

**Dokumentation**

**Order-Management**

**Raphael Wirth & Ricardo Coelho**

Inhalt

[1 Applikation Inbetriebnahme II](#_Toc65311357)

[1.1 Download II](#_Toc65311358)

[1.2 Datenbank II](#_Toc65311359)

[1.3 Testdaten II](#_Toc65311360)

[2 Unser Vorgehen III](#_Toc65311361)

[2.1 Sourcecode Versionierung III](#_Toc65311362)

[2.2 Kanban III](#_Toc65311363)

[3 UI IV](#_Toc65311364)

[3.1 Mockups vs Endergebnis IV](#_Toc65311365)

[3.2 Modal Forms V](#_Toc65311366)

[4 ERM VII](#_Toc65311367)

[4.1 Speziallfall Bills VII](#_Toc65311368)

[5 Klassendiagramm VIII](#_Toc65311369)

[5.1 Links Views VIII](#_Toc65311370)

[5.1.1 Forms VIII](#_Toc65311371)

[5.1.2 Views VIII](#_Toc65311372)

[5.2 Mitte Services IX](#_Toc65311373)

[5.3 Rechts DB-Entities IX](#_Toc65311374)

[5.3.1 Ausnahme IX](#_Toc65311375)

[5.4 Assoziationen X](#_Toc65311376)

[6 CTE – Categories View XI](#_Toc65311377)

[7 YOY Comparison XII](#_Toc65311378)

[7.1 Query XII](#_Toc65311379)

[7.2 Klasse XII](#_Toc65311380)

[8 Bills XIII](#_Toc65311381)

[8.1 Netto Brutto Berechnung XIII](#_Toc65311382)

[9 Rückblick auf das Projekt XIV](#_Toc65311383)

[9.1 Raphael XIV](#_Toc65311384)

[9.2 Ricardo XIV](#_Toc65311385)

# Applikation Inbetriebnahme

## Download

Der Sourcecode von *order-management* wird mit Git verwaltet. Als Plattform wird [www.github.com](http://www.github.com) verwendet. Um den Sourcecode herunter zu laden kann entweder dieser Command ausgeführt werden:

git clone https://github.com/ricardo17coelho/order-management.git

Oder er kann von folgender URL manuell als .zip heruntergeladen werden:

* <https://github.com/ricardo17coelho/order-management>

## Datenbank

*Order-Management* braucht zwingend eine laufende MSSQL-Datenbank, die auf dem gleichen Host läuft, wie die Applikation. Der angemeldete Windows-User muss sich zwingend mit dieser Datenbank verbinden dürfen. (Trusted\_Connection=True)

Bevor man die Applikation startet, muss man die Datenbank vorbereiten. Dafür sollte das Visual Studio Projekt geöffnet werden und in der Package Manager Console folgender Command ausgeführt werden:

Update-Database

Dies generiert die nötigen Tabellen mit allen dazugehörigen Spalten und Beziehungen.

## Testdaten

Dieser Command generiert nicht nur sämtliche Tabellen, sondern auch einige Testdaten, damit *order-management* dann auch getestet werden kann. Dies wurde in der Methode onModelCreating() im DbContext implementiert.

# Unser Vorgehen

## Sourcecode Versionierung

Uns war von Anfang an klar, dass wir den Sourcecode mit Git auf Github verwalten wollten.

## Kanban

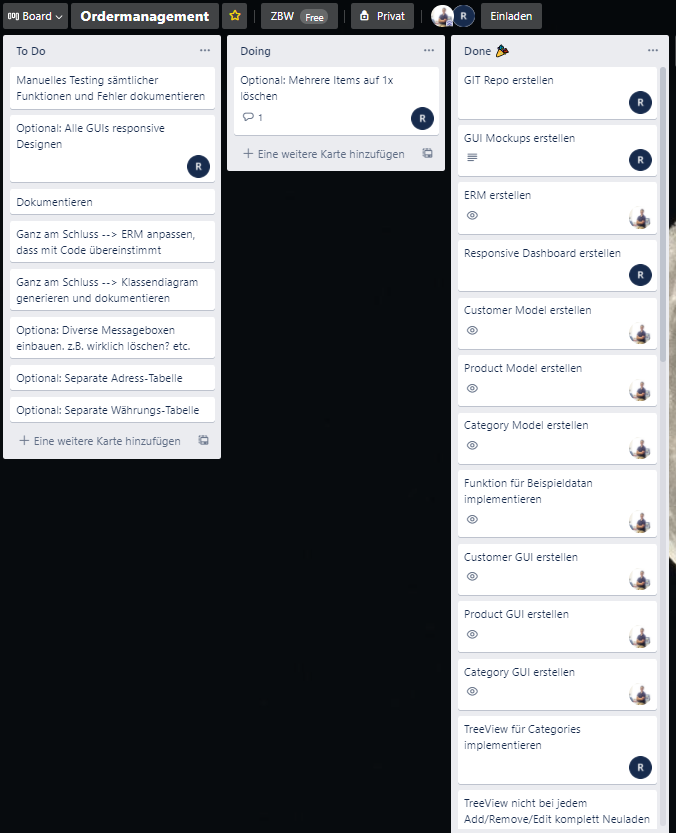
Als zusätzliche Hilfe für die Übersicht der Tasks verwendeten wir ein Kanban-Board auf [www.trello.com](http://www.trello.com).

