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

< Engineer Name >

Customer Engineer

v3.1

Conditions and Terms of Use

Microsoft Confidential

This training package is proprietary and confidential, and is intended only for uses described in the training materials. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or software included in such packages is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Training package content, including URLs and other Internet Web site references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Copyright and Trademarks

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

For more information, see Use of Microsoft Copyrighted Content at
http://www.microsoft.com/en-us/legal/intellectualproperty/Permissions/default.aspx

Azure, Internet Explorer, Microsoft, SQL Server, Visual Studio, Windows, and Windows Server are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other Microsoft products mentioned herein may be either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are property of their respective owners.

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?

- Security, in information technology (IT), is the defense of digital information and IT assets against internal and external, malicious and accidental threats. This defense includes detection, prevention and response to threats through the use of security policies, software tools and IT services.
- The National Institute of Standards and Technology (NIST) suggests a framework with five pillars or functions of security:

- Identify
- Protect
- Detect
- Respond
- Recover

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

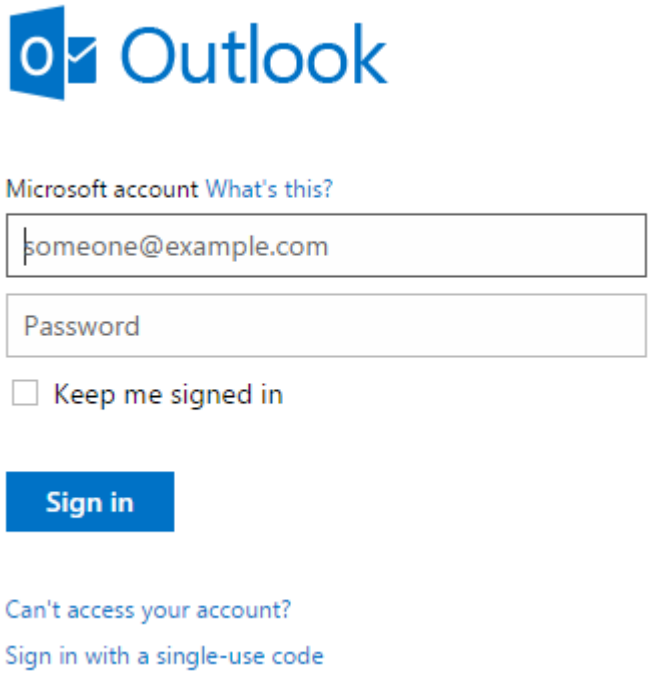
- How do we *represent* a user in our application?
- Typically: A collection of key : value pairs that describe a specific user
  - A pair is referred to as a **claim**
  - The collection of claims makes up an **Identity**
- Represented in code as a model we can create, store, and manipulate
- Can be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

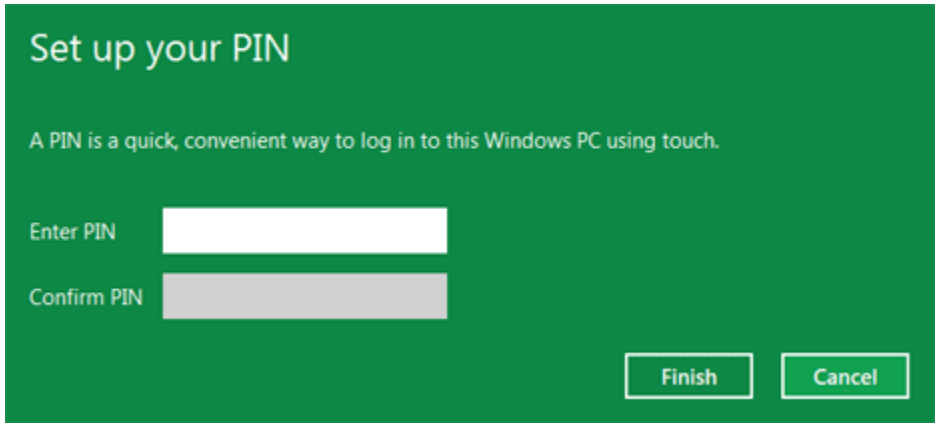
```
{
  “userID”: “83b6734e”,
  “username”: “SuzyQ”,
  “Name”: “Suzy”,
  “givenName”: “Q”,
  “premiumMember”: true
}
```

