Unit 8

BASIC CONCEPTS ON ASP.NET CORE SECURITY

BASIC CONCEPTS ON ASP.NET CORE SECURITY

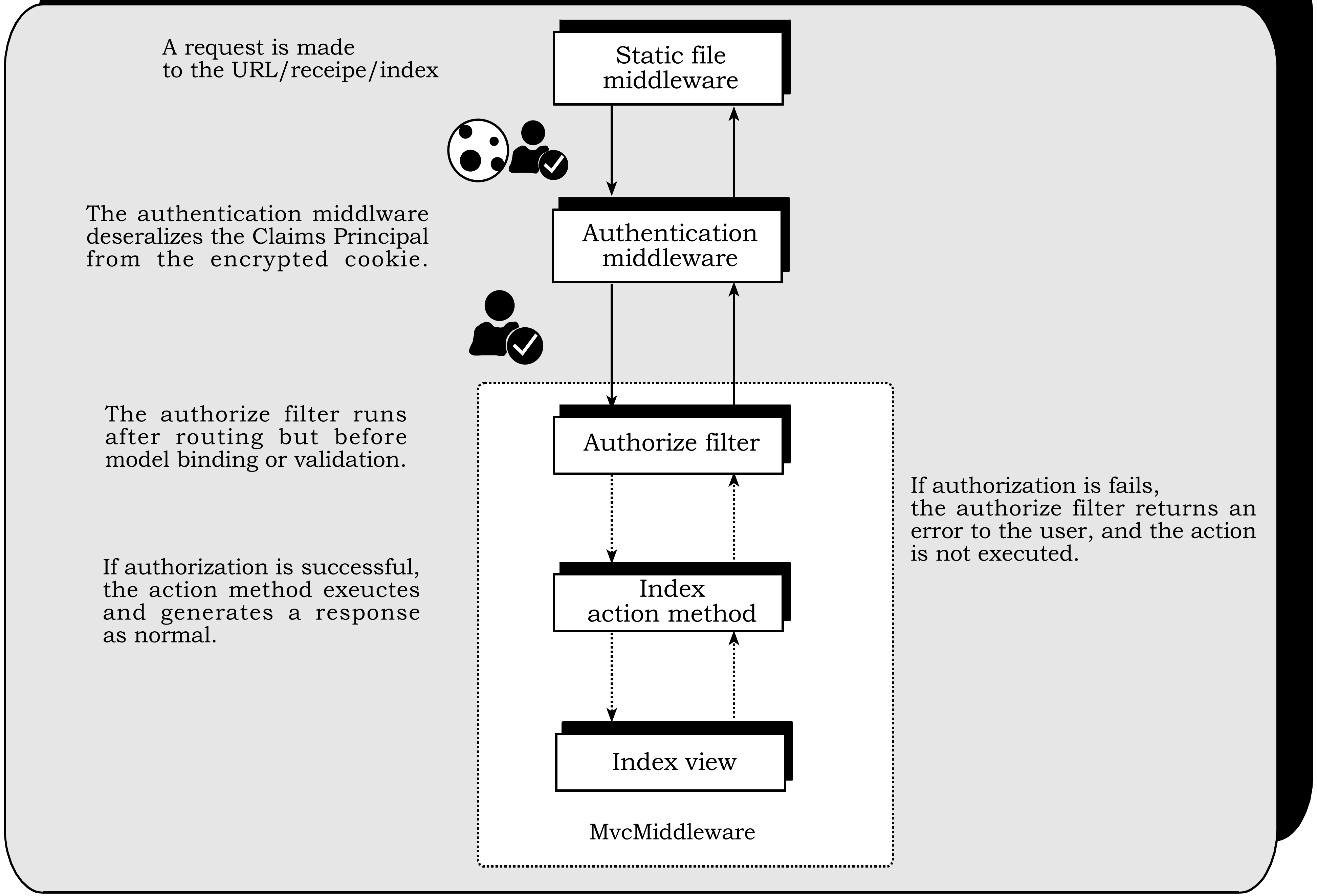
- This Unit shows how to add users to an ASP.NET Core application by adding authentication. With authentication, users can register and log in to your app using an email and password. Whenever you add authentication to an app, you inevitably find you want to be able to restrict what some users can do. The process of determining whether a user can perform a given action on your app is called authorization.
- The two concepts are often used together, but they're definitely distinct:

a. Authentication—The process of determining who made a request

b. Authorization—The process of determining whether the requested action is allowed

