Entity Framework 6.0

JAN STAECKER FÜR PASS, FEBRUAR 2015

Übersicht

1. Was ist Entity Framework 6.0
2. DataContext
3. Lazy Loading
4. Verwenden Async, Tasks, Threads
5. Transactions

1.
Was ist das Entity Framework?

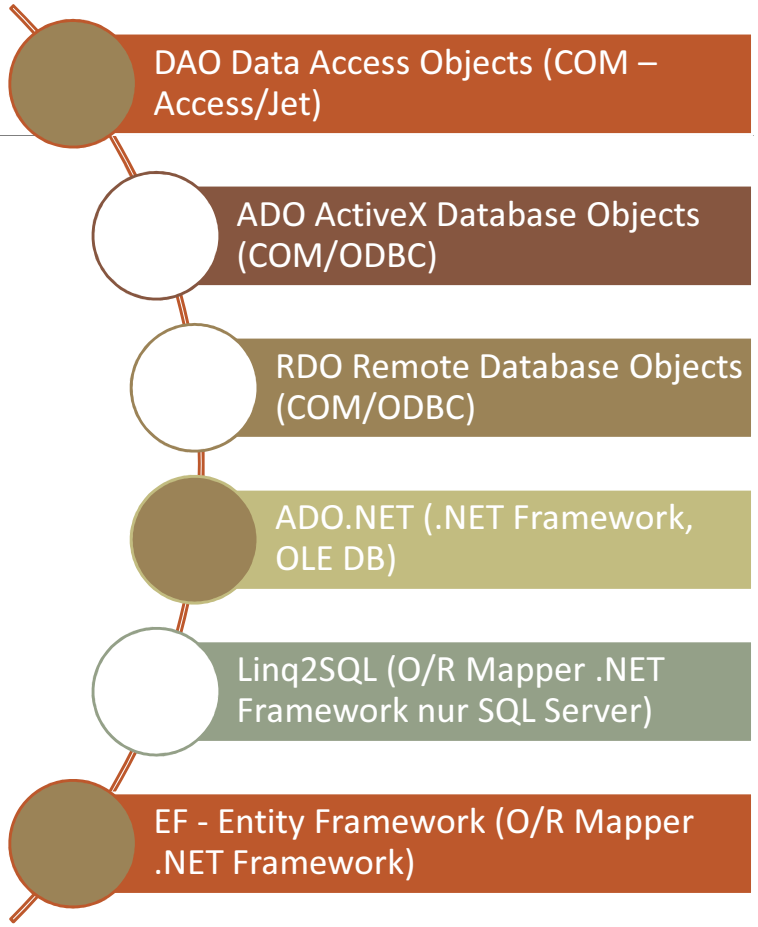
Entity Framework

Eine weitere Datenschicht, die den Entwickler deutlich „einfacher“ an die Datenbank setzt, als alle anderen Datenschichten zuvor.

Im Laufe der Zeit hat Microsoft uns viele Datenschichten beschert.

Das EF ist eigentlich das ADO.NET Entity Framework.

Aktuell ist DAO immer noch nötig, da man VBA für Access programmiert.



Verbesserungen des EF von 4 zu 6

Es gibt eine Menge von Verbesserungen im EF.

Also kein Grund auf einem alten EF sitzen zu bleiben.

Nur beim alten MVC Projekt muss man MVC updaten, was kein einfaches Unterfangen ist.

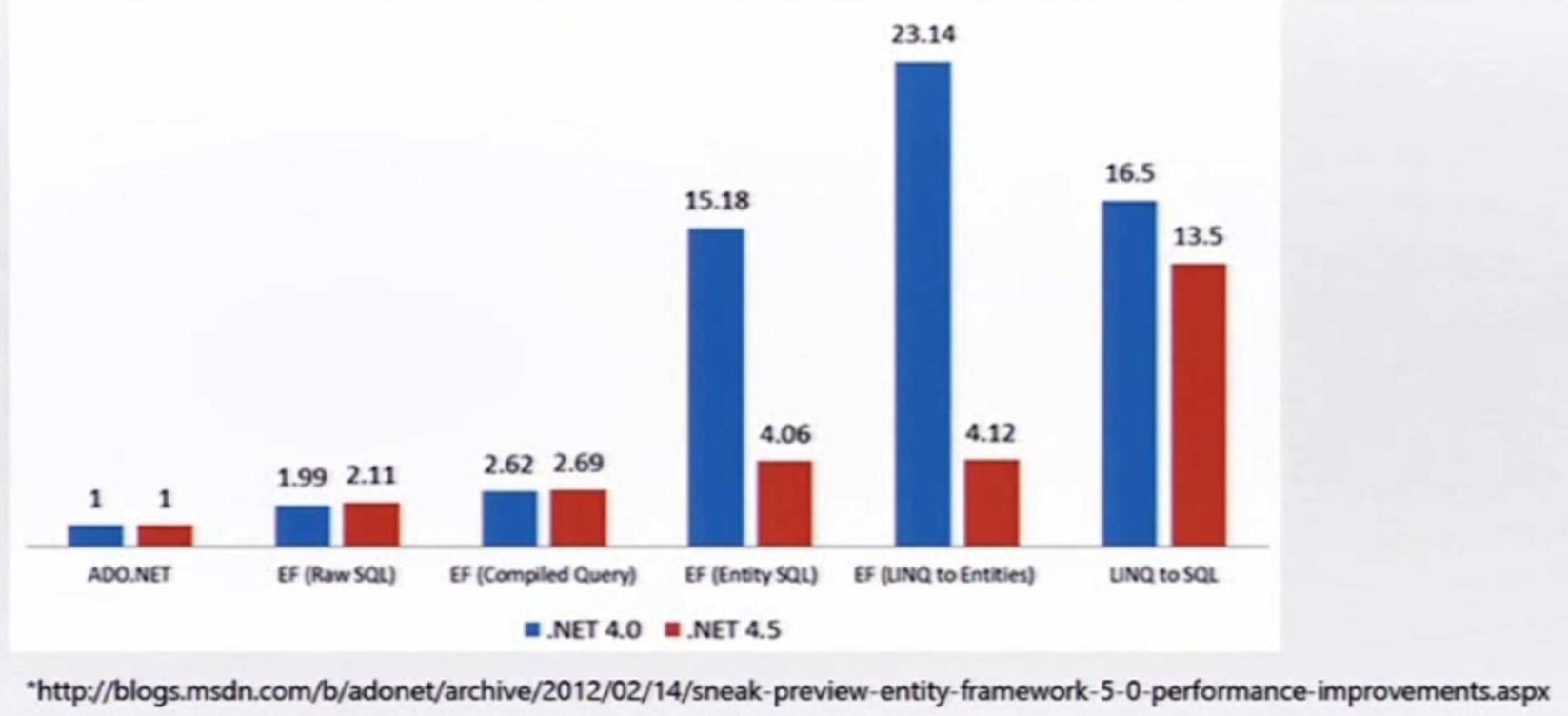
Historie des Entity Frameworks

Mit Visual Studio 2012 und .NET Framework 4.0 kann man einiges aus EF bis 4.3 nutzen.

Aktuell würde ich VS 2013 und .NET Framework > 4.5/4.5.1 empfehlen.

Screenshot aus Vortrag Evolution, Frankfurt 2012:
http://www.microsoft.com/germany/msdn/events/vs-evolution-2012/default.aspx

Performance Vergleich durch Verwendung von .NET 4.5 statt 4.0



*http://blogs.msdn.com/b/adonet/archive/2012/02/14/sneak-preview-entity-framework-5-0-performance-improvements.aspx

Verbesserungen im EF Modeldesigner

Jede Änderung im Datenmodell meines alten LinQ2SQL Projekt dauert ca. 5 Minuten.

Im Test bei des EF-Teams: The model contains 1005 entity sets and 4227 association sets:

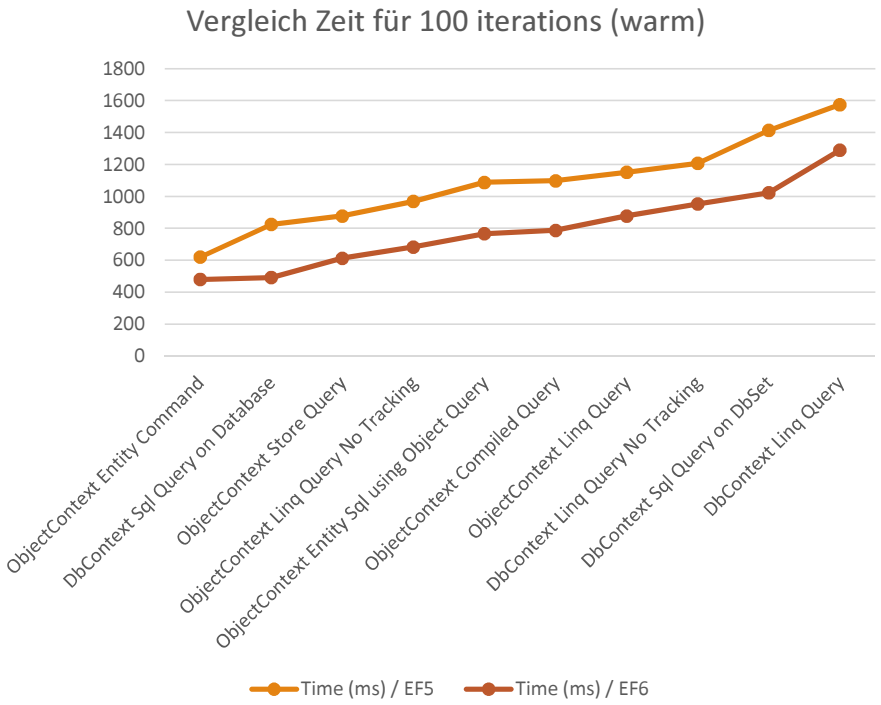
Visual Studio 2010.	SSDL Generation: 2 hr 27 min
Entity Framework 4.	Mapping Generation: 1 second
	CSDL Generation: 1 second
	ObjectLayer Generation: 1 second
	View Generation: 2 h 14min

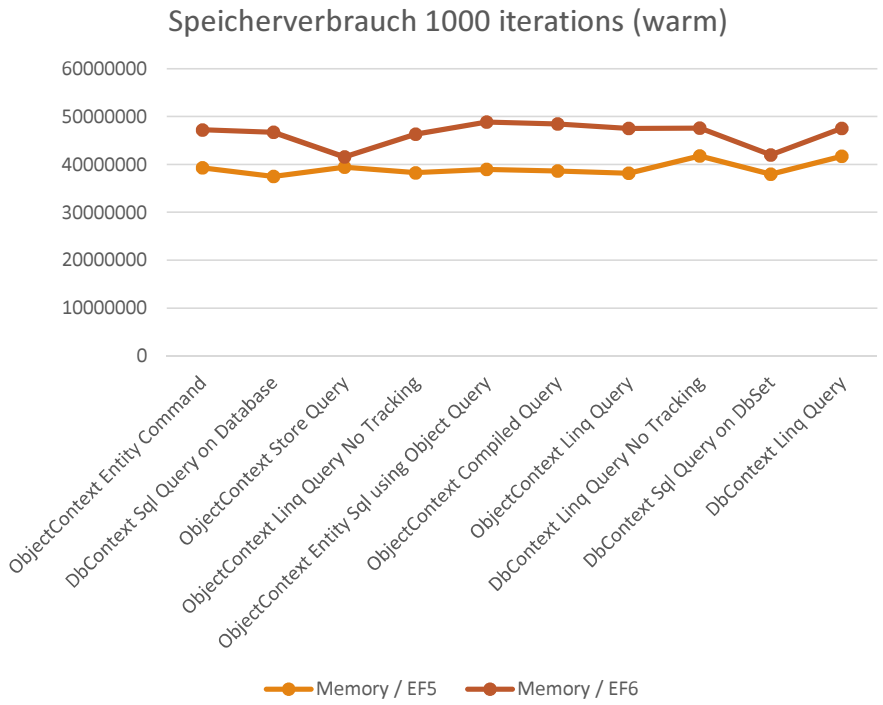
Gerade SSDL (der store-layer) wird im Wesentlichen auf dem SQL Server generiert

Visual Studio 2013.	SSDL Generation: 1 second
Entity Framework 6.	Mapping Generation: 1 second
	CSDL Generation: 1 second
	ObjectLayer Generation: 1 second
	View Generation: 28 seconds.

