



# **EF CORE**

## **Databasetoegang met EntityFrameworkCore**

### **Cursus**

Deze cursus is eigendom van VDAB Competentiecentra ©

versie: 10/07/2020



## INHOUD

1	Inleiding .....	7
1.1	Doelstelling.....	7
1.2	Vereiste voorkennis.....	7
1.3	Nodige software .....	7
1.4	Wat is Entity Framework Core?.....	7
2	SQL Server .....	9
2.1	Een database server .....	9
2.2	Een database .....	9
2.3	SQL Server Management Studio .....	11
2.4	SQL Commando's in SQL Server Management Studio .....	14
2.5	Oefening .....	15
3	De object relational mismatch.....	17
3.1	Granularity.....	17
3.2	Inheritance .....	18
3.2.1	Table per concrete class (TPC) .....	18
3.2.2	Table per hierarchy (TPH) .....	19
3.2.3	Table per type (TPT).....	20
3.3	Associaties .....	21
3.3.1	Eén-op-veel associaties.....	21
3.3.2	Veel-op-veel associaties .....	21
3.3.3	Navigeren door associaties.....	23
3.4	ORM (object-relational mapping) .....	23
4	De solution .....	25
4.1	Opbouw voorbeeld solution.....	25
4.2	EF Core toevoegen .....	26
4.3	Het entity data model .....	27

4.3.1	De entities .....	27
4.3.2	Associaties.....	28
4.4	De DbContext-class.....	29
4.5	De connectionstring.....	29
4.5.1	De connectionstring hardcoded in de DbContext-class.....	30
4.5.2	De connectionstring in appsettings.json.....	30
4.6	Oefening.....	31
5	Migrations.....	33
5.1	Een eerste migration.....	33
5.2	De tabelstructuur verfijnen met annotations.....	34
5.2.1	De naam van de tabel instellen.....	34
5.2.2	De naam van een kolom instellen.....	34
5.2.3	Een kolom instellen als (niet) verplicht in te vullen .....	35
5.2.4	Het maximum aantal tekens in een varchar-kolom instellen .....	35
5.2.5	Het kolomtype instellen.....	35
5.2.6	De primary key instellen .....	35
5.2.7	De foreign key instellen .....	36
5.2.8	Een property niet opnemen in de databasetable .....	36
5.2.9	Toepassing.....	36
5.3	Class Diagram.....	38
5.4	Enums en partial classes .....	39
5.5	Fluent API.....	40
5.5.1	Property mapping .....	40
5.5.2	Type mapping.....	41
5.6	Oefening.....	41
5.7	Seeding.....	42
5.8	Oefening.....	44
6	SQL Logging.....	45
6.1	Logging in Visual Studio .....	45
6.2	Logging in SQL Server Profiler.....	46

6.3	Oefening .....	46
7	De user interface .....	47
7.1	Dependencies .....	47
7.2	Oefening .....	47
8	Query's .....	49
8.1	Een eenvoudig voorbeeld .....	49
8.2	LINQ-query's .....	50
8.3	Query-methods .....	51
8.4	LINQ-query's vs. query methods .....	51
8.5	Een entity zoeken op basis van de PK-waarde .....	53
8.6	Gedeeltelijke objecten ophalen .....	53
8.7	Groeperen in query's .....	54
8.8	Lazy en Eager loading .....	55
8.8.1	Lazy Loading .....	55
8.8.1.1	Met gebruik van proxy's .....	55
8.8.1.2	Met dependency injection .....	57
8.8.2	Eager Loading .....	58
8.9	ToList .....	60
8.10	Oefening .....	61
9	Entity's toevoegen .....	63
9.1	Eén entiteit toevoegen .....	63
9.2	Meerdere entiteiten toevoegen .....	64
9.3	Meerdere entiteiten van een verschillend type toevoegen .....	65
9.4	Entiteiten én bijhorende nieuwe geassocieerde entiteiten toevoegen .....	65
9.5	Entiteiten én bijhorende bestaande geassocieerde entiteiten toevoegen .....	66
9.6	Oefening .....	67
10	Entity's wijzigen .....	69

10.1	Eén entity wijzigen .....	69
10.2	Meerdere entity's lezen en slechts enkele daarvan wijzigen .....	70
10.3	Entity's wijzigen die indirect zijn ingelezen via associaties .....	70
10.4	Een associatie van een entity wijzigen .....	71
10.4.1	De associatie wijzigen vanuit de veel-kant .....	71
10.4.2	De associatie wijzigen vanuit de één-kant .....	72
10.5	Oefening.....	72
11	Entity's verwijderen .....	73
11.1	Voorbeeld .....	73
11.2	Oefening.....	73
12	Unit testing.....	75
12.1	InMemory .....	75
12.2	SQLite .....	82
13	Transacties .....	85
13.1	Kenmerken van transacties .....	85
13.2	Isolation level .....	85
13.3	De method SaveChanges .....	86
13.4	Eigen transactiebeheer met TransactionScope .....	87
13.5	Voorbeeld .....	88
13.5.1	Ingebouwd transactiebeheer .....	89
13.5.2	Eigen transactiebeheer .....	91
13.6	Oefening.....	92
14	Optimistic record locking.....	93
14.1	Probleemstelling .....	93
14.2	Oplossing zonder timestampveld .....	94
14.3	Oplossing met timestampveld .....	96
14.4	Oefening.....	97

15	Scaffolding.....	99
15.1	De database bieren .....	99
15.2	Migrations .....	99
16	De change tracker .....	101
16.1	Statussen van de ChangeTracker .....	101
16.2	Een voorbeeld .....	101
16.3	Het StateChanged en Tracked event.....	103
17	ORM in Entity Framework .....	105
17.1	Owned types mapping .....	105
17.1.1	Een adres property .....	105
17.1.2	Migration .....	106
17.1.3	Mappen van owned types .....	106
17.2	Inheritance .....	107
17.2.1	Table per hierarchy (TPH) .....	107
17.2.2	Table per type (TPT) en Table per concrete class (TPC) .....	109
17.3	Associaties tussen entity's.....	109
17.3.1	Veel-op-veel relatie .....	109
17.3.2	Een associatie naar dezelfde tabel .....	115
17.4	Oefening 1 .....	120
17.5	Oefening 2 .....	121
18	Appendix : Oplossing oefeningen .....	123
18.1	Databasescript EFTuincentrum .....	123
18.2	Solution Taken en database EFBank .....	123
18.3	Initiële migration .....	123
18.4	Seeding .....	124
18.5	Logging.....	124
18.6	References.....	124
18.7	Alfabetische klantenlijst met rekeningen .....	124

---

18.8	Zichtrekening toevoegen .....	125
18.9	Geld storten .....	125
18.10	Klant verwijderen.....	126
18.11	Geld overschrijven .....	126
18.12	Voornaam wijzigen .....	128
18.13	Personeelsleden – managers.....	128
18.14	Artikels .....	130



# 1 Inleiding

---

## 1.1 Doelstelling

In deze cursus leer je het .NET Entity Framework gebruiken. Het entity framework, afgekort EF, brengt objecten in het interne geheugen in verband met records in een relationele database. Op die manier kunnen we op een vlotte manier gegevens bewaren, ophalen, bewerken en verwijderen in een database.

## 1.2 Vereiste voorkennis

Voor deze cursus is volgende voorkennis vereist :

- SQL
- OOP
- Basiskennis C#
- TDD

## 1.3 Nodige software

Bij de aanvang van de cursus veronderstellen we dat volgende software geïnstalleerd is :

- Visual Studio 2019 mét de recentste update of Visual Studio Code
- .NET Core 3.0
- Een SQL Server, eventueel de met Visual Studio meegeleverde SQL Server LocalDb
- Microsoft SQL Server Management Studio

## 1.4 Wat is Entity Framework Core?

Entity Framework is een object-relational mapper (ORM) van Microsoft die het mogelijk maakt om .NET ontwikkelaars te laten werken met een database, gebruik makend van .NET objecten die worden gemapt aan database-tabellen.

Concreet zal de programmeur wijzigingen aanbrengen in één of meerdere objecten in het interne geheugen, EF zorgt ervoor dat deze wijzigingen worden doorgevoerd in de bijhorende database. Dit verhoogt de productiviteit van de .NET-ontwikkelaar. Oudere technologieën zoals ADO.NET vergden heel wat codeerwerk en waren veel arbeidsintensiever.

Wie EF gebruikt werkt standaard met de Code-First techniek. Dit betekent dat de ontwikkelaar eerst het data model maakt en op basis daarvan de tabellenstructuur laat maken door het EF. Heb je reeds een database dan kan je het EF het data model laten maken. We noemen dit ook scaffolding en we spreken dan van Database-First.

EF kan overweg met tal van verschillende soorten databases. Wij zullen enkel werken met SQL Server databases.

Momenteel is EF toe aan versie nummer 3.1.0. De release-cyclus staat los van de .NET Core release cyclus. Je vindt alle informatie over EF Core op <https://docs.microsoft.com/nl-nl/ef/core/>



## 2 SQL Server

---

### 2.1 Een database server

Zoals gezegd kan EF Core overweg met verschillende soorten database servers. In deze cursus zullen we telkens gebruik maken van de in Visual Studio ingebouwde database server localDb.

LocalDb is een database server, bedoeld om te gebruiken tijdens de ontwikkeling van jouw software project. Van zodra jouw project in productie gaat zal je eerder een Microsoft SQL Server of SQL Server Express gebruiken. Dit zijn ook database servers van Microsoft. SQL Express is wel beperkter in zijn mogelijkheden :

- Een database is op SQL Express maximaal 4 GB groot
- SQL Express gebruikt maximaal 1GB RAM (ook als er meer beschikbaar is)
- SQL Express gebruikt maximaal 1 processor (ook als er meerdere beschikbaar zijn)

Er zijn twee manieren om jouw database te beheren. Ofwel gebruik je SQL Server Management Studio. Dit is een tool van Microsoft waarmee je niet alleen de inhoud van de tabellen in je database kan bekijken maar ook editen. Je kan de structuur van de tabellen veranderen, scripts uitvoeren,...

De meeste van deze taken kan je ook vanuit Visual Studio uitvoeren. Je doet dit in het venster Server Explorer. Na het leggen van een Data Connection kan je ook van hieruit wijzigingen aanbrengen in de database.

### 2.2 Een database

In deze cursus en in de bijhorende taken zullen we werken met enkele databases.

Eén van die databases is de database EFBieren. Deze bevat volgende 3 tabellen :

- **Brouwers** Gegevens over de brouwers.
- **Soorten** Gegevens over de biersoorten, (alcoholvrij, alcoholarm, ...)
- **Bieren** Gegevens over de bieren zelf.

De tabel Brouwers bevat volgende velden :

Veldnaam	Type	Lengte	Opmerkingen
BrouwerNr	Int		Primary Key én AutoNumber
BrNaam	Nvarchar	50	
Adres	Nvarchar	50	
PostCode	SmallInt		
Gemeente	Nvarchar	50	
Omzet	Int		Jaarlijkse omzet uitgedrukt in hectoliter

Enkele voorbeeldrecords van deze tabel :

BrouwerNr	BrNaam	Adres	PostCode	Gemeente	Omzet
1	Achouffe	Route du Village 32	6666	Achouffe-Wibrin	10000
2	Alken	Stationstraat 2	3570	Alken	950000
3	Ambly	Rue Principale 45	6953	Ambly-Nassogne	500
4	Anker	Guido Gezellelaan 49	2800	Mechelen	3000
6	Artois	Vaartstraat 94	3000	Leuven	4000000
8	Bavik	Rijksweg 33	8531	Bavikrove	110000
9	Belle Vue - Molenbeek	Henegouwenkaai 33	1080	Sint-Jans-Molenbeek	NULL
10	Belle Vue - Zuun	Steenweg naar Bergen	1600	Sint- Pieters-Leeuw	NULL
11	Belle Vue	Delaunoy-straat 58-60	1080	Sint-Jans-Molenbeek	300000
12	Bie (De)	Stoppelweg 26	8978	Watou	280
13	Binchoise	Faubourg St. Paul 38	7130	Binche	700

De tabel Soorten bevat volgende velden :

Veldnaam	Type	Lengte	Opmerkingen
SoortNr	Int		Primary Key én AutoNumber
Soort	Nvarchar	50	Geïndexeerd met toelating van dubbels

Enkele voorbeeldrecords van deze tabel :

SoortNr	Soort
2	Alcoholarm
3	Alcoholvrij
4	Ale
5	Alt
6	Amber
8	Bierette
11	Bitter

De tabel Bieren bevat volgende velden :

Veldnaam	Type	Lengte	Opmerkingen
BierNr	Int		Primary Key én AutoNumber
Naam	Nvarchar	100	Geïndexeerd met toelating van dubbels
BrouwerNr	Int		
SoortNr	Int		
Alcohol	Real		

Enkele voorbeeldrecords van deze tabel :

BierNr	Naam	BrouwerNr	SoortNr	Alcohol
4	A.C.O.	104	18	7
5	Aalbeeks St. Comeliusbier (=Kapittel pater (Het))	113	18	6,5
7	Aardbeien witbier	56	53	2,5
8	Aarschots kruikenbier (=St. Sebastiaan grand cru)	105	15	7,6
10	Abt Bijbier (Nen)	33	18	7
11	Adler	51	42	6,75
12	Aerts 1900	81	14	7
13	Affligem blond (Abdij)	100	33	7

Het relatiediagram van deze database :



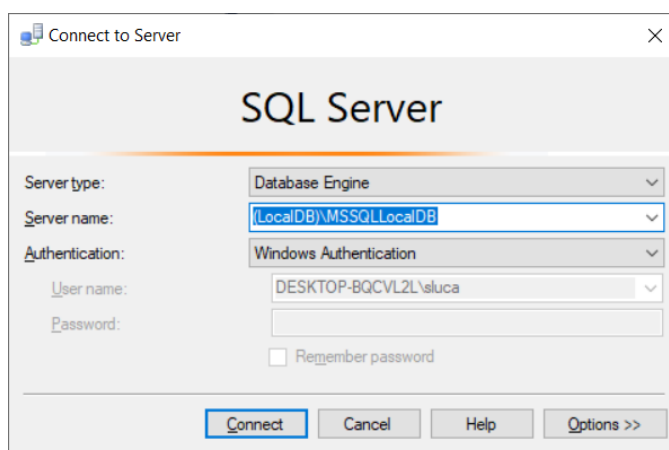
Verderop in deze cursus zullen we deze database gebruiken om scaffolding te demonstreren.

## 2.3 SQL Server Management Studio

Zoals we eerder aangegeven hebben kunnen we de tool SQL Server Management Studio gebruiken om onze databases te beheren.

We proberen nu één en ander uit met een database EFBieren. Deze database maken we aan met het databasescript *EFcreateBieren.sql*.

- Start de tool SQL Server Management Studio.
- In het venster *Connect to Server* tik je de naam van de database server : **(localdb)\mssqllocaldb**. Dit is de naam van de database server die ingebouwd is in Visual Studio.
- Bij *Authentication* kies je voor *Windows Authentication*. Op deze manier log je in met dezelfde user waarmee je ook in Windows ben ingelogd.



- Klik op *Connect*.

In SQL Server Management Studio vind je standaard aan de linkerkant de Object Explorer. In dit venster vind je een overzicht van de databaseservers waarmee je verbonden bent. Per databaseserver kan je meerdere databases beheren.

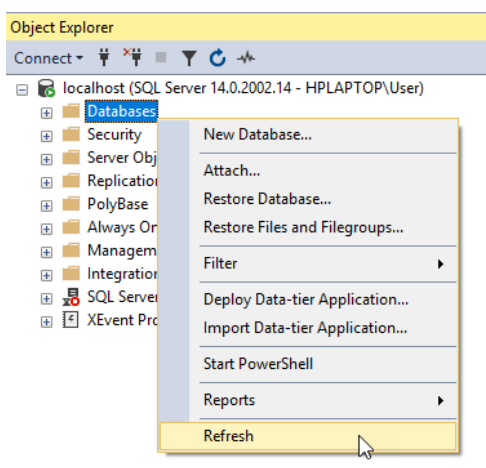
Aan de rechterkant van het scherm is er plaats voor meerdere queryvensters waarin je allerlei SQL commando's kan intikken en laten uitvoeren. Het resultaat van een query wordt onder het queryvenster getoond.

We voeren nu het script uit dat de database EFBieren zal aanmaken.

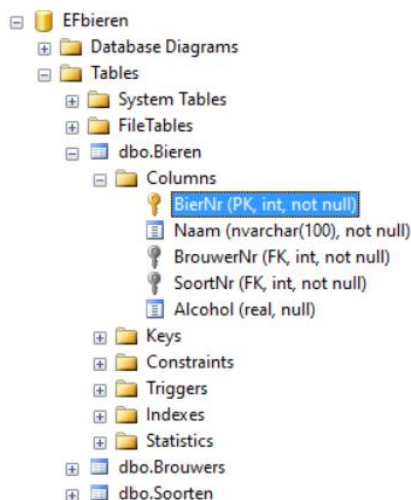
- Kies in het menu *File* de opdracht *Open – File*.
- Selecteer in de folder waar jouw oefenbestanden staan het bestand *EFcreateBieren.sql* en klik op *Open*.
- Klik in de knoppenbalk op de knop *Execute*.

Het script wordt nu uitgevoerd : de database wordt aangemaakt, tabellen en indexen wordt gecreëerd en gegevens worden toegevoegd. Als alles goed verloopt kunnen we nu de nieuwe database bekijken.

- Klik in de *Object Explorer* met de rechtermuistoets bij de database server *LocalDb* op *Databases* en kies *Refresh*.



- Klap de folder *Databases* open en je ziet de database *EFBieren*.
- Klap nu ook opeenvolgend de folders *Tables*, *dbo.Bieren* en *Columns* open. Je ziet nu alle velden van de tabel *Bieren*.
- *BierNr* is de primary key. Klik met de rechtermuistoets op de kolom *BierNr* en kies *Properties*. Je krijgt een eigenschappenvenster te zien waarop je o.a. ziet dat de property *Identity* op *true* staat, met *Identity Seed* op 1 en *Identity Increment* op 1. Dit betekent dat dit een autonummerveld is beginnend op 1 en telkens 1 hoger.



We kunnen voor elke tabel makkelijk de eerste 1000 records raadplegen en ook de eerste 200 records editen.

- Klik met de rechtermuistoets op `dbo.Bieren` en kies `Select Top 1000 Rows`. Je ziet de eerste 1000 records. De query die hiervoor zorgt kan je wijzigen en opnieuw uitvoeren.
- Klik met de rechtermuistoets op `dbo.Bieren` en kies `Edit Top 200 Rows`. Je kan de eerste 200 records wijzigen. De kolom `BierNr` is read-only want het is een autonummerveld.

We maken even een zijstapje naar Visual Studio want ook daar kunnen we de database inkijken.

- Start Visual Studio. Je krijgt de kans een nieuw project aan te maken of een bestaande te openen. Kies rechts onderaan voor *Continue without code*.
- Kies in het menu `View – Server Explorer`. In dit venster kunnen we een verbinding leggen met een SQL Server database.
- Klik met de rechtermuistoets op *Data Connections* en kies *Add Connection...*
- In het venster *Add Connection* tik je bij *Server name* de naam van jouw server. Wij gebruiken `localdb` dus dit is dan `(localdb)\mssqllocaldb`. Je tikt deze naam best zelf in. Klap je de keuzelijst open dan kan het een tijdje duren vooraleer de beschikbare servers uitgelijst worden.

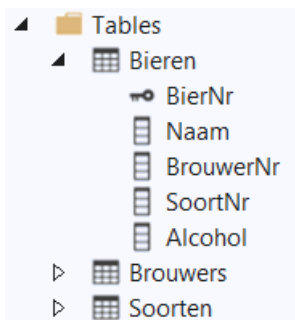
The screenshot shows the 'Add Connection' dialog box in Visual Studio. The dialog is titled 'Add Connection' and has a close button (X) in the top right corner. Below the title bar, there is a text box with the instruction: 'Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.'

The dialog is divided into several sections:

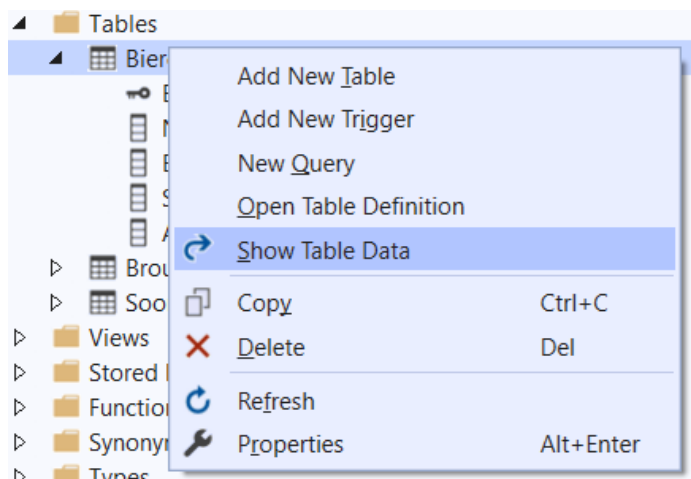
- Data source:** A text box containing 'Microsoft SQL Server (SqlClient)' and a 'Change...' button.
- Server name:** A text box containing '(localdb)\mssqllocaldb' and a 'Refresh' button.
- Log on to the server:** A section containing an 'Authentication' dropdown menu set to 'Windows Authentication', a 'User name:' text box, a 'Password:' text box, and a checkbox labeled 'Save my password'.
- Connect to a database:** A section with two radio buttons: 'Select or enter a database name:' (selected) and 'Attach a database file:'. The 'Select or enter a database name:' option has a dropdown menu showing 'EFbieren'. The 'Attach a database file:' option has a 'Browse...' button and a 'Logical name:' text box.
- Advanced...** A button located below the 'Connect to a database' section.
- Test Connection** A button located at the bottom left.
- OK** and **Cancel** buttons located at the bottom right.

- Onderaan bij *Connect to a database* tik je de naam van de database.
- Test eventueel de connectie met een klik op de knop *Test Connection*.

Als alles goed verloopt wordt de database nu opgenomen in de lijst Data Connections in Server Explorer. Je kan de structuur van de tables bekijken door de folder Tables open te klappen en vervolgens ook de tabel zelf.



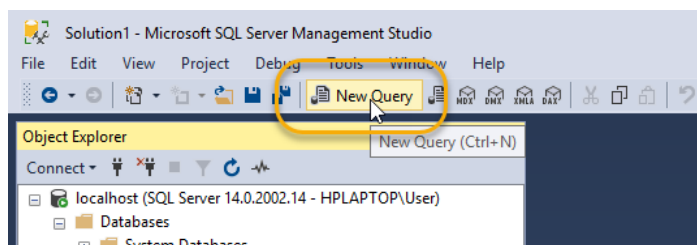
Een rechtermuisklik op de tabelnaam geeft je toegang tot een aantal opdrachten zoals het bekijken van de inhoud (*Show Table Data*) of de structuur (*Open Table Definition*) van de tabel.



## 2.4 SQL Commando's in SQL Server Management Studio

In SQL Server Management Studio kan je allerlei query's uitvoeren op één of meerdere tabellen. Dit kon je reeds zien toen we de top 1000 rows van de tabel Bieren bekeken. We proberen nog enkele eenvoudige query's uit te voeren.

- Kies in de toolbar de knop *New Query*.



In het nieuwe Queryvenster kan je nu de gewenste SQL commando's intikken.

Vooraleer je een query kan uitvoeren moet je aangeven in welke database je wil werken. Je doet dit met het use-commando.

- Tik het commando **use EFBieren** en klik op de knop *Execute*. Als dat lukt krijg je de melding *Command(s) completed successfully*.



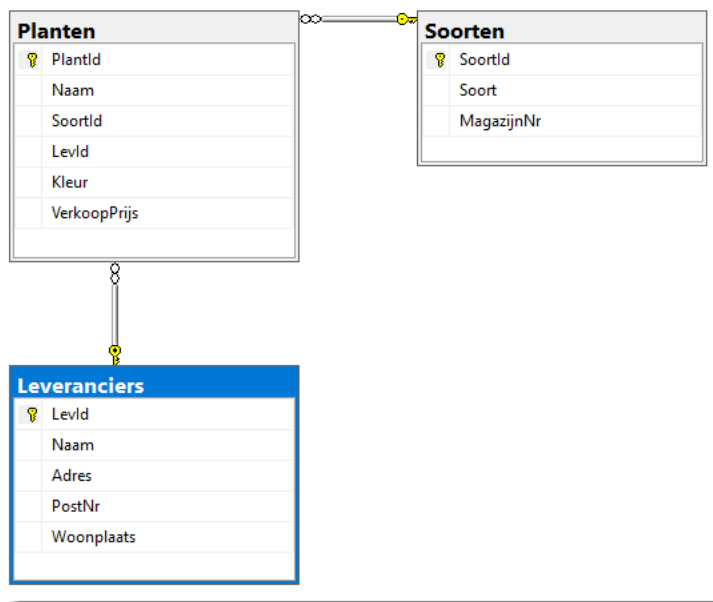
- Vervang het vorige commando door het commando **select \* from bieren where alcohol > 5** en klik op *Execute*. Je ziet alle bieren met een alcoholpercentage boven de 5%.

Bij het sluiten van SSMS zal er gevraagd worden of je de openstaande queryvensters wil opslaan. Aan jou de keuze om eventuele query's op te slaan.

## 2.5 Oefening

Maak aan de hand van het SQL-script *EFcreateTuincentrum.sql* de database *EFTuincentrum* aan in SQL Server Management Studio.

Het relatiediagram van de database *EFTuincentrum* :



Probeer ook hier enkele query's uit.



## 3 De object relational mismatch

Je stelt data in het interne geheugen voor als objecten. Dit zijn instanties van classes. Diezelfde data wordt in een database voorgesteld als records in een table.

De manier waarop je data voorstelt als objecten stemt niet altijd 100% overeen met de manier waarop je data opslaat als records in een table. Dit verschil in voorstelling van data noemen we de 'object-relational' mismatch. Dit verschil heeft meerdere aspecten die we hieronder uitleggen.

### 3.1 Granularity

Granularity ('korreligheid') geeft aan in welke mate je data kan opsplitsen in onderdelen.

Een **database** heeft slechts twee granularity-niveaus : tabellen en kolommen.

- Tabellen bevatten kolommen
- Een kolom bevat een waarde (getal, datum, tekst) die je niet verder kan opsplitsen

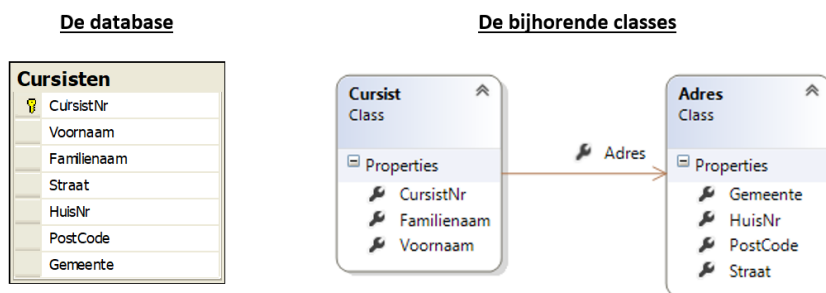
**Objecten in C#** hebben op het eerste zicht maar twee granularity-niveaus: classes en properties.

- Classes bevatten properties
- Een property kan een enkelvoudige waarde bevatten (int, decimal, ...) maar kan echter ook een reference zijn naar een object dat ook properties bevat en die properties...

Op die manier is de granularity van classes oneindig.

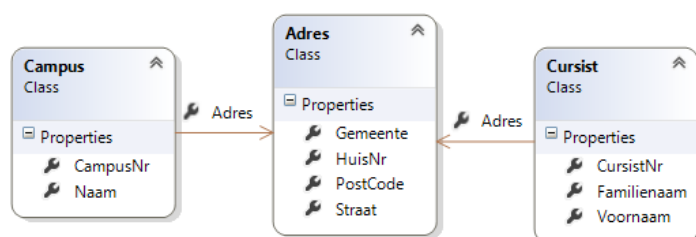
Een voorbeeld : beschouw het gegeven **cursist**.

In de database bevat één table alle cursisteigenschappen. In C# zijn deze eigenschappen verdeeld over twee classes :



De class Cursist bevat een reference naar de class Adres. De class Adres bevat de properties Gemeente, HuisNr, PostCode en Straat.

Een aparte class Adres is handig: je kan die ook gebruiken vanuit andere classes :



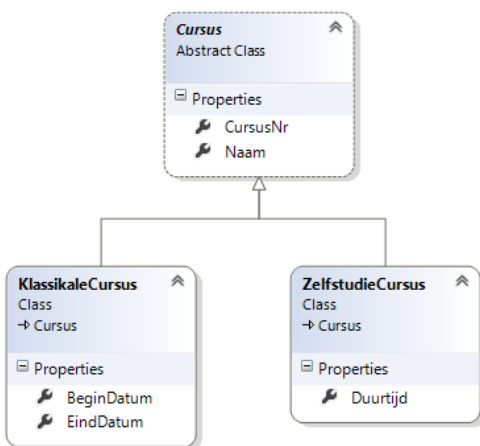
Een class kan zelfs meerdere keren verwijzen naar één andere class. Een class Klant kan bijvoorbeeld twee keer naar de class Adres verwijzen : één maal voor een facturatieadres en één maal voor een leveringsadres.



### 3.2 Inheritance

Inheritance is een essentieel onderdeel van C#.

Voorbeeld : de classes KlassikaleCursus en ZelfstudieCursus erven van de class Cursus :



Inheritance bestaat niet in een database. Je kan er inheritance enkel nabootsen en dat op drie verschillende manieren :


- a) table per concrete class (TPC)
- b) table per class hierarchy (TPH)
- c) table per type (TPT)

We bekijken deze oplossingen nu verder in detail.

#### 3.2.1 Table per concrete class (TPC)

Bij Table per concrete class bevat de database één table per niet-abstracte class. De table bevat kolommen voor alle property's van de class, inclusief de geërfde property's.

Voorbeeld :

KlassikaleCursussen	
	CursusNr
	Naam
	BeginDatum
	EindDatum

Zelfstudie cursussen	
	CursusNr
	Naam
	DuurTijd

Een voorbeeld van mogelijke data :

#### KlassikaleCursussen

CursusNr	Naam	BeginDatum	EindDatum
1	Frans voor beginners	01-09-2007	11-09-2007
2	Frans voor gevorderden	12-09-2007	22-09-2007

#### ZelfstudieCursussen


CursusNr	Naam	DuurTijd
1	Franse correspondentie	5
2	Engelse correspondentie	5

Nadeel van table per concrete class : als je bijvoorbeeld een property toevoegt aan de base class, dan moet je in de database aan meerdere tabellen een extra kolom toevoegen. Bijvoorbeeld : je voegt een property *Prijs* toe aan de class *Cursus*. Deze kolom moet dan worden toegevoegd aan de tabel *KlassikaleCursussen* én aan de tabel *ZelfstudieCursussen*.

### 3.2.2 Table per hierarchy (TPH)

Bij Table per hierarchy bevat de database één table voor de complete class-inheritance hiërarchie. Deze tabel bevat kolommen voor alle property's van alle klassen van de hiërarchie én een kolom die aangeeft bij welke subclass een record hoort.

