Domain-Driven Design with ASP.NET Core

Building an App with Clean Architecture Concepts

Car Rental System

Created by Ivaylo Kenov for the Code It Up Initiative

Required Software

- Visual Studio 2019 Community
- .NET Core 3.1
- SQL Server Developer Edition
- Optional: Latest NodeJS LTS
- Optional: Angular CLI

System Requirements – Car Rental Dealers

We need to design a system in which car dealers can publish their cars for rent. Each car ad must contain manufacturer, model, category, image, and price per day. Categories have a description and must be one of the following - economy, compact, estate, minivan, SUV, and cargo van. Additionally, each vehicle should list the following options: with or without climate control, number of seats, and transmission type.

The system should allow users to filter the cars by category, manufacturer, and price range anonymously. Ads can be sorted by manufacturer or by price.

When a user chooses a car, she needs to call the dealer on the provided phone and make the arrangement. The dealer then needs to edit the car ad as "currently unavailable" manually. The system must not show the unavailable cars.

1. Defining the Initial Domain Model

First, we need to extract our domain model from the requirements. We need to "search" for entities, value objects, or aggregates. We can define:

- *Dealer* – entity, aggregate root
- *Phone number* = value object, part of dealer aggregate
- *Car ad* – entity, aggregate root
- *Car manufacturer* – entity, part of car ad aggregate
- *Car category* – entity, part of car ad aggregate
- *Car options* – value object, part of car ad aggregate

The phone number can be a simple text property of the dealer, but since it carries lots of validation logic, it is a good idea to be extracted in a separate class – part of the dealer aggregate.

Since the car make, category and options are part of the car ad – we can mark these as one aggregate entity too. The car options do not need an identifier and work as a single unit so we will use them as a value object.

Of course, it can be argued whether the car ad is a root and whether it should be part of the dealer aggregate. There is no perfect solution, and there is not a right or wrong answer. It is subjective, depending on the developer's point of view. The value proposition we can consider here is that there will be a lot of business logic about the car ad directly, and for this reason, it should be defined as a root. Code like *carAdService.GetCarAdCategories()* makes more sense than *dealerService.GetCarAdCategories().*

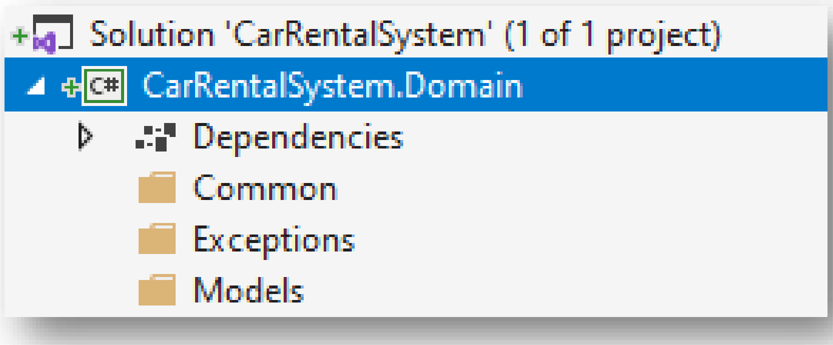
Now, open **Visual Studio 2019** and create a new blank solution named **CarRentalSystem**. Following DDD, it is advised to start your code with the domain business logic initially, without taking into consideration any UI or persistence infrastructure.

This is exactly what we are going to do. Add a new class library for **.NET Standard** to the solution. Name it **CarRentalSystem.Domain**. Open the project file and enable non-nullable reference types:

```
<PropertyGroup>
  <TargetFramework>netstandard2.0</TargetFramework>
  <LangVersion>8.0</LangVersion>
  <Nullable>enable</Nullable>
  <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
</PropertyGroup>
```

*Note: since we are going to convert this project to a unit test one in the next section, you may want to choose **.NET Core** class library initially.*

Now, delete the template class and create three project folders – **Common, Exceptions, Models**:



Before adding our domain classes, we need to define some base ones:

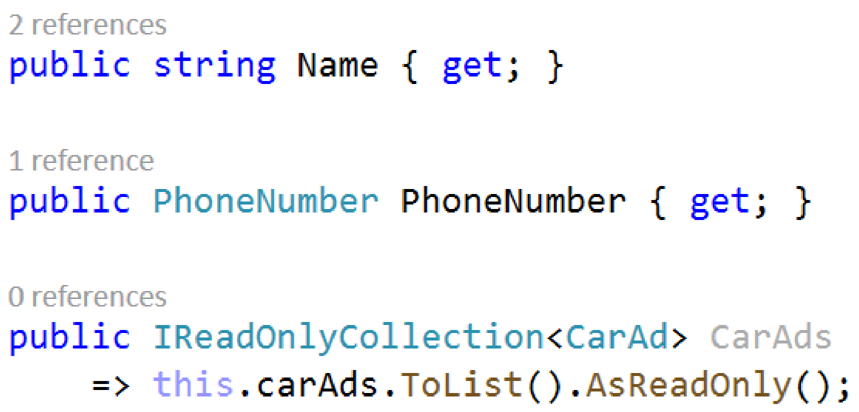
- **Entity** – contains common logic for entities – identifier and equality
- **ValueObject** – contains common logic for value objects - equality
- **Enumeration** – contains common enumeration methods
- **Guard** – contains common validation methods
- **BaseDomainException** – a base class for all domain exceptions

For brevity, these are provided for you in the **Code Helpers > Domain > Common** folder. Copy the base exception class in your project's **Exceptions** folder and copy the other four in the **Common** folder. Make sure you get to know these classes before you start using them.

Before we start implementing our classes in the **Models** folder, we need to have a convention for recognizing the different object types. For this reason, we are going to put separate aggregates in separate subfolders in the **Models** folder. This approach makes sense since aggregate parts are logically connected to each other. All other objects will be put in the main **Models** folder. We will need to have a good way to identify aggregate roots, however. A good solution is to create an empty **IAggregateRoot** marker interface, which will be implemented by all aggregate roots in our solution.

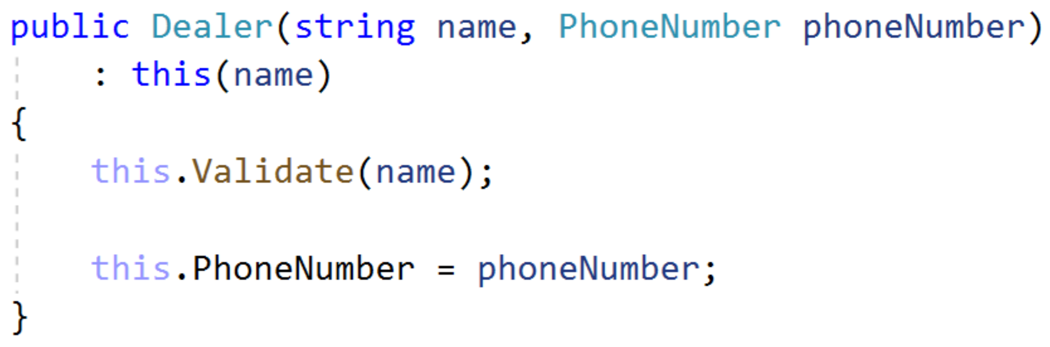
Now, it is your turn. Implement the domain classes and add validation logic as you see fit. Make sure you follow the general DDD rules:

- Do not confuse domain objects with database schema and entities. Try not to think about any persistence layer while you design the classes.
- All domain objects should be immutable and read-only through their properties. Do not expose setters or whole collections:



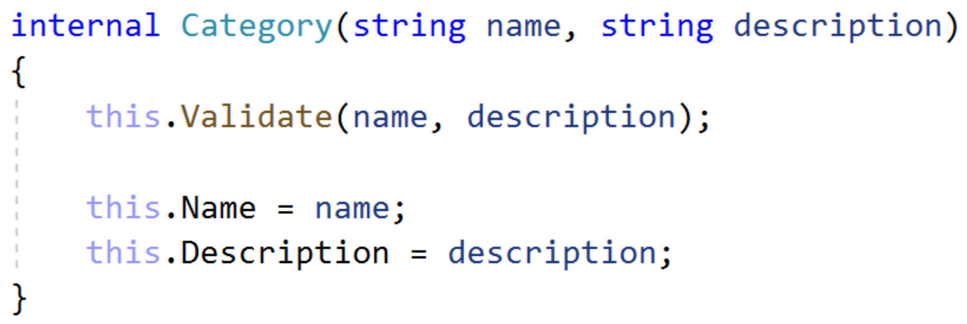
```
2 references
public string Name { get; }

1 reference
public PhoneNumber PhoneNumber { get; }

0 references
public IReadOnlyCollection<CarAd> CarAds
    => this.carAds.ToList().AsReadOnly();
```

- Constructors should create a valid object in terms of state:



```
public Dealer(string name, PhoneNumber phoneNumber)
    : this(name)
{
    this.Validate(name);

    this.PhoneNumber = phoneNumber;
}
```

- Only the aggregate root constructors should be *public*. The others should be *internal*:



```
internal Category(string name, string description)
{
    this.Validate(name, description);

    this.Name = name;
    this.Description = description;
}
```

- All mutating operations and behaviors should be done through methods:

```
3 references
public bool IsAvailable { get; private set; }

0 references
public void ChangeAvailability()
    => this.IsAvailable = !this.IsAvailable;
```

- Do not create two-way relationships. They are not needed in DDD. For example, the car ad may have a manufacturer property, but the manufacturer may not have a collection property of car ads. Relationships should be based on the business domain and the logic behind them.
- Create an exception class for each domain aggregate. Do not use generic exceptions to indicate domain-related errors:

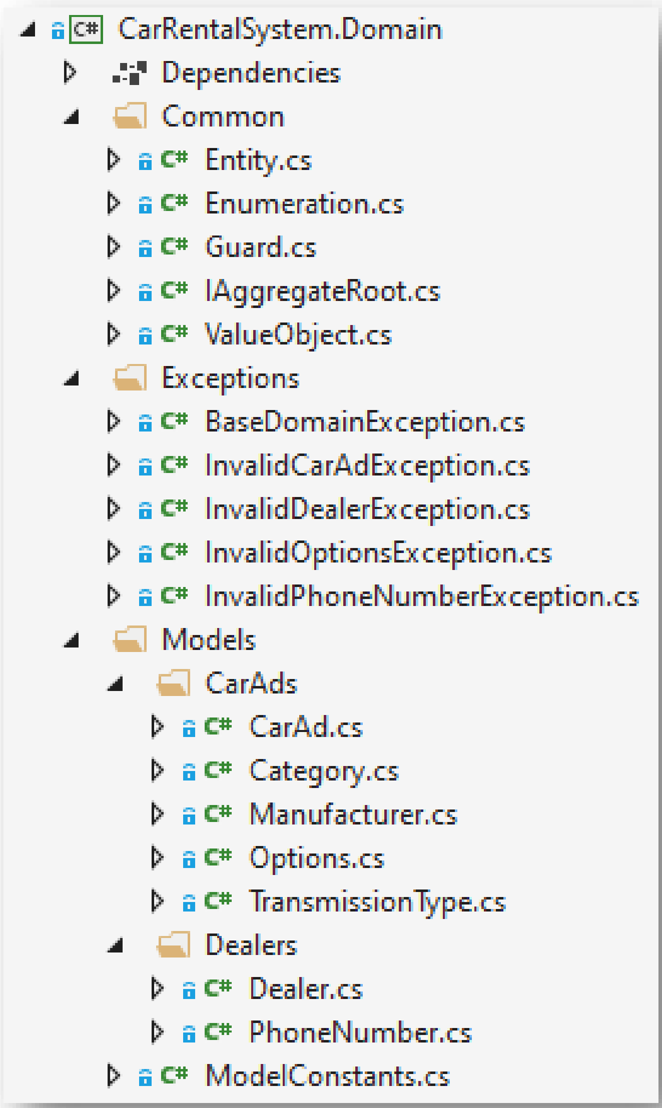
```
Guard.ForStringLength<InvalidCarAdException>(
    name,
    MinNameLength,
    MaxNameLength,
    nameof(this.Name));

Guard.ForStringLength<InvalidCarAdException>(
    description,
    MinDescriptionLength,
    MaxDescriptionLength,
    nameof(this.Description));
```

- Mark every object as an entity, a value objects, or an enumeration. Mark the aggregate roots as well:

```
public class CarAd : Entity<int>, IAggregateRoot
```

- Do not seek perfection. It is not possible to get the perfect domain model during the initial design. The classes will evolve as we add additional layers to our solution. Additionally, we may need to modify our implementations when there are new or changed project requirements.

Here is a potential project structure for our implementation:



```
CarRentalSystem.Domain
    Dependencies
    Common
        Entity.cs
        Enumeration.cs
        Guard.cs
        IAggregateRoot.cs
        ValueObject.cs
    Exceptions
        BaseDomainException.cs
        InvalidCarAdException.cs
        InvalidDealerException.cs
        InvalidOptionsException.cs
        InvalidPhoneNumberException.cs
    Models
        CarAds
            CarAd.cs
            Category.cs
            Manufacturer.cs
            Options.cs
            TransmissionType.cs
        Dealers
            Dealer.cs
            PhoneNumber.cs
        ModelConstants.cs
```

You can find an implementation example of this section in the **Partial Solutions > 1. Defining the Initial Domain Model** folder.

2. Domain Model Unit Tests

The domain model is perfect for unit tests. The usual logic under test is the one with the object validation and the state mutability operations.

We are going to write the test cases directly in the source project. We are not going to separate them into different assemblies. These are the benefits of this approach:

- *Better encapsulation* – internal members are easier to validate.
- *Better maintainability* – developers can comfortably navigate between the component under test and the assertion code.
- *Refactoring* – moving classes in between folders and syncing them with the test structure is not a cumbersome task.
- *Code coverage* – code without tests is spotted faster.

If you do not like this approach, feel free to introduce a separate test assembly in your solution.

Now, let us write some tests. For educational purposes we are not going to aim for high code coverage. We will just show a few concepts and examples.

First, we need to install a test runner and an assertion framework – *xUnit* and *FluentAssertions* are good enough. The problem with installing these frameworks in the source project is that the test code and assemblies will be deployed to production. The fix is easy – we are going to use conditional compilation in our solution configuration.

Additionally, we need to convert the target framework to **netcoreapp3.1** to run our tests.

```xml
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <LangVersion>8.0</LangVersion>
    <Nullable>enable</Nullable>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  </PropertyGroup>

  <ItemGroup Condition="'$(Configuration)' == 'Release'">
    <Compile Remove="**\*.Specs.cs" />
    <Compile Remove="**\*.Fakes.cs" />
  </ItemGroup>

  <ItemGroup Condition="'$(Configuration)' != 'Release'">
    <PackageReference Include="FluentAssertions" Version="5.10.3" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.6.1" />
    <PackageReference Include="xunit" Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.1" />
  </ItemGroup>
```

The above configuration is provided in the **Code Helpers > Domain** folder. You may copy it from there.

After our setup is ready and the project compiles - we are going to assert the validation logic for the car category domain object. Add a **Category.Specs.cs** C# file next to the **Category** one and write the following *Arrange-Act-Assert* unit tests:

```csharp
public class CategorySpecs
{
    [Fact]
    0 references
    public void ValidCategoryShouldNotThrowException()
    {
        // Act
        Action act = () => new Category("Valid name", "Valid description text");

        // Assert
        act.Should().NotThrow<InvalidCarAdException>();
    }

    [Fact]
    0 references
    public void InvalidNameShouldThrowException()
    {
        // Act
        Action act = () => new Category("", "Valid description text");

        // Assert
        act.Should().Throw<InvalidCarAdException>();
    }
}
```

Now, let us assert the state mutability for the car ad object. Add a **CarAd.Specs.cs** C# file next to the **CarAd** one and write this test:

```
public class CarAdSpecs
{
    [Fact]
    0 references
    public void ChangeAvailabilityShouldMutateIsAvailable()
    {
        // Arrange
        var carAd = new CarAd(
            new Manufacturer("Valid manufacturer"),
            "Valid model",
            new Category("Valid category", "Valid description text"),
            "https://valid.test",
            10,
            new Options(true, 4, TransmissionType.Automatic),
            true);

        // Act
        carAd.ChangeAvailability();

        // Assert
        carAd.IsAvailable.Should().BeFalse();
    }
}
```

The above arrange section is quite "unpleasant". Let us improve it by installing *FakeItEasy*:

```
<ItemGroup Condition="'$(Configuration)' != 'Release'">
  <PackageReference Include="FakeItEasy" Version="6.0.1" />
  <PackageReference Include="FakeItEasy.Analyzer.CSharp" Version="6.0.0" />
  <PackageReference Include="FluentAssertions" Version="5.10.3" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.6.1" />
  <PackageReference Include="xunit" Version="2.4.1" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.4.1" />
</ItemGroup>
```

Then, create a **CarAd.Fakes.cs** C# file next to the **CarAd** one and add the following dummy factory:

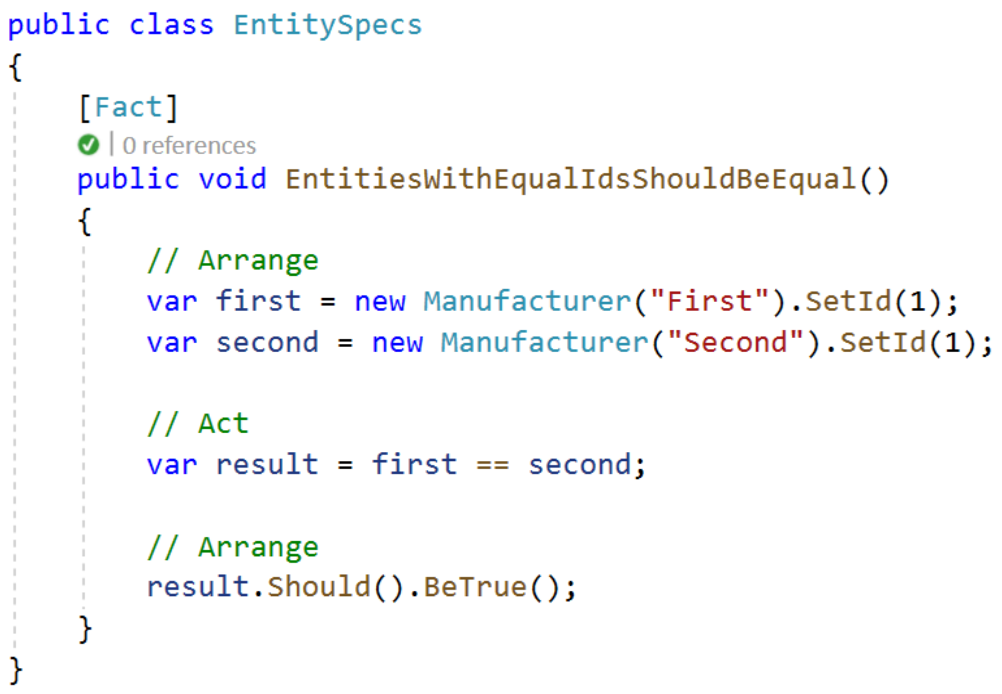
```
public class CarAdFakes
{
    0 references
    public class CarAdDummyFactory : IDummyFactory
    {
        0 references
        public bool CanCreate(Type type) => true;

        0 references
        public object? Create(Type type)
            => new CarAd(
                new Manufacturer("Valid manufacturer"),
                "Valid model",
                new Category("Valid category", "Valid description text"),
                "https://valid.test",
                10,
                new Options(true, 4, TransmissionType.Automatic),
                true);

        0 references
        public Priority Priority => Priority.Default;
    }
}
```

