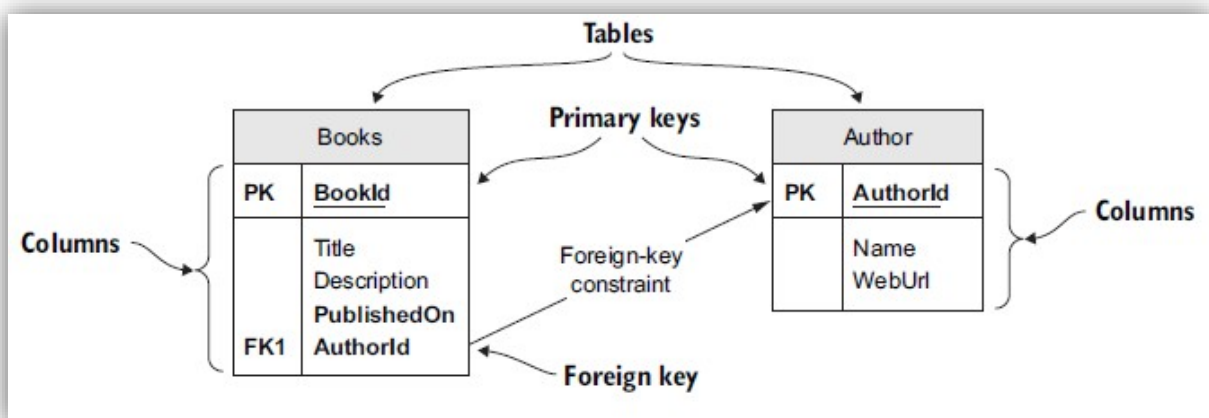


Entity Framework Core – Hoe werkt het ?

Entity Framework (EF) is een Object Relational Mapper (ORM) die programmeurs toelaat om op basis van .NET objecten te werken met een databank.

Modelleren

Aan de hand van een voorbeeld zullen we de werking van EF verduidelijken. Laten we starten met een eenvoudig relationeel databank model dat bestaat uit twee tabellen, een tabel voor boeken en een tabel voor auteurs. Een boek wordt uniek geïdentificeerd door middel van een Id (BookId) en dat is in de databank gemodelleerd door middel van een Primary Key (PK). Ook auteurs worden uniek geïdentificeerd door middel van een sleutel (PK), namelijk de AuthorId. Het verband tussen beiden is gelegd via een Foreign-key constraint die de AuthorId uit de Books tabel koppelt aan de AuthorId uit de Author tabel. Verder bevatten beide tabellen nog een aantal kolommen voor de attributen.



Een voorbeeld ziet er dan als volgt uit, waarbij de rijen uit de verschillende tabellen gekoppeld worden op basis van de AuthorId.