Anfangs definierten wir einige Tasks und teilten sie einander zu. Die ersten Tasks waren:

* Ricardo: GIT Repo erstellen
* Ricardo: GUI Mockups erstellen
* Raphael: ERM erstellen
* Ricardo: Dashboard (Homescreen) erstellen
* Raphael: Sämtliche Entity Models erstellen

Danach erstellten wir immer weitere Tasks, priorisierten sie und teilten sie einander zu.

Dies ist nur ein kleiner Ausschnitt von unserem Kanban-Board:



# UI

## Mockups vs Endergebnis

Warum abweicht unser Endprodukt von erstellten Mockups?

Am Anfang wollten wir ein einfaches Design für all die Views gestalten und mit Hilfe von verschiedener Farbe, die «views» unterscheiden.

Jedoch haben wir schnell gemerkt, dass WindowsForms sehr limitiert ist.

Sowohl ich als auch Raphael sind daran gewohnt mit guten Frameworks in WEB Bereich zu arbeiten, welche uns viele Möglichkeiten anbieten und deswegen, dachten wir dass wir hier auch umsetzen könnten, aber das war nicht wirklich möglich.

|  |  |  |
| --- | --- | --- |
| **Vor-/Nachteile** | **Mockup** | **Ergebnis** |
| + Schnellzugriff + Zähler - Responsive - Mehrmals gleiche Aktion |  |  |
| + Keine Zusätzliche Fenster  - Unübersichtlich - Responsive |  |  |
| + Keine Zusätzliche Fenster  - Unübersichtlich - Responsive |  |  |
| + Zugriff auf Produkte |  |  |
| + Keine Zusätzliche Fenster  - Unübersichtlich - Responsive |  |  |
| + Keine Zusätzliche Fenster  - Unübersichtlich - Responsive |  |  |
| + Diverse Einstellung auf eine Seite |  |  |

## Modal Forms

Wir haben uns am Angang entschieden, alle Grids + Formulare auf ein Panel darzustellen.

Im Verlauf von der Entwicklung haben wir nun gemerkt, dass das nicht so wirklich übersichtlich ist. Deswegen haben wir uns dann entschieden, all die Formulare in einer separaten Modal Fenster anzuzeigen.

Somit erreichen wir ein besseres Design für unsere Grid und die ADD und DELETE Operationen, können sauber über dieses Modal Dialog gelöst werden.

Beim Klicken auf «Add» wird das Formular gelöscht und eine neuen Datensatz kann erfasst werden.

Wenn 1 Datensatz im Grid ausgewählt ist, dann kann man auf «Edit» Button drücken, das Modal wird geöffnet und die Felder werden mit der ausgewählte Datensatz gefüllt.

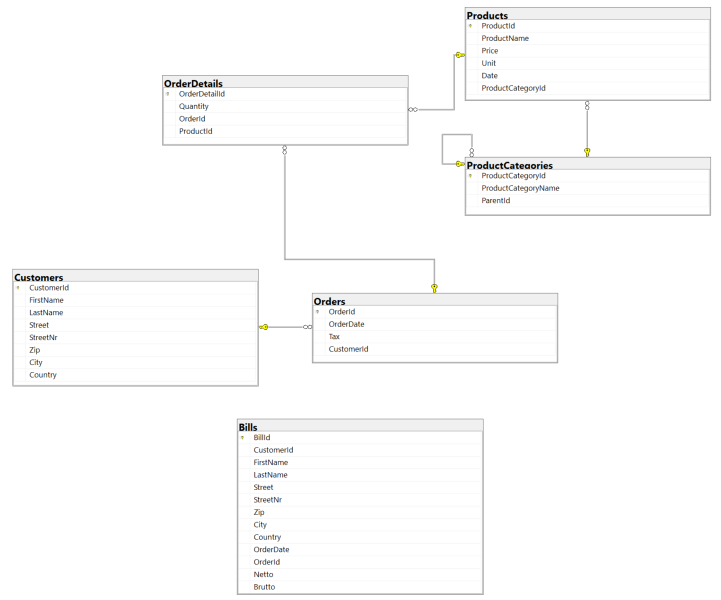
Beim Speichern wird dann beachtet, ob die Datensatz bereits ein ID hat oder nicht.

