# .NET Core: Developing Cross-Platform Web Apps with ASP.NET Core – Workshop*PLUS*

Wael Kdouh - @waelkdouh

Senior Consultant

v2.1

## How to View This Presentation

- To switch to **Notes Page** view:
    - On the ribbon, click the **View** tab, and then click **Notes Page**
- To navigate through notes, use the Page Up and Page Down keys
    - Zoom in or zoom out, if required
- In the **Notes Page** view, you can:
    - Read any supporting text
        - Terminology List—a list of terms used in this course is provided in the Notes section.
    - Add notes to your copy of the presentation, if required
- Take the presentation files home with you

# Module 9: Security

## Module Overview

## Module 9: Security

## Section 1: Security Fundamentals

## Lesson: Overview

## What Is Security? How to Think About It?

- **Prevention**
  - Prevent the system from reaching compromised state
  - For example, Secure Development Lifecycle
- **Detection and Recovery**
  - Detect that the system has been compromised and recover it to secure state
  - For example, Intrusion Detection Systems (IDS)
- **Resilience**
  - Ensure minimum functionality in the compromised state
  - For example, redundancy or diversity in physical infrastructure or technology
- **Deterrence**
  - Deter the malicious users/mechanisms from malicious acts
  - For example, Law enforcement, legislations, international collaboration

## Security Principles

- Do not trust anything (including user input)
- Know the weakest link
- Multiple layers of security
- Least privilege
- Secure fallback when things go wrong
- Universally check access permissions
- Minimize shared information
- Do not depend on secrecy
- Keep it simple (KISS)

## Identity

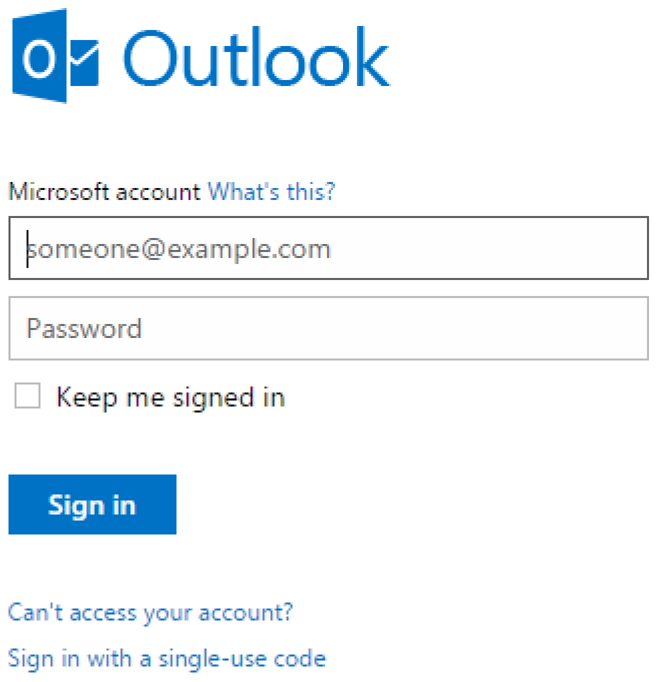
- How do we *represent* a user in our application?
- Typically: A collection of key : value pairs that describe a specific user
  - A pair is referred to as a **claim**
  - The collection of claims makes up an **Identity**
- Represented in code as a model we can create, store, and manipulate
- Can be unique to your app, or shared across apps (Single Sign On)

```
{
  “userID”: “83b6734e”,
  “username”: “SuzyQ”,
  “Name”: “Suzy”,
  “givenName”: “Q”,
  “premiumMember”: true
}
```

```
{
  “userID”: “ba35b637”,
  “username”: “JohnDoe”,
  “Name”: “John”,
  “givenName”: “Doe”,
  “premiumMember”: false
}
```

## Authentication

- Verifying the users are who they say they are



Outlook

Microsoft account What's this?

someone@example.com

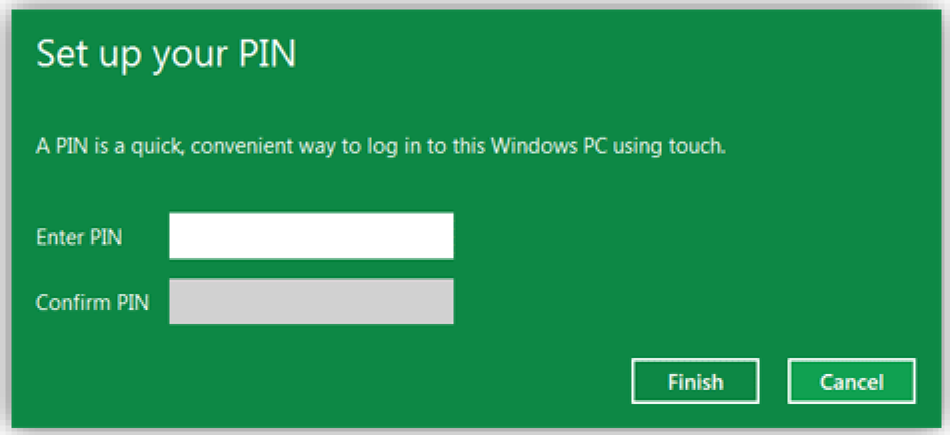
Password

Keep me signed in

Sign in

Can't access your account?

Sign in with a single-use code



Set up your PIN

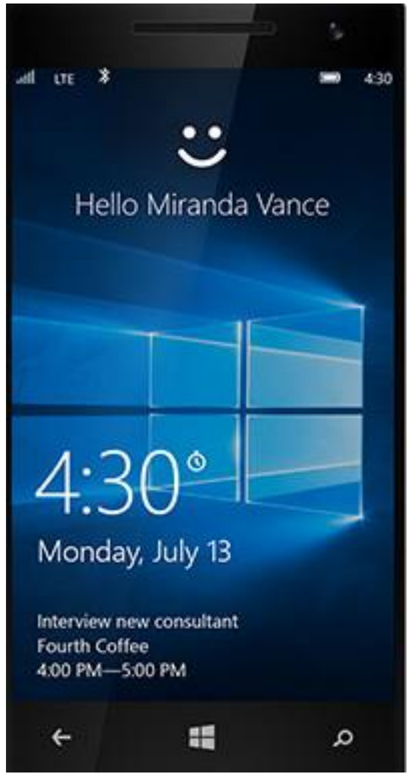
A PIN is a quick, convenient way to log in to this Windows PC using touch.

Enter PIN

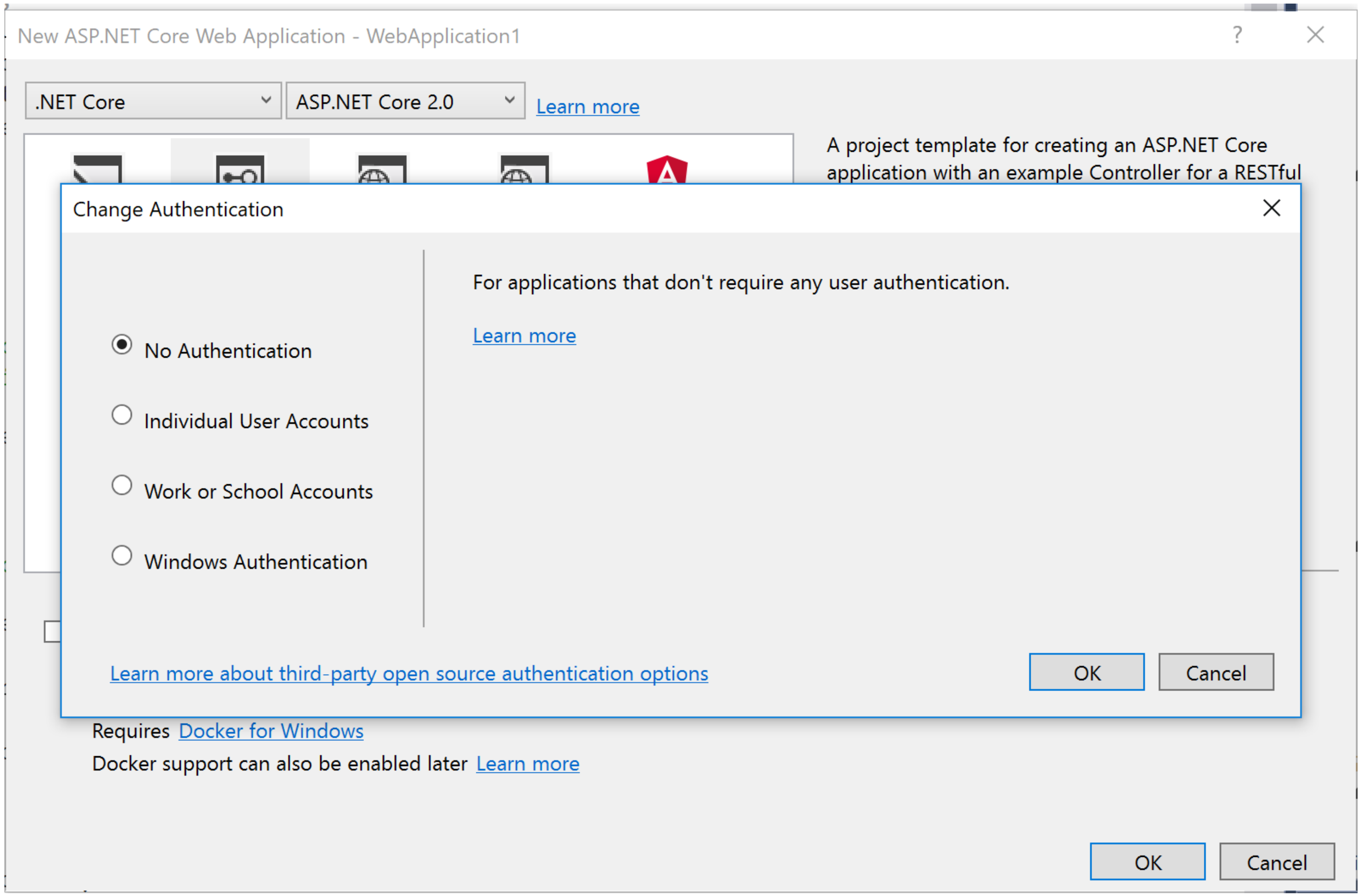
Confirm PIN

Finish

Cancel



# ASP.NET Core Template Authentication Methods



New ASP.NET Core Web Application - WebApplication1

.NET Core | ASP.NET Core 2.0 | Learn more

A project template for creating an ASP.NET Core application with an example Controller for a RESTful

Change Authentication

- [✓ No Authentication]
- Individual User Accounts
- Work or School Accounts
- Windows Authentication

For applications that don't require any user authentication.

Learn more

Learn more about third-party open source authentication options

Requires Docker for Windows

Docker support can also be enabled later Learn more

## ASP.NET Core Template Authentication Methods

- No authentication
- Individual User Accounts
  - Store user accounts in-app (ASP.NET Identity)
  - Connect to an existing user store in the cloud (OpenID compliant Identity Provider)
    - e.g., Azure AD B2C
- Work or School Accounts
  - Active Directory
  - Azure Active Directory
  - Office 365
- Windows Authentication
  - Internet Information Services (IIS) Windows Authentication module

## Authorization

- What can a user *do*?

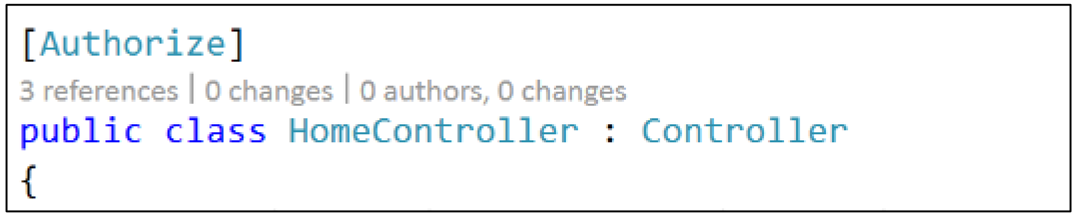
- Many strategies for approaching this important question:
    - Role-Based Authorization
    - Claims-Based Policy Authorization
    - Manual Custom Authorization

```
{
  “userID”: “83b6734e”,
  ...
  “role”: “SysAdmin”,
  “canEditForm”: true,
  “dob”: “1/1/1985”
}
```

```
{
  “userID”: “ba35b637”,
  ...
  “role”: “SDET2”,
  “canEditCode”: true,
  “dob”: “1/1/1970”
}
```

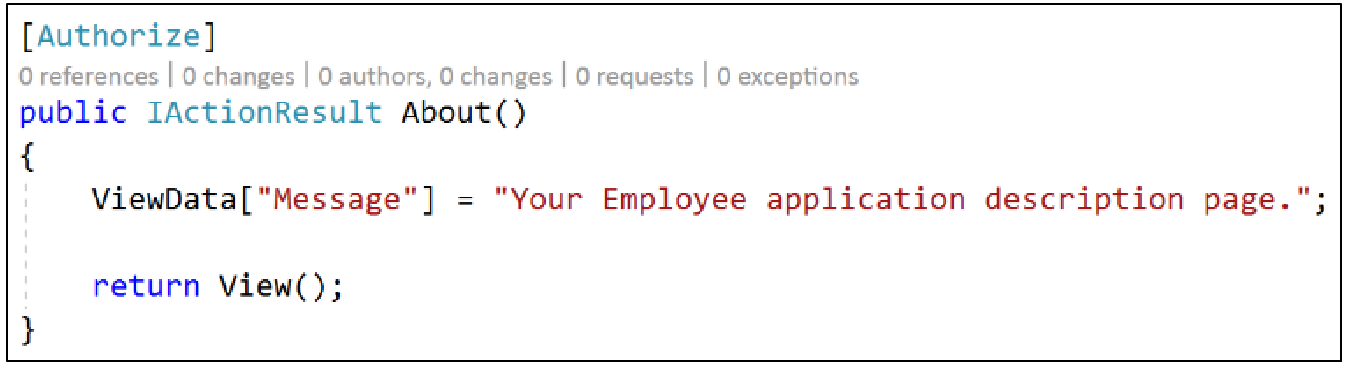
# Authentication with [Authorize] attribute

- `[Authorize]` attribute by itself is used to require an authenticated user
- `[Authorize]` attribute can be used to restrict access to:
  - Specific action methods in a controller
  - Controller ➔ every action method within the controller
- `[Authorize]` should be applied to each controller/action except login/register methods
  - Controller



```
[Authorize]
3 references | 0 changes | 0 authors, 0 changes
public class HomeController : Controller
{
```

  - Action



```
[Authorize]
0 references | 0 changes | 0 authors, 0 changes | 0 requests | 0 exceptions
public IActionResult About()
{
    ViewData["Message"] = "Your Employee application description page.";

    return View();
}
```

## Demo: ASP.NET MVC Authentication

Module 9: Security

Section 2: ASP.NET Identity

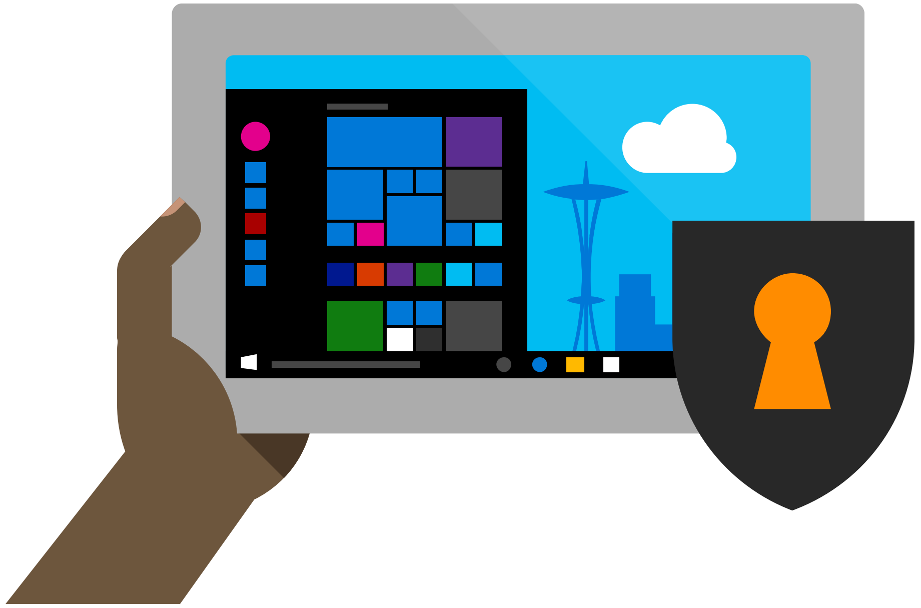
Lesson: Overview

## ASP.NET Identity

Seamless and unified experience for enabling authentication in ASP.NET apps on-premises and in the cloud.