Quelle: https://msdn.microsoft.com/en-us/data/hh949853.aspx

Vergleich EF 5 zu EF 6





Quelle: https://msdn.microsoft.com/en-us/data/hh949853.aspx

EF 6.1 kennt nun weitere SQL Server Datentypen wie „Entity.Spatial.DbGeometry“

Aus meiner Tabelle der Bayrischen Regierungsbezirke liegen die Umrisse als Multipolygon vor.

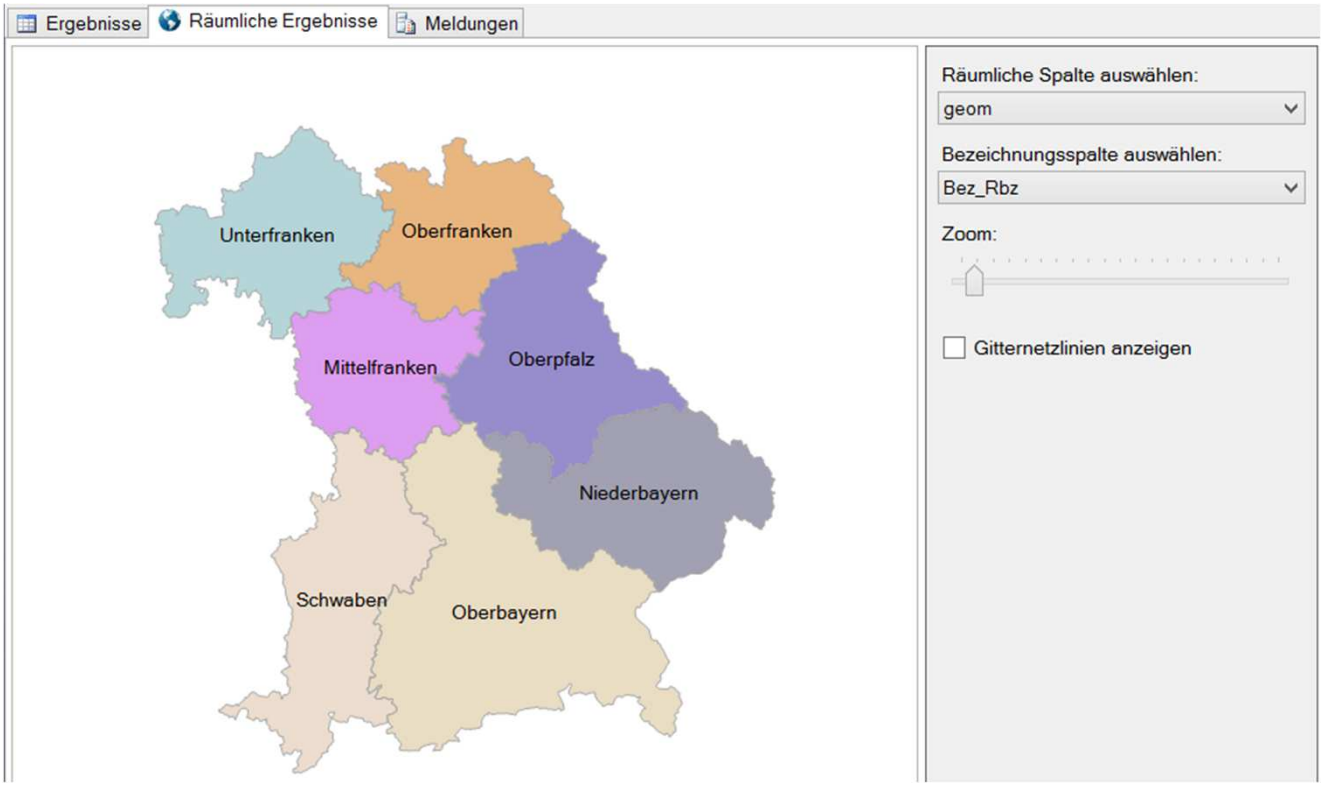
Beim Laden bekomme ich den Datentyp: DbGeometry

```
var bez = new Model.BavEntities().Bezirk.ToList();
var geo = bez[0].geom;
```

SQL Server Management Studio stellt die Bezirke in Geokoordinaten schön dar.

```
SELECT [ID],
[Bez_Rbz] ,
[geom]

FROM

[Bav].[dbo].[Bezirk]
```

wird zu:



Quelle: Geo-Daten des Bayerischen Vermessungsamts / www.lfpbuender.org

Der Datentyp HierarchyID wird noch nicht unterstützt

Somit gibt es einige Datenbanken, welche nicht komplett per Entity Framework genutzt werden können. Insbesondere die FileTables, welche die HierarchyID nutzen.

Es gibt eine NuGet Paket, welches HierarchyID bereits nutzt.

Quelle: https://www.nuget.org/packages/EntityFrameworkWithHierarchyId

```
Fehler 6005: Der Datentyp 'hierarchyid' wird derzeit für die Entity Framework-Zielversion nicht unterstützt. Die Spalte 'HierarchyID' in der Tabelle 'Bav.dbo.TestHierarchy' wurde ausgeschlossen.
```

Übersicht

A) Beispiel mit WinForms

B) Diagnose, wenn Speichern fehl schlägt

C) Log des DataContext. Das ist neu in EF 6.0

D) DataAnnotations in MVC

A. Beispiel

NuGet Paket: Entity Framework 6.x zum Projekt hinzufügen (Dafür ist Internetzugang wichtig)

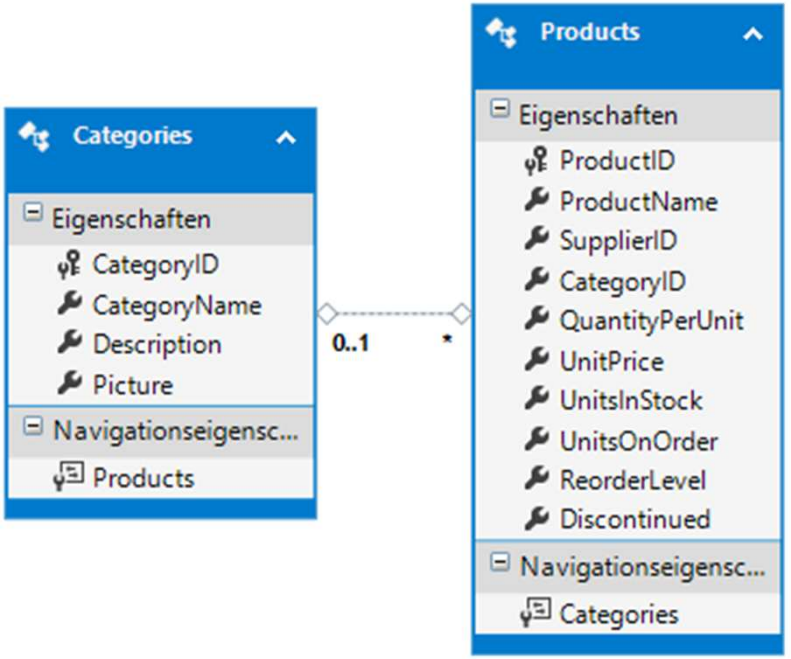
ADO.NET Entity Data Model hinzufügen

Verbinden mit der gewünschten Datenbank

Tabellen/Abfragen/Procedures abhaken

-> schon haben wir das Datenmodell im Projekt

Wichtig: Bei Änderungen Compilieren, dann werden erst die Klassen für Categories und Products angelegt.



Generierter Klassen Code

Der generierter Code für die Klasse sieht sehr übersichtlich aus.

Ohne weiteren Attribute, Kommentare oder EF-Code.

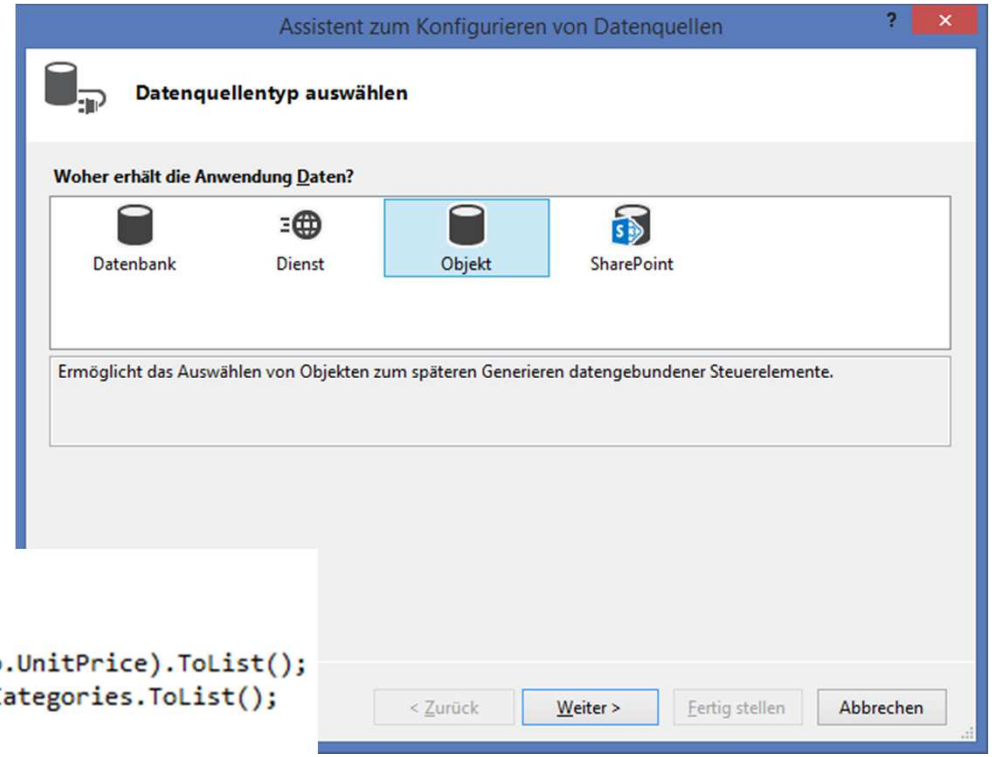
```
using System;
using System.Collections.Generic;

public partial class Categories
{
    public Categories()
    {
        this.Products = new HashSet<Products>();
    }

    public int CategoryID { get; set; }
    public string CategoryName { get; set; }
    public string Description { get; set; }
    public byte[] Picture { get; set; }

    public virtual ICollection<Products> Products { get; set; }
}
```

Grid mit Objekt-Datenquelle

Anbindung an Grid etwa zur Designzeit über BindingSources.

