

(ORM) Object relational mapping

Van code naar een database en terug, met het Entity Framework (EF) Core van Microsoft.

Jelle Krupe

17 september 2024

DE HAAGSE
HOGESCHOOL

Agenda

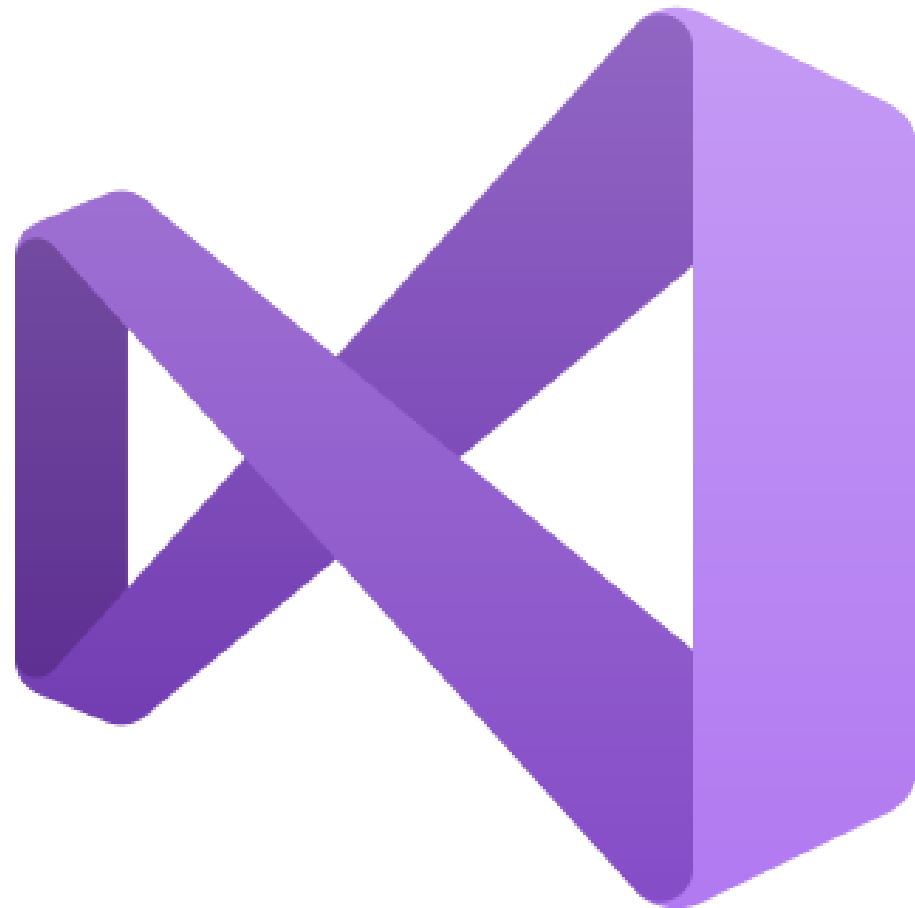


- Uitwerkingen vorige week bekijken
- Data persistentie
- XML & JSON
- Object relational mapper (ORM)
- Code
- Waar te beginnen
- Migraties

Announcements

- De toets in week 8 gaat over alles stof tot dat moment.
- In week 8 (de dag voor de toets) is er nog een Wrap-up van alle stof tot dat moment. Hier gaan we snel door de stof en beantwoorden we jullie vragen (hier komt vanzelf meer informatie over). Als het goed is weet je op dit moment alles al want de toets is de volgende dag, maar extra herhaling kan nooit kwaad.
- Aanstaaende donderdag hebben wij een aantal opdrachten voor jullie (in toetsvorm), zodat jullie jezelf kunnen toetsen in hoeverre je het op het nominale niveau van WPFW loopt.

Uitwerkingen HC2.2



Data persistentie

- Iets wegschrijven en later weer in dezelfde staat ophalen. (Database)
 - Opslaan
 - Afdwingen consistentie met bijvoorbeeld constraints
 - Ophalen
 - Efficient, schaalbaar
 - Query taal?
- Er zijn verschillende soorten vormen van databases:
 - Relationeel (Semester 3)
 - Document databases (Semester 4)
 - Graph databases (Semester 4)

Data persistentie

- Kunnen we zomaar alles opslaan in een database?
 - Eerst serializatie
 - Dit is de staat van een object veranderen, zodat het object persistent en/of transporteerbaar wordt
 - Bijvoorbeeld: XML of JSON

XML & JSON

- XML en JSON zijn formaten waarmee data vanaf een web server kan worden opgehaald
- Ieder heeft zijn voor en nadelen, over het algemeen is JSON toegankelijker voor het doorsnede gebruik
- Wij blijven focussen op JSON, maar weet dat je ook XML kan gebruiken

