

Here is the text without timestamps:

Intro

CR operations are extremely helpful to learn when you are just starting out a particular language, so in this video we will cover how to create a new .NET Web API in .NET 8 and we will create read, update, and delete resources in our database. We will use .NET Framework Core and connect to a SQL Server database using a code-first approach, so let's dive right into it.

I have opened Visual Studio and this is what we will use to create our new (link unavailable) Core Web API.

Create a new project

This is the first screen that you will get from Visual Studio, and over here on the right-hand side, you can click on this button to create a new project from the templates that are available to us. On the next screen, you'll be getting an option to choose the templates. In the filters, I have C#, all platforms, and web selected, and from the list, we are looking to create a .NET Web API.

If you browse through this, this is the template that we are looking for. It reads that it is a project template for creating a restful Web API using (link unavailable) Core controllers or minimal APIs. In this example, we are only going to use controllers and we are not going to touch minimal APIs, but this is a template that we are going to use, so we will select that and click on the next button.

On this screen, we want to give a name for the project, and I will give the name as Employee Admin Portal, and that is the same as the solution name. If you want to have a different name, you can clearly change that, and this is the location that I'm going to store my project in, here, so I will click on next.

Over here, Visual Studio is asking us the framework that we want to build this project on, and if I open the drop-down, it will show you the installed versions of .NET SDK and runtime that you have on your machine. For example, I have .NET 5, 6, 7, and 8. Clearly, .NET 8 is the latest version as of now at the time of recording, and I will select that.

If you don't have this option available, you first have to update your Visual Studio or reinstall a fresh copy. After that, I will not touch any of the other settings, so I'll keep it as default and just want to focus on this point that we are using controllers, so that is checked, and after that, I will click on the create button.

It takes a few seconds for Visual Studio to create this template for this project, and it'll look something like this. We have the overview page, which we can close, but the main magic is happening over here on the right-hand side. These are the files and folders that have been created by Visual Studio for our .NET Web API project.

Project Overview

If you see on the top, we have the solution file, which is kind of a structure and holds multiple projects together. It holds one or more projects. At the moment, we only have one project, and that is also set as default, so that we can run this project and test our changes.

After the solution, we have multiple projects, and this is the project that we have. If I double-click on the project name, it will open me the project file, and the project file basically has the target framework, the one that we are using, which is .NET 8, and it also stores the packages that we will install later in this video, in the project file.

Launch Settings

Over here, after that, we have the Properties folder, which has the launch settings.json, and as the name suggests, it's the launch settings. It basically holds all the profiles in which we can run our application. For example, we have the HTTP profile, and we also have the HTTPS profile.

Currently, if you see over here, the play button has the HTTPS profile, and I can switch between HTTP, HTTPS, and IIS Express profiles, but for now, I will use the HTTPS profile. These are the application URLs that my solution will run on. So if you open your launch settings.json, your localhost port number would be completely different.

Controllers Folder

The next thing we have in the solution is the Controllers folder, and this is the folder in which we will add our endpoints, which are basically converted from controllers. So we will add a new controller, and we will add various endpoints or operations like get, post, or delete, as well, inside this controller that we will create.

But by default, we get an example controller, which is the WeatherForecast controller. Then we have the appsettings.json file, and any settings related to the application, for example, the connection string, can go over here. Later in this video, we will come over here and add the connection string to our SQL Server database.

Program.cs File

Here is the text without timestamps:

...then we have the program.cs file which is the main file in our application and this is also the entry file, so the code starts to execute from this file whenever we are deploying and hosting our application and running our API over here. So as you can see, we are building some Services inside the application and we are finally building that. After that, we have the middleware, the HTTP request Pipeline, and that's in this particular order as well. This is an important file and we are going to use this file to add some dependencies related to the DB context, which is an Entity Framework term, but we are going to inject that inside the program, and we are going to later use that DB context that we have injected over here inside the controller, so that we can make a connection to the database.

HTTP File

That we get inside the .NET 8 Web APIs is this file over here, which is the HTTP file. Or if we open it, it shows you that this can be used to make a request to this endpoint over here, and to make this request, I can just run the application and I can also view this from The View toolbar, view other windows, and endpoint Explorer over here. We will get a list of all the controllers and the endpoints that are available inside the controller, and you can also generate this request.

But we already have that, so I'll just delete that over here. Once the application is running, so the browser is running and it's running Swagger as usual, and over here, if I send the request, we are getting the response back as 200 okay. So it's basically making a request to the WeatherForecast controller, and there's this randomly generated weather forecast over here as an array, and we're getting the result back over here. But we still have the option to use Swagger, and you know, execute our request from here. So there's another way added from Microsoft to basically run and test your APIs.

But I'm still going to use Swagger because that's on Open API 3.0, and that's a very good way to test our application, because not only it allows you to test the application, it also shows you the schemas that are available, the post requests, the body that it needs, and the response that it's going to send back. So in our application, when we build the endpoints, we are going to use Swagger to test it.

It's now time to write some code in our application so that we can build our .NET Web API for these employees and adding the details in the database. But before we do that, we have to install some packages that we would need to make a connection to the database, and these are Entity Framework Core packages.

Install Packages

To install the packages, which are NuGet packages, we have to come over here on dependencies, right-click on dependencies, manage NuGet packages, and it'll open up a window. And in the browse section over here, we will basically browse for these two packages and install them one by one.

So the first package is Microsoft.EntityFrameworkCore.SqlServer. I'll copy that and paste it in the browse section, and we are basically looking for some version of .NET 8, because we are running on .NET 8 framework. So this is the latest version, and if you're coming here after a year, and maybe .NET 9 is available, just use the same framework as you have installed.

So I'll click on this package and click on install, and similarly, I will also browse and install the second package, which is Microsoft.FrameworkCore.Tools, and I'll select that and install that as well.

Now that we have installed the two packages, they should end up in the project file, and if I open the project file, we can see that the first package is over here, and the second package is coming over here as well. So the project file also contains the references to the packages that we have installed.

The next thing that we will do is that we will create the entity that we are going to...

Here is the text without timestamps:

The next thing that we will do is that we will create the entity that we are going to save in the database, so that will be our models and what state we want to save our data in the database.

Create Entity

So first of all, we will create a models folder, so right-click on the project over here and we will come to add a new folder, and I will name the folder as models. And inside the models folder, I will create another folder and call it entities.

Entities are basically something with an identity, so this entity will have a primary key as identity, and this is the condition of the employee that we are going to save in the database.

So in here, we will create a new class, so add a new class, and we will call this class as Employee, so I'll call this Employee.cs.

Now in this Employee class, we will add some properties, and these properties will signify the columns that we will have in the table that we are going to create in our SQL Server database.

The first property that we will have will be an identifier and that will be of type Guid, which is a globally unique identifier.

The second property that we will store is the name of the employee, which will be a type of string.

The third one will be the email address, another string type.

The fourth one will be phone, and we will use a string again to store something like plus and spaces in between.

The next one will be the salary, and this will be of type decimal.

Now you can see that we are getting some warnings and some green squiggles over here that it says non-nullable property name must contain a non-null value when exiting the Constructor.

So basically, it's asking us that this property of type string is that a required field, and if it is, it should be exiting, you know, from The Constructor as a non-nullable property.

But we want to make some properties as required fields and some properties as optional.

So basically, there's another way, a new way to do it in C# 12, which is from the required attribute.

So inside the property over here, we can use the required keyword, and this will make sure that when we are about to initialize this employee object, we have to specify a value for name, otherwise it will throw us a compilation error.

And we won't be able to initialize an employee object without the name property.

And if your business is requiring the same thing for uh the email address and the phone, that's totally fine as well.