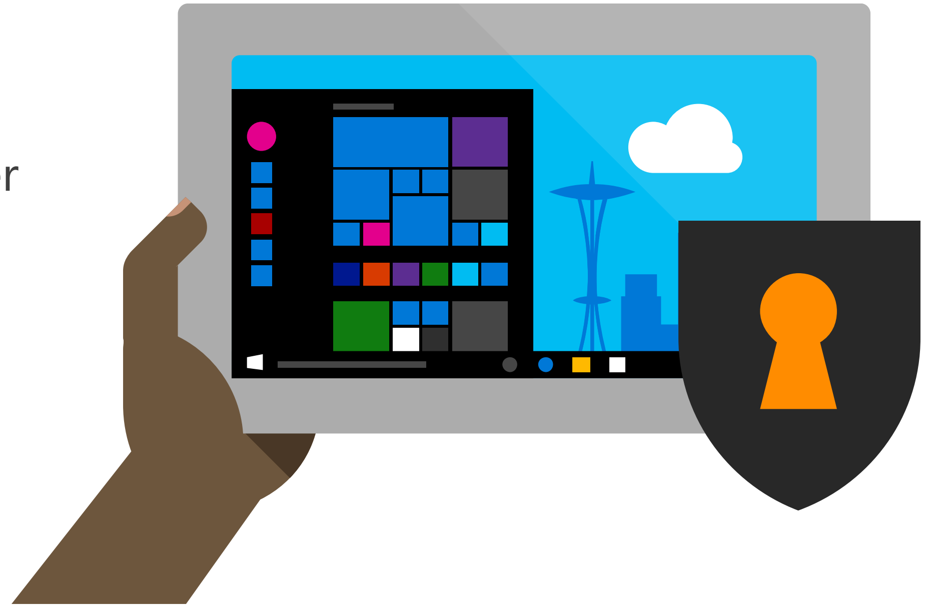
# ASP.NET Identity

- **Easily pluggable user profile**
  - Complete control over the schema of user and profile information
- **Persistence control**
  - SQL Server (Default), Microsoft SharePoint, Azure Storage Table Service, NoSQL databases
- **Role Provider**
  - Role-based authorization
- **Claims-based Authentication**
  - Includes rich information about user's identity



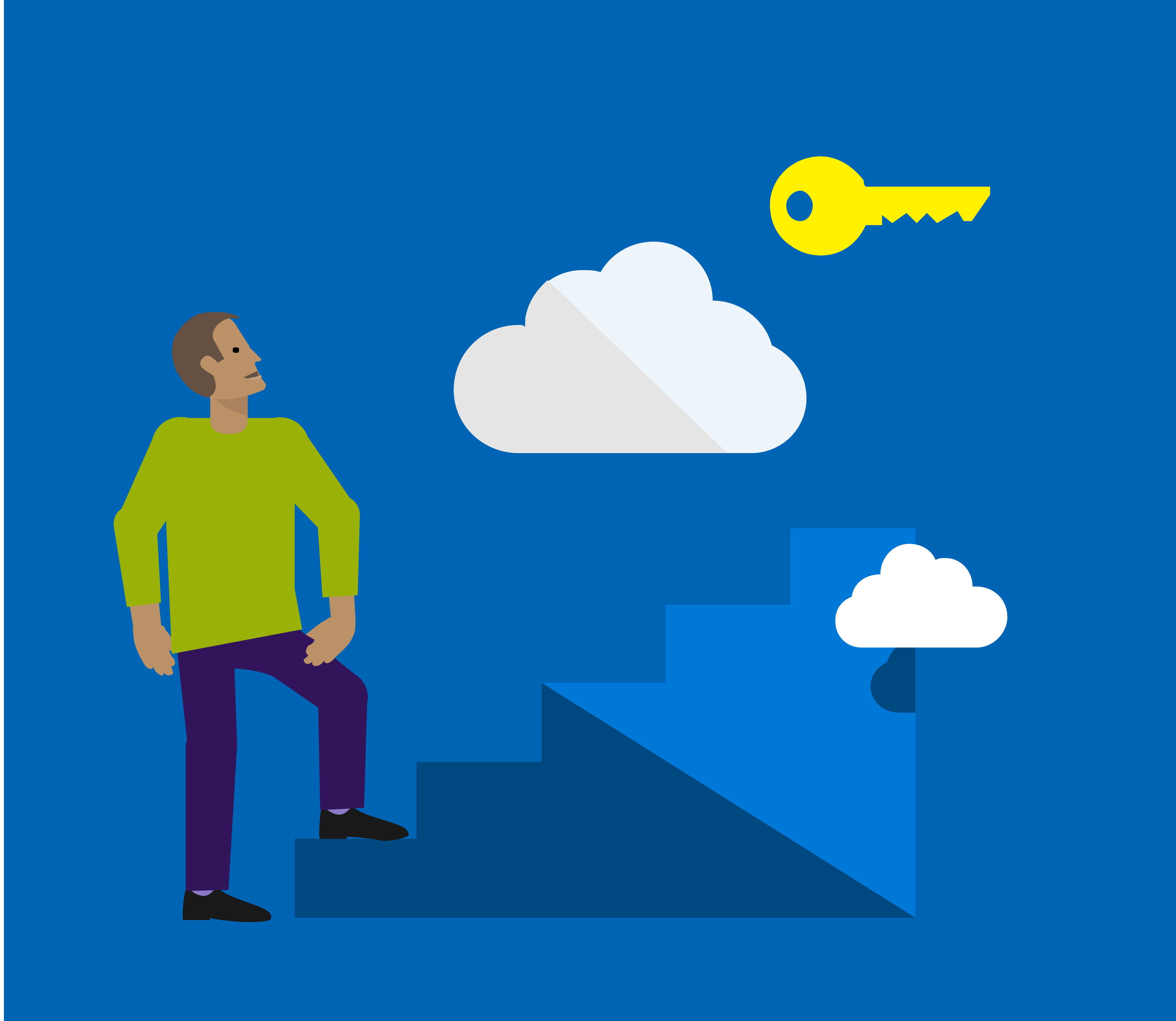
# ASP.NET Identity

- **Unit Testability**
  - Authentication/authorization logic independently testable
- **Social Login Providers**
  - Microsoft account, Facebook, Google, Twitter, and others...
- **Azure AD**
  - Single and multi-organization support
- **Azure AD B2C**
  - Managed OAuth/OpenID compliant Identity provider
- **NuGet package**
  - Agility in release of new features and bug fixes

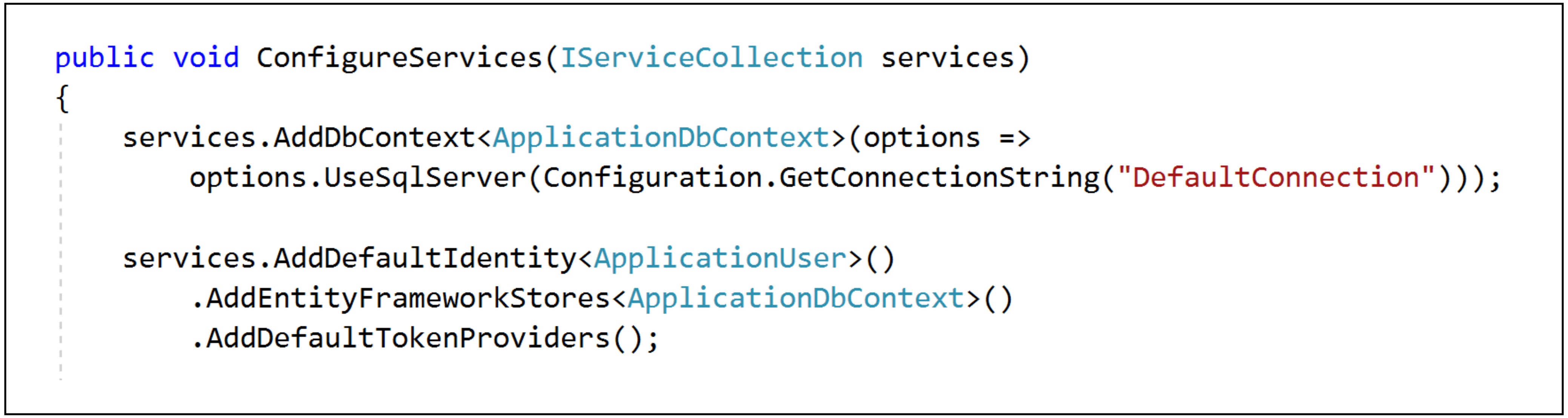


## Features

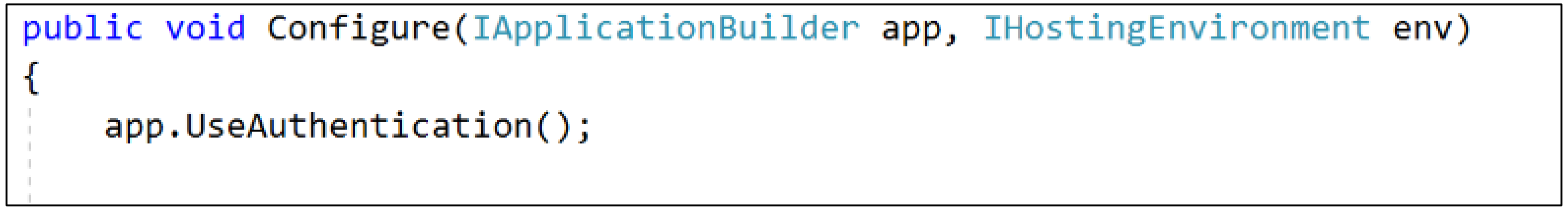
- Two-Factor authentication
- Email/phone verification
- Roles and Claims
- Profile
- User Management
- Role Management
- Password policy enforcement
- User password management
- Account lockout
- Extensibility



# ASP.NET Identity Configuration



```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

    services.AddDefaultIdentity<ApplicationUser>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();
```



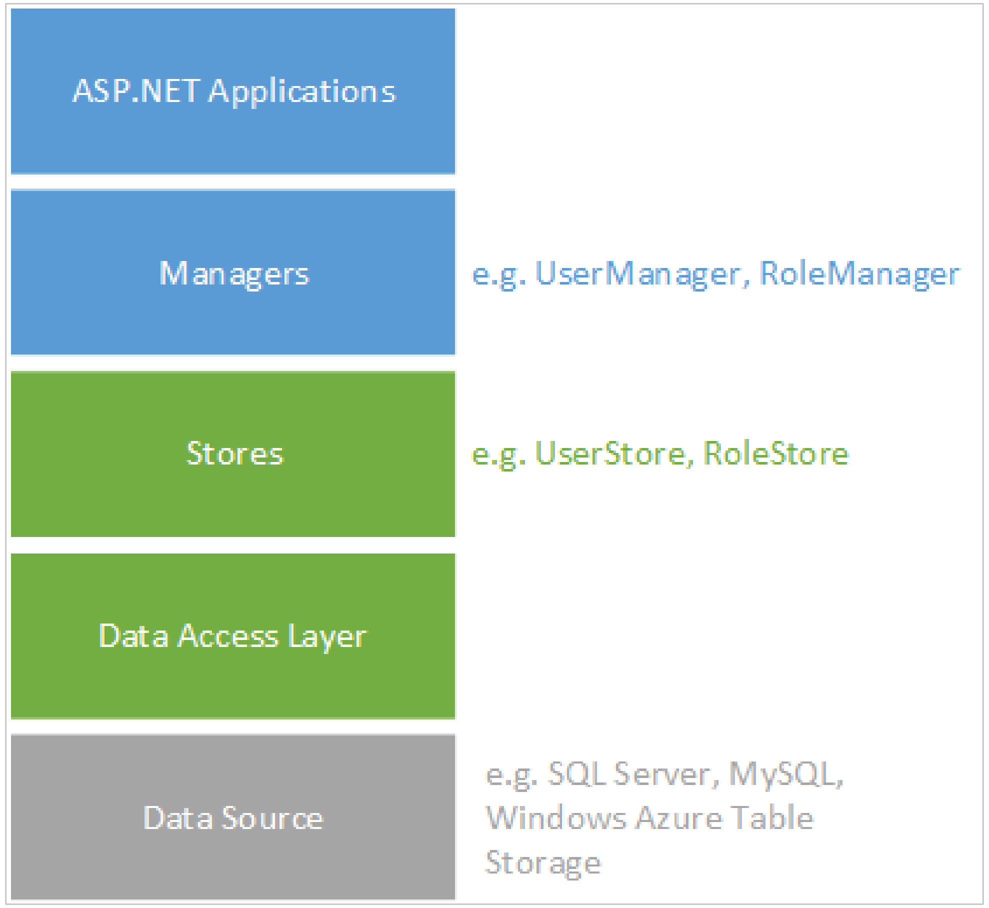
```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseAuthentication();
```

Startup.cs

# ASP.NET Identity Architecture

- **Managers**
  - High-level classes
  - Operations such as create user
  - Completely decoupled from stores

- **Stores**
  - Lower-level classes
  - Closely coupled with the persistent mechanism
  - Store users, roles, claims through Data Access Layer (DAL)


ASP.NET Applications
Managers
e.g. UserManager, RoleManager
Stores
e.g. UserStore, RoleStore
Data Access Layer
Data Source
e.g. SQL Server, MySQL, Windows Azure Table Storage

## ASP.NET Identity Key Classes

- **`IdentityUser`** – Represents web application user
- **`EmailService, SmsService`** – Notified during two-factor authentication
- **`UserManager`** – APIs to CRUD (Create, Read, Update, and Delete) user, claim, and auth information via UserStore
- **`RoleManager`** – APIs to CRUD roles via RoleStore
- **`UserStore`** – Talks to data store to store user, user login providers, user claims, user roles,
    - IUserStore, IUserLoginStore, IUserClaimStore, IUserRoleStore
- **`RoleStore`** – Talks to the data store to store roles
- **`SigninManager`** – High level API to sign in (single or two-factor)

## Module 9: Security

## Section 3: Authorization

## Lesson: Authorization Methodologies

# Roles-Based Authorization

- `[Authorize]` attribute can be used to restrict access to specific users and roles
  - Restricting StoreManagerController to Administrators only

