COMP4701 – Fall 2025 Web Application Development

4 - State Management

Dr. Abdullah Al-Hamdani

State Management

- State management determines how you store information over the lifetime of the application
  - How you keep track of the data moving in and out of your application and how you ensure it’s available when needed.
- The most significant difference between programming for the web and programming for the desktop
  - In a traditional Windows application, memory is always available and only one user is considered
  - In web applications, thousands of users can simultaneously run the same application on the same computer
    - State Management helps maintain information across requests in a stateless HTTP environment.

- Different ways to store and retrieve data between requests in ASP.NET Core applications.

Storage approach	Storage mechanism
Query strings	used to retrieve the variable values in the HTTP query string
TempData	Temporary storage across requests, usually cleared once accessed.
Hidden fields	Storing data as hidden fields
Cookies	Small pieces of data stored on the client’s browser, allowing persistence across sessions.
Session	Maintains user-specific data for the duration of a session for one user.
Cache	Stores data on the server temporarily for fast access to enhance performance, one copy for all users but for a specific time.
Application	Data that remains consistent across the application, one copy for accessible to all users (e.g., app-wide configuration).

Transferring Information Between Pages

- Move from one page to another page using C# code
- Response.Redirect("URL");
    - The URL to redirect the client to (e.g., moving to URL page).
    - This must be properly encoded for use in http headers where only ASCII characters are allowed.
- Move to Page2:

```
Response.Redirect("page2");
```

- Move to page 2 and passing parameters (Query String)

```
Response.Redirect("page2?p1=123&p2=abc");
```

1- Query String

- Common approach is to pass information using a query string in the URL

http://www.google.com/search?q=Oman+cities

- Advantages:
  - –Query string is lightweight
  - –Does not make any kind of burden on the server

Disadvantages of Query String

- Information is limited to simple strings, which must contain URL-legal characters
- Information is clearly visible to the user and anyone else who cares an eavesdrop/spy on the Internet
- The user may change query string
- Many browsers impose a limit on the length of a URL, so large amount of information cannot be placed on query string

Use of Query String

- Put a hyperlink with link "newpage?recordID=10"
- Response.Redirect("newpage?recordID=10");
- Response.Redirect("newpage?recordID=10&mode=full");
- Retrieve the value by Request.Query:
  - string id = Request.Query["recordID"];
  - Information is always string
  - Check for null reference
- Information is visible and unencrypted

Query String

Example: In page 1

```
string item = "Mobile";
int price = 123;
Response.Redirect($"Page2?item={item}&price={price}");
```

In Page 2: retrieve the data in OnGet function:

```
public void OnGet()
{  //there is a need to check for null values
      string myItem = Request.Query["item"];
      int myPrice = int.Parse(Request.Query["price"]!);
       //share dat with view page
      ViewData["Message"]=$"The price for {myItem} is {myPrice} OMR";
}
```

2- TempData Collection

- ViewData and ViewBag are used to share data within the same page (within Model, View and Control)
  - Data is removed when moving to another page.
- TempData is used for passing data from one request to the next request.
- TempData stores data until it's read in another request.
  - Data is removed from memory when moving to a third page
  - Keep(String) and Peek(string) methods can be used to examine the data without deletion at the end of the request.
    - TempData.Keep marks all items in the dictionary for retention.

Example – First Page

Example: Creating tempData variables in Page1

```
public class Page1Model : PageModel {
    [TempData]
    public string myItem { get; set; }
    [TempData]
    public int myPrice { get; set; }

  public void OnGet()
   {
        myItem = "Computer";
        myPrice = 123;
        RedirectToPage("Page2");
    }
}
```

Example – second Page

Example: Receiving tempData variables in Page2

The following code displays TempData["myItem"], but at the end of the request, TempData["myItem"] is deleted:

```
@page
@model IndexModel

<h1>Peek Contacts</h1>
@{
    if (TempData["myItem"] != null)
    {
        <h3>Item: @TempData["myItem"] </h3>
    }
    if (TempData["myprice"] != null)
    {
        <h3>price: @TempData["myPrice"] </h3>
    }
}
```

