# Dependency Injection

* + [Life Cycles](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection)-https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-8.0
  + [Microsoft Extensions Dependency Injection](https://learn.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection?view=dotnet-plat-ext-7.0)- https://learn.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection?view=net-8.0&viewFallbackFrom=dotnet-plat-ext-7.0
  + [Autofac](https://autofac.org/)-https://autofac.org/

Autofac is a library used in .NET applications that helps manage dependencies between classes through a concept known as \*\*Inversion of Control (IoC)\*\*. Here's a simple breakdown:

## What is Autofac?

- \*\*Dependency Management\*\*: Autofac allows developers to define how different parts of an application depend on each other. Instead of creating instances of classes directly (which can lead to complicated and hard-to-maintain code), Autofac takes care of creating these instances for you.

- \*\*Components\*\*: In Autofac, classes are treated as components. When you need an instance of a class, you "resolve" it from a container that holds all the registered components.

## How Does It Work?

1. \*\*Registration\*\*: At the start of your application, you register your components with Autofac. This tells Autofac how to create instances of your classes and what dependencies they need.

2. \*\*Resolution\*\*: When your application runs and needs a component, it requests (or resolves) it from the Autofac container. Autofac then looks at the registered components, creates the required instances, and injects any dependencies automatically.

3. \*\*Lifetime Management\*\*: Autofac also manages the lifecycle of these components, ensuring that they are created and disposed of properly, which helps prevent memory leaks.

## Benefits

- \*\*Clean Code\*\*: By using Autofac, your code becomes cleaner and easier to maintain because it separates the creation of objects from their usage.

- \*\*Testability\*\*: It makes unit testing easier since you can easily swap out real implementations with mock ones during tests.

- \*\*Flexibility\*\*: You can change how components are created without modifying the classes that use them, making your application more adaptable to changes.

In summary, Autofac simplifies the process of managing dependencies in .NET applications, leading to cleaner, more maintainable code.

* + [Scrutor](https://github.com/khellang/Scrutor)- Scrutor is a library for .NET that enhances the built-in dependency injection (DI) capabilities of ASP.NET Core applications. It simplifies the registration of services and their dependencies, making it easier to manage complex object graphs. Here’s a concise overview of its key features and functionalities:
  + ## Key Features of Scrutor
  + ### 1. \*\*Assembly Scanning\*\*
  + - \*\*Automatic Registration\*\*: Scrutor allows you to automatically register services from assemblies based on conventions. This means you can scan your assemblies for classes that implement specific interfaces and register them without manually specifying each one.
  + ### 2. \*\*Decorators\*\*
  + - \*\*Service Decoration\*\*: It supports the decorator pattern, enabling you to wrap existing services with additional functionality without modifying the original service.
  + ### 3. \*\*Open Generics\*\*
  + - \*\*Generic Type Registration\*\*: Scrutor supports registering open generic types, allowing you to register a service for all implementations of a given interface.
  + ### 4. \*\*Flexible Registration Options\*\*
  + - \*\*Customization\*\*: You can customize how services are registered, including specifying lifetimes (transient, scoped, singleton) and conditions under which services should be registered.
  + ### 5. \*\*Integration with ASP.NET Core\*\*
  + - \*\*Seamless Integration\*\*: Scrutor works seamlessly with ASP.NET Core’s built-in dependency injection system, enhancing its capabilities while maintaining compatibility.
  + ## Example Usage
  + Here’s a simple example of how you might use Scrutor in an ASP.NET Core application:
  + ```csharp
  + public void ConfigureServices(IServiceCollection services)
  + {
  + // Automatically register all implementations of IMyService in the current assembly
  + services.Scan(scan => scan
  + .FromAssemblyOf<IMyService>()
  + .AddClasses(classes => classes.AssignableTo<IMyService>())
  + .AsImplementedInterfaces()
  + .WithTransientLifetime());
  + }
  + ```
  + In this example:
  + - The `Scan` method is used to find all classes that implement `IMyService` and register them with a transient lifetime.
  + ## Conclusion
  + Scrutor is a powerful tool for simplifying dependency injection in .NET applications, providing features that enhance flexibility and maintainability. By leveraging assembly scanning, decorators, and open generics, it allows developers to manage their service registrations more effectively.
  + https://github.com/khellang/Scrutor

[Application Settings & Configurations](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration)- https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-8.0

[Filters & Attributes](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters)- <https://learn.microsoft.com/en-us/aspnet/core/mvc/controllers/filters?view=aspnetcore-8.0>

# Authentication

Lately I've seen many posts on this subreddit where new .NET developers complained about the complexity of authentication and authorization in the .NET ecosystem.

I am Mike, a .NET developer and founder of a new SaaS startup, [powertags.com](https://powertags.com/), developed entirely in .NET. In this post, I'd like to share with you my understanding of authentication in .NET from a high level, with some historical context added.

The goal here is not to write another "how to" quick guide. Our developers have stated that they're confused by too many guides, because they all looked different. They find it hard to understand what really is going on.

So here I want to show the "big" picture and help point developers to the right directions. The only prerequisites are fundamental knowledge of .NET DI, cookie authentication, the OAuth2/OIDC protocols and their common flows.

This is still a relatively lengthy read simply because there are quite a few topics to cover. But I'll try to make it easy to read. So grab a coffee, and let's get started!

# 1. Know your flavors of cookies

The first few parts of this post mostly discuss server-side projects such as MVC, Razor Pages and Blazor Server. We'll mention JS SPA/Blazor WASM clients later.

Chances are your main web app will use cookies and their claims to authenticate your users. A typical .NET project's authentication DI configuration starts like this:

services.AddAuthentication(options =>

{

options.DefaultScheme = "Cookies";

options.DefaultChallengeScheme = "SomeProviderScheme";

})

.AddCookie("Cookies", options =>

{

//Configure some cookie options like expirations and security

});

Here, we're saying we want to use cookies to sign our users in. The question is, who helps create these cookies and make sure they taste good?

Almost nobody wants to do it all by themselves these days. Instead, this job is delegated to your chosen .NET authentication middleware that will be chained after the config.

**Understand authentication schemes**

Note the reference of "schemes" in the codes above, which act as name identifiers. You will see more schemes referenced later down the chain. Why do we need schemes?

Well, the cookies you saw above represent the "final" cookies that your users ultimately will carry as tickets into your app. Prior to that, your users may have to jump through some hoops.

In case of local authentication via something like ASP.NET Core Identity (discussed later), it's straight forward. They enter usernames/passwords, get authenticated, and cookies made.

But if your user had to go through external OIDC authentication providers, those providers will likely use their own cookies on their own domains to authenticate your users first.

When the said provider redirects your user back to your app, "intermediate" cookies are often created by your middleware to capture the authentication results containing the claims and tokens at the external provider. Some of these cookies may be cleared after.

Finally, the external results are parsed and verified, and your app's own auth cookies are created.

Essentially, different schemes help distinguish different identity providers and their intermediate cookies created by your authentication middleware.

**Why in some codes I do not see .AddAuthentication or .AddCookie at all?**

If an app uses cookies authentication but you don't see these methods, chances are they were simply called under the wrapper of some other identity config helper methods.

For example, when using ASP.NET Core Identity (discussed later), your startup config will contain something like AddIdentity. Such extension methods will configure a set of cookie auth defaults for you, including their default scheme names and cookie options.

When this happens, they almost always also expose option delegates for you to customize/override the defaults, such as Identity's ConfigureApplicationCookie.

# 2. Meet the most versatile .NET OIDC middleware

In section 1, we mentioned that cookie authentications are typically assisted by some kind of middleware, depending on the authentication provider you choose. These days, OIDC with code flow + PKCE protection is the recommended standard for confidential clients.

The all-purpose, universal OIDC authentication middleware for ASP.NET Core is Microsoft.AspNetCore.Authentication.OpenIdConnect.

This middleware works with any dedicated OAuth2/OIDC identity provider. It can be your self-hosted IDP like IdentityServer, or third parties like Microsoft, Google, FB, Auth0, Okta etc.

Let's look at the basic structure of DI config methods when using this setup:

services.AddAuthentication(options =>

{

options.DefaultScheme = "Cookies";

options.DefaultChallengeScheme = "MicrosoftAccount";

})

.AddCookie("Cookies", options =>

{

//Configure some cookie options like expiration and security

})

.AddOpenIdConnect("MicrosoftAccount", options =>

{

//Configure series of OIDC options like flow, authority, etc

});

The example above registers a single OIDC provider identified by the scheme "MicrosoftAccount". In reality, you could have multiple providers, each with a different scheme.

Your sign-in link, which can be generated via some tag helper or component, can specify the provider by their scheme name to "challenge", if your app has multiple OIDC providers.

If some page/controller decorated with just [Authorize] is directly requested, then the DefaultChallengeScheme is challenged.

The challenged user is redirected to the external IDP, performs auth, and is redirected back to your app (redirect URIs are customizable if you don't like the defaults). The middleware takes care of all the hard work in between, and issues the final cookie to sign your user in.

The basic options you need to configure when using this universal OIDC middleware include standard OAuth2/OIDC parameters such as response\_type (flow), authority, client ID, client secret, requested scopes, sign-in and sign-out paths, etc.

These parameters should all be well-documented by your identity provider and their well-known OIDC endpoints. Make sure to grab the correct ones, because some of them have multiple versions.

Additionally, it's a common requirement to perform some kind of claim shaping/mapping via options to make them more standardized for your own app, especially if you deal with multiple providers.

Lastly, the middleware exposes plenty of event handlers such as OnTokenResponseReceived and OnTicketReceived for you to hook into every step of the flow. This allows you to intercept the flow and add your custom logic, such as token handling, if needed.

**The advantage of using** Microsoft.AspNetCore.Authentication.OpenIdConnect: Single package, OIDC provider-agnostic, can customize every parameter and hook into any event you need.

**Requirement:** It is your job to correctly configure the options. Some provider may require some extra options. You should be able to find what you need via docs/Google/SO.

# 3. The "helpful-but-also-confusing" provider packages

Microsoft.AspNetCore.Authentication.OpenIdConnect does require developers to configure IDP-specific options manually. Various other packages aimed to simplify this process, with common parameters such as endpoints hard coded into the libraries.

The problem is that there are multiple providers out there, each publishing/deprecating their own packages. Even for a single provider, multiple SKUs and terminologies can exist.

For example, Microsoft alone has personal accounts and work/school (Azure AD) accounts. Azure AD itself also has Azure AD B2B/B2C. They specify their 1.0 endpoint and 2.0 endpoint, with the latter then rebranded as "Microsoft Identity Platform"...

The end result is a messy web of packages and methods and a whole bunch of tutorials and docs out there, some of them referencing outdated libraries, to perform essentially the same fundamental tasks underneath the surface.

Attempts to "hide away" the configuration options may make it easier to get started short term, but on the flip side, they also make things less customizable.

Real world applications may have non-generic requirements before, during or after authentication. When developers encounter unexpected behaviors and scenarios using these convenience packages, they may have no idea how to resolve them.

**It is important to understand that** **there is no "magic" happening** when you see the AddXYZProvider methods being used to configuration .NET OIDC cookie authentication. They're merely syntax sugars meant to apply a default set of configurations for you.

OIDC is /OIDC. The core protocols are standardized and once you realize this, it should be easy to see why it is entirely possible, and sometimes recommended, to ignore all these packages and simply stick with Microsoft.AspNetCore.Authentication.OpenIdConnect

This is not to say you should never use the helper packages. If your requirements are basic for a specific provider and the out-of-the-box configurations will work for you, by all means go ahead. Just make sure to find the latest versions supported by the provider.

But at the very least you should know how are some of the core parameters configured in them, such as the flow, the default sign-in/sign-out cookie scheme names used, the default callback paths, tokens options etc, which can be seen from the docs or just via Intellisense.

Finally, always keep in mind that you can fall back to the universal package to customize more options should the needs ever arise.

# 4. Where does ASP.NET Core Identity fit into the picture?

So far we've discussed OIDC authentication that involves challenging a dedicated identity provider. Let's backtrack a bit to local auth and local auth + external/social IDP hybrid sites.

