CarRentalManagement MVC Application: Documentation

Introduction

The **CarRentalManagement MVC Application** is a web-based platform designed to facilitate the management of car rental operations. This system enables users (admin, customers, and vendors) to log in, manage accounts, and interact with various services such as booking vehicles, managing rentals, and administrative tasks.

The application is built using the **Model-View-Controller (MVC)** architectural pattern, which separates the logic of the application into three interconnected components:

- **Model**: Represents the data and business logic.
- **View**: Represents the user interface (UI).
- **Controller**: Acts as an intermediary between the Model and the View, processing user input and updating the View.

The application ensures efficient management of rental operations by providing functionalities such as user authentication, car booking, account management, and admin dashboards.

Architecture Overview

1. MVC Architecture

The **CarRentalManagement MVC Application** is built on the **ASP.NET Core MVC** framework. This framework follows the **Model-View-Controller (MVC)** design pattern to manage the interactions between the user and the system. Here's a breakdown of the architecture components:

Model

The **Model** represents the application's data and business rules. It is responsible for interacting with the database and performing logic such as user authentication, car rentals, and managing rental transactions.

Key components of the **Model**:

- **Data Models**: These are C# classes that define the properties of various entities in the system, such as `Admin`,`Customer`, `Car`, `Booking`, Registration etc.
- **Business Logic**: This is the core logic of the system, typically housed in services (`Booking Service`), which are responsible for processing data and executing business rules.
- **Data Access**: The application uses **Entity Framework Core** (EF Core) for data persistence, allowing the application to interact with a relational database using ORM (Object-Relational Mapping).

View

The **View** component is responsible for rendering the user interface. It receives data from the **Controller** and displays it to the user. In this application, the views are implemented using **Razor** (a templating engine that integrates HTML with C# code), which is a part of the ASP.NET Core MVC framework.

Key components of the **View**:

- **Login Page**: Displays a form for users to input their credentials and authenticate.
- **Dashboard**: Displays information specific to different types of users (Admin, Customer, Vendor), such as car availability, booking status, and rental management.
- **Forms**: Allows users to perform actions like registering, updating profile details, and booking a car.

Controller

The **Controller** serves as the intermediary between the **Model** and the **View**. It handles user inputs, updates the **Model**, and returns the appropriate **View**.

Key responsibilities of the **Controller**:

- **Handling HTTP Requests**: The controller listens for HTTP requests (GET, POST) from the user and responds accordingly.
- **User Authentication**: The controller verifies user credentials and manages session or cookie-based authentication.

- **Data Manipulation**: It processes data from the model and passes it to the view for rendering. It may also manipulate data, such as creating or updating records (e.g., car rentals or bookings).
- **View Binding**: The controller ensures that the correct view is rendered with the necessary data. This could include validating user input and displaying appropriate error messages.

Controllers used in this application

1. **AdminController** manages system-wide operations and ensures proper roles and permissions for other users in the system. The `AdminController` in the application defines several action methods. Here's a summary of each action:
   - **Login (GET)**:

     Displays the login view for the admin to input their username and password.
     URL: `/Admin/Login`
     HTTP Method: GET
     Logging: Logs the action when it's called.

   - **Register (GET)**:

     Displays the registration form for an admin to sign up.
     URL: `/Admin/Register`
     HTTP Method: GET
     Logging: Logs the action when it's called.

   - **Register (POST)**:

     Handles the registration logic for creating a new admin.
     Checks if the provided username already exists.
     Adds the new admin to the database and redirects to the login page upon success.
     URL: `/Admin/Register`
     HTTP Method: POST
     Logging: Logs the flow of the registration process.

   - **Login (POST)**:

     Handles the login logic for admins and users.
     Verifies the username and password against the database.

On successful login, sets session and cookies for the user/admin and redirects to the respective dashboard.
If login fails, it returns an error message.
URL: `/Admin/Login`
HTTP Method: `POST`
Logging: Logs the attempt and results of the login.

- **AdminDashboard**:

  Displays the dashboard for the admin.
  If the admin is not logged in, redirects to the login page.
  URL: `/Admin/AdminDashboard`
  HTTP Method: `GET`
  Logging: Logs the action when the dashboard is accessed.

- **UserDashboard**:

  Displays the dashboard for a user (if the admin is not logged in).
  If the user is not logged in, redirects to the login page.
  URL: `/Admin/UserDashboard`
  HTTP Method: `GET`
  Logging: Logs the action when the user dashboard is accessed.

