



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.

The screenshot shows a web application interface. At the top, there is a navigation bar with links for Home, Privacy, Products, and Search Products. Below the navigation bar, there is a heading "Create New" and a "Create New" button. The main content area displays three product items in a grid:

Image	Name	Description	Price	Action Buttons
	hamburger with Mustard	Low grade food	\$1.22	Show Details Edit Delete
	Chicken Basket	A basket full of yummy wings!	\$727.00	Show Details Edit Delete
	Appetizer - Escargot Puff	Nam congue, risus semper porta volutpat, quam pede lobortis ligula, sit amet eleifend pede libero quis orci. Nullam molestie nibh in lectus. Pellentesque at nulla. Suspendisse potenti. Cras in purus eu magna.	\$44.06	Show Details Edit Delete

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

The screenshot shows the Solution Explorer window in Visual Studio. The window title is "Solution Explorer". It displays the project structure for "ASPCoreFirstApp":

- Solution 'ASPCoreFirstApp' (1 of 1 project)
 - ASPCoreFirstApp
 - Connected Services
 - Dependencies
 - Properties
 - wwwroot
 - Controllers
 - HomeController.cs
 - ProductsAPIController.cs
 - ProductsController.cs

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 return 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).

```

9  using Microsoft.AspNetCore.Mvc;
10
11 namespace ASPCoreFirstApp.Controllers
12 {
13     [ApiController]
14     [Route("api/[controller]")]
15     public class ProductsAPIController : ControllerBase
16     {
17     }
--
```

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

```

Startup.cs ProductsAPIController.cs* _productCard.cshtml Index-noajax.cshtml site.js
ASPCoreFirstApp ASPCoreFirstApp.Controllers.ProductsAPIController

24
25 // no route specified since this is the default
26 // /api/productsapi
27 [HttpGet] ← Each method has a route (URL) pattern to follow.
28 public ActionResult <IEnumerable<ProductModel>> Index()
29 {
30     return repository.AllProducts();
31 }
32
33 [HttpGet("searchresults/{searchTerm}")]
34 // GET /api/productsapi/searchresults/xyz ← Each method has a return data type listed in <angle brackets>. The first two methods return lists of products.
35 public ActionResult<IEnumerable<ProductModel>> SearchResults(string searchTerm)
36 {
37     List<ProductModel> productList = repository.SearchProducts(searchTerm);
38     return productList;
39 }
40 // not used in a REST api. This method returns a data entry form
41 //public IActionResult SearchForm()
42 //{
43 //    return View();
44 //}
45
46 [HttpGet("showoneproduct/{Id}")]
47 // GET /api/productsapi/showoneproduct/3 ← ActionResult instead of IActionResult
48 public ActionResult <ProductModel> ShowOneProduct(int Id)
49 {
50     return repository.GetProductById(Id);
51 }
52 // not used in a REST api. This method returns a data entry form
53 //public IActionResult ShowEditForm(int Id)
54 //{
55 //    return View(repository.GetProductById(Id));
56 //}
57
58 [HttpPost("processedit")]
59 // GET /api/productsapi/processedit/product ← Two methods don't make sense in an API. They are commented out.
60 public ActionResult<IEnumerable<ProductModel>> ProcessEdit(ProductModel product)
61 {
62     repository.Update(product);
63     return repository.AllProducts();
64 }
65
66 [HttpPost("ProcessEditReturnOneItem")]
67 // GET /api/productsapi/processeditreturnoneitem/product ← The PUT method is for performing updates.
68 public ActionResult <ProductModel> ProcessEditReturnOneItem(ProductModel product)
69 {
70     repository.Update(product);
71     return repository.GetProductById(product.Id);
72 }

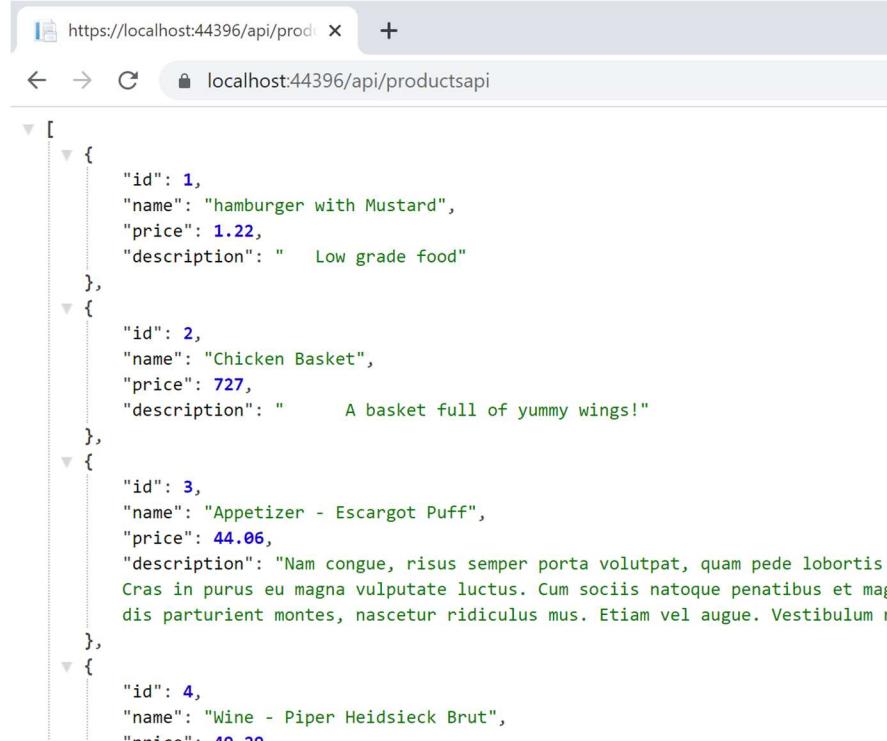
```

