1. ***Brief snapshot of the features that will be implementing in this app.***

**Web App** with **.Net 8, Entity Framework and Angular 17 and 18.** This is an example of a Developers Connecting app, a social kind of app which has features common to many different types of apps, such as registration and login, we will use **ASP.Net identity** to do that.

Also got **live presence**, so users will be able to see who else is online, **at the same time.**

We can like other users as well, and there's an area where **users can view the list of members that they like and who like them.** We'll have the ability **to edit a profile** and persist those changes to our **back-end database**, which is going to **be our API and using Entity Framework.**

There is also **a system for uploading a photo**, so users can upload their own photos. They can **switch the photo** **and it updates** everywhere in our app.

Also, a **messaging system** so users can message other users and **this is live**. So, if a user is connected somewhere else in the world, then they'll be able to see this message coming in and **there'll be live chats** between two different users.

Also, implementing **pagination** with **caching**, so, we can instantly go to other user’s profiles and it remembers what we've done and if they go to a different page and reset the filters, then we see all of this in place.

1. ***Technical features***

Using **dotnet** (was dotnet core). That is going to form the **back-end of the application** the **API**. Also adding **angular** **framework** to provide the **client-side,** and build a single page application. Display the user interface to the client. Also going to use **Entity Framework** that provide us with the **ability to query our database**. To build this app, we'll be using **HTML bootstrap to provide the styling framework** and by adding a touch of CSS.

*Hint\** it is useful to have the NVM (node version manager), for choosing between deferent node.js versions and not need to uninstall and reinstall.

1. ***Simple walking skeleton***

We're going to **have some data in a database,** then **create an API project** that's going to **fetch the data out of that database,** also **create an angular project** that's going to **query our API and receive the data from our API that comes from our database**, and then we're going to **display it in the client's browser**. It will be really simple initially and then it will add complexity.

* First, implement the **basic API functionality** and have an introductory understanding of using the **dotnet command line interface (CLI)**. A command line interface that we can use to do various things with dotnet.
* **API controllers and endpoints**, and understand the structure of an API controller.
* **Entity Framework**
* We're going to take a look at **the API project structure**. What involves in a .Net project, how our application starts up.
* The **configuration and environment variables** that it uses inside our application.
* Then **commit the code into source control**, which allows us to take effectively a backup of our code and it also contains a version history of the code as well.

In the cmd id we run *dotnet –info* then we will see all the different SDKs we have installed on our computer. When we create the project from the CLI it automatically creates it by using the latest SDK. But if I want to use an older one I need to create this *global.json* as we see below:



Also, we have the list of all the runtimes installed.

Useful CLI commands:

* dotnet -h: for help, where it lists all available commands
* add, new, sln are the most common commands.

For our case the steps are:

Type *mkdir ProfessionalsConnectApp* -> *cd ProfessionalsConnectApp -> dotnet new sln -> dotnet new webapi -h* (for help to see what it includes) ->

**

We are going to use this option.

*-> dotnet new webapi -controllers -n API*, so this will create a folder with the name API and inside

-> *dotnet sln list,* we see there are no projects found so we need to add it.

*-> dotnet sln add API,* and the result



When we open the project and run the server `***dotnet run***`, it says that **listens to a port**, and how the CLI command knows to listen to this port? In the properties folder the ***launchSettings.json***file where we got a number of **different settings and profiles** of what's going to happen when the application starts. The first one in the list is typically the default one.



That's the reason my application started on that particular URL. And it started in **development mode** because that's the setting contained in **this environment variables section.**  In this project we will use postman instead of swagger.

When we **installed the .Net SDK is that dot net has added a self-signed certificate** so that it's **trusted by our browser when we use this URL**, and let’s restart the server. We can see it's running on these two URLs that I've specified there.

**An important file is the Program.cs class**. This is our main entry point for our application. When we run the .net run command this is where it looks into and here is **the App.run method. Then this executes and effectively runs our application.** Inside this program class the first area is the **services area,** where we **register any service** that our application needs inside this part of the file, **then the application is getting built**, and then is the section to **configure the HTTP request pipeline**. The stuff that's contained in the **HTTP request pipeline** is considered middleware and we can put things inside here that allow us to do something with the HTTP request on its way into our API service or on its way out of our API service.

So, if we wanted to check if a user is authorized to access an API endpoint, then we've got middleware such as use authorization or if we wanted to redirect them, then we could use the Https redirection middleware contained.

