Contents

LIST OF FIGURES

COOL BLUE WEB SHOP DECISIONS AND ASSUMPTIONS

This document details out the assumptions and decisions I made in the re-design and extension of the proposed Web Shop API.

Architecture Decision

A simple layered architecture, consisting of:

1. Models
2. Repositories
3. Services
4. Controllers

This is to ensure appropriate separation of concerns. Code would be easily maintainable.

Also, unit tests can be created separately without any effect on other modules of the application.

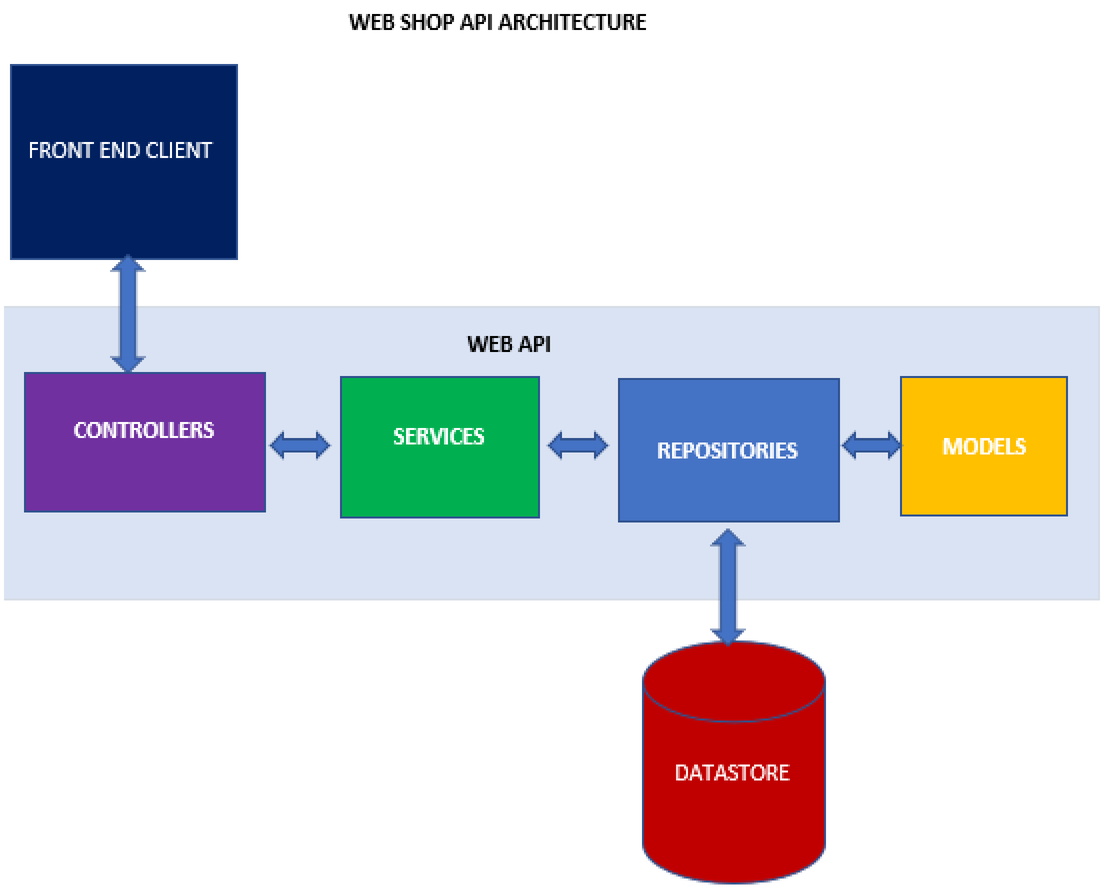

Figure 1 Showing Architectural Model for Web Shop API

In this simple layered architecture, the assumption is that the frontend client makes a request to the Controller which holds the endpoints. The Endpoint then calls the Services which talks to the Repositories. The Repositories then make a direct communication with the assumed datastore.

Models

The Models represent the data objects and their properties that data would be mapped to

Here I include Dto objects as Dto would represent what presents data to the user interface.

Repositories

The Repositories handle all of CRUD operations that need to be done. I have created and implemented interfaces in their respective classes here.

Services

In the services I have included all the business logic. This allows for me to define logic separate from any object definitions while adhering to SOLID principles.

Controllers

The controllers contain the actual API that will consumed by any request from any application.

Task 1[Bug Fix]

I noticed that the condition for charging insurance on a laptop sale was not met.

I had to ensure that whenever the product is a laptop the insurance value of 500 should be charged whether its sales price was less than 500 or not.

Task 2 [Refactoring]

I changed the position of the condition that handled when the salesprice is less than 500 in order to make the criteria.

I also removed some variable names (`productTypeName`, hasInsurance) that were not being used anywhere in the code to prevent the application from creating unnecessary and unused memory locations at runtime.

I changed the name of the method to CalculateInsuranceCharge from GetProductType. Calculate InsuranceCharge suggests more of what the method does than GetProductType.

I also changed the name of the controller to InsuranceController to match the fact we are creating a behaviour that calculates insurance charges.

I calculate the insurance value of the product by first checking if its insurable.

If the product is not insurable, its insurancevalue is 0.

Task 3[Feature 1]

I created an endpoint called `GetOrderInsurance` . The endpoint receives a payload of an order which has a list of products.

A product may be insurable or not. If the product is not insurable, its insurancevalue is 0

I sum up the insurance values of all the products in the order to get the orderinsurancevalue.

The endpoint now returns the orderinsurancevalue with the order details.

Task 4[Feature 2]

In my method that calculates the insurance on an order, I add a check to see if the order contains a digital camera product.

Once I find one or more digital cameras in the order, I add an additional **500 Euros** to the order insurance value.

The order insurance value will now be the sum of all the insurance values on each product and the additional 500Euros (as digital camera insurance)

If I do not find a digital camera in the order the orderinsurancevalue is calculated as usual as the sum of each products insurancevalue.

Task 5[Feature 3]

I created an endpoint called `UploadSurchargeRates`. This endpoint receives input for surcharge rates added on each product type and saves.

In my method that calculates insurance charge, I add a check for the surcharge on the product type.

If the product type has no surcharge rate then the surcharge on the product is 0.

The surcharge rate value is then added to the insurance value of the product and is presented with the insurance value as the **insurancevalue** of the particular product.

This Is done by a method called GetSurcharge

Logging

I chose to use nLog. nLog is a .Net Library that allows me to do add logs of my API requests for reference.

For every API request, there are several checks and decisions and responses. I capture these events in the logs and save them to a file

Unit Testing

I created a class to do Insurance Service unit test.

I employed the use of the Moq Library in my unit test to simulate my data.

The Moq library allows me to manipulate the mock object in many ways. It allows me to set mock methods to return specific values and to match the specific arguments when the test method is called.