

Backend Development

Tuur Vanhoutte

31 maart 2021

Inhoudsopgave

1 .Net Core & Web API	1
1.1 .NET Historiek	1
1.1.1 .NET Framework	1
1.1.2 Rond 2012 enkel strategische veranderingen	1
1.1.3 Ontwikkeling .NET Core	1
1.1.4 Tijdslijn	1
1.2 Soorten .NET applicaties	2
1.2.1 Type applicaties in .NET 5:	2
1.2.2 Ondersteuning voor verschillende talen	2
1.2.3 Toekomst desktop development	2
1.3 ASP.NET Core	2
1.3.1 Verschillende soorten applicaties zijn mogelijk:	2
1.3.2 ASP.NET Web API	3
1.4 HTTP (Herhaling)	3
1.5 API Design	3
1.5.1 Richtlijnen bij het opstellen van API URL's	3
1.6 Anatomie van een ASP.NET Web API Project	4
1.6.1 csproj file	4
1.6.2 Program.cs	4
1.6.3 Startup.cs	4
1.6.4 Appsetting.json	6
1.7 Controllers	6
1.7.1 Endpoints	7
1.8 Model binding	9
1.8.1 Voorbeelden	10
1.9 Configuration	11
1.9.1 appsettings.json	11
1.9.2 Azure Keyvault	13
1.10 Wat moet je kennen?	13
2 Web API DTO, Validation, Versioning	13
2.1 Validation	13
2.1.1 Attributen	14
2.1.2 Ingebouwde attributen:	14
2.1.3 Custom Validation Attributes	15
2.2 QueryString	15
2.2.1 Werking	16
2.3 DTO	16
2.3.1 Werking	17
2.4 Versioning	18
2.4.1 Registreren bij services	19
2.4.2 Configuratie	19
2.4.3 Controller attributes	19
2.4.4 Endpoints mappen aan een specifieke versie	20
2.4.5 Testen	20
2.5 Binary bestanden	20
2.5.1 Werking	20
2.5.2 Testen	21
2.5.3 Bestanden downloaden	21
2.6 CSV files	22

2.6.1	CSV Helper	22
2.6.2	Werking	22
2.7	Caching	22
2.7.1	Response Caching (HTTP Caching)	23
2.7.2	Output Caching	24
2.8	Wat moet je kennen?	25
3	Docker & Docker-compose	25
3.1	Waarom?	25
3.2	What?	25
3.3	Docker images	26
3.3.1	Image layers	26
3.4	Lightweight	26
3.5	Microservices en Docker	27
3.5.1	Demo	27
3.6	Docker basiscommando's	28
3.6.1	Images ophalen	28
3.6.2	Containers runnen	28
3.6.3	Interactie & logs	29
3.6.4	Containers builden	29
3.7	Dockerfile	29
3.7.1	Dockerfile optimalisatie	29
3.8	Docker Compose	30
3.8.1	Docker Compose terminologie	31
3.9	Wat moet je kennen?	33
4	EF Core	33
4.1	Probleemstelling	33
4.1.1	Niets fout met SQL en relationele databases!	33
4.1.2	Toch gebruiken we nog plain SQL	34
4.2	EF Core	34
4.2.1	Database First	34
4.2.2	Code First	34
4.3	EF Core installatie	35
4.4	Models	36
4.4.1	Eén-op-veel-relaties	36
4.4.2	Enkele attributen	37
4.4.3	Fluent API	37
4.5	EF Context	37
4.5.1	Aanmaken van de database en/of tabellen	38
4.6	Seeding	38
4.6.1	In C# code	39
4.7	Relations	39
4.7.1	One to Many	39
4.7.2	One to One	40
4.7.3	Many to Many	40
4.8	SELECT, INSERT, UPDATE, DELETE	41
4.8.1	Hoe halen we records op (SELECT)?	41
4.8.2	Hoe voegen we een record toe (INSERT)?	41
4.8.3	Hoe verwijderen we records (DELETE)?	42
4.8.4	Hoe halen we records op (SELECT)?	42
4.8.5	Toevoegen gerelateerde data (één op veel)	42

4.8.6	Hoe gerelateerde records ophalen	43
4.9	Querying van data	44
4.9.1	Ophalen van alle Blogs <i>zonder</i> Posts	44
4.9.2	Ophalen van alle Blogs <i>met</i> Posts	44
4.9.3	Ophalen van 1 specifiek item	45
4.9.4	Voorbeeld: comments van Post ophalen (3 tabellen diep)	45
4.10	Generated Values	46
4.11	Extra info	46
4.11.1	Wat moet je zeker kennen?	46
5	Services & Repositories	46
5.1	Probleemstelling	46
5.1.1	Wat hebben we reeds gezien?	46
5.1.2	Probleemstelling	47
5.1.3	Oplossing: Design Patterns	47
5.1.4	Repository & Service pattern	47
5.2	Repositories	48
5.2.1	In de praktijk	48
5.2.2	Probleem	48
5.2.3	Oplossing	49
5.3	Dependency Injection (DI)	50
5.3.1	Wat is dat nu eigenlijk	50
5.3.2	Werking	50
5.3.3	In de praktijk	51
5.4	Services	52
5.4.1	In de praktijk	52
5.5	Wat is belangrijk?	53
6	Testing	53
6.1	Probleemstelling	53
6.2	Waarom testing?	54
6.3	2 soorten	54
6.3.1	Manual testing	54
6.3.2	Automated test	54
6.4	Types tests	55
6.4.1	Unit test	55
6.4.2	Integration test	55
6.4.3	Functional test	55
6.4.4	Acceptance tests	56
6.4.5	Performance test	56
6.4.6	Smoke test	56
6.4.7	Regression test	56
6.4.8	Stress test	56
6.5	Test Frameworks	56
6.5.1	XUnit	56
6.6	Unit test in detail	57
6.6.1	Voorbeeld	57
6.6.2	Exceptions testen	58
6.7	Integration test in detail	58
6.7.1	Testen van de volledige flow	58
6.7.2	Toevoegen van een test	59
6.7.3	WebApplicationFactory	59

6.8	Test driven development	60
6.9	Code Coverage	60
6.9.1	Wat zal Code Coverage o.a. proberen te bepalen?	60
6.10	Wat is belangrijk?	60

1 .Net Core & Web API

1.1 .NET Historiek

1.1.1 .NET Framework

- Ontwikkeling .NET Framework & C# begonnen einde Jaren 90 (Opkomst van Java, Microsoft moest reageren ⇒ C#)
- Versie 1.0 .NET Framework op 13 February 2002
- Windows Only
 - Beperkt open source via Home | Mono (mono-project.com)
- Was vooral framework voor desktop applicaties (Winforms, WPF)
- Web applicaties via ASP.NET Forms
- Laatste versie .NET Framework is 4.8 (april 2019)
- Geen verdere ontwikkeling meer van .NET Framework

1.1.2 Rond 2012 enkel strategische veranderingen

- Dominantie Windows was minder groot geworden
 - Opkomst mobile devices (iPhone, Android)
 - Cloud was belangrijker aan het worden (veel Linux)
 - Open-source werd belangrijker
- Microsoft zag een shift van .NET Framework naar andere technologie (Node, Python, etc...)

1.1.3 Ontwikkeling .NET Core

- Cross platform (Windows/Linux/macOS)
- Volledig open-source
 - Samsung TV
 - ARM Raspberry Pi
 - ...
- Clean-up van bestaande .NET Framework code
- Zoveel mogelijk compatibel
- Built for the Cloud

Sinds November 2020 spreken we niet meer over .NET Framework of .NET Core maar over .NET 5, .NET 6, .NET 7, ...

1.1.4 Tijdslijn

- .NET 5 ⇒ November 2020
- .NET 6 ⇒ November 2021 (LTS)
- .NET 7 ⇒ November 2022

- .NET 8 ⇒ November 2023 (LTS)
- LTS ⇒ 3 jaar support
- Zonder LTS ⇒ 1 jaar support

Alles volledig opensource via <https://github.com/dotnet>

1.2 Soorten .NET applicaties

1.2.1 Type applicaties in .NET 5:

- ASP.NET Web Applicaties
- Console Applicaties
- Xamarin
- Winforms & WPF Desktop applicaties

1.2.2 Ondersteuning voor verschillende talen

- C# (wij gebruiken dit)
- Visual Basic
- F# (functional programming)
- C++ (desktop development)

1.2.3 Toekomst desktop development

- Zeker nog belangrijk
- Niet alles kan in de browser (vb zware 3D apps, interfacing met machines etc)
- Zit niet meer in MCT opleiding (wel Xamarin natuurlijk)
- Project MAUI
 - 1 framework voor .NET/Xamarin/Windows Desktop
 - Introducing .NET Multi-platform App UI | .NET Blog : <https://devblogs.microsoft.com/dotnet/introducing-net-multi-platform-app-ui/>

1.3 ASP.NET Core

= Framework voor het bouwen van Web Applicaties

1.3.1 Verschillende soorten applicaties zijn mogelijk:

- Klassieke server-side framework (Razor) (zoals PHP,...)
- Realtime Framework (web sockets etc...) met als naam Signal R
- Frontend Framework Blazor (C# in de browser)
 - Web Assembly support
- **Framework om API's te bouwen , webapi (wij gebruiken dit)**

Cross platform zowel om uit te voeren als te onwikkelen (Visual Studio & Visual Studio Code zijn cross-platform)

1.3.2 ASP.NET Web API

= Framework voor het bouwen van API's voor toepassingen

Deze toepassingen zijn:

- Front-End Web App (Vue, Angular, Blazor,...)
- Desktop Applications
- Alles wat HTTP calls kan uitvoeren en JSON begrijpt

Zelfde als HTTP Triggers in Azure Functions (cfr. Module IoT Cloud Semester 3)

1.4 HTTP (Herhaling)

= Hyper Text Transfer Protocol

- Onderliggende protocol waarop Internet werkt
- Opvragen van tekst, bestanden vanaf servers
- Request *meestal* afkomstig van een webbrowser maar ook smartphone, IoT device
- HTTP zal bepalen hoe een request en response er moeten uitzien
- HTTP bevat een aantal commando's (HTTP Verbs)
- HTTP is stateless, het zal dus geen rekening houden met voorgaande requests
- HTTP is niet sessionless, we kunnen cookies (client-side) gebruiken om data bij te houden
- HTTP is relatief eenvoudig

1.5 API Design

- In deze module gaan we API's programmeren
- De bedoeling is dat frontend applications deze gebruiken
- Onze API's zullen enkel JSON data ontvangen en terugsturen
- Oudere API's systemen keren soms ook XML terug (deze zien wij niet)
- Een API noemen we ook soms een endpoint
- De vorm en naamgeving is hier belangrijk

1.5.1 Richtlijnen bij het opstellen van API URL's

- Gebruik Engelse woorden
- Start met het woord 'api':
 - <http://www.mct.be/api>
- In de URL plaatsen we de naam van objecten die we wensen op te halen
 - <http://www.mct.be/api/courses>

- Willen we specifieke cursus ophalen op basis van zijn code steken we dit in de URL
 - <http://www.mct/be/api/courses/MCT2IOTCLOUD>
- Willen we de weken ophalen, plaatsen we dit ook in de URL
 - <http://www.mct/be/api/courses/MCT2IOTCLOUD/weeks>
- Zoeken in de weken kunnen we als volgt gaan doen:
 - <http://www.mct/be/api/courses/MCT2IOTCLOUD/weeks?q=iothub>
- Wil je een bepaalde week ophalen:
 - <http://www.mct/be/api/courses/MCT2IOTCLOUD/weeks/1>
- Wil je de docent ophalen kan dit op deze manier in de URL:
 - <http://www.mct/be/api/courses/MCT2IOTCLOUD/weeks/1/teacher>

1.6 Anatomie van een ASP.NET Web API Project

1.6.1 csproj file

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <RootNamespace>backend_lab01_wijn</RootNamespace>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="5.0.1" NoWarn="NU1605" />
    <PackageReference Include="Microsoft.AspNetCore.Authentication.OpenIdConnect" Version="5.0.1" NoWarn="NU1605" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="5.6.3" />
  </ItemGroup>
</Project>
```

Figuur 1: csproj voorbeeld

- Beschrijft project
- Zal ingelezen worden door Visual Studio
- Bevat alle referenties naar gebruikte nuget packages
- dotnet build etc zal deze file gebruiken
- We mogen deze wijzigen (maar weet wat je doet)

1.6.2 Program.cs

- Main entry point van ASP.NET API applicatie
- Hier begint alles
- Soms moeten we hier iets instellen
- Voorlopig niet van belang

1.6.3 Startup.cs

- Naam spreekt voor zich, bevat alle startup code voor onze API
- Hier bepalen we welke **services** we wensen te gebruiken van ASP.NET

