GT Motive - Prueba Técnica .NET
Documentación del Proyecto

Índice

1. Introducción 3
2. Arquitectura Clean 3
3. Capa Domain 4
   - 3.1 Entities y Aggregates 4
   - 3.2 Interfaces 6
   - 3.3 Common 6
   - 3.4 Factories 6
   - 3.5 Repositories y Services 7
   - 3.6 Value Objects 7
   - 3.7 Specifications 8
4. Capa Application 8
   - 4.1 Interfaces 8
   - 4.2 Casos de Uso 9
5. Capa Infraestructure 9
   - 5.1 Unit of Work 10
   - 5.2 Bases de Datos 11
   - 5.3 Contexto de Base de Datos 11
   - 5.4 Infraestructure Configurations 11
6. Capa Api 12
   - 6.1 UseCases 12
   - 6.2 Controllers 13
   - 6.3 UserInterfaceExtensions 13
7. Capa Host 14
   - 7.1 Fichero Program.cs 14
   - 7.2 Ejecución y pruebas de la aplicación 14
8. Tests 14
   - 8.1 Unit Tests 15

8.2 Infrastructure Tests 15
8.3 Functional Tests 15
9. Mejoras 16

1. Introducción

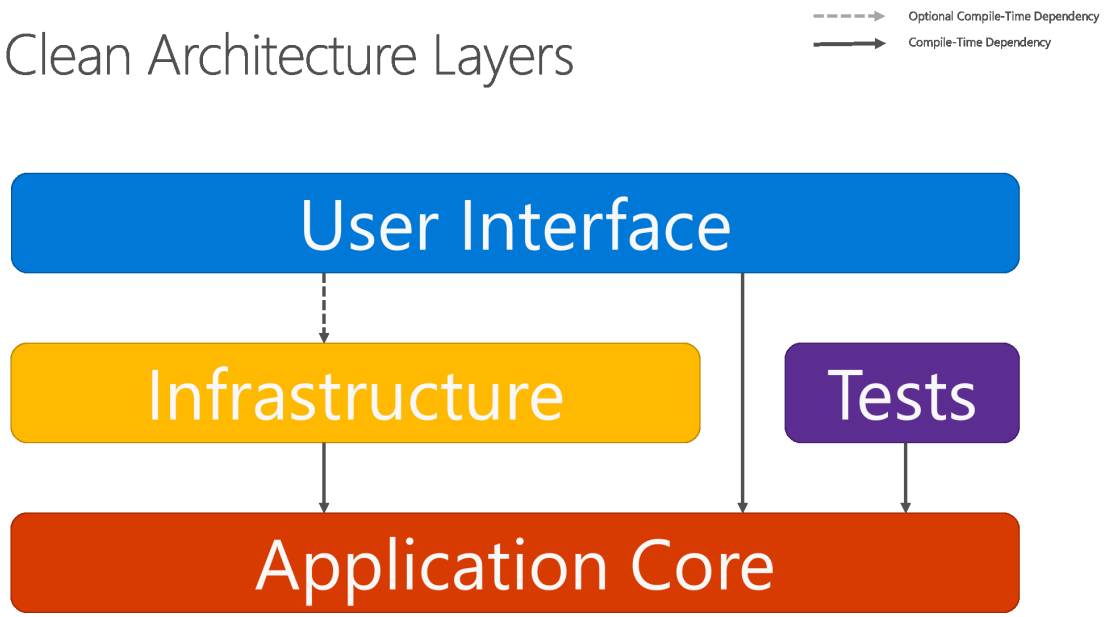
Este documento detalla el diseño arquitectónico y funcional seguido en el proyecto **GtMotive.Estimate.Microservice**, así como algunos de los patrones de diseño aplicados. Este proyecto proporciona una plataforma integral para la gestión de alquileres de vehículos, abarcando desde la reserva inicial hasta la devolución del vehículo.

La aplicación se ha desarrollado utilizando una **Arquitectura Clean**, que se caracteriza por su separación clara de responsabilidades, la independencia de capas y su facilidad para mantener y escalar el código.

2. Arquitectura Clean

"Clean Architecture" es un concepto popularizado por Robert Cecil Martin conocido como "Uncle Bob". Clean Architecture, se compone de un conjunto de patrones, prácticas y principios para crear una arquitectura de software que sea simple, comprensible, flexible, comprobable y mantenible.