The code shows several methods in the ProductsAPIController. Annotations explain the design choices:

- Line 27:** Each method has a route (URL) pattern to follow.
- Line 35:** Each method has a *return data type* listed in <angle brackets>. The first two methods return lists of products.
- Line 41:** **ActionResult** instead of **IActionResult**
- Lines 46-56:** Two methods don't make sense in an API. They are commented out.
- Line 58:** The **PUT** method is for performing updates.
- Line 68:** Each return statement no longer is for a **View**, but simply data that is serialized into **JSON** text.

Testing.

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

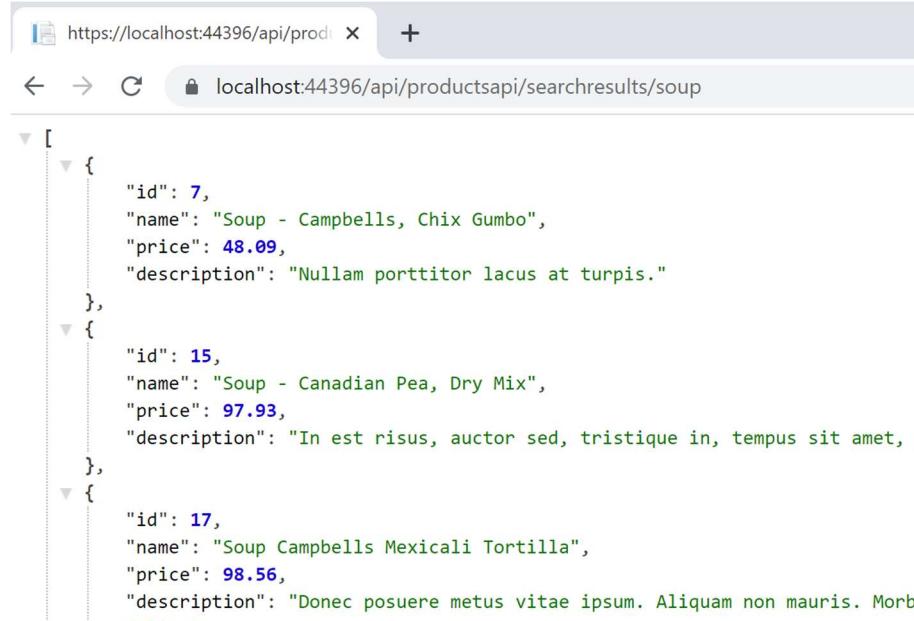


A screenshot of a web browser window displaying a JSON array of product objects. The URL in the address bar is `localhost:44396/api/productsapi`. The page content shows a list of four products:

```
[{"id": 1, "name": "hamburger with Mustard", "price": 1.22, "description": "Low grade food"}, {"id": 2, "name": "Chicken Basket", "price": 7.27, "description": "A basket full of yummy wings!"}, {"id": 3, "name": "Appetizer - Escargot Puff", "price": 44.06, "description": "Nam congue, risus semper porta volutpat, quam pede lobortis Cras in purus eu magna vulputate luctus. Cum sociis natoque penatibus et magis parturient montes, nascetur ridiculus mus. Etiam vel augue. Vestibulum r"}, {"id": 4, "name": "Wine - Piper Heidsieck Brut", "price": 40.00, "description": "Piper Heidsieck Brut is a sparkling wine from France. It has a pale yellow color and a delicate, fruity flavor with hints of citrus and peach. The bubbles are fine and persistent, providing a light and refreshing texture. It's perfect for aperitifs or as an accompaniment to various dishes."}]
```

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

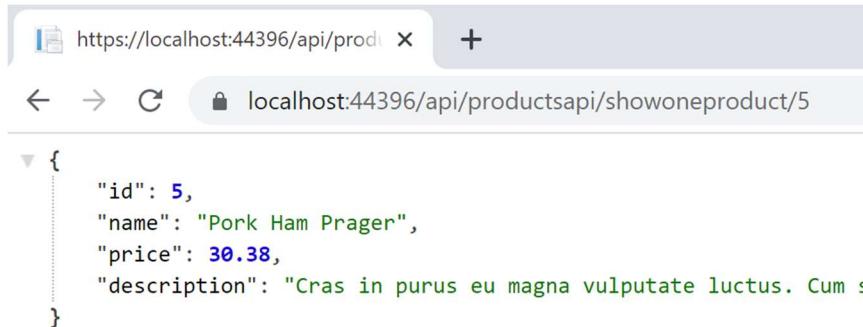


A screenshot of a web browser window displaying a JSON array of product objects filtered by the search term "soup". The URL in the address bar is `localhost:44396/api/productsapi/searchresults/soup`. The page content shows a list of three products:

```
[{"id": 7, "name": "Soup - Campbells, Chix Gumbo", "price": 48.09, "description": "Nullam porttitor lacus at turpis."}, {"id": 15, "name": "Soup - Canadian Pea, Dry Mix", "price": 97.93, "description": "In est risus, auctor sed, tristique in, tempus sit amet, :"}, {"id": 17, "name": "Soup Campbells Mexicali Tortilla", "price": 98.56, "description": "Donec posuere metus vitae ipsum. Aliquam non mauris. Morb: ---"}]
```

