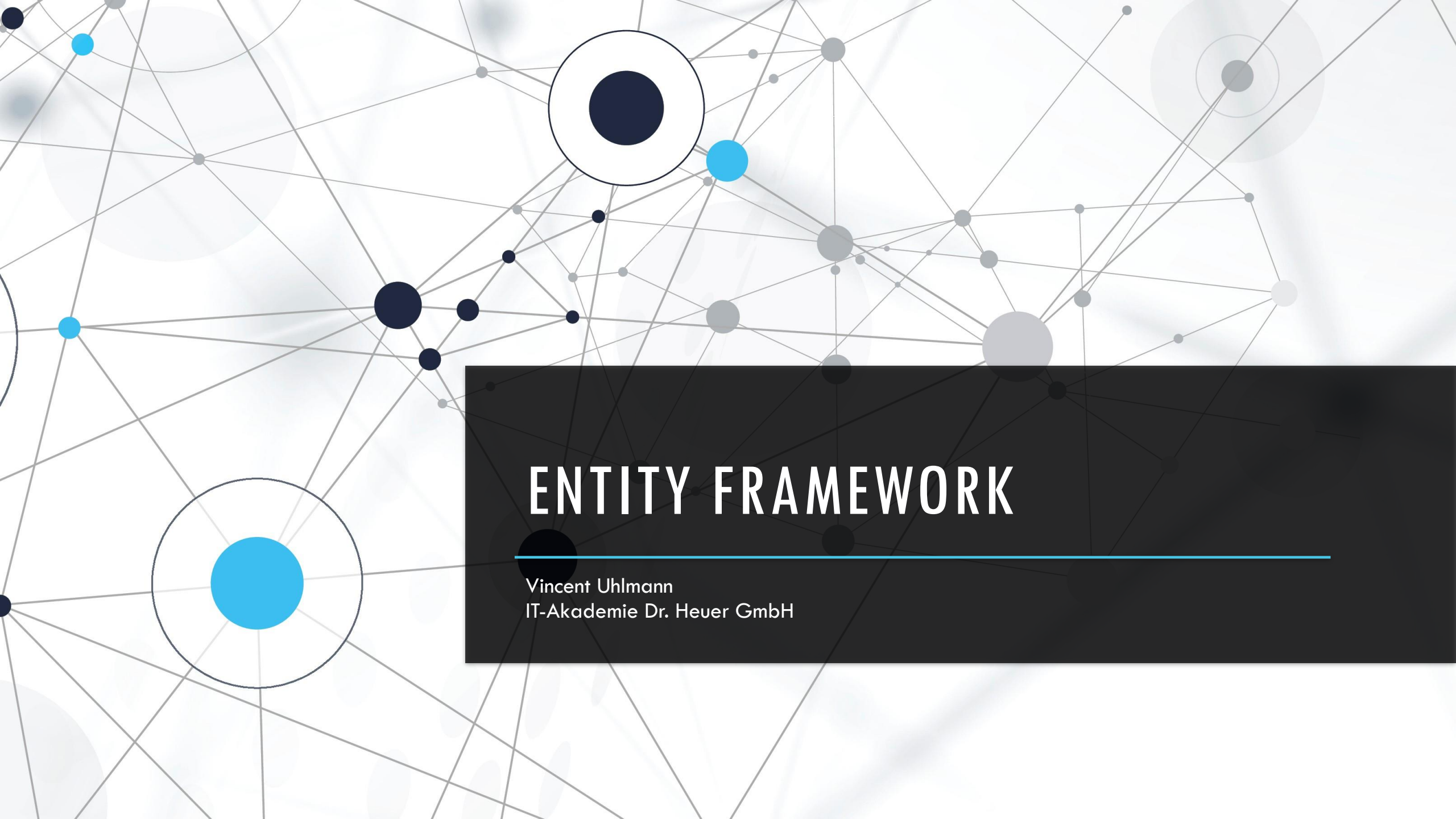


ENTITY FRAMEWORK

Vincent Uhlmann
IT-Akademie Dr. Heuer GmbH

ENTITY FRAMEWORK

- **Was ist das Entity Framework (EF)**
- Ein ORM für .NET
- ORM steht für Object-Relational Mapping
- ORM sorgt für das Mapping zwischen den Datenbanktabellen und den Objekten einer Programmiersprache wie C#
- Erlaubt die Arbeit mit einer Datenbank unter Verwendung von .NET-Objekten
- Unterstützt LINQ-Abfragen, Änderungsverfolgung, Updates und Schema-Migrationen