Mit der Info über die Felder kann man das Grid schön konfigurieren.

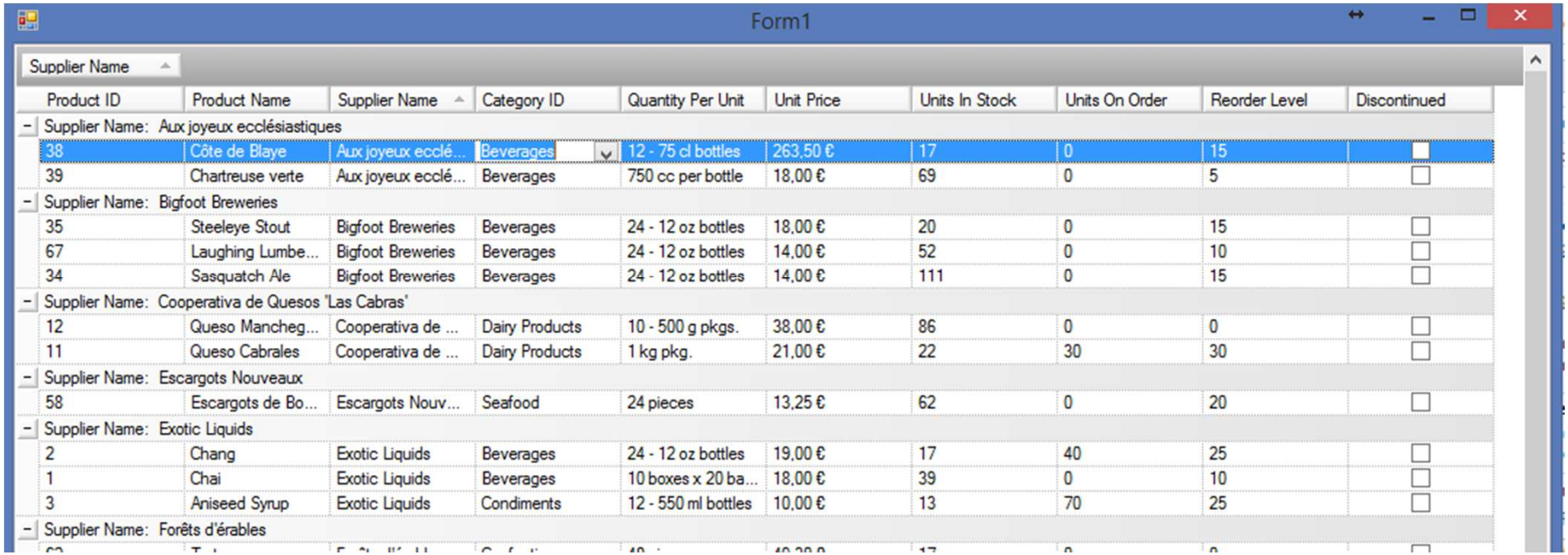
Datenoperationen erfolgen bequem über LinQ-Statments:




```
private void Form1_Load(object sender, EventArgs e)
{
    var dc = new nwEntities();
    gridEX1.DataSource = dc.Products.OrderByDescending(p=>p.UnitPrice).ToList();
    gridEX1.DropDowns["CategoryDropDown"].DataSource = dc.Categories.ToList();
}
```

Anzeige der Daten über ein Grid (GridEx)

Das Grid kann nun selbst gruppieren



Den Namen des Suppliers banden wir über Partial Classes an.

Partial Classes erlauben uns neben den automatisch generierten weitere Eigenschaften, Methoden, Events etc… zu implementieren

```
namespace EFBeispiel.Model
{
    public partial class Products
    {
        public string SupplierName
        {
            get { return Suppliers.CompanyName; }
        }

    }
}
```

Bei den Products laden wir die Suppliers noch gleich mit.

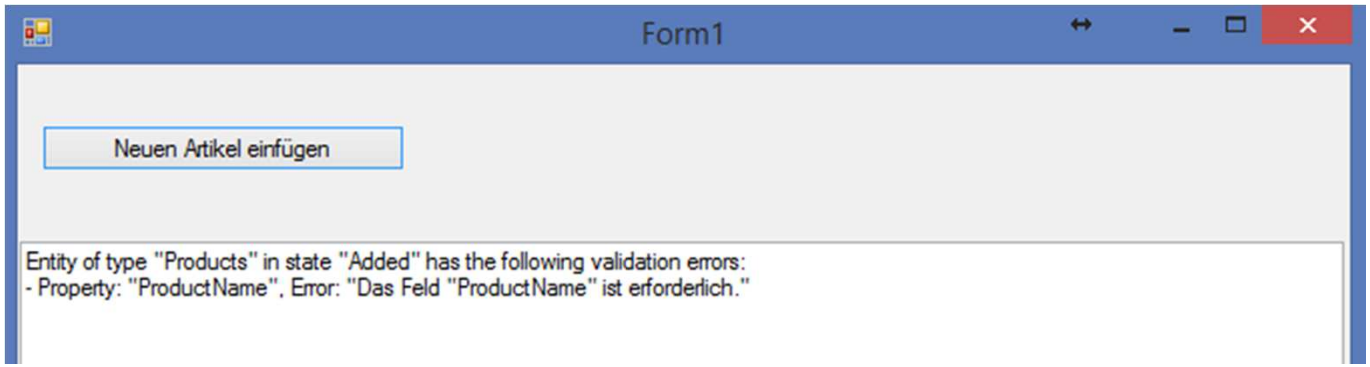
```
private void Form1_Load(object sender, EventArgs e)
{
    var dc = new nwEntities();
    gridEX1.DataSource = dc.Products.Include("Suppliers").OrderByDescending(p=>p.UnitPrice).ToList();
    gridEX1.DropDowns["CategoryDropDown"].DataSource = dc.Categories.ToList();
}
```

B. Diagnose von Problemen beim Speichern

Der DataContext skriptet alle Änderungen in SQL und schickt dies an die Datenbank, wenn wir DataContext.SaveChanges() aufrufen.

Zwangsläufig kann es hier zu Fehlern kommen. Diese gilt es zu ermitteln.

Insbesondere in der Entwicklungsumgebung ist das schon von unschätzbarem Wert, wenn man als Entwickler den Hinweis irgendwie bekommt.



Codebeispiel zur DbEntityValidationException:

```
private static bool TrySave(nwEntities dc, out string error)
{
    error = "";
    try
    {
        dc.SaveChanges();
        return true;
    }
    catch (DbEntityValidationException e)
    {
        var sb = new StringBuilder();

        foreach (DbEntityValidationResult eve in e.EntityValidationErrors)
        {
            const string format = "Entity of type \"{0}\" in state \"{1}\" has the following validation errors:\r\n";
            sb.AppendFormat(format, eve.Entry.Entity.GetType().Name, eve.Entry.State);
            foreach (DbValidationError ve in eve.ValidationErrors)
            {
                const string formatProp = "- Property: \"{0}\", Error: \"{1}\" \r\n";
                sb.AppendFormat(formatProp, ve.PropertyName, ve.ErrorMessage);
            }
        }
        error = sb.ToString();
        // throw;
        return false;
    }
}
```

C. Log der Zugriffe eines DataContext

Etwa auf einer speziellen Webseite kann ich in einen Label die Daten aus dem Session-Objekt des Users ausgeben. In diesem Fall kann erzwinge ich „https:\\“

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Settings.Default.ForceHTTPS)
    {
        string url = Request.Url.AbsoluteUri;

        if (!url.StartsWith("https://"))
        {
            string[] parts = url.Split('/');
            parts[0] = "https:";
            parts[2] = "www.Foo.de";
            url = string.Join("/", parts);
            Response.Redirect(url);
        }
    }
    LogLabel.Text = Global.InitLog.Replace("\r\n", "<br />") + (string) Session["DbLog"];
}
```

DataContext.Database.Log benötigt eine Funktion (Delegate) als Parameter

Wir rufen also nicht eine Funktion aus oder lesen, was im Log steht, sondern übergeben unsere Log-Funktion, welche aus Operationen im DataContext aufgerufen wird.

```
protected override void Page_Load(object sender, EventArgs e)
{
    base.Page_Load(sender, e);
    Session["DbLog"] = "";
    var dataContext = new IsEntities();          //new ContextE
    dataContext.Database.Log = DbLog;
```

```
public void DbLog(string log)
{
    if (Session["DbLog"] == null)
    {
        Session["DbLog"] = "";
    }
    Session["DbLog"] += log.Replace("\r\n", "\n").Replace("\n", "<br />") + "<br />";
}
```

Log in WinForms

Winforms nutzen seit .NET 2.0 einen eigenen UI-Thread.
Somit muss man dort mit dem Log in den UI Thread hineinkommen.

Man sieht einmal die Übergabe der Funktion

Das andere Mal rufen wir unsere eigene Funktion Log() direkt auf.

```
List<Products> products;
using (var dc = new nwEntities())
{
    if (doLogCheckBox.Checked) dc.Database.Log = Log;
    products = await dc.Products.Include(p=>p.Categories).ToListAsync();
    Log(string.Format("\t{0} categories loaded via 'Include()'.", dc.Categories.Local.Count));
}
```

Log in den UI Thread umbiegen

Die WinForms-Bibliothek liefert uns schon alles, was wir benötigen.

Wenn ein Aufruf im UI Thread benötigt wird, wird dies per Invoke() aufgerufen und landet danach im Else-Zweig bei der eigentlichen Ausgabe.

```
private void Log(string logText)
{
    if (textBox1.InvokeRequired)
        textBox1.Invoke(new MethodInvoker(() => Log(logText)));
    else
        textBox1.AppendText(DateTime.Now.ToString("HH:mm:ss.fff") + "\t" + logText + "\r\n");
}
```

Beispielausgabe

Mehr zum asynchronen Abrufen von Daten später.

Wir erkennen, wie das EF den Aufruf in SQL umsetzt.

Das Log können / müssen wir für jeden DataContext einschalten.

