

Q2 2023



SuperHeroProject

VEJLEDNING I FULLSTACK PROJEKTET, FRA START TIL SLUT

Jack Baltzer, jabb@tec.dk, 2023

Super Hero Project

Indholdsfortegnelse

PROJEKT OPSTART	4
GITHUB ASSIGNMENT	4
TILFØJ SOLUTION	5
TILFØJ API PROJECT	7
GIT COMMIT OG PUSH	9
GIT DEV BRANCH	9
STRUKTUREN	9
CONTROLLER LAGET	9
SERVICE LAGET	9
REPOSITORY LAGET	9
INTERFACES	9
DATABASEN	9
GLOBAL USINGS	10
ENTITYFRAMEWORKCORE	10
DATABASECONTEXT	11
CONNECTIONSTRING	11
ADD-MIGRATION	12
UPDATE-DATABASE	12
HEROREPOSITORY	13
<i>GetAllAsync()</i>	13
UNITTESTS	14
HEROREPOSITORYTESTS	14
<i>GetAllAsync_ShouldReturnListOfHeroes_WhenHeroesExists()</i>	15
<i>GetAllAsync_ShouldReturnEmptyListOfHeroes_WhenNoHeroesExists()</i>	16
HERORESPONSE DATATRANSFEROBJECT	17
HEROSERVICE	17
<i>GetAllAsync()</i>	18
MOQ	19
HEROSERVICETESTS	19
<i>GetAllAsync_ShouldReturnListOfHeroResponses_WhenHeroesExists()</i>	19
<i>GetAllAsync_ShouldReturnEmptyListOfHeroResponses_WhenNoHeroesExists()</i>	20
<i>GetAllAsync_ShouldThrowNullException_WhenRepositoryReturnsNull()</i>	20
HEROCONTROLLER	21
<i>GetAllAsync()</i>	21
HEROCONTROLLERTESTS	22
<i>GetAllAsync_ShouldReturnStatusCode200_WhenHeroesExists()</i>	22



Super Hero Project

<i>GetAllAsync_ShouldReturnStatusCode204_WhenNoHeroesExist()</i>	23
<i>GetAllAsync_ShouldReturnStatusCode500_WhenExceptionIsRaised()</i>	23
SWAGGER	24
CREATE HERO	25
IHEROREPOSITORY	25
<i>CreateAsync()</i>	25
HEROREPOSITORY	25
<i>CreateAsync()</i>	25
HEROREPOSITORYTESTS	25
<i>CreateAsync_ShouldAddNewIdToHero_WhenSavingToDatabase()</i>	25
<i>CreateAsync_ShouldFailToAddNewHero_WhenHeroIdAlreadyExists()</i>	26
HEROREQUEST	26
HEROSERVICE	27
<i>MapHeroToHeroResponse()</i>	27
<i>MapHeroRequestToHero()</i>	28
<i>GetAllAsync()</i>	28
IHEROSERVICE	28
<i>CreateAsync()</i>	28
HEROSERVICE	28
<i>CreateAsync()</i>	28
HEROSERVICETESTS	29
<i>CreateAsync_ShouldReturnHeroResponse_WhenCreateIsSuccess()</i>	29
<i>CreateAsync_ShouldThrowNullException_WhenRepositoryReturnsNull()</i>	29
HEROCONTROLLER	30
<i>CreateAsync()</i>	30
HEROCONTROLLERTESTS	30
<i>CreateAsync_ShouldReturnStatusCode200_WhenHeroIsSuccessfullyCreated()</i>	31
<i>CreateAsync_ShouldReturnStatusCode500_WhenExceptionIsRasied()</i>	31
SWAGGER	32
FIND EN HERO BASERET PÅ ID	32
IHEROREPOSITORY	33
<i>FindByIdAsync()</i>	33
HEROREPOSITORY	33
<i>FindByIdAsync()</i>	33
HEROREPOSITORYTESTS	33
<i>FindByIdAsync_ShouldReturnHero_WhenHeroExists()</i>	33
<i>FindByIdAsync_ShouldReturnNull_WhenHeroDoesNotExist()</i>	34
IHEROSERVICE	34
<i>FindByIdAsync()</i>	34
HEROSERVICE	34
<i>FindByIdAsync()</i>	34
HEROSERVICETESTS	34
<i>FindByIdAsync_ShouldReturnHeroResponse_WhenHeroExists()</i>	34
<i>FindByIdAsync_ShouldReturnNull_WhenHeroDoesNotExists()</i>	35
HEROCONTROLLER	35
<i>FindByIdAsync()</i>	35



Super Hero Project

HEROCONTROLLERTESTS	36
<i>FindByIdAsync_ShouldReturnStatusCode200_WhenHeroExists()</i>	36
<i>FindByIdAsync_ShouldReturnStatusCode404_WhenHeroDoesNotExist()</i>	36
<i>FindByIdAsync_ShouldReturnStatusCode500_WhenExceptionIsRaised()</i>	37
SWAGGER	37
OPDATER EN HERO	37
IHEROREPOSITORY	37
<i>UpdateByIdAsync()</i>	37
HEROREPOSITORY	37
<i>UpdateByIdAsync()</i>	37
HEROREPOSITORYTESTS	38
<i>UpdateByIdAsync_ShouldChangeValuesOnHero_WhenHeroExists()</i>	38
<i>UpdateByIdAsync_ShouldReturnNull_WhenHeroDoesNotExist()</i>	39
IHEROSERVICE	39
<i>UpdateByIdAsync()</i>	39
HEROSERVICE	39
<i>UpdateByIdAsync()</i>	39
HEROSERVICETESTS	39
<i>UpdateByIdAsync_ShouldReturnHeroResponse_WhenUpdateIsSuccess()</i>	39
<i>UpdateByIdAsync_ShouldReturnNull_WhenHeroDoesNotExists()</i>	40
HEROCONTROLLER	40
<i>UpdateByIdAsync()</i>	40
HEROCONTROLLERTESTS	41
<i>UpdateByIdAsync_ShouldReturnStatusCode200_WhenHeroIsUpdated()</i>	41
<i>UpdateByIdAsync_ShouldReturnStatusCode404_WhenHeroDoesNotExist()</i>	42
<i>UpdateByIdAsync_ShouldReturnStatusCode500_WhenExceptionIsRaised()</i>	42
SWAGGER	42
SLET EN HERO	42
IHEROREPOSITORY	43
<i>DeleteByIdAsync()</i>	43
HEROREPOSITORY	43
<i>DeleteByIdAsync()</i>	43
HEROREPOSITORYTESTS	43
<i>DeleteByIdAsync_ShouldReturnDeletedHero_WhenHeroIsDeleted()</i>	43
<i>DeleteByIdAsync_ShouldReturnNull_WhenHeroDoesNotExist()</i>	44
IHEROSERVICE	44
<i>DeleteByIdAsync()</i>	44
HEROSERVICE	44
<i>DeleteByIdAsync()</i>	44
HEROSERVICETESTS	45
<i>DeleteByIdAsync_ShouldReturnHeroResponse_WhenDeleteIsSuccess()</i>	45
<i>DeleteByIdAsync_ShouldReturnNull_WhenHeroDoesNotExists()</i>	45
HEROCONTROLLER	45
<i>DeleteByIdAsync()</i>	45
HEROCONTROLLERTESTS	46
<i>DeleteByIdAsync_ShouldReturnStatusCode200_WhenHeroIsDeleted()</i>	46



Super Hero Project

DeleteByIdAsync_ShouldReturnStatusCode404_WhenHeroDoesNotExist()	46
DeleteByIdAsync_ShouldReturnStatusCode500_WhenExceptionIsRaised()	46
SWAGGER	46
ANGULAR	47
FORSIDE KOMPONENTET	49
ROUTING OG LAZY-LOADING AF COMPOSER	49
APP.COMPONENT SOM FLAT, INLINE KOMPONENT	50
HERO MODEL	51
ANGULER HEROSERVICE	51
ENVIRONMENT VARIABLER	52
ANGULAR HEROSERVICE HTTPCLIENTMODULE	52
ANGULAR HEROSERVICE GETALL()	53
FRONTPAGECOMPONENT HEROSERVICE	54
CORS	54
ANGULAR OG GITHUB	55
ANGULAR HERO ADMINISTRATION	55
HEROADMIN VARIABLERNE OG OPSÆTNINGEN	57
HEROADMIN TEMPLATE	57
ANGULAR HEROADMIN DELETE HERO	59
ANGULAR HEROADMIN CREATE	60
ANGULAR HEROADMIN ANNULER FUNKTION	60
ANGULAR HEROADMIN EDIT	61
ANGULAR HEROSERVICE FINDBYID()	62
ANGULAR REACTIVE FORMS	64

Projekt Opstart

Github Assignment

Accepter Github Assignment via det link der ligger på Itslearning.



Super Hero Project

Når assignment er accepteret, og dit repository er færdig konfigureret af Github, så skal der tilføjes en [.gitignore](#) fil til repository.

The screenshot shows a GitHub repository page for 'super-hero-project-JackBaltzer'. In the 'Code' tab, there is a 'Quick setup — if you've done this kind of thing before' section. It includes instructions to 'Set up in Desktop' or 'HTTPS / SSH' and a link to the repository's URL. A note at the bottom of this section says: 'Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.' The word '.gitignore' is circled in red.

Vælg VisualStudio og commit tilføjelsen i bunden af siden.

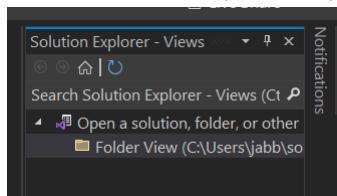
The screenshot shows the same GitHub repository page. In the 'Code' tab, a new file named '.gitignore' is being created in the 'main' branch. A dropdown menu 'Choose .gitignore template' is open, with 'VisualStudio' selected. This dropdown is circled in red. The code editor shows the default Visual Studio .gitignore template:

```
1 ## Ignore Visual Studio temporary files, build results, and
2 ## files generated by popular Visual Studio add-ons.
3 ##
4 ## Get latest from https://github.com/github/gitignore/blob/master/VisualStudio.gitignore
5
```

Tilføj Solution

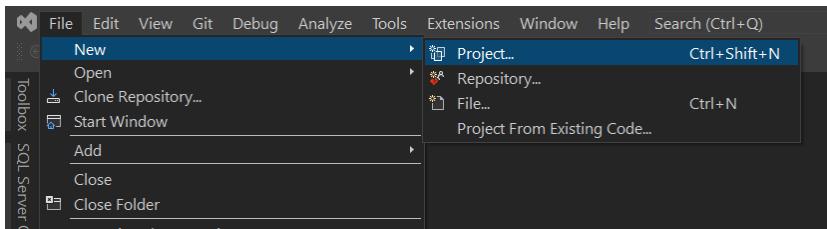
Efter [.gitignore](#) er tilføjet, kan vi trække repository ned til vores lokale computer, så opgaven kan løses. Sørg for at placere repository et fornuftigt sted. [Undlad synkroniseret mapper som dropbox og onedrive.](#)

Når Visual Studio op i en mappe uden nogen solutions eller projects.

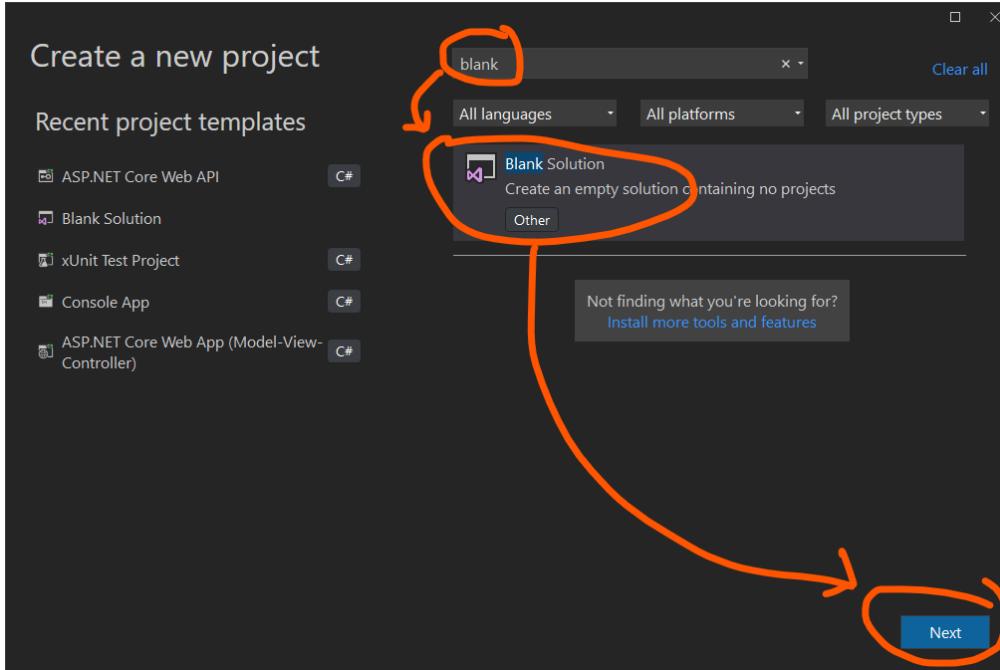


Super Hero Project

Det først vi tilføjer er en Blank Solution via **File -> New -> Project**.



Sørg for at vælge **Blank Solution** og klik videre.



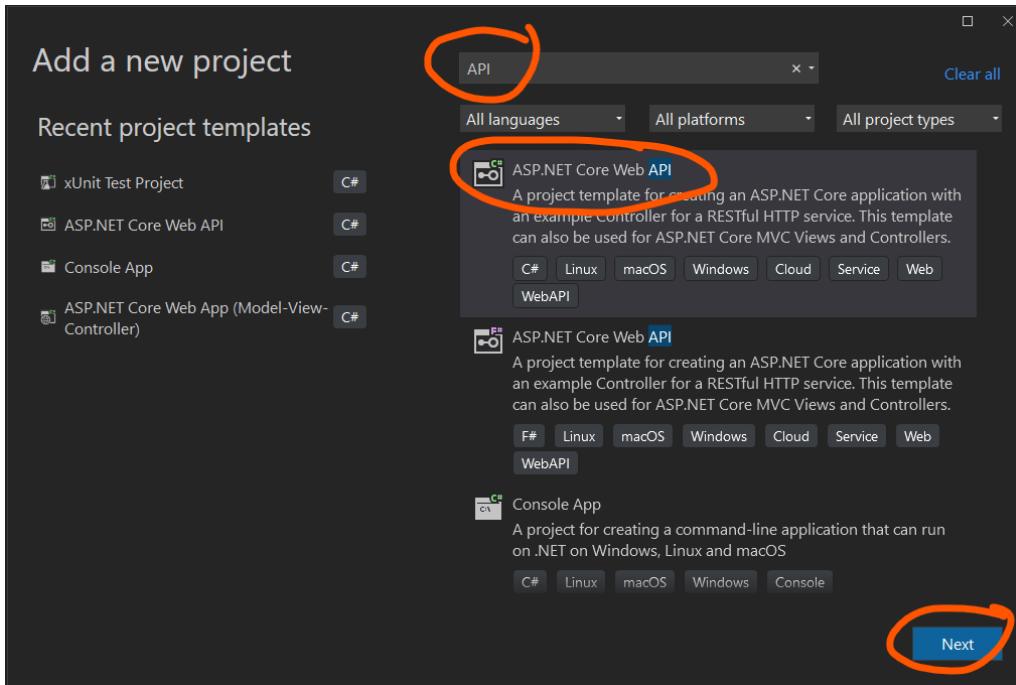
Giv din solution navnet **Super Hero Project** og sørge for placeringen er i roden af det repository du har hentet ned. Derefter har du en tom solution hvor du kan tilføje de projekter der skal arbejdes med i opgaven.



Super Hero Project

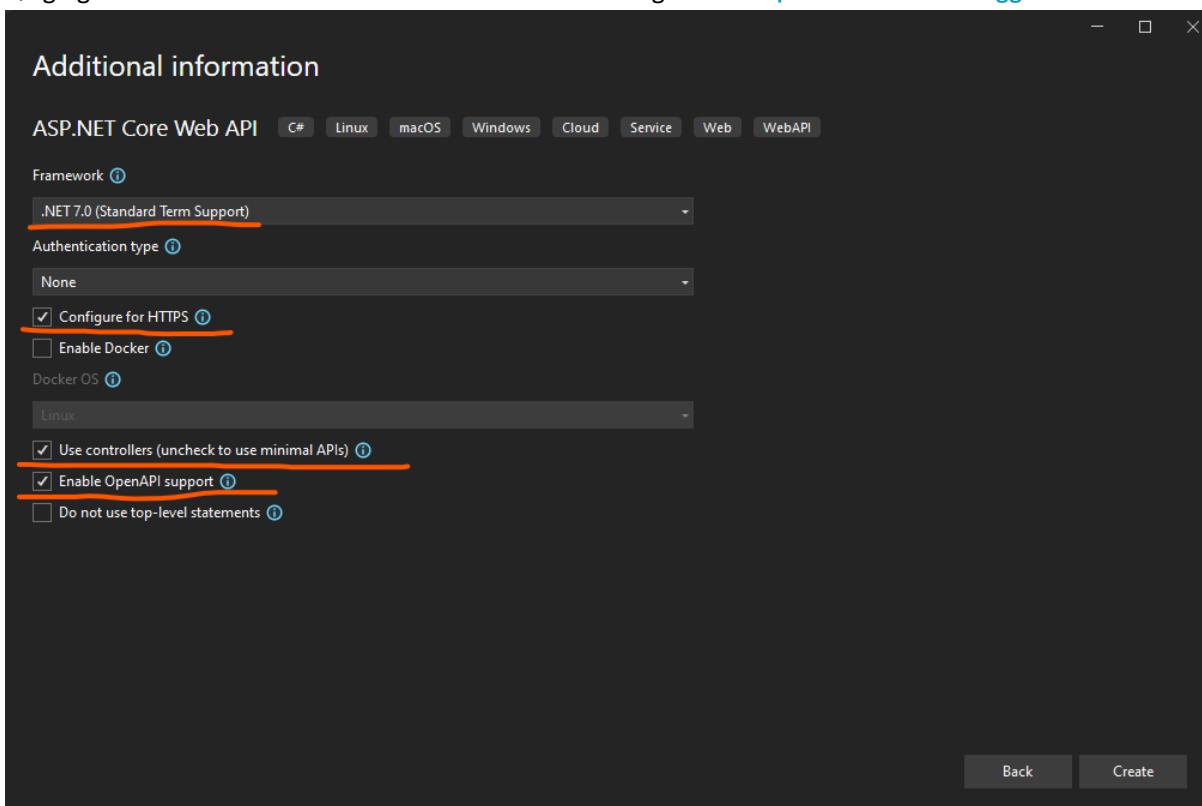
Tilføj API project

Så opret et nyt projekt, ved at højreklikke på solution og væge **Add -> New Project**. Sørg for at typen er **ASP.NET Core Web API (C#)** og kald det **SuperHeroAPI** da dette er selve **API** projektet.