Authorization in ASP.NET Core

- The ASP.NET Core framework has authorization built in, so you can use it anywhere in your app, but it's most common to apply authorization as part of MVC. For both traditional web apps and web APIs, users execute actions on your controllers. Authorization occurs before these actions execute, as shown in figure 1. This lets you use different authorization requirements for different action methods. As you can see in figure, authorization occurs as part of MvcMiddleware, after AuthenticationMiddewarehas authenticated the request.
- Authorization, is checking whether a particular user has permission to execute an action. In ASP.NET Core, you'd achieve this by checking whether a user has a particular claim.
- A request is made to the URL /recipe/index. MvcMiddleware.The authentication middleware deserializes the ClaimsPrincipal from the encrypted cookie. The authorize filter runs after routing but before model binding or validation. If authorization is successful, the action method executes and generates a response as normal. If authorization fails, the authorize filter returns an error to the user, and the action is not executed.



A request is made to the URL/receipe/index

The authentication middlware deseralizes the Claims Principal from the encrypted cookie.

The authorize filter runs after routing but before model binding or validation.

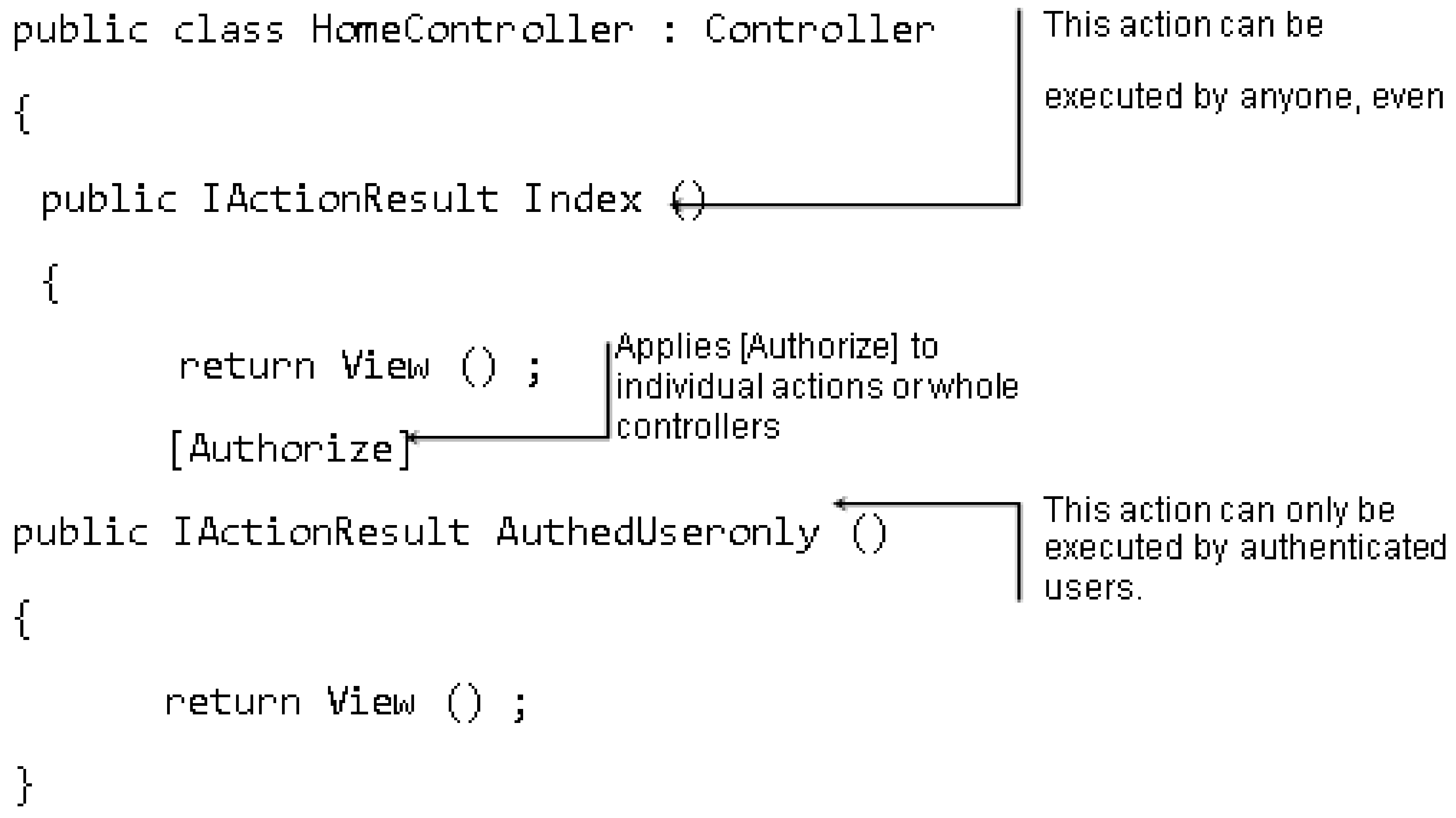
If authorization is successful, the action method exeuctes and generates a response as normal.

