Introduction to ASP.NET Core Development

ASP.NET Core 1.0 and Visual Studio 2015

About this Lab Manual

This lab manual consists of a series of hands-on lab exercises designed to guide the reader through the process of building a complete ASP.NET Core e-commerce style application.

System Requirements

- **Visual Studio 2015 (any edition) with Update 3 (or later)**
- **.NET Core 1.0 for Visual Studio**
  - Instructions and download links are available from <https://www.microsoft.com/net/core#windows>
- **LocalDB or SQL Server (any version)**
  - Installed by default as part of the Visual Studio installation process
  - To confirm the installation of LocalDB, you can execute [`sqllocaldb i`] from a command prompt. This should list the LocalDB instances that are available. If the command is not recognized then LocalDB is not installed.
  - If not installed, you can download an installer for LocalDB from <http://www.microsoft.com/en-us/download/details.aspx?id=29062> Choose the file named **SqlLocalDB.MSI**
  - If using a version of SQL Server other than LocalDB, you must be able to connect to the database with sufficient permissions to **create a new database**
- An **internet connection** is required to download and install NuGet packages from <https://api.nuget.org>

The complete lab manual for this course consists of 15 hands-on exercises.

This sample consists of lab 1, 2, and 9.

Lab 1

Objectives

- Create a new **ASP.NET Core application**

Procedures

1. In Visual Studio, choose [**File > New > Project...**] from the menu.

   a. On the left side of the dialog, select [**Installed > Other Project Types > Visual Studio Solutions**] and choose the **Blank Solution** template.

   > *Although not required, we are starting with a blank solution so that we can have better control over our project names and default namespaces.*

   b. Name the solution **EComm** and choose an appropriate **location** for the solution.

   > *You will be working with this solution throughout the course so make sure it is a location you can easily navigate to later.*

   c. Click **OK**.

2. Right-click on the **EComm** solution in Solution Explorer, choose **[Add > New Solution Folder]** and name the folder **src.**

3. Add a second folder to the solution named **test**.

   > *The **src** folder will contain all of the code that will become part of what we will deploy. The **test** folder will contain our unit tests.*

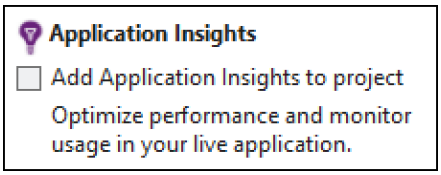
4. Right-click on the src folder and choose **[Add > New Project...]**

   a. On the left side of the dialog, select [**Installed > Visual C# > Web**]

   b. On the right side, select **ASP.NET Core Web Application (.NET Core).**

   > *Notice that there is also an option for "ASP.NET Core Web Application (.NET Framework)". This would create an application that targets the full .NET Framework but is limited to Windows-based operating systems.*

c. If you have a checkbox for **Application Insights**, ensure this box is **not checked**.



> *Although we will not be using it in this course, Application Insights can provide some very useful features. You can learn about Application Insights at https://www.visualstudio.com/application-insights*

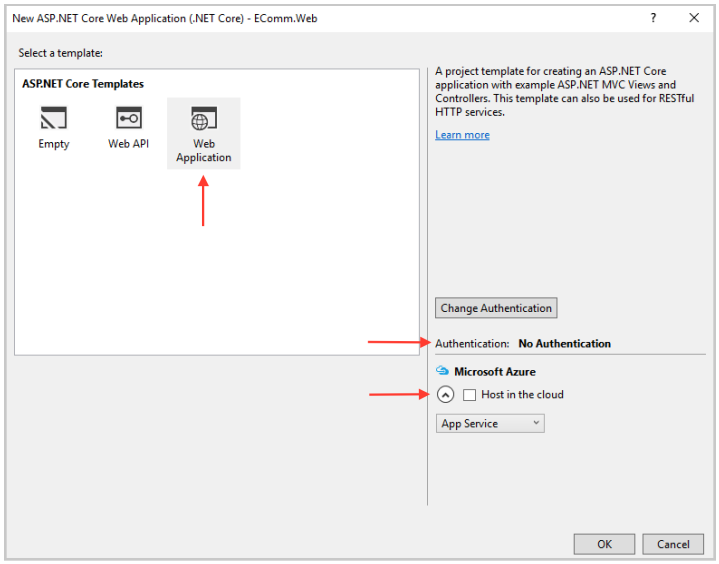
d. Name the project **EComm.Web** and click **OK.**

5. In the new project dialog, select the **Web Application** template.

a. Ensure that the authentication type is set to **No Authentication**. Use the **Change Authentication** button to change this if necessary.

