Exercise 6 : Controllers And Actions

Objective

In this exercise you will build controllers for three key entities: Forums, Threads and Posts. You will write actions that return views for five scenarios: List, Details, Create, Edit and Delete. Finally, you will provide simple views to be returned by each of your action results.

This exercise will take around 90 minutes.

Referenced material

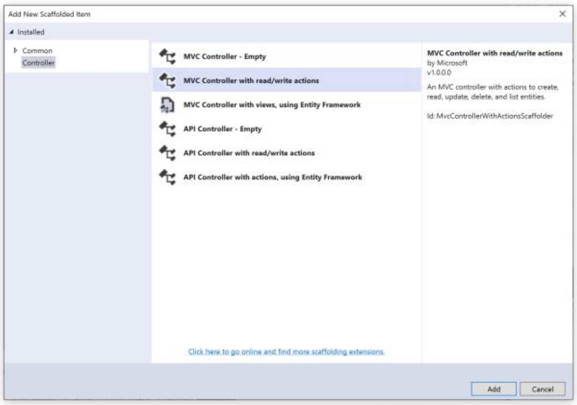
This exercise is based on material from the chapter "Controllers And Actions”.

Actions – Forum Controller

<table>
<tr><td></td><td>Open the 'Begin' solution in Visual Studio and compile (Shift Ctrl+B).</td></tr>
<tr><td></td><td>Delete the existing ForumController in the Areas/Users/Controllers folder</td></tr>
<tr><td></td><td>

Add a new ForumController to the same folder. This time, we are going to use the “Scaffolding” process to create a template, which will save us a little bit of time:

In **Solution Explorer**, right click on the Areas/Users/Controllers folder
Choose Add, Controller
Choose MVC Controller With Read/Write Actions



Add New Scaffolded Item

Installed

Common

Controller

MVC Controller - Empty

MVC Controller with read/write actions

MVC Controller with views, using Entity Framework

API Controller - Empty

API Controller with read/write actions

API Controller with actions, using Entity Framework

MVC Controller with read/write actions
by Microsoft
v1.0.0.0
An MVC controller with actions to create, read, update, delete, and list entities.

Id: MvcControllerWithActionsScaffolder

Click here to go online and find more scaffolding extensions.

Click Add
Enter the name, ForumController
Click Add to add the new controller

</td></tr>
<tr><td></td><td>Take a moment to look through the resulting file: ForumController.cs
Several read actions have been added to the class including **Index** and **Details**.
There are also several write actions.</td></tr>
<tr><td></td><td>

Re-add the code that was in the previous ForumController:

```
    [Area("Users")]
    public class ForumController : Controller
    {
        private readonly ForumDbContext context;

        public ForumController(ForumDbContext context)
        {
            this.context = context;
        }

        // GET: Forum
        public ActionResult Index()
        {
            return View(ForumViewModel.FromForums(context.Forums));
        }
```

</td></tr>
</table>

Right-click on the Index action, and choose Go To View. This will open the view that we created in a previous lab.

Scroll to the bottom of the view. You will find three calls to @Html.ActionLink – one for Edit, one for Details, and the third one for Delete.

At the moment, each of these ActionLink calls has its RouteValues parameter set to an empty anonymous object, but a comment indicates that we should amend this empty object to include the primary key from the item.

Let's do that:

```
        @Html.ActionLink("Edit", "Edit", new { id = item.ForumId }) |
        @Html.ActionLink("Details", "Details", new { id = item.ForumId }) |
        @Html.ActionLink("Delete", "Delete", new { id = item.ForumId })
```

(When we created the ForumViewModel class, did you wonder why we included the ForumId property, and then configured it to not be shown?

It seemed a bit counter-intuitive to include it if we weren't intending to show it. But now it becomes clear – we included it because the view needs it, not to display to the user but to be able to link to the next action.)

