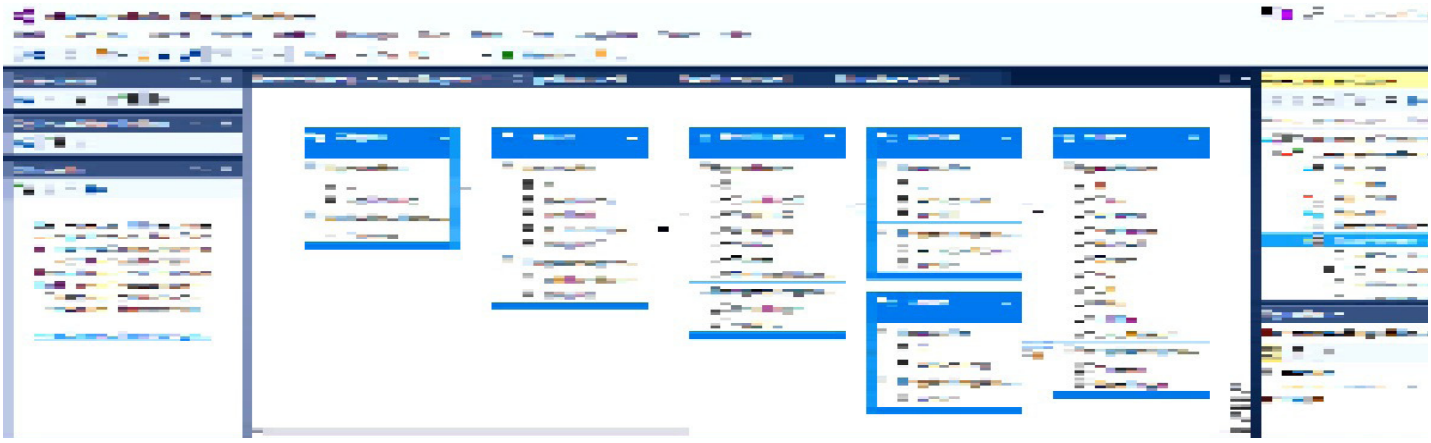
AUSGABE 02/2017

# DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



## TOP-THEMEN:



André Minhorst Verlag

## Impressum

DATENBANKENTWICKLER

Borkhofer Str. 17
47137 Duisburg

Redaktion: Dipl.-Ing. André Minhorst

# Datumsfunktionen

**Ohne Datumsfunktionen kommt man in keiner Programmiersprache aus. Das ist unter C# natürlich nicht anders. Wer allerdings von Access/VBA kommt, muss sich ein wenig umstellen: Einzelne Funktionen wie Date() oder Time(), mit denen Sie beispielsweise das aktuelle Datum oder die aktuelle Uhrzeit ermittelt haben, suchen Sie hier vergeblich. Stattdessen nutzen Sie verschiedene Eigenschaften des Objekts DateTime. Der vorliegende Artikel zeigt die wichtigsten Elemente der C#-Programmiersprache zum Thema Datum und erläutert, wie das Datum hier im Hintergrund behandelt wird.**

## Aktuelles Datum und aktuelle Uhrzeit ausgeben

Als erstes Beispiel wollen wir das aktuelle Datum und die aktuelle Uhrzeit in zwei Textfeldern eines WPF-Fensters ausgeben. Die Steuerelemente dafür definieren wir wie folgt:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="*"></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Label Content="Aktuelles Datum (DateTime.Now.ToShortDateString):"></Label>
    <TextBox x:Name="txtAktuellesDatum" Grid.Column="1"></TextBox>
    <Button x:Name="btnAktuellesDatum" Content="Aktualisieren" Grid.Column="2" Click="btnAktuellesDatum_Click"></Button>
    <Label Content="Aktuelle Zeit (DateTime.Now.ToShortTimeString):" Grid.Row="1"></Label>
    <TextBox x:Name="txtAktuelleZeit" Grid.Row="1" Grid.Column="1"></TextBox>
    <Button x:Name="btnAktuelleZeit" Content="Aktualisieren" Grid.Row="1" Grid.Column="2"
        Click="btnAktuelleZeit_Click"></Button>
</Grid>
```

Dies ist das Grundgerüst, weitere Elemente rüsten wir im Laufe dieses Artikels nach. In diesem Fall haben wir ein Grid aufgebaut, dass drei Spalten enthält – eines mit der Bezeichnung, eines mit dem per C# ermittelten Inhalt und eines mit einer Schaltfläche, um den Inhalt des Textfeldes zu aktualisieren oder andere Funktionen auszuführen.

Das obere Textfeld soll das aktuelle Datum anzeigen, das untere die aktuelle Uhrzeit. Die Beschriftungen verraten bereits die dort verwendeten C#-Funktionen. In der Konstruktormethode **MainWindow()** füllen wir die Textfelder gleich beim Anzeigen des Fensters:

```
public MainWindow() {
    InitializeComponent();
    txtAktuellesDatum.Text = DateTime.Now.ToShortDateString();
    txtAktuelleZeit.Text = DateTime.Now.ToLongTimeString();
}
```

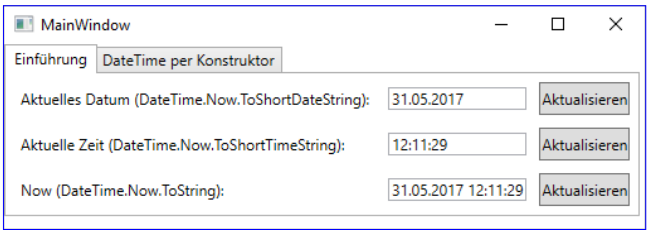



**Bild 1:** Ausgabe in Textfeldern

Hier sehen Sie, dass wir das aktuelle Datum mit dem Ausdruck **DateTime.Now.ToShortDateString()** ermitteln. Dabei ist **DateTime** das zentrale Objekt, wenn es sich um Datum- und Zeitangaben dreht. Nach diesem einführenden Beispiel schauen wir uns an, was die **DateTime**-Klasse alles bietet. Das Ergebnis sieht wie in Bild 1 aus.

## Die DateTime-Klasse

Während wir unter VBA einfach Funktionen wie **Date()** oder **Time()** nutzen konnten, um die aktuelle Uhrzeit zu ermitteln, benötigen wir unter C# natürlich die Methoden eines Objekts für diesen Zweck – beziehungsweise die statischen Methoden einer Klasse. In diesem Fall nutzen wir die Methode **Now** der Klasse **DateTime**, um auf das aktuelle Datum und die aktuelle Uhrzeit zuzugreifen, also **DateTime.Now**. Allerdings reicht dies noch nicht aus, denn wir benötigen noch ein Ausgabeformat. **DateTime.Now** liefert nämlich eine Struktur des Typs **DateTime** zurück, deren Inhalt so nicht direkt ausgegeben werden kann.

Dazu nutzen wir dann wie im obigen Beispiel etwa die Methoden **ToShortDateString()** oder **ToLongTimeString()**.

## Datentyp für das Datum und Ticks

Der Datentyp zum Speichern von Datumswerten heißt unter C# **DateTime**. Genaugenommen handelt es sich um eine Struktur, mit der Sie Zeitangaben vom 1.1.0001 bis zum 31.12.9999 darstellen können. Intern wird das Datum als **long**-Wert in der Einheit **Tick** gespeichert. Ein Tick entspricht hundert Nanosekunden – das sollte für die meisten Anwendungsfälle ausreichend genau sein. Damit steht die interne Speicherung im Gegensatz zu der etwa unter VBA: Hier wird ein Datum als Fließkommazahl gespeichert, wobei der Teil vor dem Komma dem Datum und der Teil nach dem Komma der Uhrzeit entspricht.

Neben den Ticks als **long**-Wert enthält **DateTime** auch noch die **Kind**-Eigenschaft, welche speichert, ob es sich um ein lokales Datum, ein UTC-Datum (mehr dazu weiter unten) oder ein unspezifisches Datum handelt.

Die verschiedenen Darstellungen wie etwa langes Datum, kurzes Datum, lange Uhrzeit, kurze Uhrzeit oder auch Kombinationen sind lediglich Umrechnungen der gespeicherten Anzahl Ticks in einen Datumswert und die Ausgabe als String.

## Statische Funktionen der DateTime-Klasse

Die **DateTime**-Klasse liefert eine Reihe Funktionen rund um Datumsangaben, die Sie kennen sollten:

**Compare**: Erwartet zwei **DateTime**-Objekte und gibt folgende Werte zurück: **-1**, wenn das erste Datum kleiner als das zweite Datum ist, **0**, wenn beide Daten genau gleich sind und **1**, wenn das zweite Datum kleiner als das erste Datum ist.

**DaysInMonth**: Erwartet die Angabe von Jahr und Monat als Parameter und liefert die Anzahl der Tage dieses Monats zurück. Beispiel: **DaysInMonth(2017, 1)** liefert **31**.

**Equals**: Gibt für die beiden als Parameter übergebenen **DateTime**-Objekte an, ob diese gleich sind. Beispiel: **DateTime.Equals(new DateTime(2017, 1, 1), new DateTime(2017, 1, 1)).ToString()** liefert **true**.

**FromBinary**: Liest ein Datum von einer binären Repräsentation eines Datums im **long**-Format wieder in ein Datum ein. Dazu benötigen Sie zunächst die binäre Form, welche wir im Vorgriff auf die Methoden von Objekten auf Basis der Klasse **DateTime** mit der Methode **ToBinary** erstellen. Wir erstellen hier ein Datum mit **Now** und speichern seine mit **ToBinary()** erstellte binäre Repräsentation in der Variablen **lngDat**. Dann geben wir das Originaldatum, die binäre Variante und das aus dieser Variante zurückverwandelte **DateTime**-Objekt aus:

```
DateTime dat = DateTime.Now;
long lngDat = dat.ToBinary();
Console.WriteLine("Orig. Datum: {0}", dat.ToString());
Console.WriteLine("Datum binär: {0}", lngDat.ToString());
Console.WriteLine("FromBinary:  {0}", DateTime.FromBinary(lngDat).ToString());
```

**FromFileTime**: Wie FromBinary, allerdings wird hier eine mit der Methode ToFileTime erstellte Anzahl von Ticks wieder zurückverwandelt.

**FromFileTimeUTC**: Wie **FromFileTime**, nur dass die mit der Methode **ToFileTimeUTC** erstellte Anzahl von Ticks für eine UTC-Zeit wieder zurückverwandelt wird.

**FromOADate**: Wandelt ein Datum, dass unter VBA/OLE erstellt wurde (also die Repräsentation durch eine Fließkommazahl, deren Vorkommastellen das Datum und deren Dezimalstellen die Uhrzeit markieren), in ein **DateTime**-Objekt um.

**IsLeapYear**: Gibt an, ob es sich bei dem als Parameter übergebenen Jahr um ein Schaltjahr handelt. Beispiel: **DateTime.IsLeapYear(2016).ToString()** liefert **true**.

**MaxValue**: Liefert den größtmöglichen Wert eines **DateTime**-Objekts. Beispiel: **DateTime.MaxValue().ToString()** liefert **31.12.9999 23:59:59**.

**MinValue**: Liefert den kleinstmöglichen Wert eines **DateTime**-Objekts. Beispiel: **DateTime.MinValue().ToString()** liefert **01.01.0001 00:00:00**.

**Now**: Liefert ein **DateTime**-Objekt mit dem aktuellen Datum und der aktuellen Uhrzeit. Das **DateTime**-Objekt offeriert weitere Methoden (siehe unten).

**Parse**: Erwartet eine Zeichenkette mit einem Datum, welches dann in ein **DateTime**-Objekt konvertiert wird, zum Beispiel **DateTime.Parse("1.2.2017 10:11:12").ToString()**.

**ParseExact**: Wie Parse, nur dass hier noch das genaue Format vorgegeben werden muss: **DateTime.ParseExact("01.02.2017 10:11:12", "dd.MM.yyyy hh:mm:ss", null).ToString();**

**ReferenceEquals**: Vergleicht zwei Instanzen und gibt an, ob diese gleich sind. Beispiel: **DateTime.ReferenceEquals(new DateTime(2017,1,1), new DateTime(2017,1,1)).ToString()** liefert **false**, da es sich ja um zwei unabhängig erstellte Instanzen handelt.

**SpecifyKind**: Art des **DateTime**-Objekts festlegen. Der erste Parameter nimmt das **DateTime**-Objekt entgegen, wobei es sich um ein bestehendes oder auch um ein neu erstelltes **DateTime**-Objekt handeln kann. Der zweite Parameter erwartet ein Element der Enumeration **DateTimeKind**, also etwa **UTC**, **Local** oder **Unspecified**.

**Today**: Liefert ein **DateTime**-Objekt mit dem aktuellen Datum. Beispiel: **DateTime.Today.ToString()**

**TryParse**: Erwartet eine Zeichenkette mit einem Datum, welches in ein **DateTime**-Objekt konvertiert wird, das mit dem zweiten Parameter zurückgegeben wird. Die Funktion liefert den Wert **true** zurück, wenn die Zeichenkette geparst werden konnte, sonst false.

**TryParseExact**: Wie **TryParse**/**ParseExact**.

**UTCNow**: Liefert ein **DateTime**-Objekt, welches das aktuelle Datum und die Uhrzeit für die koordinierte Weltzeit enthält, also für die Zeitzone, in der sich Greenwich/London befindet.

### Zur koordinierten Weltzeit

Die von der Funktion **UTCNow** gelieferte koordinierte Weltzeit liefert die aktuelle Uhrzeit plus Datum aus Greenwich (früher GMT/Greenwich Mean Time, jetzt UCT/Universal Coordinated Time). Wozu benötigt man diese Zeit, die sich zum Beispiel während der Sommerzeit um zwei Stunden von der Zeit in Deutschland unterscheidet? Wenn Sie eine Datenbank nutzen, die von Benutzern in verschiedenen Zeitzonen verwendet wird, dann sollten Sie Datumsangaben in der Datenbank in dieser Zeit speichern.

Alternativ müssen Sie die Zeitzone in einem separaten Feld speichern, damit Sie später beispielsweise nachvollziehen können, in welcher Reihenfolge Bestellungen eingegangen sind. Sonst würde eine Bestellung, die um 12:00 Uhr lokaler Zeit in Deutschland eingegangen ist gegenüber einer Bestellung, die um 9:00 Uhr lokaler Zeit an der amerikanischen Ostküste eingegangen ist, in der falschen Reihenfolge gespeichert werden – vorausgesetzt, die Zeitzone wurde nicht mitgespeichert. Wenn Sie die Zeiten in UTC-Zeit umwandeln, würde 12:00 Uhr deutscher Zeit 10:00 Uhr UTC entsprechen und 9:00 Uhr lokaler Zeit an der amerikanischen Ostküste wäre 13:00 Uhr UTC.

### DateTime-Objekte per Konstruktor erstellen

Neben den Methoden wie **Now**, **Today** oder **UTCNow**, die automatisch das aktuelle Datum beziehungsweise die aktuelle Uhrzeit ermitteln, und Methoden wie **Parse** oder **TryParse**, denen Sie einen Datumsstring übergeben, der in ein **DateTime**-Objekt umgewandelt werden soll, gibt es noch verschiedene Konstruktor-Methoden der **DateTime**-Klasse. Das heißt, dass Sie die dem Datum und/oder der Uhrzeit zugrundeliegenden Informationen beim Initialisieren als Parameter übergeben.

Dazu haben wir ein kleines Beispiel erstellt, dass Sie auf der Registerseite **DateTime per Konstruktor** des Beispielprojekts finden. Hier kann der Benutzer Tag, Monat und Jahr eingeben und erhält das Datum in der kurzen Version. Dazu ermitteln wir den

**int**-Wert des jeweiligen Inhalts der Textfelder mit der **Parse**-Methode und übergeben Jahr, Monat und Tag dann dem Konstruktor von **DateTime**. Dieser landet dann wiederum im entsprechenden Textfeld:

```
private void btnDateTime_TagMonatJahr_Click(object sender, RoutedEventArgs e) {
    int tag = int.Parse(txtTag_TagMonatJahr.Text);
    int monat = int.Parse(txtMonat_TagMonatJahr.Text);
    int jahr = int.Parse(txtJahr_TagMonatJahr.Text);
    DateTime datAktuell = new DateTime(jahr, monat, tag);
    txtDateTime_TagMonatJahr.Text = datAktuell.ToShortDateString();
}
```

Das Gleiche erledigen wir direkt darunter mit der Uhrzeit. Dabei müssen wir auf jeden Fall auch ein gültiges Datum angeben, da es keine Überladung des Konstruktors gibt, der die Uhrzeit in Stunden, Minuten und Sekunden annimmt, aber kein Datum.

Hier verwenden wir **ToLongTimeString**, damit auch die Sekunden angezeigt werden:

```
private void btnDateTime_StundeMinuteSekunde_Click(object sender, RoutedEventArgs e) {
    int stunde = int.Parse(txtStunde_StundeMinuteSekunde.Text);
    int minute = int.Parse(txtMinute_StundeMinuteSekunde.Text);
    int sekunde = int.Parse(txtSekunde_StundeMinuteSekunde.Text);
    DateTime datUhrzeit = new DateTime(1, 1, 1, stunde, minute, sekunde);
    txtDateTime_StundeMinuteSekunde.Text = datUhrzeit.ToLongTimeString();
}
```

Die verschiedenen Überladungen des Konstruktors nehmen die folgenden Parameter entgegen (die Varianten mit Umwandlungen in andere Kalender als den gregorianischen lassen wir weg):

- Kein Parameter: Liefert das Datum 01.01.0001 00:00:00.
- Anzahl der Ticks (long): Speichert direkt den **long**-Wert in **DateTime**.
- Ticks und **DateTimeKind**: Speichert die Ticks unter Anwendung eines weiteren Parameters, der angibt, ob die Zeit als lokale Zeit (**DateTimeKind.Local**, **2**) oder als UTC-Zeit (**DateTimeKind.UTC**, **1**) gespeichert werden soll.
- Jahre, Monate, Tage
- Jahre, Monate, Tage, Stunden, Minuten, Sekunden
- Jahre, Monate, Tage, Stunden, Minuten, Sekunden und **DateTimeKind**
- Jahre, Monate, Tage, Stunden, Minuten, Sekunden, Millisekunden

- Jahre, Monate, Tage, Stunden, Minuten, Sekunden, Millisekunden und **DateTimeKind**

### Funktionen der DateTime-Struktur

Nach ein paar einführenden Beispielen zum grundlegenden Umgang mit den Funktionen eines **DateTime**-Objekts hier nun die Übersicht der verschiedenen Methoden dieser Klasse:

**Add**: Fügt einer **DateTime** einen Wert hinzu. Dazu übergeben Sie der **Add**-Methode eines der Elemente **FromDays**, **FromHours**, **FromMilliSeconds**, **FromMinutes**, **FromSeconds** oder **FromTicks** und geben dafür wiederum in Klammern den zu addierenden Wert dieser Einheit an. Wenn Sie also dem aktuellen Datum drei Tage hinzufügen wollen, können Sie erst das aktuelle Datum in **dat** schreiben, dann mit **Add(TimeSpan.FromDays(3))** drei Tage hinzufügen und das Ergebnis in **dat2** speichern und dieses dann ausgeben:

```
DateTime dat = DateTime.Now;
DateTime dat2 = dat.Add(TimeSpan.FromDays(3));
MessageBox.Show(dat2.ToString());
```

Dies würde also der Funktion **DateAdd** von VBA entsprechen, wobei der erste Parameter durch die entsprechende Funktion wie FromDays entspricht:

```
? DateAdd("d", 3, Date())
```

**AddDays**: Dies ist eine Alternative zur **Add**-Methode mit Verwendung von **FromDays**. Hier geben Sie einfach nur die Anzahl der zu addierenden Tage als Parameter an, die Einheit wird ja durch **AddDays** bereits explizit angegeben:

```
DateTime dat = DateTime.Now;
MessageBox.Show(dat.AddDays(3).ToString());
```

In diesem Fall wird zwar das Datum ausgegeben, dass um drei Tage später als das angegebene Datum liegt, aber es wird – genau wie bei der **Add**-Methode von **DateTime** – nicht in dem Datum gespeichert, welches die **AddDays**-Methode anwendet. Dazu müssten Sie noch eine Zuweisung wie die folgende vornehmen:

```
dat = dat.AddDays(3);
```

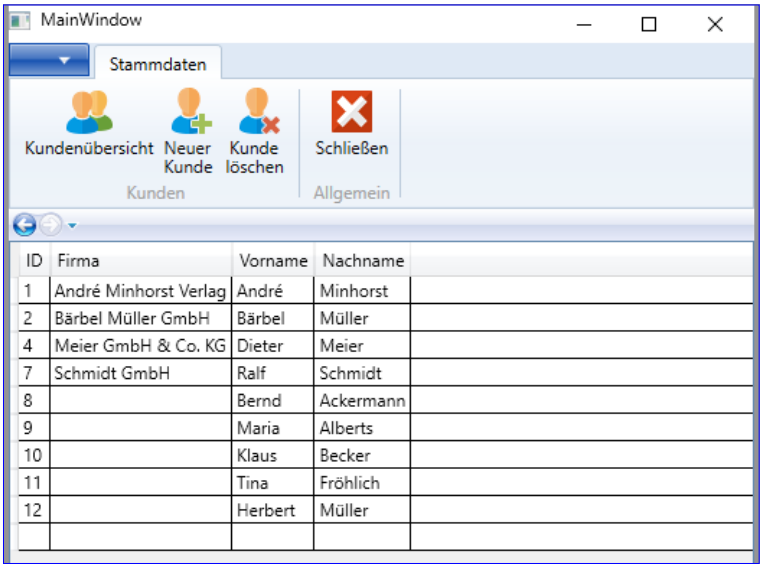
**AddHours**, **AddMilliseconds**, **AddMinutes, AddMonths**, **AddSeconds**, **AddTicks**, **AddYears**: Diese Funktionen fügen die als Parameter angegebene Anzahl der im Funktionsnamen angegebenen Einheit hinzu. Interessant ist hier, dass die **Add...**-Funktionen für alle Einheiten zur Verfügung stehen. Für **Add** stehen die beiden Funktionen **FromYears** und **FromMonths** nicht zur Verfügung. Grundsätzlich sind die **Add...**-Funktionen einfacher anzuwenden als **Add(TimeSpan.From...)**. Es gibt aber eine Ausnahme, die wir weiter unten unter **Das TimeSpan-Objekt** erläutern.

**CompareTo**: Die **CompareTo**-Methode einer Instanz der Klasse **DateTime** funktioniert ähnlich wie die statische **CompareTo**-Methode der **DateTime**-Klasse. Allerdings rufen Sie diese ja als Methode eines **DateTime**-Objekts auf, das Sie zuvor initialisiert

# DataGrid im Detail

**Bisher haben wir das DataGrid immer in einer Minimalausführung verwendet – also mit einfachen Text-Steuerelementen. Das sieht nicht nur langweilig aus, sondern es nutzt auch die Fähigkeiten des DataGrid-Steuerelements nicht annähernd aus. Also schauen wir uns in diesem Artikel an, welche Möglichkeiten der Formatierung des DataGrid-Steuerelements wie bisher in unserer Lösung »Bestellverwaltung« eingesetzt haben.**

Um genau zu sein, haben wir das DataGrid in den bisherigen Beispielen so genutzt wie ein einfaches **ListView**-Steuerelement – wir haben ihm eine Datenherkunft zugewiesen und die enthaltenen Daten einfach in den einzelnen Spalten des Steuerelements angezeigt (siehe Bild 1).




| ID | Firma | Vorname | Nachname |
|---|---|---|---|
| 1 | André Minhorst Verlag | André | Minhorst |
| 2 | Bärbel Müller GmbH | Bärbel | Müller |
| 4 | Meier GmbH & Co. KG | Dieter | Meier |
| 7 | Schmidt GmbH | Ralf | Schmidt |
| 8 | | Bernd | Ackermann |
| 9 | | Maria | Alberts |
| 10 | | Klaus | Becker |
| 11 | | Tina | Fröhlich |
| 12 | | Herbert | Müller |

**Bild 1:** Einfaches DataGrid-Steuerelement

### Hinweis

Die folgenden Beispiele basieren auf der Verwendung eines Entity Data Models der Datenbank **Bestellverwaltung** als Datenquelle.

### Datenquellen

Normalerweise zeigt das DataGrid automatisch alle Spalten und alle Zeilen der Datenquelle an. Eine individuelle Anzeige erreichen Sie, indem Sie zunächst die automatische Anzeige aller Spalten deaktivieren. Dazu stellen Sie das Attribut **AutoGenerateColumns** auf **False** ein. Sie müssen dann individuelle Spalten angeben – siehe weiter unten.

### Rasterlinien

Die Rasterlinien werden standardmäßig als dünne, schwarze Linien angezeigt. Die generelle Sichtbarkeit können Sie mit dem Attribut **GridLinieVisibility** beeinflussen, das die Werte **All**, **Horizontal**, **None** und **Vertical** entgegennimmt. Die Farbe der Rasterlinien lässt sich mit den Eigenschaften **HorizontalGridLinesBrush** und **VerticalGridLinesBrush** individuell einstellen:

```
<DataGrid ItemsSource="{Binding Kunden}" HorizontalGridLinesBrush="LightGray" VerticalGridLinesBrush="LightGray">
```

### Standardfunktionen

Das DataGrid bietet schon in der Grundausstattung einige Funktionen, die etwa in der Datenblattansicht von Access enthalten sind – aber anders genutzt werden können. Diese Möglichkeiten können Sie nach Bedarf einschränken, indem Sie das entsprechende in Klammern angegebene Attribut auf den Wert **False** einstellen:

- Sortieren der Zeilen nach dem Inhalt einer Spalte. Dazu klicken Sie mit der Maus auf den Kopf der Spalte, nach welcher die Daten sortiert werden sollen. Der erste Klick sortiert die Daten in aufsteigender, der nächste in absteigender Reihenfolge (**CanUserSortColumns**).

- Sortieren der Spalten: Sie können die Reihenfolge der Spalten ganz einfach per Drag and Drop einstellen (**CanUserReorderColumns**).

- Ändern der Spaltenbreiten: Per Ziehen des Trennstrichs rechts von einem Spaltenkopf ändern Sie die Breite dieser Spalte (**CanUserResizeColumns**).

- Außerdem können Sie auch die Zeilenhöhe anpassen, indem Sie auf den Strich unter dem grauen Bereich links von der ersten Spalte nach oben oder unten ziehen (**CanUserResizeRows**).

### Textspalten

Wenn Sie **AutoGenerateColumns** auf **False** eingestellt haben, müssen Sie selbst angeben, welche Steuerelementtypen für welches Feld der Datenherkunft im DataGrid angezeigt werden sollen. Wenn die Spalten automatisch generiert werden, kommt das zum Datentyp der anzuzeigenden Eigenschaft zum Einsatz. Wenn Sie die folgenden vier Felder der Datenherkunft anzeigen wollen, würden Sie dazu vier **DataGridTextColumn**-Elemente verwenden:

```
<DataGrid ItemsSource="{Binding Kunden}" AutoGenerateColumns="False" HorizontalGridLinesBrush="LightGray"
        VerticalGridLinesBrush="LightGray">
    <DataGrid.Columns>
        <DataGridTextColumn Binding="{Binding ID}" Header="ID"></DataGridTextColumn>
        <DataGridTextColumn Binding="{Binding Firma}" Header="Firma"></DataGridTextColumn>
        <DataGridTextColumn Binding="{Binding Vorname}" Header="Vorname"></DataGridTextColumn>
        <DataGridTextColumn Binding="{Binding Nachname}" Header="Nachname"></DataGridTextColumn>
    </DataGrid.Columns>
</DataGrid>
```

Dabei geben Sie mit **Binding="{Binding <Feld>}"** das zu bindende Feld der Datenquelle an und mit **Header** die Spaltenüberschrift.

### Andere Spaltentypen

Es gibt noch eine ganze Reihe weiterer Spaltentypen, mit denen Sie Felder mit anderen Datentypen besser als mit einem Textfeld anzeigen können:

- **DataGridCheckBoxColumn**: Kontrolllkästchen, zeigt **True** (beziehungsweise **1**) als markiert und **False** (**0**) als nicht markiert an – siehe weiter unten.

- **DataGridComBoxColumn**: Kombinationsfeld. Dieses Element hat, wie weiter unten sehen werden, gerade bei der Datenbindung eine Besonderheit.

- **DataGridHyperlinkColumn**: Zeigt den enthaltenen Text als Link an.

- **DataGridTemplateColumn**: Spaltentyp zum Definieren eigener Vorlagen, auch mit anderen Steuerelementen als **TextBox**, **CheckBox**, **ComboBox** und **Hyperlink**

## DataGrid mit CheckBox

Auf der Seite **Produktuebersicht.xaml** der Beispielanwendung **Bestellverwaltung** haben wir ein Kontrollkästchen zum DataGrid hinzugefügt. Die Definition sieht wie folgt aus, die Ansicht zur Laufzeit finden Sie in Bild 2:

```
<DataGrid x:Name="dgProdukte" Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="2" ItemsSource="{Binding Produkte}"
AutoGenerateColumns="false" CanUserAddRows="False">
    <DataGrid.Columns>
        <DataGridTextColumn Binding="{Binding Path=ID}" Header="ID"  />
        <DataGridTextColumn Binding="{Binding Path=Bezeichnung}" Header="Produkt" />
        <DataGridCheckBoxColumn Binding="{Binding Path=Lieferbar}" Header="Lieferbar" />
    </DataGrid.Columns>
    ...
</DataGrid>
```

## DataGrid mit ComboBox

Während die Definition von Textfeldern und Kontrollkästchen im DataGrid relativ einfach ist, wird es beim Kombinationsfeld etwas komplizierter: Hier können Sie nämlich nicht einfach die Eigenschaften wie **ItemsSource**, **SelectedItem**, **DisplayMemberPath** oder **SelectedValuePath** verwenden. Schauen wir uns die Grundlagen für die Anzeige etwa der Kategorien in der Seite **Produktuebersicht.xaml** an. Hier benötigen wir in der Code behind-Klasse eine Liste mit den Kategorien:

```
private List<Kategorie> kategorien;
public List<Kategorie> Kategorien {
    get { return kategorien; }
    set { kategorien = value; }
}
```

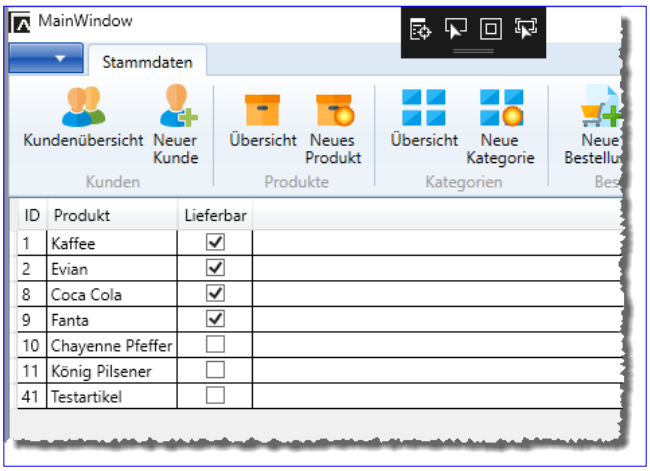



**Bild 2:** DataGrid mit Kontrollkästchen

Diese Liste müssen Sie natürlich noch initialisieren, was in der Konstruktor-Methode geschieht:

```
public Produktuebersicht(long kundeID = 0) {
    ...
    kategorien = new List<Kategorie>(dbContext.Kategorien);
}
```

Nun kommt der interessante Teil, nämlich die Definition des Kombinationsfeldes in der **xaml**-Datei:

```
<DataGridComboBoxColumn Header="Kategorie" SelectedValueBinding="{Binding KategorieID}" DisplayMemberPath="Bezeichnung"
        SelectedValuePath="ID">
    <DataGridComboBoxColumn.ElementStyle>
```

```
        <Style TargetType="{x:Type ComboBox}">
            <Setter Property="ItemsSource" Value="{Binding Path=DataContext.Kategorien,
                RelativeSource={RelativeSource AncestorType={x:Type Page}}}" />
        </Style>
    </DataGridComboBoxColumn.ElementStyle>
    <DataGridComboBoxColumn.EditingElementStyle>
        <Style TargetType="{x:Type ComboBox}">
            <Setter Property="ItemsSource" Value="{Binding Path=DataContext.Kategorien,
                RelativeSource={RelativeSource AncestorType={x:Type Page}}}" />
        </Style>
    </DataGridComboBoxColumn.EditingElementStyle>
</DataGridComboBoxColumn>
```

Wie Sie sehen, bringen wir die Attribute **SelectedValueBinding** (zum Binden an das Feld **KategorieID** der übergeordneten Datenquelle), **DisplayMemberPath** und **DisplayValuePath** noch direkt im Element **DataGridComboBoxColumn** unter – also ganz so, als ob wir ein **ComboBox**-Element in einer Detail-Ansicht verwenden. Die Quelle der Einträge für das Kombinationsfeld können wir jedoch nicht einfach als **ItemsSource** angeben.

Stattdessen verwenden wir zwei **DataGridComboBoxColumn.ElementStyle**-Elemente, mit denen wir die Eigenschaft **ItemsSource** einstellen, und zwar auf das Element **DataContext.Kategorien** des übergeordneten **Page**-Elements.

Das ist prinzipiell identisch mit der Vorgehensweise über das Attribut **ItemsSource**, allerdings etwas kompliziert formuliert – was jedoch am Design von WPF liegt und schlicht nicht anders funktioniert.

Beachten Sie, dass wenn Sie das DataGrid in einem **Window**- statt in einem **Page**-Element anzeigen, für das Attribut **AncestorType** den Wert **{x:Type Window}** angeben. Das Ergebnis sieht schließlich wie in Bild 3 aus.

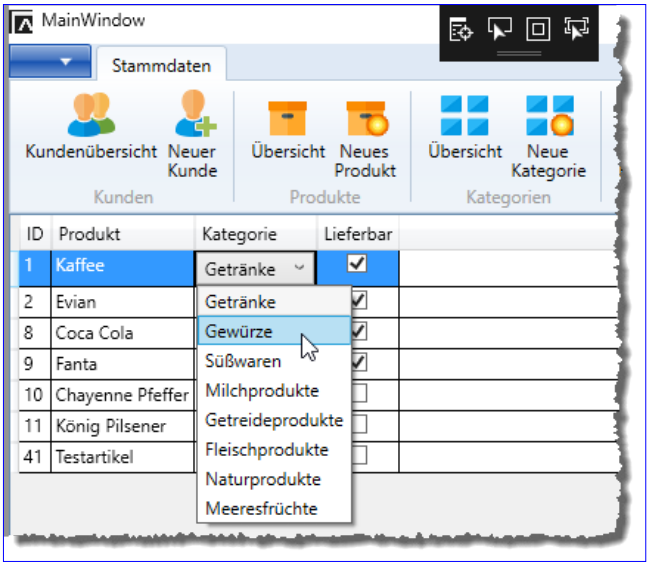



**Bild 3:** DataGrid mit Kombinationsfeld

| ID | Firma | Vorname | Nachname | Kunde |
|---|---|---|---|---|
| 1 | André Minhorst Verlag | André | Minhorst | Minhorst, André |
| 2 | Bärbel Müller GmbH | Bärbel | Müller | Müller, Bärbel |
| 4 | Meier GmbH & Co. KG | Dieter | Meier | Meier, Dieter |
| 7 | Schmidt GmbH | Ralf | Schmidt | Schmidt, Ralf |
| 8 | | Bernd | Ackermann | Ackermann, Bernd |
| 9 | | Maria | Alberts | Alberts, Maria |
| 10 | | Klaus | Becker | Becker, Klaus |
| 11 | | Tina | Fröhlich | Fröhlich, Tina |
| 12 | | Herbert | Müller | Müller, Herbert |

**Bild 4:** DataGrid mit zwei verknüpften Feldern

## Felder verknüpfen mit MultiBinding

Wenn Sie einmal zwei Felder der Datenherkunft eines **DataGrids** verknüpfen wollen, kann das wie in Bild 4 aussehen. In einem Textfeld unter Access/VBA ist so etwas einfach: Man gibt einfach die zu verknüpfenden Elemente mit den entsprechenden Textverkettungssymbolen ein, also etwa **=Nachname & ", " & Vorname**. Das gelingt unter WPF/C# leider

# Webservice testen am Beispiel von DHL

**Wer einen Webshop betreibt, kennt sich zumindest mit den Webseiten des einen oder anderen Paketdienstes aus. Dabei gibt es beim Erstellen von Versandetiketten für Pakete verschiedene Stufen der Automatisierung. Wir wollen zur Stufe der höchsten Automatisierung gelangen und unsere Daten für den Versand direkt an den Webservice eines Anbieters, in diesem Fall DHL, weiterleiten und die erstellten Versandetiketten in druckbarer Form herunterladen. Dazu erfahren Sie in diesem Artikel, wie Sie einen Webservice mithilfe des Tools SoapUI testen.**

## Von der Handarbeit zur Vollautomatisierung

Wer es nicht so mit dem Internet hat, holt sich ein paar Paketscheine von der Post, füllt diese händisch mit den Adressen der Empfänger aus und sendet diese dann ab. Bei kleinen Mengen mag das ein sinnvoller Ansatz sein. Wer hingegen gleich ein paar Lieferungen pro Tag verschickt, gibt die Daten vielleicht direkt in die Eingabemaske der Webseite **dhl.de** ein. Hier können Sie alle relevanten Daten angeben – von der Paketgröße über das Gewicht bis hin zu den Absender- und Empfängerdaten. Wer regelmäßig Pakte versendet, möchte aber vielleicht noch etwas Geld sparen und nutzt den Geschäftskundentarif statt den für Privatkunden. Dann ist **www.dhl-geschaeftskundenportal.de** die richtige Adresse. Hier benötigen Sie allerdings ein Kundenkonto – auf die Schnelle ein Paket verschicken und per Paypal bezahlen können Sie hier nicht. Als registrierter Kunde loggen Sie sich dann auf der Seite ein und können über den Menüpunkt **Versenden|Versenden (Intraship)** zur Versandschnittstelle wechseln (siehe Bild 1).

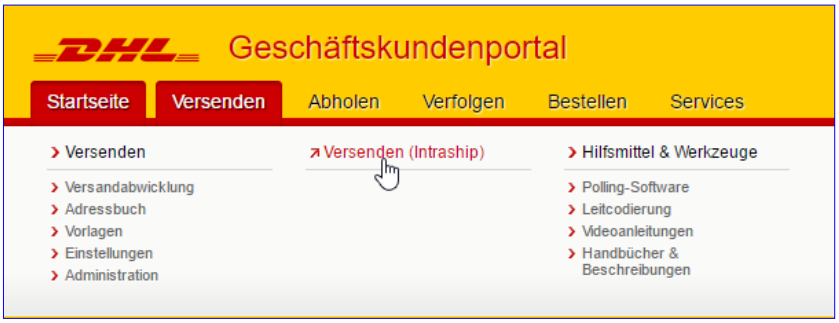



**Bild 1:** Aufruf des Dienstes **Intraship**

Hinweis: Zum Testen der in späteren Beispielen vorgestellten Zugriffe auf den Webservice von DHL benötigen Sie ein Entwickler-Konto bei DHL. Wie Sie dieses anlegen, erfahren Sie weiter unten.


**Bild 2:** Versandschnittstelle

Hier haben Sie dann einige weitere Möglichkeiten. Dazu gehört das Anlegen eines neuen Auftrags, wobei Sie allerdings nur das Gewicht und die Empfängeradresse eintragen müssen – die Absenderadresse für Ihr Kundenkonto wird in den Einstellungen gespeichert


**Bild 3:** Auswahl einer Empfängerliste

und automatisch hinzugefügt. Außerdem können Sie hier einen Import von Aufträgen durchführen, wozu sie auf den entsprechenden Link klicken (siehe Bild 2).

Damit landen Sie auf einer Seite, auf der Sie die Datei mit den dafür vorbereiteten auswählen und die enthaltenen Daten einlesen können. Nach dem Import zeigt die Seite die eingelesenen Sendungen in einer Liste an und Sie können diese markierten und ausdrucken oder auch stornieren. Nach dem Ausdrucken öffnet sich ein weiteres Fenster mit einer PDF-Datei, welche die Versandetiketten enthält. Diese drucken Sie nun noch auf entsprechendem Papier aus. Dieses stellt DHL seinen Geschäftskunden kostenlos zur Verfügung (siehe Bild 3).

Nun wollen wir dies alles komplett automatisieren. Das heißt, dass wir gar keinen Internetbrowser mehr öffnen wollen, sondern aus unserer Anwendung heraus einen Webservice aufrufen wollen, dem wir die Absender- und Empfängerdaten und die Angaben zur Sendung übermitteln wollen und der uns die Etiketten dann in der Antwort zur Verfügung stellen soll.

## DHL für Entwickler

Damit landen wir dann auf der Seite **entwickler.dhl.de**. Hier können Sie nun unkompliziert ein Entwicklerkonto erstellen – die einzelnen Schritte wollen wir hier nicht darstellen. Nach dem Anlegen des Entwicklerkontos und dem


**Bild 4:** Einige der verfügbaren APIs

Anmelden im Portal finden Sie eine Auswahl der verfügbaren APIs (siehe Bild 4). Uns interessiert im Moment vor allem die API für den Geschäftskundenversand.

## Webservice ansehen

Wenn Sie sich in den Informationen zur API für Geschäftskunden umsehen, finden Sie zunächst heraus, wo Sie die WSDL-Datei des Webservices finden, nämlich unter folgendem Link:

```
https://cig.dhl.de/cig-wsdls/com/dpdhl/wsdl/geschaeftskunden-
versand-api/2.2/geschaeftskundenversand-api-2.2.wsdl
```

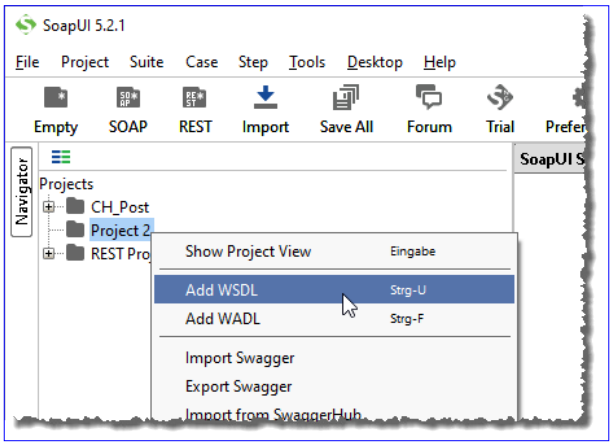



**Bild 5:** Hinzufügen einer WSDL

Was Sie damit alles anstellen können, schauen wir uns mit einem speziellen Tool namens SoapUI an, mit dem Sie den Webservice auch testen können, ohne auch nur eine Zeile Code zu programmieren. Dieses laden Sie von der Seite **https://www.soapui.org/** herunter und installieren es. Danach starten Sie SoapUI und klicken auf den Menüeintrag **File|Create Empty Project**.

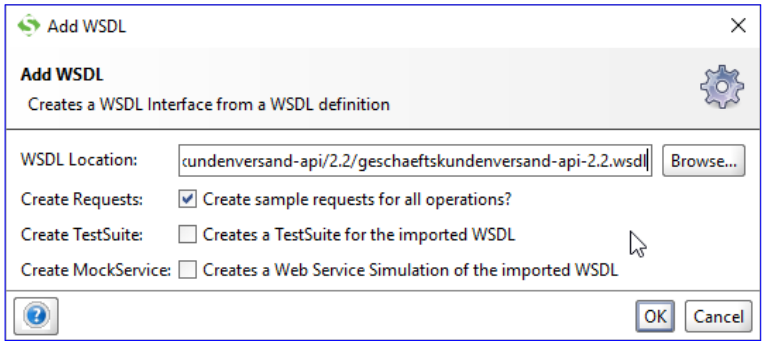



**Bild 6:** WSDL eingeben

Klicken Sie dann mit der rechten Maustaste auf den neuen Eintrag und wählen Sie den Kontextmenü-Eintrag **Add WSDL** aus (siehe Bild 5).

Es erscheint ein kleiner Dialog, in dem Sie unter **WSDL Location** den Link mit der WSDL-Datei eintragen (siehe Bild 6). Behalten Sie die Einstellung **Create sample requests for all Operations?** bei und klicken Sie auf **OK**.

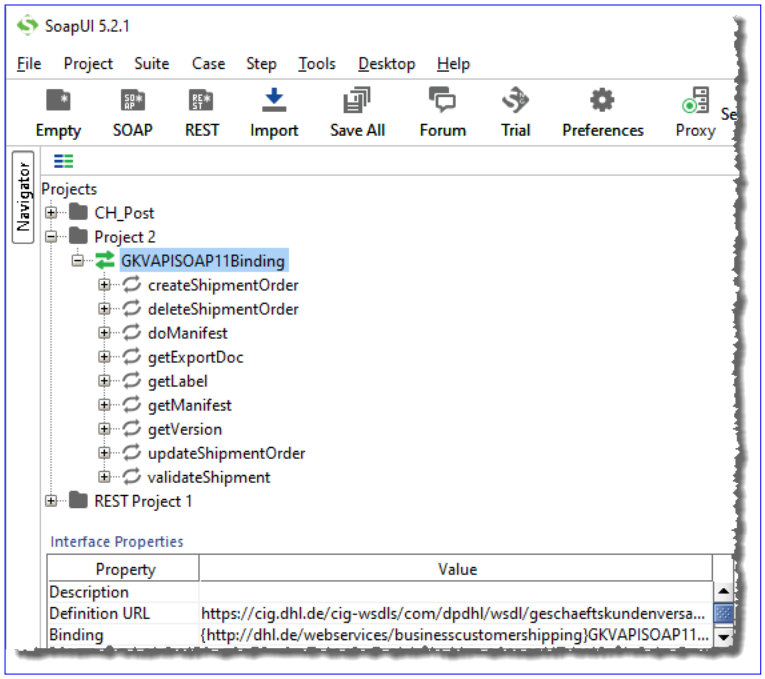



**Bild 7:** Funktionen des Webservices

Anschließend zeigt **SoapUI** im Bereich **Projects** alle Funktionen des Webservices mit der soeben analysierten WSDL-Datei an (siehe Bild 7).

Nun wollen wir einen Request zum Testen der Funktion **getLabel** des Webservice erstellen – **getLabel** hört sich so an, also ob es unsere Aufgabe, ein Label auf Basis der Versanddaten zu erstellen, erfüllen könnte. Dazu klicken Sie wieder mit der rechten Maustaste auf **getLabel**

```
<soapenv:Envelope xmlns:soapenv="...">
   <soapenv:Header>
      <cis:Authentification>
         <cis:user>?</cis:user>
         <cis:signature>?</cis:signature>
      </cis:Authentification>
   </soapenv:Header>
   <soapenv:Body>
      <bus:GetLabelRequest>
         <bus:Version>
            <majorRelease>?</majorRelease>
            <minorRelease>?</minorRelease>
            <!--Optional:-->
            <build>?</build>
         </bus:Version>
         <!--1 to 30 repetitions:-->
         <cis:shipmentNumber>?</cis:shipmentNumber>
         <!--Optional:-->
         <labelResponseType>?</labelResponseType>
      </bus:GetLabelRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

**Listing 1:** Request **getLabel** mit Platzhaltern

und wählen den Eintrag **New request** aus dem Kontextmenü aus. Geben Sie nun als Namen **getLabelRequest** ein und schauen Sie sich den Inhalt des nun geöffneten Fensters an.

Hier finden wir ein Element vor, dass die für die Authentifizierung notwendigen Daten entgegennimmt und einen Body, in den Sie eine Version eingeben und bis zu 30 **shipmentNumbers** eingeben können. Das sieht doch eher danach aus, als ob man hier bereits erstellte Label anfragen kann statt neue anzulegen. Also müssen wir wohl doch noch einen Blick auf die übrigen Funktionen des Webservice werfen.

## Request testen

Schnell zeigt sich, dass **CreateShipment-Order** die passende API-Funktion ist. Also erstellen wir einen neuen Request auf Basis diese Funktion. Das schöne am Tool SoapUI

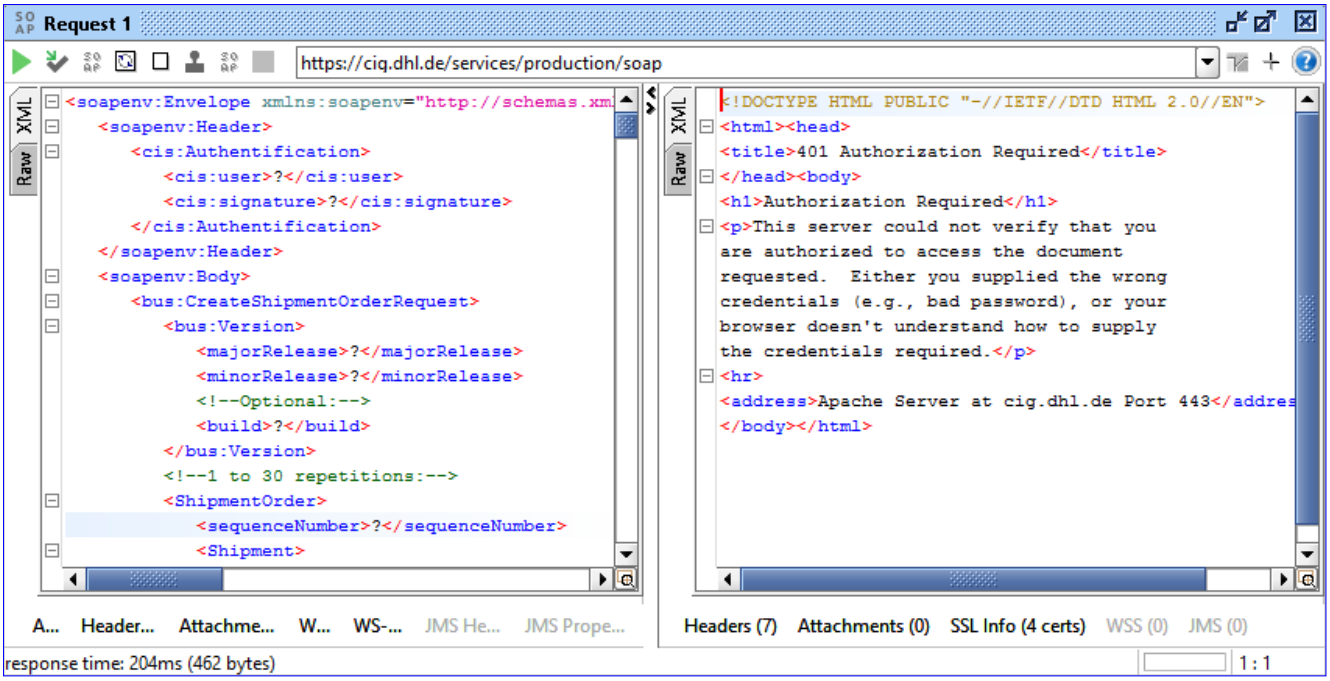



**Bild 8:** Absenden eines ersten Requests

ist: Sie können bereits jetzt einfach einmal einen Request absenden und erhalten die Antwort direkt im gleichen Fenster. In diesem Fall weist uns die Antwort darauf hin, dass wir keine Authentifizierungsdaten eingegeben haben (siehe Bild 8).