Es un enfoque de diseño de software que busca separar las preocupaciones y mantener un código limpio y modular. Se basa en los principios de diseño **SOLID** y promueve la independencia de las capas del sistema.



En este proyecto, se ha adoptado la arquitectura Clean para garantizar una alta cohesión y un bajo acoplamiento entre los componentes del sistema. Esto permite gestionar de manera efectiva la complejidad del código y facilita la prueba y la evolución del sistema a lo largo del tiempo.

Clean Architecture y **Arquitectura Hexagonal** son conceptos diferentes, aunque comparten algunas similitudes y principios fundamentales. Ambos enfoques buscan promover la independencia de la lógica de negocio respecto a los detalles de implementación técnica y la separación de responsabilidades, pero difieren en la manera en que organizan esas separaciones y enfoques.

3. Capa Domain

En la capa de dominio del sistema, se definen las entidades y los agregados que representan los conceptos fundamentales del negocio. La integridad del dominio del negocio se mantiene mediante la aplicación de reglas y restricciones dentro de los agregados. Además, se ha considerado la implementación de objetos de dominio para trasladar lógica de negocio fuera de los casos de uso.

3.1 Entities y Aggregates

Customer y **Vehicle** son entidades, ya que tienen una identidad propia. **Rental** es un agregado, ya que encapsula una transacción de negocio que involucra a varias entidades.

La clase Rental sería un agregado en el contexto del diseño de dominio. Un agregado es un patrón en el diseño de dominio, donde un grupo de entidades se tratan como una sola unidad para cualquier operación de persistencia.

En este caso, Rental es un agregado porque encapsula las operaciones de negocio que involucran a varias entidades, como son en este caso a un Customer y a un Vehicle. Estas operaciones pueden incluir la creación de un nuevo alquiler, la actualización del estado de un alquiler, y el cálculo del costo total de un alquiler. Al tratar Rental como un agregado, nos aseguramos de que estas operaciones se realicen de manera atómica y consistente. Vehicle controla en su propio constructor que no puedan crearse vehículos de más de 5 años.

Además, en un agregado, hay una entidad raíz que controla el acceso a las demás entidades del agregado. En este caso, Rental sería la entidad raíz, y Customer y Vehicle serían entidades que forman parte del agregado. Esto significa que cualquier interacción con Customer y Vehicle en el contexto de un alquiler debe pasar a través de la entidad Rental.

Esto ayuda a mantener la integridad del dominio del negocio, ya que las reglas de negocio y las restricciones de integridad se pueden aplicar dentro del agregado.

Se ha considerado implementar un agregado que represente una flota. Una flota sería una colección de vehículos que pertenecen a una misma compañía o servicio de alquiler. La clase **Fleet** sería un agregado en el contexto del diseño de dominio. Fleet sería la entidad raíz y Vehicle serían entidades que forman parte del agregado.

También se ha considerado mover lógica de negocio a los objetos de dominio. Por ejemplo, se han añadido los métodos **CanRent** y **IsAvailable** a los objetos Customer y Vehicle respectivamente. Esto hará que el caso de uso donde se utilice sea más ligero y se centre más en orquestar el flujo de la operación, mientras que los objetos de dominio o los servicios de dominio se encargan de las reglas de negocio.

Para definir y documentar de manera algo más concisa el modelo de datos de la aplicación, al no ser experto en el mundo de la automoción, se ha realizado una pequeña investigación para recopilar información y las características más importantes de cada entidad.

Para hacer la clase Vehicle más completa, se han considerado unas cuantas propiedades. A destacar **VehicleType** y **VehicleStatus** que son enumeraciones. **ManufactureDate** representa la fecha en que el vehículo fue fabricado y **FirstRegistrationDate** representa la fecha en que el vehículo fue matriculado por primera vez. Estas fechas podrían ser útiles por ejemplo para calcular la edad del vehículo o para determinar cuándo es necesario realizar el mantenimiento. También se ha añadido el VIM (Vehicle Identification Number), que es un código alfanumérico que identifica de manera específica y única a cada coche.

