***MyShop, a state-of-the-art webshop, Part 2***

In this *2nd MyShop exercise* we'll use what we've learned in the presentation about the *Unit Of Work* design pattern. The start of this exercise will be the final version of the exercise of *Part 1*. In this version of our webshop, it is possible to retrieve all customers and orders. And we can also add new orders to our database.

***1. A Use Case for Unit Of Work***

As stated in the presentation, you don't have to use a design pattern for the pattern itself. There must be a valid reason, such as a functionality that cannot be programmed without the design pattern in question. So let us first try to think of a situation where we would get stuck in our webshop without the *Unit Of Work* design pattern.

Open the *Part 1 solution* of *MyShop* and modify the *Initialize* method of your *DBInitializer* class. Make sure a previously existing database gets deleted and re-created every time you start your project:

**context.Database.EnsureDeleted();**

context.Database.EnsureCreated();

We do this for demonstration purposes. When you start your project, there always will be the initial one order and two customers in your database.

Now go to the *Create Order* page and try to add **two orders** for the same customer with name *Emma Davis (2x)*:

Afbeelding met tekst

Automatisch gegenereerde beschrijving

In the perfect scenario, there now would be two orders added to the database and one customer (*Emma Davis*). But as you can see in the customer overview, the customer was also added twice, which is obviously not what we want.

Afbeelding met tekst

Automatisch gegenereerde beschrijving

So let's try to fix this. The strategy is this: when adding an order, we first check whether the customer already exists in the database. If this is the case, we overwrite his address data with the newly entered address data. Otherwise, we add the new customer. Then we create the order with the existing or new customer. **And all of this has to be done in one transaction!** We do not want that, due to an system error, the customer is added/modified, but the order is not created.

One more thing: for the sake of simplicity, to check whether the customer already exists, we use his name. So we assume that all our customers have unique names. Which, of course, in reality is not the case. No time to waste, let's start with the implementation.

***2. Extend the Repository***

First add an extra method *Find* to your *IRepository* interface. This method has a *filter* parameter (a LINQ expression) and returns all objects that meet that filter. We'll use the *Find* method later to check whether there's already a customer with the same name in our database.

Next implement the *Find* method (as specified in the interface) in your *GenericRepository*. Since your abstract class now implements the interface, your solution should compile again.

***3. Unit Of Work***

In the root folder of the *Infrastructure* project, create an *interface* *IUnitOfWork.cs* with a getter for the three repositories of your project and the *SaveChanges* method. Implement the interface in a *UnitOfWork.cs* class as explained in the presentation:

- make sure to inject the *ShoppingContext*

- let each getter return a repository instance that has been instantiated using the same *ShoppingContext* instance as the other repository instances

- also implement the *SaveChanges* method using the same *ShoppingContext* instance

Add the *IUnitOfWork* interface to the services of your *WebShop* so that it can be used using *Dependency Injection*.

***4. Modify the OrderController***

Now modify the *OrderController*. Instead of injecting the repositories, inject the *IUnitOfWork* interface and alter all methods to use the injected *UnitOfWork* class as well.

Now comes the hardest part. Modify the *Create* (*POST*) method of your *OrderController* to implement our *Use Case* (*Two Orders for the same Customer)*:

- check if the customer name already exists in the database

- if this is the case, change the *ShippingAddress*, *PostalCode*, *City* and *Country* of the existing customer with the new data. **Update** the customer data.

- otherwise, create a new customer

- create the order (with the new or modified customer) and **commit all changes** using the *UnitOfWork* class

Now try out whether what you have programmed actually works. Add again two orders for the same customer *Emma Davis*. Remember that the database is re-created on start-up. Check if your customer is only created once!