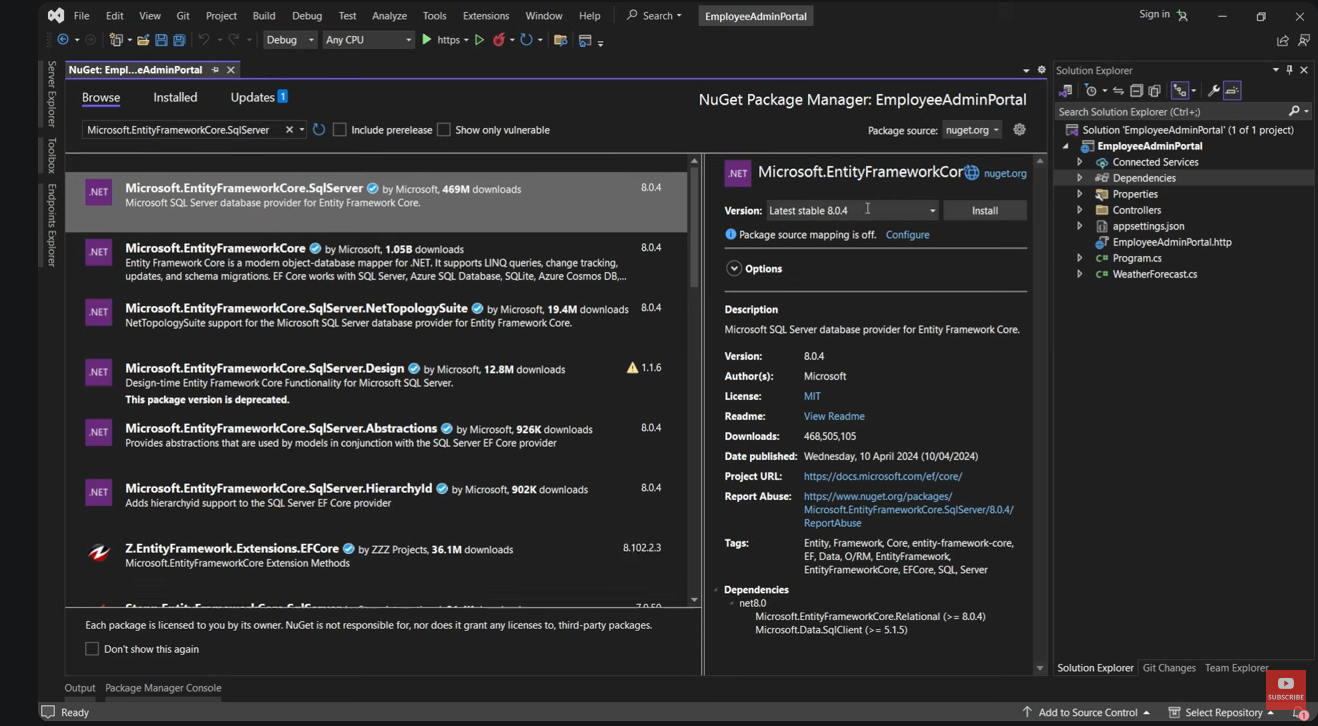
You can add the required attribute to email and phone as well.

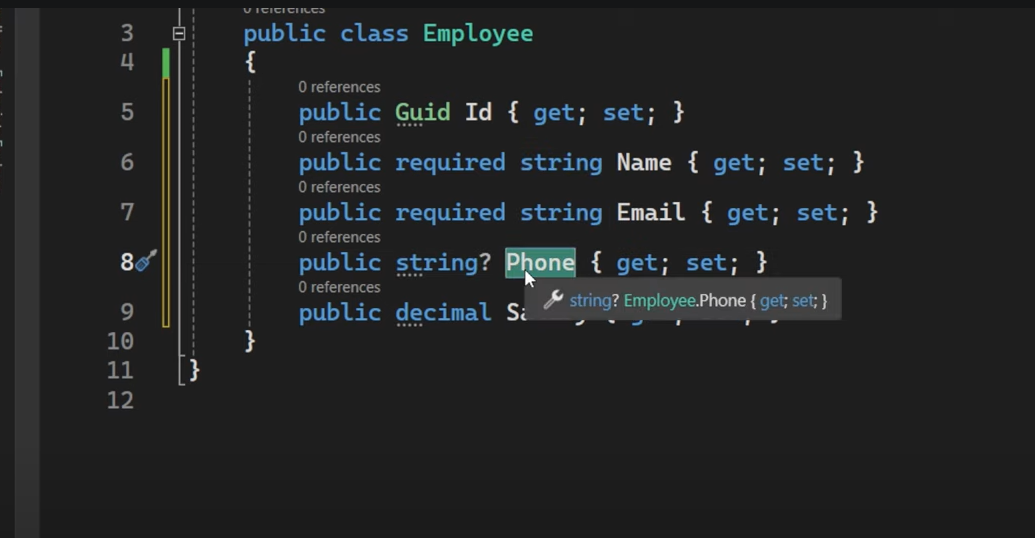
But just for the sake of this example, let's say the phone was not required.

So that will make it a nullable property, that means the phone can be a string or a nullable.

And in that case, we have to use a question mark to make this as a nullable property.

So now we have defined our employee object.





Here is the text without timestamps:

Time to create the DB context class, which is also required by Entity Framework Core to make a connection between the database and the application.

So to create the DB context class, I will first create a folder for that on the project, right-click, add a new folder, and I will call it data.

Inside the data folder, I will create a new class, and I will call that class the DB context, so I'll say ApplicationDBContext.cs.

This ApplicationDBContext class will basically inherit from another class, and that will be the DB context class, so DB context, and this class comes from the package that we installed, which is Microsoft.EntityFrameworkCore.

So I'll select that, and you will see the using statement on the top over here as well.

Then we will create a Constructor for this class, so Ctrl+R to create a Constructor, and we will pass the parameter DB context options, so DB context options of type, and the type is the DB context itself, so ApplicationDBContext.

After that, we will give a name for this, so let's say options, and I'll minimize this, and we'll also pass these options to the base class, so base also pass the options over here.

You can write the Constructor manually as well, but there's another way to do it. If you come over here and imagine this is not there, you can click on the class, press Ctrl+, or Ctrl period, and you can see that Visual Studio is suggesting us to generate the Constructor ApplicationDBContext with the options parameter.

If I select that, it's basically giving us the same thing, along with the type of the DB context options, which is optional for now. If we have multiple DB contexts, only then we have to give the type of the DB context over here.

So for this one, anything is fine. You can write this manually or you can generate it from the intellisense.

The next thing that we are going to do in this DB context class is that we will add a property for the collection that we are going to store in the database.

That means we are going to store employees in the database, so we want to have a DbSet type of the employees.

So I'll create a property, and the shortcut is "prop" + tab to create a property.

Now the property type will be a DbSet, and it takes a type, and the type is the Employee class that we created.

So the Employee entity that we created, and then the name of this will be Employees.

We usually have the plural form of the entity that we created over here.

The next thing that we are going to do is to add a connection string over here, and we add a connection string in the appsettings.json file.

Here is the text without timestamps:

Add Connection String

String inside the app settings.json file, so after the allowed host section, I will come over here and create a section for connection strings, and it's basically an object of type Json.

In this, I will add a key-value pair. I can name my connection string anything, so let's say this is the default connection.

And after that, I'll press colon, and inside string, I will give the value of the connection string, and this can be a little different for you over here because this is what we have installed, and you know this is the SQL Server Developer Edition that we have installed, and this is SQL Server Management Studio that is used to view the SQL Server instance that you have installed in your system.

So to check if you have the correct version installed, you can come over here inside SQL Server Management Studio, you can try connecting to the server that you have, or you should install it.

So this is the connection, the server name that I have installed, and this is Windows authentication.

Just to check if this is the correct server, I can connect it, and you can see that it instantly connected, and in the databases over here, I can see a lot many databases over here.

So once you know what the server is that you have installed in your Local Host, now you can come over here inside the connection string and define the server.

So the server is server is equal to, and then the value of the server name, which in my case was this, and in your case, it could be something different.

Once you have defined the server, we'll put a semicolon and define the database.

