Lab: Routing

Step 0: MVC Application Conventions (Review)

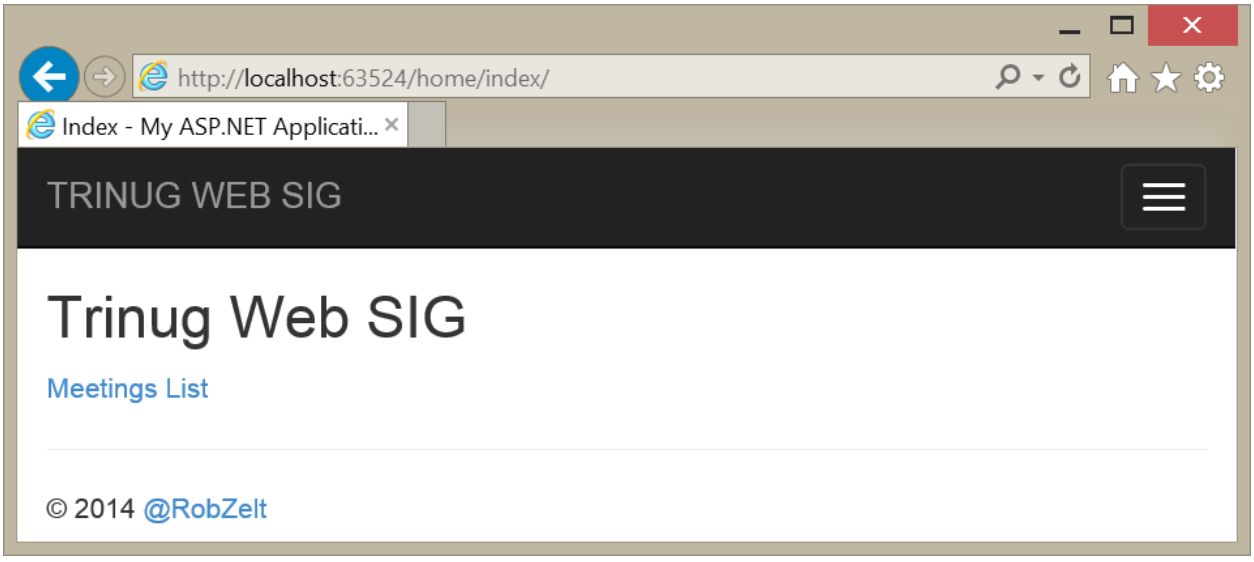
One of the primary characteristics of ASP.NET MVC is "convention over configuration." In simple terms, if you play by the rules, things will "just work." Some of those rules are:

1. Controllers go in the Controllers folder. (They can also go elsewhere, but let's keep things simple for now.)
2. Controllers are named "xxxxController," where "xxxx" is the controller name.
3. Public methods in a controller are called "actions."
4. Views go in the Views folder.
5. Every controller has a corresponding folder in the Views folder.
6. If an action needs a view, the view has the same name as the action, and that view lives in a subfolder in the Views folder, where that subfolder has the same name as the controller.

In other words, if the Home controller has an action called Index, and that action has a view, then

1. The HomeController class has a public method named Index.
2. The Views folder has a subfolder called Home.
3. The Home folder has a view called Index.cshtml (assuming Razor view engine and C#).

Following this convention, we know that we can navigate to the Index view of the Home controller, like this:



We also know that this "Home" controller and its "Index" action and view are the default, and we don't have to specify them at all:

How does this work?

Following the MVC conventions, you can write a lot of web pages without knowing how this works, but sooner or later, you'll need to know more.

That's when you need to know about routing. Routing is the step where ASP.NET takes the incoming request, and determines which controller to load, and which action to call.

It's not magic, it's just infrastructure. The good news is that you can customize your routing, and you can use that to customize your URLs. But first, you need to know how the default routing works.

Step 1: Default Routing

ASP.NET sets up routing for your application at application start, which runs in global.asax.cs:

```
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcDemo
{
    public class MvcApplication : HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

That last line is the important one. It calls the RegisterRoutes method in RouteConfig. This is a generated class that Visual Studio MVC templates create when you create your project. It lives in the App_Start folder. Let's take a look:

```
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcDemo
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index",
 id = UrlParameter.Optional }
            );
        }
    }
}
```

On the surface, RegisterRoutes is pretty simple. It takes a RouteCollection, and adds routes to it. MVC uses these routes to decide what controller and action to call.

IgnoreRoute does exactly what you think it should do; it tells MVC to ignore any URL that matches that strange-looking pattern. We'll come back to that.

MapRoute is much more interesting, and has five overloads. It tells MVC, "Here's a routing pattern to match." MVC will use this pattern to match incoming requests to your controllers and actions.

As you'll see, you can call MapRoute multiple times, and give it a number of different routes. MVC will try to match each route, in the order you've called MapRoute, and when it can match a route to a controller and action, it will stop searching, and use that route.

This is important! The route you see above is the default route. Until you know what you're doing here (and usually thereafter as well), this should always be the last route you add to the RouteCollection, and you should not change it in any way.

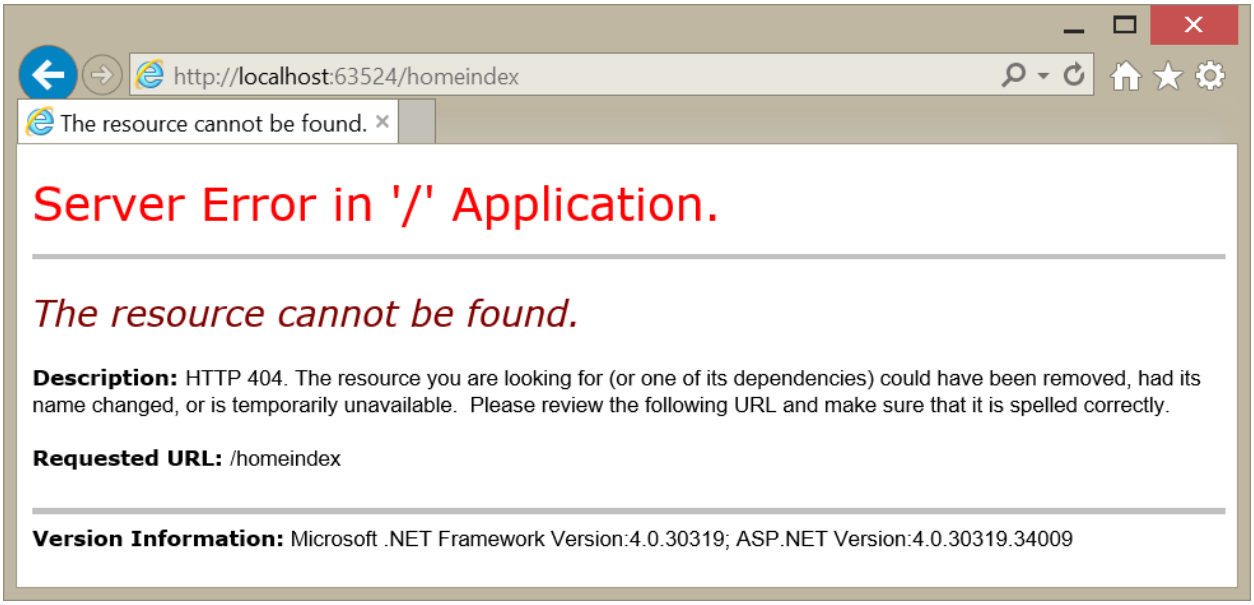
Let's take a look at this route. It has three parameters, a name, a URL pattern, and defaults.

Lagniappe: Did you know that you can specify parameter names in a method call? This was added in C# 4.0.

Names are optional. You can use a route name to look up a route from your application, and make changes to it. Some people think this is a bad idea, though.

The URL pattern is crucial. The curly braces enclose names for substitution parameters. “Controller” and “action” are names that MVC knows about, and it will use these to match a controller class and action method. The “id” parameter does not have any special meaning.

Anything that is not enclosed in curly braces must be present in the URL “as is”. In the default route, these are the slashes that separate the parameters. These are important! “home/index” will match the home controller and index action. “homeindex” will not.

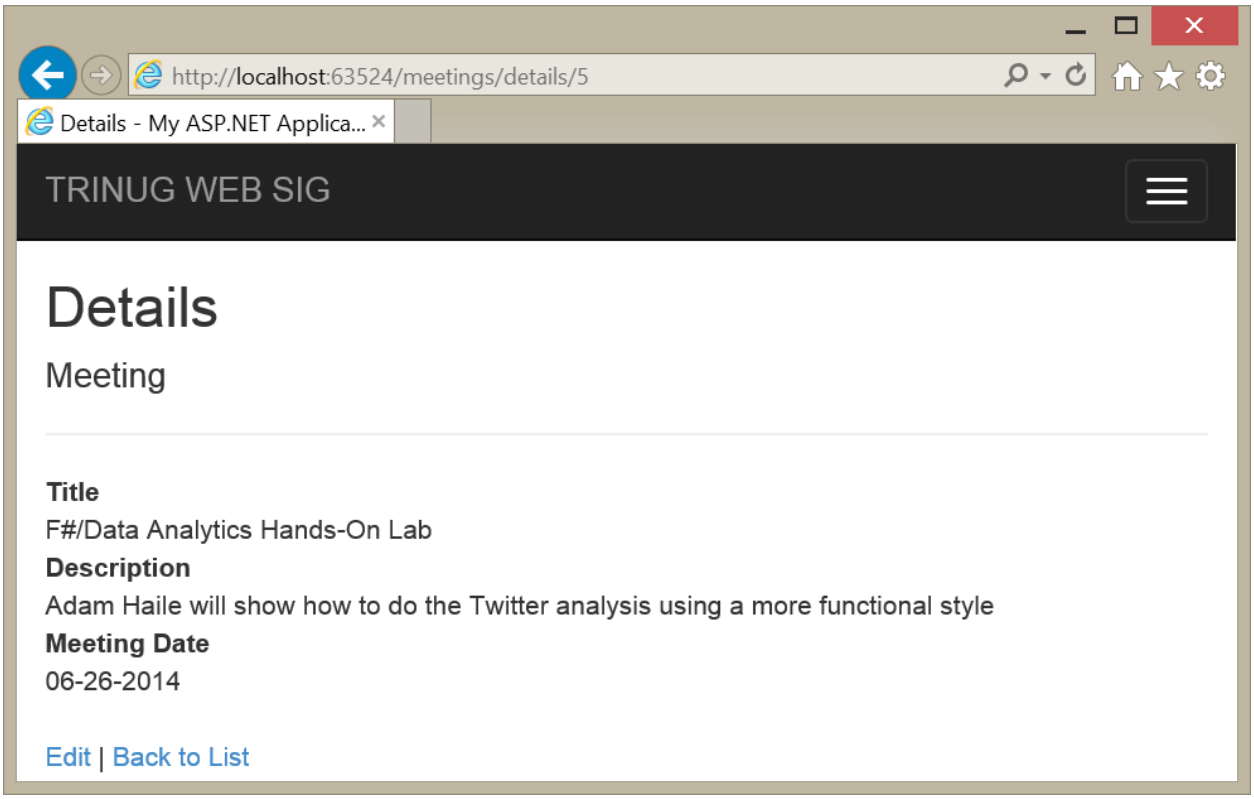


The defaults parameter tells the MVC route engine what to do when URL parameters are not present. This is an anonymous class, and the properties should match the parameter names in the URL. In the default route definition, the default for “controller” is “Home”, and the default for “action” is “Index”. “id” has an interesting default: UrlParameter.Optional. If the requested action has an “id” parameter, then the id value from the request will be passed to the method as a parameter value. If the requested URL doesn’t have an “id”, two things happen. First, the “id” isn’t necessary for a route to match. Second, the route won’t match any action that requires the “id” parameter.

Step 2: Using the Default Route

Let’s take a look, using the Details action in the Meetings controller:

```
// GET: Meetings/Details/5
public ActionResult Details(int id)
{
    //Load the specified Meeting by the Meeting ID provided.
    Meeting meeting = meetingRepository.Get(id);
    return View(meeting);
}
```

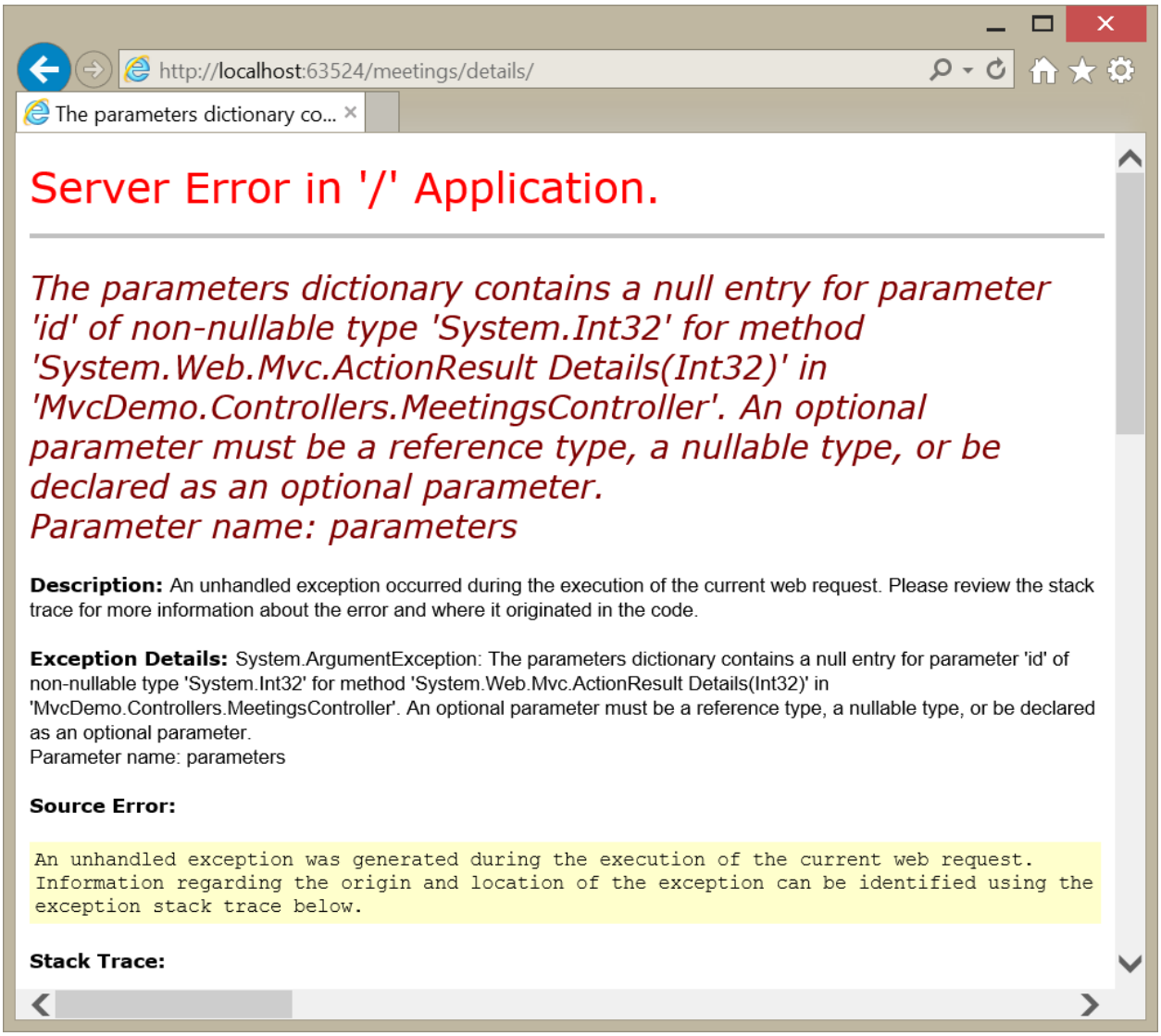


It works exactly like you'd expect it to work. The "controller" parameter maps to "meetings"; the "action" parameter maps to "details", and the "id" parameter maps to 5.

As we've noted, "id" is optional. What happens if we leave it off?

We get the dreaded Yellow Page of Death, that's what happens! However, read the error message. A lot of that should make sense now. MVC tried to route to the MeetingsController, and its Details(Int32) method.

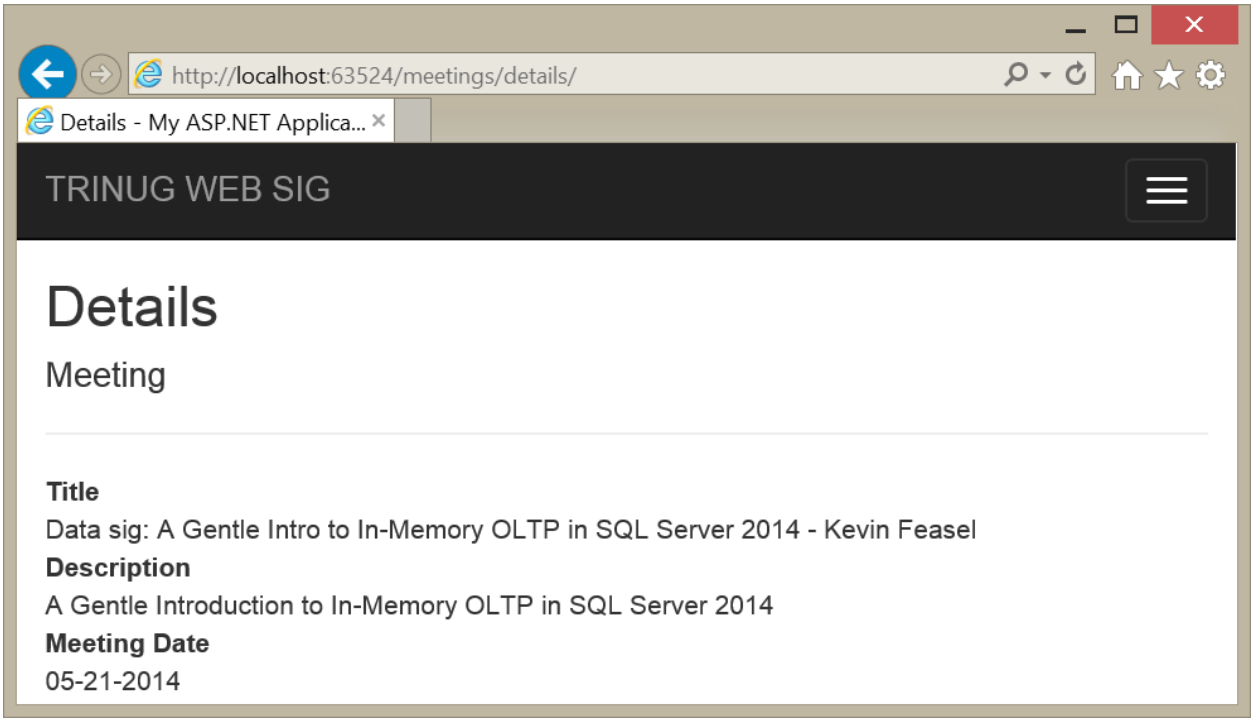
But there are some interesting hints in the error message, as well. "An optional parameter must be a reference type, a nullable type, or be declared as an optional parameter." Int32 isn't a reference type, but we can try the other two.



Let's try an optional parameter first.