From now on, every time we need a dummy car ad instance, we can just write **A.Dummy<CarAd>()**, like so:

```
public class CarAdSpecs
{
    [Fact]
    ✓ | 0 references
    public void ChangeAvailabilityShouldMutateIsAvailable()
    {
        // Arrange
        var carAd = A.Dummy<CarAd>();

        // Act
        carAd.ChangeAvailability();

        // Assert
        carAd.IsAvailable.Should().BeFalse();
    }
}
```

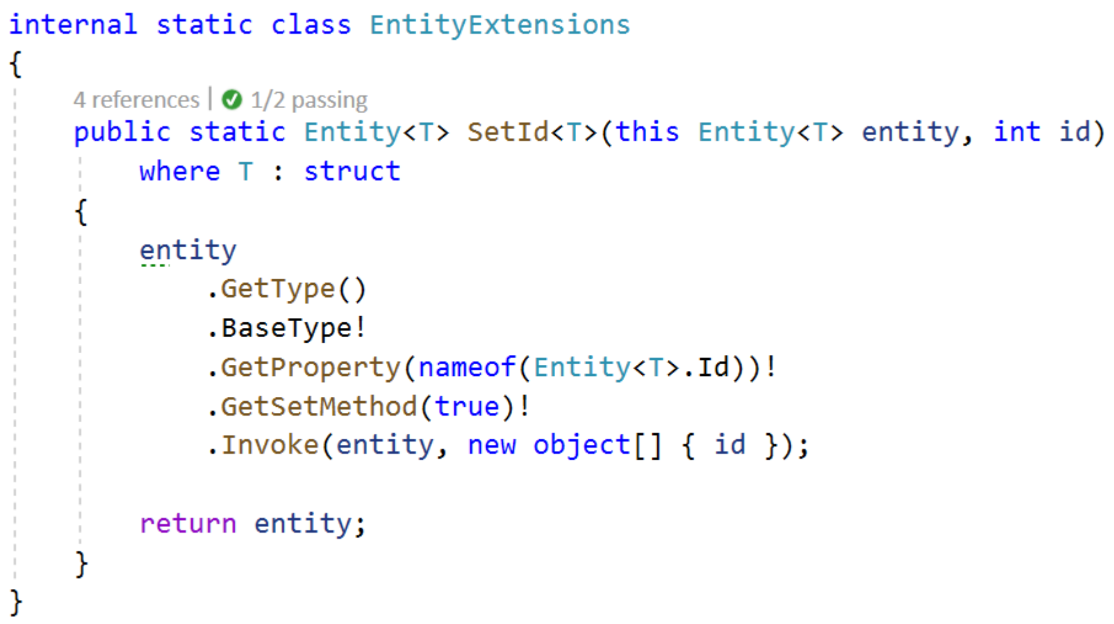
Much cleaner!

It is your turn to write tests. Add at least one for a validation logic and at least one for a state mutability. It will be a good idea to assert the base classes as well. They have quite important equality methods. For example, here is an entity test:



```
public class EntitySpecs
{
    [Fact]
    ✓ | 0 references
    public void EntitiesWithEqualIdsShouldBeEqual()
    {
        // Arrange
        var first = new Manufacturer("First").SetId(1);
        var second = new Manufacturer("Second").SetId(1);

        // Act
        var result = first == second;

        // Arrange
        result.Should().BeTrue();
    }
}
```

SetId is an extension method added for testing purposes:



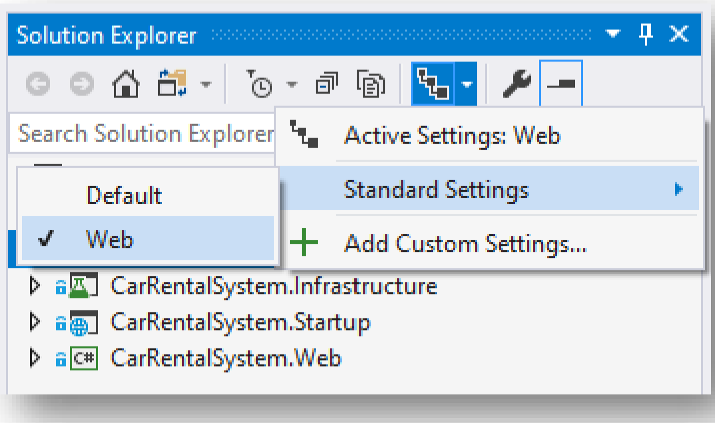
```
internal static class EntityExtensions
{
    4 references | ✓ 1/2 passing
    public static Entity<T> SetId<T>(this Entity<T> entity, int id)
        where T : struct
    {
        entity
            .GetType()
            .BaseType!
            .GetProperty(nameof(Entity<T>.Id))!
            .GetSetMethod(true)!
            .Invoke(entity, new object[] { id });

        return entity;
    }
}
```

In a production application, we should aim for higher code coverage. These tests may seem easy and unimportant, but the domain layer is the core of our business. It should be robust and bug-free.

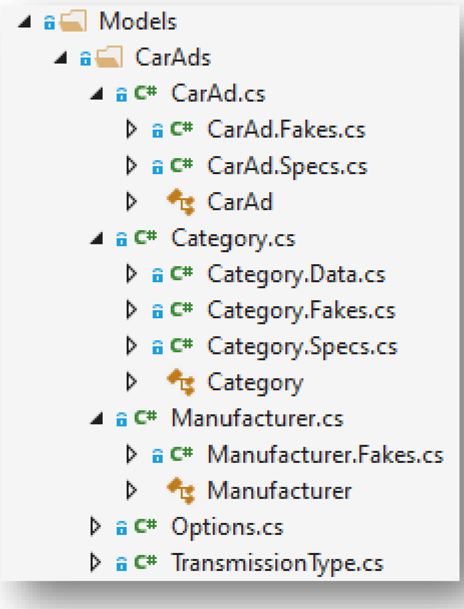
If you do not like the test file being situated next to the class itself, you may hide it by adding the following to your project file:

```xml
<ItemGroup>
  <ProjectCapability Include="DynamicFileNesting"/>
</ItemGroup>
```

Afterward, in the solution explorer, you need to enable web file nesting:



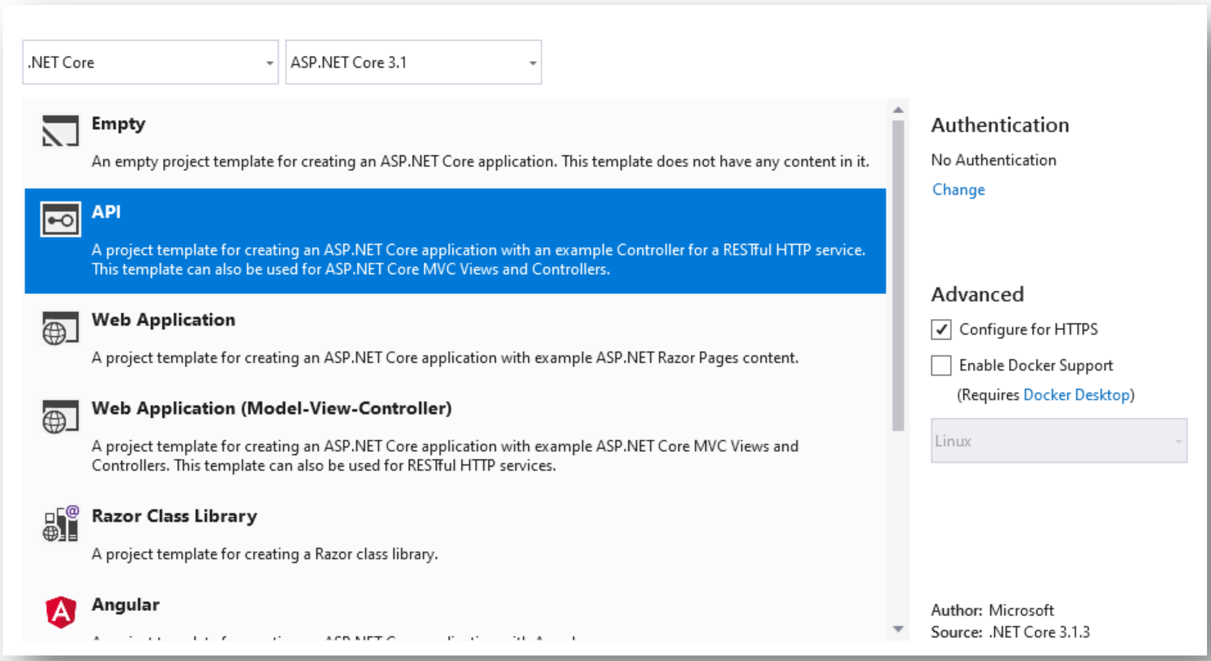
With this option turned on, the test files will be hidden like so:



You can find an implementation example of this section in the **Partial Solutions > 2. Domain Model Unit Tests** folder.

3. Introducing a Presentation Layer

We will now add the presentation layer because we need to start validating whether our application and infrastructure code is wired correctly. Add an **ASP.NET Core 3.1 API** project to the solution and name it **CarRentalSystem.Startup**:



.NET Core | ASP.NET Core 3.1

Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Web Application
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

Web Application (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

Razor Class Library
A project template for creating a Razor class library.

Angular

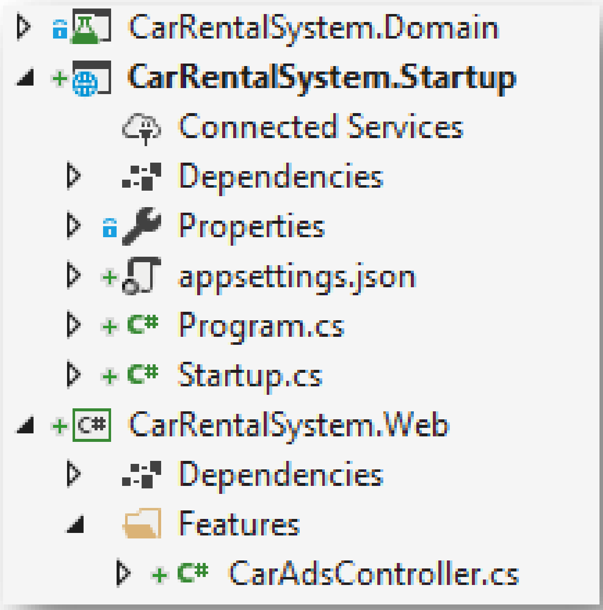
Authentication
No Authentication
Change

Advanced
[✓] Configure for HTTPS
[ ] Enable Docker Support
(Requires Docker Desktop)
Linux

Author: Microsoft
Source: .NET Core 3.1.3

Set the newly create project to be start up project for the solution. We named it **Startup** because this project will just contain our bootstrapping logic. Delete everything except **Program.cs**, **Startup.cs**, **appsettings.json**, and **launchSettings.json** (in the **Properties** folder). Clean these files from unnecessary comments, namespaces, and other not needed code.

Then, create another project – a new **.NET Core** class library. Name it **CarRentalSystem.Web**. This project will contain our *HTTP* request-response logic. Reference it by **Startup**, add **Features** folder in it, and add **CarAdsController**:



Here is the **CarAdsController** code:

```
[ApiController]
[Route("[controller]")]
0 references
public class CarAdsController : ControllerBase
{
    private static readonly Dealer Dealer = new Dealer("Dealer", "+12345678");

    [HttpGet]
    0 references
    public IEnumerable<CarAd> Get() => Dealer
        .CarAds
        .Where(c => c.IsAvailable);
}
```

You will need to reference the **ASP.NET Core** framework and the **Domain** project to compile the controller:

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.1</TargetFramework>
</PropertyGroup>

<ItemGroup>
  <FrameworkReference Include="Microsoft.AspNetCore.App" />
</ItemGroup>

<ItemGroup>
  <ProjectReference Include="..\CarRentalSystem.Domain\CarRentalSystem.Domain.csproj" />
</ItemGroup>
```

You can find the above configuration in the **Code Helpers > Presentation** folder.

As a final step for this section - disable nullable reference types for all new projects:

```
<LangVersion>8.0</LangVersion>
<Nullable>enable</Nullable>
<TreatWarningsAsErrors>true</TreatWarningsAsErrors>
```

In the next sections, we are going to use the **Get** action to validate whether our architecture and infrastructure code is working correctly. Make sure you set the startup project of your solution to be **CarRentalSystem.Startup**.

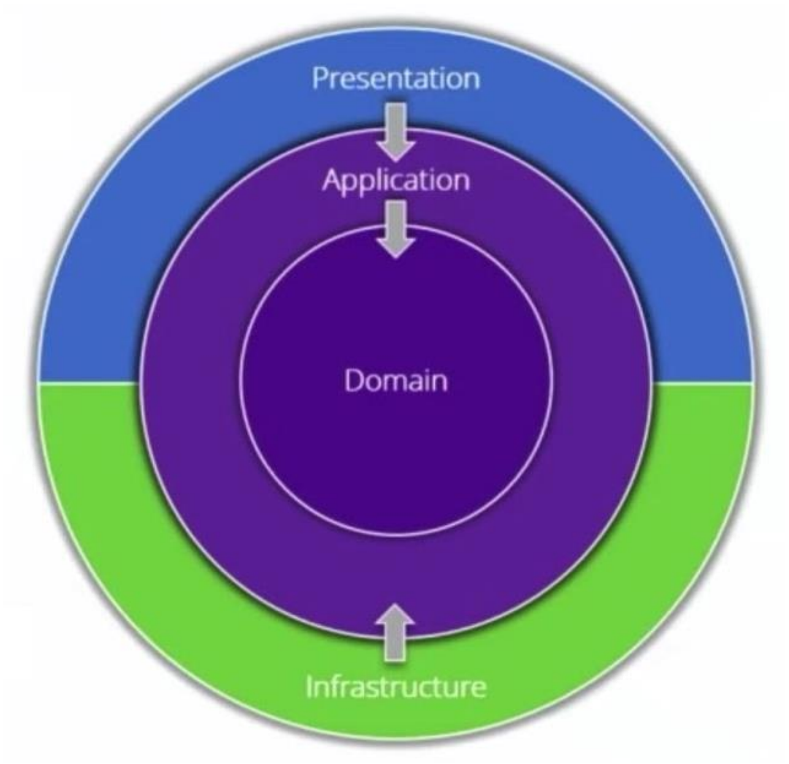
Once again, you can find the provided source in the **Partial Solutions > 3. Introducing a Presentation Layer** folder.

4. The Infrastructure Layer and Persistence

In this section, we are going to add our data layer by using **SQL Server** as a data storage and **Entity Framework Core 3.1** as an ORM. Using an object-relational with the domain model may force us to make some modifications to the underlying classes, and that is entirely OK if we follow the main rules of domain-driven design:

- Keep the domain models immutable & read-only. If mutability is needed – it is better to create a separate class just for the data layer.
- Do not add ORM-specific logic to the domain objects – data annotations, for example. These attributes should not be used in DDD. Use the Entity Framework fluent configuration instead.

Following clean architecture - the persistence logic and all other third-party dependencies should be part of the infrastructure layer:



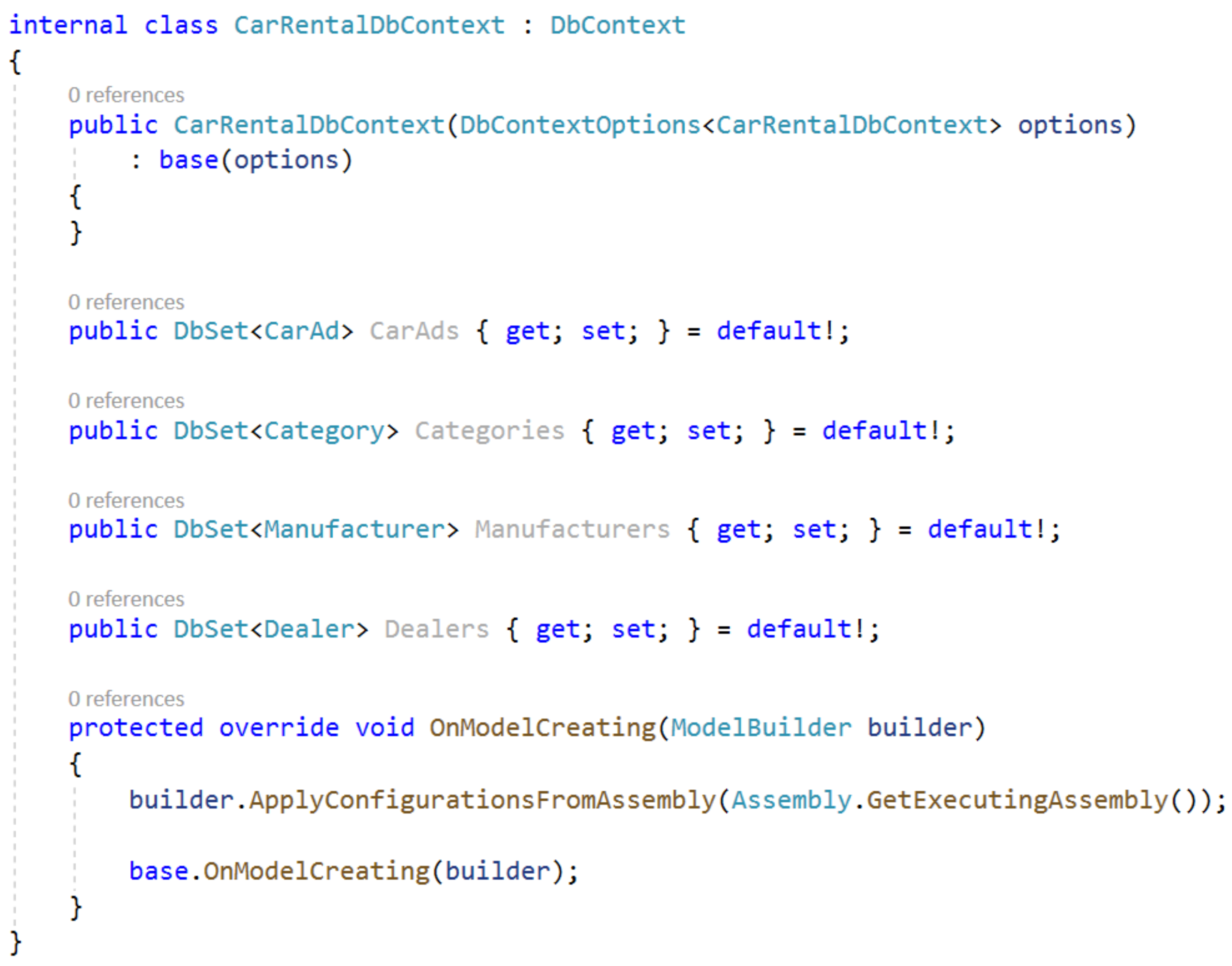
Add a new **.NET Core** class library to the solution and name it **CarRentalSystem.Infrastructure**. Open the project file and disable nullable reference types. Make sure its target framework is **netcoreapp3.1**, reference the **Domain** project, and install these packages from **NuGet**:

- **Microsoft.EntityFrameworkCore.SqlServer**

- **Microsoft.EntityFrameworkCore.Tools**

Afterwards, add a folder **Persistence** at the root of the project.

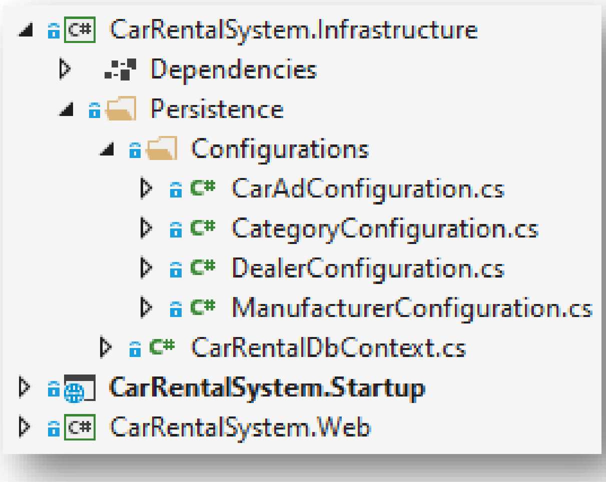
Create **CarRentalDbContext**, define database sets for every domain entity, and set the model builder to search for configurations in the current assembly:



```
internal class CarRentalDbContext : DbContext
{
    0 references
    public CarRentalDbContext(DbContextOptions<CarRentalDbContext> options)
        : base(options)
    {
    }

    0 references
    public DbSet<CarAd> CarAds { get; set; } = default!;

    0 references
    public DbSet<Category> Categories { get; set; } = default!;

    0 references
    public DbSet<Manufacturer> Manufacturers { get; set; } = default!;

    0 references
    public DbSet<Dealer> Dealers { get; set; } = default!;

    0 references
    protected override void OnModelCreating(ModelBuilder builder)
    {
        builder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());

        base.OnModelCreating(builder);
    }
}
```

Make sure the **CarRentalDbContext** class is marked as *internal*. It is a persistence detail, and it should not be visible to any other layer. Additionally, we need to tell the compiler it is safe to ignore the uninitialized **DbSet** properties with the null-forgiving operator (!).

Now, add a **Configurations** folder in the **Persistence** one and start creating database configurations for each domain model:



Here is the **CategoryConfiguration** implementation:

```
using static Domain.Models.ModelConstants.Common;
using static Domain.Models.ModelConstants.Category;

0 references
internal class CategoryConfiguration : IEntityTypeConfiguration<Category>
{
    3 references
    public void Configure(EntityTypeBuilder<Category> builder)
    {
        builder
            .HasKey(c => c.Id);

        builder
            .Property(c => c.Name)
            .IsRequired()
            .HasMaxLength(MaxNameLength);

        builder
            .Property(c => c.Description)
            .IsRequired()
            .HasMaxLength(MaxDescriptionLength);
    }
}
```

<u>Now it is your turn</u>. Implement the other three configurations. Some hints are provided for your convenience.

For the *CarAd-Manufacturer* and *CarAd-Category* relationships, you may use the following code in **CarAdConfiguration** class:

```
builder
    .HasOne(c => c.Manufacturer)
    .WithMany()
    .HasForeignKey("ManufacturerId")
    .OnDelete(DeleteBehavior.Restrict);

builder
    .HasOne(c => c.Category)
    .WithMany()
    .HasForeignKey("CategoryId")
    .OnDelete(DeleteBehavior.Restrict);
```

Since we do not have two-way navigational properties and foreign key columns, we need to configure the relationships with strings.

And for the *CarAd-Options* ownership, we need to use the **OwnsOne** method, because **Options** is a value object and not an entity:

```
builder
    .OwnsOne(c => c.Options, o =>
    {
        o.WithOwner();

        o.Property(op => op.NumberOfSeats);
        o.Property(op => op.HasClimateControl);

        o.OwnsOne(
            op => op.TransmissionType,
            t =>
            {
                t.WithOwner();

                t.Property(tr => tr.Value);
            });
    });
```

As for the **DealerConfiguration**, here are the more difficult parts:

```
builder
    .OwnsOne(
        d => d.PhoneNumber,
        p =>
        {
            p.WithOwner();

            p.Property(pn => pn.Number);
        });

builder
    .HasMany(pr => pr.CarAds)
    .WithOne()
    .Metadata
    .PrincipalToDependent
    .SetField("carAds");
```

Because the **CarAds** collection is read-only, we need to explicitly tell **Entity Framework Core** to use the private field by providing its name.

Now, we will wire the database classes to the rest of the solution.

First, create an **InfrastructureConfiguration** class at the root of the project and add the following code:

```
public static class InfrastructureConfiguration
{
    1 reference
    public static IServiceCollection AddInfrastructure(
        this IServiceCollection services,
        IConfiguration configuration)
        => services
            .AddDbContext<CarRentalDbContext>(options => options
                .UseSqlServer(
                    configuration.GetConnectionString("DefaultConnection"),
                    b => b.MigrationsAssembly(
                        typeof(CarRentalDbContext).Assembly.FullName)));
}
```

Afterwards, we need to install the **Microsoft.EntityFrameworkCore.Design NuGet** package in the **CarRentalSystem.Startup** project. Then we need to reference the **Infrastructure** project in it.

Then go to the **Startup** class and update the **ConfigureServices** method to add the infrastructure services:

```
public void ConfigureServices(IServiceCollection services)
    => services
        .AddInfrastructure(this.Configuration)
        .AddControllers();
```

Finally, open the **appsettings.json** file, and add your connection string:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "YourConnectionString"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

You can use **LocalDB** if you do not have **SQL Server** installed:

```
Server=(localdb)\\mssqllocaldb;Database=CarRentalSystem;Trusted_Conn
ection=True;MultipleActiveResultSets=true
```

The **LocalDB** data is accessible through the **View > SQL Server Object Explorer** option in **Visual Studio**.

Make sure your startup project is correct. Then open the **Package Manager Console**, choose the **Infrastructure** project, and add our first migration by calling:

```
Add-Migration InitialDomainTables -OutputDir "Persistence/Migrations"
```

Ooops, an exception. Turns out our domain models are not suitable for **Entity Framework Core** in their current form:

```
No suitable constructor found for entity type 'CarAd'. The following
constructors had parameters that could not be bound to properties of
the entity type: cannot bind 'manufacturer', 'category', 'options' in
 'CarAd(Manufacturer manufacturer, string model, Category category,
string imageUrl, decimal pricePerDay, Options options, bool
isAvailable)'.
```

Unfortunately, we will need to add a little bit of "dirt" in our domain classes. **Entity Framework Core** wants constructors that bind non-navigational properties, but according to the **Domain-Driven Design** principles, entities cannot be created with an invalid state. The solution is to add additional *private* constructors to our domain model classes for **Entity Framework Core** to use.

Here is the **CarAd** private constructor, for example:

```
private CarAd(
    string model,
    string imageUrl,
    decimal pricePerDay,
    bool isAvailable)
{
    this.Model = model;
    this.ImageUrl = imageUrl;
    this.PricePerDay = pricePerDay;
    this.IsAvailable = isAvailable;

    this.Manufacturer = default!;
    this.Category = default!;
    this.Options = default!;
}
```

Once again, we need to tell the compiler it is safe to ignore the uninitialized properties.

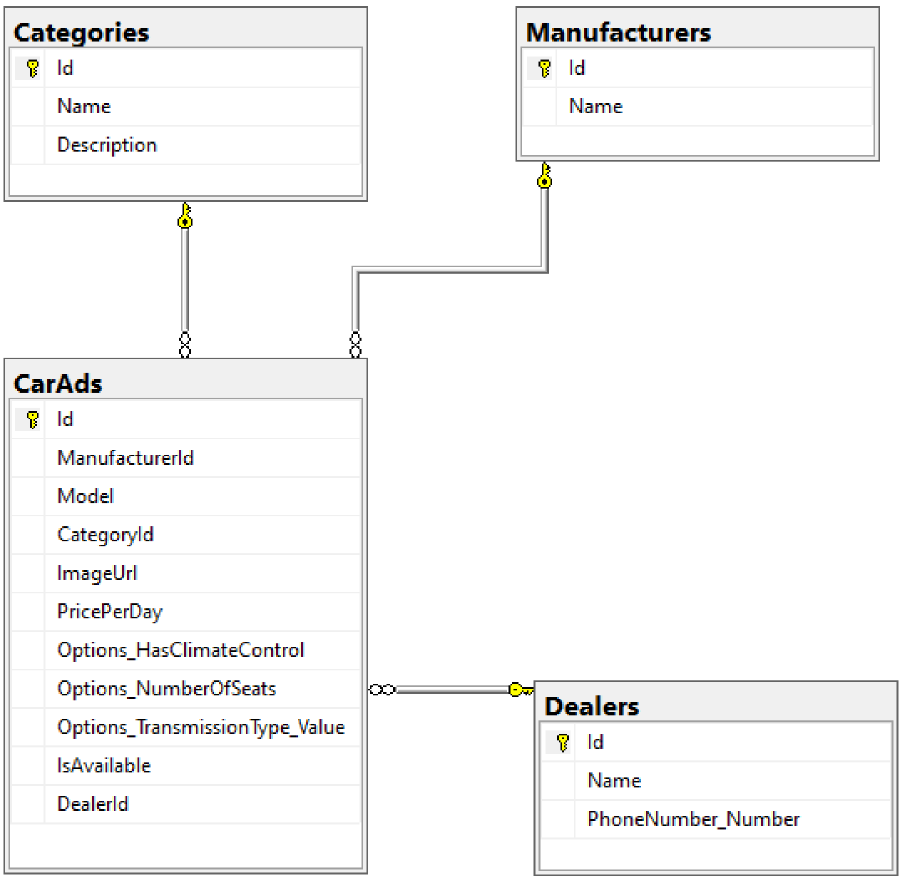
Add all other needed constructors and generate the first migration successfully this time.

Everything fine? Good! Now create the database:

```
Update-Database
```

Check the database and the created schema.

Here is a potential database diagram of the tables:



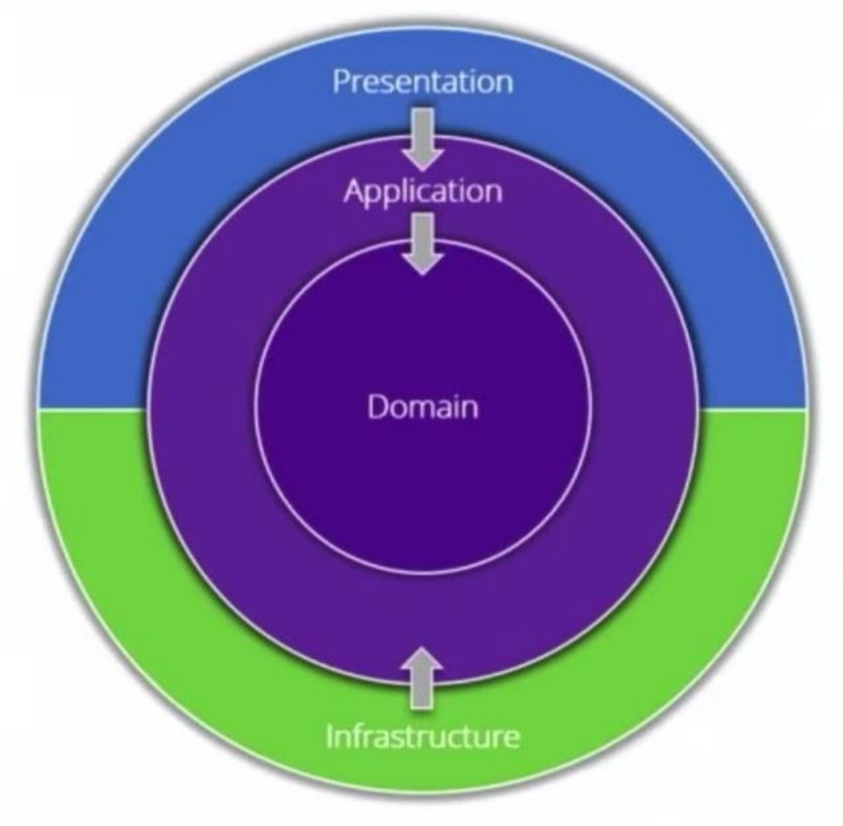
The persistence layer is now done! You may check the provided solution in the **Partial Solutions > 4. The Infrastructure Layer and Persistence** folder.

5. The Application Layer and Repositories

In this section, we are going to finish our project structure by adding the application layer. It is responsible for the business logic of our solution. We will use it to expose the database to the presentation layer through interfaces.

Add a new **.NET Core** class library to the solution and name it **CarRentalSystem.Application**. Open the project file and disable nullable reference types. Make sure its target framework is **netcoreapp3.1** and reference the **Domain** project.

Since we now have all parts of the project structure, we need to make sure the dependencies follow the clean architecture diagram:



Update all project references like so:

- **Domain** – references no other project.
- **Application** – references **Domain**.
- **Infrastructure** – references **Application**.
- **Web** – references **Application**.
- **Startup** – references **Infrastructure** and **Web**.

Create a **Contracts** folder in the **Application** project. Add the following **IRepository** interface:

```
public interface IRepository<out TEntity>
    where TEntity : IAggregateRoot
{
    2 references
    IQueryable<TEntity> All();

    1 reference
    Task<int> SaveChanges(CancellationToken cancellationToken = default);
}
```

The purpose of this abstraction is to restrict the access of the non-domain layers to aggregate roots only. It serves as an anti-corruption layer to our domain.

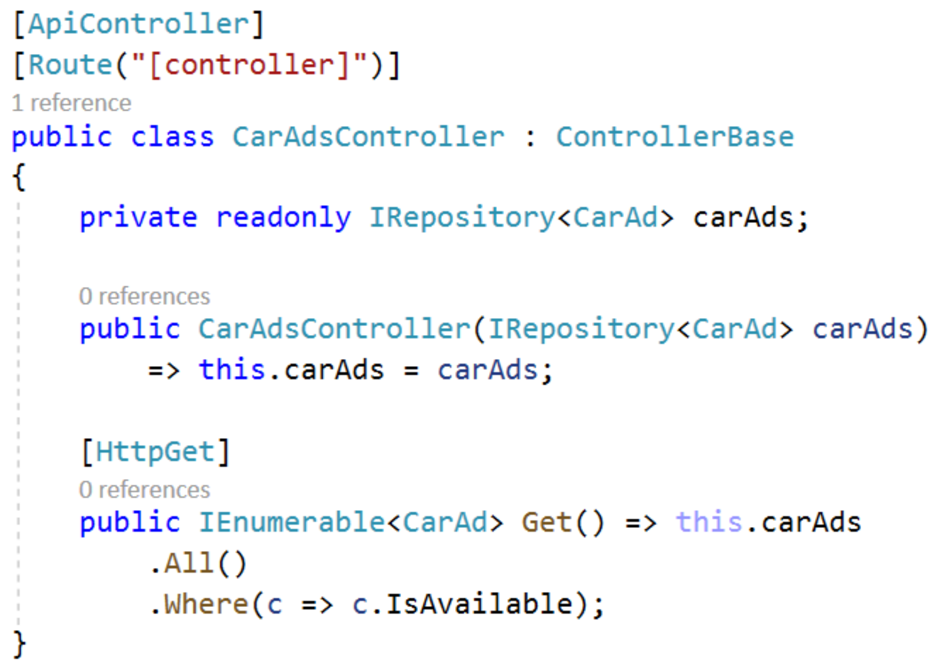
Now, go to the **Infrastructure** project and add a **Repositories** folder in the **Persistence** one. Create the following **DataRepository** class:

```
internal class DataRepository<TEntity> : IRepository<TEntity>
    where TEntity : class, IAggregateRoot
{
    private readonly CarRentalDbContext db;

    0 references
    public DataRepository(CarRentalDbContext db) => this.db = db;

    1 reference
    public IQueryable<TEntity> All() => this.db.Set<TEntity>();

    1 reference
    public Task<int> SaveChanges(CancellationToken cancellationToken = default)
        => this.db.SaveChangesAsync(cancellationToken);
}
```

Then go to the **InfrastructureConfiguration** class and register the repository in the service provider:

```
public static IServiceCollection AddInfrastructure(
    this IServiceCollection services,
    IConfiguration configuration)
    => services
        .AddDbContext<CarRentalDbContext>(options => options
            .UseSqlServer(
                configuration.GetConnectionString("DefaultConnection"),
                b => b.MigrationsAssembly(typeof(CarRentalDbContext)
                    .Assembly.FullName)))
        .AddTransient(typeof(IRepository<>), typeof(DataRepository<>));
```

Finally, go to the **Web** project, and update the **CarAdsController** to use the repository:



```
[ApiController]
[Route("[controller]")]
1 reference
public class CarAdsController : ControllerBase
{
    private readonly IRepository<CarAd> carAds;

    0 references
    public CarAdsController(IRepository<CarAd> carAds)
        => this.carAds = carAds;

