Entity Relations

Customizing Entity Models

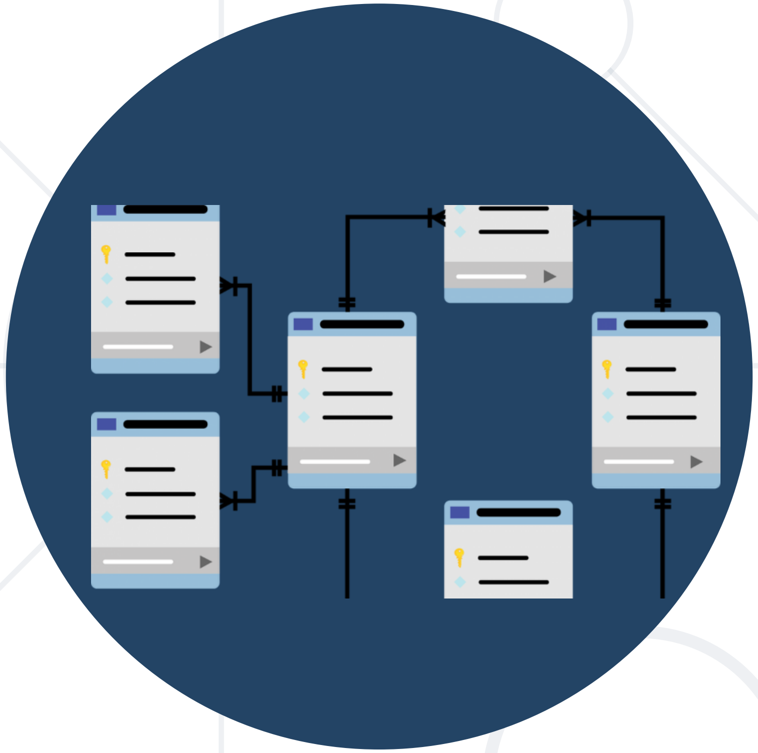
SoftUni Team

Technical Trainers

Software University

https://about.softuni.bg/

Table of Contents

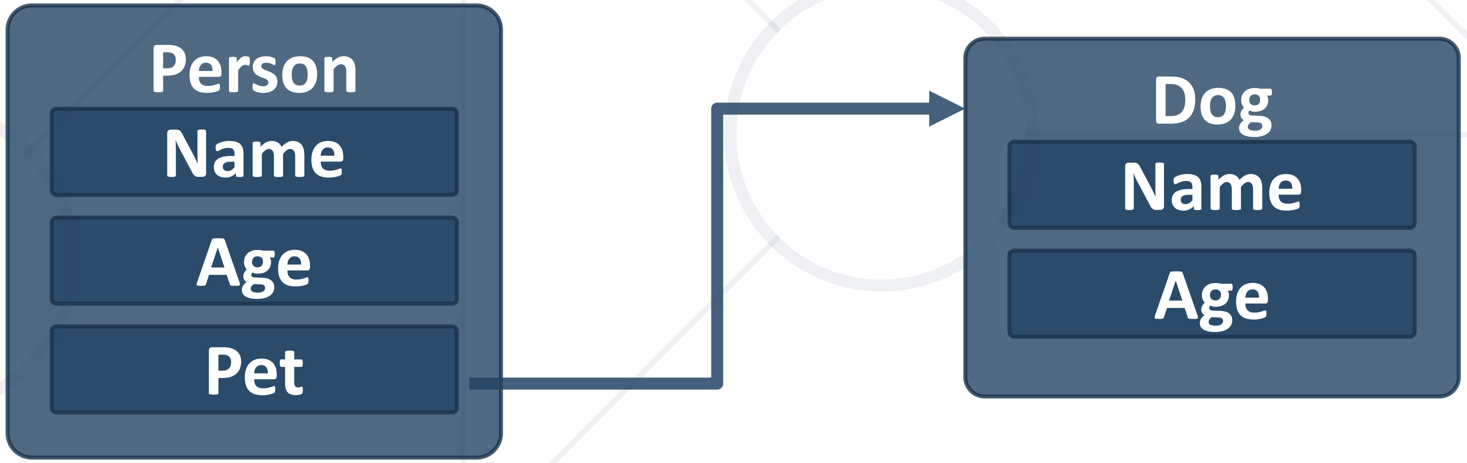


Object Composition

Describing Database Relationships

Object Composition

- Object composition denotes a "**has-a**" relationship
  - E.g. the **car** has an **engine**
- Defined in C# by one object having a property that is a reference to another



