BeAvis Car Rental Software Design Specifications

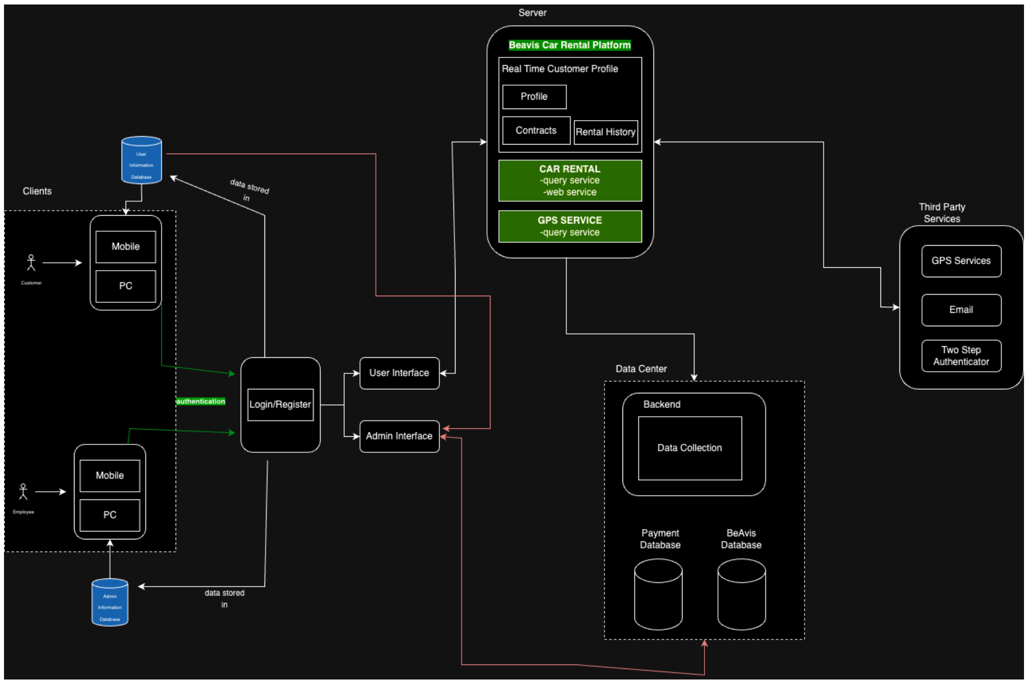
Prepared by: Caroline Wilkins, Yearly Gonzalez, Ken Robinson

System Description

This document contains the necessary information regarding the BeAvis Car Rental system in order to build the client’s preferred product. The text below will include the necessary components to construct the system itself. This document provides an Architectural diagram of the major system components, a UML Class Diagram, a detailed description of said diagram, an approximated timeline for the development process, and a general software development plan. The client, BeAvis Car Rentals, has requested the following system with the intention of creating a more convenient and accessible process for their customers to rent vehicles with. The BeAvis Car Rental system is a multi-platform program that allows customers to complete the car rental process, locate BeAvis establishments, and prepare to both pick up and drop off their reserved vehicle. During the usage of the BeAvis Car Rental system, customers will be able to virtually complete the entirety of the car rental process. Customers will be able to virtually view, book, and locate their BeAvis rental car in a single space through the utilization of this system.

Software Architecture Overview

Software Architecture Diagram

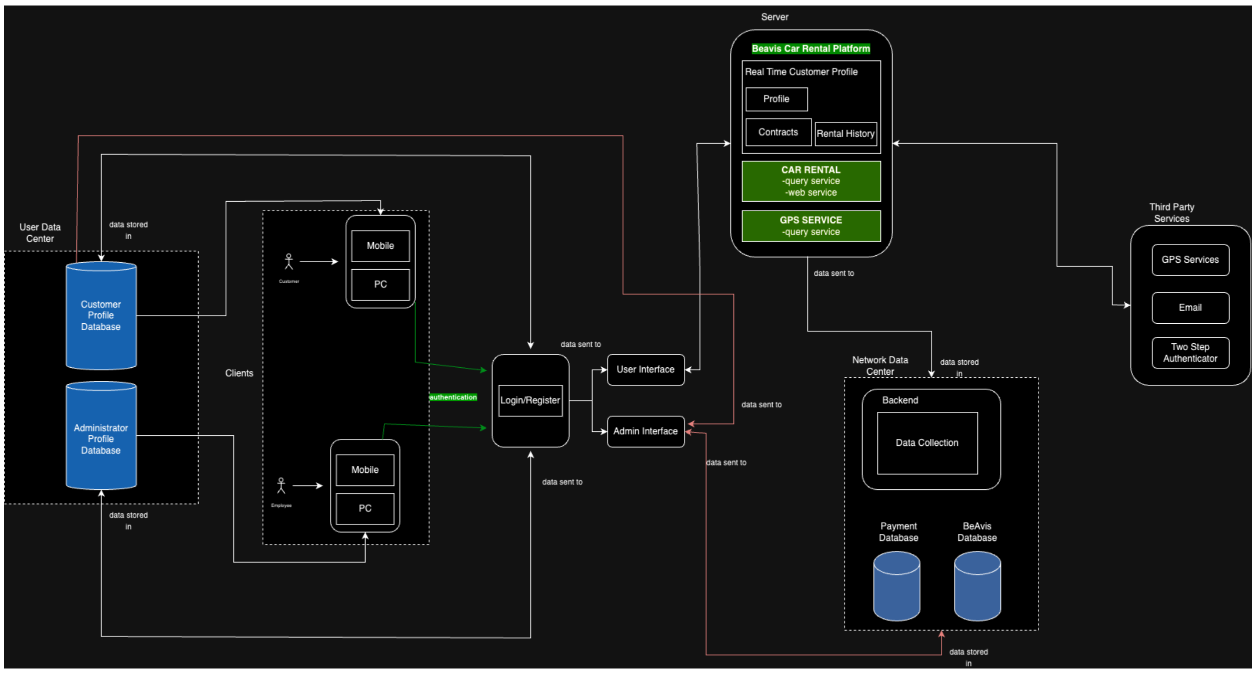


The above figure is the first rendition of the BeAvis system SWA diagram.