> *We will be using authentication in our application but we will not be using all of the features that are added by the built-in template. Instead, we will add the items we need manually.*

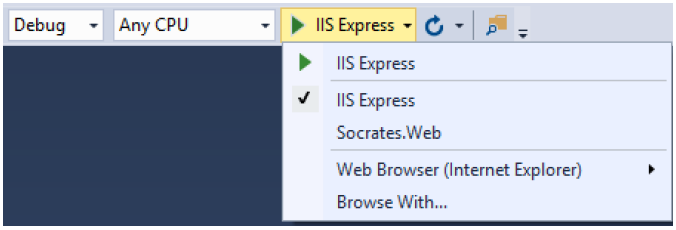
b. If there is a checkbox labeled **Host in the cloud**, make sure that box is **not checked**.



c. Click **OK**.

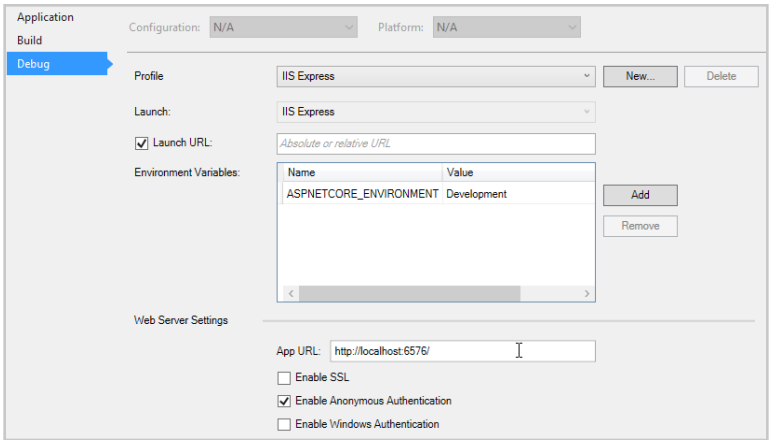
*If the project was successfully created, you should be looking at the **Project_Readme.html** file that says "Welcome to ASP.NET Core" at the top.*

6. In the Visual Studio toolbar, examine the **debug target** list.



*There are two options in the debug target list - **IIS Express** and **EComm.Web**. In the first case, your application will be hosted by IIS Express (acting as a reverse proxy for Kestrel). In the second case, your application will be launched as a console application (**dotnet.exe** will execute the Main method in Program.cs)*

7. **Right-click** on the **EComm.Web** project in **Solution Explorer**, choose **Properties**, and select the **Debug** tab.



*This dialog allows you to control how your application will be launched by Visual Studio (port number, SSL, etc.) This information is stored in a file named **launchSettings.json** under the **Properties** node of your project.*

8. Ensure **IIS Express** is selected in the target list and **run** the application.

Visual Studio should launch your default web browser and display the home page included with the Visual Studio template.

9. **Explore** the pages that are included with the template (home, about, and contact).

10. **Stop** the application and **run** it again with **EComm.Web** selected in the target list. Examine the messages displayed in the console window that appears.

> *These messages appear because the default template has logging configured. We will discuss logging later in the course.*

End of Lab

Lab 2

Objectives

- Return a **string** from a controller action
- Return **HTML** from a controller action
- Return **JSON** from a controller action

Procedures

1. If not already open, re-open your solution from Lab 1.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder of Lab 2.*

2. Open **HomeController.cs** for editing.

   > *This controller is already accessible via web requests because of the routing rule defined in the application's **Configure** method. That rule specifies that the default controller is **HomeController** and the default action is **Index**.*

3. Modify the **Index** action so that it returns a simple **string** as the response.

```
public IActionResult Index()
{
  return Content("Hello from HomeController");
}
```

4. **Run** the application and check the result. Be sure to also view the **source** of the page returned to the browser and notice that nothing additional (beyond the string itself) is returned.

   > *Since MVC is not returning any additional content, building a web service would be as simple as defining an action that returns XML or JSON.*

5. Return to Visual Studio and modify the **Index** action so that it returns some simple **HTML**.

```
public IActionResult Index()
{
  return Content("<strong>Hello</strong> from <em>HomeController</em>",
                 "text/html");
}
```

> *Notice that we have to specify the content type of "text/html" so that the browser will interpret the data appropriately. Of course, most of the time, you will want to move your HTML out into a separate view file.*

6. **Run** the application and check the result.

