



CST-350 Activity 2 Login and Registration Guide

Table of Contents

<i>CST-350 Activity 2 Login and Registration Guide</i>	1
<i>Introduction</i>	1
<i>Part 1 Login and Registration</i>	3
Add a User Model:.....	6
About Hashing.....	7
Create a Login Model:	8
Create a Login View:	9
About Razor ASP helper tags	13
User Manager Interface:.....	17
UserCollection:	18
Summary of the UserManager:.....	22
Integrate UserCollection with the Controller:.....	23
What You Just Learned: Part 1 - Login and Registration:	28
Deliverables:	29
<i>Part 2 Session Variables:</i>	30
Setting Up Sessions:	32
Register a new user:.....	39
What You Just Learned in Part 2:	45
Check for Understanding.....	46
Answers:	Error! Bookmark not defined.
Deliverables:	48

Introduction

In this lesson, you will build a simple Login and Registration system using ASP.NET Core with an N-Layer Web Application Architecture. This activity will guide you through the process of creating routes, controllers, models, and views to develop a fully functional login page.

Learning Goals

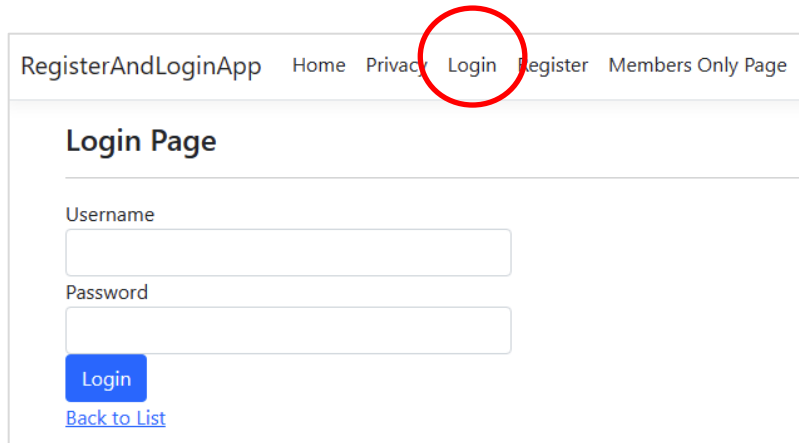
1. **Create and Manage User Accounts:** Build a registration and login system.

2. **Secure Password Storage:** Utilize hashing and salting techniques to securely store user passwords.
3. **Session Handling:** Configure and use session variables to track logged-in users and manage their session data.
4. **Dynamic Content Generation:** Create Razor views based on domain models..

By the end of this activity, you will have developed a login page, registration page, and restricted members-only page, providing a practical understanding of how to build and secure user authentication systems in ASP.NET Core applications.

Here is a preview of the actions we will build.

1. Login page
2. Login success / failure pages
3. Registration page
4. “Members Only” page restricted to users who are logged in.



RegisterAndLoginApp Home Privacy **Login** Register Members Only Page

Login Page

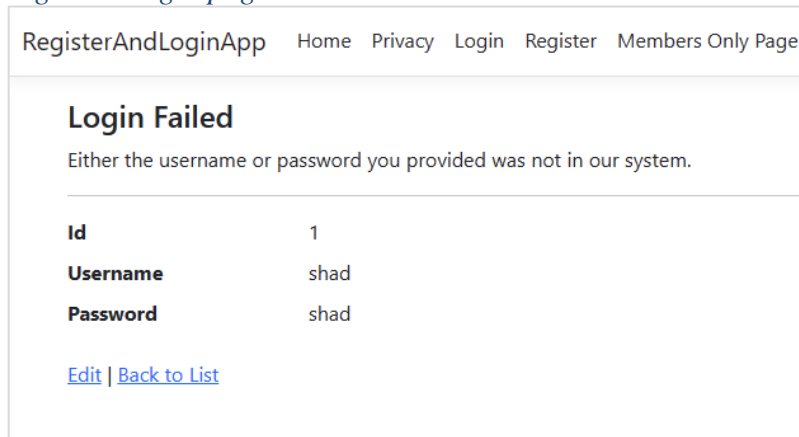
Username

Password

[Login](#)

[Back to List](#)

Figure 1 Login page that we will create in this lesson.



RegisterAndLoginApp Home Privacy Login Register Members Only Page

Login Failed

Either the username or password you provided was not in our system.

Id	1
Username	shad
Password	shad

[Edit](#) | [Back to List](#)

Figure 2 A "Fail" page where the user is told that the username or password is incorrect.

Figure 3 A registration page where the user creates a username and password.

Figure 4 A "Success" page where the newly registered user can login.

Figure 5 A "Members Only" page that shows content restricted to users who have successfully logged in. The status message shows that the user's credentials are saved in a server session variable.

Part 1 Login and Registration

1. Create a new .NET MVC Application:

- a. Select File → New Project. Choose the ASP.NET Web Application. Name your application **RegisterAndLoginApp**.
- b. Click Create.

Configure your new project

ASP.NET Core Web App (Model-View-Controller) C# Linux macOS Windows Cloud Service Web

Project name
RegisterAndLoginApp

Location
C:\Users\shadsluiter\source\repos

Solution name ⓘ
RegisterAndLoginApp

☒ Place solution and project in the same directory

Project will be created in "C:\Users\shadsluiter\source\repos\RegisterAndLoginApp\"

Back Next

Figure 6 Creating a new project.

- c. In the Controllers folder, right-click and choose Add → Controller... menu items.

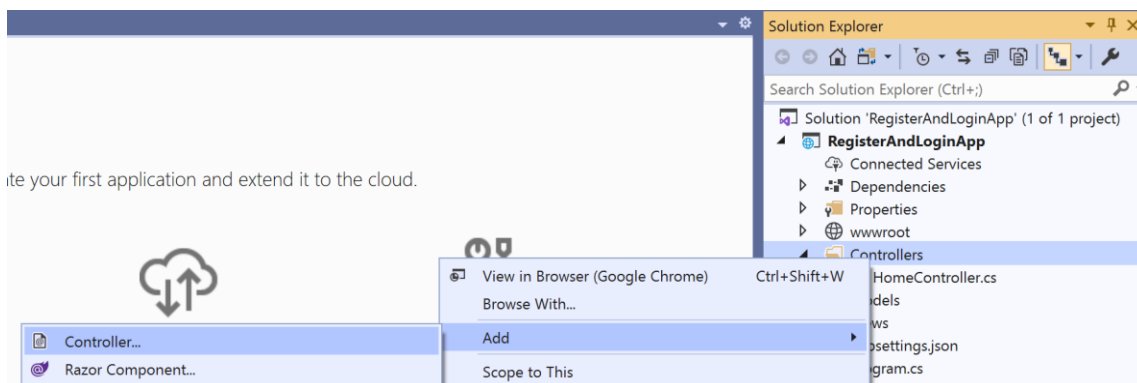


Figure 7 Adding a new controller

- d. Select the 'MVC Controller – Empty' type. Click the Add button. Name your Controller 'Login' and click the Add button.

Add New Scaffolded Item

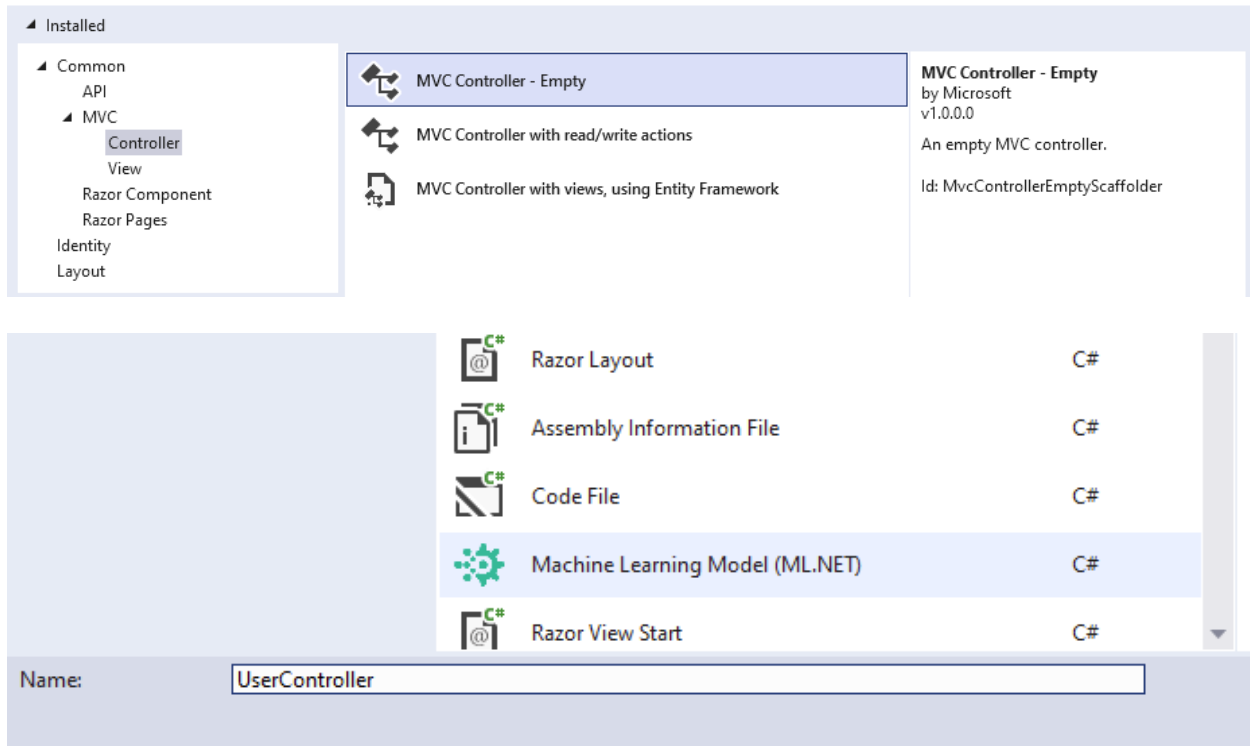


Figure 8 The controller will manage user accounts

- e. You should see the Login Controller file as a `UserController.cs` class in the Controllers folder.

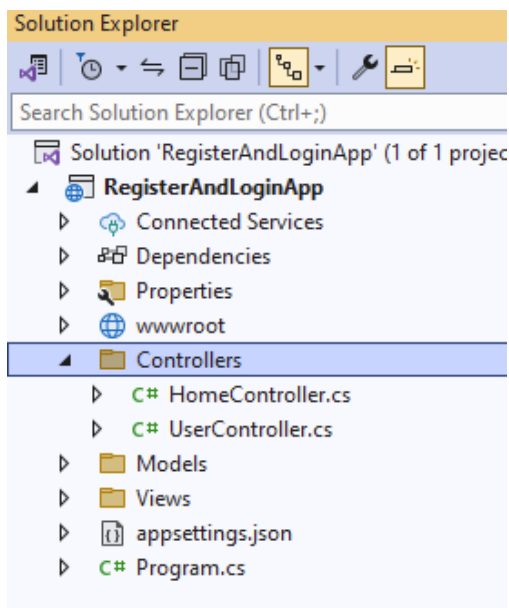


Figure 9 `UserController` is now part of the project.

Add a User Model:

1. Add a new Item to the Models folder. Right-click on the Models folder. Choose Add > New Item.

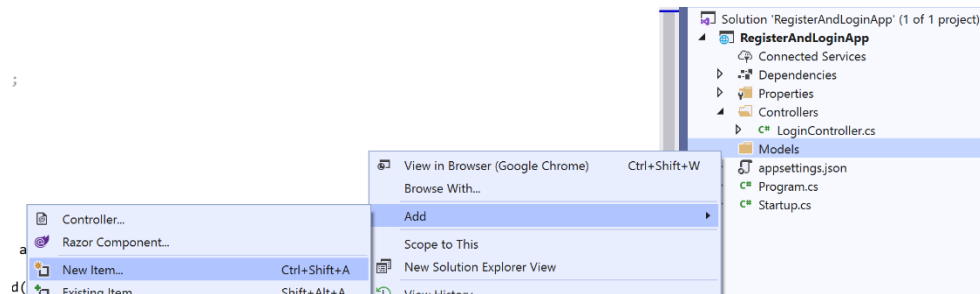


Figure 10 Adding a new item to the Models folder

2. Select Class and enter **UserModel** for the name. Click Add.

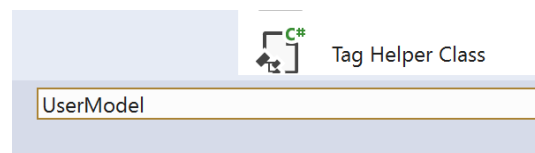


Figure 11 New model is named UserModel

3. Add 3 properties, **Id**, **Username**, and **PasswordHash**, **Salt** and **Groups** to the UserModel class with both setter and getter methods.

```
1  using Microsoft.AspNetCore.Cryptography.KeyDerivation;
2  using System;
3  using System.Security.Cryptography;
4
5  namespace RegisterAndLoginApp.Models
6  {
7      public class UserModel
8      {
9          public int Id { get; set; }
10         public string Username { get; set; }
11         public string PasswordHash { get; set; }
12         public byte[] Salt { get; set; }
13         public string Groups { get; set; }
14     }
15 }
```

```
5  namespace RegisterAndLoginApp.Models
6  {
7      public class UserModel
8      {
9          public string Id { get; set; }
10         public string Username { get; set; }
11         public string PasswordHash { get; set; }
12         public byte[] Salt { get; set; }
13         public string Groups { get; set; }
14     }
15 }
```

Figure 12 UserModel has three properties

About the Model

You have just created a class with the name “Model”. The model is the third part of the MVC triad.