Software Architecture Description

The included software architecture diagram represents the software system components of the BeAvis Car Rental system. The flow follows a left to right pattern beginning with the potential clients, those being customers and employees each with the option of PC or mobile. The clients are both connected to the login or registration component through authorization. There are two databases that store client account information, one is designated for customers and the other for employees. Data collected from the authorization is stored in one of the two account information databases with respect to the client's credentials. Due to the difference in functionality in regards to client types, there are two components for software interface. Following the path of a customer client, the interface both sends and receives data from the BeAvis platform itself. This server consists of several components in order to fulfill all necessary operations. The real time customer profile component includes the various elements needed to fulfill all customer rental management functions. It stores the data for customer profile, rental contracts, and rental history. The remaining two components of the BeAvis server are the car rental functions and GPS functions. The former will consist of query and web service operations whilst the latter requires only that of query. Outside of the user interface, the server also connects to the data center and third party services. The latter includes gps services, email and two step authenticator connectivity. The former collects the data from the system, it consists of the backend for data collection, the BeAvis database, and a database holding customer payment methods to adhere to system requirements of security. This data center component also has a two way connection between itself and the administrator interface to allow employees to view and alter system information. Finally, the admin interface has data sent to it from the user information database to allow the viewing of customer account data when necessary.

Revised Software Architecture Diagram

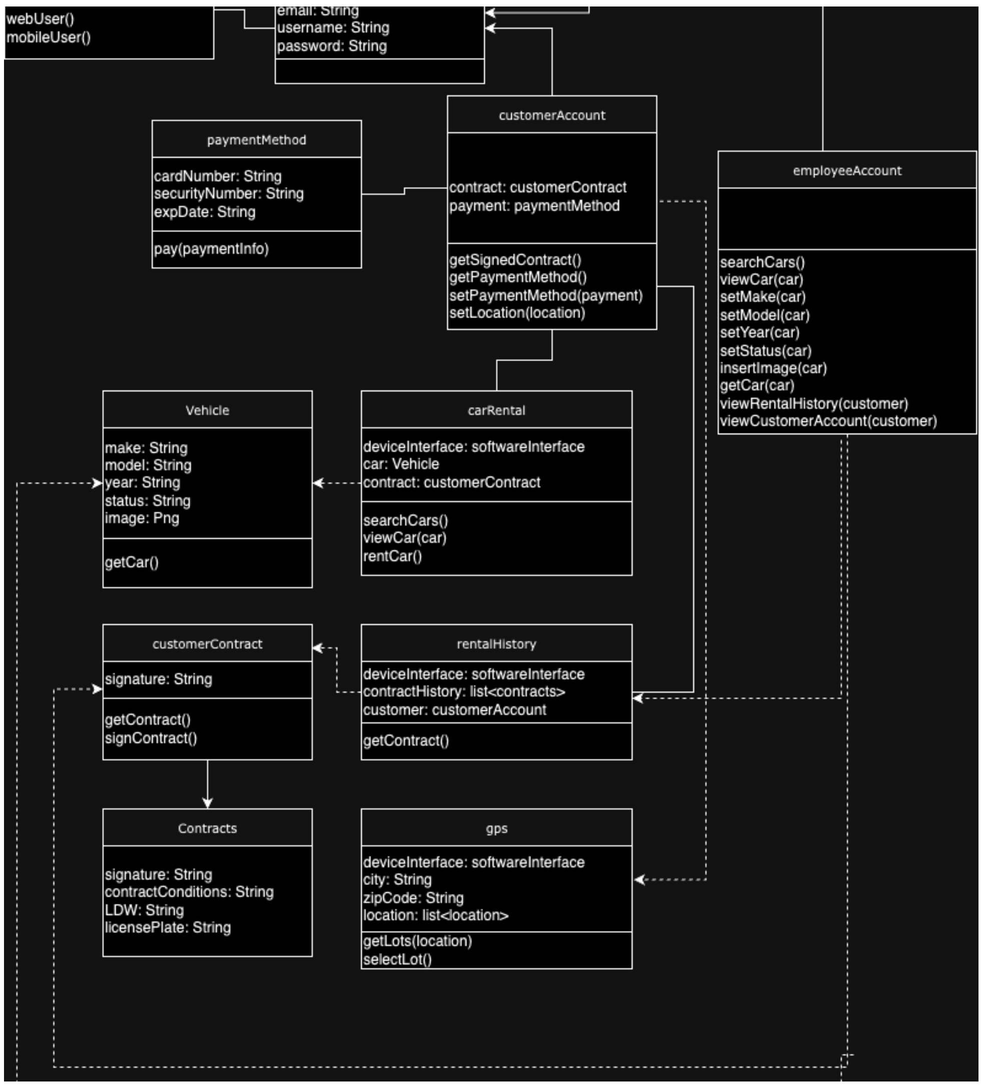


The above figure is the revised rendition of the BeAvis system SWA diagram.

Amendments:

The SWA diagram above was altered to more effectively divide the databases in a way that reflects the proposed data management plan. There are now two data centers present to encompass the wide range of information that the BeAvs system must access and store. Where there was initially one Data Center of same naming, the current SWA diagram now has a Network Data Center as well as a User Data Center. The former has no other changes aside from the name, continuing to hold backend data as well as both the Payment and BeAvis databases. The newly added User Data Center stores the Customer and Administrator databases. The goal of this new diagram is to represent the division of similar data among databases in our proposed data management plan. It provides a better focus on the databases themselves as well as the flow of information through the newly added labels noting where data is stored or sent to. In short, this revised diagram holds the same foundation as the initial diagram, however, it more evidently showcases a handling of data throughout the system itself.

