Section 25: Identity, Authorization, Security [MVC and Web API]

06 May 2025 11:45

🧠 **What is ASP.NET Core Identity?**
It's a **full-blown user account management system** built into ASP.NET Core.
Basically, it's like your app's personal **user bouncer + profile manager + gatekeeper** 💂

🏗️ **Key Features:**

- ✅ **User Registration** with CreateAsync
- 🔐 **Login / Authentication**
- 🔄 **External Logins** (Google, Facebook, GitHub, etc.)
- 👤💼 **Role-based Access Control** (RBAC)
- 📧 **Email Confirmation**
- 🔁 **Password Reset / Update**
- 🧲 **Claims-based Auth (if needed)**

⚙️ **The Identity Architecture (Real Talk Breakdown):**

Layer	Description
DB Context	IdentityDbContext (inherits from EF DbContext) – holds tables like AspNetUsers, AspNetRoles
Stores	UserStore, RoleStore – acts like **repositories** on top of the DB context
Managers	UserManager, SignInManager, RoleManager – holds **business logic** (create user, verify password, etc.)
Controllers / Services	These sit at the top and use the managers to handle actual HTTP requests

📦 **So flow looks like:**
Controller → UserManager or SignInManager → UserStore → IdentityDbContext → Database

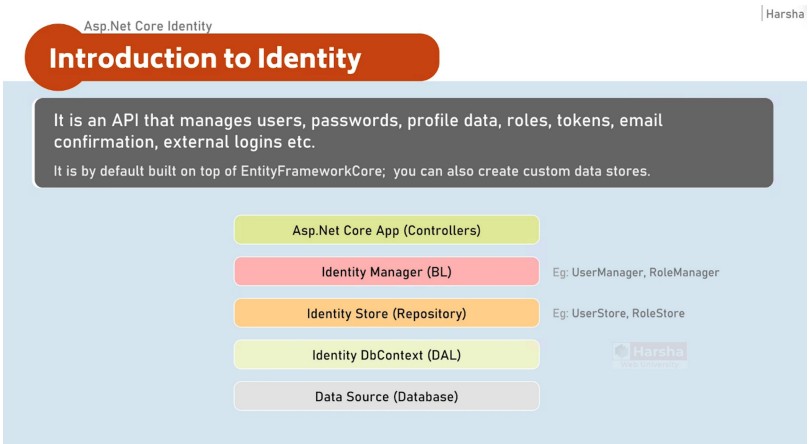
🛠️ **Wanna switch DB from SQL Server to PostgreSQL, MySQL, or even Azure Table Storage?**
✅ Totally possible, you just need to **implement a custom store** or use a community package.

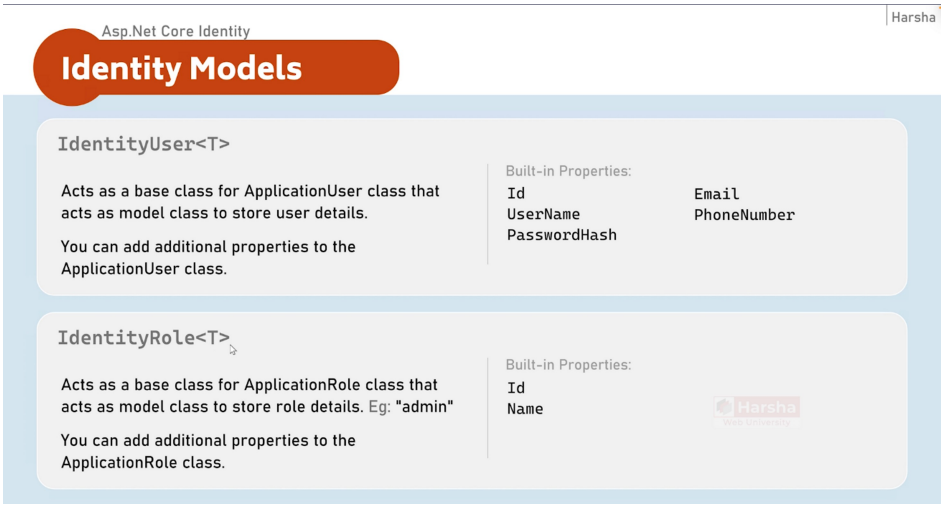
🤔 **Difference between DbContext and IdentityDbContext?**

- DbContext = your basic EF Core base class.
- IdentityDbContext = pre-built by Microsoft, includes identity-specific tables like Users, Roles, UserRoles, UserClaims, etc.
- You can **extend it** with your own custom properties (like adding ProfilePictureUrl or IsPremiumUser to the ApplicationUser class).

🧪 **And yes, it's testable too!**
You can mock UserManager in your unit tests or use in-memory DBs to simulate real scenarios 🚀

🧠 **Pro Tip for Interviews:**
Clean Architecture + ASP.NET Core Identity = ultimate modular and testable user management. Managers encapsulate logic, stores abstract data access, and IdentityDbContext gives you the schema out of the box — swappable, extensible, and fully secured.





🤗 **Working with ASP.NET Core Identity**

While working with ASP.NET Core Identity,
you have to commonly use **two entity classes**:
IdentityUser and **IdentityRole**.
These are much like the entity classes
that we have used earlier in the project,
such as Country and Person.
In the application DB context,
we were using that entity classes such as Person class.
We were creating DB context of the Person type, right?
Exactly similar to that,
in case of Identity, the **DbSets are created** based on these two entity classes:

- IdentityUser
- IdentityRole

🥷 **IdentityUser: User Details Entity**
The IdentityUser entity class
represents the **user details** — starting from:

- username
- password
- email
- phone number
- etc.

And also, you can add additional properties that are not exist in that.
It mainly consists of **five important properties**:

- ID (I mean the automatically generated GUID)
- username
- password
- email
- phone number

Of course, the password will not be stored as a simple string.
Because if it is stored as a simple string,
anyone who can access the database can see the user's passwords, right?
So it is not at all safe —

that is why almost all the websites store the passwords as a **hash**.
You should **never** store the passwords as a plain string.
So these are the five built-in and predefined properties in the IdentityUser entity class.
But you can create a **child class** based on this IdentityUser,
and there you can **add additional properties** that you want.
For example, you would like to store:

- the address of the user
- location of the user
- or anything like that

You can add those properties in the child class of the IdentityUser.

🚀 **Generic IdentityUser<T>**
And it is a **generic class**.
So the generic parameter T represents the **data type of the ID**.
By default, it is a **string**.
But you can store the same as **int**, or **GUID**, or any other data type that you want.
Generally, **GUID** is recommended.
Because GUID is **unlimited**.
If we are using int as a data type —
since the int values are limited,
you may not be able to store a large number of users.
But the GUID offers **unlimited number of users**,
because there is no limit of number of GUIDs that can be generated.
And it is completely **universally unique**.

🛡️ **IdentityRole: Role Details Entity**
Exactly just like we have IdentityUser to represent the user details,
we have IdentityRole to represent the **role of the user**.
Here **role** means the **type of user**,
for example:

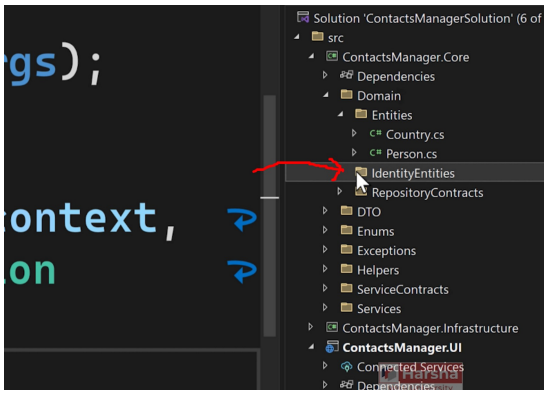
- customer user
- administrator user
- manager user
- etc.

So how many types of users your application supports or has,
that many number of IdentityRoles can be created.
And exactly much like you will create a child class that inherits from the IdentityUser,
you will also create a **child class** from this IdentityRole.
And here T represents the **data type of the ID**.
By default, it has only **two properties**:

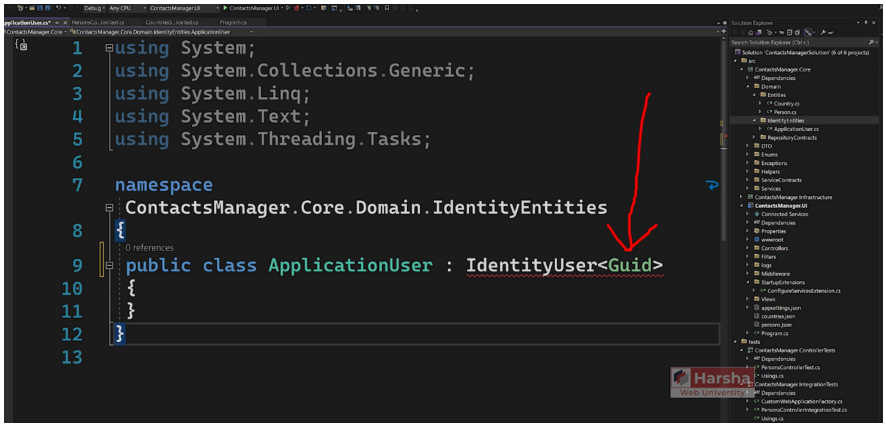
- ID (the GUID, automatically generated)
- Name (the name of the role, for example Administrator, Manager, etc.)

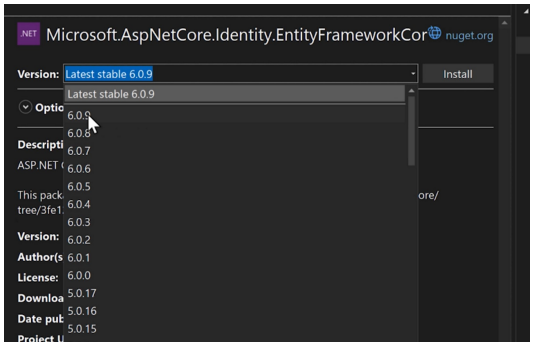
But apart from that,
you can **optionally add additional properties** that you want
in the child class of IdentityRole.

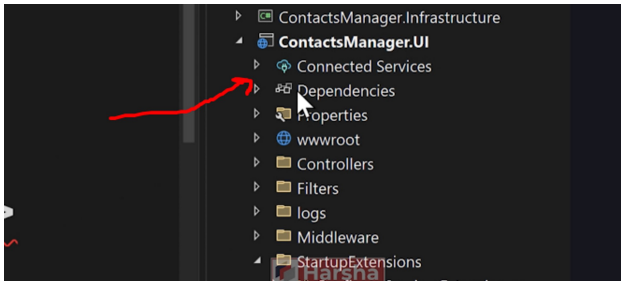
Let me show the same practically. 🚀

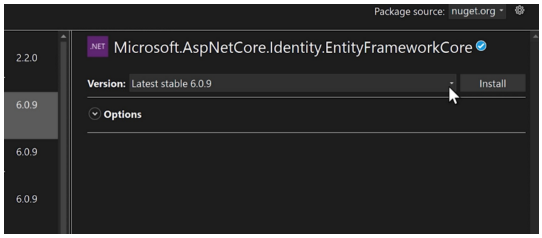


By default this class is not recognized, we need to install some nuget packages.

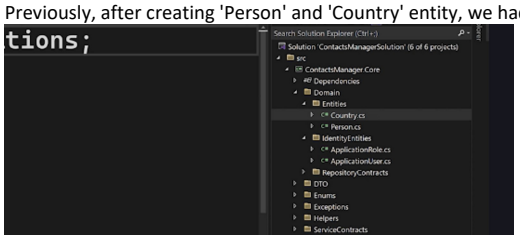




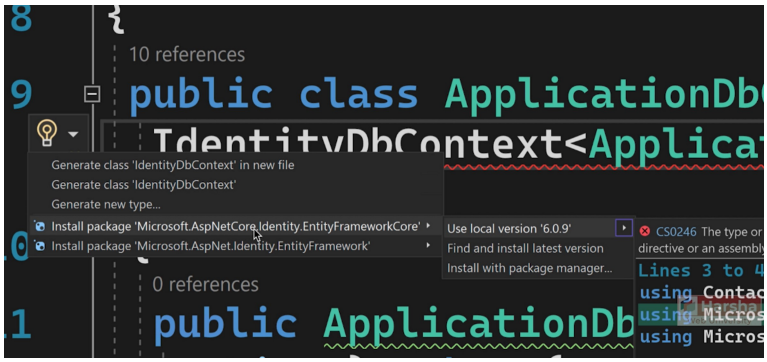




Previously, after creating 'Person' and 'Country' entity, we had to create DbContext class



```
using Microsoft.EntityFrameworkCore;

namespace Entities
{
    10 references
    public class ApplicationDbContext :
        IdentityDbContext<ApplicationUser, ApplicationRole,
        Guid>
    {
        0 references
        public ApplicationDbContext(DbContextOptions
            options) : base(options)
        {
        }
```



🔄 Regular DbContext vs IdentityDbContext

The **regular DbContext** that we were using earlier
doesn't offer any predefined DbSets.
Of course, you can create your own DbSets —
for example, Countries and Persons DbSets.
But **alternatively**...
The IdentityDbContext offers the **predefined DbSets**
that are needed to store:

- users
- roles
- and the mapping between users and roles

📦 Behind the Scenes of Identity

To store the user details along with their roles,
you need some **predefined tables** —
around **four to five tables**.
That means those four or five tables
should be reflected as **DbSets**, right?
✅ And those DbSets are **already created** in the IdentityDbContext.

🤔 How Does It Know Which Types to Use?

IdentityDbContext doesn't know the actual data types
used for User and Role, right?
That is why **we are supplying the corresponding data types.**
We are saying:

> "Hey IdentityDbContext,
> create a DbSet for ApplicationUser,
> and create another DbSet for ApplicationRole."

🧩 What We've Done So Far

So far in this lecture:
✅ We have created the entity classes —
ApplicationUser and ApplicationRole.
✅ We have inherited the IdentityDbContext
into our **ApplicationDbContext**.
✅ In the **Core project**, we installed two packages:

- Microsoft.AspNetCore.Identity
- Microsoft.AspNetCore.Identity.EntityFrameworkCore

✅ In the **Infrastructure project**,
we installed **only one package**:

- Microsoft.AspNetCore.Identity.EntityFrameworkCore

🔧 What's Next?

We have created the essential entity classes and DbContext setup.
Now in the further steps, we have to:

Add Identity as a service into the IoC container
Work on creating:

- **UserStore**
- **UserManager** classes

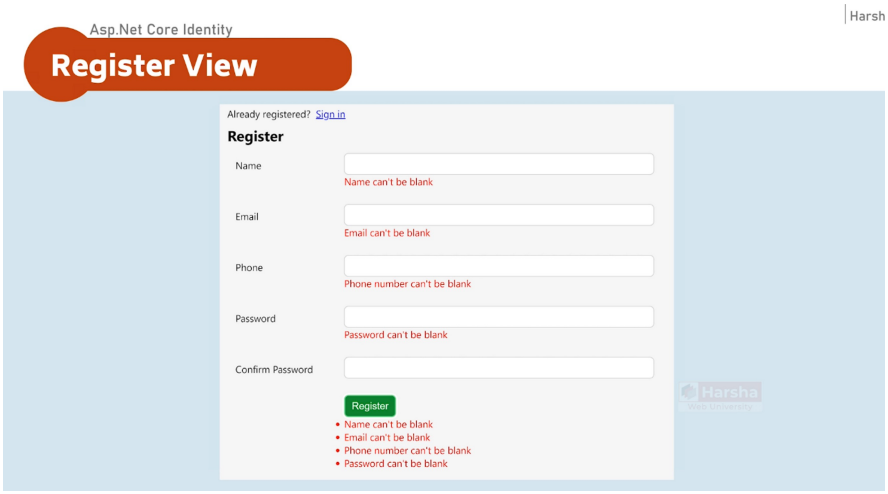
The UserManager class will be used in the **controller**
to create and manage user accounts.
Then you'll be able to build:

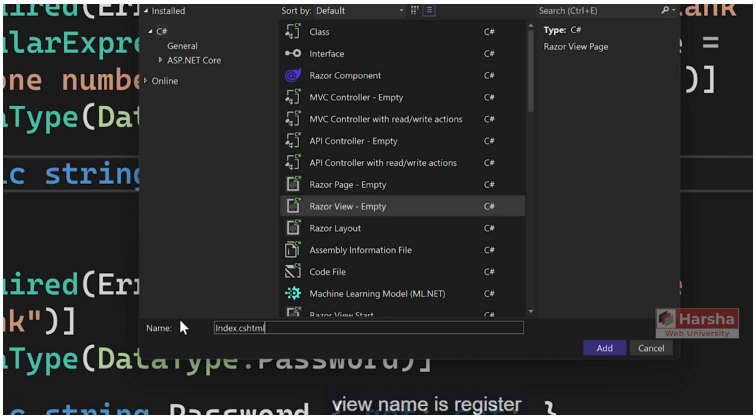
- Registration form
- Login form
- And other user-related UI for end users

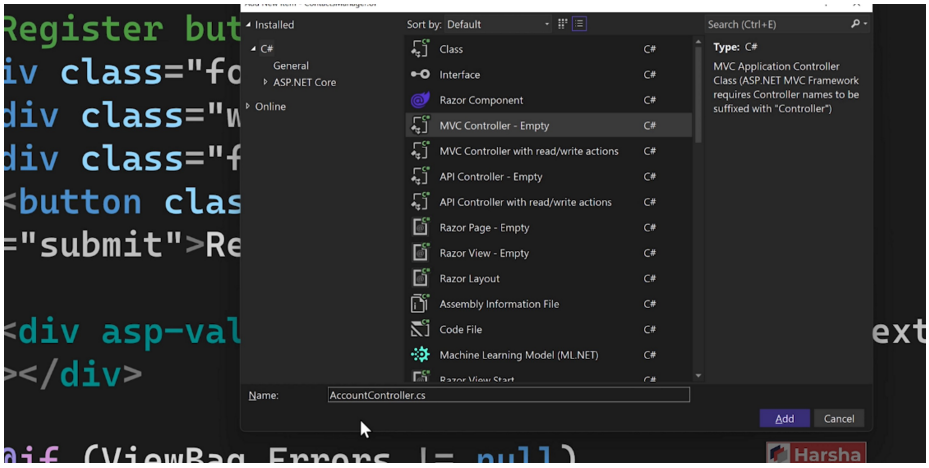
Asp.Net Core Identity

Register View

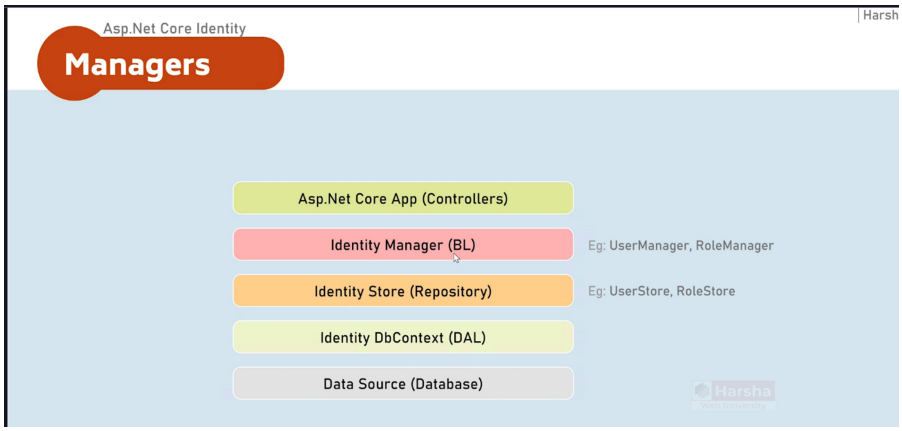
"Now let us try to create the **Register View** with these fields.
And when the user clicks on the **Register** button,
we have to store all these user details as a **user account**
into the **Identity database**.
Okay, let us try!"





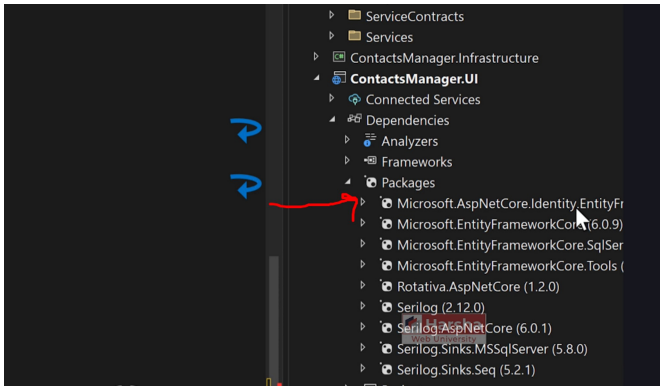


Before manipulating user and roles information, we have to add 'identity' as a service into the IOC container.



```
PersonsUpdaterService>();
services.AddScoped<IPersonsSorterService,
 PersonsSorterService>();

services.AddDbContext<ApplicationDbContext>
 (options =>
{
 options.UseSqlServer
  (configuration.GetConnectionString
  ("DefaultConnection"));
});

services.AddTransient<PersonsListActionFilter>();

services.AddIdentity();

services.AddHttpLogging(options =>
```

But make sure we have this following package installed.



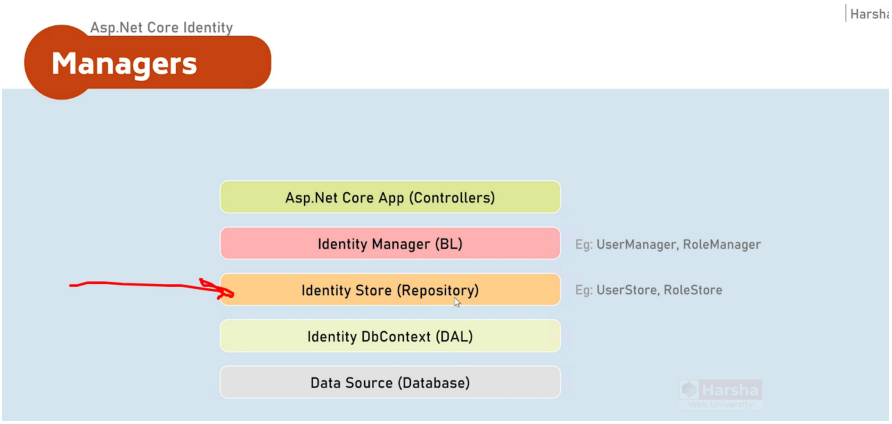
Now it understands we have to create users table with these properties
And roles table with these properties,
And try to store data using this particular DB context.

```
//Enable Identity in this project
services.AddIdentity<ApplicationUser,
 ApplicationRole>()
 .AddEntityFrameworkStores<ApplicationDbContext>
  ();
```

```
  (configuration.GetConnectionString
  ("DefaultConnection"));
});

services.AddTransient<PersonsListActionFilter>();

//Enable Identity in this project
services.AddIdentity<ApplicationUser,
 ApplicationRole>()
 .AddEntityFrameworkStores<ApplicationDbContext>()

 .AddUserStore<UserStore<ApplicationUser,
  ApplicationRole, ApplicationDbContext, Guid>>()

 .AddRoleStore<RoleStore<ApplicationRole,
  ApplicationDbContext, Guid>>();


services.AddHttpLogging(options =>
```

```
using Microsoft.AspNetCore.Identity;
using System;

namespace
 ContactsManager.Core.Domain.IdentityEntities
{
 3 references
 public class ApplicationUser : IdentityUser<Guid>
 {
  0 references
  public string? PersonName { get; set; }
 }
}
```

You can assume that Repository will be generated automatically

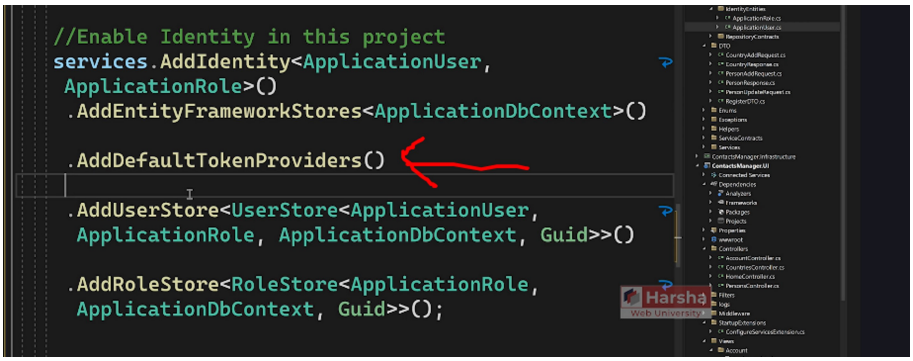


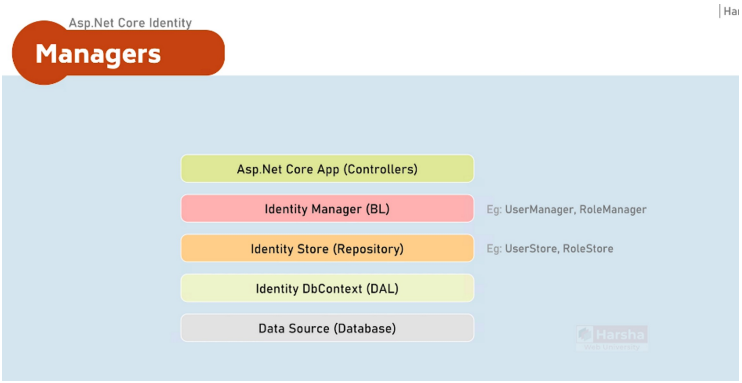
To ensure smooth operation for critical user actions like login (email and phone verification/confirmation), password recovery (forgot and reset password), we must enable the default token providers in ASP.NET Core.
In these scenarios, random tokens need to be generated at runtime. For instance, during a password reset, an OTP (One-Time Password) might be automatically generated and sent to the user's email. The user then enters this OTP in the application to confirm their account.

