Unit 6
State Management on ASP.NET Core Application

HTTP

- HTTP is a stateless protocol
- **HTTP** is called as a stateless protocol because each request is executed independently, without any knowledge of the requests that were executed before it, which means once the transaction ends the connection between the browser and the server is also lost.
- So, we need to take additional steps to manage state between the requests.
- We can manage State in our application using following strategies
  1. Server Side Strategies
     - In server-side State Management all the information is stored in the server memory.
     - Due to this functionality there is more secure domains at the server side in comparison to Client-Side State Management.
     - This maintains information on the Server machine using
       a. Session State
       b. TempData
       c. HttpContext
       d. Cache
  2. Client Side Strategies
     - In client-side State Management, the state related information will directly get stored on the client-side.
     - That specific information will travel back and communicate with every request generated by the user then afterwards provides responses after server-side communication.
     - This maintains information on the Client machine using
       a. Cookies
       b. QueryString
       c. HiddenFields

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.
- Some important facts about Session
  1. A Session cookie is specific to the browser session
  2. When a browser session ends, it deletes the session cookie
  3. If the application receives a cookie for an expired session, it creates a new session that uses the same session cookie
  4. 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
  5. Session state is ideal for storing user data that are specific to a particular session but doesn't require permanent storage across sessions
  6. An application deletes the data stored in session either when we call the ISession.Clear implementation or when the session expires
  7. We should not store sensitive data in session state. The user might not close the browser and clear the session cookie. Some browsers maintain valid session cookies across browser windows.
- 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 as follows

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSession();//must be placed before AddMvc()
    services.AddMvc();
}
```

Or

```
public void ConfigureServices(IServiceCollection services)
{
        services.AddSession(options =>
            {
                options.IdleTimeout = TimeSpan.FromSeconds(100);//set custom
                                                                timeout
            });
              services.AddMvc();
}
```

- Then, we need to enable session state in the Configure() method in the same class:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
        app.UseSession();
 }
```

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

WAP to demonstrates how to set and read a value from a session.

Controller: HomeController.cs

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
namespace SessionDemo.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            HttpContext.Session.SetString("uname", "Bhesh");
            HttpContext.Session.SetString("pwd", "11111111");
            return (RedirectToAction("Get"));
        }
        public IActionResult Get()
        {
            string uname = HttpContext.Session.GetString("uname").ToString();
            string pwd = HttpContext.Session.GetString("pwd").ToString();
            ViewBag.username = uname;
            ViewBag.password = pwd;
            return View();
        }
    }
}
```

View: Get.cshtml

```
<html>
<body>
    Username: @ViewBag.username;
    <br />
    Password: @ViewBag.password;
</body>
</html>
```

In the above program, when a request is made as /Home/Index then the username and passwordd value will be stored in sessioni and will redirect to next action method Get which reads the session values, set them in viewbag and displays them in view named Get.cshtml.

Write ASP.net Core MVC application to create a login page that allows user to input username and password. When user presses the submit button, validate the user login and redirect to welcom page that shows the welcome message. If the user directly enters the URL of welcome page, it must automatically redirect to login page. [username:bhesh Password: 1111]

Model: Login.cs

```
namespace SessionDemo.Models
{
    public class Login
    {
        public string username { get; set; }
        public string password { get; set; }
    }
}
```

Controller: HomeController.cs

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using SessionDemo.Models;
namespace SessionDemo.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult showloginpage()
        {
            return View(null);
        }
        [HttpPost]
        public IActionResult verifylogin(Login l)
        {
            string uname = "bhesh";
            string pwd = "11111111";
            if (l.username == uname && l.password == pwd)
            {
                HttpContext.Session.SetString("uname", uname);
              return (RedirectToAction("welcome"));
            }
            else
            {
                return Content("Invalid Uername or Password");
            }
        }
        public IActionResult welcome()
        {
            string uname = HttpContext.Session.GetString("uname");
            if (uname == null)
            {
                return (RedirectToAction("showloginpage"));
            }
            else
            {
                return View();
            }
        }
    }
}
```

