BeAvis Software Design Specification

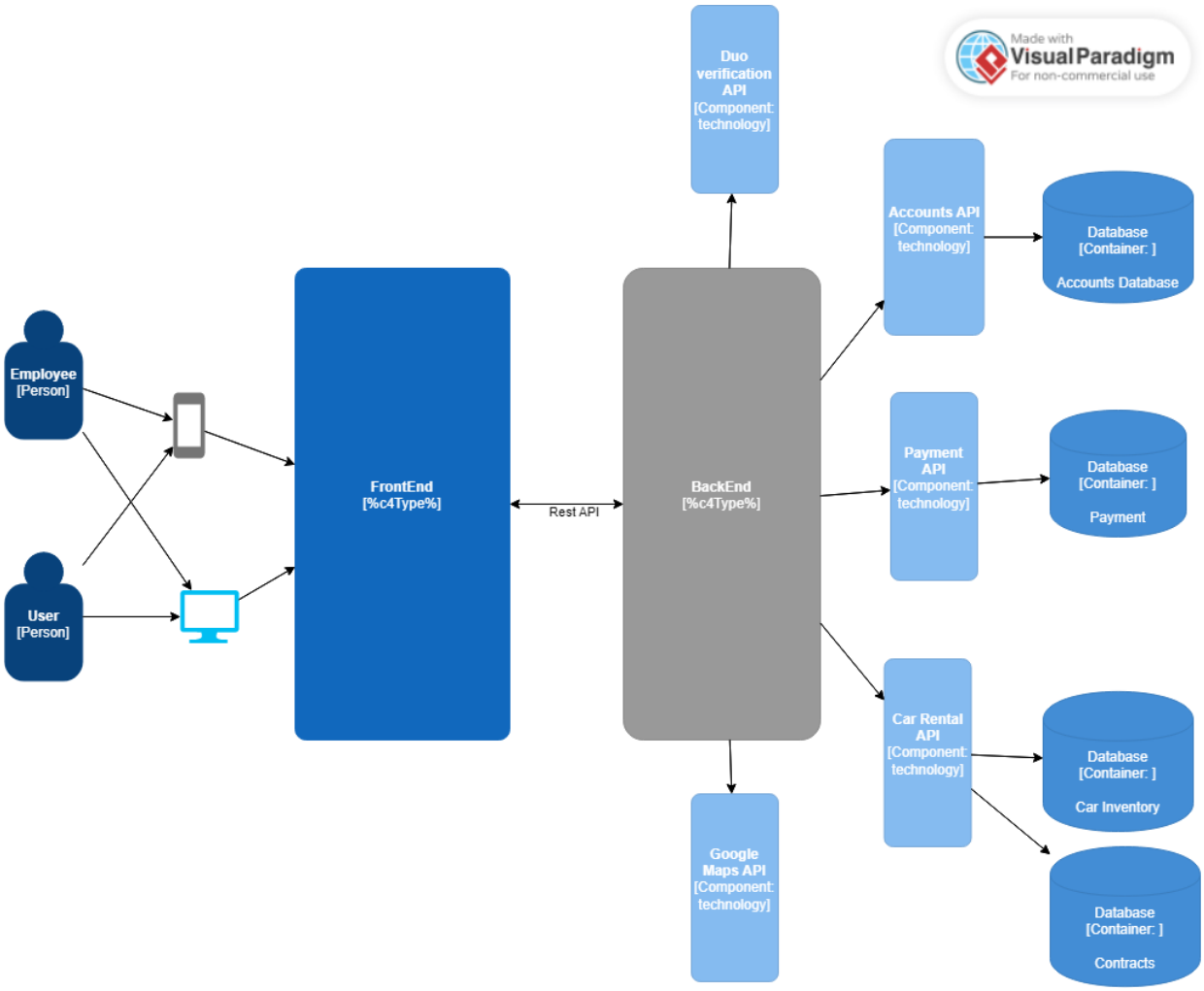
Prepared by: Pranav Nambiar, Jake Kraemer, Dennis Baltazar

1 System Description:

BeAvis has requested a car rental software system that will replace their existing paper system. This system is needed because BeAvis's paper system is outdated and cannot keep up with the increase in demand.The new software system will streamline the process of renting a car for the customer, while allowing employees to manage the rental car company. This document is organized into four sections: System Description, Software Architecture Overview, Development Plan, Timeline and Verification Test Plan. These four sections will specify the design and testing for the BeAvis software system.