If authorization is fails, the authorize filter returns an error to the user, and the action is not executed.

Authorization in ASP.NET Core

- There's an even more basic level of authorization that you haven't considered yet— only allowing authenticated users to execute an action. There are only two possibilities:
  - The user is authenticated- The action executes as normal.
  - The user is unauthenticated - The user can't execute the action.
- You can achieve this basic level of authorization by using the [Authorize] attribute. You can apply this attribute to your actions, to restrict them to authenticated (logged-in) users only. If an unauthenticated user tries to execute an action protected with the [Authorize] attribute in this way, they'll be redirected to the login page.

Authorization in ASP.NET Core



```
public class HomeController : Controller

{

 public IActionResult Index ()

 {

        return View () ;

        [Authorize]

public IActionResult AuthedUseronly  ()

{

        return View () ;

}
```

This action can be

executed by anyone, even

Applies [Authorize] to individual actions or whole controllers

This action can only be executed by authenticated users.

ASP.NET Core Identity

- ASP.NET Core Identity adds user interface (UI) login functionality to ASP.NET Core web apps and manages users, passwords, profile data, roles, claims, tokens, email confirmation, and more.
- Users can create an account with the login information stored in Identity

Create a Web app with authentication

- Create an ASP.NET Core Web Application project with Individual User Accounts.
  - In Visual Studio, Select File > New > Project.
  - Select ASP.NET Core Web Application. Name the project WebApp1 to have the same namespace as the project download. Click OK.
  - Select an ASP.NET Core Web Application, then select Change Authentication.
  - Select Individual User Accounts and click OK.

ASP.NET Core Identity

- The generated project provides ASP.NET Core Identity as a Razor Class Library. The Identity Razor Class Library exposes endpoints with the Identity area.
- For example:
    - /Identity/Account/Login
    - /Identity/Account/Logout
    - /Identity/Account/Manage

Apply migrations

- Apply the migrations to initialize the database and Run the following command in the Package Manager Console (PMC):
- PM> Update-Database