The Model in the context of MVC (Model-View-Controller) represents the application's data and business logic. It is responsible for directly managing the data, logic, and rules of the application. For instance, in a web application that handles products, the Model would be responsible for retrieving product information from a database, updating it, and ensuring it adheres to business rules and validations. The Model acts as the central component that defines how data is stored, retrieved, and manipulated, providing the necessary data to the Controller, which in turn passes it to the View for presentation. This separation ensures that the data logic is isolated from the user interface, promoting a clean and maintainable code structure.

UserModel properties explained

1. **Id** This property serves as the unique identifier for each user in the system.
2. **Username** This property stores the username of the user.
3. **PasswordHash** This property stores the hashed version of the user's password. Instead of storing plain text passwords, which is insecure, the application hashes passwords using a cryptographic algorithm before storing them in the database. This enhances security by making it difficult for attackers to retrieve the original password even if they gain access to the database.
4. **Salt** This property stores the salt used during the password hashing process. A salt is a random value added to the password before hashing. This ensures that even if two users have the same password, their password hashes will be different. It helps in protecting against dictionary attacks and rainbow table attacks.
5. **Groups** This property stores the group memberships of the user in a string format. For example, “**admin, users, students**” This allows the application to manage user roles and permissions effectively. Different groups can have different levels of access to various parts of the application.

About Hashing

Hashing is an important concept in web development when it comes to storing sensitive information like passwords. Instead of saving plain text passwords, which can be easily compromised, applications use hashing to enhance security. Here's what you need to know about hashing:

What is Hashing?

Hashing is the process of converting an input (like a password) into a fixed-length string of characters, which is typically a hash code. This process uses a hash function, which is a one-way function that makes it nearly impossible to revert the hashed output back to the original input.

Why Use Hashing?

Security: Plain text passwords are vulnerable if the database is compromised. Hashing ensures that even if an attacker gains access to the database, they cannot easily retrieve the original passwords.

Irreversibility: Hash functions are designed to be one-way, meaning the hashed output cannot be converted back to the original input.

Uniqueness: Even small changes in the input result in significantly different hash codes, making it hard to guess passwords.

Key Concepts and Terminology

Hash Function: An algorithm that takes an input and produces a fixed-length string, typically a sequence of characters. Common hash functions include SHA-256, bcrypt, and PBKDF2.

Hash Code: The output of a hash function. It's a fixed-length string that represents the original input.

Salt: A random value added to the input of the hash function before hashing. This ensures that even if two users have the same password, their hashes will be different. Salting helps protect against dictionary attacks and rainbow table attacks.

Salting and Hashing: Combining a password with a salt value before applying the hash function. This technique enhances security by ensuring unique hash outputs for identical passwords.

When to Use Hashing

Storing Passwords: Always hash passwords before storing them in the database. This is a standard practice for enhancing security in web applications.

Data Integrity: Hashing can be used to verify data integrity. For example, if you download a file, you can hash it and compare it with the provided hash to ensure the file has not been tampered with.

Authentication: When a user logs in, hash the provided password and compare it with the stored hashed password. If they match, the authentication is successful.

Create a Login Model:

While a user model is useful for storing information about a person, a model that matches the properties of a form is useful for capturing and passing data with a form. Very often this type of model is called a “ViewModel”, emphasizing the fact that it is associated with a view.

1. Create a new class LoginViewModel in the Models folder.

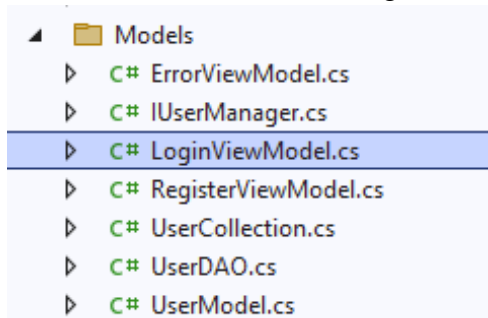


Figure 13 Location of the LoginViewModel.

2. Give the viewmodel two properties `UserName` and `Password`. The `[Required]` attribute is for form validation. It is not necessary for the application to work, but forces the user to fill in all fields in the Login form.

```
1 using System.ComponentModel.DataAnnotations;
2
3 namespace RegisterAndLoginApp.Models
4 {
5     public class LoginViewModel
6     {
7         [Required]
8         public string Username { get; set; }
9         [Required]
10        public string Password { get; set; }
11    }
12 }
13
14
```

Figure 14 LoginViewModel using data annotations to mark all fields as "required"

Create a Login View:

1. Right-click on any part of the code body for the Login Controller `Index()`. Select the **Add View** menu option.

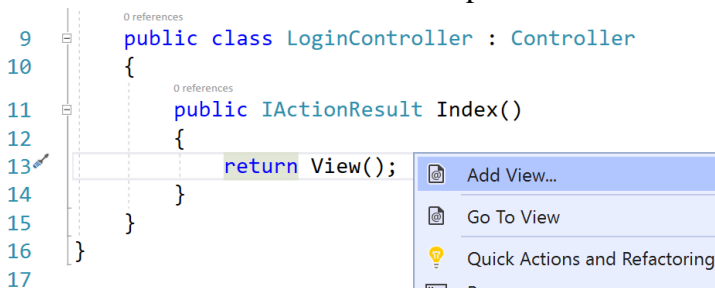


Figure 15 Adding a new view for the Index method.

2. Select **Razor View**.

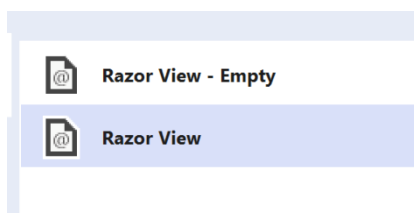


Figure 16 Choosing Razor View

3. Leave the View name as **Index**.
4. Choose the **Create** template.
5. From the **Model** class dropdown, select the **LoginViewModel**.

6. Click Add.

View name: Index

Template: Create

Model class: LoginViewModel (RegisterAndLoginApp.Models)

Options

- ☒ Create as a partial view
- ☒ Reference script libraries
- ☒ Use a layout page

(Leave empty if it is set in a Razor _viewstart file)

Add Cancel

Figure 17 Using "Create" template with LoginViewModel.

7. Open the new file. You should see a bunch of HTML and C# code that is already generated for you.

```

1  @model RegisterAndLoginApp.Models.LoginViewModel
2
3  <h4>LoginViewModel</h4>
4  <hr />
5  <div class="row">
6      <div class="col-md-4">
7          <form asp-action="Index">
8              <div asp-validation-summary="ModelOnly" class="text-danger"></div>
9              <div class="form-group">
10                 <label asp-for="Username" class="control-label"></label>
11                 <input asp-for="Username" class="form-control" />
12                 <span asp-validation-for="Username" class="text-danger"></span>
13             </div>
14             <div class="form-group">
15                 <label asp-for="Password" class="control-label"></label>
16                 <input asp-for="Password" class="form-control" />
17                 <span asp-validation-for="Password" class="text-danger"></span>
18             </div>
19             <div class="form-group">
20                 <input type="submit" value="Create" class="btn btn-primary" />
21             </div>
22         </form>
23     </div>
24 </div>
25
26 <div>
27     <a asp-action="Index">Back to List</a>
28 </div>
29
30 @section Scripts {
31     @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
32 }

```

Figure 18 Razor code for an input form.

8. Test the application to see the new form displayed in the browser.

RegisterAndLoginApp Home Privacy Login f

LoginViewModel

Username

Password

[Create](#)

[Back to List](#)

Figure 19 The user form when rendered in the browser.

9. Modify the page to make it into a **login** form instead of a "Create" form.
 - a. Change the **title**.
 - b. Change the password input to **type password**.
 - c. Change the asp-action to ProcessLogin

d. Change the text of the submit button to **Login**.

```
1  @model RegisterAndLoginApp.Models.LoginViewModel
2
3  <h4>Login</h4>
4  <hr />
5  <div class="row">
6    <div class="col-md-4">
7      <form asp-action="ProcessLogin">
8        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
9        <div class="form-group">
10         <label asp-for="Username" class="control-label"></label>
11         <input asp-for="Username" class="form-control" />
12         <span asp-validation-for="Username" class="text-danger"></span>
13       </div>
14       <div class="form-group">
15         <label asp-for="Password" class="control-label"></label>
16         <input asp-for="Password" type="password" class="form-control" />
17         <span asp-validation-for="Password" class="text-danger"></span>
18       </div>
19       <div class="form-group">
20         <input type="submit" value="Login" class="btn btn-primary" />
21       </div>
22     </form>
23   </div>
24 </div>
25
26 <div>
27   <a asp-action="Index">Back to List</a>
28 </div>
29
30 @section Scripts {
31   @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
32 }
```

Figure 20 Modify the input form to transform it to a login form.

10. Run the application to check the validity of the code you modified.

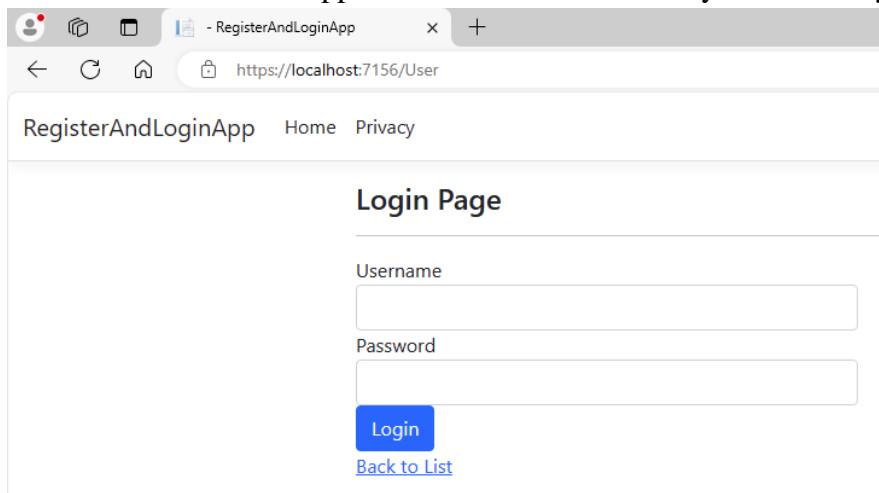


Figure 21 The “Create” form now looks like a Login form after some minor modifications.



- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

About Razor ASP helper tags

You will notice some unusual tags in Razor forms that are not part of HTML nor CSS. In the login page, **asp-action**, **asp-for**, and **asp-validation-for** are tag helpers that instruct the server to render specific HTML output.

You should inspect the rendered HTML code of the Login form to see the resulting HTML properties generated because of the helper tags.