So database is something that you want to create, because the database does not exist at the moment, and this is the code-first approach, because we have written code for the employees collection first, and now we are about to create the database for it.

So this could be anything, but for this example, I'm writing that this will be the EmployeesDB.

After that, we have to write two things, which is the trusted connection.

Trusted connection is equal to True is equal to true, and the last thing is trust server certificate is equal to true as well.

So once you have defined all of these four things inside the connection string, we have done our work inside the app settings.json.

And we are going to use this connection string inside the program.cs file.

In the Solution Explorer, we will open program.cs file, and we have a few services that we are injecting in the Builder over here.

But we are going to also inject the DB context class that we created, so that we can use it inside the controller or any other class inside our application.

And the way to do it is using the Builder class.

So the Builder object, do services, do add DB context, and this is the method that we are now getting.

Add DB context of type, and the type that we have is the ApplicationDBContext.

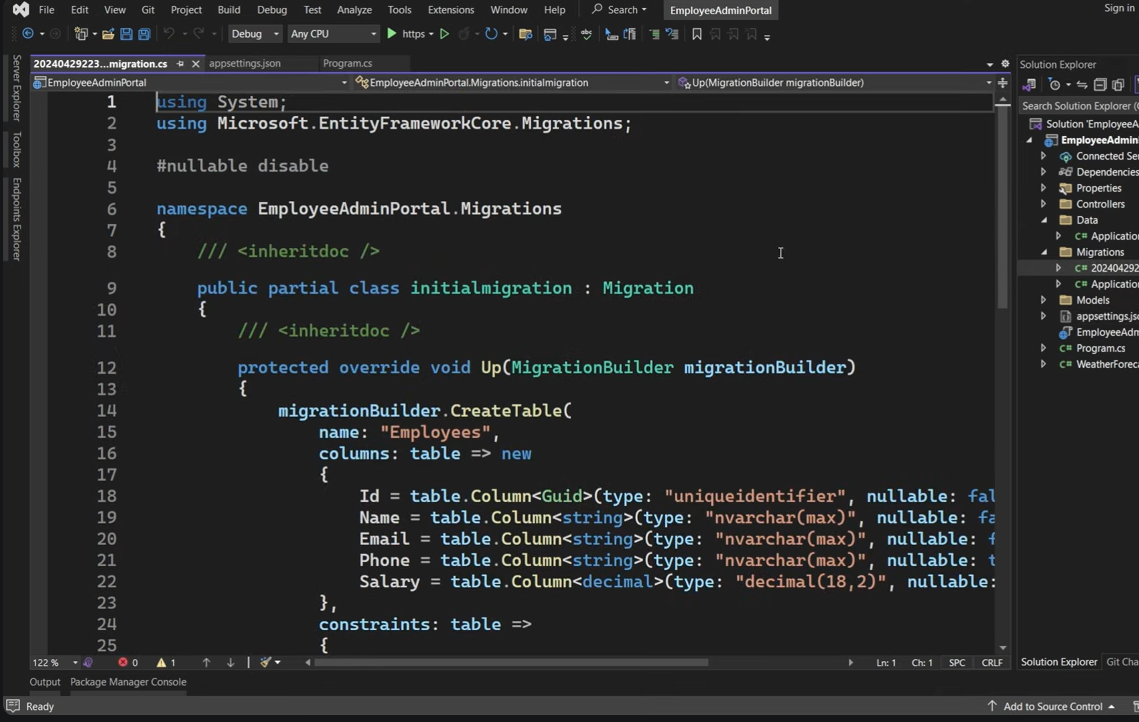
So over here, we will write ApplicationDBContext.

Press Ctrl+. period, and we will use the Employee Portal.Data, because that resides in the Data Portal over here.

After that, we are going to use the options parameter.

So options dot, we have the UseSqlServer method, and I'm going to shift this over here in the second line, so that it's more visible.

So Use SqlServer.



Here is the text without timestamps:

SQL server has or takes different types of parameters, and one of them is the connection string, and it's of type string.

We are going to fetch the connection string from the app settings that we have, and the way to do it is using the Builder again.

So Builder, do configuration, do get connection string, and we want to get the connection string with a certain name.

If I open the app settings again, this is the name of the connection string that we have, so copy the name, and if it's different for you, you can copy the name and then paste it over here, so that there's no error.

I will complete the bracket and I will add the semicolon to end the statement.

So this is the way that we are injecting our ApplicationDBContext inside the program.cs file.

Now we are just one step away from creating our database, and that step is the Entity Framework Core migrations.

That is a way for the application to create some files in C# and then execute these files, and because we have given the connection as well, they will execute this C# classes and convert that to SQL to create a new database and a new table as well.

To run Entity Framework Code migrations, we will first come to tools, we will go to NuGet package manager, and open Package manager console.

This will open a window over here where we will write two statements, two commands.

The First Command that we will write is "Add-Migration", and after that, inside the double quotes, we can give a name for the migration.

Because we are creating the database for the first time, we can call it "Initial Migration", and after that, press enter to run this command.

It takes a few seconds for it to run the command, but once it has done it, it will create a new migration class.

You can also access the migration class from this folder that has been created.

We have also got a new migrations folder and we have got the initial migration class.

If you open the class, we'll have two methods, up and down method.

This is basically a way for Visual Studio and C# to know what it has to create inside the SQL Server database.

So it's creating a table that we have, the employees table, and it knows about all the fields that we want to create in the table.

Because we defined them inside the entities which are connected to the DB context over here.

So it creates a C# version of what it wants to execute inside the database.

The next step then becomes to execute this.

I'll come back to package manager console, and the next step is to run this migration.

The command to do that is "Update-Database". Press enter, and it will run the migrations that it hasn't run before.

It knows the connection.

You'll get an error like this one, and it says that only the invariant culture is supported in globalization invariant mode.

This is a new error that we now get in .NET Core 8 or EF Core 8 actually.

The way to go ahead with this one is that over here, inside the project file, we have this property that says invariant globalization as true.

You just have to make it as false and save the changes.

Now, if we come back and run the statement again, it should now run fine.

It should now run the migration that you created and execute it in the SQL Server database.

So it looks like the migration has run successfully.

But there's only one way to check it, and that's by connecting to this server that you have.

I'm already connected to the server.

So I just need to refresh it.

And if I refresh it and open the databases.

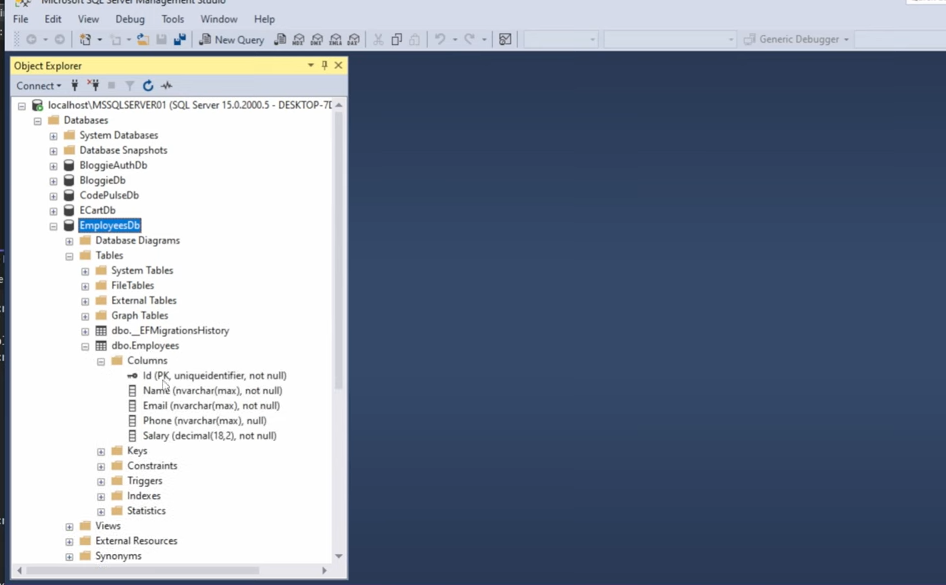
I now have a new EmployeesDB.

