BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

SPECIALIZATION COMPUTER SCIENCE

DIPLOMA THESIS

ASP.NET Web Application

Supervisor

Assoc. Prof. Dr. Andreea Vescan

Author

Purdea Bogdan-Florin

**2018**

**Contents**

[**Chapter 1 - Introduction** 3](#_Toc522750117)

[**1.1. Scope, motivation, importance** 3](#_Toc522750118)

[**1.2. Structure of the thesis for each chapter** 3](#_Toc522750119)

[**Chapter 2 - Software development life cycle** 4](#_Toc522750120)

[**2.1. ASP.NET Application Life Cycle** 4](#_Toc522750121)

[**2.1.1. Creation of ASP.NET Environment** 4](#_Toc522750122)

[**2.1.2. Life Cycle Events** 6](#_Toc522750123)

[**Chapter 3 - Technologies used in the development of the application** 8](#_Toc522750124)

[**3.1. Visual Studio 2017** 8](#_Toc522750125)

[**3.2. Entity Framework Core** 8](#_Toc522750126)

[**3.3. ASP.NET Core MVC** 8](#_Toc522750127)

[**Chapter 4 - Store Web Application** 10](#_Toc522750128)

[**4.1. Requirements** 10](#_Toc522750129)

[**4.1.1. Specifications** 10](#_Toc522750130)

[**4.1.2. Characteristics of entities** 10](#_Toc522750131)

[**4.1.3. Use cases** 10](#_Toc522750132)

[**4.2. Design** 13](#_Toc522750133)

[**4.3. Implementation** 13](#_Toc522750134)

[**4.3.1. The data access layer (DAL) implementation** 13](#_Toc522750135)

[**4.3.2. The RESTful service implementation** 19](#_Toc522750136)

[**4.3.3. The user interface (UI) implementation** 22](#_Toc522750137)

[**4.4. Testing** 28](#_Toc522750138)

[**4.5. Deployment** 28](#_Toc522750139)

[**Chapter 5 - Conclusions** 29](#_Toc522750140)

[**Bibliography** 30](#_Toc522750141)

# **Chapter 1 - Introduction**

## **1.1. Scope, motivation, importance**

## **1.2. Structure of the thesis for each chapter**

The first chapter makes a brief introduction in the contents of the thesis.

The second chapter is the theoretical chapter describing the life cycle of an ASP.NET application from the moment a user makes a request until the moment the request is rendered on the browser.

The third chapter presents all the technologies used in developing the application: Visual Studio 2017, Entity Framework Core and ASP.NET Core MVC.

The fourth chapter describes the requirements of the application, as well as the design, implementation, testing and deployment of the application.

# **Chapter 2 - Software development life cycle**

## **2.1. ASP.NET Application Life Cycle**

### **2.1.1. Creation of ASP.NET Environment**

An ASP.NET Application starts its life cycle when a user makes a request to the server. The request is sent by the browser to the web server, Internet Infomation Services (IIS) by default. IIS sends the request to the Internet Server Application Programming Interface (ISAPI) which checks the extension of the file requested by the browser and depending on the file extension the request is processed. For instance, if the page is an ‘.ASPX’ file, then it will be passed to ‘aspnet\_isapi.dll’ for processing.

The Internet Server Application Programming Interface (ISAPI) is collection of Windows-based web server services which provides improvements to the functionalities provided by IIS.

For the first request to the website for any resource in an application, an instance of the ApplicationManager class is created, which is the application domain that request is processed in. The application domain provides isolation between applications hosted on the same server and enable each application to be unloaded separately. In the application domain is created an instance of the HostingEnvironment class, which offers access to information about the application, like the name of the folder that stores the application.

In Figure 2.1. this stage is presented.

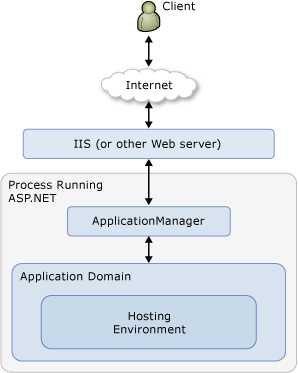


Figure 2.1. Diagram illustrating what happens when ASP.NET receives the first request for the application

After the creation of the application domain and the instantiation of the HostingEnvironment object, the necessary core ASP.NET application objects such as HttpContext, HttpRequest and HttpResponse are created and initialized.

The HttpContext class contains objects specific to the current application request, such as the HttpRequest and HttpResponse objects. The HttpRequest object constains information about the current request, which includes cookies and browser information. The HttpResponse object constains the response that is sent to the client, which includes all the rendered output and cookies.