```
public void ConfigureServices(IServiceCollection services)
```

Figuur 2: ConfigureServices

```
services.AddControllers();
```

Figuur 3: Controllers

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "backend_lab01_wijn", Version = "v1" });
});
```

Figuur 4: Swagger

Definitie 1.1 (Dependency Injection) *Dependency Injection is het injecteren van services in de ASP.NET Container*

(komen we later nog op terug)

```
/* THIS METHOD GETS CALLED BY THE LARUNTIME. USE THIS METHOD TO CONFIGURE THE HTTP PIPELINE */
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
```

Figuur 5: Configureren van de HTTP Pipeline

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseSwagger();
    app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "backend_lab01_wijn v1"));
}
```

Figuur 6: Configureren van services

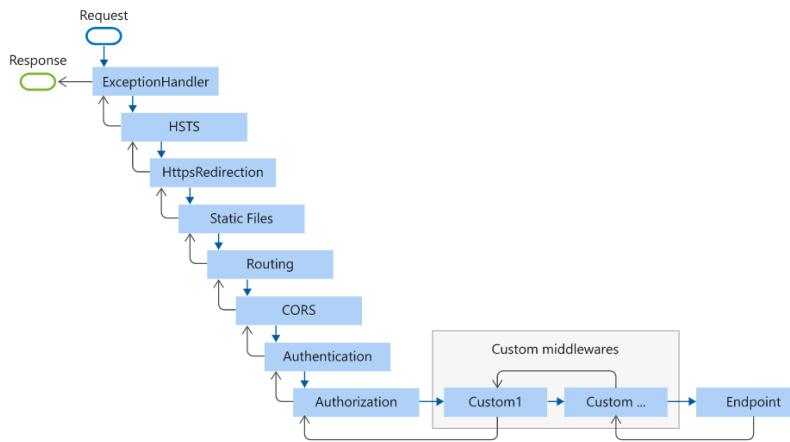
```
app.UseRouting();

app.UseAuthorization();
```

Figuur 7: Configureren van Routing, Security etc...

HTTP Pipeline

- We spreken van middleware componenten in de pipeline
- Deze componenten zijn verbonden en geven de data door aan elkaar
- Ieder component zal zijn bijdrage leveren
- Endpoint is onze code in de controller
- Request en Response gaan door deze pipeline
- ASP.NET Core Middleware: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-5.0>



Figuur 8: De HTTP Pipeline met middleware componenten

1.6.4 Appsetting.json

- = Configuratie info nodig in de applicatie
 - Niet inchecken in GitHub (.gitignore gebruiken)
 - Bv.: Mailserver settings, connectionstrings etc...
 - We zien later (in een labo) hoe we deze kunnen uitlezen
 - We kunnen verschillende files gebruiken:
 - Appsettings.Development.json
 - Appsettings.Production.json
 - ...
 - Configuration in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-5.0>

1.7 Controllers

- Belangrijkste concept in ASP.NET
- Zorgt voor de verwerking van de requests
- Zal een response terugkeren naar de aanvrager (bv. onze React JS applicatie)
- Is klasse in map **Controllers** in onze applicatie
- Naam moet eindigen op **Controller**
- Klasse moet erven van **ControllerBase**
- Attribute `[ApiController]` moet er staan om aan te duiden dat het een API is als we `[Route]` gebruiken op Controller niveau
- Via `[Route]` kunnen we de URL bepalen

```
[ApiController]
[Route("[controller]")]
public class WineController : ControllerBase
{
    public WineController()
    {
    }
}
```

Figuur 9: Voorbeeld Controller

1.7.1 Endpoints

Controller bevat Endpoints = functies die we uitvoeren

- Endpoint bevat HTTP Verb
 - GET
 - POST
 - PUT
- Endpoints voeren de code uit
 - Data ophalen of toevoegen
 - Berekening doen
 - ...

<pre>[HttpGet] 0 references public IEnumerable<Wine> GetWines() { return _wines; }</pre>	<pre>[HttpPost] 0 references public Wine Post(Wine wine) { wine.WineId = _wines.Count + 1; _wines.Add(wine); return wine; }</pre>
--	---

Figuur 10: HTTP GET & POST

Route

- Een route is de URL naar het Endpoint
- Er zijn verschillende routes mogelijk per Endpoint
- Dezelfde routes zijn mogelijk zolang het HTTP Verb verschillend is
- We kunnen parameters megeven in de route via {}
- Indien geen routes: Default is GET met route = naam van Controller

```
[HttpGet]
[Route("wines")]
[Route("wijnen")]
0 references
public IEnumerable<Wine> GetWines()
{
    return _wines;
}

[HttpDelete]
[Route("wines/{wineId}")]
0 references
public ActionResult Delete(int wineId)
{
    return Ok();
}
```

Figuur 11: Meerdere routes per endpoint mogelijk (links). Delete request met parameters (rechts)

```
[ApiController]
[Route("[controller]")]
public class WineController : ControllerBase {

    [HttpGet]
    [Route("wines")]
    [Route("wijnen")]
    public IEnumerable<Wine> GetWines()
    {
        return _wines;
    }
}
```

Figuur 12: <https://localhost:5001/wine/wines> en <https://localhost:5001/wine/wijnen>

```
[ApiController]
[Route("[controller]")]
public class WineController : ControllerBase {

    [HttpGet]
    public IEnumerable<Wine> GetWines()
    {
        return _wines;
    }
}
```

Figuur 13: /wine haalt hij uit de naam WineController

```
[ApiController]
[Route("api")]
public class WineController : ControllerBase {

    [HttpGet]
    [Route("wines")]
    public IEnumerable<Wine> GetWines()
    {
        return _wines;
    }
}
```

Figuur 14: Best practice: met api in de naam

```

[ApiController]
[Route("api")]
public class WineController : ControllerBase {
    [HttpGet]
    [Route("wines/{year}")]
    public IEnumerable<Wine> GetWines(int year)
    {
        return _wines;
    }
}

```

Figuur 15: Route met parameter. <https://localhost:5001/api/wines/2005>

- Endpoints kerent altijd iets terug:
 - Status codes
 - Data die we opvragen
- Niet altijd makkelijk wat je moet kiezen
- Status codes uit HTTP standaard 5 grote verdelingen
 - 1xx: **Informational** – Communicates transfer protocol-level information.
 - 2xx: **Success** – Indicates that the client's request was accepted successfully.
 - 3xx: **Redirection** – Indicates that the client must take some additional action in order to complete their request.
 - 4xx: **Client Error** – This category of error status codes points the finger at clients.
 - 5xx: **Server Error** – The server takes responsibility for these error status codes.

Figuur 16: Status codes

Overzicht statuscodes: <https://restfulapi.net/http-status-codes/>

Meest gebruikt:

- 200 OK
- 201 Created
- 400 Bad Request, 401 Unauthorized, 403 Forbidden, 415 Unsupported Media Type
- 500 Internal Server Error

Retournen van data en statuscodes: veel ingebouwde klassen

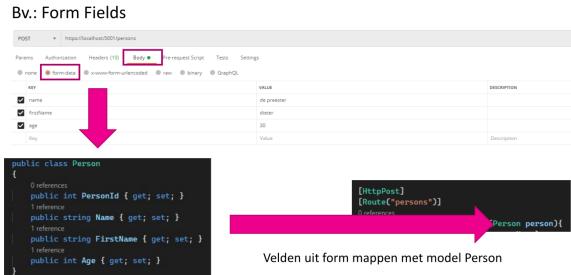
- OkObjectResult(data)
- Ok()
- NotFoundObjectResult(data)
- NotFound()
- StatusCodeResult(int statuscode)

1.8 Model binding

- Default enkel bij HTTP Post ⇒

- ASP.NET Core volgorde voor ophalen van key/values:
 - Form fields
 - Request body (enkel bij gebruik [ApiController] attribuut)
 - Route
 - Querystring

1.8.1 Voorbeelden



Figuur 17: Voorbeeld: Form Fields



Figuur 18: Voorbeeld: Request Body (enkel bij gebruik van [ApiController] attribuut)

Via Route attribuut kunnen we parameters doorgeven:

```

[HttpPost]
[Route("persons/{name}/{firstName}/{age}")]
  
```

Figuur 19: We bepalen de parameter namen tussen de {parameter naam}

```

[HttpPost]
[Route("persons/{name}/{firstName}/{age}")]
public ActionResult AddPersonRoute(string name, string firstName, int age){
  
```

Figuur 20: In de C# functie voorzien we dezelfde naam als in de route

```
[HttpPost]
[Route("persons/{name}/{firstName}/{age}")]
0 references
public ActionResult AddPersonRoute([FromRoute]Person person){
```

Figuur 21: We kunnen ook de waarden uit de route direct opslaan in het object: via **[FromRoute]** attribuut



Figuur 22: Kan ook via Querystring

1.9 Configuration

- Bepaalde info willen we niet hardcoden
 - Mailserver, connectionstring, password, ...
- We moeten dit buiten de applicatie kunnen opslaan (appsettings.json, Azure Keyvault)

1.9.1 appsettings.json

- JSON file die bij de applicatie zit
- Verschillende versies mogelijk
 - Development/Testing/Production..
- Eerst geladen
- Bevat dingen die zowel voor development als production geldig zijn
- appsettings.Development.json (naargelang de omgeving inladen)
 - Overriden wat reeds in appsettings.json zit
- JSON file

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "MailServerSettings": {
    "ServerName" : "server",
    "UserName" : "test-mailer",
    "Password" : "test-passwrd"
  }
}
```

Figuur 23: Probeer met Secties te werken (zie MailServerSettings)

- We gebruiken Option pattern voor inladen data uit configuratie files
- Eerst klasse maken die overeenkomt met de waarden in de JSON file

```
public class MailServerSettings
{
    0 references
    public string ServerName { get; set; }
    0 references
    public string UserName { get; set; }
    0 references
    public string Password { get; set; }
}
```

Figuur 24: Klasse die overeenkomt met het MailServerSettings object in appsettings.json

- Registreren van de sectie in de startup en koppelen aan de klasse
- Hierdoor krijgen we een strongly typed settings file

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<MailServerSettings>(Configuration.GetSection("MailServerSettings"));
}
```

Figuur 25: Configuration.GetSection("SectionNaam")

Gebruik in de controller:

- IOptions<MailServerSettings> via CONSTRUCTOR (CTOR) binnenbrengen
- Value property zal deze uitlezen en opslaan in property _mailserverSettings
- Hierdoor kunnen we deze vlot uitlezen in de controller en gebruiken

```
private readonly MailServerSettings _mailserverSettings;
0 references
public DemoController(IOptions<MailServerSettings> options, ILogger<DemoController> logger)
{
    _mailserverSettings = options.Value;
    _logger = logger;
}
```

Figuur 26

In VSCode:

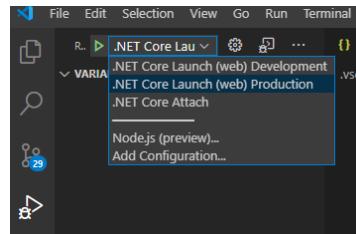
- launch.json openen
- Hier kan je de profielen vinden
- Copy/paste profile
- Per profiel kan je instellingen maken zoals URL of ASPNETCORE_ENVIRONMENT

