Exercise 14 : State

Objective

In this exercise you use a variety of different techniques to manage state in your application.

This exercise will take around 80 minutes.

Referenced material

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

Maintaining Session State

•	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.
•	We are going to start by adding a "Recently Viewed Posts" list to the application. Our first task is to create a view-model to represent this list, which will be based on the Post model. Add a class to the Areas/Users/ViewModels folder, called RecentPostViewModel. Replace the blank class with the following two classes. Because the classes are very closely related to each other, we're going to choose to put them both in the same file.

```
    public class RecentPostViewModel
    {
        public const int MAX_RECENT_ITEMS = 5;

        public int PostId { get; set; }
        public string Title { get; set; }

        public static RecentPostViewModel FromPost(Post post)
        {
            return new RecentPostViewModel
            {
                PostId = post.PostId,
                Title = post.Title
            };
        }
    }

    public static class RecentPostViewModelExtensions
    {
        public static void AddRecentPost(this List<RecentPostViewModel> list,
                                              RecentPostViewModel post)
        {
            // If the item is already in the list, remove it so that it
            // gets re-inserted at the top
            int index = list.FindIndex(p => p.PostId == post.PostId);
            if (index != -1)
            {
                list.RemoveAt(index);
            }

            // Add the item to the top of the list
            list.Insert(0, post);
            // Remove items from the bottom if the list is too big
            if (list.Count > RecentPostViewModel.MAX_RECENT_ITEMS)
            {
                list.RemoveRange(RecentPostViewModel.MAX_RECENT_ITEMS,
                        list.Count - RecentPostViewModel.MAX_RECENT_ITEMS);
            }
        }

    }
```

The second class contains an extension method, which allows us to add a new item to a list in such a way that it maintains the concept of a recently-used list, ensuring that it doesn't contain duplicates and doesn't get too big. Take a look over the logic, and check you understand how it works.

- Add a field to the top of the PostController class to represent the recently used list, and add some code to the Details() method to add a post to this list:

```
    [Area("Users")]
    public class PostController : Controller
    {
        private List<RecentPostViewModel> recentPosts =
                             new List<RecentPostViewModel>();
        private readonly ForumDbContext context;
        ....
        ....
        ....

        public ActionResult Details(int id)
        {
            var post = context.Posts.Single(p => p.PostId == id);
            recentPosts.AddRecentPost(RecentPostViewModel.FromPost(post));
            return View(PostDetailsViewModel.FromPost(post));
        }
```

- Before we go any further, let's see if what we've done so far works.

  Set a breakpoint on the new line of code you have just added to the Details() method. Run the program by pressing F5 (*not* Ctrl-F5, since we want to enter debug mode), and follow these steps:

  Go to the Posts view, and click the Details button next to one of the posts.

  The program will stop at the breakpoint. Hover over the word `recentPosts`, and confirm that it contains no items. That's what we expect at this point.

  Press F10 to step over the line that adds the recently used post.

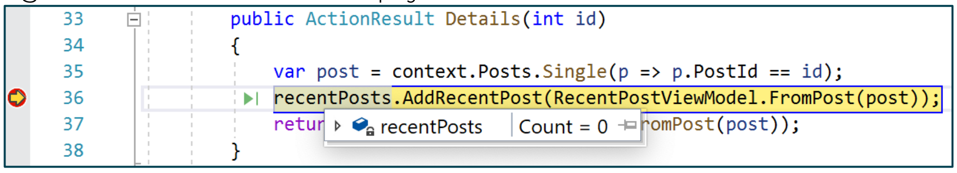
  Hover over `recentPosts` once again, and confirm that it now contains 1 item. Again, this is what we expect.

  Press F5 to allow the program to continue running.

  Back in your web browser, go back to the Posts list, and click the Details button next to a different post.

  When the program stops at the breakpoint again, check the contents of the recently used list.

  We'd expect that it should still contain the first item we added to it a moment ago. But it doesn't. It's empty.

  ```
  public ActionResult Details(int id)
  {
      var post = context.Posts.Single(p => p.PostId == id);
      recentPosts.AddRecentPost(RecentPostViewModel.FromPost(post));
      retur                                       romPost(post));
  }
  ```

  Something has gone wrong. Stop the program, and remove the breakpoint.

- The problem is that we are storing the recently used list in a field. Each new instance of the controller results in a new set of fields, and that means a new recently used list. And since ASP.Net creates a new instance of the controller for every single request, that means that every request gets a brand new, empty, recently used list.

  Clearly, having a field is not going to work. **Remove the field** from the top of the class. (Don't worry about the errors in the code where we reference that field – we'll fix them in a moment.)

- We can fix this by making use of "session state".

  To enable session state, we need to add two new services in Program.cs. The first of these services is a distributed cache – we've chosen to use the memory cache (which is actually not suitable for distributed caching, but will be perfect for our needs – we will investigate this in the If You Have Time section at the end of this lab). The second service we're adding is the Session service:

  ```
      var builder = WebApplication.CreateBuilder(args);

      // Add services to the container.
      services.AddDistributedMemoryCache();
  ```

```
    services.AddSession();
```

The other configuration item we need to change is to add the Session middleware to the pipeline. We configure this in the Configure method:

```
            ....
            ....
            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthentication();
            app.UseAuthorization();

            app.UseSession();

            ....
```

The position of this line *is* important – it must come *after* UseRouting() and *before* the mapped controller routes.