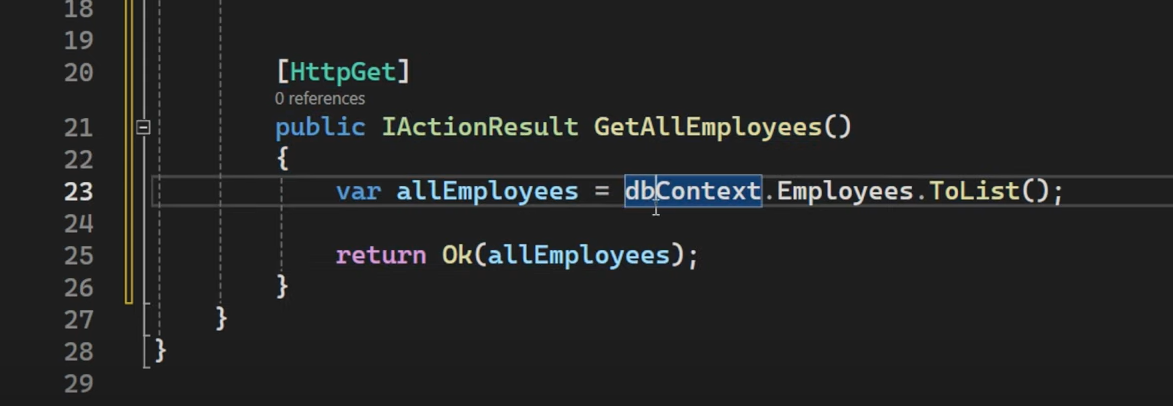
Which should have one table.

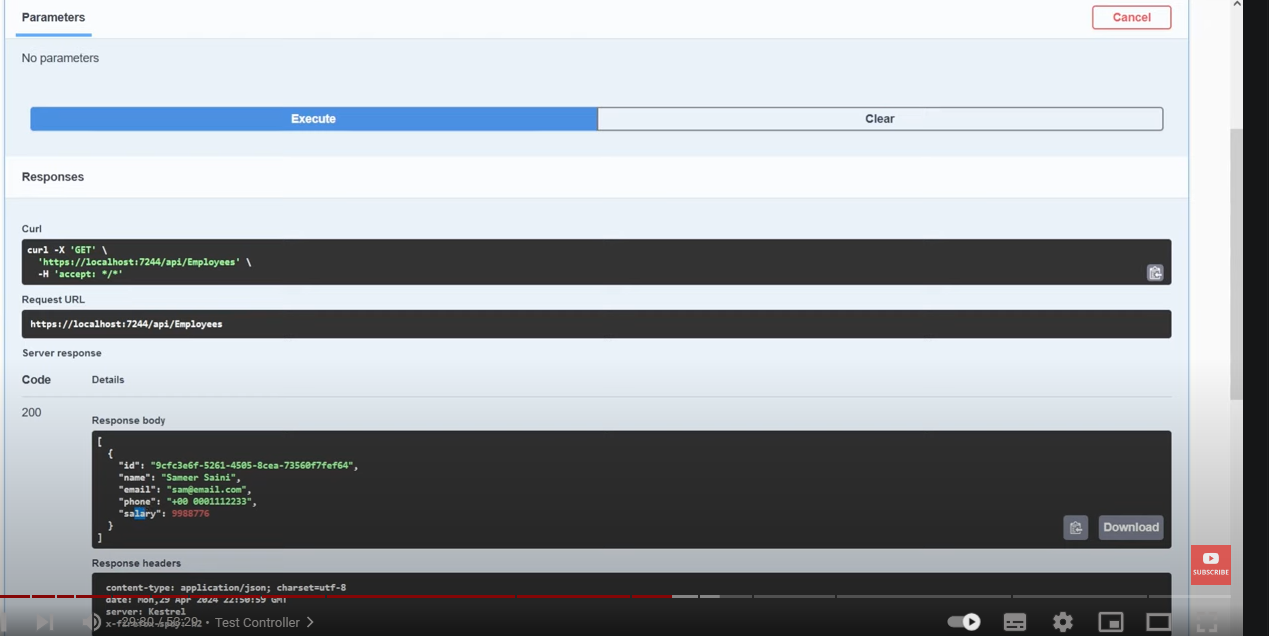
And that is the Employees table.

And it is over there.

Which has the exact number of columns and the exact type of columns that we have defined in the entity that we created.







Here is the text without timestamps:

Test Controller

Browser inside the browser, I can see a different controller, which is the EmployeesController, and it has the various endpoints available for us.

The first one is the GET operation.

So I'll expand that, click on "Try it out", and click on "Execute".

It's now trying to make a connection to the database and fetching all the results back.

So as you can see, we have the server response over here, and the server gave us the 200 success response.

So basically, the request was successful, but the result was empty.

And that is true because we don't have any employees in the database at the moment.

What we can do is we can come over here and add an employee manually, just to test this function.

So I will come over here, edit top 200 rows.

I want to now give the values of these fields, starting with the identifier.

So I will come online and find a GUID generator.

I will copy that and paste it in the ID.

Then I can give a name.

Let's give it my name or your name, and some email.

So this is a very genuine email.

And after that, we can specify a phone number.

And then maybe some salary as well.

And once you tab it, your row will be saved.

And if I want to select it again, we can select this.

And you can see now we have one entry in this table.

And now if we come back to Swagger to test our endpoint.

We can execute this again.

But this time, we get the result from the database back.

So you can see that we got the ID back, name, email, phone, and salary from this database.

So you can see how easy it is to create this endpoint, which is the GET endpoint that we have.

And it's basically one line of code.

We've just extracted that into two lines by creating this variable.

But you can do this thing.

You can just copy it over here, and that becomes your one line of code.

And this is all it is for the GET operation.

That we have.

The next operation that we want to build is the functionality to add a new employee from the application.

And for that, we want to create a new action method over here in this controller.

Let's create the structure for the action method for adding a new employee.

And that will be the POST method.

Because that is, as per the RESTful operations.

So we use the HTTP word of POST.

So HTTP POST is what we use for the add operation.

And we'll define the structure of this action method.

So public IActionResult.

And we can call this AddEmployee.

We will define the method structure now.

To add an employee, we need a few details of the employee.

If we come back to the models folder, we have the Employee class.

And apart from the ID, we need the name, email, phone, and salary.

And from these four properties, we know that the name, email, and salary are required.

And phone is an optional field.

So it's, you know, if the user wants to give it, they can.

And it's basically optional.

So we want to accept the parameters as the body of this POST request.

That's the HTTP body.

And we want to create a new object for this one.

Which will be as a carrier of this data.

And for these type of objects, we use DTOs, which are Data Transfer Objects.

And as the name suggests, they basically transfer the data from one operation to another.

And for this one, we will create a DTO inside the models folder.

So over here, right-click on the models folder.

Add a class.

And I will name this class as AddEmployeeDTO.

And this AddEmployeeDTO class will have similar properties to the Entities class.

Because we want to accept the name, email, phone, and salary.

So what I can do is I can copy these four properties and paste it inside here.

So these are the properties that I want to accept from the user.

Now I can use this object inside the controller as a parameter.

So inside the AddEmployee method.

I will define the parameter type as AddEmployeeDTO.

Give it a name.

Let's say addEmployeeDTO.

The first thing.

Here is the text without timestamps:

Add Employee DTO

That we will do over here is that we will map this AddEmployeeDTO.

And we will convert this into the entity that we have.

Because the DB context class that we are going to use to add this employee only accepts the entities over here.

So to tell you what I mean.

We want to ultimately save this employee in the database.

And we do that using the DB context file.

And we have injected that over here.

So we'll say DbContext.

Employees table, if you want to call it that.

Dot.

We have the Add method available to us.