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

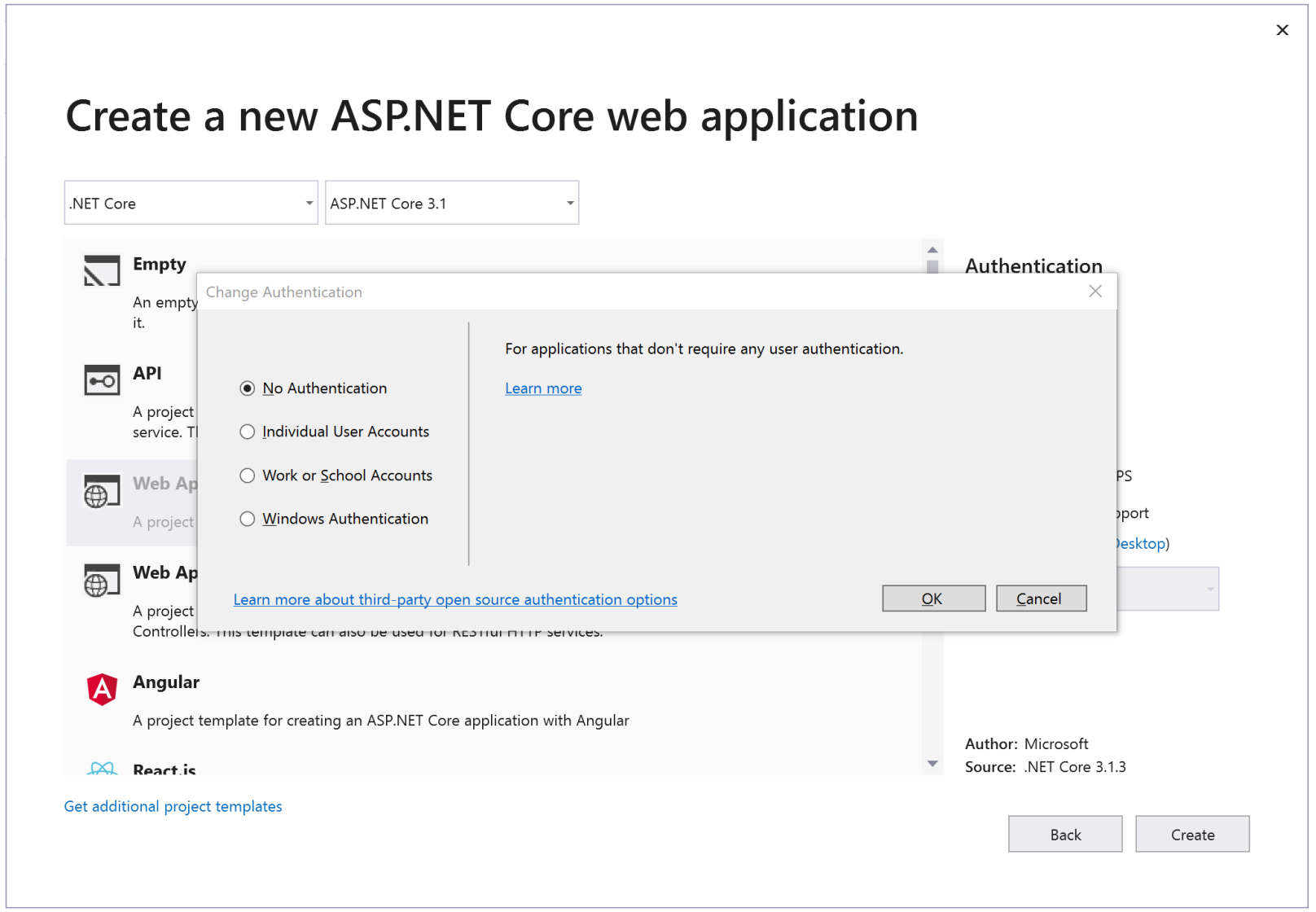
Authentication

- Verifying the users are who they say they are





ASP.NET Core Template Authentication Methods



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

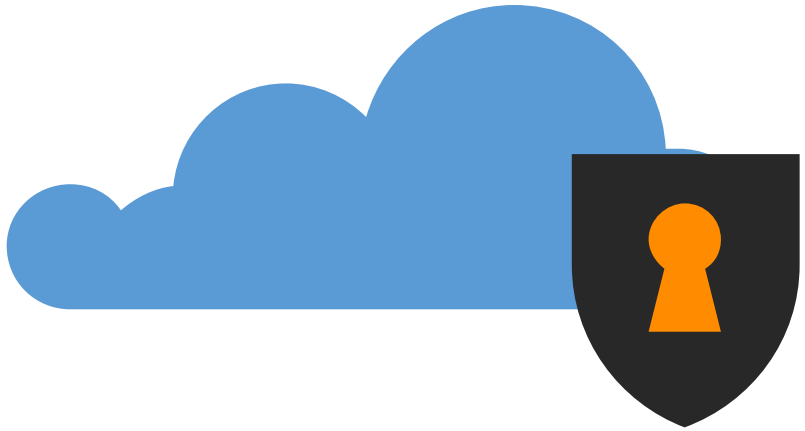
Module 9: Security

Section 2: ASP.NET Identity

Lesson: Overview

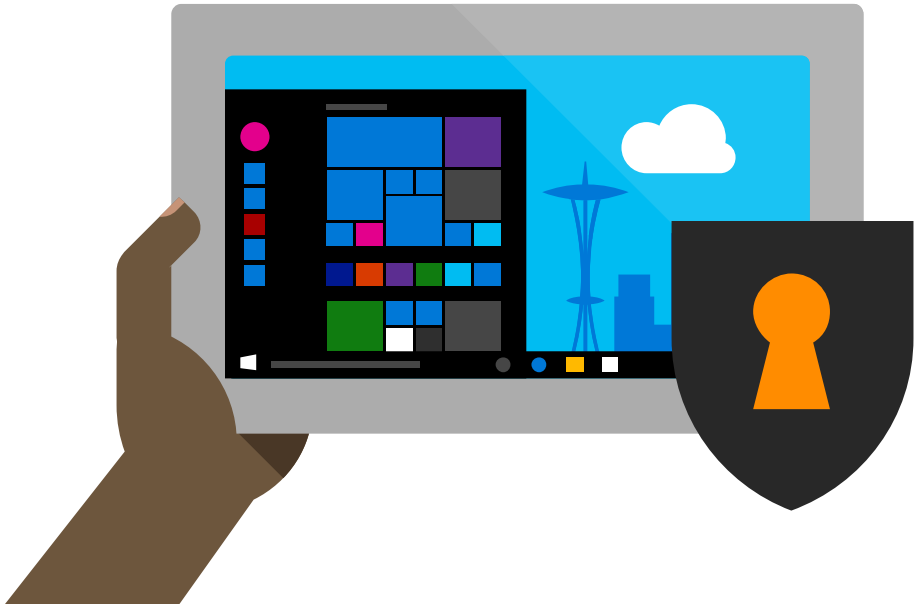
ASP.NET Identity

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



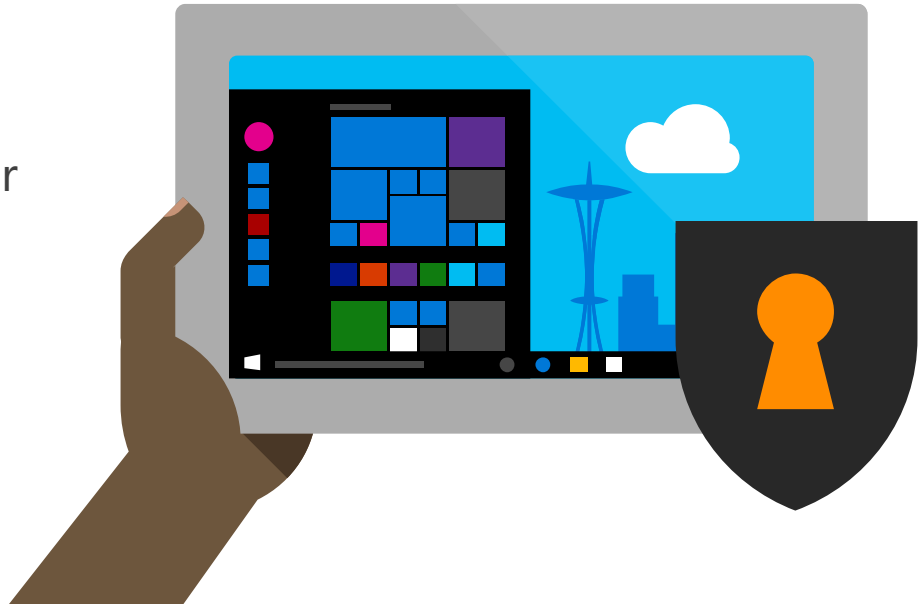
ASP.NET Identity