    [HttpGet]
    0 references
    public IEnumerable<CarAd> Get() => this.carAds
        .All()
        .Where(c => c.IsAvailable);
}
```

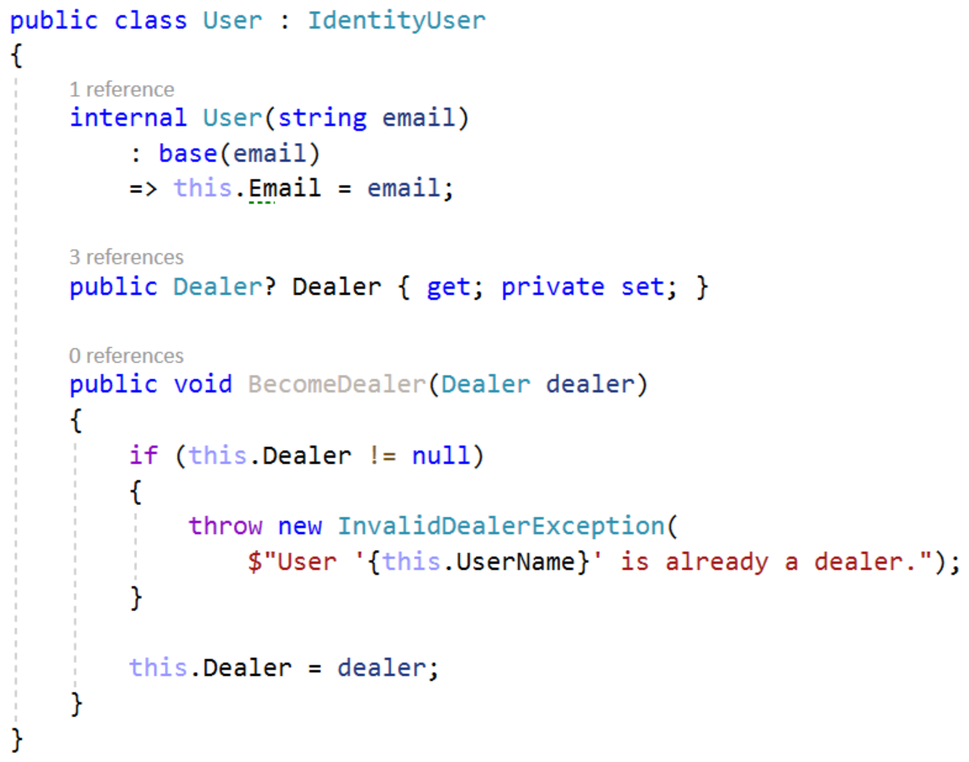
Run the project, hit the *"/CarAd"* route, and validate that all our pieces are working correctly together. Great job!

This part's solution is available in the **Partial Solutions > 5. The Application Layer and Repositories** folder.

6. Authentication with Identity

We are now going to add authentication to our system. We will start with the *Identity* database tables.

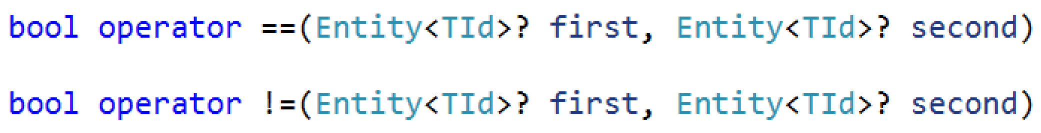
Install **Microsoft.AspNetCore.Identity.EntityFrameworkCore** into the **Infrastructure** project and create a folder named **Identity**. Add a **User** class in it:



```
public class User : IdentityUser
{
    1 reference
    internal User(string email)
        : base(email)
        => this.Email = email;

    3 references
    public Dealer? Dealer { get; private set; }

    0 references
    public void BecomeDealer(Dealer dealer)
    {
        if (this.Dealer != null)
        {
            throw new InvalidDealerException(
                $"User '{this.UserName}' is already a dealer.");
        }

        this.Dealer = dealer;
    }
}
```

As you can see from the defined constructor - we try to follow the best DDD practices in the **User** class. Still, we are forced to do some null-related shenanigans. The reason is that the built-in **UserManager** type is responsible for creating new users, and it will be a violation of the single responsibility principle if we override its behavior to create dealer objects too.

The above **User** code will not compile, unless you update the operators in the **Entity** base class to support nullable objects:



```
bool operator ==(Entity<TId>? first, Entity<TId>? second)

bool operator !=(Entity<TId>? first, Entity<TId>? second)
```

To wire the **User** class to the database, we need to change the **DbContext** and add a user configuration class.

First, make the **CarRentalDbContext** inherit from **IdentityDbContext<User>**:

```
internal class CarRentalDbContext : IdentityDbContext<User>
```

Second, add a **UserConfiguration** class in the **Persistence** > **Configurations** folder:

```
public class UserConfiguration : IEntityTypeConfiguration<User>
{
    4 references
    public void Configure(EntityTypeBuilder<User> builder)
    {
        builder
            .HasOne(u => u.Dealer)
            .WithOne()
            .HasForeignKey<User>("DealerId")
            .OnDelete(DeleteBehavior.Restrict);
    }
}
```

Open the Package Manager Console and create a new migration:

```
Add-Migration UserTable -OutputDir "Persistence/Migrations"
```

Before we begin, we need to migrate the database, but let us implement automatic migrations in our code for better convenience.

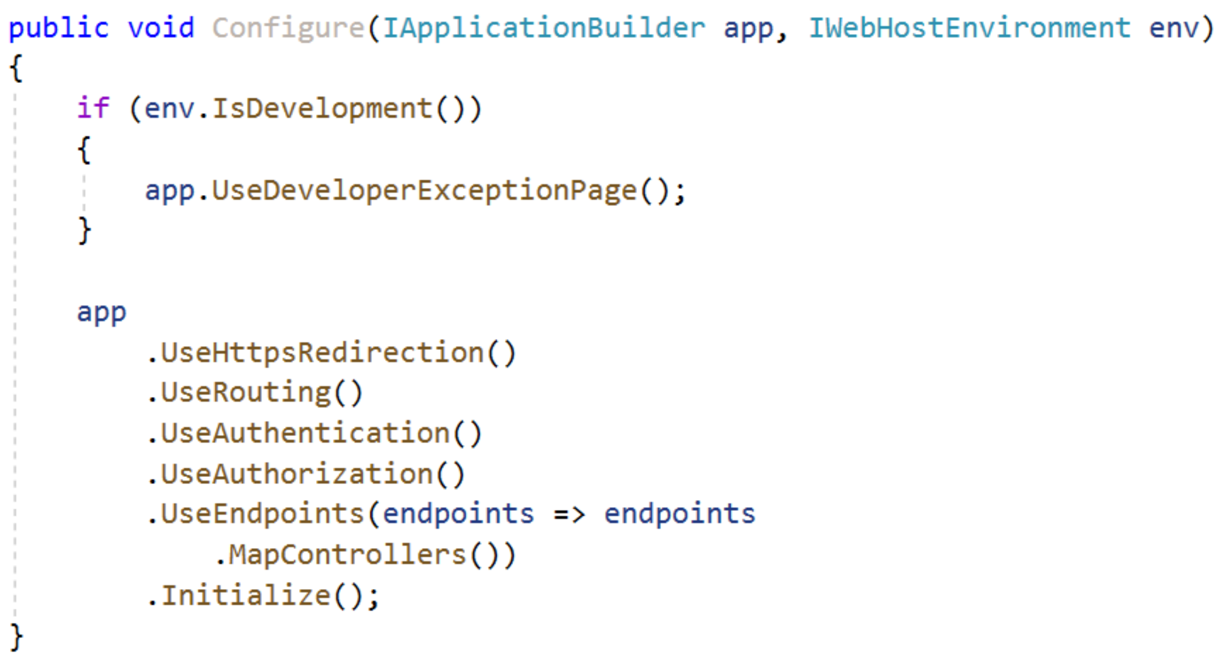
Go to the **Code Helpers > Identity** folder and:

- Copy the **IInitializer** file to the root of the **Infrastructure** project.
- Copy the **CarRentalDbInitializer** file to the **Infrastructure** > **Persistence** folder.
- Copy the **ApplicationInitialization** file to the root of the **Startup** project.

Open **InfrastructureConfiguration** and register the database initializer:

```
.AddDbContext<CarRentalDbContext>(options => options
    .UseSqlServer(
        configuration.GetConnectionString("DefaultConnection"),
        b => b.MigrationsAssembly(typeof(CarRentalDbContext)
            .Assembly.FullName)))
.AddTransient<IInitializer, CarRentalDbInitializer>()
.AddTransient(typeof(IRepository<>), typeof(DataRepository<>));
```

Go to the **Startup** file and update the **Configure** method:



```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app
        .UseHttpsRedirection()
        .UseRouting()
        .UseAuthentication()
        .UseAuthorization()
        .UseEndpoints(endpoints => endpoints
            .MapControllers())
        .Initialize();
}
```

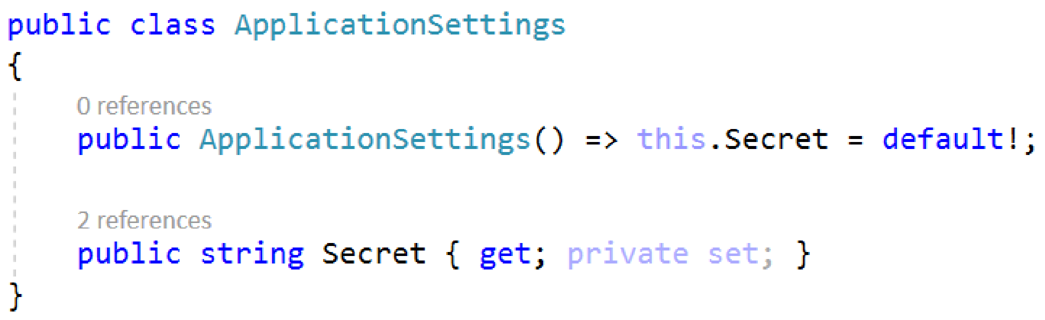
We added the **UseAuthentication** and **Initialize** calls.

We are done with the database. Run the application and validate that the database is now migrated.

Let us continue by defining the interfaces and classes our business logic needs. Go to the **Application** project and copy the **Result** type from the **Code Helpers > Identity** folder.

We are going to use this class in our application layer in case we need to return either a successful result or an error message. The successful result may or may not have additional data.

Next to the **Result** class, add **ApplicationSettings**:



```
public class ApplicationSettings
{
    0 references
    public ApplicationSettings() => this.Secret = default!;

    2 references
    public string Secret { get; private set; }
}
```

Then install **Microsoft.Extensions.Options.ConfigurationExtensions** into the **Application** project and add **ApplicationConfiguration** to it:

```
public static IServiceCollection AddApplication(
    this IServiceCollection services,
    IConfiguration configuration)
    => services
        .Configure<ApplicationSettings>(
            configuration.GetSection(nameof(ApplicationSettings)),
            options => options.BindNonPublicProperties = true);
```

Now go to the **Startup** class and update the **ConfigureServices** method to add the application services:

```
public void ConfigureServices(IServiceCollection services)
    => services
        .AddApplication(this.Configuration)
        .AddInfrastructure(this.Configuration)
        .AddControllers();
```

Then open the **appsettings.json** file, and add the new section:

```
"ApplicationSettings": {
  "Secret": "YourRand0mSecret"
},
"ConnectionStrings": {
  "DefaultConnection": "YourConnectionString"
},
"Logging": {
  "LogLevel": {
    "Default": "Information",
    "Microsoft": "Warning",
    "Microsoft.Hosting.Lifetime": "Information"
  }
},
"AllowedHosts": "*"
```

You can validate that the application settings are configured correctly by using the **CarAdsController**:

```
[ApiController]
[Route("[controller]")]
1 reference
public class CarAdsController : ControllerBase
{
    private readonly IRepository<CarAd> carAds;
    private readonly IOptions<ApplicationSettings> settings;

    0 references
    public CarAdsController(
        IRepository<CarAd> carAds,
        IOptions<ApplicationSettings> settings)
    {
        this.carAds = carAds;
        this.settings = settings;
    }

    [HttpGet]
    0 references
    public object Get() => new
    {
        Settings = this.settings,
        CarAds = this.carAds
            .All()
            .Where(c => c.IsAvailable)
            .ToList()
    };
}
```

If everything is working correctly, then go to the **Application** project and add a folder named **Features**. Create another folder inside of it – **Identity**. Add these two classes there:

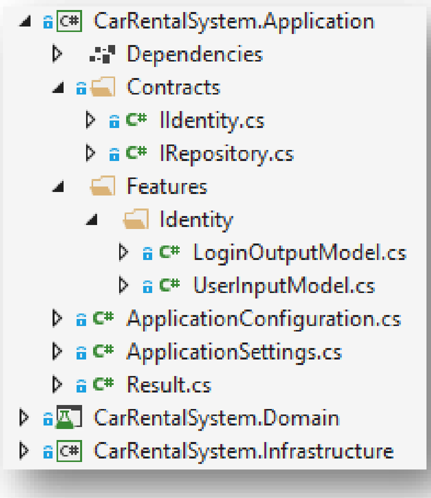
```
public class UserInputModel
{
    0 references
    public UserInputModel(string email, string password)
    {
        this.Email = email;
        this.Password = password;
    }

    3 references
    public string Email { get; }

    3 references
    public string Password { get; }
}
```

```
public class LoginOutputModel
{
    1 reference
    public LoginOutputModel(string token)
        => this.Token = token;

    1 reference
    public string Token { get; }
}
```

Afterwards, create an **IIdentity** interface in the **Contracts** folder:

```
public interface IIdentity
{
    1 reference
    Task<Result> Register(UserInputModel userInput);

    1 reference
    Task<Result<LoginOutputModel>> Login(UserInputModel userInput);
}
```

Your application project should look like this:



```
◢ CarRentalSystem.Application
    ▷ Dependencies
    ◢ Contracts
        ▷ IIdentity.cs
        ▷ IRepository.cs
    ◢ Features
        ◢ Identity
            ▷ LoginOutputModel.cs
            ▷ UserInputModel.cs
    ▷ ApplicationConfiguration.cs
    ▷ ApplicationSettings.cs
    ▷ Result.cs
▷ CarRentalSystem.Domain
▷ CarRentalSystem.Infrastructure
```

The **IIdentity** interface will be implemented by the **Infrastructure** layer. Go to it and install **Microsoft.AspNetCore.Authentication.JwtBearer** from **NuGet**.

Go to the **Code Helpers > Identity** folder and:

- Copy the **IdentityService** file to the **Infrastructure** > **Identity** folder.
- Copy the **InfrastructureConfiguration** file to the root of the **Infrastructure** project. Replace the existing file.
- Copy the **IdentityController** file to the **Web** > **Features** folder.

Now, go to the **Web** project and install a third-party *JSON* parser - **Microsoft.AspNetCore.Mvc.NewtonsoftJson**. We need it because our input models do not contain parameterless constructors.

Create a **WebConfiguration** class at the root of the **Web** project:

```
public static class WebConfiguration
{
    1 reference
    public static IServiceCollection AddWebComponents(
        this IServiceCollection services)
    {
        services
            .AddControllers()
            .AddNewtonsoftJson();

        return services;
    }
}
```

Use the **AddWebComponents** method in the **Startup** file:

```
public void ConfigureServices(IServiceCollection services)
    => services
        .AddApplication(this.Configuration)
        .AddInfrastructure(this.Configuration)
        .AddWebComponents();
```

We have implemented the authentication in our system.

This part's solution is available in the **Partial Solutions > 6. Authentication with Identity** folder.

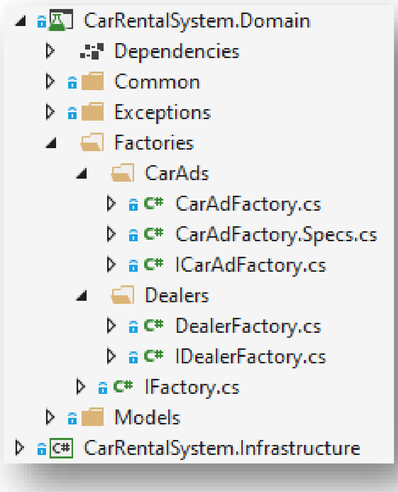
7. Creating Entities with Builder Factories

Go to the **Domain** project and make sure every entity has an internal constructor, even the aggregate roots. We are now going to add a factory layer which will be responsible for instantiating valid objects. Constructors are excellent for this purpose, but since our entities have a lot of properties, the *Builder* pattern will be more suitable and convenient from a developer's point of view.

Create a folder **Factories** in the **Domain** project, and add an **IFactory** interface in it:

```
public interface IFactory<out TEntity>
    where TEntity : IAggregateRoot
{
    0 references
    TEntity Build();
}
```

As you can see, factories allow only aggregate roots. Now, add the following structure:

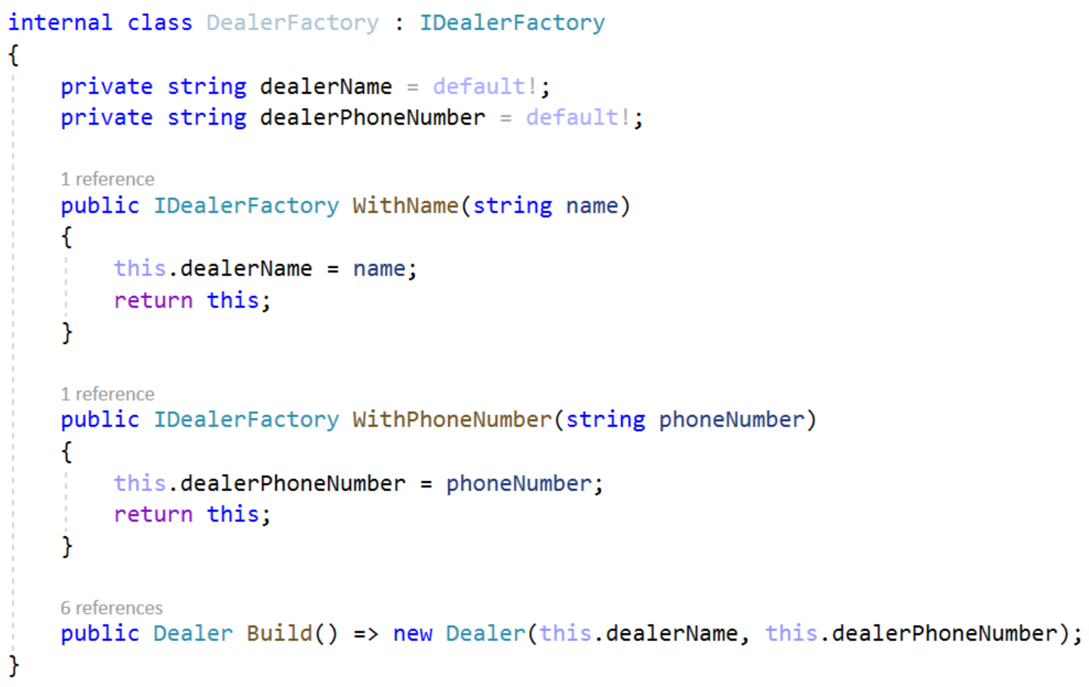