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**



A screenshot of a web browser window. The address bar shows "https://localhost:44396/api/prod...". The main content area displays a JSON object:

```
{  
  "id": 5,  
  "name": "Pork Ham Prager",  
  "price": 30.38,  
  "description": "Cras in purus eu magna vulputate luctus. Cum s..."  
}
```

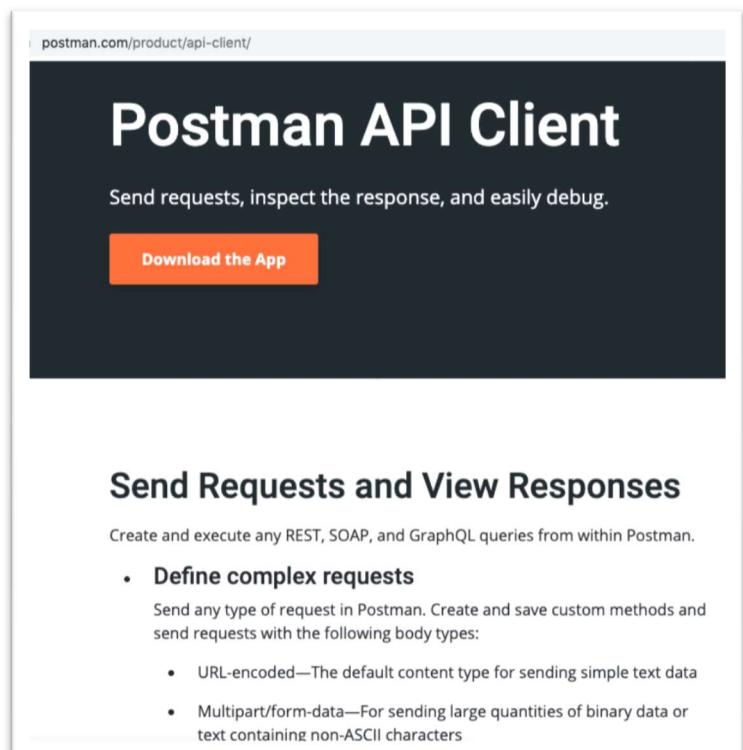
6. Take a screenshot of the app at this stage. Paste it into a Word document and caption the image with a brief explanation of what you just demonstrated.

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**.



The screenshot shows the official Postman API Client website. The header reads "postman.com/product/api-client/" and "Postman API Client". Below the header, it says "Send requests, inspect the response, and easily debug." and has a "Download the App" button. The main content area is titled "Send Requests and View Responses" and includes the following text:
Create and execute any REST, SOAP, and GraphQL queries from within Postman.

- **Define complex requests**
Send any type of request in Postman. Create and save custom methods and send requests with the following body types:
 - URL-encoded—The default content type for sending simple text data
 - Multipart/form-data—For sending large quantities of binary data or text containing non-ASCII characters

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

The screenshot shows the Postman interface with the following annotations:

- A box labeled "GET request method." points to the "Method" dropdown which is set to "GET".
- A box labeled "URL endpoint" points to the "URL" field which contains "https://localhost:44396/api/productsapi".
- A box labeled "Send it!" points to the "Send" button.
- A box labeled "Status 200 means everything worked well." points to the status bar at the top right which shows "Status: 200 OK".
- A box labeled "JSON text results show an array of all items in the catalog!" points to the JSON response body, which is displayed in a code editor-like format with line numbers 1 through 18. The JSON data represents a catalog of food items with fields: id, name, price, and description.

```
1 {
2   "id": 1,
3   "name": "hamburger with Mustard",
4   "price": 1.22,
5   "description": "    Low grade food"
6 },
7 {
8   "id": 2,
9   "name": "Chicken Basket",
10  "price": 727.00,
11  "description": "    A basket full of yummy wings!"
12 },
13 {
14   "id": 3,
15   "name": "Appetizer - Escargot Puff",
16   "price": 44.06,
17   "description": "Nam congue, risus semper porta volutpat, quam pede lobortis ligula, sit amet eleifend pede libero quis orci. Nullam molestie nibh in lectus. Pellentesque at nulla. Suspendisse potenti. Cras in purus eu magna vulputate luctus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Vivamus vestibulum sagittis sapien. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Etiam vel augue. Vestibulum rutrum enim
```

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.”