Voorbeeld :

Cursussen	
	CursusNr
	Naam
	BeginDatum
	EindDatum
	DuurTijd
	Soort

In dit voorbeeld is er een extra kolom *Soort* opgenomen naast de kolommen uit beide subclasses. De kolom *Soort* bevat K als het record een klassikalecursus is en Z als het record een zelfstudie cursus voorstelt.

Een voorbeeld van mogelijke data :

#### Cursussen

CursusNr	Naam	BeginDatum	EindDatum	DuurTijd	Soort
1	Frans voor beginners	01/09/2007	11/09/2007		K
2	Frans voor gevorderden	12/09/2007	22/09/2007		K
3	Franse correspondentie			5	Z
4	Engels correspondentie			5	Z

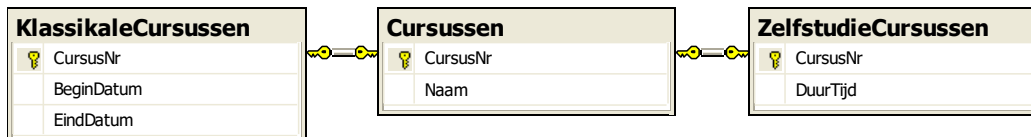
Nadeel van table per class hierarchy : je kan op de kolommen die horen bij de property's van de subclasses geen constraints toepassen, want een constraint geldt voor alle records van de tabel. Je kan bijvoorbeeld geen not-null constraint op de kolom *DuurTijd* plaatsen want deze kolom vul je enkel in bij een *Zelfstudie*cursus.

### 3.2.3 Table per type (TPT)

Bij Table per type bevat de database één table per class uit de class inheritance.

Iedere table bevat enkel kolommen voor de property's die de bijhorende class niet erft. Een table die hoort bij een subclass bevat een primary key die ook een foreign key is naar de primary key van de table die hoort bij de base class.

Voorbeeld :



De kolom **CursusNr** is in de tabel **Cursussen** een autonummer kolom maar niet in de tabellen **KlassikaleCursussen** en **ZelfstudieCursussen**.

Een voorbeeld van mogelijke data :

**Cursussen**

CursusNr	Naam
1	Frans voor beginners
2	Frans voor gevorderden
3	Engels voor beginners
4	Engels voor gevorderden
5	Franse correspondentie
6	Engelse correspondentie

**KlassikaleCursussen**

CursusNr	BeginDatum	EindDatum
1	01-09-2007	11-09-2007
2	12-09-2007	22-09-2007
3	01-09-2007	11-09-2007
4	12-09-2007	22-09-2007

**ZelfstudieCursussen**

CursusNr	DuurTijd
5	5
6	5

Nadeel van table per type : je moet twee tables joinen om de gegevens van **KlassikaleCursussen** of **ZelfstudieCursussen** op te halen. Dit benadeelt de performantie.

Je ziet dat geen enkele van de drie inheritance-nabootsing perfect is.

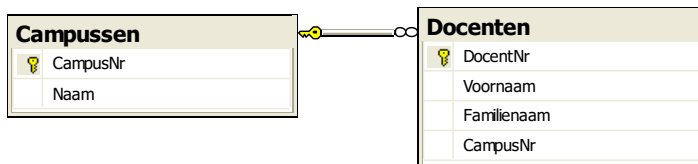
### 3.3 Associaties

We bekijken nu hoe we één-op-veel en veel-op-veel associaties voorstellen in een database en hoe we navigeren doorheen een associatie.

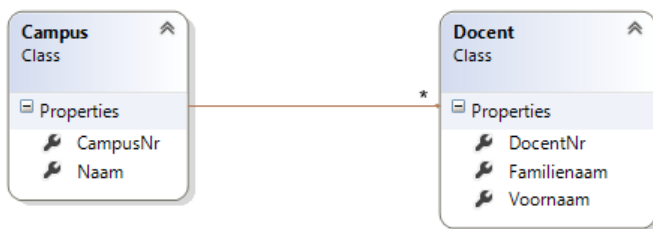
#### 3.3.1 Eén-op-veel associaties

Je stelt een één-op-veel-associatie in de database voor met twee tables. De table aan de veel-zijde van de associatie bevat een foreign-key-kolom, die verwijst naar de primary-key-kolom van de table aan de één-zijde van de associatie.

Voorbeeld :



In C# stel je dezelfde één-op-veel-associatie voor met twee classes, die je verbindt met een associatie:



De class aan de veel-zijde van de associatie (Docent) bevat één reference met als type de class aan de één-zijde van de associatie (Campus). Docent bevat dus een reference-variabele van het type Campus, waarmee je bijhoudt welke campus bij de docent hoort.

```

public class Docent
{
    public int DocentNr { get; set; }
    public string Familiennaam { get; set; }
    public string Voornaam { get; set; }
    public Campus Campus { get; set; }
}
  
```

De class aan de één-zijde van de associatie (Campus) bevat een verzameling references met als type de class aan de veel-zijde van de associatie. Campus bevat dus een verzameling reference-variabelen van het type Docent, waarmee je bijhoudt welke docenten bij de campus horen.

```

public class Campus
{
    public int CampusNr { get; set; }
    public string Naam { get; set; }
    public ICollection<Docent> Docenten { get; set; }
}
  
```

#### 3.3.2 Veel-op-veel associaties

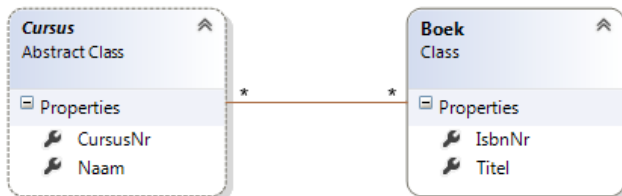
In een relationele database zijn er enkel één-op-één en één-op-veel relaties. Om een veel-op-veel relatie voor te stellen is een tussentabel nodig. De oorspronkelijke tabellen hebben dan een één-op-veel relatie met deze tussentabel.

Voorbeeld :



In een cursus worden meerdere boeken gebruikt. Een boek kan ook in meerdere cursussen worden gebruikt.

In C# kan je deze veel-op-veel relatie tussen Cursus en Boek gewoon uitdrukken zonder tussenclass :



Beide classes bevatten een verzameling references met als type de class aan de andere zijde van de associatie.

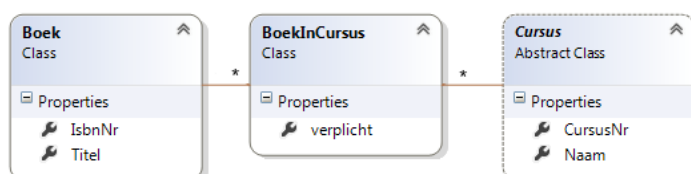
De class Cursus bevat dus een verzameling reference-variabelen van het type Boek, waarmee je bijhoudt welke boeken bij de cursus horen.

```
public class Cursus
{
    public int CursusNr { get; set; }
    public string Naam { get; set; }
    public ICollection<Boek> Boeken { get; set; }
}
```

De class Boek bevat een verzameling reference-variabelen van het type Cursus, waarmee je bijhoudt welke cursussen bij het boek horen.

```
public class Boek
{
    public string IsbnNr { get; set; }
    public string Titel { get; set; }
    public ICollection<Cursus> Cursussen { get; set; }
}
```

Er is echter (voorlopig) één groot probleem : EF Core ondersteunt voorlopig geen veel-op-veel associaties zonder tussenclass ! Bovendien heb je ook een tussenclass nodig van zodra je over de associatie zelf informatie bijhoudt in één of meerdere property's. In onderstaand voorbeeld houden we bij of een boek al dan niet verplicht te gebruiken is in een bepaalde cursus :



Je leert in paragraaf 17.3.1 hoe je dit in EF Core oplost met een tussentabel.



### 3.3.3 Navigeren door associaties

Je navigeert in C# van een object naar een geassocieerd object door de reference-variabele te volgen die de associatie definieert.

In onderstaande code tonen we de titels van de boeken die bij een cursus horen :

```
public void ToonCampusEnBijbehorendeDocenten(Campus campus)
{
    Console.WriteLine(campus.Naam);

    foreach (var docent in campus.Docenten)
    {
        Console.WriteLine($"{docent.DocentId} {docent.Voornaam} {docent.Familienaam}");
    }
}
```

Verderop in de cursus leer je dergelijke code schrijven.

## 3.4 ORM (object-relational mapping)

Je stelt dus gegevens in C# op een andere manier voor dan in een database. Dit betekent dat je ergens een vertaalslag zal moeten doen tussen deze verschillende visies. Deze vertaalslag zelf uitschrijven is niet gemakkelijk en vergt veel tijd.

ORM is juist het converteren van de object-georiënteerde visie naar de database-visie. Een ORM-library helpt je deze conversie te doen. EF Core is een ORM-library van Microsoft.

Een ORM-library biedt volgende voordelen :

- ✓ Productiviteit

Zelf de code schrijven die de vertaalslag doet tussen de twee verschillende visies vraagt veel code en tijd. Een ORM-library neemt je veel werk uit handen.

- ✓ Onderhoudbaarheid

Het databaseschema wijzigt in de tijd. Het class diagram eveneens. Deze visies continu op mekaar afstemmen gaat gemakkelijker mét een ORM-library dan zonder.

- ✓ Databasemerk onafhankelijk

Een ORM-library neemt de verschillen tussen databases voor zijn rekening.



## 4 De solution

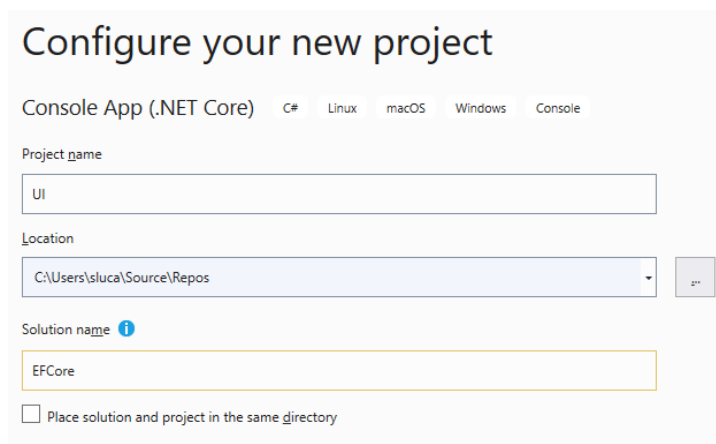
---

Zoals eerder gezegd zullen we in deze cursus leren werken met het Entity Framework Core. We zullen dit doen aan de hand van een console-application. Deze applicatie bouwen we op in dit hoofdstuk.

### 4.1 Opbouw voorbeeld solution

We beginnen met het opbouwen van de solution.

- Maak in Visual Studio een nieuw project van het type *Console App (.NET Core)*. Je noemt het project *UI* en de solution *EFCore*.



We voegen nog een tweede project toe, namelijk van het type Class Library (.NET Core).

- Voeg een .NET Core class library toe en noem deze *Model*.
- Verwijder de class *Class1.cs*.

In het project *Model* komen de entities die gekoppeld zullen worden aan de tabellen in de database.

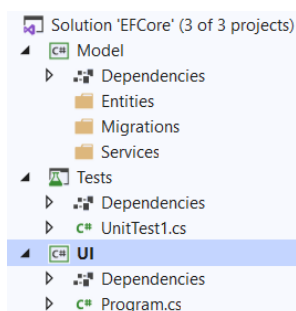
We voegen nog enkele folders toe aan het project *Model*.

- Voeg in het project *Model* de folders *Entities*, *Migrations* en *Services* toe.

We voegen nog een derde project toe, een test-project.

- Voeg aan de solution een project toe van het type *MSTest Test Project (.NET Core)* en noem het *Tests*.

Het resultaat :



## 4.2 EF Core toevoegen

EF Core wordt standaard niet toegevoegd aan een project. We moeten dit zelf doen.

In de volgende paragraaf zullen we in het project Model de entities toevoegen die we via het entity framework gaan koppelen met de databasetabellen. Het is dus in dit project dat we EF Core zullen toevoegen.

- Kies in het menu voor *Tools – NuGet Package Manager – Manage NuGet Packages for Solution...*
- Klik links bovenaan op *Browse...*
- In het zoekveld tik je : *microsoft.entityframeworkcore*
- Zoek in de lijst de package *Microsoft.EntityFrameworkCore* en klik erop.
- Vink ernaast het project *Model* aan en kies naast *Version:* voor versie **3.1.5**. Klik op *Install*.
- Vink in het venster *Preview Changes* de optie *Do not show this again* aan en klik op *OK*.
- Aanvaard de licentievoorwaarden met een klik op *I Accept*.

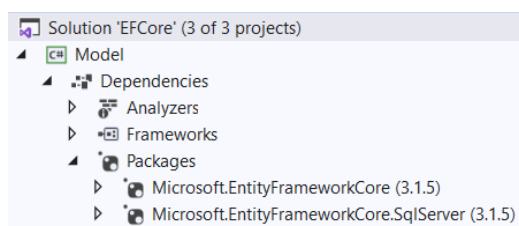
In de Solution Explorer kan je zien dan bij het project Model, onder Dependencies en Packages nu EFCore 3.1.5 is vermeld.

Op dezelfde manier voegen we nu ook de package *Microsoft.EntityFrameworkCore.SqlServer* toe. Deze package zorgt ervoor dat we EFCore kunnen toepassen op een SQL Server database.

Een overzicht van andere EF Core database providers vind je op <https://docs.microsoft.com/en-us/ef/core/providers>.

- Kies opnieuw *Tools – NuGet Package Manager – Manage NuGet Packages for Solution...*
- Deze keer zoek je *microsoft.entityframeworkcore.sqlserver*
- Klik de juiste package aan, vink het project Model aan, kies versie **3.1.5** en klik op *Install*.
- Aanvaard de licentievoorwaarden met een klik op *I Accept*.

In de Solution Explorer zie je bij het project Model nu beide packages staan.



### Tip

Je kan een package ook nog op andere manieren installeren.

#### Met commando's in de package manager console

Je installeert de package *Microsoft.EntityFrameworkCore* in het project *Model* via het commando :

```
install-package microsoft.entityframeworkcore -version 3.1.5 -projectname Model
```

### Via het .csproj-bestand

Je dubbelklikt in de Solution Explorer op de naam van het project Model. Een bestand met de naam Model.csproj wordt geopend.

Voeg binnen een ItemGroup de nodige packagereferences toe.

Voorbeeld :

```
<Project Sdk="Microsoft.NET.Sdk">
  ...
  <ItemGroup>
    <PackageReference Include="microsoft.entityframeworkcore" Version="3.1.5" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="3.1.5"/>
  </ItemGroup>
</Project>
```

### Via de command line interface (CLI)

Je start de Developer Command Prompt for VS2019.

Verplaats je naar de folder waar jouw project opgeslagen is.

Tik het commando :

```
dotnet add package Microsoft.EntityFrameworkCore --version 3.1.5
```

## 4.3 Het entity data model

Om te kunnen communiceren met de onderliggende database is het voor EF noodzakelijk om te beschikken over een entity data model.

Dit entity data model (EDM) bestaat uit 3 delen : het conceptueel model, het storage model en de mapping.

Het **conceptueel model** bevat de structuur van de entity classes. Een entity class is een class die een gegeven uit de werkelijkheid voorstelt. De classes Docent en Campus zijn bijvoorbeeld entity classes. De objecten die we maken op basis van deze classes noemen we entities. Het conceptueel model bevat ook de associaties tussen de entity classes.

Het **storage model** bevat de structuur van iedere tabel en de relaties tussen deze tabellen.

De **mapping** bevat informatie over welke tabel bij welke entity class hoort. Het legt dus het verband tussen de entity classes en de database.

Conceptueel model, storage model en mapping worden door EF gegenereerd en geconfigureerd op basis van conventions. Je kan hier eventueel zelf wijzigingen of toevoegingen in aanbrengen.

### 4.3.1 De entities

We starten met een class Campus.

- Maak in de map Entities (in het project Model) een class Campus aan.
- Vul de class aan met onderstaande code :

```
public class Campus
{
```

```

    public int CampusId { get; set; }
    public string Naam { get; set; }
    public string Straat { get; set; }
    public string Huisnummer { get; set; }
    public string Postcode { get; set; }
    public string Gemeente { get; set; }
}

```

De class Docent is nog onbekend en daar maken we nu verandering in.

- Voeg in dezelfde folder ook een class Docent toe :

```

public class Docent
{
    public int DocentId { get; set; }
    public string Voornaam { get; set; }
    public string Familienaam { get; set; }
    public decimal Wedde { get; set; }
}

```

#### 4.3.2 Associaties

Tussen de classes Campus en Docent bestaat een 1-op-veel relatie. We drukken dit uit door in de classes Docent en Campus extra properties op te nemen :

```

public class Campus
{
    public int CampusId { get; set; }
    public string Naam { get; set; }
    public string Straat { get; set; }
    public string Huisnummer { get; set; }
    public string Postcode { get; set; }
    public string Gemeente { get; set; }
    public virtual ICollection<Docent> Docenten { get; set; }
}
(1)

public class Docent
{
    public int DocentId { get; set; }
    public string Voornaam { get; set; }
    public string Familienaam { get; set; }
    public decimal Wedde { get; set; }
    public int CampusId { get; set; }
    public virtual Campus Campus { get; set; }
}
(2)
(3)

```

- (1) In een campus werken meerdere docenten. We drukken dit uit door aan de Campus-class een ICollection<Docent> toe te voegen.
- (2) Een docent is verbonden aan één campus. We nemen de id van de campus op in de Docent-class.
- (3) Bijkomend voegen we aan de Docent-class een virtual property Campus toe. Deze property is van het type Campus.

De property's Docenten in de class Campus en Campus in de class Docent noemen we **navigation property's**.

## 4.4 De DbContext-class

In het project Model gaan we nu een nieuwe class aanmaken die een voorstelling is van de database. Zo een voorstelling noemen we een context class. We gebruiken hiervoor een class die erft van DbContext.

Aan de hand van deze context class spreken we de database aan :

- gegevens opvragen uit de database
- gegevens toevoegen aan de database
- gegevens aanpassen in de database
- gegevens verwijderen uit de database
- connecteren met de database
- het model en de configuraties instellen
- configuratie van de change tracking
- caching
- transaction management
- ...

Een DbContext-class implementeert de interface **IDisposable**. Dit betekent dat je een DbContext-object moet 'opkuisen' na gebruik. Indien je het DbContext-object aanmaakt met het sleutelwoord `using`, dan gebeurt deze opkuis automatisch.

Intern gebruikt een DbContext databaseconnecties. Bij de opkuis van de DbContext sluit .NET deze connecties. Het is belangrijk de DbContext niet langer dan noodzakelijk levend te houden, zodat .NET connecties vlot kan sluiten.

Onze context bevat **DbSet<TEntity>** property's, één voor elke entity die gemapt wordt aan een tabel uit de database. Een DbSet is een collectie van objecten van een bepaalde entity.

We proberen dit uit in ons project.

- Maak in het project *Model* in de map *Entities* een class met de naam *EFOpleidingenContext* :

```
public class EFOpleidingenContext : DbContext           (1)
{
    public DbSet<Campus> Campussen { get; set; }         (2)
    public DbSet<Docent> Docenten { get; set; }
}
```

- (1) Je maakt een class die afgeleid is van DbContext.
- (2) Je maakt per entity class een property in je DbContext class. Deze property is van het type DbSet<TEntity>.

## 4.5 De connectionstring

Om een connectie te kunnen maken met een database hebben we 3 gegevens nodig : de naam van de databaseserver, de naam van de database en de usergegevens. Deze 3 gegevens gebruiken we eerder al op p. 13 om een connectie te maken met een database. In een programma kunnen we die gegevens niet intikken maar slaan we die op in een zogenaamde *connectionstring*.

Op de website <https://www.connectionstrings.com/> krijg je een overzicht van connectionstrings voor diverse databaseservers.

Je kan in een programma de connectionstring hardcoded opnemen in de code of opslaan in een configuratiebestand.

#### 4.5.1 De connectionstring hardcoded in de DbContext-class

We nemen de connectionstring hardcoded op in de DbContext-class.

- Voeg aan de class EFopleidingenContext.cs onderstaande method *OnConfiguring()* toe :

```
public class EFopleidingenContext : DbContext
{
    ...
    protected override void OnConfiguring(DbContextOptionsBuilder
        optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
            "Server=(localdb)\mssqllocaldb;Database=EFopleidingen;Trusted_Connection=true;" (1)
            ,options => options.MaxBatchSize(150)); (2) (3)
    }
}
```

- (1) Met UseSqlServer() geven we aan dat we een SQL Server database gaan gebruiken. Deze method geef je een connectionstring mee als parameter en eventuele options.
- (2) In de connectionstring geven we aan dat we localdb gebruiken als database server (Server=(localdb)\mssqllocaldb), EFopleidingen als database (Database=EFopleidingen) en dat we inloggen met de momenteel ingelogde windowsgebruiker (Trusted\_Connection=true).
- (3) Met de method MaxBatchSize() geven we het maximaal aantal SQL commando's aan dat in één keer naar de database kan gestuurd worden.

#### 4.5.2 De connectionstring in appsettings.json

We nemen de connectionstring op in een appsettings.json bestand. Op die manier kan je de connectionstring wijzigen zonder dat je de code moet hercompileren.

Om het bestand appsettings.json te kunnen inlezen moet je een extra NuGet package installeren, namelijk de NuGet package `Microsoft.Extensions.Configuration.Json`.

- Voeg versie 3.1.5 van deze package toe aan het project Model.

Je voegt nu de appsettings.json file toe.

- Klik met de rechtermuistoets op het project Model en kies Add-New Item...
- Kies in de categorie Web voor een JSON File en noem deze appsettings.json.
- Voeg er onderstaande code aan toe :

```
{
  "ConnectionStrings": {
    "EFopleidingen":
      "Server=(localdb)\mssqllocaldb;Database=EFopleidingen;Trusted_Connection=True;"
  }
}
```

Standaard wordt dit nieuwe configuratiebestand na een build van het project niet mee gekopieerd naar de outputdirectory. We passen dit aan.



- Selecteer in de Solution Explorer het bestand appsettings.json.
- Pas in het propertiesvenster de property Copy to Output Directory aan naar **Copy always**.

Je voegt nu de nodige code toe in de DbContext-class om de connectionstring in te lezen uit het appsettings.json bestand.

- Wijzig de class EFopleidingenContext.cs als volgt :

```
public class EFopleidingenContext : DbContext
{
    public static IConfigurationRoot configuration;                                (1)
    public DbSet<Campus> Campussen { get; set; }
    public DbSet<Docent> Docenten { get; set; }
    protected override void OnConfiguring(DbContextOptionsBuilder
        optionsBuilder)
    {
        configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetParent(AppContext.BaseDirectory).FullName)
            .AddJsonFile("appsettings.json", false)
            .Build();                                                            (2)
        var connectionString =
            configuration.GetConnectionString("efopleidingen");                (3)
        if (connectionString != null)
        {
            optionsBuilder.UseSqlServer(connectionString,                        (4)
                options => options.MaxBatchSize(150));
        }
    }
}
```

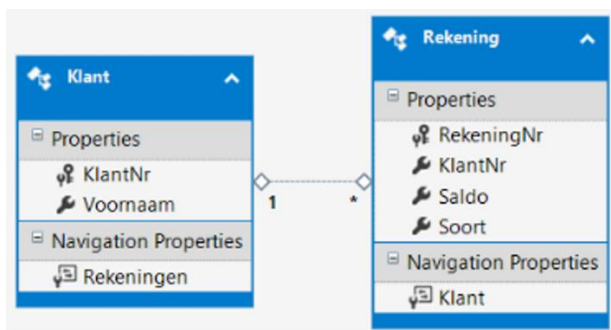
- (1) Om de configuratiegegevens op te halen gebruiken we een IConfigurationRoot.
- (2) We geven de naam en de plaats op van het appsettings.json bestand.
- (3) We halen de connectionstring uit het configuratiebestand.
- (4) En gebruiken deze string i.p.v. de hardcoded connectionstring.

Alle voorbereidingen zijn nu getroffen om de database te kunnen aanmaken. Dit doen we in het volgende hoofdstuk.

## 4.6 Oefening

Maak in een nieuwe solution *Taken* een console applicatie (UI), een testproject (Tests) en een class library (Model) aan zoals we in dit hoofdstuk hebben gedaan.

Maak in Model entityclasses voor de volgende entity's :



**Klant**

Property	type
KlantNr	integer
Voornaam	string

**Rekening**

Property	type
RekeningNr	string
KlantNr	integer
Saldo	decimal
Soort	char

Voeg ook de nodige navigationproperties toe. Tussen Klant en Rekening bestaat een één-veel-relatie. Een klant kan meerdere rekeningen hebben, een rekening behoort toe tot één klant.

Maak in de class *Rekening* ook een method *Storten*. Deze void method heeft een parameter *Bedrag* en verhoogt het saldo met dit bedrag.

Voeg ook de nodige packages toe.

Maak een DbContext class met daarin de nodige DbSet's en de connectionstring naar een database *EfBank* op (localdb)\mssqllocaldb.

## 5 Migrations

---

We beschikken nu over een Entity Data Model. Op basis van dit model kunnen op de database server een nieuwe database laten maken. Ook wanneer dit model wijzigt kunnen we de nodige wijzigingen in de database laten doorvoeren. Wijzigingen worden bewaard in een migration. Dit is een C#-script waarin code staat die de wijzigingen doorvoert (een method *Up*) of terug ongedaan (een method *Down*) maakt.

### 5.1 Een eerste migration

We beginnen met het maken van een eerste migration die ervoor zal zorgen dat de database wordt aangemaakt. We gebruiken hiervoor een tool. Met het commando **dotnet ef migrations add** zullen we een eerste migration aanmaken en die vervolgens uitvoeren met het commando **dotnet ef database update**.

We installeren eerst de dotnet ef tool.

- Open de NuGet Package Manager Console (*Tools – NuGet Package Manager – Package Manager Console*).
- Tik in de console het onderstaande commando en druk op Enter.

```
dotnet tool install --global dotnet-ef --version 3.1.5
```

De kans bestaat dat je na het uitvoeren van dit commando een foutmelding krijgt : “Tool ‘dotnet-ef’ is already installed.”. Dit commando is misschien ooit reeds uitgevoerd en moet slechts één keer op jouw toestel uitgevoerd worden.

Om deze tool te kunnen gebruiken moeten we nog een NuGet package toevoegen, namelijk de package *Microsoft.EntityFrameworkCore.Design*.

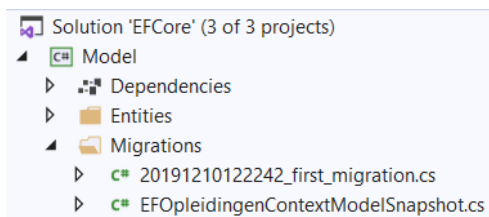
- Kies in het menu *Tools – NuGet Package Manager – Manage NuGet Packages for Solution*.
- Installeer de package *Microsoft.EntityFrameworkCore.Design* voor het project *Model*.
- Kies versie 3.1.5 en klik op *Install*.

We kunnen nu de tool gebruiken om een eerste migration aan te maken.

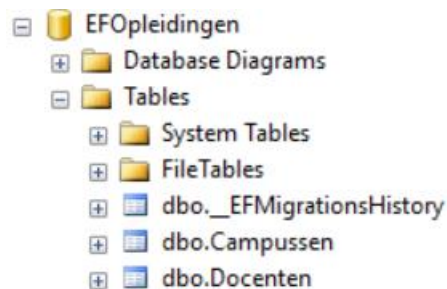
- Build de solution.
- Start de Package Manager Console.
- Normaal bevind je je nu in de folderstructuur in de map waar de solutionfile zich bevindt. Het commando dat de migration aanmaakt moet uitgevoerd worden in de root van het project *Model*. Controleer met het commando `dir` of je in deze folder bevindt. Zoniet verplaats je dan naar de folder *Model* met het commando `cd Model`
- Voer het onderstaande commando uit om de migration aan te maken :

```
dotnet ef migrations add first_migration
```

Het resultaat van deze opdracht is een bestand *EFOpleidingenContextModelSnapshot.cs* en een bestand waarvan de naam begint met de huidige datum en tijd en eindigt met de naam van de migration. Beide bestanden zijn opgeslagen in een folder *Migrations* in het project *Model*.



- Tik in de Package Manager Console het commando **dotnet ef database update** om de migration uit te voeren. De database wordt nu gemaakt.
- Start SSMS, refresh de databaselijst en bekijk het resultaat.



## 5.2 De tabelstructuur verfijnen met annotations

De dotnet ef tool heeft zich bij de aanmaak van de eerste migration gebaseerd op standaard conventies om de namen van de tabellen te kiezen, de namen en types van de columns, de keuze van de primary key's, enz.

Wil je van deze standaard conventies afwijken dan kan dat door gebruik te maken van annotations.

### 5.2.1 De naam van de tabel instellen

Standaard zal EF de naam van de DbSet in de DbContext gebruiken voor de naam van de bijhorende table in de database. Indien je hiervan wil afwijken dan gebruik je de annotation [Table] om de tabel een andere naam te geven.

Voorbeeld :

```
[Table("Naties")]
public class Land
{
    ...
}
```

### 5.2.2 De naam van een kolom instellen

De naam van de kolom die hoort bij een property is standaard dezelfde als de naam van de property. Vind je deze naam niet goed dan kan je deze instellen met het attribuut [Column].

Voorbeeld :

```
public class Docent
{
    ...
    [Column("Maandwedde")]
    public decimal Wedde { get; set; }
}
```

### 5.2.3 Een kolom instellen als (niet) verplicht in te vullen

Wil je de kolom, die bij een property hoort, instellen als verplicht in te vullen, dan voeg je het attribuut [Required] toe aan de property.

Een kolom die hoort bij een property met een primitief data-type is standaard verplicht in te vullen. Je kan hiervan afwijken door de property nullable te maken.

Voorbeeld :

```
public class Docent
{
    ...
    [Required]
    public string Voornaam { get; set; }
    public bool? HeeftRijbewijs { get; set; }
    ...
}
```

### 5.2.4 Het maximum aantal tekens in een varchar-kolom instellen

Je kan met het attribuut [StringLength] het maximum aantal tekens in een varchar-kolom instellen.

Voorbeeld :

```
public class Campus
{
    ...
    [StringLength(50)]
    public string Gemeente { get; set; }
    ...
}
```

### 5.2.5 Het kolomtype instellen

Je kan met het attribuut [Column] een afwijkend kolomtype instellen. Bij een DateTime property hoort standaard een datetime-kolom in de databasetabel. Je kan dit veranderen in een date-kolom via onderstaande notatie.

Voorbeeld :

```
public class Docent
{
    ...
    [Column(TypeName = "date")]
    public DateTime InDienst { get; set; }
    ...
}
```

### 5.2.6 De primary key instellen

Als EF een property vindt met de naam Id, aanziet EF deze property als de primary key. Vindt EF deze kolom niet maar wel een property waarvan de naam gelijk is aan de naam van de class, gevolgd door Id, dan aanziet EF deze property als de primary key.

Je kan hiervan afwijken door het attribuut [Key] te plaatsen voor de property die jij als primary key wil instellen.

Je kan bovendien instellen of de primary key al dan niet een autonummering veld wordt via het attribuut DatabaseGenerated. Je geeft deze de waarde DatabaseGenerated.None als je dat niet wil, of DatabaseGenerated.Identity als je dat wél wil.

