Entity Framework CORE

Dipl.-Ing. Msc. Paul Panhofer Bsc.

Entity Framework Core Jump start

Entity Framework Core Jump start

Object Relational Mapping

Relational impedance mismatch

Objektorientierte Programmiersprachen kapseln Daten in Objekten. Relationale Datenbanken basieren dagegen auf dem mathematischen Konzept der relationalen Algebra.

Dieser konzeptionelle Widerspruch ist in der Programmentwicklung als **Relational impedance mismatch** bekannt.

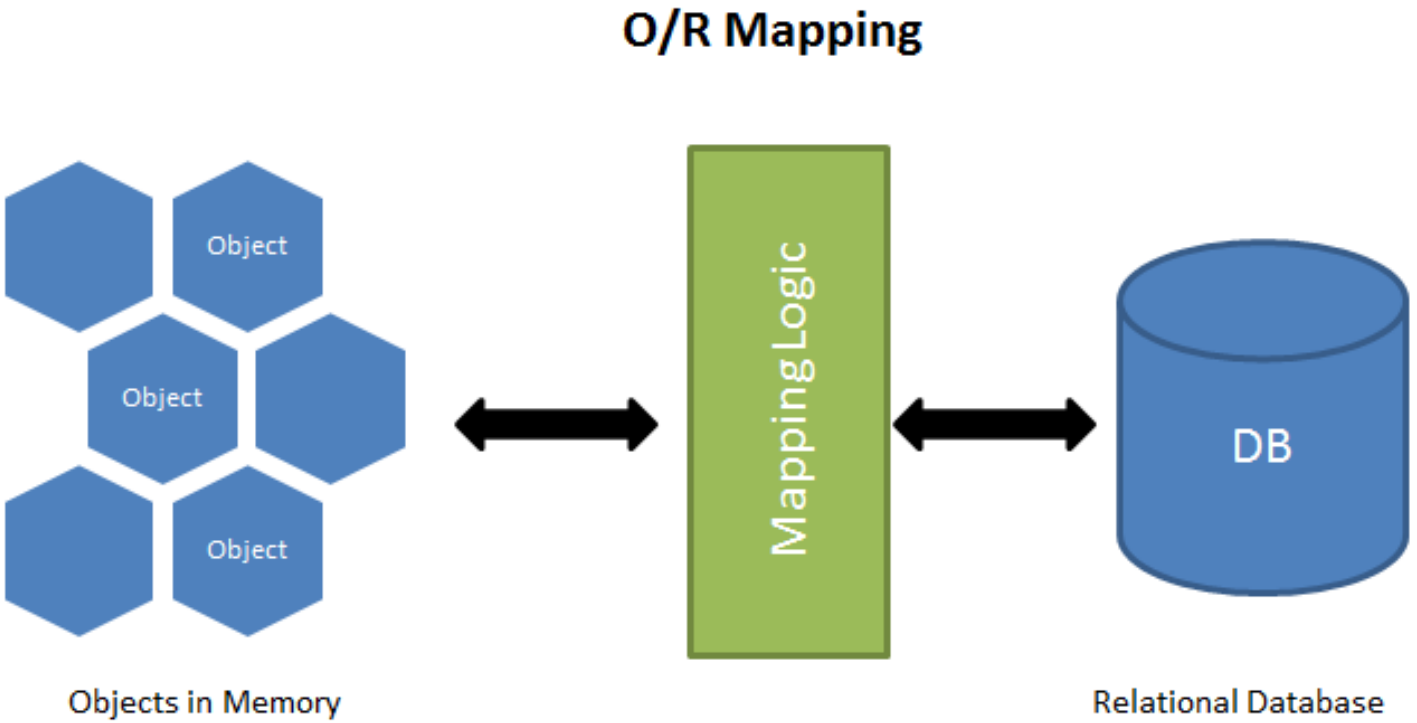
Object Relational Mapping

Object Relational Mapping ist eine Programmtechnik zur **Konvertierung** von Daten zwischen relationalen Datenbanken und OO Programmiersprachen.

EF CORE ist eine Implementierung von ORM für `.net`.

Entity Framework Core Jump start

Object Relational Mapping



Object Relational Mapping

Grundlegende Techniken

ORM bildet Klassen auf Tabellen ab. Ein Objekt entspricht dabei einer **Tabellenezeile**. Objektattributwerte werden in Tabellenspalten verwaltet.

Die Identität eines Objekt wird durch den Primärschlüssel der Tabelle bestimmt.

Object Relational Mapping

Grundlegende Techniken

In Tabellen gespeicherte Datensätze und Fremdschlüssel werden beim Lesen automatisiert in Objekte und Referenzen **umgewandelt**.

Beim Schreiben in die Datenbank findet eine Konvertierung in umgekehrter Richtung statt.