View: showloginpage.cshtml

```
@model SessionDemo.Models.Login
<html>
<body>
    <form method="post" action="verifylogin">
        <label asp-for="username"></label>
        <input asp-for="username" />
        <br />
        <label asp-for="password"></label>
        <input asp-for="password" />
```

```
            <br />
            <input type="submit" value="submit" />
        </form>
    </body>
    </html>
```

View: welcome.cshtml

```
@using Microsoft.AspNetCore.Http;
<html>
<body>
    Welcome : <h1>@Context.Session.GetString("uname")</h1>
</body>
</html>
```

TempData

- ASP.NET Core exposes the TempData property which can be used to store data until it is read.
- 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.
- We can use the Keep () and Peek () methods to examine the data without deletion.
  1. TempData.Keep() or 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.
  2. TempData.Peek(string key): This method gets the value of the passed key from TempData and retains it for the next request i.e. don't' mark it for deletion

Example:

Controler: HomeController.cs

```
using Microsoft.AspNetCore.Mvc;
namespace TempDataDemo.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult First()
        {
            TempData["uname"] = "bhesh";//This will Persist for the next request until it is
                                                                                    read
            return( RedirectToAction("Second"));
        }
        public IActionResult Second()
        {
            return View();
        }
        public IActionResult Third()
        {
            return View();
        }
    }
}
```

View: Second.cshtml

```
<html>
<body>
    @*Username:<h1>@TempData["uname"]</h1>
    @{
        TempData.Keep();//if removed then in the value of Tempdata wont be availablle in the
                                                                                next request
    }*@

    @{
        var nam = TempData.Peek("uname");//will return the value without marking for deletion
    }
    Username:<h1>@nam</h1>
```

```
        @Html.ActionLink("Click me", "Third");

    </body>
    </html>
```

View: Third.cshtml

```
    <html>
    <body>
        Username:<h1>@TempData["uname"]</h1>
    </body>
    </html>
```

So, when we run the application by navigating to /First it will store "uname" value in TempData and self redirects to /Second. In the view associated with /Second , we have read the TempData value, this will mark the TempData key for deletion but use of Peek() or Keep() method retains it for next request without deletion. So in the Next request made by user by clicking "Click me" link, still we can get the value associated with TempData with key "uname".

HttpContext

- A HttpContext object holds information about the current HTTP request.
- Whenever we make a new HTTP request or response then the Httpcontext object is created.
- It can hold information like, Request, Response, Server, Session, Item, Cache, User's information like authentication and authorization and much more.

Write ASP.net core MVC program to create a page with two texbox for entering name & address and a submit button. When the user clicks the submit button display the information in another page. Use concept of HttpContext

Controller:HomeController.cs

```
using Microsoft.AspNetCore.Mvc;
namespace TempDataDemo.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
        public IActionResult formsubmitted()
        {
            return View();
        }
    }
}
```

View: Index.cshtml

```
<html>
<body>
    <form method="post" action="Home/formsubmitted">
        Name:
        <input type="text" name="txt_name" />
        <br />
        Address:
        <input type="text" name="txt_add" />
        <br />
        <input type="submit" value="submit" />
    </form>
</body>
</html>
```

View:formsumitted.cshtml

```
<html>
<body>
```

```
        @{
            string name = Context.Request.Form["txt_name"];
            string address = Context.Request.Form["txt_add"];
        }
        Name:@name;
        <br />
        Address:@address;
    </body>
    </html>
```

Also the controller and view can be written as

Controller:HomeController.cs

```
using Microsoft.AspNetCore.Mvc;
namespace TempDataDemo.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
        public IActionResult formsubmitted()
        {
            string name = HttpContext.Request.Form["txt_name"];
            string address = HttpContext.Request.Form["txt_add"];
            ViewBag.name = name;
            ViewBag.address = address;
            return View();
        }
    }
}
```

View: Index.cshtml

```
<html>
<body>
    <form method="post" action="Home/formsubmitted">
        Name:
        <input type="text" name="txt_name" />
        <br />
        Address:
        <input type="text" name="txt_add" />
        <br />
        <input type="submit" value="submit" />
    </form>
</body>
</html>
```

View:formsubmitted.cshtml

```
<html>
<body>
    Name:@ViewBag.name
    <br />
    Address:@ViewBag.address
</body>
</html>
```

Cache

- Caching is a technique of storing frequently used data in a temporary storage area.
- Caching improves performance and scalability.
- When we implement caching on data, the copy of data is stored in the temporary storage area. Hence, when the same data is requested next time, it is picked up from temporary storage area, loading it much faster than from the original source.