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

  - Restricting controller/action to **any** of multiple roles (logical OR)

```
[Authorize(Roles = "Administrator, SuperAdmin")]
public class StoreManagerController : Controller
```

  - Restricting controller/action to **all** of multiple roles (logical AND)

```
[Authorize(Roles = "Administrator”), Authorize(Roles = “SuperAdmin")]
public class StoreManagerController : Controller
```

  - Restricting controller/action to multiple users & roles

```
[Authorize(Users = “User1, User2", Roles = "SuperAdmin")]
public IActionResult Create(Album album)
```

## Claims-Based Policy Authorization - I

- `[Authorize]` attribute can be used to restrict access to users with specific claims
  - Create a policy for requiring a claim or claim value

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("EmployeeOnly", policy => policy.RequireClaim("EmployeeNumber"));

        options.AddPolicy(“FounderOnly”, policy =>
            policy.RequireClaim(“EmployeeNumber”, “1”, “2”, “3”, “4”, “5”));
    });
}
```

Startup.cs

## Claims-Based Policy Authorization - II

- `[Authorize]` attribute can be used to restrict access to users with specific claims
  - Restricting controller/action to **all** of multiple Policies (logical AND)

```
[Authorize(Policy = “EmployeeOnly”), Authorize(Policy = “FounderOnly")]
public class StoreManagerController : Controller
```

  - Restricting controller/action to any of multiple Policies (logical OR)

```
[Authorize(Policy = “EmployeeOnly, FounderOnly")]
public IActionResult Create(Album album)
```

## Custom Policy Authorization - I

- Implement `IAuthorizationRequirement` as a representation of the requirement
    - Does not need to actually contain any data or logic

```
public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public int MinimumAge { get; private set; }

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

## Custom Policy Authorization - II

- Inherit `AuthorizationHandler<T>` as a way to enact the requirement
  - Override the `HandleRequirementAsync` method

```
public class MinimumAgeHandler : AuthorizationHandler<MinimumAgeRequirement>
{
    protected override Task HandleRequirementAsync(AuthorizationHandlerContext context,
        MinimumAgeRequirement requirement)
    {
         if (!context.User.HasClaim(c => c.Type == ClaimTypes.DateOfBirth))
         {
              return Task.CompletedTask;
         }
         var dateOfBirth = Convert.ToDateTime(context.User.FindFirst(c =>
                                                c.Type == ClaimTypes.DateOfBirth).Value);

         // Calculate Age and determine if >= payload of MinimumAgeRequirement
         // Return context.Succeed(requirement); if true!
}
```

## Custom Policy Authorization - III

- Register the Authorization Handler in the IoC container
  - Add a policy to the Policy collection

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    services.AddAuthorization(options =>
    {
        options.AddPolicy(“Over21", policy =>
            policy.Requirements.Add(new MinimumAgeRequirement(21)));
    });

    services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
}
```

Startup.cs

## Custom Policy Authorization - IV

- `[Authorize]` attribute can be used to restrict access to users that pass custom policies
  - Restricting controller/action to a custom policy (logical AND)

```
[Authorize(Policy = “Over21”)]
public class StoreManagerController : Controller
```

## Demo: ASP.NET Core Identity

## Module 9: Security

## Section 4: OIDC (OpenID Connect) and OAuth 2.0

## Lesson: Overview

## OIDC (OpenID Connect) and OAuth 2.0

**OIDC – Authentication protocol**

**OAuth 2.0 – Authorization Protocol**

## OpenID Connect and OAuth 2.0

- **OAuth 2.0 is purely for authorization, not authentication**
  - OAuth 2.0 does not tell the client who the user is
  - Person granting access might not be the real user (resource owner)
  - Does not have a notion of an "identity“
  - Access Token contains claims about the delegated access rights

- **OpenID Connect builds on OAuth 2.0 and adds authentication information**
  - OIDC add user identity request to OAuth 2.0 request
  - ID Token: JWT with at least a “sub” claim to identify the end user (“subject”)
  - UserInfo Endpoint: returns more claims about the end user (JSON/JWT)
  - OIDC is pure authentication protocol if access token is not requested

## Terminology

### Client

- Application that needs to use the resource
- Various types –
  - browser-Web-App, Native, Daemons, etc.
- Often end-user facing
- E.g. Snapfish “Print shop” application

### Resource Server

- Hosts the resource
- Typically an API provider
  - E.g., Microsoft Graph API
- Trusts tokens from an Authorization Server
- E.g. OneDrive “Photo library”

### Resource Owner

- Owner of the requested resource
- Typically the user of the application

  E.g. “Owner of the OneDrive account/photos”

### Authorization Server

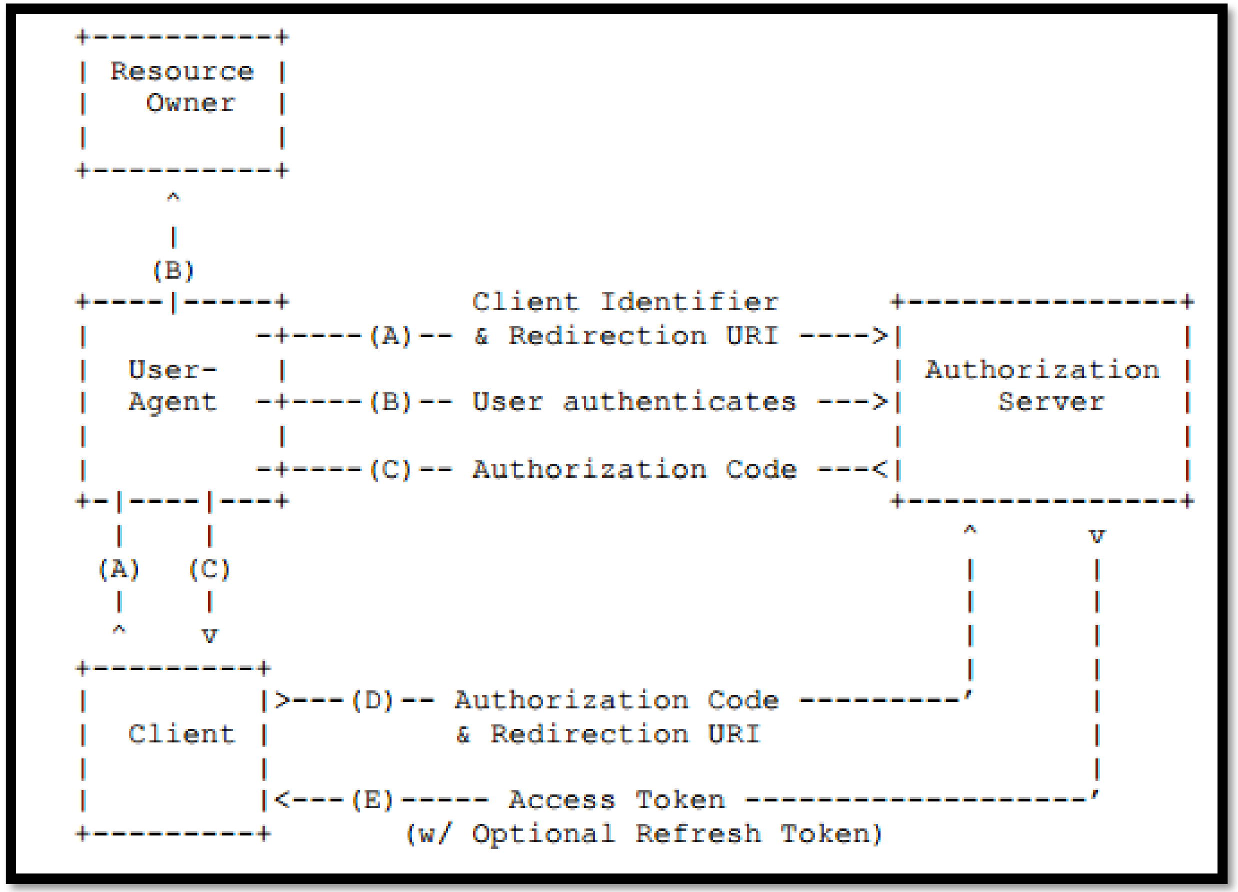
- Issues access tokens to clients
- Authenticates resource owners
- Gets access consent from the resource owner
- Could be “Photo library provider”

## OAuth 2.0 Flows

- Authorization Code flow
- Implicit Grant flow
- Client Credentials flow
- Resource Owner Password Credentials flow

## Authorization Code Flow

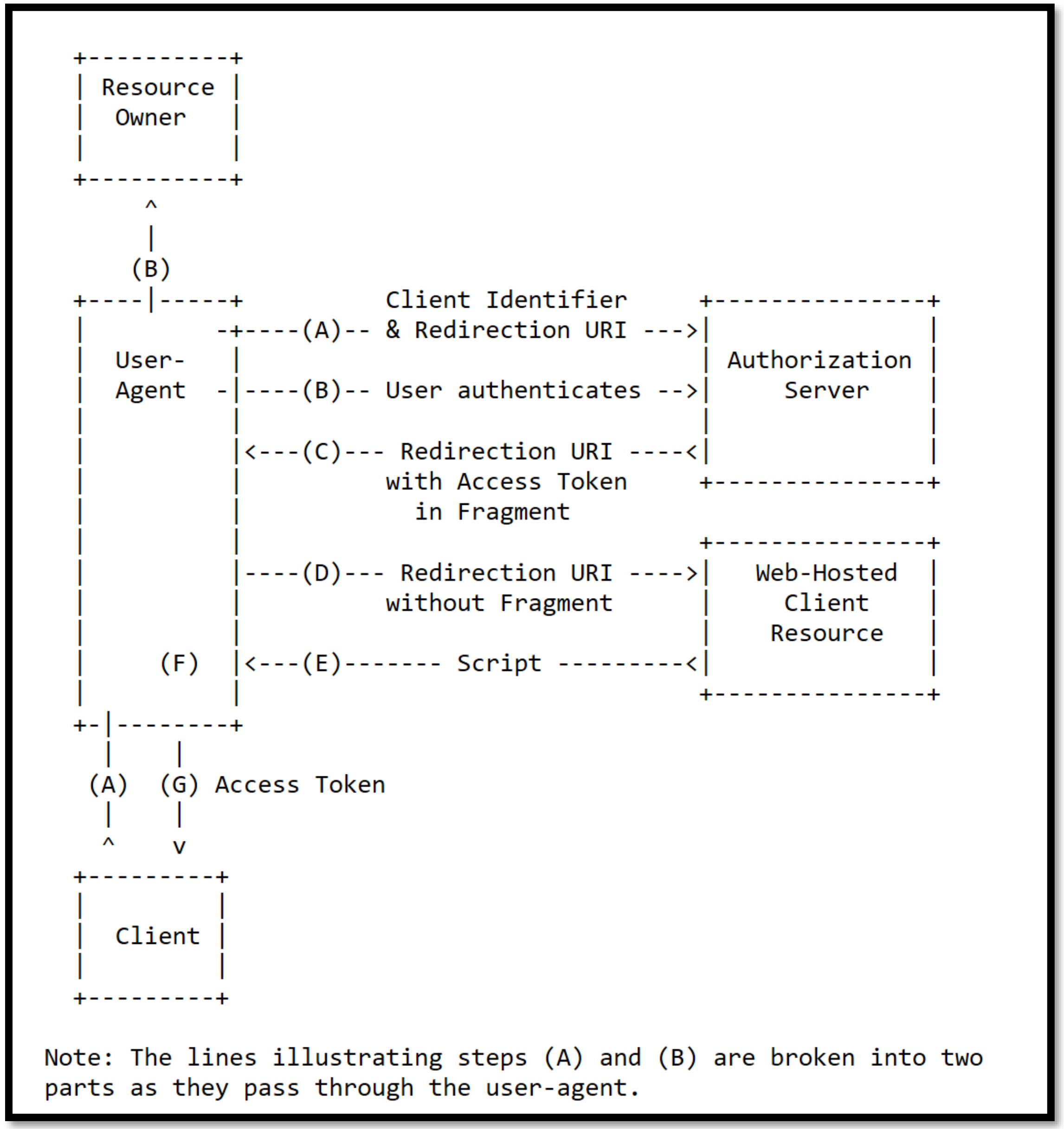
- Obtain both access tokens and refresh tokens
- Minimizes token exposure
- Client must be capable of interacting with the resource owner's user-agent and capable of receiving incoming requests (via redirection) from the authorization server.



```
+----------+
| Resource |
|   Owner  |
|          |
+----------+
     ^
     |
    (B)
+----|-----+          Client Identifier      +---------------+
|         -+----(A)-- & Redirection URI ---->|               |
|  User-   |                                 | Authorization |
|  Agent  -+----(B)-- User authenticates --->|     Server    |
|          |                                 |               |
|         -+----(C)-- Authorization Code ---<|               |
+-|----|---+                                 +---------------+
  |    |                                         ^      v
 (A)  (C)                                        |      |
  |    |                                         |      |
  ^    v                                         |      |
+---------+                                      |      |
|         |>---(D)-- Authorization Code ---------'      |
|  Client |          & Redirection URI                  |
|         |                                             |
|         |<---(E)----- Access Token -------------------'
+---------+       (w/ Optional Refresh Token)
```

# Implicit Grant Flow

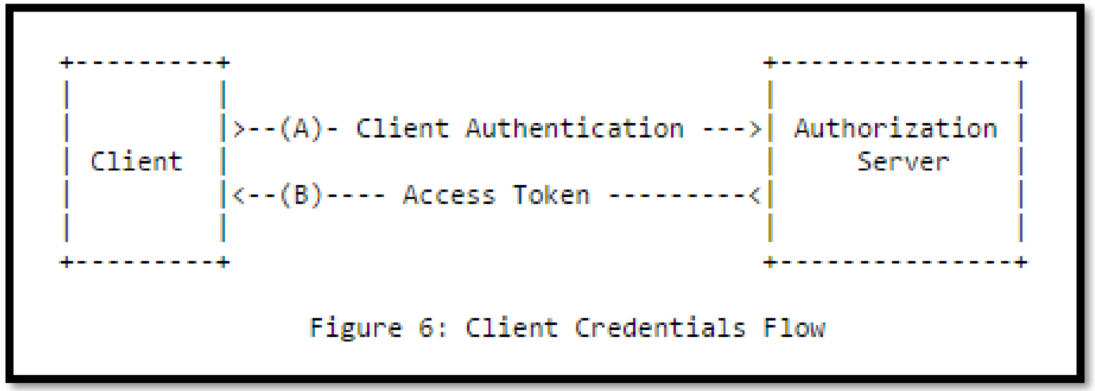
- For browser-based clients, e.g. SPA.
- Browser code is the client needs to control server re-direction to avoid losing state



```
 +----------+
 | Resource |
 |  Owner   |
 |          |
 +----------+
      ^
      |
     (B)
 +----|-----+          Client Identifier     +---------------+
 |         -+----(A)-- & Redirection URI --->|               |
 |  User-   |                                | Authorization |
 |  Agent  -|----(B)-- User authenticates -->|     Server    |
 |          |                                |               |
 |          |<---(C)--- Redirection URI ----<|               |
 |          |          with Access Token     +---------------+
 |          |            in Fragment
 |          |                                +---------------+
 |          |----(D)--- Redirection URI ---->|   Web-Hosted  |
 |          |          without Fragment      |     Client    |
 |          |                                |    Resource   |
 |     (F)  |<---(E)------- Script ---------<|               |
 |          |                                +---------------+
 +-|--------+
   |    |
  (A)  (G) Access Token
   |    |
   ^    v
 +---------+
 |         |
 |  Client |
 |         |
 +---------+