```

    "configurations": [
      {
        "name": ".NET Core Launch (web) Development",
        "type": "coreclr",
        "request": "launch",
        "preLaunchTask": "build",
        "program": "${workspaceFolder}/bin/Debug/net5.0/backend-theorie01-deon.dll",
        "args": [],
        "cwd": "${workspaceFolder}",
        "stopOnEntry": false,
        "serverReadyAction": {
          "action": "openExternalUrl",
          "url": "http://localhost:5001"
        },
        "patterns": "\\\\[\\w]+ listening on:\\\\https://\\\\\\$"
      },
      {
        "name": ".NET Core Launch (web) Production",
        "type": "coreclr",
        "request": "launch",
        "preLaunchTask": "build",
        "program": "${workspaceFolder}/bin/Debug/net5.0/backend-theorie01-deon.dll",
        "args": [],
        "cwd": "${workspaceFolder}",
        "stopOnEntry": false,
        "serverReadyAction": {
          "action": "openExternalUrl",
          "url": "http://localhost:5001"
        },
        "patterns": "\\\\[\\w]+ listening on:\\\\https://\\\\\\$"
      }
    ]
  }
}

```

Figuur 27: launch.json



Figuur 28: In de dropdown kies je wat je wil starten

Visual Studio 2019

- We bepalen de profielen in de map Properties
- In launchSettings.json

1.9.2 Azure Keyvault

- Kluis in Azure waar alle info zit
- Beste oplossing voor productie projecten

1.10 Wat moet je kennen?

- Zorg dat je de HTTP Request/Response kan uitleggen
- Wat zijn statuscodes en wanneer gebruik ik welke
- Een duidelijke API URL kunnen opstellen
- Manieren van modelbinding
- Hoe configureren je omgeving met profielen

2 Web API DTO, Validation, Versioning

2.1 Validation

- Altijd valideren:

- In client applicatie (Xamarin, React, Vue, Angular)
- In de backend
- We gaan er altijd vanuit dat de binnenkomende data foutief of ongeldig kan zijn
- Vertrouw niemand
- WebAPI bevat ingebouwd systeem om validatie te doen

2.1.1 Attributen

```
public class Customer
{
    public Guid CustomerId { get; set; }
    [Required(ErrorMessage="Verplicht")]
    [MaxLength(50)]
    public string Name { get; set; }
    [Required]
    [MaxLength(20)]
    public string FirstName { get; set; }
    [Range(0,120)]
    public int Age { get; set; }
    [Required(ErrorMessage="E-mail Verplicht")]
    [EmailAddress(ErrorMessage="Ongeldige e-mailadres")]
    public string Email { get; set; }
}
```

Figuur 29: Validation attributes bij de properties

- Attributen op de properties van het model
- Optioneel foutmelding meegeven aan attribuut, anders default waarde
- **[ApiController]** attribuut zorgt voor automatische validatie en HTTP Statuscode 400 indien niet OK

```
[ApiController]
[Route("api")]
public class CustomerController : ControllerBase
```

Figuur 30: ApiController boven de Controller-klasse

JSON Data	Model met verplicht veld	Resultaat
<pre>{ ... "age" : 30 }</pre>	<pre>public class Person { [Required] 0 references public string Name { get; set; } 0 references public int Age { get; set; } }</pre>	<p>Body Cookies Headers (4) Test Results</p> <p>Pretty Raw Preview Visualize JSON ↗</p> <pre>1 { 2 "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1", 3 "title": "One or more validation errors occurred.", 4 "status": 400, 5 "traceId": "00-7fd93c498f79ab408751f958359354ee-1cddd0a7b1bd942-00", 6 "errors": [7 {"Name": [8 "The Name field is required." 9] 10 } 11]</pre>

Figuur 31: Je kan de error name veranderen: [Required(ErrorMessage="Sorry verplicht")]

2.1.2 Ingebouwde attributen:

- [CreditCard] = creditcard formaat

- [Compare] = 2 properties moeten gelijk zijn in een model
- [EmailAddress] = emailadres
- [Phone] = telefoonnummer
- [Range] = property value moet in een bepaald bereik zitten
- [RegularExpression]
- [Required] = vereist, niet leeg
- [StringLength] = maximum lengte
- [Url] = moet in URL-formaat zijn

2.1.3 Custom Validation Attributes

- Zelf een Attribute maken: klasse erft van ValidationAttribute
- Override van methode ValidationResult
- In deze methode schrijven we de controle code
- Gebruiken in model [CustomerTypeAttribute]

<https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-5.0>

```
0 references
public class CustomerTypeAttribute : ValidationAttribute
{
    0 references
    public CustomerTypeAttribute()
    {

    }

    0 references
    public string GetErrorMessage() => $"Invalid customer type";

    0 references
    protected override ValidationResult IsValid(object value, ValidationContext validationContext)
    {
        var customer = (Customer)validationContext.ObjectInstance;

        if ((customer.CustomerType == "Privé") || (customer.CustomerType == "Business"))
        {
            return ValidationResult.Success;
        }
        return new ValidationResult($"Invalid customer type");
    }
}

[CustomerTypeAttribute]
public string CustomerType { get; set; }
```

Figuur 32

2.2 QueryString

= parameter in de url

- Handig als we op verschillende manieren data wensen terug te keren zonder meerdere endpoints te maken
- **QueryString mogen nooit verplicht zijn**

- Gebruiken we vooral bij:
 - Meegeven hoe we wensen te sorteren ?sort=asc
 - Lets zoeken vb: ?q=zoekterm
 - Lets in bepaalde taal opvragen vb: ?language=nl-BE
 - Lets al dan niet includen of excluden
 - Filters

2.2.1 Werking

- Waarde als parameter van de functie maar met default waarde vb: includeAddress
- IncludeAddress zit niet in de URL enkel CustomerId
- Geven we querystring niet mee dan zal includeAddress gelijk zijn aan false

```
[HttpGet]
[Route("customer/{customerId}")]
public ActionResult<Customer> GetCustomerByCustomerId(Guid customerId, bool includeAddress = true)
{
    var customer = _customers.Where(c => c.CustomerId == customerId).SingleOrDefault();
    if (includeAddress == false){
        customer.Address = null;
    }
    return new OkObjectResult(customer);
}
```

Figuur 33

2.3 DTO

- = Data Transfer Object
- We kennen reeds models:
 - POST = Omzetten van JSON naar C# object
 - GET = Omzetten van C# naar JSON object
- Soms willen we niet alle data terugkeren: alleen terugsturen wat echt nodig is
- We voegen DTO klasse toe ⇒ Alleen die properties sturen we terug
- Veel werk: alles manueel overzetten, bij nieuwe properties code aanpassen op verschillende plaatsen



Figuur 34: DTO moeten we manueel aanmaken en properties kopiëren

Oplossing:

- een **mapper** gebruiken
- Nuget package die automatisch properties zal kopiëren tussen objecten
- Meest gebruikte in .NET wereld is **Automapper**
- <https://github.com/AutoMapper/AutoMapper>

2.3.1 Werking

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddAutoMapper(typeof(Startup));
}

```

Figuur 35: In StartUp.cs: registreren van de AutoMapper service in WebAPI

```
public class CustomerDTO
{
    0 references
    public Guid CustomerId { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public string FirstName { get; set; }

}
```

Figuur 36: Klasse toevoegen in DTO-map, bv: CustomerDTO

```
public class AutoMapping : Profile
{
    0 references
    public AutoMapping()
    {
        CreateMap<Customer, CustomerDTO>();
    }
}
```

Figuur 37: DTO profile toevoegen, bv: AutoMapping.cs

In de AutoMapping klasse definiëren we de mapping tussen Customer en CustomerDTO

```
private readonly IMapper _mapper;
public CustomerController(IMapper mapper, ILogger<CustomerController> logger)
{
    _mapper = mapper;
    _logger = logger;
}
```

Figuur 38: We voeren de mapping uit door de automapper te injecteren in de controller (=dependency Injection)

```
[HttpGet]
[Route("customers")]
public List<CustomerDTO> GetCustomers()
{
    return _mapper.Map<List<CustomerDTO>>(_customers);
}
```

Figuur 39: We passen de return value aan de controller en keren lijst van CustomerDTO terug. Via de Map-functie van de mapper zetten we deze om.

2.4 Versioning

- Software veranderingen zijn er altijd, ook als software in productie is
- We moeten verschillende versies kunnen bouwen van onze API
- We moeten er rekening mee houden dat niet iedere client direct de nieuwe versie zal gebruiken
⇒ oude versie blijven ondersteunen

- Hoe pakken we dit aan ?
- **Microsoft.AspNetCore.Mvc.Versioning** is de extra nuget package die we toevoegen

2.4.1 Registreren bij services

```
services.AddControllers();
services.AddApiVersioning();
```

Figuur 40: Eerst registreren bij de services: **AddApiVersioning()**



Figuur 41: Also we nu naar de endpoint surfen krijgen we een foutmelding dat we geen versie hebben meegegeven

2.4.2 Configuratie

- We kunnen een default API versie instellen: **DefaultApiVersion**
- **AssumeDefaultVersionWhenUnspecified**: als er geen versie werd meegegeven, we veronderstellen dat dit default is
- We stellen ook in dat WebAPI de versie moet melden: **ReportApiVersion**

```
services.AddApiVersioning(config =>
{
    config.DefaultApiVersion = new ApiVersion(1, 0);
    config.AssumeDefaultVersionWhenUnspecified = true;
    config.ReportApiVersions = true;
});
```

Figuur 42

2.4.3 Controller attributes

- We stellen versies in op Controller niveau [ApiController]
- Verschillende versies zijn mogelijk
- Via [Deprecated] kunnen we melden dat een API zal verdwijnen op termijn
- Response header zal api-deprecated-versions terugkeren



Figuur 43

2.4.4 Endpoints mappen aan een specifieke versie

```
[HttpGet]
[Route("customers")]
[MapToApiVersion("1.0")]
public List<Customer> GetCustomers()
{
    return _customers;
}

[HttpGet]
[Route("customers")]
[MapToApiVersion("2.0")]
public List<CustomerDTO> GetCustomersDTO()
{
    return _mapper.Map<List<CustomerDTO>>(_customers);
}
```

Figuur 44: [MapToApiVersion]

2.4.5 Testen

The screenshot shows the 'Headers' tab in Postman. It lists several HTTP headers with their values and descriptions:

HEADER	VALUE	DESCRIPTION
Cache-Control	no-cache	
Postman-Token	calculated when request is sent	
Host	calculated when request is sent	
User-Agent	PostmanRuntime/7.26.8	
Accept	*/*	
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
api-version	1.0	

Figuur 45: Via de HTTP header geven we de versie door

2.5 Binary bestanden

- Naast JSON data moeten we soms ook met binary data werken
- Word, PDF, afbeeldingen, CSV ...
- We moeten deze kunnen opladen naar een Web API
- We slaan deze dan op
 - File system van de server (niet ok voor productie, wel om te testen)
 - Blob Storage system op Azure (ok voor productie)

2.5.1 Werking

- We gebruiken HTTP Post
- List<IFromFile> bevat de bestanden
- Via for lus overlopen en wegschrijven naar C:\Temp\test.csv
- System.IO.File zorgt voor aanmaken file en wegschrijven data

```

[HttpPost]
[Route("customer/logo")]
public async Task<IActionResult> UploadImage(List<IFormFile> files)
{
    long size = files.Sum(f => f.Length);

    foreach (var formFile in files)
    {
        if (formFile.Length > 0)
        {
            var filePath = @"C:\temp\test.csv";

            using (var stream = System.IO.File.Create(filePath))
            {
                await formFile.CopyToAsync(stream);
            }
        }
    }

    return Ok(new { count = files.Count, size });
}

```

Figuur 46: Werken met binary files

2.5.2 Testen

```

[HttpPost]
[Route("customer/logo")]
public async Task<IActionResult> UploadImage(List<IFormFile> files)
{
    long size = files.Sum(f => f.Length);
}

```

Figuur 47: Testen via postman

2.5.3 Bestanden downloaden

- We keren een FileResult terug
- Deze bevat:
 - Stream met data
 - mimeType = beschrijving file
 - De naam van de file

```

[HttpGet]
[Route("customer/logo")]
public async Task<FileResult> DownloadFile()
{
    var byteArray = await System.IO.File.ReadAllBytesAsync(@"C:\temp\test.csv");
    string mimeType = "application/csv";
    Stream stream = new MemoryStream(byteArray);
    return new FileStreamResult(stream, mimeType)
    {
        FileDownloadName = "test.csv"
    };
}

```

Figuur 48: Downloading files

2.6 CSV files

= Comma-Sepreated Values

- Veel gebruikt formaat zowel in wereld van AI, Administratieve apps,...
- Veel software kan overweg met dit formaat bv.: Excel (import/export)
- Is flat tekst formaat met een separator (komma, puntkomma, dubbelpunt, ...)
- Separator zal de data scheiden van elkaar
- Bevat meestal header, eerste rij in file
- In de wereld van AI veel gebruikt als formaat voor datasets
- Handig als we files kunnen genereren via Web API

2.6.1 CSV Helper

- = Gratis en open source Nuget package voor lezen en schrijven van CSV files
- Goede documentatie en voorbeelden aanwezig
- <https://joshclose.github.io/CsvHelper/>

2.6.2 Werking

- StreamWriter zal file wegschrijven
- CsvWriter zal CSV inhoud genereren
- Daarna zelfde procedure voor downloaden als andere file
- We slaan dus file eerst lokaal op, op de server

```
[HttpGet]
[Route("customer/export")]
public async Task<FileResult> Export()
{
    using (var writer = new StreamWriter(@"C:\temp\export.csv"))
    using (var csv = new CsvWriter(writer, CultureInfo.InvariantCulture))
    {
        csv.WriteRecords(_customers);
    }
    var byteArray = await System.IO.File.ReadAllBytesAsync(@"C:\temp\export.csv");
    string mimeType = "application/csv";
    Stream stream = new MemoryStream(byteArray);
    return new FileStreamResult(stream, mimeType)
    {
        FileDownloadName = "export.csv"
    };
}
```

Figuur 49: Voorbeeld CsvWriter

2.7 Caching

- Applicatie sneller en schaalbaar maken
- We lezen niet altijd alle data uit de database
 - Response is sneller ⇒ database access (I/O in het algemeen is traag)
 - Bv: lijst van landen: deze lijst zal niet veel wijzigen
- 2 manieren in WebAPI:
 1. Response caching

2. Output caching

2.7.1 Response Caching (HTTP Caching)

- HTTP Standaard RFC 7234: <https://tools.ietf.org/html/rfc7234>
- Hier gaan we de server zo weinig mogelijk inspannig vragen om een resultaat terug te keren
- We sturen de caching via de HTTP Headers
- Response caching zal niks opslaan op de server
 - We slaan op bij de client of op proxy servers in het netwerk
 - Server zal **Cache-Control** header toevoegen in response
 - Private: client
 - Public: op een proxy server

Key	Value
Date	Mon, 11 Jan 2021 13:40:40 GMT
Content-Type	text/plain; charset=utf-8
Server	Kestrel
Cache-Control	private,max-age=15
Transfer-Encoding	chunked

Figuur 50: Cache-Control in HTTP headers

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMemoryCache();
    services.AddResponseCaching();
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "backend_theorie02_demo", Version = "v1" });
    });
}
```

Figuur 51: In StartUp.cs moeten we een paar services registreren: services.AddResponseCaching()

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseSwagger();
        app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "backend_theorie02_demo v1"));
    }

    app.UseHttpsRedirection();
    app.UseRouting();
    app.UseAuthorization();
    app.UseResponseCaching();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Figuur 52: **app.UseResponseCaching()** toevoegen op de juiste plaats!

```
[HttpGet]
[Route("timestamp2")]
[ResponseCache(Duration = 15, Location = ResponseCacheLocation.Any)]
0 references
public ActionResult GetDateTime2()
{
    return Ok(LongTimeOperation());
}
```

Figuur 53: Attribute [ResponseCache] toevoegen

ResponseCache attribuut

- Duration in seconden
- Location: waar slaan we de cache op
 - Any = public
 - Client = private, geen caching op proxy
 - None = geen cache

```
[HttpGet]
[Route("timestamp2")]
[ResponseCache(Duration = 15, Location = ResponseCacheLocation.Any)]
0 references
public ActionResult GetDateTime2(){
    return Ok(LongTimeOperation());
}
```

Figuur 54

- <https://www.codeproject.com/Articles/1111260/Response-Caching-and-In-Memory-Caching-in-ASP-NET>
- <https://www.keycdn.com/blog/http-cache-headers>

2.7.2 Output Caching

- We gaan resultaat bijhouden in een cache aan de server kant
- Memory intensief
- Schaalt niet altijd goed, zeker bij meerdere servers
- Oplossingen
 - IMemoryCache, standaard in .NET core, eenvoudige maar geen scaling
 - Redis
 - * Schaalt wel, maar duurder
 - * Extra service op Azure
 - * Zelf hosten als Docker container

```
services.AddMemoryCache();
```

```
private IMemoryCache _cache;
8 references
private static List<Person> _persons;

1 reference
private readonly ILogger<DemoController> _logger;
2 references
private readonly IMapper _mapper;

0 references
public DemoController(IMapper mapper, ILogger<DemoController> logger, IMemoryCache memoryCache)
{
    _mapper = mapper;
    _logger = logger;
    _cache = memoryCache;
```

Figuur 55: IMemoryCache via services activeren in ConfigureServices

- In de methode controleren via TryGetValue of de waarde aanwezig is

- Indien niet aanwezig: toevoegen via Set met als laatste parameter hoelang de waarde in de cache moet blijven

```
[HttpGet]
[Route("timestamp2")]
0 references
public ActionResult GetDateTime(){
    DateTime timestamp;
    if (!_cache.TryGetValue<DateTime>("timestamp", out timestamp))
    {
        timestamp = DateTime.Now;
        _cache.Set<DateTime>("timestamp", timestamp, DateTime.Now.AddSeconds(5));
    }
    return Ok(timestamp);
}
```

Figuur 56

2.8 Wat moet je kennen?

- Wat is AutoMapper en hoe kan je dit gebruiken?
- Hoe werkt versioning in API's?
- Hoe kan je model validation doen?
- Wat is caching, waarom gebruiken we dit?
- Wat zijn de verschillende soorten caching in Web API?
- Hoe kan je best CSV bestanden verwerken?
- Hoe kan je binare bestanden opladen?
- Wanneer gebruiken we best querystrings en hoe?

3 Docker & Docker-compose

3.1 Waarom?

- Docker wil het mogelijk maken om software op ieder systeem te krijgen
- "If it runs on my computer, it will run on your computer"
- Microservices
- DevOps
- Resource usage

3.2 What?

- Docker != container
- Docker = ecosysteem voor het creëren en draaien van containers
 - Docker CLI
 - Docker Engine

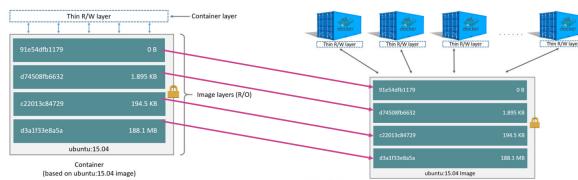
- Docker Image
- Docker Container
- Docker Hub
- Docker Compose
- Docker Swarm

3.3 Docker images

- = een snapshot van een filesystem
- Heeft een startup commando: executable die iets zal uitvoeren
- Heeft een gelaagde structuur ('layered structure')

Instantie van een image = container

3.3.1 Image layers



Figuur 57: Image layers

- De RUN, COPY, ADD commandos zullen allemaal een nieuwe read-only layer maken
- Top layer = container layer = de 'writeable' layer
- Als je een container delete, zal alleen de container layer gedeleteet worden
 - Image zal blijven bestaan
 - Om de data te behouden: gebruik persistente volumes (zie later)

3.4 Lightweight

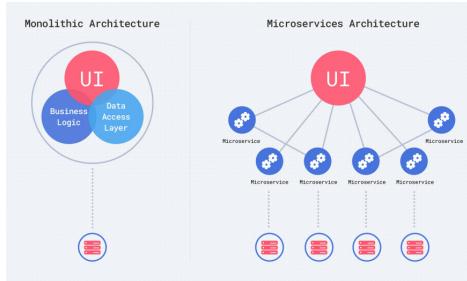
Docker images zijn heel klein in disk size, omdat:

- De kernel van de host wordt gedeeld met Docker
- De container heeft geen OS nodig
- Om nog minder disk space te gebruiken ⇒ layers delen
- Er zijn kleine community images:
 - Alpine Linux

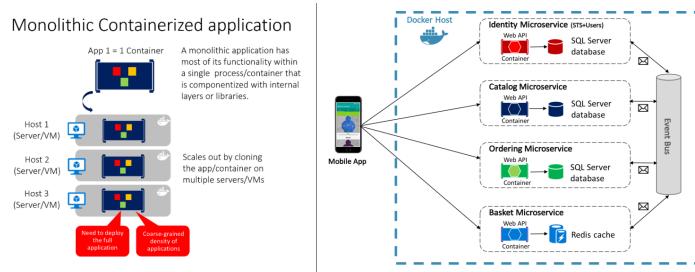
3.5 Microservices en Docker

Definitie 3.1 *Microservices zijn een software development techniek die ervoor zorgen dat de structuur van onze applicatie losgekoppeld wordt in verschillende kleine services die gekoppeld zijn aan elkaar*

- Lightweight
- Omgekeerde = monolithic architecture
 - 1 server
 - UI, Backend, Data Access Layer zit samen



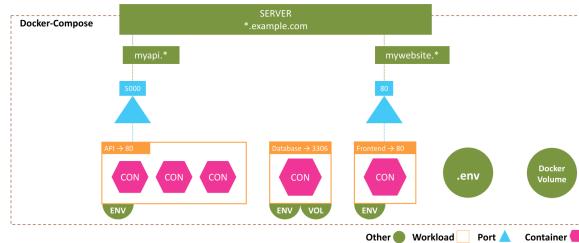
Figuur 58: Bij microservices: elke service heeft zijn eigen server



Figuur 59: Links: containerized, monolithische applicatie. Rechts: Containerized microservices

Microservices != containerization (maar het is wel een logische stap)

3.5.1 Demo



Figuur 60: Een voorbeeld van een app die gebruik maakt van microservices en containers

3.6 Docker basiscommando's

3.6.1 Images ophalen

- Docker Hub
- GCP Container Registry
- AWS Elastic Container Registry (ECR)
- Azure Container Registry (ACR)
- 'Bring Your Own Registry' (binnekort bij Howest!)
- Er zijn ook private registries en repositories, waar je inloggegevens voor nodig hebt
- Standaard = Docker Hub

Zoeken naar images

- hub.docker.com
- GitHub
- ...
- docker search <keyword>

Downloaden van images

```
1 docker pull <image-naam>
2
3 # voorbeelden:
4 docker pull elasticsearch
5 docker pull nathansegers/some-custom-image
6 docker pull python:3.7-alpine
```

- Image tagging: een specifieke versie
- default = latest
- Belangrijk voor versioning (APIs)!

3.6.2 Containers runnen

```
1 # run een docker image
2 docker run <image-name> <alternative command>
3
4 docker create <image-name>
5 docker start <container-id>
6
7 # stop een container
8 docker stop (graceful) / docker kill (immediate)
9 # toon de huidige containers (--all toont ook de gestopte containers)
10 docker ps (--all)
```

3.6.3 Interactie & logs

⇒ geen GUI!

```
1 # interactieve terminal:  
2 docker exec -it <container-id> <command>  
3  
4 # toon logs van een container  
5 docker logs <container-id>  
6  
7 # verwijder alle gestopte containers, netwerken, images, build cache:  
8 docker system prune
```

3.6.4 Containers builden

```
1 # vergeet het puntje niet! dit betekent: onze huidige map  
2 docker build .  
3  
4 # maak een image uit een bestaande container  
5 docker commit -c <startup-command>
```

3.7 Dockerfile

= Een bestand met commando's om een image te bouwen.

- Volgorde is belangrijk!
- Gebruikt cache wanneer mogelijk (snellere builds)
-

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 as build  
COPY . .  
RUN dotnet publish -c release -o /app  
WORKDIR /app  
ENTRYPOINT ["dotnet", "mydotnetapp.dll"]
```

Figuur 61: Voorbeeld Dockerfile

- FROM: build image
- COPY: bestanden die we nodig hebben kopiëren
- RUN: commando's uitvoeren, programma's installeren
- WORKDIR: verander de huidige map
- ENTRYPOINT: startupcommando, met argumenten

3.7.1 Dockerfile optimalisatie

- Herinner u: RUN, COPY, ADD voegt een nieuwe layer toe
- Wanneer we images bouwen, zullen de layers cache gebruikt worden
- We hebben in het vorig voorbeeld slechts 1 COPY commando gebruikt

- Stel dat we enkel de broncode van 1 bestand veranderd hebben:
 - Dependencies zijn onveranderd
 - Dependencies zullen toch opnieuw geïnstalleerd worden, omdat de laag veranderd is

Beter:

```

1 # packages installeren: als deze laag niet veranderd is, zal hij cache gebruiken
2 COPY *.csproj .
3 RUN dotnet restore

4
5 # pas daarna gaan we alle andere files kopiëren naar onze source map
6 COPY .
7 # dan publiceren
8 RUN dotnet publish -c release -o /app

```

3.8 Docker Compose

= meerdere containers runnen

- In plaats van een dockercommando met enorm veel parameters:
 - docker build .
 - docker run -p 3000:3000 -v /app/node_modules -v \$(pwd) app ca6cc440a6ef
- Gebruiken we een .yaml bestand:
 - docker-compose up –build

```

1 version: "3"
2 services:
3   web:
4     build: .
5     ports:
6       - "3000:3000"
7     volumes:
8       - /app/node_modules
9       - .:/app

