Security & Identity

Security, XSS, CSRF, ASP.NET Core Identity, JWT

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Table of Contents

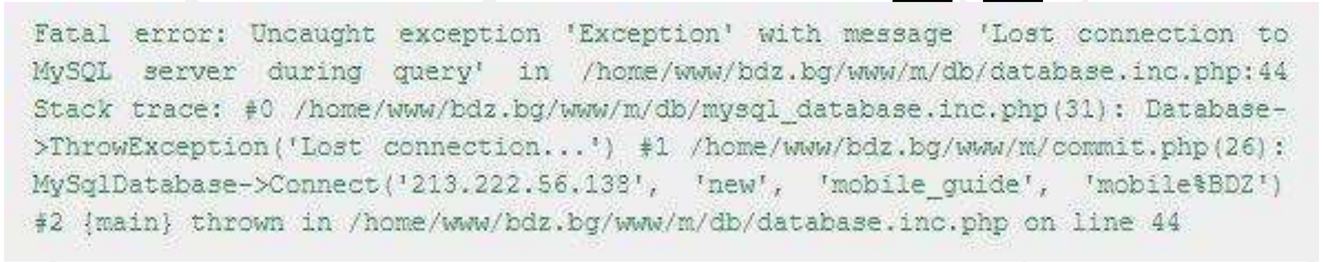
Have a Question?

sli.do

#csharp-web

Most Common Web Security Problems

- **SQL** Injection
- Cross-site Scripting (**XSS**)
- URL/HTTP manipulation attacks (**Parameter Tampering**)
- Cross-site Request Forgery (**CSRF**)
- Brute Force Attacks (also **DDoS**)
- Insufficient **Access** Control
- Too much **information** in Errors
- Missing **SSL** (HTTPS) / **MITM**
- Phishing/Social Engineering
- Security flows in other software we use



```
Fatal error: Uncaught exception 'Exception' with message 'Lost connection to
MySQL server during query' in /home/www/bdz.bg/www/m/db/database.inc.php:44
Stack trace: #0 /home/www/bdz.bg/www/m/db/mysql_database.inc.php(31): Database-
>ThrowException('Lost connection...') #1 /home/www/bdz.bg/www/m/commit.php(26):
MySqlDatabase->Connect('213.222.56.138', 'new', 'mobile_guide', 'mobile%BDZ')
#2 {main} thrown in /home/www/bdz.bg/www/m/db/database.inc.php on line 44
```

https://www.exploit-db.com/

XSS

Cross Site Scripting (XSS)

XSS

- **The Razor view engine** secures you against **XSS** by default
  - If you decide to break it – @Html.Raw(…)
- There are several rules you must follow to be secured:
  - Never put untrusted data into your HTML output
  - Before putting untrusted data somewhere, ensure it is secured
    - Encoded, Parsed, Validated, Checked for malicious contents
  - Untrusted data can be inputted anywhere in the application
    - URLs, HTML Elements, HTML Attributes, JavaScript code etc.

XSS

- ASP.NET Core provides you with anything needed to secure your app
  - Razor automatically encodes all output sourced from variables

```
@{ var untrustedInput = "<\"123\">"; }

@untrustedInput
```

```
&lt;&quot;123&quot;&gt;
```

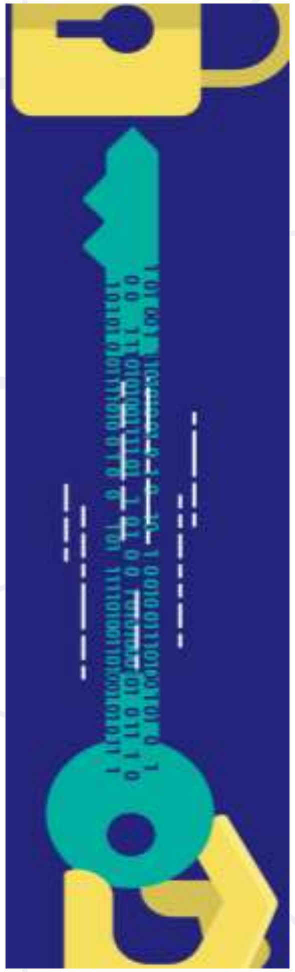
  - You can inject Encoders directly to your Views and use them.

```
@using System.Text.Encodings.Web;
@inject JavaScriptEncoder encoder;

@{ var untrustedInput = "<\"123\">"; }

<script>
    document.write("@encoder.Encode(untrustedInput)");
</script>
```

```
<script>
document.write("\u003C\u0022123
\u0022\u003E");
   </script>
```

XSS

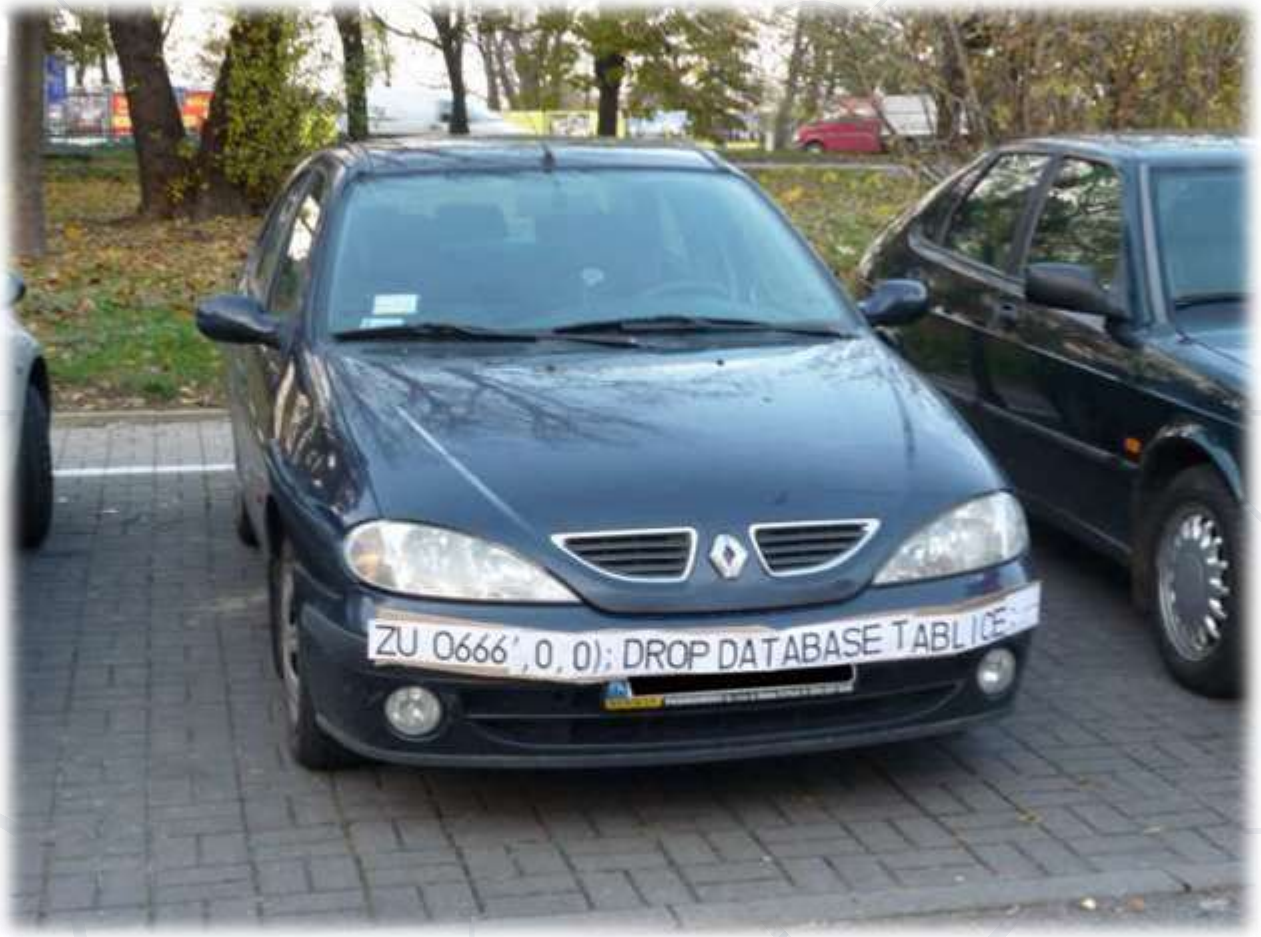
- You can also use ASP.NET Core **Encoder Services**
  - **HtmlEncoder** `<"123">` `&#x3C;&#x22;123&#x22;&#x3E;`
  - **JavaScriptEncoder** `<"123">` `u003C\u0022123\u0022\u003E`
  - **UrlEncoder** `<"123">` `%3C%22123%22%3E`
- Alternatively you can use the static methods
  - **WebUtility.HtmlEncode** and **WebUtility.HtmlDecode**
  - **WebUtility.UrlEncode** and **WebUtility.UrlDecode**