Voorbeeld :

```
public class Land
{
    [Key, DatabaseGenerated(DatabaseGeneratedOption.None)]
    public string Code { get; set; }
    ...
}
```

### 5.2.7 De foreign key instellen

Wanneer jouw sleutels eindigen op Id zoals in ons Opleidingen-voorbeeld, bepaalt EF dus zelf welk veld de primary key is. EF doet hetzelfde voor de foreign key. Eindigt een potentiële foreign key niet op Id dan moet je zelf de foreign key aanduiden. Je doet dit door boven de navigation property een annotation te plaatsen met als parameter de naam van het foreign key veld.

Voorbeeld :

```
public class Inwoner
{
    [Key, DatabaseGenerated(DatabaseGeneratedOption.None)]
    public string RijksregisterNr { get; set; }
    ...
    public string Code { get; set; }
    [ForeignKey("Code")]
    public virtual Land Land { get; set; }
}
```

Er is een één-veel relatie tussen Land en Inwoner. In de class Inwoner geven we aan dat het veld Code de foreign key is door boven de navigation property Land de annotation `[ForeignKey("Code")]` te plaatsen.

Als alternatief kan je ook een ForeignKey-annotation plaatsen boven het foreign key veld. Als parameter geef je dan de naam van de navigation property mee.

### 5.2.8 Een property niet opnemen in de databasetable

Je kan aangeven dat een property niet moet opgenomen worden in de databasetable door er het attribuut `[NotMapped]` boven te plaatsen.

Voorbeeld :

```
public class Campus
{
    ...
    [NotMapped]
    public string NietOpnemen { get; set; }
}
```

### 5.2.9 Toepassing

We passen nu een aantal van deze attributen toe in ons voorbeeld.

- Wijzig de entiteiten Campus en Docent zoals hieronder weergegeven. Voeg ook de entiteit Land toe.

```

[Table("Campussen")]
public class Campus
{
    [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int CampusId { get; set; }
    [Required]
    [Column("CampusNaam")]
    public string Naam { get; set; }
    public string Straat { get; set; }
    public string Huisnummer { get; set; }
    public string Postcode { get; set; }
    [StringLength(50)]
    public string Gemeente { get; set; }
    [NotMapped]
    public string Commentaar { get; set; }
    public virtual ICollection<Docent> Docenten { get; set; }
}

[Table("Docenten")]
public class Docent
{
    [Key]
    public int DocentId { get; set; }
    [Required]
    [MaxLength(20)]
    public string Voornaam { get; set; }
    [Required]
    [MaxLength(30)]
    public string Familienaam { get; set; }
    [Column("Maandwedde", TypeName = "decimal(18,4)")]
    public decimal Wedde { get; set; }
    [Column(TypeName = "date")]
    public DateTime InDienst { get; set; }
    public bool? HeeftRijbewijs { get; set; }
    [ForeignKey("Land")]
    public string LandCode { get; set; }
    [ForeignKey("Campus")]
    public int CampusId { get; set; }
    public virtual Campus Campus { get; set; }
    public virtual Land Land { get; set; }
}

[Table("Landen")]
public class Land
{
    [Key, DatabaseGenerated(DatabaseGeneratedOption.None)]
    public string LandCode { get; set; }
    public string Naam { get; set; }
    public virtual ICollection<Docent> Docenten { get; set; }
}

```

- Voeg nu ook aan *EFOpleidingenContext.cs* een *DbSet<Land>* toe :

```

public class EFOpleidingenContext : DbContext
{
    public DbSet<Campus> Campussen { get; set; }
    public DbSet<Docent> Docenten { get; set; }
    public DbSet<Land> Landen { get; set; }
}

```

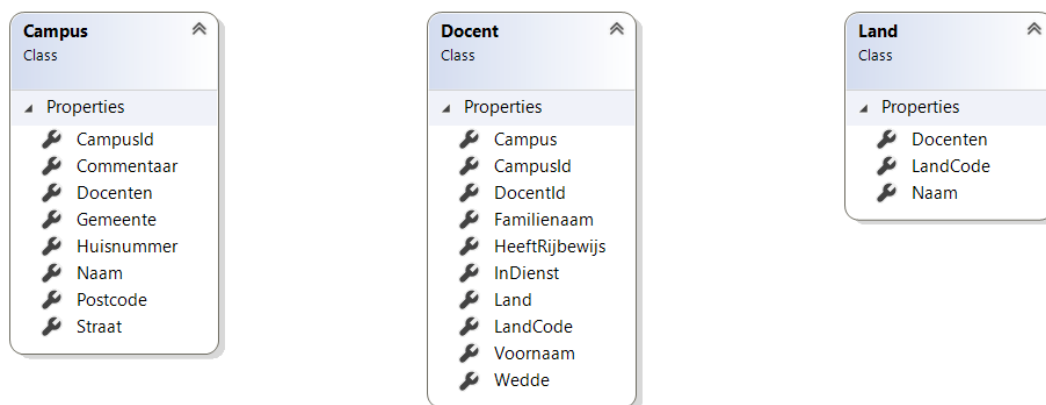
We voeren deze wijzigingen nu door in de database.

- Tik in de Package Manager Console onderstaand commando om een nieuwe migration aan te maken :  
`dotnet ef migrations add met_annotations`
- Voer de migration door met het commando `dotnet ef database update`

### 5.3 Class Diagram

Je kan de entiteiten ook visueel voorstellen in een Class Diagram.

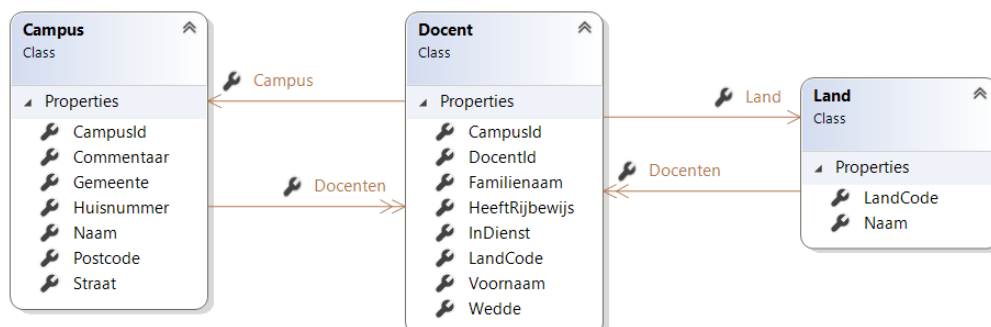
- Klik in de Solution Explorer met de rechtermuistoets op het project Model en kies Add-New Item.
- Tik in het zoekvenster rechts bovenaan de zoekterm *class diagram*.
- Selecteer een Class Diagram, geef deze eventueel een naam en klik op Add.
- Sleep nu de classes Campus, Docent en Land naar het class diagram.



Merk op dat hier zowel de gewone property's als de navigation property's door elkaar getoond worden. Je kan de navigation property's ook als pijlen tussen de entities voorstellen.

- Klik in de entiteit Campus met de rechtermuistoets op de property Docenten en kies Show as Collection Association. Er verschijnt een pijl van Campus naar Docent met een dubbele punt aan de kant van de entiteit Docent.
- Klik in de entiteit Docent met de rechtermuistoets op de property Campus en kies Show as Association. Er verschijnt een pijl van Docent naar Campus met een enkele punt aan de kant van de entiteit Campus.
- Voeg op een analoge manier associaties toe tussen Docent en Land.

Het resultaat :





## 5.4 Enums en partial classes

EF kan ook overweg met enums en partial classes. We proberen één en ander uit.

We voegen aan de class *Docent* een property *Geslacht* toe. Deze property is van het type *Geslacht*, een enum-type met *Man*, *Vrouw* en *X* als mogelijke waarden.

Daarnaast maken we van de class *Docent* een partial class. We voegen aan de class *Docent* in een tweede sourcefile, *DocentExtension.cs*, een property *Naam* toe die bestaat uit de voor- en familienaam van de docent en een property *Geboortedatum*.

- Maak van de class *Docent.cs* een partial class en voeg een property *Geslacht* toe :

```
public partial class Docent
{
    ...
    public Geslacht Geslacht { get; set; }
}
```

- Voeg in een nieuwe (partial) class *DocentExtension.cs* de property's *Naam* en *Geboortedatum* toe :

```
public partial class Docent
{
    public string Naam
    {
        get { return Voornaam + " " + Familienaam; }
    }
    [Column(TableName = "date")]
    public DateTime Geboortedatum { get; set; }
}
```

- Maak in het project *Model* in de folder *Entities* een nieuwe codefile *Geslacht.cs* aan en voeg er volgende code aan toe :

```
public enum Geslacht
{
    Man,
    Vrouw,
    X
}
```

We maken een nieuwe migration aan en voeren die door.

- Tik in de Package Manager Console het commando `dotnet ef migrations add geslachtsnaam`. Een nieuwe migration wordt aangemaakt.
- Voer de migration door met het commando `dotnet ef database update`

Bekijk het resultaat in SSMS : in de tabel *Docenten* is een veld *Geslacht* toegevoegd met als kolomtype een int. Daarnaast is er geen extra veld *Naam* toegevoegd maar wel een veld *Geboortedatum*.

### Tip

Je kan de laatst aangemaakte migration ook terug verwijderen met het commando :  
`dotnet ef migrations remove`

Met het commando `dotnet ef migrations script` genereer je een SQL script op basis van alle migrations.

## 5.5 Fluent API

Fluent API is een alternatief én een uitbreiding van annotations om de mapping tussen entityclasses en tabellen uit te drukken. Je configureert de mappings in de contextclass door de method *OnModelCreating()* te override. De configuratie die je via Fluent API uitdrukt heeft voorrang op eventuele annotations.

### 5.5.1 Property mapping

We starten met de mapping van property's. Zoals gezegd kunnen we het standaard gedrag van EF overschrijven door de mapping via code uit te drukken in de method *OnModelCreating()*.

Het instellen van de **primary key** gebeurt via conventie door de naam van de property te bekijken en vast te stellen of deze bestaat uit de naam van de entity en 'id'. Een alternatief is de annotation [Key]. Met fluent API doe je dit met de volgende code :

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Campus>()
        .HasKey(c => c.CampusId);
}
```

Met bovenstaande code geven we aan dat de entiteit Campus een sleutel heeft die bestaat uit de CampusId.

Willen we op een entiteit een **samengestelde sleutel** toepassen dan gaat dit met volgende code :

```
modelBuilder.Entity<Orderlijn>()
    .HasKey(o => new { o.Order, o.Orderlijn });
```

We kunnen via de annotation [DatabaseGenerated(...)] aangeven of op een sleutelveld al dan niet **autonummering** moet toegepast worden. Dit geven we aan in fluent API met volgende code :

```
modelBuilder.Entity<Land>()
    .Property(l => l.LandCode)
    .ValueGeneratedNever(); // geen autonummering

modelBuilder.Entity<Campus>()
    .Property(c => c.CampusId)
    .ValueGeneratedOnAdd(); // wel autonummering
```

De annotation [StringLength] kunnen we vervangen door de method **HasMaxLength()** :

```
modelBuilder.Entity<Docent>()
    .Property(d => d.Voornaam)
    .HasMaxLength(20);
```

Met de method **IsRequired()** geven we aan dat het veld verplicht is in te vullen :

```
modelBuilder.Entity<Docent>()
    .Property(d => d.Voornaam)
    .IsRequired();
```

Je kan op één of meerdere velden een index plaatsen zodat er sneller op kan gezocht worden. Je kan deze index bovendien uniek maken. Met onderstaande code voeg je een **unieke index** toe.

```
modelBuilder.Entity<Docent>()
    .HasIndex(d => d.Emailadres)
    .IsUnique();
```

Je kan met fluent API ook aangeven dat een property **niet gemapt** moet worden :

```
modelBuilder.Entity<Campus>()  
    .Ignore(c => c.Commentaar);
```

De method **HasColumnName** geeft een property een afwijkende veldnaam.

```
modelBuilder.Entity<Docent>()  
    .Property(d => d.Wedde)  
    .HasColumnName("Maandwedde");
```

Tenslotte kunnen we ook het gegevenstype aangeven met de method **HasColumnType()**.

```
modelBuilder.Entity<Docent>()  
    .Property(d => d.Wedde)  
    .HasColumnType("decimal(18, 4)");
```

### 5.5.2 Type mapping

Met type mapping overschrijven we het standaard gedrag van EF Core voor de mapping van een class.

Onderstaande code zorgt ervoor dat een **entity** Test **niet** wordt **gemapt** :

```
modelBuilder.Ignore<Test>();
```

We geven een tabel een afwijkende naam met volgende code :

```
modelBuilder.Entity<Campus>().ToTable("Campussen");
```

## 5.6 Oefening

Creëer in de solution *Taken* op basis van het entity data model de tabellen Banken en Rekeningen in de database EFBank. Voor de veldeigenschappen en de mapping kan je een mix gebruiken van de standaard conventies, annotations en fluent api.

### Klant

Property	type	Omschrijving
KlantNr	integer	PK
Voornaam	string	Not null

### Rekening

Property	type	Omschrijving
RekeningNr	string	PK
KlantNr	integer	FK, not null
Saldo	decimal	not null
Soort	char	not null

## 5.7 Seeding

We kunnen de tabellen van de database via code voorzien van initiële gegevens. Dit kunnen eventueel ook testgegevens zijn. We noemen dit seeding van de database.

Om één of meerdere tabellen te seeden voeg je code toe in de contextclass in de method *OnModelCreating()*. Je past de method *HasData()* toe op de modelBuilder om een record toe te voegen aan een tabel.

Opgelet : sleutelwaarden moeten altijd expliciet meegegeven worden, ook wanneer er autonummering voorzien is voor een veld.

- Voeg onderstaande code toe in de contextclass *EFOpleidingenContext.cs* :

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Campus>().HasData(
        new Campus { CampusId = 1, Naam = "Andros", Straat = "Somersstraat",
            Huisnummer = "22", Postcode = "2018", Gemeente = "Antwerpen" },
        new Campus { CampusId = 2, Naam = "Delos", Straat = "Oude Vest",
            Huisnummer = "17", Postcode = "9200", Gemeente = "Dendermonde" },
        new Campus { CampusId = 3, Naam = "Gavdos", Straat = "Europalaan",
            Huisnummer = "37", Postcode = "3600", Gemeente = "Genk" },
        new Campus { CampusId = 4, Naam = "Hydra", Straat = "Interleuvenlaan",
            Huisnummer = "2", Postcode = "3001", Gemeente = "Heverlee" },
        new Campus { CampusId = 5, Naam = "Ikaria", Straat = "Vlamingstraat",
            Huisnummer = "10", Postcode = "8560", Gemeente = "Wevelgem" },
        new Campus { CampusId = 6, Naam = "Oinousses", Straat = "Akkerstraat",
            Huisnummer = "4", Postcode = "8400", Gemeente = "Oostende" }
    );

    modelBuilder.Entity<Land>().HasData(
        new Land { LandCode = "BE", Naam = "België" },
        new Land { LandCode = "NL", Naam = "Nederland" },
        new Land { LandCode = "DE", Naam = "Duitsland" },
        new Land { LandCode = "FR", Naam = "Frankrijk" },
        new Land { LandCode = "IT", Naam = "Italië" },
        new Land { LandCode = "LU", Naam = "Luxemburg" }
    );

    modelBuilder.Entity<Docent>().HasData(
        new Docent { DocentId = 001, Voornaam = "Willy", Familienaam =
            "Abbeloos", Wedde = 1400m, HeeftRijbewijs = new Nullable<bool>(),
            InDienst = new DateTime(2019, 1, 1), CampusId = 4 },
        new Docent { DocentId = 002, Voornaam = "Joseph", Familienaam =
            "Abelshausen", Wedde = 1800m, HeeftRijbewijs = true, InDienst =
            new DateTime(2019, 1, 2), CampusId = 2 },
        new Docent { DocentId = 003, Voornaam = "Joseph", Familienaam =
            "Achten", Wedde = 1300m, HeeftRijbewijs = false, InDienst =
            new DateTime(2019, 1, 3), CampusId = 3 },
        new Docent { DocentId = 004, Voornaam = "François", Familienaam =
            "Adam", Wedde = 1700m, HeeftRijbewijs = new Nullable<bool>(),
            InDienst = new DateTime(2019, 1, 4), CampusId = 1 },
        ...
        new Docent { DocentId = 309, Voornaam = "Jozef", Familienaam =
            "Wouters", Wedde = 1100, HeeftRijbewijs = true, InDienst =
            new DateTime(2019, 11, 7), CampusId = 1 }
    );
}
```

Hierboven zijn slechts een aantal docentgegevens afgebeeld. Je vindt de code voor de volledige lijst in het bestand *EFopleidingenContext.cs* bij de oefenbestanden. Neem ze zeker allemaal over, je hebt ze later nodig om een aantal query's uit te proberen.

We voegen nu een migration toe die deze data zal toevoegen in de database en updaten daarna de database.

- Tik in de Package Manager Console het commando `dotnet ef migrations add seeding`. Een nieuwe migration wordt aangemaakt.
- Voer de migration door met het commando `dotnet ef database update`
- Je kan de inhoud van de tabellen bekijken in SSMS.

De tabellen beschikken nu over de gewenste startdata. Wanneer je de data in de method *OnModelCreating()* verandert door gegevens te wijzigen, te verwijderen of toe te voegen en een nieuwe migration aanmaakt dan worden de gegevens in de bijhorende databasetabellen navenant aangepast. Wis dus zeker niet zomaar alle data in *OnModelCreating()* of alle gegevens worden bij een volgende migration gewist !

We proberen enkele wijzigingen uit.

- Voeg in de lijst met landen Groot-Brittannië toe met landcode GB :

```
modelBuilder.Entity<Land>().HasData(  
    ...  
    new Land { LandCode = "GB", Naam = "Groot-Brittannië" }  
);
```

- Schrap de code voor docent 309, Jozef Wouters in de lijst met docenten :

```
modelBuilder.Entity<Docent>().HasData(  
    ...  
    new Docent { DocentId = 309, Voornaam = "Jozef", ... }  
);
```

- Wijzig de gemeentenaam voor campus 2, Delos, in Wondelgem, de postcode in 9032 :

```
modelBuilder.Entity<Campus>().HasData(  
    ...  
    new Campus  
    {  
        CampusId = 2,  
        Naam = "Delos",  
        Straat = "Oude Vest",  
        Huisnummer = "17",  
        Postcode = "9032",  
        Gemeente = "Wondelgem"  
    },  
    ...  
);
```

- Tik in de Package Manager Console het commando `dotnet ef migrations add seeding_change`. Een nieuwe migration wordt aangemaakt.
- Bekijk de inhoud van de nieuwe migrationsfile in de folder *Migrations*. Enkel de wijzigingen t.o.v. de vorige migration zijn opgenomen.
- Voer de migration door met het commando `dotnet ef database update`

Wanneer je de inhoud van een **sleutelveld** verandert moet je oppassen. De migration wijzigt enkel de waarde in de primary key maar niet de foreign key in gerelateerde tabellen. Méér nog : de gerelateerde records in de docententabel worden eerst verwijderd en vervolgens terug toegevoegd! Dit komt omdat de campus met id 6 eerst wordt verwijderd en daarvoor mogen er geen bijhorende docenten bestaan. Die worden dus ook gewist. Na de verwijdering van de campus wordt deze terug toegevoegd met de correcte id. De gewiste docenten worden terug toegevoegd maar met de oude id!

- Wijzig de campusid voor campus 6, Oinouses, in 7 :

```
modelBuilder.Entity<Campus>().HasData(
    ...
    new Campus
    {
        // CampusId = 6
        CampusId = 7,
        Naam = "Oinouses",
        ...
        Gemeente = "Oostende"
    }
);
```

- Tik in de Package Manager Console het commando `dotnet ef migrations add campus6naar7`. Een nieuwe migration wordt aangemaakt.
- Bekijk de inhoud van de nieuwe migrationsfile in de folder *Migrations*.
- We voeren deze migration NIET door maar wissen deze met `dotnet ef migrations remove`
- Verander in *OnModelCreating()* de campusid van campus Oinouses terug naar 6.

## 5.8 Oefening

Voeg in EFBank via seeding onderstaande data toe aan de tabellen Klanten en Rekeningen.

Klanten:

KlantNr	KlantVoornaam
1	Marge
2	Homer
3	Lisa
4	Maggie
5	Bart

Rekeningen:

RekeningNr	KlantNr	Saldo	Soort
123-4567890-02	1	1000	Z
234-5678901-69	1	2000	S
345-6789012-12	2	500	S

## 6 SQL Logging

---

Door EF Core te gebruiken weten we niet altijd precies welke SQL commando's in de database worden uitgevoerd. Op het eerste zicht is dat geen probleem want het is dan ook de bedoeling dat EF Core het opstellen van SQL Commando's van ons overneemt.

Wil je toch weten wat er precies achter de schermen gebeurt dan zijn er twee opties :

- Logging in Visual Studio Console
- Logging via de SQL Server Profiler

### 6.1 Logging in Visual Studio

Om aan logging te kunnen doen via de Visual Studio Console moet je eerst een NuGet package installeren, namelijk `Microsoft.Extensions.Logging.Console`.

- Kies in het menu voor *Tools – NuGet Package Manager – Manage NuGet Packages for Solution...*
- Klik links bovenaan op *Browse...*
- In het zoekveld tik je : *microsoft.extensions.logging.console*
- Zoek in de lijst de package *Microsoft.Extensions.Logging.Console* en klik erop.
- Vink ernaast het project *Model* aan en kies naast *Version:* voor versie 3.1.5. Klik op *Install*.
- Aanvaard de licentievoorwaarden met een klik op *I Accept*.

In een dotnet-core programma kan je een beroep doen op services uit een servicecollection. Logging is eveneens zo een service en daarom voegen we deze toe aan de servicecollection. Dit gebeurt in het contextbestand.

- Voeg in *EFOpleidingenContext.cs* onderstaande vetgedrukte code toe :

```
...  
using Microsoft.Extensions.DependencyInjection;  
using Microsoft.Extensions.Logging;  
  
namespace Model.Entities  
{  
    public class EFOpleidingenContext : DbContext  
    {  
        public DbSet<Campus> Campussen { get; set; }  
        public DbSet<Docent> Docenten { get; set; }  
        public DbSet<Land> Landen { get; set; }  
  
        private ILoggerFactory GetLoggerFactory()  
        {  
            IServiceCollection serviceCollection = new ServiceCollection();  
  
            serviceCollection.AddLogging (  
                builder => builder.AddConsole()  
                    .AddFilter(DbLoggerCategory.Database.Command.Name,  
                        LogLevel.Information)  
            );  
  
            return serviceCollection.BuildServiceProvider()  
                .GetService<ILoggerFactory>();  
        }  
    }  
}
```

```
protected override void OnConfiguring(DbContextOptionsBuilder
    optionsBuilder)
{
    ...
    if (connectionString != null)
    {
        optionsBuilder.UseSqlServer(connectionString,
            options => options.MaxBatchSize(150))
            .UseLoggerFactory(GetLoggerFactory())           (1)
            .EnableSensitiveDataLogging(true);              (2)
    }
}
```

(1) We gebruiken de `Iloggerfactory` die in de method `GetLoggerFactory()` wordt aangemaakt.

(2) Ook gevoelige data zoals de parameters van de SQL commando's wordt getoond bij de logging.

In het verdere verloop van deze cursus zullen we regelmatig het effect van deze logging bekijken.

## 6.2 Logging in SQL Server Profiler

Je kan ook aan logging doen via het programma SQL Server Profiler.

- Zoek in het Windows Start Menu naar *SQL Server Profiler* en start het programma.
- Kies vervolgens in het menu *File – New Trace...*
- Maak een connectie met de database server *LocalDb*.
- In het *Trace Properties* window dat verschijnt tik je bij *Trace name*: een naam voor de trace.
- Klik vervolgens op *Run*. Je krijgt dan een venster te zien waarin je al het verkeer van de SQL Server te zien waarop je ingelogd bent.

We proberen dit even uit.

- Ga naar SSMS en tik in een nieuw queryvenster het commando `select * from campussen`
- Bekijk het resultaat in SQL Server Profiler.
- Je kan de trace stoppen.

## 6.3 Oefening

Pas ook logging toe via Visual Studio in de solution *Taken*.



## 7 De user interface

---

Nu de database aangemaakt is en voorzien van data kunnen we deze in een programma gebruiken. We kunnen gegevens lezen, toevoegen, wijzigen en verwijderen via een console applicatie, een webapplicatie, ... In deze cursus ligt de focus op EF Core, dus gebruiken we een eenvoudige console applicatie als interface.

### 7.1 Dependencies

In het project UI maken we gebruik van de class library *Model*. Hiervoor moeten we een reference toevoegen.

- Klik met de rechtermuistoets op het project *UI* en kies *Add-Project Reference...*
- Vink het project *Model* aan en klik op *OK*.
- Maak bovendien, mocht dit nog niet het geval zijn, van het project UI het opstartproject.

### 7.2 Oefening

Voorzie de consoleapplicatie in de solution *Taken* van de nodige references.



## 8 Query's

In dit hoofdstuk zullen we aan de hand van query's gegevens opzoeken in de database EFopleidingen. We gebruiken hiervoor de dbcontext-class EFopleidingenContext. Deze bevat de property's *Campussen*, *Docenten* en *Landen* waarmee je de gelijknamige tabellen uit de database mee raadpleegt.

### 8.1 Een eenvoudig voorbeeld

In een eerste voorbeeld overlopen we alle docenten. We doen dit door op de property *Docenten* uit de class *EFopleidingenContext* een foreach-iteratie uit te voeren. EF Core zet dan volgende stappen :

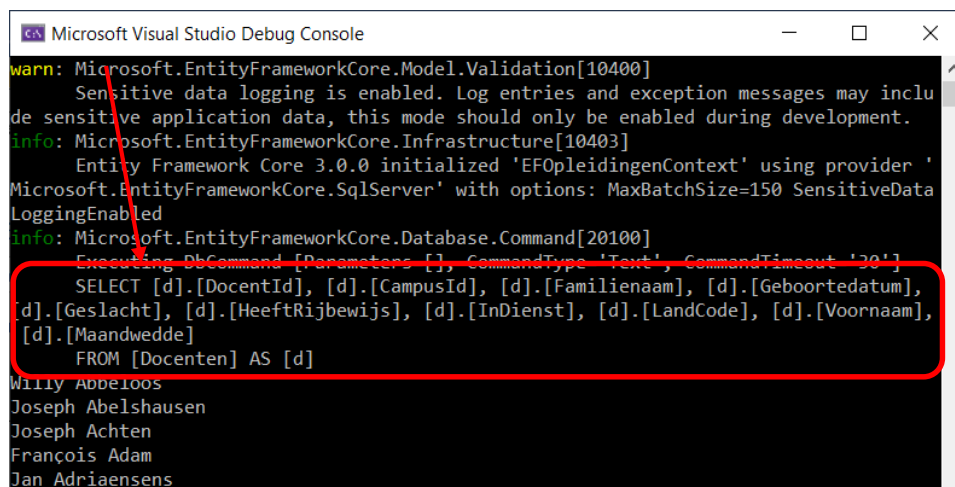
- EF Core stuurt een SQL-select statement naar de database om álle records uit de bijhorende tabel *Docenten* op te vragen
- EF Core maakt van ieder record een *Docent*-object
- EF Core verzamelt deze *Docent*-objecten in de property *Docenten* van het object *EFopleidingenContext*.

We proberen dit uit in het UI project.

- Neem onderstaande code over in *Program.cs* :

```
static void Main(string[] args)
{
    using (var context = new EFopleidingenContext())
    {
        foreach (var docent in context.Docenten)
        {
            Console.WriteLine(docent.Naam);
        }
    }
}
```

Je kan het programma nu uitproberen. Bemerkt dat je helemaal bovenaan het console-venster het SQL statement kan zien via de logging :



```
Microsoft Visual Studio Debug Console
warn: Microsoft.EntityFrameworkCore.Model.Validation[10400]
      Sensitive data logging is enabled. Log entries and exception messages may include sensitive application data, this mode should only be enabled during development.
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 3.0.0 initialized 'EFopleidingenContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: MaxBatchSize=150 SensitiveDataLoggingEnabled
info: Microsoft.EntityFrameworkCore.Database.Command[20100]
      Executing DbCommand [Parameters [], CommandType 'Text', CommandTimeout '30']
      SELECT [d].[DocentId], [d].[CampusId], [d].[Familiennaam], [d].[Geboortedatum], [d].[Geslacht], [d].[HeeftRijbewijs], [d].[InDienst], [d].[LandCode], [d].[Voornaam], [d].[Maandwedde]
      FROM [Docenten] AS [d]
Willi Abbeels
Joseph Abelshausen
Joseph Achten
François Adam
Jan Adriaenssens
```

## 8.2 LINQ-query's

Records lezen met een foreach zoals in de vorige paragraaf is beperkt in zijn mogelijkheden : je kan niet sorteren of filteren. We kunnen wel een LINQ query uitvoeren op de property Docenten. EF Core zet dan volgende stappen :

- EF Core vertaalt de LINQ query naar een SQL select statement
- EF Core stuurt dit SQL statement naar de database om records uit de bijhorende tabel op te vragen
- EFCore maakt van ieder gevonden record een entity
- EFCore verzamelt deze entity's in een verzameling
- Je kan met een foreach itereren over deze verzameling

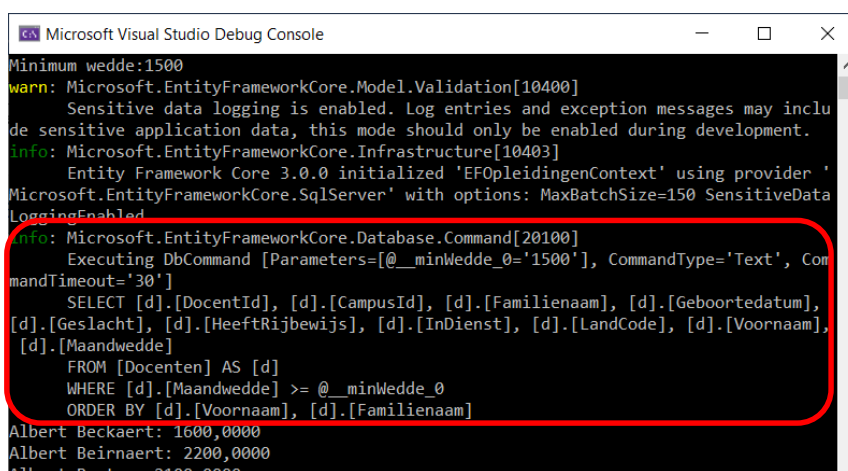
We proberen dit uit.

