method retrieves the user's date of birth and returns it as a string. This information may be used for age verification or personalized services.
- `setDateOfBirth(dateOfBirth: String): void` – Given a new dateOfBirth as a parameter, this method updates the user's date of birth. It does not return a value and ensures that the date of birth is in the correct format and within valid constraints.

- `getDriverLicense(): String` - The getDriverLicense method retrieves the user's driver's license information and returns it as a string. This is an important piece of information for verification purposes, especially in the context of car rentals. The driver's license information typically includes the license number, expiration date, and other relevant details.

- `setDriverLicense(driverLicense: String): void` - Given a new driverLicense as a parameter, the setDriverLicense method updates the user's driver's license information. This operation does not return a value and ensures that the provided driver's license information is stored securely and in the correct format. The user may be required to provide a scanned or photographed image of their driver's license for validation.

- `makeRental(car : Car, payment: Payment, carLocation: CarLocation, rental: Rental, system: System): void` - Given the car class, the payment class, the carLocation class, the rental class, and the system class, the method takes the user through the process of renting a car from start to finish.

<u>**Payment Class**</u>

- `viewPayment(): String` - The viewPayment method is used to view payment information. It does not take any parameters, and it returns a string containing payment details. This operation typically displays payment information such as the card type, last four digits of the card number, and the expiration date to the user. The returned string provides a formatted representation of the payment details for user reference or confirmation.

- `calculateTotal(carCostPerDay: double, numberOfDays: int, additionalCharges: double): int` - The calculateTotal method is responsible for calculating the total payment amount for a car rental. The method performs the calculation using the provided parameters and returns the total payment amount as an integer. The total payment amount is typically computed by multiplying the carCostPerDay by the numberOfDays and then adding the additional charges such as taxes or fees. The result is an integer representing the final payment amount.

- `payConfirmation(): String` - The payConfirmation method confirms the payment. It does not take any parameters, and it returns a string representing a unique transaction ID. This transaction ID serves as a reference and proof of the successful payment transaction. It can be used for tracking and customer support purposes.

- `getPaymentInfo(): String` – The getPaymentInfo method retrieves the user's payment information and returns it as a string. This information typically includes details such as the card type, the last four digits of the card number, and the expiration date.
- `setPaymentInfo(userPaymentInfo: String): void` – Given new payment information as a parameter (userPaymentInfo), the setPaymentInfo method updates the user's payment information. It does not return a value and ensures that the provided payment information is securely stored.
- `getCardNumber(): int` – The getCardNumber method retrieves the card number and returns it as an integer. This operation returns the full card number, which is typically not displayed for security reasons.
- `setCardNumber(cardNumber: int): void` – Given a new card number as a parameter (cardNumber), the setCardNumber method updates the card number. It does not return a value and ensures that the card number is securely stored and validated.
- `getAmount(): double` – The getAmount method retrieves the payment amount and returns it as a double. This amount represents the total payment to be processed.
- `setAmount(amount: double): void` – Given a new payment amount as a parameter (amount), the setAmount method updates the payment amount. It does not return a value and ensures that the payment amount is stored accurately.
- `getCardHolder(): String` – The getCardHolder method retrieves the cardholder's name and returns it as a string. This is the name associated with the payment method.
- `setCardHolder(cardHolder: String): void` – Given a new cardholder's name as a parameter (cardHolder), the setCardHolder method updates the cardholder's name. It does not return a value and ensures that the cardholder's name is stored accurately.
- `getCardType(): String` – The getCardType method retrieves the card type (e.g., Visa, MasterCard) and returns it as a string. This indicates the type of credit or debit card associated with the payment method.
- `setCardType(cardType: String): void` – Given a new card type as a parameter (cardType), the setCardType method updates the card type. It does not return a value and ensures that the card type is stored accurately.

- `getExpDate(): String` – The getExpDate method retrieves the card's expiration date and returns it as a string. This is typically represented in the format "MM/YYYY."

- `setExpDate(expDate: String): void` – Given a new expiration date as a parameter (expDate), the setExpDate method updates the card's expiration date. It does not return a value and ensures that the expiration date is stored accurately.

- `getSecurityCode(): int` – The getSecurityCode method retrieves the card's security code (CVV or CVC) and returns it as an integer. This code is used for additional security verification during payment.

- `setSecurityCode(securityCode: int): void` – Given a new security code as a parameter (securityCode), the setSecurityCode method updates the card's security code. It does not return a value and ensures that the security code is stored securely.

- `getAccountingNum(): int` – The getAccountingNum method retrieves additional accounting information associated with the payment method and returns it as an integer. This information is sometimes used in business or corporate accounts.

- `setAccountingNum(accountingNum: int): void` – Given new accounting information as a parameter (accountingNum), the setAccountingNum method updates the accounting number. It does not return a value and ensures that the accounting information is stored accurately.

- `getRoutingNum(): int` – The getRoutingNum method retrieves the routing number associated with the payment method and returns it as an integer. Routing numbers are commonly used for bank transfers and direct deposits.

- `setRoutingNum(routingNum: int): void` – Given a new routing number as a parameter (routingNum), the setRoutingNum method updates the routing number. It does not return a value and ensures that the routing number is stored accurately.

- `makeOnlinePayment(PaymentInfo: String): void` – This function facilitates online payments with a single argument, PaymentInfo, which is a string containing all necessary payment information. It returns a boolean value, where true indicates a successful payment transaction, and false signifies a failed or declined payment.

- `makeCreditCardPayment(PaymentInfo: String): void` – This function handles credit card payments with the CreditCardInfo parameter, which is a string containing credit card details. It returns a boolean value, where true indicates a successful credit card payment, and false represents a declined or failed transaction.

Employee Class

- `login(employeeID: int, password: string): void` – The login method is used for employee login. It takes an employeeID (integer) and a password (string) as parameters to verify the employee's identity and initiate a session. This operation does not return a value.

- `logout(): void` – The logout method is used to log out the employee, terminating the session. It does not take any parameters and does not return a value.

- `editCarInfo(carInfo: String): String` – The editCarInfo method allows employees to edit car information. It takes a carInfo (string) parameter, which represents the updated car information, and returns a string indicating the result of the edit operation, such as success or failure.

- `getUserInfo(): String` – The getUserInfo method is used to retrieve information about the employee. It does not take any parameters and returns a string containing details about the employee, including their name, contact information, and role.

- `checkAvailableCars(carInfo: String, location: String, startDate: Date, endDate: Date): String` – The checkAvailableCars method is used to check the availability of cars for a specific location and time period. It takes parameters such as carInfo (string), location (string), startDate (Date), and endDate (Date) to determine the availability of cars. It returns a string containing information about available cars during the specified time frame.

- `getEmployeeUsername(): String` – The getEmployeeUsername method retrieves the employee's username and returns it as a string. This username is used for employee identification.

- `setEmployeeUsername(employeeUsername: String): void` – Given a new employeeUsername as a parameter, the setEmployeeUsername method updates the employee's username. It does not return a value and ensures that the new username is stored accurately.

- `getEmployeePassword(): String` – The getEmployeePassword method retrieves the employee's password and returns it as a string. This password is typically used for employee authentication.

- `setEmployeePassword(employeePassword: string): void` – Given a new employeePassword as a parameter, the setEmployeePassword method updates the employee's password. It does not return a value and ensures that the new password is stored securely.

- `getEmployeeID(): int` – The getEmployeeID method is used to retrieve the employee's unique identification number (ID). It does not take any parameters and returns the employee's ID as an integer.

- `setEmployeeID(employeeID:int): void` – Given a new employeeID as a parameter, the setEmployeeID method updates the employee's identification number (ID). It does not return a value and ensures that the new ID is stored accurately.

- `updateCarStatus(car : Car, carLocation: CarLocation): void` - This function is used to update the status, location, and availability of a car. It takes three parameters: car, which represents the car to be updated, carLocation, which specifies the current or new location of the car. Given the car class and car location as a parameter, updateCarStatus method updates the car's status when it undergoes or finishes maintenance.

Rental Class:

- `updateRent(isRenting: boolean): void` – The updateRent method is used to update the rental status of a vehicle. It takes an isRenting parameter, a boolean value, where true indicates that the vehicle is currently being rented, and false indicates that it is not. This method does not return a value and is responsible for updating the rental status of the vehicle.

- `viewRent(): String` – The viewRent method is used to view details about the rental. It does not take any parameters and returns a string containing information about the rental, including the rental period, vehicle details, and rental charges.

- `rentConfirmation(): String` – The rentConfirmation method provides a confirmation message for a rental. It does not take any parameters and returns a string representing a confirmation message, typically including details of the rental, such as the rental period, vehicle details, and charges.

- `cancelRent(): void` – The cancelRent method is used to cancel a rental reservation. It does not take any parameters and does not return a value. This operation typically involves releasing the reserved vehicle and canceling the rental booking.

- `rentalHistory(): String` – The rentalHistory method is used to view the rental history. It does not take any parameters and returns a string containing the user's rental history, including past rental transactions, rental dates, and vehicle details.

- `getRentalReturnDate(): String` – The getRentalReturnDate method retrieves the expected return date of a rental and returns it as a string. This date indicates when the rented vehicle is expected to be returned.

- `setRentalReturnDate(rentalReturnDate: String): void` – Given a new rentalReturnDate as a parameter, the setRentalReturnDate method updates the expected return date of the rental. It does not return a value and ensures that the new return date is stored accurately.

- `getRentalDateOut(): String` – The getRentalDateOut method retrieves the rental date (start date) and returns it as a string. This date represents the beginning of the rental period.

- `setRentalDateOut(rentalDateOut: String): void` – Given a new rentalDateOut as a parameter, the setRentalDateOut method updates the rental date (start date). It does not return a value and ensures that the new start date is stored accurately.

System Class:

- `findUser(username: String): String` – The findUser method is used to search for user information based on a username. It takes a username parameter as a string and returns a string containing user information. This operation typically retrieves and displays user details such as name, contact information, and account status.

- `findAgreementContracts(username: String): String` – The findAgreementContracts method is used to search for agreement contracts associated with a specific user based on their username. It takes a username parameter as a string and returns a string containing agreement contract details. This operation retrieves and displays information about agreements, terms, and conditions that the user has accepted or is currently bound by.

- `checkInsurance(validInsurance: boolean): boolean` – The checkInsurance method is used to verify whether a user has valid insurance. It takes a validInsurance parameter as a boolean, where true indicates valid insurance, and false indicates invalid insurance. It returns a boolean value, where true confirms that the insurance is valid, and false confirms that it is not valid.

- `getInsurance(): String` – The getInsurance method retrieves information about the user's insurance and returns it as a string. This typically includes details about the type of insurance, coverage, and expiration date.
- `setInsurance(insurance: String): void` – Given new insurance information as a parameter (insurance), the setInsurance method updates the user's insurance details. It does not return a value and ensures that the new insurance information is stored accurately.
- `getAgreementContracts(): String` – The getAgreementContracts method retrieves information about the user's agreement contracts and returns it as a string. This typically includes the terms and conditions of agreements that the user has accepted.
- `setAgreementContracts(agreementContracts: String): void` – Given new agreement contract information as a parameter (agreementContracts), the setAgreementContracts method updates the user's agreement contracts. It does not return a value and ensures that the new contract information is stored accurately.
- `getUsernames(): String` – The getUsernames method retrieves usernames associated with the system and returns them as a string. This can be used to display a list of usernames for administrative or reference purposes.
- `setUsernames(username: String): void` – Given new username information as a parameter (username), the setUsernames method updates the list of usernames in the system. It does not return a value and ensures that the new username is stored accurately.
- `getUserPasswords(): String` – The getUserPasswords method retrieves user passwords and returns them as a string. This is typically for administrative purposes and may involve masked or encrypted passwords.
- `setUserPasswords(userPassword: String): void` – Given new user password information as a parameter (userPassword), the setUserPasswords method updates user passwords in the system. It does not return a value and ensures that the new password information is stored securely.

- `getRegistration(): String` – The getRegistration method retrieves registration information for users and returns it as a string. This typically includes details about the user's registration status and date of registration.

- `setRegistration(registration: String): void` – Given new registration information as a parameter (registration), the setRegistration method updates the user's registration details. It does not return a value and ensures that the new registration information is stored accurately.

Car Class:

- `carInfo(carVIN: String,carModel: String, carYear: int, carColor: String, carPrice: double): String` – The carInfo method is used to set or update information about a car. This method allows for setting or updating information about a car, including its VIN, model, year, color, and rental price. It is typically used during car registration or updating car details in the inventory.

- `updateCar(carModel: String): void` – The updateCar method is used to update the car model. It takes a carModel parameter as a string, representing the new car model. This method does not return a value and is responsible for updating the car's model information.

- `viewCar(carModel: String): String` – The viewCar method allows users to view information about a specific car model. It takes a carModel parameter as a string and returns a string containing details about the specific car model. This can include information such as the car's VIN (Vehicle Identification Number), year, color, and price.

- `deleteCar(carVIN: String): void` – The deleteCar method is used to remove a car from the inventory based on its VIN (Vehicle Identification Number). It takes a carVIN parameter as a string and does not return a value. This operation typically involves marking the car as no longer available for rental.

- `registerCar(carVIN: String): void` – The registerCar method registers a new car in the inventory by specifying its VIN. It takes a carVIN parameter as a string and does not return a value. This operation typically involves adding the new car to the list of available vehicles for rental.

- `isAvailable(rentLocation: String): boolean` – The isAvailable method checks the availability of a car at a specific rental location. It takes a rentLocation parameter as a string, representing the location to check, and returns a boolean value. If the car is available at the specified location, it returns true, otherwise, it returns false.

- `getCarModel(carVIN: string): String` – The getCarModel method retrieves the car model associated with a specific car's VIN. It takes a carVIN parameter as a string and returns the car model as a string.

- `getCarYear(carVIN: string): void` – The getCarYear method retrieves the car's manufacturing year associated with a specific car's VIN. It takes a carVIN parameter as a string and returns the car year as an integer.

- `getCarVIN(): String` – The getCarVIN method retrieves the Vehicle Identification Number (VIN) of the car and returns it as a string. The VIN is a unique identifier for each car.

- `setCarVIN(carVIN: String): void` – Given a new VIN as a parameter (carVIN), the setCarVIN method updates the car's VIN. It does not return a value and ensures that the new VIN is stored accurately.

- `getCarModel(): String` – The getCarModel method retrieves the car model and returns it as a string. The car model typically represents the make and model of the car.

- `setCarModel(carModel: String): void` – Given a new car model as a parameter (carModel), the setCarModel method updates the car model. It does not return a value and ensures that the new car model is stored accurately.

- `getCarYear(): int` – The getCarYear method retrieves the manufacturing year of the car and returns it as an integer. This indicates the year in which the car was manufactured.

- `setCarYear(carYear: int): void` – Given a new car year as a parameter (carYear), the setCarYear method updates the car's manufacturing year. It does not return a value and ensures that the new year is stored accurately.

- `getCarColor(): String` – The getCarColor method retrieves the color of the car and returns it as a string. This represents the exterior color of the vehicle.