**Once upon a time...**

The ASP.NET Core Identity (often referred to as "Identity") has been around since the very early days of ASP.NET. That was a time when external OIDC providers and social logins were not very popular, and most sites only offered local password logins.

Obviously designing a secure user credential store and login/logout mechanism is no easy feat, so .NET developers were encouraged to use Identity for good reasons.

The architectural design of Identity is a bit controversial. The kind of abstraction and extensibility it's built upon have been subject to some criticism. We won't discuss that here.

Out of the box, if integrated with EF Core (the most common and easy way), Identity is a functional local user store that can perform basic password authentications with minimal customization.

This "basic" level of setup would be considered somewhat inadequate today, since by default it does not configure mainstream features such as TOTP MFA and account recovery.

**Scaffolders' shock**

Because Identity was designed in a way such that all of its daily operations are abstracted via UserManager and SignInManager, and users are discouraged from directly manipulating the underlying database, customizing and extending it is a more involved process.

You start by "scaffolding" Identity, which essentially "unhides" the set of MVC controllers or Razor pages that contain the underlying authentication flow logic from the library.

This is where some developers immediately become a bit intimidated: going from "almost no code" to suddenly a few pages of codes to work with.

But the reality is that as long as you're willing to take a deep breath, gather some patience, and spend some time to dive in and just follow the code, you'll likely find them not hard to understand. Watching some tutorial video on Identity will help a lot, too.

In a nutshell, Identity is mostly customized/extended in 3 areas:

1. Extending some base entities such as IdentityUser so that you can add your own properties/fields, including EF navigation properties.
2. Implementing some interfaces and services if you need to add custom logic to the abstracted sign-in, sign-out, account retrieval and persistence methods.
3. Editing the front-end controllers/razor pages to add/update features and business logic to your user authentication flows, such as MFA and account recovery.

These customizations can be a bit tedious at times, but they're all well-documented.

**How does Identity work with external/social IDPs? Do you need to use Identity?**

If your app wants to support local login only or both local and external logins, and you want those local accounts along with passwords to be stored in your own database, then Identity is still the preferred framework to use, because again, dealing with passwords is no joke!

When a project is created via one of those MS quick start templates that enables both Identity and external IDPs (Microsoft/Google etc), your auth DI configuration looks like this:

services.AddDefaultIdentity(...);

...

services.AddAuthentication()

.AddMicrosoftAccount(microsoftOptions => { ... })

.AddGoogle(googleOptions => { ... })

As previously described, these syntax sugars configure a bunch of default cookie auth and OIDC config options for you based on "common use" scenarios.

You're 100% free to utilize the APIs provided by these packages to customize/override some of these options, or even ditch some of them and go with the standard libraries.

To integrate Identity with external logins, the framework's IdentityUser entity, which represents an individual user, can have many IdentityUserLogin entities linked, each representing a specific external login account the user has.

Some apps may enforce one user, one login; while others may allow multiple linked logins. The UserManager class provides built-in methods to look up users by their external logins so the correct user account can be located.

Your UI controllers define logic such as automatic user creation when someone logs in for the first time via an external provider. You can see the default examples from the scaffolded files.

Additional entities such as IdentityUserToken are also included in the framework to help you store user's external login tokens should you need them.

Finally, there are role and claim stores for authorization purposes.

As you can see, besides storing local credentials, **Identity does provide an out-of-the-box framework to help you manage and associate** your users with their external identities.

**But if your app does NOT need local password auth**, is it still worth the effort to utilize and customize Identity, just for the account linking and role/claim features?

The answer for many is probably no. You're free to implement your own user stores and management logic in this scenario. Be careful, however, if your app does need to store user's sensitive external tokens, especially refresh tokens!

# 5. The easier paths – using a managed identity provider

The "managed identity provider" I refer to here are full-service providers such as Auth0, Okta, and Azure AD B2C (note: Azure AD B2C is a separate product from the regular Azure AD).

Each of them is a dedicated OIDC identity provider with its own authority, so using the example in section 2, you can configure them via a single AddOpenIdConnect call. Each provider also provides its own helper SDKs to simply configurations, as discussed in section 3.

They support local password logins. Login pages and user credentials are hosted on their infrastructure, so you no longer need ASP.NET Core Identity for that purpose. They most likely provide built-in MFA support for password logins, too.

At the same time, they allow you to choose a variety of external/social providers such as Microsoft and Google. In that case, they act as the "middle man" to facilitate the flows, redirection, and linking of your users, so you don't have to add them in your own middleware.

Choosing a identity provider is often a business decision that is situational to each app and organization. I will try to summarize some consideration points here.

**Advantages of using a managed IDP service:**

* Less complexity in your own code, no need to use ASP.NET Core Identity
* No need to store sensitive user passwords locally
* Less management UIs to implement, the ones at IDP tend to be quite polished
* Support multiple social providers via a single configuration in your middleware
* Can integrate additional non-OIDC providers (such as SAML, AD etc) that otherwise can be more complex to implement on your own if you need them
* Can secure your APIs/SPAs via JWT tokens (will discuss in next section)
* Potential developer support from your chosen vendor

**Potential disadvantages:**

* Pricing model may not align with your business needs and scaling, this is vendor and app specific.
* Full trust and dependency of your user credentials on a third party. Potential compliance and regulatory complications for certain industries.
* May lack some fine-grained control over the configuration of some specific flows and providers, especially those involving external/social providers, tokens and scopes.

**Honorable mention: Azure App Service "easy auth"**

Some developers often ask what is the absolute "easiest" and "fastest" way to get authentication done in .NET without compromising security.

I'd personally say it's Azure App Service's integrated authentication, aka "easy auth", which can be enabled via a few clicks from the portal. It requires minimal or no code in your app.

Obviously, this requires that you host your app with Azure App Service, which in my opinion works great with .NET. 1-click deploy from Visual Studio is nice, too.

This kind of auth is essentially a gateway-like middleware that sits in front of your entire app. No request can come in without passing through the layer of authentication that is fully managed by Azure.

Of course, as with most "easy" things go, there are limitations. You will have access to the claims and tokens in your app, but you have little control over the authentication process itself. You cannot make anything publicly accessible – it has to secure the whole app.

The best use case for "easy auth" is an internal app that you just want to lock behind a gate, without much requirement for login flow customization and account linking/management, though some do make it work for public sites as well.

# 6. SPAs and web APIs

So far we have mostly discussed server-side cookie authentication facilitated through OIDC middleware and identity providers. SPAs need something extra.

Client side SPAs cannot access databases directly, so most SPA projects will use some web API to retrieve data.

First of all, you still need to show the user a login page and potentially redirect them to external IDPs to perform authentication first, so that you can utilize the resulting cookies, claims and tokens.

Currently, "code flow with PKCE protection without client secret" is the recommended flow for most SPA clients.

**How this part is done depends on the specific SPA:**

Angular and React have well-established JS components/SDKs to handle the OIDC flows.

Blazor WASM uses Microsoft.AspNetCore.Components.WebAssembly.Authentication or some higher level vendor package that wraps it (such as MSAL wasm packages). Microsoft.AspNetCore.Components.WebAssembly.Authentication itself is built upon JS OIDC packages.

In a sense, authentication of JS SPAs in .NET is fundamentally driven by JS itself – this should not be surprising because these front-end SPAs are powered by JS after all. The .NET's role is to provide the supporting components (such as razor components) and routes, facilitate the JS calls and utilize the authentication results.

This means a fair bit of unavoidable abstractions are usually in play when it comes to SPA authentications in .NET. You'll want to carefully follow the MS documentation for each specific type of SPA and analyze the provided quick start templates.

At first glance, things may be a bit confusing because of the abstractions and oftentimes "conventions" used to wire things up. But as long as you understand the fundamental concepts described above, you'll eventually learn to recognize the purposes of these packages and components, and how to customize them if needed by looking up the relevant documentation.

**Understand SPA's inherent trust issues**

Because SPAs such as React and Blazor WASM live on the client side and **inherently cannot be trusted**, the UI logic you implement such as "showing dashboard only for logged in users" is only for the "general" user experience.

In other words, there is a possibility that the UI codes can be tampered with and an unauthorized user can unhide that dashboard. So the key to secure SPAs is at the API level – they may see the dashboard page, but we can deny them access to the underlying data API.

The standard go-to implementation to secure web APIs is JWT tokens. To make this work, there are three major moving parts involved.

**Attaching access tokens to make authorized API requests**

We have two major patterns here. First one is all done on client side. During the prior page-level authentication flow, we save the access and refresh tokens on the client. We'll then attach the access token to the HTTP client on every request against the API.

All SPAs likely do have components/packages to assist in this process. Most current tutorials and Microsoft docs demonstrate this approach.

Token storage and handling at client side adds a level of complexity with potential security concerns. On the other hand, this approach does not require any backend for SPA hosting, so the SPA can be standalone. There is no SameSite requirement.

The second pattern is **backend-for-front, or BFF**. This is a more sophisticated pattern that requires a backend setup. Unless you're willing to roll your own, you will likely utilize either a managed IDP that supports this pattern, or set up IdentityServer (discussed next section).

The BFF pattern is considered more secure because tokens are no longer handled at the client side. Instead, the client uses encrypted cookies to authenticate against the BFF backend, and the BFF in turn acts like a proxy server to access the API on client's behalf.

The BFF backend takes care of token retrieval, storage, cache, refresh, and all that logic in between. The tradeoff is the added complexity to set up and configure the backend itself, as well as SameSite/domain SPA hosting requirements.

**Securing the web API via JWT token**

Because API projects do not need authentication UIs and they simply validate the bearer tokens carried by incoming HTTP requests, they're more straight forward to set up. In most cases, you simply use the Microsoft.AspNetCore.Authentication.JwtBearer package, like this:

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

//Configure various JwtBearerOptions such as IDP authority here...

});

At a minimum, you'll configure some standard parameters like IDP authority, audience and validation parameters.

You may need to add a few more configurations such as back channel token introspection if you wish to use reference tokens, a more secure type of tokens that is subject to per-request validation against the issuing IDP.

Finally, you may need to configure role-based or policy-based authorization policies for your API to properly implement authorization – to validate if an authenticated user has enough privilege to access certain API resources, a topic that is not discussed in depth in this post.

Now we have an important question:

**Who can issue/sign/refresh/revoke tokens and deal with scopes and consents?**

This is really the job of a full-blown identity provider. To implement the standards properly, you almost certainly need to use either a 3rd-party managed identity provider, or host your own IdentityServer.

I say "properly" here, because there are probably guides out there that describe "how to do X without Y". Depending on your particular business needs, some of those methods may work for you. It's all situational.

But if you're trying to build a modern application that involves web APIs and/or SPAs, and you want to implement a standard-compliant OIDC setup that's "by the books", you'll certainly need to use an OIDC identity provider that's "by the books".

ASP.NET Core Identity alone does not deal with token issuance as it is not an OIDC identity provider. It has no concept of OAuth2/OIDC "clients" or "grants". It is an identity store and a set of APIs to interreact with the store.

**The "easy" way is definitely a managed identity provider,** as discussed previously in section 5. They shield all the plumbing work and infrastructure away from you, providing you with just the endpoints to call, and a fully-polished UI to configure your clients.

But if you do not want to use them, then IdentityServer is the way to go.

# 7. What is IdentityServer? Is it really "complex"?

Before we start, here's a little background on IdentityServer for the new comers: IdentityServer is .NET's only native, fully-fledged, OIDC-certified authentication server, created by Brock and Dominick, two expert developers in the security industry.

It used to be completely free (up to IdentityServer4). For years Microsoft's own docs have stated that IdentityServer is the product they recommend for serious OAuth2/OIDC needs, and they still say that today.

I don't know why MS didn't just buy it or something, but let's just say the creators can only maintain it for free for so long, so today it's a commercial product (Duende IdentityServer) with a free community license for smaller companies and non-profits (<$1 million revenue).

The reputation of IdentityServer being a "complex" product and "difficult" to learn mostly arises from the following aspects.

**It does not attempt to hide things from you**

IdentityServer provides several "quick start" templates in MVC or Razor Pages that cover most project types. You don't need to "scaffold" them. They are out there for you to see and customize from the get go. But you have to be willing to dive in.