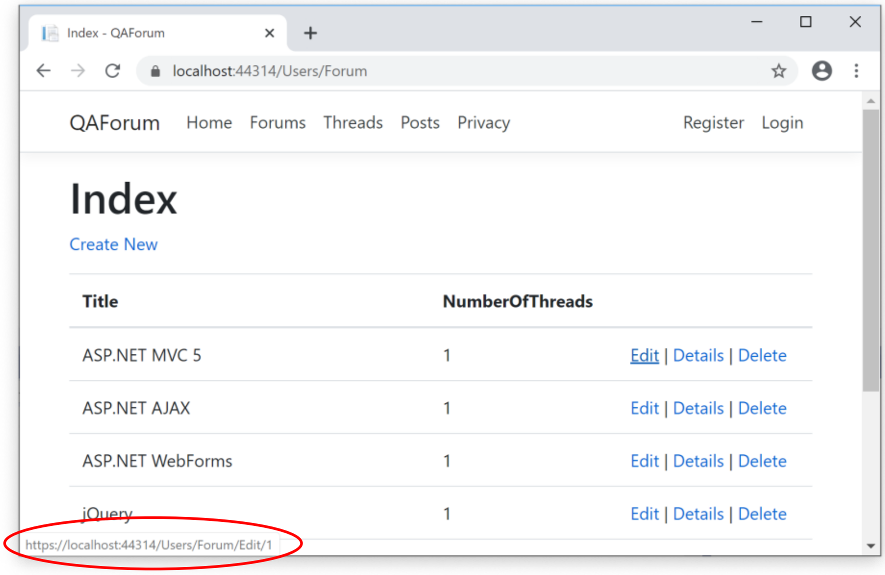
It's worth pointing out here that, if you choose a model type for your view that's part of a DbContext, the scaffolder will automatically do this step for you. It's only because of our choice to use view-models which are not stored in entity framework that we need to do this step manually.

Also, at the top of the file, change the text in the <h1>:

```
<h1>Welcome to the Forums!</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
```

Press F5, and when your program starts, go to the Forums page.
Hover the mouse over the Edit, Details and Delete links next to each forum. Depending on the web browser that you are using, you should see the URL that the link is going to take you to – often in the bottom left corner:



Index

Title	NumberOfThreads	
ASP.NET MVC 5	1	Edit | Details | Delete
ASP.NET AJAX	1	Edit | Details | Delete
ASP.NET WebForms	1	Edit | Details | Delete
jQuery	1	Edit | Details | Delete

https://localhost:44314/Users/Forum/Edit/1

As you move between links for each of the different forums, notice how the final part of the URL changes to match the id of each forum.

(There is no point clicking on these links yet – since we haven't written the code for them, they won't work.)

Stop the program.

Let's build a Details action next. That shows the details of each forum.

For some data entities, there will be many properties – too many to include them all in a list view, but we might want a lot more included in a details view than the list view. If this were the case, we'd need to make a new view-model that included the relevant properties.

But the Forum entity is very simple, and the list view already includes all the details we need.

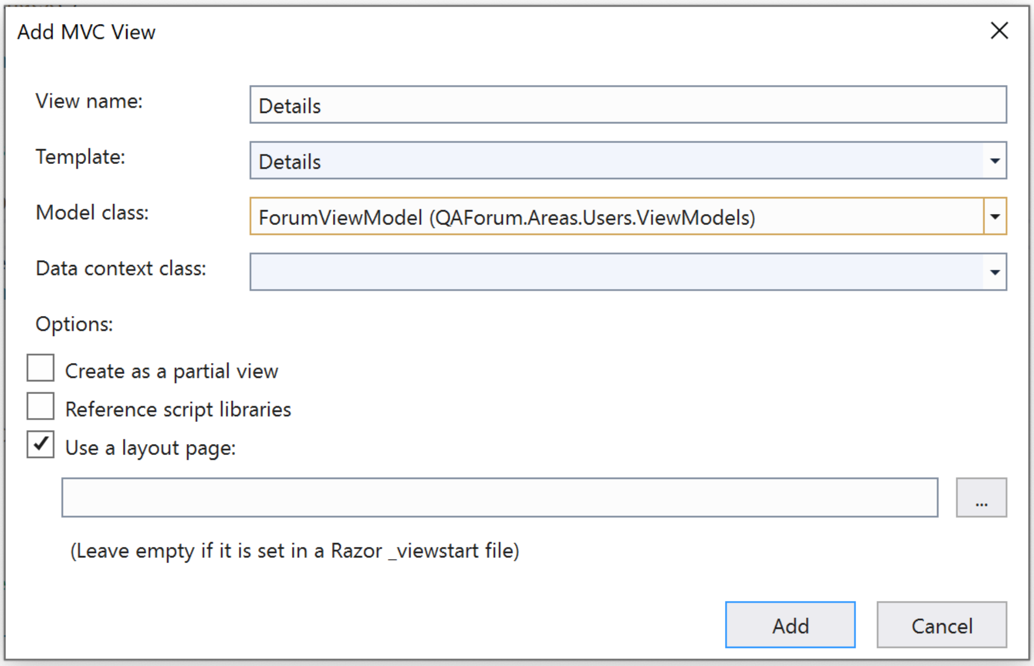
Therefore, we can use the existing view-model. (There is a school of thought that every view should have its own view-model. This is a valid argument – sharing view-models might make it harder to maintain the program if one view were to change its required data. But for these labs we're going to share view-models where the underlying data for two views is the same.)

