Software Requirements Specification for BeAvis Car Rental System

Prepared by: Arman Atwal, Sahib Singh, Catherine Dang

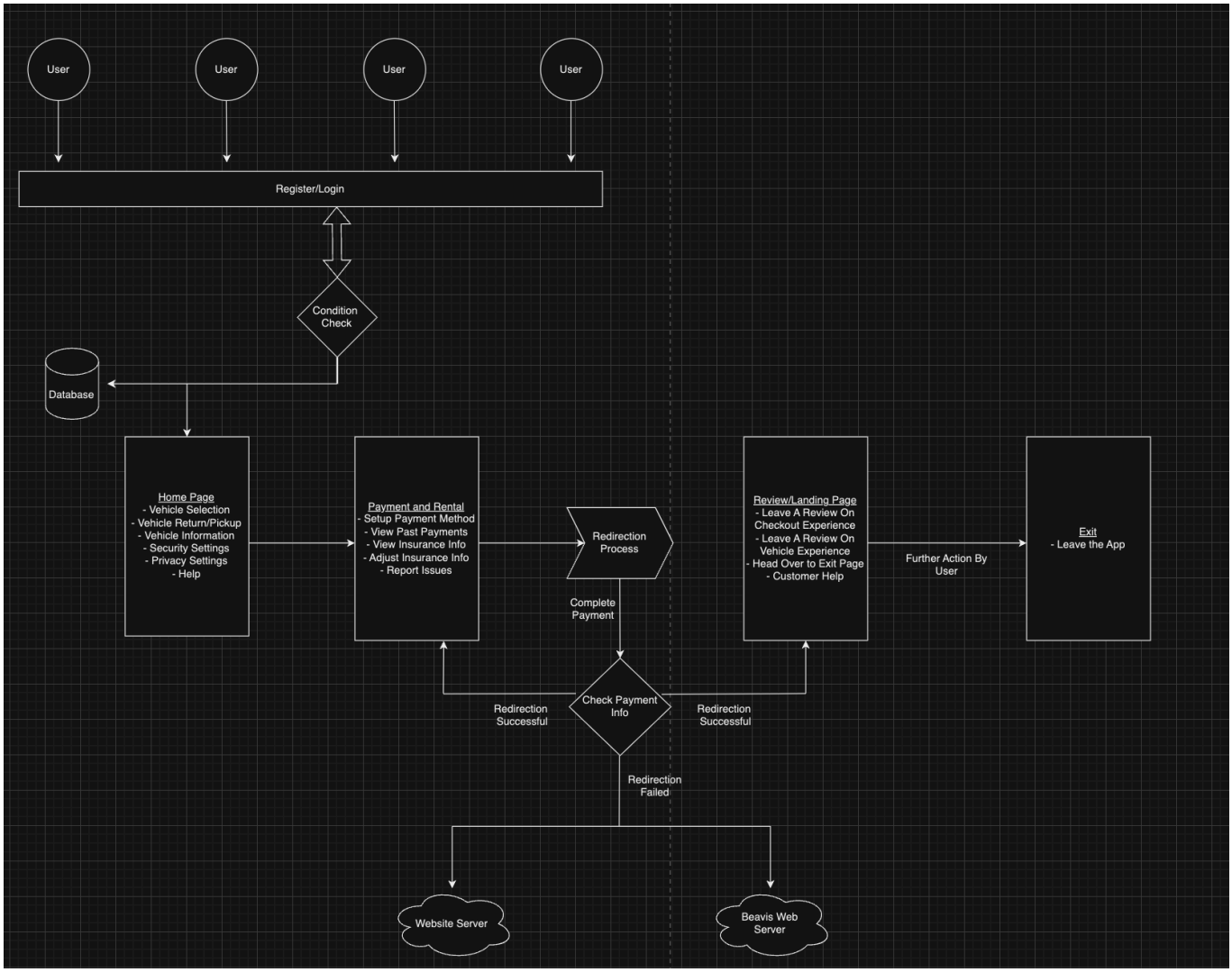
System Description

The BeAvis Car Rental System is a cutting-edge system that has been painstakingly designed to transform the experience of renting a car. It serves two main user groups: clients looking for a hassle-free car rental experience and personnel tasked with managing rental operations effectively. At the heart of our design philosophy is accessibility, which guarantees that users may interact with the system easily through mobile applications compatible with both iOS and Android, as well as web browsers on various computer systems. Our persistent dedication to user friendliness is shown in a feature-rich environment that includes user account setup, a simple vehicle selection procedure, a thorough presentation of vehicle information, and a strong and secure payment processing structure.

Compliance with rules and regulations, particularly those relating to data protection, is crucial. The system provides optional two-factor authentication, a strong protection for user accounts, to bolster security. Additionally, it accepts a variety of payment options, guaranteeing the absolute separation of user data and delicate payment information. Real-time responsiveness is essential, and the system is designed to respond quickly to user queries and provide real-time updates on car availability. With the potential to support up to 15,000 concurrent users and 5,000 concurrent rentals, scalability has been carefully examined, allowing a smooth expansion in tandem with business growth.

