PROGRAMMEERTECHNIEKEN EN TESTEN–Databanken

JENS BAETENS

Databanken

Wat is een databank?

- Verzameling van gegevens die in een bepaalde structuur staat.
- Voorbeelden van Databases systemen zijn:
  - SQL server
  - MariaDB
  - MySql
  - MongoDB (NoSQL)
  - ….
- In deze cursus maken we gebruik van een RDBMS (relational database management system). Dit bestaat uit:
  - Tabellen
  - Kolommen
  - Rijen

SQL - Select

```
SELECT columnName
FROM tableName
WHERE condition
GROUP BY columnName
HAVING condition
ORDER BY columnName;
```

SQL - Insert

```
INSERT INTO tablename (columnName)
VALUES (value);
```

SQL - Delete

DELETE FROM tableName WHERE condition;

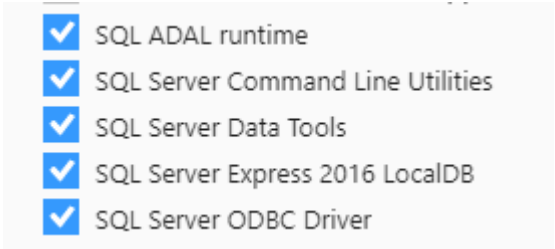
SQL - Update

UPDATE tableName SET columnName = value WHERE conditions;

Databanken in WPF/C#/.net Framework

Databank in VS

- Ga naar Tools > SQL server > new Query
  - Indien dit niet lukt check visual studio installer =>



- Kies een server => MSSQLLOCALDB
- Creëer een databank
  - CREATE DATABASE demo;
- Controleer in SQL Server Object Browser of de databank bestaat

Databanken aanspreken in C#

In C# hebben we 2 manieren die frequent gebruikt worden om databanken aan te spreken.

- ADO.NET
  - ActiveX Data Objects
  - Framework om queries te maken en te sturen naar RDBMS
  - Zelf queries schrijven

- Entity Framework
  - Is een ORM (Object Relational Mapping) framework
  - Minder repetitief werk.
  - Query-a-like syntax

ADO.net

```
private List<string> SearchUser(string name)
{
    List<string> names = new List<string>();
    SqlConnection connection =
        new SqlConnection("data source=(localdb)\\MSSQLLOCALDB;initial catalog=Messages;");

    SqlCommand command = new SqlCommand($"SELECT * FROM dbo.authors WHERE first_name = '{name}';", connection);

    connection.Open();
    SqlDataReader reader = command.ExecuteReader();
    while (reader.Read())
    {
        names.Add($"{reader[1]} {reader[2]}");
    }
    reader.Close();


    return names;
}
```

Connectie instellen

Connectie openen

SQL Query ingeven

Commando uitvoeren

Welk probleem zou hier kunnen voorvallen ?

SQL injection

```
1  SearchUser("Tom';DROP TABLE dbo.authors; --");
```

SELECT *
FROM dbo.authors
WHERE first_name = 'Tom';

DROP TABLE dbo.authors;

--' ;

Er bestaan technieken om ons hier tegen te wapenen.

Zaken zoals placeholders en SQLParameters kunnen er voor zorgen dat dit niet optreed

Maar het gevaar is er nog steeds aangezien we er steeds aan moeten denken!

ADO.NET Nadelen

- Fout gevoelig:
  - Applicatie compileert maar kan toch een exception geven op de query.

```
SqlCommand command = new SqlCommand($"SELECT * FROM dbo.authors WERE first_name = '{name}';", connection);
```