Navigation Properties

- Navigation properties create a **relationship** between entities
- Is either an **Entity Reference** (one to one or zero) or an **ICollection** (one to many or many to many)
- They provide **fast querying** of related records
- Can be **modified** by **directly** setting the reference

Entity Reference – One to One or Zero

```
public class Student {
  public int StudentId {get; set;}
  public string StudentName {get; set;}
  public virtual StudentAddress Address {get; set;}
}
```

Navigation property

```
public class StudentAddress {
  public int StudentAddressId {get; set;}
  public string Address {get; set;}
  public int Zipcode {get; set;}
  public virtual Student Student {get; set;}
}
```

Navigation property

f() => API

Fluent API

Working with Model Builder

Fluent API

- **Code First** maps your POCO(Plain Old CLR Objects) classes to tables using a **set of conventions**
  - E.g., property named "**Id**" maps to the **Primary Key**
- Can be customized using **annotations** and the **Fluent API**
  - Fluent API is another way to **configure** your **domain classes**
  - The **Code First Fluent API** is most commonly accessed by overriding the **OnModelCreating**

Working with Fluent API

- **Fluent API (Model Builder)** allows **full control** over DB mappings
  - Custom names of objects (columns, tables, etc.) in the DB
  - Validation and data types
  - Define complicated entity relationships
- Custom mappings are placed inside the **OnModelCreating** method of the DB context class

```
protected override void OnModelCreating(DbModelBuilder builder)
{
  builder.Entity<Student>().HasKey(s => s.StudentKey);
}
```

Fluent API: Renaming DB Objects

Software University

- Specifying Custom Table name

```
modelBuilder.Entity<Order>()
      .ToTable("OrderRef", "Admin");
```

Optional schema name

- Custom Column name/DB Type

```
modelBuilder.Entity<Student>()
   .Property(s => s.Name)
   .HasColumnName("StudentName")
   .HasColumnType("varchar");
```

Fluent API: Column Attributes

- Explicitly set Primary Key

```
modelBuilder
     .Entity<Student>().HasKey("StudentKey");
```

- Other column attributes

```
modelBuilder.Entity<Person>()
  .Property(p => p.FirstName)
  .IsRequired()
  .HasMaxLength(50)
```

```
modelBuilder.Entity<Post>()
  .Property(p => p.LastUpdated)
  .ValueGeneratedOnAddOrUpdate()
```

Fluent API: Miscellaneous Config

- Do not include property in DB (e.g. business logic properties)

```
modelBuilder
      .Entity<Department>().Ignore(d => d.Budget);
```

- Disabling cascade delete
  - If a FK property is non-nullable, cascade delete is **on by default**

```
modelBuilder.Entity<Course>()
   .HasRequired(t => t.Department)
   .WithMany(t => t.Courses)
   .HasForeignKey(d => d.DepartmentID)
   .OnDelete(DeleteBehavior.Restrict);
```

Throws exception on delete

Specialized Configuration Classes

- Mappings can be placed in entity-specific classes

```
public class StudentConfiguration
  : IEntityTypeConfiguration<Student>
{
  public void Configure(EntityTypeBuilder<Student> builder)
  {
     builder.HasKey(c => c.StudentKey);
  }
}
```

Specify target model

- Include in **OnModelCreating**:

```
builder.ApplyConfiguration(new StudentConfiguration());
```

Attributes

Custom Entity Framework Behavior

Attributes

- EF Code First provides a set of **DataAnnotation attributes**
  - You can override default Entity Framework behavior
- To access nullability and size of fields:

```
using System.ComponentModel.DataAnnotations;
```

- To access schema customizations:

```
using System.ComponentModel.DataAnnotations.Schema;
```

- For a full set of configuration options you need the **Fluent API**

Key Attributes (1)

- [**Key**] – explicitly specify **primary key**
  - When your PK column doesn't have an "Id" suffix

```
[Key]
public int StudentKey { get; set; }
```

  - **Composite key** is only defined using **Fluent API** for now

```
builder.Entity<EmployeesProjects>()
     .HasKey(k => new { k.EmployeeId, k.ProjectId });
```

Key Attributes (2)