Sørg for at framework er **.NET 7.0**, og at projektet er sat op til **HTTPS**.

Sørg også for at der er flueben ud for **Use Controllers** og **Enable OpenAPI** så vi får **Swagger** funktionalitet.



Super Hero Project

Når projektet er tilføjet, så find `Properties/launchSettings.json` og fjern profilen `IIS Express` da vi ikke er interesseret i at køre siden fra en IIS Express server, vi vil i stedet køre på en `Kestrel` server instans. Så får vi en CommandPromt vindue hvor vi kan se fejl og andre interessante logs.

```
        },
    ],
    "IIS Express": [
        "commandName": "IISExpress",
        "launchBrowser": true,
        "launchUrl": "swagger",
        "environmentVariables": {
            "ASPNETCORE_ENVIRONMENT": "Development"
        }
    ]
}
```

Jeg går ud fra du har helt styr på krølleparanteser og kommaer.

Derefter trykkes **[CTRL] + [F5]** for at builde og launche projektet, så du kan bekræfte at altting fungerer. Det er muligt der kommer en advarsel om sikkerhedsrisiko, fordi vi benytter et `self-signed-certificate` til at håndtere `HTTPS` forbindelsen. Accepter risikoen.

Nu burde `Swagger` blive indlæst i browser og du kan teste WeatherForecast:

The screenshot shows the Swagger UI interface for the SuperHeroAPI v1. At the top, it displays the URL `https://localhost:7150/swagger/index.html`. Below this, the title "SuperHeroAPI 1.0 OAS3" is shown, along with the definition URL `https://localhost:7150/swagger/v1/swagger.json`.

The main content area is titled "WeatherForecast". It shows a "GET /WeatherForecast" operation. Under the "Parameters" section, it says "No parameters". Below this are "Responses" and "Code" sections. The "Code" section for status code 200 shows a "Response body" containing the following JSON array:

```
[{"date": "2022-10-27T10:15:52.1234472+02:00", "temperatureC": -12, "temperatureF": 11, "summary": "Warm"}]
```



Super Hero Project

Når du har set det virker, så **slet** filerne `WeatherForecast.cs` og `WeatherForecastController.cs`, da vi ikke længere skal bruge dem til noget.

Git commit og push

Commit og push de nuværende ændringer til Github, skriv en passende besked til commit, f.eks `SuperHeroAPI added` da det er hvad vi har foretaget på nuværende tidspunkt.

Git dev branch

Opret en ny udvikler branch kaldet `dev`, så `main` kan holdes til funktionel (og gennemtestet) kode, al kode du skriver commites til `dev branch`, og når du er færdig med en feature, så merges den færdige kode ind i `main` via en `pull request`.

DU MÅ IKKE KODE PÅ MAIN! Det er forbudt.

Strukturen

API projektet bliver delt op i nogle lag som hver især har et specifikt formål.

Controller Laget

Her håndteres de forespørgsler der kommer ind udefra, og svar returneres med relevant data.

Service Laget

Når controlleren modtager data, er det et DataTransferObject der modtages, det objekt skal oversættes til en Database Entitet, som sendes til Repository laget. Og når der kommer svar fra Repository, er det også en Database Entitet, som skal sendes til Controlleren i form af et DataTransferObjekt.

Repository Laget

Når data skal gemmes i databasen, er det repository laget der udfører arbejdet. Der kommer data fra servicelaget, som indsættes eller opdateres i databasen, og derefter returneres data tilbage til servicelaget.

Interfaces

For at sikre hver klasse har adgang til det korrekte lag, benytter vi interfaces og Dependency Injection.

Det betyder at Repository laget og Service laget implementers via et interface, og vi sætter Program.cs op til at servere de konkrete klasser efter behov.

Interfaces giver os også et værktøj til at teste hvert lag, da vi kan styre meget præcist hvad der skal ske i et test scenarie ved at benytte Moq (mere om det senere!)

Databasen

Opret en mappe kaldet `Database` i roden af `SuperHeroAPI` projektet. I denne mappe placeres de værktøjer der er nødvendige for at kunne arbejde med databasen. Vi benytter en `Code First` tilgang, dvs der er endnu ikke oprettet nogen database eller tabeller, alt sammen sker via koder fra vores projekt.



Super Hero Project

Databasen deles op i to koncepter, det ene er data-objekter vi skal kunne arbejde med **Entiteter**, og det andet er hvordan de data-objekter hænger sammen i forhold til hinanden **Context**.

Vi får én **context**, men flere **entiteter**, så opret en mappe kaldet **Entities** inde i **Database** mappen. I **Entities** mappen oprettes en klasse kaldet **Hero.cs**, det er den klasse der kommer til at repræsentere en helt i vores system.

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;

namespace SuperHeroAPI.Database.Entities
{
    public class Hero
    {
        [Key]
        public int Id { get; set; }

        [Column(TypeName = "nvarchar(32)")]
        public string HeroName { get; set; } = string.Empty;

        [Column(TypeName = "nvarchar(32)")]
        public string RealName { get; set; } = string.Empty;

        [Column(TypeName = "nvarchar(32)")]
        public string Place { get; set; } = string.Empty;

        [Column(TypeName = "smallint")]
        public short DebutYear { get; set; } = 0;
    }
}
```

Global Usings

Noget forholdsvis nyt i **.NET**, er muligheden for at trække **usings** ud i en central placering, frem for at have hver fil starte med en masse using-statements. Der er masser af fordele (og enkelte ulemper) ved globale usings, så lad os afprøve konceptet her.

Opret en fil kaldet **Usings.cs** i rodten af **SuperHeroAPI** projektet. Og alle usings der er i brug i alle filer lige nu, flyttes fra filen og over i **Usings.cs**, hvor der tilføjes **global** foran.

```
global using System.ComponentModel.DataAnnotations.Schema;
global using System.ComponentModel.DataAnnotations;
```

Fremover flytter vi manuelt alle usings ind i denne fil, og sætter dem som global. Det skal du selv huske på!

Der hvor der kan opstå udfordringer med global usings, er hvis to klasser hedder det samme, men ligger i forskellige namespaces. Det kunne f.eks. være **SuperHeroAPI.Database.Entities.Hero** og **SuperHeroAPI.Migrations.Hero** der vil opstå en **ambiguous reference** som skal håndteres... den tager vi når den opstår.

EntityFrameworkCore

Da vi skal arbejde med **Object Relations Mapping**, har jeg valgt **EntityFrameworkCore** som vores Mapper. Den tilføjes til projektet via Nuget packes eller PackageManagerConsole.

Installer følgende pakker, vælg den nyeste version 7.x.x for dem alle:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Tools



Super Hero Project

- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Design

DatabaseContext

Opret en klasse-fil kaldet `DbContext.cs` i roden af `Database` mappen.

Lad klassen `DbContext` nedarve fra `DbContext`, som ligger i `Microsoft.EntityFrameworkCore`, den du tilføjer i din globale usings. Tilføj også `Database.Entities` til din globale usings.

I `DbContext` skal vi have en constructor der kan modtage nogle `DbContextOptions`, som benyttes når context instancieres de forskellige steder vi kommer til det (tests og repositories).

Derudover skal vi tilføje `DbSet<Hero>` så vi får en tabel kaldet `Hero` med de felter `Hero` entiteten indeholder.

Og til sidst opsættes en `seeding` af databasen, så der er lidt data at arbejde med.

```
namespace SuperHeroAPI.Database
{
    public class DatabaseContext : DbContext
    {
        public DatabaseContext(DbContextOptions<DatabaseContext> options) : base(options) { }

        public DbSet<Hero> Hero { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Hero>().HasData(
                new Hero
                {
                    Id = 1,
                    HeroName = "Superman",
                    RealName = "Clark Kent",
                    Place = "Metropolis",
                    DebutYear = 1938
                },
                new Hero
                {
                    Id = 2,
                    HeroName = "Iron Man",
                    RealName = "Tony Stark",
                    Place = "Malibu",
                    DebutYear = 1963
                });
        }
    }
}
```

ConnectionString

Database forbindelsen ændres afhængig af om vi arbejder lokalt, om vi tester, eller om vi er på en server. Derfor oprettes en connectionstring i `appsettings.json` filen. Her kan flere tilføjes efter behov, og vi kan skrive lidt logik der henter den korrekte connectionstring senere.

I mit eksempel arbejder jeg på en `LocalDB`, da det er en ret simpel fil-baseret database og den kan løse alle de udfordringer vi har behov for her i dette projekt.



Super Hero Project

Tilføj følgende `ConnectionStrings` område til `appsettings.json`:

```
{  
  "ConnectionStrings": {  
    "Default": "Data Source = (localdb)\\MSSQLLocalDB; Initial Catalog = SuperHeroProject"  
  },  
  "Logging": {  
    "LogLevel": {  
      "Default": "Information",  
      "Microsoft.AspNetCore": "Warning"  
    }  
  },  
  "AllowedHosts": "*"  
}
```

Og i `Program.cs` tilføjes `AddDbContext` til `builder.Services`, med `UseSqlServer` opsætningen.

```
var builder = WebApplication.CreateBuilder(args);  
  
// Add services to the container.  
builder.Services.AddDbContext<DatabaseContext>(options =>  
{  
  options.UseSqlServer(builder.Configuration.GetConnectionString("Default"));  
});  
  
builder.Services.AddControllers();
```

Add-Migration

For at oprette database og tabellen `Hero`, skal vi køre en Package Manager Console kommando:

`Add-Migration Hero`

```
Package source: All | Default project: SuperHeroAPI  
PM> Add-Migration Hero  
Build started...  
Build succeeded.  
Microsoft.EntityFrameworkCore.Infrastructure[10403]  
  Entity Framework Core 6.0.10 initialized 'DatabaseContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer:6.0.10' with options: None  
To undo this action, use Remove-Migration.  
PM>
```

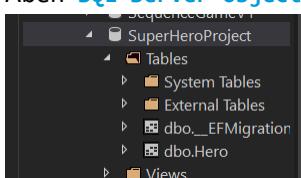
Derefter opretter EFCore en mappe kaldet `Migrations` og indsætter nogle filer i den, som beskriver hvordan databasen ser ud og hvordan den skal oprettes.

Update-Database

Databasen er IKKE oprettet på nuværende tidspunkt, det sker først når kommandoen `Update-Database` køres, og resultatet er en masse SQL der afvikles, og consollen ender med at se sådan her ud:

```
Package Manager Console  
Package source: All | Default project: SuperHeroAPI  
SET IDENTITY_INSERT [hero] ON;  
INSERT INTO [Hero] ([Id], [DebutYear], [HeroName], [Place], [RealName])  
VALUES (2, CAST(1963 AS smallint), N'Iron Man', N'Malibu', N'Tony Stark');  
IF EXISTS (SELECT * FROM [sys].[identity_columns] WHERE [name] IN (N'Id', N'DebutYear', N'HeroName', N'Place', N'RealName') AND [object_id] = OBJECT_ID(N'[Hero]'))  
  SET IDENTITY_INSERT [hero] OFF;  
Microsoft.EntityFrameworkCore.Database.Command[20101]  
  Executed DbCommand (1ms) [Parameters=[], CommandType='Text', CommandTimeout='30']  
  INSERT INTO [_EFMigrationsHistory] ([MigrationId], [ProductVersion])  
  VALUES (N'20221026105039_Hero', N'6.0.10');  
Done.  
PM>
```

Åben `SQL Server Object Explorer` vinduet i Visual Studio, og se den nye database derinde:



Super Hero Project

Højreklik på `dbo.Hero` og vælg **View Data** for at se der er indsat de to helte fra `DbContext` filen.

Max Rows: 1000					
	Id	HeroName	RealName	Place	DebutYear
1	1	Superman	Clark Kent	Metropolis	1938
2	2	Iron Man	Tony Stark	Malibu	1963
	NULL	NULL	NULL	NULL	NULL

Commit til Github.

HeroRepository

Nu er vi klar til at arbejde på databasen, til det skal vi bruge Repository klassen. Formålet med den klasse er at repræsentere de handlinger vi kan udføre på databasen, på den Hero entiteten.

Opret en mappe kaldet **Repositories** i roden af **SuperHeroAPI** projektet, og opret en klasse-fil kaldet **HeroRepository.cs** i den mappe.

Da vi skal tilføje tests til vores projekt, og da den service vi bygger senere skal benytte dette **HeroRepository**, så opretter vi et interface der beskriver hvad **HeroRepository** kan gøre.

I mine eksempler opretter jeg **IHeroRepository** inde i samme fil som **HeroRepository**, for at holde antallet af filer nede. Men det er også ok at oprette en fil specifikt til interfacet.

GetAllAsync()

Til at starte med koncentrer vi os om den enkelte metode til at hente alle Heroes ud af databasen. Og da vi arbejder med **Tasks** forventes det at metoderne er **async**, derfor har jeg tilføjet **Async** til metodenavnet.

HeroRepository skal også gøre brug af den **ConnectionString** vi har defineret, det sker i constructor metoden.

```
namespace SuperHeroAPI.Repositories
{
    public interface IHeroRepository
    {
        Task<List<Hero>> GetAllAsync();
    }

    public class HeroRepository : IHeroRepository
    {
        private readonly DatabaseContext _context;

        public HeroRepository(DatabaseContext context)
        {
            _context = context;
        }

        public async Task<List<Hero>> GetAllAsync()
        {
            return await _context.Hero.ToListAsync();
        }
    }
}
```

Det sidste der mangler, er at håndtere **Dependency Injection**. Det klares ved at tilføje en **AddScoped** kommando i **Program.cs**. Dette kunne også være en **AddTransient** i stedet, men netop ved database Dependency Injection



Super Hero Project

giver det mening at have den samme instans igennem hele requesten, frem for en ny ved hver metode kald.

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IHheroRepository, HeroRepository>();

// Add services to the container.
builder.Services.AddDbContext<DatabaseContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("Default"));
});
```

Commit til Github.

UnitTests

Formålet med UnitTests er at sikre den kode vi skriver, den fungerer som vi forventer. Og specifikt imens vi udvikler koden, er det vigtigt at vide hvornår vi får skrevet kode der får noget andet kode til at fejle.

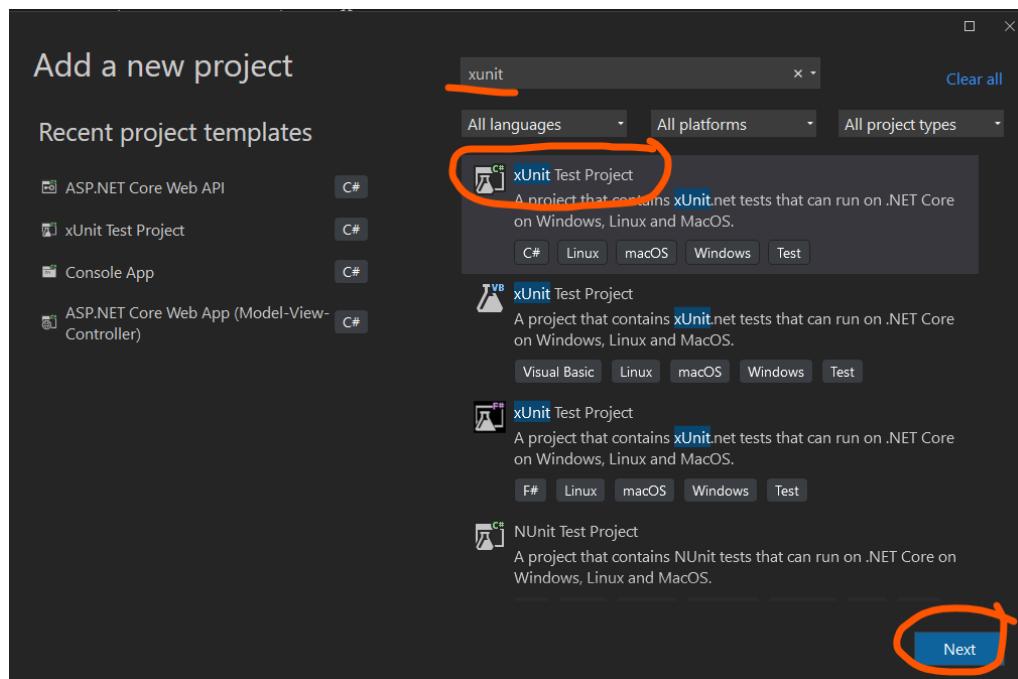
Derfor benytter vi UnitTests i dette projekt. Hver metode vi skriver, sørger vi også for at skrive tests så vi hele tiden kan tjekke at koden er ok.

Nu har vi en metode i [HeroRepository](#), som skal returnere en liste af [Hero](#). Dvs vi forventer der altid er en liste, og ikke pludselig et [null](#) objekt, eller en enkelt [Hero](#).