New URL

All results contain the word “lamb”

```
[{"id": 19, "name": "Lamb Shoulder Boneless Nz", "price": 59.16, "description": "Cras mi pede, malesuada in, imperdi"}, {"id": 44, "name": "Lamb Tenderloin Nz Fr", "price": 84.62, "description": "Nulla tempus. Vivamus in felis eu sapien cursus vestibulum. Proin eu mi. Nulla ac enim. In tempor, turpis r"}, {"id": 70, "name": "Lamb - Pieces, Diced", "price": 10.99, "description": "Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris nonummy nisl. Sed ut perspiciatis unde omnis iste natus error sit amet, consectetur adipisci velit. Nam dui ligula, fringilla vel, pretium at, tincidunt id, nunc. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."}]
```

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.

The screenshot shows the Postman interface. At the top, there's a header bar with 'Postman' and navigation links like 'File', 'Edit', 'View', 'Help'. Below it is a toolbar with 'New', 'Import', 'Runner', and a search icon. The main workspace is titled 'My Workspace' with a dropdown menu. A 'Launchpad' tab is selected, showing a green 'GET https://localhost:44396/api/pro...' button. To the right of this button is a callout box with the text 'URL points to showoneproduct/6' and a red arrow pointing to the URL. Below the URL is another callout box with the text 'Only one product is returned.' and a red arrow pointing to the response preview area. The 'Untitled Request' section shows a 'GET' method and the URL 'https://localhost:44396/api/productsapi/showoneproduct/6'. Below this are tabs for 'Params', 'Authorization', 'Headers (7)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. The 'Headers' tab is active, showing a table with one row: 'KEY' (Key) and 'DESCRIPTION' (Description). The 'Body' tab is also active, displaying a JSON response. The response is a single object with the following properties:

```
1  {
2    "id": 6,
3    "name": "Cookie - Dough Variety",
4    "price": 19.25,
5    "description": "Morbi vestibulum, velit id pretium iaculis, diam erat fermentum justo, nec condimentum neque sapien placerat ante.\n      Nulla justo. Aliquam quis turpis eget elit sodales scelerisque. Mauris sit amet eros. Suspendisse accumsan tortor quis turpis.\n      Sed ante."}
```

At the bottom of the body preview, there are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'JSON'. To the right of the preview area, there are status indicators: a red icon, 'Status: 200 OK', 'Time: 2.72 s', 'Size: 497 B', and a 'Save Response' button. There are also icons for copy and search.

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.”

The screenshot shows the Postman interface with the following annotations:

- PUT (i.e., update) is the request method type**: Points to the method dropdown.
- processededit is the method**: Points to the URL path.
- “Raw” and “JSON” body settings**: Points to the Body tab and the JSON dropdown.
- Body of the request contains the “form” data**: Points to the request body code.
- Successfully returns the entire list of data. Notice that item #1 has changed.**: Points to the response body code.

```

1 {
2   "Id":1,
3   "Name":"Mega Hot Dog",
4   "Price":10.25,
5   "Description":"Low grade food but in large quantity"
6 }
    
```

```

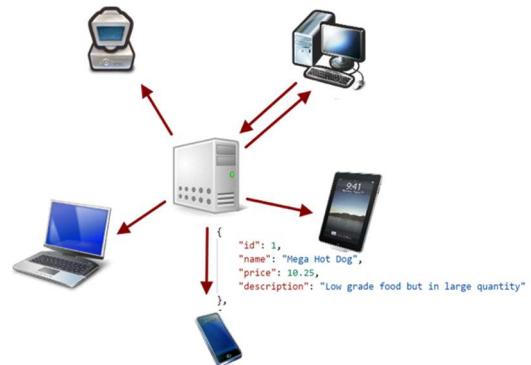
1 {
2   "id": 1,
3   "name": "Mega Hot Dog",
4   "price": 10.25,
5   "description": "Low grade food but in large quantity"
6 },
7 {
8   "id": 2,
9   "name": "Chicken Basket",
10  "price": 727.00,
11  "description": "A basket full of yummy wings!"
12 }
    
```

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.

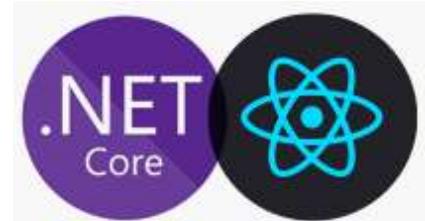


Picture how a user connects to an airline: the company **webpage**, the **mobile app**, **email**, and **texting** servers all connect to the same central set of data about flights, tickets, and related records. The central server is probably running a REST API.

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:

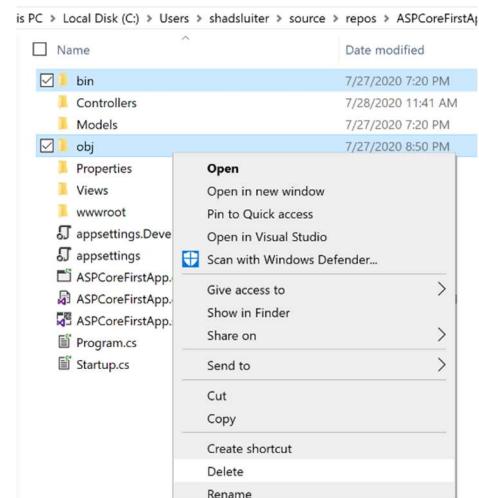
- Java Enterprise
- Java Spring and Spring Boot

- 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.