En la arquitectura Clean, generalmente se recomienda tener un solo constructor en las entidades de dominio que toma todas las propiedades necesarias para que la entidad sea válida. Esto ayuda a garantizar que la entidad siempre esté en un estado válido y reduce la posibilidad de errores. Esto significa que se deberían pasar todas las propiedades que sean necesarias para que la entidad sea válida a través del constructor. Se incluye también un por defecto un constructor privado sin parámetros para poder enlazar las entidades y agregados con el EntitiyFramework.

Por ejemplo, en el caso del vehículo, según un criterio personal, se podrían considerar que las propiedades como Model, Brand, Color, ManufactureDate y VIN son obligatorias. Estas propiedades se han marcado como private set para

garantizar que solo se puedan cambiar dentro de la clase Vehicle, lo que ayuda a mantener la integridad del estado de la entidad.

Además, en este tipo de arquitecturas, generalmente se recomienda mantener las entidades de dominio lo más simples posible y libres de cualquier lógica que no sea de dominio. Esto incluye por ejemplo los decoradores de validación de datos, que son específicos de la infraestructura y no forman parte de la lógica de dominio.

3.2 Interfaces

En la Clean Architecture, las entidades y agregados del dominio deben contener la lógica de negocio y no deben depender de detalles de infraestructura, como la persistencia de datos. Además, estos suelen ser clases concretas en lugar de interfaces, ya que representan los conceptos del dominio del negocio y su comportamiento

3.3 Common

Es común tener una clase base que contenga propiedades comunes a todas las entidades y agregados, como es en este caso la **EntityBase**. Es una clase abstracta con una propiedad Id de tipo **Guid**. El Id se genera automáticamente en el constructor de la clase, por lo que cada vez que se crea una nueva entidad o agregado, se le asignará un Id único. Luego, las entidades y agregados pueden heredar de EntityBase para obtener la propiedad Id.

3.4 Factories

Las Factories pueden ser útiles para crear instancias de entidades o agregados con un estado inicial específico. En nuestro caso, se ha agregado un "Factory" para Fleet y Rental, suponiendo que puedan tener una lógica de creación más compleja; por ejemplo, que al crear un Rental se necesite validar que el Vehicle está disponible y que el Customer tiene una licencia válida o que al crear un Fleet se necesite realizar ciertas operaciones, como agregar un conjunto inicial de Vehicle a la flota. Sin embargo, parece que la creación de Customer y Vehicle es bastante directa y se realiza a través del constructor de cada clase por lo que no se ha considerado necesario tener un Factory para estas entidades.

La implementación de las interfaces de Factories, también deberían estar en la capa de dominio, ya que se ocupa de la creación de objetos de dominio. Se han incluido en la misma carpeta Factories.

3.5 Repositories y Services

En la Arquitectura Clean, se recomienda definir **interfaces para los repositorios** y servicios en la capa de dominio. Estas interfaces proporcionan una abstracción de las operaciones que se pueden realizar en las entidades y agregados del dominio. Las implementaciones concretas de estas interfaces se colocarían en la capa de infraestructura (o en la capa de aplicación, dependiendo de si la lógica de negocio implica interactuar con servicios externos o infraestructura)

Se incluye un repositorio genérico, ya que es una buena práctica si todos los repositorios tienen los mismos métodos básicos de CRUD. En este caso de los repositorios, sí se considera una buena práctica definir una interfaz específica para cada tipo de entidad, incluso si no tiene métodos adicionales a los de la interfaz genérica. Esto se debe a varias razones: flexibilidad para el futuro, claridad del código y facilitar la inyección de dependencias.

Por ejemplo, en el repositorio de Vehicles se define un método que comprueba si un vehículo puede ser o no creado y que utiliza el método del constructor que hace la comprobación de la fecha de fabricación. Se implementa este método del repositorio con idea de que si se establecen más criterios adicionales puedan situarse ahí.

Los servicios de dominio pueden ser útiles para encapsular la lógica de negocio que no encaja naturalmente en una entidad o valor de objeto. Estos servicios de dominio trabajan con entidades y agregados para realizar operaciones de negocio.

No todas las entidades o agregados necesitan un servicio de dominio. Solo suele ser necesario un servicio de dominio si se tiene lógica de negocio que no encaja en una entidad o valor de objeto.