Entity Framework Core Jump start

Entität

Als Entität wird in der Datenmodellierung ein Objekt bezeichnet, das in einer Datenbank gespeichert werden kann.

Um Objekte einer Klasse als Entitäten designieren zu können, müssen sie mit dem **Datenbankkontext** einer Anwendung registriert werden.

Entität

Beispiel: Defaultimplementierung

```
public class Dish {

    public int Id { get; set; }

    public string Name { get; set; }

    public string Description { get; set; }

    public float Price { get; set; }

}
```

Entität

Beispiel: Defaultimplementierung

```
CREATE TABLE DISH (
   ID              INT      NOT NULL,
   NAME            VARCHAR,
   DESCRIPTION     VARCHAR,
   PRICE           DECIMAL,
   PRIMARY KEY (ID)
);
```

Entität

Annotationen

Durch die Verwendung von Annotationen kann die **Struktur** der einer Entität zugeordneten Tabelle adaptiert werden.

Annotationen erlauben die Einbindung von Metadaten in den Quelltext eines Programms.

Entität

Annotation: Table

Mit der Table Annotation wird der Name der Tabelle einer Entität bestimmt.

```
[Table("DISHES")]
[Comment("Dish Entity")] // optional
public class Dish {

    public int Id { get; set; }

    ...
}
```

Entität

Annotation: Required

Für die mit der Required Annotation ausgezeichneten Attribute, werden in der Datenbank `not null` **Constraints** generiert.

Entität

Annotation: Table

```
[Table("DISHES")]
public class Dish {
    public int Id { get; set; }
    [Required]
    public string Name { get; set; }
    [Required]
    public string Description { get; set; }
    [Required]
    public float Price { get; set; }
}
```

Entität

Annotation: NotMapped

Die mit der NotMapped Annotation ausgezeichneten Attribute, werden nicht in der Datenbank abgebildet.

```
[Table("DISHES")]
public class Dish {
    public int Id { get; set; }
    ...
    [NotMapped]
    public DateTime LoadedFromDatabase { get; set; }
}
```

Entität

Annotation: Column

Mit der Column Annotation kann für ein Attribut der **Datenbanktyp** und der **Spaltenname** bestimmt werden.

Hinweis: Defaultmäßig wird der Name des Attributs als Spaltenname gewählt.

Entität

Annotation: Column

```
[Table("DISHES")]
public class Dish {
    [Column("DISH_ID")]
    public int Id { get; set; }
    [Required][Column("NAME", TypeName="VARCHAR(100)")]
    public string Name { get; set; }
    [Column("DESCRIPTION", TypeName="VARCHAR(255)")]
    public string Description { get; set; }
    [Required][Column("PRICE", TypeName="DECIMAL(5,2)")]
    public float Price { get; set; }
}
```

Entität

Annotation: Key

Mit der Key Annotation wird ein Attribut als **Schlüssel** designiert.

Hinweis: Eine Property mit der Id Bezeichnung wird defaultmäßig als Schlüssel ausgezeichnet.

Entität

Annotation: Key

```
[Table("DISHES")]
public class Dish {

    [Column("DISH_ID")]
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    ...

}
```

Entität

```
[Table("DISHES")]
public class Dish {

    [Column("DISH_ID")]
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    [Required, StringLength(100)][Column("NAME")]
    public string Name { get; set; }

    [Required][Column("PRICE", TypeName="DECIMAL(5,2)")]
    public float Price { get; set; }

}
```