If we click on API (API.csproj), then this our project file contains the target framework that I'm using.

The API.http file is just another way of testing an API endpoint in Visual Studio 2022.

Some configuration files are the appsettings.Development.json for developer mode and Appsettings.json, which is used whatever mode we're running our application in.

This is a professionals connect app, so it focuses on users. We will create something that's referred **as an entity (model).** When we will use **Entity Framework** into our application, each **entity is going to represent a table** in that database, this folder will be called as entities. So, we create a class in the folder with the name ‘AppUser’.

*Hint \* At each one of our classes goes inside a namespace. A namespace is at logical naming structure and if I wanted to call a class app user somewhere else in my app, I can do so, but only if it's inside a different namespace, otherwise we'll have a conflict. When creating a class, typically you would create the namespace as the physical name of the folder where you've created the class. So, typically this would be API entities as the namespace. And then if I decided for whatever reason, I wanted to create an app user class inside the controller’s folder, I could do so as long as it used a different namespace.*

First part here is its access modifier. How visible should this class be to the rest of my application? When it comes to Entity Framework, we have to make these public properties.

Each one of these properties that we create is going to represent a column in the EF. We're just going to have an ID and a name property effectively.

As of C-sharp 11, we have another modifier that we can use in our properties, and that's to specify required. It means that we cannot create an app user without also specifying the username and this nullable thing is actually part of the nullable flag that we have inside our .csproj.



1. **Entity Framework**

**EF is an object relational mapper (ORM) and its job is to translate our code into SQL commands that update our tables in the database**. In the dotnet 3.5 Framework we used ADO.Net code.

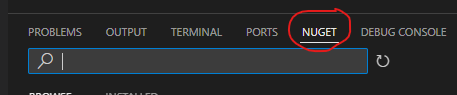
When we add **Entity Framework**, we need to **create an important class that derives from the DB context class that we get with Entity Framework**. This **class acts as a bridge** between our domain or entity classes and the database, and this DBcontext class is the primary class we use for interacting with our database.

AppUser ------ EF (DbContext)---------DB

With **SQLite** we don’t need to install anything, and it simply adds a file into our projects folder for the database. EF use **querying**, **change tracking** it will keep track of changes occurring in our entities which need to be submitted to the database, it also **saves** the database and Entity Framework will execute, insert, update and delete commands to the database, Dbcontext class gives us a save changes method that we can use, it also gives us **concurrency**, it uses optimistic concurrency by default to protect overwriting changes made by another user, it also deals with **transactions** and it provides automatic transaction management whilst querying Or saving data. It includes first level **caching** out of the box, so repeated querying will return data from the cache instead of hitting the database. It offers **built in conventions**, and when I was talking about why the ID property needs to be named, ID is because Entity Framework follows conventions and includes a set of default rules which automatically configure the Entity Framework schema, or the model that we use to create our database. We can also **configure** our entities, and there's ways to configure these so that we can override the conventions if we wish to do so, offers us **migrations**. So, we don't need to create our database manually Entity framework can do this for us.

1. **Use of EF**

From the Nuget package we will install the two Entity Framework packages that we need and when we do install any packages then they get listed inside our .csproj file.



Microsoft.EntityFrameworkCore.SQLite version 8.0.8 (matches .Net runtime version)

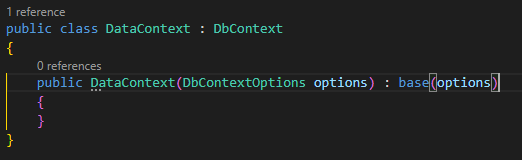
Microsoft.EntityFramework.Design version 8.0.8 (matches .Net runtime version)

1. **Adding DbContext in our class**

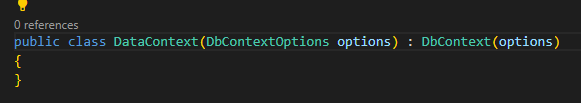
Inside the API create a new folder and call it data, inside the data folder create a new file call this DataContext. **We need to derive from an Entity framework class called Dbcontext**. We need to tell it where to find our database and how to get to our database. **A Dbcontext instance represents a session between the database and can be used to query and save instances of our entities.** We have an entity called the app user and the Dbcontext is a combination of the unit of work and repository patterns.

**We need to generate a constructor that gives our data context some options.** When a new instance of this data context is created then the constructor is executed. In this case it allows us to pass some options to the parent class the DB context and those options will be our database connection string, for example.