Do.

Add.

And if I open the bracket.

You can see that the Add method on this Employees table accepts this type of file.

Which is the Entities Employee.

So this method needs the Employees entity type.

But we have available with us the EmployeesDTO.

So we have to somehow convert this from the EmployeeDTO to the Employee entity.

To do that.

I will create a variable over here.

Let's say employeeEntity is equal to new Employee.

And this comes from the Employee entity over here.

So press control Dot.

And we have the import statement of Entities the folder.

After that.

If I press enter.

We have the properties available to us.

And you can see that the required properties are also there.

The ones that the class needs for us to fill.

So we won't fill the ID.

Because Entity Framework Core handles the ID for us.

So first of all.

Name.

And that comes from this parameter that we have.

Then we have the email.

Which is also required field.

And it also comes from the addEmployeeDTO email property.

Then we have the phone.

Is equal to addEmployeeDTO.

Phone.

And finally.

We have the salary.

Also coming from that object.

Now a lot of people think that this is just a waste of code.

And why are we even accepting a different type of object.

Then converting it into the entity.

And passing the entity.

And the reason for that is.

Entities are separate from the DTOs.

And by adding DTOs.

We achieve a separation of concern.

We are basically exposing this information to the outside world.

This DTO class.

Whereas the Employee entity.

Is an image of what we have in the table or the database.

And you can have different names.

So instead of name.

We could have had full name.

And you know.

To the outside user.

We could only expose name.

Or let's say separate fields.

As first name and last name.

So it gives us a lot of modularity.

And you know.

By achieving this type of separation.

Between DTOs and entities.

We make our code quite reusable.

And generic.

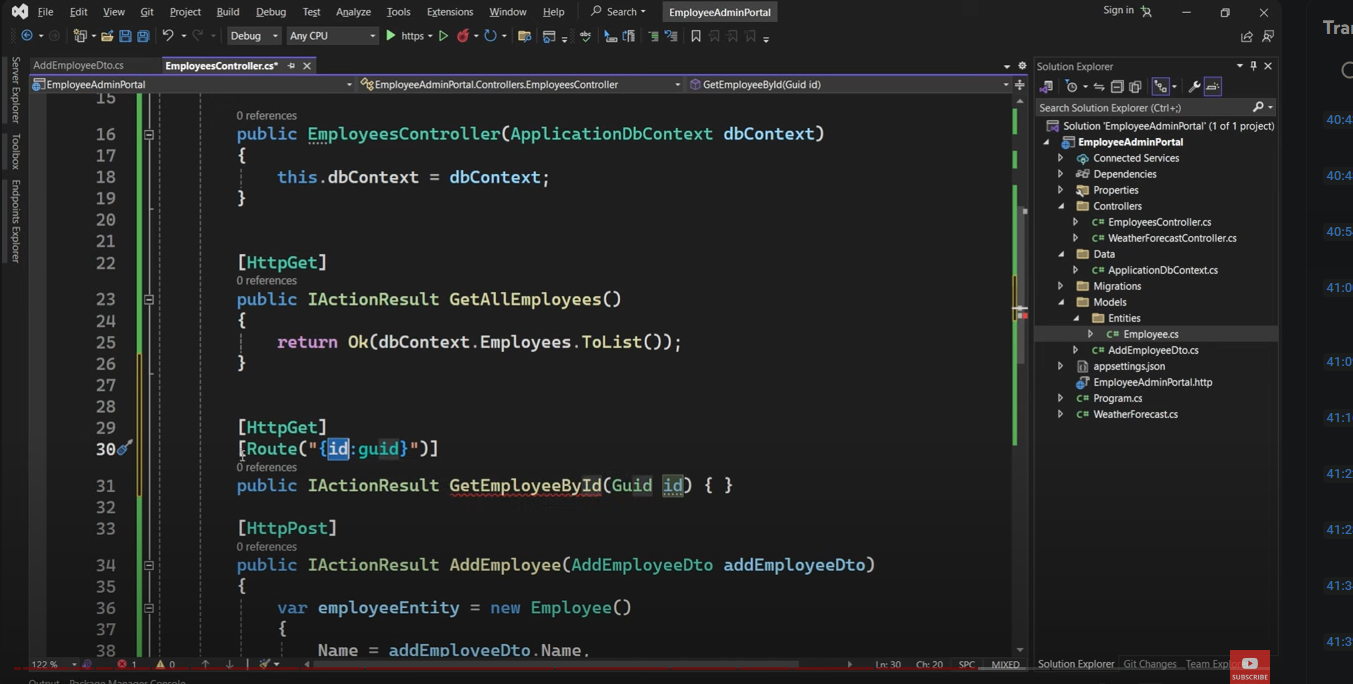
And readable as well.

So now that we have converted our AddEmployeeDTO into a Employee entity.

We can now use this employee entity.

And pass it to the Add method over here.

Now I'll put a semicolon to end the.



Here is the text without timestamps:

Update Employee

Operations inside the CRUD operations, which is create and retrieve, and now we are left with two operations, which is the update and delete.

So let's first do the update operation.

And for update, we follow the convention of HTTP PUT.

So we use the PUT verb inside the restful resource or restful practices when we want to update a particular resource.

So we will use HTTP PUT.

And after that, we will define the name the structure of the method.

So public IActionResult update.

Employee.

Now this method will also need to uh the information of the employee that you're updating.

Now you know similar to how we were doing things in the get method.

Because we were getting an employee based on the ID.

We also want to use a similar structure for the employee or for the update method.

That means we need an identifier for the employee that we are trying to update.

And we also need a parameter which we can map to.

So we'll have the first parameter as identifier of type GUID.

The second parameter for this method is the things that we want to update.

So we basically want an object like this one.

And that will have the information that we want to update in this employee.

So we can create another DTO.

So right-click on models, add a new class.

And let's call this class the UpdateEmployeeDTO.

And this will have a similar structure to the AddEmployeeDTO.

So if we want to update name, email, and phone.

We can do that.

If you also want to update the salary.

We can also copy that as well.

So this EmployeeDTO will have all the properties inside the Employee collection that we want to update.

Now we will use this UpdateEmployeeDTO inside the UpdateEmployee method as a second parameter.

So employee UpdateEmployeeDTO.

We can give it a name.

And this is what we have over here.

We have two parameters to this method.

The first thing we will do is we will first try to find this employee.

So similar to how we did this stuff in the GetEmployee method.

We will create a variable called employee.

Is equal to.

And we will try to search this employee by ID.

So DbContext.

Employees collection.

Find method.

Pass the ID to it.

Now this could be an employee if it was found.

Or it could be null.

So first of all.

We will say if employee is null.

That means it was not found.

Return a not found back.

So return the not found method.

And if it was found.

We want to update it.

So basically.

We want to say employee name is equal to.

Use this information that we are getting from the client.

And use this name property.

So we want to change this name to the value that we are getting from this DTO.

Similarly.

We will also do it for email.

And we will have the email from the DTO.

Similarly for phone.

And salary as well.

If you have more number of properties.

You can also update those properties over here.

Now that we have updated these properties.

The last thing we want to do is to save changes.

So if you forget to save changes.

You will not see the results in this database.

So first of all.

We'll call DbContext.

SaveChanges method.

And this method will make sure that any updates that you have over here.

Is also shown or reflected in the database.

And finally.

We can return an okay response.

And we can return this updated employee back.

So this is the update method in CRUD.

And we are using the PUT verb.

So it's now time to test this.

We get this new operation over here.

Which is the update operation.

Represented by the keyword the verb PUT.

And I want to try it out.

And it also needs an identifier of the employee that we want to update.

You can come back to the database.

And let's say we want to update this particular employee.

So pass the identifier.

And also pass this information that you want to change it to.

So for example.

We have the employee as John Doe.

And we want to change this information.

So let's say this is the information.

I want to change it to.

I've made changes to all the properties.

And if I click on execute.

This should return us a 200 success response.