UML Diagram

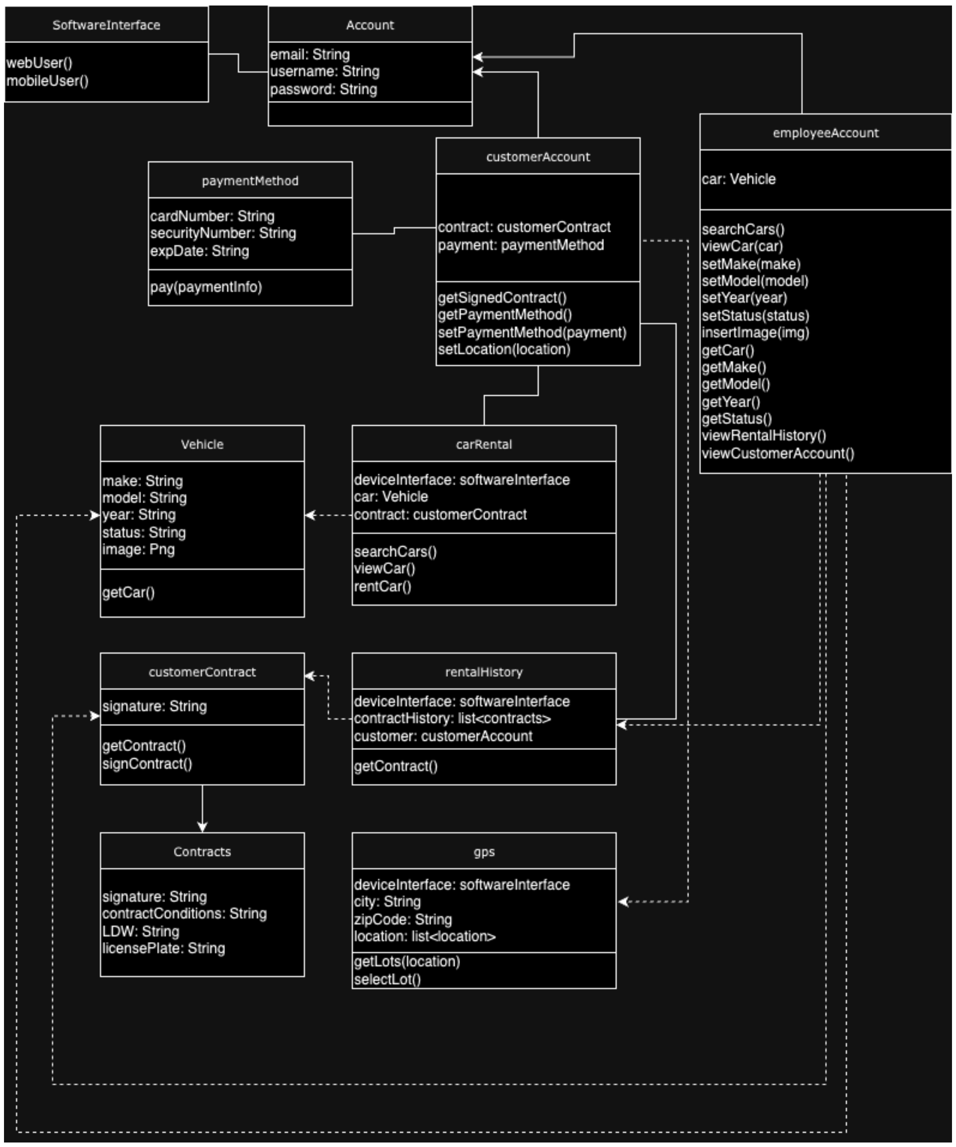


The above figure is the first rendition of the BeAvis system UML diagram.

UML Class Overview

The included figure is a UML diagram for the BeAvis system. The diagram begins with the **SoftwareInterface** class. This class is responsible for whether the software will boot the web or mobile version of the application. It includes no operations and two attributes being that of *webUser()* or *mobileUser()*. Once the device version is set, the diagram follows the user through account creation or login. There are three account classes in the software: Account, customerAccount, and employeeAccount. The **Account** class is the general class for the accounts. Its attributes consist of the various components necessary to create and maintain an account through the system, those being an *email*, *username*, and *password*. Each attribute here is a string data type to be inherited by the other two account classes. From there the customerAccount and employeeAccount are specialized classes that deal specifically with customers and employees accounts respectively. The **customerAccount** class includes two attributes to coincide with its four available operations. The *contract* attribute is a customerContract object whilst the *payment* attribute is a paymentMethod object. These components are necessary for the functionality of the getSignedContract(), getPaymentMethod(), setPaymentMethod(payment), and setLocation(location) operations which allow for the obtaining of completed customer contracts, establishment and viewing of payment methods, and setting of customer location. The **employeeAccount** class is our most complex class as the employee should be able to access most information in the software as well as modify it. There are no attributes for this class, however, it includes the ten following operations: searchCars(), viewCars(car), setMake(car), setModel(car), setYear(car), setStatus(car), insertImage(car), getMake(car), getModel(car), getYear(car), getStatus(car), getCar(car), viewRentalHistory(customer), and viewCustomerAccount(customer). The **paymentMethod** class is the object that holds all the relevant payment information that is saved in the user's account within the software. It includes three attributes, *cardNumber*, *securityNumber*, and *expDate*, all of the string data type. The singular operation of this class is the function to complete a payment, or pay(paymentInfo). The **carRental** class is responsible for all functions related to the renting of cars themselves, it can be considered the core of the software. There are three components within this class, those being, *deviceInterface*, *car*, and *contact*. These attributes are objects of the softwareInterface, Vehicle, and customerContract classes respectively. The operations for this class, or searchCars(), viewCar(car), and rentCar(), allow users to search through all available cars, view a singular car, and begin the renting process. The **Vehicle** class will be responsible for creating objects for every individual vehicle, including all the details of the vehicle. It is largely composed of attributes rather than operations, with its only function being that of getCar(). The components of this *class*, *make*, *model*, *year*, *status*, and *image*, are almost all string variables that cover general information of the vehicles with an exception of the *image* attribute being of png datatype. The **rentalHistory** class is responsible for creating an object that will hold the history of the customer's contract history. It has an individual operation of getContract() and three attributes begin, *deviceInterface*, *car*, and *contract*. The aforementioned variables are data types softwareInterface, vehicle object, and customerContract object respectively. The **customerContract** and the **Contracts** classes are both related to the contracts that will be signed with the customers, the difference being the Contracts class itself will be responsible for creating the contracts while the customerContract will create an object that is responsible for holding the proof of the users agreement with said contract. The customerContract class has a single attribute, *signature* of the string data type, to allow for contracts to be signed. It includes two operations, those being getContract() and signContract(). The Contracts class lacks any operations, instead having four string attributes including *signature*, *contractConditions*, *LDW*, and *licensePate*. The last class is the **gps** class which is responsible for creating objects that hold the gps data for vehicles and lot locations in order to find the nearest lot to the customer. Its four attributes consist of a softwareInterface object named *deviceInterface*, two string variables named *city* and *zipCode*, and a list of the two variables called *location*. There are only two operations for this class with those being getLots(location) and selectLot().