- `setCarColor(carColor: String): void` – Given a new car color as a parameter (carColor), the setCarColor method updates the car's color. It does not return a value and ensures that the new color is stored accurately.

- `getCarPrice(): double` – The getCarPrice method retrieves the rental price of the car and returns it as a double. This indicates the cost of renting the car.

- `setCarPrice(carPrice: double): void` – Given a new rental price as a parameter (carPrice), the setCarPrice method updates the car's rental price. It does not return a value and ensures that the new price is stored accurately.

<u>Car Rental Location Class:</u>

- `nearestLocation(): void` – The nearestLocation method is used to find the nearest car rental location. It does not take any parameters and does not return a value. This operation typically involves determining the user's current location or preferences and identifying the nearest available rental location.

- `directionToNearest(requestRental: boolean): String` – The directionToNearest method provides directions to the nearest car rental location. It takes a requestRental parameter, a boolean value, to determine if the user wants to rent a car at the nearest location. It returns a string containing directions to the nearest location, which can include a map or step-by-step instructions.

- `enableLocationServices(locationServices: boolean): boolean` – The enableLocationServices method allows the user to enable or disable location services. It takes a locationServices parameter, a boolean value, to indicate whether the location services are enabled or disabled. It returns a boolean value, where true indicates that the services are enabled, and false indicates they are disabled.

- `getRentLocation(): String` – The getRentLocation method retrieves the name or identifier of the car rental location and returns it as a string. This identifies the specific car rental branch or location.

- `setRentLocation(rentLocation: String): void` – Given a new rentLocation as a parameter, the setRentLocation method updates the car rental location. It does not return a value and ensures that the new location information is stored accurately.

- `getCity(): String` – The getCity method retrieves the city where the car rental location is situated and returns it as a string.
  setCity(city: String): void: Given a new city as a parameter (city), the setCity method updates the city where the car rental location is situated. It does not return a value and ensures that the new city information is stored accurately.

- `getState(): String` – The getState method retrieves the state or region where the car rental location is situated and returns it as a string.
- `setState(state: String): void` – Given a new state or region as a parameter (state), the setState method updates the state information for the car rental location. It does not return a value and ensures that the new state information is stored accurately.
- `getZipCode(): String` – The getZipCode method retrieves the ZIP code or postal code associated with the car rental location and returns it as a string.
- `setZipCode(zipCode: String): void` – Given a new ZIP code or postal code as a parameter (zipCode), the setZipCode method updates the ZIP code information for the car rental location. It does not return a value and ensures that the new ZIP code information is stored accurately.
- `getCountry(): String` – The getCountry method retrieves the country where the car rental location is situated and returns it as a string.
- `setCountry(country: String): void` – Given a new country as a parameter (country), the setCountry method updates the country information for the car rental location. It does not return a value and ensures that the new country information is stored accurately.

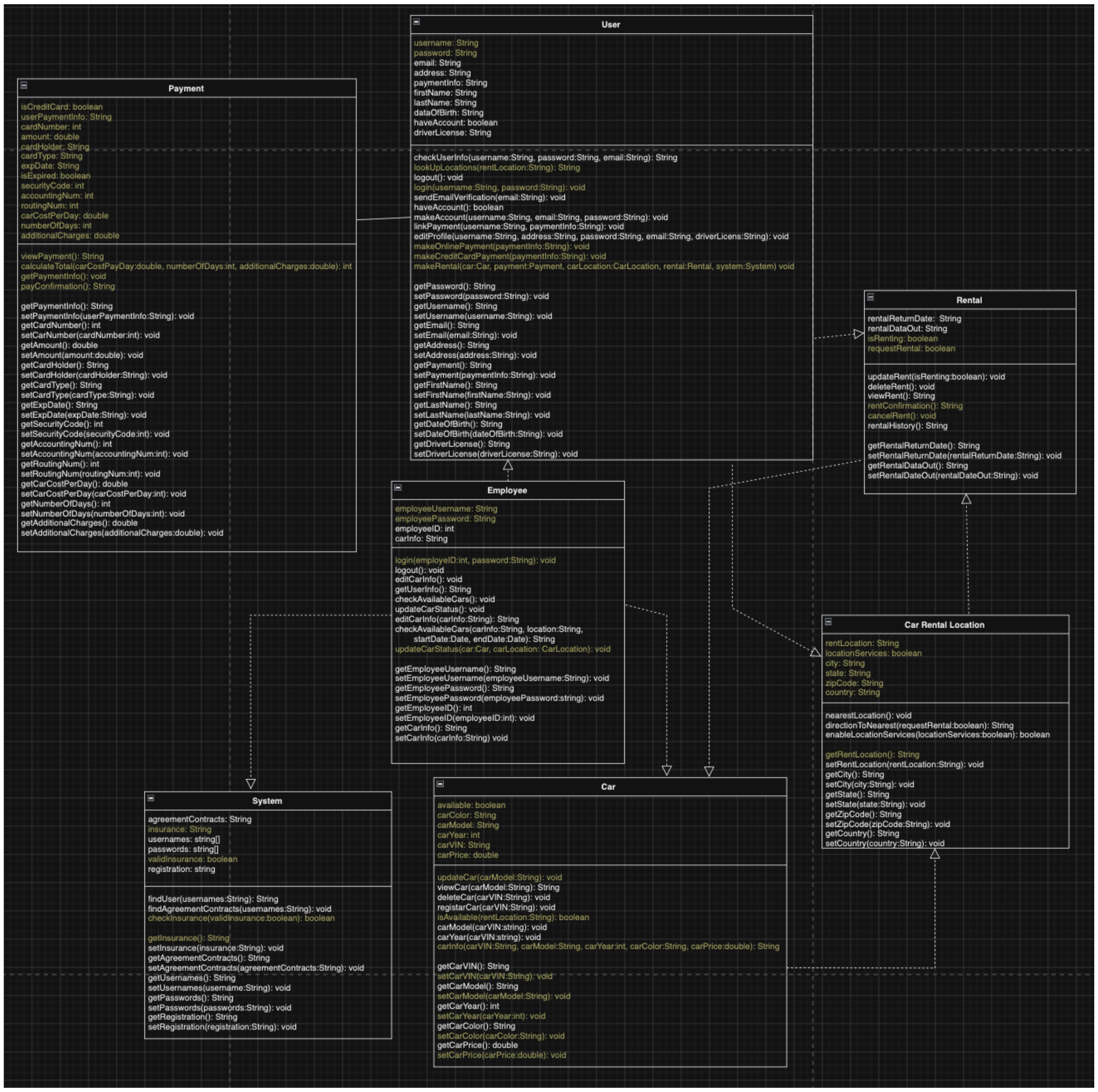
3 Development Plan & Timeline

Estimated Time	Task	Team Member Responsible
Day 1	Getting familiar with the software system, brainstorming ideas, and partitioning tasks	Collaborative
	Writing the Development plan and timeline section	James Duong
Day 2 - 3	Creating the Software Architecture Diagram	Sukruti Mallesh
	Creating the UML Diagram	Alyssa Rivera and James Duong
Day 4	Meeting up to make sure everyone is on track and to review each other’s work	Collaborative
Day 5 - 6	Writing the Architectural diagram of all major components	Sukruti Mallesh
	Writing the description of classes, attributes, and operations	Collaborative
	Writing the overview of the system	Alyssa Rivera
Day 7	Reviewing each other’s work and putting the finishing touches	Collaborative

4 Software Design Specification Test Plan

Updated Design Specification

4.1 Updated Design Diagram:



```
User
username: String
password: String
email: String
address: String
paymentInfo: String
firstName: String
lastName: String
dataOfBirth: String
haveAccount: boolean
driverLicense: String

checkUserInfo(username:String, password:String, email:String): String
lookUpLocations(rentLocation:String): String
logout(): void
login(username:String, password:String): void
sendEmailVerification(email:String): void
haveAccount(): boolean
makeAccount(username:String, email:String, password:String): void
linkPayment(username:String, paymentInfo:String): void
editProfile(username:String, address:String, password:String, email:String, driverLicens:String): void
makeOnlinePayment(paymentInfo:String): void
makeCreditCardPayment(paymentInfo:String): void
makeRental(car:Car, payment:Payment, carLocation:CarLocation, rental:Rental, system:System) void

getPassword(): String
setPassword(password:String): void
getUsername(): String
setUsername(username:String): void
getEmail(): String
setEmail(email:String): void
getAddress(): String
setAddress(address:String): void
getPayment(): String
setPayment(paymentInfo:String): void
getFirstName(): String
setFirstName(firstName:String): void
getLastName(): String
setLastName(lastName:String): void
getDateOfBirth(): String
setDateOfBirth(dateOfBirth:String): void
getDriverLicense(): String
setDriverLicense(driverLicense:String): void
```

```
Payment
isCreditCard: boolean
userPaymentInfo: String
cardNumber: int
amount: double
cardHolder: String
cardType: String
expDate: String
isExpired: boolean
securityCode: int
accountingNum: int
routingNum: int
carCostPerDay: double
numberOfDays: int
additionalCharges: double

viewPayment(): String
calculateTotal(carCostPayDay:double, numberOfDays:int, additionalCharges:double): int
getPaymentInfo(): void
payConfirmation(): String

getPaymentInfo(): String
setPaymentInfo(userPaymentInfo:String): void
getCardNumber(): int
setCarNumber(cardNumber:int): void
getAmount(): double
setAmount(amount:double): void
getCardHolder(): String
setCardHolder(cardHolder:String): void
getCardType(): String
setCardType(cardType:String): void
getExpDate(): String
setExpDate(expDate:String): void
getSecurityCode(): int
setSecurityCode(securityCode:int): void
getAccountingNum(): int
setAccountingNum(accountingNum:int): void
getRoutingNum(): int
setRoutingNum(routingNum:int): void
getCarCostPerDay(): double
setCarCostPerDay(carCostPerDay:int): void
getNumberOfDays(): int
setNumberOfDays(numberOfDays:int): void
getAdditionalCharges(): double
setAdditionalCharges(additionalCharges:double): void
```

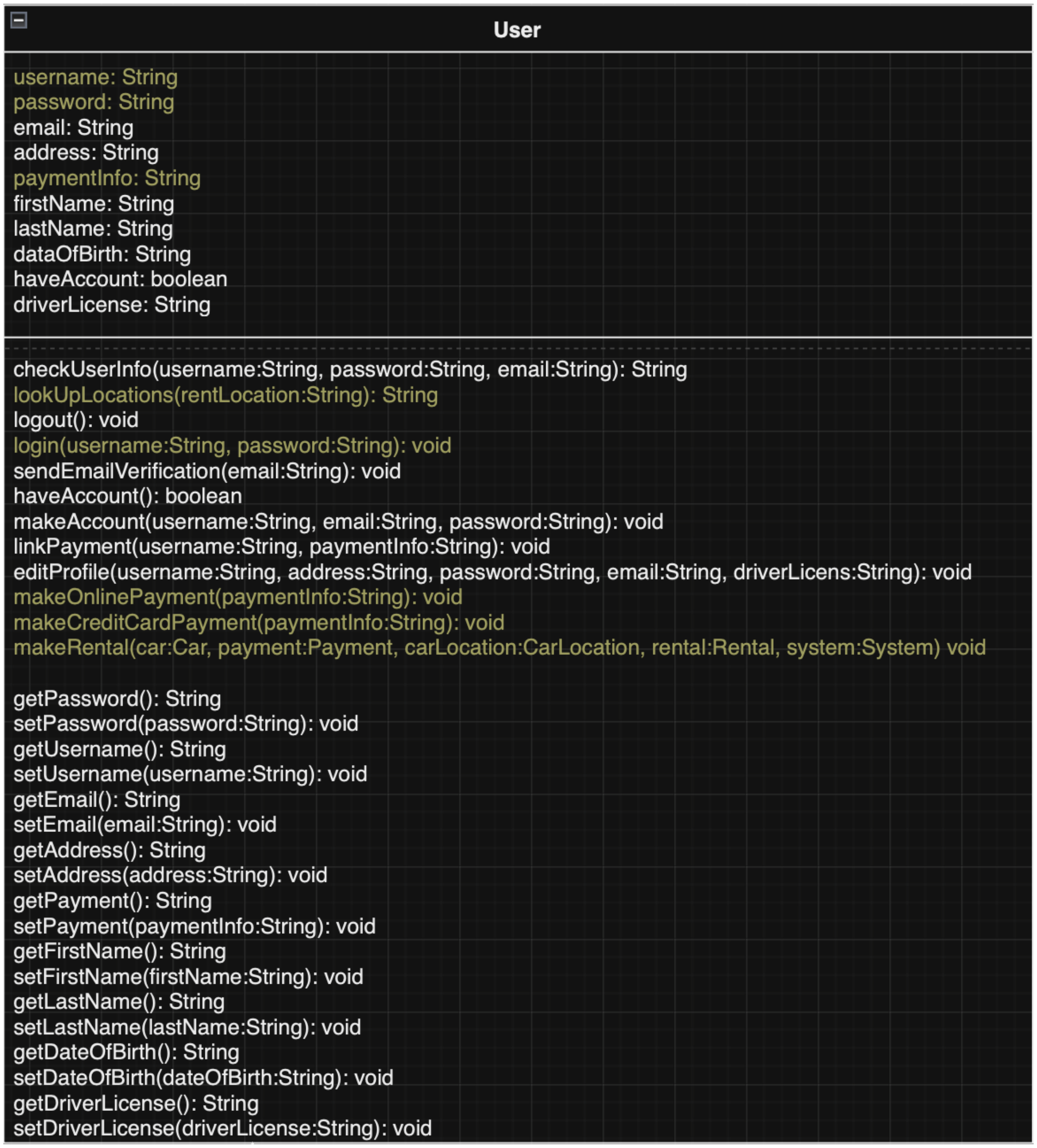
```
Rental
rentalReturnDate:  String
rentalDataOut: String
isRenting: boolean
requestRental: boolean

updateRent(isRenting:boolean): void
deleteRent(): void
viewRent(): String
rentConfirmation(): String
cancelRent(): void
rentalHistory(): String

getRentalReturnDate(): String
setRentalReturnDate(rentalReturnDate:String): void
getRentalDataOut(): String
setRentalDateOut(rentalDateOut:String): void
```

```
Employee
employeeUsername: String
employeePassword: String
employeeID: int
carInfo: String

login(employeID:int, password:String): void
logout(): void
editCarInfo(): void
getUserInfo(): String
checkAvailableCars(): void
updateCarStatus(): void
editCarInfo(carInfo:String): String
checkAvailableCars(carInfo:String, location:String,
        startDate:Date, endDate:Date): String
updateCarStatus(car:Car, carLocation: CarLocation): void

getEmployeeUsername(): String
setEmployeeUsername(employeeUsername:String): void
getEmployeePassword(): String
setEmployeePassword(employeePassword:string): void
getEmployeeID(): int
setEmployeeID(employeeID:int): void
getCarInfo(): String
setCarInfo(carInfo:String) void
```