- **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



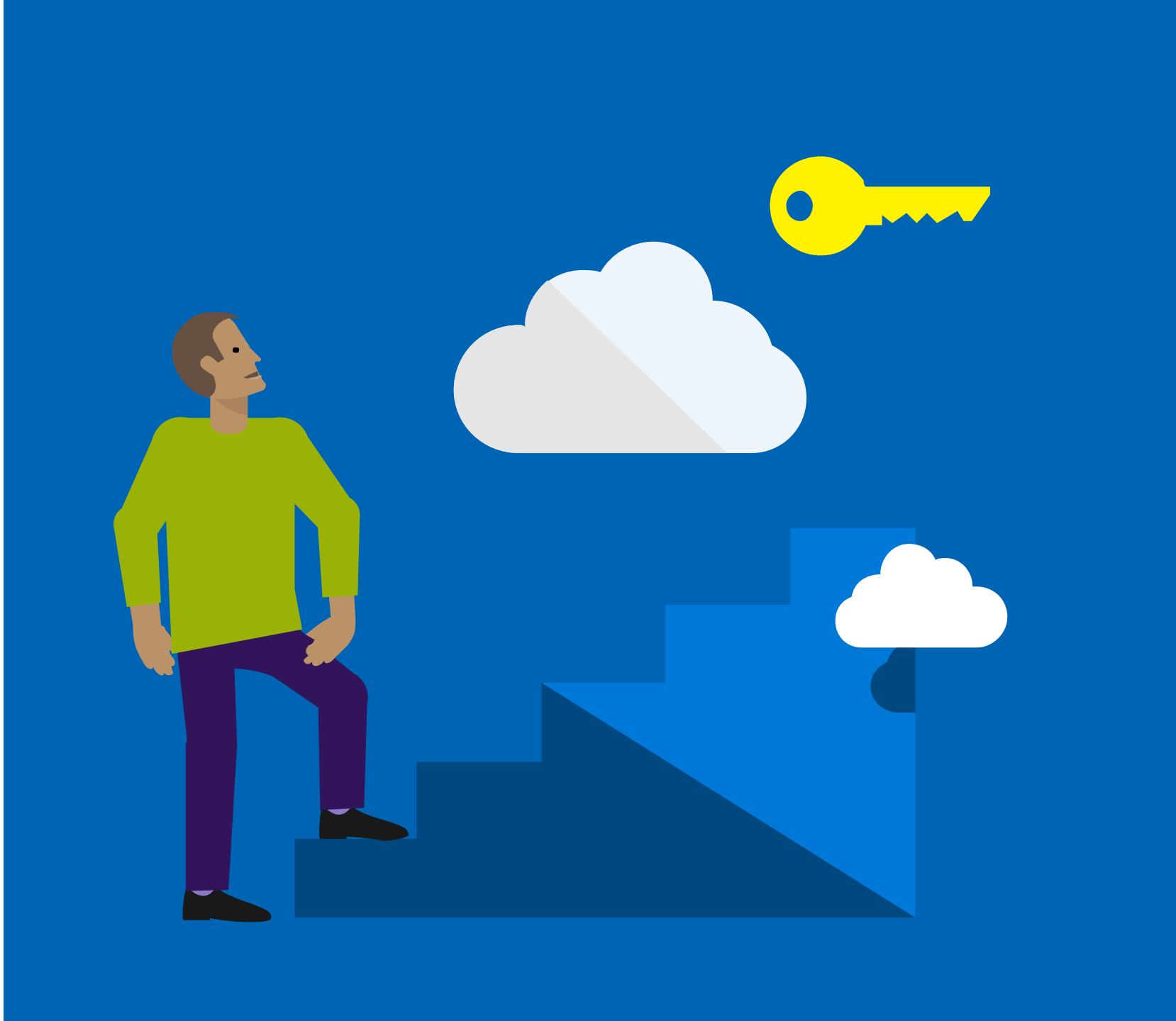
ASP.NET Identity

- **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

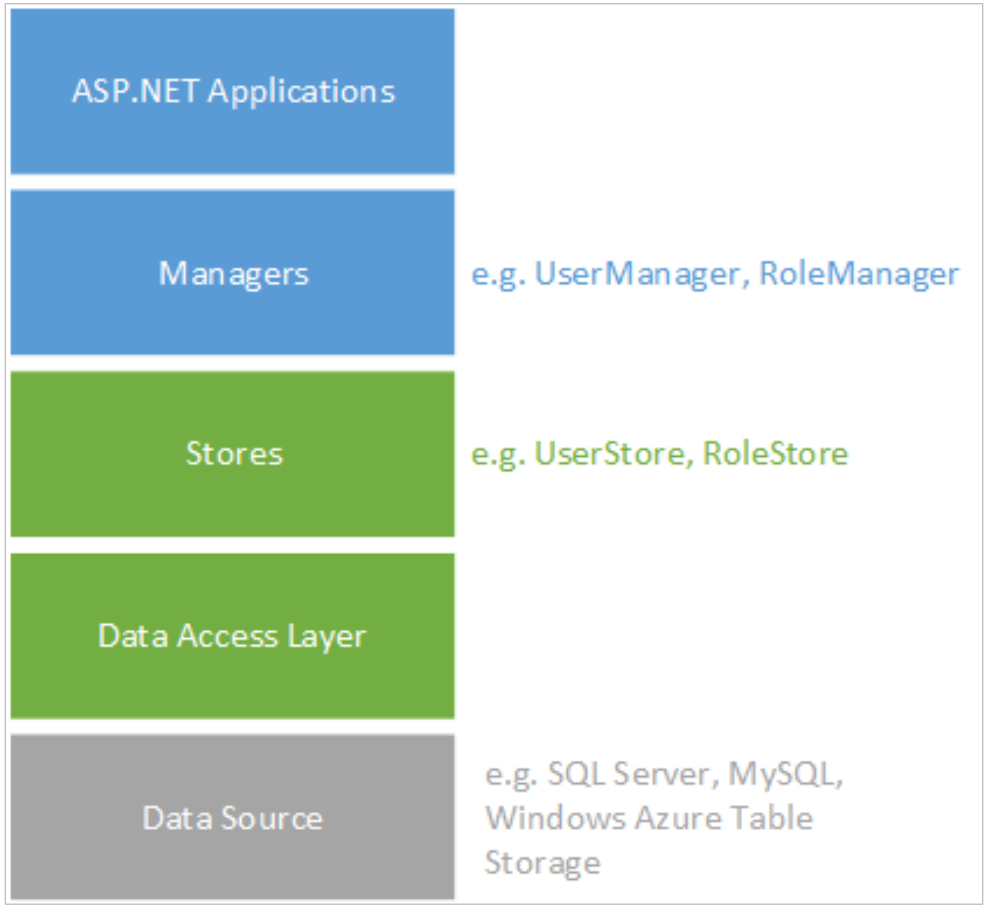
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));

    services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();
    //...
}
```

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    //...

    app.UseAuthentication();
    app.UseAuthorization();

    //...
}
```

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 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