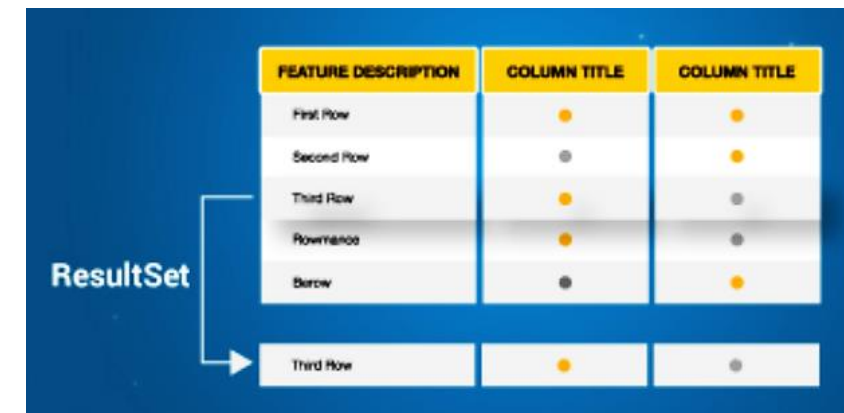
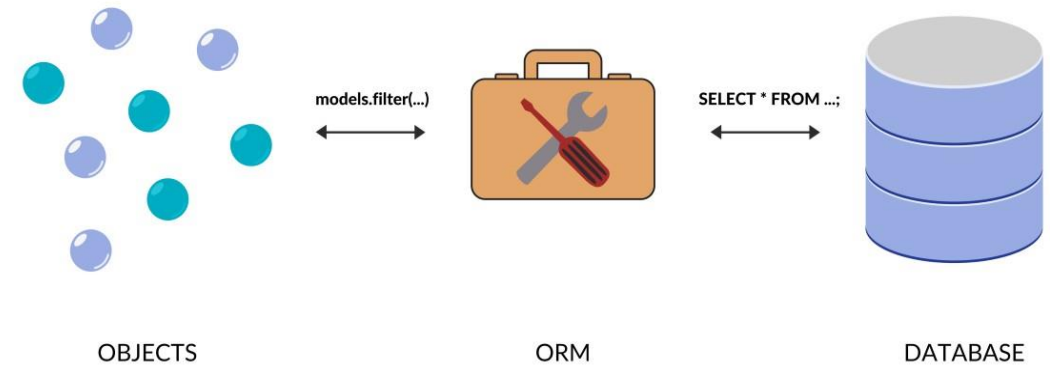
```
<studentsList>
  <student id="1">
    <firstName>Greg</firstName>
    <lastName>Dean</lastName>
    <certificate>True</certificate>
    <scores>
      <module1>70</module1>
      <module12>80</module12>
      <module3>90</module3>
    </scores>
  </student>
  <student ind="2">
    <firstName>Wirt</firstName>
    <lastName>Wood</lastName>
    <certificate>True</certificate>
    <scores>
      <module1>80</module1>
      <module12>80.2</module12>
      <module3>80</module3>
    </scores>
  </student>
</studentsList>
```

```
{
  "first_name": "John",
  "last_name": "Smith",
  "is_alive": true,
  "age": 27,
  "address": {
    "street_address": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postal_code": "10021-3100"
  },
  "phone_numbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [
    "Catherine",
    "Thomas",
    "Trevor"
  ],
  "spouse": null
}
```

Object Relational Mapper

Waar en hoe

- De ORM zit tussen de applicatie en de database
- De ORM vertaalt opdrachten van een applicatie naar opdrachten voor een database (SQL)
- Data uit de database door de ORM komen niet als resultset, maar je krijgt direct objecten in code.



GetPerson zonder ORM

```
public async Task GetPerson(int id)
{
    Person person = null;
    using (var connection = new SqlConnection(_connectionString))
    {
        var command = new SqlCommand("SELECT Id, Name, Age FROM Persons WHERE Id = @Id", connection);
        command.Parameters.AddWithValue("@Id", id);

        await connection.OpenAsync();
        using (var reader = await command.ExecuteReaderAsync())
        {
            if (await reader.ReadAsync())
            {
                person = new Person
                {
                    Id = reader.GetInt32(0),
                    Name = reader.GetString(1),
                    Age = reader.GetInt32(2)
                };
            }
        }
    }
}
```