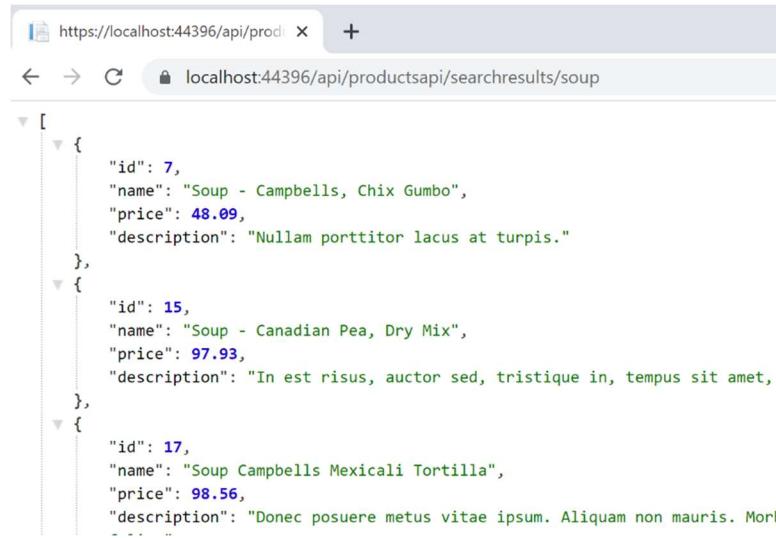


Part 2 Data Transfer Objects (DTO)

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

Starting Point

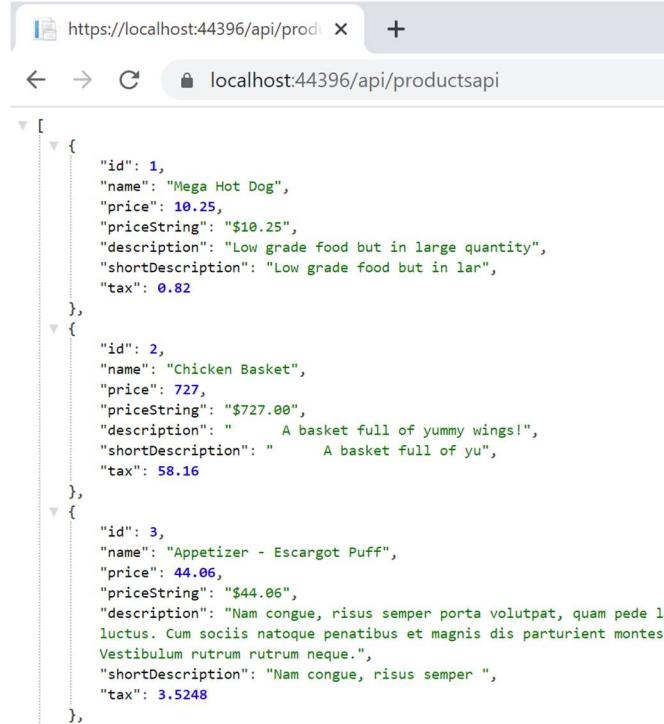
You will need the code from a previous activity. You will use your Browser and **Postman** to test your REST(ful) Services. After the previous tutorial, our application provided a REST service API for **products** as seen here.



A screenshot of a web browser window. The address bar shows "localhost:44396/api/productsapi/searchresults/soup". The main content area displays a JSON array of product objects. The first few objects are:

```
[{"id": 7, "name": "Soup - Campbells, Chix Gumbo", "price": 48.09, "description": "Nullam porttitor lacus at turpis."}, {"id": 15, "name": "Soup - Canadian Pea, Dry Mix", "price": 97.93, "description": "In est risus, auctor sed, tristique in, tempus sit amet, :"}, {"id": 17, "name": "Soup Campbells Mexicali Tortilla", "price": 98.56, "description": "Donec posuere metus vitae ipsum. Aliquam non mauris. Morb: ..."}]
```

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



A screenshot of a web browser window. The address bar shows "localhost:44396/api/productsapi". The main content area displays a JSON array of product objects. The first few objects are:

```
[{"id": 1, "name": "Mega Hot Dog", "price": 10.25, "priceString": "$10.25", "description": "Low grade food but in large quantity", "shortDescription": "Low grade food but in lar", "tax": 0.82}, {"id": 2, "name": "Chicken Basket", "price": 727, "priceString": "$727.00", "description": "A basket full of yummy wings!", "shortDescription": "A basket full of yu", "tax": 58.16}, {"id": 3, "name": "Appetizer - Escargot Puff", "price": 44.06, "priceString": "$44.06", "description": "Nam congue, risus semper porta volutpat, quam pede luctus. Cum sociis natoque penatibus et magnis dis parturient montes, Vestibulum rutrum rutrum neque.", "shortDescription": "Nam congue, risus semper ", "tax": 3.5248}...]
```

