Asp.net Core Identity

install
Microsoft.AspNetCore.Identity.EntityFrameworkCore;

bez DbSet ne se dodava klasata vo baza

create user asynnc
sign in async
site **metodi** so **async** se koristat so **await** napred za da ceka da ne se sluci user da e sekogas null koga ke dojde do ifot

```
    var user= await userManager.FindByIdAsync(addUserToRole.UserId);

        var role = await roleManager.FindByNameAsync(addUserToRole.roleName);

        if (user!=null && role != null)
        {
            var rez = await userManager.AddToRoleAsync(user, role.Name);
            return RedirectToAction("Index", "Home");

        }
```

Identity

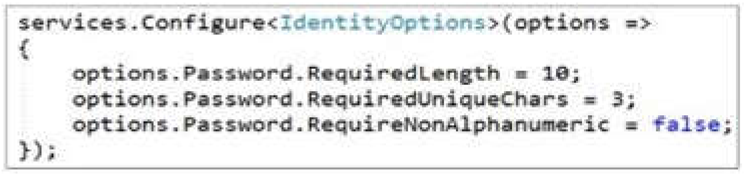
1.Adding and Configuring database

```
var con_string = configuration["ConnectionStrings:MyConnectionString"];

        services.AddDbContext<AppDbContext>(options =>
        options.UseSqlServer(configuration.GetConnectionString("MyConnectionString")));
```

2. to add idnetity services in DIC

```
        services.AddIdentity<UserIdentityUser, IdentityRole>(setupAction =>
        {

            setupAction.Password.RequiredLength = 4;
            setupAction.Password.RequiredUniqueChars = 1;
            setupAction.User.RequireUniqueEmail = true;

        }).AddEntityFrameworkStores<AppDbContext>();//vo koja baza da se cuvaat
```

OR



```
services.Configure<IdentityOptions>(options =>
{
    options.Password.RequiredLength = 10;
    options.Password.RequiredUniqueChars = 3;
    options.Password.RequireNonAlphanumeric = false;
});
```

3.Your app DbContext(objecto to acess datatbase) must inherit from IdentityDbContext
if the IdentityUser is cutom defined in another class it must be specefied as generic otherwise the migrations will be empty it will not detect the new custom properties

```
public class AppDbContext :IdentityDbContext<UserIdentityUser>
  {

    //  DbSet<Employee> Employees { get; set; }

      public AppDbContext(DbContextOptions<AppDbContext> options): base(options)
      {
      }

      protected override void OnModelCreating(ModelBuilder builder)
      {
         base.OnModelCreating(builder);//mora base bidejki vo OnModelCreating na
IdentityDbContext se pravat mapiranja za da modelot se mapira vo bazata
      } }
```

4. for custom IdentityUser

```
 public class UserIdentityUser : IdentityUser
  {
      public string Description { get; set; }//for more custom properties
  }
```

5.Configure middleware pipeline

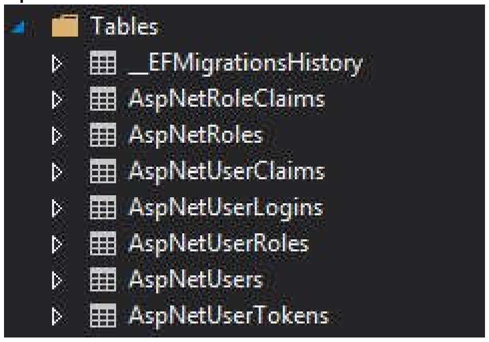
UseRouting()->Will try to select a route to execute but doesn't actually execute the route. It will select the endpoint for the current request.

UseAuthentication()->Populates the User(code that runs before this won't have a valid HttpContext.User property)

UseAuthorization()->Will look at the populated user and the current endpoint to determine if an authorization policy needs to be applied.

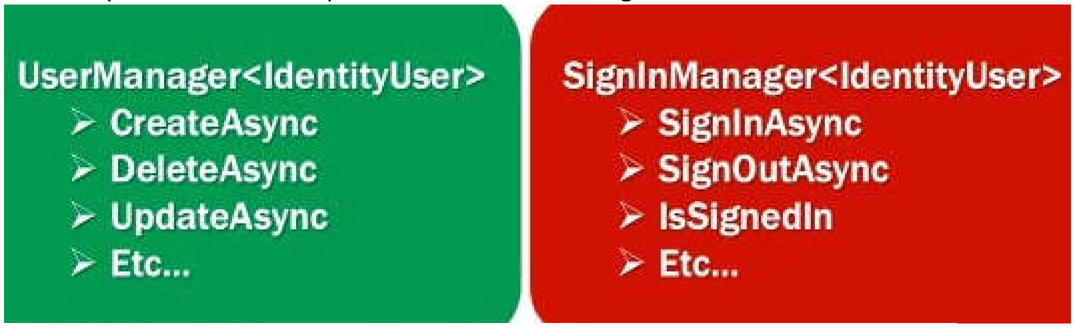
UseEndpoints()->Executes the current endpoint

6. Migrations
add-migration
update-Dtabase



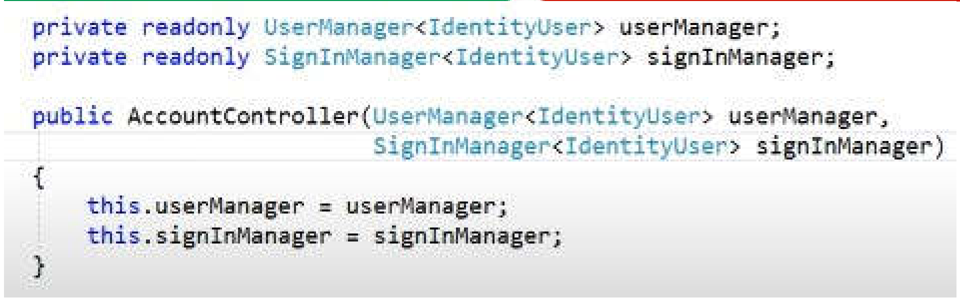
```
Tables
  __EFMigrationsHistory
  AspNetRoleClaims
  AspNetRoles
  AspNetUserClaims
  AspNetUserLogins
  AspNetUserRoles
  AspNetUsers
  AspNetUserTokens
```

7.User Manimulation UserManager< class represent the user > SignInManager<class represent the user>
Generic parameter that represent the user it manages



UserManager<IdentityUser>
- CreateAsync
- DeleteAsync
- UpdateAsync
- Etc...

SignInManager<IdentityUser>
- SignInAsync
- SignOutAsync
- IsSignedIn
- Etc...



```
private readonly UserManager<IdentityUser> userManager;
private readonly SignInManager<IdentityUser> signInManager;

public AccountController(UserManager<IdentityUser> userManager,
                         SignInManager<IdentityUser> signInManager)
{
    this.userManager = userManager;
    this.signInManager = signInManager;
}
```

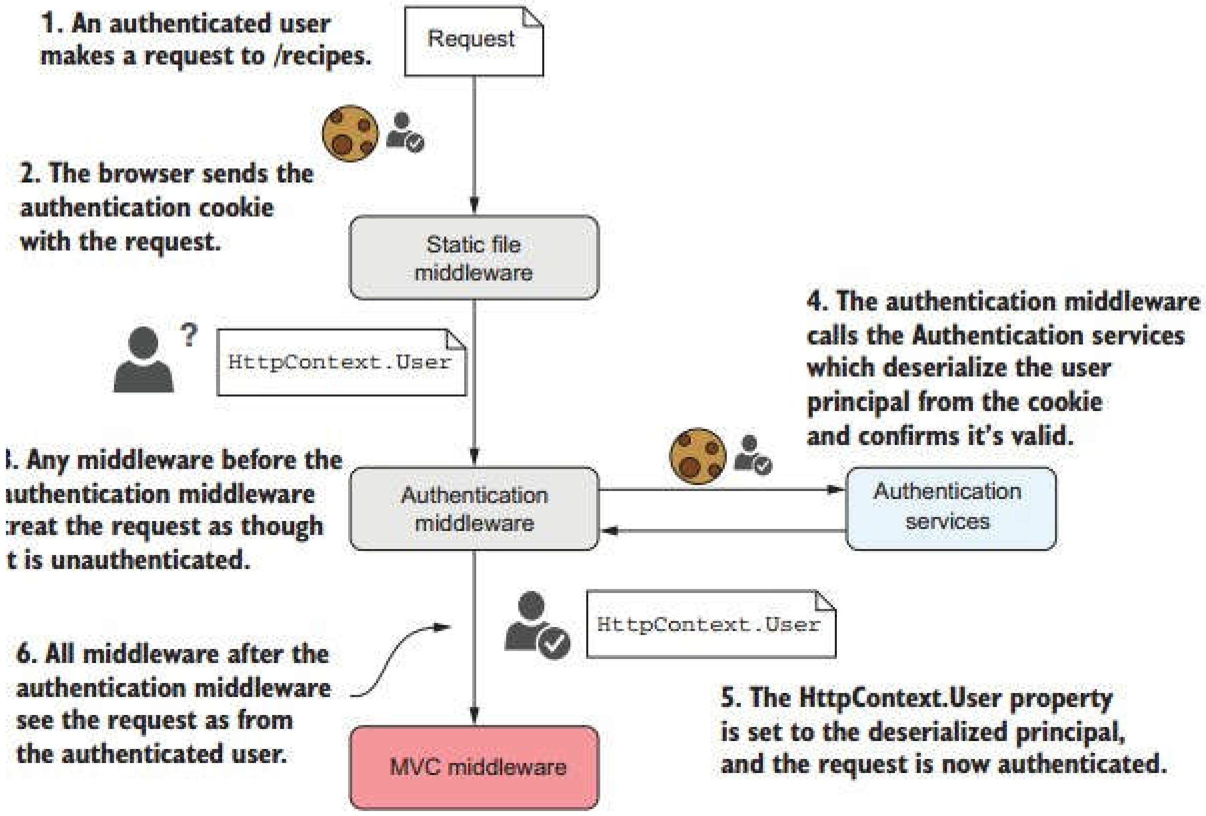
8.Razor
@* *@ //comments
 Role based authorization check in views in asp.net core mvc
@using Microsoft.AspNetCore.Identity
@inject SignInManager<IdentityUser> SignInManager
@if(SignInManager.IsSignedIn(User) && User.IsInRole("Admin"))
   {
   }

Claims based authorization check in views in asp.net core mvc

@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService authorizationService;
Pass the user and the name of the policy as parameters to AuthorizeAsync() method of IAuthorizationService. Succeeded property returns true if the policy is satisfied, otherwise false.

```
@if ((await authorizationService.AuthorizeAsync(User, "EditRolePolicy")).Succeeded)
{
}
```

9. Authentication


1. An authenticated user makes a request to /recipes.
Request
2. The browser sends the authentication cookie with the request.
Static file middleware
?
HttpContext.User
4. The authentication middleware calls the Authentication services which deserialize the user principal from the cookie and confirms it's valid.
3. Any middleware before the authentication middleware treat the request as though it is unauthenticated.
Authentication middleware
Authentication services
HttpContext.User
6. All middleware after the authentication middleware see the request as from the authenticated user.
5. The HttpContext.User property is set to the deserialized principal, and the request is now authenticated.
MVC middleware

Browsers will automatically
send this cookie with all requests made to your app, so you don't need to provide your password with every request.
by default asp.net core identity when it is not specefied it uses cookie authentication if it's specefied you shoud include addCookie

```
services.AddAuthentication().AddCookie();
```

A session cookie is created and stored within the session instance of the browser. A session cookie does not contain an expiration date and is permanently deleted when the browser window is closed. A persistent cookie on the other hand is not deleted when the browser window is closed. It usually has an expiry date and deleted on the date of expiry.

- LoginPath - this is the relative path requests will be redirected to when a user attempts to access a resource but has not been authenticated.
- AccessDeniedPath - this is the relative path requests will be redirected to when a user attempts to access a resource but does not pass any authorization policies for that resource.

```
// raboti za mestenje na LoginPath i AccessDeniedPath
   services.ConfigureApplicationCookie( options =>
            {

                options.LoginPath = new PathString("/Account/fffff/");
                options.AccessDeniedPath = new PathString("/Account/ddddd/");

            });
//ne raboti za mestenje na LoginPath i AccessDeniedPath

   services.AddAuthentication().AddCookie(
           config => {
               config.LoginPath = new PathString("/Account/fffff/");
               config.AccessDeniedPath = new PathString("/Account/ddddd/");
           }



           );
```

[Authorize] is used just to authenticate the user
Authentication is the process of determining a user's identity
schemes should be used to authenticate the user
Specifying the default scheme results in the HttpContext.User property being set to that identity. If that behavior isn't desired, disable it by invoking the parameterless form of AddAuthentication.

For example, the following code registers authentication services and handlers for cookie and JWT bearer authentication schemes:

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
  .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options =>
Configuration.Bind("JwtSettings", options))
  .AddCookie(CookieAuthenticationDefaults.AuthenticationScheme, options =>
Configuration.Bind("CookieSettings", options));
```

The AddAuthentication parameter JwtBearerDefaults.AuthenticationScheme is the name of the scheme to use by default when a specific scheme isn't requested.

```
services.AddAuthentication()
    .AddCookie(options => {
        options.LoginPath = "/Account/Unauthorized/";
        options.AccessDeniedPath = "/Account/Forbidden/";
    })
    .AddJwtBearer(options => {
        options.Audience = "http://localhost:5001/";
        options.Authority = "http://localhost:5000/";
    });
```

In the preceding code, two authentication handlers have been added: one for cookies and one for bearer.

Selecting the scheme with the Authorize attribute

[Authorize] attribute means you just need to authenticate(who are you)
The [Authorize] attribute specifies the authentication scheme or schemes to use regardless of whether a default is configured. For example:

```
private const string AuthSchemes =
        CookieAuthenticationDefaults.AuthenticationScheme + "," +
        JwtBearerDefaults.AuthenticationScheme;

 [Authorize(AuthenticationSchemes = AuthSchemes)]
public class MixedController : Controller
    // Requires the following imports:
    // using Microsoft.AspNetCore.Authentication.Cookies;
    // using Microsoft.AspNetCore.Authentication.JwtBearer;
```

In the preceding example, both the cookie and bearer handlers run and have a chance to create and append an identity for the current user. By specifying a single scheme only, the corresponding handler runs.

```
 [Authorize(AuthenticationSchemes =
    JwtBearerDefaults.AuthenticationScheme)]
public class MixedController : Controller
```

In the preceding code, only the handler with the "Bearer" scheme runs. Any cookie-based identities are ignored.

duplicate schemas name

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
        {
        })
        .AddJwtBearer("AzureAD", options =>
        {
        });
}
```

Only one JWT bearer authentication is registered with the default authentication scheme JwtBearerDefaults.AuthenticationScheme. Additional authentication has to be registered with a unique authentication scheme.

authorization with beerer token only

```
  [Authorize(AuthenticationSchemes =JwtBearerDefaults.AuthenticationScheme)]
```

Selecting the scheme can be done with policies

If you prefer to specify the desired schemes in policy, you can set the AuthenticationSchemes collection when adding your policy:

```
services.AddAuthorization(options =>
{
    options.AddPolicy("Over18", policy =>
    {
        policy.AuthenticationSchemes.Add(JwtBearerDefaults.AuthenticationScheme);
        policy.RequireAuthenticatedUser();
        policy.Requirements.Add(new MinimumAgeRequirement());
    });
});
```

In the preceding example, the "Over18" policy only runs against the identity created by the "Bearer" handler. Use the policy by setting the [Authorize] attribute's Policy property:

```
 [Authorize(Policy = "Over18")]
public class RegistrationController : Controller
```

10.Authorization

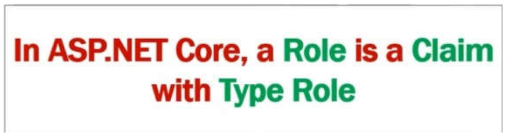
```
///  adding  Filters  globally
            services.AddControllersWithViews(configure=> {
                var policy = new AuthorizationPolicyBuilder()
.RequireAuthenticatedUser()  // ///  adding Authorization(the user must be logged in)
.AddRequirements(new SomeClass)   //za custom SomeClass koja ke bide Requirement odnosno ke nasledi od IAuthorizationRequirement
.Build();
```

```
  configure.Filters.Add(new AuthorizeFilter(policy)); ///  adding  Filters  globally
          });

//za da se iskoristat policy treba [Autthorize("policy name")]
          services.AddAuthorization(configure =>
          {
                    configure.AddPolicy("CanEnterSecurity", policyBuilder =>

             policyBuilder.RequireClaim("BoardingPassNumber")
             .RequireRole("Admin") //the user must have roleName Admin

             .RequireClaim("key", "value") //The user must have the specified claim. Optionally, with one
of the specified values.

             .RequireClaim("key", "value", "value1", "value2") //A list of allowed values can also be
specified. the user must have key claim with a value of claim1 or  claim2, or claim3

             .RequireAuthenticatedUser() //The required user must be authenticated. Creates a policy
similar to the default[Authorize] attribute, where you don't set a policy

             .RequireUserName("username") //The user must have the specified username.

             .RequireAssertion(context =>  //Executes the provided lambda function, which returns a
bool, indicating whether the policy was satisfied.
                          context.User.IsInRole("Admin") &&
                          context.User.HasClaim(claim => claim.Type == "Edit Role" && claim.Value ==
"true") ||
                          context.User.IsInRole("Super Admin"))

             .Requirements(new ManageAdminRolesAndClaimsRequirement()) //custom requirement

.AuthenticationSchemes.Add(CookieAuthenticationDefaults.AuthenticationScheme)//Selecting the
scheme can be done with policies only this policy only runs against the identity created by the cookie
authentication

   options.InvokeHandlersAfterFailure = false; // If you do not want the rest of the handlers to be
called, when a failure is returned, set InvokeHandlersAfterFailure property to false. The default
is true.

             );

          });
```

ASP.net core supports **role based, claim based** and **policy based** authorization

When the Authorize attribute is used in it's simplest form, without any parameters, it only checks if the user is authenticated. This is also called simple authorization.

Role-based


In ASP.NET Core, a Role is a Claim with Type Role

Claims based authorization is relatively new and is the recommended approach. With it we can also use claims from external identity providers like Facebook, Google, Twitter etc. We will discuss using external identity providers and the claims they provide in our upcoming videos.

Role based authorization is still supported in asp.net core for backward compatibility. While Claims based authorization is the recommended approach, depending on your application authorization requirements you may use role based authorization, claims based authorization or a combination of both.

Role-based authorization checks can be applied either against a controller or an action within a controller.

Role Based Authorization Example

Only those users who are members of the Administrator role can access the actions in the AdministrationController

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

Multiple Roles Example

Multiple roles can be specified by separating them with a comma.The actions in this controller are accessible only to those users who are members of either Administrator or User role.

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

Multiple Instances of Authorize Attribute

To be able to access the actions in this controller, users have to be members of both - the Administrator role and the User role.

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

Role Based Authorization Check on a Controller Action

Members of the Administrator role or the User role can access the controller and the ABC action, but only members of the Administrator role can access the XYZ action.The action Anyone() can be accessed by anyone inlcuding the anonymous users as it is decorated with AllowAnonymous attribute.

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

   [Authorize(Roles = "Administrator")]
   public ActionResult XYZ()
   {
   }

   [AllowAnonymous]
   public ActionResult Anyone()
   {
   }
}
```

claim-based

claims are informations about the user

There are 2 simple steps to implement Claims based authorization in asp.net core.

Create a claims policy

Use the policy on a controller or a controller action

Claims are policy based. We create a policy and include one or more claims in that policy. We then need to register the policy. Creating and registering a claims policy is typically done in one step in ConfigureServices() method of the Startup class.

```
services.AddAuthorization(options =>
{
   options.AddPolicy("DeleteRolePolicy",
        policy => policy.RequireClaim("Delete Role"));
});
```
The options parameter type is AuthorizationOptions
Use AddPolicy() method to create the policy
The first parameter is the name of the policy and the second parameter is the policy itself
To satisfy this policy requirements, the logged-in user must have Delete Role claim
Using Claims Policy for Authorization Checks

The policy can then be used on a controller or a controller action.

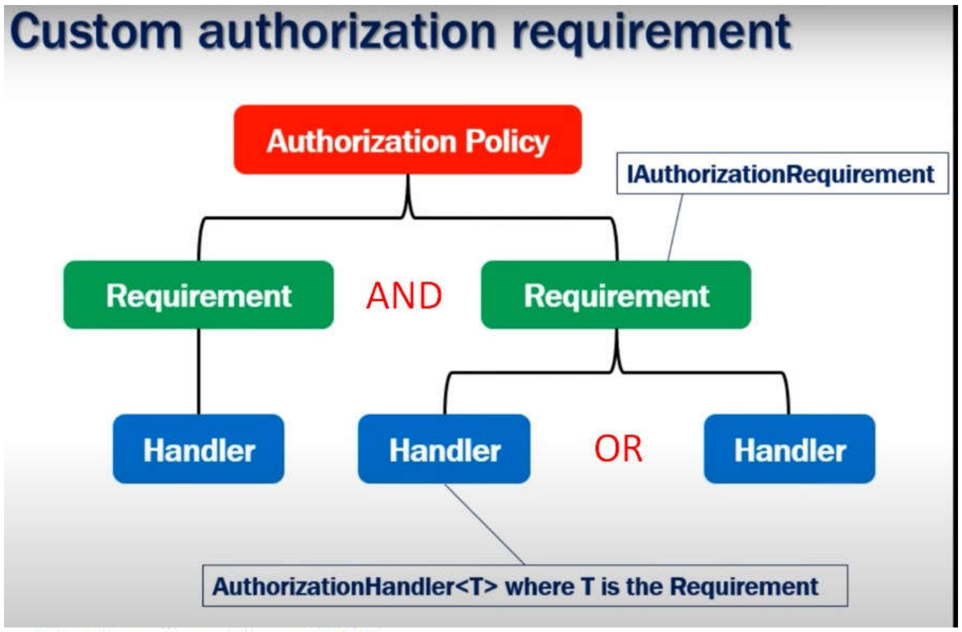
```
[HttpPost]
[Authorize(Policy = "DeleteRolePolicy")]
public async Task<IActionResult> DeleteRole(string id)
{
    // Delete Role
}
```

Adding Multiple Claims to Policy
```
services.AddAuthorization(options =>
{
    options.AddPolicy("DeleteRolePolicy",
        policy => policy.RequireClaim("Delete Role")
                        .RequireClaim("Create Role")
        );
});
```
you can also specified a value that needs to be satisfied
RequireClaim(claim, values) The user must have the specified claim. Optionally, with one of the specified values.

```
   // Add all the claims
            List<Claim> claims = new List<Claim>() {
            new Claim("key","value"),
            new Claim("key","value"),
            new Claim("key","value"),
            new Claim("key","value")
            };


    result = await userManager.AddClaimsAsync(user, claims);
    // Get all the user existing claims and delete them
    var claims = await userManager.GetClaimsAsync(user);
    var result = await userManager.RemoveClaimsAsync(user, claims);
```

policy-based
A policy defines the requirements you must meet in order for a request to be authorized.

```
services.AddAuthorization(configure =>
          {
              configure.AddPolicy("CanEnterSecurity", policyBuilder =>

              policyBuilder.RequireClaim("BoardingPassNumber")
              .RequireRole("Admin")//the user must have roleName Admin

              .RequireClaim("key", "value")//The user must have the specified claim. Optionally, with one
of the specified values.

              .RequireClaim("key", "value", "value1", "value2")//A list of allowed values can also be
specified. the user must have key claim with a value of claim1, claim2, or claim3

              .RequireAuthenticatedUser()//The required user must be authenticated. Creates a policy
similar to the default[Authorize] attribute, where you don’t set a policy

              .RequireUserName("username")//The user must have the specified username.