- **Logout**:

  Logs out the admin or user by clearing session data and deleting cookies.
  Redirects to the login page after logging out.
  URL: `/Admin/Logout`
  HTTP Method: `POST` (but typically used for clearing the session)
  Logging: Logs the logout action.

2. **BookingController** handles customer reservations and interacts with both `CustomerController` and `CarController` to validate bookings. The Booking`Controller` in the application defines several action methods. Here's a summary of each action:
   - **Index**:

     Fetches a list of bookings, including related car and customer data.
     Displays a message if no bookings are found.

- **Add (GET)**:

  Renders the form to add a new booking.
  Retrieves and passes a list of available cars and customers to the view.

- **Add (POST)**:

  Adds a new booking using a stored procedure `AddBooking` and handles the booking details.
  Validates the model and logs any validation errors.
  Logs the stored procedure execution result (success or failure).
  Returns the appropriate view based on success or failure.

- **Details**:

  Fetches and displays details of a booking based on its ID, including related car and customer data.
  Logs whether the booking was found or not.

3. **CarController** ensures that cars are accurately registered, updated, and available for booking. The Car`Controller` in the application defines several action methods. Here's a summary of each action:

- **Index**:
  Fetches and displays a list of cars using the `IGenericRepository`.
  If no cars are found, a warning is logged, and a message is displayed to the user.
  If there is an error while fetching cars, it logs the error and returns a 500 status code.

- **Add (GET)**:

  Renders the form for adding a new car.
  No database interaction, just rendering the form view.

- **Add (POST)**:

  Handles the form submission to add a new car to the database.
  Validates the model and logs any warnings if the model state is invalid.
  Attempts to add the new car using the `Register` method of the repository.

If successful, the user is redirected to the `Index` view with a success message.
If there's an error, the model error is logged, and the user is prompted to try again.

- **Details**:

  Fetches and displays details for a specific car based on its ID.
  Logs warnings if the car is not found or if the ID is invalid.
  Returns a 500 status code if there is an exception while fetching data.

- **Edit (GET)**:

  Retrieves the details of a car for editing, based on its ID.
  If the car is not found, a warning is logged, and the `NotFound` result is returned.
  If there is an exception, a 500 status code is returned.

- **Edit (POST)**:

  Handles the form submission to update an existing car's details.
  Validates the model and logs any warnings if validation fails.
  Attempts to update the car using the `Update` method of the repository.
  If successful, the user is redirected to the `Index` view with a success message.
  Logs errors if there's an issue with the update.

- **Delete**:

  Deletes a car based on its ID.
  If the car is not found, a warning is logged, and the `NotFound` result is returned.
  If successful, the car is deleted, and the user is redirected to the `Index` view with a success message.
  Logs errors if there's an issue with the deletion process.

4. **CustomerController** facilitates user details, profile updates, and viewing historical bookings.

Dependencies:

`IHttpClientFactory`: To create HTTP clients for making requests to the Web API.

`ILogger<CustomerController>`: To log information, warnings, and errors during the operation of the controller.

Action Methods:

- **Index**:

  Fetches a list of customers from the Web API.
  If the API call is successful, it returns the customer list to the view.
  If there is an error (e.g., failed HTTP request), it logs the issue and returns an error view.

- **Add (GET)**:

  Renders the form to add a new customer.
  Logs the action of rendering the form for adding a customer.

- **Add (POST)**:

  Handles the form submission to add a new customer.
  It checks if the model state is valid. If not, it returns the view with the model to allow the user to correct any input.
  If valid, it makes an HTTP POST request to the Web API to create a new customer.
  If the API request is successful, a success message is set, and the user is redirected to the `Index` action.
  If the request fails, it logs the error and re-renders the view with the customer data.

- **Details**:

  Fetches and displays details of a specific customer by their `id`.
  If the customer is found, it renders the customer details view.
  If the customer is not found, it logs a warning and returns a `NotFound` response.

- **Delete**:

  Deletes a specific customer by their `id`.
  Sends a DELETE request to the Web API to remove the customer.

If successful, the user is redirected to the `Index` action.
If there is an error, it logs the failure and displays an error view.

- **Edit (GET)**:

  Fetches the current details of a customer for editing based on their `id`.
  Renders the edit view with the customer data for updating.
  If the customer does not exist, it returns a `NotFound` response.

- **Edit (POST)**:

  Handles the form submission to update the customer's details.
  Validates the model, and if valid, sends a PUT request to the Web API to update the customer.
  On success, the user is redirected to the `Index` view.
  On failure, it logs the error and returns the view with the model for correction.