```
CarRentalSystem.Domain
  Dependencies
  Common
  Exceptions
  Factories
    CarAds
      CarAdFactory.cs
      CarAdFactory.Specs.cs
      ICarAdFactory.cs
    Dealers
      DealerFactory.cs
      IDealerFactory.cs
    IFactory.cs
  Models
CarRentalSystem.Infrastructure
```

This is the **IDealerFactory** interface:

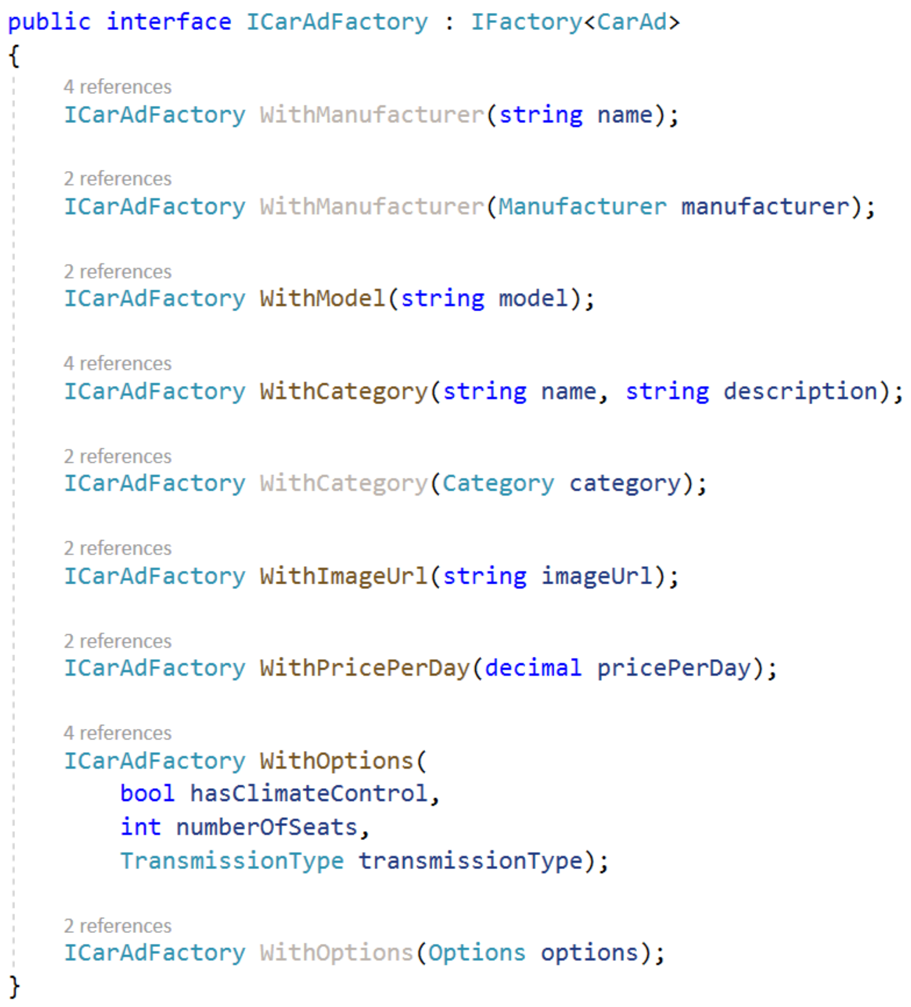
```
public interface IDealerFactory : IFactory<Dealer>
{
    1 reference
    IDealerFactory WithName(string name);

    1 reference
    IDealerFactory WithPhoneNumber(string phoneNumber);
}
```

And this is its implementation:



```
internal class DealerFactory : IDealerFactory
{
    private string dealerName = default!;
    private string dealerPhoneNumber = default!;

    1 reference
    public IDealerFactory WithName(string name)
    {
        this.dealerName = name;
        return this;
    }

    1 reference
    public IDealerFactory WithPhoneNumber(string phoneNumber)
    {
        this.dealerPhoneNumber = phoneNumber;
        return this;
    }

    6 references
    public Dealer Build() => new Dealer(this.dealerName, this.dealerPhoneNumber);
}
```

Note the internal access modifier. The **ICarAdFactory** interface:



```
public interface ICarAdFactory : IFactory<CarAd>
{
    4 references
    ICarAdFactory WithManufacturer(string name);

    2 references
    ICarAdFactory WithManufacturer(Manufacturer manufacturer);

    2 references
    ICarAdFactory WithModel(string model);

    4 references
    ICarAdFactory WithCategory(string name, string description);

    2 references
    ICarAdFactory WithCategory(Category category);

    2 references
    ICarAdFactory WithImageUrl(string imageUrl);

    2 references
    ICarAdFactory WithPricePerDay(decimal pricePerDay);

    4 references
    ICarAdFactory WithOptions(
        bool hasClimateControl,
        int numberOfSeats,
        TransmissionType transmissionType);

    2 references
    ICarAdFactory WithOptions(Options options);
}
```

We add all possible ways to create a complex object. During further development, we will decide which of these methods will be deemed unnecessary.

It is your turn. Implement the **CarAdFactory** and write unit tests for it in the **CarAdFactorySpecs** class. Validate that the factory cannot create a car without its related entities.

Builder factories save us time from writing too many (and too long) constructors, but they do not have compile-time type safety. For this reason, it is a good idea to add methods like this one (only if the entity does not have a lot of properties):

```
public Dealer Build() => new Dealer(this.dealerName, this.dealerPhoneNumber);

0 references
public Dealer Build(string name, string phoneNumber)
    => this
        .WithName(name)
        .WithPhoneNumber(phoneNumber)
        .Build();
```

After you are done, install the **Microsoft.Extensions.DependencyInjection** and **Scrutor** packages to the **Domain** project.

Now go to the **Code Helpers > Factories** folder and copy the two files there at the root of the **Domain** project. Here is the **DomainConfiguration** one:

```
public static IServiceCollection AddDomain(this IServiceCollection services)
    => services
        .Scan(scan => scan
            .FromCallingAssembly()
            .AddClasses(classes => classes
                .AssignableTo(typeof(IFactory<>)))
            .AsMatchingInterface()
            .WithTransientLifetime());
```

We are using *Scrutor* to register factories automatically. Just make sure to call the **AddDomain** method in the **Startup** file:

```
public void ConfigureServices(IServiceCollection services)
    => services
        .AddDomain()
        .AddApplication(this.Configuration)
        .AddInfrastructure(this.Configuration)
        .AddWebComponents();
```

Factories – done! The solution of this section is in the **Partial Solutions > 7. Creating Entities with Builder Factories** folder.

8. Simplifying Business Logic with CQRS and MediatR

We are now going to separate the business logic with *CQRS*. Commands are business rules which change the state of the application, and queries just fetch information without mutating any data.

First, install **MediatR.Extensions.Microsoft.DependencyInjection** to the **Application** project. Then configure **MediatR** in the **ApplicationConfiguration**:

```
public static IServiceCollection AddApplication(
    this IServiceCollection services,
    IConfiguration configuration)
    => services
        .Configure<ApplicationSettings>(
            configuration.GetSection(nameof(ApplicationSettings)),
            options => options.BindNonPublicProperties = true)
        .AddMediatR(Assembly.GetExecutingAssembly());
```

Afterwards, install **MediatR** to **Web** project and create **ApiController** at its root:

```
using MediatR;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.DependencyInjection;

[ApiController]
[Route("[controller]")]
2 references
public abstract class ApiController : ControllerBase
{
    private IMediator? mediator;

    0 references
    protected IMediator Mediator
        => this.mediator ??= this.HttpContext
            .RequestServices
            .GetService<IMediator>();
}
```

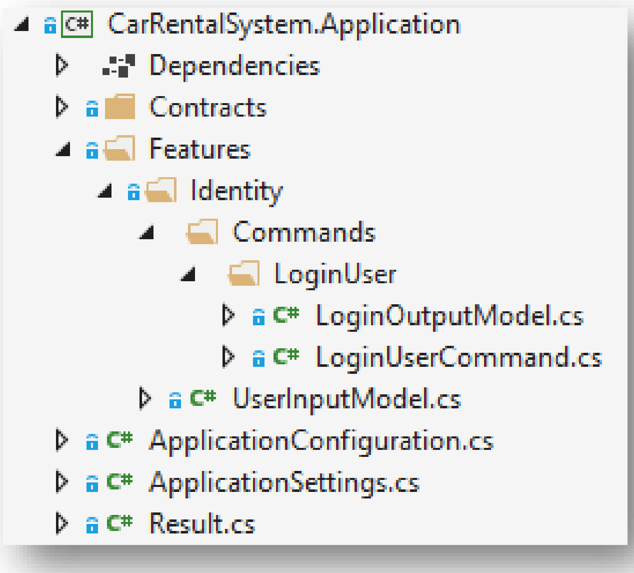
Make sure you reference the **Microsoft.Extensions.DependencyInjection** namespace. Otherwise, the code will not compile.

All our controllers should now inherit from the **ApiController**:

```
public class IdentityController : ApiController
```

Go to the **IdentityController** and delete the **Get** method. It was there only for testing purposes.

Go to the **Application** project and create a **LoginUserCommand** class and the folder structure shown below. Move the **LoginOutputModel** too. Make sure you fix its namespace.



```
CarRentalSystem.Application
    Dependencies
    Contracts
    Features
        Identity
            Commands
                LoginUser
                    LoginOutputModel.cs
                    LoginUserCommand.cs
            UserInputModel.cs
    ApplicationConfiguration.cs
    ApplicationSettings.cs
    Result.cs
```

Open the **LoginUserCommand** and inherit the **UserInputModel** and its constructor because the properties are the same.

Add an inner class **LoginUserCommandHandler** like so:

```
public class LoginUserCommand : UserInputModel, IRequest<Result<LoginOutputModel>>
{
    0 references
    public LoginUserCommand(string email, string password)
        : base(email, password)
    {
    }

    1 reference
    public class LoginUserCommandHandler : IRequestHandler<LoginUserCommand, Result<LoginOutputModel>>
    {
        private readonly IIdentity identity;

        0 references
        public LoginUserCommandHandler(IIdentity identity) => this.identity = identity;

        0 references
        public async Task<Result<LoginOutputModel>> Handle(
            LoginUserCommand request,
            CancellationToken cancellationToken)
            => await this.identity.Login(request);
    }
}
```

Let us first implement the whole request-response pipeline, and then we are going to explain the details of the above implementation.

Go to the **Web** project and create a folder **Common**. Copy the **ResultExtensions** file from the **Code Helpers > CQRS folder** in it. This class provides friendly extensions methods to convert objects into *HTTP* action results.

Finally, make the **Login** method in the **IdentityController** work with the new command:

```
[HttpPost]
[Route(nameof(Login))]
1 reference
public async Task<ActionResult<LoginOutputModel>> Login(LoginUserCommand command)
    => await this.Mediator.Send(command).ToActionResult();
```

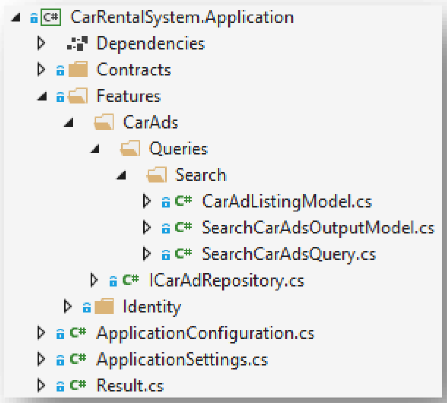
Here are the rules you need to follow with *CQRS* in the application layer:

- Add a command or query class (the input model) containing the request properties. We will cover validation in one of the next sections.
- Implement the **IRequest<TResponse>** interface.
- **TResponse** should be:
  - **Result** if the possible responses are either success with no data or error messages.
  - **Result<TOutputModel>** if the possible responses are either success with specific response data or error messages.
  - An **OutputModel** if you do not need to handle error messages in the business logic.
- Input and output models should only define properties which the particular use cases require.
- Input or output models should not inherit or reference any domain models.
- Input or output models should be encapsulated and serializable.
- Aim to have private setters in your models.
- Do not use the same input or output model for multiple scenarios. You may extract base classes for code reuse.
- Do not use the same model for both input and output scenarios.
- Create an inner handler class for each command and query and implement **IRequestHandler<TRequest, TResponse>**. The **TRequest** should be the input model class.
- Inject in the handler class all the services you may need – repositories, factories, etc. Return the expected **TResponse** result.

If you follow these rules, it will be easy for you to separate the *HTTP* logic from the business one. The controller should always delegate the request to inner services. Its single responsibility should be binding the request data and producing an action result.

Let us now create a query for searching the car ads in the system.

Add files and folders to match the following structure:



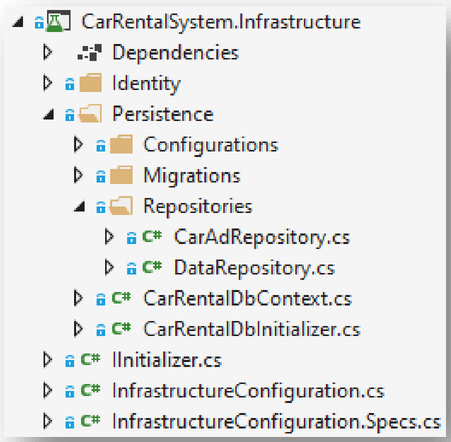
```
CarRentalSystem.Application
  Dependencies
  Contracts
  Features
    CarAds
      Queries
        Search
          CarAdListingModel.cs
          SearchCarAdsOutputModel.cs
          SearchCarAdsQuery.cs
      ICarAdRepository.cs
    Identity
  ApplicationConfiguration.cs
  ApplicationSettings.cs
  Result.cs
```

The **CarAdListingModel**, **SearchCarAdsOutputModel**, and **ICarAdRepository** source code is available in the **Code Helpers > CQRS** folder.

Copy the **CarAdRepository** too and add it to **Infrastructure > Persistence** > **Repositories**.

Additionally, copy the **InfrastructureConfiguration** and its specification file at the root of the **Infrastructure** project.

Your folder structure should look like the following:



```
CarRentalSystem.Infrastructure
  Dependencies
  Identity
  Persistence
    Configurations
    Migrations
    Repositories
      CarAdRepository.cs
      DataRepository.cs
    CarRentalDbContext.cs
    CarRentalDbInitializer.cs
  IInitializer.cs
  InfrastructureConfiguration.cs
  InfrastructureConfiguration.Specs.cs
```

Unfortunately, the new classes will not compile, until you install some test related dependencies like **Microsoft.EntityFrameworkCore.InMemory**:

```
<ItemGroup Condition="'$(Configuration)' == 'Release'">
  <Compile Remove="**\*.Specs.cs" />
  <Compile Remove="**\*.Fakes.cs" />
</ItemGroup>

<ItemGroup Condition="'$(Configuration)' != 'Release'">
  <PackageReference Include="Microsoft.EntityFrameworkCore.InMemory" Version="3.1.4" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.6.1" />
  <PackageReference Include="xunit" Version="2.4.1" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.4.1" />
</ItemGroup>
```

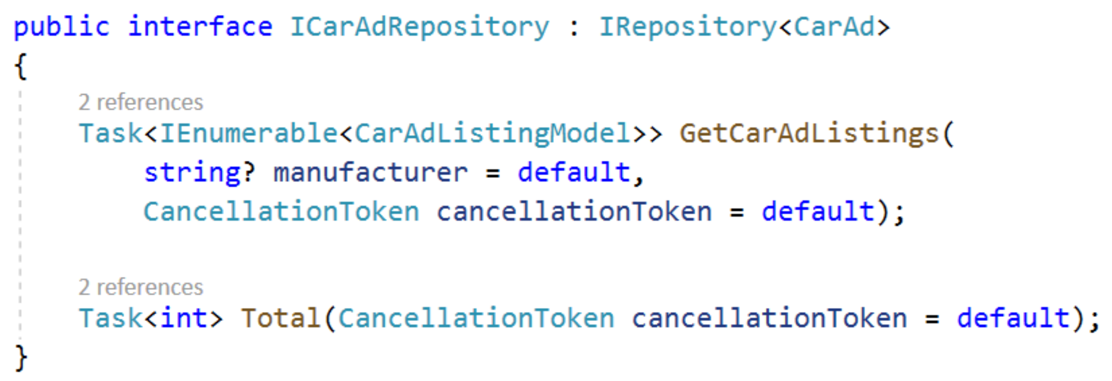
You will find the **CarRentalSystem.Infrastructure.csproj** in the **Code Helpers > CQRS** folder. Run the solution tests to make sure our code is working.

Let us examine some of the code we copied.

The repositories' sole purpose is to call the persistence layer and query the database. Do not put domain or application logic in them. They should be marked as internal, and their registration is done automatically by scanning the assembly:

```
internal static IServiceCollection AddRepositories(this IServiceCollection services)
    => services
        .Scan(scan => scan
            .FromCallingAssembly()
            .AddClasses(classes => classes
                .AssignableTo(typeof(IRepository<>)))
            .AsMatchingInterface()
            .WithTransientLifetime());
```

A repository's interface should not return **IQueryable** collections directly. Use **IEnumerable**. Domain entities are allowed for both input or output. Our **ICarAdRepository** is looking great:



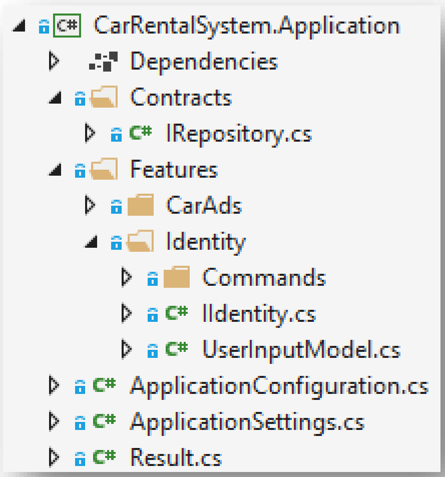
```
public interface ICarAdRepository : IRepository<CarAd>
{
    2 references
    Task<IEnumerable<CarAdListingModel>> GetCarAdListings(
        string? manufacturer = default,
        CancellationToken cancellationToken = default);

    2 references
    Task<int> Total(CancellationToken cancellationToken = default);
}
```

The base repository, however, is not following these rules. It leaks the abstraction by providing **IQueryable** and domain details. Let us fix it:

```
public interface IRepository<out TEntity>
    where TEntity : IAggregateRoot
{
}
```

Remember – the sole purpose of the base repository interface is to provide an anti-corruption layer for our aggregate roots.

While you are at it – move the **IIdentity** interface to the **Features > Identity** folder. Make sure you fix its namespace too.



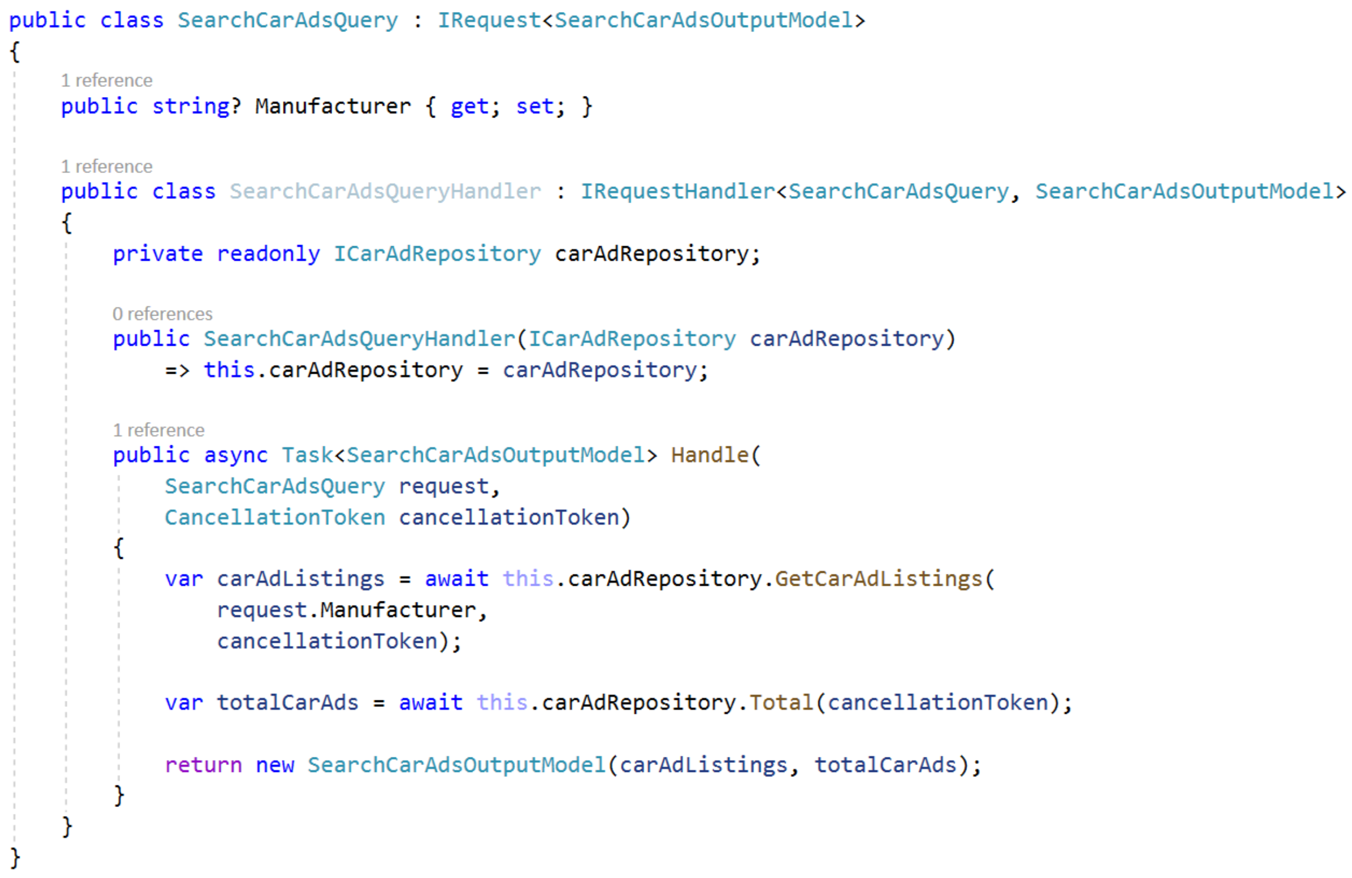
```
CarRentalSystem.Application
    Dependencies
    Contracts
        IRepository.cs
    Features
        CarAds
        Identity
            Commands
            IIdentity.cs
            UserInputModel.cs
    ApplicationConfiguration.cs
    ApplicationSettings.cs
    Result.cs
```

Now make the **DataRepository** abstract and protected:

```
internal abstract class DataRepository<TEntity> : IRepository<TEntity>
    where TEntity : class, IAggregateRoot
{
    private readonly CarRentalDbContext db;

    1 reference
    protected DataRepository(CarRentalDbContext db) => this.db = db;

    1 reference
    protected IQueryable<TEntity> All() => this.db.Set<TEntity>();
}
```

It is time for our query. It should be implemented using the same principles as the command we wrote earlier. There is a small difference though. Since queries contain optional data – there is no need to encapsulate the properties behind a constructor.

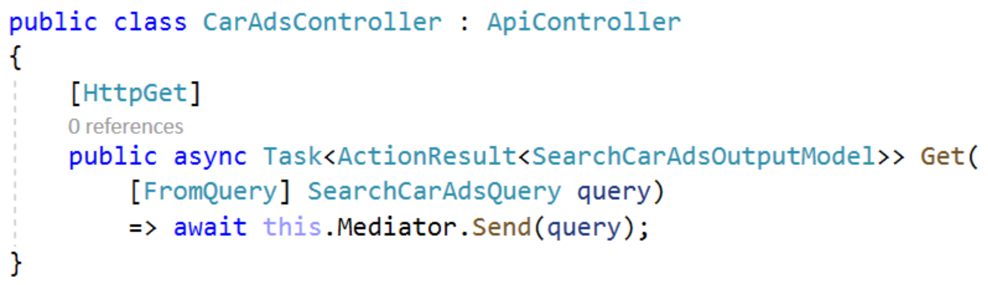
Here is the **SearchCarAdsQuery** code:



```
public class SearchCarAdsQuery : IRequest<SearchCarAdsOutputModel>
{
    1 reference
    public string? Manufacturer { get; set; }

    1 reference
    public class SearchCarAdsQueryHandler : IRequestHandler<SearchCarAdsQuery, SearchCarAdsOutputModel>
    {
        private readonly ICarAdRepository carAdRepository;

        0 references
        public SearchCarAdsQueryHandler(ICarAdRepository carAdRepository)
            => this.carAdRepository = carAdRepository;

        1 reference
        public async Task<SearchCarAdsOutputModel> Handle(
            SearchCarAdsQuery request,
            CancellationToken cancellationToken)
        {
            var carAdListings = await this.carAdRepository.GetCarAdListings(
                request.Manufacturer,
                cancellationToken);

            var totalCarAds = await this.carAdRepository.Total(cancellationToken);

            return new SearchCarAdsOutputModel(carAdListings, totalCarAds);
        }
    }
}
```

The **Manufacturer** property should be a nullable string because it is optional in our logic. Its setter is public, because otherwise the built-in complex type model binder will not be able to transform the *GET* request.

It is time for our **CarAdsController** to slim down a little bit:



```
public class CarAdsController : ApiController
{
    [HttpGet]
    0 references
    public async Task<ActionResult<SearchCarAdsOutputModel>> Get(
        [FromQuery] SearchCarAdsQuery query)
        => await this.Mediator.Send(query);
}
```

Done. Our query is implemented.

<u>Now it is your turn</u>. Implement the **Register** action in the **IdentityController** with CQRS.

Done already? Super easy!

We can further reduce the controller code by introducing these protected methods in the base **ApiController** class:

```
1 reference
protected Task<ActionResult<TResult>> Send<TResult>(IRequest<TResult> request)
    => this.Mediator.Send(request).ToActionResult();

1 reference
protected Task<ActionResult> Send(IRequest<Result> request)
    => this.Mediator.Send(request).ToActionResult();

1 reference
protected Task<ActionResult<TResult>> Send<TResult>(IRequest<Result<TResult>> request)
    => this.Mediator.Send(request).ToActionResult();
```

And then the actions:

```
public class IdentityController : ApiController
{
    [HttpPost]
    [Route(nameof(Register))]
    1 reference
    public async Task<ActionResult> Register(CreateUserCommand command)
        => await this.Send(command);

    [HttpPost]
    [Route(nameof(Login))]
    1 reference
    public async Task<ActionResult<LoginOutputModel>> Login(LoginUserCommand command)
        => await this.Send(command);
}
```

Neat and tidy!

This section’s solution is available in the **Partial Solutions > 8. Simplifying Business Logic with CQRS and MediatR** folder.

9. Integration Tests of Web Features

We implemented a lot of functionalities and wired many tools to work together in a nice and clean solution. It is time to add some integration tests to make sure that our system is robust and stable.

Go to the **Domain** model and install **Bogus** from **NuGet**. It should be included only in testing scenarios. **Bogus** is a library which allows us to create random fake data.

Then copy **CarAd.Fakes**, **Category.Fakes**, **Manufacturer.Fakes** and **Options.Fakes** from **Code Helpers > Integration Tests** to **Domain > Models > CarAds**.

We are going to use **MyTested.AspNetCore.Mvc** and **FluentAssertions** for our integration tests. Go to the **Startup** project and prepare it:

```
<ItemGroup Condition="'$(Configuration)' == 'Release'">
  <Compile Remove="**\*.Specs.cs" />
</ItemGroup>

<ItemGroup Condition="'$(Configuration)' != 'Release'">
  <PackageReference Include="FluentAssertions" Version="5.10.3" />
  <PackageReference Include="MyTested.AspNetCore.Mvc.Universe" Version="3.1.2" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.6.1" />
  <PackageReference Include="xunit" Version="2.4.1" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.4.1" />
</ItemGroup>
```

The above code is provided for you in the **Code Helpers > Integration Tests** folder.

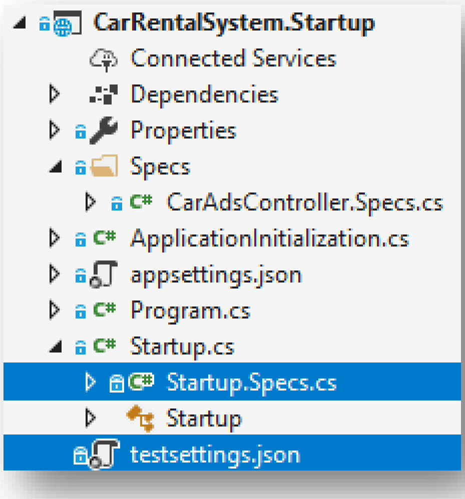
By default, the *.NET SDK* generates a **Program.cs** file with a **Main** method for all unit test assemblies. However, we already have an entry point in our **Startup** project, so we need to disable that compilation feature:

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.1</TargetFramework>
  <LangVersion>8.0</LangVersion>
  <Nullable>enable</Nullable>
  <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  <GenerateProgramFile>false</GenerateProgramFile>
</PropertyGroup>
```

First things first. **MyTested.AspNetCore.Mvc** requires a **TestStartup** class and a **testsettings.json** file to be present at the root of the test project. Copy them from the **Code Helpers > Integration Tests** folder. Please note that the **TestStartup** class is in the **Startup.Specs** file.

Now add a **Specs** folder and create a **CarAdsController.Specs.cs** file in it.

This should be the structure of your **Startup** project:



```
CarRentalSystem.Startup
  Connected Services
  Dependencies
  Properties
  Specs
    CarAdsController.Specs.cs
  ApplicationInitialization.cs
  appsettings.json
  Program.cs
  Startup.cs
    Startup.Specs.cs
    Startup
  testsettings.json
```

The **testsettings.json** file contains fake application settings and a hint to **MyTested.AspNetCore.Mvc** that our web project is the same as our test one:

```
"General": {
  "WebAssemblyName": "CarRentalSystem.Startup"
},
"ApplicationSettings": {
  "Secret": "My Fake Secret"
},
"ConnectionStrings": {
  "DefaultConnection": "My Fake Connection String"
}
```

We need to include the **testsettings.json** file only when the project compilation is not in **Release** configuration:

```
<ItemGroup Condition="'$(Configuration)' == 'Release'">
  <Compile Remove="**\*.Specs.cs" />
  <Content Update="testsettings.json">
    <CopyToOutputDirectory>Never</CopyToOutputDirectory>
  </Content>
</ItemGroup>
```

Good! Now go to the **CarAdsControllerSpecs** class to write your first integration tests. **MyTested.AspNetCore.Mvc** automatically mocks our database, so we just need to provide fake data. You can use the **CarAdFakes.Data.GetCarAds** method.

We need three tests for now – one for an empty query, one for validating whether only the available car ads are returned, and one for asserting whether a valid query filters the results.

Here are the first two:

```
[Theory]
[InlineData(2)]
0 references
public void GetShouldReturnAllCarAdsWithoutAQuery(int totalCarAds)
    => MyController<CarAdsController>
        .Instance(instance => instance
            .WithData(A.CollectionOfDummy<CarAd>(totalCarAds)))

        .Calling(c => c.Get(With.Empty<SearchCarAdsQuery>()))

        .ShouldReturn()
        .ActionResult<SearchCarAdsOutputModel>(result => result
            .Passing(model => model
                .CarAds.Count().Should().Be(totalCarAds)));

[Fact]
0 references
public void GetShouldReturnAvailableCarAdsWithoutAQuery()
    => MyController<CarAdsController>
        .Instance(instance => instance
            .WithData(CarAdFakes.Data.GetCarAds()))

        .Calling(c => c.Get(With.Empty<SearchCarAdsQuery>()))

        .ShouldReturn()
        .ActionResult<SearchCarAdsOutputModel>(result => result
            .Passing(model => model
                .CarAds.Count().Should().Be(10)));
```

A good idea is to separate the different AAA parts with a blank line. Try to write the third method by yourself.

The tests above do not validate the route configuration of the asserted actions. Let us refactor them to include the full request pipeline. This is the first one:

```
[Theory]
[InlineData(2)]
✓ | 0 references
public void GetShouldReturnAllCarAdsWithoutAQuery(int totalCarAds)
    => MyPipeline
        .Configuration()
        .ShouldMap("/CarAds")

        .To<CarAdsController>(c => c.Get(With.Empty<SearchCarAdsQuery>()))
        .Which(instance => instance
            .WithData(A.CollectionOfDummy<CarAd>(totalCarAds)))

        .ShouldReturn()
        .ActionResult<SearchCarAdsOutputModel>(result => result
            .Passing(model => model
                .CarAds.Count().Should().Be(totalCarAds)));
```

Refactor the other two.

These tests are straightforward because they do not require mocking. Let us assert that the **Login** action in the **IdentityController** returns a token when a user enters valid credentials.

Copy **IdentityService**, **IdentityService.Fakes**, **IJwtTokenGenerator**, **JwtTokenGeneratorService** and **JwtTokenGeneratorService.Fakes** from **Code Helpers > Integration Tests** to **Infrastructure > Identity**.

If you examine the files, you will see that the token generation is extracted in a separate service class. Go to **InfrastructureConfiguration** to register it:

```
services.AddTransient<IIdentity, IdentityService>();
services.AddTransient<IJwtTokenGenerator, JwtTokenGeneratorService>();
```

Fake objects for **UserManager<User>** and **IJwtTokenGenerator** are also provided. Before using them in a test, we need to register them in the test services which **MyTested.AspNetCore.Mvc** is using.

Go to the **TestStartup** class and write the following:

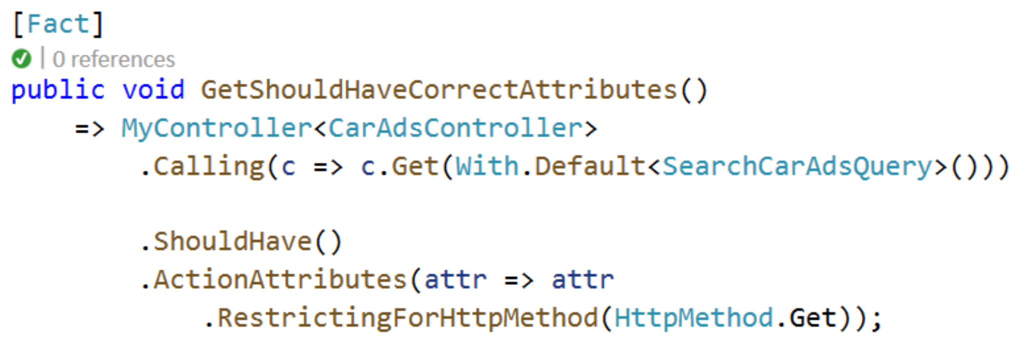
```
public class TestStartup : Startup
{
    0 references
    public TestStartup(IConfiguration configuration)
        : base(configuration)
    {
    }

    0 references
    public void ConfigureTestServices(IServiceCollection services)
    {
        base.ConfigureServices(services);

        ValidateServices(services);

        services.ReplaceTransient<UserManager<User>>(_ => IdentityFakes.FakeUserManager);
        services.ReplaceTransient<IJwtTokenGenerator>(_ => JwtTokenGeneratorFakes.FakeJwtTokenGenerator);
    }

    1 reference
    private static void ValidateServices(IServiceCollection services)
    {
        var provider = services.BuildServiceProvider();

        provider.GetRequiredService<ICarAdFactory>();
        provider.GetRequiredService<IMediator>();
        provider.GetRequiredService<ICarAdRepository>();
        provider.GetRequiredService<IControllerFactory>();
    }
}
```

We are using the **TestStartup** for two purposes – replacing hard to test services with fake ones and validating that all application parts are registered.

Now that we provided the needed mocks, we can write our test. Create **IdentityController.Specs** in **Startup > Specs** and add the following code:

```
[Theory]
[InlineData(
    IdentityFakes.TestEmail,
    IdentityFakes.ValidPassword,
    JwtTokenGeneratorFakes.ValidToken)]
| 0 references
public void LoginShouldReturnToken(string email, string password, string token)
    => MyPipeline
        .Configuration()
        .ShouldMap(request => request
            .WithLocation("/Identity/Login")
            .WithMethod(HttpMethod.Post)
            .WithJsonBody(new
            {
                Email = email,
                Password = password
            }))

        .To<IdentityController>(c => c
            .Login(new LoginUserCommand(email, password)))

        .Which()
        .ShouldReturn()
        .ActionResult<LoginOutputModel>(result => result
            .Passing(model => model.Token.Should().Be(token)));
```

We can even add unit tests for our controllers like so:



```
[Fact]
✓ | 0 references
public void GetShouldHaveCorrectAttributes()
    => MyController<CarAdsController>
        .Calling(c => c.Get(With.Default<SearchCarAdsQuery>()))

        .ShouldHave()
        .ActionAttributes(attr => attr
            .RestrictingForHttpMethod(HttpMethod.Get));
```

<u>Validate the Login and Register attributes too</u>.

Good job! Our code improved a lot by introducing these tests.

You can find the tests from this section in the **Partial Solutions > 9. Integration Tests of Web Features** folder.

10. Creating Entities and Adding Validation

As per our requirements – all registered users should be dealers. And all dealers should be able to add car ads. Let us implement these functionalities. Additionally, our application does not have any validation logic – we should add one as soon as possible because… Well, you know, validation is important…

Go to **Application > Features > Identity** and create an **IUser** interface. We need to expose the **User** object in our commands:

```
public interface IUser
{
    2 references
    void BecomeDealer(Dealer dealer);
}
```

Our **User** class should implement the above interface.

Go to **Infrastructure > Identity** and make the **IdentityService** return a **Result<IUser>**:

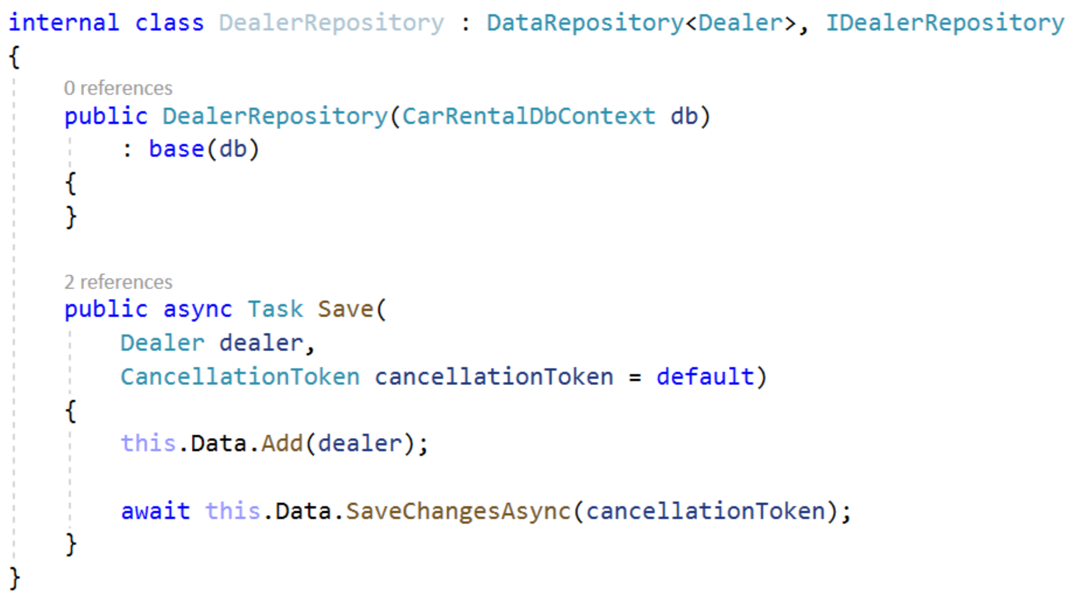
```
public async Task<Result<IUser>> Register(UserInputModel userInput)
{
    var user = new User(userInput.Email);

    var identityResult = await this.userManager.CreateAsync(user, userInput.Password);

    var errors = identityResult.Errors.Select(e => e.Description);

    return identityResult.Succeeded
        ? Result<IUser>.SuccessWith(user)
        : Result<IUser>.Failure(errors);
}
```

Fix the **IIdentity** interface too.