- Now that we've enabled session state, we can modify the Details() method in the Post controller as follows:

```
    public ActionResult Details(int id)
    {
        var post = context.Posts.Single(p => p.PostId == id);

        List<RecentPostViewModel> recentPosts;
        var json = HttpContext.Session.GetString("RECENT_POSTS");
        if (json == null)
        {
            recentPosts = new List<RecentPostViewModel>();
        }
        else
        {
            recentPosts =
                JsonSerializer.Deserialize<List<RecentPostViewModel>>(json);
        }

        recentPosts.AddRecentPost(RecentPostViewModel.FromPost(post));

        HttpContext.Session.SetString("RECENT_POSTS",
                                JsonSerializer.Serialize(recentPosts));
        return View(PostDetailsViewModel.FromPost(post));

    }
```

Note, when you resolve the missing "using" statement for `JsonSerializer`, that we have added the namespace `System.Text.Json`. Since this may not be the only option available, make sure you choose the right one!

- Once again, set a breakpoint on the `recentPosts.AddRecentPost` line, and repeat the test that we did above. This time, you should find that the list of posts gets retained from one request to the next.

Now that we've seen the reason for using session state, and how it works, we should think about how to integrate it into the architecture of our program.

There's a fair bit of boilerplate code needed every time we access session state (serialising and de-serialising the data, checking whether the item exists in the session, ensuring we use the same key each time). We will need to use this data in more than one place, so we should extract this boilerplate code into a reusable method.

We're going to use the repository pattern to do this. We will create an interface for our repository, as well as creating a repository class that contains this boilerplate code. Using the interface together enables easy testing should we require it.

We are going to make extensive use of dependency injection. Pay attention to how we use dependency injection to inject the repository into the controller, and also to inject a class into the repository which gives us access to the HttpContext object (from which we can find the Session object).

- Add a new sub-folder to the Areas/Users, called Repositories.

- Into that folder, add an interface called IStateRepository:

```
    public interface IStateRepository
    {
        public void SetRecentPosts(List<RecentPostViewModel> recentPosts);
        public List<RecentPostViewModel> GetRecentPosts();

    }
```

- Our StateRepository class will need to access the HTTP context. To enable this, add the following line to ConfigureServices() in Program.cs:

```
    var builder = WebApplication.CreateBuilder(args);

    // Add services to the container.
    builder.Services.AddHttpContextAccessor();
```

- In the Repository folder, add a class, StateRepository:

```
public class StateRepository : IStateRepository
{
    private const string RECENT_POSTS_KEY = "RECENT_POSTS";
    private readonly IHttpContextAccessor httpContextAccessor;

    public StateRepository(IHttpContextAccessor httpContextAccessor)
    {
        this.httpContextAccessor = httpContextAccessor;
    }

    public List<RecentPostViewModel> GetRecentPosts()
    {
        var json = httpContextAccessor.HttpContext.Session.GetString
                                            ("RECENT_POSTS");
        if (json == null)
        {
            return new List<RecentPostViewModel>();
        }
        else
        {
            return JsonSerializer.Deserialize<List<RecentPostViewModel>>(json);
        }
    }

    public void SetRecentPosts(List<RecentPostViewModel> recentPosts)
    {
        httpContextAccessor.HttpContext.Session.SetString(
                RECENT_POSTS_KEY, JsonSerializer.Serialize(recentPosts));
    }
}
```

- And now, back in Program.cs, ensure that the repository is registered so that it can be injected. We'll also register another interface, IActionContextAccessor, because we're going to need that soon, and adding it now saves us going backwards and forwards repeatedly:

```
            Builder.Services.AddHttpContextAccessor();
            Builder.Services.AddSingleton<IActionContextAccessor,
                                ActionContextAccessor>();
            Builder.Services.AddScoped<IStateRepository, StateRepository>();
```

- Now that our repository is complete and registered, we can modify the PostController to use the repository.

  Use constructor injection to get hold of an instance of the repository:

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

            public PostController(ForumDbContext context, IStateRepository state)
            {
                this.context = context;
                this.state = state;

            }
```

  And then modify the Details() method to use the repository instead of accessing the state directly itself:

```
            public ActionResult Details(int id)
            {
                var post = context.Posts.Single(p => p.PostId == id);
                var recentPosts = state.GetRecentPosts();
                recentPosts.AddRecentPost(RecentPostViewModel.FromPost(post));
                state.SetRecentPosts(recentPosts);
                return View(PostDetailsViewModel.FromPost(post));