- **ForeignKey** – explicitly **link** navigation property and foreign key property within the same class
- Works in **either direction** (FK to navigation property or navigation property to FK)

```
public class Client
{
  …
  [ForeignKey("Order")]
  public int OrderRefId { get; set; }
  public Order Order { get; set; }
}
```

Table name

Renaming Objects (1)

- **Table** – manually specify the name of the table in the DB

```
[Table("StudentMaster")]
public class Student
{
   …
}
```

```
[Table("StudentMaster", Schema = "Admin")]
public class Student
{
   …
}
```

Renaming Objects (2)

- **Column** – manually specify the name of the column in the DB
  - You can also specify order and explicit data type

```
public class Student
{
  …
  [Column("StudentName", Order = 2, TypeName="varchar(50)")]
  public string Name { get; set; }
}
```

Optional parameters

Entity Validation

- **Required** – mark a nullable property as **NOT NULL** in the DB
  - Will throw an exception if not set to a value
  - Non-nullable types (e.g. **int**) will **not throw** an exception (will be set to language-specific default value)
- **MinLength** – specifies min length of a string (client validation)
- **MaxLength** / **StringLength** – specifies max length of a string (both client and DB validation)
- **Range** – set lower and/or upper limits of numeric property (client validation)

Other Attributes

- **Index** – create index for column(s)
  - Primary key will always have an index

```
[Index(nameof(Url))]
public class Student
{
   public string Url { get; set; }
}
```

- **NotMapped** – property will not be mapped to a column
  - For business logic properties

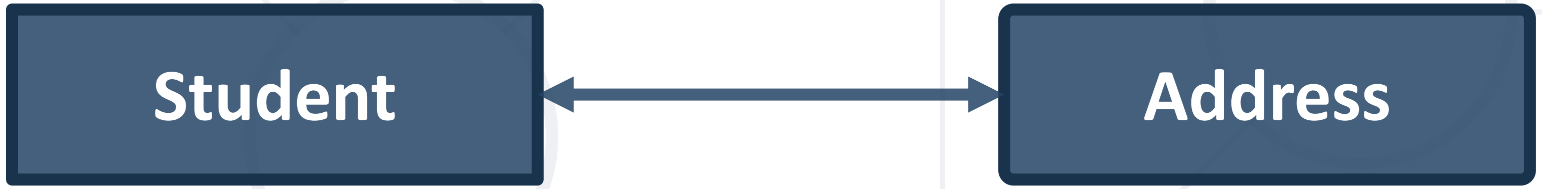
```
[NotMapped]
public string FullName => this.FirstName + this.LastName
```

Table Relationships

Expressed As Properties and Attributes

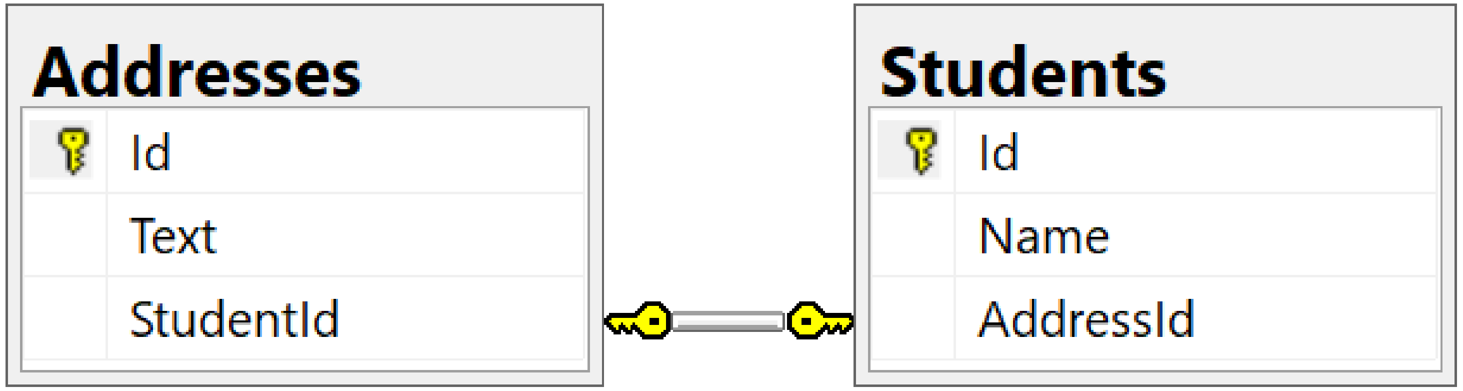
One-to-Zero-or-One

- Expressed in SQL Server as a shared primary key
- Relationship direction must be explicitly specified with a **ForeignKey** attribute
- ForeignKey is placed above the key property and contains the name of the navigation property and vice versa