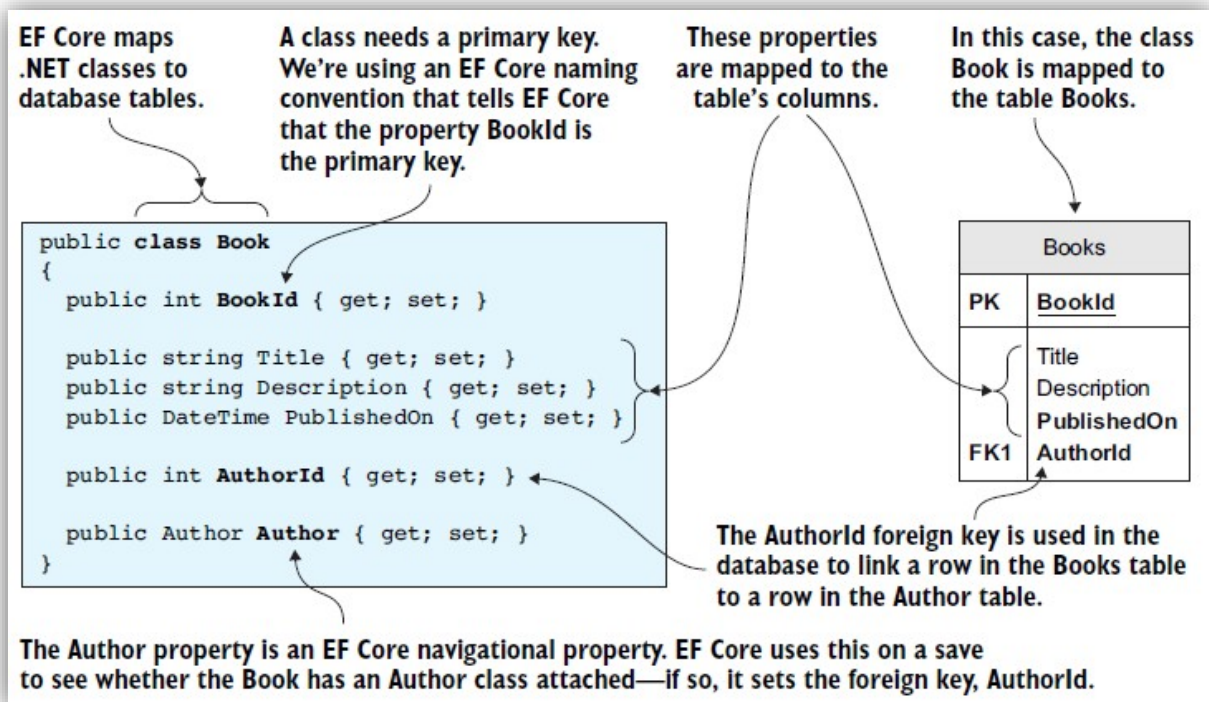
The diagram shows two data tables. The left table has columns: **Book**, **Title**, **Description**, **AvailableFrom**, and **Auth**. The right table has columns: **Auth**, **Name**, and **WebUrl**. A **Rows** label with arrows points to the **Auth** column in both tables, indicating the relationship between the rows.

Book	Title	Description	AvailableFrom	Auth
1	Refactoring	Improving h	08-Jul-1999	1
2	Patterns of Enterprise Ap	Written in d	15-Nov-2002	1
3	Domain-Driven Design	Linking bus	30-Aug-2003	2
4	Quantum Networking	Entanged q	01-Jan-2057	3

Auth	Name	WebUrl
1	Martin Fowler	http://ma
2	Eric Evans	http://don
3	Future Person	null

In C# werken we echter niet met tabellen, maar met klassen en objecten. In de volgende figuur zien we het verband tussen de klasse **Book** en de tabel **Books** in de databank. Bij het mappen van een klasse naar een tabel moeten we een primaire sleutel (PK) definiëren en dat kan op verschillende manieren. Hier in het voorbeeld maken we gebruik van EF Core Naming Convention die stelt dat een property met als naam de klasse, gevolgd door Id (**BookId**) automatisch als PK wordt gezien. De andere

attributen worden gekoppeld aan de kolommen in de tabel. En voor de link met het object Author wordt er gebruik gemaakt van de Foreign Key (FK) constraint. Het object Author noemen we een navigational property (een property om dus naar een andere klasse/object te gaan).



De klasse `Author` heeft als PK `AuthorId` die eveneens automatisch is toe gekent door EF.

```
public class Author
{
    public int AuthorId { get; set; }
    public string Name { get; set; }
    public string WebUrl { get; set; }
}
```

Holds the primary key of the `Author` row in the DB. Note that the foreign key in the `Book` class has the same name.

De toegang tot de databank wordt geregeld door middel van de klasse `AppDbContext` die overerft van de klasse `DbContext`. Door middel van de property `DbSet<Book> Books` leggen we het verband met de tabel `Books` in de databank. Het type `<Book>` dat we meegeven aan `DbSet` definieert het verband tussen de attributen van de klasse en de kolommen uit de tabel (zoals eerder beschreven). Door middel van de methode `OnConfiguration` kunnen we configuratie info definiëren, in dit voorbeeld beperken we ons door aan te geven dat we met SQL Server gaan werken (`optionBuilder.UseSqlServer`) en geven we ook de connectiestring mee.