ASP.NET Core Identity

Test Register and Login

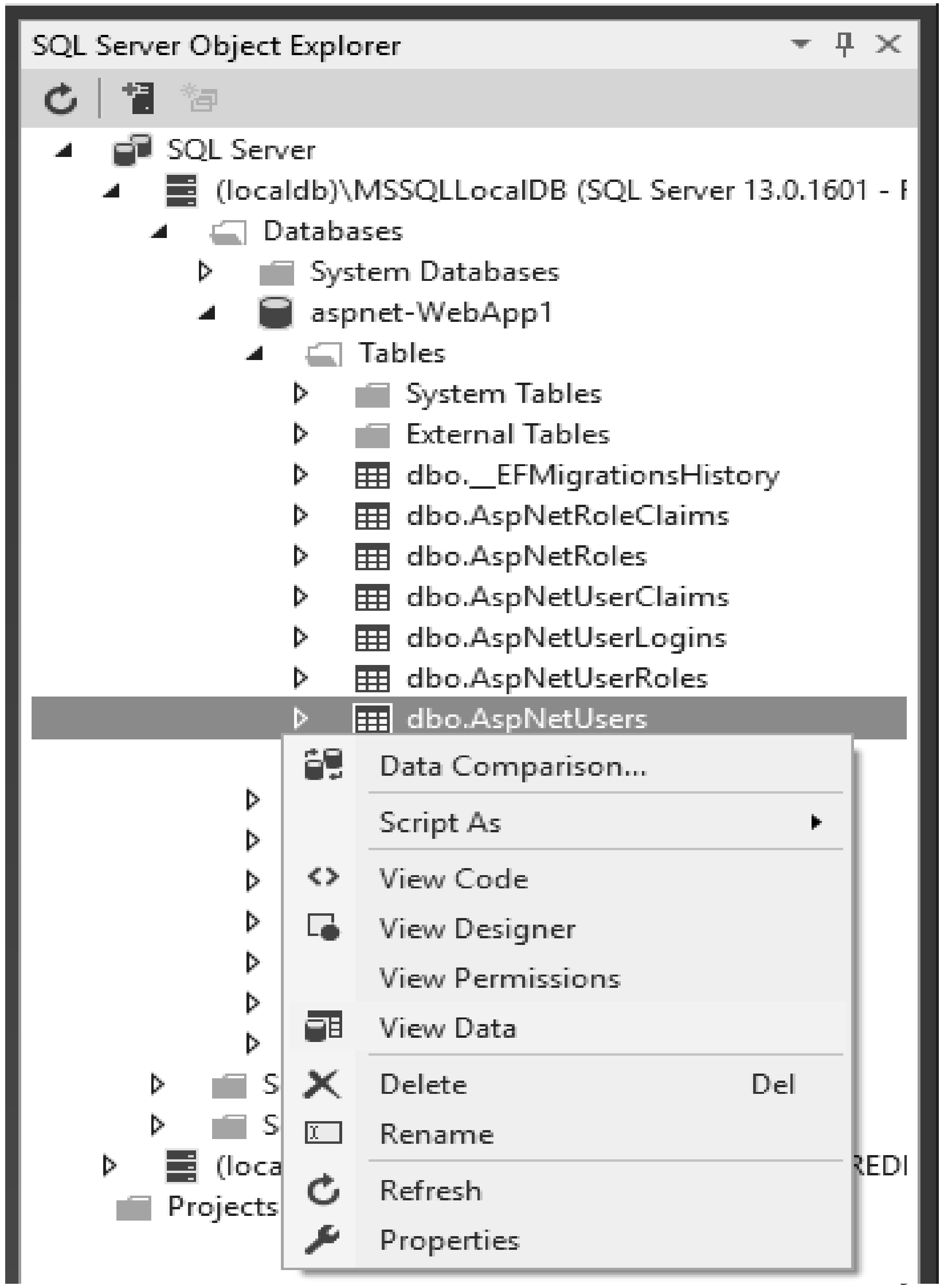
- Run the app and register a user.

Depending on your screen size, you might need to select the navigation toggle button to see the Register andLogin links.

View the Identity database

- From the View menu, select
- SQL Server Object Explorer

Navigate to (localdb)MSSQLLocalDB (SQL Server 13). Right-click on dbo.AspNetUsers > View Data:



ADDING AUTHENTICATION TO APPS AND IDENTITY SERVICE CONFIGURATIONS

- Services are added in ConfigureServices.
- The typical pattern is to call all the Add{Service} methods, and then call all the services.Configure{Service} methods.