```

- docker-compose.yml maakt het makkelijker om meerdere containers te draaien
- docker-compose.yml maakt het mogelijk om relaties tussen containers te leggen

```

version: '3.8'
services:
  api:
    build:
      context: .
    ports:
      - 5000:80
      - 5001:443
    environment:
      - ASPNETCORE_URLS=https://+;http://+
      - ASPNETCORE_HTTPS_PORT=5001
      - ASPNETCORE_ENVIRONMENT=Development
    volumes:
      - .aspnet/https:/https/
  db:
    image: mariadb:10.5.8
    env_file:
      - .env
    volumes:
      - ./db:/docker-entrypoint-initdb.d
      - database:/var/lib/mysql

```

Figuur 62: Voorbeeld docker-compose.yml

```

1 # docker compose opstarten (of eerst builden en dan opstarten):
2 docker-compose up (--build)
3
4 # stoppen
5 docker-compose down

```

3.8.1 Docker Compose terminologie

```

api:
  image: nathansegers/custom-api:latest
  build: ./web/api
  ports:
    - 5000:80
    - 5001:443
  volumes:
    - ./shared_data/:/mnt/data/shared
  env_file:
    - .env
  depends_on:
    - db

```

Figuur 63: Voorbeeld

In een docker-compose.yml bestand vind je volgende termen:

- naam van de service
 - In bovenstaand voorbeeld: ‘api’
- image

- de image die we gebruiken voor deze service:
- nathansegers/custom-api:latest
- build
 - In welke map we gaan builden
 - Relatief pad, ten opzichte van de context van het docker-compose.yml bestand
- expose vs ports
 - expose:
 - * Opent een poort **binnen** het Docker netwerk
 - * Niet beschikbaar buiten docker
 - * Een applicatie die intern op poort 3306 draait, zal niet beschikbaar zijn op localhost:3306
 - * Geeft wel toegang aan andere services in hetzelfde netwerk
 - ports
 - * = interne port mapping
 - * Een applicatie die op poort 80 draait, zal beschikbaar zijn via localhost:5000
- networks
 - We kunnen docker containers een netwerk geven zodat ze met elkaar kunnen communiceren
 - Alle services zitten standaard in 1 netwerk
- volumes
 - Om bestanden te injecteren in een docker container
 - Persistente storage
 - Kan gedeeld worden met meerdere containers
 - 2 manieren:
 1. Named volumes
 2. Relatieve of absolute paden

```

services:
  api:
    volumes:
      - ./shared_data:/mnt/data/shared
      - named_volume:/mnt/persistent
volumes:
  named_volume:

```

Figuur 64: Relatieve/absolute paden (boven) vs named volume (beneden)

- environments (.env)
 - environment variables komen in dit bestand:
 - connectionstrings
 - ports

- databasenamen
- ...
- depends_on
 - Start de service na deze service(s)
 - Hier: start de api nadat de db service is opgestart
 - Dit is nodig als een service iets nodig heeft die een andere service eerst moet opstarten

3.9 Wat moet je kennen?

- Wat is docker? Waarom is het relevant voor ons?
- Wat zijn microservices?
- Hoe Dockerfiles schrijven
- De verschillende docker-compose termen
- Hoe communiceren tussen services in een Docker Network

4 EF Core

4.1 Probleemstelling

- We gebruiken databases voor opslag van data
- Relationale databases is de standaard, meest gebruikt
- We gebruiken SQL in relationele databases voor INSERT, UPDATE, DELETE, ...
- NoSQL databases worden nu meer en meer gebruikt:
 - MongoDB
 - CosmosDB
 - ElasticSearch

4.1.1 **Niets fout met SQL en relationele databases!**

- Voordelen:
 - SQL statements werken zeer snel
 - Weinig nieuwe kennis nodig
- Nadelen:
 - Soms lastig om te onderhouden
 - We schrijven SQL code **in** de C# code
 - Fouten zien we pas tijdens uitvoeren applicatie
- Wanneer we SQL en NoSQL gebruiken binnen 1 applicatie ⇒ meerdere APIs nodig

4.1.2 Toch gebruiken we nog plain SQL

- Snelheid is nog altijd de beste, zeker als we stored procedures gebruiken die in de database draaien
- Super complexe queries en joins beter via SQL of Stored Procedure
- Bestaande toepassingen migreren naar EF Core is niet eenvoudig, deels migreren ook niet altijd

4.2 EF Core

= Entity Framework Core

- Object Relationale Mapper (ORM)
- We mappen models (klassen) met tabellen in de database
- Via **LINQ** statements kunnen we operaties uitvoeren op de data
- We praten met de database via de **EF Context** ⇒ speciale klasse die zorgt voor communicatie met de database



Figuur 65: Database-tabel mappen naar een klasse

2 manieren van werken:

- Database First
- Code first

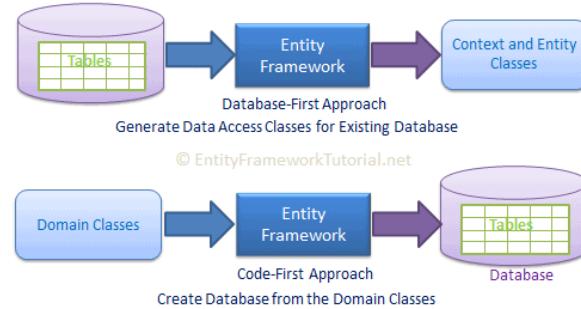
4.2.1 Database First

- We vertrekken van een bestaande database
- EF Core zal onze klassen genereren
- Vooral om bestaande toepassingen te moderniseren

4.2.2 Code First

- Wij maken klassen (modellen)
- EF Core zal de database maken

Wij zien enkel Code First



Figuur 66: Database-First vs Code-First

4.3 EF Core installatie

We moeten een **EF Core Provider** toevoegen aan het project:

- Zorgt voor een onderliggende communicatie met de database
- Per database is er een provider
 - Microsoft.EntityFrameworkCore.SqlServer (Azure)
 - Microsoft.EntityFrameworkCore.Sqlite (Lokaal SQL Lite)
 - Pomelo.EntityFrameworkCore.MySql (MySQL)
 - CosmosDB ⇒ NoSQL Support
 - <https://docs.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli>

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="5.0.1" NoWarn="NU1605" />
    <PackageReference Include="Microsoft.AspNetCore.Authentication.OpenIdConnect" Version="5.0.1" NoWarn="NU1605" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="5.0.1">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="5.0.1" />
    <PackageReference Include="Swashbuckle.AspNetCore" Version="5.6.3" />
  </ItemGroup>
</Project>
```

Figuur 67

Toevoegen kan op 2 manieren:

- VSCode > Command Palette > nu command > <provider naam>
- Via commandline in de map van je project
 - dotnet add package Microsoft.EntityFrameworkCore.SqlServer

Daarnaast moeten ook de EF tools op je systeem staan: dotnet tool install –global dotnet-ef

4.4 Models

Dit zijn klassen die onze database tables voorstellen. Een ID bepalen we door:

- Het veld Id te noemen
- Naam van de klasse + Id \Rightarrow BlogId
- Attribuut [Key]
- Dit zal autonummer worden in de database

The image shows two side-by-side code snippets. The left snippet defines a `Blog` class with a `BlogId` attribute and a `Post` class with a `PostId` attribute. Both attributes are highlighted with a pink rectangle. The right snippet shows the same classes, but the `BlogId` attribute in the `Blog` class is annotated with the `[Key]` attribute, also highlighted with a pink rectangle.

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

```
public class Blog
{
    public int Id { get; set; }
    public string Url { get; set; }
    public int Rating { get; set; }
    public List<Post> Posts { get; set; }
}

public class Blog
{
    [Key]
    public int BlogNumber { get; set; }
    public string Url { get; set; }
    public int Rating { get; set; }
    public List<Post> Posts { get; set; }
}
```

Figuur 68: 3 manieren om een uniek ID aan te duiden

4.4.1 Eén-op-veel-relaties

The image shows a code snippet for a one-to-many relationship. The `Blog` class has a `Posts` navigation property, which is highlighted with a pink rectangle. The `Post` class has a `Blog` foreign key, also highlighted with a pink rectangle. A large pink arrow points from the `Blog` class towards the `Post` class, indicating the relationship. The word "Navigation property" is written below the arrow.

```
public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public List<Post> Posts { get; } = new List<Post>();
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```



Navigation property

Figuur 69: Eén-op-veel-relaties

- Blog bevat één of meerdere posts \Rightarrow `List<Post>`

- Post kan slechts in één Blog voorkomen
- In de Post moeten we BlogId definiëren en kunnen we ook het Blog object zelf definiëren
- Dit noemen we een Navigation Property

4.4.2 Enkele attributen

```

1 // Kolom zal in de database blog_id noemen en niet BlogId
2 [Column("blog_id")]
3
4 // verplicht veld ook op database niveau
5 [Required]
6
7 // Provider zal dit meenemen bij database generatie,
8 // veld in database zal ook max 500 karakters (nvarchar(500) zijn.
9 // Anders is het nvarchar(max)
10 [MaxLength]
11
12 // Provider zal dit instellen in database als decimal (5,2)
13 [Column(TypeName = "decimal(5,2)")]
14
15 //
16 [Comment("database commentaar")]

```

4.4.3 Fluent API

- We kunnen alle attributen ook via de Fluent API instellen
- Fluent API zijn functies die we kunnen aanroepen binnen OnModelCreating
- Veel gebruikt voor configuratieinstallingen
- Ook enige optie als we **geen toegang** hebben tot de source code van de modellen:
 - Dan kan je **geen attributen** plaatsen
 - Enkel Fluent API mogelijk

```

modelBuilder.Entity<Blog>()
    .Property(b => b.Url)
    .HasComment("The URL of the blog");

modelBuilder.Entity<Blog>()
    .Property(b => b.Score)
    .HasPrecision(14, 2);

modelBuilder.Entity<Blog>()
    .Property(b => b.LastUpdated)
    .HasPrecision(3);

```

Figuur 70: Fluent API

4.5 EF Context

(of DbContext)

- Erft altijd van DbContext
- Klasse is verantwoordelijk voor de communicatie met de database via een sessie die deze opzet met de database
- We bepalen de tabellen, via DbSet

- We bepalen ook welke database we gebruiken via OnConfiguring waar we ConnectionString opgeven
- DbContext zal ook de relaties aanmaken en beheren
- Ook queries uitvoeren tov de database is een taak

```
public class BloggingContext : DbContext
{
    0 references
    public DbSet<Blog> Blogs { get; set; }
    0 references
    public DbSet<Post> Posts { get; set; }

    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlite("Data Source=blogging.db");
}
```

Figuur 71: BloggingContext erft van DbContext

4.5.1 Aanmaken van de database en/of tabellen

- **Migrations** beschrijven in C# code de aanmaak van de tabellen
- Bij iedere update moeten we een nieuw migratie aanmaken
- Er zal een map **Migrations** aangemaakt worden waarin alle veranderingen opgeslagen worden

```
1 # toevoegen van het ef tool om migraties te maken
2 dotnet add package Microsoft.EntityFrameworkCore.Design
3
4 # aanmaken van een migratie
5 dotnet ef migrations add <naam migratie>
6
7 # de aanpassingen uitvoeren:
8 dotnet ef database update
```

- Migratiebestanden bevatten C# code die tabellen zal genereren in de database
- Wijzigt zelf **niks** in deze bestanden
- Bij iedere verandering in C# code (nieuwe tabel, wijziging veld) moeten we een nieuwe migratie toevoegen: dotnet ef migrations add <naam migratie>
- Daarna moeten we de database updaten: dotnet ef database update

4.6 Seeding

- = Opvullen van tabellen met **default** data (landen, postcodes, ...)
- Handig bij testen van software (unit en functional test)
- Seeding is onderdeel van migrations
 - Eerst migrations uitvoeren
 - Dan eventueel data seeding

4.6.1 In C# code

```
0 references
protected override void OnModelCreating(ModelBuilder modelBuilder){

    modelBuilder.Entity<Blog>().HasData(
        new Blog(){
            BlogId = 1,
            Name = "MCT Blog",
            Url = "https://www.mct.be",
        },
        new Blog(){
            BlogId = 2,
            Name = "Wired",
            Url = "https://www.wired.com"
        }
    );

    modelBuilder.Entity<Post>().HasData(
        new Post[]{
            PostId = 1,
            Author = "Johan",
            Title = "VR/AR",
            Content = "VR/AR in het ...",
            BlogId = 1
        }
    );
}
```

