

ADO.NET UND EF CORE



- Sourcen mit Beispielen zum Skript finden sie unter [florianwachs/AspNetWebServicesCourse \(github.com\)](https://github.com/florianwachs/AspNetWebServicesCourse)

ENTITY FRAMEWORK (CORE)

ENTITY FRAMEWORK

- Open Source
- Aktuelle Versionen (2024-04)
 - .NET Full-Framework: EF 6.1
 - .NET : EF (Core) 68.0.x
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools
- **Wichtig: Nur die Microsoft.EntityFrameworkCore Pakete verwenden**

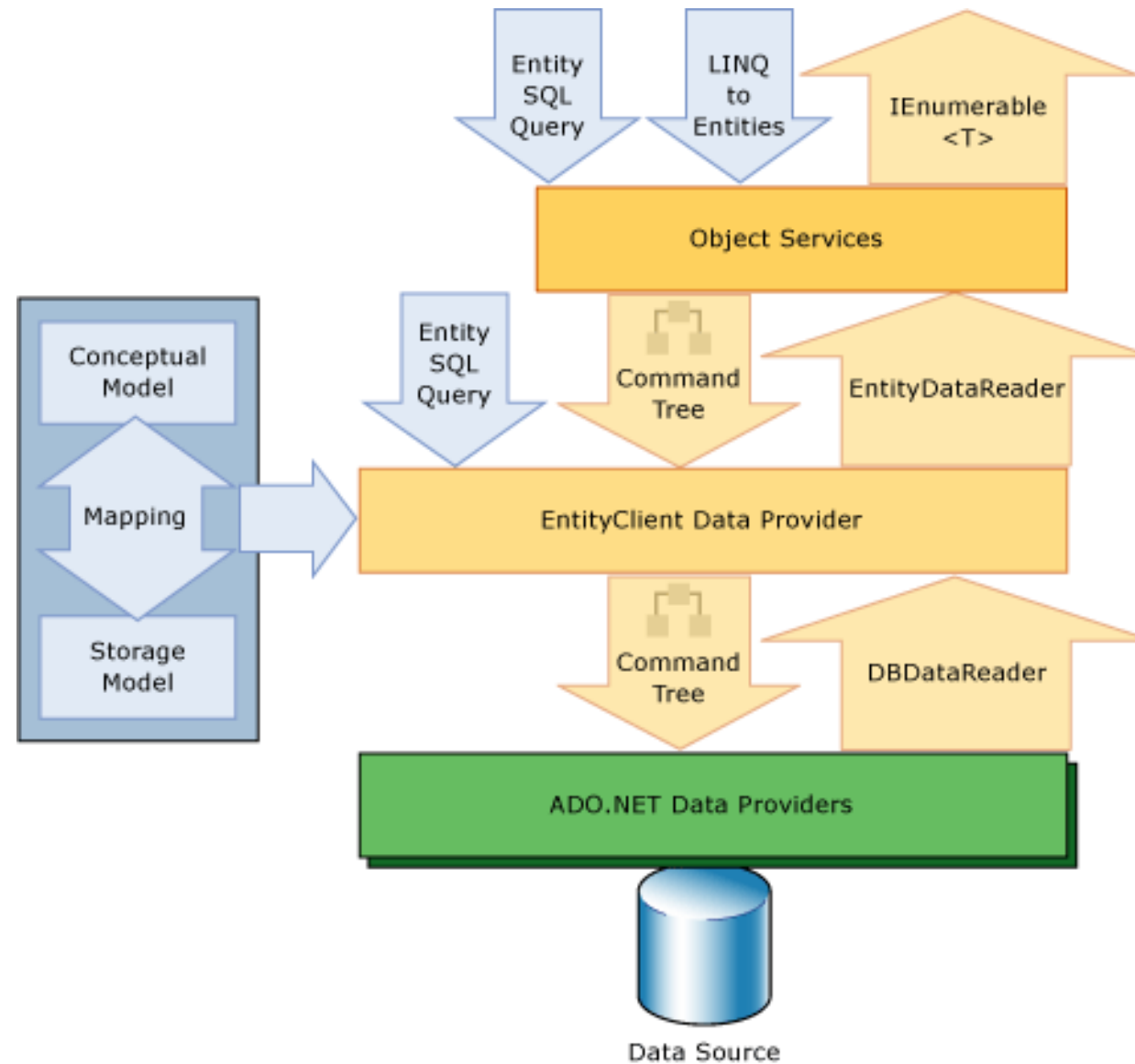
ENTITY FRAMEWORK

- Interaktion mit relationalen Datenbanken über ein Objektmodell das direkt die Business-Objekte der Anwendung abbildet
- Object Relational Mapper
- Favorisierte API für Datenzugriff in .NET
- Kümmert sich um den Aufbau und die Ausführung von SQL-Statements
- Umwandlung von Abfrageergebnissen in Business-Objekte
- Änderungsverfolgung an Business-Objekten
- LINQ ist Grundbestandteil der Architektur, nicht nachträglich hinzugefügt