            }
```

  That's a much neater implementation, with all the logic for setting and retrieving state neatly packaged together in its own reusable class.

Finally, we're ready to display the recent posts list. We will create a view component to show the list, and include that view component in every page in the Areas/Users section of our website.

Once again, we will make extensive use of dependency injection. We will inject the state repository into our view component. We will also modify our layout page, using dependency injection to inject some system objects that are required by the layout to decide whether to include the view component.

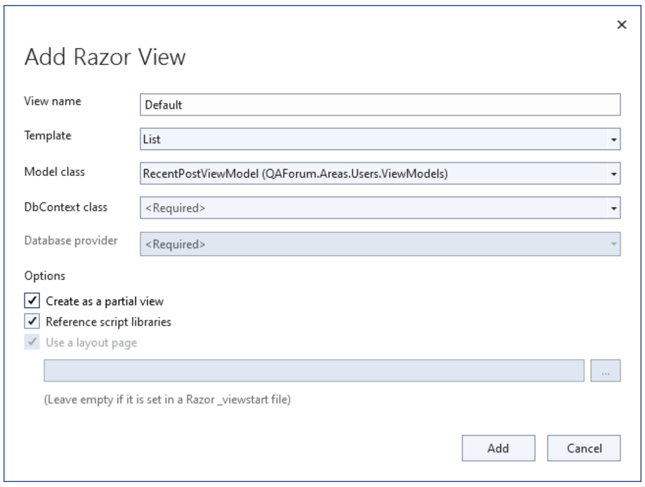
- Let's a create a view component that will show the post list.
  In the Areas/Users/ViewComponents folder, add a class called RecentPostsViewComponent:

```
    public class RecentPostsViewComponent : ViewComponent
    {
        private readonly IStateRepository state;

        public RecentPostsViewComponent(IStateRepository state)
        {
            this.state = state;
        }

        public IViewComponentResult Invoke()
        {
            return View(state.GetRecentPosts());

        }

    }
```

- Add the view for the view component.

  First, add a Components folder within /Areas/Users/Views/Shared. (We previously had this folder in our project, but we removed it when we moved the only file in it to a different location).

  Then, inside Areas/Users/Views/Shared/Components, add another sub-folder called RecentPosts.
  Right-click on the RecentPosts folder, and select Add, View. Set the view name to Default. Choose the List template, with a model type of RecentPostViewModel. Ensure that "Create as Partial View" is selected:



Add Razor View

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

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

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

- Replace the code in the view with the following:

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

```
<h5>Recently Viewed Posts</h5>

<div class="small">
    @if (Model.Count() == 0)
    {
        <p>No recent posts</p>
    }
    @foreach (var item in Model)
    {
        <p>
            @Html.ActionLink(Html.ValueFor(modelItem => item.Title),
                           "Details", "Post", new { id = item.PostId })
        </p>
    }
</div>
```

Then, build the application so that intellisense can find the new view component.

- We're going to arrange things so that the recent posts list gets shown on all of the Forum, Thread and Post pages.

  To do that, we're going to add some partial pages to the layout view. The partial pages will be within Areas/Users/Views, and so will be only be included for views in that area.

  Modify /Views/Shared/_Layout.cshtml. Add the following lines to the top of the file:

```
@using Microsoft.AspNetCore.Mvc.Razor
@using Microsoft.AspNetCore.Mvc.Infrastructure
@inject IRazorViewEngine Engine
@inject IActionContextAccessor ActionContextAccessor

<!DOCTYPE html>
<html lang="en">
<head>
```

  And then replace the `@RenderBody()` line with the following:

```
    <div class="container">
        <main role="main" class="pb-3">
            <div class="row align-items-start">
                @if (Engine.FindView(ActionContextAccessor.ActionContext,
                                     "_AfterBodyPartial", false).Success)
                {
                    <div class="col-md-10">
                        @RenderBody()
                    </div>
                    <div class="d-none d-md-block col-md-2">
                        <partial name="_AfterBodyPartial" />
                    </div>
                }
                else
                {
                    <div class="flex-grow-1">
                        @RenderBody()
                    </div>
                }
            </div>
        </main>

    </div>
```

•	Now, add a partial view, _AfterBodyPartial, to /Areas/Users/Views/Shared. Make it an empty partial view, and then replace the contents as shown here:
	`<div class="ml-2 mt-5 border rounded p-2">`
	`    <vc:recent-posts></vc:recent-posts>`
	`</div>`
	The Bootstrap classes which are being added to the various <div> tags here and in the previous step create a layout where, so long as the browser window's width is not too small to show the recently used list, it is positioned on the right hand portion of the screen, with a rounded border.
	
	On small screens, the recently used list is not shown.

•	Run the program. Check that the recent post list does not show on the home page or the privacy page, but that it does show on all the Forum, Thread and Post pages. Go to the Post Details for several posts, and check that it updates correctly. While the program is running, start a second web browser, and copy/paste the URL from the first browser into the second. This is simulating having a second user accessing the website at the same time as the first user. Check that each user/browser has their own, independent list of recent posts.

In this part of the lab, we have seen how to use session state to keep track of information from one request to the next. We have also made extensive use of dependency injection, to inject our own classes and also some system classes into various components of our system – a topic which we began exploring at the very start of the course, but which has continued to be important throughout.

Maintaining Session State

In the previous section, you saw how session state can be used to maintain state throughout the user's entire session.

Sometimes, we need to maintain state for much shorter periods of time. When the user submits a form, such as a Create form, the web browser sends two requests to the server in a very short space of time. The first request contains the form data – the server saves the data, and responds with a status of 302-Found. This 302-Found response includes a new URL, and the browser then sends a further request to the server at the new URL. In MVC, we implement this with a call to RedirectToAction(), which returns a RedirectToActionResult.

We may need to pass data from the action of the first request, on to the action that we are redirecting to. Since these actions handle two different requests, they can not share data without using some kind of state management technique, but using session state is not appropriate because the data does not need to be maintained for the whole session.

For this scenario, TempData is perfect. TempData allows data to be stored during one request, and will maintain it only until another request retrieves it.

When a user adds a post, we want to show a message indicating that the post has been added.
But, after adding a post, the user is redirected to the Index action. How is the Index action to know whether a post has just been added?
We will use TempData to allow the Create action and the Index action to communicate with each other.

- Open the PostController, and add the following line to the top of the class:

```
    [Area("Users")]
    public class PostController : Controller
    {
        private readonly string SUCCESS_MESSAGE = "SUCCESS_MESSAGE";
```

- Now, locate the [HttpPost] Create method, and add a line of code which stores a message in TempData if the post was saved successfully:

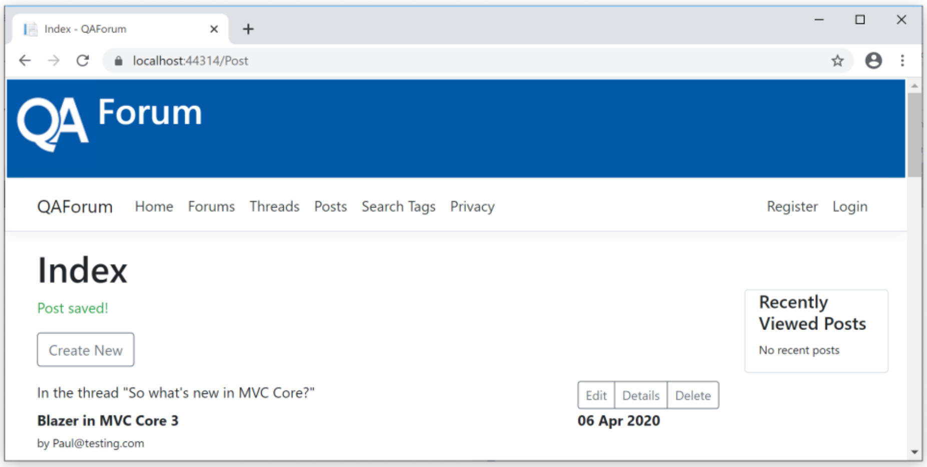
```
        public ActionResult Create(PostWriteViewModel viewModel)
        {
            try
            {
                if (ModelState.IsValid)
                {
                    ....
                    context.Posts.Add(post);
                    context.SaveChanges();

                    TempData[SUCCESS_MESSAGE] = "Post saved!";
                    return RedirectToAction(nameof(Index));
                }
                else
```

- After we set this message in TempData, we then call RedirectToAction, to redirect to the Index action.
So, next, locate the Index action, and add some code to retrieve the message:

```
        [TypeFilter(typeof(CustomErrorAttribute),
            Arguments = new object[] { "PostError"})]
        public IActionResult Index()
        {
            ViewBag.Message = TempData[SUCCESS_MESSAGE];
            return View();