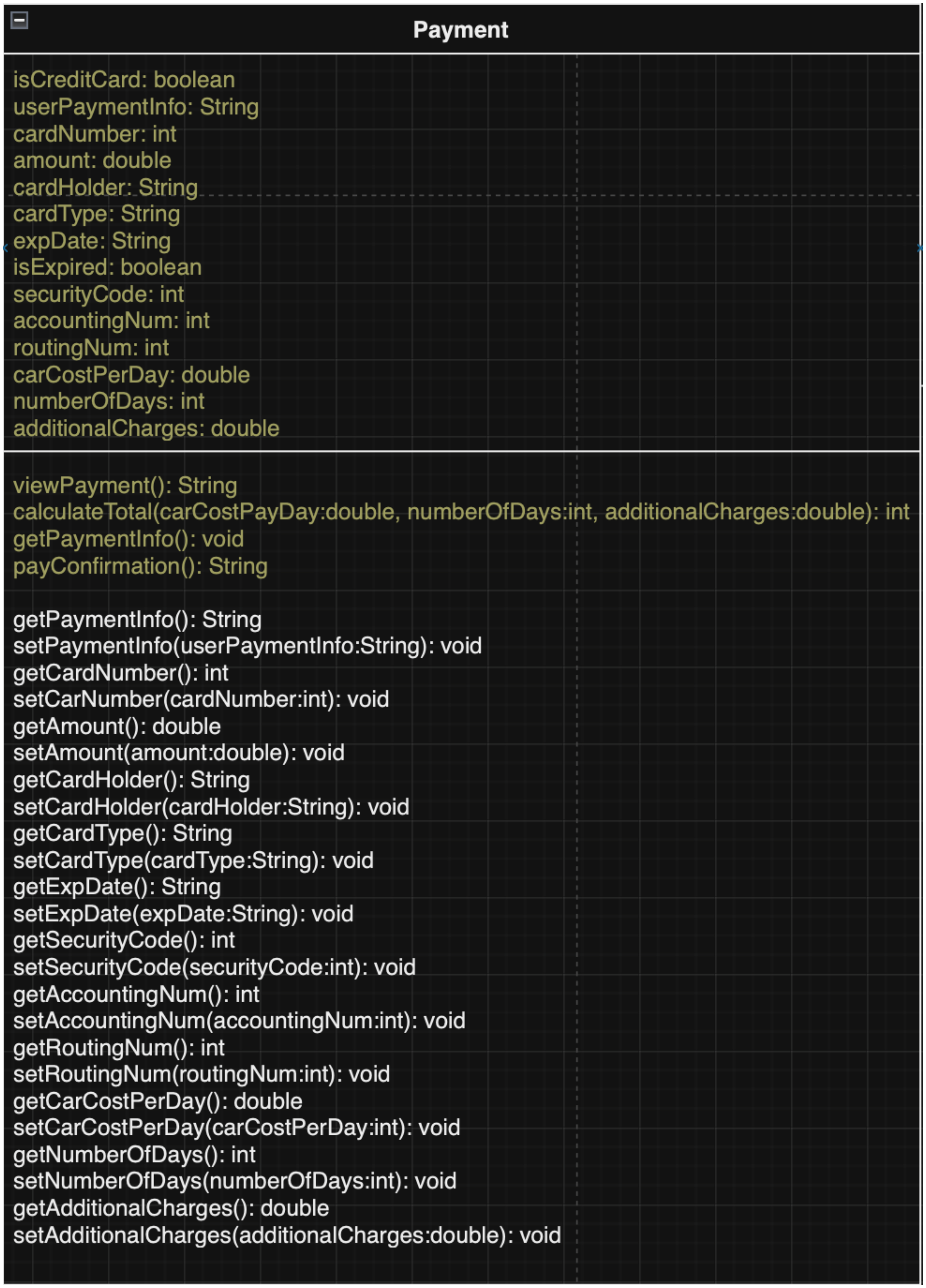
```
Car Rental Location
rentLocation: String
locationServices: boolean
city: String
state: String
zipCode: String
country: String

nearestLocation(): void
directionToNearest(requestRental:boolean): String
enableLocationServices(locationServices:boolean): boolean

getRentLocation(): String
setRentLocation(rentLocation:String): void
getCity(): String
setCity(city:String): void
getState(): String
setState(state:String): void
getZipCode(): String
setZipCode(zipCode:String): void
getCountry(): String
setCountry(country:String): void
```

```
System
agreementContracts: String
insurance: String
usernames: string[]
passwords: string[]
validInsurance: boolean
registration: string

findUser(usernames:String): String
findAgreementContracts(usernames:String): void
checkInsurance(validInsurance:boolean): boolean

getInsurance(): String
setInsurance(insurance:String): void
getAgreementContracts(): String
setAgreementContracts(agreementContracts:String): void
getUsernames(): String
setUsernames(username:String): void
getPasswords(): String
setPasswords(passwords:String): void
getRegistration(): String
setRegistration(registration:String): void
```

```
Car
available: boolean
carColor: String
carModel: String
carYear: int
carVIN: String
carPrice: double

updateCar(carModel:String): void
viewCar(carModel:String): String
deleteCar(carVIN:String): void
registarCar(carVIN:String): void
isAvailable(rentLocation:String): boolean
carModel(carVIN:string): void
carYear(carVIN:string): void
carInfo(carVIN:String, carModel:String, carYear:int, carColor:String, carPrice:double): String

getCarVIN(): String
setCarVIN(carVIN:String): void
getCarModel(): String
setCarModel(carModel:String): void
getCarYear(): int
setCarYear(carYear:int): void
getCarColor(): String
setCarColor(carColor:String): void
getCarPrice(): double
setCarPrice(carPrice:double): void
```



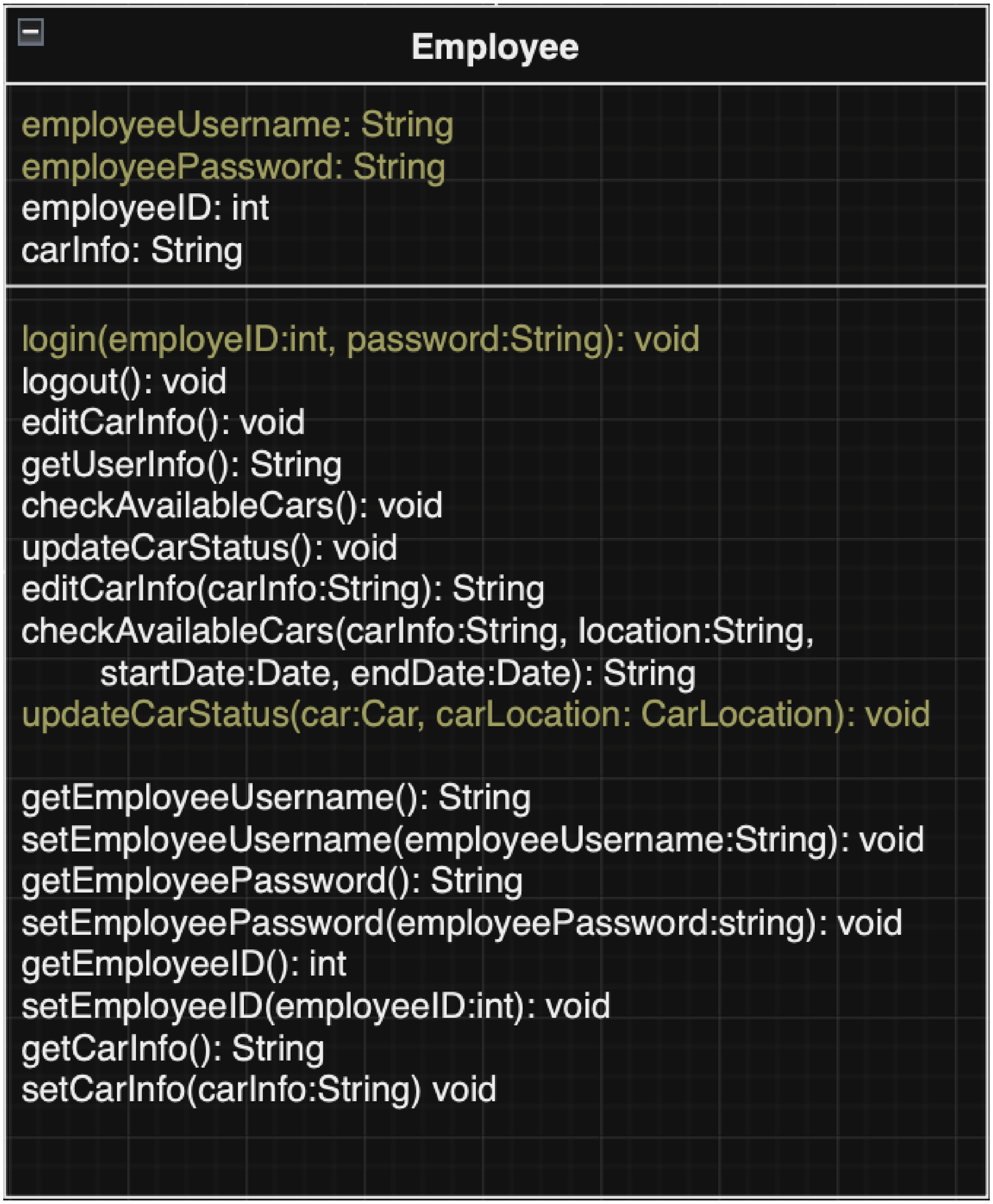
```
User

username: String
password: String
email: String
address: String
paymentInfo: String
firstName: String
lastName: String
dataOfBirth: String
haveAccount: boolean
driverLicense: String

checkUserInfo(username:String, password:String, email:String): String
lookUpLocations(rentLocation:String): String
logout(): void
login(username:String, password:String): void
sendEmailVerification(email:String): void
haveAccount(): boolean
makeAccount(username:String, email:String, password:String): void
linkPayment(username:String, paymentInfo:String): void
editProfile(username:String, address:String, password:String, email:String, driverLicens:String): void
makeOnlinePayment(paymentInfo:String): void
makeCreditCardPayment(paymentInfo:String): void
makeRental(car:Car, payment:Payment, carLocation:CarLocation, rental:Rental, system:System) void

getPassword(): String
setPassword(password:String): void
getUsername(): String
setUsername(username:String): void
getEmail(): String
setEmail(email:String): void
getAddress(): String
setAddress(address:String): void
getPayment(): String
setPayment(paymentInfo:String): void
getFirstName(): String
setFirstName(firstName:String): void
getLastName(): String
setLastName(lastName:String): void
getDateOfBirth(): String
setDateOfBirth(dateOfBirth:String): void
getDriverLicense(): String
setDriverLicense(driverLicense:String): void
```



Payment

```
isCreditCard: boolean
userPaymentInfo: String
cardNumber: int
amount: double
cardHolder: String
cardType: String
expDate: String
isExpired: boolean
securityCode: int
accountingNum: int
routingNum: int
carCostPerDay: double
numberOfDays: int
additionalCharges: double
```

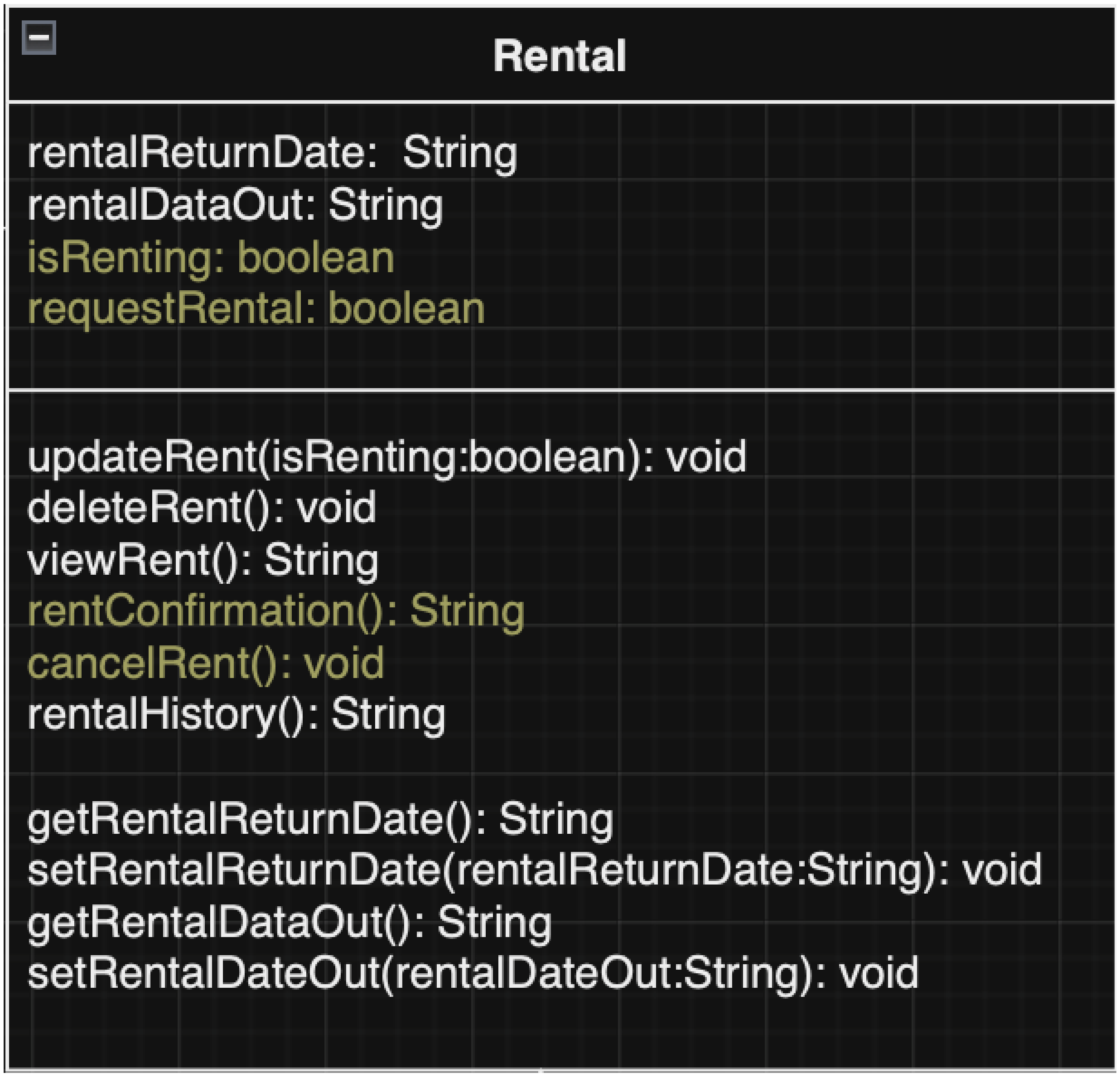
```
viewPayment(): String
calculateTotal(carCostPayDay:double, numberOfDays:int, additionalCharges:double): int
getPaymentInfo(): void
payConfirmation(): String

getPaymentInfo(): String
setPaymentInfo(userPaymentInfo:String): void
getCardNumber(): int
setCarNumber(cardNumber:int): void
getAmount(): double
setAmount(amount:double): void
getCardHolder(): String
setCardHolder(cardHolder:String): void
getCardType(): String
setCardType(cardType:String): void
getExpDate(): String
setExpDate(expDate:String): void
getSecurityCode(): int
setSecurityCode(securityCode:int): void
getAccountingNum(): int
setAccountingNum(accountingNum:int): void
getRoutingNum(): int
setRoutingNum(routingNum:int): void
getCarCostPerDay(): double
setCarCostPerDay(carCostPerDay:int): void
getNumberOfDays(): int
setNumberOfDays(numberOfDays:int): void
getAdditionalCharges(): double
setAdditionalCharges(additionalCharges:double): void
```