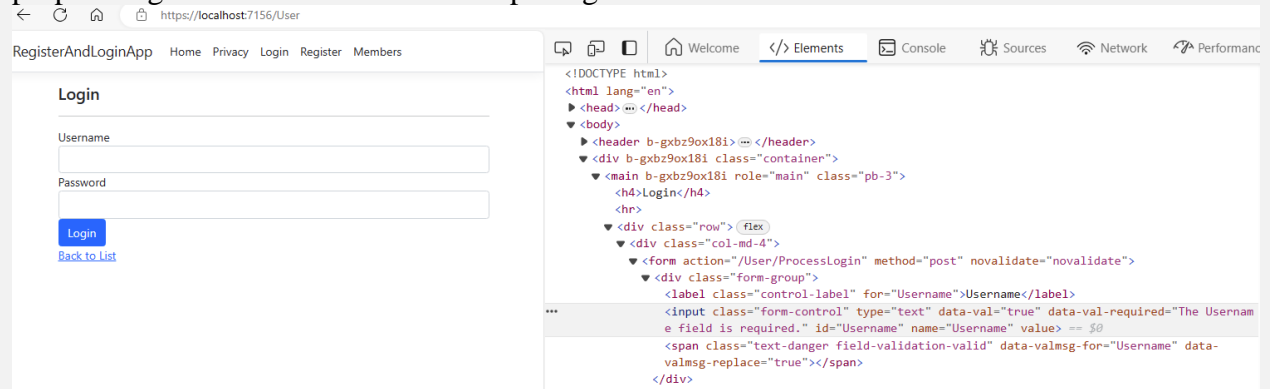


Figure 22 Inspecting the rendered HTML code for the Login view.

asp-action

The **asp-action** tag helper is used to specify the action method that should be called when a form is submitted.

Example:

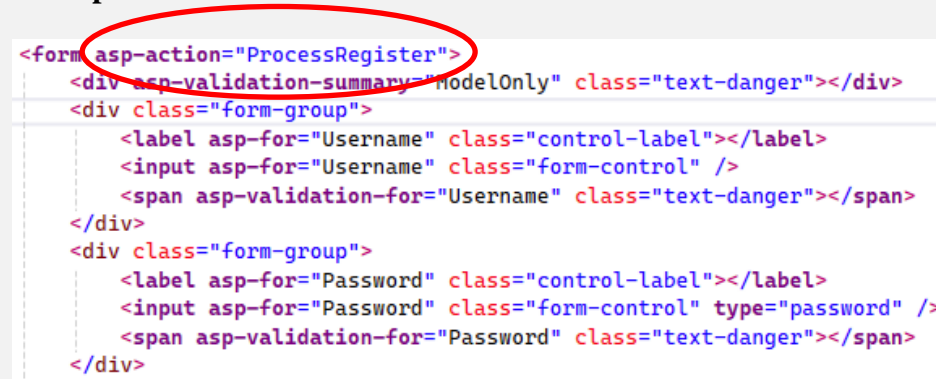


Figure 23 `ProcessRegister` is the method name that this form will be sent to.

asp-action="ProcessRegister" tells the form to submit to the **ProcessRegister** action method of the current controller.

```
<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <header b-gxbz90x18i>...</header>
    <div b-gxbz90x18i class="container">
      <main b-gxbz90x18i role="main" class="pb-3">
        <h4>Login</h4>
        <hr>
        <div class="row">
          <div class="col-md-4">
            <form action="/User/ProcessLogin" method="post" novalidate>
              <div class="form-group">
                <label class="control-label" for="Username">Username
                ...
                <input class="form-control" type="text" data-val="true" data-valmsg="The field is required." id="Username" name="Username" ...>
                <span class="text-danger field-validation-valid" data-valmsg-replace="true"></span>
              </div>
            </form>
          </div>
        </div>
      </main>
    </div>
  </body>
</html>
```

Figure 24 The `asp-action` helper tag renders into a `/User/ProcessLogin` HTML tag.

asp-for

The **asp-for** tag helper is used to bind an HTML element to a model property. It generates the correct **id** and **name** attributes for form fields based on the model property names.

asp-for="Username":

On the **<label>** element: It generates a **for** attribute that matches the **id** of the input field for the **Username** property.

On the **<input>** element: It **binds** the input field to the **Username** property of the model, generating the appropriate **id**, **name**, and **value** attributes.

This binding makes sure that the form fields are correctly associated with the model properties and that data is properly posted back to the server.

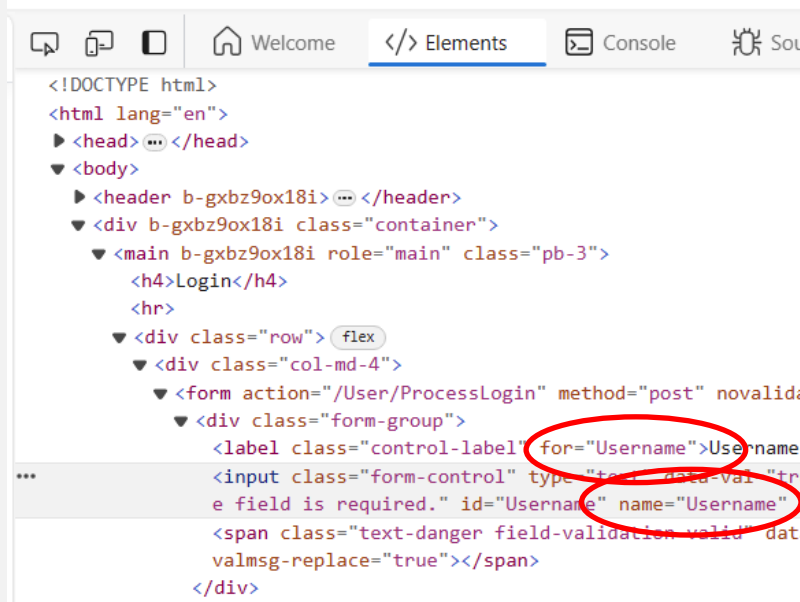


Figure 25 The `asp-for` helper tag renders into a `for="Username"` HTML tag for a label and an input

asp-validation-for

The **asp-validation-for** tag helper is used to display validation messages for a specific model property. It generates the necessary HTML to show validation error messages when the model validation fails.

asp-validation-for="Username":

On the `` element: It binds the validation message to the **Username** property of the model. If there are any validation errors for this property, they will be displayed inside this `` element.

Other Razor Helper Tags

Here are some additional razor tags will appear in `<a href>` links and other code in Razor forms.

asp-controller: Specifies the controller for a request.

asp-route: Specifies route parameters.

asp-route-{value}: Passes custom route values.

asp-area: Specifies an MVC area.

asp-items: Populates options in a `<select>` element.

asp-validation-for: Displays validation messages for a model property.

asp-validation-summary: Displays a summary of validation messages.

asp-antiforgery: Adds an anti-forgery token to a form.

11. Add Menu bar links for convenience. Open the `_Layout.cshtml` file and create a Login and Register link.

```

1  <html lang="en">
2  <head>
3    <meta charset="utf-8" />
4    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
5    <title>@ViewData["Title"] - RegisterAndLoginApp</title>
6    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.min.css" />
7    <link rel="stylesheet" href="/css/site.css" asp-append-version="true" />
8    <link rel="stylesheet" href="/RegisterAndLoginApp.styles.css" asp-append-version="true" />
9  </head>
10 <body>
11 <div class="header">
12 <div class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white border-bottom box-shadow mb-3">
13 <div class="container-fluid">
14 <a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">RegisterAndLoginApp</a>
15 <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target=".navbar-collapse" aria-controls="navbarSupportedContent"
16   aria-expanded="false" aria-label="Toggle navigation">
17   <span class="navbar-toggler-icon"></span>
18 </button>
19 <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
20 <ul class="navbar-nav flex-grow-1">
21 <li class="nav-item">
22 <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
23 </li>
24 <li class="nav-item">
25 <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
26 </li>
27 <li class="nav-item">
28 <a class="nav-link text-dark" asp-area="" asp-controller="User" asp-action="Index">Login</a>
29 </li>
30 <li class="nav-item">
31 <a class="nav-link text-dark" asp-area="" asp-controller="User" asp-action="Register">Register</a>
32 </li>
33 </ul>
34 </div>
35 </div>
36 </div>
37 </div>
38 </div>
39 </div>

```

Figure 26 New links added to the navbar to show Login and Register screens.

12. Run the program. You should be able to see the new links and easily get to the login page.

RegisterAndLoginApp Home Privacy Login Register

Login

Username

Password

Login

[Back to List](#)

Figure 27 Login and Register links are now visible in the navbar.



- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

Expanding the Model Concept

In the context of our application, the **Model** not only represents individual data entities but also encompasses the logic required to manage collections of these entities.

A class to Manage Multiple User Accounts

We will create a new class called **UserCollection** in the Models folder. This class will be responsible for managing a collection of **User** objects.

The **UserCollection** class will include methods for adding, removing, and validating user credentials.

In this version of the UserCollection, the operations will all be performed on a `List<UserModel>` variable. In future versions, you will adapt this design to work with a SQL database.

User Manager Interface:

We will apply some **object-oriented programming design concepts** to this project by creating an **interface** and applying it to two classes. Recall that an interface provides a list of method names that apply to all classes that implement the interface. An interface in object-oriented programming is sometimes called a “**contract**” that classes must fulfil.

In this case, we will create two classes that manage user accounts. We will use the interface to ensure that the method names for the two classes are identical, making them interchangeable within the application.

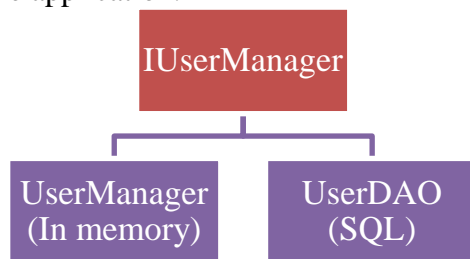


Figure 28 IUserManager will be the pattern for two (or more) classes whose responsibility is to manage the list of known users on this system.

1. Create a new item in the Models folder. Name it **IUserManager.cs**. In C#, it is common convention to begin interface file names with a capital I.

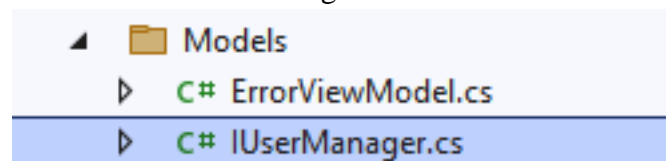


Figure 29 IUserManager file is in the models folder.

2. Specify the following method names GetAllUsers, GetUserById, AddUser, DeleteUser, UpdateUser and CheckCredentials. These are the methods that will allow the application to manage all CRUD operations for user accounts.

```

1 namespace RegisterAndLoginApp.Models
2 {
3     // serves as the pattern for future user manager classes
4     // version 1 = in-memory list of users.
5     // version 2 = sql database
6     // future versions will have the same methods but different underlying storage (Mongo, JSON text file etc)
7
8     public interface IUserManager
9     {
10         public List<UserModel> GetAllUsers(); // return all users stored in the system
11         public UserModel GetUserById(int id); // given id number, find the matching user
12         public int AddUser(UserModel user); // add a new user to the list / db. Use during registration
13         public void DeleteUser(UserModel user); // remove the user who matches
14         public void UpdateUser(UserModel user); // find the user with matching id and replace it
15         public int CheckCredentials(string username, string password); // verify logins
16     }
17 }

```

Figure 30 IUserManager contains seven method names.

UserCollection:

The UserCollection class will manage a list of users. It will implement the methods specified in the IUserManager interface. In the near future you will replace UserCollecion with a SQL-enabled data access object.

1. Create a new class in the Models folder. Name it **UserCollection.cs**

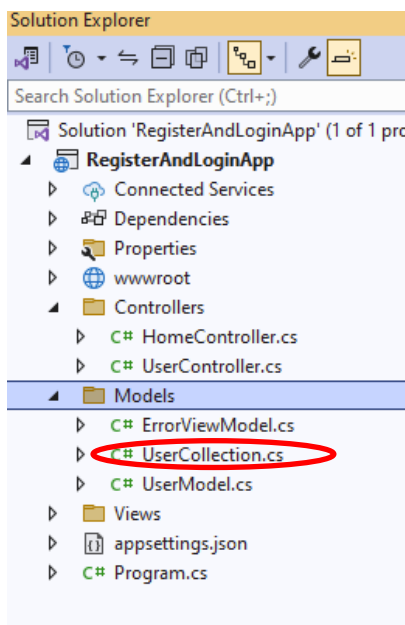


Figure 31 New file "UserCollection" in the Models folder.

2. Implement the interface by adding : IUserManager after the class name.

```

1 namespace RegisterAndLoginApp.Models
2 {
3     public class UserCollection : IUserManager
4     {
5     }
6 }
7

```

Figure 32 UserCollection implements the IUserManager interface but an error is indicated.

