

Software Development Assignment

Introduction

For this assignment , I created a webAPI based on the Onion Architecture.

This allows for **Appropriate separation of concerns** and vastly helps **code testability** as it makes it possible for layers to be abstracted and tested as a unit.

I ended up not implementing a database, but I adopted the **repository pattern** and have in-memory storage.

Throughout the project, I tried using meaningful names.

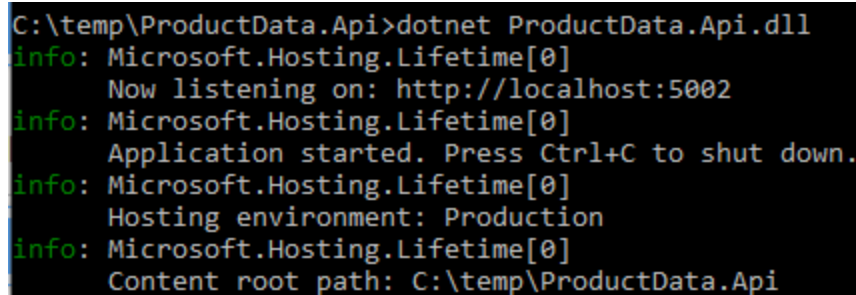
This document will be a guide to all my decisions and technical choices.

User Story:

At Coolblue, we want to be able to insure the products that we sell to a customer, so that we get money back in case the product gets lost or damaged before reaching the customers. For that, we need a REST API that is going to be used by Coolblue webshop. You're going to get access to our [Product Information API](#):

1. Unzip it onto your machine
2. Navigate to the unzipped folder in a terminal
3. On the terminal, type **dotnet ProductData.Api.dll** and hit *Enter*.

Your screen should look like as shown below if the API has successfully started:



```
C:\temp\ProductData.Api>dotnet ProductData.Api.dll
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5002
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\temp\ProductData.Api
```

The Products API is Swagger enabled, and you can access it by navigating to **http://localhost:5002/** in your browser. In the example the port is 5002.

Functionality already implemented [here](#):

There is an existing endpoint that, given the information about the product, calculates the total cost of insurance according to the rules below:

- If the product sales price is less than € 500, no insurance required
- If the product sales price=> € 500 but < € 2000, insurance cost is € 1000
- If the product sales price=> € 2000, insurance cost is €2000
- If the type of the product is a *smartphone* or a *laptop*, add € 500 more to the insurance cost.

** Download the current code, so you can continue with the further tasks.*

Task 1 [BUGFIX]:

The financial manager reported that when customers buy a laptop that costs less than € 500, insurance is not calculated, while it should be € 500.

Solution:

There was an error with the verification.

```
if (toInsure.SalesPrice < 500)
    toInsure.InsuranceValue = 0;
```

The if clause that calculated the price was assigning 0 to the *InsuranceValue* property if the *SalesPrice* property was less than 500.

Despite it having a functioning nested if condition that would check for the product type name and add €500, it was never reached as it belonged to the else condition.

Task 2 [REFACTORING]:

It looks like the already implemented functionality has some quality issues. Refactor that code, but be sure to maintain the same behavior.

Solution:

- Separation of concerns. Creating Controllers, Services and Model Objects in order for things to be structured and respect SOLID principles.
- Creating a static class as a helper to compare *ProductTypeName* to be less error prone.
- Refactor the *InsuranceValue* property from nested if conditions to a switch expression that improves readability and performance.
- Stop passing variables by reference, as it can be a major problem to refactor and maintain the API.
- Reading the external API's through *appsettings.json*
- Initial controller was conducting 2 outgoing calls to fetch the same data.Optimized outgoing calls.

Task 3 [FEATURE 1]:

Now we want to calculate the insurance cost for an *order* and for this, we are going to provide all the products that are in a shopping cart.

Assumptions:

An **Order** consists of an ID and a list of products that need to be insured.

I assume you still want to make calls to the external API, so there is no model regarding Product or ProductType because of this.

Solution:

For this feature, an endpoint is made available.

```
[HttpPost]
[Route("orders")]
2 references
public async Task<ActionResult<OrderDto>> CalculateInsurance([FromBody] OrderDto toInsure)
{
```

Note : This feature implements task 5 surcharge rate.

Task 4 [FEATURE 2]:

We want to change the logic around the insurance calculation. We received a report from our business analysts that digital cameras are getting lost more than usual. Therefore, if an order has one or more digital cameras, add € 500 to the insured value of the order.

Solution:

Create **DslCheck** method.(In hindsight , this should have been a method from the Order class).

```
//Verifies if any product of a product list is a Digital Camera
1 reference
private bool DslCheck(List<InsuredProduct> insuredProducts)
{
    var hasDSL = insuredProducts.Exists(item => item.ProductTypeName == ProductTypeName.DigitalCameras.ToString());
    return hasDSL;
}
```

If this evaluates to true, then add € 500 to the insurance value of the order.

Task 5 [FEATURE 3]:

As a part of this story we need to provide the administrators/back office staff with a new endpoint that will allow them to upload surcharge rates per product type. This surcharge will then need to be added to the overall insurance value for the product type.

Solution:

For this feature, a new endpoint is made available.

```
[HttpPost]
[Route("surcharge/{productId:int:min(0)}/{surcharge:float}")]
6 references | 7/7 passing
public async Task<ActionResult<SurchargeDto>> UploadSurcharge(int productId, float surcharge)
{
```

- As requested, this feature uploads a surcharge.
- After uploading, it also updates the value of the insurance of all insured products that are stored.
- This method will update the rate of an already created surcharge rate, if it is used with the same product type id as one already in existence.

Final Thoughts:

In hindsight, I could've implemented a db and an ORM.

The model could have been refined with usage of entities and value objects.

This would mean more verifications and strict rules and a less prone to mistake API.

Could've created custom exceptions and threw those to better log the errors within the flow of some processes.

Thanks for reading this document !

João Soares