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

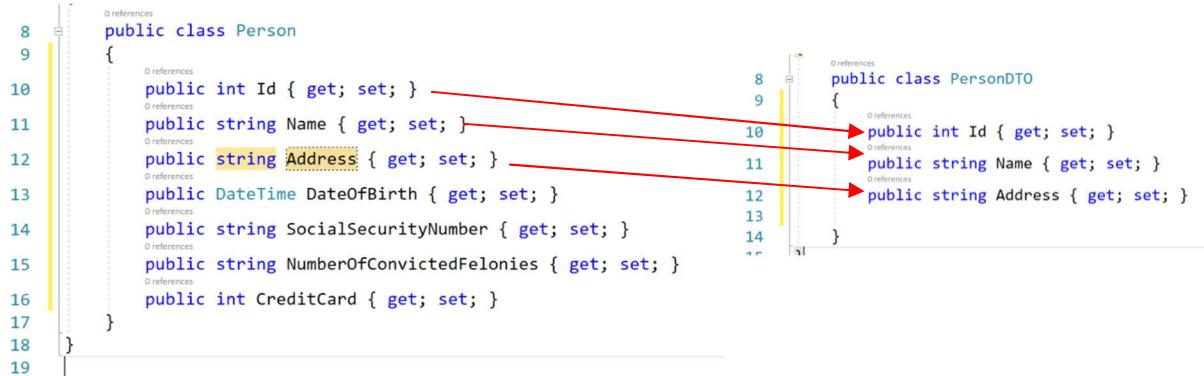
Security is the first reason why a DTO should be implemented. Suppose you had a user class that contained the following properties; some of these are obviously sensitive and should not be displayed on a public API where all properties of the user are shared.

```
8  public class Person
9  {
10     public int Id { get; set; }
11     public string Name { get; set; }
12     public string Address { get; set; } // This is sensitive
13     public DateTime DateOfBirth { get; set; }
14     public string SocialSecurityNumber { get; set; } // This is sensitive
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.

```

10    public class ProductModel
11    {
12        [DisplayName("Id number")]
13        public int Id { get; set; }
14        [DisplayName("Product Name")]
15        public string Name { get; set; }
16
17        [DataType(DataType.Currency)]
18        [DisplayName("Cost to customer")]
19        public decimal Price { get; set; }
20        [DisplayName("What you get...")]
21        public string Description { get; set; }
22
23
24
25
26
27
28
29

```



```

8    public class ProductDTO
9    {
10        public int Id { get; set; }
11        public string Name { get; set; }
12        public decimal Price { get; set; }
13        public string PriceString { get; set; }
14        public string Description { get; set; }
15        public string ShortDescription { get; set; }
16        public decimal Tax { get; set; }
17
18        public ProductDTO(int id, string name, decimal price, string description)
19        {
20            Id = id;
21            Name = name;
22            Price = price;
23            PriceString = string.Format("{0:C}", price);
24            Description = description;
25            ShortDescription = description.Length <= 25 ? description : description.Substring(0, 25);
26            Tax = price * 0.08M;
27        }
28    }
29

```

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.

```

59
60 [HttpGet("showoneproduct/{Id}")]
61 [ResponseType(typeof(ProductDTO))]
62 // GET /api/productsapi/showoneproduct/3
63
64 public ActionResult <ProductDTO> ShowOneProduct(int Id)
65 {
66     // get the correct product from the database.
67     ProductModel product = repository.GetProductById(Id);
68
69     // create a new DTO based on the product
70     ProductDTO productDTO = new ProductDTO(product.Id, product.Name, product.Price,
71     product.Description);
72
73     // return the DTO instead of the product
74     return productDTO;
}

```

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

```

27
28 // no route specified since this is the default
29 // /api/productsapi
30 [HttpGet]
31 [ResponseType(typeof(List < ProductDTO > ))]
32 public IEnumerable<ProductDTO> Index()
33 {
34     // get the products
35     List<ProductModel> productList = repository.AllProducts();
36     // translate into DTO
37     IEnumerable<ProductDTO> productDTOList = from p in productList
38             select
39             new ProductDTO(p.Id, p.Name, p.Price,
40             p.Description);
41
42     return productDTOList;
}

```

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

```

44     // 
45     // /api/productsapi/index2
46     [HttpGet("index2")]
47     [ResponseType(typeof(List<ProductDTO>))]
48     public IEnumerable<ProductDTO> Index2()
49     {
50         // translate into DTO
51         List<ProductModel> productList = repository.AllProducts();
52         // translate into DTO
53         List<ProductDTO> productDTOList = new List<ProductDTO>();
54         foreach (ProductModel p in productList)
55         {
56             productDTOList.Add(new ProductDTO(p.Id, p.Name, p.Price, p.Description));
57         }
58
59         return productDTOList;
}

```

The search method is similar...

