**What's New in .NET 7 for Authentication and Authorization**

Let’s explore the new .NET 7 features for improving and simplifying authentication and authorization support in .NET applications.

**TABLE OF CONTENTS**

* [**Default Authentication Scheme**](https://auth0.com/blog/whats-new-in-dotnet-7-for-authentication-and-authorization/#Default-Authentication-Scheme)
* [Simplified Configuration](https://auth0.com/blog/whats-new-in-dotnet-7-for-authentication-and-authorization/#Simplified-Configuration)
* [Authorization Policies for Specific Endpoints](https://auth0.com/blog/whats-new-in-dotnet-7-for-authentication-and-authorization/#Authorization-Policies-for-Specific-Endpoints)
* [The user-jwts Tool](https://auth0.com/blog/whats-new-in-dotnet-7-for-authentication-and-authorization/#The--user-jwts--Tool)
* [Dynamic Authentication Requests in Blazor WASM](https://auth0.com/blog/whats-new-in-dotnet-7-for-authentication-and-authorization/#Dynamic-Authentication-Requests-in-Blazor-WASM)
* [Blazor WASM Authentication Diagnostics](https://auth0.com/blog/whats-new-in-dotnet-7-for-authentication-and-authorization/#Blazor-WASM-Authentication-Diagnostics)
* [Conclusion](https://auth0.com/blog/whats-new-in-dotnet-7-for-authentication-and-authorization/#Conclusion)
* [**AUTH0 DOCS**  
  Implement Authentication in Minutes](https://auth0.com/docs)

The release of .NET 7 continues the simplification effort that began with .NET 5. Following this line, the new release provides .NET developers with a few features related to authentication and authorization that make .NET developers' lives a bit easier. Let's take a quick look at those features, which range from simplifications in authentication configuration to the addition of new authorization test tools, to improvements to Blazor's authentication support.

**Default Authentication Scheme**

When you configure authentication for your application, you need to register the authentication service through AddAuthentication(). For example, the following is the code needed to configure JwtBearer as the authentication scheme in an ASP.NET Core Web API:

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options =>

{

options.Authority = $"https://{builder.Configuration["Auth0:Domain"]}";

options.TokenValidationParameters =

new Microsoft.IdentityModel.Tokens.TokenValidationParameters

{

ValidAudience = builder.Configuration["Auth0:Audience"],

ValidIssuer = $"{builder.Configuration["Auth0:Domain"]}"

};

});

Although you are defining just one authentication scheme (JwtBearerDefaults.AuthenticationScheme), the AddAuthentication() method requires that you specify the default scheme to use when it is not specified in your API endpoints (see [this document for more details on multiple authentication schemes in ASP.NET Core](https://learn.microsoft.com/en-us/aspnet/core/security/authorization/limitingidentitybyscheme?view=aspnetcore-6.0#use-multiple-authentication-schemes)).

Starting with .NET 7, the default scheme is no longer required when you define just one authentication scheme. It is automatically inferred by the framework. In practice, you can write the previous code as follows:

builder.Services.AddAuthentication() //👈 no default scheme specified

.AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options =>

{

options.Authority = $"https://{builder.Configuration["Auth0:Domain"]}";

options.TokenValidationParameters =

new Microsoft.IdentityModel.Tokens.TokenValidationParameters

{

ValidAudience = builder.Configuration["Auth0:Audience"],

ValidIssuer = $"{builder.Configuration["Auth0:Domain"]}"

};

});

In case you need to restore the old behavior for any reason, you can disable the new feature using the following statement:

AppContext.SetSwitch("Microsoft.AspNetCore.Authentication.SuppressAutoDefaultScheme", true);

**Simplified Configuration**

One of the recurrent critiques of .NET authentication and authorization is its complexity (see [this thread](https://www.reddit.com/r/dotnet/comments/vttrib/is_auth_way_too_hard_in_net/) and [this one](https://social.msdn.microsoft.com/Forums/en-US/4d3844d5-9175-49ba-8c2f-8257f2c67639/authentication-and-authorization-is-too-complicated-too-abstract?forum=aspdotnetcore), for example). Actually, .NET provides developers with an articulate system for managing authentication and authorization. This system is great for the flexibility it offers, but it may be hard for a beginner to digest all the details. It may be hard even for a more experienced developer who is not used to dealing with the identity features every day.