```
// GET: Meetings/Details/5
public ActionResult Details(int id = 1)
{
    //Load the specified Meeting by the Meeting ID provided.
    Meeting meeting = meetingRepository.Get(id);
    return View(meeting);
}
```

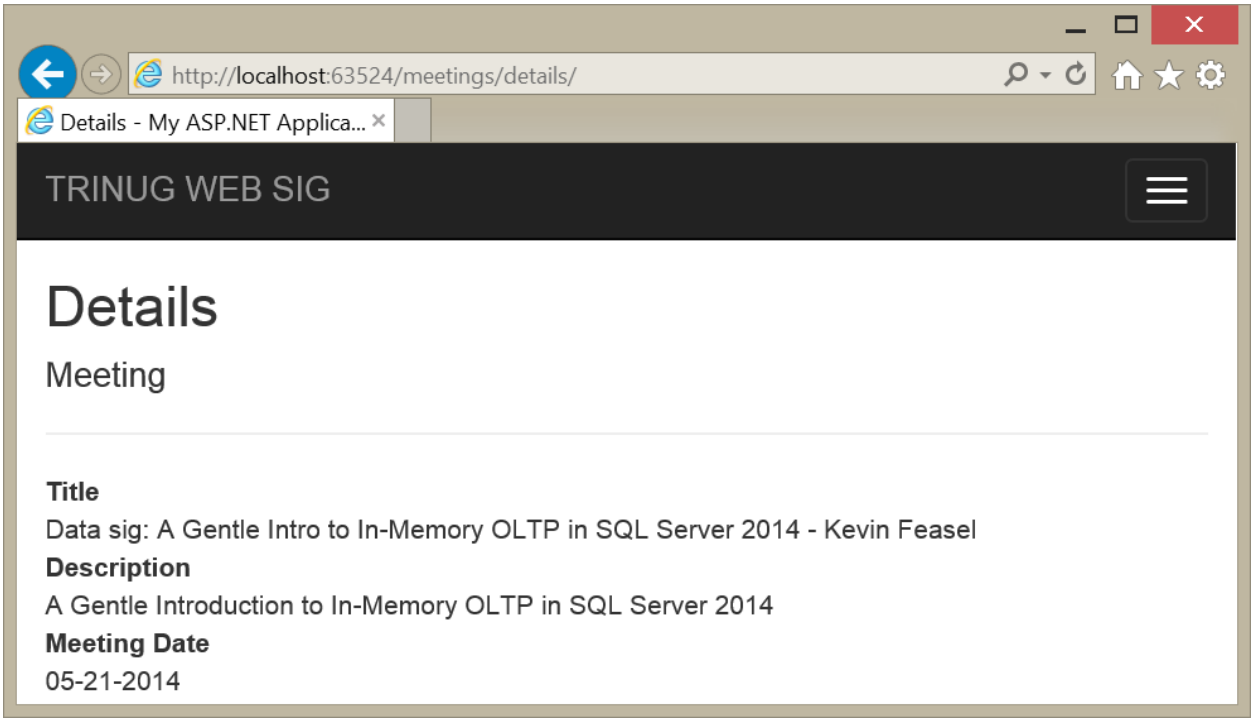
We've supplied a default value now for "id", so it is now optional, but not nullable.



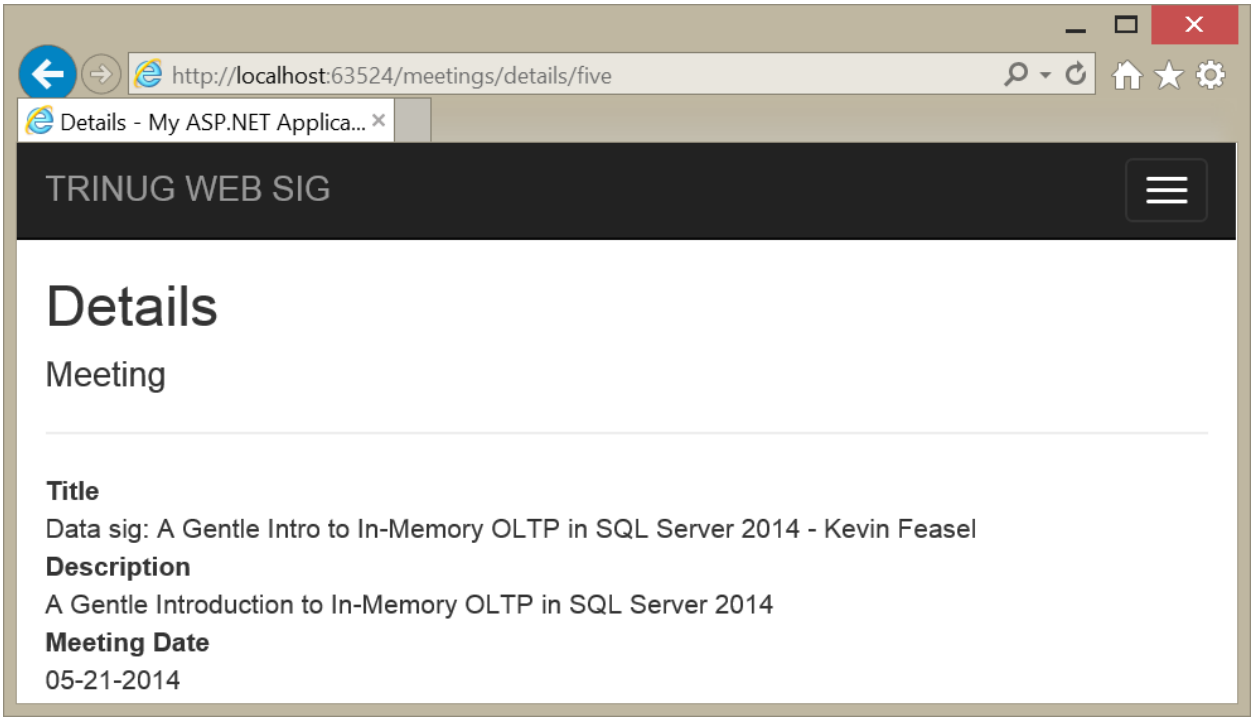
Now let's try making "id" nullable:

```
// GET: Meetings/Details/5
public ActionResult Details(int? id)
{
    //Load the specified Meeting by the Meeting ID provided.
    Meeting meeting = meetingRepository.Get(id.HasValue ? id.Value : 1
);
    return View(meeting);
}
```

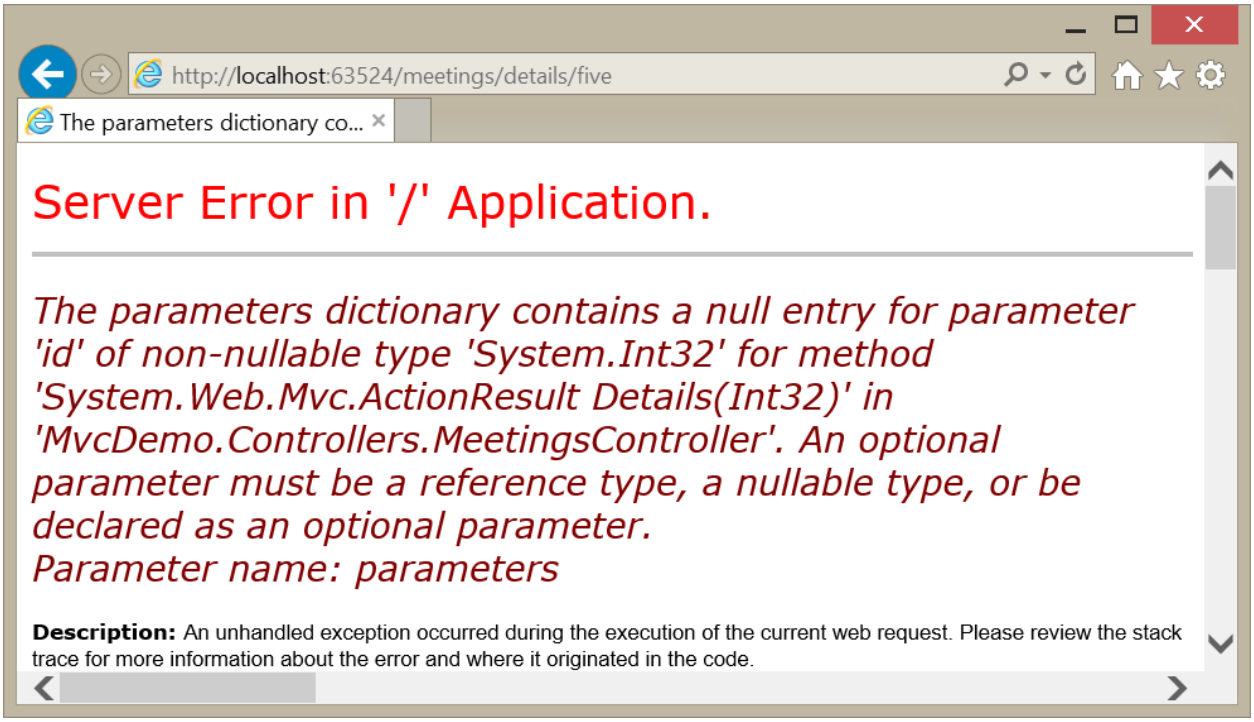
This works too, but note that we now have to code the method to handle null values. Making the parameter optional by providing a default value keeps the code easier to read.



What happens if the id value doesn't match the type?



When the "id" value is a string, and the parameter is optional or nullable, we get the action's default value, which is meeting 1. If we go back to the original code, which required Int32, then we might not like the result:

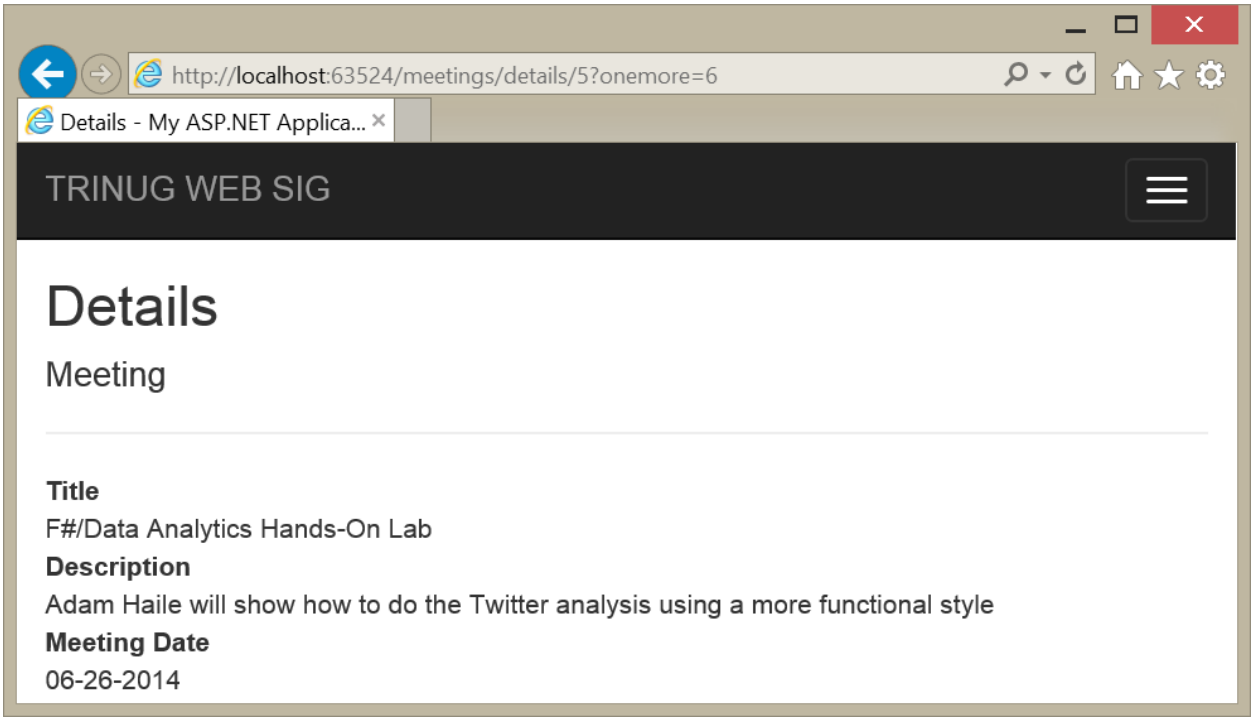


Boom. The Yellow Page of Death returns. Note that this could work, though, if there is an override for the Details action that takes a string parameter called “id”.

Can you pass additional parameters, and still map to the routing?

That didn't work so well, did it? The requested URL didn't match any registered routing pattern, and the result is a 404 – page (or resource) not found.

What about a query string?



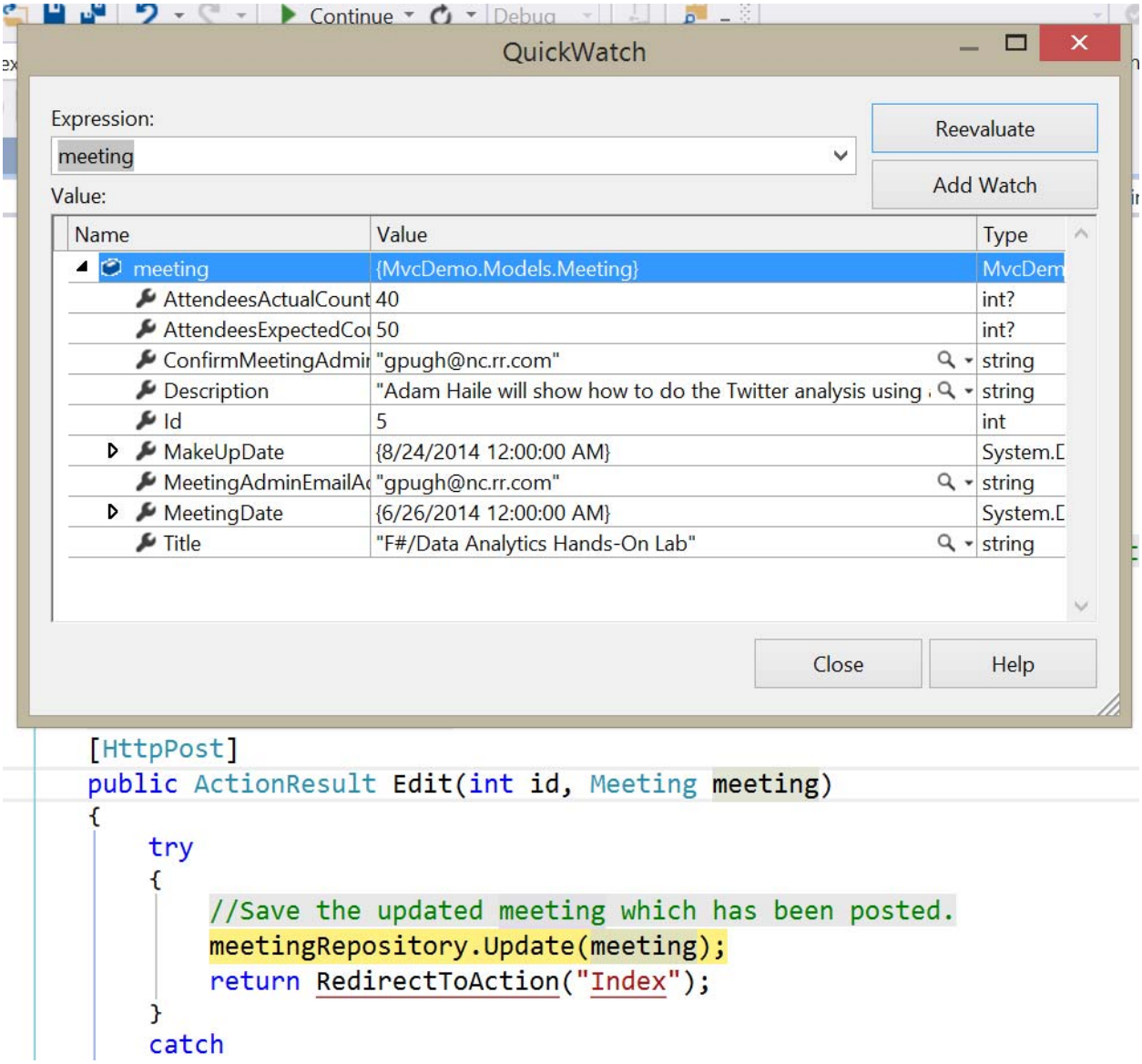
That works, and you can dig the query string out of the HttpRequest object. It's a little more work than with web forms, though, and we're breaking the MVC patterns here.

Let's look at one more case. Last month, we created and edited meetings, which resulted in a HTTP POST request, with all the form values in a request header. Look at the action that handles the POST:

```
// POST: Meetings/Edit/5
[HttpPost]
public ActionResult Edit(int id, Meeting meeting)
{
    try
    {
        //Save the updated meeting which has been posted.
        meetingRepository.Update(meeting);
        return RedirectToAction("Index");
    }
    catch
    {
        return View(meeting);
    }
}
```

We've already seen MVC reject a URL with an extra parameter. What about an action that has a parameter not specified in the route?

First, Meeting is a reference type, so the router can pass a null value, so there's no problem matching the route pattern. However, this is a POST request, so something special happens here. Let's set a break point in this method, edit an existing meeting, and see what happens.



So routing worked, even though there is no route registered that has a parameter called "meeting."

Something else happened, though, and it's pretty cool. All the HTML control parameters from the Edit form are in a response header, with names that match the Meeting class property names. That's an advantage of using the HtmlHelpers to set up a form. When handling the POST request, MVC creates a Meeting class, which is defined in the Models folder, and binds the form values to the appropriate

properties. This I called “Model Binding”, and it is a very powerful way to pass data from a form to your MVC app, without having to write the code to fetch all those values.

Open the Edit.cshtml file, and see how the model binding is set up, using those HtmlHelpers.

Step 3: Add a Route Definition

Let’s add a new route that provides a shortcut for people who know exactly what meeting they want to know about. (We’ll pretend they have the id values memorized.)

```
routes.MapRoute(
    name: "Meeting",
    url: "Meeting{id}",
    defaults: new { controller = "Meetings", action = "Details",
        id = UrlParameter.Optional },
    constraints: new { id = @"\d+" }
    );

routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index",
        id = UrlParameter.Optional }
);
```

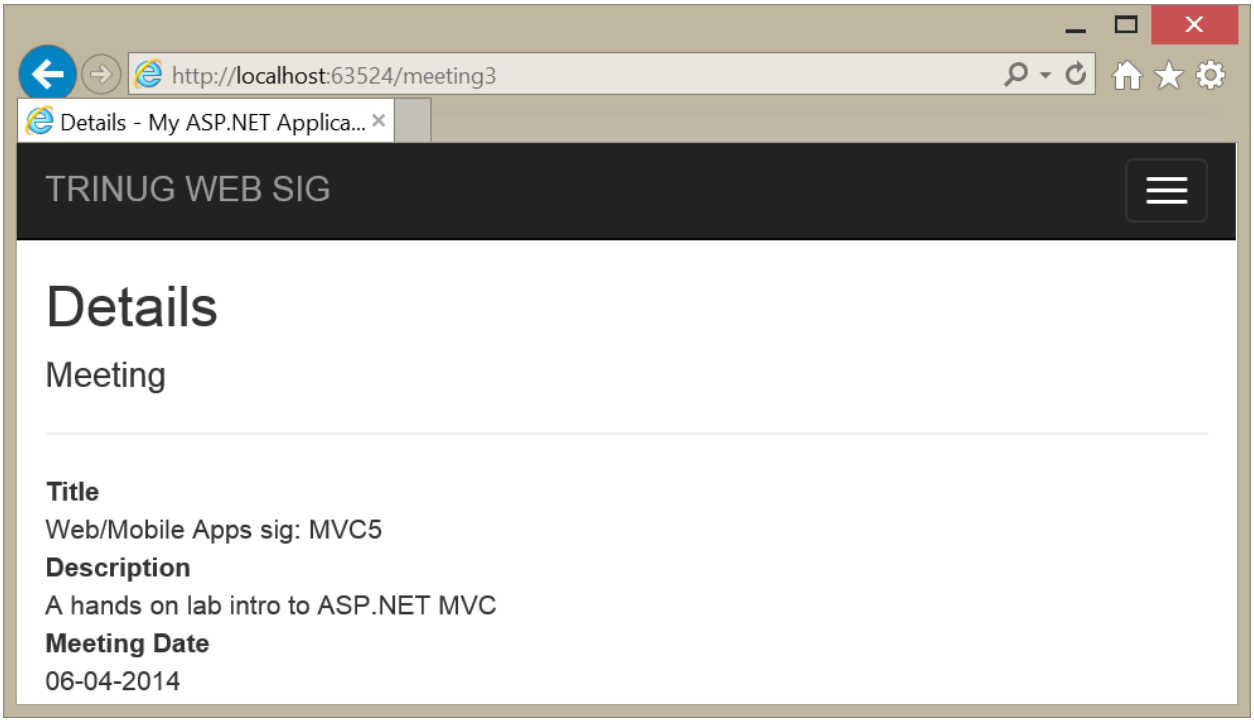
First, note that the new route definition goes before the default route. All new routes should always be added before the default route, because the default route should always be there to catch anything the new routes don’t.

Second, let’s look at the URL parameter, because it’s interesting. The word “Meeting” is not in curly braces, so it is not a substitution parameter. It must appear in the URL exactly as the word “meeting”, not case sensitive. Also, there is no slash between “meeting” and the “id” parameter (in curly braces), so there can be no slash in that position in the URL.

Third, the defaults parameter still defines default values for “controller” and “action”, even though these are not part of the URL. In this case, a route match will always map to the Meetings controller and the Details action.

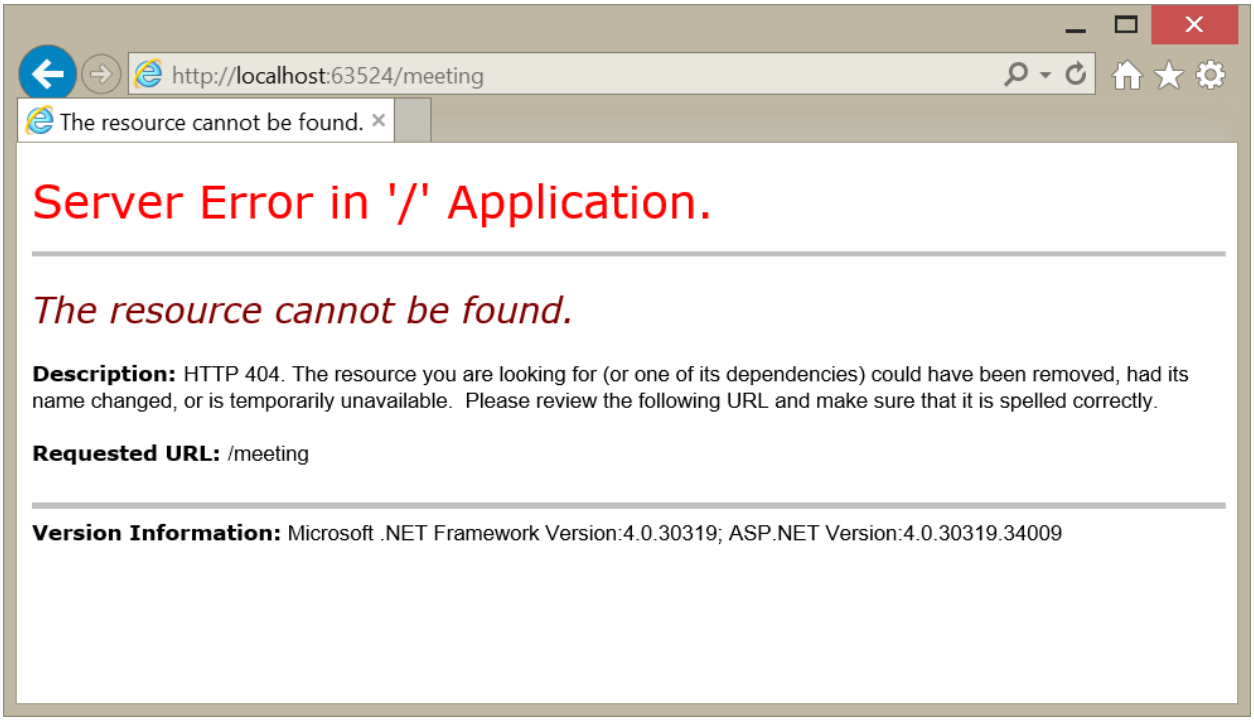
Fourth, we’ve added a new parameter on MapRoute. The constraints parameter lets us use regular expressions to constrain the value of a parameter – in this case, the “id” parameter must be a string of one or more numbers.

So what does this look like in action?



We have a match with the new route!

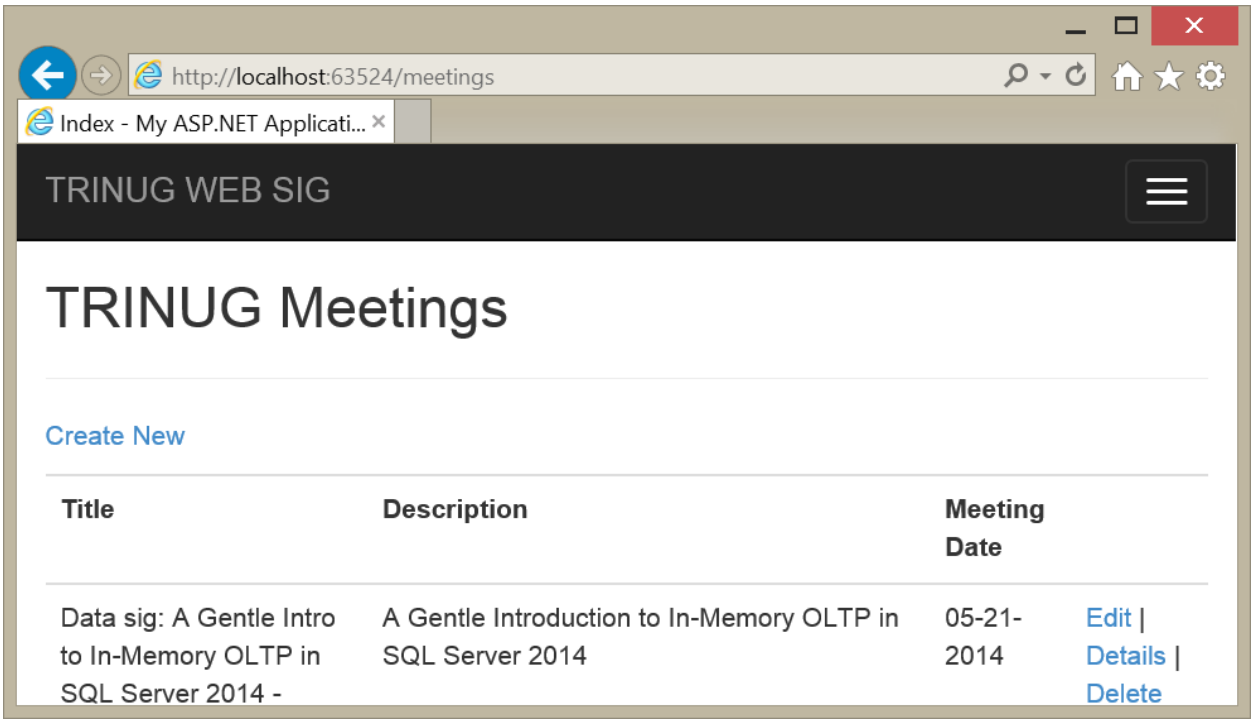
What happens if we don't supply a meeting number?



There's no match, and a 404 – no resource, page, or route. What happened?

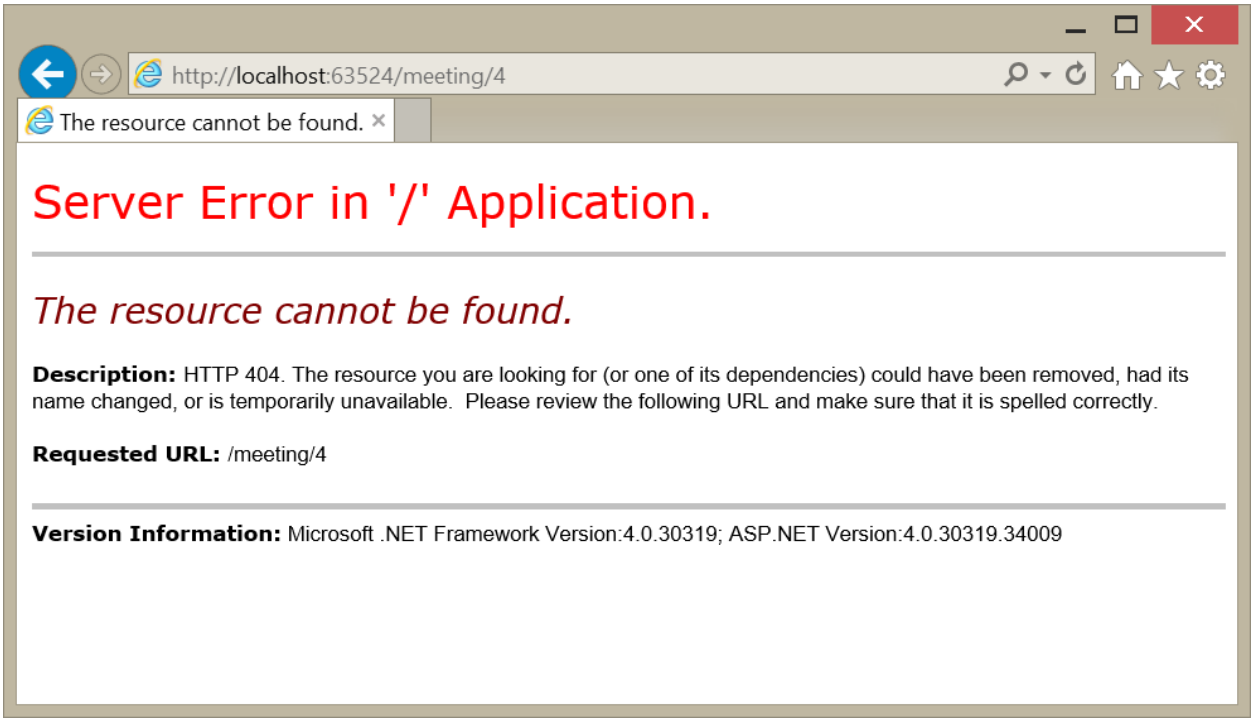
The defaults parameter specified that “id” is optional, but the constraints specifies that “id” must be a string of digits, at least one digit long. Therefore, the “id” parameter is not truly optional, the route did not match, and MVC moved on to try the default route.

The default route did not match because there is no controller named “Meeting”. What happens if we use the correct controller name?



Is this a surprise? No, because routing drops through to the default route, and it matches. We have a “meetings” controller, and we have a default value for “action” (“Index”), so we route to the MeetingsController and its Index method.

What happens if someone puts the slash in the URL? Isn’t that what someone might expect?

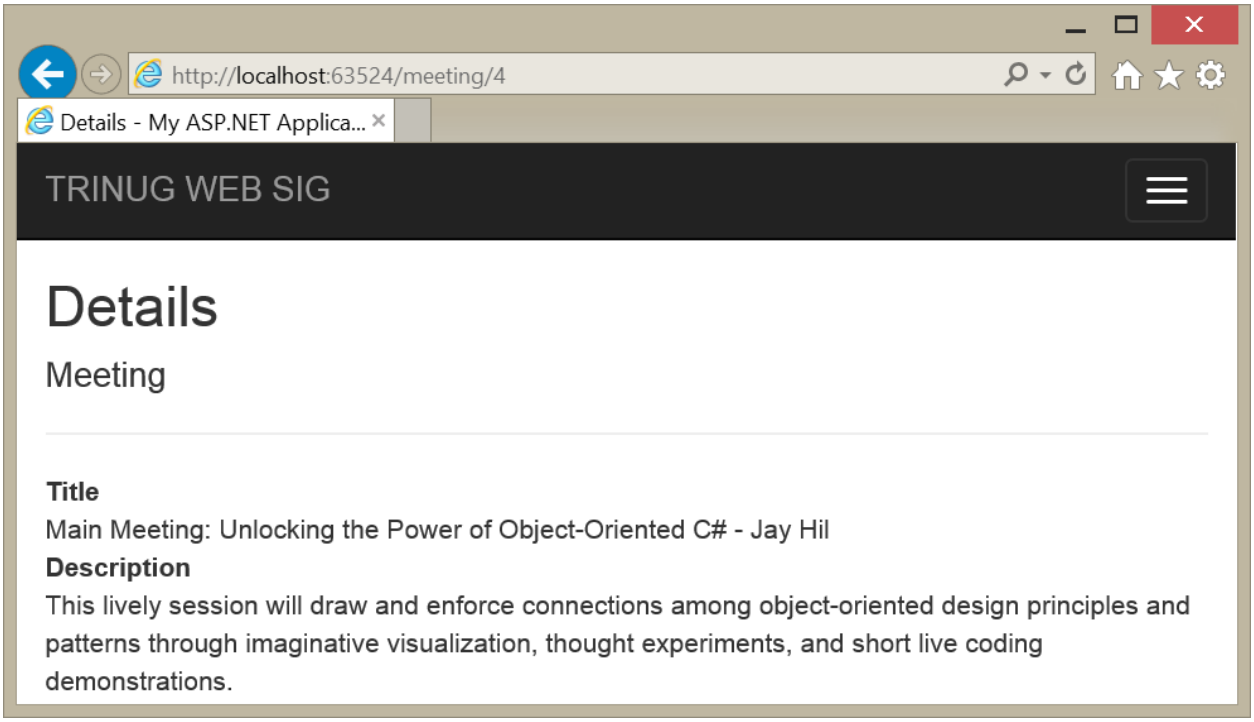


We get a 404 again, because the URL parameter didn't contain a slash. But we can fix that by adding another route.

