Exercise 12 : Routing

Objective

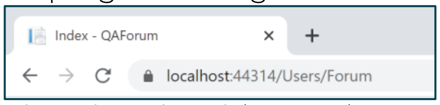
In this exercise you will create a series of routes to map to URLs that follow a certain structure. You will implement route constraints to ensure that routes contain valid data. You will install MyTestedAsp to test your routes.

This exercise will take around 30 minutes.

Referenced material

This exercise is based on material from the chapter "Routing”.

Refining the existing routes

	Open the 'Begin' solution in Visual Studio and compile (Shift Ctrl+B). The ‘Begin’ solution is identical to the ‘End’ solution from the previous lab.
	Run your program and go to the Forums page. Look at the URL.  localhost:44314/Users/Forum It’s very handy to be able to split our code into areas, but it creates URLs that are more complex than they need to be. Let’s see if we can remove the /Users part of the URL.

	Open Program.cs. Add a new endpoint, as shown here:
	`    app.UseAuthorization();`
	
	`    app.MapControllerRoute(`
	`        name: "users-area",`
	`        pattern: "{controller}/{action=Index}/{id?}",`
	`        defaults: new { area = "Users" });`
	
	`app.MapControllerRoute(`
	`    name: "areas",                          ....`
	Note the following points:
	Since the area is not specified as part of the URL, it needs to be included in the defaults: parameter.
	
	This endpoint is more specific than any other routes that we already have, so it has to be listed first, before the existing endpoints.
	
	The URL /Home/Privacy might match three different actions depending which endpoint is chosen:
	The Home controller in the Users area (if we had one).
	
	The Privacy controller in the Home area (if we had one), with the Action defaulting to Index
	The Home controller that is not in an area.
	
	However, the {controller} and {action} route parameters only match when a controller or action is found. Since there is no HomeController in the Users area, this URL does not match the first endpoint.
	
	In the second endpoint, we have included the :exists constraint on the {area} parameter. Since there is no such area as [Area(“Home”)] in the program, this endpoint is not used either.
	
	Therefore, the URL /Home/Privacy will match the final endpoint, and will be mapped to the Privacy action in the Home controller (with no area), which is correct.
	
	Run your program, and check a variety of different pages (including the Privacy page, and at least one of the Users pages) to check that they all still work, and – importantly – that they are all accessed by the correct URL.

Adding a Dedicated Conventional Route

You are going to create a route that can match to the following URL:

/Tagged/mvc

Where mvc is essentially a search parameter (not an action or controller!).

As things stand, the existing endpoints would try to interpret Tagged as either a controller or an area. This is not what we want. Not only do we not have a controller or action called Tagged, it would be also impractical to create an action for every possible tag parameter.

What we want to do is to send it to the PostController, action TagSearch and have the search string as a parameter

Define a new route. Add the following endpoint to Program.cs:

```
    app.UseAuthorization();

    app.MapControllerRoute(
        name: "tags",
        pattern: "Tagged/{tag?}",
        defaults: new { area = "Users",
        controller = "Post",
        action = "TagSearch" });

    app.MapControllerRoute(

        name: "users-area",
```

Since this is a very specific route (it’s specific to one particular action), it comes at the very start of the list of endpoints.

Very soon, we’re going to create the TagSearch action.

When we do that, we would like it to share the same view that is currently used by the PostListViewComponent. So, let’s start by moving that view to a place where it can be shared with the new action.

In the folder /Areas/Users/Views/Shared/Components/PostList, find the file Default.cshtml. Move that file to /Areas/Users/Views/Posts
Rename the file from Default.cshtml to _PostList.cshtml

Modify /Areas/Users/ViewComponents/PostListViewComponent.cs so that it can find the view in its new location:

```
        public IViewComponentResult Invoke(int? threadId)
        {
            IQueryable<Post> posts = context.Posts;
            if (threadId.HasValue)
            {
                posts = posts.Where(p => p.ThreadId == threadId);
            }
            return View("~/Areas/Users/Views/Post/_PostList.cshtml",
                PostViewModel.FromPosts(posts));

        }
```

(Note that, when specifying the full path of the view, it’s necessary to also include its extension.)