ENTITY FRAMEWORK

- **Warum das Entity Framework verwenden**
- Vereinfacht die Datenbankinteraktionen
- Abstraktion der Datenbankoperationen
- Verbessert die Wartbarkeit und Lesbarkeit des Codes

ENTITY FRAMEWORK

- **Vorteile des Entity Frameworks**
- Erhöht die Produktivität, da weniger SQL-Abfragen manuell geschrieben werden müssen
- Durch die Typisierung werden Compile-Time-Fehler anstatt Laufzeitfehler ausgelöst
- Die Unterstützung verschiedener Datenbanken ermöglicht den einfachen Wechsel zwischen verschiedenen Anbietern
- Durch LINQ können die Abfragen stark vereinfacht werden

ENTITY FRAMEWORK

- **Grundsätzliche Architektur**
- **Model** / **Entity:** Die Klassen, die die Datenbanktabellen repräsentieren
- **Context:** Die Klasse, die die Datenbankverbindung und –operationen verwaltet
- **Migration:** Verwaltung und Aktualisierung des Datenbankschemas

ENTITY FRAMEWORK

- **EF Core und SQLite-Pakete installieren**
- Um das Entity Framework zu benutzen, müssen mehrere Pakete installiert werden

- **Microsoft.EntityFrameworkCore**
- Das Hauptpaket für EF Core, das die Kernfunktionalitäten bereitstellt, einschließlich der LINQ-Unterstützung

- **Microsoft.EntityFrameworkCore.Sqlite**
- Das SQLite-spezifische Provider-Paket. Ermöglicht EF Core, mit SQLite-Datenbanken zu interagiere

ENTITY FRAMEWORK

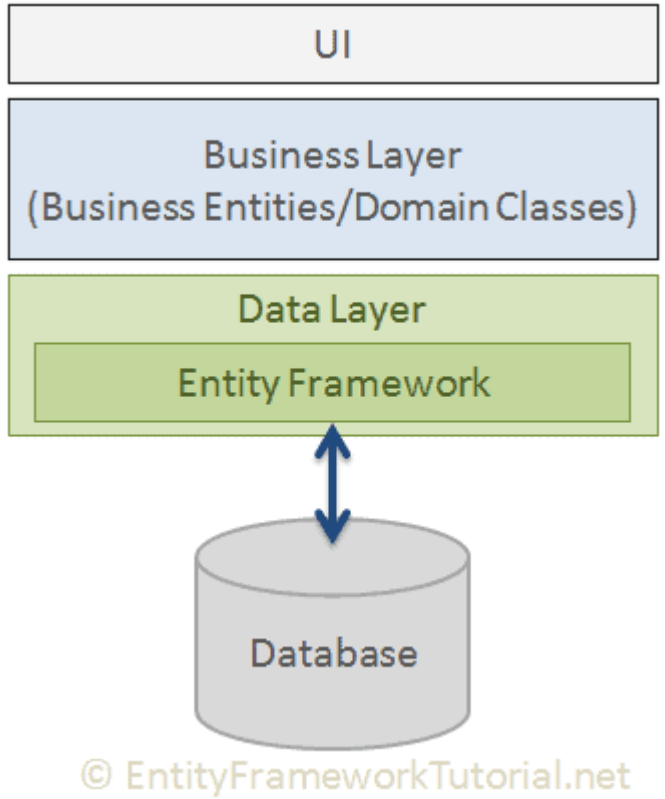
- **Microsoft.EntityFrameworkCore.Design**
- Bietet zusätzliche Tools für die Entwicklung, wie z.B. Migrationen und das Scaffolden von Datenbankmodellen

- **Microsoft.EntityFrameworkCore.Tools**
- Die Entity Framework Core Tools helfen zur Entwurfszeit bei Entwicklungsaufgaben. Diese werden hauptsächlich verwendet, um Migrationen zu verwalten und ein Gerüst für DbContext und Entitätstypen durch Reverse Engineering eines Datenbankschemas zu erstellen.