```
08:57:29.916        Opened connection asynchronously at 11.02.2015 08:57:29 +01:00

08:57:29.947        SELECT
    [Extent1].[CategoryID] AS [CategoryID],
    [Extent1].[CategoryName] AS [CategoryName],
    [Extent1].[Description] AS [Description],
    [Extent1].[Picture] AS [Picture]
    FROM [dbo].[Categories] AS [Extent1]
08:57:29.947

08:57:29.962        -- Executing asynchronously at 11.02.2015 08:57:29 +01:00

08:57:29.978        -- Completed in 0 ms with result: SqlDataReader

08:57:29.994

08:57:30.025        Closed connection at 11.02.2015 08:57:30 +01:00

08:57:30.041                9 other Categories loaded.
```

D) DataAnnotations in MVC (Model-View-Controller)

Wir haben einerseits feste Längen für Texte als auch eigene Vorstellungen, welche Daten in der Datenbank abgelegt werden können.

Das EF nutzt dies für WinForms oder WebForms nicht.
In MVC auch leider nicht für den „Database First“ Ansatz.

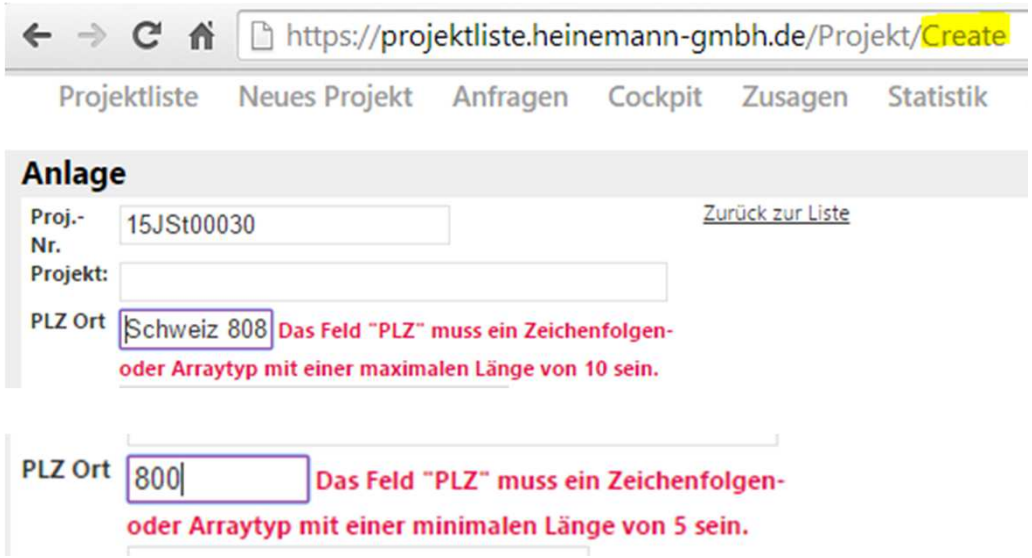
Dennoch kann man MVC *überreden*, diese Vorgaben per Data Annotations deklarativ zu berücksichtigen.

```
[MaxLength(10)]
[MinLength(5)]
[Display(Name = "PLZ")]
[Required]
public string ProjektPlz { get; set; }
```

```
// ReSharper disable UnusedMember.Global
[Key]
public int ProjektID { get; set; }

[DataType(DataType.Date)]
[Display(Name = "Erstellungsdatum")]
public DateTime? AngebotsDatum { get; set; }
```

MVC Seite Projekt/Create wird per Razor quasi automatisch erstellt

Bei der Validierung ziehen unsere DataAnnotations nun automatisch. MVC erzeugt das clientseitige JavaScript selbst. Für mich ein Traum.



Wie bewerkstelligen wir das?

Das EF wird bei jeder Änderung unsere DataAnnotations überschreiben. Ideal wäre, wenn es diese selbst noch aus der Datenbank zöge. In einer Partial Class können wir keine Attribute zu bestehenden Properties anlegen.

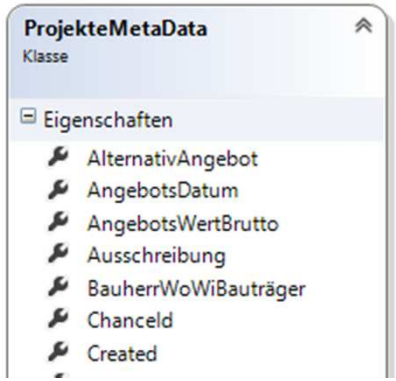
Somit liegen unsere Annotations in einer Klasse: ProjekteMetaData neben der Klasse Projekte

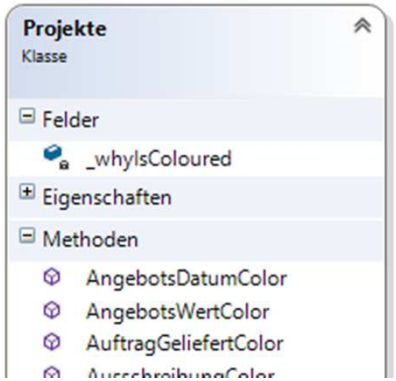
Die Verbindung schafft das Attribut MetadataType in der Partial Class.

Projekte.partial.cs

ForeCast | ForeCast.Models.Projekte

```
7   namespace ForeCast.Models
8   {
9       [MetadataType(typeof (ProjekteMetaData))]
10      public partial class Projekte
11      {
```





2. DataContext

DEN DATACONTEXT KANN MAN SICH ALS POOL VORSTELLEN, IN DEM ALLE GELADENEN / GEÄNDERTEN UND NEUEN DATENSÄTZE LIEGEN.

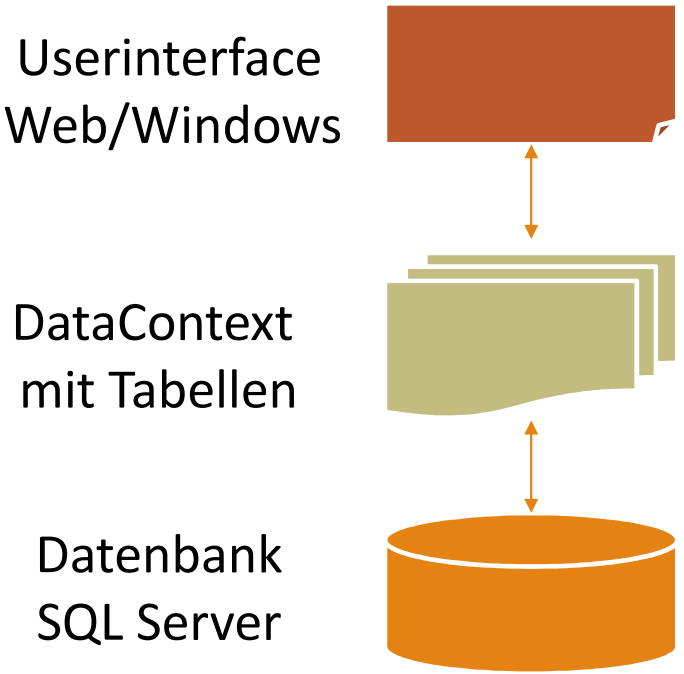
Fluss der Daten zwischen SQL Server und Applikation

Das Entity Framework hält schon geladene Daten intern in einem Pool (DataContext) von Tabellen.

Wir müssen uns eigentlich nicht darum kümmern, sondern können möglichst unbeschwert auf die Objekte (Records) zugreifen.

Nicht im Pool vorliegende referenzierte Datensätze lädt LazyLoading nach.

Schließt man den DataContext, so kann keine Kommunikation mit der DB mehr stattfinden!

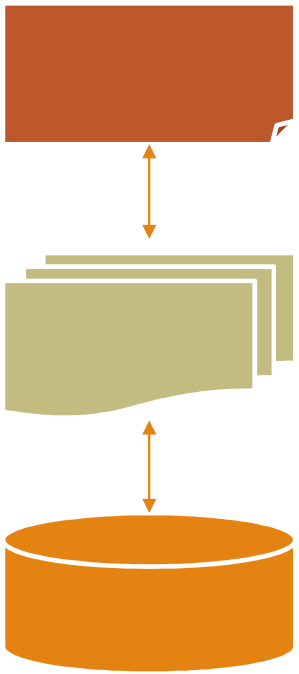


Speichern von Änderungen

Egal wie viele Änderungen in unseren Objekten vorliegen, diese werden immer alle per Transaktion an die Datenbank geschickt.

Bricht ein Einfügen/Löschen/Ändern von einem Datensatz, so sind alle weiteren Useraktionen verloren. (Bis man nicht den Fehler korrigiert)

Jedes Mal, wenn man SaveChanges() aufruft, werden alle geskripteten Änderungen an die Datenbank geschickt – aber diese verweigert ggf. erneut Problemdatensatze anzunehmen.



Kommen wir mit einem DataContext aus?

Web Applikation

Wir können durchaus für jede Seite/Page die Daten mit neuen DataContext frisch holen und darstellen.

Bietet sich bei kleinen Applikationen mit wenig Traffic (wenige Aufrufe in der Minute) durchaus an.

Wenn etwas nicht funktioniert, kann man dem User dies signalisieren – er kann es dann erneut probieren. Schick ist es dies zu Protokollieren und zeitnah zu verbessern.

Windows Applikation

Wenn der User über Tage die Applikation offen lässt, dann müssen wir mit einigen Kollisionen rechnen.

Somit ist es hier deutlich schwieriger mit einem DataContext über längere Zeit zu leben.

Ideal wäre also dem Benutzer immer die „besten“ und „aktuellsten“ Daten zu bieten. Leider scheitert das an der Performance.

Wie kann unsere Applikation schneller, zuverlässiger und wartbarer werden?

In jeder Richtung stehen wir vor einem steilen Anstieg:

- Komplexität
- Performanz
- Aktualität

Alles treibt die Kosten des Projekts nach oben.

Ideen? Was können wir machen

EINE APPLIKATION IST EINE APPLIKATION…

Ziele sind:

- Performance zu verbessern
- Datenaktualität/Verringerung von Kollisionen
- Code einfacher/übersichtlicher zu halten

BENUTZER HABEN EIN THEMA

- Die Anwendung läuft zu langsam
- Es kommt immer wieder zu Abstürzen, weil der SQL Server die Annahme verweigert und wir nicht den aktuellen Stand der Daten vorhalten
- Der Code ist kaum zu warten, weil zu komplex und/oder dann zu wenig getestet wird.

Tipp: Bei der Darstellung eines Datensatzes einer Dimension wird diese immer aktualisiert

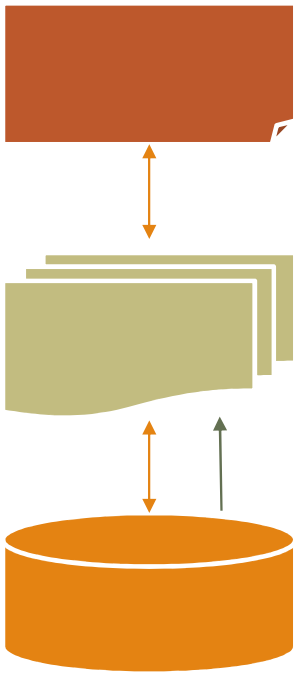
Wenn wir also einen Firmenstandort ansehen oder ein Produkt, so ist es wenig Aufwand hier zu aktualisieren.

Wenn hingegen eine Liste von Standorten/Produkten verwendet wird, sollten die schon geladenen Daten reichen.

Wenn wir in einem DataContext aktualisieren wollen, dürfen wir diesen nicht schließen.

Einzelne Items/Entitäten im Datacontext können so direkt aktualsisiert werden.