Vi skal have et nyt projekt til at håndtere alle tests. Vi opretter [ET](#) testprojekt, som kommer til at indeholde samtlige tests på samtlige lag... dvs HeroRepository, HeroService og HeroController, samt senere skal du selvstændigt tilføje TeamRepository, TeamService og TeamController. Alle tests samles i det test projekt vi opretter nu.

HeroRepositoryTests

Højreklik på [Super Hero Project](#) solution, og vælg [Add Project](#), kald projektet for [SuperHeroTests](#) og sørge for det er [.NET 7](#).



Super Hero Project

Opret en projekt reference til [SuperHeroAPI](#), og slet den [UnitTest1.cs](#) fil der er oprettet som standard.

Det er smart at sætte projektstrukturen op fornuftigt fra starten af. Vi ved der skal testes på [Repositories](#), [Services](#) og [Controllers](#), så opret de tre mapper i roden af "SuperHeroTests" projektet.

I [Repositories](#) mappen, oprettes en klasse-fil kaldet [HeroRepositoryTests.cs](#).

Inden vi kan forsætte, skal vi installere endnu en NugetPackage: [Microsoft.EntityFrameworkCore.InMemory](#) på [SuperHeroTests](#) projektet.

Dette skal gøres da vores tests ikke skal ændre på den eksisterende database, men i stedet køres de på en hurtig hukommelsesbaseret database der droppes efter hver test.

Start med at rette [internal](#) til [public](#), så test klassen bliver tilgængelig for Test Explorer.

Derefter sættes constructor metoden op, så den arbejder på [InMemoryDatabase](#) og instancierer et [HeroRepository](#) objekt med den database instans.

```
namespace SuperHeroTests.Repositories
{
    public class HeroRepositoryTests
    {
        private readonly DbContextOptions<DbContext> _options;
        private readonly DatabaseContext _context;
        private readonly HeroRepository _heroRepository;

        public HeroRepositoryTests()
        {
            _options = new DbContextOptionsBuilder<DbContext>()
                .UseInMemoryDatabase(databaseName: "HeroRepositoryTests")
                .Options;

            _context = new(_options);
            _heroRepository = new(_context);
        }
    }
}
```

[GetAllAsync_ShouldReturnListOfHeroes_WhenHeroesExists\(\)](#)

Den første test vi skriver, i [HeroRepositoryTests](#) klassen, er den hvor vi tester at der kommer en liste med 2 Heroes ud af databasen.

I den nedenstående kode, læg mærke til [\[Fact\]](#) oven over metode definitionen. Den attribut gør at [XUnit](#) ved denne test skal køre 1 gang når vi prøver at køre testen.

Ved [//Arrange](#) området starter vi med at sikre at databasen er helt tom, det er for at forhindre andre tests senere hen ændre på databasen når testen kører (*vi er async, der kan ske ting i en anden rækkefølge end vi forventer*)

OBS, det er her vi kan støde på den føromtalte "ambiguous reference" så vær 100% sikker på du IKKE refererer til Migrations mappen... der vil ALDRIG være behov for at pege på Migrations her. Hero er Database.Entities.Hero

Når databasen er ryddet, indsætter vi [2 Heroes](#) så vi ved der er 2 helte i tabellen.

Derefter udføres selve testen under [//Act](#) området. Det er her vi kører den metode vi ønsker at teste.



Super Hero Project

I `//Assert` området ser vi på resultatet af metoden. Først sikrer vi at resultatet `ikke er null`, derefter bekræfter vi at det er en `List<Hero>`, og afslutningsvis tjekker vi at der er **2** elementer i listen.

```
[Fact]
public async void GetAllAsync_ShouldReturnListOfHeroes_WhenHeroesExists()
{
    // Arrange
    await _context.Database.EnsureDeletedAsync();

    _context.Hero.Add(new Hero
    {
        Id = 1,
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    });

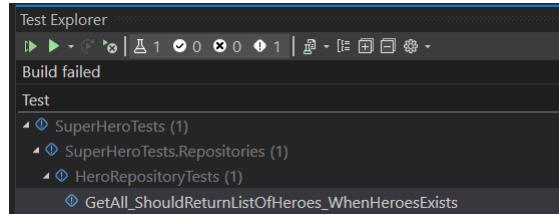
    _context.Hero.Add(new Hero
    {
        Id = 2,
        HeroName = "Iron Man",
        RealName = "Tony Stark",
        Place = "Malibu",
        DebutYear = 1963
    });

    await _context.SaveChangesAsync();

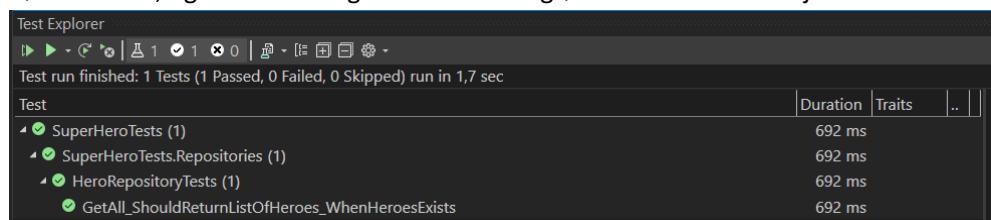
    // Act
    var result = await _heroRepository.GetAllAsync();

    // Assert
    Assert.NotNull(result);
    Assert.IsType<List<Hero>>(result);
    Assert.Equal(2, result.Count);
}
```

Find `Test Explorer` vinduet, og se der er fundet 1 test i projektet:



Kør alle tests, og der burde vi gerne ende med grønne flueben hele vejen ned i test-træet



GetAllAsync_ShouldReturnEmptyListOfHeroes_WhenNoHeroesExists()

Da testen er meget tæt på den forrige, kan det svare sig at kopiere hele testen, og ændre lidt på koden.

Navnet skal ændres, da vi nu forventer at teste at der kommer en tom liste ud af databasen.

Vi skal stadig sikre databasen er helt tom.



Super Hero Project

Det er vigtigt at vi tester at det er en tom liste der hentes, det klarer `Assert.Empty()` metoden.

```
[Fact]
public async void GetAllAsync_ShouldReturnEmptyListOfHeroes_WhenNoHeroesExists()
{
    // Arrange
    await _context.Database.EnsureDeletedAsync();

    // Act
    var result = await _heroRepository.GetAllAsync();

    // Assert
    Assert.NotNull(result);
    Assert.IsType<List<Hero>>(result);
    Assert.Empty(result);
}
```

Kør tests og se de begge er succesfulde.

Commit til Github.

HeroResponse DataTransferObject

Når vi får listen af helte ud af **Controlleren** engang ude i fremtiden, så skal det ikke være selve **Database Entiteten** der sendes direkte op igennem alle lagene og direkte ud til brugeren. Det vil som såden ikke være et problem med **Hero**, men hvad nu hvis det var et **User** objekt, med **passwords** og lign. Der skal vi sikre det kun er de data **VI** vælger, som sendes ud af API.

Til det formål benyttes et **DataTransferObject**. Vi opretter en mappe kaldet **DTOs** i roden af **SuperHeroAPI** projektet, og i den mappe oprettes en klasse-fil kaldet **HeroResponse.cs**, hvor alle de egenskaber vi vil eksponere til omverdenen beskrives:

```
namespace SuperHeroAPI.DTOs
{
    public class HeroResponse
    {
        public int Id { get; set; }

        public string HeroName { get; set; } = string.Empty;

        public string RealName { get; set; } = string.Empty;

        public string Place { get; set; } = string.Empty;

        public short DebutYear { get; set; }
    }
}
```

Så er vi klar til at oprette **HeroService**, som kan returnere en liste af **HeroResponses**.

Husk at tilføje en **global using SuperHeroAPI.DTOs**

HeroService

Nu hvor vi kan trække helte ud af databasen, og vi ved koden kan bestå testen. Så skal vi have tilføjet den service som kommer til at stå for kommunikationen imellem Controller og Repository.

Tilføj en mappe kaldet **Services** i roden af **SuperHeroAPI** projektet, og tilføj en klasse-fil kaldet **HeroService.cs** til den mappe.



Super Hero Project

Ved [HeroRepository](#), har vi et interface til at beskrive klassen, det samme skal vi tilføje ved [HeroService](#). Så kan vi benytte os af [Dependency Injection](#), og vi får lettere ved at skrive tests.

Så opret et [interface](#) kaldet [IHheroService](#) sammen med [HeroService](#) klassen.

Vi koncentrer os stadig kun om [GetAllAsync\(\)](#)

I [HeroService](#) constructoren, sætter vi klassen op med Dependency Injection, så den modtager interfacet af vores [HeroRepository](#)... dette er vigtigt, da det hjælper os til at skrive tests lige om lidt.

```
namespace SuperHeroAPI.Services
{
    public interface IHheroService
    {
        Task<List<HeroResponse>> GetAllAsync();
    }

    public class HeroService : IHheroService
    {
        private readonly IHheroRepository _heroRepository;

        public HeroService(IHheroRepository heroRepository)
        {
            _heroRepository = heroRepository;
        }

        public Task<List<HeroResponse>> GetAllAsync()
        {
            throw new NotImplementedException();
        }
    }
}
```

GetAllAsync()

[GetAllAsync](#) metoden skal starte med at trække på [HeroRepository.GetAllAsync\(\)](#), som vi ved returnere en liste.

Dog vil jeg gerne demonstrere hvordan man kan sikre sin kode endnu bedre, ved at vi håndterer hvis heroes alligevel er [null](#) efter vi har været i databasen. Der kaster vi en [ArgumentNullException](#).

Skulle vi komme forbi exception, så returneres listen af [Hero](#), dog skal vi lige huske at omsætte [Entiteten Hero](#) til en [HeroResponse](#).

```
public async Task<List<HeroResponse>> GetAllAsync()
{
    List<Hero> heroes = await _heroRepository.GetAllAsync();

    // if heroes are null at this point, something has gone wrong, throw an exception
    if (heroes == null)
    {
        throw new ArgumentNullException();
    }

    return heroes.Select(hero => new HeroResponse
    {
        Id = hero.Id,
        HeroName = hero.HeroName,
        RealName = hero.RealName,
        Place = hero.Place,
        DebutYear = hero.DebutYear
    }).ToList();
}
```

Husk at tilføje en [AddScoped](#) til [Program.cs](#)

```
builder.Services.AddScoped<IHheroService, HeroService>();
```



Super Hero Project

Moq

Det smarte ved tests, er at vi kan **simulere** funktionalitet så vi altid har 100% styr på hvad en test gør. Og specielt når vi allerede har sat klasserne op med interfaces, så bliver det endnu lettere.

Der findes flere nugetPackages som kan hjælpe os med dette, og her har jeg valgt **moq** da det virker som et af de mest populære af slagsen.

Installer **moq** via NugetPacketManager, og så er vi klar til **HeroServiceTests**.

HeroServiceTests

Vi skal have en klasse til at teste **HeroService** funktionaliteten, så opret en klasse-fil kaldet **HeroServiceTests.cs** i **Services** mappen i **SuperHeroTests** projektet. Sæt klassen op så den kan instanciere en **HeroService** og sikre at den **HeroService** har et **Mock.Object** af **IHeroRepository** klassen:

```
namespace HeroTests.Services
{
    public class HeroServiceTests
    {
        private readonly HeroService _heroService;
        private readonly Mock<IHeroRepository> _heroRepositoryMock = new();

        public HeroServiceTests()
        {
            _heroService = new(_heroRepositoryMock.Object);
        }
    }
}
```

Der er 3 scenarier vi skal teste.

Første test er den hvor vi sikre der kommer en liste af **HeroResponses** ud af servicen, når der er **Heroes** i databasen.

GetAllAsync_ShouldReturnListOfHeroResponses_WhenHeroesExists()

Vi starter med at sætte den liste af **Heroes** (entiteten) op, som vi vil have Repository skal returnere.

Derefter binder vi **Mock** op så når vi rammer metoden **GetAllAsync()** i Repository, så skal den blot returnere listen af heroes.

Det er vigtigt at forstå, at vi ALDRIG vil ramme den faktiske **HeroRepository** klasse her. Vi simulerer at vi rammer den.

Det er også vigtigt at forstå der kan være tests hvor man skal sætte flere forskellige **moq** metoder op. Samtlige metoder der bor i en klasse der injectes via Dependency Injection, skal sættes op når de er relevante.



Super Hero Project

Act og **Assert** er lige som i Repository testen.

```
[Fact]
public async void GetAllAsync_ShouldReturnListOfHeroResponses_WhenHeroesExists()
{
    // Arrange
    List<Hero> heroes = new()
    {
        new()
        {
            Id = 1,
            HeroName = "Superman",
            RealName = "Clark Kent",
            Place = "Metropolis",
            DebutYear = 1938
        },
        new()
        {
            Id = 2,
            HeroName = "Iron Man",
            RealName = "Tony Stark",
            Place = "Malibu",
            DebutYear = 1963
        }
    };

    _heroRepositoryMock
        .Setup(x => x.GetAllAsync())
        .ReturnsAsync(heroes);

    // Act
    var result = await _heroService.GetAllAsync();

    // Assert
    Assert.NotNull(result);
    Assert.IsType<List<HeroResponse>>(result);
    Assert.Equal(2, result?.Count);
}
```

GetAllAsync_ShouldReturnEmptyListOfHeroResponses_WhenNoHeroesExists()

Testen der kigger på den tomme liste, er rimelig simpel at sætte op når vi har den forrige test. Kopier, og ret den til så det er en tom liste der returneres. Husk at rette til så det er en **Assert.Empty()** der kaldes til sidst.

```
[Fact]
public async void GetAllAsync_ShouldReturnEmptyListOfHeroResponses_WhenNoHeroesExists()
{
    // Arrange
    List<Hero> heroes = new();

    _heroRepositoryMock
        .Setup(x => x.GetAllAsync())
        .ReturnsAsync(heroes);

    // Act
    var result = await _heroService.GetAllAsync();

    // Assert
    Assert.NotNull(result);
    Assert.IsType<List<HeroResponse>>(result);
    Assert.Empty(result);
}
```

GetAllAsync_ShouldThrowNullException_WhenRepositoryReturnsNull()

Den sidste test på **GetAllAsync**, er den hvor vi tester **ArgumentNullException** scenariet.

Læg mærke til hvordan **ReturnsAsync()** er sat op til at kaste en exception.

For at undgå der rent faktisk bliver kastet en exception, er vi nødt til at sætte **Act** op lidt anderledes. Vi opretter en **async Task action()** som kan udføres i et test kald i **Assert.ThrowsAsync()** som tager sig af exception



Super Hero Project

håndteringen så den ikke afbryder testen.

```
[Fact]
public async void GetAllAsync_ShouldThrowNullException_WhenRepositoryReturnsNull()
{
    // Arrange
    List<Hero> heroes = new();

    _heroRepositoryMock
        .Setup(x => x.GetAllAsync())
        .ReturnsAsync(() => throw new ArgumentNullException());

    // Act
    async Task action() => await _heroService.GetAllAsync();

    // Assert
    var ex = await Assert.ThrowsAsync<ArgumentNullException>(action);
    Assert.Contains("Value cannot be null", ex.Message);
}
```

Det er en **success**, når denne test støder på en **ArgumentNullException**.

Gem alt og Commit til Github.

HeroController

Nu er vi faktisk klar til at sætte en **Controller** op som kan returnere en liste af Heroes.

Opret en **Empty API Controller** i mappen **Controllers** i **SuperHeroAPI** projektet. Kald den for **HeroController.cs**.

Det er vigtigt det er en **API controller** og ikke en **MVC controller**!

Controlleren skal være afhængig af **HeroService**, så sørge for at sætte constructoren op så den modtager **IHeroService**:

```
namespace SuperHeroAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class HeroController : ControllerBase
    {
        private readonly IHeroService _heroService;

        public HeroController(IHeroService heroService)
        {
            _heroService = heroService;
        }
    }
}
```

GetAllAsync()

Når en klient rammer vores API er det controlleren der håndterer forespørgslen. Der skal være en route og en metode til hver af de forespørgsler vi ønsker at håndtere. Uover at definere et endpoint til vores route, så skal vi også bestemme hvilken slags forespørgsel vi ønsker at tillade.

I hent alle helte scenariet, er det et ønske at brugere kan ramme endpoint: **/api/hero** og det skal være et **GET** kald der benyttes. Controlleren allerede har en overordnet route: **Route("api/[controller]")** som oversættes til **api/hero**, da controlleren hedder **HeroController**. Derfor kan vi sætte GetAll håndteringen op på efterfølgende billede.



Super Hero Project

Læg mærke til retur typen, det er en `Task<IActionResult>` som er et værktøj der giver os muligheden for at generere passende `HttpStatusCodes` samt returnere data når det er relevant.

`Try/catch` blokkens formål er at håndtere de exceptions der måske kan opstå, så vores API forbliver responsiv selv om der skulle opstå exceptions.

```
[HttpGet]
public async Task<IActionResult> GetAllAsync()
{
    try
    {
        List<HeroResponse> heroes = await _heroService.GetAllAsync();

        if (heroes.Count() == 0)
        {
            return NoContent();
        }

        return Ok(heroes);
    }
    catch (Exception ex)
    {
        return Problem(ex.Message);
    }
}
```

HeroControllerTests

Lige som ved de to andre lag, skal vi have en separat klasse til at teste `HeroController`. Opret en klasse-fil kaldet `HeroControllerTests` i `Controllers` mappen, i `SuperHeroTests` projektet.

Sørg for constructor kan opsætte en instans af `HeroController`, med et `HeroServiceMock`.

```
namespace SuperHeroTests.Controllers
{
    public class HeroControllerTests
    {
        private readonly HeroController _heroController;
        private readonly Mock<IHeroService> _heroServiceMock = new();

        public HeroControllerTests()
        {
            _heroController = new(_heroServiceMock.Object);
        }
    }
}
```

GetAllAsync_ShouldReturnStatusCode200_WhenHeroesExists()

Den første test på `HeroController`, bør være det scenarie hvor det går godt og der returneres en statuskode **200(Ok)**

Opsætningen minder meget om `GetAllAsync` testen på service laget, dog er det `HeroResponses` vi ser på her. Først oprettes en liste af `HeroResponses`, som vores `HeroServiceMock` kan servere.