Once all the core ASP.NET objects are initialized, the application is started by creating an instance of the HttpApplication class that is assigned to the request. In case the application has a Global.asax file, ASP.NET creates an instance of the Global.asax class instead that is derived from the HttpApplication class.

Figure 2.2. illustrates this stage.

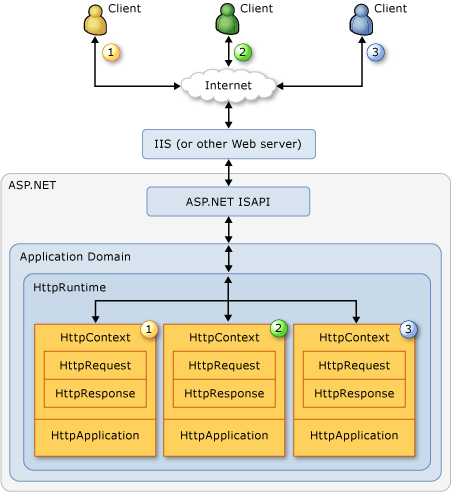


Figure 2.2. Diagram illustrating what happens when an HttpApplication object is assigned to the request

The request is then processed by the HttpApplication pipeline. While the request is processed HTTP module events, handlers and page events are executed by the HttpApplication class.

### **2.1.2. Life Cycle Events**

Once the instance of HttpApplication class is created it starts processing requests. The processing of a request goes through three different sections: HttpModule, Page and HttpHandler. From these sections the HttpApplication istance invokes different events that can be changed or extended by the developer.

HttpModule and HttpHandler helps inject custom logic before and after the processing of the ASP.NET page. HttpHandler is an extension based processor used to inject logic based in file extensions like ‘.ASPX’, ‘.HTML’. HttpModule is an event based processor used to inject logic in the events of the ASP.NET pipeline.

Figure 2.3. depicts how HttpHandler handles requests based on the extension name of the file.

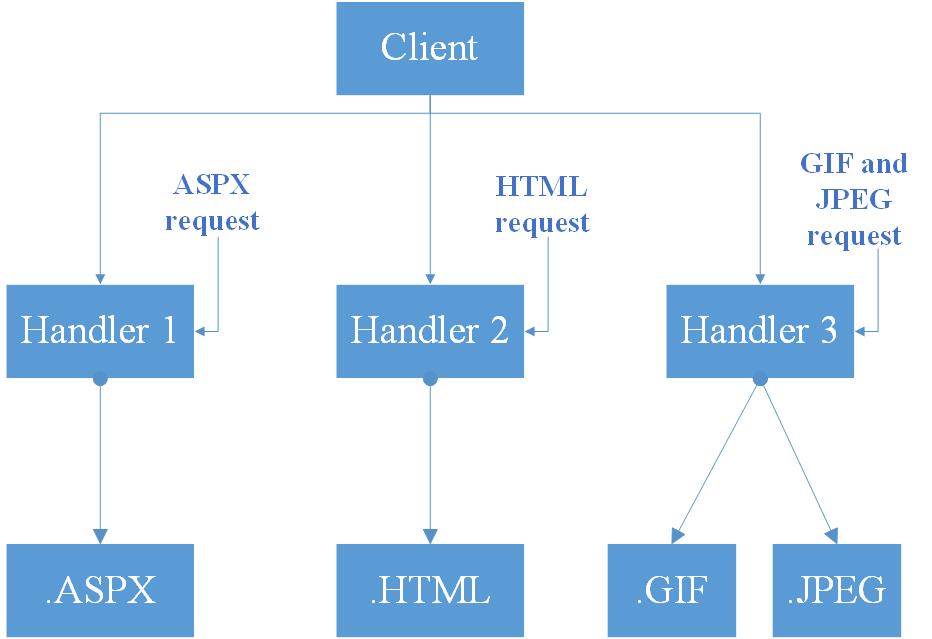


Figure 2.3. Diagram showing how HttpHandler works

Figure 2.4. depicts how the request pipeline invokes the events offered by HttpModule for injecting pre-processing and post-processing logic.