2. Authentication and Authorization

Authentication and authorization are essential aspects of the **CarRentalManagement MVC Application**, ensuring that users can only access appropriate features based on their roles.

- **User Authentication**: The system allows users to authenticate using a **username** and **password**. Once authenticated, session cookies or JWT tokens are used to track the user's session and maintain login state.
- **Role-Based Authorization**: The application has various user roles, such as:
  - **Admin**: Has full access to all resources and management features.
  - **Customer**: Can view available cars and make bookings.

3. Loggers

The application uses **Serilog** for structured and efficient logging. Serilog is configured to log critical application events, making it easier to debug and monitor application behavior.

4. Database Layer

The application uses **Entity Framework Core (EF Core)** to manage the data layer. EF Core is an ORM framework that facilitates interactions between the application and the relational database.

Key data entities:

- **User**: Represents the user data, including username, password, phone number, and role.
- **Car**: Represents the car information, such as model, availability, price, and make, seat availability, fuel type details.
- **Booking**: Represents the booking information, including user details, car details, and rental dates.

EF Core supports database migrations, which allow easy creation and updating of the database schema as the application evolves.

5. Web API Integration

The application utilizes a Web API for handling backend operations. The `CustomerController` communicates with the API using an `IHttpClientFactory` instance.

Example API Endpoints:

- **Get all customers**: `GET http://localhost:5171/api/Customer`
- **Get customer by ID**: `GET http://localhost:5171/api/Customer/{id}`
- **Add customer**: `POST http://localhost:5171/api/Customer`
- **Edit customer**: `PUT http://localhost:5171/api/Customer/{id}`
- **Delete customer**: `DELETE http://localhost:5171/api/Customer/{id}`

6. CORS Configuration

For the application to function correctly when making API calls, CORS (Cross-Origin Resource Sharing) has been enabled. This allows the application to interact with APIs hosted on different domains.

7. Repository Layer

The application implements the Repository Pattern to separate data access logic from business logic. This design makes the application easier to maintain and test.

Generic Repository

The `GenericRepository<T>` class provides a reusable implementation of the repository pattern for CRUD operations. It works with any entity type (T) and uses the Entity Framework Core's `DbContext` to interact with the database.

Key Methods:

- **GetByData(int Id)**: Retrieves an entity by its ID. Throws a `KeyNotFoundException` if the entity is not found.
- **GetAll()**: Retrieves all entities of type T.
- **Register(T Entity)**: Adds a new entity to the database.
- **Update(T Entity)**: Updates an existing entity in the database.
- **Delete(T Entity)**: Removes an entity from the database.
- **Save()**: Saves changes to the database.

The `IGenericRepository<T>` interface defines the contract for the `GenericRepository` class, ensuring consistency and reusability across different entity types.

8. Client-Side Interactivity

The client-side of the application is designed to provide a seamless and responsive experience for users. This is achieved using **Bootstrap 4** for responsive design and **JavaScript** to handle interactivity.

- **Forms**: The login form and other interactive forms are styled using Bootstrap to be responsive and user-friendly.
- **Error Handling**: Client-side error handling ensures that the user sees real-time validation messages when they input invalid data, such as missing fields or incorrect password formats.

- **JavaScript**: Used to manage dynamic behavior like clearing error messages when the user starts typing, handling form submissions, and providing interactivity.

9. Error Handling and Validation

Proper error handling ensures that users receive meaningful feedback and can easily resolve issues. The application uses **Model Validation** to validate user input on both the client and server sides.