ENTITY FRAMEWORK

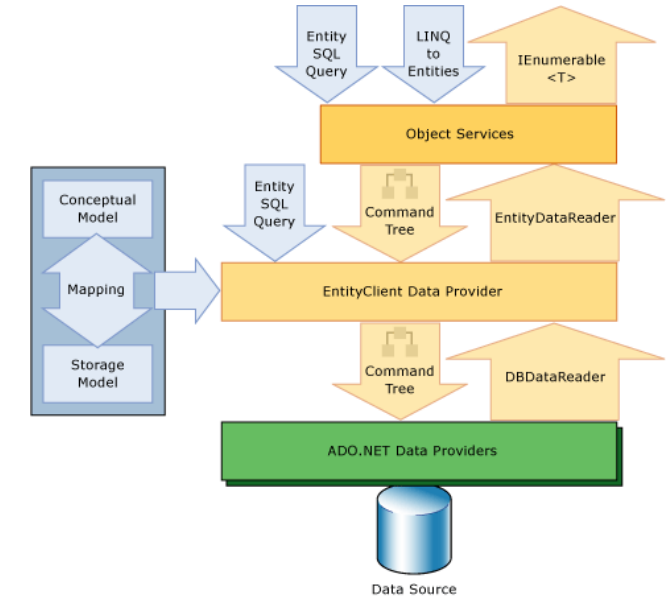
- Reduzierte Abhängigkeiten zum physikalischen Datenbankschema und zu relationalen Datenbanken selbst
- Hohe Testbarkeit: die Business-Objekte können POCO's sein und enthalten damit keine Datenbankzugriffslogik und auch keine harten Abhängigkeiten zum EF
- Es werden keine Collections aus Zeilen und Spalten bearbeitet, sondern Collections aus typisierten Objekten, sog. Entitäten
- Entity Framework Core verwendet den code-first Ansatz, kann aber ein code-first Modell aus einer vorhandenen Datenbank erstellen

OVERVIEW



OVERVIEW

- Object Services
 - Managen die Client-Seite der Entities
 - Verfolgen und speichern Änderungen
 - Verwalten Beziehungen zwischen den Entities
- Entity Client
 - Bilden die Entity-Kommandos auf den darunterliegenden Data Provider ab
- Entwurf in drei Modellen
 - Domänenmodell (Entitäten und Beziehungen)
 - Logisches Modell (Abbildung auf Tabellen, Normalisierung)
 - Physisches Modell (bezieht sich auf Funktionen eines bestimmten Datenmoduls, Indizierung, Partitionierung)



VORTEILE EINES OR-MAPPERS

- Bekanntes Umfeld in der objektorientierten Entwicklung
 - Kenntnisse von SQL, Datenbankschemas nicht zwingend notwendig
 - Höhere Entwicklungsgeschwindigkeit da die SQL-Tabellen zu einem späteren Zeitpunkt generiert werden können (In-Memory-Database)
- Automatische Änderungserkennung an Objekten und Generierung von SQL für das Update
- Abstraktion von der zugrundeliegenden Datenbank-Technologie
 - MS SQL
 - Postgres
 - SQLite
 - MySQL

NACHTEILE EINES OR-MAPPERS

- Queries können inperformant werden (n+1 Problem)
- Insgesamt ist die Performance relativ gesehen zur direkten Implementierung langsamer (high-traffic)

CODE-FIRST| Entitäten

- Entitäten sind Plain Old CSharp Objects
- Über Attribute können Properties konfiguriert werden
 - Validation
 - Mapping auf DB-Columns
 - Primär / Fremdschlüssel
 - [NotMapped]
- Die Konfiguration kann aber auch getrennt vom Modell über EntityConfiguration-Klassen geregelt werden
- ICollection-Member die **virtual** sind, unterstützen **Lazy-Loading**

