TapRide

The Cab Booking System is a comprehensive web application designed to streamline ride booking and management for both users and drivers. Below is a brief overview of its key features:

**User Features**

1. **Ride Booking**:
   * Users can book rides by providing pickup and drop-off locations.
   * Real-time fare calculation and ride confirmation.
2. **Ride History**:
   * View past rides with details such as date, time, fare, and locations.
   * Pagination for easy navigation through ride history.
3. **Rating System**:
   * Rate completed rides and provide feedback.
   * Ratings are stored and displayed for future reference.
4. **Payment History**:
   * View payment transactions for completed rides.
   * Detailed breakdown of fare and payment methods.

**Driver Features**

1. **Driver Status Management**:
   * Drivers can update their availability status (OnDuty or Unavailable).
   * Status persists across sessions and can be updated dynamically.
2. **Completed Rides**:
   * View a list of completed rides with pagination.
   * Detailed ride information, including fare and locations.
3. **Driver Statistics**:
   * View performance metrics such as total rides completed, earnings, and average ratings.

**Authentication and Security**

1. **Role-Based Access**:
   * Separate dashboards for users and drivers based on their roles.
   * Secure login using JWT tokens.
2. **Session Management**:
   * Automatic session expiration handling with redirection to the login page.

**Additional Features**

1. **Responsive Design**:
   * Optimized for both desktop and mobile devices.
2. **Error Handling**:
   * User-friendly error messages and toast notifications for failed actions.
3. **Modern UI**:
   * Clean and intuitive interface using Angular and Bootstrap.

This feature-rich system ensures a seamless experience for both users and drivers, making ride booking and management efficient and reliable.

**Technologies Used**

**Frontend**

1. **Angular**:
   * Framework for building dynamic and responsive single-page applications (SPAs).
   * Used for implementing components, routing, and state management.
2. **Bootstrap**:
   * CSS framework for responsive and mobile-first design.
   * Used for styling and layout.
3. **HTML5**:
   * Markup language for structuring the web pages.
4. **CSS3**:
   * Styling language for designing the user interface.
5. **TypeScript**:
   * Superset of JavaScript used for writing scalable and maintainable code.
6. **RxJS**:
   * Library for reactive programming and handling asynchronous data streams.
7. **Toastr**:
   * Library for displaying toast notifications for success, error, and info messages.

**Backend**

1. **ASP.NET Core**:
   * Framework for building RESTful APIs.
   * Used for handling business logic and database interactions.
2. **Entity Framework Core**:
   * ORM (Object-Relational Mapping) tool for database operations.
3. **SQL Server**:
   * Relational database for storing user, driver, ride, and payment data.
4. **JWT (JSON Web Tokens)**:
   * Used for secure authentication and authorization.

**Development Tools**

1. **Visual Studio Code**:
   * Code editor for writing and debugging the application.
2. **Swagger**:
   * API testing tool for verifying backend endpoints.
3. **Git**:
   * Version control system for managing code changes.

**Deployment**

1. **IIS (Internet Information Services)**:
   * Used for hosting the backend API.
2. **Angular CLI**:
   * Tool for building and deploying the frontend application.

These technologies work together to create a robust, scalable, and user-friendly cab booking system.

**Installation**

Follow the steps below to set up and run the project on your local machine:

**Prerequisites**

Ensure you have the following installed on your system:

1. **Node.js** (v16 or higher): [Download Node.js](vscode-file://vscode-app/c:/Users/2408367/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)
2. **Angular CLI** (v15 or higher): Install globally using:

npm install -g @angular/cli

1. **Git**: [Download Git](vscode-file://vscode-app/c:/Users/2408367/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)

**Steps to Install**

**Clone the Repository**

1. Use the following command to clone the project repository:
2. git clone [vishalmaurya8/CabFrontend](https://github.com/vishalmaurya8/CabFrontend)

**Navigate to the Project Directory**

1. Change to the project directory:

cd CabSystem/CabUI/AuthUI

**Install Dependencies**

1. Run the following command to install all required dependencies:

npm install

Running the Project

1. Start the Development Server

Use the Angular CLI to start the development server:Ng serve

This will start the application and serve it at http://localhost:4200.

2. Access the Application

Open your browser and navigate to:

<http://localhost:4200>

**Backend Setup**

Ensure the backend server is running for the application to function correctly:

1. Clone the backend repository (if applicable).
2. Follow the backend setup instructions provided in its documentation.
3. Start the backend server (e.g., https://localhost:7109).

**Optional Commands**

* **Build the Project**: To create a production build:

ng build --prod

* **Run Unit Tests**: To execute unit tests:

ng test

**Troubleshooting**

If you encounter issues during installation:

1. Ensure all prerequisites are installed.
2. Delete the [node\_modules](vscode-file://vscode-app/c:/Users/2408367/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html" \o ") folder and reinstall dependencies:

rm -rf node\_modules

npm install

1. Check for Angular CLI version compatibility:

ng version

**User Dashboard Features Documentation**

This section details the various functionalities available to a user on their dashboard, complete with descriptions and visual aids (screenshots) to guide the reader.