Problem: One-to-Zero-or-One

- Create **database** with two tables: **Students** and **Addresses**
- The relationship of these tables should be **one to one**
- Use **Attributes** wherever you can



One-to-Zero-or-One: Implementation (1)

- Using the ForeignKey Attribute

```
public class Student
{
  [Key]
  public int Id { get; set; }
  public string Name { get; set; }
  [ForeignKey("Address")]
  public int AddressId { get; set; }
  public Address Address { get; set; }
}
```

Attributes

Attributes

One-to-Zero-or-One: Implementation (2)

- Using the ForeignKey Attribute

```
public class Address
{
  public int Id { get; set; }
  public string Text { get; set; }
  [ForeignKey(nameof(Student))]
  public int StudentId { get; set; }
  public Student Student { get; set; }
}
```

One-to-Zero-or-One: Fluent API

- **HasOne → WithOne**

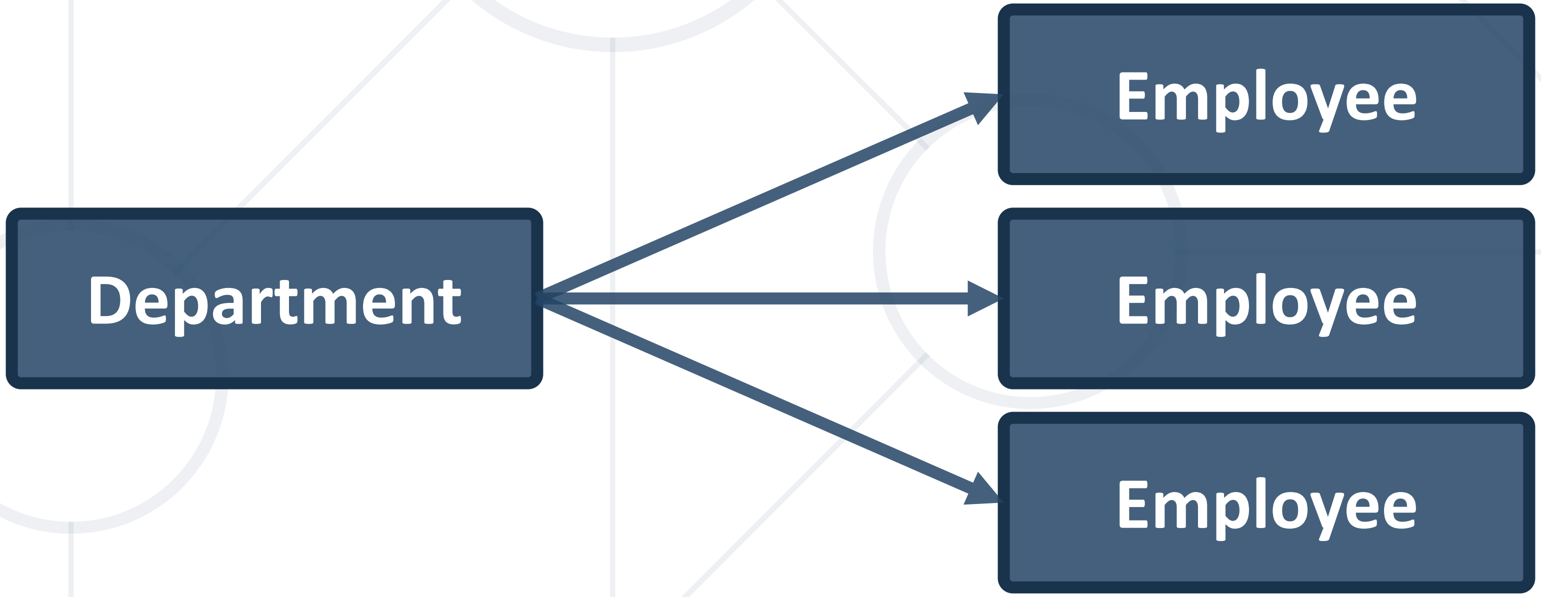
```
modelBuilder.Entity<Address>()
    .HasOne(a => a.Student)
    .WithOne(s => s.Address)
    .HasForeignKey<Address>(a => a.StudentId);
```

