Introduction to ASP.NET Core 3

ASP.NET 3 Core Development

Visual Studio 2019

About this Lab Manual

This lab manual consists of a series of hands-on lab exercises for learning to build ASP.NET Core 3 Web applications using Visual Studio 2019.

System Requirements

- **.NET Core 3.1.100 SDK**
  - Available to download from <https://dotnet.microsoft.com/download>
- **Visual Studio 2019** (any edition) version **16.4** (or later)
  - You can check the version number of Visual Studio 2019 by selecting [Help > About Microsoft Visual Studio]
  - Visual Studio 2019 Community Edition is available as a free download from <https://www.visualstudio.com/>
- **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**
- **Postman** application for testing and debugging Web APIs
  - Available as a free download from <https://www.getpostman.com/>
  - A different web debugging proxy application (such as Fiddler) can be used if necessary
- An **internet connection** is required to download and install NuGet packages from <https://api.nuget.org>

Lab 10

Objectives

- Create the **product edit form**

Procedures

1. If not already open, re-open your **EComm** solution from the previous lab.

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

2. Use what you have learned to add a form for editing a product. A list of requirements is provided below. Step-by-step instructions are also provided on the next page but you should try your best to accomplish the task without using those instructions.

   a. The **ProductController** should have an **Edit** action with a route of "**product/edit/{id}**".

   b. Use the **Edit template** when adding the **view** and ensure **reference script libraries** is selected (to enable client-side data validation).

   c. The **SupplierId** property should be represented as a **drop-down list** of **company names**. Create a **view model** (ProductEditViewModel) to provide the collection of SelectListItem objects (and the other data) to the view.

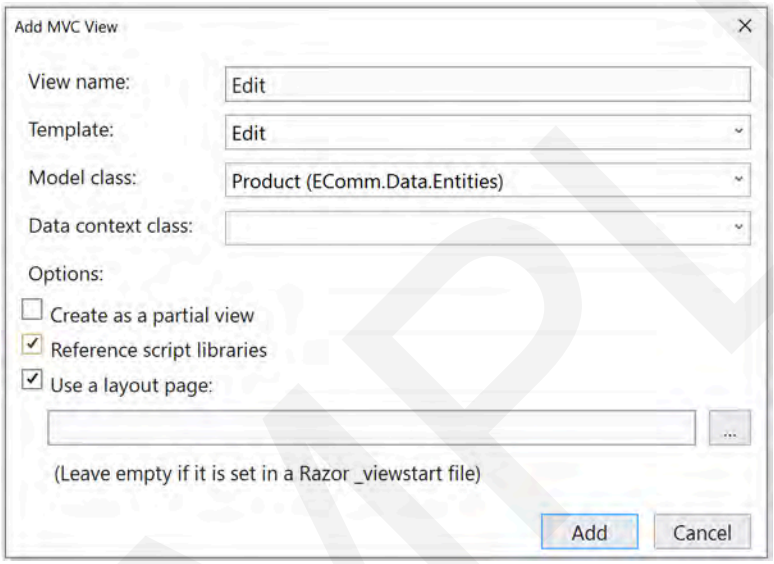
   d. Ensure that the **Edit links** in the product list take the user to the correct form.

   e. The form should display the correct data but you do **not** need to handle the submitted data at this point (that will be the next lab).

2. Open **ProductController.cs** for editing and add a new action named **Edit**.

```
[HttpGet("product/edit/{id}")]
public async Task<IActionResult> Edit(int id)
{
  var product = await _repository.GetProduct(id, includeSupplier: true);
  return View(product);
}
```

3. Right-click somewhere in the code for the **Edit** action, select **[Add View...]**, and make the following selections:

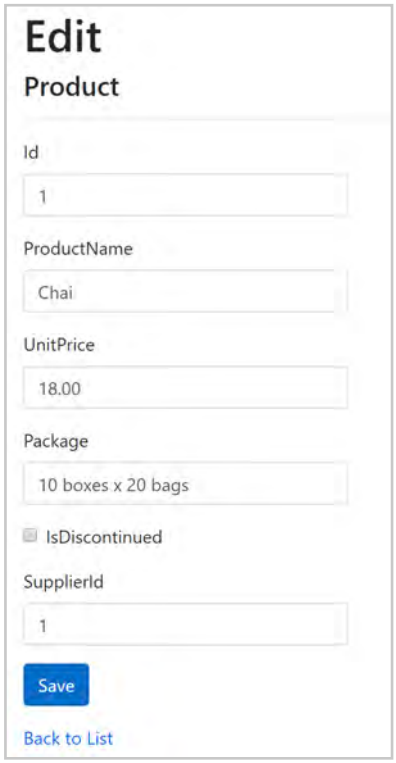


> *Notice that "Reference script libraries" is checked this time. This will insert JavaScript references into the view so that we can enable client-side data validation in a future lab.*

4. Open the **view** for the **product list** (/Views/Shared/Components/ProductList/Default.cshtml) and modify the **Edit link**.

```
@Html.ActionLink("Edit", "Edit", "Product", new { id=item.Id }) |
```

5. **Run** the application and click the Edit link for one of the products and check the edit form. The **Save** button will **not** work yet.