Falls ja, werden die Felder in der DB geändert, falls nicht, wird eine neue Datensatz in der DB angelegt.

|  |  |
| --- | --- |
| Add Mode | Edit Mode |
|  |  |
|  |  |
|  |  |
|  |  |

# ERM

Aufgrund der Vorgaben, wurde folgendes ERM entworfen und im EF Code-First Ansatz entwickelt.



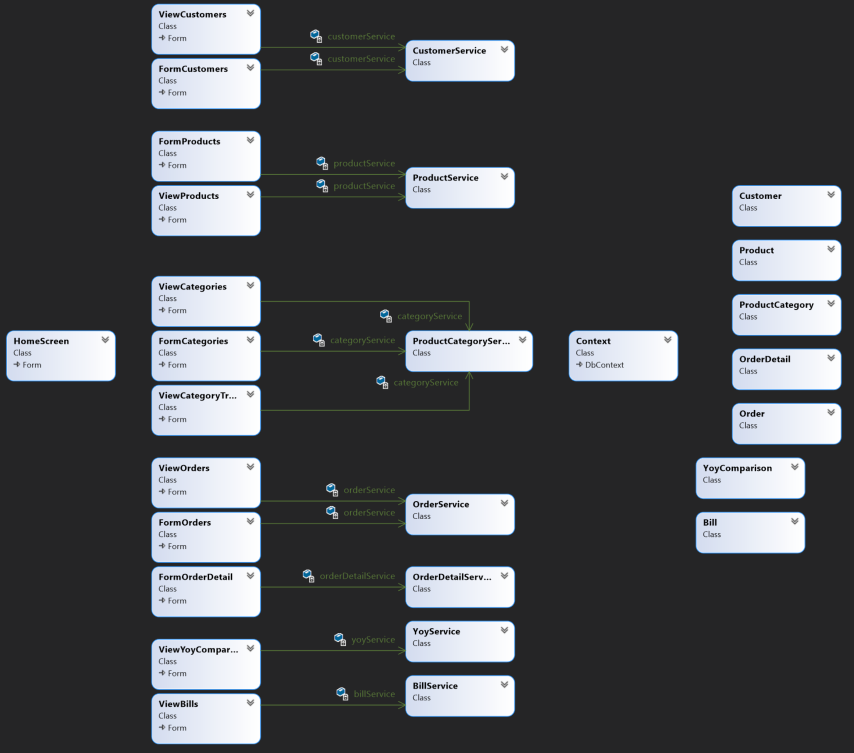
## Speziallfall Bills

Auffällig ist, dass Bills keine Assoziationen zu anderen Tabellen hat. Das heisst hier werden gewisse Daten redundant gespeichert.

Dies ist aber absichtlich so, weil falls z.B. ein Kunde seinen Nachnamen, oder seine Adresse ändert, soll dies KEINE Auswirkungen auf die bereits bestehenden Rechnungen haben. Sondern innerhalb der Rechnungen sollen z.B. Adressänderungen nachvollziehbar sein.

# Klassendiagramm

Folgendes Diagramm zeigt der Aufbau der Applikation.



Das Ganze von links nach rechts in 3 Bereiche aufgeteilt werden.

## Links Views

Auf der linken Seite befinden sich sämtliche Views. Wir unterscheiden immer zwischen Forms und Views (und dem Homescreen, welcher den Eintritt in die Applikation abbildet).

### Forms

Forms sind Formulare, welche Textfelder enthalten, wo der User Daten eingeben kann. Diese Daten werden dann schlussendlich auf der Datenbank abgespeichert.