Again, the codes in them are not inherently complex. As long as you understand the basic OAuth2/ODC flows, you just trace things from challenge to callback, and they will make sense. A video tutorial will help a lot, too.

**You'll probably use** **ASP.NET** **Core Identity along with IdentityServer**

IdentityServer is a dedicated authentication authority, so it needs a user store itself. Instead of re-inventing the wheel, it naturally makes sense that IdentityServer was designed out of the box to integrate with Identity's user store.

This means you may have to customize Identity, too. Everything mentioned previously in section 4 regarding Identity applies here, except that you don't have to scaffold Identity's set of UI controllers/razor pages separately. You only need IdentityServer's own templates.

**IdentityServer + Identity + External/Social identity providers = more hoops**

Previously we discussed ASP.NET Core Identity + external/social login. With IdentityServer added to the mix, it now behaves as the middle man to facilitate the external login flow and redirect, just like a 3rd-party managed identity provider does for you.

Let's say you want to support Microsoft and Google sign-ins via IdentityServer. In your IdentityServer project you will have configurations that look like this (if using the universal OIDC middleware):

services.AddIdentity(...);

services.AddIdentityServer(options => ...)

.AddAspNetIdentity(...);

...

services.AddAuthentication()

.AddOpenIdConnect("MicrosoftAccount",options => ...)

.AddOpenIdConnect("Google", options => ...)

Here, the ASP.NET Core Identity and IdentityServer extension methods will configure their default cookie and config options under the wrappers (which you can customize later). External providers are then registered, each with a distinct scheme, for challenge.

When your user performs a login, a series of redirects and cookies are involved:

1. Your web app, through its own OIDC middleware, challenges your IdentityServer. A login page is shown with multiple external provider choices, each with a scheme.
2. A chosen link is clicked to challenge a specific provider, say MicrosoftAccount.
3. Microsoft authenticates the user via cookies on their own domain at their site
4. The user is redirected back to IdentityServer's callback controller. A temporary cookie is created and parsed to obtain the Microsoft authentication result, then cleared later.
5. IdentityServer looks up the user based on info in Microsoft's claims, or creates a new user if applicable. These actions are performed via the Identity APIs (UserManager etc), against the integrated Identity user store.
6. IdentityServer issues its own cookie, performing a successful user sign-in under the authority of IdentityServer at the IdentityServer site, and redirects user back to the web app that initiated this entire sequence.
7. Your web app parses IdentityServer's authentication result via its own OIDC middleware and creates its own cookies to perform a sign-in at the app site.

Note that the above sequence is a high level view that does not include series of additional round-trip requests (some of them via back channels) between each party depending on the OIDC flow used. But those requests are largely taken care of by the middleware.

Because there are more steps involved, the process may appear more "complex". But when you break down each individual step, they're just following the chain.

You're free to customize this entire process using all the controllers and event handlers exposed to you, including any custom business logic that you otherwise may not be able to insert via 3rd-party managed IDPs.

To show you a real example, when I developed [powertags.com](https://powertags.com/), which requires additional OAuth2 scopes from Microsoft/Google (for calendar access etc), I ran into an issue where Google would display these "sensitive scope" checkboxes that are unchecked by default.

This is a big problem that plagued many developers because users often ignore them and proceed without granting those scopes. This renders the app useless and actions have to be taken after the fact to detect such failures and prompt the user to re-authenticate.

Via IdentityServer and its external controller callback, I could easily add a custom service after the initial redirect from Google to check the tokens against its TokenInfo endpoint.

If missing scopes are detected, I'd redirect the user back to the sign-in page with a message explaining the situation, and prompting them to sign in via Google again. This remediates the issue before the user account is even created, instead of waiting for errors to occur later.

**IdentityServer supports a rich set of flows, options and extensibility**

The biggest reason for most apps to choose IdentityServer over plain Identity is the fact it's a fully-fledged authentication server that supports multiple clients, multiple types of flows, and issues JWT tokens (self-contained or reference) out of the box to secure your APIs.

Client configurations can be hard-coded and loaded in memory, persisted in database via EF integration or whichever other store you want if you implement your own IClientStore.

Similarly, operation data such as signing keys, tokens, consents and other grant types can be store via EF or a custom IPersistedGrantStore.

Want to shape user profile and claims? There is IProfileService. Want to add custom token refresh logic? Try IRefreshTokenService.

The interfaces mentioned above are just some of the many ways you can extend and customize IdentityServer should the default implementations do not fit your needs. In most cases they should, though.

IdentityServer's config extension methods provide plenty of options for you to enable additional features built into the framework, such as token cleanup and state data cache.

The latest community, business and enterprise editions of IdentityServer includes the BFF framework to secure your JS and Blazor WASM SPAs the modern way.

**IdentityServer may had inadequate docs in the past, but things have improved**

Some complained about a lack of IdentityServer documentations in the past, but I believe that situation has definitely improved. These days, Duende has provided lots of docs on their official site. Google/SO returns plenty of search results for common questions, too.

For years, Brock and Dominick have always been responsive on their Github discussion boards and issue trackers, regardless if a paid or free customer raises a question.

# 8. Where to go from here

If you have made it this far, I sincerely hope by now you have a much better understanding on the overall structure of ASP.NET Core authentication as well as the differences and use cases among the various choices out there.

With these knowledge, you should be better equipped to make a more informed decision on the kind of authentication you want to pursue for your project. From there, you should look up relevant documentations and tutorials in a more targeted way.

When you do read them, hopefully things will be clearer for you as you recognize the fundamental elements hidden behind some of the wrappers and syntax sugars.

As for me, I learned most of what I wrote from watching video tutorials on Pluralsight. Instructors such as Kevin Dockx have excellent courses there. I can't thank them enough.

We can't possibly try to dig into everything as developers, and oftentimes we don't really want to. But when it comes to authentication and authorization, I believe it's worth investing some serious learning hours to learn it systematically. Half-baked security sucks.

IdentityServer4 is an open-source framework used in .NET applications for implementing authentication and authorization. It helps developers manage user identities and secure APIs. Here’s a simple breakdown of its main features and concepts:

### What is IdentityServer4?

- \*\*Authentication\*\*: IdentityServer4 allows users to log in to your application securely. It verifies user credentials and issues tokens that can be used to access protected resources.

- \*\*Authorization\*\*: It helps manage what users can do once they are logged in. You can define permissions and roles, ensuring that users only access resources they are allowed to.

### Key Features

1. \*\*Token-Based Authentication\*\*:

- IdentityServer4 uses tokens (like JWTs) to authenticate users. When a user logs in, they receive a token that they can use for subsequent requests without needing to log in again.

2. \*\*Support for Multiple Protocols\*\*:

- It supports various authentication protocols like OAuth 2.0 and OpenID Connect, making it flexible for different use cases.

3. \*\*Single Sign-On (SSO)\*\*:

- IdentityServer4 enables single sign-on, allowing users to log in once and gain access to multiple applications without needing to log in again.

4. \*\*Customizable\*\*:

- You can customize the authentication flow, user interface, and how user data is stored, allowing it to fit your specific application needs.

5. \*\*Integration with ASP.NET Core\*\*:

- It integrates seamlessly with ASP.NET Core applications, making it easy to add authentication and authorization features.

### Example Use Case

Imagine you have a web application where users need to log in to access their profiles and settings. By using IdentityServer4, you can:

- Set up a login page where users enter their credentials.

- Authenticate the user and issue a token.

- Protect your API endpoints so that only authenticated users with valid tokens can access them.

### Conclusion

IdentityServer4 simplifies the process of managing user identities and securing applications in .NET. By providing robust authentication and authorization features, it helps developers create secure applications while focusing on their core business logic.

<https://identityserver4.readthedocs.io/en/latest/>

<https://auth0.com/>

### Auth0

Auth0 is a cloud-based identity management service that helps developers manage user authentication and authorization in applications. Here’s a simple breakdown of what it does:

- \*\*User Authentication\*\*: Auth0 allows users to log in to your application using various methods, including email/password, social logins (like Google or Facebook), and enterprise logins (like Active Directory).

- \*\*Secure API Access\*\*: It helps secure APIs by issuing tokens that applications can use to verify users' identities without needing to store sensitive information.

- \*\*Single Sign-On (SSO)\*\*: With Auth0, users can log in once and gain access to multiple applications without needing to log in again for each one.

- \*\*Customizable\*\*: You can customize the login experience and manage user roles and permissions according to your application’s needs.

### OpenID Connect

OpenID Connect is a simple identity layer built on top of the OAuth 2.0 protocol. It allows clients (like web or mobile applications) to verify the identity of users based on the authentication performed by an authorization server (like Auth0). Here’s how it works:

- \*\*Identity Token\*\*: When a user logs in, OpenID Connect provides an identity token that contains information about the user, such as their name and email address.

- \*\*Standardized Protocol\*\*: It standardizes how authentication is done, making it easier for developers to implement user login features across different platforms.

- \*\*Interoperability\*\*: Because it’s built on OAuth 2.0, it works well with other services and APIs that support this protocol.

### Summary

In simple terms:

- \*\*Auth0\*\* is a service that makes it easy to manage user logins and secure access to applications.

- \*\*OpenID Connect\*\* is a protocol that allows applications to verify users’ identities in a standardized way.

Together, they help developers create secure and user-friendly authentication systems for their applications.

MonogDB- <https://learn.microsoft.com/en-in/aspnet/core/tutorials/first-mongo-app?view=aspnetcore-8.0&tabs=visual-studio>

SQL- <https://sqlflow.gudusoft.com/#/>

<https://newsletter.techworld-with-milan.com/p/how-to-learn-sql>

# Authorization

# ORM

Object-relational mapping (ORM) is like a translator between your object-oriented C# code and the relational database, eliminating the tedious task of writing SQL queries for basic CRUD operations. Using ORM frameworks like Entity Framework, you can **manipulate data as objects in your code, making it more readable and maintainable**. This speeds up development, minimizes errors, and lets you focus on complex business logic rather than wrestling with database syntax.

For **Entity Framework**, you need to know the following:

* DbContext and DbSet for managing database connections and querying data
* Code-First and Database-First approaches for defining data models
* Migrations for managing database schema changes
* Querying data using LINQ and raw SQL
* Tracking changes and saving data

Hi everyone, I'm Daniel Roth, principal product manager on the ASP.NET Core team.

And I'm Sofia Abdala, senior software engineer on the ASP.NET Core team. In this video series, Sofia and I are going to teach you how to build web apps and services using ASP.NET Core.

So Sofia, what is ASP.NET Core?

Well, Dan, ASP.NET Core is a modern web framework for building fast and secure web apps. You can use it to build richly interactive web UIs and powerful backend services. It's open source and cross-platform, and what's even better is that it's completely free to use.

Okay, that sounds cool, but what makes ASP.NET Core different from all the other web frameworks?

Yes, ASP.NET Core is different because it's a complete and cohesive framework for all of your web app needs. It can be used to build small-scale web applications and large global-scale services. ASP.NET Core comes with built-in servers, middleware, and app models to support all of your web app needs. In fact, it powers some of the most powerful apps on the web, like Microsoft 365, Azure, Bing, Stack Overflow, and many more.

Cool! So it's a production-ready framework with everything that I want already built in.

What I love about ASP.NET Core is that I get to write all my code using C#, which is easy to use, full of modern language features, and has a community of enthusiastic developers. You get great tooling using Visual Studio and Visual Studio Code, and with .NET and C# in your developer tool belt, you can write not only web apps but also desktop apps, mobile apps, games, and much more.

Alright, so how can you get started using ASP.NET Core?

To get started building APIs and services with ASP.NET Core, check out my series on backend web development with minimal APIs. And to build your first web app using ASP.NET Core, check out my video series on front-end web development using Blazor.

# Hosting model for .NET APIs 2

Hi! In the last video of this ASP.NET beginner series, we defined web APIs and talked about ASP.NET Core. In this video, we're going to get our hands dirty with ASP.NET Core and write some code. In particular, we're going to gain an understanding of ASP.NET Core's hosting model.

To get started, I'm going to create a new ASP.NET Core project in VS Code. Let's head over to my VS Code instance and create a new .NET project using the C# extension.

