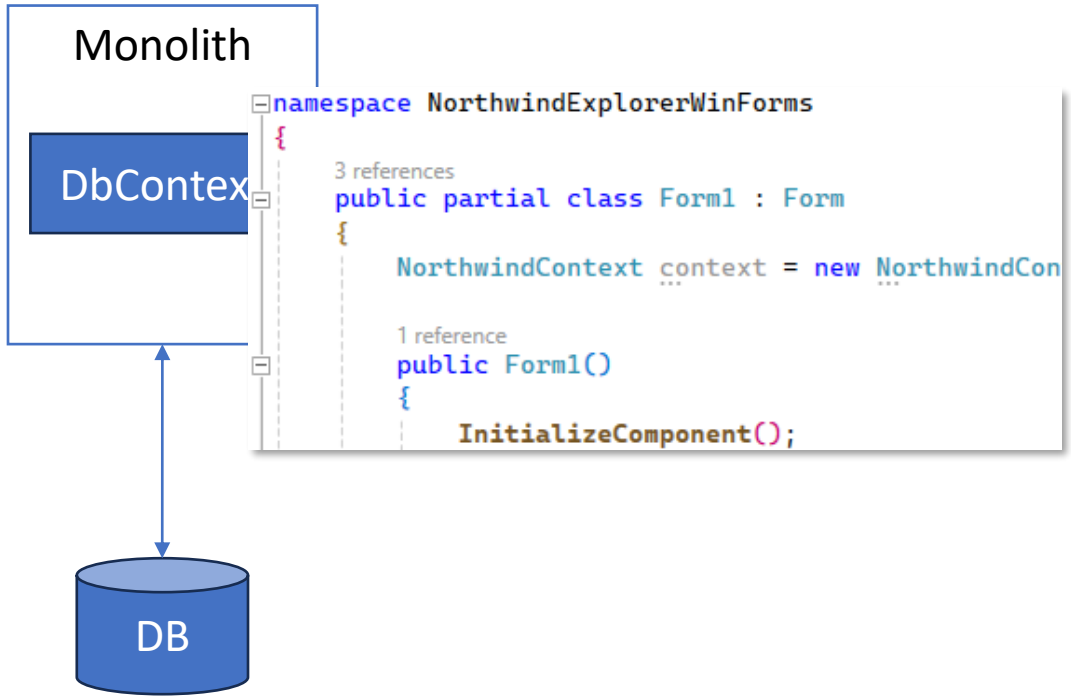
Fehlersuche und –vermeidung mit Entity Framework