3.6 Value Objects

Son clases que representan conceptos simples en el dominio. Son objetos inmutables que no tienen identidad y se utilizan principalmente para describir aspectos del dominio. Para el agregado **Rental**, se ha incluido un objeto de valor **RentalPeriod**. Este objeto de valor representa el periodo de tiempo durante el

cual un vehículo está alquilado; incluye una verificación en el constructor para asegurarse de que la fecha de inicio es anterior a la fecha de fin, y también sobrescribe los métodos Equals y GetHashCode para proporcionar una comparación de igualdad basada en el valor.

3.7 Specifications

El **patrón de especificación** se utiliza comúnmente en la capa de dominio, ya que encapsula las reglas de negocio que se aplican a los objetos de dominio. En nuestro caso, se ha creado una especificación **CustomerCanRentSpecification**, que encapsula la regla de que un cliente puede alquilar si no tiene un alquiler existente.

Además, se ha creado una interfaz **ISpecification**, que todas las especificaciones deben implementar.

Estos archivos proporcionan una implementación básica del patrón de especificación. También se podría extender este patrón para soportar especificaciones compuestas (usando operadores lógicos como AND y OR), para encapsular las consultas a la base de datos, etc.

4. Capa Application

La capa de Aplicación es la que coordina las operaciones de alto nivel que implican objetos de dominio. Esta capa contiene la lógica de negocio que no pertenece a los objetos de dominio y generalmente se implementa en servicios de aplicación. Estos servicios de aplicación orquestan la ejecución de la lógica de negocio y delegan en los objetos de dominio para las operaciones de negocio de bajo nivel.

4.1 Interfaces

Se utilizan varias interfaces que son comunes y que ya estaban definidas en el proyecto y son suficientes para implementar los **casos de uso**. Además, se considera necesario definir algunas interfaces adicionales. Por ejemplo, las IXXXOutputPort y las IXXXUseCase.

4.2 Casos de Uso

Los UseCases (o Casos de Uso) en la capa de aplicación son una parte fundamental de la arquitectura limpia y la arquitectura hexagonal. Son una herramienta poderosa para organizar y estructurar la lógica de negocio en la aplicación. Encapsulan la lógica de negocio específica de una operación o proceso. Esto permite que esté bien definida, sea fácil de entender, fácil de probar y esté separada de otras preocupaciones, como la persistencia de datos o la interfaz de usuario. También fomentan la reutilización de código y el Desacoplamiento

En este proyecto, se realiza la implementación de los casos de uso específicos aprovechando las interfaces existentes (IUseCase, IUseCaseInput, IUseCaseOutput, IOutputPortStandard e IOutputPortNotFound).

En estas implementaciones de los casos de uso, se realizan llamadas a los repositorios definidos mediante interfaces en la capa Domain. En algunos casos se utilizan los métodos CRUD de la interfaz genérica **IGenericRepository**. A destacar también la utilización del puerto de salida **IOutputPortNotFound**, que ya venía implementado. Se utiliza para manejar algunos casos de error, por ejemplo en los que el vehículo no existe o no está disponible, o el cliente ya tiene un alquiler, etc.

Se han utilizado algunas prácticas recomendadas. Por ejemplo, para el caso de uso de CreateRental, se pasa parte de la lógica de negocio a los CanRent y IsAvailable de los objetos Customer y Vehicle respectivamente, en la capa Domain. Se ha utilizado el **patrón de diseño de especificación** para encapsular las reglas de negocio en **CustomerCanRentSpecification** utilizando su "IsSatisfiedBy". Y se ha utilizado el patrón de diseño de Factory para encapsular la lógica de creación de Rental.

El proyecto podría ampliarse en un futuro. Por ejemplo, se podrían necesitar más casos de uso para otras operaciones, como actualizar un alquiler, eliminar un alquiler, etc.

5. Capa Infraestructure

La capa de Infraestructure es la responsable de **implementar las interfaces** definidas en la capa de Aplicación. Por ejemplo, si hay interfaces para repositorios en la capa Application, se implementarían en esta capa. Esto podría

implicar escribir código que interactúe con una base de datos para guardar y recuperar entidades.

Además, esta capa en una aplicación típica de DDD (Domain-Driven Design) incluye todas las implementaciones de los repositorios definidos en la capa de Dominio, así como cualquier otra lógica relacionada con la persistencia de datos, como la configuración de la base de datos, las migraciones de la base de datos, la comunicación de red y cualquier otra operación técnica necesaria para ejecutar la aplicación.

