Understanding Routes

As well as models, views, and controllers, MVC applications use the ASP.NET routing system, which decides

how URLs map to controllers and actions.

A route is a rule that is used to decide how a request is handled.

When Visual Studio creates the MVC project, it adds some default routes to get you started. You can request

any of the following URLs, and they will be directed to the Index action on the HomeController:

• /

• /Home

• /Home/Index

So, when a browser requests *http://yoursite/* or *http://yoursite/Home*, it gets back the output

from HomeController’s Index method. You can try this yourself by changing the URL in the browser. At the

moment, it will be http://localhost:57628/, except that the port part may be different. If you append /

Home or /Home/Index to the URL, you will see the same Hello World result from the MVC application. ok

This is a good example of benefiting from following conventions implemented by ASP.NET Core MVC.

In this case, the convention is that I will have a controller called HomeController and it will be the starting

point for the MVC application. The default configuration that Visual Studio creates for a new project assumes

I will follow this convention. Since I did follow the convention, I automatically got support for the URLs in

the preceding list. If I had not followed the convention, I would need to modify the configuration to point to

whatever controller I had created instead. For this simple example, the default configuration is all I need. Ok

* Rendering Web Pages

The output from the previous example wasn’t HTML—it was just the string Hello World.

To produce an HTML response to a browser request, I need a view, which tells MVC how to generate a response to a request

from a browser.

using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {

public class HomeController : Controller {

public ViewResult Index() { //Render a View

return View("MyView"); // Name of the view “MyView

}

}

}

When I return a ViewResult object from an action method, I am instructing MVC to render a view. I

create the ViewResult object by calling the View method, specifying the name of the view that I want to use,

which is MyView.

* Creating View

To create the view needed for this example,

1. expand the Views folder in the Solution Explorer, right-click

the Home folder.

1. Select Add ➤ New Item
2. Drill down to the ASP.NET Core ➤ Web ➤ ASP.NET category using the left pane
3. Select the *MVC View Page*
4. *MyView.cshtml* and click the Add button to create the view ok

Copy and paste

This is an expression that will be interpreted by the Razor view engine, which processes the contents of

views and generates HTML that is sent to the browser.

This is a simple Razor expression, and it tells Razor that I chose not to use a layout, which is like a template for the HTML that will be sent to the browser

I am going to ignore Razor for the moment and come back to it later.

@{  
    Layout = null;  
}

Html normal  
<!DOCTYPE html>  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title>Index</title>  
</head>  
<body>  
    <div>  
        Hello World (from the view)  
    </div>  
</body>  
</html>

When I first edited the Index action method, it returned a string value.

This meant that MVC did nothing except pass the string value as is to the browser. Now, that the Index method returns a ViewResult, MVC renders a view and returns the HTML it produces. I told MVC which view should be used, so it used

the naming convention to find it automatically. The convention is that the view has the name of the action

method and is contained in a folder named after the controller: /Views/Home/MyView.cshtml.

https://localhost:5001/home/index/MyView.cshtml

I can return other results from action methods besides strings and ViewResult objects.

For example, if I return a RedirectResult, the browser will be redirected to another URL. If I return an

*HttpUnauthorizedResult*, I can prompt the user to log in. These objects are collectively known as action

results. The action result system lets you encapsulate and reuse common responses in actions.

* Adding Dynamic Output

The whole point of a web application platform is to construct and display dynamic output.

In MVC, it is the controller’s job to construct some data and pass it to the view, which is responsible for rendering it to HTML.

One way to pass data from the controller to the view is by using the ***ViewBag*** object, which is a member

of the Controller base class. **ViewBag** is a dynamic object to which you can assign arbitrary properties,

making those values available in whatever view is subsequently rendered. Listing 2-5 demonstrates passing

some simple dynamic data in this way in the HomeController.cs file.

namespace PartyInvites.Controllers {

public class HomeController : Controller {

public ViewResult Index() {

int hour = DateTime.Now.Hour;

**ViewBag.Greeting** = hour < 12 ? "Good Morning" : "Good Afternoon";

return View("MyView");

}

}

}

The addition to the listing is a Razor expression that is evaluated when MVC uses the view to generate

a response. When I call the View method in the controller’s Index method, MVC locates the MyView.cshtml

view file and asks the Razor view engine to parse the file’s content. Razor looks for expressions like the one

I added in the listing and processes them. In this example, processing the expression means inserting the value assigned to the ViewBag.*Greeting* property in the action method into the view.

There’s nothing special about the property name *Greeting*; you could replace this with any property name

and it would work the same, just as long as the name you use in the controller matches the name you use in the

view. You can pass multiple data values from your controller to the view by assigning values to more than one

property.