Usual way of creating a class with a contractor



And the option to use a primary constructor:



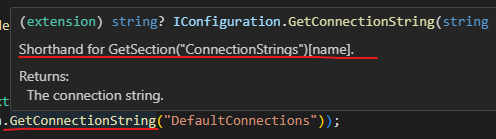
.Net eight comes with C sharp 12 And one of the features of C sharp 12 is this concept of a primary constructor.

Now another thing that we're going to add inside here is we need to **tell our DB context about our entities.** We use a DbSet<> a DbSet can be used to query and save instances of a type of entity. We use Linq queries against a DB set, which are then translated into queries against the database. If we call this prop Users then this is going to be the name of our table in our database.

**It will use the ID as the primary key for the database** if I was to call it something else anything but ID then this would not be valid or Entity Framework would not know which one of these to choose to use as a primary key. **Then we need to tell our application about this DB context class.** Similarly, when we add a new service to our application, we need to **register it in the program class**, then we pass the options, because when we add a service or register a service it is going to use this is via dependency injection and through dependency injection, then dot net is going to create a new instance of our data context class. Then via the constructor, it's going to pass the options that we specify when we register this service. The connection string we're going to get this from configuration.

1. **Create the connection string**

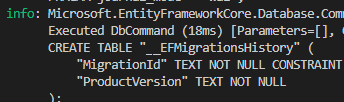
We need to provide this connection string. The **configuration files are the appsettings.json** which **is read from no matter** which **mode** we're using and currently from our **launchSettings.json, we're running in development mode.** Also have an **appsettings.development.json** and when we're running in development mode then this one is read from. The Appsettings.json is read from if we're running in production, in development, in staging and testing. So the **appsettings.Development.json is good place to add our development connection string**. After the logging section use ConnectionStrings which matches the data context registration when we added it to our program.cs class and use the GetConnectionString method.



Then we need to **create a new migration**, which effectively Entity Framework is going to take a look at our code inside the data context. We need to see what you currently have installed by running *dotnet tool list -g* and this will tell you what global net tools you have available. Navigate to nuget.org go to versions for the *dotnet ef* command then click on the one that matches the runtime that we are using, and this is the command that we'll go ahead and use to install it. So, copy this into my clipboard and run it to vs code. Then in vs code we can run *dotnet ef -h* to see the commands and finally **add a migration by typing *dotnet ef migrations add InitialCreate -o Data/Migrations.***

1. **Create database**

Let’s create our database based on the code in this migration class, make sure the app is not running, and run ***dotnet ef database update***. Once that has happened, then we'll see some outputs in our log files. In the terminal we can see that it created a table called *\_\_EFMigrationsHistory*, this keeps track of the migrations that have already been applied to the database.



Then creating the users table, and then inserting a new entry into the *\_\_EFMigrationsHistory* table. Inside our solution explorer we now have a file called *connecting.db* and if I click on this then I can't open my database using VS code in this way because it's either binary or uses an unsupported text encoding, and it's an SQLite file.

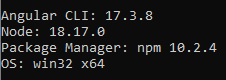
1. **API Controller**

Let's create a **new API controller** so we can create an API endpoint and **receive HTTP requests that** we can then use to respond with the data from our database. We will ensure that we've got a way to query and return data from our database to whatever client is requesting that data. Inside our controller’s folder at the top, we have this **ApiController attribute**, **this indicates that a type and all derived types are used to serve HTTP API responses**. We also give our API controller a route. So now we will create our controller for the Users, and also create a method GetUsers() and return a list of users, so we need access to the DB. **Dependency injection means that the framework is responsible for creating a new instance of our data context class, and then disposing of that class that we inject after we finished using it.** So, when we inject something like the data context into our class, we don't need to worry about its disposal. To use DataContext in our class we need to add it as a parameter to the contractor and then assign it with the readonly field in out class. Then use it in our method to retrieve data from the DB.

1. **Walking skeleton Angular**

Use of **angular CLI**, like .Net CLI it allows us to run our application and create a new angular application. Also going to **review the angular project files**. How **angular bootstraps** the application and displays the results in the client browser. We've got our API set up, we're going to need to make an HTTP request to our API and to do that, we're going to make use of the **angular HTTP client service** to allow us to go and fetch the data from our API. Also, run our angular application over the **Https protocol**, so, we're going to need some **certificates** and we're going to get our browser to trust those certificates. We'll also do is add some additional **packages**, the styling framework we're going to use which is bootstrap as well as some icons from Font Awesome.

1. **Angular Application**

**We're going to install and configure an angular application that's going to consume our .Net API back end. In the **angular docs**, there are the prerequisites and to use the angular framework. Also need an **active or maintenance long term support (LTS) version of NodeJS** and **check the version compatibility guide**. It also tells us what version of node we need to use with what version of angular.

*ng version / node --version / npm --version / nvm version*

To create an angular project the command is: *ng new client*, then to start the app run *npm start.* Some things from angular 16 have changed and for example it used to be modular in its architecture. It also used to have an app.modulets file, nowadays and the default when we create an angular project is that **we create our components as standalone components, they no longer have to be part of a module and that means that instead of adding something to the angular module file, you add it to the angular component that you're using.**

1. **HTTP Request in angular**

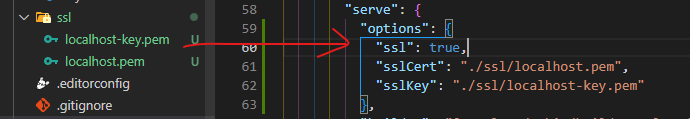
At this version of angular we don’t use the constructor, so in order to inject things into our components angular now provide an inject function that we can use. (inject is importing from angular Core). Now an angular component goes through several stages of its life so, we have events that we can hook into an angular component. After that we make an Http Get request and we're observing the HTTP response that's coming back from our API server. The typical way of dealing with asynchronous code in JavaScript is to use a promise or an async await, similar to how we use it in C sharp. But angular is different it uses observables. When we try our request, the angular app has been blocked by Cors policy though access control.

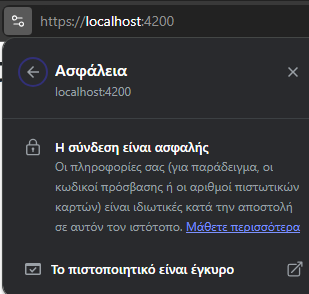
1. **Use Cors**

It's cause from Cross-origin resource sharing, this is a security feature implemented in browsers to restrict web pages such as ours, from making requests to a different domain that served the initial web page. So, our web page is being served by our angular server, localhost 4200, which is a different origin from our API server, which is on localhost 5001. **Cors protects us against various security risks, such as cross-site scripting and data theft, by ensuring that resources** and our list of users is a resource **are loaded from trusted origins as specified by the server**. Then we add Cors as service and middleware, because we need to affect the request on its way back to the client.

1. **Use of mkcert**

Use of a package called mkcert, it is a GitHub repository, this is a simple zero config tool to generate self-signed certificates. We're developing an application, and we use self-signed certificates that are not suitable for use on the internet, we want proper certificates that other browsers can trust. To install it, first install chocolatey (for windows), and then run *choco install mkcert,* in PowerShell as admin. In vs code we need to install it after the installation to the software. The command is *mkcert -install.*  So once that has successfully completed, then we can use a command from mkcert to create certificates with this command *mkcert localhost.* It created 2 new files and we need to tell angular about these files.



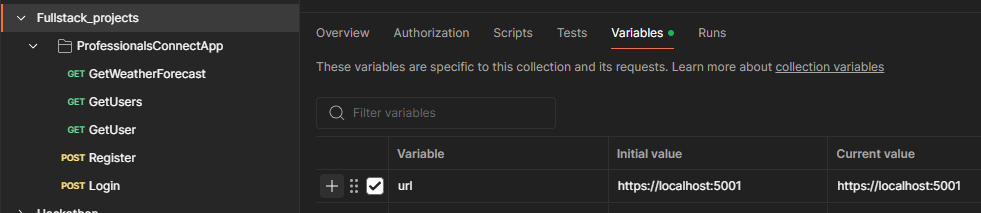
Then we start the app and go back to the bowser, we see it runs on HTTPs, the connection is secure and the certificate is valid. So, mkcert sets up some security features that allows it to trust this certificate. Also, another reason that worked is because the configuration of Cors.

1. **Safe storage of password**

How to **store** something like **a password in our database**. Option one we could store the passwords in clear text, which is obviously a terrible idea. The user obviously enters their password in clear text. **The password goes across the wire or the internet in clear text** but **it's protected because we use SSL security to encrypt the data on the way from the client to the server.** So, we're fine with the users passing up their passwords to us in clear text. Option two is **hashing the password**, which means that we take their password, **add or apply a hashing algorithm to it**, and we end up with a password hash, and we store the password hash in the database. Hashing is one way only and you **cannot calculate from a hash what it was before it was hashed**. So, in theory this is a secure way of storing our passwords. But the problem here is that if a user continues and have a weak password, even time they log in the same algorithm is applied to it, which means we get the same result. So, this is also not a great way of storing a password. Option three is **hashing and salting the password,** a **password salt or a salt applied to a hashing algorithm is going to scramble the hash.** So, if 2 users have the same password, after hashing and salted, it will be 2 different password hashes.The password salt is another randomized string that we pass into the computed hash that then scrambles the hash to make it very different to the original calculated hash.

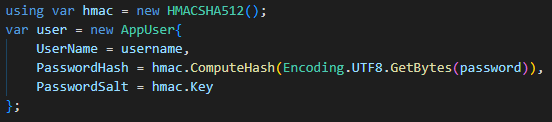
Hint\* When it comes to coding do not repeat yourself (DRY) principle is one of the features that we can use inside C sharp with the concept of inheritance.

Hint\* When we make requests in the postman, we can set some variable and use them in our endpoint in order not to reuse the same like over and over again.



1. **Password hashing and salting**

We're not going to store the passwords in clear text and we do need to encrypt them with HMACSHA12 which we can use to encrypt text or use a hashing algorithm to encrypt text. Using statement releases all resources used by the current instance (here is the hash algorithm class) and it's going to call the (hash algorithm) dispose method. Then we can use hmac.key to salt our password.



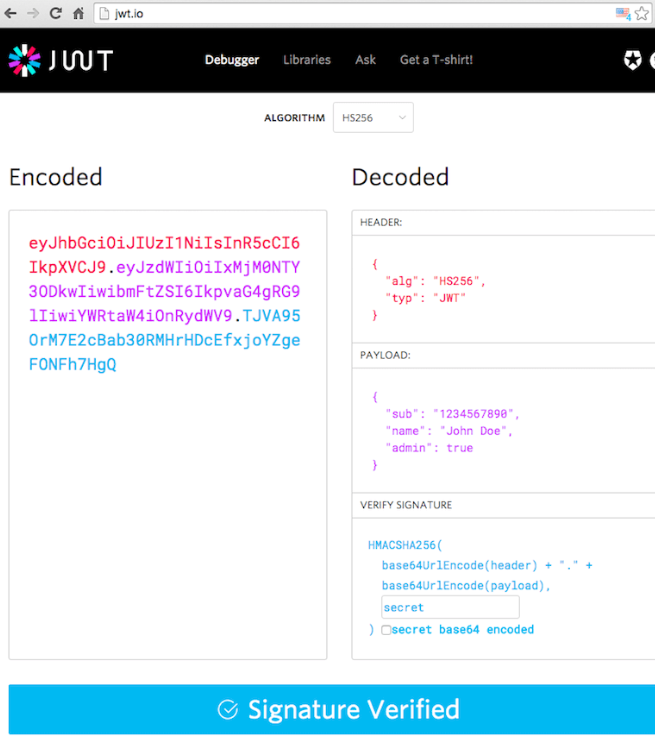
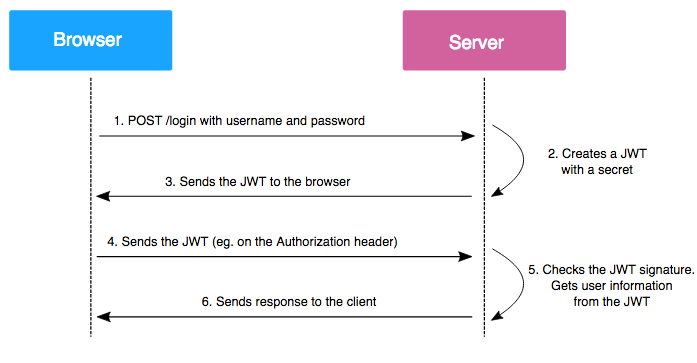
We've got the account controller that's deriving from our base API controller and one of the features of the API controller attribute is auto binding to parameters in our controller. We've got two parameters in our controller and there's a number of different ways we can pass this information in an HTTP request. We can use HTTP headers, the body of the HTTP request, or query string parameters. Depending on the type of information or the types that are stored in here, then the API controller attributes is going to automatically bind to the parameters.

1. **DTO (data transfer object)**

Used to transfer data between two different layers of code. In our case, we just want an object where we can specify that we're expecting a username and password as a parameter.

1. **JSON Web Tokens (JWT)**

APIs don't maintain session states, so each request is independent. Authentication is typically handled using tokens, specifically JSON Web Tokens (JWTs), which are lightweight and easy to send with every request. **JWTs are self-contained, storing credentials, claims, and timestamps**. A JWT consists of three parts: **the header (algorithm and token type), the payload (user claims and credentials), and the signature (encrypted by the server).** When a user logs in, the server validates their credentials and issues a JWT, which the client stores, often in browser storage. This token is sent with every request to access protected resources. The server verifies the token using its private key, without needing to check a database. JWTs are portable, don't require cookies, and are efficient for mobile use.



Creating a service in .NET to generate a JSON Web Token (JWT) for user authentication. When a user logs in, the system validates their credentials and returns a JWT, which is then used to authenticate API requests. The service responsible for creating this token is defined using an interface (`ITokenService`), which abstracts the implementation details. This interface is implemented by a `TokenService` class, ensuring that the method to create the token is correctly executed. The service is registered with a scoped lifetime, meaning a new instance is created per client request. Using interfaces in this manner provides abstraction, decoupling, and easier testing since the implementation details of the service remain hidden. This pattern is common in .NET development, ensuring flexibility and maintainability in code.

The process of configuring a service in .NET to generate a JSON Web Token (JWT) for user authentication, involves creating a service that signs the token with a secret key stored securely on the server. The service uses an injected configuration setting, `tokenKey`, to create a symmetric security key. The key is validated to ensure it meets security requirements. Claims, which are pieces of user-related information, are then added to the token. A `TokenDescriptor` is created, which includes these claims, an expiration date, and the signing credentials. The token is generated using the `JwtSecurityTokenHandler`, which writes the token and returns it to the client upon login or registration. The setup ensures that tokens are securely generated, signed, and can be used to authenticate subsequent API requests. This method offers a secure way to manage user sessions without maintaining state on the server.

To generate and manage JSON Web Tokens (JWT) securely in an application, several steps are essential. First, a secret token key must be established in the application's configuration files. This key is crucial for creating and signing JWTs and must be kept secure to prevent unauthorized access. Typically, the key is stored in the `appsettings.json` file for production environments, separate from development settings, to ensure security. When a user registers or logs in, the application should return only the necessary information, such as the username and token, without exposing sensitive data like password hashes. To facilitate this, a Data Transfer Object (DTO) is created specifically to contain just the username and the JWT. The token service, implemented through dependency injection, is responsible for generating the JWT. It uses the provided user information to create the token, which includes claims like the user's username. The JWT is then returned to the client for subsequent authentication requests. If the token key is too short (less than 64 characters), the application will throw an exception, ensuring the security of the token. Once the token is generated, it can be tested using tools like Postman and decoded using sites like jwt.io to inspect its contents. The encoded JWT includes a header, payload, and signature, where the payload contains the user’s claims and the signature is generated using the secret key. Secure storage of this key is critical to prevent token tampering or unauthorized access.

**Observables** and **Signals** are two approaches in Angular for handling data and state management, each suited for different scenarios.

**Observables:**

* **Asynchronous Data Handling**: Observables are ideal for managing asynchronous operations, like HTTP requests, where data is received over time.
* **Lazy Execution**: They only execute when subscribed to, and can be unsubscribed to prevent memory leaks.
* **RxJS Integration**: Angular uses RxJS to provide powerful operators for transforming data streams.
* **Usage**: Observables are commonly used in services to handle HTTP requests, and you must subscribe to them to receive data. They are essential for dealing with data that changes over time.

**Signals:**

* **Synchronous State Management**: Introduced in Angular 16, signals handle state management in a simpler and more efficient way compared to observables. They are not meant for asynchronous operations.
* **Easier to Use**: Signals are simpler, more declarative, and require less boilerplate code, making them more readable and easier to manage.
* **Automatic Cleanup**: Signals automatically manage their lifecycle, reducing the risk of memory leaks.
* **Performance**: Signals improve performance by only reevaluating when dependencies change, making them more efficient for managing state in Angular applications.

**Comparison**:

* **Observables** are still necessary for asynchronous tasks (e.g., HTTP requests) but can be complex due to RxJS.
* **Signals** offer a more streamlined approach for synchronous tasks and state management, providing better readability and performance.

Overall, **signals** are recommended for state management in Angular, while **observables** remain crucial for handling asynchronous operations.