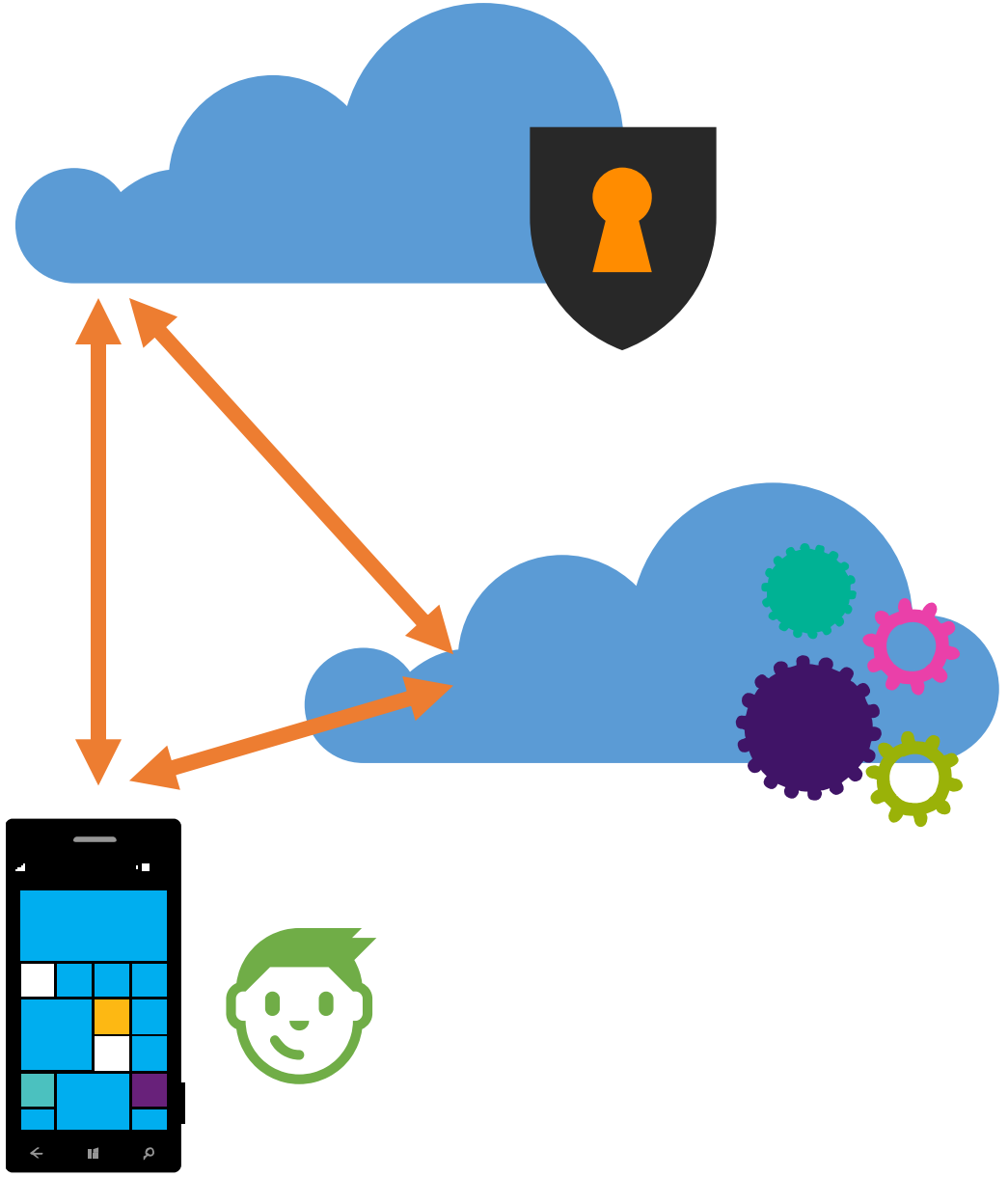
Module 9: Security

Section 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

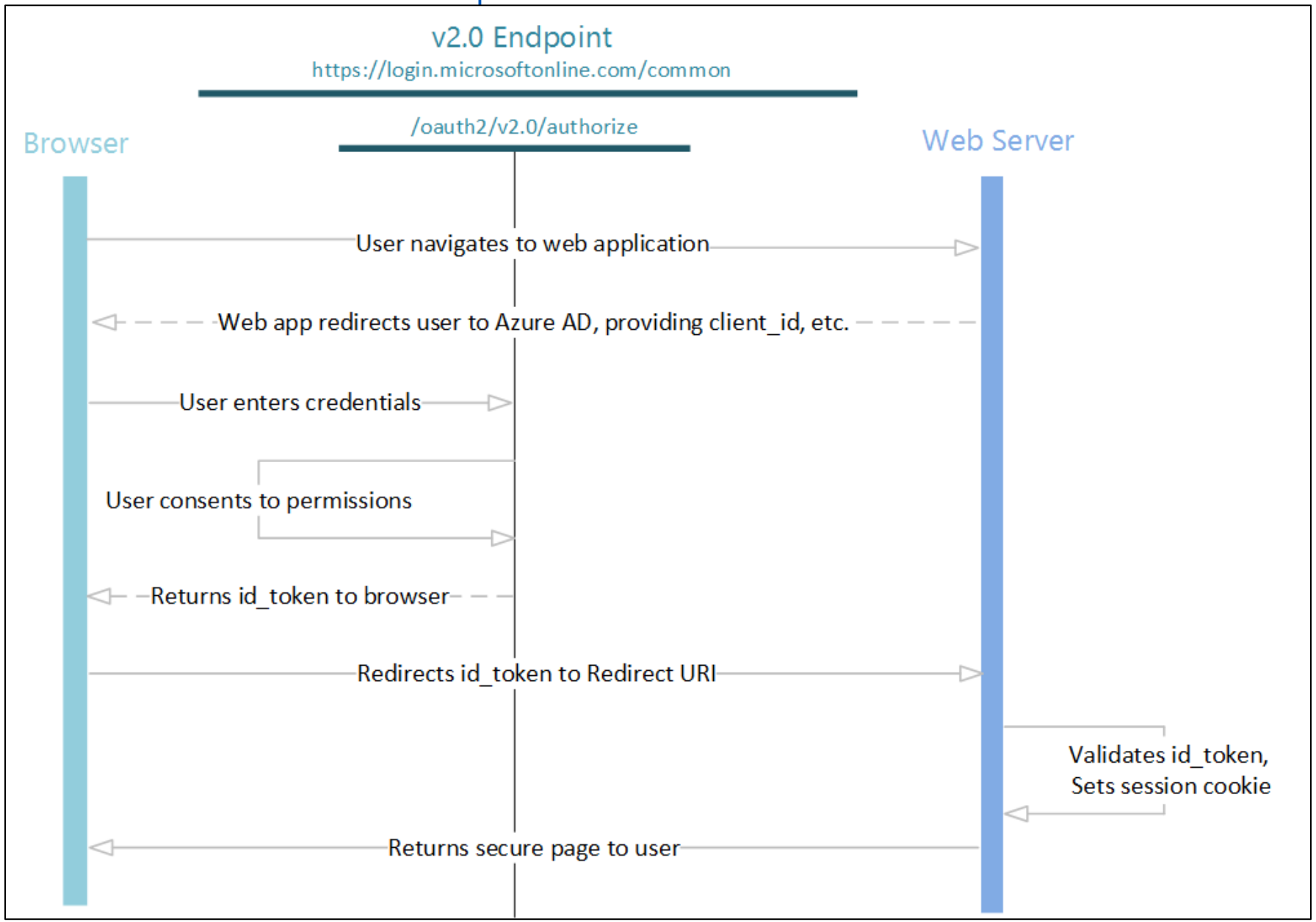
- Your applications *rely* on the identity provider to verify user identities
- Applications need to be registered with the Identity Provider in order to be reliant
- Every application is uniquely identified by a Client ID or Application ID
- Every application is verified via a public/private or shared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



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 Manager
  - [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