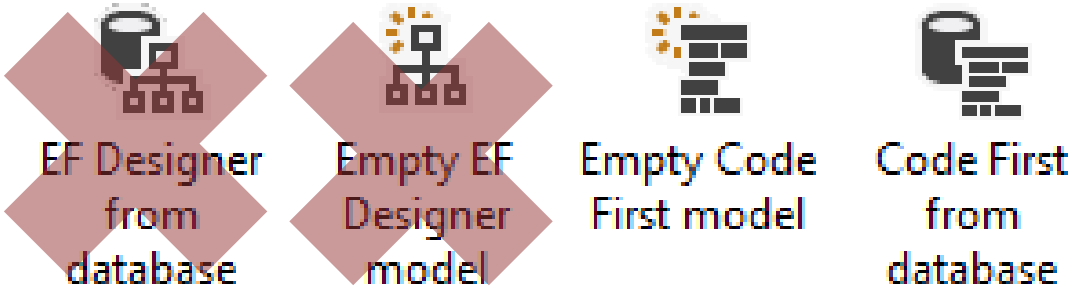
- Gevoelig voor SQL injection
- Veel repetitief werk.
  - Elke keer opnieuw moet de connectie opgezet worden. Dit kan wel in een aparte klasse maar nog steeds overhead aanwezig

EntityFramework

- Is een ORM (Object Relational Mapping) framework
  - We gaan SQL tabellen en relaties mappen op objecten in onze code.
- Geen SQL injection mogelijk
  - SQL injection wordt voor ons “gecheckt”
- Minder repetitief werk
  - Er wordt heel wat voor ons af gehandeld

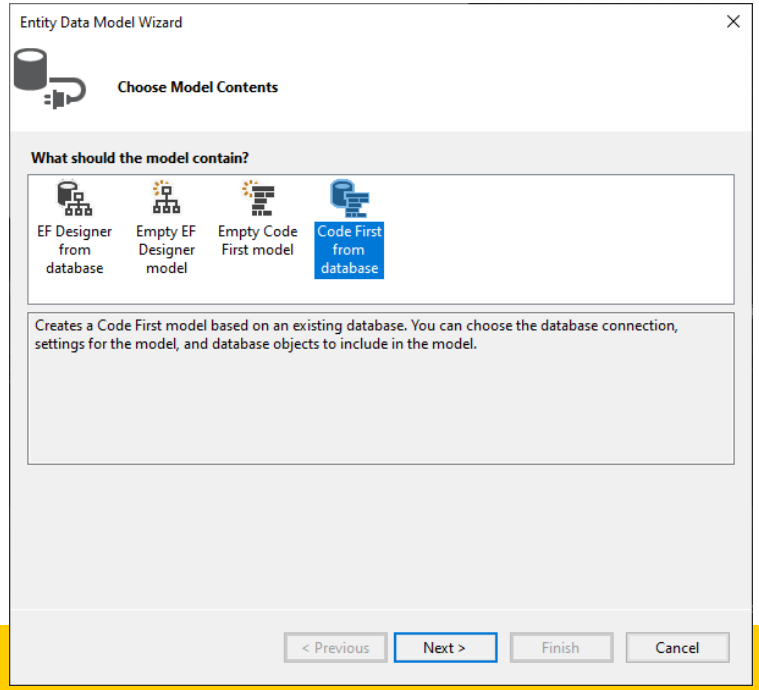
EntityFramework

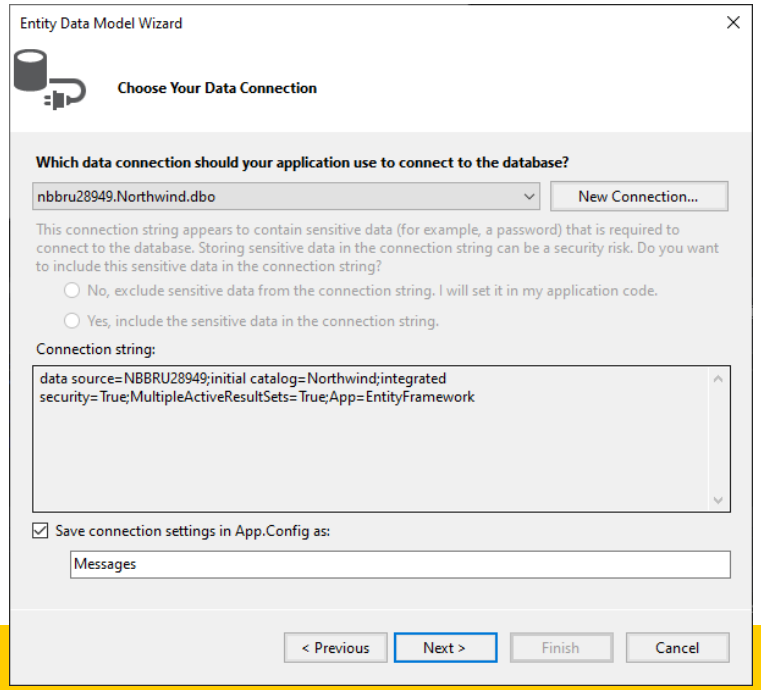
Er zijn 4 manieren om met EntityFramework te werken in .NET framework. Voor .NET CORE is dit anders