Note: The lines illustrating steps (A) and (B) are broken into two
parts as they pass through the user-agent.
```

## Client Credentials Flow

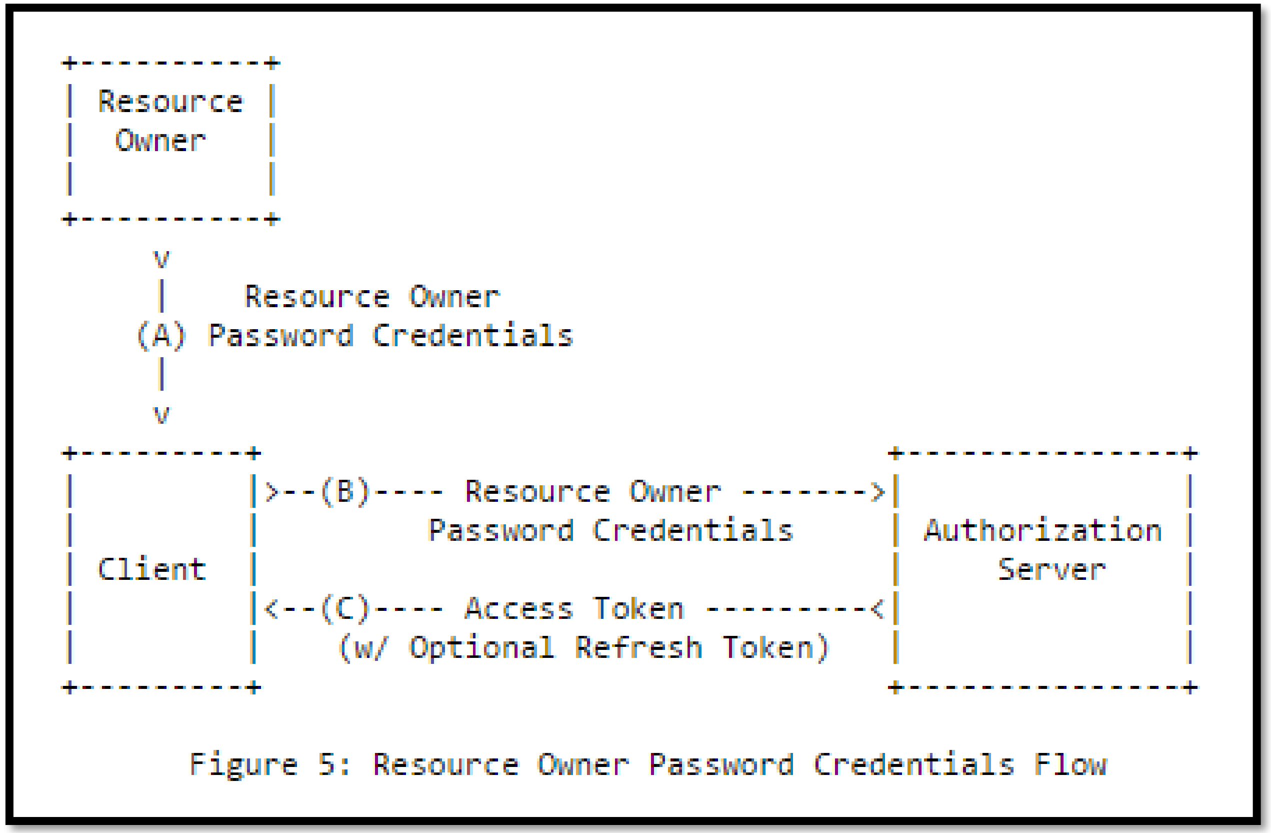
- Confidential Clients Only
- No human involved (batch service)
- Should be used by confidential clients only



```
+---------+                                  +---------------+
|         |                                  |               |
|         |>--(A)- Client Authentication --->| Authorization |
| Client  |                                  |     Server    |
|         |<--(B)---- Access Token ---------<|               |
|         |                                  |               |
+---------+                                  +---------------+

                Figure 6: Client Credentials Flow
```

# Resource Owner Password Credentials Flow

- Client obtains Resource Owner's Username and Password
- Also useful to migrate from other protocols, like Basic Authentication
- Should be used as last resort – considered an anti-pattern for user authentication



```
     +----------+
     | Resource |
     |  Owner   |
     |          |
     +----------+
          v
          |    Resource Owner
         (A) Password Credentials
          |
          v
     +---------+                                  +---------------+
     |         |>--(B)---- Resource Owner ------->|               |
     |         |         Password Credentials     | Authorization |
     | Client  |                                  |     Server    |
     |         |<--(C)---- Access Token ---------<|               |
     |         |    (w/ Optional Refresh Token)   |               |
     +---------+                                  +---------------+
```

Figure 5: Resource Owner Password Credentials Flow

## OIDC Flows

| | Authorization Code | Implicit Grant | Hybrid |
|---|---|---|---|
| All tokens returned from Authorization Endpoint | No | Yes | No |
| All tokens returned from Token Endpoint | Yes | No | No |
| Tokens sent via user agent | No | Yes | No |
| Clients can be authenticated (e.g. using client secret) | Yes | No | Yes |
| Can use refresh tokens | Yes | No | Yes |
| Communication in one round trip | No | Yes | No |
| Most communication server-to-server | Yes | No | |

## OpenID Connect - Hybrid Flow

- Hybrid Flow is a combination of Authorization Code Flow and Implicit Grant
- Allows immediate use of an identity token and optionally retrieve an authorization code via one round trip to the STS
- Confidential Clients Only
- Can obtain an authorization code and tokens from the authorization endpoint and can also request tokens from the token endpoint.

# Tokens

## Access Token and ID Token

- OIDC

## Access Token

- OAuth 2.0

## Refresh Token

- Can be obtained by both OIDC and OAuth 2.0 protocols

## Endpoints

- **Authorize**
  - Use to identity a user to obtain an authorization code
  - Later exchange for an Access Token

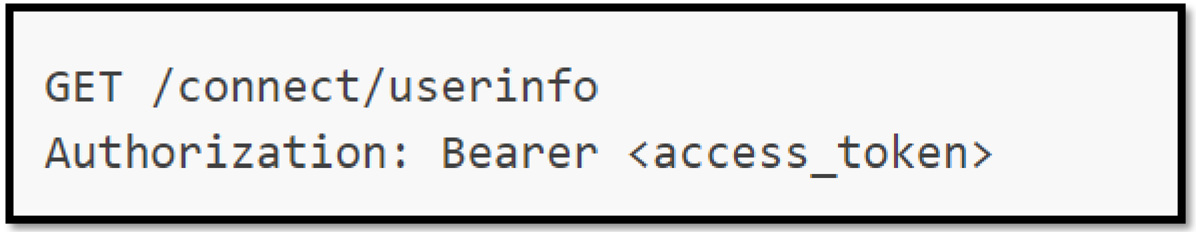
```
GET /connect/authorize?
    client_id=client1&
    scope=openid email api1&
    response_type=id_token token&
    redirect_uri=https://myapp/callback&
    state=abc&
    nonce=xyz
```

- **Token**
  - Use this endpoint to access token
  - Supports password, authorization code,
    client credentials and refresh tokens grant types

```
POST /connect/token

    client_id=client1&
    client_secret=secret&
    grant_type=authorization_code&
    code=hdh922&
    redirect_uri=https://myapp.com/callback
```

## Endpoints

- **UserInfo**
  - Can be used to retrieve identity information about a user
  - Caller needs to provide the valid access token



```
GET /connect/userinfo
Authorization: Bearer <access_token>
```

- **Discovery**
  - To retrieve metadata about Identity Server
  - Provide information like issuer name, key material, supported scopes, etc.

    E.g. https://contoso.com/.wellknown/openid-configuration

## Demo: Integrate Azure AD into ASP.NET Core App using OpenID Connect middleware

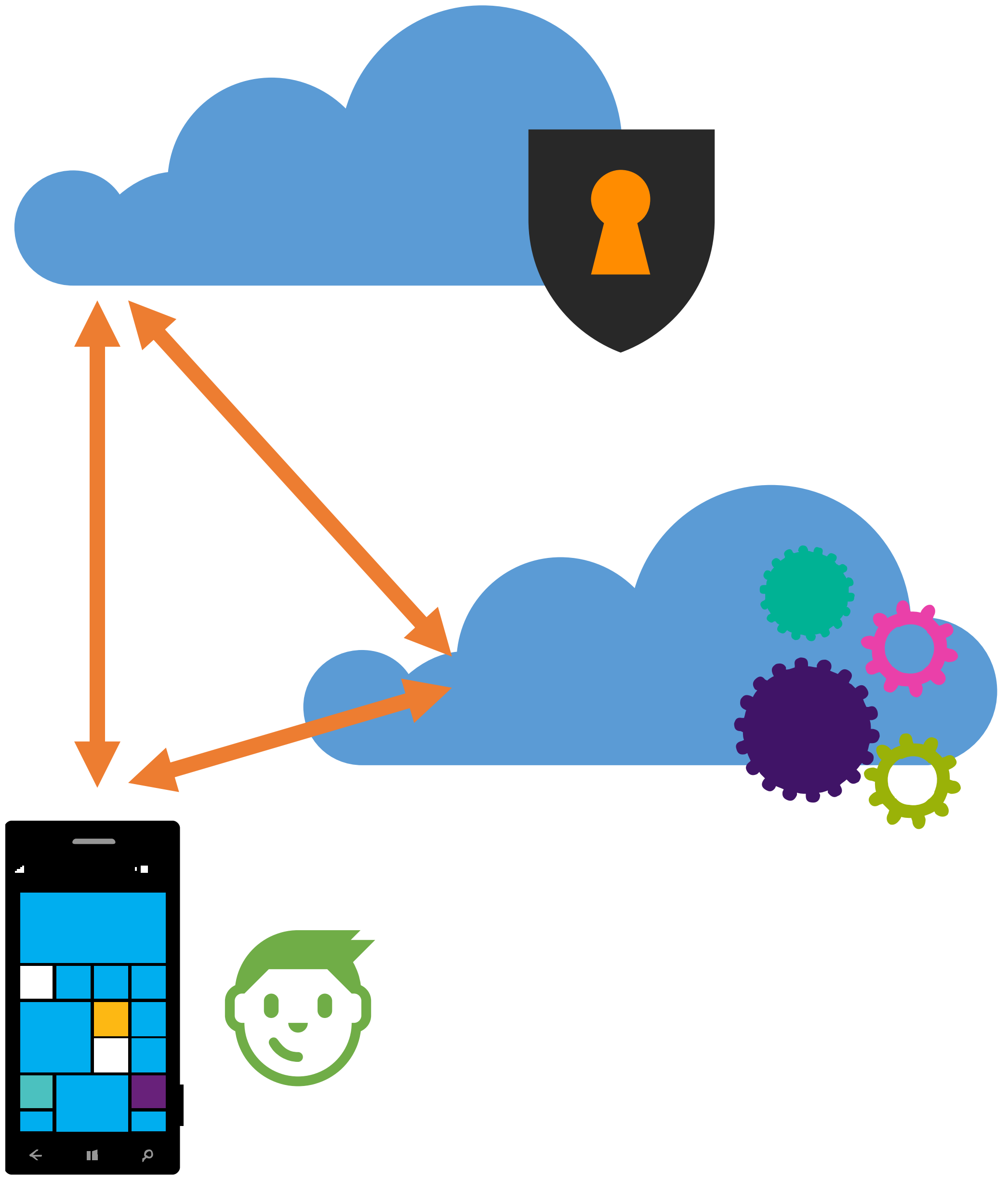
## Module 9: Security

## Section 5: External Identity Providers

## Lesson: Identity as a Service

## Identity Broker Pattern

- A very powerful pattern for achieving Single Sign On (SSO) across all of your applications
- This pattern is used by Social Identity Providers like Google, Facebook, Microsoft, etc.
- OpenID and OAuth are examples of this pattern
- Azure AD and Azure AD B2C are both OpenID/OAuth compliant, managed Identity Providers



## Identity Broker Pattern – Trusted Party

- The Trusted party or Identity Provider is the source of truth for user Identities
- A separate service from your applications
- Can be hosted/managed or custom made

- Allow Single Sign On (SSO)
- Allows Identity to be "as a service"

- "Sign in with…"
    - Microsoft Account
    - Work or School Account
    - Facebook
    - Google
    - Etc.

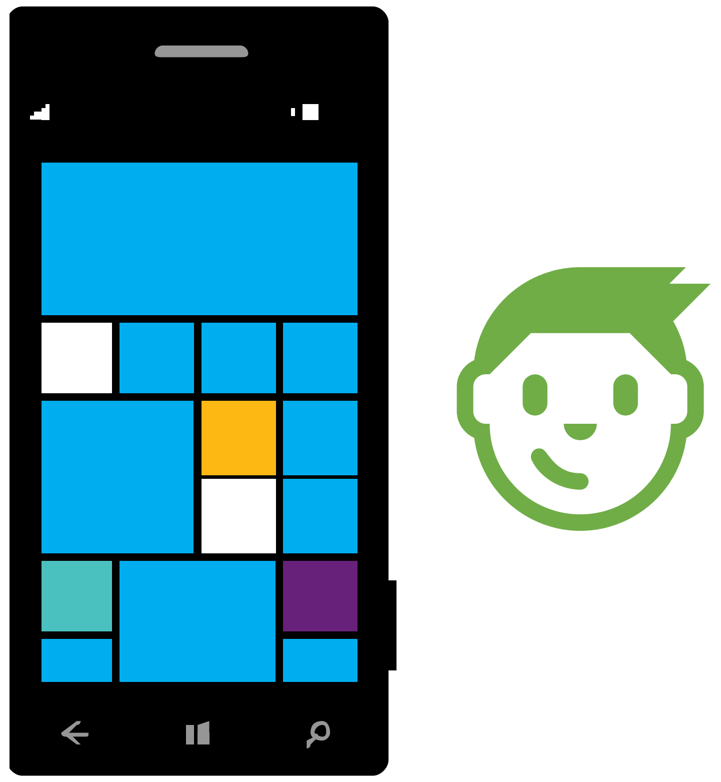
## Identity Broker Pattern – Reliant Party

- Your applications *rely* on the identity provider to verify user identities
- Applications need to be registered with the Identity Provider in order to be reliant
- Every application is uniquely identified by a Client ID or Application ID
- Every application is verified via a public/private or shared key

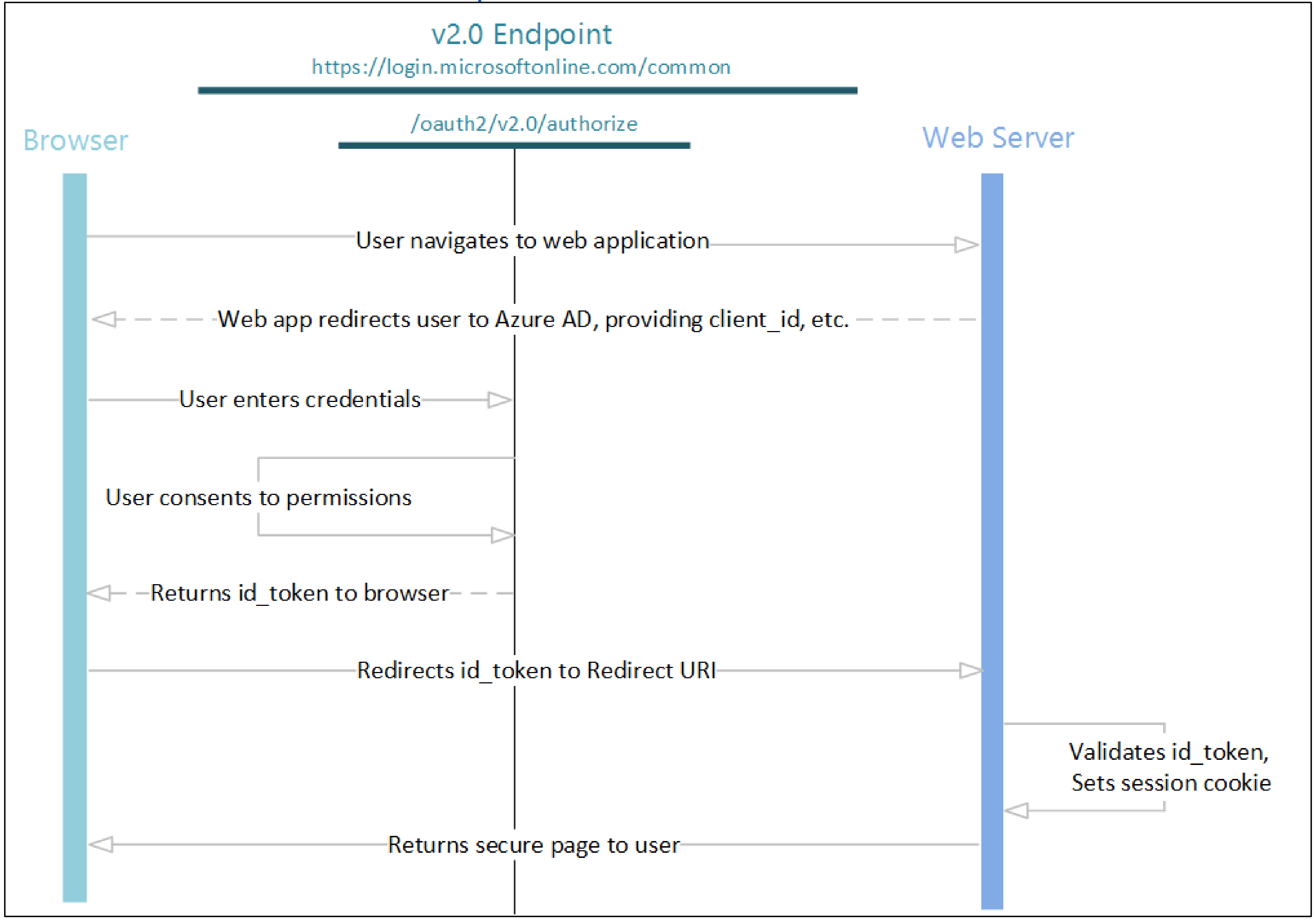
- Redirect authentication flows to the Identity Provider

# Identity Broker Pattern – User

- Users register with the Identity Provider (Trusted Party)
- Attempts to access Reliant Party applications redirects the user to the Identity Provider for authentication
- Once a user has a proof of Authentication, it can be used for all Reliant Party applications the user is authorized for
    - This creates Single Sign On!



# Identity Broker Pattern – OpenID

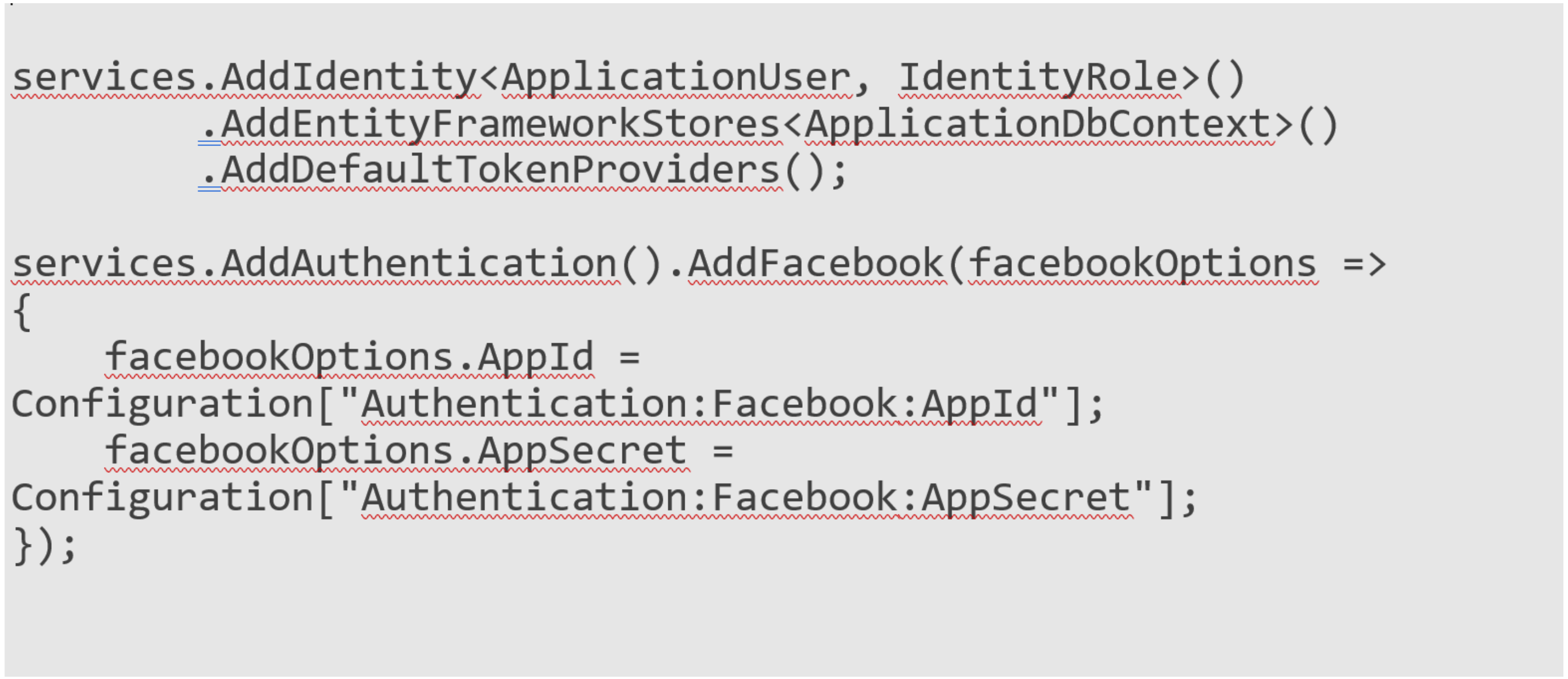

v2.0 Endpoint
https://login.microsoftonline.com/common
/oauth2/v2.0/authorize
Browser
Web Server
User navigates to web application
Web app redirects user to Azure AD, providing client_id, etc.
User enters credentials
User consents to permissions
Returns id_token to browser
Redirects id_token to Redirect URI
Validates id_token,
Sets session cookie
Returns secure page to user

## Authentication with External Providers

- External providers
  - Facebook, Twitter, Microsoft, Google, etc.
- Configuration
  - Application ID
  - Application Secret
  - Website URL
- Storage of App Secret
  - **Do not** store in config file
  - [Best Practice] Secret Manger
  - [Best Practice] Application Settings in Azure

# Authentication with Facebook

- One of the external IdP can be Facebook. Use this guide to follow steps
- Register App in Facebook
- Install Microsoft.AspNetCore.Authentication.Facebook Nuget Package
-                 dotnet add package Microsoft.AspNetCore.Authentication.Facebook
- Modify Startup.cs ConfigureServices method:



```
services.AddIdentity<ApplicationUser, IdentityRole>()
        .AddEntityFrameworkStores<ApplicationDbContext>()
        .AddDefaultTokenProviders();

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

## Demo: Authentication Using External Provider

## Module 9: Security

## Section 6: ASP.NET Identity Strategies

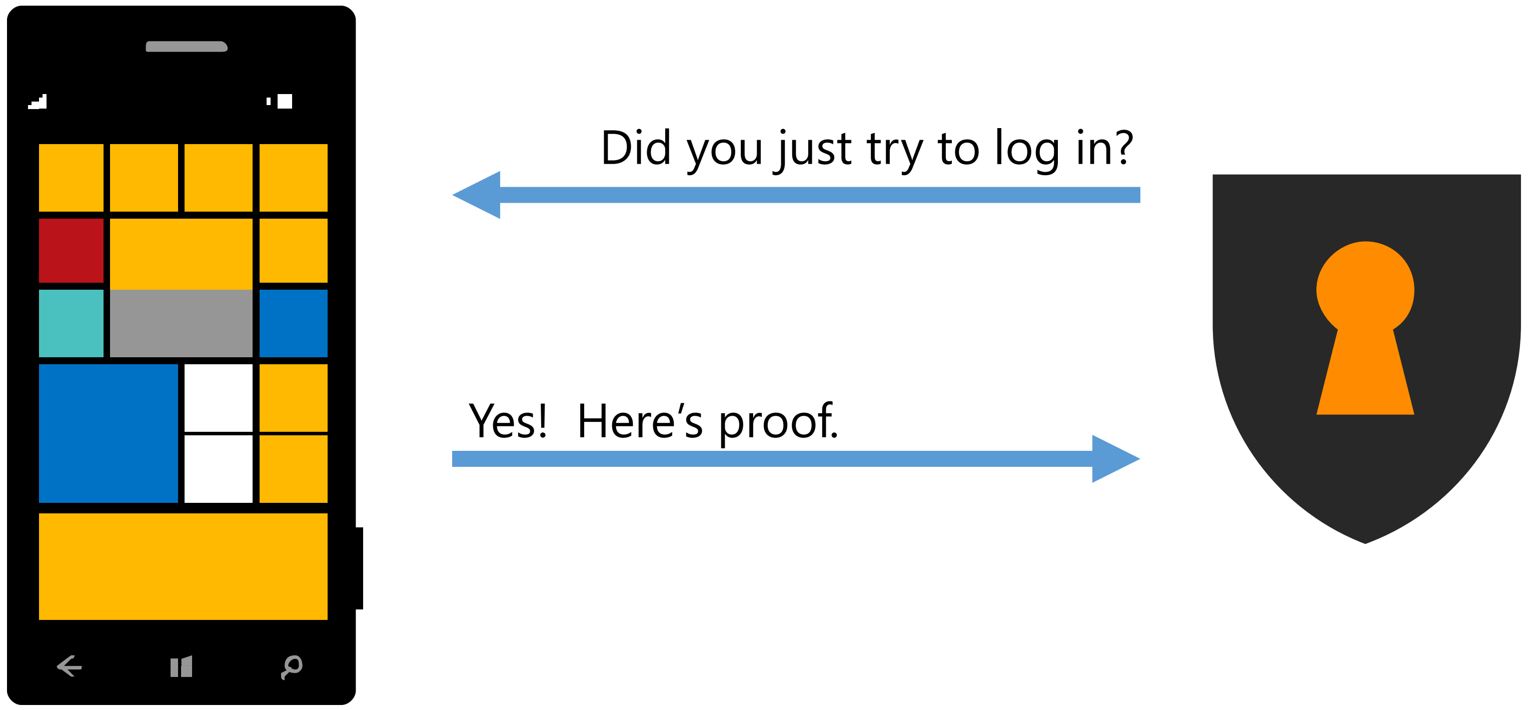
## Lesson: ASP.NET Identity Strategies

## Recommendations

- Utilize Secure Sockets Layer (TLS/SSL - HTTPS) everywhere
    - Attacker on network can steal your cookies and hijack your session
    - Yes, even login page needs to be protected
    - Any page user can access while logged in should be protected
- Enforce a strong password policy (more an art than a science)
- Use Cross-Site Request Forgery (CSRF) tokens everywhere for post methods
- Do not allow unlimited login attempts
    - Brute forcers dream. Script kiddies abound.

## Recommendations (continued)

- **If** security requirements demand it, you can change password hashing method
- Consider shortening OnValidateIdentity times to expire sessions
- Two-Factor authentication is highly recommended for enhanced security



## Note that…

- Password expiration is not built-in
    - It is not right for every system, a good policy but consider it carefully
- Identity is not multi-tenant or multi-app by default
    - Use Azure AD or add Tenant IDs to users for multi-tenancy
    - Put Identity in a separate SQL server to share across apps (*not* true SSO)

## Module 9: Security

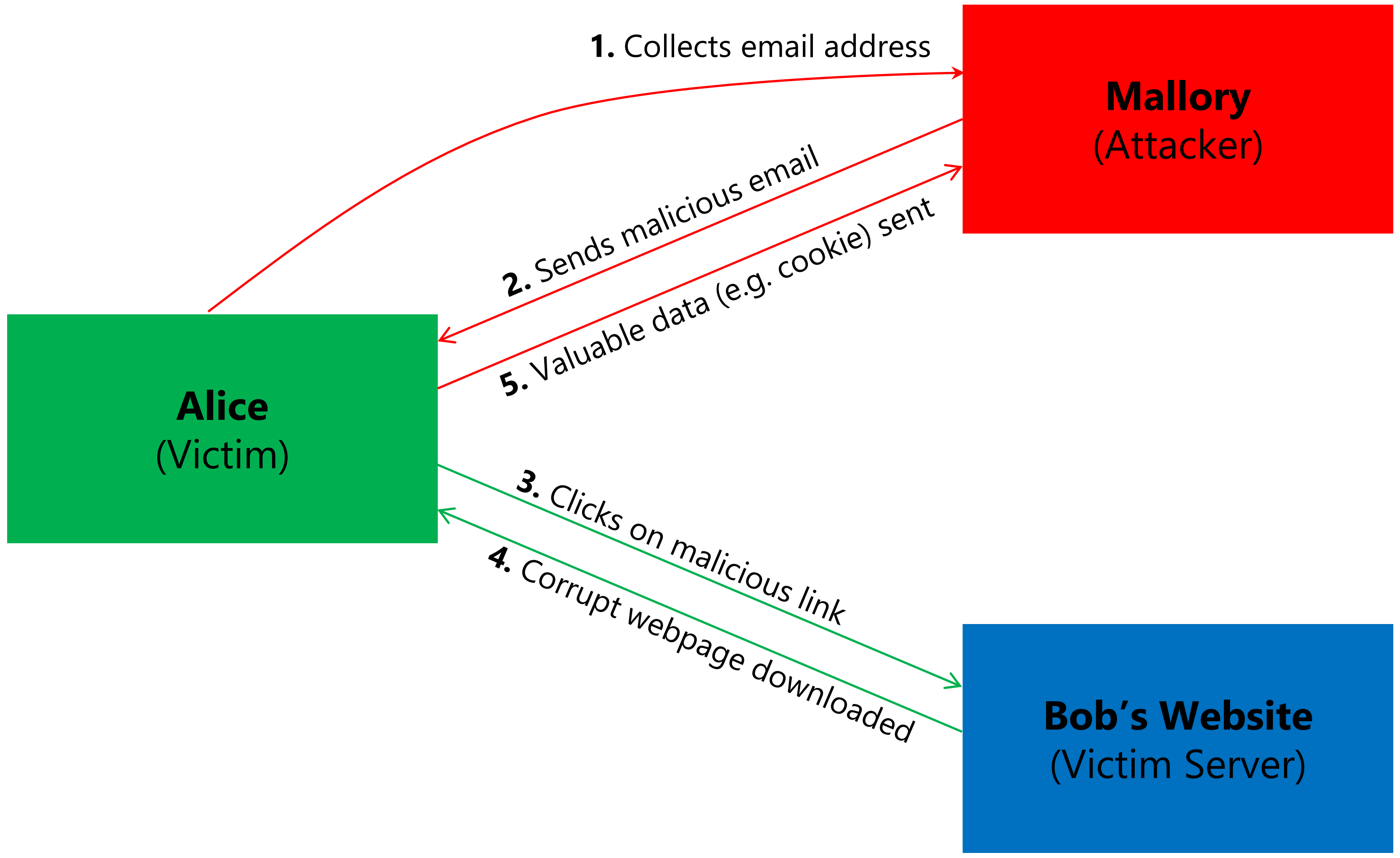
## Section 7: Security Threats and Defenses

## Lesson: Web Attacks and Defenses

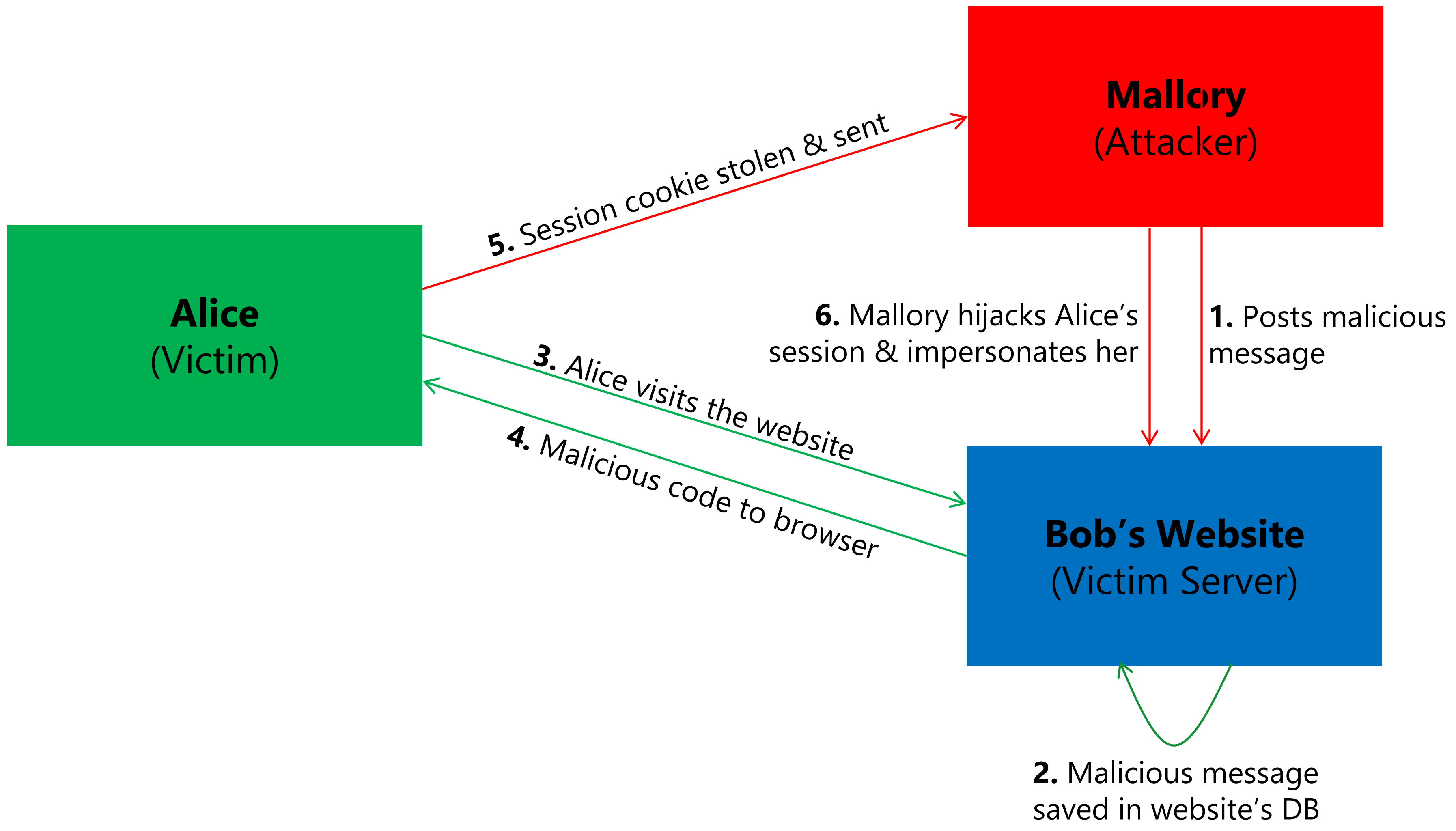
## Cross-Site Scripting (XSS) Attack

- XSS vulnerability allows an attacker to inject malicious JavaScript into pages generated by a web application
- Malicious script executes in victim client's browser
  - To gain access to sensitive webpage content, session cookies, etc.
- Methods for injecting malicious code:
  - **Active or Reflected Injection**
    - Attack script directly reflected back to the user from the victim site
    - Victim user participates directly in the attack
    - Often done through social engineering tricks, such as malicious email
  - **Passive or Stored Injection**
    - Malicious code is saved in the backend database using user input
    - Potentially more dangerous because all users of the web application may be compromised

# XSS Reflected Attack


**1.** Collects email address
**Mallory**
(Attacker)
**2.** Sends malicious email
**5.** Valuable data (e.g. cookie) sent
**Alice**
(Victim)
**3.** Clicks on malicious link
**4.** Corrupt webpage downloaded
**Bob's Website**
(Victim Server)

## XSS Stored Attack


**Mallory**
(Attacker)
**Alice**
(Victim)
**Bob's Website**
(Victim Server)
**1.** Posts malicious message
**2.** Malicious message saved in website's DB
**3.** Alice visits the website
**4.** Malicious code to browser
**5.** Session cookie stolen & sent
**6.** Mallory hijacks Alice's session & impersonates her

## XSS Defense

- Never trust any input to your website
- Ensure that your app validates all user input, form values, query strings, cookies, information received from third-party sources, for example, OpenID
- Use whitelist approach instead of trying to imagine all possible hacks
    - It is not possible to know all permutations
- Remove/encode special characters
    - HTML encoding
    - JavaScript encoding

## HTML Encoding

- All output on your pages should be HTML-encoded or HTML-attribute-encoded
  - `@Html.Encode(Model.FirstName)`
  - `@Model.FirstName`
- URL Encoding:
  - `@Url.Encode(Url.Action(“index”, “home”, new {name=ViewData[“name”]}))`
- Razor View Engine automatically HTML-encodes output

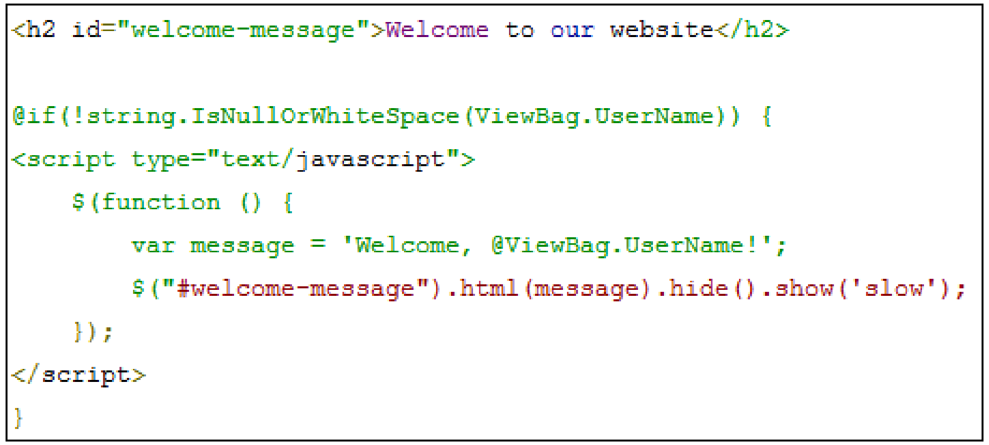
Malicious User Input (without encoding)

<script>alert(“XSS!”)</script>

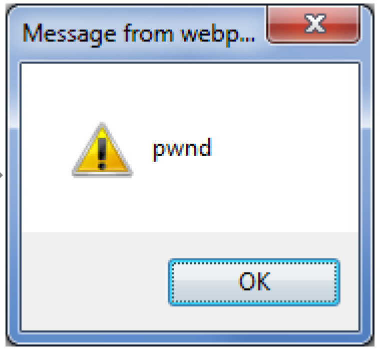
HTML-Encoded User Input

&lt;script&gt;alert(‘XSS!’)&lt;/script&gt;

# JavaScript Encoding



```
<h2 id="welcome-message">Welcome to our website</h2>

@if(!string.IsNullOrWhiteSpace(ViewBag.UserName)) {
<script type="text/javascript">
    $(function () {
        var message = 'Welcome, @ViewBag.UserName!';
        $("#welcome-message").html(message).hide().show('slow');
    });
</script>
}
```

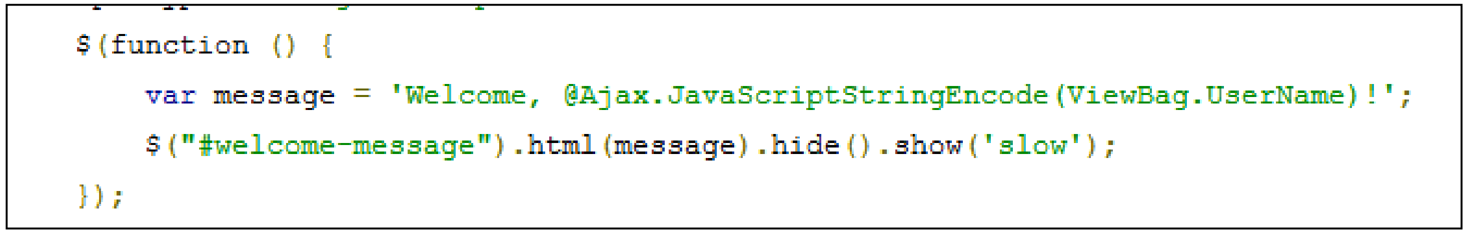


Message from webp...

pwnd

```
http://localhost:XXXXX/?UserName=Waqar\x3cscript\x3e%20alert(\x27pwnd\x27)%20\x3c/script\x3e
```

## JavaScript Encoding Fix



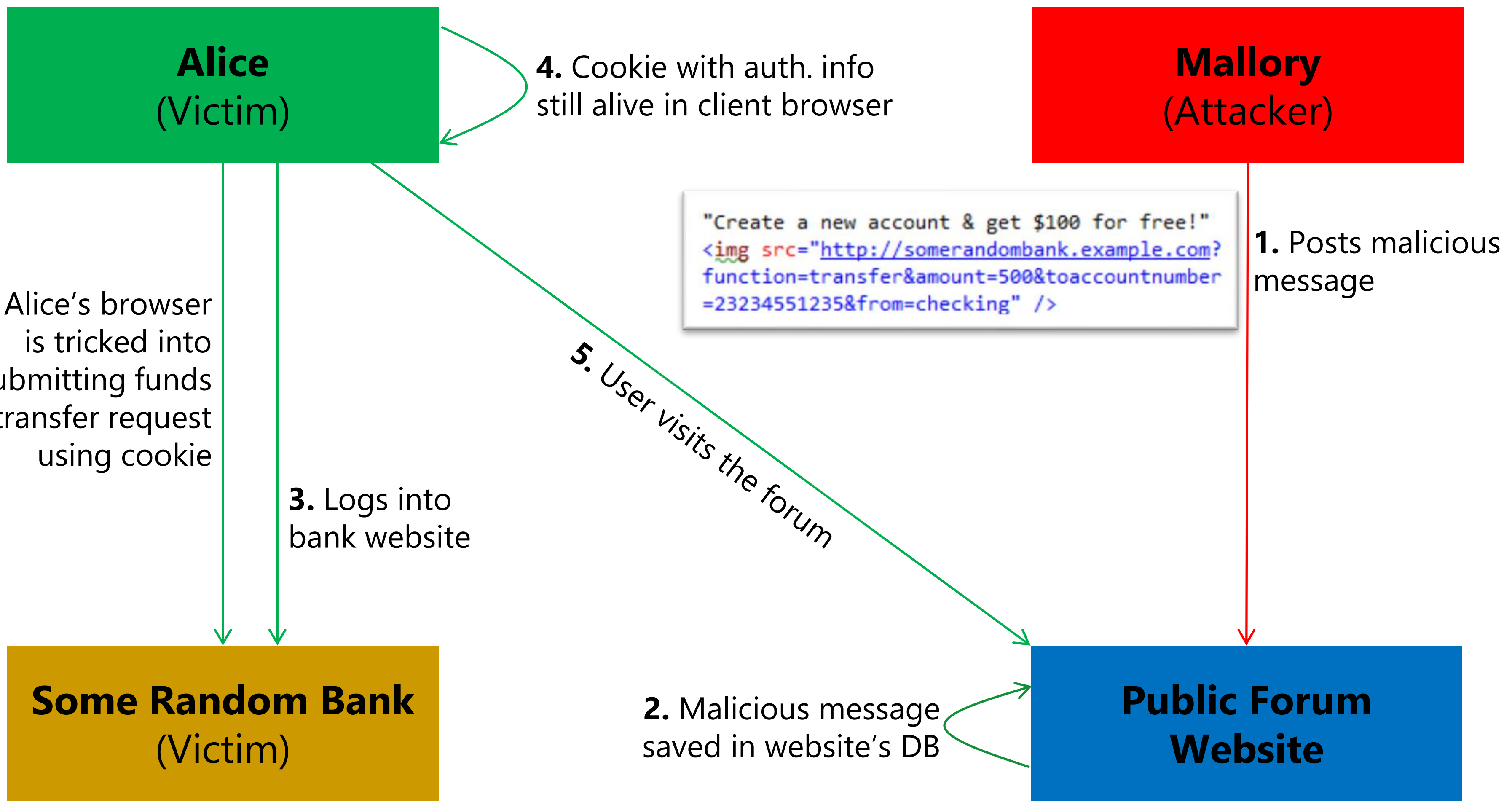
```
    $(function () {
        var message = 'Welcome, @Ajax.JavaScriptStringEncode(ViewBag.UserName)!';
        $("#welcome-message").html(message).hide().show('slow');
    });
```

## Demo: Cross-Site Scripting Attack

## CSRF Attack

- CSRF attack tricks a browser into misusing its authority to represent a user to remote website
- CSRF exploits user’s trust in a browser
    - Confused Deputy Attack against a web browser
- Characteristics of “at-risk” sites:
    - Reliance on user identity
    - Perform actions on input from authenticated user *without* requiring explicit authorization

# CSRF Attack (continued)


Alice
(Victim)
4. Cookie with auth. info still alive in client browser
Mallory
(Attacker)
1. Posts malicious message
6. Alice's browser is tricked into submitting funds transfer request using cookie
3. Logs into bank website
5. User visits the forum
Some Random Bank
(Victim)
2. Malicious message saved in website's DB
Public Forum Website

```
"Create a new account & get $100 for free!"
<img src="http://somerandombank.example.com?
function=transfer&amount=500&toaccountnumber
=23234551235&from=checking" />
```

## CSRF Defense

- **AntiForgery token**: A hidden form field that is validated when the form is submitted
  - Both Html Helper and Tag Helper based forms will *automatically* create an AntiForgery token and include it as a hidden field

```
<form asp-controller="Manage" asp-action="ChangePassword" method="post">

</form>
```