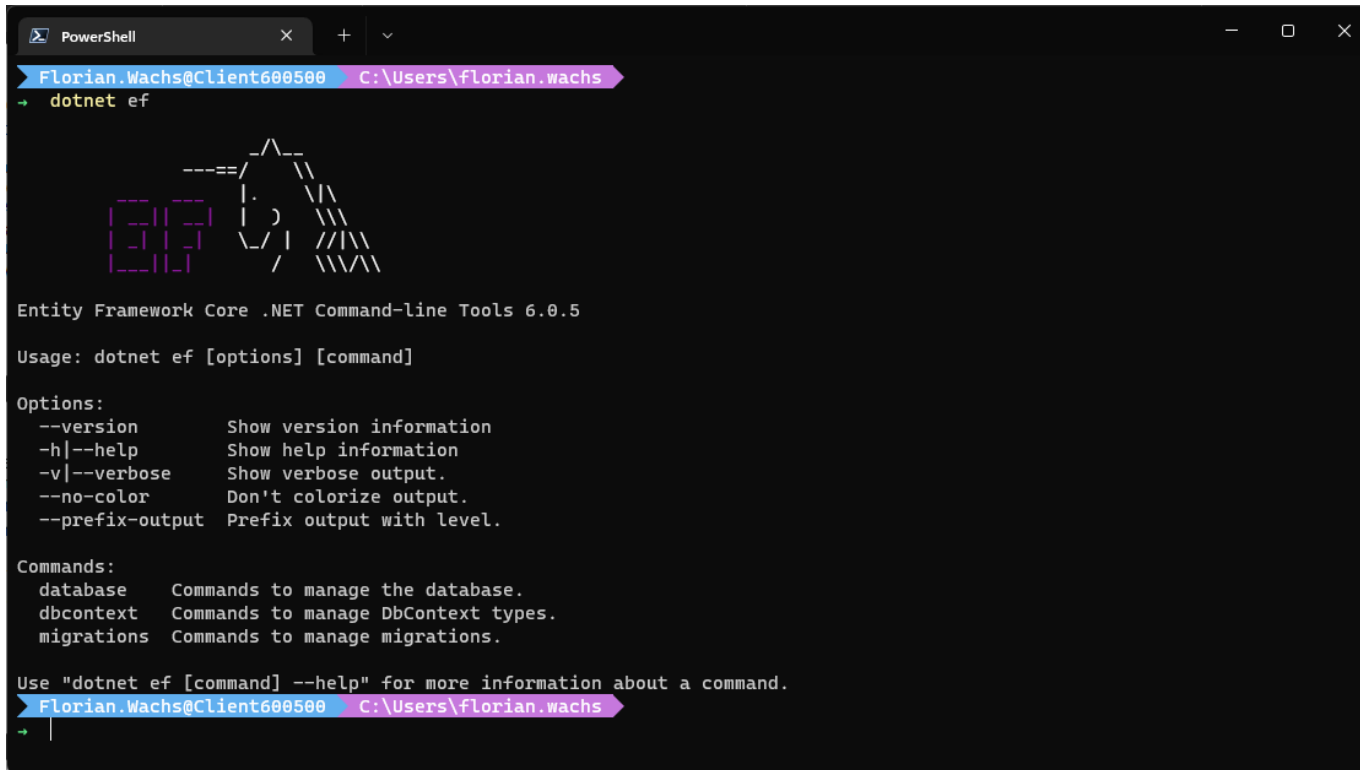
EF CORE| Building Blocks

- Datenprovider auswählen
 - MS SQLServer, PostgreSQL, Sqlite...
- DbContext und Entitäten modellieren
- DbContext im Dependency Injection System konfigurieren und registrieren
- Migrationen für das Datenmodell generieren
- Migrationen auf die Datenbank anwenden

EF CORE| Installation

- NuGet: Microsoft.EntityFrameworkCore.SqlServer
 - Paket für Microsoft SQL-Server , localdb
 - Ist im ASP.NET Core Metapackage bereits enthalten
- NuGet: Microsoft.EntityFrameworkCore.*
 - Weitere DB Provider wie CosmosDB, Postgres, MySql, SQLite verfügbar
- NuGet: Microsoft.EntityFrameworkCore.Tools.DotNet
 - Commandozeilen Tool (seit .NET (Core) 2.2 SDK bereits global installiert)
- NuGet: Microsoft.EntityFrameworkCore.Design
 - Paket für die Erstellung von Migrationen

EF CORE| Installation



```
PowerShell
Florian.Wachs@Client600500 C:\Users\florian.wachs
dotnet ef

Entity Framework Core .NET Command-line Tools 6.0.5

Usage: dotnet ef [options] [command]

Options:
  --version          Show version information
  -h|--help          Show help information
  -v|--verbose       Show verbose output.
  --no-color         Don't colorize output.
  --prefix-output    Prefix output with level.

Commands:
  database          Commands to manage the database.
  dbcontext          Commands to manage DbContext types.
  migrations        Commands to manage migrations.

Use "dotnet ef [command] --help" for more information about a command.
Florian.Wachs@Client600500 C:\Users\florian.wachs
```