Employee

```
employeeUsername: String
employeePassword: String
employeeID: int
carInfo: String
```

```
login(employeID:int, password:String): void
logout(): void
editCarInfo(): void
getUserInfo(): String
checkAvailableCars(): void
updateCarStatus(): void
editCarInfo(carInfo:String): String
checkAvailableCars(carInfo:String, location:String,
        startDate:Date, endDate:Date): String
updateCarStatus(car:Car, carLocation: CarLocation): void

getEmployeeUsername(): String
setEmployeeUsername(employeeUsername:String): void
getEmployeePassword(): String
setEmployeePassword(employeePassword:string): void
getEmployeeID(): int
setEmployeeID(employeeID:int): void
getCarInfo(): String
setCarInfo(carInfo:String) void
```

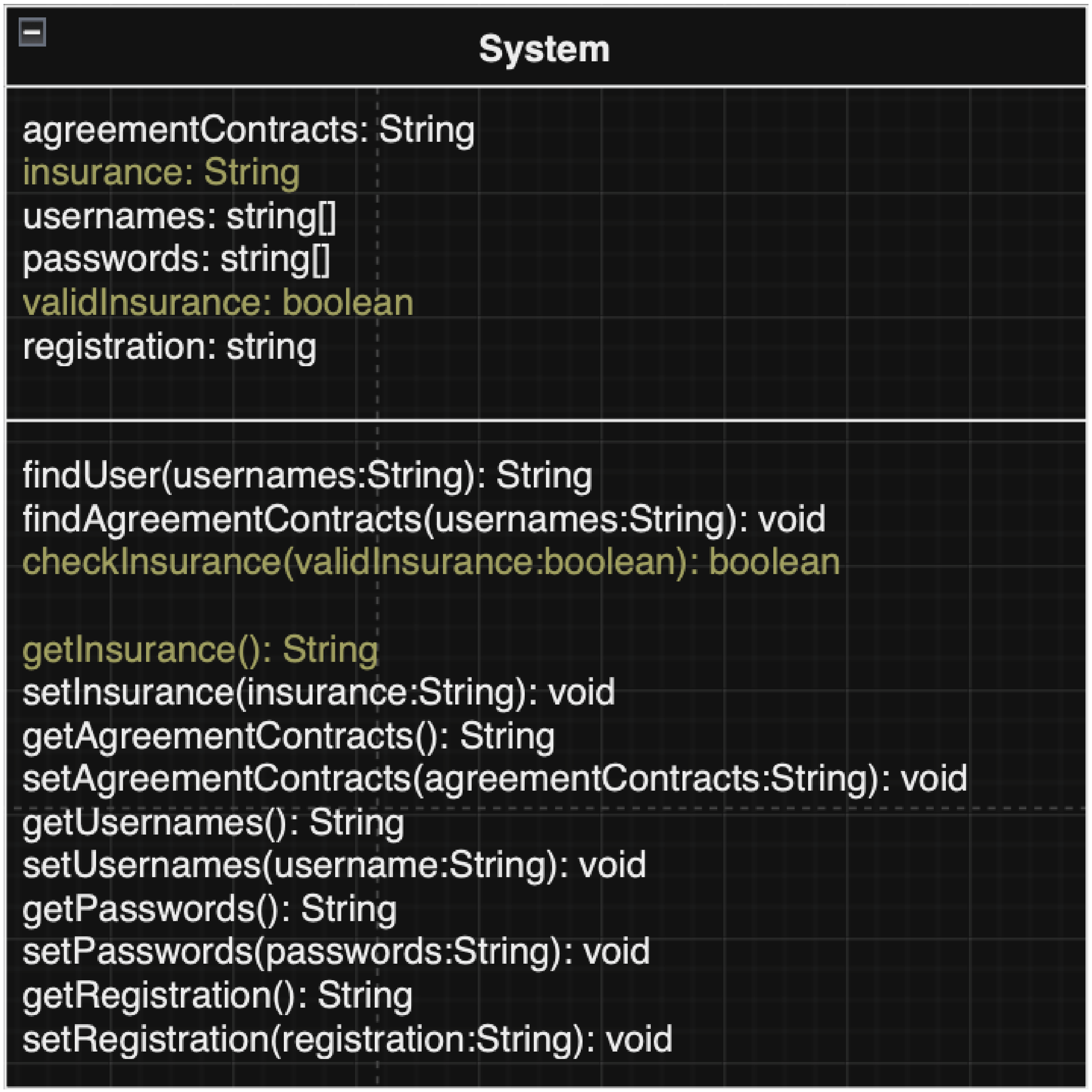


Rental

rentalReturnDate: String
rentalDataOut: String
isRenting: boolean
requestRental: boolean

updateRent(isRenting:boolean): void
deleteRent(): void
viewRent(): String
rentConfirmation(): String
cancelRent(): void
rentalHistory(): String

getRentalReturnDate(): String
setRentalReturnDate(rentalReturnDate:String): void
getRentalDataOut(): String
setRentalDateOut(rentalDateOut:String): void

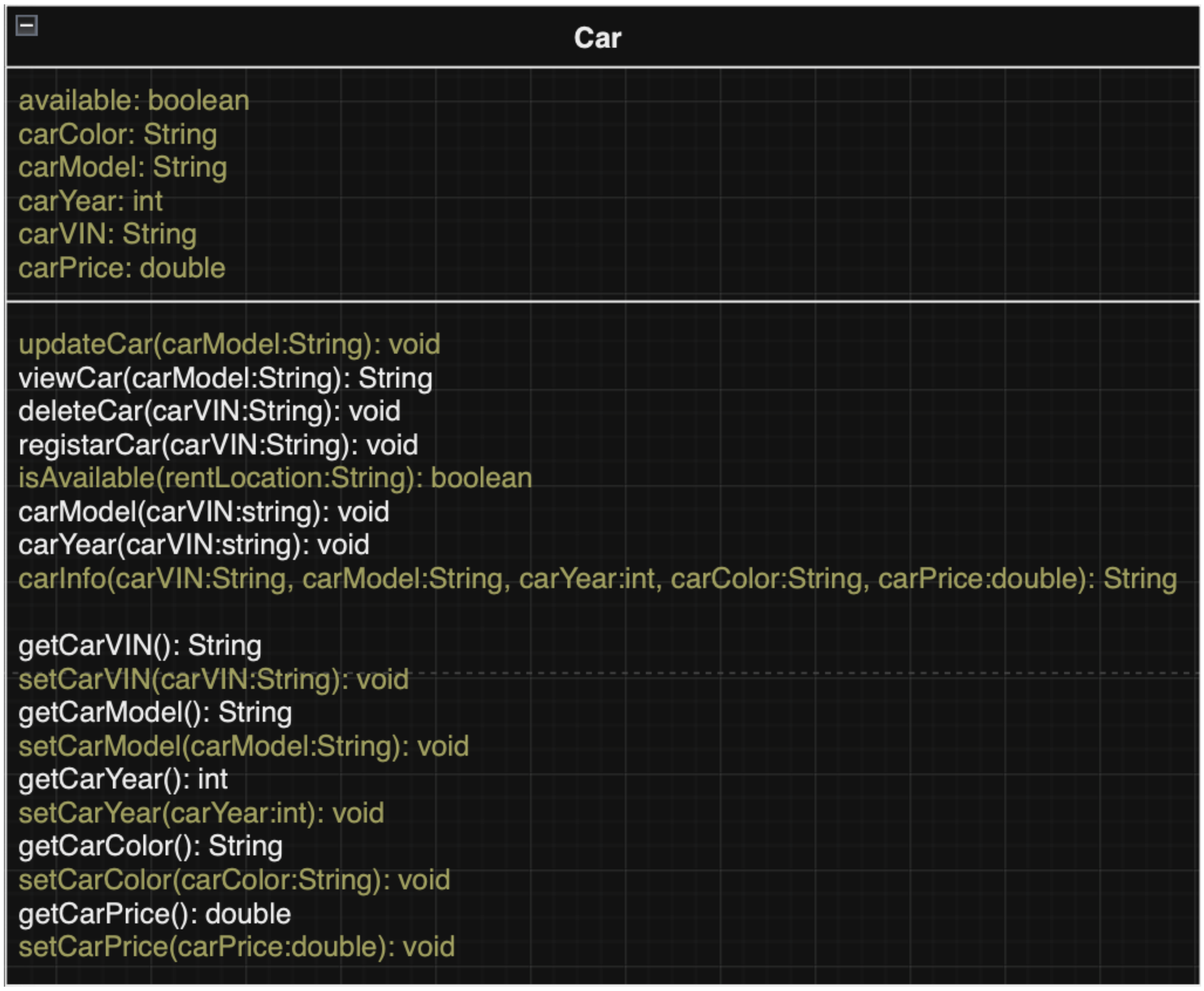


System

agreementContracts: String
insurance: String
usernames: string[]
passwords: string[]
validInsurance: boolean
registration: string

findUser(usernames:String): String
findAgreementContracts(usernames:String): void
checkInsurance(validInsurance:boolean): boolean

getInsurance(): String
setInsurance(insurance:String): void
getAgreementContracts(): String
setAgreementContracts(agreementContracts:String): void
getUsernames(): String
setUsernames(username:String): void
getPasswords(): String
setPasswords(passwords:String): void
getRegistration(): String
setRegistration(registration:String): void

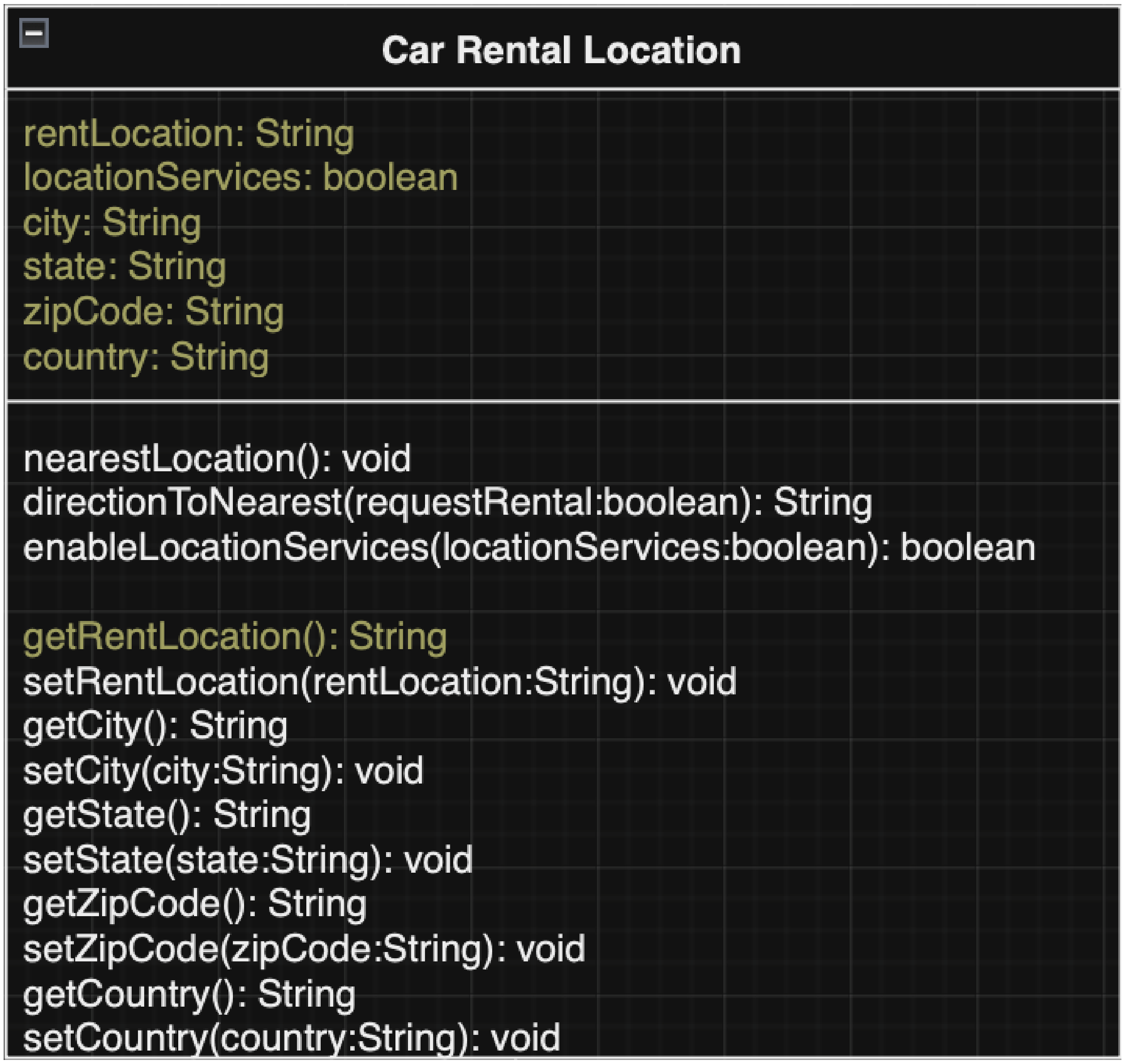


Car

available: boolean
carColor: String
carModel: String
carYear: int
carVIN: String
carPrice: double

updateCar(carModel:String): void
viewCar(carModel:String): String
deleteCar(carVIN:String): void
registarCar(carVIN:String): void
isAvailable(rentLocation:String): boolean
carModel(carVIN:string): void
carYear(carVIN:string): void
carInfo(carVIN:String, carModel:String, carYear:int, carColor:String, carPrice:double): String

getCarVIN(): String
setCarVIN(carVIN:String): void
getCarModel(): String
setCarModel(carModel:String): void
getCarYear(): int
setCarYear(carYear:int): void
getCarColor(): String
setCarColor(carColor:String): void
getCarPrice(): double
setCarPrice(carPrice:double): void



Car Rental Location

rentLocation: String
locationServices: boolean
city: String
state: String
zipCode: String
country: String

nearestLocation(): void
directionToNearest(requestRental:boolean): String
enableLocationServices(locationServices:boolean): boolean

getRentLocation(): String
setRentLocation(rentLocation:String): void
getCity(): String
setCity(city:String): void
getState(): String
setState(state:String): void
getZipCode(): String
setZipCode(zipCode:String): void
getCountry(): String
setCountry(country:String): void

4.2 Explanation of Modifications

Added `makeOnlinePayment(PaymentInfo: String): void` in User class - There was no option to make an online payment in the User class previously. This method is centered around payment through the online system.