Code First from database

Op basis van een **bestaande database** gaan we onze code genereren.
Hiervoor moet de database reeds bestaande zijn!



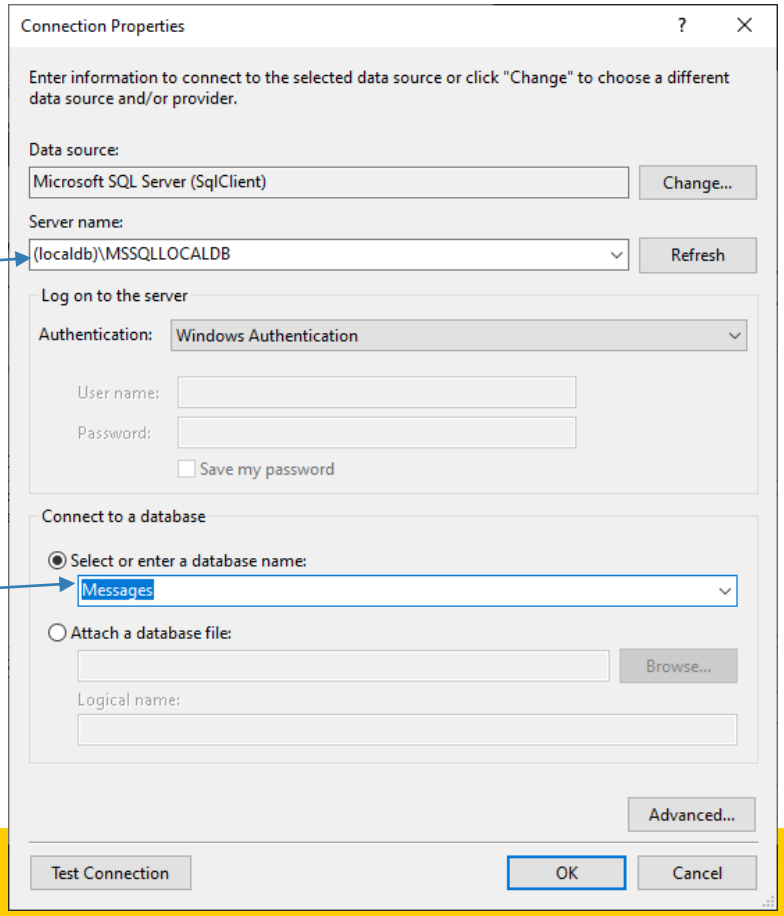


Code First from database

Maak indien nodig een nieuwe connectie aan.

Kies de servernaam
Op je lokale machine normaal gezien (localdb)\MSSQLLOCALDB.
Deze wordt niet altijd automatisch herkend