- ASP.NET Core supports different kinds of caching such as **In-Memory Cache** (will be discussed here), Response Cache etc.
- The In-Memory Cache stores data in the memory of Web Server where a web application is hosted.

Before we work with cache

- We have installed **Microsoft.Extensions.Caching.Memory" package** in our application using NuGet Package
- In-Memory caching is a service so we have to register it in the **ConfigureServices** method of Startup class as below.

```
public void ConfigureServices(IServiceCollection services)
{
     services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
     services.AddMemoryCache();
}
```

- Now, we can create ImemoryCache interface instance in the Controller using constructor dependency injection and use it to store and retrieve cached data.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Caching.Memory;
namespace TempDataDemo.Controllers
{
    public class HomeController : Controller
    {
        IMemoryCache _cache;
        public HomeController(IMemoryCache _cache)
        {
            this._cache = _cache;
        }
    }
}
```

- IMemoryCache interfaces provides following extensions methods
  1. Get():This method is used to get cached data. It takes cache key as a parameter and returns data stored based on this key.
  2. GetOrCreate():If the data exists for a cache key, then this method reads that data and returns. If cache data for that cache key doesn't exist, then it writes data in the cache.
  3. Remove (): This method is used to remove the cache from the memory.
  4. Set(): This method writes data in the cache.
- Cache can be removed automatically also by using **MemoryCacheEntryOptions**
  1. AbsoluteExpiration:- will expire the entry after a set amount of time
  2. SlidingExpiration: – will expire the entry if it hasn't been accessed in a set amount of time.

```
string key = "FirstKey";//Key Name
// Set cache options.
var cacheEntryOptions = new MemoryCacheEntryOptions().
SetSlidingExpiration(TimeSpan.FromMinutes(5)); //Cache expires after 5 minutes of inactivity
_cache.Set(key, "Hello", cacheEntryOptions);//store Hello with in cache with key FirstKey
                                                    with provided option
```

or using AbsoluteExpiratoin

```
_cache.Set(key, "Hello", TimeSpan.FromSeconds(5)); //Cache with key=”FirstKey” and
                                                   value=”Hello” expires after 5 minutes
```

WAP to demonstrate the concept of storing and reading cached data.

Controller: Homecontroller.cs

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Caching.Memory;
using System;
```

```
using System.Collections.Generic;
namespace TempDataDemo.Controllers
{
    public class HomeController : Controller
    {



        IMemoryCache _cache;
         public HomeController(IMemoryCache _cache)
         {
             this._cache = _cache;
         }

         public IActionResult Index()
         {
             List<string> sd = new List<string>();
             if (_cache.Get("cached_student_details") == null)
             {
                 sd = GetStudentDetails();
                 _cache.Set("cached_student_details", sd, TimeSpan.FromSeconds(60));//expires
                                                                                    in 60 seconds
             }
             else
             {
                 sd = (List<string>)_cache.Get("cached_student_details");
             }
             return View(sd);
         }
         public List<string> GetStudentDetails()
         {
             List<string> nm = new List<string>();
             nm.Add("Ram");
             nm.Add("Hari");
             nm.Add("Shiva");
             return (nm);
         }
    }
}
```

View:Index.cshtml

```
@model IEnumerable<string>
<html>
<body>
    @{
        foreach (string nm in Model)
        {
            @nm;
            <br />
        }
    }
</body>
</html>
```

In the above example, when the user start the application by entering URL /Index, first the cache is tested for getting student details, if it is available there then it is returned from the cache otherwise student details are obtained following normal data access operation (database access or file access etc, here for simplicity we have populated them in list) then writing a copy of it to cache with key "cached_student_details" with absolute expiry time of 60 seconds. No any request is made before 60 seconds, it is served via cache after that cache is destroyed automatically.

