Car Rental Management System

This program is designed to manage the cars of a car rental agency, with the primary goal of not only maintaining information about the cars and users but also recording the rental agreements for each car. Given the expanding nature of this business, APIs are provided separately to be used in various other applications such as mobile apps, web platforms, etc.

Unfortunately, due to the time constraints of this test, I was unable to write a web-based application to utilize the mentioned services. I apologize for this, and kindly request you to pay attention to the quality of the microservice code instead.

For the implementation of this program, I chose C# as I believe Object-Oriented Design is the best programming paradigm when implementing business logic, and I have greater proficiency in C#. Regarding the architecture, I decided to use Clean Architecture because this program is expected to grow gradually, and writing tests for such applications is crucial, which is one of the main advantages of this architecture (currently, 21 unit tests have been written for the domain of this program). Considering the relationships between entities in this program, I chose a SQL Server database. Among relational databases, SQL Server has the best compatibility with .NET applications, and Entity Framework, which can speed up software development given our limited time.

As mentioned, testability is crucial in such applications. For better readability of the tests, we have written the largest challenge in designing this system was the contract entity. This entity could be placed in both aggregates of user and car or even considered as an independent aggregate itself (for the relationship between the car and the customer). Based on our current business, I considered this entity as an entity of the car. However, if this business continues, there is a possibility that it will change.

As mentioned, testability was very important to me in designing this program, which is why I tried to structure it in a way that simplifies testing. Therefore, alongside the test files, I have created a Builder class for each entity to avoid dealing with object creation complexities in testing and to have cleaner test code. This makes it easier to maintain tests in the long run. Additionally, I used the Factory Method Pattern for all entities, as it encapsulates the creation of complex entities, leading to more readable code. Also, if there is a need for IO-Validation during object creation, it is done in this method.