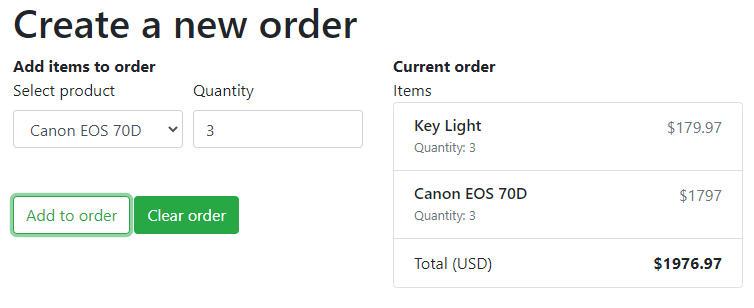
***5. Some small improvements***

- Modify your webshop and show the name of the customer who placed the order in the order list:

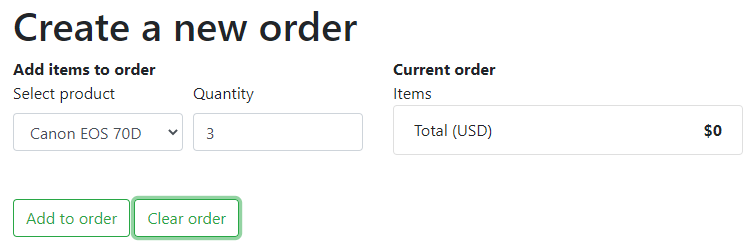
Afbeelding met tekst

Automatisch gegenereerde beschrijving

- Add an extra button *Clear order* to remove all lineitems in the current order. This button is only activated if your current order contains lineitems. You will need some *JavaScript* and *Vue.js* knowledge for this one. Do your best!



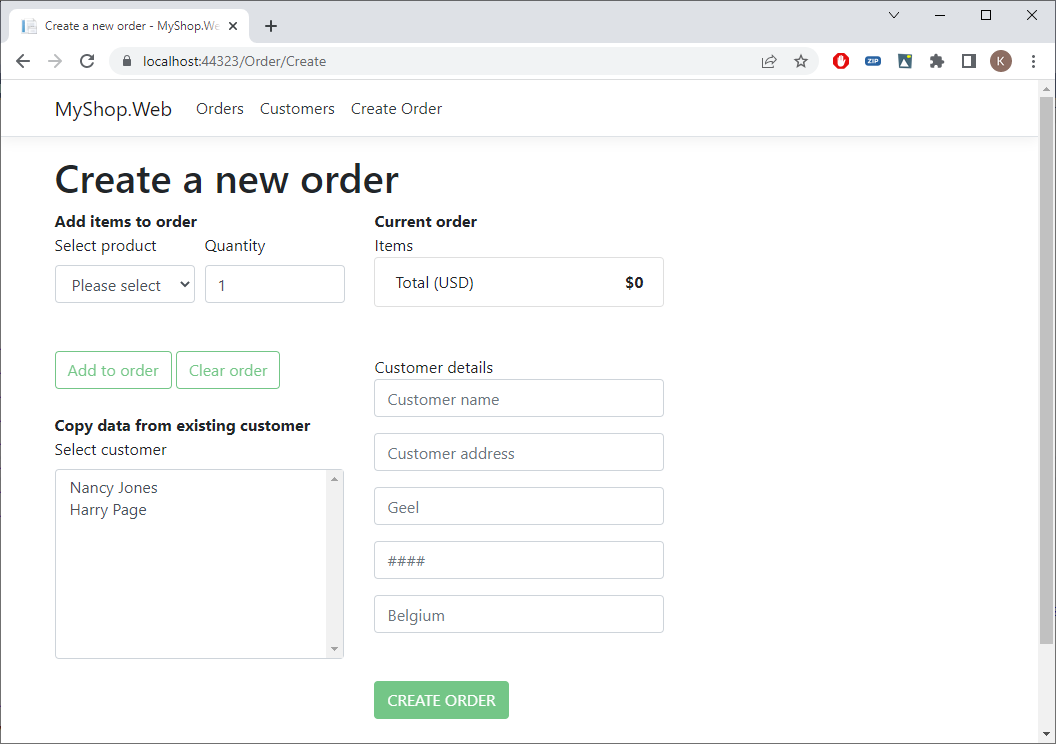
And after clicking *Clear order*:



***6. Better UX***

Let's make our end user's life a little easier. Instead of having to enter the address data of an existing customer each time again, we show a list with existing customers and give the user the possibility to copy the address data into the text boxes.