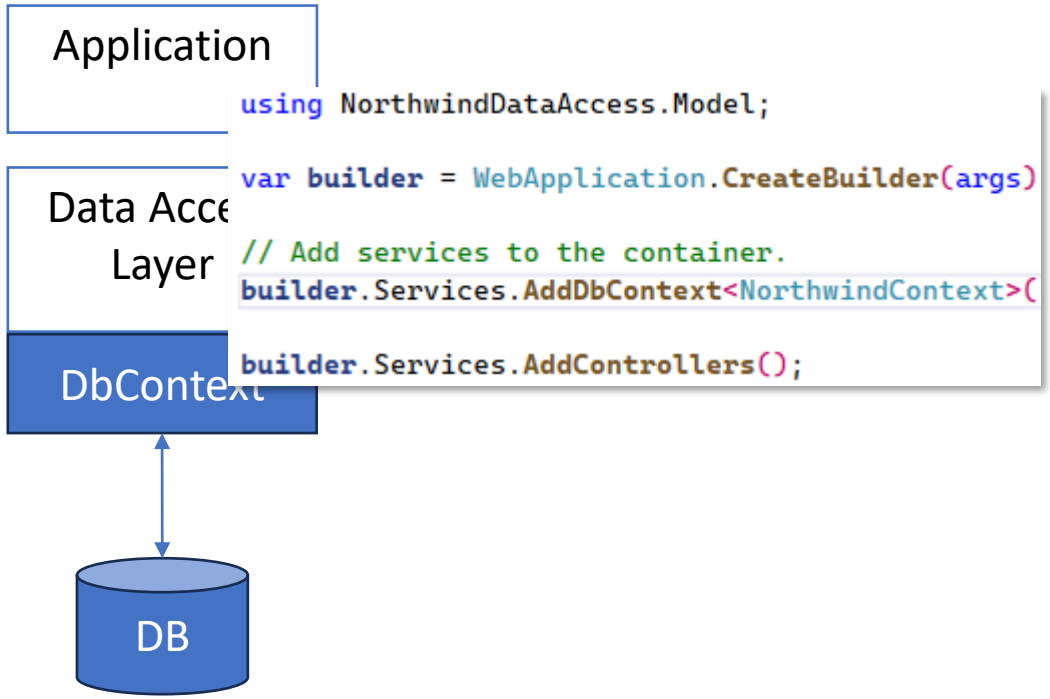
Speaker: Olaf Lischke

Speaker: Olaf Lischke

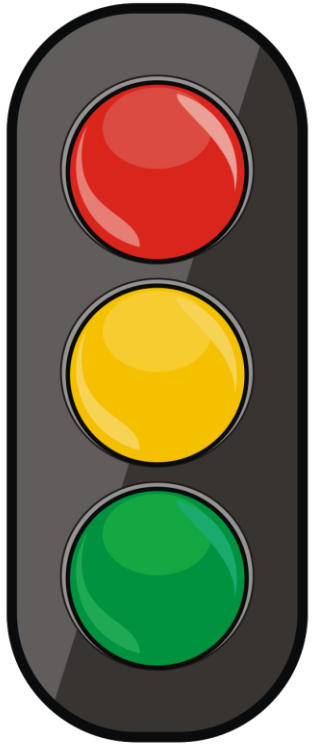
- macht .NET, seit es .NET gibt
- versucht, Projekte und Seminare zu kombinieren
- singt Tenor in Chören und Musikprojekten
- zockte schon auf dem ZX 81, heute ausschließlich auf PC
- fotografiert, seit er eine Kamera halten kann
- fliegt, wenn Wetter und Zeit es zulassen (TMG/SEP)

Entity Framework – Real-World Architekturen





Datenoperationen mit EF Core



Komplexe Datenstrukturen schreiben

Unverständliche Exceptions?

Einfache Daten schreiben

Konfiguration? FluentApi?

Einfache Daten abfragen
Komplexe Datenstrukturen abfragen

LINQ?

Fehlersuche

Refresher:
DbContext und sein ChangeTracker – Teil I

- Tracking-Verhalten DbContext

  `QueryTrackingBehavior`

  `.TrackAll` (Default)

  `.NoTracking`

  `.NoTrackingWithIdentityResolution`

  konfigurierbar auf Instanz-Ebene

- Tracking-Verhalten einer Query

  `IQueryable` Extension-Methods

  `.AsTracking()`

  `.AsNoTracking()`

  `.AsNoTrackingWithIdentityResolution()`

  verwendbar an jedem IQueryable

```
context.ChangeTracker.QueryTrackingBehavior
    = QueryTrackingBehavior.NoTracking;
```

```
private NorthwindContext GetContext(string connectionString)
{
    DbContextOptions<NorthwindContext> options = (new DbContextOptionsBuilder<Nor
        .UseSqlServer(connectionString)
        .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking)
        .Options;

    return new NorthwindContext(options);
```

Refresher:

DbContext und sein ChangeTracker – Teil II

- Zustand der Entitäten (Tracking)

  `EntityState`-Enum:
  - `Detached`
  - `Added`
  - `Deleted`
  - `Modified`
  - `Unchanged`