The BeAvis Car Rental System's technology stack has been carefully chosen to maximize development productivity and operational effectiveness. The system, which is poised to establish itself as an industry leader in the brutally competitive automotive rental business, essentially reflects a commitment to efficiency, tight security, and sustained revenue growth.

Software Architecture Overview



Description of SWA for Users:

The Software Architecture Diagram for the BeAvis system provides a clear and concise representation of how users interact with the system and the various components involved in the process. It effectively outlines the user journey from registration or login to completing a rental and leaving reviews. Here's a brief breakdown of the components and flow in our diagram:

User Registration/Login:

- Users start their journey by either registering for a new account or logging into an existing one.

Database:

- Successful login or registration grants users access to the database, where their account information is stored.

Home Page:

- Once authenticated, users are directed to the home page, which serves as the central hub for various actions and services.
- Home Page Components:
    - Vehicle Selection
    - Vehicle Return/Pickup
    - Vehicle Information
    - Security Settings
    - Privacy Settings
    - Help

Payment Details/Rental:

- From the home page, users can initiate the rental process, including setting up payment methods, viewing past payments, accessing insurance information, adjusting insurance details, and reporting issues.

Redirection Process:

- After setting up payment details, users are redirected to complete their payment. At this stage, the system verifies the payment information provided by the user.
- If the payment information is incorrect, users are sent back to the payment page to make corrections.
- If the payment information is correct, users proceed to the review page.

Review Page:

- The review page allows users to provide feedback on their checkout experience, vehicle experience, and access customer help if needed.
- Review Page Components:
    - Leave A Review On Checkout Experience
    - Leave A Review On Vehicle Experience
    - Head Over to Exit Page
    - Customer Help

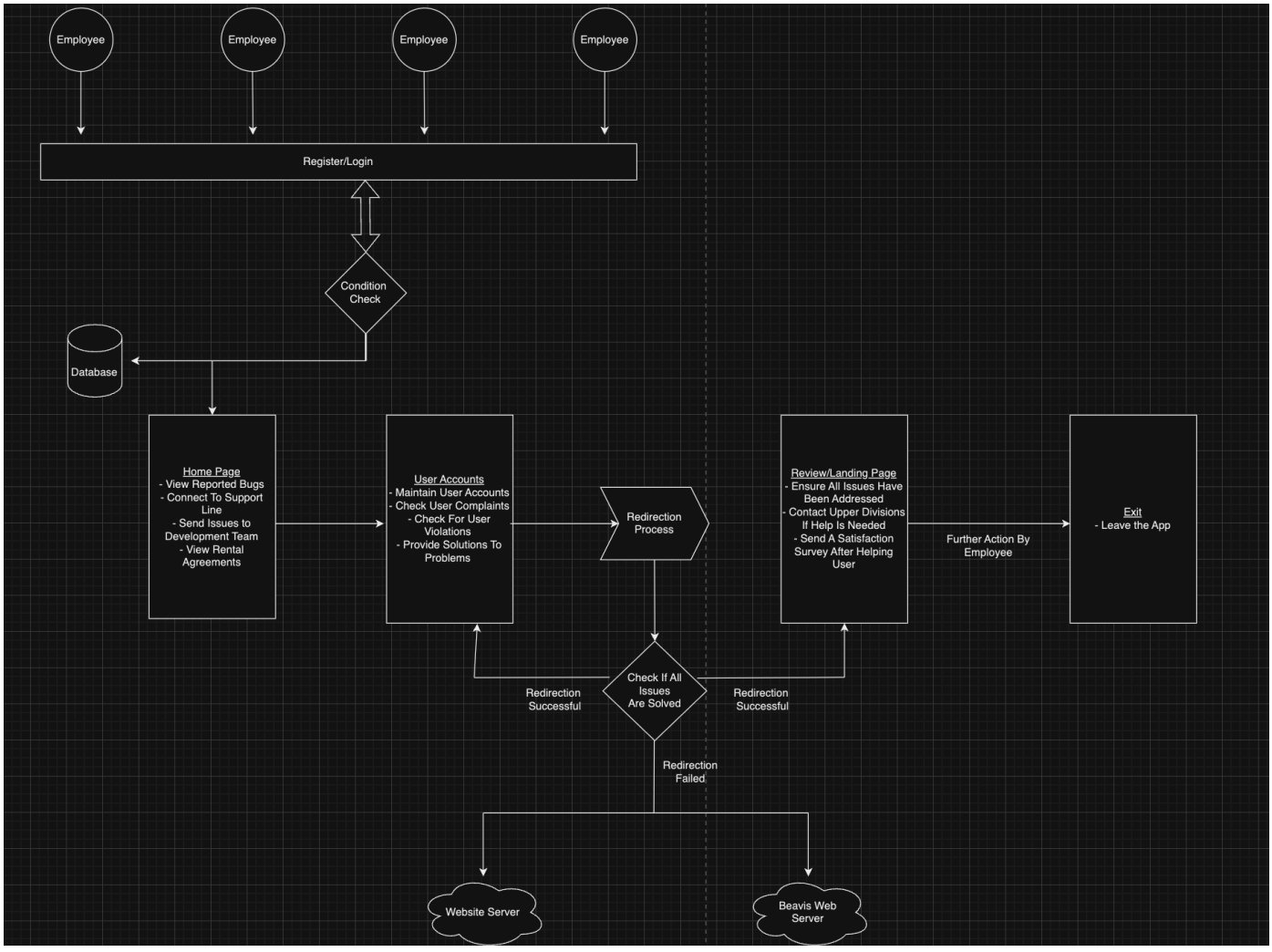
Exit Page:

- The exit page marks the conclusion of the user journey within the system.

Website Server and BeAvis Web Server:

- The entire process is connected to both the website server and the BeAvis web server, which handle the processing and communication required to facilitate user interactions.

Software Architecture Overview Continued



Description of SWA for Employees:

The Software Architecture Diagram for the BeAvis system provides a clear and organized visualization of how employees navigate and interact with the system. It effectively illustrates the flow of actions and decisions that employees can take within the system. Here's a breakdown of the components and flow we've described:

User Authentication Flow:

- Employees can either register for an account or log in to the system.
- Successful authentication grants access to the database and directs them to the home page.
- Unsuccessful login attempts prompt employees to retry the login process.

Home Page:

- The home page serves as the central dashboard for employees and provides quick access to key functionalities:
    - View Reported Bugs: Access to information about reported issues.
    - Connect To Support Line: A feature for immediate assistance.
    - Send Issues to Development Team: The ability to escalate issues to the development team.
    - View Rental Agreements: Access to rental agreements for reference.

User Accounts:

- Employees can navigate to the user accounts section, which encompasses various management tasks:
    - Maintain User Accounts: Employee access to user account management functions.
    - Check User Complaints: Monitoring and reviewing user complaints.
    - Check For User Violations: Reviewing any user violations or breaches of terms.
    - Provide Solutions To Problems: Resolving user issues and concerns.

Redirection Process:

- After interacting with user accounts, there is a redirection process based on the outcome:
    - If user issues are resolved, employees proceed to the review page.
    - If not, they are directed back to user accounts for further action.

Review Page:

- The review page focuses on ensuring the satisfactory resolution of user issues:
    - Ensure All Issues Have Been Addressed: Confirming that all user concerns are adequately resolved.
    - Contact Upper Divisions If Help Is Needed: The ability to escalate issues further.
    - Send A Satisfaction Survey After Helping User: Collecting feedback from users to gauge their satisfaction with the resolution.

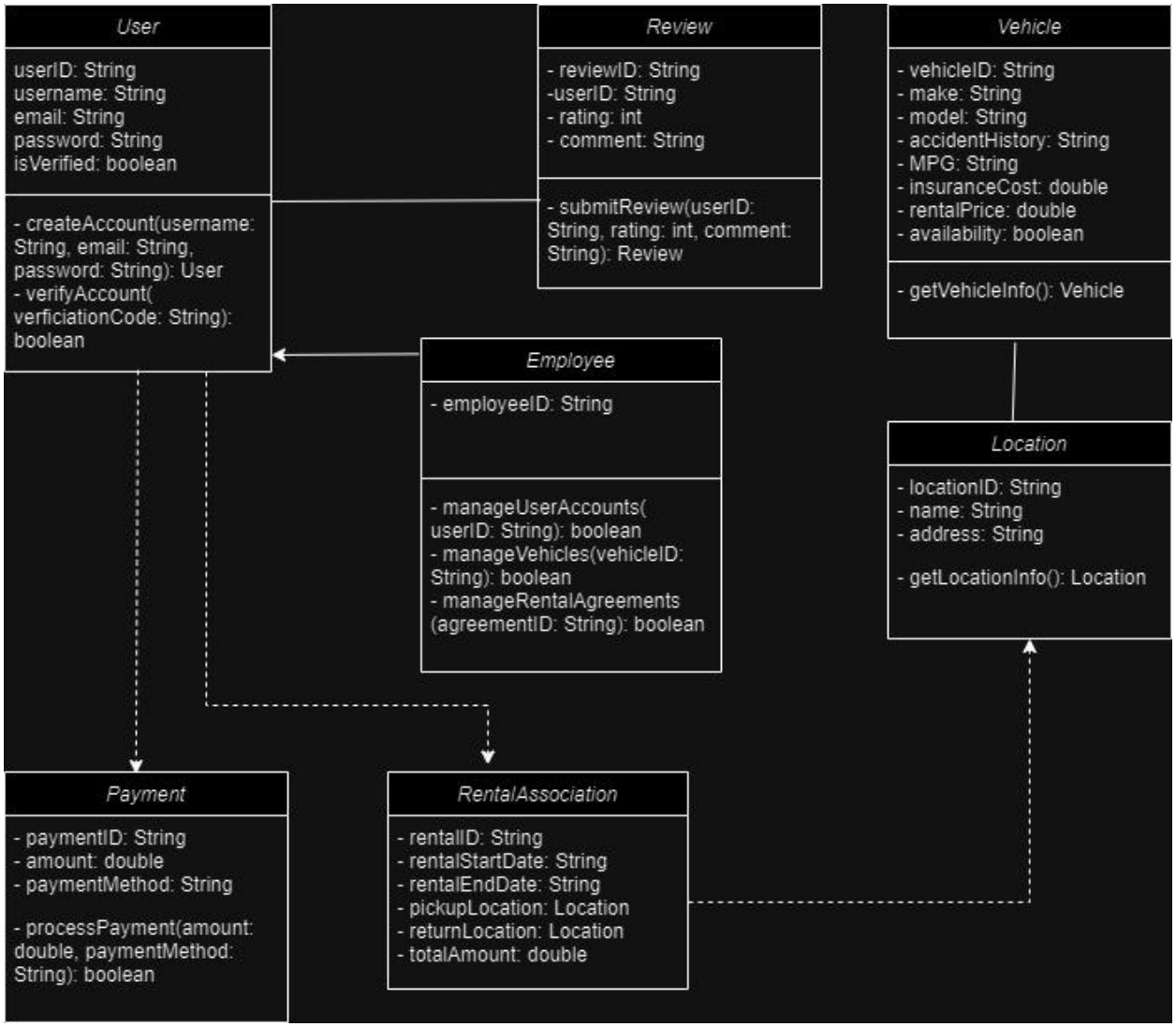
Exit Page:

- The exit page marks the conclusion of the employee's session within the system.

Server Connectivity:

- The entire process is connected to both the website server and the BeAvis web server, ensuring seamless data exchange and functionality

UML Class Diagram



Description of UML Class Diagram

The UML diagram for the BeAvis Car Rental System is designed to capture the system's structure and interactions, ensuring a streamlined user experience.

- The User Class serves as the foundation, which has the necessary attributes such as userID(String), username(String), email(String), password(String), and isVerified(boolean). Users engage with the system through defined function interfaces, such as createAccount(username: String, email: String, password: String): User, allowing the creation of accounts and verifyAccount(verificationCode: String): boolean for secure verification processes.

- The Vehicle Class contains the details of available rental cars, which includes properties like vehicleID(String), make(String), model(String), accidentHistory(String), MPG(String), insuranceCost(double), rentalPrice(double), and availability(boolean). Users interact with vehicles via functions like getVehicleInfo(): Vehicle, enabling them to retrieve information before making rental decisions.
- Rental locations are represented by the Location Class, featuring attributes like locationID(String), name(String), and address(String). The getLocationInfo(): Location function interface manages access to location details, helping users in choosing convenient pickup and return points.
- For smooth transactions, the Payment Class manages financial components, storing paymentID(String), amount(double), and paymentMethod(String). The system ensures secure payments through the processPayment(amount: double, paymentMethod: String): boolean function interface, where users initiate transactions with specified amounts and payment methods.
- User feedback is facilitated by the Review Class, which has attributes such as reviewID(String), userID(String), rating(int), and comment(String). Users can submit reviews via the submitReview(userID: String, rating: int, comment: String): Review function, enhancing transparency and trust within the system.
- The RentalAssociation Class manages the rental process, which includes details like rentalID(String), rentalStartDate(String), rentalEndDate(String), pickupLocation(Location), returnLocation(Location), and totalAmount(double). This class ensures a comprehensive record of each transaction, facilitating efficient management by both users and employees.
- The Employee Class, a subclass of User, includes specialized attributes such as employeeID(String). Employees perform administrative tasks via functions inherited from the User class, enabling them to manage user accounts, vehicle inventory, and rental agreements effectively.

Development Plan, Responsibilities, and Timeline

Project Duration: 12 months

Timeline:

Month 1-2: Project Initiation and Planning

- Define project scope, requirements, and objectives.
- Create a detailed project plan and timeline.
- Select the technology stack and tools.
- Set up the development environment and version control system.

Month 3-4: System Architecture and Backend Development

- Design the system architecture.
- Develop the backend infrastructure, including the server and database.
- Implement user authentication and registration.

Month 5-6: Frontend Development

- Create user interfaces for web, iOS, and Android.
- Implement the home page and its components.
- Develop the vehicle selection and payment setup features.

Month 7: Payment Processing and Integration

- Implement payment processing and integrate with payment gateways.
- Develop the redirection process for payment.
- Test payment flows and security.

Month 8-10: Employee Functionalities and Testing

- Develop employee-specific features for user account management.
- Implement issue resolution and escalation functionalities.
- Conduct thorough testing, including security and compliance checks.

