# Unit 8 Securing in ASP.NET Core Application

## Authentication:

**Authentication** is the process of checking the identity of the users accessing our application. Most of the applications have a feature for logging in and the application validates user identity against any trusted source such as a database or external login providers (i.e. Facebook, Gmail, Twitter, etc.). This process is called Authentication.

**Authorization** is the process of validating privileges to access a resource of the application. After successful login to the application, the authorization mechanism checks whether the login user has privileges to access the application resource.

## Asp.net core identity:

ASP.NET Core Identity is a membership system that adds login functionality to ASP.NET Core web applications. It includes features for user registration, login, authentication, authorization, and managing user roles. It is built on top of ASP.NET Core and can be easily integrated into any ASP.NET Core application.

**Key features of ASP.NET Core Identity include:**

- **User Authentication:** It provides various options for user authentication, including username/password, external login providers (such as Google, Facebook, Microsoft, etc.), and multi-factor authentication.
- **User Authorization:** It allows you to define roles and policies to control access to different parts of your application based on user roles or specific requirements.
- **User Management:** It provides APIs for managing user accounts, such as creating, updating, deleting, and searching for users.
- **Password Hashing**: It securely stores user passwords using cryptographic hashing algorithms to protect them from unauthorized access.
- **Token-based Authentication:** It supports token-based authentication using JSON Web Tokens (JWT), which allows for stateless authentication and enables building distributed systems.
- **Two-Factor Authentication (2FA):** It supports two-factor authentication for additional security by requiring users to provide a second form of verification, such as a code sent to their mobile device.
- **Account Confirmation and Email Verification**: It includes features for confirming user email addresses and verifying accounts through email confirmation links.
- **Integration with ASP.NET Core Identity UI:** ASP.NET Core Identity UI provides default UI pages for user registration, login, logout, password reset, etc., which can be easily customized to match the look and feel of your application.

**Cross Site Scripting (XSS) in ASP .NET Core**
**Cross Site Scripting (XSS)**

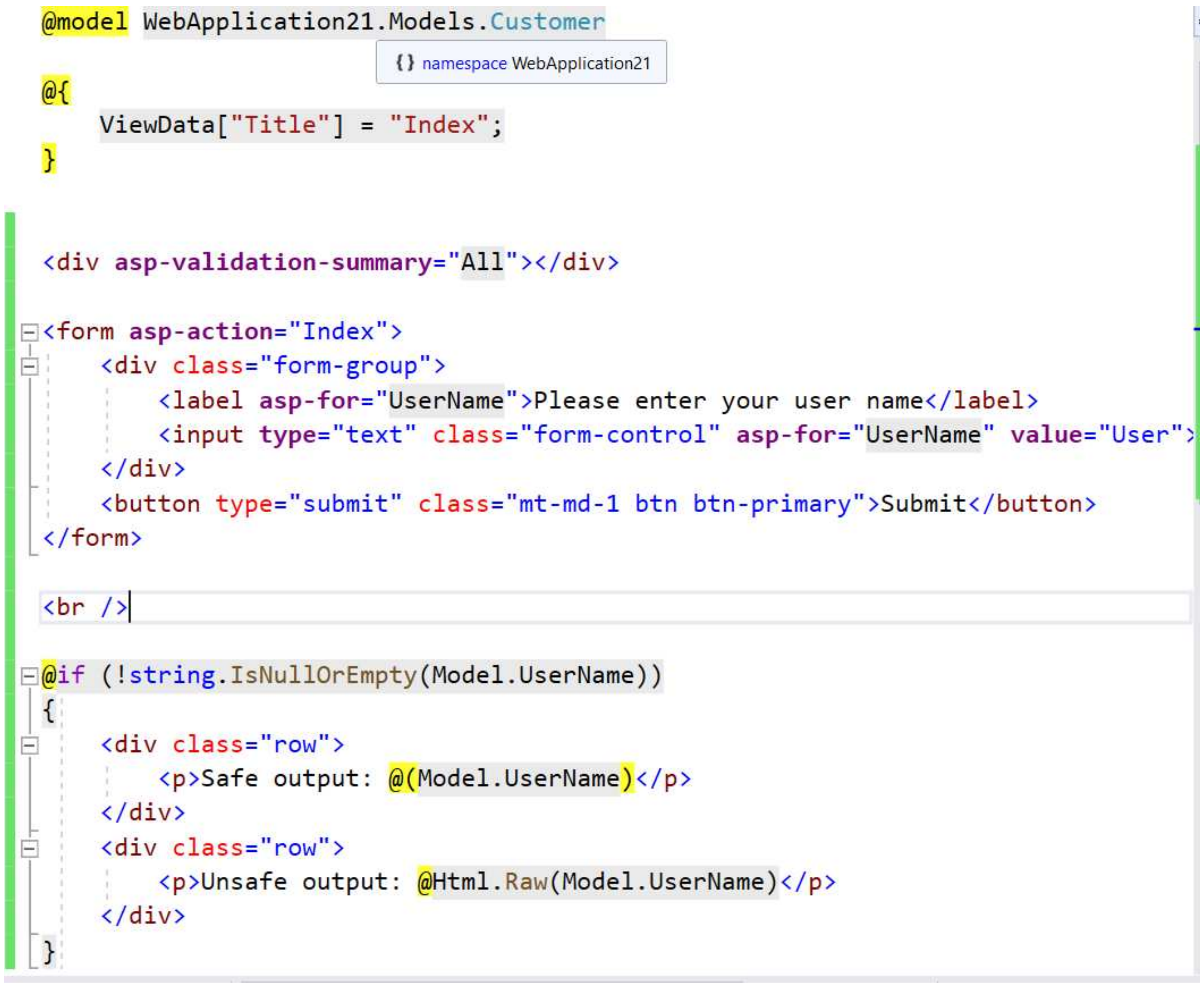
- Cross-Site Scripting (XSS) attacks are a common security vulnerability that occurs when an attacker injects **malicious scripts** into web applications, which are then executed in the context of other users' browsers.
- Cross site scripting is the injection of malicious code in a web application, usually, **JavaScript** but could also be **CSS or HTML**.
- When attackers manage to inject code into your web application, this code often gets also saved in a database.

**Preventing XSS attacks**

1. Validate every user input, either reject or sanitize unknown character, for example, < or > which can be used to create
2. Test every input from an external source.
3. Use HTTP Only for cookies so it is not readable by JavaScript (therefore an attacker can't use JavaScript to read your cookies)
4. Use markdown instead of HTML editors

**Example Preventing XSS Attacks in Asp.net core**

The following code creates a form where the user can enter his user name. The input is displayed once in a safe way and once in an unsafe way.



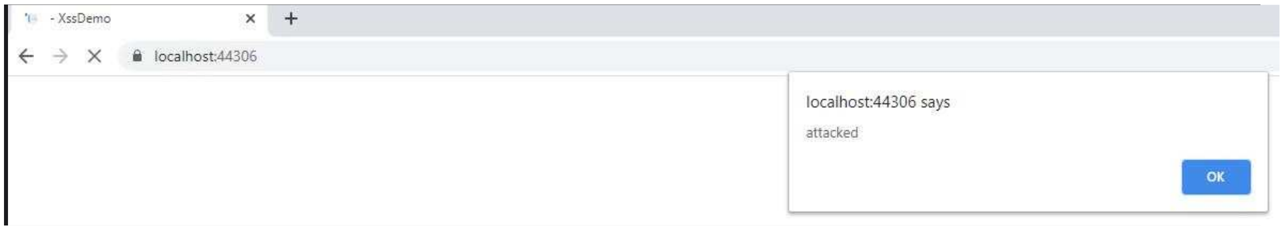
```
@model WebApplication21.Models.Customer
                                {} namespace WebApplication21
@{
    ViewData["Title"] = "Index";
}


<div asp-validation-summary="All"></div>

<form asp-action="Index">
    <div class="form-group">
        <label asp-for="UserName">Please enter your user name</label>
        <input type="text" class="form-control" asp-for="UserName" value="User">
    </div>
    <button type="submit" class="mt-md-1 btn btn-primary">Submit</button>
</form>

<br />

@if (!string.IsNullOrEmpty(Model.UserName))
{
    <div class="row">
        <p>Safe output: @(Model.UserName)</p>
    </div>
    <div class="row">
        <p>Unsafe output: @Html.Raw(Model.UserName)</p>
    </div>
}
```

When a user enters his user name everything is fine. But when an attacker enters Javascript, the Javascript will be executed when the text is rendered inside the unsafe output

tag. When you enter the following code as your name:

<script>alert(“attacked”)</script>

and click submit, an alert windows will be displayed.



```
localhost:44306 says
attacked
OK
```

The injected code got executed

## SQL Injection

SQL Injection is a common security vulnerability that can occur in ASP.NET Core MVC applications if user input is not properly sanitized or validated before being used in SQL queries.

### Here are some ways to prevent SQL Injection attacks in ASP.NET Core MVC:

**Parameterized Queries**: Instead of concatenating user input directly into SQL queries, use parameterized queries or stored procedures. Parameterized queries separate the SQL code from the user input, preventing malicious input from altering the structure of the SQL query.

```
SqlConnection connection = new SqlConnection();
string query = "SELECT * FROM Users WHERE Username = @username AND Password =
@password";
SqlCommand command = new SqlCommand(query, connection);
command.Parameters.AddWithValue("@username", username);
command.Parameters.AddWithValue("@password", password);
```

**ORMs (Object-Relational Mappers)**: Consider using an ORM like Entity Framework Core, which abstracts away direct SQL queries and automatically sanitizes input. ORMs use parameterized queries under the hood and provide a safer way to interact with the database.

```
var user = db.Users.Where(u => u.Username == username && u.Password == password).FirstOrDefault;
```

**Input Validation**: Validate and sanitize user input before using it in SQL queries. Use validation mechanisms provided by ASP.NET Core, such as data annotations, model binding, or custom validation logic.

```
[HttpPost]
public IActionResult Login(LoginViewModel model)
{
   if (ModelState.IsValid)
   {
          }
}
```

**Stored Procedures:** Use stored procedures to encapsulate database logic and avoid dynamic SQL generation. Stored procedures can help prevent SQL Injection by defining a predefined interface for interacting with the database.

## CSRF  In ASP.NET Core

Cross-Site Request Forgery, also known as CSRF (pronounced as “See-Surf”), XSRF, One-Click Attack, and Session Riding, is a type of attack where the attacker forces the user to execute unwanted actions in an application that the user is logged in. The attacker tricks the user into performing actions on their behalf. The impact of this attack depends on the level of permissions that the user has. For example, in a vulnerable bank website, the attacker could transfer an amount of money from the victim’s account or take ownership of the whole account.

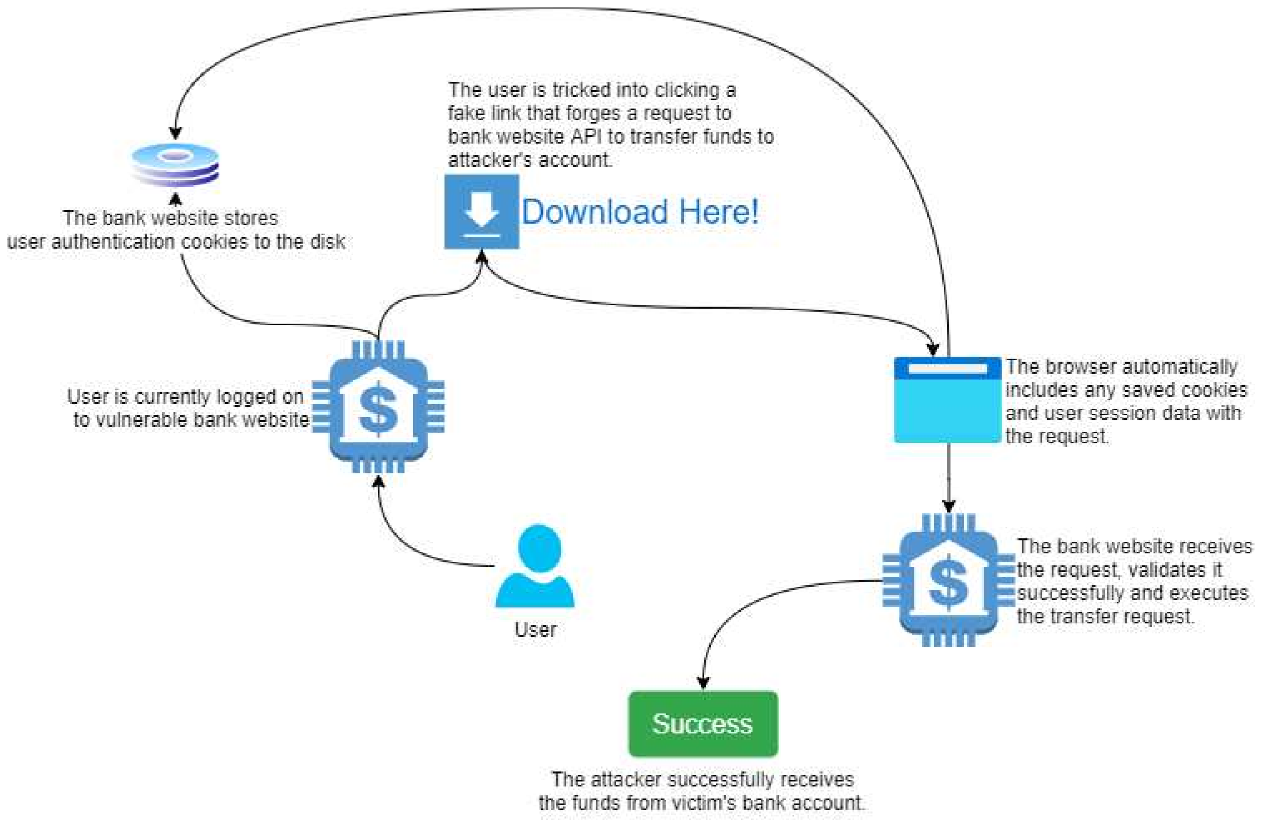
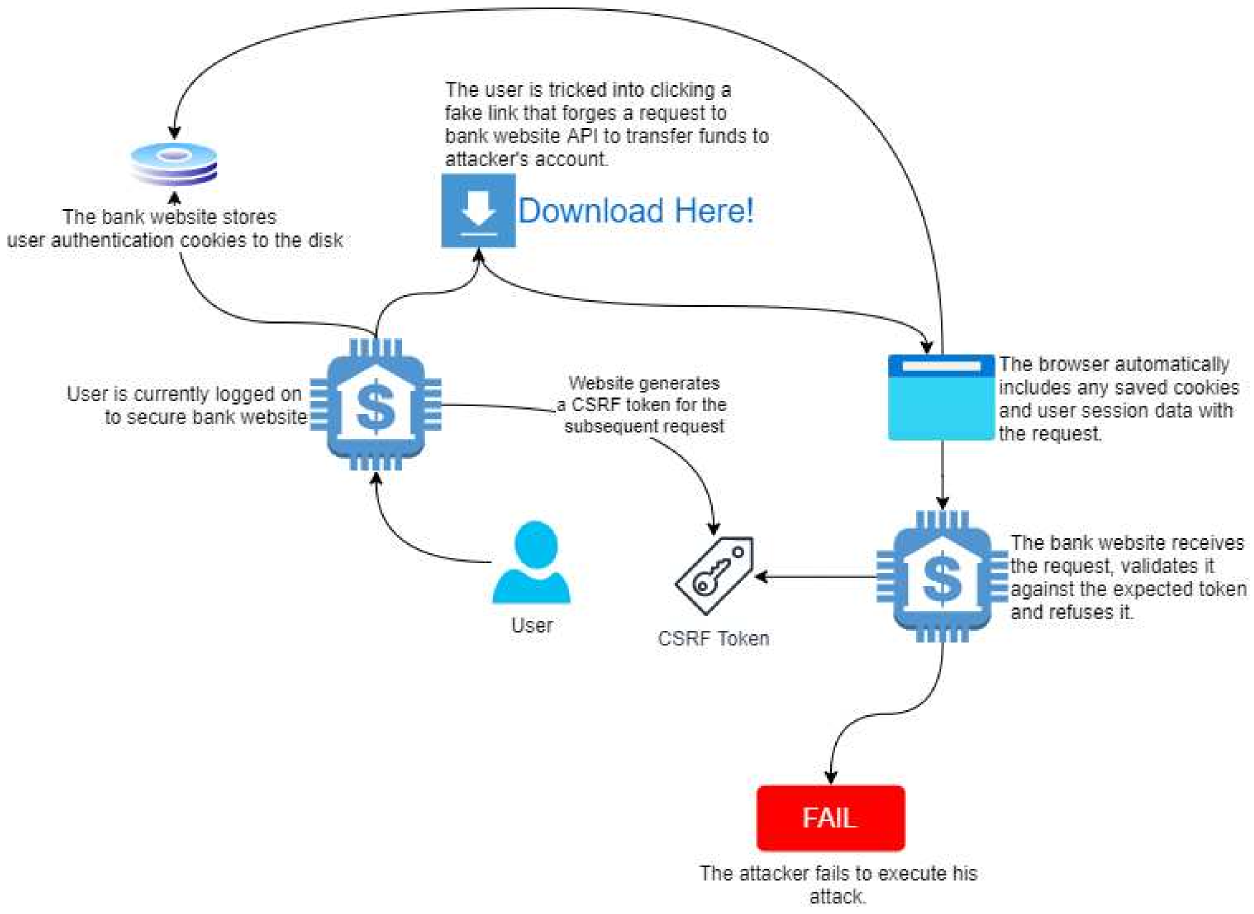

The user is tricked into clicking a fake link that forges a request to bank website API to transfer funds to attacker's account.
Download Here!
The bank website stores user authentication cookies to the disk
User is currently logged on to vulnerable bank website
The browser automatically includes any saved cookies and user session data with the request.
User
The bank website receives the request, validates it successfully and executes the transfer request.
Success
The attacker successfully receives the funds from victim's bank account.

Fig: How attacking done using CSRF

Summary of above Fig:

- A vulnerable website.
- A user who is currently logged on to that website.
- Session and other user cookies that the browser may include in requests.
- Easy-to-predict request parameter.
- User visits a harmful page or clicks on a fake link that executes a forged request to the vulnerable website.

**Preventing using (Anti-Forgery Tokens)**
An anti-forgery token, also called CSRF token, is a unique, secret, unpredictable parameter generated by a server-side application for a subsequent HTTP request made by the client.

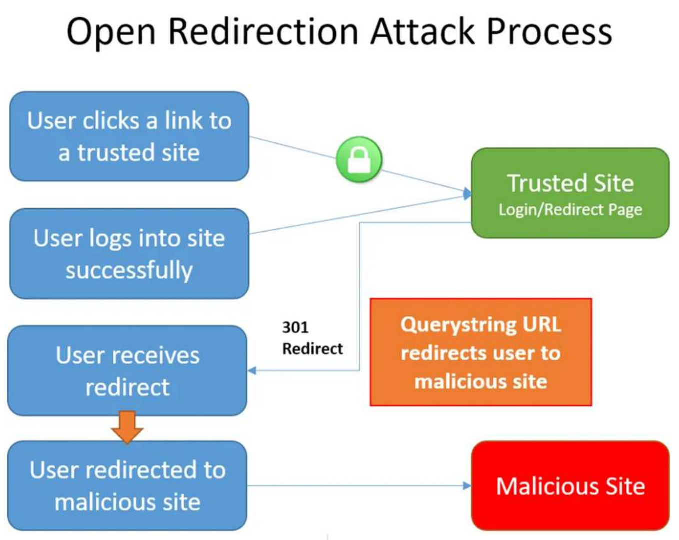

The user is tricked into clicking a fake link that forges a request to bank website API to transfer funds to attacker's account.
Download Here!
The bank website stores user authentication cookies to the disk
User is currently logged on to secure bank website
Website generates a CSRF token for the subsequent request
The browser automatically includes any saved cookies and user session data with the request.
User
CSRF Token
The bank website receives the request, validates it against the expected token and refuses it.
FAIL
The attacker fails to execute his attack.

**Open redirect attacks**

Web applications frequently redirect unauthenticated users to a login page when they access resources that require authentication. After successful authentication, The Redirection URL typically includes return URL query string so that the user can be returned to the original Requested URL.

www.facebook.com#hacked.com

Redirection URL is specified in the query string of the request, an attacker can tamper (interfere with (something) to cause damage or make unauthorized alterations.) with the query string. Query string allows users to redirect to an external, malicious site.


Open Redirection Attack Process
User clicks a link to a trusted site
Trusted Site
Login/Redirect Page
User logs into site successfully
301 Redirect
Querystring URL redirects user to malicious site
User receives redirect
User redirected to malicious site
Malicious Site

1.5 Open Redirect Flow

**Protecting from open redirect:**

**Use LocalRedirect Method**

The local Redirect method is the same as the Redirect method but throws an exception when a nonlocal URL is specified.

**Use Custom Error Page for Error Handling**
Sometimes, Exception may not handle properly and can lead to the exposure of sensitive information such as database configuration info, table names, stored procedures, data structures, and programming coding structure to users.
**Secure Login**
Securing Login with strong username and password
Always use .net core identity feature.
Use Captcha Like Match Captcha, Letter Captcha in your login because bot can't fill captcha
Block IP address if use fails login for more than 1 time
Include Alphabets (A-Z & a-z), Digits (0-9) & Special Characters (! , @, ., #, $, %, ^, &,* and more) in your password and make it strong.

**Use Encryption**

**Clear Cookies**
Manage cookie and remove cookie after logout.
**Use SSL**
Use SSL stands for Secure Socket Layer to make the communication between Client & Server Side Encrypted using a very strong Key.
**keep Framework & Libraries Updated**
**What is Claim and Policies in terms of Authorization:**

**Roles**: Roles represent a collection of **permissions or privileges** that are typically assigned to a group of users based on their responsibilities or functions within an organization.

Example: In a company, there might be roles such as "Manager," "Employee," and "Administrator." Each role would have a predefined set of permissions associated with it. For instance, the Manager role might have permissions to approve certain requests, view reports, and manage team members, while an Employee role might only have permissions to submit requests and access certain resources.

```
[Authorize(Roles = "Admin")]
public class SalesController : Controller
{

}

[Authorize(Roles = "User")]
public class PromotionController : Controller
{

}
```

**Claims**: claims can include attributes such as their identity, group membership, or other properties. **Claims are used to make decisions about access to resources**.

Example: Consider an online banking system. A claim might be the user's account type, such as "Savings Account Holder" or "Checking Account Holder." Based on these claims, the system can determine what actions the user is allowed to perform, such as transferring funds between accounts or viewing transaction history.

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("RequireAdminRole", policy =>
        policy.RequireClaim(ClaimTypes.Role, "Admin"));
});
```

**Policies**: Policies are rules that dictate how access control decisions should be made based on roles, claims, or other factors. Policies define what actions are **allowed or denied under certain conditions**.

Example: In a healthcare organization, there might be a policy that dictates who can access patient medical records. The policy might state that only healthcare professionals directly involved in a patient's care can view their records. This policy would be enforced based on the roles and claims of the users attempting to access the records.

```
[Authorize(Policy = "RequireManagerRole")]
 public class SalesController : Controller
 {
    // Actions for Sales Representatives
 }

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

```
public class PromotionController : Controller
{
   // Actions for managing promotions
}

[Authorize(Policy = "AdminOnly")]
public class AdminController : Controller
{
   // Actions for administrators
}
```

**Lab: Write a asp.net core mvc application to demonstrate simple login authentication using Asp.net core identity.(aspnetIdentityUserExample)**

Example Demonstrate(Authentication and Authorization in asp.net core)
Install All Package from Nuget

**EntityFrameworkCore**

**EntityFrameworkCore.SqlServer**

**EntityFrameworkCore.Tool**

**EntityFrameworkCore.Design**

**Microsoft.AspNetCore.Identity.EntityFrameworkCore**

Adding a class **AppDbContext.cs** Inside **Model** Folder

```
public class AppDbContext:IdentityDbContext
   {
      public AppDbContext(DbContextOptions<AppDbContext> options):base(options)
      {

      }
      protected override void OnModelCreating(ModelBuilder builder)
      {
         base.OnModelCreating(builder);

      }
   }
```

Also Add **RegisterViewModel and LoginViewModel** inside **Model** Folder

```
public class RegisterViewModel
    {
        public string Email { get; set; }
        public string Password { get; set; }
        public string ConfirmPassword { get; set; }
    }

public class LoginViewModel
    {
        [Required]
        [EmailAddress]
        public string Email { get; set; }
        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
        [Display(Name = "Remember me?")]
        public bool RememberMe { get; set; }
    }
```

Add Line Of code inside **Program.cs**

```
Builder.Services.AddDbContextPool<AppDbContext>(options => options.UseSqlServer("Data Source=(localdb)\\MSSQLLocalDB; Initial Catalog=UserIdentityExample; Integrated Security=true"));
Builder.Services.AddIdentity<IdentityUser,IdentityRole>().AddEntityFrameworkStores<AppDbContext>
();
```

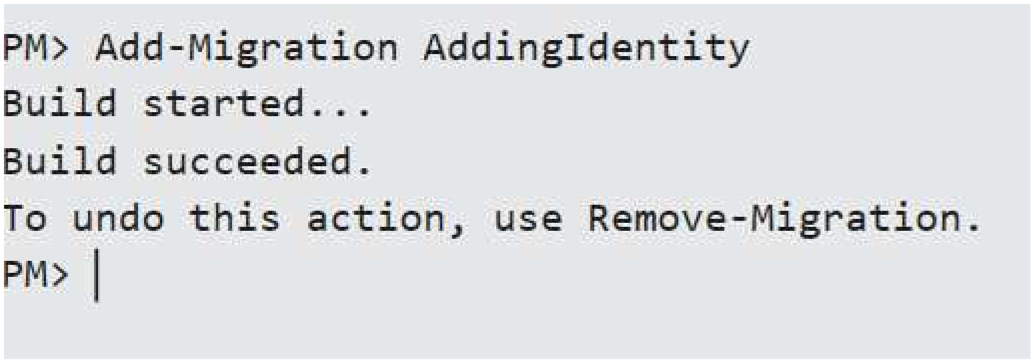
Also Add below line code inside **Program.cs**

```
app.UseAuthentication();
```

**Now, Go To Tools>Nuget Package Manager>Package Manager Console**

**Type:**

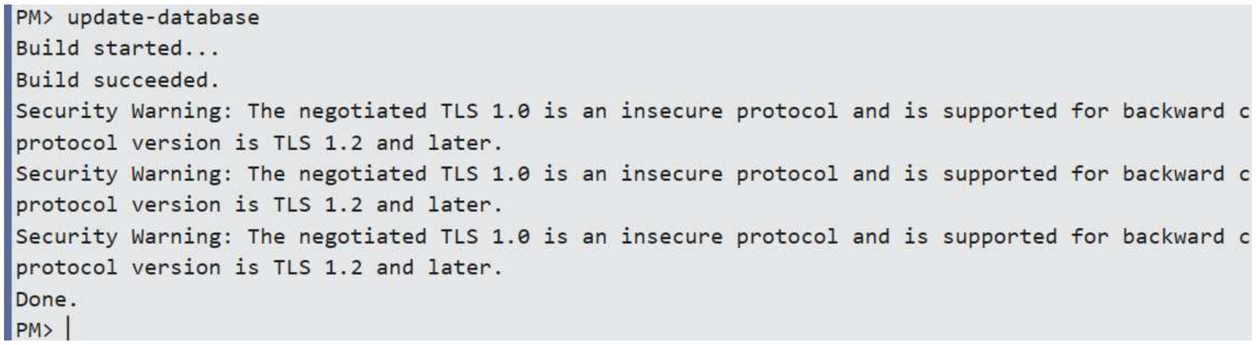
**Add-Migration AddingIdentity**



```
PM> Add-Migration AddingIdentity
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> |
```

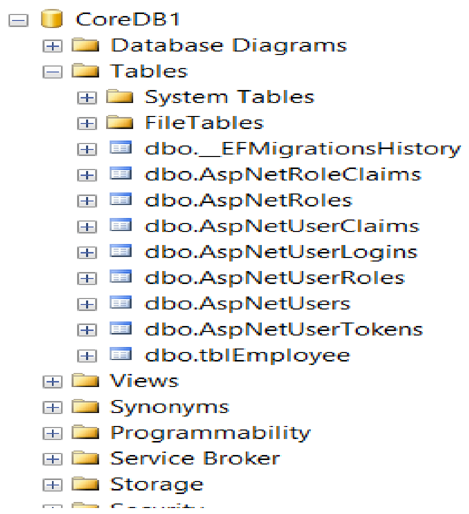
**After That Type**

Update-database



```
PM> update-database
Build started...
Build succeeded.
Security Warning: The negotiated TLS 1.0 is an insecure protocol and is supported for backward c
protocol version is TLS 1.2 and later.
Security Warning: The negotiated TLS 1.0 is an insecure protocol and is supported for backward c
protocol version is TLS 1.2 and later.
Security Warning: The negotiated TLS 1.0 is an insecure protocol and is supported for backward c
protocol version is TLS 1.2 and later.
Done.
PM> |
```

Now it will generate following tables in database as you can find in figure.



```
CoreDB1
    Database Diagrams
    Tables
        System Tables
        FileTables
        dbo.__EFMigrationsHistory
        dbo.AspNetRoleClaims
        dbo.AspNetRoles
        dbo.AspNetUserClaims
        dbo.AspNetUserLogins
        dbo.AspNetUserRoles
        dbo.AspNetUsers
        dbo.AspNetUserTokens
        dbo.tblEmployee
    Views
    Synonyms
    Programmability
    Service Broker
    Storage
```

Now Create **AccountController** inside **Controller** Folder

```
namespace ACHSAuthentication.Controllers
{

    public class AccountController : Controller
    {
        private readonly UserManager<IdentityUser> userManager;
        private readonly SignInManager<IdentityUser> signinManager;
        public AccountController(UserManager<IdentityUser> userManager, SignInManager<IdentityUser>
signinManager)
        {
            this.userManager = userManager;
            this.signinManager = signinManager;
        }
        [HttpGet]
        public IActionResult Login()
        {
            return View();
        }

    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Login(LoginViewModel model, string ReturnUrl)
        {

                var result = await signinManager.PasswordSignInAsync(model.Email, model.Password,
model.RememberMe, false);
                if (result.Succeeded)
                {
                    if (!string.IsNullOrEmpty(ReturnUrl))
                    {
                        return Redirect(ReturnUrl);
                    }
                    else
                    {
                        return RedirectToAction("Index", "Home");
                    }

                }
                else
                {

                    ModelState.AddModelError("", "Invalid Attempts");
                }


            return View(model);
        }
        public IActionResult Register()
        {
            return View();
        }
        [HttpPost]
        public async Task<IActionResult> Register(RegisterViewModel model)
        {
            if(ModelState.IsValid)
            {
                var user = new IdentityUser() { UserName = model.Email, Email = model.Email };
               var result=await userManager.CreateAsync(user, model.Password);
                if(result.Succeeded)
                {
                    await signinManager.SignInAsync(user, isPersistent: false);
                    return RedirectToAction("Index", "Home");
                }
                foreach (var error in result.Errors)
                {
                    ModelState.AddModelError("", error.Description);
                }
            }
            return View();
        }

        public async Task<IActionResult> Logout()
        {
            await signinManager.SignOutAsync();
            return RedirectToAction("Login", "Account");
        }
    }

}
```

Now Create **Register.cshtml**

```
@model ACHSAuthentication.Models.RegisterViewModel

@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Register</title>
    <link href="~/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
</head>
<body>
    <div class="container">

        <h4>New User</h4>
        <hr />
        <div class="row">
            <div class="col-md-4">
                <form asp-action="Register">
                    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
                    <div class="form-group">
                        <label asp-for="Email" class="control-label"></label>
                        <input asp-for="Email" class="form-control" />
                        <span asp-validation-for="Email" class="text-danger"></span>
                    </div>
                    <div class="form-group">
                        <label asp-for="Password" class="control-label"></label>
                        <input asp-for="Password" class="form-control" />
                        <span asp-validation-for="Password" class="text-danger"></span>
                    </div>
                    <div class="form-group">
                        <label asp-for="ConfirmPassword" class="control-label"></label>
                        <input asp-for="ConfirmPassword" class="form-control" />
                        <span asp-validation-for="ConfirmPassword" class="text-danger"></span>
                    </div>
                    <div class="form-group">
                        <input type="submit" value="Register" class="btn btn-primary" />
                        <a href="@Url.Action("Login")">Login</a>
                    </div>
                </form>
            </div>
        </div>
    </div>

</body>
</html>
```

**Now Create Login.cshtml inside Account Folder**

```
@model ACHSAuthentication.Models.LoginViewModel

@{
    ViewData["Title"] = "Login";
    Layout = null;
}
<link href="~/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
<div class="container">
    <h1>Login</h1>
    <div class="row">
        <div class="col-md-4">
            <form asp-action="Login">
                <div asp-validation-summary="ModelOnly" class="text-danger"></div>
                <div class="form-group">
                    <label asp-for="Email" class="control-label"></label>
                    <input asp-for="Email" class="form-control" />
                    <span asp-validation-for="Email" class="text-danger"></span>
                </div>
                <div class="form-group">
                    <label asp-for="Password" class="control-label"></label>
                    <input asp-for="Password" class="form-control" />
                    <span asp-validation-for="Password" class="text-danger"></span>
                </div>
                <div class="form-group form-check">
                    <label class="form-check-label">
                        <input class="form-check-input" asp-for="RememberMe" />
@Html.DisplayNameFor(model => model.RememberMe)
                    </label>
                </div>
                <div class="form-group">
                    <input type="submit" value="Log In" class="btn btn-primary" />
                    <a href="@Url.Action("Register")" class="btn btn-success">REgister</a>
                </div>
            </form>
        </div>
    </div>
</div>
```

**For Authorization us [Authorize] Attribute**

```
          [Authorize]
        public IActionResult Privacy()
        {
            return View();
        }
```