You must have a class that inherits from the EF Core class **DbContext**. This class holds the information and configuration for accessing your database.

```
public class AppDbContext : DbContext
{
    private const string ConnectionString =
        @"Server = (local db)\nsssql local dv;
        Database=MyFirstEfCoreDb;
        Trusted_Connection=True";

    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder
            .UseSqlServer(connectionString);
    }

    public DbSet<Book> Books { get; set; }
}
```

The database connection string holds information about the database:

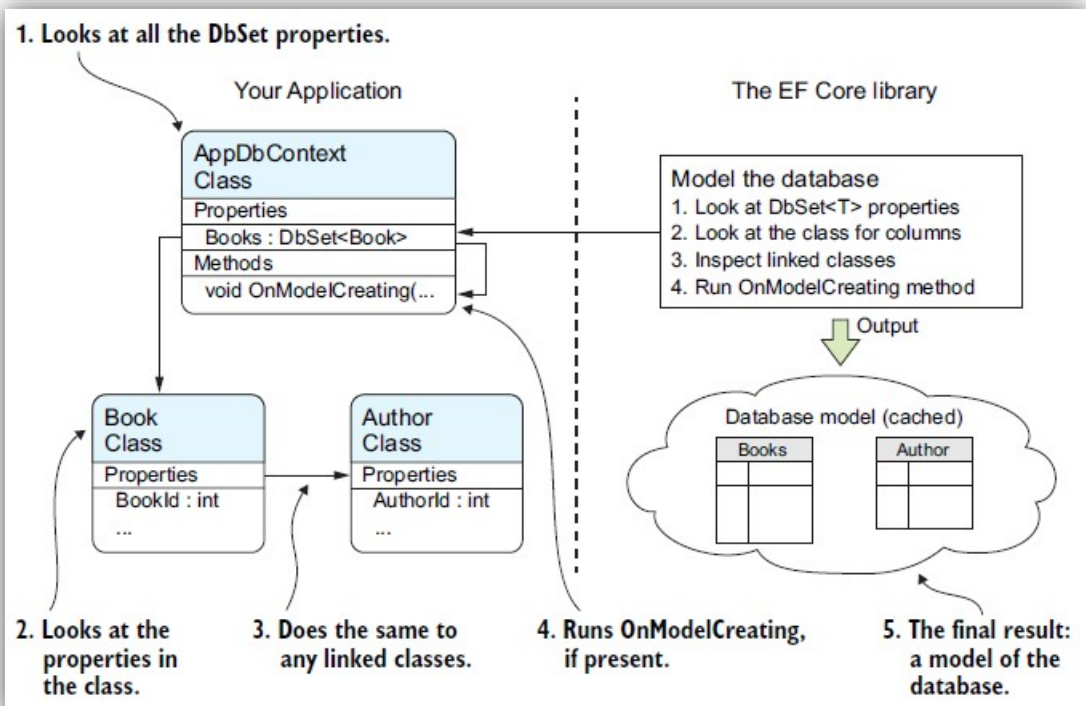
- How to find the database server
- The name of the database
- Authorization to access the database

In a console application, you configure EF Core's database options by overriding the **OnConfiguring** method. In this case you tell it you're using an SQL Server database by using the **UseSqlServer** method.

By creating a property called **Books** of type **DbSet<Book>**, you tell EF Core that there's a database table named **Books**, and it has the columns and keys as found in the **Book** class.

Our database has a table called **Author**, but you purposely didn't create a property for that table. EF Core finds that table by finding a navigational property of type **Author** in the **Book** class.

Van zodra de klasse **AppDbContext** wordt aangemaakt start er een modelleringsproces zoals beschreven in de volgende figuur. EF core bekijkt de **DbSet<T>** properties en linkt deze aan tabellen in de databank, analyseert de klassen om de link te leggen met de kolommen, analyseert de verbanden tussen de klassen (navigational properties) en voert uiteindelijk de **OnModelCreating** methode uit wat uiteindelijk resulteert in een **model** van de databank.



Gebruik

Eénmaal de link gelegd tussen de klassen en het databankmodel kunnen we aan de slag. Het volgende codefragment geeft weer hoe we gegevens uit de databank kunnen opvragen. We starten met het maken van een `AppDbContext` object waarmee we alle toegang tot de databank wordt geregeld. Het benaderen van een tabel gebeurt door middel van de `DbSets`, in dit voorbeeld `Books`. Indien we enkel gegevens wensen te lezen dan gebruiken we de methode `AsNoTracking`, die ervoor zorgt dat de objecten niet worden getraceerd (later hierover meer). Om ook de `Author` objecten op te vragen die bij de boeken horen moeten we dit expliciet meegeven en dat kan door de methode `Include`.

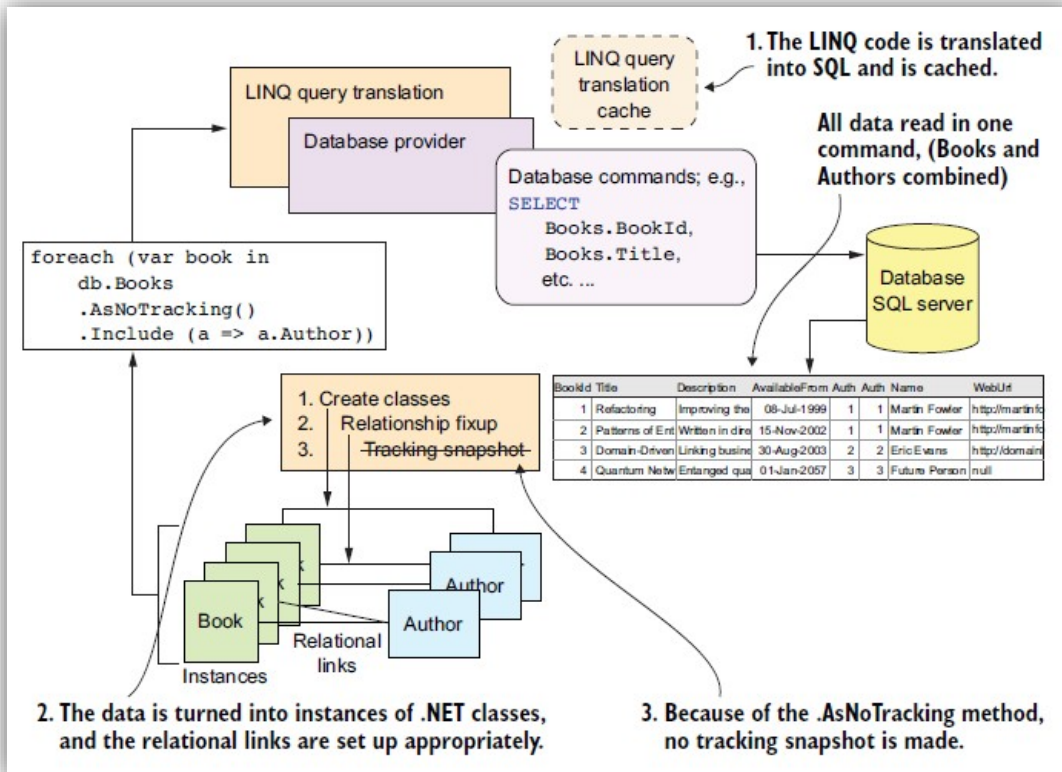
```
public static void ListAll()
{
    using (var db = new AppDbContext())
    {
        foreach (var book in
            db.Books.AsNoTracking()
                .Include(a => a.Author))
        {
            var webUrl = book.Author.WebUrl ?? null
                ? "- no web URL given -"
                : book.Author.WebUrl;
            Console.WriteLine(
                $"{book.Title} by {book.Author.Name}");
            Console.WriteLine("    " +
                "Published on " +
                $"{book.PublishedOn:dd-MMM-yyyy}" +
                $" {webUrl}");
        }
    }
}
```

You create the application's `DbContext` through which all database accesses are done.

Reads all the books. `AsNoTracking` indicates this is a read-only access.

The "include" causes the author information to be eagerly loaded with each book. See chapter 2 for more on this.

Het query proces wordt in de volgende figuur nog eens schematisch voorgesteld. We starten met een LINQ query die wordt omgezet in een SQL statement. Dit SQL statement wordt uitgevoerd in de databank en het resultaat wordt als een 'tabel' terug gegeven. Deze resultaten worden nu vertaald naar de .NET klassen, waarbij ook de relaties worden opgebouwd. Indien de `AsNoTracking` methode werd gebruikt zal er geen snapshot worden gemaakt van de opgevraagde objecten. Het resultaat is dan een collectie van objecten.



Wanneer we objecten wensen op te halen van de database dan zullen we wel gebruik maken van het trackingsysteem van EF. Van elk opgevraagd object wordt de toestand bijgehouden in de "context", dit kan doordat er een snapshot wordt gemaakt van de objecten op het moment dat ze worden opgevraagd. Elke verandering die we nu doen op één van deze objecten wordt gedetecteerd door het trackingsysteem waardoor EF weet wat er is veranderd en de nodige update queries kan opbouwen en uitvoeren.

In het volgende voorbeeld vragen we één specifiek boek op, namelijk het boek met als titel 'Quantum Networking'. Door middel van de Include-methode hebben we ook de info van de auteur. Daarna veranderen we het attribuut WebUrl van het object Author, deze verandering wordt door EF gedetecteerd en bijgehouden. Wanneer we nu de methode SaveChanges die bij onze context hoort (overgeerd van DbContext) uitvoeren worden alle veranderingen doorgevoerd naar de databank.

```
public static void ChangeWebUrl()
{
    Console.WriteLine("New Quantum Networking WebUrl > ");
    var newWebUrl = Console.ReadLine();

    using (var db = new AppDbContext())
    {
        var book = db.Books
            .Include(a => a.Author)
            .Single(b => b.Title == "Quantum Networking");

        book.Author.WebUrl = newWebUrl;
        db.SaveChanges();
        Console.WriteLine("... SaveChanges called.");

        ListAll();
    }
}
```

To update the database, you change the data that was read in.

