ASP.NET Core 2.0 Security and Identity

Money Yu

STUDY4.TW
為學習而生

Learn. Imagine. Build.
.NET Conf

About Me

Money Yu (紙鈔)

- Study4TW Member
- Azure Taiwan Co-Organizer

- .NET、.NET Core、ASP.NET、Azure、IoT

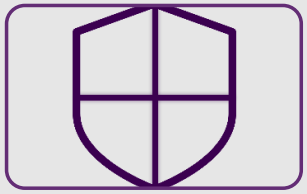
- Facebook: 魚紙鈔
- Linkedin: abc12207

Contents

Hosting Securely

Authentication

Authorization



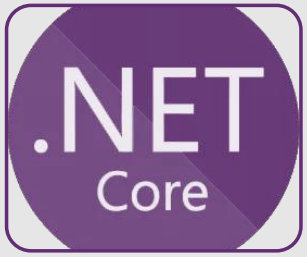
Data Protection

Hosting your Application Securely

Hosting options

Kestrel and IIS

Kestrel and Nginx



Kestrel Only

Kestrel on the Edge

Kestrel in ASP.NET Core V2 is now supported on the edge

- Multiple configuration options for security.
- HTTPS support.
- We still recommend a proxy.

Kestrel Configuration

- Kestrel in ASP.NET v1.x
  - Keep-Alive timeouts.
  - Request Header limits.
  - Request/Response buffer sizes.
  - Request line size.
  - Request header limits.
  - Request header count limits.
- Kestrel in ASP.NET v2.0
  - Request/Response body timeouts & data rates.
  - Total client connections.
  - Internal request draining support.

Kestrel Configuration

- Kestrel in ASP.NET v2.1
  - Default HTTPS
  - Transport Configuration
  - Url-Prefix

Authenticating your Users

Authentication in ASP.NET Core

- Template Support
  - No Authentication.
  - Individual User Accounts.
  - Work and School Accounts.
  - Windows Authentication.
- Social Authentication
  - Facebook.
  - Twitter.
  - Google.
  - Microsoft Account.
  - etc: https://bit.ly/2xxUKLB

Template Options

- Individual User Accounts
  - Local database - EF models for users, accounts, groups, etc.
  - Opinionated.
- Authentication via OIDC
  - Work and School Accounts, via Azure AD and OIDC (OpenID Connect).
  - Recent SAML/WS-Fed support.
- Windows Authentication
  - Needs IIS.
  - Local domain joined servers.
  - No automatic impersonation any more.

ASP.NET Core 1.0 Authentication Changes

- No more custom identity classes
  - Everything is ClaimsPrincipal based.
- Multiple authentication middleware
  - They don’t have to run automatically.
  - Authorization can cherry pick which authentication scheme to use.
  - Only one middleware can run automatically.
- Cookie authentication
  - Finally understands unauthenticated or forbidden.

ASP.NET Core 2.0 Authentication Changes

- Authentication is now a single service
  - No more individual authentication middlewares.
  - Turned on with `app.UseAuthentication()` in `Configure(IApplicationBuilder)`.
  - Each authentication type is added and configured in `ConfigureServices()`.

- TOTP
  - ASP.NET Identity now has support for, and defaults to TOTP with authenticator apps.

Demo:
TOTP (Time-based One-Time Password)

Configuring Authentication in v1

```
app.UseCookieAuthentication(new CookieAuthenticationOptions
() {  AccessDeniedPath = "/Account/Forbidden/",
    LoginPath = "/Account/Unauthorized/",
    AuthenticationScheme =
    CookieAuthenticationDefaults.AuthenticationScheme,
    AutomaticAuthenticate = true,  AutomaticChallenge =
    true
});
app.UseTwitterAuthentication(…);
```

Configuring Authentication in v2

```
// In ConfigureServices(IServiceCollection services)
services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme =
    CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultSignInScheme =
    CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
    CookieAuthenticationDefaults.AuthenticationScheme;
})
.AddCookie(options =>
{
    options.LoginPath = new PathString("/Account/Login/");
    options.AccessDeniedPath = new PathString("/Account/Forbidden/");
});

// In Configure(IApplicationBuilder app, IHostingEnvironment env)
app.UseAuthentication();
```

Add Google

Configuring Authentication

- DefaultScheme
  - When everything is the same handler.
- DefaultAuthenticate
  - Called to construct the identity for a request.
- DefaultSignin
  - Called when a user triggers a sign-in.
- DefaultChallenge
  - Called when a challenge is triggered, for example hitting `[Authorize]`.