Entität

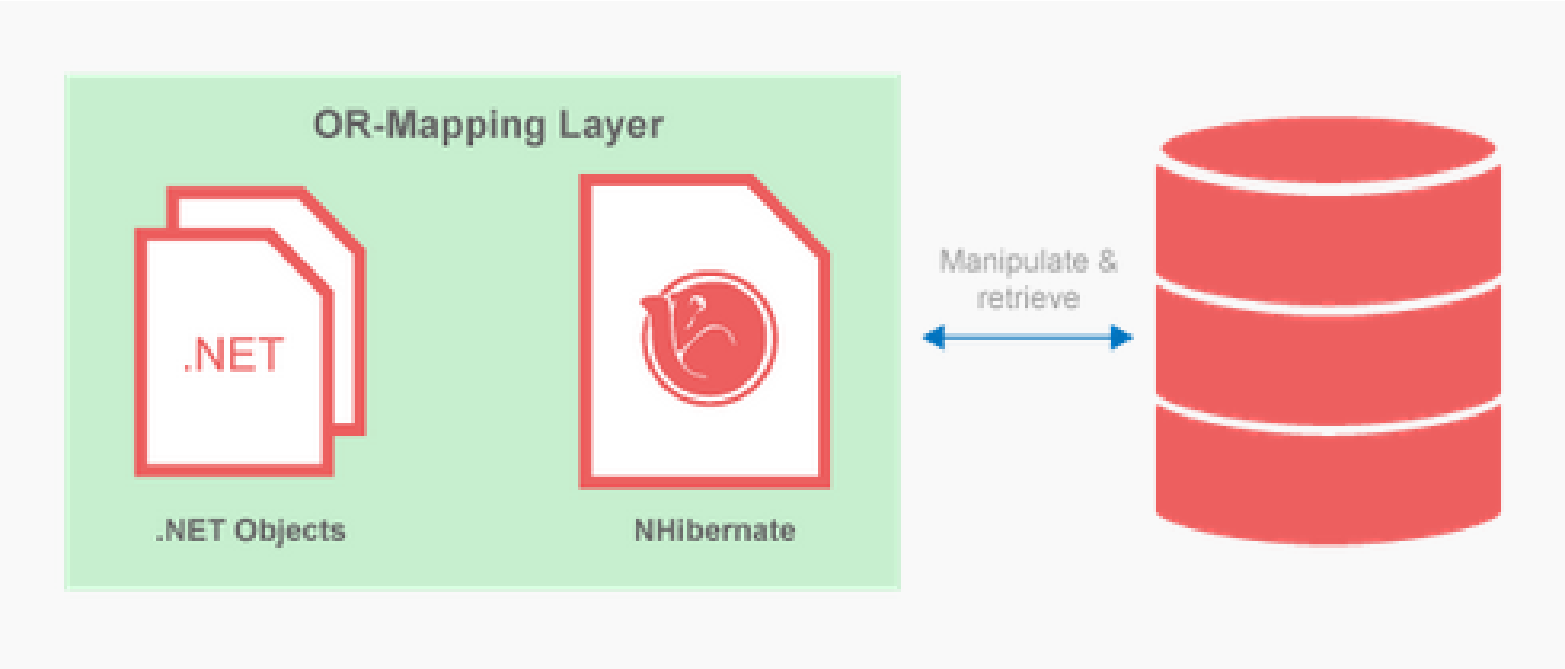
```
CREATE TABLE DISH (
   DISH_ID  INT           NOT NULL AUTO INCREMENT,
   NAME     VARCHAR(100)  NOT NULL,
   PRICE    DECIMAL(5,2)  NOT NULL,
   PRIMARY KEY (DISH_ID)
);
```

Entity Framework Core Jump start

DBConext

Der DBContext ist die zentrale Schnittstelle einer `.new Core` Anwendung zur Datenbank. Die DBContext Schnittstelle wird verwendet um den Relational impedance mismatch aufzulösen.

DBContext


OR-Mapping Layer
.NET
.NET Objects
NHibernate
Manipulate &
retrieve

DBContext

Registrieren von Entitäten

Klassen müssen innerhalb des DBContext **registriert** werden, um als Entität erkannt zu werden.

```
public class CookbookContext : DbContext {
   // Registrieren einer Klasse als Entitaet
   public DbSet<Dish> Dishes { get; set; }
   public
       CookbookContext(DbContextOptions<CookbookContext>
       options) : base(options) {...}
}
```

DBConext

Entitäten konfiguriern

Der DBContext wird gleichzeitig verwendet um Entiäten zu konfigurieren.

Hinweis: Eine Entität kann zur Gänze im DBContext konfiguriert werden.

DBContext

Fluent API vs. Annotationen

```
public class CookbookContext : DbContext {
    ...
    protected override void OnModelCreating(ModelBuilder
        builder){
        builder.Entity<Dish>()
            .ToTable("DISHES")
            .Property(d => d.Name)
            .HasColumnName("TITLE")
            .HasColumnType("VARCHAR(50)")
            .IsRequired();

        builder.Entity<Dish>()
            .Property(d => d.Description)
            ...
    }
}
```

DBContext

Fluent API: Primary Key

```
public class CookbookContext : DbContext {

   protected override void OnModelCreating(ModelBuilder
       builder){
      builder.Entity<Dish>()
         .HasKey(d => d.Id)
         .ValueGeneratedOnAdd();
      ...
   }
}
```

DBContext

Fluent API: unique Constrain

```
public class CookbookContext : DbContext {

   protected override void OnModelCreating(ModelBuilder
       builder){
      builder.Entity<Dish>()
         .HasIndex(d => d.Name)
         .IsUnique(true);
      ...
   }
}
```

Erweiterte Konzepte

ORM dient als **Brücke** zwischem dem relationalen Datenbankmodell und der objektorientierten Programmierung.

Bestimmte Konzepte der OOP müssen dabei für das relationale Modell neu angedacht werden.

- Vererbung
- Objektreferenzen
- Zusammengesetzte Schlüssel

Erweiterte Konzepte

Vererbung

Zur thematischen Abgrenzung logischer Konzepte werden Entitäten mit gleichen oder ähnlichen Attributen in Vererbungbeziehungen abgebildet.