Derefter køres `HeroController.GetAllAsync()`, men vi er kun interesseret i statuskoden, så vi caster resultatet til et `IStatusCodeActionResult` med det samme.



Super Hero Project

Og i **Assert** bekræfter vi at det var kode **200** der kom ud af testen.

```
[Fact]
public async void GetAllAsync_ShouldReturnStatusCode200_WhenHeroesExists()
{
    // Arrange
    List<HeroResponse> heroes = new();
    heroes.Add(new HeroResponse()
    {
        Id = 1,
        HeroName = "man",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    });
    heroes.Add(new HeroResponse()
    {
        Id = 2,
        HeroName = "Iron Man",
        RealName = "Tony Stark",
        Place = "Malibu",
        DebutYear = 1963
    });

    _heroServiceMock
        .Setup(x => x.GetAllAsync())
        .ReturnsAsync(heroes);

    // Act
    var result = (IStatusCodeActionResult)await _heroController.GetAllAsync();

    // Assert
    Assert.Equal(200, result.StatusCode);
}
```

GetAllAsync_ShouldReturnStatusCode204_WhenNoHeroesExist()

Testen der koncentrerer sig om situationen hvor der ikke findes helte i databasen, der hvor controlleren returnerer **NoContent()**, skal ganske enkelt sættes op uden at tilføje helte til **HeroResponse** listen, samt **Assert** skal bekræfte det er statuskode **204** der kom ud af testen.

```
[Fact]
public async void GetAllAsync_ShouldReturnStatusCode204_WhenNoHeroesExist()
{
    // Arrange
    List<HeroResponse> heroes = new();

    _heroServiceMock
        .Setup(x => x.GetAllAsync())
        .ReturnsAsync(heroes);

    // Act
    var result = (IStatusCodeActionResult)await _heroController.GetAllAsync();

    // Assert
    Assert.Equal(204, result.StatusCode);
}
```

GetAllAsync_ShouldReturnStatusCode500_WhenExceptionIsRaised()

Vi har sat vores service op så den kan kaste en **ArgumentNullException** i tilfældet hvor repository serverer et null objekt. Det scenarie skal vores controller kunne håndtere og derfor tester vi om vi får statuskode **500** hvis service kaster en **exception**.

Da vi egentlig ikke er interesseret i den specifikke exceptiontype her, så tester vi på en generisk Exception, frem for den specifikke **ArgumentNullException**. Vi ville være nødt til at opsætte 2 tests, hvis vi ville teste på begge Exceptions typer, men vi har kun 1 catch blok i controlleren, så det er lidt overflødig her.



Super Hero Project

```
[Fact]
public async void GetAllAsync_ShouldReturnStatusCode500_WhenExceptionIsRaised()
{
    // Arrange
    List<HeroResponse> heroes = new();

    _heroServiceMock
        .Setup(x => x.GetAllAsync())
        .ReturnsAsync(() => throw new Exception("This is an exception"));

    // Act
    var result = (IStatusCodeActionResult)await _heroController.GetAllAsync();

    // Assert
    Assert.Equal(500, result.StatusCode);
}
```

Swagger

Nu er vi klar til at teste at hele stacken kan modtage en request i Controlleren, og igennem Service laget bede Repository om at trække alle Heroes ud af Databasen, for derefter at lade Service konvertere Hero til HeroResponse som Controlleren kan sende tilbage til klienten.

Tryk **[CTRL]+ [F5]** og via swagger der åbnes i en browser afprøver du </api/Hero> endpointet, som gerne skulle vise de 2 helte der rent faktisk ligger i databasen. (*prøv gerne at ændre på værdierne direkte i databasen*)

The screenshot shows the Swagger UI interface for the `/api/Hero` endpoint. At the top, it says "Hero". Below that, there's a blue button for "GET" and the URL `/api/Hero`. Under "Parameters", it says "No parameters". There are "Execute" and "Clear" buttons. In the "Responses" section, under "Code", there's a "200" entry. The "Details" tab is selected, showing the "Response body" which contains the following JSON:

```
[{"id": 1, "heroName": "Superman", "realName": "Clark Kent", "place": "Metropolis", "debutYear": 1938}, {"id": 2, "heroName": "Iron Man", "realName": "Tony Stark", "place": "Malibu", "debutYear": 1963}]
```



Super Hero Project

Create Hero

For at oprette en Hero i vores system, starter vi igen på databaseniveau, og sætter **Repository** op så den kan indsætte en **Hero Entitet**.

Interfacet **IHeroRepository** skal opdateres, så vi kender signaturen på den metode der håndterer indsættelsen. Vi skal beslutte hvad der sendes ind i metoden, samt hvad der kommer ud af den igen.

Umiddelbart burde det være simpelt, vi sender et udfyldt **Hero** entitet objekt til metoden, og så indsættes **Hero** i databasen og derefter returneres **Hero** entiteten inklusiv den nye Id databasen har tildelt **Hero**... men hvad hvis vi har en **unique** værdi på entiteten, så kan databasen forsøge at indsætte noget, men ikke få lov til det... hvad skal der så returneres?

Det vil faktisk løses helt af sig selv, da Entityframework vil kaste en exception i det tilfælde, og det kan vores controller håndtere hvis vi tilføjer try/catch blokke.

Så **CreateAsync()** skal sådan set blot modtage en **Hero**, og returnere en **Hero**.

IHeroRepository

CreateAsync()

Tilføj **CreateAsync()** signaturen til dit **IHeroRepository** interface.

```
public interface IHeroRepository
{
    Task<List<Hero>> GetAllAsync();
    Task<Hero> CreateAsync(Hero newHero);
}
```

HeroRepository

CreateAsync()

Implementationen af metoden i **HeroRepository**.

```
public async Task<Hero> CreateAsync(Hero newHero)
{
    _context.Hero.Add(newHero);
    await _context.SaveChangesAsync();
    return newHero;
}
```

Nu har vi tilføjet en metode til vores system, og skal derfor opsætte de nødvendige UnitTests på den metode.

HeroRepositoryTests

CreateAsync_ShouldAddNewIdToHero_WhenSavingToDatabase()

Den første af de 2 tests vi er interesseret i her, er når oprettelsen er en success.

Vi bekræfter at det ikke er en **null** værdi der kommer ud af metoden.

Samt at datatypen er en **Hero** entitet.



Super Hero Project

Vi forventer det bliver tallet **1** helten får som **Id** af databasen, da databasen burde være helt tom.

```
[Fact]
public async void CreateAsync_ShouldAddNewIdToHero_WhenSavingToDatabase()
{
    // Arrange
    await _context.Database.EnsureDeletedAsync();

    int expectedNewId = 1;

    Hero hero = new()
    {
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    };

    // Act
    var result = await _heroRepository.CreateAsync(hero);

    // Assert
    Assert.NotNull(result);
    Assert.IsType<Hero>(result);
    Assert.Equal(expectedNewId, result?.Id);
}
```

CreateAsync_ShouldFailToAddNewHero_WhenHeroIdAlreadyExists()

Testen der sikrer der opstår fejl når en Id forsøges indsæt, hvis Id allerede findes, er meget lig den forrige test.

Forskellen er at vi indsætter helten i **Arrange**, og så sætter **CreateAsync** op så den kaldes under **Assert**. Da det er samme **Hero** objekt der indsættes i **CreateAsync** de gange metoden kaldes, så vil første kald sikre **Hero** får en **Id**, og den **Id** vil være tilstede anden gang **Hero** forsøges indsæt, og derfor burde det fejle.

Det er vigtigt at teksten i **Assert.Contains** er skrevet korrekt, ellers fejler testen... så **dobbelttjek!**

```
[Fact]
public async void CreateAsync_ShouldFailToAddNewHero_WhenHeroIdAlreadyExists()
{
    // Arrange
    await _context.Database.EnsureDeletedAsync();

    Hero hero = new()
    {
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    };

    // save hero to DB once, giving it an initial Id (1)
    await _heroRepository.CreateAsync(hero);

    // Act
    // because we expect an exception, this is a setup, the actual call happens in test
    async Task action() => await _heroRepository.CreateAsync(hero);

    // Assert
    var ex = await Assert.ThrowsAsync<ArgumentException>(action);
    Assert.Contains("An item with the same key has already been added", ex.Message);
}
```

HeroRequest

Der er også et interface på vores **HeroService**, som skal opdateres, og her skal vi igen overveje hvilke data vi sender ind i metoden og hvad vi forventer at få tilbage.

Da det er en opret handling, forventer vi at have modtaget en **Hero** fra controlleren, men der er spørgsmålet hvordan har controlleren modtaget data?

Vi har besluttet at entiteter ikke er tilgængelige for Controllerne, så vi er nødt til at opfinde et DTO som kan



Super Hero Project

controlleren kan arbejde med. Lige som `HeroResponse`, er den `Hero` der sendes ud, så opretter vi en `HeroRequest`, som er den `Hero` vi kan modtage i Controller.

Opret en klasse-fil kaldet `HeroRequest.cs` i `DTOs` mappen i `SuperHeroAPI` projektet. Og beskriv de egenskaber en `HeroRequest` objekt skal opfylde.

`HeroName`, `RealName` og `DebutYear` er påkrævede, `Place` er optional derfor ingen `[Required]` ved `Place`.

Vi har sat databasen op til at `HeroName`, `RealName` og `Place` maks må være på `32 tegn`, så det giver mening at sikre der ikke kommer flere tegn som skal klippes af.

`DebutYear` defaulter til tallet `0`, men vi er også interesseret i at sikre der ikke kommer for store værdier ind, da databasen feltet kun kan håndtere værdier mellem `-32.768` og `32.767`. Vi vælger her at sige `DebutYear` skal være mellem `0` og `2100`.

```
namespace SuperHeroAPI.DTOs
{
    public class HeroRequest
    {
        [Required]
        [StringLength(32, ErrorMessage = "HeroName cannot be longer than 32 chars")]
        public string HeroName { get; set; } = string.Empty;

        [Required]
        [StringLength(32, ErrorMessage = "RealName cannot be longer than 32 chars")]
        public string RealName { get; set; } = string.Empty;

        [Required]
        [Range(0, 2100, ErrorMessage = "DebutYear must be between 0 and 2100")]
        public short DebutYear { get; set; }

        // this is nullable on purpose, to demonstrate nullables
        [StringLength(32, ErrorMessage = "Place cannot be longer than 32 chars")]
        public string? Place { get; set; }
    }
}
```

HeroService

Mapping fra `HeroRequest` til `Hero`, og mapping fra `Hero` til `HeroResponse` vil være noget vi kommer til at gøre ofte, så for at minimere mulighederne for introducere fejl, giver det mening at oprette metoder til de handlinger.

MapHeroToHeroResponse()

Konceptet er rimelig simpelt, tag imod et `Hero` objekt, og returner et `HeroResponse` objekt med de værdier vi ønsker at gøre tilgængelige. Her kunne det være et `User` objekt, hvor f.eks. `Password` ikke sendes med ud.

```
private HeroResponse MapHeroToHeroResponse(Hero hero)
{
    return new HeroResponse
    {
        Id = hero.Id,
        HeroName = hero.HeroName,
        RealName = hero.RealName,
        Place = hero.Place,
        DebutYear = hero.DebutYear
    };
}
```



Super Hero Project

MapHeroRequestToHero()

For at oversætte en `HeroRequest` til et `Hero` objekt, skal vi håndtere at `Place` værdien KAN være `null`, da den er optional i `HeroRequesten`. Så ved Place benytter vi en `null-coalescing operator ??` til at sikre `Hero` får en værdi.

```
private Hero MapHeroRequestToHero(HeroRequest heroRequest)
{
    return new Hero
    {
        HeroName = heroRequest.HeroName,
        RealName = heroRequest.RealName,
        DebutYear = heroRequest.DebutYear,
        Place = heroRequest.Place ?? string.Empty
    };
}
```

GetAllAsync()

Nu hvor vi har en metode til at oversætte `Hero` til `HeroResponse`, så kan vi opdatere `GetAll` metoden, så den også benytter `MapHeroToHeroResponse` metoden.

```
public async Task<List<HeroResponse>> GetAllAsync()
{
    List<Hero> heroes = await _heroRepository.GetAllAsync();

    // if heroes are null at this point, something has gone wrong, throw an exception
    if (heroes == null)
    {
        throw new ArgumentNullException();
    }

    return heroes.Select(hero => MapHeroToHeroResponse(hero)).ToList();
}
```

IHeroService

CreateAsync()

Vi skal som altid bruge lidt tid på at tænke over hvad vi forventer service `Create` skal modtage og returnere. Det er givet at den skal modtage en `HeroRequest`, og det forventes at der kommer en `HeroResponse` retur. Men er der scenarier hvor vi mangler data?

Vi ved at vi på nuværende tidspunkt benytter `IHeroRepository` til at arbejde på databasen, og lige nu får vi enten en `Hero` tilbage, eller `Repository` kaster en exception... dvs vi har altid data, ellers er der sket en fejl.

Men, for at fremtidssikre koden en smule, vil det være nødvendigt at tjekke på om der kommer `null` fra `Repository`, og her kan vi vælge om vi vil `returnere null` eller `kaste en exception`. I dette eksempel viser jeg hvordan vi kaster en exception, så vi kan beskrive `CreateAsync()` som følger, uden nullables:

```
public interface IHeroService
{
    Task<List<HeroResponse>> GetAllAsync();
    Task<HeroResponse> CreateAsync(HeroRequest newHero);
}
```

HeroService

CreateAsync()

Nu hvor vi har skrevet mapping metoderne i vores `HeroService`, så bliver det ret simpelt at oversætte `HeroRequest` til `Hero`, og sende den til `HeroRepository`. Samt retur konverteringen bliver også meget lige til.



Super Hero Project

Hvis der af en eller anden grund skulle komme et `null` op af `Repository`, så sørger vi for at kaste en `ArgumentNullException`.

```
public async Task<HeroResponse> CreateAsync(HeroRequest newHero)
{
    var hero = await _heroRepository.CreateAsync(MapHeroRequestToHero(newHero));

    if (hero == null)
    {
        throw new ArgumentNullException();
    }

    return MapHeroToHeroResponse(hero);
}
```

HeroServiceTests

CreateAsync_ShouldReturnHeroResponse_WhenCreateIsSuccess()

Først opsætter vi de `HeroRequest` og `Hero` objekter vi vil benytte i testen. Derefter sætter vi `HeroRepositoryMock` op så den serverer `Hero` objektet, når `HeroRepository.CreateAsync()` rammes med et hvilket som helst `Hero` objekt

Vi tester derefter på at alle de værdier vi har sat, rent faktisk også er til stede når `Hero` er indsats, deriblandt den forventede `Id`.

Det er vigtigt at forstå at der IKKE bliver indsats noget i nogen database i denne test, ej heller i en `InMemoryDatabase`. Vi bestemmer præcis hvilke data der er i spil, så det er vigtigt at sikre data er korrekte... vi kunne let sætte `Hero.Id` til `3`, men teste efter om den er `1`, hvilket selvfølgelig aldrig vil være korrekt.

```
[Fact]
public async void CreateAsync_ShouldReturnHeroResponse_WhenCreateIsSuccess()
{
    // Arrange
    HeroRequest newHero = new()
    {
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    };
    int heroId = 1;
    Hero hero = new()
    {
        Id = heroId,
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    };

    _heroRepositoryMock
        .Setup(x => x.CreateAsync(It.IsAny<Hero>()))
        .ReturnsAsync(hero);

    // Act
    var result = await _heroService.CreateAsync(newHero);

    // Assert
    Assert.NotNull(result);
    Assert.IsType<HeroResponse>(result);
    Assert.Equal(hero.Id, result.Id);
    Assert.Equal(hero.HeroName, result.HeroName);
    Assert.Equal(hero.RealName, result.RealName);
    Assert.Equal(hero.Place, result.Place);
    Assert.Equal(hero.DebutYear, result.DebutYear);
}
```

CreateAsync_ShouldThrowNullException_WhenRepositoryReturnsNull()

I denne test vil vi fokusere på hvad der sker hvis den værdi der kommer fra `Repository` er `null`. Det sættes op i `ReturnsAsync()` lidt i stil med den måde vi sætter den op til at kaste exceptions.



Super Hero Project

I dette tilfælde skal vi tjekke at en specifik exception kastes, så der bruger vi `Assert.ThrowsAsync<>` til at bekræfte det er en `ArgumentNullException` vi får ud af testen.

```
[Fact]
public async void CreateAsync_ShouldThrowNullException_WhenRepositoryReturnsNull()
{
    // Arrange
    HeroRequest newHero = new()
    {
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    };

    _heroRepositoryMock
        .Setup(x => x.CreateAsync(It.IsAny<Hero>()))
        .ReturnsAsync(() => null);

    // Act
    async Task action() => await _heroService.CreateAsync(newHero);

    // Assert
    var ex = await Assert.ThrowsAsync<ArgumentNullException>(action);
    Assert.Contains("Value cannot be null", ex.Message);
}
```

HeroController

CreateAsync()

Når vi vil ramme `HeroController` og indsætte en ny `Hero`, så skal vi sende data via en `HttpPost` metode. Det er standarden for en `RESTfull` API at køre `POST` ind som opret.

Derudover så vil `POST` have selve data `payload` med i dens `Body`, så det ikke står som læsbar tekst i adressen, som ved en `GET` request.

Ellers er der to mulige svar fra `CreateAsync`.

Den ene er `Ok()` hvor `heroResponse` vil blive `JSON.Stringified` og sendt sammen med en `StatusCode 200`.