2 Software Architecture Overview:

BeAvis Software Architecture Diagram:



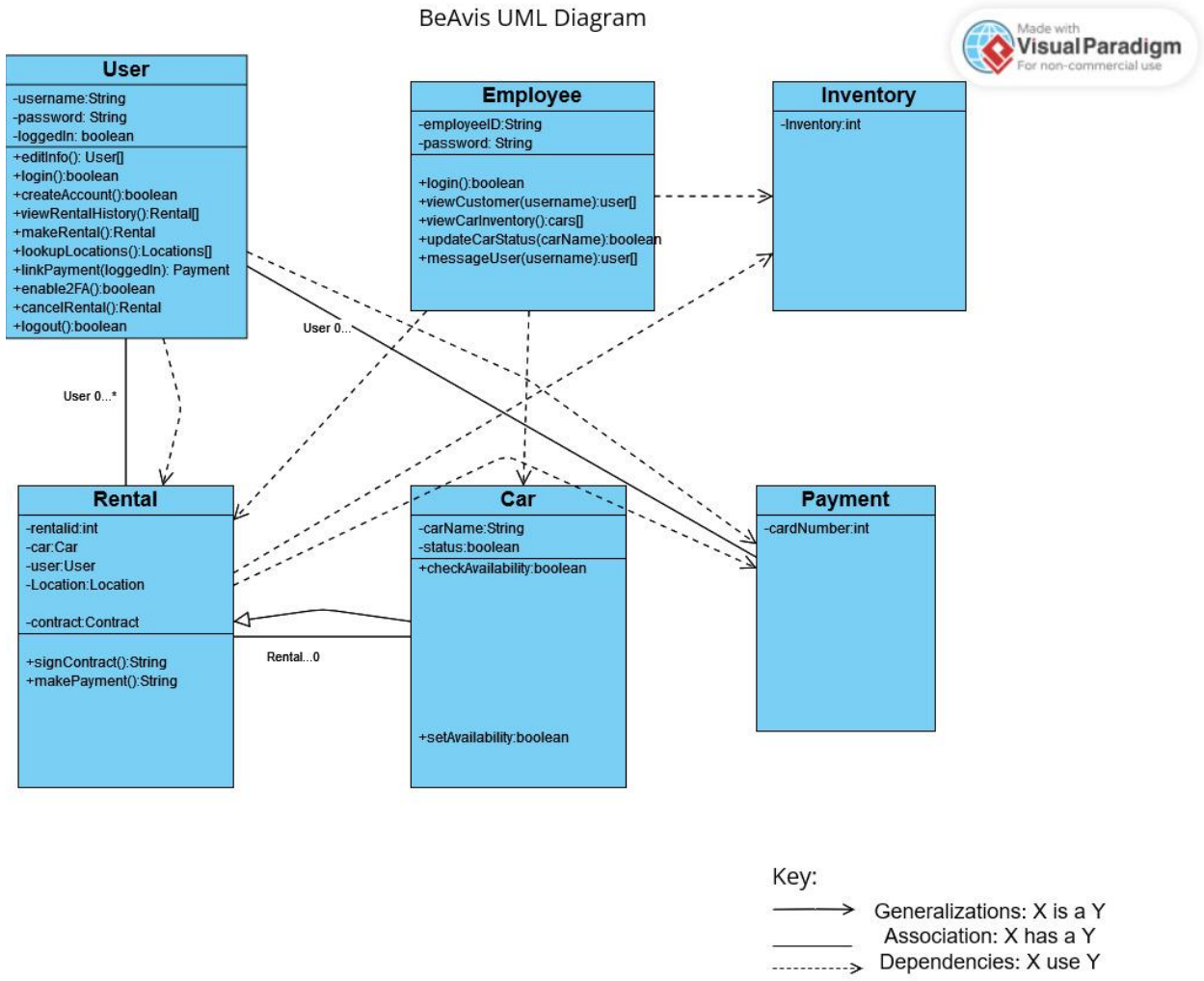
BeAvis Software Architecture Diagram description:

The BeAvis Software System replaces the old paper system by using state of the art software infrustructure to streamline business needs. We focused the software architecture on the requirements given on the software. The architecture addresses the needs of a fully functioning online car rental system allowing a user to use their location to safely make a binding car rental contract.