- Zet de aanwezige code in de Main()-method in *Program.cs* in commentaar.
- Voeg in *Main()* onderstaande code toe :

```
Console.WriteLine("Minimum wedde:");
if (decimal.TryParse(Console.ReadLine(), out decimal minWedde))
{
    using (var context = new EFopleidingenContext())
    {
        var query = from docent in context.Docenten
                    where docent.Wedde >= minWedde
                    orderby docent.Voornaam, docent.Familienaam
                    select docent;

        foreach (var docent in query)
        {
            Console.WriteLine("{0}: {1}", docent.Naam, docent.Wedde);
        }
    }
}
else
    Console.WriteLine("Geef een getal in !");
```

- Voeg bovenaan de code onderstaand using-statement toe :
- ```
using System.Linq;
```
- Probeer het programma uit en bekijk het SQL statement via de logging-informatie.



```
Microsoft Visual Studio Debug Console
Minimum wedde:1500
warn: Microsoft.EntityFrameworkCore.Model.Validation[10400]
      Sensitive data logging is enabled. Log entries and exception messages may include sensitive application data, this mode should only be enabled during development.
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 3.0.0 initialized 'EFopleidingenContext' using provider 'Microsoft.EntityFrameworkCore.SqlServer' with options: MaxBatchSize=150 SensitiveDataLoggingEnabled
info: Microsoft.EntityFrameworkCore.Database.Command[20100]
      Executing DbCommand [Parameters=[@__minWedde_0='1500'], CommandType='Text', CommandTimeout='30']
      SELECT [d].[DocentId], [d].[CampusId], [d].[Familienaam], [d].[Geboortedatum], [d].[Geslacht], [d].[HeeftRijbewijs], [d].[InDienst], [d].[LandCode], [d].[Voornaam], [d].[Maandwedde]
      FROM [Docenten] AS [d]
      WHERE [d].[Maandwedde] >= @__minWedde_0
      ORDER BY [d].[Voornaam], [d].[Familienaam]
Albert Beckaert: 1600,0000
Albert Beirnaert: 2200,0000
Albert Bonten: 2100,0000
```

### 8.3 Query-methods

Naast LINQ kan je ook query-methods zoals Where en OrderBy gebruiken om een query te definiëren.

- Wijzig in het programma de query-definitie als volgt :

```
var query = context.Docenten
    .Where(docent => docent.Wedde >= minWedde)    (1)
    .OrderBy(docent => docent.Voornaam)           (2)
    .ThenBy(docent => docent.Familienaam);        (3)
```

- (1) Je gebruikt de method Where om records te filteren. Je geeft een lambda-expressie mee, waarin je de filter definieert.
- (2) Je gebruikt de method OrderBy om records te sorteren. Je geeft een lambda-expressie mee, waarin je sortering definieert.
- (3) Als je op meerdere properties wil sorteren (in dit voorbeeld op voornaam én familienaam), pas je op het resultaat van de OrderBy-method de method ThenBy toe.

- Probeer opnieuw uit.

De methods OrderBy en ThenBy sorteren oplopend. De methods OrderByDescending en ThenByDescending sorteren aflopend.

### 8.4 LINQ-query's vs. query methods

LINQ-query's zijn meestal leesbaarder dan query's met querymethods.

Sommige programmaonderdelen zijn meer onderhoudbaar met query-methods dan met LINQ-queries. In het volgende voorbeeld ziet de gebruiker de docenten met een wedde vanaf een in te tikken grens. De gebruiker kiest daarna hoe hij die docenten sorteert.

We tonen eerst het programma met een LINQ-query's. Het bevat drie sterk gelijkaardige query's.

```
static void Main(string[] args)
{
    Console.WriteLine("Minimum wedde:");
    if (decimal.TryParse(Console.ReadLine(), out decimal minWedde))
    {
        Console.WriteLine("Sorteren:1=op wedde, 2=op familienaam, 3=op voornaam:");
        var sorterenOp = Console.ReadLine();

        using (var context = new EF0pleidingenContext())
        {
            IQueryable<Docent> query;                                (1)
            switch (sorterenOp) {
                case "1":
                    query = from docent in context.Docenten          (2)
                           where docent.Wedde >= minWedde
                           orderby docent.Wedde
                           select docent;

                    break;
                case "2":
                    query = from docent in context.Docenten          (2)
                           where docent.Wedde >= minWedde
                           orderby docent.Familienaam
                           select docent;

                    break;
```

```

        case "3":
            query = from docent in context.Docenten
                    where docent.Wedde >= minWedde
                    orderby docent.Voornaam
                    select docent;
            break;
        default:
            Console.WriteLine("Verkeerde keuze");
            query = null;
            break;
    }
    if (query != null)
        foreach (var docent in query)
            Console.WriteLine("{0}: {1}", docent.Naam, docent.Wedde);
    else
        Console.WriteLine("U tikte geen getal");
}
}
}

```

(1) Het type van de variabele *query* is een LINQ query die Docent-entity's teruggeeft.

(2) De drie query's in het switch-statement lijken sterk op elkaar.

De versie met query-methods bevat maar één query-definitie :

```

static void Main(string[] args)
{
    Console.Write("Minimum wedde:");
    if (decimal.TryParse(Console.ReadLine(), out decimal minWedde))
    {
        Console.Write("Sorteren:1=op wedde, 2=op familienaam, 3=op voornaam:");
        var sorterenOp = Console.ReadLine();
        Func<Docent, Object> sorteerLambda;
        switch (sorterenOp) {
            case "1":
                sorteerLambda = (docent) => docent.Wedde;
                break;
            case "2":
                sorteerLambda = (docent) => docent.Familienaam;
                break;
            case "3":
                sorteerLambda = (docent) => docent.Voornaam;
                break;
            default:
                Console.WriteLine("Verkeerde keuze");
                sorteerLambda = null;
                break;
        }
        if (sorteerLambda != null)
        {
            using (var context = new EFopleidingenContext())
            {
                var query = context.Docenten
                    .Where(docent => docent.Wedde >= minWedde)
                    .OrderBy(sorteerLambda);
                foreach (var docent in query)
                    Console.WriteLine("{0}: {1}", docent.Naam, docent.Wedde);
            }
        }
    }
}

```

```
        else
            Console.WriteLine("U tikte geen getal");
    }
}
```

- (1) We definiëren een Func-delegate.
- (2) We kennen er telkens een andere lambda aan toe.

## 8.5 Een entity zoeken op basis van de PK-waarde

Je hoeft geen LINQ-query te schrijven om een entity te zoeken op zijn primary-key-waarde. Voor zo'n zoekoperatie kan je de Find-method gebruiken. Deze method retourneert de gezochte entiteit als de meegegeven keywaarde gevonden wordt, zoniet dan wordt er een null-waarde teruggegeven.

In onderstaand voorbeeld gebruiken we de Find-method op de property Docenten, om een docent te zoeken op zijn docentnummer.

- Neem onderstaande code over :

```
static void Main(string[] args)
{
    using (var context = new EF0pleidingenContext())
    {
        Console.Write("DocentNr. :");
        if (int.TryParse(Console.ReadLine(), out int docentNr))
        {
            var docent = context.Docenten.Find(docentNr);
            Console.WriteLine(docent == null ? "Niet gevonden" : docent.Naam);
        }
        else
            Console.WriteLine("U tikte geen getal");
    }
}
```

- (1) Je geeft het docentnummer mee aan de Find-method.
- (2) Afhankelijk van het resultaat geef je de tekst "Niet gevonden" weer of de docentnaam.

- Probeer uit.

## 8.6 Gedeeltelijke objecten ophalen

De query's die je tot nu toe maakte, lezen uit de records alle kolommen en vullen hiermee per entity alle bijbehorende property's. Dit kan de performantie benadelen als je in een programma-onderdeel slechts enkele property's per entity nodig hebt.

Je gebruikt als oplossing een LINQ-query, waarin je slechts enkele property's opvraagt. EF Core vertaalt zo'n LINQ-query naar een SQL-select-statement dat enkel de kolommen leest die bij die property's horen.

- Neem onderstaand voorbeeld over :

```
static void Main(string[] args)
{
    using (var context = new EF0pleidingenContext())
    {
        var query = from campus in context.Campussen
                    orderby campus.Naam
                    select new { campus.CampusId, campus.Naam };
    }
}
```

```

        foreach (var campusDeel in query)
            Console.WriteLine("{0}: {1}", campusDeel.CampusId,
                               campusDeel.Naam);
    }
}

```

(1) Het resultaat van deze query is een verzameling objecten. Het type van deze objecten is een anonieme tijdelijke class met twee property's: CampusId en Naam.

- Probeer uit.

Je kan deze query ook uitschrijven met query-methods. De code wordt dan :

```

var query = context.Campussen
    .OrderBy(campus => campus.Naam)
    .Select(campus => new { campus.CampusId, campus.Naam });

```

## 8.7 Groeperen in query's

Je kan in een query de objecten groeperen. Je gebruikt hiervoor de sleutelwoorden group, by en into.

In onderstaand voorbeeld groepeer je de docenten op voornaam.

- Neem onderstaande code over :

```

static void Main(string[] args)
{
    using (var context = new EF0pleidingenContext())
    {
        var query = from docent in context.Docenten
                    group docent by docent.Voornaam
                    into voornaamGroep
                    select new {
                        Voornaam = voornaamGroep.Key,
                        Aantal = voornaamGroep.Count()
                    };

        foreach (var voornaamStatistiek in query)
        {
            Console.Write(voornaamStatistiek.Voornaam + ": ");
            Console.WriteLine(voornaamStatistiek.Aantal + " keer.");
        }
    }
}

```

(1)

(1) Het resultaat van deze query is een verzameling objecten. Deze hebben twee property's : een voornaam (de key waarop gegroepeerd werd) en het aantal records per groep.

- Probeer uit.



## 8.8 Lazy en Eager loading

Wanneer je de gegevens van docenten ophaalt heb je soms ook de gegevens van de bijhorende campussen nodig. Je kan deze bijkomende gegevens meteen inlezen of ze uitgesteld lezen. Het ene heet Eager Loading het andere Lazy Loading.

### 8.8.1 Lazy Loading

Bij Lazy Loading wordt het ophalen van de gegeven uit de databases dus uitgesteld tot het ogenblik waarop ze echt nodig zijn. Lazy Loading is niet standaard. Je moet expliciet aangeven dat je Lazy Loading wilt gebruiken.

Lazy Loading heeft het voordeel dat je minder grote hoeveelheden gegevens in één keer inleest. Aan de andere kant betekent dit dan ook dat je meerdere requests naar de database stuurt.

Je kan Lazy Loading op twee manieren implementeren : mét of zonder proxies. In het laatste geval gebruik je dependency injection (ILazyLoader). De methode mét proxies werkt niet bij UWP, Xamarin en ASP.NET.

#### 8.8.1.1 Met gebruik van proxy's

We installeren eerst de NuGet package *Microsoft.EntityFrameworkCore.Proxies*.

- Start de NuGet package manager en installeer voor het project *Model* de package *Microsoft.EntityFrameworkCore.Proxies*. Kies versie 3.1.5.

We activeren de proxies in de method *OnConfiguring()* in de class *EFOpleidingenContext*.

- Open het bestand en voeg onderstaande vetgedrukte regel toe in de method *OnConfiguring()* :

```
protected override void OnConfiguring(DbContextOptionsBuilder
    optionsBuilder)
{
    ...
    optionsBuilder
        .UseSqlServer(connectionString,
            options => options.MaxBatchSize(150))
        .UseLoggerFactory(GetLoggerFactory())
        .EnableSensitiveDataLogging(true)
        .UseLazyLoadingProxies();
    ...
}
```

Daarnaast moeten we alle navigation properties virtual maken.

- Controleer of in onderstaande classes de navigation properties virtual zijn.

In de class *Docent* :

```
public partial class Docent
{
    ...
    public int CampusId { get; set; }
    public virtual Campus Campus { get; set; }
}
```

In de class Campus :

```
public class Campus
{
    ...
    public virtual ICollection<Docent> Docenten { get; set; }
}
```

We proberen lazy loading nu uit.

- Neem onderstaande code over in *Main()* :

```
using (var context = new EFopleidingenContext())
{
    Console.WriteLine("Voornaam:");
    var voornaam = Console.ReadLine();

    var query = from docent in context.Docenten
                where docent.Voornaam == voornaam
                select docent;                                     (1)

    foreach (var docent in query)
        Console.WriteLine("{0} : {1}", docent.Naam, docent.Campus.Naam); (2)
}
```

- (1) Je leest enkel de docent-entity's.
- (2) Je toont voor elke docent zijn naam en de campusnaam. Deze is nog niet beschikbaar dus stuurt EF Core een SQL-statement naar de database om de campus in te lezen.
- Probeer uit. Je krijgt je een foutmelding.

Oorzaak van de fout is dat je meerdere zogenaamde resultsets tegelijk hebt open staan. Eén voor de docenten en per docent nog eens een extra resultset voor de bijhorende campus. Als we dit willen doen moeten we een extra attribuut *multipleactiveresultsets* toevoegen aan de connectionstring en deze de waarde *true* geven.

- Wijzig de connectionstring in *EFopleidingenContext.cs* als volgt :

```
"Server=(localdb)\mssqllocaldb;Database=EFopleidingen;
Trusted_Connection=true;MultipleActiveResultSets=true"
```

- Probeer opnieuw uit.

Je ziet via de logging duidelijk de verschillende SQL-statements. De docenten worden eerst ingelezen. Daarna wordt elke campus waarvoor info nodig is en die nog niet werd gelezen één keer ingelezen.

```
FROM [Campussen] AS [c]
WHERE ([c].[CampusId] = @_p_0) AND @_p_0 IS NOT NULL
Jean Aerts : Delos
info: Microsoft.EntityFrameworkCore.Database.Command[20100]
      Executing DbCommand [Parameters=[@_p_0='5'], CommandType='Text', CommandTimeout='30']
      SELECT [c].[CampusId], [c].[Gemeente], [c].[Huisnummer], [c].[CampusNaam], [c].[Postcode], [c].[Straat]
      FROM [Campussen] AS [c]
      WHERE ([c].[CampusId] = @_p_0) AND @_p_0 IS NOT NULL
Jean Belvaux : Ikaria
Jean Bogaerts : Delos
info: Jean Brankart : Oinouses
: Microsoft.EntityFrameworkCore.Database.Command[20100]
      Executing DbCommand [Parameters=[@_p_0='6'], CommandType='Text', CommandTimeout='30']
      SELECT [c].[CampusId], [c].[Gemeente], [c].[Huisnummer], [c].[CampusNaam], [c].[Postcode], [c].[Straat]
      FROM [Campussen] AS [c]
      WHERE ([c].[CampusId] = @_p_0) AND @_p_0 IS NOT NULL
Jean Vliegen : Andros
info: Microsoft.EntityFrameworkCore.Database.Command[20100]
      Executing DbCommand [Parameters=[@_p_0='1'], CommandType='Text', CommandTimeout='30']
      SELECT [c].[CampusId], [c].[Gemeente], [c].[Huisnummer], [c].[CampusNaam], [c].[Postcode], [c].[Straat]
      FROM [Campussen] AS [c]
      WHERE ([c].[CampusId] = @_p_0) AND @_p_0 IS NOT NULL
```

### 8.8.1.2 Met dependency injection

We kunnen lazy loading ook activeren via dependency injection. Hiervoor installeren we eerst de NuGet package *Microsoft.EntityFrameworkCore.Abstractions*.

- Start de NuGet package manager en installeer voor het project *Model* de package *Microsoft.EntityFrameworkCore.Abstractions*. Kies versie 3.1.5.

We injecteren in de class *Campus* een *ILazyLoader*.

- Wijzig de class *Campus* als volgt :

```
public class Campus
{
    private readonly ILazyLoader lazyLoader;
    public Campus(ILazyLoader lazyLoader)           (1)
    {
        this.lazyLoader = lazyLoader;
    }
    public Campus()                                (2)
    {
        Docenten = new List<Docent>();
    }
    ...

    //public virtual ICollection<Docent> Docenten { get; set; }   (3)
    private ICollection<Docent> docenten;
    public ICollection<Docent> Docenten             (4)
    {
        get => lazyLoader.Load(this, ref docenten);
        set => docenten = value;
    }
}
```

- (1) Je injecteert een *ILazyLoader* en slaat die op in een private variabele.
- (2) Je maakt ook een tweede, parameterloze constructor die gebruikt wordt in de seeding in *EFOpleidingenContext.cs*.
- (3) Je vervangt de oude virtual property *Docenten*...
- (4) ...door een nieuwe. In de getter laad je de bijhorende private variabele *docenten* via de lazyloader.

We doen net hetzelfde in de class *Docent*.

- Wijzig de class *Docent* als volgt :

```
public partial class Docent
{
    private readonly ILazyLoader lazyLoader;
    public Docent() { }
    public Docent(ILazyLoader lazyLoader)
    {
        this.lazyLoader = lazyLoader;
    }
    ...
}
```

```
//public virtual Campus Campus { get; set; }
private Campus campus;
public Campus Campus
{
    get => lazyLoader.Load(this, ref campus);
    set => campus = value;
}
}
```

- In EFContext.cs kan je de methodoproep `.UseLazyLoadingProxies()` verwijderen.

```
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
{
    ...
    optionsBuilder
        .UseSqlServer(connectionString,
            options => options.MaxBatchSize(150))
        .UseLoggerFactory(GetLoggerFactory())
        .EnableSensitiveDataLogging(true)
        .UseLazyLoadingProxies();
    ...
}
```

- Je kan opnieuw uitproberen.

### 8.8.2 Eager Loading

Eager loading is net het tegengestelde van lazy loading : je leest in één keer meteen alle informatie in. Je stuurt slechts één request naar de database maar de resultset is wel veel groter.

Je past eager loading toe door in de LINQ-query niet enkel de Docent-entity's te lezen maar ook al de geassocieerde Campus-entity's. Je gebruik hiervoor in de query de `Include()`-method.

We proberen dit uit. We herstellen eerst de classes Docent en Campus naar hun oorspronkelijke versie.

- Verwijder in *Campus.cs* de lazyloader private variabele en de constructor die een `ILazyLoader` als parameter heeft. De constructor die Docenten initialiseert op een lege lijst laat je staan.
- Maak in dezelfde class de property Docenten terug virtual.

```
[Table("Campussen")]
public class Campus
{
    public Campus()
    {
        Docenten = new List<Docent>();
    }
    ...
    public virtual ICollection<Docent> Docenten { get; set; }
}
```

- Doe net hetzelfde in de class *Docent.cs*.

```
[Table("Docenten")]
public partial class Docent
{
    public Docent() { }
    ...
}
```

```

    [Required]
    public virtual Campus Campus { get; set; }
}

```

- Wijzig de code in het hoofdprogramma nu als volgt :

```

using (var context = new EFopleidingenContext())
{
    Console.WriteLine("Voornaam:");
    var voornaam = Console.ReadLine();
    var query = from docent in context.Docenten.Include("Campus")
                where docent.Voornaam == voornaam
                select docent;
    foreach (var docent in query)
        Console.WriteLine("{0} : {1}", docent.Naam, docent.Campus.Naam);
}

```

(1)

- (1) Je gebruikt de method Include op de property Docenten. Als parameter geef je de naam van de associatie mee.
- Probeer uit. In de logging-informatie zie je dat het SQL-statement deze keer een inner join bevat.

```

SELECT [d].[DocentId], [d].[CampusId], [d].[Familiennaam], [d].[Geboortedatum],
[d].[Geslacht], [d].[HeeftRijbewijs], [d].[InDienst], [d].[LandCode], [d].[Voornaam],
[d].[Maandwedde], [c].[CampusId], [c].[Gemeente], [c].[Huisnummer], [c].[CampusNaam]
, [c].[Postcode], [c].[Straat]
FROM [Docenten] AS [d]
INNER JOIN [Campussen] AS [c] ON [d].[CampusId] = [c].[CampusId]
WHERE ([d].[Voornaam] = @__voornaam_0) AND @__voornaam_0 IS NOT NULL
Roger Baens : Delos
Roger Baguet : Andros
Roger Blockx : Ikarria

```

Een tweede voorbeeld : deze keer lezen we de campus-objecten waarvan de naam een zoekwoord bevat. We lezen meteen ook de bijhorende docenten.

- Neem onderstaande code over in het hoofdprogramma :

```

using (var context = new EFopleidingenContext())
{
    Console.WriteLine("Geef (een deel van) de naam van de campus:");
    var deelNaam = Console.ReadLine();
    var query = from campus in context.Campussen.Include("Docenten")
                where campus.Naam.Contains(deelNaam)
                orderby campus.Naam
                select campus;

    foreach (var campus in query)
    {
        var campusNaam = campus.Naam;
        Console.WriteLine(campusNaam);
        Console.WriteLine(new string('-', campusNaam.Length));
        foreach (var docent in campus.Docenten)
            Console.WriteLine(docent.Naam);
        Console.WriteLine();
    }
}

```

Je kan deze query ook schrijven met query methods i.p.v. een LINQ query :

```

var query = context.Campussen.Include("Docenten")
    .Where(campus => campus.Naam.Contains(deelNaam))
    .OrderBy(campus => campus.Naam);

```

## 8.9 ToList

In de bovenstaande voorbeelden hebben we telkens een query gedefinieerd en er vervolgens doorheen gelust met een foreach.

EF Core stuurt een SQL-statement naar de database op het moment dat we de query gebruiken en niet bij de definitie. Dit heeft als gevolg dat als we de query twee maal na elkaar doorlopen, er twee maal (hetzelfde) SQL-statement naar de database wordt gestuurd.

Daarnaast is het zo dat de query moet uitgevoerd worden binnen het using-statement. Je kan de query dus niet definiëren, doorgeven aan een andere method en daar laten uitvoeren.

We lossen beide problemen op door de ToList()-method uit te voeren op de query.

- Neem onderstaand voorbeeld over in het hoofdprogramma :

```
List<Campus> campussen; (1)
```

```
using (var context = new EFopleidingenContext())
{
    var query = from campus in context.Campussen
                orderby campus.Naam
                select campus;
    campussen = query.ToList(); (2)
}
```

```
foreach (var campus in campussen) (3)
    Console.WriteLine(campus.Naam);
```

```
Console.WriteLine();
foreach (var campus in campussen) (4)
    Console.WriteLine(campus.Naam);
```

- (1) We definiëren een List van Campus-objecten.
  - (2) Na de definitie van de query voeren we deze uit door er de ToList()-method op los te laten.
  - (3) We lussen buiten het using-statement een eerste maal doorheen de List van Campus-objecten in het interne geheugen.
  - (4) En ook een tweede maal zonder dat er een extra SQL-statement wordt uitgevoerd.
- Probeer uit.

Je kan de query én de List opbouwen in één method en de List doorgeven aan een andere method die over de List iterateert :

```
static void Main(string[] args)
{
    foreach (var campus in FindAllCampussen())
        Console.WriteLine(campus.Naam);
}

static List<Campus> FindAllCampussen()
{
    using (var context = new EFopleidingenContext())
    {
        return (from campus in context.Campussen
                orderby campus.Naam
                select campus).ToList();
    }
}
```

Je kan de ToList()-method ook toepassen op query's die je definieert met query-methods i.p.v. LINQ.

## 8.10 Oefening

Toon in de solution *Taken* een alfabetische lijst van de klanten. Je toont per klant zijn naam, zijn rekeningen en het totale saldo van de rekeningen van de klant.

|                       |
|-----------------------|
| Bart                  |
| Totaal:0              |
| Homer                 |
| 345-6789012-12:500,00 |
| Totaal:500,00         |
| ...                   |





## 9 Entity's toevoegen

---

Behalve entiteiten uitlijsten kan je via EF Core ook entity's toevoegen. Je kan één of meerdere entity's toevoegen en ook bij het toevoegen meteen de associatie leggen met een gerelateerde entiteit.

### 9.1 Eén entiteit toevoegen

Om een entity toe te voegen in de database moet je volgende stappen zetten :

1. Je maakt een entity aan in het interne geheugen en je vult de property's van die entity in.
2. Je voegt deze entity toe aan de verzameling gelijkaardige entity's in de DbContext. Je doet dit met de method `Add()` van deze DbSet. In ons voorbeeld beschikken de property's `Docenten` en `Campussen` uit `EFOpleidingenContext` over een method `Add`.
3. Je roept op de DbContext de method `SaveChanges()` op. EF Core stuurt op dat moment een insert-statement naar de database om de entity als een record toe te voegen.

We proberen dit uit.

- Voeg onderstaande code toe in het hoofdprogramma :

```
var campus = new Campus
{
    Naam = "Campus01",
    Straat = "Straat01",
    Huisnummer = "1",
    Postcode = "1111",
    Gemeente = "Gemeente01"
};

using (var context = new EFOpleidingenContext())
{
    context.Campussen.Add(campus);           (1)
    context.SaveChanges();                   (2)
    Console.WriteLine(campus.CampusId);      (3)
}
```

- (1) Je voegt de entiteit toe aan de DbSet *Campussen* in de DbContext-class. De entiteit is nu nog niet opgeslagen in de database.
- (2) Je slaat de entiteit op door de method *SaveChanges()* toe te passen op de contextclass.
- (3) De campusid van de nieuwe campus wordt automatisch ingevuld en kan meteen opgevraagd worden.

De method `SaveChanges()` doet echter meer dan we vermoeden. Het controleert namelijk alle objecten van alle DbSets in de contextclass. Deze objecten hebben een bepaalde state. In bovenstaand voorbeeld heeft de nieuwe campus de state *toegevoegd*. Alle objecten met een state *toegevoegd* zullen via een insert-statement aan de database toegevoegd worden. Dit gebeurt in één transactie. Dit betekent dat als er één insert niet lukt er geen enkele insert doorgaat.

- Start het programma.

Er worden 2 SQL-statements uitgevoerd : een insert-statement dat het record toevoegt en een select-statement dat de nieuwe campusid ophaalt :

```
INSERT INTO [Campussen] ([Gemeente], [Huisnummer], [CampusNaam], [Postcode], [Straat])
VALUES (@p0, @p1, @p2, @p3, @p4);
SELECT [CampusId]
FROM [Campussen]
WHERE @@ROWCOUNT = 1 AND [CampusId] = scope_identity();
```

Je kan nu controleren in SSMS of in de Visual Studio Server Explorer of de campus goed is toegevoegd.

## 9.2 Meerdere entiteiten toevoegen

Je kan aan een DbSet meerdere entiteiten toevoegen door meerdere keren de Add()-method te gebruiken of je kan ze ook in één keer toevoegen met de AddRange()-method.

- Neem onderstaande code over in het hoofdprogramma :

```
var campus2 = new Campus
{
    Naam = "Campus02", Straat = "Straat02", Huisnummer = "2",
    Postcode = "2222", Gemeente = "Gemeente02"
};

var campus3 = new Campus
{
    Naam = "Campus03", Straat = "Straat03", Huisnummer = "3",
    Postcode = "3333", Gemeente = "Gemeente03"
};

var campus4 = new Campus
{
    Naam = "Campus04", Straat = "Straat04", Huisnummer = "4",
    Postcode = "4444", Gemeente = "Gemeente04"
};

var campus5 = new Campus
{
    Naam = "Campus05", Straat = "Straat05", Huisnummer = "5",
    Postcode = "5555", Gemeente = "Gemeente05"
};

using (var context = new EFopleidingenContext())
{
    context.Campussen.AddRange(campus2, campus3);           (1)
    context.Campussen.AddRange(new List<Campus> {campus4,campus5}); (2)
    context.SaveChanges();
}
```

- (1) Je voegt meerdere campussen toe door ze als parameters mee te geven aan de AddRange-method.
  - (2) Je voegt meerdere campussen toe door ze als een List van campussen mee te geven aan de AddRange-method.
- Probeer de code uit.

### 9.3 Meerdere entiteiten van een verschillend type toevoegen

Je kan ook meerdere entiteiten in één keer toevoegen, ook al zijn ze van een verschillend type. Je doet dit eveneens met de AddRange-method.

- Neem onderstaande code over in het hoofdprogramma :

```
var campus6 = new Campus
{
    Naam = "Campus06", Straat = "Straat06", Huisnummer = "6",
    Postcode = "6666", Gemeente = "Gemeente06"
};

var docent1 = new Docent
{
    Familienaam = "Docent01", Voornaam = "Voornaam01",
    Wedde = 1111, CampusId = 1
};

using (var context = new EFOPleidingenContext()) {
    context.AddRange(campus6, docent1);
    context.SaveChanges();
}
```

(1)

- (1) De campus en de docent worden samen toegevoegd.

- Probeer ook deze code uit.

### 9.4 Entiteiten én bijhorende nieuwe geassocieerde entiteiten toevoegen

Om een nieuwe entity met een nieuwe, bijhorende geassocieerde entity op te slaan volstaat het één van beide entity's toe te voegen aan de DbContext via de Add-method. Wanneer je daarna de SaveChanges-method uitvoert, voegt EF Core beide entiteiten (records) toe aan de database.

In een eerste voorbeeld maken we een nieuwe campus en een nieuwe docent. We associëren de docent met de campus vanuit het standpunt van de campus. Daarna bewaren we beide entiteiten in één beweging.

- Neem onderstaande code op in het hoofdprogramma :

```
var campus7 = new Campus {
    Naam = "Campus07", Straat = "Straat07", Huisnummer = "7",
    Postcode = "7777", Gemeente = "Gemeente07"
};

var docent2 = new Docent {
    Voornaam = "Voornaam02", Familienaam = "Docent02", Wedde = 2222
};

campus7.Docenten.Add(docent2);

using (var context = new EFOPleidingenContext()) {
    context.Campussen.Add(campus7);
    context.SaveChanges();
}
```

(1)

- (1) De docent wordt geassocieerd met de campus door deze toe te voegen aan de verzameling docenten van die campus.

- Probeer deze code uit.

In een tweede voorbeeld voegen we opnieuw een campus en een docent toe. We associëren de docent met de campus vanuit het standpunt van de docent.

- Neem onderstaande code op in het hoofdprogramma :

```
var campus8 = new Campus
{
    Naam = "Campus08", Straat = "Straat08", Huisnummer = "8",
    Postcode = "8888", Gemeente = "Gemeente08"
};

var docent3 = new Docent {
    Voornaam = "Voornaam03", Familienaam = "Docent03", Wedde = 3333
};

docent3.Campus = campus8;

using (var context = new EFopleidingenContext())
{
    context.Docenten.Add(docent3);
    context.SaveChanges();
}
```

(1)

- (1) We associëren de docent met de campus door de property Campus van de docent in te vullen.

- Probeer deze code uit.

## 9.5 Entiteiten én bijhorende bestaande geassocieerde entiteiten toevoegen

We kunnen ook nieuwe entiteiten toevoegen met een associatie naar een bestaande entiteit.

Als voorbeeld voegen we een nieuwe docent toe die tot een bestaande campus moet behoren. Dit kan op twee manieren :

1. Je leest de geassocieerde entity (de bestaande campus) en associeert die aan de nieuwe entity via de Campus-property.
2. Je associeert de bestaande campus aan de docent via de foreign-key property CampusId.

In onderstaande code proberen we beide uit.

- Neem onderstaande code op toe in het hoofdprogramma :

```
var docent4 = new Docent {
    Voornaam = "Voornaam04", Familienaam = "Docent04", Wedde = 4444
};

using (var context = new EFopleidingenContext())
{
    var campus1 = context.Campussen.Find(1);
    if (campus1 != null)
    {
        context.Docenten.Add(docent4);
        docent4.Campus = campus1;
        context.SaveChanges();
    }
    else
        Console.WriteLine("Campus 1 niet gevonden");
}

var docent5 = new Docent
```

(1)

(2)

(3)

(4)

```

{
    Voornaam = "Voornaam05", Familienaam = "Docent05",
    Wedde = 5555, CampusId = 1
};
using (var context = new EFOPleidingenContext())
{
    context.Docenten.Add(docent5);
    context.SaveChanges();
}

```

- (1) Je leest de bestaande campus die aan de nieuwe docent moet worden gekoppeld.
- (2) Als die campus bestaat voeg je de docent toe.
- (3) Je zet de Campus-property van de docent op de gevonden campus.
- (4) Je bewaart de wijzigingen.
- (5) Je zet de CampusId-property van de docent op de id van de gevonden campus.
- (6) Je voegt de docent toe.
- (7) Je bewaart de wijzigingen.