**1. Book Ride**

This feature allows users to initiate and schedule a ride from their current location or a specified pickup point to their desired destination.

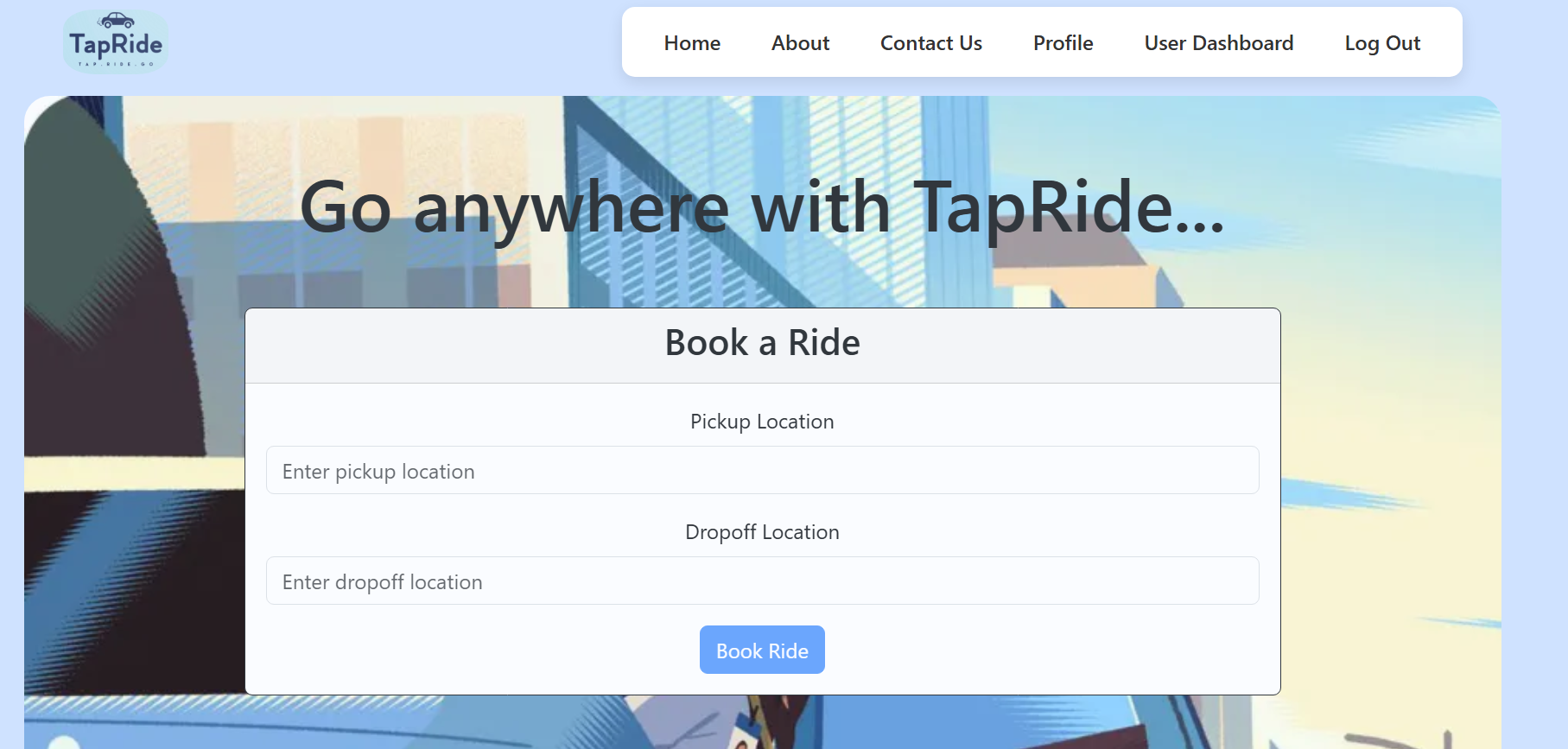
**Description:** The "Book Ride" interface provides fields for users to input their pickup location, destination, and optionally, select a vehicle type. It often includes a map view to visually confirm the route and estimated fare. Upon successful input, the system searches for available drivers and provides an estimated time of arrival (ETA) and fare details.

**Key Steps & Interactions:**

* **Entering Pickup Location:** Users can either use their current GPS location or manually input an address.
* **Entering Destination:** Users provide the address where they wish to be dropped off.
* **Selecting Vehicle Type (Optional):** If multiple vehicle categories are available (e.g., Economy, Premium, SUV), users can choose their preferred option.
* **Fare Estimation:** The system calculates and displays the estimated cost of the ride based on distance, time, and vehicle type.
* **Confirmation:** A final confirmation step before the ride request is dispatched to drivers.