Revised UML Diagram



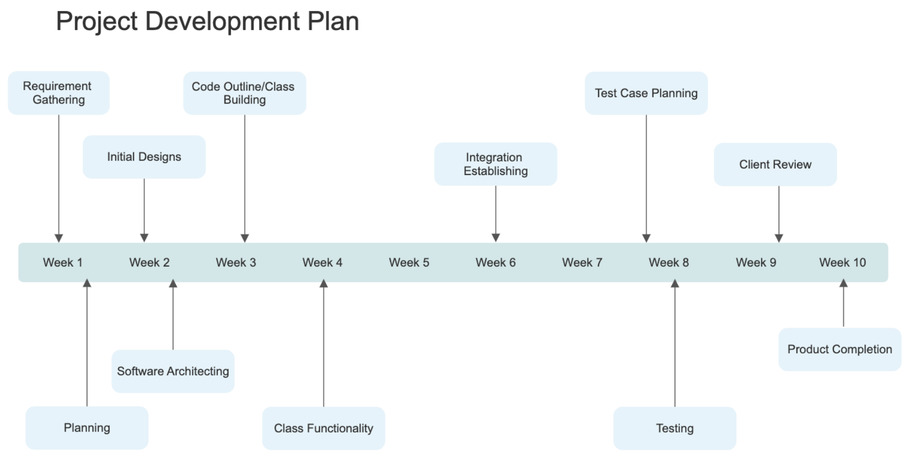
The above figure is the revised rendition of the BeAvis system UML diagram.

Amendments:

The UML diagram above was altered to allow for more efficient functionality in accordance with client demands. In the **employeeAccount** class, the setters statements were changed to take different parameters. Originally, the operations were as follows: setMake(car), setModel(car), setYear(car), setStatus(car). These operations will instead take parameters fitting to the feature rather than the *car* attribute. They will be setMake(make), setModel(model), setYear(year), setStatus(status). The attributes necessary to fulfill these operations are already present in the **vehicle** class. The getter statements were also changed to no longer take any parameters and thus avoid logical errors. They are to be getMake(), getModel(), getYear(), getStatus(), and getCar(). This removal of parameters is also to be done to the viewCar(), viewRentalHistory(), and viewCustomerAccount() operations in the same class. In the carRental class, the viewCar() operation will also take no parameters.

Development Plan and Timeline

Project Timeline



Timeline Description

The above figure details a general software development timeline. The development of the BeAvis software is to be divided into eleven tasks over a period of ten weeks. The first task to be completed is **requirement gathering** to be designated to the *business analyst team*. They will obtain a list of software components and needs from the client within half of a week. The latter half of the first week is set for **planning** by the *project manager*. A general development plan for product development is to be created including risk analysis, scope, and testing. The former half of the second week focuses on the planning of **initial designs**. The *design team* will create proposals for design UX/UI and settle on general design direction in accordance to requirements given by the client. The latter half of this week revolves around **software architecting**. The *software architect* will create a software development plan including software architecture and unified modeling language diagrams to be used by the development team. Over the entirety of the third week the *software developers* will be tasked with **code outline and class building**. This covers the implementation of a general version of each class. The following week will stretch into **class functionality**, also designated to the *software developers*. The development team will implement attributes and operations to each class in accordance with the architecture diagram at this time. Once completed, *software developers* will move to a two week period of **integration establishing**. This brings the tasks of securing necessary integration with external systems and allowing for connectivity with all noted third-party applications. Over the first half of the eighth week, *QA engineers* will be in charge of **test case planning**, or establishing a form of testing and developing necessary test cases. The task of **testing** itself will fall onto the *testers and software engineers* as they spend the latter half of week eight as well as the former half of week nine running through various test cases. This is to cover the conducting of unit, integration, and system tests, completing static code analysis, and fixing bugs as they are found. Once the system itself has been thoroughly tested, the remainder of the ninth week will be spent with the *business analyst team* as they conduct a **client review**. This will consist of the presenting of the finished product to the client for assessment of features and performance. The final week of this project is to be reserved for **project completion** as a whole. All teams will work together to deploy the current system to the production field. Members will monitor and fix bugs with user feedback as well as plan for future improvements.