Example – Second Page

Example: Receiving tempData variables in Page2

The following markup is similar to the preceding code, but uses Keep to preserve the data at the end of the request

```
@page
@model IndexModel
<h1>Peek Contacts</h1>
@{  if (TempData["myItem"] != null)
    {
        <h3>Item: @TempData["myItem"] </h3>
    }
    if (TempData["myPrice"] != null)
    {
        <h3>Price: @TempData["myPrice"] </h3>
    }
    TempData.Keep;
}
```

Example – second Page

Example: Receiving tempData variables in Page2

The following markup is similar to the preceding code, but uses Peek to preserve the data at the end of the request

```
@page
@model IndexModel

<h1>Peek Contacts</h1>
@{
    if (TempData.Peek("myItem") != null)
    {
        <h3>Item: @TempData.Peek("myItem") </h3>
    }
    if (TempData.Peek("myPrice") != null)
    {
        <h3>Price: @TempData.Peek("myPrice") </h3>
    }
}
```

Sharing Classes/Collections

- TempData cannot directly store complex objects (like lists or custom classes), because TempData only stores strings or serializable data.

- There is a need to convert objects/collections to strings using JSON serialization.

```
using Newtonsoft.Json;
```

Example

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Page 1 - Storing data

```
using Newtonsoft.Json;

var students = new List<Student>
    {
        new Student { Id = 1, Name = "Ali" },
        new Student { Id = 2, Name = "Muna" }
    };

    TempData["Students"] = JsonConvert.SerializeObject(students);
```

Page 2 - Retrieving data

```
using Newtonsoft.Json;


var students = new List<Student>();
if (TempData["Students"]!=null)    {
       students =  JsonConvert.DeserializeObject<List<Student>>
                                                              (TempData["Students"]!)!;
}
```

- Cookies are small files that are created on the client’s hard drive
- They can be easily used by any page in the application
- They can be retained between visits, which allows for truly long-term storage
- They are limited to simple strings
- They are easily accessible and readable
- Some users disable cookies on their browsers
- Users can manually delete cookies

Writing Cookies information to Customer Computer

```
@{
    string uname = "xyz"; //get data from form
    string pass = "1234567"; //get data from form

    //1- create a new CookieOptions object and set expiration date
    CookieOptions option = new CookieOptions();
    option.Expires = DateTime.Now.AddDays(30);

    //Add information to user cookies
    Response.Cookies.Append("myLoginName", uname, option);
    Response.Cookies.Append("myPass", pass, option);
}
```

Reading Cookies

- To read a cookie, use the indexer of this class to retrieve the HttpCookie object for a given cookie name:

```
var cookie = Request.Cookies["cookieName"];
```

  - If the cookie does not exist, the indexer returns null.

- You can also use the Cookies collection’s Get method to retrieve a cookie:

```
var cookie = Request.Cookies.Get("cookieName");
```

  - If the cookie does not exist, this method returns null as well.

Update a Cookie in ASP.NET Core

- To update a cookie, you will need to retrieve the cookie from the Request object using the following piece of code:

```
var cookie = Request.GetCookies("cookieName");
```

- Write the updated cookie back to the Response object using the SetCookie method,

```
Response.SetCookie(cookie);
```

Deleting a Cookie

1. Use the Delete method of the Cookie object as shown below:

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

OR

2. Use Response object and set the Expires property of the cookie to a date in the past

```
Response.Cookies["cookieName"].Expires =
                                        DateTime.Now.AddDays(-1);
```

Session State

- HTTP is a stateless protocol.
  - This means that a Web server treats each HTTP request for a page as an independent request.
  - The server retains no knowledge of variable values that were used during previous requests.
- Session state allows you to store user data for the duration of a user's session.
  - ASP.NET session state identifies requests from the same browser during a limited time window as a session, and provides a way to persist variable values for the duration of that session.
- The data is stored on the server and is accessible across multiple requests from the same user.