Authenticate/Challenge/Signin 1.0 v 2.0

```
// 1.0
using Microsoft.AspNetCore.Http.Authentication;

context.Authentication.    Authenticate|Challenge|
SignInAsync("scheme");

// 2.0
using Microsoft.AspNetCore.Authentication;
 context.Authenticate|Challenge|SignInAsync("scheme");
```

Do it yourself

- **Cookie Authentication**
  - Everything ends up in cookie middleware.
  - Cookie middleware encrypts and signs cookies identifying a user principal.
  - Cookie middleware uses Data Protection - keys must be synced between servers.
  - You can use the cookie middleware to serialize your own `ClaimsIdentity` and authenticate subsequent requests.
- **But**
  - Once a cookie is dropped it's the sole source of truth unless you implement a validator.
  - Validators can reject principals, or replace them.
  - Validators on raw Cookie Middleware run on every request.

Demo:
Baking your own cookies

Selecting service to authorize with

```
[Authorize(ActiveAuthenticationSchemes = "Bearer")]
public class ApiController : Controller
{
}
```

Coming to ASP.NET Core 2.1

- Identity as a service
  - Using ASP.NET identity as backing store for OIDC service.
  - Same flow used for Facebook, Google et al.
  - Introduces support for mobile, native and SPA.
  - Swap out to Azure B2C, Identity Server, OpenIddict, ASOS etc.
- WS-FED client support

Controlling access via Authorization

Authorization

- Features
  - Code based authorization policies, requirements, and requirement handlers.
  - Resource based authorization.
  - DI based.
  - Authentication scheme filtering.
  - Custom policy providers.

- Workshop
  - https://github.com/blowdart/AspNetAuthorizationWorkshop

Policies, requirements and handlers

- Policies
  - Made up of one or more requirements.
- Requirements
  - `IAuthorizationRequirement`.
  - Can have multiple handlers.
  - All requirements must succeed for the policy to pass.
  - All requirements are evaluated, even if previous ones fail.
- Handlers
  - `AuthorizationHandler<IAuthorizationRequirement>`.
  - Must be registered in DI system.

Requirements

```
using Microsoft.AspNetCore.Authorization;


namespace AuthorizationLab
{
    public class OfficeEntryRequirement :
    IAuthorizationRequirement
    {
    }
}
```

Handlers

```
public class HasBadgeHandler :
AuthorizationHandler<OfficeEntryRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        OfficeEntryRequirement requirement)
    {
        if (!context.User.HasClaim(c => c.Type == "BadgeNumber" &&
                                        c.Issuer ==
                                        "https://contoso.com"))
        {
            return Task.CompletedTask;
        }

        context.Succeed(requirement);
        return Task.CompletedTask;
    }
}
```

Policies

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy("BuildingEntry",
                          policy => policy.Requirements.Add(
                                        new OfficeEntryRequirement())
                                        );
    });

    // Register the handler for the requirement.
    services.AddSingleton<IAuthorizationHandler, HasBadgeHandler>();
}
```

Authorizing

```
[Authorize(Policy = "BuildingEntry")]
public class BuildingController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

Demo:
Authorization

What should a handler return?

- Successful Evaluation
  - Call `context.Succeed(requirement)`.
- Unsuccessful Evaluation
  - Do nothing.
- Horrific Circumstances
  - The user has just been fired, but not everything has been updated.
  - My database server is on fire.
  - Call `context.Fail()`.

Things to remember

- Your handlers must be registered in DI
  - As they're in DI you can inject repos, etc.
  - Remember your DI scope otherwise EF will get upset.
- Check your claims issuer

Imperative Checks

- Evaluate your policies in code
  - Inject `IAuthorizationService` into your controller.
  - Call `AuthorizeAsync()`.
  - If failed return `ForbiddenResult()`.

Adjusting UI in MVC Views

- Use Razor's DI system
  - `@inject IAuthorizationService AuthorizationService`.
  - Put in `_ViewImports.cshtml` to make globally available.
  - Call `Authorization.AuthorizeAsync()`.
- Imperative checks in views
  - Remember to duplicate the checks in your controllers.