Software Development Plan

The BeAvis Car Rental Software is being developed to bring a more user friendly dynamic to the prior car rental system in place at BeAvis. Prior to the development of this software, BeAvis users experienced difficulty in accessing the car rental information, as well as navigating the different steps in the car rental process without in-person assistance. The proposed car rental software system resolves these issues on the user end by providing a simple virtual interface that allows the user to view, reserve, and locate their rental car in a single designated space. In increasing visual simplicity on the user end, the software system successfully addresses the issues that the prior rental system posed.

The main development tasks required of this system include design, coding, integration, and testing. The design development process entails creating a UX/UI design that aligns with the views of our clients at BeAvis. Our design team has worked closely with our software architects: Caroline Wilkins, Ken Robinson, and Yearly Gonzalez to create the software architecture required of our proposed system. The first and second phase of our design process was completed by members of our design team. The next phase of development, coding, is outlined by the development and functionality testing of different class implementations. The UML diagram created can be referenced as a visual to the coding phase of the rental system development process. The integration of these proposed architecture and coding systems to be implemented in this software will be closely tested with the pre-existing hardware and external systems that BeAvis uses at their car rental centers. The implementation phase of development will be passed along to members of our software development and engineering teams prior to the testing and client review phases. Finally, our testing phase of development will require cross-functionality testing between the elements of the car rental software architecture and coding scripts. All teams will collaborate in daily meetings to ensure software efficiency and confer about elements of the software that may need to be debugged or enhanced.

The development will closely follow a predetermined project schedule, as demonstrated by our project development timeline. Before the true stages of development begin, our software development team will take approximately half of a week to gather system requirements which will be followed by project manager review, taking another half week to conduct risk analysis, scope, testing, and development planning. After a week of planning and creating proposals for UX/UI design, the development process will begin.

The BeAvis Car Rental Software development team will utilize the evolutionary prototyping model of software development. Our clients have provided us with the initial concept, design, and implementation of the prototype through their system requirements documentation. As developers, we will create and refine the software architecture and coding scripts until they run in tandem effectively. In working closely with our clients to implement client feedback, the car rental system prototype will not be released until our development team and clients reach a formal agreement on the software's release. The evolutionary prototyping model promotes quality assurance and will allow us to put forth high-quality software.

Development of the new rental software will require a number of resources that will need to be funded on the client-end. Databases for payment information, administrative information, and vehicle information will need to be set up electronically for the management and storage of user and corporate data. Beavis clientele have the option of choosing the most efficient and cost-effective platform for doing such. A financial agreement will also need to be set in place for third-party service providers.

Risk analysis will be conducted on all features throughout the development process. Our main concern lies in information security. Dealing with personal payment and identification information on the user-end will require a high-level of security and specific access-points for clientele. Two-factor authentication will mitigate this challenge on the user-end. For software administrators, only a select few members of the Beavis corporate team should have access to the financial databases set in place. Limiting access and verifying one's identity should encourage and instill information security within the system.

Throughout the development process we will track different metrics to monitor system performance. Time complexity of our coding scripts will be reported after major changes are made to the script in order to track system performance and run-time. Each department within the software development team will participate in biweekly progress meetings to bring potential challenges to light, and to celebrate successes of the evolving software program. As a result of performance tracking, reports will be sent over to Beavis corporate to communicate progress after each of these biweekly meetings.

Testing Plans

Test Plan

Test Set One (Customer Functions)

Unit:

Test Case: Set Payment- setPayment(payment)

Objective: Ensure the user is able to input their debit or credit card information and store said method for future use.

Input: User specified payment method which includes a paymentMethod object.

Test Steps: paymentMethod object is created, arbitrary values are assigned to the attributes cardNumber, securityNumber, expDate. String variable is created and set to have the same previously mentioned arbitrary values as the paymentMethod object. setPayment method is called and utilizes the paymentMethod object as the parameter. If-Else statements are utilized to test equivalence between paymentMethod and test string value.

Expected Result: User payment information stored in a separate database within the system. The printed result is to be "true" if the string value of the paymentMethod is equivalent to the test string variable. Result is "false" if these values are not equivalent.

Description: The above testing outline will check the functionality of the BeAvis system at a unit level. It has the explicit goal of addressing one of the many features available for users on the customer side, setting a payment method. It utilizes the <u>setPayment(payment)</u> operation in the customerAccount class. This operation is meant to allow customers to insert their credit or debit card information and store it for future use. The test itself will require the creation of a **paymentMethod** object. This requires arbitrary values to be input for **paymentMethod**'s attributes, *carNumber*, *securityNumber*, and *expDate*. For purposes of comparison, a String variable will be initialized to contain the same arbitrary values. The true testing portion will consist of an if-else statement comparing the values. To obtain the String value of the **paymentMethod** object, the operation <u>getPaymentMethod()</u> must be called. The if-else statement will test for equivalence. If the two values are equal, the test will return "true" and vice versa if they are not. This will conclude the test.

Integration:

Test Case: Rent Car- rentCar()

Objective: Ensure the user is able to rent a car of their choosing.

Input: User specified vehicle in the form of a chosen car.

Test Steps: A car is chosen by the User in the system based on the **Vehicle** objects attributes that will be displayed in the UI. This function is called from the **carRental** class. After the system has pulled up the attributes of the desired **Vehicle** object, the User can choose to proceed to "rent" the car. After this step is chosen, the function <u>rentCar()</u> will change the value in the *status* attribute of the **Vehicle** object to display the car being rented out. Lastly a *customerContract* will be generated that will contain the proof of rental and sent back to the **customerAccount** class.

Expected Result: The expected result is that the attribute in the **Vehicle** class for the given **Vehicle** object named *status* will change its value, as well as an object, *customerContract*, is generated for the User. The Test will succeed if the *status* attribute is changed to the right option, and a proper *customerContract* object is created with the correct information. The Test will fail if either of these two fail to change correctly.

