CHAPTER 6

STATE MANAGEMENT ON ASP.NET CORE APPLICATION

CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- State Management on stateless HTTP,
- Server-side strategies: Session State, TempData, Using HttpContext,
- Cache Client-side strategies: Cookies, Query Strings, Hidden Fields

STATE MANAGEMENT ON STATELESS HTTP

HTTP is a stateless protocol. So, HTTP requests are independent messages that don't retain user values or app states. We need to take additional steps to manage state between the requests.

State can be managed in our application using several approaches.

Storage Approach	Description
Cookies	HTTP cookies. May include data stored using server-side app code.
Session state	HTTP cookies and server-side app code
TempData	HTTP cookies or session state
Query strings	HTTP query strings
Hidden fields	HTTP form fields
HttpContext	Server-side app code
Cache	Cache Server-side app code

SERVER-SIDE STRATEGIES: SESSION STATE, TEMPDATA, USING HTTPCONTEXT

Session State

Session state is an ASP.NET Core mechanism to store user data while the user browses the application. It uses a store maintained by the application to persist data across requests from a client. We should store critical application data in the user's database and we should cache it in a session only as a performance optimization if required.

ASP.NET Core maintains the session state by providing a cookie to the client that contains a session ID. The browser sends this cookie to the application with each request. The application uses the session ID to fetch the session data.

While working with the Session state, we should keep the following things in mind:

- A Session cookie is specific to the browser session
- When a browser session ends, it deletes the session cookie
- If the application receives a cookie for an expired session, it creates a new session that uses the same session cookie
- An application doesn't retain empty sessions
- The application retains a session for a limited time after the last request. The app either sets the session timeout or uses the default value of 20 minutes

- Session state is ideal for storing user data that are specific to a particular session but doesn't require permanent storage across sessions
- An application deletes the data stored in session either when we call the ISession.Clear implementation or when the session expires
- There's no default mechanism to inform the application that a client has closed the browser or deleted the session cookie or it is expired

A Session State Example

We need to configure the session state before using it in our application. This can be done in the ConfigureServices() method in the Startup.cs class:

services.AddSession();

The order of configuration is important and we should invoke the UseSession() before invoking UseMVC().

Let's create a controller with endpoints to set and read a value from the session:

```
public class WelcomeController : Controller
{
    public IActionResultIndex()
    {
        HttpContext.Session.SetString("Name", "John");
        HttpContext.Session.SetInt32("Age", 32);
        return View();
    }

    public IActionResultGet()
    {
        User newUser = new User()
        {
            Name = HttpContext.Session.GetString("Name"),
            Age = HttpContext.Session.GetInt32("Age").Value
        };
        return View(newUser);
    }
}
```

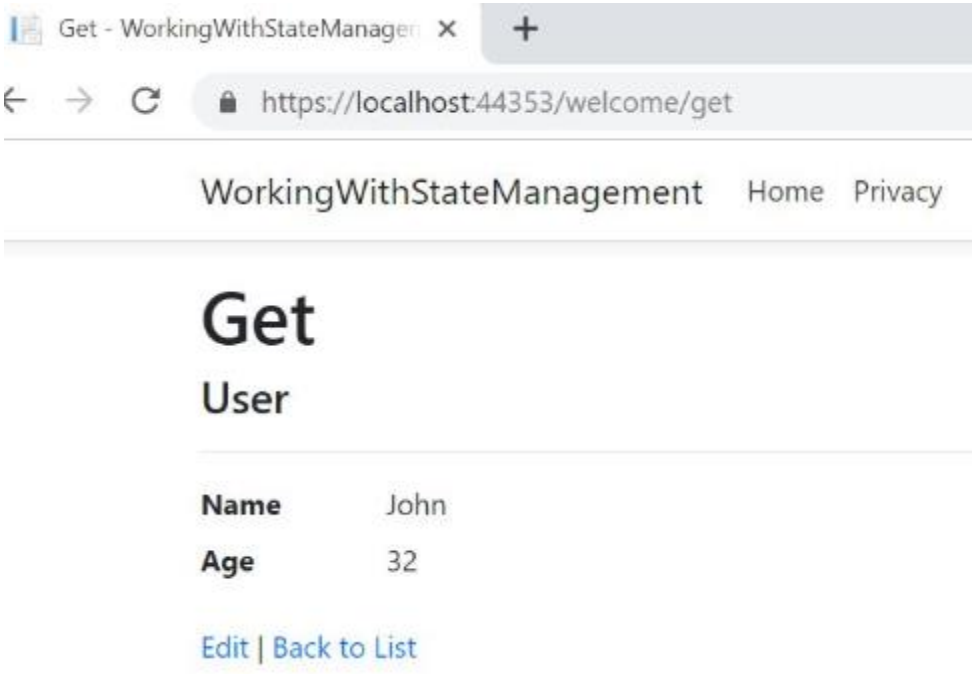
The Index() method sets the values into session and Get() method reads the values from the session and passes them into the view.