DATENBANKMODELL DEFINIEREN

- Klasse ProductEntity repräsentiert die Products Tabelle
- Id, Name, Price sind die Spalten der Tabelle
- Die Id Eigenschaft ist der Primärschlüssel

```
public class ProductEntity
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```



DATENBANKKONTEXT ERSTELLEN

- **ApplicationDbContext:** Eine Klasse, die von DbContext erbt und die Verbindung zur Datenbank sowie die Verwaltung der Datenbankoperationen übernimmt
- **DbSet<ProductEntity>:** Eine Sammlung von ProductEntity-Objekten, die mit der Products-Tabelle in der Datenbank verknüpft ist
- **OnConfiguring:** Methode zur Konfiguration der Datenbankverbindung. Hier wird SQLite als Datenbankanbieter festgelegt

DATENBANKKONTEXT ERSTELLEN

```
public class ApplicationContext : DbContext
{
    public DbSet<Product> Products { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        string folder = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
        string dbPath = Path.Combine(folder, "test.db");
        optionsBuilder.UseSqlite("Data Source=" + dbPath);
    }
}
```

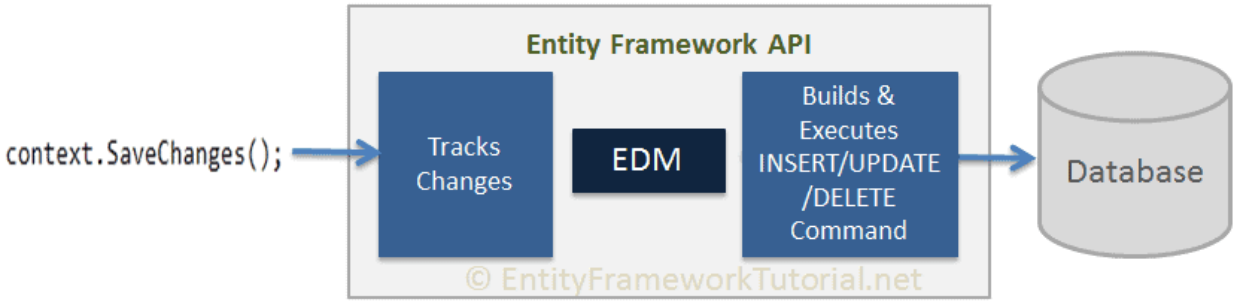
ENSURECREATED

- Erzeugt die Datenbank und das Schema, falls sie noch nicht existieren
- Überprüft nicht auf Schemaänderungen, wenn die Datenbank bereits existiert
- Nur für einfache, prototypische Anwendungen und Testszenarien geeignet
- Einmal verwendet, erlaubt EnsureCreated keine weiteren Migrationen

```
using var context = new ApplicationContext();
context.Database.EnsureCreated();
```

DATEN HINZUFÜGEN

- **Datenbank initialisieren und Daten hinzufügen**
- **EnsureCreated:** Stellt sicher, dass die Datenbank und die Tabellen erstellt werden, wenn sie noch nicht existieren
- **Add:** Fügt ein Product-Objekte zur Products-Tabelle hinzu
- **AddRange:** Fügt mehrere Product-Objekte zur Products-Tabelle hinzu
- **SaveChanges:** Speichert die vorgenommenen Änderungen in der Datenbank



DATEN HINZUFÜGEN

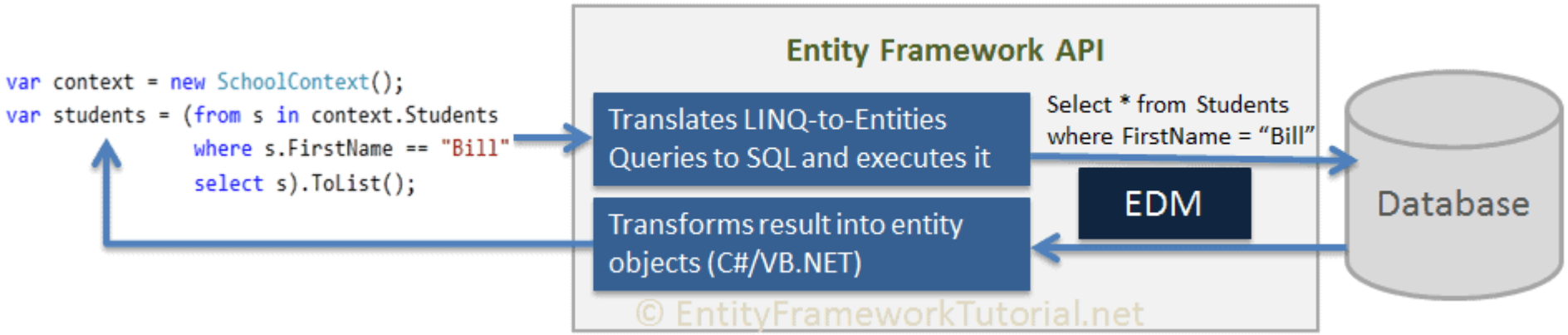
```
using var context = new ApplicationContext();