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddDbContext<ApplicationDbContext>(options =>
       // options.UseSqlite(
 options.UseSqlServer(
 Configuration.GetConnectionString("DefaultConnection")));
 services.AddDefaultIdentity<IdentityUser>(
    options=>options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();
```

```
services.AddRazorPages();
services.Configure<IdentityOptions>(options =>
    {
        // Password settings.
   options.Password.RequireDigit = true;
   options.Password.RequireLowercase = true;
   options.Password.RequireNonAlphanumeric = true;
   options.Password.RequireUppercase = true;
   options.Password.RequiredLength = 6;
   options.Password.RequiredUniqueChars = 1;

           // Lockout settings.
   options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
   options.Lockout.MaxFailedAccessAttempts = 5;
   options.Lockout.AllowedForNewUsers = true;
```

```
    // User settings.
   options.User.AllowedUserNameCharacters =
           "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
   options.User.RequireUniqueEmail = false;
   });
  services.ConfigureApplicationCookie(options =>
       {
           // Cookie settings
   options.Cookie.HttpOnly = true;
   options.ExpireTimeSpan = TimeSpan.FromMinutes(5);
   options.LoginPath = "/Identity/Account/Login";
   options.AccessDeniedPath = "/Identity/Account/AccessDenied";
   options.SlidingExpiration = true;
      });
 }
```

- The preceding highlighted code configures Identity with default option values. Services are made available to the app through dependency injection.
- The template-generated app doesn't use authorization.
- app.UseAuthorization is included to ensure it's added in the correct order should the app add authorization.
- UseRouting, UseAuthentication, UseAuthorization, and UseEndpoints must be called in the order shown in the preceding code.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
 if (env.IsDevelopment()) {
     app.UseDeveloperExceptionPage();
     app.UseDatabaseErrorPage();
 }
 else {
   app.UseExceptionHandler("/Error");
   app.UseHsts();
 }
 app.UseHttpsRedirection();
 app.UseStaticFiles();
 app.UseRouting();
 app.UseAuthentication();
 app.UseAuthorization();
app.UseEndpoints(endpoints =>
       {
   endpoints.MapRazorPages();
       });
 }
```

AUTHORIZATION: ROLES, CLAIMS AND POLICIES, SECURING CONTROLLERS AND ACTION METHODS

- When an identity is created it may belong to one or more roles. For example, Admin1 may belong to the Administrator and User roles whilst User1 may only belong to the User role. How these roles are created and managed depends on the backing store of the authorization process. Roles are exposed to the developer through the IsInRole method on the ClaimsPrincipal class.

Roles

- Role-based authorization checks are declarative—the developer embeds them within their code, against a controller or an action within a controller, specifying roles which the current user must be a member of to access the requested resource.

Roles

- For example, the following code limits access to any actions on the AdministrationController to users who are a member of the Administrator role:

```
[Authorize(Roles = "Administrator")]
public class AdministrationController : Controller{}
```

- You can specify multiple roles as a comma separated list:

```
[Authorize(Roles = "HRManager,Finance")]
public class Salary Controller : Controller {}
```

- This controller would be only accessible by users who are members of the HRManager role or the Finance role.

Roles

- If you apply multiple attributes then an accessing user must be a member of all the roles specified; the following sample requires that a user must be a member of both the PowerUser and ControlPanelUser role.

```
[Authorize(Roles = "PowerUser")]
[Authorize(Roles = "ControlPanelUser")]
public class ControlPanelController : Controller
{
}
```