Den anden svarmulighed er `Problem()` hvor der er opstået en fejl i systemet og vi sender en `StatusCode 500` return til klienten.

```
[HttpPost]
public async Task<IActionResult> CreateAsync([FromBody] HeroRequest newHero)
{
    try
    {
        HeroResponse heroResponse = await _heroService.CreateAsync(newHero);

        return Ok(heroResponse);
    }
    catch (Exception ex)
    {
        return Problem(ex.Message);
    }
}
```

Teknisk set kan der også opstå `StatusCodes 400, badRequest`, men det sker i .NET frameworket, det er uden for scope af vores testscenarier.

HeroControllerTests

Vi skal huske at teste de metoder vi tilføjer, så de to scenarier med `Ok()` og `Problem()` testes i `HeroControllerTests` klassen.



Super Hero Project

CreateAsync_ShouldReturnStatusCode200_WhenHeroIsSuccessfullyCreated()

Første test skal håndtere både en **HeroRequest** og en **HeroResponse**, selv om det egentlig kun er selve **StatusCode** værdien der tjekkes på.

```
[Fact]
public async void CreateAsync_ShouldReturnStatusCode200_WhenHeroIsSuccessfullyCreated()
{
    // Arrange
    HeroRequest newHero = new()
    {
        HeroName = "Superman",
        RealName = "Clark Kent",
        DebutYear = 1938,
        Place = "Metropolis"
    };

    int heroId = 1;

    HeroResponse heroResponse = new()
    {
        Id = heroId,
        HeroName = "Superman",
        RealName = "Clark Kent",
        DebutYear = 1938,
        Place = "Metropolis"
    };

    _heroServiceMock
        .Setup(x => x.CreateAsync(It.IsAny<HeroRequest>()))
        .ReturnsAsync(heroResponse);

    // Act
    var result = (IStatusCodeActionResult)await _heroController.CreateAsync(newHero);

    // Assert
    Assert.Equal(200, result.StatusCode);
}
```

CreateAsync_ShouldReturnStatusCode500_WhenExceptionIsRasied()

Da vores **Controller** har en **try/catch** blok, kan vi lade koden køre uden at håndtere exceptions som vi har i **Service** og **Repository** lagene, vi skal blot teste at tallet der kommer er **500**.

Det er ligemeget hvilken type exception der er opstået i testen, vi ved at når der opstår en exception, rammer vi **catch** blokken som så returnerer **Problem()** der er en StatusCode **500**.

```
[Fact]
public async void CreateAsync_ShouldReturnStatusCode500_WhenExceptionIsRasied()
{
    // Arrange
    HeroRequest newHero = new()
    {
        HeroName = "Superman",
        RealName = "Clark Kent",
        DebutYear = 1938,
        Place = "Metropolis"
    };

    _heroServiceMock
        .Setup(x => x.CreateAsync(It.IsAny<HeroRequest>()))
        .ReturnsAsync(() => throw new Exception("This is an exception"));

    // Act
    var result = (IStatusCodeActionResult)await _heroController.CreateAsync(newHero);

    // Assert
    Assert.Equal(500, result.StatusCode);
}
```

Bekræft at alle tests går succesfuldt igennem!



Super Hero Project

Swagger

Kør projektet [CTRL] + [F5] og se Swagger selv har opdaget der er en metode mere i vores Controller, og at metoden er af typen [HttpPost](#). Prøv at køre metoden, med en ny Hero:

POST /api/Hero

Parameters

No parameters

Request body

application/json

```
{ "heroName": "Spider-Man", "realName": "Peter Parker", "debutYear": 1962, "place": "New York City" }
```

Curl

```
curl -X 'POST' \
  'https://localhost:7150/api/Hero' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{' \
    "heroName": "Spider-Man", \
    "realName": "Peter Parker", \
    "debutYear": 1962, \
    "place": "New York City" \
  }'
```

Request URL

https://localhost:7150/api/Hero

Server response

Code Details

200 Response body

```
{ "id": 3, "heroName": "Spider-Man", "realName": "Peter Parker", "place": "New York City", "debutYear": 1962 }
```

Copy Download

Eksperimenter med forskellige værdier, og prøv at fjerne nogle af fejterne.

Er der eventuelt noget med DebutYear? Prøv det af... Måske der skal rettes lidt i en DTO...

Gem og commit til Github.

Find en Hero baseret på Id

Det er smart at have en metode til at finde enkelte entiteter, baseret på deres [Id](#). Det er ikke helt det samme som at hente en specifik entitet, da dette er en "søgning" efter en bestemt entitet, som måske ikke findes.



Super Hero Project

Tanken er at når vi vil opdatere, slette eller hente en specifik Hero, så spørger vi databasen om at finde den pågældende **Hero**, men vi kan lige så godt få **null** ud, som en **Hero**, uden at det er en fejl.
Så alle vores lag skal kunne håndtere **null/Hero** som valide værdier.

IHeroRepository

FindByIdAsync()

Skal kunne modtage en **integer**, og returnerer en **nullable Hero**.

```
public interface IHeroRepository
{
    Task<List<Hero>> GetAllAsync();
    Task<Hero> CreateAsync(Hero newHero);
    Task<Hero?> FindByIdAsync(int heroId);
```

HeroRepository

FindByIdAsync()

EntityFramework har en indbygget metode til at finde en entitet baseret på dens **primærnøgle**. Metoden modtager værdien, og returnerer enten **null** hvis entiteten ikke findes, eller selve **entiteten** hvis den eksisterer.

```
public async Task<Hero?> FindByIdAsync(int heroId)
{
    return await _context.Hero.FindAsync(heroId);
```

HeroRepositoryTests

FindByIdAsync_ShouldReturnHero_WhenHeroExists()

Når vi tester den nye **FindByIdAsync**, så er det helt lige ud af landevejen, at indsætte en **Hero** i vores **InMemoryDatabase** og derefter hente den ud igen.

Læg mærke til der står **result?** i den sidste **Equals** tjek, det er fordi result jo **KAN** være **null**, så skal vi sikre den findes, før vi tilgår dens egenskaber.

```
[Fact]
public async void FindByIdAsync_ShouldReturnHero_WhenHeroExists()
{
    // Arrange
    await _context.Database.EnsureDeletedAsync();

    int heroId = 1;

    _context.Hero.Add(new()
    {
        Id = heroId,
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    });

    await _context.SaveChangesAsync();

    // Act
    var result = await _heroRepository.FindByIdAsync(heroId);

    // Assert
    Assert.NotNull(result);
    Assert.IsType<Hero>(result);
    Assert.Equal(heroId, result?.Id);
}
```



Super Hero Project

FindByAsync_ShouldReturnNull_WhenHeroDoesNotExist()

Når der skal testes på scenariet hvor vi forsøger at hente en helt der ikke findes, så er det blot at lade være med at indsætte helte i databasen. Et alternativ kunne være at forsøge at hente en **Hero** med en meget stor **Id**. f.eks. `int.MaxValue`, som er **2.147.483.647**.

```
[Fact]
public async void FindByIdAsync_ShouldReturnNull_WhenHeroDoesNotExist()
{
    // Arrange
    await _context.Database.EnsureDeletedAsync();

    // Act
    var result = await _heroRepository.FindByIdAsync(1);

    // Assert
    Assert.Null(result);
}
```

IHeroService

FindByAsync()

Da vi ved **FindByAsync** fra Repository kan returnere enten en **Hero** eller **null**, og at **null** er en valid værdi, så skal vi sætte **HeroService** op så den også kan håndtere den mulige **null** retur type. Her er det en **HeroResponse** der kan være **null**.

```
public interface IHeroService
{
    Task<List<HeroResponse>> GetAllAsync();
    Task<HeroResponse> CreateAsync(HeroRequest newHero);
    Task<HeroResponse?> FindByIdAsync(int heroId);
}
```

HeroService

FindByAsync()

Selve implementationen af **FindByAsync** i Service er super nem nu hvor vi har en mapping metode til at håndtere oversættelsen af **Hero** til **HeroResponse**.

Vi skal bare lige huske at tjekke om vi har en **Hero**, før vi forsøger at mappe til **HeroResponse**.

```
public async Task<HeroResponse?> FindByIdAsync(int heroId)
{
    var hero = await _heroRepository.FindByIdAsync(heroId);

    // hero CAN be null here, if hero doesn't exist, this is NOT an error.
    if (hero != null)
    {
        return MapHeroToHeroResponse(hero);
    }

    return null;
}
```

HeroServiceTests

FindByAsync_ShouldReturnHeroResponse_WhenHeroExists()

Når vi tester **HeroService**, så kan vi teste om den kan oversætte fra et **Hero** objekt til et **HeroResponse** objekt. Det kunne jo være man ved en fejl havde fået byttet om på **HeroName** og **RealName** i mapping funktionen, så



Super Hero Project

derfor de ekstra `Assert.Equals` i testen.

```
[Fact]
public async void FindByIdAsync_ShouldReturnHeroResponse_WhenHeroExists()
{
    // Arrange
    int heroId = 1;

    Hero hero = new()
    {
        Id = heroId,
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    };

    _heroRepositoryMock
        .Setup(x => x.FindByIdAsync(It.IsAny<int>()))
        .ReturnsAsync(hero);

    // Act
    var result = await _heroService.FindByIdAsync(heroId);

    // Assert
    Assert.NotNull(result);
    Assert.IsType<HeroResponse>(result);
    Assert.Equal(hero.Id, result?.Id);
    Assert.Equal(hero.HeroName, result?.HeroName);
    Assert.Equal(hero.RealName, result?.RealName);
    Assert.Equal(hero.DebutYear, result?.DebutYear);
    Assert.Equal(hero.Place, result?.Place);
}
```

FindByIdAsync_ShouldReturnNull_WhenHeroDoesNotExists()

Vi skal også huske at teste scenariet hvor `Repository` returnerer værdien `null`.

```
[Fact]
public async void FindByIdAsync_ShouldReturnNull_WhenHeroDoesNotExists()
{
    // Arrange
    int heroId = 1;

    _heroRepositoryMock
        .Setup(x => x.FindByIdAsync(It.IsAny<int>()))
        .ReturnsAsync(() => null);

    // Act
    var result = await _heroService.FindByIdAsync(heroId);

    // Assert
    Assert.Null(result);
}
```

HeroController

FindByIDAsync()

For at finde en `Hero` baseret på dens `Id`, skal vi modtage en `Id` i systemet. Her er metoden sat op til at modtage et tal fra route data. Stien til metoden endpoint er `/api/hero/1` hvor `1` er den parameter vi har defineret i `[Route("{heroId}")]` attributten.

Der er 3 scenarier vi skal kunne håndtere i vores `Controller`.

Først returnerer vi `NotFound() 404`, hvis Service returnerer `null`.

Derefter `Ok() 200` med en `HeroResponse` når vi har modtaget en fra Service.

Og til sidst håndterer vi alle de exceptions der måtte opstå, med vores catch blok, som returnerer `Problem()`



Super Hero Project

500.

```
[HttpGet]
[Route("{heroId}")]
public async Task<IActionResult> FindByIdAsync([FromRoute] int heroId)
{
    try
    {
        var heroResponse = await _heroService.FindByIdAsync(heroId);

        if (heroResponse == null)
        {
            return NotFound();
        }

        return Ok(heroResponse);
    }
    catch (Exception ex)
    {
        return Problem(ex.Message);
    }
}
```

HeroControllerTests

FindByIdAsync_ShouldReturnStatusCode200_WhenHeroExists()

Testen er som de andre **HeroController** tests, fokuseret på den **StatusCode** der forventes at blive sendt, når vi finder den **Hero** vi søger.

```
[Fact]
public async void FindByIdAsync_ShouldReturnStatusCode200_WhenHeroExists()
{
    // Arrange
    int heroId = 1;

    HeroResponse hero = new()
    {
        Id = heroId,
        HeroName = "man",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    };

    _heroServiceMock
        .Setup(x => x.FindByIdAsync(It.IsAny<int>()))
        .ReturnsAsync(hero);

    // Act
    var result = (IStatusCodeActionResult)await _heroController.FindByIdAsync(heroId);

    // Assert
    Assert.Equal(200, result.StatusCode);
}
```

FindByIdAsync_ShouldReturnStatusCode404_WhenHeroDoesNotExist()

Testen på **null** fra Servicelaget foreventes at få en StatusCode **404**, **NotFound()**.

```
[Fact]
public async void FindByIdAsync_ShouldReturnStatusCode404_WhenHeroDoesNotExist()
{
    // Arrange
    int heroId = 1;

    _heroServiceMock
        .Setup(x => x.FindByIdAsync(It.IsAny<int>()))
        .ReturnsAsync(() => null);

    // Act
    var result = (IStatusCodeActionResult)await _heroController.FindByIdAsync(heroId);

    // Assert
    Assert.Equal(404, result.StatusCode);
}
```



Super Hero Project

FindByAsync_ShouldReturnStatusCode500_WhenExceptionIsRaised()

Og den sidste test hvor vi tester Exceptions, er som vi har set et par gange nu. Det er stadig irrelevant hvilken type Exception der bliver kastet.

```
[Fact]
public async void FindByIdAsync_ShouldReturnStatusCode500_WhenExceptionIsRaised()
{
    // Arrange
    int heroId = 1;

    _heroServiceMock
        .Setup(x => x.FindByIdAsync(It.IsAny<int>()))
        .ReturnsAsync(() => throw new Exception("This is an exception"));

    // Act
    var result = (IStatusCodeActionResult)await _heroController.FindByIdAsync(heroId);

    // Assert
    Assert.Equal(500, result.StatusCode);
}
```

Swagger

Test funktionaliteten via Swagger, husk at køre serveren [CTRL] + [F5].

Gem og commit til Github.

Opdater en Hero

For at ændre værdierne på en Hero, skal vi kunne sende et Id og en HeroRequest med de nye værdier til vores system. Den opdaterede HeroRequest skal sendes fra Controller og ned igennem lagene, for tilsidst at opdatere værdierne i databasen.

Dog skal vi også kunne håndtere at en Id måske ikke findes i databasen, eller at f.eks. Place i Hero er optional. Så der er lidt mere kode i en rediger-stack.

IHeroRepository

UpdateByIdAsync()

Da vi forsøger at opdatere en **Hero** baseret på den **Id**, så giver det god mening at navngive metoden derefter.

Og da vi ved **Id** måske ikke findes, skal vi sørge for at returntypen er **nullable**.

Metoden skal modtage både en **Id** og selve det opdaterede data i form af et **Hero** objekt.

```
public interface IHeroRepository
{
    Task<List<Hero>> GetAllAsync();
    Task<Hero> CreateAsync(Hero newHero);
    Task<Hero?> FindByIdAsync(int heroId);
    Task<Hero?> UpdateByIdAsync(int heroId, Hero updateHero);
}
```

HeroRepository

UpdateByIdAsync()

Først forsøger vi at hente **Hero** ved at benytte vores **FindByID**. Hvis der kommer noget ud af det kald, så kan vi opdatere værdierne efter behov, og tilsidst huske at gemme dem i databasen.



Super Hero Project

Afslutningsvis returneres det **Hero** objekt som måske er **null**, eller er **opdateret**.

```
public async Task<Hero?> UpdateByIdAsync(int heroId, Hero updateHero)
{
    var hero = await FindByIdAsync(heroId);

    if (hero != null)
    {
        // set required values
        hero.HeroName = updateHero.HeroName;
        hero.RealName = updateHero.RealName;
        hero.DebutYear = updateHero.DebutYear;

        // handle optional values
        if (updateHero.Place != string.Empty)
        {
            hero.Place = updateHero.Place;
        }

        await _context.SaveChangesAsync();
    }

    return hero; // can be null here!
}
```

HeroRepositoryTests

UpdateByIdAsync_ShouldChangeValuesOnHero_WhenHeroExists()

Der er en del opsætning vi skal håndtere i denne test, da vi først skal indsætte en Hero i databasen, og derefter opdatere den. Og vi er nødt til at tjekke at resultatet har de korrekte værdier.

```
[Fact]
public async void UpdateByIdAsync_ShouldChangeValuesOnHero_WhenHeroExists()
{
    // Arrange
    await _context.Database.EnsureDeletedAsync();
    int heroId = 1;
    Hero newHero = new()
    {
        Id = heroId,
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    };
    _context.Hero.Add(newHero);
    await _context.SaveChangesAsync();
    Hero updateHero = new()
    {
        Id = heroId,
        HeroName = "new Superman",
        RealName = "new Clark Kent",
        Place = "new Metropolis",
        DebutYear = 1999
    };
    // Act
    var result = await _heroRepository.UpdateByIdAsync(heroId, updateHero);

    // Assert
    Assert.NotNull(result);
    Assert.IsType<Hero>(result);
    Assert.Equal(heroId, result?.Id);
    Assert.Equal(updateHero.HeroName, result?.HeroName);
    Assert.Equal(updateHero.RealName, result?.RealName);
    Assert.Equal(updateHero.Place, result?.Place);
    Assert.Equal(updateHero.DebutYear, result?.DebutYear);
}
```



Super Hero Project

UpdateByIdAsync_ShouldReturnNull_WhenHeroDoesNotExist()

For at teste om vi kan få `UpdateById` til at returnere `null`, så undlader vi blot at indsætte en `Hero` i databasen, så burde der komme `null` fra den `FindByAsync` der kaldes inde i Repository metoden.

```
[Fact]
public async void UpdateByIdAsync_ShouldReturnNull_WhenHeroDoesNotExist()
{
    // Arrange
    await _context.Database.EnsureDeletedAsync();

    int heroId = 1;
    Hero updateHero = new()
    {
        Id = heroId,
        HeroName = "new Superman",
        RealName = "new Clark Kent",
        Place = "new Metropolis",
        DebutYear = 1999
    };

    // Act
    var result = await _heroRepository.UpdateByIdAsync(heroId, updateHero);

    // Assert
    Assert.Null(result);
}
```

IHeroService

UpdateByIdAsync()

Servicen skal have en `UpdateByIdAsync` metode, som skal kunne returnere et `HeroResponse` objekt hvis opdateringen er en success. Hvis `Hero` ikke findes, skal metoden returnere `null`.