Der relationale Entwurf abstrahiert 2 Formen von Vererbungsbeziehungen.

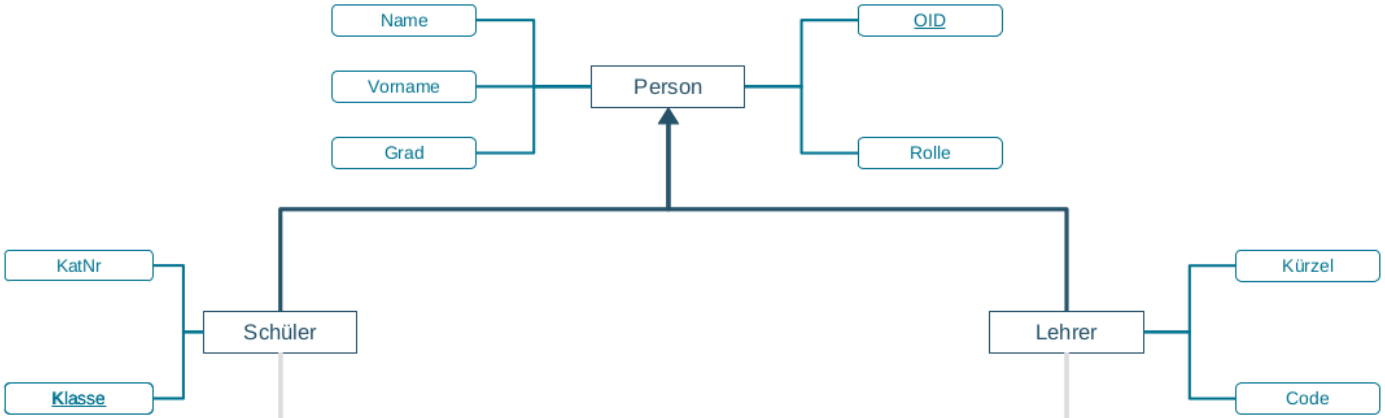
Erweiterte Konzepte

Vererbung: Single Table

Bei der Single Table Modellierung werden die Werte der Objekte der Basisentität und aller Subentitäten, gesammelt in einer einzelnen Tabelle eingetragen.

Erweiterte Konzepte

Vererbung: Single Table



OID	Name	Vorname	Grad	Rolle	KatNr	Klasse	Kürzel	Code	Typ
1	Panhofer	Paul	Dip.-Ing.	KV			PANH	**********	Lehrer
2	Lauer	Christian		SP	5	4aHIT			Schüler
3	Bauer	Iris	Mag.	KV			BAU	**********	Lehrer
4	Kauer	Alexander			4	4aHiT			Schüler
5	Reifenberger	Mika			12	4aHit			Schüler

Schlüssel — Felder — Diskriminator

Entity Framework Core Jump start

Erweiterte Konzepte

Vererbung: Single Table

```
[Table("PERSONEN")]
public class Person {
    [Key]
    [Column("OID")]
    public int Id { get; set; }

    [Required, StringLength(100)]
    [Column("VORNAME")]
    public string FirstName { get; set; }
    ...

}
```

Erweiterte Konzepte

Vererbung: Single Table

```
public class Student : Person {

    [Column("KAT_NR")]
    public int Code { get; set; }

    [Required, StringLength(4)]
    [Column("KLASSE")]
    public string ClassCode { get; set; }

}
```

Erweiterte Konzepte

Vererbung: Single Table

```
public class Teacher : Person {

    [Required, StringLength(4)]
     [Column("CODE")]
     public string Code { get; set; }

     [Required, StringLength(2)]
     [Column("KUERZEL")]
     public string Short { get; set; }

}
```

Erweiterte Konzepte

Vererbung: Single Table

```
public class UniversityContext : DbContext {
    ...
    protected override void OnModelCreating(ModelBuilder
        builder){
        builder.Entity<Person>()
            .HasDiscriminator<string>("TYP")
            .HasValue<Student>("SCHUELER")
            .HasValue<Teacher>("LEHRER");
        ...
    }
}
```