3. Hover over the red error text. Right click. Choose Quick fixes. Choose Implement interface.

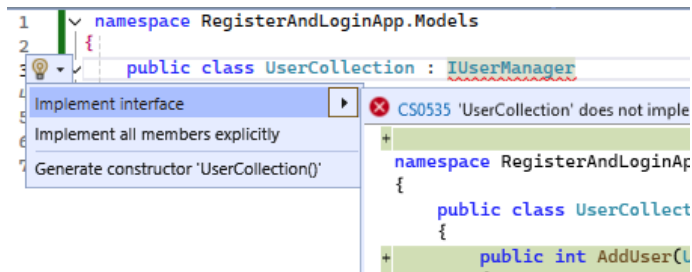


Figure 33 Resolving the interface error. Implementing all the method names contained in the interface.

4. The resulting methods are created but remain to be coded.



Figure 34 Newly created methods have no body inside their curly brackets.

5. Add a private `_users` variable to manage a list of users.
6. In the constructor, assign a new List of users to initialize the `_users` variable.
7. Call a method “GenerateUserData” that will be created in the next step.

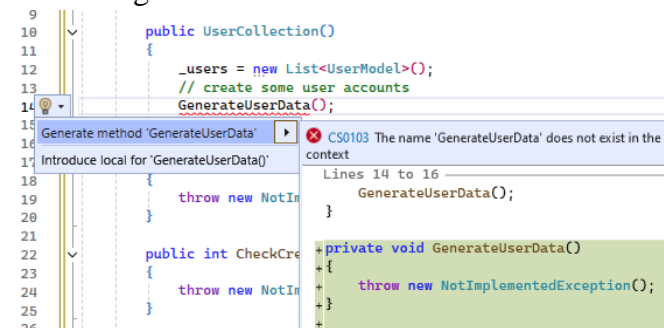
```

1
2 namespace RegisterAndLoginApp.Models
3 {
4     public class UserCollection : IUserManager
5     {
6         // This is an in-memory list of users. In a real application, this would be a database connection.
7         // By convention, the underscore prefix indicates a private field.
8         private List<UserModel> _users;
9
10        public UserCollection()
11        {
12            _users = new List<UserModel>();
13            // create some user accounts
14            GenerateUserData();
15        }
16
17        public int AddUser(UserModel user)
18        {
19
20        }
21    }
22 }

```

Figure 35 `_users` variable and constructor will initialize the set of users.

8. Right-click on the `GenerateUserData` name and generate a new method.



```

9
10 public UserCollection()
11 {
12     _users = new List<UserModel>();
13     // create some user accounts
14     GenerateUserData();
15 }
16
17 private void GenerateUserData()
18 {
19     throw new NotImplementedException();
20 }
21
22 public int CheckCre
23 {
24     throw new NotIn
25 }
26

```

Figure 36 Auto-creating the new method `GenerateUserData`

9. Modify the `AddUser` method to generate an id number automatically for the new user and add him/her to the list.

```

1
2 namespace RegisterAndLoginApp.Models
3 {
4     public class UserCollection : IUserManager
5     {
6         // This is an in-memory list of users. In a real application, this would be a database connection.
7         // By convention, the underscore prefix indicates a private field.
8         private List<UserModel> _users;
9
10        public UserCollection()
11        {
12            _users = new List<UserModel>();
13            // create some user accounts
14            GenerateUserData();
15        }
16
17        private void GenerateUserData()
18        {
19            throw new NotImplementedException();
20        }
21
22        public int AddUser(UserModel user)
23        {
24            // set the user's ID to the next available number
25            user.Id = _users.Count + 1;
26            _users.Add(user);
27            return user.Id;
28        }
29    }
30 }

```

Figure 37 `AddUser` generates a new id number for the new user. It assigns the id number to be the length of the list + 1

10. Create two or more new users in the GenerateUserData method. Use the AddUser method to ensure the new user accounts get a valid id number.

```
10
17 private void GenerateUserData()
18 {
19     UserModel user1 = new UserModel();
20     user1.Username = "Harry";
21     user1.SetPassword("prince");
22     user1.Groups = "Admin";
23     AddUser(user1);
24
25     UserModel user2 = new UserModel();
26     user2.Username = "Megan";
27     user2.SetPassword("princess");
28     user2.Groups = "Admin, User";
29     AddUser(user2);
30 }
```

Figure 38 Generating two new user accounts.

11. Complete the CheckCredentials method. This will be used to verify a login attempt.

```
37
38 public int CheckCredentials(string username, string password)
39 {
40     // given a username and password, find a matching user. return the user's ID.
41     foreach (UserModel user in _users)
42     {
43         if (user.Username == username && user.VerifyPassword(password))
44         {
45             return user.Id;
46         }
47     }
48     // no matches found. invalid login.
49     return 0;
50 }
```

Figure 39 Checking credentials means finding a valid username and matching password.

12. DeleteUser and GetAllUsers are simple list methods.

```
51
52 public void DeleteUser(UserModel user)
53 {
54     _users.Remove(user);
55 }
56
57 public List<UserModel> GetAllUsers()
58 {
59     return _users;
60 }
```

Figure 40 GetAll and DeleteUser simply perform a list operation on the _users list.

13. One approach to find a user id is to use a for loop.

```

61
62 // given an id number. find the user with the matching id
63 public UserModel GetUserById(int id)
64 {
65     // use a for loop to iterate through the list of users
66     foreach (UserModel user in _users)
67     {
68         if (user.Id == id)
69         {
70             return user;
71         }
72     }
73     // if no user is found, return null
74     return null;
75 }

```

Figure 41 For loop used to find a matching user id.

13. You could use a lambda expression to find a matching id using one line of code. Either approach is valid.

```

61
62 // given an id number. find the user with the matching id
63 public UserModel GetUserById(int id)
64 {
65     // using a lambda expression to find the user with the matching ID
66     return _users.Find(u => u.Id == id);
67 }

```

Figure 42 Use a lambda expression on the list to find a matching user id.

14. Use the GetUserById method in the UpdateUser method.

```

77 public void UpdateUser(UserModel user)
78 {
79     // find matching id number
80     UserModel findUser = GetUserById(user.Id);
81
82     // replace the user with the new user
83     if (findUser != null)
84     {
85         int index = _users.IndexOf(findUser);
86         _users[index] = user;
87     }
88 }
89

```

Figure 43 Updating a user means finding the matching user and replacing it.

15. Alternately, you could use a lambda expression and perform the action in one line.

```

77 public void UpdateUser(UserModel user)
78 {
79     // find the user with the matching id and replace it
80     _users[_users.FindIndex(u => u.Id == user.Id)] = user;
81 }

```

Figure 44 Lambda expression that performs the same function as multiple lines of code.

Summary of the UserManager:

In case you missed any of the updates above, here is one version of the UserManager in complete form.

```

2 namespace RegisterAndLoginApp.Models
3 {
4     public class UserCollection : IUserManager
5     {
6         // This is an in-memory list of users. In a real application, this would be a database connection.
7         // By convention, the underscore prefix indicates a private field.
8         private List<UserModel> _users;
9
10        public UserCollection()
11        {
12            _users = new List<UserModel>();
13            // create some user accounts
14            GenerateUserData();
15        }
16
17        private void GenerateUserData()
18        {
19            UserModel user1 = new UserModel();
20            user1.Username = "Harry";
21            user1.SetPassword("prince");
22            user1.Groups = "Admin";
23            AddUser(user1);
24
25            UserModel user2 = new UserModel();
26            user2.Username = "Hogan";
27            user2.SetPassword("princess");
28            user2.Groups = "Admin, User";
29            AddUser(user2);
30        }
31
32        public int AddUser(UserModel user)
33        {
34            // set the user's ID to the next available number
35            user.Id = _users.Count + 1;
36            _users.Add(user);
37            return user.Id;
38        }
39
40        public int CheckCredentials(string username, string password)
41        {
42            // given a username and password, find a matching user. return the user's ID.
43            foreach (UserModel user in _users)
44            {
45                if (user.Username == username && user.VerifyPassword(password))
46                {
47                    return user.Id;
48                }
49            }
50            // no matches found. invalid login
51            return 0;
52        }
53
54        public void DeleteUser(UserModel user)
55        {
56            _users.Remove(user);
57        }
58
59        public List<UserModel> GetAllUsers()
60        {
61            return _users;
62        }
63
64        // given an id number. find the user with the matching id
65        public UserModel GetUserById(int id)
66        {
67            return _users.Find(u => u.Id == id);
68        }
69
70        public void UpdateUser(UserModel user)
71        {
72            // find the user with the matching id and replace it
73            _users[_users.FindIndex(u => u.Id == user.Id)] = user;
74        }
75    }
76 }

```

Figure 45 Complete code for the UserManager class.



Integrate UserCollection with the Controller:

1. Add a static instance of the UserCollection class to the UserController.
2. Add a **ProcessLogin()** method to the **Login** controller that handles the Login View Form Post. This version of the login handler displays the username and

password and validates only one user. In future code, we will authenticate the user with a database connection.

About Dependencies

By instantiating the **UserCollection** class within the controller, we have created a **dependency**. The UserController now **depends on** the UserCollection in order to exist. The UserController is **tightly coupled** to the UserCollection class, which can lead to several issues:

1. Hard to Test: When a class directly instantiates its dependencies, it becomes difficult to test the class in isolation.
2. Reduced Flexibility: Tight coupling makes it harder to switch out the UserCollection for another implementation without modifying the UserController. If you decide to change how users are managed (e.g., switch to a database-backed UserCollection), you will need to change the UserController as well.

Dependency Injection (DI)

In a future lesson, a design pattern called Dependency Injection will be introduced to address these issues. For now, you should be aware that creating this instance of a UserManager will work just fine and seems innocent enough, but is considered a bad practice in software design.

```
6 namespace RegisterAndLoginApp.Controllers
7 {
8     public class UserController : Controller
9     {
10         static UserCollection users = new UserCollection();
11
12         public IActionResult Index()
13         {
14             return View();
15         }
16
17         public IActionResult ProcessLogin(LoginViewModel loginViewModel)
18         {
19             var result = users.CheckCredentials(loginViewModel.Username, loginViewModel.Password);
20
21             if ( result > 0 )
22             {
23                 var user = users.GetUserById(result);
24                 return View("LoginSuccess", user);
25             }
26             else
27             {
28                 return View("LoginFailure");
29             }
30         }
31     }
32 }
```

Figure 46 UserController now has a ProcessLogin method that utilizes a "static" UserCollection class.

The Importance of the "static" keyword in this context

In the UserController code, the UserCollection instance is declared as **static**. This ensures that the UserCollection "remembers" the added users between requests made in the web app.

Explanation of static

In C#, the static keyword is used to declare a *member that belongs to the type itself rather than to a specific object*. This means that static members are shared across all instances of the class and are accessed via the class name rather than an instance.

Shared State Across Requests:

In a web application, each request is typically handled by a new instance of the controller. Without using *static*, each new request would create a new instance of `UserCollection`, resulting in loss of previously added user data.

By declaring `UserCollection` as static, the same instance is shared across all requests and all instances of the `UserController`. This allows the application to "remember" the users added during different requests.

Example Workflow with static

1. Initial Request:

A user submits a registration form. The `UserController` adds the new user to the static `UserCollection` instance.

The user is added to the `Users` list within `UserCollection`.

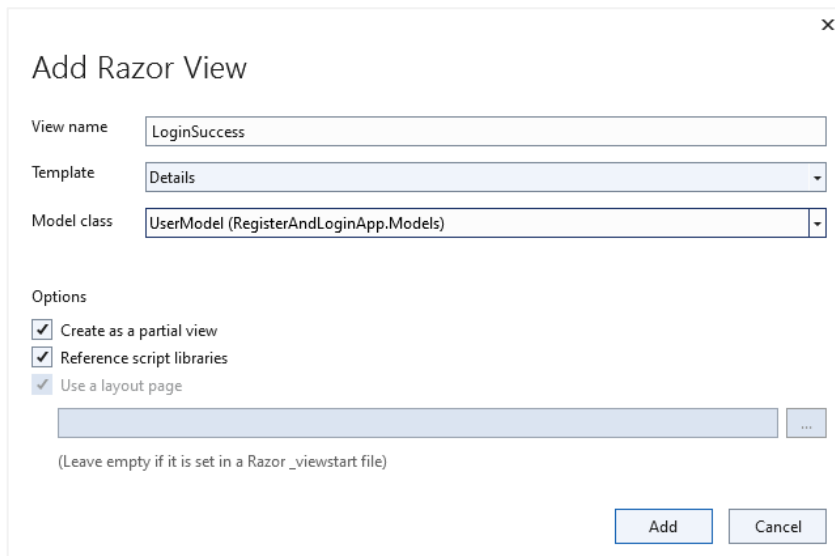
2. Subsequent Request:

Another user attempts to log in. The `UserController` checks the credentials against the same static `UserCollection` instance.