Danach sind die Entity Framework Commands über die CLI anwendbar

dotnet tool install --global dotnet-ef
dotnet tool update --global dotnet-ef

EF CORE| Entitäten

Sollen Ids nicht automatisch erzeugt werden, kann dieses Attribute dies verhindern

```
public class Author
{
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int Id { get; set; }

    [Required]
    [MaxLength(255)]
    public string FirstName { get; set; }

    [Required]
    [MaxLength(255)]
    public string LastName { get; set; }

    [Range(0, 150)]
    public int Age { get; set; }

    public ICollection<ContactInfo> ContactInfos { get; set; }
    public ICollection<BookAuthorRel> BookRelations { get; set; }
}
```

Validierungsattribute können verwendet werden, um die Konsistenz der Daten sicherzustellen

Relationen werden unterstützt, komplizierte Fälle wie many-to-many müssen aber noch nachkonfiguriert werden

EF CORE| DbContext

- Einstiegspunkt für das EF um das Objektmodell verwalten zu können und damit zu arbeiten
- Entitäten die direkt abgefragt und verarbeitet werden sollten in `DbSet<TEntity>`-Properties veröffentlicht werden
- Über den Context kann das Modell vor der ersten Verwendung konfiguriert werden
- Der Context bietet Lifecycle-Hooks wie `SaveChanges()` und `ShouldValidateEntity()` die überschrieben werden können

EF CORE| DbContext

- Über den Context wird auch der zu verwendende Data Provider konfiguriert (meist in .config, .json)
- Über den Context kann auf den **Change Tracker** zugegriffen werden, welcher Änderungen an den Entitäten aufzeichnet
- Über **SaveChanges()** werden die Änderungen in der DB persistiert
- Enthält einen Cache für bereits geladene Entitäten
 - Daher sollten Entitäten nicht am EF vorbei modifiziert werden

EF CORE| DbContext

```
public class BookDbContext : DbContext
{
    // Die Options enthalten Informationen für die DB-Connection mit der das EF-Framework
    // auf die DB zugreifen soll
    public BookDbContext(DbContextOptions<BookDbContext> options)
        : base(options)
    {
    }

    // Die Entitäten die direkt abgefragt werden können sollen,
    // werden über DbSet angegeben. Es müssen nicht alle Entitäten
    // angegeben werden
    public DbSet<Book> Books { get; set; }
    public DbSet<Author> Authors { get; set; }

    // Hier können noch Konfigurationen an Entitäten
    // und Conventions durchgeführt werden, bevor
    // das Modell benutzbar ist
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
    }
}
```

EF CORE| DbContext

OnModelCreating erlaubt die Modifizierung von Conventions oder die direktere Definition von Relationen, was bei many-to-many leider aktuell noch notwendig ist

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Entity<Book>().Property(b => b.ReleaseDate).HasColumnType("datetime2");
}
```

ColumnTypes können auch über Attribute in den Entitäten gesetzt werden

EF CORE| DBContext

OnModelCreating erlaubt die Modifizierung von Conventions oder die direktere Definition von Relationen.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{

    modelBuilder.Entity<BookAuthorRel>()
        .HasKey(t => new { t.BookId, t.AuthorId });

    modelBuilder.Entity<BookAuthorRel>()
        .HasOne(pt => pt.Book)
        .WithMany(p => p.AuthorRelations)
        .HasForeignKey(pt => pt.BookId);

    modelBuilder.Entity<BookAuthorRel>()
        .HasOne(pt => pt.Author)
        .WithMany(t => t.BookRelations)
        .HasForeignKey(pt => pt.AuthorId);

}
```

EF CORE| DbContext

EntityConfigurations helfen OnModelCreating
kompakt und übersichtlich zu halten

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new BookConfiguration());
}
```

```
public class BookConfiguration : IEntityTypeConfiguration<Book>
{
    public void Configure(EntityTypeBuilder<Book> builder)
    {
        builder.Property(b => b.Isbn).IsRequired();
        builder.Property(b => b.Title).IsRequired().HasMaxLength(500);
    }
}
```

EF CORE| DbContext

```
public void Query(BookDbContext context)
{
    // Sucht Entitäten nach ihrer Id
    context.Books.FindAsync();

    // Liefert alle Bücher
    context.Books.ToListAsync();

    // Linq Expressions werden soweit möglich vom Linq Provider in SQL Statements
    // umgewandelt und direkt in der Datenbank ausgeführt (endlich auch OrderBy...)
    context.Authors.Where(author => author.Age > 18 && author.FirstName.StartsWith("K"))
        .OrderBy(author => author.LastName).ToListAsync();