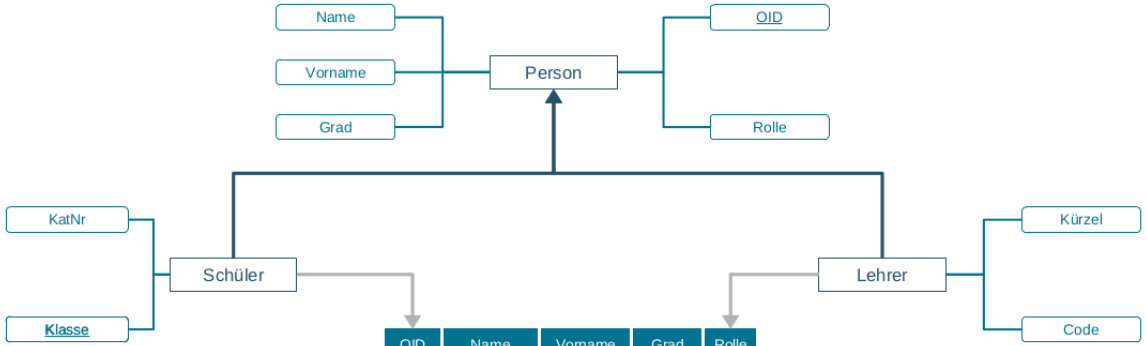
Erweiterte Konzepte

Vererbung: Joined Table

Die Daten der Objekte werden verteilt auf mehrere Tabellen eingetragen.

Erweiterte Konzepte

Vererbung: Joined Table



OID	Name	Vorname	Grad	Rolle
1	Panhofer	Paul	Dip.-Ing.	KV
2	Lauer	Christian		SP
3	Bauer	Iris	Mag.	KV
4	Kauer	Alexander		
5	Reifenberger	Mika		
6	Birnzain	Birgit	Mag	KV
7	Weidenauer	Martin		
8	Kretschmaier	Matthias		
9	Aufmesser	Lukas		
10	Hofstäter	Lukas		

OID	KatNr	Klasse
2	5	4aHIT
4	4	4aHIT
5	12	4aHit
7	21	4aHit
8	5	4aHit
9	1	4aHit
10	2	4aHit

OID	Kürzel	Code
1	PANH	**********
3	BAU	**********
6	BIR	**********

Entity Framework Core Jump start

Erweiterte Konzepte

Vererbung: Joined Table

```
[Table("PERSONEN")]
public class Person {
    [Key]
    [Column("OID")]
    public int Id { get; set; }
   ...
    public string FirstName { get; set; }
    ...

}
```

Erweiterte Konzepte

Vererbung: Joined Table

```
[Table("SCHUELER")]
public class Student : Person {
      ...
      public int Code { get; set; }
       ...
      public string ClassCode { get; set; }

}
```

Erweiterte Konzepte

Vererbung: Joined Table

```
[Table("LEHRER")]
public class Teacher : Person {
      ...
    public string Code { get; set; }
    ...
    public string Short { get; set; }

}
```

Erweiterte Konzepte

Relation: 1:1 Relation

Eine 1:1 Relation besteht zwischen 2 Entitäten wenn eine der Entitäten eine einfache Referenz auf die andere Entität definiert.

Erweiterte Konzepte

Relation: 1:1 Relation

```
[Table("STUDENTS")]
public class Student : Person {

      ...
     public int Code { get; set; }
    ...
     public string ClassCode { get; set; }

}
```

Erweiterte Konzepte

Relation: 1:1 Relation

```
[Table("MATRICULATION_CARDS")]
public class MatriculationCard {
     [Key]
     [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
     public int Id { get; set; }
   ...
     [column(("STUDENT_ID")]
     public int StudentId { get; set; }

     public Student Student { get; set; }
}
```