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.
- Since, 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

- HttpCookie is accessible from Request.Cookies.

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

Writing Cookie

HttpCookie can be written using Response Object.

- CookieOption is available to extend the cookie behavior.

```
String key="Username";
String value="Bhesh";
CookieOptions option = new CookieOptions();
option.Expires = DateTime.Now.AddMinutes(20);//Cookie automatically destroyed after 20
                                                                                minutes
Response.Cookies.Append(key, value, option);
```

Deleting Cookie

Cookies are deleted by using the key name as below

```
HttpContext.Response.Cookies.Delete("abc"); //Deletes Cookie identified with key "abc"
```

WAP to demonstrates the concept of reading and writing cookies.

Controller: HomeController.cs

```
using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
namespace CookieDemo.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            if (HttpContext.Request.Cookies["uname"] == null)
            {
                return View();//if user is information not available then redirect to Index
                                                                                Page
            }
            else
            {
                return (RedirectToAction(nameof(welcome)));//if user information already
                                                        stored redirect to welcome page
            }
```

```
        }
        [HttpPost]
        public IActionResult Submitform(IFormCollection frm)
        {
            string nam = frm["txt_name"].ToString();
            //string nam = HttpContext.Request.Form["txt_name"].ToString();
            CookieOptions opt = new CookieOptions();
            opt.Expires = DateTime.Now.AddSeconds(60);//Cookie Expires in 60 second
            HttpContext.Response.Cookies.Append("uname", nam, opt);//Key=uname
            return (RedirectToAction(nameof(welcome)));
            //return (RedirectToAction("welcome"));
        }
        public IActionResult welcome()
        {
            return View();
        }
    }
}
```

View: Index.cshtml

```
<html>
<body>
    <form method="post" action="Home/submitform">
        Enter Username:<input type="text" name="txt_name" />
        <br />
        <input type="submit" value="submit" />
    </form>
</body>
</html>
```

View: welcome.cshtml

```
<html>
<body>
    Welcome:
    <h1>@Context.Request.Cookies["uname"].ToString()</h1>
</body>
</html>
```

In the above program when the user starts the application by navigating the URL /index, an attempt is made to check if the username is already stored in cookie or not by reading the cookie with key "uname". If already available, the request is redirected to welcome page. But if not stored in cookie the request is redirected to index page where user can enter username. Upon clicking the submit button, this action is handled by "submitform" action method. This method first read the form values of textbox, write it to cookie, set expiry time to 60 second and finally redirect to welcome page. Now, whenever any next request is made to index page before 60 seconds it is redirected to welcome page.

Session Cookie Vs Persistent Cookie

- Upon a successful login, a cookie is issued and this cookie is sent with each request to the server.
- The server uses this cookie to know that the user is already authenticated and logged-in.
- This cookie can either be a session cookie or a persistent cookie.
- **A session cookie** is created and stored within the session instance of the browser. A session cookie does not contain an expiration date and is permanently deleted when the browser window is closed.
- **A persistent cookie** on the other hand is not deleted when the browser window is closed. It usually has an expiry date and deleted on the date of expiry.

Query Strings

- A query string is a part of a uniform resource locator (URL) that assigns values to specified parameters. A query string commonly includes fields added to a base URL by a Web browser or other client application
- Typical URL containing a query string is as follows:

https://abc.com?name=ram

The question mark is used as a separator, and is not part of the query string.

- When we need to send multiple parameters then we need to separate them by some delimiter. Multiple query parameters are separated by the ampersand, "&":

https://abc.com?name=ram&add=pkr

- We can pass a limited amount of data from one request to another by adding it to the query string of the new request.
- 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.