**Screenshots to Include:**

* **Screenshot 1.1:**



RideController

using AutoMapper;

using CabSystem.DTOs;

using CabSystem.Exceptions; // ✅ Use your existing custom exception namespace

using CabSystem.Models;

using CabSystem.Repositories;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using System.Security.Claims;

namespace CabSystem.Controllers

{

[Route("api/[controller]")]

[ApiController]

[Authorize]

public class RideController : ControllerBase

{

private readonly IRideRepository \_rideRepository;

private readonly IMapper \_mapper;

private readonly IPaymentRepository paymentRepository;

private readonly IRideFareService fareService;

public RideController(IRideRepository rideRepository, IMapper mapper, IPaymentRepository paymentRepository, IRideFareService fareService)

{

\_rideRepository = rideRepository;

\_mapper = mapper;

this.paymentRepository = paymentRepository;

this.fareService = fareService;

}

//USER-ONLY: Get rides by userId

[Authorize(Roles = "User")]

[HttpGet("user/{userId}")]

public async Task<IActionResult> GetRidesByUserId(int userId)

{

var rides = await \_rideRepository.GetRidesByUserIdAsync(userId);

if (rides == null || !rides.Any())

throw new NotFoundException($"No rides found for user ID {userId}");

var rideDtos = \_mapper.Map<IEnumerable<RideDTO>>(rides);

return Ok(rideDtos);

}

//USER-ONLY: Book a ride

[Authorize(Roles = "User")]

[HttpPost("book")]

public async Task<IActionResult> BookRide([FromBody] CreateRideDTO dto)

{

var userId = GetUserIdFromToken();

// 🛑 Check if there's any unpaid ride

var unpaidRide = await \_rideRepository.GetLatestUnpaidRideByUserIdAsync(userId);

if (unpaidRide != null)

throw new BadRequestException("You already have an unpaid ride. Please complete payment first.");

var ride = \_mapper.Map<Ride>(dto);

ride.UserId = userId;

ride.Status = "Requested";

ride.Fare = fareService.CalculateFare(dto.PickupLocation, dto.DropoffLocation);

var result = await \_rideRepository.BookRideAsync(ride);

return Ok(\_mapper.Map<RideDTO>(result));

}

private int GetUserIdFromToken()

{

var userIdClaim = User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.NameIdentifier);

if (userIdClaim == null)

throw new UnauthorizedAccessException("User ID not found in token.");

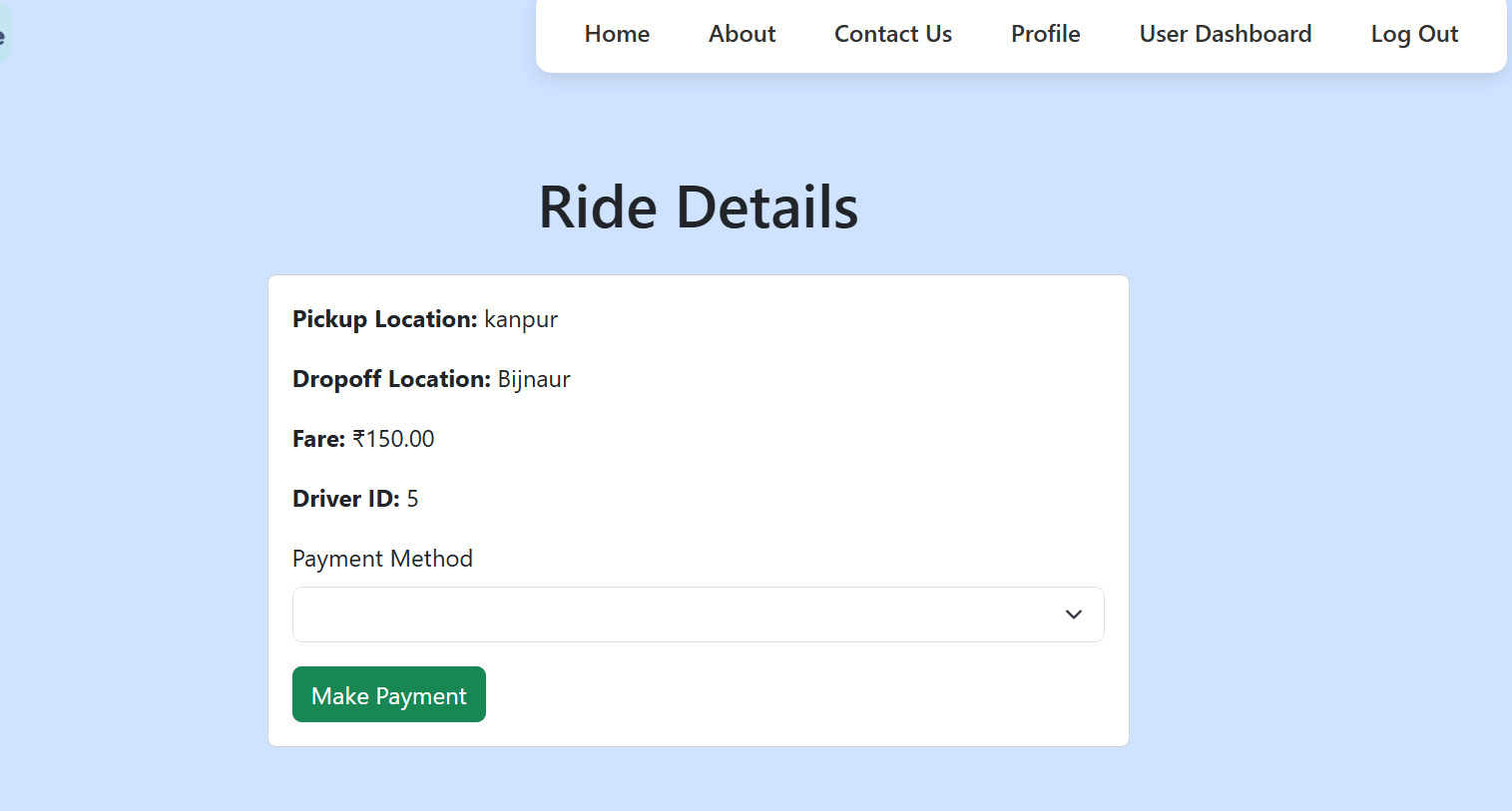
return int.Parse(userIdClaim.Value);