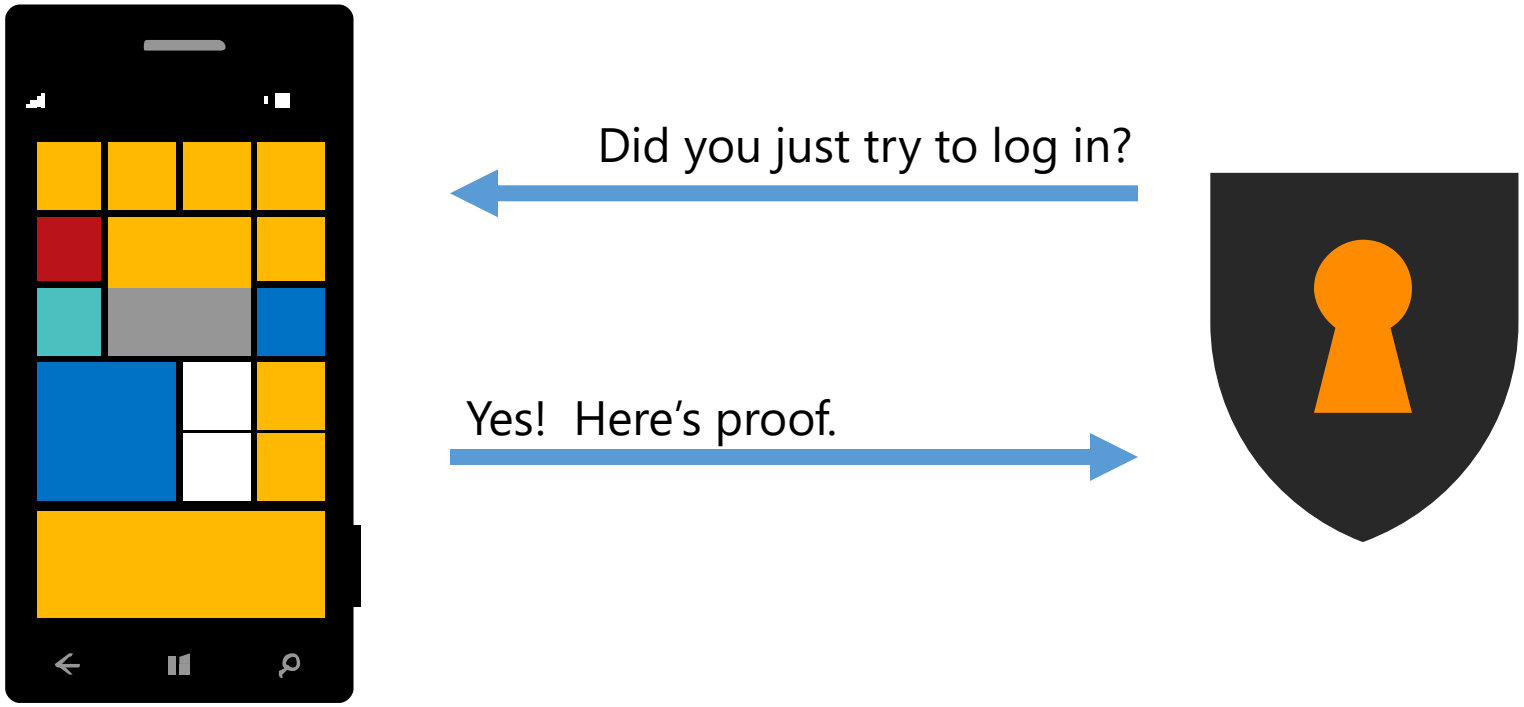
Lesson: ASP.NET Identity Strategies

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: Secret Management

Lesson: Secret Manager

Secret Manager

- Environment variables are used to avoid storage of app secrets in code or in local configuration files
  - Connection Strings, Passwords, Certificates etc

- Secret Manager
  - command line tool
  - stores sensitive data during the development of an ASP.NET Core project
  - local store: **%APPDATA%\Microsoft\UserSecrets\<*user_secrets_id*>\secrets.json**
  - **user_secrets_id >** UserSecretsId value specified in the .csproj file

- In Production, use Azure Vault
  - Secret Manager is **only** for development

Secret Manager (continued)

- To enable secret storage, run in cli
  - dotnet user-secrets init

- UserSecretsId is added to .csproj file

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.1</TargetFramework>
  <UserSecretsId>79a3edd0-2092-40a2-a04d-dcb46d5ca9ed</UserSecretsId>
</PropertyGroup>
```

Secret Manager (continued)

- Set a secret

```
dotnet user-secrets set "Movies:ServiceApiKey" "12345"
```

- List secrets

```
dotnet user-secrets list
```

```
Result
====
Movies:ConnectionString = Server=(localdb)\mssqllocaldb;Database=Movie-1;Trusted_Connection=True;MultipleActiveResultSets=true
Movies:ServiceApiKey = 12345
```

- Remove a secret

```
dotnet user-secrets remove "Movies:ConnectionString"
```

Secret Manager (continued)

- User secrets can be retrieved via the Configuration API

```
public class Startup
{
    private string _moviesApiKey = null;

