ASP.NET Core Activity #4

Part 1 AJAX with jQuery

Goal:

This tutorial will present techniques for performing partial page updates in four parts:

1. JavaScript and HTML only example

This example will demonstrate the use of AJAX using only HTML and JavaScript code. This will help the student understand the mechanics and code without including the extra features of Razor syntax, controllers, and partial views of the ASP.NET framework.

2. ASP.NET Core simple page example

The second example will demonstrate a relatively static page with one set of controls. It will demonstrate the use of the ASP.NET helper methods that deal with AJAX and partial page updates.

3. ASP.NET Core image gallery example

In this example, we will enhance the Products app used in previous tutorial to demonstrate the benefits of a partial page update in a large amount of data being pulled from a distant and slow server. We will also use the Bootstrap Model component to make updates to the page without having to load a separate view.

4. ASP.NET Core buttons app example

In this example of partial page updates, we will apply what we know to the game board example in the Buttons app. This will help you prepare for your own version of Minesweeper in the course milestones.

AJAX programming background

About AJAX

AJAX (Asynchronous JavaScript and XML) is a technique used to enable partial page updates. Without AJAX, any changes made to a webpage require the entire page to be refreshed, wasting bandwidth and causing a poor user experience, especially over a slow communication link.

With Ajax, web applications can send and retrieve data from a server asynchronously (in the background) without interfering with the display and behavior of the existing page. By decoupling the data interchange layer from the presentation layer, Ajax allows webpages to change content dynamically without the need to reload the entire page.

The “X” in AJAX has been replaced with the “J” of JSON data. However, “AJAJ” doesn’t sound as pretty as AJAX.

Ajax is not a single technology, but rather a group of technologies. HTML and CSS can be used in combination to mark up and style information. The webpage can then be modified by JavaScript to dynamically display—and allow the user to interact with—the new information.

jQuery is a JavaScript library that became extremely popular in part because of its AJAX function. The standard "fetch()" function within JavaScript can now be used to execute Ajax on webpages, allowing websites to load content onto the screen without refreshing the page. Ajax is not a new technology, or different language, just existing technologies used in new ways.

About jQuery

In the history of the development of JavaScript and its frameworks, jQuery stands as one of the most significant contributors to JavaScript’s success. Many developers now love to hate jQuery, but it continues to serve its original purpose well, which is to add visual and dynamic effects to an existing webpage. Newer frameworks, such as Angular, React, and Vue, have superseded jQuery’s capabilities to design and build complex web apps.

How did jQuery’s popularity grow? In early years of web development, 2000–2005, JavaScript was used to enhance the behavior of a page rather than drive its essential functions. JavaScript was designed to enable features such as animations, date pickers, and slideshows.

Additionally, JavaScript, an interpreted language, was not handled in a standardized manner on all browsers. Internet Explorer was notorious for interpreting instructions differently and thus using different commands than other browsers. At the time, Microsoft was dominate in the browser wars, which allowed them the flexibility to break with standard practices. JavaScript code needed to have if statements or case statements in many methods to check which browser was being used and to branch instructions accordingly. jQuery was the easiest method to build standardized code, where the browser differences were buried inside methods of its library. One set of instructions worked on all browsers.

jQuery also made JavaScript easier to write. Selecting and managing an element in the browser’s DOM is as easy as referencing a CSS id or class. For example, $(“#itemID”).hide( ) hides the itemID element.

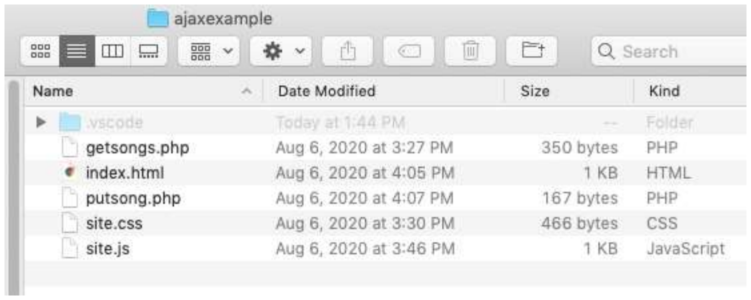
The need for jQuery has faded. Browsers today are pretty standardized. Standard JavaScript now includes many of the features that made jQuery so popular. Managing a Single Page Application (SPA) is better done with React.

However, jQuery still functions well as a support library for simple page operations. Bootstrap, the extremely popular CSS library, relies on jQuery to perform many of its functions. Bootstrap 5, released in 2020, however, drops the use of jQuery and relies completely on standard JavaScript. ASP.NET Core applications usually include Boostrap and jQuery as dependencies.

Setup the Server Environment

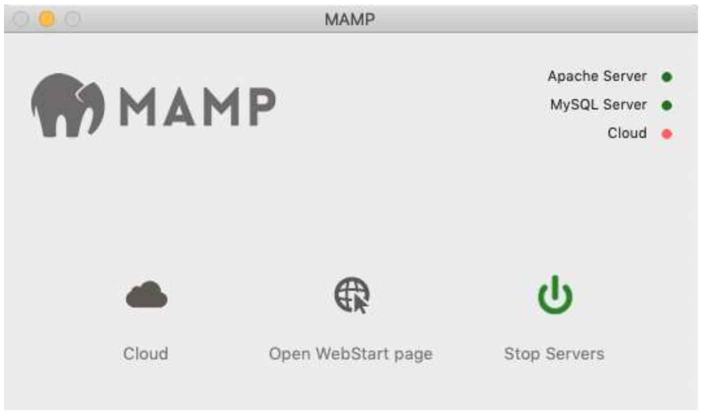
This coding example needs to have a PHP server and text editor.

1. Install **MAMP** (not pro version), WAMP, XAMP, or **UsbWebserver.net** on your computer and open the **htdocs** folder where we will put the files of our web app.

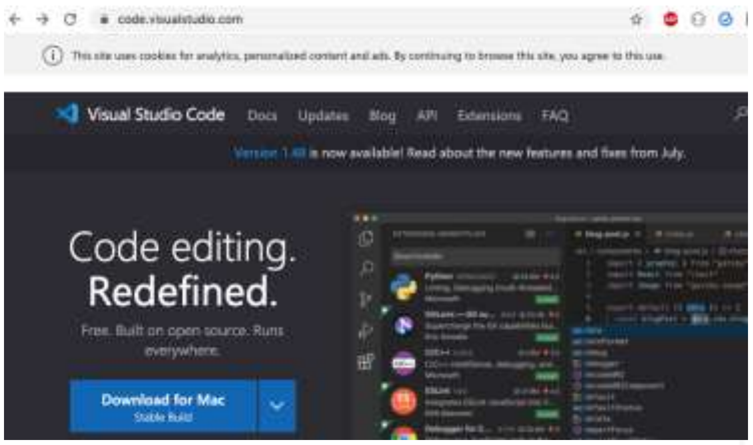


Name	Date Modified	Size	Kind
.vscode	Today at 1:44 PM	--	Folder
getsongs.php	Aug 6, 2020 at 3:27 PM	350 bytes	PHP
index.html	Aug 6, 2020 at 4:05 PM	1 KB	HTML
putsong.php	Aug 6, 2020 at 4:07 PM	167 bytes	PHP
site.css	Aug 6, 2020 at 3:30 PM	466 bytes	CSS
site.js	Aug 6, 2020 at 3:46 PM	1 KB	JavaScript

This shows that MAMP is installed and the Apache Server and MySQL server are both running.



2. Install a **text editor**. For this example, I will be using **Visual Studio Code**.



This site uses cookies for analytics, personalized content and ads. By continuing to browse this site, you agree to this use.

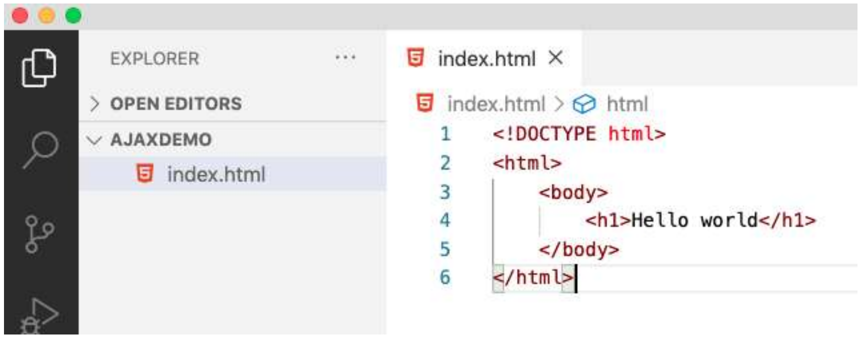
Version 1.48 is now available! Read about the new features and fixes from July.

Code editing. Redefined.

Free. Built on open source. Runs everywhere.

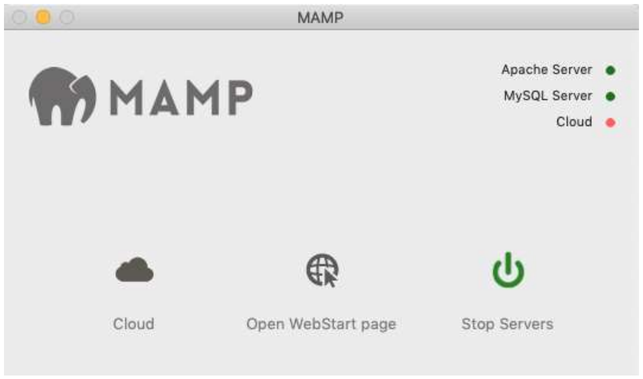
3. Create a new folder called **ajaxdemo** in the **htdocs** folder of the MAMP server.

4. Create an **index.html** file and put some placeholder code in it.

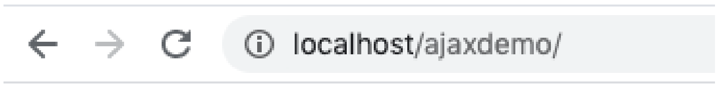


```
<!DOCTYPE html>
<html>
    <body>
        <h1>Hello world</h1>
    </body>
</html>
```

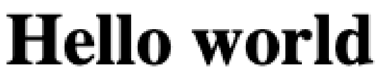
5. Test the new page on the server.
   a. Click Open WebStart page.



   b. Navigate to the URL **localhost/ajaxdemo**



6. **Capture** a **screenshot** of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.

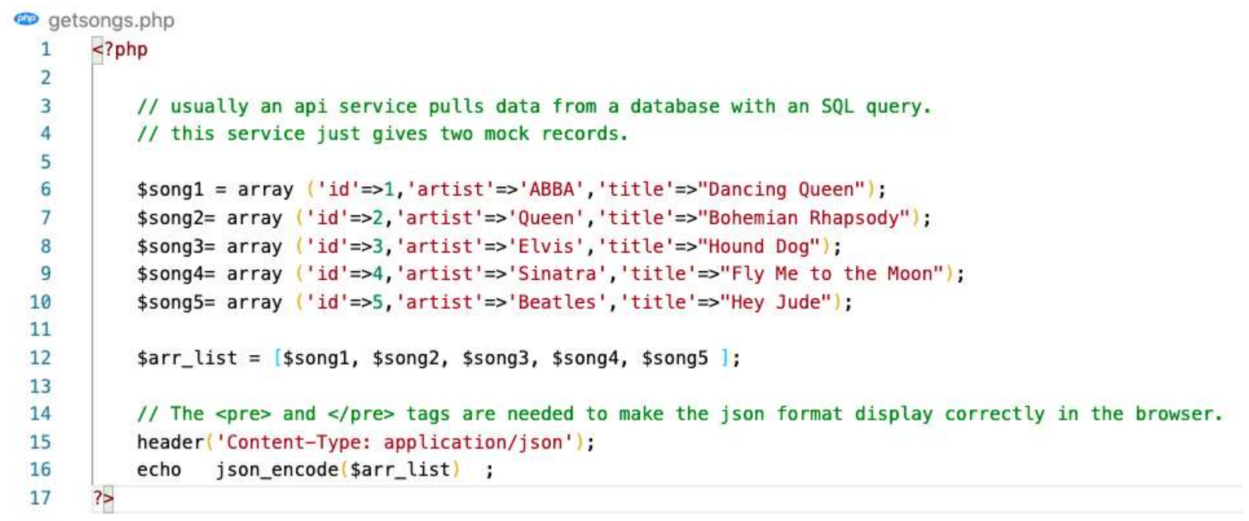


Hello world

Create a backend server

This example will create a mock server environment where two URLs provide some JSON-formatted data that a webpage can consume.

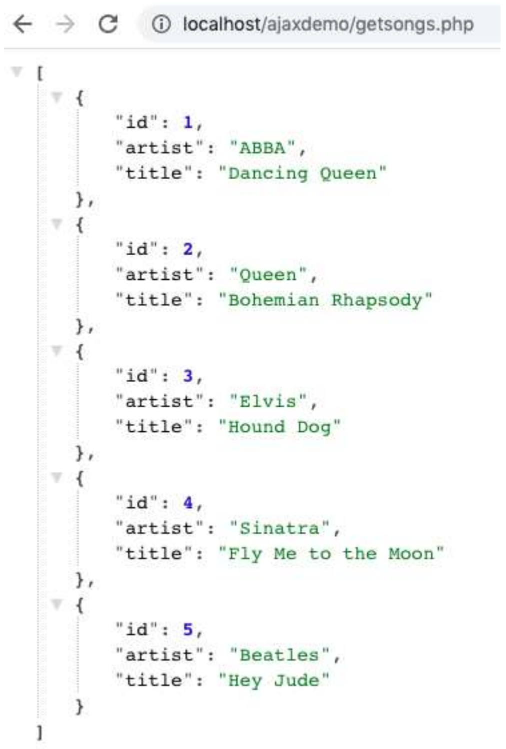
1. Create a new file in the htdocs folder called **getsongs.php**.
2. Code the file to echo a list of JSON-formatted song titles as shown here. The code generates an array of associative arrays. These arrays contain associated pairs of data.



getsongs.php

```php
<?php

    // usually an api service pulls data from a database with an SQL query.
    // this service just gives two mock records.

    $song1 = array ('id'=>1,'artist'=>'ABBA','title'=>"Dancing Queen");
    $song2= array ('id'=>2,'artist'=>'Queen','title'=>"Bohemian Rhapsody");
    $song3= array ('id'=>3,'artist'=>'Elvis','title'=>"Hound Dog");
    $song4= array ('id'=>4,'artist'=>'Sinatra','title'=>"Fly Me to the Moon");
    $song5= array ('id'=>5,'artist'=>'Beatles','title'=>"Hey Jude");

    $arr_list = [$song1, $song2, $song3, $song4, $song5 ];

    // The <pre> and </pre> tags are needed to make the json format display correctly in the browser.
    header('Content-Type: application/json');
    echo   json_encode($arr_list)  ;
?>
```

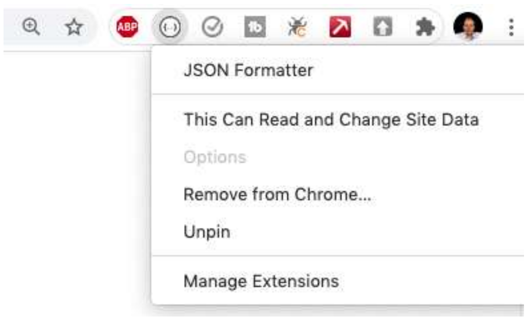
The browser should display the JSON data with the URL **localhost/ajexdemo/getsongs.php**



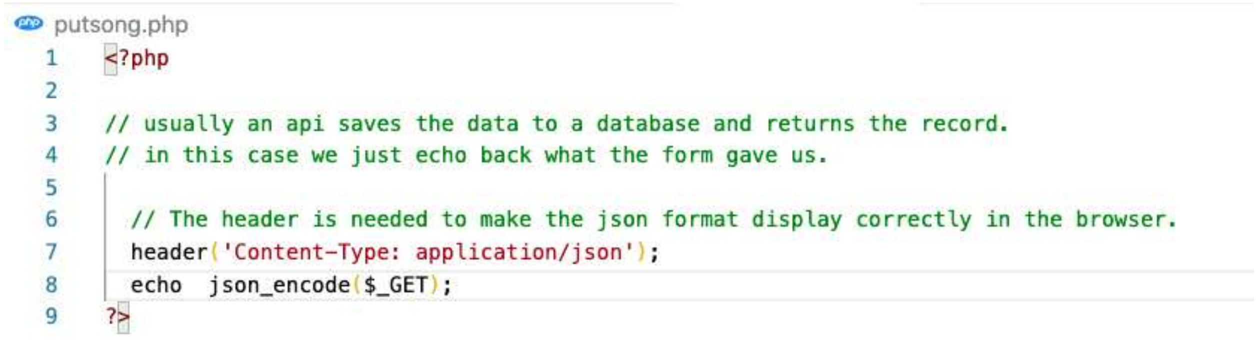
localhost/ajaxdemo/getsongs.php

```
[
  {
    "id": 1,
    "artist": "ABBA",
    "title": "Dancing Queen"
  },
  {
    "id": 2,
    "artist": "Queen",
    "title": "Bohemian Rhapsody"
  },
  {
    "id": 3,
    "artist": "Elvis",
    "title": "Hound Dog"
  },
  {
    "id": 4,
    "artist": "Sinatra",
    "title": "Fly Me to the Moon"
  },
  {
    "id": 5,
    "artist": "Beatles",
    "title": "Hey Jude"
  }
]
```

I am using a **browser plugin** called JSON formatter in Chrome to make the data more readable. The application will work equally well with or without "pretty" JSON formatting.



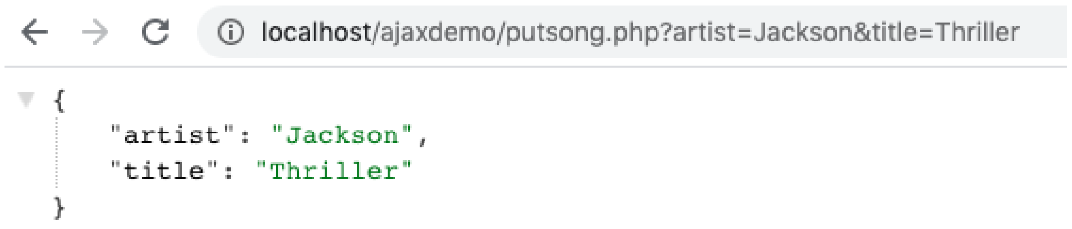
3. Add a new file called **putsong.php** that will echo back a response to a submit form command.



putsong.php

```php
<?php

// usually an api saves the data to a database and returns the record.
// in this case we just echo back what the form gave us.

  // The header is needed to make the json format display correctly in the browser.
  header('Content-Type: application/json');
  echo  json_encode($_GET);
?>
```

4. Test the file with a URL that includes some GET parameters in the address bar of the browser as shown. Notice that any parameter you supply is echoed back with its value.



```
localhost/ajaxdemo/putsong.php?artist=Jackson&title=Thriller
```

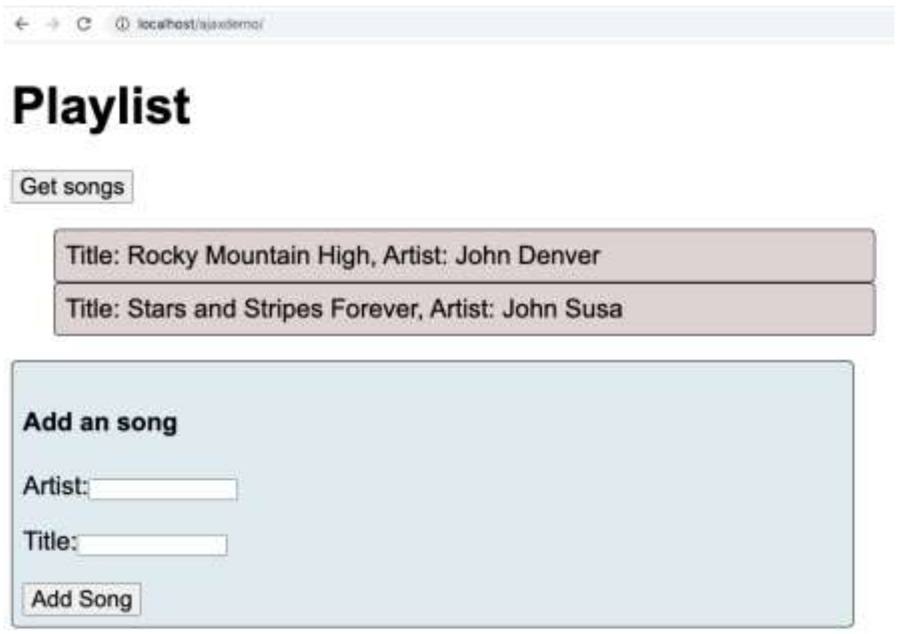
```
{
    "artist": "Jackson",
    "title": "Thriller"
}
```

Now our simple backend REST API is ready. These two files will serve as a very simple REST service for our front-end application. A real REST service would be able to query and store data in a database.