    // Es lassen sich auch nur einzelne Properties abrufen
    context.Authors.Select(a => new {a.Age}).ToListAsync();
    // Oder mit Tuple
    context.Authors.Select(a => (a.Age)).ToListAsync();
}
```

EF CORE| DbContext

```
public void Edit(BookDbContext context)
{
    var book = GetBook();

    context.Books.AddAsync(book);
    context.Books.Update(book);
    context.Books.Remove(book);

    // Alle Änderungen müssen explizit gespeichert werden
    context.SaveChangesAsync();
}
```

EF CORE| DbContext

```
public void QueryRelations(BookDbContext context)
{
    // Eager loading
    context.Books.Include(b => b.Authors).ThenInclude(a => a.Addresses);

    //Explicit und Lazy-Loading finden Sie hier:
    // https://docs.microsoft.com/de-de/ef/core/querying/related-data
}
```


EF CORE| DbContext

```
void UseSqlServerLocalDb(IServiceCollection services)
{
    services.AddDbContext<BookDbContext>(options =>
        options.UseSqlServer(builder.Configuration.GetConnectionString("LocalDb")));
}
```

Über die options können je nach DB-Provider weitere Eigenschaften der Verbindung zur DB beeinflusst werden

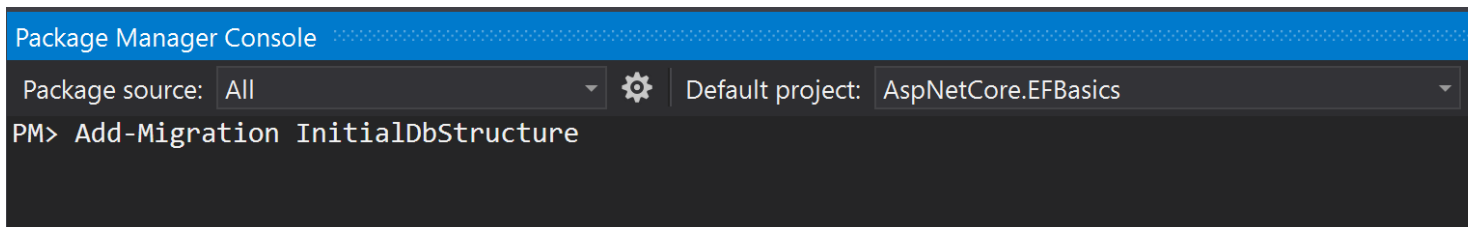
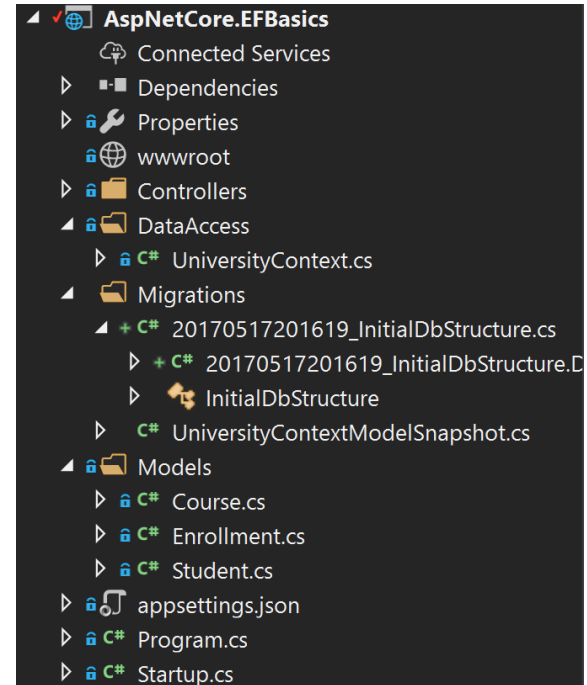
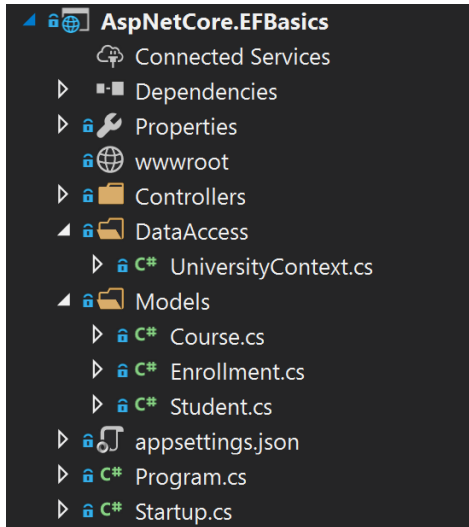
EF CORE| DbContext

