Übersicht über Leistungsaspekte
Reine Datenbankleistung

- Bei relationalen Datenbanken übersetzt EF die LINQ-Abfragen der Anwendung in SQL-Anweisungen, die von der Datenbank ausgeführt werden; diese SQL-Anweisungen selbst können mehr oder weniger effizient laufen.
- Beispiele:
  - Indizierung

```
modelBuilder.Entity<Mitarbeiter>()
            .HasIndex(u => u.Email)
            .IsUnique();
```

  - Umschreiben von LINQ-Abfragen kann EF dazu bringen, eine bessere SQL-Abfrage zu generieren
  - Funktionen oder Gruppierungsabfragen sollten auf dem Datenbankserver definiert werden

Übersicht über Leistungsaspekte - Netzwerk

- Datenübertragung im Netzwerk
  - Wie bei jedem Netzwerksystem ist es wichtig, die Datenmenge zu begrenzen, die über die Leitung hin- und hergeht. Dazu gehört, dass nur Daten gesendet und geladet werden, die tatsächlich benötigt werden. Dazu gehört auch, dass der so genannte "kartesischen Explosionseffekt" beim Laden zusammengehöriger Einheiten vermieden wird. (Kreuzprodukt!)
- Netzwerk-Roundtrips
  - Abgesehen von der Datenmenge, die hin- und hergeschickt wird, sind auch die Netzwerk-Roundtrips von Bedeutung, da die Zeit, die eine Abfrage in der Datenbank benötigt, durch die Zeit, die Pakete zwischen Ihrer Anwendung und Ihrer Datenbank hin- und herreisen, in den Schatten gestellt werden kann. Je weiter der Datenbankserver entfernt ist, desto höher ist die Latenzzeit und desto teurer ist jeder Roundtrip. Mit dem Aufkommen der Cloud befinden sich die Anwendungen immer weiter von der Datenbank entfernt, und "geschwätzige" Anwendungen, die zu viele Roundtrips durchführen, leiden unter Leistungseinbußen. Daher ist es wichtig, genau zu wissen, wann die Anwendung die Datenbank kontaktiert, wie viele Roundtrips sie durchführt und ob diese Zahl minimiert werden kann.

Übersicht über Leistungsaspekte
EF-Laufzeit-Overhead

- Schließlich verursacht EF selbst einen gewissen Laufzeit-Overhead bei Datenbankoperationen: EF muss die Abfragen von LINQ zu SQL kompilieren (obwohl das normalerweise nur einmal gemacht werden sollte), die Änderungsverfolgung verursacht einen gewissen Overhead (kann aber deaktiviert werden), usw. In der Praxis dürfte der EF-Overhead für reale Anwendungen in den meisten Fällen vernachlässigbar sein, da die Ausführungszeit der Abfrage in der Datenbank und die Netzwerklatenz die Gesamtzeit dominieren.

Empfehlungen für Abfragen

- Untersuchung des generierten SQL-Statements direkt in der Datenbank (SQl Server: SQL Profiler, Ausführungsplan → Kostenanalyse)
- Verwendung von Select, um nur die benötigten Spalten zu laden
- Verwendung von Paging (take, skip) und/oder Filtern von Suchvorgängen, um die Anzahl der zu ladenden Zeilen zu reduzieren
- Die Verwendung von Lazy Loading beeinträchtigt die Datenbankleistung
- Immer die AsNoTracking-Methode zu schreibgeschützten Abfragen hinzufügen

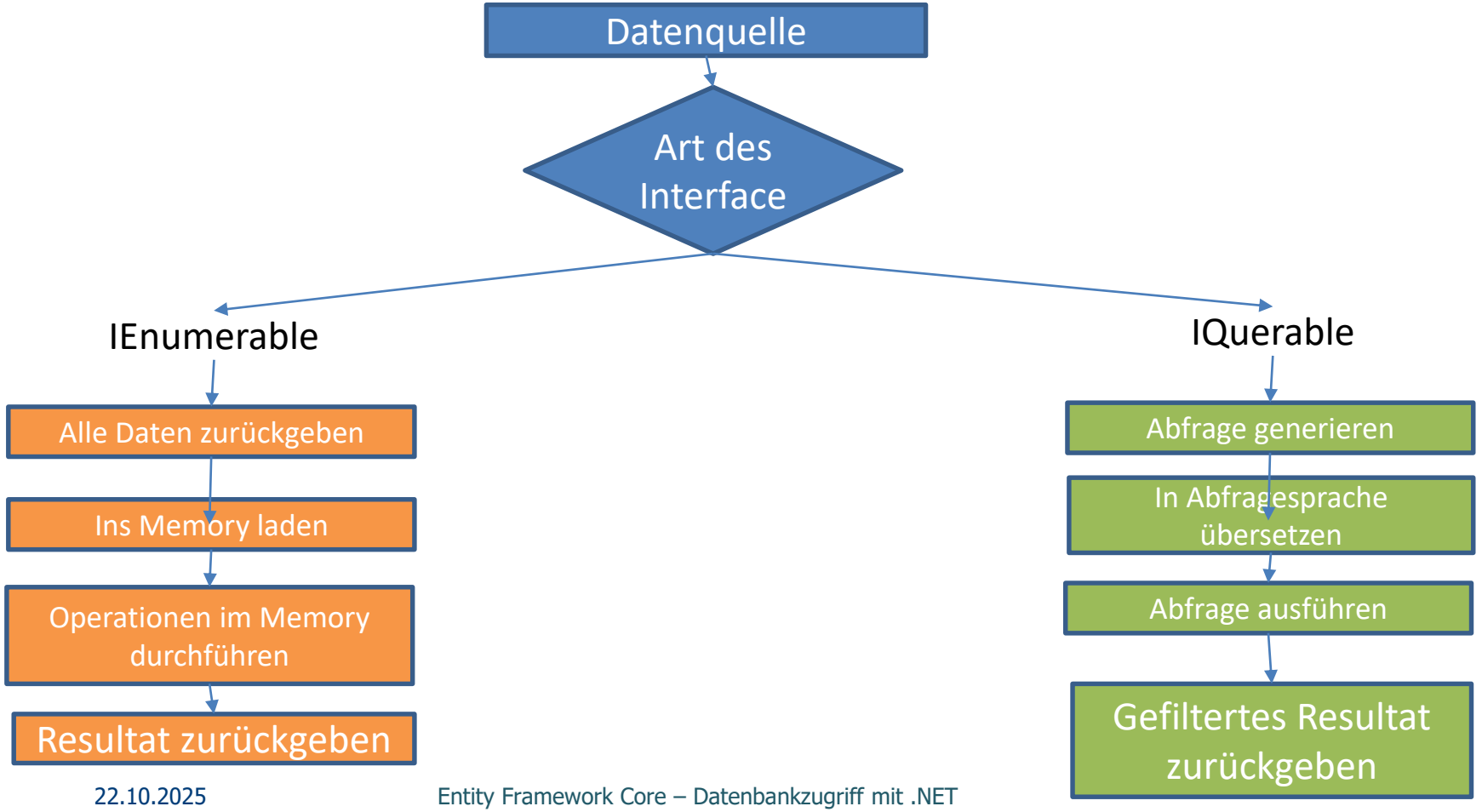
```
var liste = dbctx.MitarbeiterListe
   .Include(m => m.Abteilung)
   .AsNoTracking()
   .ToList();
```

- Verwendung der asynchronen Version von EF Core-Befehlen zur Verbesserung der Skalierbarkeit
- Sicherstellen, dass der Code für den Datenbankzugriff isoliert/entkoppelt ist, damit er für die Leistungsoptimierung bereit ist

BEISPIELE ZUR ANALYSE VON ABFRAGEN MITTELS BENCHMARKING

Der Unterschied von IEnumerable und IQuerable

- Es ist ein Unterschied, der sich vor allem bei großen Datenmengen zeigt, ob die Daten im lokalen Memory oder auf dem Datenbankserver bearbeitet werden.



Der Unterschied von IEnumerable und IQuerable

```
IEnumerable<Mitarbeiter> mitListe = dbctx.MitarbeiterListe
        .Include(m => m.Abteilung)
        .AsEnumerable();
    mitListe = mitListe.Where(m => m.Geburtsdatum.Value.Year>1980);
    foreach (var m in mit)
    {
        Display($"{m.Abteilung.Abteilungsbezeichnung}: {m.Nachname}, {m.Vorname}");
    }


IQueryable<Mitarbeiter> mitListe = dbctx.MitarbeiterListe
    .Include(m => m.Abteilung)
    .AsQueryable();
mitListe = mitListe.Where(m => m.Geburtsdatum.Value.Year > 1980);
foreach (var m in mitListe)
{
    Display($"{m.Abteilung.Abteilungsbezeichnung}: {m.Nachname}, {m.Vorname}");
}
```

AsNoTracking

- Wenn die Abfrage nur das Ziel hat, Daten für die Anzeige, aber nicht für Insert, Update oder Delete-Operationen, bereitzustellen, kann man sie mit .AsNoTracking versehen.
- Dadurch verringert sich der Speicherbedarf durchaus merklich.

Das N+1 - Problem

- Ist es sinnvoll, bei JOIN-Problemen zunächst einzelne Abfragen durchzuführen und die Ergebnisse anschließend zu filtern oder gleich mit einem INCLUDE-Befehl zu arbeiten?

```
//LoadAbteilung WithInclude

var mitListe = dbctx.MitarbeiterListe
    .Include(m => m.Abteilung);
```

```
//LoadAbteilungExtra

var mitListe = dbctx.MitarbeiterListe;
var abtListe = dbctx.AbteilungListe;
foreach (var item in abtListe)
{
   List<Mitarbeiter> lokaleMitListe = mitListe
      .Where(m => m.AbteilungId == item.AbteilungId);
}
```

- Der Benchmarktest ergibt für das Beispiel:

Method	Mean	Error	StdDev	Gen0	Allocated
LoadAbteilungWithInclude	1.208 us	0.7714 us	0.0423 us	0.1698	1.05 KB
LoadAbteilungExtra	74.639 us	23.9246 us	1.3114 us	1.9531	13.09 KB

Interceptors

- Mit Interceptors können wir benutzerdefiniertes Verhalten in verschiedenen Phasen der EF Core-Operationspipeline einfügen und erhalten so eine bessere Kontrolle über die Dateninteraktionsprozesse. Darüber hinaus können wir durch die Verwendung von Interceptors unsere Datenbankoperationen feinabstimmen, Geschäftsregeln durchsetzen, die Datenintegrität sicherstellen und vieles mehr.
- EF Core Interceptors sind Klassen, die die Schnittstelle Microsoft.EntityFrameworkCore.Diagnostics.IInterceptor implementieren.
  - Im Wesentlichen ruft das Framework Instanzen dieser Klassen in verschiedenen Phasen unserer Dateninteraktionen auf, je nach deren Typ. Wir verwenden Interceptors, um Operationen zu überwachen, zu protokollieren, zu ändern oder abzubrechen, bevor oder nachdem EF Core sie ausführt.

Interceptor	Beschreibung
SaveChangesInterceptor	Hooks into the SaveChanges process
DbCommandInterceptor	Intercepts and modifies database commands
TransactionInterceptor	Provides control over transaction operations
DbConnectionInterceptor	Monitors and modifies database connection events

Beispiel: Logging auf dem Level Information

```
optionsBuilder
  .UseSqlServer(connectionString)
  .LogTo(Console.WriteLine,LogLevel.Information);
```

- Hier wird eine asynchrone Abfrage getestet:

```
mitListe = await (dbctx.MitarbeiterListe
    .TagWith($"Abfrage Mitarbeiter in Abteilung {abt.Abteilungsbezeichnung} ")
    .Where(m => m.AbteilungId==abtId)
    .ToListAsync());
```

- Logging-Information:

```
info: 21.07.2024 12:59:16.326 RelationalEventId.CommandExecuted[20101]
(Microsoft.EntityFrameworkCore.Database.Command)
    Executed DbCommand (6ms) [Parameters=[@__abtId_0='?' (DbType = Int32)], CommandType='Text', CommandTimeout='30']
    -- Abfrage Mitarbeiter in Abteilung Produktion

    SELECT [m].[mitarbeiterId], [m].[abteilungId], [m].[datum_austritt], [m].[datum_eintritt], [m].[durchwahl], [m].[email],
[m].[extern], [m].[funktionId], [m].[geburtsdatum], [m].[geschlecht], [m].[nachname], [m].[NiederlassungId], [m].[prsnr],
[m].[RowVersion], [m].[vorname]
    FROM [mitarbeiter] AS [m]
    WHERE [m].[abteilungId] = @__abtId_0
```

Einige Hinweise:

Bei der Verwendung von Entity Framework Core (EF Core) in Anwendungen mit hohem Datenverkehr ist die Leistung von größter Bedeutung. Hier sind einige Überlegungen und Strategien zur Optimierung von EF Core für hohen Datenverkehr:

- Verwenden Sie AsNoTracking:
  - Bei der Abfrage von Daten zu reinen Lesezwecken sollte .AsNoTracking() verwendet werden, was den Overhead der Änderungsverfolgung erheblich reduzieren kann, insbesondere in Szenarien, in denen Daten nicht verändert werden.
- Effizienter Abfrageentwurf:
  - Verwenden Sie die Projektion (z.B. .Select()), um nur die benötigten Spalten anstelle ganzer Entitäten abzurufen. Verwenden Sie Include() mit Bedacht, um das Laden unnötiger Bezugsdaten zu vermeiden, die teuer sein können. Filtern Sie Abfragen so früh wie möglich, um die Größe des zu verarbeitenden Datensatzes zu reduzieren.
- Batching-Operationen:
  - Nutzen Sie Methoden wie SaveChanges(true) oder ExecuteSqlRaw(), um mehrere Operationen in einem einzigen Datenbankdurchlauf zu bündeln, wenn möglich.
- Begrenzen Sie die Verwendung von Lazy Loading:
  - Lazy Loading kann zu mehrfachen Datenbankabfragen führen, was die Leistung beeinträchtigen kann. Bevorzugen Sie Eager Loading mit Include() oder Explicit Loading.

Einige Hinweise:

- Vermeidung des N+1-Abfrageproblems:
  - Achten Sie auf Situationen, in denen ein N+1-Problem auftreten könnte, und verwenden Sie eager loading oder explicit loading, um es zu entschärfen.
- Kompilierte Abfragen verwenden:
  - Verwenden Sie für Abfragen, die häufig ausgeführt werden, kompilierte Abfragen, um die Leistung zu verbessern, indem Sie die Kosten für die Abfragekompilierung reduzieren.
- Optimieren Sie die Verwendung von Datenbankkontexten:
  - Verwenden Sie in Webanwendungen eine DbContext-Instanz pro Anfrage, um eine effiziente Ressourcennutzung zu gewährleisten und Speicherlecks zu vermeiden.
  - Vermeiden Sie die gemeinsame Nutzung von DbContext-Instanzen in verschiedenen Threads.
- Verbindungs-Pooling:
  - Stellen Sie sicher, dass das Connection Pooling für eine effiziente Wiederverwendung von Datenbankverbindungen aktiviert ist.
- Indizierung:
  - Stellen Sie sicher, dass Ihr Datenbankschema gut indiziert ist, insbesondere für Spalten, die häufig abgefragt oder in Verknüpfungsbedingungen verwendet werden.

Einige HInweise:

- Caching:
  - Implementieren Sie gegebenenfalls Caching-Strategien, um die Datenbanklast zu verringern, z. B. durch Verwendung von Memory Cache oder Distributed Cache für wiederholte Abfragen.
- Protokollierung und Profiling:
  - Verwenden Sie Protokollierungs- und Profiling-Tools, um langsame Abfragen oder Engpässe zu ermitteln und sie bei Bedarf zu optimieren.
- Gleichzeitigkeitsmanagement:
  - Implementieren Sie eine optimistische Gleichzeitigkeitssteuerung, um Umgebungen mit hohem Datenverkehr zu bewältigen, in denen Schreibkonflikte häufig vorkommen können.
- Datenbank-Design:
  - Sorgen Sie für einen gut durchdachten Entwurf des Datenbankschemas, um hohe Gleichzeitigkeit und große Datenmengen effizient zu handhaben.
- Überwachung und Skalierung:
  - Überwachen Sie kontinuierlich die Anwendungsleistung und bereiten Sie sich darauf vor, Ressourcen, wie z. B. Datenbankserver oder Anwendungsinstanzen, bei Bedarf zu skalieren, um eine erhöhte Last zu bewältigen.