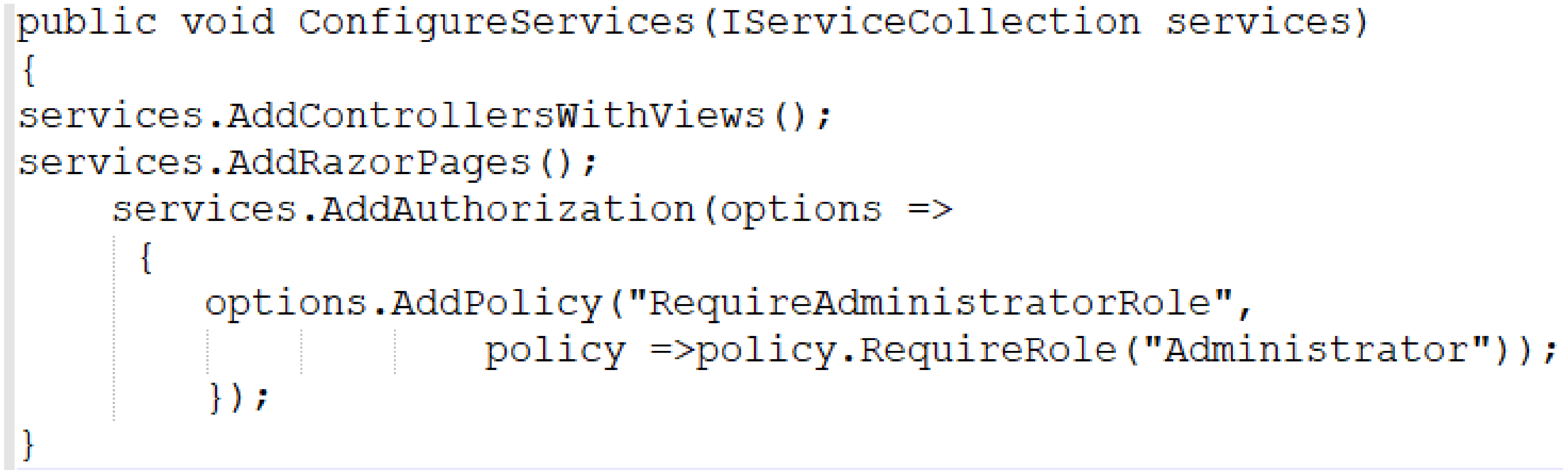
Roles

- You can further limit access by applying additional role authorization attributes at the action level:

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller {
    public ActionResult SetTime()
    {
    }
    [Authorize(Roles = "Administrator")]
    Public ActionResult ShutDown()
    {
    }
}
```

Policy based Role Checks

- Role requirements can also be expressed using the new Policy syntax, where a developer registers a policy at startup as part of the Authorization service configuration. This normally occurs in ConfigureServices() in your Startup.cs file.



```
public void ConfigureServices(IServiceCollection services)
{
services.AddControllersWithViews();
services.AddRazorPages();
    services.AddAuthorization(options =>
     {
        options.AddPolicy("RequireAdministratorRole",
                    policy =>policy.RequireRole("Administrator"));
        });
}
```

Policy based Role Checks

- Policies are applied using the Policy property on the AuthorizeAttribute attribute:

```
[Authorize(Policy = "RequireAdministratorRole")]
public IActionResult Shutdown() {
        return View();
}
```

- If you want to specify multiple allowed roles in a requirement then you can specify them as parameters to the RequireRole method:

```
options.AddPolicy("ElevatedRights", policy =>
    policy.RequireRole("Administrator", "PowerUser", "BackupAdministrator"));
```

- This example authorizes users who belong to the Administrator, PowerUser or BackupAdministrator roles.

Add Role services to Identity

- Append AddRoles to add Role services:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddDbContext<ApplicationDbContext>(options =>
   options.UseSqlServer(
       Configuration.GetConnectionString("DefaultConnection")));
  services.AddDefaultIdentity<IdentityUser>().AddRoles<IdentityRole>()
       .AddEntityFrameworkStores<ApplicationDbContext>();
  services.AddControllersWithViews();
  services.AddRazorPages();
}
```

Claims and Policies

- When an identity is created it may be assigned one or more claims issued by a trusted party. A claim is a name value pair that represents what the subject is, not what the subject can do. Claims based authorization, at its simplest, checks the value of a claim and allows access to a resource based upon that value.
- For example, if you want access to a night club the authorization process might be: The door security officer would evaluate the value of your date of birth claim and whether they trust the issuer (the driving license authority) before granting you access.
- An identity can contain multiple claims with multiple values and can contain multiple claims of the same type.

