***MyShop, a state-of-the-art webshop***

In this exercise we'll use what we've learned in the presentation to create a state-of-the-art webshop from scratch (at least a small part of it). We also add an extra bonus and divide the solution into different projects that each handle a different layer of our application. We also use inheritance to create domain-specific repositories. So this exercise will not be 100% the same as what we have seen in presentation. But the philosophy remains of course: we use a repository to decouple the application from the way data is stored in the database.

***1. Solution Setup***

No time to waste, let's get started, open *Visual Studio* 2022 and create a new *ASP.NET Core Web App (Model-View-Controller)* ***MyShop.Web*** within a solution ***MyShop*** (be sure to respect the naming of project/solution):

Afbeelding met tekst

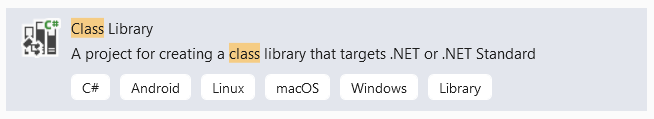
Automatisch gegenereerde beschrijving

Choose .NET *6.0*, and *Individual Accounts:*

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Next, add a second project to the same solution, a *Class Library*:



This project will contain our domain models, so we will call it ***MyShop.Domain*** (.NET 6.0):

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Add a third project with the name ***MyShop.Infrastructure*** to code the repositories:

Afbeelding met tekst

Automatisch gegenereerde beschrijving

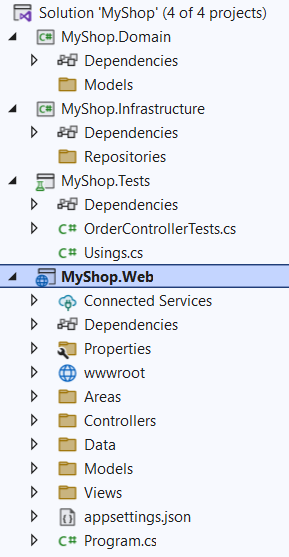
Finally add a fourth project ***MyShop.Tests*** to test our webshop:

Afbeelding met tekst

Automatisch gegenereerde beschrijving

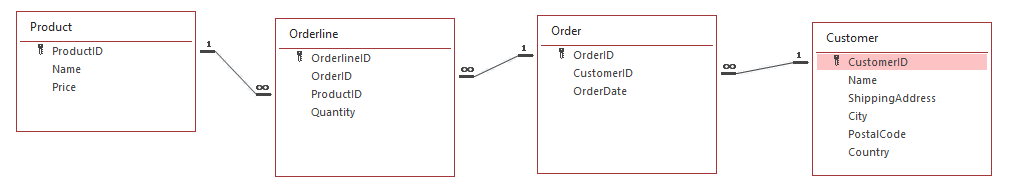
**For all projects, edit the project file and set the *nullable* property to *annotations*.**

We're almost done with the first step. Delete the class *Class1.cs* in the *Domain* and *Infrastructure* projects. Add a folder ***Models*** to the *Domain* project and a folder ***Repositories*** to the *Infrastructure* project. Rename *UnitTest1.cs* in ***OrderControllerTests.cs****.* The solution now looks like this:



***2. Models***

Next create the domain models in the *Models* folder of the *Domain* project. You do not need to do this yourself. You can copy them from the resources. For our webshop, we have customers and products. These customers will place orders, which contain multiple orderlines.



All ***ID***'s are *ints*. ***Price*** is a *decimal*, ***Quantity*** is an *int* and ***OrderDate*** is a *DateTime*. All the other fields are *strings*. Do not forget the navigation properties. Also add an extra calculated property ***OrderTotal*** to the ***Order*** model in which the total order amount is automatically calculated. This can be done using the following lambda expression:

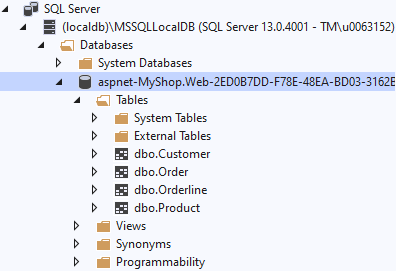
public decimal OrderTotal => Orderlines.Sum(item => item.Product.Price \* item.Quantity);

***3. Database***

Add the *ShoppingContext.cs* and *DBInitializer.cs* classes (from the *resources*) to the root of the *MyShop.Infrastructure* project. Since you are using the models from the *Domain* project, do not forget to add a reference to this project! And because we are using *Entity Framework* to communicate with our database, add the packages *Microsoft.EntityFrameworkCore.SqlServer* and *Microsoft.EntityFrameworkCore.Tools* to the *Infrastructure* project. This can be done using the *NuGet Packages Manager*. Use version *6.0.9!*

Now modify *Program.cs* of the webshop: add the *ShoppingContext* as a service and call the *Initialize* method of the *DBInitializer* to create the database. Do not forget to add the necessary references in the *Web* project!