Vi sender den `Id` med, som repræsenterer den `Hero` vi ønsker at opdatere, samt et `HeroRequest` objekt med de nye værdier vi ønsker at gemme i databasen.

```
public interface IHeroService
{
    Task<List<HeroResponse>> GetAllAsync();
    Task<HeroResponse> CreateAsync(HeroRequest newHero);
    Task<HeroResponse?> FindByIdAsync(int heroId);
    Task<HeroResponse?> UpdateByIdAsync(int heroId, HeroRequest updateHero);
}
```

HeroService

UpdateByIdAsync()

Implementationen af `UpdateByIdAsync` er enkel, nu hvor vi har mapping metoderne klar.

```
public async Task<HeroResponse?> UpdateByIdAsync(int heroId, HeroRequest updateHero)
{
    var hero = await _heroRepository.UpdateByIdAsync(heroId, MapHeroRequestToHero(updateHero));

    if (hero != null)
    {
        return MapHeroToHeroResponse(hero);
    }

    return null;
}
```

HeroServiceTests

UpdateByIdAsync_ShouldReturnHeroResponse_WhenUpdateIsSuccess()

Tests på opdateringer bliver altid lidt lange, da vi har mange værdier der skal håndteres. Men det er meget lige ud ad landevejen, hvis man går metodisk til værks. Husk at vi ikke tester selve databasen her, vi tester på om



Super Hero Project

Service kan returnere de korrekte data.

```
[Fact]
public async void UpdateByIdAsync_ShouldReturnHeroResponse_WhenUpdateIsSuccess()
{
    // NOTICE, we do not test if anything actually changed on the DB,
    // we only test that the returned values match the submitted values
    // Arrange
    HeroRequest heroRequest = new()
    {
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    };
    int heroId = 1;
    Hero hero = new()
    {
        Id = heroId,
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    };
    _heroRepositoryMock
        .Setup(x => x.UpdateByIdAsync(It.IsAny<int>(), It.IsAny<Hero>()))
        .ReturnsAsync(hero);
    // Act
    var result = await _heroService.UpdateByIdAsync(heroId, heroRequest);

    // Assert
    Assert.NotNull(result);
    Assert.IsType<HeroResponse>(result);
    Assert.Equal(heroId, result?.Id);
    Assert.Equal(heroRequest.HeroName, result?.HeroName);
    Assert.Equal(heroRequest.RealName, result?.RealName);
    Assert.Equal(heroRequest.Place, result?.Place);
    Assert.Equal(heroRequest.DebutYear, result?.DebutYear);
}
```

UpdateByIdAsync_ShouldReturnNull_WhenHeroDoesNotExists()

Når `UpdateById` skal håndtere at en `Hero` ikke findes, så er det et simpelt tjek på om result er `null` når det kommer ud af `Repository`.

```
[Fact]
public async void UpdateByIdAsync_ShouldReturnNull_WhenHeroDoesNotExists()
{
    // Arrange
    HeroRequest heroRequest = new()
    {
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    };
    int heroId = 1;

    _heroRepositoryMock
        .Setup(x => x.UpdateByIdAsync(It.IsAny<int>(), It.IsAny<Hero>()))
        .ReturnsAsync(() => null);

    // Act
    var result = await _heroService.UpdateByIdAsync(heroId, heroRequest);

    // Assert
    Assert.Null(result);
}
```

HeroController

UpdateByIdAsync()

For at sende en request om at opdatere værdier, så skal vi benytte `HttpPut` request metoden. Og vi sørger for at den sidste værdi i routen repræsenterer den `heroId` vi ønsker at opdatere.



Super Hero Project

De faktiske værdier bliver sendt i den medsendte **body**.

```
[HttpPut]
[Route("{heroId}")]
public async Task<IActionResult> UpdateByIdAsync([FromRoute] int heroId, [FromBody] HeroRequest updateHero)
{
    try
    {
        var heroResponse = await _heroService.UpdateByIdAsync(heroId, updateHero);

        if (heroResponse == null)
        {
            return NotFound();
        }

        return Ok(heroResponse);
    }
    catch (Exception ex)
    {
        return Problem(ex.Message);
    }
}
```

HeroControllerTests

UpdateByIdAsync_ShouldReturnStatusCode200_WhenHeroIsUpdated()

De tests vi skal udføre på `UpdateByIdAsync` er meget i stil med dem vi allerede har skrevet, så kopier og tilpas efter behov.

```
[Fact]
public async void UpdateByIdAsync_ShouldReturnStatusCode200_WhenHeroIsUpdated()
{
    // Arrange
    HeroRequest updateHero = new()
    {
        HeroName = "Superman",
        RealName = "Clark Kent",
        DebutYear = 1938,
        Place = "Metropolis"
    };

    int heroId = 1;

    HeroResponse heroResponse = new()
    {
        Id = heroId,
        HeroName = "Superman",
        RealName = "Clark Kent",
        DebutYear = 1938,
        Place = "Metropolis"
    };

    _heroServiceMock
        .Setup(x => x.UpdateByIdAsync(It.IsAny<int>(), It.IsAny<HeroRequest>()))
        .ReturnsAsync(heroResponse);

    // Act
    var result = (IStatusCodeActionResult)await _heroController.UpdateByIdAsync(heroId, updateHero);

    // Assert
    Assert.Equal(200, result.StatusCode);
}
```



Super Hero Project

UpdateByIdAsync_ShouldReturnStatusCode404_WhenHeroDoesNotExist()

Controlleren skal returnere 404 når der ikke findes en Hero med den ønskede Id. Dvs når der kommer null fra HeroService.

```
[Fact]
public async void UpdateByIdAsync_ShouldReturnStatusCode404_WhenHeroDoesNotExist()
{
    // Arrange
    HeroRequest updateHero = new()
    {
        HeroName = "Superman",
        RealName = "Clark Kent",
        DebutYear = 1938,
        Place = "Metropolis"
    };

    int heroId = 1;

    _heroServiceMock
        .Setup(x => x.UpdateByIdAsync(It.IsAny<int>(), It.IsAny<HeroRequest>()))
        .ReturnsAsync(() => null);

    // Act
    var result = (IStatusCodeActionResult)await _heroController.UpdateByIdAsync(heroId, updateHero);

    // Assert
    Assert.Equal(404, result.StatusCode);
}
```

UpdateByIdAsync_ShouldReturnStatusCode500_WhenExceptionIsRaised()

StatusCode 500 sker som ved de forrige tests, når en exception bliver kastet fra HeroService.

```
[Fact]
public async void UpdateByIdAsync_ShouldReturnStatusCode500_WhenExceptionIsRaised()
{
    // Arrange
    HeroRequest updateHero = new()
    {
        HeroName = "Superman",
        RealName = "Clark Kent",
        DebutYear = 1938,
        Place = "Metropolis"
    };

    int heroId = 1;

    _heroServiceMock
        .Setup(x => x.UpdateByIdAsync(It.IsAny<int>(), It.IsAny<HeroRequest>()))
        .ReturnsAsync(() => throw new Exception("This is an exception"));

    // Act
    var result = (IStatusCodeActionResult)await _heroController.UpdateByIdAsync(heroId, updateHero);

    // Assert
    Assert.Equal(500, result.StatusCode);
}
```

Swagger

Test i swagger, at du kan opdatere en [Hero](#).

Hvad sker der hvis man undlader at sende [Place](#) med i opdateringen? Burde der ske noget andet?

Gem alt og commit til Github.

Slet en Hero

Det sidste vi mangler, for at have en fuld [CRUD](#), er slet funktionaliteten.



Super Hero Project

Konceptet er at vi sender en `Id` til `SuperHeroAPI` og så forventer vi at den ønskede `Hero` slettes, og det slettede `HeroResponse` objekt kommer retur så vi se hvad vi har slettet. Dog skal vi også huske at håndtere et `Id` som ikke findes i databasen.

IHeroRepository

DeleteByIdAsync()

```
public interface IHeroRepository
{
    Task<List<Hero>> GetAllAsync();
    Task<Hero> CreateAsync(Hero newHero);
    Task<Hero?> FindByIdAsync(int heroId);
    Task<Hero?> UpdateByIdAsync(int heroId, Hero updateHero);
    Task<Hero?> DeleteByIdAsync(int heroId);
}
```

HeroRepository

DeleteByIdAsync()

Det er kun hvis der findes en `Hero` med den ønskede `Id` vi skal fjerne noget, ellers returneres `Hero`, som vil være `null`. Vi returnerer også `Hero` hvis den bliver slettet.

```
public async Task<Hero?> DeleteByIdAsync(int heroId)
{
    var hero = await FindByIdAsync(heroId);

    if (hero != null)
    {
        _context.Remove(hero);
        await _context.SaveChangesAsync();
    }

    return hero;
}
```

HeroRepositoryTests

DeleteByIdAsync_ShouldReturnDeletedHero_WhenHeroIsDeleted()

```
[Fact]
public async void DeleteByIdAsync_ShouldReturnDeletedHero_WhenHeroIsDeleted()
{
    // Arrange
    await _context.Database.EnsureDeletedAsync();

    int heroId = 1;
    Hero newHero = new()
    {
        Id = heroId,
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    };

    _context.Hero.Add(newHero);
    await _context.SaveChangesAsync();

    // Act
    var result = await _heroRepository.DeleteByIdAsync(heroId);

    // Assert
    Assert.NotNull(result);
    Assert.IsType<Hero>(result);
    Assert.Equal(heroId, result?.Id);
}
```



Super Hero Project

DeleteByIdAsync_ShouldReturnNull_WhenHeroDoesNotExist()

```
[Fact]
public async void DeleteByIdAsync_ShouldReturnNull_WhenHeroDoesNotExist()
{
    // Arrange
    await _context.Database.EnsureDeletedAsync();

    // Act
    var result = await _heroRepository.DeleteByIdAsync(1);

    // Assert
    Assert.Null(result);
}
```

Er der et testscenarie som rent faktisk tester at **Hero** er blevet **fjernet** fra databasen? Hvis ikke, hvordan skulle sådan en test se ud?

IHeroService

DeleteByIdAsync()

```
public interface IHeroService
{
    Task<List<HeroResponse>> GetAllAsync();
    Task<HeroResponse> CreateAsync(HeroRequest newHero);
    Task<HeroResponse?> FindByIdAsync(int heroId);
    Task<HeroResponse?> UpdateByIdAsync(int heroId, HeroRequest updateHero);
    Task<HeroResponse?> DeleteByIdAsync(int heroId);
}
```

HeroService

DeleteByIdAsync()

```
public async Task<HeroResponse?> DeleteByIdAsync(int heroId)
{
    var hero = await _heroRepository.DeleteByIdAsync(heroId);

    if (hero != null)
    {
        return MapHeroToHeroResponse(hero);
    }

    return null;
}
```



Super Hero Project

HeroServiceTests

DeleteByIdAsync_ShouldReturnHeroResponse_WhenDeleteIsSuccess()

```
[Fact]
public async void DeleteByIdAsync_ShouldReturnHeroResponse_WhenDeleteIsSuccess()
{
    // Arrange
    int heroId = 1;

    Hero hero = new()
    {
        Id = heroId,
        HeroName = "Superman",
        RealName = "Clark Kent",
        Place = "Metropolis",
        DebutYear = 1938
    };

    _heroRepositoryMock
        .Setup(x => x.DeleteByIdAsync(It.IsAny<int>()))
        .ReturnsAsync(hero);

    // Act
    var result = await _heroService.DeleteByIdAsync(heroId);

    // Assert
    Assert.NotNull(result);
    Assert.IsType<HeroResponse>(result);
    Assert.Equal(hero.Id, result?.Id);
}
```

DeleteByIdAsync_ShouldReturnNull_WhenHeroDoesNotExists()

```
[Fact]
public async void DeleteByIdAsync_ShouldReturnNull_WhenHeroDoesNotExists()
{
    // Arrange
    int heroId = 1;

    _heroRepositoryMock
        .Setup(x => x.DeleteByIdAsync(It.IsAny<int>()))
        .ReturnsAsync(() => null);

    // Act
    var result = await _heroService.DeleteByIdAsync(heroId);

    // Assert
    Assert.Null(result);
}
```

HeroController

DeleteByIdAsync()

```
[HttpDelete]
[Route("{heroId}")]
public async Task<IActionResult> DeleteByIdAsync([FromRoute] int heroId)
{
    try
    {
        var heroResponse = await _heroService.DeleteByIdAsync(heroId);

        if (heroResponse == null)
        {
            return NotFound();
        }

        return Ok(heroResponse);
    }
    catch (Exception ex)
    {
        return Problem(ex.Message);
    }
}
```



Super Hero Project

HeroControllerTests

DeleteByIdAsync_ShouldReturnStatusCode200_WhenHeroIsDeleted()

```
[Fact]
public async void DeleteByIdAsync_ShouldReturnStatusCode200_WhenHeroIsDeleted()
{
    // Arrange
    int heroId = 1;

    HeroResponse heroResponse = new()
    {
        Id = heroId,
        HeroName = "Superman",
        RealName = "Clark Kent",
        DebutYear = 1938,
        Place = "Metropolis"
    };

    _heroServiceMock
        .Setup(x => x.DeleteByIdAsync(It.IsAny<int>()))
        .ReturnsAsync(heroResponse);

    // Act
    var result = (IStatusCodeActionResult)await _heroController.DeleteByIdAsync(heroId);

    // Assert
    Assert.Equal(200, result.StatusCode);
}
```

DeleteByIdAsync_ShouldReturnStatusCode404_WhenHeroDoesNotExist()

```
[Fact]
public async void DeleteByIdAsync_ShouldReturnStatusCode404_WhenHeroDoesNotExist()
{
    // Arrange
    int heroId = 1;

    _heroServiceMock
        .Setup(x => x.DeleteByIdAsync(It.IsAny<int>()))
        .ReturnsAsync(() => null);

    // Act
    var result = (IStatusCodeActionResult)await _heroController.DeleteByIdAsync(heroId);

    // Assert
    Assert.Equal(404, result.StatusCode);
}
```

DeleteByIdAsync_ShouldReturnStatusCode500_WhenExceptionIsRaised()

```
[Fact]
public async void DeleteByIdAsync_ShouldReturnStatusCode500_WhenExceptionIsRaised()
{
    // Arrange
    int heroId = 1;

    _heroServiceMock
        .Setup(x => x.DeleteByIdAsync(It.IsAny<int>()))
        .ReturnsAsync(() => throw new Exception("This is an exception"));

    // Act
    var result = (IStatusCodeActionResult)await _heroController.DeleteByIdAsync(heroId);

    // Assert
    Assert.Equal(500, result.StatusCode);
}
```

Swagger

Husk at teste alt funksjonaliteten i [Swagger](#), samt sikre at alle tests stadig går igennem.

Gem alt og commit til Github.



Super Hero Project

Angular

På nuværende tidspunkt er vi klar til at introducere hjemmeside delen af projektet, hvilket vil være på [Angular](#) platformen.

Sørg for at have [NodeJs LTS](#) installeret ([version 18.15.0](#)), og derefter Angular CLI med følgende kommando

```
npm install -g @angular/cli@15.2.4
```

Det er muligt angular kommer med en (Unsupported) besked vedr Node versionen, det er ikke et problem.

```
Angular CLI: 15.2.4
Node: 18.14.2
Package Manager: npm 9.5.0
OS: win32 x64

Angular:
...
Package          Version
-----
@angular-devkit/architect    0.1502.4 (cli-only)
@angular-devkit/core         15.2.4 (cli-only)
@angular-devkit/schematics   15.2.4 (cli-only)
@schematics/angular          15.2.4 (cli-only)

C:\Users\Jack Baltzer\source\repos[]
```

For at oprette angular projektet, skal vi have åbnet et [CMD](#) vindue, navigere ind i Super Hero Project mappen. Dvs vi skal stå i den mappe hvor [.sln](#) filen og [SuperHeroAPI](#) og [SuperHeroTests](#) mapperne ligger.

```
Microsoft Windows [Version 10.0.19044.1645]
(c) Microsoft Corporation. Alle rettigheder forbeholdes.

C:\Users\jabb\source\repos\Tec\2022\Q4\super-hero-project-JackBaltzer\Super Hero Project>dir
Volume in drive C is OSDisk
Volume Serial Number is C4EC-7665

Directory of C:\Users\jabb\source\repos\Tec\2022\Q4\super-hero-project-JackBaltzer\Super Hero Project

27-10-2022 11:49    <DIR>        .
27-10-2022 11:49    <DIR>        ..
27-10-2022 11:49            98 .editorconfig
27-10-2022 11:49       1.878 Super Hero Project.sln
28-10-2022 10:05    <DIR>        SuperHeroAPI
27-10-2022 07:19    <DIR>        SuperHeroTests
                           2 File(s)     1.976 bytes
                           4 Dir(s)   27.542.695.936 bytes free
```

Når vi er i den korrekte mappe, køres kommandoen `ng new SuperHeroClient --skip-tests`:

```
C:\Users\jabb\source\repos\Tec\2022\Q4\super-hero-project-JackBaltzer\Super Hero Project>ng new SuperHeroClient --skip-tests
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? (Use arrow keys)
> CSS
SCSS  [ https://sass-lang.com/documentation/syntax#scss           ]
Sass   [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
Less   [ http://lesscss.org                                         ]
```

Vælg **Yes** ved routing, og **CSS** som stylesheet.

Derefter opretter Angular de nødvendige filer og mapper til vores client, samt henter og installerer de nødvendige [node_modules](#)... det kan godt tage lidt tid.