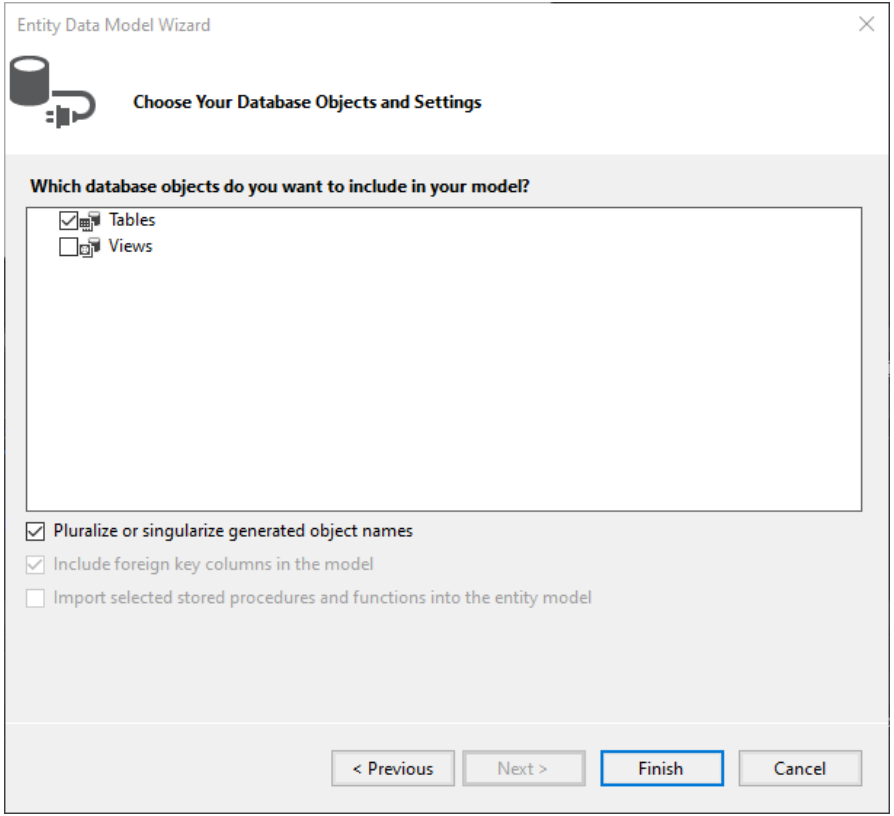
Kies de database die je wil gebruiken.



Code First from database

Kies de tabellen die overgenomen moeten worden.

Als alles goed gaat, zie je dat er na deze wizard, enkele klasses aangemaakt zijn.



Code First without database

- Wanneer er nog geen database is
- Database wordt gemaakt op basis van klasses
- **Heeft in dit vak de voorkeur.**

Dit kunnen we doen aan de hand van wizard.
MAAR eigenlijk zo simpel dat we dit niet nodig hebben.

Code First without database

Volgende zaken zijn nodig om dit te implementeren

- NuGet Packages
- DbContext
- ConnectionString

Zorg er verder voor dat je enkele dataklasses hebt met volgende eigenschappen

- Default constructor
- Public property met getter en setter voor elk field/member in je dataklasse

Welke NuGet Packages nodig

- **.Net framework**
  - **EntityFramework**
- .Net core
  - EntityFrameworkCore
  - EntityFrameworkCore.SqlServer
  - EntityFrameworkCore.Tools

DbContext

- Dit is het belangrijkste onderdeel om een database aan te spreken a.d.h.v. entitity framework
- Maak een **klasse** die **overerft** van **DbContext**.

DbContext - ConnectionString

- We moeten in onze DbContext klasse aangeven hoe we met de database zullen communiceren.
  - Waar staat de database
  - Hoe connecteren we
  - Welke “taal” wordt er door de database gesproken
  - Hoe noemt de database
  - ….

ConnectionString

- Is een string die aangeeft hoe we moeten **connecteren met een database**
- ConnectionString is afhankelijk van de data provider.
- Vb:
  - **OleDB**

    "Provider=Microsoft.Jet.OleDb.4.0;Data Source=D:\Data\Northwind.mdb"
  - **SQL Server**

    "Data Source=TestServer;Initial Catalog=Pubs;User ID=Dan;Password=training"

ConnectionString

- "Data Source=TestServer;Initial Catalog=Pubs;User ID=Dan;Password=training"
  - Data source:
    - Server waarop de Databank draait
    - Data Source=TestServer;
  - Initial Catalog
    - Databank die we op de server gaan gebruiken
    - Initial Catalog=Pubs;
  - User ID
    - Username om op de databank in te loggen wanneer deze beveiligd is
    - User ID=Dan;
  - Password
    - Password om op de databank in te loggen wanneer deze beveiligd is
    - Password=training

DbContext – ConnectionString definiëren

Het definiëren van de ConnectionString gebeurt in de **app.config** file.