Start your webshop application and check if the database is created with the correct tables/data:



***4. Webshop***

We will not code a complete webshop of course, this would lead us too far. We only code a page which lists all the customers and orders and allows the user to add new orders. And you don't even have to write this programme code yourself.

Add the *CustomerController.cs* en *OrderController.cs* to the *Controllers* folder of the webshop. Add the corresponding view folders (*Customer* and *Order*) to the *Views* folder. Overwrite the layout template *\_Layout.cshtml* in the *Shared* folder.

For a better user experience *Vue.js* (a JavaScript framework) is used to create the new orders. *Vue.js* is beyond the scope of this course, but if you are interested, you can always take a look at the *Create* view of *Order*.

In short: all data (order, customer, lineitems) that is entered by the user ends up in a *CreateOrderModel* object. This object will be passed to the *Create* method of our *OrderController* to insert the data into our database. To make sure all of this works correctly, add the models *CreateOrderModel.cs*, *CustomerModel.cs* and *LineItemModel.cs* to the *Models* folder of your webshop.

By now, you should have a working webshop application:

Afbeelding met tekst

Automatisch gegenereerde beschrijving

in which you can retrieve all orders:

Afbeelding met tekst

Automatisch gegenereerde beschrijving

list all customers:

Afbeelding met tekst

Automatisch gegenereerde beschrijving

and add a new order to the database (using *Vue.js*):

Afbeelding met tekst

Automatisch gegenereerde beschrijving

***5. Repositories***

Now apply what we've learned in the lesson to make our code reusable, maintainable and above all testable.

First in the *Repositories* folder, create an interface *IRepository.cs* with the following methods:

- *Add* gets an object as parameter and returns the inserted object

- *Update* gets an object as parameter and returns the updated object

- *Get* gets an int as parameter and returns the corresponding object

- *All* has no parameters and returns all objects

- *SaveChanges* has no parameters nor return value

Next, create an ***abstract*** class *GenericRepository* (in the *Repositories* folder) that implements this interface. We use an *abstract class* to indicate that the class is intended only to be a base class of other classes, not instantiated on its own. Mark all methods (except *SaveChanges*) as ***virtual*** to indicate they can be overridden in the derived classes.

Now we can inherit our *GenericRepository* base class to the repositories for our domain classes. Since we use inheritance, the methods will be automatically present in our derived repositories. And since the methods are virtual, they can be overridden if a different implementation is necessary.

Inherit the generic repository into three repositories (Order, Customer and Product). *Orderlines* does not have its own repository because orderlines is used as part of order.

In the constructor, call the constructor of the base class. Override the *Update* method in the derived classes to indicate what properties of the entitiy can be updated. Below you can find the *ProductRepository* as an example (write the code of the *OrderRepository* and *CustomerRepository* yourself). For an *Order*, the *OrderDate* and *Orderlines* can be update. And for a *Customer* the updatable fields are *Name*, *City*, *PostalCode*, *ShippingAddress* and *Country*.

using MyShop.Domain.Models;

using System.Linq;

namespace MyShop.Infrastructure.Repositories

{

public class ProductRepository : GenericRepository<Product>

{

public ProductRepository(ShoppingContext context) : base(context)

{

}

public override Product Update(Product entity)

{

var product = \_context.Products

.Single(p => p.ProductID == entity.ProductID);

product.Price = entity.Price;

product.Name = entity.Name;

return base.Update(product);

}

}

}

In the *OrderRepository*, overwrite the *All* method as well and use eager loading to retrieve the *Orderlines* and *Product* information.

***6. Controllers***

We'll use dependency injection to inject our repositories into our controllers. Register the three repositories in *Program.cs*. Use you *depency injection* knowledge!

Inject the repositories in the *CustomerController* en *OrderController*. You don't have to inject the *ShoppingContext* any more. As a final step, use the methods of the repositories in the controllers and check if your application still works after refactoring.