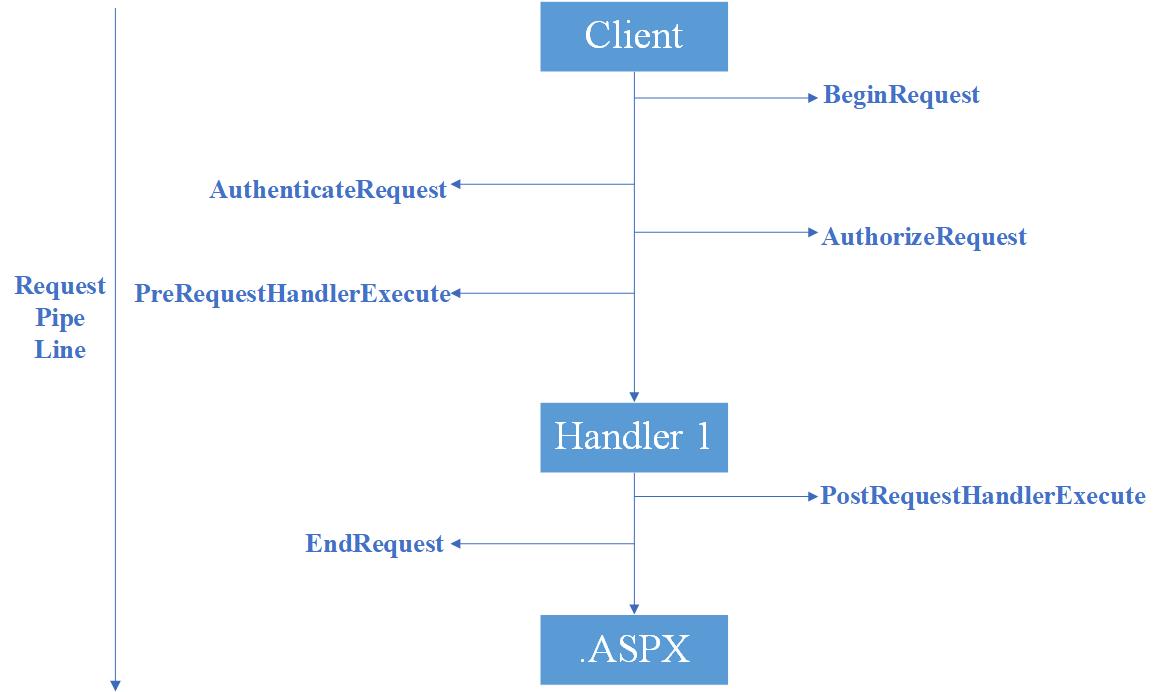


Figure 2.4. Diagram showing how the events from HttpModule are invoked by the request pipe line for a request

After the client request processing starts and before the ASP.NET page object is created, , pre-processing events provided by HttpModule are invoked through which the developer can inject custom pre-processing logic. The most important pre-processing events are BeginRequest, AuthenticateRequest, AuthorizeRequest, ResolveRequestCache, AcquireRequestState and PreRequestHandlerExecute.

Once those events are executed, the ProcessRequest event provided by HttpHandler is invoked which holds the HttpHandler logic. After the HttpHandler logic is executed the ASP.NET page object is created.

While the ASP.NET page object is created another events, which serve as placeholder for the logic inside the ASP.NET page, are invoked. The most important page events are Init, Load, Validate, Event, Render and Unload.

After the ASP.NET page object is executed and unloaded from memory, post-processing events provided by HttpModule are invoked which can be used to inject custom post-processing logic. The most important post-processing events are PostRequestHandlerExecute, ReleaseRequestState, UpdateRequestCache and EndRequest.

# **Chapter 3 - Technologies used in the development of the application**

The technologies used in developing the application are Visual Studio 2017, Entity Framework Core and ASP.NET Core MVC.

## **3.1. Visual Studio 2017**

## **3.2. Entity Framework Core**

“Entity Framework (EF) Core is a lightweight, extensible, and cross-platform version of the popular entity framework data access technology”. [3]

“EF Core can serve as an object-relational mapper (O/RM), enabling .NET developers to work with a database using .NET objects, and eliminating the need for most of the data-access code they usually need to write”. [3]

Data access is peformed using a model made up of entity classes and a derived context that represents a session with the database, allowing querying and saving data. This model can be either generated from an existing database, hand coded to match the database or used to create the database through EF Migration.

## **3.3. ASP.NET Core MVC**

“ASP.NET Core MVC is a rich framework for building web apps and APIs using the Model-View-Controller design pattern”. [7]

“The ASP.NET Core MVC framework is a lightweight, open source, highly testable presentation framework optimized for use with ASP.NET Core. ASP.NET Core MVC provides a patterns-based way to build dynamic websites that enable a clean separation of concerns”. [7]

The features of ASP.NET Core MVC are: routing, model binding, model validation, dependency injection, filters, areas, Web APIs, testability, razor view engine, strongly typed views, tag helpers and view components.