(You might ask what's the point of the details view if it only shows the same details as the list view, and that's a very valid question! We will add more to the details view in a later lab, to make it more useful.)

Amend the Details method in the Forum controller as follows:

```
        public ActionResult Details(int id)
        {
            var forum = context.Forums.Single(f => f.ForumId == id);
            return View(ForumViewModel.FromForum(forum));
        }
```

Then, right-click on the Details action, and select Add View. Change the template to Details, and change the model class to ForumViewModel:



Add MVC View

View name: Details

Template: Details

Model class: ForumViewModel (QAForum.Areas.Users.ViewModels)

Data context class:

Options:

□ Create as a partial view

□ Reference script libraries

☑ Use a layout page:

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

Click Add

A new file, Details.cshtml, is created.

At the bottom of the page is a call the Html.ActionLink, which requires you to add the id in (the same way we did in the Index view):

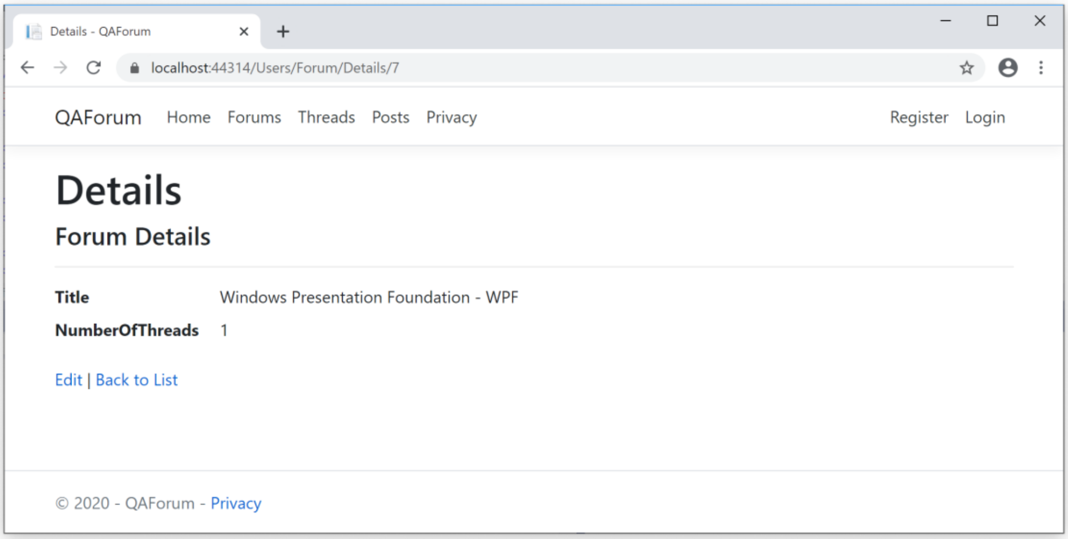
```
<div>
    @Html.ActionLink("Edit", "Edit", new { id = Model.ForumId }) |
    <a asp-action="Index">Back to List</a>
</div>
```

Also, modify the <h4> element towards the top of the page:

```
<h1>Details</h1>

<div>
    <h4>Forum</h4>
    <hr />
```

Run your program. Go to the Forum page, then click the Details link next to one of the forums. Check it shows the details of that forum.



Details - QAForum
localhost:44314/Users/Forum/Details/7
QAForum Home Forums Threads Posts Privacy
Register Login

Details

Forum Details

Title	Windows Presentation Foundation - WPF
NumberOfThreads	1

Edit | Back to List

© 2020 - QAForum - Privacy

When you're satisfied that this code works, stop the program.

Next, we'll work on creating a new forum.

Once again, we start by considering the view-model that we want to use. The existing view-model is not suitable, because it contains a property called NumberOfThreads. Users will not be expected to provide this value when they add a forum, nor when they edit it.

What's more, the ForumId is not going to be entered by the user. It will be automatically assigned by the database. So this should not be part of the view-model for create a forum.

