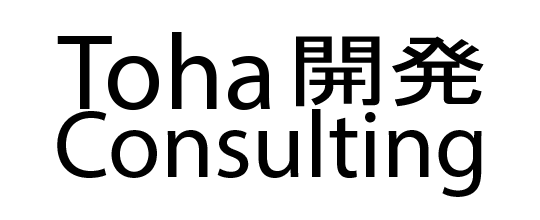
|  |  |
| --- | --- |
| https://lh6.googleusercontent.com/Z3T1eJ49qCbdSHtd-IGGEPaR5XgjwhI5y0LYYW4Sx72ADEtm4DSPF1WGejCZslY3Jb5BKPGiCPY7HobLGf2AxxESllXzVkLN9f1auwxJeGjcPDW4BMlXj2ixbST2_TSlIsfPF-CM  Web API in asp.net  For E-Commerce | Abstract  Learn to implement a Web API to your current website to dynamically and securely retrieve data from a database. Create a registration page and authenticate it using web tokens.  Dr. Drew Hwang  CIS 451 |

Web API in asp.net

By



**Toha Consulting Associates**

*Project Manager*: Carlo Padilla        *Developer/Project Assist*: Tyler Hawkins

*Developer*: Mitzi Montes            *Developer*: Johnny Marquez

*Developer*: Ryan Thai

Contents

[**Toha Consulting Associates** 1](#_Toc515972075)

[Getting Started 3](#_Toc515972076)

[Client-side Setup 23](#_Toc515972077)

[Security and Security Implementation 41](#_Toc515972078)

Convert to using SSL……………………………………………………………………………………………………………………..……50

[Add Registration Code 51](#_Toc515972079)

## Getting Started

Prerequisites:

* Newest version of Microsoft Visual Studio 2017 with .NET Core SDK 2.0

Note: You can download free copy if you are a student at Cal Poly Pomona at the following link - <https://e5.onthehub.com/WebStore/Security/Signin.aspx?ws=330817f9-cb9b-e011-969d-0030487d8897&vsro=8&rurl=%2fWebStore%2fProductsByMajorVersionList.aspx%3fws%3d330817f9-cb9b-e011-969d-0030487d8897%26vsro%3d8>

* POSTMAN to test your API

Note: You can download a free version at the following link – <https://www.getpostman.com>

POSTMAN basics YouTube video - <https://www.youtube.com/watch?v=t5n07Ybz7yI>

* Ecommerce project website

What is a web API?

Before we get started with the guide, let us first define what a web API is. A web API is a service. Technically there is no difference between a web API and a web service other than the way a web API is intended to be used. Badrinarayanan Lakshmiraghavan explains it well:

Let’s say I have a web application where a user can post his thoughts in the form of a short

message. A user can log in to my application from a browser, add new posts or update the ones she posted in the

recent past, or even delete the old ones. In other words, users can perform create, read, update, and delete (CRUD)

operations on their posts using my web application. My application became so popular that there are folks who want

to integrate this CRUD functionality into their mobile apps so that users can perform CRUD operations from their

mobile devices without logging on to my web application while they are away from their normal computers.

I can now create a web service to support the CRUD operations. Technically it is a web service, but it is an

application programming interface (API) to interact with my web application, except that it is over the Web.

Traditionally, APIs are a bunch of classes with properties and methods that are part of a reusable component to

interact with another application. This scenario is exactly that, except that my API is not available in the form of a

software component, but over the Web instead. It is a web API!

Now that we better understanding of what a web API is, let’s get started with the guide. While this guide has code, it is extremely recommended that you type it out yourself and review it to better understand the concepts. Copy and pasting code will not give you the intended outcome because there are parts in the code that require you to use your own configurations/settings and understand the logic to achieve the intended outcome.

First open Microsoft Visual Studio and create a new project:

* File > New > Project > Visual C#(under “Installed Window”) > .NET Core (Click drop down arrow if necessary) > ASP.NET Core web application > Name it eCommerceAPI > OK > empty > OK

After creating the project, you must install NuGet packages:

* Right click eCommercAPI > Manage Nuget packages > browse > search automapper > install the one by Jimmy Bogard > Click OK on the “Preview Changes” window > Click “I Accept” on the “License Acceptance”

Now, we set the user secrets connection string to connect to the correct database:

* Solution Explorer > Right click eCommerceAPI > Manage User secrets

Note: Data source string needs to be all on one line.

* The example below is for reference only, you must insert the path of your ecommerce database mdf file which should be located under App\_Data folder.

{

"connectionStrings": {

"sqlConnectionEcomm": "Data Source=(LocalDB)\\MSSQLLocalDB;AttachDbFilename=C:\\inetpub\\wwwroot\\MyEcommerceSite\\App\_Data\\OnlineStore.mdf;Integrated Security=True"

}

}

**Entities** are objects that represent a database entry. It’s the form the data takes when on the server and when it’s between the server and database.

Next we create an entities folder and add the appropriate classes:

* Go to the solution explorer and right click ecommerAPI > Add > New Folder > Name it “Entities” > Right click Entities folder > Add > New Item > Class > Name it Cart.cs > Add.

Note: Variable names should your match your database column names.

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace eCommerceAPI.Entities

{

public class Cart

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int id { get; set; }

public string CartID { get; set; }

public string ProductID { get; set; }

public string ProductName { get; set; }

public decimal Price { get; set; }

public int Quantity { get; set; }

public decimal LineTotal { get; set; }

}

}

* Create another Entity class inside the Entities folder by right clicking Entities folder > Add > New Item > Class > Name it Product.cs > Add.

Note: As mentioned before class names and variables name must match your database.

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace eCommerceAPI.Entities

{

public class Product

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int ProductID { get; set; }

public string ProductNo { get; set; }

public string ProductName { get; set; }

public string ProductDescription { get; set; }

public decimal Price { get; set; }

public int SubCategoryID { get; set; }

public string Featured { get; set; }

public int MainCategoryID { get; set; }

}

}

* Add another Entity named SqlDbContext.cs

using Microsoft.EntityFrameworkCore;

namespace eCommerceAPI.Entities

{

public class SqlDbContext : DbContext

{

public DbSet<Cart> Cart { get; set; }

public DbSet<Product> Product { get; set; }

public SqlDbContext(DbContextOptions<SqlDbContext> options)

: base(options)

{

}

}

}

When data is sent or received from outside server it is in the form of a **Model**,a **Data Transfer Object (DTO)** in particular. Entity Framework allows us to shift between Entities and Models. Think of the **DTO** as a package that is packed up to ship and the **Entity** as the contents of the package itself.

Now, we are going to add a folder named Models:

* Right click eCommerceAPI > Add > New Folder > Name it Models
* Add new classes CartDTO and ProductDTO

Note: Redo same steps as before to add the new classes.

namespace eCommerceAPI.Models

{

public class CartDTO

{

public int id { get; set; }

public string CartID { get; set; }

public string ProductID { get; set; }

public string ProductName { get; set; }

public decimal Price { get; set; }

public int Quantity { get; set; }

public decimal LineTotal { get; set; }

}

}

namespace eCommerceAPI.Models

{

public class CartDTO

{

public int id { get; set; }

public string CartID { get; set; }

public string ProductID { get; set; }

public string ProductName { get; set; }

public decimal Price { get; set; }

public int Quantity { get; set; }

public decimal LineTotal { get; set; }

}

}

For **Services** we will only be using Entity Framework which helps us convert between **Entity** and **DTO**. The use of generics allows late binding of data types.

Add a new folder named Services and add the following classes to that folder

* Create Services folder > Add new class IGenericEFRepository.cs

Note: Follow the same steps as before to add new classes to the services folder.

using System.Collections.Generic;

namespace eCommerceAPI.Services

{

public interface IGenericEFRepository

{

IEnumerable<TEntity> Get<TEntity>() where TEntity : class;

TEntity Get<TEntity>(int id, bool includeRelatedEntities = false) where TEntity : class;

void Add<TEntity>(TEntity item) where TEntity : class;

void Delete<TEntity>(TEntity item) where TEntity : class;

bool Exists<TEntity>(int id) where TEntity : class;

bool Save();

}

}

* Add a class named GenericEFRepository.cs to the Services folder

using eCommerceAPI.Entities;

using System.Collections.Generic;

using System.Linq;

using System.Reflection;

namespace eCommerceAPI.Services

{

public class GenericEFRepository : IGenericEFRepository

{

private SqlDbContext \_db;

public GenericEFRepository(SqlDbContext db)

{

\_db = db;

}

public IEnumerable<TEntity> Get<TEntity>() where TEntity : class

{

return \_db.Set<TEntity>();

}

public TEntity Get<TEntity>(int id, bool includeRelatedEntities = false) where TEntity : class

{

var entity = \_db.Set<TEntity>().Find(new object[] { id });

if (entity != null && includeRelatedEntities)

{

// Get the names of all DbSets in the DbContext

var dbsets = typeof(SqlDbContext)

.GetProperties(BindingFlags.Public | BindingFlags.Instance)

.Where(z => z.PropertyType.Name.Contains("DbSet"))

.Select(z => z.Name);

// Get the names of all the properties (tables) in the generic

// type T that is represented by a DbSet

var tables = typeof(TEntity)

.GetProperties(BindingFlags.Public | BindingFlags.Instance)

.Where(z => dbsets.Contains(z.Name))

.Select(z => z.Name);

// Load all the tables referenced by the generic type T

if (tables.Count() > 0)

foreach (var table in tables)

\_db.Entry(entity).Collection(table).Load();

}

return entity;

}

public void Add<TEntity>(TEntity item) where TEntity : class

{

\_db.Add<TEntity>(item);

}

public void Delete<TEntity>(TEntity item) where TEntity : class

{

\_db.Set<TEntity>().Remove(item);

}

public bool Exists<TEntity>(int id) where TEntity : class

{

return \_db.Set<TEntity>().Find(new object[] { id }) != null;

}

public bool Save()

{

return \_db.SaveChanges() >= 0;

}

}

}

**Controllers** handles the routing of our API requests and tells the API what to do. This is where we introduce HTML verbs starting with GET.

Next, we create a folder named controllers and add the following classes:

* Create new folder name Controllers > Add > New Item > API Controller > Name the class ProductController.cs

using eCommerceAPI.Entities;

using eCommerceAPI.Models;

using eCommerceAPI.Services;

using AutoMapper;

using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

using System.Linq;

namespace eCommerceAPI.Controllers

{

[Route("api/product")]

public class ProductController : Controller

{

IGenericEFRepository \_rep;

public ProductController(IGenericEFRepository rep)

{

\_rep = rep;

}

// GET api/product

[HttpGet]

public IActionResult Get()

{

var items = \_rep.Get<Product>();

var DTOs = Mapper.Map<IEnumerable<ProductDTO>>(items);

return Ok(DTOs);

}

// GET api/product/:productId:

[HttpGet("{productId}", Name = "GetGenericProduct")]

public IActionResult Get(int productId)

{

var products = \_rep.Get<Product>().Where(p =>

p.ProductID.Equals(productId));

var DTOs = Mapper.Map<IEnumerable<ProductDTO>>(products);

return Ok(DTOs);

}

}

}

Now, we are going to add the following code to Startup.cs:

* Right click Startup.cs in the Solution Explorer > Open

Note: Read through the comments in the code to understand the concepts.

using System;

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Configuration;

using eCommerceAPI.Services;

using eCommerceAPI.Entities;

using eCommerceAPI.Models;

using Microsoft.EntityFrameworkCore;

namespace eCommerceAPI

{

public class Startup

{

public IConfiguration Configuration { get; set; }

public Startup(IConfiguration configuration)

{

Configuration = configuration;

}

// This method gets called by the runtime. Use this method to add services to the container.

// For more information on how to configure your application, visit https://go.microsoft.com/fwlink/?LinkID=398940

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

var conn = Configuration["connectionStrings:sqlConnectionEcomm"];

//SqlDbContext is our connection to the DB using our connection string from secrets.json(conn)

services.AddDbContext<SqlDbContext>(options =>

options.UseSqlServer(conn));

//Entity Framework allows interaction with DB, Generics allow data types to be assigned at runtime

services.AddScoped(typeof(IGenericEFRepository), typeof(GenericEFRepository));

//Using AutoMapper Package to Map Entities to DTOs AND vice versa

//Entities represent tables in the DB

//Data Transfer Object is used to turn entity data into a response object OR convert request data into an entity model

AutoMapper.Mapper.Initialize(config =>

{

config.CreateMap<Entities.Cart, Models.CartDTO>();

config.CreateMap<Models.CartDTO, Entities.Cart>();

config.CreateMap<Entities.Product, Models.ProductDTO>();

config.CreateMap<Models.ProductDTO, Entities.Product>();

});

}

// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

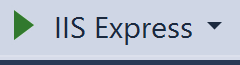
}

app.UseMvc();

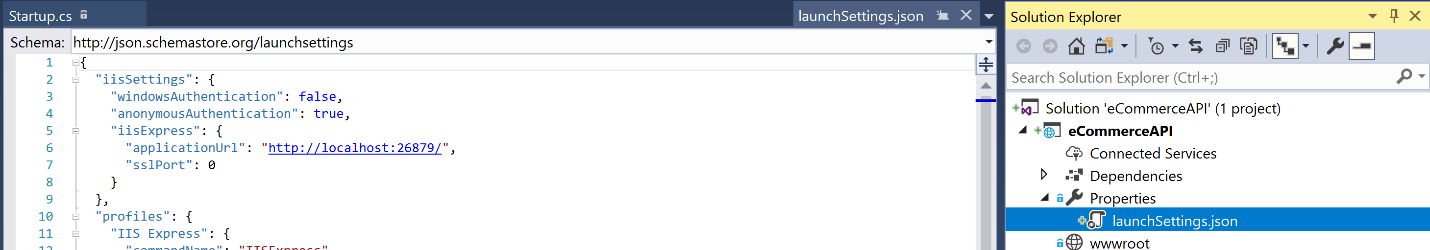
}

}

}

We now test the project by running it in IIS Express  and testing it in the downloaded program POSTMAN. Make certain that you use the localhost port that your project is using. If you are unsure you can run the project and the localhost port should be displayed in the URL or you can open the projects launchSettings.json file to view the port.



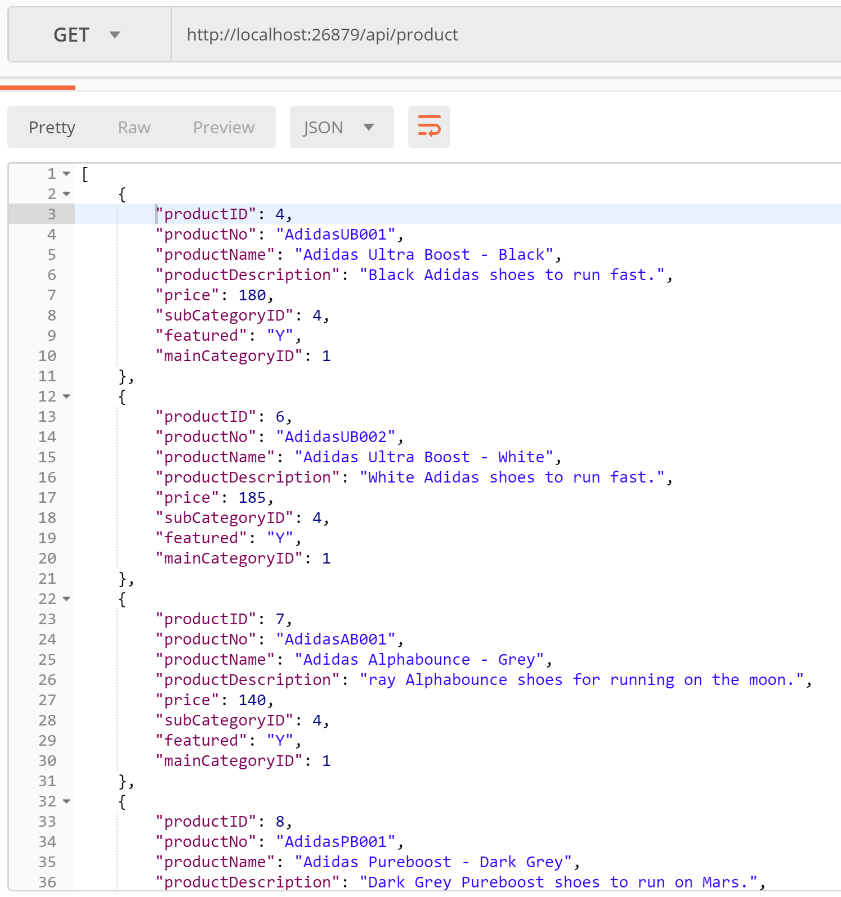


The concepts used in the following pages are part of a very important aspect of a client and server communication. Remember that the HTTP client (browser) is always initiating the request to the HTTP server. The HTTP protocol gives us a few tools to express its intention to the server: the HTTP methods, HTTP headers, and the URI. The URI is a system for identifying pieces of information on the network. The HTTP Methods contain 8 methods for requesting a URI, but we are going to only use the following 4: GET, POST, PUT, DELETE. For more information visit the following link – <https://dev.opera.com/articles/http-lets-get-it-on/>. We start by using the most commonly used, GET.