              .RequireAssertion(context => //Executes the provided lambda function, which returns a
bool, indicating whether the policy was satisfied.
                              context.User.IsInRole("Admin") &&
                              context.User.HasClaim(claim => claim.Type == "Edit Role" && claim.Value ==
"true") ||
                              context.User.IsInRole("Super Admin"))

              .Requirements(new ManageAdminRolesAndClaimsRequirement())//custom requirement

.AuthenticationSchemes.Add(CookieAuthenticationDefaults.AuthenticationScheme)//Selecting the
scheme can be done with policies only this policy only runs against the identity created by the cookie
authentication
   ) ;  });
```


Custom authorization requirement
Authorization Policy
IAuthorizationRequirement
Requirement
AND
Requirement
Handler
Handler
OR
Handler
AuthorizationHandler<T> where T is the Requirement

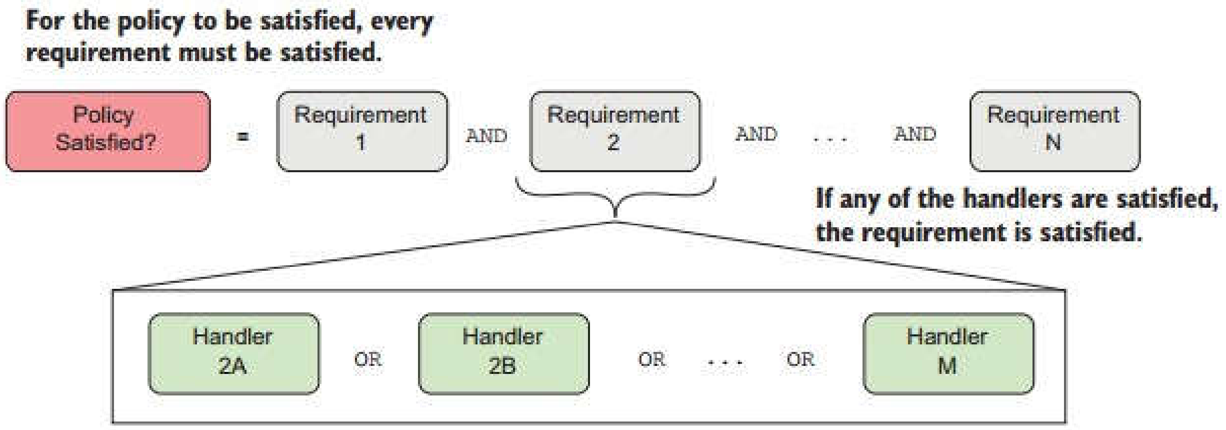
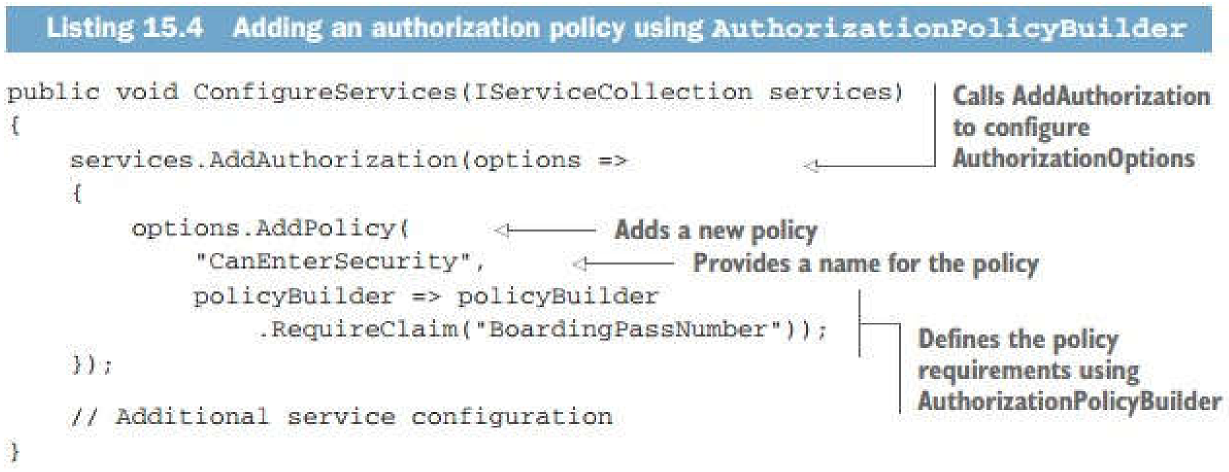

For the policy to be satisfied, every requirement must be satisfied.
Policy Satisfied?
=
Requirement 1
AND
Requirement 2
AND
. . .
AND
Requirement N
If any of the handlers are satisfied, the requirement is satisfied.
Handler 2A
OR
Handler 2B
OR
. . .
OR
Handler M

Figure 15.5 For a policy to be satisfied, every requirement must be satisfied. A requirement is satisfied if any of the handlers are satisfied.

Custom authorization requirements and Custom authorization handlers. These are very useful and powerful concepts that help us implement even the most complex authorization needs of an application.


Listing 15.4 Adding an authorization policy using `AuthorizationPolicyBuilder`

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy(
            "CanEnterSecurity",
            policyBuilder => policyBuilder
                .RequireClaim("BoardingPassNumber"));
    });

    // Additional service configuration
}
```

Calls AddAuthorization to configure AuthorizationOptions

Adds a new policy

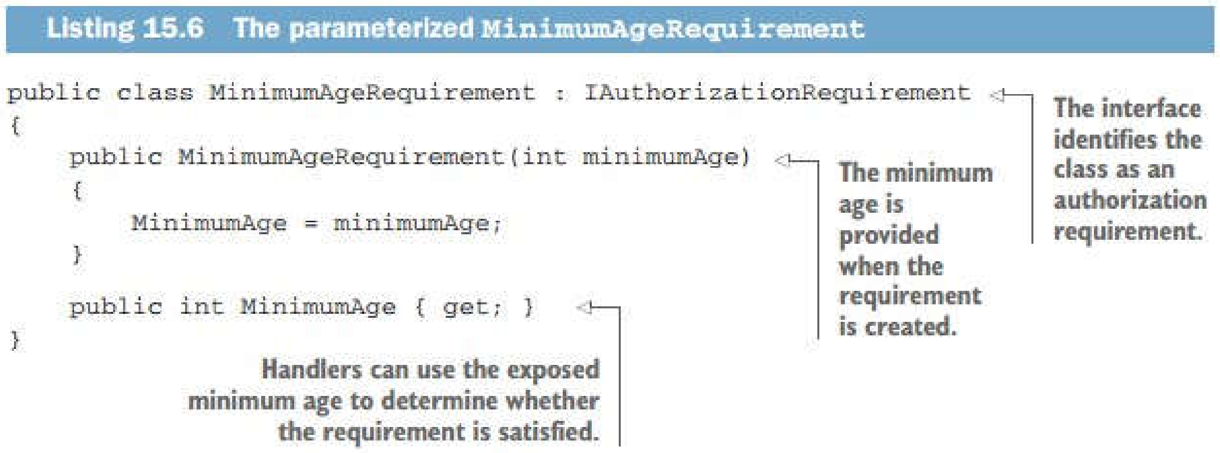
Provides a name for the policy

Defines the policy requirements using AuthorizationPolicyBuilder

When a user attempts to execute the AirportLounge action method, the authorization filter added by the [Authorize] attribute validates the "CanAccessLounge" policy. It loops through all of the requirements in the policy, and all of the handlers for
each requirement, calling the HandleRequirementAsync method for each.

1. Create requirements

public class ManageAdminRolesAndClaimsRequirement : IAuthorizationRequirement
{


Listing 15.6 The parameterized `MinimumAgeRequirement`

```
public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public MinimumAgeRequirement(int minimumAge)
    {
        MinimumAge = minimumAge;
    }

    public int MinimumAge { get; }
}
```

The interface identifies the class as an authorization requirement.

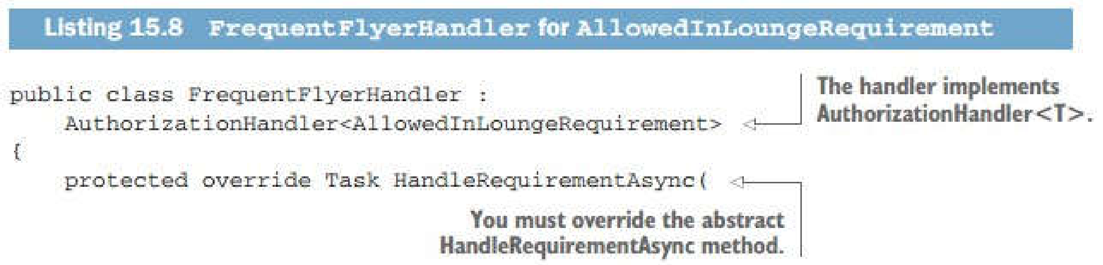
The minimum age is provided when the requirement is created.

Handlers can use the exposed minimum age to determine whether the requirement is satisfied.

}

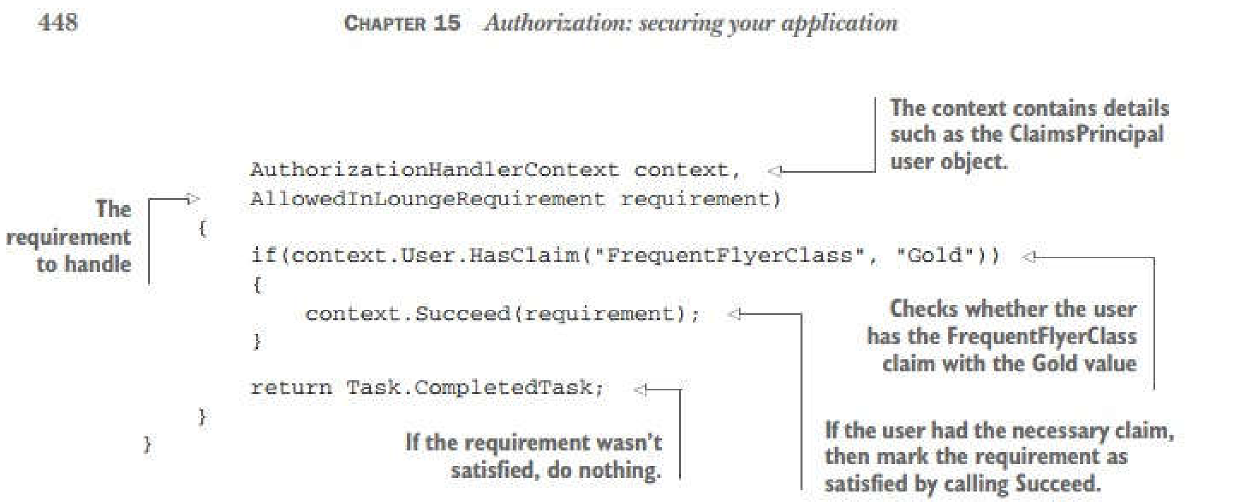
2. Create the Handler

public class CanEditOnlyOtherAdminRolesAndClaimsHandler :
  AuthorizationHandler<ManageAdminRolesAndClaimsRequirement>
{



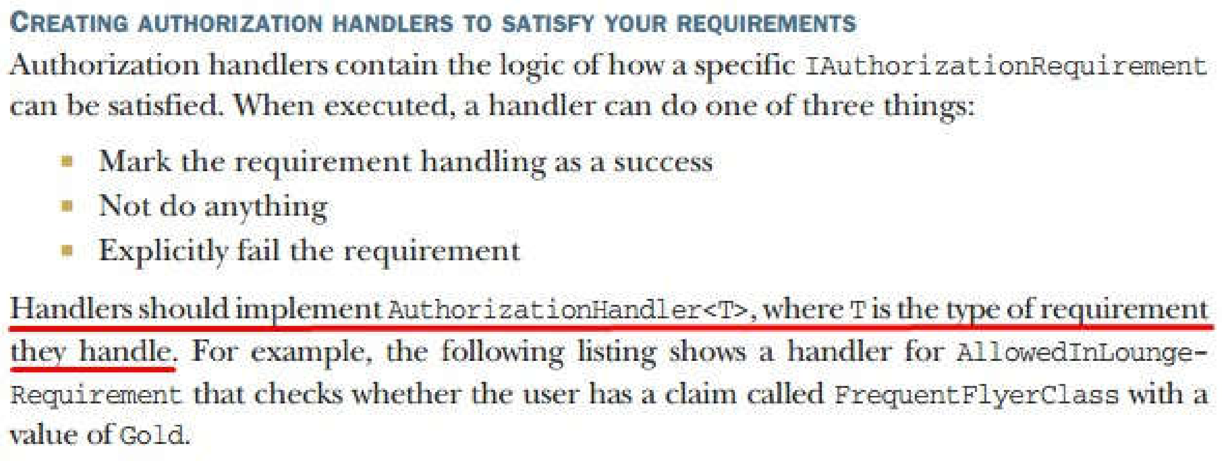
Listing 15.8 `FrequentFlyerHandler` for `AllowedInLoungeRequirement`

```
public class FrequentFlyerHandler :
    AuthorizationHandler<AllowedInLoungeRequirement>      <- The handler implements AuthorizationHandler<T>.
{
    protected override Task HandleRequirementAsync(      <- You must override the abstract HandleRequirementAsync method.
```



```
        AuthorizationHandlerContext context,      <- The context contains details such as the ClaimsPrincipal user object.
        AllowedInLoungeRequirement requirement)   <- The requirement to handle
    {
        if(context.User.HasClaim("FrequentFlyerClass", "Gold"))   <- Checks whether the user has the FrequentFlyerClass claim with the Gold value
        {
            context.Succeed(requirement);   <- If the user had the necessary claim, then mark the requirement as satisfied by calling Succeed.
        }
        return Task.CompletedTask;   <- If the requirement wasn't satisfied, do nothing.
    }
}
```

protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
ManageAdminRolesAndClaimsRequirement requirement)
{
context.Succeed(requirement); // Succeed() method specifies that the requirement is successfully evaluated.
return Task.CompletedTask; //we return Task.CompletedTask and the access is not authorised.



CREATING AUTHORIZATION HANDLERS TO SATISFY YOUR REQUIREMENTS

Authorization handlers contain the logic of how a specific `IAuthorizationRequirement` can be satisfied. When executed, a handler can do one of three things:

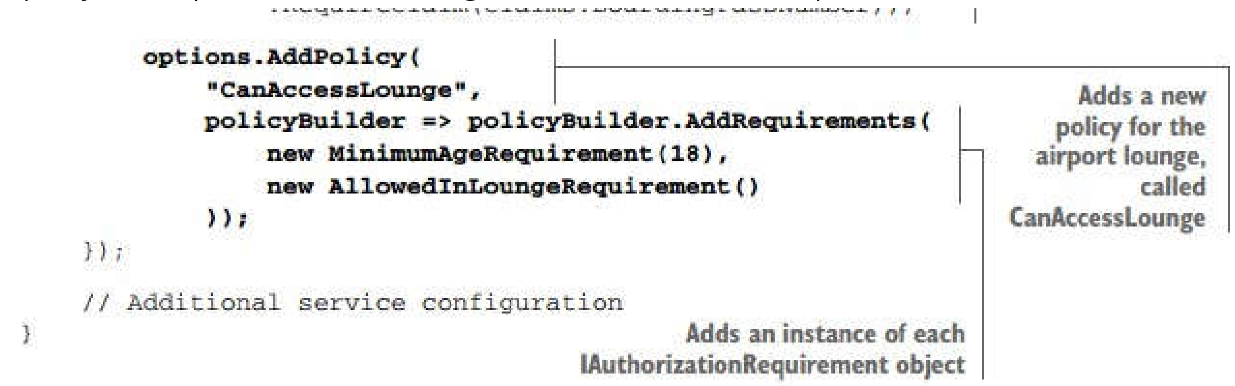
- Mark the requirement handling as a success
- Not do anything
- Explicitly fail the requirement

Handlers should implement `AuthorizationHandler<T>`, where `T` is the type of requirement they handle. For example, the following listing shows a handler for `AllowedInLoungeRequirement` that checks whether the user has a claim called `FrequentFlyerClass` with a value of `Gold`.

```
}
  }
```

3. Create the policy

```
  policy.AddRequirements(new ManageAdminRolesAndClaimsRequirement()));
```



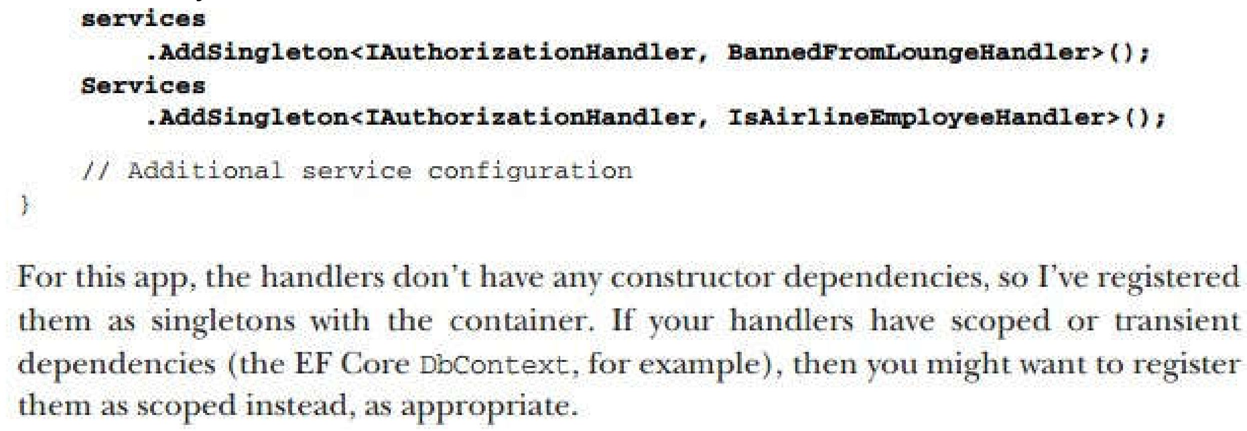
```
        options.AddPolicy(
            "CanAccessLounge",
            policyBuilder => policyBuilder.AddRequirements(
                new MinimumAgeRequirement(18),
                new AllowedInLoungeRequirement()
            ));
    });

    // Additional service configuration
}
```

Adds a new policy for the airport lounge, called CanAccessLounge

Adds an instance of each IAuthorizationRequirement object

4. Add the Handler to Dependency Injection Container

```
services.AddSingleton<IAuthorizationHandler,
    CanEditOnlyOtherAdminRolesAndClaimsHandler>();
```



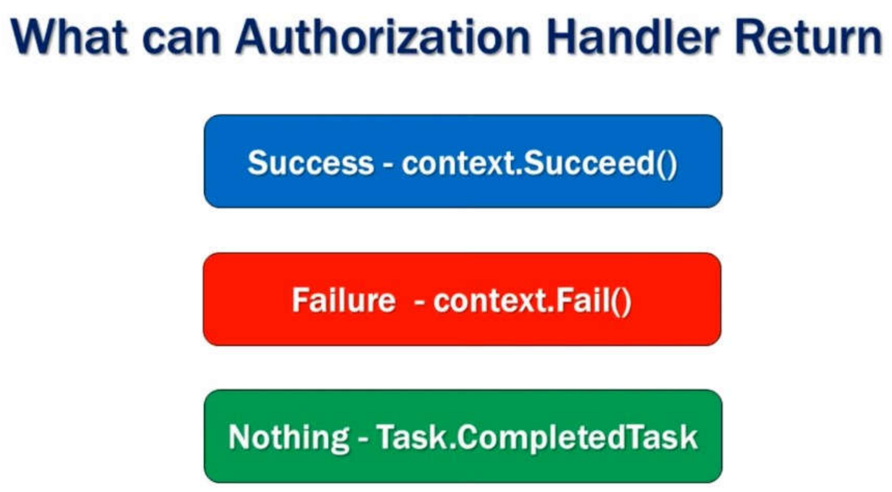
```
    services
        .AddSingleton<IAuthorizationHandler, BannedFromLoungeHandler>();
    Services
        .AddSingleton<IAuthorizationHandler, IsAirlineEmployeeHandler>();

    // Additional service configuration
}
```

For this app, the handlers don't have any constructor dependencies, so I've registered them as singletons with the container. If your handlers have scoped or transient dependencies (the EF Core `DbContext`, for example), then you might want to register them as scoped instead, as appropriate.

5. Apply the policy

```
[Authorize(Policy = "EditRolePolicy")]
```

Authorization Handler Returns



What can Authorization Handler Return

Success - context.Succeed()

Failure - context.Fail()

Nothing - Task.CompletedTask

It's very important we understand what the handler returns and the impact it can have on other handlers. If you have multiple handlers for a requirement

- Failure takes precedence over success. This means
- When one of the handlers return failure, the policy fails even if the other handlers return success
- If none of the handlers return an explicit success, the policy will not succeed.
- For a policy to succeed, an explicit success must be returned from one of the handlers, and no other handler must return an explicit failure
- In general, do not return failure from a handler, as other handlers for the same requirement may succeed.
- Only return an explicit failure, when you want to guarantee the failure of the policy even when the other handlers succeed.
- By default, all handlers are called, irrespective of what a handler returns (success, failure or nothing). This is because in the other handlers, there might be something else going on besides evaluating requirements, may be logging for example.
- If you do not want the rest of the handlers to be called, when a failure is returned, set InvokeHandlersAfterFailure property to false. The default is true.

Set InvokeHandlersAfterFailure property to false

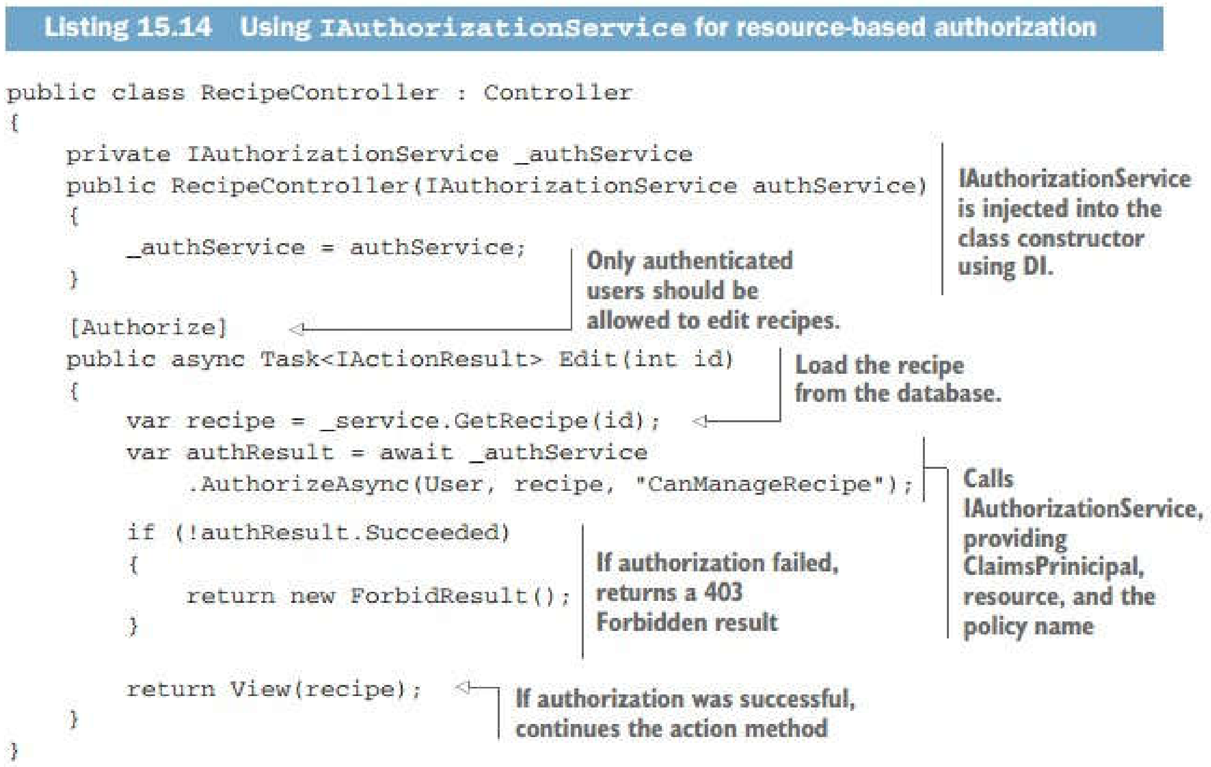
```
•       services.AddAuthorization(options =>
{
    options.AddPolicy("EditRolePolicy", policy =>
        policy.AddRequirements(new ManageAdminRolesAndClaimsRequirement()));

    options.InvokeHandlersAfterFailure = false; // If you do not want the rest of the handlers to be
called, when a failure is returned, set InvokeHandlersAfterFailure property to false. The default
is true.    });
```

Resource Base Authorization

In order to find out who created the Recipe, you must first load it from the database.