Session State

- Key Features:
  - Data is stored server-side and identified by a session ID stored in a cookie on the client.
  - Ideal for storing user-specific data that needs to persist between requests.
- ASP.NET Core maintains session state by providing a cookie to the client that contains a session ID.
- The cookie session ID:
  - Is sent to the app with each request.
  - Is used by the app to fetch the session data.

Session Tracking

- When the client presents the session ID, ASP.NET Core looks up the corresponding session and retrieves the objects stored previously
- Session ID is sent to the client in two ways:
  - **Using cookies:** store in the client computer during the session.
  - **Using modified URLs:** This allows using session state with clients that don’t support cookies
- Use session state carefully: When a large number of clients connects to the server, performance may decrease, even session information is small

Session Options

- Session uses a cookie to track and identify requests from a single browser.

Option	Description
Cookie	Determines the settings used to create the cookie. Name defaults to SessionDefaults.Cookie.Name (.AspNetCore.Session). Path defaults to SessionDefaults.Cookie.Path (/). HttpOnly defaults to true (Indicates whether a cookie is inaccessible by client-side script) IsEssential defaults to false (Indicates if this cookie is essential for the application to function correctly. If true then consent policy checks may be bypassed).
IdleTimeout	Indicates how long the session can be idle before its contents are abandoned. The default is 20 minutes.
IOTimeout	The maximum amount of time allowed to load a session from the store or to commit it back to the store. The default is 1 minute.

Configuring Session in ASP.Net Core

- Setting Up Session in ASP.NET Core: To use session state, you need to configure it in the **Startup.cs/program.cs** file

```
builder.Services.AddDistributedMemoryCache();
//1- Adding Session to Web Application
builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromMinutes(5);
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
});
```

Configuring Session in ASP.Net Core

- Using the Session in the Application in the **Startup.cs/program.cs** file

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

```
app.UseSession();
```

```
app.MapRazorPages();
```

```
app.Run();
```

Set and Get Session values

- Session state is accessed from a Razor Page PageModel class or MVC Controller class with HttpContext.Session that uses ISession extension with the following methods methods:
  - Get(ISession, String)
  - GetInt32(ISession, String)
  - GetString(ISession, String)
  - SetInt32(ISession, String, Int32)
  - SetString(ISession, String, String)

Example

- In Page 1: set values using

```
HttpContext.Session.SetString("myName", "Sultan");
HttpContext.Session.SetInt32("myAge", 33);
Respone.Redirect("Page2");
```

- In page 2, we can retrieve the data as follows:

```
public void OnGet() {
    if(string.IsNullOrEmpty(HttpContext.Session.GetString("myName"))) {
            string name = HttpContext.Session.GetString("myName");
            int age = HttpContext.Session.GetInt32("myAge").ToString();
            ViewData["message"] = $" User {myName} is {myAge} years old.";
  }
}
```

Sharing Objects and Collections

- Complex types must be serialized by the user using another mechanism, such as JSON.
- Use the following sample code to serialize objects:

```
public static class SessionExtensions {
        public static void Set<T>(this ISession session, string key, T value) {
             session.SetString(key, JsonSerializer.Serialize(value));
         }
         public static T? Get<T>(this ISession session, string key) {
            var value = session.GetString(key);
            return value == null ? default :  JsonSerializer.Deserialize<T>(value);
          }
         }
```

Set and Get Complex Types

- Set the complex types in Page 1:

```
DateTime currentTime = DateTime.Now; // Requires SessionExtensions from sample.
HttpContext.Session.Set<DateTime>("SessionKeyTime", currentTime);
```