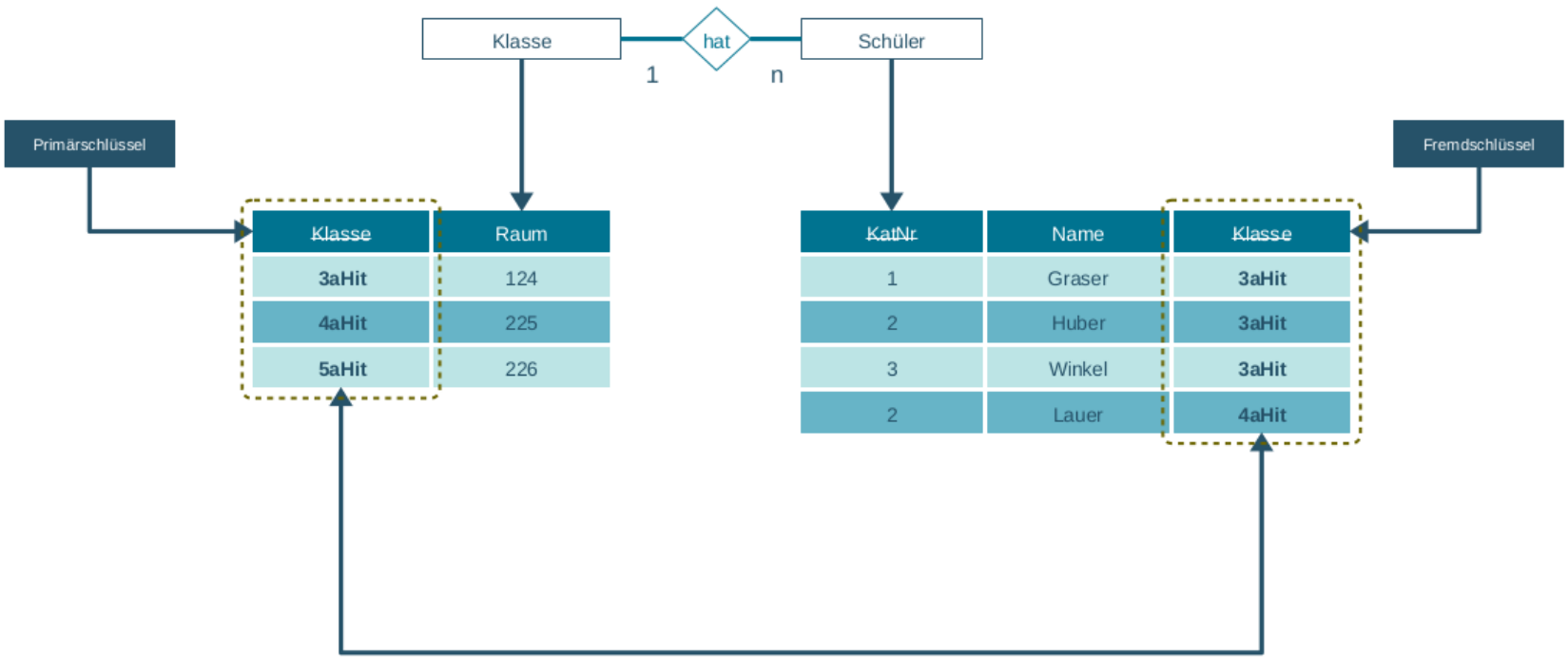
Erweiterte Konzepte

Relation: 1:1 Relation

```
public class UniversityContext : DbContext {
    ...
    protected override void OnModelCreating(ModelBuilder
        builder){
       builder.Entity<MatriculationCard>()
          .HasOne(m => m.Student)
          .WithOne()
          .HasForeignKey<MatriculationCard>(m =>
              m.StudentID);
       ...
    }
}
```

Erweiterte Konzepte

Relation: 1:n Relation



1:n Relation

Erweiterte Konzepte

Relation: 1:n Relation

```
[Table("CLASS_ROOMS")]
public class ClassRoom{

    [Column("ROOM_CODE"), StringLength(4)]
    [Key]
    public string ClassId { get; set; }


}
```

Erweiterte Konzepte

Relation: 1:n Relation

```
[Table("SCHUELER")]
public class Student {
    [Column("STUDENT_ID")]
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    ...
    public ClassRoom Room { get; set;}
    [Column("CLASS_CODE")]
    public int ClassCode { get; set; }
}
```