ASP.NET Core provides predefined token providers to facilitate this token and OTP generation. These include:

- Email Token Provider
- Phone Number Token Provider
- Security Stamp Based Token Provider
- Data Protector Token Provider
- Etc.

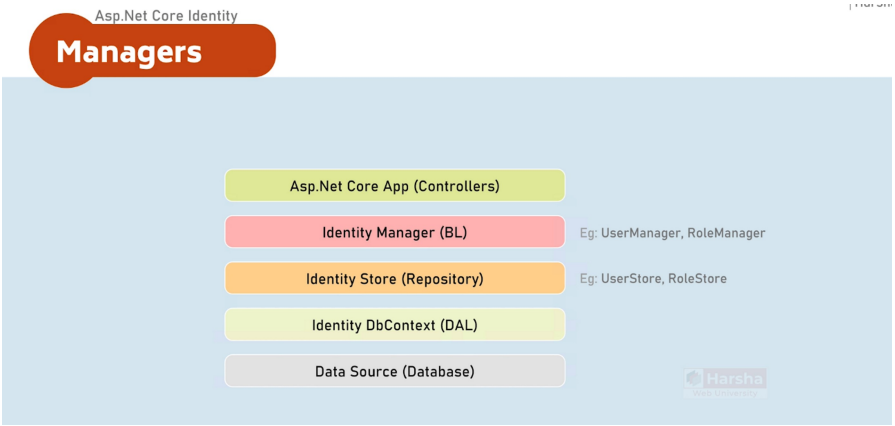
Therefore, to register these default token services, we need to include the AddDefaultTokenProviders() statement in our application's service configuration. While we may not utilize all of these providers, adding this service is essential for the overall token generation and validation processes to function correctly.





NET Core Identity:

- **Managers** (like UserManager) hold the business logic for user and role operations.
- Specifically, UserManager provides methods for creating, deleting, updating, and retrieving users.
- UserManager relies on the **User Store**, which is the repository layer.
- The **User Store** is built on top of the **IdentityDbContext**, which acts as the data access layer.
- **IdentityDbContext** uses Entity Framework Core, meaning your LINQ queries and SaveChanges() calls are translated into SQL executed against the database.
- As a developer, you interact with the **Manager classes** (like UserManager) in your controllers, similar to how you use your own service classes for business logic. You don't directly work with the DbContext or the Store.
- There's also a SignInManager for authentication-related logic, which you'll cover later. For now, the focus is on UserManager.



Asp.Net Core Identity

Managers

UserManager

Provides business logic methods for managing users.
It provides methods for creating, searching, updating and deleting users.

Methods:

- CreateAsync()
- DeleteAsync()
- UpdateAsync()
- IsInRoleAsync()
- FindByEmailAsync()
- FindByIdAsync()
- FindByNameAsync()

SignInManager

Provides business logic methods for sign-in and sign-in functionality of the users.
It provides methods for creating, searching, updating and deleting users.

Methods:

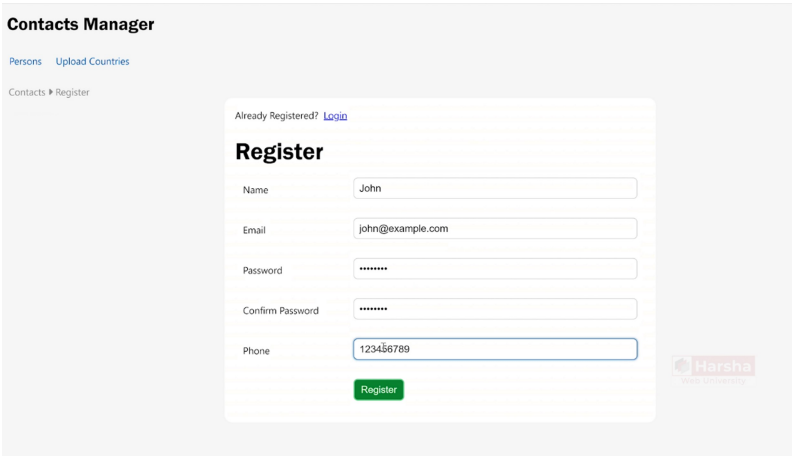
- SignInAsync()
- PasswordSignInAsync()
- SignOutAsync()
- IsSignedIn()

```
using Microsoft.AspNetCore.Mvc;

namespace ContactsManager.UI.Controllers
{
 [Route("[controller]/[action]")]
 1 reference
 public class AccountController : Controller
 {
  private readonly UserManager<ApplicationUser>
   _userManager;

  0 references
  public AccountController
   (UserManager<ApplicationUser> userManager)
  {
   _userManager = userManager;
  }
```

```
[HttpPost]
0 references
public IActionResult Register(RegisterDTO
 registerDTO)
{
 //Check for validation errors
 if (ModelState.IsValid == false)
 {
  ViewBag.Errors = ModelState.Values.SelectMany
   (temp => temp.Errors).Select(temp =>
   temp.ErrorMessage);
  return View(registerDTO);
 }

 ApplicationUser user = new ApplicationUser() { }
 //TO DO: Store user registration details into
  Identity database
```

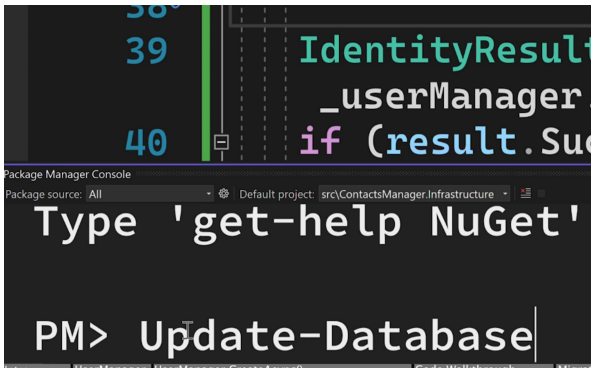
```
//Check for validation errors
if (ModelState.IsValid == false)
{
 ViewBag.Errors = ModelState.Values.SelectMany
  (temp => temp.Errors).Select(temp =>
  temp.ErrorMessage);
 return View(registerDTO);
}

ApplicationUser user = new ApplicationUser()
 { Email = registerDTO.Email, PhoneNumber =
 registerDTO.Phone, UserName = registerDTO.Email,
 PersonName = registerDTO.PersonName };

_userManager.CreateAsync();

//TO DO: Store user registration details into
 Identity database
```

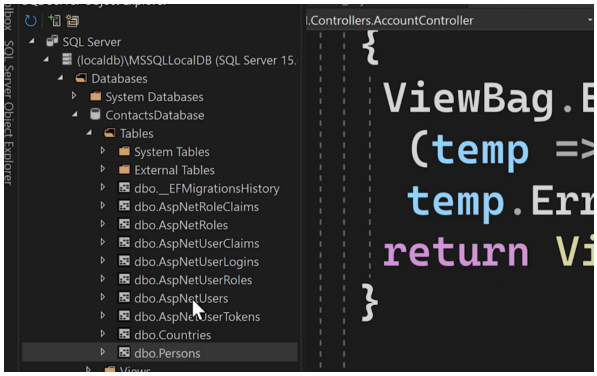
```
{
    //check for validation errors
    if(ModelState.IsValid == false)
    {
        ViewBag.Errors = ModelState.Values.SelectMany(temp => temp.Errors).Select(temp => temp.ErrorMessage
        return View(registerDTO);
    }

    ApplicationUser user = new ApplicationUser
    {
        Email = registerDTO.Email,
        PhoneNumber = registerDTO.Phone,
        UserName  = registerDTO.Email,
        PersonName = registerDTO.PersonName
    };

IdentityResult result = await _userManager.CreateAsync(user);
 //return RedirectToAction(nameof(PersonsController.Index), "Persons");
if (result.Succeeded)
{
        return RedirectToAction(nameof(PersonsController.Index), "Persons");
}
else
{
    foreach(IdentityError error in result.Errors)
    {
        ModelState.AddModelError("Register",error.Description);
    }

    return View(registerDTO);
}
}
```

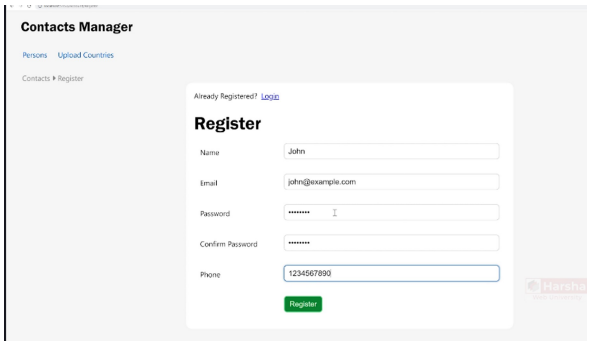


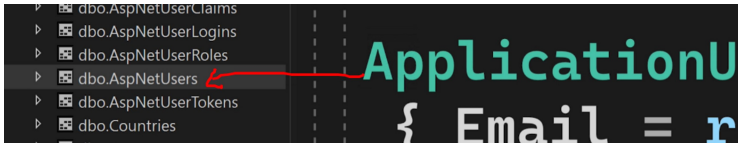
Select 'Infrastrcutre' project because we have dbcontext in there.

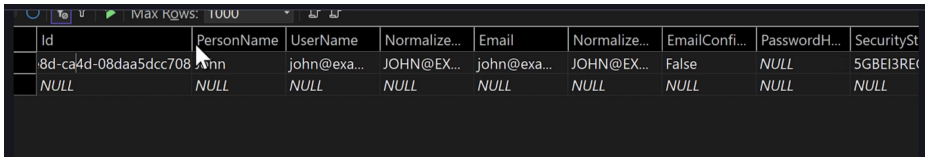
Our application DB context class already inherits from Identity DB context, so the Identity DB context class internally has all the essential DB sets that are required for creation of the tables such as ASP Net Users, ASP Net Roles, and other essential tables. So, we have not created the DB sets ourselves, but by inheriting from Identity DB context, all those DB sets are automatically created. But before running the `Update-Database` command, first, we have to add a migration, right? So, create a migration. The command is `Add-Migration <MigrationName>`, for example.


39
IdentityResult
_userManager.
40
if (result.Su
Package Manager Console
Package source: All
Default project: src\ContactsManager.Infrastructure
Type 'get-help NuGet'
PM> Update-Database


SQL Server
(localdb)\MSSQLLocalDB (SQL Server 15.
Databases
System Databases
ContactsDatabase
Tables
System Tables
External Tables
dbo.__EFMigrationsHistory
dbo.AspNetRoleClaims
dbo.AspNetRoles
dbo.AspNetUserClaims
dbo.AspNetUserLogins
dbo.AspNetUserRoles
dbo.AspNetUsers
dbo.AspNetUserTokens
dbo.Countries
dbo.Persons
.Controllers.AccountController
{
ViewBag.E
(temp =>
temp.Err
return Vi
}


Contacts Manager
Persons
Upload Countries
Contacts ▸ Register
Already Registered? Login
Register
Name
John
Email
john@example.com
Password
Confirm Password
Phone
1234567890
Register


dbo.AspNetUserClaims
dbo.AspNetUserLogins
dbo.AspNetUserRoles
dbo.AspNetUsers
dbo.AspNetUserTokens
dbo.Countries
ApplicationU
{ Email = r

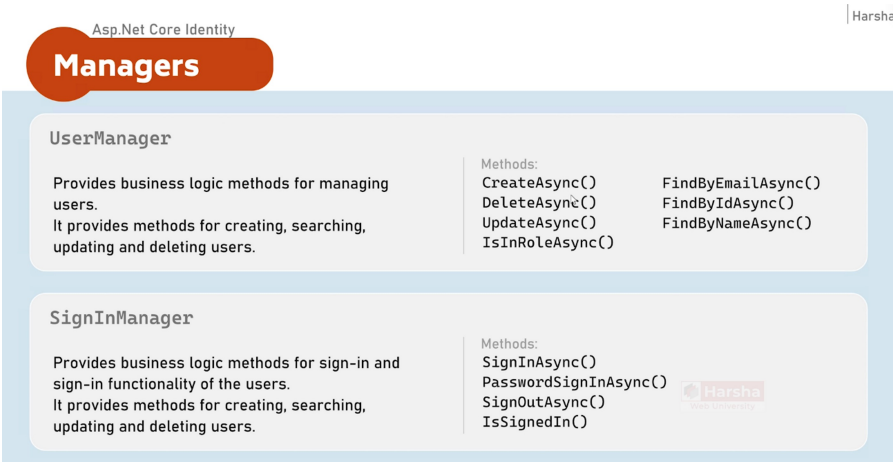

Max Rows: 1000
Id
PersonName
UserName
Normalize...
Email
Normalize...
EmailConfi...
PasswordH...
SecuritySt
8d-ca4d-08daa5dcc708
John
john@exa...
JOHN@EX...
john@exa...
JOHN@EX...
False
NULL
5GBEI3RE(
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL

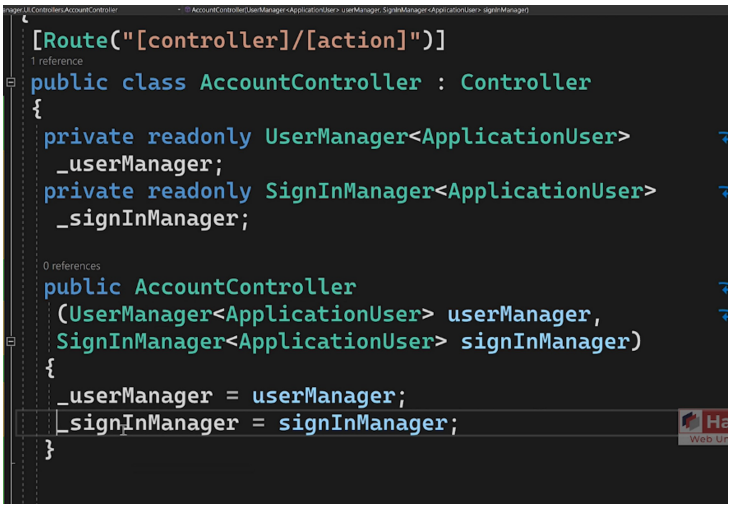

Asp.Net Core Identity
SignInManager

We have already explored the `UserManager` class to manipulate user information. Similarly, the `SignInManager` class provides business logic methods for login and logout functionalities.

For signing in (logging in), there are two primary methods: - If the application user object already exists and needs to sign in, we use the `SignInAsync` method. - If the application user object has not been created yet, but we have a username and password for login, we need to validate the credentials and log in. In this case, we use the `PasswordSignInAsync` method.
The `SignOutAsync` method, as the name suggests, is used to sign out (log out).
To check if the current user is signed in, you can use the `IsSignedIn` method.

When a user signs in, an authentication cookie is automatically created in the browser. This cookie is then automatically submitted with all subsequent requests. For example: - On the login page, when the user signs in with the correct username and password, the authentication cookie is created on the server side and sent to the browser. - The browser stores this cookie in its memory. - When the user navigates to another URL (e.g., `/Persons/Index`), the browser automatically includes the authentication cookie in the request to prove that the user is already logged in. - The server receives the request, verifies the cookie's validity, and, if valid, allows access to the current user's ID or name based on the cookie.
Let me demonstrate this practically.





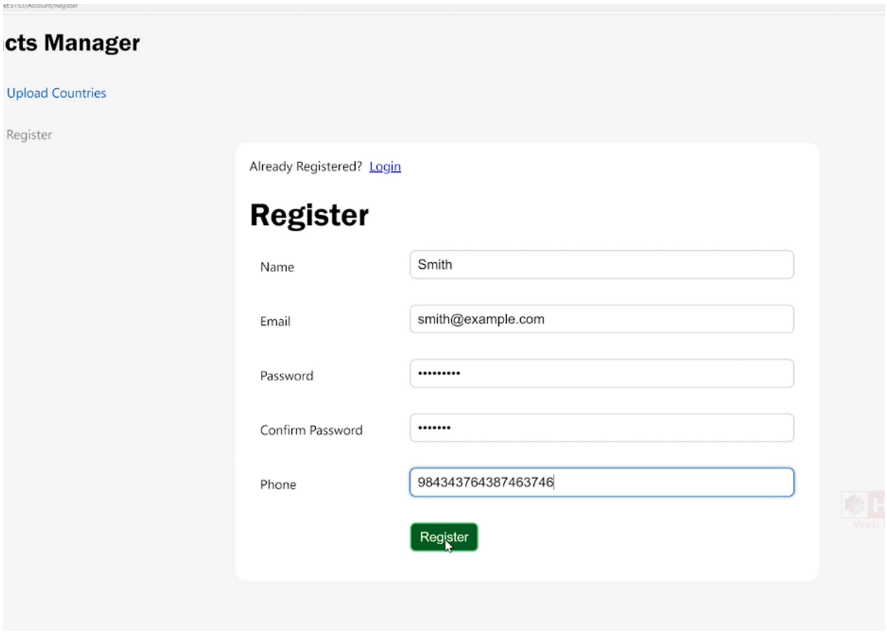
This particular selected user will be logged in. That means it creates a cookie and sends the same to the browser as a part of the response.

```
return View(registerDTO);
}

ApplicationUser user = new ApplicationUser()
 { Email = registerDTO.Email, PhoneNumber =
 registerDTO.Phone, UserName = registerDTO.Email,
 PersonName = registerDTO.PersonName };

IdentityResult result = await
 _userManager.CreateAsync(user,
 registerDTO.Password);
if (result.Succeeded)
{
 //Sign in
 await _signInManager.SignInAsync(user);

 return RedirectToAction(nameof
  (PersonsController.Index), "Persons");
}
```

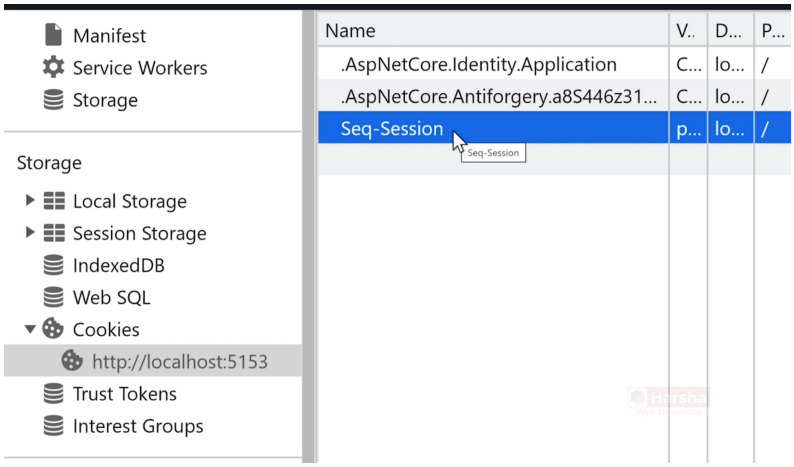
True means the cookie will be persistent into the browser even after the browser is closed.

```
{
 //Sign in
 await _signInManager.SignInAsync(user,
  isPersistent: true);
```

```
<div class="flex-1 pr" id="search-box-div">
    <div class="navbar account-links">

        <ul>
            <li>
                @User.Identity?.Name
            </li>
            <li>
                <a asp-controller="Account" asp-action="Register">Register</a>
            </li>
            <li>
```



After signIn, a cookie is stored to the browser. After signout, cookie will be deleted.



The **authentication middleware** plays a crucial role in processing requests. Here's how it works:

When a request is made to a specific URL (e.g., `/Persons/Index`), and the user is already logged in, an **identity cookie** is present in the browser. This cookie is automatically included in the request's cookies.

The **authentication middleware**:
- Reads the identity cookie.
- Extracts the user ID and username from it.
- Makes this information available in the `User` property, accessible in both the **controller** and the **view**.
This allows you to access the current user's details at any point in an action method or view.

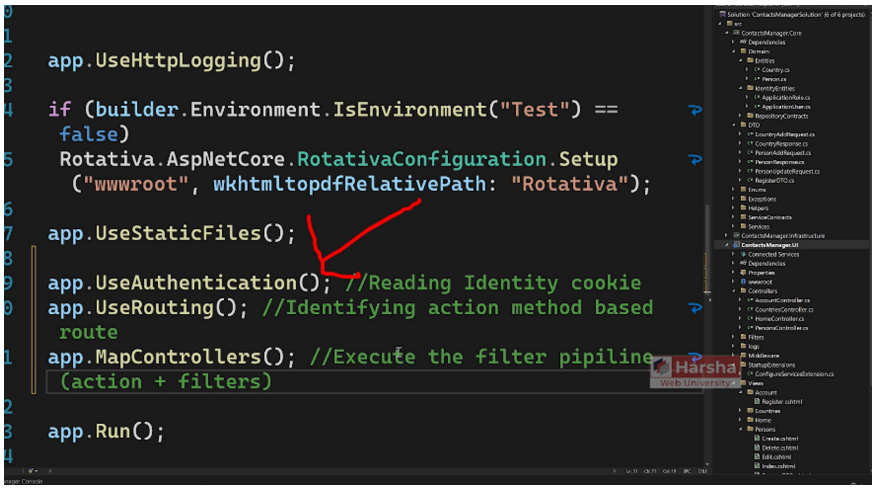
The middleware responsible for this is `UseAuthentication`, and it must be configured **before** `UseRouting` in the middleware pipeline. Here's why:
- `UseRouting` determines which action method to execute based on the requested route.
- `MapControllers` (also known as **endpoint middleware**) executes the selected action method and the associated filter pipeline (as discussed in the previous section).

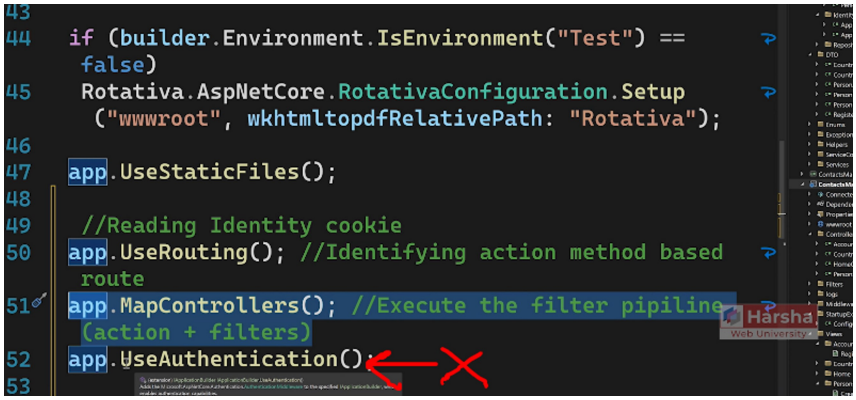
Middleware Responsibilities:
1. **UseAuthentication**: Reads the identity cookie and extracts user details.
2. **UseRouting**: Identifies the appropriate action method based on the route.
3. **MapControllers**: Executes the action method and the filter pipeline (action method + filters).

Importance of Middleware Order:
The sequence of middleware is critical:
- `UseAuthentication` must be placed **above** `UseRouting` in the pipeline.
- If `UseAuthentication` is placed after `MapControllers`, it will not execute before the controller action methods or filters. As a result, the identity cookie won't be read, and the current user's details won't be available in the action methods, filters, or views.

For example, placing `UseAuthentication` after `MapControllers` would render it ineffective because the cookie-reading logic would not have executed by the time the controller or view tries to access the user details.

Thus, `UseAuthentication` must always be configured **above** `UseRouting` to ensure proper functionality.





Asp.Net Core Identity

Login / Logout Buttons

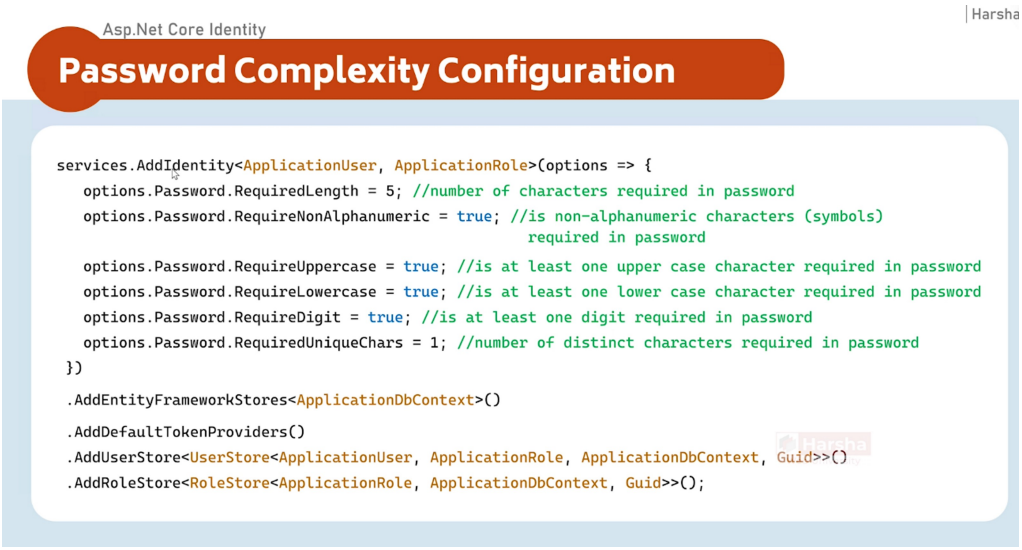
Password Complexity Configuration

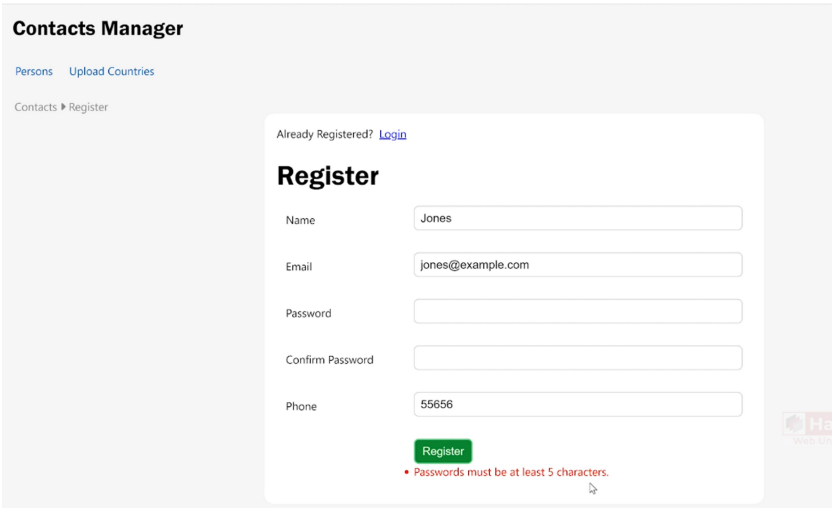
When adding identity services to the services collection, you can configure the complexity requirements for passwords during user registration. These settings allow you to define how strong a password must be. Here's what you can configure:

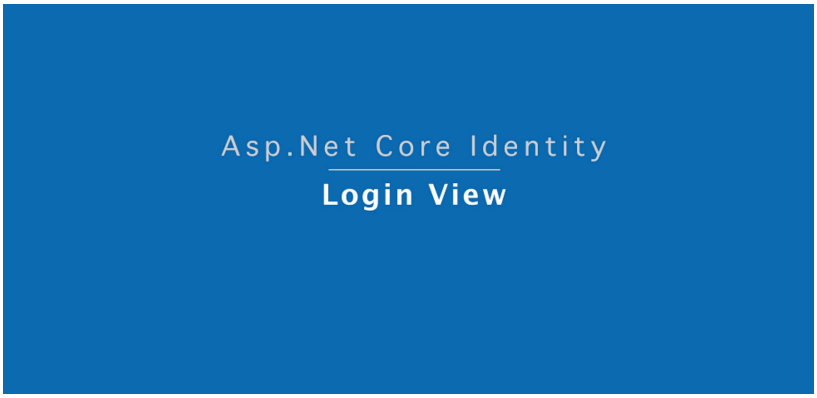
- **Required Length**: Set the minimum length of the password (e.g., 8 or 10 characters).
- **Non-Alphanumeric Characters**: Specify whether the password must include at least one non-alphanumeric character (e.g., symbols like `#`, `@`, `$`, etc.) by setting `true` or `false`.
- **Uppercase Letters**: Define whether at least one uppercase letter is required (`true` or `false`).
- **Lowercase Letters**: Define whether at least one lowercase letter is required (`true` or `false`).
- **Digits**: Specify whether at least one digit is required (`true` or `false`).
- **Unique Characters**: Set the minimum number of distinct characters required in the password to prevent repetitive characters (e.g., a password like `aaaaaa` is weak). This ensures a stronger password by enforcing diversity in characters.

These properties are configured using a lambda expression when adding identity services to the services collection.

Let me demonstrate this practically.







Creating the Login View

So far, we have created the registration page. Now, let's focus on creating the **login page**. Here's the process:

User Input: The user enters their **email** and **password** on the login page.

Form Submission: When the user clicks the **submit button**, the application validates the provided email and password against the database.

Validation:

- If the email and password match a record in the database, the user is authenticated.
- The SignInManager is used to sign in the user, which creates an **identity cookie** in the browser.

Identity Cookie:

- As long as the identity cookie exists in the browser, the user is considered **logged in**.
- If multiple users access the application from different browsers, each browser stores the corresponding identity cookie for the respective user (e.g., Browser 1 stores User1's cookie, Browser 2 stores User2's

cookie).

Server-Side Behavior:
- The server does not store identity cookies. It is the browser's responsibility to include the identity cookie in the Request.Cookies for every subsequent request.
- If the server receives a valid identity cookie from the browser, it assumes the user associated with that cookie is logged in.

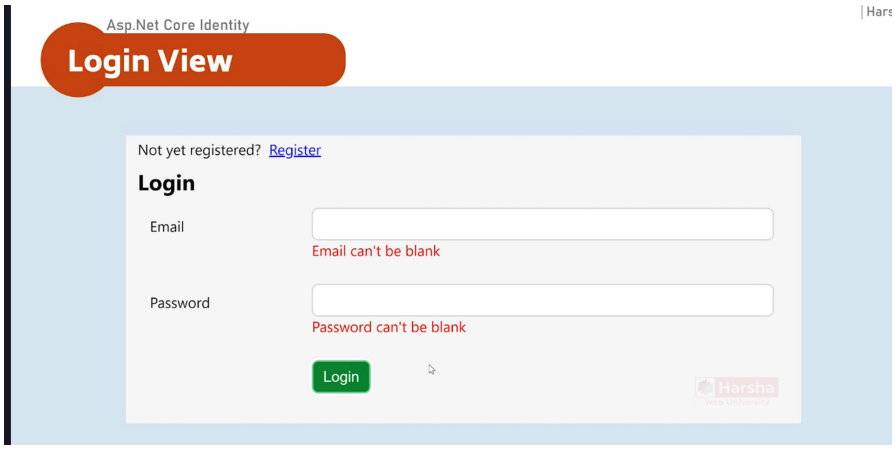
Steps to Create the Login View
Let's create the **login view** to implement this functionality. The view will include:
- A form for the user to input their email and password.
- A submit button to trigger the validation and sign-in process.

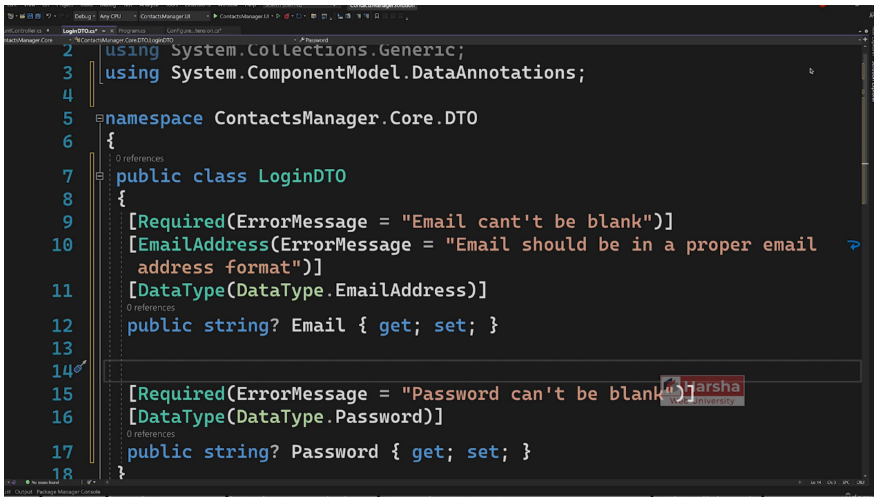
Below is an example of how the login view can be structured (we'll demonstrate the practical implementation next):

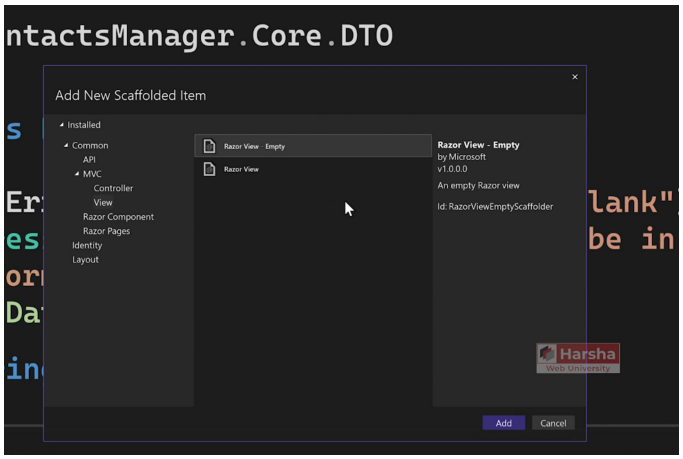
- **View (Razor)**: A form with fields for email and password, and a submit button.

- **Controller**: An action method to handle the form submission, validate credentials using SignInManager, and create the identity cookie upon successful login.

Let's proceed with creating the login view and the associated logic.



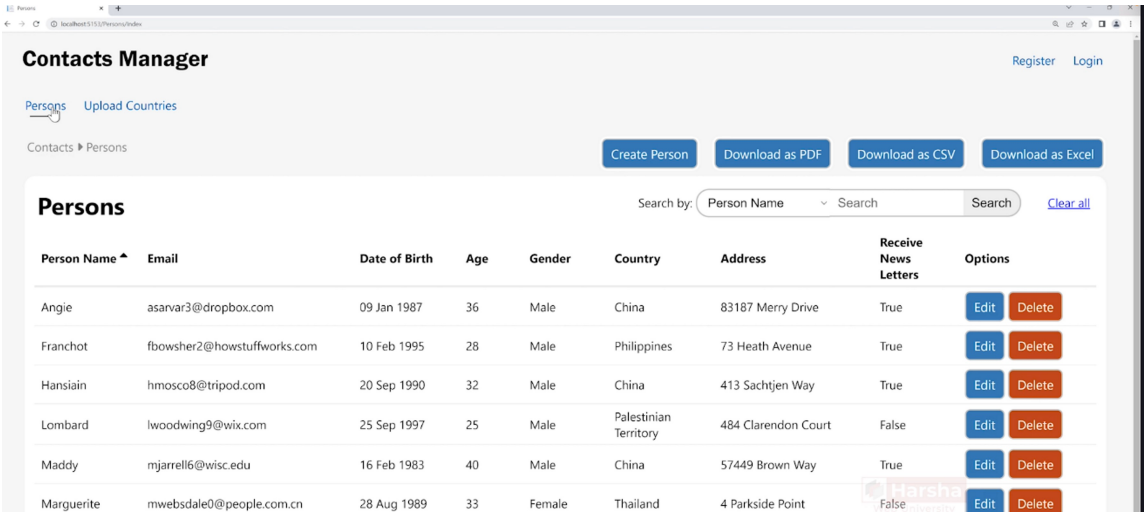


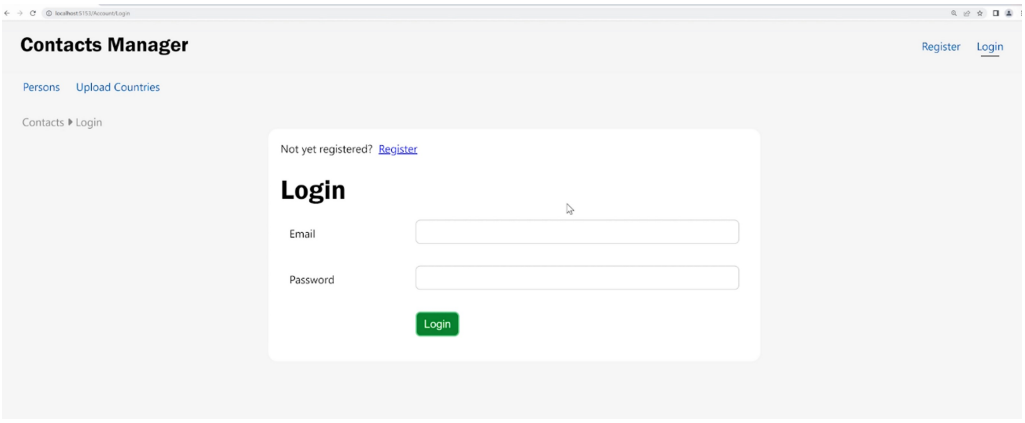



```
[HttpGet]
0 references
public IActionResult Login()
{
    return View();
}

[HttpPost]
0 references
public async Task<IActionResult> Login(LoginDTO loginDTO)
{
    if(ModelState.IsValid == false)
    {
        ViewBag.Errors = ModelState.Values.SelectMany(temp => temp.Errors).Select(temp => temp.ErrorMessage);
        return View(loginDTO);
    }

    var result = await _signInManager.PasswordSignInAsync(loginDTO.Email, loginDTO.Password, isPersistent: false,lockoutOnFailure:false);

    if(result.Succeeded)
    {
        return RedirectToAction(nameof(PersonsController.Index), "Persons");
    }
    else
    {
        ModelState.AddModelError("Login", "Invalid login attempt");
    }

    return View(loginDTO);
}
```

Asp.Net Core Identity

Authorization Policy





Configuring Authorization Policy for Restricted Page Access Requirement
All pages, such as the **Persons page**, **Upload page**, etc., should only be accessible to users who are logged in. If a user is not logged in and attempts to access these pages (e.g., by clicking hyperlinks), the request should be rejected, and the user should be automatically redirected to the **login page**. This behavior is similar to websites like Gmail, where pages like the inbox or sent items are only accessible after account login.

Solution
To enforce this, we need to implement an **authorization policy** in the application. This policy ensures that requests to protected pages include a valid **identity cookie**. If the identity cookie is not present (indicating the user is not logged in), the user is redirected to the /Account/LoginURL.

Steps to Implement

Add Authorization Services:
- Manually configure the authorization service in the application's service collection using AddAuthorization.
- Define an authorization policy that requires authenticated users (i.e., requests must include a valid identity cookie).

Apply the Policy:
- Enforce the policy globally or on specific controllers/actions to restrict access to authenticated users only.
- Configure the application to redirect unauthenticated users to the login page.

Asp.Net Core Identity

Authorization Policy

Harsha

```
services.AddAuthorization(options =>
  {
   var policy = new AuthorizationPolicyBuilder().RequireAuthenticatedUser().Build();
   options.FallbackPolicy = policy;
  });
```

Open Program.cs file, these 4 methods should be in order.

```
48
49
50  app.UseRouting(); //Identifying action method based on route
51  app.UseAuthentication(); //Reading Identity cookie
52  app.UseAuthorization(); //Validates access permissions of the user
53  app.MapControllers(); //Execute the filter pipiline (action +
     filters)
54
55  app.Run();
```

Open ConfigureServiceExtension.cs file

```
     Guid>>();
74
75
76   services.AddAuthorization(options =>
77   {
78    options.FallbackPolicy = new AuthorizationPolicyBuilder
      ().RequireAuthenticatedUser()
79   });
80   services.AddHttpLogging(options =>
81   {
82    options.LoggingFields =
```

What this policy represent is that, it ensures a rule for all the user such that, in order to access any action method of the entire application, the user must be logged in. That means the request should have 'identity cookie'. If not, it will redirect to 'account/login' page.

```
Guid>>();

services.AddAuthorization(options =>
{
 options.FallbackPolicy = new AuthorizationPolicyBuilder
 ().RequireAuthenticatedUser().Build(); //enforces authoriation
  policy (user must be authenticated) for all the action
 methods
});

services.ConfigureApplicationCookie(options => {
 options.LoginPath = "~/Account/Login";
});

services.AddHttpLogging(options =>
```



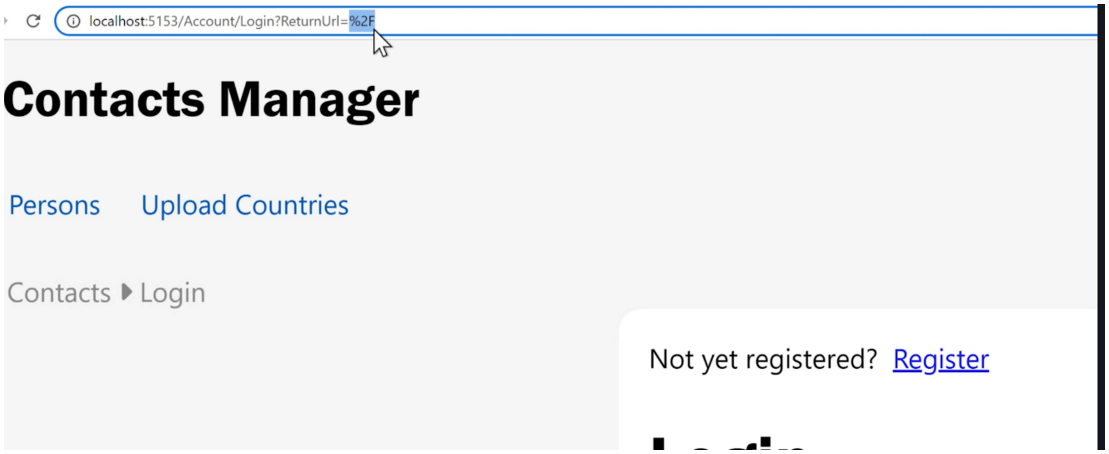
```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;

namespace ContactsManager.UI.Controllers
{
 [Route("[controller]/[action]")]
 [AllowAnonymous]
 1 reference
 public class AccountController : Controller
 {
  private readonly UserManager<ApplicationUser> _userManager;
  private readonly SignInManager<ApplicationUser> _signInManager;

  0 references
  public AccountController(UserManager<ApplicationUser>
```

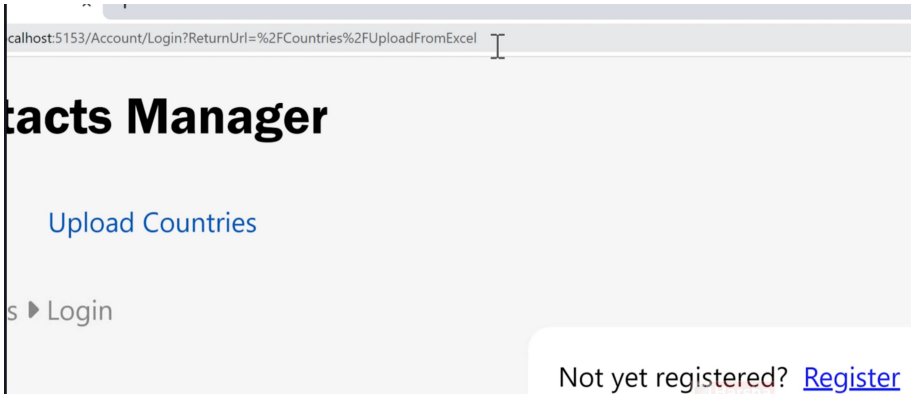
user without user account

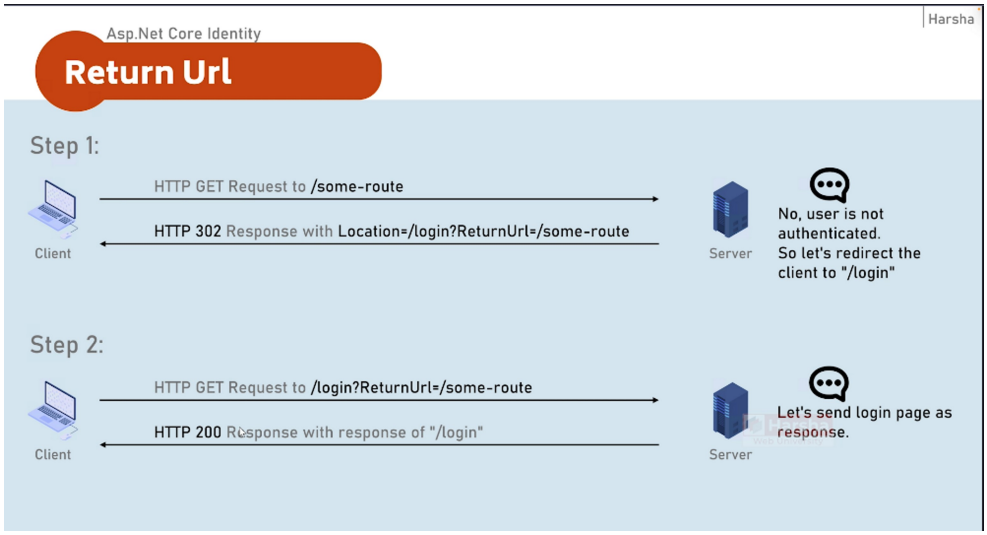
Asp.Net Core Identity

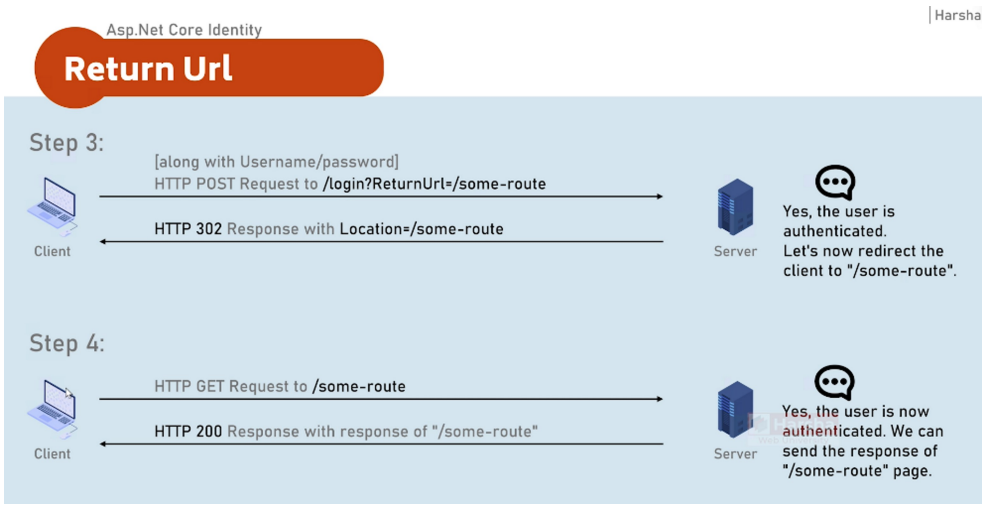
ReturnUrl

```
 .AddRoleStore<RoleStore<ApplicationRole, ApplicationDbContext
 Guid>>();

services.AddAuthorization(options =>
{
 options.FallbackPolicy = new AuthorizationPolicyBuilder
 ().RequireAuthenticatedUser().Build(); //enforces authoriati
  policy (user must be authenticated) for all the action
 methods
});

services.ConfigureApplicationCookie(options => {
 options.LoginPath = "/Account/Login";
});

services.AddHttpLogging(options =>
```


localhost:5153/Account/Login?ReturnUrl=%2F
Contacts Manager
Persons
Upload Countries
Contacts ▸ Login
Not yet registered? Register


calhost:5153/Account/Login?ReturnUrl=%2FCountries%2FUploadFromExcel
tacts Manager
Upload Countries
s ▸ Login
Not yet registered? Register


Harsha
Asp.Net Core Identity
Return Url
Step 1:
HTTP GET Request to /some-route
HTTP 302 Response with Location=/login?ReturnUrl=/some-route
Client
Server
No, user is not authenticated. So let's redirect the client to "/login"
Step 2:
HTTP GET Request to /login?ReturnUrl=/some-route
HTTP 200 Response with response of "/login"
Client
Server
Let's send login page as response.



Handling Return URL in Login Flow

Overview

In the service collection, we configured the application cookie authentication with the login path set to /Account/Login. This means that if a user is not logged in and tries to access a protected page, they are automatically redirected to the login page. Additionally, a query string parameter called returnUrlis appended to the login URL, containing the **originally requested URL** (encoded). For example, if the user tries to access /Persons/Indexwithout being logged in, the redirect URL might look like:
/Account/Login?returnUrl=%2FPersons%2FIndex

Here, %2Frepresents the forward slash (/) due to URL encoding, so returnUrl=/Persons/Index.

Purpose of returnUrl

The returnUrlparameter serves to track the original URL the user attempted to access. After successful login, the application should redirect the user to this URL, providing a seamless user experience. For instance, if the user tried to access /Countries/UploadFromExcelbut was redirected to the login page, the returnUrlwould be /Countries/UploadFromExcel. After logging in, the user should be redirected back to /Countries/UploadFromExcel.

Developer Responsibility

As developers, we need to:

Capture the returnUrlquery string parameter in the login action method.
After successful authentication, redirect the user to the returnUrl(if valid) using a redirect result.

How It Works Internally

Below is a step-by-step explanation of the login flow with returnUrl:

Initial Request to Protected Route:
- The user requests a protected URL, e.g., /Persons/Index.
- Since the user is not authenticated, the ASP.NET Core authorization policy triggers a redirect to the login page (/Account/Login).
- The server returns an **HTTP 302 (Found)** response with a Locationheader set to:
  /Account/Login?returnUrl=%2FPersons%2FIndex
  Here, returnUrlholds the encoded value of the original URL (/Persons/Index).

Browser Redirects to Login Page:
- The browser receives the HTTP 302 response and automatically sends a **GET** request to /Account/Login?returnUrl=%2FPersons%2FIndex.
- The server responds with an **HTTP 200 (OK)** response, rendering the login view (the login form).

User Submits Login Form:
- The user enters their email and password in the login form and clicks the submit button.
- This triggers a **POST** request to /Account/Login, including the returnUrlas a query string parameter (e.g., /Account/Login?returnUrl=%2FPersons%2FIndex) and the form data (email and password).

Server Authenticates User:
- The server receives the POST request and validates the credentials using SignInManager.
- If the credentials are correct, the server:
  - Signs in the user, creating an identity cookie.
  - Checks the returnUrlparameter to ensure it is valid (e.g., using Url.IsLocalUrlto prevent open redirect attacks).
  - Returns an **HTTP 302 (Found)** response with the Locationheader set to the returnUrl(e.g., /Persons/Index).

Browser Redirects to Original URL:
- The browser receives the HTTP 302 response and sends a **GET** request to the returnUrl(e.g., /Persons/Index).
- Since the user is now authenticated (the identity cookie is included in the request), the server processes the request and returns an **HTTP 200 (OK)** response with the Persons/Indexview.

User Sees the Requested Page:
- The user is now on the /Persons/Indexpage, as originally intended.

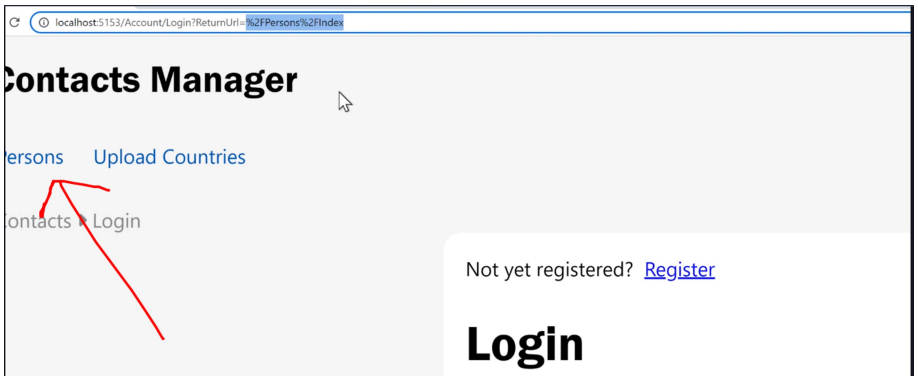
Key Points

- The returnUrlquery string parameter carries the original requested URL across requests and responses, ensuring the user is redirected to their intended destination after login.
- The server does not store the returnUrl; it relies on the browser to include it in requests.
- The login action method (POST) must handle the returnUrland issue a redirect to it after successful authentication.

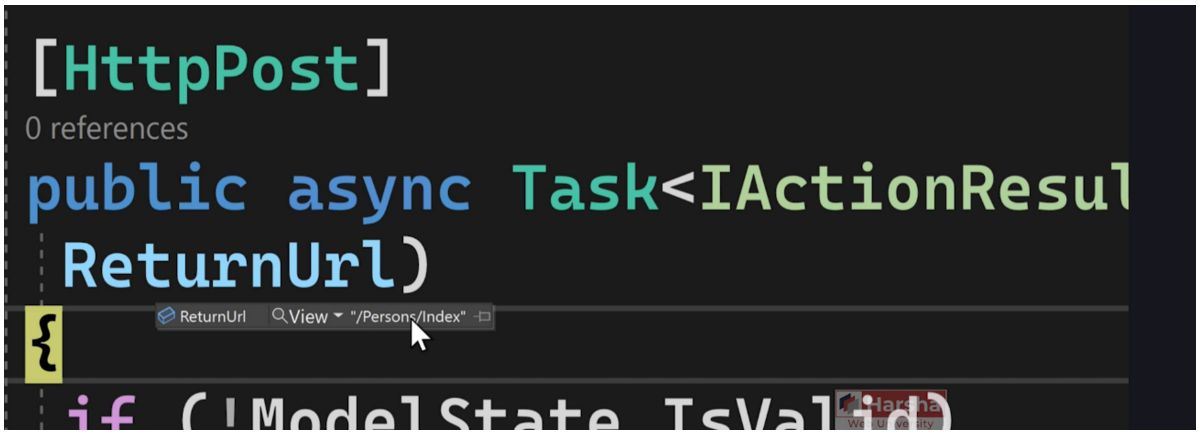
```
<i class="fa-solid fa-caret-right"></i>
<span>Login</span>
</div>

<div class="form-container">
Not yet registered? <a asp-action="Register" asp-
controller="Account" class="link-hover">Register</a>
<h2 class="text-large">Login</h2>

<form asp-controller="Account" asp-action="Login" asp-route-
ReturnUrl="@Context.Request.Query["ReturnUrl"]" method="post">

@*Email*@
<div class="form-field flex">
<div class="w-25">
```

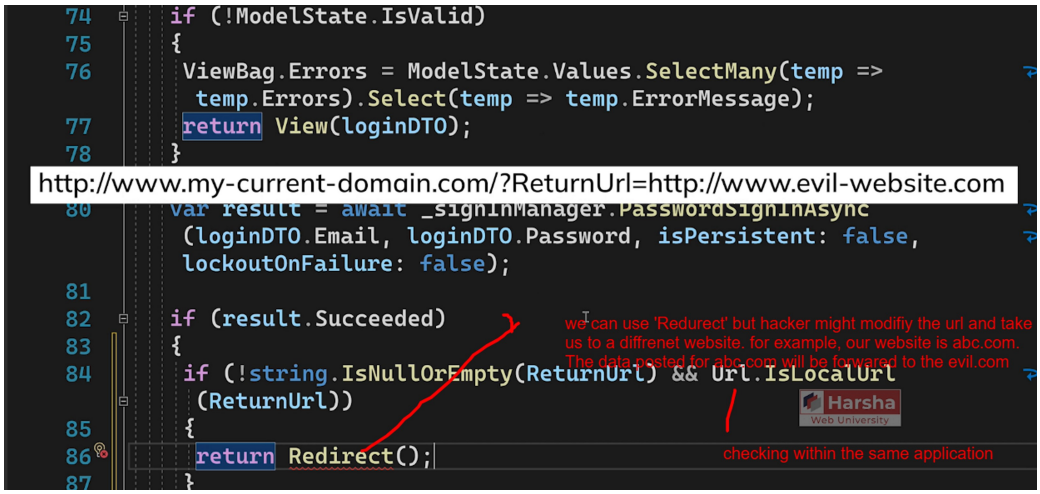
```
70
71      [HttpPost]
        0 references
72      public async Task<IActionResult> Login(LoginDTO loginDTO, string?
          ReturnUrl)
73      {
74        if (!ModelState.IsValid)
75        {
76          ViewBag.Errors = ModelState.Values.SelectMany(temp =>
            temp.Errors).Select(temp => temp.ErrorMessage);
77          return View(loginDTO);
78        }
```

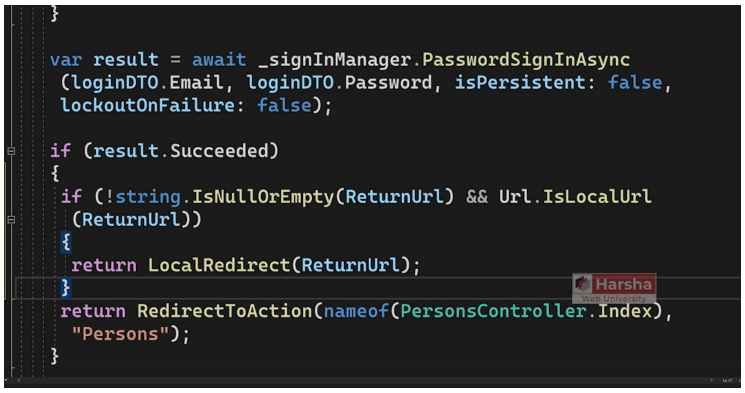
I have clicked on 'Person' hyperlink



Now fill up ID and pass and click on 'Login'

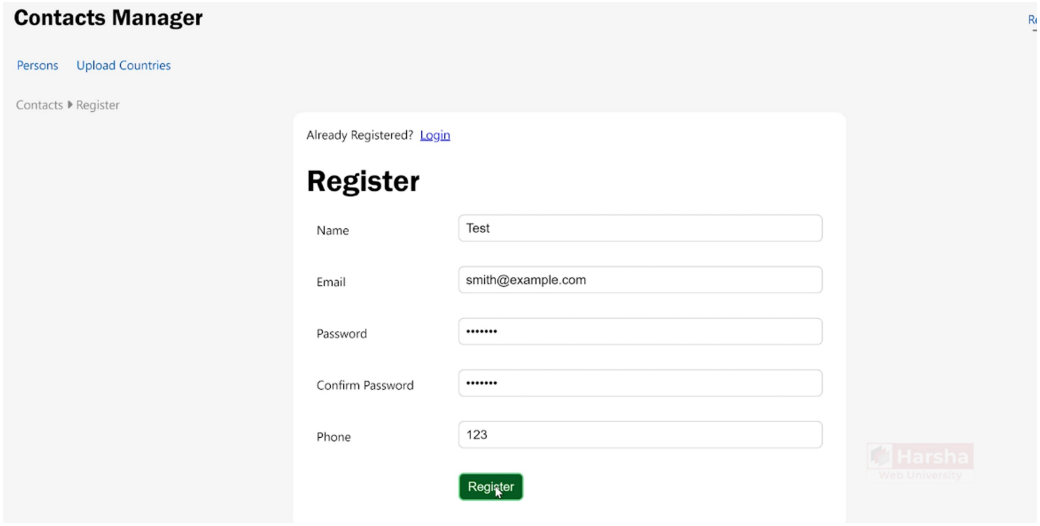


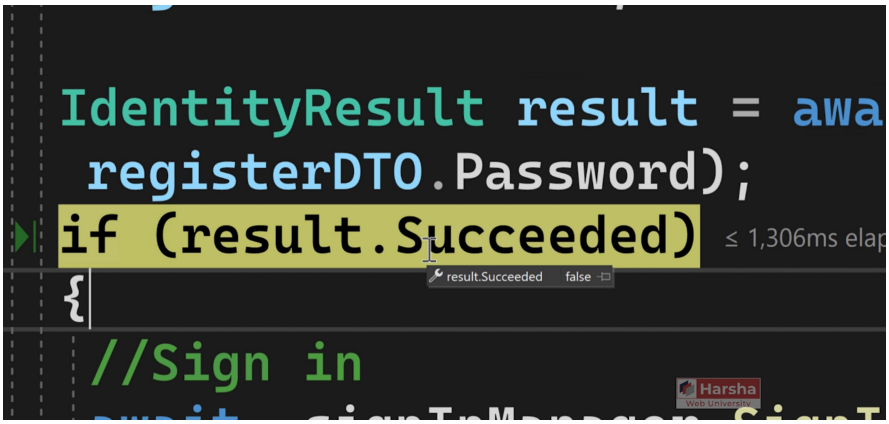




Asp.Net Core Identity

Remote Validation





Client-Side Email Validation to Prevent Duplicate Email Registration
Problem

In the registration page, if a user enters an email address that already exists in the database (e.g., smith@example.com), the server detects the duplicate during the POSTrequest when calling UserManager.CreateAsync. The result indicates failure (Succeeded = false) with a validation error stating that the email is already taken. This triggers a full page refresh, which:

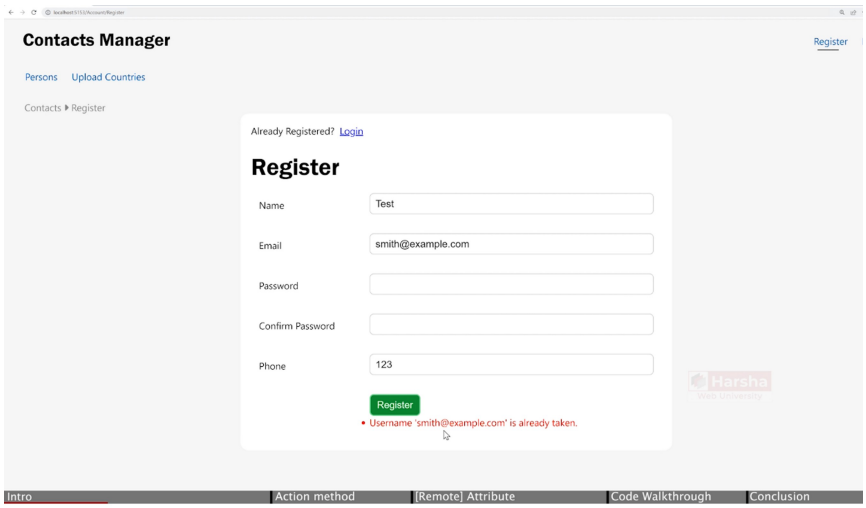
- Retains the entered values in most text boxes (except password fields, which are cleared for security reasons).
- Causes an unnecessary page reload, degrading the user experience.

The browser cannot validate the email's uniqueness client-side because it lacks access to the database. To improve this, we can implement **client-side asynchronous validation** to check the email's uniqueness without a full page refresh.
Solution

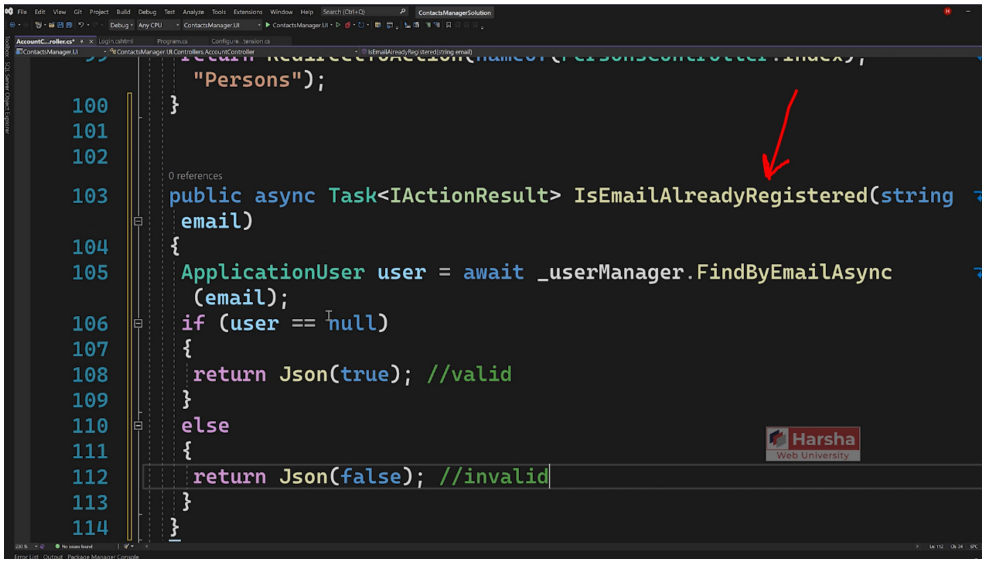
To avoid page refreshes, we can:
Make an **asynchronous HTTP GET request** from the browser when the user enters an email address.
Validate the email on the server and return true(valid) or false(invalid, already exists).
Display an error message on the client side without reloading the page.

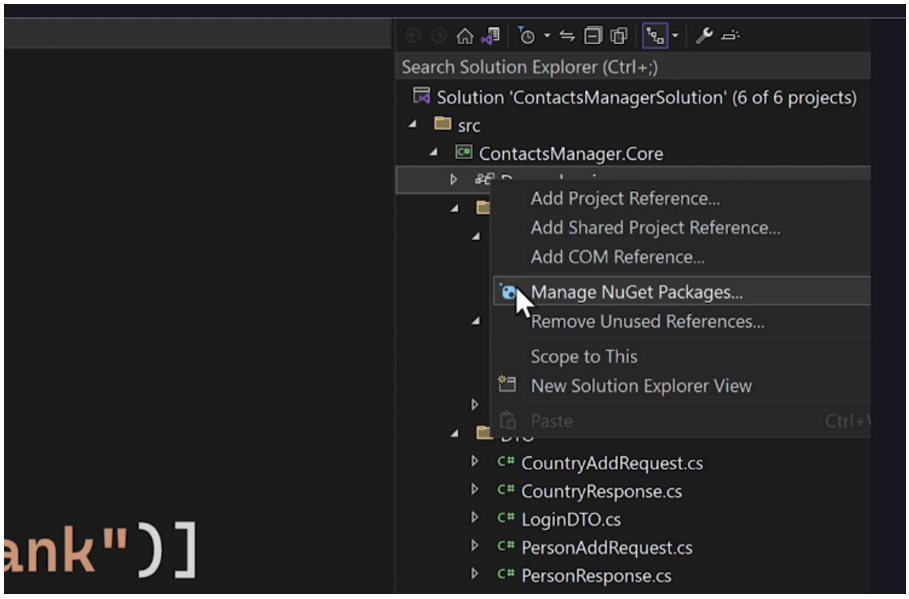
Fortunately, ASP.NET Core simplifies this process by automatically generating the necessary JavaScript code for asynchronous validation. We only need to:

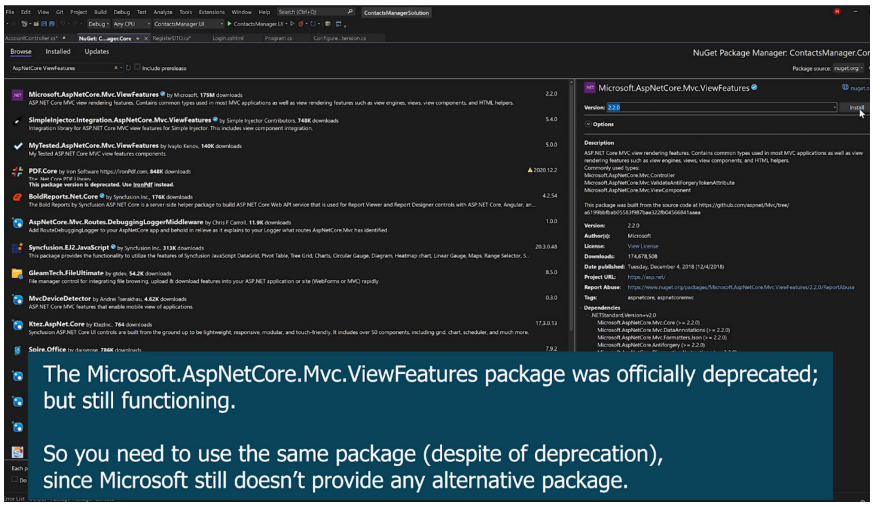
- Create an **action method** in the controller to validate the email and return a boolean result.
- Apply **remote validation** to the email property in the model using the [Remote]attribute.


Contacts Manager
Register
Persons
Upload Countries
Contacts ▸ Register
Already Registered? Login
Register
Name
Test
Email
smith@example.com
Password
Confirm Password
Phone
123
Register
• Username 'smith@example.com' is already taken.
Intro
Action method
[Remote] Attribute
Code Walkthrough
Conclusion


Harsha
Asp.Net Core Identity
Remote Validation
Async HTTP GET Request to /Controller/Action?PropertyName=value
HTTP 200 Response [true or false]
true: valid
false: invalid
Client
Server
Validate the value & return either true or false that indicates whether the value is valid or invalid.
Model class
public class ModelClassName
{
[Remote(action: "action name", controller: "controller name", ErrorMessage = "error message")]
public type PropertyName { get; set; }
}

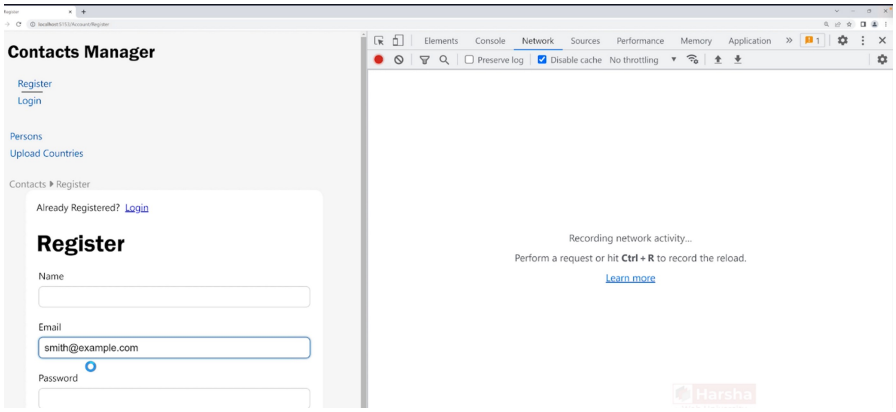

"Persons");
}
0 references
public async Task<IActionResult> IsEmailAlreadyRegistered(string email)
{
ApplicationUser user = await _userManager.FindByEmailAsync(email);
if (user == null)
{
return Json(true); //valid
}
else
{
return Json(false); //invalid
}
}
Harsha
Web University

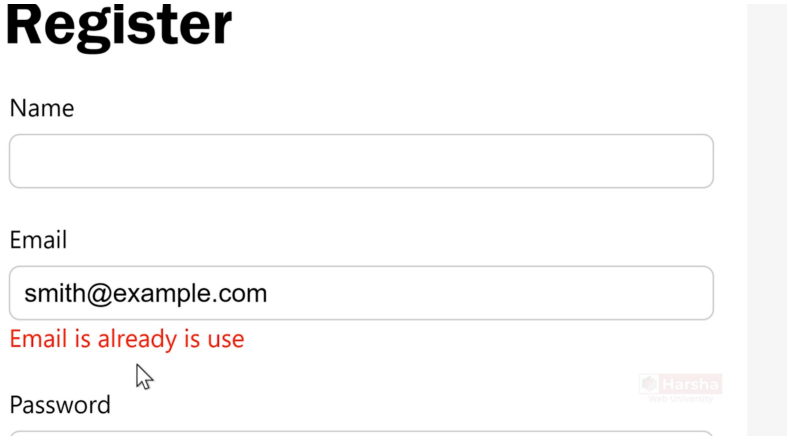





```
namespace ContactsManager.Core.DTO
{
    4 references
    public class RegisterDTO
    {
        [Required(ErrorMessage = "Name can't be blank")]
        4 references
        public string PersonName { get; set; }


        [Required(ErrorMessage = "Email can't be blank")]
        [EmailAddress(ErrorMessage = "Email should be in a proper email address format")]
        [Remote(action: "IsEmailAlreadyRegistered", controller: "Account", ErrorMessage = "Email is already is use")]
        5 references
        public string Email { get; set; }
```

When I type email and press the tab key, the browser automatically sends a request





Asp.Net Core

Conventional Routing

Asp.Net Core | Harsha

Conventional Routing

Conventional routing is a type of routing system in asp.net core that defines route templates applied on all controllers in the entire application.

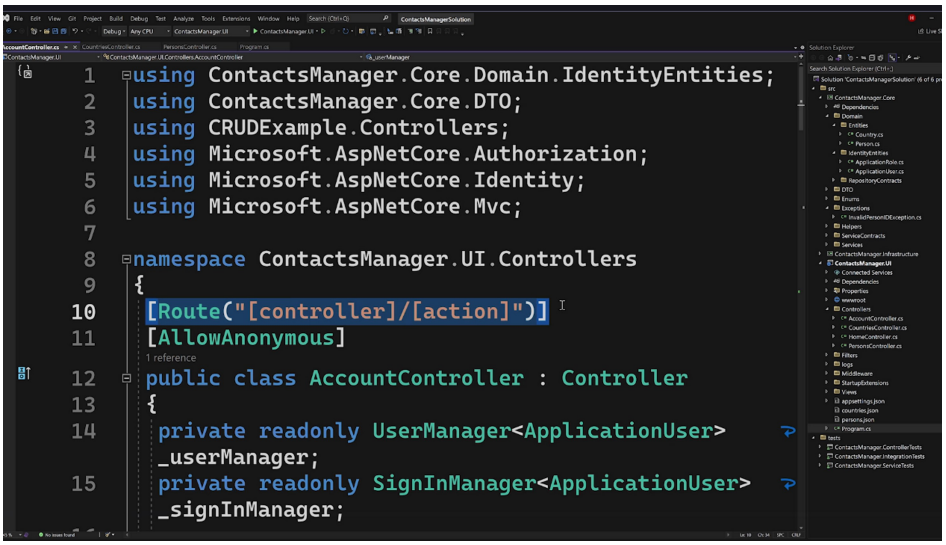
You can override this using attribute routing on a specific action method.

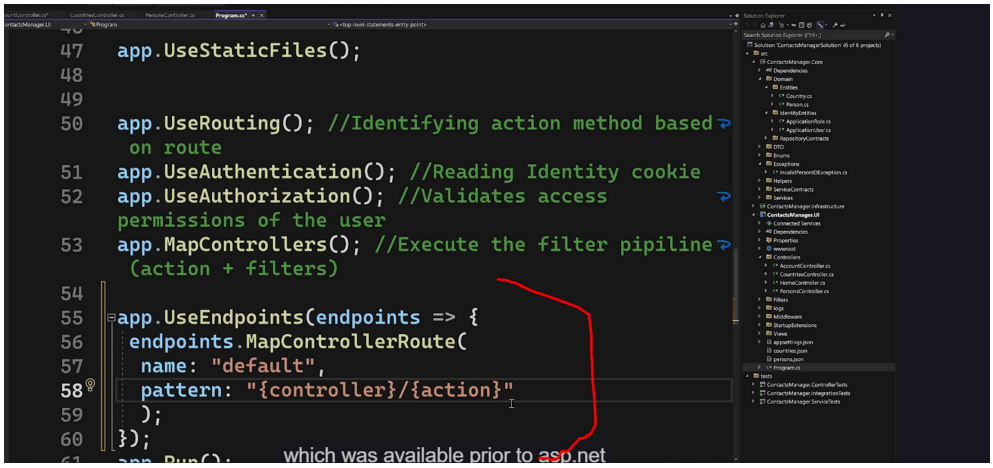
```
endpoints.MapControllerRoute(
  name: "default",
  pattern: "{controller=Persons}/{action=Index}/{id?}"
);
```

ASP.NET Core supports two types of routing: the first one is attribute routing and the other one is conventional routing. We have already used attribute routing in almost all the lectures from the beginning till now, but we are going to use conventional routing just for this particular lecture. Let's try to compare conventional routing and attribute routing and find out which is better. Attribute routing is more flexible. Conventional routing is only applicable if all the controllers have the same routing template, not different for each action method or each controller.

For example, you have a simple web application and it contains very few controllers with few action methods, and assume that almost all the controllers and action methods try to apply the same routing template, for example, first controller name and then action name and then optionally ID. So, if you want to apply the same routing template for all the controllers and all the action methods of the entire application, then conventional routing is better. But what if your application grows or you would like to define your routes based on the user-specific requirements? You want to make the routes clear, easy to understand for the end users, or sometimes for search engine optimization. In that case, defining your own routes individually for every action is a better choice, and that is exactly done by attribute routing. So, most of the ASP.NET Core developers prefer using attribute routing except for very small applications, maybe in practice time. So, in practice time, it's okay to use conventional routing, but for medium to large-scale applications where an extreme level of flexibility in defining the routes is required, attribute routing is recommended.

Let me show an example of conventional routing. For example, in this particular application, if you focus on these controllers, particularly an account controller, almost all the action methods are going to have the same routing template, that is, controller slash action. So, instead of defining for each controller and each action method individually, we can try to define the same globally for the entire application. Of course, attribute routing is applicable for specific action methods which you don't want to apply the conventional routing. For example, let's say we want to apply the conventional routing, meaning a common routing template, for this particular account controller, so comment out this route.





Managing User Roles in ASP.NET Core for Large Web Applications

Overview

In large web applications, such as an **insurance website**, multiple user roles exist, each with distinct access privileges. For example:

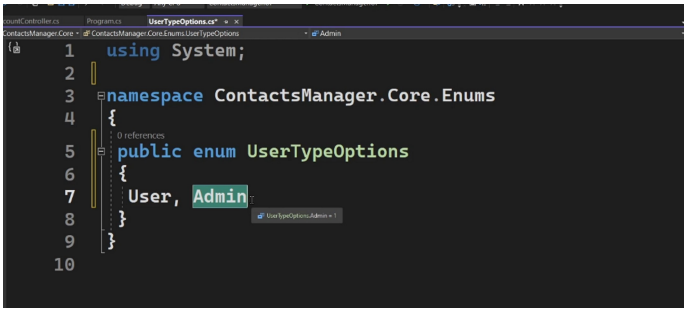
- **Administrators**: Manage the entire website.
- **Customers**: Purchase insurance policies.
- **Employees**: Perform tasks within the insurance company.

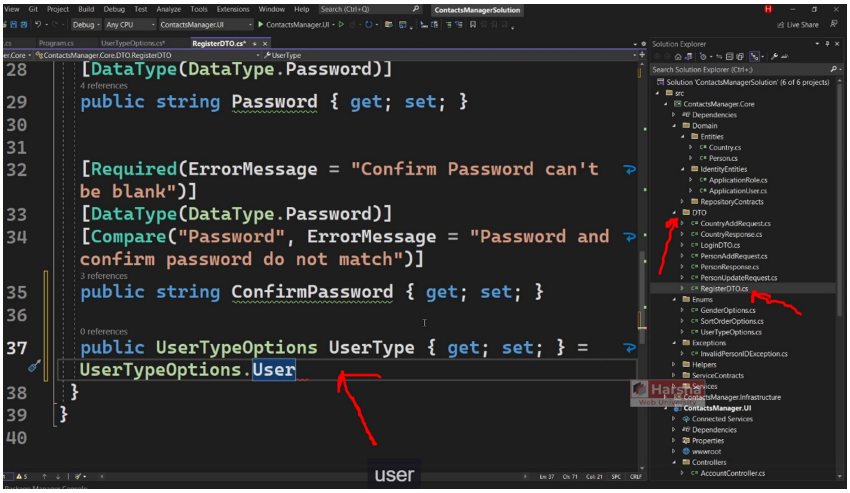
Each **user role** has access to a specific set of pages, and certain pages are restricted to specific roles. For instance:

- Administrator-only pages (e.g., site management) should not be accessible to customers or employees.
- Employee-only pages (e.g., internal tools) should not be accessible to customers or administrators.
- Customer-only pages (e.g., policy purchase) should be restricted to customers.

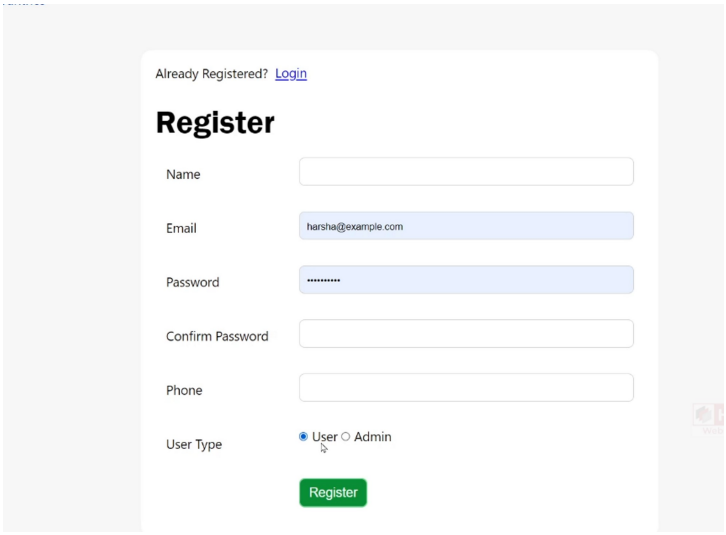
As a developer, you need to create and manage **user roles** and control access to pages based on these roles. This guide explains how to implement role-based authorization in ASP.NET Core.

Let's say in our application, we have two type of user.

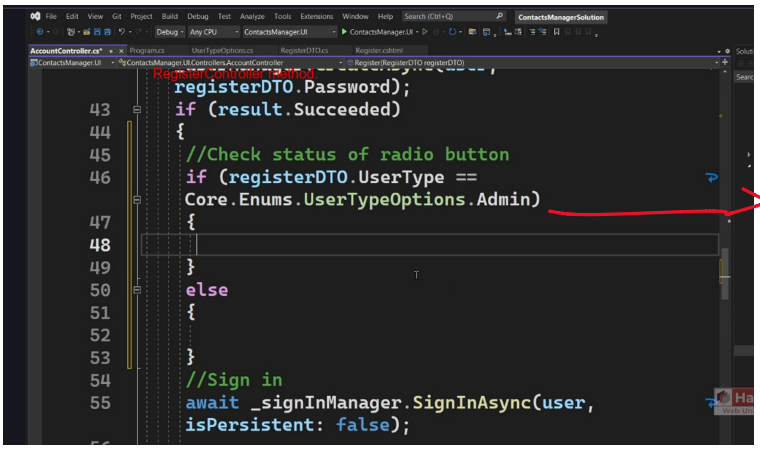




```
<input asp-for="Phone" class="form-input" />
<span asp-validation-for="Phone" class="text-red"></span>
</div>
</div>


@*User Type*@
<div class="form-field flex">
    <div class="w-25">
        <label asp-for="UserType" class="form-label pt">User Type</label>
    </div>
    <div class="flex-1">
        <input type="radio" id="User" asp-for="UserType" value="User" checked />
        <label for="User">User</label>

        <input type="radio" id="Admin" asp-for="UserType" value="Admin" />
        <label for="Admin">Admin</label>
    </div>
</div>
```



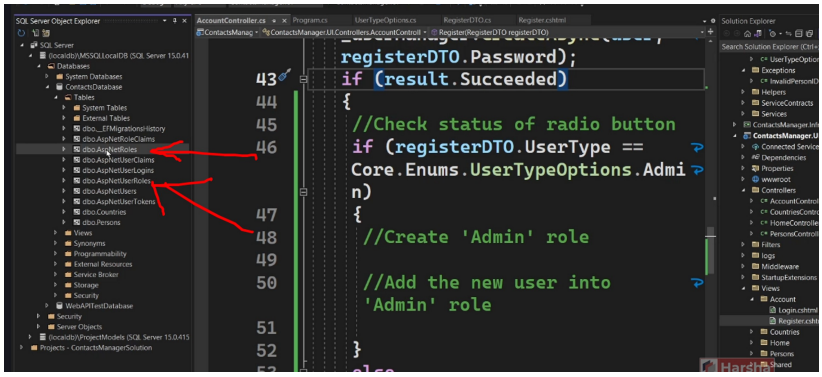
Regsiter() method of AuthController class.



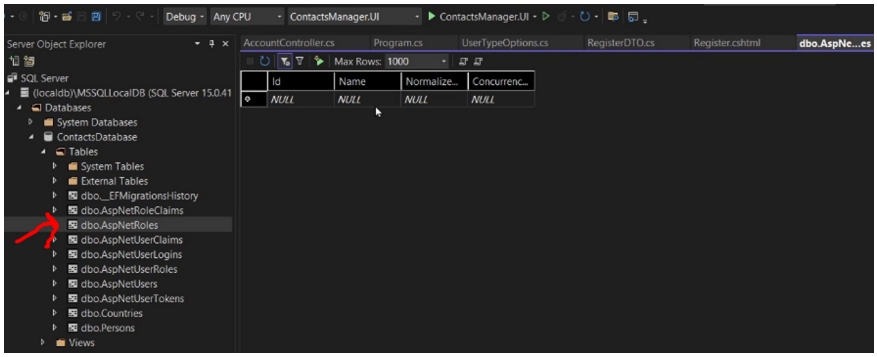
```
registerDTO.Password);
if (result.Succeeded)
{
 //Check status of radio button
 if (registerDTO.UserType ==
 Core.Enums.UserTypeOptions.Admin)
 {

 }
 else
 {

 }
 //Sign in
 await _signInManager.SignInAsync(user,
 isPersistent: false);
```

If it is true, we have to add 'admin' role to the asp.net roles table. Then we have to add particular new user into the admin role.

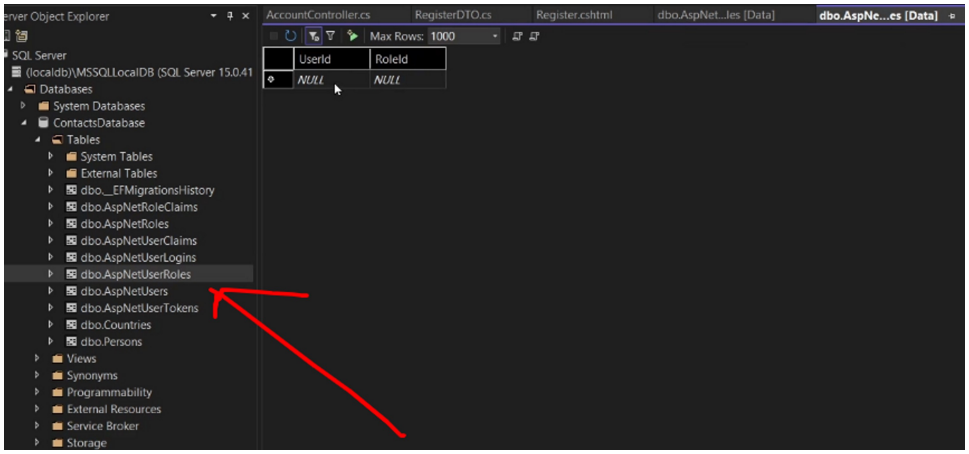
Asp.net roles tables stores all the available roles as a plain list. Ex: Customer Role, Employee Role etc.

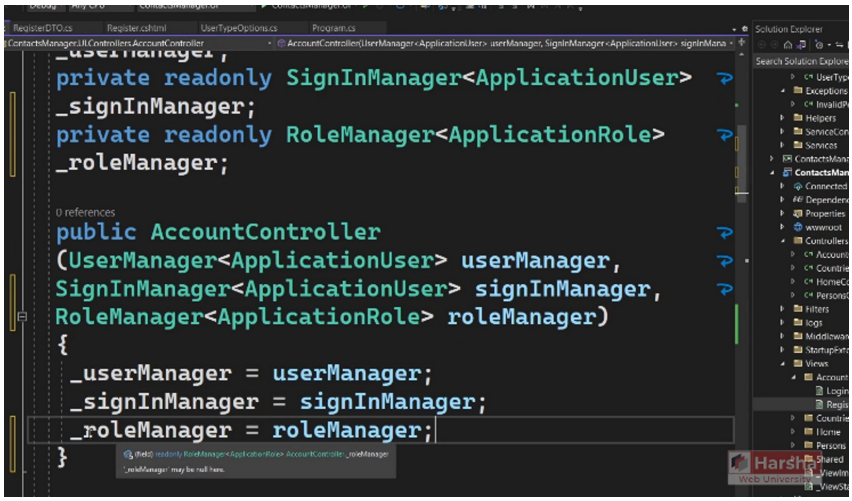


But in this particular table, it doesn't store which users are related to which role.



For that, there is separate table, It contains two foreign keys.





Creating an Admin Area in ASP.NET Core

Overview
In ASP.NET Core, an **area** is a way to organize related **controllers**, **views**, and **models** into separate groups, typically based on modules or user roles. Areas help maintain a clean project structure by isolating components related to specific

functionality or user types.

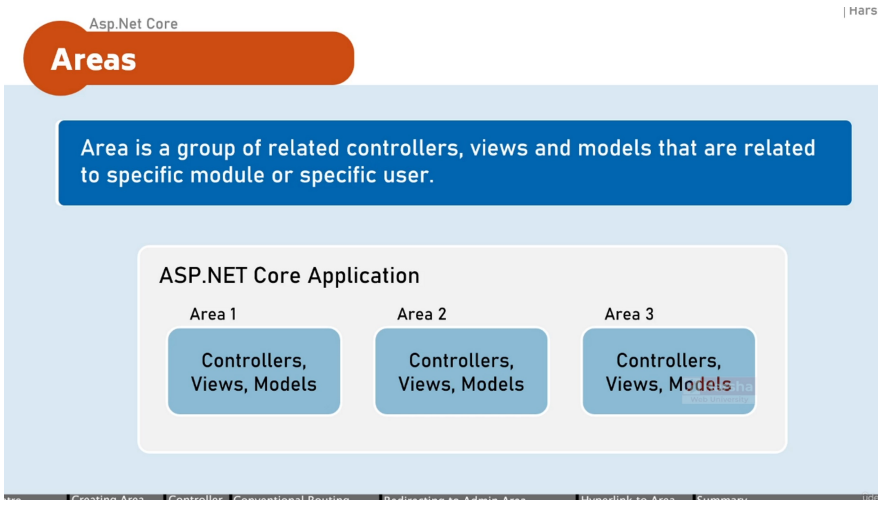
Use Cases

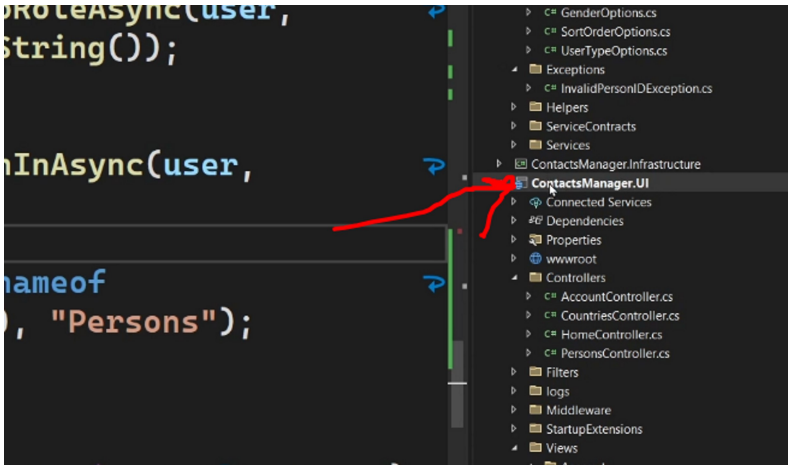
- **Modules**: Group controllers, views, and models by application modules (e.g., Front Office, Accounting).
- **User Roles**: Organize components by user roles (e.g., Admin, Customer, Employee).
- **Example**: In an organization's web application, you might create:
  - A **Front Office Area** for front office employees.
  - An **Accounting Area** for accounting staff.
  - An **Admin Area** for administrators.

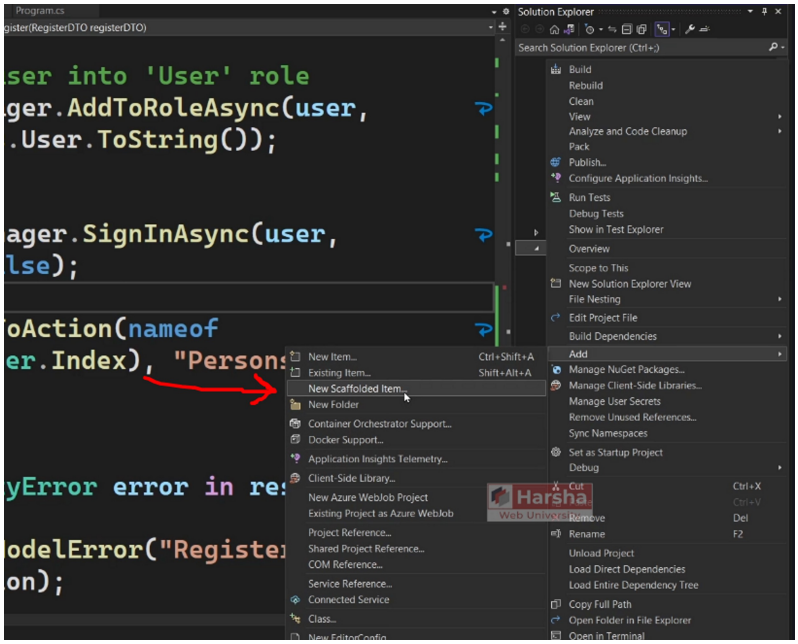
In this guide, we'll create an **Admin Area** to group all controllers, views, and models related to the **Administrator** role in the application.

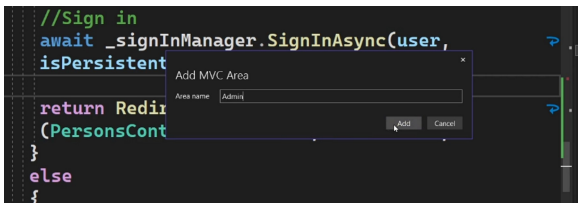
Why Use Areas?

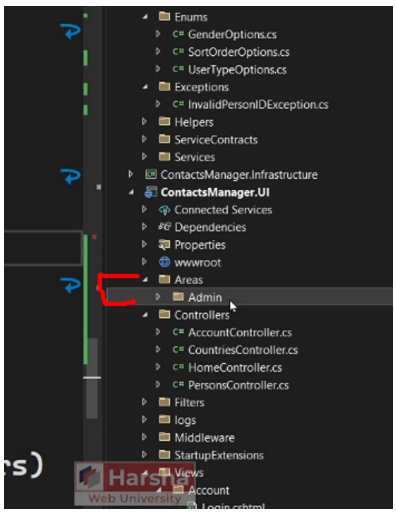
- **Organization**: Keeps related components (controllers, views, models) in separate folders for better maintainability.
- **Isolation**: Prevents naming conflicts and separates logic for different modules or roles.
- **Scalability**: Simplifies managing large applications with multiple modules or user types.

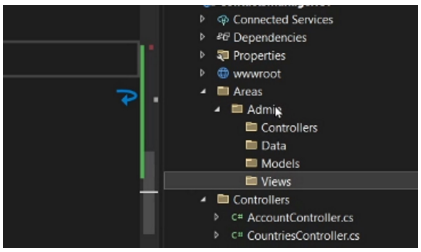


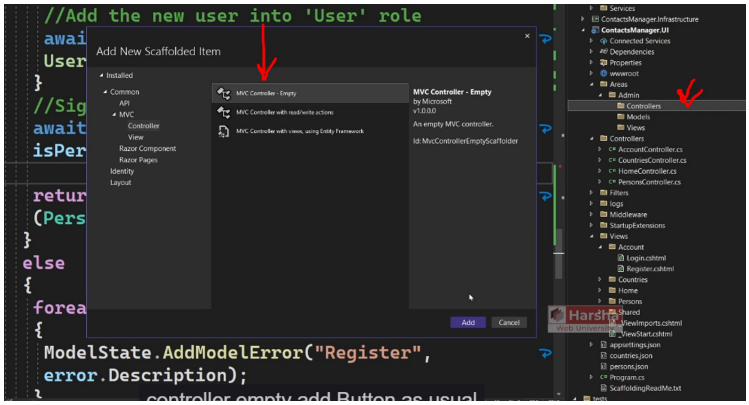



Program.cs
gister(RegisterDTO registerDTO)
ser into 'User' role
ger.AddToRoleAsync(user,
.User.ToString());
ager.SignInAsync(user,
lse);
oAction(nameof
er.Index), "Person
yError error in re
odelError("Registe
on);
Solution Explorer
Search Solution Explorer (Ctrl+;)
Build
Rebuild
Clean
View
Analyze and Code Cleanup
Pack
Publish...
Configure Application Insights...
Run Tests
Debug Tests
Show in Test Explorer
Overview
Scope to This
New Solution Explorer View
File Nesting
Edit Project File
Build Dependencies
Add
Manage NuGet Packages...
Manage Client-Side Libraries...
Manage User Secrets
Remove Unused References...
Sync Namespaces
Set as Startup Project
Debug
Cut Ctrl+X
Remove Del
Rename F2
Unload Project
Load Direct Dependencies
Load Entire Dependency Tree
Copy Full Path
Open Folder in File Explorer
Open in Terminal
New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Scaffolded Item...
New Folder
Container Orchestrator Support...
Docker Support...
Application Insights Telemetry...
Client-Side Library...
New Azure WebJob Project
Existing Project as Azure WebJob
Project Reference...
Shared Project Reference...
COM Reference...
Service Reference...
Connected Service
Class...
Harsha
Web University

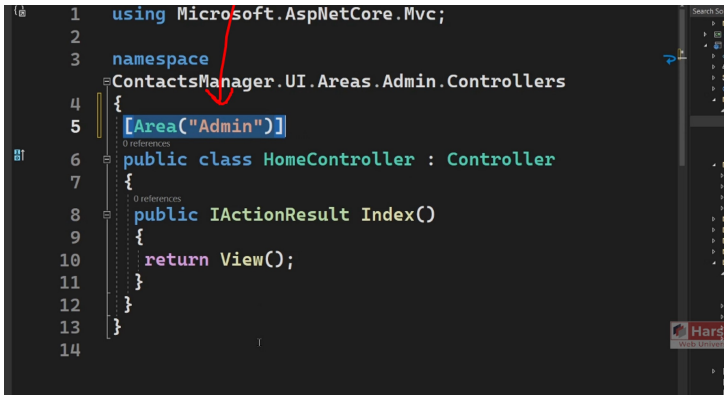

//Sign in
await _signInManager.SignInAsync(user,
isPersistent
return Redir
(PersonsCont
}
else
{
Add MVC Area
Area name
Admin
Add
Cancel

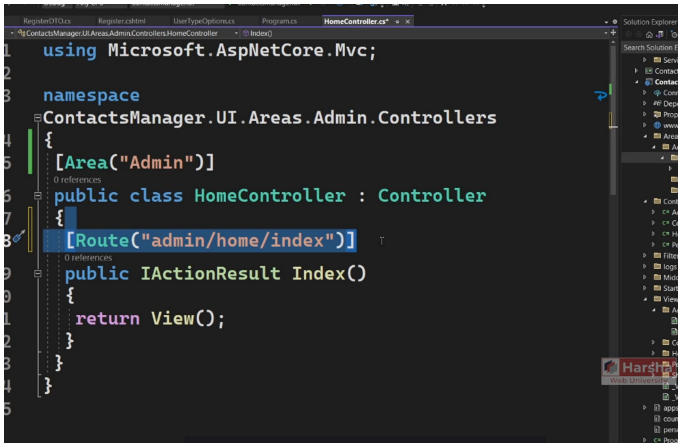

Enums
GenderOptions.cs
SortOrderOptions.cs
UserTypeOptions.cs
Exceptions
InvalidPersonIDException.cs
Helpers
ServiceContracts
Services
ContactsManager.Infrastructure
ContactsManager.UI
Connected Services
Dependencies
Properties
wwwroot
Areas
Admin
Controllers
AccountController.cs
CountriesController.cs
HomeController.cs
PersonsController.cs
Filters
logs
Middleware
StartupExtensions
Views
Account
Harsha
Web University

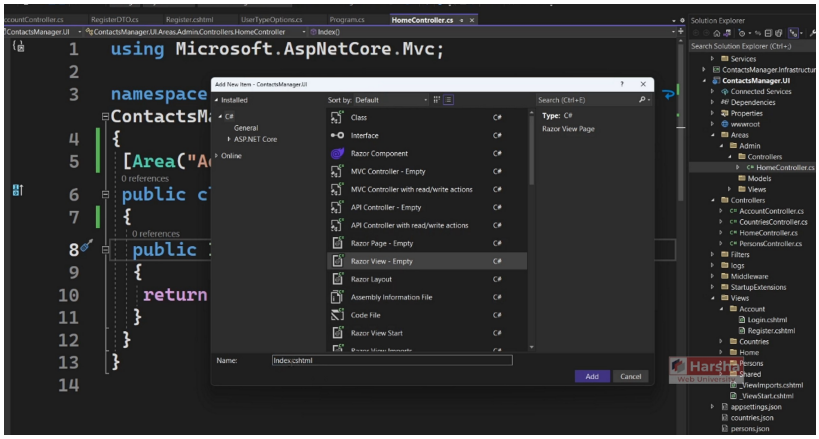

Connected Services
Dependencies
Properties
wwwroot
Areas
Admin
Controllers
Data
Models
Views
Controllers
AccountController.cs
CountriesController.cs

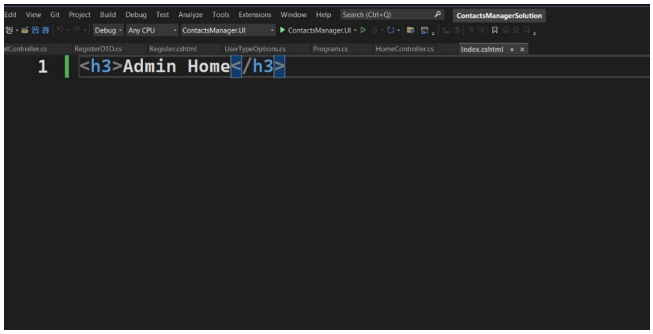


In order to represent this 'HomeController' is belong 'Admin' area.