- Retrieve information in Page 2:

```
DateTime currentTime;
 if (HttpContext.Session.Get<DateTime>("SessionKeyTime") != null) {
    currentTime = HttpContext.Session.Get<DateTime>("SessionKeyTime");
 }
```

Application State

- Application state allows you to store global objects that can be accessed by any client
- Similar to session state
- Information is hold on the server
- Example: Global counter
- Items in application state never time out
- They last until the application or server is restarted, or the application domain refreshes itself
- Application state isn’t often used

Application State - Example

1- Create a settings class that will store application-wide data to share with all users for the duration of the application

```
    public class AppSettings
    {
        public string ApplicationName { get; set; }
                                = "Application State App";
        public int NoUsers { get; set; } = 0;
    }
```

Application State - Example

2- Register AppSettings class as a singleton service in the program.cs page

```
using WebApplication97.Pages;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorPages();
// Registers application-wide state
builder.Services.AddSingleton<AppSettings>();
```

Application State - Example

3- Adding an instance of the AppSettings class in PageModel class

- Add a property from AppSettings class in the PageModel class and add it as a parameter to the class constructor "appSettings"
- Use another property to maintain the information "Productist" in the page class

```
public class IndexModel : PageModel {
    private readonly ILogger<IndexModel> _logger;
    private readonly AppSettings _appSettings;
    public string ApplicationName { get; private set; }
    public int  noU {  get; private set; }
    public IndexModel(ILogger<IndexModel> logger, AppSettings appSettings)
    {
        _logger = logger;
        _appSettings = appSettings;
    }
```

Application State - Example

4- Use onGet to retrieve and update the date

```
public void OnGet()
{
    //reteive data
    ApplicationName =_appSettings.ApplicationName;
   noU = _appSettings. NoUsers;
   //update data
   _appSettings.NoUsers++;
}
```

5- Display information in the Razor Page

Cache State

- Application state is a useful place to store small amounts of often-used data that does not change from one user tonother
- Similar to Application state, Cache state is used to share data with all users but for specific time.
  - You can set extensive properties like priority and expiration
- Similar to Sessions, Cache state can be expired after specific time.

Example – Cache State

1. Adding an instance of the Cache state in
   - Add a property from IMemoryCache class in the PageModel class and add it as a paramenter to the class constructor "_cache"
   - Use another property to maintain the information "Productist" in the page class

```
public class IndexModel : PageModel
{
    private readonly ILogger<IndexModel> _logger;
    public List<string> ProductList { get; set; }
    public  IMemoryCache _cache;
    public IndexModel(ILogger<IndexModel> logger, IMemoryCache cache)
    {
        _cache = cache;
        _logger = logger;
    }
```

Example – Cache State

2. In the OnGet function, set the values for the collection and share it in the Cache state

```
public void OnGet()
{   // Try to get cached data
    if (!_cache.TryGetValue("ProductList", out List<string> productList)){
        // Simulate fetching data from a data source
        productList = new List<string> {"Product A","Product B","Product C"};
        // Define cache settings
        var cacheOptions = new MemoryCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromMinutes(5));
        // Set the data in cache
        _cache.Set("ProductList", productList, cacheOptions);
      }

     // Set the data to the property for use in Razor Pag
   ProductList = productList;
}
```

Example – Cache State

3- Use property to retrieve data in the razor page

```
<h2>Product List</h2>
<ul>
    @foreach (var product in Model.ProductList)
    {
        <li>@product</li>
    }
</ul>
```

Example – Cache State

5- Updating the collection in the cache

```
if (_cache.TryGetValue("ProductList",out List<string> productList))
{
    productList.Add(productList.Count.ToString()); //adding new element
    // Define cache settings
    var cacheOptions = new MemoryCacheEntryOptions()
                                  .SetSlidingExpiration(TimeSpan.FromMinutes(5));
    //update data in the cache
    _cache.Set("ProductList", productList, cacheOptions);
}
```

6- You can run more than one application at the same time and they will share the same values for the collection from the cache state.