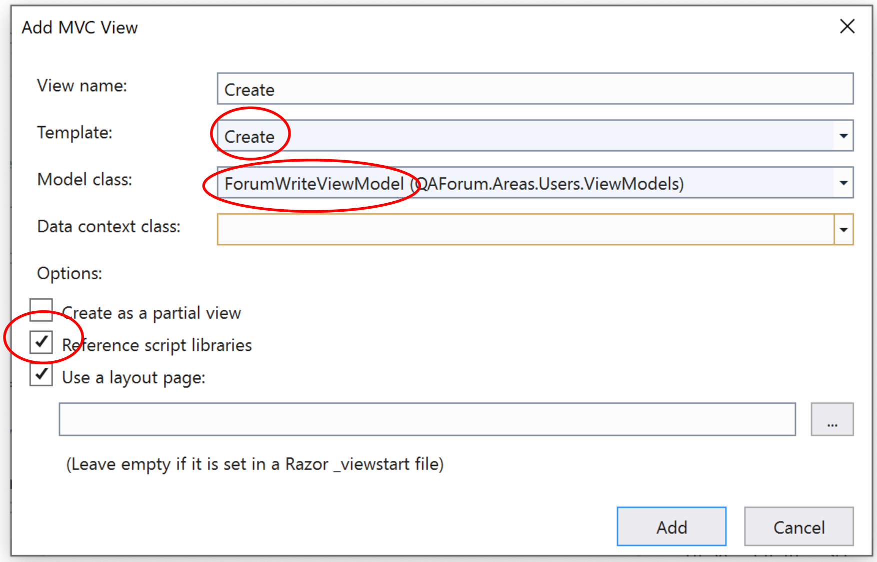
Let's create a new view-model, in the ViewModels folder:

```
    public class ForumWriteViewModel
    {
        public string Title { get; set; }

        public static ForumWriteViewModel FromForum(Forum forum)
        {
            return new ForumWriteViewModel
            {
                Title = forum.Title,
            };
        }
    }
```

When the user clicks the Create link at the top of the Forum page, it will send a GET request to the server. In response, the server needs to send a blank page to the web browser.

To do this, right-click in the Create method (either of the overloads will work fine), and select Add View. Use the Create template, with the ForumWriteViewModel as the model. Make sure that Reference Script Libraries is checked, too. (We will need this to be checked for a later lab – you should ensure it's checked every time you add a Create or Edit view, at least for now.)



Add MVC View

View name: Create

Template: Create

Model class: ForumWriteViewModel (QAForum.Areas.Users.ViewModels)

Data context class:

Options:

□ Create as a partial view

☑ Reference script libraries

☑ Use a layout page:

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

Click Add, and the view will be created. Change the contents of the <h4> at the top:

```
<h1>Create</h1>

<h4>Forum</h4>
```

In the view, you will see this line:

```
        <form asp-action="Create">
```

Although it’s not obvious just from looking at it, the HTML which is generated by this includes an action of POST:

```
        <form action="/Users/Forum/Create" method="post">
```

So, when the user clicks the Create button, the data they entered will be handled by the second overload of Create in the controller, the one which is decorated with the [HttpPost] attribute.

Change the parameter of that method to be a ForumWriteViewModel, and then, under the relevant comment, add code to create a Forum and add it to the database:

```
        public ActionResult Create(ForumWriteViewModel viewModel)
        {
            try
            {
                // TODO: Add insert logic here
                Forum forum = new Forum
                {
                    Title = viewModel.Title
                };
                context.Forums.Add(forum);
                context.SaveChanges();

                return RedirectToAction(nameof(Index));
            }
            catch
            {
                return View(viewModel);
            }
        }
```

Note that, after successfully adding the new forum to the database, we use the RedirectToAction helper method to generate a RedirectResult. This will send a response to the web browser (with response code 302, meaning Found) telling it that the next page it needs is the Index action. The browser will then make a new request, asking for the Index action from the server.

Test your work, and make sure you can add forums.

Editing data is similar. And our ForumWriteViewModel is perfect for the job, since the data the user enters when creating a forum is the same as the data they enter when editing a forum. (The Forum Id is needed in order to update a forum, but this is passed from one view to the next through the URL, and does not need to be part of the view-model.)