First show a select list with existing customers in the *Create Order* page:

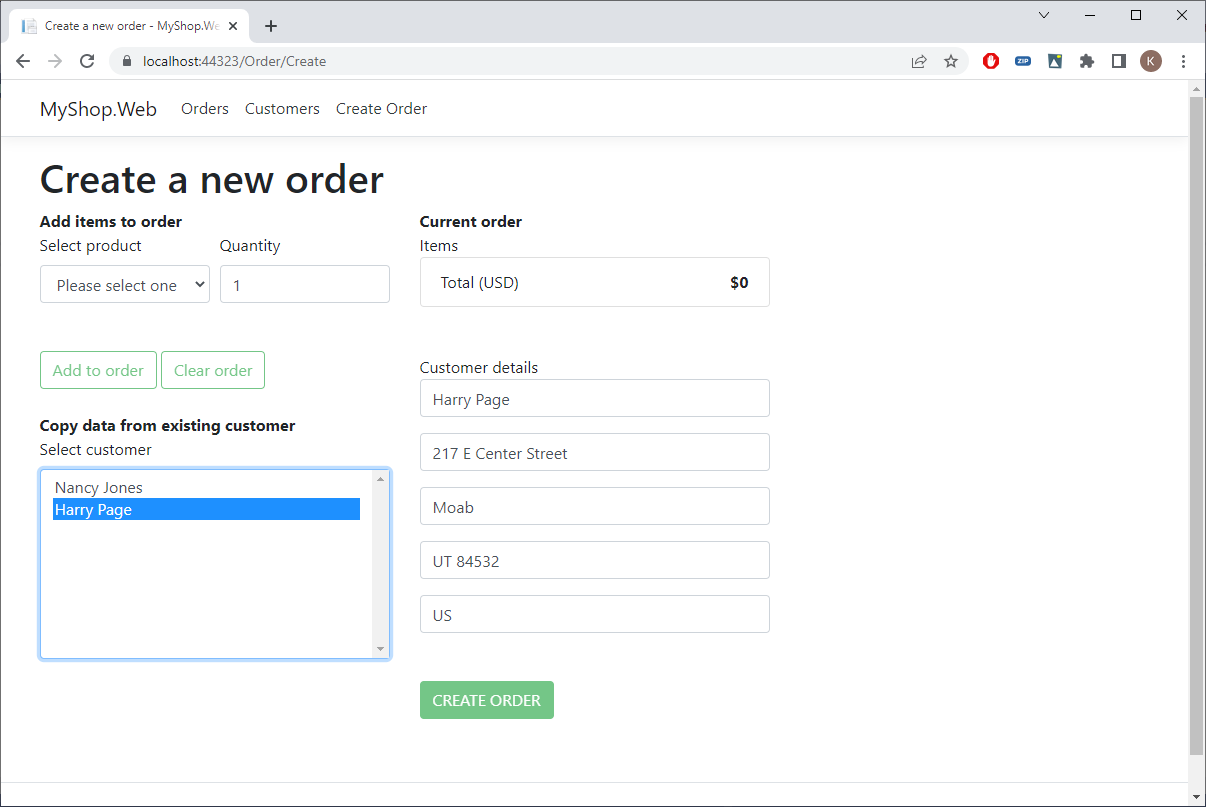


Tip:

- Use a *ViewModel* with products and customers.

- Have a look at the code for the *select* with products and do the same for the customers. Use *size="8"*.

Next, after selecting an existing customer, copy the customer name and all address data to the text boxes. So after clicking *Harry Page*:



Tip:

In *Vue.js*, to execute a method when an item is clicked in a *select*, you use

<select ... v-on:change="customerSelected">