GetPerson zonder ORM

Deze code is ter vergelijking met de volgende slide.
Je hoeft het nu nog niet te begrijpen.

```
public async Task GetPerson(int id)
{
    var person = await _context.Persons.FindAsync(id);

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

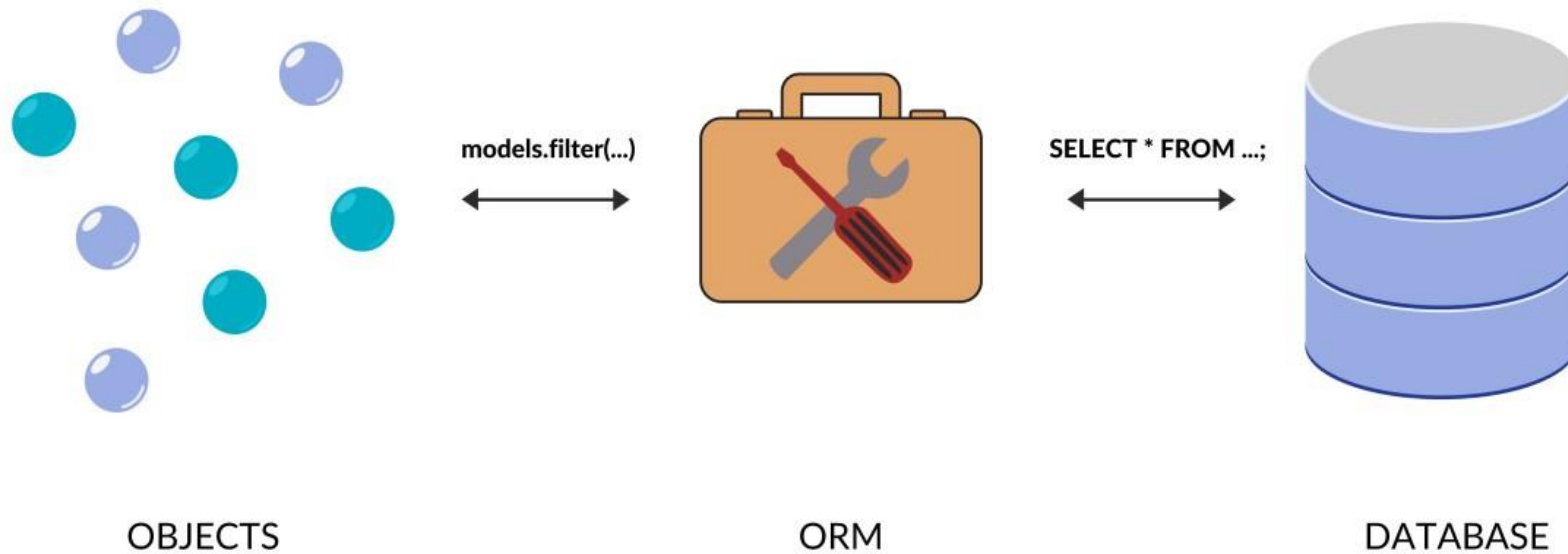
    return person;
}
```

Object Relational Mapper

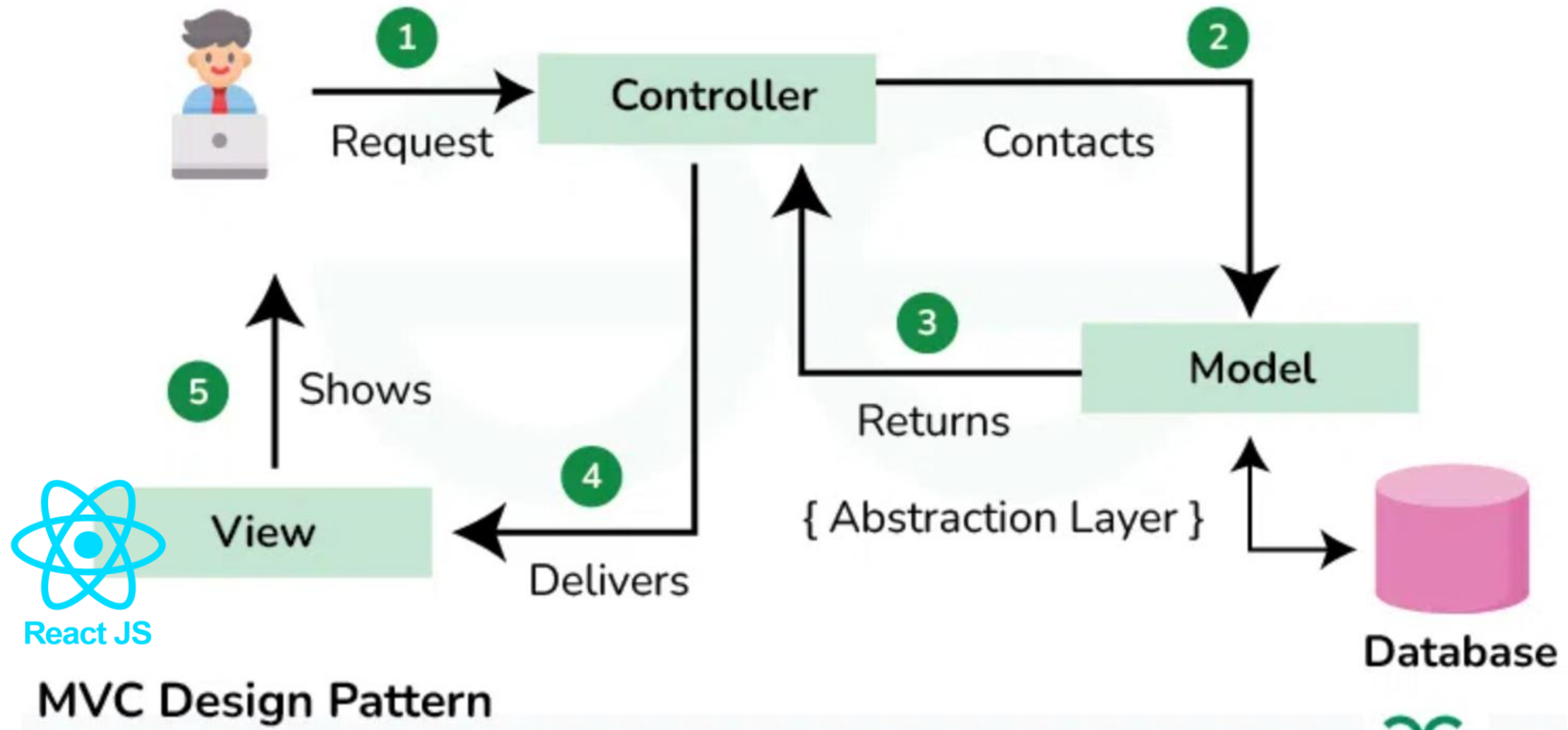
Wat zijn voor- en nadelen?

