

Exercise14: State

Objective

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

This exercise will take around 50 minutes.

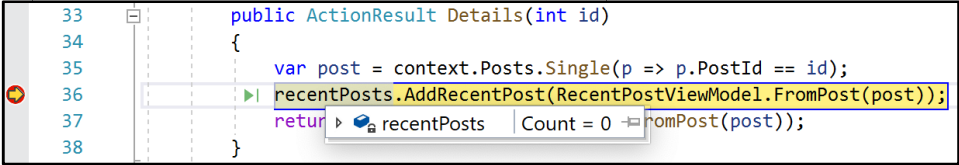
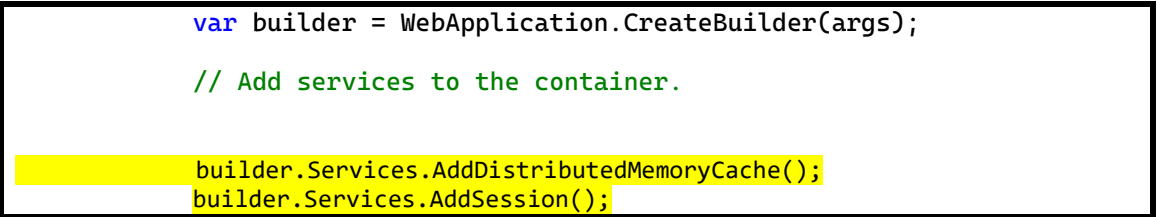
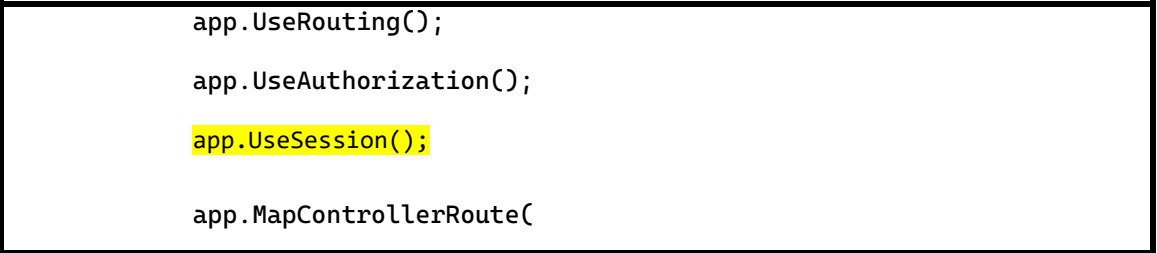
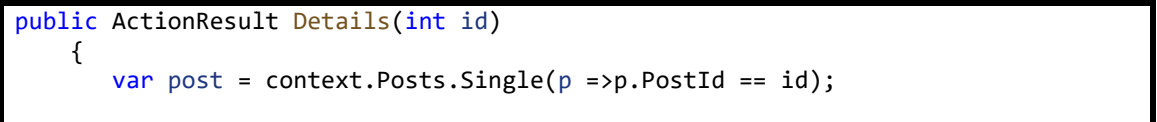
Referenced material

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

Maintaining Session State

1	Open the 'Begin' solution in Visual Studio and compile (Shift Ctrl+B).
2	<p>We are going to start by adding a “Recently Viewed Posts” list to the application.</p> <p>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 ViewModels folder, called RecentPostViewModel.</p> <p>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.</p> <pre>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 } }</pre>

	<pre> 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); } } </pre> <p>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.</p>
3	<p>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:</p> <pre> 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)); } } </pre>
4	<p>Before we go any further, let's see if what we've done so far works.</p> <p>Set a breakpoint on the new line of code you have just added to the Details() method. Run the program by pressing F5 (<i>not</i> Ctrl-F5, since we want to enter debug mode), and follow these steps:</p> <ul style="list-style-type: none"> • 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 <code>recentPosts</code>, 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 <code>recentPosts</code> 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.

	<ul style="list-style-type: none"> 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.  <p>Something has gone wrong. Stop the program, and remove the breakpoint.</p>
5	<p>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.</p> <p>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.)</p>
6	<p>We can fix this by making use of “session state”.</p> <p>To enable session state, we need to add two new services in the Program.cs file (Remember this must be done BEFORE we call the Build() method).</p>  <pre>var builder = WebApplication.CreateBuilder(args); // Add services to the container. builder.Services.AddDistributedMemoryCache(); builder.Services.AddSession();</pre> <p>The other configuration item we need to change is to add the Session middleware to the pipeline (Remember this must be done AFTER we call the Build() method).</p>  <pre>app.UseRouting(); app.UseAuthorization(); app.UseSession(); app.MapControllerRoute(</pre> <p>The position of this line <i>is</i> important – it must come <i>after</i> UseRouting() and <i>before</i> MapControllerRoute().</p>
7	<p>Now that we've enabled session state, we can modify the Details() method in the Post controller as follows:</p>  <pre>public ActionResult Details(int id) { var post = context.Posts.Single(p => p.PostId == id);</pre>

	<pre> 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)); } </pre> <p>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.</p>
8	<p>Once again, set a breakpoint on the <code>recentPosts.AddRecentPost</code> 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.</p>

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).