- Probeer uit.

In de bovenstaande voorbeelden hebben we de associatie telkens ingesteld vanuit de veel-kant van de relatie : de docenten. We kunnen de associatie ook instellen vanuit de één-kant van de relatie.

- Neem onderstaande code over in het hoofdprogramma :

```

var docent = new Docent {
    Voornaam = "Voornaam06", Familienaam = "Docent06", Wedde = 6666
};

using (var context = new EFOPleidingenContext())
{
    var campus = context.Campussen.Find(1);
    if (campus != null)
    {
        campus.Docenten.Add(docent);
        context.SaveChanges();
    }
    else
        Console.WriteLine("Campus 1 niet gevonden");
}

```

- (1) We zoeken de campus met campusid = 1
- (2) Als we die vinden dan voegen we die toe aan de docentenlijst van deze campus

## 9.6 Oefening

In de solution *Taken* laten we de gebruiker een zichtrekening (Soort = 'Z') toevoegen. Je toont eerst een alfabetische lijst van de klanten. De gebruiker kies daaruit één klant door het klantnummer in te tikken.

Tikt de gebruiker geen getal in dan toon je de foutmelding *Tik een getal*.

Tikt de gebruiker een onbestaand klantnummer dan toon je *Klant niet gevonden*.

Wordt er wél een geldig klantnummer ingegeven dan vraag je het rekeningnummer van de nieuwe rekening. Je moet op dit rekeningnummer geen controle uitvoeren.

Je maakt met deze gegevens een nieuwe zichtrekening voor de gekozen klant. Het saldo zet je op 0.



## 10 Entity's wijzigen

---

Je kan de gegevens, opgeslagen in één of meerdere entity's wijzigen. Ook de entity's die je indirect inleest omdat ze gerelateerd zijn aan een andere entity kan je aanpassen. In dit hoofdstuk bekijken we tenslotte ook hoe je een associatie verandert.

### 10.1 Eén entity wijzigen

Om een entity te wijzigen volstaat het een property van de ingelezen entity te veranderen en vervolgens de method `SaveChanges` op te roepen op de context. EF Core stuurt dan een update SQL-statement naar de database.

Een voorbeeld : we geven één docent opslag.

- Neem onderstaande code over in het hoofdprogramma :

```
Console.Write("DocentNr.:");
if (int.TryParse(Console.ReadLine(), out int docentNr))
{
    using (var context = new EFopleidingenContext())
    {
        var docent = context.Docenten.Find(docentNr);
        if (docent != null)
        {
            Console.WriteLine("Wedde:{0}", docent.Wedde);
            Console.Write("Bedrag:");
            if (decimal.TryParse(Console.ReadLine(), out decimal bedrag))
            {
                docent.Opslag(bedrag);
                context.SaveChanges();
            }
            else
                Console.WriteLine("Tik een getal");
        }
        else
            Console.WriteLine("Docent niet gevonden");
    }
}
else
    Console.WriteLine("Tik een getal");
```

- In de class *Docent.cs* voeg je een method *Opslag()* toe :

```
public partial class Docent
{
    ...
    public void Opslag(decimal bedrag)
    {
        Wedde += bedrag;
    }
}
```

- Je kan het programma uitproberen. Controleer daarna de waarde van de wedde van de betrokken docent.

## 10.2 Meerdere entity's lezen en slechts enkele daarvan wijzigen

Meestal zal je in jouw programma meerdere entiteiten inlezen en er slechts enkele daarvan wijzigen. Een oproep van de method `SaveChanges` op de context zorgt ervoor dat voor alle aangepaste entiteiten een update-statement naar de database wordt gestuurd.

Een voorbeeld : je geeft alle docenten met een wedde tot een bepaalde bovengrens een opslag van €100.

- Neem onderstaande code over in het hoofdprogramma :

```
Console.Write("Bovengrens : ");
if (int.TryParse(Console.ReadLine(), out int grens))
{
    using (var context = new EFopleidingenContext()) {
        foreach (var docent in context.Docenten)           (1)
            if (docent.Wedde <= grens) docent.Opslag(100m);   (2)
        context.SaveChanges();                               (3)
    }
}
else
    Console.WriteLine("Tik een getal");
```

- (1) We lezen alle docenten in.
  - (2) De docenten met een wedde tot aan de bovengrens krijgen een opslag.
  - (3) Een oproep van de `SaveChanges` method stuurt de nodige update-statements naar de database.
- Probeer uit en controleer de nieuwe weddes.

Opmerking : het bovenstaande programma kan uiteraard ook efficiënter geschreven worden door bij de selectie van de docenten reeds de voorwaarde op te nemen via een where-clausule. We deden dit niet zodat we over een set entiteiten zouden beschikken waarvan er slechts enkele gewijzigd waren.

## 10.3 Entity's wijzigen die indirect zijn ingelezen via associaties

Soms lees je een entity in en lees je via de navigation property één of een verzameling geassocieerde entity's in.

Voorbeeld : je leest een Campus-entity. Via de `Docenten`-property kunnen één of meerdere geassocieerde docenten-entity's worden gelezen. Wanneer we de gegevens van deze docenten wijzigen zullen er na de oproep van de `SaveChanges`-method ook voor deze entity's update-statements naar de database worden gestuurd.

- Neem onderstaande code over in het hoofdprogramma :

```
using (var context = new EFopleidingenContext()) {
    var campus1 = context.Campussen.Include("Docenten")
        .FirstOrDefault(c => c.CampusId == 1);
    if (campus1 != null) {
        foreach (var docent in campus1.Docenten)
            docent.Opslag(10M);
        context.SaveChanges();
    }
}
```

- Probeer uit en controleer de nieuwe weddes.



## 10.4 Een associatie van een entity wijzigen

Ook een associatie van een entiteit kan wijzigen. Een docent kan bijvoorbeeld veranderen van campus. We kunnen dit doen vanuit de veel-kant van de relatie of vanuit de één-kant.

### 10.4.1 De associatie wijzigen vanuit de veel-kant

Ja kan een associatie op twee manieren wijzigen vanuit de veel-kant :

1. De te associëren entity inlezen en associëren aan de te wijzigen entity. In het voorbeeld betekent dit dat de nieuwe campus wordt gelezen en dat de docent aan deze campus wordt geassocieerd.
2. De te associëren entity associëren via de foreign key property. In het voorbeeld betekent dit dat de campusid-property in de docent entity wordt gewijzigd.

We proberen beide methodes nu uit.

Als eerste voorbeeld verhuizen we docent 1 naar campus 6.

- Neem onderstaande code over in het hoofdprogramma :

```
using (var context = new EFopleidingenContext())
{
    var docent1 = context.Docenten.Find(1);           (1)
    if (docent1 != null)
    {
        var campus6 = context.Campussen.Find(6);      (2)
        if (campus6 != null)
        {
            docent1.Campus = campus6;                 (3)
            context.SaveChanges();
        }
        else
            Console.WriteLine("Campus 6 niet gevonden");
    }
    else
        Console.WriteLine("Docent 1 niet gevonden");
}
```

- (1) De te wijzigen docent wordt opgehaald.
- (2) Deze docent moet gekoppeld worden aan campus 6. We halen deze campus op.
- (3) We associëren de docent met de nieuwe campus via de property Campus.

- Probeer de code uit.

We kunnen de associatie ook via de foreign key wijzigen. Op deze manier moet de te associëren entity niet ingelezen worden wat performantiewinst betekent.

In onderstaand voorbeeld verhuizen we docent 1 naar campus 2.

- Neem onderstaande code over in het hoofdprogramma.

```
using (var context = new EFopleidingenContext())
{
    var docent1 = context.Docenten.Find(1);
    if (docent1 != null)
    {
```

```

        docent1.CampusId = 2;
        context.SaveChanges();
    }
    else
        Console.WriteLine("Docent 1 niet gevonden");
}

```

(1) We veranderen de associatie door de campusid een andere waarde te geven.

- Probeer de code uit.

#### 10.4.2 De associatie wijzigen vanuit de één-kant

Je kan een associatie ook wijzigen vanuit de één-kant van de relatie. In ons voorbeeld waarbij we een docent van campus laten verhuizen betekent dit dat we dit zullen doen vanuit de kant van de campus. We doen dit door de docent toe te voegen aan de docentenlijst van de nieuwe campus. Je hoeft de docent niet te verwijderen uit de docentenlijst van de oude campus.

We proberen dit uit met een voorbeeld : we verhuizen docent 1 naar campus 3.

- Neem onderstaande code over in het hoofdprogramma en probeer daarna uit.

```

using (var context = new EFopleidingenContext())
{
    var docent1 = context.Docenten.Find(1);
    if (docent1 != null)
    {
        var campus3 = context.Campussen.Find(3);
        if (campus3 != null)
        {
            campus3.Docenten.Add(docent1);
            context.SaveChanges();
        }
        else
            Console.WriteLine("Campus 3 niet gevonden");
    }
    else
        Console.WriteLine("Docent 1 niet gevonden");
}

```

(1) De docent wordt ingelezen.

(2) De campus wordt ingelezen.

(3) De docent wordt toegevoegd aan de docentenlijst van de campus.

### 10.5 Oefening

In de solution *Taken* laten we de gebruiker geld storten op een rekening.

Je vraagt eerst het nummer van de rekening waarop de gebruiker het geld wil storten. Als de gebruiker een onbestaand rekeningnummer ingeeft toon je een gepaste foutmelding.

Bij een geldig rekeningnummer vraag je het te storten bedrag. Wanneer de gebruiker geen getal ingeeft of een getal kleiner of gelijk aan 0 dan toon je opnieuw een gepaste foutmelding.

Bij een geldig getal voer je de storting door.

## 11 Entity's verwijderen

---

Entiteiten moeten uiteraard ook verwijderd kunnen worden. We doen dit door eerst de entiteit in te lezen. Vervolgens geven we deze entiteit mee als parameter aan de method `Remove` van de bijhorende dbset uit de contextclass. Vervolgens wordt op diezelfde contextclass de `SaveChanges`-method opgeroepen.

### 11.1 Voorbeeld

Een voorbeeld : we laten de gebruiker een docentid intikken en verwijderen vervolgens de bijhorende docent.

- Neem onderstaande code over in het hoofdprogramma en probeer daarna uit :

```
Console.WriteLine("Nummer docent:");
if (int.TryParse(Console.ReadLine(), out int docentNr))
{
    using (var context = new EFopleidingenContext())
    {
        var docent = context.Docenten.Find(docentNr);           (1)
        if (docent != null)
        {
            context.Docenten.Remove(docent);                   (2)
            context.SaveChanges();
        }
        else
            Console.WriteLine("Docent niet gevonden");
    }
}
else
    Console.WriteLine("Tik een getal");
```

(1) Je leest de docent.

(2) Je verwijdert de docent uit de dbset `Docenten` uit de contextclass.

### 11.2 Oefening

We laten de gebruiker een klant verwijderen op voorwaarde dat deze geen rekeningen heeft.

Je vraagt eerst het klantnummer van de te verwijderen klant. Geef eventueel een passende foutboodschap wanneer er geen getal of een onbestaand nummer wordt ingetikt. Ook wanneer de gebruiker nog rekeningen heeft toon je een foutmelding. Zoniet dan verwijder je de klant.



## 12 Unit testing

---

We hebben in onze solution ook een test-project opgenomen. In dit hoofdstuk voegen we enkele eenvoudige tests toe. Verderop in de cursus zullen we regelmatig testen toevoegen.

Om onze unit testen uit te voeren gaan we geen gebruik maken van een echte externe database maar van een in-memory test-database. Deze database is snel en kunnen we bovendien on the fly aanmaken.

We proberen dit uit met twee verschillende in-memory databases InMemory en SQLite:

a) **InMemory** : dit is een non-relational database

- ✓ Snel
- ✓ De volgorde wordt gerespecteerd
- ✗ Niet relationeel.
  - Je kan gegevens bewaren die niet voldoen aan de referentiële integriteit
  - Concurrency met **TimeStamp** werkt hier niet
- ✗ Ondersteunt geen SQL

b) **SQLite** : dit is een relational database

- ✓ Relationele database. FK constraint validatie.
- ✓ SQL
- ✓ Minimale OS ondersteuning
- ✓ Werkt zonder server-process
- ✓ Geen configuratie nodig
- ✓ Maakt gebruik van transacties
- ✓ Staat dicht bij een real-life database
- ✓ De testen zijn betrouwbaarder
- ✗ Trager dan In-memory database (tot 5 maal)
- ✗ De volgorde wordt niet gerespecteerd
- ✗ Is beperkter dan SQL Server

### 12.1 InMemory

We proberen eerst de InMemory database uit.

Wanneer we onze testen gaan uitvoeren met de InMemory database mogen we geen gebruik maken van de SQL Server database die nu gedefinieerd staat in de *OnConfiguring()* method van de DbContext class. We gebruiken wél de InMemory database gespecificeerd in het test-project. Om te vermijden dat bij het testen de database in de *OnConfiguring* method wordt geactiveerd, gaan we vooraf kijken of de database reeds geconfigureerd is. Als dat het geval is dan betekent dit dat de code geactiveerd is vanuit het testproject, zoniet dan configureren we de SQL Server.

- Wijzig in de class *EFOpleidingenContext.cs* de code in de method *OnConfiguring()* als volgt :

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        configuration = new ConfigurationBuilder()
        ...
        optionsBuilder.UseSqlServer(...)
            .UseLoggerFactory(GetLoggerFactory())
            .EnableSensitiveDataLogging(true);
    }
}
```

- (1) Enkel als er nog geen database is geconfigureerd configureren we de SQL Server.

We moeten ook de seeding overslaan. Daarvoor gebruiken we een boolean in de DbContext-class. We kijken opnieuw of de database reeds geconfigureerd is. In dat geval zitten we dus in test-modus en kan de boolean op true. We voeren enkel de seeding uit als de boolean op false staat.

- Voeg in *EFOpleidingenContext.cs* een boolean *testMode* toe.

```
public class EFOpleidingenContext : DbContext
{
    ...
    bool testMode = false;
    ...
}
```

- In de method *OnConfiguring()* voeg je aan het if-statement een else-block toe waarin je deze boolean op true zet.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        ...
    }
    else
    {
        testMode = true;
    }
}
```

- Je gebruikt deze boolean nu in *OnModelCreating()* om te bepalen of de seeding moet uitgevoerd worden of niet.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    if (!testMode)
    {
        modelBuilder.Entity<Campus>().HasData(
            new Campus { ...
        }
    }
}
```

Om de InMemory-database te configureren zullen we een constructor van de contextclass oproepen. Die voegen we nu toe.

- Voeg onderstaande constructor toe in *EFOpleidingenContext.cs* :

```
public EFOpleidingenContext(DbContextOptions<EFOpleidingenContext> options) :  
    base(options) { }
```

Aangezien er nu geen defaultconstructor meer wordt aangemaakt moeten we ook een parameterloze constructor toevoegen.

- Voeg onderstaande constructor toe in *EFOpleidingenContext.cs* :

```
public EFOpleidingenContext() { }
```

In de test-class zullen we methods testen uit een *DocentService*. We maken deze service aan in de folder *Services* in het project *Model*.

- Voeg onderstaande class *DocentService* toe in de folder *Services* :

```
public class DocentService  
{  
    private EFOpleidingenContext context;  
    public DocentService(EFOpleidingenContext context)  
    {  
        this.context = context;  
    }  
    public void ToevoegenDocent(Docent docent)  
    {  
        throw new NotImplementedException();  
    }  
    public IEnumerable<Docent> GetDocentenVoorCampus(int campus)  
    {  
        throw new NotImplementedException();  
    }  
    public Docent GetDocent(int id)  
    {  
        throw new NotImplementedException();  
    }  
}
```

We implementeren alvast één van deze methods.

- Vul de code voor de method *GetDocentenVoorCampus* als volgt aan :

```
public IEnumerable<Docent> GetDocentenVoorCampus(int campus)  
{  
    return context.Docenten.Where(x => x.CampusId == campus).ToList();  
}
```

We moeten enkel nog de package *Microsoft.EntityFrameworkCore.InMemory* installeren en dan is het project *Model* klaar.

- Installeer de NuGet package *Microsoft.EntityFrameworkCore.InMemory*, versie 3.1.5 voor het project *Model*.

We gaan verder met het testproject *Tests*. Aangezien we o.a. services gaan gebruiken uit het Model-project moeten we een reference leggen vanuit *Tests* naar *Model*.

- Klik met de rechtermuistoets op het project *Tests* en kies *Add – Project Reference...*
- Vink het project *Model* aan en klik op OK.

In het project *Tests* maken we nu een nieuwe testclass aan.

- Voeg in het project *Tests* een class toe met de naam *UnitTestsInMemory.cs*.

```
[TestClass]
public class UnitTestsInMemory
{
}
```

We voegen nu een testmethod toe. Daarin configureren we de inmemory-database en voegen we een aantal campussen en docenten toe. Tenslotte testen we een method uit de docentservice.

- Voeg in de class *UnitTestsInMemory.cs* onderstaande method toe :

```
[TestMethod]
public void GetDocentenVoorCampus_Docenten_AantalIsZesDocenten()
{
    // Arrange
    var options = new DbContextOptionsBuilder<EFOpleidingenContext>()
        .UseInMemoryDatabase("InMemoryDatabase")
        .Options;                                     (1)

    using (var context = new EFOpleidingenContext(options)) (2)
    {
        // Toevoegen campussen
        context.Campussen.Add(new Campus() {
            CampusId = 1, Naam = "Andros", Straat = "Somersstraat",
            Huisnummer = "22", Postcode = "2018", Gemeente = "Antwerpen"
        });   (3)
        context.Campussen.Add(new Campus() {
            CampusId = 2, Naam = "Delos", Straat = "Oude Vest",
            Huisnummer = "17", Postcode = "9200", Gemeente = "Dendermonde"
        });

        // Toevoegen docenten
        context.Docenten.Add(new Docent() {
            DocentId = 001, Voornaam = "Willy", Familienaam = "Abbeloos",
            Wedde = 1500m, HeeftRijbewijs = new Nullable<bool>(),
            InDienst = new DateTime(2019, 1, 1), CampusId = 1
        });
        context.Docenten.Add(new Docent() {
            DocentId = 002, Voornaam = "Joseph", Familienaam = "Abelshausen",
            Wedde = 1800m, HeeftRijbewijs = true,
            InDienst = new DateTime(2019, 1, 2), CampusId = 2
        });
        context.Docenten.Add(new Docent() {
            DocentId = 003, Voornaam = "Joseph", Familienaam = "Achten",
            Wedde = 1300m, HeeftRijbewijs = false,
            InDienst = new DateTime(2019, 1, 3), CampusId = 1
        });
        context.Docenten.Add(new Docent() {
            DocentId = 004, Voornaam = "François", Familienaam = "Adam",
            Wedde = 1700m, HeeftRijbewijs = new Nullable<bool>(),
            InDienst = new DateTime(2019, 1, 4), CampusId = 2
        });
    }
}
```



```

context.Docenten.Add(new Docent() {
    DocentId = 005, Voornaam = "Jan", Familienaam = "Adriaensens",
    Wedde = 2100m, HeeftRijbewijs = true,
    InDienst = new DateTime(2019, 1, 5), CampusId = 1
});

context.Docenten.Add(new Docent() {
    DocentId = 006, Voornaam = "René", Familienaam = "Adriaensens",
    Wedde = 1600m, HeeftRijbewijs = false,
    InDienst = new DateTime(2019, 1, 6), CampusId = 2
});

context.Docenten.Add(new Docent() {
    DocentId = 007, Voornaam = "Frans", Familienaam = "Aerenhouts",
    Wedde = 1300m, HeeftRijbewijs = new Nullable<bool>(),
    InDienst = new DateTime(2019, 1, 7), CampusId = 1
});

context.Docenten.Add(new Docent() {
    DocentId = 008, Voornaam = "Emile", Familienaam = "Aerts",
    Wedde = 1700m, HeeftRijbewijs = true,
    InDienst = new DateTime(2019, 1, 8), CampusId = 1
});

context.Docenten.Add(new Docent() {
    DocentId = 009, Voornaam = "Jean", Familienaam = "Aerts",
    Wedde = 1200m, HeeftRijbewijs = false,
    InDienst = new DateTime(2019, 1, 9), CampusId = 3
});

context.Docenten.Add(new Docent() {
    DocentId = 010, Voornaam = "Mario", Familienaam = "Aerts",
    Wedde = 1600m, HeeftRijbewijs = new Nullable<bool>(),
    InDienst = new DateTime(2019, 1, 10), CampusId = 1
});

context.SaveChanges();

var docentService = new DocentService(context);           (4)

// Act
var docenten = docentService.GetDocentenVoorCampus(1);    (5)

// Assert
Assert.AreEqual(6, docenten.Count());

    }
}

```

- (1) Je legt een verbinding met de inmemory-database.
- (2) Je voorziet de method van een dbcontext. Op die manier hebben we toegang tot de data in de database.
- (3) Je voegt een aantal campussen en docenten toe. Je bewaart deze met een SaveChanges.
- (4) Je voorziet de method van een docentservice.
- (5) Je roept de method GetDocentenVoorCampus voor de campus met id 1.
- (6) Het resultaat van deze oproep zou 6 moeten zijn.

Je voert de test nu uit.

- Open de Test Explorer en kies Run All Tests. De test slaagt. Ook al is er een docent gekoppeld aan een onbestaande campus (met id 3). De inmemory-database houdt dus geen rekening met referentiële integriteit.
- Wijzig in de assert de 6 in een 5. De test faalt. Wijzig terug naar 6.

We maken een tweede test. We testen of het ophalen van een docent met id gelijk aan 0 een `argumentexception` oplevert.

- Wijzig in `DocentService` de method `GetDocent()` als volgt :

```
public Docent GetDocent(int id)
{
    if (id == 0)
    {
        throw new ArgumentException(nameof(id));
    }
    return context.Docenten.FirstOrDefault(x => x.DocentId == id);
}
```

(1)

- (1) Wanneer er naar een docent wordt gevraagd met id gelijk aan 0 dan throwen we een `argumentexception`.

We maken nu een nieuwe testmethod waarin we de gegevens opvragen van een docent met id gelijk aan 0.

- Voeg in de testclass onderstaande method toe :

```
[TestMethod, ExpectedException(typeof(ArgumentException))]
public void GetDocent_Docent0_ThrowArgumentException()
{
    // Arrange
    var options = new DbContextOptionsBuilder<EF0pleidingenContext>()
        .UseInMemoryDatabase("InMemoryDatabase")
        .Options;

    using (var context = new EF0pleidingenContext(options))
    {
        var docentService = new DocentService(context);

        // Act
        var docent = docentService.GetDocent(0);

        // Assert
    }
}
```

- Voer test uit. De test slaagt.

Voor een derde test maken we in de `docentservice` een method waarin we een nieuwe docent toevoegen. Als deze docent geen landcode heeft dan krijgt deze de landcode BE. We testen deze method.

- Voeg in de class `DocentService` aan de method `ToevoegenDocent()` onderstaande code toe :

```
public void ToevoegenDocent(Docent docent)
{
    if (docent.LandCode == string.Empty || docent.LandCode == null)
        docent.LandCode = "BE";

    context.Docenten.Add(docent);
}
```

We schrijven een test waarin we een land toevoegen en ook een docent zonder land. Na toevoeging van de docent zou deze als landcode 'BE' moeten hebben.

- Voeg onderstaande testmethod toe in de testclass :

```
[TestMethod]
public void ToevoegenDocent_DocentZonderLand_DocentHeeftLandBE()
{
    // Arrange
    var options = new DbContextOptionsBuilder<EF0pleidingenContext>()
        .UseInMemoryDatabase("InMemoryDatabase")
        .Options;

    using (var context = new EF0pleidingenContext(options))
    {
        context.Landen.Add(new Land() {
            LandCode = "BE", Naam = "België"
        });
        context.SaveChanges();

        var docentService = new DocentService(context);

        var docent = new Docent() {
            DocentId = 20, Voornaam = "Fanny", Familienaam = "Kiekeboe",
            Wedde = 10100, InDienst = new DateTime(2019, 1, 1), CampusId = 1
        };

        // Act
        docentService.ToevoegenDocent(docent);
        context.SaveChanges();

        // Assert
        var docent1 = docentService.GetDocent(20);
        Assert.AreEqual("BE", docent1.LandCode);
    }
}
```

- (1) Je voegt een land toe, België.
  - (2) Je maakt een nieuwe docent-object, zonder landcode.
  - (3) Je voegt de docent toe via de servicemethod.
  - (4) Je vraagt de toegevoegde docent op.
  - (5) Het land van de docent zou België moeten zijn.
- Voer alle testen uit. Alle testen slagen.

Merk op dat alle testen beginnen met dezelfde code, de configuratie van de inmemory-database. Je kan deze code beter verplaatsen naar een initialisatiegedeelte.

- Wijzig de code in de testclass UnitTestsInMemory als volgt :

```
[TestClass]
public class UnitTestsInMemory
{
    DbContextOptions<EF0pleidingenContext> options;

    [TestInitialize]
    public void Initializer() {
        options = new DbContextOptionsBuilder<EF0pleidingenContext>()
            .UseInMemoryDatabase("InMemoryDatabase")
            .Options;
    }
    ...
}
```

- In de testmethods schrap je nu de declaratie van de options-variabele.

Misschien heb je het al gemerkt, de verschillende tests hebben invloed op elkaar. Wanneer je in de ene test een docent toevoegt dan heeft dit gevolgen voor een andere test. Dit is niet goed.

We lossen dit op door in elke test gebruik te maken van een verschillende inmemory-database.

Praktisch gezien gebruiken we in elke test een database waarvan de naam begint met "InMemoryDatabase" en eindigt met een unieke id.

- Wijzig de code in *Initializer()* als volgt :

```
[TestInitialize]
public void Initializer() {
    options = new DbContextOptionsBuilder<EFopleidingenContext>()
        .UseInMemoryDatabase($"InMemoryDatabase{Guid.NewGuid()}")
        .Options;
}
```

De unittesten zijn nu onafhankelijk van elkaar.

## 12.2 SQLite

SQLite is een database systeem dat een bestand op de harde schijf gebruikt om de gegevens in op te slaan. Daarnaast kan je er ook voor kiezen om de gegevens op te slaan in het interne geheugen. De naam van de datasource is dan ":memory:".

We installeren eerst de package Microsoft.EntityFrameworkCore.Sqlite.

- Start de Nuget Package Manager en zoek naar *Microsoft.EntityFrameworkCore.Sqlite*.
- Installeer versie 3.1.5 van deze package voor het project *Model*.

In het testproject *Tests* maken we een nieuwe testclass aan.

- Voeg aan het project *Tests* een nieuwe class *UnitTestsSqlite.cs* toe.
- Kopieer de variabele *options* en alle tests (inclusief de initializer) uit de class *UnitTestsInMemory.cs* en plak ze in *UnitTestsSqlite.cs*.
- Wijzig de code in *Initializer()* als volgt :

```
[TestInitialize]
public void Initializer() {
    var connectionStringBuilder = new SqlConnectionStringBuilder
    { DataSource = ":memory:" };
    var connection = new SqlConnection(connectionStringBuilder.ToString());
    options = new DbContextOptionsBuilder<EFopleidingenContext>()
        .UseSqlite(connection)
        .Options;
}
```

Wanneer je de tests nu zou uitproberen dan zullen er zeker nogal wat fouten optreden. We moeten een aantal wijzigingen aanbrengen omdat we deze keer met SQLite werken en niet met InMemory.

Vooreerst is er de referentiële integriteit. In de test *GetDocentenVoorCampus\_Docenten\_AantalIsZesDocenten()* krijgt een docent de onbestaande campusid 3. Dit zorgt sowieso voor een fout.

- Wijzig in *GetDocentenVoorCampus\_Docenten\_AantalIsZesDocenten()* de *CampusId* 3 naar 2.

Ook in de test `ToevoegenDocent_DocentZonderLand_DocentHeeftLandBE` ontbreekt een campus. Bij de creatie van de docent geven we de docent een campusid 1. Deze campus wordt in deze test niet aangemaakt.

- Voeg in de test `ToevoegenDocent_DocentZonderLand_DocentHeeftLandBE` na het toevoegen van het land de nodige code toe die een campus toevoegt met id = 1 :

```
[TestMethod]
public void ToevoegenDocent_DocentZonderLand_DocentHeeftLandBE()
{
    ...
    using (var context = new EFopleidingenContext(options)) {
        context.Landen.Add(new Land() { LandCode = "BE", Naam = "België" });
        context.Campussen.Add(new Campus() {
            CampusId = 1, Naam = "Andros", Straat = "Somersstraat",
            Huisnummer = "22", Postcode = "2018", Gemeente = "Antwerpen"
        });
        context.SaveChanges();
    }
    ...
}
```

We kunnen er best voor zorgen dat de verschillende tests, net als bij de `InMemory` testclass, onafhankelijk van elkaar kunnen werken. Daarvoor moeten we twee dingen doen : de connection naar de database expliciet openen en de database terug initialiseren op een lege database.

We openen de connection telkens aan het begin van de test. Aangezien de connection in `Initializer()` wordt geïnitieerd moeten we de connection class scope geven.

Daarnaast verwijderen én creëren we de database meteen na het aanmaken van de context.

- Wijzig de code in `UnitTestsSqlite.cs` als volgt :

```
public class UnitTestsSqlite
{
    DbContextOptions<EFopleidingenContext> options;
    SqlConnection connection;
    [TestInitialize]
    public void Initializer()
    {
        ...
        var connection = new SqlConnection(connectionStringBuilder.ToString()); (1)
        ...
    }
    [TestMethod]
    public void GetDocentenVoorCampus_Docenten_AantalIsZesDocenten()
    {
        connection.Open();
        using (var context = new EFopleidingenContext(options))
        {
            context.Database.EnsureDeleted();
            context.Database.EnsureCreated();
            ...
        }
    }
}
```

