

Exercise 8a : Views

Objective

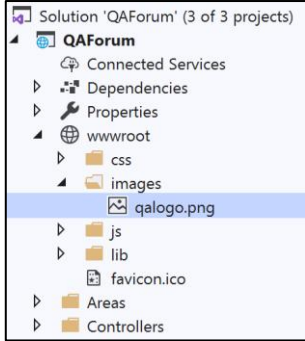
In this exercise, you will use a number of techniques to improve the views that were scaffolded for us, including adding drop-down boxes and images, and making use of partial views for reusability.

This exercise will take around 90 minutes.

Referenced material

This exercise is based on material from the chapter "Views".

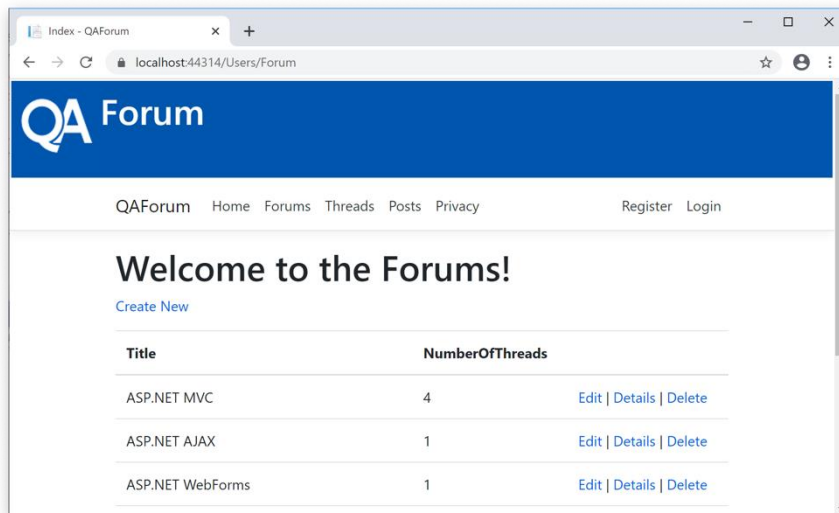
Changing the Layout

| | |
|--|--|
| | <p>Open the 'Begin' solution in Visual Studio and compile (Shift Ctrl+B). This project is identical to the End from the previous lab</p> |
| | <p>Add the QA logo to the websites content folder: Locate the folder 'images' in the Assets folder and drag it onto your wwwroot folder.</p>  <p>Check that qalogo.png is included in the project</p> |
| | <p>Add styles for page-header by adding this to the bottom of wwwroot/css/site.css</p> <pre> /* QAForum styles */ .page-header { padding-bottom: 9px; border-bottom: 1px solid #eee; margin: 2px 2px; background-color: rgb(0, 88, 167); color: rgb(254, 254, 254); } .page-header h1 { padding-top: 9px; } </pre> |
| | <p>In _Layout.cshtml, immediately below the <header> tag, add a banner:</p> <pre> <body> <header> <div class="page-header"> <div class="clearfix"> <h1> Forum</h1> </div> </div> <nav class="navbar navbar-expand-sm.... </pre> |

Build and run the app.

Note that many web browsers will cache CSS, so you'll need to force your browser to refresh the cache. In most browsers, Ctrl-F5 will do this.

Check out the new look! Note that it's been applied to all of the pages on the website, because they all share the same layout page.



Adding Drop-Downs

As we identified in a previous lab, the user experience when adding a thread or a post is not great at the moment. The user has to enter a forum id (for a thread) or a thread id (for a post), and there's no way we could expect a real use to know the appropriate id.

Let's change things so that the user gets presented with a drop-down box instead

The views will need a collection of selection list items. So, add the following class to the ViewModels folder. The class contains extension functions which get the required data from the ForumDbContext:

```
public static class SelectListHelpers
{
    public static IEnumerable<SelectListItem> GetForumsSelectListItems
        (this ForumDbContext context, int? selectedForumId = null)
    {
        return context.Forums.Select(f => new SelectListItem(
            f.Title,
            f.ForumId.ToString(),
            f.ForumId == selectedForumId
        ));
    }

    public static IEnumerable<SelectListItem> GetThreadsSelectListItems
        (this ForumDbContext context, int? selectedThreadId = null)
    {
        return context.Threads.Select(t => new SelectListItem(
            t.Title,
            t.ThreadId.ToString(),
            t.ThreadId == selectedThreadId
        ));
    }
}
```

| | |
|--|--|
| | <pre> } </pre> |
| | <p>Now, amend the following view-models: ThreadCreateViewModel ThreadEditViewModel</p> <p>In each case, add the following property:</p> <pre> public IEnumerable<SelectListItem> ForumSelectListItems { get; set; } </pre> <p>Add a similar property to PostWriteViewModel (notice that it has a different name):</p> <pre> public IEnumerable<SelectListItem> ThreadSelectListItems { get; set; } </pre> |
| | <p>Each of the three view-models with the new properties now needs to be amended so that the new property is initialised. Add code to each one as shown here:</p> <p>ThreadCreateViewModel</p> <pre> public static ThreadCreateViewModel WithForumSelectListItems (ForumDbContext context) { return new ThreadCreateViewModel { ForumSelectListItems = context.GetForumsSelectListItems() }; } </pre> <p>ThreadEditViewModel</p> <pre> public static ThreadEditViewModel FromThread (Models.Thread thread, ForumDbContext context) { return new ThreadEditViewModel { Title = thread.Title, ForumId = thread.ForumId, UserName = thread.UserName, ForumSelectListItems = context.GetForumsSelectListItems(thread.ForumId) }; } </pre> <p>PostWriteViewModel</p> <pre> public static PostWriteViewModel FromPost (Post post, ForumDbContext context) { return new PostWriteViewModel { ThreadId = post.ThreadId, Title = post.Title, UserName = post.UserName, PostBody = post.PostBody, PostDateTime = post.PostDateTime, ThreadSelectListItems = context.GetThreadsSelectListItems(post.ThreadId) }; } public static PostWriteViewModel WithThreadSelectListItems (ForumDbContext context) { return new PostWriteViewModel { ThreadSelectListItems = context.GetThreadsSelectListItems() }; } </pre> |

Modify the controllers to work with these redesigned view-models. The Create methods in both the thread and post controllers will now need to create an empty view-model to pass to the view – empty, that is, except for the collection of select list items. The Edit methods already create view-models to pass to the view, but the view-models now need a reference to the context in order to fetch the required data.

ThreadController

```
public ActionResult Create()
{
    return View
        (ThreadCreateViewModel.WithForumSelectListItems(context));
}

public ActionResult Edit(int id)
{
    var thread = context.Threads.Single(t => t.ThreadId == id);
    return View(ThreadEditViewModel.FromThread(thread, context));
}
```

PostController

```
public ActionResult Create()
{
    return View(PostWriteViewModel.WithThreadSelectListItems(context));
}

public ActionResult Edit(int id)
{
    Post post = context.Posts.Single(p => p.PostId == id);
    return View(PostWriteViewModel.FromPost(post, context));
}
```

Now, the views have all the information they need to display the drop-downs. Modify them as follows, in each case removing the <input> that gets the id from the user with a <select> instead:

Views/Thread/Create

```
<input asp-for="ForumId" class="form-control" />
<select asp-for="ForumId" asp-items="@Model.ForumSelectListItems"
    class="form-control"></select>
```

Views/Thread/Edit

```
<input asp-for="ForumId" class="form-control" />
<select asp-for="ForumId" asp-items="@Model.ForumSelectListItems"
    class="form-control"></select>
```

Views/Post/Create

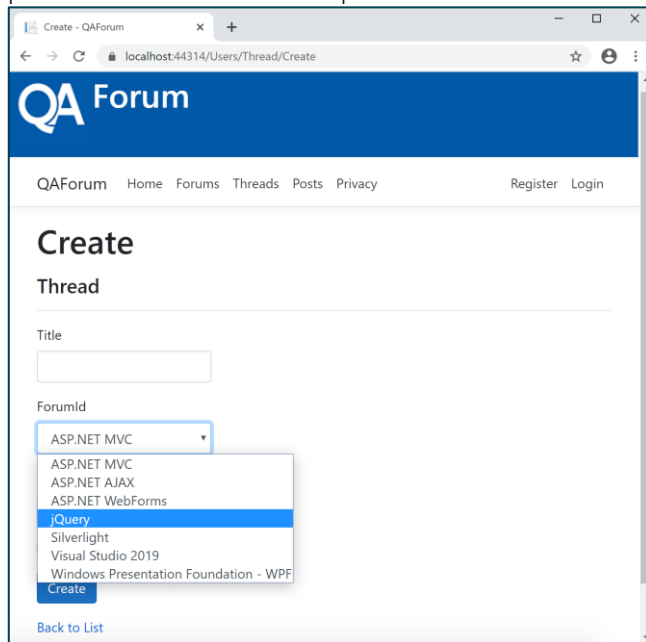
```
<input asp-for="ThreadId" class="form-control" />
<select asp-for="ThreadId" asp-items="@Model.ThreadSelectListItems"
    class="form-control"></select>
```

Views/Post/Edit

```
<input asp-for="ThreadId" class="form-control" />
<select asp-for="ThreadId" asp-items="@Model.ThreadSelectListItems"
    class="form-control"></select>
```

Test your changes.

You should now be able to create or edit threads and posts, and to choose the parent item from a drop-down instead of having to enter an id!



However, there is a problem. Create a post, and enter a date of 01/01/0001. That date is outside the range that is allowed by the database, so an exception will be thrown.

And when the exception is thrown, the view-model that is used to re-create the view doesn't contain the select list items.

1

Make the following changes to the controllers:

ThreadController (in both the Create and Edit HttpPost methods):

```
catch (Exception e)
{
    ViewBag.ErrorMessage = e.GetBaseException().Message;
    viewModel.ForumSelectListItems = context.GetForumsSelectListItems();
    return View(viewModel);
}
```

PostController (in both the Create and Edit HttpPost methods):

```
catch (Exception e)
{
    ViewBag.ErrorMessage = e.GetBaseException().Message;
    viewModel.ThreadSelectListItems = context.GetThreadsSelectListItems();
    return View(viewModel);
}
```

Now run the same test again (with the invalid date). Test both the Create and Edit views. That's better – the dropdown list is still there even after an exception!

Changing Column Names

| | |
|--|--|
| | <p>On the Forums page is a column called "NumberOfThreads". Surely that column name should have spaces in it?</p> <p>But the column name comes from the name of the property in the view-model, and property names can't have spaces.</p> <p>We can get around this problem by using a [Display] attribute on the view-model. Add this line to the ForumViewModel class:</p> <pre>[Display(Name = "Number of Threads")] public int NumberOfThreads { get; set; }</pre> <p>Run the program again, and take another look at the Forums page. That looks better!</p> <p>Spend a minute or two going through each of the view-models, adding [Display] attributes where appropriate. You can use them to add spaces, but you can also change the display names to something more human-friendly or more appropriate to the context of the view if you wish.</p> <p>For example, when adding or editing a thread, the view-model contains a property called ForumId, and although that column does indeed contain the forum id, the id is not shown to the user. Instead, the user sees the dropdown containing the name of the forum. So you might want to change the display name to "Forum".</p> |
|--|--|

Partial Views

| | |
|--|---|
| | <p>The application currently has the ability to show a list of threads, on the Threads/Index page.</p> <p>We are now going to add to ability to see a similar list of threads on the Forum/Details page, so that we see the threads in a forum when we look at the forum's details.</p> <p>So, the displaying of a list of threads will be required in two different places. That's a job for a partial view!</p> |
|--|---|

Add a partial view:

Open the file **Areas\Users\Controllers\ThreadController.cs**

Right click anywhere inside the **Index** method and choose **Add View**

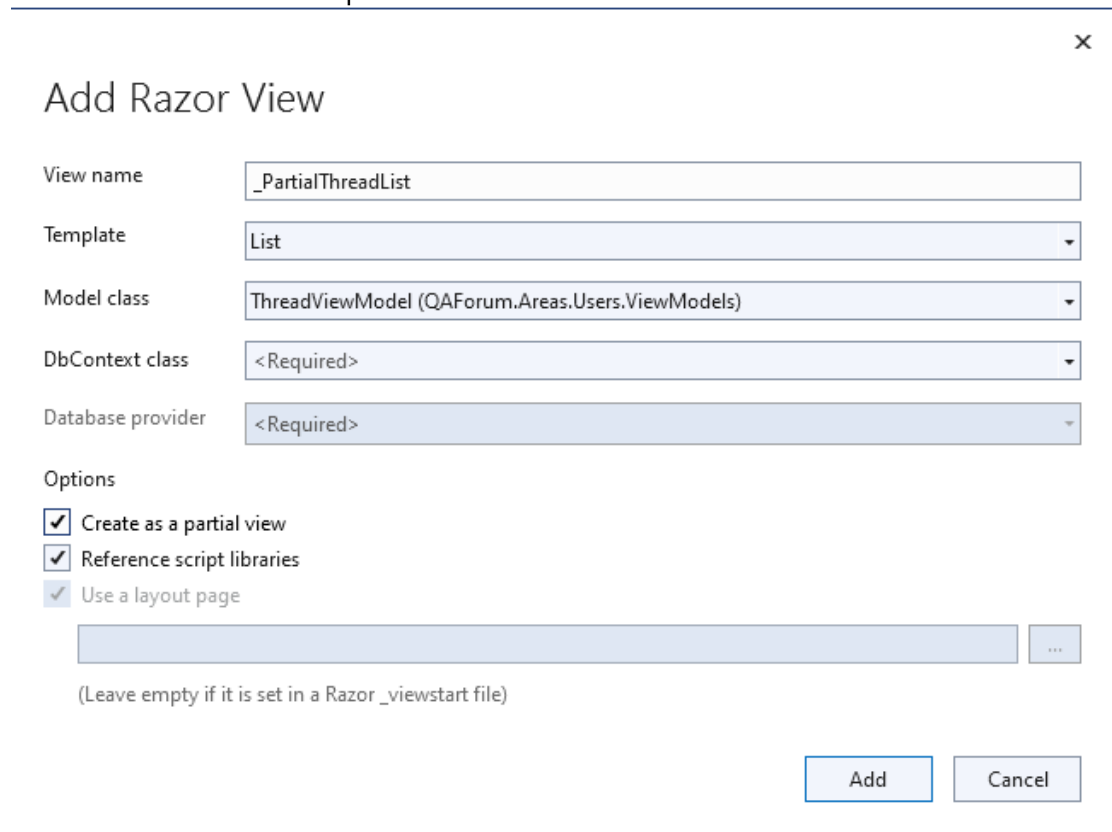
Name the view **_PartialThreadList**

Note: there is a common convention that any .cshtml file that is not a full page is named with a leading underscore.

Choose **List** for the **Template** type

Select the **ThreadViewModel** for the **Model** class

Check the "Create as a partial view" checkbox



Click **Add** to create the partial view

Fix that ActionLinks at the bottom of the new partial view:

```
<td>
    @Html.ActionLink("Edit", "Edit", new { id=item.ThreadId }) |
    @Html.ActionLink("Details", "Details", new { id=item.ThreadId }) |
    @Html.ActionLink("Delete", "Delete", new { id=item.ThreadId })
</td>
```

Modify the Thread Index view to use the partial view:

Open the file **Views\Thread\Index.cshtml** in the Users area

Go to just after the **<h1>** element and remove (or comment out) everything from here to the end of the file

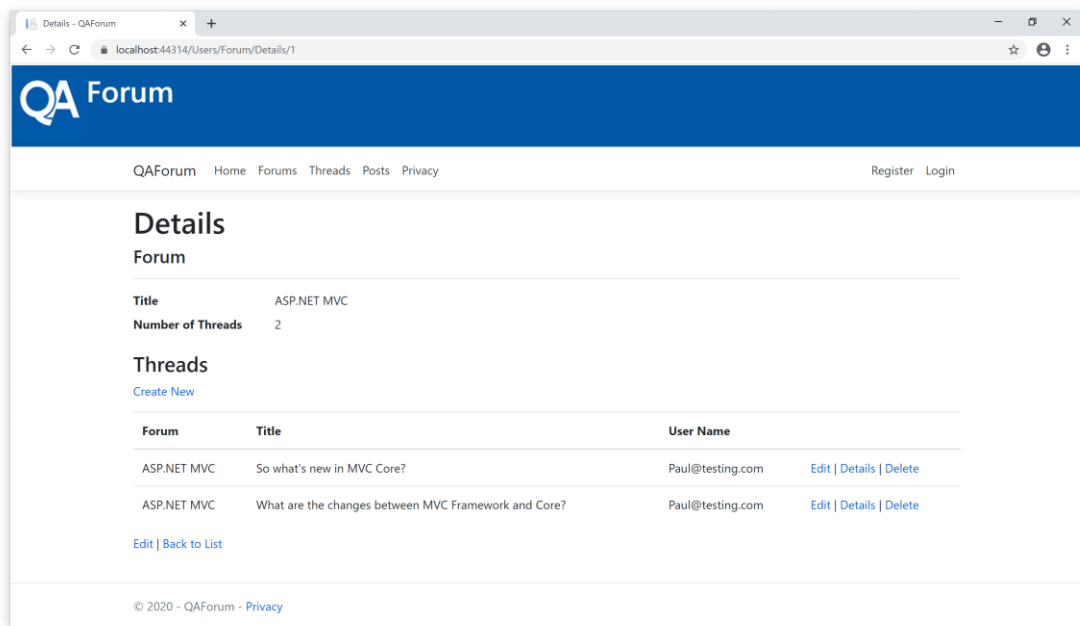
Add in our new PartialView:

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

Run your program to test your changes. Go to the Threads view, and confirm you can still see the list of threads, and that all the links work

| | |
|--|---|
| | <p>Add a Threads list to the ForumViewModel:</p> <pre> public class ForumViewModel { public int NumberOfThreads { get; set; } public IEnumerable<ThreadViewModel> Threads { get; set; } public static ForumViewModel FromForum(Forum forum) { return new ForumViewModel { NumberOfThreads = forum.Threads.Count, Threads = ThreadViewModel.FromThreads(forum.Threads) }; } } </pre> |
| | <p>Modify the forum details view to display the threads: There are two <div> tags on the page. Add this markup in between the two divs:</p> <pre> </dl> </div> <div> <h3>Threads</h3> <partial name="~/Areas/Users/Views/Thread/_PartialThreadList.cshtml" model="@Model.Threads" /> </div> <div> @Html.ActionLink("Edit", "Edit", new { id = Model.ForumId }) </pre> <p>Since the partial view is in a different folder the code has to use the full path in the <partial> tag helper, specifying the extension of the view as well its path. For the partial view to render it needs a model to be passed, of type IEnumerable<Threads>. We added it to the view-model for this purpose.</p> |
| | <p>Test the view, by going to the Forums page, then clicking the Details link next to any forum (make sure you choose a forum which has at least one thread though!)</p> |

At first, everything looks good!



But hover over the Details link next to one of the threads, and look at the URL it's taking you to.

You will notice that the hyperlink goes to /Users/**Forum**/Details/x – this is not right. It should be /Users/**Thread**/Details/x.

The reason it's done this is because the link does not specify the controller, and by default it links to an action in the same controller as the current action. Since the page we are viewing is a Forum page, it tries to link us to another forum page

To fix this, we need to specify the controller explicitly. Modify `_PartialThreadList.cshtml`:

```
<td>
    @Html.ActionLink("Edit", "Edit", "Thread", new { id=item.ThreadId }) |
    @Html.ActionLink("Details", "Details", "Thread", new { id=item.ThreadId }) |
    @Html.ActionLink("Delete", "Delete", "Thread", new { id=item.ThreadId })
</td>
```

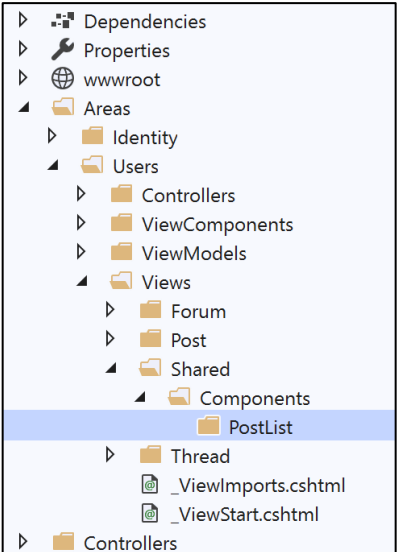
Test this, and check that all the links now work.

View Components

Now, we're going to do something similar with posts, so that when you look at the details view for a thread you can see the posts in that thread.

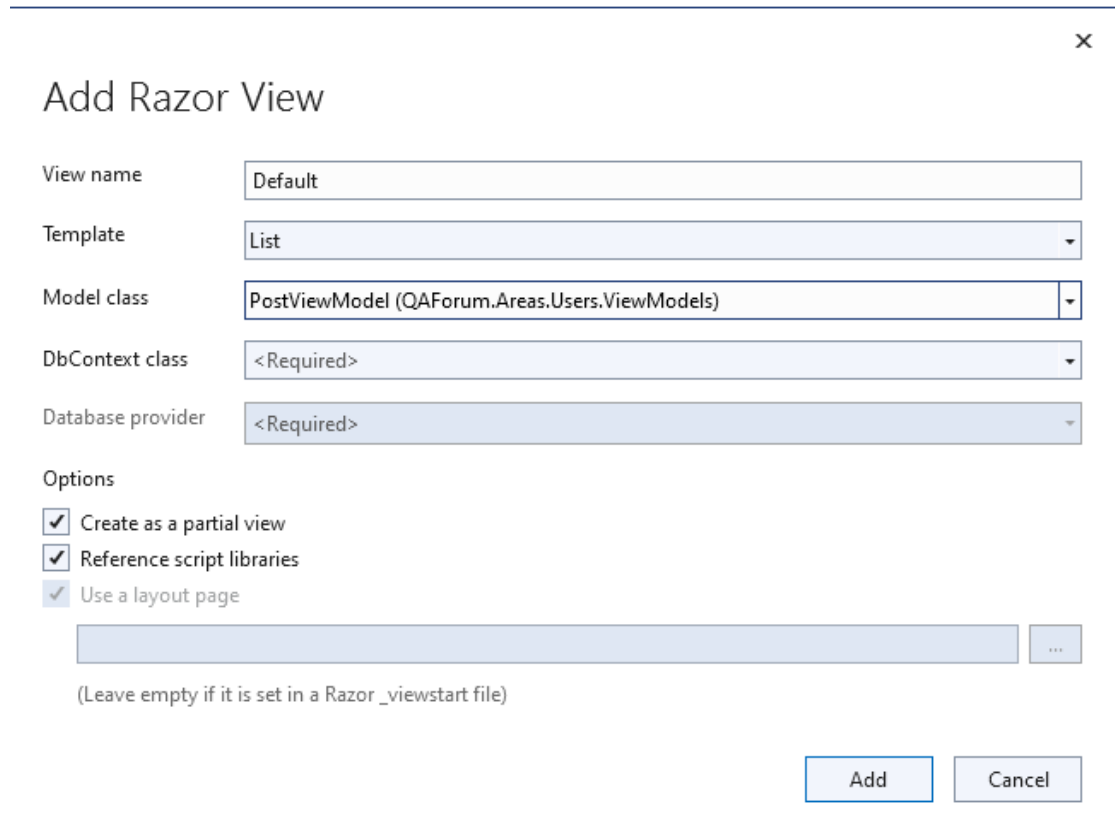
To show you a different technique, we're going to use View Components to achieve this. Note that consistency of the design is important for real systems – using two different techniques to achieve the same thing within a single application is not good practice. The only reason we're doing that here is to show you a variety of techniques.

Create a folder `/Areas/Users/ViewComponents`

| | |
|--|---|
| | <p>In that folder, add a class called <code>PostListViewComponent</code>. Make it inherit from <code>ViewComponent</code>, and use constructor injection to inject a <code>ForumDbContext</code>:</p> <pre> public class PostListViewComponent : ViewComponent { private readonly ForumDbContext context; public PostListViewComponent(ForumDbContext context) { this.context = context; } } </pre> <p>Note that view components can be located in any folder in your project, but you should choose a sensible folder, as we have done here.</p> |
| | <p>Add a method that will invoke the view component. This method must be called <code>Invoke</code>. It can take whatever parameters we need, and it will return a view in a similar way to the way an action returns a view. Note the return type: <code>IViewComponentResult</code>.</p> <pre> public IViewComponentResult Invoke(int? threadId) { IQueryable<Post> posts = context.Posts; if (threadId.HasValue) { posts = posts.Where(p => p.ThreadId == threadId); } return View(PostViewModel.FromPosts(posts)); } </pre> |
| | <p>Under the <code>Areas/Users/Views</code> folder, make a sub-folder called <code>Shared</code>.</p> <p>Within the <code>shared</code> folder, make a sub-folder called <code>Components</code>. Within this, make another sub-folder called <code>PostList</code>.</p> <p>Note that the names of these folders is important. View components look for their views in a sub-folder called <code>Components</code>, which must itself be either within the <code>Shared</code> folder, or within the folder with the same name as the controller that is being used. Within this, a sub-folder with the same name as the view component must hold the view.</p>  |
| | <p>Right-click on the <code>PostList</code> folder, and select <code>Add</code>, then <code>View</code>.</p> <p>Set the view name to <code>Default</code>. Again, the name does matter here. Unless you configure it to something different (which is not recommended if there is only</p> |

one view for the component), view components will look for a view called Default.

Choose the List template, and the PostViewModel model class. Make sure that Create As Partial View is selected.



Fix the ActionLinks at the bottom of the view. Having learnt from our experience of the thread list, we'll also make sure we specify the controller in the ActionLink calls:

```
<td>
    @Html.ActionLink("Edit", "Edit", "Post", new { id=item.PostId }) |
    @Html.ActionLink("Details", "Details", "Post", new { id=item.PostId }) |
    @Html.ActionLink("Delete", "Delete", "Post", new { id=item.PostId })
</td>
```

Now, go to the PostController.

View Components are most useful when the master view does not have access to the required model at all. So change the Index action:

```
public IActionResult Index()
{
    return View(PostViewModel.FromPosts(context.Posts));
}
```

And now, right-click in the Index action and select Go To View.

Delete everything after the <h1>. Also, importantly, delete the @model line from the top of the view.

Below the <h1>, add the line:

```
@addTagHelper *, QAForum
```

Compile your program to ensure that the tag helpers can be found by intellisense. Then, add this line below the @addTagHelper:

```
<vc:post-list thread-id="null"></vc:post-list>
```

| | |
|--|--|
| | <p>The entire view should now look like this:</p> <pre><code>@{ ViewData["Title"] = "Index"; } <h1>Index</h1> @addTagHelper *, QAForum <vc:post-list thread-id="null"></vc:post-list></code></pre> <p>Note how the camel-case component name and parameters (PostList and ThreadId) have been turned into kebab-case (post-list and thread-id).</p> |
| | <p>Run your program, check you can still see the list of posts correctly</p> |
| | <p>It's not very convenient to have to use the @addTagHelper command every time we want to use the view component. Delete the line from Default.cshtml, and instead add the same line to the bottom of /Areas/Users/Views/_ViewImports.cshtml.</p> <p>Once you've done that, any view inside the Users area will be able to use any tag helper (not just view components) in the QAForum namespace, without needing to explicitly have an @addTagHelper line of its own.</p> <p>Check that everything still works.</p> |
| | <p>Finally, we need to amend the Views/Thread/Details.cshtml file.</p> <p>The model for this view is ThreadViewModel, which does not have any record of the list of posts that go with the thread. But this does not matter. By using a view component, the component is able to fetch its own data, and does not need to be given any data except for the parameters.</p> <p>Add the following code between the two divs:</p> <pre><code></div> </div> <div> <h3>Posts</h3> <vc:post-list thread-id="@Model.ThreadId"></vc:post-list> </div> <div> @Html.ActionLink("Edit", "Edit", new { id = Model.ThreadId }) </code></pre> <p>Run your program, go to the Thread list, and look at the Details of a thread. Make sure you can see the posts that are in that thread.</p> |
| | <p>Take a moment to think about the pros and cons of the two different methods we've used to allow code re-use – partial views, and view components.</p> <p>With partial views, we needed to ensure that all of the data needed by the partial view was also available to the master view. Once this was done, setting up a partial view was very straightforward.</p> <p>However, it's not always easy (or desirable) for the master view to gather together all the data needed by every partial view it uses. View components</p> |

required a little bit setting up, but they were able to retrieve their own model data and do their own processing.

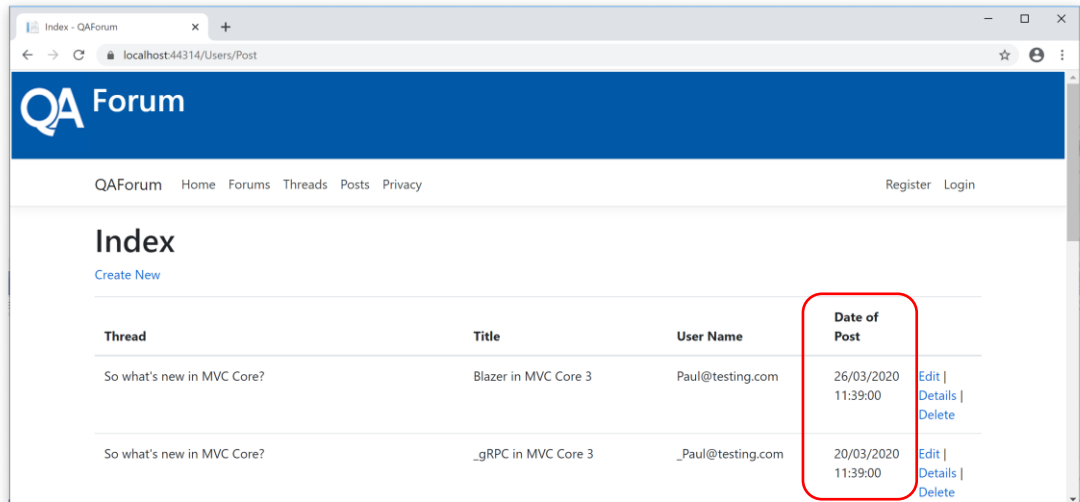
Which technique do you think is most appropriate in this instance?

Can you think of a different scenario where the other technique would be preferred?

Display Templates

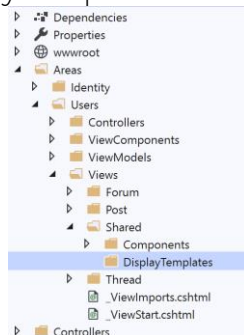
Run the program, and look at the list of posts.

The PostDateTime column doesn't look very neat, does it?



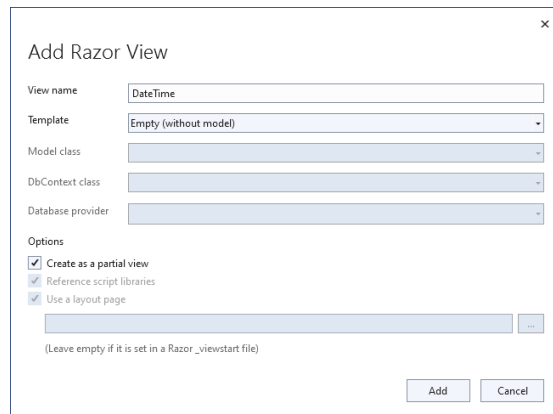
Let's see if we can improve it.

Under the folder Areas/Users/Views/Shared, create a sub-folder called DisplayTemplates.



Note that the folder *must* be called DisplayTemplates

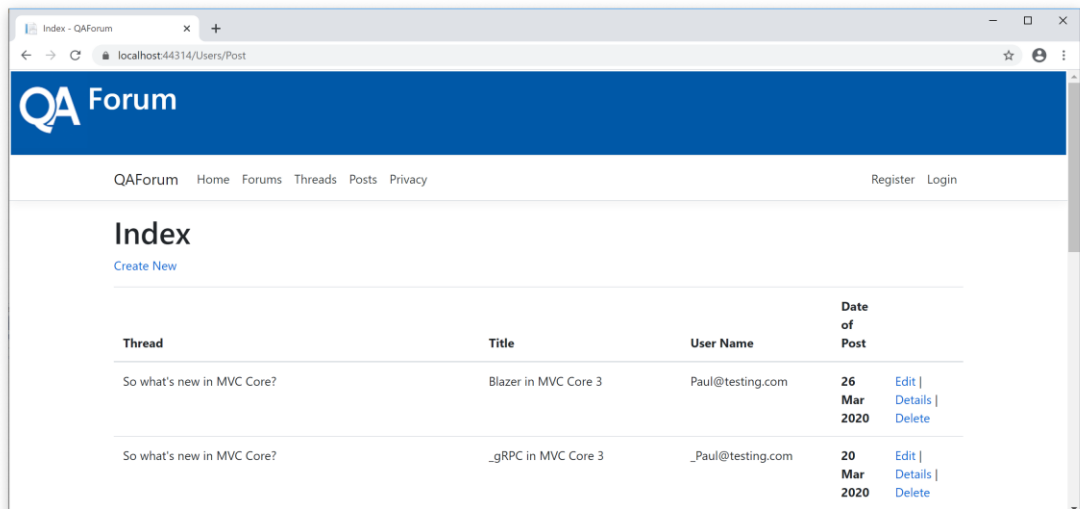
Right-click on DisplayTemplates, and select Add, View.
Name the template DateTime. Use the Empty template, and ensure the Partial View checkbox is ticked:



In the new file, add this code:

```
@model DateTime
<strong>@Model.ToString("dd MMM yyyy")</strong>
```

Run the program, and go back to the posts list. You can see that, because the name of the display template matches the data type of the column in question, it has automatically been applied in every view where the @Html.DisplayFor() helper function has been used – that includes the Post list and details views.



But each row here is very long, and that's not creating a view that's pleasing to look at. Let's add another template

Add another view to the DisplayTemplates folder. This time, call the view ShortenedString, but again choose the Empty template and ensure Partial View is selected.

Here is the code for the ShortenedString template:

```
@model string
@{
    const int maxLength = 30;

    if (Model.Length <= maxLength)
    {
        @Model
    }
}
```

```

else
{
    @(Model.Substring(0, maxLength) + "...")
}
}

```

This time, since the name of the template does not match the name of the data type, we need to apply it specifically in the places where we want it. This has the advantage that we can pick and choose where we want the template applied.

The only place where this template is to be used is when viewing the thread name within the post list. So, modify the PostViewModel class as follows:

```

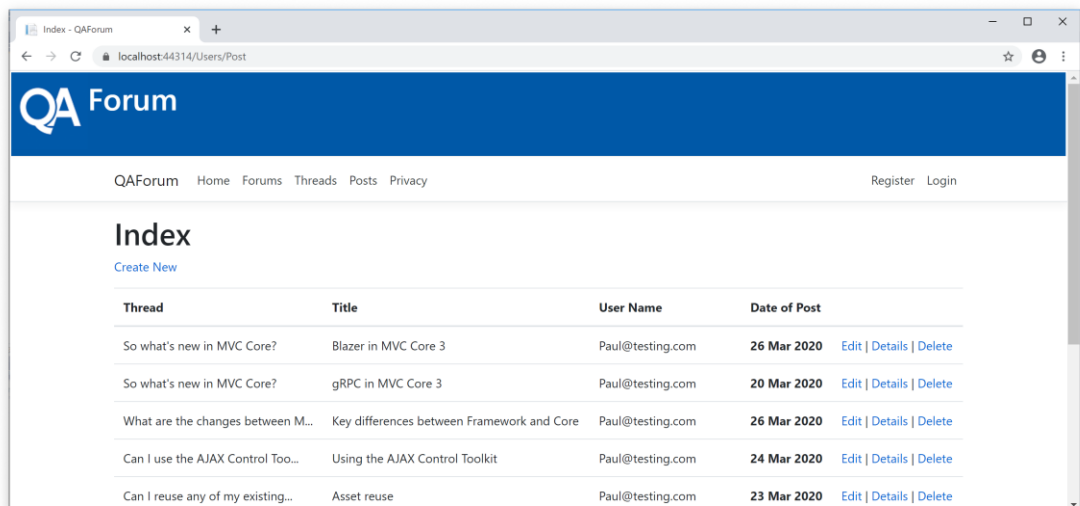
public int PostId { get; set; }

[UIHint("ShortenedString")]
public string Thread { get; set; }

public string Title { get; set; }

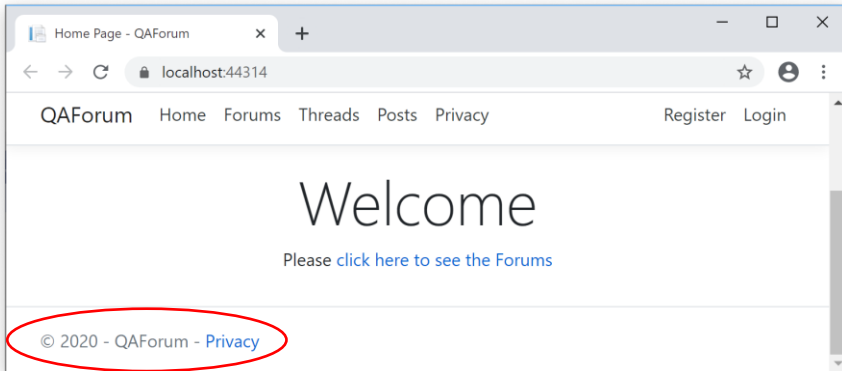
```

Have another look at the post list now:



Much better! But check that the Details view has not been affected, because it does not have the [UIHint] attribute on its view-model.

If You Have Time – Creating a Tag Helper

| | |
|--|--|
| | <p>At the bottom of every screen in our program is a copyright message. This has been created by the layout page:</p>  <p>Our final task for this chapter is to change this to include not only the company name but also an email address. And we're going to do this using a tag helper.</p> |
| | <p>Create a new folder at the root level of your application (i.e. not in the Areas folder), called TagHelpers</p> |
| | <p>Inside that folder, create a class called CompanyDetailsTagHelper. To start with, the tag helper won't do very much. It will just change a <company-details> tag (which is not recognised by web browsers) to an <a> tag and add some content. The code for this class is as follows:</p> <pre>public class CompanyDetailsTagHelper : TagHelper { public override void Process(TagHelperContext context, TagHelperOutput output) { output.TagName = "a"; output.Content.SetContent("admin@qaforums.com"); } }</pre> |
| | <p>In an earlier step, we added an @addTagHelper to the _ViewImports.cshtml file in the Users area. But we want this tag helper to be available in the layout file, which is <i>not</i> in that area. So we need to add the same tag helper to the root views folder. In /Views/_ViewImports.cshtml, add to the end of the file:</p> <pre>@addTagHelper *, QAForum</pre> <p>The, recompile your program to ensure that intellisense is able to find the tag helper.</p> |
| | <p>Open _Layout.cshtml. Scroll to the bottom of the file, and find the <footer> section. Change it as follows:</p> <pre><footer class="border-top footer text-muted"> <div class="container"> &copy; @DateTime.Now.Year <company-details></company-details> </div> </footer></pre> <p>Note, as you type the <company-details> tag, that when you type the first > symbol, it automatically adds the </company-details> to your code. Also, note how intellisense is aware of the change from Pascal case to kebab case.</p> |
| | <p>Run the program, and check you can see the new email address at the bottom of each page.</p> |

| | |
|--|--|
| | <p>Tag helpers can make use of dependency injection.</p> <p>Let's inject the company details, instead of hard coding them. Add this class to the Models folder:</p> <pre> public class CompanyDetails { public string CompanyName { get; set; } public string Email { get; set; } } </pre> |
| | <p>Modify the tag helper so that the company details are injected:</p> <pre> public class CompanyDetailsTagHelper : TagHelper { private readonly CompanyDetails companyDetails; public CompanyDetailsTagHelper(CompanyDetails companyDetails) { this.companyDetails = companyDetails; } public override void Process(TagHelperContext context, TagHelperOutput output) { output.TagName = "a"; output.Content.SetContent(companyDetails.Email); } } </pre> |
| | <p>Now, add this line to the ConfigureServices method in the Program.cs file:</p> <pre> Builder.Services.AddSingleton<CompanyDetails>(new CompanyDetails { CompanyName = "QA Forums", Email = "admin@qaforums.com" }); </pre> <p>Check that your email address is still shown at the bottom of every page.</p> |
| | <p>Although our tag helper adds an <a> tag to our page, it doesn't set its href attribute. To set this, add the following line:</p> <pre> public override void Process(TagHelperContext context, TagHelperOutput output) { output.TagName = "a"; output.Content.SetContent(companyDetails.Email); output.Attributes.Add("href", "mailto:" + companyDetails.Email); } </pre> <p>Now, you should get a link that you can click on at the bottom of each page.</p> |

Finally, tag helpers can have attributes of their own. Let's add an attribute to our tag helper. The data type of this attribute will be an enum, so first of all add the enum to the top of the CompanyDetailsTagHelper.cs file, just above (but not inside) the class:

```
public enum DisplayOptions
{
    EmailOnly,
    NameOnly,
    Both
}
```

Now, add a property into the class, and also decorate the class with a [HtmlTargetElement] attribute, which enables us to specify the HTML attributes:

```
[HtmlTargetElement("company-details", Attributes = "display-options")]
public class CompanyDetailsTagHelper : TagHelper
{
    public DisplayOptions DisplayOptions { get; set; }
    private readonly CompanyDetails companyDetails;
```

Add some code to make use of this new property:

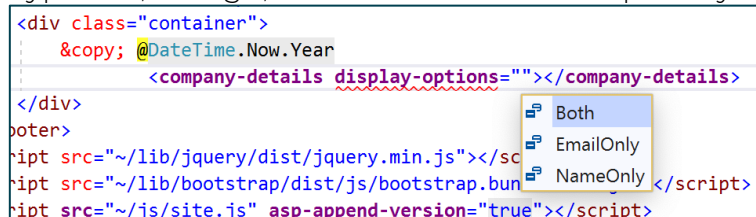
```
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    string content = "";
    switch (DisplayOptions)
    {
        case (DisplayOptions.EmailOnly):
            content = companyDetails.Email;
            break;
        case (DisplayOptions.NameOnly):
            content = companyDetails.CompanyName;
            break;
        case (DisplayOptions.Both):
            content = $"{companyDetails.CompanyName} [{companyDetails.Email}]";
            break;
    }

    output.TagName = "a";
    output.Content.SetContent(content);
    output.Attributes.Add("href", "mailto:" + companyDetails.Email);
}
```

Recompile your program to make sure intellisense picks up this new property. Then, in the layout file, re-type the <company-details> attribute:

```
<div class="container">
    &copy; @DateTime.Now.Year
    <company-details display-options="NameOnly"></company-details>
</div>
```

As you type this, though, note the intellisense help that you get:



```
<div class="container">
    &copy; @DateTime.Now.Year
    <company-details display-options="NameOnly"></company-details>
</div>
<script src="~/lib/jquery/dist/jquery.min.js"></script>
<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
<script src="~/js/site.js" asp-append-version="true"></script>
```

Intellisense is fully aware of the data type of the property, and offers suggestions that are appropriate to the data type!

Pick whichever option you like out of the three available, and test that it works.

| | |
|--|---|
| | Congratulations, you have now seen how to use a range of view features in your MVC application! |
|--|---|