Month 11: Final Testing and Quality Assurance

- Perform comprehensive testing, including load testing and security audits.
- Identify and address any remaining bugs and issues.
- Ensure compliance with data privacy laws.

Month 12: Deployment and Launch

- Set up production servers and databases.
- Deploy the system to production.
- Monitor system performance and address any post-launch issues.

- Launch the BeAvis Car Rental System to customers and employees.

Roles and Responsibilities:

Lead Developer/Architect (Programmer):

- Overall technical ownership of the project.
- Define the system's architecture, design patterns, and technology stack.
- Develop and oversee the implementation of critical system components.
- Ensure code quality, scalability, and security.
- Collaborate with other team members for integration and testing.

Frontend Developer (Programmer):

- Develop user interfaces for web and mobile applications (iOS and Android).
- Implement the user registration and login process.
- Create the home page and its components (vehicle selection, payment setup, etc.).
- Develop the review and exit pages.

Backend Developer (Programmer):

- Set up and maintain the server infrastructure.
- Implement the database schema and handle data storage.
- Develop APIs for user authentication and interaction with user accounts.
- Implement payment processing and integration with payment gateways.
- Create APIs for employee functionalities (user account management, issue resolution, etc.).

Database Administrator (DBA):

- Design and optimize the database schema.
- Ensure data integrity, security, and backup procedures.
- Optimize database performance for high concurrency.
- Collaborate with backend developers for data retrieval and storage.

Quality Assurance Engineer (QA):

- Create test plans, test cases, and test data.
- Perform manual and automated testing of the system.
- Identify and report bugs and issues.
- Ensure the system complies with legal and regulatory requirements.

DevOps Engineer:

- Set up continuous integration/continuous deployment (CI/CD) pipelines.
- Manage server deployment and scaling.
- Ensure system availability, monitoring, and logging.

- Collaborate with other developers to optimize performance.

Software Verification Test Plan

(Note: We didn't feel like we had to remake our UML diagram, so there is no new one)

Unit Testing

Test 1: Verify Account Creation (User Class - Create Account Function)

- **Objective:** Ensure the createAccount function in the User class correctly creates user accounts with valid inputs.
- **Test Description:**
    - **Test Case 1.1:** Create an account with valid input data.
        - Input:
            - Username: "TestUser1"
            - Email: "testuser1@example.com"
            - Password: "StrongP@ssw0rd"
        - Execute the createAccount function with the provided data.
        - Output:
            - The function should return a new User object.
            - The user data should be successfully stored in the database.
    - **Test Case 1.2:** Handling invalid input data.
        - Input:
            - Username: "InvalidUser"
            - Email: "invalid_email" (invalid email format)
            - Password: "weak" (weak password)
        - Execute the createAccount function with the provided data.
        - Output:
            - The function should raise appropriate error messages for invalid input.
            - No user account should be created in the database.
- **Expected Outcome:**
    - **Test Case 1.1:** The function should successfully create a user account, and a new User object should be returned. User data should be stored in the database.
    - **Test Case 1.2:** The function should handle and report errors for invalid input data. No user account should be created.

Test 2: Verify Payment Processing (Payment Class - Process Payment Function)

- **Objective:** Ensure the processPayment function in the Payment class correctly processes payments with valid payment methods.
- **Test Description:**
    - **Test Case 2.1:** Process a payment with a valid credit card.

        - Input:
            - Amount: $100
            - Payment Method: "Credit Card"
        - Execute the processPayment function with the provided data.
        - Output:
            - The function should return true.
            - The payment should be recorded in the transaction history (a database).
    - **Test Case 2.2:** Handling an invalid payment method.
        - Input:
            - Amount: $50
            - Payment Method: "Invalid Method"
        - Execute the processPayment function with the provided data.
        - Output:
            - The function should return false.
            - No payment should be recorded in the transaction history.
- **Expected Outcome:**
    - **Test Case 2.1:** The function should successfully process the payment, return true, and record the payment in the transaction history.
    - **Test Case 2.2:** The function should return false for an invalid payment method, and no payment should be recorded.

Integration Testing

Test 3: Verify User-Vehicle Interaction

- **Objective:** Ensure the interaction between the User and Vehicle classes works as expected.
- **Test Description:**
    - **Test Case 3.1:** User retrieving vehicle information.
        - Create a test user account.
        - Create a test vehicle and mark it as available.
        - The user attempts to retrieve vehicle information using the getVehicleInfo function.
        - Output:
            - The function should return correct vehicle information.
            - The vehicle's availability status should change to "unavailable."
    - **Test Case 3.2:** Handling an unavailable vehicle.
        - Create a test user account.
        - Create a test vehicle and mark it as unavailable.
        - The user attempts to retrieve vehicle information using the getVehicleInfo function.

        - Output:
            - The function should return an error or message indicating that the vehicle is unavailable.
- **Expected Outcome:**
    - **Test Case 3.1:** The interaction should allow the user to retrieve correct vehicle information, and the vehicle's availability status should be updated.
    - **Test Case 3.2:** The system should handle cases where the user attempts to retrieve information for an unavailable vehicle.