HtmlSanitizer

- HtmlSanitizer is a .NET library for cleaning HTML fragments and documents from constructs that can lead to XSS attacks
- **https://github.com/mganss/HtmlSanitizer**
- Install the **HtmlSanitizer** NuGet package, then:

```
var sanitizer = new HtmlSanitizer();
var html = @"<script>alert('xss')</script><div onload=""alert('xss')""
        style=""background-color: test"">Test<img src=""test.gif""
        style=""background-image: url(javascript:alert('xss')); margin: 10px""></div>";
var sanitized = sanitizer.Sanitize(html, "http://www.example.com");
Debug.Assert(sanitized == @"<div style=""background-color: test"">Test
        <img style=""margin: 10px"" src=""http://www.example.com/test.gif""></div>");
```



SQL Injection

SQL Injection

- The following SQL commands are executed:
  - Usual search (no **SQL injection**):

```
SELECT * FROM Messages WHERE MessageText LIKE '%Nikolay.IT%'"
```

  - SQL-injected search (matches **all records**):

```
SELECT * FROM Messages WHERE MessageText LIKE '%%%%'"
```

```
SELECT * FROM Messages WHERE MessageText LIKE '%' or 1=1 --%'"
```

  - SQL-injected **INSERT** command:

```
SELECT * FROM Messages WHERE MessageText
LIKE '%'; INSERT INTO Messages(MessageText, MessageDate)
VALUES ('Hacked!!!', '1.1.1980') --%'"
```

SQL Injection

- Original SQL Query:

```
string sqlQuery = "SELECT * FROM user WHERE name = '" + username + "' AND
pass='" + password + "'";
```

- Setting username to **John** & password to **' OR '1'= '1** produces

```
string sqlQuery = SELECT * FROM user WHERE name = 'Admin' AND
pass='' OR '1'='1'
```

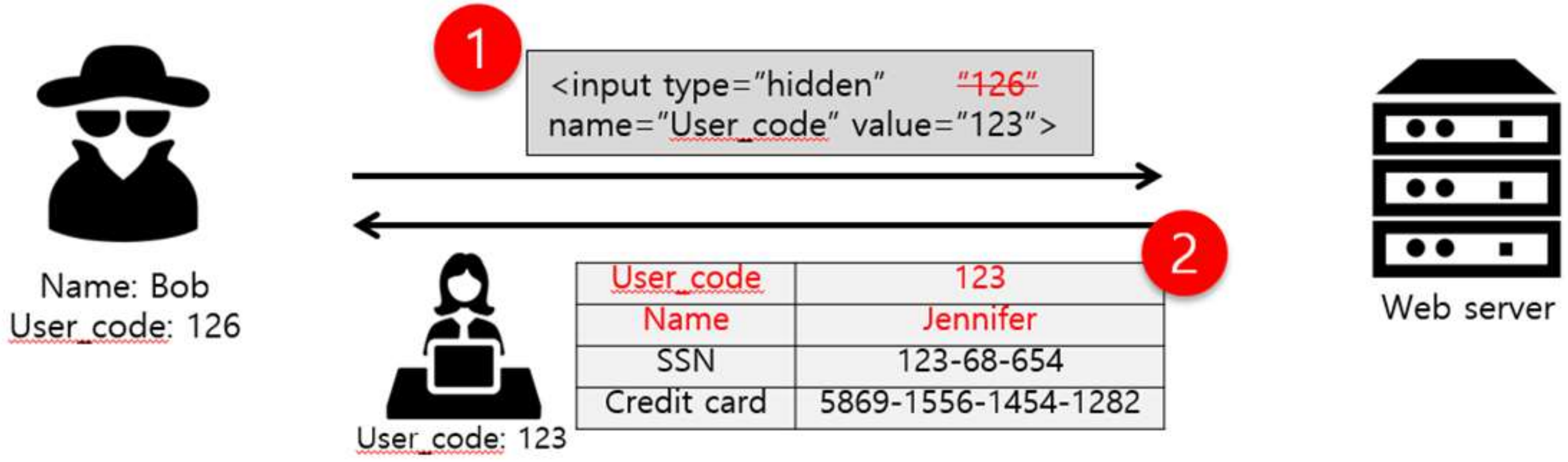
- The result:
  - The user with **username** – “**Admin**” will login **WITHOUT** password
  - The **pass query** will turn into an **bool** expression which is **always true**

Prevent SQL Injection in EF Core

- When using LINQ-to-Entities by default Entity Framework Core **escapes all parameters** before executing the SQL query
- When introducing any user-provided values into a **raw SQL query**, care must be taken to avoid SQL injection attacks
  - By using **SqlParameter** or **interpolated strings** we are protected

```
var user = "Nikolay.IT";
var blogs = context.Blogs
    .FromSqlInterpolated($"EXECUTE dbo.GetLastPostsForUser {user}").ToList();
```

```
var user = new SqlParameter("user", "Nikolay.IT");
var blogs = context.Blogs
    .FromSqlRaw("EXECUTE dbo.GetLastPostsForUser @user", user).ToList();
```