Create a policy in ConfigureServices by calling AddAuthorization()
Define one or more requirements for the policy
Define one or more handlers for each requirement
Register the handlers in the DI container

```
public void ConfigureServices(IServiceCollection services)
{
 services.AddAuthorization(options => {
 options.AddPolicy("CanManageRecipe", policyBuilder =>
 policyBuilder.AddRequirements(new IsRecipeOwnerRequirement()));
 });
}
```



Listing 15.14 Using `IAuthorizationService` for resource-based authorization

```
public class RecipeController : Controller
{
    private IAuthorizationService _authService
    public RecipeController(IAuthorizationService authService)
    {
        _authService = authService;
    }
    [Authorize]
    public async Task<IActionResult> Edit(int id)
    {
        var recipe = _service.GetRecipe(id);
        var authResult = await _authService
            .AuthorizeAsync(User, recipe, "CanManageRecipe");
        if (!authResult.Succeeded)
        {
            return new ForbidResult();
        }

        return View(recipe);
    }
}
```

IAuthorizationService is injected into the class constructor using DI.

Only authenticated users should be allowed to edit recipes.

Load the recipe from the database.

Calls IAuthorizationService, providing ClaimsPrinicipal, resource, and the policy name

If authorization failed, returns a 403 Forbidden result

If authorization was successful, continues the action method

Resource-based authorization handlers are essentially the same as the authorization handler implementations you saw in section 15.4.2. The only difference is that the handler also has access to the resource being authorized.
To create a resource-based handler, you should derive from the AuthorizationHandler<TRequirement, TResource> base class, where TRequirement is the type of requirement to handle, and TResource is the type of resource that you provide when calling IAuthorizationService.

Listing 15.15 `IsRecipeOwnerHandler` for resource-based authorization

```
public class IsRecipeOwnerHandler :
        AuthorizationHandler<IsRecipeOwnerRequirement, Recipe>
    {
        private readonly UserManager<ApplicationUser> _userManager;

        public IsRecipeOwnerHandler(
            UserManager<ApplicationUser> userManager)
        {
            _userManager = userManager;
        }
```

Implements the necessary base class, specifying the requirement and resource type

Injects an instance of the UserManager<T> class using DI

```
        protected override async Task HandleRequirementAsync(
            AuthorizationHandlerContext context,
            IsRecipeOwnerRequirement requirement,
            Recipe resource)
        {
            var appUser = await _userManager.GetUserAsync(context.User);
            if(appUser == null)
            {
                return;
            }

            if(resource.CreatedById == appUser.Id)
            {
                context.Succeed(requirement);
            }
        }
    }
```

As well as the context and requirement, you're also provided the resource instance.

If you aren't authenticated, then appUser will be null.

Checks whether the current user created the Recipe by checking the CreatedById property

If the user created the document, Succeed the requirement; otherwise, do nothing.

Role Manager

RoleManager < class represent the role >

Asp.Net Core Identity Tables

AspNetUsers
The user profile table itself. This is where ApplicationUser is serialized to. We'll take a closer look at this table shortly.

AspNetUserClaims
The claims associated with a given user. A user can have many claims, so it's modeled as a many-to-one relationship.

AspNetUserLogins and AspNetUserTokens
These are related to third-party logins.When configured, these let users sign in with a Google or Facebook account(for example), instead of creating a password on your app.

AspNetUserRoles, AspNetRoles, and AspNetRoleClaims
These tables are somewhat of a legacy left over from the old role-based permission model of the pre-.NET 4.5 days, instead of the claims-based permission model. These tables let you
define roles that multiple users can belong to. Each role can be assigned a number of claims. These claims are effectively inherited by a user principal when
they are assigned that role.

AspNetRoleClaims
?
AspNetRoles
Roles that you have created

AspNetUserRoles
many to many replationship between AspNetUsers and AspNetRoles

Register User

```
    [HttpPost]
        [AllowAnonymous]
        public async Task<IActionResult> Register(RegisterViewModel registerView)
        {
            if (ModelState.IsValid)
            {
                var user = new UserIdentityUser { Email = registerView.Email, UserName =
registerView.Name };

                var result = await userManager.CreateAsync(user, registerView.Password);//checks if user is
valid and create it in database

                if (result.Succeeded)//if user is successfully created sign in
                {
                    await signInManager.SignInAsync(user, false);///Updates the HttpContexxt.User principle
and sets a cookie containing the serilized principle

                    return RedirectToAction("Index", "Home");
                }

                foreach (var error in result.Errors)
                {

                    ModelState.AddModelError("", error.Description.ToLower());
                }

            }

                return View(registerView);

        }
```

Login User

```
[HttpPost]
        [AllowAnonymous]
        public async Task<IActionResult> Login(LoginViewModel loginView,string returnUrl)
        {


            if (ModelState.IsValid)
            {
                var result = await signInManager.PasswordSignInAsync(loginView.Email,
loginView.Password, false, false);//the sign in succeeded so the HttpContext.User  and authentication
cookie have been set

                if (result.Succeeded)
                {
                    if(!string.IsNullOrEmpty(returnUrl) && Url.IsLocalUrl(returnUrl))

                        return LocalRedirect(returnUrl);

                    else
                        return RedirectToAction("Index","Home");

                }
                else
                {
                    ModelState.AddModelError("", "sign in failed");
                    return View(loginView);
                }

            }
            else
            {
                return View(loginView);
            }



        }
```

Logout User

```
[HttpPost]
        public async Task<IActionResult> Logout()
        {
            await signInManager.SignOutAsync();// Replaces the HttpContexxt.User with an anonymous
principle and deletes the authentication cookie

            return RedirectToAction("Index", "Home");
        }
```

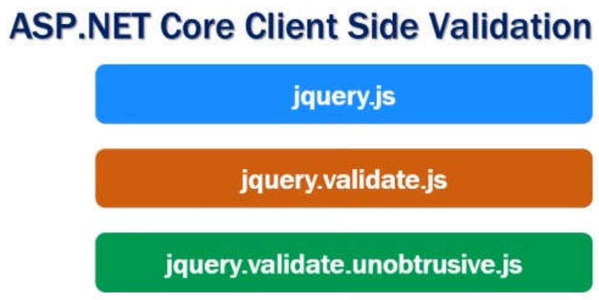
Create Role

```
   [HttpPost]
        [Authorize(Roles = "Admin")]
        public IActionResult CreateRole(CreateRoleViewModel model)
        {
            if (ModelState.IsValid)
            {
                var role = new IdentityRole(model.RoleName);

                var result = roleManager.CreateAsync(role);

                if (result.Result.Succeeded)
                {
                    return RedirectToAction("Index", "Home");
                }

                foreach (var error in result.Result.Errors)
                {

                    ModelState.AddModelError("", error.Description.ToLower());
                }
            }
            return View(model);
        }
```

Add User To Role

```
        [HttpPost]
        [Authorize(Roles = "Admin")]
        public async Task<IActionResult> AddUserToRole(AddUserToRoleViewModel
addUserToRole)
        {
            var user = await userManager.FindByIdAsync(addUserToRole.UserId);

            var role = await roleManager.FindByNameAsync(addUserToRole.roleName);

            var isAlreadyInRole = await userManager.IsInRoleAsync(user, role.Name);//if
the user already has the role

            if (user != null && role != null && !isAlreadyInRole)
            {
                var rez = await userManager.AddToRoleAsync(user, role.Name);


                if(rez.Succeeded)
                return RedirectToAction("Index", "Home");

                foreach (var error in rez.Errors)
                {
                    ModelState.AddModelError("", error.Description);
                }
                return View(addUserToRole);

            }
            else
            {
                ModelState.AddModelError("", $"user {user} role {role} already has the
role {isAlreadyInRole}");
                return View(addUserToRole);
            }



        }
```

Client Side Validation


ASP.NET Core Client Side Validation
jquery.js
jquery.validate.js
jquery.validate.unobtrusive.js

```
    <link href="~/lib/twitter-bootstrap/css/bootstrap.css" rel="stylesheet" />
    <script src="~/lib/jquery/jquery.js"></script>
    <script src="~/lib/jquery-validate/jquery.validate.js"></script>
    <script src="~/lib/jquery-validation-
unobtrusive/jquery.validate.unobtrusive.js"></script>
    <script src="~/lib/twitter-bootstrap/js/bootstrap.js"></script>
```

//// libman.json

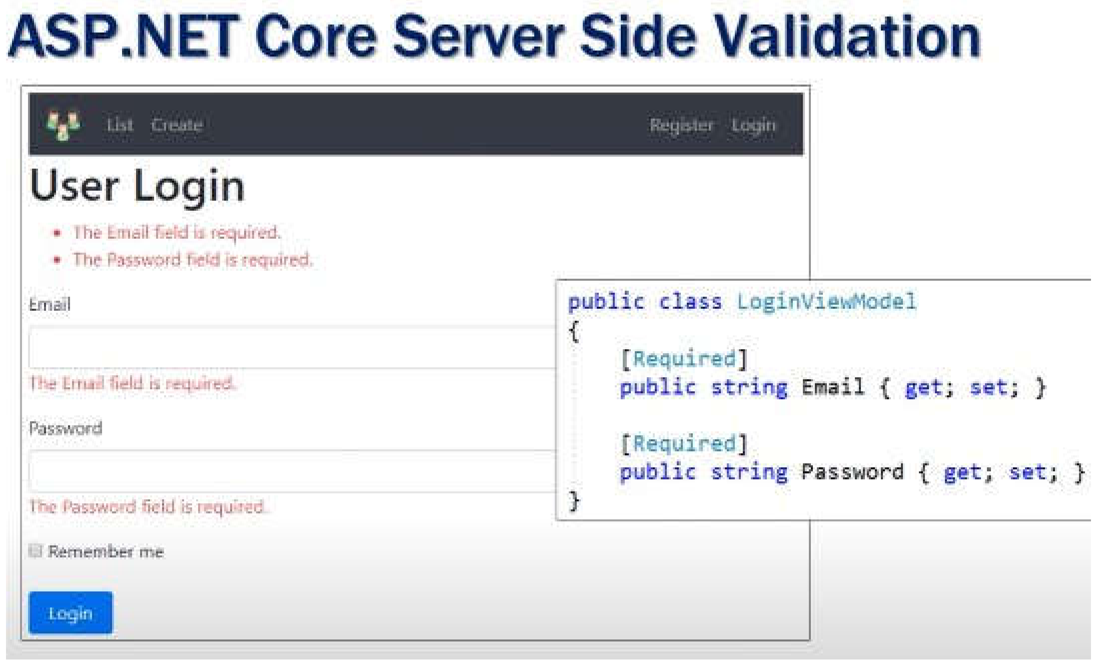
```
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": [
    {
      "library": "jquery@3.5.1",
      "destination": "wwwroot/lib/jquery/"
    },
    {
      "library": "twitter-bootstrap@4.5.0",
      "destination": "wwwroot/lib/twitter-bootstrap/"
    },
    {
      "library": "jquery-validate@1.19.1",
      "destination": "wwwroot/lib/jquery-validate/"
    }
  ,
{
  "library": "jquery-validation-unobtrusive@3.2.11",
  "destination": "wwwroot/lib/jquery-validation-unobtrusive/"}]}
```

Make sure the following client-side validation libraries are loaded in the order specified jquery.js jquery.validate.js jquery.validate.unobtrusive.js
Unobtrusive Validation allows us to take the already-existing server side validation attributes and use them to implement client-side validation. We do not have to write a single line of custom

JavaScript code. All we need is the above 3 script files in the order specified. How does client side validation work in ASP.NET Core ASP.NET Core tag helpers work in combination with the model validation attributes and generate the following HTML. Notice in the generated HTML we have data-* attributes.
 [input id="Email" name="Email" type="email" data-val="true" data-val-required="The Email field is required." /] The data-* attributes allow us to add extra information to an HTML element. These data-* attributes carry all the information required to perform the client-side validation. It is the unobtrusive library (i.e jquery.validate.unobtrusive.js) that reads these data-val attributes and performs the client side validation.

Bootstrap uses jQuery for JavaScript plugins (like modals, tooltips, etc). However, if you just **use** the CSS part of **Bootstrap**, you don't need **jQuery**.

Server Side Validation


ASP.NET Core Server Side Validation
List
Create
Register
Login
User Login
The Email field is required.
The Password field is required.
Email
The Email field is required.
Password
The Password field is required.
Remember me
Login

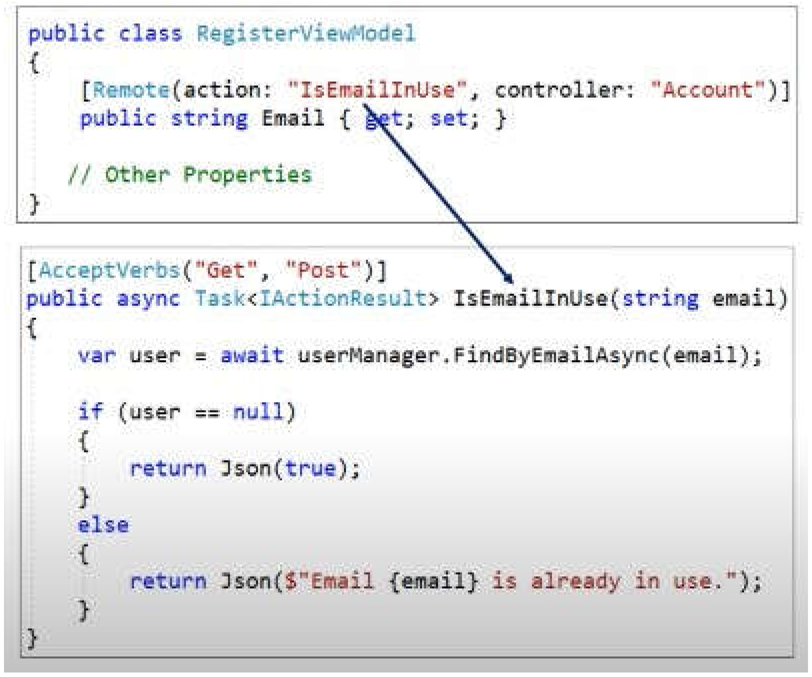
```
public class LoginViewModel
{
    [Required]
    public string Email { get; set; }

    [Required]
    public string Password { get; set; }
}
```

Remote validation

Remote validation allows a controller action method to be called using client side script. This is very useful when you want to call a server side method without a full page post back.

Checking, if the provided email is already taken by another user can only be done on the server
Remote validation allows a controller action method to be called using client side script
Client Side scripts required in this order

```
    <script src="~/lib/jquery/jquery.js"></script>
    <script src="~/lib/jquery-validate/jquery.validate.js"></script>
    <script src="~/lib/jquery-validation-unobtrusive
```

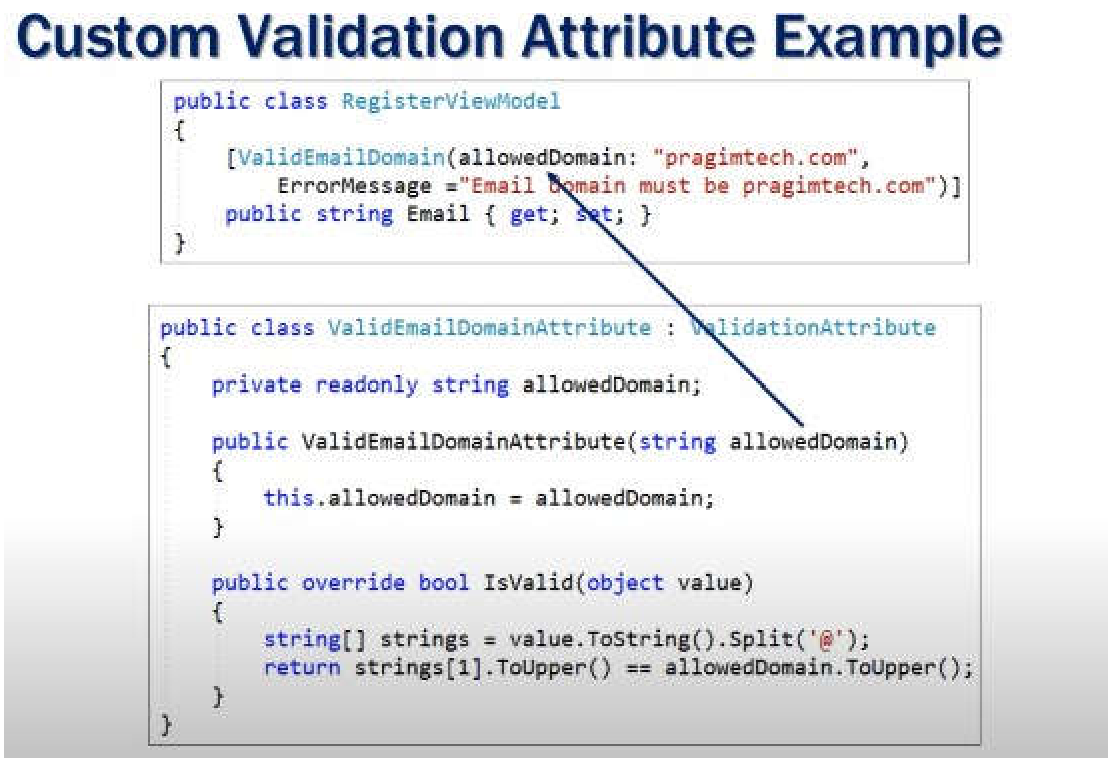


```
public class RegisterViewModel
{
    [Remote(action: "IsEmailInUse", controller: "Account")]
    public string Email { get; set; }

   // Other Properties
}
```

```
[AcceptVerbs("Get", "Post")]
public async Task<IActionResult> IsEmailInUse(string email)
{
    var user = await userManager.FindByEmailAsync(email);

    if (user == null)
    {
        return Json(true);
    }
    else
    {
        return Json($"Email {email} is already in use.");
    }
}
```

string email e najverojatno name na input tagot

Custom validation attribute



Custom Validation Attribute Example

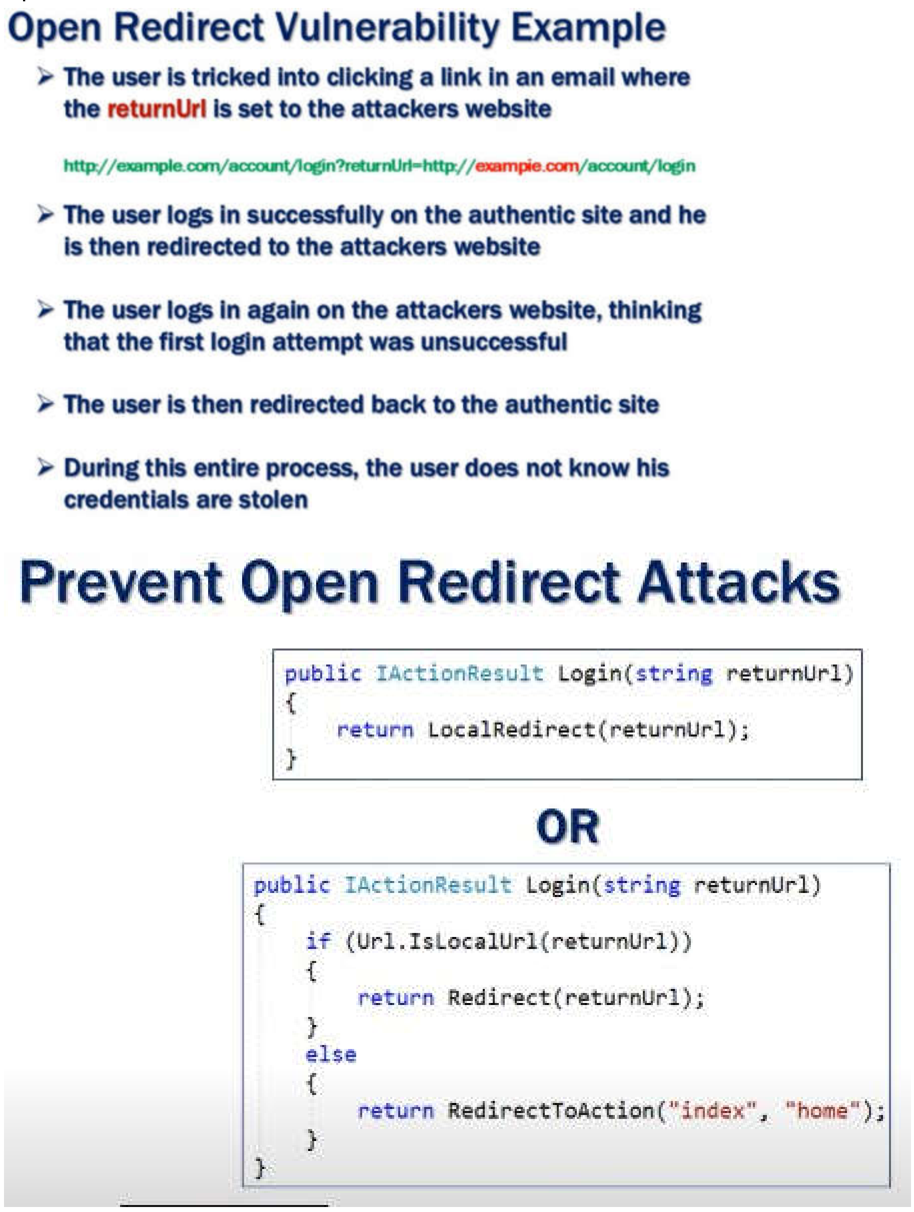
```
public class RegisterViewModel
{
    [ValidEmailDomain(allowedDomain: "pragimtech.com",
        ErrorMessage ="Email domain must be pragimtech.com")]
    public string Email { get; set; }
}
```

```
public class ValidEmailDomainAttribute : ValidationAttribute
{
    private readonly string allowedDomain;

    public ValidEmailDomainAttribute(string allowedDomain)
    {
        this.allowedDomain = allowedDomain;
    }

    public override bool IsValid(object value)
    {
        string[] strings = value.ToString().Split('@');
        return strings[1].ToUpper() == allowedDomain.ToUpper();
    }
}
```

ErrorMessage postoi vo ValitadioAttribute kako public i se nadleduva
object value doaga od input filed za Email bidejki atributot e na email

Application Vulnerability

Open Redirect



Open Redirect Vulnerability Example

- The user is tricked into clicking a link in an email where the returnUrl is set to the attackers website

  http://example.com/account/login?returnUrl=http://exampie.com/account/login

- The user logs in successfully on the authentic site and he is then redirected to the attackers website
- The user logs in again on the attackers website, thinking that the first login attempt was unsuccessful
- The user is then redirected back to the authentic site
- During this entire process, the user does not know his credentials are stolen

Prevent Open Redirect Attacks

```
public IActionResult Login(string returnUrl)
{
    return LocalRedirect(returnUrl);
}
```

OR

```
public IActionResult Login(string returnUrl)
{
    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
    else
    {
        return RedirectToAction("index", "home");
    }
}
```

Api Authentication and Authorization with JSON Web Token(JWT)

1.install
Microsoft.AspNetCore.Authentication.JwtBearer

2.
```
services.AddAuthentication()
.AddCookie(); //must be called for coookie explicitly if services.AddAuthentication() is used
.AddJwtBearer(cfg => {
cfg.TokenValidationParameters = new TokenValidationParameters() {
ValidateIssuer = true,
ValidIssuer = _config["Security:Tokens:Issuer"],
ValidateAudience = true,
ValidAudience = _config["Security:Tokens:Audience"],
ValidateIssuerSigningKey = true,
IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Security:Tokens:Key"])), }; });
```

Issuer - is the principal that has issued JWT. If token has different issuer than expected, the validation will fail and caller will receive 401 unauthorized.

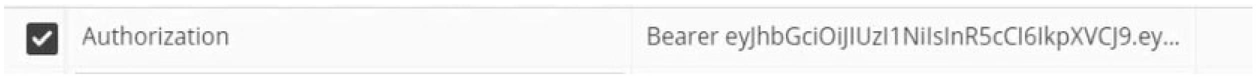
Audience - is the recipient that JWT is intended for. If token contains different audience than expected, the validation will fail and caller will receive 401 unauthorized.

Signing Key, is the key you use for signing the token.

ValidateIssuer – Gets or sets a value indicating whether the Issuer should be validated. True means Yes validation required.

ValidateAudience – Gets or sets a boolean to control if the audience will be validated during token validation.

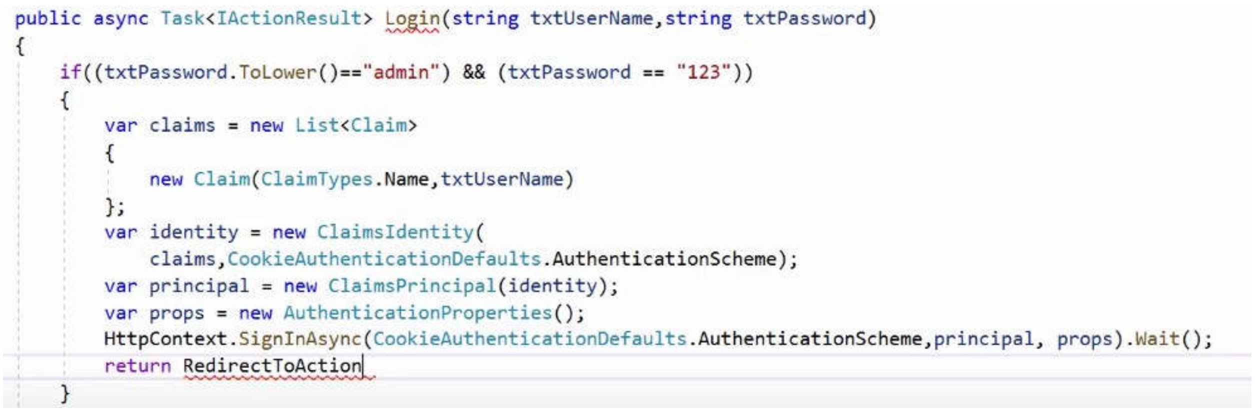
```
   var claims = new[]
         {
          new Claim(JwtRegisteredClaimNames.Sub, "Max John"),
          new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),
        };

            var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(configuration["Security:Tokens:Key"]));
            var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

            var token = new JwtSecurityToken(configuration["Security:Tokens:Issuer"],
              configuration["Security:Tokens:Audience"],
              claims,
              expires: DateTime.Now.AddMinutes(30),
              signingCredentials: creds);


            var tokenText = new JwtSecurityTokenHandler().WriteToken(token);
//tokenot treba da bide poveke od 16 karakteri

            return Ok(tokenText);
```



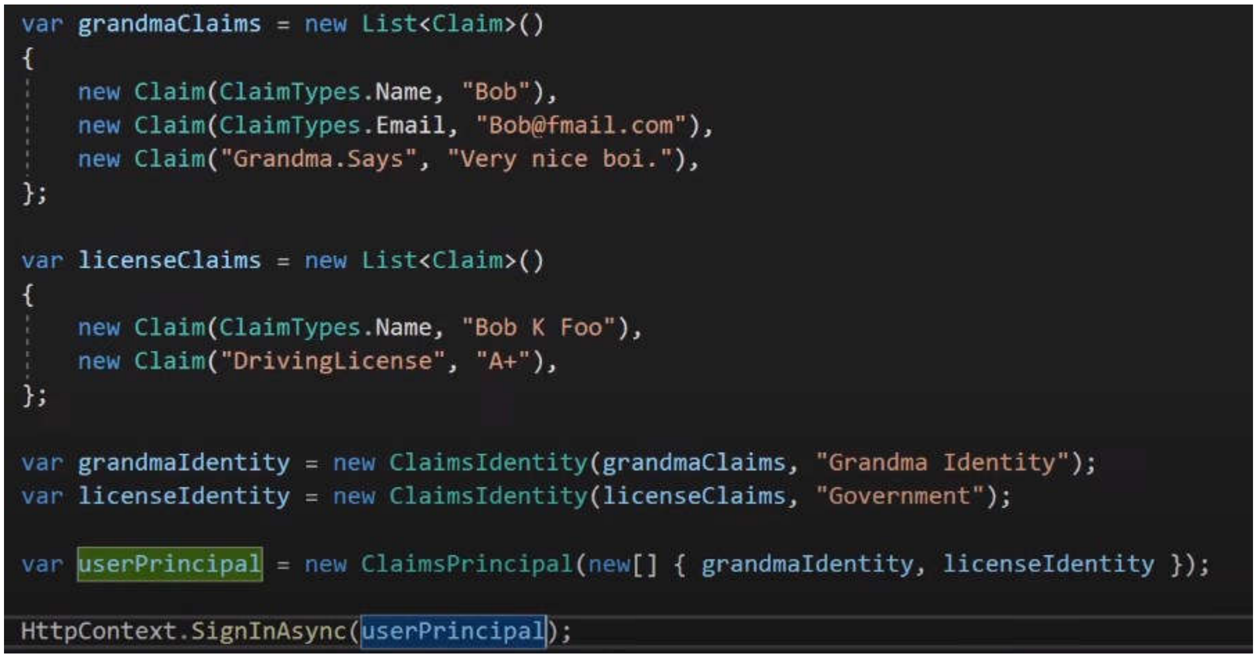
☑ Authorization | Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey...

Basics (Claims/ClaimsIdentity/ClaimsPrincipal/Authorization)

HttpContet.SignInAsync setup the cookie



```
public async Task<IActionResult> Login(string txtUserName,string txtPassword)
{
    if((txtPassword.ToLower()=="admin") && (txtPassword == "123"))
    {
        var claims = new List<Claim>
        {
            new Claim(ClaimTypes.Name,txtUserName)
        };
        var identity = new ClaimsIdentity(
            claims,CookieAuthenticationDefaults.AuthenticationScheme);
        var principal = new ClaimsPrincipal(identity);
        var props = new AuthenticationProperties();
        HttpContext.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme,principal, props).Wait();
        return RedirectToAction
    }
```



```
var grandmaClaims = new List<Claim>()
{
    new Claim(ClaimTypes.Name, "Bob"),
    new Claim(ClaimTypes.Email, "Bob@fmail.com"),
    new Claim("Grandma.Says", "Very nice boi."),
};

var licenseClaims = new List<Claim>()
{
    new Claim(ClaimTypes.Name, "Bob K Foo"),
    new Claim("DrivingLicense", "A+"),
};

var grandmaIdentity = new ClaimsIdentity(grandmaClaims, "Grandma Identity");
var licenseIdentity = new ClaimsIdentity(licenseClaims, "Government");

var userPrincipal = new ClaimsPrincipal(new[] { grandmaIdentity, licenseIdentity });

HttpContext.SignInAsync(userPrincipal);
```

External identity providers in asp.net core

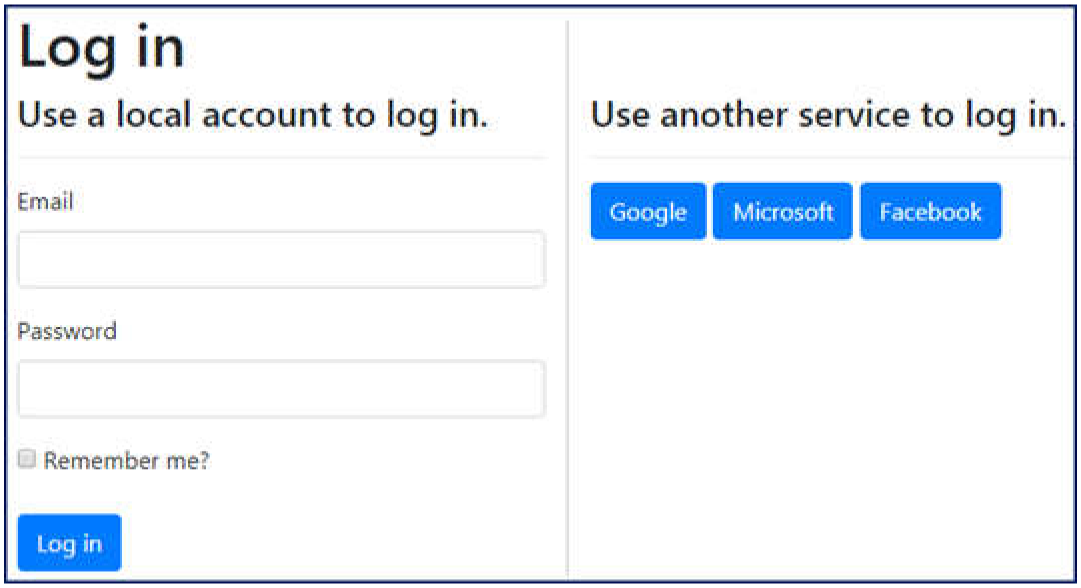
Most people these days have accounts already created with third party applications like Microsoft, Google, Facebook, Twitter etc. Why do these users have to create yet another account with our application. It's not a great user experience to ask the user to create yet another user account just to login to our application. We want to reuse their existing user account with Facebook, Google, Microsoft, Twitter etc.

We trust these third party applications and use them to authenticate and identify who the user is. For this reason these third party applications are commonly called **trusted identity providers**. To be more accurate, we call them **trusted external identity providers**, as these third party applications are external to our own application. Windows authentication can also be used as an external identity provider.
Allowing users to reuse their existing accounts to log into our application benefits end users from having to remember yet another username and password. It also benefits us as application developers as we no longer have to store and maintain the highly sensitive information such as the username and password in our application database. It is now the responsibility of the external authentication provider such as Facebook or Google for example.

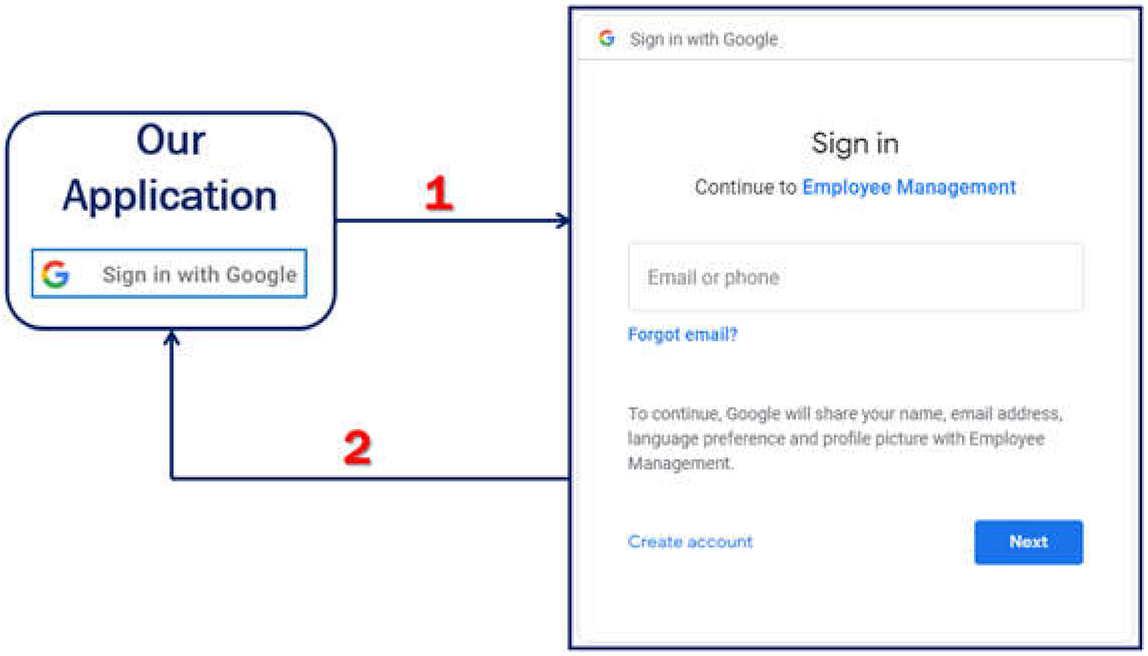
Asp.net core has built-in support for integrating these **external authentication providers**. Integrating these external authentication providers follows a similar pattern. If we understand how to integrate one of the external authentication providers, implementing others is not that different.
These are all different ways of essentially authenticating the same user. We are just providing him/her different options to login to our application. When a given user logs in using these different authentication providers we do not want to end up creating multiple accounts for that same user. We have to associate these different ways of logging in, to the same user account in our application. We will discuss how to do this in our upcoming videos as we progress in this course.

How external identity providers work in asp.net core Google

If a user wants to use his/her Google account to signin to our application, they click the **Google** button.


Log in
Use a local account to log in.
Email
Password
Remember me?
Log in
Use another service to log in.
Google
Microsoft
Facebook

Our application then redirects the user to Google signin page. Here the user provides his/her google login credentials.

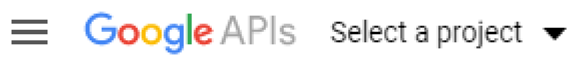

Our Application
Sign in with Google
1
2
Sign in with Google
Sign in
Continue to Employee Management
Email or phone
Forgot email?
To continue, Google will share your name, email address, language preference and profile picture with Employee Management.
Create account
Next

Upon a successful login, google will then send the user back to our application and a pre-configured callback function is executed. The code in this callback function checks the identity received from the external identity provider and sign-in that user into our application. We will see all this in action in our upcoming videos.

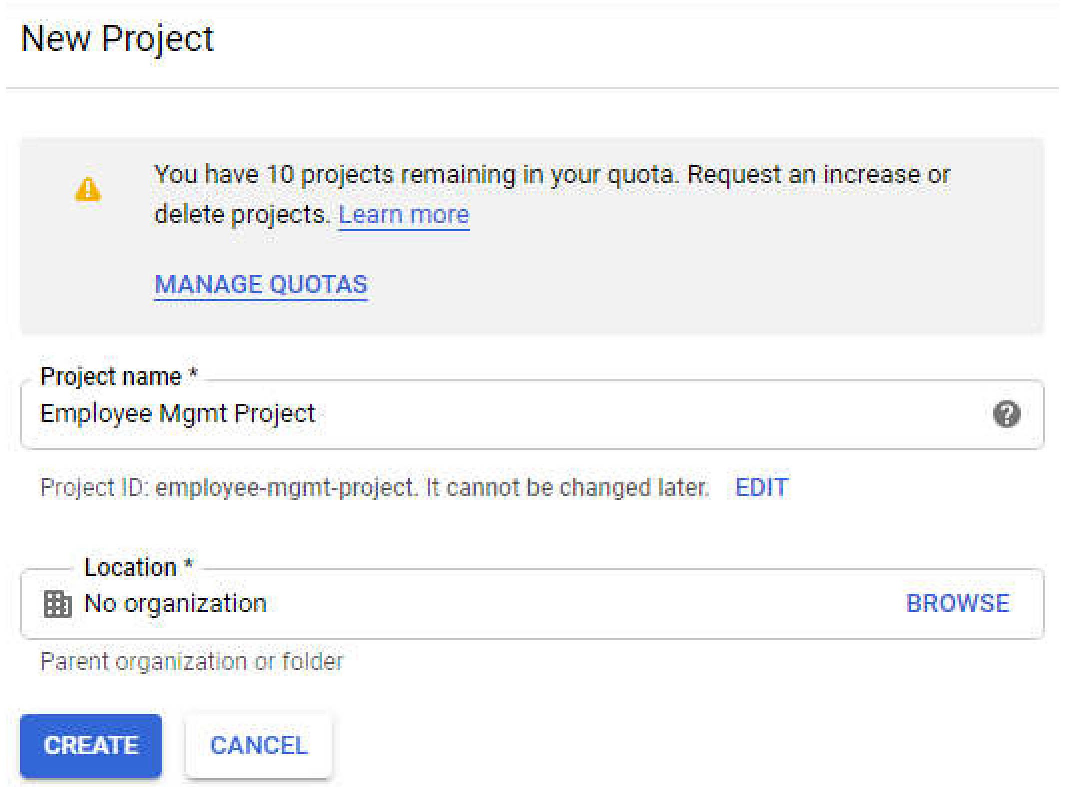
To use an external identity provider like **Google**, we have to first register our application with

Google. Upon successful registration we will be provided with **Client Id** and **Client Secret** which we need to use Google authentication.

Step 1 : Create a project if you do not have one already



Click on Select a project dropdownlist and then, click New Project link in the popup window that appears. Give your project a meaningful name and then click Create. It takes a few seconds to create the project.


New Project

You have 10 projects remaining in your quota. Request an increase or delete projects. Learn more

MANAGE QUOTAS

Project name *
Employee Mgmt Project

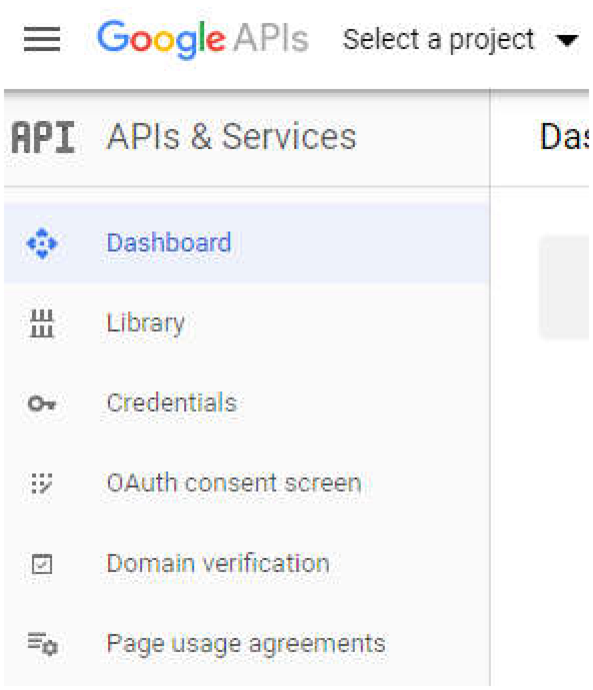
Project ID: employee-mgmt-project. It cannot be changed later. EDIT

Location *
No organization BROWSE

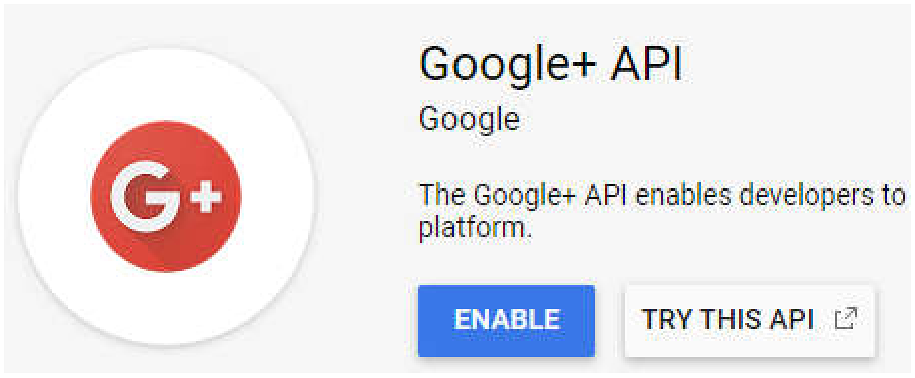
Parent organization or folder

CREATE CANCEL

Step 2 : Enable Google+ API

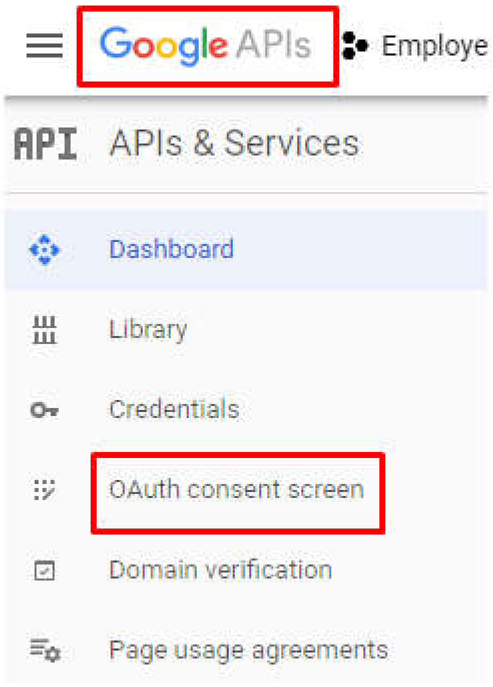


Click on the Library tab on the left and search for Googleplus API and enable it.



Step 3 : Configure OAuth consent screen

Click on the OAuth consent screen tab on the left. If you do not see OAuth consent screen tab, click on Google APIs banner image on the top left hand corner.

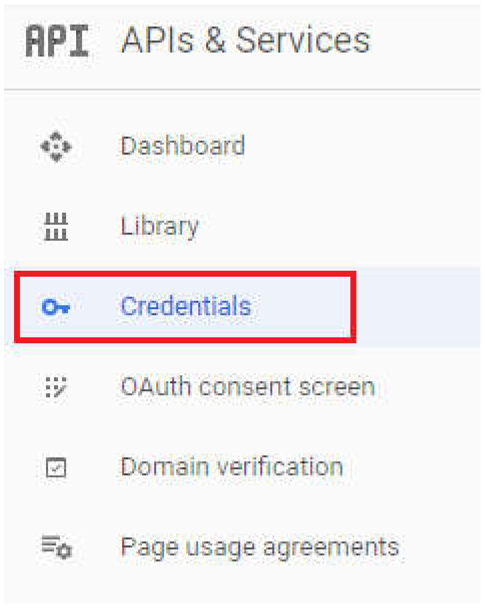


On the OAuth consent screen, the only required field is the Application name. This is the name that will be shown to end users asking for their consent.

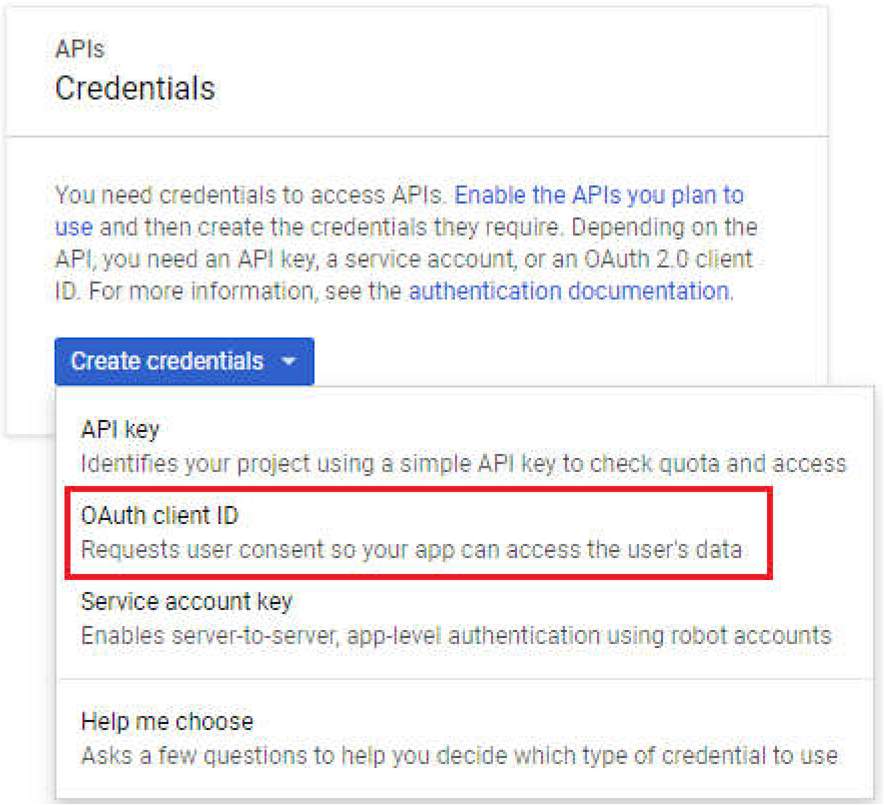
If this is not entirely clear at the moment, please do not worry. In our upcoming videos, when we actually integrate google authentication and see the consent screen in action it will be much clear at that point.

Step 4 : Create OAuth client credentials

Click on the Credentials tab on the left navigation menu.



On the subsequent page, click Create credentials button. From the dropdownlist, select OAuth client ID.


APIs
Credentials

You need credentials to access APIs. Enable the APIs you plan to use and then create the credentials they require. Depending on the API, you need an API key, a service account, or an OAuth 2.0 client ID. For more information, see the authentication documentation.

Create credentials

API key
Identifies your project using a simple API key to check quota and access

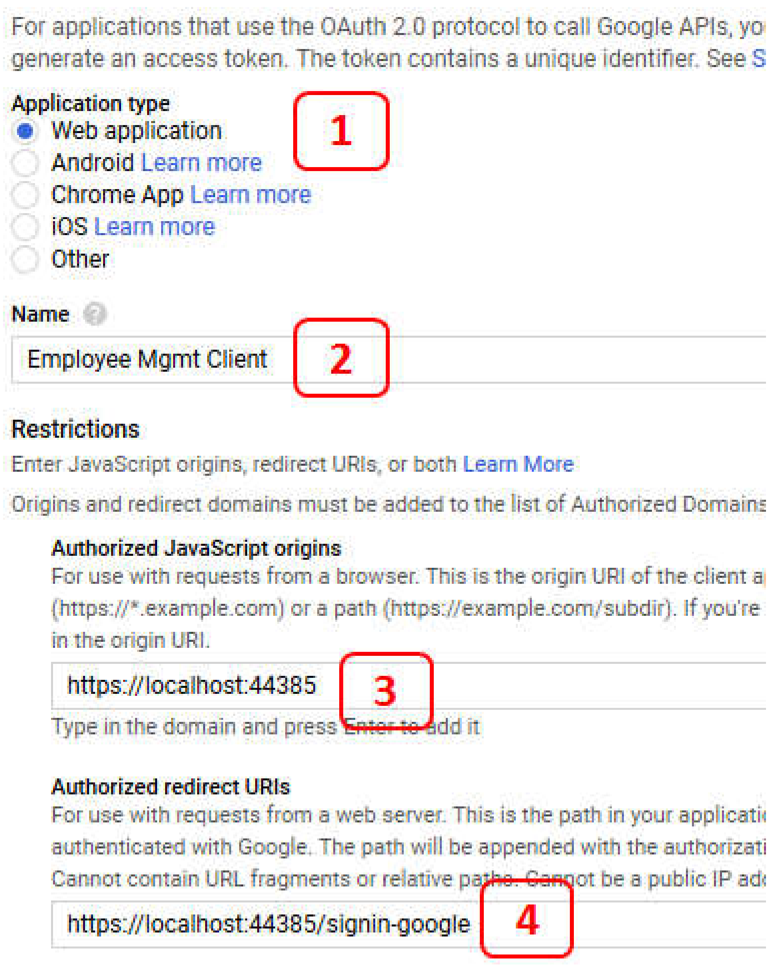
OAuth client ID
Requests user consent so your app can access the user's data

Service account key
Enables server-to-server, app-level authentication using robot accounts

Help me choose
Asks a few questions to help you decide which type of credential to use

On the next screen (i.e Create OAuth client ID)

- Select Web application as the Application type
- Provide a meaningful name for the OAuth client.
- **Authorized JavaScript origins** - This is the URL of where our application is running. To get this URL, on your localhost, right click on the project name in Solution Explorer in Visual Studio and select Properties. On the Debug tab, you will find the App URL.
- **Authorized redirect URIs** - This is the path in our application that users are redirected to after they are authenticated by Google. The default path in asp.net core is signin-google. So the complete redirect URI is Application Root URI/signin-google. If we do not like this default path signin-google we can change it. We will discuss how to do this in our next video, when we discuss integrating google authentication into our asp.net core application.


For applications that use the OAuth 2.0 protocol to call Google APIs, yo
generate an access token. The token contains a unique identifier. See S

Application type
[✓ Web application]
Android Learn more
Chrome App Learn more
iOS Learn more
Other

Name
Employee Mgmt Client

Restrictions
Enter JavaScript origins, redirect URIs, or both Learn More
Origins and redirect domains must be added to the list of Authorized Domains

Authorized JavaScript origins
For use with requests from a browser. This is the origin URI of the client a
(https://*.example.com) or a path (https://example.com/subdir). If you're
in the origin URI.
https://localhost:44385
Type in the domain and press Enter to add it

Authorized redirect URIs
For use with requests from a web server. This is the path in your applicati
authenticated with Google. The path will be appended with the authorizati
Cannot contain URL fragments or relative paths. Cannot be a public IP ad
https://localhost:44385/signin-google

Enable Google Authentication in ASP.NET Core

Include the following configuration in ConfigureServices() method of the Startup class. We discussed registering our application with Google and obtaining Client Id and Secret in our previous video.

```
services.AddAuthentication().AddGoogle(options =>
   {
      options.ClientId = "XXXXX";
      options.ClientSecret = "YYYYY";


   });
```

The code required for Google authentication including this AddGoogle() method is present in Microsoft.AspNetCore.Authentication.Google nuget package. Since I am using ASP.NET Core 2.2, this package is automatically included in the project as part of the meta package. If you are using older versions of ASP.NET Core you have to manually install this nuget package.

```
public class LoginViewModel
{
    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [Display(Name = "Remember me")]
    public bool RememberMe { get; set; }

    public string ReturnUrl { get; set; }

    // AuthenticationScheme is in Microsoft.AspNetCore.Authentication namespace
    public IList<AuthenticationScheme> ExternalLogins { get; set; }
}


HttpGet]
[AllowAnonymous]
public async Task<IActionResult> Login(string returnUrl)
{
    LoginViewModel model = new LoginViewModel
    {
        ReturnUrl = returnUrl,
        ExternalLogins =
        (await signInManager.GetExternalAuthenticationSchemesAsync()).ToList()
    };

    return View(model);
}
```

ReturnUrl is the URL the user was trying to access before authentication. We preserve and pass it between requests using ReturnUrl property, so the user can be redirected to that URL upon successful authentication.

ExternalLogins property stores the list of external logins (like Facebook, Google etc) that are enabled in our application. You will better understand what this property does in just a bit, when we actually use it.

Login Action in AccountController

- Populate ReturnUrl and ExternalLogins properties of LoginViewModel and then pass the instance to the view.
- GetExternalAuthenticationSchemesAsync() method of SignInManager service, returns the list of all configured external identity providers like (Google, Facebook etc).
- At the moment we only have one external identity provider configured and that is Google.

Login Page (Login.cshtml)

The following is the code specific to external login.

```
   <h1>External Login</h1>
    <hr />
    @{
        if (Model.ExternalLogins.Count == 0)
        {
            <div>No external logins configured</div>
        }
        else
        {
            <form method="post" asp-action="ExternalLogin" asp-route-
returnUrl="@Model.ReturnUrl">
                <div>
                    @foreach (var provider in Model.ExternalLogins)
                    {
                        <button type="submit" class="btn btn-primary"
                                name="provider" value="@provider.Name"
                                title="Log in using your @provider.DisplayName account">
                            @provider.DisplayName
                        </button>
                    }
                </div>
            </form>
        }
    }
```

- We are looping through each external login provider we have in Model.ExternalLogins
- For each external login provider a submit button is dynamically generated
- At the moment we only have one external identity provider configured and that is Google, so we get one Submit button.

- This submit button is inside a form. The form method attribute value is post and asp-action attribute value is ExternalLogin
- So when the submit button is clicked the form is posted to ExternalLogin action in AccountController
- The login provider is Google, so in the foreach loop, provider.Name returns Google.
- Since the button name is set to provider, asp.net core model binding maps the provider name which is Google to provider parameter on the ExternalLogin action.

ExternalLogin action in AccountController

```
[AllowAnonymous]
[HttpPost]
public IActionResult ExternalLogin(string provider, string returnUrl)
{
    var redirectUrl = Url.Action("ExternalLoginCallback", "Account",
                        new { ReturnUrl = returnUrl });
    var properties = signInManager
        .ConfigureExternalAuthenticationProperties(provider, redirectUrl);
    return new ChallengeResult(provider, properties);
}
```

Upon successful authentication, Google redirects the user back to our application and the following ExternalLoginCallback action is executed.

```
[AllowAnonymous]
public async Task<IActionResult>
            ExternalLoginCallback(string returnUrl = null, string remoteError = null)
{
    returnUrl = returnUrl ?? Url.Content("~/"); //the return url default is specified in authorize
redirect uri when Creating OAuth client credentials in step 4

    LoginViewModel loginViewModel = new LoginViewModel
    {
        ReturnUrl = returnUrl,
        ExternalLogins =
                (await signInManager.GetExternalAuthenticationSchemesAsync()).ToList()
    };

    if (remoteError != null)
    {
        ModelState
            .AddModelError(string.Empty, $"Error from external provider: {remoteError}");

        return View("Login", loginViewModel);
    }

    // Get the login information about the user from the external login provider
    var info = await signInManager.GetExternalLoginInfoAsync();
```

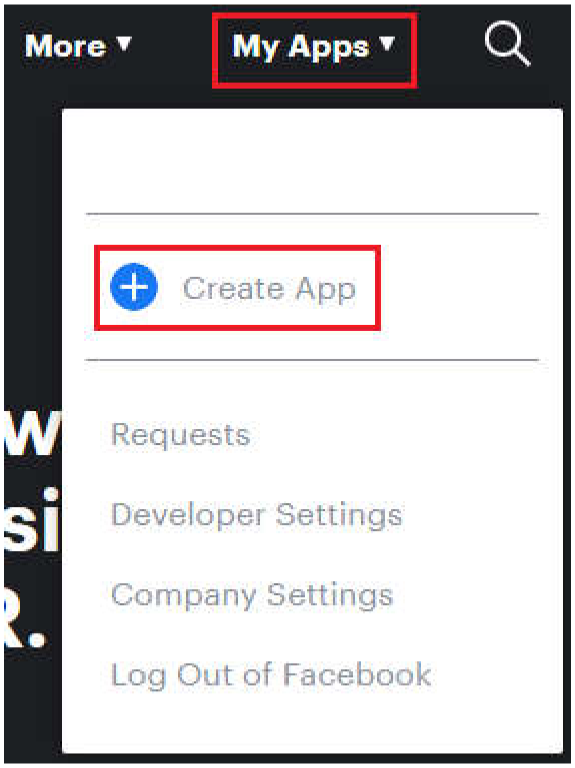
```
if (info == null)
{
    ModelState
        .AddModelError(string.Empty, "Error loading external login information.");

    return View("Login", loginViewModel);
}

// If the user already has a login (i.e if there is a record in AspNetUserLogins
// table) then sign-in the user with this external login provider
var signInResult = await signInManager.ExternalLoginSignInAsync(info.LoginProvider,
    info.ProviderKey, isPersistent: false, bypassTwoFactor: true);

if (signInResult.Succeeded)
{
    return LocalRedirect(returnUrl);
}
// If there is no record in AspNetUserLogins table, the user may not have
// a local account
else
{
    // Get the email claim value
    var email = info.Principal.FindFirstValue(ClaimTypes.Email);

    if (email != null)
    {
        // Create a new user without password if we do not have a user already
        var user = await userManager.FindByEmailAsync(email);

        if (user == null)
        {
            user = new ApplicationUser
            {
                UserName = info.Principal.FindFirstValue(ClaimTypes.Email),
                Email = info.Principal.FindFirstValue(ClaimTypes.Email)
            };

            await userManager.CreateAsync(user);
        }

        // Add a login (i.e insert a row for the user in AspNetUserLogins table)
        await userManager.AddLoginAsync(user, info);
        await signInManager.SignInAsync(user, isPersistent: false);

        return LocalRedirect(returnUrl);
    }

    // If we cannot find the user email we cannot continue
    ViewBag.ErrorTitle = $"Email claim not received from: {info.LoginProvider}";
    ViewBag.ErrorMessage = "Please contact support on Pragim@PragimTech.com";
```

```
        return View("Error");
    }
}
```

Facebook

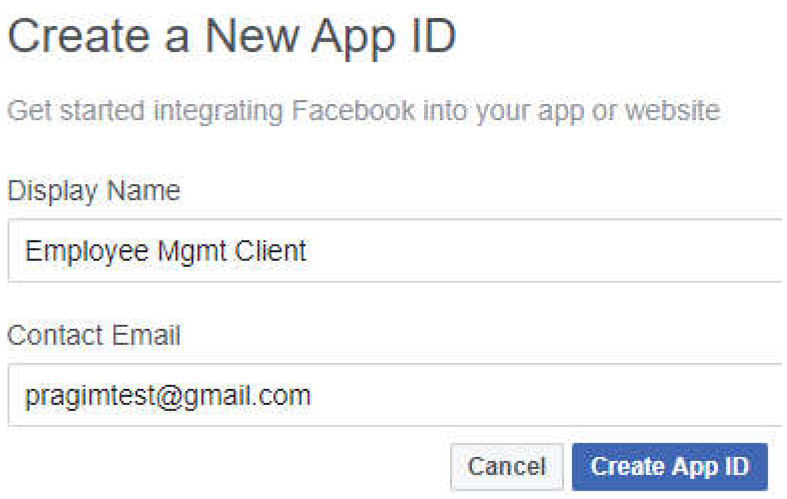
Step 1 : Log into https://developers.facebook.com

Step 2 : Click on My Apps dropdown and click Create App button



Step 3 : Create a New App ID

Specify a display name for the client application. This is the name that is displayed on the consent screen. The email associated with your Facebook account is in the Contact Email textbox. Click Create App ID button to create the app.


Create a New App ID

Get started integrating Facebook into your app or website

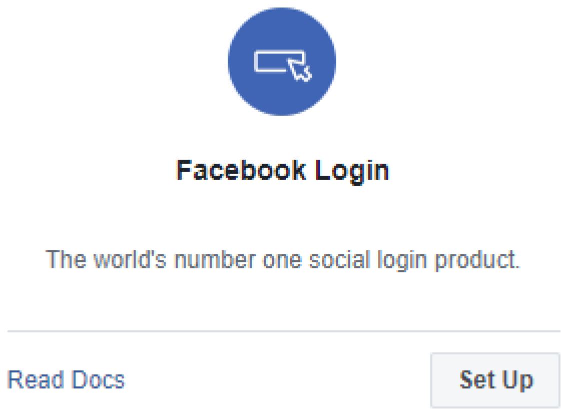
Display Name

Employee Mgmt Client

Contact Email

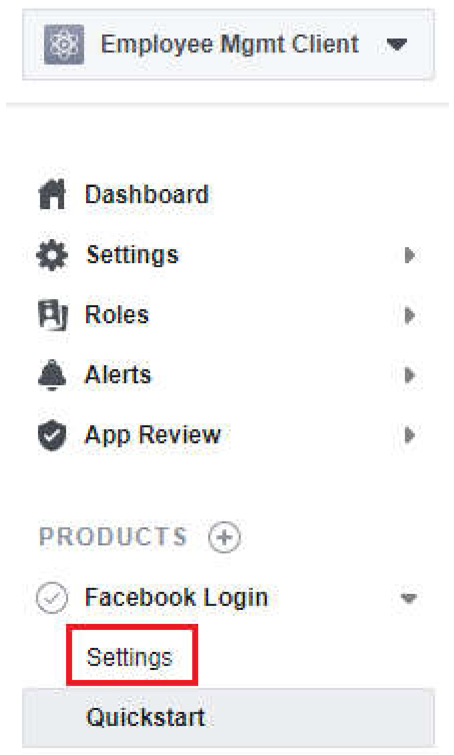
pragimtest@gmail.com

Step 4 : Click Setup button the Facebook Login product


Facebook Login

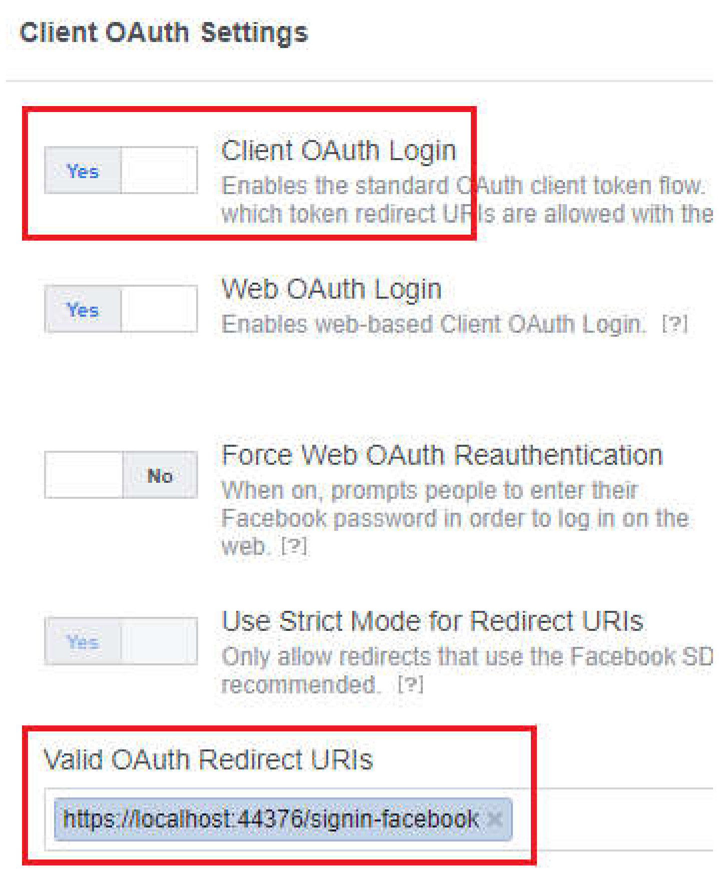
The world's number one social login product.

Step 5 : Click on the Settings tab under Facebook Login on the left navigation menu



Step 6 : Enable Client OAuth Login.

Also specify Valid OAuth Redirect URI. This is the URI at which your application is hosted. To this URI append that path segment /signin-facebook. Finally click Save Changes.



Client OAuth Settings

Yes Client OAuth Login
Enables the standard OAuth client token flow. which token redirect URIs are allowed with the

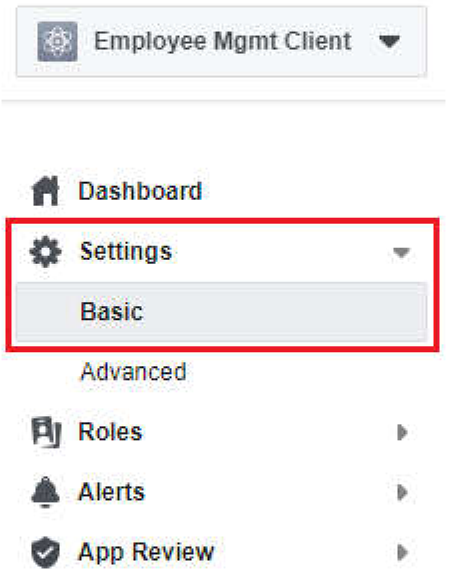
Yes Web OAuth Login
Enables web-based Client OAuth Login. [?]

No Force Web OAuth Reauthentication
When on, prompts people to enter their Facebook password in order to log in on the web. [?]

Yes Use Strict Mode for Redirect URIs
Only allow redirects that use the Facebook SD recommended. [?]

Valid OAuth Redirect URIs
https://localhost:44376/signin-facebook

Step 7 : On the left navigaton menu, click on the Basic tab under Settings to obtain App ID and App Secret



We need this App ID and App Secret to integrate Facebook authentication in our asp.net core application.

Enable Google authentication in ASP.NET Core

We configure and enable Google authentication in ConfigureServices() method of Startup class

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication()
        .AddGoogle(options =>
        {
            options.ClientId = "XXXXX";
            options.ClientSecret = "XXXXX";
        });
}
```

We use AddGoogle() extension method to configure Google authentication in ASP.NET Core

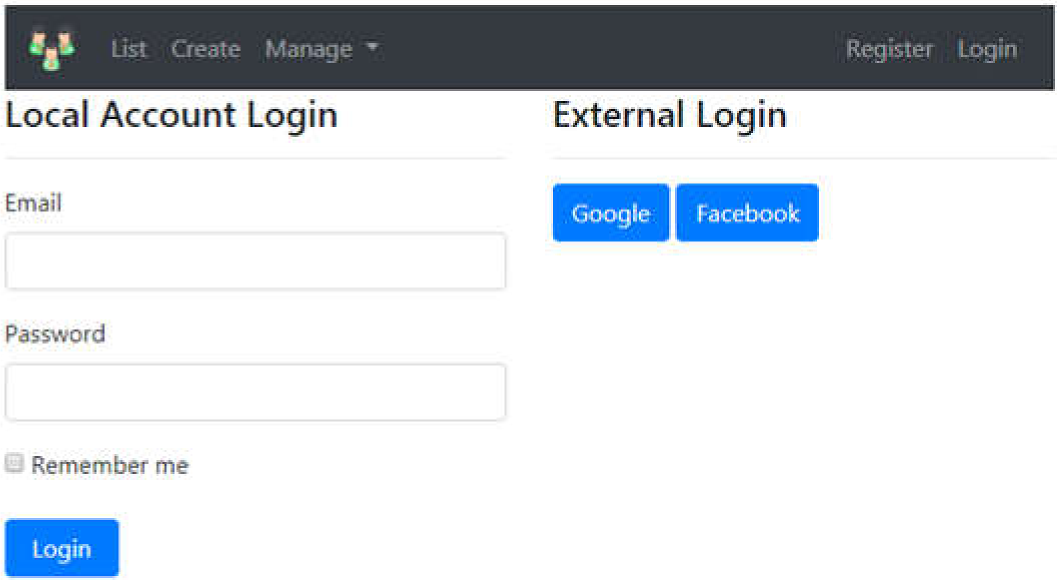
Enable Facebook authentication in ASP.NET Core

Just like AddGoogle(), we have AddFacebook() extension method to configure Facebook authentication in ASP.NET core.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication()
        .AddGoogle(options =>
        {
            options.ClientId = "XXXXX";
            options.ClientSecret = "XXXXX";
        })
        .AddFacebook(options =>
        {
            options.AppId = "XXXXX";
            options.AppSecret = "XXXXX";
        });
}
```

Login View

Since we have both Google and Facebook authentication integrated, we see the respective external login provider button automatically displayed on the Login view.



Local Account Login

Email

Password

Remember me

External Login

```
@model LoginViewModel

@{
    ViewBag.Title = "User Login";
}

<div class="row">
    <div class="col-md-6">
        <h1>Local Account Login</h1>
        <hr />
        <form method="post">
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Email"></label>
                <input asp-for="Email" class="form-control" />
                <span asp-validation-for="Email" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Password"></label>
                <input asp-for="Password" class="form-control" />
                <span asp-validation-for="Password" class="text-danger"></span>
            </div>
            <div class="form-group">
                <div class="checkbox">
                    <label asp-for="RememberMe">
                        <input asp-for="RememberMe" />
                        @Html.DisplayNameFor(m => m.RememberMe)
                    </label>
                </div>
            </div>
            <button type="submit" class="btn btn-primary">Login</button>
        </form>
    </div>

    <div class="col-md-6">
        <h1>External Login</h1>
        <hr />
        @{
            if (Model.ExternalLogins.Count == 0)
            {
                <div>No external logins configured</div>
            }
            else
            {
                <form method="post" asp-action="ExternalLogin" asp-route-
returnUrl="@Model.ReturnUrl">
                    <div>
                        @foreach (var provider in Model.ExternalLogins)
                        {
                            <button type="submit" class="btn btn-primary"
                                    name="provider" value="@provider.Name"
                                    title="Login usin your @provider.DisplayName account">
                                @provider.DisplayName
                            </button>
                        }
                    </div>
                </form>
            }
        }
    </div>
</div>
```

Redirect request to Facebook

When the Facebook button is clicked, our asp.net core application must redirect the request to Facebook for authentication. This is done by ExternalLogin() action in AccountController. The code in this method is written in a generic way, so it works for both Google and Facebook authentication.

```
[AllowAnonymous]
[HttpPost]
public IActionResult ExternalLogin(string provider, string returnUrl)
{
    var redirectUrl = Url.Action("ExternalLoginCallback", "Account",
                        new { ReturnUrl = returnUrl });

    var properties =
        signInManager.ConfigureExternalAuthenticationProperties(provider, redirectUrl);

    return new ChallengeResult(provider, properties);
}
```

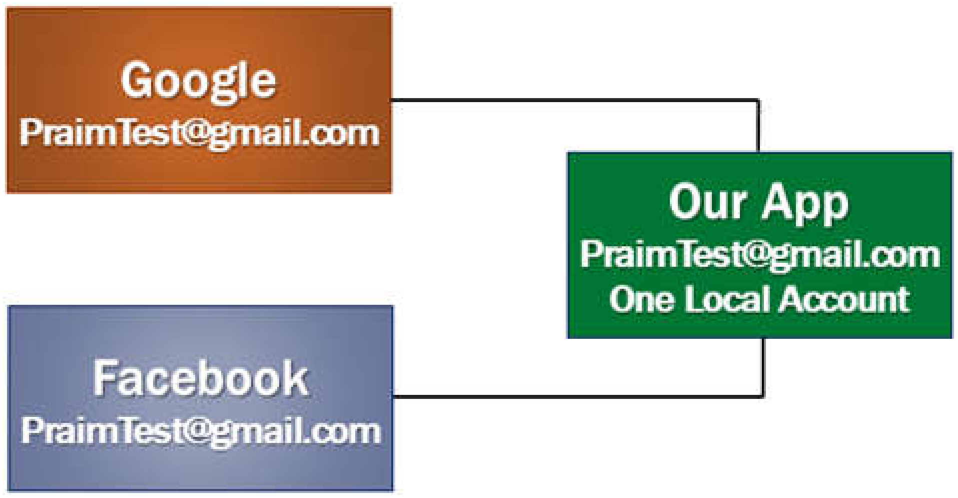
Handle External Login Information received from Facebook

After the user is successfully authenticated by Facebook, the request is redirected back to our application, and the following ExternalLoginCallback() action in AccountController is executed. The code in this method is also written in a generic way, so it works for both Google and Facebook authentication.

```
[AllowAnonymous]
public async Task<IActionResult>
    ExternalLoginCallback(string returnUrl = null, string remoteError = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");

    LoginViewModel loginViewModel = new LoginViewModel
    {
        ReturnUrl = returnUrl,
        ExternalLogins =
(await signInManager.GetExternalAuthenticationSchemesAsync()).ToList()
    };

    if (remoteError != null)
    {
        ModelState.AddModelError(string.Empty, $"Error from external provider: {remoteError}");

        return View("Login", loginViewModel);
    }

    var info = await signInManager.GetExternalLoginInfoAsync();
    if (info == null)
    {
        ModelState.AddModelError(string.Empty, "Error loading external login information.");

        return View("Login", loginViewModel);
    }

    var signInResult = await signInManager.ExternalLoginSignInAsync(info.LoginProvider,
                            info.ProviderKey, isPersistent: false, bypassTwoFactor: true);

    if (signInResult.Succeeded)
    {
        return LocalRedirect(returnUrl);
    }
    else
    {
        var email = info.Principal.FindFirstValue(ClaimTypes.Email);

        if (email != null)
        {
            var user = await userManager.FindByEmailAsync(email);

            if (user == null)
            {
                user = new ApplicationUser
                {
                    UserName = info.Principal.FindFirstValue(ClaimTypes.Email),
                    Email = info.Principal.FindFirstValue(ClaimTypes.Email)
                };

                await userManager.CreateAsync(user);
            }

            await userManager.AddLoginAsync(user, info);
            await signInManager.SignInAsync(user, isPersistent: false);

            return LocalRedirect(returnUrl);
        }

        ViewBag.ErrorTitle = $"Email claim not received from: {info.LoginProvider}";
        ViewBag.ErrorMessage = "Please contact support on Pragim@PragimTech.com";

        return View("Error");
    }
}
```

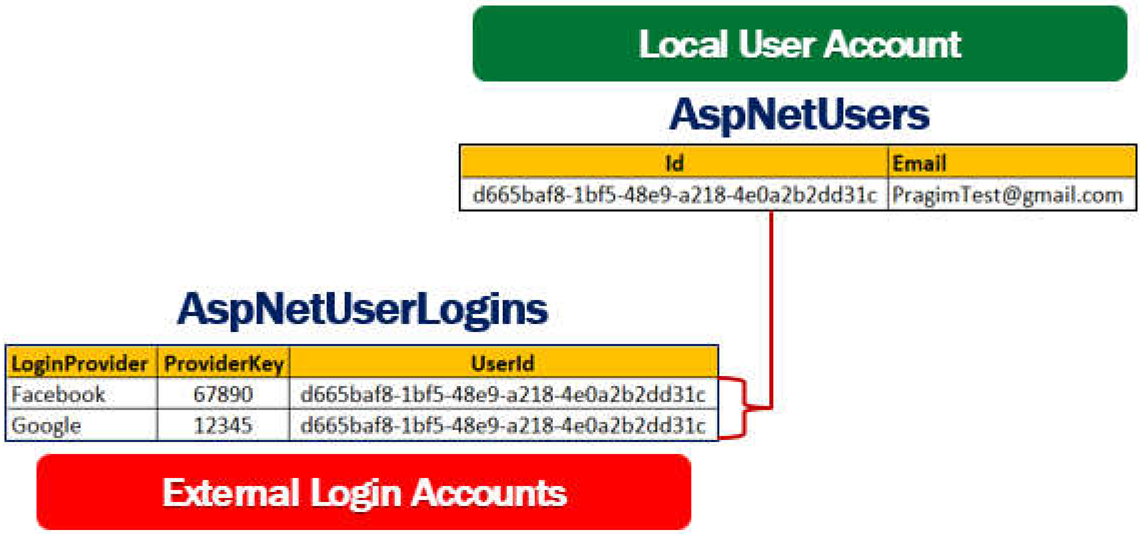
Linking external accounts to a local user account

In this example, the user with email (PragimTest@gmail.com) has an account with both Facebook and Google. He can use either of these external accounts to log into our application.



We want to link both these external accounts (Facebook and Google) to the same local account in our asp.net core application. Local user accounts are in AspNetUsers table and external login accounts are in AspNetUserLogins table.

Notice, for the user with email (PragimTest@gmail.com), there are 2 rows in AspNetUserLogins table. One row is for the Facebook login and the other for Google login. Both these rows link to the one row in AspNetUsers table.



Id	Email
d665baf8-1bf5-48e9-a218-4e0a2b2dd31c	PragimTest@gmail.com

LoginProvider	ProviderKey	UserId
Facebook	67890	d665baf8-1bf5-48e9-a218-4e0a2b2dd31c
Google	12345	d665baf8-1bf5-48e9-a218-4e0a2b2dd31c

User Secrets
The main use of Secret Manager is to keep production secrets like database connection strings, API and encryption keys out of source control.

Why you should not store secrets in configuration files

We usually store database connection strings, third party service credentials, API and encryption keys in configuration files like web.config in class asp.net and appSettings.json in asp.net core.

These configuration files are part of the project. So when they are committed to the source control repository, everyone who has access to the repository will have access to the sensitive data in these files and could be misused.

From security standpoint, it is not a good idea, to store passwords or other sensitive data in configuration files or source code.

Use of Secret Manager

Secret Manager allows developers to store and retrieve sensitive data during the development of an ASP.NET Core application. It stores sensitive data i.e user secrets in a file with name secrets.json.

To add this file to your project, right click on the project name in Solution Explorer in Visual Studio and select Manage User Secrets from the context menu. This adds secrets.json file.

The structure of this file is similar to appSettings.json. The important point to keep in mind is, this file is not part of the project folder. It is located outside of the project folder at the following path.

C:\Users\{UserName}\AppData\Roaming\Microsoft\UserSecrets\{ID}

- {UserName} is the windows user name that you use to log into the computer.
- {ID} is a GUID (Globally Unique Identifier)

On a single computer you may have multiple asp.net core projects and a secrets.json file for each project. It is this GUID, that links a given secrets.json file to a given asp.net core project. To establish this link, UserSecretsId node is included in the .csproj file.

```
<PropertyGroup>
  <UserSecretsId>490dfb7b-5991-4b34-bdd0-f961453843ef</UserSecretsId>
</PropertyGroup>
```

A given secrets.json file can be shared by multiple projects

Using Secret Manager to store database connection string

From best practices standpoint, we do not want to store database connecting strings anymore in appSettings.json file. So, move the following database connection string from appSettings.json file to secrets.json file.

```
{
  "ConnectionStrings": {
    "EmployeeDBConnection":
      "server=(localdb)\\MSSQLLocalDB;database=EmployeeDB;Trusted_Connection=true"
  }
}
```

Access secrets from Secrets.json file

In ASP.NET Core application configuration settings can come from different configuration sources like

1. appsettings.json
2. User secrets
3. Environment variables
4. Command-line arguments

We discussed appSettings.json file in Part 9 and Environment variables in Part 14 of ASP.NET Core tutorial.

Out of the box, IConfiguration service is setup to read configuration information from all the various configuration sources in asp.net core. For example, to read the database connection string from secrets.json file, inject and use IConfiguration service.

```
public class Startup
{
    private IConfiguration _config;
```

```
  public Startup(IConfiguration config)
  {
     _config = config;
  }

  public void ConfigureServices(IServiceCollection services)
  {
     services.AddDbContextPool<AppDbContext>(options =>
     options.UseSqlServer(_config.GetConnectionString("EmployeeDBConnection")));

     // Rest of the code
  }
}
```

Please note that, if you have a configuration setting with the same key in multiple configuration sources, the later configuration sources override the earlier configuration sources

CreateDefaultBuilder() method of the WebHost class which is automatically invoked when the application starts, reads the configuration sources in a specific order. To see the order in which the configuration sources are read, please check out ConfigureAppConfiguration() method on the following link
https://github.com/aspnet/MetaPackages/blob/release/2.2/src/Microsoft.AspNetCore/WebHost.cs

User secrets in production

To protect sensitive data, secrets.json file is deliberately kept outside of the project folder. This file is not checked into source control repository. This means secrets.json file is not copied onto the production server, when we actually build and deploy. So, where will the application find database connection string.

Well, on a production server store the database connection string in an environment variable. If you remember, IConfiguration service is setup to read configuration information from all the following configuration sources.

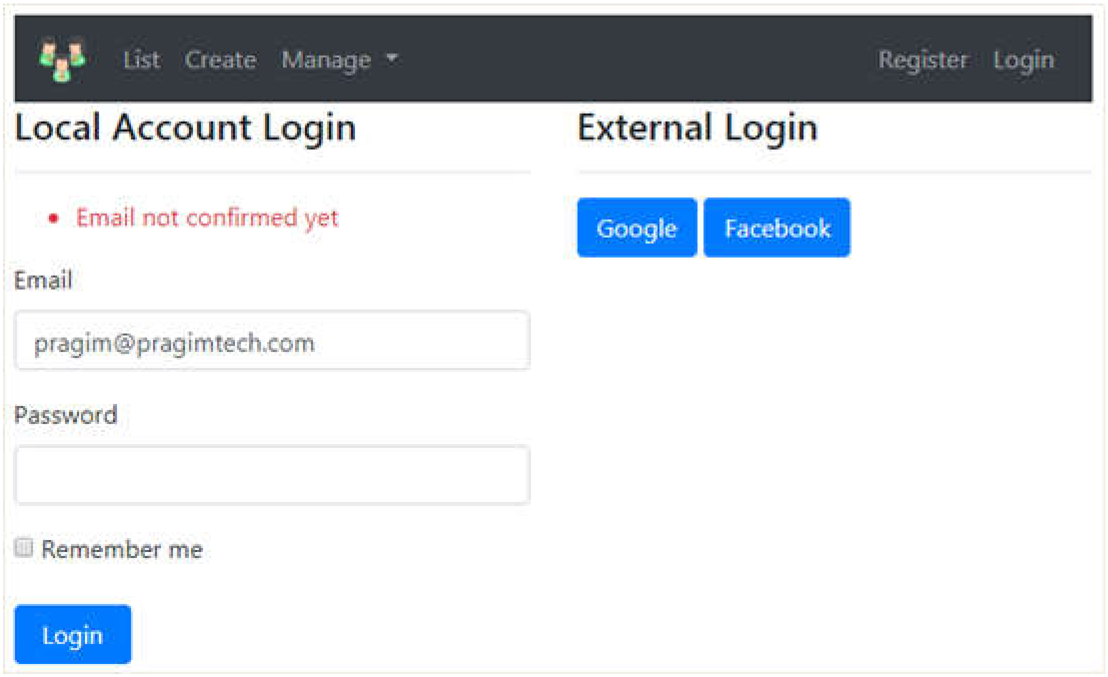
- appsettings.json
- User secrets
- Environment variables
- Command-line argument

This means, in spite of not having secrets.json file on the production server, our application should work just fine because it will find the required database connection string in the environment variable.

Secret Manager isn't for staging or production server, it should only be used on development machine. For production always use either environment variables, Azure Key Vault, or 3rd party production secret management system.

Email Confirmation

If email address is not confirmed and if the user tries to login we want to display the validation error - Email not confirmed yet



Local Account Login

- Email not confirmed yet

Email

pragim@pragimtech.com

Password

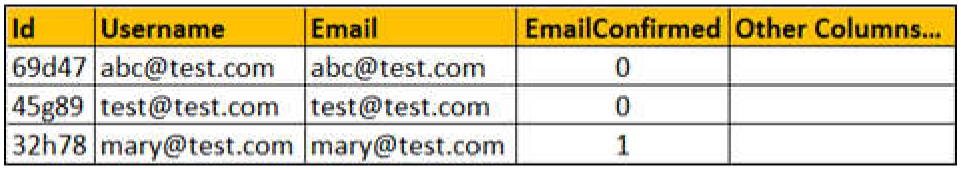
Remember me

Login

External Login

Google Facebook

In ASP.NET core, application users are stored in AspNetUsers table. EmailConfirmed column in this table is used to determine if a given email address is confirmed.



Id	Username	Email	EmailConfirmed	Other Columns...
69d47	abc@test.com	abc@test.com	0	
45g89	test@test.com	test@test.com	0	
32h78	mary@test.com	mary@test.com	1	

In this video we will discuss how to block a login if email address is not confirmed and in our next video we will implement email confirmation.

In ConfigureServices() method of the Startup class, set RequireConfirmedEmail property to true.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>(options =>
    {
        options.SignIn.RequireConfirmedEmail = true;
    })
    .AddEntityFrameworkStores<AppDbContext>();
}
```

Let's say RequireConfirmedEmail property is set to true and the email address is not confirmed yet. If we now use the SignInManager service PasswordSignInAsync() method to sign-in the user we get NotAllowed as the result, even if we supply the correct username and password.

The same is true with ExternalLoginSignInAsync() method of SignInManager service. We use ExternalLoginSignInAsync() method to sign-in the user using an external login provider like Facebook, Google etc. If the email address associated with external login account is not confirmed, signin result will be NotAllowed.

The following Login action in AccountController, blocks the login and displays Email not confirmed yet error.

```
[HttpPost]
[AllowAnonymous]
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl)
{
    model.ExternalLogins =
        (await signInManager.GetExternalAuthenticationSchemesAsync()).ToList();

    if (ModelState.IsValid)
    {
        var user = await userManager.FindByEmailAsync(model.Email);

        if (user != null && !user.EmailConfirmed &&
                    (await userManager.CheckPasswordAsync(user, model.Password)))
        {
            ModelState.AddModelError(string.Empty, "Email not confirmed yet");
            return View(model);
        }

        var result = await signInManager.PasswordSignInAsync(model.Email,
                                model.Password, model.RememberMe, false);

        if (result.Succeeded)
        {
            if (!string.IsNullOrEmpty(returnUrl) && Url.IsLocalUrl(returnUrl))
            {
                return Redirect(returnUrl);
            }
            else
            {
                return RedirectToAction("index", "home");
            }
        }

        ModelState.AddModelError(string.Empty, "Invalid Login Attempt");
    }

    return View(model);
}
```

This error message is displayed only, if the Email is not confirmed AND the user has provided correct username and password.

If you are wondering, why do we need to check if the user has provided correct username and password. Well, this is to avoid account enumeration and brute force attacks.

Let's understand, what might happen if we display this validation message - Email not confirmed yet, without checking if the provided email address and password combination is correct. An attacker might try random emails and as soon as he sees the message, Email not confirmed yet, he knows this is a valid email that could be used to login. He waits for couple of days until the email is confirmed by the actual owner, and can then try random passwords with that email address to gain access.

In order to avoid these types of account enumeration and brute force attacks, display the validation error, only upon providing the correct email address and password combination.

We also want to block the login if an external login account (like Facebook, Google etc) is used and the email address associated with that external account is not confirmed. The section that blocks the login is commented.

```
[AllowAnonymous]
public async Task<IActionResult>
    ExternalLoginCallback(string returnUrl = null, string remoteError = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");

    LoginViewModel loginViewModel = new LoginViewModel
    {
        ReturnUrl = returnUrl,
        ExternalLogins =
        (await signInManager.GetExternalAuthenticationSchemesAsync()).ToList()
    };

    if (remoteError != null)
    {
        ModelState.AddModelError(string.Empty,
            $"Error from external provider: {remoteError}");
        return View("Login", loginViewModel);
    }

    var info = await signInManager.GetExternalLoginInfoAsync();
    if (info == null)
    {
        ModelState.AddModelError(string.Empty,
            "Error loading external login information.");

        return View("Login", loginViewModel);
    }

    // Get the email claim from external login provider (Google, Facebook etc)
    var email = info.Principal.FindFirstValue(ClaimTypes.Email);
    ApplicationUser user = null;

    if (email != null)
    {
        // Find the user
        user = await userManager.FindByEmailAsync(email);

        // If email is not confirmed, display login view with validation error
        if (user != null && !user.EmailConfirmed)
        {
            ModelState.AddModelError(string.Empty, "Email not confirmed yet");
            return View("Login", loginViewModel);
        }
    }

    var signInResult = await signInManager.ExternalLoginSignInAsync(info.LoginProvider,
        info.ProviderKey, isPersistent: false, bypassTwoFactor: true);

    if (signInResult.Succeeded)
    {
        return LocalRedirect(returnUrl);
    }
    else
    {
        if (email != null)
        {
            if (user == null)
            {
                user = new ApplicationUser
                {
                    UserName = info.Principal.FindFirstValue(ClaimTypes.Email),
                    Email = info.Principal.FindFirstValue(ClaimTypes.Email)
                };

                await userManager.CreateAsync(user);
            }
            await userManager.AddLoginAsync(user, info);
            await signInManager.SignInAsync(user, isPersistent: false);

            return LocalRedirect(returnUrl);
        }

        ViewBag.ErrorTitle = $"Email claim not received from: {info.LoginProvider}";
        ViewBag.ErrorMessage = "Please contact support on Pragim@PragimTech.com";

        return View("Error");
    }
}
```

Please note : With the external login the user does not provide username and password to our application. Upon successful authentication, the user is redirected to the ExternalLoginCallback() action in our application. So we know the user is already authenticated and hence we display the validation error - Email not confirmed, without the need to check if the provided username and password combination is correct.

Generate email confirmation token in asp.net core

In ASP.NET core generating email confirmation token is straight forward. Use UserManager service GenerateEmailConfirmationTokenAsync() method. This method takes one parameter. The user for whom we want to generate the email confirmation token.

```
var token = await userManager.GenerateEmailConfirmationTokenAsync(user);
```

Build the email confirmation link

Once we have the token generated, build the email confirmation link. The user simply clicks this link to confirm his email. This link executes, ConfirmEmail action in Account controller. The user ID and the email confirmation token are passed in the query string. Model binding in ASP.NET core maps the values from the query string parameters to the respective parameters on the ConfirmEmail action.

```
var confirmationLink = Url.Action("ConfirmEmail", "Account",
    new { userId = user.Id, token = token }, Request.Scheme);
```

The generated confirmation link would look like the following

```
https://localhost:44304/Account/ConfirmEmail?userId=987009e3-7f78-445e-8bb8-
4400ba886550&token=CfDJ8Hpirs
```

The last parameter Request.Scheme returns the request protocol such as Http or Https. This parameter is required to generate the full absolute URL. If this parameter is not specified, a relative URL like the following will be generated.

```
/Account/ConfirmEmail?userId=987009e3-7f78-445e-8bb8-4400ba886550&token=CfDJ8Hpirs
```

Confirming the email

Use ConfirmEmailAsync() method of the UserManager service to confirm the email. To this method we pass 2 parameters. The user whose email we want to confirm and them email confirmation token. Upon successful email confirmation, this method sets EmailConfirmed column to True in AspNetUsers table.

```
var result = await userManager.ConfirmEmailAsync(user, token);
```

Add ASP.NET core default token providers

In ConfigureServices() method of the Startup class, call AddDefaultTokenProviders() method to add the asp.net core default token providers that generate tokens for email confirmation, password reset, two factor authentication etc.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>(options =>
    {
        options.SignIn.RequireConfirmedEmail = true;
    })
    .AddEntityFrameworkStores<AppDbContext>()
    .AddDefaultTokenProviders();
}
```

You should add, either the default token providers or your own custom token providers that can generate tokens. Otherwise you would get the following runtime exception.

```
NotSupportedException: No IUserTwoFactorTokenProvider<TUser> named 'Default' is
registered.
```

Register and ConfirmEmail actions

```
public class AccountController : Controller
{
    private readonly UserManager<ApplicationUser> userManager;
    private readonly SignInManager<ApplicationUser> signInManager;
    private readonly ILogger<AccountController> logger;

    public AccountController(UserManager<ApplicationUser> userManager,
                            SignInManager<ApplicationUser> signInManager,
                            ILogger<AccountController> logger)
    {
        this.userManager = userManager; this.signInManager = signInManager;
        this.logger = logger;
    }

    [HttpPost]
    [AllowAnonymous]
    public async Task<IActionResult> Register(RegisterViewModel model, string returnUrl)
    {
        if (ModelState.IsValid)
        {
            var user = new ApplicationUser
            {
                UserName = model.Email,
                Email = model.Email,
                City = model.City,
            };

            var result = await userManager.CreateAsync(user, model.Password);

            if (result.Succeeded)
            {
                var token = await userManager.GenerateEmailConfirmationTokenAsync(user);

                var confirmationLink = Url.Action("ConfirmEmail", "Account",
                    new { userId = user.Id, token = token }, Request.Scheme);


                if (signInManager.IsSignedIn(User) && User.IsInRole("Admin"))
                {
                    return RedirectToAction("ListUsers", "Administration");
                }

                ViewBag.ErrorTitle = "Registration successful";
                ViewBag.ErrorMessage = "Before you can Login, please confirm your " +
                        "email, by clicking on the confirmation link we have emailed you";
                return View("Error");
            }

            foreach (var error in result.Errors)
            {
                ModelState.AddModelError(string.Empty, error.Description);
            }
        }

        return View(model);
    }

    [AllowAnonymous]
    public async Task<IActionResult> ConfirmEmail(string userId, string token)
    {
        if (userId == null || token == null)
        {
            return RedirectToAction("index", "home");
        }

        var user = await userManager.FindByIdAsync(userId);
        if (user == null)
        {
            ViewBag.ErrorMessage = $"The User ID {userId} is invalid";
            return View("NotFound");
        }

        var result = await userManager.ConfirmEmailAsync(user, token);
        if (result.Succeeded)
        {
            return View();
        }

        ViewBag.ErrorTitle = "Email cannot be confirmed";
        return View("Error");
    }

    // rest of the code
}
```

ConfirmEmail View

```
<h3>Thank you for confirming your email</h3>
```

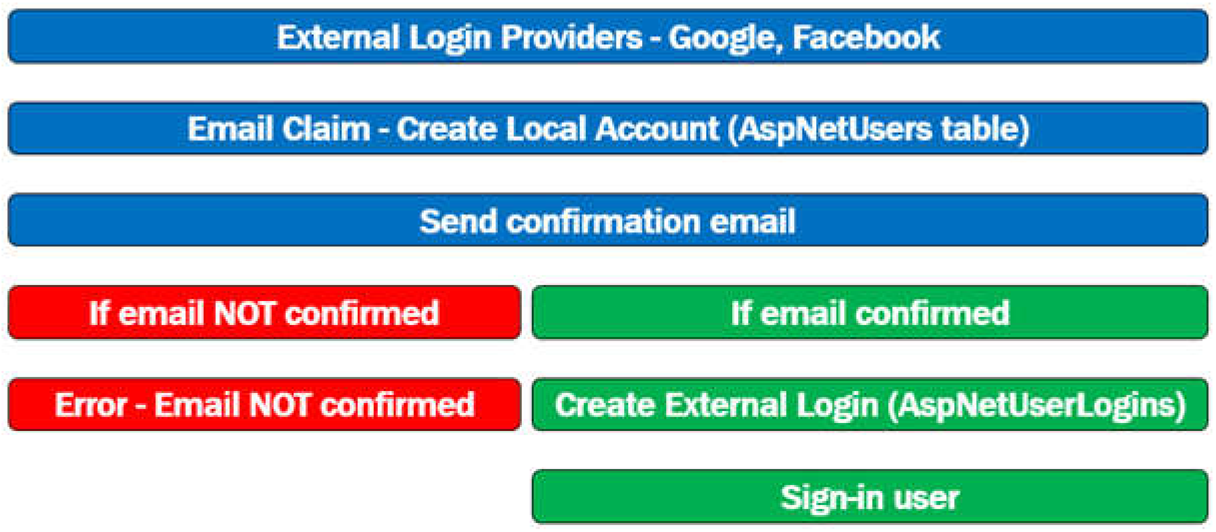
External login email confirmation in asp net core

we will discuss **how to confirm the email received from external login providers such as Google, Facebook etc**.

When we use external providers like Google or Facebook to login, we receive the user email from these external login provider. We then use this email to create a local user account.

Upon creating a local user account, send email confirmation link. If the email address is not confirmed, do not allow the user to login and display the error - Email not confirmed yet.

On the other hand, if the email address is confirmed, create an external login (AspNetUserLogins) and sign-in the user.


External Login Providers - Google, Facebook
Email Claim - Create Local Account (AspNetUsers table)
Send confirmation email
If email NOT confirmed
If email confirmed
Error - Email NOT confirmed
Create External Login (AspNetUserLogins)
Sign-in user

```
[AllowAnonymous]
public async Task<IActionResult>
ExternalLoginCallback(string returnUrl = null, string remoteError = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");

    LoginViewModel loginViewModel = new LoginViewModel
    {
        ReturnUrl = returnUrl,
        ExternalLogins =
        (await signInManager.GetExternalAuthenticationSchemesAsync()).ToList()
    };

    if (remoteError != null)
    {
        ModelState.AddModelError(string.Empty,
            $"Error from external provider: {remoteError}");

        return View("Login", loginViewModel);
    }

    var info = await signInManager.GetExternalLoginInfoAsync();
    if (info == null)
    {
        ModelState.AddModelError(string.Empty,
            "Error loading external login information.");

        return View("Login", loginViewModel);
    }

    var email = info.Principal.FindFirstValue(ClaimTypes.Email);
    ApplicationUser user = null;

    if (email != null)
```

```
{
    user = await userManager.FindByEmailAsync(email);

    if (user != null && !user.EmailConfirmed)
    {
        ModelState.AddModelError(string.Empty, "Email not confirmed yet");
        return View("Login", loginViewModel);
    }
}

var signInResult = await signInManager.ExternalLoginSignInAsync(
                            info.LoginProvider, info.ProviderKey,
                            isPersistent: false, bypassTwoFactor: true);

if (signInResult.Succeeded)
{
    return LocalRedirect(returnUrl);
}
else
{
    if (email != null)
    {
        if (user == null)
        {
            user = new ApplicationUser
            {
                UserName = info.Principal.FindFirstValue(ClaimTypes.Email),
                Email = info.Principal.FindFirstValue(ClaimTypes.Email)
            };

            await userManager.CreateAsync(user);

            // After a local user account is created, generate and log the
            // email confirmation link
            var token = await userManager.GenerateEmailConfirmationTokenAsync(user);

            var confirmationLink = Url.Action("ConfirmEmail", "Account",
                            new { userId = user.Id, token = token }, Request.Scheme);

            logger.Log(LogLevel.Warning, confirmationLink);

            ViewBag.ErrorTitle = "Registration successful";
            ViewBag.ErrorMessage = "Before you can Login, please confirm your " +
                "email, by clicking on the confirmation link we have emailed you";
            return View("Error");
        }

        await userManager.AddLoginAsync(user, info);
        await signInManager.SignInAsync(user, isPersistent: false);

        return LocalRedirect(returnUrl);
        }

        ViewBag.ErrorTitle = $"Email claim not received from: {info.LoginProvider}";
        ViewBag.ErrorMessage = "Please contact support on Pragim@PragimTech.com";

        return View("Error");
    }
}
```

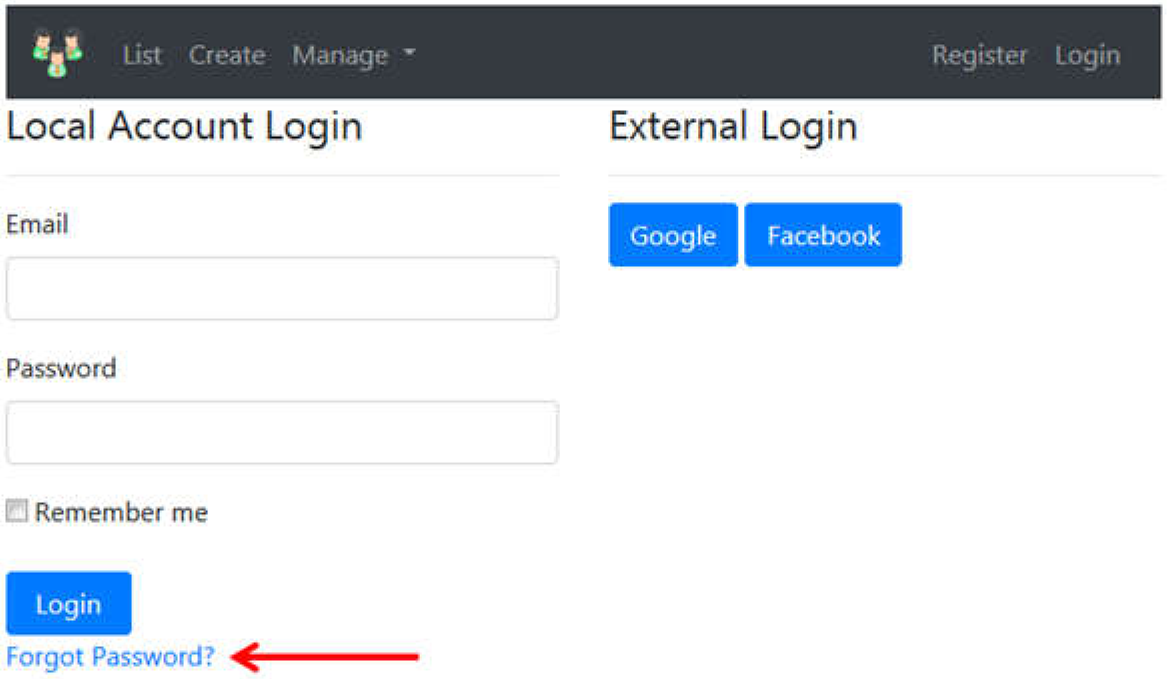
```
[HttpGet]
[AllowAnonymous]
public async Task<IActionResult> ConfirmEmail(string userId, string token)
{
    if (userId == null || token == null)
    {
        return RedirectToAction("index", "home");
    }

    var user = await userManager.FindByIdAsync(userId);

    if (user == null)
    {
        ViewBag.ErrorMessage = $"The User ID {userId} is invalid";
        return View("NotFound");
    }

    var result = await userManager.ConfirmEmailAsync(user, token);

    if (result.Succeeded)
    {
        return View();
    }

    ViewBag.ErrorTitle = "Email cannot be confirmed";
    return View("Error");
}
```

Forgot password in asp net core

we will discuss **how to implement forgot password functionality in asp.net core mvc.**

Forgot Password link

The first step is to include the Forgot Password link on the Login view.



The following is the HTML for that

```
<div>
   <a asp-action="ForgotPassword">Forgot Password?</a>
</div>
```

Forgot Password View Model

We just need the user email address to send the password reset link. So the ForgotPasswordViewModel class contains just one property - Email

```
namespace EmployeeManagement.ViewModels
{
    public class ForgotPasswordViewModel
    {
        [Required]
        [EmailAddress]
        public string Email { get; set; }
    }
}
```

Forgot Password Action Methods

Include the following HTTP GET and POST ForgotPassword() actions in the Account controller. The code in the actions is commented and self-explanatory.

```
[HttpGet]
[AllowAnonymous]
public IActionResult ForgotPassword()
{
    return View();
}
```

```
[HttpPost]
[AllowAnonymous]
public async Task<IActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        // Find the user by email
        var user = await userManager.FindByEmailAsync(model.Email);
        // If the user is found AND Email is confirmed
        if (user != null && await userManager.IsEmailConfirmedAsync(user))
        {
            // Generate the reset password token
            var token = await userManager.GeneratePasswordResetTokenAsync(user);

            // Build the password reset link
            var passwordResetLink = Url.Action("ResetPassword", "Account",
                    new { email = model.Email, token = token }, Request.Scheme);

            // Log the password reset link
            logger.Log(LogLevel.Warning, passwordResetLink);

            // Send the user to Forgot Password Confirmation view
            return View("ForgotPasswordConfirmation");
        }

        // To avoid account enumeration and brute force attacks, don't
        // reveal that the user does not exist or is not confirmed
        return View("ForgotPasswordConfirmation");
    }
    return View(model);
}
```

Forgot Password View

```
@model ForgotPasswordViewModel

<h2>Forgot Password</h2>
<hr />
<div class="row">
    <div class="col-md-12">
        <form method="post">
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Email"></label>
                <input asp-for="Email" class="form-control" />
                <span asp-validation-for="Email" class="text-danger"></span>
            </div>
            <button type="submit" class="btn btn-primary">Submit</button>
        </form>
    </div>
</div>
```

Forgot Password Confirmation View

```
<h4>
    If you have an account with us, we have sent an email
    with the instructions to reset your password.
</h4>
```

At this point, run the project. Navigate to /Account/ForgotPassword and provide your registered email address. Upon submitting the form, the password reset link will be logged to a file and looks like the following.

https://localhost:44305/Account/ResetPassword?email=pragim@pragimtech.com&token=CfDJ8HpirsZUXNxBvU8n%2...

At the moment, if we try to use the password reset link, we get a 404 error. This is because we do not have ResetPassword() action in the AccountController. We will discuss how to implement this in our next video.

Passowrd Reset

we will discuss, how to implement **password reset functionality in asp.net core**. This is continuation to our previous video Part 115. Please watch Part 115 from asp.net core tutorial before proceeding.

To be able to reset the user password we need the following

1. Email,
2. Password reset token,
3. New Password and
4. Confirm Password

The user provides the new password and confirmation password. Email and reset token are in the password reset link.

Reset Password View Model

```
namespace EmployeeManagement.ViewModels
{
    public class ResetPasswordViewModel
    {
        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }

        [DataType(DataType.Password)]
        [Display(Name = "Confirm password")]
        [Compare("Password",
            ErrorMessage = "Password and Confirm Password must match")]
        public string ConfirmPassword { get; set; }

        public string Token { get; set; }
    }
}
```

Reset Password View

We are using 2 hidden input fields to store email address and password reset token as we need them on postback.

```
@model ResetPasswordViewModel


<h2>Reset Password</h2>

<hr />

<div class="row">

    <div class="col-md-12">

        <form method="post">

            <div asp-validation-summary="All" class="text-danger"></div>

            <input asp-for="Token" type="hidden" />

            <input asp-for="Email" type="hidden" />

            <div class="form-group">

                <label asp-for="Password"></label>

                <input asp-for="Password" class="form-control" />

                <span asp-validation-for="Password" class="text-danger"></span>

            </div>

            <div class="form-group">

                <label asp-for="ConfirmPassword"></label>

                <input asp-for="ConfirmPassword" class="form-control" />

                <span asp-validation-for="ConfirmPassword" class="text-danger"></span>

            </div>

            <button type="submit" class="btn btn-primary">Reset</button>

        </form>
    </div>

</div>
```

Reset Password Action Methods

Include the following HTTP GET and POST RestPassword() actions in the AccountController. The code in these 2 actions is commented and self-explanatory.

```
[HttpGet]

[AllowAnonymous]

public IActionResult ResetPassword(string token, string email)

{

    // If password reset token or email is null, most likely the

    // user tried to tamper the password reset link

    if (token == null || email == null)

    {

        ModelState.AddModelError("", "Invalid password reset token");

    }

    return View();

}


[HttpPost]

[AllowAnonymous]

public async Task<IActionResult> ResetPassword(ResetPasswordViewModel model)

{

    if (ModelState.IsValid)

    {

        // Find the user by email
        var user = await userManager.FindByEmailAsync(model.Email);


        if (user != null)

        {

            // reset the user password

            var result = await userManager.ResetPasswordAsync(user, model.Token,
model.Password);

            if (result.Succeeded)

            {

                return View("ResetPasswordConfirmation");

            }

            // Display validation errors. For example, password reset token already

            // used to change the password or password complexity rules not met

            foreach (var error in result.Errors)

            {

                ModelState.AddModelError("", error.Description);

            }

            return View(model);

        }


        // To avoid account enumeration and brute force attacks, don't

        // reveal that the user does not exist

        return View("ResetPasswordConfirmation");

    }

    // Display validation errors if model state is not valid

    return View(model);

}
```

Reset Password Confirmation View

```
<h4>
   Your password is reset. Please click <a asp-action="Login">here to login</a>
</h4>
```

we will understand, **how asp.net core generates and validates tokens** i.e Password Rest Token and Email Confirmation Token for example.

We discussed generating and using

- Email Confirmation Token in Parts 113 and 114
- Password Reset Token in Parts 115 and 116

ASP.NET Core built-in UserManager service provides useful methods to generate and validate these tokens. For example,

To generate Email Confirmation Token, we use GenerateEmailConfirmationTokenAsync() method

```
var token = await userManager.GenerateEmailConfirmationTokenAsync(user);
```

To generate Password Reset Token, we use GeneratePasswordResetTokenAsync() method

```
var token = await userManager.GeneratePasswordResetTokenAsync(user);
```

Both these methods, GenerateEmailConfirmationTokenAsync() and GeneratePasswordResetTokenAsync() internally calls GenerateUserTokenAsync() method.

ASP.NET Core is open source. So, if you take a look at the UserManager class source code on their official github page, you will see both these methods call GenerateUserTokenAsync() method.
https://github.com/aspnet/AspNetCore/blob/release/2.2/src/Identity/Extensions.Core/src/UserManager.cs

Notice, both these methods call GenerateUserTokenAsync() method.

```
public virtual Task<string> GenerateEmailConfirmationTokenAsync(TUser user)
{
    ThrowIfDisposed();
    return GenerateUserTokenAsync(user,
        Options.Tokens.EmailConfirmationTokenProvider, ConfirmEmailTokenPurpose);
}

public virtual Task<string> GeneratePasswordResetTokenAsync(TUser user)
{
    ThrowIfDisposed();
    return GenerateUserTokenAsync(user,
        Options.Tokens.PasswordResetTokenProvider, ResetPasswordTokenPurpose);
}
```

GenerateUserTokenAsync() has 3 parameters.

- The user the token will be for
- The token provider which will actually generate the token
- The token purpose - For example, password reset or email confirmation. Token generated for a given purpose must be used only for that purpose. For example, an email confirmation token cannot be used to reset a password.

DataProtectorTokenProvider

Both Email Confirmation Tokens and Password Reset Tokens are generated by the built-in DataProtectorTokenProvider. The source code of this class is on the following github page https://github.com/aspnet/Identity/blob/release/2.2/src/Identity/DataProtectionTokenProvider.cs

GenerateAsync() method generates the token and ValidateAsync() method validates the token

DataProtectorTokenProvider GenerateAsync()

As you can see from the code, the generated token contains

- Token Creation Time
- User ID
- Token Purpose
- Security Stamp

All this data is then encrypted and then base 64 encoded so it can sent over the network. ASP.NET Core uses **Data Protection API** (in short **DP API**) for encryption. Later in this video series, we will discuss how to use DP API and encrypt our application data like query strings for example.

```
public virtual async Task<string> GenerateAsync(string purpose,
   UserManager<TUser> manager, TUser user)
{
   if (user == null)
   {
      throw new ArgumentNullException(nameof(user));
   }
   var ms = new MemoryStream();
   var userId = await manager.GetUserIdAsync(user);
   using (var writer = ms.CreateWriter())
   {
      writer.Write(DateTimeOffset.UtcNow);
      writer.Write(userId);
      writer.Write(purpose ?? "");
      string stamp = null;
      if (manager.SupportsUserSecurityStamp)
      {
         stamp = await manager.GetSecurityStampAsync(user);
      }
      writer.Write(stamp ?? "");
   }
   var protectedBytes = Protector.Protect(ms.ToArray());
   return Convert.ToBase64String(protectedBytes);
}
```

DataProtectorTokenProvider ValidateAsync()

As the name implies ValidateAsync() method validates the token. First it is decoded from Base 64 and then decrypted. Decryption is done by the Data Protection API (DP API).

The token creation time is read from the token. To this, the the token life span is added. If this computed DateTime is less than current UTC DateTime, the token has expired. So the method returns false. Tokens cannot be used after they have expired. The default token lifespan is one day. We can change this to meet our application requirements. We will discuss how to do this in our next video.

The User ID in the token is compared against the User ID the token is being used. If the User IDs are not equal, the method return false invalidating the token. A token generated for a given user must only be used by that user.

The purpose from the token is read to make sure it is used for the intended purpose. Token generated for a purpose must be used only for that purpose. If we try to use it for a different purpose, token validation fails. For example, if a token is generated for a given user for the purpose of resetting password, it can be used only by that specific user for resetting his password. If we try to use it for a different user or purpose, token validation fails. Simply put, Password Reset Token cannot be used to confirm an email address.

The security stamp in the token is compared with the user current security stamp in the database. If they are different, the token validation fails.

```
public virtual async Task<bool> ValidateAsync(string purpose,
                        string token, UserManager<TUser> manager, TUser user)
{
   try
   {
      var unprotectedData = Protector.Unprotect(Convert.FromBase64String(token));
      var ms = new MemoryStream(unprotectedData);
      using (var reader = ms.CreateReader())
      {
         var creationTime = reader.ReadDateTimeOffset();
         var expirationTime = creationTime + Options.TokenLifespan;
         if (expirationTime < DateTimeOffset.UtcNow)
         {
            return false;
         }

         var userId = reader.ReadString();
         var actualUserId = await manager.GetUserIdAsync(user);
```

```
            if (userId != actualUserId)
            {
                return false;
            }
            var purp = reader.ReadString();
            if (!string.Equals(purp, purpose))
            {
                return false;
            }
            var stamp = reader.ReadString();
            if (reader.PeekChar() != -1)
            {
                return false;
            }

            if (manager.SupportsUserSecurityStamp)
            {
                return stamp == await manager.GetSecurityStampAsync(user);
            }
            return stamp == "";
        }
    }
    // ReSharper disable once EmptyGeneralCatchClause
    catch
    {
        // Do not leak exception
    }

    return false;

}
```

we will discuss, **how to set password reset token lifetime** i.e specifying how long the token will be valid.

This is continuation to our previous video Part 117. Please watch Part 117 from asp.net core tutorial.

Both, password reset token and email confirmation token are generated by the built-in DataProtectorTokenProvider class. In our previous video, we discussed in detail, how this class generates and validates these tokens.

So, it is the DataProtectorTokenProvider class that generates the password reset token. The token life span is controlled by DataProtectionTokenProviderOptions class. You can see this yourself, if you look at the source code of DataProtectorTokenProvider class at the following link.
https://github.com/aspnet/Identity/blob/release/2.2/src/Identity/DataProtectionTokenProvider.cs

The default token life span is 1 day. You can see this yourself if you look at the source code of DataProtectionTokenProviderOptions class at the following link.
https://github.com/aspnet/Identity/blob/release/2.2/src/Identity/DataProtectionTokenProviderOptions.cs

From security standpoint, password reset token is a bit sensitive so it make sense to reduce the time it is valid for. The following code sets the token life span to 5 hours.

```
public void ConfigureServices(IServiceCollection services)
{
    // rest of the code

    services.AddIdentity<ApplicationUser, IdentityRole>(options =>
    {
        options.Password.RequiredLength = 10;
        options.Password.RequiredUniqueChars = 3;

        options.SignIn.RequireConfirmedEmail = true;
    })
    .AddEntityFrameworkStores<AppDbContext>()
    .AddDefaultTokenProviders();

    // Set token life span to 5 hours
    services.Configure<DataProtectionTokenProviderOptions>(o =>
        o.TokenLifespan = TimeSpan.FromHours(5));

    // rest of the code
}
```

The above code, not only sets password reset token life span to 5 hours. It also sets the life span of all the tokens generated by DataProtectorTokenProvider class to 5 hours. This may not be the behaviour you want. For example, the email confirmation token is also generated by DataProtectorTokenProvider class. So this means even the email confirmation token life span is 5 hours.

In general email confirmation tokens can live a little longer than password reset tokens. For example, let's say we want to change the life span of email confirmation token to 3 days. To achieve this we have to create a custom DataProtectorTokenProvider and DataProtectionTokenProviderOptions.

we will discuss **how to set token lifespan for a specific type of token**. This is continuation to our previous video Part 118. Please watch Part 118 from ASP.NET Core tutorial before proceeding.

The built-in DataProtectorTokenProvider can generate different types of tokens like Email Confirmation Token, Password Reset Token for example.

The default lifespan for all these token types is 1 day. One way to change the default lifespan is by using the built-in DataProtectionTokenProviderOptions. This class sets the lifespan of all the token types to the same value. In the following example to 5 hours. We discussed this in detail in our previous video.

```
services.Configure<DataProtectionTokenProviderOptions>(o =>
        o.TokenLifespan = TimeSpan.FromHours(5));
```

If you want to set the lifespan of just a specific type of token, you can do so by creating a custom token provider. For example, let's set the lifespan of email confirmation token type to 3 days.

Create custom email confirmation token provider options

By default, it is the built-in DataProtectionTokenProviderOptions class that controls the token lifespan of all token types. If you want to set a specific lifespan for just the email confirmation token type, create a CustomEmailConfirmationTokenProviderOptions class.

Make this custom class inherit from the built-in DataProtectionTokenProviderOptions class. There are 2 important reasons why we do this.

1. The TokenLifespan property is inherited from the base class and
2. It allows an instance of this class to be passed as an argument to the DataProtectorTokenProvider class.

```
public class CustomEmailConfirmationTokenProviderOptions
    : DataProtectionTokenProviderOptions
{ }
```

Create custom email confirmation token provider

It is the built-in DataProtectorTokenProvider class that generates the email confirmation token. Our custom provider class gets all the functionality required to generate tokens by inheriting from the DataProtectorTokenProvider class. We do not have to write any special logic in this custom provider class to generate tokens. It will be taken care by the base DataProtectorTokenProvider class. All we need is a constructor which in turn calls the base class constructor. Our CustomEmailConfirmationTokenProviderOptions instance is passed to the base class constructor.

```
public class CustomEmailConfirmationTokenProvider<TUser>
    : DataProtectorTokenProvider<TUser> where TUser : class
{

  public CustomEmailConfirmationTokenProvider(IDataProtectionProvider dataProtectionProvid
er,
                                    IOptions<CustomEmailConfirmationTokenProviderOptions> options)
        :base(dataProtectionProvider, options)
    { }
}
```

Register custom token provider

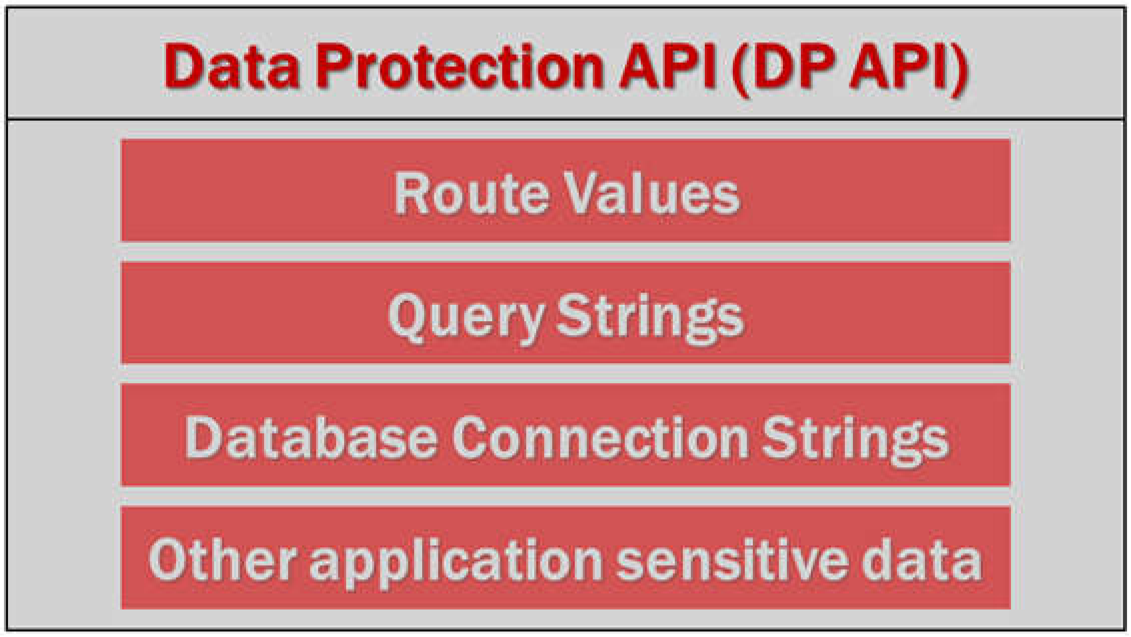
```
public void ConfigureServices(IServiceCollection services)
{
    // Rest of the code

    services.AddIdentity<ApplicationUser, IdentityRole>(options =>
    {
        options.Tokens.EmailConfirmationTokenProvider = "CustomEmailConfirmation";
    })
    .AddEntityFrameworkStores<AppDbContext>()
    .AddDefaultTokenProviders()
    .AddTokenProvider<CustomEmailConfirmationTokenProvider
            <ApplicationUser>>("CustomEmailConfirmation");

    // Rest of the code
}
```

Change email confirmation token lifespan

```
public void ConfigureServices(IServiceCollection services)
{
    // Changes token lifespan of all token types
    services.Configure<DataProtectionTokenProviderOptions>(o =>
            o.TokenLifespan = TimeSpan.FromHours(5));

    // Changes token lifespan of just the Email Confirmation Token type
    services.Configure<CustomEmailConfirmationTokenProviderOptions>(o =>
            o.TokenLifespan = TimeSpan.FromDays(3));
}
```

we will discuss **encryption and decryption with an example in asp.net core**.

We will discuss how to encrypt and decrypt route values. The same techniques can be used to encrypt query strings, database connection strings and other application sensitive data. It is the Data Protection API (DP API in short), that we will be using for our encryption needs.


Data Protection API (DP API)
Route Values
Query Strings
Database Connection Strings
Other application sensitive data

Let's understand encrypting route values with an example. To view a specific employee details the following is the URL we use. The integer value (5) in the URL at the end is the ID of the employee.

```
https://localhost:1111/home/details/5
```

We want to encrypt, so it's not readable.

```
https://localhost:44376/home/details/CfDJ8J-n2P...
```

Take a look at the asp.net core DataProtectorTokenProvider class source code https://github.com/aspnet/Identity/blob/release/2.2/src/Identity/DataProtectionTokenProvider.cs

It is this class that generates email confirmation token, password reset token etc. It is also responsible for encrypting and decrypting these tokens. We discussed this in detail in Part 117 of asp.net core tutorial.

We use Protect() and Unprotect() methods of IDataProtector interface to encrypt and decrypt respectively.

The following are the steps to encrypt and decrypt route values

Create purpose string

This class holds the purpose strings required for encryption and decryption. At the moment we have, only one, purpose string. We will discuss the use of purpose string in just a bit.

```
public class DataProtectionPurposeStrings
{
    public readonly string EmployeeIdRouteValue = "EmployeeIdRouteValue";
}
```

Register purpose string class with DI container

Register the class that contains purpose strings with the asp.net core dependency injection container. This allows us to inject an instance of this class into any controller throughout our application. ConfigureServices method is in the Startup class.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<DataProtectionPurposeStrings>();
}
```

Model property to hold encrypted ID

As the name implies, EncryptedId property holds the encrypted employee id. NotMapped attribute specifies that this property must be excluded from mapping it to a database table column. NotMapped attribute is in System.ComponentModel.DataAnnotations.Schema namespace.

```
public class Employee
{
    public int Id { get; set; }

    [NotMapped]
    public string EncryptedId { get; set; }

    // rest of the properties
}
```

IDataProtector Protect and Unprotect methods

IDataProtector is required in the HomeController. In this Index() action we encrypt the employee id values and in the Details() they are decrypted.

```
using EmployeeManagement.Models;

using EmployeeManagement.Security;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.DataProtection;

using Microsoft.AspNetCore.Mvc;

using System;

using System.Linq;


namespace EmployeeManagement.Controllers

{

    [Authorize]

    public class HomeController : Controller

    {

        private readonly IEmployeeRepository _employeeRepository;

        // It is through IDataProtector interface Protect and Unprotect methods,

        // we encrypt and decrypt respectively

        private readonly IDataProtector protector;


        // It is the CreateProtector() method of IDataProtectionProvider interface

        // that creates an instance of IDataProtector. CreateProtector() requires

        // a purpose string. So both IDataProtectionProvider and the class that

        // contains our purpose strings are injected using the contructor
```

```
public HomeController(IEmployeeRepository employeeRepository,

                      IDataProtectionProvider dataProtectionProvider,

                      DataProtectionPurposeStrings dataProtectionPurposeStrings)

{

    _employeeRepository = employeeRepository;

    // Pass the purpose string as a parameter

    this.protector = dataProtectionProvider.CreateProtector(

        dataProtectionPurposeStrings.EmployeeIdRouteValue);

}


[AllowAnonymous]

public ViewResult Index()

{

    var model = _employeeRepository.GetAllEmployee()

                    .Select(e =>

                    {

                        // Encrypt the ID value and store in EncryptedId property

                        e.EncryptedId = protector.Protect(e.Id.ToString());

                        return e;

                    });

    return View(model);

}


// Details view receives the encrypted employee ID

[AllowAnonymous]

public ViewResult Details(string id)
```

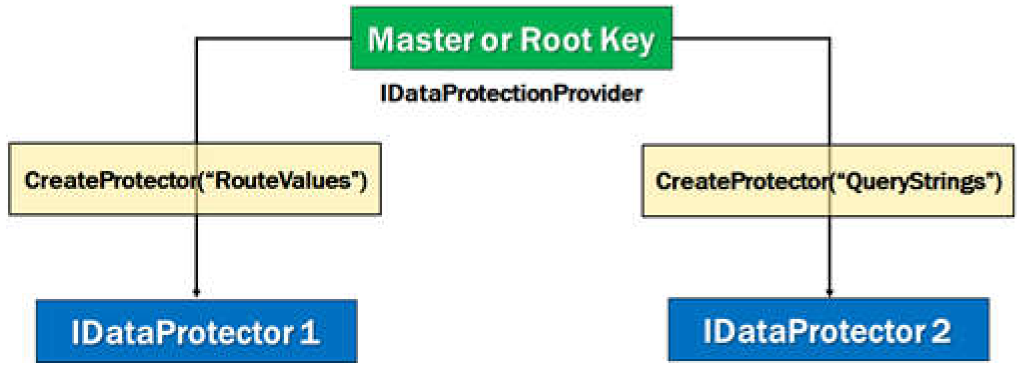
```
        {
            // Decrypt the employee id using Unprotect method
            string decryptedId = protector.Unprotect(id);
            int decryptedIntId = Convert.ToInt32(decryptedId);

            Employee employee = _employeeRepository.GetEmployee(decryptedIntId);

            return View(employee);
        }

        // rest of the code
    }
}
```

Encrypted ID in View

In the view, bind the EncryptedId to the View action link.

```
@model IEnumerable<Employee>

@{
    ViewBag.Title = "Employee List";
}

<div class="card-deck">
    @foreach (var employee in Model)
    {
        <div class="card-footer text-center">
        <a asp-controller="home" asp-action="details"
          asp-route-id="@employee.EncryptedId"
          class="btn btn-primary m-1">View</a>
      </div>
  }
</div>
```

Purpose string in ASP.NET Core



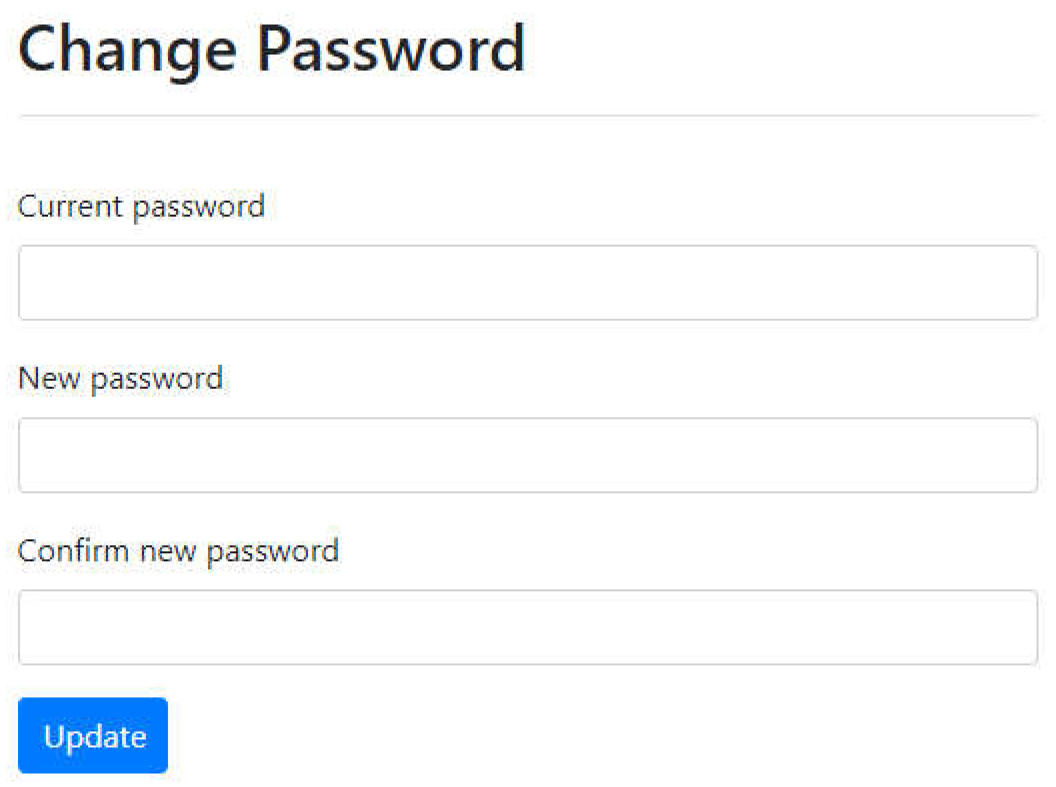
You can think of purpose string as an encryption key. This key is then combined with the master or root key to generate a unique key. The data that is encrypted by a given combination of purpose string and root key can only be decrypted by that same combination of keys.

The purpose string is inherent to the security of the data protection system, as it provides isolation between cryptographic consumers, even if the root keys are the same.

Change Password
we will discuss how to implement change password view in asp.net core MVC.

Change password view example

A logged in user can change his password using the following change password view.


Change Password

Current password

New password

Confirm new password

Why current password is required

This is to make sure it is the actual account owner who is changing the password. Though you are already logged-in, asking for the current password prevents a malicious from changing another user password if that user has briefly walked away from the computer or forgot to logout from a public computer.

UserManager ChangePasswordAsync Method

```
var result = await userManager.ChangePasswordAsync(user,
   model.CurrentPassword, model.NewPassword);
```

It is the ChangePasswordAsync() method of the UserManager service that we use to change a logged-in user password. This method takes 3 parameters.

1. The logged-in user whose password is being changed
2. Current password
3. New password

SignInManager RefreshSignInAsync() Method

```
await signInManager.RefreshSignInAsync(user);
```

Call SignInManager service RefreshSignInAsync() method, after the password is successfully changed. As the name implies, this method refreshes the logged-in user sign-in cookie.

Change Password ViewModel

```
public class ChangePasswordViewModel
{
    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "Current password")]
    public string CurrentPassword { get; set; }

    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "New password")]
    public string NewPassword { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm new password")]
    [Compare("NewPassword", ErrorMessage =
        "The new password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}
```

Change Password View

```
@model ChangePasswordViewModel

<h2>Change Password</h2>
<hr />
<div class="row">
    <div class="col-md-12">
        <form method="post">
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="CurrentPassword"></label>
                <input asp-for="CurrentPassword" class="form-control" />
                <span asp-validation-for="CurrentPassword" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="NewPassword"></label>
                <input asp-for="NewPassword" class="form-control" />
                <span asp-validation-for="NewPassword" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ConfirmPassword"></label>
                <input asp-for="ConfirmPassword" class="form-control" />
                <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
            </div>
            <button type="submit" class="btn btn-primary">Update</button>
        </form>
    </div>
</div>
```

Change Password Confirmation View

```
<h4>
    Your password is successfully changed.
</h4>
```

HttpGet ChangePassword Action

```
[HttpGet]
public IActionResult ChangePassword()
{
    return View();
}
```

HttpPost ChangePassword Action

```
[HttpPost]
public async Task<IActionResult> ChangePassword(ChangePasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = await userManager.GetUserAsync(User);
        if (user == null)
        {
            return RedirectToAction("Login");
        }

        // ChangePasswordAsync changes the user password
        var result = await userManager.ChangePasswordAsync(user,
            model.CurrentPassword, model.NewPassword);

        // The new password did not meet the complexity rules or
        // the current password is incorrect. Add these errors to
        // the ModelState and rerender ChangePassword view
        if (!result.Succeeded)
        {
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError(string.Empty, error.Description);
            }
            return View();
        }
        // Upon successfully changing the password refresh sign-in cookie
        await signInManager.RefreshSignInAsync(user);
        return View("ChangePasswordConfirmation");
    }

    return View(model);
}
```

Change password menu item in Layout view

```
<li class="nav-item dropdown">
    <a class="nav-link dropdown-toggle" href="#" id="navbarDropdownMenuLink"
        data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
        Manage
    </a>
    <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
        <a class="dropdown-item" asp-controller="Administration" asp-action="ListUsers">
            Users
        </a>
        <a class="dropdown-item" asp-controller="Administration" asp-action="ListRoles">
            Roles
        </a>
        <a class="dropdown-item" asp-controller="Account"
            asp-action="ChangePassword">
            Password
        </a>
    </div>
</li>
```

In this video we will discuss how to set password on a local user account that is linked to an external login like Google, Facebook etc. This allows the user to login using either the local user account or an external login.

Let us understand this with an example. If an external login provider like Google or Facebook is used to login, the Password column will be null for the corresponding local user account in AspNetUsers table. A local user account is usually linked to an external login using the email address.

It is possible to add a password for the local account that is linked to an external account. The obvious benefit of this is that, the user can use either the external login or the local account to login.

Add password to a local account

To add a password to a local user account that is linked to an external login, use AddPasswordAsync() method of the UserManager service.

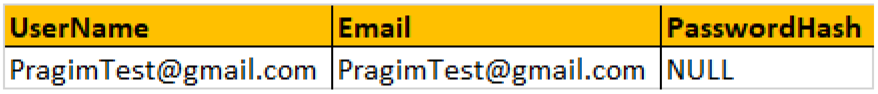
```
userManager.AddPasswordAsync(user, model.NewPassword);
```

We pass 2 parameters to this method

- The user object for whom we want to add the password
- The new password

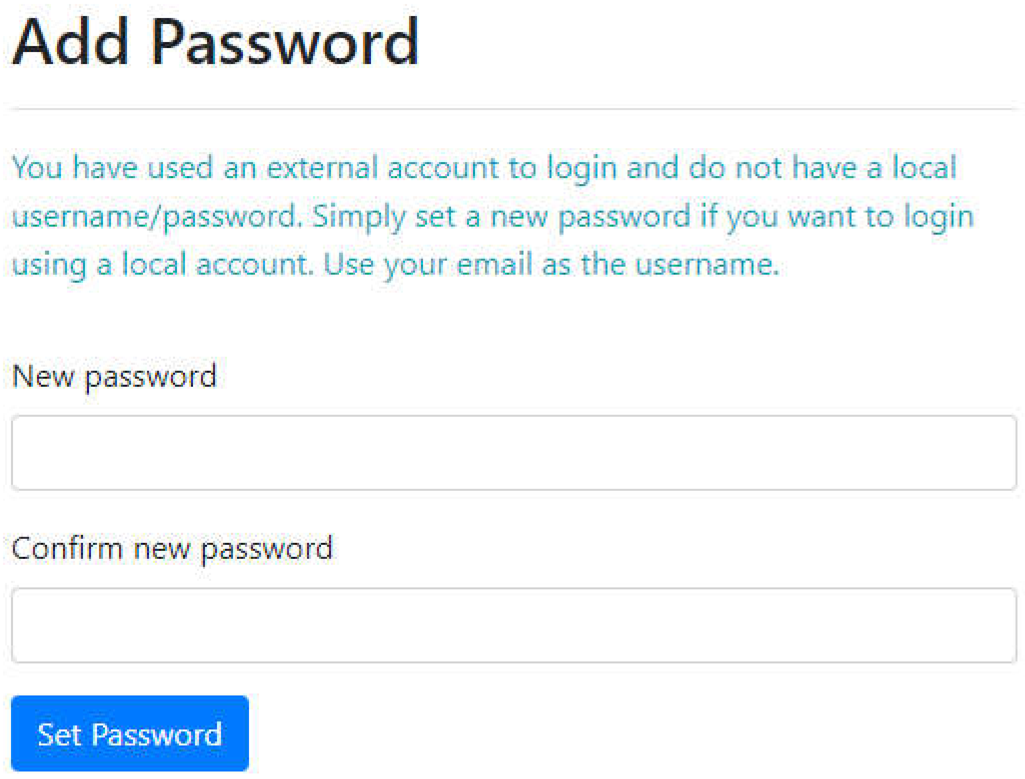
UserManager AddPasswordAsync example

Consider the following local user account in AspNetUsers table



UserName	Email	PasswordHash
PragimTest@gmail.com	PragimTest@gmail.com	NULL

This local account is linked to my external Google login. So the PasswordHash column is NULL. We want to set password on the local user account, so we can either use local username and password or the google account to login. The following is the view that can be used to set the password.



Add Password

You have used an external account to login and do not have a local username/password. Simply set a new password if you want to login using a local account. Use your email as the username.

Add Password View Model

```
public class AddPasswordViewModel
{
    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "New password")]
    public string NewPassword { get; set; }

    [DataType(DataType.Password)]
    [Display(Name = "Confirm new password")]
    [Compare("NewPassword", ErrorMessage =
       "The new password and confirmation password do not match.")]
    public string ConfirmPassword { get; set; }
}
```

Add Password View

```
@model AddPasswordViewModel

<h2>Add Password</h2>
<hr />
<p class="text-info">
    You have used an external account to login and do not have a local username and
    password. Simply set a new password if you want to login using a local account.
    Use your email as the username.
</p>
<div class="row">
    <div class="col-md-12">
        <form method="post">
            <div asp-validation-summary="All" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="NewPassword"></label>
                <input asp-for="NewPassword" class="form-control" />
                <span asp-validation-for="NewPassword" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="ConfirmPassword"></label>
                <input asp-for="ConfirmPassword" class="form-control" />
                <span asp-validation-for="ConfirmPassword" class="text-danger">
                </span>
            </div>
            <button type="submit" class="btn btn-primary"
                    style="width:auto">
                Set Password
            </button>
        </form>
    </div>

</div>
```

Add Password Confirmation View

```
<h3>You have successfully set a local password. You can now use either
   your local user account or an external account to login</h3>
```

AddPassword Actions

```
[HttpGet]
public async Task<IActionResult> AddPassword()
{
    var user = await userManager.GetUserAsync(User);

    var userHasPassword = await userManager.HasPasswordAsync(user);

    if (userHasPassword)
    {
        return RedirectToAction("ChangePassword");
    }

    return View();
}

[HttpPost]
public async Task<IActionResult> AddPassword(AddPasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = await userManager.GetUserAsync(User);

        var result = await userManager.AddPasswordAsync(user, model.NewPassword);

        if (!result.Succeeded)
        {
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError(string.Empty, error.Description);
            }
            return View();
        }

        await signInManager.RefreshSignInAsync(user);

        return View("AddPasswordConfirmation");
    }

    return View(model);
}
```

Modify the code in HttpGet ChangePassword() action to redirect the user to AddPassword() action if the user has singed in using an external login account and tries to change password.

```
[HttpGet]
public async Task<IActionResult> ChangePassword()
{
    var user = await userManager.GetUserAsync(User);

    var userHasPassword = await userManager.HasPasswordAsync(user);

    if (!userHasPassword)
    {
        return RedirectToAction("AddPassword");
    }

    return View();
}
```

What is account lockout

Account lockout is locking (i.e disabling) the account after too many failed logon attempts. Most banks lock the account after 5 failed attempts. After how many failed attempts, should the account be locked out, depends on the lockout policy of the company. The number of failed attempts after which an account should be locked is configurable in asp.net core.

Why should we lock accounts

Account lockout is to prevent attackers from brute-force attempts to guess a user's password. After certain number of failed logon attempts the account will be temporarily locked for a specified amount of time. How long the account should be locked is again configurable and we will see this in action in just a bit.

Let's say we lock the account for 15 minutes after 5 failed logon attempts. After 15 minutes the user will get another 5 attempts to logon. After 5 failed attempts the account will be locked for another 15 minutes. So this means, it will take many years for an attacker to successfully crack the password.

An organisation may also have password change policy, meaning the password must be changed every 1 or 2 months. So account lockout policy combined with password change policy makes it extremely difficult for an attacker to brute-force (i.e guess) password and gain access.

Configure account lockout options in asp.net core

Account lockout options are configured in ConfigureServices() method of the Startup class.

MaxFailedAccessAttempts - Specifies the number of failed logon attempts allowed before the account is locked out. The default is 5.

DefaultLockoutTimeSpan - Specifies the amount of the time the account should be locked. The default it 5 minutes.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddIdentity<ApplicationUser, IdentityRole>(options =>
    {
        options.Lockout.MaxFailedAccessAttempts = 5;
        options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(15);
    });

    // Rest of the code
}
```

Enable account lockout

Modify the code in Login() action in AccountController to enable account lockout. The code is commented where required.

```
[HttpPost]
[AllowAnonymous]
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl)
{
    model.ExternalLogins =
        (await signInManager.GetExternalAuthenticationSchemesAsync()).ToList();

    if (ModelState.IsValid)
    {
        var user = await userManager.FindByEmailAsync(model.Email);

        if (user != null && !user.EmailConfirmed &&
            (await userManager.CheckPasswordAsync(user, model.Password)))
        {
            ModelState.AddModelError(string.Empty, "Email not confirmed yet");
            return View(model);
        }

        // The last boolean parameter lockoutOnFailure indicates if the account
        // should be locked on failed logon attempt. On every failed logon
        // attempt AccessFailedCount column value in AspNetUsers table is
        // incremented by 1. When the AccessFailedCount reaches the configured
        // MaxFailedAccessAttempts which in our case is 5, the account will be
        // locked and LockoutEnd column is populated. After the account is
        // lockedout, even if we provide the correct username and password,
        // PasswordSignInAsync() method returns Lockedout result and the login
        // will not be allowed for the duration the account is locked.
        var result = await signInManager.PasswordSignInAsync(model.Email,
                                model.Password, model.RememberMe, true);

        if (result.Succeeded)
        {
            if (!string.IsNullOrEmpty(returnUrl) && Url.IsLocalUrl(returnUrl))
            {
                return Redirect(returnUrl);
            }
            else
            {
                return RedirectToAction("index", "home");
            }
        }

        // If account is lockedout send the use to AccountLocked view
        if (result.IsLockedOut)
        {
            return View("AccountLocked");
        }

        ModelState.AddModelError(string.Empty, "Invalid Login Attempt");
    }

    return View(model);
}
```

Account Locked View

After the account is locked, there are 2 options. Wait for the account lockout time to expire which in our case is 15 minutes and then try again or request password reset if you have forgotten the password.

```
<h3 class="text-danger">
    Your account is locked, please try again after sometime or you may
    <a asp-action="ForgotPassword" asp-controller="Account">
        reset your password by clicking here
    </a>
</h3>
```

Set lockout end date on successful password reset

If the user is lockedout, password reset can be requested. Upon successful password reset, set the account lockout end date to current UTC date time, so the user can login with the new password. Use SetLockoutEndDateAsync() method of the UserManager service for this.

```
[HttpPost]
[AllowAnonymous]
public async Task<IActionResult> ResetPassword(ResetPasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = await userManager.FindByEmailAsync(model.Email);
        if (user != null)
        {
            var result =
                await userManager.ResetPasswordAsync(user, model.Token, model.Password);
            if (result.Succeeded)
            {
                // Upon successful password reset and if the account is lockedout, set
                // the account lockout end date to current UTC date time, so the user
                // can login with the new password
                if (await userManager.IsLockedOutAsync(user))
                {
                    await userManager.SetLockoutEndDateAsync(user, DateTimeOffset.UtcNow);
                }
                return View("ResetPasswordConfirmation");
            }

            foreach (var error in result.Errors)
            {
                ModelState.AddModelError("", error.Description);
            }
            return View(model);
        }

        return View("ResetPasswordConfirmation");
    }
    return View(model);
}
```