Module 9: Security

Student Lab Manual

Instructor Edition (Book Title Hidden Style)

Conditions and Terms of Use

Microsoft Confidential

This training package is proprietary and confidential, and is intended only for uses described in the training materials. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or software included in such packages is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Training package content, including URLs and other Internet Web site references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Copyright and Trademarks

Lab 9: Security

Introduction

In this lab, we are going to get back to adding functionality to MyShuttle, by implementing authentication. In order to do this, we will configure an identity provider, add the logic behind the Login and Register pages that we created in Module 7, and modify the main page to show which, if any, carrier is logged in.

Objectives

After completing this lab, you will be able to:

- Allow your web app to consume an identity provider for user authentication, and management.
- Complete the flow of user registration and login and query the authenticated user.
- Feedback model state errors to the view following server side validation failure.

Prerequisites (if applicable)

This lab will use the output from **Lab 8**. You can choose either to continue with the solution you ended up with following the previous lab, or start with the solution included in the folder **Labs\Module 09 - Security\Begin**. No additional SDKs or tools are required for this lab.

Scenario

So far, the website has not been able to login any user or authenticate them, and so we have not been able to secure any of our controller methods. In this lab, we will rectify that.

Carriers need to be able to register an account on the system, and then be able to login. Once logged in, their username will be displayed in the toolbar, and they will be given the option to logout.

System Requirements

To complete this lab, you need:

- Microsoft Visual Studio 2019 or higher
- Microsoft SQL Server (any edition)

Estimated time to complete this lab

30-60 minutes

Exercise 1: Implement Identity

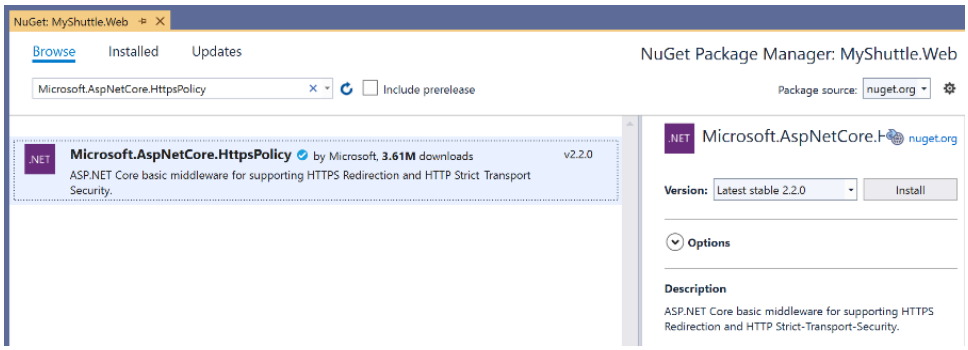
Objectives

In this exercise, you will:

- Complete the lab objectives and implement the scenario mentioned earlier.

Task 1: Configure the App to Use Identity

1. Open the **Startup.cs** file located in MyShuttle.Web's root folder.
2. Let's add HTTPS redirection.
    - Add nuget package **"Microsoft.AspNetCore.HttpsPolicy"**



- Add inside Configure method before "UseStaticFiles" call **"app.UseHttpsRedirection();"**. Should look:

```
0 references | Sychev Igor, 123 days ago | 1 author, 1 change | 0 exceptions
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseHttpsRedirection();
    app.UseStaticFiles();
```

3. Look at how we **already** added (in earlier module) this block of code in the **ConfigureServices()** method, just after the **services.ConfigureDataContext()** call:

```
//Add Identity services to the services container

services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<MyShuttleContext>()
    .AddDefaultTokenProviders();
```

   For the dependencies we added (in earlier module) using statements for MyShuttle.Model (ApplicationUser), and Microsoft.AspNetCore.Identity.EntityFramework (IdentityRole).

   We needed this earlier in the app build because the IdentityProvider is used while seeding the database. We are now going to finish configuring it so we can consume it within our services used in the pipeline.

4. Add in the following UseAuthentication() and UseAuthorization() middleware. Make sure to reorder the middleware pieces as shown below:

```
app.UseStaticFiles();
app.UseAuthentication();
app.ConfigureRoutes();
```

   It is important to get the order right, that UseAuthentication() comes before ConfigureRoutes().

Task 2: Tie Login and Register Methods to Identity

1. Open the **MyShuttle.Web / CarrierController.cs** class in the same project, and locate the **Login()** method that is called for `[HttpPost]` requests.

   Here we will use the sign-in manager to do all the hard work for us: validate the user's password, act on the **remember me** check box and let us know if the login was successful or not.

2. Add the following member **properties** at the top of the CarrierController **class**:

```
public SignInManager<ApplicationUser> SignInManager {get; private set; }
public UserManager<ApplicationUser> UserManager {get; private set; }
```

3. Resolve the dependencies on ApplicationUser by adding a using statement for **MyShuttle.Model**, and one for **Microsoft.AspNetCore.Identity**.