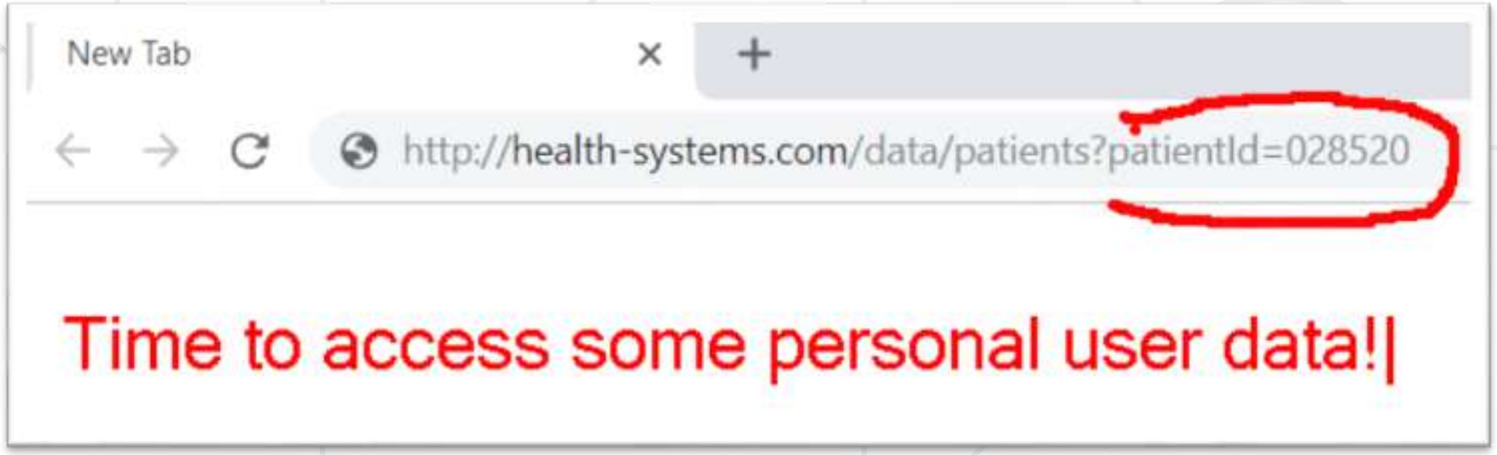
```
<input type="hidden"     ~~"126"~~
name="User_code" value="123">
```

User_code	123
Name	Jennifer
SSN	123-68-654
Credit card	5869-1556-1454-1282

Parameter Tampering

Parameter Tampering

- **Parameter Tampering** is the manipulation of **parameters** exchanged between **client** and **server**
  - Altered query strings, request bodies, cookies
  - Skipped data validations, Injected additional parameters



http://health-systems.com/data/patients?patientId=028520

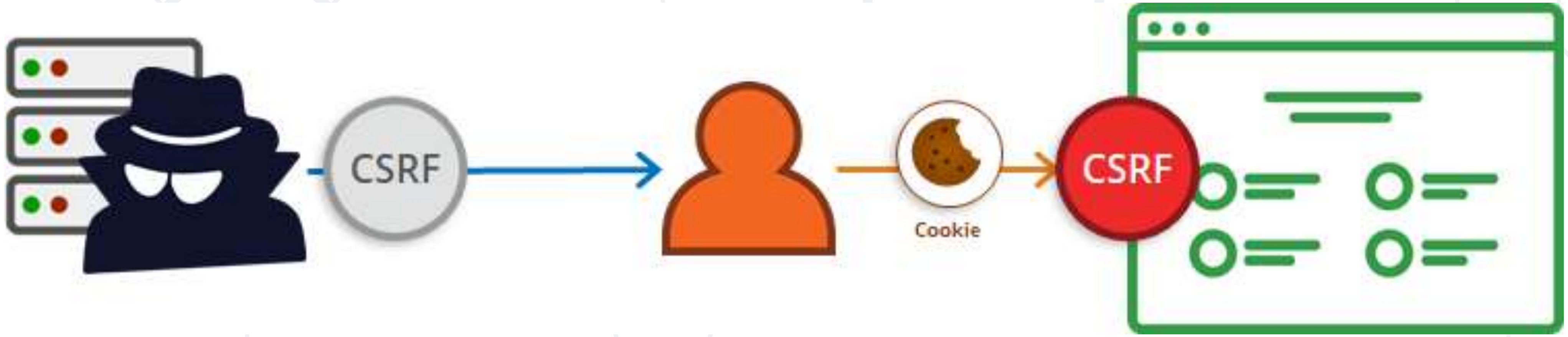
Time to access some personal user data!|

CSRF

Cross-Site Request Forgery

Cross-Site Request Forgery

- **Cross-Site Request Forgery** (**CSRF** / **XSRF**) is a web security attack over the HTTP protocol
  - Allows **executing unauthorized commands** on behalf of some user
    - By using his cookies stored in the browser
  - The user has valid permissions to execute the requested command
  - The attacker uses these permissions maliciously, unbeknownst to the user



Cross-Site Request Forgery

- What **Cross-Site Request Forgery** actually is:

```
<!-- SOME MULTI-COLOR USELESS CLICKBAIT CONTENT -->

<form action="http://good-banking-site.com/api/account" method="post">
    <input type="hidden" name="Transaction" value="withdraw">
    <input type="hidden" name="Amount" value="1000000">
    <input type="submit" value="Click to collect your prize!">
</form>
```

- The user can even **misclick** the button accidentally
  - This will still trigger the attack
  - Security against such attacks is necessary
    - It protects both **your app** and **your clients**

AutoValidateAntiforgeryToken

- When you use the **<form>** tag helper in ASP.NET Core it will automatically add a special hidden field in the form, with random value called **anti-forgery token**
- Then you should require this token to be send
  - For a specific action

```
[AutoValidateAntiforgeryToken]
public IActionResult SendMoney(…) { … }
```

  - For all action in a given controller

```
[AutoValidateAntiforgeryToken]
public class ManageController : Controller
```

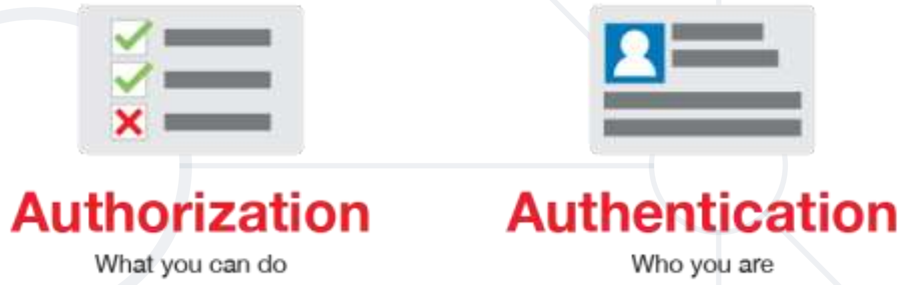
  - Globally for the whole application

```
services.AddMvc(options =>
    options.Filters.Add(new AutoValidateAntiforgeryTokenAttribute()));
```

ASP.NET Core Identity

Authentication vs. Authorization

- **Authentication**
  - The process of verifying the identity of a user or computer
  - Questions: **Who are you**? How you prove it?
  - Credentials can be password, smart card, external token, etc.
- **Authorization**
  - The process of determining what a user is permitted to do on a computer or network
  - Questions: **What are you allowed to do**? Can you see this page?