}

}

}

* **Screenshot 1.2:**



**2. Make Payment for Ride**

This feature facilitates the secure processing of payments for completed rides.

**Description:** Once a ride concludes, the user is prompted to make a payment. This section details the available payment methods and the process of completing the transaction. The system ensures secure handling of financial information.

**Key Steps & Interactions:**

* **Ride Summary & Total Fare:** Display of the ride details (distance, duration, actual fare).
* **Payment Method Selection:** Users can choose from pre-saved payment methods (e.g., credit/debit card, digital wallet) or add a new one.
* **Payment Confirmation:** After selecting a method, the user confirms the payment.
* **Payment Processing:** The system processes the transaction.
* **Success/Failure Notification:** Immediate feedback on the payment status.

**Screenshots to Include:**

* **Screenshot 2.1:**

using AutoMapper;

using CabSystem.DTOs;

using CabSystem.Exceptions;

using CabSystem.Models;

using CabSystem.Repositories;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using System.Security.Claims;

namespace CabSystem.Controllers

{

[Route("api/[controller]")]

[ApiController]

[Authorize] // ✅ Protect these endpoints

public class PaymentController : ControllerBase

{

private readonly IPaymentRepository \_paymentRepo;

private readonly IMapper \_mapper;

private readonly IRideRepository rideRepository;

public PaymentController(IPaymentRepository paymentRepo, IMapper mapper, IRideRepository rideRepository)

{

\_paymentRepo = paymentRepo;

\_mapper = mapper;

this.rideRepository = rideRepository;

}

private int GetUserIdFromToken()

{

var userIdClaim = User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.NameIdentifier);

if (userIdClaim == null)

throw new UnauthorizedAccessException("User ID not found in token.");

return int.Parse(userIdClaim.Value);

}

[Authorize(Roles = "User")]

[HttpPost]

public async Task<IActionResult> MakePayment([FromBody] CreatePaymentDTO dto)

{

if (!ModelState.IsValid)

throw new BadRequestException("Invalid payment data");

var userId = GetUserIdFromToken();

// 🔍 Get the latest ride not completed or paid

var ride = await rideRepository.GetLatestUnpaidRideByUserIdAsync(userId);

if (ride == null)

throw new NotFoundException("No unpaid or uncompleted ride found for payment.");

// ✅ Create payment

var payment = new Payment

{

RideId = ride.RideId,

Amount = ride.Fare,

Method = dto.Method,

Status = "Paid",

Timestamp = DateTime.UtcNow

};

var inserted = await \_paymentRepo.InsertPaymentAsync(payment);

// ✅ Mark ride as completed

ride.Status = "Completed";

await rideRepository.UpdateRideAsync(ride);

var response = \_mapper.Map<PaymentDTO>(inserted);

return Ok(response);

}

[Authorize(Roles = "User")]

[HttpGet("my-payments")]

public async Task<IActionResult> GetMyPayments()

{

var userId = GetUserIdFromToken();

var payments = await \_paymentRepo.GetPaymentsByUserIdAsync(userId);

if (payments == null || !payments.Any())

throw new NotFoundException("No payments found for your account.");