- Voordelen:
 - Code bevat geen SQL meer (Security)
 - Minder code nodig, want je hoeft niet meermaals queries te schrijven
 - Codequeries maakt de code leesbaarder en daardoor makkelijker te begrijpen
 - Onafhankelijk van de keuze voor het Database Management Systeem (DBMS). Er kan bijvoorbeeld gebruik gemaakt worden van: SQL Server, MySQL, PostgreSQL, etc.
- Nadeel:
 - Je zit gebonden aan kaders van het ORM er is dus een beperkte hoeveelheid customalisatie. (Hier zal je niet snel tegenaan lopen)

Begrijpt iedereen het concept tot dusver?



MVC pattern?



Model (Klasse)

- Een 'Model' is een klasse met alleen maar eigenschappen en gedrag. Bijvoorbeeld:

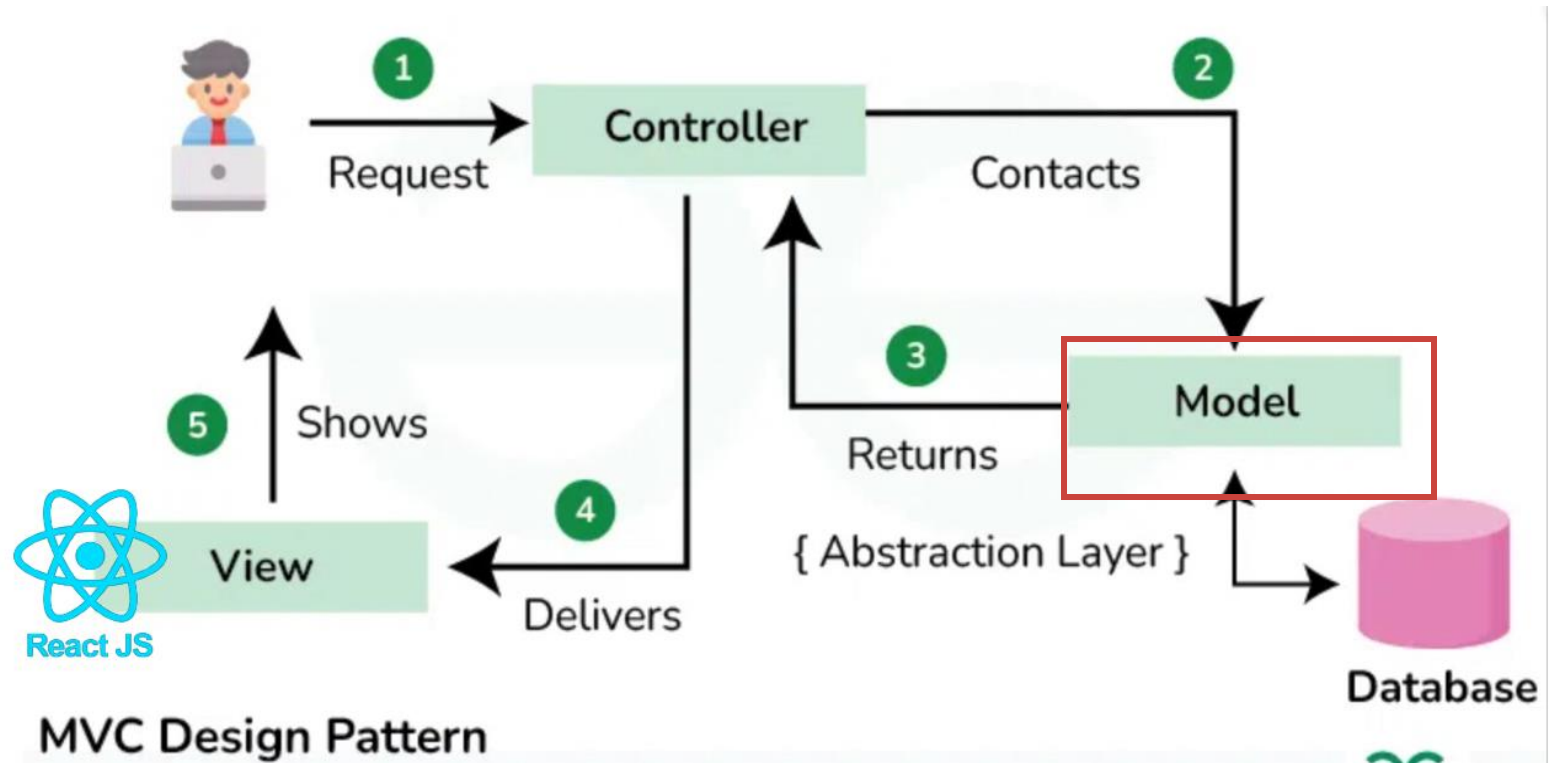
- **Fiets**

- **Eigenschappen**

- Stuur
 - Wielen
 - Frame

- **Gedrag**

- Versnellen
 - Remmen



ORM | Model naar database tabel I

Hoe wordt een model klasse vertaald naar de database

```
class Student
{
    public string Naam { get; set; }
    public int Leeftijd { get; set; }
    public bool PropedeuseGehaald { get; set; }
}
```

```
CREATE TABLE Student (
    Naam varchar(255),
    Leeftijd int,
    PropedeuseGehaald bit
);
```

- Klassen worden tabellen
- Objecten worden rijen
- Properties worden kolommen
- int wordt int
- String wordt varchar/...
- bool wordt bit

ORM | Model naar database tabel II

```
class Student
{
    public string Naam { get; set; }
    public int Leeftijd { get; set; }
    public bool? PropedeuseGehaald { get; set; }
}
```

```
CREATE TABLE Student (
    Naam varchar(255),
    Leeftijd int NOT NULL,
    PropedeuseGehaald bit
);
```

- Klassen worden tabellen
- Objecten worden rijen
- Properties worden kolommen
- int wordt int
- String wordt varchar/...
- bool wordt bit
- Nullable variabelen worden niet NOT NULL

ORM | Model naar database tabel III

```
class Student
{
    public int Id { get; set; }