### Views

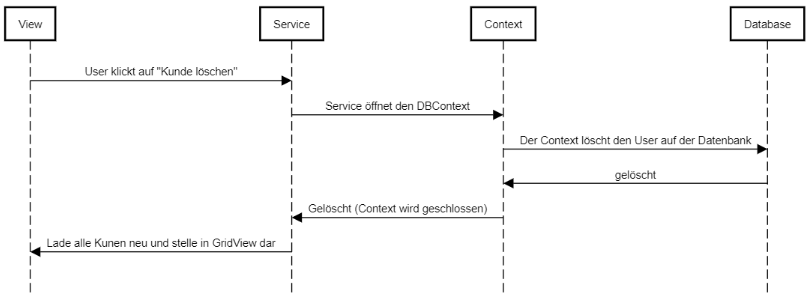
Auf den Views befindet sich jeweils lediglich ein Data-Grid-View welches Daten veranschaulicht.

## Mitte Services

Für jedes Entity gibt es einen dazugehörigen Service. Diese bilden jeweils die Verbindung zur Datenbank ab. Dort drin gibt es jeweils Methoden wie z.B.:

* GetAll()
* GetByName(string name)
* Add(Entity entity)
* Delete(Entity entity)
* Etc.

Jede Methode, die eine Verbindung zur Datenbank benötigt, kümmert sich selbst um das Öffnen und Schliessen vom DBContext, damit dieser immer nur möglichst kurz offen bleibt.



## Rechts DB-Entities

Ganz rechts befinden sich die DB-Entities, welche so auch im ERM wiederzufinden sind.

### Ausnahme

Folgende Klasse ist eine Ausnahme und die findet man so nicht in der Datenbank.

* YoyComparison

In der Methode OnModelCreating() im DbContext wurde definiert, dass für diese Klasse keine Tabelle erzeugt werden soll:

modelBuilder.Entity<YoyComparison>().HasNoKey().ToView(null);

Diese Klasse wird lediglich als «Query-Objekt» verwendet. D.h. Diese Objekte werden nicht in der Datenbank gespeichert, aber wir benötigen sie als Objektstruktur für gewisse Queries und für die Darstellung im ViewYoyComparison.

## Assoziationen

Der Übersichtshalber wurden absichtlich nicht sämtliche Assoziationen eingezeichnet. Konkret gemeint ist damit z.B. das FormProduct.

Gemäss ERM hat jedes Produkt eine dazugehörige Produkt-Kategorie. Das heisst man muss diese beim Erfassen eines Produktes auswählen können. Damit das möglich ist, verwendet das FormProduct nicht NUR den ProductService, sondern auch den ProductCategoryService.

Genau gleich, wie z.B. das FormOrder zusätzlich auch den ProductService UND den CustomerService verwendet etc.

# CTE – Categories View

«No pain, No gain».

So kann ich am besten die CTE beschreiben, nach dem hier viel Zeit investiert wurde.

var query = @";WITH RecurseTable " +

"(ProductCategoryId, ProductCategoryName, ParentId, Level) " +

"AS (SELECT " +

"ProductCategoryId," +

"ProductCategoryName," +

"ISNULL(ParentId, 0)," +

"0 AS Level " +

"FROM ProductCategories " +

"WHERE ParentId IS NULL " +

"UNION ALL " +

"SELECT " +

"pcat.ProductCategoryId," +

"pcat.ProductCategoryName," +

"pcat.ParentId," +

"Level + 1 " +

"FROM ProductCategories AS pcat " +

"INNER JOIN RecurseTable AS rec " +

"ON rec.ProductCategoryId = pcat.ParentId " +

") " +

"SELECT " +

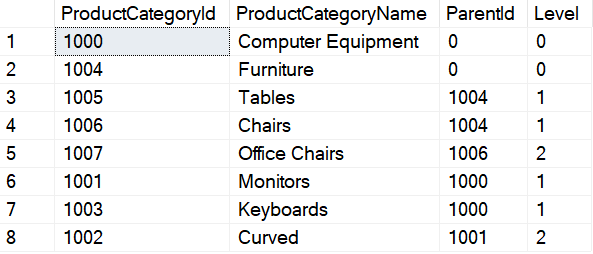
"ProductCategoryId," +

"ProductCategoryName," +

"ParentId," +

"Level " +

"FROM RecurseTable";



Die Aufruft von C# code erfolgt mit als Query string mit Hilfe von EntityFrameworkCore.

Danach erhalten wir als Ergebnis eine Tabelle mit ID der Kategory, Beschreibung, ParentID, welche Identifiziert zu welche übergeordnet Kategorie diese Kategorie gehört und als letzten mit «Level» wird angezeigt zu welchem Level sich diese Kategorie gehört.

Danach wird diese Liste iteriert. Als erstes nimmt man ein Node vom Level 0 und dann werden alle items mit dem gleichen ParentID zu diesen Node hinzugefügt bis alle items in der TreeView hinzugefügt wurden.

