**Setup w/ VS-2017…**

**1)** Start a New Project, for C# choose “ASP.NET Web Application (.NET Framework)”. DON'T choose .NET Core. Also choose a simple project name for a short namespace, like **BookApp**.

**Gloss over the project files…**

**2)** Make sense of the project structure:

**2a)** We have folders for **/Models, /Views,** and **/Controllers** (Models and Controllers are C# files, Views are C#+HTML files)

**2b)** The entry point of our app is in **Global.asax.cs**… a C# class inheriting from **System.Web.HttpApplication**

**2c)** We have some assets for the HTML output… CSS in **/Content**, JS in **/Scripts**, and HTML templates in **/Views**

**3)** Quick review our four main places of interest:

**3a)** **Where we define our routes?** **Global.asax.cs** can hold our routes, but in this project sample the routes are configured in a separate file: **/App\_Start/RouteConfig (Global** just calls into **RouteConfig)**

**3b)** **Where do we put our controller?** **/Controllers/HomeController** has three methods (called “actions”).

**3c)** **Where do we put our HTML view?** **/Views/Home/…**

**3d)** **Where do put our Model? /Models/**

**Figure out routing -> Controller actions**

**Naming Conventions**

Notice that our route, controller, action, AND template are connected by naming conventions. This style is called “convention over configuration”.

**4)** Build the default project and view it in the browser. First try to understand how the app takes a URL and returns a view. Start with the browser’s index URL and compare it with the **routes.MapRoute()** method. So if don’t specify anything, we go to the homepage, hence “default”.

**5)** But what about the other pages? Look at the URL’s for “About” and “Contact”. These follow the pattern of /Controller/Action. This is just like the **routes.MapRoute()**. Home=Controller, About|Contact=Controller methods (“actions”).

**6)** Look inside **HomeController**. Here we see three methods (“actions”). Note that all our URL’s, except for errors, route here. (Try visiting a URL that doesn’t match!) So if our URL contains Home/About, we go to **HomeController**, call **About()**, and the returned view is **About.cshtml**. (If we wanted to, **View()** has overloads that let us specify a different template).

**Experiment with our template and controller**

**7)** Let’s try changing a variable in our template. Look in **/Views/Home/Index.cshtml**. Here we see that there’s a block of code at the top:

@{

ViewBag.Title = "Home Page";

}

This is Razor syntax. Razor looks for the @symbol and then looks for C# syntax. Razor helps us mix C# with HTML. Inside the brackets is C#.

The above Razor syntax corresponds to our HTML <head><title> tag (which you see on the browser window, not on the rendered page). **ViewBag** is an object that can contain an arbitrary number of variables, of different types. Let’s try changing it and see the results.

**8)** Let’s go one step deeper. We can pass data from our **HomeController** to this template. Open **/Controllers/HomeController’s Index()** and let’s declare a new property and store it in **ViewBag**. Let’s make a new heading to display at the top of page, replacing the “ASP.NET”. We’ll call this **ViewBag.PageTitle**.

**ActionResult is abstract**

Using the **View()** method we are returning a subclass of **ActionResult** called **ViewResult**).

We don’t need to change our controllers return call. Notice **HomeController** returns an **ActionResult**. **ViewBag** is passed into this object for us. And as we noticed, the view template is loaded by convention, so the **Index()** action calls **View()** which returns the **Index.cshtml** view. So **ViewBag** is passed to our template for us.

So with the property assigned in the Controller, are we done? No. We need to display this variable somewhere in our template. We have to put Razor syntax somewhere in our Index template to display this (between some tags, like <h1>). We don’t need brackets for one variable, just the @ sign: **@ViewBag.PageTitle**

Here’s an example:

<div class="jumbotron">

<h1>**@ViewBag.PageTitle**</h1>

<p class="lead">ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.</p>

</div>

When we are not inside brackets **{}**, there’s no need for a semicolon **;** to end a Razor statement.

**Create a Model and pass it to the view**

**9)** We can go farther still. Let’s add a Model. The simplest model is easy: Just a C# class. No inheritance, no special namespace (though by convention we put it under **BookApp.Models**), and no special functions. A model can be simple.

Our model will will be a Book object:

BookApp.Models

public class Book

{

public int ID { get; set; }

public string BookName { get; set; }

public string AuthorName { get; set; }

public string ISBN { get; set; }

}

We should call this “Book.cs” and place it in the Models namespace.

Our Book model exists, but how can we hook it up to our app? We need to add it to the HomeController and to the template.

**10)** First, look in HomeController. To use our model we need to add its namespace at the top **using BookApp.Models;**

We also need to instantiate the model. No need for an explicit constructor. Instead we’ll use C#’s handy object initializer syntax:

Book book = new Book

{

ID = 1,

BookName = "Design Pattern by GoF",

AuthorName = "GoF",

ISBN = "NA"

};

Want the old object instantiation syntax? Sure…

Book book = new Book;

book.ID = 1;

book.BookName = "Design Pattern by GoF";

book.AuthorName = "GoF";

book.ISBN = "NA";

**Model vs. ViewBag**

Models are a little different than **ViewBag**. Our model gives us Intellisense, namespacing, and compile-time checking. This means VS checks that our controller AND view properly reference our model before we run the app. In contrast, **ViewBag** has no intellisense on properties and will compile even if your view references a variable that doesn’t exist!

For models, we need to pass them to the action method’s View() function: **View(book);**

And our controller is ready. But not the template…

**11)** Let’s go back to our Razor template Index.cshtml. Here we need to make two changes.

First, we need to tell Razor about our Book model’s namespace. We can add this at the top of the template, starting with our special @ symbol:

**@model BookApp.Models.Book**

The word “model” after the @ MUST be **LOWERCASE**.

**@model** functions like a specialized “using” statement. And did you notice the Intellisense there? VS knows our model!

Second, we can now put our model properties in our template. To do this we need some more Razor syntax. Razor offers us many methods to insert data and generate HTML. Lots of these methods fall under **@Html.\***. The one we want is **@Html.DisplayFor().**

**DisplayFor() does what?**

**DisplayFor()** will actually check to see if a specific model property has its own DisplayTemplate (i.e. HTML fragment to customize its display). Otherwise it does something like .ToString() on the property, as we use it here.

Insert them wherever you want, like so:

<div class="row">

<div class="col-md-4">

<h2>**@Html.DisplayFor(book => book.BookName)**</h2>

<ul>

<li>Book ID: **@Html.DisplayFor(book => book.BookID)** </li>

<li>Book Author: **@Html.DisplayFor(book => book.AuthorName)** </li>

</ul>

**-----------UNFINISHED-------------**

**Create a new action, and new view**