* GET <http://localhost:26879/api/product> and compare the results to the following:

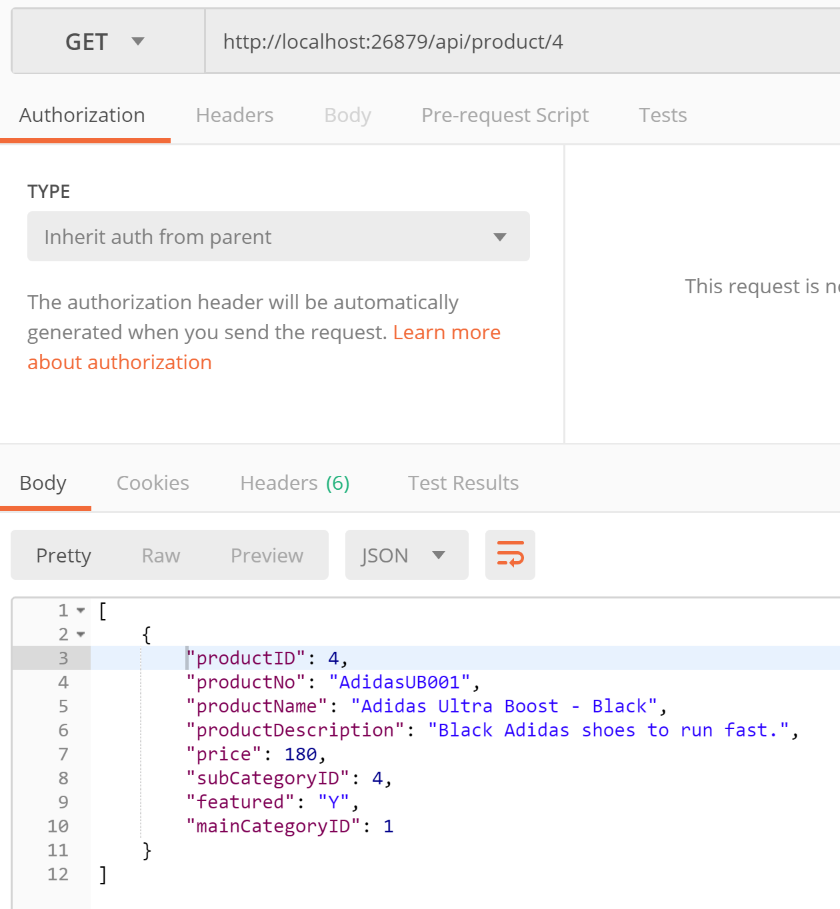
Note: Remember to change the port number to your projects localhost port number.

* GET requests are the most common. When you type a web address like facebook.com you are actually performing a GET request. Something you can try is entering your localhost endpoint (<http://localhost:26879/api/product>) into your web browser and it will load and return a JSON string.
* Web browsers typically only issue GETs, but we can use special tools like Postman, curl, or AJAX for the other HTML verbs like POST and DELETE.



* GET <http://localhost:26879/api/product/4>

Note: The 4 in the link should be replaced wit a product number that matches your database.



Now we add code to be able to do a POST:

* Open ProductController class and add the following code under the 2 GETs:

Note: POST is to create, in this case a new Product entry. Remember to stop the program before adding the following code.

// POST api/product/

[HttpPost]

public IActionResult Post([FromBody]ProductDTO DTO)

{

if (DTO == null) return BadRequest();

if (!ModelState.IsValid) return BadRequest(ModelState);

var itemToCreate = Mapper.Map<Product>(DTO);

\_rep.Add(itemToCreate);

if (!\_rep.Save()) return StatusCode(500,

"A problem occurred while handling your request.");

var createdDTO = Mapper.Map<ProductDTO>(itemToCreate);

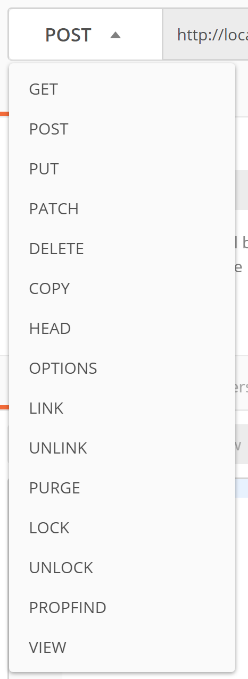
return CreatedAtRoute("GetGenericProduct",

new { productId = createdDTO.ProductID }, createdDTO);

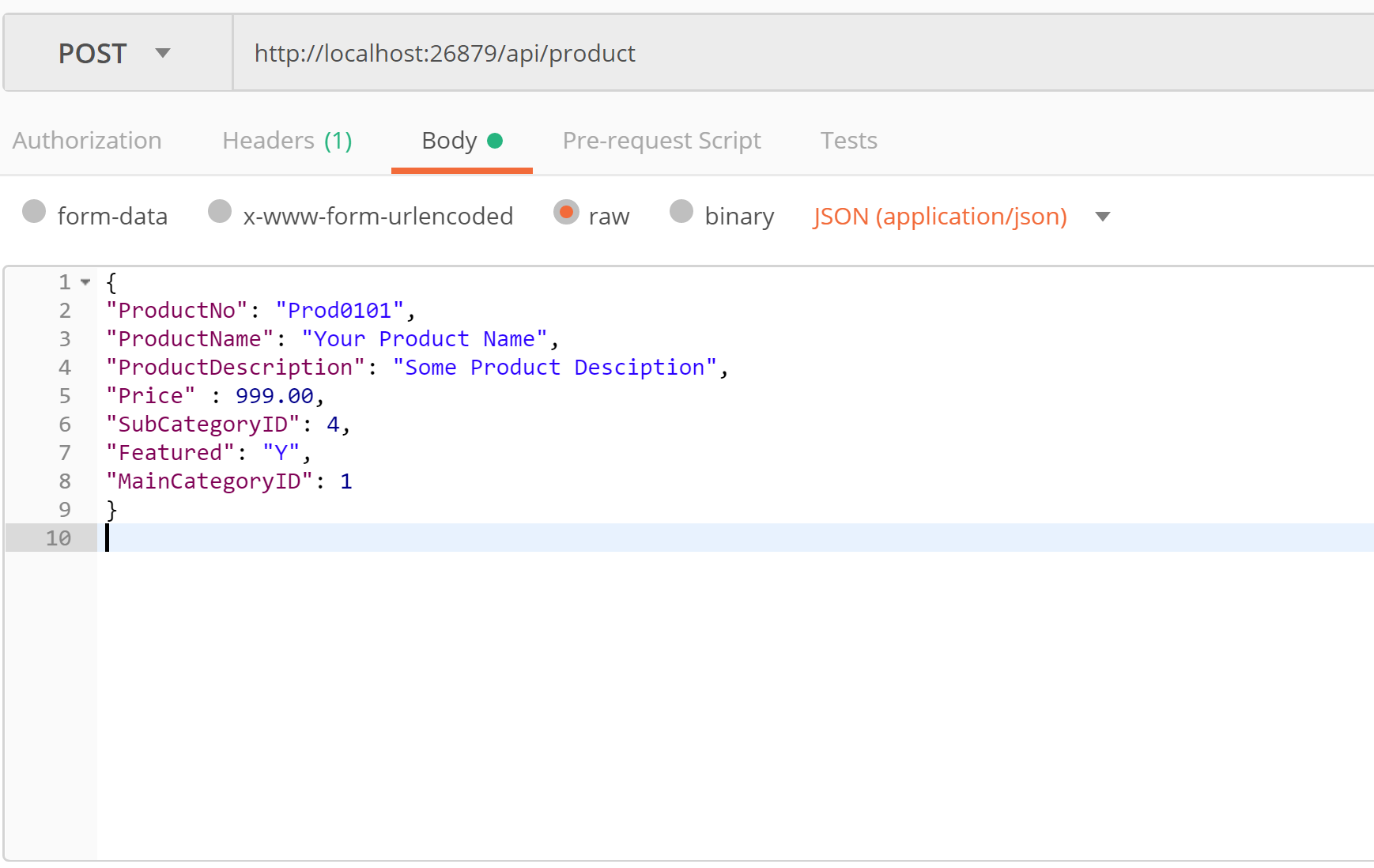
}

Now we test POST with POSTMAN, run the project in IIS Express and test the following:

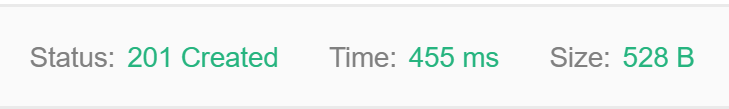
* Change GET to POST in POSTMAN



* POST <http://localhost:26879/api/product>
* Remember your port number will likely be different.
* To create a new entry we need to provide the parameters so we will include JSON in the body of the request.
* Create a new json object by going to Body > raw > JSON (application/json) in POSTMAN
* Finally, send the request to retrieve the results



* Check bottom right of POSTMAN to check if the POST was successful, if it was successful it should say:



* Now do a GET to retrieve the item just added
* GET <http://localhost:26879/api/product>
* Compare results:



Now we add a DELETE method to ProductController class:

// DELETE api/product/:productId:

[HttpDelete("{productId}")]

public IActionResult Delete(int productId)

{

if (!\_rep.Exists<Product>(productId)) return NotFound();

var entityToDelete = \_rep.Get<Product>(productId);

\_rep.Delete(entityToDelete);

if (!\_rep.Save()) return StatusCode(500,

"A problem occurred while handling your request.");

return NoContent();

}

Test using POSTMAN, remember to run project with IIS:

* DELETE <http://localhost:26879/api/product/43>

Note: The number 43 should be replaced with a productid from your database.

* “Status 204 No Content” signifies that the deletion was successful.



* Validate that the item was deleted by doing a GET

Add PUT method to ProductController which is an update to an existing entry similar to an overwrite, it needs all values to be passed to be performed:

// PUT api/product/:productId:

[HttpPut("{id}")]

public IActionResult Put(int id, [FromBody]ProductUpdateDTO DTO)

{

if (DTO == null) return BadRequest();

if (!ModelState.IsValid) return BadRequest(ModelState);

var entity = \_rep.Get<Product>(id);

if (entity == null) return NotFound();

Mapper.Map(DTO, entity);

if (!\_rep.Save()) return StatusCode(500,

"A problem happened while handling your request.");

return NoContent();

}

The PUT method is different from the other methods, it needs a new ProductUpdateDTO class added to Models, it is the same as ProductDTO except without ID. This is because the controller will get the ID from the localhost URI (Uniform Resourcse Identifier) and won’t be changing it :

// PUT api/product/:productId:

[HttpPut("{id}")]

public IActionResult Put(int id, [FromBody]ProductUpdateDTO DTO)

{

if (DTO == null) return BadRequest();

if (!ModelState.IsValid) return BadRequest(ModelState);

var entity = \_rep.Get<Product>(id);

if (entity == null) return NotFound();

Mapper.Map(DTO, entity);

if (!\_rep.Save()) return StatusCode(500,

"A problem happened while handling your request.");

return NoContent();

}

You also need to add the new DTO to Startup.cs for AutoMapper under “config.CreateMap<Models.ProductDTO, Entities.Product>();”:

config.CreateMap<Entities.Product, Models.ProductUpdateDTO>();

config.CreateMap<Models.ProductUpdateDTO, Entities.Product>();

* Test in POSTMAN
* PUT <http://localhost:26879/api/product/11>

Note: You want to change the number 11 to the ID you want to update in your database.

* Create JSON for the new values you want to update

{

"ProductNo": "Prod0102",

"ProductName": "Your Product Name",

"ProductDescription": "Some Product Desciption",

"Price" : 99.00,

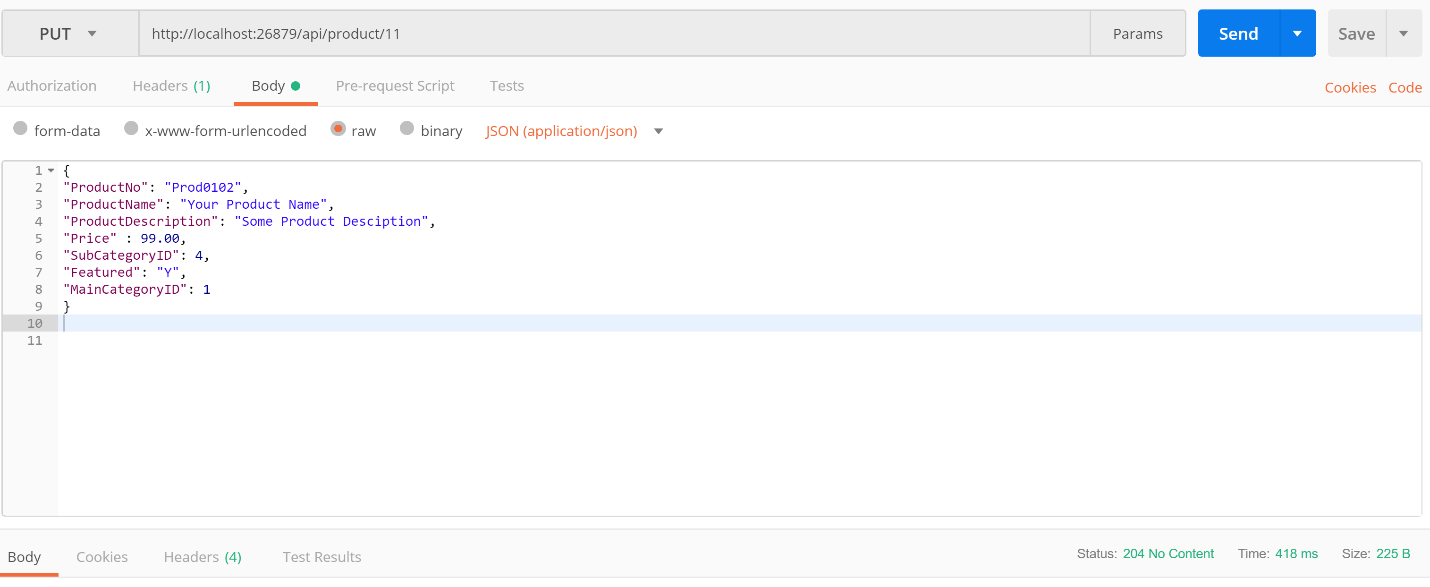
"SubCategoryID": 4,

"Featured": "Y",

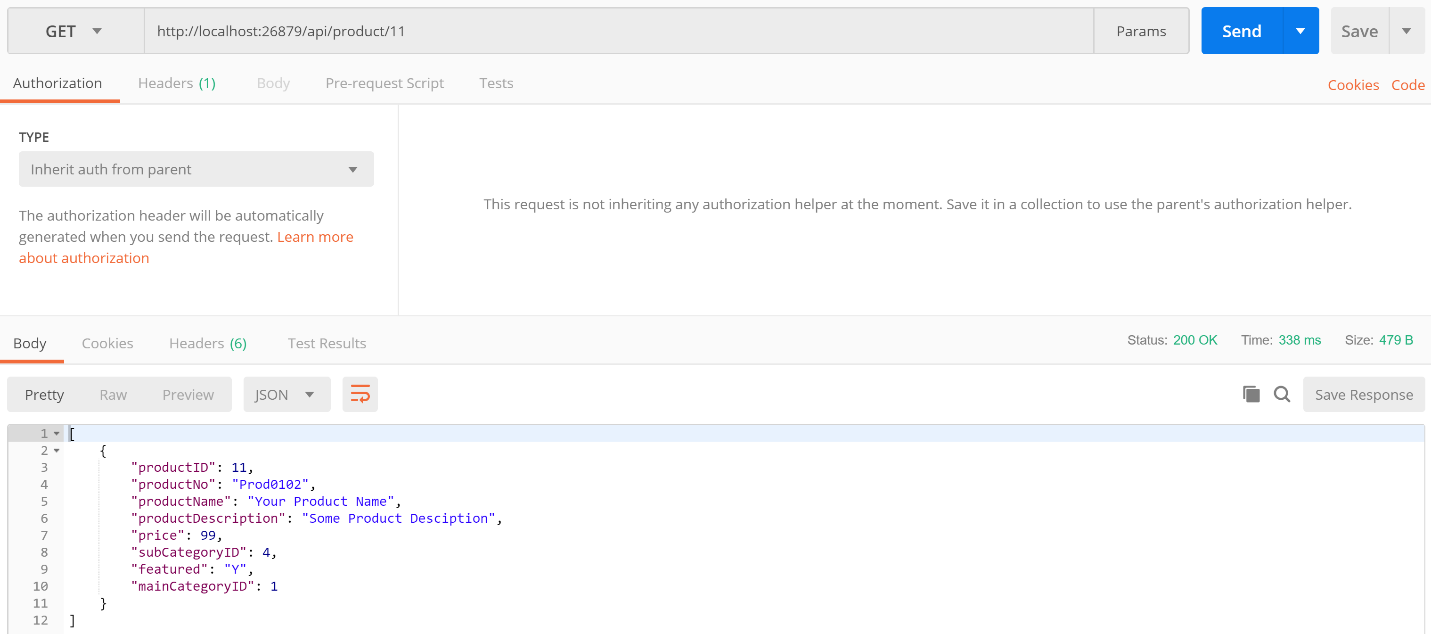
"MainCategoryID": 1

}

* “204 No Content” indicates the PUT was successful



* Validate by returning the updated item using a GET



Next, we focus on cart:

* Right click Controllers folder > Add new item API Controller Class > Name it CartController.cs

using eCommerceAPI.Entities;

using eCommerceAPI.Models;

using eCommerceAPI.Services;

using AutoMapper;

using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

using System.Linq;

using Microsoft.AspNetCore.JsonPatch;

namespace AspNetCoreWebAPI.Controllers

{

[Route("api/cart")]

public class CartController : Controller

{

IGenericEFRepository \_rep;

public CartController(IGenericEFRepository rep)

{

\_rep = rep;

}

// GET api/cart

[HttpGet]

public IActionResult Get()

{

var item = \_rep.Get<Cart>();

var DTOs = Mapper.Map<IEnumerable<CartDTO>>(item);

return Ok(DTOs);

}

// GET api/cart/:cartID:

[HttpGet("{cartId}")]

public IActionResult Get(string cartId)

{

var carts = \_rep.Get<Cart>().Where(c =>

c.CartID.Equals(cartId));

var DTOs = Mapper.Map<IEnumerable<CartDTO>>(carts);

return Ok(DTOs);

}

// DELETE api/cart/:id:

[HttpDelete("{id}")]

public IActionResult Delete(int id)

{

if (!\_rep.Exists<Cart>(id)) return NotFound();

var entityToDelete = \_rep.Get<Cart>(id);

\_rep.Delete(entityToDelete);

if (!\_rep.Save()) return StatusCode(500,

"A problem occurred while handling your request.");

return NoContent();

}

// PATCH api/cart/:id:

[HttpPatch("{id}")]

public IActionResult Patch(int id, [FromBody]JsonPatchDocument<CartUpdateDTO> DTO)

{

if (DTO == null) return BadRequest();

if (!ModelState.IsValid) return BadRequest(ModelState);

var entity = \_rep.Get<Cart>(id);

if (entity == null) return NotFound();

var entityToPatch = Mapper.Map<CartUpdateDTO>(entity);

DTO.ApplyTo(entityToPatch, ModelState);

TryValidateModel(entityToPatch);

if (!ModelState.IsValid) return BadRequest(ModelState);

Mapper.Map(entityToPatch, entity);

if (!\_rep.Save()) return StatusCode(500,

"A problem happened while handling your request.");

return NoContent();

}

}

}

* You also need to add CartUpdateDTO.cs to Models folder like previous steps.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace eCommerceAPI.Models

{

public class CartUpdateDTO

{

public string CartID { get; set; }

public string ProductID { get; set; }

public string ProductName { get; set; }

public decimal Price { get; set; }

public int Quantity { get; set; }

public decimal LineTotal { get; set; }

}

}

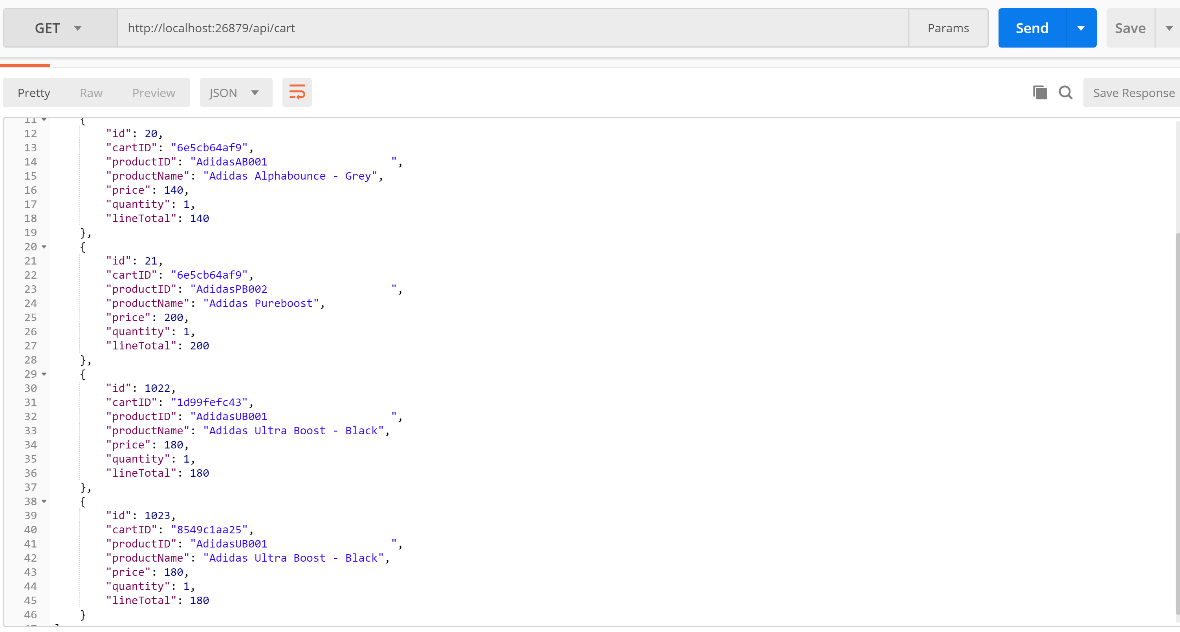
* Add to Startup.cs the following code for AutoMapper below the previously added.

config.CreateMap<Models.CartUpdateDTO, Entities.Cart>();

config.CreateMap<Entities.Cart, Models.CartUpdateDTO>();

To test Cart we again use POSTMAN, running the project in IIS Express. CartController has GET, GET id, DELETE, and PATCH. Patch is the only new method and it is similar to update.

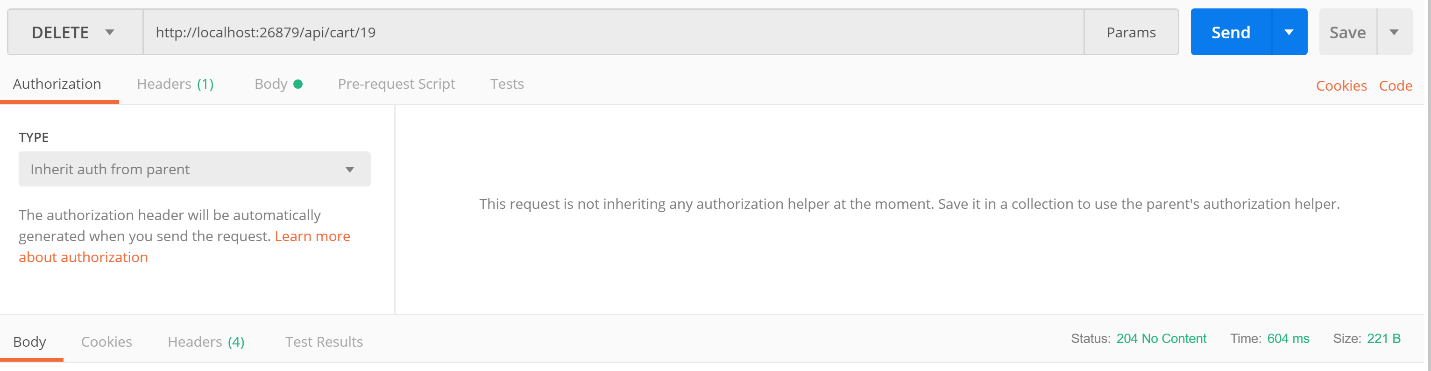
* GET <http://localhost:26879/api/cart>



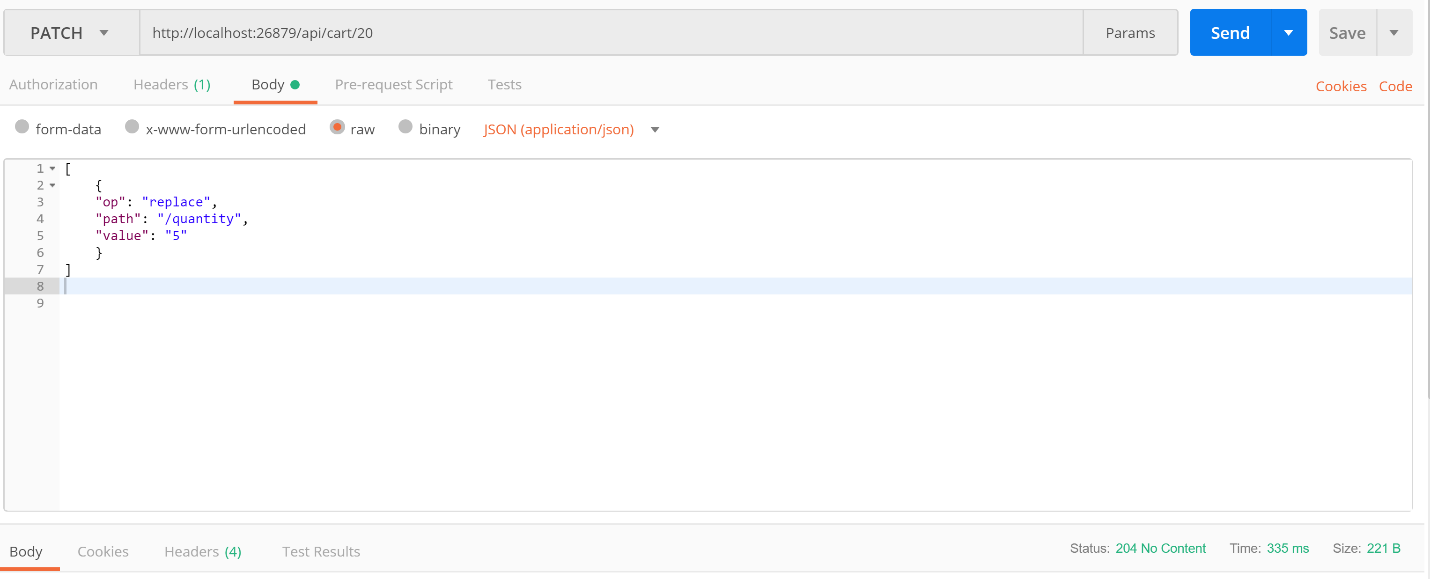
* GET <http://localhost:26879/api/cart/19>

Note: This number should be changed to fit your database cart.

* DELETE <http://localhost:26879/api/cart/19>



* PATCH is a new feature. It acts similar to an update but unlike PUT it only replaces what it is told to replace. We will use patch to update Quantity to 5 for cart entry 20. JSON for PATCH is below.
* PATCH <http://localhost:26879/api/cart/20>



* “204 No Content” indicates a successful PATCH.
* Validate with a GET to see if the quantity updated.

## Client-side Setup

Server-side setup is completed. The guide will now focus on implementing the client side. Launch another instance of Visual Studio and keep the API one open as well. You should have 2 Visual Studios open at the same time.

* In newly opened Visual Studio open your ecommerce website solution.
* Manage NuGet Packages and install NewtonSoft.Json by James-Newton King.
* While in NuGet also install System.Net.Http.



* Right click project name > Add New Item > Web Form in VB > Name it ApiClient and use your master page.
* Right click ApiClient.aspx > Set as Start Page

First, we will make a simple call to get all products using GET and display them in a gridview:

* Go to ApiClient.aspx and add Asnyc= “true” to the Page element like:

<%@ Page Async="true" Title="" Language="VB" MasterPageFile="~/MasterPage.master" AutoEventWireup="false" CodeFile="ApiClient.aspx.vb" Inherits="ApiClient" %>

Within your Content Placeholder add the following code:

* Go to toolbox > Add a button > give it ID="btnAllProducts" and Text="Get All Products"
* Go to toolbox > Add a gridview > give it ID="gvAllProducts"
* Should look similar to this:

<%@ Page Async="true" Title="" Language="VB" MasterPageFile="~/MasterPage.master" AutoEventWireup="false" CodeFile="ApiClient.aspx.vb" Inherits="ApiClient" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">

</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1" Runat="Server">

<!--Get All Products-->

<asp:Button ID="btnAllProducts" runat="server" Text="Get All Products" />

<asp:GridView ID="gvAllProducts" runat="server"></asp:GridView>

</asp:Content>

Now go to the code behind ApiClient.aspx.vb and do the following:

* Import the following packages.

Imports System.Data

Imports System.Net.Http

Imports Newtonsoft.Json

Create a global httpClient right under the code line Inherits Inherits System.Web.UI.Page:

* Dim httpClient As New HttpClient
* HttpClient is similar to POSTMAN. It allows us to send HTTP requests to generate results from the methods we used earlier on.
* Generate an onclick event sub for btnAllProducts and make it Async. Empty sub looks like this.

Private Async Sub btnAllProducts\_ClickAsync(sender As Object, e As EventArgs) Handles btnAllProducts.Click

End Sub

* Add the following code to the click event. Localhost port is the port your API uses.

Dim uri As String = "http://localhost:50360/api/product"

Dim task = Await httpClient.GetAsync(uri)

Dim jsonString = Await task.Content.ReadAsStringAsync()

If task.IsSuccessStatusCode Then

Dim table As DataTable = JsonConvert.DeserializeObject(Of DataTable)(jsonString)

gvAllProducts.DataSource = table

gvAllProducts.DataBind()

End If

* Your code should be like this.

Imports System.Data

Imports System.Net.Http

Imports Newtonsoft.Json

Partial Class ApiClient

Inherits System.Web.UI.Page

Dim httpClient As New HttpClient

Private Async Sub btnAllProducts\_ClickAsync(sender As Object, e As EventArgs) Handles btnAllProducts.Click

Dim uri As String = "http://localhost:50360/api/product"

Dim task = Await httpClient.GetAsync(uri)

Dim jsonString = Await task.Content.ReadAsStringAsync()

If task.IsSuccessStatusCode Then

Dim table As DataTable = JsonConvert.DeserializeObject(Of DataTable)(jsonString)

gvAllProducts.DataSource = table

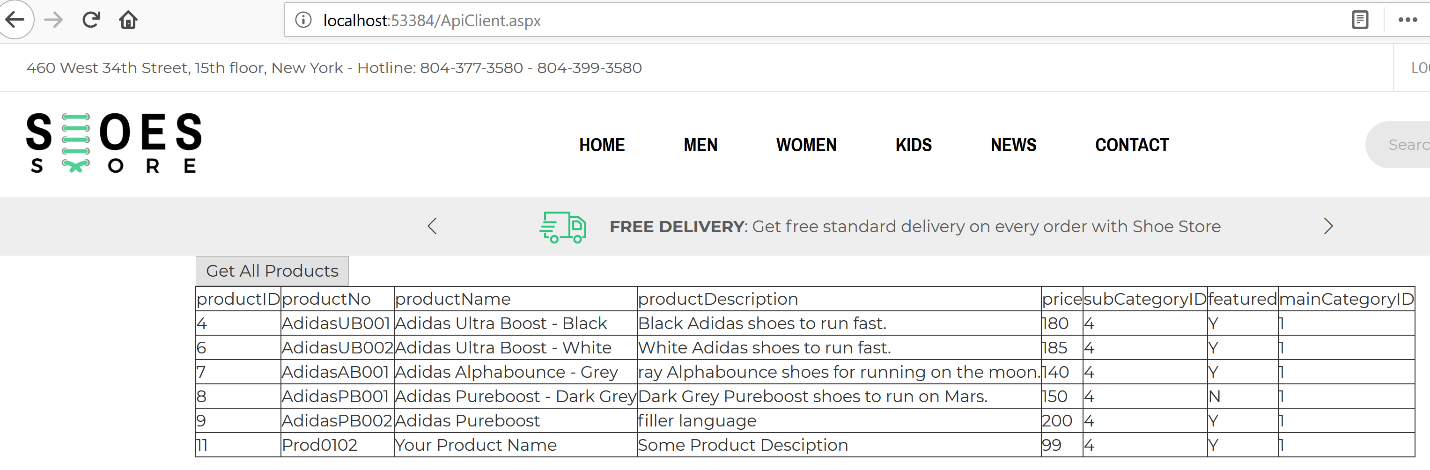
gvAllProducts.DataBind()

End If

End Sub

End Class

Now we are going to test using IIS Express. Client and server both need to be running so they can work together. Client sends requests to server and server responds back. Client is then able to issue a response. You should open a web page in your browse and use <http://localhose:55345/ApiClient.aspx> with the port number being different from your API port number. Click the get button and you should retrieve a table with data if your database is populated.