Added `makeCreditCardPayment(PaymentInfo: String): void` in User class - There was no option to make a payment with a credit card in the User class previously despite having the attributes for credit card information like expDate, securityCode, etc. This method is centered around payment with a credit card.

Added `makeRental(car : Car, payment: Payment, carLocation: CarLocation, rental: Rental, system: System): void` in User class - Method was added because a make rental method did not exist in the User class. The method takes classes as a parameter which gives the method all the necessary attributes and methods it needs to complete the rental process from start to end.

Added `updateCarStatus(car : Car, carLocation: CarLocation): void` in Employee class - Method was added because the previous updateCarStatus method was not going to function correctly without the necessary parameters. For it to function correctly the method was going to need attributes like rentLocation() so that the car at the correct location can be modified, carModel so that the correct car can be modified, etc.

Deleted `makePayment(): void` in Payment class - The method was too broad and did not account for specific payments like payment with a credit card or online payment. It would also be difficult to integrate a single method that attempts to use multiple types of payments. A payment method was also unsuited in the payment class and better fitted in the User class, as the user is the one making a payment.

Deleted `makeRental(): void` in Rental class - The method did not take any parameters when it should have and was also better fitted in the User class rather than the Rental class.

Deleted `updateCarStatus(isavailable:boolean): void` in Employee class - The parameters of the method should also have included car and carLocation. IsAvaiable was also necessary, however, it would be easier to just take the car class as a parameter because there were also other necessary attributes and methods from that class.

5 Software Design Specification Test Plan

5.1 Test Plan

Unit Test

Authentication(login) of the user - `login(username:String, password:String): void`

Authentication(login) of the employee - `login(employeeID: int, password: string): void`

Setting the car information - `carInfo(carVIN:String, carModel:String, carYear:int, carColor:String, carPrice:double): String`

Integration Test

Payment through an online method like Paypal - `makeOnlinePayment(PaymentInfo: String)`

Payment method through Credit Card - `makeCreditCardPayment(PaymentInfo: String)`

Lookup rent locations - `lookUpLocations(rentLocation:String): String`

Checking if car is available in a specific location - `isAvailable(rentLocation: String): boolean`

System Test

User makes a car rental - `makeRental(car : Car, payment: Payment, carLocation: CarLocation, rental: Rental, system: System) : void`

Logging in to update car status for maintenance - `updateCarStatus(car : Car, carLocation: CarLocation, employee: Employee) : void`

Canceling car rent - `cancelRent(): void`

Unit Test

Authentication for User Login

Authentication(login) of the user - login(username:String, password:String): void

```
Test Case: Verify that a user can successfully log in with a valid
username and password.
Test Input: Valid username and password
Expected Output: Successful login
```

Test 1: Valid login

- Description: Verify that a user can successfully log in with a valid username and password.
- Test Input: Valid username and password
- Expected Output: Successful login, user session created
- Test Vector: Valid username and password
- Scope: This test validates the core functionality of user login, ensuring that valid credentials lead to a successful login and the creation of a user session.
- Failures Covered: This test covers the scenario where a user login should succeed.

```
Function TestValidUserLogin:
    username = "valid_username"
    password = "valid_password"
    result = Login(username, password)
    if result == "Successful login" and UserSessionExists():
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Test 2: Invalid username

- Description: Verify that login fails when an invalid username is provided.
- Test Input: Invalid username, valid password
- Expected Output: Login failed error
- Test Vector: Invalid username, but a valid password
- Scope: This test focuses on verifying that an invalid username results in a login failure and that no user session is created.
- Failures Covered: This test covers the scenario where the username is incorrect.

```
Function TestInvalidUsername:
    username = "invalid_username"
    password = "valid_password"
    result = Login(username, password)
    if result == "Login failed error" and not UserSessionExists():
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Test 3: Invalid password

- Description: Verify that login fails when an invalid password is provided.
- Test Input: Valid username, invalid password
- Expected Output: Login failed error
- Test Vector: Valid username, but an invalid password
- Scope: This test validates that an incorrect password leads to a login failure and ensures that no user session is created.
- Failures Covered: This test covers the scenario where the password is incorrect.

```
Function TestInvalidPassword:
    username = "valid_username"
    password = "invalid_password"
    result = Login(username, password)
    if result == "Login failed error" and not UserSessionExists():
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Authentication for Employee Login

Authentication(login) of the employee - login(employeeID: int, password: string): void

```
Test Case: Verify that an employee can successfully log in with a
valid employee ID and password.
Test Input: Valid employee ID and password
Expected Output: Successful login
```

Test 1: Valid employee login

- Description: Verify that an employee can successfully log in with a valid employee ID and password.
- Test Input: Valid employee ID and password
- Expected Output: Successful login, employee session created
- Test Vector: Valid employee ID and password
- Scope: This test validates the core functionality of employee login, ensuring that valid credentials lead to a successful login and the creation of an employee session.
- Failures Covered: This test covers the scenario where a valid employee login should succeed.

```
Function TestValidEmployeeLogin:
    employeeID = 12345
    password = "valid_password"
    result = EmployeeLogin(employeeID, password)
    if result == "Successful login" and EmployeeSessionExists():
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Test 2: Invalid employee ID

- Description: Verify that login fails when an invalid employee ID is provided.
- Test Input: Invalid employee ID, valid password
- Expected Output: Login failed error
- Test Vector: Invalid employee ID, but a valid password
- Scope: This test focuses on verifying that an invalid employee ID results in a login failure and that no employee session is created.
- Failures Covered: This test covers the scenario where the employee ID is incorrect.

```
Function TestInvalidEmployeeID:
    employeeID = 54321
    password = "valid_password"
    result = EmployeeLogin(employeeID, password)
    if result == "Login failed error" and not
EmployeeSessionExists():
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Test 3: Invalid employee password

- Description: Verify that login fails when an invalid employee password is provided.
- Test Input: Valid employee ID, invalid password
- Expected Output: Login failed error
- Test Vector: Valid employee ID, but an invalid password

- Scope: This test validates that an incorrect employee password leads to a login failure and ensures that no employee session is created.
- Failures Covered: This test covers the scenario where the employee password is incorrect.

```
Function TestInvalidEmployeePassword:
    employeeID = 12345
    password = "invalid_password"
    result = EmployeeLogin(employeeID, password)
    if result == "Login failed error" and not
EmployeeSessionExists():
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Setting the car information

Setting the car information - carInfo(carVIN:String, carModel:String, carYear:int, carColor:String, carPrice:double): String

```
Test Case: Verify that car information can be set successfully.
Test Input: Verify with valid car information including carVIN,
carModel, carYear, carColor, and carPrice.
Expected Output: "Car information set successfully"
```

Test 1: Valid car information

- Description: Verify that car information can be set successfully with valid input.
- Test Input: A complete set of valid car information (carVIN, carModel, carYear, carColor, carPrice).
- Expected Output: "Car information set successfully."
- Test Vector: Valid car VIN, model, year, color, and price
- Scope: This test ensures that the system can accept and store valid car information.
- Failures Covered: This test covers the scenario where car information is successfully set.

```
Function TestValidCarInfo:
    carVIN = "ABC123"
    carModel = "Sedan"
    carYear = 2022
    carColor = "Blue"
    carPrice = 25000.00

    result = SetCarInfo(carVIN, carModel, carYear, carColor,
carPrice)

    if result == "Car information set successfully":
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Test 2: Car VIN already exists

- Description: Verify that setting car information with an existing car VIN results in an error.
- Test Input: A car information set with a car VIN that already exists in the system, along with valid car model, year, color, and price.
- Expected Output: "Car VIN already exists."
- Test Vector: Existing car VIN, valid model, year, color, and price
- Scope: This test focuses on detecting the case where a duplicate car VIN is detected and handled correctly.
- Failures Covered: This test covers the scenario where a car VIN already exists.

```
Function TestDuplicateCarVIN:
    carVIN = "XYZ789" // Assuming this VIN already exists
    carModel = "SUV"
    carYear = 2023
    carColor = "Red"
    carPrice = 30000.00

    result = SetCarInfo(carVIN, carModel, carYear, carColor,
carPrice)

    if result == "Car VIN already exists":
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Test 3: Invalid car model and year

- Description: Verify that setting car information with an empty car model and a negative car year results in an error.

- Test Input: A car information set with a valid car VIN, an empty car model, a negative car year, along with valid car color and price.
- Expected Output: "Invalid car model and year."
- Test Vector: Valid car VIN, empty model, negative year, valid color, and price
- Scope: This test ensures that the system correctly identifies and handles invalid car model and year values.
- Failures Covered: This test covers the scenario where the car model and year are invalid.

```
Function TestInvalidCarModelAndYear:
   carVIN = "DEF456"
    carModel = "" // Empty model
    carYear = -1 // Negative year
    carColor = "Silver"
    carPrice = 28000.00

    result = SetCarInfo(carVIN, carModel, carYear, carColor,
carPrice)

    if result == "Invalid car model and year":
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Test 4: Invalid car color

- Description: Verify that setting car information with a car color containing special characters results in an error.
- Test Input: A car information set with a valid car VIN, car model, year, and car color that contains special characters, along with a valid car price.
- Expected Output: "Invalid car color."
- Test Vector: Valid car VIN, model, year, special character in color, valid price
- Scope: This test validates that the system can detect and handle car color values with special characters correctly.
- Failures Covered: This test covers the scenario where the car color contains special characters.

```
Function TestInvalidCarColor:
    carVIN = "LMN789"
    carModel = "Convertible"
    carYear = 2023
    carColor = "Black & White" // Color contains special characters
    carPrice = 35000.00

    result = SetCarInfo(carVIN, carModel, carYear, carColor,
carPrice)

    if result == "Invalid car color":
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Test 5: Invalid car price

- Description: Verify that setting car information with a car price of zero results in an error.
- Test Input: A car information set with a valid car VIN, car model, year, color, and a car price of zero.
- Expected Output: "Invalid car price."
- Test Vector: Valid car VIN, model, year, color, zero price
- Scope: This test ensures that the system correctly handles and detects invalid car prices, specifically a price of zero.
- Failures Covered: This test covers the scenario where the car price is invalid.

```
Function TestInvalidCarPrice:
    carVIN = "PQR123"
    carModel = "Hatchback"
    carYear = 2021
    carColor = "Green"
    carPrice = 0.00 // Zero price

    result = SetCarInfo(carVIN, carModel, carYear, carColor,
carPrice)

    if result == "Invalid car price":
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Integration Test

Payment through Online Method (e.g., PayPal)

Payment through an online method - makeOnlinePayment(PaymentInfo: String)

```
Test Case: Verify that the system can process an online payment
successfully.
Test Input: Payment information
Expected Output: Payment processed successfully
```

Test 1: Successful online payment

- Description: Validates that the system successfully processes an online payment, from user input to payment approval and transaction ID generation.
- Scope: This test ensures the core functionality of online payment processing.
- Failures Covered: Covers the scenario where online payment succeeds.
- Steps:
    - User enters payment details
    - Payment service requests online payment gateway
    - Gateway approves payment
    - Payment service returns transaction ID
- Expected Output: Payment successful

```
Function TestSuccessfulOnlinePayment:
    paymentInfo = "valid_payment_info"
    result = MakeOnlinePayment(paymentInfo)
    if result == "Payment successful":
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Test 2: Failed online payment

- Description: Tests the handling of a declined online payment, including error reporting.
- Scope: Ensures the system handles payment failures correctly.
- Failures Covered: Addresses the scenario where online payment fails.
- Steps:
    - User enters payment details
    - Payment service requests online payment gateway
    - Gateway declines payment

    - Payment service returns error
- Expected Output: Payment failed error

```
Function TestFailedOnlinePayment:
    paymentInfo = "invalid_payment_info"
    result = MakeOnlinePayment(paymentInfo)
    if result == "Payment failed error":
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Payment through Credit Card

Payment method through Credit Card - makeCreditCardPayment(PaymentInfo: String)

`Test Case: Verify that the system can process a credit card payment successfully.`
`Test Input: Payment information for a credit card`
`Expected Output: Payment processed successfully`

Test 1: Successful credit card payment

- Description: Validates the successful processing of a credit card payment, including interaction with the credit card processor and the return of a transaction ID.
- Scope: Ensures that credit card payments are processed correctly.
- Failures Covered: Covers the scenario where credit card payment succeeds.
- Steps:
    - User enters card details
    - Payment service contacts credit card processor
    - Processor approves payment
    - Payment service returns transaction ID
- Expected Output: Payment successful

```
Function TestSuccessfulCreditCardPayment:
    paymentInfo = "valid_credit_card_info"
    result = MakeCreditCardPayment(paymentInfo)
    if result == "Payment successful":
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Test 2: Declined credit card

- Description: Tests how the system handles a declined credit card payment, including error reporting.
- Scope: Verifies the correct handling of credit card payment failures.
- Failures Covered: Addresses the scenario where credit card payment fails.
- Steps:
    - User enters card details
    - Payment service contacts credit card processor
    - Processor declines payment
    - Payment service returns error
- Expected Output: Payment failed error

```
Function TestDeclinedCreditCardPayment:
    paymentInfo = "invalid_credit_card_info"
    result = MakeCreditCardPayment(paymentInfo)
    if result == "Payment failed error":
        Print("Test Passed")
    else:
        Print("Test Failed")
```

`Lookup Rental Locations`

`Lookup rent locations - lookUpLocations(rentLocation:String): String`

```
Test Case: Verify that the system can correctly look up rental
locations based on the provided location.
Test Input: Rent location name
Expected Output: List of rental locations matching the input
```

Test 1: Successful location lookup

- Description: Validates that the system can correctly look up rental locations, query location details, and display the information to the user.
- Scope: Ensures that location lookup functions as intended.
- Failures Covered: Addresses the scenario where location lookup is successful.
- Steps:
    - User enters desired rental location
    - User class calls lookUpLocations() with location
    - LookUpLocations() queries CarRentalLocation for info
    - CarRentalLocation returns location details
    - User class displays location info to user
- Expected output: Location information displayed

```
Function TestSuccessfulLocationLookup:
    rentLocation = "desired_location"
    result = LookUpLocations(rentLocation)
    if result == "Location information displayed":
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Test 2: Invalid location

- Description: Tests the system's response to an invalid or non-existent location and ensures the correct display of an error message.
- Scope: Verifies the handling of incorrect or non-existent location input.
- Failures Covered: Covers the scenario where the specified location is invalid or non-existent.
- Steps:
    - User enters invalid location

    - User class calls lookUpLocations() with invalid location
    - LookUpLocations() unable to find location
    - CarRentalLocation returns error
    - User class displays "Location not found" error
- Expected output: "Location not found" error displayed

```
Function TestInvalidLocationLookup:
    rentLocation = "invalid_location"
    result = LookUpLocations(rentLocation)
    if result == "Location not found error":
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Test 3: Location services disabled

- Description: Tests how the system handles location services being disabled on the user's device and the display of an appropriate error message.
- Scope: Validates the response to disabled location services.
- Failures Covered: Addresses the scenario where location services are disabled.
- Steps:
    - Location services disabled on device
    - User class calls lookUpLocations()
    - LookUpLocations() unable to determine location
    - CarRentalLocation returns error
    - User class displays "Enable location services" error
- Expected output: "Enable location services" error displayed

```
Function TestLocationServicesDisabled:
    DisableLocationServices()
    result = LookUpLocations()
    if result == "Enable location services error":
        Print("Test Passed")
    else:
        Print("Test Failed")
```

Checking Car Availability in a Specific Location

Checking if car is available in a specific location - `isAvailable(rentLocation: String): boolean`

Test Case: Verify that the system can check car availability in a specific location successfully.
Test Input: A location (rentLocation) for which car availability is being checked.
Expected Output: A boolean value indicating whether a car is available (true) or not available (false) at the specified location.