Figuur 72: Seeding in C#

- Override de methode OnModelCreating
- Via modelBuilder ⇒ HasData()
 - Toevoegen entities
 - Let op: autonummer moet je **WEL** invullen
- Daarna zal deze code via migraties uitgevoerd worden

4.7 Relations

4.7.1 One to Many

Eén Blog heeft één of meerdere posts

- Blog.BlogId = primary key
- Post.BlogId = foreign key
- Post.Blog = reference navigation property
- Blog.Post = collection navigation property
- Post.Blog = inverse navigation property

Relaties zullen default aangemaakt worden wanneer er navigation properties aanwezig zijn. EF Core zal dit detecteren en default zien als navigation properties

```

public class Blog
{
    5 references
    public int BlogId { get; set; }
    2 references
    public string Name { get; set; }
    2 references
    public string Url { get; set; }
    0 references
    public int Rating { get; set; }
    1 reference
    public DateTime Created { get; set; }
    4 references
    public List<Post> Posts { get; set; }
}

public class Post
{
    1 reference
    public int PostId { get; set; }
    1 reference
    public string Title { get; set; }
    1 reference
    public string Content { get; set; }
    2 references
    public string Author { get; set; }
    [JsonIgnore]
    1 reference
    public int BlogId { get; set; }
    1 reference
    public DateTime Created { get; set; }
    [JsonIgnore]
    0 references
    public Blog Blog { get; set; }
    1 reference
    public List<Comment> Comments { get; set; }
}

```

Figuur 73: One to many

4.7.2 One to One

- Automatische detectie van de navigation properties
- We kunnen dit ook configureren via de Fluent API

```

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }

    public BlogImage BlogImage { get; set; }
}

public class BlogImage
{
    public int BlogImageId { get; set; }
    public byte[] Image { get; set; }
    public string Caption { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}

modelBuilder.Entity<Blog>()
    .HasOne(b => b.BlogImage)
    .WithOne(i => i.Blog)
    .HasForeignKey<BlogImage>(b => b.BlogForeignKey);

```

Figuur 74

4.7.3 Many to Many

- In beide modellen Collection navigation properties
- Automatische generatie van tussentabel

```

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public ICollection<Tag> Tags { get; set; }
}

public class Tag
{
    public string TagId { get; set; }

    public ICollection<Post> Posts { get; set; }
}

```

Figuur 75: Many to Many

4.8 SELECT, INSERT, UPDATE, DELETE

4.8.1 Hoe halen we records op (SELECT)?

```

[HttpGet]
[Route("blogs")]
0 references
public async Task<List<Blog>> GetBlogs(){
    using(BloggingContext context = new BloggingContext()){
        return await context.Blogs.ToListAsync<Blog>();
    }
}

```

Figuur 76: Records ophalen (HttpGet)

1. BloggingContext aanmaken via **using** statement
2. Via context.Blogs table aanspreken
3. **ToToListAsync** zal de query uitvoeren

```

Microsoft.EntityFrameworkCore.Database.Command: Information: Executed DbCommand (10ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT "b"."BlogId", "b"."Rating", "b"."Url"
FROM "Blogs" AS "b"

```

Figuur 77: SQL generatie door EF Core Context

4.8.2 Hoe voegen we een record toe (INSERT)?

```

[HttpPost]
[Route("blogs")]
0 references
public async Task<ActionResult> AddBlog(Blog blog){
    using(BloggingContext context = new BloggingContext()){
        context.Blogs.Add(blog);
        await context.SaveChangesAsync();
        return Ok();
    }
}

```

Figuur 78: Records toevoegen (HttpPost)

1. BloggingContext aanmaken via **using** statement
2. Object toevoegen aan context.Blogs
3. **SaveChangesAsync** aanroepen om weg te schrijven (asynchroon)

```
Microsoft.EntityFrameworkCore.Infrastructure: Information: Entity Framework Core 5.0.1 initialized 'BloggingContext' using provider 'Microsoft.EntityFrameworkCore.Sqlite' with options: None
Microsoft.EntityFrameworkCore.Database.Command: Information: Executed DbCommand (3ms) [Parameters=@p0='?', @p1=? (Size = 10)], CommandType='Text', CommandTimeout='30'
    INSERT INTO "Blogs" ("Rating", "Url")
    VALUES (@p0, @p1);
    SELECT "BlogId"
    FROM "Blogs"
WHERE changes() = 1 AND "rowid" = last_insert_rowid;
Microsoft.EntityFrameworkCore.Database.Command: Information: Executed DbCommand (0ms) [Parameters=@p2='?', @p3=? (Size = 11), @p4=? (Size = 8)], CommandType='Text', CommandTimeout='30'
    INSERT INTO "Posts" ("BlogId", "Content", "Title")
    VALUES (@p2, @p3, @p4);
    SELECT "PostId"
    FROM "Posts"
WHERE changes() = 1 AND "rowid" = last_insert_rowid;
```

Figuur 79: SQL generatie door EF Core Context

4.8.3 Hoe verwijderen we records (DELETE)?

```
[HttpDelete]
[Route("blog/{blogId}")]
0 references
public async Task<ActionResult> DeleteBlog(int blogId){
    using(BloggingContext context = new BloggingContext()){
        var blog = await context.Blogs.Where(b => b.BlogId == blogId).SingleOrDefaultAsync<Blog>();
        if(blog != null){
            context.Blogs.Remove(blog);
            await context.SaveChangesAsync();
        }
        return Ok();
    }
}
```

Figuur 80: Records deleten (HttpDelete)

4.8.4 Hoe halen we records op (SELECT)?

1. BloggingContext aanmaken via **using** statement
2. Eerst blog ophalen via de Linq Where clause
3. We verwijderen de juiste blog uit de Context
4. **SaveChangesAsync** zodat de database op de hoogte is

4.8.5 Toevoegen gerelateerde data (één op veel)

Bv: hoe post toevoegen aan bestaande blog?

1. Eerst blog opzoeken met zijn Posts
2. Via Add methode voegen we een nieuwe Post toe en roepen we **SaveChangesAsync**

```
[HttpPost]
[Route("blog/{blogId}/posts")]
0 references
public async Task<ActionResult> AddPost(int blogId,Post post){
    using(BloggingContext context = new BloggingContext()){
        var blog = await context.Blogs.Include(b => b.Posts).Where(b => b.BlogId == blogId).SingleOrDefaultAsync<Blog>();
        if(blog != null){
            blog.Posts.Add(post);
            await context.SaveChangesAsync();
        }
        return Ok();
    }
}
```

Figuur 81: Gerelateerde data toevoegen (HttpPost met blogId parameter)

- Via de using statement maken we de context aan
- Binnen de {} kunnen we objecten:

- manipuleren en wijzigen,
- toevoegen
- verwijderen
- De context zal deze veranderingen tracken
- Pas als we **SaveChangesAsync()** aanroepen zullen de veranderingen plaatsvinden in de database

4.8.6 Hoe gerelateerde records ophalen

```
[HttpGet]
[Route("blogs")]
0 references
public async Task<List<Blog>> GetBlogs(){
    using(BloggingContext context = new BloggingContext()){
        return await context.Blogs.Include(b => b.Posts).ToListAsync<Blog>();
    }
}
```

Figuur 82: Gerelateerde data ophalen (HttpGet blogs + de posts bij elke blog)

- Via **Include** doen we een Join met een andere table
- Er zal **Exception** ontstaan bij serialisatie

```
System.Text.Json.JsonException: A possible object cycle was detected. This can either be due to a cycle or if the object depth is larger than the maximum allowed depth of 32.
at System.Text.Json.ThrowHelper.ThrowJsonException_InvalidOperationException(CycleDetected(maxDepth))
at System.Text.Json.Serialization.JsonConverter`1.TryWrite(Utf8JsonWriter writer, T& value, JsonSerializerOptions options, WriteStack& state)
```

Figuur 83

- Oplossing: [JsonIgnore] toevoegen zodat serialisatie dit zal negeren

```
2 references
public class Post
{
    0 references
    public int PostId { get; set; }
    0 references
    public string Title { get; set; }
    0 references
    public string Content { get; set; }
    [JsonIgnore]
    0 references
    public int BlogId { get; set; }
    [JsonIgnore]
    0 references
    public Blog Blog { get; set; }
}
```

Figuur 84

```
{
  "blogId": 2,
  "url": "www.mct.be",
  "rating": 4,
  "posts": [
    {
      "postId": 2,
      "title": "hello world",
      "content": "dit is wat tekst"
    }
  ]
},
```

Figuur 85: Resultaat: JSON met voor elke blog alle posts van die blog

4.9 Querying van data

- EF Core gebruikt **LINQ** (Language Integrated Query) om data op te halen uit de database
- We sturen LINQ query via Context naar de onderliggende database provider
- Deze database provider zal zorgen voor de vertaling naar SQL
- Je kan de SQL zien: toevoegen van Logger

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
{
  options.UseLoggerFactory(LoggerFactory.Create(builder => builder.AddDebug()));
  options.UseSqlite("Data Source=blogging.db");
}
```

Figuur 86: Toevoegen van een Logger: in OnConfiguring

4.9.1 Ophalen van alle Blogs *zonder* Posts

```
using(BloggingContext context = new BloggingContext()){
  return await context.Blogs.ToListAsync<Blog>();
}
```

Figuur 87

- Context aanmaken
- Via Tabel Blogs methode ToListAsync() aanroepen
- ToListAsync() zal de query effectief uitvoeren, dit is het moment dat we naar de database gaan

4.9.2 Ophalen van alle Blogs *met* Posts

- Context aanmaken
- Via Tabel Blogs methode Include() aanroepen waar we meegeven welke gerelateerde tabel we willen mee ophalen
- b => b.Posts ⇒ lambda expressie

- Op einde roepen we terug `ToListAsync()` om alles op te halen
- `ToListAsync()` zal de query effectief uitvoeren

```
using(BloggingContext context = new BloggingContext()){
    return await context.Blogs.Include(b => b.Posts).ToListAsync();
}
```

Figuur 88

4.9.3 Ophalen van 1 specifiek item

```
using(BloggingContext context = new BloggingContext()){
    return await context.Blogs.Where(b => b.BlogId == blogId).SingleOrDefaultAsync();
}
```

Figuur 89: Ophalen van 1 specifiek item

- Via Where clause lambda meegeven: `b => b.BlogId == blogId`
- **SingleOrDefaultAsync()**
 - query uitvoeren
 - Single ⇒ 1 resultaat
 - OrDefault ⇒ indien niks gevonden null terugkeren

4.9.4 Voorbeeld: comments van Post ophalen (3 tabellen diep)

```
using(BloggingContext context = new BloggingContext()){
    return await context.Blogs
        .Include(b => b.Posts)
        .ThenInclude(c => c.Comments)
        .ToListAsync();
}
```

Figuur 90: De comments van Post ophalen

- `Include` zal eerst de Post ophalen
- `ThenInclude` zal voor iedere Post de comments ophalen
- `ToListAsync()` zal terug de query uitvoeren

4.10 Generated Values