In the controller, when the user chooses to edit a forum, we need to retrieve the current details of that forum from the database and present them to the user. This is different to creating a forum. So, the first Edit action needs to include this extra step:

```
        public ActionResult Edit(int id)
        {
            var forum = context.Forums.Single(f => f.ForumId == id);
            return View(ForumWriteViewModel.FromForum(forum));
        }
```

Create a view for editing forums. Use the Edit template, and set the model class to ForumWriteViewModel. Once again, change the contents of the <h4> to simply the word “Forum”.

Finally, add code to the HttpPost version of the Edit method to edit the data and save it back to the database. Don't forget to change the parameters!

```
        public ActionResult Edit(int id, ForumWriteViewModel viewModel)
        {
            try
            {
                // TODO: Add update logic here
                var forum = context.Forums.Single(f => f.ForumId == id);
                forum.Title = viewModel.Title;
                context.SaveChanges();

                return RedirectToAction(nameof(Index));
            }
            catch
            {
                return View(viewModel);
            }
        }
```

Then run the program, and test that you can edit the name of the forums

The next step is to get the Delete action to work.
When the user clicks the Delete button, they will see a confirmation screen, which is quite similar to the Details screen. Because of this, we can use the same view-model as was used for the Details action

Edit the Get version of the Delete method to retrieve the data:

```
        public ActionResult Delete(int id)
        {
            var forum = context.Forums.Single(f => f.ForumId == id);
            return View(ForumViewModel.FromForum(forum));
        }
```

Add a view. Use the Delete template, and set the model to ForumViewModel. As before, change the contents of the <h4> at the top of the page.

Write the code to delete a forum in the [HttpPost] Delete method:

```
        public ActionResult Delete(int id, IFormCollection collection)
        {
            var forum = context.Forums.Single(f => f.ForumId == id);
            try
            {
                // TODO: Add delete logic here
                context.Forums.Remove(forum);
                context.SaveChanges();

                return RedirectToAction(nameof(Index));
            }
            catch
            {
                return View(ForumViewModel.FromForum(forum));
            }
        }
```

Note that, for this method, we leave the IFormCollection parameter in place. We don't need to use it at all – the only data we need is the id. But the Get method also needs an id, so we have to leave the second parameter in place to ensure that the two methods are sufficiently different to meet the C# rules about method overloading.

Test the delete method by deleting the forum you created earlier in the lab. It should delete without any problems.

But now, try to delete one of the pre-existing forums. What happens? Do you know why?

Read on, and we'll explain it

When you try to delete one of the existing forums, an exception occurs. We don't know what the details of that exception are (yet), but this exception results in control going into the "catch" block in the Delete action.

And that catch block… well, it simply re-displays the same view again, with no indication to the user that anything has gone wrong (apart from the fact that the button didn't seem to do anything).

Let's see if we can improve things. Change the catch block in the Delete action as follows:

```
            catch (Exception e)
            {
                ViewBag.ErrorText = e.GetBaseException().Message;
                return View(ForumViewModel.FromForum(forum));
            }
```

Right-click on the Delete action, and select Go To View. Modify the view by adding a line to show the error message, as shown below:

```
<h1>Delete</h1>

<p class="text-danger">@ViewBag.ErrorText</p>

<h3>Are you sure you want to delete this?</h3>
```

Now, try to delete one of the pre-existing forums again.

You should now see an error message! It's not a great error message – it's not very user-friendly, and it's also not good practice to show exception messages to the user in general in case they contain sensitive data. But it's enough for us to work out what's going on. It shows that there's a foreign key violation – we can't delete this forum, because the forum contains threads. We'd need to delete the threads it contains first (and also delete any posts in those threads).

(This behaviour was configured in the ForumDbContext class, in the OnModelCreating method. The rules got copied into the initial migration we created in the previous chapter, and then applied to the database. It is now SQL Server's responsibility to ensure that these rules are not violated.)

We'll leave you to think about how this error message could be improved. But for now, it's enough that we can at least see the error message

Apply the same changes to the Create and Edit actions and their corresponding views, so that any errors in those actions also get shown to the user.

Actions – Thread Controller

Now, we'll apply similar changes to the Thread controller.
As you work through this section, any time you add the Thread class to your code and then use Ctrl-. to add the relevant "using" section, take care to add the correct namespace. You need the QAForum.Models namespace, *not* the System.Threading namespace!

Start by:
Deleting the existing Thread controller
Adding a new Thread controller, using the appropriate scaffolding option
Adding back the injected DbContext, and the [Area] attribute, and also the code in the Index action

Modifying the Index view, to ensure that the Edit/Details/Delete links include the thread id.