    public string Naam { get; set; }
    public int Leeftijd { get; set; }
    public bool? PropedeuseGehaald { get; set; }
    public List<Resultaat> Resultaten { get; set; }
}
```

```
CREATE TABLE Student (
    StudentID int NOT NULL,
    Naam varchar(255),
    Leeftijd int NOT NULL,
    PropedeuseGehaald bit,
    PRIMARY KEY (StudentID)
);
CREATE TABLE Resultaat (
    ...
    StudentID int,
    FOREIGN KEY (StudentID)
    REFERENCES Student(StudentID)
    ...
);
```

- Klassen worden tabellen
- Objecten worden rijen
- Properties worden kolommen
- int wordt int
- String wordt varchar/...
- bool wordt bit
- Nullable variabelen worden niet NOT NULL
- Niet-primitieve instance variabelen krijgen foreign keys
- Id wordt de primary key

ORM | Model naar database tabel IV

```
class Student
{
    public string Naam { get; set; }
    public int Leeftijd { get; set; }
    public bool? PropedeuseGehaald { get; set; }
    public List<Resultaat> Resultaten { get; set; }
    public DoetToets(double resultaat)
    {
        // ...
    }
}
```

```
CREATE TABLE Student (
    StudentID int NOT NULL,
    Naam varchar(255),
    Leeftijd int NOT NULL,
    PropedeuseGehaald bit,
    PRIMARY KEY (StudentID)
);
CREATE TABLE Resultaat (
    ...
    StudentID int,
    FOREIGN KEY (StudentID)
    REFERENCES Student(StudentID)
    ...
);
```

- Klassen worden tabellen
- Objecten worden rijen
- Properties worden kolommen
- int wordt int
- String wordt varchar/...
- bool wordt bit
- Nullable variabelen worden niet NOT NULL
- Niet-primitieve instance variabelen worden foreign keys
- Id wordt de primary key
- Methoden worden gedrag
- Transacties word in code .SaveChanges()
- LINQ wordt SQL

Meer op meer & overerving

- Hoe maken we een meer op meer relatie?
- Zo simpel als een lists maken en dan weet de ORM dat het een meer op meer relatie is:

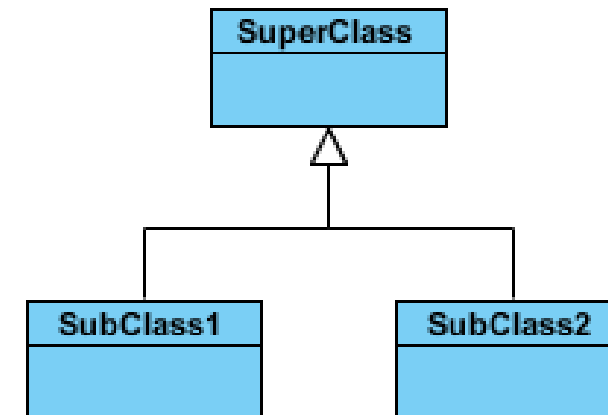
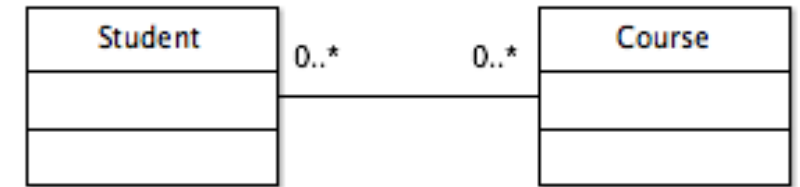
```
class Student { public List<Klas> Volgt { get; set; } }
```

```
class Klas { public List<Student> Studenten { get; set; } }
```

- Zo maken we een overerving:

```
class BrightspaceGebruiker {  
}
```

```
class Student : BrightspaceGebruiker  
{  
    public int Id { get; set; }  
    public string Naam { get; set; }  
    public List<Resultaat> Resultaten {get; set; }  
}
```



Controller (Klasse)

Waar maken we objecten aan?

Voor nu:

- Verantwoordelijk voor de logica:
 - Behandelen gebruikersinvoer
 - Aanroepen modellen
 - Het teruggeven van data uit bijvoorbeeld de database

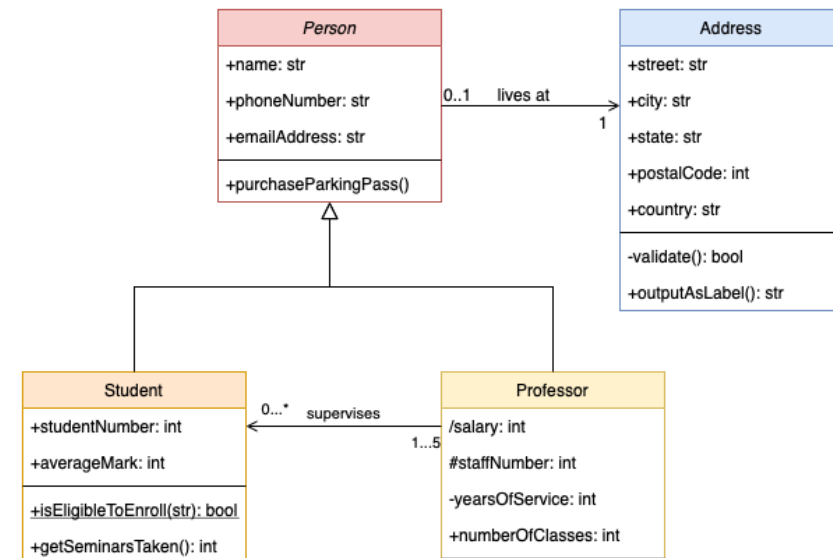
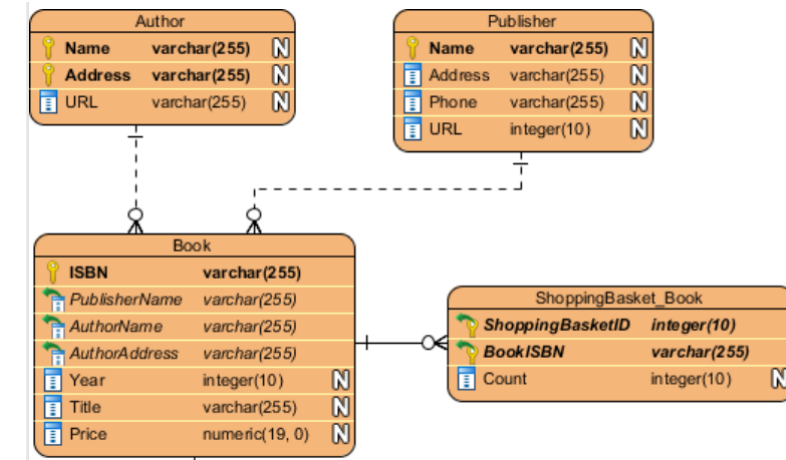
Hier komen we volgende week pas op terug

- Routebeheer
- Actiemethoden (CRUD)
- Gegevensvalidatie

Waar beginnen we?