```
{  
  "ConnectionStrings": {  
    "Default": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=Books;Integrated Security=SSPI"  
  },  
  "Logging": {  
    "LogLevel": {  
      "Default": "Warning"  
    }  
  },  
  "AllowedHosts": "*"   
}
```

EF CORE| Migrations

- Änderungen am Objektmodell müssen in die Datenbank übertragen werden
- Migrations sind ein Feature von EF die dies erleichtern sollen
- Neue Migration über **Add-Migration [name]**
- Auf die Datenbank anwenden mit **Update-Database**
- Visual Studio->View->Other Windows->**Package Manager Console**
 - Default project muss auf das Projekt mit dem DbContext eingestellt sein für das die Migrations erzeugt werden sollen
- Mit Remove-Migration können Migrationen auch wieder entfernt werden

EF CORE| Migrations



EF CORE| Migrations

- Verwendung der dotnet cli
- dotnet tool install --global dotnet-ef
- dotnet ef migrations add Initial

```
→ dotnet ef migrations add "initial"  
Build started...  
Build succeeded.  
Done. To undo this action, use 'ef migrations remove'
```

EF CORE| Migrations

```
public partial class Initial : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Books",
            columns: table => new
            {
                Id = table.Column<string>(nullable: false),
                Title = table.Column<string>(maxLength: 500, nullable: false),
                Isbn = table.Column<string>(nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Books", x => x.Id);
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Books");
    }
}
```

MIGRATIONEN In der Anwendung ausführen

```
public class DbSeeder
{
    public async Task Seed(IServiceProvider provider)
    {
        // Um auf das DI-System zuzugreifen muss ein neuer Scope erstellt werden,
        // in dem die erzeugten Objekte "leben"
        using var scope = provider.CreateScope();
        var dbContext = scope.ServiceProvider.GetRequiredService<BookDbContext>();

        // Mittels Migrate werden alle ausstehenden Db-Migrationen angewendet.
        // Vorsicht wenn mehrere Instanzen versuchen das Upgrade der DB auszuführen.
        // In Produktivsystemen führt man das DB-Upgrade meist getrennt vom Applikationsstart aus.
        await dbContext.Database.MigrateAsync();

        // Häufig werden beim initialen Anlegen der DB einige Stammdaten benötigt.
        // Der Prozess des Befüllens wird oft als Seeding bezeichnet.
        await SeedDb(dbContext);
    }
}
```

Es gibt auch `EnsureCreated()`, das legt aber nur die DB ohne Migrationsmöglichkeit an

MIGRATIONEN In der Anwendung ausführen

```
var app = builder.Build();

// Zwischen Build() und Run() ist eine günstige Gelegenheit für einfache Anwendungen / Microservices
// die Migrationen auszuführen.
// Falls mehrere Instanzen der gleichen App gestartet werden und auf die gleiche DB zeigen sind komplizierte
// Migrationsstrategien notwendig, etwa das Migration-Bundle Feature oder direkte SQL Erzeugung mit dem EF-Tool
var seeder = new DbSeeder();
await seeder.Seed(app.Services);