Test 4: Verify Location-Rental Association Interaction

- **Objective:** Ensure the interaction between the Location and RentalAssociation classes works as expected.
- **Test Description:**
    - **Test Case 4.1:** Associating pickup and return locations.
        - Create a test rental association.
        - Create two test locations (pickup and return).
        - Associate the pickup and return locations with the rental association.
        - Output:
            - The rental association should correctly record the selected pickup and return locations.
    - **Test Case 4.2:** Handling unavailable or invalid locations.
        - Create a test rental association.
        - Attempt to associate pickup and return locations that are unavailable or invalid.
        - Output:
            - The system should handle these cases appropriately, with clear error messages or exceptions.
- **Expected Outcome:**
    - **Test Case 4.1:** The interaction should correctly associate pickup and return locations, and the system should store this information.
    - **Test Case 4.2:** The system should handle cases where the selected locations are unavailable or invalid.

System Testing

Test 5: End-to-End Rental Process

- **Objective:** Verify the end-to-end rental process, including account creation, vehicle rental, payment processing, and review submission.
- **Test Description:**
    - **Test Case 5.1:** Complete an end-to-end rental process.
        - A user creates an account.

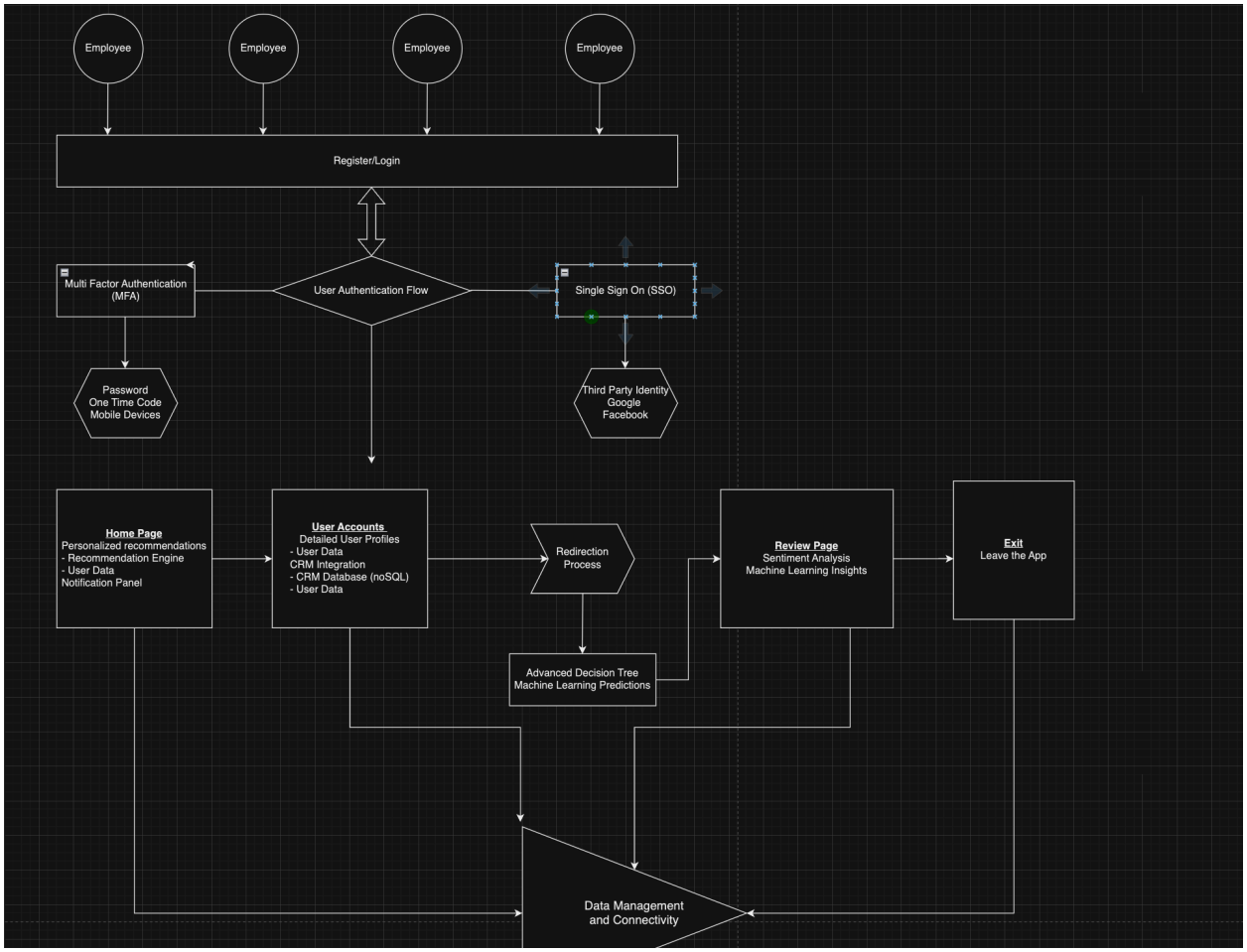
        - The user rents a vehicle.
        - The user makes a payment.
        - The user submits a review.
        - Output:
            - Each step of the process should be completed successfully.
            - User account, rental transaction, payment, and review should be correctly recorded.
    - **Test Case 5.2:** Handling issues during the process.
        - Introduce issues at various stages of the rental process (e.g., payment failure, review submission failure).
        - Output:
            - The system should handle these issues gracefully and provide clear error messages.
- **Expected Outcome:**
    - **Test Case 5.1:** The end-to-end process should complete successfully, and all associated data should be correctly recorded.
    - **Test Case 5.2:** The system should handle issues at different stages of the process and provide informative error messages.