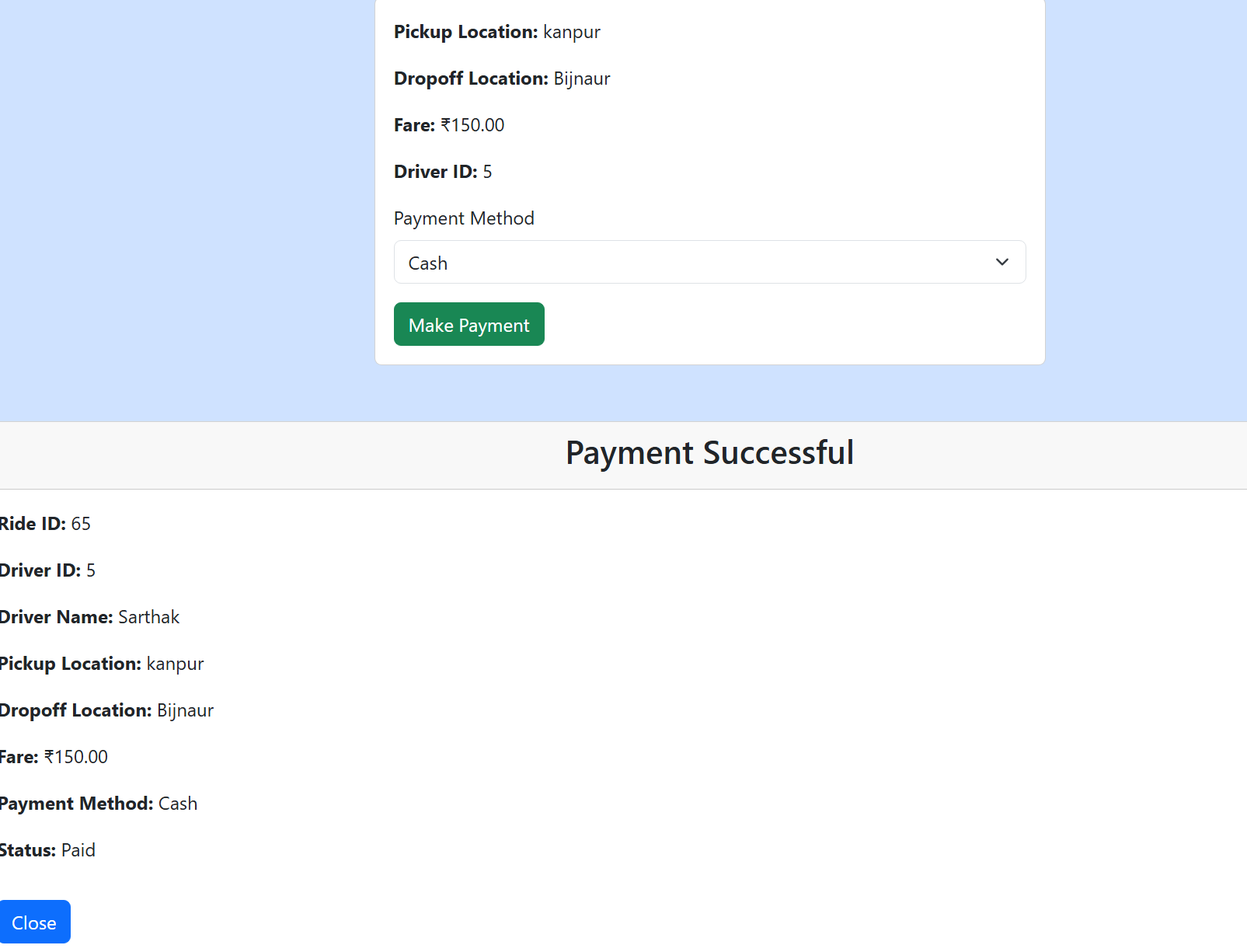
var result = \_mapper.Map<List<PaymentDTO>>(payments);

return Ok(result);

}

}

}

* **Screenshot 2.2:**
* 

using CabSystem.Data;

using CabSystem.Exceptions;

using CabSystem.Models;

using Microsoft.EntityFrameworkCore;

namespace CabSystem.Repositories

{

public class PaymentRepository : IPaymentRepository

{

private readonly CabSystemContext \_context;

public PaymentRepository(CabSystemContext context)

{

\_context = context;

}

public async Task<Payment> InsertPaymentAsync(Payment payment)

{

\_context.Payments.Add(payment);

await \_context.SaveChangesAsync();

//Re-fetch with Ride included

return await \_context.Payments

.Include(p => p.Ride)

.ThenInclude(r => r.Driver)

.ThenInclude(d => d.User)

.FirstOrDefaultAsync(p => p.PaymentId == payment.PaymentId);

}

public async Task<List<Payment>> GetAllPaymentsAsync()

{

return await \_context.Payments.ToListAsync();

}

public async Task<Payment?> GetPaymentByRideIdAsync(int rideId)

{

return await \_context.Payments.FirstOrDefaultAsync(p => p.RideId == rideId);