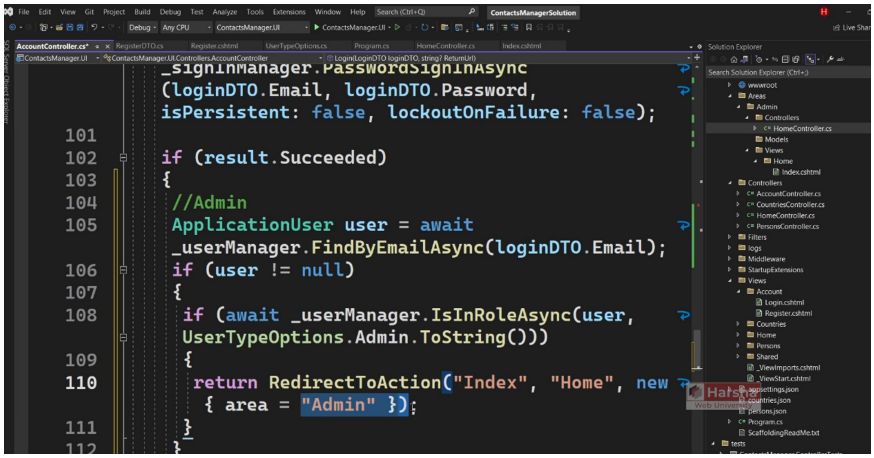


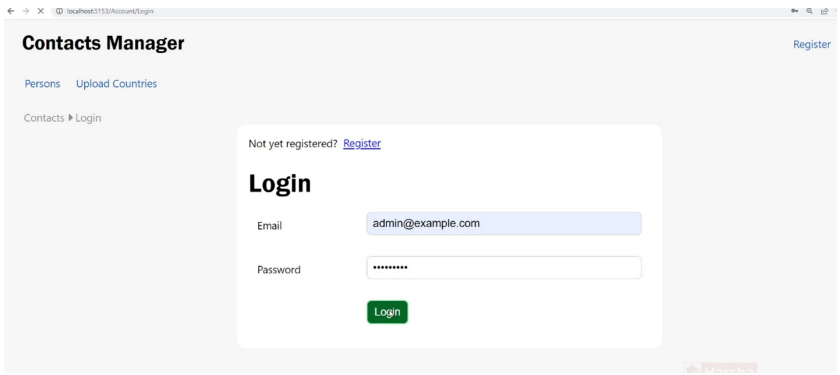


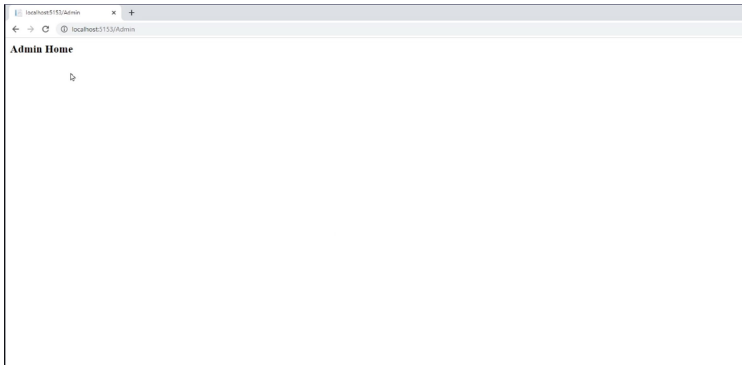


The next part is that, we have to redirect this 'HomeController' in the admin area if the admin is logged in.

Go to AccountController.cs file



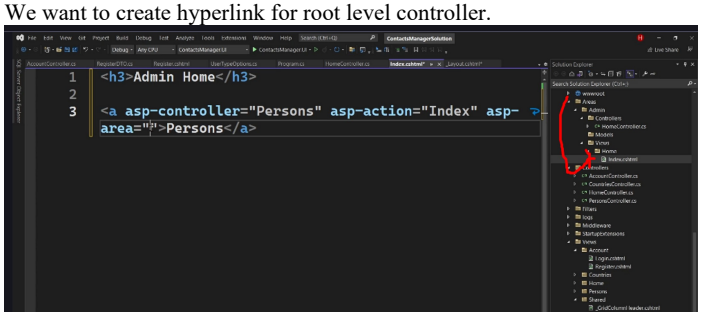


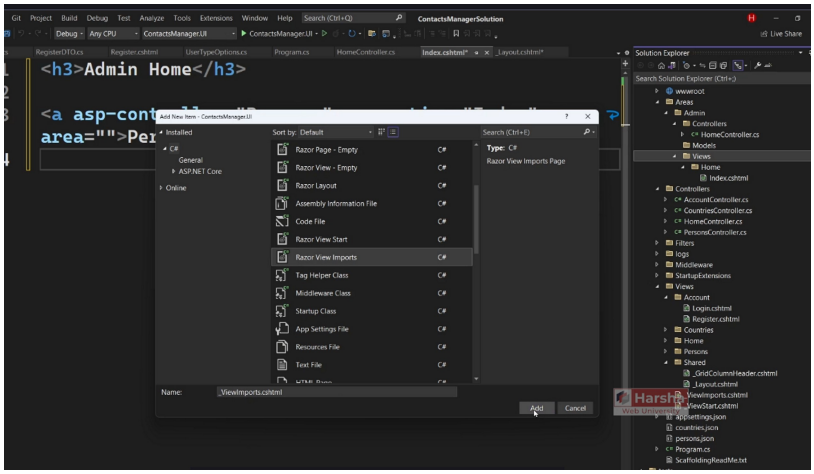


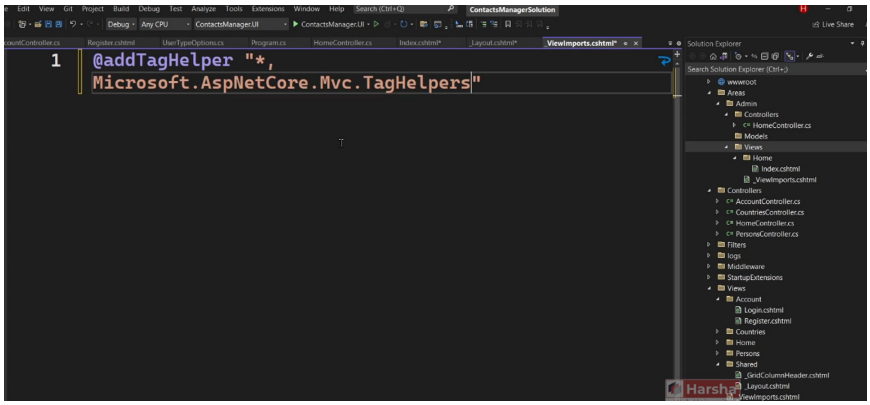
Layout.cshtml file

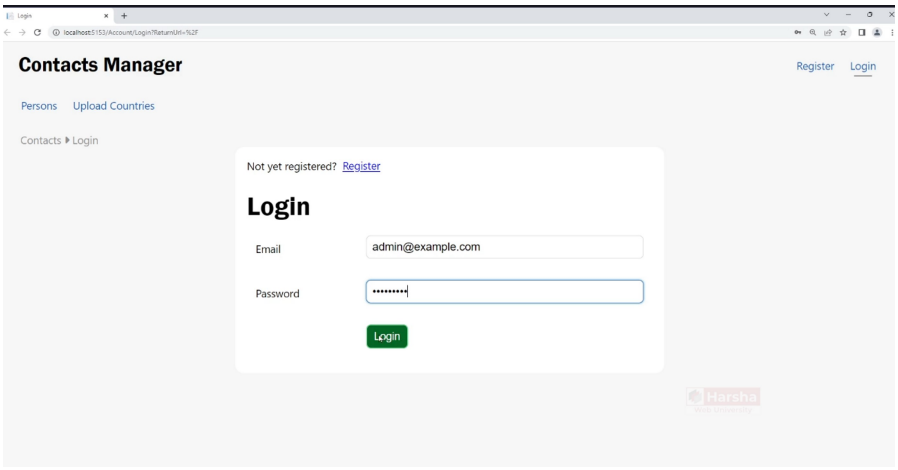
```
@User.Identity?.Name
</li>

<li>
<a asp-controller="Home" asp-
action="Index" asp-area="Admin">Admin</a>
</li>
```

We want to create hyperlink for root level controller.

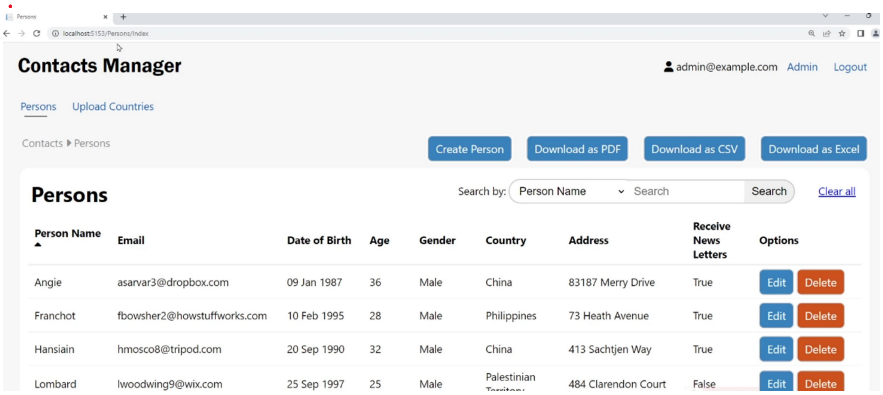






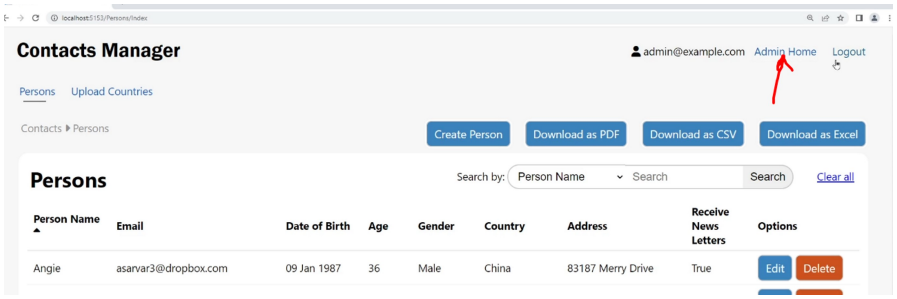


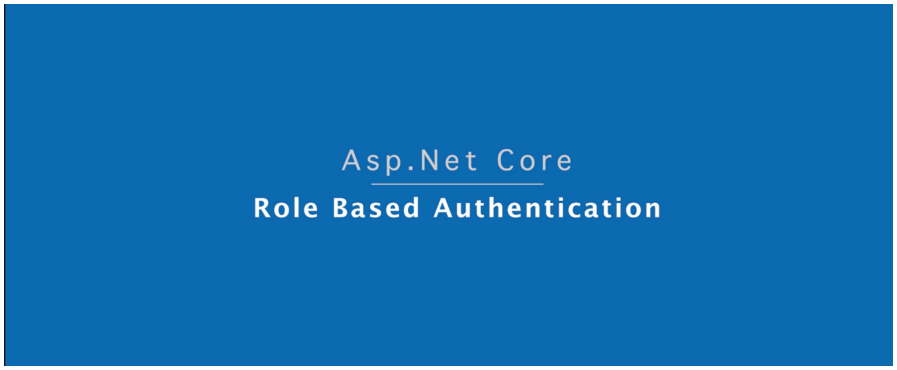
We navigates back to root level controller.



Layout.cshtml

Let's try to log in as a normal user, in that case this link should not appear.



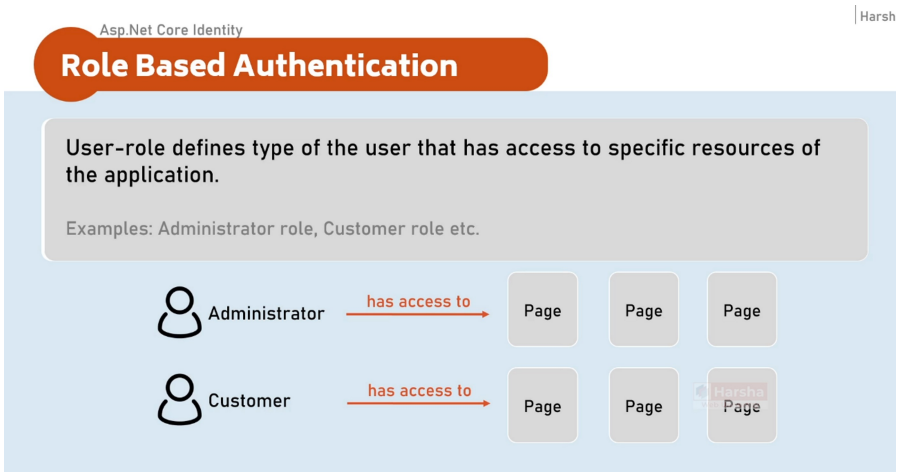


Implementing Role-Based Authentication in ASP.NET Core
Overview

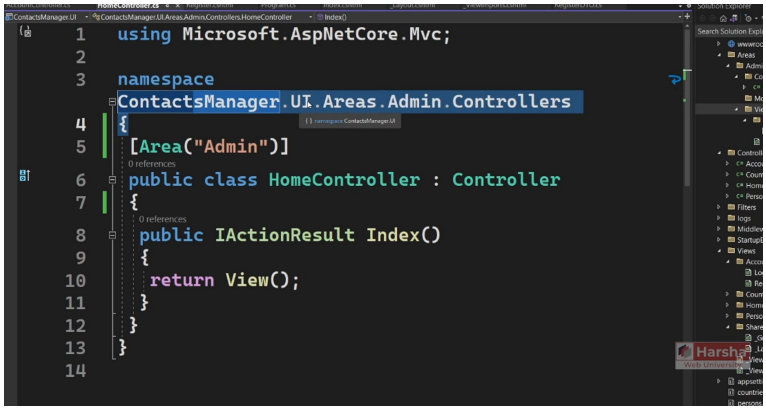
Role-based authentication restricts access to specific pages or endpoints based on the user's role (e.g., Administrator, Customer, Employee). In ASP.NET Core, this is achieved using the **Authorize** filter, a predefined authorization filter that limits access to endpoints for users with specific roles.

This guide explains how to use the [Authorize]attribute to implement role-based authentication.
Concept

- **Role-Based Authentication**: Grants access to a set of pages or endpoints only to users with a particular role.
- **Authorize Filter**: A built-in ASP.NET Core filter that enforces role-based access control by checking the user's role in their identity cookie.
- **Use Case**: For example, only Administrators can access admin dashboards, only Customers can access policy purchase pages, and only Employees can access internal tools.



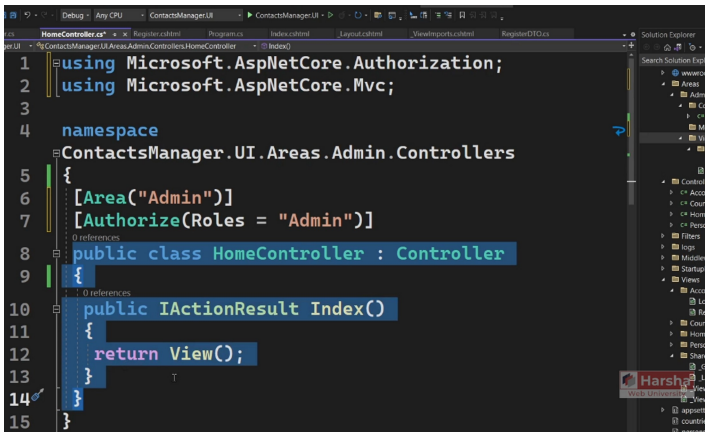
For example, this is your HomeController of 'Admin' area.



```
using Microsoft.AspNetCore.Mvc;

namespace
ContactsManager.UI.Areas.Admin.Controllers
{
 [Area("Admin")]
 0 references
 public class HomeController : Controller
 {
  0 references
  public IActionResult Index()
  {
   return View();
  }
 }
}
```

So I would like to allow this Home controller to be accessible only for administrator users, not for remaining users. To do so, this area attribute is not enough. You have to write the Authorize attribute in addition, that comes from Microsoft.AspNetCore.Authorization.

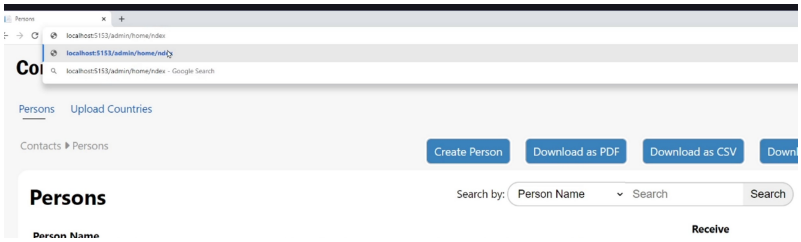
Since we have applied at the controller level. This is applicable for all the action method.

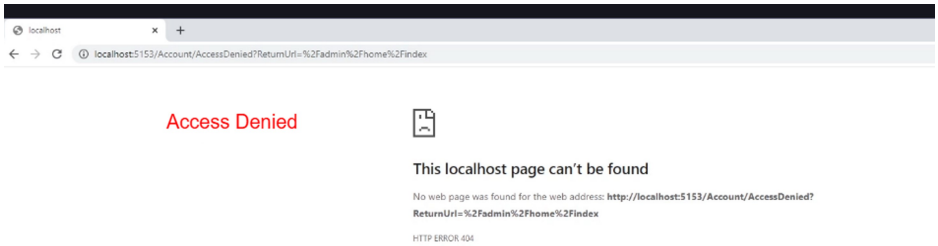


```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace
ContactsManager.UI.Areas.Admin.Controllers
{
 [Area("Admin")]
 [Authorize(Roles = "Admin")]
 0 references
 public class HomeController : Controller
 {
  0 references
  public IActionResult Index()
  {
   return View();
  }
 }
}
```

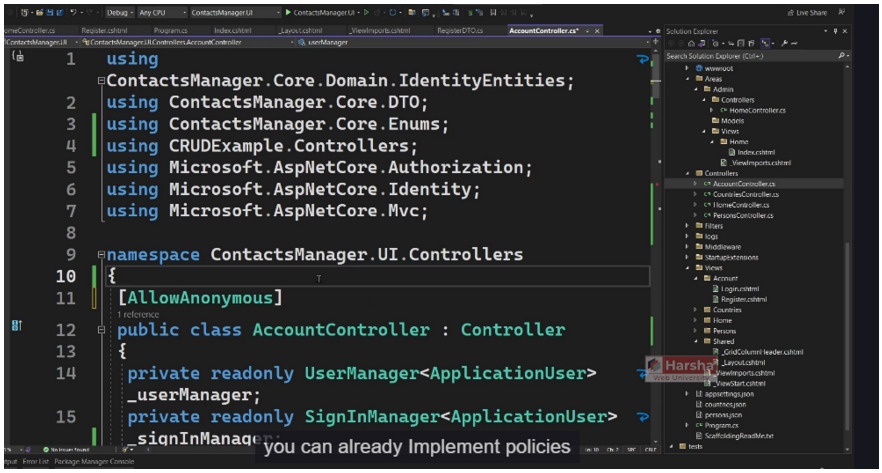
Now if someone tries to access this Admin home page without 'Admin' role, it automatically redirects to Login Page.

For example, I have logged in as a non-administrator user and trying to access admin home page.





Sometimes role-based authentication scenarios will be much more complex. For example, you would like to allow the user to access the particular action method or controller only if the user is not logged in. Otherwise, you would like to restrict access to the particular action method or controller for all the users in the particular role but except for single user or single set of users. So for this complex or rare kind of situations or scenarios, you have two options actually. You can either create a custom authorization filter to implement that particular functionality, or you can also achieve the similar kind of functionality by using policies and apply the same by using the existing authorized attribute. Let me show how to use the authorized policies. Let's say for example.



Let's say for example, assume our scenario is, the Account controller action methods should be accessible only if the user is not logged in. So if the user is already logged in, either as administrator user or normal user, then the Account slash Login or Register or any other action methods should not be accessible. In that case, for the Account controller, you can already implement policies.

Open 'ConfigureServiceExtension.cs' file

When you make a request to this 'Register' get action method, automatically it will execute this particular policy.

```
{
 _userManager = userManager;
 _signInManager = signInManager;
 _roleManager = roleManager;
}
```

AccountControler

```
[HttpGet]
[Authorize("NotAuthorized")]
0 references
public IActionResult Register()...

[HttpPost]
0 references
public async Task<IActionResult> Register
(RegisterDTO registerDTO)
```

So, in this policy you can define to allow or not allow access to the particular action method.

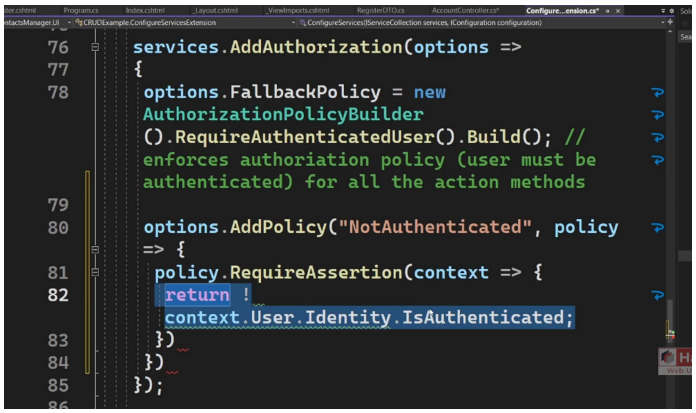
```
services.AddAuthorization(options =>
{
 options.FallbackPolicy = new
 AuthorizationPolicyBuilder
 ().RequireAuthenticatedUser().Build(); //
 enforces authoriation policy (user must be
 authenticated) for all the action methods

 options.AddPolicy("NotAuthenticated", policy
 => {

 })
});

services.ConfigureApplicationCookie(options =>
{
 options.LoginPath = "/Account/Login";
```

If true, that means user has access to particular action method.

If false, that means user has not access to the particular resource.

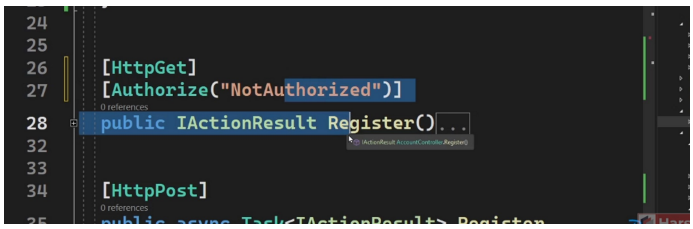
```
services.AddAuthorization(options =>
{
 options.FallbackPolicy = new
 AuthorizationPolicyBuilder
 ().RequireAuthenticatedUser().Build(); //
 enforces authoriation policy (user must be
 authenticated) for all the action methods

 options.AddPolicy("NotAuthenticated", policy
 => {
  policy.RequireAssertion(context => {
   return false;
  })
 })
});

services.ConfigureApplicationCookie(options =>
```

If the user is already logged in,

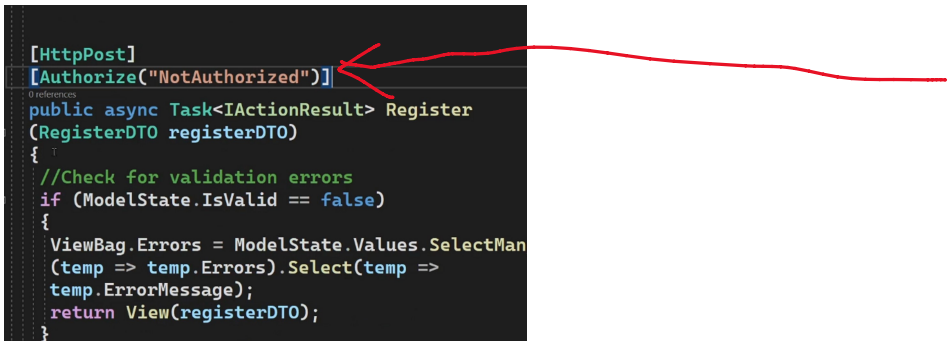


```
services.AddAuthorization(options =>
{
 options.FallbackPolicy = new
 AuthorizationPolicyBuilder
 ().RequireAuthenticatedUser().Build(); //
 enforces authoriation policy (user must be
 authenticated) for all the action methods

 options.AddPolicy("NotAuthenticated", policy
 => {
  policy.RequireAssertion(context => {
   return !
   context.User.Identity.IsAuthenticated;
  })
 })
});
```

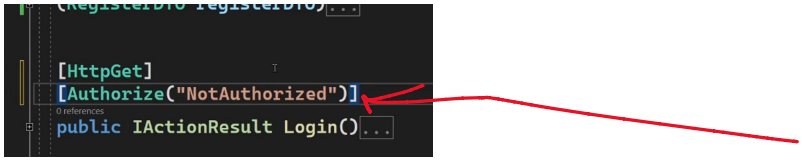
This method will not be accessible.