ASP.NET Core MVC routing is based on ASP.NET Core’s routing, which is a powerful URL-mapping component that offer comprehensible and searchable URLs.

Convetion-based routing enables global definitions and mappings to specific action methods on given controllers of the URL formats accepted by the application. “When an incoming request is received, the routing engine parses the URL and matches it to one of the defined URL formats, and the calls the associated controller’s action method”. [7]

Attribute routing enables defining application routes next to the controller and action to which they are associated.

Model binding is the attempt of the Core MVC to reconstitute the specified type from name/value pairs made available using reflection. It makes available a ModelState object that contains error information for every property that fails to bind. The binding engine sets ModelState.IsValid to false if one or more properties can’t be assigned, otherwise it sets ModelState.IsValid to true.

The model validation is done by using data annotation validation attributes on model objects properties. Those attributes are checked on the client side before the values are posted on the server and on the server before the controller action method is called.

In ASP.NET Core MVC controllers can request services through their constructors and views can use the @inject directive. Those features are made available by the built-in support for dependency injection of ASP.NET Core.

Core MVC allows the creation of filters that can be configured to run at certain point within the execution pipeline, before or after an action method for example, or in the events of unhandled exceptions. Filters can run globally or can be applied to controllers or action methods as attributes. For example, the Authorize filter that is included in the Core MVC framework:



“ASP.NET Core MVC views use the Razor view engine to render views. Razor is a compact, expressive and fluid template markup language for defining views using embedded C# code”. [7]

“Tag helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. Tag Helpers bind to specific elements based on the element name and its attributes. They provide the benefits of server-side rendering while still preserving an HTML editing experience”. [7]

Some of the built-in tag helpers, designed to be used instead of their respective HTML helpers, are: the form tag helper, the anchor tag helper and the input tag helper. Those tag helpers can be seen in the next example:



View components allow packing rendering logic so that it can be reused throughout the application.

In addition to being a great platform for building web sites, ASP.NET Core MVC has great support for building Web APIs.

# **Chapter 4 - Store Web Application**

## **4.1. Requirements**

### **4.1.1. Specifications**

The web application is destined to store administrators that have to manage the products of the store and to the possible customers of the store that are interested in one or more of the products made available by the store.

There are two types of users: administrator and customer. A user has to register if he has no account, otherwise login the account in order to access the functionalities of the application.

The store administrator has the option to see a record of the products available in the store app and add, update or remove them.

The application offers to the possible customers the option to add to a shopping cart one or more products that can later be ordered and the option to see the history of their orders.

### **4.1.2. Characteristics of entities**

* User: name, email, password, role
* Category: name
* Product: description, model name, model number, is featured, product image, product image large, product image thumb, unit cost, current price, units in stock, category id
* Customer: full name, email address, user id
* Order: order date, ship date, customer id, order total, billing address, shipping address, customer phone
* Order detail: order id, product id, quantity, unit cost, line item total

### **4.1.3. Use cases**

The main actors of the application are the administrator of the store and the customer of the store.

Here are some use cases for the application:

1. Name: View products

Actor: Anonymous user

Description: When the web application is opened it displays a record of all the products in the store

Precondition: -

Postcondition: A record of all the products in the store is displayed on the user interface

Basic flow: 1.Open web application

2.The record is displayed on the user interface

Exceptions: -

1. Name: Add products

Actor: Administrator

Description: The administrator enters the characteristics of a product and clicks on the add button then the new product is memorized to the applications database

Precondition: The admin is logged in and the entered characteristics are valid and the product does not exist beforehand

Postcondition: The product is added in the database

Basic flow: 1.Enter characteristics for a product

2.Click on the add button

3.Create and memorize the product in the database

Exceptions: Invalid characteristics

1. Name: Delete product

Actor: Administrator

Description: The administrator selects a product from a list of all the products and clicks on the delete button then the selected product is removed from the applications database

Precondition: The admin is logged in

Postcondition: The product is deleted from the database

Basic flow: 1.A product is selected

2.Click on the delete button

3.Delete the product from the database

Exceptions: There was no product selected

1. Name: Make an order

Actor: Customer

Description: The client selects one of more products to be added to the shopping cart and when the Check out button is pressed an order is created.

