ASP.NET Core Activity 5

Part 1 REST Services

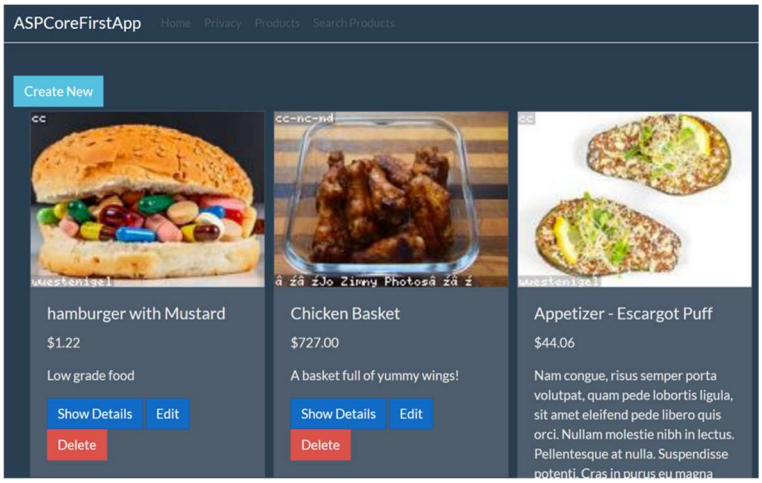
Goal:

In this activity, you will learn how to build a REST(ful) Service in ASP.NET

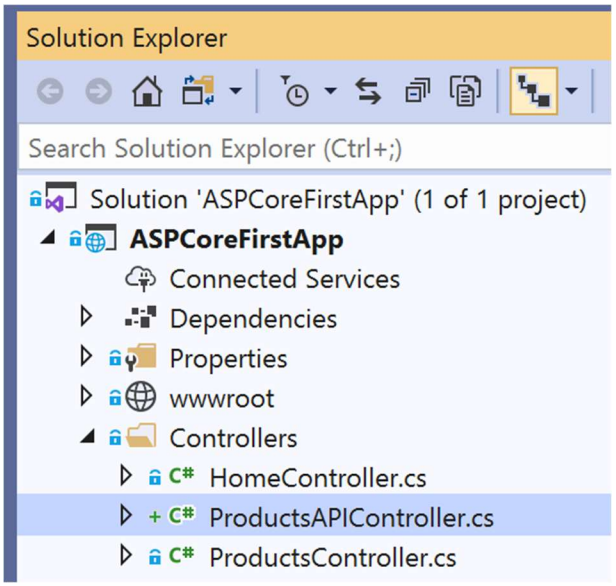
You will need the code from a previous activity and you will use your Browser and **Postman** to test your REST(ful) Services.

Create a Rest Controller

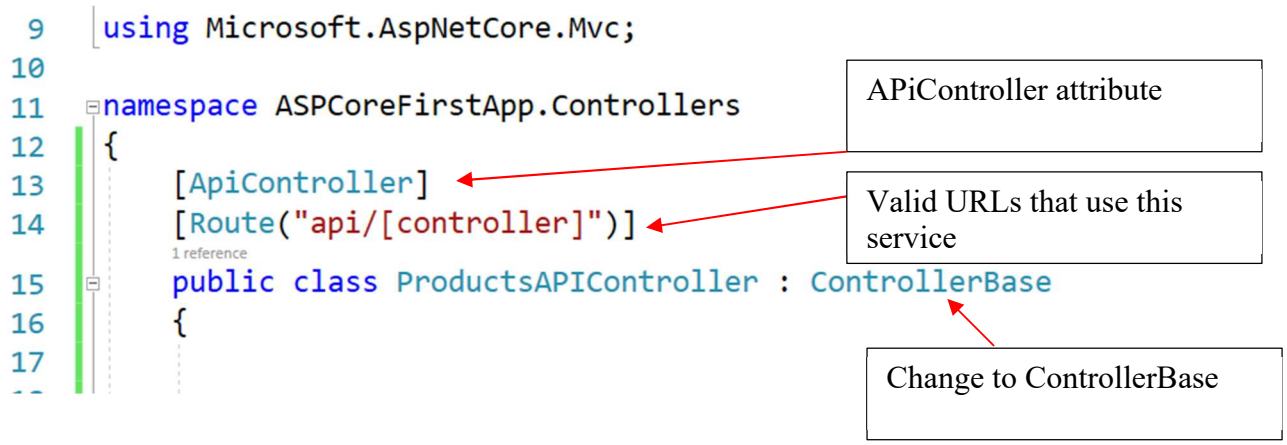
1. Open the **Products application** that was created in a previous lesson. We will extend the application to use the same data in a REST API format.



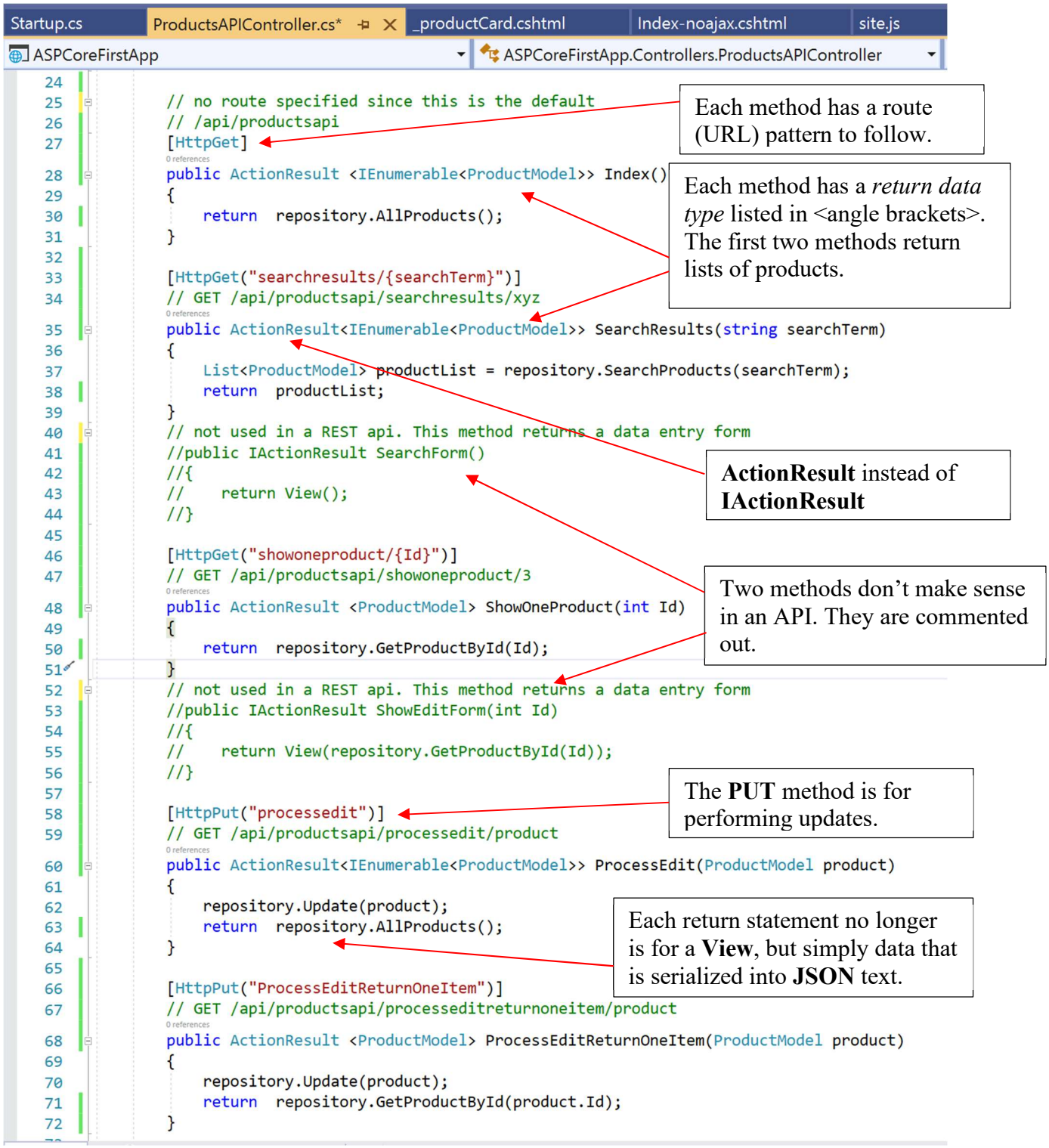
2. Duplicate (copy and paste) the **ProductsController**.
3. Rename the new controller to **ProductsAPIController**.