Setting the Scene

Imagine that a friend has decided to host a New Year’s Eve party and that she has asked me to create a web

app that allows her invitees to electronically RSVP. She has asked for these four key features:

• A home page that shows information about the party

• A form that can be used to RSVP

• Validation for the RSVP form, which will display a thank-you page

• A summary page that shows who is coming to the party

Designing a Data **Model**

In MVC, the M stands for model, and it is the most important part of the application. The model is the

representation of the real-world objects, processes, and rules that define the subject, known as the **domain**,

of the application. The model, often referred to as a domain model, contains the C# objects (known as domain objects) that make up the universe of the application and the methods that manipulate them.

The views and controllers expose the domain to the clients in a consistent manner, and a well-designed MVC

application starts with a well-designed model, which is then the focal point as controllers and views are

added.

**Domain** class that I will call GuestResponse

Model – Empty Class - GuestResponse.cs

Creating a Second Action and a Strongly Typed View

One of my application goals is to include an RSVP form, which means I need to define an action method

that can receive requests for that form.

A single controller class can define multiple action methods, and the convention is to group related actions

together in the same controller.

New action method to the Home controller.

*@model Class1.Models.GuestResponse*

This content is mostly HTML but with the addition of a @model Razor expression, which is used to create

a strongly typed view. A strongly typed view is intended to render a specific model type, and if I specify the

type I want to work with (the GuestResponse class in the PartyInvites.Models namespace in this case),

MVC can create some helpful shortcuts to make it easier. I will take advantage of the strongly typed feature

shortly.

@model Class1.Models.GuestResponse  
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
    <html>  
        <head>  
            <meta name="viewport" content="width=device-width" />  
                <title>RsvpForm</title>  
        </head>  
        <body>

            <**form** **asp-action**="RsvpForm" method="post">

                <p>  
                    <**label** **asp-for**="Name">Your name:</**label**>  
                    <**input** **asp-for**="Name" />  
                </p>  
                <p>  
                    <**label** **asp-for**="Email">Your email:</**label**>  
                    <**input** **asp-for**="Email" />  
                </p>  
                <p>  
                    <**label** **asp-for**="Phone">Your phone:</**label**>  
                    <**input** **asp-for**="Phone" />  
                </p>

I have defined a **label** and input element for each property of the **GuestResponse model class** (or, in

the case of the WillAttend property, a select element).

Each element is associated with the model property using the **asp-for attribute**, which is another tag helper attribute.

                <p>  
<label>Will you attend?</label>  
<**select** **asp-for**="WillAttend">  
<**option** **value**="">Choose an option</**option**>  
<**option** **value**="true">Yes, I'll be there</**option**>  
<**option** **value**="false">No, I can't come</**option**>  
</**select**>  
</p>  
<button type="submit">Submit RSVP</button>  
</**form**>  
</body>  
    </html>

Receiving Form Data

I have not yet told MVC what I want to do when the form is posted to the server. As things stand, clicking the

Submit RSVP button just clears any values you have entered into the form.

That is because the form posts

back to the *RsvpForm* action method in the *Home controller*, which just tells MVC to render the view again. ok

To receive and process submitted form data, I am going to use a core controller feature.

I will add a second

*RsvpForm* action method to create the following:

• A method that responds to **HTTP GET** requests:

A **GET** request is what a browser issues normally each time someone clicks a link.

This version of the action will be responsible for displaying the initial blank form when someone first visits /Home/

*RsvpForm*.

• A method that responds to **HTTP POST** requests:

By default, forms rendered using *Html.BeginForm()* are submitted by the browser as a

**POST request**. This version of the action will be responsible for receiving submitted data and deciding what to do

with it.

Handing **GET** and **POST** requests in separate C# methods helps to keep my controller code tidy since the

two methods have different responsibilities. Both action methods are invoked by the same URL, but MVC

makes sure that the appropriate method is called, based on whether I am dealing with a GET or POST request.

public class **HomeController** : **Controller** {

public ViewResult Index() {

int hour = DateTime.Now.Hour;

ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";

return View("MyView");

}

**[HttpGet]**

public ViewResult RsvpForm() {

return View();

}

**[HttpPost]**

public ViewResult RsvpForm(**GuestResponse** *guestResponse*) {

// TODO: store response from guest

return View();

}

}

}

I have added the **HttpGet** attribute to the existing **RsvpForm action method.**

This tells MVC that this method should be used only for GET requests.

I then added an overloaded version of the **RsvpForm method**, which accepts a **GuestResponse** object. I applied the **HttpPost** attribute to this method, which tells MVC that the new method will deal with POST requests.

I also imported the Class1.Models namespace—this is just so I can refer to the

GuestResponse model type without needing to qualify the class name.