Precondition: The customer is logged in and one of more products are in the shopping cart

Postcondition: The order is created and stored in the database

Basic flow: 1.Add one or more products to the shopping cart

2.Click on the check out button to make the order

3.Create and store the order with its information in the database

Exceptions: Empty shopping cart

1. Name: Update an order

Actor: Customer

Description: The client enters the information he wants to change for an order and the order is updated in the applications database

Precondition: The customer is logged in and the entered information is valid and the order to be updated exists

Postcondition: The order is updated in the database

Basic flow: 1.Enter the information to be changed for an order

2.Click on the button to update the order

3.Update the information of the order in the database

Exceptions: Invalid information given by the client

The order to be updated does not exist

1. Name: Add a product in the shopping cart

Actor: Customer

Description: The customer clicks the add to cart button for a product

Precondition: The customer is logged in and the product is not in the shopping cart

Postcondition: The product is added in the shopping cart

Basic flow: 1.Click the add to cart button for a product

2.The product is added to the shopping carts list of products

Exceptions: Product already in the shopping cart

## **4.2. Design**

The application is formed of three projects: one is the data access layer (DAL) and the other two are the RESTful service and the user interface (UI). The last two form the ASP.NET Core web application.

The data access layer (DAL) provides simplified access to the data used by the application stored in a relational database.

The RESTful service provides the functionalities and data operations of the application.

The user interface is the way through which the user interacts with the web application.

Both the RESTful service and the user interface use the Model-View-Controller (MVC) design pattern. For building the RESTful service only the Model and Controller parts are used from the MVC and for building the user interface the View part is brought into the development process alongside the Model and Controller parts.

The model holds the data representations of the objects and the view models which are composed of one of more models and shaped specifically for the view that uses them.

The view is the UI of the application. Views accept commands and render the results of those commands to the user.

The controller takes commands or requests from the user through the view, forward them to the appropriate service and then send the application data to the view.

## **4.3. Implementation**

### **4.3.1. The data access layer (DAL) implementation**

The data access layer is a console application and is implemented using Entity Framework Core.

Building the foundation of the data access layer is done by creating the context class (DbContext), creating the base entity class, creating and configuring the model classes, add the model to the context as a DbSet<T>, creating a migration and running it to update the database and adding a strongly typed repository for the model.

“The MSDN Library defines the DbContext instance as a combination of the Unit of Work and Repository patterns. It provides access to the model collections in your data model (populating the collections from the data store when needed), exposes CRUD functionality on your collections (in addition to the functions exposed through the DbSet<T> specialized collection classes), and groups changes into a single call as a unit of work”. [4]

“Each model collection is represented by a DbSet<T> specialized collection property in the DbContext class. Each DbSet<T> property contains all of the like entities (e.g., Category records) tracked in the context”. [4]

The context class inherits from DbContext. It has two constructors,it overrides the OnConfiguring method and the OnModelCreatingMethod from the DbContext class and it has a DbSet<T> property for each model. In Source code 4.1. is presented this database context class.



Source code 4.1. The database context class

The base entity class holds the properties that are common for all the models, in this case the primary key named Id and a Timestamp to which are applied some data annotations to help shape the database. In Source code 4.2. is presented this base entity class.



Source code 4.2. The base entity class

After that each model class inherits from this base entity class and additionally hold the properties specific to that model with their own data annotations. For each of those there is a Table attribute that specifies the name of the table and the database schema. In Source code 4.3. is the Category model class, the other models follow the same idea.



Source code 4.3. The Category model class

Entity Framework migrations provide a mechanism to update the physical data store to match the shape of the application data model when application models are changed.

“When a migration is created, EF examines model classes (and their data annotations) that can be reached from the context through the DbSet<T> properties, any related classes (and their data annotations), and any Fluent API commands (covered later in this chapter). From all of this information, EF constructs the current shape of the data model. EF then compares this shape with the existing database and creates the necessary change set to migrate the database to the current code-based model. This change set is saved in a migration file that is named using a timestamp and the friendly name provided by the developer. Executing the change set updates the database”. [4]

In the base repository interface are defined all the common methods to all the model repositories. The interface limits the type of T to the base entity class as shown in Source code 4.4.



Source code 4.4. The base repository interface

The base repository class implements the interface IDisposable and base repository interface, again limiting the type of T to the base entity class. The protected variable for the application context (the Db property) is instantiated in the constructors and disposed of in the Dispose method. The The DbSet<T> property (the Table property) is the gateway into and out of the specific T table in the database. This is presented in Source code 4.5.



Source code 4.5. The base repository class with its constructor methods and dispose method

Creating a new T record in the database through Entity Framework is a three-step process. First a new instance of the T class is created, and the properties set accordingly, with the Id and Timestamp properties being set by the SQL Server. After, the newly created instance is added to the appropriate DbSet<T> collection property of the application context or added directly to the application context. The final step is the SaveChanges() method call on the application context. The Add method in Source code 4.6. makes a call to the Add method on the Table property where the creating of a new record happens.