4. Change the interface that is implemented in this class from **Controller** to **ControllerBase**. The **Controller** class derives from **ControllerBase** and adds some members that are needed to support Views. In a REST API, which we are building here, the controller does not returns Views. It is more appropriate to implement the **ControllerBase** interface instead of the **Controller** interface.
5. Add a new attribute to the class, **[ApiController]**. This will help the controller interpret URLs and parameters correctly, as well as validate data models. Validation errors automatically trigger an HTTP 400 response.
6. Add another attribute, **[Route("api/[controller]")]**. This will indicate which URLs the service will respond to. In this case, http://localhost:4423/api/productsapi will lead to the default end point (index).

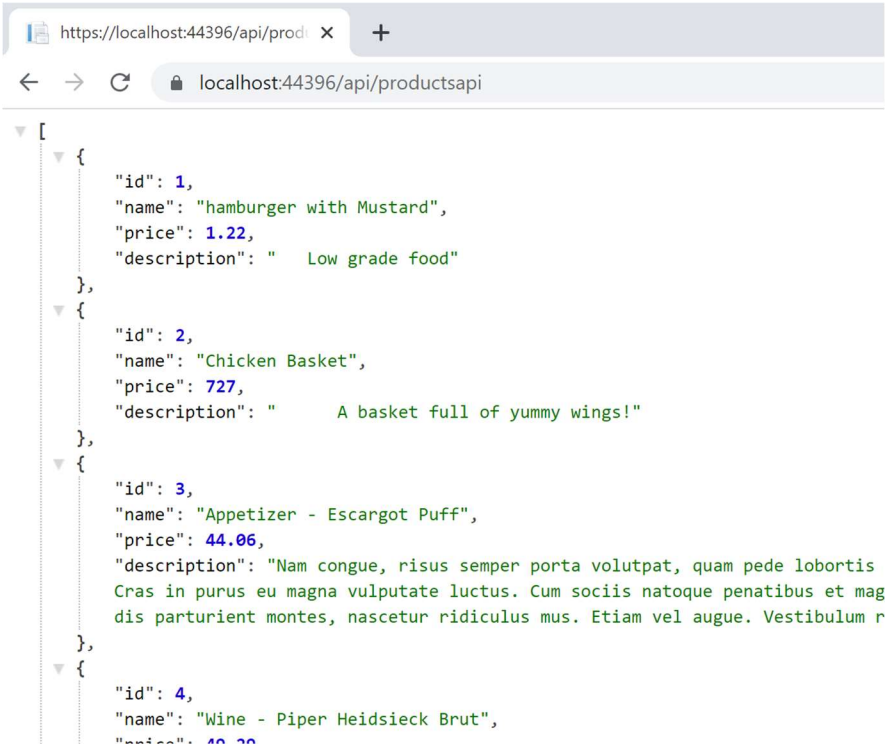


7. Make the following changes to the methods in the controller.

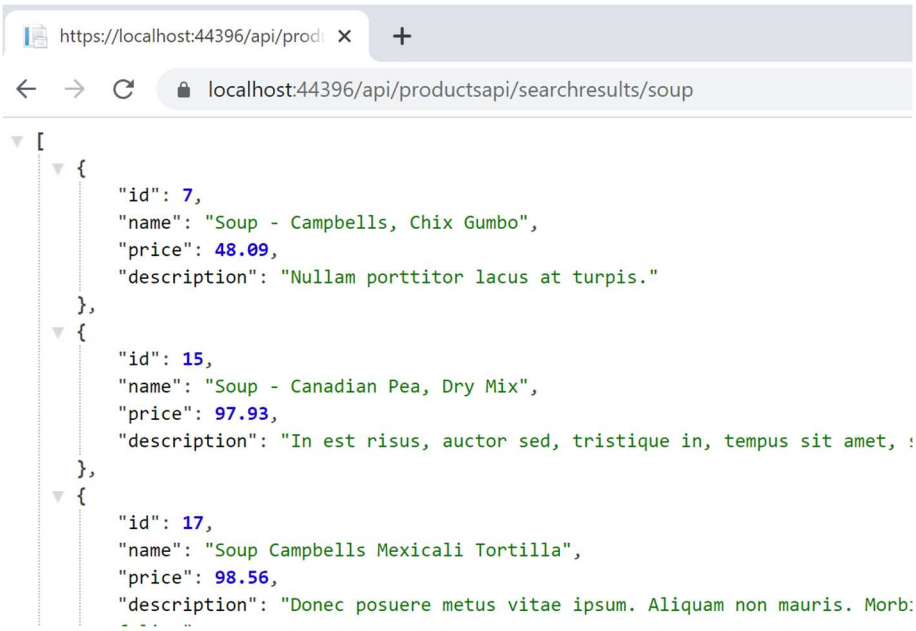


Testing.

1. Get all the products in the database. **localhost:44396/api/productsapi** (default route)

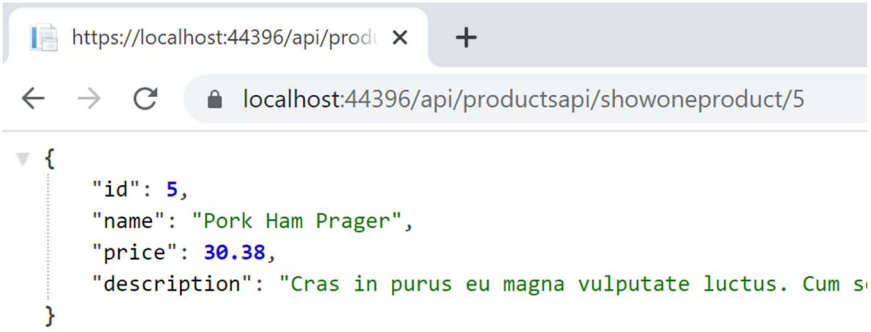


2. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.
3. Search for products in the database. **localhost:44396/api/productsapi/searchresults/soup**



4. Take a screenshot of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

5. Get one product. **localhost:44396/api/productsapi/showoneproduct/5**

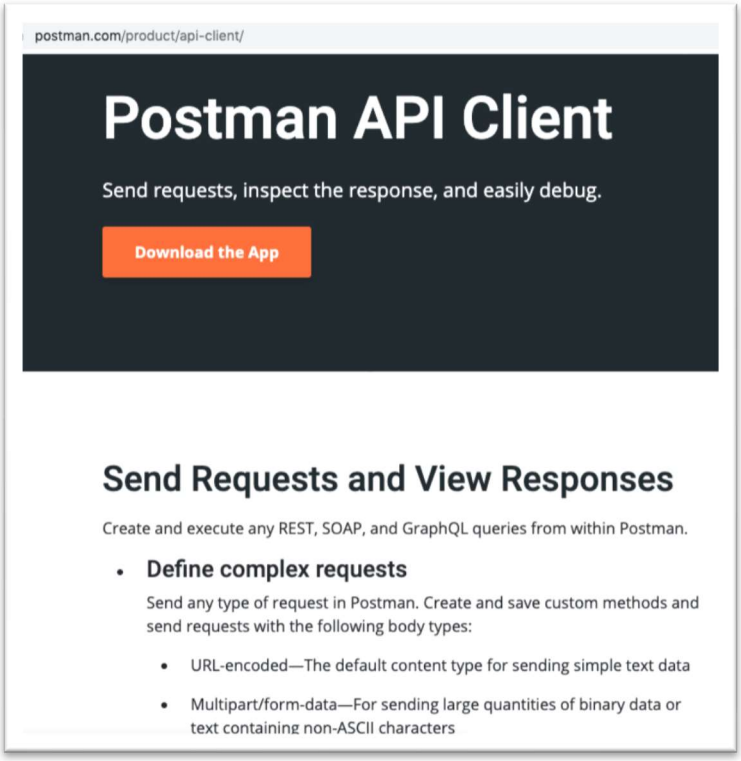


6. Take a screenshot of the app at this stage. Paste it into a Word document and caption the image with a brief explanation of what you just demonstrated.