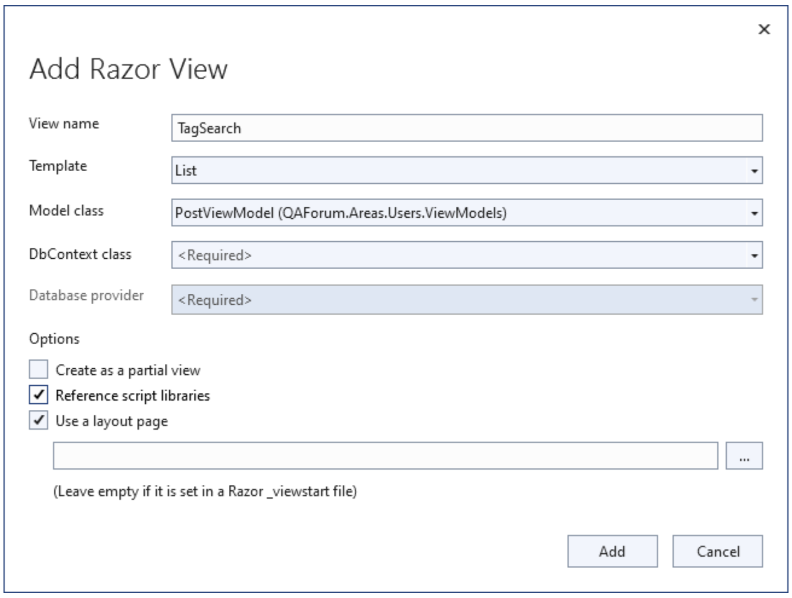
Now, we can create the TagSearch action.

In Areas/Users/Controllers/PostController.cs, add the following action to the end of the controller's code:

```
        public ActionResult TagSearch(string tag)
        {
            IEnumerable<Post> posts = new List<Post>();
            if (!string.IsNullOrEmpty(tag))
            {
                tag = tag.ToLower();
                posts = from p in context.Posts
                        where p.PostBody.ToLower().Contains(tag) ||
                            p.Title.ToLower().Contains(tag)
                        select p;
            }
            else
            {
                posts = context.Posts;
            }
            ViewBag.Message = "Posts for Tag: " + tag;
            return View(PostViewModel.FromPosts(posts));
        }
```

The code evaluates the incoming **tag** parameter. If it is Null or Empty it gets the full list of posts. If the tag parameter contains a value, it searches for posts (content or title) that contain the tag keyword instead.

Add a view to display the results:
Right click inside the **TagSearch** action and choose **Add View**
Name the new view **TagSearch**
Choose **List** for the **Template**
Select **PostViewModel** for the **Model class**


Add Razor View

View name: TagSearch
Template: List
Model class: PostViewModel (QAForum.Areas.Users.ViewModels)
DbContext class: <Required>
Database provider: <Required>

Options
- [ ] Create as a partial view
- [x] Reference script libraries
- [x] Use a layout page

(Leave empty if it is set in a Razor _viewstart file)

Click **Add** to add the new view

Replace the entire contents of the generated view with the following:

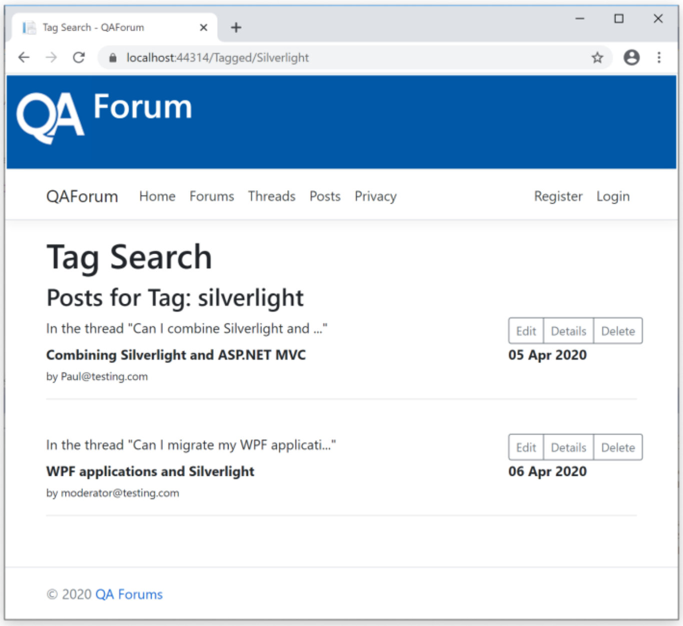
```
@model IEnumerable<QAForum.Areas.Users.ViewModels.PostViewModel>

@{
    ViewData["Title"] = "Tag Search";
}
```

```
<h1>Tag Search</h1>
<h3>@ViewBag.Message</h3>

<partial name="_PostList" />
```