Also kümmern wir uns erst einmal um diese Daten. Diese finden Sie in der Onlinedokumentation. In diesem Fall heißt der Benutzer **2222222222_01** und das Kennwort **pass**. Außerdem müssen wir den Endpunkt für den Webservice ändern, denn nach dem Einlesen der WSDL-Datei lautet dieser auf dem Endpunkt für den Produktivbetrieb. Zum Testen gibt es natürlich einen eigenen Endpunkt, der **https://cig.dhl.de/services/sandbox/soap** lautet und den Sie in SoapUI für die Eigenschaft **Endpoint** in den **Request Properties** einstellen. Allerdings ist der Sandbox-Endpunkt dort noch nicht vorhanden, weshalb Sie diesen noch hinzufügen müssen.

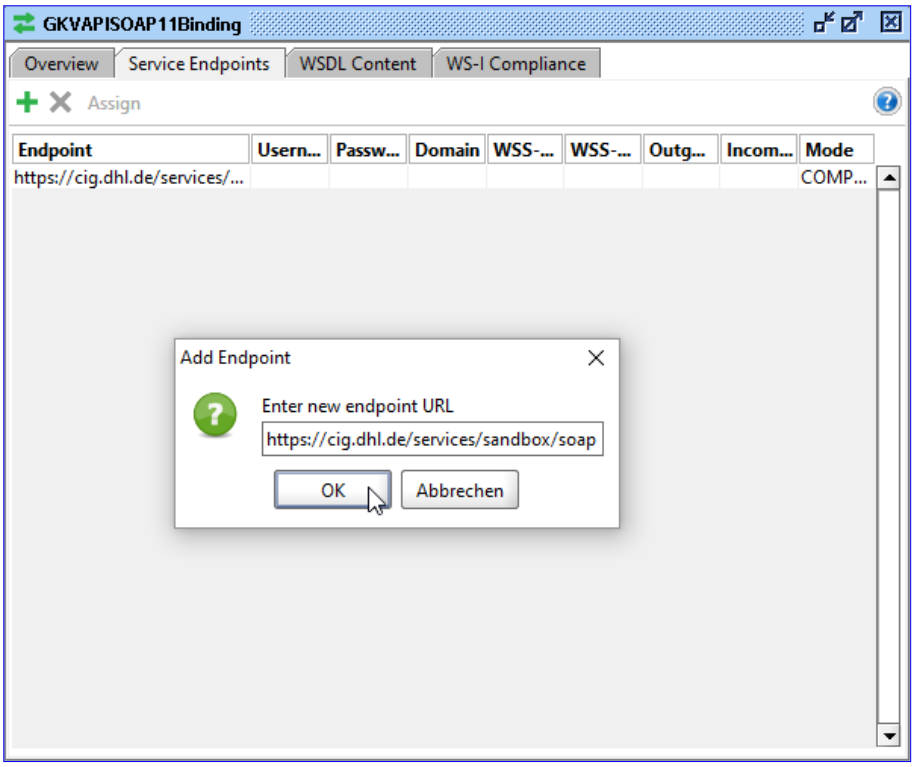



**Bild 9:** Neuen Endpoint hinzufügen

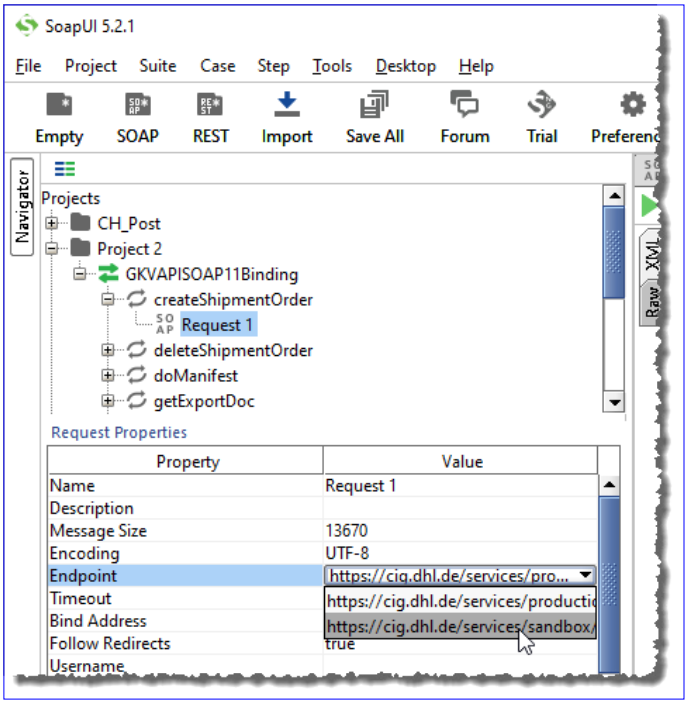



**Bild 10:** Neuen Endpoint auswählen

## Endpunkt zum Testen anlegen

Dazu klicken Sie im Bereich **Navigator** doppelt auf das Element unterhalb des Projekt-Elements, dass in unserem Beipsiel **GKVAPISOAP11Binding** genannt wurde. Es erscheint ein gleichnamiger Dialog, in dem Sie auf die Registerseite **Service Endpoints** wechseln.

Hier klicken Sie auf das Plus-Symbol, um einen neuen Endpoint anzulegen und tragen dort den Wert **https://cig.dhl.de/services/sandbox/soap** ein (siehe Bild 9).

## Neuen Endpunkt auswählen

Nach dem Schließen des Dialogs können Sie in den Eigenschaften des Requests auf Basis von **createShipmentOrder** den neuen Endpunkt auswählen (siehe Bild 10).

# Webservice mit C# am Beispiel von DHL-Etiketten

**Mit Webservices lässt sich der Abruf von Daten bestimmter Anbieter aus dem Internet stark vereinfachen. Während Sie etwa zum Herunterladen eines Versandetiketts einige Minuten investieren müssen, um die Adressen und weitere Sendungsdaten aus einer Kundendatenbank in ein Onlineformular zu übertragen, könnten Sie das auch per Mausklick erledigen. Voraussetzung sind einige Zeilen Code und die Registrierung beim entsprechenden Webservice. In diesem Fall schauen wir uns das am Beispiel von Versandetiketten des Paketdienstleisters DHL an.**

Im Artikel Webservice testen am Beispiel von DHL haben wir uns bereits angesehen, wie Sie die für einen Webservice wichtige WSDL-Datei ermitteln und damit über ein Tool wie SoapUI die XML-Vorlage für einen Request ermitteln, füllen und testen. Dort haben wir außerdem ein Entwickler-Konto angelegt, mit dem wir auf den Webservice von DHL zugreifen können. Schließlich haben wir einen funktionierenden Aufruf an die Sandbox-Version des Webservice, also eine zu Testzwecken bestimmte Umgebung, abgesetzt und mit dem zurückgelieferten Link ein PDF-Dokument mit dem Versandetikett erhalten (siehe Bild 1).


**Bild 1:** Aufruf des Dienstes **Intraship**

Nun wollen wir uns die Techniken ansehen, die dafür nötig sind, einen Request an einen Webservice abzusenden und die Antwort auszuwerten. Hier interessiert uns im vorliegenden Beispiel natürlich vor allem, wie wir den Link zum PDF-Dokument mit dem Versandetikett verarbeiten und das PDF-Dokument auf der Festplatte speichern oder ausdrucken.

## Anfrage zusammenstellen und abschicken

Wir wollen zunächst den XML-Request, den wir im Artikel Webservice testen am Beispiel von DHL ermittelt haben, an den Webservice schicken und den Request entgegennehmen. Dabei müssen wir natürlich auch noch die Authentifizierung berücksichtigen. Um die Funktion zum Ermitteln des Response-Dokuments aufzurufen, verwenden wir ein Fenster in einer WPF-Anwendung, das wir mit einer entsprechenden Schaltfläche ausstatten.

Diese gibt dann einfach den Code in einem Meldungsfenster aus (siehe Listing 1).

```
private void btnEinfacherAufruf_Click(object sender, RoutedEventArgs e) {
    MessageBox.Show(GetResponse());
}
```

**Listing 1:** Aufruf der Funktion zum Senden eines Requests an den Webservice

Um die verschiedenen nachfolgend verwendeten Klassen zu nutzen, die sich auf den Umgang mit dem Webservice, XML-Dokumenten und das Lesen und Schreiben von Dokumenten beschäftigen, benötigen wir noch drei Namespaces, die wir mit dem **using**-Schlüsselwort einbinden:

```
using System.Net;
using System.Xml;
using System.IO;
```

Die Funktion **GetResponse**, die den Ablauf steuert, finden sieht wie folgt aus:

```
public string GetResponse() {
    HttpWebRequest request = CreateRequest();
    XmlDocument requestXML = GetRequestXML();
    using (Stream stream = request.GetRequestStream()) {
        requestXML.Save(stream);
    }
    using (WebResponse response = request.GetResponse()) {
        using (StreamReader rd = new StreamReader(response.GetResponseStream())) {
            var result = rd.ReadToEnd();
            return result;
        }
    }
}
```

Hier ermitteln wir mit der Funktion **CreateRequest** ein neues Objekt des Typs **HttpWebRequest** und speichern es in der Variablen **request**. Dabei werden Informationen wie die Header für den Aufruf, die Authentifikation und weitere Informationen in einem Objekt zusammenstellt – mehr dazu weiter unten. Danach füllen wir ein Objekt des Typs **XmlDocument** namens **requestXML**, und zwar ebenfalls mithilfe einer Funktion. Diese heißt **GetRequestXML** und stellt einfach nur das XML-Dokument zusammen, das als Inhalt des Requests verwendet werden soll. Wir haben also nun ein Objekt des Typs **HttpWebRequest**, welches die Daten für die Steuerung des Requests enthält, und ein XML-Dokument mit dem zu übergebenen Request.