You may choose to add CSS and HTML to style your page or just use <br /> twice to keep space between the functions. This guide does not include styling to keep the code readable and simpler.

First, we will work on adding the retrieval of product by ID.

* Add the following after the btnAllProducts.

<br />

<br />

<!--Get Product by ID-->

<label>Get Product ID:</label>

<input type="text" id="tbProductID" runat="server" />

<asp:Button ID="btnProductID" runat="server" Text="Get Product by ID" />

<asp:GridView ID="gvProductID" runat="server" ></asp:GridView>

Public Async Sub btnProductID\_ClickAsync(sender As Object, e As EventArgs) Handles btnProductID.Click

Dim uri As String = "http://localhost:50360/api/product/" & tbProductID.Value

Dim task = Await httpClient.GetAsync(uri)

Dim jsonString As String = Await task.Content.ReadAsStringAsync()

If task.IsSuccessStatusCode Then

Dim table As DataTable = JsonConvert.DeserializeObject(Of DataTable)(jsonString)

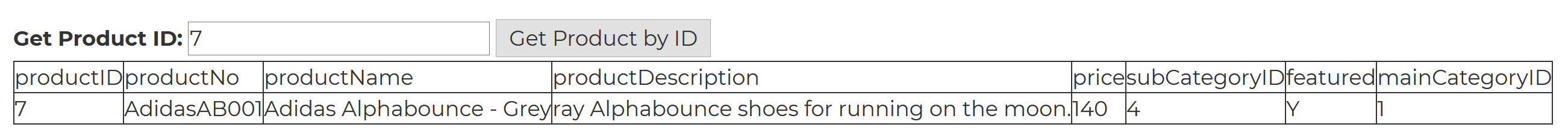
gvProductID.DataSource = table

gvProductID.DataBind()

End If

End Sub

* Test the button



Second, we add the functionality for Add and Update product input fields (ApiClient.aspx), there is no code behind for this. These fields are used for inputs to create the JSON for both add and update products. You can choose to create a separate input form for update if you want.

<label for="tbProductNo">ProductNo:</label>

<input type="text" id="tbProductNo" runat="server" />

<br />

<label for="tbProductName">ProductName:</label>

<input type="text" id="tbProductName" runat="server" />

<br />

<label for="tbProductDescription">ProductDescription:</label>

<input type="text" id="tbProductDescription" runat="server" />

<br />

<label for="tbPrice">Price:</label>

<input type="text" id="tbPrice" runat="server" />

<br />

<label for="tbSubCategoryID">SubCategoryID:</label>

<input type="text" id="tbSubCategoryID" runat="server" />

<br />

<label for="tbFeatured">Featured:</label>

<input type="text" id="tbFeatured" runat="server" />

<br />

Third, we add a button for adding a new product and code to retrieve the inputted information from the user and create a JSON object to send to the database.

* Add the button

<!--Create Product-->

<asp:Button ID="btnCreateProduct" runat="server" Text="Add New Product"/>

* Add the code for the button

Private Async Sub btnCreateProduct\_ClickAsync(sender As Object, e As EventArgs) Handles btnCreateProduct.Click

Dim myJson As String = ("{'ProductNo': '" & tbProductNo.Value & "', 'ProductName': '" & tbProductName.Value & "', 'ProductDescription': '" \_

& tbProductDescription.Value & "', 'Price': '" & tbPrice.Value & "', 'SubCategoryID': '" & tbSubCategoryID.Value \_

& "', 'Featured': '" & tbFeatured.Value & "', 'MainCategoryID': '" & tbMainCategoryID.Value & "'}")

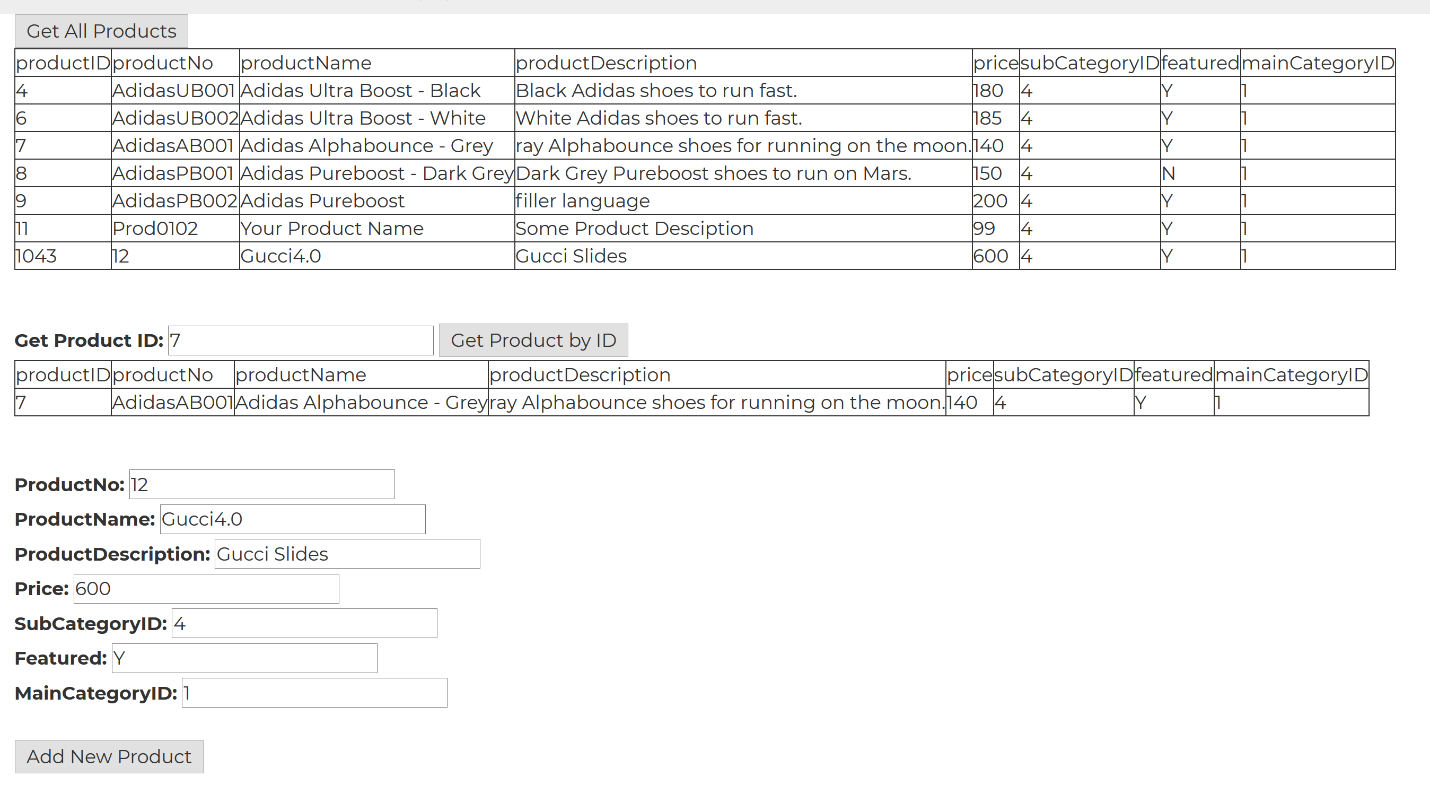
Dim uri As String = "http://localhost:50360/api/product/"

Dim response = Await httpClient.PostAsync(uri, New StringContent(myJson, Encoding.UTF8, "application/json"))

btnAllProducts\_ClickAsync(btnAllProducts, EventArgs.Empty)

End Sub

* Test the newly added button and code by inputting the information and clicking the button to create a new product



Next, we add a button to update a product and its code behind:

* Add the button

<!--Update Product by ID-->

<label for="tbUpdateProductID" >Update Product ID:</label>

<input type="text" id="tbUpdateProductID" runat="server" />

<asp:Button ID="btnUpdateProduct" runat="server" Text="Update Existing Product"/>

* Add the code for the button

Private Async Sub btnUpdateProduct\_ClickAsync(sender As Object, e As EventArgs) Handles btnUpdateProduct.Click

Dim myJson As String = ("{'ProductNo': '" & tbProductNo.Value & "', 'ProductName': '" & tbProductName.Value & "', 'ProductDescription': '" \_

& tbProductDescription.Value & "', 'Price': '" & tbPrice.Value & "', 'SubCategoryID': '" & tbSubCategoryID.Value \_

& "', 'Featured': '" & tbFeatured.Value & "', 'MainCategoryID': '" & tbMainCategoryID.Value & "'}")

Dim uri As String = "http://localhost:50360/api/product/" & tbUpdateProductID.Value

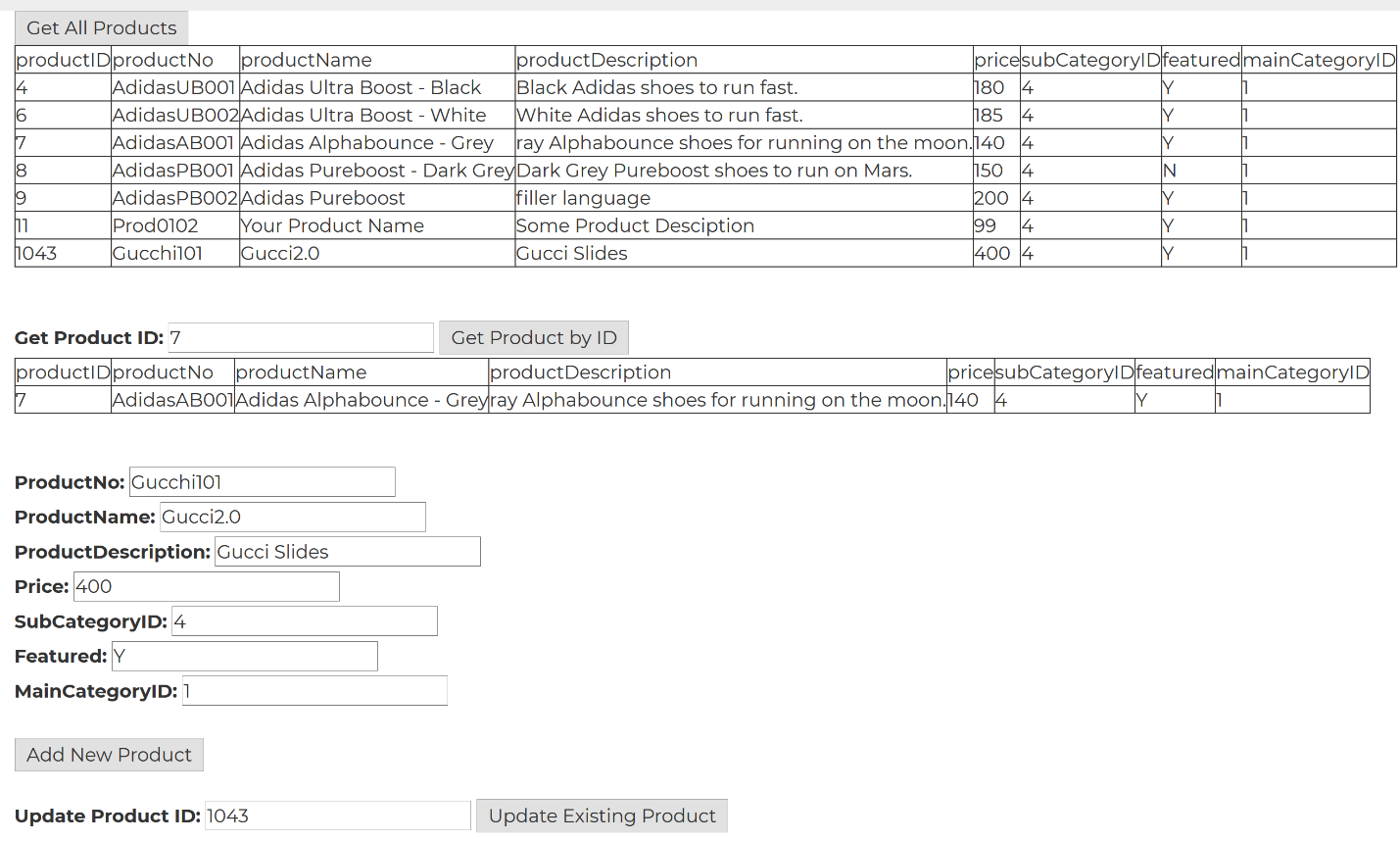
Dim response = Await httpClient.PutAsync(uri, New StringContent(myJson, Encoding.UTF8, "application/json"))

btnAllProducts\_ClickAsync(btnAllProducts, EventArgs.Empty)

btnProductID\_ClickAsync(btnProductID, EventArgs.Empty)

End Sub

* Test the newly added button and code by inputting the information and clicking the button to update a product



Now we add the delete button and its functionality:

* Add button

<!--Delete Product by ID-->

<label>Delete Product ID:</label>

<input type="text" id="tbDeleteProductID" runat="server" />

<asp:Button ID="btnDeleteProductID" runat="server" Text="Delete Product by ID" />

* Add the code to button

Private Async Sub btnDeleteProductID\_ClickAsync(sender As Object, e As EventArgs) Handles btnDeleteProductID.Click

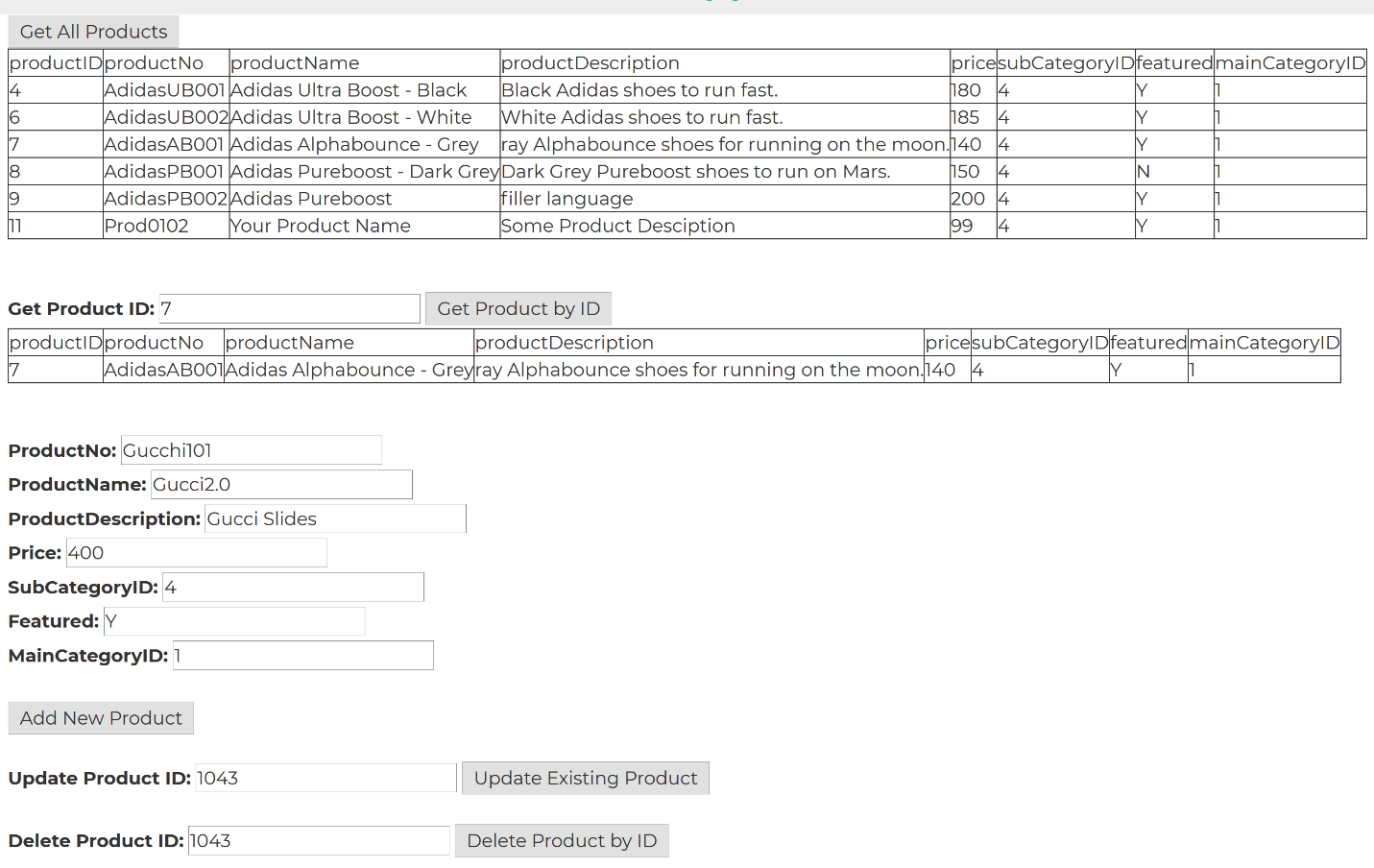
Dim uri As String = "http://localhost:50360/api/product/" & tbDeleteProductID.Value