Test 1: Car Available in the Specified Location

- Description: Validates the correct identification of car availability and the return of 'true' when cars are available at the specified location.
- Scope: Ensures that car availability checks function as intended.
- Failures Covered: Covers the scenario where cars are available at the specified location.
- Steps:
    - User requests to check car availability at a specific location.
    - The system checks its database and confirms that there are cars available at the specified location.
    - The system returns a boolean value of 'true'.
- Expected Output: true (Car is available at the specified location).

```
Function TestCarAvailabilityInSpecifiedLocation:
    rentLocation = "Downtown"
    isCarAvailable = CheckCarAvailability(rentLocation)

    if isCarAvailable == true:
        Print("Test Passed: Car is available in the specified
location.")
    else:
        Print("Test Failed: Car is not available in the specified
location.")
```

Test 2: Car Not Available in the Specified Location

- Description: Ensures that the system correctly identifies the absence of cars and returns 'false' when no cars are available at the specified location.
- Scope: Validates the handling of scenarios where cars are not available.
- Failures Covered: Addresses the scenario where no cars are available at the specified location.

- Steps:
    - User requests to check car availability at a specific location.
    - The system checks its database and finds that there are no cars available at the specified location.
    - The system returns a boolean value of 'false'.
- Expected Output: false (Car is not available at the specified location).

```
Function TestCarNotAvailableInSpecifiedLocation:
    rentLocation = "Suburb"
    isCarAvailable = CheckCarAvailability(rentLocation)

    if isCarAvailable == false:
        Print("Test Passed: Car is not available in the specified
location.")
    else:
        Print("Test Failed: Car is available in the specified
location.")
```

Test 3: Invalid Location - Wrong Location Specified

- Description: Tests the system's response when an incorrect or non-existent location is provided for a car availability check and ensures the return of 'false.'
- Scope: Verifies the handling of incorrect or non-existent location input.
- Failures Covered: Covers the scenario where the specified location is incorrect or non-existent.
- Steps:
    - User requests to check car availability at a specific location.
    - The system checks its database and does not find any cars at the specified location as it is incorrect.
    - The system returns a boolean value of 'false'.
- Expected Output: false (Car is not available at the specified location due to an incorrect or non-existent location).

```
Function TestInvalidLocation:
    rentLocation = "Invalid Location"
    isCarAvailable = CheckCarAvailability(rentLocation)

    if isCarAvailable == false:
        Print("Test Passed: Car is not available due to an
incorrect or non-existent location.")
    else:
        Print("Test Failed: Car is available at the specified
location, which is incorrect.")
```

System Test

User Makes a Car Rental

User makes a car rental - makeRental(car : Car, payment: Payment, carLocation: CarLocation, rental: Rental, system: System)

Test Case: Verify the end-to-end process of a user making a car rental, including selecting a car, making a payment, specifying the car location, and updating the rental system.
Test Input: Car, payment details, car location, rental information, and the system components
Expected Output: Successful car rental with updated records

Test 1: Successful rental

- Description: Validates the end-to-end process of a user making a car rental, including car selection, payment, booking, and user confirmation.
- Scope: Ensures the entire car rental process functions correctly.
- Failures Covered: Covers scenarios where the rental is successful.
- Steps:
    - User selects car
    - Rental service checks availability
    - User makes payment
    - Rental service approves and books car
    - Car marked as unavailable
    - Confirmation sent to user
- Expected Output: Rental confirmed and booked

```
Function TestSuccessfulCarRental:
    user = GetUser()
    car = SelectCar(user)
    rentalService = GetRentalService()
    if rentalService.CheckAvailability(car) == "Available":
        payment = MakePayment(user, car)
        if payment == "Payment successful":
            rentalService.BookCar(car)
            car.MarkAsUnavailable()
            NotifyUser("Rental confirmed and booked")
        else:
            NotifyUser("Payment failed error")
    else:
        NotifyUser("Rental failed error")
```

Test 2: Rental failure due to unavailable car

- Description: Tests the system's response when the selected car is unavailable for rental.
- Scope: Ensures the system handles scenarios where cars are unavailable.
- Failures Covered: Covers scenarios where the selected car is unavailable.
- Steps:
    - User selects car
    - Rental service checks availability
    - Car is unavailable
    - Rental service notifies user
- Expected Output: Rental failed error

```
Function TestUnavailableCarRental:
    user = GetUser()
    car = SelectCar(user)
    rentalService = GetRentalService()

    if rentalService.CheckAvailability(car) == "Unavailable":
        NotifyUser("Rental failed error")
    else:
        payment = MakePayment(user, car)
        if payment == "Payment successful":
            rentalService.BookCar(car)
            car.MarkAsUnavailable()
            NotifyUser("Rental confirmed and booked")
        else:
            NotifyUser("Payment failed error")
```

Logging in to Update Car Status for Maintenance

Logging in to update car status for maintenance - updateCarStatus(car : Car, carLocation: CarLocation, employee: Employee).

Test Case: Verify that an employee can log in successfully and update the status of a car for maintenance.
Test Input: Car, car location, and employee login
Expected Output: Car status updated for maintenance

Test 1: Successful car status update

- Description: Validates that an employee can log in, update the status of a car for maintenance, and receive a successful confirmation.
- Scope: Ensures that the process of updating car status for maintenance is functioning as intended.
- Failures Covered: Covers scenarios where the car status is successfully updated.
- Steps:
    - Employee logs in.
    - Employee searches for a car.
    - Employee changes status to unavailable.
    - Car status updated
- Expected Output: Car status update successful

```
Function TestSuccessfulCarStatusUpdate:
    employee = GetEmployee()
    car = SelectCarForMaintenance(employee)

    if car != null:
        employeeLogInResult = EmployeeLogIn(employee)
        if employeeLogInResult == "Successful login":
            car.MarkAsUnavailableForMaintenance()
            NotifyEmployee("Car status update successful")
        else:
            NotifyEmployee("Car status update failed")
    else:
        NotifyEmployee("Car not found for maintenance")
```

Test 2: Car status update failure due to invalid login

- Description: Tests the system's response when an employee attempts to update car status with invalid login credentials.
- Scope: Verifies the handling of login failures during car status updates.
- Failures Covered: Covers scenarios where the employee's login is invalid.
- Steps:
    - Invalid employee credentials entered
    - Login fails
    - Car status not updated
- Expected Output: Car status update failed

```
Function TestInvalidLoginCarStatusUpdate:
    employee = GetEmployee()
    car = SelectCarForMaintenance(employee)

  if car != null:
        employeeLogInResult =
EmployeeLogInWithInvalidCredentials(employee)
        if employeeLogInResult == "Login failed error":
            NotifyEmployee("Car status update failed")
        else:
            NotifyEmployee("Car status update successful")
    else:
        NotifyEmployee("Car not found for maintenance")
```

Canceling a Car Rental

Canceling car rent - cancelRent(): void

Test Case: Verify the end-to-end process of a user canceling a car rental, including initiating the cancellation, updating the car status, handling refunds (if applicable), and notifying the user.
Test Input: Rental details, user request to cancel the rental, car status, payment information, and system components.
Expected Output: Successful rental cancellation with updated records and user notifications.

Test 1: Successful rental cancellation

- Description: Validates the end-to-end process of a user canceling a car rental, including initiating the cancellation, updating the car status, handling refunds (if applicable), and notifying the user.
- Scope: Ensures the entire rental cancellation process functions correctly.
- Failures Covered: Covers scenarios where the rental is successfully canceled.
- Steps:

    - User initiates the cancellation of a rental.
    - The system retrieves the rental details associated with the user's request.
    - The system checks if the rental status is "booked" or "in-progress" and verifies that the user is the one who made the reservation.
    - If the conditions are met, the system proceeds with the cancellation; otherwise, it notifies the user that the cancellation is not possible.
    - The system updates the rental status to "canceled."
    - The system releases the car and changes its status to "available" in the car inventory.
    - The system calculates any refund due to the user based on the cancellation policy.
    - If a refund is applicable, the system processes the refund and updates the user's payment information.
    - The system sends a confirmation to the user, indicating that the rental has been canceled successfully.
    - The user receives the confirmation and checks the refund amount (if applicable) in their payment account.
    - The user confirms that the car is available again for booking by checking the car inventory.
    - The user can make a new reservation for the car if desired.
- Expected Outcome:

    - The rental is successfully canceled.
    - The car's status is updated to "available" in the inventory.
    - If applicable, the user receives a refund and is notified of the cancellation.
    - The user can confirm that the car is available for booking again in the system.

```
Function TestSuccessfulRentalCancellation:
    userRequest = UserInitiatesCancellation()
    rentalDetails = GetRentalDetails(userRequest)

    if rentalDetails.status == "booked" or rentalDetails.status ==
"in-progress" and rentalDetails.user == userRequest:
        rentalDetails.status = "canceled"
        car = rentalDetails.car
        car.status = "available"

        if rentalDetails.refundPolicy != null:
            refundAmount = CalculateRefund(rentalDetails)
            ProcessRefund(userRequest, refundAmount)

        SendConfirmation(userRequest, "Rental canceled
successfully.")
        Print("Test Passed: Rental canceled successfully.")
    else:
        SendNotification(userRequest, "Rental cancellation not
possible.")
        Print("Test Failed: Rental cancellation not possible.")
```

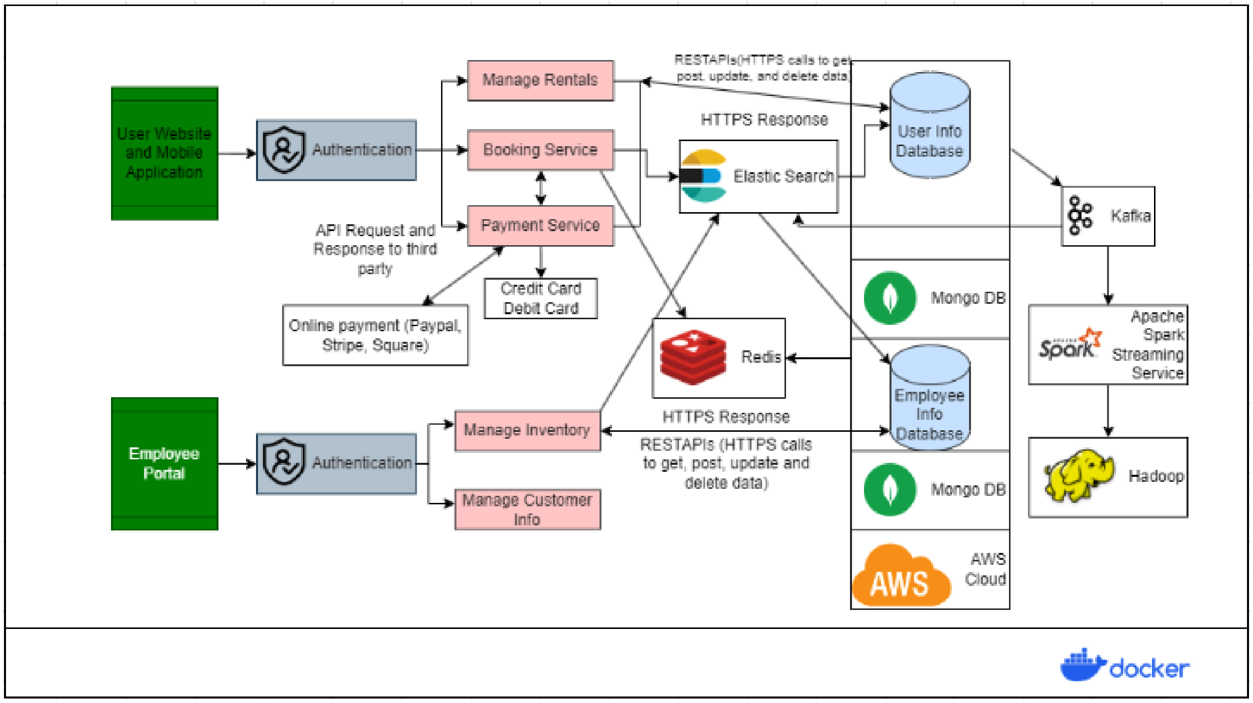
Test 2: Rental cancellation not possible (e.g., due to status)

- Description: Tests the system's response when a user attempts to cancel a rental that has already concluded.
- Scope: Verifies that the system correctly identifies when a rental cannot be canceled.
- Failures Covered: Covers scenarios where rental cancellation is not possible due to the rental status.
- Steps:
    - User initiates the cancellation of a rental.
    - The system retrieves the rental details associated with the user's request.
    - The system checks that the rental status is "completed," indicating that the rental has already concluded.
    - The system notifies the user that the rental cannot be canceled at this stage.
- Expected Outcome: The user is notified that the rental cannot be canceled because it has already been completed.

```
Function TestRentalCancellationNotPossible:
    userRequest = UserInitiatesCancellation()
    rentalDetails = GetRentalDetails(userRequest)

    if rentalDetails.status == "completed":
        SendNotification(userRequest, "Rental cancellation not
possible.")
        Print("Test Passed: Rental cancellation not possible.")
    else:
        Print("Test Failed: Rental should have been canceled.")
```

6 Software Design 2.0

6.1 Updated Software Architecture Diagram



The main components of the system are:

- User Website and Mobile Application: This is the frontend that customers would use to view rental cars, make bookings, manage their account, etc. It would be a responsive website as well as iOS and Android apps.

- User Info Database: Stores information about users like name, contact details, login credentials, etc. This is a MongoDB database.

- Employee Info Database: Stores information about employees like name, roles, contact details, etc. This is also a MongoDB database.

- Employee Portal: An internal web portal used by employees to manage rentals, inventory, customer information, etc.

- Authentication Service: Handles user authentication and authorization. Uses Redis for caching user sessions.

- Payment Service: Handles payments from users. Integrates with payment gateways like PayPal, Stripe, Square, etc., and also has a credit/debit card option.

- Booking Service: Handles rental bookings and scheduling. Uses Kafka for streaming booking data.