```
modelBuilder.Entity<Blog>()
    .Property(b => b.Created)
    .HasDefaultValue(DateTime.Now);

modelBuilder.Entity<Post>()
    .Property(b => b.Created)
    .HasDefaultValue(DateTime.Now);

modelBuilder.Entity<Comment>()
    .Property(b => b.Created)
    .HasDefaultValue(DateTime.Now);
```

Figuur 91

- Bij relationele databases hebben we de mogelijkheid om default waardes op te geven
- Handig voor bv Timestamps
- OnModelCreating override:
 - Bij toevoegen Post, Blog of Comment
 - Automatische DateTime.Now() in veld 'Created'

4.11 Extra info

- <https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx>
- <https://docs.microsoft.com/en-us/ef/core/>

4.11.1 Wat moet je zeker kennen?

- Wat en waarom EF Core?
- Opbouw ⇒ model, migrations
- Welke attributen zijn er?
- De commands om migraties te maken en updates te doen
- Hoe kan je data opvragen?
- Hoe zitten de relaties in elkaar?

5 Services & Repositories

5.1 Probleemstelling

5.1.1 Wat hebben we reeds gezien?

- We kunnen API's maken
- We kunnen deze hosten in Docker containers
- We kunnen wegschrijven naar databases die ook in Docker draaien

5.1.2 Probleemstelling

- Weinig structuur in onze code
 - Controller bevat database code (EF Core)
 - * We halen data op in de controller
 - * We voegen data in de database vanuit de controller
 - * Weinig herbruikbaarheid van code mogelijk
 - Geen vaste structuur waar o.a. business logica komt, bv: prijsberekening
 - * Stel dat we de prijs moeten berekenen, dan gaan we dat tot nu doen in de controller zelf
- Controllers moeten zo clean mogelijk zijn
 - Geen logica in controller methode
 - Enkel validatie en doorverwijzing naar andere laag

5.1.3 Oplossing: Design Patterns

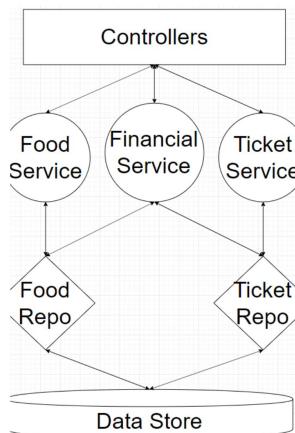
We introduceren **Design Patterns**:

- We gaan meer structuur geven aan onze applicatie
- Zorgt voor meer betrouwbare code
- Makkelijker voor onderhoud en aanpassingen
- Nieuwe teamleden zullen sneller weg vinden in de code
- Patterns zijn **niet** verbonden aan een taal (zowel in C#, Python, ...)

Opgepast:

- Design Patterns lossen niet alles op
- introduceren soms meer complexiteit en meer werk

5.1.4 Repository & Service pattern



Figuur 92

We introduceren Repository & Service pattern

- Controller zal praten met Service laag
 - Service laag (Business layer0 bevat de Business logica)
 - Business logica is:
 - * Berekeningen, beslissingen
 - * Ophalen van data uit de database om beslissingen te nemen
- Service laag zal praten met Repository laag
 - Repository bevat code om met database te spreken
 - Hier zullen we onze LINQ queries schrijven
- Er zijn veel andere patterns, bv: Mediator pattern <https://www.dofactory.com/net/mediator-design-pattern>

5.2 Repositories

- Bevat enkel de code om database operaties uit te voeren ⇒ single responsibility principe
- We proberen per tabel een repository te maken, maar je kan/mag hier van afwijken
- Via de Context voeren we LINQ queries uit
- Context zal via **Dependency Injection (DI)** aanspreekbaar zijn in de repository

5.2.1 In de praktijk

```
public class VaccinTypeRepository
{
    0 references
    public async Task<List<VaccinType>> GetVaccins()
    {
        RegistrationContext context = new RegistrationContext(null);
        return await context.VaccinTypes.ToListAsync();
    }
}
```

- Map Repositories in project
- We voegen VaccinTypeRepository.cs toe
- In deze class gaan we onze methode schrijven, bv: GetVaccins()

5.2.2 Probleem

- We maken context aan in de repository
- We moeten null parameters doorgeven
- Hierdoor is RegistrationContext **tightly coupled** met VaccinTypeRepository
- Ze kunnen **niet** zonder elkaar bestaan
- Dit is niet zo goed naar herbruikbaarheid
- Zeer lastig om te testen

```

public class VaccinTypeRepository
{
    0 references
    public async Task<List<VaccinType>> GetVaccins()
    {
        RegistrationContext context = new RegistrationContext(null);
        return await context.VaccinTypes.ToListAsync();
    }
}

```

Figuur 93

5.2.3 Oplossing

- We moeten de context **loose coupled** maken
- Ze moeten elkaar begrijpen, maar ze mogen niet afhankelijk zijn van elkaar
- Nu gaan we er vanuit dat RegistrationContext zal verwijzen naar onze Docker SQL server
- Bij testen willen we een andere database gebruiken ⇒ in deze opstelling lastig
- We wensen dus flexibiliteit

```

public interface IRegistrationContext
{
    1 reference
    DbSet<VaccinType> VaccinTypes { get; set; }
    0 references
    DbSet<VaccinationRegistration> VaccinationRegistrations { get; set; }
    0 references
    DbSet<VaccinationLocation> VaccinationLocations { get; set; }
}

0 references
public class RegistrationContext : DbContext, IRegistrationContext
{
}

```

Figuur 94: Abstractie van de RegistrationContext: we maken een C# interface

- We maken abstractie van de RegistrationContext
- We gaan bepalen welke methodes er aanwezig zijn, maar niet wat ze moeten doen
- We introduceren een C# interface

```

public interface IVaccinTypeRepository
{
    1 reference
    Task<List<VaccinType>> GetVaccins();
}

1 reference
public class VaccinTypeRepository : IVaccinTypeRepository
{
}

```

Figuur 95

- De interface zal specificeren welke methodes er aanwezig zijn
- Iedere klasse die de interface gaat implementeren kan zijn eigen kan dan eigen context voorzien
- We gaan dit ook doen voor Repositories & Services. Altijd een interface voorzien

Hoe moeten we deze nu gebruiken?

- Hoe moet de Repository de RegistrationContext gebruiken ?
- Hoe moet de Controller de Repository gebruiken ?

⇒ Dependency Injection

5.3 Dependency Injection (DI)

- We gebruiken dit reeds enkele weken
- Alles in ASP.NET WebAPI is DI based
 - Swagger gebruikt DI
 - services.AddControllers() doet dit ook

5.3.1 Wat is dat nu eigenlijk

- Dit is terug een Design pattern
- Doelstelling is Inversion of Control (IoC) bekomen
 - Het aanmaken van objecten gaan we uit handen geven aan een **IoC container**
 - We zijn gewoon om VaccinTypeRepository p = new VaccinTypeRepository() te schrijven
 - De IoC container zal dit voor ons doen achter de schermen
 - Via DI zal dit object beschikbaar zijn als we het nodig hebben
 - Indien je niet tevreden bent kan je de ASP.NET IoC Container uitpluggen en zelf maken

5.3.2 Werking

- De objecten die we wensen te gebruiken via DI moeten we registreren in de IoC container
- Dit zal gebeuren in Startup.cs in de methode ConfigureServices
- Er zijn 3 manieren van registratie:
 1. AddTransient
 - Bij iedere HTTP request zal er een nieuwe instantie gemaakt worden van het object
 - Tijdens hetzelfde request ook een nieuw object indien nodig
 2. AddScoped
 - Bij iedere HTTP request een nieuw object
 - Maar tijdens hetzelfde request zal het object hergebruikt worden
 3. AddSingleton
 - Bij de eerste request zal er object gemaakt worden
 - Daarna zal hetzelfde object gebruikt worden als het nog eens nodig is
 - Ook over verschillende users ⇒ oppassen

Service Type	In the scope of a given http request	Across different http requests
Transient	New Instance	New Instance
Scoped	Same Instance	New Instance
Singleton	Same Instance	Same Instance

Figuur 96: <https://www.c-sharpcorner.com/article/understanding-addtransient-vs-addscoped-vs-addsingleton/>

5.3.3 In de praktijk

We registreren in Startup.cs in de IoC Container de objecten RegistrationContext en VaccinTypeRepository.

Als er nu ergens in onze applicatie een IRegistrationContext of IVaccinTyperepository interface nodig is, dan zal er een object gemaakt van hun respectievelijk type RegistrationContext en VaccinTypeRepository:

```
services.AddTransient<IRegistrationContext, RegistrationContext>();
services.AddTransient<IVaccinTypeRepository, VaccinTypeRepository>();
```

Figuur 97

- In onze VaccinTypeRepository hebben we de RegistrationContext nodig want anders kunnen we niet praten met de database
- We maken gebruik van constructor injection om een instantie van RegistrationContext binnen te brengen in onze VaccinTypeRepository
- De IoC container van ASP.NET Core heeft voor IRegistrationContext een RegistrationContext object geregistreerd in Startup.cs
- Bij het uitvoeren van het project ziet .NET dat er een IRegistrationContext nodig is bij VaccinTypeRepository. ASP.NET vraagt aan de IoC container een object waar de interface IRegistrationContext voor is geregistreerd en zal dit binnen brengen via de constructor.

```
public class VaccinTypeRepository : IVaccinTypeRepository
{
    2 references
    private IRegistrationContext _context;

    0 references
    public VaccinTypeRepository(IRegistrationContext context)
    {
        _context = context;
    }

    1 reference
    public async Task<List<VaccinType>> GetVaccins()
    {
        return await _context.VaccinTypes.ToListAsync();
    }
}
```

Figuur 98

```
System.InvalidOperationException: Unable to resolve service for type 'RegistrationAPI.Data.IRegistrationContext' while attempting to activate 'RegistrationAPI.Repositories.VaccinTypeRepository'.
   at Microsoft.Extensions.DependencyInjection.ServiceLookup.ServiceProviderEngine.GetService(IServiceDescriptor descriptor)
   at Microsoft.Extensions.DependencyInjection.ServiceProvider.GetService(Type serviceType)
   at Microsoft.Extensions.DependencyInjection.ServiceProvider.get_ServiceProvider()
   at Microsoft.AspNetCore.Http.Internal.HttpProtocol.ProcessRequests(HttpContext context)
   at Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Http.HttpProtocol.ProcessRequests(HttpContext context)
   at Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Http.HttpProtocol.ProcessRequests(HttpContext context)
Connection id "0HM6D0AL7R06:88888886": An unhandled exception was thrown by the application.
System.InvalidOperationException: Unable to resolve service for type 'RegistrationAPI.Data.IRegistrationContext' while attempting to activate 'RegistrationAPI.Repositories.VaccinTypeRepository'.
   at Microsoft.Extensions.DependencyInjection.ServiceLookup.ServiceProviderEngine.GetService(IServiceDescriptor descriptor)
   at Microsoft.Extensions.DependencyInjection.ServiceProvider.GetService(Type serviceType)
   at Microsoft.Extensions.DependencyInjection.ServiceProvider.get_ServiceProvider()
   at Microsoft.AspNetCore.Http.Internal.HttpProtocol.ProcessRequests(HttpContext context)
   at Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Http.HttpProtocol.ProcessRequests(HttpContext context)
Connection id "0HM6D0AL7R06:88888886": An unhandled exception was thrown by the application.
System.InvalidOperationException: Unable to resolve service for type 'RegistrationAPI.Data.IRegistrationContext' while attempting to activate 'RegistrationAPI.Repositories.VaccinTypeRepository'.
   at Microsoft.Extensions.DependencyInjection.ServiceLookup.ServiceProviderEngine.GetService(IServiceDescriptor descriptor)
   at Microsoft.Extensions.DependencyInjection.ServiceProvider.GetService(Type serviceType)
   at Microsoft.Extensions.DependencyInjection.ServiceProvider.get_ServiceProvider()
   at Microsoft.AspNetCore.Http.Internal.HttpProtocol.ProcessRequests(HttpContext context)
   at Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Http.HttpProtocol.ProcessRequests(HttpContext context)
```

Figuur 99: Stel dat we IRegistrationContext niet geregistreerd hebben in de Startup.cs: veel gemaakte error!