Makes sure the author information is eager loaded with the book

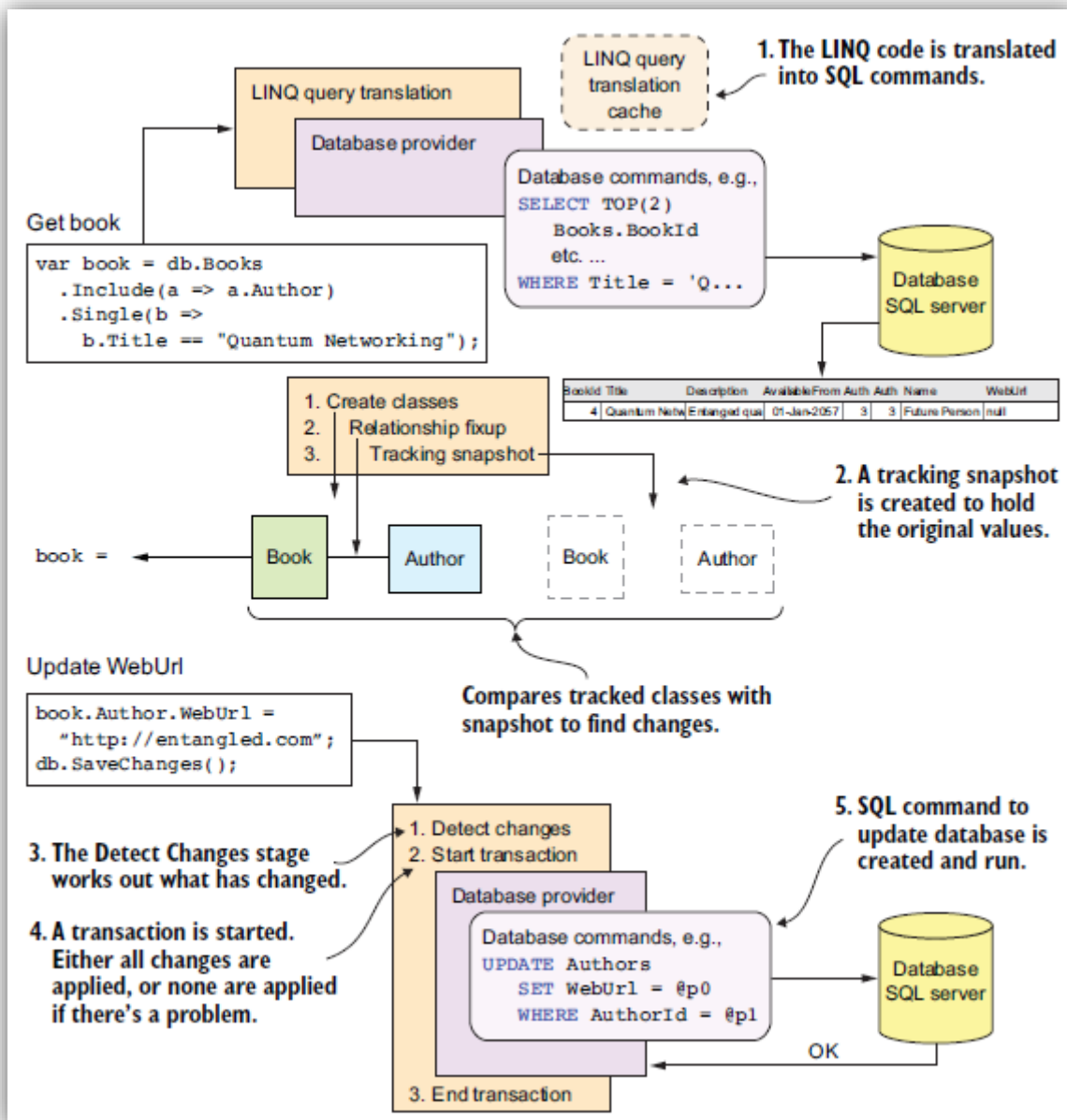
Selects only the book with the title Quantum Networking

SaveChanges tells EF Core to check for any changes to the data that has been read in and write out those changes to the database.

Lists all the book information

Reads in from the console the new URL

In de volgende figuur wordt het proces nog eens duidelijk geschetst. We starten met een LINQ query om een specifiek boek op te vragen. Van zodra de gegevens uit de databank worden omgezet naar de .NET objecten wordt er ook een snapshot gemaakt die de originele waarden bijhoudt. Na het aanpassen van het object en het oproepen van de functie `SaveChanges` starten de volgende handelingen. Eerst worden de veranderingen gedetecteerd door te vergelijken met de waarden in het snapshot. Daarna wordt er een transactie gestart met daarin de update statements en uiteindelijk wordt de transactie beëindigd. Aangezien er met transacties wordt gewerkt, zullen ofwel alle aanpassingen door worden gevoerd of geen enkele.



Opmerking : alle figuren komen uit het boek 'Entity Framework Core In Action'.

Referenties

Entity Framework Core In Action, Jon P. Smith, Manning

<https://blog.oneunicorn.com/2012/03/10/secrets-of-detectchanges-part-1-what-does-detectchanges-do/>