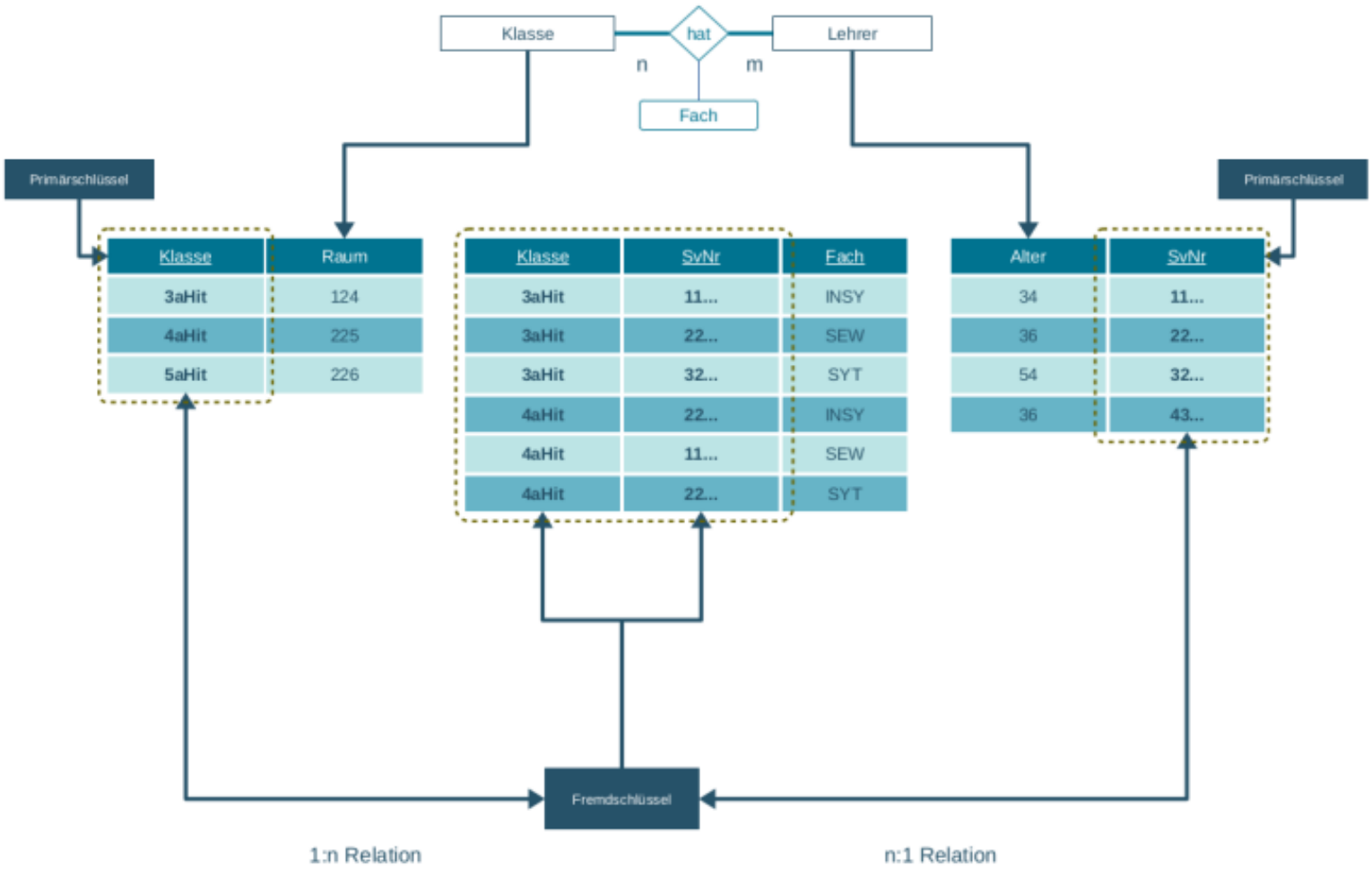
Erweiterte Konzepte

Relation: 1:n Relation

```
public class UniversityDbContext : DbContext {
    ...
    protected override void OnModelCreating (ModelBuilder
        builder){
        builder.Entity<Student>()
            .HasOne( s => s.Room )
            .WithMany()
            .HasForeignKey( s => s.ClassCode);
        ...
    }
}
```

Erweiterte Konzepte

Relation: n:m Relation



Erweiterte Konzepte

Relation: n:m Relation

```
[Table("CLASS_ROOMS")]
public class ClassRoom{

    [Column("KLASSE")]
    [Key]
    public string ClassId { get; set; }

    [Required, StringLength(3)]
    [Column("RAUM")]
    public string RoomCode { get; set; }

}
```

Erweiterte Konzepte

Relation: n:m Relation

```
[Table("KLASSE")]
public class Teacher {

    [Column("SVNR")]
    [Key]
    public int SocialSecurity { get; set; }

    [Required, Range(0, 200)]
    [Column("ALTER")]
    public int Age { get; set; }

}
```

Erweiterte Konzepte

Relation: n:m Relation

```
[Table("TIME_TABLES_JT")]
public class LectureAssingment {
    [Column("SVNR")]
    public int SocialSecurity { get; set; }
    public Teacher Teacher { get; set; }

    [Column("CLASS_ROOM_CODE")]
    public string ClassId { get; set; }
    public ClassRoom ClassRoom { get; set; }

    [Required, StringLength(20)]
    [Column("COURSE")]
    public string Course { get; set; }
}
```

Erweiterte Konzepte

Relation: n:m Relation

```
public UniversityContext : DbContext {
    protected void OnModelCreated(ModelBuilder builder){
        builder.Entity<LectureAssingment>
            .HasKey( l => new { l.SocialSecurity, l.
                ClassId } );

        builder.Entity<LectureAssignment>
            .HasOne(l => l.Teacher)
            .WithMany()
            .HasForeignKey(l.SocialSecurity);

        builder.Entity<LectureAssignment>
            .HasOne(l => l.ClassRoom)
            .WithMany()
            .HasForeignKey(l.ClassId);
    }
}
```