And when it does.

This would mean that our information has been changed in the database.

To confirm that.

We can come over here.

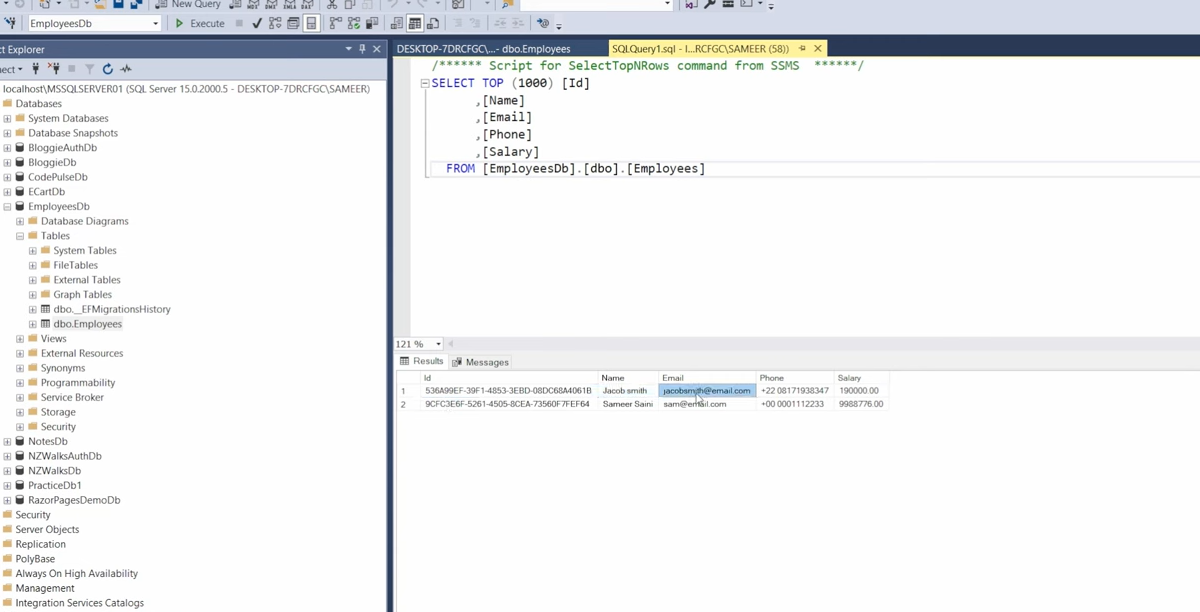
And select this again.

Or execute this again.

The information for John Doe.

Has now been changed to Jacob Smith.

And you can see the changes reflected on the name, email, phone, and salary as well.



Here is the text without timestamps:

Delete Employee

You can see that our update operation is also working as expected.

The final operation or the final action method that we want to add is the delete method.

To delete a resource by its identifier.

And we use the HTTP Delete attribute.

That refers to the delete verb in terms of restful practices.

And we will create a structure.

And it's suggesting me the structure which is correct.

Delete an Employee.

It will take an identifier.

But it wants to take the identifier from the route.

If we follow REST.

So I will follow the same route over here as well.

So make sure the identifier matches the parameter.

And that is all the information we want to have for this delete operation.

To delete a particular employee.

Only the only thing we need is the unique identifier.

And we will search the employee in the database first.

If we are able to find it.

We will delete it.

Otherwise.

We'll send a not found back.

So again.

We'll say which variable employee is equal to.

DbContext dot.

Employees collection.

Do find method.

And pass the identifier.

After that.

We'll say if employee is null.

Then we want to return a not found back.

And otherwise.

We want to delete this employee from the database.

So we will say.

DbContext.

Employees collection.

And we have a remove method.

We don't have a delete method.

But we have the remove method.

Which takes the entity.

If I hover over it.

It takes the employee entity.

And this is the employee entity that we got.

So we will pass it to the remove method.

And don't forget to save changes.

So DbContext.

Save changes.

So in any other operation.

Except from the Get method.

You have to remember to save the changes.

So that it reflects the changes inside the database as well.

And after saving the changes.

You can return the okay response.

If you want to pass any information inside.

You can pass that.

But like I'm sending an okay response.

With an empty object back.

Time to test this.

We have the delete method now available to us.

We'll try it out.

We will pass it the identifier of an employee that we want to delete.

I'll select this.

And pass it.

If I execute.

It.

I get the success response back.

I don't get a response back.

Because I'm not expecting that.

But if I now come back to SQL Server.

And execute the statement again.

I only have one employee left.

So you can see.

We have finished the CRUD operations in .NET 8 Web API.

And it was a very easy solution.

And you know.

There's much more to Web APIs.

And you can find all of that in my course.

That has been linked in the description.

And you know.

If you like this course or this video.

Please give it a thumbs up.

Subscribe to the channel.

Because that means a lot.

So that this video and the channel can reach more users or more students like yourself.

That's it for this one.

Thank you for watching.

And I'll see you in the next one.

Here is the text without timestamps:

Today, I'll show you how to add authentication and authorization in your (link unavailable) Core Web Application with JSON Web Token.

Open up Visual Studio.

Create a new (link unavailable) Core Web Application.

Select that.

If you can't find it, search for it in the search bar.

Click Next.

Give it a name.

Give it a location.

And then click Next again.

And then here I'm going to choose .NET Core 3.1.

No authentication type.

And click Create.

So here we are.

A few things that we need to do is to install three packages.

Install Packages

If you open up Solution Explorer.

Then right-click on the Dependencies.

And Manage NuGet Packages.

The first one is Microsoft (link unavailable) Core Authentication JWT Bearer.

One important thing to note is that if you have created a .NET Core 3.1 application.

Then make sure you choose 3.1.24 JWT Bearer.

Because anything above that will not be compatible with .NET Core 3.1.

So just make sure that you choose that.

If you're working with .NET 5.

Then higher versions of the JWT Bearer package will work.

But in this case, 3.1.

That's all you need to do.

The next package that we need to install is Identity Model Tokens.

This is it.

And you can choose this one.

It's fine for the rest of the packages.

To install the latest stable versions.

And then last but not least is Identity Models Tokens JWT.

Okay.

Let's finish installing.

So I'm gonna paste in the description section.

All the packages.

So that you can go ahead and copy and paste them as you wish.

Okay.

So installing the latest one.

That's finished.

Okay.

Following the next step is to set up appsettings.json.

Setup appsettings.json

So if we close everything.

And then we go to the root of our project.

And then open up appsettings.json.

I've got a snippet that I've already created.

I'm just gonna paste that in.

So this is a new section.

JWT.

That I have created.

That's got a key.

That's randomly generated.

Link in the description.

With a website.

That's just generated a random string of characters.

And numbers.

And then we've got an issuer.

And an audience.

And we need two addresses here.

Which in our case are going to be similar.

So if we right-click on the web project.

On the web app.

And then go to Properties.

And then navigate to the Debug tab.

Scroll down.

And you will see that right next to.

So in the web server setting.

In the settings section.

We've got Enable SSL.

And then we've got an address here.

That URL is what we need to paste in here.

Both the issuer.

And the audience.

So the server that's gonna issue the JWT token.

And the audience.

The one that's gonna use it.

Are gonna have the same address.

So that's it with setting up appsettings.json.

Following.

We need to set up startup.cs file.

Setup Startup.cs

So if we open up startup.cs.

And scroll down to Configure Services.

So we've got that method right here.

It should be on line 24.

Okay.

So inside the Configure Services.

Inside startup.cs.

We need to add the configuration part for the JWT Bearer.

So that's the first thing.

So we do services.AddAuthentication.

And then in here.

We define JWT.

We say um.

We pass in the authentication scheme as JWT Bearer Defaults.

And that's not it.

By the way.

If you haven't got this already.

Um imported in your startup.cs.

Just press Ctrl + dot.

And then hit Enter.

And then one of the packages.

Uh should just import automatically.