Dit is een xml file waar we enkele configuraties kunnen definiëren.

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false" />
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.8" />
  </startup>
  <entityFramework>
    <providers>
      <provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
    </providers>
  </entityFramework>
  <connectionStrings>
    <add name="users" connectionString="data source = (localdb)\MSSQLLOCALDB; Initial catalog=users;" providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

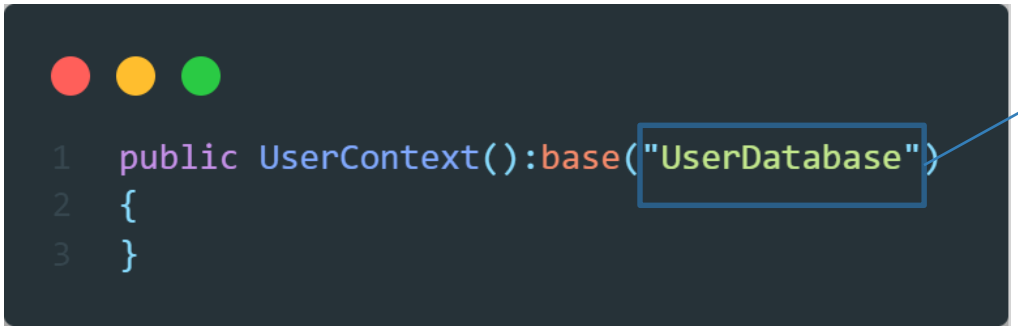
Name om later de connectionstring terug te vinden

Connectionstring met alle info

Providername, deze geeft aan welk type db we gebruiken

DbContext – ConnectionString instellen

We moeten nu nog aangeven dat de DbContext de juiste connectionstring gebruikt. Dit kunnen we door de juiste naam aan de base class door te geven.



DbContext – Tabellen defininiëren

- DbContext ≈ Database
- Database bestaat uit tabellen => DbContext bestaat uit ≈ tabellen
- Tabellen in DbContext zijn van type DbSet<T>

```
class UserContext : DbContext
{
    public UserContext() : base("UserDatabase")
    {
    }

    public DbSet<User> users { get; set; }
}
```

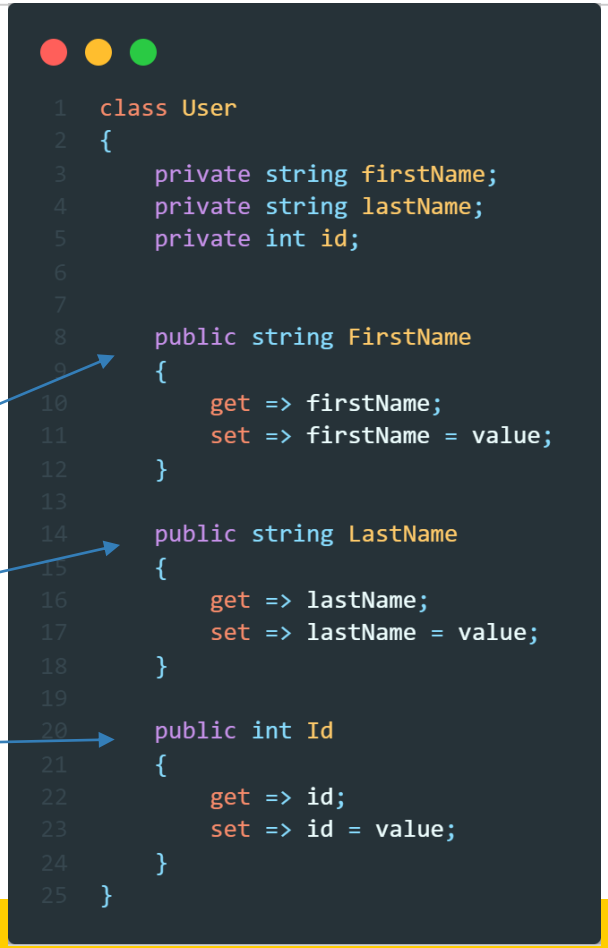
```
class User
{
    private string firstName;
    private string lastName;
    private int id;


    public string FirstName
    {
        get => firstName;
        set => firstName = value;
    }

    public string LastName
    {
        get => lastName;
        set => lastName = value;
    }

    public int Id
    {
        get => id;
        set => id = value;
    }
}
```

DbContext – Tabellen defininiëren

DbSet<T> ≈ Tabel
Tabel bestaat uit rijen en kolommen =>
DbSet<T> bestaat uit ≈rijen en ≈kolommen

DbSet<T> is een soort lijst (zoals List of Array) met elementen van type T waarbij T eender welk object kan zijn. Elk item in de lijst stelt een record of rij voor.