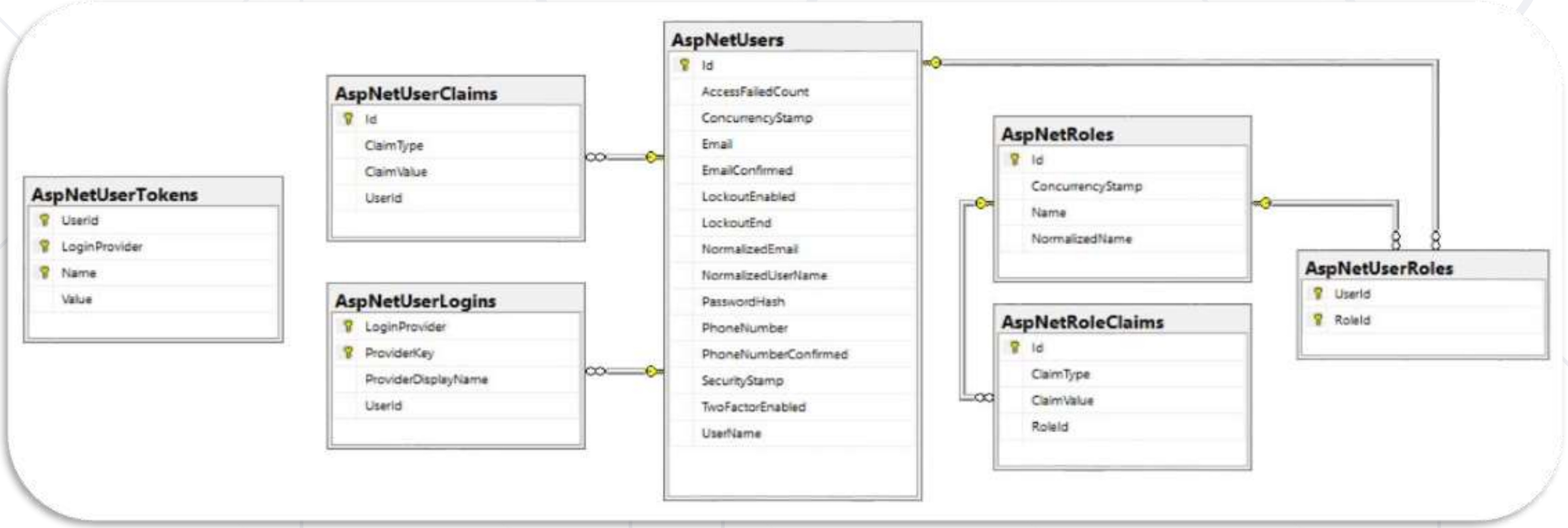


ASP.NET Identity

- The **ASP.NET Core Identity** system
  - Authentication and authorization system for ASP.NET Core
  - Supports ASP.NET Core MVC, Pages, Web API (JWT), SignalR
  - Handles **Users**, **User Profiles**, **Login** / **Logout**, **Roles**, etc.
  - Handles cookie consent and GDPR
  - Supports external login providers
    - Facebook, Google, Twitter, etc.
  - Supports database, Azure, Active Directory, Windows Users, etc.

ASP.NET Core Identity

- Typically, the **ASP.NET Core** identity data is stored in relational database
  - Data is persisted using **Entity Framework Core**
  - You have some control over the internal database schema



ASP.NET Identity System Setup

- Setup **ASP.NET Identity**
  - Using the ASP.NET **project templates** from Visual Studio
    - And then customize it

RECOMMENDED

  - **By hand**
    - Install NuGet packages, manual configuration, create EF mappings (models), view models, controllers, views, pages, etc.
- Required NuGet packages
  - **Microsoft.AspNetCore.Identity.EntityFrameworkCore** (Models)
  - **Microsoft.AspNetCore.Identity.UI** (Pages)

ASP.NET Core Project Template Authentication

- **ApplicationDbContext.cs**
  - Holds the EF data context
  - Provides access to the application's data using model objects
- **Startup.cs**
  - Can configure cookie-based (or JWT) authentication
  - May enable external login (e.g. Facebook login)
  - Can change default identity settings
  - Can enable **RoleManager** with `.AddRoles<IdentityRole>()`

ASP.NET Core Identity Settings

- Identity settings can be defined in **Startup.cs**

```
public void ConfigureServices(IServiceCollection services)
{
  ...
  services.AddDefaultIdentity<IdentityUser>(options =>
      {
          // password, lockout, emails, user, etc.
          options.SignIn.RequireConfirmedAccount = false;
          options.Password.RequireNonAlphanumeric = false;
          options.Lockout.MaxFailedAccessAttempts = 5;
          options.User.RequireUniqueEmail = true;
      })
    .AddRoles<IdentityRole>() // This is required for using roles
    .AddEntityFrameworkStores<ApplicationDbContext>()
}
```

ASP.NET Core Identity

- In **Configure()** there are 2 middlewares involved with identity
  - **UseAuthentication()** adds authentication middleware to the request pipeline
  - **UseAuthorization()** adds authorization to the request pipeline
- There are also **DI Services** for helping us with identity
  - **SignInManager** – sign-in, sign-out, two-factor auth, lockout, etc.
  - **UserManager** – create, read, update or delete users data
  - **RoleManager** – create, read, update or delete roles data

User Registration

```
var newUser = new IdentityUser()
{
    UserName = "John",
    Email = "john@gmail.com",
    PhoneNumber = "+359 2 981 981"
};

var result = await userManager.CreateAsync(newUser, "S0m3@Pa$$");

if (result.Succeeded)
    // User registered
else
    // result.Errors holds the error messages
```

User Login / Logout

- **Login**

```
bool rememberMe = true;
bool shouldLockout = false;
var signInStatus = await signInManager.PasswordSignInAsync(
    "John", "S0m3@Pa$$", rememberMe, shouldLockout);

if (signInStatus.Succeeded)
    // Sucessfull login
else
    // Login failed
```

- **Logout**

```
await signInManager.SignOutAsync();
```

ASP.NET Authorization Attributes

- Use the [**Authorize**] and [**AllowAnonymous**] attributes to configure **Authorized** / **Anonymous access** for **Controller** / **Action**

```
[Authorize]
public class AccountController : Controller
{
  // GET: /Account/Login (anonymous)
  [AllowAnonymous]
  public async Task<IActionResult> Login(string returnUrl) { … }

  // POST: /Account/LogOff (for logged-in users only)
  [HttpPost]
  public async Task<IActionResult> Logout() { … }
}
```

Check the Currently Logged-In User

```
// GET: /Account/Roles (for logged-in users only)
[Authorize]
public ActionResult Roles()
{
    var currentUser = await userManager.GetUserAsync(this.User);
    var roles = await userManager.GetRolesAsync(currentUser);
    ...
}
```

```
// GET: /Account/Data (for logged-in users only)
[Authorize]
public ActionResult Data()
{
    var currentUser = await userManager.GetUserAsync(this.User);
    var currentUserUsername = await userManager.GetUserNameAsync(currentUser);
    var currentUserId = await userManager.GetUserIdAsync(currentUser);
    ...
}
```

Add User to a Role

- Adding a User to existing role:

```
var roleName = "Administrator";
var roleExists = await roleManager.RoleExistsAsync(roleName);

if (roleExists)
{
    var user = await userManager.GetUserAsync(User);
    var result = await userManager.AddToRoleAsync(user, roleName);

    if (result.Succeeded)
        // The user is now Administrator
}
```

Require Logged-In User in Certain Role

- Give access only to Users in Role "**Administrator**":

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

- Give access if User's Role is "**User**", "**Student**" or "**Trainer**":

```
[Authorize(Roles="User, Student, Trainer")]
public ActionResult Roles()
{
    …
}
```

Check the Currently Logged-In User's Role

```
// GET: /Home/Admin (for logged-in admins only)
[Authorize]
public ActionResult Admin()
{
    if (this.User.IsInRole("Administrator"))
    {
        ViewBag.Message = "Welcome to the admin area!";
        return View();
    }

    return this.View("Unauthorized");
}
```