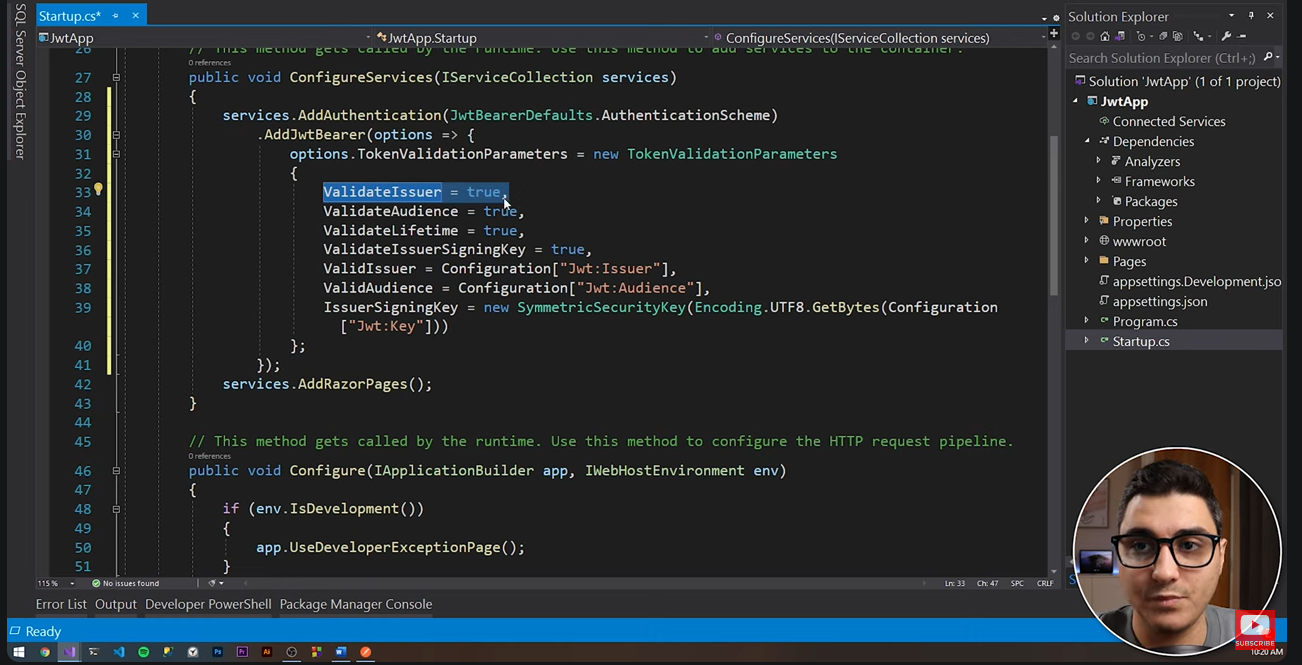
Inside startup.cs.

Then following.

We don't put a semicolon at the end.

We do a dot JWT Bearer.

And then we define.



Here is the text without timestamps:

An options object that we need to define.

So we need, in order for this to work properly, we need to set this up with our custom properties.

We've got our custom options.

We need to define the options.TokenValidationParameters.

Um, and obviously we need to initialize that with the TokenValidationParameters object.

So, uh, we need to uh tell the API that every time you receive a request with where you've got the authorized tag in it.

Then validate the issuer that we have defined earlier inside appsettings.json.

Obviously validate the audience as well.

Validate lifetime.

And then validate issuer sign-in key.

So we need to validate that as well.

If we are going to provide a signing key.

It could be optional.

But just to make it more secure.

To add an extra layer of security.

We are going to define a private key.

Then the valid issuer that we define is um from the configuration.

So from appsettings.json.

And we're accessing it using jwt:issuer.

That's the path to it right.

Jt is the section.

And then issuer is the actual value.

So this is what we're going to import in there.

And this um this is better than just pasting this string in here.

Because this is an app secret.

So um when you get to deploy this.

You'll obviously be able to replace the value in app settings.json.

With um the issuer on the server.

And the audience address on the server.

And so on.

Um.

So following.

We define the uh the valid audience.

This is going to be that the token will be validated against uh in here as well.

And then the issuer signing key.

We are doing a symmetric key.

So we're creating a symmetric security key.

Um.

And obviously we're encoding that jwt key.

Um.

So that we can create the symmetric security key.

So that's how we um that's how we uh define the options of the token validation parameters.

Following.

We need to add a couple of lines again in this Configure Services method.

That's going to be services.AddMVC.

And services.AddControllers.

And that is it with Configure Services.

Then in the next part.

What we need to do is add a couple of lines.

A couple of configurations inside the startup.cs Configure method.

So this is different to Configure Services.

In here.

We we are aiming for the Configure method.

And in here.

We need to add authentication.

And uh map the controllers.

Because if we don't do that uh when we get to create the API controllers.

That we're gonna use throughout the application.

We won't be able to access them.

So very important uh to do this step.

Um.

So the first thing that we want to do in here is.

App.UseAuthentication.

Yeah.

So that we can uh.

We can allow for the API endpoints to authenticate the users.

And then last but not least.

Endpoints.MapControllers.

Map controllers.

That's it with setting up startup.cs.

Right now.

Let's add a models folder to hold our three models.

One will be the User model.

Uh.

That's gonna hold all the data about a user.

So this is gonna be some sort of like an (link unavailable) user uh entity in the EF Core.

But obviously we're not going to work within the EF Core database.

With the SQL Server database.

Uh.

Therefore.

We're just going to simulate the presence of it.

So User model.

User Login.

That's going to have only username and password.

And the UserConstants.

That's gonna be some sort of like a database holder for us.

Okay.

So um.

Let's right-click on the project.

Add a new folder.

Models.

And then in here.

Like I said.

Three classes.

The first one.

User Model.

User Model.

And this is gonna be a public class.

And I'm pasting in the properties right now.

So we've got a username.

A password.

An email address.

A security role.

A surname.

And a given name.

So pretty simple.

Then the second one will be User Login.

Here is the text without timestamps:

And obviously you can pass in the API the username and the password.

But it's uh, it's better to do it like this.

It's best practice.

You'll see what I mean.

So User Login.

And this is going to have just a couple of strings.

Username and password.

That's it.

And then lastly the User Constants.

So right-click.

Add new class.

And let's call this User Constants.

And this just holds a list of two users.

That I have already created for us.

Um, one is called JSON Admin.

And it's an administrator.

It's got a role of an administrator.

And then one is called.

The other one is called a List Seller.

And it's got a security role of seller.

So this is it with this step.

With setting up our models.

That we're going to use throughout this application.

Add API Controllers

Okay.

Following.

Here is the meaty part.

We'll add the API controller.

So the first API controller.

That we want to add.

Is the Login Controller.

Which is the one.

That will authenticate the user.

As well as will generate the token.

For the user.

Based on the user details.

That are found in the database.

Based on the username and password.

So let us create a Controllers folder.

So Controllers.

Then in here.

Let's create an API Controller.

So right-click.

Add new item.

And then type in API.

If you don't already see it.

And let's call this Login Controller.

So this is an API Controller.

That's inheriting from ControllerBase.

Okay.

The first thing.

That we want to do.

Obviously.

Is to initialize.

The uh.

To inject the IConfiguration.

So that we can have access to all the details.

From appsettings.json.

So I'm just going to find a private IConfiguration.

Property.

At the top of the field.

At the top of this class.

That's going to be called config.

And then.

If you type in ctor.

And double tab.

Um.

Then we can actually inject this.

So I'm just gonna copy and paste this here.

Without the underscore.

Um.

So that I can do config dot.

Config equals to.

Uh.

The config object.

That's passed in.

So this is how we inject things.

Um.

Then the very first method.

Is obviously.

Login.

So this is going to allow anonymous.

This is an annotation.

That will um.

Prevent the authorization.

Process.

The authentication process.

To happen.

Um.

At the point of calling this method.

