UNIT-8

Securing in ASP.NET Core Application

⊛ Authentication:

The process of determining who made a request is called authentication. Once a request is authenticated and we know who is making the request, we can determine whether they are allowed to execute an action on our server. This process is called authorization.

Q. What is ASP.NET Core Identity? How to Add authentication to apps and identify service configurations? [Imp], [Model Set Q.]

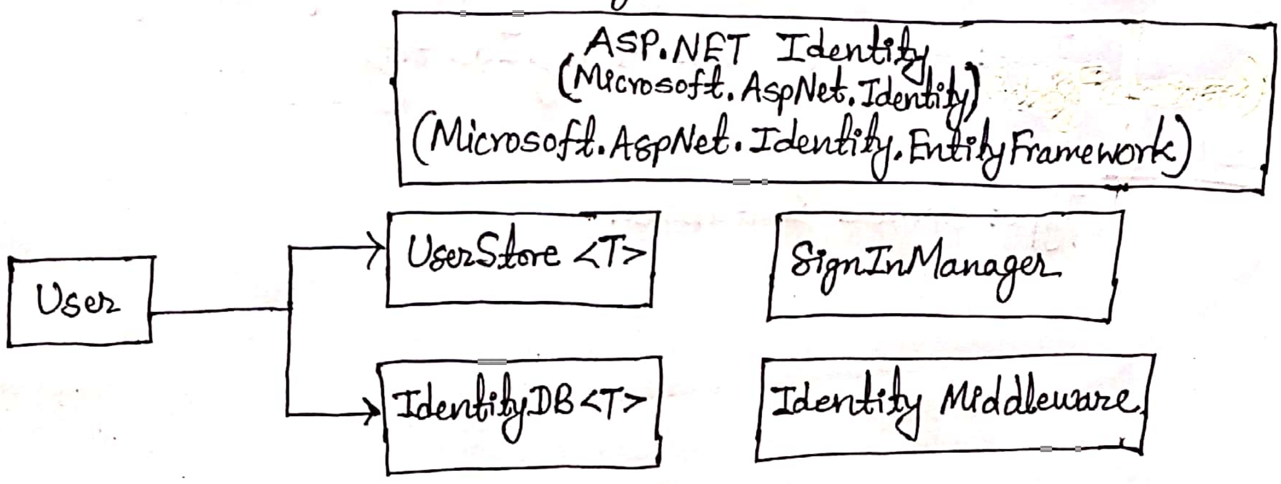
Solution:

ASP.NET Core Identity:

→ It is an API that supports user interface (UI) login functionality.

→ It manages users, passwords, profile data, roles, claims, tokens, email configurations, and more.

Users can create an account with the login information stored in identity or they can use an external login provider. Supported external login providers include Facebook, Google, Microsoft Account, and Twitter. Identity is typically configured using a SQL Server database to store user names, passwords, and profile data. Alternatively, another persistent store can be used, for example, Azure Table Storage.



Steps to Add authentication to apps and identity service configurations:

Step1: Install All Package from Nuget.

EntityFrameworkCore
EntityFrameworkCore.SqlServer
EntityFrameworkCore.Tool
EntityFrameworkCore.Design.

Step2: Adding AppDbContext inside Model Folder and inherits from IdentityDbContext Class.

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

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

Step3: Add Line of code inside Startup.cs inside ConfigureServices Method

```
services.AddDbContextPool<AppDbContext> (options => options.UseSqlServer
 ("Database connection string"));
services.AddIdentity<IdentityUser,IdentityRole>().AddEntityFrameworkStores
 <AppDbContext> ();
```

Step4: Add below line code inside Startup.cs inside Configure Method.

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

Step5: Now, Go To Tools > Nuget Package Manager>Package Manager Console

Type: Add-Migration AddingIdentity

Step6:

Type: Update Database.

⊛ Authorization: The process of determining whether the requested action is allowed.

Roles:

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

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

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

}
```

We can specify multiple roles as a comma separated list:

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

}
```

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

Claims: ~~and Policies:~~ A claim is a name value pair that represents what the subject is, not what the subject can do. Claims based authorization, at its simplest, checks the value of a claim and allows access to a resource based upon that value.

For example, if we want to access a night club the authorization process might be: The door security officer would evaluate the value of our date of birth claim and whether they trust the issuer (the driving license authority) before granting us access.

An identity can contain multiple claims with multiple values and can contain multiple claims of the same type.

Policies: If we apply multiple policies to a controller or action, then all policies must pass before access is granted.