Postman

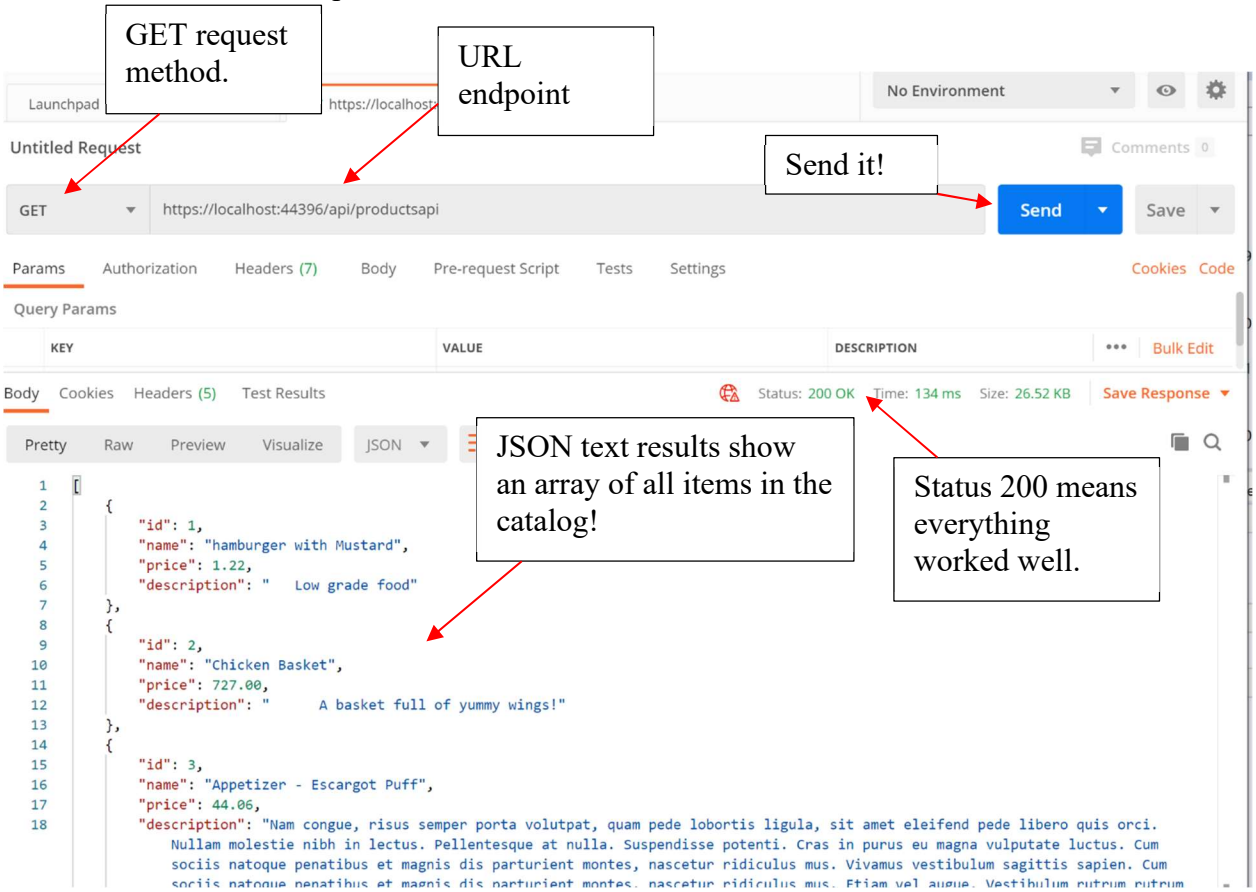
Postman is a developer tool for testing REST APIs like the one we are building.

Since APIs lack a GUI, API testing is performed at the message layer and requires a different type of tool than a web browser. A web browser is able to perform GET requests quite easily. Performing a POST form submit requires a complete form to be implemented. PostMan allows us to experiment and test requests much easier than a web browser.



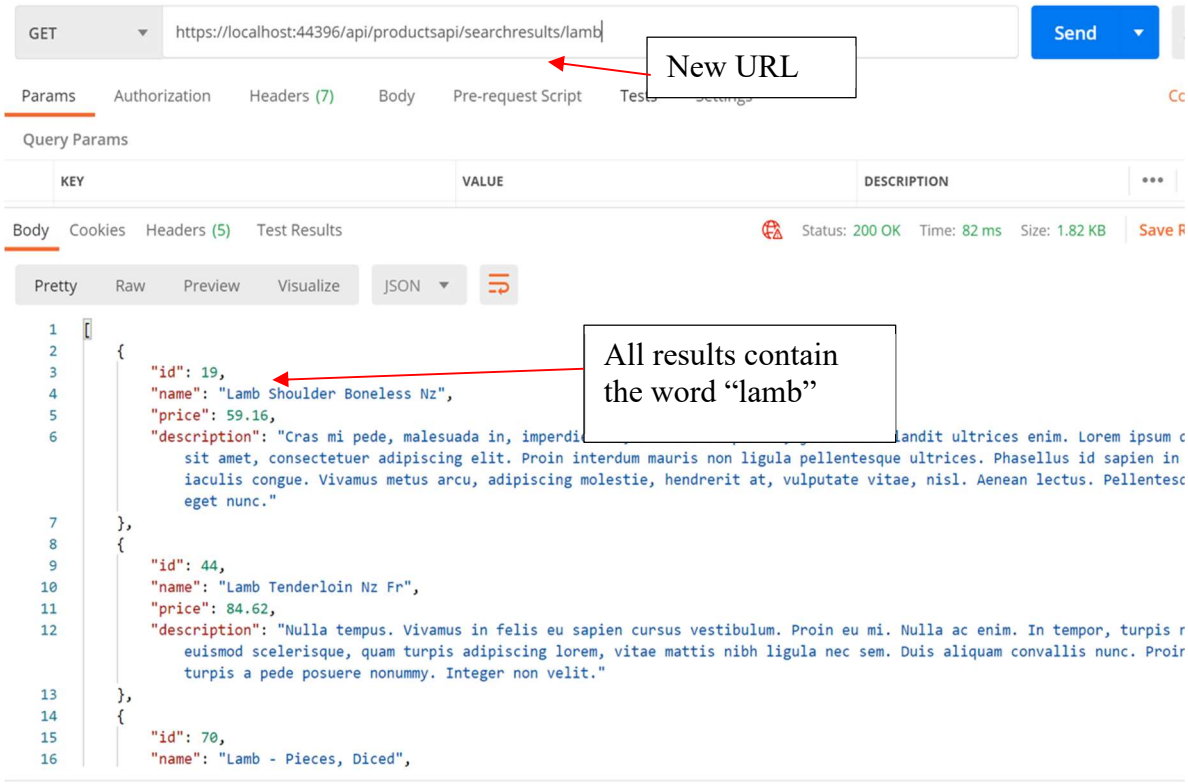
1. Ensure that **PostMan is installed** on your computer.
2. Perform a test request with PostMan on each of the API endpoints: **index**, **searchresults**, and **showoneproduct**.

This shows the **index** endpoint. Index is the default method so no "index" is needed in the URL.



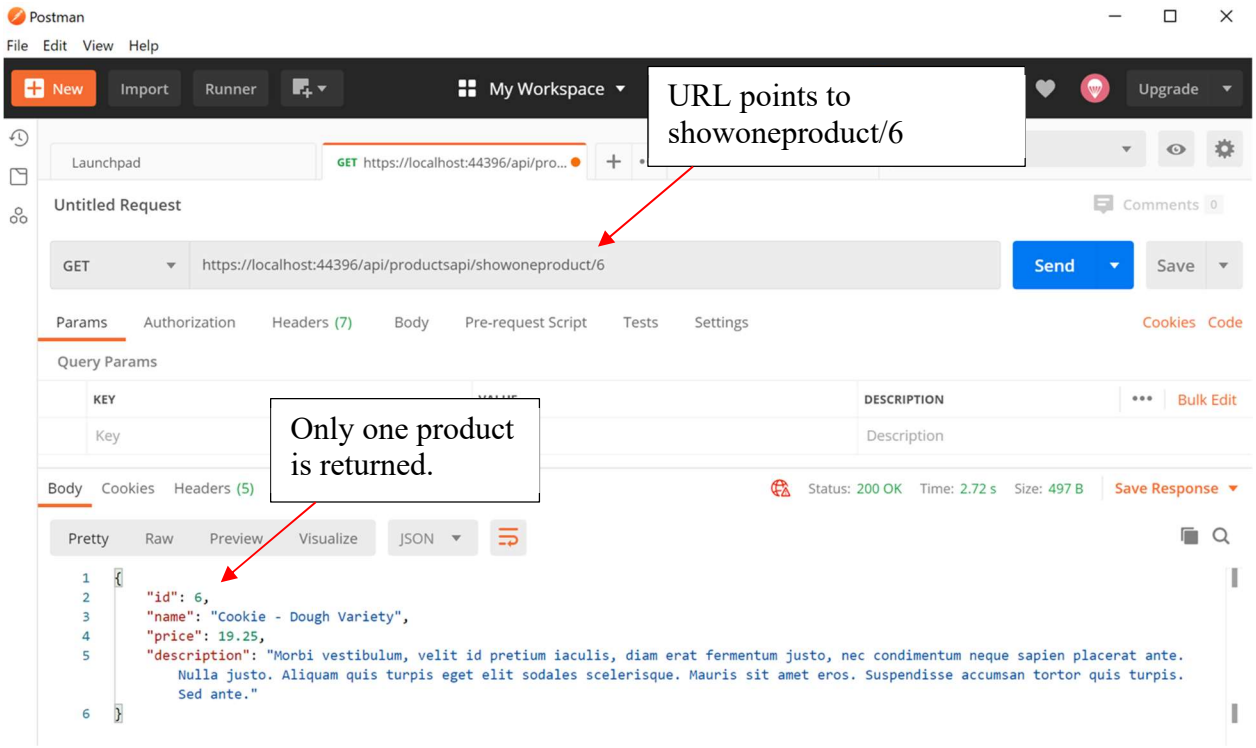
3. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