if (!context.Products.Any())
{
    context.Products.AddRange(
        new Product { Name = "Apple", Price = 1.20M },
        new Product { Name = "Banana", Price = 0.80M }
    );
    context.SaveChanges();
}
```

DATEN ABFRAGEN

```
using var context = new ApplicationContext();

foreach (var product in context.Products)
{
    Console.WriteLine($"ID: {product.Id}, Name: {product.Name}, Price: {product.Price}");
}
```



DATEN ÄNDERN

```
using var context = new ApplicationContext();

var product = context.Products.FirstOrDefault();

if (product != null)
{
    product.Price = 1.20m; // Preis aktualisieren
    context.SaveChanges();
}
```

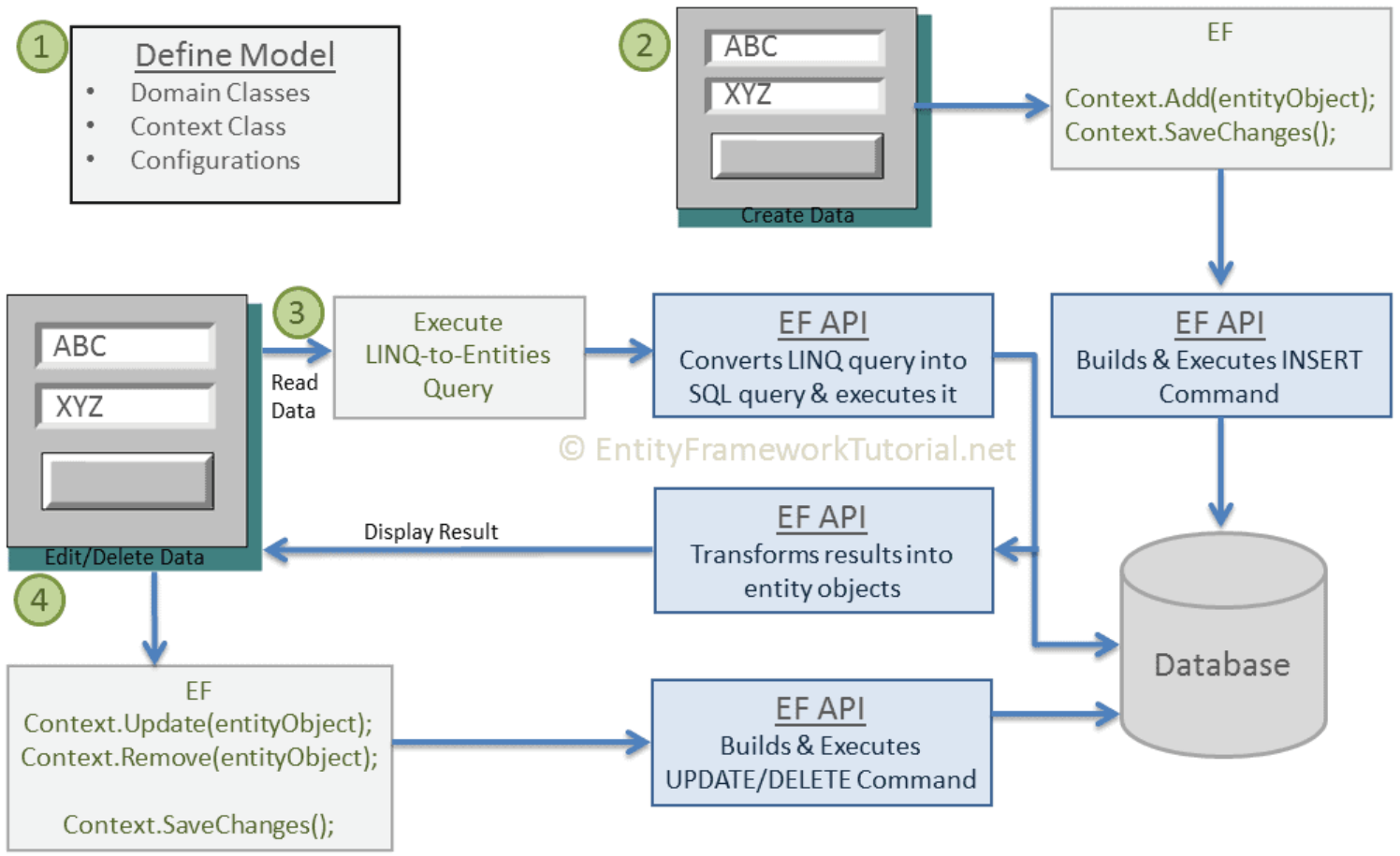
DATEN LÖSCHEN