Create a **Dealers** folder in **Application > Features**. Add **IDealerRepository** in it:

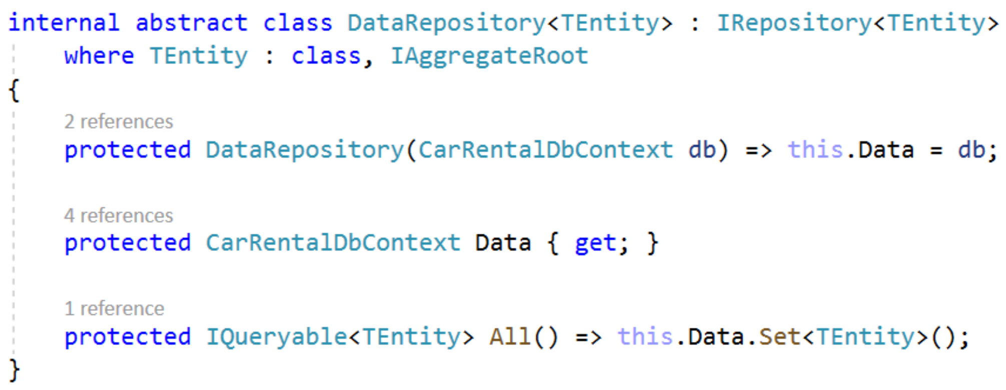
```
public interface IDealerRepository : IRepository<Dealer>
{
    2 references
    Task Save(Dealer dealer, CancellationToken cancellationToken = default);
}
```

Implement the above interface in **Infrastructure > Persistence > Repositories**:



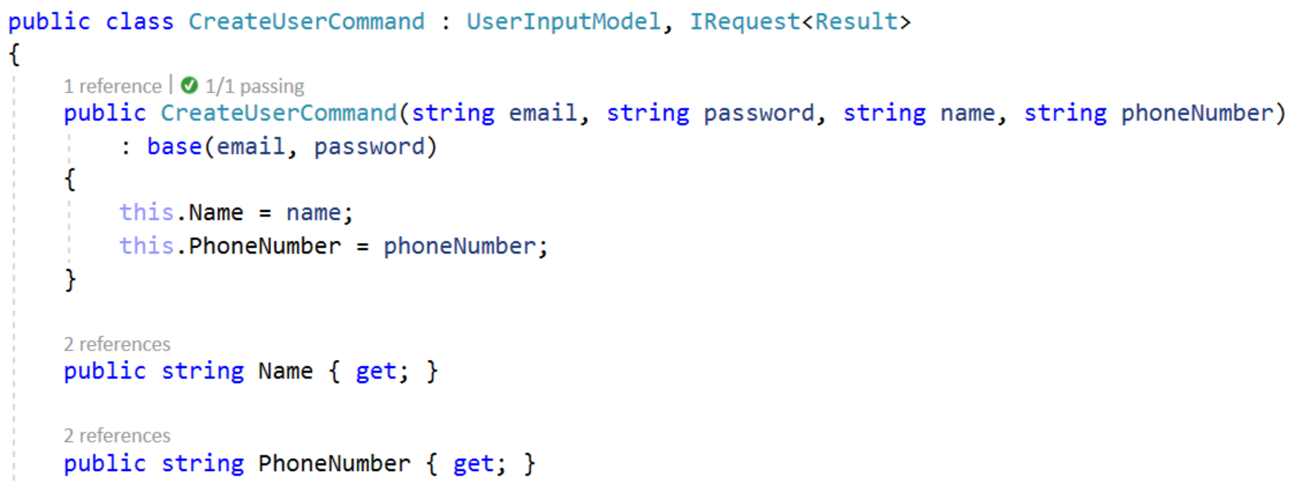
```
internal class DealerRepository : DataRepository<Dealer>, IDealerRepository
{
    0 references
    public DealerRepository(CarRentalDbContext db)
        : base(db)
    {
    }

    2 references
    public async Task Save(
        Dealer dealer,
        CancellationToken cancellationToken = default)
    {
        this.Data.Add(dealer);

        await this.Data.SaveChangesAsync(cancellationToken);
    }
}
```

You will need to change the base **DataRepository** to expose the **DbContext**:



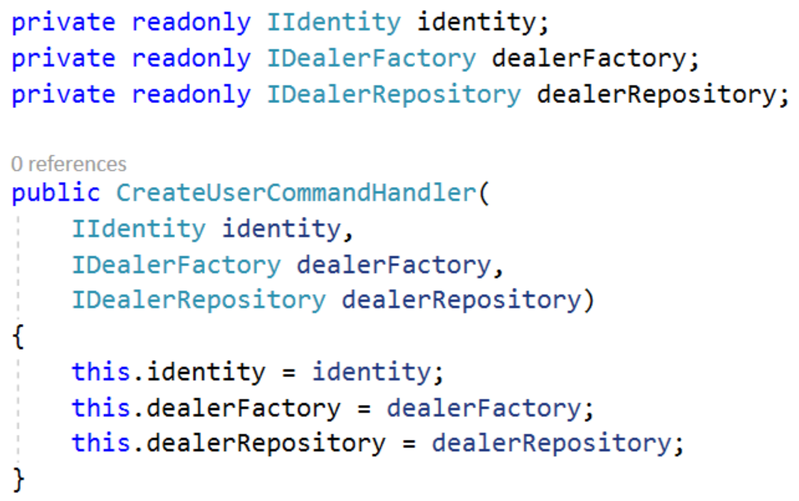
```
internal abstract class DataRepository<TEntity> : IRepository<TEntity>
    where TEntity : class, IAggregateRoot
{
    2 references
    protected DataRepository(CarRentalDbContext db) => this.Data = db;

    4 references
    protected CarRentalDbContext Data { get; }

    1 reference
    protected IQueryable<TEntity> All() => this.Data.Set<TEntity>();
}
```

All preparations are ready. Now go to the **Application > Features > Identity > Commands > CreateUser**. Update the **CreateUserCommand** to receive a name and a phone number:



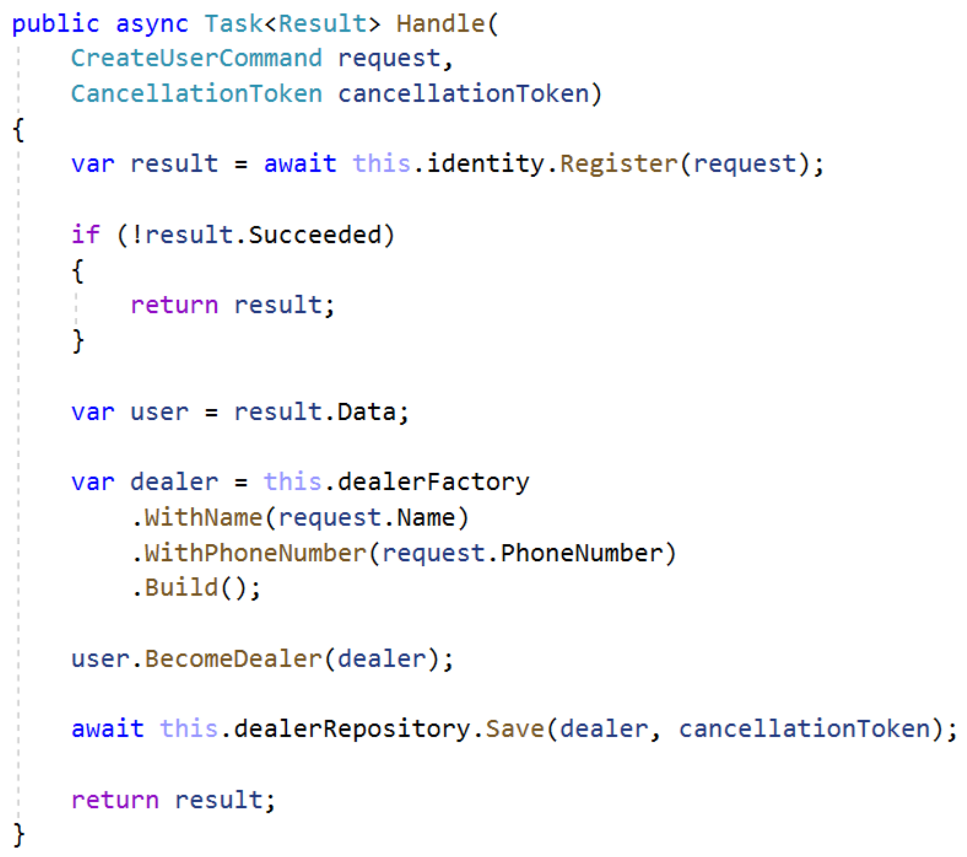
```
public class CreateUserCommand : UserInputModel, IRequest<Result>
{
    1 reference | ✓ 1/1 passing
    public CreateUserCommand(string email, string password, string name, string phoneNumber)
        : base(email, password)
    {
        this.Name = name;
        this.PhoneNumber = phoneNumber;
    }

    2 references
    public string Name { get; }

    2 references
    public string PhoneNumber { get; }
```

Finally, we should update the handler's logic. It should create a user, then create a dealer, then update the user and save it to the database. We need to inject **IIdentity**, **IDealerFactory** and **IDealerRepository**:



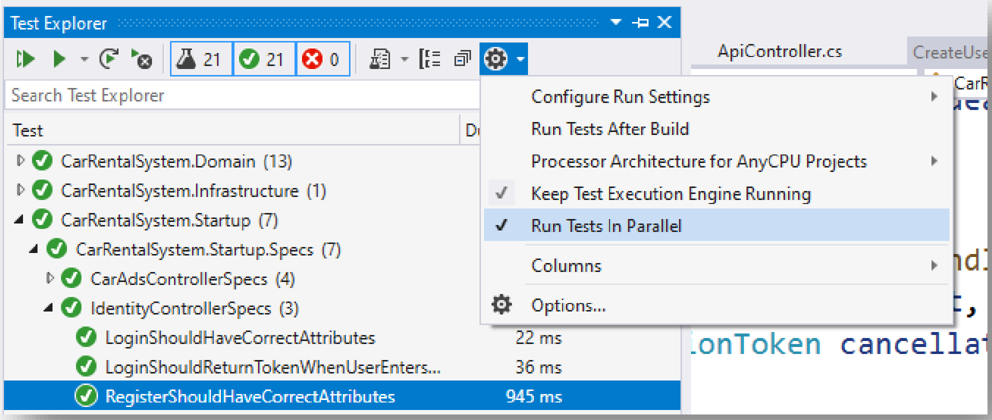
```
private readonly IIdentity identity;
private readonly IDealerFactory dealerFactory;
private readonly IDealerRepository dealerRepository;

0 references
public CreateUserCommandHandler(
    IIdentity identity,
    IDealerFactory dealerFactory,
    IDealerRepository dealerRepository)
{
    this.identity = identity;
    this.dealerFactory = dealerFactory;
    this.dealerRepository = dealerRepository;
}
```

Here is the logic:



```
public async Task<Result> Handle(
    CreateUserCommand request,
    CancellationToken cancellationToken)
{
    var result = await this.identity.Register(request);

    if (!result.Succeeded)
    {
        return result;
    }

    var user = result.Data;

    var dealer = this.dealerFactory
        .WithName(request.Name)
        .WithPhoneNumber(request.PhoneNumber)
        .Build();

    user.BecomeDealer(dealer);

    await this.dealerRepository.Save(dealer, cancellationToken);

    return result;
}
```

Try to build the solution and fix any failing unit test. You may turn on the **"Run Tests In Parallel"** feature for better performance:



```
CarRentalSystem.Domain (13)
CarRentalSystem.Infrastructure (1)
CarRentalSystem.Startup (7)
  CarRentalSystem.Startup.Specs (7)
    CarAdsControllerSpecs (4)
    IdentityControllerSpecs (3)
      LoginShouldHaveCorrectAttributes            22 ms
      LoginShouldReturnTokenWhenUserEnters...     36 ms
      RegisterShouldHaveCorrectAttributes        945 ms
```

Now we are going to create car ads, but first we will need some preparations. Car ad categories should be seeded to the database before the application is used by the end users. We are now going to add infrastructure for data seeding.

Add an **IInitialData** interface to the **Domain > Common** folder:

```
public interface IInitialData
{
    2 references
    Type EntityType { get; }

    2 references
    IEnumerable<object> GetData();
}
```

Then copy the **Category.Data** file from **Code Helpers > Creating Entities** to **Domain > Models > CarAds**.

Register the category initial data class in the **DomainConfiguration**:

```
public static IServiceCollection AddDomain(this IServiceCollection services)
    => services
        .Scan(scan => scan
            .FromCallingAssembly()
            .AddClasses(classes => classes
                .AssignableTo(typeof(IFactory<>)))
            .AsMatchingInterface()
            .WithTransientLifetime())
        .AddTransient<IInitialData, CategoryData>();
```

Finally, replace the **CarRentalDbInitializer** with the one provided in the **Code Helpers > Creating Entities** folder.

Running the application should populate the category data.

We can use the **CategoryData** class to add more domain logic in our **CarAd** entity:

```
private static readonly IEnumerable<Category> AllowedCategories
    = new CategoryData().GetData().Cast<Category>();

2 references
internal CarAd(
    Manufacturer manufacturer,
    string model,
    Category category,
    string imageUrl,
    decimal pricePerDay,
    Options options,
    bool isAvailable)
{
    this.Validate(model, imageUrl, pricePerDay);
    this.ValidateCategory(category);

    this.Manufacturer = manufacturer;
    this.Model = model;
```

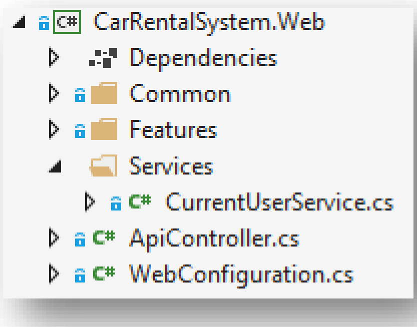
And the **ValidateCategory** method:

```
private void ValidateCategory(Category category)
{
    var categoryName = category.Name;

    if (AllowedCategories.Any(c => c.Name == categoryName))
    {
        return;
    }

    var allowedCategoryNames = string.Join(", ", AllowedCategories.Select(c => $"'{c.Name}'"));

    throw new InvalidCarAdException(
        $"'{categoryName}' is not a valid category. Allowed values are: {allowedCategoryNames}.");
}
```

We will also need to know the current request user to create a car ad. Add the following interface to **Application > Contracs**:

```
public interface ICurrentUser
{
    3 references
    string UserId { get; }
}
```

The implementation of that interface should come from the web project since it is the only one familiar with request data. Create a folder **Services** and add **CurrentUserService** in it:



Here is the implementation:

```
public class CurrentUserService : ICurrentUser
{
    0 references
    public CurrentUserService(IHttpContextAccessor httpContextAccessor)
    {
        var user = httpContextAccessor.HttpContext?.User;

        if (user == null)
        {
            throw new InvalidOperationException(
                "This request does not have an authenticated user.");
        }

        this.UserId = user.FindFirstValue(ClaimTypes.NameIdentifier);
    }

    3 references
    public string UserId { get; }
}
```

Easy enough - we just need to search for the proper identifier claim if the authentication user identity.

Now, it is your turn. Create a command for creating car ads. It should contain the following properties:

```
public CreateCarAdCommand(
    string manufacturer,
    string model,
    int category,
    string imageUrl,
    decimal pricePerDay,
    bool climateControl,
    int numberOfSeats,
    int transmissionType)
{
```

For the handler you will need these injected services:

```
public CreateCarAdCommandHandler(
    ICurrentUser currentUser,
    IDealerRepository dealerRepository,
    ICarAdRepository carAdRepository,
    ICarAdFactory carAdFactory)
```

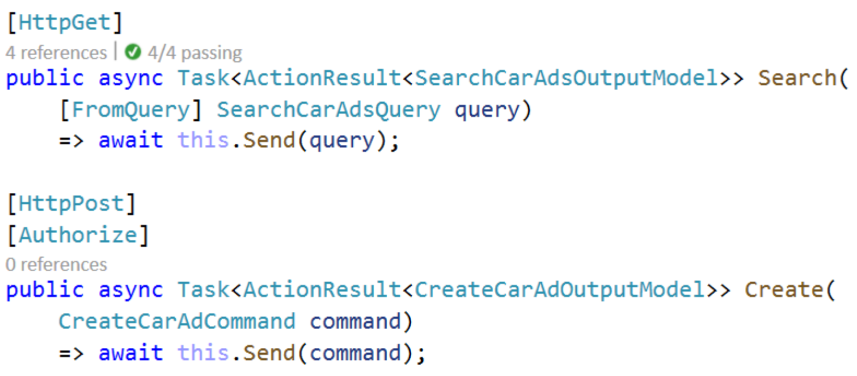
And this is the business logic:

```
var userId = this.currentUser.UserId;
var dealer = await this.dealerRepository
    .FindByUser(userId, cancellationToken);

var category = await this.carAdRepository
    .GetCategory(request.Category, cancellationToken);

var manufacturer = await this.carAdRepository
    .GetManufacturer(request.Manufacturer, cancellationToken);

var factory = manufacturer == null
    ? this.carAdFactory.WithManufacturer(request.Manufacturer)
    : this.carAdFactory.WithManufacturer(manufacturer);

var carAd = factory
    .WithModel(request.Model)
    .WithCategory(category)
    .WithImageUrl(request.ImageUrl)
    .WithPricePerDay(request.PricePerDay)
    .WithOptions(
        request.ClimateControl,
        request.NumberOfSeats,
        Enumeration.FromValue<TransmissionType>(request.TransmissionType))
    .Build();

dealer.AddCarAd(carAd);

await this.carAdRepository.Save(carAd, cancellationToken);

return new CreateCarAdOutputModel(carAd.Id);
```

You need to implement the missing methods by yourself.