```
[TestMethod, ExpectedException(typeof(ArgumentException))]  
public void GetDocent_Docent0_ThrowArgumentException()  
{  
    connection.Open(); (2)  
    using (var context = new EF0pleidingenContext(options))  
    {  
        context.Database.EnsureDeleted(); (3)  
        context.Database.EnsureCreated();  
        ...  
    }  
}  
[TestMethod]  
public void ToevoegenDocent_DocentZonderLand_DocentHeeftLandBE()  
{  
    connection.Open(); (2)  
    using (var context = new EF0pleidingenContext(options))  
    {  
        context.Database.EnsureDeleted(); (3)  
        context.Database.EnsureCreated();  
        ...  
    }  
}
```

- (1) We verplaatsen de declaratie van de variabele connection.
- (2) Aan het begin van de test openen we expliciet de connection naar de Sqlite database.
- (3) Aan de start van het using-block verwijderen we de database en maken we die terug aan.

De tests zouden nu moeten lukken.

## 13 Transacties

---

Bij een database transactie worden meerdere SQL-statements als één geheel aanzien. Het is de verantwoordelijkheid van de database ervoor te zorgen dat...

- ...ofwel de volledige transactie lukt, wat wil zeggen dat alle SQL-statements binnen de transactie uitgevoerd zijn. Dit noemt men een **commit** van de transactie.
- ...ofwel de volledige transactie mislukt. Bijvoorbeeld bij een fout in de database, een fout in jouw applicatie, stroomuitval,... Dit wil zeggen dat alle SQL-statements binnen de transactie ongedaan gemaakt worden. Dit noemt men een **rollback** van de transactie.

Een voorbeeld van een transactie is het overschrijven van geld van een spaarrekening naar een zichtrekening bij dezelfde bank. Hiervoor zijn twee update-statements nodig:

1. Een statement dat het te transfereren geld aftrekt van het saldo van de spaarrekening
2. Een statement dat het te transfereren geld bijtelt bij het saldo van de zichtrekening

### 13.1 Kenmerken van transacties

Transacties hebben 4 kenmerken ook wel gekend als de ACID kenmerken :

#### Atomicity

De SQL-statements die tot de transactie behoren vormen één geheel. Een database voert een transactie helemaal of niet uit. Als er halverwege de transactie een fout gebeurt, dan brengt de database de bijgewerkte records terug in hun oorspronkelijke toestand.

#### Concistency

De transactie breekt geen databaseregels. Als een kolom bijvoorbeeld geen duplicaten mag bevatten, dan zal de database de transactie afbreken (rollback) op het moment dat je toch probeert het duplicaat toe te voegen.

#### Isolation

Gedurende een transactie zijn de bewerkingen van de transactie niet zichtbaar voor andere lopende transacties. Om dit te bereiken vergrendelt de database de bijgewerkte records tot het einde van de transactie.

#### Durability

Een voltooide transactie is definitief vastgelegd in de database, zelfs al valt de computer uit juist na het voltooiën van de transactie.

### 13.2 Isolation level

Het isolation level van een transactie definieert hoe de transactie beïnvloed wordt door handelingen van andere gelijktijdige transacties. Als meerdere transacties op eenzelfde moment in uitvoering zijn, kunnen volgende problemen optreden:

- Dirty read

Dit gebeurt als een transactie data leest die een andere transactie geschreven heeft, maar nog niet gecommit heeft. Als die andere transactie een rollback doet, is de data gelezen door de eerste transactie verkeerd.

- Nonrepeatable read

Dit gebeurt als een transactie meerdere keren dezelfde data leest en per leesopdracht deze data wijzigt. De oorzaak zijn andere transacties die tussen de leesoperaties van de eerste transactie dezelfde data wijzigen. De eerste transactie krijgt geen stabiel beeld van de gelezen data.

- Phantom read

Dit gebeurt als een transactie meerdere keren dezelfde data leest en per leesoperatie meer records leest. De oorzaak zijn andere transacties die records toevoegen tussen de leesoperaties van de eerste transactie. De eerste transactie krijgt geen stabiel beeld van de gelezen data.

Je verhindert één of meerdere van deze problemen door het isolation level van de transactie in te stellen. Er zijn 4 isolation levels : read uncommitted, read committed, repeatable read en serializable.

Deze isolation levels hebben volgende kenmerken :

| ↓ Isolation level ↓ | Dirty read kan optreden | Nonrepeatable read kan optreden | Phantom read kan optreden |
|---------------------|-------------------------|---------------------------------|---------------------------|
| Read uncommitted    | Ja                      | Ja                              | Ja                        |
| Read committed      | Nee                     | Ja                              | Ja                        |
| Repeatable read     | Nee                     | Nee                             | Ja                        |
| Serializable        | Nee                     | Nee                             | Nee                       |

Het lijkt aanlokkelijk altijd het isolation level Serializable te gebruiken, want deze keuze lost alle problemen op. Het is echter zo dat Serializable het traagste isolation level is. De isolation levels van snel naar traag:



Read uncommitted → Read committed → Repeatable read → Serializable



Je moet dus per programma-onderdeel analyseren welke problemen (dirty read, ...) de goede werking van dat programma-onderdeel benadelen. Je kiest daarna een isolation level dat deze problemen oplost. Het isolation level read uncommitted wordt zelden gebruikt, omdat het geen enkel probleem oplost.

### 13.3 De method SaveChanges

Je leerde al de method SaveChanges() van de DbContext kennen.

We herhalen nog eens wat deze method doet:

- Het stuurt voor iedere entity die je aan de DbContext hebt toegevoegd een insert-SQL-statement naar de database.



- Het stuurt voor iedere entity die je gelezen én gewijzigd hebt een update-SQL-statement naar de database.
- Het stuurt voor iedere entity die je verwijderd hebt ten opzichte van de object context een delete-SQL-statement naar de database.

De method `SaveChanges()` verzamelt al deze bewerkingen zelf in één transactie. Jij hoeft dus in veel gevallen geen transactiebeheer te doen.

De method `SaveChanges()` gebruikt read committed als transaction isolation level.

### 13.4 Eigen transactiebeheer met `TransactionScope`

In sommige gevallen kan je de ingebouwde transacties van de method `SaveChanges()` niet gebruiken:

- Je wil een ander isolation level dan read committed gebruiken.
- Je wil een distributed transaction doen. Bij een dergelijke transaction bevinden de records die tot de transaction behoren zich niet in één maar in meerdere databases.

Je kan je eigen transactiebeheer doen met de class **`TransactionScope`**.

Je maakt een `TransactionScope`-object binnen een using-structuur. Alle databasebewerkingen die je binnen deze using-structuur uitvoert, behoren automatisch tot één en dezelfde transactie.

```
using (var transactionScope = new TransactionScope())
{
    ...
}
```

Wanneer alle bewerkingen goed aflopen voer je op het `transactionScope` object de method `Complete` uit. Bij het uitvoeren van deze method gebeurt een commit van alle bewerkingen van alle database-connecties die je uitgevoerd hebt binnen de using-structuur.

Als je de using-structuur verlaat zonder de method `Complete` uit te voeren (bijvoorbeeld omdat er een exception optreedt), gebeurt er automatisch een rollback van alle bewerkingen van alle databaseconnecties die je opende binnen de using-structuur.

De method `SaveChanges` detecteert automatisch een lopende transactie die je met `TransactionScope` gestart hebt. De method `SaveChanges` start dan geen eigen transactie, maar doet zijn bewerkingen binnen jouw transactie.

Je beëindigt de transactie met een oproep van de `TransactionScope`-method `Complete()`.

Je kan `transactionScopes` nesten. Je kan dus binnen een transaction een soort sub-transaction opstarten. Dit ziet er in code als volgt uit :

```
using (var transactionScope = new TransactionScope())
{
    using (var transactionScope2 = new TransactionScope())
    {
        ...
    }
}
```

Hoe de binnenste transaction zich gedraagt ten opzichte van de buitenste transaction kan je instellen door de `TransactionScope` via een constructor een `scopeoption` mee te geven. Deze `scopeoption` is een enum die volgende waarden kan hebben :

- `TransactionScopeOption.Required`

Het `TransactionScope` object start een transactie als het object niet in een ander `TransactionScope` object genest is. Als er wel genest is, gebruikt het object de transactie die al door het omringende `TransactionScope`-object gestart werd.

- `TransactionScopeOption.RequiresNew`

Het `TransactionScope` object start een nieuwe transactie, zelfs als het object genest is in een ander `TransactionScope`-object.

- `TransactionScopeOption.Suppress`

Alle databasebewerkingen binnen dit `TransactionScope`-object behoren niet tot een transactie, zelfs als het `TransactionScope`-object genest is in een ander `TransactionScope`-object.

Ook het `isolationlevel` kan je aangeven via de `TransactionScope` constructor. Je geeft daartoe een parameter mee van het type `TransactionOption`. Dit object heeft een property `IsolationLevel`.

## 13.5 Voorbeeld

We proberen transactions uit in een voorbeeld. In een tabel `CursusVoorraden` worden de voorraden van cursussen in magazijnen bijgehouden.

De tabel heeft volgende structuur :

| DESKTOP-BQCVL2L\....CursusVoorraden |             |           |                          |
|-------------------------------------|-------------|-----------|--------------------------|
|                                     | Column Name | Data Type | Allow Nulls              |
| 🔑                                   | MagazijnNr  | int       | <input type="checkbox"/> |
| 🔑                                   | CursusNr    | int       | <input type="checkbox"/> |
|                                     | AantalStuks | int       | <input type="checkbox"/> |
|                                     | RekNr       | int       | <input type="checkbox"/> |

In een magazijn kan er een bepaald aantal stuks van een cursus aanwezig zijn. De kolom `RekNr` geeft aan in welk rek deze cursussen liggen.

We zullen cursussen van het éne magazijn transfereren naar een ander magazijn.

We maken deze tabel nu aan.

- Voeg in het project *Model* in de map *Entities* een class *Voorraad* toe :

```
[Table("CursusVoorraden")]
public class Voorraad
{
    public int MagazijnNr { get; set; }
    public int CursusNr { get; set; }
    [Required]
    public int AantalStuks { get; set; }
    [Required]
    public int RekNr { get; set; }
}
```

De tabel Voorraad heeft een samengestelde sleutel (MagazijnNr+CursusNr). We stellen die in via de method `OnModelCreating()`. In dezelfde method voegen we ook via seeding wat data toe. Bovenaan de class `EFOpleidingenContext` voeg je tenslotte een extra DbSet toe voor de tabel CursusVorraden.

- Voeg in `EFOpleidingenContext.cs` bovenaan een extra DbSet toe :

```
public class EFOpleidingenContext : DbContext
{
    public DbSet<Campus> Campussen { get; set; }
    public DbSet<Docent> Docenten { get; set; }
    public DbSet<Land> Landen { get; set; }
    public DbSet<Voorraad> Voorraden { get; set; }
    ...
}
```

- In de method `OnModelCreating()` definieer je de samengestelde sleutel en voeg je seeding data toe :

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Voorraad>().HasKey(table => new {
        table.MagazijnNr,
        table.CursusNr
    });
    if (!testMode)
    {
        modelBuilder.Entity<Voorraad>().HasData
        (
            new Voorraad { MagazijnNr = 1, CursusNr = 10,
                           AantalStuks = 100, RekNr = 3 },
            new Voorraad { MagazijnNr = 2, CursusNr = 10,
                           AantalStuks = 1000, RekNr = 17 },
            new Voorraad { MagazijnNr = 1, CursusNr = 20,
                           AantalStuks = 200, RekNr = 12 },
            new Voorraad { MagazijnNr = 2, CursusNr = 20,
                           AantalStuks = 2000, RekNr = 23 },
            new Voorraad { MagazijnNr = 1, CursusNr = 30,
                           AantalStuks = 300, RekNr = 4 },
            new Voorraad { MagazijnNr = 2, CursusNr = 30,
                           AantalStuks = 3000, RekNr = 9 }
        );
    }
    ...
}
```

We maken nu een nieuwe migration aan.

- Start de package manager console en ga met de instructie `cd model` naar de map `Model`.
- Tik het commando `dotnet ef migrations add metvoorraden` om de migration aan te maken.
- Je voert de migration door via het commando `dotnet ef database update`

We proberen nu op twee verschillende manieren een voorraadtransfer te doen : één keer met het ingebouwde transactionbeheer via `SaveChanges()` en één keer met eigen transactionbeheer.

### 13.5.1 Ingebouwd transactiebeheer

We voegen eerst in `Program.cs` een method `VoorraadTransfer()` toe die een voorraad cursussen transfereert van het éne magazijn naar het andere.

- Voeg onderstaande method toe in `Program.cs` :

```

static void VoorraadTransfer(int cursusNr, int vanMagazijnNr,
                             int naarMagazijnNr, int aantalStuks)
{
    using (var context = new EFopleidingenContext())
    {
        var vanVoorraad = context.Voorraden.Find(vanMagazijnNr, cursusNr); (1)
        if (vanVoorraad != null) (2)
        {
            if (vanVoorraad.AantalStuks >= aantalStuks) (3)
            {
                vanVoorraad.AantalStuks -= aantalStuks; (4)
                var naarVoorraad =
                    context.Voorraden.Find(naarMagazijnNr, cursusNr); (5)
                if (naarVoorraad != null) (6)
                {
                    naarVoorraad.AantalStuks += aantalStuks; (7)
                }
                else (8)
                {
                    naarVoorraad = new Voorraad
                    {
                        CursusNr = cursusNr,
                        MagazijnNr = naarMagazijnNr,
                        AantalStuks = aantalStuks
                    };
                    context.Voorraden.Add(naarVoorraad); (9)
                }
                context.SaveChanges(); (10)
            }
            else
            {
                Console.WriteLine("Te weinig voorraad voor transfer");
            }
        }
        else
        {
            Console.WriteLine("Artikel niet gevonden in magazijn {0}",
                              vanMagazijnNr);
        }
    }
}

```

- (1) We zoeken het voorraad-record op voor een bepaalde cursus in het magazijn van waaruit de transfer vertrekt.
- (2) Als we een dergelijk record vinden...
- (3) ...en de aanwezige voorraad is minstens zo groot als het aantal te transfereren cursussen...
- (4) ...dan verminderen we de voorraad in het vertrekmagazijn.
- (5) We zoeken het voorraad-record op voor een bepaalde cursus in het magazijn waar de transfer naartoe moet.
- (6) Als we dat record vinden...
- (7) ...dan passen we de voorraad aan.
- (8) Vinden we geen dergelijke record dan zijn er nog geen dergelijke cursussen in het bestemmingsmagazijn en moeten we een nieuw voorraadrecord aanmaken...
- (9) ...en toevoegen aan de database.
- (10) We eindigen de transactie met een SaveChanges-oproep.

We gebruiken deze method nu in het hoofdprogramma.

- Voeg onderstaande code toe in het hoofdprogramma :

```

static void Main(string[] args)
{
    try
    {
        Console.Write("Cursusnr.");
        var cursusNr = int.Parse(Console.ReadLine());

        Console.Write("Van magazijn nr.");
        var vanMagazijnNr = int.Parse(Console.ReadLine());

        Console.Write("Naar magazijn nr.");
        var naarMagazijnNr = int.Parse(Console.ReadLine());

        Console.Write("Aantal stuks.");
        var aantalStuks = int.Parse(Console.ReadLine());

        VoorraadTransfer(cursusNr, vanMagazijnNr,
                        naarMagazijnNr, aantalStuks);
    }
    catch (FormatException)
    {
        Console.WriteLine("Tik een getal");
    }
}

```

- Controleer de voorraad van een cursus en probeer daarna een transfer uit, bijvoorbeeld 5 exemplaren van cursus 10 van magazijn 1 naar magazijn 2.

### 13.5.2 Eigen transactiebeheer

Via de logging-informatie kon je in het bovenstaande voorbeeld zien dat EF Core in totaal 4 SQL-statements uitvoerde. Het programma doet de voorraadaanpassing correct op voorwaarde dat geen andere gebruikers tegelijk aanpassingen doen. Gedurende jouw transactie worden er immers geen records gelockt.

In een tweede versie van dit programma beheer je de transactie zelf zodat deze problemen zich niet meer kunnen voordoen. Je plaatst het isolation level van de transactie op repeatable read. Dan lockt de database de records die je leest tot het einde van de transactie. Zo kunnen andere gebruikers tussendoor de zelfde records niet wijzigen.

- Pas de method VoorraadTransfer() als volgt aan :

```

static void VoorraadTransfer(int cursusNr, int vanMagazijnNr,
                           int naarMagazijnNr, int aantalStuks)
{
    var transactionOptions = new TransactionOptions           (1)
    {
        IsolationLevel = IsolationLevel.RepeatableRead
    };
    using (var transactionScope = new TransactionScope(
        TransactionScopeOption.Required, transactionOptions)) (2)
    {
        using (var context = new EF0pleidingenContext())
        {
            ...
            context.SaveChanges();
            transactionScope.Complete();                       (3)
            ...
        }
    }
}

```

- (1) We definiëren een TransactionOptions-object waarvan we het isolationlevel op Repeatable-Read zetten.
  - (2) We starten een transactionscope met de gedefinieerde options.
  - (3) Aan het einde van de transaction roepen we de method *Complete()* op. Zonder deze oproep zou de transaction gerollbackt worden.
- Probeer uit.

### 13.6 Oefening

Je laat de gebruiker geld overschrijven van de ene rekening naar een andere.

Je vraagt eerst het rekeningnummer waarvan het geld wordt overgeschreven, daarna het rekeningnummer naar waar het geld wordt overgeschreven en tenslotte het over te schrijven bedrag.

Toon gepaste foutmeldingen als de gebruiker een onbestaand van-rekeningnummer of naar-rekeningnummer intikt, geen getal intikt voor het bedrag of een getal intikt dat kleiner is of gelijk aan nul. Als het saldo van de van-rekening kleiner is dan het over te schrijven bedrag, toon je de foutmelding Saldo ontoereikend.

## 14 Optimistic record locking

---

In het voorbije hoofdstuk heb je gezien dat transactions een oplossing kunnen zijn voor allerlei problemen die zich kunnen voordoen wanneer meerdere gebruikers dezelfde records lezen en aanpassen.

Transactions zorgen echter voor een zekere overhead en vertragen de uitvoering van het programma omdat telkens records gelockt moeten worden. In dit hoofdstuk stellen we met optimistic record locking een alternatief voor.

Bij optimistic record locking wordt het gelezen record niet echt gelockt. Bij het aanpassen van het record controleert EF Core of de waarden in het record nog dezelfde zijn als op het moment dat het record gelezen werd. Als dit niet het geval is (omdat ondertussen een andere applicatie dit record wijzigde), werpt EF Core een exception.

De manier waarop je optimistic record locking activeert verschilt een beetje, naargelang de table al of niet een timestamp-kolom heeft. Een timestamp-kolom is een kolom in de database waarin de database zelf bij iedere recordwijziging een andere waarde invult. In dit hoofdstuk proberen we optimistic record locking uit mét en zonder timestamp-veld.

### 14.1 Probleemstelling

In onderstaand voorbeeld vullen we de cursusvoorraad aan. Tussen het inlezen van de huidige voorraad en het wegschrijven van de nieuwe voorraad kan een andere gebruiker de voorraad ook veranderen. Dit leidt tot een probleem.

We proberen dit uit.

- Voeg onderstaande method toe in Program.cs :

```
static void VoorraadBijvulling(int cursusNr, int magazijnNr, int aantalStuks)
{
    using (var context = new EFopleidingenContext())
    {
        var voorraad = context.Voorraden.Find(magazijnNr, cursusNr);      (1)
        if (voorraad != null)
        {
            voorraad.AantalStuks += aantalStuks;                          (2)
            Console.WriteLine("Pas nu de voorraad aan met de Server Explorer,"
                              + " druk daarna op Enter");
            Console.ReadLine();   (3)
            context.SaveChanges();   (4)
        }
        else
            Console.WriteLine("Voorraad niet gevonden");
    }
}
```

- (1) Je zoekt de voorraad van een cursus in een magazijn.
- (2) Je verhoogt de voorraad met een aantal stuks.
- (3) We laten hier even een pauze zodat de voorraad elders kan gewijzigd worden.
- (4) We slaan alle wijzigingen op.

- Neem nu onderstaande code over in het hoofdprogramma :

```
static void Main(string[] args)
{
    try
    {
        Console.WriteLine("Cursusnr.");
        var cursusNr = int.Parse(Console.ReadLine());

        Console.WriteLine("Magazijn nr.");
        var magazijnNr = int.Parse(Console.ReadLine());

        Console.WriteLine("Aantal stuks toevoegen.");
        var aantalStuks = int.Parse(Console.ReadLine());

        VoorraadBijvulling(cursusNr, magazijnNr, aantalStuks);
    }
    catch (FormatException)
    {
        Console.WriteLine("Tik een getal");
    }
}
```

- Bekijk de voorraad van cursus 10 in magazijn 1.
- Start het programma.
- Verhoog de voorraad van cursus 10 in magazijn 1 met 100.
- Op het moment dat je de boodschap 'Pas nu de voorraad...' krijgt schakel je over naar SSMS.
- Wijzig de voorraad van cursus 10 in magazijn 1 naar 666.
- Schakel terug over naar het programma en druk op Enter.
- Controleer de voorraad in SSMS. De wijziging via SSMS is terug ongedaan gemaakt.

## 14.2 Oplossing zonder timestampveld

Op dit moment heeft de table CursusVoorraden geen timestamp-kolom. In dit geval activeren we optimistic record locking op de volgende manier. Je wijzigt de class Voorraad door vóór alle properties die meegenomen moeten worden voor de locking de annotation [ConcurrencyCheck] te plaatsen.

- Wijzig de class Voorraad als volgt :

```
[Table("CursusVoorraden")]
public class Voorraad
{
    [ConcurrencyCheck]
    public int MagazijnNr { get; set; }
    [ConcurrencyCheck]
    public int CursusNr { get; set; }
    [Required]
    [ConcurrencyCheck]
    public int AantalStuks { get; set; }
    [Required]
    [ConcurrencyCheck]
    public int RekNr { get; set; }
}
```

Opmerking : als **alternatief** voor de annotations kan je ook volgende code toevoegen in de OnModelCreating() methode van de context:

```
modelBuilder.Entity<Voorraad>().Property(a => a.MagazijnNr).IsConcurrencyToken();
```



```
modelBuilder.Entity<Voorraad>().Property(a => a.CursusNr).IsConcurrencyToken();  
modelBuilder.Entity<Voorraad>().Property(a => a.AantalStuks).IsConcurrencyToken();  
modelBuilder.Entity<Voorraad>().Property(a => a.RekNr).IsConcurrencyToken();
```

Bij het lezen van een record onthoudt EF Core nu de waarden van de kolommen waarbij de annotation [ConcurrencyCheck] staat vermeld.

Bij het wijzigen van het record stuurt EF Core volgend SQL-statement naar de database :

```
update Voorraden  
set AantalStuks = @0  
where MagazijnNr=@1  
and CursusNr=@2  
and AantalStuks = @3  
and RekNr = @4
```

In dit statement krijgt parameter @0 de waarde van het nieuwe aantal stuks. De parameters @1, @2, @3 en @4 krijgen de originele waarden van de kolommen.

Er zal dus pas een wijziging doorgevoerd worden als de waarden van de kolommen bij het updaten nog steeds dezelfde zijn als bij het inlezen van het record.

Als dat niet het geval is – iemand anders heeft de waarden intussentijd veranderd – dan wordt er een DbUpdateConcurrencyException geworpen.

We passen de code nu aan.

- Wijzig de method VoorraadBijvulling als volgt :

```
static void VoorraadBijvulling(int cursusNr, int magazijnNr, int aantalStuks)  
{  
    using (var context = new EF0pleidingenContext())  
    {  
        var voorraad = context.Voorraden.Find(magazijnNr, cursusNr);  
        if (voorraad != null)  
        {  
            voorraad.AantalStuks += aantalStuks;  
            Console.WriteLine("Pas nu de voorraad aan met de Server Explorer,"  
                             + " druk daarna op Enter");  
            Console.ReadLine();  
            try  
            {  
                context.SaveChanges();  
            }  
            catch (DbUpdateConcurrencyException)  
            {  
                Console.WriteLine("Voorraad werd intussen door een andere"  
                                   + " applicatie aangepast.");  
            }  
        }  
        else  
            Console.WriteLine("Voorraad niet gevonden");  
    }  
}
```

We proberen opnieuw uit.

- Bekijk de voorraad van cursus 10 in magazijn 1.
- Start het programma.
- Verhoog de voorraad van cursus 10 in magazijn 1 met 100.

- Op het moment dat je de boodschap 'Pas nu de voorraad...' krijgt schakel je over naar SSMS.
- Wijzig de voorraad van cursus 10 in magazijn 1 naar 666.
- Schakel terug over naar het programma en druk op Enter. Deze keer krijg je een foutmelding. De voorraad wordt niet gewijzigd maar blijft op de waarde 666 staan die je manueel hebt ingegeven.

### 14.3 Oplossing met timestampveld

Als een table geen timestamp-kolom heeft wordt het where-deel van het update-SQL-statement bij optimistic record locking langer naargelang de table meer kolommen bevat. Het uitvoeren van een update-SQL-statement met een lang where-deel is nadelig voor de performantie.

Een oplossing is het toevoegen van een timestamp-kolom. Bij iedere wijziging van een record plaatst de database zelf een andere waarde in deze timestamp-kolom.

Je moet dan in het where-deel van het update-statement niet meer op iedere kolomwaarde controleren of het door een andere gebruiker werd bijgewerkt, maar enkel op deze timestamp-kolom.

We proberen dit uit.

- Verwijder in de class Voorraad alle `[ConcurrencyCheck]` annotations.
- Je voegt nu aan de class Voorraad een timestamp-kolom toe :

```
[Table("CursusVoorraden")]
public class Voorraad
{
    public int MagazijnNr { get; set; }
    public int CursusNr { get; set; }
    [Required]
    public int AantalStuks { get; set; }
    [Required]
    public int RekNr { get; set; }
    [Timestamp]
    public byte[] Aangepast { get; set; }
}
```

(1)

- (1) Je voegt een extra veld *Aangepast* toe van het type `byte[]`. Dit veld krijgt de annotation `[Timestamp]`.

Opmerking : als **alternatief** voor de annotation kan je ook volgende code toevoegen in de `OnModelCreating()` methode van de context:

```
modelBuilder.Entity<Voorraad>().Property(a => a.Aangepast).IsRowVersion();
```

We maken nu een nieuwe migration aan om de extra kolom toe te voegen.

- Start de package manager console en ga met de instructie `cd model` naar de map *Model*.
- Tik het commando `dotnet ef migrations add mettimestamp` om de migration aan te maken.
- Je voert de migration door via het commando `dotnet ef database update`
- Controleer de tabelstructuur.

De tabelstructuur ziet er nu zo uit :

|   | Column Name | Data Type | Allow Nulls                         |
|---|-------------|-----------|-------------------------------------|
| 🔑 | MagazijnNr  | int       | <input type="checkbox"/>            |
| 🔑 | CursusNr    | int       | <input type="checkbox"/>            |
|   | AantalStuks | int       | <input type="checkbox"/>            |
|   | RekNr       | int       | <input type="checkbox"/>            |
|   | Aangepast   | timestamp | <input checked="" type="checkbox"/> |

We proberen opnieuw uit.

- Bekijk de voorraad van cursus 10 in magazijn 1.
- Start het programma.
- Verhoog de voorraad van cursus 10 in magazijn 1 met 100.
- Op het moment dat je de boodschap 'Pas nu de voorraad...' krijgt schakel je over naar SSMS.
- Wijzig de voorraad van cursus 10 in magazijn 1 naar 999.
- Schakel terug over naar het programma en druk op Enter. Je krijgt opnieuw een foutmelding. De voorraad wordt niet gewijzigd maar blijft op de waarde 999 staan die je manueel hebt ingegeven.

Via de logging-informatie zie je dat het update-statement vrij kort blijft :

```
UPDATE [CursusVoorraden] SET [AantalStuks] = @p0  
WHERE [MagazijnNr] = @p1 AND [CursusNr] = @p2 AND [Aangepast] = @p3;
```

## 14.4 Oefening

De gebruiker kan de voornaam van een klant corrigeren.

Je vraagt het klantnummer van de te wijzigen klant. Wordt er geen getal ingegeven of een foutief klantnummer dan geef je een foutmelding.

Je vraagt de nieuwe voornaam van de klant.

Tussen het lezen van de klant en het wijzigen van de klant, kan een andere gebruiker dezelfde klant wijzigen. Je vangt deze fout op met optimistic record locking. Wanneer zich een fout voordoet toon je de foutmelding "Een andere gebruiker wijzigde deze klant!".



## 15 Scaffolding

---

In hoofdstuk 3 bouwden we een database op, startend vanuit een aantal entityclasses en een dbcontext-class. Deze manier van werken noemen we code-first. Soms gebeurt het dat we reeds over een bestaande database beschikken. We kunnen dan op basis van deze database de entity classes laten aanmaken. Dit noemen we scaffolding.

### 15.1 De database bieren

We proberen dit uit aan de hand van een database *EFBieren*. In de folder met oefenbestanden vind je een databasescript *EFcreateBieren.sql* waarmee je deze database kan aanmaken.

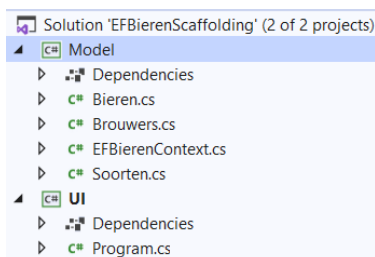
- Start SQL Server Management Studio en connecteer met de server LocalDb.
- Open het script *EFcreateBieren.sql* en voer het uit.
- Je kan deze database eventueel ook in Visual Studio toevoegen in de Server Explorer.

We zullen deze database nu gebruiken in een nieuwe solution.

- Maak een nieuwe console app *UI* in een nieuwe solution *EFBierenScaffolding*.
- Voeg er ook een class library (.NET Core) aan toe met de naam *Model*.
- Voeg aan het project *Model* de nuget packages *Microsoft.EntityFrameworkCore*, *Microsoft.EntityFrameworkCore.SqlServer* en *Microsoft.EntityFrameworkCore.Design* toe. Kies telkens versie 3.1.5.

We gaan nu de database *EFBieren* scaffolden.

- Build de solution.
- Start de package manager console en ga met de instructie `cd model` naar de map *Model*.
- Tik het commando `dotnet ef dbcontext scaffold "Server=(localdb)\mssqllocaldb;Database=EFBieren;Trusted_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer`
- Controleer het resultaat :



### 15.2 Migrations

Op dit moment kunnen we nog geen migrations toepassen op onze database. Dit kan nochtans nodig zijn wanneer we in de toekomst bijvoorbeeld een extra tabel of extra veld in een tabel willen toevoegen.

- Tik in de Package Manager Console het commando `dotnet ef migrations add initial`

Het resultaat is een extra folder *Migrations* in het project *Model*.

- Open in de folder *Migrations* de nieuwe migrationfile. Deze start met een tijdstip en eindigt op *\_initial.cs*.

Zoals steeds bevat de migrationfile een method *Up()* en een method *Down()*. Aangezien de database reeds bestaat mag alle code uit de method *Up()* verdwijnen.

- Wis alle code in de method *Up()*.

We voeren nu de initiële migration uit. Op deze manier wordt in de database de tabel *\_\_EFMigrationsHistory* aangemaakt.

- Tik in de Package Manager Console het commando **dotnet ef database update**

De database is nu klaar om eventuele wijzigingen te ondergaan.

## 16 De change tracker

Bij het oproepen van de `SaveChanges()` method moet EF Core weten welke objecten werden gewijzigd om zo de gepaste SQL Statements (Update, Insert, Delete) te kunnen genereren. Dit gebeurt door middel van de `ChangeTracker` property van de `DbContext` die de gemaakte wijzigingen in de entities opvolgt.

De `ChangeTracker` property bevat een collectie met de naam `Entries` die alle op te volgen entities bevat.

Het opvolgen van de entities gebeurt vanaf het ogenblik dat die opgeladen wordt in de context. Bij het oproepen van de `SaveChanges()` method zal EF Core de correcte SQL statements uitvoeren op de database. De `SaveChanges()` method roept intern de `DetectChanges()` method op die de gewijzigde gegevens identificeert.

### 16.1 Statussen van de ChangeTracker

De `ChangeTracker` werkt met statussen die de soort van wijziging identificeert, het kan volgende statussen (uit de enum `Microsoft.EntityFrameworkCore.EntityState`) bevatten :

| Status    | Gevolg van de actie...                                                | SQL statement |
|-----------|-----------------------------------------------------------------------|---------------|
| Unchanged | Geen. De entity is ongewijzigd.                                       | -             |
| Added     | Add(). De entity is toegevoegd aan de context.                        | Insert        |
| Modified  | Wijziging van de waarde van één of meerdere property's van de entity. | Update        |
| Deleted   | Remove(). De entity werd verwijderd uit de context.                   | Delete        |
| Detached  | Geen. De entity wordt niet opgevolgd.                                 | -             |

### 16.2 Een voorbeeld

We bekijken in een voorbeeld de changetracker-statussen van enkele entities.

- Open de solution `EFCore`.
- Neem onderstaande code over in het hoofdprogramma en voer daarna uit :

```
static void Main(string[] args)
{
    using (var context = new EF0pleidingenContext())
    {
        Console.WriteLine("-----\nWijzigingen\n-----");

        // UnChanged
        var land0 = context.Landen.First();
        Console.WriteLine("\n" + land0.LandCode + " - " + land0.Naam +
            " - " + context.Entry(land0).State + "\n");           (1)

        // Added
        var land1 = new Land { LandCode = "AB", Naam = "abcdef" };
        context.Landen.Add(land1);
        Console.WriteLine("\n" + land1.LandCode + " - " + land1.Naam +
            " - " + context.Entry(land1).State + "\n");           (2)
    }
}
```

```

// Modified
var land2 = context.Landen.Where(c => c.LandCode == "FR")
                           .FirstOrDefault();
land2.Naam = "France";
Console.WriteLine("\n" + land2.LandCode + " - " + land2.Naam +
    " - " + context.Entry(land2).State + "\n");           (3)

// Deleted
var land3 = context.Landen.Where(c => c.LandCode == "LU")
                           .FirstOrDefault();
context.Landen.Remove(land3);
Console.WriteLine("\n" + land3.LandCode + " - " + land3.Naam +
    " - " + context.Entry(land3).State + "\n");           (4)

// Detached - Disconnected data
var land4 = new Land { LandCode = "XY", Naam = "xyz" };
Console.WriteLine("\n" + land4.LandCode + " - " + land4.Naam +
    " - " + context.Entry(land4).State + "\n");           (5)

Console.WriteLine("-----\nNa wijzigingen\n-----");
context.ChangeTracker.DetectChanges();
Console.WriteLine("\nHasChanges: {0}\n",
    context.ChangeTracker.HasChanges());                 (6)

foreach (var entry in context.ChangeTracker.Entries())    (7)
{
    Console.WriteLine("Entity: {0}, Status: {1}",
        entry.Entity.GetType().Name, entry.State);       (8)
    foreach (var x in entry.Properties)                   (9)
    {
        Console.WriteLine(
            $"Property '{x.Metadata.Name}' " +
            $"is {(x.IsModified ? "modified" : "not modified")} " +
            $"- Current value: '{x.CurrentValue}' " +
            $"- Original value: '{x.OriginalValue}'");
    }
    Console.WriteLine("\n");
}
}
}

```

In een eerste stuk van bovenstaand programma zorgen we ervoor dat vijf entiteiten een bepaalde status krijgen.

- (1) land0 is een ingelezen entiteit dat we verder niet gewijzigd hebben (=> unchanged)
- (2) land1 is een nieuw en toegevoegd land (=> added)
- (3) land2 is een land met een nieuwe naam (=> modified)
- (4) land3 is een land dat verwijderd werd (=> deleted)
- (5) land4 is een land dat niet tot de context behoort (=> detached)

Na de wijzigingen gebruiken we enkele ChangeTracker-methods.

- (6) De method DetectChanges controleert de entiteiten op eventuele veranderingen in status. De method HasChanges vertelt ons of er al dan niet veranderingen zijn.
- (7) We overlopen een lijst entity-entry's, verkregen door een oproep van de method Entries(). Dit zijn allemaal entiteiten die door de ChangeTracker worden gevolgd. De entiteit land4 is daar dus niet bij.
- (8) Per entiteit tonen we de status...



- (9) ...en voor elke property van de entiteit of deze al dan niet is gewijzigd, de oude en de nieuwe waarde.

Merk op dat we de method `SaveChanges()` niet oproepen in het programma. De wijzigingen worden dus niet opgeslagen.

### 16.3 Het `StateChanged` en `Tracked` event

De `ChangeTracker` beschikt over een aantal interessante events : `StateChanged` en `Tracked`. De event `StateChanged` wordt geraised wanneer de status van een entiteit verandert. `Tracked` treedt op wanneer een entiteit door de `ChangeTracker` gevolgd wordt.

In onderstaand programma koppelen we aan beide events een eventhandler zodat we goed kunnen zien wanneer welke event zich voordoet.

- Voeg aan `EFOpleidingenContext.cs` onderstaande vetgedrukte code toe :

```
public class EFOpleidingenContext : DbContext
{
    ...
    public EFOpleidingenContext()
    {
        ChangeTracker.StateChanged += StateChanged;           (1)
        ChangeTracker.Tracked += Tracked;                     (2)
    }
    public EFOpleidingenContext(DbContextOptions<EFOpleidingenContext> options) :
        base(options)
    {
        ChangeTracker.StateChanged += StateChanged;           (1)
        ChangeTracker.Tracked += Tracked;                     (2)
    }
    private void StateChanged(object sender, EntityStateChangedEventArgs e)
    {
        Console.WriteLine("Status is gewijzigd" +
            $" van {e.OldState} naar {e.NewState}");           (3)
    }
    private void Tracked(object sender, EntityTrackedEventArgs e)
    {
        Console.WriteLine("Nieuw te volgen entity");           (4)
    }
}
```

- (1) In beide constructors koppelen we een eventhandler `StateChanged` aan de gelijknamige event.
- (2) Ook aan de event `Tracked` koppelen we een eventhandler.
- (3) In `StateChanged()` tonen we de oude en de nieuwe status.
- (4) In `Tracked()` geven we aan dat een entiteit wordt gevolgd.

- Probeer uit. De eventhandler zorgen voor extra informatie.



## 17 ORM in Entity Framework

---

In dit laatste hoofdstuk bekijken we enkele speciale gevallen binnen ORM. We leren werken met owned types of ook wel complex types genoemd. Verder bekijken we ook inheritance en associaties binnen EF Core.

### 17.1 Owned types mapping

Een owned type is een property van een entiteit waarin specifieke gegevens worden bewaard. Een adres kan bijvoorbeeld een owned type zijn.

Een owned type heeft **geen sleutel** en kan dus **niet onafhankelijk** bestaan. Het kan enkel een property zijn van een entiteit of van een ander owned type. Je kan ook **geen associaties** leggen van en naar owned types. Owned types kunnen ook **niet null** zijn en kunnen ook **niet erven** van andere owned types. Je definieert ze in een class.

Het voordeel van een owned type is dat we deze gegroepeerde gegevens in meerdere entiteiten kunnen hergebruiken of meerdere keren in éénzelfde entiteit.

Zowel een Docent als een Campus krijgen een Adres, bestaande uit een straat, huisnummer, postcode en een gemeente. Zonder owned types zou dit betekenen dat we in beide classes 4 property's moeten toevoegen. Een betere oplossing is dus deze 4 property's één keer te beschrijven in een class Adres. Docent en Campus hebben dan elk één property van het type Adres.

#### 17.1.1 Een adres property

We proberen dit uit in de solution *EFCore*.

In de class *Campus* vinden we de property's Straat, Huisnummer, Postcode en Gemeente. Dit zijn allemaal property's die te maken hebben met een adres. We vervangen deze property's door een owned type met de naam Adres.

In de class *Docent* willen we twee soorten adresgegevens toevoegen : ThuisAdres en VerblijfsAdres.

- Voeg in het project Model onderstaande class Adres toe in de folder Entities :

```
public class Adres
{
    public Adres() { }

    public Adres(string straat, string huisnummer,
                string postcode, string gemeente) {
        Straat = straat;
        Huisnummer = huisnummer;
        Postcode = postcode;
        Gemeente = gemeente;
    }

    public string Straat { get; set; }
    public string Huisnummer { get; set; }
    public string Postcode { get; set; }
    public string Gemeente { get; set; }
    public string Adreslijn => $"{Straat} {Huisnummer}\n{Postcode} {Gemeente}";
}
```

Opgepast : wanneer je een geparameterizeerde constructor toevoegt moet je ook een parameterloze constructor toevoegen. EF Core heeft deze achter de schermen nodig.

We voegen nu aan Campus en Docent extra Adres-property's toe.

- **Vervang** in de class Campus de property's Straat, Huisnummer, Postcode en Gemeente door een property Adres :

```
public class Campus
{
    ...
    public Adres Adres { get; set; }
    ...
}
```

- Voeg aan de class Docent de property's ThuisAdres en VerblijfsAdres toe :

```
public class Docent
{
    ...
    public Adres ThuisAdres { get; set; }
    public Adres VerblijfsAdres { get; set; }
    ...
}
```

- Verwijder in *EFOpleidingenContext.cs* in de method *OnModelCreating()* bij de seeding van de campussen de gegevens voor de straat, het huisnummer, de postcode en de gemeente.
- Breng eventueel ook in het testproject de nodige aanpassing aan.

### 17.1.2 Migration

Via een nieuwe migration kunnen we deze wijzigingen nu ook doorvoeren in de database.

- Start de Package Manager Console.
- Tik eventueel het commando `cd Model` om naar de projectfolder *Model* te gaan.
- Voer het onderstaande commando uit om de migration aan te maken :

```
dotnet ef migrations add metadres
```

Je krijgt nu een **foutmelding**. EF Core gaat ervan uit dat elk type in het model een entity is en elke entity moet een key hebben. In Adres is er geen property dat kan worden beschouwd als een key, maar hier wil je ook geen key, want eigenlijk is Adres geen Entity Type, maar een Owned Type. We lossen dit op in de volgende paragraaf.

### 17.1.3 Mappen van owned types

Owned types kunnen door EF Core niet automatisch afgeleid worden. Om EF Core duidelijk te maken dat Adres geen entiteit is moeten we de owned types expliciet mappen in *OnModelCreating()*.

- Voeg in *EFOpleidingenContext.cs* in *OnModelCreating()* onderstaande code toe :

```
modelBuilder.Entity<Campus>().OwnsOne(s => s.Adres);
modelBuilder.Entity<Campus>().OwnsOne(s => s.Adres)
    .Property(b => b.Straat).HasColumnName("Straat");
modelBuilder.Entity<Campus>().OwnsOne(s => s.Adres)
    .Property(b => b.Huisnummer).HasColumnName("HuisNr");
modelBuilder.Entity<Campus>().OwnsOne(s => s.Adres)
    .Property(b => b.Postcode).HasColumnName("PostCd");
modelBuilder.Entity<Campus>().OwnsOne(s => s.Adres)
    .Property(b => b.Gemeente).HasColumnName("Gemeente");
```

```

modelBuilder.Entity<Docent>().OwnsOne(s => s.ThuisAdres);
modelBuilder.Entity<Docent>().OwnsOne(s => s.ThuisAdres)
    .Property(b => b.Gemeente).HasColumnName("GemeenteThuis");
modelBuilder.Entity<Docent>().OwnsOne(s => s.ThuisAdres)
    .Property(b => b.Huisnummer).HasColumnName("HuisNrThuis");
modelBuilder.Entity<Docent>().OwnsOne(s => s.ThuisAdres)
    .Property(b => b.Postcode).HasColumnName("PostCdThuis");
modelBuilder.Entity<Docent>().OwnsOne(s => s.ThuisAdres)
    .Property(b => b.Straat).HasColumnName("StraatThuis");

modelBuilder.Entity<Docent>().OwnsOne(s => s.VerblijfsAdres);
modelBuilder.Entity<Docent>().OwnsOne(s => s.VerblijfsAdres)
    .Property(b => b.Gemeente).HasColumnName("GemeenteVerblijf");
modelBuilder.Entity<Docent>().OwnsOne(s => s.VerblijfsAdres)
    .Property(b => b.Huisnummer).HasColumnName("HuisNrVerblijf");
modelBuilder.Entity<Docent>().OwnsOne(s => s.VerblijfsAdres)
    .Property(b => b.Postcode).HasColumnName("PostCdVerblijf");
modelBuilder.Entity<Docent>().OwnsOne(s => s.VerblijfsAdres)
    .Property(b => b.Straat).HasColumnName("StraatVerblijf");

```

We voeren nu de wijziging door naar de database.

- Tik nu in de package manager console het commando `dotnet ef migrations add metadres`.
- Voer de migration door met het commando `dotnet ef database update`.
- Controleer de tabel Campussen in de database. Je merkt geen veranderingen.
- Controleer de tabel Docenten in de database. Je ziet o.a. onderstaande kolommen.

```

GemeenteThuis (nvarchar(max), null)
HuisNrThuis (nvarchar(max), null)
PostCdThuis (nvarchar(max), null)
StraatThuis (nvarchar(max), null)
GemeenteVerblijf (nvarchar(max), null)
HuisNrVerblijf (nvarchar(max), null)
PostCdVerblijf (nvarchar(max), null)
StraatVerblijf (nvarchar(max), null)

```

## 17.2 Inheritance

In paragraaf 3.2 Inheritance maakte je al kennis met de 3 verschillende manieren waarop we inheritance kunnen nabootsen in een database. We proberen dit nu uit in EF Core.

### 17.2.1 Table per hierarchy (TPH)

Table per hierarchy is de default wijze waarop we met EF Core inheritance voorstellen in een database. Alle classes worden dus aan één tabel gekoppeld.

We proberen dit nu uit. We maken drie classes aan : *TPHCursus* en daarvan afgeleid, *TPHZelfstudieCursus* en *TPHKlassikaleCursus*.

- Voeg onderstaande classes *TPHCursus*, *TPHZelfstudieCursus* en *TPHKlassikaleCursus* toe in het project *Model* in de folder *Entities* :

```

[Table("TPHCursussen")]
public abstract class TPHCursus
{
    public int Id { get; set; }
    public string Naam { get; set; }
}

```

```

public class TPHZelfstudieCursus : TPHCursus
{
    public int AantalDagen { get; set; }
}

public class TPHKlassikaleCursus : TPHCursus
{
    public DateTime Van { get; set; }
    public DateTime Tot { get; set; }
}

```

Voor elk van deze classes voegen we in *EFOpleidingenContext.cs* een DbSet-property toe.

- Voeg onderstaande code toe in *EFOpleidingenContext.cs* :

```

public DbSet<TPHCursus> TPHCursussen { get; set; }
public DbSet<TPHKlassikaleCursus> TPHKlassikaleCursussen { get; set; }
public DbSet<TPHZelfstudieCursus> TPHZelfstudieCursussen { get; set; }

```

We voeren deze toevoeging nu door in de database via een nieuwe migration.

- Voeg een nieuwe migration toe met het commando  
dotnet ef migrations add tph
- Voer de migration door met het commando  
dotnet ef database update

Bekijk het resultaat in SSMS : er is één extra tabel *TPHCursussen* aangemaakt. Deze tabel heeft naast de property's uit *TPHKlassikaleCursus* én *TPHZelfstudieCursus* ook een veld *Discriminator*. Met dit veld wordt bepaald of het over een klassikale cursus gaat of een zelfstudiecursus. Voeg je een zelfstudiecursus toe dan komt de waarde 'TPHZelfstudieCursus' in het veld, voor een klassikale cursus is dit 'TPHKlassikaleCursus'.

We gebruiken deze nieuwe tabel in een voorbeeldprogramma.

- Vervang de code in het hoofdprogramma door onderstaande code :

```

using (var context = new EFOpleidingenContext())
{
    context.TPHCursussen.Add(new TPHKlassikaleCursus
    {
        Naam = "Frans in 24 uur",
        Van = DateTime.Today, Tot = DateTime.Today
    });
    context.TPHCursussen.Add(new TPHZelfstudieCursus
    {
        Naam = "Engels in 24 uur", AantalDagen = 1
    });
    context.SaveChanges();
}

```

- Probeer uit.

Het resultaat :

|   | Id | Naam             | Discriminator       | Van                         | Tot                         | AantalDagen |
|---|----|------------------|---------------------|-----------------------------|-----------------------------|-------------|
| 1 | 1  | Frans in 24 uur  | TPHKlassikaleCursus | 2019-10-14 00:00:00.0000000 | 2019-10-14 00:00:00.0000000 | NULL        |
| 2 | 2  | Engels in 24 uur | TPHZelfstudieCursus | NULL                        | NULL                        | 1           |

Ben je niet tevreden met de standaard naam voor de discriminator-kolom of met de waarden in dit veld dan kan je dit via fluent-api in de method *OnModelCreating()* aanpassen.

We wijzigen de naam van het veld *Discriminator* in *CursusType*, de inhoud van dit veld is ofwel K voor een klassikale cursus, Z voor een zelfstudiecursus.

- Voeg onderstaande code toe in *OnModelCreating()* in *EFOpleidingenContext.cs* :

```
modelBuilder.Entity<TPHCursus>()  
    .ToTable("TPHCursussen")  
    .HasDiscriminator<string>("CursusType")           (1)  
    .HasValue<TPHKlassikaleCursus>("K")              (2)  
    .HasValue<TPHZelfstudieCursus>("Z");              (3)
```

(1) Het discriminator-veld krijgt de naam *CursusType*.

(2) Een klassikale cursus krijgt de waarde K.

(3) Een zelfstudiecursus krijgt de waarde Z.

- Voeg een nieuwe migration toe met het commando

```
dotnet ef migrations add tph2
```

- Voer de migration door met het commando

```
dotnet ef database update
```

De gevraagde wijzigingen in de tabelstructuur zijn doorgevoerd. De inhoud van het discriminator-veld geldt enkel voor nieuwe records en is niet veranderd voor de reeds aanwezige records.

## 17.2.2 Table per type (TPT) en Table per concrete class (TPC)

TPT en TPC zijn niet standaard geïmplementeerd in EF Core.

## 17.3 Associaties tussen entity's

In paragraaf 4.3.2 legden we een één-op-veel relatie tussen Campus en Docent. We tonen nu een concreet voorbeeld van een veel-op-veel relatie en voegen ook een associatie toe vanuit een class naar zichzelf.

### 17.3.1 Veel-op-veel relatie

Zoals we reeds in paragraaf 3.3.2 vermeldde moet je om een veel-op-veel-associatie uit te drukken tussen twee classes een tussenclass gebruiken.

In een eerste voorbeeld leggen we een veel-op-veel relatie tussen Boeken en Cursussen. Een boek kan in meerdere cursussen gebruikt worden en in een cursus kunnen meerdere boeken gebruikt worden.

- Voeg in het project *Model* in de folder *Entities* onderstaande class *Boek* toe :

```
public class Boek  
{  
    public Boek()  
    {  
        BoekenCursussen = new List<BoekCursus>();  
    }  
    public int BoekNr { get; set; }  
    public string IsbnNr { get; set; }  
}
```

```

        public string Titel { get; set; }
        public virtual ICollection<BoekCursus> BoekenCursussen { get; set; }
    }

```

- Voeg in het project *Model* in de folder *Entities* onderstaande class *Cursus* toe :

```

public class Cursus
{
    public Cursus()
    {
        BoekenCursussen = new List<BoekCursus>();
    }

    public int CursusNr { get; set; }
    public string Naam { get; set; }
    public virtual ICollection<BoekCursus> BoekenCursussen { get; set; }
}

```

- Voeg in het project *Model* in de folder *Entities* een tussenclass *BoekCursus* toe :

```

public class BoekCursus
{
    public int CursusNr { get; set; }
    public int BoekNr { get; set; }
    public int VolgNr { get; set; }

    public Boek Boek { get; set; }
    public Cursus Cursus { get; set; }
}

```

In de contextclass voegen we nu drie DbSet's toe.

- Voeg onderstaande code toe in *EFOpleidingenContext.cs* :

```

public DbSet<Boek> Boeken { get; set; }
public DbSet<Cursus> Cursussen { get; set; }
public DbSet<BoekCursus> BoekenCursussen { get; set; }

```

De namen van de tabellen die horen bij de entiteiten, de primary en foreign keys, veldeigenschappen en seeding data voegen we ook toe in *OnModelCreating()*.

- Voeg onderstaande code toe in *EFOpleidingenContext.cs* in de method *OnModelCreating()* :

```

modelBuilder.Entity<Boek>().ToTable("Boeken"); (1)
modelBuilder.Entity<Cursus>().ToTable("Cursussen");
modelBuilder.Entity<BoekCursus>().ToTable("BoekenCursussen");

modelBuilder.Entity<Boek>().HasKey(c => c.BoekNr); (2)
modelBuilder.Entity<Cursus>().HasKey(c => c.CursusNr);
modelBuilder.Entity<BoekCursus>().HasKey(c => new { c.BoekNr, c.CursusNr });

modelBuilder.Entity<Cursus>().Property(b => b.Naam) (3)
    .IsRequired().HasMaxLength(50);
modelBuilder.Entity<Boek>().Property(b => b.Titel)
    .HasMaxLength(150);
modelBuilder.Entity<Boek>().Property(b => b.IsbnNr)
    .IsRequired().HasMaxLength(13);
modelBuilder.Entity<Boek> (
    b => { b.HasIndex(e => e.IsbnNr).IsUnique(); } ); (4)
modelBuilder.Entity<BoekCursus>() (5)
    .HasOne(x => x.Boek)
    .WithMany(y => y.BoekenCursussen)

```



```

        .HasForeignKey(x => x.BoekNr);
modelBuilder.Entity<BoekCursus>()
    .HasOne(x => x.Cursus)
    .WithMany(y => y.BoekenCursussen)
    .HasForeignKey(x => x.CursusNr);
modelBuilder.Entity<Boek>().HasData(
    new Boek { BoekNr = 1, IsbnNr = "0-0705918-0-6",
        Titel = "C++ : For Scientists and Engineers" },
    new Boek { BoekNr = 2, IsbnNr = "0-0788212-3-1",
        Titel = "C++ : The Complete Reference" },
    new Boek { BoekNr = 3, IsbnNr = "1-5659211-6-X",
        Titel = "C++ : The Core Language" },
    new Boek { BoekNr = 4, IsbnNr = "0-4448771-8-5",
        Titel = "Relational Database Systems" },
    new Boek { BoekNr = 5, IsbnNr = "1-5595851-1-0",
        Titel = "Access from the Ground Up" },
    new Boek { BoekNr = 6, IsbnNr = "0-0788212-2-3",
        Titel = "Oracle : A Beginner's Guide" },
    new Boek { BoekNr = 7, IsbnNr = "0-0788209-7-9",
        Titel = "Oracle : The Complete Reference" }
);
modelBuilder.Entity<Cursus>().HasData(
    new Cursus { CursusNr = 1, Naam = "C++" },
    new Cursus { CursusNr = 2, Naam = "Access" },
    new Cursus { CursusNr = 3, Naam = "Oracle" }
);
modelBuilder.Entity<BoekCursus>().HasData(
    new BoekCursus { CursusNr = 1, BoekNr = 1, VolgNr = 1 },
    new BoekCursus { CursusNr = 1, BoekNr = 2, VolgNr = 2 },
    new BoekCursus { CursusNr = 1, BoekNr = 3, VolgNr = 3 },
    new BoekCursus { CursusNr = 2, BoekNr = 4, VolgNr = 1 },
    new BoekCursus { CursusNr = 2, BoekNr = 5, VolgNr = 2 },
    new BoekCursus { CursusNr = 3, BoekNr = 4, VolgNr = 1 },
    new BoekCursus { CursusNr = 3, BoekNr = 5, VolgNr = 2 },
    new BoekCursus { CursusNr = 3, BoekNr = 6, VolgNr = 3 }
);

```

- (1) We voorzien de tabellen die bij de entiteiten horen van een naam.
- (2) We bepalen de primary keys.
- (3) Een aantal velden krijgen een maximale lengte of zijn verplicht in te vullen.
- (4) We leggen een index op IsbnNr die uniek moet zijn.
- (5) We bepalen een aantal foreign keys.
- (6) We seeden de tabel Boeken...
- (7) ..., de tabel Cursussen...
- (8) ...en de tabel BoekenCursussen.

We kunnen de wijzigingen nu via een nieuwe migration doorvoeren.

- Voeg een nieuwe migration toe met het commando  
dotnet ef migrations add boekencursussen
- Voer de migration door met het commando  
dotnet ef database update

In het hoofdprogramma toon je nu alle cursussen en de boeken die daarin worden gebruikt.

- Vervang de code in het hoofdprogramma door onderstaande code en probeer uit :

```

using (var context = new EFopleidingenContext()) {
    var query = from cursus in context.Cursussen
                .Include("BoekenCursussen.Boek")
                orderby cursus.Naam
                select cursus;

    foreach (var cursus in query) {
        Console.WriteLine(cursus.Naam);
        foreach (var boekCursus in cursus.BoekenCursussen)
            Console.WriteLine("\t{0}:{1}", boekCursus.VolgNr,
                              boekCursus.Boek.Titel);
    }
}

```

Het resultaat (zonder tracking-informatie) :

```

Access
1:Relational Database Systems
2:Access from the Ground Up
C++
1:C++ : For Scientists and Engineers
2:C++ : The Complete Reference
3:C++ : The Core Language
Oracle
1:Relational Database Systems
2:Access from the Ground Up
3:Oracle : A Beginner's Guide

```

In een tweede voorbeeld voegen we een extra boek toe aan de cursus C++.

- Vervang de code in het hoofdprogramma door onderstaande code en probeer uit :

```

var nieuwBoek = new Boek() { IsbnNr = "0-201-70431-5",
                             Titel = "Modern C++ Design" };           (1)

var transactionOptions = new TransactionOptions
{
    IsolationLevel = IsolationLevel.Serializable
};

using (var transactionScope = new TransactionScope(                     (2)
    TransactionScopeOption.Required, transactionOptions))
{
    using (var context = new EFopleidingenContext())
    {
        var query = from cursus in context.Cursussen.Include("BoekenCursussen")
                    where cursus.Naam == "C++"
                    select new {
                        Cursus = cursus,
                        HoogsteVolgNr = cursus.BoekenCursussen.Max(bc => bc.VolgNr) (3)
                    };

        var queryResult = query.FirstOrDefault();                       (4)
        if (queryResult != null)                                       (5)
        {
            context.BoekenCursussen.Add(new BoekCursus {              (6)
                Boek = nieuwBoek,
                Cursus = queryResult.Cursus,
                VolgNr = queryResult.HoogsteVolgNr + 1
            });
            context.SaveChanges();                                     (7)
        }
        transactionScope.Complete();                                   (8)
    }
}

```

- (1) We maken een nieuwe Boek-entiteit.
- (2) Een nieuwe transactie wordt opgezet.
- (3) We maken een query die de cursus C++ ophaalt. Op basis van deze cursus wordt een nieuw object gemaakt dat bestaat uit de cursus zelf en het hoogste volgnummer van de bijhorende boeken.
- (4) We voeren de query uit.
- (5) Wanneer deze query een resultaat oplevert...
- (6) ...voegen we in de tussentabel *BoekenCursussen* een record toe. Boek en cursus kennen we reeds, het nieuwe volgnummer is het nummer uit de query plus 1.
- (7) We bewaren het toegevoegde record...
- (8) ...en beëindigen de transactie.

In een volgend voorbeeld leggen we een veel-op-veel relatie tussen activiteiten en docenten.

- Voeg een nieuwe class *Activiteit* toe :

```
[Table("Activiteiten")]
public class Activiteit
{
    public Activiteit() {
        DocentenActiviteiten = new List<DocentActiviteit>();
    }

    [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int ActiviteitId { get; set; }
    [Required]
    [MaxLength(50)]
    public string Naam { get; set; }
    public virtual ICollection<DocentActiviteit> DocentenActiviteiten
        { get; set; }
}
```

- (1) De bijhorende tabel geef je via een annotation de naam Activiteiten
- (2) Je initialiseert de navigationproperty DocentenActiviteiten op een lege List<DocentActiviteit>
- (3) De id van de class is de PK met autonummering
- (4) De naam is verplicht in te vullen en maximaal 50 tekens lang

We passen de class Docent aan door er ook een navigationproperty DocentenActiviteiten aan toe te voegen.

- Wijzig Docent.cs als volgt :

```
public partial class Docent
{
    public Docent() {
        DocentenActiviteiten = new List<DocentActiviteit>();
    }
    ...
    public virtual ICollection<DocentActiviteit> DocentenActiviteiten
        { get; set; }
}
```

We voegen nu ook de tussenclass *DocentActiviteit* toe.

- Voeg onderstaande class toe in de folder *Entities* :

```
[Table("DocentenActiviteiten")]
public class DocentActiviteit
{
    public int DocentId { get; set; } // Key
    public int ActiviteitId { get; set; } // Key
    public int AantalUren { get; set; }
    public Docent Docent { get; set; }
    public Activiteit Activiteit { get; set; }
}
```

Tenslotte brengen we ook in de contextclass *EFOpleidingenContext* enkele wijzigingen aan.

- Wijzig de class *EFOpleidingenContext* als volgt aan :

```
public class EFOpleidingenContext : DbContext
{
    ...
    public DbSet<Activiteit> Activiteiten { get; set; }
    public DbSet<DocentActiviteit> DocentenActiviteiten { get; set; } (1)
    ...
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        ...
        modelBuilder.Entity<DocentActiviteit>()
            .HasKey(c => new { c.DocentId, c.ActiviteitId });
        modelBuilder.Entity<DocentActiviteit>()
            .HasOne(x => x.Docent)
            .WithMany(y => y.DocentenActiviteiten)
            .HasForeignKey(x => x.DocentId);

        modelBuilder.Entity<DocentActiviteit>()
            .HasOne(x => x.Activiteit)
            .WithMany(y => y.DocentenActiviteiten)
            .HasForeignKey(x => x.ActiviteitId);
    }
}
```

We kunnen de wijzigingen nu via een nieuwe migration doorvoeren.

- Voeg een nieuwe migration toe met het commando  
dotnet ef migrations add activiteiten
- Voer de migration door met het commando  
dotnet ef database update

We voegen nu via het hoofdprogramma enkele gegevens toe.