Let's auto-generate a view to display the model values by right-clicking on the Get() method and using the "Add View" option.

Now let's run the application and navigate to /welcome.

This will set the session values.

Now let's navigate to /welcome/get:



TempData

ASP.NET Core exposes the TempData property which can be used to store data until it is read. We can use the Keep() and Peek() methods to examine the data without deletion. TempData is particularly useful when we require the data for more than a single request. We can access them from controllers and views.

TempData is implemented by TempData providers using either cookies or session state.

A TempData Example

Let's create a controller with three endpoints. In the First() method, let's set a value into TempData. Then let's try to read it in Second() and Third() methods:

```
public class TempDataController : Controller
```

```
{
    public IActionResultFirst()
    {
        TempData["UserId"] = 101;
        return View();
    }

    public IActionResultSecond()
    {
        var userId = TempData["UserId"] ?? null;
        return View();
    }
    public IActionResultThird()
    {
        var userId = TempData["UserId"] ?? null;
        return View();
    }
}
```

Now let's run the application by placing breakpoints in the Second() and Third() methods.

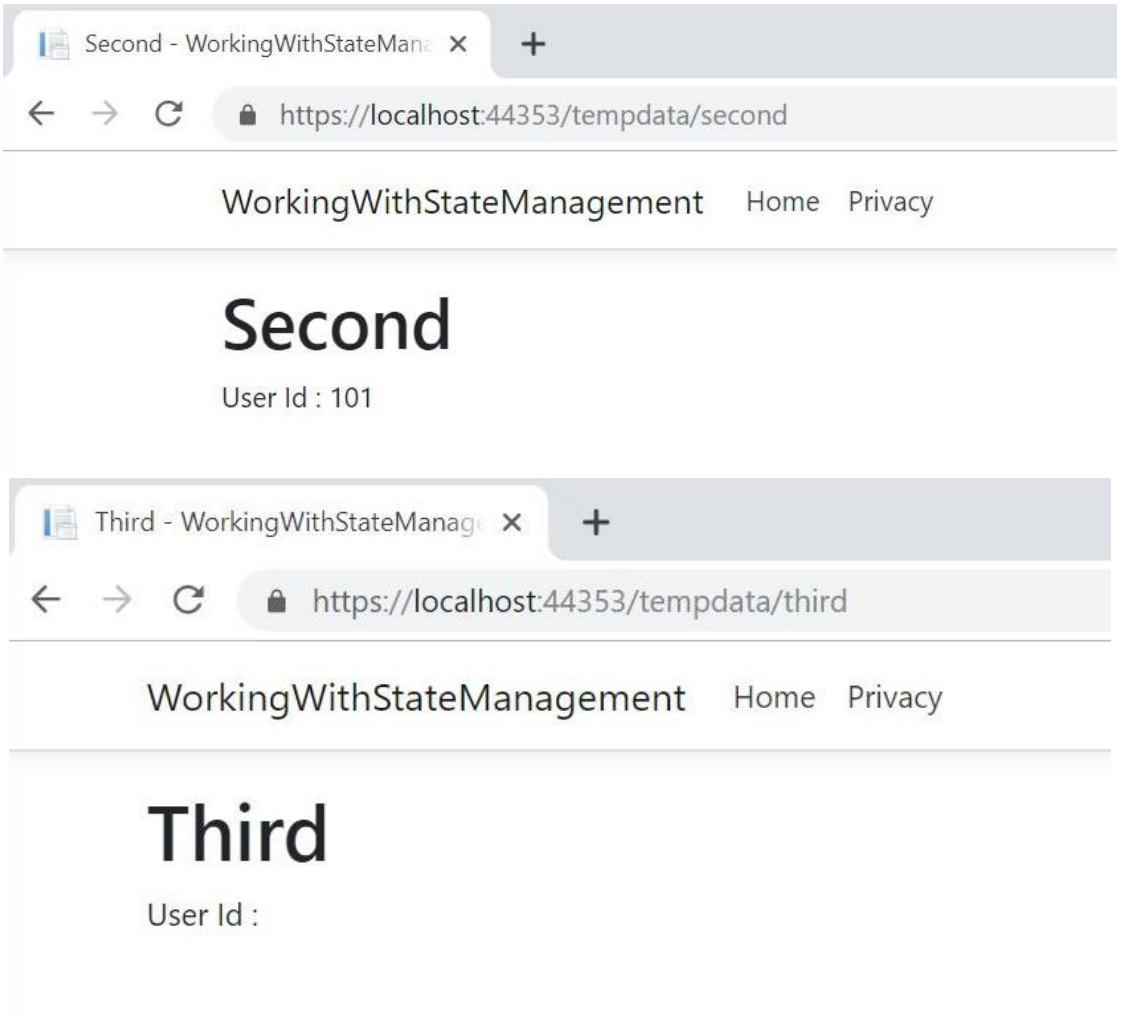
We can see that the TempData is set in the First() request and when we try to access it in the Second() method, it is available. But when we try to access it in the Third() method, it is unavailable as is retains its value only till its read.