This shows the **searchresults** endpoint for “lamb.”



4. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

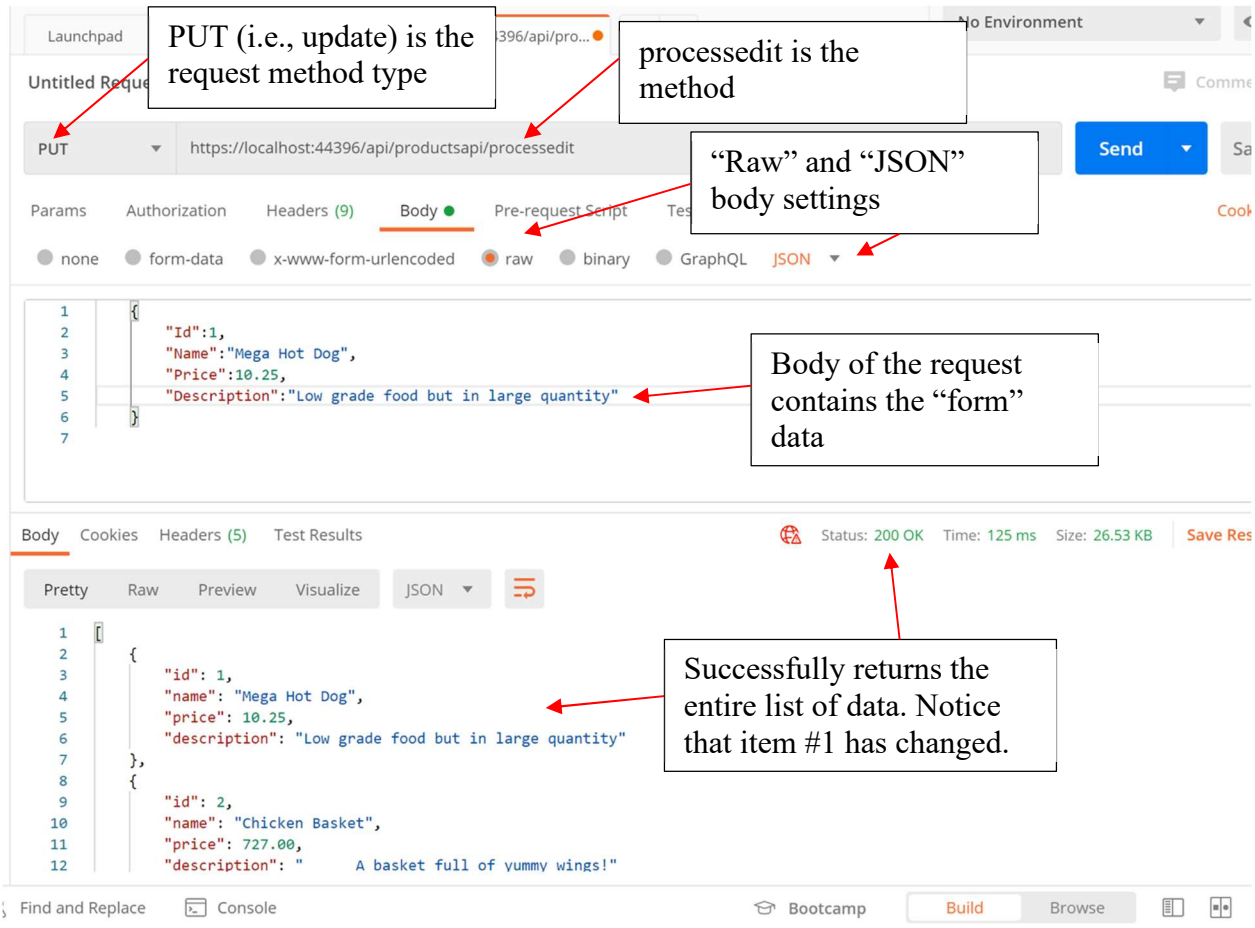
This shows the **showoneproduct** endpoint for item #6.



5. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

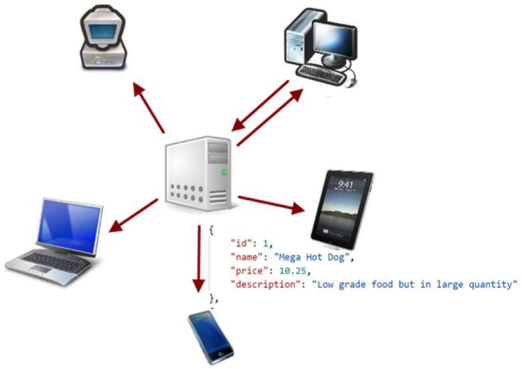
So far these are identical results to what we saw in the browser.

6. Perform a PUT command that updates the contents of one of the records in the database. In this example, the first item is changed to a “Mega Hot Dog.”



7. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

About REST APIs in relation to software development



REST APIs are the backend

As you can see from the examples above, the exact same data is being served by the API service as was via the Views and Razor pages. The only difference is the format. The REST API is a true "Back End" service, ready to connect to every imaginable type of application.

Picture how a user connects to an airline: the company **webpage**, the **mobile** app, **email**, and **texting** servers all connect to the same central set of data about flights, tickets, and related records. The central server is probably running a REST API.

Servers can connect to other servers. A collection of APIs can form a web of services called "Microservices." It is possible to build an API to serve inside the application.

Front End Options

Most of the examples in this course have used Razor forms to create webpages. Instead of building all of forms for the app using Razor, we could build the entire user interface with HTML and JavaScript, using AJAX calls to communicate with the API service. In fact, a popular method for building applications is to build a REST API using ASP.NET as the back end and React, Vue, or Angular for the front end. The code for the API and code for the web client can reside on the same server or on different servers. Mobile apps require some kind of server back end to handle their data. An ASP.NET server is a good option for the back end.

In 2020, Microsoft released Blazor, a front-end client technology that uses the C# language, but runs on the client browser much like React and JavaScript. All of these designs require a REST service on the back end.

Other Back End Options

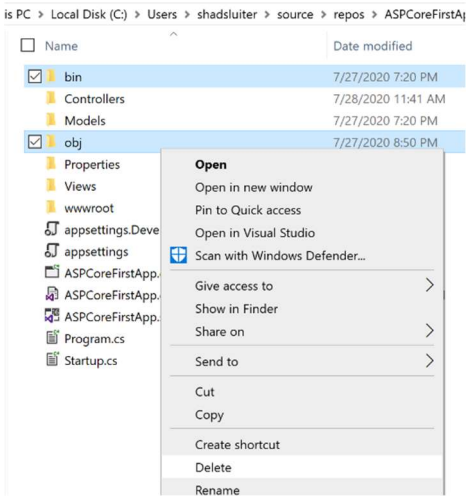
ASP.NET Core is only one of many options for building a REST API. The JSON text format is an open standard used by all technology companies. Some common frameworks for building REST APIs include:

- Java Enterprise
- Java Spring and Spring Boot

- PHP Laravel
- JavaScript NodeJS
- Python Flask and Django
- Ruby on Rails

Each of these is capable of producing indistinguishable JSON text in REST APIs to the clients. A company can theoretically swap out one back end service for another and the front end clients would never know the difference.