ASP.NET Core User Manager

- **UserManager<TUser>** - APIs for managing users in a persistence store

Category		
AddClaimsAsync(…)	FindByEmailAsync(…)	GenerateChangeEmailTokenAsync(…)
AddToRoleAsync(…)	FindByIdAsync(…)	GenerateEmailConfirmationTokenAsync(…)
IsInRoleAsync(…)	FindByNameAsync(…)	GeneratePasswordResetTokenAsync(…)
GetUserId(…)	GetClaimsAsync(…)	GetAuthenticationTokenAsync(…)
ConfirmEmailAsync(…)	GetEmailAsync(…)	IsEmailConfirmedAsync(…)
ChangeEmailAsync(…)	GetRolesAsync(…)	CreateSecurityTokenAsync(…)
CreateAsync(…)	GetUserAsync(…)	ResetPasswordAsync(…)
DeleteAsync(…)	CheckPasswordAsync(…)	RemoveFromRoleAsync(…)
Dispose(…)	UpdateAsync(…)	RemoveClaimsAsync(…)

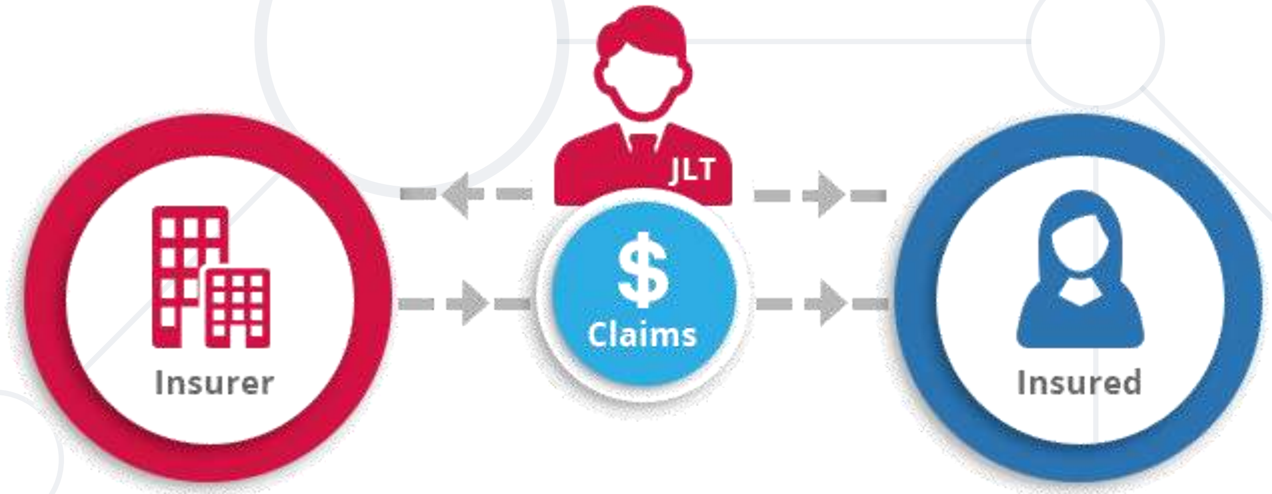
Identity Claims

Claims

- **Claim**-based identity is a common technique used in applications
  - Applications acquire identity info about their users through **Claims**
- A **Claim** is a statement that one subject makes about itself
  - It can be about a name, group, ethnicity, privilege, association etc.
  - The subject making the claim is a **provider**
- **Claim**-based identity **simplifies** authentication logic
  - Commonly used in individual application parts, or micro-apps
  - Claims data is usually represented as key-value pairs

Claims

- In **ASP.NET Core**, **Claim**-based auth checks are **declarative**
  - The developer embeds them against a **Controller** or an **Action**
  - The developer specifies **required claims** to access the functionality
- **Claims requirements** are policy based
  - The developer must register a policy expressing claims requirements
- **Claims** are **name-value** pairs



Policies

- The simplest type of **claim** policy checks only for the **presence** of a claim
  - The **value** of the **claim** is not checked

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

```
[Authorize(Policy = "EmployeeOnly")]
public IActionResult VacationBalance()
{
    //This action is accessible only by Identities with the "EmployeeOnly" Claim...
    return View();
}
```

Identity – Extending & Scaffolding

Scaffolding ASP.NET Core Identity

- **Identity** is provided as a **Razor Class Library** using **Razor Pages**
- The **scaffolder** can be configured to generate source code
  - If you need to modify the code and change the behavior
- Most of the necessary code is generated by the **scaffolder**
  - Your project will need an **update**, before the process is complete
- The **scaffolder** generates a helpful ***ScaffoldingReadme.txt*** file
  - Contains instructions on what's needed to complete the scaffolding
- **Source control** is suggested, before attempting **scaffolding**

Extending ASP.NET Core Identity

- **ApplicationUser.cs** – can add user functionality
- Extends the **user** information for the ASP.NET Core application derived from **IdentityUser**
  - **Id** (unique User Id, string holding a **GUID**)
    - E.g. **313c241a-29ed-4398-b185-9a143bbd03ef**
  - **Username** (unique username), e.g. **maria**
  - **Email** (email address – can be unique), e.g. **mm@gmail.com**
- May hold **additional fields**, e.g. first name, last name, date of birth

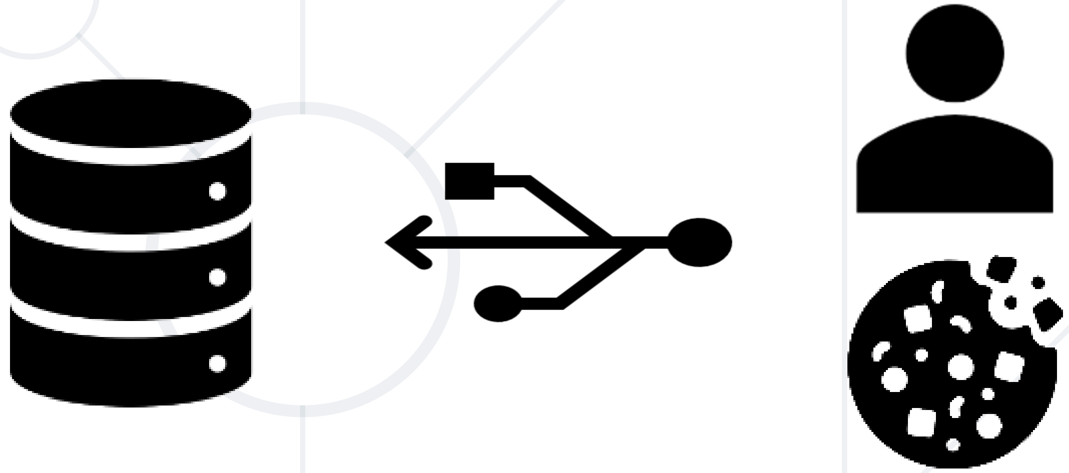
Authentication Types

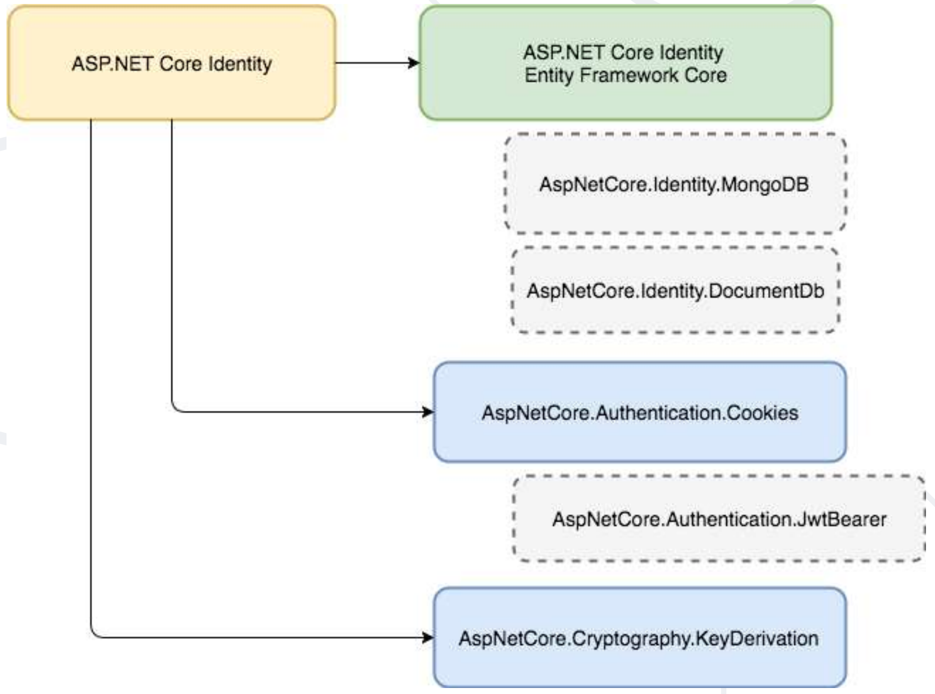
Authentication Types

- There are many types of auth in **ASP.NET Core** applications
  - **Cookie-based** Authentication & Authorization (Identity)
  - **Windows** Authentication & Authorization
  - **Cloud-based** Authentication & Authorization
  - **JSON Web Tokens** (JWT) Authentication & Authorization

Cookie-Based Authentication & Authorization

- **Cookie-Based** auth is the **ASP.NET Core** app auth mechanism
  - Authentication is entirely **Cookie-based**
  - This is a major difference from **ASP.NET MVC**
  - The **Principal** is based on **claims**




ASP.NET Core Identity
ASP.NET Core Identity Entity Framework Core
AspNetCore.Identity.MongoDB
AspNetCore.Identity.DocumentDb
AspNetCore.Authentication.Cookies
AspNetCore.Authentication.JwtBearer
AspNetCore.Cryptography.KeyDerivation

Windows Authentication & Authorization

- **Windows** auth is a more complex auth mechanism
  - Relies on the operating system to authenticate users
  - Credentials are hashed before sent across the network
  - Best suited for intranet environments
    - Clients, Users, Servers belong to the same Windows domain (AD)

Cloud-based Authentication & Authorization

- **Cloud-based** auth is a more modern authentication approach
  - Authentication & Authorization work is outsourced
  - An **external platform** handles the User functionality
  - Ensures flexibility and speed
  - Greatly decouples the auth functionality from the others

JWT Authentication & Authorization

- **JSON Web Tokens** is a modern JavaScript-based auth mechanism
  - Compact and self-contained
  - Focused on signed tokens
    - Work with claims
    - Data is encrypted
  - Used for auth & information exchange
  - Commonly used, when developing **REST**
  - Extremely simple to comprehend
  - Used in Angular/React/Vue.js/Blazor applications

Social Accounts

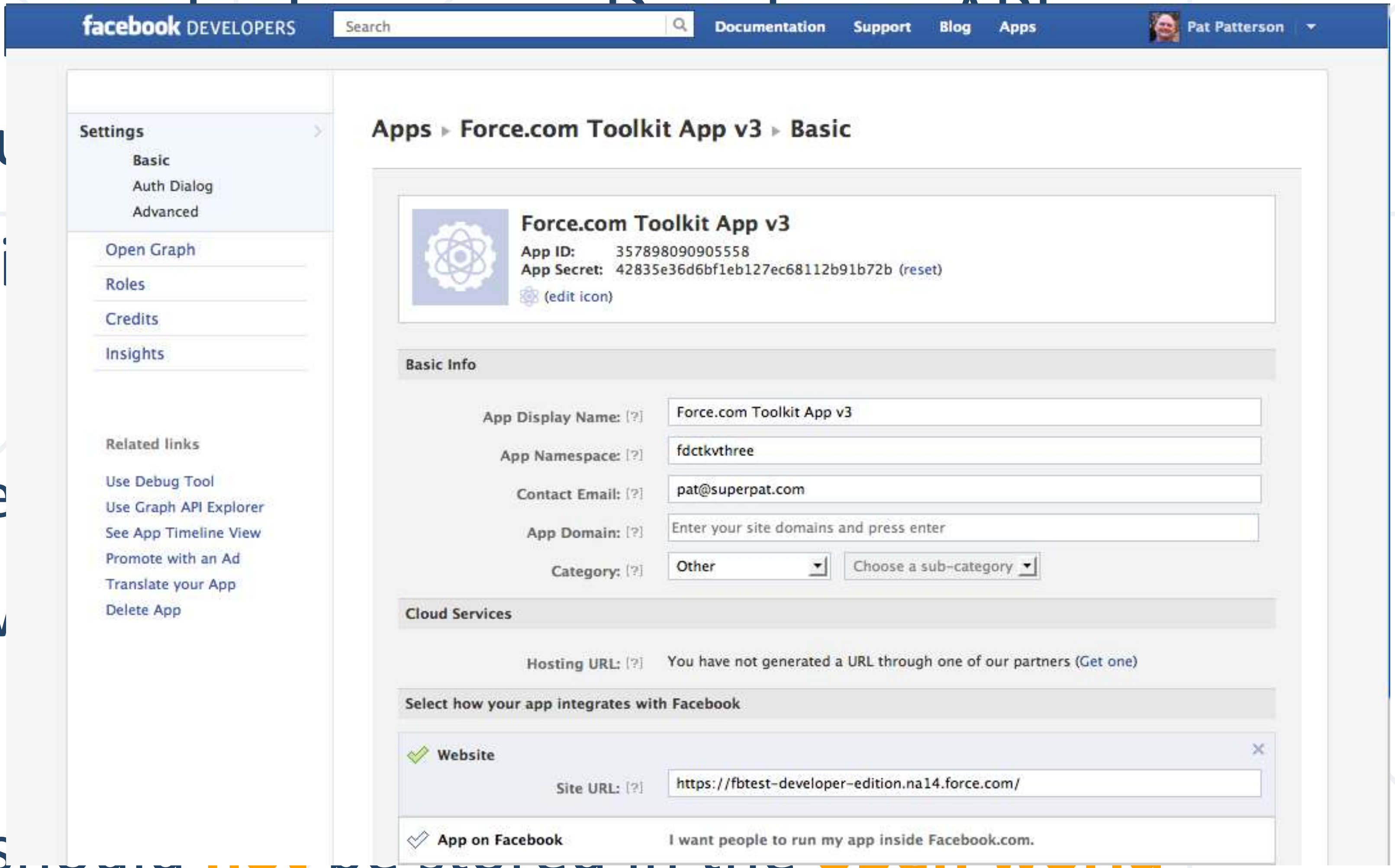
Social Accounts

- Enabling users to sign in with their existing credentials is convenient
  - Shifts the complexities of managing the sign-in process to third party
  - Enhances user experience by minimizing their auth activities
- **ASP.NET Core** supports built-in external login providers for:
  - Google
  - Facebook
  - Twitter
  - Microsoft