```
using var context = new ApplicationContext();

var product = context.Products.FirstOrDefault();

if (product != null)
{
    context.Products.Remove(product);
    context.SaveChanges();
}
```

ABLAUF


1
Define Model
Domain Classes
Context Class
Configurations
2
ABC
XYZ
Create Data
EF
Context.Add(entityObject);
Context.SaveChanges();
ABC
XYZ
Edit/Delete Data
3
Read Data
Execute LINQ-to-Entities Query
EF API
Converts LINQ query into SQL query & executes it
EF API
Builds & Executes INSERT Command
© EntityFrameworkTutorial.net
Display Result
EF API
Transforms results into entity objects
Database
4
EF
Context.Update(entityObject);
Context.Remove(entityObject);
Context.SaveChanges();
EF API
Builds & Executes UPDATE/DELETE Command

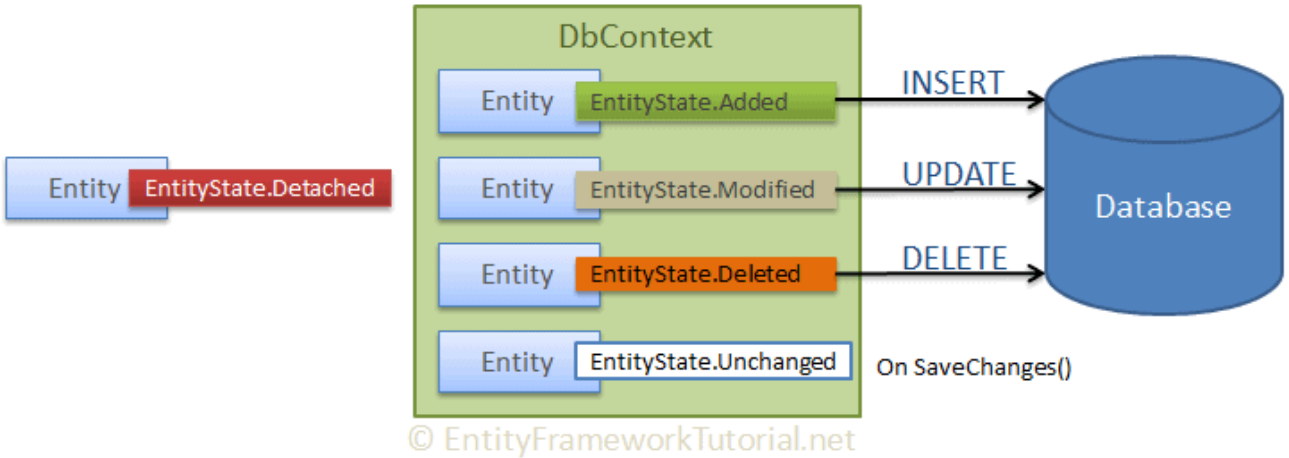
CHANGE TRACKER

- Der Change Tracker ist ein Teil des Entity Frameworks, das Änderungen an den Objekten überwacht, die in einem DbContext geladen oder in diesem erstellt wurden
- Die gemerkten Änderungen können mit SaveChanges in die Datenbank geschrieben werden
- Jeder Entität wird dabei ein Zustand zugewiesen (z.B. Added, Modified, Deleted…)