The added users from the previous requests are still present in the `Users` list, allowing the login process to authenticate correctly.

3. Right-click inside the `ProcessLogin` method to create a new View:

- a. Name the view `LoginSuccess`.
- b. Choose the `Details` template.
- c. Select the `UserModel`.



The screenshot shows the 'Add Razor View' dialog box. The 'View name' field contains 'LoginSuccess'. The 'Template' dropdown is set to 'Details'. The 'Model class' dropdown is set to 'UserModel (RegisterAndLoginApp.Models)'. Under the 'Options' section, the following checkboxes are checked: 'Create as a partial view', 'Reference script libraries', and 'Use a layout page'. At the bottom, there are 'Add' and 'Cancel' buttons.

Figure 47 Adding a `LoginSuccess` view

4. Modify the HTML to tell the user that the login was successful.

```

1  @model RegisterAndLoginApp.Models.UserModel
2
3  <div>
4      <h4>Login Success</h4>
5      <hr />
6      <dl class="row">
7          <dt class="col-sm-2">
8              @Html.DisplayNameFor(model => model.Id)
9          </dt>
10         <dd class="col-sm-10">
11             @Html.DisplayFor(model => model.Id)
12         </dd>
13         <dt class="col-sm-2">
14             @Html.DisplayNameFor(model => model.Username)
15         </dt>
16         <dd class="col-sm-10">
17             @Html.DisplayFor(model => model.Username)
18         </dd>
19         <dt class="col-sm-2">
20             @Html.DisplayNameFor(model => model.PasswordHash)
21         </dt>
22         <dd class="col-sm-10">
23             @Html.DisplayFor(model => model.PasswordHash)
24         </dd>
25         <dt class="col-sm-2">
26             @Html.DisplayNameFor(model => model.Salt)
27         </dt>
28         <dd class="col-sm-10">
29             @Html.DisplayFor(model => model.Salt)
30         </dd>
31         <dt class="col-sm-2">
32             @Html.DisplayNameFor(model => model.Groups)
33         </dt>
34         <dd class="col-sm-10">
35             @Html.DisplayFor(model => model.Groups)
36         </dd>
37     </dl>
38 </div>
39 <div>
40     @Html.ActionLink("Edit", "Edit", new { /* id = Model.PrimaryKey */ }) |
41     <a asp-action="Index">Back to List</a>
42 </div>

```

Figure 48 Modified form to show login success message.

5. Run the application. You should be able to login with either “Harry” or “Megan” accounts.

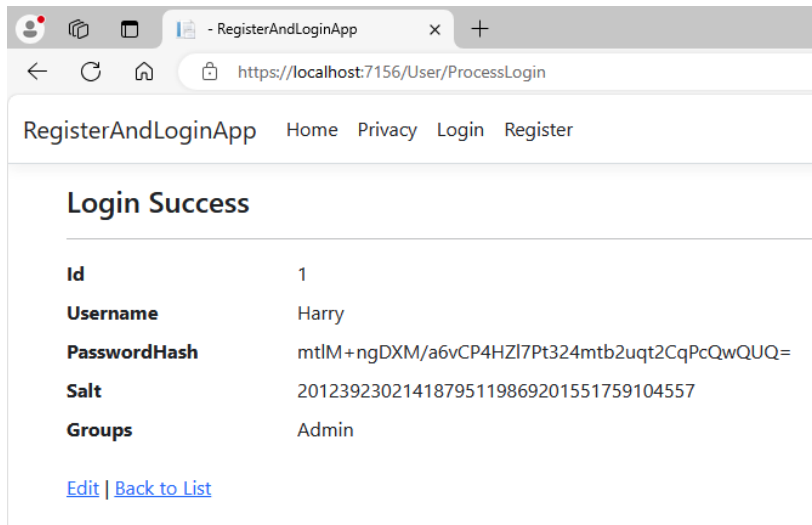


Figure 49 The "Harry" login account successfully logged in. The login success page should show the usermodel properties.



- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

6. Add another View called "LoginFailure" that is used to tell the user that the login was not successful.

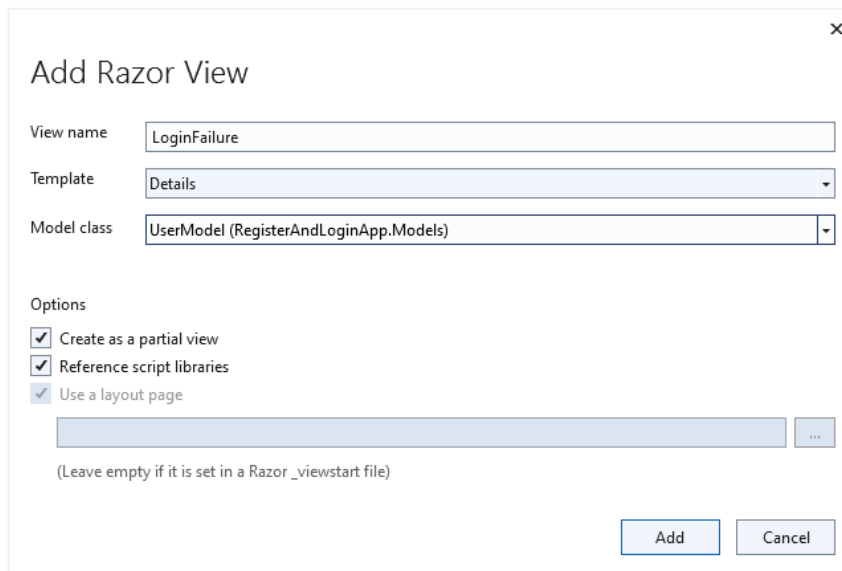


Figure 50 Adding a LoginFailure view

7. Open the LoginFaillure view and modify it to show a failure status message.

```

1  @model RegisterAndLoginApp.Models.UserModel
2
3  <div>
4
5      <h2>Login Failure</h2>
6      <p>Invalid username or password.</p>
7      <hr />
8
9  </div>
10 <div>
11     @Html.ActionLink("Edit", "Edit", new { /* id = Model.PrimaryKey */ }) |
12     <a asp-action="Index">Back to List</a>
13 </div>
14

```

Figure 51 LoginFailure status message.

8. Run the program. You should be able to test the login page with both a success and failure:

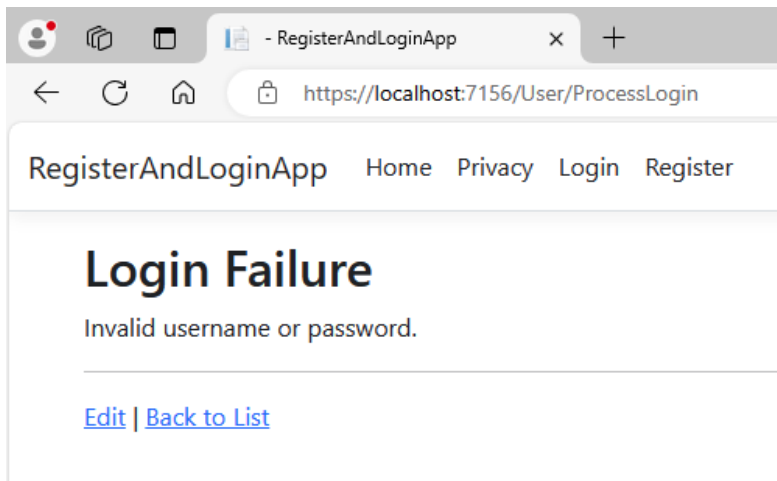


Figure 52 Login failure view is displayed.



- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

What You Just Learned: Part 1 - Login and Registration:

In Part 1 of this activity, you learned how to build a login and registration system in ASP.NET Core using the MVC (Model-View-Controller) pattern. Here are the key concepts and skills covered:

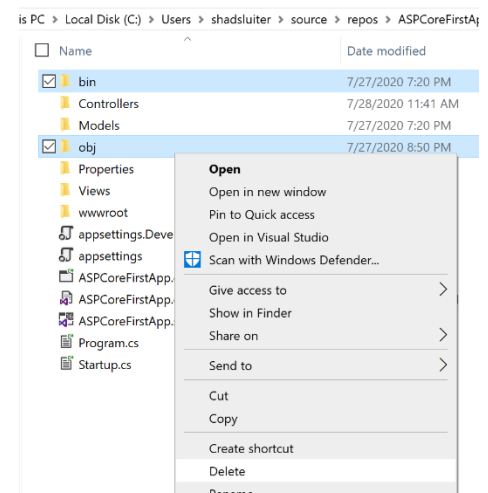
1. **Creating a .NET MVC Application:**
 - You created a new ASP.NET Web Application and set up the initial project structure, including Controllers and Models.
2. **Building User Models:**

- You defined a UserModel class with properties for Id, Username, PasswordHash, Salt, and Groups.
 - You explored the importance of hashing and salting passwords to enhance security.
3. **Developing Login and Registration Views:**
 - You created a LoginViewModel and associated views using Razor syntax.
 - You modified generated Razor views to create a functional login form, including validation using data annotations.
 4. **Using Razor ASP Helper Tags:**
 - You learned about ASP helper tags such as asp-action, asp-for, and asp-validation-for, which simplify form handling and validation.
 5. **Implementing UserCollection:**
 - You created a UserCollection class that implements an IUserManager interface, managing a list of users and providing methods for adding, removing, and validating users.
 6. **Integrating UserCollection with Controllers:**
 - You added a static instance of UserCollection to the UserController to handle user login and registration processes.
 - You created methods in the UserController to process login and registration requests and display success or failure messages.
 7. **Testing and Validating the Application:**
 - You tested the application by running it and verifying that the login and registration functionalities work as expected, including creating new users and authenticating them.
 8. **Understanding Dependencies and Session Management:**
 - You briefly touched upon the importance of managing dependencies and the concept of session variables, setting the stage for further exploration in Part 2.

Here at the end of Part 1, you have a working login and registration system that demonstrates core concepts of user authentication and MVC architecture in ASP.NET Core.

Deliverables:

1. This activity has multiple parts. Complete all parts before submitting.
2. Create a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson. Be sure to explain the key words introduced in this lesson.
4. Turn the Word document into a PDF.



5. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.
6. Attach the PDF separately from the zip file. Multiple files can be uploaded with an assignment.

Part 2 Session Variables:

The login process shown in the previous section is not complete. An HTTP web application is considered “**stateless**”, meaning that each request from the browser is independent of the previous and next requests. Simply because a “Login Success” message was displayed, does not mean that the server will remember this logged in state on the next mouse click. To implement a “remember login” feature we must introduce session management.

Most applications provide security parameters to ensure that only authenticated users can access certain features. For instance, anonymous users cannot post to the application, purchase products, or update any data. Admin users, on the other hand, have privileges to update product data, delete other users, and perform major changes to the system. To manage access to these resources, a web application typically tracks a user’s session. In the following section, we will configure the application to create a session for each user who logs in.

How Session Variables Work:

1. Starting a Session:

When a user logs in, a session is created. The server assigns a unique session ID to the user's session and stores it in a session cookie in the user's browser. The server maintains a list of active session in its memory so there is a physical upper limit to the number of simultaneous users that a single server can support.

This session ID is used to track the user across multiple requests.

2. Storing User Information:

User-specific data, such as username, user ID, or roles, can be stored in session variables. For example, `HttpContext.Session.SetString("User", userJson)` stores the serialized user data in a session variable named "User".

3. Accessing Session Data:

Throughout the user's session, the stored session variables can be accessed to retrieve user-specific information. This is useful for personalizing the user experience, maintaining login status, and controlling access to protected resources.

4. Ending a Session:

The session ends when the user logs out or after a specified period of inactivity. The server clears the session data, and the session cookie is removed from the user's browser.

Alternatives to Session Variables

While session variables are a popular method for maintaining user state, there are other options:

1. Cookies:

Cookies are stored on the client-side and can be used to persist user state across sessions and even after the browser is closed. However, cookies have size limitations and can be less secure if not managed properly.

2. Token-Based Authentication:

JSON Web Tokens (JWT) are commonly used in modern web applications. After a user logs in, a token is issued and sent to the client. The client includes this token in the header of subsequent requests. Tokens are stateless and can be used across different domains and services.