The backend of the architecture relies on four core databases that store all crucial information needed to meet customer demands. An accounts database keeps record of all needed information for a user to create an account and login to our application. The information can be accessed by our applications through our accounts API which aggregates data such as username, password, and employeeID using service classes that clean and send data from this server to usable endpoints. The Payment Database contains information that is needed to complete a payment to rent a car. The payment database is separated from the accounts database to be able to regulate clients' most private information. All information regarding car inventory and contracts at the core of the BeAvis car rental model is handled by the car rental API. The Car Rental API is the core of the BeAvis software system providing all information needed to be able to find a car in the user's location and to ensure that a binding contract for insurance purposes is in place. The Car Rental API will rely on two databases. The Car Inventory database will draw crucial information such as the user location, car name, availability, and rentalID to properly handle and distribute data needed to service our car rental application. Data regarding contracts are kept in a separate database from inventory to have a secure way to audit information related to contracts held by users.

To implement the BeAvis software the devs would recommend using a .NET Core Framework. A .NET core framework would allow BeAvis to host the needed API endpoints securely to any operating system. Through a .NET Core framework we could derive data crucial to the functionality of our application in API endpoints using dependency injection. The BeAvis software system will require a ton of interaction between databases to be able to verify information such as rentalId, location, user information all confirm that a user is purchasing a rental. Dependency injection passes objects like car and user to the API at runtime of a given location. This allows the software to be adaptable and testable as developers and test engineers can easily mock test car rentals. The service classes described would use ADO.NET libraries for data access management to provide data to our service classes. This ADO.NET infrastructure allows us to create safe connections between the data that drives our application and our users. A .NET framework would be free to implement and would allow our application to be coded in a variety of languages. Applications created in a .NET framework can run on all of the main operating systems including iOS and Android using cross platform development tools.

The backend of the software also relies on third party location and multi factor authentication systems to verify the location and the identity of the user. The front end of the software hosts a web server that handles https requests and responses. The front end has two separate user interfaces for the web and mobile application that have separate UI to get the information needed from the user for the application to run.



BeAvis UML class description:

The objects of the BeAvis system are designed to represent all interacting parts of the car rental process. When a user wants to rent a car a user object is created to keep track of the username and password of the user so they can access previous car rentals. An employee object is separate from the user so that an employee can log in to the application using an employeeID and password. While the user can lookup cars,

make a rental, and lookup nearby rentals an employee objects main functions are to view and manage car inventory requiring a separate object to perform the secure functions. For a user to pay for a rental a separate payment object is created to store credit card information for a user required to complete a payment. A rental object depends on inventory and payment objects to be able to create contracts that bind a user to a car. The cra object stores all information about a rental car needed to complete a rental.

3 Development Plan and Timeline:

The BeAvis rental car system application is expected to be completed in four months using both the phone and website platforms. The work will be divided between different development teams, who will start work immediately and simultaneously. We have attached a list of the tasks.

We will not be developing the payment system in-house. Instead, we will be purchasing a software system to integrate with our application. The payment system of choice will be Stripe, which is widely used and efficient.

Our front-end team will be responsible for designing the user interface for the app and web platform. They have a good grasp of current trends and will ensure the interface has a clean, modern look while remaining user-friendly.

The backend for the system will be developed by our backend development team, led by Joe, along with a few interns. This will provide a good learning opportunity for our interns. They will also be integrating cloud services such as AWS.

Updates to the system will be handled by our backend development team as well if needed. This includes fixing any bugs that occur as well as adding additional features that BeAvis may want to add in the future.

Updates to the interface of both web and mobile applications will be handled by our front-end team at the request of BeAvis.

