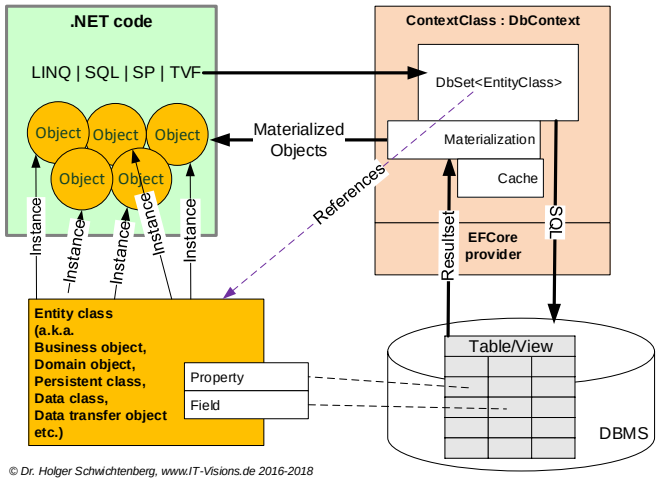
Spickzettel („Cheat Sheet“): Entity Framework Core: Client-API

Autor: Dr. Holger Schwichtenberg (www.IT-Visions.de)

www.IT-Visions.de
Dr. Holger Schwichtenberg

Artefakte, Vorgehensweisen und Kontextarten



© Dr. Holger Schwichtenberg, www.IT-Visions.de 2016-2018

Häufig benötigte Namensräume

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.EntityFrameworkCore.Infrastructure;
using Microsoft.EntityFrameworkCore.ChangeTracking;
```

Grundkonstruktion

Kontext nach Gebrauch vernichten, da dieser einen Cache besitzt!

```
using (WWWingsContext ctx = new WWWingsContext())
{ … }
```

Alternativ:

```
WWWingsContext ctx = new WWWingsContext();
…
ctx.Dispose()
```

Ladeoperationen

Laden eines Datensatzes anhand des Primärschlüssels mit **Find()**

```
int no = 123;
Flight f = ctx.FlightSet.Find(no);
Console.WriteLine($"Flight #{f.FlightNo} from {f.Departure} to
{f.Destination} has {f.FreeSeats} free seats");
```

Laden aller Datensätze einer Tabelle

```
List<Flight> liste = ctx.FlightSet.ToList();
foreach (var f in liste) { Console.WriteLine("Flight #" + f.FlightNo + " …"); }
```

Selektives Laden mit LINQ

```
List<Flight> flightSet = (from f in ctx.FlightSet
                          where f.Departure == "Berlin" && f.FreeSeats > 0
                          && f.Date > DateTime.Now orderby f.Date
                          select new Flight  { FlightNo = f.FlightNo,
                                               FreeSeats = f.FreeSeats,
                                               Timestamp = f.Timestamp }
                          ).Skip(20).Take(5).ToList();
foreach (var f in liste) { Console.WriteLine("Flight #" + f.FlightNo + " …"); }
```

Datenbankview abfragen (ab v2.1)

Klasse anlegen, die dem View genau entspricht

```
public class DepartureStatView
 {
  public string Departure { get; set; }
  public int FlightCount { get; set; }
 }
```

In Kontextklasse dafür DbQuery<T> anlegen

```
public DbQuery<DepartureStatView> DepartureStatView { get; set; }
```

View abfragen wie Tabelle (Objekte sind aber "Detached!")

```
List<DepartureStatView> view = (from f in ctx.DepartureStatView
                                where f.FlightCount > 0
                                orderby f.FlightCount descending
                                select f).ToList();
foreach (var c in view)
{
   Console.WriteLine($"{c.FlightCount:000} Flights from {c.Departure}.");
}
```

Informationen über die Datenbank

```
Console.WriteLine(ctx.Database.IsSqlServer());
Console.WriteLine(ctx.Database.GetDbConnection().ConnectionString);
Console.WriteLine(ctx.Database.GetDbConnection().DataSource);
```

Timeout festlegen

```
ctx.Database.SetCommandTimeout(new TimeSpan(0,0,10)); // 10sec
```

CUD-Operationen mit EF Core-API

Objekt im RAM anlegen und persistieren

```
var f = new Flight();
f.FlightNo = 123456;
f.Departure = "Essen/Mülheim";
f.Destination = "Redmond";
f.Date = new DateTime(2019, 8, 1);
f.Seats = 320;
f.Pilot = ctx.PilotSet.FirstOrDefault();
f.Copilot = null; // no one assigned yet
ctx.FlightSet.Add(f);
int countChanges = ctx.SaveChanges();
Console.WriteLine("Number of saved changes: " + countChanges);
```

Objekt im RAM ändern und persistieren

```
Flight f = ctx.FlightSet.Find(123456);
f.FreeSeats--;
f.Date = f.Date.AddHours(2);
f.Memo = "Flight is delayed!";
int countChanges = ctx.SaveChanges();
Console.WriteLine("Number of saved changes: " + countChanges);
```

Objekt im RAM löschen und das Löschen persistieren

```
Flight f = ctx.FlightSet.Find(123456);
if (f != null) ctx.FlightSet.Remove(f);
int countChanges = ctx.SaveChanges();
Console.WriteLine("Number of saved changes: " + countChanges);
```

Transaktionen mit System.Transactions

SaveChanges() erzeugt immer automatisch eine Transaktion. Es ist möglich, mehrere SaveChanges()-Aufrufe zu einer Transaktion zusammenzufassen.

```
using System.Transactions;
…
var tso = TransactionScopeOption.Required;
var to = new TransactionOptions();
to.IsolationLevel = IsolationLevel.ReadCommitted;
using (var t = new TransactionScope(tso, to))
 {
    var f1 = ctx.FlightSet.Find(123);
    f1.FreeSeats--;
    var c1 = ctx.SaveChanges();
    Console.WriteLine("Number of saved changes: " + c1);
    var f2 = ctx.FlightSet.Find(256);
    f2.FreeSeats--;
    var c2 = ctx.SaveChanges();
    Console.WriteLine("Number of saved changes: " + c2);
    t.Complete(); // otherwill transaction will roll back
    Console.WriteLine("Transaction successful!");
 }
```

Konflikterkennung

```
var f = ctx.FlightSet.Find(123);
f.FreeSeats -= 2;
try    {
    var c = ctx.SaveChanges();
    Console.WriteLine("SaveChanges: Number of saved changes: " + c);
   }
catch (DbUpdateConcurrencyException ex)    {
    var dbValue =
       ctx.Entry(f).GetDatabaseValues().GetValue<short?>("FreeSeats");
    Console.WriteLine($"annother user has changed freeSeats: {dbValue}!");
  … }
```

Konfliktlösung

Werte des anderen Benutzers übernehmen

```
ctx.Entry(f).Reload();
```

oder: Werte des anderen Benutzers überschreiben

```
ctx.Entry(f).OriginalValues.SetValues(ctx.Entry(f).GetDatabaseValues());
int anzahl = ctx.SaveChanges();
```

Über den Autor

Dr. Holger Schwichtenberg gehört zu den bekanntesten Experten für Webtechniken und .NET in Deutschland. Er hat zahlreiche Fachbücher veröffentlicht und spricht regelmäßig auf Fachkonferenzen. Sie können ihn und seine Kollegen für Entwicklungsarbeiten, Schulungen, Beratungen und Coaching buchen.
E-Mail: **buero@IT-Visions.de**
Website: **www.IT-Visions.de**
Weblog: **www.dotnet-doktor.de**

Spickzettel („Cheat Sheet“): Entity Framework Core: Client-API

Autor: Dr. Holger Schwichtenberg (www.IT-Visions.de)

www.IT-Visions.de
Dr. Holger Schwichtenberg

SQL-Abfrage aufrufen, die Entitätstyp liefert

SQL-Abfrage, die Resultset liefert (Entitätstyp oder andere Klasse)

```
string departure = "Berlin"; string destination = "Paris";
int minSeats = 10;
List<Flight> flightSet = ctx.FlightSet.FromSql<Flight>("Select * from Flight
where departure = {0} and destination = {1} and Freeseats > {2}",
departure, destination, minSeats).ToList();
```

Ab EF Core 3.0 überall: FromSqlRaw() statt FromSql()

Stored Procedure aufrufen, die Entitätstyp liefert

Where-Bedingung wird im Client ausgeführt ☹

```
string departure = "Berlin";
List<Flight> flightSet = ctx.FlightSet.FromSql("EXEC GetFlightsFromSP {0}",
departure).Where(x => x.FreeSeats > 0).ToList();
```

Table Valued Function, die Entitätstyp liefert

Where-Bedingung wird im DBMS ausgeführt ☺

```
string departure = "Berlin";
List<Flight> flightSet = ctx.FlightSet.FromSql("select * from
GetFlightsFromTVF({0})", departure).Where(x => x.FreeSeats > 0).ToList();
```

SQL-Abfrage, die keinen Entitätstyp liefert (ab v2.1)

POCO-Klasse anlegen, die dem Resultset genau entspricht:

```
public class DepartureGroup
 {
  public string Departure { get; set; }
  public int FlightCount { get; set; }
  public Int16? MinFreeSeats { get; set; }
  public Int16? MaxFreeSeats { get; set; }
  public Int32? SumFreeSeats { get; set; }
  public Int32? AvgFreeSeats { get; set; }
 }
```

In Kontextklasse dafür DbQuery<T> anlegen:

```
public DbQuery<DepartureGroup> DepartureGroup { get; set; }
```

Ab EF Core 3.0: DbSet<T> statt DbQuery<T>

Im Client:

```
var sql = "SELECT Departure, COUNT(FlightNo) AS FlightCount,
Min(FreeSeats)AS MinFreeSeats, Max(FreeSeats) AS MaxFreeSeats,
Sum(FreeSeats) AS SumFreeSeats, Avg(FreeSeats) AS AvgFreeSeats FROM
Flight GROUP BY Departure";
List<DepartureGroup> groupSet = ctx.DepartureGroup.FromSql(sql).ToList();
   Console.WriteLine(groupSet.Count);
```

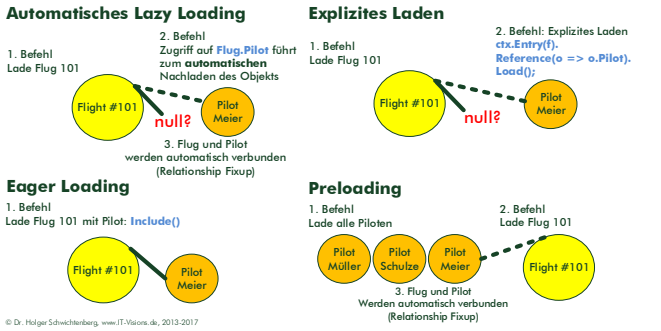
SQL-DML-Anweisung, die Daten löscht

```
int count = ctx.Database.ExecuteSqlCommand("Delete from Flight where
FlightNo > {0}", 10000);
Console.WriteLine("Number of deleted records: " + count);
```

Liste aller Instanzen einer Klasse im Kontext-Cache

```
LocalView<Flight> flightSet = ctx.FlightSet.Local;
foreach (var f in flightSet)
{
    Console.WriteLine("Flight #" + f.FlightNo + " from " + f.Departure);
}
```

Ladestrategien



Explizites Lazy Loading (ab v1.1): Zugriff auf Pilot und seine Detaildaten ist nur möglich mit expliziten Load()-Anweisungen für jede einzelne Ebene.

```
var flightSet = (from f in ctx.FlightSet where f.Departure == "Berlin" select f);
foreach (var f in flightSet.ToList())
{
 Console.WriteLine("Flight #" + f.FlightNo + " from " + f.Departure);
 ctx.Entry(f).Reference(x => x.Pilot).Load();
 ctx.Entry(f.Pilot).Reference(x => x.Detail).Load();
 Console.WriteLine("Pilot " + f.Pilot.FullName + " in " + f.Pilot.Detail.City);
}
```

Automatisches Lazy Loading (ab v2.1): Zugriff auf Pilot und seine Detaildaten ist möglich, führt aber jeweils zu einer eigenen Abfrage.

Zwei Vorrausetzungen:

a) Install-package Microsoft.EntityFrameworkCore.Proxies

b) In OnConfiguring():

```
builder.UseLazyLoadingProxies(true).UseSqlServer(…);
```

```
var flightSet = (from f in ctx.FlightSet where f.Departure == "Berlin" select f);
foreach (var f in flightSet.ToList())
{
 Console.WriteLine("Flight #" + f.FlightNo + " from " + f.Departure);
 Console.WriteLine("Pilot " + f.Pilot.FullName + " in " + f.Pilot.Detail.City);
}
```

Eager Loading mit Include(): Pilot und seine Detaildaten werden direkt zusammen mit den Flügen geladen

```
List<Flight> flightSet = (from f in ctx.FlightSet.Include(f=>f.Pilot.Detail) where
f.Departure == "Berlin" select f).ToList();
foreach (var f in flightSet)
{
 Console.WriteLine("Flight #" + f.FlightNo + " from " + f.Departure);
 Console.WriteLine("Pilot " + f.Pilot.FullName + " in " + f.Pilot.Detail.City);
}
```

Preloading: Alle Piloten und alle Details der Piloten werden vorab in den Cache geladen und werden beim Laden der Flüge automatisch mit den Flügen verbunden, auch wenn das Ergebnis des Vorladens gar keiner Variablen zugewiesen wird.

```
ctx.PilotSet.Include(p=>p.Detail).ToList();
List<Flight> flightSet = (from f in ctx.FlightSet.Include(f => f.Pilot.Detail)
where f.Departure == "Berlin" select f).ToList();
foreach (var f in flightSet)
{
 Console.WriteLine("Flight #" + f.FlightNo + " from " + f.Departure);
 Console.WriteLine("Pilot " + f.Pilot.FullName + " in " + f.Pilot.Detail.City);
}
```

Leistungsoptimierung durch NoTracking

Laden ohne Änderungsverfolgung ("Detached") ist deutlich schneller!

```
List<Flight> flightSet = (from f in ctx.FlightSet.AsNoTracking() select f).ToList();
var myFlight = flightSet.ElementAt(10);
Console.WriteLine("Object State: " + ctx.Entry(myFlight).State); // = Detached
```

Änderungsverfolgung für alle folgenden Abfragen ausschalten

```
ctx.ChangeTracker.QueryTrackingBehavior =
QueryTrackingBehavior.NoTracking;
List<Flight> flightSet = (from f in ctx.FlightSet select f).ToList();
```

Eine Abfrage mit Änderungsverfolgung trotz der Deaktivierung

```
List<Flight> flightSet = (from f in ctx.FlightSet.AsTracking() select f).ToList();
```

Nachträgliches Attach() erlaubt bei Detached-Objekten dann dennoch eine Änderungsverfolgung

```
List<Flight> flightSet = (from f in ctx.FlightSet.AsNoTracking() select f).ToList();
var myFlight = flightSet.ElementAt(10);
ctx.FlightSet.Attach(myFlight);
myFlight.FreeSeats--;
int c = ctx.SaveChanges();
Console.WriteLine("Number of saved changes: " + c);
```

Protokollierung aller SQL-Befehle in die Konsole

Zwei Vorrausetzungen:

a) Install-package Microsoft.Extensions.Logging.Console

b) In OnConfiguring():

```
builder.UseLoggerFactory(MyLoggerFactory).UseSqlServer(…);
```

Wichtig: Nur eine globale LoggerFactory! Nicht für jede Kontextinstanz die LoggerFactory neu erzeugen (Memory Leak!)

```
public static readonly LoggerFactory MyLoggerFactory
= new LoggerFactory(new[] {
      new ConsoleLoggerProvider((category, level)
        => category == DbLoggerCategory.Database.Command.Name
          && level == LogLevel.Information, true) });
```