Properties van de de objecten T zijn kolommen.



```
class User
{
    private string firstName;
    private string lastName;
    private int id;


    public string FirstName
    {
        get => firstName;
        set => firstName = value;
    }

    public string LastName
    {
        get => lastName;
        set => lastName = value;
    }

    public int Id
    {
        get => id;
        set => id = value;
    }
}
```

Database constraints

In het vak data en informatieverwerking zagen we dat een tabel een aantal **constraints** kan hebben

- Primary key
- Not Null
- Foreign key
- …

Met behulp van entity framework kunnen we deze constrains op 2 manieren toevoegen

- Data annotaties (voor eenvoudige zaken)

- Fluent API (voor complexere zaken)

Data Annotations

- [Key]
  - Elke data klasse moet een key object hebben => Primary key
- [Column(“blog_id")]
  - Wanneer we willen dat een bepaalde property in de database een andere kolomnaam krijgt.
- [Column(TypeName = "varchar(200)")]
  - Wanneer we willen dat een bepaalde property een bepaald type krijgt in de databank
- [MaxLength(500)]
  - Een maximale lengte geven aan een string
- [Required]
  - Aangeven dat een bepaald veld in de databank niet null kan zijn.
- [NotMapped]
  - By Default zijn alle public properties een veld in de databank

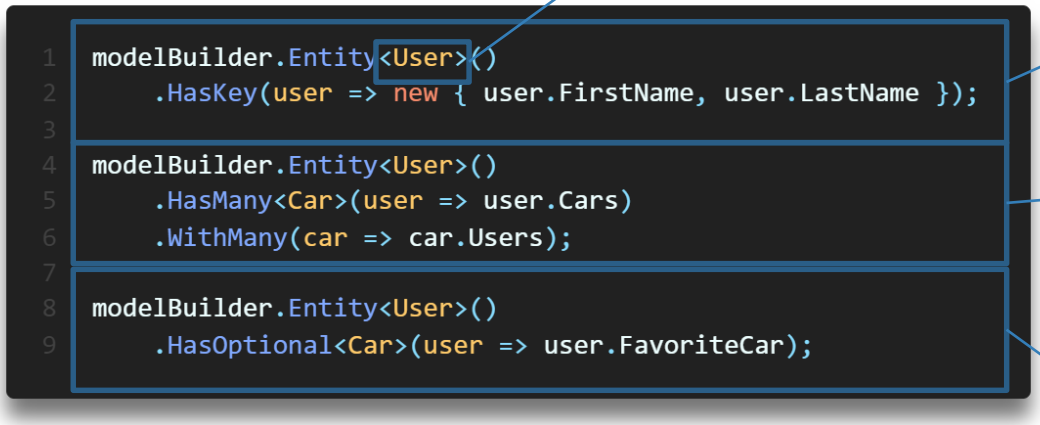
Fluent API

Wanneer we met data annotaties niet alles kunnen instellen wat we willen kunnen we gebruik maken van de Fluent API

- Dit kan door in de DbContext de OnModelCreating methode te overriden.

Dit is niet altijd nodig. Veel zaken gebeuren automatisch. Soms kan het nuttig zijn bepaalde zaken toch bewust in te stellen, zodat er voor EntityFramework geen twijfel is.

Fluent API - Entity



De tabel waarop we wijzigingen willen doen.