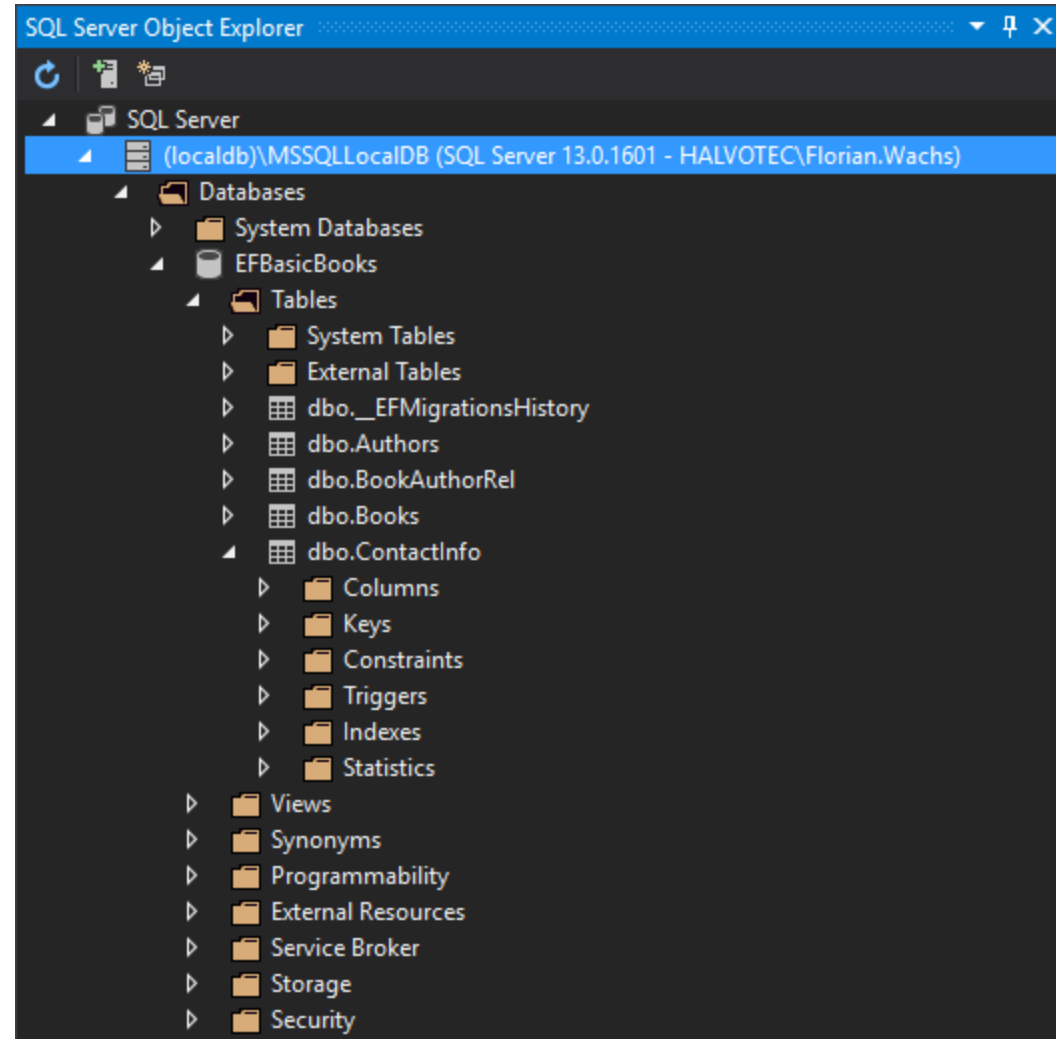
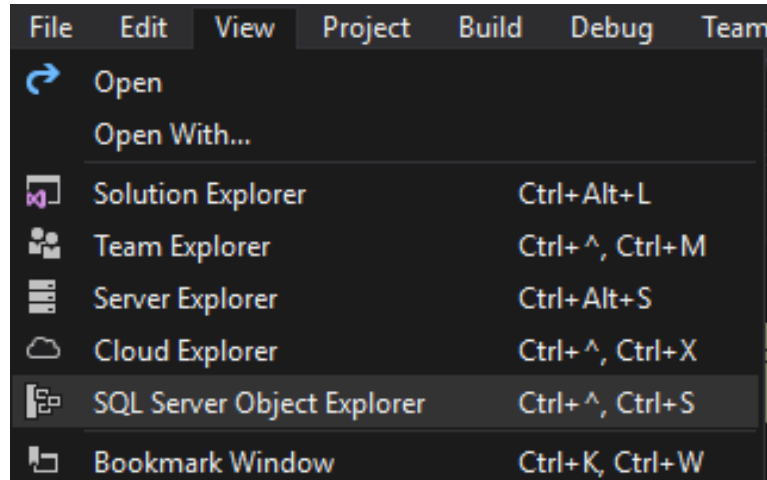
// ...

app.Run();
```


LOCALDB

- Im Visual Studio wird ab Version 2015 eine eigene SQL-Server-Engine mitinstalliert, welche für Entwicklerzwecke optimiert ist
- LokalDb hat nahe zu alle Features einer Produktivversion vom SQL-Server
- Es gibt Leistungseinschränkungen was die max. Anzahl an Cores und Speicher betrifft
- Wird eine Verbindung per Connection-String hergestellt, startet LocalDb automatisch
 - Server=(**localdb**)\mssql**localdb**;Database=EFBasicBooks;Trusted_Connection=True;
 - Falls die DB nicht existiert, wird sie angelegt
- LocalDb benötigt keine Administratorberechtigung

LOCALDB



ENTITY FRAMEWORK CORE IN ASP.NET MINIMAL API

Eine Instanz des Contexts kann per
Dependency Injection übergeben werden.
Für jeden Request wird ein frischer
Context erzeugt

```
void RegisterMinimalApiRoutes(WebApplication app)
{
    app.MapGet("/api/v1/books", GetBooks);
}
```