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddAuthentication()
        .AddGoogle(googleOptions => { ... })
        .AddFacebook(facebookOptions => { ... })
        .AddTwitter(twitterOptions => { ... })
        .AddMicrosoftAccount(microsoftOptions => { ... });
    ...
}
```

Social Accounts

- Each External Login
  - You have to configu
  - That application wi
    - Application ID
    - Application Secre
  - These credentials v
    - You authenticate
  - These credentials should **not** be stored in the **open world**


facebook DEVELOPERS | Search | Documentation | Support | Blog | Apps | Pat Patterson

Settings
Basic
Auth Dialog
Advanced
Open Graph
Roles
Credits
Insights

Related links
Use Debug Tool
Use Graph API Explorer
See App Timeline View
Promote with an Ad
Translate your App
Delete App

Apps ▸ Force.com Toolkit App v3 ▸ Basic

Force.com Toolkit App v3
App ID: 357898090905558
App Secret: 42835e36d6bf1eb127ec68112b91b72b (reset)
(edit icon)

Basic Info

App Display Name: [?]	Force.com Toolkit App v3
App Namespace: [?]	fdctkvthree
Contact Email: [?]	pat@superpat.com
App Domain: [?]	Enter your site domains and press enter
Category: [?]	Other / Choose a sub-category

Cloud Services

Hosting URL: [?] You have not generated a URL through one of our partners (Get one)

Select how your app integrates with Facebook

✓ Website
Site URL: [?] https://fbtest-developer-edition.na14.force.com/

✓ App on Facebook — I want people to run my app inside Facebook.com.

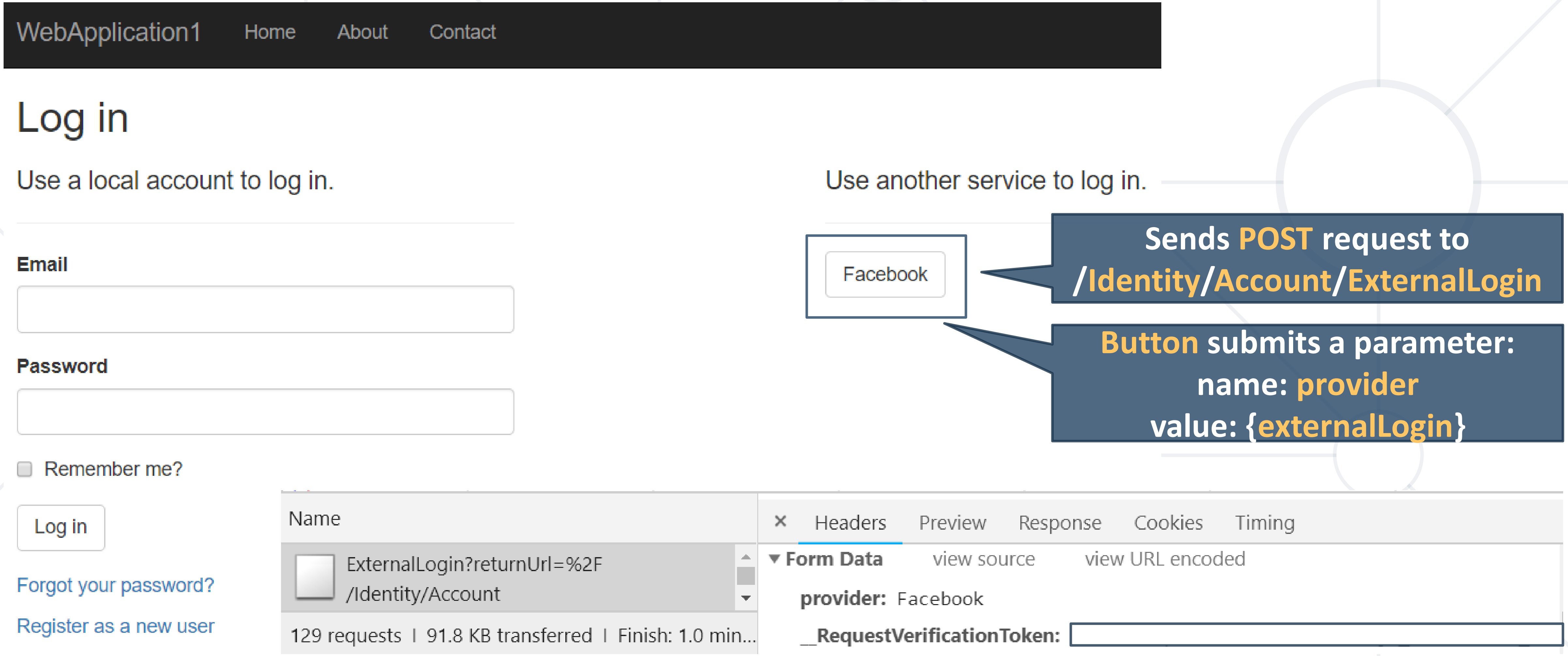
Social Accounts

- On the back-end, it is quite simple, and quite clean
- Example: **Facebook**

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddAuthentication()
        .AddFacebook(facebookOptions => {
            facebookOptions.AppId = Configuration["Authentication:Facebook:AppId"];
            facebookOptions.AppSecret = Configuration["Authentication:Facebook:AppSecret"];
    });
    ...
}
```

- If you use the **default ASP.NET Core Login** page, this will add a **form**

Social Accounts



WebApplication1 Home About Contact

Log in

Use a local account to log in.

Email

Password

Remember me?

Log in

Forgot your password?

Register as a new user

Use another service to log in.

Facebook

Sends POST request to /Identity/Account/ExternalLogin

Button submits a parameter:
name: provider
value: {externalLogin}

Name

ExternalLogin?returnUrl=%2F
/Identity/Account

129 requests | 91.8 KB transferred | Finish: 1.0 min...

× Headers Preview Response Cookies Timing

▾ Form Data view source view URL encoded

provider: Facebook

__RequestVerificationToken: [redacted]

JSON Web Tokens

JWT

JSON Web Tokens

- **JWT** is a method for representing claims between two parties
  - An open, industry standard – RFC 7519
  - Easy to use, and at the same time – absolutely secured
- When the user successfully **authenticates** (login) using their credentials:
  - A **JSON Web Token** is generated and returned
    - It must be stored (in **local** / **session** storage, **cookies** are also an option)
- Whenever a protected route is accessed, the user agent sends the **JWT**
  - Typically in an **Authorization** header, using the **Bearer** schema

JSON Web Tokens

- **JWT** is absolutely **stateless**, nothing is stored on the server
- Here is an example of an encoded and decoded **JSON Web Token**

The parts of the **token** are separated by **dots**

As any normal auth **JWT** also has an **expiration**

Encoded

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI
6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDI
yfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6y
JV_adQssw5c
```

The parts of the **token** are in a strict **order**

The token data does not change the **token format**

Decoded

Header: (algorithm, token type)

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload: (data)

```
{ "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Verify Signature

```
HMACSHA256(base64UrlEncode(H...) +
"." + base64UrlEncode(P...), key)
```

JWT in ASP.NET Core

- **JWT** in **ASP.NET Core** is configured in **ConfigureServices()**
  - Install **Microsoft.AspNetCore.Authentication.JwtBearer**

```
public class JwtSettings
{
    public string Secret { get; set; }
}
```

appsettings.json

```
{
  "JwtSettings": {
    "Secret": "super-secret"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