5. Capture a **screenshot** of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.

The Front End

Our application will consist of a single page that contains a list of songs followed by a data entry form. Eventually, the application will look like this.



localhost/ajaxdemo/

Playlist

Get songs

Title: Rocky Mountain High, Artist: John Denver

Title: Stars and Stripes Forever, Artist: John Susa

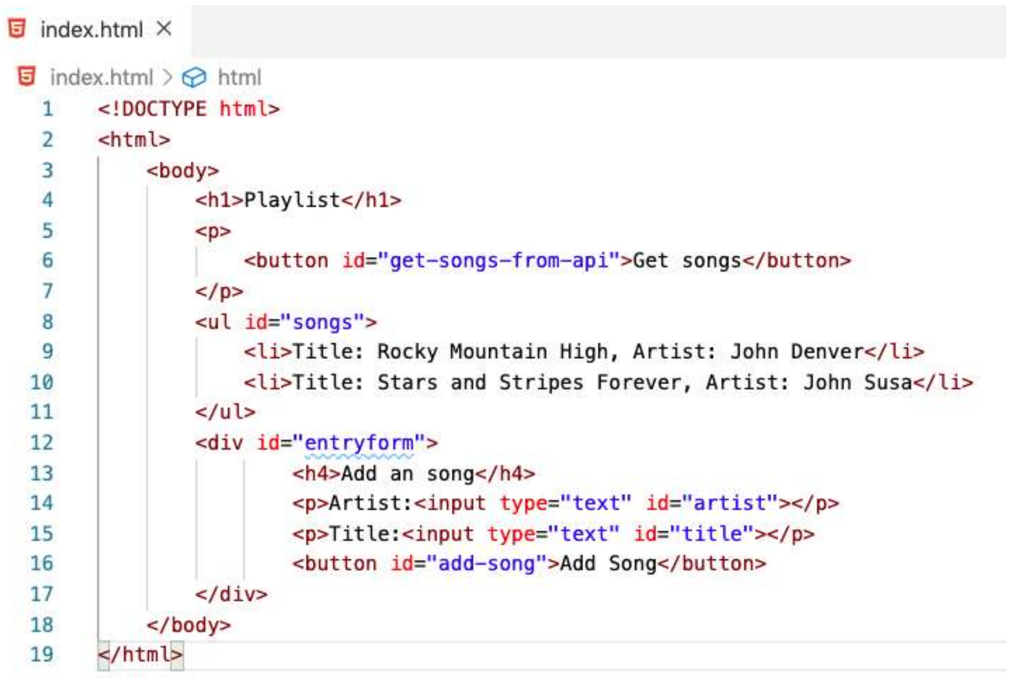
Add an song

Artist:

Title:

Add Song

1. Open the **index.html** file and replace the "hello world" contents with the following code.



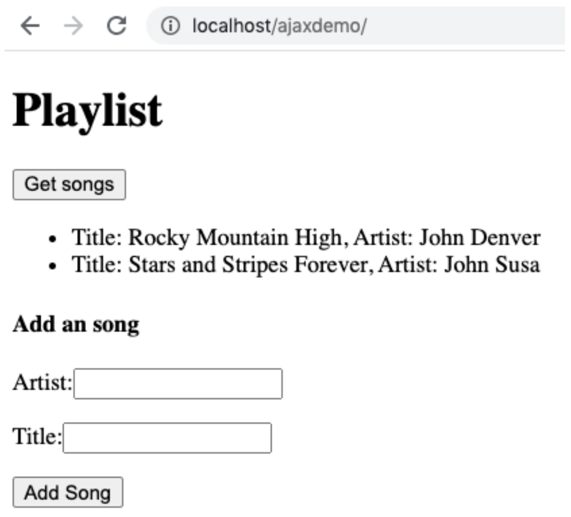
index.html ×

index.html > html

```
<!DOCTYPE html>
<html>
    <body>
        <h1>Playlist</h1>
        <p>
            <button id="get-songs-from-api">Get songs</button>
        </p>
        <ul id="songs">
            <li>Title: Rocky Mountain High, Artist: John Denver</li>
            <li>Title: Stars and Stripes Forever, Artist: John Susa</li>
        </ul>
        <div id="entryform">
                <h4>Add an song</h4>
                <p>Artist:<input type="text" id="artist"></p>
                <p>Title:<input type="text" id="title"></p>
                <button id="add-song">Add Song</button>
        </div>
    </body>
</html>
```

2. Open the URL at **localhost/ajaxdemo** to see the results. No CSS formatting is applied yet. No JavaScript code functions yet.

3. Capture a **screenshot** of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.



localhost/ajaxdemo/

Playlist

Get songs

- Title: Rocky Mountain High, Artist: John Denver
- Title: Stars and Stripes Forever, Artist: John Susa

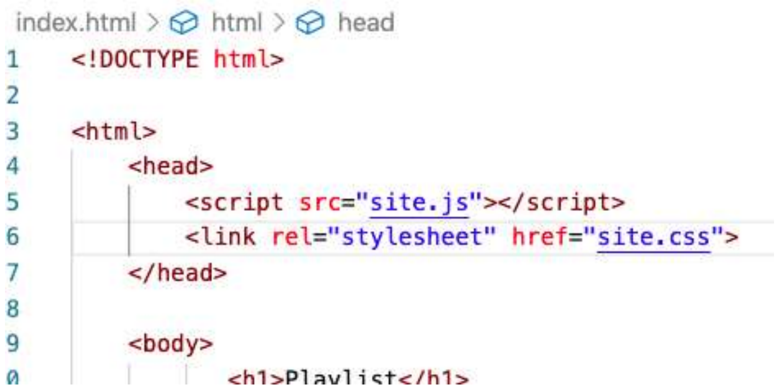
Add an song

Artist:

Title:

Add Song

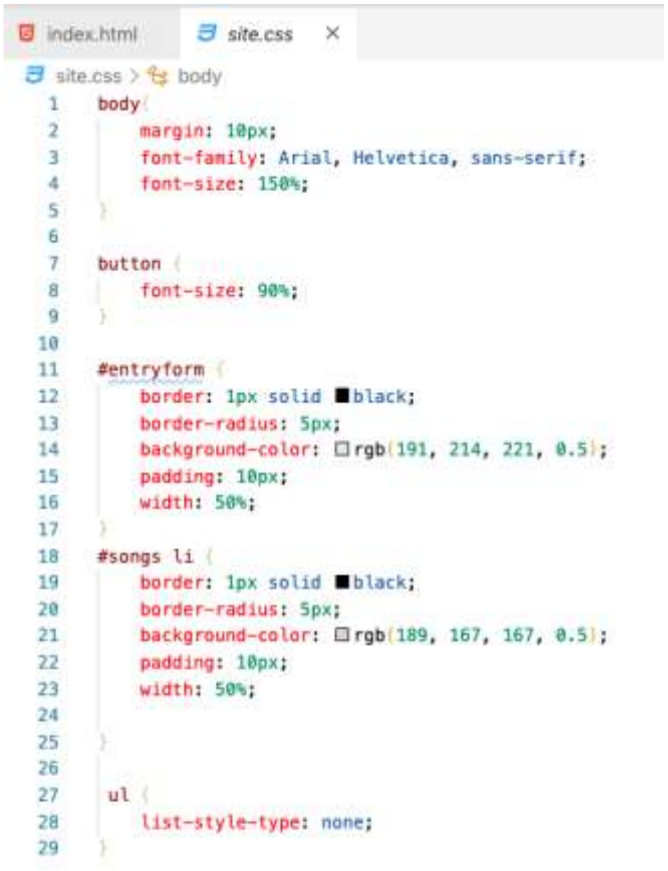
4. Add two **external file links**, one for the **CSS** styling, the other for a **JavaScript** file. Neither file exists yet, but we will create them soon. Notice that I put them in the <head> section of the page.



index.html > html > head

```
<!DOCTYPE html>

<html>
    <head>
        <script src="site.js"></script>
        <link rel="stylesheet" href="site.css">
    </head>

    <body>
        <h1>Playlist</h1>
```

5. Create a file **site.css** and add some styling. You can use my example if you wish or code your own styles. This application will work without any custom styles, but certainly looks nicer with a little bit of work.



```
body {
    margin: 10px;
    font-family: Arial, Helvetica, sans-serif;
    font-size: 150%;
}

button {
    font-size: 90%;
}

#entryform {
    border: 1px solid ■black;
    border-radius: 5px;
    background-color: □rgb(191, 214, 221, 0.5);
    padding: 10px;
    width: 50%;
}
#songs li {
    border: 1px solid ■black;
    border-radius: 5px;
    background-color: □rgb(189, 167, 167, 0.5);
    padding: 10px;
    width: 50%;

}

 ul {
    list-style-type: none;
}
```

6. Capture a **screenshot** of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.

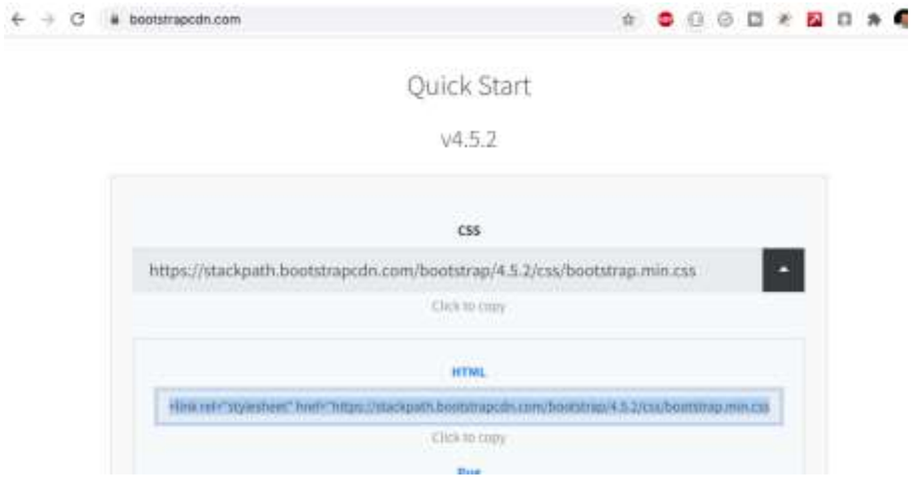
Install jQuery, Bootstrap CSS and Bootstrap JavaScript

There are two ways to install Bootstrap and jQuery on your page. The first is to download the necessary libraries and store them in a folder in the project. A second method is to **link directly to a file o**n an external server. Content Delivery Network (CDN) servers provide Bootstrap and hundreds of other libraries for free on their site.

1. Go to **boostrapcdn.com**

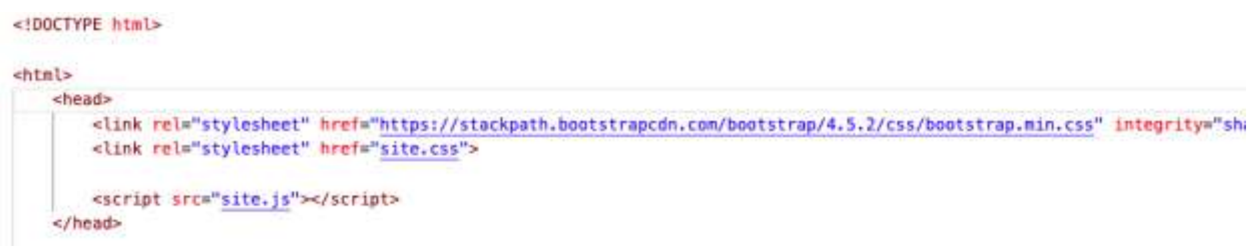
2. **Copy** the “html” version of the CSS link.

3. **Paste** the link into the index.html page. The proper location for a <style> section is in the <head> of a page.



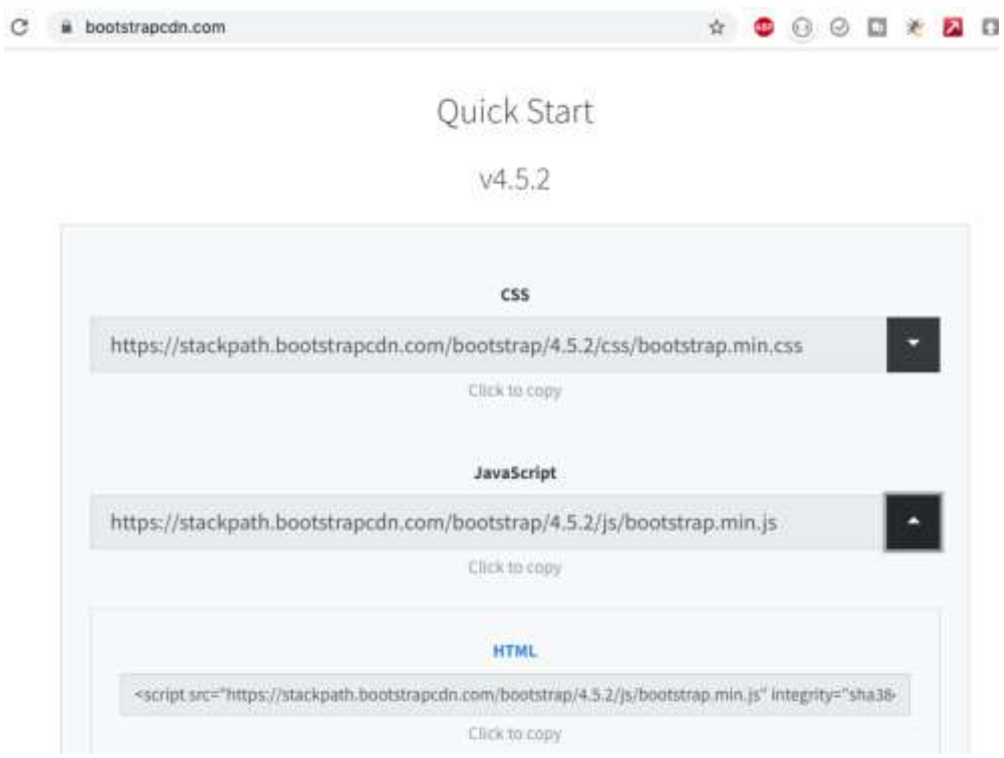
https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css



```
<!DOCTYPE html>

<html>
    <head>
        <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" integrity="sh
        <link rel="stylesheet" href="site.css">

        <script src="site.js"></script>
    </head>
```

4. Copy the "html" version of the Bootstrap JavaScript link.



bootstrapcdn.com

Quick Start

v4.5.2

CSS

https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css

Click to copy

JavaScript

https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js

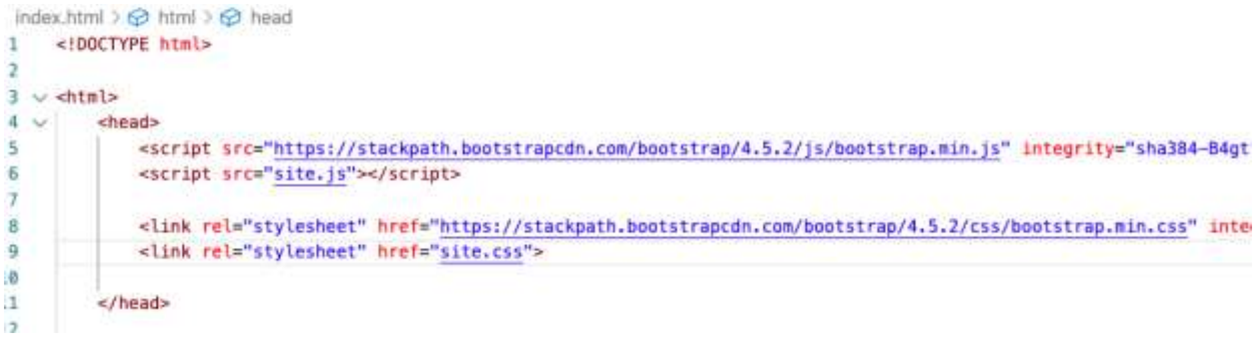
Click to copy

HTML

<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js" integrity="sha38

Click to copy

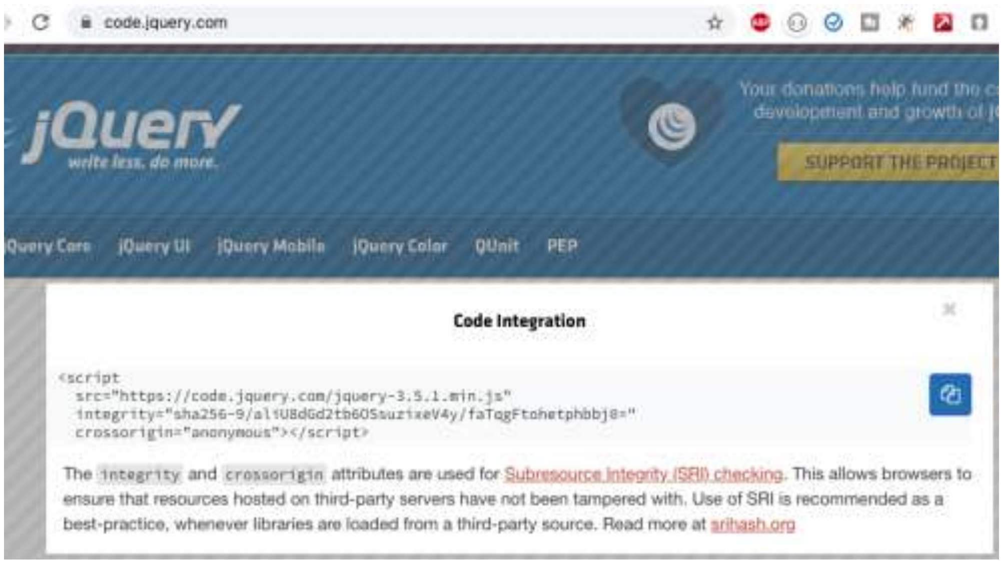
5. Past the link in **index.html** in the <head> section of the page.



index.html > html > head

```
<!DOCTYPE html>

<html>
    <head>
        <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js" integrity="sha384-B4gt
        <script src="site.js"></script>

        <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" inte
        <link rel="stylesheet" href="site.css">

    </head>
```

6. The **jQuery CDN** link is found on another website **code.jquery.com** where you can get the latest version, which at this time is 3.5.1. Copy the link.

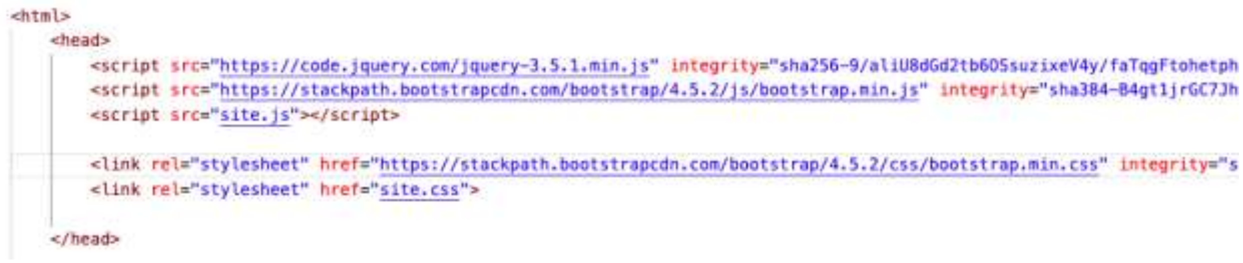


Code Integration

```
<script
  src="https://code.jquery.com/jquery-3.5.1.min.js"
  integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0="
  crossorigin="anonymous"></script>
```

The `integrity` and `crossorigin` attributes are used for Subresource Integrity (SRI) checking. This allows browsers to ensure that resources hosted on third-party servers have not been tampered with. Use of SRI is recommended as a best-practice, whenever libraries are loaded from a third-party source. Read more at srihash.org

7. Paste the link in **index.html** <head> section.

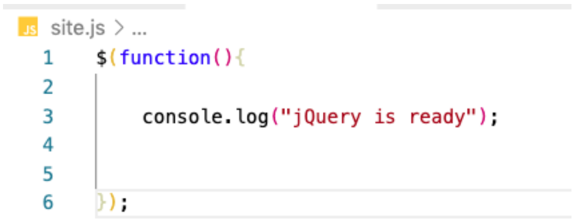


```
<html>
    <head>
        <script src="https://code.jquery.com/jquery-3.5.1.min.js" integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetph
        <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.min.js" integrity="sha384-B4gt1jrGC7Jh
        <script src="site.js"></script>

        <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" integrity="s
        <link rel="stylesheet" href="site.css">

    </head>
```

JavaScript Coding

Finally, we are ready to start the main lesson of this activity, the JavaScript.

1. Create a new file called **site.js** and place the following code in the file.



site.js > ...

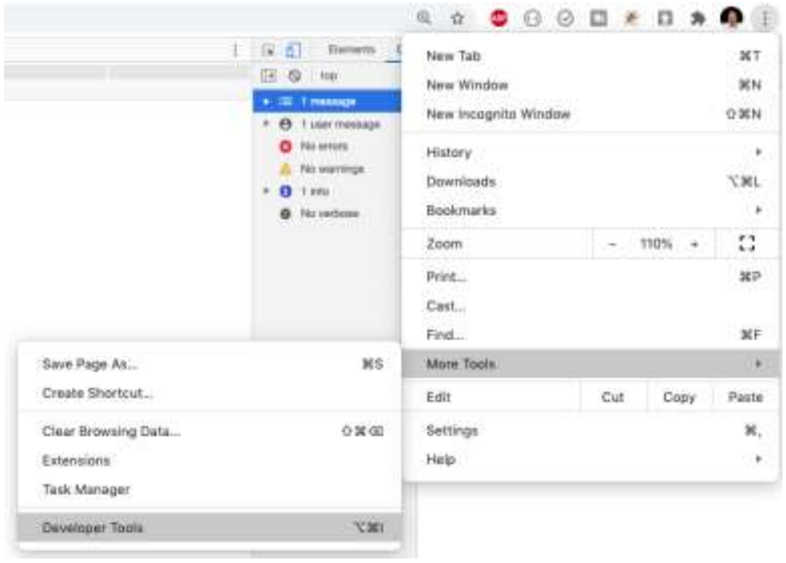
```
$(function(){

    console.log("jQuery is ready");


});
```

Explanation:

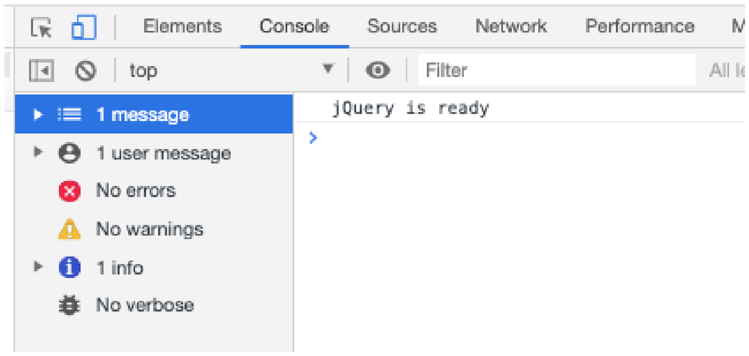
All jQuery functions begin with the $ character.

This function waits until the entire webpage is loaded before it starts to execute any code. The ready message will tell us if everything is loaded correctly.

2. Open the development tools in your browser. Chrome has a small menu where you can **choose More Tools > Developer Tools** to display the console.