```

[ApiController]
0 references
public class VaccinatieController : ControllerBase
{
    3 references
    private IVaccinTypeRepository _vaccinTypeRepository;

    0 references
    public VaccinatieController(IVaccinTypeRepository vaccinTypeRepository)
    {
        _vaccinTypeRepository = vaccinTypeRepository;
    }

    [HttpGet]
    [Route("/vaccins")]
    public async Task<List<VaccinType>> GetVaccins()
    {
        return await _vaccinTypeRepository.GetVaccins();
    }
}

```

Figuur 100: We passen nu ook onze Controller aan en injecteren de repository in de Controller

5.4 Services

De service layer bevat:

- Business Logica
 - Prijsberekening
 - Procedures
 - Andere services aanroepen, vb mail sturen
- Aanroepen naar databas
 - Soms is het gewoon doorgeven van method calls
 - Veel overhead code soms

We gaan niet voor iedere Repository een service maken, de service zal meer op functioneel niveau gaan groeperen, bijvoorbeeld:

- PriceService
 - ProductRepository
 - DiscountRepository
 - ...
- OrderService
 - ProductRepository
 - DiscountRepository
 - EmailService ⇒ service aanroepen vanuit andere services

5.4.1 In de praktijk

- Mapje Services in het project
- Klasse VaccinationService
- We injecteren de IVaccinTyperepository in de service
- Daarna kunnen we de repository gebruiken

```

public interface IVaccinationService
{
    0 references
    Task<List<VaccinType>> GetVaccins();
}

0 references
public class VaccinationService : IVaccinationService
{
    2 references
    private readonly IVaccinTypeRepository _vaccinTypeRepository;
    0 references
    public VaccinationService(IVaccinTypeRepository vaccinTypeRepository)
    {
        _vaccinTypeRepository = vaccinTypeRepository;
    }
    0 references
    public async Task<List<VaccinType>> GetVaccins()
    {
        return await _vaccinTypeRepository.GetVaccins();
    }
}

```

Figuur 101

Nu moeten we de service registreren zodat we deze kunnen gebruiken:

```
services.AddTransient<IVaccinationService, VaccinationService>();
```

Figuur 102: Registratie VaccinationService

In de controller injecteren we nu de IVaccinationService en niet meer de Repository

```

[ApiController]
0 references
public class VaccinatieController : ControllerBase
{
    2 references
    private IVaccinationService _vaccinationService;

    0 references
    public VaccinatieController(IVaccinationService vaccinationService)
    {
        _vaccinationService = vaccinationService;
    }

    [HttpGet]
    [Route("/vaccins")]
    0 references
    public async Task<List<VaccinType>> GetVaccins()
    {
        return await _vaccinationService.GetVaccins();
    }
}

```

Figuur 103: Injectie IVaccinationService

5.5 Wat is belangrijk?

- Wat is doel van repository en services?
- Wat is DI en IoC?
- Waarom interfaces nodig?
- Welke zijn de drie manieren om services te registeren in .NET Core?

6 Testing

6.1 Probleemstelling

- Hoe groter het project hoe meer werk om te testen

- Gevolg is dat we dit durven vergeten
- Bij aanpassingen of uitbreidingen moet de bestaande code blijven werken
- We willen geen werkende code kapot maken door aanpassingen of uitbreidingen
- We moeten systeem hebben waar we kunnen op vertrouwen en die automatisch onze code gaat testen

6.2 Waarom testing?

- We willen zeker zijn dat de code werkt onder verschillende omstandigheden, met verschillende input
- Kwaliteit code verhogen
- Verplicht code opdelen in kleine stukken die we kunnen testen
- Je bent verplicht na te denken over de project requirements
- Automatisch ga je meer error handling in je code gaan schrijven
- Gemoedsrustig bij refactoring code als je weet dat er test aanwezig zijn

6.3 2 soorten

- Manueel
- Automatisch

6.3.1 Manual testing

- Persoon zal manueel testing uitvoeren
 - Door de applicatie navigeren
 - Data invullen in de invoerschermen
 - Controleren of dit voldoet aan de eisen
- Dit is zeer duur
- Foutgevoelig: mensen vergeten dingen
- Je moet zeer goede scenario's hebben en deze moeten up to date zijn
- We doen dit als laatste stap in project, als validatie of acceptatie van de klant

6.3.2 Automated test

- Machines gaan testscripts uitvoeren
- Dit gaat van het testen van methodes tot volledige user interfaces
- Dit is goedkoper en sneller
- Toch vraagt dit ook goed geschreven test scripts en scenario's
- Grote voordeel is dat we het automatisch kunnen laten uitvoeren
- We kunnen dit opnemen in een CI/CD pipeline

6.4 Types tests

- Unit tests
- Integration tests
- Functional tests
- Acceptance tests
- Performance test
- Smoke test
- Regression test
- Stress test

6.4.1 Unit test

- Kleine compacte test
- Test heel kleine eigenschappen van de applicatie
- Dit zijn meestal methodes die we testen
- Heel eenvoudige te automatisteren
- Zeer snel uit te voeren door de developer
- Bv.: bereken de prijs van een parkeerticket

6.4.2 Integration test

- Een test die verschillende onderdelen samen zal testen
- Werkt de code goed samen met de database ?
- Werken de verschillende microservices samen ?
 - Bv: spreken we de externe API correct aan?
- Complexer om op te zetten
- Trager bij uitvoeren
- Maakt soms gebruik van dummy databases of in memory databases

6.4.3 Functional test

- Controleren van de business requirements van de applicatie, doet de applicatie wat de klant vraagt ?
- Controleren van wat we inputten en wat er uitkomt, stappen er tussen en hoe het werkt is niet belangrijk
- Zal meestal manueel verlopen, naar het einde van het project toe
- Goede requirements belangrijk

6.4.4 Acceptance tests

- Gaat verder dan functional test, we testen ook requirements maar nog aantal extra zaken
- Volledig applicatie moet up and running zijn
- Zo dicht mogelijk naar productie omstandigheden testen
- Is de performance goed om op te leveren ?
- Meestal laatste test voor oplevering

6.4.5 Performance test

- Testen van applicatie onder heavy load
- Testen van stabiliteit, schaalbaarheid
- Functioneel is niet belangrijk, wel hoeveel kunnen we het systeem duwen tot het ondergaat
- Men kan dit automatizeren

6.4.6 Smoke test

- Eerste test na aanpassing om te beslissen of ze verder gaan testen of niet
- Soms ook **intake test** genoemd

6.4.7 Regression test

- Na het doen van aanpassingen testen of dingen die werkten voor de aanpassingen nog steeds werken

6.4.8 Stress test

- Testen hoe systeem zal reageren als we veel meer load toepassen dan wat moet volgens de requirements
- Bv.: storage en databases extreem belasten

6.5 Test Frameworks

Er is ondersteuning voor verschillende test frameworks:

- MS Test
- Nunit
- **XUnit**
- vAplus (Stress Testing framework van Sizing Servers Lab (Howest))

6.5.1 XUnit

```
1 dotnet new xunit --name <naam-test-project>
```

```
<ItemGroup>
|   | <ProjectReference Include = "...\\Testing\\testing.csproj" />
</ItemGroup>
```

Figuur 104: In het test project moeten we referentie leggen naar onze API

6.6 Unit test in detail

- Testen van één klein stukje code
- We weten wat we als uitkomst verwachten
- We testen ook de mogelijk dingen die kunnen fout lopen
- We schrijven eerst code en dan test
- We schrijven eerst de test en dan de code (test driven development)

6.6.1 Voorbeeld

- Unit test zonder parameter
- [Fact] Attribuut boven de test methode
- Assert zal het resultaat controleren
- Via **dotnet test** commando project uitvoeren

```
[Fact]
0 references | Run Test | Debug Test
public void IsFreeTicket()
{
    ParkingService service = new ParkingService();
    Assert.Equal<decimal>(0, service.CalculatePrice(1));
}
```

Figuur 105: Voorbeeld

- Tweede manier van testen via [Theory]
- Meerdere input data voor zelfde test
- Resultaat altijd zelfde (Assert.True)
- Via [InlineData] waarden doorgeven
- Zal automatisch meerdere tests genereren

```
[Theory]
[InlineData(0)]
[InlineData(1)]
[InlineData(2)]
[InlineData(5)]
[InlineData(10)]
0 references | Run Test | Debug Test
public void IsValidTicketDate(int hours)
{
    ParkingService service = new ParkingService();
    Assert.True(service.IsFreeTicket(hours));
}
```

Figuur 106

- Assert.True
- Assert.False
- Assert.Contains
- Assert.Equals
- ...
- Assert zal check doen of waarde correct is en al dan niet de test doen lukken of mislukken

6.6.2 Exceptions testen

- Ontstaan er exceptions bij foutieve data?
- Ontstaat de juiste exception?

```

[Service]
public decimal CalculatePrice(int hours)
{
    if(hours < 0)
        throw new ArgumentException("Invalid hours");

    if (hours <= 24)
        return 0;

    if (hours <= 48)
        return hours * 2;

    return hours * 1;
}

[Fact]
0 references | Run Test | Debug Test
public void DayTicket_Exception()
{
    ParkingService service = new ParkingService();
    Assert.Throws<ArgumentException>(() => service.CalculatePrice(-5));
}

```

Figuur 107: Voorbeeld

```

PS C:\src\backend-theorie07-demo\unitests> dotnet test
Determining projects to restore...
All projects are up-to-date for restore.
  testing → C:\src\backend-theorie07-demo\Testing\bin\Debug\net5.0\testing.dll
  unitests → C:\src\backend-theorie07-demo\unitests\bin\Debug\net5.0\unitests.dll
Test run for 'C:\src\backend-theorie07-demo\unitests\bin\Debug\net5.0\unitests.dll (.NETCoreApp,Version=v5.0)'
Microsoft (R) Test Execution Command Line Tool Version 16.9.1
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed| Failed| 0 Passed| 0 Skipped| 0 Total| 0 Duration: 270 ms - unitests.dll (net5.0)

```

Figuur 108: dotnet test zal ons project starten en testen

6.7 Integration test in detail

Van controller ⇒ tot in de database

- Controller
- Service
- Repository
- Database

We testen dus de volledige flow ⇒ trager

6.7.1 Testen van de volledige flow

- We moeten dus onze applicatie (API) opstarten in het Test project
- We gaan deze applicatie delen over de verschillende tests
 - **IClassFixture** gebruiken om applicatie te delen

- We maken klasse die erft van **IClassFixture** met als parameter onze **StartUp**
- **WebApplicationFactory** zal **TestServer** opstarten
- In de **ctor** moeten we onze StartUp meegeven via **WebApplicationFactory**

```
public class WeatherControllerTests : IClassFixture<WebApplicationFactory<testing.Startup>>
{
    2 references
    private HttpClient _client { get; }

    0 references
    public WeatherControllerTests(WebApplicationFactory<testing.Startup> fixture)
    {
        _client = fixture.CreateClient();
    }
}
```

6.7.2 Toevoegen van een test

- **_client** zal HTTP Request aanroepen via URL

```
1   response.StatusCode.Should().Be(HttpStatusCode.OK)
```

- Fluent assertion zal controleren of we een OK (200) terugkrijgen

```
var response = await _client.GetAsync("/weatherforecast");
response.StatusCode.Should().Be(HttpStatusCode.OK);
```

- We deserialiseren de `response.Content`
- Via fluent assertions controleren we of er 5 inzitten

```
var forecast = JsonConvert.DeserializeObject<WeatherForecast[]>(await response.Content.ReadAsStringAsync());
forecast.Should().HaveCount(5);
```

```
Assert.True(forecast.Length == 5);
```

Figuur 109

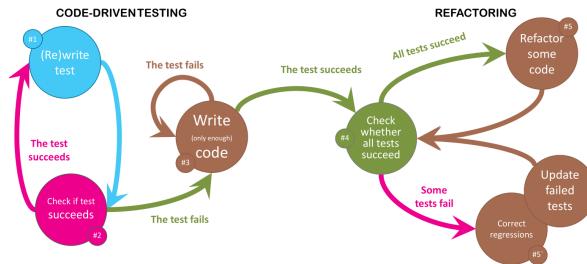
6.7.3 WebApplicationFactory

- `WebApplicationFactory` ⇒ opstarten test server met ons project
- We kunnen startup overriden ⇒ service uitpluggen
- We erven van `WebApplicationFactory<Startup>`
- Daarna kunnen we `ConfigureWebHost` overriden en services registreren specifiek voor test project

```
public class CustomApiWebApplicationFactory : WebApplicationFactory<Sneakers.API.Startup>
{
    0 references
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureTestServices(services =>
        {
            services.AddTransient<IBlobService, FileBlobService>();
        });
    }
}
```

6.8 Test driven development

- Tot nu toe schrijven we onze code en daarna de test die we nodig hebben
- Bij Test Driven Development werken we andersom
- We schrijven een reeks testen die falen
- Als een test lukt, doen we iets verkeerd omdat we nog geen code geschreven hebben ⇒ herschrijven tot de test faalt
- Daarna schrijven we onze programmalogica tot onze testen niet meer falen



Figuur 110: Overzicht TDD

6.9 Code Coverage

- Code coverage/Test Coverage wil zeggen de hoeveelheid source code die getest zal worden door alle geschreven test
- Hoe hoger de test coverage hoe meer source code van het programma getest zal worden door test wat zou moeten resulteren in minder bugs
- We maken hiervoor terug gebruik van opensource framework

6.9.1 Wat zal Code Coverage o.a. proberen te bepalen?

- **Function Coverage:** hoeveel functies in de source code worden aangeroepen door test
- **Branch Coverage:** hoeveel if/else statements worden getest en welke paden (true or false)
- **Line Coverage:** hoeveel lijnen source code worden getest

6.10 Wat is belangrijk?

- Waarom testing?
- Welke testen zijn er en wanneer gebruiken we deze
- Wat is **test driven development**?
- Je moet **unittests** en **integrationtest** kunnen schrijven