- Handling von Entitäten
  - `DbContext/DbSet-Methoden`

    `.Add(Entity)`
    `.Attach(Entity)`
    `.Delete(Entity)`
    `.Update(Entity)`

  → schreiben alle **NICHT** in die DB
  → regeln State der gg. Entity und abhängiger Entities!

```
Customer alfki = context.Customers.Find("ALFKI");
var state = context.Entry(alfki).State;
```

- Änderungen persistieren (speichern)

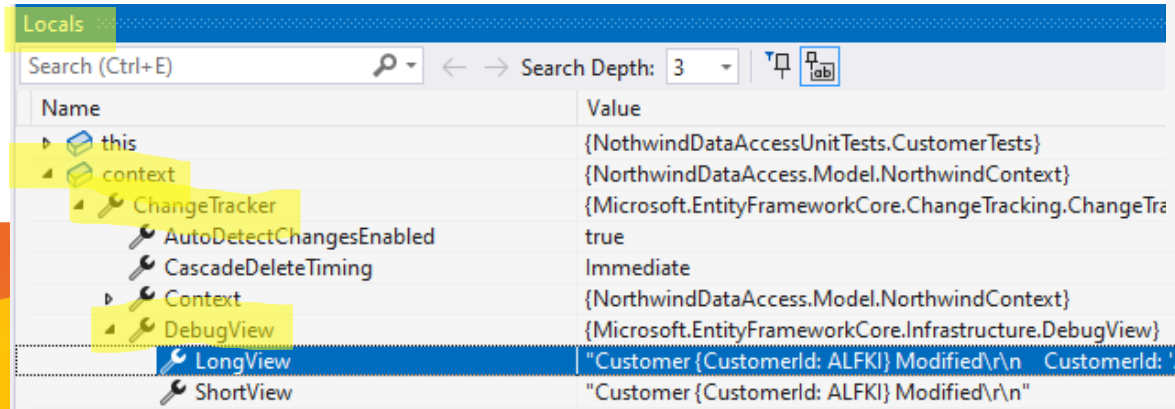
  `DbContext.SaveChanges()`
  `DbContext.SaveChangesAsync()`

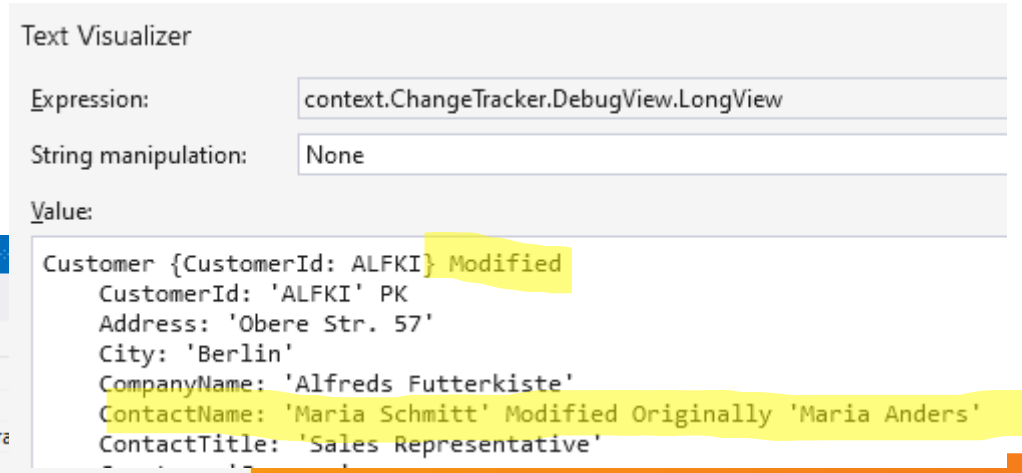
Datenoperationen debuggen

- Direktfenster/Immediate Window
  - Zustände abfragen mit ?
  - Methoden direkt aufrufen

```
Immediate Window
? alfki.ContactName
"Maria Anders"
? context.Entry(alfki).State
Unchanged
? context.Entry(alfki).State
Modified
context.ChangeTracker.DetectChanges()
Expression has been evaluated and has no value
```

- DbContext.ChangeTracker.DebugView
  - Zustand der Entitäten ansehen
  - **Tipp:** DetectChanges()!