Source code 4.6. The Add method in the base repository class

Reading a record from the database means executing a LINQ statement against the appropriate DbSet<T> and using the data to create a new instance of the class T. The GetAll method in Source code 4.1.7. reads all the records of T type from the database.



Source code 4.7. The GetAll method in the base repository class

Updating a record is done by reading the record to be updated from the database and make the desired edits to it, call Update on the DbSet<T> or on the application context and call the SaveChanges() method on the application context. The Update method in Source code 4.8. makes a call to the Update method on the Table property where the updating of a record is done.

Source code 4.8. The Update method in the base repository class

Deleting a record is done by reading the record to be changed from the database, call Remove on the DbSet<T> or on the application context and call the SaveChanges() method on the application context. The Delete method in Source code 4.9. makes a call to the Delete method on the Table property where the deleting of a record is done.



Source code 4.9. The Delete method in the base repository class

The model repository interface inherits from the base repository interface and define the methods specific only to the appropriate model. In Source code 4.10. shows the Category repository interface, the other interfaces following the same pattern.



Source code 4.10. The Category repository interface

Every model repository inherits the base repository class and implements the specific model repository interface, have two constructors to match the base constructors and override the base methods GetAll() and GetRange() to sort by a specific property. In Source code 4.11. is the Category repository, the other repositories follow the same idea.



Source code 4.11. The Category repository class

And with this the data access layer implementation is concluded.

### **4.3.2. The RESTful service implementation**

The RESTful service is built using ASP.NET Core MVC as an MVC Web API application.

An ASP.NET Core application is a console application that creates a web server in its Main method. The Main method invokes WebHost.CreateDefaultBuilder, which creates a web application host following the builder pattern. The builder has methods that define the web server and the startup class. The UseStartup method on WebHostBuilder specifies the Startup class for the application. The Build and Run methods build the IWebHost object that hosts the application and begins listening for HTTP requests. In the Source code 4.12. is the default Main method created by the Visual Studio Web API template.



Source code 4.12. The main method of an ASP.NET Core application

The Startup class defines the HTTP request handling pipeline, initiates the configuration system and sets up the dependency injection container. In the constructor of the Startup class a new instance of the ConfigurationBuilder class is created and the resulting IConfigurationRoot is assigned to a public property of the class (Configuration in this case). The ConfigureServices method is used to configure all the services needed by the application and insert them into the dependecy injection contained. The Configure method is used to set up the application to respond to HTTP requests. The Startup class can be seen in Source Code 4.13.



Source code 4.13. The Startup class with its constructor and methods

“A *controller* is a class that inherits from the abstract Controller class or ends in the name Controller. Inheriting from the base Controller class is the most efficient way to develop a controller due to the helper methods contained in the base class. As mentioned earlier, convention dictates that MVC controllers are located in the Controllers directory of the project. When referring to controller in routes, the Controller suffix is dropped”. [4]

“*Actions* are simply methods on controllers. Actions can take parameters, and while they can return anything, they typically return an IActionResult (Task<IActionResult> for async operations) or a class that implements IActionResult”. [4]

Most of the controllers in the applications service provide read-only data using HTTP Get commands, but some of them contain in addition read-write actions using HTTP Post, HTTP Put and HTTP Delete. The most complex from all those controllers is the ShoppingCartController.

At the top of the class is the route attribute which defines the standard route with the literal api followed by the controller name, then extends the standard route with a customerId variable. An instance of the ShoppingCart repository interface is injected into the constructor and assigned to a class level variable. In Source code 4.14. is the ShoppingCartControllers route attribute and constructor.



Source code 4.14. The ShoppingCartController class from the RESTful service with its route and constructor

The GetShoppingCartRecord action method adds the variable place holder {productId} to the route and returns a single record from the cart for a customerId and productId. The GetShoppingCart action method has a route name assigned to it for future use and it gets all records for a customerId. Those action methods can be seen in Source Code 4.15.



Source code 4.15. The GetShoppingCartRecord and the GetShoppingCart action methods

The Create action method responds to HTTP Post requests. The customerId is passed through the route and the new cart record is sent in through the body of the request as JSON. Core MVC is instructed to use model binding by the [FromBody] attribute to create an instance of the ShoppingCartRecord class from the JSON contained in the request body. If a new ShoppingCartRecord is created succesfully from the JSON passed through the request message, the DateCreated is set to the current date, the CustomerId is set to the route value customerId, and the object is added to the database. The last line of the method creates the URL for the route named GetShoppingCart using the supplied values, adds the URL to the header of the HTTP response, and returns an HTTP 201 (Created). This action method is shown in Source Code 4.16.