        }
```

- Finally, right-click in the Index action and choose Go To View.
Add the message to the top of the view:

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

<h1>Index</h1>

<p class="text-success">@ViewBag.Message</p>

<p>
    <a asp-action="Create" class="btn btn-outline-secondary">Create New</a>
</p>


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

•	Run the program, add a post, and check that the message gets displayed.  Index - QAForum · localhost:44314/Post · QA Forum · QAForum Home Forums Threads Posts Search Tags Privacy Register Login · Index · Post saved! · Create New · In the thread "So what's new in MVC Core?" · Blazer in MVC Core 3 · by Paul@testing.com · Edit Details Delete · 06 Apr 2020 · Recently Viewed Posts · No recent posts

Notice that we didn't use the repository pattern here. The main reason for that is because we were adding nothing more than a string to TempData. If the Index method tries to extract the string and it's not there, it will receive a "null" value, which is perfectly acceptable in this case. There is no need to seralise and deserialise a string before adding it to TempData. Also, the only place this piece of data is being used is in a single controller, the Post controller – so we can keep things like the key for the data in a private constant within the Post controller.

If we had needed to do anything more complex than this, using the repository pattern may have been the better option.

Using Cookies to Store Data

Session state and Temp Data both store data on the server. For many state management scenarios, this is ideal, but it does use resources on the server, and sometimes it's more convenient to store state on the web browser, using cookies.

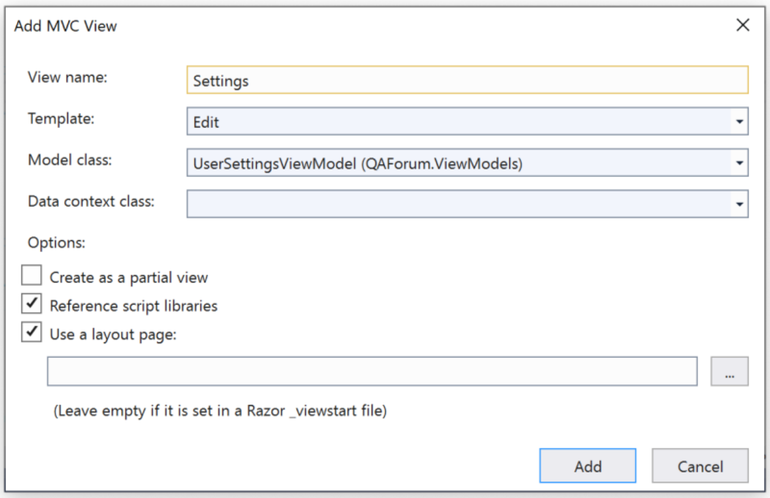
When storing state on the web browser, we must always be aware that the user is now able to delete or modify that data, and this poses security concerns. But for something like user options, where there are no security issues, cookies can be a useful tool.

•	Let's add a dark mode option to our website.
	Right-click on the root of the QAForum project, and add a folder called ViewModels.
	Into this new folder, add the following class:
	`public class UserSettingsViewModel`
	`{`
	`    [Display(Name = "Dark Mode")]`
	`    public bool DarkMode { get; set; }`
	`}`

- Open the HomeController, and add the following two methods:

```
        public IActionResult Settings()
        {
            var viewModel = new UserSettingsViewModel();
            return View(viewModel);
        }

        [HttpPost]
        public IActionResult Settings(UserSettingsViewModel viewModel)
        {
            // Save the settings - TO DO


            return RedirectToAction("Index");

        }
```

- Right-click in one of the Settings methods, and select Add View.
  Add a view with the Edit template, using a model class of UserSettingsViewModel. Make sure that Create As Partial View is *not* selected:



Add MVC View

View name: Settings

Template: Edit

Model class: UserSettingsViewModel (QAForum.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)

- Modify the resulting view. Remove the <h4> tag from the top of the view, and remove the <div> from the end of the file which includes a “back to list” link:

```
<h1>Settings</h1>

~~<h4>UserSettingsViewModel</h4>~~
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Settings">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group form-check">
                <label class="form-check-label">
                    <input class="form-check-input" asp-for="DarkMode" />
                        @Html.DisplayNameFor(model => model.DarkMode)
                </label>
            </div>
            <div class="form-group">
                <input type="submit" value="Save" class="btn btn-primary" />
            </div>
        </form>
    </div>
```

```
</div>

~~<div>~~
~~    <a asp-action="Index">Back to List</a>~~
~~</div>~~
```

- Next, we will add a link to the settings page on to our navigation bar.
  Open /Views/Share/_Layout.cshtml, and add the following navigation item:

```
<div class="navbar-collapse collapse d-sm-inline-flex flex-sm-row-reverse">
    <partial name="_LoginPartial" />
    <ul class="navbar-nav">
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home"
                                          asp-action="Settings">&#9881;</a>
        </li>
    </ul>
    <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home"
                                          asp-action="Index">Home</a>
        </li>
```

  (Note that &#9881; is the code for the Unicode character showing a Gear symbol. It should work on most modern browsers, but you could replace it with the word "Settings" if it doesn't display correctly.)

- Now, it's time to save the settings data into a cookie.
  Add the following constant into the HomeController class:

```
public const string DARK_COOKIE = "Dark";
```

  Modify the two Settings actions in the HomeController as follows:

```
        public IActionResult Settings()
        {
            var viewModel = new UserSettingsViewModel
            {
                DarkMode = Request.Cookies[DARK_COOKIE] == "1" ? true : false
            };
            return View(viewModel);
        }

       [HttpPost]
        public IActionResult Settings(UserSettingsViewModel viewModel)
        {
            // Save the settings
            Response.Cookies.Append(DARK_COOKIE, viewModel.DarkMode ? "1" : "0",
               new CookieOptions { Expires = DateTimeOffset.Now.AddYears(10) });
            return RedirectToAction("Index");

        }
```

  Notice how we store the cookie in the Response object to send to the web browser, and we retrieve it from the Request object when the browser sends it back to the server. It's not possible to create a cookie that never expires, so we set it to last 10 years.
  Although 10 years is plenty of time, it's also a good idea to re-send the cookie every time the user visits the home page, so that the expiry date keeps getting extended:

```
        public IActionResult Index()
        {
            var dark = Request.Cookies[DARK_COOKIE] == "1" ? true : false;
            Response.Cookies.Append(DARK_COOKIE, dark ? "1" : "0",
               new CookieOptions { Expires = DateTimeOffset.Now.AddYears(10) });

            return View();
        }
```

- In the Assets folder is a file called bootstrap-dark.css. Drag this file onto wwwroot/lib/bootstrap/dist/css. (We found this dark Bootstrap theme on the website https://bootswatch.com/)
- Open bundleconfig.json (from the root of the project), and add the following section:

```
[
  {
    "outputFileName": "wwwroot/css/siteplusbootstrap.css",
    "inputFiles": [
      "wwwroot/lib/bootstrap/dist/css/bootstrap.css",
      "wwwroot/css/site.css"
    ],
    "minify": {
      "enabled": true
    }
  },
  {
    "outputFileName": "wwwroot/css/siteplusbootstrap-dark.css",
    "inputFiles": [
      "wwwroot/lib/bootstrap/dist/css/bootstrap-dark.css",
      "wwwroot/css/site.css"
    ],
    "minify": {
      "enabled": true
    }
  },
  {

    "outputFileName": "wwwroot/js/site.min.js",
```

- Add the following to the top of _Layout.cshtml. We will need the HttpContext to get access to the cookies (yes, that's exactly the same HttpContext that we used earlier to get access to the Session object in the StateRepository class!):

```
@using Microsoft.AspNetCore.Mvc.Razor
@using Microsoft.AspNetCore.Mvc.Infrastructure
@using Microsoft.AspNetCore.Http
@using QAForum.Controllers
@inject IRazorViewEngine Engine
@inject IActionContextAccessor ActionContextAccessor
@inject IHttpContextAccessor HttpContextAccessor
```

- Then, modify the next section of the layout, so that it checks the value of the cookie, and includes the appropriate version of Bootstrap:

```
@{
    bool dark =
     HttpContextAccessor.HttpContext.Request.Cookies[HomeController.DARK_COOKIE]
            == "1";
}

<!DOCTYPE html>
<html lang="en"><head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - QAForum</title>
    <environment include="Development">
        @if (dark)
        {
            <link rel="stylesheet"
                  href="~/lib/bootstrap/dist/css/bootstrap-dark.css" />
        }
        else
        {
            <link rel="stylesheet"
                  href="~/lib/bootstrap/dist/css/bootstrap.css" />
        }
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment exclude="Development">
        @if (dark)
        {
            <link rel="stylesheet"
                  href="~/css/siteplusbootstrap-dark.min.css" />
        }
        else
        {
            <link rel="stylesheet" href="~/css/siteplusbootstrap.min.css" />
        }
    </environment>
</head>
```

- The default navigation bar in /Views/Share/_Layout.cshtml needs a minor tweak, otherwise it won't appear correctly in dark mode. Remove the "bg-white" class from the <nav> element, as shown below. Also, add the class "text-dark" to the navbar-brand link. And then, add a check on whether we're using light mode or dark mode, and change the navbar toggler button as appropriate:

```
<body>
    <header>
        <div class="page-header">
            <div class="clearfix">
                <img class="float-left" src="~/images/qalogo.png" />
                <h1> Forum</h1>
            </div>
        </div>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm
                         navbar-light ~~bg-white~~ border-bottom box-shadow mb-3">
            <div class="container">
                <a class="navbar-brand text-dark" asp-area=""
                         asp-controller="Home" asp-action="Index">QAForum</a>
                <button class="navbar-toggler
                          @(dark?"navbar-dark":"navbar-light")" type="button"
                          data-toggle="collapse" data-target=".navbar-collapse"
                          aria-controls="navbarSupportedContent"
                          aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
```

- Finally, try out the program. Switch to dark mode and back.
  Check that, in both light and dark mode, you can close your web browser and re-start it, and that it remembers your setting. This setting is saved as a cookie by your web browser.

Warning! When using cookies, the law says that you must have consent from the user in most cases. You can read more about the law on the Information Commissioner's Office website: https://ico.org.uk/for-organisations/guide-to-pecr/cookies-and-similar-technologies/

- Let's ensure we get the required consent from the user.
  To enable this feature, add the following to the top of ConfigureServices in Setup.cs:

```
        public void ConfigureServices(IServiceCollection services)
        {
            services.Configure<CookiePolicyOptions>
                (options => options.CheckConsentNeeded = context => true);
```

  Then, add the following to the Configure method. As always when adding to the pipeline, the order of the middleware items does matter, so make sure you add this in the right place:

```
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            ....
            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseCookiePolicy();
```

- Modfy /Views/Shared/_Layout.cshtml to include the cookie consent banner:

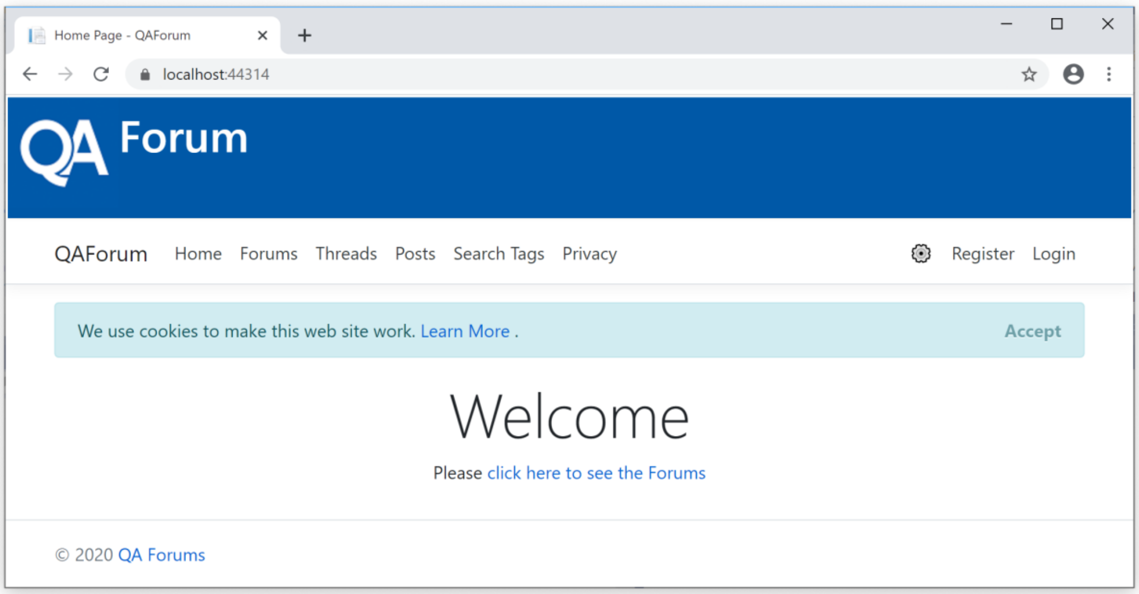
```
    </header>
    <div class="container">
        <partial name="_CookieConsentPartial" />
        <main role="main" class="pb-3">
            <div class="row align-items-start">
                @if (Engine.FindView(ActionContextAccessor.ActionContext,
                                          "_AfterBodyPartial", false).Success)
                {
                    <div class="col-md-10">
                        @RenderBody()
                    </div>
```

- And now, create a view in /Views/Shared called _CookieConsentPartial.cshtml. Use the Empty template, and create a Partial view.
  We got the outline for this file from the Microsoft website, and tweaked it a little: https://docs.microsoft.com/en-us/aspnet/core/security/gdpr?view=aspnetcore-3.1. Of course you can customise this as much as required for your own applications:

```
@using Microsoft.AspNetCore.Http.Features

@{
    var consentFeature = Context.Features.Get<ITrackingConsentFeature>();
    var showBanner = !consentFeature?.CanTrack ?? false;
    var cookieString = consentFeature?.CreateConsentCookie();
}

@if (showBanner)
{
    <div id="cookieConsent" class="alert alert-info alert-dismissible fade show"
                     role="alert">
        We use cookies to make this web site work.
        <a asp-area="" asp-controller="Home" asp-action="Privacy">
            Learn More
        </a>.
        <button type="button" class="accept-policy close" data-dismiss="alert"
```

```
                aria-label="Close" data-cookie-string="@cookieString">
            <span aria-hidden="true">Accept</span>
        </button>
    </div>
    <script>
        (function () {
            var button = document.querySelector
                    ("#cookieConsent button[data-cookie-string]");
            button.addEventListener("click", function (event) {
                document.cookie = button.dataset.cookieString;
            }, false);
        })();
    </script>
}
```

- Run the program. Notice the cookie banner underneath the navigation bar. DO NOT accept the cookie policy yet!



Home Page - QAForum
localhost:44314
QA Forum
QAForum Home Forums Threads Posts Search Tags Privacy Register Login
We use cookies to make this web site work. Learn More . Accept
Welcome
Please click here to see the Forums
© 2020 QA Forums

- Try using the website. Notice that none of the features we have added in this lab work any more. The dark mode does not work. The post list keeps clearing itself. The “post added” message does not show.

  With the cookie consent features enabled, only “essential” cookies are allowed until the user accepts the cookie policy. Our dark mode cookie is definitely not essential. And both Session and TempData make use of cookies to store the user’s session id, which is also not considered to be essential.

  Note that, if we wanted session cookies to be considered essential, we could modify the entry in Starup.cs/ConfigureServices which adds sessions, as follows:
  ```
        services.AddSession(options => options.Cookie.IsEssential = true);
  ```
  But since it’s possible for users to use our website without session cookies, albeit with some features not working properly, it would be wrong to do so for our application.

- Accept the cookie policy. Everything should start working again.

If You Have Time: Investigating State Memory Cache Options

- Run the application, and *open a new web browser*. Copy the application's URL into the new web browser. Add some posts to your recently used post list in the new web browser.

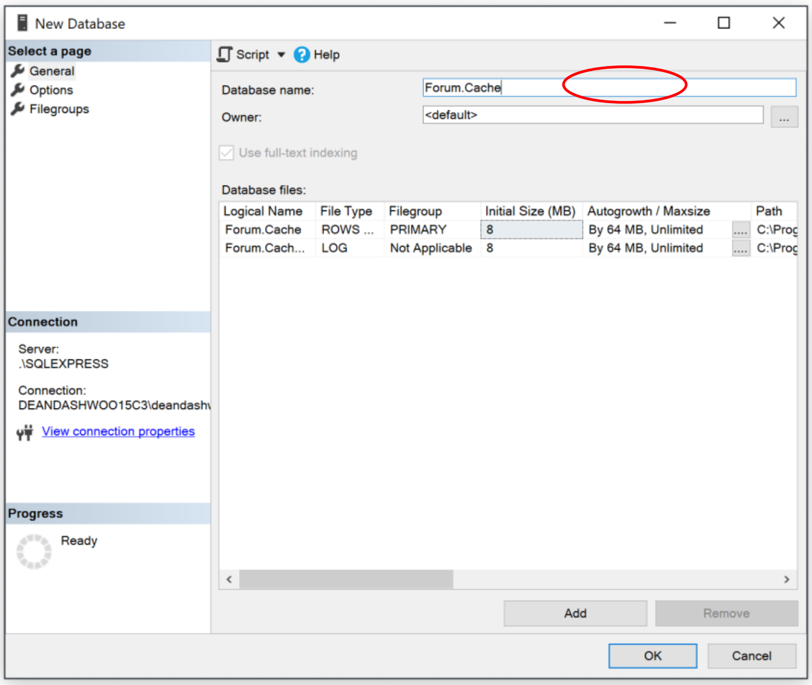
  Close down the application. The web browser that started automatically may close, but the one you opened for this step will remain open.

  Re-start the application. Open a new page in the separate web browser, and check that, even though you're using the same web browser instance as before you stopped the server, the recent post list has been cleared.

  That's because session state is currently stored in application memory. When the application stops, it loses all its state.

  The implications of this are even more serious if we are distributing our workload by using a web farm or a web garden. If this were the case, it could be that each request from a user gets directed to a different server, with a different set of state data in its memory. If we want our system to work in this environment, the state must be stored somewhere other than in server application memory.

- We are going to store our session state in SQL Server. Open SQL Server Management Studio, and in the object explorer, right-click on Databases and choose New Database.
  In the Database Name box, enter "Forum.Cache", and then press Ok.

  New Database

  Database name: Forum.Cache

  Owner: <default>

  Use full-text indexing

  Database files:

Logical Name	File Type	Filegroup	Initial Size (MB)	Autogrowth / Maxsize	Path
Forum.Cache	ROWS ...	PRIMARY	8	By 64 MB, Unlimited	C:\Proc
Forum.Cach...	LOG	Not Applicable	8	By 64 MB, Unlimited	C:\Proc

  Connection

  Server:
  .\SQLEXPRESS

  Connection:
  DEANDASHWOO15C3\deandash

  Progress

  Ready

- Click on the start menu, and type "CMD". Press return to open a command prompt. Type the following command:

  ```
  dotnet tool install --global dotnet-sql-cache --version 3.1.0
  ```

  Close the command prompt.

- Click the start menu, and again type “CMD”. Press return to open another command prompt (it sometimes doesn’t work to run the second command in the same window as the first). Type the following command:

  ```
  dotnet sql-cache create "Data Source=.\sqlexpress;Initial
  Catalog=Forum.Cache;Integrated Security=True;" dbo ForumCache
  ```

  (Note that this command must all be typed on one line. If you copy and paste from this document, it may not work because it spreads over two lines in the document.)
  This command adds the required table to the Forum.Cache database. It calls the table ForumCache.
  Close the command prompt.

- Open Nuget Package Manager for the QAForum project. Go to the Installed tab.

  Check the version of Microsoft.EntityFrameworkCore.SqlServer that is installed.

  Install the following package, being sure to use the same version number as the version of entity framework that is installed:
  Microsoft.Extensions.Caching.SqlServer

- Open appsettings.json. Add a new connection string, as follows:

  ```
    "ConnectionStrings": {
      "Forum.Users": "Server=.\\SqlExpress;Database=Forum.Users;
                      Trusted_Connection=True;MultipleActiveResultSets=true",
      "Forum.Data": "Server=.\\SqlExpress;Database=Forum.Data;
                      Trusted_Connection=True;MultipleActiveResultSets=true",
      "Forum.Cache": "Server=.\\SqlExpress;Database=Forum.Cache;
                      Trusted_Connection=True;MultipleActiveResultSets=true"

    },
  ```

- Open Startup.cs. Find the line in ConfigureServices() which adds the distributed memory cache. Comment out that line, and replace it with the following:

  ```
  //services.AddDistributedMemoryCache();
  services.AddDistributedSqlServerCache(options =>
  {
      options.ConnectionString = Configuration.GetConnectionString("Forum.Cache");
      options.SchemaName = "dbo";
      options.TableName = "ForumCache";

  });
  ```

- Repeat the test you did earlier, where you discovered that stopping and re-starting the application loses session data.
  This time, you should find that session data gets retained, even after stopping and re-starting the application. That is because the data is stored in SQL Server.

Comment out the AddDistributedSqlServerCache() call, and re-instate the AddDistributedMemoryCache() call.

Congratulations! You have now seen a variety of different techniques for maintaining state in an ASP.Net application.