We'll select "dotnet new" and then choose the ASP.NET Core empty template. I'll select a target folder for the new project, putting it in my source directory, and I'll name it "MyNewApp." You can call yours whatever you'd like.

Once we've created our new project, we're going to head over to the Program.cs file that was created for our project. In this file, ASP.NET Core apps are configured. It will launch a host that's responsible for starting the application, configuring an underlying HTTP server, and setting up the pipeline for processing requests and responses over HTTP, which is a protocol of the internet.

In summary, it will set up everything we need to start writing our web API. In this minimal hosting model, the WebApplicationBuilder provides us with APIs for configuring the application host. We'll experiment with some of these APIs in future videos in the series, but for now, we can take a look at some of the properties available by typing `Builder.` We'll see that we have access to the host, the environment, details about its configuration, and other attributes that we can enhance.

There are many ways to customize your ASP.NET Core application, but for now, it's important to know that it's possible. Once we've configured our host by writing code in between lines one and three in this file, we can build our web application.

The web application, defined here, configures the request-response pipeline behind the scenes. It allows users to configure route handlers in their application and exposes a `Run` method that we can call to start our HTTP server and initiate our HTTP request processing pipeline.

In this video, we took a look at the code that's generated when we create a new ASP.NET Core web template in VS Code, and we learned a little bit about the hosting model that's responsible for running our ASP.NET Core applications. In the next video in this series, we're going to start building our first web API using ASP.NET Core's minimal APIs.

# Introduction to APIs

Hi everyone, I'm Sofia, a software engineer on the ASP.NET Core team at Microsoft. In this .NET beginner series, I want to talk to you about back-end web development with .NET, specifically focusing on building web APIs with ASP.NET Core.

Before we get started, let's define a few terms. First, what are web APIs? Well, before we can talk about web APIs, we have to talk about APIs. APIs are application program interfaces, and they provide a way for applications to communicate with each other. Web APIs are simply APIs that communicate over the Internet. They typically implement a request-response pattern, where one application, usually a client like your web browser, sends a request to another application, typically a server, which responds with the requested data.

Because these clients communicate with each other, it's important that there is a standardized format they can agree upon for understanding each other's messages. JSON and XML are formats that you might be familiar with for describing how data is communicated between clients and servers. We'll cover all of these concepts in future videos in the series, but first, we want to get familiar with ASP.NET Core.

ASP.NET Core is the technology that we'll be using in this beginner series. ASP.NET Core is .NET's open-source, high-performance, and cross-platform application framework for building web APIs. Because it's cross-platform, it can run on Windows, Linux, macOS, and even in a Docker container. It's pretty neat!

ASP.NET Core has built-in support for building web APIs, and we'll explore all of the features included in ASP.NET Core for building web APIs in this series.

So far, we've defined terms like web APIs and developed an understanding of ASP.NET Core, which is .NET's cross-platform and high-performance platform for building web APIs. In the next video in this series, we're going to start looking at some code and build our first web API using .NET.

# Implementing a Web API in .NET

In the first video of this series, we talked about web APIs and introduced ASP.NET Core. In the last video, we started taking a look at some code and discussed the minimal hosting model. In this video, we're going to bridge the two concepts together and talk about minimal APIs.

We discussed how web APIs follow a request-response pattern: a client sends a request to a server, which processes that request and sends back a response. In ASP.NET Core, minimal APIs allow us to describe how requests should be processed by a server using an entity known as an endpoint. Minimal APIs provide succinct methods for registering new endpoints to handle incoming requests in our web API.

If we look at the code, we’ll observe three important components for defining how a web request should be handled: the HTTP method, the URL route, and the actual handler that executes when an incoming request matches the method and the route. For example, when I call the `map get` method, “GET” is an HTTP method indicating that a client is requesting information from the server. The second argument is the route pattern, specifying that when a client sends a GET request to the root route, the logic in my handler should execute.

To demonstrate this, I can run the application, which will launch a web server listening for requests. Instead of using a web browser to send a request, I'll use VS Code's support for sending HTTP requests via HTTP files. I’ll create a new file, specify the HTTP method as “GET,” and provide the URL for my application. When I send the request, I expect to receive a response from the server with "Hello World."

Next, to fully showcase the power of web APIs, I want to build an API that follows the CRUD pattern for application development. CRUD stands for Create, Read, Update, and Delete, describing how users can interact with objects in an application. We’ll create a sample application that allows users to manage a to-do list.

I’ll start by stopping my running application and clearing the previous code. To represent the to-dos, I'll implement a record type that includes an ID, a name, a due date, and a completion status. To maintain a list of to-dos, I'll declare a variable to store them in memory, acknowledging that this data won't persist outside the application’s lifetime.

Now, let's implement our API to allow users to create, read, update, and delete to-dos. I’ll start by adding a handler for creating new to-dos using the `map post` method. This registers a handler for POST requests to the `/todos` route. When a client sends a POST request, the handler will add the provided task to the list of to-dos and return a response indicating that the task has been created.

For the response, I’ll return a status code of 201, indicating that a resource has been created. Minimal APIs automatically handle deserialization of the request body into a complex type, like our to-do, and they also return appropriate HTTP status codes based on the outcome of the request.

Next, I’ll create a handler to retrieve to-dos by their ID. This handler will respond to GET requests and utilize route parameters to extract the ID from the URL. If a to-do with the specified ID exists, it will return a 200 OK status along with the to-do; if not, it will return a 404 Not Found status.

To get all to-dos, I’ll implement another endpoint that responds to GET requests at the `/todos` route, simply returning the list of to-dos. Minimal APIs handle the serialization of this list into the response automatically.

Finally, I’ll implement an endpoint for deleting to-dos based on their ID, using a DELETE method. This will return a 204 No Content status when a to-do is successfully deleted.

In the next video, we’ll focus on testing our application to ensure that the endpoints behave as expected.

Constructor Dependency

Sure! Here’s a revised version of your transcript without timestamps:

---

What's up, YouTube? This is Dennis Panut for Tutorials. I'm back with another amazing video, and this one is about dependency injection—a topic I struggled with when I started programming. It can seem complex, but once you get it, it’s really powerful.

In this video, I’ll break down the different parts of dependency injection. I recorded this for the Cop Master Classes update, which you can find linked in the description, and also for my updated ASP.NET course.

Part one will cover what the "dependency" in dependency injection is. Then, part two will introduce the super simple constructor dependency injection. After that, we'll look at setter dependency injection, and finally, interface dependency injection. Understanding interfaces is key for this to make sense, but even with a little knowledge, you’ll see why they are so powerful.

Let’s get started! In this video, we’re diving into dependencies. Dependency injection is a powerful concept in object-oriented programming. To understand it, we first need to look deeper into dependencies.

Dependencies refer to objects or components that a class needs to function properly. Think of them as the necessary tools or services a class relies on to perform its tasks. For example, imagine you’re a builder needing tools like a hammer, a saw, and nails. These tools are your dependencies.

Let’s look at a practical example in code. We’ll start with a simple `Hammer` class. The `Hammer` will have a method called `Use`, which simulates hammering nails.

Now, let’s create a `Builder` class. This class will need a hammer, so we’ll define a private `Hammer` field. In the constructor, we’ll set that hammer to a new instance. Here, the builder is responsible for creating its dependencies. If the builder doesn’t have a hammer when going to work, it can’t do its job.

So far, our builder only has a hammer, but we need to bring in more tools, like a saw. Let’s create a `Saw` class with a `Use` method that simulates sawing wood.

Now, we’ll modify the `Builder` class to include the saw. We’ll also need to provide a new saw instance. This means our builder depends on both the hammer and the saw.

Now that we understand what dependencies are, let’s discuss dependency injection—how we inject those dependencies.

Dependency injection (DI) is a design pattern that achieves inversion of control (IoC) between classes and their dependencies. Instead of a class creating its own dependencies, they are provided from the outside. In our example, instead of the builder going out to buy tools, someone brings them to the builder. This allows the builder to focus on building without worrying about acquiring the tools.

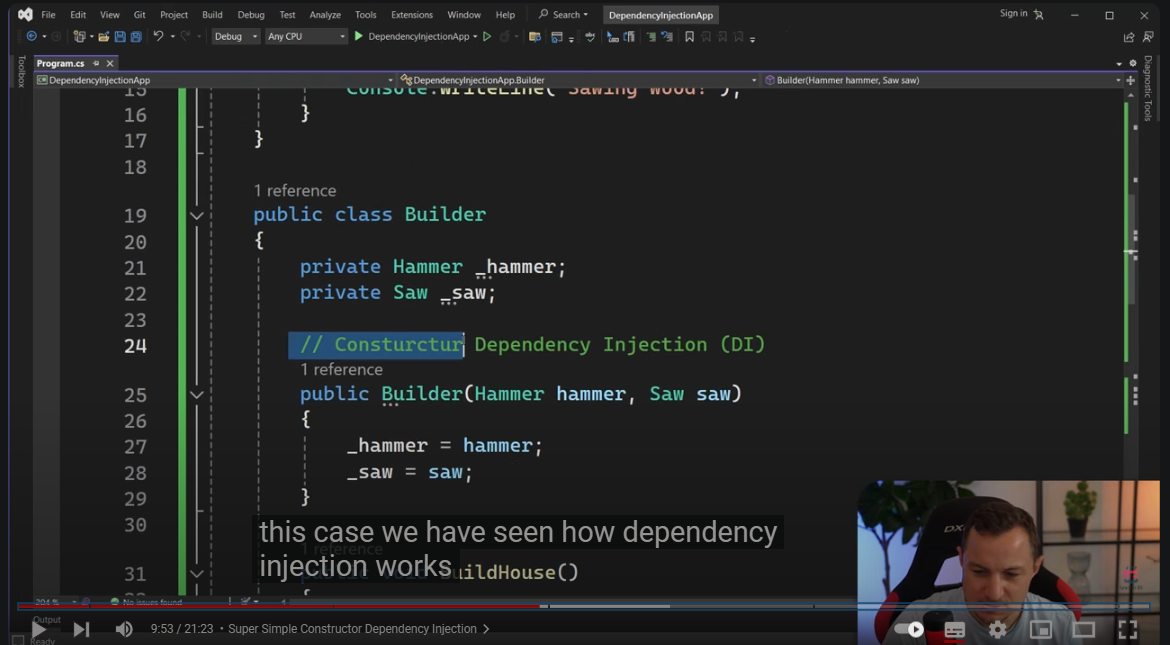
There are different types of dependency injection: constructor injection, where dependencies are provided through a class constructor; setter injection, where they are provided through a property or method; and interface injection, where dependencies are provided through an interface the class implements.

Let’s revisit our builder example to understand dependency injection, particularly constructor injection. Instead of the builder creating its own hammer and saw, we’ll modify the class to require these tools when it is instantiated.

Now, when we create a new `Builder`, we will pass in the hammer and saw as arguments. This way, the builder no longer creates its dependencies but receives them through injection.

Let’s use our builder to build a house, providing the hammer and saw. When we run this example, we’ll see the output confirming that the hammering and sawing occurred, and the house was built.

Now you have a simple understanding of what dependency injection is. In the next video, we’ll dive deeper into setter dependency injection. Thanks for watching!



Sure! Here’s the revised version of your transcript without timestamps:

---

Okay, so we saw constructor dependency injection. Now, let's look at setter dependency injection. How is it different? In this approach, we'll use properties instead of constructors.

For instance, we'll create a property for the hammer and another property for the saw. These properties should be capitalized for convention. With setter dependency injection, we don’t need to call the constructor at all; we can have a default constructor provided by default.

This means we can create a new builder without passing anything. It will create an empty builder object, but the properties for the hammer and saw won't be assigned yet. Before using them, we need to set these properties on the builder.

If we attempt to call the build house method without setting the hammer or saw, we’ll encounter a null reference exception. To fix this, we can simply assign the previously created hammer and saw to the builder’s properties.

Now, we can run our code, and it will indicate that the builder is hammering nails and sawing wood, successfully building the house.

This example demonstrates setter dependency injection, where we set the dependencies after the object is created. The flexibility of setter injection allows us to change the dependencies after the builder object has been instantiated.