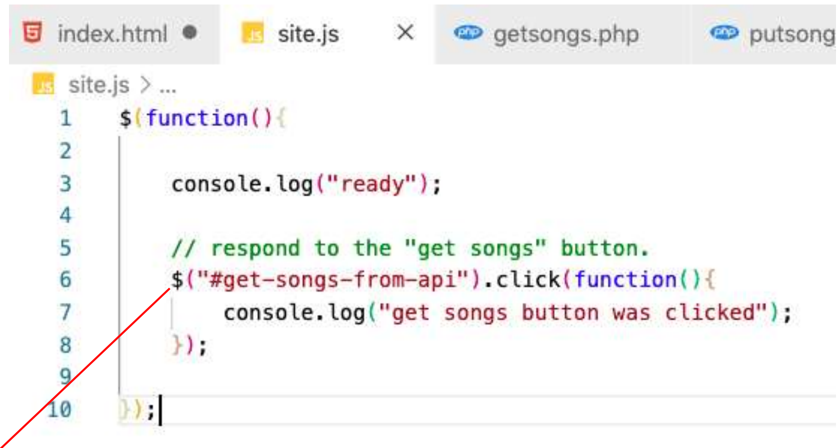


You should see a message “jQuery is ready” in the console when the application loads.



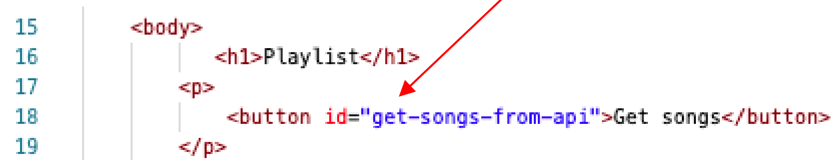
```
jQuery is ready
```

3. Add some more jQuery code that responds to a button click.



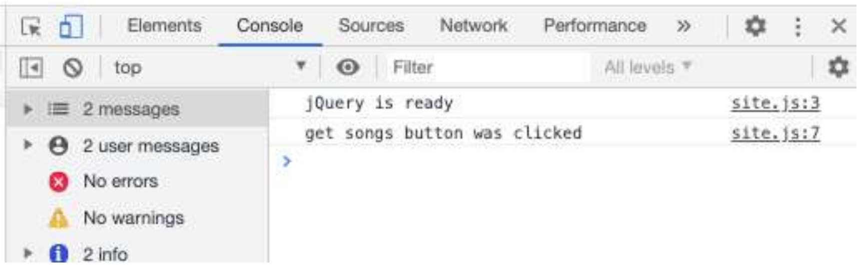
```
$(function(){

    console.log("ready");

    // respond to the "get songs" button.
    $("#get-songs-from-api").click(function(){
        console.log("get songs button was clicked");
    });

});
```

Notice that the button in the HTML code has **id = “get-songs-from-api”** as a property. The id value in the HTML code must match the selector used here.



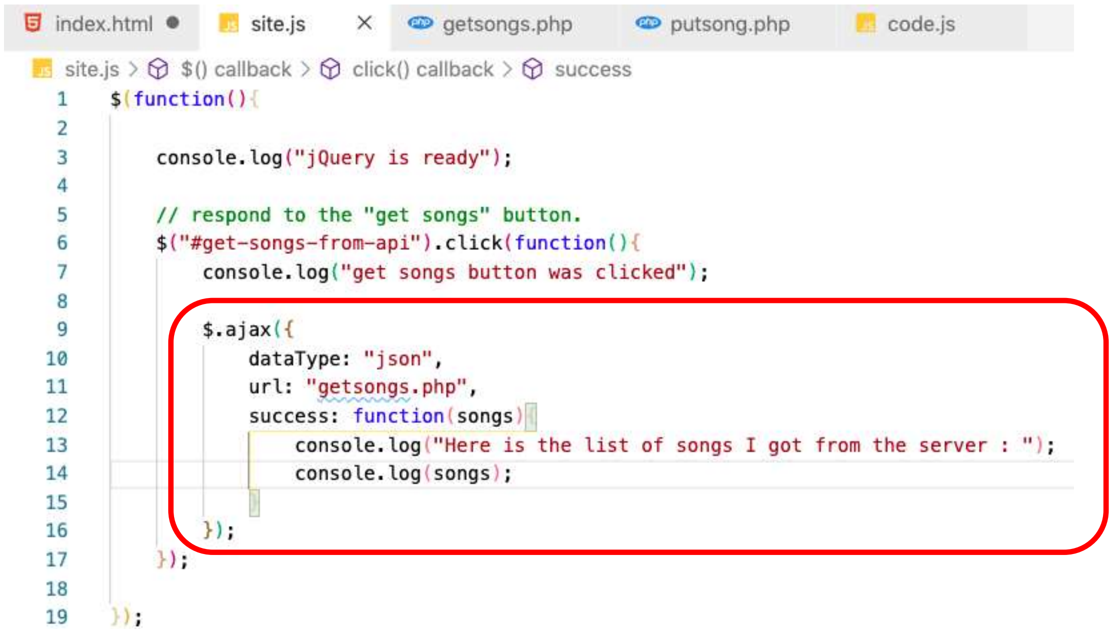
```
<body>
        <h1>Playlist</h1>
    <p>
      <button id="get-songs-from-api">Get songs</button>
    </p>
```

4. **Save and refresh** the page. You should see a console message when you click the “Get Songs” button.



```
jQuery is ready                          site.js:3
get songs button was clicked             site.js:7
```

5. Add some more code to **site.js** that performs an AJAX request to the server.



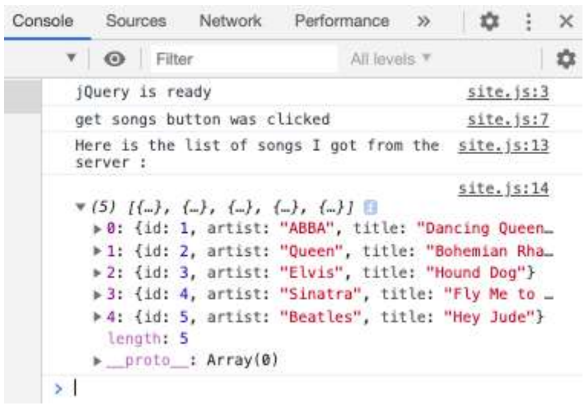
```
$(function(){

    console.log("jQuery is ready");

    // respond to the "get songs" button.
    $("#get-songs-from-api").click(function(){
        console.log("get songs button was clicked");

        $.ajax({
            dataType: "json",
            url: "getsongs.php",
            success: function(songs){
                console.log("Here is the list of songs I got from the server : ");
                console.log(songs);
            }
        });
    });

});
```

The $.ajax( ) function initiates a connection to a server. The values that are listed in the { cURLy brackets } are the parameters of the function call. The three parameters are:

- **datatype** – we are expecting to receive a json string.
- **URL** – we are requesting data from **getsongs.php** that we created earlier in this lesson.
- **success** – is the function that will execute once the server responds with some data, which we assigned to the variable **songs**.

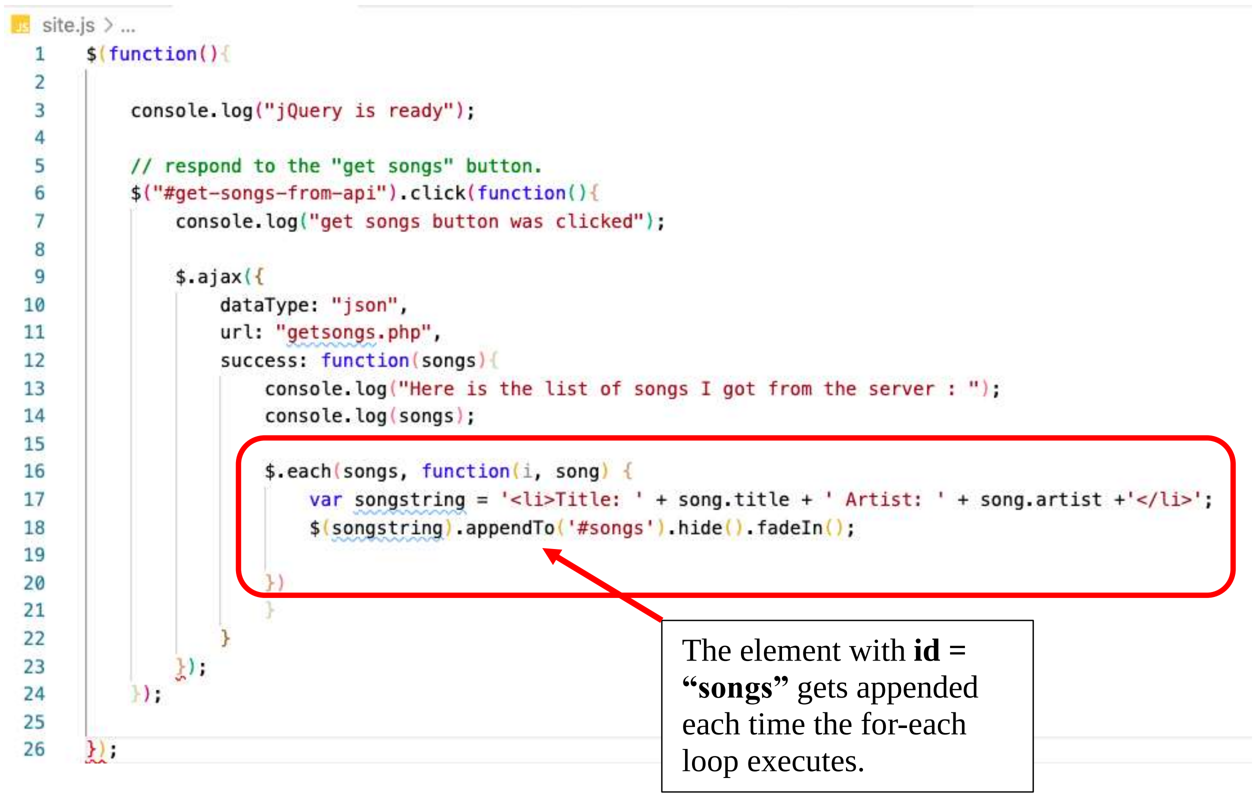
6. Save the file and refresh the page in the browser. We should see some new data in the browser console window. Expand the array of JSON objects to see all of the songs.

7. Capture a **screenshot** of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.



```
jQuery is ready                                      site.js:3
get songs button was clicked                         site.js:7
Here is the list of songs I got from the            site.js:13
server :
                                                     site.js:14
▼(5) [{…}, {…}, {…}, {…}, {…}]
  ▶0: {id: 1, artist: "ABBA", title: "Dancing Queen…
  ▶1: {id: 2, artist: "Queen", title: "Bohemian Rha…
  ▶2: {id: 3, artist: "Elvis", title: "Hound Dog"}
  ▶3: {id: 4, artist: "Sinatra", title: "Fly Me to …
  ▶4: {id: 5, artist: "Beatles", title: "Hey Jude"}
   length: 5
  ▶__proto__: Array(0)
>
```

8. Add some more code to **site.js** to update the contents of the page.



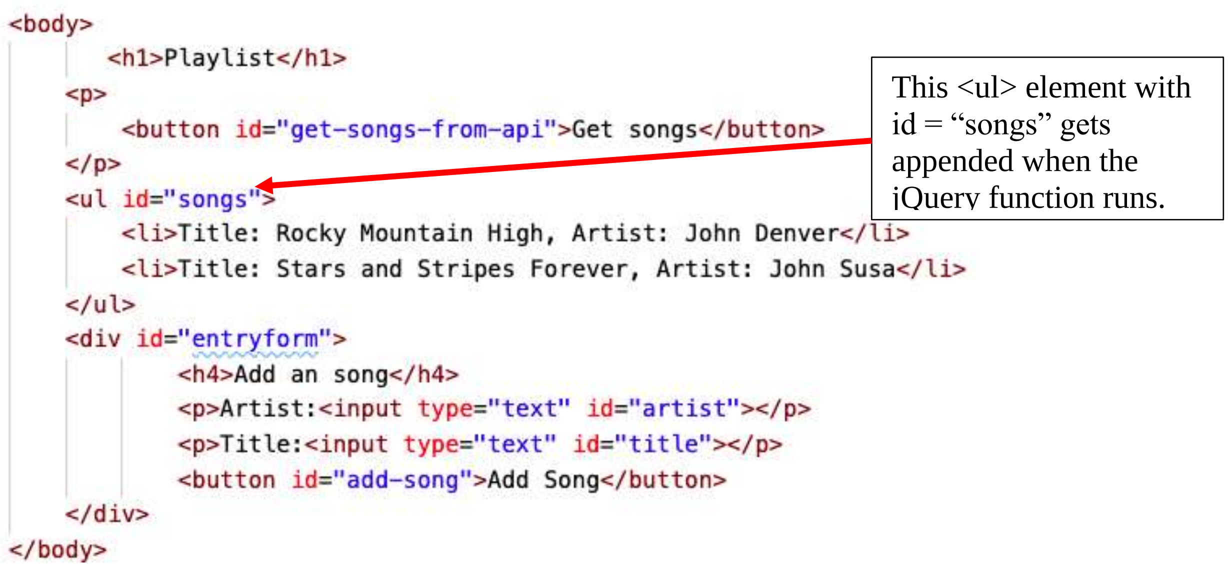
```
$(function(){

    console.log("jQuery is ready");

    // respond to the "get songs" button.
    $("#get-songs-from-api").click(function(){
        console.log("get songs button was clicked");

        $.ajax({
            dataType: "json",
            url: "getsongs.php",
            success: function(songs){
                console.log("Here is the list of songs I got from the server : ");
                console.log(songs);

                $.each(songs, function(i, song) {
                    var songstring = '<li>Title: ' + song.title + ' Artist: ' + song.artist +'</li>';
                    $(songstring).appendTo('#songs').hide().fadeIn();

                })
                }
            }
        });
    });

});
```

The element with **id = "songs"** gets appended each time the for-each loop executes.

The function **$.each** is a "**for each**" loop in jQuery. It could be read as:

"For each *song* with the index value of *i* in the list of *songs*, do the following function…"

The function creates a piece of HTML code that represents a new line in the song list and then appends it to the page element with the id = “songs”.



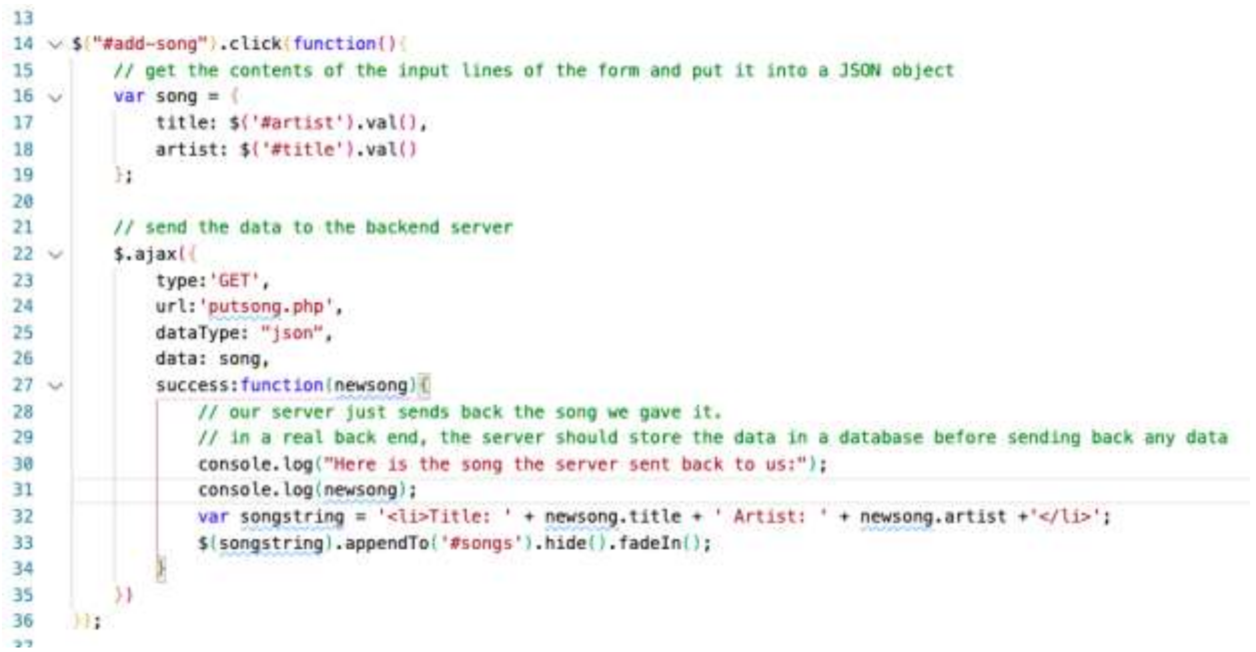
```
<body>
        <h1>Playlist</h1>
    <p>
        <button id="get-songs-from-api">Get songs</button>
    </p>
    <ul id="songs">
        <li>Title: Rocky Mountain High, Artist: John Denver</li>
        <li>Title: Stars and Stripes Forever, Artist: John Susa</li>
    </ul>
    <div id="entryform">
            <h4>Add an song</h4>
            <p>Artist:<input type="text" id="artist"></p>
            <p>Title:<input type="text" id="title"></p>
            <button id="add-song">Add Song</button>
    </div>
</body>
```

This <ul> element with id = “songs” gets appended when the jQuery function runs.

The .hide() and .fadeIn( ) functions are “**chained**” unto the end of the **appendTo** command, which provides some animation effect. You can omit these commands and the app will still function properly, but without the fade-in effect.

9. Capture a **screenshot** of the app at this stage. Put the image into a Microsoft Word document with a caption explaining what you have just demonstrated.

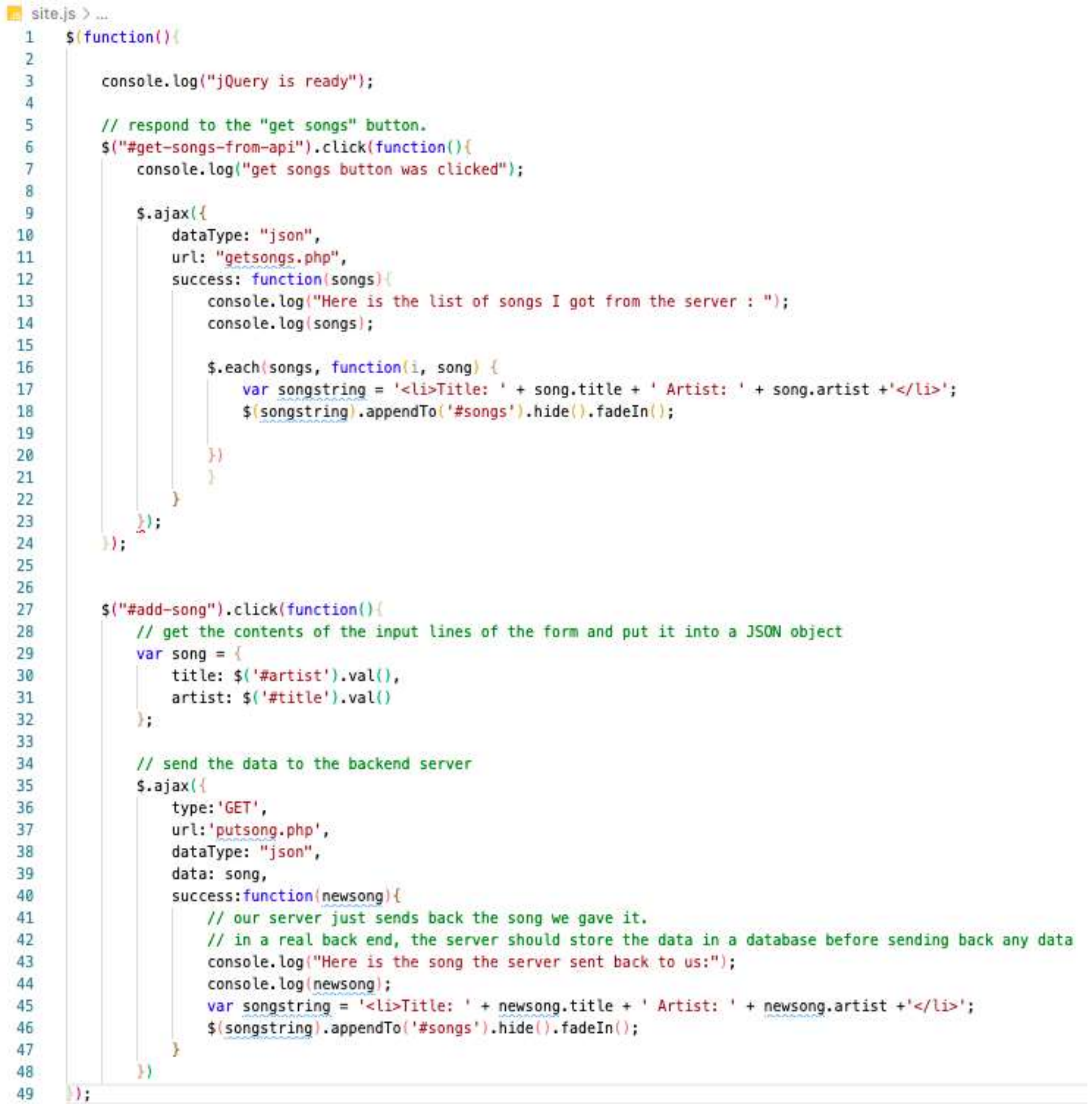
10. Add another function to **site.js** to handle the “**Add Song” button click**.



```
$("#add-song").click(function(){
    // get the contents of the input lines of the form and put it into a JSON object
    var song = {
        title: $('#artist').val(),
        artist: $('#title').val()
    };

    // send the data to the backend server
    $.ajax({
        type:'GET',
        url:'putsong.php',
        dataType: "json",
        data: song,
        success:function(newsong){
            // our server just sends back the song we gave it.
            // in a real back end, the server should store the data in a database before sending back any data
            console.log("Here is the song the server sent back to us:");
            console.log(newsong);
            var songstring = '<li>Title: ' + newsong.title + ' Artist: ' + newsong.artist +'</li>';
            $(songstring).appendTo('#songs').hide().fadeIn();
        }
    })
});
```

This function works almost like the first AJAX example. It uses an AJAX request to communicate with the server. The notable difference is that this function **sends** two values, **title** and **artist**, to the server and then receives a JSON message in return. As you will recall from earlier in the lesson, the **putsong.php** file simply repeats back to the client the information that was submitted. In this function, the return message is named **newsong**, which is parsed and appended to the element with the id of **songs**.