Adding claims checks

- First you need to build and register the policy. This takes place as part of the Authorization service configuration, which normally takes part in ConfigureServices() in your Startup.cs file.

```
public void ConfigureServices(IServiceCollection services) {
services.AddControllersWithViews();
services.AddRazorPages();
services.AddAuthorization(options =>
    {
    options.AddPolicy("EmployeeOnly", policy =>
 policy.RequireClaim("EmployeeNumber"));
    });
}
```

Adding claims checks

- In this case the EmployeeOnly policy checks for the presence of an EmployeeNumber claim on the current identity. You then apply the policy using the Policy property on the AuthorizeAttribute attribute to specify the policy name;

```
[Authorize(Policy = "EmployeeOnly")]
public IActionResult VacationBalance() {  return View();  }
```

- The AuthorizeAttribute attribute can be applied to an entire controller, in this instance only identities matching the policy will be allowed access to any Action on the controller.

```
[Authorize(Policy = "EmployeeOnly")]
public class VacationController:Controller {
        public ActionResult VacationBalance() { }
}
```

Policies

- If you apply multiple policies to a controller or action, then all policies must pass before access is granted. For example:

```
[Authorize(Policy = "EmployeeOnly")]
public class SalaryController : Controller{
        public ActionResult Payslip()
        {
        }
        [Authorize(Policy = "HumanResources")]
        public      ActionResult    UpdateSalary()
        {
        }
}
```

Policy-based authorization in ASP.NET Core

- Underneath the covers, role-based authorization and claims-basedauthorization use a requirement, a requirement handler, and a pre-configured policy.
- An authorization policy consists of one or more requirements. It's registered as part of the authorization service configuration, in the Startup.ConfigureServices method:

```
public void ConfigureServices(IServiceCollection services) {
    services.AddControllersWithViews();
    services.AddRazorPages();
    services.AddAuthorization(options =>
    {
        options.AddPolicy("AtLeast21", policy =>
        policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });
}
```

Apply policies to MVC controllers

- If you're using Razor Pages, see Apply policies to Razor Pages in this document.
- Policies are applied to controllers by using the [Authorize] attribute with the policy name. For example:

```
using Microsoft.AspNetCore.Authorization;
Using Microsoft.AspNetCore.Mvc;
[Authorize(Policy = "AtLeast21")]
public class AlcoholPurchaseController : Controller {
  public IActionResult Index() => View();
}
```

Apply policies to Razor Pages

- Policies are applied to Razor Pages by using the [Authorize] attribute with the policy name. For example:

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc.RazorPages;
[Authorize(Policy = "AtLeast21")]
public class AlcoholPurchaseModel : PageModel
{
}
```

- Policies cannot be applied at the Razor Page handler level, they must be applied to the Page.Policies can be applied to Razor Pages by using an authorization convention.

Securing Action Method in Controller

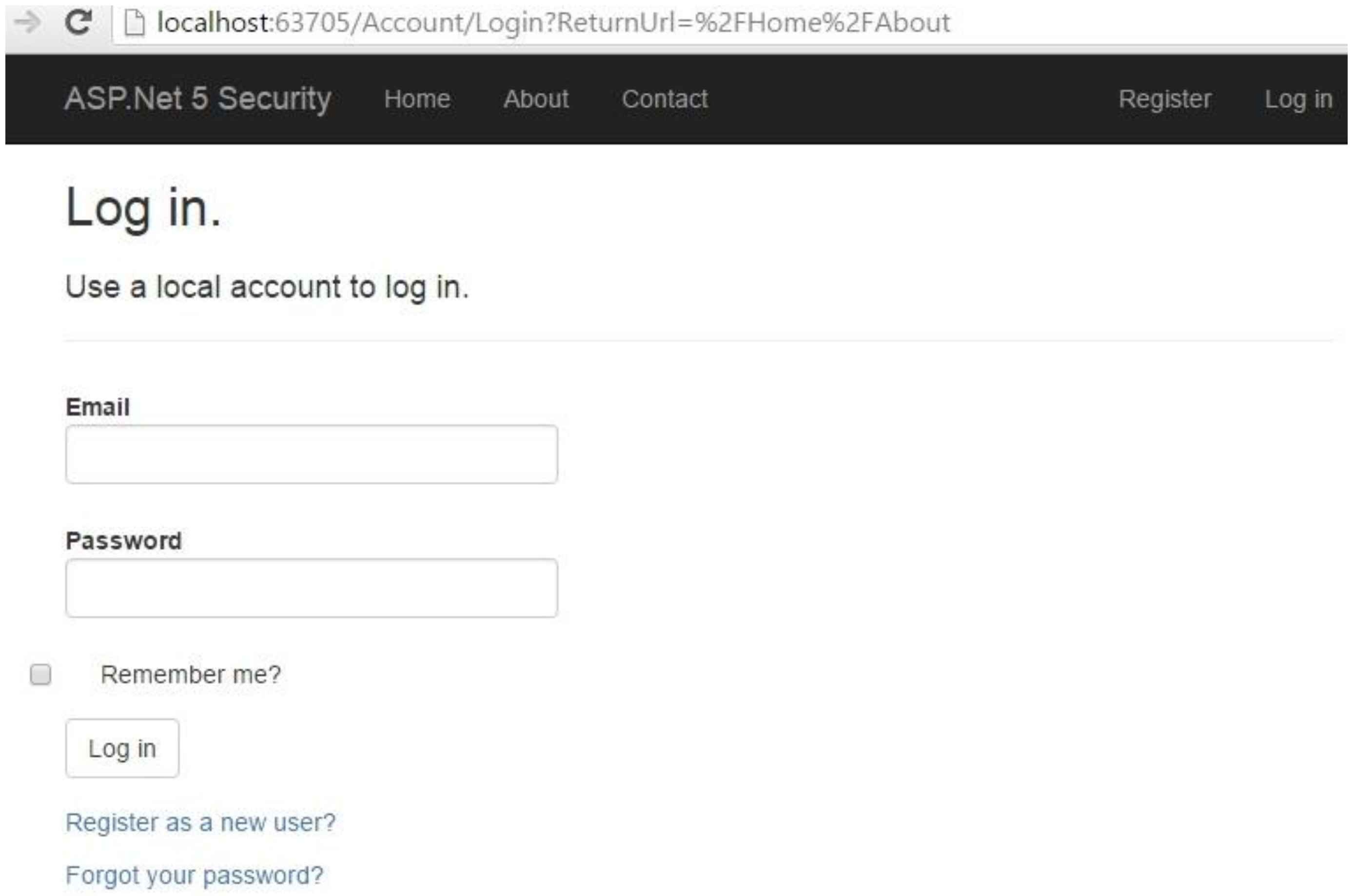
- Let’s assume that the About page is a secure page and only authenticated users should be able to access it. We just have to decorate the About action method in the Home controller with an[Authorize] attribute:

```
[Authorize]
public IActionResult About() {
   ViewData["Message"] = "This is my about page";
   return View();
}
```

- Making the preceding change will redirect user to the log-in page when user tries to access the log-in page without logging in to the application:

Securing Action Method in Controller

- Making the preceding change will redirect the user to the log-in page when the user tries to access the log-in page without logging in to the application:



localhost:63705/Account/Login?ReturnUrl=%2FHome%2FAbout

ASP.Net 5 Security Home About Contact Register Log in

Log in.

Use a local account to log in.

Email

Password

Remember me?

Log in

Register as a new user?

Forgot your password?