In our previous constructor example, we couldn't change the type of hammer or saw after creation. However, with setter injection, we can use polymorphism. For example, we could define a `RubberHammer` class that inherits from `Hammer`, giving us the option to override the default hammer after instantiation.

You can combine constructor and setter dependency injection, enhancing flexibility further.

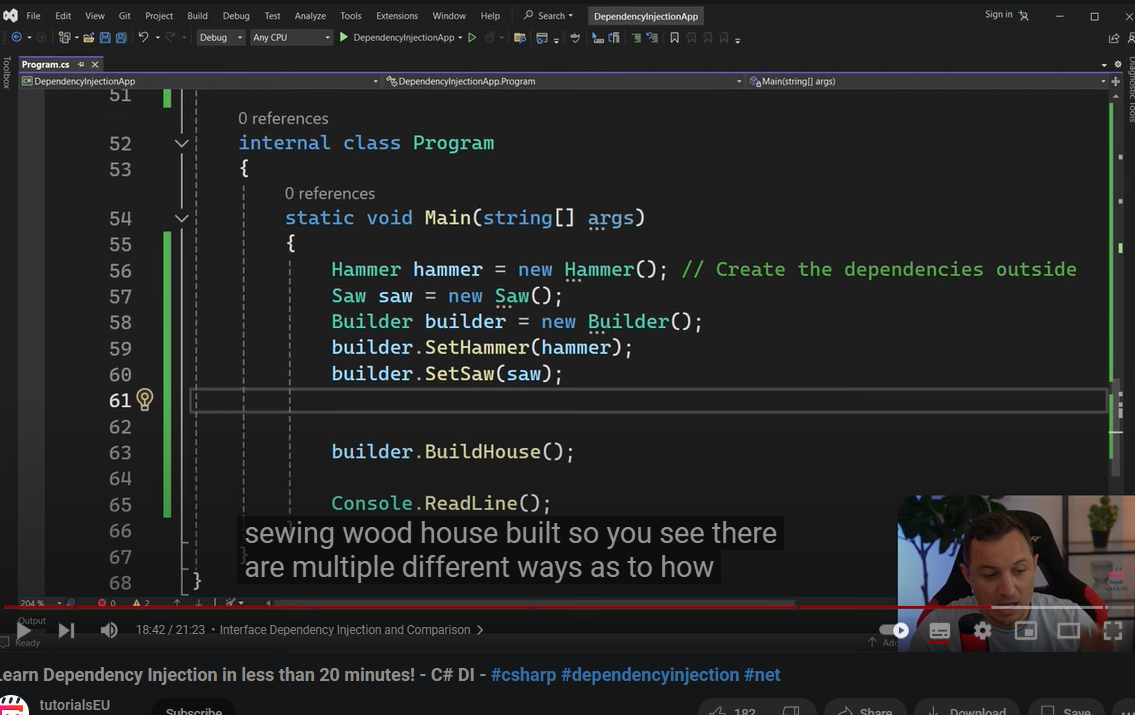
This is just a portion of the Cop Master Class. If you want to learn more about C programming, check out the link in the description below. If there's anything missing from the course, let us know, and we’ll add it.

For those who are familiar with C and want to level up to ASP.NET, there’s another link in the description for the full-stack developer course.

Thanks for your patience! Now, let’s get back to the content. In the next video, we’ll look at interface dependency injection. See you there!

---

Let me know if you need any more adjustments!



Sure! Here’s the revised version of your transcript without timestamps:

---

Now let's look at the third type of dependency injection, which is interface injection. This approach involves providing dependencies through an interface that the class implements. The interface defines the methods for setting these dependencies.

We'll create a public interface called `IToolUser`. This interface will require implementing classes to set the hammer and saw. The builder will now implement `IToolUser`, meaning it needs to set both the hammer and the saw before it can build the house.

We'll define private fields for the hammer and saw, and when using these tools, we'll call their respective methods. Next, we need to implement the methods to set the hammer and saw. By implementing the interface, we create the required methods, which will assign the provided hammer and saw to the private fields.

This setup allows for flexibility, as multiple tool users can implement the interface. For example, besides the builder, a mechanic could also use a saw and hammer, and we would still need to set those dependencies using the same interface methods.

Now that we have the builder set up, we can pass the hammer and saw to the builder using the interface methods. When we run the code, it will indicate that the house has been built.

To summarize, in this case, the builder class implements the `IToolUser` interface, which defines the methods for setting dependencies. The advantage of interface injection is that it allows for external sources to provide dependencies, enhancing flexibility.

Now, let's compare the three types of dependency injection we’ve covered:

1. \*\*Constructor Injection\*\*:

- \*\*Pros\*\*: Ensures all dependencies are provided at creation time.

- \*\*Cons\*\*: Dependencies cannot be changed after object creation.

2. \*\*Setter Injection\*\*:

- \*\*Pros\*\*: Allows changing dependencies after object creation.

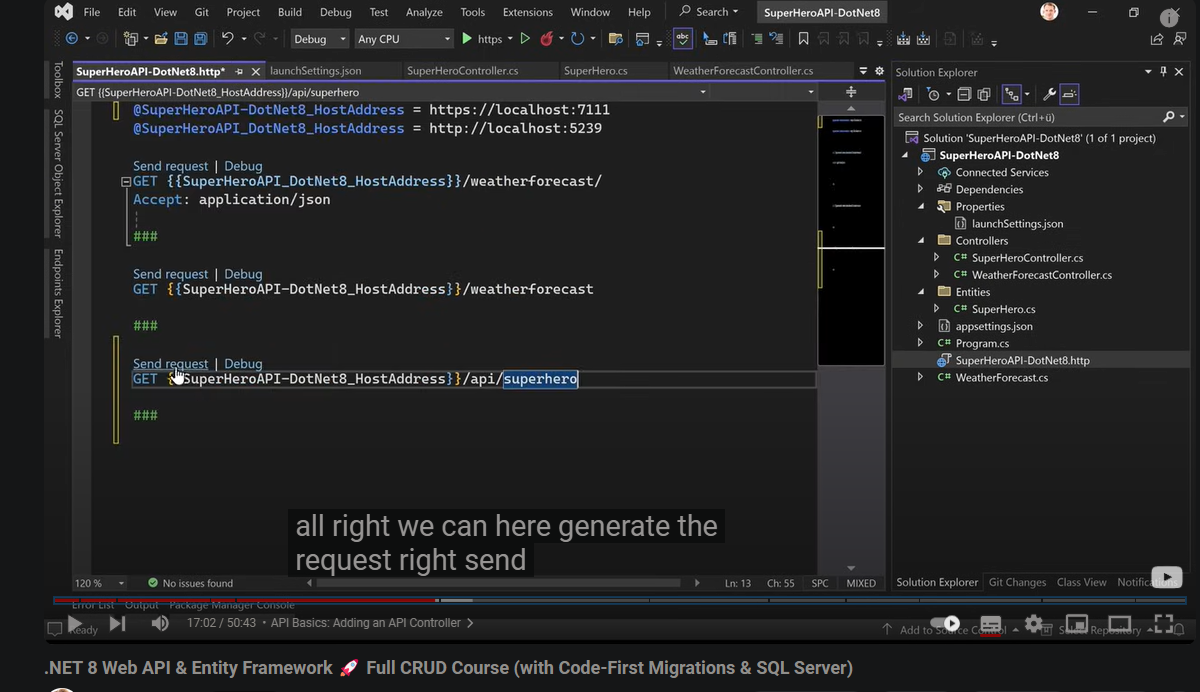
- \*\*Cons\*\*: Dependencies might not be set at creation time, leading to potential null reference exceptions.

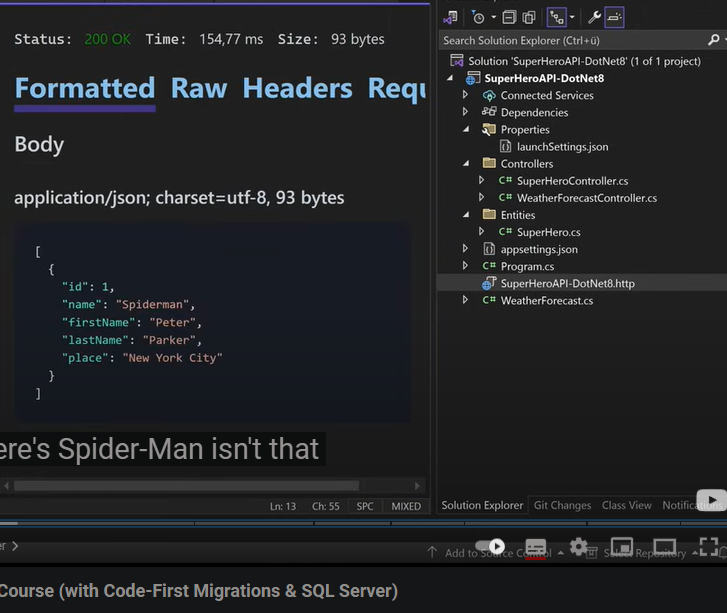
3. \*\*Interface Injection\*\*:

- \*\*Pros\*\*: Provides a clear contract for injecting dependencies through interfaces.

- \*\*Cons\*\*: Requires additional interfaces to define for setting dependencies.

In conclusion, we've demonstrated three methods of dependency injection using the builder example. Each method has its advantages and trade-offs. Understanding these can help you choose the best approach for your specific use case. By leveraging dependency injection, you can create more modular, flexible, and testable code.





Here's the formatted transcript for better readability:

---

\*\*Creating a GET Method for Superhero API\*\*

Let's create a simple GET method that returns a superhero without any database for now. The HTTP GET is the request method we’ll use.

If you're a beginner with .NET 8 and creating a web API, these attributes (like `[ApiController]`) indicate that this class is an API controller, serving HTTP API responses. You can access this controller using specific endpoints, typically by typing the URLs directly into your browser for GET requests.

\*\*Understanding Routes and Launch Settings\*\*

When you check the \*\*launchSettings.json\*\* file, you’ll see our local application URL, which for this example is `https://localhost:7111`. When you go to your controller, you'll find the API route defined.

The route structure follows this pattern: `/api/{controllerName}`. For our \*\*SuperheroController\*\*, this would be `/api/superhero`.

Let’s verify that by starting the application. Swagger opens by default, allowing us to interact with the API.

\*\*Swagger Interface Overview\*\*

In Swagger, you can see example values, although they might not be 100% correct at this stage. You can also find the request URL, which can be copied for direct access.

When you enter the URL in the browser's address bar, it will make the GET call, and you can view the response in the network tab of your developer tools for better formatting and detail.

\*\*Implementing the Superhero GET Method\*\*

Now, let's return to our \*\*SuperheroController\*\*. We’ll create a GET method:

```csharp

[HttpGet("all-heroes")]

public async Task<IActionResult> GetAllHeroes()

{

var heroes = new List<Superhero>

{

new Superhero { Id = 1, Name = "Spider-Man", FirstName = "Peter", LastName = "Parker", Place = "New York City" }

};

return Ok(heroes);

}

```

Here, we're creating a list of superheroes, starting with one superhero—Spider-Man. We then return this list with a status code of 200 (OK), indicating everything is fine.