Description: This test case will test both the **carRental** and **Vehicle** classes, as well as having an object that is sent back to **customerAccount**, in order to ensure that the functions are correctly pulling and modifying the data across more than just one class. The goal of this test is to ensure that a created **Vehicle** object will have its *status* modified by an outside class, as well as generating a *customerContract* that will be generated in one class and sent to another. To begin the test the User would select their preferred car and then select the option to rent the car. This would initiate the rentCar() function. This function would proceed to change the *status* attribute of the "Car" **Vehicle** object that will be used as the input, to "Rented". Following this, a *customerContract* would be generated by **carRental** class in order to store the proof of the rental as well as an electronic signature of the User.

System: getSignedContract()

Test Case: Signed Contract - getSignedContract()

Objective: Ensure that there is a signed contract in place between Beavis and the renter.

Input: Signature of user purchasing rental.

Test Steps: Contract is shared on a pop-up screen after renter makes it past the rental choices page as long as getContract() is successfully run. If a signature is not acquired/imputed from the user after getSignature() is called, they will not be able to make it to the next stage of the process, purchasing the rental. Tests will need to be run on different operating systems to make sure that the contract pop-up is successfully displayed and all users going through the rental process will be able to interact with the contract.

Expected Result: The expected result is that getContract() and getSignature() successfully returns a signature object to the database.

Description: This test case will test the CustomerContract and Contract classes which will return a String object to the database that can be shared between both classes. As long as the Signature string matches the renter name, the test will successfully pass. If the Signature string is null or not equal to the renter name, the test will fail. The contract is essential to the system design as an unsigned contract will not permit the rental process to continue.

Test Set Two (Employee Functions)

Unit:

Test Case: Set Status- setStatus

Objective: Ensure employee users are able to input or update the status of a specified vehicle.

Input: User specified status which includes a Vehicle object as the parameter.

Test Steps: Vehicle object is created, arbitrary values are assigned to the attributes status, make, model, year, and image**.** Test string variable is created and set to have a given status. setStatus method is called and utilizes the Vehicle object as the parameter. If-Else statements are utilized to test equivalence between getStatus and test string value.

Expected Result: Data regarding the vehicle's status is updated and stored within the system. The printed result is to be "true" if the value of getStatus is equivalent to the test string variable. Result is "false" if these values are not equivalent.

Description: The set status test verified the functionality of a specified BeAvis system feature at the unit level. The class to be tested is **employeeAccount**. The goal of this test is to successfully allow employee users of the system to alter the given status of a specified vehicle. In order to begin the test a vehicle object must first be made, most attributes can be arbitrary and simple as they will not impact the results of the test. The only attribute to make note of is *status*, set through the use of the setStatus(status) method operation. A String variable will be created for purposes of comparison and must be the same value as the chosen value for *status*. Following this setup, the testing itself must take place by way of an if-else statement. Using the previously created vehicle object, getStatus() will be called and the returned value will be compared to the test string. If the values are equivalent "true" will be returned and the test will be passed, vice versa holds if the values are unequal. This concludes the test.

Integration:

Test Case: Get Car- getCar()

Objective: Ensure Employee users are able to view and modify cars in the system.

Input (if any): Input is a specific **Vehicle** object specified by the Employee user.

Test Steps: The Employee user will select a specified **Vehicle** object that will be chosen from a list. The Employee will then select to execute the getCar() function on the chosen object. Finally the function will pull the **Vehicles** attributes from the **Vehicle** class and display it to the Employee User.

Expected Result: The expected output is that the data for a chosen **Vehicle** object is displayed to the Employee User, ready to be used, modified, or closed. The Test succeeds if the data for the right **Vehicle** object is loaded for the Employee User. The Test fails if the wrong data is loaded.

Description: This test is meant to check if the **employeeAccount** class can interact with the **Vehicle** class by accessing objects created by that class. The goal is to test this via loading the data of a chosen object with a function in the **employeeAccount** class. The test begins by choosing a car you want to check the data on. The specific car choice should not matter, so when performing multiple tests, different cars should be chosen. The chosen car will be chosen from a list, from which executing the getCar() function will be an option. This function should then retrieve the attributes attached to this object, such as: *status, make, model, year,* and *image*. Lastly the function will display these attributes to the Employee User, causing the test to succeed if the correct attributes are displayed.

System: viewCustomerAccount(customer)

Test Case: Display Customer Account Information-viewCustomerAccount(customer)

Objective: Ensure customer account information is able to be displayed to users once they have confirmed their identity from the login process.

Input: User contract and payment method are the attributes needed to be returned from the getSignedContract(), getPaymentMethod(), setPaymentMethod(payment), and setLocation(location) methods.

Test steps: Ensure that customerContract and paymentMethod are correctly returned when the tests passed through the aforementioned input methods are run. Tests should be able to account for missing payment methods and contracts and correctly guide user input back to where information is needed by the system.

Expected Results: customerContract and paymentMethod should return Strings in line with what the renter input into the rental database.

Description: customerAccount class should correctly implement methods so that they can be used by the other classes in order to display customer account information at every step of the rental process. The program should keep track of each update to the customer account which will be reviewed in entirety at the final checkout stage of the rental process. The tests will be run in order to evaluate the speed and accuracy of the rental software so that proper security and efficiency are maintained.

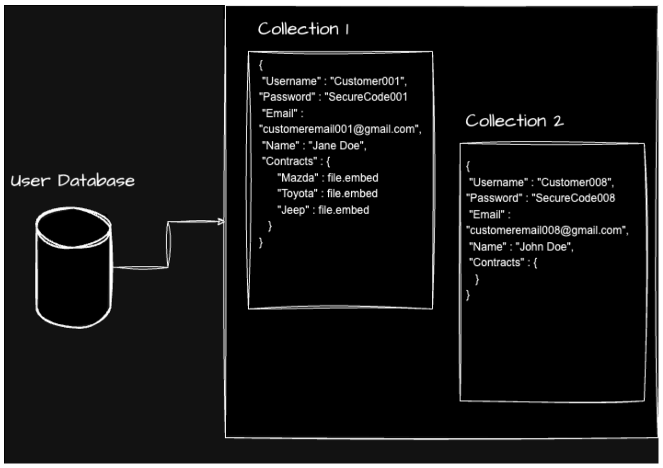
Data Management

Data Management

Data Management

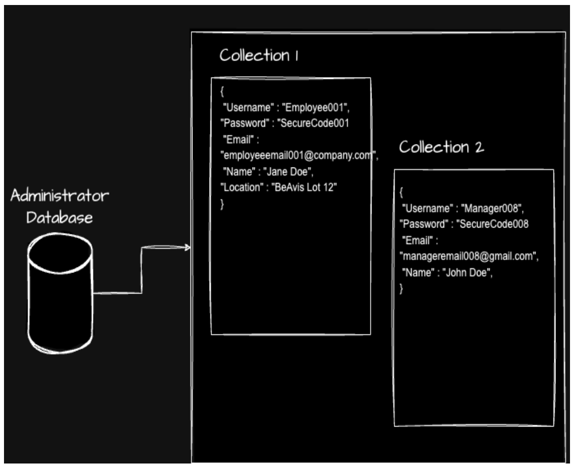
In order to fully manage the data needed to operate the BeAvis system, a total of four databases must be integrated. These databases include the user database, administrative database, payment database, and the BeAvis database itself. Below is the development plan for the aforementioned databases including diagrams and descriptions of their functionality.

User/Administrator Databases



```
{
 "Username" : "Customer001",
"Password" : "SecureCode001
 "Email" :
"customeremail001@gmail.com",
 "Name" : "Jane Doe",
 "Contracts" : {
     "Mazda" : file.embed
     "Toyota" : file.embed
     "Jeep" : file.embed
   }
}
```

```
{
 "Username" : "Customer008",
"Password" : "SecureCode008
 "Email" :
"customeremail008@gmail.com",
 "Name" : "John Doe",
 "Contracts" : {
   }
}
```

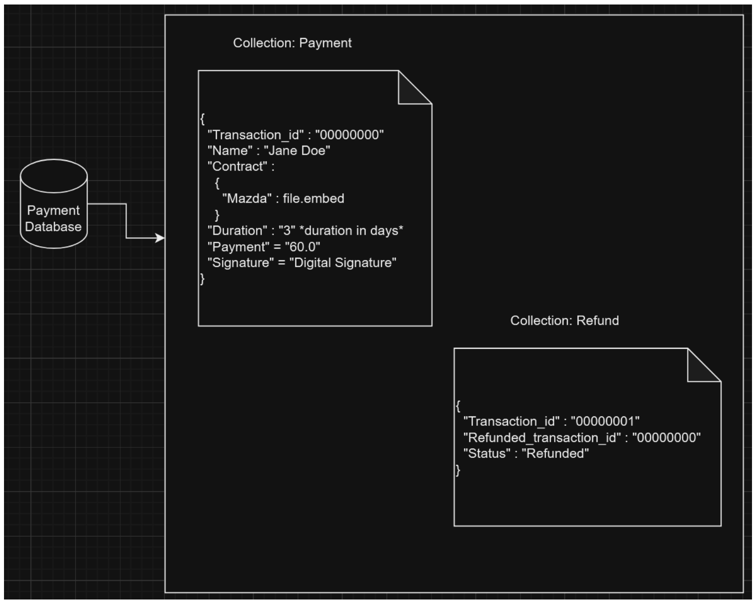


```
{
 "Username" : "Employee001",
"Password" : "SecureCode001
 "Email" :
"employeeemail001@company.com",
 "Name" : "Jane Doe",
"Location" : "BeAvis Lot 12"
}
```

```
{
 "Username" : "Manager008",
"Password" : "SecureCode008
 "Email" :
"manageremail008@gmail.com",
 "Name" : "John Doe",
}
```

The above diagrams showcase the Customer and Administrator database portion of the BeAvis system. The data management plan for the BeAvis car rental system utilizes two different databases to store its information of the user profiles. Though these databases could be combined, choosing to divide between the two will ensure ease of usage for employees. Client requirements call for an intuitive design, this request extends to employees as well. Including a singular database for all forms of profiles regardless of one's level of access has the potential to cause confusion when employees must access the information. Calling upon operations that require searching through and obtaining customer information would be needlessly convoluted if there was the addition of every employee profile. Furthermore, the separation of data simplifies the process of access distribution. The operations available between employees and customers are very different and it is imperative to maintain high security in regards to which profiles are given upper-level access. Despite the general distinction between the two databases, the structures and technology used to construct them are relatively identical. The profile databases are both to be noSQL, document structures. The overall flexibility of noSQL allows for scalability that accounts for the amount of users BeAvis may have. Though an SQL system has the potential to provide a more straightforward structure, noSQL in this instance grants flexibility that increases efficiency. As a whole, the noSQL document structure split between customer and administrator databases creates an efficient and convenient way to access profile information.

Payment Database

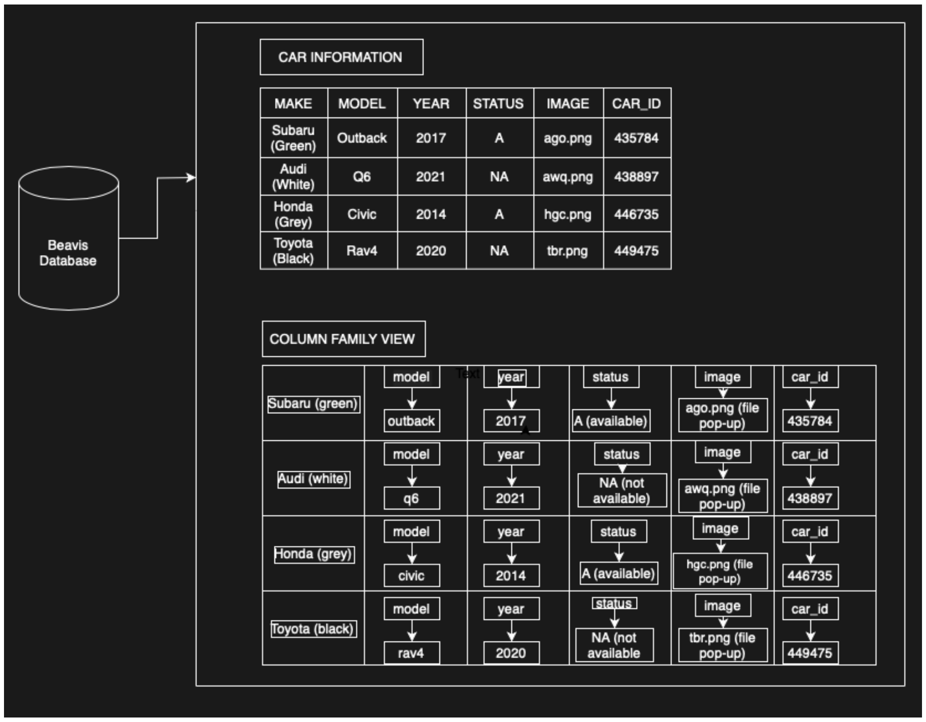


```
{
  "Transaction_id" : "00000000"
  "Name" : "Jane Doe"
  "Contract" :
    {
      "Mazda" : file.embed
    }
  "Duration" : "3" *duration in days*
  "Payment" = "60.0"
  "Signature" = "Digital Signature"
}
```

```
{
  "Transaction_id" : "00000001"
  "Refunded_transaction_id" : "00000000"
  "Status" : "Refunded"
}
```

The attached diagram is an example of the Payment Database structure. Similar to the databases above, the Customer and Administrator databases, this Payment database will also be a document type of noSQL. The purpose of this database is to securely store the information that has to do with money transactions, and therefore one does not need to query the database often. Everytime a transaction is finalized, the specifics of the transaction are added to a “document”, along with a unique id, to be saved in the database. One would only have to query the database to determine the amount of money to return when refunding or when asked to show proof of sale. In the cases of refunding, a new “transaction” document would be created when a refund was successful, that would include the unique id of the “transaction” that was being refunded as well as a confirmation of the refund itself. As stated above, the flexibility and intuitiveness of the Document NoSQL structure makes it ideal for our purposes. Although we could use SQL to achieve the same results, the need to be able to store and handle large amounts of the payment data, as well as our focus on scalability and availability, makes the Document NoSQL structure the best option for our database structures.

BeAvis Database



MAKE	MODEL	YEAR	STATUS	IMAGE	CAR_ID
Subaru (Green)	Outback	2017	A	ago.png	435784
Audi (White)	Q6	2021	NA	awq.png	438897
Honda (Grey)	Civic	2014	A	hgc.png	446735
Toyota (Black)	Rav4	2020	NA	tbr.png	449475

The diagram displayed is an example of how we intend to structure the Beavis Database. Using a wide column database structure within NoSQL, car information will be stored along with all necessary information needed to properly keep track of car inventory. This same wide column structure can be used by the database for all car identification information that can be transferred from renter input into the database (i.e. keep track of miles driven, miles in tank, and input fields noting any car difficulties). In the example to the left, keys are column titles and values are the row values stored in the database for each car entered into the system. The column family view shows the grouping of similar columns into column families, which in this case is car identification information. The row key, or left most column, uniquely identifies the row, or car, in the family which makes querying the related column values a fast and efficient process. Because of the direct relation in car identification information, using key-value pairs within a wide column structure is ideal for data management.

Data Management Strategy

The goal of this database design is to ensure that the obtaining and alteration of information is both efficient and secure. Initially, there were potential plans to include more databases or, alternatively, diminish the amount to no more than two. The final decision of four databases allows the data to be divided logically without excessive pulls from the code to obtain information from areas that are only slightly disjointed. This is seen largely in the user and BeAvis databases. The former holds user account information, this includes general data such as usernames and passwords. In addition to this, however, signed customer contracts are stored and pulled from this database. Similarly, the BeAvis database holds data for every car available across the BeAvis lots rather than splitting this into a database per location. Information is divided by their purpose or reason of access. Signed contracts and customer information are all used by employees to manage current customers. This division technique allows information in similar yet not entirely equal categories to be accessed easily from one area without branching into another database. Issues with storage capacity and data formatting are handled through the use of noSQL databases. Due to the sheer amount of data across the various locations and accounts, it is beneficial for the BeAvis system to have databases that are both scalable and flexible. The rigid formatting of SQL databases is not efficient for the large-scale data accessing, storing, and manipulation that this system requires. Furthermore, the flexible structure of the various noSQL databases allows for the needs of specific operations to easily be met. Utilizing Wide-Column and Document databases allows data to be stored in malleable structures. The former is fitting for the BeAvis database as it is capable of storing the unique vehicle information in an editable format and high scalability to address the high volume of vehicles available across BeAvis locations. The latter allows for unique profiles and payment methods to be stored in documents that are capable of including their own classes. For the two user and payment databases, storing information in unique documents allows additional information to be added if needed for a specific user or payment method. It adds for more flexibility in storage whilst also accounting for the need for security with the personal information stored in the databases. These decisions will allow the BeAvis system to access the necessary information to complete its various functions without becoming inefficient due to the amounts and types of data being stored.