- Manage Rentals: Business logic to handle rental workflow like checking in/out cars, assigning cars to bookings, calculating rental costs, etc.

- Manage Inventory: Business logic to track cars available for renting, occupied cars, cars due for maintenance, etc. Uses Elasticsearch for searching/indexing inventory.

- Manage Customer Info: Business logic to handle customer accounts, loyalty programs, promotions, etc.

- Hadoop Cluster: Used for storing and analyzing large amounts of rental and customer data to generate business reports.

The different services communicate via REST APIs. The frontend and backend are decoupled. The backend services can scale independently. AWS cloud is used for hosting the infrastructure. Docker containers are used for easy deployment and scaling of services. Key databases like MongoDB, Elasticsearch, and Redis provide persistence, search, and caching capabilities. The overall architecture allows BeAvis to handle user traffic, bookings, and billing in a robust and scalable manner.

Frontend Components:
- User Interface: The system provides a responsive website and mobile applications for users, ensuring accessibility across various devices.
- Decoupled Frontend and Backend: Decoupling the frontend and backend allows for independent scaling and development, promoting flexibility and maintainability.

Databases:
- User Info Database (MongoDB): MongoDB is chosen for its flexibility in handling semi-structured data, suitable for user-related information.
- Employee Info Database (MongoDB): MongoDB is used for storing employee information, maintaining consistency in the choice of database technology.

Internal Employee Portal:

- An internal web portal for employees enhances operational efficiency by providing a centralized platform for managing rentals, inventory, and customer information.

Authentication Service:
- The use of Redis for caching user sessions in the Authentication Service contributes to efficient session management and faster authentication.

Payment Service:
- Integration with external payment gateways like PayPal, Stripe, and Square allows users to choose their preferred payment method.
- Direct integration with payment processors for credit/debit card payments ensures secure and streamlined transactions.

Booking Service:
- Kafka is employed for streaming booking data, providing a scalable and real-time solution for handling booking events.
- REST APIs facilitate communication between the frontend and the Booking Service.

Business Logic Components:
- Dedicated services for managing rentals, inventory, and customer information showcase a modular and organized approach to business logic.
- Elasticsearch is utilized for efficient searching and indexing of inventory.

Hadoop Cluster:
- The use of a Hadoop cluster for storing and analyzing large amounts of rental and customer data reflects a commitment to data-driven decision-making and reporting.

Communication and Scalability:
- REST APIs serve as the communication bridge between different components, promoting interoperability.
- Docker containers enhance deployment and scaling flexibility, contributing to a more robust and scalable system.
- The ability to scale backend services independently allows for optimized resource allocation based on demand.

Cloud Infrastructure:
- Leveraging AWS for hosting provides a reliable and scalable cloud infrastructure.

- Docker containers and cloud infrastructure together contribute to a more efficient and scalable deployment model.

The decoupled nature of the frontend and backend, along with the emphasis on scalability and flexibility, positions the system to handle user traffic, bookings, and billing effectively.

The key components in the architecture diagram interact with each other via the various connectors:

- Users interact with the system via the Website and Mobile Apps. These frontend components connect to the backend services via REST APIs.

- The REST APIs connect to the Authentication Service to verify user identities and authorize access. The Authentication Service uses Redis to cache user sessions. It writes session data to Redis and reads it when needed for request authentication.

- For rental bookings, the frontend connects to the Booking Service API. The Booking Service uses Kafka to stream booking data to other services.

- To check inventory availability for a booking, the Booking Service interacts with the Manage Inventory service using REST APIs.

- For making payments, the frontend connects to the Payment Service API. The Payment Service interacts with external API gateways for online payment like PayPal. For credit/debit card payments, the Payment Service integrates directly with payment processors like Stripe to charge the cards.

- To record rental transactions, the Booking Service sends data to the Manage Rentals service using REST APIs. Manage Rentals records transactions in the MongoDB database.

- For customer profiles and loyalty programs, the frontend uses the Manage Customer Info service APIs. Customer data is stored in MongoDB.

- Manage Inventory uses Elasticsearch to index and search for available inventory for rentals.

- Hadoop cluster runs analytics on historical data from MongoDB, Elasticsearch etc. to generate reports and insights.

- The Employee Portal uses REST APIs to connect to services like Manage Bookings, Manage Rentals, Manage Inventory etc. to support employee workflows.

- All services use Docker containers for deployment. The containers are hosted on AWS cloud infrastructure.

In summary, the components interact via REST APIs, a Kafka message bus, database connections, and calls to external payment gateways in a decoupled way to facilitate the rental system functions.

6.2 Data Management Strategy

Non-SQL (Document-Based - MongoDB)

1. Design Decisions:

a. Choice of Database Technology:
- MongoDB was selected as the database technology for this system due to its flexibility, scalability, and ability to handle diverse and evolving data schemas. As a document database, MongoDB stores data in JSON-like documents that can vary in structure, allowing heterogeneous data to be stored without restricting documents to a predefined schema. This schema-less design accommodates fluid data models and frequent iterations as product requirements evolve.

MongoDB is chosen due to the variable nature of car rental data. Different makes/models of cars have different attributes that need to be tracked. MongoDB's flexible, schema-less documents can easily accommodate this heterogeneous data.

In addition, MongoDB provides horizontal scalability and native sharding capabilities to support massive datasets and high throughput. Its distributed architecture can scale out across commodity servers, providing high availability and fault tolerance. These factors make MongoDB well-suited for rapidly growing applications that require flexibility and scalability.

b. Number of Databases:
- The initial implementation will utilize a single MongoDB database, with separate collections to store different entity types or domains. This approach reduces the overhead associated with managing multiple databases and simplifies application

code during early development. However, horizontal scaling can be achieved by sharding collections as the system grows.

As the system grows, databases can be split logically based on access patterns, security domains, or other criteria. For example, a separate reporting database could be created to isolate analytics workloads from operational systems. Sharding can also distribute collections across multiple physical databases as dataset size and throughput requirements increase.

Initially, a single MongoDB database will be used called 'rental-database'. Separate databases may be warranted in the future to isolate analytical workloads from operational systems.

2. Logical Data Split:

a. Collections:
- Data is logically split into collections based on the nature of entities. For example:
- User Collection: Stores user information, including username, password, email, address, payment information, first name, last name, date of birth, and driver's license.
- Payment Collection: Stores customer payment information such as card number, amount in the card, holder of the card, card type, expiration date, security code, accounting number, and routing information. Also stores information regarding the rental amount such as additional payments, cost of renting per day, and number of days rented.
- Employee Collection: Contains car info, employee username, password, and ID
- Rental Collection: Contains rental information regarding the return date, date out, and rented cars.
- System Collection: Stores important company information, including customer agreement contracts, insurance, username, password, and car registration.
- Car Collection: Manages dynamic inventory. Stores information about available vehicles and their specifications like car color, model, year, VIN, and price.
- Car Rental Location Collection: Stores information regarding car rental locations such as their city, state, zip code, and country.

3. Possible Alternatives:

a. Alternative Technologies:
- Relational SQL databases like MySQL provide stricter data governance through schema enforcement, foreign key constraints, and ACID transactions. This improves data integrity but reduces flexibility.

Other NoSQL databases like Cassandra and CouchDB have different architectures optimized for specific use cases. Cassandra excels at high throughput for time-series data, while CouchDB focuses on ease of syncing across devices, but MongoDB's document-oriented model is chosen for its simplicity and ease of use.

b. Data Organization:
- A key-value NoSQL database could store related entities in a single table, denormalizing for fast simple lookups. However, this can complicate querying and indexing beyond simple key retrieval.

A graph database like Neo4j could explicitly model relationships between entities for data-intensive connectivity analysis. But graph traversals may be overkill for this application.

MongoDB provides a balance between performance and query flexibility.

4. Tradeoffs:

a. Flexibility vs. Structure:
- MongoDB sacrifices some of the rigid structure enforced by traditional relational databases in favor of greater flexibility. This flexibility can be advantageous when dealing with evolving data schemas.
- MongoDB's design allows for easy adaptation to changes in car attributes, but it comes with the tradeoff of less stringent enforcement of schemas and constraints compared to traditional databases.

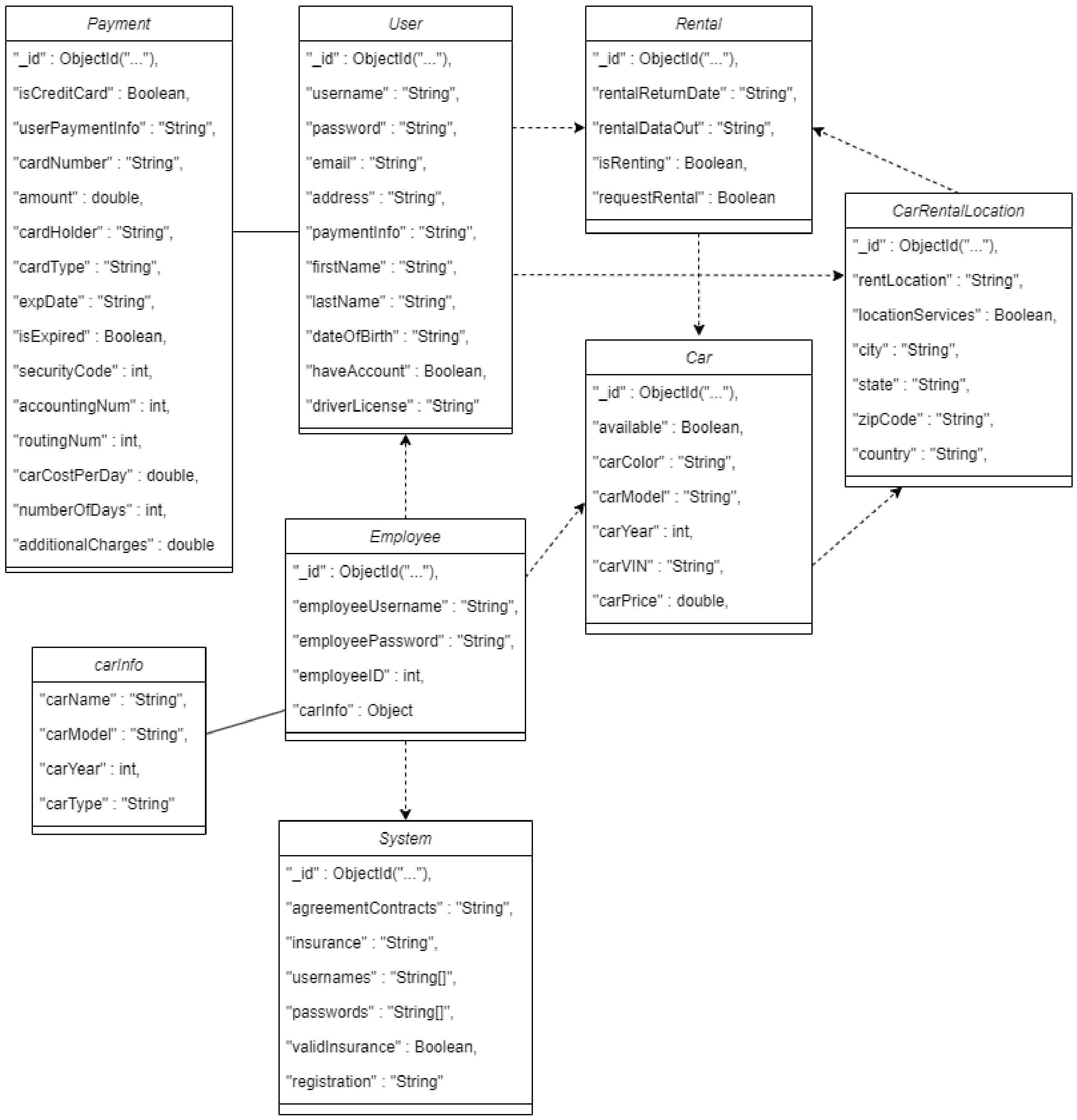
b. Consistency vs. Scalability:
- MongoDB provides eventual consistency by default, making it suitable for distributed and scalable systems. However, in scenarios where immediate consistency is critical, a traditional SQL database might be more appropriate.
- MongoDB's inherent support for horizontal scaling on commodity hardware is beneficial for handling large volumes of rental data, contributing to its scalability in comparison to SQL databases.

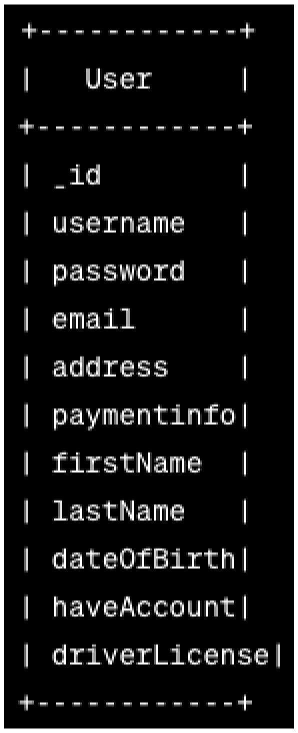
c. Query Complexity:
- While MongoDB supports rich query capabilities, complex queries involving joins across multiple collections can be more challenging compared to traditional SQL databases.
- While MongoDB has some limitations on complex queries, it offers simpler mappings to code objects, enhancing developer productivity and ease of use.

In summary, MongoDB's design choices prioritize flexibility and scalability, making it well-suited for dynamic and distributed environments. However, these advantages come with tradeoffs, including a more permissive approach to data structures and potential complexities in certain types of queries compared to traditional SQL databases.