3. Database Storage:

User state can be stored in a database. Each time a user makes a request, the application queries the database to retrieve the user's state. This method is more persistent but can introduce additional latency due to frequent database queries.

4. In-Memory Caching:

Services like Redis or Memcached can store user state in memory. This method is faster than querying a database but can be more complex to set up and manage.

How Session Variables Work in User Login

1. Starting a Session:

When a user logs in, a session is created. The server assigns a unique session ID to the user's session and stores it in a session cookie in the user's browser.

2. Storing User Information:

Upon successful login, user-specific data such as username, user ID, or roles can be stored in session variables. For example, `HttpContext.Session.SetString("User", userJson)` stores the serialized user data in a session variable named "User".

3. Accessing Session Data:

Throughout the user's session, you can access the stored session variables to retrieve user-specific information. This is useful for personalizing the user experience, maintaining login status, and controlling access to protected resources.

4. Ending a Session:

The session ends when the user logs out or after a specified period of inactivity. The server clears the session data, and the session cookie is removed from the user's browser.

Example of Session Variable in User Login

1. User Logs In:

User submits login credentials (username and password).

Server verifies credentials and starts a session.

User information is serialized and stored in a session variable.

```
if (users.CheckCredentials(UserName, password))
{
    // Save the user data in the session
    string userJson = ServiceStack.Text.JsonSerializer.SerializeToString(userData);
    HttpContext.Session.SetString("User", userJson);
    return View("LoginSuccess", userData);
}
else
{
    return View("LoginFailure", userData);
}
```

2. Accessing Session Data:

Protected pages or features check for the session variable to verify the user's login status.

```
public IActionResult Secret()
{
```

```

var userJson = HttpContext.Session.GetString("User");
if (string.IsNullOrEmpty(userJson))
{
    return RedirectToAction("Index", "Home");
}
var user = ServiceStack.Text.JsonSerializer.DeserializeFromString<UserModel>(userJson);
return View("SecretClub", user);
}

```

3. User Logs Out:

The session is cleared, ending the user's session and removing the stored session data.

```

public IActionResult Logout()
{
    HttpContext.Session.Clear();
    return RedirectToAction("Index", "Home");
}

```

Why are sessions not enabled by default?

By default, the web applications created by ASP do not support sessions. We need to configure these in the Program.cs file. Even though it would be convenient for us if sessions were automatically enabled, there are other considerations.

1. Minimal Features and Performance

ASP.NET Core aims to be a lean and high-performance framework. By not including unnecessary features by default, it keeps the framework lightweight and fast.

2. Modularity and Customization

Not all web applications require session management. By making it an optional feature, developers can include it only when their application specifically needs to maintain user state across requests.

There are modern alternatives to session management, such as token-based authentication (e.g., JWT), which do not rely on server-side session state and are more suitable for distributed systems and microservices architectures.

4. Security Considerations

Session management can introduce security risks, such as session hijacking and fixation attacks. By not enabling sessions by default, ASP.NET Core encourages developers to explicitly consider and implement appropriate security measures if they choose to use sessions.

Setting Up Sessions:

1. Add dependency to serialize strings to json. We will save a string value in the session. A JSON string can hold multiple values in one string.

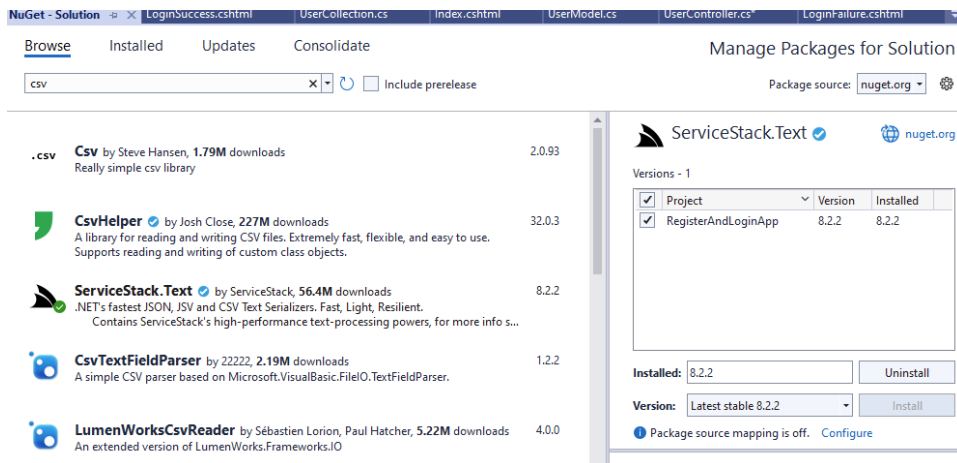


Figure 53 Adding ServiceStack.Text dependency

2. Open Program.cs and add some configurations to allow sessions.

```

2  {
3      public class Program
4      {
5          public static void Main(string[] args)
6          {
7              var builder = WebApplication.CreateBuilder(args);
8
9              // Add services to the container.
10             builder.Services.AddControllersWithViews();
11
12             // Add session services
13             builder.Services.AddDistributedMemoryCache();
14             builder.Services.AddSession(options =>
15             {
16                 options.IdleTimeout = TimeSpan.FromMinutes(30); // Set the session timeout
17                 options.Cookie.HttpOnly = true; // Make the session cookie HTTP only
18                 options.Cookie.IsEssential = true; // Make the session cookie essential
19             });
20             var app = builder.Build();
21
22             // Configure the HTTP request pipeline.
23             if (!app.Environment.IsDevelopment())
24             {
25                 app.UseExceptionHandler("/Home/Error");
26                 // The default HSTS value is 30 days. You may want to change this for production
27                 // aka.ms/aspnetcore-hsts.
28                 app.UseHsts();
29             }
30
31             app.UseHttpsRedirection();
32             app.UseStaticFiles();
33             app.UseRouting();
34             app.UseSession(); // added to enable sessions and keep login state
35             app.UseAuthorization();
36             app.MapControllerRoute(
37                 name: "default",
38                 pattern: "{controller=Home}/{action=Index}/{id?}");
39
40             app.Run();
41         }
42     }

```

Figure 54 Program.cs is the configuration file for an asp.net web app.

3. Create new folder in the project.

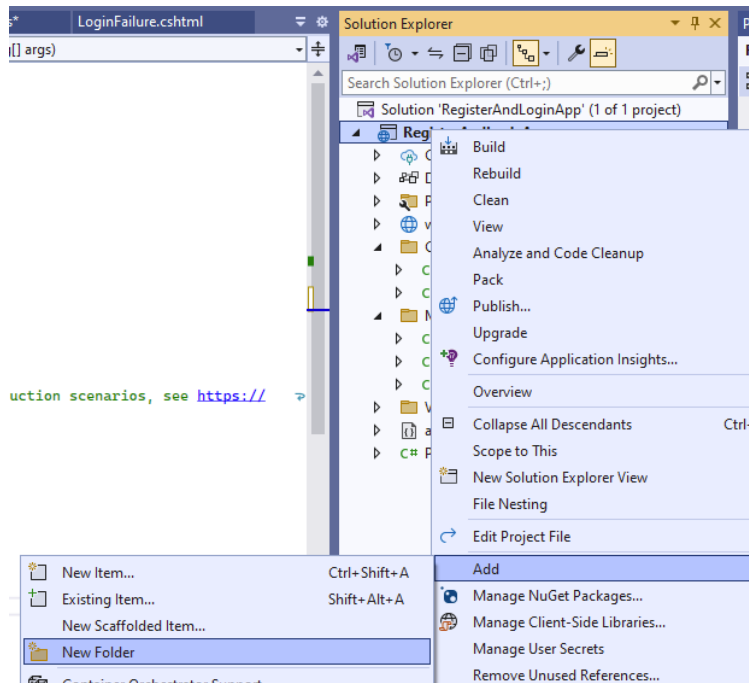


Figure 55 Create new folder in project.

4. Create new file SessionCheckFilter

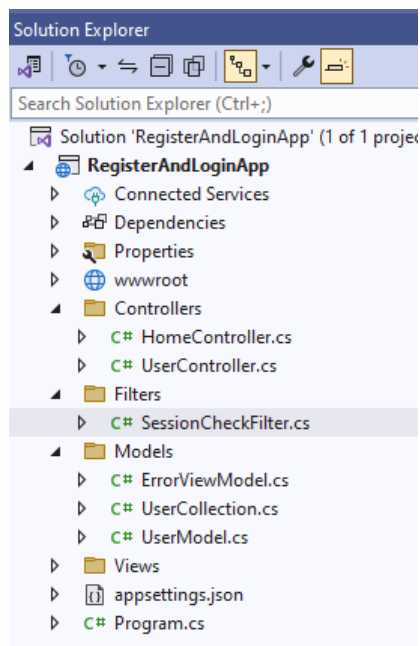


Figure 56 New file in the Filters folder

5. Extend the ActionFilterAttribute class.

```

1  using Microsoft.AspNetCore.Mvc;
2  using Microsoft.AspNetCore.Mvc.Filters;
3
4  namespace RegisterAndLoginApp.Filters
5  {
6      public class SessionCheckFilter: ActionFilterAttribute
7      {
8          public override void OnActionExecuting(ActionExecutingContext context)
9          {
10             if (context.HttpContext.Session.GetString("User") == null)
11             {
12                 context.Result = new RedirectResult("/User/Index");
13             }
14         }
15     }
16 }
17

```

Figure 57 Filter checks to see if session value has been set. If not, the user is directed back to the login page.

6. In the UserController, modify the ProcessLogin method to save the user value as a Json string in the session.

```

public IActionResult ProcessLogin(string UserName, string password)
{
    UserCollection users = new UserCollection();
    users.MakeSomeFakeData();

    UserModel userData = new UserModel { Id = 1, Username = UserName, Password = password };

    if (users.CheckCredentials(UserName, password))
    {
        // save the user data in the session
        string userJson = ServiceStack.Text.JsonSerializer.SerializeToString(userData);
        HttpContext.Session.SetString("User", userJson);
        return View("LoginSuccess", userData);
    }
    else
    {
        return View("LoginFailure", userData);
    }
}

```

Figure 58 UserController and the new methods.

7. Create a new empty view called MembersOnly.cshtml

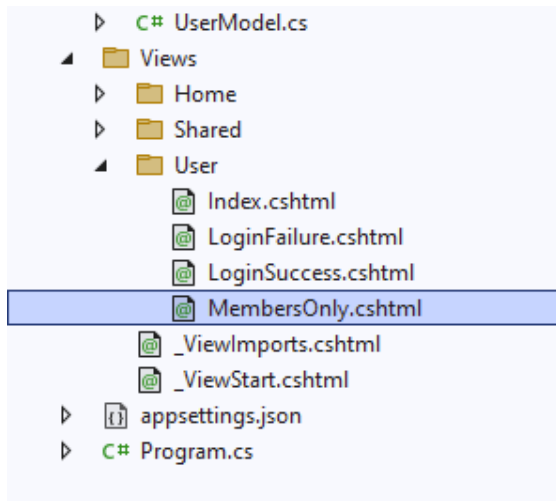


Figure 59 A new page *MembersOnly.cshtml* is in the *Views>User* folder.

8. In this view, verify the user is logged in in the razor code by checking the session value.

```

1  @using ServiceStack.Text
2  @*
3      For more information on enabling MVC for empty projects, visit https://go.microsoft.com/fwlink/?LinkID=397860
4  *@
5  @{
6      <h1>Private Club</h1>
7      <p>Welcome to the private club</p>
8
9      string? user = Context.Session.GetString("User");
10
11      var userString = ServiceStack.Text.JsonSerializer.SerializeToString(user);
12      if (string.IsNullOrEmpty(user))
13      {
14          <p>You are not logged in</p>
15      }
16      else
17      {
18          <p>You are logged in as @userString</p>
19
20          <!-- print formatted json -->
21          <pre>@user</pre>
22      }
23  }
24  }

```

Figure 60 Code on the *MembersOnly* page.