Address contains FK to Student

- If **StudentId** property is **nullable** (**int?**), relation becomes **One-To-Zero-Or-One**

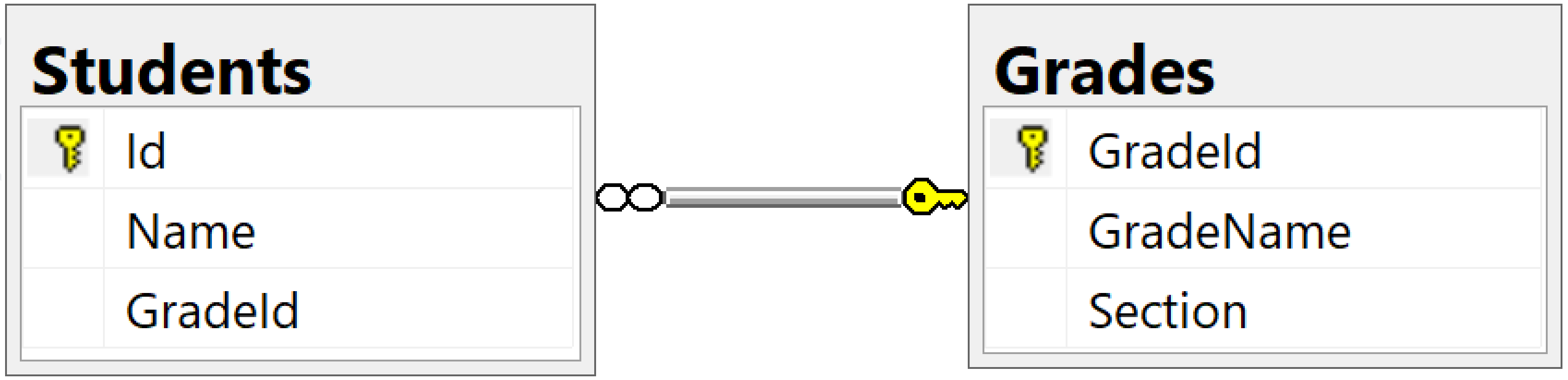
One-to-Many

- Most common type of relationship
- Implemented with a **collection** inside the **parent entity**
  - The collection should be **initialized** in the **constructor**!



Problem: One-to-Many

- Create **database** with two tables: **Students** and **Grades**
- The relationship of these tables should be **one to many**



One-to-Many: Implementation (1)

- **Grade** has **many students**

```
public class Grade
{
  public int GradeId { get; set; }
  public string GradeName { get; set; }
  public string Section { get; set; }

  public ICollection<Student> Students { get; set; }
}
```

One-to-Many: Implementation (2)

- **Student** have one **Grade**

```
public class Student
{
  public int Id { get; set; }
  public string Name { get; set; }
  public int GradeId { get; set; }
  public Grade Grade { get; set; }
}
```