4. Add a constructor that will have instances of the SignInManager and UserManager **injected**:

```
public CarrierController(SignInManager<ApplicationUser> signInManager,
UserManager<ApplicationUser> userManager)
{
    SignInManager = signInManager;
```

```
        UserManager = userManager;
    }
```

5. Replace the `if (ModelState.IsValid)` block of **Login** (**HttpPost**) method with the following: [*Note: Only replace the "if" portion of the if/else, inside the try{ }*]

```
    if (ModelState.IsValid)
    {
        var signInStatus = await SignInManager.PasswordSignInAsync(model.UserName,
model.Password, model.RememberMe, lockoutOnFailure: false);

        if (signInStatus.Succeeded)
        {
            if(string.IsNullOrEmpty(returnUrl))
            {
                return RedirectToAction("Index", "Home");
            }

            return RedirectToLocal(returnUrl);
        }
        else
        {
            ModelState.AddModelError("", "Invalid username or password.");

            return View(model);
        }
    }
```

6. Notice that the error message is added to ModelState above should the sign in fail (in the else-statement): `ModelState.AddModelError("", "Invalid username or password.");`

   This will then appear in the ValidationSummary block of html we added in an earlier lab.

7. Ensure the Logoff method also calls the **SignInManager**. Add the following statement before return statement in the **LogOff()** method.

```
    await SignInManager.SignOutAsync();
```

8. When a **Register** request is **posted**, use the identity provider to create a new user account, sign in with it and then redirect to the relevant action.

   Replace the if block of **Register** (**HttpPost**) method with the following code: [*Note: only replace the "if" portion of the if/else block, in the try{ } block*]

```
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser { UserName = model.UserName, Email =
model.UserName };
        var result = await UserManager.CreateAsync(user, model.Password);

        if (result.Succeeded)
        {
            await SignInManager.SignInAsync(user, isPersistent: false);
```

```
            return RedirectToAction(nameof(HomeController.Index), "Home");
        }
        else
        {
            return RedirectToAction("Register", "Carrier");
        }
    }
```

9. Finally, for this controller, we need to configure it to only allow calls to the relevant functions like Logoff, if there is a user logged in. It is a good practice to ensure that your controller is protected by default. That is, all methods can only be called by an authorized user. This is achieved with the **[Authorize]** attribute, which can be applied to methods or classes.

10. Add **[Authorize]** to the CarrierController class declaration.

    Note that the methods have already been marked up in this controller class appropriately with **[AllowAnonymous]** if users are allowed to call the method without being authenticated. And that the method attributes override the controller attribute.

Task 3: Modify the Toolbar to Display Logged in User

1. Open the **_Layout.cshtml** file in Views/Shared, and **locate** the following markup for the action link to Login:

```
<li>
    @Html.ActionLink("Login", "Login", "Carrier", null, new { @Class = "login-
item" })
</li>
```

2. Replace that section with the following code:

```
@if (User.Identity.IsAuthenticated)
{
    <li>
        <a>@User.Identity.Name!</a>
    </li>

    <li>
        @Html.ActionLink("Log off", "LogOff", "Carrier", null, new { @Class =
"login-item" })
    </li>
}
else
{
    <li>
        @Html.ActionLink("Login", "Login", "Carrier", null, new { @Class =
"login-item" })
    </li>
}
```

2. **Review the code** – you can see that the toolbar will display “LogOff” or “Login”, depending on whether the user is authenticator or not.

It uses the current context's Identity to check whether the user is authenticated or not. If we are, then display a link with the user's name, and another link for the Logoff action. If we are not authenticated, just display the **Login** link.

Task 4: Test the Identity Provider, Register, and Login Functionality

1. Build and run the application.
2. Use a browser to navigate to `http://localhost:xxxx/carrier/register`
3. Enter a username, for example, **myuser123@contoso.com**, a password – ensure the password you choose conforms to the rules below – there is no error handling or display on this page yet, so if something fails, you will be redirected to the same page. If there are no errors, you will be logged in and redirected to the home page.
    - At least 6 characters long
    - Must contain an upper-case character
    - Must contain at least one number
    - Must contain at least one non-alphanumeric – for example, "!", "@", "$" etc.

Task 5: Relax Password Requirement Rules

1. Edit the **ConfigureService()** method in **Startup.cs** file for the **MyShuttle.Web** project by replacing our current **services.AddIdenity()** with the following code.

```
public void ConfigureServices(IServiceCollection services)
{
    services.ConfigureDataContext(Configuration);

    services.AddIdentity<ApplicationUser, IdentityRole>(p =>
    {
        p.Password.RequireDigit = false;
        p.Password.RequireLowercase = false;
        p.Password.RequireUppercase = false;
        p.Password.RequireNonAlphanumeric = false;
    })
    .AddEntityFrameworkStores<MyShuttleContext>()
    .AddDefaultTokenProviders();

    services.ConfigureDependencies();
    services.AddMvc();
}
```

2. **Running the application again, register a different username, and note it will now work with relaxed password rules applied.**