Deliverables:



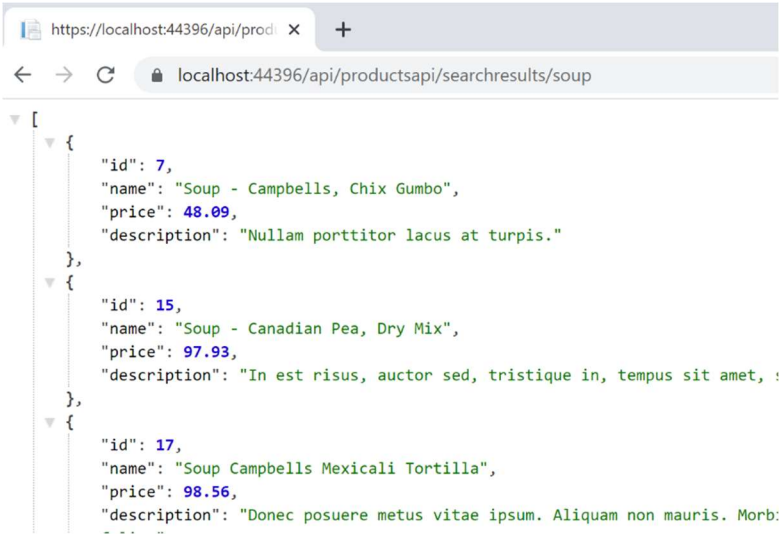
1. This activity has multiple parts. Complete all parts before submitting.
2. Submit a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
4. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.

Part 2 Data Transfer Objects (DTO)

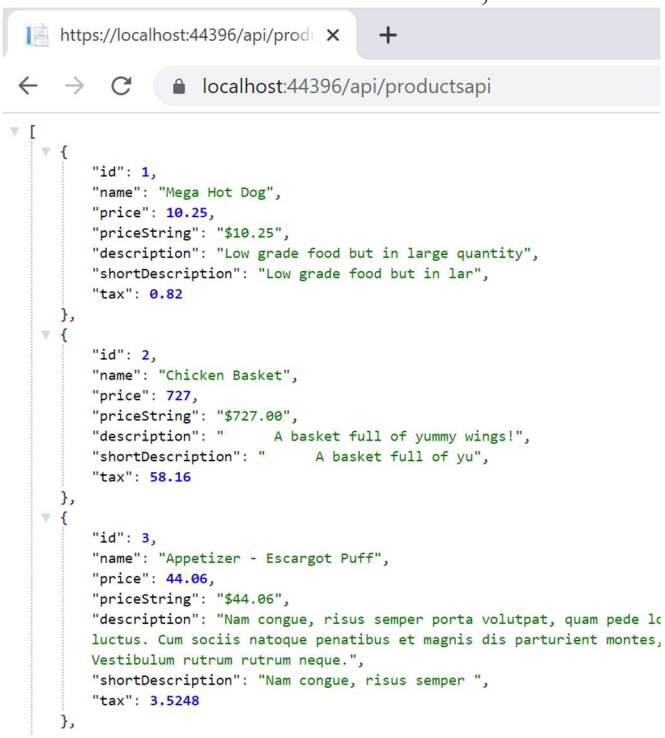
Goal: In this activity, you will learn how to translate a model class into a publicly viewable data transfer object.

Starting Point

You will need the code from a previous activity. You will use your Browser and **Postman** to test your REST(ful) Services. After the previous tutorial, our application provided a REST service API for **products** as seen here.



After we are finished with this lesson, the API will also provide an API with some new values.



What is a DTO?

A data transfer object is a class that is used for public display of another object in the app. There are a number of reasons why programmers use them, especially when creating REST API services.

DTOs protect private information

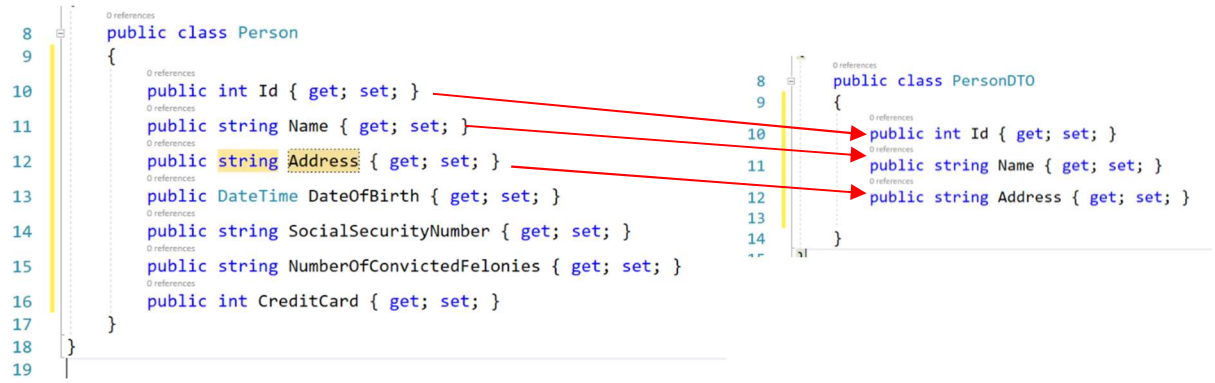
Security is the first reason why a DTO should be implemented. Suppose you had a user class that contained the following properties; some of these are obviously sensitive and should not be displayed on a public API where all properties of the user are shared.

```
8   public class Person
9   {
10      public int Id { get; set; }
11      public string Name { get; set; }
12      public string Address { get; set; }
13      public DateTime DateOfBirth { get; set; }
14      public string SocialSecurityNumber { get; set; }
15      public string NumberOfConvictedFelonies { get; set; }
16      public int CreditCard { get; set; }
17  }
18 }
19
```

A DTO for this case might be better to share only publicly known information.

```
8   public class PersonDTO
9   {
10      public int Id { get; set; }
11      public string Name { get; set; }
12      public string Address { get; set; }
13
14  }
```

In order to create the DTO, a translation between properties is required. Some are transferred; others are intentionally ignored.



DTOs allow transfers of bundles of information

Suppose you wanted to combine two classes into one object. For example, an order consists of customer information, products, and quantities in a shopping cart. An OrdersDTO would combine some of these properties together, making for a more efficient data transfer.

```
public class Order
{
    public string OrderNo { get; set; }
    public int NumberOfItems { get; set; }
    public int TotalAmount { get; set; }
    public Customer Customer { get; set; }
}

public class Customer
{
    public int CustomerID { get; set; }
    public string FullName { get; set; }
    public string Postcode { get; set; }
    public string ContactNo { get; set; }
}

public class OrderDTO
{
    public int OrderId { get; set; }
    public int NumberOfItems { get; set; }
    public int TotalAmount { get; set; }
    public int CustomerId { get; set; }
    public string Name { get; set; }
    public string Postcode { get; set; }
    public string MobileNo { get; set; }
}
```

DTOs can provide additional information

In the example we are going to implement, we will include a formatted currency value of the price, a short version of the description, and a calculated tax amount in the DTO.

```
45 references
public class ProductModel
{
    [DisplayName("Id number")]
    11 references
    public int Id { get; set; }
    [DisplayName("Product Name")]
    10 references
    public string Name { get; set; }

    [DataType(DataType.Currency)]
    [DisplayName("Cost to customer")]
    9 references
    public decimal Price { get; set; }
    [DisplayName("What you get...")]
    9 references
    public string Description { get; set; }
```

```
public class ProductDTO
{
    1 reference
    public int Id { get; set; }
    1 reference
    public string Name { get; set; }
    1 reference
    public decimal Price { get; set; }
    1 reference
    public string PriceString { get; set; }
    1 reference
    public string Description { get; set; }
    1 reference
    public string ShortDescription { get; set; }
    1 reference
    public decimal Tax { get; set; }

    5 references
    public ProductDTO(int id, string name, decimal price, string description)
    {
        Id = id;
        Name = name;
        Price = price;
        PriceString = string.Format("{0:C}", price);
        Description = description;
        ShortDescription = description.Length <= 25 ? description : description.Substring(0, 25);
        Tax = price * 0.08M;
    }

}
```