}

public async Task<List<Payment>> GetPaymentsByUserIdAsync(int userId)

{

return await \_context.Payments

.Include(p => p.Ride)

.ThenInclude(r => r.Driver)

.ThenInclude(d => d.User)

.Include(p => p.Ride)

.ThenInclude(r => r.User)

.Where(p => p.Ride.UserId == userId)

.ToListAsync();

}

public async Task<Payment> InsertPaymentForLatestUnpaidRideAsync(int userId, string method)

{

var ride = await \_context.Rides

.Include(r => r.Payment)

.Where(r => r.UserId == userId && (r.Payment == null || r.Payment.Status != "Paid"))

.OrderByDescending(r => r.RideId)

.FirstOrDefaultAsync();

if (ride == null)

throw new NotFoundException("No unpaid ride found.");

var payment = new Payment

{

RideId = ride.RideId,

Amount = ride.Fare,

Method = method,

Status = "Paid",

Timestamp = DateTime.UtcNow

};

\_context.Payments.Add(payment);

await \_context.SaveChangesAsync();

return payment;

}

}

}

**3. Receipt Card for Ride**

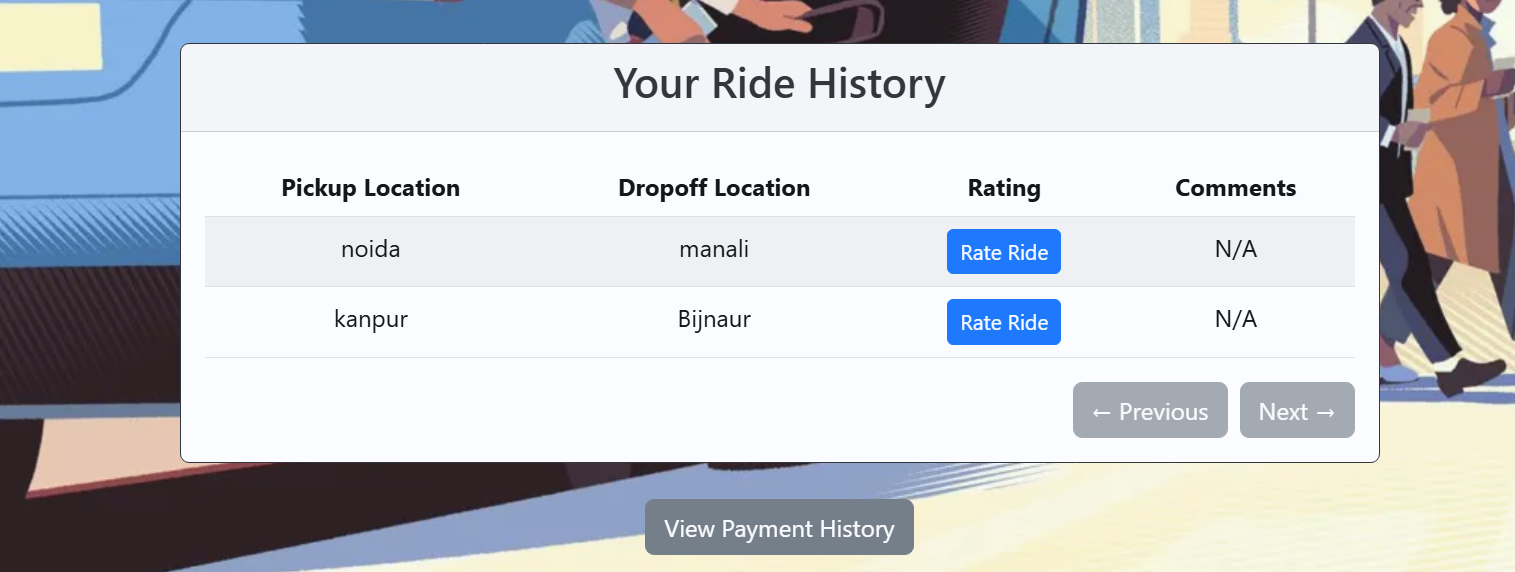
This feature provides a digital receipt for a completed and paid ride, offering a detailed breakdown of the transaction.

**Description:** Upon successful payment, a digital receipt card is generated for the user. This card serves as a record of the transaction, providing details such as the ride ID, date, time, pickup and drop-off locations, driver information, fare breakdown, and payment method used.