```
routes.MapRoute(
    name: "Meeting-slash-id",
    url: "Meeting/{id}",
    defaults: new { controller = "Meetings", action = "Details",
        id = UrlParameter.Optional },
    constraints: new { id = @"\d+" }
    );
```

There's no conflict between this route and our other new route; either the URL has a slash, or it doesn't. So it can go either before or after our first new route, as long as it comes before the default route.

The big question is, does it work?



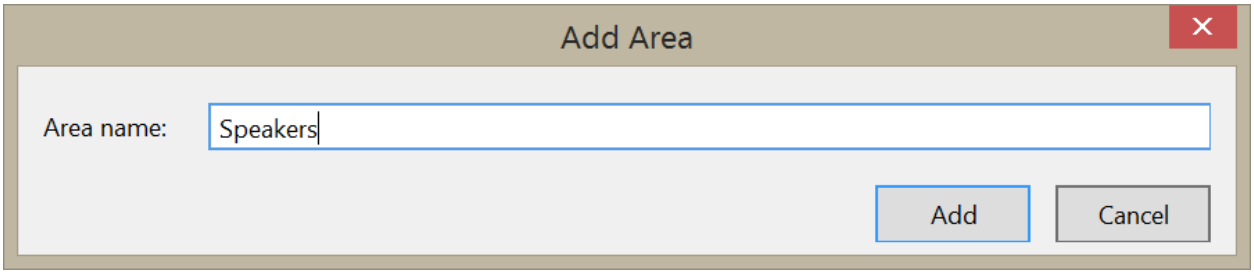
Yes, it works.

Step 4: Add an Area

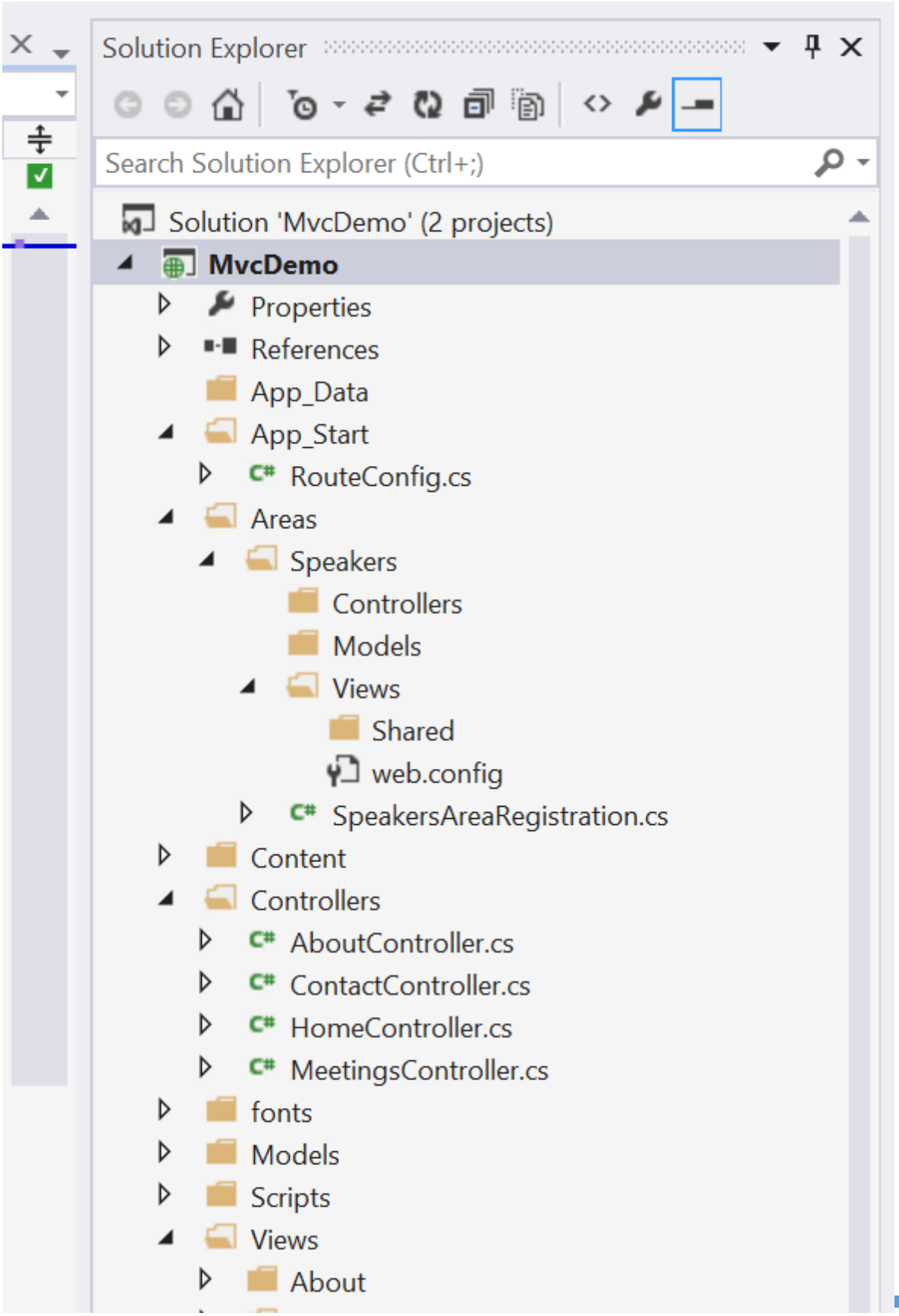
Web sites have a way of growing. I work on a commercial application that has over 1000 pages. How do you manage that kind of growth in MVC?

The answer is that you create an area. Let's say that we want to add a section of our site for our speakers. Maybe they want forums, or a place to host code, or whatever. For now, we just want to add that Speakers area, so that we have room to grow.

Right-click on the project in the Solutions Explorer. In the context menu, click on "Add…" and in the next context menu, click on Area. You get a prompt for the area name:



You will get a new folder in your project, named Areas. It has a subfolder called Speakers. The Speakers folder has lots of interesting things!


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'MvcDemo' (2 projects)
MvcDemo
Properties
References
App_Data
App_Start
RouteConfig.cs
Areas
Speakers
Controllers
Models
Views
Shared
web.config
SpeakersAreaRegistration.cs
Content
Controllers
AboutController.cs
ContactController.cs
HomeController.cs
MeetingsController.cs
fonts
Models
Scripts
Views
About

The Speakers area has its own folders for Controllers, Views, and Models. The Views folder has its own folder for shared views, and its own web.config. Speakers also has its own routing:

```
using System.Web.Mvc;

namespace MvcDemo.Areas.Speakers
{
    public class SpeakersAreaRegistration : AreaRegistration
    {
        public override string AreaName
        {
            get
            {
                return "Speakers";
            }
        }