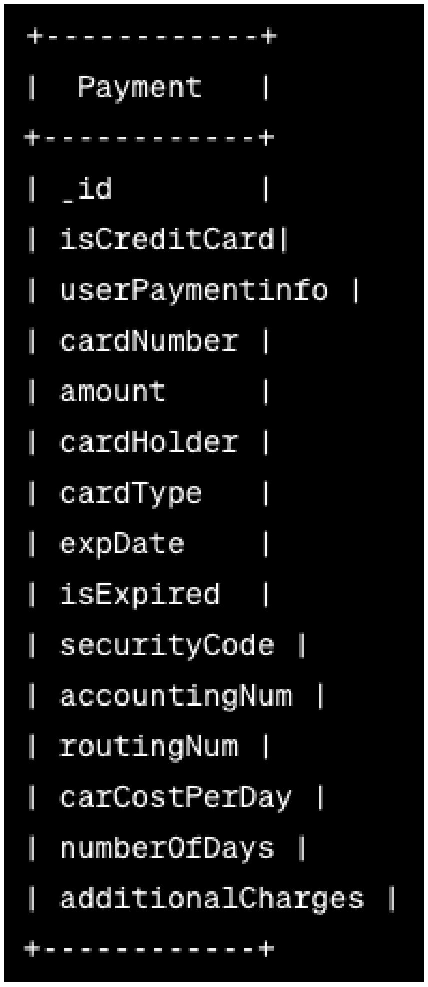
Diagram:

MongoDB Schema Diagram

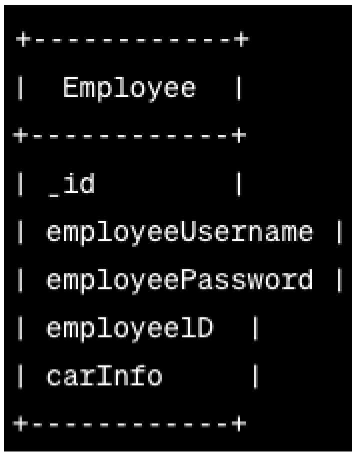




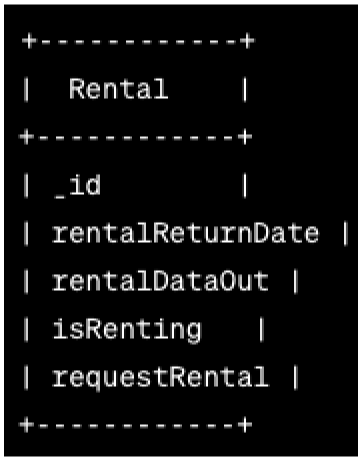
```
+------------+
|   User     |
+------------+
| _id        |
| username   |
| password   |
| email      |
| address    |
| paymentinfo|
| firstName  |
| lastName   |
| dateOfBirth|
| haveAccount|
| driverLicense|
+------------+
```



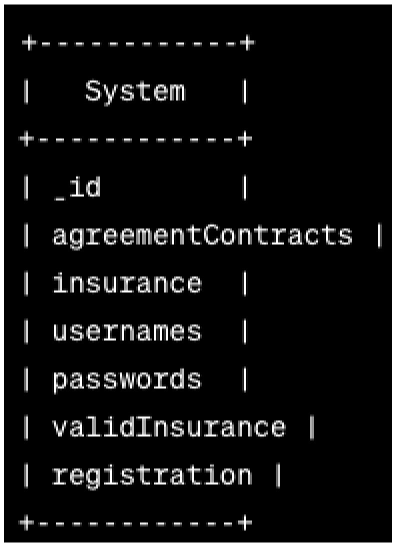
```
+------------+
|  Payment   |
+------------+
| _id        |
| isCreditCard|
| userPaymentinfo |
| cardNumber |
| amount     |
| cardHolder |
| cardType   |
| expDate    |
| isExpired  |
| securityCode |
| accountingNum |
| routingNum |
| carCostPerDay |
| numberOfDays |
| additionalCharges |
+------------+
```



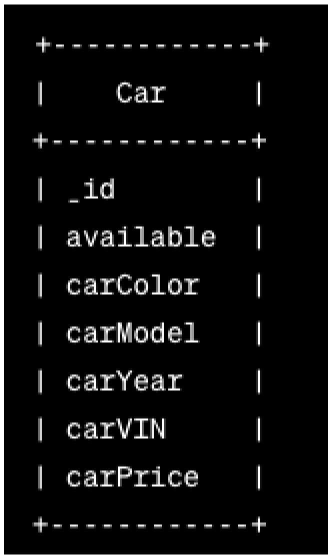
```
+------------+
|  Employee  |
+------------+
| _id        |
| employeeUsername |
| employeePassword |
| employeeID  |
| carInfo     |
+------------+
```



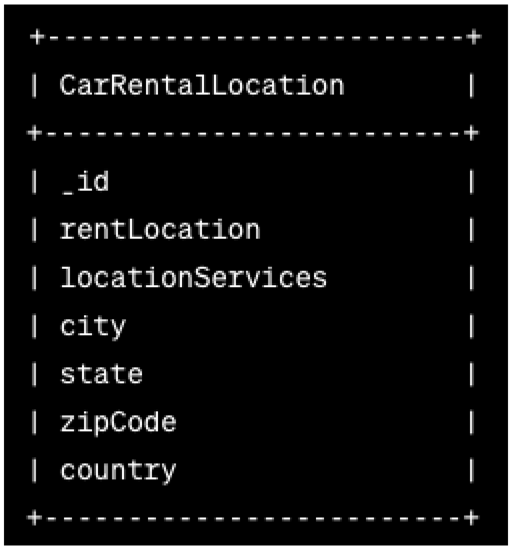
```
+-----------+
|  Rental    |
+-----------+
| _id         |
| rentalReturnDate |
| rentalDataOut |
| isRenting   |
| requestRental |
+-----------+
```



```
+-----------+
|   System   |
+-----------+
| _id         |
| agreementContracts |
| insurance   |
| usernames   |
| passwords   |
| validInsurance |
| registration |
+-----------+
```



```
+-----------+
|    Car     |
+-----------+
| _id         |
| available   |
| carColor    |
| carModel    |
| carYear     |
| carVIN      |
| carPrice    |
+-----------+
```



```
+----------------------+
| CarRentalLocation      |
+----------------------+
| _id                    |
| rentLocation           |
| locationServices       |
| city                   |
| state                  |
| zipCode                |
| country                |
+----------------------+
```

User Collection:

- _id: The unique identifier for each user.
- username: The username chosen by the user.
- password: The hashed password for user authentication.
- email: User's email address.
- address: User's physical address.
- paymentinfo: Information related to the user's payment, potentially referencing a Payment document.
- firstName: User's first name.
- lastName: User's last name.
- dateOfBirth: User's date of birth.

- haveAccount: A boolean indicating whether the user has an account.
- driverLicense: User's driver's license information.

Payment Collection:

- _id: The unique identifier for each payment transaction.
- isCreditCard: A boolean indicating whether the payment method is a credit card.
- userPaymentinfo: Information related to the user's payment.
- cardNumber: Credit card number.
- amount: Payment amount.
- cardHolder: Name of the cardholder.
- cardType: Type of the credit card.
- expDate: Expiry date of the credit card.
- isExpired: A boolean indicating whether the credit card is expired.
- securityCode: Security code of the credit card.
- accountingNum: Accounting number for payment.
- routingNum: Routing number for payment.
- carCostPerDay: Cost per day for car rental associated with the payment.
- numberOfDays: Number of days for which the car is rented.
- additionalCharges: Additional charges associated with the payment.

Employee Collection:

- _id: The unique identifier for each employee.
- employeeUsername: The username of the employee.
- employeePassword: The hashed password for employee authentication.
- employeeID: The unique identifier for each employee.
- carInfo: Information about the car assigned to the employee, including car name, model, year, and type.

Rental Collection:

- _id: The unique identifier for each rental transaction.
- rentalReturnDate: The anticipated return date for the rental.
- rentalDataOut: The date when the rental was initiated.
- isRenting: A boolean indicating whether the car is currently being rented.
- requestRental: A boolean indicating whether a rental has been requested.

System Collection:

- _id: The unique identifier for each system record.

- agreementContracts: Information about the agreement contracts.
- insurance: Information about the insurance coverage.
- usernames: An array of usernames associated with the system.
- passwords: An array of passwords associated with the system.
- validInsurance: A boolean indicating whether the insurance is valid.
- registration: Information related to the system registration.

Car Collection:

- _id: The unique identifier for each car.
- available: A boolean indicating whether the car is available for rental.
- carColor: The color of the car.
- carModel: The model name of the car.
- carYear: The manufacturing year of the car.
- carVIN: The Vehicle Identification Number of the car.
- carPrice: The price of the car.

CarRentalLocation Collection:

- _id: The unique identifier for each car rental location.
- rentLocation: The name of the rental location.
- locationServices: A boolean indicating whether location services are available.
- city: The city where the rental location is situated.
- state: The state where the rental location is situated.
- zipCode: The ZIP code of the rental location.
- country: The country where the rental location is situated.

Each collection represents a specific entity in the application, and the fields within each document provide detailed information about that entity. These collections collectively define the structure and relationships within our MongoDB database for a car rental system.

MongoDB JSON Objects:

```
// MongoDB Collection: User
[
  {
    "_id": ObjectId("..."),
    "username": "john_doe",
    "password": "hashed_password_1",
    "email": "john.doe@example.com",
    "address": "123 Main St, City",
    "paymentInfo": "1234-5678-9012-3456",
    "firstName": "John",
    "lastName": "Doe",
    "dateOfBirth": "1990-01-01",
    "haveAccount": true,
    "driverLicense": "ABC123XYZ"
  },
  {
    "_id": ObjectId("..."),
    "username": "alice_smith",
    "password": "hashed_password_2",
    "email": "alice.smith@example.com",
    "address": "456 Oak St, Town",
    "paymentInfo": "9876-5432-1098-7654",
    "firstName": "Alice",
    "lastName": "Smith",
    "dateOfBirth": "1985-05-15",
    "haveAccount": true,
    "driverLicense": "XYZ789ABC"
  },
  {
    "_id": ObjectId("..."),
    "username": "bob_jones",
    "password": "hashed_password_3",
    "email": "bob.jones@example.com",
    "address": "789 Pine St, Village",
    "paymentInfo": "5678-1234-5678-4321",
    "firstName": "Bob",
    "lastName": "Jones",
    "dateOfBirth": "1978-09-30",
    "haveAccount": true,
    "driverLicense": "LMN456PQR"
  }
]
```

```
// MongoDB Collection: Payment
[
  {
    "_id": ObjectId("..."),
    "isCreditCard": true,
    "userPaymentinfo": "1234-5678-9012-3456",
  "cardNumber": 1234567890123456,
  "amount": 500.0,
  "cardHolder": "John Doe",
  "cardType": "Visa",
  "expDate": "12/25",
  "isExpired": false,
  "securityCode": 123,
  "accountingNum": 987654321,
  "routingNum": 123456789,
  "carCostPerDay": 50.0,
  "numberOfDays": 7,
  "additionalCharges": 20.0
},
{
  "_id": ObjectId("..."),
  "isCreditCard": false,
  "userPaymentinfo": "7890-1234-5678-9012",
  "cardNumber": 9876543210987654,
  "amount": 750.0,
  "cardHolder": "Alice Smith",
  "cardType": "MasterCard",
  "expDate": "10/23",
  "isExpired": false,
  "securityCode": 456,
  "accountingNum": 123987456,
  "routingNum": 987654321,
  "carCostPerDay": 60.0,
  "numberOfDays": 5,
  "additionalCharges": 15.0
},
{
  "_id": ObjectId("..."),
  "isCreditCard": true,
  "userPaymentinfo": "5678-9012-3456-7890",
  "cardNumber": 5678901234567890,
  "amount": 1000.0,
  "cardHolder": "Bob Johnson",
  "cardType": "American Express",
  "expDate": "08/22",
  "isExpired": false,
  "securityCode": 789,
  "accountingNum": 456789012,
  "routingNum": 654321987,
  "carCostPerDay": 70.0,
  "numberOfDays": 10,
  "additionalCharges": 25.0
}
```

```
]
```

```
// MongoDB Collection: Employee
[
  {
    "_id": ObjectId("..."),
    "employeeUsername": "john_employee",
    "employeePassword": "hashed_password",
    "employeeID": 123456,
    "carInfo": {
      "carName": "Toyota Camry",
      "carModel": "XLE",
      "carYear": 2021,
      "carType": "Sedan"
    }
  },
  {
    "_id": ObjectId("..."),
    "employeeUsername": "alice_employee",
    "employeePassword": "hashed_password_2",
    "employeeID": 789012,
    "carInfo": {
      "carName": "Honda Accord",
      "carModel": "Touring",
      "carYear": 2020,
      "carType": "Coupe"
    }
  },
  {
    "_id": ObjectId("..."),
    "employeeUsername": "bob_employee",
    "employeePassword": "hashed_password_3",
    "employeeID": 456789,
    "carInfo": {
      "carName": "Ford Explorer",
      "carModel": "Limited",
      "carYear": 2022,
      "carType": "SUV"
    }
  }
]
```

```
// MongoDB Collection: Rental
[
  {
    "_id": ObjectId("..."),
    "rentalReturnDate": "2023-12-01",
    "rentalDataOut": "2023-11-01",
    "isRenting": true,
    "requestRental": false
  },
  {
    "_id": ObjectId("..."),
    "rentalReturnDate": "2023-12-15",
    "rentalDataOut": "2023-11-15",
    "isRenting": false,
    "requestRental": true
  },
  {
    "_id": ObjectId("..."),
    "rentalReturnDate": "2023-12-10",
    "rentalDataOut": "2023-11-10",
    "isRenting": true,
    "requestRental": true
  }
]
```

```
// MongoDB Collection: System
[
  {
    "_id": ObjectId("..."),
    "agreementContracts": "Agreement123",
    "insurance": "InsuranceXYZ",
    "usernames": ["user1", "user2"],
    "passwords": ["password1", "password2"],
    "validInsurance": true,
    "registration": "RegistrationABC"
  },
  {
    "_id": ObjectId("..."),
    "agreementContracts": "Agreement456",
    "insurance": "InsuranceABC",
    "usernames": ["user3", "user4"],
    "passwords": ["password3", "password4"],
    "validInsurance": false,
    "registration": "RegistrationDEF"
  },
  {
    "_id": ObjectId("..."),
    "agreementContracts": "Agreement789",
    "insurance": "InsuranceLMN",
    "usernames": ["user5", "user6"],
    "passwords": ["password5", "password6"],
    "validInsurance": true,
    "registration": "RegistrationGHI"
  }
]
```

```
// MongoDB Collection: Car
[
  {
    "_id": ObjectId("..."),
    "available": true,
    "carColor": "Blue",
    "carModel": "Toyota Camry",
    "carYear": 2021,
    "carVIN": "4T1G11AK4MU581787",
    "carPrice": 25000.0
  },
  {
    "_id": ObjectId("..."),
    "available": false,
    "carColor": "Red",
    "carModel": "Honda Accord",
    "carYear": 2020,
    "carVIN": "1HGCV1F15LA117562",
    "carPrice": 22000.0
  },
  {
    "_id": ObjectId("..."),
    "available": true,
    "carColor": "Silver",
    "carModel": "Ford Explorer",
    "carYear": 2022,
    "carVIN": "1FMSK8FH3NGA59195",
    "carPrice": 35000.0
  }
]
```

```
// MongoDB Collection: CarRentalLocation
[
  {
    "_id": ObjectId("..."),
    "rentLocation": "Downtown",
    "locationServices": true,
    "city": "San Diego",
    "state": "California",
    "zipCode": "54321",
    "country": "United States"
  },
  {
    "_id": ObjectId("..."),
    "rentLocation": "Beachfront",
    "locationServices": true,
    "city": "Coastal City",
    "state": "Seaside State",
    "zipCode": "98765",
    "country": "Canada"
  },
  {
    "_id": ObjectId("..."),
    "rentLocation": "Hillside",
    "locationServices": false,
    "city": "Highland Town",
    "state": "Mountain State",
    "zipCode": "12345",
    "country": "Mexico"
  }
]
```

Conclusion:

The choice of MongoDB as the document-based, NoSQL database for the car rental software system is driven by the need for flexibility in handling diverse and semi-structured data. The logical organization of data into collections aligns with the different entities within the system. While MongoDB introduces trade-offs in terms of consistency and query complexity, these are deemed acceptable in exchange for the scalability and adaptability required for a dynamic and growing car rental platform.