Super Hero Project

Når den er færdig, burde CMD vinduet se ud som her. Læg mærke til at Angular har opdaget at vi allerede er inde i et **git-repository**, så alt er som det skal være.

```
CREATE SuperHeroClient/src/environments/environment.prod.ts (51 bytes)
CREATE SuperHeroClient/src/environments/environment.ts (658 bytes)
CREATE SuperHeroClient/src/app/app-routing.module.ts (245 bytes)
CREATE SuperHeroClient/src/app/app.module.ts (393 bytes)
CREATE SuperHeroClient/src/app/app.component.html (23115 bytes)
CREATE SuperHeroClient/src/app/app.component.ts (219 bytes)
CREATE SuperHeroClient/src/app/app.component.css (0 bytes)
✓ Packages installed successfully.
  Directory is already under version control. Skipping initialization of git.

C:\Users\jabb\source\repos\Tec\2022\Q4\super-hero-project-JackBaltzer\Super Hero Project>[]
```

Kør applikationen, og bekræft at klienten kan køre. Benyt kommandoen: **npm start** og følgende burde dukke op i command vinduet

```
C:\Users\jabb\source\repos\Tec\2022\Q4\super-hero-project-JackBaltzer\Super Hero Project>npm start
> super-hero-client@0.0.0 start
> ng serve

✓ Browser application bundle generation complete.

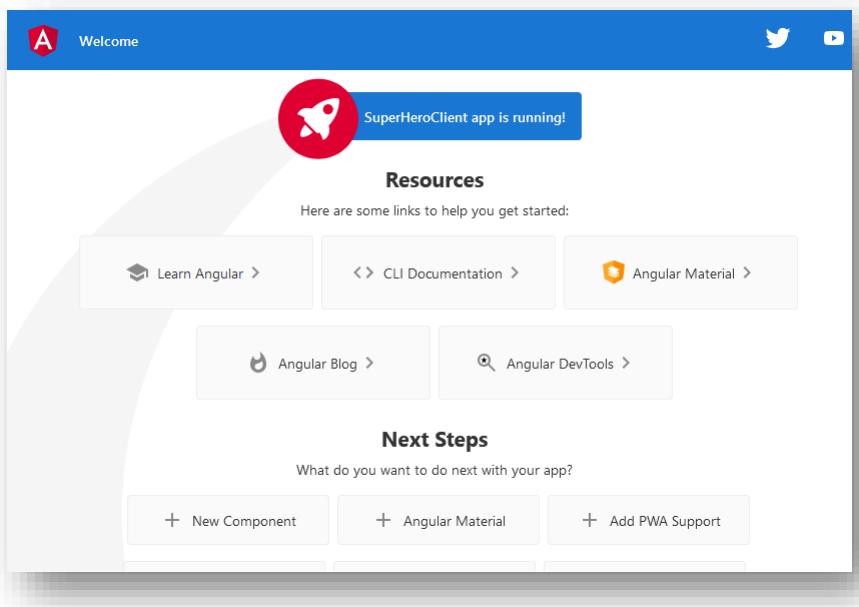
Initial Chunk Files      | Names          | Raw Size
vendor.js                | vendor         | 2.10 MB |
polyfills.js              | polyfills     | 318.07 kB |
styles.css, styles.js    | styles         | 210.10 kB |
main.js                   | main          | 50.29 kB |
runtime.js                | runtime        | 6.53 kB |

| Initial Total | 2.67 MB

Build at: 2022-11-02T11:42:02.279Z - Hash: 535f3fc9a3bb280 - Time: 19010ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **

✓ Compiled successfully.
[]
```

åben en browser og besøg <http://localhost:4200> du kan også holde [CTRL] nede og klikke på linket fra cmd. Browseren bør indlæse en side der ser ud som denne:



Super Hero Project

Forside komponentet

Vi skal have et komponent som repræsenterer forsiden på vores client. Her vil vi benytte en af Angular's nyere funktioner, kaldet **Standalone Components**. Det er super smart og simplificerer vores koder markant.

Udover **Standalone**, vil vi også benytte en flad fil struktur, og minimere mængden af filer for hvert komponent.

`ng g c frontpage -t -s --standalone --flat` (*alt er med små bogstaver*)

Her er en forklaring på linjen.

ng = vi kører en angular kommando.

g = generer, dvs vi ønsker at oprette noget.

c = component, det er denne type fil vi gerne vil oprette.

frontpage = navnet på det komponent vi ønsker at oprette (*inklusiv eventuelle mapper*)

-t = inline-template, dvs der oprettes ikke en **.html** template fil, template skrives direkte i Component filen.

-s = inline-styles, dvs der kommer ingen **.css** fil til komponentet, css skrives også direkte i Component filen.

--standalone = komponentet oprettes som standalone, dvs ingen tilføjelse i **app.module.ts** filen, samt at dette komponent selv sørger for de **imports** der er nødvendige.

--flat = der oprettes ingen mappe til komponentet, det er ret unødvendigt da komponentet bliver til 1 fil.

Angular CLI opretter en enkelt fil kaldet **frontpage.component.ts**, i roden af app mappen, som indeholder følgende koder

```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-frontpage',
  standalone: true,
  imports: [CommonModule],
  template: `
    <p>
      frontpage works!
    </p>
  `,
  styles: [
  ]
})
export class FrontpageComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }
}
```

Routing og Lazy-Loading af komponenter

For at få indlæst **FrontpageComponent** som vores forside, skal vi opsætte routing modulet så det indlæser **Frontpage** når der står <http://localhost:4200> i adressebaren.



Super Hero Project

Det klares i `app-routing.module.ts`, hvor vi tilføjer stien og den handling der skal ske når stien rammes.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

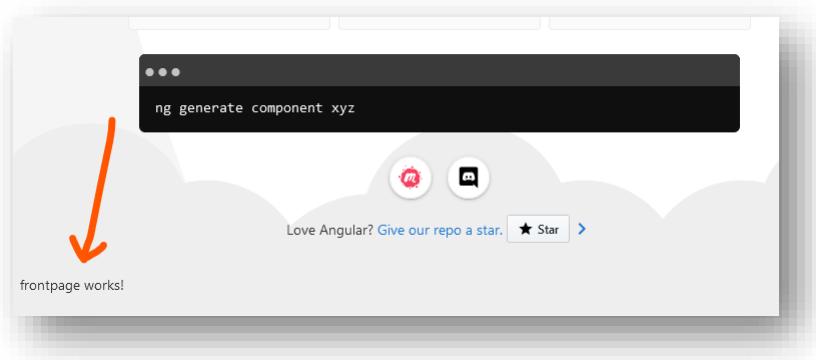
const routes: Routes = [
  { path: '', loadComponent: () => import('./frontpage.component').then(it => it.FrontpageComponent) }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

`path` er hvad der skal stå i adressebaren, udover <http://localhost:4200>

`loadComponent` er den handling der skal ske. Her har vi en anonym funktion (*arrow function*) som udføres når routen rammes. `import` er et promise som `resolves` i `then` metoden, når filen er hentet. Dvs det er først når nogen faktisk besøger <http://localhost:4200> modulet bliver indlæst, frem for at være en del af `main` pakken der hentes fra serveren.

Køres siden i browseren, vil den se ud som før med det klassiske angular forside, men se nederst på siden, der står frontpage works:



app.component som flat, inline komponent

Lad os omskrive `app.component.ts` en smule, så det også optræder som et `flat inline component`. Fjern de områder i `@Component`, som definerer `templateUrl` og `stylesUrls`, og skriv i stedet følgende:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `<router-outlet></router-outlet>`,
  styles: []
})
export class AppComponent {
  title = 'SuperHeroClient';
}
```

Det er super vigtigt at lægge mærke til de 2 back-ticks i template. De skal være sådan: `` den sættes med **[SHIFT]** og tasten til venstre for **backspace** (den med | pipe)... den nuværende side vil virke fint, men lige så snart der skal udskrives variabler og andet logik, samt din template skal indeholde linjeskift, så fejler template



Super Hero Project

Hvis det er normale ' eller " apostrof eller gæseøje.

```
  <!-- @Component({  
    selector: 'app-root',  
    template: `<router-outlet></router-outlet>`,  
    styles: []  
})  
export class AppComponent {  
  -->
```

`router-outlet` er det tag hvor angular indsætter det komponent som routing-modulet henter.

Slet også de to filer `app.component.html` og `app.component.css`, de er ikke længere nødvendige.

Hvis `app.component template` vokser sig stor engang i fremtiden, kan man altid oprette en ny `app.component.html` og pege på den med `templateUrl` egenskaben.

Hero Model

For at repæsentere en Hero i vores Angular applikation, kan vi oprette en kode struktur der minder om vores DTO HeroResponse. Det vil gøre vores arbejde med Hero meget lettere.

Så opret en mappe kaldet `_models` i `app` mappen. Og i `_models` oprettes en fil kaldet `hero.ts`:

```
export interface Hero {  
  id: number;  
  heroName: string;  
  realName: string;  
  place?: string;  
  debutYear: number;  
}
```

Læg mærke til egenskaberne står med **små forbogstaver**, det er vigtigt de står som de ser ud i `Swagger`, dog uden gæseøjne, da det er en egenskab og ikke en JSON objekt.

```
200 Response body  
{  
  "id": 1,  
  "heroName": "Superman",  
  "realName": "Clark Kent",  
  "place": "Metropolis",  
  "debutYear": 1938  
}
```

Angular HeroService

Det vil være smart at have et værktøj som kan tage sig af kommunikationen imellem client og server, "template" og "database" hvis man vil se lidt på det som serverens lag-indeling.

Vi opretter en service, til at klare data overførsel.

```
C:\Users\jabb\source\repos\Tec\2022\Q4\super-hero-project-JackBaltzer\Super Hero Project\SuperHeroClient> ng g s _services/hero  
CREATE src/app/_services/hero.service.ts (133 bytes)  
C:\Users\jabb\source\repos\Tec\2022\Q4\super-hero-project-JackBaltzer\Super Hero Project\SuperHeroClient>
```

`ng` = vi vil køre en AngularCLI commando.

`g` = der er noget vi ønsker at generere.

`s` = det er en service vi vil genere

`_services/hero` = navnet på den service vi ønsker, inklusiv mappen hvor vi ønsker at placere servicen.



Super Hero Project

Resultatet bliver en ny fil kaldet hero.service.ts i mappen _services:

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class HeroService {

  constructor() { }
}
```

environment variable

For at kunne kommunikere med vores [API](#), skal angular kende stien til API'en. Den vil være forskellig når vi udvikler og når siden er færdig og lægges på nettet, så vi skal have fat i [environment.ts](#) filen og tilføje en egenskab kaldet [apiUrl](#):

```
export const environment = {
  production: false,
  apiUrl: 'https://localhost:7150/api/'
};
```

Det er den adresse du kan se når [Swagger](#) afprøves, dog uden [Hero](#), da vi i environment filen er generiske:

```
Curl
curl -X 'GET' \
'https://localhost:7150/api/Hero' \
-H 'accept: */*'
```

Request URL

Når vi udgiver applikationen på dens rette domæne, skal vi tilføje til [environment.prod.ts](#) i stedet.

For at indlæse [environment](#) værdien i [HeroService](#), skal vi blot importere environment og oprette en variabel, og så lige huske at tilføje [hero](#) til [apiUrl](#):

```
import { Injectable } from '@angular/core';
import { environment } from 'src/environments/environment';

@Injectable({
  providedIn: 'root'
})
export class HeroService {

  private readonly apiUrl = environment.apiUrl + 'hero';

  constructor() { }
}
```

Angular HeroService HttpClientModule

Vi kunne selv skrive vores egen logik til at kommunikere med API'en, men det er lidt smartere at benytte det værktøj som Angular stiller til rådighed: [HttpClientModule](#).



Super Hero Project

Det indlæses via `app.module.ts` filen, den kan serveres via dependency injection efter behov, i vores app.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http'; ←

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule ←
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Og i `HeroService` beder vi om at få `HttpClient` serveret i `constructoren`:

```
import { HttpClient } from '@angular/common/http'; ←
import { Injectable } from '@angular/core';
import { environment } from 'src/environments/environment';

@Injectable({
  providedIn: 'root'
})
export class HeroService {

  private readonly apiUrl = environment.apiUrl + 'hero';

  constructor(private http: HttpClient) { }
}
```

Angular HeroService getAll()

Metoden til at hente alle Heroes fra API'en er rimelig simpel, dog vil det være smart at benytte endnu et værktøj som Angular benytter ofte, nemlig Observables, som gør det let at håndtere asynkrone kald og giver muligheden for at opdatere data "live" efter behov.

```
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { environment } from 'src/environments/environment';
import { Hero } from '../_models/hero'; ←

@Injectable({
  providedIn: 'root'
})
export class HeroService {

  private readonly apiUrl = environment.apiUrl + 'hero';

  constructor(private http: HttpClient) { }

  getAll(): Observable<Hero[]> {
    return this.http.get<Hero[]>(this.apiUrl);
  }
}
```



Super Hero Project

Denne opsætning gør det muligt for os at bede API om en liste af heroes (ramme det endpoint i API der serverer alle Heroes) og lade vores Angular service konvertere dem til et array af _models/Hero objekter.

FrontpageComponent heroService

Da HeroService er defineret som en @Injectable, kan den serveres via Dependency Injection i et komponents constructormetode, lige som vi så med HttpClient.

Derudover så kan vi trække på servicen, når komponentet indlæses via komponentes ngOnInit metode. Her kan vi modtage et array af Hero, i en variabel vi har oprettet til formålet, kaldet heroes.

Og i templetten, kan vi løbe igennem alle de heroes der bliver returneret, og udskrive dem på skærmen.

```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HeroService } from './_services/hero.service'; [REDACTED]
import { Hero } from './_models/hero'; [REDACTED]

@Component({
  selector: 'app-frontpage',
  standalone: true,
  imports: [CommonModule],
  template: `
    <p>frontpage works!</p>
    <div *ngFor="let hero of heroes">
      <h3>{{ hero.heroName }}</h3>
      <p>{{ hero.realName }} {{ hero.debutYear }}</p>
    </div>
  `,
  styles: [
  ]
})
export class FrontpageComponent implements OnInit {
  heroes: Hero[] = []; [REDACTED]

  constructor(private heroService: HeroService) { }

  ngOnInit(): void {
    this.heroService.getAll().subscribe(x => this.heroes = x);
  }
}
```

CORS

Det virker desværre ikke helt endnu, da API serveren ikke er sat op til at kunne tage imod forespørgsler fra andre adresser end dens egen. Hvis vi prøver at køre angular forsiden nu, får vi følgende fejlbesked i browseren:

```
⚠️ Forespørgsel til fremmed websted blokeret: Politikken for samme oprindelse tillader ikke læsning af fjerressourcen https://localhost:7150/api/hero. (Årsag: CORS-headeren 'Access-Control-Allow-Origin' findes ikke). Statuskode: 200. [Læs mere]
⚠️ ▶ ERROR
  ▶ Object { headers: {...}, status: 0, statusText: "Unknown Error", url: "https://localhost:7150/api/hero", ok: false, name: "HttpErrorResponse", message: "Http failure response for https://localhost:7150/api/hero: 0 Unknown Error", error: error }
```



Super Hero Project

Vi skal tilpasse Program.cs filen i SuperHeroAPI projektet. Her tilføjes en UseCors regl, som åbner op for al kommunikation til serveren, med alle headers og alle requestmetoder.

```
app.UseHttpsRedirection();
app.UseCors(policy => policy.AllowAnyHeader().AllowAnyMethod().AllowAnyOrigin());
app.UseAuthorization();
```

Genstart API serveren med [CTRL]+[F5] og prøv at genindlæse angluar siden i browseren. Nu burde der vises en liste af Heroes (med mindre du har slettet alle fra din database).



Angular og Github

Gem og commit til github... Dog skal du lige sikre dig at `node_modules` IKKE bliver synkroniseret til github...

Spørg hvis der er den mindste tvivl om hvad der menes.

Angular Hero Administration

Lad os oprette et `standalone` component til at håndtere den fulde **CRUD** på heroes.

```
ng g c admin/hero -t -s --standalone --flat
```

Her oprettes en mappe kaldet `admin`, og `hero` komponentet oprettes i den mappe. Det er blot for at strukturere vores app en smule, denne admin mappe oprettes.



Super Hero Project

Vi opretter en route til Hero administrationen i `app-routing.module.ts` og sørger for **lazy-loading**:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  { path: '', loadComponent: () => import('./frontpage.component').then(it => it.FrontpageComponent) },
  { path: 'admin/hero', loadComponent: () => import('./admin/hero.component').then(it => it.HeroComponent) }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Og nu hvor vi har 2 sider vi kan nagivere imellem, så giver det mening at tilføje en lille menu til vores applikation.

Her ændres på `app.component.ts`, hvor template området udvides med en simpel lille **nav**.

Læg mærke til `a` tagget ikke har en `href`, men i stedet har fået et `routerLink`. Det gør at siden ikke navigerer fysisk imellem de to routes, men i stedet lader angular udskifte indholdet baseret på det komponent som router modulet serverer... det er smart. Husk at `router-outlet` stadig skal være tilstede.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <nav>
      <a routerLink="/">Forside</a> |
      <a routerLink="/admin/hero">Hero Admin</a>
    </nav>
    <router-outlet></router-outlet>
  `,
  styles: []
})
export class AppComponent {
  title = 'SuperHeroClient';
}
```



Super Hero Project

HeroAdmin variablene og opsætningen

Vores adminpanel har brug for 2 områder. En liste over de **Heroes** vi har i databasen, og en formular til at oprette/rette en enkelt **Hero**. Tilpas **admin/hero.component.ts** filen, med følgende tilføjelser.