Debido a que los requisitos del proyecto es trabajar sin dependencias externas y para simplificar, se ha utilizado **Entity Framework Core InMemory** para trabajar con la persistencia de datos en memoria. Esto es suele ser útil para pruebas o para prototipos rápidos donde no se necesita una base de datos persistente. Se realiza con la opción **UseInMemoryDatabase** en el método **ConfigureServices** de la clase **Startup**

5.1 Unit of Work

La interfaz IUnitOfWork que ya existe en el proyecto es bastante estándar y debería ser suficiente para la mayoría de los casos de uso. Esta interfaz proporciona un método **Save** que se utiliza para confirmar todos los cambios en la base de datos, que es el comportamiento principal que se espera de un **patrón Unit of Work**.

Sin embargo, en este caso es necesario trabajar adicionalmente con repositorios específicos (como un IRentalRepository), por lo que se han añadido esas propiedades en la interfaz IUnitOfWork. Esto permite a los casos de uso obtener acceso a los repositorios a través del IUnitOfWork, en lugar de inyectar cada repositorio individualmente.

También hay que comentar que, aunque es común ver la **interfaz IUnitOfWork** en la capa de Dominio, suele ser conveniente situarla en la capa Application, ya que es más un concepto de infraestructura. Por tanto, en esta solución **se ha movido a esa capa**.

La **implementación** de UnitOfWork en la capa Infraestructure crea instancias de los repositorios de forma perezosa, es decir, las instancias de los repositorios no se crean hasta que se accede a ellas por primera vez. Esto puede mejorar el rendimiento al evitar la creación innecesaria de objetos. Las propiedades del repositorio son privadas en esta implementación del UnitOfWork, pero aún deben ser públicas en la interfaz IUnitOfWork para que puedan ser utilizadas en otras partes de la aplicación.

Además, se ha añadido una comprobación en el método Dispose para evitar deshacerse del contexto de la base de datos más de una vez. Se ha añadido un método Dispose(bool) que realiza la lógica de disposición. El método Dispose() llama a Dispose(true) y luego a GC.SuppressFinalize(this). Esto asegura que los recursos sean liberados correctamente y que el recolector de basura no intenta finalizar el objeto innecesariamente.

5.2 Bases de Datos

Se ha optado por utilizar como solución de persistencia **Entity Framework Core en memoria**. Se han instaldo los NuGet de Microsoft.EntityFrameworkCore y Microsoft.EntityFrameworkCore.InMemory. También ha sido necesario el Microsoft.EntityFrameworkCore.Relational, ya que estaba dando un error sin indicar que era porque faltaba este paquete.

Se han desinstalado los nuget de MongoDB, al optar por la otra solución de persistencia y no utilizarse. Además, la versión 2.17.0 que tenía el proyecto contaba con una vulnerabilidad que dificultaba la instalación de otros NuGet.

5.3 Contexto de Base de Datos

Se crea la clase **GtMotiveContext**, que incluye todos los **DbSet** necesarios para las entidades y agregados. Además, se añade un método **InitializeData** para hacer una carga inicial de algunos valores.

5.4 Infraestructure Configurations

Este archivo se utiliza para configurar y registrar los servicios y las dependencias que se utilizarán en la aplicación.

Se agrega un contexto de base de datos **GtMotiveContext** a los servicios y configura este contexto para usar una base de datos en memoria llamada **"GtMotiveDB"**.

Registra varios repositorios y la UnitOfWork con el contenedor de inyección de dependencias. Estos repositorios abstraen las operaciones de la base de datos para diferentes entidades como Vehicle, Customer, Rental y Fleet.

En resumen, este archivo se encarga de configurar los servicios y las dependencias que se utilizarán en la aplicación, y de registrarlos con el contenedor de inyección de dependencias.

6. Capa Api

La capa Api contiene la lógica relacionada con la **gestión de las solicitudes HTTP** y la **generación de respuestas HTTP**. Esta capa depende de los servicios proporcionados por la capa **ApplicationCore** y la capa **Infrastructure**.

Para seguir las prácticas recomendadas en una Clean Architecture, se estructura esta capa la siguiente manera:

- **Controllers**: cada controlador se encargará de manejar las solicitudes HTTP para un recurso específico (por ejemplo, VehiclesController, RentalsController, etc.). Los controladores no deben contener lógica de negocio. En su lugar, deben delegar en los casos de uso para realizar el trabajo real.
- **UseCases**: cada caso de uso se encargará de una operación específica que la aplicación puede realizar (por ejemplo, CreateVehicle, ListAvailableVehicles, RentVehicle, ReturnVehicle). Los casos de uso deben ser independientes entre sí y cada uno debe tener una única responsabilidad.