List of Tasks:

1. Create required databases (**4 Months Hard Deadline**)

2. Create APIs for respective databases (**4 Months Hard Deadline**)
3. Develop backend connection to APIs for databases/GPS/Duo (**1 week Hard Deadline after development**)
4. Develop Web/Mobile UI (front end) (1 Month Soft Deadline / **2 Months Hard Deadline**)
5. Connect frontend to backend with a REST API (**1 week Hard Deadline after development of both**)
6. Update app after development (check-ins every month, and any requests from BeAvis)

4 Verification Test Plan:

1. **System Tests:**
   The goal of system testing is to test the entire system to ensure that it works from end to end. In order to do this we tested if users are able to make a rental and cancel their rental. In order to make a rental they have to login,make a rental,look up the location,link payment,sign contract,make payment. If all of these are valid then the user can make a rental. In order to cancel rental the user has to login,look up the location, link payment, and then cancel rental. In both cases if the user fails in one part i.e. linkPayment, they are unable to complete the process.
   a. Make a Rental
      i. Input: Login()→ makeRental()→ lookupLocations()→ linkPayment()→ signContract()→ makePayment().
      Output:
      Rental Purchased!
      ii. Input:
      Login()→ lookupLocations()→ makeRental()→ !linkPayment()→ signContract()→ makePayment().
      Output:
      Rental purchase failed.
   b. Cancel Rental
      i. Input:
      Login()→ lookupLocations()->linkPayment()->cancelRental()
      Output:
      Rental Canceled!
      ii. Input:
      Login→ lookupLocations()->!linkPayment()->cancelRental()
      Output:
      Rental Cancel failed.

2. **Integration Tests:**
   The goal of testing integration is to ensure that inferences among integrated units work as intended. The update car status method in the employee class uses the car class to allow an employee to update the status of a car. The method takes in a car name and outputs a boolean status that updates the car class. The most obvious test cases for this method is a car name string included in our inventory and a car name string that is not in our inventory. These tests cover expected outputs that change the status from available to unavailable. To test edge cases we include a test for an empty string which might return an unexpected output but is expected to throw an appropriate error. We also included a test for a car name that was once included in the inventory to ensure that our inventory is updated. The expected output should be an error that shows the car was once in our inventory but no longer is. The link payment function is to link a credit card number from the payment class to the user class when the user is making a payment. The method will only accept two inputs true or false reflecting if the user is logged in or not. If the user is logged in the car number will be returned. If the user is not logged in the car number will not be given to the user and the payment will fail.
   a. Employee updateCarStatus(carName): boolean
      i. Input: Valid car name in entered [carName: 'Toyota Camry']
         Output: If car status is false (not available) → car status becomes true (available) [status: False] → [status: True]
         If car status is true(available) → car status becomes false (not available) [status: True] → [status: False]
      ii. Input: Invalid car name entered [carName: 'Ford Pinto']
         Output: Car status is not changed. An error is thrown to show the car name is not in inventory.
      iii. Input: No carName given [carName: '']
         Output: Car status is not changed. An error is thrown to show no car name is not given.
      iv. Input: Once valid car name that is no longer valid [carName: 'Ford Bronco']
         Output: Car status is not changed. An error is thrown to show the car name is no longer valid.
   b. User linkPayment(loggedIn): Payment
      i. Input: User is logged in [loggedIn: True']

Output: Card number linked to username and password [Payment.cardNumber: int]

ii. Input: User is not logged in [loggedIn: ‘False’]
Output: No car number is given. Error is thrown to show the user is not logged in.

3. **Unit Tests:**

a. createAccount():

If all inputs valid, account is created (true)

1. **Enter name:**
   Input: Tom Hanks
   Output: Valid name
   Create username:
   Input: U$ername
   Output: Invalid username, username can not contain special characters
   Create password:
   Input: Beavis123
   Output: Invalid password, password must include at least one upper case and lower case letter, one number, and one special character.
   Enter email address:
   Input: beavisemail123@gmail.com
   Output: Valid email
2. **Enter name:**
   Input: Tom Hanks
   Output: Valid name
   Create username:
   Input: Username
   Output: Valid username
   Create password:
   Input: Beavi$123
   Output:Valid password
   Enter email address:
   Input: beavisemail123@gmail.com
   Output: Valid email

b. enable2FA():

If input is valid, 2fa is activated (true)

1. **Enter Phone number:**

Input:123-456-7890

Output: Enter the code sent to your phone

Enter verification code:

Input: 6056

Output: Code invalid, please try again.

2. **Enter Phone number:**

   Input:123-456-7890

   Output: Enter the code sent to your phone

   Enter verification code:

   Input: 7665

   Output: Code valid, 2FA activated