```
DataContext.Entry(product).Reload();
```



Aktualisieren eines Datensatzes

Dem Benutzer kann man ein Refresh der kompletten Dimension anbieten und alles neu laden.

Initial hat man den Code ja eh schon einmal verwendet.

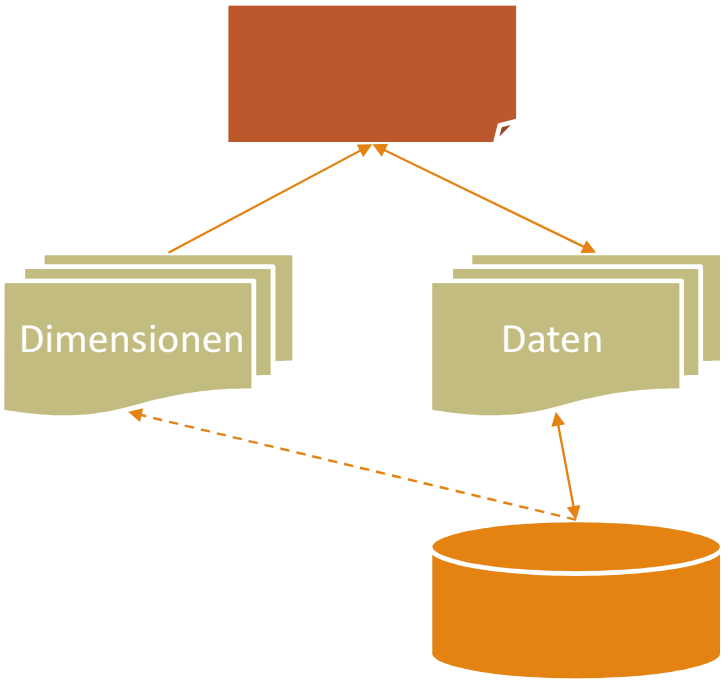
```
public void RefreshAll()
{
    DataContext = new Model.NorthwindEntities();
    productsBindingSource.DataSource = DataContext.Products.Take(10).ToList();
}

private void EinenDatensatzToolStripButton_Click(object sender, EventArgs e)
{
    if (productsBindingNavigator.BindingSource == null) return;
    var product = productsBindingNavigator.BindingSource.Current as Model.Products;
    if (product != null)
    {
        DataContext.Entry(product).Reload();
        productsBindingSource.ResetBindings(true);
    }
}
```

Vorladen von Dimensionen

Viele Daten ändern sich nur ganz selten.
Diese werden oft als Dimensionen, Nachschlagetabellen bezeichnet:

- Die Standorte der Firma
- Die Kategorien der Artikel
- Ausprägungen

Der DataContext für die Dimensionen muss nicht offen bleiben!
Das EF-Datenmodell muss nicht das gleiche wie das EF-Modell der Daten sein.



Vor- und Nachteil des Vorladens

VORTEILE

Es sind immer alle geladenen Dimensionen im Speicher verfügbar.

Im Code lassen sich Dictionaries zum Nachschlagen aufbauen. (Etwa die Artikel)

Beim Webserver kann man dies in den ApplicationStart legen. Webseiten werden schneller.

Beim Start eines Windowsprogramms toleriert der Benutzer eine längere Startphase leichter als bei der Arbeit mit dem Programm

NACHTEILE

Wenn sich was ändert, müssen wir doch Aktualisieren (Refresh)

Wenn wir das gleiche EF Datenmodell nehmen, so können wir schwer verhindern, dass nicht doch per LazyLoading Daten von Dimensionen geladen werden und somit doppelt mit unterschiedlichem Inhalt vorliegen

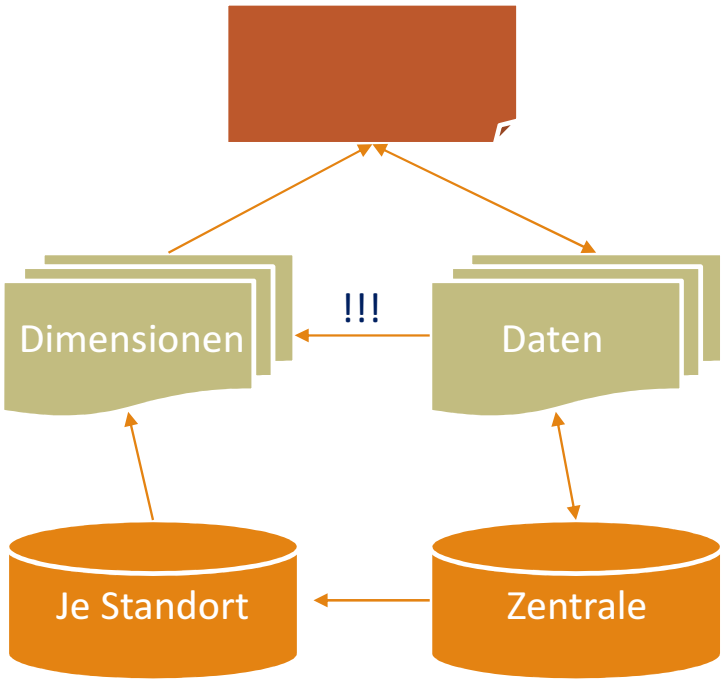
Wenn wir unterschiedliche Datenmodelle nehmen, müssen wir bei LinQ Abfragen aufpassen, die Dimensions-Texte einbeziehen

Dimensionen in Standorten

Meine Kunden haben teilweise Standorte, welche weiter weg sind, jedoch nutzen wir nicht auch non-Web-Applikations.

Jobs übertragen die Dimensionen lokal auf weiteren SQL Servern, etwa über Nacht oder auf Anforderung.

Die Applikation zieht beim Start lokal die Dimensionen und nur die jeweiligen Fakten zum aktuellen Vorgang des Mitarbeiters von der Zentrale.



Steuern des Datacontext

Wir fügen einen neuen Konstruktor zu unserem DataContext hinzu.

Je nachdem ob unser Benutzer im Netzwerk Hannover gefunden wurden (etwa IP-Nummer), kann er die passenden Daten von der lokalen Datenbank holen.

Auch für den DataContext können wir partial Classes nutzen. Das EF würde sonst diese Quelltextdatei mit eigenem Inhalt wieder überschreiben.

```
public static async Task LoadLookupData()
{
      using (LvEntities context = new LvContext(Program.UseDbHan).GetContext())
                {… Laden … }

}
```

DataContext dynamisch erzeugen.

Wir können etwa für jeden Standort einen eigenen SqlConnectionStringBuilder() nutzen und den Context passend übergeben.

Der Kollege in Hannover bekommt nun von seinem SQL Server (und seinem lokalen CRM-System) die Daten schneller geladen, als aus Bayern.

```
var sqlBuilderHan = new SqlConnectionStringBuilder
{
    DataSource = @"khh-dbs1\KOHLHAAS",
    InitialCatalog = "LV",
    IntegratedSecurity = true,
    MultipleActiveResultSets = true,
    ApplicationName = "LV2"
};
```

```
var context = new LvEntities();
context.Database.Connection.ConnectionString = (IsHan ? sqlBuilderHan : sqlBuilderGer).ToString();
return context;
```

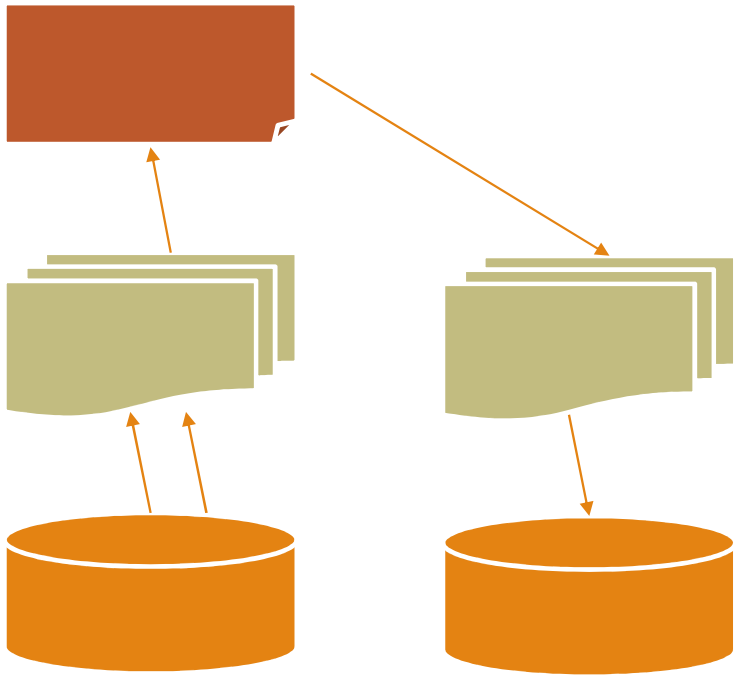
Speichern von Änderungen in weiterem DataContext vermeidet Kollisionen

Eine Möglichkeit besteht durchaus, Änderungen jeweils in einem eigenen DataContext durchzuführen.

Also einen DataContext nur zum Lesen von möglichst viel Informationen zu nutzen (und ggf. ständig offen zu lassen)

Einen zweiten DataContext nur kurz offen zu halten, etwa an dem frischen Datensatz die Änderung zu machen und dies an die Datenbank zu übergeben.
Wichtig ist es danach im ersten DataContext diesen Datensatz zu aktualisieren.

Bedeutet: wir greifen 3x auf die Datenbank zu.



SQL Abfragen nutzen für lange Laufzeit

Wenn wir (etwa für eine Jahresstatistik) alle Daten per LinQ holen und selbst auswerten – kann dies die Datenbank eigentlich besser.

Somit erstellen wir eine eigene Abfrage und laden diese in unser Datenmodell.

In anderen Fällen bietet sich auch ein täglicher Job an, der statt der Abfrage eine echte Tabelle füllt.

Nachteile:

Entity Framework kann von Abfragen keinen Primärschlüssel erstellen. Diesen muss man immer nachjustieren.

Relationen zu Dimensionen müssen wir von Hand setzen.

Datenlage unterscheidet sich von den aus Tabellen geladenen Daten.

Intranet Hauptseite aktuell halten

Die Startseite wird mehrfach am Tag aufgerufen und offen gehalten.

In meinem Fall soll diese alle 20 Sekunden aktualisiert werden.

Laden wir die kompletten Daten immer wieder wirklich neu, so stellt dies einen enormen Traffic für den SQL Server dar.

Abhilfe: Die Webseite merkt sich, wann der letzte Datenabzug des „WebParts“ war und cache´t den kompletten Textblock.

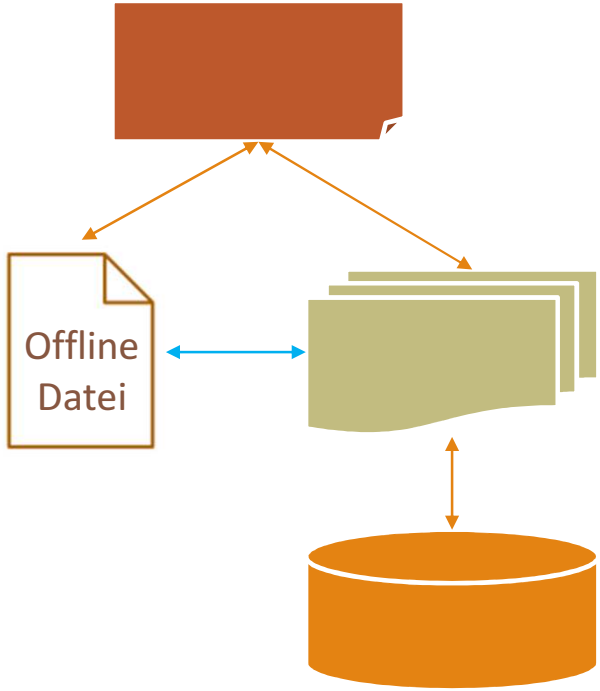
In der Datenbank wird eine Tabelle per Trigger gefüllt, welche das letzte Änderungsdatum für diese Informationen enthält.

Nur bei frischerem Änderungsdatum, muss die Seite den Block neu rendern und ausliefern.

Variante: Offline Applikation

Die Applikation arbeitet auf einem Repository, welches:

- entweder über ein oder mehrere DataContexte gespeist wird.
- oder aus einer Datei de-serialisiert wird. Nach dem Arbeiten/Speichern wird eine neu serialisierte Version gespeichert.



Excel hilft bei vielen Klassen

Die Properties benötigen entsprechenden Code für Serialisierung und Deseralisierung.

Da wir mit Partial Classen durchaus weitere Eigenschaften einbringen, kann es geschickt sein, feste Properties zur Serialisierung heranzuziehen. Bei Änderungen im Modell können wir auf die Änderungen per Code entgegenwirken.

Im Beispiel hab ich die Properties der OrderDetails-Tabelle in die Spalten A-G eingefügt.

In den Spalten H und K findet man den Code für die Serialisierung und Deserialisierung.

A	B	C	D	E	F	G	H	K
public	int	OrderID	{	get;	set;	}	info.AddValue("OrderID", OrderID);	OrderID = (int) si.GetValue("OrderID", typeof(int));
public	int	ProductID	{	get;	set;	}	info.AddValue("ProductID", ProductID);	ProductID = (int) si.GetValue("ProductID", typeof(int));
public	decimal	UnitPrice	{	get;	set;	}	info.AddValue("UnitPrice", UnitPrice);	UnitPrice = (decimal) si.GetValue("UnitPrice", typeof(decimal));
public	short	Quantity	{	get;	set;	}	info.AddValue("Quantity", Quantity);	Quantity = (short) si.GetValue("Quantity", typeof(short));
public	float	Discount	{	get;	set;	}	info.AddValue("Discount", Discount);	Discount = (float) si.GetValue("Discount", typeof(float));

Klasse mit [Serializable] Attribut versehen

Wir markieren nicht nur diese Klasse als [Serializable] und somit auch alle abhängigen Klassen, sondern benötigen auch die Schnittstelle ISerializable