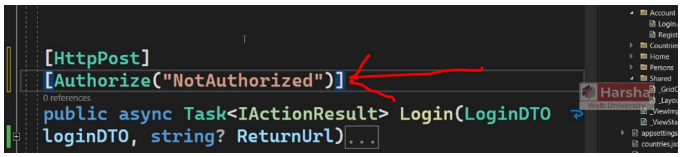
```
[HttpGet]
[Authorize("NotAuthorized")]
0 references
public IActionResult Register()...

[HttpPost]
0 references
```

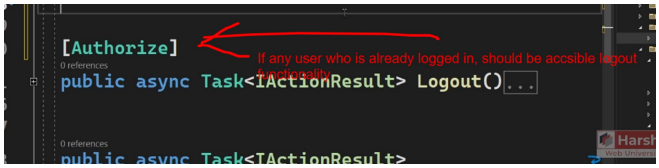


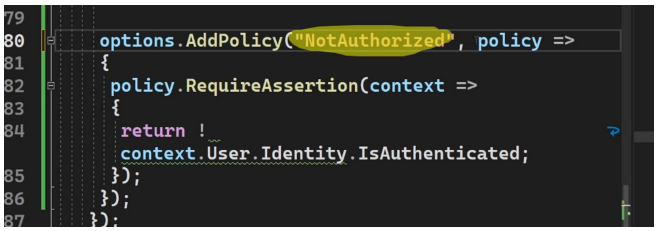
```
[HttpPost]
[Authorize("NotAuthorized")]
0 references
public async Task<IActionResult> Register
(RegisterDTO registerDTO)
{
 //Check for validation errors
 if (ModelState.IsValid == false)
 {
  ViewBag.Errors = ModelState.Values.SelectMan
  (temp => temp.Errors).Select(temp =>
  temp.ErrorMessage);
  return View(registerDTO);
 }
```



```
[HttpGet]
[Authorize("NotAuthorized")]
0 references
public IActionResult Login()...
```

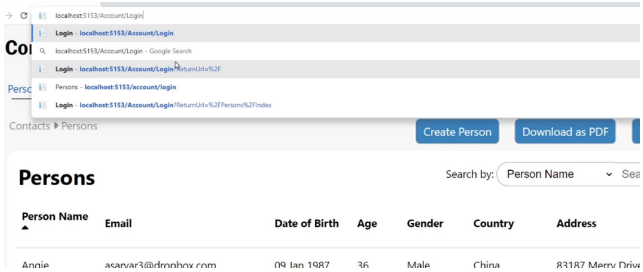


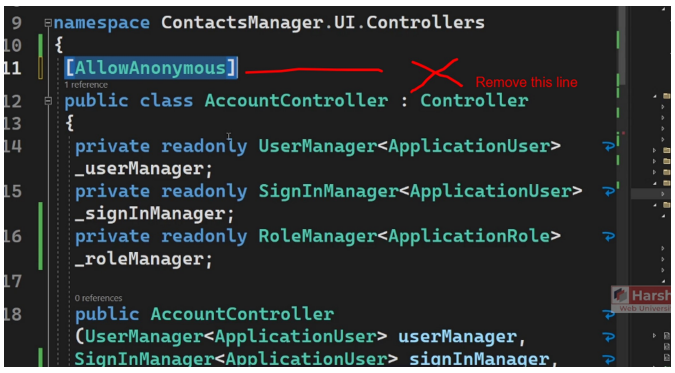


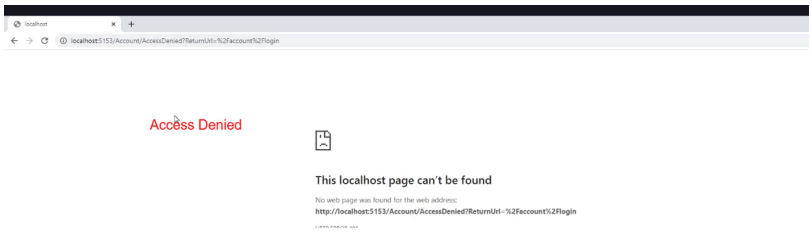


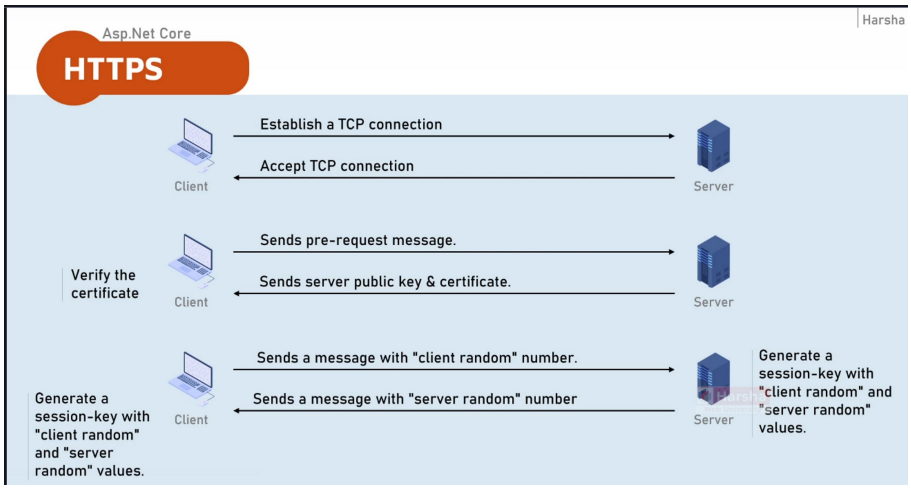
I have logged in as a normal user and trying to access login

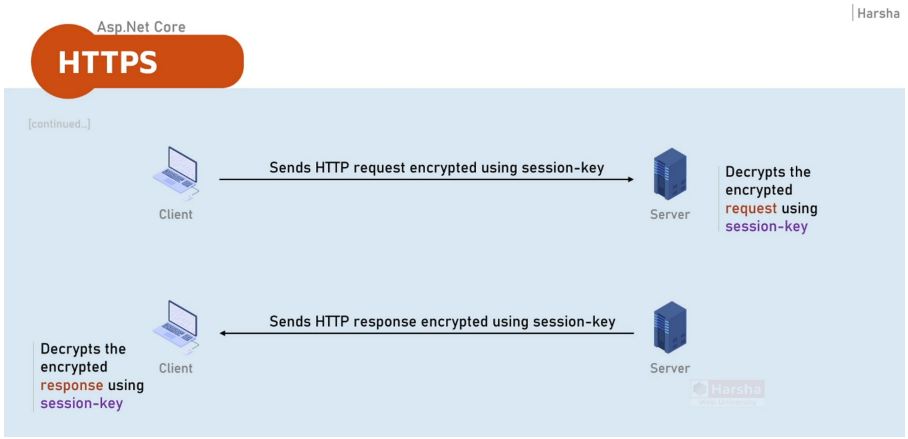
I should not access this url.











Go to program.cs file,

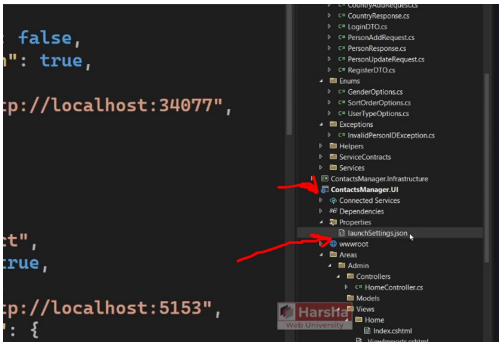
It informs this browser to use 'Https'

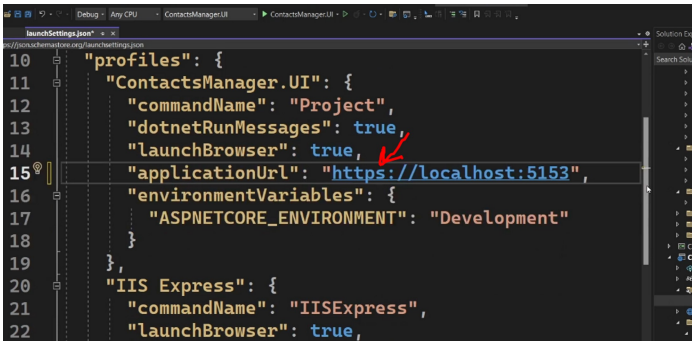
```
28  //create application pipeline
29  if (builder.Environment.IsDevelopment())
30  {
31      app.UseDeveloperExceptionPage();
32  }
33  else
34  {
35      app.UseExceptionHandler("/Error");
36      app.UseExceptionHandlingMiddleware();
37  }
38
39  app.UseHttpsRedirection();
40
41  app.UseSerilogRequestLogging();
42
43
```

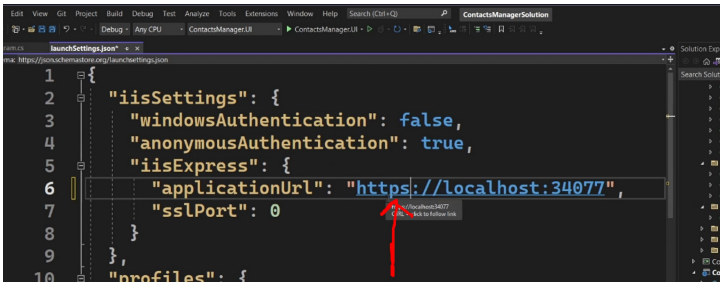
It forces the browser to enable Https for all the requests and responses. It is part of the request pipeline so that it adds that information as a part of the response header and based on that client understands that it has to enable https first before sending any actual request.

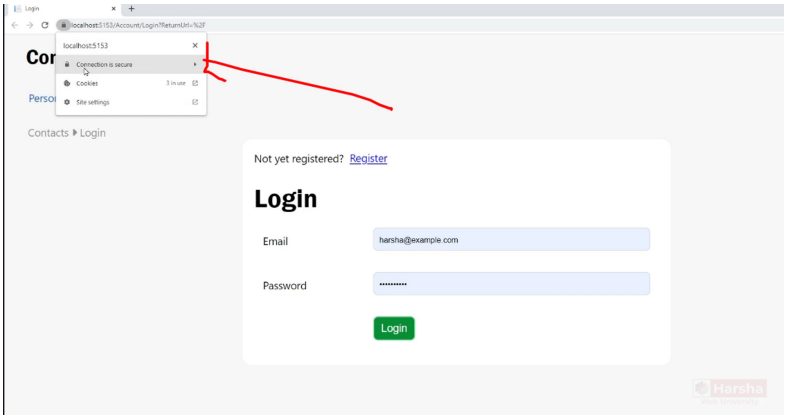
```
32  }
33  else
34  {
35      app.UseExceptionHandler("/Error");
36      app.UseExceptionHandlingMiddleware();
37  }
38
39  app.UseHsts();
40  app.UseHttpsRedirection();
41
42  app.UseSerilogRequestLogging();
43
```

Now go to 'launchSettings.json' file.

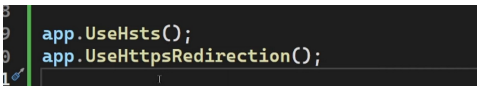








These line should be in order.



Understanding Cross-Site Request Forgery (CSRF) Attacks
Cross-site request forgery (CSRF) is a type of attack where a malicious request is sent to a server, and the server executes it as if it were a legitimate request. For example, there is a legitimate website called original.com. An attacker creates a fake website, attacker.com, and makes a request to original.comfrom that site. The endpoint of original.comexecutes that particular request, assuming it is a valid or genuine request. But actually, it is a malicious request that either steals data or performs unwanted operations on the databases.
This topic is divided into two parts. In the first part, that is, in this lecture, we will try to understand how this particular forgery request is being made and what is the actual process behind it. In the next part, that is, in the next lecture, we will try to prevent CSRF by implementing an anti-forgery token.

Understanding the Mechanism of a CSRF Attack
Let us try to understand how this Cross-Site Request Forgery (CSRF) works. That means, let us try to understand how the attacker sends a forged request to the server. For example, there is our genuine website that is bank.com, and we are the developers of bank.com. The client or the browser first logs into bank.com. So, as a result, bank.comgenerates an authentication token and sends the same as a cookie, so the client stores that particular authentication token as a cookie. So, generally, we call them an authentication token or authentication cookie. So, this process happens in general.

But here, the point is the attacker wants to steal some information or wants to perform some unwanted operation on the database of bank.com. For example, the developer of attacker.com wants to steal some money from bank.com, I mean from one of the customers of bank.com, that is the user who opens this particular browser. So, what this attacker.com will try to do is that he or she tries to create an email or website that looks like a genuine website, and the attacker includes a link. So, if the user clicks on that particular link, it automatically sends a POST request to the bank.com server.

For example, it creates a form, and the action attribute of the form tag contains the address targeting the bank.comserver. For example, this is the path /transfer. Assume that attacker.comsends an email to the user saying that you need to authenticate yourself to receive some money. So, the user may feel that he receives money, and the user tries to click on the particular link. Then, in that particular link, in a web page, he creates a form like this and automatically submits this form that makes a request to bank.comas mentioned here.
So, in the same browser, the user has opened two tabs: in one tab, he opened bank.comand already authenticated, so he already got the authentication cookie, and in the same browser, in another tab, the user opens the link provided by attacker.com. So, in the same browser, both bank.comand attacker.comhave been opened. And now, attacker.comis trying to send a request to bank.comthrough this kind of form because the user clicks on the link provided by the attacker.

But keep this scenario aside; one of the rules for the browser is that the browser should automatically submit the cookie as a part of the request without writing any additional code for all the subsequent requests. So, if you make any kind of request to bank.com, it should and will include the authentication token as a cookie in all the subsequent requests. As long as that cookie exists in the browser memory, that cookie will be and should be submitted to the server.

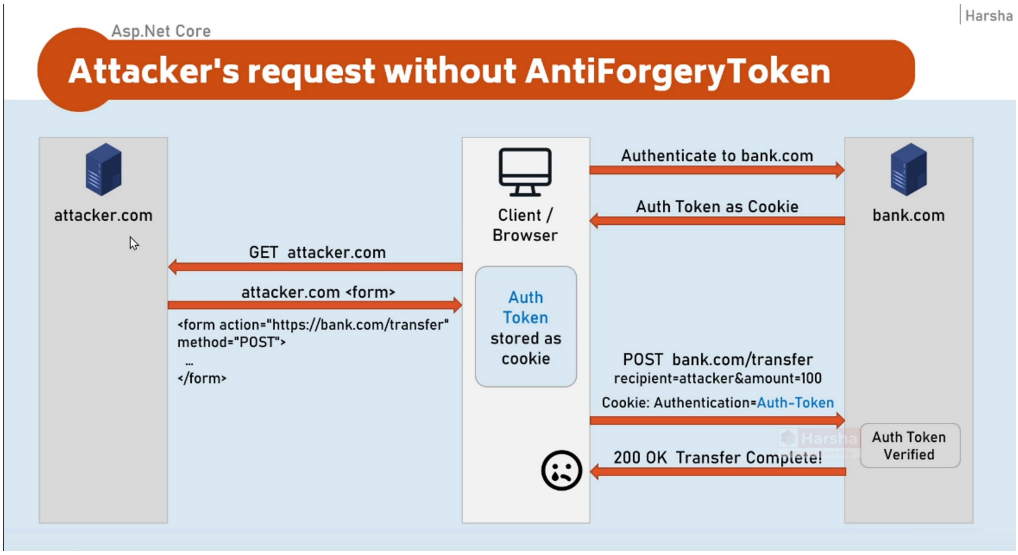
So, by following that rule, while making a request through this form to bank.com, the authentication cookie also will be submitted to the server. So, along with the POST request to bank.com/transfer, the authentication cookie automatically gets added to that particular request. And bank.comverifies that particular authentication token from the request, and it sounds valid. So, the endpoint at bank.comassumes that the request is a legit request and executes the same normally.

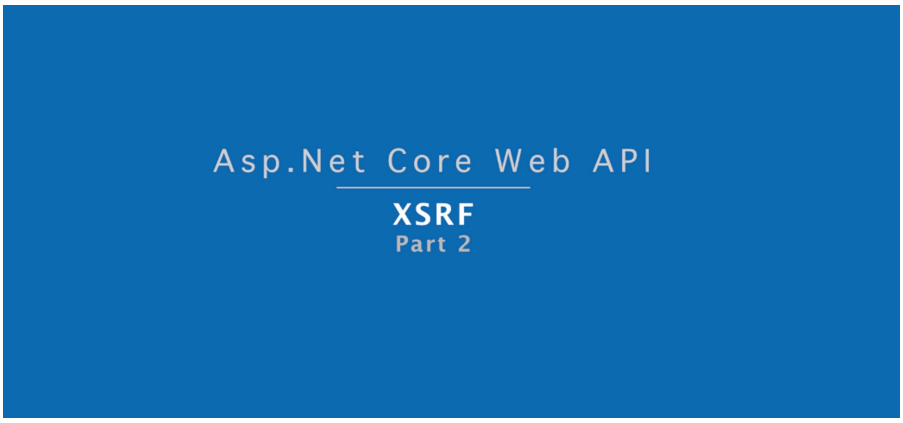
So, as per the code of that particular endpoint at bank.com, money gets transferred to the attacker because he gave the inputs like that, for example, recipient equal to attacker and some amount that he wants. So, overall, money has been transferred from the user's account into the attacker.combank account. So, here, the victim is the user, and the culprit is the attacker.

The reason for the whole problem is the bank.comendpoint is not in a position to verify the source of this particular cookie. As soon as the authentication token is included as a part of the cookie, it assumes that it is a legit request. But it may not be sometimes like this. See, if the user tries to click on any one of the links in bank.com, in the legit website, then this authentication process seems to be okay. But the same process happens when attacker.comsends a request like this.

So, the server endpoint here is unable to differentiate between the legit request and a fake request or forged request. This is the problem. This is how attackers take advantage of loopholes in server implementation. So, how do you overcome this problem in particular to ASP.NET Core? The answer is by using an anti-forgery token.

So, how do you enable this anti-forgery token, and how does it work exactly and internally? We will try to understand this in the next lecture.

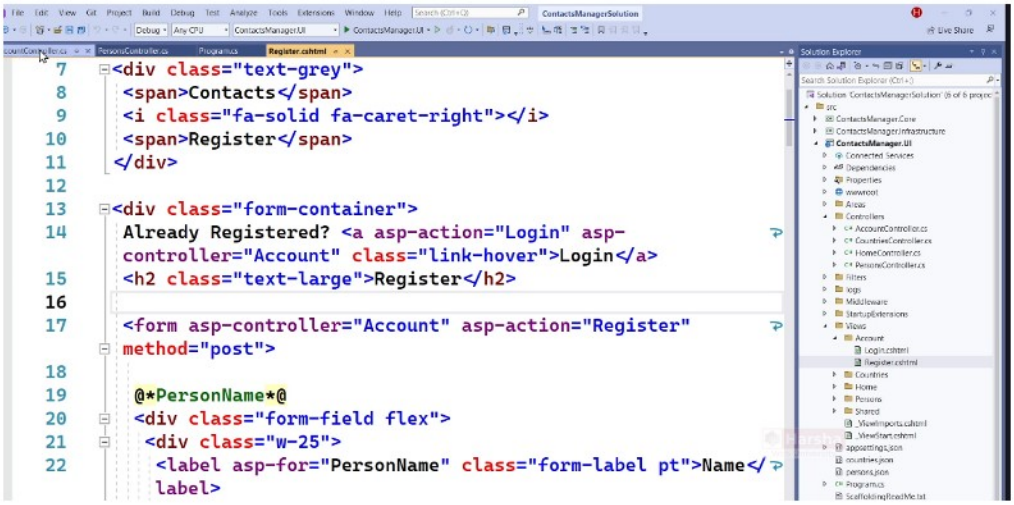




In the last lecture, we have understood how the XSRF requests are being sent and what is the problem there. The main problem is the bank.comendpoint cannot differentiate between the requests, either it is a legit request that is made from a page of bank.comor from a page of attacker.com. So, in order to identify the differentiation, let us try to implement the security against XSRF by using an anti-forgery token. It is strongly recommended by Microsoft to implement an anti-forgery token in every page, I mean in every View and controller.

I mean in every View and controller, but it is as simple as go to The View, for example let's focus on account controller register View, and in every view make sure that you are using the form tag helpers that is asp-controller and asp-action for every form, and also in the corresponding HTTP POST action method.

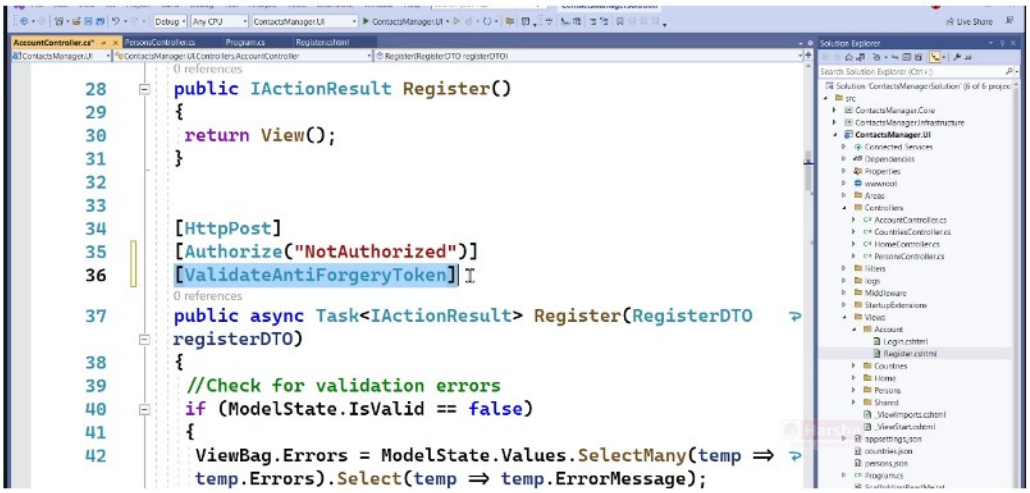
I mean in every View and controller, but it is as simple as go to The View, for example let's focus on account controller register View, and in every view make sure that you are using the form tag helpers that is asp-controller and asp-action for every form, and also in the corresponding HTTP POST action method.



```
<div class="text-grey">
 <span>Contacts</span>
 <i class="fa-solid fa-caret-right"></i>
 <span>Register</span>
</div>

<div class="form-container">
 Already Registered? <a asp-action="Login" asp-
 controller="Account" class="link-hover">Login</a>
 <h2 class="text-large">Register</h2>

 <form asp-controller="Account" asp-action="Register"
 method="post">

  @*PersonName*@
  <div class="form-field flex">
   <div class="w-25">
    <label asp-for="PersonName" class="form-label pt">Name</
    label>
```

For example, HTTP POST register.
You have to manually add an attribute
called ValidateAntiForgeryToken.
And that's it.

It is as simple as that.
Now our register HTTP POST endpoint is
secured from XSRF requests,
so it automatically prevents against XSRF.



```
0 references
public IActionResult Register()
{
 return View();
}


[HttpPost]
[Authorize("NotAuthorized")]
[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> Register(RegisterDTO
registerDTO)
{
 //Check for validation errors
 if (ModelState.IsValid == false)
 {
  ViewBag.Errors = ModelState.Values.SelectMany(temp =>
  temp.Errors).Select(temp => temp.ErrorMessage);
```

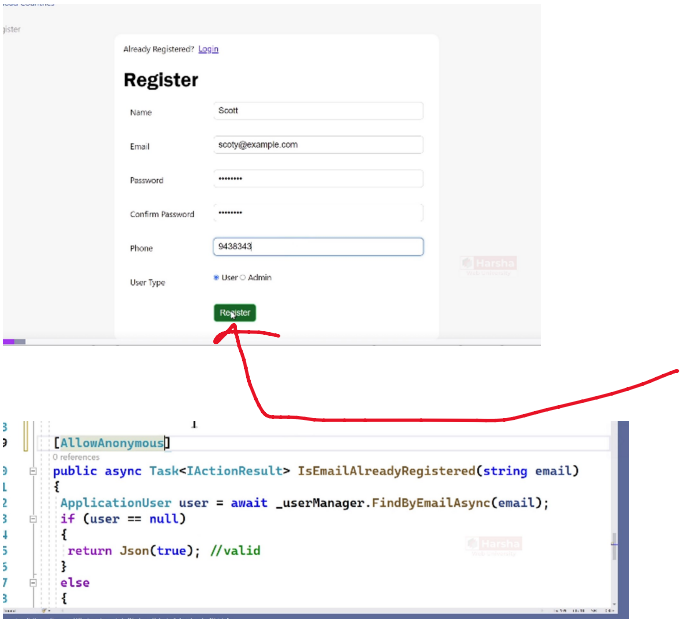
So step 1 is, In the form you must use the form tag helpers.

```
 </div>

<div class="form-container">
 Already Registered? <a asp-action="Login" asp-
 controller="Account" class="link-hover">Login</a>
 <h2 class="text-large">Register</h2>

 <form asp-controller="Account" asp-action="Register"
 method="post">

  @*PersonName*@
  <div class="form-field flex">
   <div class="w-25">
    <label asp-for="PersonName" class="form-label pt">Name</
```

2. In the Controller Action method, you have to add 'validateAntiForgeryToken'

```
31      }
32
33
34      [HttpPost]
35      [Authorize("NotAuthorized")]
36      [ValidateAntiForgeryToken]
        0 references
37      public async Task<IActionResult> Register(RegisterDTO
        registerDTO)
38      {
39          //Check for validation errors
40          if (ModelState.IsValid == false)
41          {
42              ViewBag.Errors = ModelState.Values.SelectMany(temp =>
```

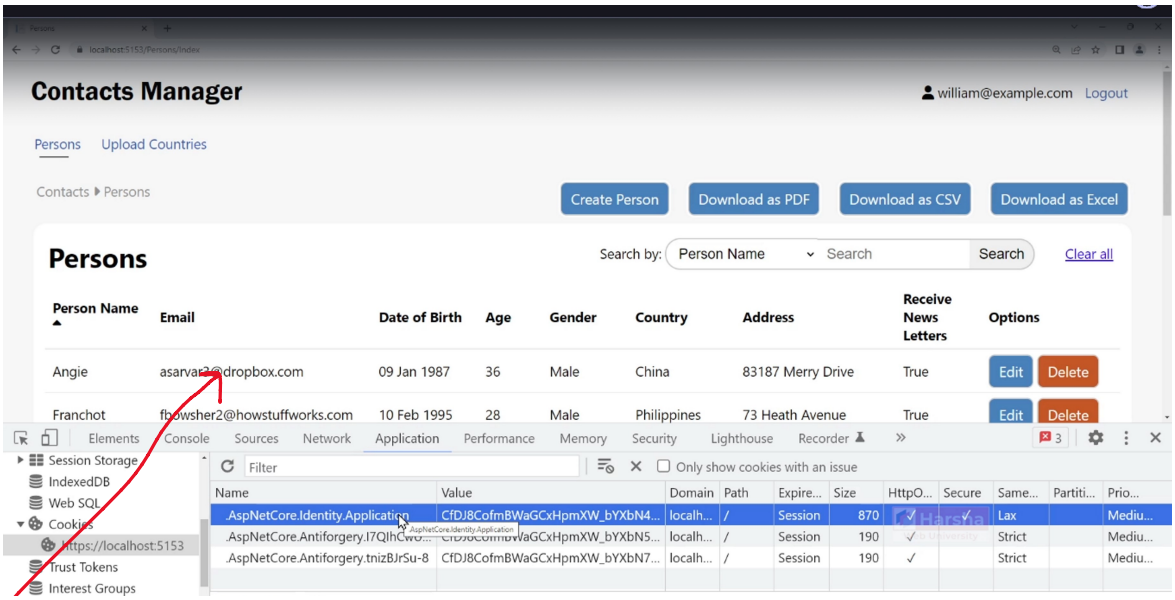
Now let's see how does this work. The application runs nomrally.
If you give proper inputs with this form

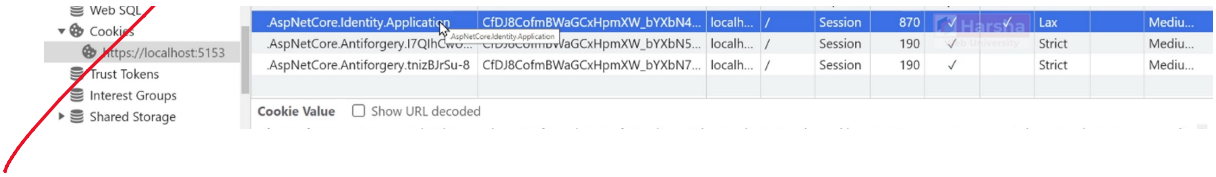



```
[AllowAnonymous]
0 references
public async Task<IActionResult> IsEmailAlreadyRegistered(string email)
{
    ApplicationUser user = await _userManager.FindByEmailAsync(email);
    if (user == null)
    {
        return Json(true); //valid
    }
    else
    {
```

The root cause of XSRF problem is this particular identity cookie will be automatically submitted as a part of the request for all the request made to our server.

For example, our doman is 'bank.com'



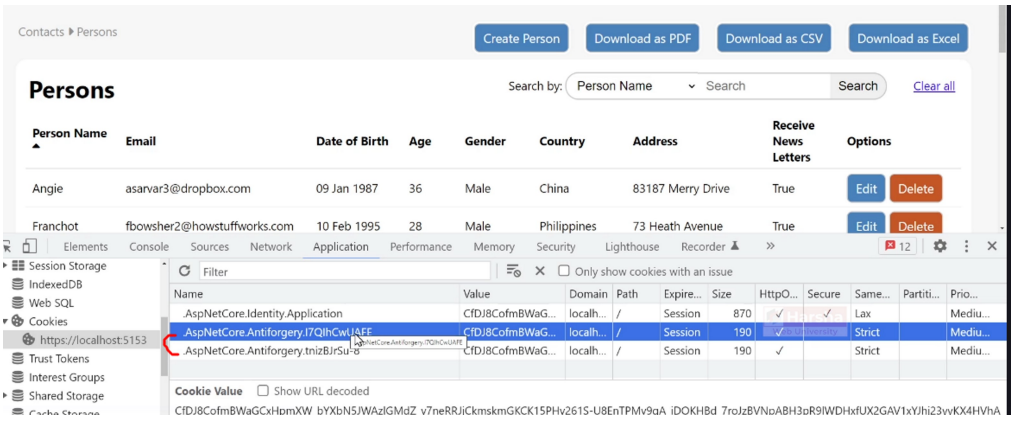


So this particular cookie gets submitted for all the request to 'bank.com'. But this problem does not occur
In case of register page in this endpoint. Because we have enabled 'AnitForgeToken' .

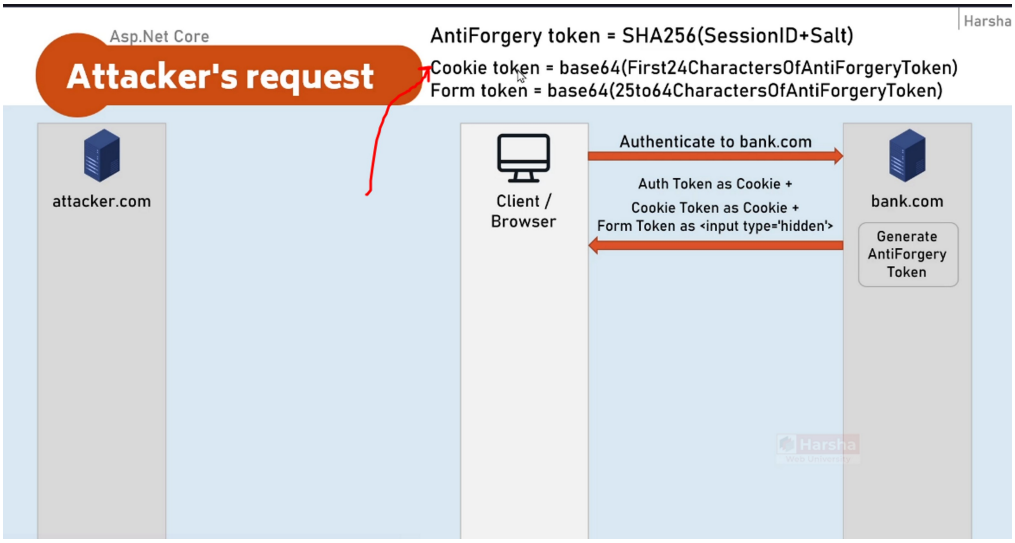
But how does it work exactly internally?
So, in this case, the anti-forgery token is automatically generated by the server.
When you enable the anti-forgery token, ASP.NET Core creates a token at runtime.
Here, the anti-forgery token is a hash value that is generated by a hashing algorithm called SHA-256 — Secure Hash Algorithm 256.
SHA-256 is a hashing algorithm that generates a hash based on the session ID and a salt value.
When the user logs in, the server automatically generates a session — that means a place to store the user details on the server side.
In the session, the user can store additional details of that particular user if required.
For every session, a unique ID is generated — that is called the session ID.
And here, the salt value is a randomly generated value, much like a random number.
So, the session ID concatenated with the salt value — both of these inputs will be hashed using the hashing algorithm called SHA-256.
As a result, you'll get a hash value — and that hash value is called the anti-forgery token.
According to the SHA-256 algorithm, the hash value contains 64 hexadecimal characters.
Out of those 64 characters, the first 24 characters of the hash value are converted into a Base64 string — that is called the **cookie token**.
The remaining characters — from the 25th to the 64th — are also converted into a Base64 string, and that is called the **form token**.
So, the same hash value (aka the anti-forgery token) is split into two parts:

- the cookie token
- and the form token.

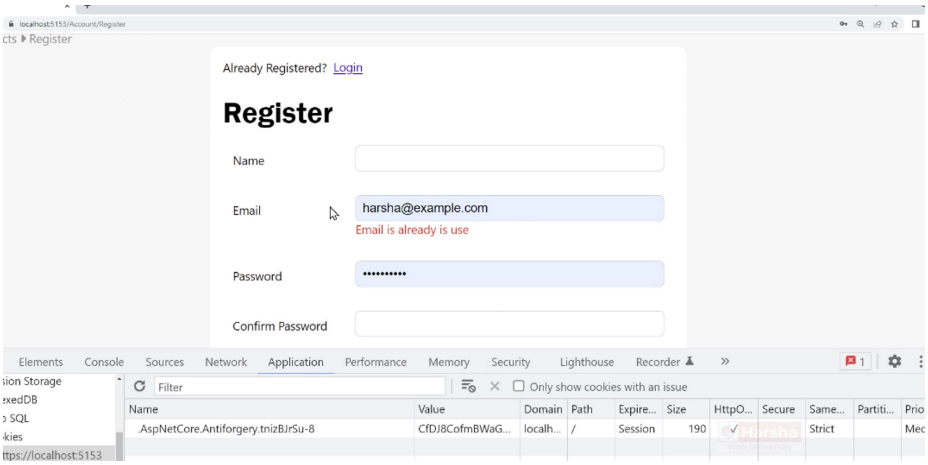
Of course, they are not the actual hash values, but Base64 strings derived from the original hash value.
The first part is converted to a Base64 string — that's the cookie token.
The next part is also converted to a Base64 string — that's the form token.
So, how are the cookie token and form token used to identify whether the request is a valid request or a forged one?
Let's reimagine a scenario of an attack.
That means, the client (or user) first navigates to bank.com and logs in.
For example, I'm the victim, and I just logged into my bank.com.
At this point, the authentication cookie is generated as usual, since I logged in with my credentials.
So, I got an **authentication token**, and at the same time, I got new tokens: the **cookie token** and the **form token**.
Here, the server sends the cookie token as a cookie,
and the form token is included as a hidden input field inside the form tag.

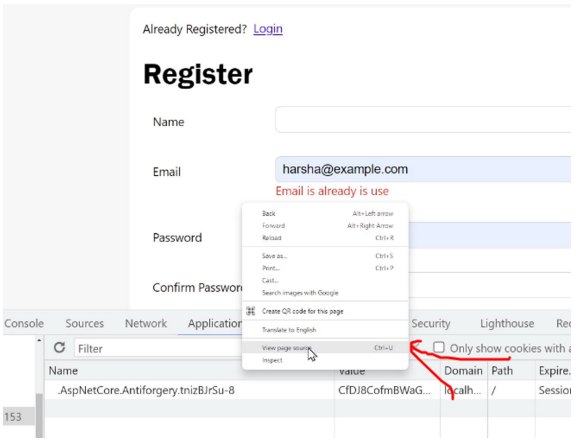


so that is the reason here you can see
anti-forged token
but we see two because one may be a
water one
here this cookie called asp.net core dot
anti forgery is your cookie token over
Here.



But where is this 'form token'? The form token gets automatically included as a part of The form. For example, navigate to the 'Register' form and click 'view page source'

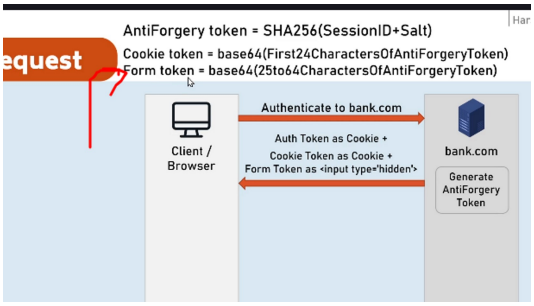




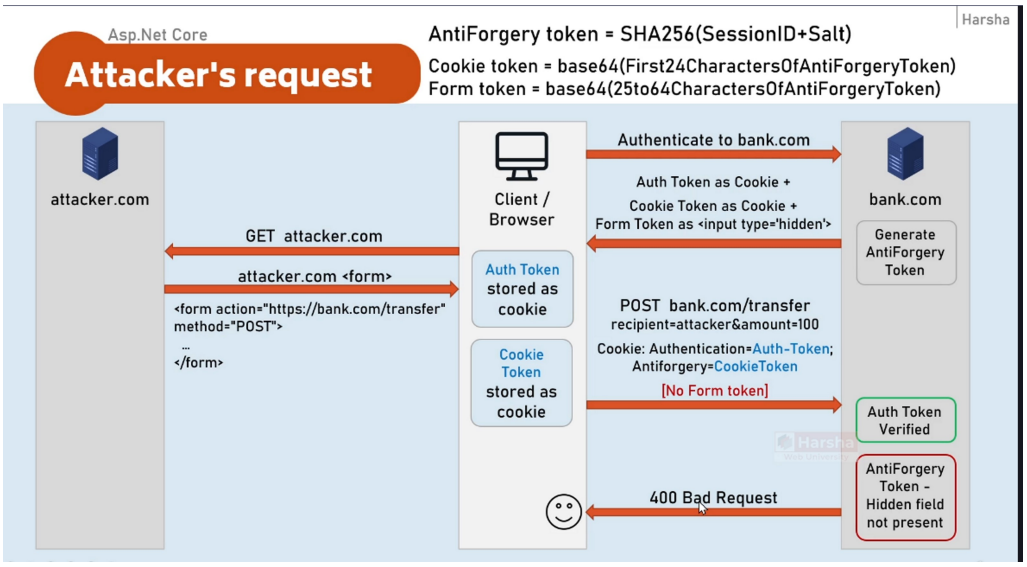
Scroll down. This is base64 string version of this form token.
This is not actual hash value but this is the hash value converted into base 64 string.

Every time you re-fresh you will see a different value over here.


135 <label for="Admin">Admin</label>
136
137
138
139
140
141
142
143 <button class="button button-green-back" type="submit">Register</button>
144
145 <li style="display:
146
147
148
149
150
151 <input name="__RequestVerificationToken" type="hidden" value="CfDJ8CofmBWaGCxHpmXW_bYX
152
153
154
155
156
157
158
159
160 <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>


AntiForgery token = SHA256(SessionID+Salt)
Cookie token = base64(First24CharactersOfAntiForgeryToken)
Form token = base64(25to64CharactersOfAntiForgeryToken)
Client / Browser
Authenticate to bank.com
Auth Token as Cookie +
Cookie Token as Cookie +
Form Token as <input type='hidden'>
bank.com
Generate AntiForgery Token


Asp.Net Core
Attacker's request
AntiForgery token = SHA256(SessionID+Salt)
Cookie token = base64(First24CharactersOfAntiForgeryToken)
Form token = base64(25to64CharactersOfAntiForgeryToken)
attacker.com
Client / Browser
Authenticate to bank.com
Auth Token as Cookie +
Cookie Token as Cookie +
Form Token as <input type='hidden'>
bank.com
Generate AntiForgery Token
Auth Token stored as cookie
Cookie Token stored as cookie
Anti forgery token (cookie token + form token) gets generated only when you have a form in your view; not always.



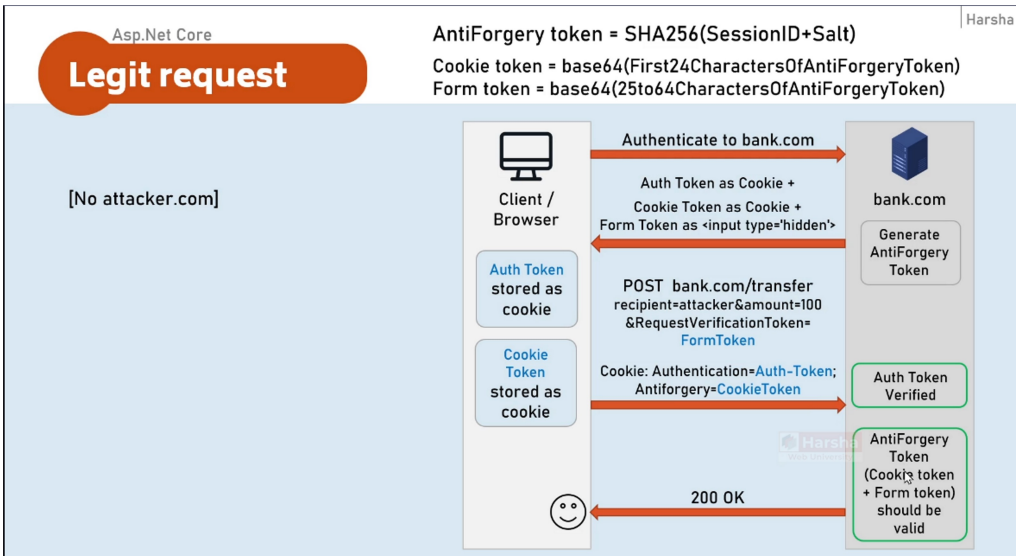
So, we can confidently say that the **cookie token** gets added as a cookie in the browser,
and another token — the **form token** — is available as an input type="hidden" within the form.
Now, in another browser tab in the same browser window, the user opens **attacker.com**
and tries to click on a link that was sent by the attacker.
For example, it could be an attractive message saying:
"Click here to receive money" or **"Click here to claim a prize amount."**
So, the victim (the logged-in user) clicks on that particular link.
And because of the sneaky code in attacker.com, it **automatically creates a form and submits it**.
The user **doesn't even click** on the submit button of this malicious form —
JavaScript on attacker.com does that behind the scenes.
So what happens?
It makes a **POST request to bank.com** — without the user's knowledge.
Now here's the catch:
Browsers **automatically include all cookies** with every request.
So when that POST request goes to **bank.com**,
it includes the **authentication token** AND the **cookie token** in the request headers.
So yeah, it looks like a legit request at first.
It even contains values like:

- recipient=attacker
- amount=some_value
  All included in the **request body**.

And the request headers?
They include something like:

- Authentication=auth_token
- Anti-Forgery=cookie_token

BUT — here's the key thing:
The **form token is missing**.
That's the form token which was embedded in the original form inside the real bank.com page
as <input type="hidden" ...> — and the attacker can't guess it or generate it.
Now, when this request hits the server, the **authentication middleware** expects **both tokens**.
Why? Because it needs both the **cookie token** and **form token** to validate the request.
That's how the anti-forgery check works.
In this case, yeah — the **cookie token** is there. But since the **form token** is missing,
the **ValidateAntiForgeryToken** filter is like:
❌ "Nope, this request is sus."
So, the server **rejects the request** and returns a 400 Bad Request.



But what **exactly** happens in case of a **legit request**?
Alright — same starting point.
The user navigates to **bank.com** and logs in. Boom — they get:

- authentication token
- cookie token
- form token

The **auth token** and **cookie token** are stored as cookies,
and the **form token** is stored as an <input type="hidden"> in the form.
⚠️ Remember: **these tokens aren't fixed**.
They are **regenerated** every time the page is refreshed or reopened.
Now, if the user clicks on the **submit** button on the **actual legit page** (not some sus attacker page),
then the browser does its thing:

- Includes the **auth cookie**
- Includes the **cookie token** as a cookie
- And also submits the **form token**, because it's already inside the form as an <input type="hidden">.

So when the request hits the server, it includes:

http
CopyEdit
Cookie: Authentication=auth_token; AntiForgery=cookie_token
Body: __RequestVerificationToken=form_token
Here, __RequestVerificationToken is that hidden input field holding the **form token**.
Now what does the server do?
The **ValidateAntiForgeryToken** filter kicks in and checks:

- Did I receive a **form token**? ✅
- Did I receive a **cookie token**? ✅

Cool. But we're not done yet.
The server now tries to **regenerate** the original anti-forgery token:

- It uses the **same session ID** and **same salt value**
- Re-runs the hashing process (SHA-256)
- Converts the hash to base64
- Splits it into a new **cookie token** and **form token**

And guess what?
It then compares those newly generated tokens with the ones that came in the request.
💡 If the tokens **match**, it's a green flag ✅
That means this request came from a legit source — using the same session and salt — so the server proceeds to execute the endpoint.
❌ If they **don't match** — it's a red flag
Either the tokens are fake, or came from a different session/salt — meaning the request is sus — so the server blocks it.
The cool part?
As a developer, **you don't have to manually do any of this**.
ASP.NET Core handles all the token creation, validation, and security magic for you behind the scenes.
So yeah — the endpoint only runs **if**:

- Cookie token ✅
- Form token ✅
- Both match the server-side regeneration ✅

Otherwise — boom, request gets rejected 🔒

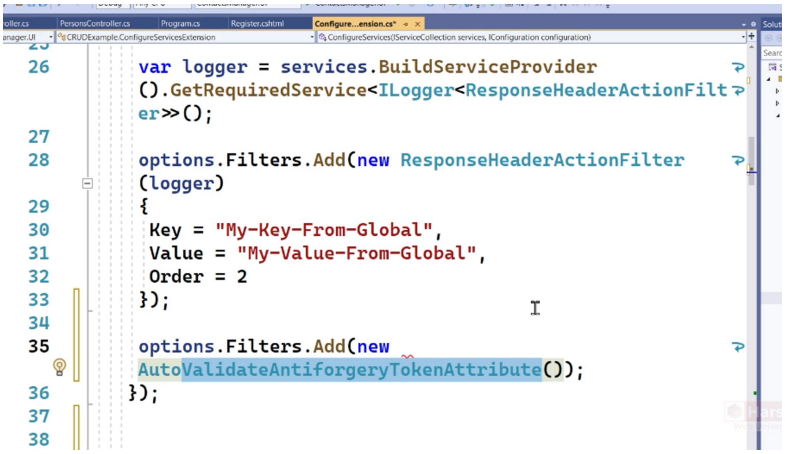
'ValidateAntiForgeryToken' is only applicable for HTTPPost, not get

```
32
33
34      [HttpPost]
35      [Authorize("NotAuthorized")]
36      [ValidateAntiForgeryToken]
        0 references
37      public async Task<IActionResult> Register(RegisterDTO
        registerDTO)
38      {
39       //Check for validation errors
40       if (ModelState.IsValid == false)
41       {
42        ViewBag.Errors = ModelState.Values.SelectMany(temp =>
         temp.Errors).Select(temp => temp.ErrorMessage);
43        return View(registerDTO);
44       }
45
```

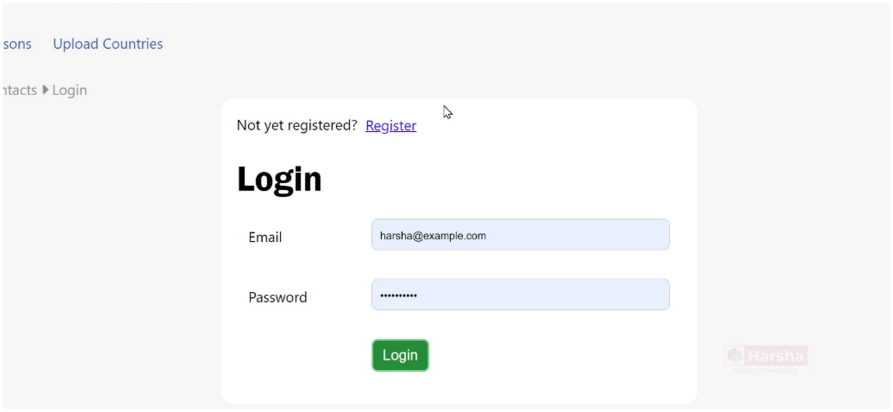
But in general, the **practical problem** is:
If you need to add a ValidateAntiForgeryToken for **every** HTTP POST action method,
it **practically takes one extra step** for the developer each time.
And yeah, it gets repetitive real quick.
So to **automate** this and avoid adding it manually to every action method,
you can just **add this attribute globally** as a **global filter**.
So just...

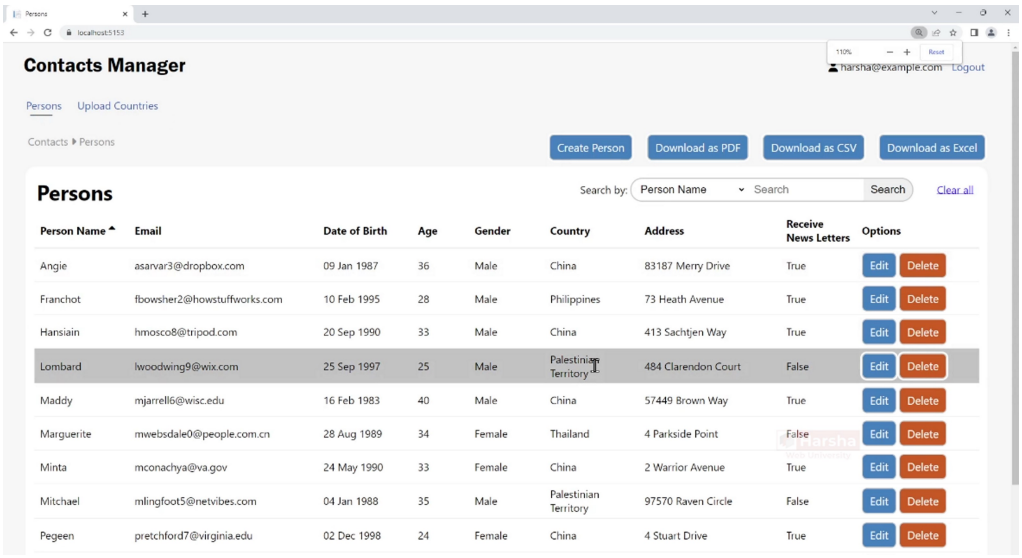
```
}


[HttpPost]
[Authorize("NotAuthorized")]
//[ValidateAntiForgeryToken]
0 references
public async Task<IActionResult> Register(RegisterDTO
registerDTO)
{
 //Check for validation errors
 if (ModelState.IsValid == false)
 {
  ViewBag.Errors = ModelState.Values.SelectMany(temp =>
  temp.Errors).Select(temp => temp.ErrorMessage);
  return View(registerDTO);
 }
```

Open 'ConfigureExtension.cs' file



```
var logger = services.BuildServiceProvider
().GetRequiredService<ILogger<ResponseHeaderActionFilt
er>>();

options.Filters.Add(new ResponseHeaderActionFilter
(logger)
{
 Key = "My-Key-From-Global",
 Value = "My-Value-From-Global",
 Order = 2
});

options.Filters.Add(new
AutoValidateAntiforgeryTokenAttribute());
});
```





Person Name	Email	Date of Birth	Age	Gender	Country	Address	Receive News Letters	Options
Angie	asarvar3@dropbox.com	09 Jan 1987	36	Male	China	83187 Merry Drive	True	Edit Delete
Franchot	fbowsher2@howstuffworks.com	10 Feb 1995	28	Male	Philippines	73 Heath Avenue	True	Edit Delete
Hansiain	hmosco8@tripod.com	20 Sep 1990	33	Male	China	413 Sachtjen Way	True	Edit Delete
Lombard	lwoodwing9@wix.com	25 Sep 1997	25	Male	Palestinian Territory	484 Clarendon Court	False	Edit Delete
Maddy	mjarrell6@wisc.edu	16 Feb 1983	40	Male	China	57449 Brown Way	True	Edit Delete
Marguerite	mwebsdale0@people.com.cn	28 Aug 1989	34	Female	Thailand	4 Parkside Point	False	Edit Delete
Minta	mconachya@va.gov	24 May 1990	33	Female	China	2 Warrior Avenue	True	Edit Delete
Mitchael	mlingfoot5@netvibes.com	04 Jan 1988	35	Male	Palestinian Territory	97570 Raven Circle	False	Edit Delete
Pegeen	pretchford7@virginia.edu	02 Dec 1998	24	Female	China	4 Stuart Drive	True	Edit Delete