**Key Information Displayed:**

* Ride ID
* Date and Time of Ride
* Pickup and Drop-off Addresses
* Driver Name and Vehicle Details
* Fare Breakdown (Base fare, surcharges, discounts, total)
* Payment Method Used

**Screenshots to Include:**

* **Screenshot 3.1:**
* 

**4. Rate That Ride**

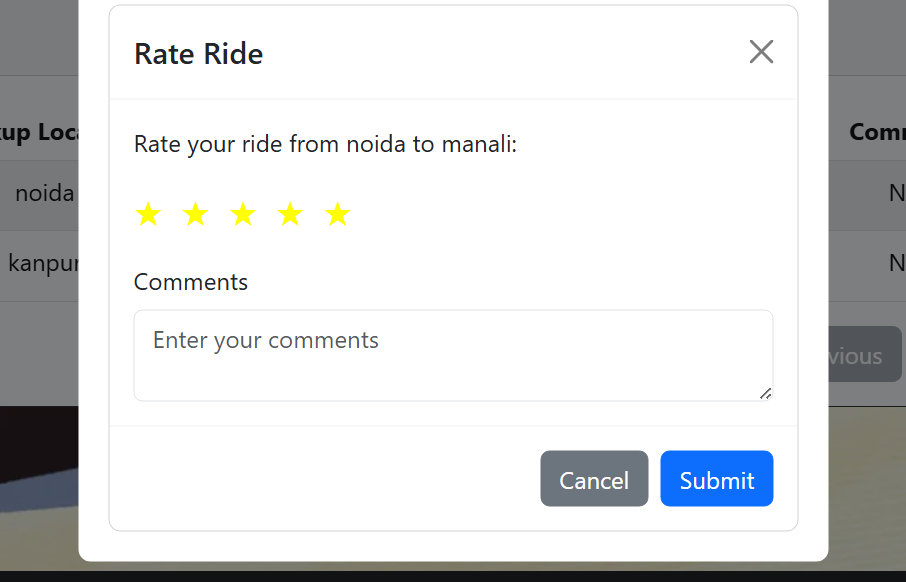
This feature allows users to provide feedback on their ride experience, contributing to driver quality and service improvement.

**Description:** After a ride is completed and paid for, users are given the opportunity to rate their experience. This typically involves a star-rating system (e.g., 1 to 5 stars) and an optional text field for additional comments. This feedback is crucial for maintaining service quality and addressing any issues.

**Key Steps & Interactions:**

* **Rating Interface:** Presentation of stars or a similar rating scale.
* **Optional Comments:** A text area for users to provide specific feedback.
* **Submission:** Button to submit the rating.

**Screenshots to Include:**

* **Screenshot 4.1:**
* 

using AutoMapper;

using CabSystem.DTOs;

using CabSystem.Exceptions;

using CabSystem.Models;

using CabSystem.Repositories;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using System.Security.Claims;

[ApiController]

[Route("api/[controller]")]

[Authorize(Roles = "User")]

public class RatingController : ControllerBase

{

private readonly IRatingRepository \_ratingRepo;

private readonly IMapper \_mapper;

public RatingController(IRatingRepository ratingRepo, IMapper mapper)

{

\_ratingRepo = ratingRepo;

\_mapper = mapper;

}

private int GetUserIdFromToken()

{

var userIdClaim = User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.NameIdentifier);

if (userIdClaim == null)

throw new UnauthorizedAccessException("User ID not found in token.");

return int.Parse(userIdClaim.Value);

}

[HttpPost]

public async Task<IActionResult> AddRating([FromBody] CreateRatingDTO dto)

{

if (!ModelState.IsValid) throw new BadRequestException("Invalid rating data");

var userId = GetUserIdFromToken();

// 🔐 Make sure this ride belongs to the user

var isOwner = await \_ratingRepo.IsRideOwnedByUserAsync(dto.RideId, userId);

if (!isOwner)

throw new UnauthorizedAccessException("You cannot rate a ride not booked by you.");

// 🛑 Check for existing rating

var existing = await \_ratingRepo.GetRatingByRideIdAsync(dto.RideId);

if (existing != null)

throw new ConflictException("Rating already exists for this ride.");

var rating = \_mapper.Map<Rating>(dto);

var result = await \_ratingRepo.AddRatingAsync(rating);

return Ok(\_mapper.Map<RatingDTO>(result));

}

[HttpPut("{rideId}")]

public async Task<IActionResult> UpdateRating(int rideId, [FromBody] UpdateRatingDto dto)

{

if (!ModelState.IsValid) throw new BadRequestException("Invalid update input");

var userId = GetUserIdFromToken();

var updated = await \_ratingRepo.UpdateRatingAsync(rideId, userId, dto.Score, dto.Comments);

if (updated == null)

throw new NotFoundException("You cannot update rating for this ride.");

return Ok(\_mapper.Map<RatingDTO>(updated));

}

[HttpGet]

public async Task<IActionResult> GetMyRatings()

{

var userId = GetUserIdFromToken();

var ratings = await \_ratingRepo.GetRatingsByUserIdAsync(userId);

return Ok(\_mapper.Map<IEnumerable<RatingDTO>>(ratings));