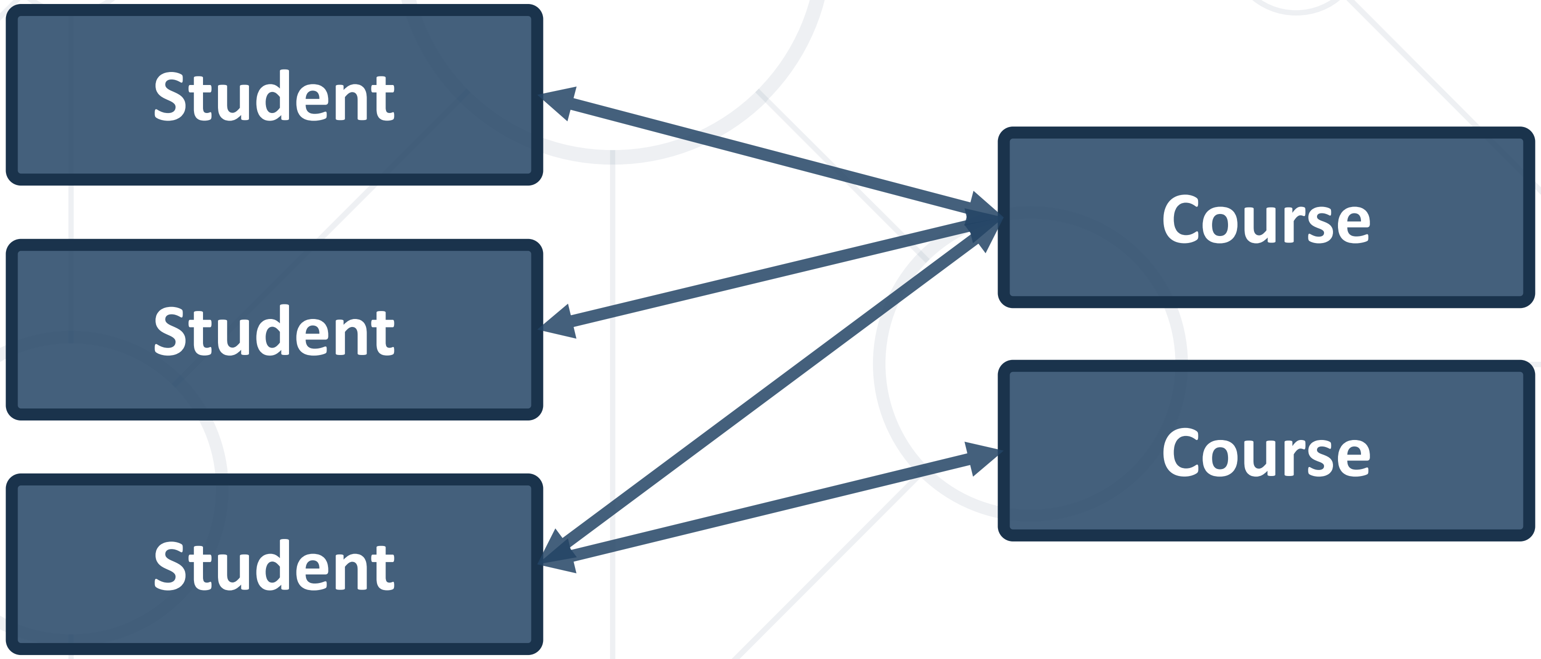
One-to-Many: Fluent API

- **HasMany → WithOne**

```
modelBuilder.Entity<Student>()
   .HasOne<Grade>(s => s.Grade)
   .WithMany(g => g.Students)
   .HasForeignKey(s => s.GradeId);
```

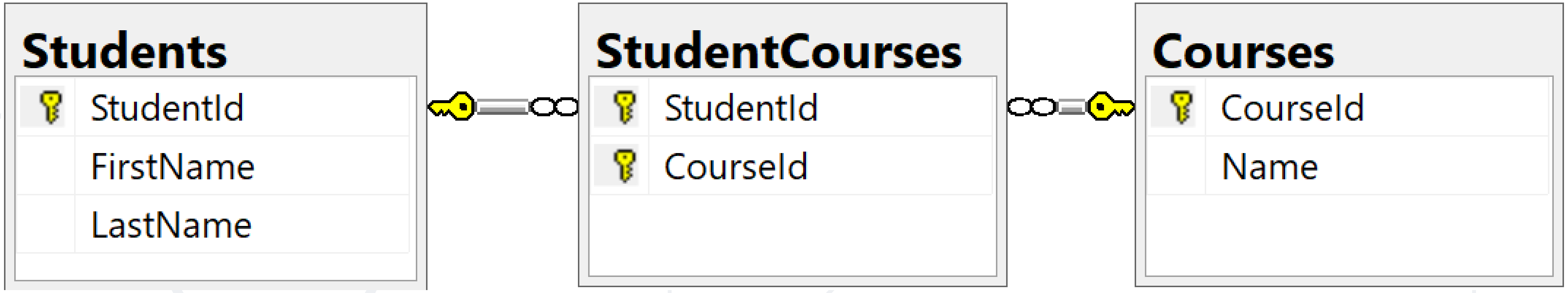
Many-to-Many

- Requires a **join entity (separate class)** in EF Core
- Implemented with collections in each entity, referring the other



Problem: Many-to-Many

- Create **database** with two three: **Students**, **StudentsCourses** and **Courses**
- The relationship of these tables should be **many to many**



Many-to-Many Implementation (1)

```
public class Course
{
  public int CourseId { get; set; }
  public string Name { get; set; }

  public ICollection<StudentCourse> StudentsCourses { get; set; }
}
```

```
public class Student
{
  public int StudentId { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }

  public ICollection<StudentCourse> StudentsCourses { get; set; }
}
```