The form should display the correct data for the selected product. However, there is more work that needs to be done. First, the Id property should not be editable (or even displayed). Second, the last form field is SupplierId. It should be possible to change the SupplierId but the UI should be a drop-down list of company names. Since the form will now need something extra (the list of suppliers to choose from), we should change this view to accept a view model that can provide that information. It will be the job of the controller to fetch the data, construct the view model object, and pass it to the view.

6. Open **/Views/Product/Edit.cshtml** for editing and delete the **form-group** used to display the **Id** property (five lines of markup).

   Since we just deleted the input for the product's Id, we need to make sure the id parameter for the Edit action is available when the form is submitted. We could use a hidden form field but we will add the Id parameter to the action of the form instead.

7. Modify the **form tag helper** to include the **Id** of the product being edited.

```
<form asp-action="Edit" asp-route-id="@Model.Id">
```

   The next step is to define a view model that can provide the view with all of the data that it needs (including the list of suppliers for the drop-down list).

8. Add a new class to the **Models** folder in EComm.Web named **ProductEditViewModel**.

   We are adding this class to the Models folder in EComm.Web (and not EComm.Data) because it is a web-specific type that supports our view.

9. Add the **properties of a Product** to **ProductEditViewModel** plus a property for the collection of suppliers. You will need to add a **using** directive for **EComm.Data.Entities**

```
public class ProductEditViewModel
{
  public int Id { get; set; }
  public string ProductName { get; set; }
  public decimal? UnitPrice { get; set; }
  public string Package { get; set; }
  public bool IsDiscontinued { get; set; }
  public int SupplierId { get; set; }
  public Supplier Supplier { get; set; }

  public IEnumerable<Supplier> Suppliers { get; set; }
}
```

> *Instead of reimplementing the properties of a Product, we could use inheritance or composition but that would require us to change the tag helpers in the view. The approach we are using also the advantage of allowing us to remove or rename some of the product properties.*

> *The controller will set the Suppliers properly but the view will need a collection of SelectListItem objects.*

10. Add a **read-only property** to **ProductEditViewModel** that provides the **list of suppliers** as a collection of **SelectListItems**. You will need to add a **using** directive.

```
public IEnumerable<SelectListItem> SupplierItems =>
  Suppliers?.Select(s => new SelectListItem
  { Text = s.CompanyName, Value = s.Id.ToString() })
  .OrderBy(item => item.Text);
```

> *We are using the null-conditional operator (Suppliers?.) to prevent an exception from being thrown if this property is ever called by the system before the Suppliers property has been set.*

> *We are using LINQ's data projection feature here but to transform the Supplier objects into SelectListItems. There are other ways this can be done.*

11. Open **/Views/Product/Edit.cshtml** for editing and change the view to be strongly-typed to a **ProductEditViewModel**

```
@model ProductEditViewModel
```

12. Modify the ** ** element for **SupplierId** to use **Supplier** as the label, remove the **validation tag helper**, and use the **select tag helper**.

```
<div class="form-group">
  <label asp-for="Supplier" class="control-label"></label>
  <select asp-for="SupplierId" asp-items="@Model.SupplierItems"
          class="form-control"></select>
</div>
```

> *ProductController will fetch the list of all the suppliers but we don't have a method in IRepository for that yet.*

13. Add a new method to **IRepository** that can be used to get all of the **suppliers**.

```
public interface IRepository
{
  Task<IEnumerable<Product>> GetAllProducts(...);
  Task<Product> GetProduct(...);
  Task<IEnumerable<Supplier>> GetAllSuppliers();
}
```

14. Add an implementation of the **GetAllSuppliers** method to **ECommDataContext**

```
public async Task<IEnumerable<Supplier>> GetAllSuppliers()
{
  return await Suppliers.ToListAsync();
}
```

15. Open **ProductController.cs** for editing and modify the **Edit** action so that it creates an instance of **ProductEditViewModel** and passes it to the view.

```
[Route("product/edit/{id}")]
public async Task<IActionResult> Edit(int id)
{
  var product = await _repository.GetProduct(id, includeSuppliers: true);
  var suppliers = await _repository.GetAllSuppliers();

  var pvm = new ProductEditViewModel {
    Id = product.Id,
    ProductName = product.ProductName,
    UnitPrice = product.UnitPrice,
    Package = product.Package,
    IsDiscontinued = product.IsDiscontinued,
    SupplierId = product.SupplierId,
    Supplier = product.Supplier,
    Suppliers = suppliers
  };
  return View(pvm);
}
```

> *We are manually populating the ProductEditViewModel here. This code could be moved into ProductEditViewModel (constructor that takes a Product) or we could use another library to help with this (e.g. AutoMapper).*

16. **Run** the application and check the appearance of the product edit form. The supplier should be represented by a drop-down box. Of course, the save button still won't work.

End of Lab