- Wijzig de code in de method *Main()* als volgt en voer uit :

```
using (var context = new EFOpleidingenContext())
{
    var campus = new Campus { Naam = "CC Wondelgem", Adres = new Adres
        { Straat = "Industrieweg", Huisnummer = "50", Postcode = "9000",
          Gemeente = "Wondelgem" } };

    var jean = new Docent { Voornaam = "Jean", Familienaam = "Smits",
        Wedde = 1000, InDienst = new DateTime(1966, 8, 1), HeeftRijbewijs =
        true, ThuisAdres = new Adres { Straat = "Keizerslaan", Huisnummer =
        "11", Postcode = "1000", Gemeente = "Brussel" }, Campus = campus };

    var kiekeboe = new Docent { Voornaam = "Marcel", Familienaam = "Kiekeboe",
```

```

        Wedde = 500, InDienst = new DateTime(1948, 10, 24), HeeftRijbewijs =
        true, ThuisAdres = new Adres { Straat = "Merholaan", Huisnummer = "1B",
        Postcode = "3000", Gemeente = "Zoersel" }, Campus = campus };

var activiteit1 = new Activiteit { Naam = "EHBO" };
var activiteit2 = new Activiteit { Naam = "Vergaderen" };
var activiteit3 = new Activiteit { Naam = "Overleggen" };
var activiteit4 = new Activiteit { Naam = "Studie" };           (1)

context.Campussen.Add(campus);
context.SaveChanges();

context.Docenten.Add(jean);
context.Docenten.Add(kiekeboe);
context.SaveChanges();

context.Activiteiten.Add(activiteit1);
context.Activiteiten.Add(activiteit2);
context.Activiteiten.Add(activiteit3);
context.Activiteiten.Add(activiteit4);
context.SaveChanges();                                       (2)

var da1 = new DocentActiviteit { DocentId = jean.DocentId,
    ActiviteitId = activiteit2.ActiviteitId };
var da2 = new DocentActiviteit { DocentId = jean.DocentId,
    ActiviteitId = activiteit3.ActiviteitId };
var da3 = new DocentActiviteit { DocentId = kiekeboe.DocentId,
    ActiviteitId = activiteit4.ActiviteitId };
var da4 = new DocentActiviteit { DocentId = kiekeboe.DocentId,
    ActiviteitId = activiteit1.ActiviteitId };
var da5 = new DocentActiviteit { DocentId = kiekeboe.DocentId,
    ActiviteitId = activiteit2.ActiviteitId };

context.AddRange(da1, da2, da3, da4, da5);
context.SaveChanges();                                       (3)

var docent = context.Docenten.Find(4);
docent.DocentenActiviteiten.Add(new DocentActiviteit { ActiviteitId = 3 });
context.SaveChanges();
    }

```

- (1) We declareren één campus, 2 docenten en 4 activiteiten
- (2) We voegen deze toe aan hun respectievelijke dbsets
- (3) We voegen enkele docentenactiviteiten toe door gebruik te maken van de id's
- (4) We koppelen docent met id 4 aan activiteit met id 3 via de navigationproperty Docenten-Activiteiten van de docent.

### 17.3.2 Een associatie naar dezelfde tabel

We bekijken nog een speciale associatie, namelijk een associatie vanuit en naar dezelfde class. Als voorbeeld beschouwen we een werknemerclass. Elke werknemer heeft een overste en die overste kan meerdere werknemers onder zich hebben.

- Voeg onderstaande class *Werknemer* toe :

```

public class Werknemer
{
    [Key]
    public int WerknemerId { get; set; }
    public string Voornaam { get; set; }
}

```

```

    public string Familienaam { get; set; }
    public virtual ICollection<Werknemer> Ondergeschikten { get; set; } =
        new List<Werknemer>();
    [InverseProperty("Ondergeschikten")]
    public virtual Werknemer Overste { get; set; }
}

```

(1)

(2)

- (1) De navigation property *Ondergeschikten* stelt de werknemers voor van een overste.
- (2) De navigation property *Overste* stelt de andere kant van de associatie voor van de navigation property die vermeld staat als parameter van de annotatie *[InverseProperty]*.

We voegen nu in de contextclass een extra DbSet toe.

- Voeg in de class *EFOpleidingenContext* onderstaande vetgedrukte code toe :

```

public class EFOpleidingenContext : DbContext
{
    ...
    public DbSet<Werknemer> Werknemers { get; set; }
    ...
}

```

We passen nu de database aan met een nieuwe migration.

- Voeg een nieuwe migration toe met het commando  
dotnet ef migrations add werknemers
- Voer de migration door met het commando  
dotnet ef database update

We voegen enkele werknemers toe via het hoofdprogramma.

- Wijzig de code in de method *Main()* als volgt en voer uit :

```

using (var context = new EFOpleidingenContext())
{
    Werknemer joe = new Werknemer
    { Voornaam = "Joe", Familienaam = "Dalton" };
    Werknemer averell = new Werknemer
    { Voornaam = "Averell", Familienaam = "Dalton", Overste = joe };

    context.Werknemers.Add(joe);
    context.Werknemers.Add(averell);
    context.SaveChanges();
}

```

Het resultaat :

| WerknemerId | Voornaam | Familienaam | OversteWerknemerId |
|-------------|----------|-------------|--------------------|
| 1           | Joe      | Dalton      | NULL               |
| 2           | Averell  | Dalton      | 1                  |

Je ziet dat EF Core zelf een kolomnaam verzonnen heeft die hoort bij de navigation property *Overste*, namelijk *OversteWerknemerId*. We kunnen deze kolomnaam ook zelf instellen, bijvoorbeeld in *Oversteld*.

- Wijzig de code in *Werknemer* als volgt :

```

public class Werknemer
{

```

```

[Key]
public int WerknemerId { get; set; }
public string Voornaam { get; set; }
public string Familienaam { get; set; }

public virtual ICollection<Werknemer> Ondergeschikten { get; set; } =
    new List<Werknemer>();
[InverseProperty("Ondergeschikten")]
[ForeignKey("OversteId")]
public virtual Werknemer Overste { get; set; }
public int? OversteId { get; set; }
}

```

(1)

(2)

(1) Met de annotation `ForeignKey` duiden we aan welke property de andere kant van de associatie is.

(2) De property `OversteId` is nullable omdat niet elke werknemer een overste heeft.

We passen opnieuw de database aan.

- Voeg een nieuwe migration toe met het commando

```
dotnet ef migrations add metoversteid
```

- Voer de migration door met het commando

```
dotnet ef database update
```

De tabel *Werknemers* beschikt nu over een kolom *Oversteld* maar alle informatie met betrekking tot overste en ondergeschikte(n) is nu wel verloren gegaan. We herstellen dit.

- Neem onderstaande code over in het hoofdprogramma en voer uit.

```

using (var context = new EFOPleidingenContext())
{
    Werknemer joe = context.Werknemers
        .Where(w => w.Voornaam == "Joe" && w.Familienaam == "Dalton")
        .FirstOrDefault();
    Werknemer averell = context.Werknemers
        .Where(w => w.Voornaam == "Averell" && w.Familienaam == "Dalton")
        .FirstOrDefault();
    averell.Overste = joe;
    context.SaveChanges();
}

```

Het resultaat in de tabel *Werknemers* :

| WerknemerId | Voornaam | Familienaam | Oversteld |
|-------------|----------|-------------|-----------|
| 1           | Joe      | Dalton      | NULL      |
| 2           | Averell  | Dalton      | 1         |

Voor de volledigheid geven we ook even aan dat je de associatie tussen de class *Werknemer* en dezelfde class *Werknemer* ook kan leggen zonder annotations maar met fluent api.

De class *Werknemer* ziet er dan zo uit :

```

public class Werknemer
{
    [Key]
    public int WerknemerId { get; set; }
    public string Voornaam { get; set; }
    public string Familienaam { get; set; }
}

```

```

    public virtual ICollection<Werknemer> Ondergeschikten { get; set; } =
        new List<Werknemer>();
    public virtual Werknemer Overste { get; set; }
    public int? OversteId { get; set; }
}

```

In de class *EFOpleidingenContext.cs* neem je volgende code over in de method *OnModelCreating()* :

```

modelBuilder.Entity<Werknemer>()
    .HasOne(x => x.Overste)
    .WithMany(y => y.Ondergeschikten)
    .HasForeignKey(x => x.OversteId)
    .HasConstraintName("FK_WerknemerOverste");

```

- In ons voorbeeld houden we het bij de methode met de annotations. Je hoeft dus niks te veranderen.

We proberen in enkele voorbeelden wel een paar selecties uit. Maar eerst voegen we data toe via het seeding-mechanisme.

- Voeg in de class *EFOpleidingenContext.cs* in de method *OnModelCreating()* de code toe die je vindt in het oefenbestand *werknemersdata.txt* :

```

var werknemers = new List<Werknemer>()
{

```

(1)

```

    new Werknemer { WerknemerId = 03, Voornaam = "Agustin",
        Familienaam = "Calleri" },
    new Werknemer { WerknemerId = 04, Voornaam = "Ai",
        Familienaam = "Sugiyama" },
    ...

```

```

    new Werknemer { WerknemerId = 198, Voornaam = "Yanina",
        Familienaam = "Wickmayer" },
    new Werknemer { WerknemerId = 199, Voornaam = "Yen-Hsun",
        Familienaam = "Lu" },
    new Werknemer { WerknemerId = 200, Voornaam = "Yung-Jan",
        Familienaam = "Chan" }
};

```

```

foreach (var werknemer in werknemers)
{
    if (werknemer.WerknemerId % 3 != 0)
        werknemer.OversteId = (1 + ((werknemer.Voornaam.Length *
            werknemer.Familienaam.Length) % werknemer.WerknemerId));
}

```

(2)

```

foreach (var werknemer in werknemers)
{
    if (werknemer.OversteId == werknemer.WerknemerId)
        werknemer.OversteId = null;
}

```

(3)

```

modelBuilder.Entity<Werknemer>().HasData(werknemers);

```

(4)

- (1) We maken een lijst met werknemerobjecten.
- (2) Twee derde van de werknemers bezorgen we min of meer random een overste.
- (3) Bij werknemers die overste geworden zijn van zichzelf zetten we de OversteId op null.

We voegen deze data nu toe aan de database via een nieuwe migration.

- Voeg een nieuwe migration toe met het commando  
dotnet ef migrations add werknemersdata



- Voer de migration door met het commando  
dotnet ef database update

In een eerste voorbeeld tonen we de werknemers die geen overste hebben.

- Neem onderstaande code over in het hoofdprogramma en voer uit :

```
using (var context = new EFOpleidingenContext())
{
    var query = from werknemer in context.Werknemers
                where werknemer.Overste == null
                orderby werknemer.Voornaam, werknemer.Familienaam
                select werknemer;

    foreach (var werknemer in query)
        Console.WriteLine($"{werknemer.Voornaam} {werknemer.Familienaam}");
}
```

In een tweede voorbeeld tonen we alle werknemers die wél een overste hebben met vermelding van die overste.

- Neem onderstaande code over in het hoofdprogramma en voer uit :

```
using (var context = new EFOpleidingenContext())
{
    var query = from werknemer in context.Werknemers.Include("Overste")
                where werknemer.Overste != null
                orderby werknemer.Voornaam, werknemer.Familienaam
                select werknemer;

    foreach (var werknemer in query)
    {
        Console.WriteLine("Werknemer : " + werknemer.Voornaam +
            werknemer.Familienaam + ", Overste : " +
            werknemer.Overste.Voornaam + werknemer.Overste.Familienaam);
    }
}
```

In een volgende voorbeeld tonen we de werknemers die overste zijn en hun ondergeschikten.

- Neem onderstaande code over in het hoofdprogramma en voer uit :

```
using (var context = new EFOpleidingenContext())
{
    var query = from overste in context.Werknemers.Include("Ondergeschikten")
                where overste.Ondergeschikten.Count != 0
                orderby overste.Voornaam, overste.Familienaam
                select overste;

    foreach (var overste in query)
    {
        Console.WriteLine("Overste : " + overste.Voornaam + " " +
            overste.Familienaam);

        foreach (var werknemer in overste.Ondergeschikten)
            Console.WriteLine("\tWerknemer : " + werknemer.Voornaam + " " +
                werknemer.Familienaam);
    }
}
```


In een laatste voorbeeld zorgen we ervoor dat werknemer 5 de overste wordt van werknemer 6.

- Neem onderstaande code over in het hoofdprogramma en voer uit :

```
using (var context = new EFopleidingenContext())
{
    var werknemer5 = context.Werknemers.Find(5);
    if (werknemer5 != null)
    {
        var werknemer6 = context.Werknemers.Find(6);
        if (werknemer6 != null)
        {
            werknemer5.Ondergeschikten.Add(werknemer6);
            context.SaveChanges();
        }
        else
            Console.WriteLine("Werknemer 6 niet gevonden.");
    }
    else
        Console.WriteLine("Werknemer 5 niet gevonden.");
}
```

## 17.4 Oefening 1

Je maakt in de solution *Taken* een class *Personeelslid* met volgende property's :

| Personeel                                                                                        |                |          |  |
|--------------------------------------------------------------------------------------------------|----------------|----------|--|
| Column Name                                                                                      | Condensed Type | Nullable |  |
|  PersoneelsNr | int            | No       |  |
| Voornaam                                                                                         | nvarchar(50)   | No       |  |
| ManagerNr                                                                                        | int            | Yes      |  |

Als het personeelslid een manager heeft dan bevat het veld *ManagerNr* het personeelsnummer van het de manager.

Je voegt volgende personeelsleden toe in de tabel *Personeelsleden*:

| PersoneelsNr | Voornaam | ManagerNr |
|--------------|----------|-----------|
| 1            | Diane    |           |
| 2            | Mary     | 1         |
| 3            | Jeff     | 1         |
| 4            | William  | 2         |
| 5            | Gerard   | 2         |
| 6            | Anthony  | 2         |
| 7            | Leslie   | 6         |
| 8            | July     | 6         |
| 9            | Steve    | 6         |
| 10           | Foon Yue | 6         |
| 11           | George   | 6         |
| 12           | Loui     | 5         |
| 13           | Pamela   | 5         |
| 14           | Larry    | 5         |
| 15           | Barry    | 5         |
| 16           | Andy     | 4         |
| 17           | Peter    | 4         |
| 18           | Tom      | 4         |
| 19           | Mami     | 2         |
| 20           | Yoshimi  | 19        |
| 21           | Martin   | 5         |

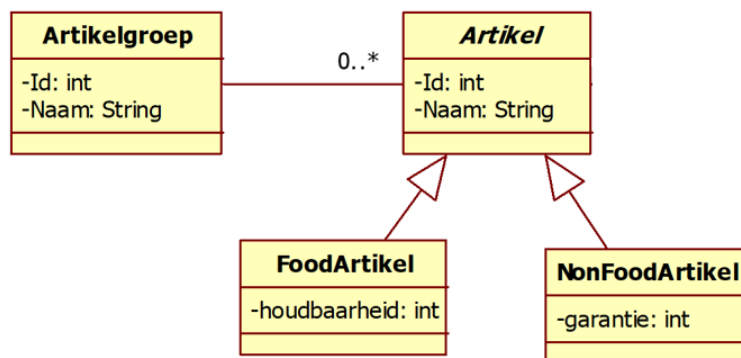
Je toont alle personeelsleden, beginnend met de hoogste in hiërarchie. Je toont per personeelslid de ondergeschikten. Bij elk lager niveau toon je een extra tab-teken voor de voornaam:

Diane  
Mary  
William  
Andy  
Peter  
Tom  
Gerard  
Loui  
Pamela  
Larry  
Barry  
Martin  
Anthony  
Leslie  
July  
Steve  
Foon Yue  
George  
Mami  
Yoshimi  
Jeff

Tip : voor deze oefening gebruik je best recursie én lazy loading.

## 17.5 Oefening 2

Je maakt in een nieuw project volgende classes en een bijbehorende database.





## 18 Appendix : Oplossing oefeningen

---

### 18.1 Databasescript EFTuincentrum

Voer het script uit zoals beschreven op p. 13.

### 18.2 Solution Taken en database EFBank

Maak een solution met de 3 projecten. Voeg de nodige nuget-packages toe en leg de references.

Voeg aan het project Model een class Klant en een class Rekening toe :

```
public class Klant
{
    public int KlantNr { get; set; }
    public string Voornaam { get; set; }
    public virtual ICollection<Rekening> Rekeningen { get; set; }
}

public class Rekening
{
    public string RekeningNr { get; set; }
    public int KlantNr { get; set; }
    public decimal Saldo { get; set; }
    public char Soort { get; set; }
    public virtual Klant Klant { get; set; }

    public void Storten(decimal bedrag) { Saldo += bedrag; }
}
```

Voeg aan het project Model een DbContext-class toe :

```
public class EFTakenContext : DbContext
{
    public DbSet<Rekening> Rekeningen { get; set; }
    public DbSet<Klant> Klanten { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder
        optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
            "Server=(localdb)\\mssqllocaldb;Database=EFBank;Trusted_Connection=true;"
            ,options => options.MaxBatchSize(150));
    }
}
```

### 18.3 Initiële migration

Wijzig de classes Klant en Rekeningen als volgt :

```
public class Klant
{
    [Key]
    public int KlantNr { get; set; }
    [Required]
    public string Voornaam { get; set; }
    public virtual ICollection<Rekening> Rekeningen { get; set; }
}

public class Rekening
{
    [Required]
    public string RekeningNr { get; set; }
    public int KlantNr { get; set; }
    public decimal Saldo { get; set; }
    public char Soort { get; set; }
    public virtual Klant Klant { get; set; }

    public void Storten(decimal bedrag) { Saldo += bedrag; }
}
```

```

[Key]
public string RekeningNr { get; set; }
[Required]
public int KlantNr { get; set; }
[Required]
public decimal Saldo { get; set; }
[Required]
public char Soort { get; set; }
[ForeignKey("KlantNr")]
public virtual Klant Klant { get; set; }
public void Storten(decimal bedrag)
{
    Saldo += bedrag;
}
}

```

Maak een nieuwe migration en voer die uit door de database te updaten.

## 18.4 Seeding

Voeg een method *OnModelCreating()* toe in de DbContext-class :

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Klant>().HasData
    (
        new Klant { KlantNr = 1, Voornaam = "Marge" },
        new Klant { KlantNr = 2, Voornaam = "Homer" },
        new Klant { KlantNr = 3, Voornaam = "Lisa" },
        new Klant { KlantNr = 4, Voornaam = "Maggie" },
        new Klant { KlantNr = 5, Voornaam = "Bart" }
    );
    modelBuilder.Entity<Rekening>().HasData
    (
        new Rekening { RekeningNr = "123-4567890-02",
            KlantNr = 1, Saldo = 1000, Soort = 'Z' },
        new Rekening { RekeningNr = "234-5678901-69",
            KlantNr = 1, Saldo = 2000, Soort = 'S' },
        new Rekening { RekeningNr = "345-6789012-12",
            KlantNr = 2, Saldo = 500, Soort = 'S' }
    );
}

```

## 18.5 Logging

Breng de wijzigingen aan zoals weergegeven in paragraaf 6.1 Logging in Visual Studio.

## 18.6 References

Leg vanuit het project UI een reference naar het project Model.

## 18.7 Alfabetische klantenlijst met rekeningen

```

using (var entities = new EFTakenContext())
{
    var query = from klant in entities.Klanten.Include("Rekeningen")
                orderby klant.Voornaam
                select klant;

    foreach (var klant in query)
    {
        Console.WriteLine(klant.Voornaam);
    }
}

```

```
        var totaleSaldo = Decimal.Zero;
        foreach (var rekening in klant.Rekeningen)
        {
            Console.WriteLine("{0}: {1}", rekening.RekeningNr, rekening.Saldo);
            totaleSaldo += rekening.Saldo;
        }
        Console.WriteLine("Totaal: {0}", totaleSaldo);
        Console.WriteLine();
    }
}
```

## 18.8 Zichtrekening toevoegen

```
using (var entities = new EFTakenContext())
{
    var query = from klant in entities.Klanten.Include("Rekeningen")
                orderby klant.Voornaam select klant;
    foreach (var klant in query)
    {
        Console.WriteLine("{0}: {1}", klant.KlantNr, klant.Voornaam);
    }
    try
    {
        Console.Write("KlantNr:");
        var klantNr = int.Parse(Console.ReadLine());
        var klant = entities.Klanten.Find(klantNr);

        if (klant == null)
        {
            Console.WriteLine("Klant niet gevonden");
        }
        else
        {
            Console.Write("RekeningNr:");
            var rekeningNr = Console.ReadLine();
            var rekening = new Rekening { RekeningNr = rekeningNr,
   Soort = 'Z' };

            klant.Rekeningen.Add(rekening);
            entities.SaveChanges();
        }
    }
    catch (FormatException)
    {
        Console.WriteLine("Tik een getal");
    }
}
```

## 18.9 Geld storten

```
using (var entities = new EFTakenContext())
{
    Console.Write("RekeningNr:");
    var rekeningNr = Console.ReadLine();
    var rekening = entities.Rekeningen.Find(rekeningNr);
    if (rekening == null)
    {
        Console.WriteLine("Rekening niet gevonden");
    }
    else
    {
        try
        {

```

```

        Console.WriteLine("Bedrag:");
        var bedrag = decimal.Parse(Console.ReadLine());
        if (bedrag <= Decimal.Zero)
        {
            Console.WriteLine("Tik een positief bedrag");
        }
        else
        {
            rekening.Storten(bedrag);
            entities.SaveChanges();
        }
    }
    catch (FormatException)
    {
        Console.WriteLine("Tik een bedrag");
    }
}

```

## 18.10 Klant verwijderen

```

try
{
    Console.WriteLine("KlantNr:");
    var klantNr = int.Parse(Console.ReadLine());
    using (var entities = new EFTakenContext())
    {
        var klant = entities.Klanten.Include("Rekeningen")
            .FirstOrDefault(k => k.KlantNr == klantNr);
        if (klant == null)
        {
            Console.WriteLine("Klant niet gevonden");
        }
        else
        {
            if (klant.Rekeningen.Count != 0)
            {
                Console.WriteLine("Klant heeft nog rekeningen");
            }
            else
            {
                entities.Klanten.Remove(klant);
                entities.SaveChanges();
            }
        }
    }
}
catch (FormatException)
{
    Console.WriteLine("Tik een getal");
}

```

## 18.11 Geld overschrijven

Voeg aan de class Rekening een method Overschrijven toe :

```

public void Overschrijven(Rekening naarRekening, decimal bedrag)
{
    if (this.Saldo < bedrag)
    {
        throw new Exception("Saldo ontoereikend");
    }
}

```



```
        else
        {
            this.Saldo -= bedrag;
            naarRekening.Saldo += bedrag;
        }
    }
}
```

Wijzig het hoofprogramma :

```
Console.Write("RekeningNr. van rekening:");
var vanRekeningNr = Console.ReadLine();

Console.Write("RekeningNr. naar rekening:");
var naarRekeningNr = Console.ReadLine();

try
{
    Console.Write("Bedrag:");
    var bedrag = decimal.Parse(Console.ReadLine());

    if (bedrag <= decimal.Zero)
        Console.WriteLine("Tik een positief bedrag");
    else
    {
        var transactionOptions = new TransactionOptions {
            IsolationLevel = IsolationLevel.RepeatableRead
        };

        using (var transactionScope = new TransactionScope(
            TransactionScopeOption.Required, transactionOptions))
        {
            using (var entities = new EFTakenContext())
            {
                var vanRekening = entities.Rekeningen.Find(vanRekeningNr);

                if (vanRekening == null)
                    Console.WriteLine("Van rekening niet gevonden");
                else
                {
                    var naarRekening = entities.Rekeningen.Find(naarRekeningNr);

                    if (naarRekening == null)
                        Console.WriteLine("Naar rekening niet gevonden");
                    else
                    {
                        try
                        {
                            vanRekening.Overschrijven(naarRekening, bedrag);
                            entities.SaveChanges();
                            transactionScope.Complete();
                        }
                        catch (Exception e)
                        {
                            Console.WriteLine(e.Message);
                        }
                    }
                }
            }
        }
    }
}
catch (FormatException)
{
    Console.WriteLine("Tik een bedrag");
}
```

## 18.12 Voornaam wijzigen

```

Console.Write("KlantNr:");
try
{
    var klantNr = int.Parse(Console.ReadLine());
    using (var entities = new EFTakenContext())
    {
        var klant = entities.Klanten.Find(klantNr);
        if (klant == null)
        {
            Console.WriteLine("Klant niet gevonden");
        }
        else
        {
            Console.Write("Voornaam:");
            klant.Voornaam = Console.ReadLine();
            entities.SaveChanges();
        }
    }
}
catch (DbUpdateConcurrencyException)
{
    Console.WriteLine("Een andere gebruiker wijzigde deze klant");
}
catch (FormatException)
{
    Console.WriteLine("Tik een getal");
}

```

In EFBankContext.cs :

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    ...
    modelBuilder.Entity<Klant>().Property(a => a.KlantNr).IsConcurrencyToken();
    modelBuilder.Entity<Klant>().Property(a => a.Voornaam).IsConcurrencyToken();
    ...
}

```

## 18.13 Personeelsleden – managers

De class Personeelslid :

```

public class Personeelslid
{
    public int PersoneelsNr { get; set; }
    public string Voornaam { get; set; }
    public int? ManagerNr { get; set; }

    public virtual ICollection<Personeelslid> Ondergeschikten { get; set; }
        = new List<Personeelslid>();
    public virtual Personeelslid Manager { get; set; }
}

```

Installeer de NuGet package *Microsoft.EntityFrameworkCore.Proxies*.

De contextclass :

```

public class EFTakenContext : DbContext
{

```

```
public DbSet<Personeelslid> Personeelsleden { get; set; }
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlServer(...)
        .UseLazyLoadingProxies();
}

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // Tabellen
    modelBuilder.Entity<Personeelslid>()
        .ToTable("Personeelsleden");

    // PK
    modelBuilder.Entity<Personeelslid>().HasKey(c => c.PersoneelsNr);

    // Properties
    modelBuilder.Entity<Personeelslid>().Property(b => b.Voornaam)
        .IsRequired()
        .HasMaxLength(50);

    // Associaties
    modelBuilder.Entity<Personeelslid>()
        .HasOne(x => x.Manager)
        .WithMany(y => y.Ondergeschikten)
        .HasForeignKey(x => x.ManagerNr)
        .HasConstraintName("FK_PersoneelslidManager");

    // Seeding
    modelBuilder.Entity<Personeelslid>().HasData
    (
        new Personeelslid { PersoneelsNr = 01, Voornaam = "Diane" },
        new Personeelslid { PersoneelsNr = 02, Voornaam = "Mary",
            ManagerNr = 1 },
        new Personeelslid { PersoneelsNr = 03, Voornaam = "Jeff",
            ManagerNr = 1 },
        new Personeelslid { PersoneelsNr = 04, Voornaam = "William",
            ManagerNr = 2 },
        new Personeelslid { PersoneelsNr = 05, Voornaam = "Gerard",
            ManagerNr = 2 },
        new Personeelslid { PersoneelsNr = 06, Voornaam = "Anthony",
            ManagerNr = 2 },
        new Personeelslid { PersoneelsNr = 07, Voornaam = "Leslie",
            ManagerNr = 6 },
        new Personeelslid { PersoneelsNr = 08, Voornaam = "July",
            ManagerNr = 6 },
        new Personeelslid { PersoneelsNr = 09, Voornaam = "Steve",
            ManagerNr = 6 },
        new Personeelslid { PersoneelsNr = 10, Voornaam = "Foon Yue",
            ManagerNr = 6 },
        new Personeelslid { PersoneelsNr = 11, Voornaam = "George",
            ManagerNr = 6 },
        new Personeelslid { PersoneelsNr = 12, Voornaam = "Loui",
            ManagerNr = 5 },
        new Personeelslid { PersoneelsNr = 13, Voornaam = "Pamela",
            ManagerNr = 5 },
        new Personeelslid { PersoneelsNr = 14, Voornaam = "Larry",
            ManagerNr = 5 },
        new Personeelslid { PersoneelsNr = 15, Voornaam = "Barry",
            ManagerNr = 5 },
        new Personeelslid { PersoneelsNr = 16, Voornaam = "Andy",
            ManagerNr = 4 },
    );
}
```

```

        new Personeelslid { PersoneelsNr = 17, Voornaam = "Peter",
                           ManagerNr = 4 },
        new Personeelslid { PersoneelsNr = 18, Voornaam = "Tom",
                           ManagerNr = 4 },
        new Personeelslid { PersoneelsNr = 19, Voornaam = "Mami",
                           ManagerNr = 2 },
        new Personeelslid { PersoneelsNr = 20, Voornaam = "Yoshimi",
                           ManagerNr = 19 },
        new Personeelslid { PersoneelsNr = 21, Voornaam = "Martin",
                           ManagerNr = 5 }
    );
    ...
}
...
}

```

Maak een nieuwe migration en update de database.

```

static void Main(string[] args)
{
    using (var context = new EFTakenContext())
    {
        var hoogstenInHierarchie =
            (from personeelslid in context.Personeelsleden
             where personeelslid.Manager == null
             select personeelslid).ToList();
        Afbeelden(hoogstenInHierarchie, 0);
    }
}

static void Afbeelden(List<Personeelslid> personeel, int insprong)
{
    foreach (var personeelslid in personeel)
    {
        Console.WriteLine(new String('\t', insprong));
        Console.WriteLine(personeelslid.Voornaam);
        if (personeelslid.Ondergeschikten.Count != 0)
        {
            Afbeelden(personeelslid.Ondergeschikten.ToList(), insprong + 1);
        }
    }
}

```

## 18.14 Artikels

De class Artikel :

```

[Table("Artikels")]
public abstract class Artikel
{
    public int Id { get; set; }
    public string Naam { get; set; }
    public virtual Artikelgroep Artikelgroep { get; set; }
    public int ArtikelgroepId { get; set; }
}

```

De class Artikelgroep :

```

[Table("Artikelgroepen")]
public class Artikelgroep
{
    public int Id { get; set; }
    public string Naam { get; set; }
}

```

```
    public ICollection<Artikel> Artikels { get; set; } = new List<Artikel>();  
}
```

De class FoodArtikel :

```
[Table("FoodArtikels")]  
public class FoodArtikel : Artikel  
{  
    public int Houdbaarheid { get; set; }  
}
```

De class NonFoodArtikel :

```
[Table("NonFoodArtikels")]  
public class NonFoodArtikel : Artikel  
{  
    public int Garantie { get; set; }  
}
```

De class EFTakenContext :

```
...  
public DbSet<Artikelgroep> Artikelgroepen { get; set; }  
public DbSet<Artikel> Artikels { get; set; }  
public DbSet<NonFoodArtikel> NonFoodArtikels { get; set; }  
public DbSet<FoodArtikel> FoodArtikels { get; set; }  
...
```

Voorbeeldprogramma :

```
static void Main(string[] args)  
{  
    using (var context = new EFBankContext())  
    {  
        var soepterrine = new NonFoodArtikel() {  
            Naam = "Villeroy & Boch", Garantie = 24  
        };  
        var grasmachine = new NonFoodArtikel() {  
            Naam = "SABO 40-spirit", Garantie = 60  
        };  
        var frietpatatjes = new FoodArtikel() {  
            Naam = "Frietaardappelen 5kg", Houdbaarheid = 1  
        };  
        var tuin = new Artikelgroep { Naam = "Tuinartikelen" };  
        var keuken = new Artikelgroep { Naam = "Keukenartikelen" };  
        tuin.Artikels.Add(grasmachine);  
        keuken.Artikels.Add(soepterrine);  
        keuken.Artikels.Add(frietpatatjes);  
        context.Artikelgroepen.Add(tuin);  
        context.Artikelgroepen.Add(keuken);  
        context.SaveChanges();  
    }  
}
```

## COLOFON

|                         |                              |
|-------------------------|------------------------------|
| Domeinexpertisemanager  | Jean Smits                   |
| Moduleverantwoordelijke | Johan Vandaele               |
| Auteurs                 | Johan Vandaele, Steven Lucas |
| Versie                  | 10/07/2020                   |

### Omschrijving module-inhoud

|                   |              |                                                                                                   |
|-------------------|--------------|---------------------------------------------------------------------------------------------------|
| Abstract          | Doelgroep    | .NET ontwikkelaar met C#                                                                          |
|                   | Aanpak       | Begeleide zelfstudie                                                                              |
|                   | Doelstelling | Gegevens bewaren, ophalen, bewerken en verwijderen in een database met behulp van EF Core.        |
| Trefwoorden       |              | EF Core                                                                                           |
| Bronnen/meer info |              | <a href="https://docs.microsoft.com/en-us/ef/core/">https://docs.microsoft.com/en-us/ef/core/</a> |