```
async Task<IResult> GetBooks(BookDbContext dbContext)
{
    var books = await dbContext.Books.ToListAsync();
    return Results.Ok(books);
}
```

ENTITY FRAMEWORK CORE IN ASP.NET CORE WEB API

Eine Instanz des Contexts kann per
Constructor Injection übergeben werden.
Für jeden Request wird ein frischer
Context erzeugt

```
public BooksController(BookDbContext bookDbContext)
{
    BookDbContext = bookDbContext;
}
```

ENTITY FRAMEWORK CORE IN ASP.NET CORE WEB API

```
[HttpGet("")]
public IEnumerable<Book> GetBooks()
{
    return BookDbContext.Books;
}
```

Über die DbSet-Properties kann auf das Datenmodell zugegriffen werden. Im Prinzip wie Linq-to-Objects für SQL

```
// GET api/books/1
[HttpGet("{id:int}", Name = "BookById")]
public IActionResult GetBookById(int id)
{
    var book = BookDbContext.Books.Find(id);
    return book != null ? (ActionResult)Ok(book) : NotFound();
}
```

Mit Find lassen sich Objekte anhand des Primary-Keys finden. Ein DbContext hat ein eigenes Caching. Wurde das Objekt bereits geladen wird es wiederverwendet.

ENTITY FRAMEWORK CORE IN ASP.NET CORE WEB API

```
[HttpPost]
public IActionResult CreateBook([FromBody] Book book)
{
    // Während des Model Bindings werden die Validatoren
    // von Book geprüft
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    BookDbContext.Books.Add(book);
    BookDbContext.SaveChanges();
}
```

Werden Änderungen durchgeführt
müssen diese mit SaveChanges final in
die DB geschrieben werden

ENTITY FRAMEWORK CORE IN ASP.NET CORE WEB API

```
public async Task<IActionResult> AddAuthorToBook(int id, [FromBody] AuthorRefDto author)
{
    var book = await BookDbContext.Books.Include(b => b.AuthorRelations).Where(b => b.Id ==
id).FirstOrDefaultAsync();

    var existingAuthor = await BookDbContext.Authors.FindAsync(author.Id);
    book.AuthorRelations.Add(new BookAuthorRel { BookId = book.Id, AuthorId = existingAuthor.Id });
    await BookDbContext.SaveChangesAsync();

    return Ok(book);
}
```

EF Core unterstützt Async-Apis, diese sollten soweit möglich verwendet werden

ENTITY FRAMEWORK CORE| Fazit & Ausblick

- Favorisierte Datenzugriffs-API von Microsoft
- EF Core wird für neue Applikationen empfohlen
 - wird auch für Cloud-Einsätze optimiert
 - Unterstützt alternative Storage-Konzepte wie NoSQL und InMemory (CosmosDB Provider)
- ORM-Mapper sind eine Abstraktion und haben Performanceimplikationen

ENTITY FRAMEWORK| Empfehlungen

- DbContext nicht direkt in Controllern verwenden (kapseln durch z.B. Repository Pattern oder CQRS)
- Nicht das Domain-Model aus den APIs liefern, DTOs mit nur den exakt benötigten Daten erstellen (Tools wie Automapper helfen dabei)
- Bei großen Importläufen immer mal wieder den Context neu erzeugen
- Für performancekritische Anwendungsfälle lieber auf Ado.Net oder Dapper zurückgreifen oder RAW-Sql über EF nutzen

ENTITY FRAMEWORK| Alternativen

- NHibernate
 - <https://github.com/nhibernate/nhibernate-core>
- Dapper.Net
 - <https://github.com/StackExchange/dapper-dot-net>

DAPPER.NET

- Erfunden und gepflegt von den StackOverflow-Betreibern
- Dünner Layer über ADO.NET (SQL-Connections)
- Trotzdem den Vorteil des Objekt-Mappings
- <https://github.com/StackExchange/Dapper>
- Kein Konzept von „Migrationen“ wie bei EF Core

DAPPER.NET

```
public void DapperExecute()
{
    // Dapper verwendet parameterisierte SQL-Statements
    string sql = "INSERT INTO Book (Title) Values (@Title)";

    // Wie bei ADO.NET üblich muss eine SqlConnection erzeugt und initialisiert werden
    using (SqlConnection connection = new SqlConnection(ConnectionString))
    {
        // Anschließend können Queries und Manipulationen vorgenommen werden.
        int affectedRows = connection.Execute(sql, new { Isbn = "123", Title = "Chuck Norris, best of all" });

        var books = connection.Query<Book>("Select * FROM Book").ToList();

        var book = connection.QuerySingleOrDefault<Book>("Select * FROM Book WHERE ISBN = '123'");
    }
}
```