```
public void ConfigureServices(IServiceCollection services)
{
    // Configure strongly typed settings objects
    var jwtSettingsSection =
            Configuration.GetSection("JwtSettings");
    services.Configure<JwtSettings>(jwtSettingsSection);

    // Configure JWT authentication
    var jwtSettings = jwtSettingsSection.Get<JwtSettings>();
    var key = Encoding.ASCII.GetBytes(jwtSettings.Secret);
    services.AddAuthentication(...)
            .AddJwtBearer(...);

    // Configure DI for application services
    services.AddScoped<IUserService, UserService>();
}
```

JWT in ASP.NET Core

- **JWT** in **ASP.NET Core** is implemented using a middleware

```
services.AddAuthentication(options => {
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(options => {
    options.RequireHttpsMetadata = false;
    options.SaveToken = true;
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(key),
        ValidateIssuer = false,
        ValidateAudience = false
    };
});
...
// Don't forget to add app.UseAuthentication(); and app.UseAuthorization();
```

JWT in ASP.NET Core

- **JWT** in **ASP.NET Core** is implemented using a middleware

```
[ApiController]
[Route("/api/[controller]")]
public class UsersController : ControllerBase
{
    private IUserService _userService;

    public UsersController(IUserService userService)
    {
        this.userService = userService;
    }

    [HttpPost("login")]
    public IActionResult Login([FromBody]LoginUserBindingModel loginUser)
    {
        ...
    }
}
```

JWT in ASP.NET Core

- **JWT** in **ASP.NET Core** is implemented using a middleware
  - The Controller Action (**Endpoint**) is kept "**thin**" to a maximum

```
...
[HttpPost("login")]
public IActionResult Login([FromBody]LoginUserBindingModel loginUser)
{
    var user = this.userService.Authenticate(loginUser.Username, loginUser.Password);

    if (user == null)
    {
        return BadRequest(new { message = "Username or password is incorrect" });
    }

    return Ok(user);
}
```

JWT in ASP.NET Core

- **JWT** in **ASP.NET Core** is implemented using a middleware

```
public class UserService : IUserService
{
    private readonly AppDbContext context;

    private readonly JwtSettings jwtSettings;

    public UserService(AppDbContext context, IOptions<JwtSettings> jwtSettings)
    {
        this.context = context;
        this.jwtSettings = jwtSettings.Value;
    }

    public User Authenticate(string username, string password)
    {
        ...
    }
}
```

JWT in ASP.NET Core

- **JWT** in **ASP.NET Core** is implemented using a middleware

```
...
public User Authenticate(string username, string password)
{
    var user = this.context.Users.SingleOrDefault(x => x.Username == username
                                                    && x.Password == password);
    if (user == null) return null; // Return null if user not found

    // Authentication successful so generate jwt token
    var tokenHandler = new JwtSecurityTokenHandler();
    var key = Encoding.ASCII.GetBytes(this.jwtSettings.Secret);
    var tokenDescriptor = new SecurityTokenDescriptor{...};

    var token = tokenHandler.CreateToken(tokenDescriptor);
    user.Token = tokenHandler.WriteToken(token);

    // Return user
}
```

JWT in ASP.NET Core

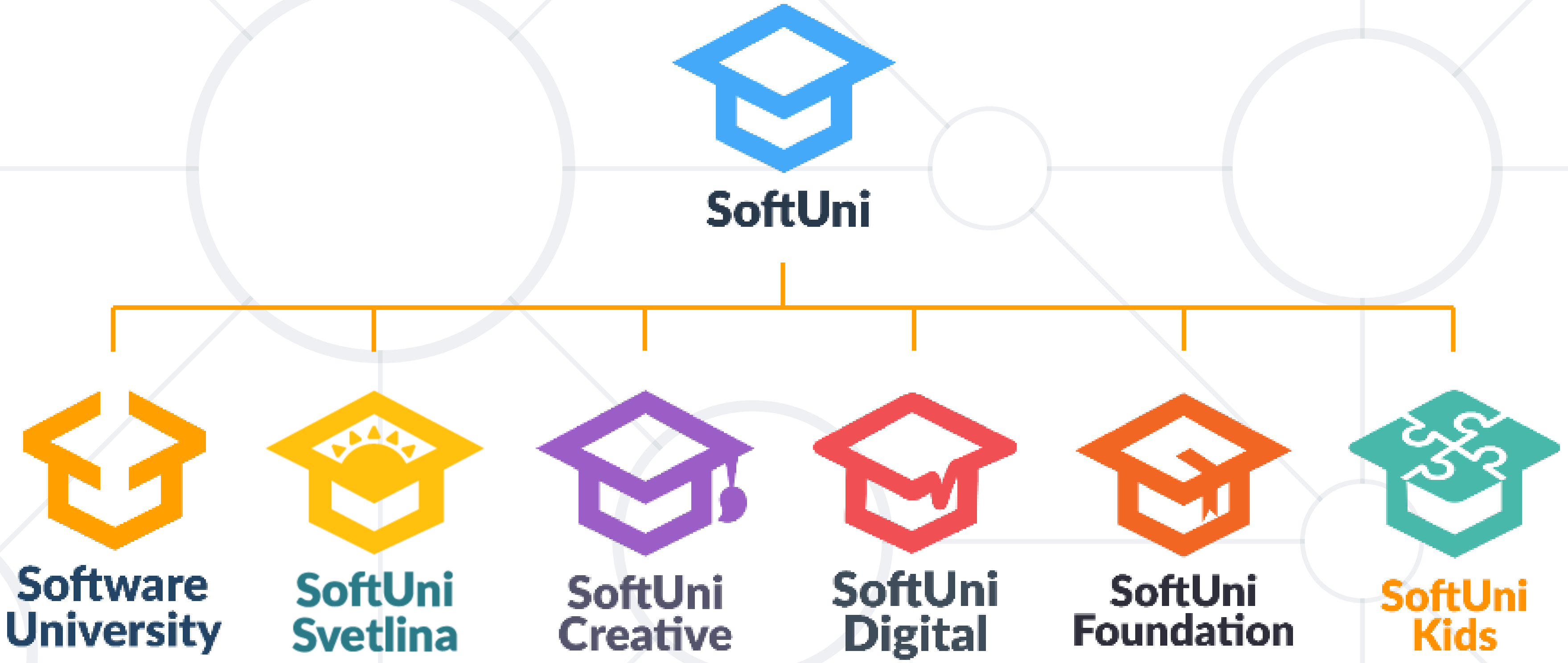
- **JWT** in **ASP.NET Core** is implemented using a middleware

```
public User Authenticate(string username, string password)
{
    ...
    var tokenDescriptor = new SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(new Claim[]
        {
            new Claim(ClaimTypes.Name, user.Username.ToString()),
            new Claim(ClaimTypes.UserIdentifier, user.Id.ToString()),
        }),
        Expires = DateTime.UtcNow.AddDays(7),
        SigningCredentials = new SigningCredentials(
                                new SymmetricSecurityKey(key),
                                SecurityAlgorithms.HmacSha256Signature

        )
    };
    ...
}
```

Summary

- Security in ASP.NET Core
  - Common security problems
  - SQL Injection, XSS, CSRF, Parameter Tampering
- ASP.NET Core Identity
  - Extending & Scaffolding
- Authentication Types
- Social Accounts
- JWT

Questions?


SoftUni
Software University
SoftUni Svetlina
SoftUni Creative
SoftUni Digital
SoftUni Foundation
SoftUni Kids

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License