Test your program. Start the program, and when it starts, change the URL to each of the following:
/Tagged
/Tagged/Mvc
/Tagged/Silverlight
The first should display all the posts. The others should display only those posts which match the tags.



Tag Search - QAForum
localhost:44314/Tagged/Silverlight
QA Forum
QAForum Home Forums Threads Posts Privacy
Register Login
Edit Details Delete

Tag Search

Posts for Tag: silverlight

In the thread "Can I combine Silverlight and ..."
Combining Silverlight and ASP.NET MVC
by Paul@testing.com
05 Apr 2020

In the thread "Can I migrate my WPF applicati..."
WPF applications and Silverlight
by moderator@testing.com
06 Apr 2020

© 2020 QA Forums

Using Attribute Routing

The problem with what we've done so far is that the user needs to know about the existence of the tag search, and needs to type the correct URL. This is not very user-friendly.

In the next section, we are going to fix this by creating another action. We're going to use attribute routing to configure the URL for this new action.

Let’s first of all create a view model for the new action. Add a new class into the /Areas/Users/ViewModels folder. Call the class TagSearchViewModel:

```
    public class TagSearchViewModel
    {
        [Display(Name = "Tag to Search")]
        public string TagToSearch { get; set; }

    }
```

Now, add a new action to the Post controller:

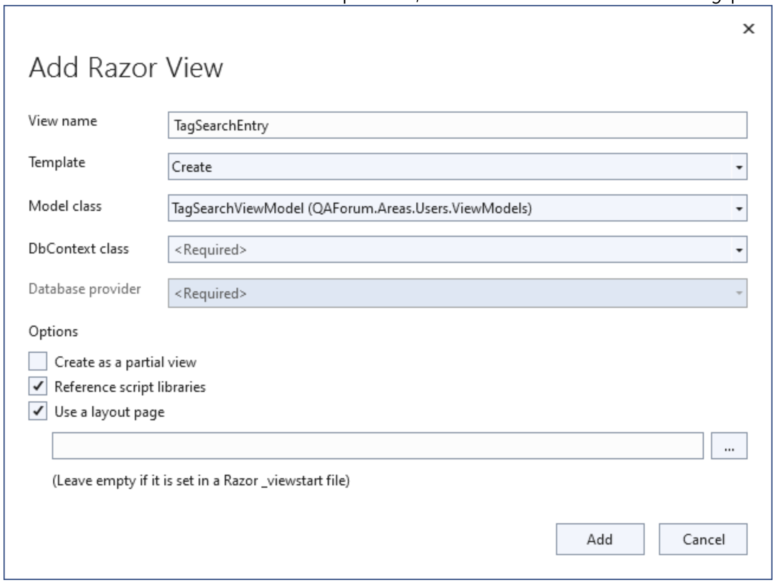
```
        [Route("TagSearch")]
        public ActionResult TagSearch()
        {
            return View("TagSearchEntry");

        }
```

Notice the [Route] attribute, which sets a route that is different to that specified in the conventions in Startup.cs

Create a view for the new action. Right-click in the action, and choose Add View. Make sure you change the name of the view to “TagSearchEntry” to match the name in the View() method in the previous step. (We had to change the name because we have two actions, both called TagSearch, which each needs their own different view.)