Stel je voor we gaan een applicatie maken

- Tijdens analyse stellen we het probleemdomein op
- Tijdens ontwerp maken we
 - een datamodel van de database [DB Design Semester 3], en
 - een klassemodel van de code [UML Semester 2]



Code vs Database first

- Database-first of code-first?
- **Database First:** Het beste voor projecten met een bestaande database of wanneer je de voorkeur geeft aan het eerst ontwerpen van het databaseschema.
- **Code First:** Het beste voor nieuwe projecten of wanneer je de voorkeur geeft aan het definiëren van het databaseschema via code en gebruik wilt maken van migraties voor schema-updates.
- Onderhoudbaarheid/wijzigbaarheid: aanpassen databaseschema -> aanpassen code & queries aanpassen

Migrations

- Migration houdt de wijzigingen bij tussen de code en database.
- Als een migration is gegenereerd, dan kan hiermee de database worden geupdate
- Een migration maakt gebruik van Fluent API om de database te modelleren.
- We roepen een migration aan als er in de code iets wijzigt, wat aanpassen voor de database met zich meebrengt. Denk bijvoorbeeld aan wijzigingen aan een model.

```
modelBuilder.Entity("Project.Models.Film", b =>
{
    b.Property<long>("Id")
        .ValueGeneratedOnAdd()
        .HasColumnType("INTEGER");

    b.Property<string>("Titel")
        .HasColumnType("TEXT");

    b.Property<int>("Year")
        .HasColumnType("INTEGER");

    b.HasKey("Id");

    b.ToTable("Films");

    b.HasData(
        new
        {
            Id = 1L,
            Titel = "The Shawshank Redemption",
            Year = 1994
        },
    );
});
```

Migrations

Commands

- `dotnet ef migrations add [name migration]`
- [...] -> invullen met een naam naar keuze, de eerste migration heet vaak initial
- `dotnet ef database update`
- Om de wijzingen in de database door te voeren
- **Als je de database leeg wil hebben** -> verwijder alle database tabellen; verwijder alles in de map migrations; maak een nieuwe migration (initial); update de database

Context

ORM connectie met de DB

```
private readonly MyContext _context;  
_context.Studenten.Add(student);  
_context.SaveChanges();
```

- De context is het object waarmee we data in de database kunnen stoppen, maar ook data mee uit de database kunnen halen.

```
public class MyContext : DbContext  
{  
  
    protected override void OnConfiguring( DbContextOptionsBuilder b) => b.UseSqlite("Data Source=database.db");  
    // b.UseSqlServer("Connection string") //Je kan de keuze van de database ook in de startup van de app plaatsen  
  
    protected override void OnModelCreating( ModelBuilder modelBuilder){  
        base.OnModelCreating(modelBuilder);  
    }  
  
    public DbSet<Student> Studenten { get; set; } //DbSet toevoegen van jouw database tabellen  
  
    //Hier kunnen eventueel relaties worden toegevoegd als het EF die niet goed pakt uit de modellen  
}
```

Context

```
class DatabaseContext : DbContext {
```

Lokale
server

```
protected override void OnConfiguring(DbContextOptionsBuilder options)
=> options.UseSqlServer("server=.;
                        database=Ormdb;
                        user id=jelle;
                        password=█;
                        trustservercertificate=true");
```

Bron: [SQL Server connection strings](#)

Zie voor veel mogelijke
connection string formats

Voorbeeld met ORM

MyContext.cs

```
public class MyContext : DbContext
{
    protected override void OnConfiguring(
        DbContextOptionsBuilder b) =>
        b.UseSqlite("Data Source=database.db");
        // b.UseSqlServer(...)
    public DbSet<Student> Students { get; set; }
}
```

Grade.cs & Student.cs

```
public class Grade
{
    public int Id { get; set; }
    public int Value { get; set; }
}

public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Grade> Grades { get; set; }
}
```

Program.cs

```
public class Program
{
    public static void Main(String[] args) {
        MyContext c = ...;
        c.Students.Add(new Student() { Naam = "Bob" });
        c.SaveChanges();
        c.Students.Single((s) => s.Id == 123)
            .Grades.Add(new Grade ...);
    }
}
```

Console output na >| database update

```
CREATE TABLE "Studenten" (
    "Id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT, "Naam"
    TEXT NULL
)
CREATE TABLE "Grade" (
    "Id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT, "Value"
    INTEGER NOT NULL,
    "StudentId" INTEGER NULL,
    FOREIGN KEY ("StudentId") REFERENCES "Studenten"
    ("Id") ON DELETE RESTRICT
)
```

Voorbeeld met ORM **Let op!**

Public class. Niet Internal class

MyContext.cs

```
public class MyContext : DbContext
{
    protected override void OnConfiguring(
        DbContextOptionsBuilder b) =>
        b.UseSqlite("Data Source=database.db");
        // b.UseSqlServer(...)
    public DbSet<Student> Students { get; set; }
}
```