\*\*Defining Response Types\*\*

To improve documentation and clarify what the API returns, we can define the return type more explicitly:

```csharp

public async Task<ActionResult<List<Superhero>>> GetAllHeroes()

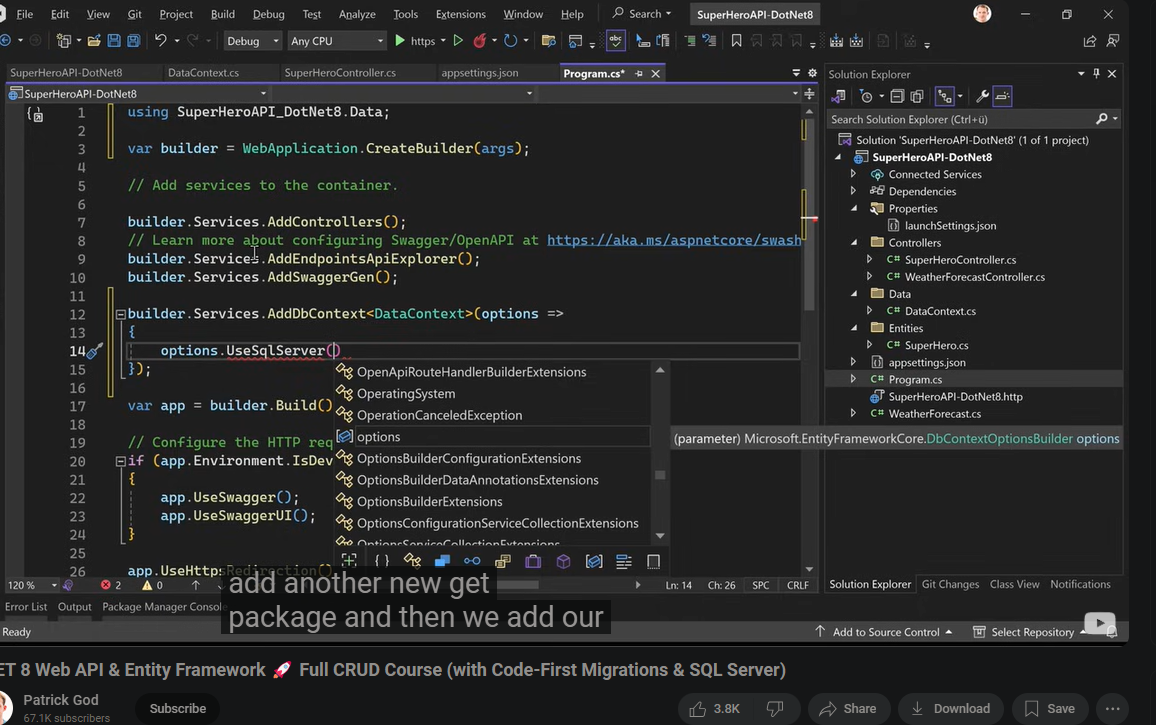
```

After restarting the application, you should see the superhero type reflected correctly in Swagger, along with example values.

\*\*Testing the POST Request\*\*

Next, let’s explore the POST method. Swagger provides a sample object, so you can send requests without needing to input data manually. This allows us to test if the API works correctly, returning Spider-Man as expected, even without a database.

After refreshing the API, you’ll notice the new endpoint available for testing your API calls directly within Visual Studio, which is quite handy.



Here's the formatted transcript for clarity:

---

\*\*Setting Up a Database for the Superhero API\*\*

What I want to do now is use a database because while we can play around without it, I want persistence. So, let's add a database.

1. \*\*Create a Data Folder\*\*:

- I’ll create a new folder called `Data`.

- In this folder, I’ll create a data context.

2. \*\*Creating the Data Context\*\*:

- This data context is for managing data access to the database.

- We inherit from `DbContext`, which is part of Entity Framework Core.

- If you see a red squiggly line, press \*\*Ctrl + .\*\* to open the quick fix menu.

- Choose to install the package `Microsoft.EntityFrameworkCore`, ensuring you get the latest version for .NET 8.

3. \*\*Setting Up the Data Context\*\*:

- After installing, the red squiggly line should disappear.

- We need some boilerplate code, starting with a constructor:

```csharp

public SuperheroDbContext(DbContextOptions<SuperheroDbContext> options) : base(options)

{

}

```

- We also need a `DbSet` for our `Superhero` model:

```csharp

public DbSet<Superhero> Superheroes { get; set; }

```

4. \*\*Connection String Configuration\*\*:

- It’s best to store the connection string in `appsettings.json` instead of hardcoding it.

- In the `appsettings.json` file, we add a `ConnectionStrings` section:

```json

"ConnectionStrings": {

"DefaultConnection": "Server=localhost\\SQLEXPRESS;Database=SuperheroDB;Trusted\_Connection=True;TrustServerCertificate=True;"

}

```

5. \*\*SQL Server Express\*\*:

- If you haven't installed SQL Server Express yet, you can find it online, along with the SQL Server Management Studio (SSMS) for managing your databases.

6. \*\*Registering the Database Context\*\*:

- Next, we need to register our `DbContext` in `Program.cs`:

```csharp

builder.Services.AddDbContext<SuperheroDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

```

7. \*\*Middleware and Services\*\*:

- In the `Program.cs` file, we set up middleware for HTTPS redirection, authorization, and API controllers.

- This will also be where you add other services as needed, like repositories or additional middleware for authentication.

8. \*\*Installing SQL Server Provider\*\*:

- We need to install the SQL Server provider for Entity Framework Core.

- Right-click the project, choose \*\*Manage NuGet Packages\*\*, and search for `Microsoft.EntityFrameworkCore.SqlServer` to install.

9. \*\*Final Steps\*\*:

- After installing the provider, ensure you add the necessary using directive:

```csharp

using Microsoft.EntityFrameworkCore;

```

By following these steps, we’ll set up our database context properly, ensuring our Superhero API can persist data effectively.

Here's the formatted transcript for clarity:

---

\*\*Setting Up Migrations for the Superhero API\*\*

1. \*\*Choosing Migration Tools\*\*:

- We have two options for managing migrations: using the .NET CLI or the tools available in Visual Studio.

- For this project, we'll use the \*\*Entity Framework Core Tools\*\* in Visual Studio, which allows us to run commands to add migrations and update the database.

2. \*\*Stopping the Application\*\*:

- First, we need to stop our application. Open the terminal, close it, and then open the \*\*Package Manager Console\*\*.

3. \*\*Creating an Initial Migration\*\*:

- In the Package Manager Console, we can run the following command:

```bash

Add-Migration InitialBuild

```

- This command will create the initial migration file. It should build successfully, and we can see the new migrations file generated.

4. \*\*Understanding the Migration File\*\*:

- The migration file includes two methods: `Up` and `Down`.

- The `Up` method defines what happens when we apply the migration (e.g., creating the `Superheroes` table).

- The `Down` method specifies what happens if we need to roll back the migration (e.g., dropping the `Superheroes` table).

5. \*\*Updating the Database\*\*:

- To apply the migration and update the database, we run:

```bash

Update-Database

```

- If we encounter an error related to globalization, we can fix it by setting the `Invariant Globalization` entry in the project file to `False`.

6. \*\*Verifying Database Changes\*\*:

- After a successful update, we can open \*\*SQL Server Management Studio\*\* to check the database.

- We refresh the database and see the `Superheroes` table created with the specified columns.

7. \*\*Adding Initial Data\*\*:

- We can manually add an entry (e.g., Spider-Man with Peter Parker) in the database using SSMS.

8. \*\*Modifying the Superhero Controller\*\*:

- Next, we will inject the `DataContext` into our controller. While this might not be the best practice, it simplifies the tutorial.

- We define a private field for `DataContext` and use dependency injection to pass it into the controller:

```csharp

private readonly DataContext \_context;

public SuperheroController(DataContext context)

{

\_context = context;

}

```

9. \*\*Updating the GET Method\*\*:

- We can now modify our HTTP GET method to retrieve superheroes from the database:

```csharp

[HttpGet]

public async Task<ActionResult<IEnumerable<Superhero>>> GetSuperheroes()

{

return await \_context.Superheroes.ToListAsync();

}

```

10. \*\*Testing the API\*\*:

- Running the application, we can use tools like Postman to test our API.

- After making changes directly in the database, we can execute the GET request to see if new entries appear without restarting the application.

11. \*\*Implementing CRUD Operations\*\*:

- We’ll implement all CRUD operations:

- \*\*Get a single superhero\*\* by ID.

- \*\*Create\*\* a new superhero with a POST request.

- \*\*Update\*\* an existing superhero with a PUT request.

- \*\*Delete\*\* a superhero with a DELETE request.

---

Let me know if you need any more changes or additional details!

Sure! Here’s the text without the timestamps:

---

Retrieving Data: How to Get a Single Entry

Now let's just call that get hero with an ID. This ID is of course a parameter, but we also have to specify the ID in the route somehow. What we can do is use the exact same attribute here, the route attribute. So, like route and then in parentheses we specify the route, something like single hero. However, this is not really REST style.

What we can do here is in curly braces specify that this is now a parameter, and our parameter is the ID. These two have to match, or we can just grab that, put it here, and combine these two attributes. If we don't do it like that, for instance, if we remove the route, save that, and restart the application, we will get an error. This is because we now have two methods in the same controller that are using the HTTP GET request method. The route would be exactly the same for both, and our web API has no idea which method it should use.

So please make sure to add the route. We just want to get a single superhero. To do that, we’ll use our context. Here, instead of returning a list, we just say find async with the given ID. If the hero is null, we can return a not found response, or if you want, you could also return a bad request and add a message like "hero not found." This is totally up to you.

Now, this returns the status code 404. A bad request would be 400. If we find the hero, we just return it. Let's save that and restart. Now we have two methods here. As you can see, with one we get Spider-Man, with two we get Iron Man, and with three we get a 404 hero not found.

For demonstration purposes, if we restart the application and add four, this time we will get a 400 hero not found. All the 400s are error status codes. A bad request is 400, 401 would be unauthenticated, and 403 would be forbidden.

Sure! Here’s the text without the timestamps:

---

Data Creation: Using POST to Create New Entries

We want to create a superhero. Let's just copy that, and now we add HTTP POST here. We can say add hero. Now, it's not always the best practice, but it is great for a quick tutorial. We enter the superhero class as a parameter. What would be better is to create a separate request object or a DTO (data transfer object) where you put only the data that you really want to see in your request.

For instance, if you have dates like date created or date updated, or even the ID, the ID is not something we need for creating a superhero. So you would create a DTO, a request object, with just these properties and send this to the web service. Then the web service would map the data from your request object to the actual entity and create this new entry in the database.

Now, HTTP POST add hero superhero hero. What I want is to create that new hero. Instead of all that, we just say context superheroes and then add hero. Now we will get a hero out of that and return all of them. But first, this method will not save that change. It just begins tracking the given entity. You have to call save changes if you want to store this new hero persistently in your database. So then await context save changes async, and in the end, we can actually return all the heroes.

Here, when we refresh, we see the POST method. Now we say generate request. When I send this request, it does not work; the body is not correct. The body is also a good hint. With the GET method, the parameter was used in the URL, but here it comes from the body. This is an attribute telling the web API that this should come from the body, but with that complex class here, the superhero, this is not really necessary. It assumes that it already comes from the body, which is correct.

When I restart this, I see this is our new call. When I open it up, you see the example value and the request body. We also get example values here. We could just hit execute and we're done, or we just enter something. Maybe we have Batman—Bruce Wayne, Gotham City. We execute, and what did I do wrong?

When you have a service, you would not return all heroes; you would return just the heroes. We can either extract this into a separate method or just copy-paste that stuff and put it here. Let's try that one more time, and now we have the option to add Hulk. Let me add another one: Hulk, Bruce Banner, Dayton, Ohio. We execute, and now we got all our heroes. Batman was also successful, so we were able to create Batman.

Now, let's move on to updating data.

Data Update: Modifying Data with PUT Method

Now we create the PUT method. HTTP PUT. We have two options: we could also add the parameter ID here in the URL or use the complete superhero object. Call this method update, grab the ID from that object, and check. We want to find the given hero with the given ID in the database. If we find that hero, we will change the properties or the data and then save that and return again the complete list of superheroes.

Update hero it is. Maybe you will also see in web services that with creating and updating heroes, you would not actually return the complete list of superheroes, especially if you have many entries. You might return nothing or just a success message. This is not always the best way to do it, but I want to show you how to build a .NET web API.

Now, let's try to find the given hero. I will copy that and paste it here. The ID is now in hero ID. Now if the hero is null, we do that, but otherwise we map the stuff. DB hero name is the hero name. We can change the name here to updated request or updated hero. Then we change also the first name and last name. The place is updated hero place.

We have to save these changes, so context save changes async, and again we just return all the heroes. Let's restart the application. Now we see our PUT method. You have to pay attention here. We overwrite everything apart from the ID.

For Spider-Man, this should be ID one. We can just copy that, paste it, and this is Spider-Man. Execute, and we see this is not correct. We could call it Amazing Spider-Man. Iron Man, same stuff. Copy that, paste it here, and execute.