Source code 4.16. The Create action method from the ShoppingCartController class

The Update action method responds to HTTP Put requests. It takes the shoppingCartRecordId as the final of the route value and the ShoppingCartRecord from the JSON contained in the body of the request message. The method first checks if the record exists and if the model binding is valid, then sets the DateCreated property of the ShoppingCartRecord to the current date and calls the Update method on the repository. Once complete, the URL of the shopping cart is added to the header of the response and a HTTP 201 is returned. The Source code 4.17. shows the Update action method.



Source code 4.17. The Update action method from the ShoppingCartController class

The Delete action method responds to HTTP Delete requests. The method adds the shoppingCartRecordId and the string version of the TimeStamp property to the route. After making sure that the timeStamp value contains quotes, as they are needed by the

JsonConvert.DeserializeObject method, the string is converted to a byte[] and is passed into the Delete method. If all is successful, a NoContentResult is returned, which is an HTTP 204. The Delete action method is shown in Source code 4.18.



Source code 4.18. The Delete action method from the ShoppingCartController class

The Purchase action method responds to HTTP Post requests and takes the literal buy at the end of the controller route. If all the values are valid, the Purchase method on the repository is called which returns the newOrderId, that is passed into the CreatedAtRoute method. The final line calls a route in a different controller (specifically in the OrdersController), passing in the customerId and the order Id. This action method can be seen in Source Code 4.19.



Source code 4.19. The Purchase action method from the ShoppingCartController class

All other service controllers behave more or less in the same manner as the ShoppingCartController. The action methods from the service controllers are called from the UI.

### **4.3.3. The user interface (UI) implementation**

The user interface is an MVC Web Application and is implemented similarly to the RESTful service, using ASP.NET Core MVC. The Main method is the same as the one in the RESTful service and does the same thing.

The Startup class for an MVC Web Application has the same role of configuration as the one in an MVC Web API.

The root URL for the RESTful service MVC Web API application has been added to the appsettings.json file under the name ServiceAddress in the section WebServiceLocator. To expose this value to the rest of the application a class is created that is hooked into the dependency injection framework, so it can be injected into any class that requires access to the URL and port information.

This class implements the interface from Source Code 4.20. where the service address property is defined.



Source code 4.20. The IWebServiceLocator interface

In the constructor of this class the value from appsettings.json is obtained and saved into the service address property. In Source code 4.21. is the definition of this class.



Source code 4.21. The WebServiceLocator class

In the IWebApiCalls interface are defined all the API call methods.

The WebApiCallsBase abstract class holds all the core methods to execute HTTP Get, Post, Put and Delete commands. The protected variables hold the URI for the web service and the core URIs for the service controllers. The constructor uses an instance of IWebServiceLocator to get the base URI of the web service through dependency injection and builds the controllers URIs. The WebApiCallsBase class properties and constructor are shown in Source code 4.22.



Source code 4.22. The WebApiCallsBase class with its constructor and properties

The GetJsonFromGetResponseAsync method executes the calls to the service and returns the contents of the response message as JSON. The GetItemAsync and GetItemListAsync use the previous method to return a single item or a list of items respectively. Those methods are shown in Source code 4.23.



Source code 4.23. The read requests methods from the WebApiCallsBase class

The ExecuteRequestAndProcessResponse method handles response messages from Post and Put calls by executing a request, checking the response for success and returning the content of the message body. The CreateStringContent method is used to format the payload for request messages by creating an instance of the StringContent class. Those methods are shown in Source code 4.24.



Source code 4.24. The helper methods from the WebApiCallsBase class

The SubmitPostRequestAsync and SubmitPutRequestAsync use the PostAsync and PutAsync helpers, and the CreateStringContent method, to create their respective tasks and processing them using the ExecuteRequestAndProcessResponse method. Those methods are presented in Source code 4.25.



Source code 4.25. The post request and put request methods from WebApiCallsBase class

The SubmitDeleteRequestAsync method uses the DeleteAsync helper to execute and HTTP Delete.



Source code 4.26. The delete request method from WebApiCallsBase class

The WebApiCalls class in Source code 4.27. encapsulated all the API calls to the Core MVC service. This class implements the interface IWebApiCalls and inherits from the class WebApiCallsBase.

The constructor of the WebApiCalls class accepts an IWebServiceLocator instance and passes it to the base class constructor. The methods in this class call the base methods from the WebApiCallsBase abstract class using specific values passed from the controllers.