9. Create a method in the *UserController* to attempt to navigate to the *MembersOnly* view. Notice the `[SessionCheckFilter]` annotation has been added above the method. This triggers a request to run the filter.

```

public IActionResult ProcessLogin(string UserName, string password)
{
    UserCollection users = new UserCollection();
    users.MakeSomeFakeData();

    UserModel userData = new UserModel { Id = 1, Username = UserName, Password = password };

    if (users.CheckCredentials(UserName, password))
    {
        // save the user data in the session
        string userJson = ServiceStack.Text.JsonSerializer.SerializeToString(userData);
        HttpContext.Session.SetString("User", userJson);
        return View("LoginSuccess", userData);
    }
    else
    {
        return View("LoginFailure", userData);
    }
}

[SessionCheckFilter]
public IActionResult MembersOnly() {

    return View();
}
}

```

Figure 61 MembersOnly is subject to the [SessionCheckFilter] annotation.

10. Add a Members Only link to the navbar

```

</button>
<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
  <ul class="navbar-nav flex-grow-1">
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
        action="Index">Home</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-
        action="Privacy">Privacy</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="User" asp-
        action="Index">Login</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="User" asp-
        action="Register">Register</a>
    </li>
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="User" asp-
        action="MembersOnly">Members</a>
    </li>
  </ul>
</div>
</div>

```

Figure 62 Members only navbar link

11. Run the program and perform a login.

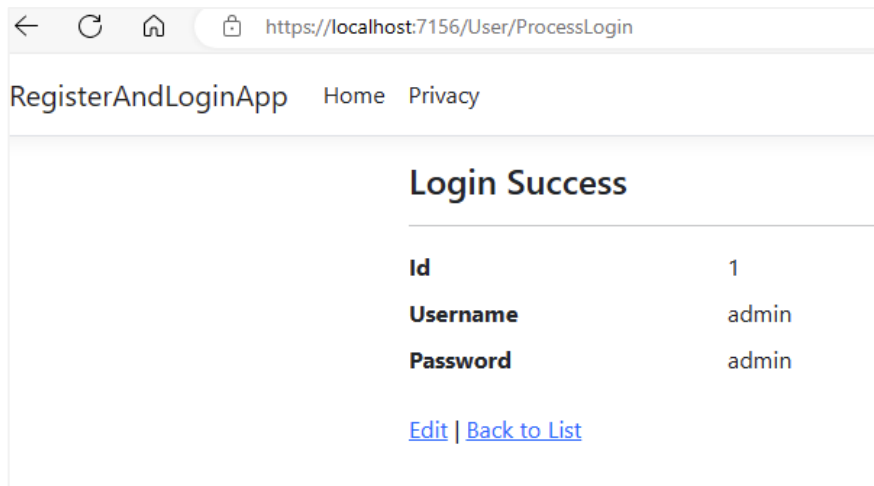


Figure 63 Login was successful with admin account.



- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

12. Navigate to MembersOnly page

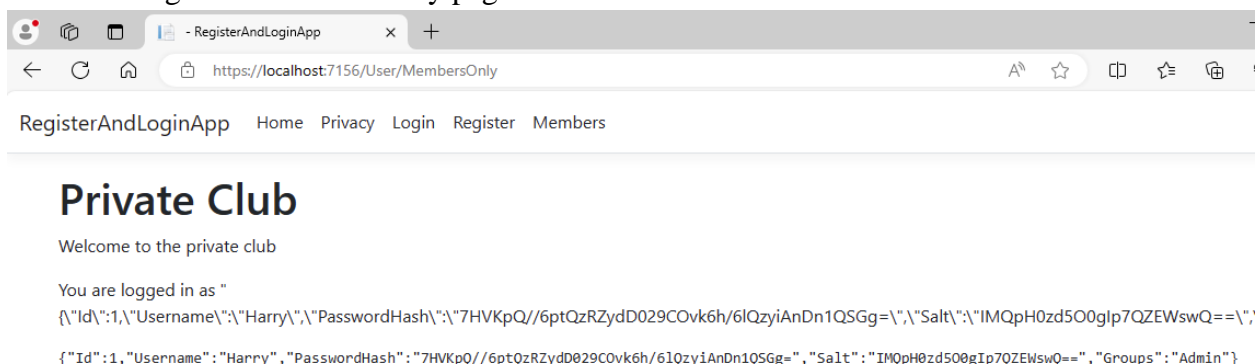


Figure 64 Navigating to the MembersOnly page was successful. The session value is displayed on the status report. JSON text can be displayed as a string or as a formatted JSON object.

13. Add a logout method to the UserController

```

32     }
33
34     [SessionCheckFilter]
35     public IActionResult MembersOnly() {
36
37         return View();
38     }
39
40     [SessionCheckFilter]
41     public IActionResult Logout()
42     {
43         HttpContext.Session.Remove("User");
44         return View("Index");
45     }
46
47 }

```

Figure 65 The Logout method destroys the session variable.

Register a new user:

The goal for this section is to create a registration form, associated viewmodel class to capture the form details and integrate the registration process in with the user controller.

Figure 66 Preview of the Register view.

1. Create a new class RegisterViewModel in the models folder. This class will match the properties of the Register form.

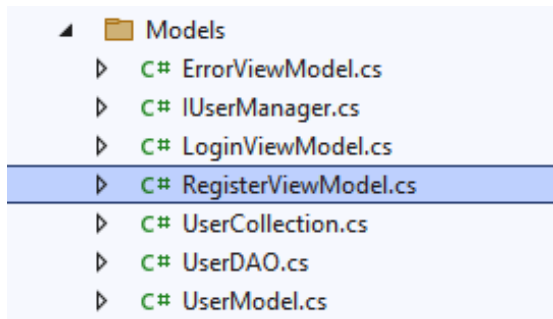


Figure 67 Location of the RegisterViewModel class.

2. Add code in the view model to correspond to two text entry controls and a list of checkbox values.

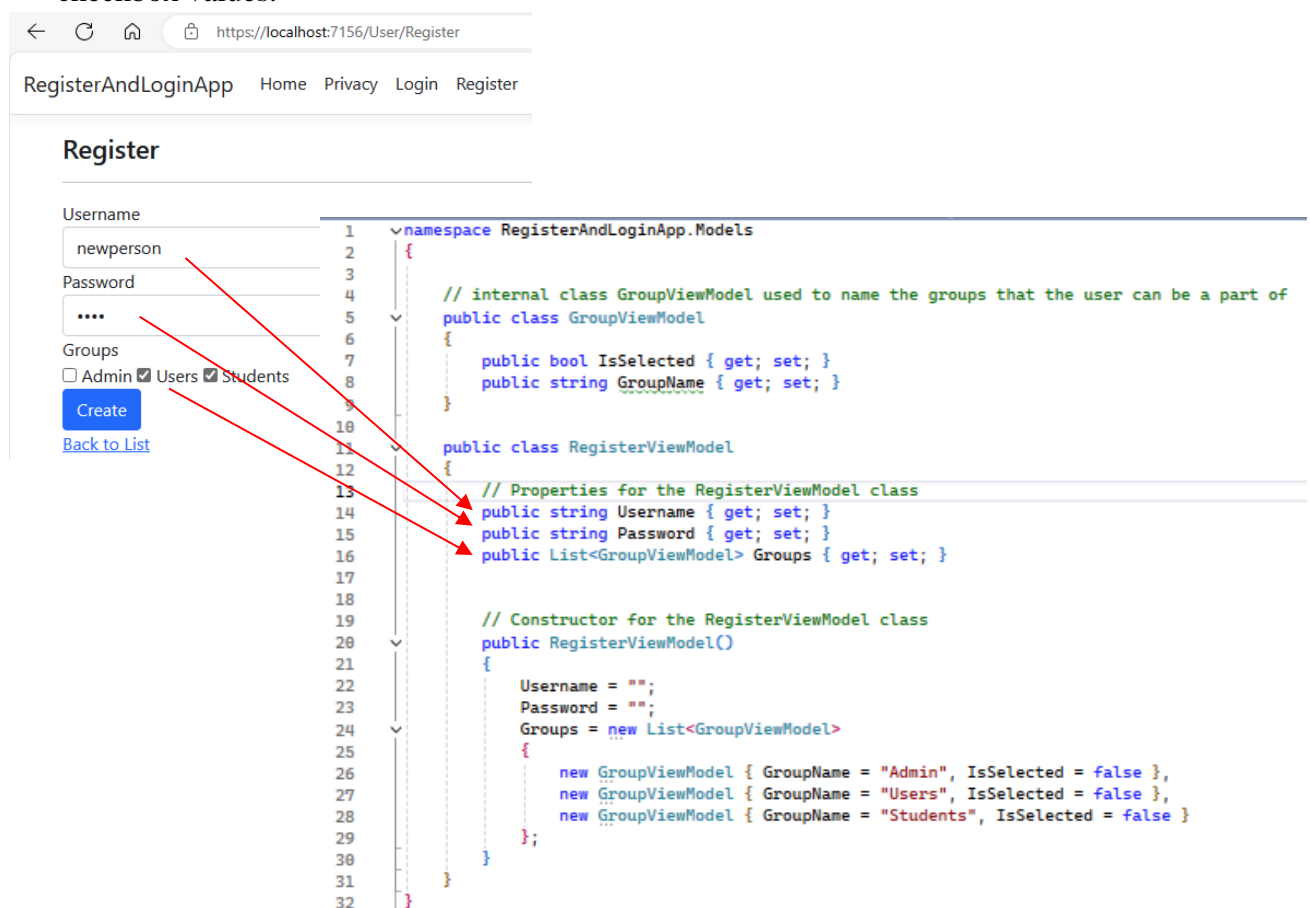


Figure 68 RegisterViewModel class has properties that correspond to the input controls on the Register form.

3. Create a new View for the Registration screen. View is based on the “Create” template for the UserModel class.

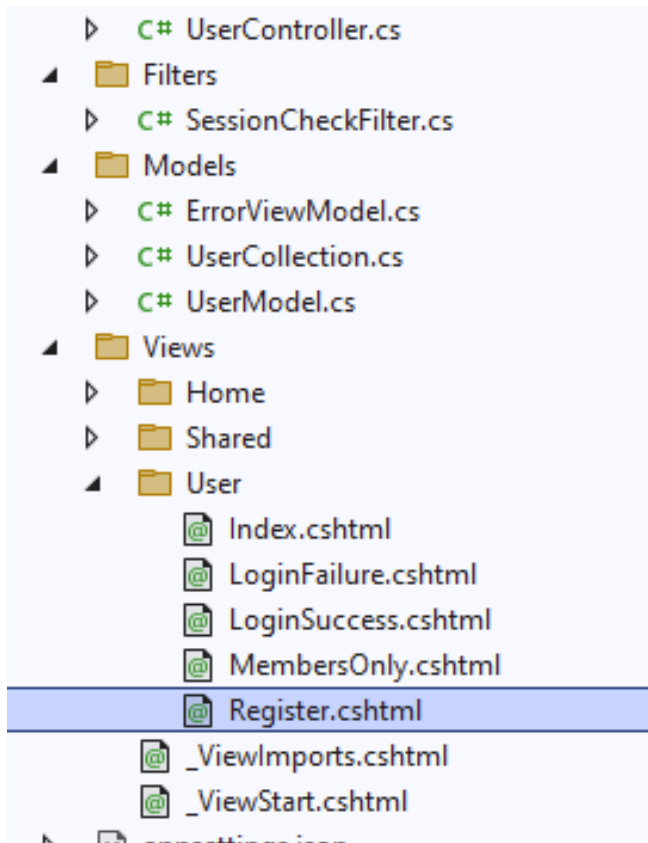


Figure 69 Register view added to the program.

4. Modify the Register form to account for the group membership checkboxes. Using the “hidden” property of an input control allows for the value of the GroupName to be included in the submit request.

```

1  @model RegisterAndLoginApp.Models.RegisterViewModel
2
3  <h4>Register</h4>
4  <hr />
5  <div class="row">
6      <div class="col-md-4">
7          <form asp-action="ProcessRegister">
8              <div asp-validation-summary="ModelOnly" class="text-danger"></div>
9              <div class="form-group">
10                 <label asp-for="Username" class="control-label"></label>
11                 <input asp-for="Username" class="form-control" />
12                 <span asp-validation-for="Username" class="text-danger"></span>
13             </div>
14             <div class="form-group">
15                 <label asp-for="Password" class="control-label"></label>
16                 <input asp-for="Password" class="form-control" type="password" />
17                 <span asp-validation-for="Password" class="text-danger"></span>
18             </div>
19             <div class="form-group">
20                 <label>Groups</label>
21                 <div>
22                     @for (int i = 0; i < Model.Groups.Count; i++)
23                     {
24                         <input type="checkbox" asp-for="Groups[i].IsSelected" />
25                         <input type="hidden" asp-for="Groups[i].GroupName" />
26                         <label>@Model.Groups[i].GroupName</label>
27                     }
28                 </div>
29             </div>
30             <div class="form-group">
31                 <input type="submit" value="Create" class="btn btn-primary" />
32             </div>
33         </form>
34     </div>
35 </div>
36
37 <div>
38     <a asp-action="Index">Back to List</a>
39 </div>
40
41 @section Scripts {
42     @{
43         await Html.RenderPartialAsync("_ValidationScriptsPartial");
44     }
45 }
46

```

Figure 70 Code to capture new user registration. Two modifications to the “Create” form make a register form.