Now, let's take a look at the database.

Removing Data: Implementing the DELETE Method

The last method we have to write is the DELETE method. GET is here for all heroes, GET for a single one, creating with POST, updating with PUT. Now let's remove one hero. We will use DELETE as the HTTP request method. We can call this delete, and we try to find this guy with the given ID.

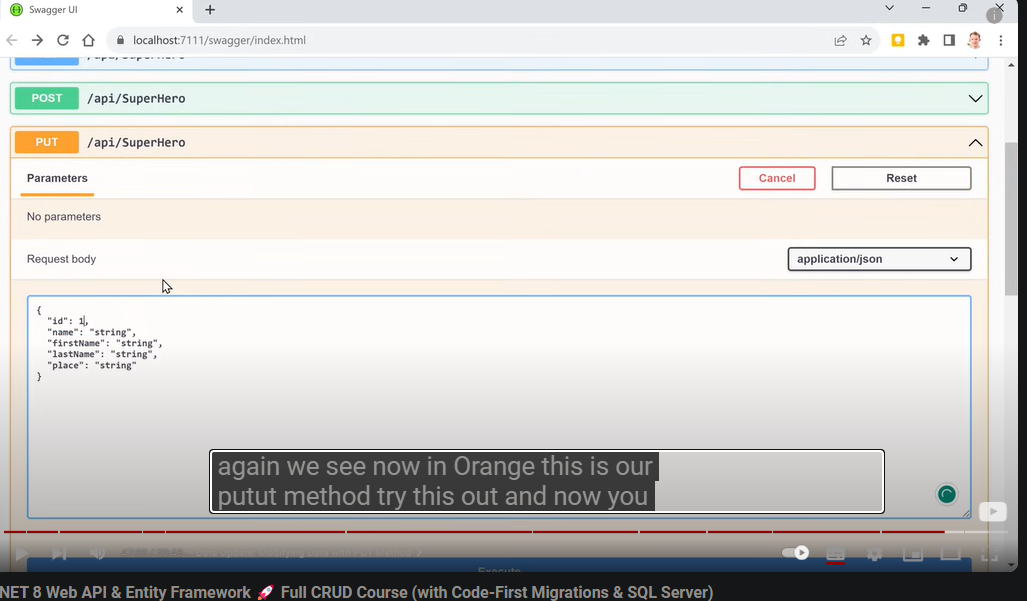
Instead of setting the properties, we say context superheroes remove, and again this method doesn’t change anything. We have to call save changes async to make this persistent. Save that, restart the application, and now in red because it’s really dangerous, try it out.

Execute for five, not found. For four, Al was deleted. Now we only have Amazing Spider-Man, Iron Man, and Batman.

Now in the database, refresh, and that's it.

I hope you learned something. If you did and liked this tutorial, please hit the like button and subscribe to my channel. It would mean a lot to me. Thank you to all my patrons supporting me; this is how this channel can grow with your subscriptions, likes, and support on my Patreon page. Check out the link in the video description below if you want to support me too.

Now, I can only say check out these videos here on the side. There's a lot coming regarding Blazor and .NET 8, so stay tuned. Thank you very much for watching, and I'm really looking forward to seeing you next time!



# .NET 8 Authentication with Identity in a Web API with Bearer Tokens & Cookies

hey friends Patrick here welcome to a

0:02

tutorial about authentication with

0:04

the.net 8 web API and oh my gosh it is

0:09

so much easier to implement

0:10

authentication now with registering for

0:13

an account logging in refresh tokens

0:16

lots of stuff even two-factor

0:19

authentication you only need some Entity

0:22

framework boilerplate stuff and then one

0:25

almost one little line of code in your

0:27

program CS we're using the older

0:30

controllers because I do not want to use

0:32

minimal apis because they look like

0:34

node.js and I don't like node.js and

0:36

express I want my old controllers please

0:38

so we're going to do this in this

Sure! Here’s the text without timestamps:

---

Actually, this feature is already available since .NET preview 4. You see here, May 16th would have been so much better. But anyways, authentication and authorization is what I am talking about in this tutorial. They are excited to introduce map identity API user, the type of the user you want to use. This is pretty much the only line you need. Well, of course, you need some more code, and this is what we’re going to do today in this tutorial. The amazing part is that you can choose if you want to use Bearer token authentication or cookie authentication. You’ll see how this works in a couple of minutes.

I’ve got Visual Studio here, and I’m using the preview edition, 17.8.0 Preview 2, because I’m using the release candidate of .NET 8. If you’re watching this after the full release of .NET 8, you don’t have to worry about that. We want t o create a Web API project. So let’s create a new project and choose an ASP.NET Core Web API. We’ll call this .NET 8 Authentication, configure for HTTPS, and I want to use controllers because I like them. It’s more structured and organized in my opinion.

Let’s pin the solution explorer. We’ve got our Program.cs, WeatherForecast.cs, the typical example. Now we want to create a data context with Entity Framework or using Entity Framework and SQL Server Express. If you don’t have Express installed and want to code along with me, I recommend installing SQL Express. Just Google for it to download it, and maybe even get the management studio for easier database management.

First step again would be a data context. Let’s create a new folder called Data. In here, I’ll create a new item and typically call this DataContext. If you’re already familiar with Entity Framework, you might know that when creating your own application database context, whatever you name it, you have to inherit from DbContext. With Control + Period, we get the Quick Fix menu, and it suggests installing the package Microsoft.EntityFrameworkCore, which we need because we want to use Entity Framework with code-first migrations. However, since I want to use identity for my authentication, I’m actually choosing something else: the IdentityDbContext.

Now, I’ll add a tiny constructor here with one argument of type DbContextOptions. We’ll name our options and choose the base constructor. We are pretty much done with our data context.

Now, let’s register the IdentityDbContext. Going back to Program.cs, we need to register our data context. So, we’ll say builder.Services.AddDbContext and that should be our data context. We also need to add some options here because we need a connection string for our SQL Server database. We could have used SQLite as well, but I typically use SQL Server.

Here, we’ll use the UseSqlServer method. We need another NuGet package for that, which is not suggested here, so let me fix that real quick. We go to Browse and search for Microsoft.EntityFrameworkCore.SqlServer. That’s the one we need. We can use the pre-release version since we’re using .NET 8.

Let’s install that package. I accept, and we also need the design package for code-first migrations. Now, let’s add the connection strings to the appsettings.json. We’ll call this DefaultConnection. In appsettings.json, we’ll add connection strings, and the default connection will be:

```plaintext

Server=.\SQLExpress;Database=.NET8.Authentication;Trusted\_Connection=True;TrustServerCertificate=True;

```

Now that everything is set up correctly, we can add authentication and authorization. We start with builder.Services.AddAuthorization. The next thing we need is to add identity API endpoints. So, we’ll say builder.Services.AddIdentityAPIEndpoints with the identity user and also add Entity Framework stores. We’ll specify the DbContext here.

Now comes the amazing part: we just add another middleware with app.MapIdentityAPI. This is the new magic. Before we can test that, we need to run our migrations. Let’s go to the Package Manager Console and ensure we’re in the proper directory. We can say .NET EF to manage our migrations. If you don’t have the Entity Framework tools installed, you can install them globally.

Once the tools are installed, we can add our initial migration with the command .NET EF migrations add Initial. If everything is correct, we’ll see the migrations folder created with the typical identity tables like aspnetroles and aspnetusers. Now we can say .NET EF database update, and with that, our tables will be created.

Now we can finally run our application. As you can see, there are lots of endpoints here, including register, login, refresh, and confirm email. Let’s try registering and logging in. When we try to register, it already indicates our email address is not correct. If we input the proper password and execute, we get success, and we have our account.

Now we can log in. After logging in, we receive our Bearer token, which expires in 3600 seconds. When we want to test it, we can authorize in our WeatherForecast controller. However, we need to add one more thing to our Swagger configuration for testing authentication.

In Program.cs, we’ll add a security definition for OAuth2. This allows us to specify some things, such as setting the authorization header. Now we have a button to authorize in Swagger. We can log in again, retrieve our token, and use it to access the weather forecast data.

Now, what about cookie authentication? We can refresh and try using cookies. We log in with the same credentials and execute. Instead of getting a token, we receive a cookie. This means you can decide between Bearer tokens or cookie authentication.

I hope you learned something and enjoyed this tutorial on this new feature. If so, hit the like button and subscribe to my channel. Thank you to all my patrons for supporting me. If you want to support me too, check out the video description for the link. Thank you for watching, and I hope to see you next time. Take care!

---

Let me know if you need any more changes!

\*\*What to Learn First as a .NET Developer\*\*

You need to be able to write a console application before you even begin to consider writing a web app of any type. As a beginner, avoid learning Visual Basic .NET or F#. Learning Visual Basic is not recommended unless you want to work on legacy code, and F# is more suited for advanced users.

Once you're ready to move on to web applications, start with ASP.NET MVC. Don’t learn Razor Pages first, as its patterns don't easily map to other frameworks like Ruby on Rails or Node.js. Avoid Blazor as your initial framework since it's overly complex for beginners. Focus on .NET MVC to grasp the core concepts of ***routing, view models, MVVM, Web API, and API controllers***, as these are essential.

You also need to understand HTML, CSS, and JavaScript—these are the foundational building blocks of web development. Don’t get distracted by abstractions over these technologies; prioritize learning how web pages are made.

If you need a single-page application framework, learn React. Avoid Angular and Blazor until you’re comfortable with MVC. For foundational soft ware architecture, familiarize yourself with dependency injection, SOLID principles, and writing testable code. You don’t need strict TDD,100% red-green-refractor at this stage, but understanding how to write testable decoupled code is crucial.

\*\*Essential Books for Beginner .NET Developers\*\*

1. \*\*Pro C# with .NET 6\*\*

2. \*\*Clean Code by Robert C. Martin\*\*—essential for any developer.

3. \*\*Dependency Injection in .NET by Mark Seemann\*\*—a classic that remains relevant.

For databases, focus on SQL Server, as most .NET applications use it. Avoid trying to learn PostgreSQL or MySQL at this stage. Use Entity Framework Code First as your ORM, and familiarize yourself with LINQ. Install SQL Server Express to practice with SQL Server Management Studio.

For databases, you need to learn SQL Server. 95%  of applications using .NET will be in SQL Server  These are very esoteric  options that there will be very niche companies

3:18

that use these with .NET. It is actually quite  rare out there, despite what you might read on

3:23

Twitter. You can use Entity Framework Code First  as your ORM. You might want to experiment with

3:28

Dapper, but you don't need to. Entity Framework  will force you to understand LINQ as well,

3:32

which is extremely important. And you should  also install SQL Server Express on your local

3:38

Windows machine so you can get used to SQL  Server Management Studio, which you'll be

\*\*Types of Companies that Use .NET\*\*

Be aware of the types of companies that use .NET—typically enterprise-level organizations like banks and insurance companies. While there are older small software companies using .NET, many new startups are not adopting it. When job hunting, carefully read job descriptions; .NET Core was rebranded as .NET 5, so ensure you’re not applying for roles that use the outdated .NET Framework.

\*\*Job Search Tips\*\*

"I'm going to hire you for as a .NET Dev,"  the sad thing is this can mean two different

5:04

things. It could mean you're a .NET Core, .NET  5+ Dev, which is great. It's cross-platform,

5:10

it's in the cloud, everyone loves it. But it  could also mean you're a .NET Framework Dev. So,

5:15

check those version numbers very carefully,  and ask your recruiter what they actually use.