```
[Serializable]
public partial class LvItemGroup : ISerializable, IGroup
{
```

Serialisierung implementieren

In die Partial Class fügen wir unseren „Excel“ Code ein.

```
protected LvItemGroup(SerializationInfo si, StreamingContext context) : this()
{
    LvItemGroupId = (int) si.GetValue("LvItemGroupId", typeof (int));
    LVID = (int) si.GetValue("LVID", typeof (int));
    Caption = (string) si.GetValue("Caption", typeof (string));
    Created = (DateTime) si.GetValue("Created", typeof (DateTime));
    UnterKategorieID = (int) si.GetValue("UnterKategorieID", typeof (int));
    Sort = (int) si.GetValue("Sort", typeof (int));
    ShowCaption = (bool) si.GetValue("ShowCaption", typeof (bool));
}

public void GetObjectData(SerializationInfo info, StreamingContext context)
{
    info.AddValue("LvItemGroupId", LvItemGroupId);
    info.AddValue("LVID", LVID);
    info.AddValue("Caption", Caption);
    info.AddValue("Created", Created);
    info.AddValue("UnterKategorieID", UnterKategorieID);
    info.AddValue("Sort", Sort);
    info.AddValue("ShowCaption", ShowCaption);
}
```

Lesen und Schreiben ist dann ein Einzeiler

Ich nutze das komplette Repository als ein Objekt mit Listen Objekten

```
using (var fs = new FileStream(path, FileMode.Open, FileAccess.Read))
{
    var formatter = new BinaryFormatter();
    try
    {
        formatter.AssemblyFormat = FormatterAssemblyStyle.Simple;
        formatter.FilterLevel = TypeFilterLevel.Low;
        offlineData = formatter.Deserialize(fs) as OfflineData;
```

```
try
{
    using (var fs = new FileStream(path, FileMode.Create, FileAccess.Write))
    {
        var binaryformatter = new BinaryFormatter();
        binaryformatter.Serialize(fs, data);
    }
    return true;
}
```

3.
Lazy Loading

Lazy Loading lädt nur explizit angeforderte Referenzen mit.
Also nicht gleich alle Artikel-Datensätze, wenn ich eine Order lade.

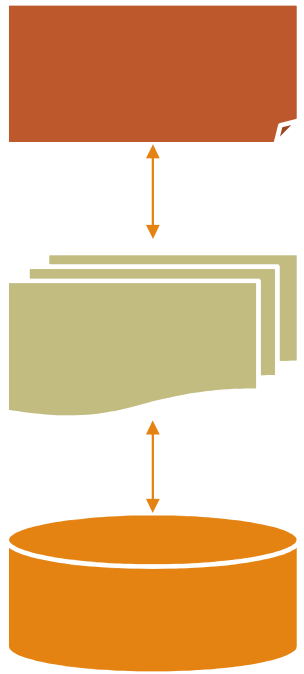
DAS GEGENSTÜCK NENNT SICH „EAGER LOADING“ UND LÄDT ALLE REFERENZEN GLEICH MIT.

Lazy Loading erlaubt das Nachladen

Laden wir eine Auswahl aus der Tabelle Bestellungen (Orders) mit dem DataContext, so sind die Order Details nicht dabei.

Laufen wir in einer Schleife über die Bestellungen und wollen die OrderDetails darstellen, so wird für jeden im DataContext fehlenden Datensatz ein SQL Statement generiert.

Abhilfe wäre beim Laden schon Include() zu verwenden.



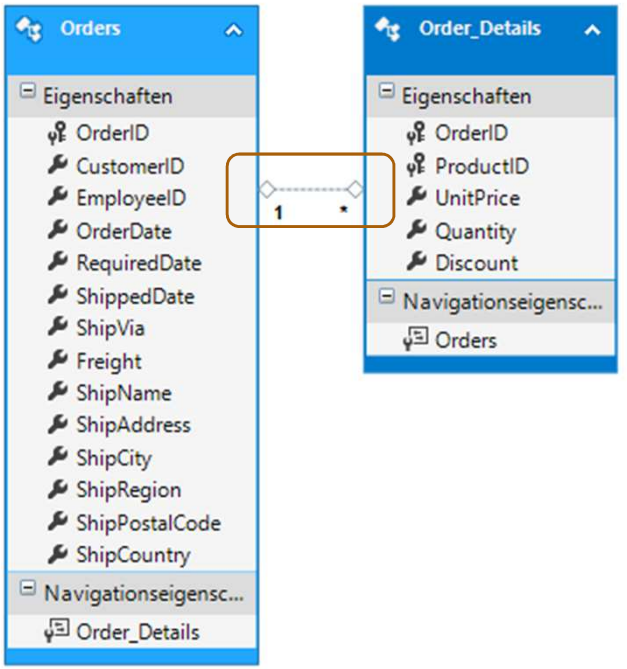
Lazy Loading

Wir laden einige Orders mit LinQ.

Greifen wir später über die Relation darauf zu, so lädt das EF die entsprechenden Datensätze nach.

Genauso kennt ein OrderDetails Datensatz solange seine Order nicht, bis diese per LazyLoading über das EF geladen wurde.

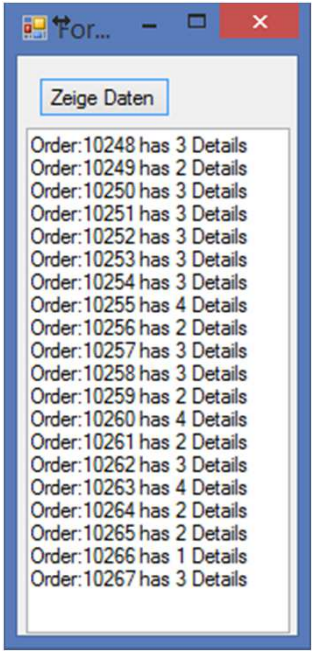
Jede Benutzung von Lazy Loading kann somit einen Datenbankaufruf zur Folge haben, wenn die Daten noch nicht vorliegen.



Beispiel 1: Ohne Lazy Loading mit Include

Nur im „Using“ – Block gibt es unseren DataContext „dc“!
Dieser lädt die Daten frisch – wir laden alle Details für eine Auswertung gleich mit.

```
IEnumerable<Orders> someOrders;
using (var dc = new NwEntities())
{

    someOrders = dc.Orders.Include("Order_Details").Take(20).ToList();
}
var sb = new StringBuilder();
foreach (var order in someOrders)
{
    const string format = "Order:{0} has {1} Details";
    sb.AppendLine(string.Format(format, order.OrderID, order.Order_Details.Count()));
}
textBox1.Text = sb.ToString();
```



Beispiel 2: Lazy Loading

Wir laufen durch die Bestellungen und holen uns die Namen der Produkte.

Danach stellen wir die im DataContext vorhandene Datenmenge dar.

Die Anzahlen nehmen zu, somit erkennt man gut, dass:

- Details werden komplett geladen

- Bei den Products werden nur die Benötigten geholt.