```

61
62     [HttpGet("searchresults/{searchTerm}")]
63     // GET /api/productsapi/searchresults/xyz
64
65     public IEnumerable<ProductDTO> SearchResults(string searchTerm)
66     {
67         List<ProductModel> productList = repository.SearchProducts(searchTerm);
68
69         // translate into DTO
70         List<ProductDTO> productDTOList = new List<ProductDTO>();
71         foreach (ProductModel p in productList)
72         {
73             productDTOList.Add(new ProductDTO(p.Id, p.Name, p.Price, p.Description));
74         }
75         return productDTOList;
}

```

Process Edit returns an entire list

```

103
104     [HttpPost("processedit")]
105     [ResponseType(typeof(List<ProductDTO>))]
106     // GET /api/productsapi/processedit/product
107     public IEnumerable<ProductDTO> ProcessEdit(ProductModel product)
108     {
109         repository.Update(product);
110         List<ProductModel> productList = repository.AllProducts();
111
112         // translate into DTO
113         IEnumerable<ProductDTO> productDTOList = from p in productList
114                                         select new ProductDTO(p.Id, p.Name, p.Price,
115                                         p.Description);
116
117         return productDTOList;
118     }

```

Return one item.

```

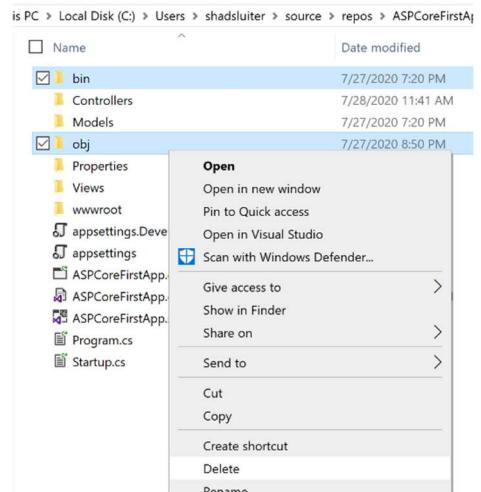
119
120 [HttpPost("ProcessEditReturnOneItem")]
121 [ResponseType(typeof( ProductDTO))]
122 // GET /api/productsapi/processeditreturnoneitem/product
123 public ActionResult <ProductDTO> ProcessEditReturnOneItem( ProductModel product)
124 {
125     repository.Update(product);
126     ProductModel updatedProduct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

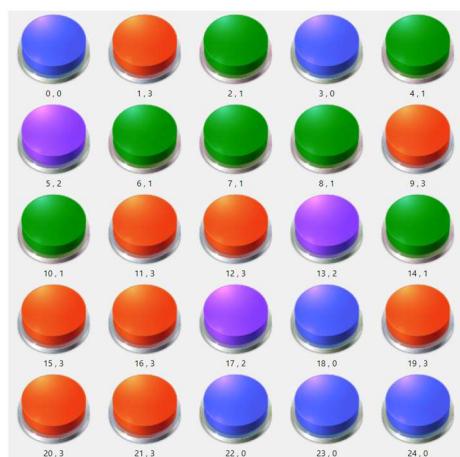
You will need the code from the **Button grid** application.

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.

ButtonGrid Home Privacy

Current Time: 8/17/2020 5:23:41 PM



Not all the buttons are the same color. See if you can make them all match.

Instructions

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

```
46  * 
47  *      0 references
48  public IActionResult ShowOneButton(int buttonNumber)
49  {
50      // add one to the button state. If greater than 4, reset to 0.
51      buttons.ElementAt(buttonNumber).ButtonState = (buttons.ElementAt(buttonNumber).ButtonState + 1) % 4;
52
53      // re-display the button that was clicked
54      return PartialView(buttons.ElementAt(buttonNumber));
55  }
56
57  *      0 references
58  public IActionResult RightClickShowOneButton(int buttonNumber)
59  {
60      // on right click always reset to 0.
61      buttons.ElementAt(buttonNumber).ButtonState = 0;
62
63      // re-display the button that was clicked
64      return PartialView("ShowOnebutton", buttons.ElementAt(buttonNumber));
65  }
```

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.

```

ShowOneButton.cshtml      site.js      Index.cshtml      ButtonModel.cs      ButtonController.cs
ButtonGrid JavaScript Content Files      $() callback

1  $(function () {
2      console.log("Page is ready");
3
4      // this works for all .game-button elements that were initially loaded
5      // but will not be bound to any dynamically created buttons.
6      // $(".game-button").click(function (event) {
7
8          // this works for any .game-button elements found on the document,
9          // even if they were dynamically created.
10         // the click listener is attached to the document (i.e. the body of the page)
11         //$(document).on("click", ".game-button", function (event){
12         //     event.preventDefault(); ← Comment out the current click handler.
13
14         //     var buttonNumber = $(this).val()
15         //     console.log("Button number " + buttonNumber + " was clicked");
16         //     doButtonUpdate(buttonNumber);
17         //});
18     $(document).on("mousedown", ".game-button", function (event) {
19         switch (event.which) {
20             case 1:
21                 alert('Left mouse button is pressed');
22                 break;
23             case 2:
24                 alert('Middle mouse button is pressed');
25                 break;
26             case 3:
27                 alert('Right mouse button is pressed');
28                 break;
29             default:
30                 alert('Nothing');
31         }
32     });
33 });
34 });
35
36 function doButtonUpdate(buttonNumber) { ← Not being used.
37     $.ajax({
38         datatype: "json",
39         method: 'POST',
40         url: '/button/showOneButton'.