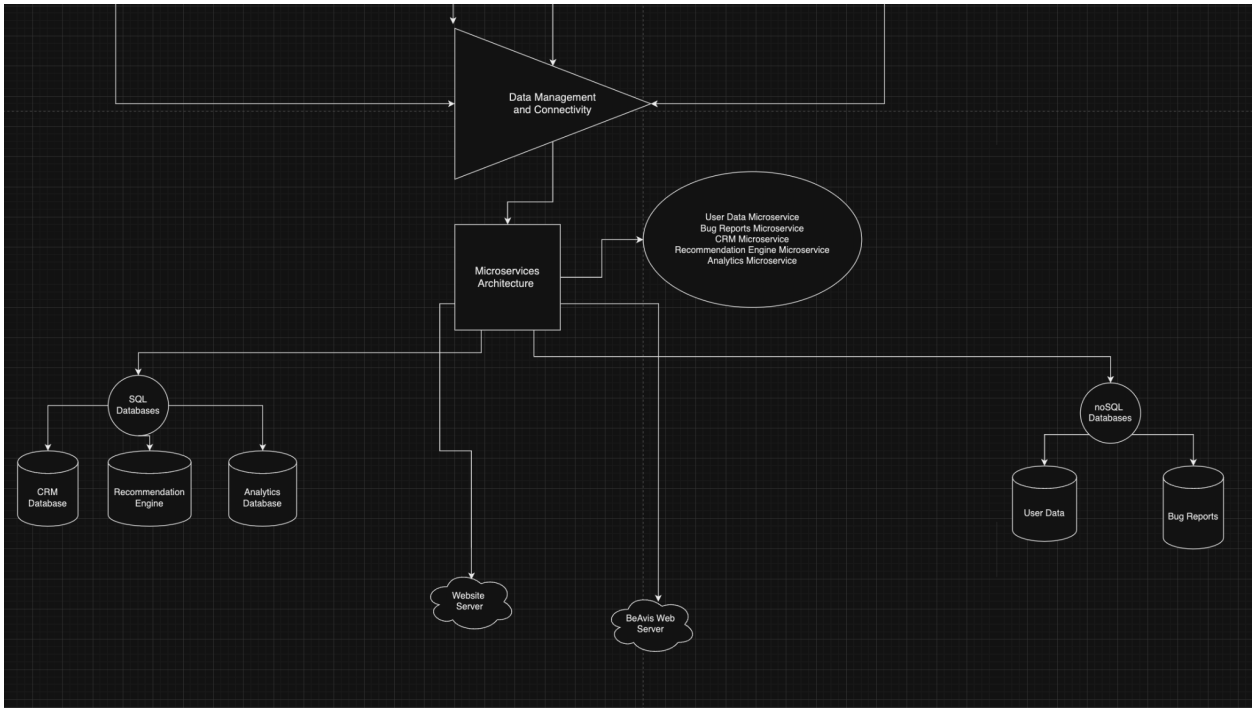
Test 6: Administrative Functions (Employee Class)

- **Objective:** Ensure that employees can effectively manage user accounts, vehicle inventory, and rental agreements.
- **Test Description:**
    - **Test Case 6.1:** Employee creating a new user account.
        - An employee logs in.
        - The employee creates a new user account for testing.
        - Output:
            - The employee should be able to create a new user account.
            - The new user account data should be correctly recorded.
    - **Test Case 6.2:** Handling unauthorized actions.
        - Attempt to perform unauthorized actions (e.g., altering system settings).
        - Output:
            - The system should enforce access control and prevent unauthorized actions.
- **Expected Outcome:**
    - **Test Case 6.1:** The employee should be able to create a new user account, and the data should be correctly recorded.
    - **Test Case 6.2:** The system should enforce access control, preventing unauthorized actions by employees.