Here are some hints:

- The user cannot create a category. Only the initial categories can be used.
- A manufacturer can be created, only if its name does not exist in the database. Otherwise, the manufacturer should be reused.
- The command should return the **ID** of the created car ad. Wrap it in an output model.
- You may extract a common **Save** method to the base **IRepository** interface.
- You may delete the unnecessary methods in the **ICarAdFactory** interface.
- Do not forget to use the **Authorize** attribute on the new controller action:



```
[HttpGet]
4 references | ✓ 4/4 passing
public async Task<ActionResult<SearchCarAdsOutputModel>> Search(
    [FromQuery] SearchCarAdsQuery query)
    => await this.Send(query);

[HttpPost]
[Authorize]
0 references
public async Task<ActionResult<CreateCarAdOutputModel>> Create(
    CreateCarAdCommand command)
    => await this.Send(command);
```

You may notice the **Get** method is now renamed to **Search**. It is always a good idea to give meaningful action names, which are tightly connected to the business logic, and not the technology itself. In this case, **Search** is way better than **Get**.

The above naming rule should be applied to all objects and methods.

Now, let us add a validation convention.

Install **FluentValidation.AspNetCore** to the **Web** project, and **FluentValidation** to the **Application** one.

Copy these files from the **Code Helpers > Creating Entities** folder:

- **RequestValidationBehavior** to **Application > Behaviours** (create the folder).
- **ModelValidationException** and **NotFoundException** to **Application > Exceptions** (create the folder).
- **ValidationExceptionHandlerMiddleware** to **Web > Middleware** (create the folder).

Go to **ApplicationConfiguration** and register the **RequestValidationBehavior**:

```
services
 .Configure<ApplicationSettings>(
     configuration.GetSection(nameof(ApplicationSettings)),
     options => options.BindNonPublicProperties = true)
 .AddMediatR(Assembly.GetExecutingAssembly())
 .AddTransient(typeof(IPipelineBehavior<,>), typeof(RequestValidationBehavior<,>));
```

Then go to **WebConfiguration** and register **FluentValidation**:

```
public static IServiceCollection AddWebComponents(this IServiceCollection services)
{
    services
        .AddScoped<ICurrentUser, CurrentUserService>()
        .AddControllers()
        .AddFluentValidation(validation => validation
            .RegisterValidatorsFromAssemblyContaining<Result>())
        .AddNewtonsoftJson();

    services.Configure<ApiBehaviorOptions>(options =>
    {
        options.SuppressModelStateInvalidFilter = true;
    });

    return services;
}
```

Finally, update the **Configure** method in the **Startup** class to use the new middleware:

```
app
    .UseValidationExceptionHandler()
    .UseHttpsRedirection()
    .UseRouting()
    .UseAuthentication()
    .UseAuthorization()
    .UseEndpoints(endpoints => endpoints
        .MapControllers())
    .Initialize();
```

This is what we did:

- We turned off the default validation and added **FluentValidation** to the application services.
- We added a pipeline behavior to **MediatR**. Every time we send a command or a query, it will be executed against that behavior. The later will run all registered validators in the system against the request object.

- If an error is found, the behavior will throw an exception, and the request pipeline will not continue.
- The exception handling middleware will catch the exception and serialize a friendly error JSON object for the client to consume.

There are three types of error responses in our application:

- An exception – we return its message as a collection of one error.
- A logic error – we return a **Result** or a **Result<T>** object with a collection of errors.
- A validation error – we return a collection of properties with their errors. In this case, we add one additional property – **ValidationDetails** to indicate that we are returning a nested collection.

This way, our API returns error messages according to a convention and the client will have an easy for implementation validation handler.

Since our validation infrastructure is ready, we can now add command validators.

Go to **Application > Features > Identity > Commands > CreateUser** and add a **CreateUserCommandValidator** class:

```
using static Domain.Models.ModelConstants.Common;

1 reference
public class CreateUserCommandValidator : AbstractValidator<CreateUserCommand>
{
    0 references
    public CreateUserCommandValidator()
    {
        this.RuleFor(u => u.Email)
            .MinimumLength(MinEmailLength)
            .MaximumLength(MaxEmailLength)
            .EmailAddress()
            .NotEmpty();

        this.RuleFor(u => u.Password)
            .MaximumLength(MaxNameLength)
            .NotEmpty();

        this.RuleFor(u => u.Name)
            .MinimumLength(MinNameLength)
            .MaximumLength(MaxNameLength)
            .NotEmpty();
    }
}
```

It is your turn. Add a regular expression validation for the phone number.

With a separate validator class, we can easily implement advanced logic. For example, in our car ad validator we can inject services and do something like:

```
public CreateCarAdCommandValidator(ICarAdRepository carAdRepository)
{
    this.RuleFor(c => c.Category)
        .MustAsync(async (category, token) => await carAdRepository
            .GetCategory(category, token) != null)
        .WithMessage("'{PropertyName}' does not exist.");

    this.RuleFor(c => c.ImageUrl)
        .Must(url => Uri.IsWellFormedUriString(url, UriKind.RelativeOrAbsolute))
        .WithMessage("'{PropertyName}' must be a valid url.")
        .NotEmpty();

    this.RuleFor(c => c.TransmissionType)
        .Must(BeAValidTransmissionType)
        .WithMessage("'{PropertyName}' is not a valid transmission type.");
}
```

This is the **BeAValidTransmissionType** method:

```
private static bool BeAValidTransmissionType(int transmissionType)
{
    try
    {
        Enumeration.FromValue<TransmissionType>(transmissionType);
        return true;
    }
    catch
    {
        return false;
    }
}
```

Validation ready! For consistency, you may want to change the domain exception messages in the **Guard** class to have the single quotes too.

You will need to fix the failing tests after the introduced validation changes.

You can find this section's solution in the **Partial Solutions > 10. Creating Entities and Adding Validation**.

11. Query Enhancements

In this section, we are going to improve our queries. Let us start with the **Select** statements. With big output models, they become complicated and troublesome to write. Using third-party tools like **AutoMapper** makes perfect sense because it will save us a lot of time. We will now integrate it into our architecture.

Install **AutoMapper.Extensions.Microsoft.DependencyInjection** (the last one must close the door) to the **Application** project. Then create a **Mapping** folder in it.

Copy the **MappingProfile** class from the **Code Helpers > Query Enhancements** folder. This class is responsible to find all mappings in our solution by convention and register them so they will be executed automatically.

Now add the following **IMapFrom** interface again to the **Mapping** folder:

```
public interface IMapFrom<T>
{
    void Mapping(Profile mapper) => mapper.CreateMap(typeof(T), this.GetType());
}
```

We are using a new *C#* feature here – default interface methods.

Go to **ApplicationConfiguration** and register **AutoMapper**:

```
public static IServiceCollection AddApplication(
    this IServiceCollection services,
    IConfiguration configuration)
    => services
        .Configure<ApplicationSettings>(
            configuration.GetSection(nameof(ApplicationSettings)),
            options => options.BindNonPublicProperties = true)
        .AddAutoMapper(Assembly.GetExecutingAssembly())
        .AddMediatR(Assembly.GetExecutingAssembly())
        .AddTransient(typeof(IPipelineBehavior<,>), typeof(RequestValidationBehavior<,>));
```

Our infrastructure code is ready. Let us now define a mapping.

Go to **CarAdListingModel** and remove the constructor. **AutoMapper** words with private setters, so we are now going to encapsulate the class with them:

```
public class CarAdListingModel
{
    0 references
    public int Id { get; private set; }

    1 reference
    public string Manufacturer { get; private set; } = default!;

    0 references
    public string Model { get; private set; } = default!;

    0 references
    public string ImageUrl { get; private set; } = default!;

    0 references
    public string Category { get; private set; } = default!;

    0 references
    public decimal PricePerDay { get; private set; }
}
```

To define a mapping in our infrastructure, all you need to do is add the **IMapFrom** interface:

```
public class CarAdListingModel : IMapFrom<CarAd>
```

The default interface **Mapping** method will configure the map conventionally for us. If we want a custom logic, we can implement it manually:

```
public string Category { get; private set; } = default!;

0 references
public decimal PricePerDay { get; private set; }

1 reference
public void Mapping(Profile mapper)
    => mapper
        .CreateMap<CarAd, CarAdListingModel>()
        .ForMember(ad => ad.Manufacturer, cfg => cfg
            .MapFrom(ad => ad.Manufacturer.Name))
        .ForMember(ad => ad.Category, cfg => cfg
            .MapFrom(ad => ad.Category.Name));
```

Finally, in our **CarAdRepository**, we need to inject **IMapper** and project our query with it:

```
public async Task<IEnumerable<CarAdListingModel>> GetCarAdListings(
    string? manufacturer = default,
    CancellationToken cancellationToken = default)
{
    var query = this.AllAvailable();

    if (!string.IsNullOrWhiteSpace(manufacturer))
    {
        query = query
            .Where(car => EF
                .Functions
                .Like(car.Manufacturer.Name, $"%{manufacturer}%"));
    }

    return await this.mapper
        .ProjectTo<CarAdListingModel>(query)
        .ToListAsync(cancellationToken);
}
```

An important rule to follow when using **AutoMapper** is to never map from input objects to domain entities. We should always use the domain methods and logic to do that transformation. It is perfectly fine to do the opposite – from domain objects to output models.

Let us move on with the queries. According to the requirements, we need to filter the car ads by category, manufacturer, and price range. For now, we only have part of that implemented.

Go to the **SearchCarAdsQuery** class and add all the other properties:

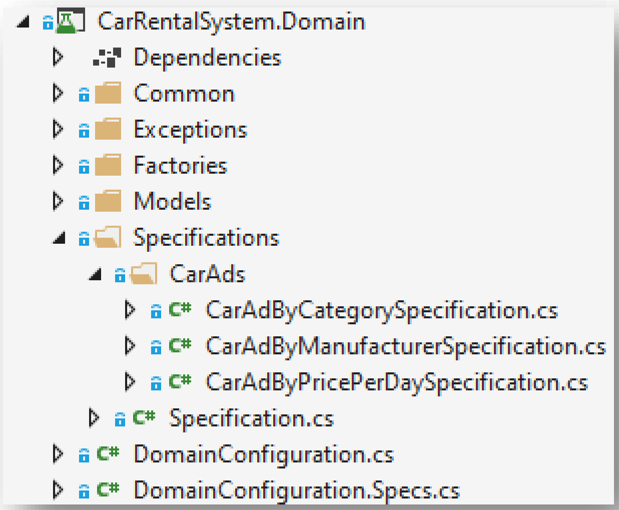
```
public class SearchCarAdsQuery : IRequest<SearchCarAdsOutputModel>
{
    2 references
    public string? Manufacturer { get; set; }

    1 reference
    public int? Category { get; set; }

    1 reference
    public decimal? MinPricePerDay { get; set; }

    1 reference
    public decimal? MaxPricePerDay { get; set; }
```

Unfortunately, the current implementation in the **CarAdRepository** will bring too many conditional statements, which is a code smell and bad design.

We will now introduce the *Specification* design pattern, which will allow us to abstract our complex queries.

Copy the **Specification** file from **Code Helpers > Query Enhancements** to **Domain > Specifications** (create the folder).

Now create the following files:


```
CarRentalSystem.Domain
  Dependencies
  Common
  Exceptions
  Factories
  Models
  Specifications
    CarAds
      CarAdByCategorySpecification.cs
      CarAdByManufacturerSpecification.cs
      CarAdByPricePerDaySpecification.cs
    Specification.cs
  DomainConfiguration.cs
  DomainConfiguration.Specs.cs
```

We have added three specifications – **CarAdByCategorySpecification**, **CarAdByManufacturerSpecification**, and **CarAdByPricePerDaySpecification**.

Here is the manufacturer one:

```
public class CarAdByManufacturerSpecification : Specification<CarAd>
{
    private readonly string? manufacturer;

    1 reference
    public CarAdByManufacturerSpecification(string? manufacturer)
        => this.manufacturer = manufacturer;

    6 references
    protected override bool Include => this.manufacturer != null;

    6 references
    public override Expression<Func<CarAd, bool>> ToExpression()
        => carAd => carAd.Manufacturer.Name.ToLower()
            .Contains(this.manufacturer!.ToLower());
}
```

Basically, we need to inherit the base **Specification** abstract class, and then override **ToExpression**, in which we must provide the query. If we need to use the specification in a domain class – we can do it though the **IsSatisfiedBy** method.

Additionally, if we want to specify for which situations the query should not execute (null values, for example), we can override the **Include** property too.

And this is the **CarAdByPricePerDaySpecification** implementation:

```
public class CarAdByPricePerDaySpecification : Specification<CarAd>
{
    private readonly decimal minPrice;
    private readonly decimal maxPrice;

    1 reference
    public CarAdByPricePerDaySpecification(
        decimal? minPrice = default,
        decimal? maxPrice = decimal.MaxValue)
    {
        this.minPrice = minPrice ?? default;
        this.maxPrice = maxPrice ?? decimal.MaxValue;
    }

    6 references
    public override Expression<Func<CarAd, bool>> ToExpression()
        => carAd => this.minPrice < carAd.PricePerDay && carAd.PricePerDay < this.maxPrice;
}
```

Implement the **CarAdByCategorySpecification** by yourself and then let us see how to use these specification classes.

Go to the **CarAdRepository**, and change the **GetCarAdListings** method:

```
public async Task<IEnumerable<CarAdListingModel>> GetCarAdListings(
    Specification<CarAd> specification,
    CancellationToken cancellationToken = default)
    => await this.mapper
        .ProjectTo<CarAdListingModel>(this
            .AllAvailable()
            .Where(specification))
        .ToListAsync(cancellationToken);
```

Much cleaner that numerous if-else conditional statements. You should also fix the **ICarAdRepository** method.

As a final step, go to the **SearchCarAdsQuery** to provide the specifications:

```
var carAdSpecification = new CarAdByManufacturerSpecification(request.Manufacturer)
    .And(new CarAdByCategorySpecification(request.Category))
    .And(new CarAdByPricePerDaySpecification(request.MinPricePerDay, request.MaxPricePerDay));

var carAdListings = await this.carAdRepository.GetCarAdListings(
    carAdSpecification,
    cancellationToken);

var totalCarAds = await this.carAdRepository.Total(cancellationToken);
```

Done! We improved our codebase a lot. Keep in mind that there is no need to change every query to specification classes. For example, these queries would be an overkill for the specification pattern:

```
public async Task<Category> GetCategory(
    int categoryId,
    CancellationToken cancellationToken = default)
    => await this
        .Data
        .Categories
        .FirstOrDefaultAsync(c => c.Id == categoryId, cancellationToken);

2 references
public async Task<Manufacturer> GetManufacturer(
    string manufacturer,
    CancellationToken cancellationToken = default)
    => await this
        .Data
        .Manufacturers
        .FirstOrDefaultAsync(m => m.Name == manufacturer, cancellationToken);
```

Before we conclude the query enhancements, make sure you fix the failing unit test in the **InfrastructureConfigurationSpecs** class:

```
// Act
var services = serviceCollection
    .AddAutoMapper(Assembly.GetExecutingAssembly())
    .AddRepositories()
    .BuildServiceProvider();
```

You can find the solution of this section in the **Partial Solutions > 11. Query Enhancements** folder.

12. Fulfilling the Requirements

In this section, you are on your own. Implement all system requirements:

- **GET /Dealers/{id}** – Returns the dealer ID, name, phone number, and the total number of car ads. Public route**.**
- **PUT /Dealers/{id}** – A dealer can edit her name and phone number. Private route.
- **GET /CarAds** – Should have paging and sorting by price or by manufacturer. Should have filtering by dealer's name. Should not return availability. Public route.
- **GET /CarAds/Mine** – Returns only the car ads created by the currently authenticated dealer. Includes availability too. It should allow the same filtering and sorting options like the **GET /CarAds** action. Private route.
- **GET /CarAds/Categories** – Returns all categories. The data should include their ID, name, description, and the total number of car ads in each category. Public route.
- **GET /CarAds/{id}**– Returns everything except availability about the car ad, including its dealer's ID, name, and phone number. Public route.
- **PUT /CarAds/{id}** – A dealer can edit her car ads. Everything except availability should be editable. Private route.
- **PUT /CarAds/{id}/ChangeAvailability** – This route allows a dealer to change the availability state of her car ads. Private route.
- **DELETE /CarAds/{id}** – This route allows a dealer to delete one of her car ads. Private route.
- **POST /Identity/Login** – Returns the user's dealer ID beside the token.
- **PUT /Identity/ChangePassword** – This route allows the currently authenticated user to change her password. Private route.
- **Implement the missing validation to the already implemented commands**.
- **Add unit and integration tests as you see fit.**

Hints:

- You may throw a **NotFoundException** in your queries in the cases where the provided ID is not found in the database.
- Editing domain entities should be done through exposed methods. Do not use setters as they should remain private.

Client Application

You may test your solution with the provided **Angular** application in the **Client** folder. Here is how to start it:

- Make sure you have the latest LTS **NodeJS** installed.
- Open a terminal and run *npm install -g @angular/cli.*
- Then navigate the terminal to the **Client** folder and run *npm install.*
- Finally, run *ng serve* and the client should be available at *localhost:4200*.

*Note: the server application should be running on the default self-hosted **.NET Core** port – 5001.*

Bonus Requirements

In this section, we will add additional architecture considerations, which may be suitable depending on the use case and business requirements:

- Add unit tests to the **Application** project. Assert configurations, commands, and queries.
- Add integration tests to the **Infrastructure** project. Assert all configurations and services.
- Add **CORS** to the system.
- Make **Scrutor** automatically register all services and initializers conventionally. For example, the **IIdentity** interface should be mapped to **IdentityService** and the **ICurrentUser** – to **CurrentUserService**.
- Add **MediatR** behaviors for request logging and performance tracking.
- Introduce database indexes. The **IsAvailable** column in the **CarAds** table is a perfect candidate. You filter by it on every search. Additionally, add unique indexes on the category, and manufacturer names.
- Introduce an internal **IRawQueries** interface in the **Persistence** infrastructure. Use **Dapper** by its implementation and write one of the commands with a raw *SQL* query.
- Add roles to the **Identity** system. Seed an administrator user to the database and give her the ability to change every piece of data in the application.
- Add a base auditable entity, which should store creating and editing information – user and date. Introduce soft delete and store who and when deleted the entity. Use the **SaveChanges** method to update the audit data.
- Separate the repositories – domain repositories and query repositories. The domain ones should live in the **Domain** project. The query ones should perform only queries mapped to an output model and should be placed in the **Application** project.
- Introduce a repository with a memory cache, which should serve as a proxy to the original one. Cache the car ad searching without any query parameters.
- Introduce a **GET /Statistics** action to return the total number of dealers and cars in the system. Introduce a response cache to the **Web** layer to improve the route's performance.

- Extract two bounded contexts – **Identity** and **Dealers**. Separate the **DbContext** by using two interfaces, and do not overlap the logic in all layers above the database. Remove the **Dealer** navigational property but keep the **User-Dealer** relationship in the database. It should be broken only if microservices are going to be extracted.
- Extract a bounded context for the statistics and add a counter for car ad views, and car ad searches. Use domain events to update the data on every created car and registered dealer.