Dentro de la carpeta **UseCases**, se incluye lo siguiente:

- Una clase que representa el caso de uso.
- Una clase que recibe la solicitud HTTP.
- Una clase que representa la solicitud del caso de uso (por ejemplo, los datos que se necesitan para crear un vehículo).
- Una clase que representa la respuesta del caso de uso (por ejemplo, los datos que se devuelven cuando se lista los vehículos disponibles).

6.1 UseCases

En una arquitectura limpia, los casos de uso en la capa de Aplicación representan las operaciones que la aplicación puede realizar, independientemente de cómo se acceda a ellas. Estos casos de uso no deben saber nada sobre HTTP, JSON, etc. Solo deben centrarse en la lógica de negocio.

Por otro lado, los casos de uso en la capa Api son **específicos de la interfaz de usuario** (en este caso, una API HTTP). Estos casos de uso se encargan de manejar las solicitudes HTTP, convertirlas en solicitudes para los casos de uso de la capa de Aplicación, ejecutar esos casos de uso y luego convertir las respuestas en respuestas HTTP. Es decir, son una especie de **"adaptadores"** que

convierten las solicitudes y respuestas HTTP en solicitudes y respuestas para los casos de uso de la capa de Aplicación, y viceversa.

Aquí es donde entran en juego los ficheros Handler, Presenter, Request y Response:

- **Handler**: componente que recibe la solicitud HTTP, la convierte en una solicitud para el caso de uso de la capa de Application y luego ejecuta ese caso de uso.
- **Presenter**: componente que toma la respuesta del caso de uso de la capa de Application y la convierte en una respuesta HTTP.
- **Request**: clase que define los datos que se necesitan para ejecutar el caso de uso en la capa Api. Esta clase puede ser similar a la clase de solicitud del caso de uso de la capa de Aplicación, pero también puede incluir detalles específicos de HTTP, como los códigos de estado, las cabeceras, etc.
- **Response**: clase que define los datos que el caso de uso en la capa Api devuelve cuando se ejecuta con éxito. Al igual que la clase Request, esta clase puede ser similar a la clase de respuesta del caso de uso de la capa de Aplicación, pero también puede incluir detalles específicos de HTTP.

6.2 Controllers

En los controladores de la capa Api se implementan varios métodos para manejar las diferentes solicitudes HTTP relacionadas con la gestión de vehículos, alquileres y flotas.

Todos estos métodos utilizan el **patrón Mediator** con **MediatR** para manejar las solicitudes, lo que permite desacoplar el código que envía una solicitud del código que la maneja.

6.3 UserInterfaceExtensions

En la clase UserInterfaceExtensions, hemos configurado el AddPresenters para registrar los presentadores en la colección de servicios. Esto permite que los presentadores sean inyectados automáticamente en los controladores y otros servicios que los necesiten. También se configuran los OutputPort y, en el caso de CreateRental, el RentalFactory y el CustomerCanRentSpecification.

7. Capa Host

En una Arquitectura Clean, la capa Host es la **capa más externa** y es responsable de interactuar con el mundo exterior, es decir, es la capa que "enciende" la aplicación. Esta capa contiene todo el código que se necesita para poner en marcha la aplicación y conectarla con los sistemas externos y es la última que se ejecuta cuando se cierra.

Algunas **responsabilidades** de la capa Host son la configuración de la aplicación, **inyección de dependencias**, arranque de la aplicación e interfaz con sistemas externos.

7.1 Fichero Program.cs

El archivo Program.cs en la capa Host es el punto de entrada de la aplicación. Aquí es donde se configura y se inicia la aplicación. Se ha añadido la configuración del NewtonsoftJson para evitar referencias circulares

7.2 Ejecución y pruebas de la aplicación

Se ha configurado el fichero **launchSettings.json** para mostrar el Swagger al ejecutar.