```

The **mousedown** event is needed to sense which mouse button was clicked.

Comment out the current click handler.

Not being used.

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

```

18
19     $(document).bind("contextmenu", function (e) {
20         e.preventDefault();
21         console.log("Right click. Prevent context menu from showing.");
22     });
23

```

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.

```

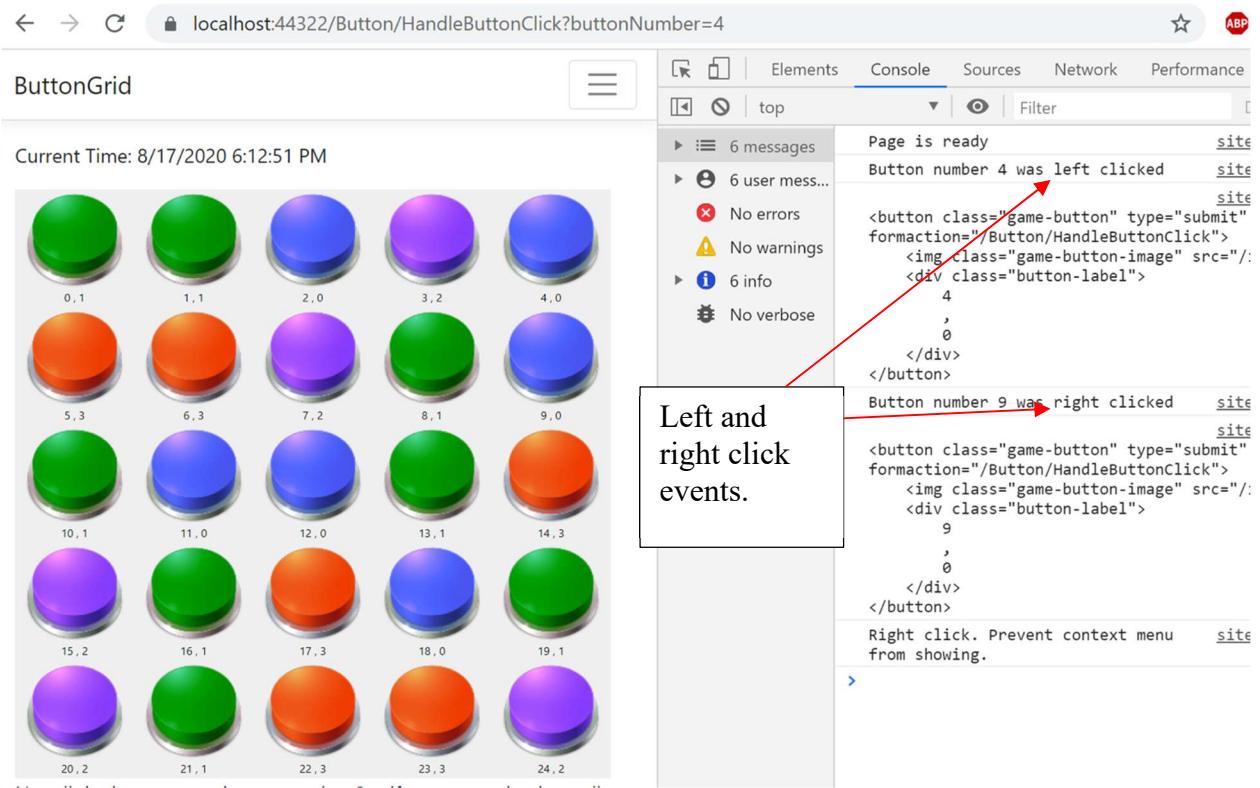
ShowOneButton.cshtml      site.js*  X  Index.cshtml  ButtonModel.cs  ButtonController.cs
ButtonGrid JavaScript Content Files  $() callback

1 $(function () {
2     console.log("Page is ready");
3
4     $(document).bind("contextmenu", function (e) {
5         e.preventDefault();
6         console.log("Right click. Prevent context menu from showing.");
7     });
8
9     $(document).on("mousedown", ".game-button", function (event) {
10        switch (event.which) {
11            case 1:
12                event.preventDefault();
13                var buttonNumber = $(this).val();
14                console.log("Button number " + buttonNumber + " was left clicked");
15                doButtonUpdate(buttonNumber, "/button>ShowOneButton");
16                break;
17            case 2:
18                alert('Middle mouse button is pressed');
19                break;
20            case 3:
21                event.preventDefault();
22                var buttonNumber = $(this).val();
23                console.log("Button number " + buttonNumber + " was right clicked");
24                doButtonUpdate(buttonNumber, "/button/RightClickShowOneButton");
25                break;
26            default:
27                alert('Nothing');
28        }
29    });
30});
31
32 function doButtonUpdate(buttonNumber, urlString) {
33     $.ajax({
34         datatype: "json",
35         method: 'POST',
36         url: urlString,
37         data: {
38             "buttonNumber": buttonNumber
39         },
40         success: function (data) {
41             console.log(data);
42             $("#" + buttonNumber).html(data);
43         }
44     });
45 }
46
47

```

Button handlers use different URL strings.

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



- 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:

- 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.
- In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson.
- 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.