Fehlervermeidung

Abstraktion des Datenzugriffs

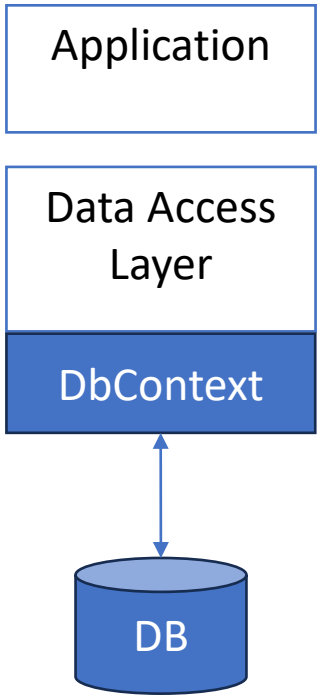
- Übliche CRUD-Operationen für Entitäten:
  - GetById
  - GetAll
  - Add
  - Remove
  - Modify

- Objektorientierung:
  - Kapselung
  - Wieder-/Weiterverwendbarkeit
  - „Separation of Concerns“
  - „Single Point of Responsibilty“

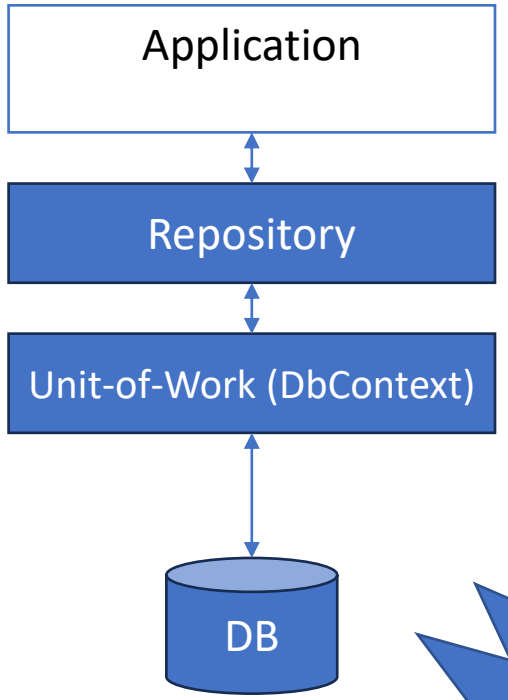
- Testbarkeit?