Se ha hecho una carga inicial de las siguientes entidades, para poder probar la ejecución con el Swagger o Postman:

- **Flota** con ID=00000000-0000-0000-0000-000000000001

- **Cliente** con ID=00000000-0000-0000-0000-000000000002

- **Vehículo** con ID=00000000-0000-0000-0000-000000000003

Para esta inicialización de datos se ha utilizado el **método Add de DbSet**, que devuelve una entidad EntityEntry. Esta entidad tiene una propiedad Entity que se puede usar para establecer la ID de la entidad después de que se ha creado.

8. Tests

Los tests son una herramienta esencial para mantener la **calidad del software** en una arquitectura clean. Permiten a los desarrolladores verificar que todas las

partes de la aplicación, desde las unidades de código más pequeñas hasta las funcionalidades completas, funcionan correctamente.

8.1 Unit Tests

Estos tests se centran en pequeñas unidades de código, como funciones o métodos individuales. Su objetivo es asegurar que cada parte del código funcione correctamente de manera aislada. En el contexto de la arquitectura clean, los tests unitarios suelen centrarse en el **dominio y la lógica de la aplicación**, que son las partes más internas de la arquitectura.

Para simular el comportamiento de objetos en las pruebas unitarias no se han utilizado Mocks ya que, en el caso de la entidad Vehicle, no hay dependencias que necesiten ser simuladas porque es una clase de entidad simple con propiedades y un método.

Los Mocks son más útiles cuando se prueban unidades de código que tienen dependencias externas, como una clase de servicio que depende de una interfaz de repositorio.

8.2 Infrastructure Tests

Estos tests verifican que la infraestructura de la aplicación, como la base de datos o los servicios externos, funciona correctamente. En la arquitectura clean, estos tests suelen implicar la capa de adaptadores, que es la que interactúa con la infraestructura externa.

Se han hecho una serie de ajustes en los ficheros existentes y se ha implementado el test de infraestructura, **VehicleControllerTests**, que se basa en la clase InfrastructureTestBase. Está diseñado para probar el endpoint **ListAvailableVehicles** del controlador **VehicleController**. En el constructor de la prueba, se crea un cliente HTTP utilizando el servidor de prueba y un mock del repositorio IVehicleRepository. Este mock se configura para devolver un vehículo específico cuando se llama al método GetAvailableVehicleAsync. La prueba envía una solicitud GET al endpoint, verifica que la respuesta tiene un código de estado exitoso y que la respuesta contiene una lista de vehículos.

8.3 Functional Tests

Los tests funcionales o de integración verifican que las diferentes partes de la aplicación funcionan correctamente juntas. En lugar de probar unidades de

código individuales, los tests funcionales prueban la funcionalidad de la aplicación de principio a fin. En la arquitectura clean, estos tests pueden implicar varias capas de la arquitectura, desde la interfaz de usuario hasta la infraestructura.

La prueba de integración implementada verifica que diferentes partes del sistema funcionen correctamente cuando se combinan. En este caso, el test verifica que el VehicleRepository interactúa correctamente con la base de datos en memoria (proporcionada por Entity Framework Core) para almacenar y recuperar entidades Vehicle. Este test no incluye un host, como un servidor web, que normalmente estaría involucrado en una solicitud completa desde un cliente hasta la base de datos y de vuelta. En su lugar, el test interactúa directamente con el VehicleRepository, por lo que se centra en la integración entre esa parte específica de la aplicación y la base de datos.

9. Mejoras

Durante el desarrollo, se han identificado algunas mejoras que con más tiempo a dedicar se podrían implementar. Algunas de ellas serían:

- Comprobar que el cliente es mayor de 18 años

- Comprobar que el cliente tiene carnet de conducir

- Comprobar que el cliente tiene un modo de pago válido

- Mejoras de gestión de excepciones

- Mejorar criterios de comprobación

- Añadir más casos de uso

- Mejorar definición de columnas y restricciones a nivel de base de datos

- Incluir más relaciones entre tablas a nivel de bases de datos

- Configurar la autenticación y la autorización de la aplicación

- Configurar cualquier otra infraestructura que la aplicación pueda necesitar, como servicios de logging, validación, etc.

- Implementar el patrón Identity Map para mejorar el rendimiento con una caché en memoria de los objetos que se han cargado de la BD

- Resolver las vulnerabilidades de algunos paqutes de NuGet de la solución

- Uso de Automapper
- Añadir modelos de datos de cache para mejorar el rendimiento