Now let's move the code to access TempData from the controller methods to the views.

Let's create a view for the Second() action method:

```
@{
        ViewData["Title"] = "Second";
        var userId = TempData["UserId"]?.ToString();
}
<h1>Second</h1>
User Id : @userId
Similarly, let's create a view for the Third() action method:
@{
        ViewData["Title"] = "Third";
        var userId= TempData["UserId"]?.ToString();
}
<h1>Third</h1>
User Id : @userId
```

Let's run the application and navigate to /first, /second and /third



We can see that TempData is available when we read it for the first time and then it loses its value. Now, what if we need to persist the value of TempData even after we read it?

We have two ways to do that:

- **TempData.Keep()/TempData.Keep(string key):** This method retains the value corresponding to the key passed in TempData. If no key is passed, it retains all values in TempData.
- **TempData.Peek(string key):** This method gets the value of the passed key from TempData and retains it for the next request.

Let's slightly modify our second view with one of these methods:

```
var userId = TempData["UserId"]?.ToString();
TempData.Keep();
// OR
var userId = TempData.Peek("UserId")?.ToString();
```

Now let's run the application and navigate to /first, /second and /third.

We can see that the TempData value persists in the third page even after its read on the second page. Great!

In this section, we have learned how to pass values from one controller action method to another or to the view using TempData.

Using HttpContext

A HttpContext object holds information about the current HTTP request. The important point is, whenever we make a new HTTP request or response then the Httpcontext object is created. Each time it is created it creates a server current state of a HTTP request and response.

It can hold information like, Request, Response, Server, Session, Item, Cache, User's information like authentication and authorization and much more.

As the request is created in each HTTP request, it ends too after the finish of each HTTP request or response.

Example to Check request processing time using HttpContext class

This example check the uses of the HttpContext class. In the global.aspx page we know that a BeginRequest() and EndRequest() is executed every time before any Http request. In those events we will set a value to the context object and will detect the request processing time.

```
protected void Application_BeginRequest(object sender, EventArgs e)
{
HttpContext.Current.Items.Add("Begintime", DateTime.Now.ToLongTimeString());
}
protected void Application_EndRequest(object sender, EventArgs e)
{
    TimeSpan diff = Convert.ToDateTime(DateTime.Now.ToLongTimeString()) -
       Convert.ToDateTime(HttpContext.Current.Items["Begintime"].ToString());
}
```

Example to access current information using HttpContext class

```
protected void Page_Load(object sender, EventArgs e)
{
       Response.Write("Request URL"+  HttpContext.Current.Request.Url)
       Response.Write("Number of Session variable" +
                                HttpContext.Current.Session.Count);
      Response.Write("current Timestamp" + HttpContext.Current.Timestamp);
      Response.Write("Object in Application level " +
                                HttpContext.Current.Application.Count);
      Response.Write("Is Debug Enable in current Mode?" +
                                HttpContext.Current.IsDebuggingEnabled);
 }
```

CACHE CLIENT-SIDE STRATEGIES: COOKIES, QUERY STRINGS, HIDDEN FIELDS

Cookies

Cookies store data in the user's browser. Browsers send cookies with every request and hence their size should be kept to a minimum. Ideally, we should only store an identifier in the cookie and we should store the corresponding data using the application. Most browsers restrict cookie size to 4096 bytes and only a limited number of cookies are available for each domain.