9	Add a new sub-folder to the root of the QAForum project called Repositories.
10	<p>Into that folder, add an interface called IStateRepository:</p> <pre> public interface IStateRepository { public void SetRecentPosts(List<RecentPostViewModel>recentPosts); } </pre>

	<pre> public List<RecentPostViewModel>GetRecentPosts(); } </pre>
11	<p>Our StateRepository class will need to access the HTTP context (It won't inherit that ability as our Controller does). To enable this, add the following line to Program.cs: (BEFORE the Build())</p> <pre> ... builder.Services.AddDistributedMemoryCache(); builder.Services.AddSession(); builder.Services.AddHttpContextAccessor(); </pre>
12	<p>In the Repository folder, add a class, StateRepository:</p> <pre> 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)); } } </pre>
13	<p>Now, back in Program.cs ensure that the repository is registered so that it can be injected.</p> <pre> ... builder.Services.AddHttpContextAccessor(); builder.Services.AddScoped<IStateRepository, StateRepository>(); </pre>
14	<p>Now that our repository is complete and registered, we can modify the PostController to use the repository.</p>

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

```
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. When we show the list of Posts, we will also show recently viewed Posts, if there are any.

- 15 Add the following code to the Index action of the PostController. To access any recently viewed posts held in Session state and put them in ViewBag.

```
public IActionResult Index()
{
    var recentPosts = state.GetRecentPosts();
    ViewBag.RecentPosts = recentPosts;
    return View(PostViewModel.FromPosts(context.Posts));
}
```

- 16 Goto the /Views/Post folder and pick up these recently viewed posts in index.cshtml.

Below the h1 tag add the following:-