5. Create two new methods in the UserControllers. The first method will display a registration form. The second method will process the new data from a new user.

```

5
6 namespace RegisterAndLoginApp.Controllers
7 {
8     public class UserController : Controller
9     {
10         static UserCollection users = new UserCollection();
11
12         public IActionResult Index()
13         {
14             return View();
15         }
16
17         public IActionResult ProcessLogin(LoginViewModel loginViewModel)
18         {
19             var result = users.CheckCredentials(loginViewModel.Username, loginViewModel.Password);
20
21             if ( result > 0 )
22             {
23                 var user = users.GetUserById(result);
24                 return View("LoginSuccess", user);
25             }
26             else
27             {
28                 return View("LoginFailure");
29             }
30         }
31
32         public IActionResult Register()
33         {
34             return View(new RegisterViewModel());
35         }
36
37         public IActionResult ProcessRegister(RegisterViewModel registerViewModel)
38         {
39             UserModel user = new UserModel();
40             user.Username = registerViewModel.Username;
41             user.SetPassword(registerViewModel.Password);
42             user.Groups = "";
43             foreach (var group in registerViewModel.Groups)
44             {
45                 if (group.IsSelected)
46                 {
47                     user.Groups += group.GroupName + ",";
48                 }
49             }
50             user.Groups = user.Groups.TrimEnd(',');
51             users.AddUser(user);
52
53             return View("Index");
54         }
55     }
56 }
57

```

Figure 71 Two methods in the UserController that handle new user registrations.

6. Add new links to the NavBar to easily navigate to the Login and Register pages.

```

17         aria-expanded="false" aria-label="Toggle navigation">
18         <span class="navbar-toggler-icon"></span>
19     </button>
20     <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
21         <ul class="navbar-nav flex-grow-1">
22             <li class="nav-item">
23                 <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
24             </li>
25             <li class="nav-item">
26                 <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
27             </li>
28             <li class="nav-item">
29                 <a class="nav-link text-dark" asp-area="" asp-controller="User" asp-action="Index">Login</a>
30             </li>
31             <li class="nav-item">
32                 <a class="nav-link text-dark" asp-area="" asp-controller="User" asp-action="Register">Register</a>
33             </li>
34             <li class="nav-item">
35                 <a class="nav-link text-dark" asp-area="" asp-controller="User" asp-action="MembersOnly">MembersOnly
36                 Page</a>
37             </li>
38         </ul>
39     </div>
40 </div>
41 </nav>
42 </header>
43 <div class="container">

```

Figure 72 Layout page now has three new links in the navbar.

7. Run the app to test the process of register, login and access to the members only page.

Figure 73 Register a new user.

8. Login with the new user credentials.

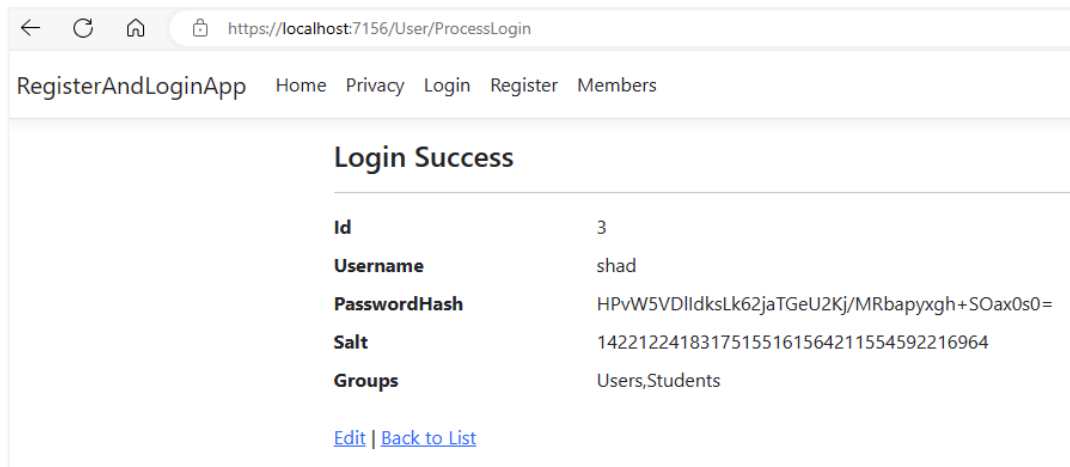


Figure 74 Successfully logged in as new user.

- Run the program. Login with a valid user. Navigate to the restricted page to confirm that a logged in user can see the “Private Club”.

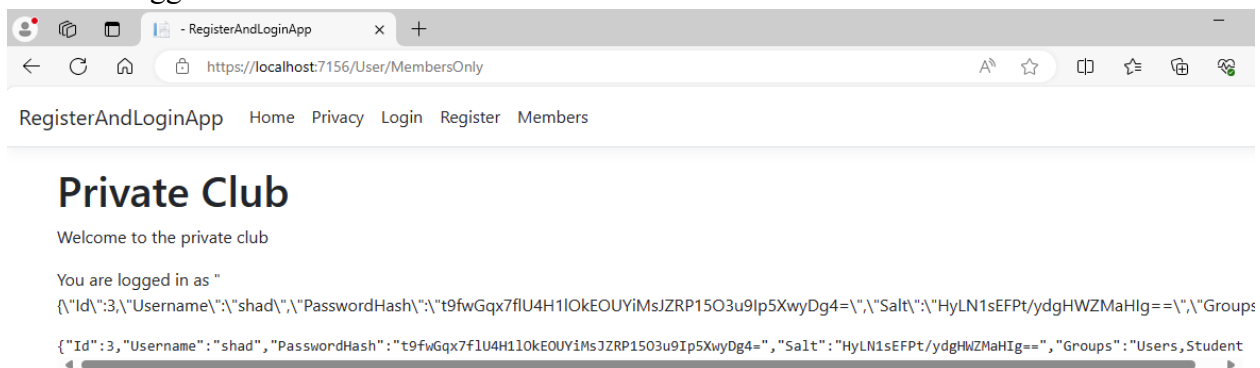


Figure 75 The new user was able to see the MembersOnly page. Session value is shown for testing purposes.



- Take a screenshot of your application running at this point.
- Paste the image into a Word document.
- Put a caption below the image explaining what is being demonstrated.

What You Just Learned in Part 2:

Stateless Nature of HTTP

HTTP is a **stateless protocol**, meaning each request from the client (browser) to the server is independent of previous requests. This means that, without additional mechanisms, the server does not retain information about users between requests.

To implement features like "remember login," web applications need to maintain the state of a user across multiple requests. This is achieved using **session management**.

Session Variables

1. Starting a Session:

When a user logs in, a session is created. The server assigns a unique session ID to the user's session and stores it in a session cookie in the user's browser. The session ID is used to track the user across multiple requests. The server maintains a list of active sessions in its memory, which has a physical upper limit on the number of simultaneous users it can support.

2. Storing User Information:

User-Specific Data: User-specific data such as username, user ID, or roles can be stored in session variables. For example, `HttpContext.Session.SetString("User", userJson)` stores the serialized user data in a session variable named "User."

3. Accessing Session Data:

Retrieving Data: Throughout the user's session, stored session variables can be accessed to retrieve user-specific information. This is useful for personalizing the user experience, maintaining login status, and controlling access to protected resources.

4. Ending a Session:

Session Termination: The session ends when the user logs out or after a specified period of inactivity. The server clears the session data, and the session cookie is removed from the user's browser.

Check for Understanding

These questions are not graded but will help you prepare for upcoming assessments.

1. What is the primary role of controllers in an ASP.NET Core MVC application?

- A. To present data to the user
- B. To manage the flow of data between the application and its users
- C. To handle database operations
- D. To configure application settings

2. How does ASP.NET Core map URL requests to the appropriate controllers and actions?

- A. Through configuration files
- B. Using URL patterns and routing mechanisms
- C. By directly mapping URLs to views
- D. By hardcoding routes in the controllers

3. What is the purpose of scaffolding in an ASP.NET Core MVC application?

- A. To manually write CRUD operations
- B. To automate the creation of basic CRUD operations and views

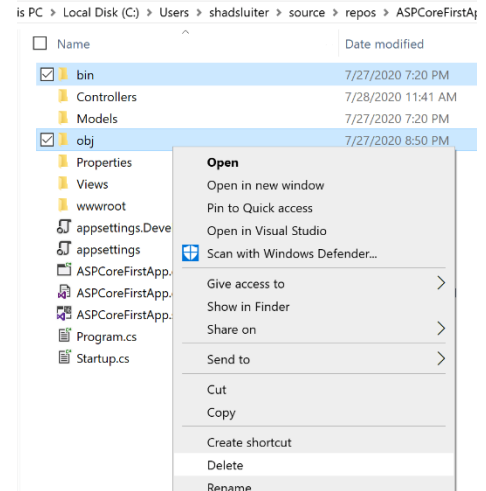
- C. To configure database connections
 - D. To manage user authentication
4. What is the difference between ViewData and ViewBag in ASP.NET Core MVC?
- A. ViewData requires explicit type casting, while ViewBag provides flexibility without type casting
 - B. ViewData is used for session management, while ViewBag is used for caching
 - C. ViewData is slower than ViewBag
 - D. ViewData is only available in controllers, while ViewBag is only available in views
5. How are Razor views used in an ASP.NET Core MVC application?
- A. To configure application settings
 - B. To present data and handle user interactions by embedding server-side C# code within HTML
 - C. To manage database operations
 - D. To handle URL routing
6. Which action result type can be used to return data in JSON format from a controller?
- A. ViewResult
 - B. JsonResult
 - C. ContentResult
 - D. FileResult
7. What is the main purpose of user authentication in an ASP.NET Core MVC application?
- A. To manage application settings
 - B. To secure the application by verifying user identities
 - C. To handle database transactions
 - D. To present data in a user-friendly format
8. What is the role of session variables in an ASP.NET Core MVC application?
- A. To store global application settings
 - B. To manage database connections
 - C. To store user-specific data and maintain login status across multiple requests
 - D. To configure routing mechanisms
9. How is form validation typically implemented in an ASP.NET Core MVC application?
- A. By writing custom validation logic in the controller
 - B. By adding data validation rules within the model class
 - C. By using JavaScript validation only
 - D. By configuring settings in the web.config file

10. Which of the following best describes the use of Razor syntax in an ASP.NET Core MVC application?

- A. Razor syntax is used to manage session variables
- B. Razor syntax is used to embed server-side C# code within HTML for dynamic content generation
- C. Razor syntax is used to configure URL routing
- D. Razor syntax is used to handle database operations

Deliverables:

1. This activity has multiple parts. Complete all parts before submitting.
2. Create a Microsoft Word document with screenshots of the application being run. Show each screen of the output and put a caption under each picture explaining what is being demonstrated.
3. In the same document, in one paragraph, write a summary of the key concepts that were demonstrated in this lesson. Be sure to explain the key words introduced in this lesson.
4. Turn the Word document into a PDF.
5. Submit a ZIP file of the project file. In order to save space, you can delete the bin and the obj folders of the project. These folders contain the compiled version of the application and are automatically regenerated each time the build or run commands are executed.
6. Attach the PDF separately from the zip file. Multiple files can be uploaded with an assignment.



Check for Understanding Answers:

1. Answer: B. To manage the flow of data between the application and its users
2. Answer: B. Using URL patterns and routing mechanisms
3. Answer: B. To automate the creation of basic CRUD operations and views
4. Answer: A. ViewData requires explicit type casting, while ViewBag provides flexibility without type casting
5. Answer: B. To present data and handle user interactions by embedding server-side C# code within HTML
6. Answer: B. JsonResult
7. Answer: B. To secure the application by verifying user identities
8. Answer: C. To store user-specific data and maintain login status across multiple requests
9. Answer: B. By adding data validation rules within the model class
10. Answer B: Razor Syntax is used to embed server side # code within HTML for dynamic content generation.