```
Order:10248 contains Queso Cabrales, Singaporean Hokkien Fried Mee, Mozzarella di Giovanni
Loaded Details : 3 - Loaded Products: 3
Order:10249 contains Tofu, Manjimup Dried Apples
Loaded Details : 5 - Loaded Products: 5
Order:10250 contains Jack's New England Clam Chowder, Manjimup Dried Apples, Louisiana Fiery H
Loaded Details : 8 - Loaded Products: 7
Order:10251 contains Gustaf's Knäckebröd, Ravioli Angelo, Louisiana Fiery Hot Pepper Sauce
Loaded Details : 11 - Loaded Products: 9
Order:10252 contains Sir Rodney's Marmalade, Geitost, Camembert Pierrot
Loaded Details : 14 - Loaded Products: 12
Order:10253 contains Gorgonzola Telino, Chartreuse verte, Maxilaku
Loaded Details : 17 - Loaded Products: 15
Order:10254 contains Guaraná Fantástica, Pâté chinois, Longlife Tofu
Loaded Details : 20 - Loaded Products: 18
Order:10255 contains Chang, Pavlova, Inlagd Sill, Raclette Courdavault
Loaded Details : 24 - Loaded Products: 22
Order:10256 contains Perth Pasties, Original Frankfurter grüne Soße
Loaded Details : 26 - Loaded Products: 24
Order:10257 contains Schoggi Schokolade, Chartreuse verte, Original Frankfurter grüne Soße
Loaded Details : 29 - Loaded Products: 25
Order:10258 contains Chang, Chef Anton's Gumbo Mix, Mascarpone Fabioli
Loaded Details : 32 - Loaded Products: 27
Order:10259 contains Sir Rodney's Scones, Gravad lax
Loaded Details : 34 - Loaded Products: 29
Order:10260 contains Jack's New England Clam Chowder, Ravioli Angelo, Tarte au sucre, Outback
Loaded Details : 38 - Loaded Products: 31
Order:10261 contains Sir Rodney's Scones, Steeleye Stout
Loaded Details : 40 - Loaded Products: 32
Order:10262 contains Chef Anton's Gumbo Mix, Uncle Bob's Organic Dried Pears, Gnocchi di nonna
Loaded Details : 43 - Loaded Products: 34
Order:10263 contains Pavlova, Guaraná Fantástica, Nord-Ost Matjeshering, Longlife Tofu
Loaded Details : 47 - Loaded Products: 35
Order:10264 contains Chang, Jack's New England Clam Chowder
Loaded Details : 49 - Loaded Products: 35
Order:10265 contains Alice Mutton, Outback Lager
Loaded Details : 51 - Loaded Products: 36
Order:10266 contains Queso Manchego La Pastora
Loaded Details : 52 - Loaded Products: 37
Order:10267 contains Boston Crab Meat, Raclette Courdavault, Lakkalikööri
Loaded Details : 55 - Loaded Products: 39
```

Code für Lazy Loading

Über die „Order“ holen wir die „order.Order_Details“.

```
var sb = new StringBuilder();
var dc = new NwEntities();
foreach (var order in dc.Orders.Take(20).ToList())
{
    var products = string.Join(", ", order.Order_Details
        .Select(d => d.Products.ProductName).ToList());

    const string format1 = "Order:{0} contains {1}\r\n";
    sb.AppendFormat(format1, order.OrderID, products);
    const string format2 = "Loaded Details : {0} - Loaded Products: {1}\r\n";
    sb.AppendFormat(format2,dc.Order_Details.Local.Count, dc.Products.Local.Count);
}

textBox1.Text = sb.ToString();
```

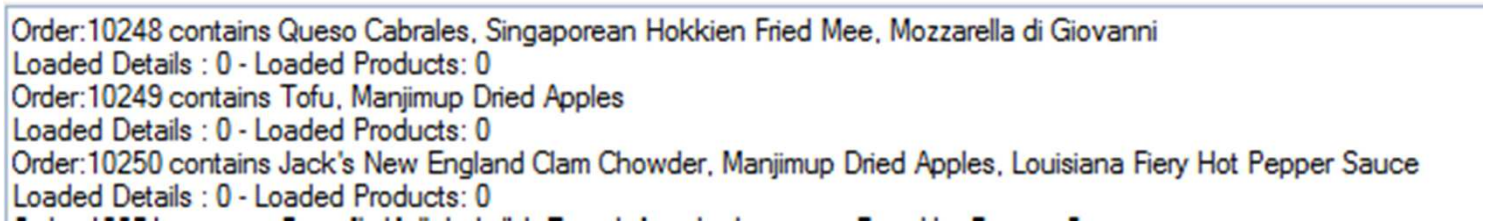
Beispiel 3: Findet hier Lazy Loading statt?

..?..

```
var sb = new StringBuilder();
var dc = new NwEntities();
foreach (var order in dc.Orders.Take(20).ToList())
{
    const string format = "Order:{0} contains {1}\r\n";
    var products = string.Join(", ", dc.Order_Details
        .Where(d => d.OrderID == order.OrderID)
        .Select(d => d.Products.ProductName).ToList());
    sb.AppendFormat(format, order.OrderID, products);
    sb.AppendFormat("Loaded Details : {0} - Loaded Products: {1}\r\n",
        dc.Order_Details.Local.Count, dc.Products.Local.Count);
}

textBox1.Text = sb.ToString();
```

Datenbankaufrufe

Es werden keine Objekte in den DataContext übernommen!



Selbst wenn wir alles vorher geladen hätten, haben wir ja EF gebeten uns den aktuellen Stand aus der Datenbank mitzuteilen.

Das kostet uns im Ernstfall enorm Performance!

Diese Abfrage stellt andere/bessere/aktuellere Daten dar, als die App in anderen Fenstern zeigt.

Ansicht im Profiler

Dies wird 20x ausgeführt mit unterschiedlichem Parameter:

```
exec sp_executesql N'SELECT
    [Extent2].[ProductName] AS [ProductName]
    FROM  [dbo].[Order Details] AS [Extent1]
    INNER JOIN [dbo].[Products] AS [Extent2] ON
            [Extent1].[ProductID] = [Extent2].[ProductID]
    WHERE [Extent1].[OrderID] = @p__linq__0',N'@p__linq__0
int',@p__linq__0=10257
```

SQL Server Profiler

..

Untitled - 1 (NGC2012)

EventClass	TextData	ApplicationName	NTUserName	LoginName	CPU	Reads	Writes	Duration
Audit Login	-- network protocol: TCP/IP set qu...	EntityFramework	Jan	NGC\Jan				
RPC:Completed	exec sp_executesql N'SELECT [...	EntityFramework	Jan	NGC\Jan	0	10	0	0
Audit Logout		EntityFramework	Jan	NGC\Jan	0	218	0	3
RPC:Completed	exec sp_reset_connection	EntityFramework	Jan	NGC\Jan	0	0	0	0
Audit Login	-- network protocol: TCP/IP set qu...	EntityFramework	Jan	NGC\Jan				
RPC:Completed	exec sp_executesql N'SELECT [...	EntityFramework	Jan	NGC\Jan	0	6	0	0
Audit Logout		EntityFramework	Jan	NGC\Jan	0	224	0	0
RPC:Completed	exec sp_reset_connection	EntityFramework	Jan	NGC\Jan	0	0	0	0
Audit Login	-- network protocol: TCP/IP set qu...	EntityFramework	Jan	NGC\Jan				
RPC:Completed	exec sp_executesql N'SELECT [...	EntityFramework	Jan	NGC\Jan	0	8	0	0
Audit Logout		EntityFramework	Jan	NGC\Jan	0	232	0	0
RPC:Completed	exec sp_reset_connection	EntityFramework	Jan	NGC\Jan	0	0	0	0
Audit Login	-- network protocol: TCP/IP set qu...	EntityFramework	Jan	NGC\Jan				
RPC:Completed	exec sp_executesql N'SELECT [...	EntityFramework	Jan	NGC\Jan	0	8	0	0
Audit Logout		EntityFramework	Jan	NGC\Jan	0	240	0	3
RPC:Completed	exec sp_reset_connection	EntityFramework	Jan	NGC\Jan	0	0	0	0

4. Laden von Daten per Thread oder Asynchron

ASYNCHRONE OPERATIONEN VERWENDEN ASYNC UND AWAIT.

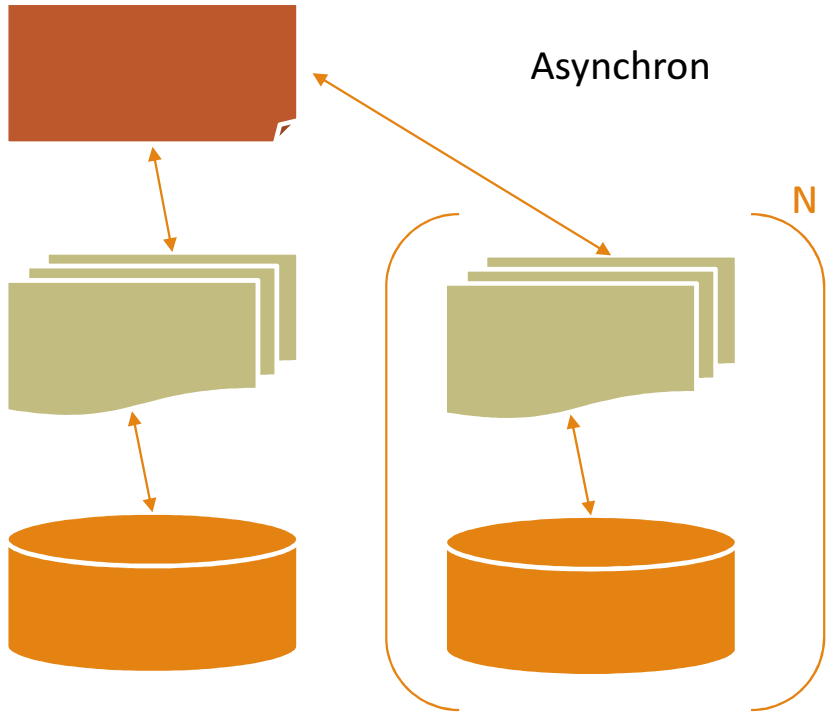
Mythos: Durch Asynchrone Programmierung wird alles einfacher, schneller ….

Starten wir im DataContext mehrere Aufgaben, so bricht der Code, wenn eine LinQ-Abfragen nicht vor der nächsten fertig ist.

Es kann also in der Entwicklungsumgebung funktionieren, später aber nicht mehr und dann hat man ggf. etwas Stress!

Synchrone und Asynchrone Aufrufe kann man zwar mischen, aber nur dann, wenn diese wiederum zeitlich nicht kollidieren!

Resümee: Wenn also eh alles aufeinander warten muss, bringt ein DataContext bei Async nur mehr Aufwand und eine komplexere Applikation.



Lösung per Threads

Rechts findet sich eine Liste von Commands, welche wir parallel in unterschiedlichen Threads laden wollen.

Methode „InitThread()“ ruft die „Execute“ Funktion des Commends auf.

Join() wartet, bis alle Threads abgearbeitet sind.

```
private static void RefreshData(Splash splash)
{
    if (splash != null) splash.AddInfo("Laden der Daten");

    var dbCommands = new Collection<DataBaseCommand>
                     {
                         new ReadMitarbeiterFromDb(),
                         new ReadStammdatenRechteFromDb(),
                         new ReadOffeneKundendienstfälleFromDb(),
                         new ReadBlumenFromDb(),
                         new ReadArtikelabrechnungsartenFromDb(),
                         new ReadInbetriebnahmegeräteFromDb(),
                         new ReadArtikelFromDb(),
                         new ReadDateipfadeFromDb(),
                         new ReadDateiartenFromDb(),
                         new ReadAufgabenFromDb(),
                         new ReadProdukteFromDb(),
                         new ReadKontakttypenFromDb(),
                         new ReadWarenprüfungFelderFromDb(),
                     };
    var threads = new Collection<Thread>();
    foreach (DataBaseCommand dataBaseCommand in dbCommands)
    {
        Thread thread = dataBaseCommand.InitThread();
        threads.Add(thread);
        thread.Start();
    }

    foreach (Thread dbCommand in threads)
        dbCommand.Join();

    if (splash != null) splash.AddInfo("Starten der Anwendung... ");
}
```

Ausgabe der Informationen an UI-Thread

Da eine WinForms-Anwendung seit .NET 2.0 einen UI-Thread hat müssen Ausgaben auch in diesem Thread erfolgen.

Somit benötigen wir bei dem Ansatz mit Threads abfragen, ob wir in den UI-Thread wechseln müssen.

Das ist bei der Nutzung von Async nicht nötig.