Await httpClient.DeleteAsync(uri)

btnAllProducts\_ClickAsync(btnAllProducts, EventArgs.Empty)

End Sub

* Test the button by inserting an product ID and deleting it



Add the button for get all cart info and its code behind:

* Add the button

<br />

<br />

<!--Get All Cart Info-->

<asp:Button ID="btnAllCarts" runat="server" Text="Get All Cart Info" />

<asp:GridView ID="gvAllCarts" runat="server"></asp:GridView>

* Add to the button’s code behind

Private Async Sub btnAllCarts\_ClickAsync(sender As Object, e As EventArgs) Handles btnAllCarts.Click

Dim uri As String = "http://localhost:50360/api/cart"

Dim task = Await httpClient.GetAsync(uri)

Dim jsonString = Await task.Content.ReadAsStringAsync()

If task.IsSuccessStatusCode Then

Dim table As DataTable = JsonConvert.DeserializeObject(Of DataTable)(jsonString)

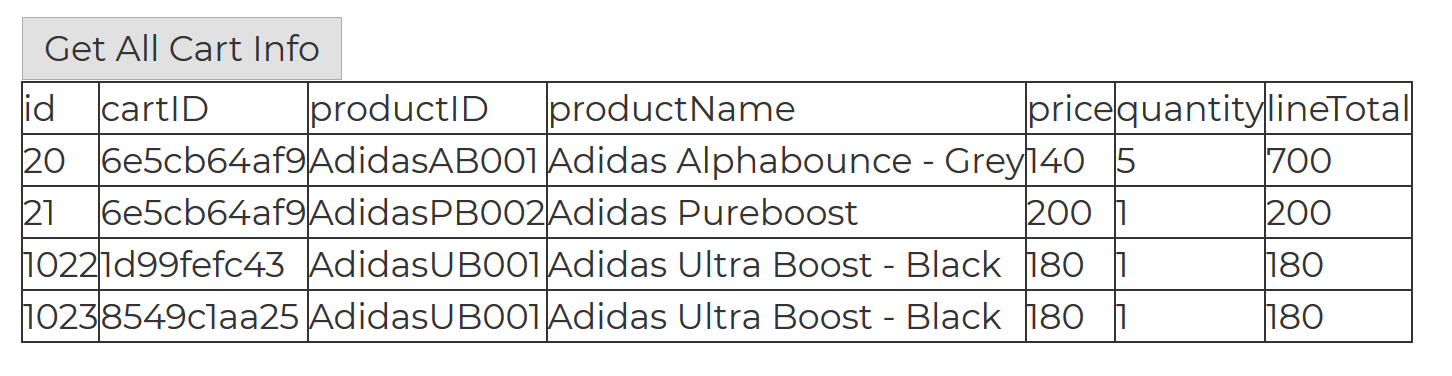
gvAllCarts.DataSource = table

gvAllCarts.DataBind()

End If

End Sub

* Test the button



Add the button to delete Cart Row by id:

* Add button

<br />

<br />

<!--Delete Cart Line-->

<label>Delete Cart Row (id):</label>

<input type="text" id="tbDeleteCartLineID" runat="server" />

<asp:Button ID="btnDeleteCartLineID" runat="server" Text="Delete Cart Line By ID" />

* Add the code behind for the button

Private Async Sub btnDeleteCartLineID\_ClickAsync(sender As Object, e As EventArgs) Handles btnDeleteCartLineID.Click

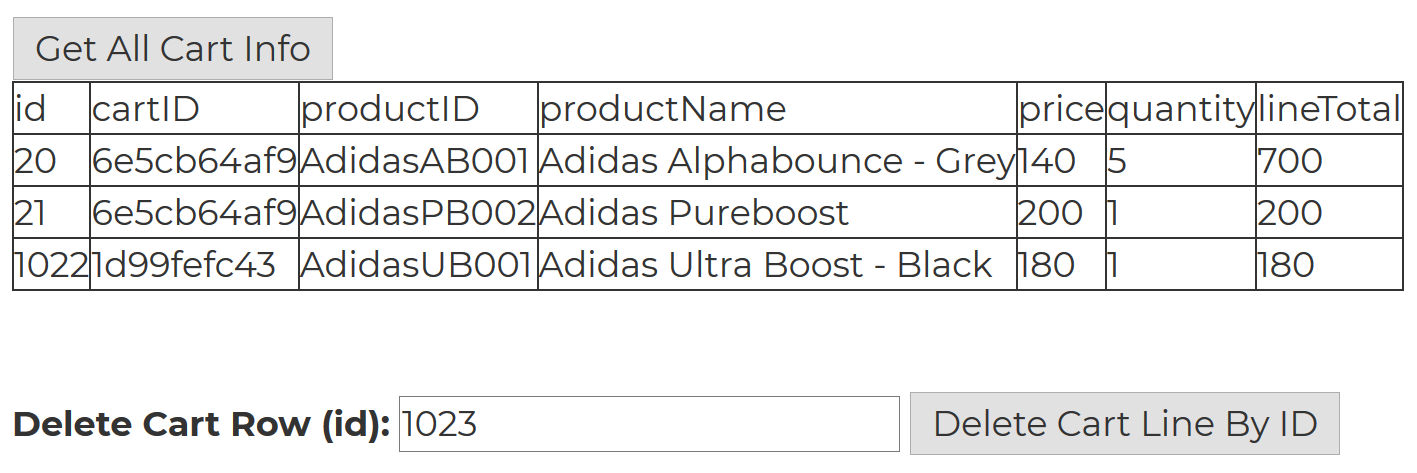
Dim uri As String = "http://localhost:50360/api/cart/" & tbDeleteCartLineID.Value

Await httpClient.DeleteAsync(uri)

btnAllCarts\_ClickAsync(btnAllCarts, EventArgs.Empty)

End Sub

* Test the button



Add the button for getting Entire Cart by CartId:

* Add button

<br />

<br />

<!--Get Cart by CartID-->

<label>Get Entire Cart (CartID):</label>

<input type="text" id="tbGetCartID" runat="server" />

<asp:Button ID="btnCartByID" runat="server" Text="Get Cart by CartID" />

<asp:GridView ID="gvCartByID" runat="server"></asp:GridView>

* Add code behind for the button

Private Async Sub btnCartByID\_ClickAsync(sender As Object, e As EventArgs) Handles btnCartByID.Click

Dim uri As String = "http://localhost:50360/api/cart/" & tbGetCartID.Value

Dim task = Await httpClient.GetAsync(uri)

Dim jsonString = Await task.Content.ReadAsStringAsync()

If task.IsSuccessStatusCode Then

Dim table As DataTable = JsonConvert.DeserializeObject(Of DataTable)(jsonString)

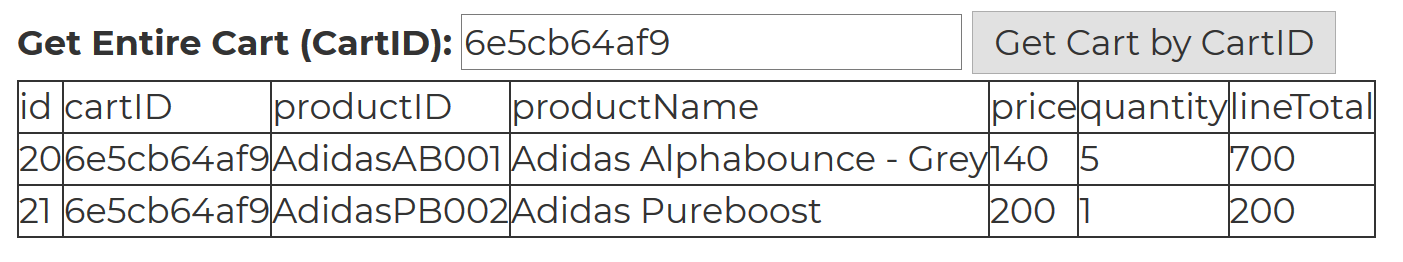
gvCartByID.DataSource = table

gvCartByID.DataBind()

End If

End Sub

* Test the button



Add the button to Get Cart using Current Browser Cookie. This button will need more code than the others.

* Add the following code to ApiClient.aspx

<!--Get Current Cart using Current Browser Cookie-->

<asp:Button ID="btnCurrentCart" runat="server" Text="Get Current Cart" />

<asp:GridView ID="gvCurrentCart" runat="server"></asp:GridView>

* Add code behind for the button

Private Async Sub btnCurrentCart\_ClickAsync(sender As Object, e As EventArgs) Handles btnCurrentCart.Click

If strCartID IsNot Nothing Then

Dim uri As String = "http://localhost:50360/api/cart/" & strCartID

Dim task = Await httpClient.GetAsync(uri)

Dim jsonString As String = Await task.Content.ReadAsStringAsync()

If task.IsSuccessStatusCode Then

Dim table As DataTable = JsonConvert.DeserializeObject(Of DataTable)(jsonString)

gvCurrentCart.DataSource = table

gvCurrentCart.DataBind()

End If

End If

End Sub

Private Sub ApiClient\_Load(sender As Object, e As EventArgs) Handles Me.Load

If (HttpContext.Current.Request.Cookies("CartID") IsNot Nothing) Then

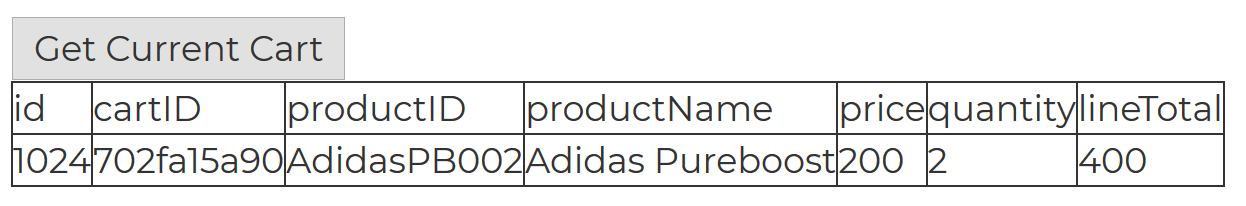
Dim CookieBack As HttpCookie = HttpContext.Current.Request.Cookies("CartID")

strCartID = CookieBack.Value

End If

End Sub

* Test the button



There are 2 subs in there, consider splitting them up for the reader loadEvent and OnclickEvent. You also need to add a global variable Dim strCartID As String put it at the top under Dim httpClient As New HttpClient.

Add the button to update cart quantity using new method PATCH.

* Add the button

<br />

<br />

<!--Change Cart Qty-->

<label>Cart Line to Update (id):</label>

<input type="text" id="tbUpdateQtyCartLineID" runat="server" />

<label style="padding-left: 10px">New Quantity:</label>

<input type="text" id="tbUpdateQtyAmt" runat="server" />

<asp:Button ID="btnUpdateCartQty" runat="server" Text="Update Quantity" />

* Add the code behind the button

Private Async Sub btnUpdateCartQty\_ClickAsync(sender As Object, e As EventArgs) Handles btnUpdateCartQty.Click

Dim myJson As String = ("[{'op': 'replace', 'path': '/quantity', 'value': '" & tbUpdateQtyAmt.Value & "'}]")

'Note [ ] in JSON'

Dim uri As String = "http://localhost:50360/api/cart/" & tbUpdateQtyCartLineID.Value

'HttpClient does not have a PatchAsync helper method but does support Patch HttpMethod so we build our own HttpRequestMethod and use SendAsync'

Dim httpPatchMethod As New HttpMethod("PATCH")

Dim request As New HttpRequestMessage(httpPatchMethod, uri)

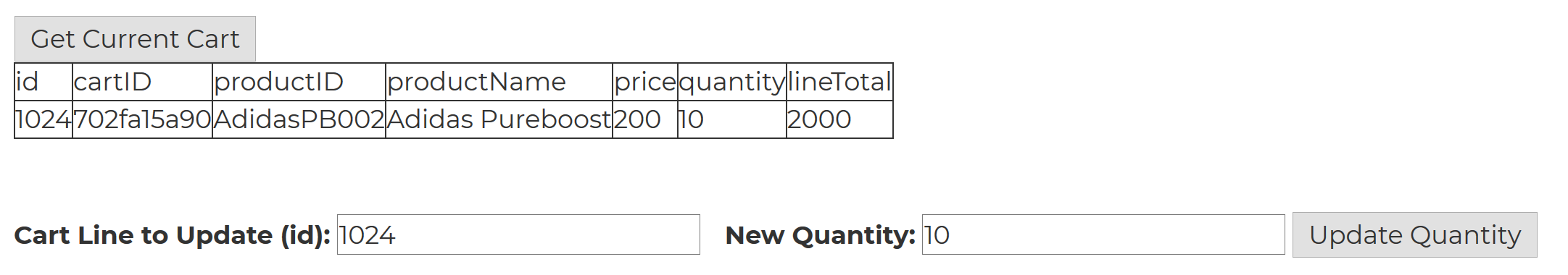
request.Content = New StringContent(myJson, Encoding.UTF8, "application/json")

Dim response = Await httpClient.SendAsync(request)

btnAllCarts\_ClickAsync(btnAllCarts, EventArgs.Empty)

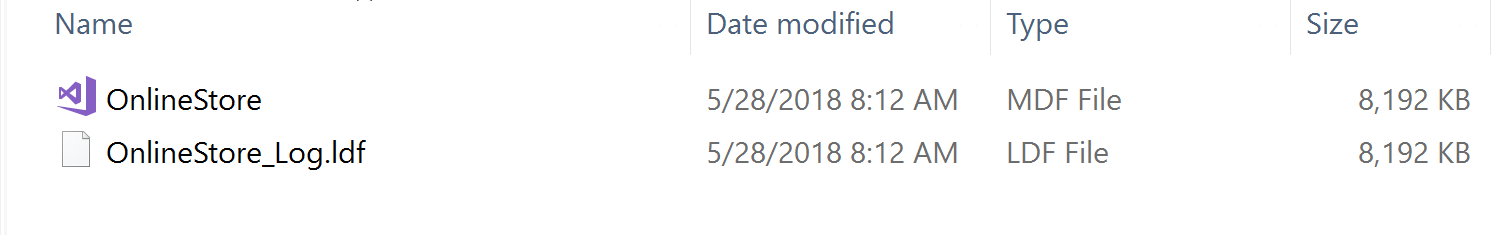
End Sub

* Test the button

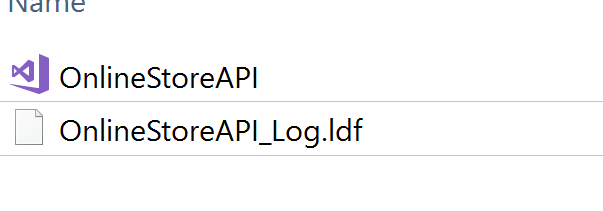


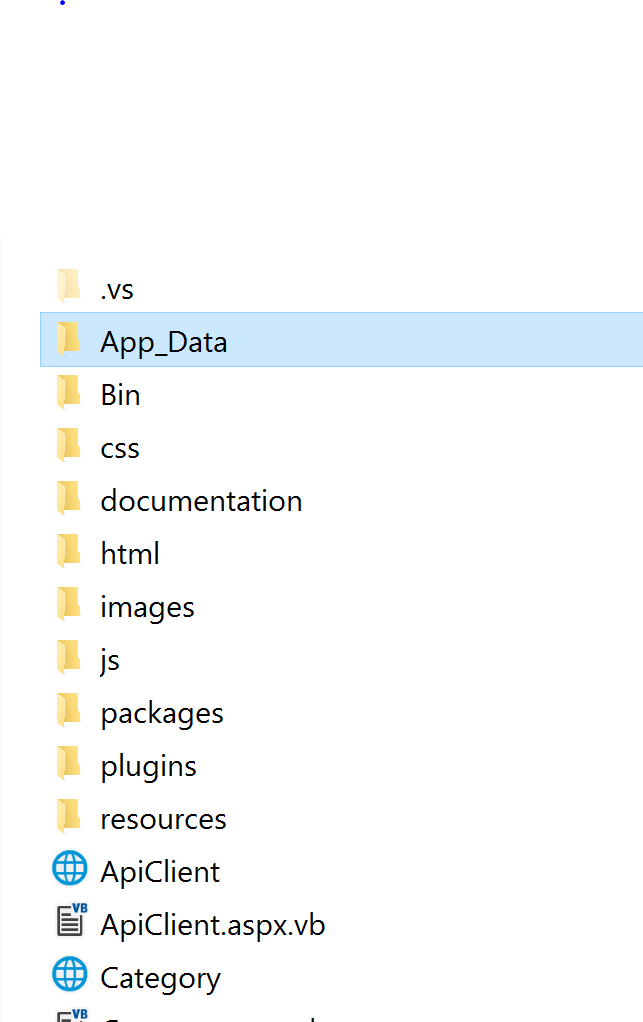
Up until now this guide has used myEcommercSite’s DB for the API, but in practical use the API uses its own database. The next section focuses on the reconfiguration of our project to have 2 databases, one for the client and one for the server. After the databases are set up we will create a function to use the JSON from the GET Cart info and import directly to our local database using OPENJSON.