Test that the Thread page works, and check that, when you hover over the links to the right of each thread, you see the correct link (including the id)

Next, add the Details action
Write the code
Create a view
Edit the sub-heading at the top of the page, and the Html.ActionLink at the bottom of the page
Test that this works too

We're going to work on the Create action next. But we're going to add an extra option, so that when a user creates a thread, they can, if they want, add a welcome post to the thread.
To do this, we will need a view-model built specially for the purpose. Here it is:

```
    public class ThreadCreateViewModel
    {
        public string Title { get; set; }
        public int ForumId { get; set; }
        public string UserName { get; set; }
        public bool AddWelcomePost { get; set; }
    }
```

Create a view for the Create action.
Modify the sub-heading to something sensible. Also, add a paragraph to contain any error messages.

Here is the code for the [HttpPost] action. Notice how easy it is to process the extra option that we've added to our view-model:

```
        public ActionResult Create(ThreadCreateViewModel viewModel)
        {
            try
            {
                // TODO: Add insert logic here
                Thread thread = new Thread
                {
                    Title = viewModel.Title,
                    ForumId = viewModel.ForumId,
                    UserName = viewModel.UserName
                };
                if (viewModel.AddWelcomePost)
                {
                    thread.Posts = new List<Post>
                    {
                        new Post
                        {
                            UserName = viewModel.UserName,
                            Title = "Welcome",
                            PostDateTime = DateTime.Now,
                            PostBody = "Welcome to this thread. We hope it 
provides you with useful information",
                            Thread = thread
                        }
                    };
                }
                context.Threads.Add(thread);
                context.SaveChanges();

                return RedirectToAction(nameof(Index));
            }
            catch (Exception e)
            {
                ViewBag.ErrorText = e.GetBaseException().Message;
                return View(viewModel);
            }
```

Try this now. Create two new threads. For the first one, leave the tick-box for adding a welcome message un-checked. For the second one, check the tick-box. Then, go to the Posts view to see that a post has been created to go with your second new thread.

Note that, when you add a thread, you have to enter a Forum Id. Make sure you enter a valid number (anything between 1 and 7 should be valid), otherwise you'll get an error. Surely it must be possible to show a list of forums, instead of expecting the user to know the forum id? Yes, it is, and we'll investigate how to do that in a later chapter.

Next, create a ThreadEditViewModel. This is similar to the ThreadCreateViewModel, but doesn't make use of the boolean flag – that was only needed for creating. Also, we need to be able to create this view-model from the underlying model:

```
    public class ThreadEditViewModel
    {
        public string Title { get; set; }
        public int ForumId { get; set; }
        public string UserName { get; set; }

        public static ThreadEditViewModel FromThread(Thread thread)
        {
            return new ThreadEditViewModel
            {
                Title = thread.Title,
                ForumId = thread.ForumId,
                UserName = thread.UserName
            };
        }
    }
```

Try to create the Edit page by yourself, using the above view-model. Look at the model answer if you need some guidance.
Test your code to make sure it works

Now try to create the Delete page by yourself. Use the ThreadViewModel as the view-model. Once you're done, test your work again.

If you have time – Post Controller

Have a go at writing the Post Controller by yourself.
When you create your view-models, simply use the same data as is in the Post model class.
You will need two view-models:
PostViewModel
This should already be in your project – we added it in the previous lab, and we already use it for the List view
This will be used by the List, Details and Delete actions

PostWriteViewModel
Does not need to include PostId, but otherwise will contain similar properties as PostViewModel
However, the Thread property (of type string) will need to be replaced by a ThreadId property (of type int)
This will be used by the Create and Edit actions
Test your work regularly as you go. Don't forget to update Html.ActionLink route values, and also don't forget to change the page headings where the automatically generated headings are not appropriate.

If you really have time – Improving to the Post Controller

	Make a duplicate of PostViewModel – call it PostDetailsViewModel
	Go to the original PostViewModel class, and remove the PostBody property
	There’s no need to include the post body on the list of posts. Re-create the Index view to use the cut-down PostViewModel class.
	Re-create the Details view to use the new PostDetailsViewModel, so that it still shows all of the post data.
	You will also need to re-create the Delete view. It doesn’t matter whether you choose to base this on the PostViewModel (which no longer includes the post body), or the PostDetailsViewModel (if you want the post body) – use whichever you prefer.