```
public void AddInfo(string infoline)
{
    if (SplashTextBox.InvokeRequired)
        SplashTextBox.Invoke(new MethodInvoker(() => AddInfo(infoline)));
    else
    {
        SplashTextBox.AppendText(infoline + "\r\n");
        Application.DoEvents();
    }
}
```

Asynchrone Lösung

Wir haben exakt die gleichen Commands.

Wir rufen dieses Mal „ExecuteAsync“ auf.

Eigentlich kämen die Ausgaben, der mit „await“ gestarteten Funktionen nacheinander dran.

Der Wunsch ist aber diese in „Erledigt“ Reihenfolge darzustellen.

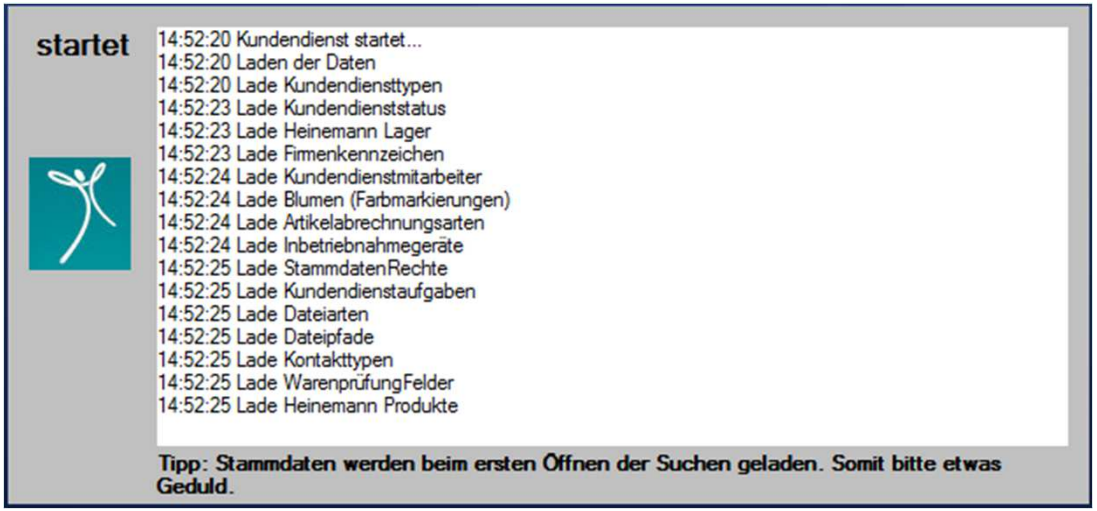
Der Benutzer möchte ja im Splash Fenster einfach nur wissen, dass die Applikation was tut.

„InCompletionOrder“ ist keine Framework-Methode, sondern muss man sich erstellen / aus der .NET Community kopieren.

```
private static async Task<bool> RefreshData2(Splash2 splash)
{
    if (splash == null) return false;
    var dbCommands = new List<SrDataBaseCommand>
                     {
                         new ReadMitarbeiterFromDb(),
                         new ReadStammdatenRechteFromDb(),
                         new ReadOffeneKundendienstfälleFromDb(),
                         new ReadBlumenFromDb(),
                         new ReadArtikelabrechnungsartenFromDb(),
                         new ReadInbetriebnahmegeräteFromDb(),
                         new ReadArtikelFromDb(),
                         new ReadDateipfadeFromDb(),
                         new ReadDateiartenFromDb(),
                         new ReadAufgabenFromDb(),
                         new ReadProdukteFromDb(),
                         new ReadKontakttypenFromDb(),
                         new ReadWarenprüfungFelderFromDb(),
                     };

    var tasks = dbCommands.Select(dbCmd => dbCmd.ExecuteAsync()).ToList();

    foreach (var task in tasks.InCompletionOrder())
    {
        var result = await task;

        splash.AddInfo(result);
    }
    splash.AddInfo("Starten der Anwendung... ");

    return true;
}
```

Ergebnis per Async

Die Tasks werden in der „erledigt“-Reihenfolge angezeigt.
In dem Screenshot finden wir die Artikel noch nicht.



Wie sehen meine Commands aus

Man sieht den Unterschied zwischen:

synchronem Aufruf per „Execute“

asynchronem Aufruf per „DoCommandAsync“.

Den Rest erledigen die Basisklassen des „Commands“

```
public class ReadMitarbeiterFromDb : SrDataBaseCommand
{
    protected override string SplashText
    {
        get { return "Lade Kundendienstmitarbeiter"; }
    }

    public override void Execute()
    {
        SrData.Mitarbeiters = DataContext.Mitarbeiter.ToList();
        SrData.MitarbeitersAktiv = SrData.Mitarbeiters.Where(m => m.Aktiv)
            .OrderBy(m => m.Kurz).ToList();

        base.Execute();
    }

    protected override async Task DoCommandAsync()
    {
        SrData.Mitarbeiters = await DataContext.Mitarbeiter.ToListAsync();
        SrData.MitarbeitersAktiv = SrData.Mitarbeiters.Where(m => m.Aktiv)
            .OrderBy(m => m.Kurz).ToList();
    }
}
```

Was ist besser?

Das Umstellen von Threads auf Tasks ist etwas mühselig. Insbesondere die Einarbeitung in das „neue“ C# 5.0 Thema.

Async kann nicht ad hoc eine Parallelität abbilden, aber entlastet uns langfristig. Insbesondere beim UI-Thread Thema trifft uns bei der Asynchronen Programmierung deutlich weniger.

Die Applikation (Winforms) friert zwar nicht mehr ein, wenn wir nebenher viel machen/laden - dennoch müssen wir dort dem User weitere Aktionen unterbinden (etwa Button enablen).

Ein Thread benötigt/reserviert sich als Standard 1 MB Speicher!

Task benötigen Threads, das Framework verteilt das aber selbst, somit werden deutlich weniger Threads verwendet.

Transaktion

DAS ENTITY FRAMEWORK WICKELT DIE DATENOPERATIONEN IM TRANSAKTIONELLEN RAHMEN AB. SOMIT WERDEN ENTWEDER ALLES DATEN ÜBERTRAGEN ODER DIE AKTION ZURÜCKGEROLLT

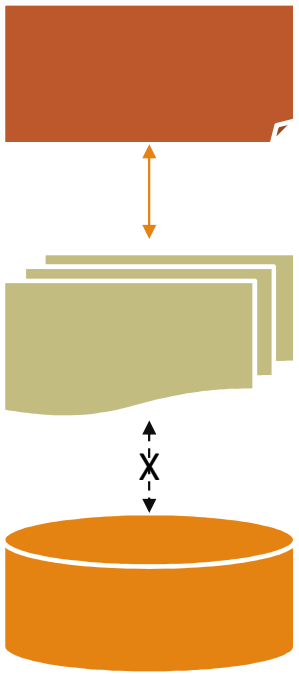
Was bedeutet es für unsere Applikation?

Im schlimmsten Fall haben wir eine Webapplikation und nur einen DataContext über alle Seiten.

Ab dem Zeitpunkt, ab dem ein Datensatz nicht mehr geschrieben werden kann – kann keine weitere Änderung mehr gespeichert werden!

Beispiele:

- Ein Datenfeld wäre zu lang,
- eine Referentielle Integrität wäre verletzt oder
- ein nicht-NULL Feld wird versucht mit NULL zu füllen.
- …



Abhilfe gekapselte Datenoperationen

Wir fügen die Entität wieder hinzu, wenn diese irgendwie nicht löschbar ist.

Genauso beim Einfügen neuer Datensätze.

Wenn sich ein Datensatz nicht Einfügen lässt, dann entfernen wir den Datensatz wieder aus dem DataContext. Die Applikation gibt die Fehlermeldung dann nach oben.

```
public bool Delete(T t)
{
    ContextTable.Remove(t);
    try
    {
        DataContext.SaveChanges();
        return true;
    }
    catch (Exception e)
    {
        Console.WriteLine(e);
        ContextTable.Add(t);

        DataContext.SaveChanges();
    }
    return false;
}
```

Einfügen von Properties per Reflektion

Wenn wie viele Tabellen haben und die Meisten, aber eben nicht alle ein Datenfeld besitzen, welches wir füllen können, dann können wir dies per Reflektion testen und das Feld passend füllen.

Die Variable „t“ soll eingefügt werden.

GetType() holt den Typ. Die Property selbst holt GetProperty() .

Wenn wir Schreiben dürfen, setzen wir den Wert zu DateTime.Now.

```
PropertyInfo prop = t.GetType().GetProperty("lastupdate", BindingFlags.Public | BindingFlags.Instance);
if (null != prop && prop.CanWrite)
{
    prop.SetValue(t, DateTime.Now, null);
}
```

Hinweis: SQL Server ohne „Kopplung“

Haben wir mehrere SQL Server in der Firma und wollen „mal schnell“ auf einem SQL Server einen Datensatz erstellen, der etwa über einen Trigger den weiteren SQL Server etwas mitteilt, so bricht dies!

Diese SQL Server müssen per DCOM (Distributed Component Object Model) administrativ untereinander verbunden werden, damit Transaktionen über diese SQL Server möglich sind.

Da unsere Aktion als Transaktion läuft – darf dies weiter unten nicht durch locker gekoppelte Datenbankserver ausgehebelt werden.

Das ist in kleineren Firmen ein erheblicher Aufwand – auf den man immer aufpassen muss. Abhilfe wäre, dass man einen Job des SQL Server Agents anstößt, welcher dann die Daten auf den entfernten Server überträgt.

Bei Applikationen wird ab Entity Framework 6.0 das EF nicht mehr vorkompiliert verwendet.

Somit kann man sich überlegen, ob man den Performancevorteil von bis zu einer Sekunde nutzt und dies nach dem Installatieren erledigt.

Beim Installieren kann man etwa mit WIX (Windows Installer Extensions) noch einen Aufruf von NGen.Exe aus dem Windows\.NET\4.5-Verzeichnis durchführen.
Man benötigt während der Installation Admin-Berechtigungen!

Siehe auch:
http://www.c-sharpcorner.com/UploadFile/ff2f08/entity-framework-6-0-ngen-exe-and-startup-performance/

Vorkompilieren per NGen

ClickOnce Applikationen werden ohne lokale Admin Rechte verteilt, dafür ist der Ort schwer herauszubekommen: