Car Rental Software System

April 21, 2023

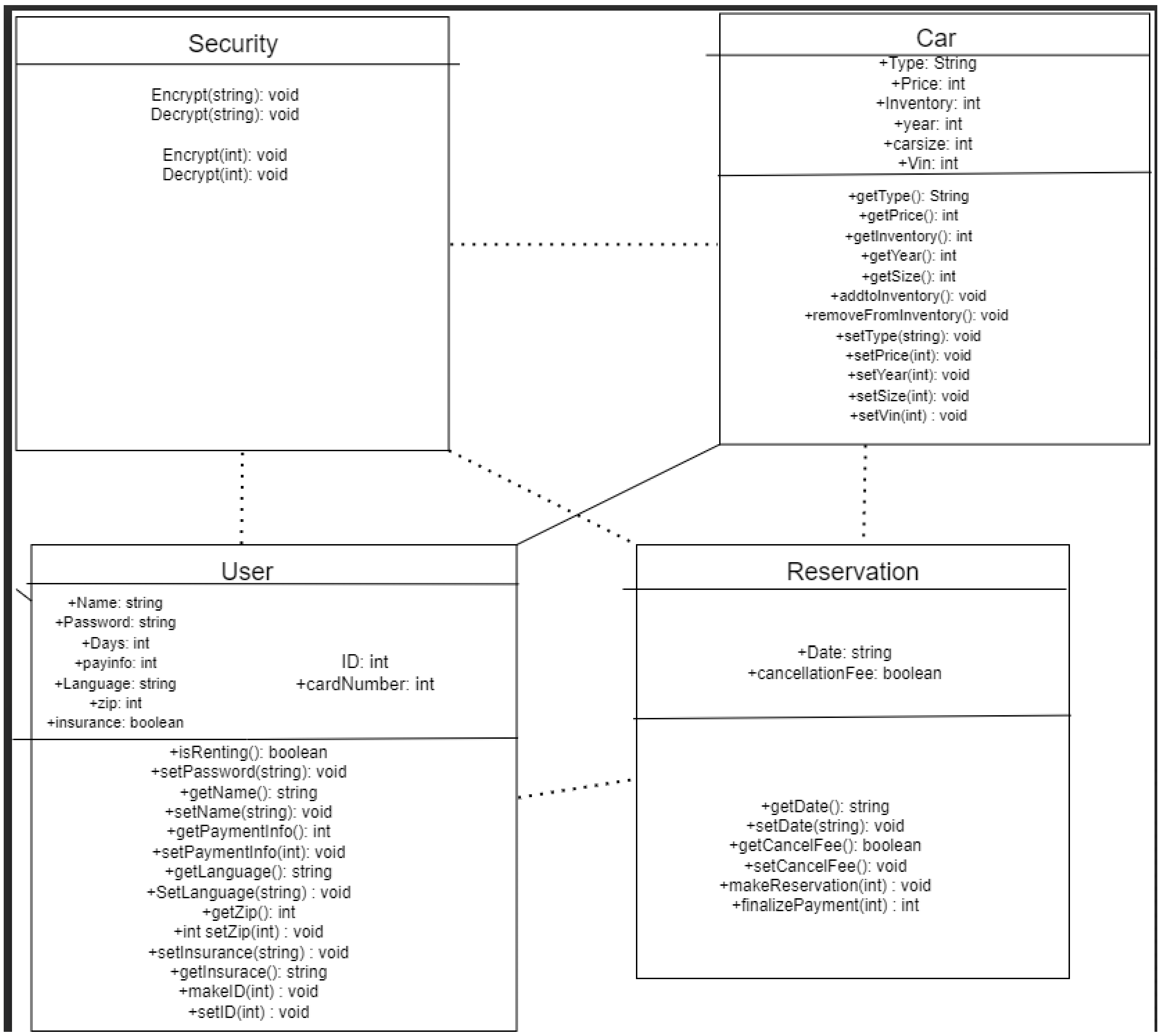
Jonathan Siegel, Andres Choucair, Kyle Phillips

System Description

The software being presented is a car rental process that works entirely electronically. By removing the traditional pen and paper process, users can view what BeAvis has to offer without having to go in person or reaching out to a representative, as our software system will provide all meaningful information regarding the cars we offer to our customers. Users can experience our software systems through desktop and/or mobile devices with a working internet. BeAvis has a website application for users who wish to access our software system through a desktop device. Alongside the desktop application users can also access our software systems on any mobile device through our mobile application. Outside the consumer perspective, BeAvis employees will be granted the ability to use the software system, as they are encouraged to consistently update information regarding the vehicles they are providing.

Software Architecture Overview

UML Diagram:



UML Diagram Description:

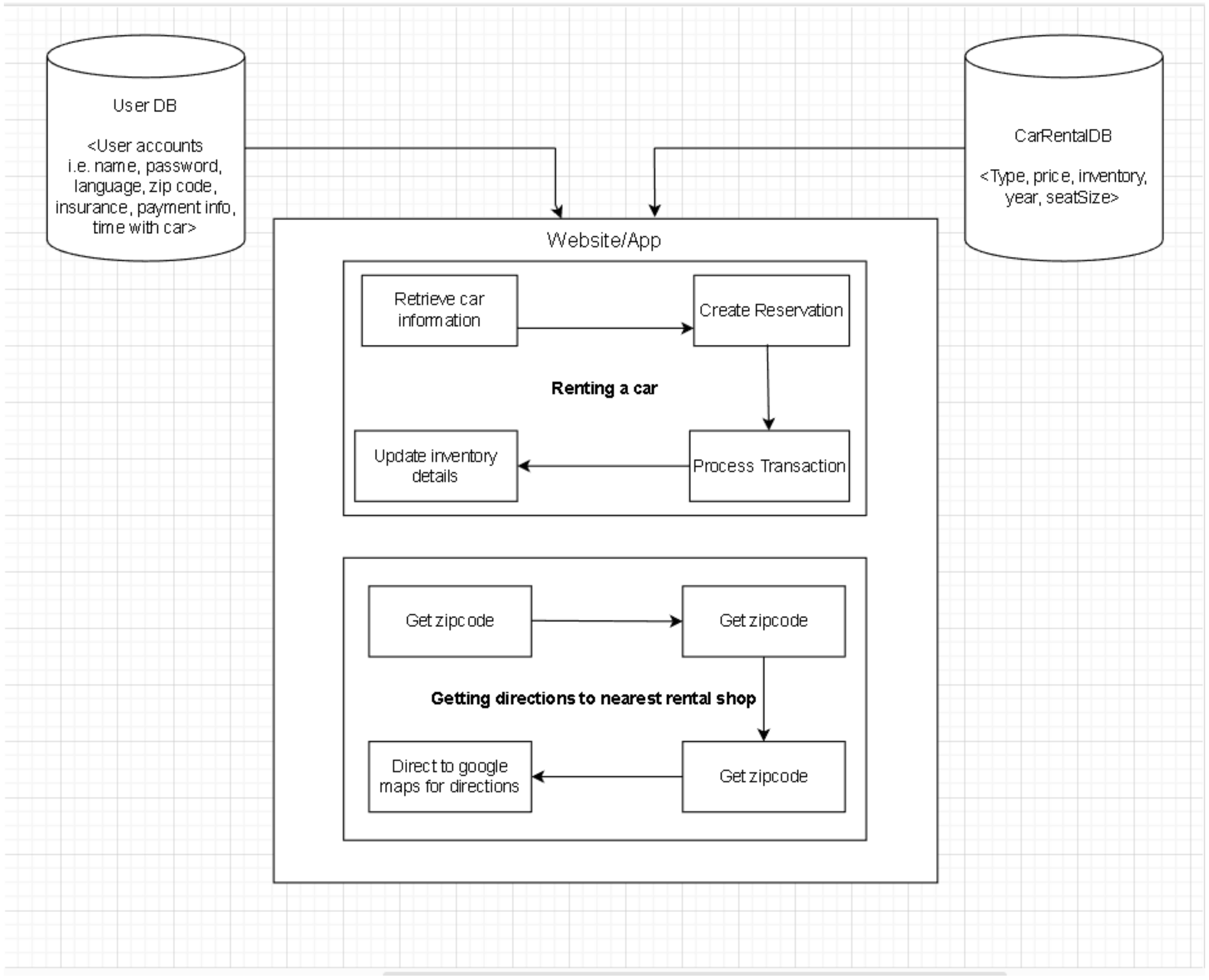
Overview: This diagram consists of 4 core classes regarding our car rental systems. These classes each rely on each other to ensure that we have an organized and also functional system. Each class has a specific commitment regarding the user using the system, the system itself, also backend work such as security. Ultimately, these four classes functioning amongst each other brings the best experience for a user using the system, and also feedback the system may provide back to the user.

The purpose of the security class is to ensure that users do not have any sensitive information or data leaked. Users using our systems should never have a password set out to the public or other sensitive information that pertains to home addresses, full name, card information. This class has two specific functions to encrypt and decrypt strings. This class is not open to the public and is only provided access to BeAvil admins only. Admins will not be able to change or edit the classes as they are private, and also for security reasons, functions come with their own unique keys to encrypt and decrypt. The software itself will call the encrypt and decrypt functions as the user is signing up for an account, as they are required to provide card information, home addresses, a full name and passwords too.

The User functionality consists of getter and setter methods so that information can be saved, and also edited at any time a user desires. It is very possible that a user would want to update their name or address or payment information, so the user should never be restricted from doing so. The setters of the user class is what allows the user to update their personal information. The getters are to ensure that the users are logged into the right account, or if they want to view what payment information they have on file. In basic, getters are for viewing purposes and setters are used for updating information.

The Car class is what allows BeAvil to present information to users about the cars. This information includes the type of car, price, size, and year. The ability to add and remove cars from a retailer is also necessary because the system should never say that a car is available when it is not and vice versa. The Car class is also provided with a variety of getters and setters so that information about the car itself can be updated, or viewed.

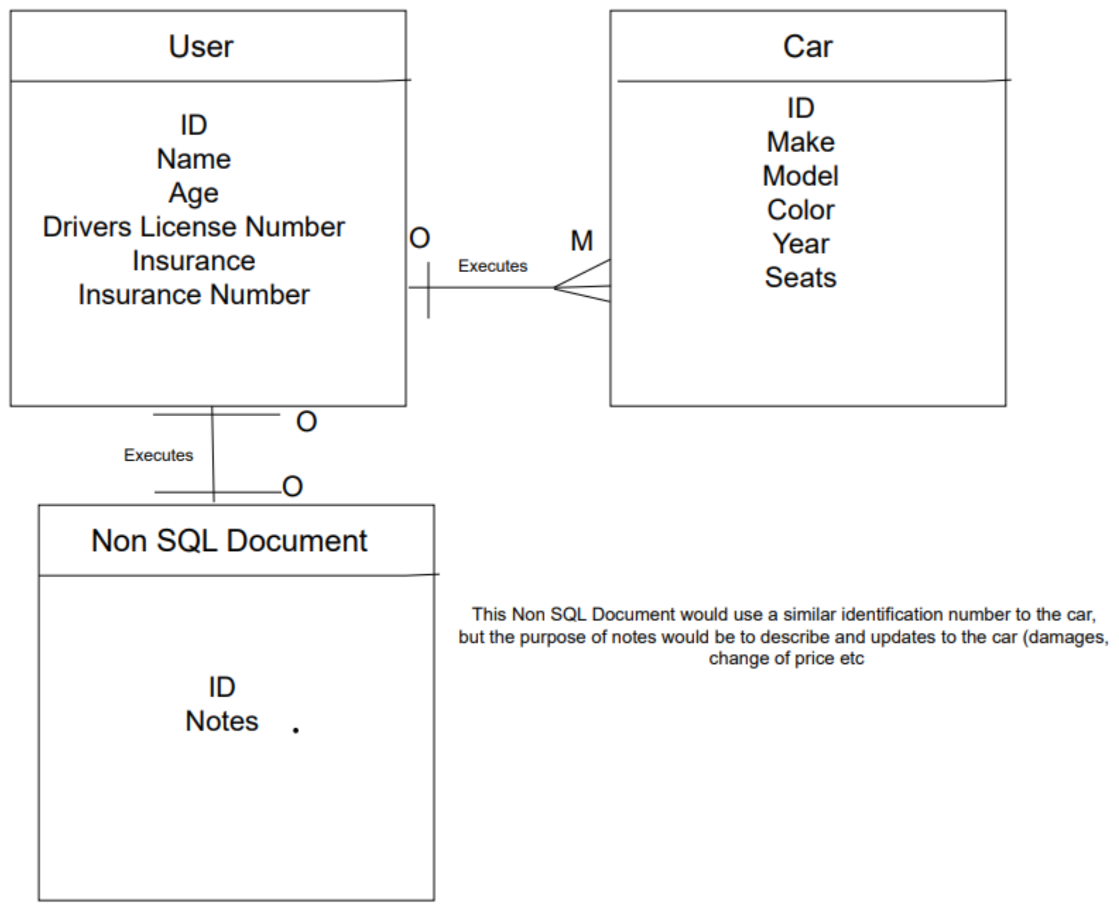
The Reservation class functions as a way to keep the rental process organized. The functions provided in the reservation allow the user to see when the car was rented out and is also provided a specific rental ID so they can view more information about their rental. Users should never have the same rental ID so a new one is generated every time a car is reserved. Getters and setters are also provided so that admin can update or view a rental ID. A cancellation fee is also included in the reservation so that if a user chooses to stop their reservation, then they would be charged a cancellation fee, which our system provides a function for that.


User DB
<User accounts i.e. name, password, language, zip code, insurance, payment info, time with car>
CarRentalDB
<Type, price, inventory, year, seatSize>
Website/App
Retrieve car information
Create Reservation
Renting a car
Update inventory details
Process Transaction
Get zipcode
Get zipcode
Getting directions to nearest rental shop
Direct to google maps for directions
Get zipcode

Software Architecture Design Description:

Our software architecture design is oriented around our databases. We split our database into two parts– userDB and CarRentalDB. This made our design more simple because it separated user information from inventory information. Both our website and our app rely on both of these databases. The website and app design has two main functions– “Renting a car” and “Getting directions to nearest rental shop.” These two functions are shown in more detail in the diagram. Each function consists of smaller functions that utilize the databases to achieve a certain goal.

Data Management Strategy


User
ID
Name
Age
Drivers License Number
Insurance
Insurance Number
Car
ID
Make
Model
Color
Year
Seats
O
Executes
M
O
Executes
O
Non SQL Document
ID
Notes
This Non SQL Document would use a similar identification number to the car, but the purpose of notes would be to describe and updates to the car (damages, change of price etc

Our management strategy for our data revolves around splitting up the databases by how we plan on using them. The two main databases that we need are called “User” and “Car”. We have a ‘one to many’ connection from User to Car. We had to connect the databases in this way because of how our App/Website’s functionality works. A user will log onto our site and be able to rent a car. Our app works by creating user objects that can rent cars and multiple car attributes can be attached to each user object. Since our app works in this way, it makes more sense to set up our database as seen in the picture above.

One alternative way that we could have set up our database is to have more databases with less information in each. For example, we could have taken some of the data out of Car and put it into a separate database in order to reduce the chances of overwhelming the searching algorithm. We could have also set up the databases in terms of size rather than similarity; so instead of having Car and User we would have had 2 or 3 randomly named databases with equal size. This setup could have helped us by balancing the time it takes to search each database.

Tradeoff Discussion

BeAvil's car rental systems will rely on two primary databases one consisting of users and another consisting of cars. These two databases are SQL databases. The reasoning behind the use of a SQL user database is that in an SQL we could define tables with rows and columns pertaining to the information stored in our database. These columns would consist of ID, Name, Age, Drivers license number, etc, which are all reflected in our data management diagram. Furthermore, SQL can be used to perform queries based on our user database allowing admin profiles using our systems to search for a specific group of people, update any information unique to the user or even gather information on user activity as well. Similarly, our car database is also an SQL database as well. We could divide up certain sections that range from make, ID, year, size, etc into individual columns. This will allow us to keep organized and track cars in our systems. Admins using our systems could also easily search for cars in the car database and update information if needed as well. An instance of a SQL query for our car database would be to look at a car's specific make and model and update that to our systems. Lastly, our NoSQL structures would be documentation. Documentation related to a car could come in text, images, videos, etc. Since we would have to handle so much volume for car documentation, it would not make sense for our systems to give it a unique row and column. However, BeAvil would use IDs from SQL databases in order to provide said documentation to each vehicle.

Development Plan and Timeline

Total Project Time: 3 months

First month: Kyle and Andres will work on completing the code for the website while Jonathan completes the code for the app.

Second month: This entire month will be dedicated to carrying out our test plan. As a team, each of us will try to break our code by testing it. We will spend one week completing our unit tests, one week completing our functional tests, and two weeks completing our system tests.

Third month: We will spend the first week of this month reflecting on what we have and deciding how to spend the last bit of project time. If we do not have any problems to fix, then each of us will be designated certain parts of code to optimize. Kyle and Andres will optimize the website's code while Jonathan optimizes the app's code.

TEST PLAN

Unit Test 1: ‘Encrypt’ function from Security class

Input: string – “BadPassword!1”

Example function call: Encrypt(BadPassword!1)

Expected Output: CbeQbttxpse!1

-The purpose of this function is to encrypt our data as it is stored in a database. Knowing this, our test will consist of inputting an example of “data” into the function and seeing if it will properly encrypt the data. We will use a caesar cipher(ROT1) to encrypt the data. If our function works properly, we can see if the location in memory matches our “expected output” as seen above.

Unit Test 2: ‘addToInventory’ function from Car class

Input: object – 2010 Mustang GTR

Example function call: addToInventory(Mustang)

Expected Output: none

-The purpose of this function is to add a car to our inventory database. Each car will be stored as an object so that we can easily access its features with other functions from the Car class like “getYear” or “getModel.” Our test will be to input an object named “Mustang” along with model, year, and other attributes. There should not be an output. We will know if the function worked if we can go into our database and see if Mustang has been entered into it. Additionally, we can see if its attributes have been properly added by trying to access them. After accessing the attributes, we will see if they are correct or not.

Functional Test 1:

First functional test will be using the Reservation and User Classes.

In User Class - setPaymentInfo(1234123412341234);

This set method sets the cardNumber variable in the User Class to that specific user’s payment information.

In Reservation Class - finalizePayment(cardNumber);

This function returns a confirmation message with “1234123412341234” in the message to show that the two functions do align correctly and function.

<u>Functional Test 2:</u>
The second functional test would consist of a User and Security class. The user is creating an account alongside, in order to prevent sensitive information from being leaked, we will encrypt our strings.

Here is an example of a user account being created with the user class

Input - name: String
Output: None
setName(“John Smith”);
Encrypt(“John Smith”); // from security class

Input - password: String
Output: None
setPassword(“John123!”);
Encrypt(“John123!”); // from security class

Input - Language: String
Output: None
setLanguage(“English”);

Input - card information: Integer
Output: None
setPaymentInfo(1234123412341234);
Encrypt(1234123412341234); // from security class

Input - Zip Code: Integer
Output: None
setZip(92103);
Encrypt(92013); // from security class

Input - Identification code for the user: Integer
Output: None
setID(12345);
Encrypt(12345); // from security class

Input - Insurance Type: String
Output: None
setID(“Geico);
Encrypt(“Geico”); // from security class

All of the functions listed above are derived from the users class. A user will be prompted to put this information upon creating an account. Afterwards all of these functions will be sent to the Security class in order to ensure that the information provided is encrypted.

Input: A string value - “John Smith”
Output: John Smith but encrypted with caesar cipher
Encrypt(string);
 //John Smith -> Kpio Tnjui (Shifted using ROT1)

Input: An integer value - 1234
Output: encrypted by encoding it with Base64 encryption. Comes in string format.
Encrypt(int);
 //1234 -> MTIzNA==

Input - A string value (encrypted): Kpio Tnjui
Output - decrypts with caesar cipher, reversed
Decrypt(string);

// Kpio Tnjui -> John Smith
// MTIzNA== -> 1234

The decrypt method checks for a match with the original value. Of course, m decrypted with the caesar cipher will not give us our password, so it checks whether it needs the caesar cipher decryption, or base64 decryption.

System Test 1:
First system test will be using all 4 classes and show how the car rental process works in general. This test will show how information is exchanged/carried over from each of our classes.

For starters, if the user hasn't created the account they would be encouraged to do so. Alongside the users class, we would also use the security class to encrypt important information.

//Users class
Input - name: String
Output: None
setName("John Smith");
Encrypt("John Smith"); // from security class

Input - password: String
Output: None
setPassword("John123!");
Encrypt("John123!"); // from security class

Input - Language: String
Output: None
setLanguage("English");

Input - card information: Integer
Output: None
setPaymentInfo(1234 1234 1234 1234);
Encrypt(1234 1234 1234 1234); // from security class

Input - Zip Code: Integer
Output: None
setZip(92103);
Encrypt(92013); // from security class

Input - Identification code for the user: Integer
Output: None
setID(12345);
Encrypt(12345); // from security class

Input - Insurance Type: String
Output: None
setID(“Geico);
Encrypt(“Geico”); // from security class

When searching for a vehicle, many functions from our car class would be called, as the car class carries information such as the type, inventory, size, year and other general information about the vehicle itself.

//Car class

function that displays the different cars in the inventory
Input - None
Output - Inventory of cars with price

```
getInventory()
  //print car types and price:
  // 2016 Honda Civic $30
  //2022 Ford F150 $40
  //2019 Hyundai Elantra $50
 //print(getYear() + getType() + getPrice() + setVin(12345ABC))
```

The getInventory will also call other subclasses such as getPrice, getYear, getType, and setVin. Our sample vin would be 12345. Our subfunctions like getYear and getType are just getters and setters our car class has. In main we would have a car type created with the following information.

```
//main
Car Hyundai = new Car(2016, Hyundai, Elantra, 30, 12345)
```

An array of cars could also be created for similar type cars.

```
//main
Car[] Hyundai = (new Car (2023, Hyundai, Kona, 40, 7890), new Car (2017,
Hyundai, 30, 12345))
```

Once a car has been successfully reserved, the remove function from main could be called

```
//main
Car.removeFromInventory(Hyundai[1])
```

Afterward, the reservation class is what allows the user to ensure that they have a reservation with the BeAvil system. One function that the reservation class uses is a makeReservation function that takes the vin number from the car class and successfully makes a reservation with our system. A date is then created to allow the user to know when the car is reserved.

```
//Reservation class
```

Input - vin number from car class
Output - None

```
//Makes reservation for the user
makeReservation(12345)
   //setDate("March 23 2023"); (sub function called to get the current date)
   //setCancelFee(); (sub function called to set a cancellation fee if the user
choose to cancel).
```

Input - Date
Output - None

```
setDate("March 23 2023");
```

Input - None
Output - None

```
// A boolean value is in place, if the cancelFee variable is true, it will then
be added to the final payment
setCancelFee();
```

Afterward, the finalized payment information will be called, It would take the price from the Car class and then use the payment information saved in the user class, and also the VIN number to match the car

Input - Vin number
Output - Total price

```
finalizePayment(vin);
  //total = 0 (placeholder value to add the price)
  // total += getPrice();
  // if(cancel == true)
        //total += 5;
 //getPaymentInfo();
return total;
```

<u>System Test 2:</u>
A stress test to see what would happen to our system if multiple users created an account at the same time, comes up with theoretical latency. The software system has a specific database that holds information about the user ranging from user name, password, zip code, payment etc.

Creating multiple accounts in main would look something amongst the lines of:

```
//main
Users[] user;

user[n] = new User("Full name", "Password", Payment info (int),
"Language", zip (int), "Insurance", ID (int))
```

Our systems can not simultaneously create accounts at the same time coming from one main or driver, in this case we would have to use a separate database or drivers to allow this.

```
//main1
Users[] user;

user[1] = new User("Jamie Burns", "JBurns123", 22, "English", 92375,
"Allstate", 9443)

//main2
Users[] user;

user[2] = new User("Abe Jones", "AbeCars34",1234 5678 9101 1213,
"Korean", 91910, "Progressive", 5712)
```

```
//main3
Users[] user;

user[2] = new User(“Michael Jackson”, “MJ1234?!”, 5678 5678 5678 5678,
“Spanish”, 94103, “StateFarm”, 9212)
```

No expected output so far

Each time we make a new user, the set functions for each of these attributes will be called. The setPassword function will automatically call the encrypt function, so for each user, we will have an expected output as a result of the encrypt function.

User[1]
Input for Encrypt: JBurns123
Example function call: Encrypt(JBurns123)
Expected Output: KCvsot123

User[2]
Input for Encrypt: AbeCars34
Example function call: Encrypt(AbeCars34)
Expected Output: BcfDbst34

User[3]
Input for Encrypt: MJ1234?!
Example function call: Encrypt(MJ1234?!)
Expected Output: NK1234?!

Having this format will allow our theoretical latency to become faster, if we relied on one main file to create accounts simultaneously, the timing to do so would become slower. At the same time, we do not want too many servers/driver files creating accounts, the system could be overwhelmed causing a crash. There is a risk between the two either being too slow, or too much. Suppose we set our theoretical latency to something our systems could handle while creating accounts at a reasonable pace, without overwhelming our system, that could be done correctly. Referring to our software architecture diagram, we have separate databases for this reason so our systems do not crash each other. Our software systems rely on two separate databases for users and cars so that we can maintain our latency.