    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        _moviesApiKey = Configuration["Movies:ServiceApiKey"];
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Run(async (context) =>
        {
            var result = string.IsNullOrEmpty(_moviesApiKey) ? "Null" : "Not Null";
            await context.Response.WriteAsync($"Secret is {result}");
        });
    }
}
```

Startup.cs

Module 9: Security

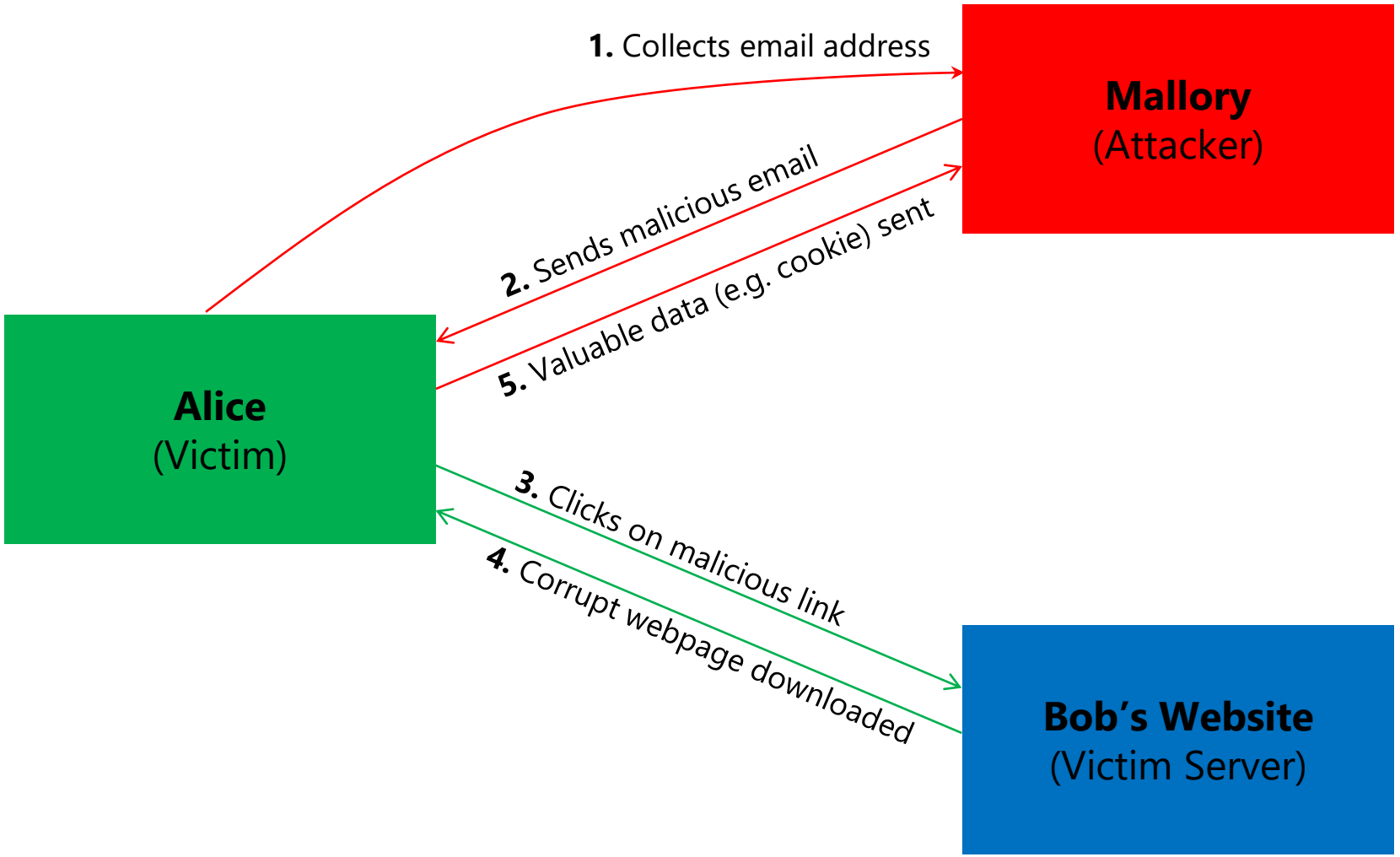
Section 8: Security Threats and Defenses

Lesson: Web Attacks and Defenses

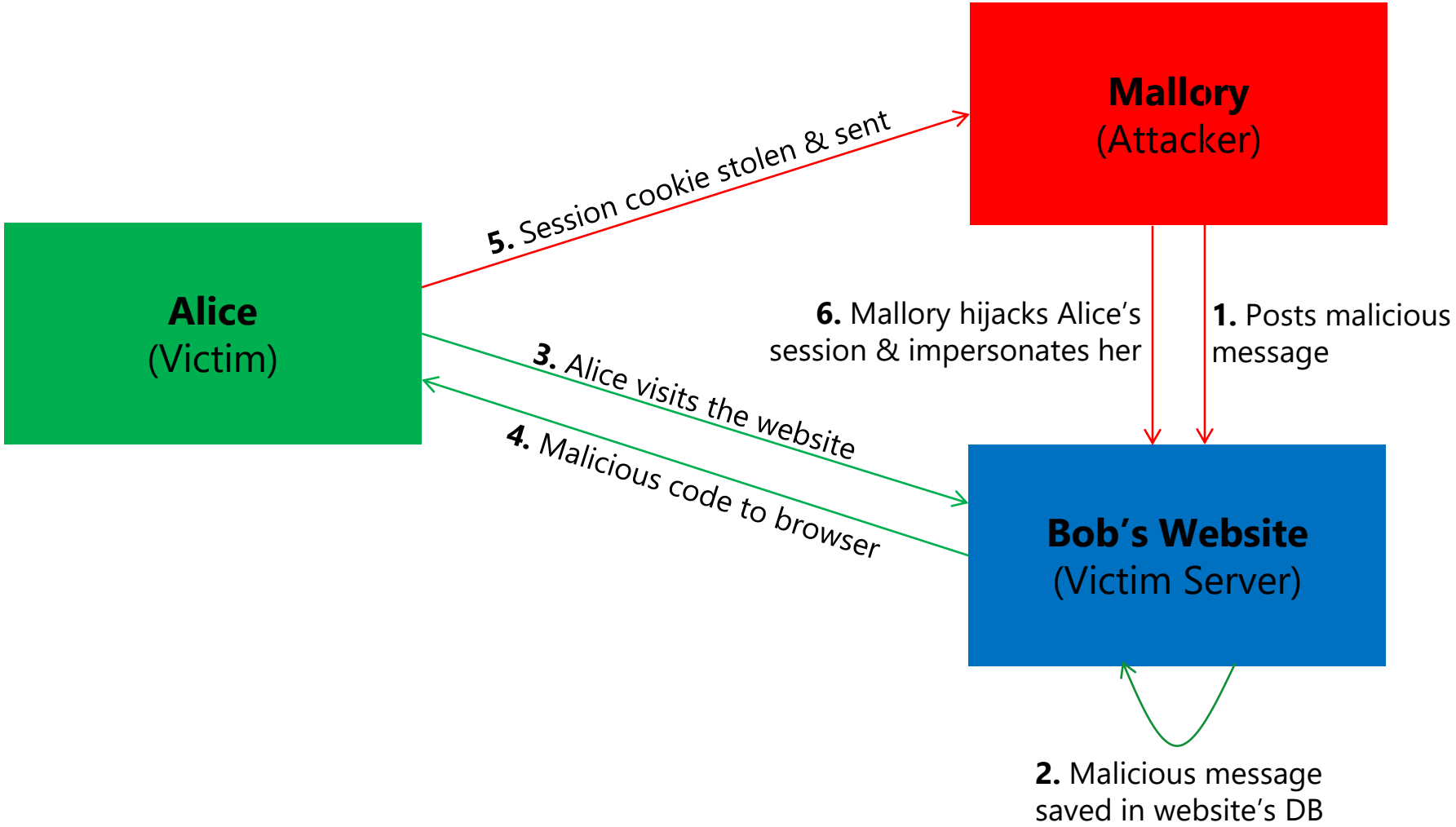
Cross-Site Scripting (XSS) Attack

- 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



XSS Stored Attack



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)

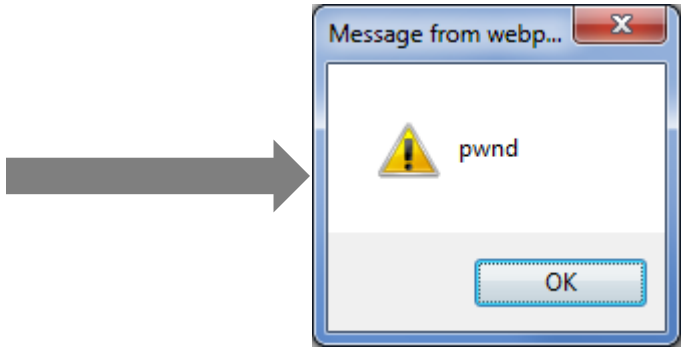
```
<script>alert("XSS!")</script>
```

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>
}
```