For Example:

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

       }
  }
```

In this example any identity which fulfills the EmployeeOnly policy can access the Payslip action. However in order to call the UpdateSalary action the identity must fulfill both the EmployeeOnly policy and the HumanResources policy.

Securing Action Method in Controller:

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

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

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

⊛Common Vulnerabilities: [Imp]

1) **Cross-site Scripting Attacks:** Cross-Site Scripting (XSS) is a security vulnerability which enables an attacker to place client side scripts (usually JavaScript) into web pages. When other users load affected pages the attacker's scripts will run, enabling the attacker to steal cookies and session tokens, change the contents of the web page through DOM manipulation or redirect the browser to another page. XSS vulnerabilities generally occur when an application takes user input and outputs it to a page without validating, encoding or escaping it.

Prevention Steps:

→ Never put untrusted data into HTML input, unless we follow below steps:

i) Before putting untrusted data inside an HTML element ensure it is HTML encoded.

ii) Before putting untrusted data inside an HTML attribute ensure it is HTML encoded.

iii) Before putting untrusted data into JavaScript place the data in an HTML element whose contents we retrieve at runtime.

iv) Before putting untrusted data into a URL query string ensure it is URL encoded.

2) **SQL Injection attacks:** SQL injection, also known as SQLI, is a common attack vector that uses malicious SQL code for backend database manipulation to access information that was not intended to be displayed. This information may include any number of items, including sensitive data, user lists, or private customer details. SQL injection usually occurs when we ask a user for input, like their username/userid, and instead of a name/id, the user gives us an SQL statement that we will unknowingly run on our database.

SQL Injection Based on 1=1 is Always True:

Let we have input field for UserId and attacker or hacker places/submits value as below:

UserId: 1 OR 1=1

Then, the SQL statement will look like this:

```
SELECT * FROM Users WHERE UserId=1 OR 1 =1;
```

The above SQL is valid and will return all rows from the "Users" table, since 1 OR 1=1 is always TRUE. What if the "Users" table contains names and passwords? Attacker will view entire table rows.

Note: Similarly SQL Injection Based on "=" is Always true. A hacker might get access to user names and passwords in database by simply inserting "OR" "=" into the user name or password fields:

User Name: "or" "="

Password : "or" "="

The code at the server will create a valid SQL statement like this:

```
SELECT * FROM Users WHERE Name=" " or "=" AND Pass=" "
or "="
```

The SQL above is valid and will return all rows from the "Users" table, since "OR""=" is always TRUE.

Use SQL Parameters for Protection:

To protect a web site from SQL injection, we can use SQL parameters. SQL parameters are values that are added to an SQL query at execution time, in a controlled manner. Note that parameters are represented in the SQL statement by a @ marker.

ASP.NET Razor Example:

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId= @0";
db.Execute(txtSQL, txtUserId);
```

Q. Write an application showing sql injection vulnerability and prevention using ado.net. [Model Set Question]. [Imp.]

Solution:

Consider the following action method that validates user login.

```
[HttpPost]
public IActionResult SubmitLogin(String uname, String pwd)
{
 SqlConnection con=new SqlConnection@"Data Source=.\SQLEXPRESS;
  Initial Catalog=db_Mac1;Integrated Security=True");
  con.Open();
 SqlCommand cmd=new SqlCommand("select * from tbl_login where
  uname='"+uname+"' and password='"+pwd+"'", con);
 SqlDataReader dr= cmd.ExecuteReader();
 if (dr.Read())
 {
   return Content ("Login Successful");
  }
 else
 {
  return Content ("Login Unsuccessful");
 }
}
```

The above action method is vulnerable to SQL injection attack. It is because we have used the form input values name and pwd with no data validation at all including Empty form validations.

Preventing SQL Injection: Using parameterized query will prevent such injection.

all same as above only 2 lines SqlCommand and cmd.parameters are different here

```
[HttpPost]
public IActionResult SubmitLogin(String uname, String pwd)
{
 SqlConnection con =new SqlConnection @"Data Source=.\SQLEXPRESS;
 Initial Catalog=db_Mac1;Integrated Security=True");
 con. Open();
SqlCommand cmd=new SqlCommand("select * from tbl_login where
 uname=@uname and password=@pwd", con);
```

```
cmd.Parameters.AddWithValue("@uname", uname);
cmd.Parameters.AddWithValue("@pwd", pwd);
SqlDataReader dr=cmd.ExecuteReader();
if(dr.Read())
{
 return Content("Login Successful");
 }
else
{ return Content("Login Unsuccessful");
 }
}
```

3) Cross-site Request Forgery (CSRF):

CSRF or Cross-Site Request Forgery is a type of web attack that uses a users own browser to post a form from one site to another. These attacks are possible because web browsers send some types of authentication tokens automatically with every request to a website.

Example: User may log in to www.mybankaccount.com and receives a cookie. Sometime later the user goes to www.malicioussite.com and is shown a completely safe form on the surface.

4) Open Redirected Attacks: An Open Redirection is when a web application or server uses a user-submitted link to redirect the user to a given website or page. The redirection typically includes a returnUrl querystring parameter so that the user can be returned to the originally requested URL after they have successfully logged in.

To Prevent Open Redirected Attacks:

1) Local Redirect In Asp.Net Core: Rather than using Redirect, use LocalRedirect so when the user tries to add another domain URL it will prevent it and give an error.

ii) **Exception Message:** The supplied URL is not local. A URL with an absolute path is considered local if it does not have a host/authority part. URL's using virtual paths ('~/') are also local.

iii) **Url.IsLocalUrl In Asp.Net Core:** If we want to use Redirects only then we can check the URL first and then perform a redirection. The code for checking the URL is shown below:

```
Url.IsLocalUrl(returnUrl)
```