}

}

**5. View Payment History**

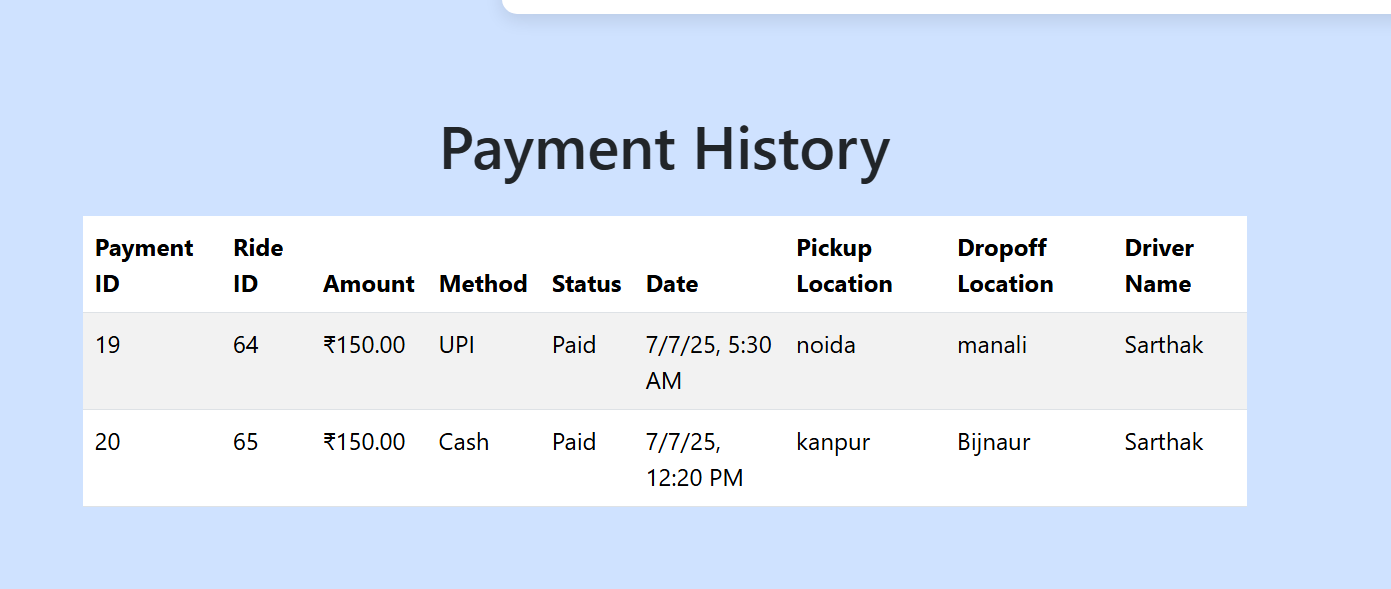
This feature enables users to access a comprehensive record of all their past ride payments.

**Description:** The "Payment History" section provides a chronological list of all payments made for rides. Each entry typically includes a summary (e.g., date, amount, basic ride details) and allows users to view the full receipt card for each transaction. This helps users track their expenses and review past rides.

**Key Information Displayed (List View):**

* Date of Payment
* Amount Paid
* Brief Ride Description (e.g., "Ride from A to B")
* Link/Button to view full Receipt Card

**Screenshots to Include:**

* **Screenshot 5.1:**
* 

**6. Edit Profile**

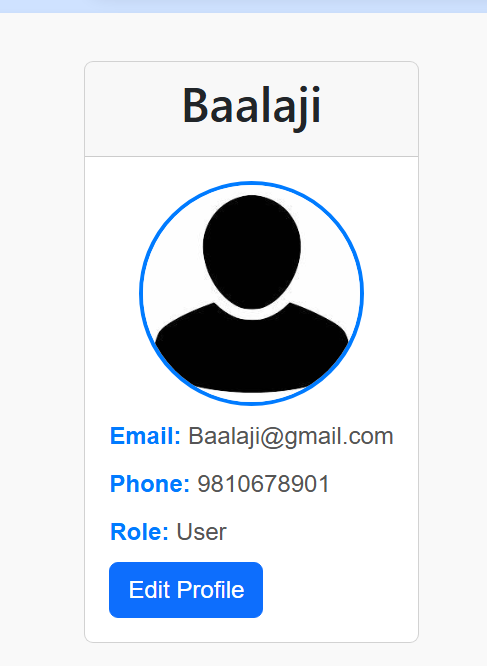
This feature allows users to manage and update their personal information and preferences.

**Description:** The "Edit Profile" section provides users with the ability to modify their account details such as name, contact information (email, phone number), profile picture, and potentially saved addresses or payment methods. It ensures users can keep their information current.

**Key Information Editable:**

* **Personal Details:** Name, email address, phone number.
* **Profile Picture:** Option to upload or change their profile image.
* **Saved Addresses:** Management of frequently used pickup/drop-off locations.
* **Payment Methods:** Adding, updating, or removing stored payment options.
* **Password:** Option to change the account password (often requires current password for security).

**Screenshots to Include:**

* **Screenshot 6.1:**
* 
* **Screenshot 6.2:**