        public override void RegisterArea(AreaRegistrationContext cont
ext)
        {
            context.MapRoute(
                "Speakers_default",
                "Speakers/{controller}/{action}/{id}",
                new { action = "Index", id = UrlParameter.Optional }
            );
        }
    }
}
```

Note that this route definition lives in an AreaRegistrationContext, not the RouteCollection that RouteConfig uses. Also take a look at the route definition. The URL begins with the constant string “Speaker”, our area name, and it contains no default value for “controller”.

Before we go any further let’s look back at that Application_Start method in global.asax.cs.

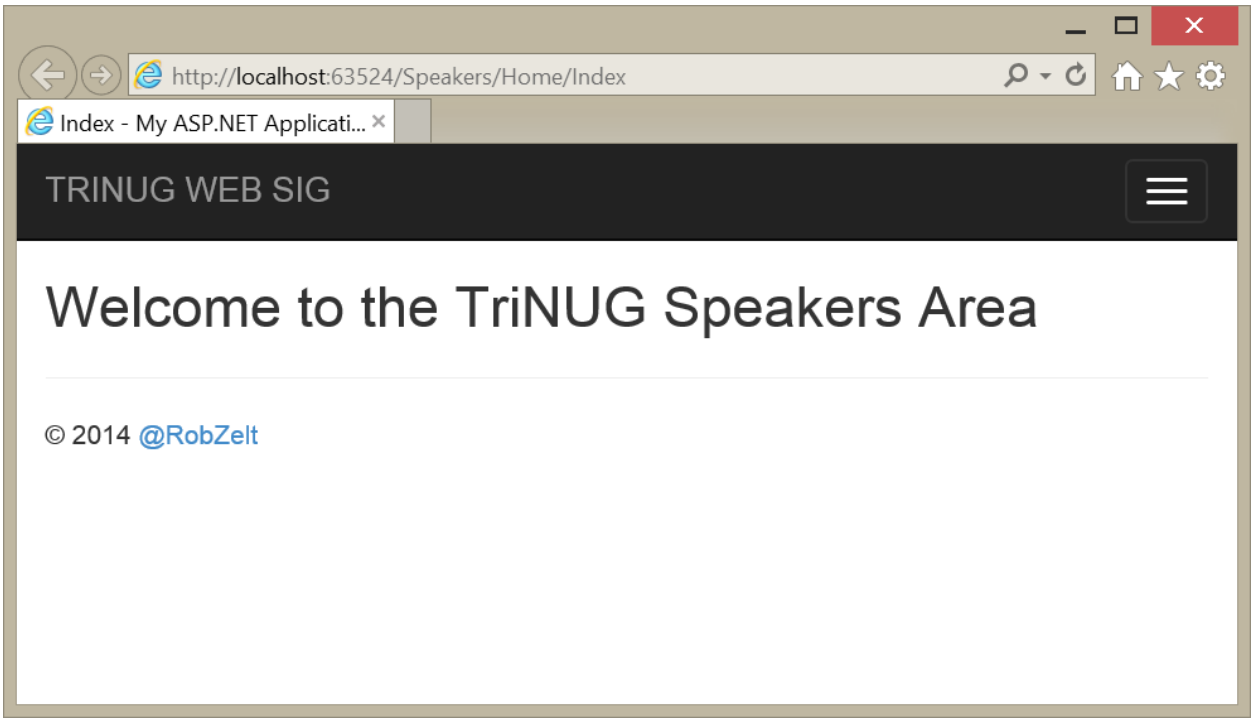
```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    RouteConfig.RegisterRoutes(RouteTable.Routes);
}
```

There’s a method there to register all areas. That was already in global.asax; we didn’t have to add that, and neither did Visual Studio.

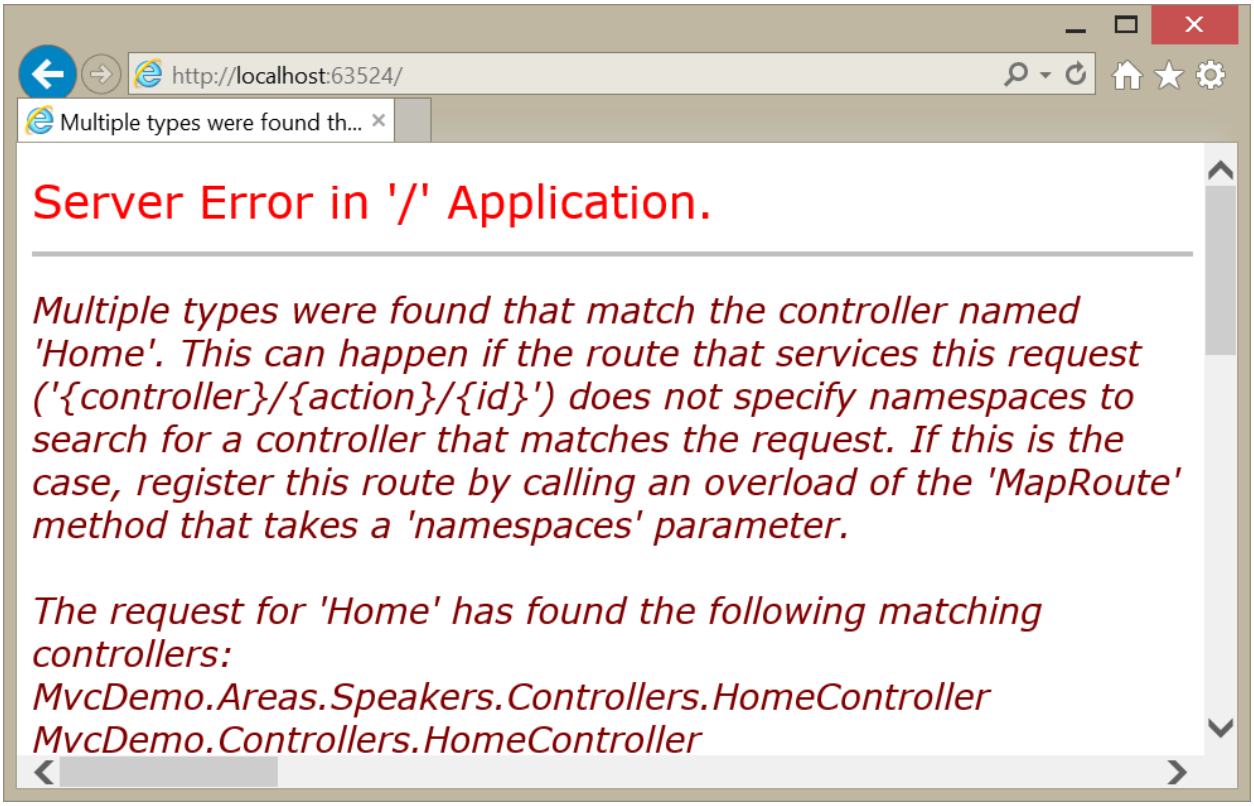
Let's add a HomeController in the Speakers/Controllers folder, and add a View for its Index action. When we create the view, we can use the Layout view in the main Views folder.

In the view, we'll just change the "index" heading to "Welcome to the TriNUG Speakers Area."

Let's start up the app. Depending on what was open in Visual Studio, you might get this:



Or this:



You might not have expected either one.

In the first case, we opened to the Home controller's Index action in the Speakers area. That's because when I started the app, I had that index.cshtml page open. If you want Visual Studio to always open on the home page of your application, here's how you do that:

1. Right-click on the project name in the Solution Explorer.
2. Open Properties.
3. On the "web" tab, click on "Specific" page.
4. Don't enter a page name.
5. Save

Now you'll open on the home page by default, except that doesn't seem to work either. MVC found two home controllers! By default, the main routing can open controllers anywhere in your project, while the area routing can only open controllers in its own area. So the Speakers home page works, but the main one doesn't. We need to fix this.

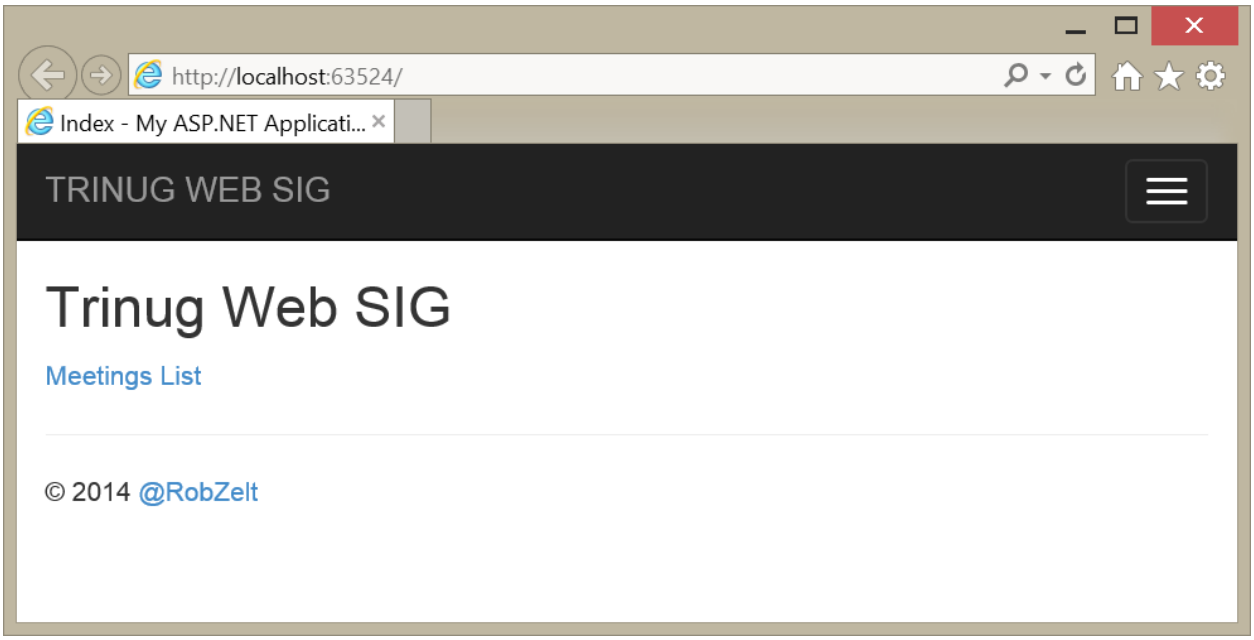
Let's go back to the RouteConfig and modify our default route. (Remember when I told you not to modify the default route? Now I'm telling you to modify the default route.)