To overcome this issue, the .NET team started an initiative aiming at simplifying the authentication and authorization configuration. The .NET 7 release introduces the first step in this direction, bringing you a simplified approach to configure ASP.NET Core Web APIs authorization based on access tokens in JWT format.

Check out [this article to learn how to protect your ASP.NET Core Web API](https://auth0.com/blog/aspnet-web-api-authorization/).

If you ever had secured an ASP.NET Core Web API with Auth0, the following code should look familiar to you:

using Microsoft.AspNetCore.Authentication.JwtBearer;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options =>

{

options.Authority = $"https://{builder.Configuration["Auth0:Domain"]}";

options.TokenValidationParameters =

new Microsoft.IdentityModel.Tokens.TokenValidationParameters

{

ValidAudience = builder.Configuration["Auth0:Audience"],

ValidIssuer = $"{builder.Configuration["Auth0:Domain"]}"

};

});

builder.Services.AddAuthorization();

var app = builder.Build();

app.UseAuthentication();

app.UseAuthorization();

Now, you can directly rely on the built-in configuration system to define your Web API's authorization options. In fact, in addition to default authentication schemes, .NET 7 automatically loads the options to configure the authentication service from the new Authentication section of the appsettings.json configuration file. With this new feature, the code shown earlier becomes as follows:

using Microsoft.AspNetCore.Authentication.JwtBearer;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddAuthentication().AddJwtBearer(); //👈 new feature

builder.Services.AddAuthorization();

var app = builder.Build();

All the configuration settings are moved to the appsetting.json file as follows:

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*",

//👇 new section

"Authentication": {

"Schemes": {

"Bearer": {

"Authority": "https://YOUR\_AUTH0\_DOMAIN",

"ValidAudiences": [ "YOUR\_AUDIENCE" ],

"ValidIssuer": "YOUR\_AUTH0\_DOMAIN"

}

}

}

//👆 new section

}

The new Authentication section keeps the configuration settings for any authentication scheme supported by your application, although currently only the JwtBearer scheme is supported.

Notice that now you don't need anymore to call UseAuthentication() and UseAuthorization(). The framework takes care of automatically adding the required middleware to the request pipeline.

**Authorization Policies for Specific Endpoints**

ASP.NET Core allows you to specify policies to make more accurate authorization decisions based on the access token content or other advanced criteria. Usually, these policies are defined globally at the application level and attached to the endpoint definitions the policy applies to.

Take a look at [this article to learn how to use policies to protect ASP.NET Core Web APIs](https://auth0.com/blog/permission-based-security-aspnet-webapi/), for example.

Defining policies globally is pretty good for reuse. However, sometimes you may need just a specific policy for one endpoint. In this case, you can specify your policy directly on the endpoint definition.

The following is an example of how you can use this new feature:

app.MapGet("/api/special-endpoint", () => "A special endpoint!")

.RequireAuthorization(p => p.RequireClaim("scope", "api:special"));

**The user-jwts Tool**

Testing a token-protected Web API could be complex. You can easily use tools like curl or Postman, but you need to pass a valid access token along with your HTTP request. To get a valid access token, you usually need a working identity and access management system at your disposal, such as [Auth0](https://auth0.com/). You need to configure your API with this system and log in by providing valid credentials.

Auth0 allows you to test your registered API using a token issued directly in the [Auth0 dashboard](https://manage.auth0.com/), but you still need to register your application with Auth0.