DTOs prevent an API from breaking

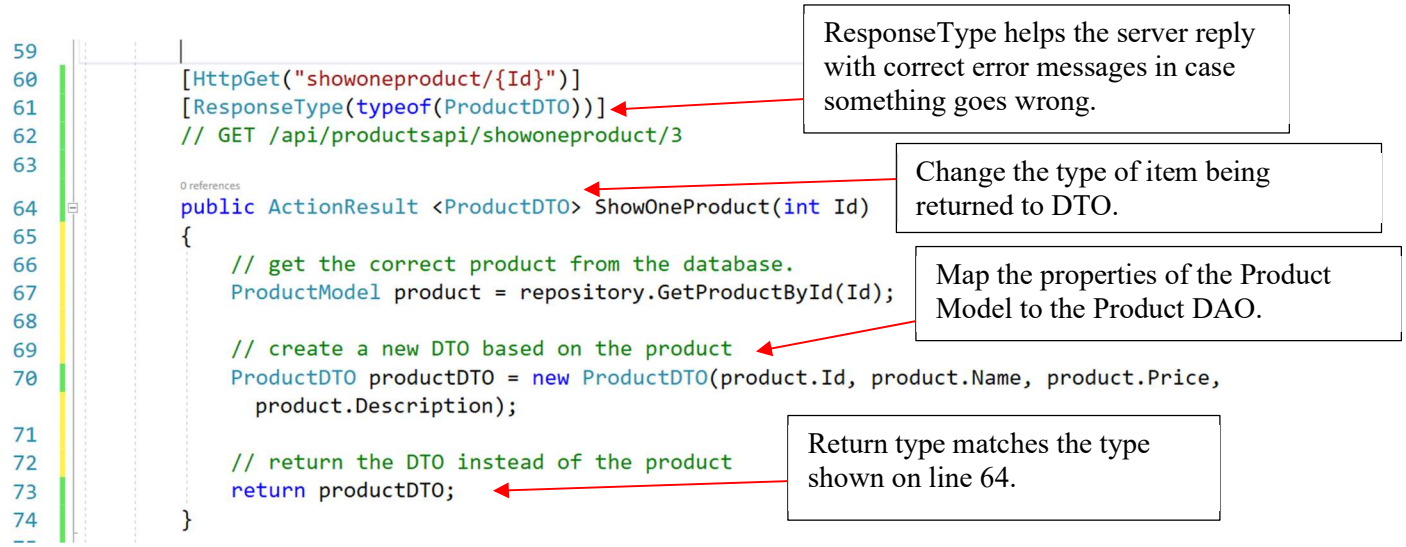
Since the DTO is the public-facing version of the object, the internal model can change without causing breaking changes to an application. The internal conversion between Model and DTO will have to change whenever a model changes, but the end users of the API will not notice any difference in the service.

Instructions

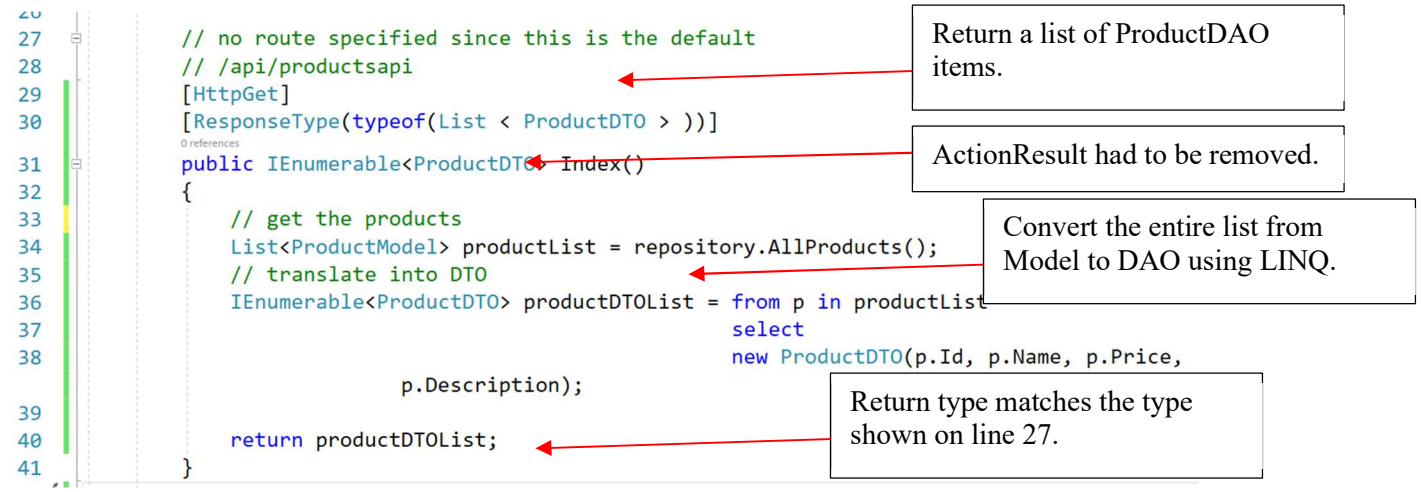
1. Create a new class in the Models folder called **ProductDTO**. Use the code listed above to define the properties and constructor for the class.

2. Open the ProductsAPIController and modify the following methods.

The ShowOneProduct method is the simplest example so we will start here.



To get all products, perform the same Model to DAO conversion, but this time with an entire list.



If you don't like LINQ, you can create the same method using a traditional foreach loop.

```
        //
        // /api/productsapi/index2
        [HttpGet("index2")]
        [ResponseType(typeof(List<ProductDTO>))]
        0 references
        public IEnumerable<ProductDTO> Index2()
        {
            // translate into DTO
            List<ProductModel> productList = repository.AllProducts();
            // translate into DTO
            List<ProductDTO> productDTOList = new List<ProductDTO>();
            foreach (ProductModel p in productList){
                productDTOList.Add(new ProductDTO(p.Id, p.Name, p.Price, p.Description));
            }

            return productDTOList;
        }
```

The search method is similar…

```
        [HttpGet("searchresults/{searchTerm}")]
        // GET /api/productsapi/searchresults/xyz
        0 references
        public IEnumerable<ProductDTO> SearchResults(string searchTerm)
        {
            List<ProductModel> productList = repository.SearchProducts(searchTerm);

            // translate into DTO
            List<ProductDTO> productDTOList = new List<ProductDTO>();
            foreach (ProductModel p in productList)
            {
                productDTOList.Add(new ProductDTO(p.Id, p.Name, p.Price, p.Description));
            }
            return  productDTOList;
        }
```

Process Edit returns an entire list

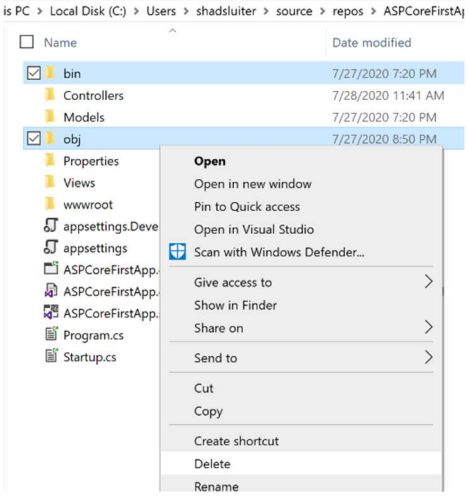
```
        [HttpPut("processedit")]
        [ResponseType(typeof(List<ProductDTO>))]
        // GET /api/productsapi/processedit/product
        0 references
        public IEnumerable<ProductDTO> ProcessEdit(ProductModel product)
        {
            repository.Update(product);
            List<ProductModel> productList = repository.AllProducts();

            // translate into DTO
            IEnumerable<ProductDTO> productDTOList = from p in productList
                                                     select new ProductDTO(p.Id, p.Name, p.Price,
                        p.Description);

            return productDTOList;

        }
```

Return one item.

```
119
120         [HttpPut("ProcessEditReturnOneItem")]
121         [ResponseType(typeof( ProductDTO))]
122         // GET /api/productsapi/processeditreturnoneitem/product
            0 references
123         public ActionResult <ProductDTO> ProcessEditReturnOneItem(ProductModel product)
124         {
125             repository.Update(product);
126             ProductModel updatedProduct = repository.GetProductById(product.Id);
127             ProductDTO productDTO = new ProductDTO(product.Id, product.Name, product.Price,
                  product.Description);
128             return productDTO;
129         }
130
131     }
```

1. Run the program and test each of the API endpoints.
2. Take a screenshot of each endpoint result. Paste the image into a Microsoft Word document. Add a caption describing what is being demonstrated.

Deliverables:



1. This activity has multiple parts. Complete all parts before submitting.
2. Submit a Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
4. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.