Nun müssen wir beide noch verheiraten. Dies geschieht im ersten **using**-Konstrukt der Funktion **GetResponse**. Das in Klammern hinter der **using**-Anweisung erzeugte Objekt wird nach dem Durchlaufen des **using**-Abschnitts wieder gelöscht. In diesem Fall handelt es sich um ein **Stream**-Objekt. Diesem weisen wir das mit der Funktion **GetRequestStream** ermittelte **Stream**-Objekt unseres **HttpWebRequest** namens **request** zu. Wir haben also mit **stream** ein **Stream**-Objekt referenziert, dass wir nun füllen können und so den übergebenen Inhalt als Request-Inhalt an das **HttpWebRequest**-Objekt übergeben. Dies erledigen wir mit der einzigen Anweisung des **using**-Konstrukts, mit der wir das in **requestXML** gespeicherte XML-Objekt in den Stream und somit in das Objekt **request** speichern.

Das zweite **using**-Konstrukt ruft dann bereits die Methode **GetResponse** von **request** auf und speichert das Ergebnis in der Variablen **response** mit dem Datentyp **WebResponse**. resonse liefert mit **GetResponseStream** wiederum einen Stream, den wir

diesmal allerdings nicht in einem **Stream**-Objekt speichern, sondern einem **StreamReader** als Konstruktorparameter übergeben. Wir erstellen also ein neues **StreamReader**-Objekt und weisen diesem den Stream zu, den wir als Antwort des Webservice erhalten haben. Diesen in **rd** gespeicherten Streamreader kapseln wir wiederum in einem **using**-Konstrukt, da wir ihn nur für die folgenden beiden Anweisungen benötigen. Die erste liest mit **ReadToEnd** den kompletten Inhalt des in **rd** gespeicherten Streams und fügt ihn der Variablen **result** hinzu. **result**, dass zu diesem Zeitpunkt das XML-Dokument mit der Antwort des Webservices enthält, wird dann schließlich als Funktionswert zurückgegeben. Die aufrufende Methode gibt den Inhalt dann schließlich wie in Bild 2 als Meldung aus.

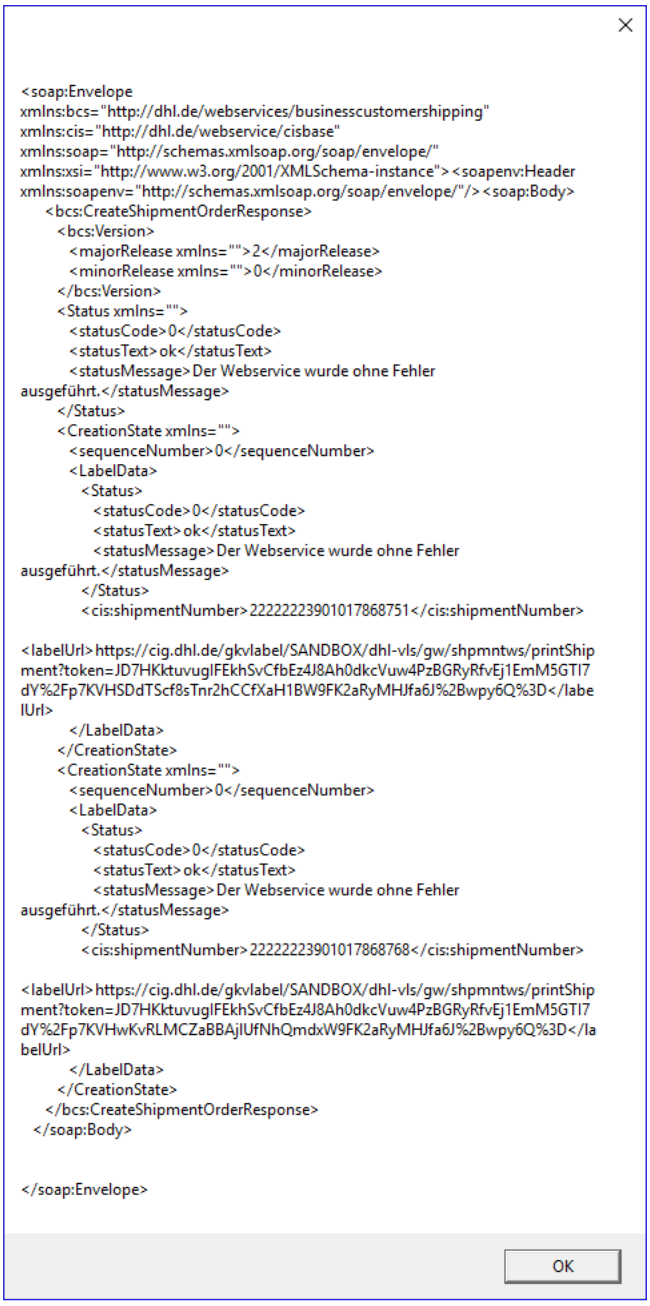



**Bild 2:** Ergebnis des Requests

## Request zusammenstellen

Die Funktion **CreateRequest** stellt den Request zusammen und gibt diesen als **HttpWebRequest**-Objekt an die aufrufende Instanz zurück. Dabei legt diese zunächst ein **Uri**-Objekt an und hinterlegt die Uri, die als Endpunkt für den Webservice dient (hier der Request für die Sandbox: **https://cig.dhl.de/services/sandbox/soap**). Danach erstellt sie das **HttpWebRequest**-Objekt auf Basis der Methode **Create** der Klasse **WebRequest**. Dieser übergibt sie das **Uri**-Objekt mit dem Endpunkt als Parameter. Das Ergebnis wird schließlich in den Typ **HttpWebRequest** umgewandelt. Danach stellt die Funktion die Zeichenkette für die Authentifizierung zusammen. Der Benutzername und das Kennwort werden zunächst in die Variablen **username** und **password** geschrieben (später fragen Sie die Daten entweder ab oder speichern direkt die kodierte Form), durch einen Doppelpunkt voneinander getrennt in einer neuen Zeichenkette aufgenommen und in einen Base64-String umgewandelt. Das Ergebnis landet in der Variablen **auth**. Danach fügt die **Add**-Methode der **Headers**-Auflistung des in **request** gespeicherten **WebHttpRequest**-Objekts einen neuen Header mit dem Namen **Authorization** und dem aus der Zeichenkette **Basic**, einem Leerzeichen und der kodierten Version aus Benutzername und Kennwort als Wert hinzu. Danach legt sie noch die Eigenschaften **ContentType** (**text/xml;charset="utf-8"**), **Accept** (**text/xml**) und **Method** (**POST**) von **request** fest. Danach gibt die Methode den Inhalt der Objektvariablen request an die aufrufende Methode zurück:

```
public HttpWebRequest CreateRequest() {
    Uri uri = new Uri("https://cig.dhl.de/services/sandbox/soap");
```

```
    HttpWebRequest request = (HttpWebRequest)WebRequest.Create(@uri);
    String username = "andreminhorst";
    String password = "*********";
    String auth = System.Convert.ToBase64String(System.Text.Encoding.UTF8.GetBytes(username + ":" + password));
    request.Headers.Add("Authorization", "Basic " + auth);
    request.ContentType = "text/xml;charset=\"utf-8\"";
    request.Accept = "text/xml";
    request.Method = "POST";
    return request;
}
```

### XML-Dokument zusammenstellen

Damit kommen wir zur letzten Funktionsmethode, die wir für den ersten Aufruf benötigen. Diese heißt **GetRequestXML** und erstellt zunächst ein neues Objekt des Typs **XmlDocument**. Die Variable für dieses Objekt heißt **requestXML**.

Es soll nun mit dem als Zeichenkette vorhandenen XML-Dokument gefüllt werden, welches wir im Artikel **Webservice testen am Beispiel von DHL** zusammengestellt und ausprobiert haben. Diese Zeichenkette laden wir mit der Methode **LoadXml** in das Objekt **requestXML** und geben dieses als Rückgabewert an die aufrufende Methode zurück:

```
public XmlDocument GetRequestXML() {
    XmlDocument requestXML = new XmlDocument();
    requestXML.LoadXml(@"<soapenv:Envelope ...>
        <soapenv:Header>
                 <cis:Authentification>
                 <cis:user>2222222222_01</cis:user>
                <cis:signature>pass</cis:signature>
            </cis:Authentification>
        </soapenv:Header>
        ---
    </soapenv:Envelope>");
    return requestXML;
}
```

Damit haben wir mit nicht allzu vielen Zeilen einen funktionierenden Aufruf eines Webservice erstellt, der in seinem Response zum Beispiel den Link zu einer PDF-Datei mit dem gewünschten Versandetikett zurückliefert.

### Nächste Schritte

Was wollen wir nun als nächstes erledigen? Folgende Aufgaben stehen noch an:

- Anpassen der Funktion **GetRequestXML**, damit wir das damit übergebene XML-Dokument dynamisch zusammenstellen können – beispielsweise auf Basis der Daten, die der Benutzer in ein Fenster eingibt oder der Daten einer Kunden-Tabelle

- Auslesen der Response-Datei, um den Link zum Versandetikett zu ermitteln
- Herunterladen des Dokuments, dass sich hinter diesem Link verbirgt
- Ausdrucken des Versandetiketts

Wir stellen das Anpassen der Funktion **GetRequestXML** nach hinten, weil wir zunächst die Funktionen zum Ermitteln, Herunterladen und Drucken des Versandetiketts programmieren wollen.