.NET 7 brings a new CLI tool that helps simplify protected Web API testing: user-jwts. This tool is integrated with the .NET CLI and allows you to generate access tokens in JWT format. The tool is based on the same infrastructure as the [secret manager](https://auth0.com/blog/secret-management-in-dotnet-applications/) and enables your application to validate the generated access token. Basically, the following command allows you to generate a new access token:

dotnet user-jwts create

You can use this token with curl, Postman, or any other HTTP client to make your call to the protected endpoints of your Web API without the need to register your application with an actual identity and access management system.

Beyond the basic usage, the user-jwts tool allows you to customize your access token by specifying the scopes, custom claims, and other properties so that you can also test endpoints with access policies. See the [official documentation](https://learn.microsoft.com/en-us/aspnet/core/security/authentication/jwt-authn) to learn more.

**Leverage Auth0's authentication and authorization services in your .NET applications.**

[**DOWNLOAD THE FREE EBOOK**](https://auth0.com/resources/ebooks/net-identity-with-auth0/?utm_source=auth0_blog&utm_medium=sc&utm_campaign=banner_dotnet_ebook)



**Dynamic Authentication Requests in Blazor WASM**

Blazor WebAssembly uses the Microsoft.AspNetCore.Components.WebAssembly.Authentication package to handle OpenID Connect authentication. This package allows you to statically define at the startup time how authentication will occur. However, there are scenarios where you may need to authenticate users at runtime with a different configuration. For example, you may need to perform [step-up authentication](https://auth0.com/docs/secure/multi-factor-authentication/step-up-authentication) or may want the user to change their account.

In .NET 7, Blazor WebAssembly supports the creation of dynamic authentication requests using the new InteractiveRequestOptions class and the NavigateToLogin() method of the NavigationManager object.

The following example shows the basic use of these new features:

InteractiveRequestOptions requestOptions = new()

{

Interaction = InteractionType.SignIn,

ReturnUrl = NavigationManager.Uri,

};

NavigationManager.NavigateToLogin("authentication/login", requestOptions);

The InteractiveRequestOptions class also provides a way to add custom parameters to the authentication request. For example, consider [Auth0's screen\_hint parameter](https://auth0.com/docs/authenticate/login/auth0-universal-login/new-experience#signup) that allows you to show the signup form instead of the default login form. You can send this parameter to Auth0 as shown below:

InteractiveRequestOptions requestOptions = new()

{

Interaction = InteractionType.SignIn,

ReturnUrl = NavigationManager.Uri,

};

requestOptions.TryAddAdditionalParameter("screen\_hint", "signup");

NavigationManager.NavigateToLogin("authentication/login", requestOptions);

**Blazor WASM Authentication Diagnostics**

Detecting authentication issues is not easy, especially if you are running a WebAssembly code, as it happens with Blazor WebAssembly applications. The .NET 7 release tries to improve its support for authentication diagnostics by introducing the debug level for the Microsoft.AspNetCore.Components.WebAssembly.Authentication library. After [enabling logging for Blazor WebAssembly](https://learn.microsoft.com/en-us/aspnet/core/blazor/fundamentals/configuration?view=aspnetcore-7.0#logging-configuration), you can add the following setting in the wwwroot/appsettings.json configuration file to get detailed information about the authentication process:

"Logging": {

"LogLevel": {

"Microsoft.AspNetCore.Components.WebAssembly.Authentication": "Debug"

}

}

Try out Auth0 authentication for free.[Get started →](https://a0.to/blog_signup)

**Conclusion**

Probably the new features introduced by .NET 7 to mitigate the complexity of authentication and authorization support will not drastically change your life as a developer. However, they confirm the effort Microsoft is making to simplify the use of its development platform. In the area of authentication and authorization, the commitment to reduce configuration complexity with an [initiative](https://github.com/dotnet/aspnetcore/issues/34545) that will continue in subsequent .NET releases looks pretty promising.

I hope that the combination of these and future simplifications will go in the right direction of making authentication and authorization integration into .NET applications less and less challenging.