```
routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home", action = "Index", id = UrlPar
ameter.Optional },
```

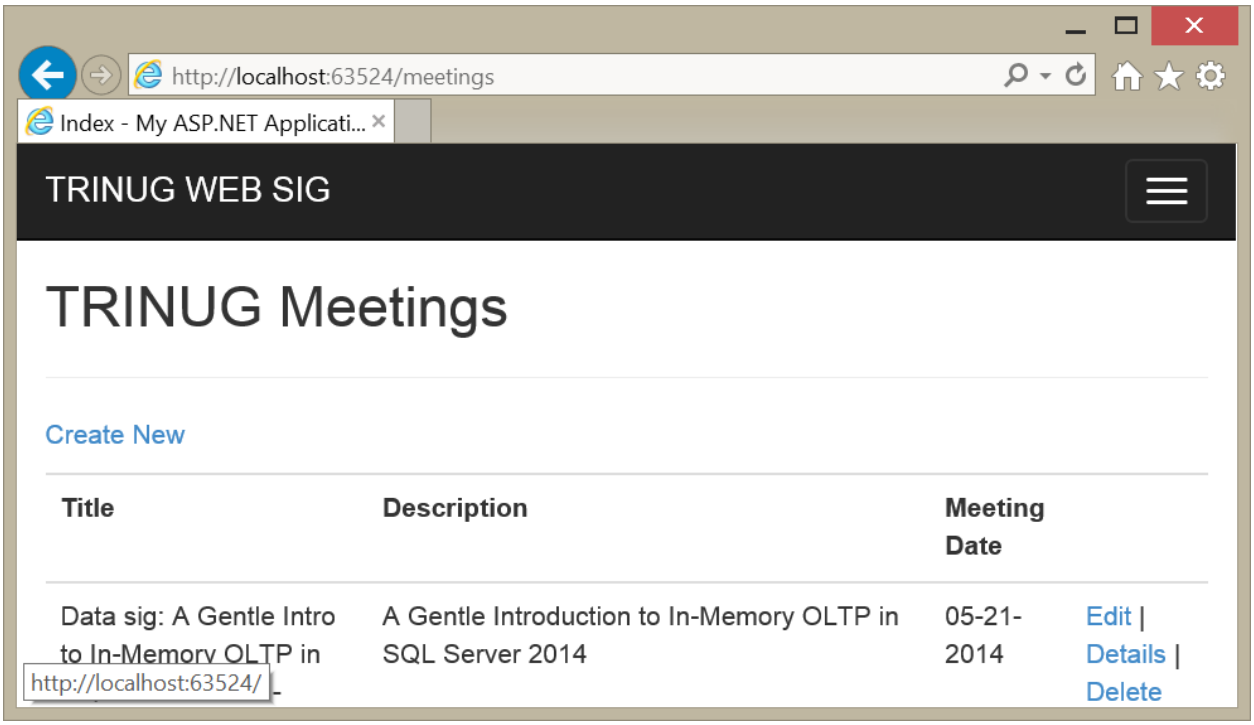
```
    namespaces: new[] { "MvcDemo.Controllers" }
);
```

We've added a "namespaces" parameter to the route. This is an array of strings that lists the namespaces for controllers in the default route. Now the default route matches only this namespace. (You can list more than one namespace, but for now this is all we need.)

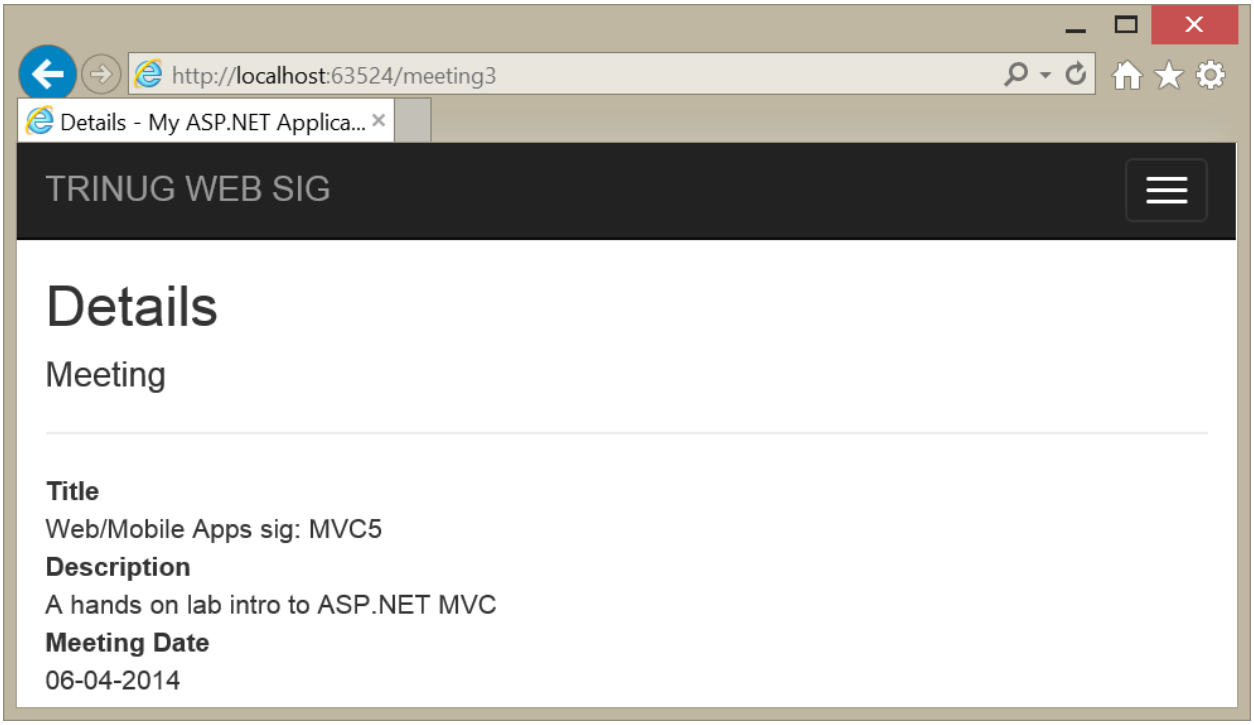
And…. It works.



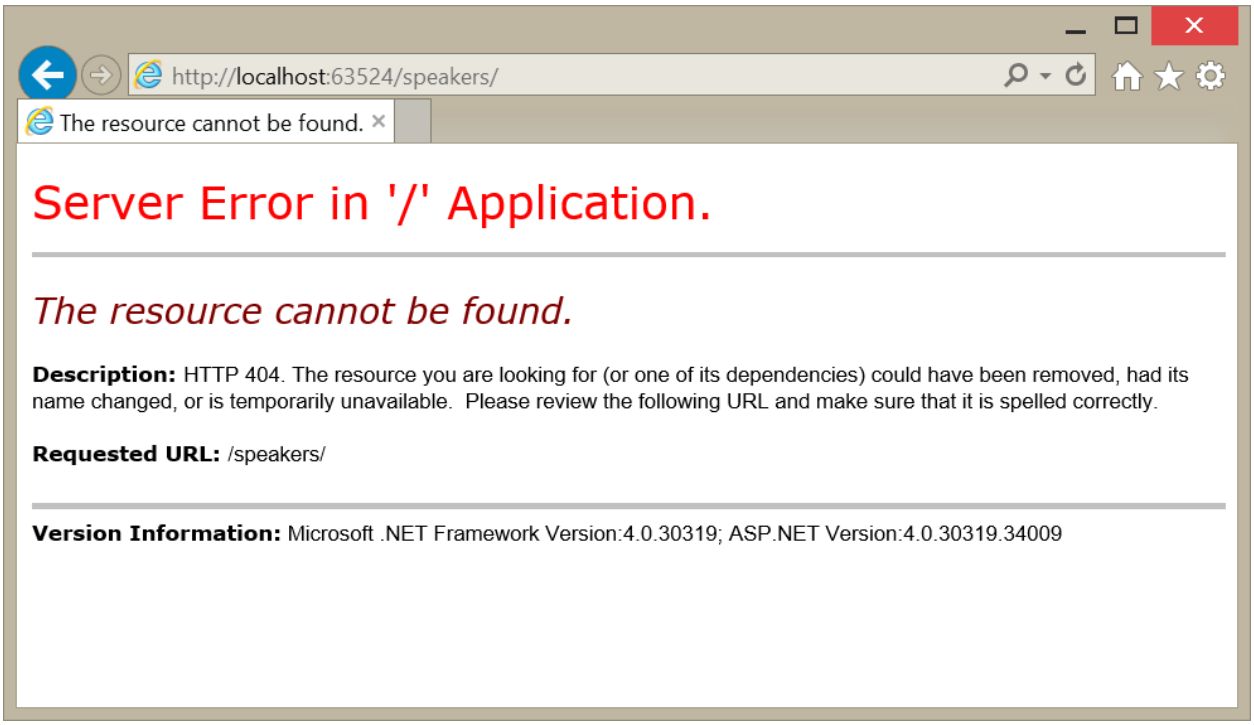
So what did we break?



Default routing works.



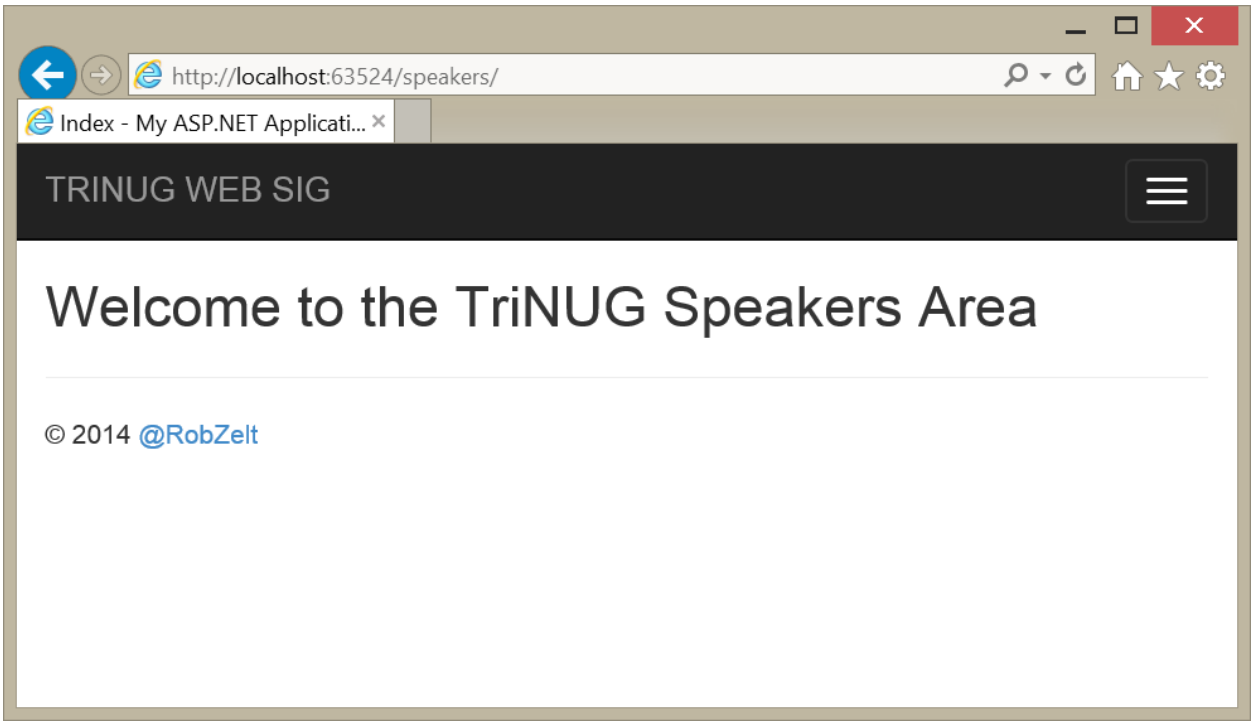
Our custom routing for specific meetings works.



Taking a default value for the controller on the Speakers area does not work. Why? The routing for the Speakers area didn't include a default controller. We can fix that.

```
public override void RegisterArea(AreaRegistrationContext context)
{
    context.MapRoute(
        "Speakers_default",
        "Speakers/{controller}/{action}/{id}",
        new { controller = "Home", action = "Index",
            id = UrlParameter.Optional }
    );
}
```

Now the Speakers area should default to the HomeController.



Now it works.

Next Steps

There are many more things you can do with routing:

1. Ignore routes (see your RouteConfig file).
2. Route to files or legacy content.
3. Add routing configuration as attributes on controllers or actions.
4. Add custom route handlers.
5. Redirect using routes.

Let's close by using RedirectToAction to provide some basic error handling.

```
// GET: Meetings/Details/5
public ActionResult Details(int id)
{
    //Load the specified Meeting by the Meeting ID provided.
    Meeting meeting = meetingRepository.Get(id);
    if (meeting == null)
    {
        return RedirectToAction("Index");
    }
    return View(meeting);
}
```