AVL challenge task - Vehicle Rental Platform

```
author: Toni Polanec
date: 2025-04-28

for: AVL-AST.doo
    - Backend Engineer position challenge task
```

Overview

This backend application serves as the core of a connected vehicle rental platform.

- Processes vehicle telemetry data (odometer readings, battery levels)
- Manages vehicles and customers
- Handles rental bookings with price calculations
- Provides reporting endpoints for rental metrics

Technologies Used

- **Backend**: .NET 8 Web API
- **Database**: PostgreSQL
- **ORM**: Entity Framework Core
- **API Documentation**: Swagger
- **Architecture**: Layered Architecture

Getting Started

Prerequisites

- .NET 8 SDK
- Entity Framework Core CLI
- PostgreSQL

Installation

1. Clone the repository:

   ```
   git clone https://github.com/tonipolanec/avl-vehiclerentalplatform.git

   cd avl-vehiclerentalplatform
   ```

2. Configure the database connection in `appsettings.json`:

```
"ConnectionStrings": {
  "DefaultConnection":
"Host=_localhost_;Database=_VehicleRentalDB_;Username=_vr_admin_;Password=_p
assword_;Include Error Detail=true"
}
```

3. Apply database migrations:

```
dotnet ef migrations add Initial --project src\VehicleRental.Infrastructure\

dotnet ef database update --project src\VehicleRental.Infrastructure\
```

Running the Application

Using .NET CLI

```
dotnet run --project .\src\VehicleRental.API\
```

The API will be available at `https://localhost:5270/`

Database Seeding

The application can automatically seed the database with test data from:

- `vehicles.csv` - Vehicle inventory
- `telemetry.csv` - Telemetry readings

but I commented out seeding of telemetry because i wrote up **Telemetry Simulator** for that.

For seeding you have to set

```
"DatabaseOptions": {
    "SeedData": false
}
```

in `appsettings.json` to `true`.

Telemetry Simulator

Telemetry Simulator is a console application that simulates real time telemetry data for this application.

Run it with:

```
dotnet run --project .\src\VehicleRental.TelemetrySimulator\
```

in separate terminal.

It reads given telemetry CSV file and simulates sending telemetry data in **real time** (when timestamp in file matches current time) to the application.

This completely replaces the need for seeding of telemetry data from CSV. For every past telemetry, it will be sent at once, and then it will continue to send telemetry data in real time.

Architecture

I focused on Layered Architecture in this project. I wanted to make it self documenting and easy to understand.

Most used architecture principles in this project are:

- **Classic .NET architecture guidelines**
  - API has controllers for HTTP requests
  - Core only has entities and enums
  - Application has DTOs and service interfaces
  - Infrastructure has implementations of services and validators - code used for database operations and validation rules
- **Single Responsibility Principle**
  - each part of code has only one responsibility
- **Major use of interfaces**
  - for every class or service I created an interface so it can be easily scaled if needed, also this makes code less error-prone and easier to maintain
  - for example, telemetry types can be easily extended if needed

Domain Model

- **Vehicle**: Represents a rental vehicle with make, model, pricing information
- **Customer**: Represents a person who can rent vehicles
- **Rental**: Tracks a booking with start/end dates and rental metrics
- **Telemetry**: Stores vehicle data like odometer readings and battery levels

Specific tools used

- **Entity Framework Core** for database operations
- **PostgreSQL** for the database
- **Swagger** for API documentation
- **XUnit** for unit testing
- **Postman** for API testing

Features

- Vehicle, Customer, Telemetry and Rental management (CRUD)
- Telemetry data processing, validation and storage
- Rental price calculation
    - price is calculated at the completion of the rental (*finishRental* endpoint)
    - price is calculated based on distance, duration and battery usage as per requirements
- Validation rules to prevent invalid rentals
    - rental dates must not overlap
    - vehicle must exist and be available
    - customer must exist and be available
- File and console logging

```
2025-04-28 19:55:11.18 [INF] --------------------------------
2025-04-28 19:55:11.22 [INF] Starting Vehicle Rental API
2025-04-28 19:55:11.22 [INF] --------------------------------
2025-04-28 19:38:14.09 [INF] Telemetry processed successfully for vehicle 2
2025-04-28 19:38:14.09 [INF] Telemetry processed successfully for vehicle 1
2025-04-28 19:38:14.12 [INF] Telemetry processed successfully for vehicle 2
2025-04-28 19:40:31.93 [INF] Customer created with id 50
2025-04-28 19:40:34.38 [INF] Customer created with id 51
2025-04-28 19:40:37.83 [INF] Rental created with id 22
2025-04-28 19:40:45.49 [ERR] Invalid operation: Vehicle is already rented during
the requested period
2025-04-28 19:40:50.54 [INF] Rental completed with id 22
2025-04-28 19:40:59.05 [INF] Rental deleted with id 22
2025-04-28 19:41:03.78 [INF] Customer deleted with id 50
2025-04-28 19:41:07.06 [INF] Customer deleted with id 51
```

API Documentation

The API documentation is available via Swagger UI when the application is running at `https://localhost:5270/swagger`

Also API documentation is also available offline in generated HTML file: `api_documentation.html`

Key Endpoints:

- `/api/vehicles` - Vehicle management
- `/api/customers` - Customer management
- `/api/rentals` - Rental management

Testing

Unit Tests

I wrote core unit tests for the API and the application logic. Those test can be found in the `tests` folder.

Run the unit tests using:

```
dotnet test
```

or using your IDE.

Postman Test Collection

Also I made Postman collections for the API. Those can be found in the `postman-setup` folder:

- `Vehicle Rental API` - contains all the endpoints for the API
  - as Swagger is not the best tool for testing, I made a Postman collection for the whole API
- `Vehicle Rental Testing` - contains a collection of requests that can be run together with one click of a button in preconfigured environment
  - this mimicks integration tests, not all endpoints are included as this was only for demo purposes

You can import the collections into Postman and use them to test the API.

Future Improvements

- add aggregation endpoints for usage analytics (graphs for battery usage, odometer readings, etc.)
- add authentication and authorization
- additional test coverage