For a complete listing of **site.js**, see the following image.



```
site.js > ...
$(function(){

    console.log("jQuery is ready");

    // respond to the "get songs" button.
    $("#get-songs-from-api").click(function(){
        console.log("get songs button was clicked");

        $.ajax({
            dataType: "json",
            url: "getsongs.php",
            success: function(songs){
                console.log("Here is the list of songs I got from the server : ");
                console.log(songs);

                $.each(songs, function(i, song) {
                    var songstring = '<li>Title: ' + song.title + ' Artist: ' + song.artist +'</li>';
                    $(songstring).appendTo('#songs').hide().fadeIn();

                })
                }
            }
        });
    });


    $("#add-song").click(function(){
        // get the contents of the input lines of the form and put it into a JSON object
        var song = {
            title: $('#artist').val(),
            artist: $('#title').val()
        };

        // send the data to the backend server
        $.ajax({
            type:'GET',
            url:'putsong.php',
            dataType: "json",
            data: song,
            success:function(newsong){
                // our server just sends back the song we gave it.
                // in a real back end, the server should store the data in a database before sending back any data
                console.log("Here is the song the server sent back to us:");
                console.log(newsong);
                var songstring = '<li>Title: ' + newsong.title + ' Artist: ' + newsong.artist +'</li>';
                $(songstring).appendTo('#songs').hide().fadeIn();
            }
        })
});
```

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 source code created in this example.

Part 2 partial page updates with AJAX

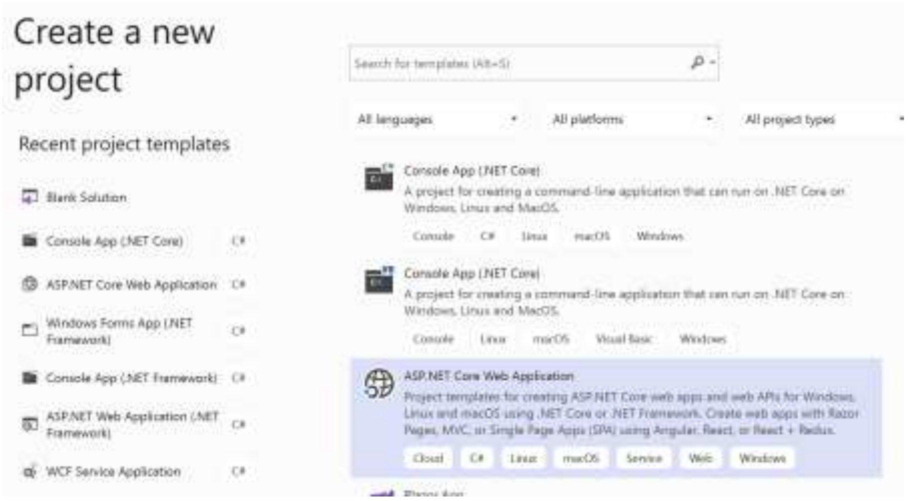
<u>Goal and Directions:</u>

In this activity, you will learn:

- how to build Razor Forms
- to create AJAX enabled Razor Forms
- to create Partial Views
- to perform Partial Page Updates

Complete the following tasks for this activity:

1. Create a new C# ASP.**NET Core Web Application project.**



Create a new project

Recent project templates

- Blank Solution
- Console App (.NET Core) C#
- ASP.NET Core Web Application C#
- Windows Forms App (.NET Framework) C#
- Console App (.NET Framework) C#
- ASP.NET Web Application (.NET Framework) C#
- WCF Service Application C#

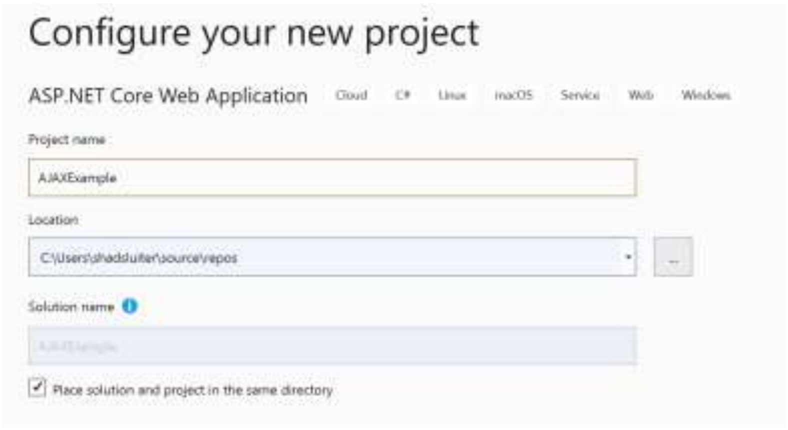
Search for templates (Alt+S)

All languages | All platforms | All project types

Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
Console C# Linux macOS Windows

Console App (.NET Core)
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.
Console Linux macOS Visual Basic Windows

ASP.NET Core Web Application
Project templates for creating ASP.NET Core web apps and web APIs for Windows, Linux and macOS using .NET Core or .NET Framework. Create web apps with Razor Pages, MVC, or Single Page Apps (SPA) using Angular, React, or React + Redux.
Cloud C# Linux macOS Service Web Windows



Configure your new project

ASP.NET Core Web Application   Cloud C# Linux macOS Service Web Windows

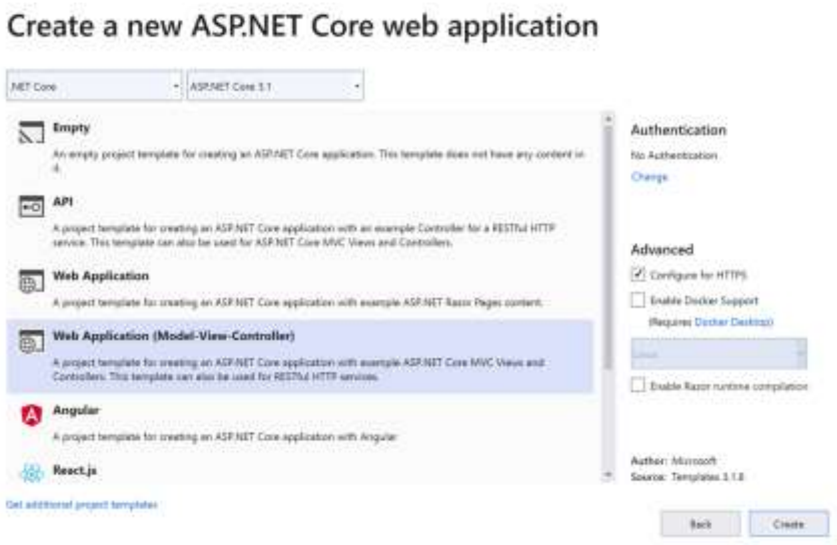
Project name
AJAXExample

Location
C:\Users\shadsluiter\source\repos

Solution name
AJAXExample

[✓] Place solution and project in the same directory

1. Choose **Web Application (Model-View-Controller)** project with No Authentication.



Create a new ASP.NET Core web application

.NET Core | ASP.NET Core 3.1

Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Web Application
A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

Web Application (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

Angular
A project template for creating an ASP.NET Core application with Angular

React.js

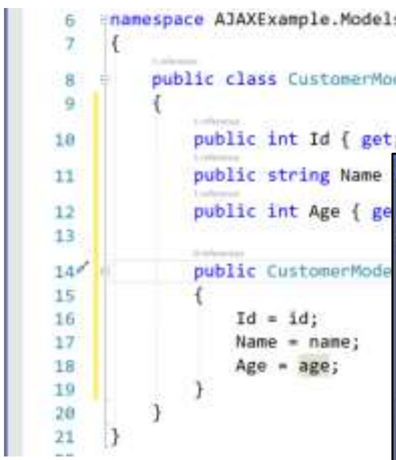
Get additional project templates

Authentication
No Authentication
Change

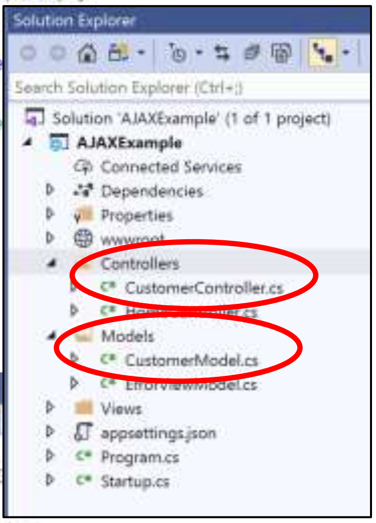
Advanced
[✓] Configure for HTTPS
[ ] Enable Docker Support
(Requires Docker Desktop)
[ ] Enable Razor runtime compilation

Author: Microsoft
Source: Templates 3.1.8

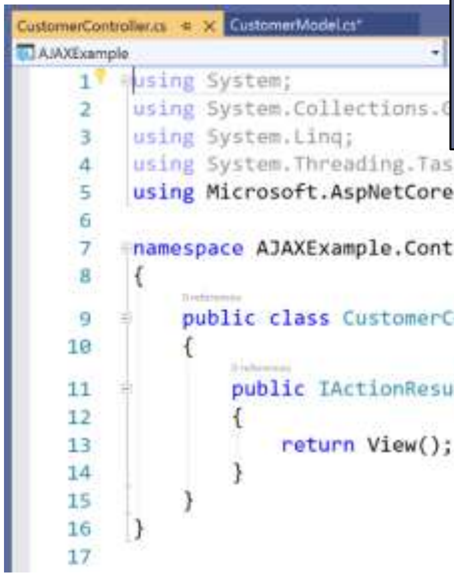
2. Create a **Model** named **CustomerModel** with the following properties. Also create a non-default constructor that initializes all the class properties:
   int ID
   string Name
   int Age



```
namespace AJAXExample.Models
{
    public class CustomerModel
    {
        public int Id { get; set; }
        public string Name
        public int Age { ge

        public CustomerMode
        {
            Id = id;
            Name = name;
            Age = age;
        }
    }
}
```

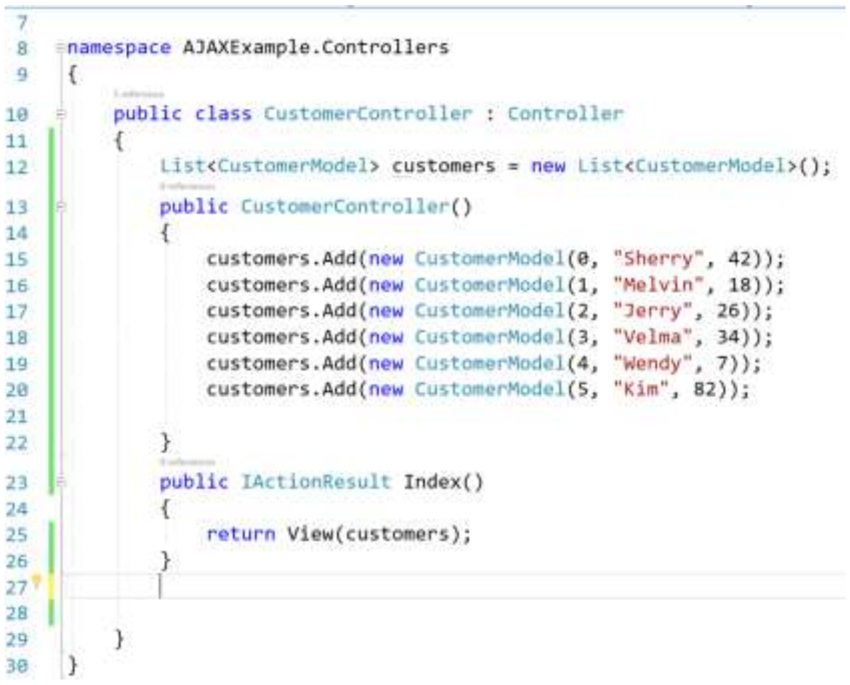


3. Create a default **Customer Controller**.



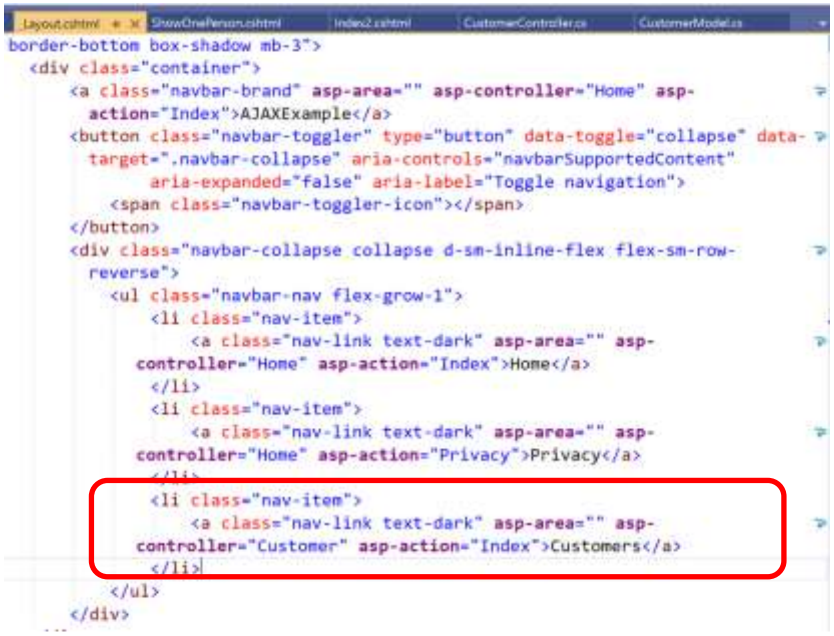
```
using System;
using System.Collections.C
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace AJAXExample.Controllers
{
    public class CustomerController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

4. In the **Customer Controller**, add a **constructor** and a **list of sample data** for the app to work with. Add **customers** as a parameter to be passed to the index view.



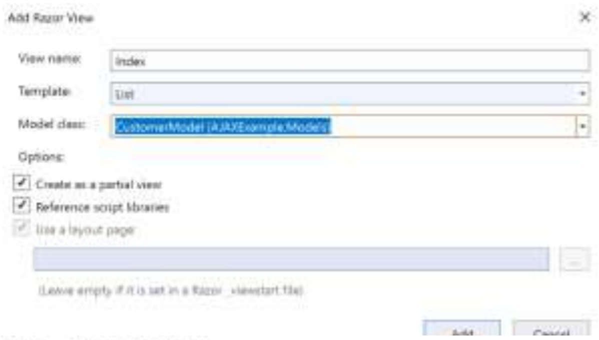
```
namespace AJAXExample.Controllers
{
    public class CustomerController : Controller
    {
        List<CustomerModel> customers = new List<CustomerModel>();
        public CustomerController()
        {
            customers.Add(new CustomerModel(0, "Sherry", 42));
            customers.Add(new CustomerModel(1, "Melvin", 18));
            customers.Add(new CustomerModel(2, "Jerry", 26));
            customers.Add(new CustomerModel(3, "Velma", 34));
            customers.Add(new CustomerModel(4, "Wendy", 7));
            customers.Add(new CustomerModel(5, "Kim", 82));

        }
        public IActionResult Index()
        {
            return View(customers);
        }


    }
}
```

5. Add a new URL to the **navbar** component. Open Views > Shared > _Layout.cshtml and add the following link.



```
border-bottom box-shadow mb-3">
  <div class="container">
      <a class="navbar-brand" asp-area="" asp-controller="Home" asp-
        action="Index">AJAXExample</a>
      <button class="navbar-toggler" type="button" data-toggle="collapse" data-
        target=".navbar-collapse" aria-controls="navbarSupportedContent"
              aria-expanded="false" aria-label="Toggle navigation">
          <span class="navbar-toggler-icon"></span>
      </button>
      <div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-
        reverse">
          <ul class="navbar-nav flex-grow-1">
              <li class="nav-item">
                  <a class="nav-link text-dark" asp-area="" asp-
            controller="Home" asp-action="Index">Home</a>
              </li>
              <li class="nav-item">
                  <a class="nav-link text-dark" asp-area="" asp-
            controller="Home" asp-action="Privacy">Privacy</a>
              </li>
              <li class="nav-item">
                  <a class="nav-link text-dark" asp-area="" asp-
            controller="Customer" asp-action="Index">Customers</a>
              </li>
          </ul>
      </div>
```

Next we will **create a view** to display all of the customers.

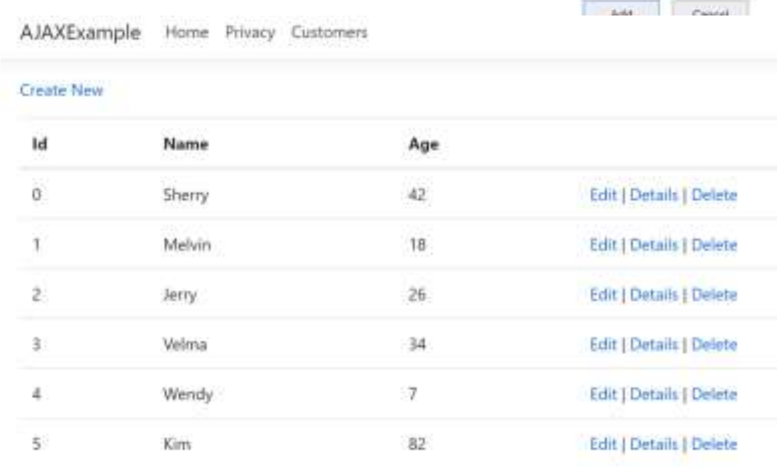


View name: Index
Template: List
Model class: CustomerModel (AJAXExample.Models)
Options:
[✓] Create as a partial view
[✓] Reference script libraries
[✓] Use a layout page
(Leave empty if it is set in a Razor _viewstart file)

6. Right-click in the **Index** method and choose **Add > View**.

7. Add a **List** view using the **Customer Model** as a data type.

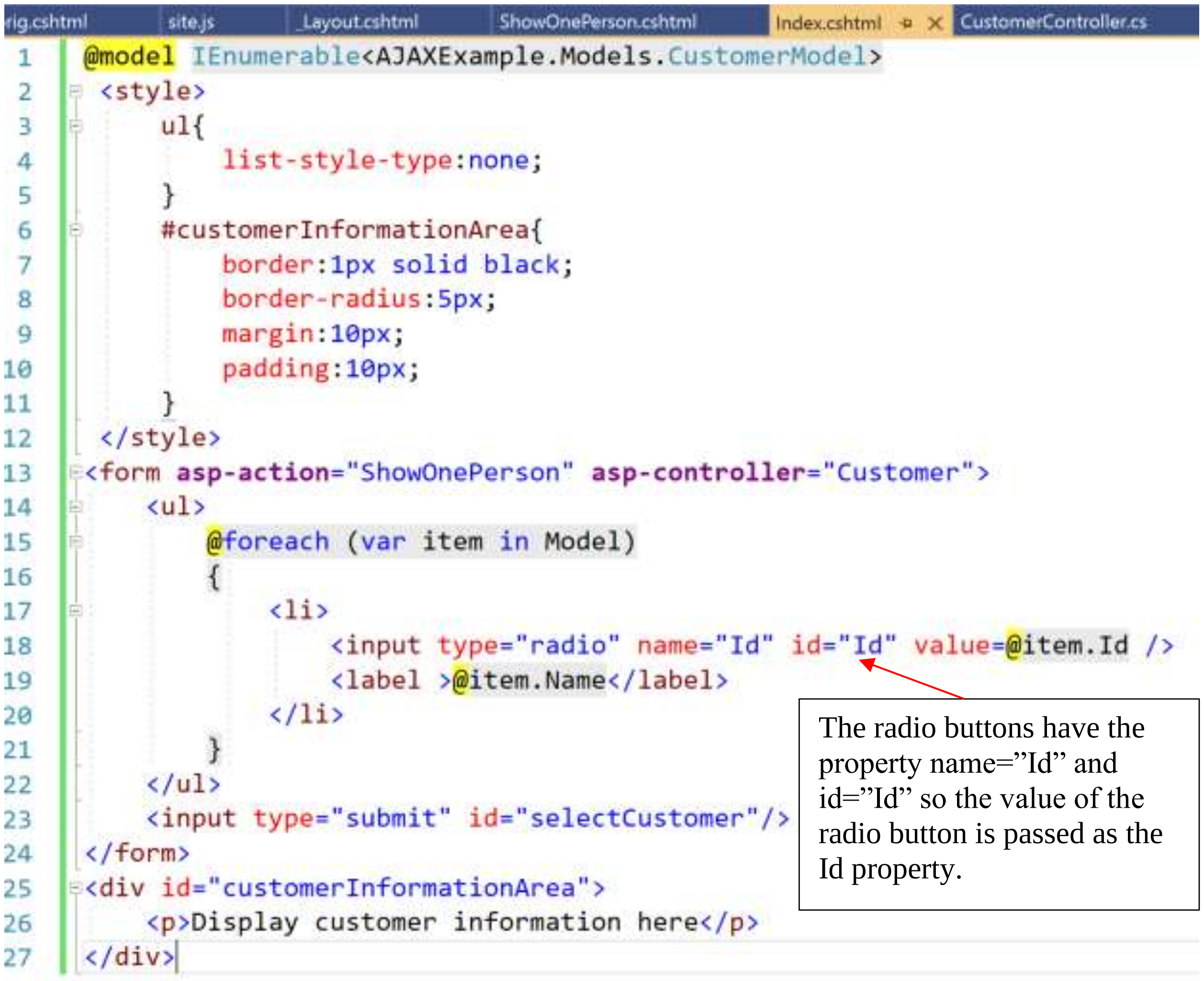
8. Run the program and view the customers in the new view



Id	Name	Age	
0	Sherry	42	Edit | Details | Delete
1	Melvin	18	Edit | Details | Delete
2	Jerry	26	Edit | Details | Delete
3	Velma	34	Edit | Details | Delete
4	Wendy	7	Edit | Details | Delete
5	Kim	82	Edit | Details | Delete

9. 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.

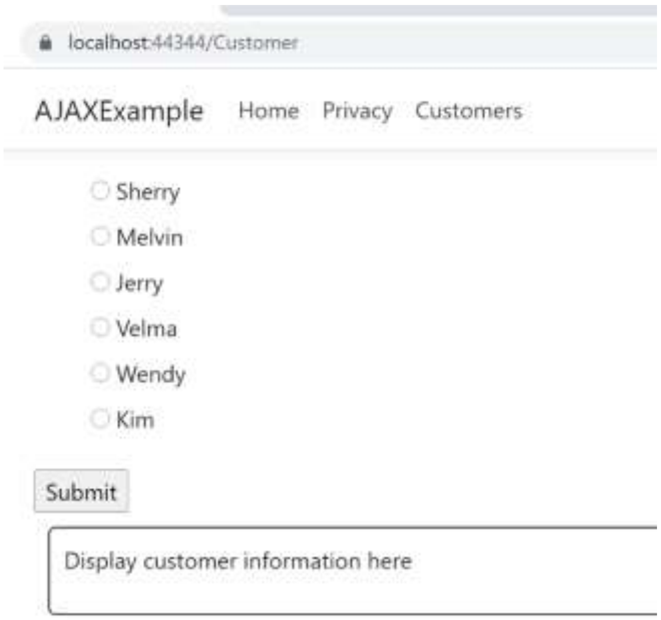
10. Delete most of the HTML and Razor code in the **index** view and replace it with the following:



```
@model IEnumerable<AJAXExample.Models.CustomerModel>
 <style>
     ul{
         list-style-type:none;
     }
     #customerInformationArea{
         border:1px solid black;
         border-radius:5px;
         margin:10px;
         padding:10px;
     }
 </style>
<form asp-action="ShowOnePerson" asp-controller="Customer">
    <ul>
        @foreach (var item in Model)
        {
            <li>
                <input type="radio" name="Id" id="Id" value=@item.Id />
                <label >@item.Name</label>
            </li>
        }
    </ul>
    <input type="submit" id="selectCustomer"/>
</form>
<div id="customerInformationArea">
    <p>Display customer information here</p>
</div>
```

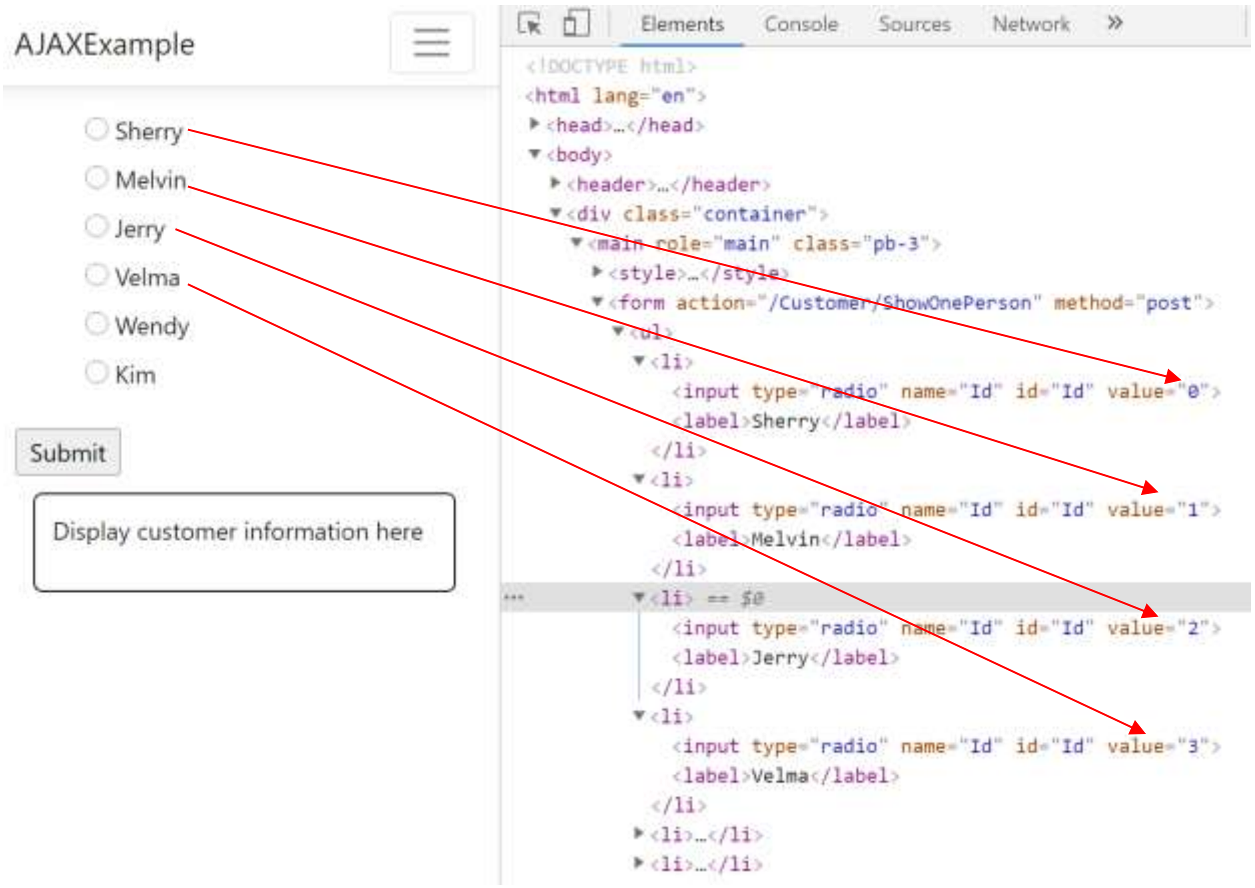
The radio buttons have the property name="Id" and id="Id" so the value of the radio button is passed as the Id property.

11. Run the app, you should get a page that looks like this.



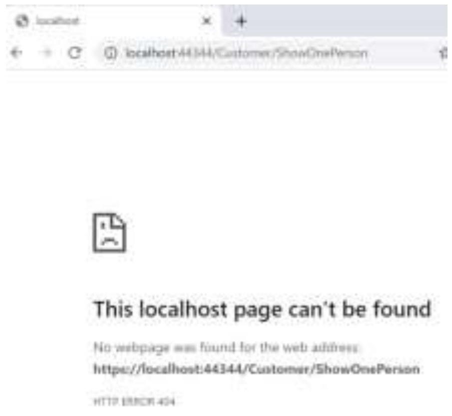
12. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

Inspect the HTML code and notice that the value of each radio button corresponds to the Id number of the customer.



```
<!DOCTYPE html>
<html lang="en">
▶<head>…</head>
▼<body>
  ▶<header>…</header>
  ▼<div class="container">
    ▼<main role="main" class="pb-3">
      ▶<style>…</style>
      ▼<form action="/Customer/ShowOnePerson" method="post">
        ▼<ul>
          ▼<li>
              <input type="radio" name="Id" id="Id" value="0">
              <label>Sherry</label>
            </li>
          ▼<li>
              <input type="radio" name="Id" id="Id" value="1">
              <label>Melvin</label>
            </li>
          ▼<li> == $0
              <input type="radio" name="Id" id="Id" value="2">
              <label>Jerry</label>
            </li>
          ▼<li>
              <input type="radio" name="Id" id="Id" value="3">
              <label>Velma</label>
            </li>
          ▶<li>…</li>
          ▶<li>…</li>
```

The submit button leads to **ShowOnePerson**, which doesn't exist yet, so a **404 error** is generated.

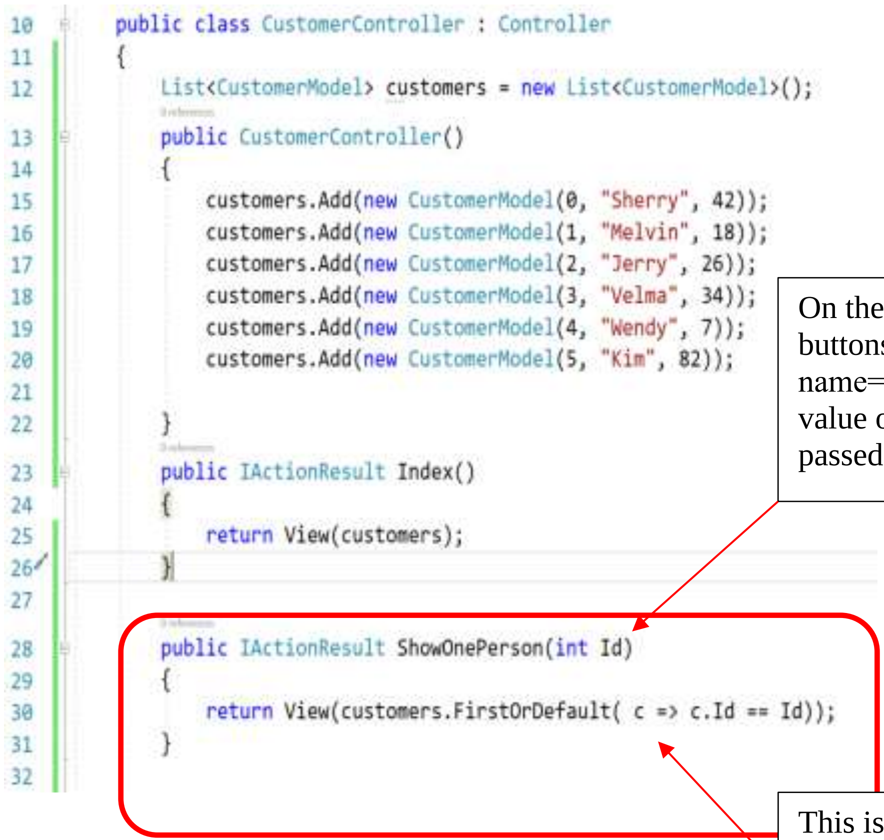


This localhost page can't be found

No webpage was found for the web address: https://localhost:44344/Customer/ShowOnePerson

HTTP ERROR 404

13. Add a new method to the **CustomerController** to handle the form's action method.

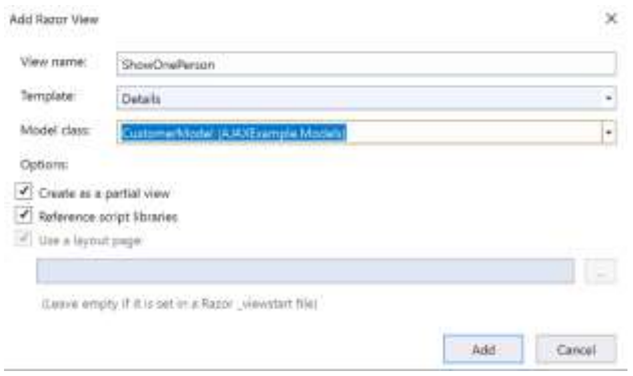


```
public class CustomerController : Controller
{
    List<CustomerModel> customers = new List<CustomerModel>();

    public CustomerController()
    {
        customers.Add(new CustomerModel(0, "Sherry", 42));
        customers.Add(new CustomerModel(1, "Melvin", 18));
        customers.Add(new CustomerModel(2, "Jerry", 26));
        customers.Add(new CustomerModel(3, "Velma", 34));
        customers.Add(new CustomerModel(4, "Wendy", 7));
        customers.Add(new CustomerModel(5, "Kim", 82));

    }

    public IActionResult Index()
    {
        return View(customers);
    }

    public IActionResult ShowOnePerson(int Id)
    {
        return View(customers.FirstOrDefault( c => c.Id == Id));
    }

```

On the form, the radio buttons have the property name="Id" and id="Id" so the value of the radio button is passed as the Id property.

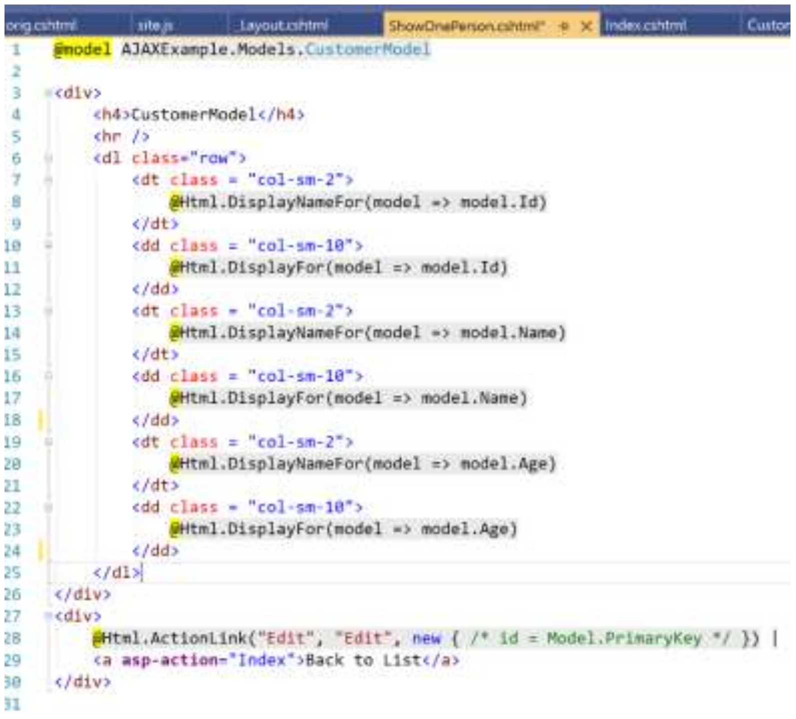
This is a C# linq function that selects the first item in the customers list whose Id property matches with the value Id.

14. Right-click inside the **ShowOnePerson** method and add a **new View**. Use the **Details** template and select the **CustomerModel** for the Model class:



View name: ShowOnePerson
Template: Details
Model class: CustomerModel (AJAXExample.Models)
Options:
✓ Create as a partial view
✓ Reference script libraries
✓ Use a layout page:
(Leave empty if it is set in a Razor _viewstart file)

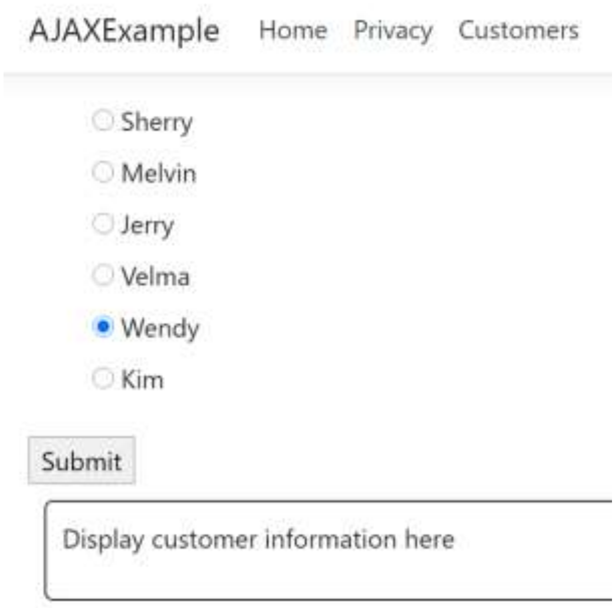
You should see the HTML and Razor code generated by Visual Studio that will display the **details of one customer**.



```
@model AJAXExample.Models.CustomerModel

<div>
    <h4>CustomerModel</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Id)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Id)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Name)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Name)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Age)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Age)
        </dd>
    </dl>
</div>
<div>
    @Html.ActionLink("Edit", "Edit", new { /* id = Model.PrimaryKey */ }) |
    <a asp-action="Index">Back to List</a>
</div>
```

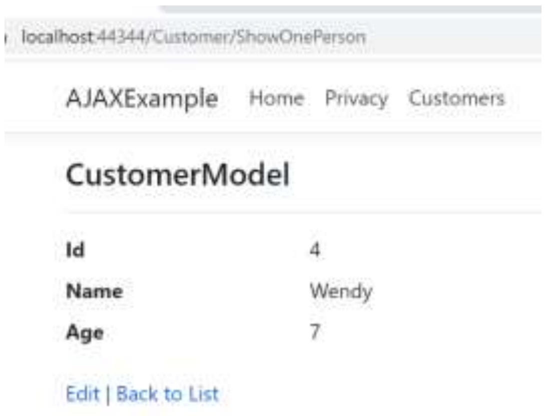
15. Run the program. You should be able to select one customer's radio button and show their details on a separate view.

16. 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.



Sherry [ ]
Melvin [ ]
Jerry [ ]
Velma [ ]
[✓ Wendy]
Kim [ ]

Display customer information here



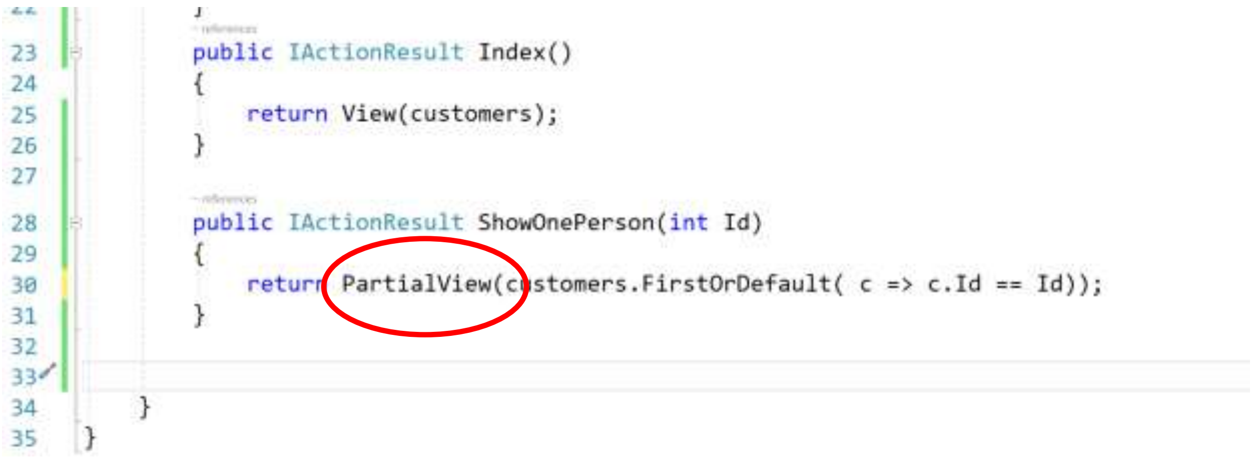
CustomerModel

Id	4
Name	Wendy
Age	7

AJAX for partial page updates

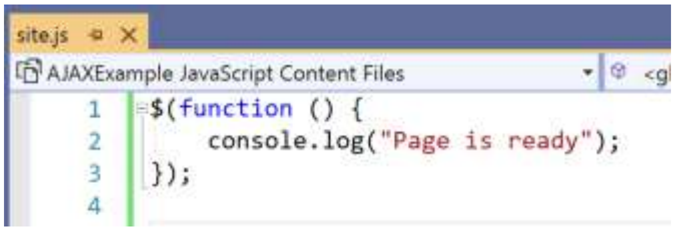
We have completed a **standard Model View Controller design** that allows us to display a different view on each button click. Now we are going to apply JavaScript code that will display the **ShowOnePerson** view *inside the Index page* **without having to perform a full-page refresh**.

1. Change the **CustomerController** to return a **PartialView** instead of a View.



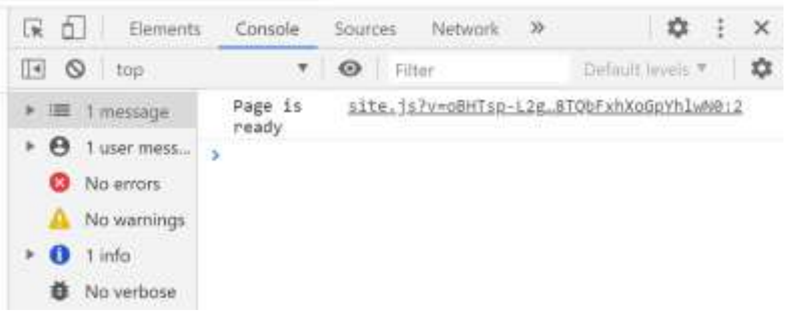
```
        public IActionResult Index()
        {
            return View(customers);
        }

        public IActionResult ShowOnePerson(int Id)
        {
            return PartialView(customers.FirstOrDefault( c => c.Id == Id));
        }

    }
}
```

2. Open the **site.js** file which is inside wwwroot > js. Add the following code.



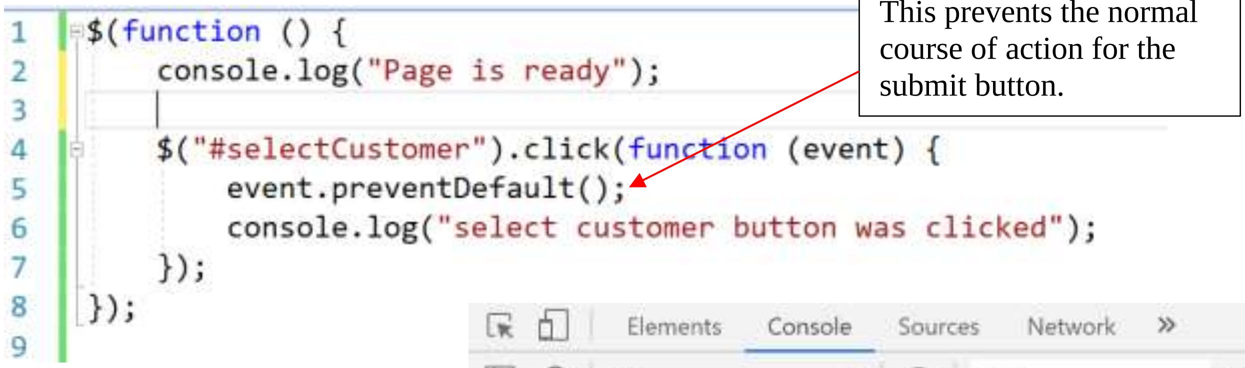
```
$(function () {
    console.log("Page is ready");
});
```

3. Run the program and check the console log window via the Developer Tools. You should see a **ready message**. This indicates the jQuery is loaded and working properly.



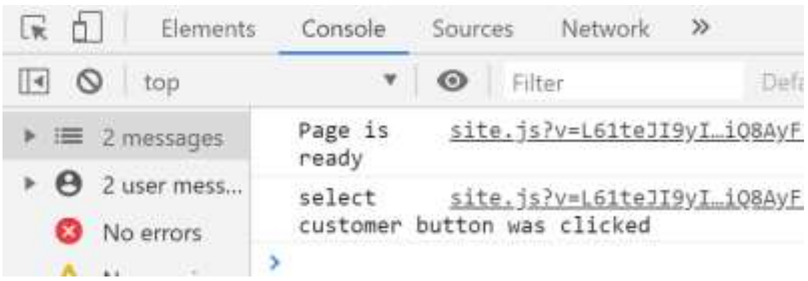
```
Page is
ready          site.js?v=oBHTsp-L2g…8TQbFxhXoGpYhlwN0:2
```

4. Add some more code to **site.js** to add a **click listener** event to the submit button. Note: you can modify javascript files, css, and view files without having to restart the application. Simply save the changes and **refresh the browser**.



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

    $("#selectCustomer").click(function (event) {
        event.preventDefault();
        console.log("select customer button was clicked");
    });
});
```