5:20

And while we're at it, we have some red flags to  watch out for. If the job description mentions

5:25

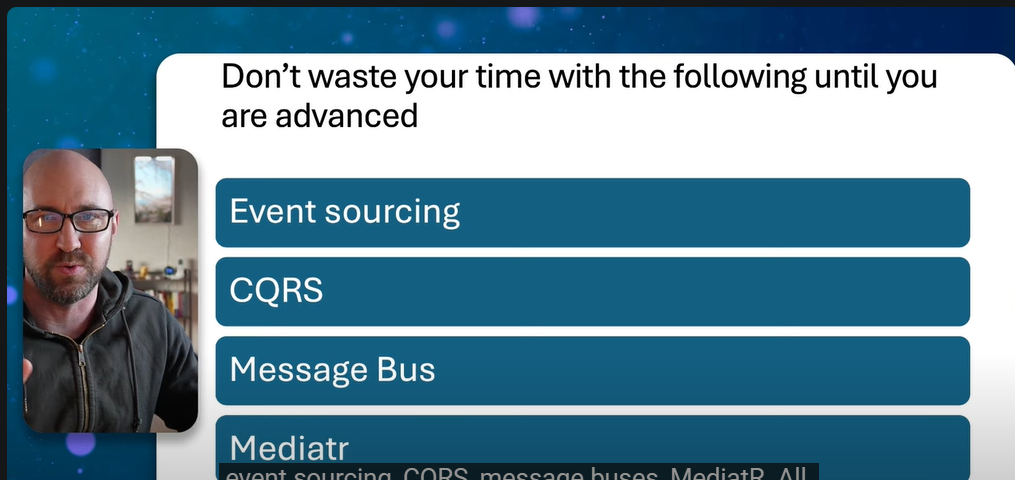
any of these in this day and age, run for the  hills: WebForms, SOAP, WCF, or MSMQ. Just don't

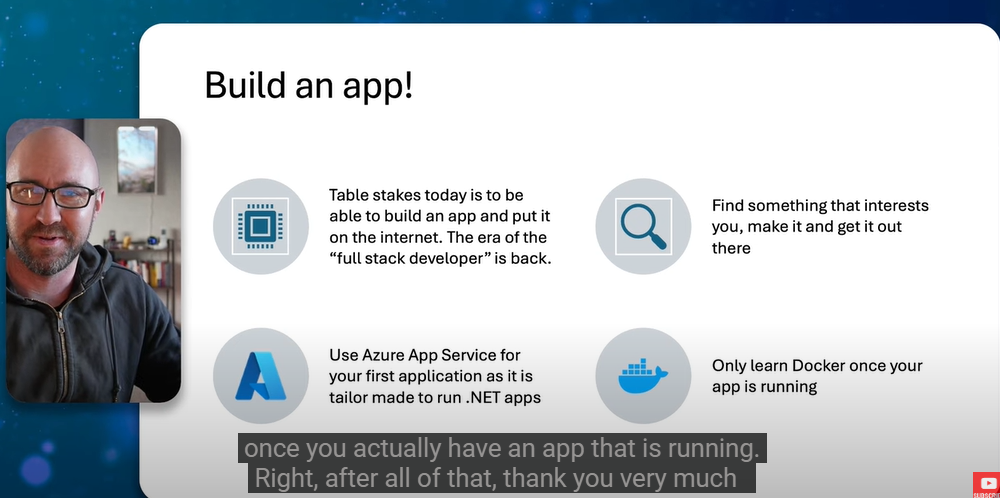
5:36

entertain these jobs if you want to be sane. Now, this is some old stuff, but there's also

5:42

some relatively new stuff that you might want  to study in your early days as a .NET developer.





\*\*What NOT to Learn When Starting Out\*\*

Avoid complex topics like event sourcing, CQRS, message buses, or MediatR until you can build a full-stack web application. These concepts are often over-engineering for most applications.

\*\*Final Advice\*\*

Before applying for a .NET developer job, build an application. You need to be full-stack today—backend, frontend, and database skills are essential. Consider using Azure App Service to deploy your app. Only learn Docker once your app is running.

Thank you for watching! If you have questions, feel free to ask in the comments, and please subscribe for more content. If you didn’t like the video, hit the like button three times to register your feedback with YouTube. Until next time, thanks!

Sure! Here’s the revised text with proper punctuation and grammar:

---

Okay, guys, so in this video, we're going to be talking about models and one-to-many relationships. But before we dive into that, people always say we need to go deeper—Teddy deeper. We're going to get philosophical in this video, and I'm not even joking.

So, why do we even need APIs? What's the big deal about APIs, and why is it that every corporation in America needs them and will pay hundreds of thousands of dollars for you to maintain and build APIs? Well, at the end of the day, an API, in this context, is basically code that sits on top of a database and allows us to interact with databases in a very safe and self-contained way. We don't just want anybody to be able to touch our data in all different types of ways. We are very protective of our data; we don’t want people just grabbing all our information in various ways, so we have an API.

But let's go a little deeper than that. Why is it that corporations want databases so badly? Corporations need databases because they allow us to store data almost like in a filing cabinet but linked together and at lightning speed. If that confuses you a little bit, just let me keep going.

First things first: databases are like a filing cabinet because they allow us to methodically store data in different categories and formats, very similar to a filing cabinet. We have these things called models, which are pretty much each part of this filing cabinet. Within the filing cabinet, we have little pieces of paper—little pieces of data that are stored in individual files within our filing cabinet. So think of the filing cabinet as the database, and the models as little forms or blueprints for forms.

Another important point is that these models and databases are no different from Excel spreadsheets. All a database is, is a fancy Excel spreadsheet that we can link together with foreign keys and primary keys, which I'm going to discuss in a second. But first, let's go ahead and build out our models, then we can start on the primary keys and foreign keys.

Okay, so the first thing we're going to do is right-click on our API folder and create a new folder where we will store our models. Remember, models are just one drawer in the filing cabinet. We are going to have a Stock model because obviously, if we're going to design a stock app, we need a place to store our stocks. We will also create a Comment model.

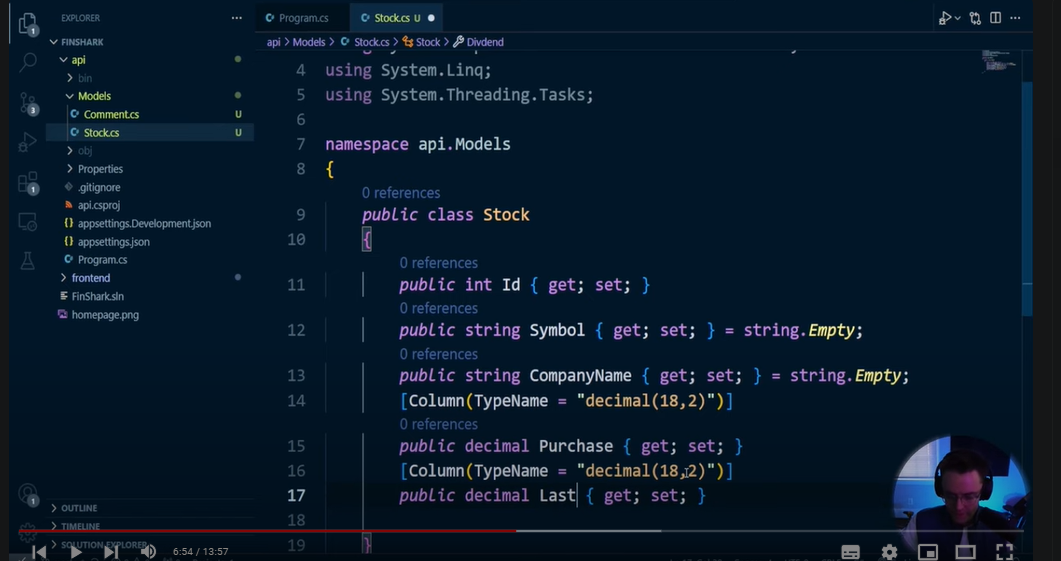
The first thing we want to do is go into our Stock model and type in `prop`, then hit tab. If you type `prop`, you'll see it generates an int property. But what is an int? We can’t just store whatever we want in our properties. If we want to put an ID, we have to use an integer as an ID.

What is an ID? Whenever we create these forms, whenever we create these pieces of data, we need to be able to uniquely identify them. Think about it this way: if you go to a metaphorical dentist office and there are three Teddys in there—my last name is very common too; Teddy Smith could actually be another Teddy Smith in this metaphorical dentist office we are about to create—so we need IDs to ensure that we uniquely identify things.

The next thing we're going to do is create another property. We’re going to have a string, and we will also have a symbol. Why do we need the string? Well, since we're using an ID—a number—to identify an entity, if we change this to a string, that wouldn't be appropriate. We want a number for the ID, and similarly, we want to be able to identify our stock with a string, also known as a symbol. If we put an integer there, that really wouldn’t make sense.

Next, we're going to add a string for the company name. Here, you want to go ahead and put an empty string by using `string.Empty`, or you’ll get what are called null reference errors that will tell you this is empty when it shouldn’t be. So, `string.Empty` is basically a way to ensure that when we store something in the database, we want an empty string rather than a null.

Next, we need to add a decimal for money. We will be dealing a lot with money as software developers, so we need to ensure that when we actually input a decimal, we will only store monetary amounts. We do this by specifying the type and limiting it to 18 digits and 2 decimal places. This will ensure it is a monetary amount.

Next, we can copy this since we are going to have a dividend. This will be the last dividend, so we will call it `lastDiv` to correctly match the Financial Modeling Prep API. If you don’t know what that is, go ahead and check out the React course.

Next, we are going to have a property for the industry. We will make sure that this is also `string.Empty`; otherwise, we’ll get an error. Looking great! Lastly, we’re going to have the market cap. The market cap is the entire value of a company. In theory, if you were to buy Apple, you would need about $1.2 trillion—which I don’t think anybody in the world has right now.

We need to set this to a long because market caps can be in the trillions, and a regular integer will not suffice. We need to prepare for scenarios where someone enters a trillion-dollar market cap, so we will need a long here.

Now, let’s work on our one-to-many relationship. A comment is the perfect example of a one-to-many relationship because it's very easy to understand. If you are going to leave a comment on a profile or a blog post, you will have a one-to-many relationship, and you need to tie them together.

This stock will have the ability to have many comments, and when we are talking about one-to-many relationships, we need a primary key and a foreign key relationship. This is actually why one-to-many relationships and primary keys in databases are so powerful: they allow you to link things together in ways that you can't do in a regular old filing cabinet, as I mentioned.

The way we will link them together is through a primary key and a foreign key. A primary key is the parent. Whenever you think of a primary key, think of the parent. Whenever you see a foreign key, think of the children. A parent can have many children, like a blog post can have many comments, but it can't be the other way around. That's how you identify a one-to-many relationship, and that’s how they are linked together.

There are different ways to link things and different ways to form these patterns that we will talk about later, but it's important to understand that whenever we link database tables—whenever we link our actual cabinets in our database—it's done through a foreign key and a primary key. Just remember: the primary key is like the parent, while the foreign key represents the children. It allows us to have multiple children tied to the parent.

But how do we actually model this? We already halfway modeled it within our Stock model. If we look at the list, a list is a data structure that allows us to have many of something. It’s almost like an array in JavaScript. We also need to form the relationship within our comments.

It may seem unintuitive, but if you think about it, whenever we have the stock, we will have many comments. So if we’re going to have many comments, we need to list them. But how are we going to link them within the comment? The way we link them is by convention.

There are many ways to form a one-to-many relationship, but in our case, we will do so by convention. Convention basically means that Entity Framework Core—if you don’t know much about it—is going to search through our code and form this relationship for us. Before, we had to do this manually, but now we can form this relationship based on convention.

To do this, we can simply create a property called `StockId`. Whenever we create a `StockId`, we will link it to the stock's ID. This `StockId` property will represent the foreign key for our stock entity.

Now, let’s put this in. We can create another property that is actually a Stock object, just like this. We need to make sure to tell our comment model that it’s a foreign key by using an attribute. If you don’t know what an attribute is, it’s a way of adding metadata to our class. We do this by typing the following code:

```csharp

using System.ComponentModel.DataAnnotations.Schema;

```

Then, right above our foreign key property, we can say that this is the foreign key and refer to `StockId`. We will also need to mark it as required, so we will add that attribute as well.

That’s going to be our one-to-many relationship! Now we can move on to our controller. Since we have our models set up, we can go ahead and move to the controller to start building out the API for our stock app. We can use a scaffolded controller to create the routes and actions needed for the API to function correctly. This is an important step in creating a fully functioning API.

So, let’s go ahead and do that, and I’ll see you in the next video!

---

Let me know if you need any further changes!