Instellen van een primary key bestaande uit 2 velden

Instellen van een veel op veel relatie

Instellen van een optionele waarde FavoriteCar

Fluent API - Properties

```
1  modelBuilder.Entity<Car>().Property(car => car.Model)
2      .HasMaxLength(50)
3      .IsOptional()
4      .HasColumnName("reeks");
```

Instellen van een maximale lengte

Instellen dat kolom model optioneel is

Instellen dat we de naam reeks gebruiken in de database

DbContext-Recap

- EntityFramework nodig
- Connection string
  - Welke server
  - Welke database
  - Welke credentials
- DbContext ≈ Database
- DbSet ≈ tabel
- DbSetItems ≈ rijen/records
- Properties ≈ kollomen

- Constraints
  - Data annotaties
  - Fluent API

Database aanspreken - Read

- Via de DbContext en de DbSet properties kunnen we databank aanspreken

```
1  UserContext userContext = new UserContext();
2  User[] users = userContext.users.ToArray();
```

- Deze kan je benaderen net zoals je een list of array zou gebruiken.

Database aanspreken - Create

- DbContext.<DbSet>.Add(...)
- Let er op dat alle niet nullable velden een waarde hebben
  - Indien niet wordt er een error opgegooid.
  - Vergeet niet op het einde steeds SaveChanges aan te roepen.

```
        UserContext userContext = new UserContext();
        User user = new User();
        user.FirstName = "John";
        user.LastName = "Doe";
        userContext.users.Add(new User());
        userContext.SaveChanges();
    }
```

Database aanspreken - Delete

- DbContext.<DbSet>.Remove(...)
- Let op cascading delete behavior
  - Error wordt opgegooid wanneer dit niet zou lukken
  - Deze kan je indien nodig instellen aan de hand van Fluent API
  - Vergeet niet op het einde steeds SaveChanges aan te roepen.

```
1  UserContext userContext = new UserContext();
2  User user = userContext.users.First(u => u.FirstName == "John");
3  userContext.users.Remove(user);
4  userContext.SaveChanges();
```

Database aanspreken - Update

- DbContext.SaveChanges()
- Let op SaveChanges werkt enkel voor data die aan de hand van die specifieke DbContext opgehaald is.

```
1    UserDbContext dbContext1 = new UserDbContext();
2    UserDbContext dbContext2 = new UserDbContext();
3
4    User user1 = dbContext1.Users.First(user => user.LastName == "Appels");
5    User user2 = dbContext2.Users.First(user => user.LastName == "Beer");
6
7    user1.FirstName = "Alex";
8    user2.FirstName = "Bert";
9
10   dbContext1.SaveChanges();
```

Database aanspreken – Update

- Wanneer je objecten veranderd die een oorsprong hebben van een andere DbContext. Kan je deze toch updaten door de state van het object aan te passen.

```
1  UserDbContext dbContext2 = new UserDbContext();
2
3  User user1 = dbContext1.Users.First(user => user.LastName == "Appels");
4  User user2 = dbContext2.Users.First(user => user.LastName == "Beer");
5
6  user1.FirstName = "Alex";
7  user2.FirstName = "Bert";
8
9  dbContext1.Entry(user2).State = EntityState.Modified;
10
11 dbContext1.SaveChanges();
```

Database aanspreken – Best practice

Het is geen goede/mooie programmeer gewoonte om overal waar je het nodig hebt je DBContext aan te spreken. We kunnen dit beter “verstoppen” in een speciale Helper Class die we een Repository of DataAccess zullen noemen.

In deze repository gaan we meestal de **CRUD** operaties toevoegen
Create, Read, Update, Delete

```
class UserRepository
{
    private UserContext context = new UserContext();

    // READ
    public List<User> GetAllUsers()
    {
        return context.users.ToList();
    }

    // CREATE
    public void CreateUser(User user)
    {
        context.users.Add(user);
        context.SaveChanges();
    }

    // DELETE
    public void DeleteUser(User user)
    {
        context.users.Remove(user);
        context.SaveChanges();
    }

    // UPDATE
    public void UpdateUser(User user)
    {
        context.SaveChanges();
    }
}
```

Migraties

Van zodra we de eerste keer onze DbContext aanspreken, wordt onze database automatisch aangemaakt.