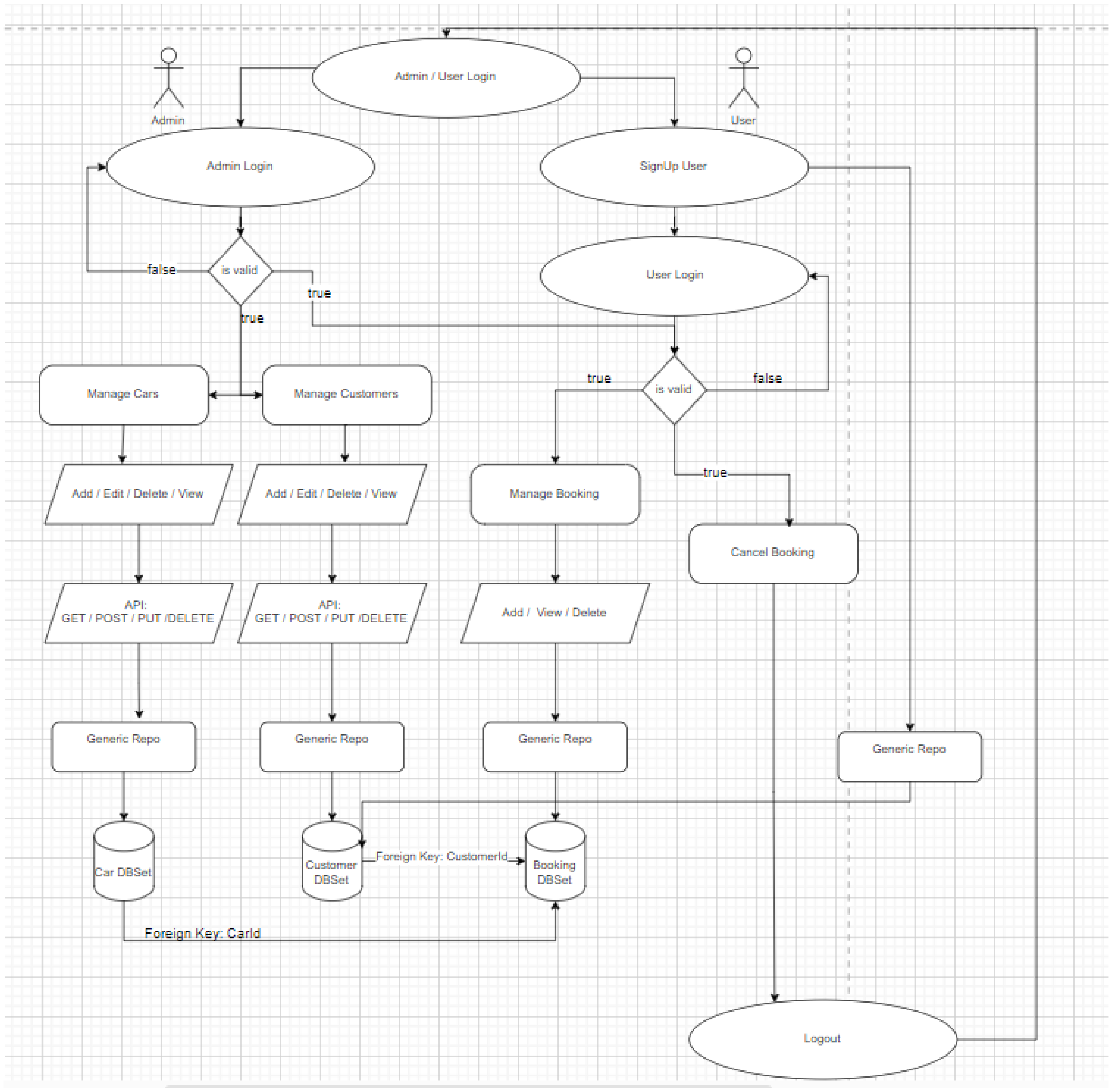
- **Server-Side Validation**: When the form is submitted, the server validates the data using **Data Annotations** in the model. If any validation fails, an error message is returned to the view, indicating what went wrong (e.g., invalid username or password).
- **Client-Side Validation**: On the client side, JavaScript is used to provide immediate feedback as the user fills out the form, preventing the need for a full page reload on validation errors.

10. Responsive Design

The application is built to be responsive and work well across different devices, including desktops, tablets, and smartphones. The use of **Bootstrap** ensures that the layout adapts automatically based on screen size, providing an optimized experience for each user.

Conclusion

The **CarRentalManagement MVC Application** provides a comprehensive and user-friendly platform for managing car rentals. Its modular architecture, based on the MVC pattern, ensures that the system is maintainable, scalable, and secure. The application supports various roles such as Admin and Customer providing specific functionalities for each role. Through the use of robust authentication, error handling, and responsive design, the system ensures a smooth user experience while maintaining the integrity of data and business processes.


Admin / User Login
Admin
User
Admin Login
SignUp User
is valid
false
true
true
User Login
true
false
is valid
true
Manage Cars
Manage Customers
Manage Booking
Cancel Booking
Add / Edit / Delete / View
Add / Edit / Delete / View
API:
GET / POST / PUT /DELETE
API:
GET / POST / PUT /DELETE
Add / View / Delete
Generic Repo
Generic Repo
Generic Repo
Generic Repo
Car DBSet
Customer DBSet
Foreign Key: CustomerId
Booking DBSet
Foreign Key: CarId
Logout