```
<h1>Index</h1>

@if (ViewBag.RecentPosts.Count > 0)
{
    <div style="background-color:aqua">
        <h2>You have recently viewed these posts:-</h2>
        @foreach (var item in ViewBag.RecentPosts)
        {
            <p>
```

	<pre> @{ string title = item.Title; } @Html.ActionLink(title, "Details", "Post", new{ id = item.PostId }) </p> } </div> </pre>
17	Make sure the code compiles.
18	Run the program.
19	Click the link for Posts. You should see the posts but no recently viewed ones at this stage.
20	<p>Click the details for one of the posts then click the Posts link again. This time you should see the recently viewed post.</p> <p>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.</p>

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.

Using TempData

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`.

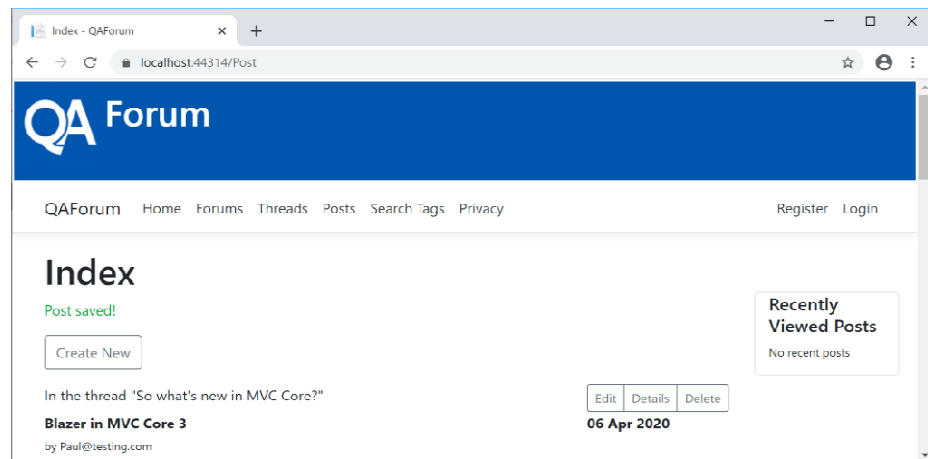
We may need to pass data from the action of the first request, on to the action that we are redirecting to. Since these actions handle two different requests, they cannot share data without using some kind of state management technique, but using session state is not appropriate because the data does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

21	When a user adds a post, we want to show a message indicating that the post has been added.
----	---

	<p>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?</p> <p>We will use TempData to allow the Create action and the Index action to communicate with each other.</p>
22	<p>Open the PostController, and add the following line to the top of the class:</p> <pre>public class PostController : Controller { Private readonly string SUCCESS_MESSAGE = "SUCCESS_MESSAGE";</pre>
23	<p>Now, locate the [HttpPost] Create method, and add a line of code which stores a message in TempData if the post was saved successfully:</p> <pre>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</pre>
24	<p>After we set this message in TempData, we then call RedirectToAction, to redirect to the Index action.</p> <p>So, next, locate the Index action in the PostController and add some code to retrieve the message:</p> <pre>public IActionResult Index() { ViewBag.Message = TempData[SUCCESS_MESSAGE]; ...</pre>
25	<p>Finally, right-click in the Index action and choose Go To View.</p> <p>Add the message to the top of the view:</p> <pre>@{ ViewData["Title"] = "Index"; } <h1>Index</h1> <p class="text-success">@ViewBag.Message</p> ...</pre>

26 Run the program, add a post, and check that the message gets displayed.



Using Cookies to Store Data

Session state and TempData 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.

27 Let's add a dark mode option to our website.
Right-click on the ViewModels folder and add a class called UserSettingsViewModel

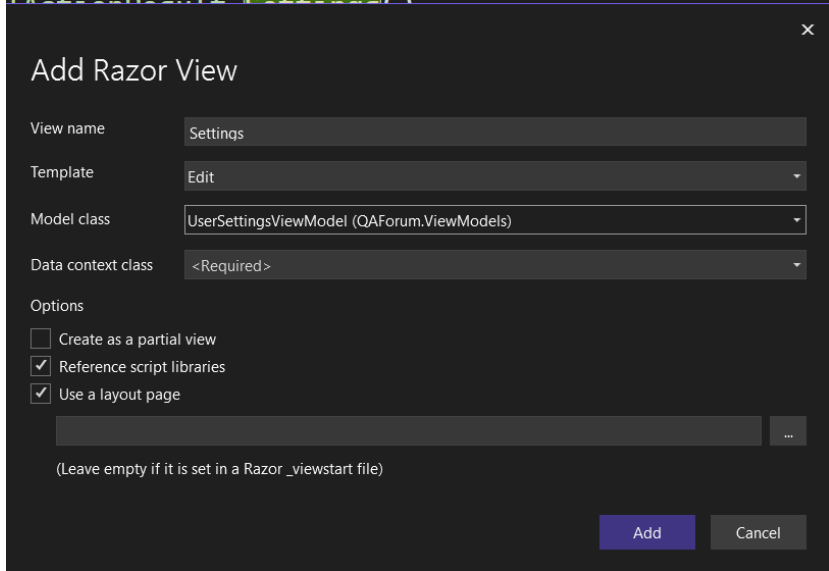
```
public class UserSettingsViewModel
{
    [Display(Name = "Dark Mode")]
    public bool DarkMode{ get; set; }
}
```

28 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");
}
```

	<pre>}</pre>
29	<p>Right-click in one of the Settings methods, and select Add View.</p> <p>Add a view with the Edit template, using a model class of UserSettingsViewModel.</p> 
30	<p>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:</p> <pre> <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> <asp-action="Index">Back to List </div> </pre>
31	<p>Next, we will add a link to the settings page on to our navigation bar.</p> <p>Open /Views/Share/_Layout.cshtml, and add the following navigation item:</p> <pre> <li class="nav-item"> </pre>

```

    <a class="nav-link text-dark" asp-controller="Home" asp-
action="Privacy">Privacy</a>
</li>

    <li class="nav-item">
        <a class="nav-link text-dark" asp-controller="Home" asp-
action="Settings">
            &#9881;</a>
    </li>

    ...

```

(Note that ⚙ 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.)

32 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) });
}

```

	<pre> return View(); } </pre>
33	<p>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/)</p>
34	
35	<p>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!):</p> <pre> @using Microsoft.AspNetCore.Mvc.Razor @using Microsoft.AspNetCore.Mvc.Infrastructure @using Microsoft.AspNetCore.Http @using QAForum.Controllers @Inject IHttpContextAccessor HttpContextAccessor </pre>
36	<p>Then, modify the next section of the layout, so that it checks the value of the cookie, and includes the appropriate version of Bootstrap:</p> <pre> @{ bool dark = HttpContextAccessor.HttpContext.Request.Cookies[HomeController.DARK_COOKIE] == "1"; } <!DOCTYPEhtml> <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> </pre>

37	<p>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:</p> <pre> <body> <header> <div class="page-header"> <div class="clearfix"> <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"> QAForum <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"> </button> </pre>
38	<p>Finally, try out the program. Switch to dark mode and back.</p> <p>Check that, in both light and dark mode, you can close your web browser and restart it, and that it remembers your setting. This setting is saved as a cookie by your web browser.</p>