```
ApplicationDBContext applicationDBContext = new ApplicationDBContext();
applicationDBContext.ChangeTracker.StateChanged += ChangeTracker_StateChanged;
```

ENTITY STATES IN EF

- **Added:** Neue Entität, die zur Datenbank hinzugefügt werden soll
- **Modified:** Bestehende Entität, die geändert wurde
- **Deleted:** Entität, die gelöscht werden soll
- **Unchanged:** Entität, die unverändert ist
- **Detached:** Entität, die nicht von einem Kontext verfolgt wird



CONNECTED VS. DISCONNECTED

- **Connected**
- Ein Zustand, in dem der DbContext während der gesamten Lebensdauer der Entität verfügbar und aktiv ist
- Der Kontext bleibt während der gesamten Datenoperation geöffnet

- **Disconnected**
- Ein Zustand, in dem die Entitäten vom urspünglichen DbContext getrennt und möglicherweise mit einem neuen DbContext verbunden werden müssen
- Der Kontext wird geschlossen und später wieder geöffnet

DISCONNECTED STATE

- Änderungen an Entitäten, die im Disconnected State vorgenommen werden, müssen korrekt an die Datenbank übertragen werden
- Dafür gibt es verschiedene Methoden, die wir nutzen können
- **Attach:** Verbindet eine getrennte Entität mit dem neuen DbContext, ohne deren Zustand zu ändern
- **Entry State:** Markiert die Entität z.B. als verändert

DISCONNECTED STATE

```
Person person;

using (var context = new ApplicationDBContext())
{
    person = context.Persons.First();
}

person.Name = "peter";

using (var context = new ApplicationDBContext())
{
    var attach = context.Attach(person);
    attach.State = EntityState.Modified;
    context.SaveChanges();
}
```

CODE FIRST VS DATABASE FIRST

- **Code First Ansatz**
- Startet mit dem Erstellen von C#-Klassen, die das Datenmodell definieren
- Die Datenbank wird basierend auf diesen Klassen generiert
- Ideal für neue Projekte, bei denen die Datenbank noch nicht existiert

- **Vorteile**
- Flexibilität in der Modellierung
- Einfach zu testen und zu ändern

```
dotnet ef migrations add InitialCreate

dotnet ef database update
```

CODE FIRST VS DATABASE FIRST

- **Database First Ansatz**
- Startet mit einer bestehenden Datenbank und generiert das Datenmodell basierend auf dem bestehenden Schema
- Ideal für Projekte mit vorhandener Datenbank

- **Vorteile**
- Einfacher Einstieg bei bestehenden Datenbanken
- Automatische Modellgenerierung

CODE FIRST VS DATABASE FIRST

- **Nachteile**
- Weniger Flexibilität in der Modellierung

```
dotnet ef dbcontext scaffold "Data Source=C:\Users\vu\Desktop\Krankenhaus.db"
Microsoft.EntityFrameworkCore.Sqlite -o TestModels
```

ONE-TO-ONE CONVENTIONS

- Eine One-to-One-Beziehung tritt auf, wenn eine Entität genau eine Instanz einer anderen Entität besitzt
- EF erkennt die Beziehung durch den Foreign Key (UserId in UserProfile)

