UNIT-6

State Management on ASP.NET Core Application

⊛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 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.
HTTP Context →	Server-side app code.
Cache →	Cache Server-side app code.

⊛. Server-side Strategies:

1) **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 carry on 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.

Example: Program to demonstrate 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", "Roshan");
      HttpContext.Session.SetString ("pwd", "123467");
      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>
```

2) TempData: TempData is a 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.

Example:

Controller: HomeController.cs

```
using Microsoft.AspNetCore.Mvc;
namespace TempDataDemo.Controllers
{
  public class HomeController: Controller
  {
    public IActionResult First()
    {
      TempData["uname"] = "Roshan"; //This will continue 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 the value of Tempdata
                       won't be available in the 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>
```

Views: Third.cshtml

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

3) 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: 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());
}
```

⊛ Cache Client-side strategies:

1) **Cookies:** Cookies store data in the user's browser. Browsers send [Imp] cookies with every request and hence their size should be kept to a minimum. 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:

```
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);
```

2) **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.

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

Now we can invoke this method by passing query string parameters:

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

3) Hidden Fields [Imp] 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. Let's add two methods in our WelcomeController:

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

→ The GET version of the SetHiddenValue() 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.