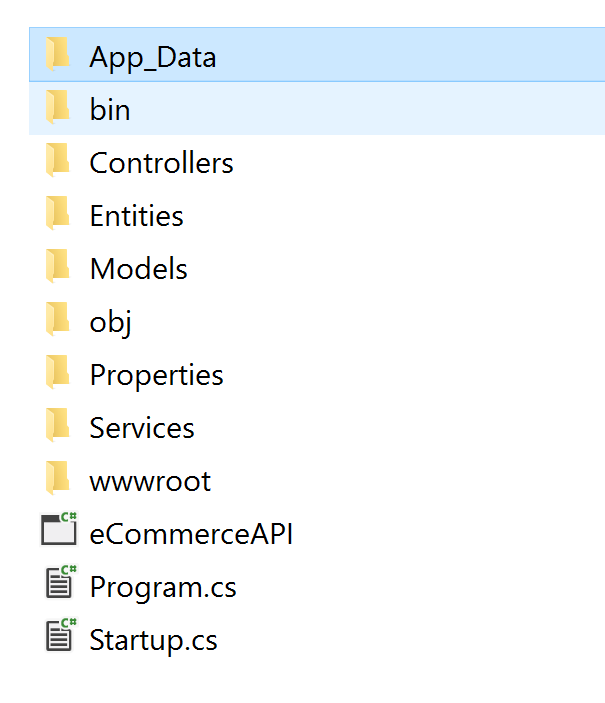
* Right click your project in Visual Studio’s Solution Explorer and click Open Folder in File Explorer to easily find it. Do this for both your API and website.
* Find your .mdf database file in your ecommerce website. It is located in the App\_Data folder inside your project folder. The database in this guide is located at C:\inetpub\wwwroot\MyEcommerceSite\App\_Data\OnlineStore.mdf. The folder will contain database file and a log file.



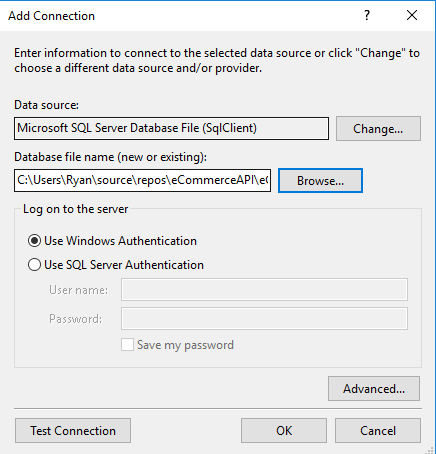
* Copy the App\_Data folder from the ecommerce website and paste it into the API.
* Open the API’s new App\_Data folder and rename the files to OnlineStoreAPI.mdf and OnlineStoreAPI\_Log.ldf.







* Now go to the Visual Studio that contains the API and go to Server Explorer > Connect to Database Icon >



* Click browse and navigate to OnlineStoreAPI.mdf > press OK
* Now the API and the website have separate databases.

Now we have to configure the database in the API to use the correct database.

* Go to the Visual Studio that contains the API project
* Solution explorer > Right click project name > Manage User Secrets
* Add a comma at the end and add another connection string called sqlConnectionAPI
* Double check the connection string syntax. It should have a double backslash for every backslash to escape. Each connection string should be on one line.
* If issues arise it is common that an incorrect connectionString is the cause.
* Again the filepaths below are for reference, yours will likely be similar.

{

"connectionStrings": {

"sqlConnectionEcomm": "Data Source=(LocalDB)\\MSSQLLocalDB;AttachDbFilename=C:\\inetpub\\wwwroot\\MyEcommerceSite\\App\_Data\\OnlineStore.mdf;Integrated Security=True",

"sqlConnectionAPI": "Data Source=(LocalDB)\\MSSQLLocalDB;AttachDbFilename=C:\\Users\\Ryan\\source\\repos\\eCommerceAPI\\eCommerceAPI\\App\_Data\\OnlineStoreAPI.mdf;Integrated Security=True"

}

}

* In the API open Startup.cs > change the variable conn configuration like below

var conn = Configuration["connectionStrings:sqlConnectionAPI"];

* Test the client and server to validate that they work with the new database. Since we just copied the database the contents will be the same. It can help to think of the API’s database as what would be a large collection of all possible products and the ecommerce store’s database as a small boutique that chooses from the larger collection what it wants to sell.

Next, we will implement code to utilize the new database:

* Add a button to import all products into website database

<br />

<br />

<!--Import All Products-->

<asp:button runat="server" id="btnImportAllProducts" text="Import All Products to Ecommerce DB" />

* Add the code behind the button
* For this we will be using OPENJSON which can import JSON directly into a SQL Server table.

Private Async Sub btnImportAllProducts\_ClickAsync(sender As Object, e As EventArgs) Handles btnImportAllProducts.Click

Dim uri As String = "http://localhost:50360/api/product"

Dim task = Await httpClient.GetAsync(uri)

Dim jsonString = Await task.Content.ReadAsStringAsync()

Dim sqlDr As SqlDataReader

Dim strSQLStatement As String

Dim cmdSQL As SqlCommand

Dim strConnectionString As String = System.Configuration.ConfigurationManager.ConnectionStrings("ConnectionStringOnlineStore").ConnectionString

Dim conn As New SqlConnection(strConnectionString)

conn.Open()

strSQLStatement = "DECLARE @json NVARCHAR(max) SET @json = N'" & jsonString & "'; INSERT INTO Product SELECT \* FROM OPENJSON(@json) WITH (productNo varchar(30), productName varchar(100), productDescription varchar(200), price numeric(7,2), subCategoryID int, featured char(1), mainCategoryID int)"

cmdSQL = New SqlCommand(strSQLStatement, conn)

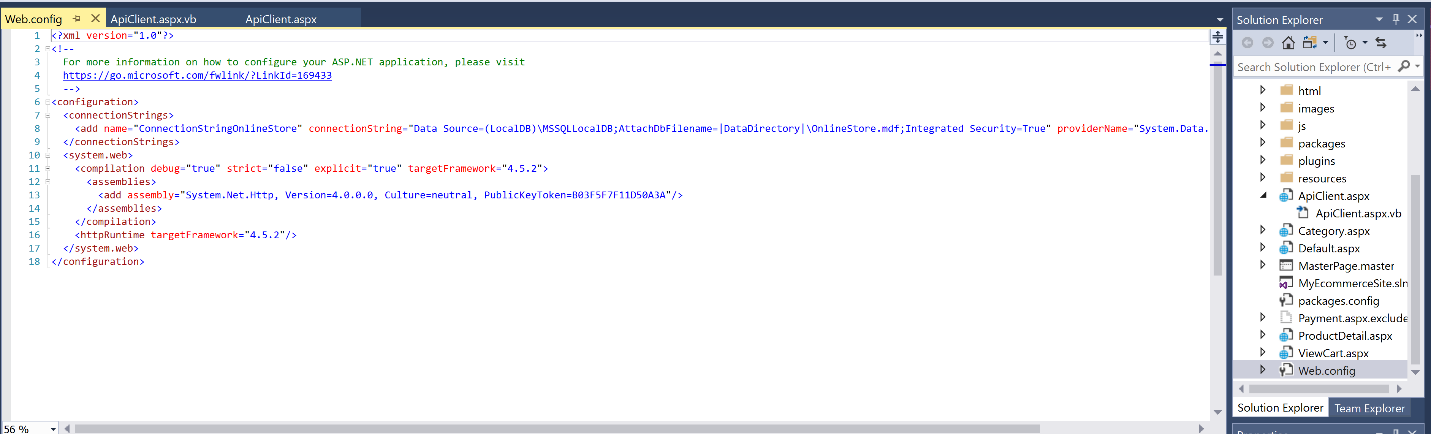
sqlDr = cmdSQL.ExecuteReader()

conn.Close()

End Sub

This will be using sqldata reader so you will need to import :

* Imports System.Data.SqlClient into ApiClient.aspx.vb
* This also uses the ConnectionString set in Web.config, which needs to be adjusted as needed



* Check your database design for strSQLStatement for data type and size

Add the button for importing a product by ID into the website database:

* Add the button

<br />

<br />

<!--Import Product by ID-->

<label>Product ID:</label>

<input type="text" id="tbImportProductID" runat="server" />

<asp:button runat="server" id="btnImportProductID" text="Import Product to Ecommerce DB" />

* Add the code behind the button

Private Async Sub btnImportProductID\_ClickAsync(sender As Object, e As EventArgs) Handles btnImportProductID.Click

Dim uri As String = "http://localhost: 50360/api/product/" & tbImportProductID.Value

Dim task = Await httpClient.GetAsync(uri)

Dim jsonString = Await task.Content.ReadAsStringAsync()

Dim sqlDr As SqlDataReader

Dim strSQLStatement As String

Dim cmdSQL As SqlCommand

Dim strConnectionString As String = System.Configuration.ConfigurationManager.ConnectionStrings("ConnectionStringOnlineStore").ConnectionString

Dim conn As New SqlConnection(strConnectionString)

conn.Open()

strSQLStatement = "DECLARE @json NVARCHAR(max) SET @json = N'" & jsonString & "'; INSERT INTO Product SELECT \* FROM OPENJSON(@json) WITH (productNo varchar(30), productName varchar(100), productDescription varchar(200), price numeric(7,2), subCategoryID int, featured char(1), mainCategoryID int)"

cmdSQL = New SqlCommand(strSQLStatement, conn)

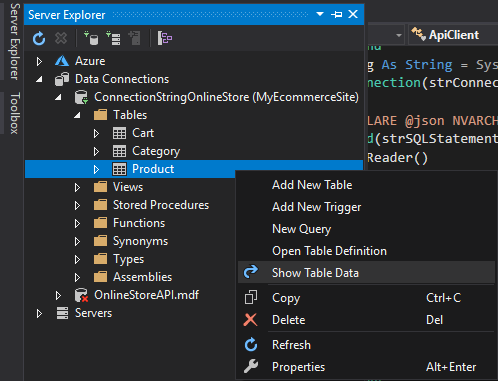
sqlDr = cmdSQL.ExecuteReader()

conn.Close()

End Sub

You can now test the importing of products into the ecommerce website database, Add products (API) and Import Products (Products get transferred to website database).

* Check your database by doing the following
* Expand your website database > Right click product > Show Table Data

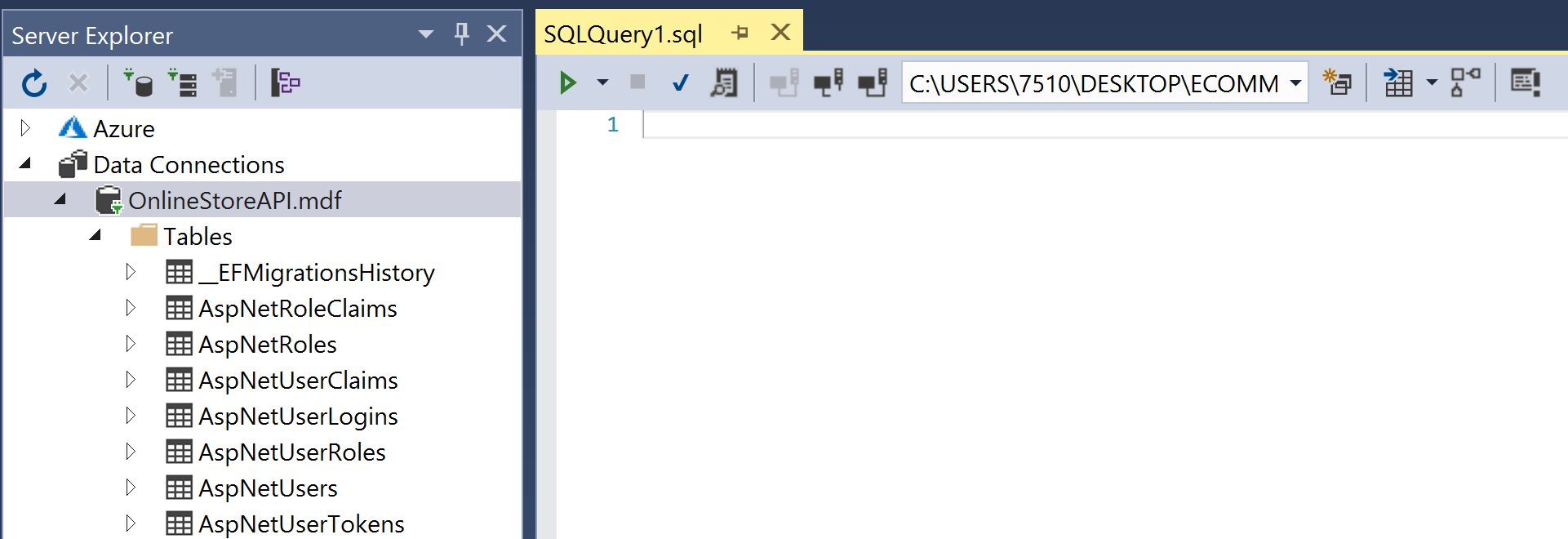


You have completed this section of the guide, the next portion will further develop onto this by adding security features.

## Security and Security Implementation

This section of the guide will demonstrate how to implement a registration form with authentication through web tokens.

* Make sure you have both the Microsoft.AspNetCore.Identity.EntityFrameworkCore and Microsoft.EntityFrameworkCore packages installed through the nuget package manager for the API. We will be using these packages to use ASP.NET Core’s Identity service to support user account management.
* Add the ASP.NET Core Identity database tables to store registered user’s information. Right click your OnlineStoreAPI.mdf database file in either the Server Explorer, or the SQL Server Object Explorer, then select “New Query” and paste the SQL query from this link <https://drive.google.com/file/d/1ij1H-EJZDYRNUmD0z6P1Hp-aED1wxjna/view?usp=sharing> into the newly generated blank sql file.
* After you’ve pasted the sql query, click the green arrow that reads “Execute”. The query will then be initiated. Then refresh the server explorer, and you should be able to see the ASP>NET tables added to your database file.



* Now create a class in the Models folder named “ApplicationUser.cs” this will create a model for our users, which will inherit from ASP.NET Core’s IdentityUser class. The code for this class can be found below:

**Path:** **/Models/ApplicationUser.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Identity;

namespace AspNetCoreWebAPI.Models

{

// Add profile data for application users by adding properties to the ApplicationUser class

public class ApplicationUser : IdentityUser

{

}

}

* We will then create another class which will inherit from the IdentityDbContext class, which is the base class for Entity Framework database context used for Identity. We will put this class in the project’s “Entities” folder, and name it “ApplicationDbContext.cs”. The content of this class is below:

**Path:** **/Entities/ApplicationDbContext.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

using AspNetCoreWebAPI.Models;

namespace AspNetCoreWebAPI.Entities

{

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>

{

public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)

: base(options)

{

}

protected override void OnModelCreating(ModelBuilder builder)

{

base.OnModelCreating(builder);

// Customize the ASP.NET Identity model and override the defaults if needed.