# YOY Comparison

## Query

Der Jahresvergleich wurde mit SqlRaw umgesetzt. Es wäre zwar eleganter die Abfrage mit Linq umzusetzen, da wir jetzt diverse «Magic Strings» haben, aber es war im Auftrag nicht vorgegeben und wir wussten nicht genau, wie man so eine Komplexe Abfrage mit Linq nachbilden kann.

SELECT **Concat**(*Datepart*(yyyy, o.orderdate), ' Q', *Datepart*(qq, o.orderdate)) AS  
       Quarter,  
       *Count*(o.orderid)                                                     AS  
       CountOrders,  
       *Avg*(od.quantity)                                                     AS  
       AvgCountProductsPerOrder,  
       *Sum*(od.quantity \* p.price)                                           AS  
       TotalRevenue  
FROM   orders o  
       INNER JOIN orderdetails od  
               ON o.orderid = od.orderid  
       INNER JOIN products p  
               ON p.productid = od.productid  
       INNER JOIN productcategories pc  
               ON pc.productcategoryid = p.productcategoryid  
WHERE  *Datediff*(year, orderdate, *Getdate*()) <= 3  
GROUP  BY **Concat**(*Datepart*(yyyy, o.orderdate), ' Q', *Datepart*(qq, o.orderdate))  
ORDER  BY **Concat**(*Datepart*(yyyy, o.orderdate), ' Q', *Datepart*(qq, o.orderdate))  
          DESC;

## Klasse

Für die einzelnen Spalten wurde eine Klasse erstellt:

public class YoyComparison

{

public string Quarter { get; set; }

public int CountOrders { get; set; }

public int AvgCountProductsPerOrder { get; set; }

public double TotalRevenue { get; set; }

}

Im DBContext haben wir definiert, dass für diese Klasse keine Tabelle generiert werden soll, da wir die Klasse nur als Objekt-Struktur für die Abfrage und für die Darstellung im DataGridView benötigen.

# Bills

Zuerst haben wir das DataGridView für die Bills nur an ein Tabellen-Übergreifendes Query gebunden. Dann haben wir allerdings bemerkt, dass die Anforderung der nachvollziehbaren Adressänderung nicht erfüllt wird.

Weil wenn man die Adresse eines Kunden geändert hat, hat sich auch die Adresse in den Bills geändert, da es ja nur eine Tabellenübergreifende Abfrage war.

Aufgrund dessen haben wir, wie man im ERM sehen kann, eine Tabelle für die Bills erstellt. Somit werden jetzt auch Rechnungen persistiert und man kann Adressänderungen nachvollziehen.

## Netto Brutto Berechnung

Damit wir den Netto und Brutto Preis berechnen können, haben wir bei jedem Order-Objekt eine «Tax» definiert.

double netto = 0;

foreach (var orderDetail in orderDetails)

{

netto += orderDetail.Quantity \* orderDetail.Product.Price;

}

var brutto = netto \* (order.Tax / 100) + netto;

Add(new Bill(customerId, firstName, lastName, street, streetNr, zip, city,

country, orderDate, orderId, netto, brutto));

# Rückblick auf das Projekt

## Raphael

Grundsätzlich bin ich ein grosser Fan von «Learning by Doing». Ich habe es auch in diesem Projekt wieder gemerkt. Mir wurde erst während dem Entwickeln vieles klar, was wir im Unterricht behandelt haben.

Ich hätte es gut gefunden, wenn dieses Projekt benotet werden würde, wie es im 2. Semester der Fall war.

Alles in allem war es ein spannendes Projekt und ich konnte einiges lernen, da ich bis anhin 0 Erfahrung mit OR-Mapper hatte. (Und auch .Net kenne ich nur vom ZBW.)

## Ricardo

Für mich war dieses Projekt zu aufwendig. Ich finde sehr schade, dass nicht benotet wird.

Ich könnte in diesem Projekt den gelernten Stoff, welche in dem Unterricht ermittelt wurde, umsetzen und habe viel dabei gelernt.

Am Anfang hatten wir ein grober Plan, wie viel all die Ansichten darstellen wollten, jedoch hat sich im Verlauf vom Projekt viel geändert.

Da ich nicht so viele Erfahrung mit C#/.Net habe war für mich ein bisschen schwierig all diesen Abhängigkeiten am Anfang zu verstehen.

Gemeinsam mit Raphael habe ich es dann langsam verstanden und wir habe ein «interessante» Lösung entwickelt.