**SLI Commands**

**Prefix:** dotnet

-h after each command opens up help

new <templateName> -o <folderName> -o specifies an output folder, else it will be created on the selected opens

run : obvious

watch run: is alert for any file changes

Startup.cs

- ConfigureServices : The dependency injector. We add classes/services here and the method takes care of them(creation, destruction etc)

- Configure: contains settings for the HTTP request pipeline

launchSettings.json : contains the obvious

**ADDING SERVICES**

Startup.cs > ConfigureServices >

services.Add() with a generic declaration

or

services.AddSpecificService(options)

Specify in options with a lambda expression what they're going to be.

**SERVICE TYPES**

1. Singleton: When created, it ends when the application ends or is explicitly told to do something.

2. Transient: Starts when the method is called and end when it ends.

3. Scoped: It exists as long as the Http request and is disposed when the request ends.

**ADD A DB**

1. Import Nuget for entity framework

2. Create Context class and inherit from DbContext

3. Add DbSets

4. Add the connection string to the appsettings(.development).json

eg. "ConnectionStrings": {

"DefaultConnection" : "Data source=dbname.db"

}

5. For dotnet 5.0.0, add the Context to the Startup.cs:

a) Refactor the IConfiguration field as private readonly

b) Add the Context to the Services configuration passing in the options the connection string

e.g services.AddDbContext<DataContext>(options =>

{

options.UseSqlite(ConfigField.GetConnectionString("DefaultConnection"));

});

6. Install ef tools depending on version

7. Create migration

eg. dotnet ef migrations add <MigrationName> -o <FolderAddress>

8. Create db : dotnet ef database update

**API CONTROLLERS**

1. Add new controller and inherit from ControllerBase (controllers without views)

or

Create a new class with all the necessary properties and decorators that all the controllers will inherit from.

2. Add attributes:

[ApiController]

[Route("routePrefix/[controller]")] (The [controller] specifies that the route will

take the controller's name inside the square brackets)

3. Add a constructor with the DataContext injected

eg. private readonly DataContext Context;

public UsersController(DataContext context)

{

Context = context;

}

4. Create the action methods as needed. Best to be asynchronous if calling the db

Add attributes for API verbs along with the route eg. [HttpPost("methodNameAsRoute")]

5. If the controllers are for API, it's best to use DTOs

The ApiController class specifies that any parameter in a API verb method

will be bound to a parallel one from the request url, if it is a "simple" one. Else, we can assign specifically

from where will the value come. [FromBody] etc

https://docs.microsoft.com/en-us/aspnet/web-api/overview/formats-and-model-binding/parameter-binding-in-aspnet-web-api

**IMPLEMENTING CORS**

Startup.cs

1) ConfigureServices: add CORS

2) Configure: a) add CORS between Routing and Authorization. Order is important in this case.

b) add a policy. AllowAnyMethod()

AllowAnyHeader()

WithOrigins(): specify the client url that will request the data

**SSL CERTIFICATES**

dotnet core: CLI dotnet dev-certs https --trust or --clean and then --trust

angular: a) copy certificate in a folder inside the app

b) add the options to angular.json > serve

eg "options": {

"sslKey": "./ssl/server.key",

"sslCert": "./ssl/server.crt",

"ssl": true,

"browserTarget": "test:build"

}

**API TOKENS**

1. Best practice is to create an Interface

2. Create a class that inherits from the Interface

3. Add Nuget System.IdentityModel.Tokens.Jwt

4. Create a: private readonly SymmetricSecurityKey key

5. Add a controller injected with IConfiguration and assign the key to a SymmetricSecurityKey

with Encoding.UTF8.GetBytes(config["TokenKey"]). This will get the key that the user

adds in the request.