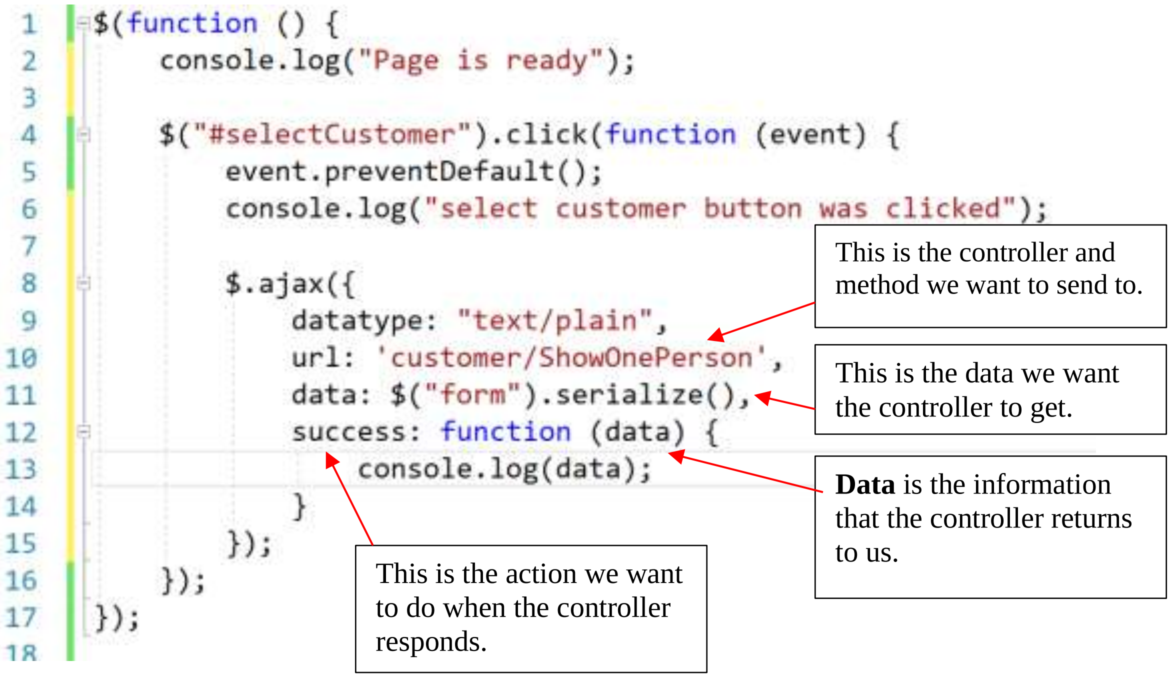
This prevents the normal course of action for the submit button.



```
Page is ready          site.js?v=L61teJI9yI…iQ8AyF
select customer button was clicked          site.js?v=L61teJI9yI…iQ8AyF
```

You should be able to see a console log message when you click the Submit button. Notice that the **preventDefault**() function prevents the form from being submitted to the controller. Essentially, nothing happens when you click the form. The JavaScript code has control now.

5. Add a jQuery AJAX function call in **site.js** that sends the form data to the controller and then prints the response to the console.



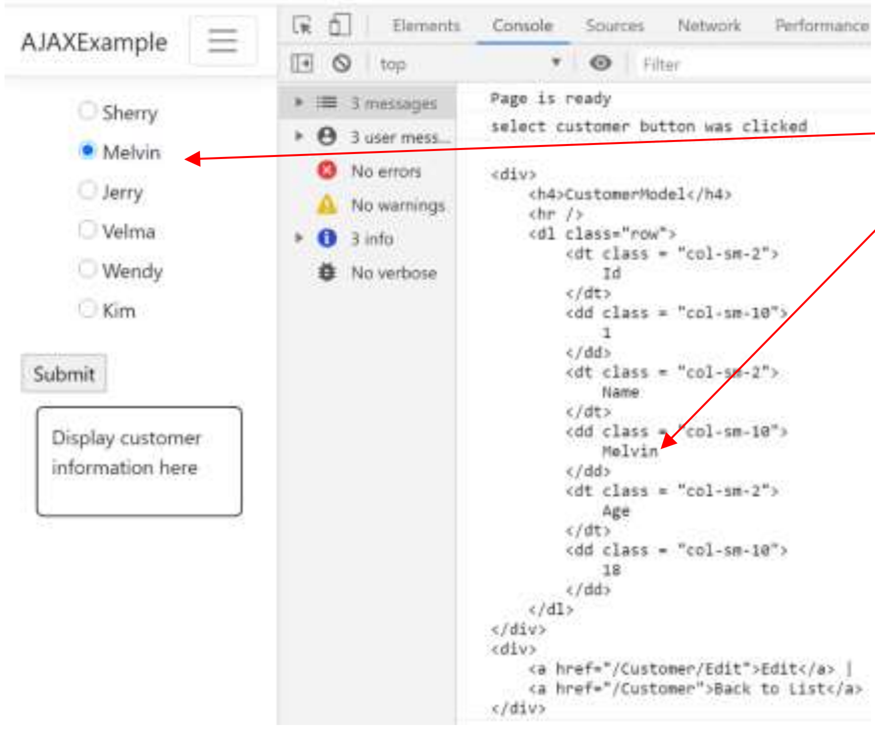
```
$(function () {
    console.log("Page is ready");

    $("#selectCustomer").click(function (event) {
        event.preventDefault();
        console.log("select customer button was clicked");

        $.ajax({
            datatype: "text/plain",
            url: 'customer/ShowOnePerson',
            data: $("form").serialize(),
            success: function (data) {
                console.log(data);
            }
        });
    });
});
```

This is the controller and method we want to send to.

This is the data we want the controller to get.

Data is the information that the controller returns to us.

This is the action we want to do when the controller responds.



( ) Sherry
[✓ Melvin]
( ) Jerry
( ) Velma
( ) Wendy
( ) Kim

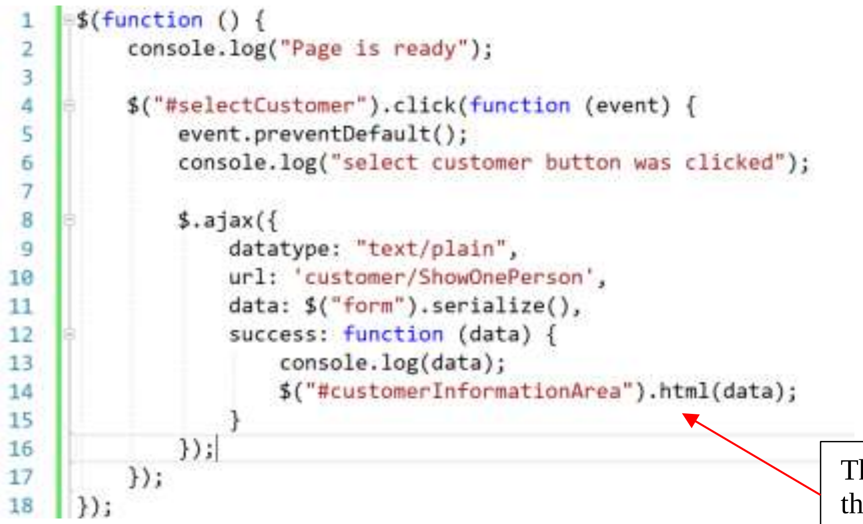
Display customer information here

```
Page is ready
select customer button was clicked

<div>
    <h4>CustomerModel</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            Id
        </dt>
        <dd class = "col-sm-10">
            1
        </dd>
        <dt class = "col-sm-2">
            Name
        </dt>
        <dd class = "col-sm-10">
            Melvin
        </dd>
        <dt class = "col-sm-2">
            Age
        </dt>
        <dd class = "col-sm-10">
            18
        </dd>
    </dl>
</div>
<div>
    <a href="/Customer/Edit">Edit</a> |
    <a href="/Customer">Back to List</a>
</div>
```

Customer #1, Melvin, was selected.

You should see that the controller **sent** the HTML code for the **ShowOnePerson** view back. The JavaScript success function allows us to display the code in any way we wish. In this case, we simply printed it to the console.

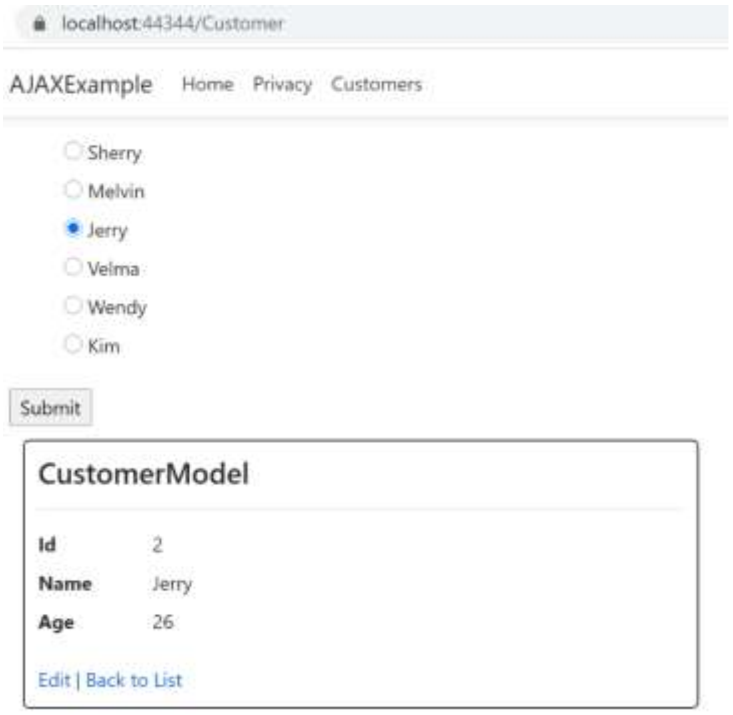
6. Add one more line that replaces a region on the page with the data sent back from the controller.



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

    $("#selectCustomer").click(function (event) {
        event.preventDefault();
        console.log("select customer button was clicked");

        $.ajax({
            datatype: "text/plain",
            url: 'customer/ShowOnePerson',
            data: $("form").serialize(),
            success: function (data) {
                console.log(data);
                $("#customerInformationArea").html(data);
            }
        });
    });
});
```

The html (data) command tells the browser to replace the contents of the item with data.

You should now see the **partial page is returned** and inserted into the <div> at the bottom of the page.



( ) Sherry
( ) Melvin
(●) Jerry
( ) Velma
( ) Wendy
( ) Kim

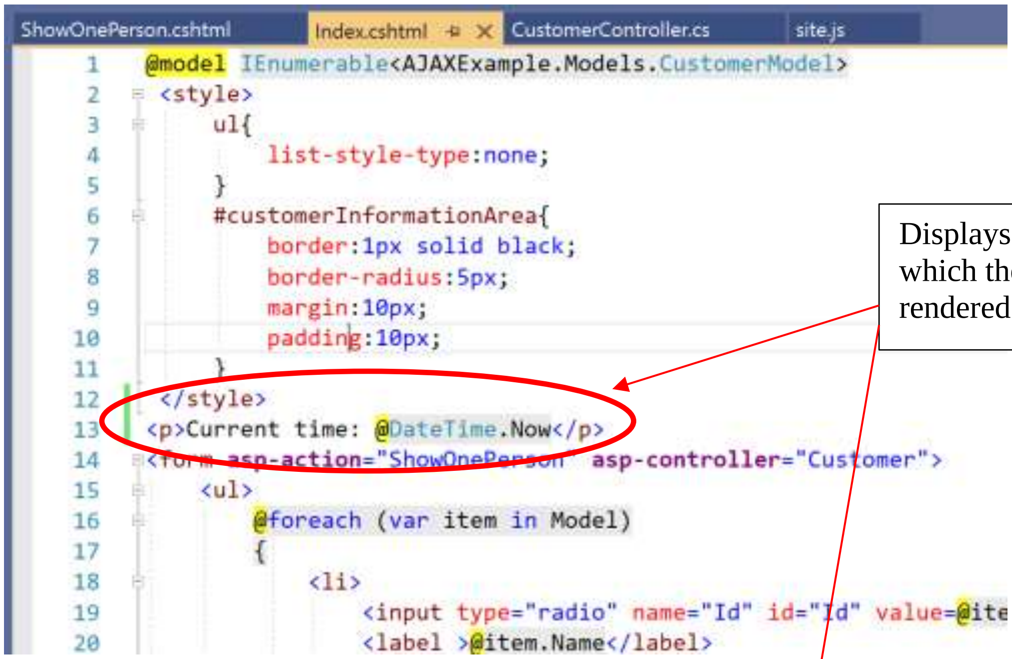
CustomerModel

Id	2
Name	Jerry
Age	26

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.

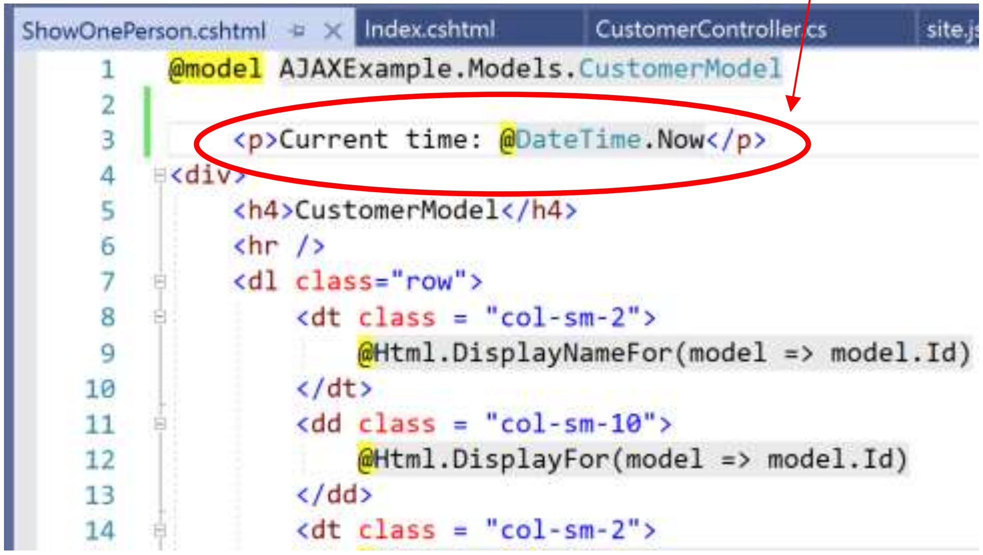
Date and Time Stamp

1. In order to emphasize the fact that the page is only **partially being refreshed**, add the following line into both the **Index.cshtml** page and the **ShowOnePerson.cshtml** page.



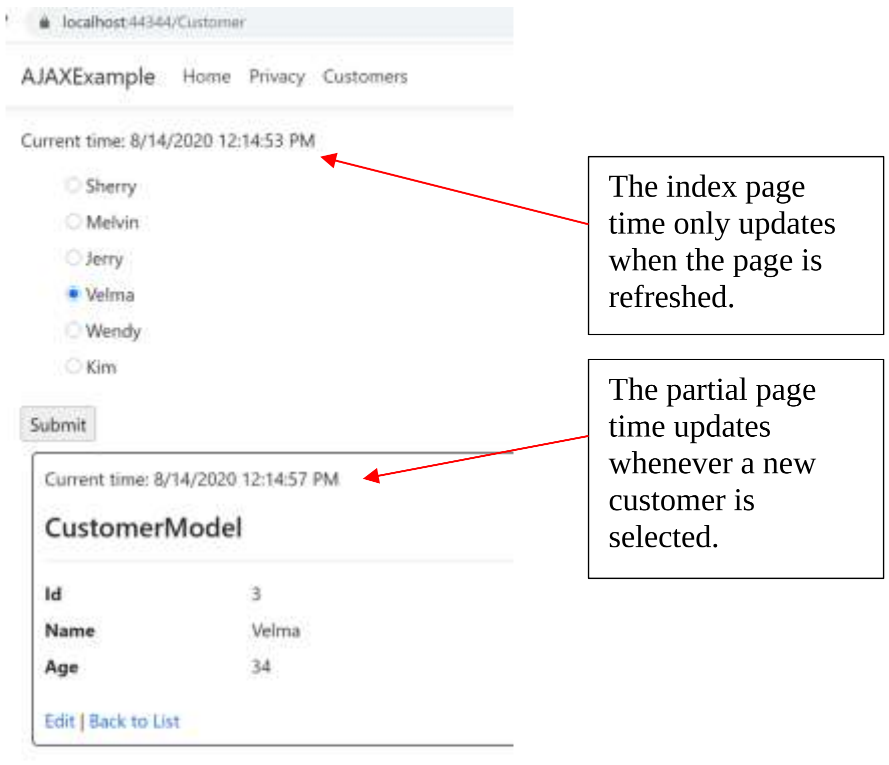
```
@model IEnumerable<AJAXExample.Models.CustomerModel>
<style>
    ul{
        list-style-type:none;
    }
    #customerInformationArea{
        border:1px solid black;
        border-radius:5px;
        margin:10px;
        padding:10px;
    }
</style>
<p>Current time: @DateTime.Now</p>
<form asp-action="ShowOnePerson" asp-controller="Customer">
    <ul>
        @foreach (var item in Model)
        {
            <li>
                <input type="radio" name="Id" id="Id" value=@ite
                <label >@item.Name</label>
```

Displays the exact second at which the page was rendered.



```
@model AJAXExample.Models.CustomerModel

<p>Current time: @DateTime.Now</p>
<div>
    <h4>CustomerModel</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Id)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Id)
        </dd>
        <dt class = "col-sm-2">
```

When a customer is selected, the partial page date is updated, but the original index page date **should remain the same**.


localhost:44344/Customer

Current time: 8/14/2020 12:14:53 PM

○ Sherry
○ Melvin
○ Jerry
● Velma
○ Wendy
○ Kim

Current time: 8/14/2020 12:14:57 PM

CustomerModel

Id	3
Name	Velma
Age	34

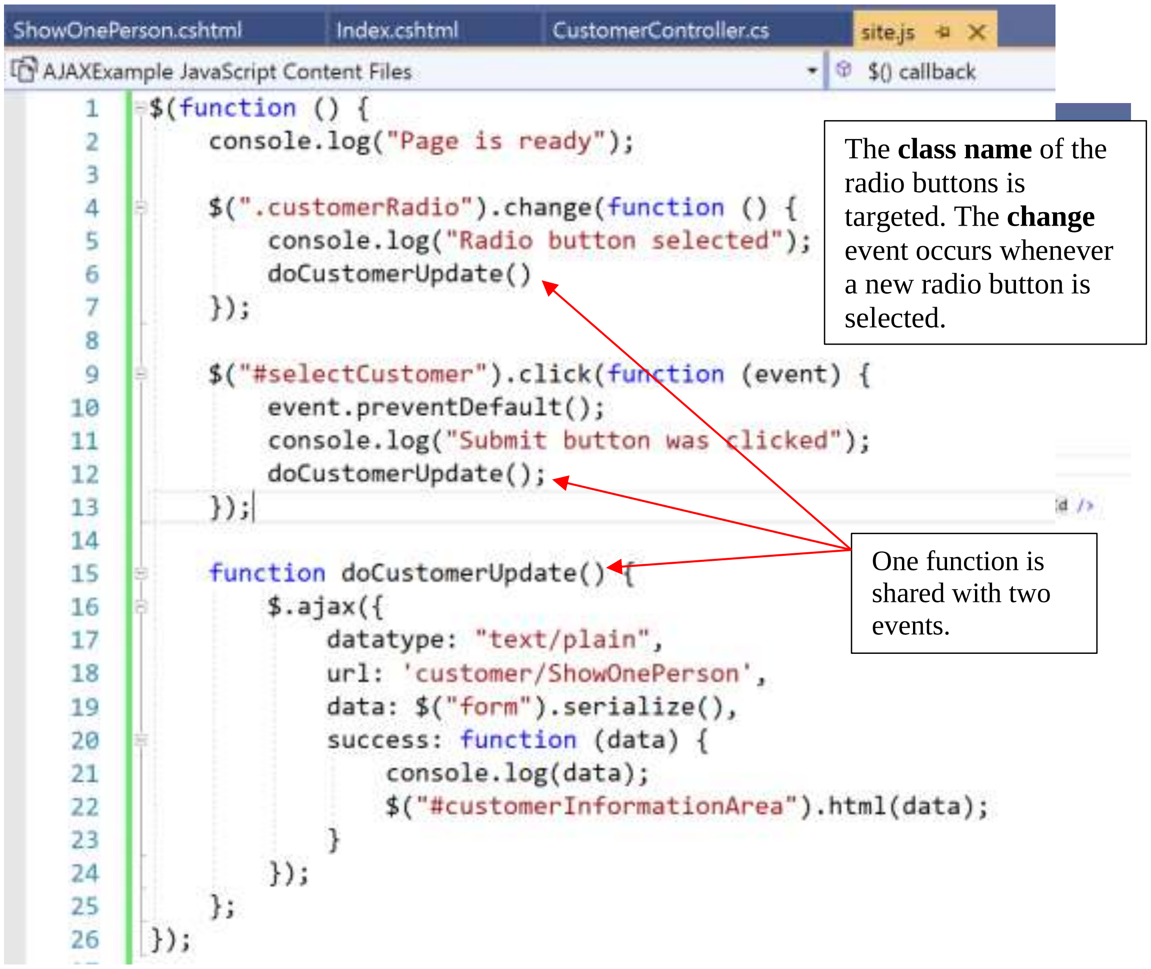
The index page time only updates when the page is refreshed.

The partial page time updates whenever a new customer is selected.

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.

Automatic Submit

With JavaScript, we can trigger a button submit command to make the experience even faster for the user.



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

    $(".customerRadio").change(function () {
        console.log("Radio button selected");
        doCustomerUpdate()
    });

    $("#selectCustomer").click(function (event) {
        event.preventDefault();
        console.log("Submit button was clicked");
        doCustomerUpdate();
    });

    function doCustomerUpdate() {
        $.ajax({
            datatype: "text/plain",
            url: 'customer/ShowOnePerson',
            data: $("form").serialize(),
            success: function (data) {
                console.log(data);
                $("#customerInformationArea").html(data);
            }
        });
    };
});
```

The **class name** of the radio buttons is targeted. The **change** event occurs whenever a new radio button is selected.