```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { Hero } from '../_models/hero'; ━━━━  
━━━  
> @Component({ ...  
})  
export class HeroComponent implements OnInit {  
  heroes: Hero[] = []; ━━━━  
  hero: Hero = this.resetHero(); ━━━━  
  
  constructor() { }  
  
  ngOnInit(): void {  
  }  
  
  resetHero(): Hero {  
    return { id: 0, heroName: '', realName: '', debutYear: 0 };  
  }  
}
```

heroes variablen er et array af **Hero**, som sættes til et tomt array fra starten af.

hero variablen, er en enkelt **Hero**, som vil være den vi arbejder med når vi opretter og rette en **Hero**.

Vi får brug for at "nulstille" den enkelte **Hero** på forskellige tidspunkter undervejs, så der er også oprettet en metode der returnerer et nulstillet **Hero** objekt.

HeroAdmin template

Den html del vi har brug for, er vist på følgende billede. Der er en samling af **input** felter, som alle er **to-vejs-bundet** med en egenskab i **Hero** objektet. For at kunne to-vejs-binde værdierne, skal vi bruge **[(ngModel)]** som findes i **FormsModule**, der skal importeres.

Udover inputfelterne, er der to knapper. 1 til at gemme, og en til at annullere. Funktionaliteten er ikke tilføjet endnu.

Under formularfelter og knapper, er der en **table**, som er her alle de eksisterende **Heroes** udskrives med en ***ngFor** løkke. Ud for hver Hero er der 2 knapper. En til at redigere den pågældende Hero, og en anden til at slette Hero. Stadig uden noget konkret funktionalitet endnu.

Det som er værd at lægge mærke til, er at der er flere "hero" objekter på siden... en i formularen, som peger på det Hero objekt som ligger i komponentet. Og en andet "hero" som ligger i tabellen, og repræsenterer det enkle Hero objekt som *ngFor løkken arbejder på mens den løber igennem samlingen af Hero objekter i Heroes arrayet. Pas på med at forveksle de to! Omdøb eventuelt det ene, hvis du finder det lettere at arbejde med.

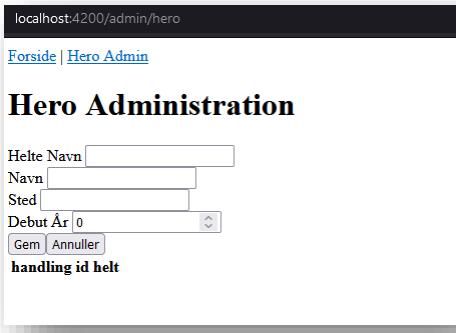


Super Hero Project

```
@Component({
  selector: 'app-hero',
  standalone: true,
  imports: [CommonModule, FormsModule],
  template: `
    <h1>Hero Administration</h1>
    Helte Navn <input [(ngModel)]="hero.heroName"><br>
    Navn <input [(ngModel)]="hero.realName"><br>
    Sted <input [(ngModel)]="hero.place"><br>
    Debut År <input type="number" [(ngModel)]="hero.debutYear"><br>
    <button>Gem</button>
    <button>Annuller</button>

    <table>
      <tr>
        <th>handling</th>
        <th>id</th>
        <th>helt</th>
      </tr>
      <tr *ngFor="let hero of heroes">
        <td>
          <button>Ret</button>
          <button>Slet</button>
        </td>
        <td>{{ hero.id }}</td>
        <td> {{ hero.heroName }}</td>
      </tr>
    </table>
  `,
  styles: [
  ]
})
```

Køres applikationen, vil den se sådan her ud i browseren:



Super Hero Project

Der mangler nogle helte! Løsningen på det problem er præcis det samme som på vores [FrontpageComponent](#), hvor `HeroService` injectes og `ngOnInit` henter benytter `getAll` til at hente alle `Heroes`:

```
export class HeroComponent implements OnInit {
  heroes: Hero[] = [];
  hero: Hero = this.resetHero();

  constructor(private heroService: HeroService) { }

  ngOnInit(): void {
    this.heroService.getAll().subscribe(x => this.heroes = x);
  }

  resetHero(): Hero {
    return { id: 0, heroName: '', realName: '', debutYear: 0 };
  }
}
```

Angular HeroAdmin Delete Hero

Nu hvor vi kan hente alle Heroes og vise dem, så lad os straks slette en af dem.

Tilføj denne `delete()` metode til [_services/hero.service.ts](#):

```
delete(heroId: number): Observable<Hero> {
  return this.http.delete<Hero>(this.apiUrl + '/' + heroId);
}
```

Her udføres en `HttpDelete` forespørgsel til [API'en](#) og vi får en `Hero` retur.

Og derefter tilføjes denne `delete()` metode til [admin/hero.component.ts](#):

```
delete(hero: Hero): void {
  if (confirm('Er du sikker på du vil slette: ' + hero.heroName + '?')) {
    this.heroService.delete(hero.id).subscribe(() => {
      this.heroes = this.heroes.filter(h => hero.id != h.id);
    });
  }
}
```

Formålet med if-sætningen, er at forhindre fejl-klik på slet knappen. Det er altid en god ide at spørge om brugeren rent faktisk ønsker at slette...

Når `heroService` er færdig med at slette, så fjerner vi den pågældende `Hero` fra `Heroes`, med en filter metode:

Find slet knappen i template koden, og tilføj denne (`click`) handler som kalder `delete()` når knappen aktiveres:

```
<tr *ngFor="let hero of heroes">
  <td>
    <button>Ret</button>
    <button (click)="delete(hero)">Slet</button>
  </td>
  <td>{{ hero.id }}</td>
  <td>{{ hero.heroName }}</td>
</tr>
```

Hvis du tester det i browseren, skal du huske at `Hero` rent faktisk bliver slettet, hvis du klikker og accepterer!

Så test det i browseren.



Super Hero Project

Angular HeroAdmin Create

Nu hvor vi kan slette, så giver det nok god mening at vi også kan oprette.

Tilføj denne `create()` metode til `_service/hero.service.ts`, som tager sig af at sende et `Hero` objekt til API'en:

```
create(hero: Hero): Observable<Hero> {
  return this.http.post<Hero>(this.apiUrl, hero);
}
```

Derefter tilføjes denne `save()` metode til `admin/hero.component.ts`:

```
save(): void {
  if (this.hero.id === 0) {
    this.heroService.create(this.hero).subscribe({
      next: (x) => {
        this.heroes.push(x);
        this.hero = this.resetHero();
      },
      error: (err) => {
        console.warn(Object.values(err.error.errors).join(' ', ''));
      }
    });
  } else {
    // update...
  }
}
```

Princippet i denne `save()` metode, er at den vil blive benyttet både ved `opret`, og ved `rediger` handlingerne.

Derfor starter metoden med en `if-sætning` der kigger på `this.hero.id`, som jo er `0` pr default.

Vi sender `hero` direkte til `heroService`, og håndtere svaret via `next` metoden i `subscribe`. Her nulstiller vi `Hero`, som sørger for at formularen nulstilles, og tilføjer den nye `Hero` til `Heroes` arrayet.

Skulle der være opstået en fejl, udskrives detaljerne via `console.warn`. (*det ændrer vi på et senere tidspunkt*)

For at udføre `save()` metoden, skal du finde den button der ligger i template, som håndterer gem handlingen.

Og tilføje `(click)` eventen:

```
Debut År <input type="number" [(ngModel)]="hero.debutYear"><br>
<button (click)="save()">Gem</button>
<button>Annuller</button>
```

Test i browseren, at der kan indsættes, test også hvad konsollen udskriver hvis der mangler værdier i formularen.

Angular HeroAdmin annuller funktion

Der er en knap til at annullere, i forbindelse med formularen. Dens funktion er at nulstille formularen, både hvis man er ved at oprette, og hvis man er ved redigere, men man alligevel ikke ønsker at gemme.

Vi opretter en metode i `admin/hero.component.ts` kaldet `cancel()` som tager sig af annulleringen.

```
cancel(): void {
  this.hero = this.resetHero();
}
```



Super Hero Project

Og den metode bindes til knappen ved formularen:

```
Debut År <input type="number" [(ngModel)]="hero.debutYear"><br>
<button (click)="save()">Gem</button>
<button (click)="cancel()">Annuller</button>
```

Angular HeroAdmin Edit

For at rette en **Hero**, skal vi håndtere to scenarier. Det første er **valget** af hvilken Hero der skal redigeres, og derefter håndtere at **sende** data til API så det kan gemmes i databasen.

Den første del kræver en ny metode i **admin/hero.component.ts**:

```
edit(hero: Hero): void {
  Object.assign(this.hero, hero);
}
```

Formålet med metoden, er at kopiere værdierne fra den medsendte **Hero**, til komponentets **Hero**, så formularen vil blive udfyldt med de ønskede værdier. Denne metode bindes på **Ret** knappen i tabellen.

```
<tr *ngFor="let hero of heroes">
  <td>
    <button (click)="edit(hero)">Ret</button>
    <button (click)="delete(hero)">Slet</button>
  </td>
  <td>{{ hero.id }}</td>
  <td>{{ hero.heroName }}</td>
</tr>
```

_services/hero.service.ts skal også opdateres, så den har en metode der kan kalde API med den ændrede **Hero** data:

```
update(hero: Hero): Observable<Hero> {
  return this.http.put<Hero>(this.apiUrl + '/' + hero.id, hero);
}
```

Derefter skal vi håndtere når der klikkes på **gem** knappen ved formularen, for nu vil **hero.id** være alt andet end 0.

```
save(): void {
  if (this.hero.id == 0) {
    this.heroService.create(this.hero).subscribe({
      next: (x) => {
        this.heroes.push(x);
        this.hero = this.resetHero();
      },
      error: (err) => {
        console.warn(Object.values(err.error.errors).join(', '));
      }
    });
  } else {
    this.heroService.update(this.hero).subscribe({
      error: (err) => {
        console.warn(Object.values(err.error.errors).join(', '));
      },
      complete: () => {
        this.heroService.getAll().subscribe(x => this.heroes = x);
        this.hero = this.resetHero();
      }
    });
  }
}
```



Super Hero Project

Angular HeroService findById()

Den sidste metode vi mangler, er den hvor vi henter en enkelt Hero ud baseret på dens id. Det er her vi forsøger at ramme `findById` endpointet på API, så lad os kalde vores `_service/hero.service.ts` metode det samme:

```
findById(heroId: number): Observable<Hero> {
  return this.http.get<Hero>(this.apiUrl + '/' + heroId);
}
```

Med den struktur vi har lige nu, har vi ikke behov for at kalde `findById`, så lad os oprette et nyt komponent som kan hente og vise en specifik Hero baseret på en id fra adressebaren:

```
ng g c hero-detail -t -s --standalone --flat
```

Nu har vi et detalje komponent kaldet `HeroDetailComponent` som skal kunne vise et enkelt `Hero` objekt. Der skal være et `Hero` objekt i komponentet, så opret en med default data, samt forbind både `HeroService` og `ActivatedRoute`, som begge importeres. `RouterModule` benyttes til at aktivere `routerLink` funktionaliteten.

I `ngOnInit` metoden kigger vi på parametrene der står i adressebaren, og tager den der hedder `heroId` og benytter den til at finde en `Hero` via vores Service.

```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HeroService } from './_services/hero.service';
import { ActivatedRoute, RouterModule } from '@angular/router';
import { Hero } from './_models/hero';

@Component({
  selector: 'app-hero-detail',
  standalone: true,
  imports: [CommonModule, RouterModule],
  template: `
    <p>hero-detail works!</p>
    <h1>{{ hero.heroName | uppercase }}</h1>
    <a routerLink="/">Forside</a>
  `,
  styles: [
  ]
})
export class HeroDetailComponent implements OnInit {
  hero: Hero = { id: 0, heroName: '', realName: '', debutYear: 0 };

  constructor(private heroService: HeroService, private route: ActivatedRoute) { }

  ngOnInit(): void {
    this.route.params.subscribe(params => {
      this.heroService.findById(params['heroId']).subscribe(hero => this.hero = hero);
    });
  }
}
```



Super Hero Project

Route til `HeroDetail`, med parametre. Læg især mærke til `:heroId` i `path`... dette er en **parameter**, hvor vi tager den værdi der står på positionen, frem for den faktiske tekst, dvs et tal for enden af adressen.

```
const routes: Routes = [
  {
    path: '', loadComponent: () =>
      import('./frontpage.component').then(it => it.FrontpageComponent)
  },
  {
    path: 'hero/:heroId', loadComponent: () =>
      import('./hero-detail.component').then(it => it.HeroDetailComponent)
  },
  {
    path: 'admin/hero', loadComponent: () =>
      import('./admin/hero.component').then(it => it.HeroComponent)
  }
];
```

Nu kan vi ændre en lille smule på `FrontpageComponent`, så der kommer et link ud for hver `Hero`, så vi kan komme let til `HeroDetail`.

Da vi har behov for at udskrive en variabel inde i routerlinket, er vi nødt til at pakke det ind i firkantparantser, så kan angular parse linket med den sti, og den variabel som står i dens value.

```
import { Component, OnInit } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HeroService } from './_services/hero.service';
import { Hero } from './_models/hero';
import { RouterModule } from '@angular/router';

@Component({
  selector: 'app-frontpage',
  standalone: true,
  imports: [CommonModule, RouterModule],
  template: `
    <p>frontpage works!</p>
    <div *ngFor="let hero of heroes">
      <h3><a [routerLink]="['/hero', hero.id]">{{ hero.heroName }}</a></h3>
      <p>{{ hero.realName }} {{ hero.debutYear }}</p>
    </div>
  `,
  styles: []
})
export class FrontpageComponent implements OnInit {
  heroes: Hero[] = [];

  constructor(private heroService: HeroService) { }

  ngOnInit(): void {
    this.heroService.getAll().subscribe(x => this.heroes = x);
  }
}
```

Test i browseren, at altting fungerer som det skal.

Gem og commit til github!



Super Hero Project

Angular Reactive Forms

Nogle gange er en simpel ngModel bare ikke helt nok... måske vil vi gerne have validerings fejlbeskeder, eller style formfelter på en bestemt måde, hvis feltet er forkert udfyldt. Her kan ReactiveFormsModule hjælpe os på vej.

Og det er relativt simpelt at sætte op, når vi allerede har vores fulde CRUD funktionalitet.

Følgende ændringer kan vi foretage på [admin/hero.component.ts](#), for at tilføje **ReactiveFormsModule**:

Vi har brug for en repræsentation af formularen i typescript, så tilføj følgende koder i **HeroComponent** klassen.

Husk at import **FormGroup**, **FormControl**, **ReactiveFormsModule** og **Validators** fra [@angular/forms](#).

Sørg også for at egenskaberne hedder det samme som din **Hero** model:

```
heroForm: FormGroup = this.resetForm();

resetForm(): FormGroup {
  return new FormGroup({
    heroName: new FormControl(null, Validators.required),
    realName: new FormControl(null, Validators.required),
    place: new FormControl(null),
    debutYear: new FormControl(0, [
      Validators.required,
      Validators.min(1),
      Validators.max(2500)])
  });
}
```

Edit funktionen skal opdateres, så vi kan sætte værdierne fra det valgte Hero objekt i formularen:

```
edit(hero: Hero): void {
  // copies the selected hero values into the form.
  // this works ONLY IF form-property-names match entity-property-names
  this.heroForm.patchValue(hero);
}
```



Super Hero Project

Save funktionen skal også opdateres en smule. Og her kommer vi også til at ændre lidt på vores **HeroService**, da vi bliver nødt til at fortælle hvilken **heroId** der skal opdateres, da **heroId** ikke er en del af formularen.

```
save(): void {
  // only do work if the form is valid after it has been touched
  if (this.heroForm.valid && this.heroForm.touched) {
    if (this.hero.id == 0) {
      this.heroService.create(this.heroForm.value).subscribe({
        next: (x) => {
          this.heroes.push(x);
          this.cancel();
        },
        error: (err) => {
          console.warn(Object.values(err.error.errors).join(', '));
        }
      });
    } else {
      this.heroService.update(this.hero.id, this.heroForm.value).subscribe({
        error: (err) => {
          console.warn(Object.values(err.error.errors).join(', '));
        },
        complete: () => {
          this.heroService.getAll().subscribe(x => this.heroes = x);
          this.cancel();
        }
      });
    }
  }
}
```

HeroService ændringen:

```
update(heroId:number, hero: Hero): Observable<Hero> {
  return this.http.put<Hero>(this.apiUrl + '/' + heroId, hero);
}
```

Så er **typescript** ændringerne udført, og vi kan nu koncentrere os om **html** ændringerne (tabellen er uændret). Hele formular området skal omskrives, så den ser ud som det følgende billede.

formGroup skal pege på det **FormGroup** objekt der er oprettet i komponentet, kaldet **heroForm**.

ngSubmit udføres når formularen submittes, her har vi **save()** metoden.

Hvert inputfelt skal pege på en **FormControl** i **heroForm** objektet.

Gem knappen tilføjes en **disabled** funktion, som sikrer knappen kun kan aktiveres hvis formularen er **valid**.



Super Hero Project

Og annuler knappen ændres til en **type button**, så kan den knap ikke submitte formularen.

```
<h1>Hero Administration</h1>

<form [formGroup]="heroForm" (ngSubmit)="save()">
  <div class="formControl">
    <label>Helte Navn</label>
    <input type="text" formControlName="heroName">
  </div>
  <div class="formControl">
    <label>Navn</label>
    <input type="text" formControlName="realName">
  </div>
  <div class="formControl">
    <label>Sted</label>
    <input type="text" formControlName="place">
  </div>
  <div class="formControl">
    <label>Debut År</label>
    <input type="number" type="text" formControlName="debutYear">
  </div>
  <button [disabled]="!heroForm.valid">Gem</button>
  <button type="button" (click)="cancel()">Annuler</button>
</form>
```

For at få det til at se lidt smørre ud, kan følgende styles tilføjes.

Brug lige lidt tid på at sikre SAMTLIGE tegn er sat, der er mange små finurligheder i den styles definition.

```
styles: [
  .formControl label{
    display: inline-block;
    width:85px;
    text-align:right;
    padding:0 5px;
  }
  .formControl{
    margin:7px 0;
  }
  input[type='text']{
    border:solid 1px #333;
  }
  input.ng-invalid.ng-touched{
    border:solid 1px red;
  }
]
```

Udskriv specifikke beskeder ud for hvert input felt, tilføj et span tag som dette, læg mærke til det er på to linjer!

```
<div class="formControl">
  <label>Helte Navn</label>
  <input type="text" formControlName="heroName">
  <span class="error" *ngIf="heroForm.get('heroName')?.invalid
    && heroForm.get('heroName')?.touched">Udfyld!</span>
</div>
```

Gem og commit til github.