Part 3 Right Mouse Click

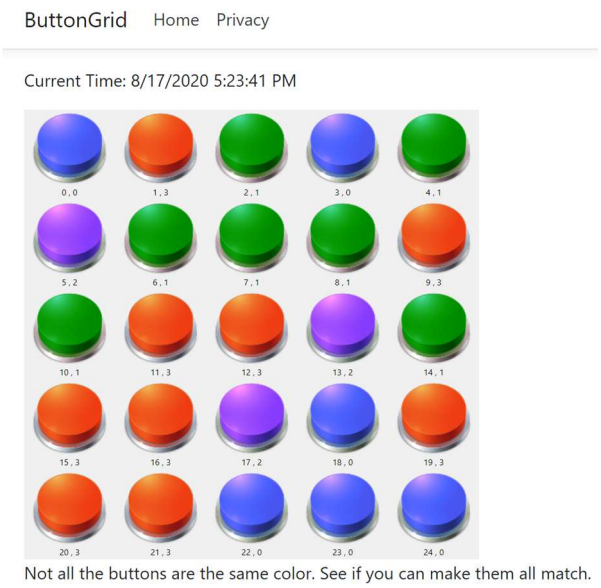
Goal and Directions:

In this activity, you will process a right mouse button click using JavaScript.

You will need the code from the **Button grid** application.

Current State

After completing the previous tutorial with the Button grid app, the buttons respond to a **left click**. We will add the right click response next.

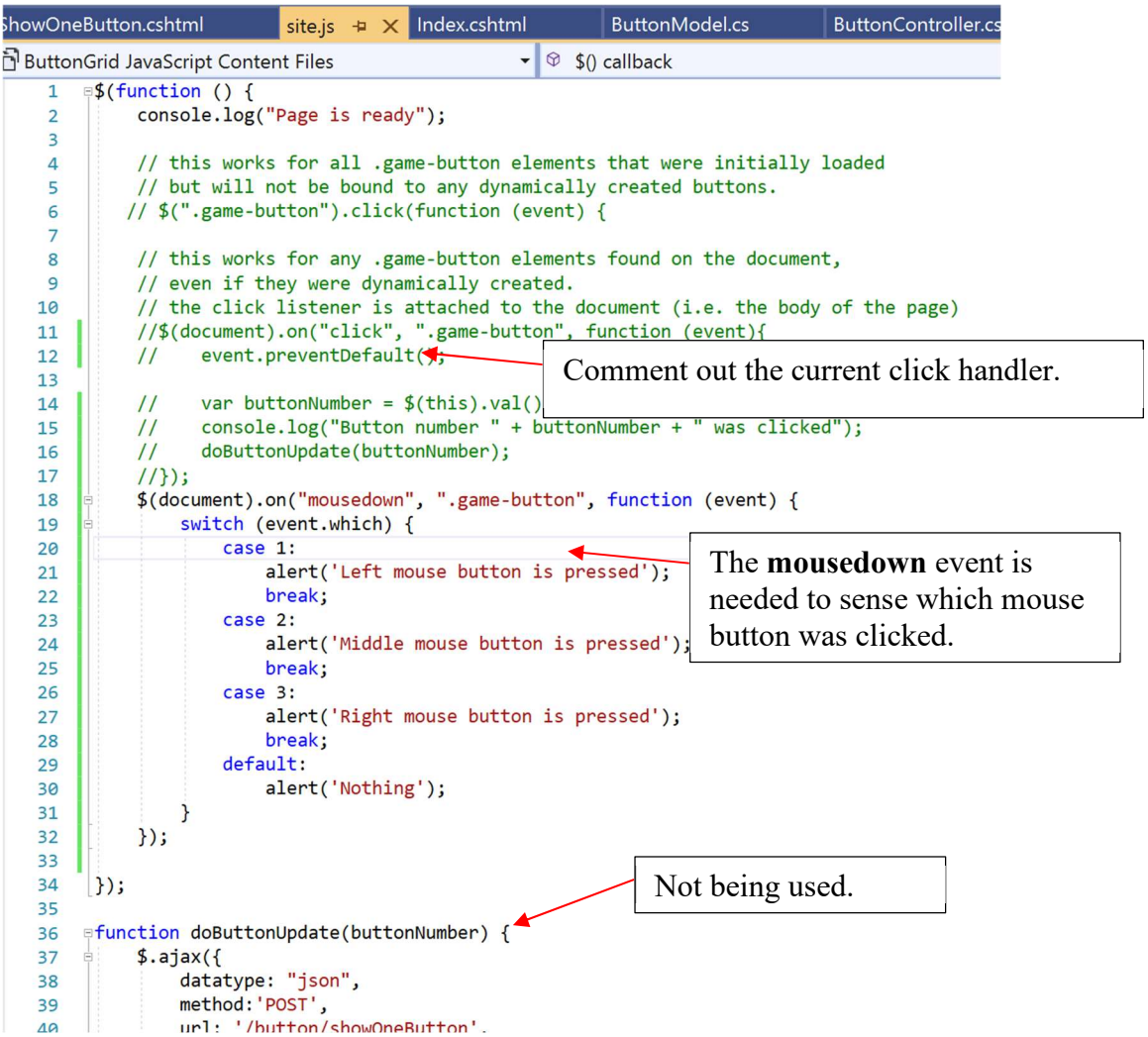


Instructions

1. In the **ButtonController**, create a new method called **RightClickShowOneButton**. The new method will reset the button state to 0.

```
46          public IActionResult ShowOneButton(int buttonNumber)
47          {
48              // add one to the button state. If greater than 4, reset to 0.
49              buttons.ElementAt(buttonNumber).ButtonState = (buttons.ElementAt(buttonNumber).ButtonState + 1) % 4;
50
51              // re-display the button that was clicked
52              return PartialView(buttons.ElementAt(buttonNumber));
53          }
54
55          public IActionResult RightClickShowOneButton(int buttonNumber)
56          {
57              // on right click always reset to 0.
58              buttons.ElementAt(buttonNumber).ButtonState = 0;
59
60              // re-display the button that was clicked
61              return PartialView("ShowOnebutton", buttons.ElementAt(buttonNumber));
62          }
63      }
64  }
```

2. In the site.js file, comment out the existing button click handler and create a new mousedown handler. The result of a mousedown will display a pop-up message whenever the right or left mouse button is clicked.