// For example, you can rename the ASP.NET Identity table names and more.

// Add your customizations after calling base.OnModelCreating(builder);

}

}

}

* We will now begin the implementation of JSON web tokens to be used for authentication. To do this, we need to define some key/value pairs within our secrets.json file to define our token’s secret key, the issuer, and the token’s length of validity.

{

"JwtKey": "JSON\_WEB\_TOKEN\_SECRET\_KEY\_DO\_NOT\_SHARE",

"JwtIssuer": "https://localhost:44339",

"JwtMinuteExpire": 60,

"connectionStrings": {

"sqlConnectionEcomm": "Data Source=(LocalDB)\\MSSQLLocalDB;AttachDbFilename=C:\\inetpub\\wwwroot\\MyEcommerceSite\\App\_Data\\OnlineStore.mdf;Integrated Security=True",

"sqlConnectionAPI": "Data Source=(LocalDB)\\\\MSSQLLocalDB;AttachDbFilename=C:\\Users\\Ryan\\source\\repos\\eCommerceAPI\\eCommerceAPI\\App\_Data\\OnlineStoreAPI.mdf;Integrated Security=True"

}

}

* We will now add the Identity services & JSON web token configurations to the **ConfigureServices**(IServiceCollection services) method in the **Startup.cs** class.

//Add Identity’s Database Context

services.AddDbContextPool<ApplicationDbContext>(options =>

options.UseSqlServer(conn));

//Implement Identity

services.AddIdentity<ApplicationUser, IdentityRole>()

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultTokenProviders();

//Add JSON Web Token Authentication

JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear(); // => remove default claims

services

.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(cfg =>

{

cfg.RequireHttpsMetadata = false;

cfg.SaveToken = true;

cfg.TokenValidationParameters = new TokenValidationParameters

{

//Recieve JWT config info from secrets.json

ValidIssuer = Configuration["JwtIssuer"],

ValidAudience = Configuration["JwtIssuer"],

IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(Configuration["JwtKey"])),

ClockSkew = TimeSpan.Zero // remove delay of token when expire

};

});

* To initialize our authentication configuration upon startup, we will need to add “app.UseAuthentication();” in the Configure method of the Startup.cs class

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)

{

loggerFactory.AddConsole();

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

//Initialize use of Authentication

app.UseAuthentication();

app.UseMvc();

}

* Now that we have configured Identity and our web tokens, we will now create a controller for our login & registration functionality. We will configure this to also return a web token upon a successful login/registration attempt. Create a class named “AccountController.cs” in the Controllers folder.
* To begin, we will need to include the following “using” statements so we can include some classes from our installed packages. We will also need to initialize some variables we will be using for the login/registration methods of the class.

**Path:** **/Controllers/AccountController.cs**

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

using System.IdentityModel.Tokens.Jwt;

using System.Linq;

using System.Security.Claims;

using System.Text;

using System.Threading.Tasks;

using AspNetCoreWebAPI.Models;

using Microsoft.AspNetCore.Cors;

using Microsoft.AspNetCore.Identity;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Extensions.Configuration;

using Microsoft.IdentityModel.Tokens;

namespace AspNetCoreWebAPI.Controllers

{

[Route("[controller]/[action]")]

public class AccountController : Controller

{

private readonly SignInManager<ApplicationUser> \_signInManager;

private readonly UserManager<ApplicationUser> \_userManager;

private readonly IConfiguration \_configuration;

public AccountController(

UserManager<ApplicationUser> userManager,

SignInManager<ApplicationUser> signInManager,

IConfiguration configuration

)

{

\_userManager = userManager;

\_signInManager = signInManager;

\_configuration = configuration;

}

* We will come back to the AccountController.cs, but first we will need to create our models for how we will login/register. In the Models folder, create two classes. We will name one “LoginDTO.cs” for our login. It will accept two parameters, email and password.

**Path:** **/Models/LoginDTO.cs**

using System.ComponentModel.DataAnnotations;

namespace AspNetCoreWebAPI.Models

{

public class LoginDTO

{

[Required]

public string Email { get; set; }

[Required]

public string Password { get; set; }

}

}

* Our next model will handle registration. We will name this “RegisterDTO.cs” It will also accept the two parameters; email and password.

**Path: /Models/RegisterDTO.cs**

using System.ComponentModel.DataAnnotations;

namespace AspNetCoreWebAPI.Models

{

public class RegisterDTO

{

[Required]

public string Email { get; set; }

[Required]

[StringLength(100, ErrorMessage = "PASSWORD\_MIN\_LENGTH", MinimumLength = 6)]

public string Password { get; set; }

[Required]

public string RegistrationCode { get; set; }

}

}

* Going back to the AccountController.cs class, we will add the login functionality to be accessed through POST request. It will use the LoginDTO.cs model that we just created and use the email & password from the JSON body to check the database for a valid identity user. It will then issue a JSON web token in the response if the login is successful.

//Login with JSON POST request. Email/password parameters required

[HttpPost]

public async Task<object> Login([FromBody] LoginDTO model)

{

var result = await \_signInManager.PasswordSignInAsync(model.Email, model.Password, false, false);

if (result.Succeeded)

{

var appUser = \_userManager.Users.SingleOrDefault(r => r.Email == model.Email);

return await GenerateJwtToken(model.Email, appUser);

}

throw new ApplicationException("INVALID\_LOGIN\_ATTEMPT");

}

* In the same class, we will add the user account registration function. Using the parameters outlined in the RegisterDTO.cs model we created, this function will create a new identity user if the email does not already exist in the database, and if the password requirements set by Identity are met. (1 uppercase, 1 lowercase, 1 number, 1 special character). It will then generate a JSON web token in the response if the request is successful.

[HttpPost]

public async Task<object> Register([FromBody] RegisterDTO model)

{

var user = new ApplicationUser

{

UserName = model.Email,

Email = model.Email

};

var result = await \_userManager.CreateAsync(user, model.Password);

if (result.Succeeded)

{

await \_signInManager.SignInAsync(user, false);

return await GenerateJwtToken(model.Email, user);

}

throw new ApplicationException("UNKNOWN\_ERROR");

}

* To handle the web token generation, we will implement our last function into the AccountController class. This will create claims associated with a given user and create the web token by signing the “JwtKey” (defined earlier in secrets.json) with the HMACSHA256 algorithm.

private async Task<object> GenerateJwtToken(string email, IdentityUser user)

{

var claims = new List<Claim>

{

new Claim(JwtRegisteredClaimNames.Sub, email),

new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),

new Claim(ClaimTypes.NameIdentifier, user.Id)

};

var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_configuration["JwtKey"]));

var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

var expires = DateTime.Now.AddMinutes(Convert.ToDouble(\_configuration["JwtMinuteExpire"]));

var token = new JwtSecurityToken(

\_configuration["JwtIssuer"],

\_configuration["JwtIssuer"],

claims,

expires: expires,

signingCredentials: creds

);

return new JwtSecurityTokenHandler().WriteToken(token);

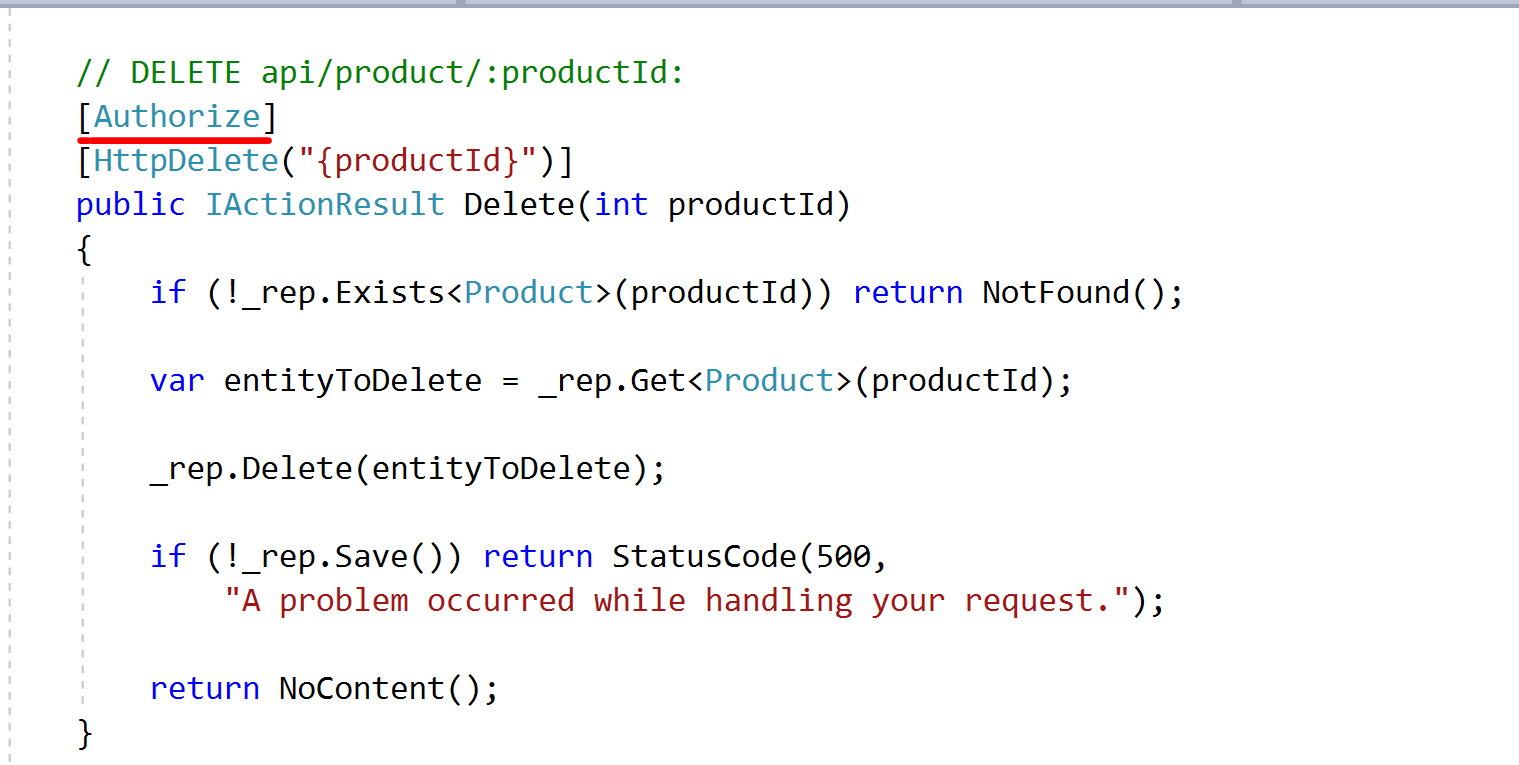
}

* Now, to test the implementation of our user account system, we can issue a POST request to the /account/register & /account/login endpoints through postman. Identity includes password strength requirements, so we need to create a password with 1 lowercase, 1 uppercase, 1 number, and 1 special character.

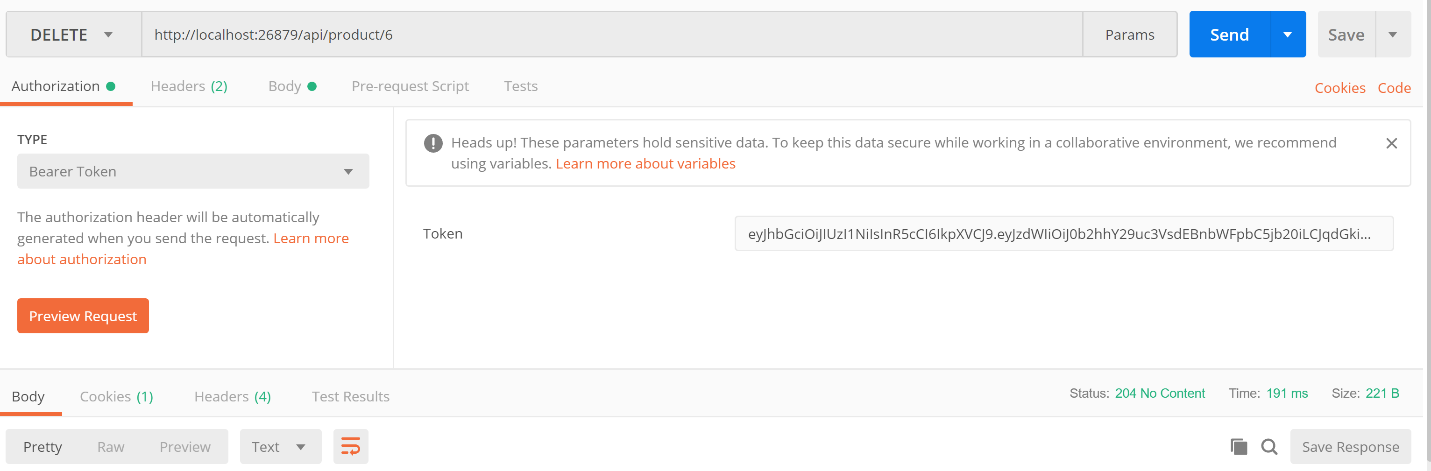


Note: Do not add the registrationCode, that will be implemented within the next steps.

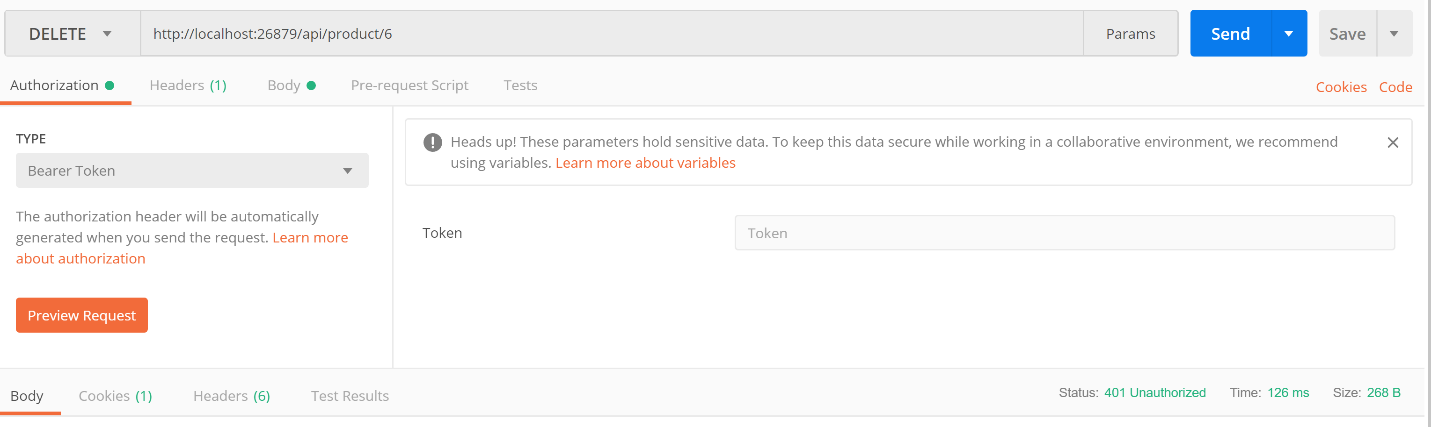
* As a result, we should see our JSON web token being issued in the response. We will now add the [Authorize] attribute above the API methods that need authorization. We will add this above the function responsible for deleting products through our API.



* Now, we will need to issue a valid web token in the Authorization header of our request when we try and delete products from our API.



* We can do this using Postman. First login to your API with valid credentials and copy the token from the response. In Postman’s “Authorization” tab, choose type “Bearer token”, and paste in your token as the value. Then, when you issue the delete request, you should see the “204 No Content” response code, indicating a successful request.
* When you issue the same request with an invalid/expired token or no token at all, you should see a “401 Unauthorized” response code.