Many-to-Many Implementation (2)

- EF Core requires a **Join Entity**

```
public class StudentCourse
{
  public int StudentId { get; set; }
  public Student Student { get; set; }

  public int CourseId { get; set; }
  public Course Course { get; set; }
}
```

Many-to-Many: Fluent API

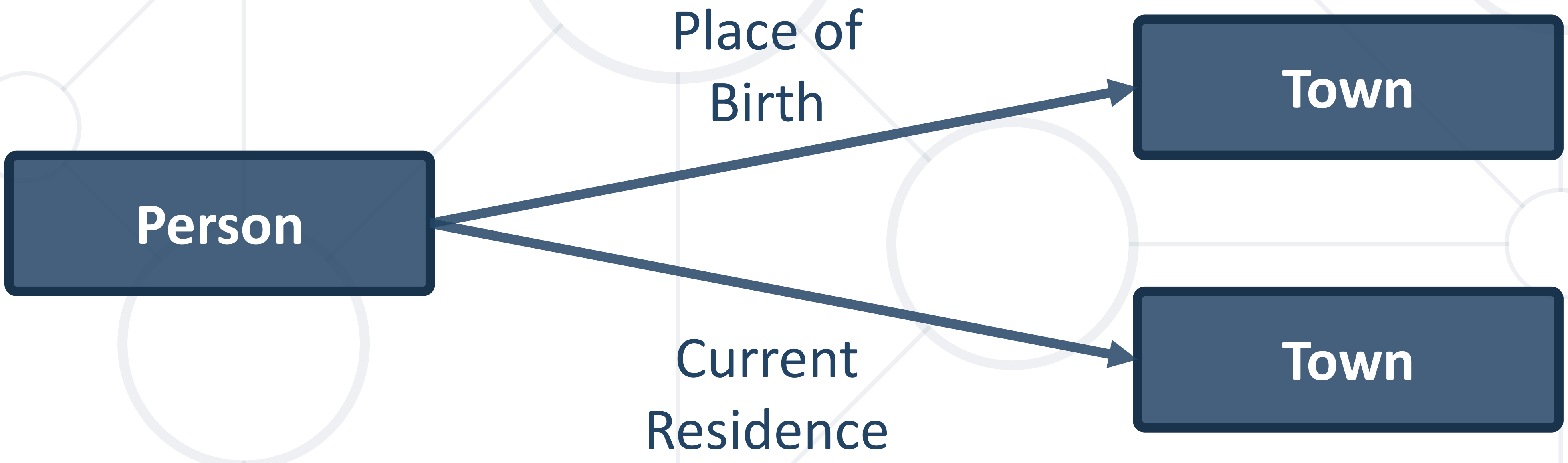
- Mapping **both sides** of relationship

```
modelBuilder.Entity<StudentCourse>()
   .HasKey(sc => new { sc.StudentId, sc.CourseId });

builder.Entity<StudentCourse>()
   .HasOne(sc => sc.Student)
   .WithMany(s => s.StudentCourses)
   .HasForeignKey(sc => sc.StudentId);

builder.Entity<StudentCourse>()
   .HasOne(sc => sc.Course)
   .WithMany(s => s.StudentCourses)
   .HasForeignKey(sc => sc.CourseId);
```

Composite Primary Key

Multiple Relations

- When two entities are related by more than one key
- Entity Framework needs help from **Inverse Properties**



Multiple Relations Implementation (1)

- Person Domain Model – defined as usual

```
public class Person
{
  public int Id { get; set; }
  public string Name { get; set; }

  public Town PlaceOfBirth { get; set; }
  public Town CurrentResidence { get; set; }
}
```

Multiple Relations Implementation (2)

- **Town** Domain Model

```
public class Town
{
  public int Id { get; set; }
  public string Name { get; set; }
  [InverseProperty("PlaceOfBirth")]
  public ICollection<Person> Natives { get; set; }
  [InverseProperty("CurrentResidence")]
  public ICollection<Person> Residents { get; set; }
}
```

Point towards related property

Summary

- The **Fluent API** gives us full control over Entity Framework object mappings
- **Attributes** can be used to express special table relationships and to customize entity behaviour
- Objects can be composed from other objects to represent complex **relationships**

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

License