Soms gaat het gebeuren dat we tijdens het ontwikkelen ons **datamodel** gaan **wijzigen**.
We gaan dus ook onze database moeten aanpassen. Dit kan echter niet zomaar.
Er zal nood zijn aan een **database migratie**! En dit gebeurt helaas **niet automatisch**.

Om dit te doen moeten we migraties "aanzetten" met andere woorden aangeven dat dit kan gebeuren.

Dit kunnen we doen door in ons project eenmalig in de Package Manager Console een commando uit te voeren: Enable-Migrations

Tools → Nuget Package Manager →

Lokale migraties

In Package Manager Console:

- **Add-Migration**
  Aangeven dat er een nieuwe migratie is.
- **Update-Database**
  Migratie uitvoeren

Dit werkt perfect voor een lokale database. Wanneer een database op locatie staat of wanneer elke eindgebruiker zijn persoonlijke database heeft, kunnen we deze commando's niet altijd gaan uitvoeren.

Migraties in code

Zorg wat er elke update verandert

Add-migration moet je nog steeds uitvoeren.

Dit commando genereerd een **migratie file**.

Migraties worden steeds gemaakt op basis van je database en voorgaande migraties.

```
public partial class add_birthdate : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.Users", "BirthDate", c => c.DateTime());
    }

    public override void Down()
    {
        DropColumn("dbo.Users", "BirthDate");
    }
}
```

Migratie in code uitvoeren. Hiervoor moeten we aan onze DbContext constructor een lijn toevoegen die aangeeft dat we steeds controleren voor nieuwe migraties.

```
public UserContext() : base("UserDatabase")
{
    Database.SetInitializer(new MigrateDatabaseToLatestVersion<UserContext,Configuration>());
}
```

Naam van de klasse met de configuratie

Automatische migraties

Er is ook een methode om migraties automatisch uit te voeren.

Om deze te gebruiken moet je bij het "aanzetten" van de migraties een parameter – EnableAutomaticMigrations toevoegen.

Enable-Migrations -EnableAutomaticMigrations

De code in de DbContext klasse om je migratie uit te voeren is ook nog steeds nodig (Zie vorige slide) → Database.SetInitializer ...

Concreet:

- Add-migration hoef je niet meer uit te voeren.
- **Niet altijd gewenst**. Geen controle wat er exact gedaan wordt.

Lambda's en databases

Om te query-en in de database kunnen we gebruik maken van lambda's

```
public List<User> GetUsersWithFirstName(string firstName)
{
    return context.users
        .Where(user => user.FirstName == firstName)
        .ToList();
}
```

Dankzij LINQ kunnen we ook query-a-like queries schrijven.

```
public List<User> GetUsersWithFirstName(string firstName)
{
    var users = from user in context.users
                where user.FirstName == firstName
                select user;
    return users.ToList();
}
```

Include

By default zijn referenties naar andere klasses niet aanwezig wanneer we iets opvragen aan de database. Bij een user met een lijst van cars kunnen we de cars niet rechtstreeks zien. Dit kan wel wanneer we deze bewust “Includen”.

```
class CarRepository
{
    UserContext context = new UserContext();

    public List<Car> getAllCars()
    {
        return context.Cars.ToList();
        // User object in car Object zal null zijn.
    }

    public List<Car> getAllCarsWithUser()
    {
        return context.Cars.Include(x => x.User).ToList();
        // User object in car Object zal NIET null zijn.
    }


    public void AddCar(Car car)
    {
        context.Cars.Add(car);
        context.SaveChanges();
    }

}
```

```
class Car
{
    private int id;
    private string model;
    private string brand;
    private Color color;
    private User user;

    public int Id { get => id; set => id = value; }
    public string Model { get => model; set => model = value; }
    public string Brand { get => brand; set => brand = value; }
    public Color Color { get => color; set => color = value; }
    public User User { get => user; set => user = value; }
}
```

!!! Om include te gebruiken met een lambda functie => using system.data.entity gebruiken

Meer info

Dit topic is zodanig breed dat we onmogelijk elke uitzondering in de les kunnen behandelen.
Voor meer info zie:

https://docs.microsoft.com/en-us/ef/ef6/
https://www.entityframeworktutorial.net/