One function is shared with two events.

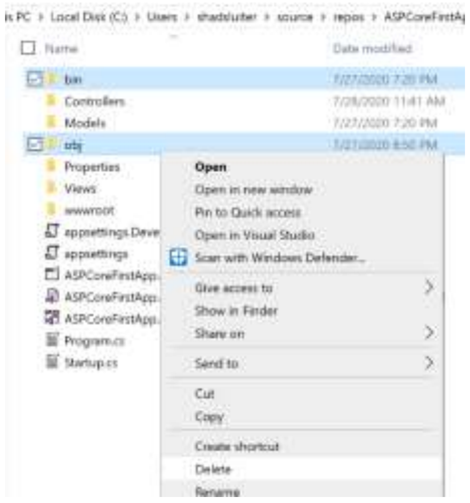
1. Add a class name to the Radio buttons in the **index.cshtml** file.

2. Refactor the **site.js** file so that the Radio Buttons and the Submit button both trigger the ajax event.

3. Run the application. You should see the partial page update occur whenever a new person is selected.

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 at each stage of development. 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 3Button Grid

AJAX, Partial Views and Partial Page Updates in ASP.NET

<u>Goal and Directions:</u>

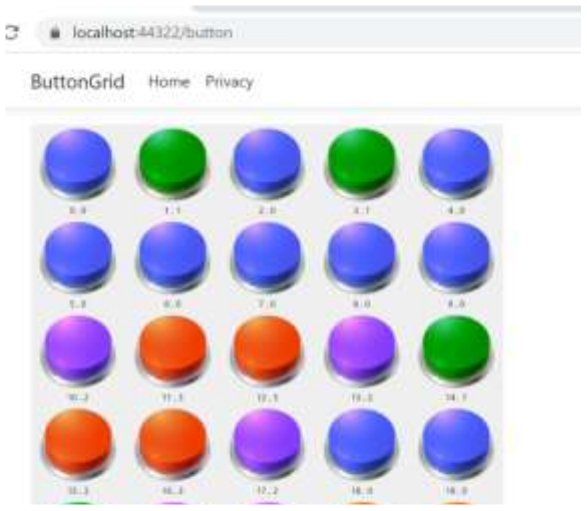
In this activity, you will learn:

- how to build Razor Forms.
- to create Partial Views.
- to perform Partial Page Updates.

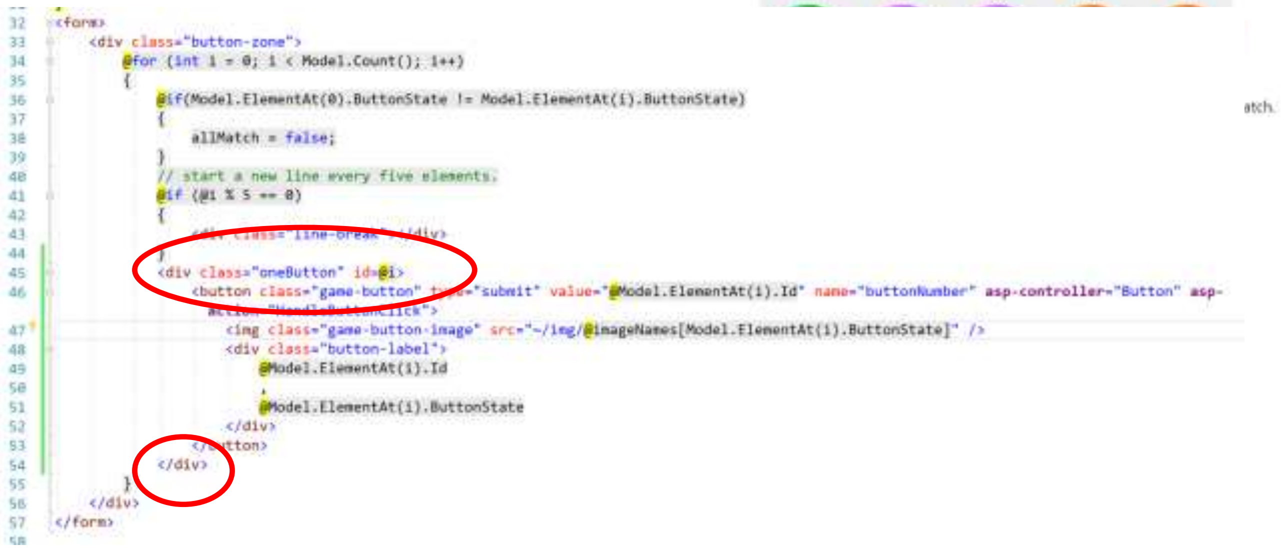
Review Current State

This activity continues with the Button grid example from a previous lesson. Currently, when a button click is handled, the entire webpage is updated. The goal of this tutorial is to show you how to update a single button.

1. Open the existing **button game**. It should look like this:



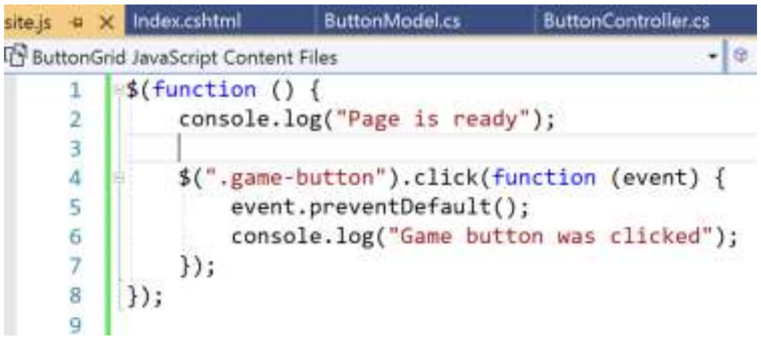
2. In the **Index.cshtml** page, we need to add a <div> around each button and provide it a **unique id number.** Use @i for the id number since @i is the loop counter.



```
<form>
    <div class="button-zone">
        @for (int i = 0; i < Model.Count(); i++)
        {
            @if(Model.ElementAt(0).ButtonState != Model.ElementAt(i).ButtonState)
            {
                allMatch = false;
            }
            // start a new line every five elements.
            @if (@i % 5 == 0)
            {
                <div class="line-break"></div>
            }
            <div class="oneButton" id=@i>
                <button class="game-button" type="submit" value="@Model.ElementAt(i).Id" name="buttonNumber" asp-controller="Button" asp-
                  action="HandleButtonClick">
                    <img class="game-button-image" src="~/img/@imageNames[Model.ElementAt(i).ButtonState]" />
                    <div class="button-label">
                        @Model.ElementAt(i).Id
                        ,
                        @Model.ElementAt(i).ButtonState
                    </div>
                </button>
            </div>
        }
    </div>
</form>
```

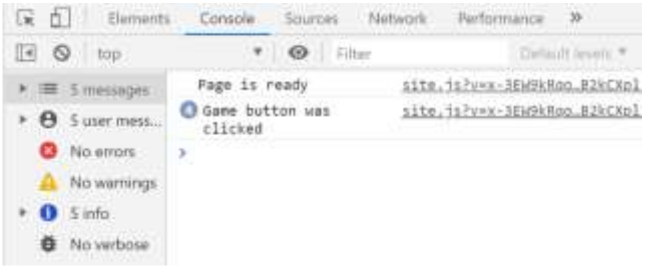
atch.

3. Open the **site.js** file (in wwwroot > js) and add the following code. This will check to see if jQuery is working properly and add a JavaScript **click listener** to every button on the grid.



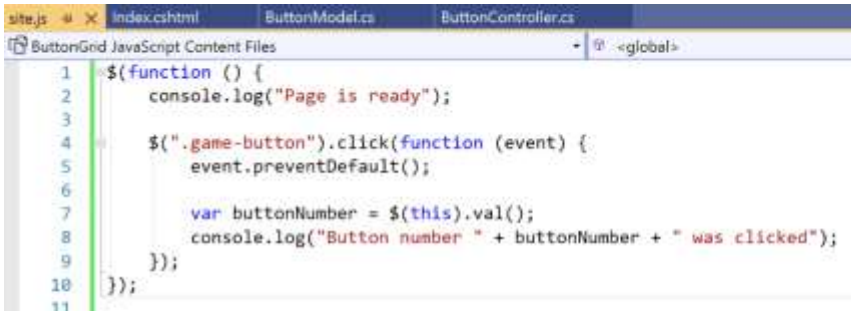
```
$(function () {
    console.log("Page is ready");

    $(".game-button").click(function (event) {
        event.preventDefault();
        console.log("Game button was clicked");
    });
});
```

4. Run the program and verify that the console log messages are appearing in the browser's console.



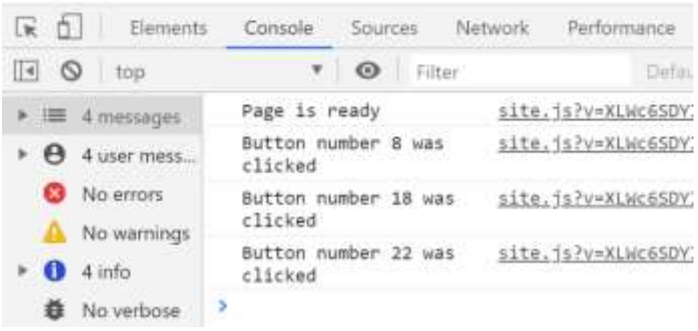
```
Page is ready                     site.js?v=x-3EW9kRqo_R2kCXpl
Game button was                   site.js?v=x-3EW9kRqo_R2kCXpl
clicked
```

5. Modify the code again to capture the **button number** in a variable. Put a confirmation message on the console.



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

    $(".game-button").click(function (event) {
        event.preventDefault();

        var buttonNumber = $(this).val();
        console.log("Button number " + buttonNumber + " was clicked");
    });
});
```

You should see the specific button number in the console log:



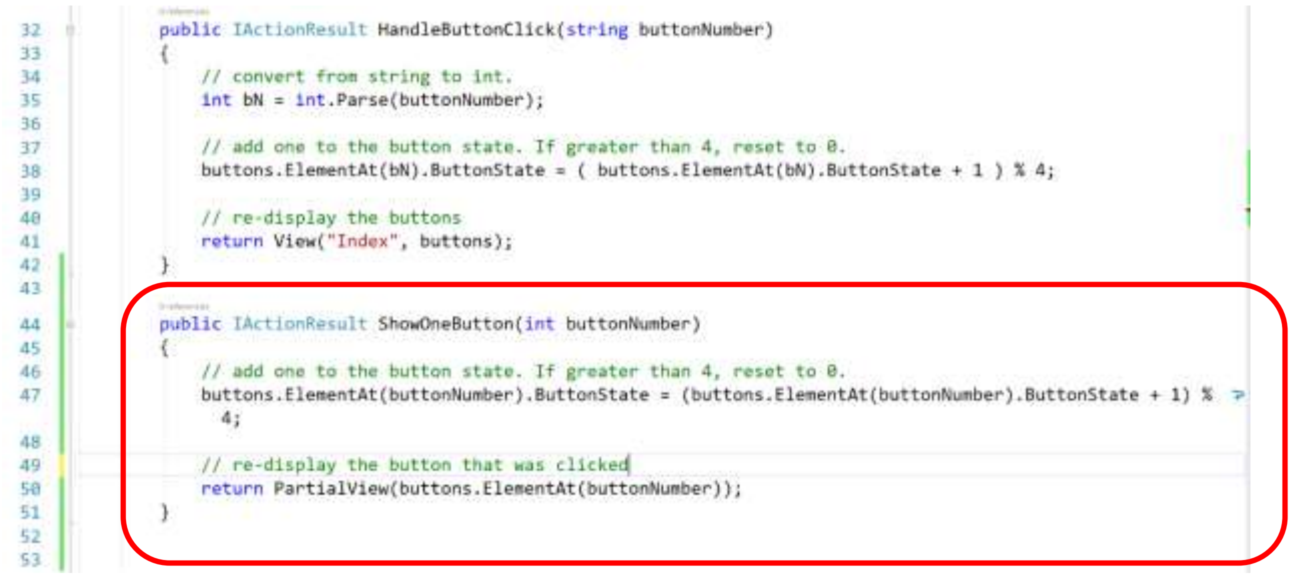
```
Page is ready                     site.js?v=XLWc6SDY
Button number 8 was               site.js?v=XLWc6SDY
clicked
Button number 18 was              site.js?v=XLWc6SDY
clicked
Button number 22 was              site.js?v=XLWc6SDY
clicked
```

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.

Create a partial view for one button

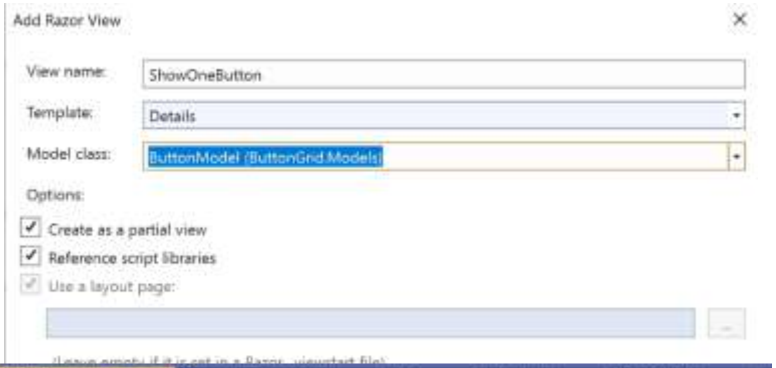
Now that we have determined that we can get a number from a button click, we need to create a method for generating the HTML code for a single button. That will allow us to update a partial view of the game board.

1. In the **ButtonController**, add a new method **ShowOneButton**.



```
public IActionResult HandleButtonClick(string buttonNumber)
{
    // convert from string to int.
    int bN = int.Parse(buttonNumber);

    // add one to the button state. If greater than 4, reset to 0.
    buttons.ElementAt(bN).ButtonState = ( buttons.ElementAt(bN).ButtonState + 1 ) % 4;

    // re-display the buttons
    return View("Index", buttons);
}

public IActionResult ShowOneButton(int buttonNumber)
{
    // add one to the button state. If greater than 4, reset to 0.
    buttons.ElementAt(buttonNumber).ButtonState = (buttons.ElementAt(buttonNumber).ButtonState + 1) %
      4;

    // re-display the button that was clicked
    return PartialView(buttons.ElementAt(buttonNumber));
}
```

2. **Right-click** in the ShowOneButton method and choose **Add View**. Create a **Details** view for the **ButtonModel**.



Add Razor View

View name: ShowOneButton

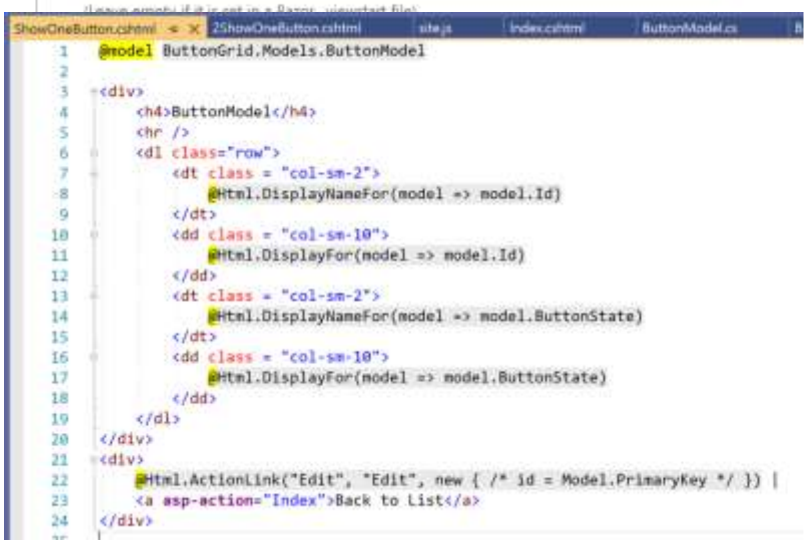
Template: Details

Model class: ButtonModel (ButtonGrid.Models)

Options:

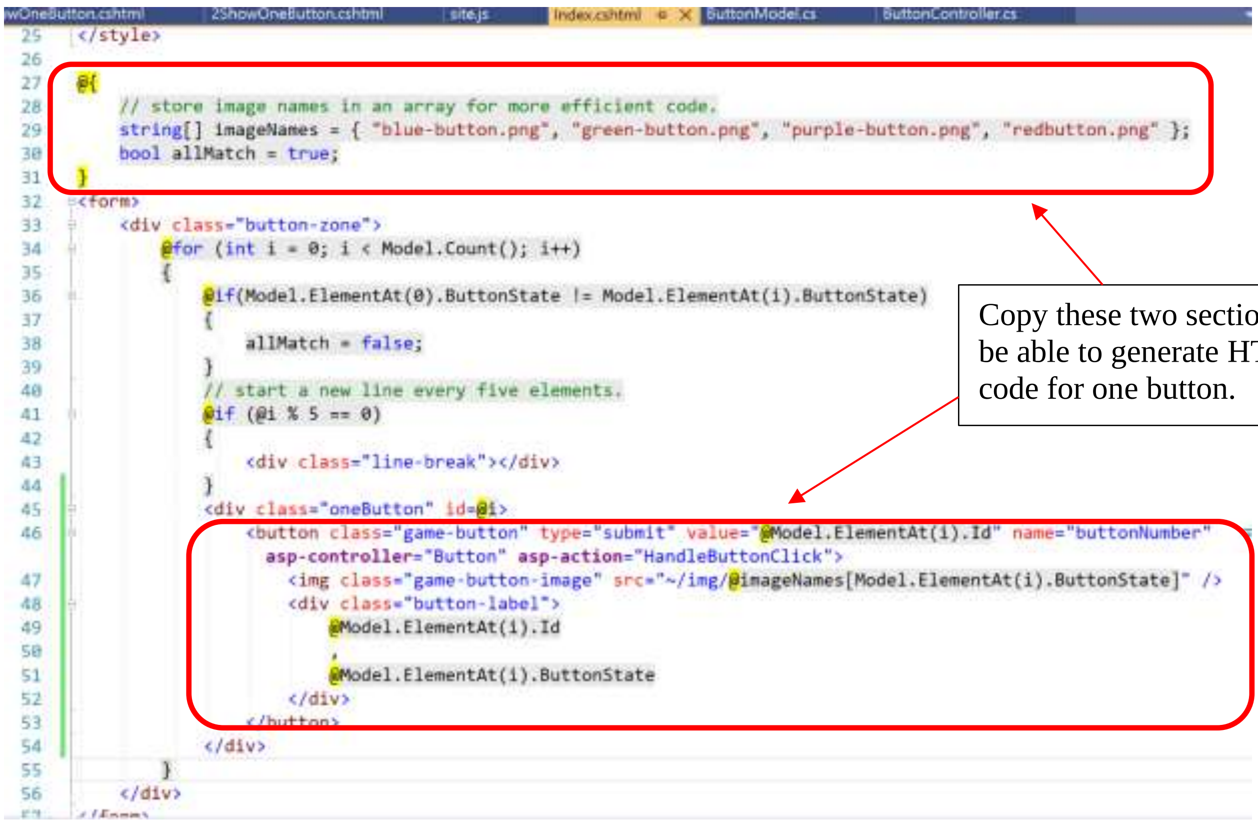
- [✓] Create as a partial view
- [✓] Reference script libraries
- [✓] Use a layout page:

This should result in showing the default Details view for a ButtonModel. None of this is very valuable except for the first line. All of it will change shortly.



```
@model ButtonGrid.Models.ButtonModel

<div>
    <h4>ButtonModel</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Id)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Id)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.ButtonState)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.ButtonState)
        </dd>
    </dl>
</div>
<div>
    @Html.ActionLink("Edit", "Edit", new { /* id = Model.PrimaryKey */ }) |
    <a asp-action="Index">Back to List</a>
</div>
```

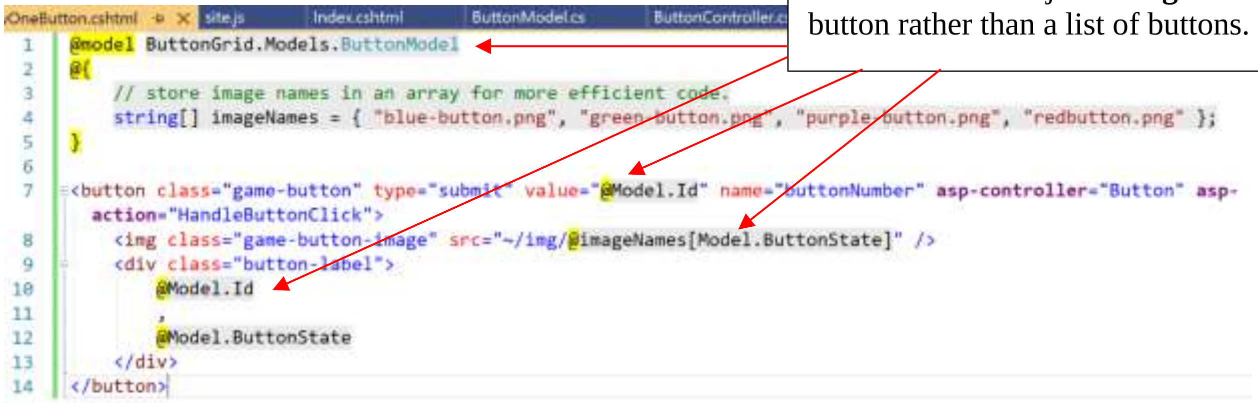
In the next step, we are going to borrow a section from the index.cshtml view.