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

JavaScript Encoding Fix

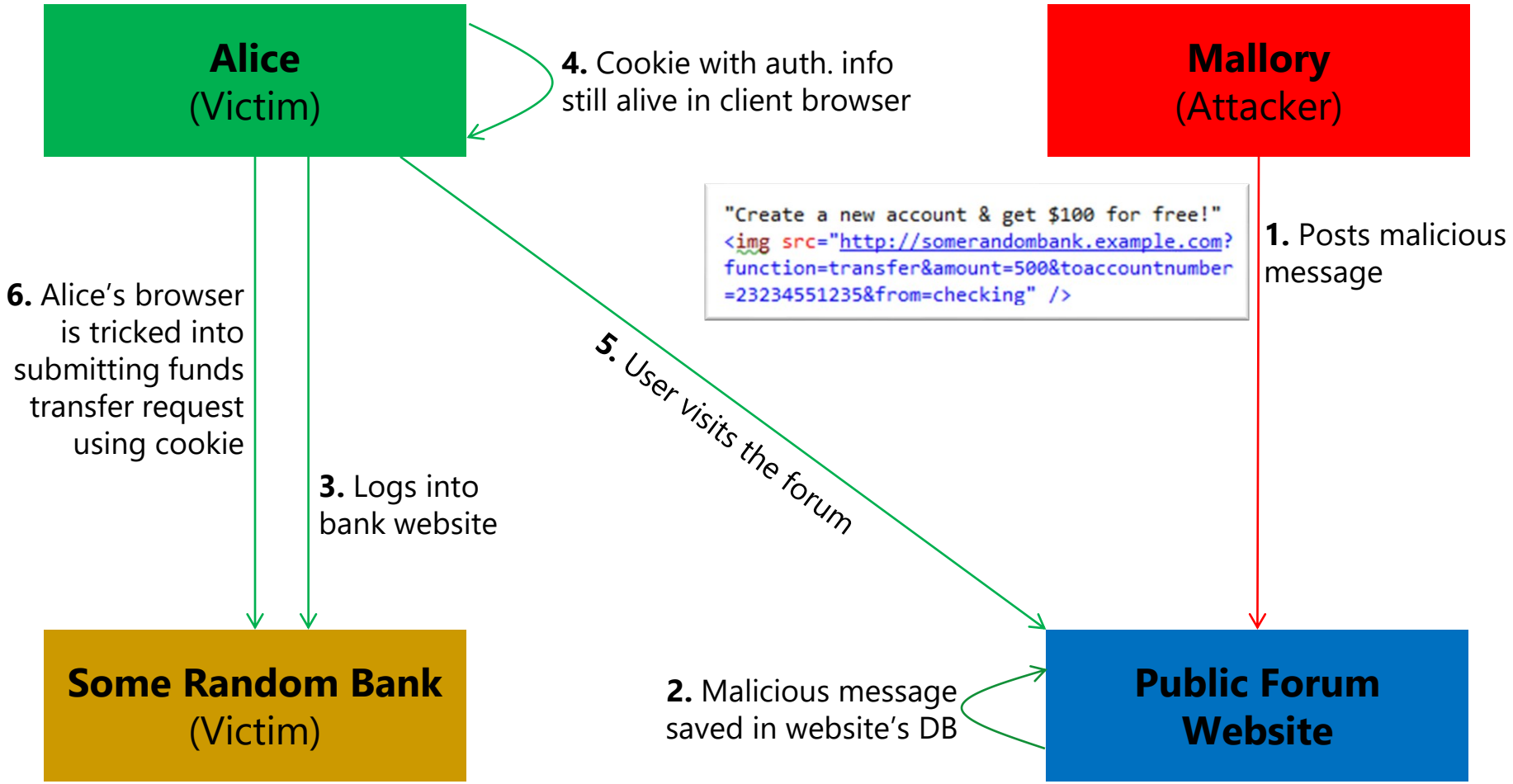
```
    $(function () {
        var message = 'Welcome, @Ajax.JavaScriptStringEncode(ViewBag.UserName)!';
        $("#welcome-message").html(message).hide().show('slow');
    });
```

Demo: Cross-Site Scripting Attack

Cross-Site Request Forgery (CSRF/XSRF) 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/XSRF Attack (continued)



CSRF Defense

- **AntiForgery token**: A hidden form field that is validated when the form is submitted
  - Both Tag Helper and HTML 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

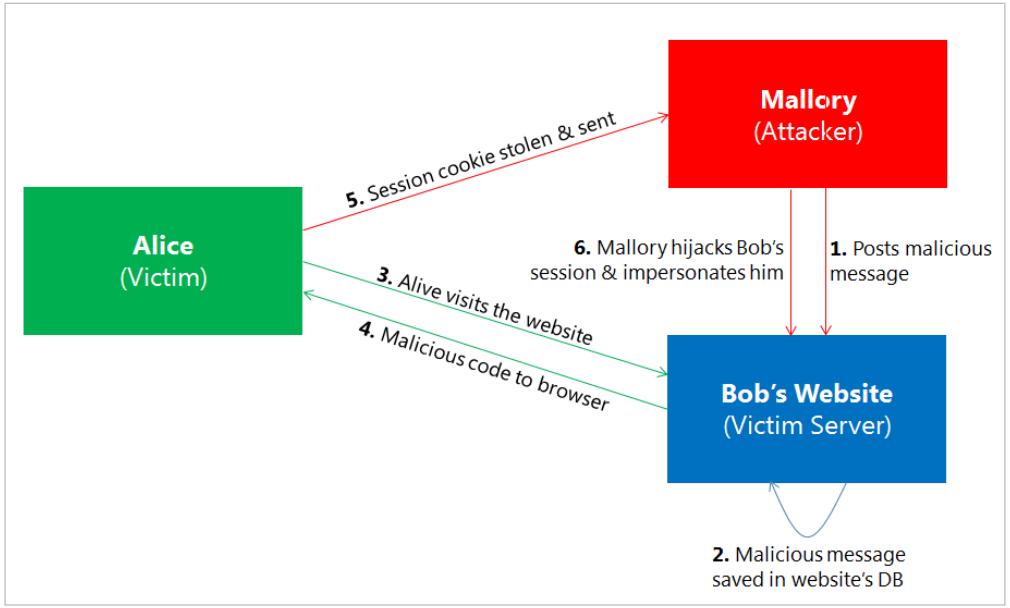
- **AntiForgery token**: A hidden form field that is validated when the form is submitted
  - Validate the token on the server side via the `[ValidateAntiForgeryToken]`

```
//
// 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



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

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

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

```
.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]