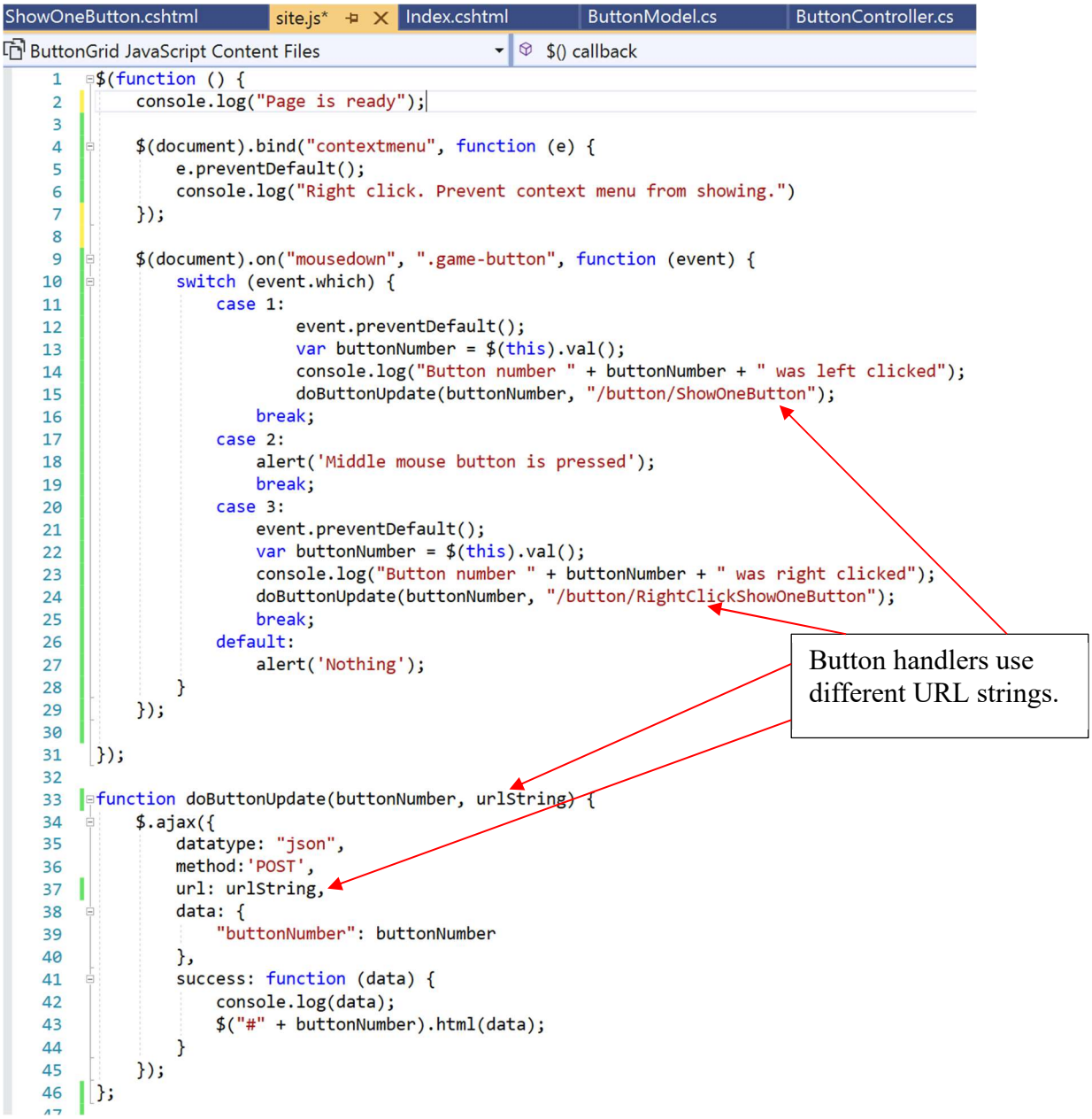



```
$(function () {
    console.log("Page is ready");

    // this works for all .game-button elements that were initially loaded
    // but will not be bound to any dynamically created buttons.
    // $(".game-button").click(function (event) {

    // this works for any .game-button elements found on the document,
    // even if they were dynamically created.
    // the click listener is attached to the document (i.e. the body of the page)
    //$(document).on("click", ".game-button", function (event){
    //    event.preventDefault();

    //    var buttonNumber = $(this).val()
    //    console.log("Button number " + buttonNumber + " was clicked");
    //    doButtonUpdate(buttonNumber);
    //});
    $(document).on("mousedown", ".game-button", function (event) {
        switch (event.which) {
            case 1:
                alert('Left mouse button is pressed');
                break;
            case 2:
                alert('Middle mouse button is pressed');
                break;
            case 3:
                alert('Right mouse button is pressed');
                break;
            default:
                alert('Nothing');
        }
    });

});

function doButtonUpdate(buttonNumber) {
    $.ajax({
        datatype: "json",
        method:'POST',
        url: '/button/showOneButton',
```

3. Bind a new event to the context menu to prevent the right-click menu from appearing on the page.

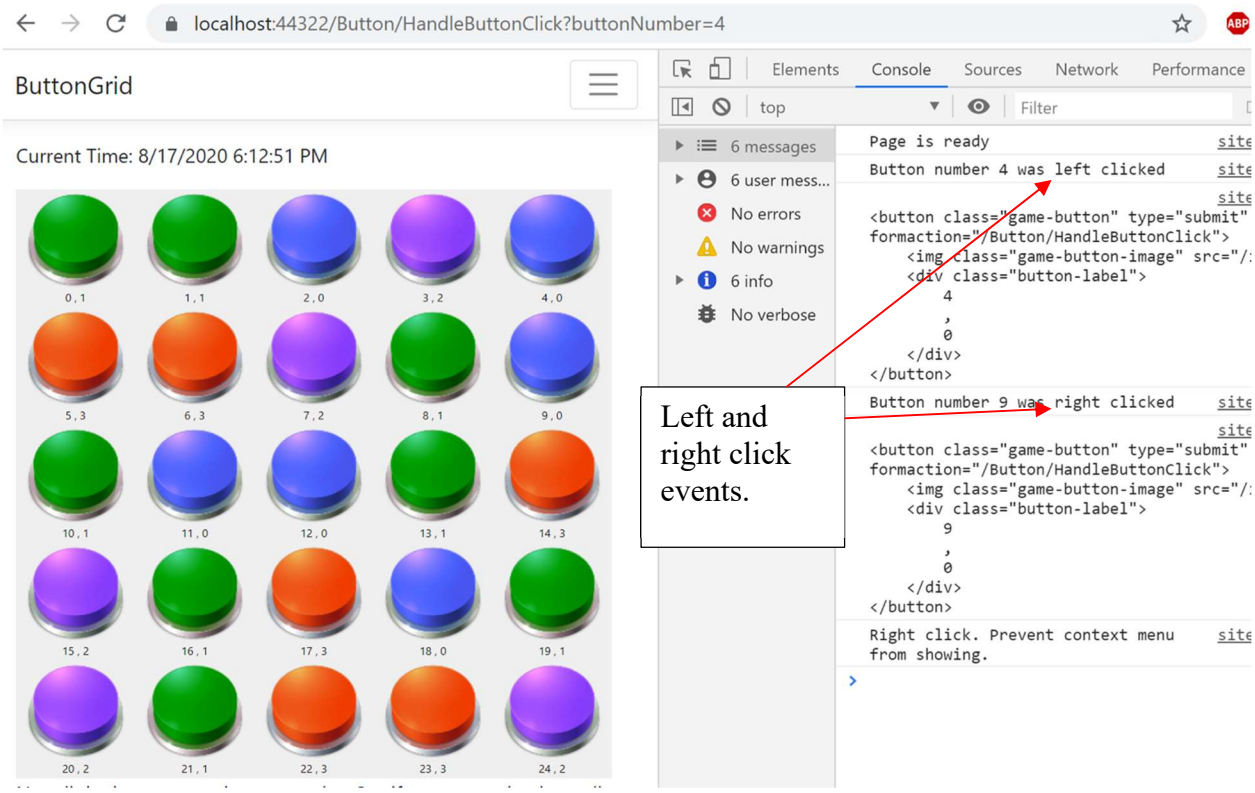
```
$(document).bind("contextmenu", function (e) {
    e.preventDefault();
    console.log("Right click. Prevent context menu from showing.")
});
```

4. Update the left and right mouse button cases to call the **doButtonUpdate** function. Notice that there is an extra parameter for the correct URL to call. In the **ButtonController**, the right mouse handler is different than the left.



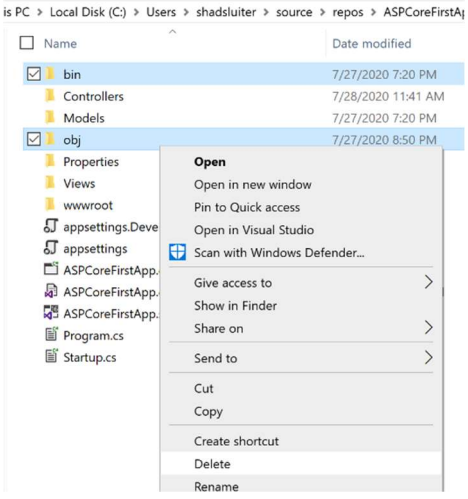

```
$(function () {
    console.log("Page is ready");

    $(document).bind("contextmenu", function (e) {
        e.preventDefault();
        console.log("Right click. Prevent context menu from showing.")
    });

    $(document).on("mousedown", ".game-button", function (event) {
        switch (event.which) {
            case 1:
                    event.preventDefault();
                    var buttonNumber = $(this).val();
                    console.log("Button number " + buttonNumber + " was left clicked");
                    doButtonUpdate(buttonNumber, "/button/ShowOneButton");
                break;
            case 2:
                alert('Middle mouse button is pressed');
                break;
            case 3:
                event.preventDefault();
                var buttonNumber = $(this).val();
                console.log("Button number " + buttonNumber + " was right clicked");
                doButtonUpdate(buttonNumber, "/button/RightClickShowOneButton");
                break;
            default:
                alert('Nothing');
        }
    });

});

function doButtonUpdate(buttonNumber, urlString) {
    $.ajax({
        datatype: "json",
        method:'POST',
        url: urlString,
        data: {
            "buttonNumber": buttonNumber
        },
        success: function (data) {
            console.log(data);
            $("#" + buttonNumber).html(data);
        }
    });
};
```

5. Run the program. Open the browser console to see the status messages for each event.



6. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

Deliverables:



1. Submit a Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
2. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
3. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.