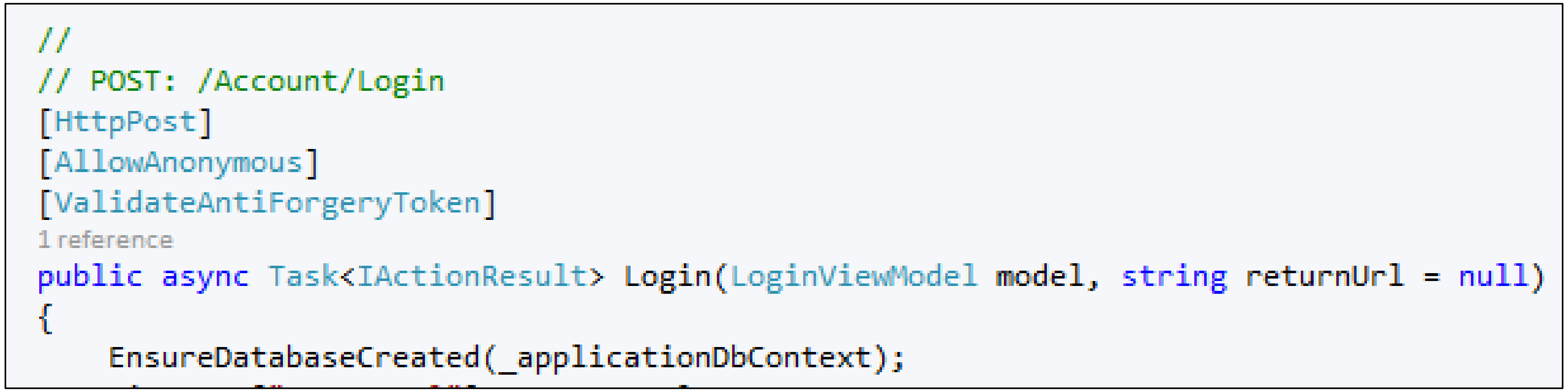
```
@using (Html.BeginForm("ChangePassword", "Manage"))
{

}
```

## Syntax of the Anti-Forgery Token

```
<% using(Html.Form("UserProfile", "SubmitUpdate")) { %>
    <%= Html.AntiForgeryToken() %>
    <!-- rest of form goes here -->
<% } %>
```

## CSRF Defense

- **AntiForgery token**: A hidden form field that is validated when the form is submitted
    - Validate the token on the server side via the `[ValidateAntiForgeryToken]`

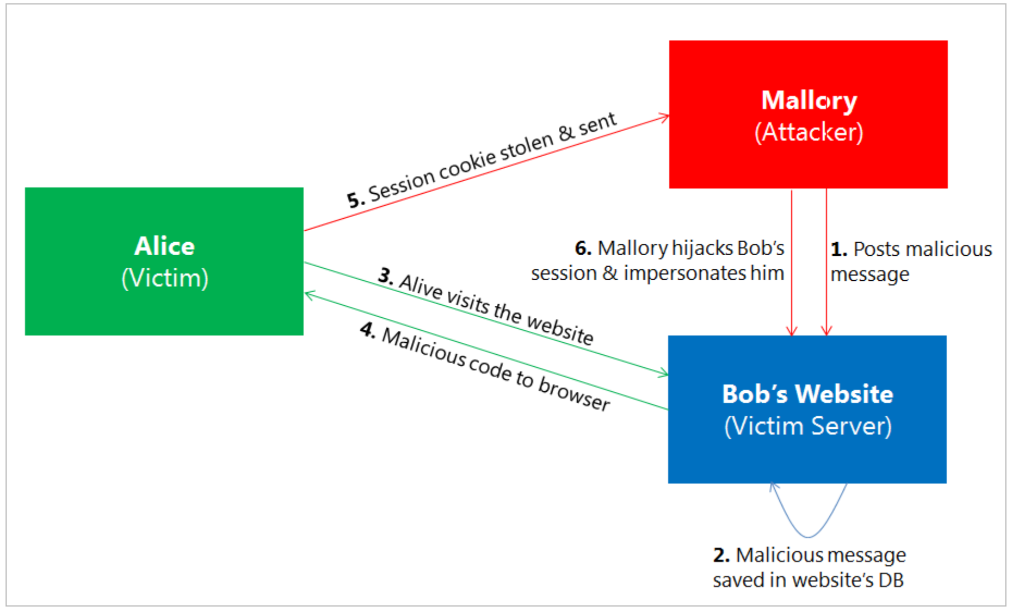


```
//
// POST: /Account/Login
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
1 reference
public async Task<IActionResult> Login(LoginViewModel model, string returnUrl = null)
{
    EnsureDatabaseCreated(_applicationDbContext);
```

## Demo: Cross-Site Request Forgery Attack

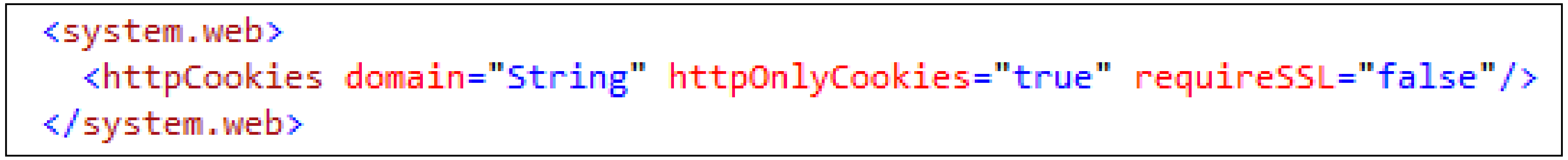
# Cookie Stealing Attack

- Attacker steals user's authentication cookie for a website to impersonate user and carry out actions on user's behalf
- Dependent on XSS attack
  - Attacker must be able to inject script on the target site
  - Script sends user's authentication cookie to attacker's remote server


Mallory
(Attacker)
Alice
(Victim)
Bob's Website
(Victim Server)
1. Posts malicious message
2. Malicious message saved in website's DB
3. Alive visits the website
4. Malicious code to browser
5. Session cookie stolen & sent
6. Mallory hijacks Bob's session & impersonates him

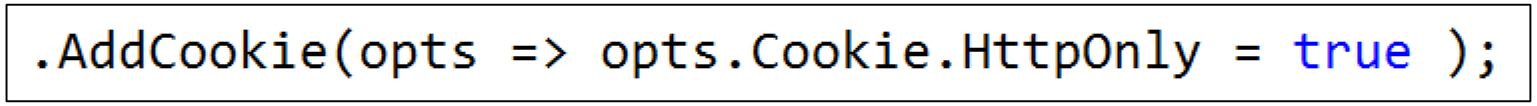
## Cookie Stealing Defense

- Prevent XSS attack on the website
- Disallow changes to the cookie from the client's browser
    - Browser will invalidate the cookie unless the server sets/changes it
    - Can be done from web.config if using IIS



```xml
<system.web>
  <httpCookies domain="String" httpOnlyCookies="true" requireSSL="false"/>
</system.web>
```

    - Can also be set when configuring Cookies in Startup.cs

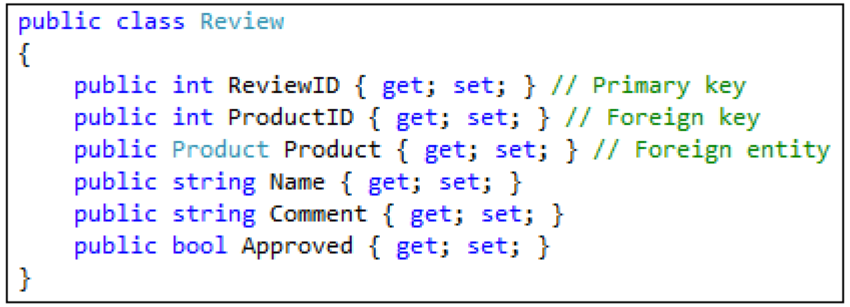


```csharp
.AddCookie(opts => opts.Cookie.HttpOnly = true );
```

# Over-Posting Attack

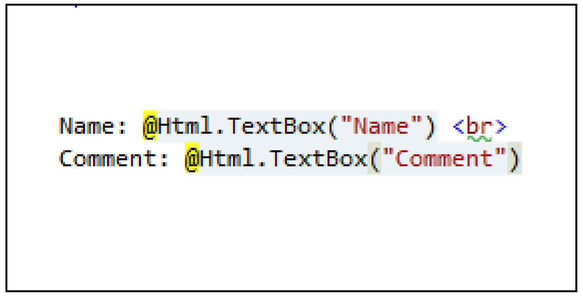
- An attacker can populate model properties that are not included in the View.

**Model**



```
public class Review
{
    public int ReviewID { get; set; } // Primary key
    public int ProductID { get; set; } // Foreign key
    public Product Product { get; set; } // Foreign entity
    public string Name { get; set; }
    public string Comment { get; set; }
    public bool Approved { get; set; }
}
```

**View**



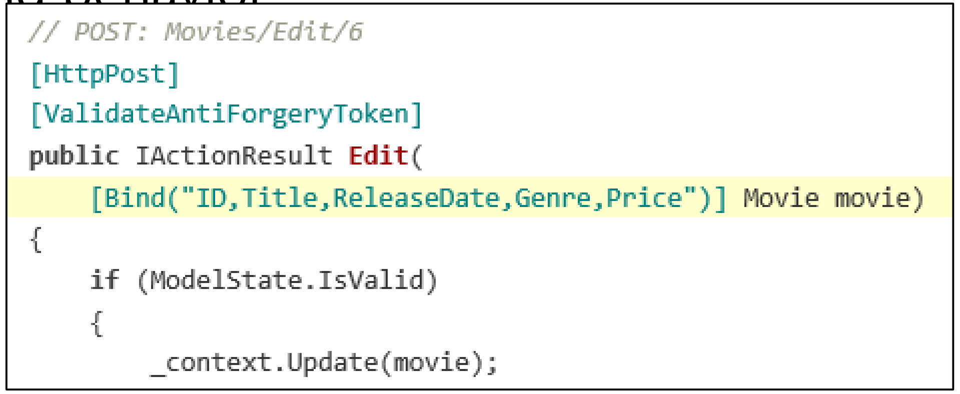
```
Name: @Html.TextBox("Name") <br>
Comment: @Html.TextBox("Comment")
```

- Attacker can add “Approved=true” to form post.
- Attacker can post values for Product, such as Product.Price, to change values in the persistent storage.

## Over-Posting Defense

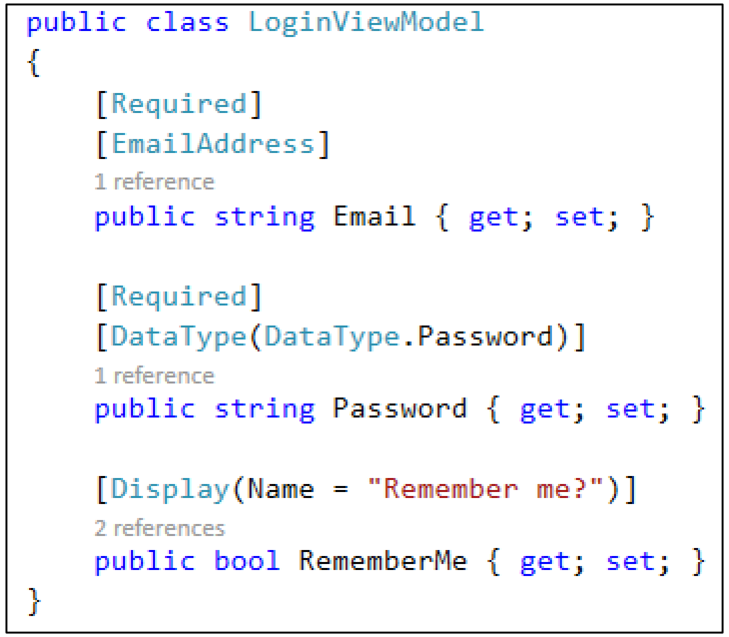
- Use [bind] attribute to explicitly control the binding behavior
  - Specifically list permitted properties

- Use View Model [recommended]



```
// POST: Movies/Edit/6
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(
    [Bind("ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Update(movie);
```

**[Bind]**



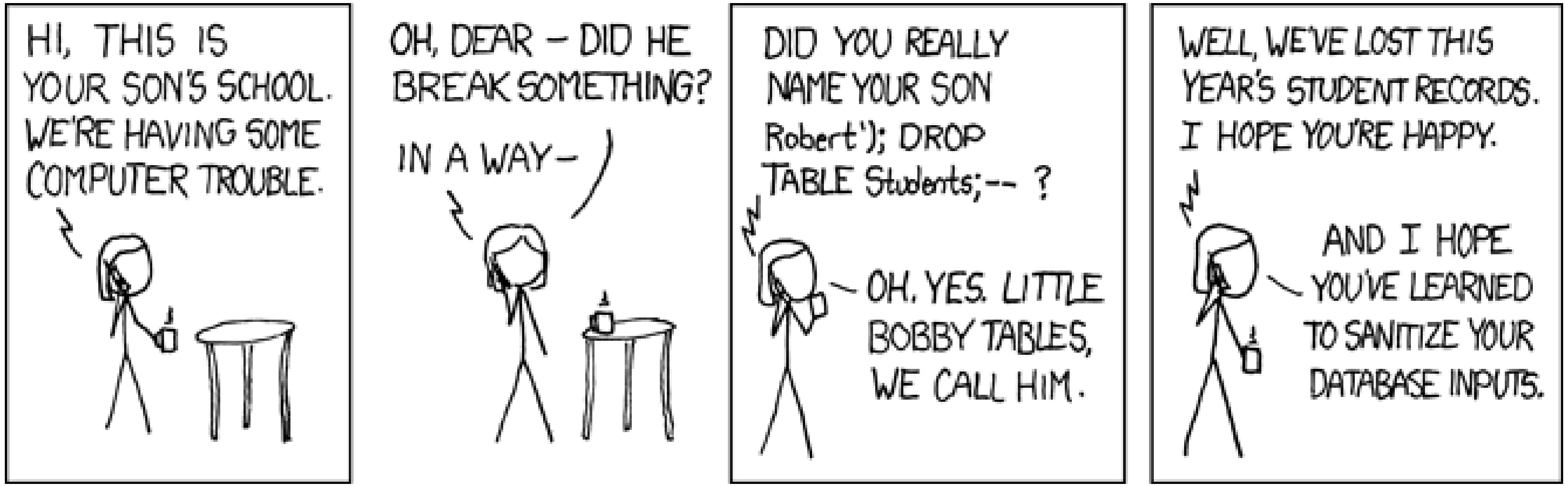
```
public class LoginViewModel
{
    [Required]
    [EmailAddress]
    1 reference
    public string Email { get; set; }

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

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

**View Model**

## Demo: Over-Posting Attack

# SQL Injection

- Malicious code is inserted into strings that are later passed to an instance of SQL Server (or other database).


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

http://xkcd.com/327/

# Threat Defense Summary

| Threat | Solution |
|---|---|
| Cross-Site Scripting (XSS) | • HTML-encode all content<br>• JavaScript encoding |
| Cross-Site Request Forgery (CSRF) | • AntiForgery token<br>• HTTPReferrer validation |
| Over-Posting | • Bind attribute; ViewModels |
| Cookie Stealing | • httpOnly cookies |
| SQL Injection | • Constrain all input<br>• Use type-safe SQL parameters with stored procs<br>• Use parameters collection with dynamic SQL<br>• Use escape routines for special characters<br>• Least-privilege database account<br>• Escape wildcard characters<br>• Avoid disclosing error information |

## Module 9: Security

## Section 8: Trending Web Attacks

## Lesson: OWASP Top 10

## Open Web Application Security Project (OWASP) Top 10 Web Security Attacks (2013)

1. Injection
2. Broken Authentication and Session Management
3. Cross-Site Scripting (XSS)
4. Insecure Direct Object References
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Missing Function Level Access Control
8. Cross-Site Request Forgery (CSRF)
9. Using Components with Known Vulnerabilities
10. Unvalidated Redirects and Forwards

# ASP.NET Defenses Against OWASP Top 10 Attacks

1. Injection
   - Use parametrized SQL queries
   - Use parametrized APIs
   - Restricted binding of Action methods
2. Broken Authentication and Session Management
   - Avoid using custom authentication modules
3. Cross-Site Scripting (XSS)
   - Encode HTML context (body, attribute, JavaScript, CSS, or URL)

## ASP.NET Defenses Against OWASP Top 10 Attacks (continued)

4. Insecure Direct Object References
   - Use random-access reference maps for mapping database key with per-user indirect reference
   - Apply server-side access control for client-side calls
5. Security Misconfiguration
   - Apply repeatable hardening process – Application Lifecycle Management (ALM) and DevOps automation
   - Encrypt sensitive sections of config file(s)
   - Update Operating System/web server/.NET framework/third-party libraries
   - Perform random audits of deployment configuration

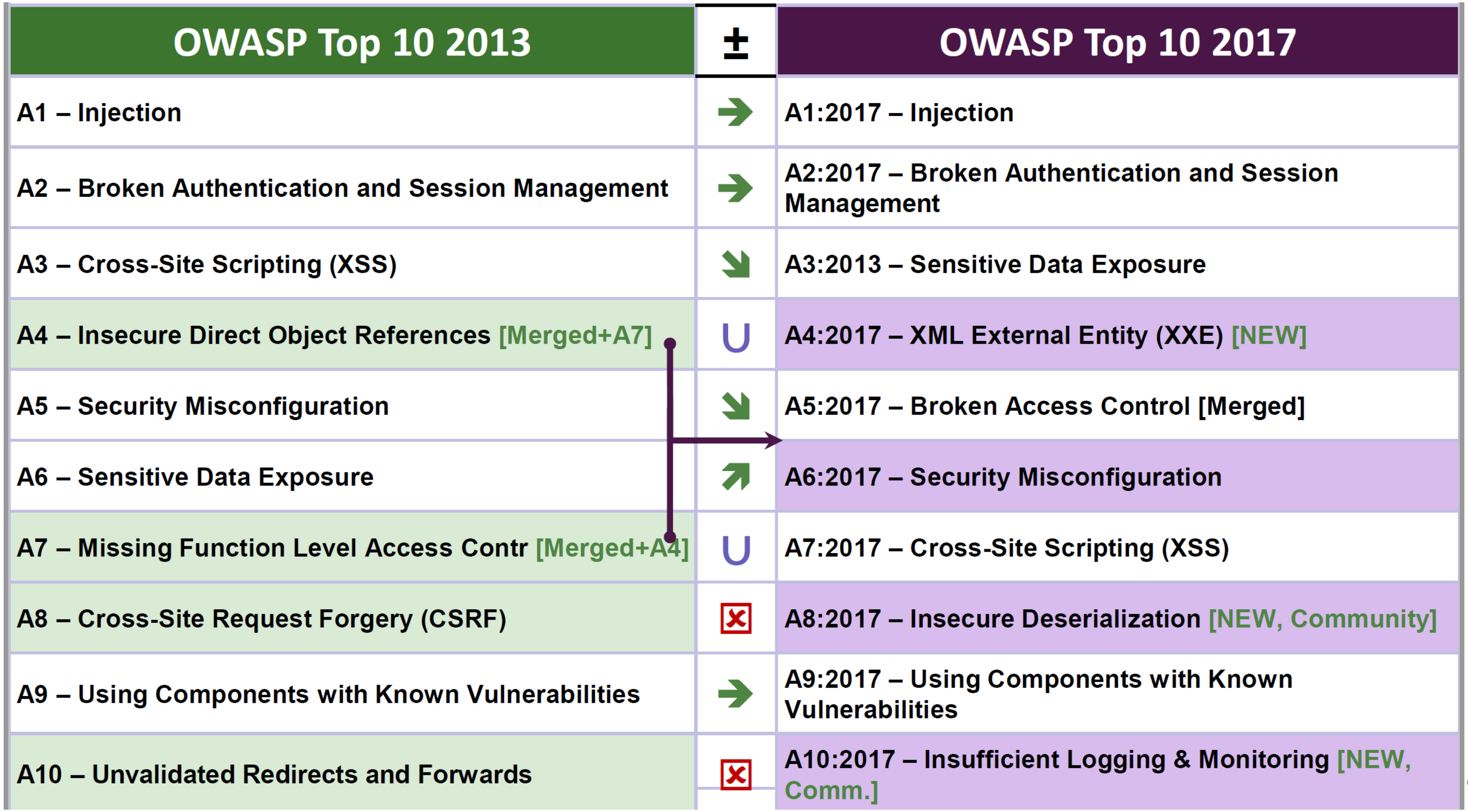
## ASP.NET Defenses Against OWASP Top 10 Attacks (continued)

6. Sensitive Data Exposure
   - Use HTTPs
   - Encrypt data stored in application database(s)
   - Use strong encryption and hashing algorithms
   - Disable caching and autocomplete on sensitive forms
7. Missing Function/Method Level Access Control
   - Use ASP.NET Identity and Roles
8. Cross-Site Request Forgery (CSRF)
   - Generate and include the anti-XSRF tokens in all views
   - Validate tokens in controllers

## ASP.NET Defenses Against OWASP Top 10 Attacks (continued)

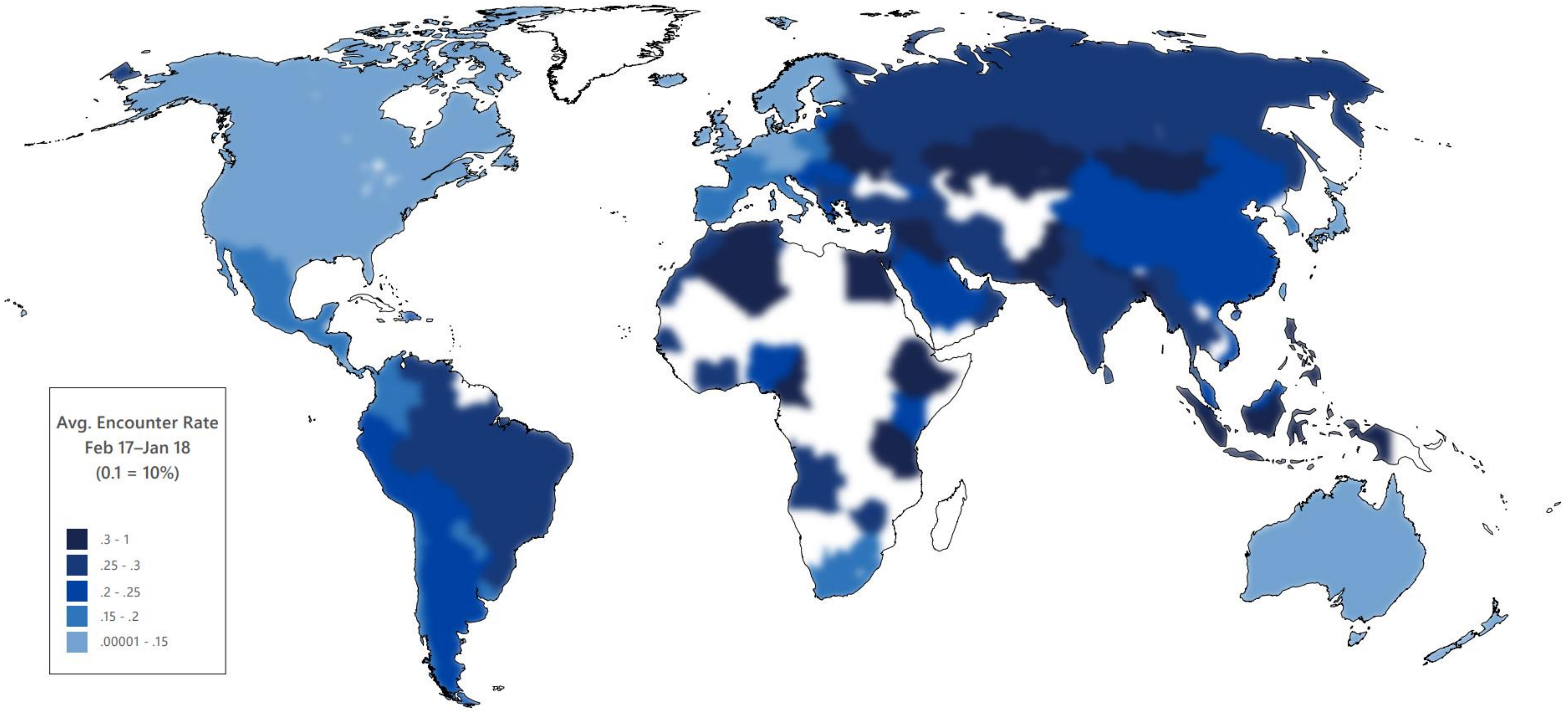
9. Using Components with Known Vulnerabilities
   - Regularly update application components
   - Formulate and enforce effective software security policy in your organization
   - OWASP Safe NuGet package
10. Unvalidated Redirects and Forwards
    - Do not involve user input or parameter in calculating the destination URL
    - If destination parameters are used, verify and authorize them per user

# OWASP Top 10 in 2017



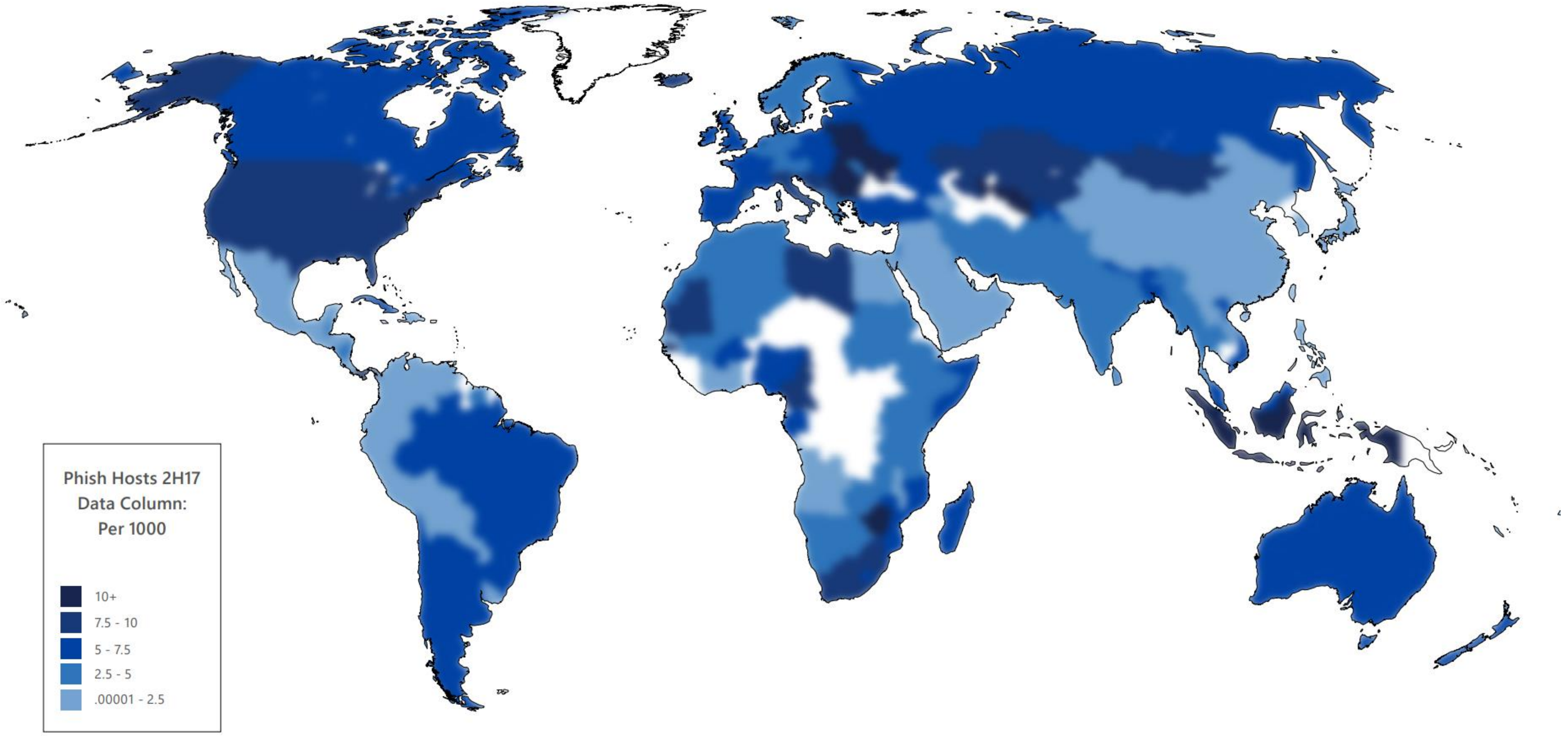
| OWASP Top 10 2013 | ± | OWASP Top 10 2017 |
|---|---|---|
| A1 – Injection | → | A1:2017 – Injection |
| A2 – Broken Authentication and Session Management | → | A2:2017 – Broken Authentication and Session Management |
| A3 – Cross-Site Scripting (XSS) | ↘ | A3:2013 – Sensitive Data Exposure |
| A4 – Insecure Direct Object References [Merged+A7] | ∪ | A4:2017 – XML External Entity (XXE) [NEW] |
| A5 – Security Misconfiguration | ↘ | A5:2017 – Broken Access Control [Merged] |
| A6 – Sensitive Data Exposure | ↗ | A6:2017 – Security Misconfiguration |
| A7 – Missing Function Level Access Contr [Merged+A4] | ∪ | A7:2017 – Cross-Site Scripting (XSS) |
| A8 – Cross-Site Request Forgery (CSRF) | ☒ | A8:2017 – Insecure Deserialization [NEW, Community] |
| A9 – Using Components with Known Vulnerabilities | → | A9:2017 – Using Components with Known Vulnerabilities |
| A10 – Unvalidated Redirects and Forwards | ☒ | A10:2017 – Insufficient Logging & Monitoring [NEW, Comm.] |

# Encounter rates by country/region, February 2017-January 2018



Reference: Microsoft Security Intelligence Report, Volume 23

# Phishing sites per 1,000 Internet hosts for locations around the world in 2H17



Reference: Microsoft Security Intelligence Report, Volume 23

## Important Security Questions

- Does the application have different users who are allowed to do different things?
- How certain do we need to be that the user is who she/he claims to be?
- What is the security level required for different parts of the application?
- How to protect sensitive parts of the application?
- How to ensure that authenticated users only do what they are allowed to do?
- What should be done to ensure that only the right people have access to sensitive data?
- How will we detect malicious behavior?
- How long will the application be down after successful attack? What is the contingency plan?

## Module Summary

- In this module, you learned about:
  - Security fundamentals
  - Authentication and authorization
  - ASP.NET Identity
  - Security threats and defenses
  - OWASP Top 10 web attacks
  - Latest web attacks trends

## Lab: Security

Microsoft