Users can easily tamper or delete a cookie. Cookies can also expire on their own. Hence we should not use them to store sensitive information and their values should not be blindly trusted or used without proper validations.

We often use cookies to personalize the content for a known user especially when we just identify a user without authentication. We can use the cookie to store some basic information like the user's name. Then we can use the cookie to access the user's personalized settings, such as their preferred color theme.

Reading Cookie

```
//read cookie from IHttpContextAccessor
string cookieValueFromContext =
                    httpContextAccessor.HttpContext.Request.Cookies["key"];
//read cookie from Request object
string cookieValueFromReq = Request.Cookies["Key"];
```

Writing cookie

In this example,SetCookie method show how to write cookies. CookieOption is available to extend the cookie behavior.

```
public void SetCookie(string key, string value, int? expireTime)
{
       CookieOptions option = new CookieOptions();

   if (expireTime.HasValue)
       option.Expires = DateTime.Now.AddMinutes(expireTime.Value);
   else
       option.Expires = DateTime.Now.AddMilliseconds(10);

       Response.Cookies.Append(key, value, option);
}
```

Remove Cookie

```
Response.Cookies.Delete(key);
```

Query strings

We can pass a limited amount of data from one request to another by adding it to the query string of the new request. This is useful for capturing the state in a persistent manner and allows the sharing of links with the embedded state.

Let's add a new method in our WelcomeController:

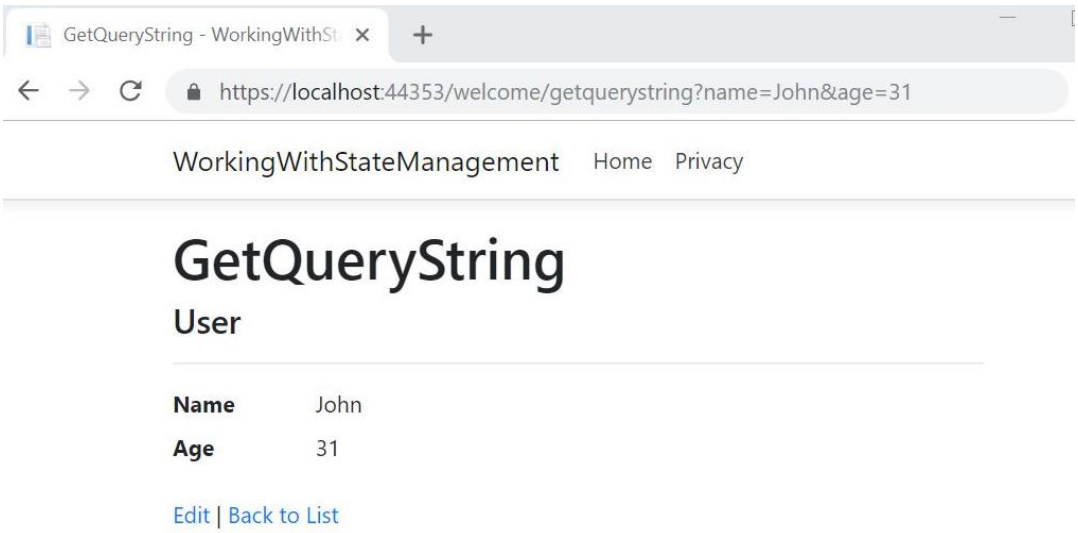
```
public IActionResultGetQueryString(string name, int age){
    User newUser = new User()
    {
        Name = name,
        Age = age
    };
    return View(newUser);
}
```

Model binding maps data from HTTP requests to action method parameters. So, if we provide the values for name and age as either form values, route values or query strings, we can bind those to the parameters of our action method.

For displaying the model values let's auto-generate a view as we did in the previous section.

Now let's invoke this method by passing query string parameters:

/welcome/getquerystring?name=John&age=31



We can retrieve both the name and age values from the query string and display it on the page.

