### **Verbindung von Programmen mit Datenbanken (ADO.NET vs. Entity Framework)**

**Warum Programme mit Datenbanken verbinden?**

* Ziel: Daten dauerhaft speichern, abrufen, ändern und löschen (CRUD-Operationen).
* Beispiele: Benutzerkonten in Webanwendungen, Produktlisten in E-Commerce-Systemen, Kundeninformationen in CRM-Systemen.

**Einführung in ADO.NET**

"Wie verbindet sich eine Anwendung mit einer Datenbank? Was benötigt man dafür?"

Was ist ein **ConnectionString** (Verbindungszeichenkette)?

* Definition: Eine Zeichenkette, die die notwendigen Informationen enthält, um eine Verbindung zu einer Datenbank herzustellen.
* Beispiele von Informationen: Servername, Datenbankname, Authentifizierungsmethoden, zusätzliche Parameter.

**Data Source**=DESKTOP-KCGE85K\SQLEXPRESS;**Initial Catalog**=Sandbox;**Integrated Security**=True;**Encrypt**=False

* **Data Source:** ist die Adresse des Datenbankservers
* **Initial Catalog:** Name der Ziel-Datenbank
* **Integrated Security:** **True** (Angemeldete Windows-User wird als Datenbankuser genommen) oder **False** (in dem Fall müssen User und Passwort mitgegeben werden)
* **Encrypt:** **True** (erzwingt die Verschlüsselung der Verbindung zwischen Anwendung und Datenbank) oder **False** (nur in Entwicklungsumgebungen empfohlen)

**using** (SqlConnection **sqlConnection** = new SqlConnection(connectionString))

{

string query = "SELECT \* FROM Students";

SqlCommand sqlCommand = new SqlCommand(query, connection);

**sqlConnection**.Open();

SqlDataReader **sqlReader** = **sqlCommand**.ExecuteReader();

while (**sqlReader**.Read())

{

int id = **sqlReader**.GetInt32(0);

string name = **sqlReader**.GetString(1);

int age = **sqlReader**.GetInt32(2);

Console.WriteLine($"ID: {id}, Name: {name}, Alter: {age}");

}

**sqlReader**.Close();

}

**SQL-Befehle (SqlCommand)**

* Um SQL-Befehle auszuführen, verwenden wir SqlCommand.
* Der SQL-Befehl kann **Parameter** enthalten (z. B. @Name, @Age, @Id), die durch Parameters.AddWithValue hinzugefügt werden.
* Dies schützt vor **SQL-Injection-Angriffen**.

**Parameter in SQL**

* Anstatt Werte direkt in die SQL-Abfrage zu schreiben, verwenden wir Platzhalter (@Name, @Age, @Id).
* command.Parameters.AddWithValue("@Name", name) fügt den tatsächlichen Wert in den Befehl ein.

**CRUD-Operationen mit ADO.NET**

|  |  |  |
| --- | --- | --- |
| Operation | Methode | SQL-Befehl |
| Create | InsertStudent | INSERT INTO Students (Name, Age) VALUES (@Name, @Age) |
| Read | GetAllStudents | SELECT \* FROM Students |
| Update | UpdateStudent | UPDATE Students SET Name = @Name, Age = @Age WHERE Id = @Id |
| Delete | DeleteStudent | DELETE FROM Students WHERE Id = @Id |

1. **Was ist ADO.NET?**  
   Sie sehen, wie SQL direkt mit der C#-Anwendung verbunden wird.
2. **Was ist der Unterschied zu Entity Framework?**  
   Sie erkennen, wie Entity Framework den Code vereinfacht, weil SQL-Befehle nicht mehr direkt geschrieben werden müssen.
3. **Warum verwenden wir Parameter?**  
   Parameter schützen vor SQL-Injection-Angriffen. Dies zeigt, warum command.Parameters.AddWithValue sicherer ist als einfache String-Konkatenation.

**Übung**

Verbinden Sie sich mittels ADO.NET mit einer Datenbank und lesen Sie aus einer Tabelle Ihrer Wahl mindestens einen Wert aus und speichern Sie diesen in einer Variablen.

**Entity Framework Core: Grundlagen**

Entity Framework Core (EF Core) ist ein **Object-Relational Mapper** (ORM) für .NET, mit dem sich relationale Datenbanken einfacher verwenden lassen. Es **abstrahiert** den direkten Zugriff auf die Datenbank und ermöglicht es, mit C#-Klassen (**Entity-Klassen**) zu arbeiten, anstatt SQL-Abfragen manuell schreiben zu müssen.