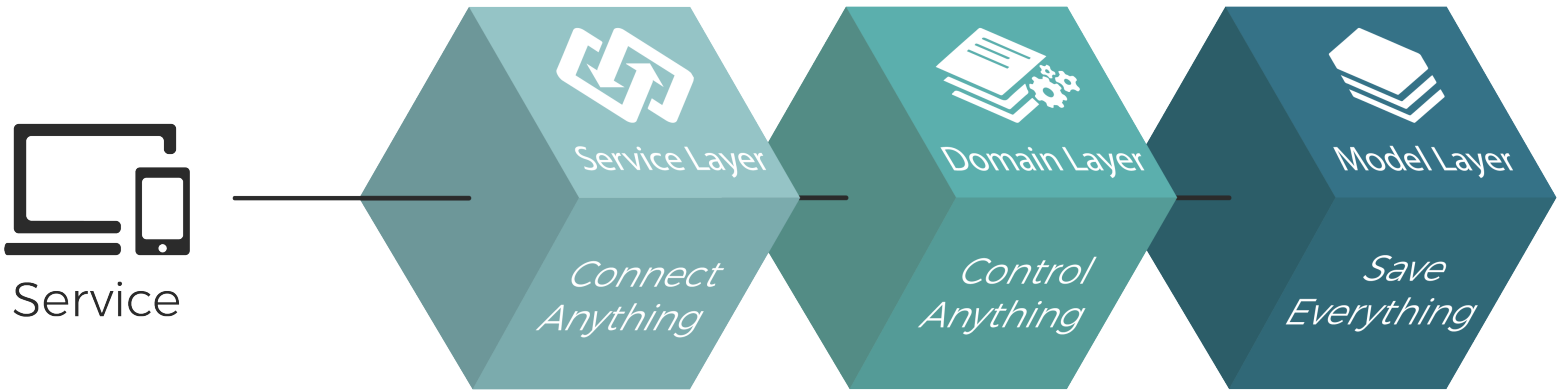
Entity Framework Core Jump start

Domainschicht

Die Domainschicht einer Anwendung ist für die **Verarbeitung** der Daten innerhalb der Anwendung verantwortlich.

Domainschicht

Anwendungsaufbau



Domainschicht

IRepository

Das `IRepository` Interface stellt die Schnittstelle zwischen dem Servicelayer und der Domainschicht verwendet.

Domainschicht

IRepository

Das `IRepository` Interface ist ein generisches Interface. Damit kann es für beliebige Entitäten implementiert werden.

```
public interface IRepository<TEntity> where TEntity :
    class {
    ...

}
```

Domainschicht

IRepository - Create

```
public interface IRepository<TEntity> where TEntity :
    class {

   // Anlegen einer Entitaet
   TEntity Create(TEntity t);

   // Anlegen mehrerer Entitaeten
   List<TEntity> CreateRange(List<TEntity> list);
    ...
}
```

Domainschicht

IRepository - Update

```
public interface IRepository<TEntity> where TEntity :
    class {
   ...
   // Aendern einer Entitaet
   void Update(TEntity t);

   // Aendern mehrerer Entitaeten
   void UpdateRange(List<TEntity> list);
    ...
}
```

Domainschicht

IRepository - Read

```
public interface IRepository<TEntity> where TEntity :
    class {
   ...
   TEntity? Read(int id);

   List<TEntity> Read(
      Expression<Func<TEntity, bool>> filter
   );

   List<TEntity> Read(int start, int count);

   List<TEntity> ReadAll();
    ...
}
```

Domainschicht

IRepository - Delete

```
public interface IRepository<TEntity> where TEntity :
    class {
   ...
   void Delete (TEntity t);
   ...
}
```

Domainschicht

ARepository

Die `ARepository` Klasse implementiert das `IRepository` Interface.

```
public interface ARepository<TEntity> :
    IRepository<TEntity> where TEntity : class {
   protected DbContext _context;
   protected DbSet<TEntity> _table;

   protected ARepository(DbContext context) {
        _context = context;
        _table = context.Set<TEntity>();
    }
    ...
}
```

Domainschicht

ARepository - Create

```
public interface ARepository<TEntity> :
    IRepository<TEntity> where TEntity : class {
   ...
    public TEntity Create(TEntity t) {
        _table.Add(t);
        _context.SaveChanges();
        return t;
    }

    public List<TEntity> CreateRange(List<TEntity> list) {
        _table.AddRange(list);
        _context.SaveChanges();
        return list;
    }
}
```

Domainschicht

ARepository - Update

```
public interface ARepository<TEntity> :
    IRepository<TEntity> where TEntity : class {
   ...
   public void Update(TEntity t) {
        _context.ChangeTracker.Clear();
        _table.Update(t);
        _context.SaveChanges();
    }

    public void UpdateRange(List<TEntity> list) {
        _context.ChangeTracker.Clear();
        _table.UpdateRange(list);
        _context.SaveChanges();
    }
    ...
}
```

Domainschicht

ARepository - Read

```
public interface ARepository<TEntity> :... {
   public TEntity? Read(int id) => _table.Find(id);

   public List<TEntity> Read(Expression<Func<TEntity,
       bool>> filter) =>
        _table.Where(filter).ToList();

   public List<TEntity> Read(int start, int count) =>
        _table.Skip(start)
        .Take(count)
        .ToList();

    public List<TEntity> ReadAll() => _table.ToList();
    ...
}
```

Domainschicht

ARepository - Delete

```
public interface ARepository<TEntity> :... {
    ...
    public void Delete(TEntity t) {
        _table.Remove(t);
        _context.SaveChanges();
     }
     ...
}
```