Source code 4.27. The WebApiCalls class with its constructor and some of the most important methods

The controllers AccountController and ManageController were generated alongside the models, view models and views needed by the ASP.NET Core Identity [6] membership system which adds login functionality to an ASP.NET Core application.

The controllers CartController, OrdersController and ProductsController all inherit from Controller and use an instance of IWebApiCalls to call into the service. The default route uses

the ProductsController shown in Source code 4.28.



Source code 4.28. The ProductsController class and its contructor

All the actions from those controllers either return a View or redirect to another action when they are done. Some of the actions use the WebApiCalls class to make the appropriate calls to the server before returning a View or redirecting to another action, for example the action in Source code 4.28.



Source code 4.29. The GetListOfProducts helper method from the ProductsController class

“Views encapsulate the presentation layer in Core MVC Web Applications. They are written using a combination of HTML markup, CSS, JavaScript, and Razor syntax”. [4]

For each controller there is a directory under the Views folder where its specific views are stored, the views are named after the action methods by default.

An example of a view is shown in Source code 4.30. This views main role is to loop through the records using the ProductAndCategoryBase display template.



Source code 4.30. The ProductList view under the View\Products folder.

The ProductAndCategoryBase display template shown in Source code 4.31. displays the properties of a product using the ProductAndCategoryBase view model.

****

Source code 4.31. The ProductAndCategoryBase view under the View\Products\DisplayTemplates folder

**“***View models* are used to shape the data into a form that is more useful for presenting the data. They are typically created by combining parts of two or more existing models into one class”. [4]

The ProductAndCategoryBase view model combines the Product and Category data so it can be easier to render in views.

And with this the implementation of the application part is done.

## **4.4. Testing**

## **4.5. Deployment**

# **Chapter 5 - Conclusions**

# **Bibliography**

[1] ASP.NET Application Life Cycle Overview for IIS 5.0 and 6.0

<https://msdn.microsoft.com/en-us/library/ms178473.aspx>

[2] ASP.NET Core fundamentals

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-2.1&tabs=aspnetcore2x>

[3] Entity Framework Core

<https://docs.microsoft.com/en-us/ef/core/>

[4] [Philip Japikse](https://www.google.ro/search?q=building+web+applications+with+visual+studio+2017:+using+.net+core+and+modern+javascript+frameworks+philip+japikse&stick=H4sIAAAAAAAAAOPgE-LVT9c3NEw2rSpOTk4xVIJwk8wM0kyzKzO0ZLKTrfST8vOz9cuLMktKUvPiy_OLsq0SS0sy8osACz88Mj4AAAA&sa=X&ved=2ahUKEwjhjpGP0fHcAhUIQpoKHRmBB84QmxMoATAcegQIBxAf), [Kevin Grossnicklaus](https://www.google.ro/search?q=building+web+applications+with+visual+studio+2017:+using+.net+core+and+modern+javascript+frameworks+kevin+grossnicklaus&stick=H4sIAAAAAAAAAOPgE-LVT9c3NEw2rSpOTk4xVOIBc3PLis0Ny9O1ZLKTrfST8vOz9cuLMktKUvPiy_OLsq0SS0sy8osAck3k2z0AAAA&sa=X&ved=2ahUKEwjhjpGP0fHcAhUIQpoKHRmBB84QmxMoAjAcegQIBxAg), [Ben Dewey](https://www.google.ro/search?q=building+web+applications+with+visual+studio+2017:+using+.net+core+and+modern+javascript+frameworks+ben+dewey&stick=H4sIAAAAAAAAAOPgE-LVT9c3NEw2rSpOTk4xVOIBc02M0wwL44u0ZLKTrfST8vOz9cuLMktKUvPiy_OLsq0SS0sy8osANx1UKD0AAAA&sa=X&ved=2ahUKEwjhjpGP0fHcAhUIQpoKHRmBB84QmxMoAzAcegQIBxAh): Building Web Applications with Visual Studio 2017, Apress, 2017

[5] [Pravinkumar Dabade](http://www.dotnetcurry.com/Author.aspx?AuthorName=Pravinkumar%20Dabade): ASP.NET Application and Page Life Cycle Overview <http://www.dotnetcurry.com/aspnet/1263/aspnet-application-page-life-cycle-interview-question>

[6] Rick Anderson: Introduction to Identity on ASP.NET Core

<https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity?view=aspnetcore-2.1&tabs=visual-studio#identity-components>

[7] Shivprasad Koirala: ASP.NET Application and Page Life Cycle <https://www.codeproject.com/Articles/73728/ASP-NET-Application-and-Page-Life-Cycle>

[8] Steve Smith: Overview of ASP.NET Core MVC

<https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-2.1>