Because this is the very first thing.

That the user is ever gonna call.

In this application.

The login method.

Obviously.

If they're not already logged in.

Okay.

So this is an HTTP Post.

And then.

This is a public.

Returns.

IActionResult.

And it's called Login.

Okay.

And this is going to take.

FromBody.

Um.

A UserLogin object.

And let's import that.

Really quick.

Let's call this UserLogin.

So again.

This UserLogin.

Only has username and password.

We could have easily just passed in those two strings.

In the parameter section.

Here is the text without formatting or timestamps:

And obviously you can pass in the API the username and password, but it's better to do it like this. It's best practice. You'll see what I mean.

So User Login, and this is going to have just a couple of strings, username and password, that's it.

And then lastly the User Constants. So right-click, add new class, and let's call this User Constants. This just holds a list of two users that I have already created for us.

One is called JSON Admin and it's an administrator, it's got a role of an administrator. And then one is called List Seller and it's got a security role of seller.

So this is it with this step with setting up our models that we're going to use throughout this application.

Add API Controllers

Okay, following, here is the meaty part. We'll add the API controller. So the first API controller that we want to add is the Login Controller, which is the one that will authenticate the user as well as generate the token for the user based on the user details that are found in the database based on the username and password.

Let us create a Controllers folder. So Controllers, then in here let's create an API Controller. So right-click, add new item, and then type in API if you don't already see it. And let's call this Login Controller.

So this is an API Controller that's inheriting from ControllerBase. Okay, the first thing that we want to do, obviously, is to initialize the IConfiguration so that we can have access to all the details from appsettings.json.

So I'm just going to find a private IConfiguration property at the top of the field at the top of this class that's going to be called config. And then if you type in ctor and double tab, then we can actually inject this.

So I'm just gonna copy and paste this here without the underscore, so that I can do config dot config equals to the config object that's passed in. So this is how we inject things.

Then the very first method is obviously Login. So this is going to allow anonymous. This is an annotation that will prevent the authorization process, the authentication process to happen, at the point of calling this method, because this is the very first thing that the user is ever gonna call in this application.

The login method, obviously, if they're not already logged in. Okay, so this is an HTTP Post, and then this is a public returns IActionResult, and it's called Login.

Okay, and this is going to take from body, a UserLogin object. And let's import that really quick. Let's call this UserLogin. So again, this UserLogin only has username and password. We could have easily just passed in those two strings in the parameter section.

ASP.NET Web API CRUD Operations - .NET8 and Entity Framework Core Tutorial- https://www.youtube.com/watch?v=6YIRKBsRWVI

ASP.NET Core Web API Authentication and Authorization with JWT (Json Web Token)- https://www.youtube.com/watch?v=kM1fPt1BcLc

Abstract Classes and Interfaces:

Abstract classes define methods without implementation, allowing derived classes to provide specific behavior. These cannot be instantiated directly.

Interfaces define a contract of methods that a class must implement. A class can implement multiple interfaces.

csharp

public abstract class Animal

{

public abstract void Speak();

}

public class Dog : Animal

{

public override void Speak()

{

Console.WriteLine("Bark!");

}

}

Static Classes and Members:

Static classes cannot be instantiated and are typically used for utility or helper methods.

Static members belong to the class itself, rather than any object. They can be accessed without creating an instance.

csharp

public static class MathHelper

{

public static int Add(int a, int b)

{

return a + b;

}

}

int result = MathHelper.Add(2, 3); // Outputs: 5

Intro to MongoDB

Audio Transcript:

Welcome, team. If you're brand new to MongoDB, this is the course for you. I'm Cam, a technical instructor here at MongoDB. And I'll be teaching this course along with many of my talented colleagues.

Together, we'll guide you through everything you need to know to get started with MongoDB. First, we'll give you an overview of MongoDB. Next, we'll also show you how to connect your database and start

running CRUD operations. Then we'll help you create a search experience for users of your app and coach you on aggregation, indexing, data modeling, and transactions.

You'll get to practice all of these concepts in hands-on labs and knowledge-check quizzes. Make sure to sign in with your email if you want to collect points and badges along the way. By the end of this

course, you'll have everything that you need to work with MongoDB. If you're interested, you can also take this course along with one of our drivers courses to get ready for your MongoDB certification.

Let's get started.

Course Introduction

Introduction to MongoDB

The Introduction to MongoDB course guides you through the foundational skills and knowledge you need to get started with MongoDB and Atlas, the developer data platform.

The course provides you with an overview of MongoDB, shows you how to connect to your database by using Atlas, and demonstrates how to run CRUD operations.

It also covers how to create a search experience for the users of your app, and coaches you on key topics such as aggregation, indexing, data modeling, and transactions.

Throughout the course, you'll get to practice all of these concepts in hands-on labs and knowledge check quizzes.

If you're interested, you can take this course in addition to one of our drivers courses to help prepare you for MongoDB certification. Be sure to sign in to track your progress and get a certificate

of completion.

This course is broken down into the following units:

Unit 01: Getting Started with MongoDB Atlas, the Developer Data Platform

Unit 02: Overview of MongoDB and the Document Model

Unit 03: Connecting to a MongoDB Database

Unit 04: MongoDB CRUD Operations: Insert and Find Documents

Unit 05: MongoDB CRUD: Replace and Delete

Unit 06: MongoDB CRUD Operations: Modifying Query Results

Unit 07: MongoDB Aggregation

Unit 08: MongoDB Indexing

Unit 09: MongoDB Atlas Search

Unit 10: Introduction to MongoDB Data Modeling

Unit 11: MongoDB Transactions

(3 language specific learning modules)

Getting Started With Mongo DB Atlas

Mongo DB and Document Model

MongoDB Data Modeling Intro

Connecting to a Mongo DB Database

Connecting to Mongo DB in C#

Mongo DB: Insert Find Replace Delete

Mongo DB CRUD: Modifying Query Results

Mongo DB Aggregation

Mongo DB Indexes

Mongo DB Atlas Search

Mongo DB Transactions

MongoDB Compass

Mongo DB VS Code Extension

https://learn.mongodb.com/learn/course/mongodb-sql-cheat-sheet/main/mongodb-sql-cheat-sheet

Build a web API with minimal API, ASP.NET Core, and .NET- https://learn.microsoft.com/en-us/training/modules/build-web-api-minimal-api/

Interact with an ASP.NET Core minimal API- https://learn.microsoft.com/en-us/training/modules/interact-api/

Persist and retrieve relational data by using Entity Framework Core- https://learn.microsoft.com/en-us/training/modules/persist-data-ef-core/

Secure a .NET web app with the ASP.NET Core Identity framework https://learn.microsoft.com/en-us/training/modules/secure-aspnet-core-identity/

Guided project - Create an AI travel agent- https://learn.microsoft.com/en-us/training/modules/guided-project-create-ai-travel-agent/

Use a database with minimal API, Entity Framework Core, and ASP.NET Core- https://learn.microsoft.com/en-us/training/modules/build-web-api-minimal-database/

Create web apps and services with ASP.NET Core, minimal API, and .NET- https://learn.microsoft.com/en-us/training/paths/aspnet-core-minimal-api/

Create a full stack application by using React and minimal API for ASP.NET Core- https://learn.microsoft.com/en-us/training/modules/build-web-api-minimal-spa/

Create accessible web apps by using ASP.NET Core- https://learn.microsoft.com/en-us/training/modules/aspnet-core-accessibility/

Interactively debug .NET apps with the Visual Studio debugger- https://learn.microsoft.com/en-us/training/modules/dotnet-debug-visual-studio/

Real Time Web App- https://learn.microsoft.com/en-us/aspnet/core/tutorials/signalr?view=aspnetcore-8.0&tabs=visual-studio

HTTP API apps

Develop HTTP services with ASP.NET Core- https://learn.microsoft.com/en-us/aspnet/core/tutorials/min-web-api?view=aspnetcore-8.0&tabs=visual-studio