6. In the CreateToken method:

a. add as the parameter from which the credentials will be generated.

b. create a list of claims that will be created from the parameter

c. create the signing credentials with the key and an algorithm

d. create a token descriptor with:

i. subject: ClaimsIdentity(claims)

ii. expiration

iii. SigningCredentials

e. create the token with a JwtSecurityTokenHandler

f. return the token by writing it with the Token Handler

7. Create a DTO to be returned from the API, which will contain the relevant data along with the token

8. Inject the Token servics to any controller that needs it and assign the token to a field

9. Return from the action methods the relevant DTOs

10.Add the token key to the appsettings.(Development).json. Never upload it in production environment.

11. Add the service to the Startup.cs

**TOKEN AUTHENTICATION**

1. Add the Nuget Microsoft.AspNetCore.Authentication.JwtBearer

2. Startup.cs > a. Add the authentication service

eg. services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuerSigningKey = true,

IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(Config["TokenKey"])),

ValidateIssuer = false,

ValidateAudience = false

};

});

b. Configure > Add the UseAuthentication after CORS and before Authorization

2. Specify on the controllers if authentication is needed

**MIDDLEWARE**

Each HTTP request travels through the middleware in *Startup.cs*, so it’s important for them to be arranged in the correct order. On top goes the exception handling.

**EXCEPTION HANDLING**

1. Create a custom class to be used as middleware. In the constructor, pass as parameters a *Request Delegate* which will handle the HTTP, a *ILogger<MiddlewareName>* for the logger and an *IHostEnvironment*. Each of the parameters sets a relevant field in the class
2. Create an asynchronous method which will take the *HttpContext* as a parameter.
   1. Add a *Try-Catch*block, where the *RequestDelegate(context)* will be tried. This passes the Http request to upper levels, but as the Exception handling is on top, it stops here.
   2. Catch any exceptions and
   3. log the error,
   4. set the response content type *(“application/json”)*
   5. set the response Status Code
   6. create a response depending on whether we are in production mode, which should return an object with the response data (status code, exception message, stack trace and anything relevant)
   7. add the json options for the serializer and create the json
   8. pass/write the json to the response
3. Add the custom exception handling at the top of the Middleware in the Startup.cs

**ANGULAR HTTP REQUESTS**

1. app.module > import HttpClientModule from angular/common/http

2. app.component

a) inject the HttpClient as a constructor parameter

b) inherit/implement OnInit

c) use the relevant method/api verb from the httpClient inside a function or in ngOnInit()

i. the response should be assigned to something in order to be used by other components etc

ii. do something with the error

iii.complete is optional

Services

1. import HttpClientModule from angular/common/http

2. inject in the constructor the HttpClient

3. specify the url

4. specify which will the request body be

5. create a function where the relevant API verb should be used

6. inject the service in a component constructor

7. use any functions and subscribe to the observable (response, error, etc)

eg. export class AccountService {

baseUrl = 'https://localhost:5001/api/';

constructor(private Http: HttpClient) { }

login(model:any){

return this.Http.post(this.baseUrl + 'account/login', model);

}

}

**ANGULAR TEMPLATE REFERENCE VARIABLES**

Declared a template variable with a # and specify inside "" of what type it is if needed.

Template variables can be used inside the template to refer to the element on which they are declared.

We can also use the *as* keyword to specify something as a template variable.

https://angular.io/guide/template-reference-variables

**ANGULAR FORMS**

- Declare a form by using ngForm as a template variable type specification.

- (ngSubmit): assign the function to be called on this event.

- [NgModel]: Creates a FormControl instance from a domain model and binds it to a form control element.

**ANGULAR SERVICES**

- Singleton

- @Injectable : can be used/injected into other components

- providedIn: 'root' : declares that the service is available in the app.module

**DATA TRANSMISSION BETWEEN COMPONENTS**

1. Parent to child

i. Declare an @Input variable inside the child component

ii. Bind the variable to a parent function or variable with []

where the child is called

eg. [childProperty]="parentProperty"

2. Child to parent

i. Declare an @Output and initialize it as an EventEmitter() from '@angular/core'

ii. Specify a way to emit data to event, eg. a function that returns a value etc

iii.Bind on the child element call the output property to something from the parent component inside ()

eg. (childProperty)="ParentFunction(event)";

iv. Use the event inside the parent component