Choose the Create template, and set the model type to TagSearchViewModel:



Add Razor View

View name	TagSearchEntry
Template	Create
Model class	TagSearchViewModel (QAForum.Areas.Users.ViewModels)
DbContext class	<Required>
Database provider	<Required>

Options

- [ ] Create as a partial view
- [x] Reference script libraries
- [x] Use a layout page

(Leave empty if it is set in a Razor _viewstart file)

In the view, locate the line which says `<form asp-action="TagSearchEntry">`

This line results in the form data being sent back to the server, to an action called TagSearchEntry, using HTTP Post, with the data encoded into the body of the request.

This is not what we want, and a lot of what we want to achieve can't be done with an HTML form. So we will delete the <form> tag complete.

We will also delete the `<input type="submit" />` tag, and replace it with a <button> instead. We can use the button's "onclick" event handler to carry out the exact actions that we want. Note that this requires JavaScript to be enabled in the web browser.

The relevant section of the view is as shown as below. Carefully remove the lines that are crossed out, and replace them as shown here:

```
~~<form asp-action="TagSearchEntry">~~
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="TagToSearch" class="control-label"></label>
        <input asp-for="TagToSearch" class="form-control" />
        <span asp-validation-for="TagToSearch" class="text-danger"></span>
    </div>
    <div class="form-group">
        ~~<input type="submit" value="Create" class="btn btn-primary" />~~
        <button class="btn btn-primary"
                onclick="location.href = '/Tagged/' +
                        document.getElementById('TagToSearch').value">
            Submit
        </button>
    </div>

~~</form>~~
```

While we're editing this view, let's also change the headings at the top of the view:

```
@{
    ViewData["Title"] = "TagSearchEntry";
}

<h1>Tag Search</h1>

<hr />
<div class="row">

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

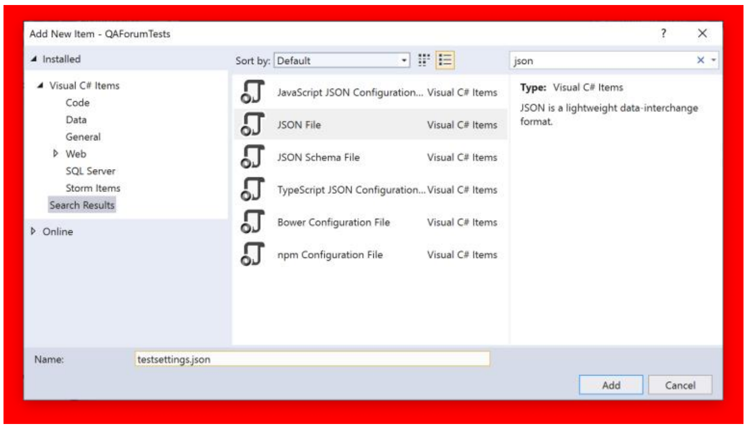
	Finally, modify /Views/Shared/_Layout.cshtml to provide a link in the navigation bar, so that users can easily find this new functionality:
	```
	<li class="nav-item">
	    <a class="nav-link text-dark" asp-area="Users" asp-controller="Post"
	                                     asp-action="Index">Posts</a>
	</li>
	<li class="nav-item">
	    <a class="nav-link text-dark" asp-area="Users" asp-controller="Post"
	                                     asp-action="TagSearch">Search Tags</a>
	</li>
	<li class="nav-item">
	    <a class="nav-link text-dark" asp-area="" asp-controller="Home"
	                                     asp-action="Privacy">Privacy</a>
	</li>
	```

	Run your application, and test your changes. Click on the Search Tags link in the navigation bar, enter a search phrase, and check that you see your search results.

If You Have Time: Testing your Routes

	Let's add some unit tests, to ensure that not only are our routes working as expected now, but also that we are quickly made aware if any future changes break them. Start by using NuGet to add the following package to your QAForumTests project: MyTested.AspNetCore.Mvc

Right-click on the test project, and select Add, then New Item.
In the search box, search for “json”. Add a new JSON file to the test project, called “testsettings.json”: You may find Visual Studio doesn't offer an option of a plain json file on which case select JavaScript JSON Configuration File.


Add New Item - QAForumTests

Installed
- Visual C# Items
  - Code
  - Data
  - General
  - Web
  - SQL Server
  - Storm Items
  - Search Results
- Online

Sort by: Default

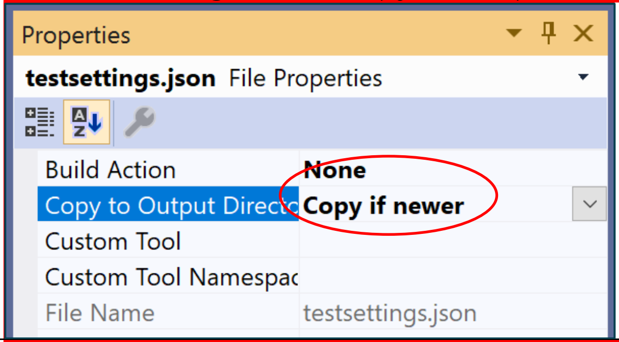
json

JavaScript JSON Configuration...	Visual C# Items
JSON File	Visual C# Items
JSON Schema File	Visual C# Items
TypeScript JSON Configuration...	Visual C# Items
Bower Configuration File	Visual C# Items
npm Configuration File	Visual C# Items

Type: Visual C# Items
JSON is a lightweight data-interchange format.

Name: testsettings.json

In Solution Explorer, click on testsettings.json. Then, in the Properties window, find the setting called “Copy to Output Directory”. Change it to “Copy if newer”:


Properties

testsettings.json File Properties

Build Action	**None**
Copy to Output Directo	**Copy if newer**
Custom Tool	
Custom Tool Namespac	
File Name	testsettings.json

Replace the contents of testsettings.json with the following:

```
{
  "ConnectionStrings": {
    "Forum.Users": "Fake",
    "Forum.Data": "Fake"
  },
  "AllowedHosts": "*"
}
```

Take a moment to compare this file to appsettings.json in the QAForum project. Note how they follow the same format, but we have replaced the real connection strings with fake ones, and removed some items that are not needed by the testing tools.

Add a new class to the test project, called RouteTests. Make sure the class is marked as public.
Into that class, paste the following test:

```
        [Fact]
        public void Home_Page_Test()
        {
            MyRouting
                .Configuration()
                .ShouldMap("/")
                .To<HomeController>(c => c.Index());
        }
```

Run the tests, and check that this new test passes.

Testing that route values get mapped to the correct parameters is very simple with MyTestedAsp. Add the following test, and check that it passes:

```
        [Fact]
        public void Forum_Details_Test()
        {
            MyRouting
                .Configuration()
                .ShouldMap("/Forum/Details/1")
                .To<ForumController>(c => c.Details(1));
        }
```

Let's add two more tests, which check some of the routing we've done during this lab:

```
        [Fact]
        public void Tagged_Test()
        {
            MyRouting
                .Configuration()
                .ShouldMap("/Tagged/TestTag")
                .To<PostController>(c => c.TagSearch("TestTag"));
        }

        [Fact]
        public void Tag_Search_Test()
        {
            MyRouting
                .Configuration()
                .ShouldMap("/TagSearch")
                .To<PostController>(c => c.TagSearch());
        }
```

Testing model binding, when data is posted to the form, requires a few more items to be configured. But the fluent syntax of MyTestedAsp still makes this relatively straightforward. So, finally, copy this test into your test clas, and check that it passes:

```
        [Fact]
        public void Forum_Create_Test()
        {
            MyRouting
                .Configuration()
                .ShouldMap(request => request
                    .WithMethod(HttpMethod.Post)
                    .WithLocation("/Forum/Create")
                    .WithFormFields(new  // N.b. anonymous object used here
                    {                    //      contains form data from browser
                        Title = "New Forum"
                    })
                    .WithAntiForgeryToken()
                ).To<ForumController>(c => c.Create(
                    new ForumWriteViewModel    // N.b. view-model used here
                    {
                        Title = "New Forum"
                    }
                ));
        }
```

Congratulations! You have now seen how to configure your application's routing, and how to test that routing.