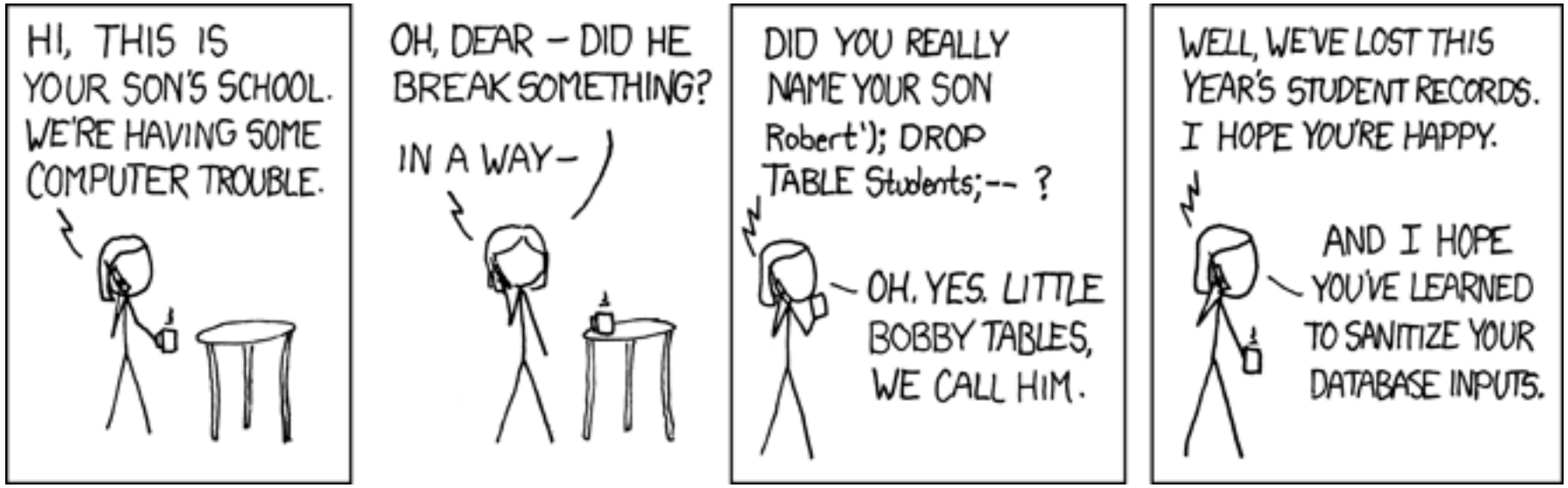
```
public class LoginViewModel
{
    [Required]
    [EmailAddress]
    1 reference
    public string Email { get; set; }

    [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).



http://xkcd.com/327/

Threat Defense Summary

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

Module 9: Security

Section 9: Trending Web Attacks

Lesson: OWASP Top 10

Open Web Application Security Project (OWASP) Top 10 Web Security Attacks (2017)

1. Injection
2. Broken Authentication
3. Sensitive Data Exposure
4. XML External Entities
5. Broken Access Control
6. Security Misconfiguration
7. Cross-Site Scripting (XSS)
8. Insecure Deserialization
9. Using Components with Known Vulnerabilities
10. Insufficient Logging & Monitoring

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
   - Avoid using custom authentication modules

3. 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

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

4. XML External Entities (XXE)
   - Use current versions of .NET which disables entity expansion by default

5. Broken Access Control
   - Except for public resources, deny by default
   - Implement access control mechanisms once and re-use them throughout the application, including minimizing CORS usage
   - Model access controls should enforce record ownership, rather than accepting that the user can create, read, update, or delete any record
   - Unique application business limit requirements enforced by domain models
   - Disable web server directory listing and ensure file metadata (e.g. .git) and backup files are not present within web roots
   - Log access control failures, alert admins when appropriate (e.g. repeated failures)
   - Rate limit API/controller access to minimize effect of automated attack tooling.
   - JWT tokens should be invalidated on the server after logout

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

6. Security Misconfiguration
   - Educate. Bridge the gap between Admins/Developers
   - SDL
   - Hardening Process
   - Patching Process
   - Release and Change Management
   - Solution Architecture (Infrastructure and Applications)
   - Scanning Tools, Monitoring Tools

7. Cross-Site Scripting (XSS)
   - All input must be validated against a whitelist of acceptable value ranges
   - Encode HTML context (body, attribute, JavaScript, CSS, or URL)
   - Use Content Security Policy
   - Protect your Cookies

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

8. Insecure Deserialization
   - Don’t use binary serializers on untrusted input (or at all, if possible)
   - Don’t allow type resolution for untrusted serialized objects
   - When deserializing, turn off all options that aren’t required by that application function. Generate and include the anti-XSRF tokens in all views

9. Using Components with Known Vulnerabilities
   - Regularly update application components
   - Formulate and enforce effective software security policy in your organization
   - Only obtain components from official sources over secure links
   - Subscribe to email alerts for security vulnerabilities related to components you use

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

10. Insufficient Logging & Monitoring
    - Ensure all login, access control failures, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts and held for sufficient time to allow delayed forensic analysis
    - Ensure that logs are generated in a format that can be easily consumed by a centralized log management solutions (Semantic Logging)
    - Establish effective monitoring and alerting such that suspicious activities are detected and responded to in a timely fashion
    - Establish or adopt an incident response and recovery plan, such as NIST 800-61 rev 2 or later
    - Azure Monitor & Azure Sentinel

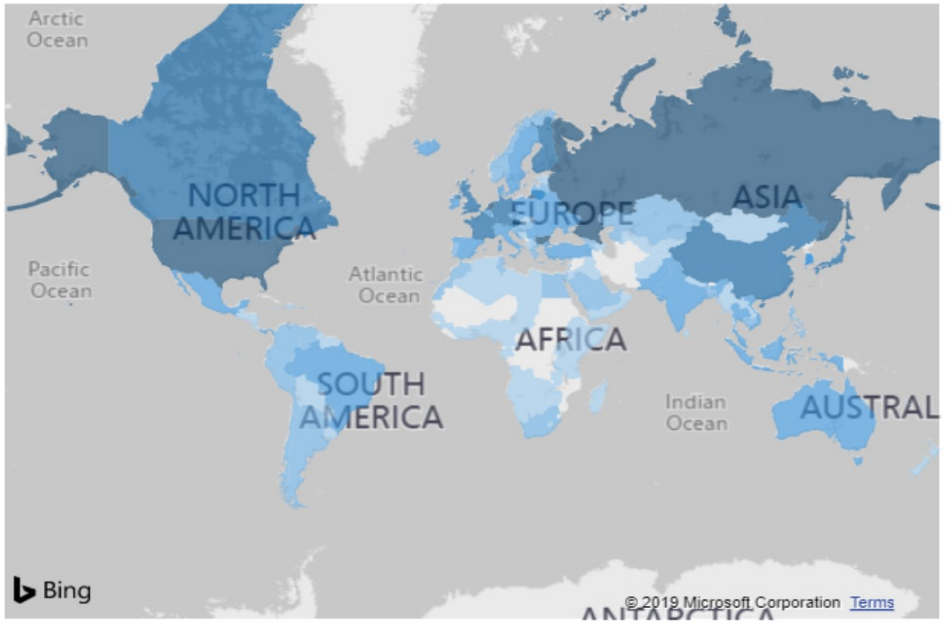
OWASP Top 10

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]	U	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]	U	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.]

Microsoft Security Intelligence Report

Cloud provider related incoming attacks 2018

Country	Incoming Attacks %
United States	19.95%
Netherlands	16.28%
Russia	14.28%
Bulgaria	11.68%
China	9.01%



https://www.microsoft.com/securityinsights/

https://www.microsoft.com/en-us/security/operations/security-intelligence-report

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
  - OIDC and OAuth 2.0
  - Security threats and defenses
  - OWASP Top 10 web attacks
  - Latest web attacks trends

Lab: Security


Microsoft