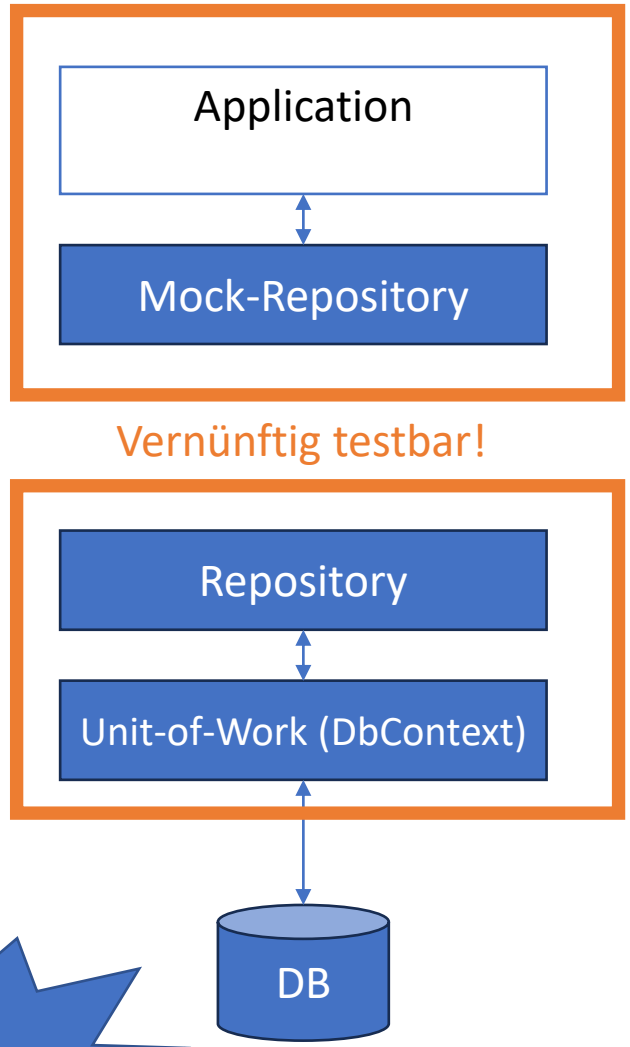
Repository Pattern Architektur

Ohne Repository



Mit Repository





Demo

Repository

- Standardisierung der CRUD-Methoden pro Entität
  - `Get(id)/GetAll()/Find()`
  - `Add()/Update()/Remove()`
- „Single point of responsibility“
  - Genau eine Stelle im Code, die die Funktionalität bereitstellt
- Generisch implementierbar

```
public interface IRepository<T> where T : class
{
    1 reference | Olaf Lischke, 2 hours ago | 1 author, 1 change
    T? GetById(int id);
    1 reference | Olaf Lischke, 2 hours ago | 1 author, 1 change
    IEnumerable<T>? GetAll();
    1 reference | Olaf Lischke, 2 hours ago | 1 author, 1 change
    IEnumerable<T>? FindAll(Expression<Func<T, bool>> predicate);
    2 references | 1/1 passing | Olaf Lischke, 2 hours ago | 1 author, 1 change
    void Add(T entity);
    1 reference | Olaf Lischke, 2 hours ago | 1 author, 1 change
    void AddRange(IEnumerable<T> entities);
    2 references | 1/1 passing | Olaf Lischke, 2 hours ago | 1 author, 1 change
    void Update(T entity);
    1 reference | Olaf Lischke, 2 hours ago | 1 author, 1 change
    void Remove(T entity);
```

- Vorteile:
  - Einheitliche Architektur
  - Testbarkeit
  - Wartbarkeit
  - Erweiterbarkeit
- Nachteil:
  - Zusätzlicher Abstraktionsschicht
  - Einschränkungen bei Nutzung ChangeTracker

Unit of Work

Martin Fowler:
"...everything one does during a business transaction which can affect the database."

- bündelt die DB-Zugriffe
- „kennt" alle Entitäten und deren Abhängigkeiten (Repositories!)
- verwaltet den DbContext
- löst `SaveChanges()` aus (einzige Stelle im Code!)
- Concurrency-Handling
- Exception-Handling

- Vorteile:
  - Kapselung des DB-Zugriffs
  - Datenkonsistenz
  - Transaktionssicherheit
- Nachteile
  - Weitere Abstraktionsschicht

```
public interface IUnitOfWork : IDisposable
{
    5 references | 2/2 passing | Olaf Lischke, 3 hours ago | 1 author, 1 change
    ICustomerRepository Customers { get; }
    3 references | 1/1 passing | Olaf Lischke, 3 hours ago | 1 author, 1 change
    IOrderRepository Orders { get; }
    2 references | Olaf Lischke, 3 hours ago | 1 author, 1 change
    IOrderDetailRepository OrderDetails { get; }
    3 references | 1/1 passing | Olaf Lischke, 3 hours ago | 1 author, 1 change
    IProductRepository Products { get; }

    3 references | 2/2 passing | Olaf Lischke, 3 hours ago | 1 author, 1 change
    int Complete();
```

Vielen Dank!

Slides und Code-Samples auf
https://github.com/olaflischke/basta-spring-2024

Für spätere Fragen:
olaf.lischke@lischke-edv.de