```
</style>

@{
    // store image names in an array for more efficient code.
    string[] imageNames = { "blue-button.png", "green-button.png", "purple-button.png", "redbutton.png" };
    bool allMatch = true;
}
<form>
    <div class="button-zone">
        @for (int i = 0; i < Model.Count(); i++)
        {
            @if(Model.ElementAt(0).ButtonState != Model.ElementAt(i).ButtonState)
            {
                allMatch = false;
            }
            // start a new line every five elements.
            @if (@i % 5 == 0)
            {
                <div class="line-break"></div>
            }
            <div class="oneButton" id=@i>
                <button class="game-button" type="submit" value="@Model.ElementAt(i).Id" name="buttonNumber"
                  asp-controller="Button" asp-action="HandleButtonClick">
                    <img class="game-button-image" src="~/img/@imageNames[Model.ElementAt(i).ButtonState]" />
                    <div class="button-label">
                        @Model.ElementAt(i).Id
                        ,
                        @Model.ElementAt(i).ButtonState
                    </div>
                </button>
            </div>
        }
    </div>
</form>
```

Copy these two sections to be able to generate HTML code for one button.

3. The **ShowOnebutton**.cshtml file should have just enough code to generate a **single square** on the game board as shown here. Notice the model is now a single button instead of a list of buttons. Some of the details will change.



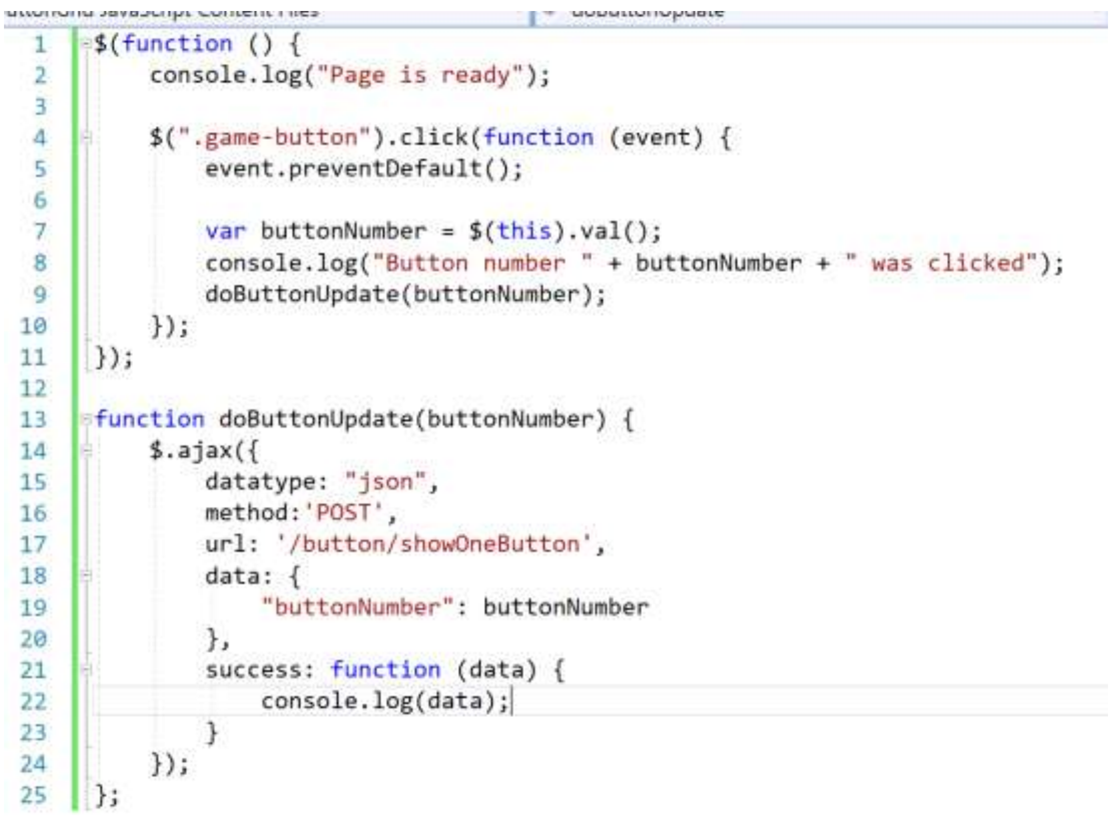
```
@model ButtonGrid.Models.ButtonModel
@{
    // store image names in an array for more efficient code.
    string[] imageNames = { "blue-button.png", "green-button.png", "purple-button.png", "redbutton.png" };
}

<button class="game-button" type="submit" value="@Model.Id" name="buttonNumber" asp-controller="Button" asp-
  action="HandleButtonClick">
    <img class="game-button-image" src="~/img/@imageNames[Model.ButtonState]" />
    <div class="button-label">
        @Model.Id
        ,
        @Model.ButtonState
    </div>
</button>
```

The model is now just a **single** button rather than a list of buttons.

AJAX update

Now we are ready to return to the **site.js** file and perform an AJAX request.

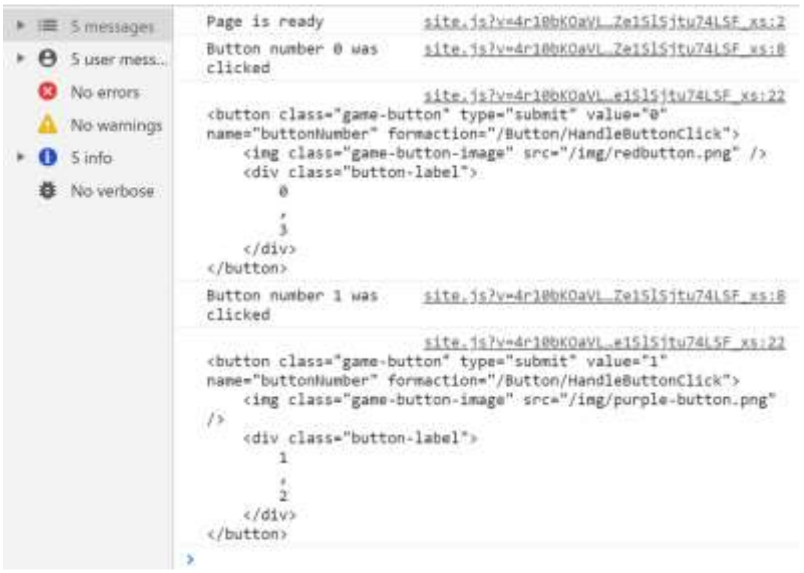
1. Open the **site.js** file and add the following code.



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

    $(".game-button").click(function (event) {
        event.preventDefault();

        var buttonNumber = $(this).val();
        console.log("Button number " + buttonNumber + " was clicked");
        doButtonUpdate(buttonNumber);
    });
});

function doButtonUpdate(buttonNumber) {
    $.ajax({
        datatype: "json",
        method:'POST',
        url: '/button/showOneButton',
        data: {
            "buttonNumber": buttonNumber
        },
        success: function (data) {
            console.log(data);
        }
    });
};
```

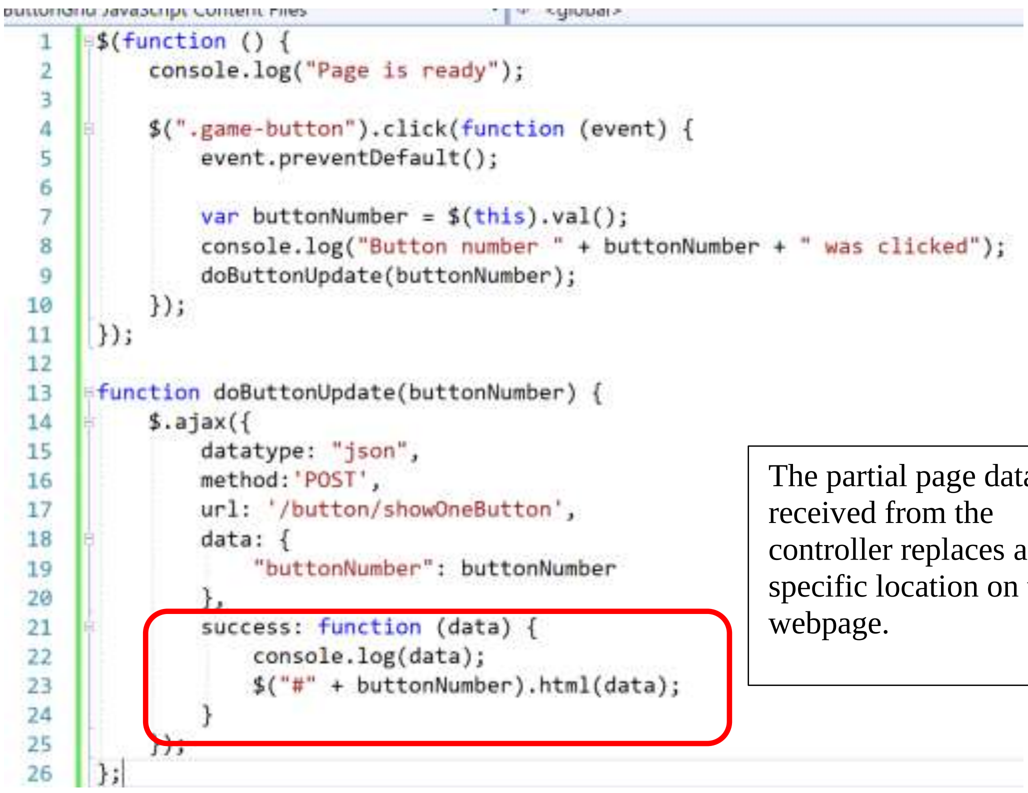
2. Run the program and confirm that the **console is printing a partial view of the button**. You should be able to confirm the button number is correct and that the button color changes on each click.

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.



```
Page is ready                    site.js?v=4r10bKOaVL_Ze1SlSjtu74LSF_xs:2
Button number 0 was              site.js?v=4r10bKOaVL_Ze1SlSjtu74LSF_xs:8
clicked
                                 site.js?v=4r10bKOaVL_e1SlSjtu74LSF_xs:22
<button class="game-button" type="submit" value="0"
name="buttonNumber" formaction="/Button/HandleButtonClick">
    <img class="game-button-image" src="/img/redbutton.png" />
    <div class="button-label">
        0
        ,
        3
    </div>
</button>
Button number 1 was              site.js?v=4r10bKOaVL_Ze1SlSjtu74LSF_xs:8
clicked
                                 site.js?v=4r10bKOaVL_e1SlSjtu74LSF_xs:22
<button class="game-button" type="submit" value="1"
name="buttonNumber" formaction="/Button/HandleButtonClick">
    <img class="game-button-image" src="/img/purple-button.png"
/>
    <div class="button-label">
        1
        ,
        2
    </div>
</button>
>
```

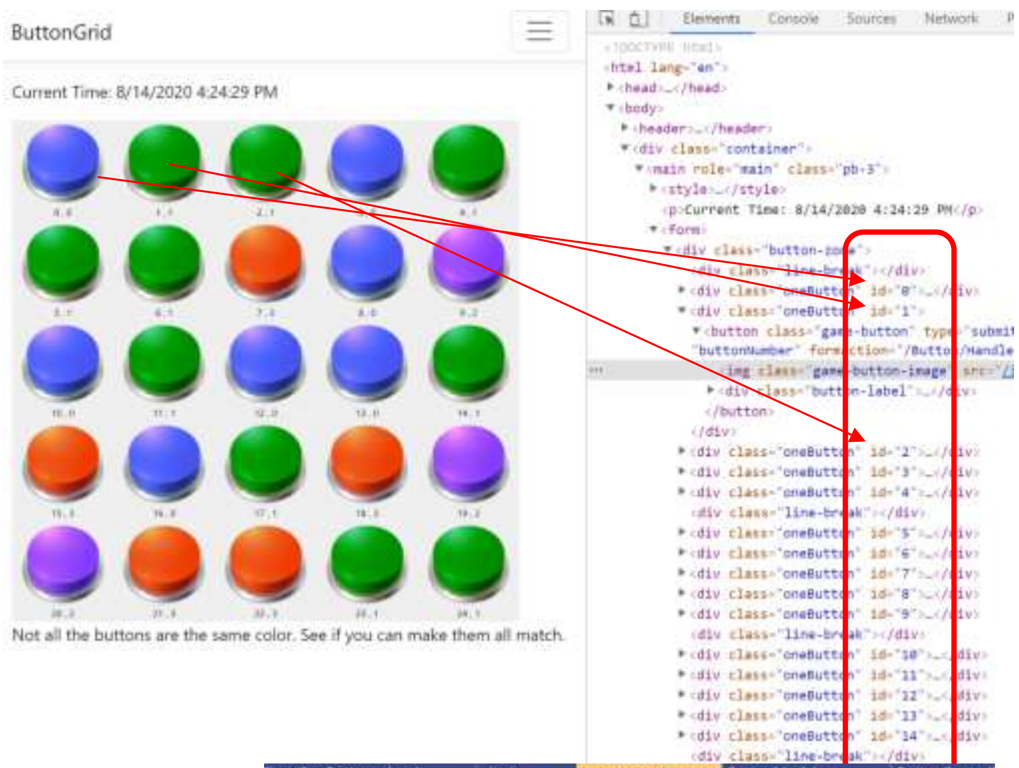
4. Add one more line to the **site.js** file to update the HTML of the button in question.



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

    $(".game-button").click(function (event) {
        event.preventDefault();

        var buttonNumber = $(this).val();
        console.log("Button number " + buttonNumber + " was clicked");
        doButtonUpdate(buttonNumber);
    });
});

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

The partial page data received from the controller replaces a specific location on the webpage.

The key to understanding how this works is to know that each button is enclosed in a <div> tag that has the correct id number, as shown here.

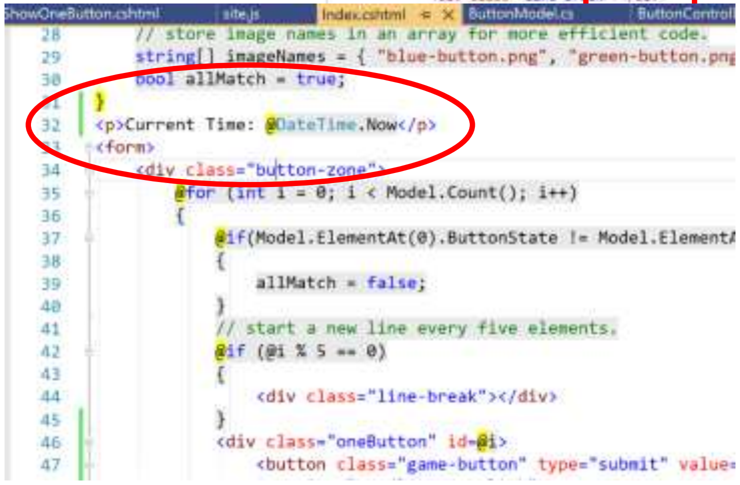


Current Time: 8/14/2020 4:24:29 PM

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

```
<!DOCTYPE html>
<html lang="en">
 ▶<head>…</head>
 ▼<body>
   ▶<header>…</header>
   ▼<div class="container">
     ▼<main role="main" class="pb-3">
       ▶<style>…</style>
        <p>Current Time: 8/14/2020 4:24:29 PM</p>
       ▼<form>
         ▼<div class="button-zone">
            <div class="line-break"></div>
           ▶<div class="oneButton" id="0">…</div>
           ▼<div class="oneButton" id="1">
             ▼<button class="game-button" type="submit
              "buttonNumber" formaction="/Button/Handle
                 <img class="game-button-image" src="/
               ▶<div class="button-label">…</div>
              </button>
            </div>
           ▶<div class="oneButton" id="2">…</div>
           ▶<div class="oneButton" id="3">…</div>
           ▶<div class="oneButton" id="4">…</div>
            <div class="line-break"></div>
           ▶<div class="oneButton" id="5">…</div>
           ▶<div class="oneButton" id="6">…</div>
           ▶<div class="oneButton" id="7">…</div>
           ▶<div class="oneButton" id="8">…</div>
           ▶<div class="oneButton" id="9">…</div>
            <div class="line-break"></div>
           ▶<div class="oneButton" id="10">…</div>
           ▶<div class="oneButton" id="11">…</div>
           ▶<div class="oneButton" id="12">…</div>
           ▶<div class="oneButton" id="13">…</div>
           ▶<div class="oneButton" id="14">…</div>
            <div class="line-break"></div>
```

5. Let's put a **time stamp** above the game board to prove to ourselves that the Ajax is doing a partial page update. The time stamp should not change unless we refresh the entire page.

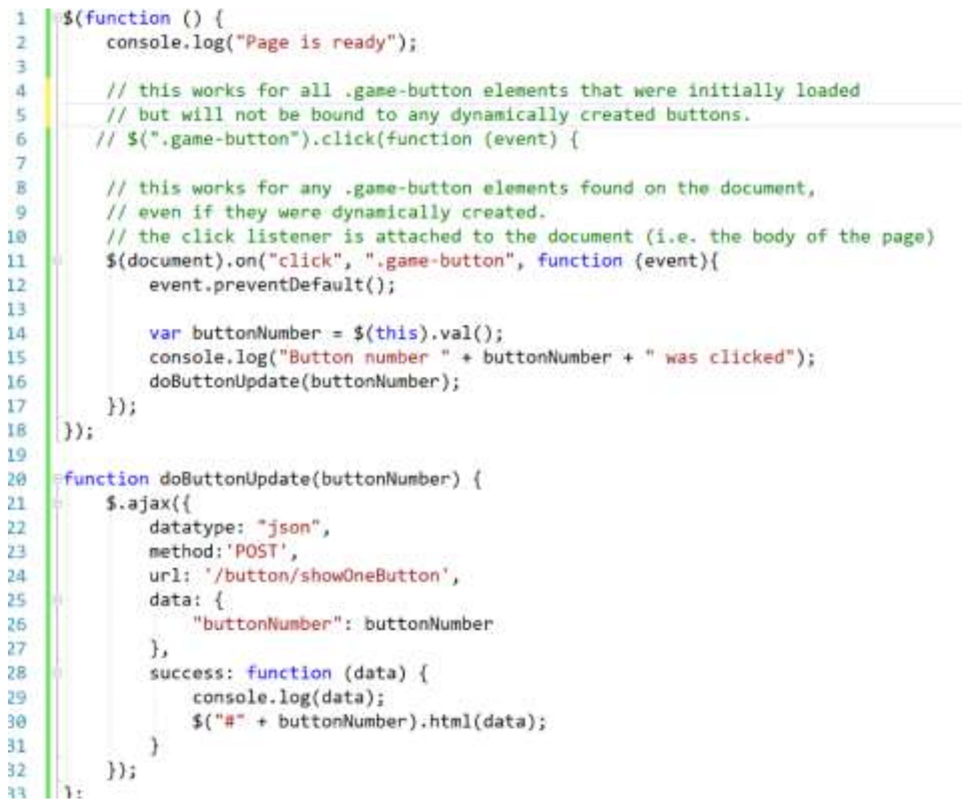


```
        // store image names in an array for more efficient code.
        string[] imageNames = { "blue-button.png", "green-button.png
        bool allMatch = true;
    }
    <p>Current Time: @DateTime.Now</p>
    <form>
        <div class="button-zone">
            @for (int i = 0; i < Model.Count(); i++)
            {
                @if(Model.ElementAt(0).ButtonState != Model.Element
                {
                    allMatch = false;
                }
                // start a new line every five elements.
                @if (@i % 5 == 0)
                {
                    <div class="line-break"></div>
                }
                <div class="oneButton" id=@i>
                    <button class="game-button" type="submit" value=
```

6. Run the program again. It almost works correctly. Only in one case does the Ajax fail. Can you determine which case causes the problem? Try to determine what is not working before reading the answer below.

If you click any button twice, the ajax function fails to trigger. Instead, the HandleButtonClick method fires which causes the entire page to reload.

The issue is often called the **“dynamically created button problem.”** If you try to attach a click event to the elements that are dynamically added to DOM, it will not work. When the page first loaded, the click events were associated only to the elements that existed at that time on the page. Any new elements, dynamically created after the initial page load are not bound to the click listener. To bind the click event to all existing and future elements, use the jQuery on() method.



```
$(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);
    });
});

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

7. Update site.js with the following code fix.

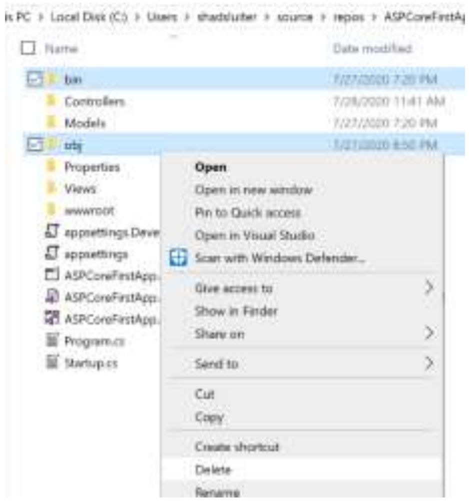
8. Take a **screenshot** of the app at this stage. Paste it into a Microsoft Word document and caption the image with a brief explanation of what you just demonstrated.

End Game Condition Challenge

Another problem arises: the end game condition doesn’t appear to be working. The problem is left for you to fix.

After fixing the end game condition, take a screenshot and paste it into Word document.

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 at each stage of development. 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.