WAP to demonstrate the concept of passing form values via query string.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using Microsoft.AspNetCore.Mvc;
namespace QueryStringDemo.Controllers
{
    public class HomeController : Controller
    {

        public IActionResult Index(string name, string address)
        {
            return Content("Name=" + name + " Address=" + address);
        }
    }
}
```

- In the above program, model binding in MVC maps HTTP request data to the parameters of the controller's action method. The parameter may either be of simple type like integers, strings, double etc. or complex types.
- So if we provide the values for name and address parameters either from form, route values or query strings, they will be automatically bind to the action methods parameter.
- So for a request like

/Home/Index/ram/pkr

Or

/Home/Index?name=ram&address=pkr

The model binding will automatically ram to name and pkr to address in the action methods.

Note: Query String can also be parsed using HttpContext object as blow

```
using Microsoft.AspNetCore.Mvc;
namespace QueryStringDemo.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            string nm = HttpContext.Request.Query["name"].ToString();
            string ad= HttpContext.Request.Query["address"].ToString();
            return Content("Name=" + nm + " Address=" + ad);
        }
    }
}
```

So for a request like

/Home/Index/ram/pkr

Or

/Home/Index?name=ram&address=pkr

The query string values i.e. ram and pkr, are the value, similarly name and address are the key. So we use these keys to parse the query strings in the Index method using HttpContext.

Hidden Fields

- HiddenField, as name implies, is hidden. Sometimes we require some data to be stored on the client side without displaying it on the page.
- This is non visual control in Asp.net Core where we can save the value.
- This is one of the types of client-side state management tools. It stores the value between the roundtrip. We can save data in hidden form fields and send back in the next request.
- Anyone can see HiddenField details by simply viewing the source of document. HiddenFields are not encrypted or protected and can be changed by anyone. So, it is not secured to store any important or confidential data like password and credit card details with this control.
- The following program demonstrates the concept

Model: Student.cs

```
namespace HiddenDemo.Models
{
    public class Student
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Address { get; set; }
    }
}
```

Controller: HomeController.cs

```
using Microsoft.AspNetCore.Mvc;
using HiddenDemo.Models;
namespace HiddenDemo.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Edit()
        {
            Student s = new Student() { Id=1, Name="Bhesh", Address="Pokhara"};
            return View(s);
        }
        [HttpPost]
        public IActionResult submitedit(Student s)
        {
            return Content("Id="+s.Id+" Name="+s.Name+" Address="+s.Address);
        }
    }
}
```

View:edit.cshtml

```
@model HiddenDemo.Models.Student
<html>
<body>
    <form method="post" action="submitedit">
        <input type="hidden" asp-for="Id" />
        Name:
        <input type="text" asp-for="Name" />
        <br />
        Address:
        <input type="text" asp-for="Address" />
        <br />
        <input type="submit" value="Submit" />
    </form>
</body>
</html>
```

When we run the program and navigate to /Home/Edit, it will display edit page which uses Student Model. This page displays student detail named Name and Address in textfield but stores the id in the hidden field. The automatic model binding will map this entire student model to the action method's submitedit() parameter S. So we can access all the field of Student from View via the Action method parameter S including the value of hidden field. Note the general html code for the hidden field look like this when it is inspected.

```
<input type="hidden" id="Id" name="Id" value="1">
```

Assignment:

1. Differentiate between ViewData, ViewBag and TempData

ViewData	ViewBag	TempData
It is Key-Value Dictionary collection	It is a type object	It is Key-Value Dictionary collection
ViewData is a dictionary object and it is property of ControllerBase class	ViewBag is Dynamic property of ControllerBase class.	TempData is a dictionary object and it is property of controllerBase class.
ViewData is Faster than ViewBag	ViewBag is slower than ViewData	NA
ViewData is introduced in MVC 1.0 and available in MVC 1.0 and above	ViewBag is introduced in MVC 3.0 and available in MVC 3.0 and above	TempData is also introduced in MVC1.0 and available in MVC 1.0 and above.
ViewData also works with .net framework 3.5 and above	ViewBag only works with .net framework 4.0 and above	TempData also works with .net framework 3.5 and above
Type Conversion code is required while enumerating	In depth, ViewBag is used dynamic, so there is no need to type conversion while enumerating.	Type Conversion code is required while enumerating
Its value becomes null if redirection has occurred.	Same as ViewData	TempData is used to pass data between two consecutive requests.
It lies only during the current request.	Same as ViewData	TempData only works during the current and subsequent request