Resource based authorization

- Acting on a resource or model
  - For example check that the current user owns the resource requested.
- Operation Based
  - Base class provided – `OperationAuthorizationRequirement`.
- Remember multiple handlers
  - A common administrator handler could be combined with resource ownership checks.

Resource authorization handlers

```
public class DoesCurrentUserOwnHandler :
AuthorizationHandler<OperationAuthorizationRequirement, Document>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        OperationAuthorizationRequirement requirement,
        Document resource)
    {
        if (document != null && document.Owner ==
        context.Identity.Name)
        {
            context.Succeed(requirement);
            return Task.CompletedTask;
        }
    }
}
```

Limiting authentication service in policy

```
options.AddPolicy("api", policy =>
{
    policy.AuthenticationSchemes.Add("Bearer");
    policy.RequireAuthenticatedUser();
});
```

Protecting your Data

Data Protection

- One stop shop for encryption
  - No more machine key.
  - Aimed at ephemeral data.
  - Removes the ability to shoot yourself in the foot.
  - Supports key rotation automatically.
  - Provides isolation for applications automatically.
  - Provides isolation based on purposes automatically.
  - Attempts to figure out where to store keys based on app platform.
  - Easy to write new key stores to match your customers' environments.
    - `IXmlRepository` – `GetAllElements()` / `StoreElement()`.
  - Custom algorithms supported (for Russia).

Defaults

- Cryptography defaults
  - 512bit master key
  - Rolled every 90 days
  - Derived keys based on purpose and every payload
  - AES-256 CBC for encryption
  - HMACSHA256 for authenticity

Storing and protecting keys

- Key Stores
  - Azure Web Applications – Special synced folder.
  - IIS with no user profile – registry, with machine DPAPI & worker process ACLed.
  - IIS with user profile - `%AppData%`, user DPAPI.
  - In memory, discarded. Or folder..
- Protection
  - DPAPI, DPAPI NG (with AD), X509 Certificate, Plain Text.
  - Azure KeyVault in ASP.NET Core 2.1

Using Data Protection

- Manually
  - Create a `IDataProtectionProvider`.
  - Create a `DataProtector` with a purpose from the `IDataProtectionProvider`.
- ASP.NET Core
  - `services.AddDataProtection()`.
  - Take `IDataProtectionProvider` in your controller constructors.
- Use
  - Call `protector.Protect()`.
  - Call `protecter.Unprotect()`.

Using Data Protection

```
var dataProtectionProvider = new EphemeralDataProtectionProvider();
var protector = dataProtectionProvider.CreateProtector("purpose");

Console.Write("Enter input: ");
string input = Console.ReadLine();

string protectedPayload = protector.Protect(input);
Console.WriteLine($"Protect returned: {protectedPayload}");

string unprotectedPayload = protector.Unprotect(protectedPayload);
Console.WriteLine($"Unprotect returned: {unprotectedPayload}");
```

Purpose strings

- Their purpose
  - Provides isolation within an application.
  - Used to derive keys from the master key.
  - `CreateProtector("Authentication")` cannot unprotect `CreateProtector("Data")`.
- Caveats
  - Do not use user input as the sole source of a purpose.
  - A good purpose would be something like `Contoso.Component.1.0`.

Configuring Data Protection

Configuration Points

Key Stores

Encryption

Key Expiry Policy

Application Name.

Configuring Data Protection

```
public void ConfigureServices(IServiceCollection
services)
{
    services.AddDataProtection()
        .PersistKeysToFileSystem(new DirectoryInfo
        (@"\\server\share\directory\"))
        .ProtectKeysWithCertificate("thumbprint")
        .SetApplicationName("my application");
}
```

Compatibility

Sharing cookies with ASP.NET

- Data Protection
  - Forward compatibility package.
  - ASP.NET Core 1 requires .NET Framework 4.5.2.
  - ASP.NET Core 2 requires .NET Framework 4.6.2.

- Cookie sharing
  - Between Katana based applications and ASP.NET Core.
  - Needs configuration to share the key ring and set a shared application name.

Summary

Q&A

Thank you

Learn. Imagine. Build.

.NET Conf