**Hauptmerkmale von EF Core**

* Plattformunabhängig
* Code-First & Database-First: Man kann das Datenbankschema aus C#-Code generieren oder aus einer bestehende Datenbank C#-Klassen generieren.
* LINQ-Unterstützung: Datenbankabfragen können mit LINQ geschrieben werden, wodurch der Code lesbarer und typsicher wird.
* Migrationen: Änderungen an der Datenbankstruktur lassen sich über Migrationsskripte verwalten.
* Unterstützt verschiedene Datenbanken: Neben SQL Server auch SQLite, PostgreSQL, MySQL usw.

**Aufbau**

**DbContext**

* **Definition**: Eine Klasse, die die Verbindung zur Datenbank verwaltet und den Zugriff auf die Tabellen (als DbSet) ermöglicht.
* **Verwendung**:

public class ApplicationDbContext : DbContext

{

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("Server=.;Database=SchoolDB;Trusted\_Connection=True;");

}

}

**Erklärung**:

**DbContext**: Hauptkomponente für die Verbindung zur Datenbank.

**OnConfiguring**: Methode, in der der Connection String definiert wird.

**DbSet**

Ein DbSet entspricht einer Datenbank-Tabelle.

**Beispiel**:

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

**CRUD-Operationen**:

* + **Hinzufügen**: context.Students.Add(new Student { Name = "John" });
  + **Speichern**: context.SaveChanges();
  + **Lesen**: var student = context.Students.FirstOrDefault(s => s.Id == 1);
  + **Ändern**: student.Name = "John Doe"; context.SaveChanges();
  + **Löschen**: context.Students.Remove(student); context.SaveChanges();

**Gemeinsame Übung**

Mini-Projekt: Einfache CRUD (Create-Read-Update-Delete) -Anwendung

1. **Projekt erstellen** (z. B. Konsolen-App).
2. **Entity-Klasse erstellen**:

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

}

1. **DbContext einrichten**:

public class ApplicationDbContext : DbContext

{

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

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer("Server=.;Database=SchoolDB;Trusted\_Connection=True;");

}

}

1. **CRUD-Operationen erstellen**:

void **InsertStudent**(string name, int age)

{

using var context = new StudentContext();

var student = new Student() { Name = name, Age = age };

var students = context.Students.Add(student);

context.SaveChanges();

}

void **GetAllStudents**()

{

using var context = new StudentContext();

var students = context.Students.ToList();

students.ForEach(s => Console.WriteLine($"{s.Id}: {s.Name} - {s.Age} Jahre"));

}

void **UpdateStudent**(string name, int age)

{

using var context = new StudentContext();

var student = context.Students.First();

student.Name = name;

student.Age = age;

context.SaveChanges();

}

void **DeleteStudent**()

{

using var context = new StudentContext();

var student = context.Students.First();

context.Remove(student);

context.SaveChanges();

}

1. **CRUD -Methoden aufrufen**

InsertStudent("Maxima Musterfrau", 20);

GetAllStudents();

UpdateStudent("Max Mustermann", 21);

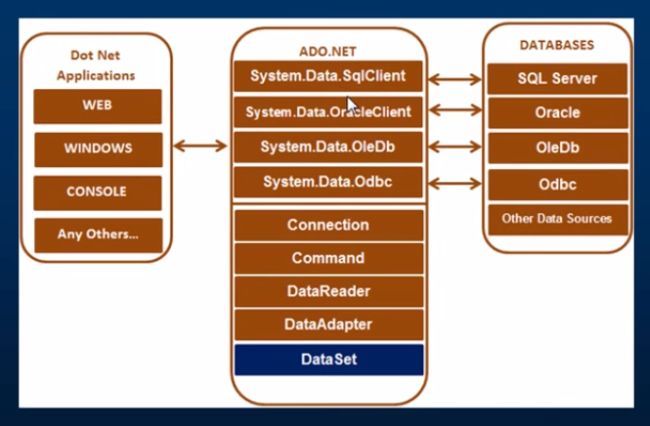
DeleteStudent();

**Einzelübung**

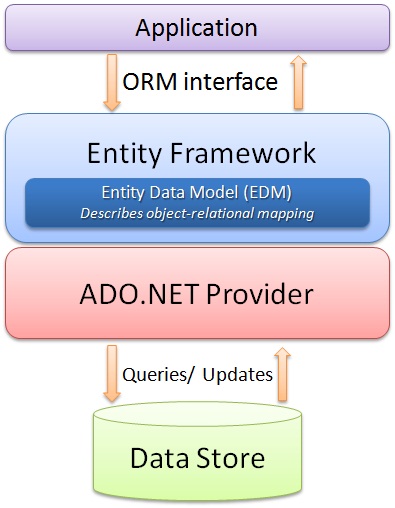
Verbinden Sie sich mittels Entity Framework Core mit einer Datenbank und führen Sie CRUD-Operationen an einer Tabelle Ihrer Wahl aus.