Grade.cs & Student.cs

```
public class Grade
{
    public int Id { get; set; }
    public int Value { get; set; }
}

public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Grade> Grades { get; set; }
}
```

Program.cs

```
public class Program
{
    public static void Main(String[] args) {
        MyContext c = ...;
        c.Students.Add(new Student() { Naam = "Bob" });
        c.SaveChanges();
        c.Students.Single((s) => s.Id == 123)
            .Grades.Add(new Grade ...);
    }
}
```

Dit pad moet verwijzen naar het .db bestand. EF core kan deze niet altijd vinden (Table not found exception). Voeg dan het hele pad toe, bijvoorbeeld:
C:\\Users\\..etc..\\ConsoleAppWpfw\\database.db"

EF Core Demo

- "Installeer" Entity Framework Core (NuGet package manager),
- `dotnet add package ...`
 - `Microsoft.EntityFrameworkCore.Design`
 - `Microsoft.EntityFrameworkCore.Sqlite`
 - `Microsoft.EntityFrameworkCore.Tools`
- Maak C# klasse aan
- Maak C# DbContext aan (connectie-instellingen)

```
public class MyContext : DbContext {  
    protected override void  
        OnConfiguring(DbContextOptionsBuilder b) =>  
        b.UseSqlite("Data Source=database.db");  
    public DbSet<Student> Studenten { get; set; }  
}
```

```
public class Grade {  
    public int Id { get; set; }  
    public int Value { get; set; }  
}
```

```
public class Student {  
    public int Id { get; set; }  
    public string Name { get; set; }  
    public List<Grade> Grades { get; set; }  
}
```

Entity Framework Core

Let op dat
je
migrations
incheckt
op git

Code First

```
dotnet add package  
Microsoft.EntityFrameworkCore  
//eenmalig  
dotnet tool install --global dotnet-ef  
//daarna: dotnet ef .....  
dotnet ef migrations add Eerste  
dotnet ef database update  
dotnet run
```

Het
aanmaken
van de
database

Showtime! Vanuit C# studenten aanmaken.

- ID's worden automatisch aangemaakt!
- Zodra je een C# model klasse aanpast, krijg je een foutmelding.
- Voeg een migration toe met een nieuwe naam
- Update de database
- Er zijn veel conventies:
 - Moet Id heten
 - Moeten properties gebruiken
 - ...

```
MyContext myContext = new MyContext();  
var Bob = new Student { Naam = "Bob" };  
Bob.Grades.Add(new Grade { Value = 5 });  
myContext.Add(Bob);  
  
// verwijderen en aanpassen gaat even makkelijk  
myContext.SaveChanges();  
Console.WriteLine(  
    myContext.Studenten.Where(s =>  
        s.Naam.StartsWith("B")).Count(  
    ));
```

Meer praktisch

- “*SQLite browser*” kan handig zijn
- Je kunt ook de queries loggen
 - `protected override void OnConfiguring(DbContextOptionsBuilder b) => b.LogTo(Console.WriteLine);`

Data annotaties

- Waar valideren? In het programma vs. in de database?
- De volgende annotaties moeten jullie kennen:

- [Key]
- [Required]
- [StringLength]
- [Table]
- [Column]
- [NotMapped]


- Voorbeeld:

```
[Table("Students")]
public class Student
{
    [Key]
    public int StudentNr { get; set; }

    [Required]
    [StringLength(20)]
    public string Naam { get; set; }

    public List<Grade> Grades { get; set; } = new List<Grade>();
}
```

Lijst instantiëren,
anders null reference exception



Fluent API (Optioneel)

```
public class MyContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) =>
        optionsBuilder.UseSqlite(@"Data Source=database.db");

    public DbSet<Student> Studenten { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.OnModelCreating(modelBuilder);
        modelBuilder.Entity<Student>()
            .Property(s => s.Naam)
            .IsRequired()
            .HasMaxLength(20);
        modelBuilder.Entity<Student>()
            .HasOne(s => s.Portfolio)
            .WithOne(p => p.Student)
            .HasForeignKey<Portfolio>(p => p.StudentId);
    }
}
```

- Met de Fluent API kan je meer dan data annotaties 👍, maar ingewikkelder en gescheiden van de code 🙄.

Database seeden

- Twee manieren:
 - In de OnModelCreating (zie migrations):
 - `modelBuilder.Entity<Student>().HasData(new Student { StudentId = 1, Naam = "Bob" });`
- Handmatig, twee manieren:
 - Als er geen data is, seed de database, en SaveChanges
 - Verwijder bestaande data, seed de database, en SaveChanges

Opdracht?

- Zie brightspace:



 [IC 3.1 Opdracht ORM - 24/25 Webprogramming, frameworks & usability FALL_SEM1 \(hhs.nl\)](#)



That's all Folks!