Updated Software Architecture Diagram & Data Management Strategy

Software Architecture Diagram (This is the features we've added on, on top of the others that exist from Assignment 2. We didn't want to rewrite what already existed. Also these changes apply to both users and employees):





Modifications from the Initial Design (Assignment 2):

Enhanced User Authentication Flow:

- **Multi-Factor Authentication (MFA):** We have introduced MFA as a fundamental part of our authentication process to strengthen security. Users are required to provide multiple authentication factors, such as a password and a one-time code sent to their mobile devices.
- **Single Sign-On (SSO):** The system now supports SSO, allowing users to log in through third-party identity providers, such as Google or Facebook, for a seamless and secure experience.

Extended Home Page Functionality:

- **Personalized Recommendations:** The home page now features a recommendation engine that leverages user data to offer personalized suggestions, enhancing user engagement and satisfaction.
- **Notifications Panel:** A new notifications panel has been integrated to keep users informed of updates, announcements, and relevant information, creating a more dynamic user experience.

User Accounts Section Enhancements:

- **Detailed User Profiles:** User profiles have been enriched to include not only basic information but also contact details, communication preferences, and historical interactions.
- **CRM Integration:** We've integrated a Customer Relationship Management (CRM) system, allowing for more comprehensive tracking of user complaints,

inquiries, and interactions. This integration enhances our ability to provide top-notch customer support.

Redirection Process Improvements:

- **Advanced Decision Tree:** Redirection decisions are now based on a more advanced decision tree, considering user behavior patterns, context, and the nature of their issues.
- **Machine Learning Predictions:** We have integrated machine learning models to predict the most appropriate redirection paths, ensuring efficient issue resolution and user satisfaction.

Review Page Enhancements:

- **Sentiment Analysis:** The review page now incorporates automated sentiment analysis to gauge user satisfaction with the resolution of their issues, allowing us to proactively address potential concerns.
- **Machine Learning Insights:** Machine learning models provide insights into recurring issues and help in proactive issue resolution, ultimately improving the quality of support.

Data Management and Connectivity:

- **Microservices Architecture:** We've migrated to a microservices architecture, breaking down the system into smaller, independent components. This approach allows different components to scale independently, improves fault tolerance, and enhances system maintainability.
- **Data Distribution:** Data is now distributed across multiple databases to improve performance, reduce data redundancy, and enable efficient data retrieval and analysis.

Data Management Strategy: Our data management strategy is designed to ensure the efficient and secure handling of persistent and potentially sensitive data. We've made the following key design decisions regarding the choice of SQL and NoSQL databases:

Multiple Databases: To improve performance and data organization, we've divided our data into multiple databases:

- **SQL Databases:**
    - **User Data (SQL):** This database contains user profiles, authentication information, and user preferences. SQL databases are suitable for structured and relational data, ensuring data integrity and consistency for user information.
    - **Bug Reports (SQL):** Bug report data is stored in this SQL database, including information about reported issues, their status, and resolution

details. Using SQL databases for structured bug report data allows for complex querying and analysis.

- **NoSQL Databases:**
  - **CRM Database (NoSQL):** User complaints, violations, and interactions are managed in this NoSQL database. NoSQL databases are well-suited for unstructured and rapidly changing data, making them ideal for capturing and organizing user interactions.
  - **Recommendation Engine (NoSQL):** Data used for personalizing user experiences and recommendations are stored in this NoSQL database. NoSQL databases can efficiently handle the semi-structured data needed for recommendation systems.
  - **Analytics Database (NoSQL):** Logs, analytics, and sentiment analysis results are collected in this NoSQL database. The flexibility of NoSQL databases allows for efficient storage and retrieval of such unstructured data.

Data Encryption and Security: To ensure the confidentiality and integrity of sensitive data, we implement robust data encryption at rest and in transit. Access controls and authentication mechanisms are enforced to protect data from unauthorized access or breaches, ensuring compliance with privacy regulations.

Trade Off Discussion:

- **Multiple Databases vs. Single Database:** Our decision to employ multiple databases introduces complexity but offers significant benefits. It enhances performance, scalability, and data isolation. Each database can be optimized for its specific data type, which is especially valuable as the system scales.
- **SQL vs. NoSQL Databases:** Our decision to use a combination of SQL and NoSQL databases takes into consideration the strengths of each type:
  - SQL databases provide strong data integrity and allow for complex queries, making them suitable for structured data like user profiles and bug reports.
  - NoSQL databases excel in handling unstructured and rapidly changing data, making them ideal for managing user interactions, recommendation data, and analytics. This approach strikes a balance between data consistency and flexibility, catering to various data requirements within the BeAvis system.
- **MFA and SSO:** While the introduction of MFA and SSO enhances security, it may introduce usability challenges for some users. To address this, we've focused on implementing user-friendly MFA methods and ensuring a seamless SSO experience.
- **Machine Learning Integration:** The integration of machine learning introduces additional computational requirements and data needs. However, it significantly improves decision-making, personalization, and the proactive resolution of user issues. We've

invested in the necessary infrastructure to support machine learning capabilities effectively.

In summary, our choice of SQL and NoSQL databases is well-suited to the specific data types and requirements of each component of the BeAvis system, optimizing data management and ensuring the best user experience.