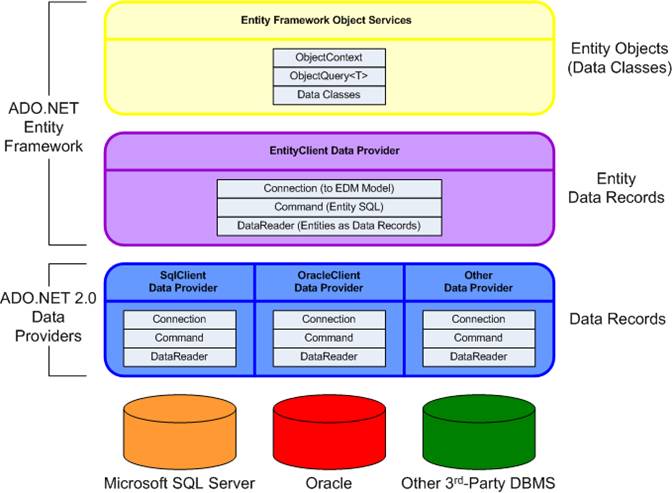
**Unterschied ADO.NET vs. Entity Framework Core**

**Direkt mit ADO.NET**



**Mithilfe von Mapper Entity Framework Core**





**Unterschiede**

|  |  |  |
| --- | --- | --- |
| Kriterium | ADO.NET | Entity Framework |
| Datenzugriff | Direkt, mit SQL-Befehlen | Abstrakt, mit LINQ-Befehlen |
| Komplexität | Komplex, viel Boilerplate-Code (viel Code für nur geringfügige Funktionalität) | Einfacher, weniger Code notwendig |
| Pflegeaufwand | Manuell SQL-Abfragen schreiben | Automatische Generierung der Abfragen |
| Lernkurve | Höher (SQL-Kenntnisse notwendig) | Flacher (LINQ-Kenntnisse reichen) |
| Typische Anwendung | Hochleistungssysteme, spezielle Anforderungen | Schnelle Entwicklung, gängige Web-Anwendungen |

|  |  |  |
| --- | --- | --- |
| Kriterium | ADO.NET | Entity Framework |
| Codeaufwand | Hoch (manuelle SQL-Befehle) | Gering (LINQ-Abfragen) |
| Lesbarkeit | Niedrig (SQL-Befehle sichtbar) | Hoch (LINQ-Befehle lesbar) |
| Fehleranfälligkeit | Hoch (menschliche Fehler möglich) | Gering (Compiler prüft) |
| SQL-Kenntnisse erforderlich? | Ja | Nein (LINQ-Kenntnisse genügen) |
| Geschwindigkeit | Höher (SQL direkt) | Etwas langsamer (durch Abstraktion) |
| Pflegeaufwand | Hoch (SQL-Änderungen nötig) | Gering (nur C#-Code ändern) |

**Warum Entity Framework?**

* Weniger Code, weniger Fehlerpotenzial.
* Bessere Lesbarkeit durch LINQ-Anfragen.
* Schnelle Entwicklung (vor allem bei kleineren Projekten).
* Typensicherheit (Compiler prüft die LINQ-Anfragen).

**IQueryable<T> vs. IEnumerable<T>**

|  |  |
| --- | --- |
| IQueryable<T> | IEnumerable<T> |
| Definition: Abfrage auf Datenquelle (z. B. DB) | **Definition**: In-**Memory-Abfrage** (z. B. Liste) |
| Ausführung: Spät (Lazy) | **Ausführung**: Sofort (Eager) |
| Verwendung: Optimal für DB-Abfragen | **Verwendung**: Optimal für Sammlungen im Speicher |

**Beispiel**:

var query = context.Students.Where(s => s.Age > 18); **// IQueryable**

var result = query.ToList(); **// Erst hier wird die Datenbank abgefragt**

**Vorteil von IQueryable**:

Die Datenbankabfrage (SQL) wird nur dann ausgeführt, **wenn die Daten wirklich benötigt werden**.

**Zusammengefasst:**

Mit ADO.NET können wir präzise und performante Abfragen ausführen, aber es erfordert mehr Aufwand und Wissen über SQL. Entity Framework automatisiert viele dieser Aufgaben und macht den Code lesbarer und wartbarer. In kleinen Projekten oder in der Ausbildung wird oft **Entity Framework** bevorzugt, aber bei Hochleistungssystemen mit feingranularer Steuerung bleibt **ADO.NET** relevant.

<https://www.c-sharpcorner.com/article/entity-framework-0-overview/>

<https://www.c-sharpcorner.com/UploadFile/18fc30/understanding-the-basics-of-ado-net/>