7. Return to Visual Studio and modify the **Index** action so that it returns the contents of an object in **JSON** format.

```
public IActionResult Index()
{
  var person = new { FirstName = "Bill", LastName = "Gates" };
  return Json(person);
}
```

> *The **Json** method of the controller will automatically serialize the public properties of the object that is passed to it (including child objects and collections).*

8. **Run** the application one more time to check the results. Note that the response you receive will depend on the browser you are using. For example, Chrome will display the JSON content directly in the browser as text while Internet Explorer will treat the JSON response as an incoming file and prompt you to open or save the file.

9. Feel free to **experiment** with returning different response types from the Index action. When you are done, change the action back to returning the **default view** for the action.

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

End of Lab

Lab 9

Objectives

- Add an **HTML form** to a view
- Create an action to handle the **form submission**
- Refactor the form and response to use **ajax**

Procedures

1. If not already open, re-open your solution from Lab 8.

   > *If you do not want to use your solution from the previous lab, you can open the solution in the Begin folder of Lab 9.*

   > *In this lab, we will begin the process of implementing shopping cart functionality. The first step will be to create a form where the user can specify how many of a particular product they would like to add to their cart.*

2. Add a form to **Views/Product/Detail.cshtml** by using the **form tag helper**. The form should go below the existing markup.

```
<form id="addToCartForm" asp-controller="Product"
  asp-action="AddToCart" asp-route-id="@Model.Id">

  <input name="quantity" size="3" />
  <input type="submit" value="Add to Cart" />
</form>
```

   > *We are specifying that this form should post to an action named AddToCart. So, we'll add that action now.*

3. Add a **new action** to **ProductController** named **AddToCart**. This action should accept two parameters (**id** of type **int** and **quantity** of type **int**). Make sure the action includes the **HttpPost** attribute.

```
[HttpPost]
public IActionResult AddToCart(int id, int quantity)
{
}
```

4. Add code to the **AddToCart** action that retrieves the correct **product**, calculates the **total cost** (based on quantity), creates an appropriate **message**, and passes the message to a **partial view**.

```
[HttpPost]
public IActionResult AddToCart(int id, int quantity)
{
  var product = _context.Products.SingleOrDefault(p => p.Id == id);
  var totalCost = quantity * product.UnitPrice;

  string message = $"You added {product.ProductName} " +
                   $"(x {quantity}) to your cart " +
                   $"at a total cost of {totalCost:C}.";

  return PartialView("_AddedToCart", message);
}
```

> *The next step we be to create the _AddedToCart partial view which should be strongly-typed to a string.*

> *Note that we are not recording the user's purchase on the server-side (the actual "add to cart" operation). We will do this in a future lab.*

5. Add a new view named **_AddedToCart.cshtml** to the **Views/Product** folder and add some markup to display the **message** and a link to "continue shopping".

```
@model string
<br>
<div id="message" class="alert alert-success" role="alert">@Model</div>
<p>
  <a asp-controller="Home" asp-action="Index">Continue Shopping</a>
</p>
```

6. **Run** the application and test the form. You should see the message but as a full-page reload (no menu or footer).

> *We could have easily used a full view that is based on our layout. However, what we really should do is change the view to submit the form asynchronously via ajax and inject the content of the partial view into the existing page.*

7. Add a ** ** element to the bottom of **Detail.cshtml** to act as a placeholder for where we will inject the response from the server.

```
<div id="message"></div>
```

8. Add a **section** to the bottom of **Detail.cshtml** named **scripts**.

```
@section scripts {
}
```

> *This section is defined as optional in _Layout.cshtml. So, we can include JavaScript that is specific to this view but know that the layout will decide where in the overall response the script will appear.*

9. Add the JavaScript that will be used to submit the form via **ajax**. This code is available in the **Assets/Code/Lab09** folder of the lab bundle if you prefer to copy-and-paste the code.

```
@section scripts {
<script type="text/javascript">
  $(document).ready(function() {
    $('form').submit(function(event) {
      var formData = {
        'id':$('input[name=id]').val(),
        'quantity':$('input[name=quantity]').val()
      };
      $.ajax({
        type:'POST',
        url:$('#addToCartForm').attr('action'),
        data:formData
      })
        .done(function(response) {
          $('#message').html(response);
         });
      event.preventDefault();
    });
  });
</script>
}
```

> *Of course, we could move this JavaScript into a separate file and use the scripts section to include that file.*

10. **Run** the application and confirm that the add to cart operation now results in a partial-page update.

End of Lab