**.NET & Angular Course Note**

1. **Dotnet:**
   1. **Basic commands to set up project**

* Create solution file (container of the project): dotnet new sln
* Create API project: dotnet new webapi -o API
* Add API project to the solution: dotnet sln add API
* Start the application: dotnet run
* Start the application with file watcher: dotnet watch run
* Make browser trust the certificate: dotnet dev-certs https –trust
  1. **.Net queries:**
* ToListAsync(): return all entries
* Find(id): return entry with the specified id
* AnyAsync(x => x.{property} == {parameter}): return Boolean whether exists
* FirstOrDefault
* SingleOrDefault: similar to FirstOrDefault, but if there are 2 more elements, it will throw error
  1. **Controllers**
* Controllers provide routine end points for where we can find our application
* [Route("[**controller**]")]: “**controller”** will be replaced by the name of that controller
* Controller needs to derive from ControllerBase class
* [ApiController]:
* automatically binds any parameter it finds in the method (body or query string)
* automatically validates the parameters that we pass up to an API endpoint based on the validation
  1. **How the program starts**
* Program.cs: dotnet run will run whatever in Main method
* Startup.cs: ConfigureServices method is the dependency injection container
  + - * services.AddDbContext<DbContextDerivedClass>(options => {

options.UseSqlite(\_config.GetConnectionString(“DefaultConnection”))

})

* appsettings.json: “ConnectionStrings: {

“DefaultConnection”: “Data source=datingapp.db”

}

* orders don’t matter
* launchSettings.json: when we run application, it looks inside this file
  1. **Dotnet program structures**
* Entities: objects in the application
  + - * Id: hardcoded for ASP.NET Core entities
      * UserName: hardcoded for ASP.NET Core entities
  1. **Entity Framework (create an Entities folder inside the project folder)**
* An Object Relational Mapper (ORM)
* Translates code into SQL commands that update tables in the database
* Important class: DbContext is the primary class we use for interacting with database
* Features:
  + - * Querying
      * Change tracking
      * Saving
      * Concurrency
      * Transactions
      * Caching
      * Configurations
      * Migrations: create database schema
* Create migration: dotnet ef migrations add InitialCreate -o Data/Migrations
* Create database: dotnet ef database update
* Drop database: dotnet ef database drop
  1. **Sending Request**
* When we send something in the body of a request, must send them as an object
* ActionResult: when we use ActionResult, we’re able to return different HTTP status codes.
* DTO: data transfer object – usually add validation at DTO level
* CreatedAtAction: return 201 Response with Location in the headers of the response
  1. **JSON Web Tokens**
* Self-contained
* Contain credentials, claims, and other info
* Header, Payload, Verify Signature
* Header:
* Algorithm: algorithm used to encrypt the signature
* Type: JWT
* Payload:
  + Information about the claims and credentials
  + Nbf: date after which the token can be used
  + Exp: date before …
  + Ias: issued at
* Signature:
  + Encrypted by a secure key of the server
* Process:
  + User sends username and password
  + The server validates credentials and return a JWT
  + User now can send JWT with further requests
  + Server now can verify JWT and sends back response
* In startup class:

services.AddScoped<ITokenService, TokenService>();

//add token and inject to our application. This is scoped to the lifetime of the Http Request

* + Singleton: stay as long as the application is still running
  + Transient: stay as long as the method is finished
  + The server validates credentials and return a JWT
  1. **Middleware**
* RequestDelegate: our http request will be handled to this to communicate with the database
* RequestDelegate: what’s next in the middleware pipeline
* The middleware class must include:
  + A public constructor with a parameter of type RequestDelegate.
  + A public method named Invoke or InvokeAsync. This method must:
    - Return a Task.
    - Accept a first parameter of type HttpContext.
  1. **Repository pattern**
* Encapsulate the logic
* Reduce duplicate query logic
* Promote testability
* The idea is we can only provide the methods that we can support for different entities
  + - Note: Strongly typing in means that we do create a class for these settings and then retrieve the config by using that class rather than accessing via a string and reduces the possibility to make a typo as we will be notified at compile time if we try input the incorrect values here.

1. **Angular:**
   1. **Basic commands to set up project**

* Create new project: ng new client –strict false
* Start server: ng serve
* An Angular app contains a tree of Angular components. Decorator (TypeScript) gives more power to the class.
* app-root: app root components (app folder)
  + - * app.component.ts: contains component
      * Change tracking
      * Interpolation: pass data from to template/views (front end)
      * Module: responsible for bootstrapping components. Usually have a decorator to tell it’s an Angular module
    - Declare available components inside the application
    - Able to import other modules
    - Bootstrap: bootstrap every component
    - Main.ts: provide code to bootstrap modules
* Tsconfig.json: TypeScript needs this.
* Create a new component: ng g c <name> --skip-tests
  1. **Angular Form**
* Set a form to be Angular form: #formName=”ngForm” => (ngSubmit)=”customMethod()”
* \*ngIf: conditional appear
* (click): Angular onclick attribute
* [hidden]: conditional display
* 2-way binding:
  + - * Name=”customName”
      * [(ngModel)]=”model”
* FormGroup: group of FormControl
* FormControl derives from NgControl
* ValidatorFn: inside the validation array
* @Self: When we use @self we tell Angular that this is a service that we do not want to re-use and to create an instance specifically for this component. We do not want to re-use the ngControl here and needs to be unique for each input.
* Everything comes with reactive form derives from AbstractControl
  1. **Services**
* Create a service: ng g s <name> --skip-tests
* Services:
  + - * Can be injected into components or other services
      * Singletons
      * Making HTTP requests
* app.UseDeveloperExceptionPage: default error catcher
  1. **Observables**
* Lazy collection of multiple values over time
  + - * Only subscribers can receive the update
      * When subscribe:
  + What to do next with data
  + What to do when error
  + What to do when success (optional)
* We can send it to JavaScript promise
* Pipe(): rxjs extension. Used by an observable to return another observable
* Parent-to-child and Child-to-parent passing data
* Parent to child: []
* Child to parent: ()
  1. **Routing**

1. const routes: Routes = [
2. {path: '', component: HomeComponent},
3. {path: 'members', component: MemberListComponent},
4. {path: 'members/:id', component: MemberDetailComponent},
5. {path: 'lists', component: ListsComponent},
6. {path: 'messages', component: MessagesComponent},
7. {path: '\*\*', component: HomeComponent, pathMatch: 'full'}
8. ];
9. **Interceptors**

* Anytime we make a request to API, we don’t need to do anything with exceptions (the error interceptor is responsible for that)
* Only one place to get hold of the state of the router: inside the constructor of the component

1. **Component Access**

* @Input: get the variable from a parent component
* @Output: export the variable to a child component
* @ViewChild: access the template reference variable

1. **Syntax**

* Partial<Object>: make every property of the object optional

Link to full project: <https://github.com/TryCatchLearn/DatingApp>