```
public class User
{
    public int UserId { get; set; }
    public string UserName { get; set; }
    public UserProfile UserProfile { get; set; }
}

public class UserProfile
{
    public int UserProfileId { get; set; }
    public string Bio { get; set; }

    public int UserId { get; set; }
    public User User { get; set; }
}
```

ONE-TO-MANY CONVENTIONS

- Eine One-to-Many-Beziehung tritt auf, wenn eine Entität mehrere Instanzen einer anderen Entität besitzt
- EF erkennt automatisch die Beziehung den Foreign Key (OrderId in Product) und die Collection (Products in Order)

```
public class Order
{
    public int OrderId { get; set; }
    public string OrderNumber { get; set; }
    public ICollection<Product> Products { get; set; }
}

public class Product
{
    public int ProductId { get; set; }
    public string ProductName { get; set; }

    public int OrderId { get; set; }
    public Order Order { get; set; }
}
```

MANY-TO-MANY CONVENTIONS

- Eine Many-To-Many-Beziehung tritt auf, wenn beide Entität mehrere Instanzen der anderen Entität besitzen
- Eine Join Tabelle wird hier automatisch generiert

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Pet> Pets { get; set; }
}

public class Pet
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Person> Persons { get; set; }
}
```

FLUENT API

- Fluent API ermöglicht es, Konfigurationen für das Datenmodell präzise und detailliert festzulegen
- Sie wird in der OnModelCreating-Methode des DbContext verwendet

- **Beispiel 1:**
- Ändern des Tabellennamens, sodass die "Blog"-Klasse in der Datenbank als "blog"-Tabelle dargestellt wird

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>().ToTable("blog");
}
```

FLUENT API

- **Beispiel 2:**
- Ändern der Spaltennamen, sodass die Post-Eigenschaften Title und Content als "title" und "content" in der Datenbank gespeichert werden

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Post>()
        .Property(p => p.Title)
        .HasColumnName("title");

    modelBuilder.Entity<Post>()
        .Property(p => p.Content)
        .HasColumnName("content");
}
```

FLUENT API

- **Beispiel 3:**
- Konfiguriert die Entitätsbeziehungen und das Verhalten beim Löschen von abhängigen Entitäten

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Nurse>(entity => {
        entity.HasOne(x => x.Address)
            .WithMany()
            .HasForeignKey(x => x.AddressId)
            .OnDelete(DeleteBehavior.Restrict);
    });
}
```

LOGGING

- Der .NET ILogger kann verwendet werden, um die ausgeführten SQL-Befehle zu protokollieren
- Hierfür wird in der OnConfiguring Methode des DbContexts, wird die UseLoggerFactory und ggfs. die EnableSensitiveDataLogging Methode angehangen

```
public class ApplicationDBContext : DbContext
{
    public static readonly ILoggerFactory _loggerFactory = LoggerFactory.Create((options) =>
    {
        options.AddDebug();
    });

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite($"Data Source=C:\\Users\\vu\\Desktop\\test.db")
            .EnableSensitiveDataLogging()
            .UseLoggerFactory(_loggerFactory);
    }
}
```

MIGRATIONEN

- Migrationen sind eine Methode, um Datenbankschemata zu verwalten und zu versionieren, indem sie Änderungen am Datenmodell verfolgen und anwenden

- **Vorteile**
- Automatische Synchronisation von Datenbank und Modell
- Rückverfolgbarkeit von Schemaänderungen

- **Wie funktioniert es?**
- Änderungen im Modell werden erkannt und in C#-Code-Dateien übersetzt, die die Änderungen beschreiben
- Diese Dateien können dann verwendet werden, um die Datenbank zu aktualisieren oder zurückzusetzen

MIGRATIONEN

- Verwalten und verfolgen von Schemaänderungen über die Zeit
- Ermöglicht das Hinzufügen, Ändern und Löschen von Datenbankschemata mittels Migrationen
- Geeignet für Entwicklungs- und Produktionsumgebungen
- Unterstützt Upgrades und Rollbacks von Schemaänderungen
- Datenbank-Updates können über die Kommandozeile, Code oder SQL-Skripte ausgeführt werden

- Beispiel Kommandozeile:

```
dotnet ef migrations add InitialCreate

dotnet ef database update
```

REFERENZEN

- **https://www.entityframeworktutorial.net/**
- **https://learn.microsoft.com/de-de/ef/core/**