As URL query strings are public, we should never use query strings for sensitive data.

In addition to unintended sharing, including data in query strings will make our application vulnerable to Cross-Site Request Forgery (CSRF) attacks, which can trick users into visiting malicious sites while authenticated. Attackers can then steal user data or take malicious actions on behalf of the user.

Hidden Fields

We can save data in hidden form fields and send back in the next request. Sometimes we require some data to be stored on the client side without displaying it on the page. Later when the user takes some action, we'll need that data to be passed on to the server side. This is a common scenario in many applications and hidden fields provide a good solution for this.

Let's add two methods in our WelcomeController:

```
[HttpGet]
public IActionResultSetHiddenFieldValue()
{
    User newUser = new User()
    {
        Id = 101,
        Name = "John",
        Age = 31
    };
    return View(newUser);
}
[HttpPost]
public IActionResultSetHiddenFieldValue(IFormCollectionkeyValues)
{
    var id = keyValues["Id"];
    return View();
}
```

The GET version of theSetHiddenValue() method creates a user object and passes that into the view.

We use the POST version of the SetHiddenValue() method to read the value of a hidden field Id from FormCollection.

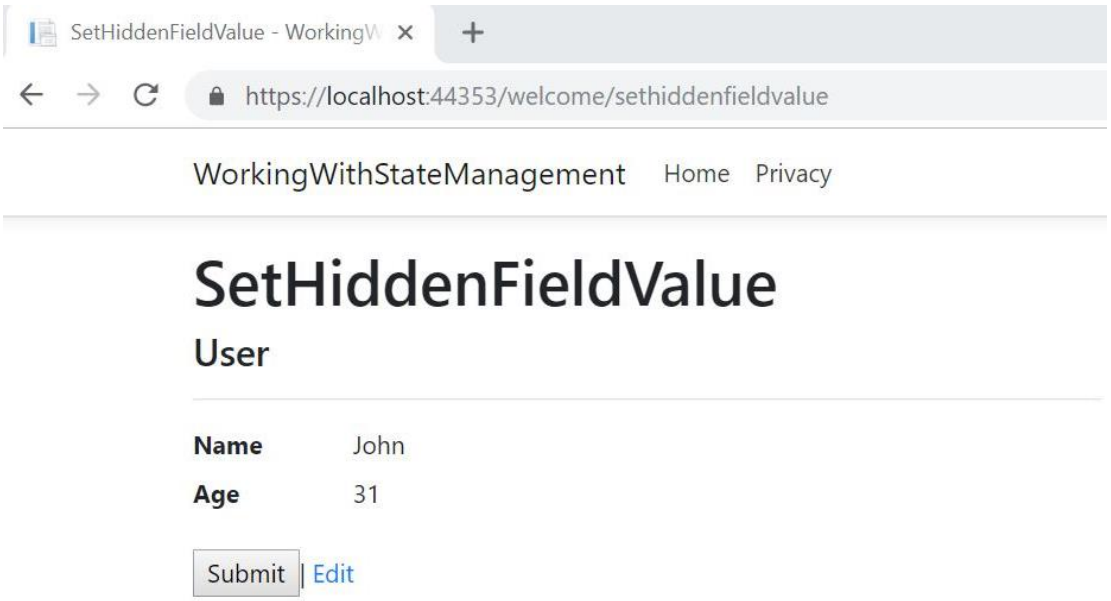
In the View, we can create a hidden field and bind the Id value from Model:

@Html.HiddenFor(model =>model.Id)

Then we can use a submit button to submit the form:

```
<input type="submit" value="Submit" />
```

Now let's run the application and navigate to /Welcome/SetHiddenFieldValue:



On inspecting the page source, we can see that a hidden field is generated on the page with the Id as the value:

```
<input id="Id" name="Id" type="hidden" value="101">
```

Now click the submit button after putting a breakpoint in the POST method. We can retrieve the Id value from the FormCollection:

Since the client can potentially tamper with the data, our application must always revalidate the data stored in hidden fields.

Discussion Exercise

1. Write about the State Management Strategies.
2. What is Session State? Show with an example to manage session state in ASP.NET Core.
3. Show the difference between TempData and Using HttpContext with suitable example.
4. How do you manage to handle state with client-side strategies?

❍❍❍