Convert to using SSL

Use SSL for point to point security

MyEcommerceAPI > Properties > launchSettings.json

Change "sslPort": 0

Change the the port you want to use. We will use 44339 "sslPort": 44339

The endpoint will now be reachable through https like so <https://localhost:44339/account/register>

Before they all used http i.e. <http://localhost:50360/api/product>

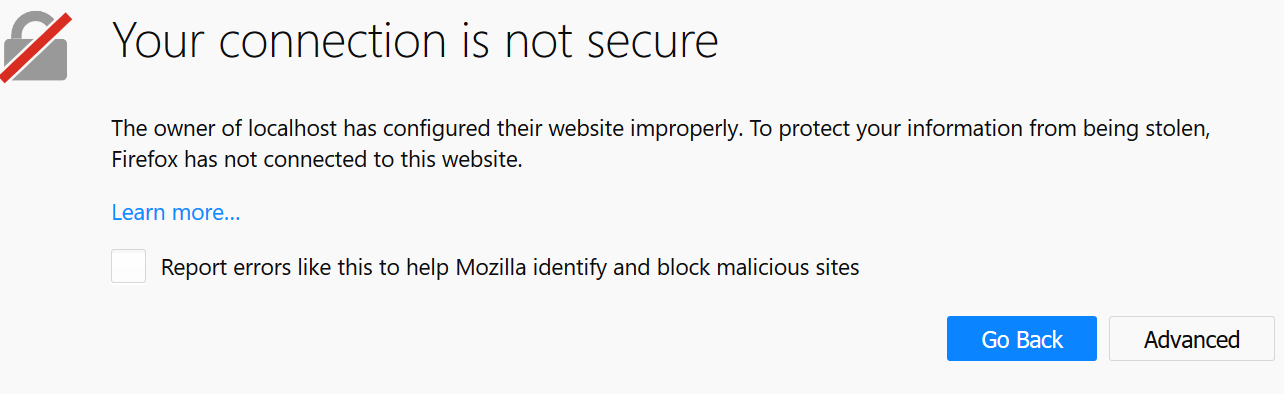
Change all the API use of http:// in your client site to https:// and change the port to match new port. This will mostly occur in ApiClient.aspx.vb

For example <http://localhost:50360/api/product> would become <https://localhost:44339/api/product>

To use SSL in POSTMAN you’ll need to turn off SSL certificate verification. This option is accessed through Postman>File>Settings>SSL certificate verification OFF

Remember to change requests URIs to <https://localhost:44339> as well.

When running the API through IIS your browser will also prompt with an Insecure Connection page stating “Your connection is not secure” like below.

To correct this you need to add an exception security certificate.

* Click Advanced
* Add Exception > Confirm Security Exception

## Add Registration Code

* We also will include a “registration code” so that not anyone can get this access to the secured parts of the API.
* Change RegisterDTO to include another parameter besides Email and Password

[Required]

public string RegistrationCode { get; set; }

* Change AccountController to check for RegistrationCode before creating an account.

[HttpPost]

public async Task<object> Register([FromBody] RegisterDTO model)

{

var myRegistrationCode = "ECOMMERCE";

if (model.RegistrationCode == myRegistrationCode)

{

var user = new ApplicationUser

{

UserName = model.Email,

Email = model.Email

};

var result = await \_userManager.CreateAsync(user, model.Password);

if (result.Succeeded)

{

await \_signInManager.SignInAsync(user, false);

return await GenerateJwtToken(model.Email, user);

}

throw new ApplicationException("UNKNOWN\_ERROR");

}

else return BadRequest("Invalid Registration Code");

}

Registration Page

Normally a registration page for the API would not be on the client site, but for brevity we will include it here. Normally the API service provider would handle the registration on their own site.

* MyEcommerceSite>Add>Add New Item>Web Form

<%@ Page Async="true" Title="" Language="VB" MasterPageFile="~/MasterPage.master" AutoEventWireup="false" CodeFile="Register.aspx.vb" Inherits="Account\_Register" %>

<asp:Content ID="Content1" ContentPlaceHolderID="head" Runat="Server">

</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="ContentPlaceHolder1" Runat="Server">

<div class="col-md-10 col-md-offset-1">

<div class="well">

<!-- This is the table to capture text fields -->

<table class="table table-bordered">

<thead>

<tr class="success">

<td colspan="2">

New User Registration

</td>

</tr>

</thead>

<tbody>

<tr>

<td>Registration Code</td>

<td>

<input type="text" id="txtCode" placeholder="Registration Code" runat="server"/>

</td>

</tr>

<tr>

<td>Email</td>

<td>

<input type="text" id="txtEmail" placeholder="Email" runat="server"/>

</td>

</tr>

<tr>

<td>Password</td>

<td>

<input type="password" id="txtPassword" placeholder="Password" runat="server"/>

</td>

</tr>

<tr>

<td>Confirm Password</td>

<td>

<input type="password" id="txtConfirmPassword" placeholder="Confirm Password" runat="server"/>

</td>

</tr>

<tr >

<td>

<asp:Button ID="btnRegister" runat="server" Text="Register" />

</td>

</tr>

</tbody>

</table>

<asp:Label ID="lblRegisterResult" runat="server" Font-Bold="true" Text="" Visible="false" CssClass="alignCenter"></asp:Label>

</div>

</div>

</asp:Content>

* Registration.aspx.vb Performs validation

Imports System.Data

Imports System.Net.Http

Imports Newtonsoft.Json

Imports System.Runtime.CompilerServices

Imports System.Net.Http.Headers

Imports System.Drawing

Imports System.Data.SqlClient

Imports System.Text.RegularExpressions

Partial Class Account\_Register

Inherits System.Web.UI.Page

Dim httpClient As New HttpClient

Private sError As String

Private Sub Account\_Register\_Load(sender As Object, e As EventArgs) Handles Me.Load

End Sub

Private Async Sub btnRegister\_Click(sender As Object, e As EventArgs) Handles btnRegister.Click

If ValidatePassword(txtPassword.Value) = True Then

If txtPassword.Value = txtConfirmPassword.Value Then

Dim myJson As String

myJson = ("{'email': '" & txtEmail.Value & "', 'password': '" & txtPassword.Value & "', 'RegistrationCode': '" & txtCode.Value.ToUpper & "'}")

Dim uri As String = "https://localhost:44339/account/register"

Dim task = Await httpClient.PostAsync(uri, New StringContent(myJson, Encoding.UTF8, "application/json"))

Dim jwtToken As String

If task.IsSuccessStatusCode Then

jwtToken = Await task.Content.ReadAsStringAsync()

lblRegisterResult.Text = "Successful Registration"

lblRegisterResult.ForeColor = Color.Green

lblRegisterResult.Visible = True

Dim tokenCookie As New HttpCookie("JwtCookie")

tokenCookie.HttpOnly = True

'tokenCookie.Secure = True //use if HTTPS for added security

tokenCookie("JWT") = jwtToken

tokenCookie.Expires = Now.AddHours(1)

Response.Cookies.Add(tokenCookie)

Else

lblRegisterResult.Text = "Incorrect Registration Code"

lblRegisterResult.ForeColor = Color.Red

lblRegisterResult.Visible = True

End If

Else

lblRegisterResult.Text = "Passwords don't match"

lblRegisterResult.ForeColor = Color.Red

lblRegisterResult.Visible = True

End If

Else

lblRegisterResult.Text = "Password needs to be at least 6 characters with 1 number, 1 upper case, 1 lower case, and 1 special character"

lblRegisterResult.ForeColor = Color.Red

lblRegisterResult.Visible = True

End If

End Sub

Function ValidatePassword(ByVal pwd As String,

Optional ByVal minLength As Integer = 6,

Optional ByVal numUpper As Integer = 1,

Optional ByVal numLower As Integer = 1,

Optional ByVal numNumbers As Integer = 1,

Optional ByVal numSpecial As Integer = 1) As Boolean

' Replace [A-Z] with \p{Lu}, to allow for Unicode uppercase letters.

Dim upper As New System.Text.RegularExpressions.Regex("[A-Z]")

Dim lower As New System.Text.RegularExpressions.Regex("[a-z]")

Dim number As New System.Text.RegularExpressions.Regex("[0-9]")

' Special is "none of the above".

Dim special As New System.Text.RegularExpressions.Regex("[^a-zA-Z0-9]")

' Check the length.

If Len(pwd) < minLength Then Return False

' Check for minimum number of occurrences.

If upper.Matches(pwd).Count < numUpper Then Return False

If lower.Matches(pwd).Count < numLower Then Return False

If number.Matches(pwd).Count < numNumbers Then Return False

If special.Matches(pwd).Count < numSpecial Then Return False

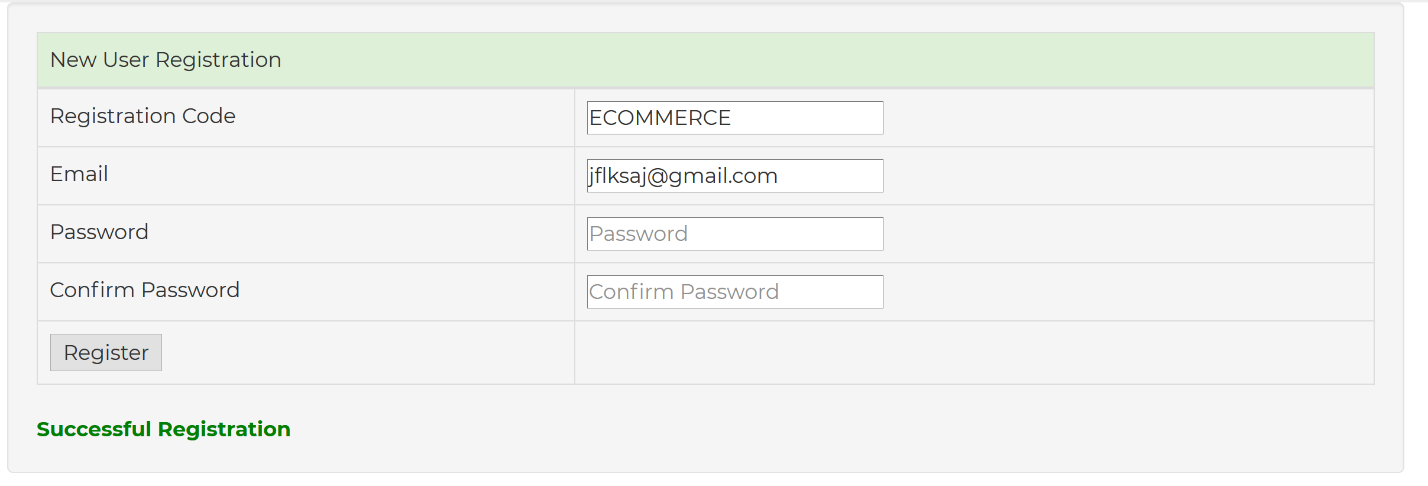
' Passed all checks.

Return True

End Function

End Class

* Test the registration page. The Registration Code is ECOMMERCE. A success message should appear



Implementing JWT authentication into our client site

We will create the ability to login and persist our authentication token.

* Add login email and password form to ApiClient.aspx
* Also label for login success/fail and button for logout
* Add a hyperlink for the registration page too

<br />

<!--API Login Credentials-->

<div id="loginDiv" runat="server" class="alignCenter">

<label>Email:</label>

<input type="text" id="tbEmail" runat="server" />

<label>Password:</label>

<input type="password" id="tbPassword" runat="server" />

<asp:Button ID="btnApiLogin" runat="server" Text="Login" />

</div>

<a href="Register.aspx">Register for API Access</a>

<asp:Label ID="lblLoginResult" runat="server" Font-Bold="true" Text="" Visible="false" CssClass="alignCenter"></asp:Label>

<asp:Button ID="btnApiLogout" runat="server" Text="Logout" Visible="false"/>

* Add logic to code behind to handle login ApiClient.aspx.vb.
* Login Button | This logs in and places the JWT into a cookie to persist until cookie expires or logout.

Private Async Sub btnApiLogin\_ClickAsync(sender As Object, e As EventArgs) Handles btnApiLogin.Click

Dim myJson As String

myJson = ("{'email': '" & tbEmail.Value & "', 'password': '" & tbPassword.Value & "'}")

Dim uri As String = "https://localhost:44339/account/login"

Dim task = Await httpClient.PostAsync(uri, New StringContent(myJson, Encoding.UTF8, "application/json"))

Dim jwtToken As String

If task.IsSuccessStatusCode Then

jwtToken = Await task.Content.ReadAsStringAsync()

lblLoginResult.Text = "Successful Login"

lblLoginResult.ForeColor = Color.Green

lblLoginResult.Visible = True

loginDiv.Visible = False

btnApiLogout.Visible = True

Dim tokenCookie As New HttpCookie("JwtCookie")

tokenCookie.HttpOnly = True

'tokenCookie.Secure = True //use if HTTPS for added security. Current ApiClient page is HTTP

tokenCookie("JWT") = jwtToken

tokenCookie.Expires = Now.AddHours(1)

Response.Cookies.Add(tokenCookie)

Else

lblLoginResult.Text = "Failed Login"

lblLoginResult.ForeColor = Color.Red

lblLoginResult.Visible = True

End If

End Sub

* Above you can see the use of HttpOnly and Secure to secure our cookie. Secure is commented out because our ApiClient page is HTTP and we did not add SSL for the client. If the page was made HTTPS then it could utilize .Secure = True.

Logout Button | logs out and makes login fields visible

Private Sub btnApiLogout\_Click(sender As Object, e As EventArgs) Handles btnApiLogout.Click

If (Not Request.Cookies("JwtCookie") Is Nothing) Then

Dim tokenCookie As HttpCookie

tokenCookie = New HttpCookie("JwtCookie")

tokenCookie.Expires = DateTime.Now.AddDays(-1D)

Response.Cookies.Add(tokenCookie)

lblLoginResult.Visible = False

loginDiv.Visible = True

btnApiLogout.Visible = False

End If

End Sub

* Add another if statement to the ApiClient\_Load event that already exists. To control which login/logout buttons are available depending on cookie status

Private Sub ApiClient\_Load(sender As Object, e As EventArgs) Handles Me.Load

If (HttpContext.Current.Request.Cookies("CartID") IsNot Nothing) Then

Dim CookieBack As HttpCookie = HttpContext.Current.Request.Cookies("CartID")

strCartID = CookieBack.Value

End If

If (HttpContext.Current.Request.Cookies("JwtCookie") IsNot Nothing) Then

loginDiv.Visible = False

btnApiLogout.Visible = True

End If

End Sub

* In the ProductController and CartController Make sure to add [Authorize] before any API methods that should require authentication Like deleting and updating. (Stuff that is not available to the public). Generally GET requests are made public and the other verbs require authorization, similar to a read-only.

// DELETE api/cart/:id:

**[Authorize]**

[HttpDelete("{id}")]

public IActionResult Delete(int id)

{

if (!\_rep.Exists<Cart>(id)) return NotFound();

var entityToDelete = \_rep.Get<Cart>(id);

\_rep.Delete(entityToDelete);

if (!\_rep.Save()) return StatusCode(500,

"A problem occurred while handling your request.");

return NoContent();

}

* Now we need to include a JWT Bearer Authorization token for API requests sent from the client that require Authorization.

Go to ApiClient.aspx.vb and add this line of code to any requests that use an [Authorize] tag on its API method.

httpClient.DefaultRequestHeaders.Authorization = New AuthenticationHeaderValue("Bearer", getToken())

* Example

Private Async Sub btnCreateProduct\_ClickAsync(sender As Object, e As EventArgs) Handles btnCreateProduct.Click

Dim myJson As String = ("{'ProductNo': '" & tbProductNo.Value & "', 'ProductName': '" & tbProductName.Value & "', 'ProductDescription': '" \_

& tbProductDescription.Value & "', 'Price': '" & tbPrice.Value & "', 'SubCategoryID': '" & tbSubCategoryID.Value \_

& "', 'Featured': '" & tbFeatured.Value & "', 'MainCategoryID': '" & tbMainCategoryID.Value & "'}")

**httpClient.DefaultRequestHeaders.Authorization = New AuthenticationHeaderValue("Bearer", getToken())**

Dim uri As String = "https://localhost:44338/api/product/"

Dim response = Await httpClient.PostAsync(uri, New StringContent(myJson, Encoding.UTF8, "application/json"))

btnAllProducts\_ClickAsync(btnAllProducts, EventArgs.Empty)

End Sub

* Also include the function getToken() in the ApiClient.aspx.vb codebehind that will grab the token from the cookie.

Function getToken() As String

Dim jwtToken As String

If (Request.Cookies("JwtCookie") IsNot Nothing) Then

If (Request.Cookies("JwtCookie")("JWT") IsNot Nothing) Then

jwtToken = Request.Cookies("JwtCookie")("JWT")

Return jwtToken

End If

End If

Return Nothing

End Function

* Test web client authorization. Try your unsecured GET endpoints without logging in and try your secured endpoints like delete after logging in.