### URL des Versandetiketts ermitteln

Nun wollen wir das XML-Dokument, dass uns als Response geliefert wurde, auseinandernehmen und die enthaltenen URLs zu den Versandetiketten ermitteln. Dazu wollen wir aus dem nachfolgend ausschnittweise dargestellten Response-XML-Dokument zunächst prüfen, ob der allgemeine **statusCode** den Wert **0** hat. Falls ja, durchlaufen wir die einzelnen **CreationState**-Elemente per Code und schauen uns dort den Wert des Label-spezifischen Elements **statusCode** an. Hat auch dieses den Wert **0**, wollen wir uns den Inhalt des Elements **sequenceNumber**, des Elements **cis:shipmentNumber** und des Elements **labelUrl** merken. Wie das genau geht, erfahren Sie gleich im Anschluss:

```
<soap:Envelope ...>
   <soapenv:Header xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"/>
   <soap:Body>
      <bcs:CreateShipmentOrderResponse>
         <bcs:Version>...</bcs:Version>
         <Status>...
            <statusCode>0</statusCode>
            ...
         </Status>
         <CreationState>
            <sequenceNumber>0</sequenceNumber>
            <LabelData>
               <Status>
                  <statusCode>0</statusCode>
                  ...
               </Status>
               <cis:shipmentNumber>22222223901017866665</cis:shipmentNumber>
               <labelUrl>https://cig.dhl.de/gkvlabel/SANDBOX/dhl-vls/gw/shpmntws/printShipment?token=JD7H...</labelUrl>
            </LabelData>
         </CreationState>
         <CreationState>
               ...
               <labelUrl>https://cig.dhl.de/gkvlabel/SANDBOX/dhl-vls/gw/shpmntws/printShipment?token=JD7HKk...</labelUrl>
               ...
```

# PDFs zusammenführen mit iTextSharp

**Im Rahmen unserer Lösung zum Erstellen von DHL-Versandetiketten benötigen wir eine Lösung, mit der wir mehrere PDF-Dokumente zu einem zusammenführen können, um dieses dann in einem Rutsch auszudrucken. Mit Bordmitteln kommen wir nicht weiter, aber es gibt ja eine Reihe von Bibliotheken, die man sich etwa als NuGet-Paket ins Projekt holen kann. In diesem Fall nutzen wir die Bibliothek iTextSharp. Diese kann noch einiges mehr, aber wir wollen erst einmal nur PDF-Dokumente zusammenführen.**

## iTextSharp

**iTextSharp** ist eine C#-Portierung des Projekts **iText 5**, für das Sie unter dem folgenden Link eine Reihe von Beispielen in der Programmiersprache Java finden: **http://developers.itextpdf.com/examples-itext5**. Mit **iText 5** können Sie eine ganze Menge Aufgaben rund um die Erstellung und Bearbeitung von PDF-Dokumenten erledigen. Aktuell wollen wir jedoch nur zwei oder mehrere PDF-Dokumente zusammenführen.

## Beispielanwendung

Um die Bibliothek **iTextSharp** in ein Projekt zu integrieren, nutzen Sie den NuGet-Manager. Ausgehend von einem geöffneten Projekt klicken Sie dann mit der rechten Maustaste auf das Projekt im Projektmappen-Explorer. Aus dem Kontextmenü wählen Sie dann den Eintrag **NuGet-Pakete verwalten ...** aus. Im nun erscheinenden NuGet-Paket-Manager wechseln Sie zum Bereich Durchsuchen und geben als Suchbegriff **iTextSharp** ein. Ganz oben in der Liste erscheint nun der Eintrag **iTextSharp**, den Sie auswählen. Klicken Sie dann auf Installieren, um das Paket zum Projekt hinzuzufügen (siehe Bild 1).

Unserem WPF-Beispielprojekt fügen wir nun auf der Seite **MainWindow.xaml** eine einfache Schaltfläche hinzu, mit der wir einen ersten Test starten wollen.

Wir gehen vereinfachend davon aus, dass wir einfach zwei Dokumente namens **1.pdf** und **2.pdf** zusammenführen wollen.

Dazu fügen wir diese zum Projekt hinzu und stellen in den Eigenschaften dieser beiden Elemente den Wert **Immer kopieren** für die Eigenschaft **In Ausgabeverzeichnis kopieren** ein. So können wir das Verzeichnis für diese beiden

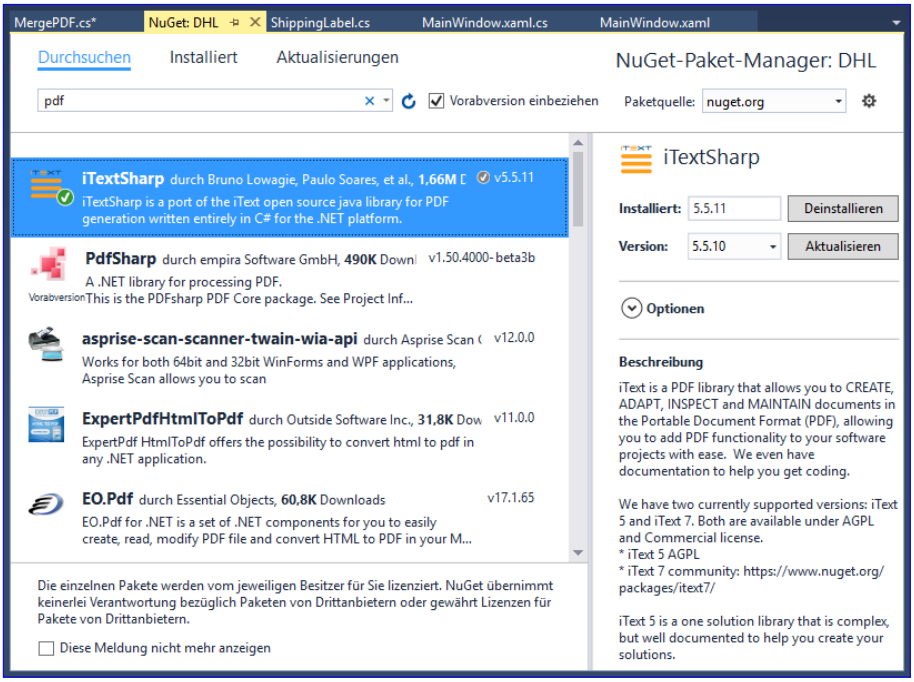



**Bild 1:** Installieren des NuGet-Pakets **iTextSharp**

Dokumente gleich direkt über die Funktion **AppDomain.CurrentDomain.BaseDirectory** ermitteln (dies liefert das Verzeichnis, in dem sich auch die **.exe**-Datei der Lösung befindet).

### Dokumente zusammenführen

Wir wollen nun eine Methode programmieren, welche diese beiden Dokumente zusammenführt und als neues PDF-Dokument abspeichert. Das geht mit der **iTextSharp**-Bibliothek wirklich sehr einfach. Für die Schaltfläche hinterlegen wir die folgende Methode, die zunächst drei Pfade in den Variablen **document1**, **document2** und **target** hinterlegt – jeweils mit dem aktuelle Anwendungsverzeichnis und den Namen **1.pdf**, **2.pdf** und **12.pdf**:

```
private void btnTestZweiDokumente_Click(object sender, RoutedEventArgs e) {
    string document1 = AppDomain.CurrentDomain.BaseDirectory + "\\1.pdf";
    string document2 = AppDomain.CurrentDomain.BaseDirectory + "\\2.pdf";
    string target = AppDomain.CurrentDomain.BaseDirectory + "\\12.pdf";
```

Dann erstellt die Methode ein neues Objekt des Typs **Document** und eines vom Typ **FileStream**. Letzteres wird auf Basis des Pfades zur Zieldatei als neue Datei angelegt:

```
    Document document = new Document();
    FileStream stream = new FileStream(target, FileMode.Create);
```

Das Objekt **pdf** mit dem Typ **PdfCopy** führt das **Document**- und das **FileStream**-Objekt zusammen. Ein Objekt des Typs **PdfReader**, das später die Ausgangsdokumente aufnimmt, wird mit dem Wert **null** initialisiert. Dann wird das **Document**-Objekt mit der **open**-Methode geöffnet:

```
    PdfCopy pdf = new PdfCopy(document, stream);
    PdfReader reader = null;
    document.Open();
```

Nun folgen die beiden Einlesevorgänge der zusammenzuführenden Dokumente. Dabei wird **reader** jeweils mit einem neuen **PdfReader**-Objekt mit dem Dokumentpfad als Konstruktor-Parameter initialisiert und für die Anwendung innerhalb des **using**-Konstrukts vorbereitet. Darin wird das Dokument aus der Objektvariablen **reader** zum **pdfCopy**-Objekt hinzugefügt:

```
    using (reader = new PdfReader(document1)) {
        pdf.AddDocument(reader);
    }
```

Das erledigen wir auch noch für das zweite Dokument, also **2.pdf**:

```
    using (reader = new PdfReader(document2)) {
        pdf.AddDocument(reader);
    }
```

Schließlich schließt die Methode das Objekt **document** mit der **Close**-Methode:

```
    document.Close();
}
```

Das Ergebnis überzeugt: Beide Dokumente erscheinen im neuen Dokument **12.pdf**.

## Anwendung bauen

Wenn wir nun schon so eine einfache Möglichkeit haben, PDF-Dokumente zusammenzustellen, wollen wir auch gleich eine praktische Anwendung daraus bauen. Diese soll es ermöglichen, über einen Datei auswählen-Dialog verschiedene PDF-Dokumente auszuwählen und diese so einer Liste hinzuzufügen. Ein Mausklick soll die Dokumente dann unter einem ebenfalls per Dateidialog auszuwählenden Dateinamen zusammenführen und speichern.

Um dies zu realisieren, benötigen wir ein Listenfeld, dass die Dateien anzeigt, sowie eine Schaltfläche zum Hinzufügen der gewünschten Dateien. Außerdem brauchen wir noch eine Schaltfläche, die den Datei speichern-Dialog anzeigt und nach der Eingabe eines entsprechenden Dateinamens die Dateien in der Reihenfolge wie im Listenfeld zusammenzuführen. Interessant wäre es natürlich noch, die Dateien im Listenfeld nach oben oder unten verschieben zu können. Dies wollen wir jedoch hintenanstellen und uns zunächst um die grundlegenden Funktionen kümmern.

Für diese Lösung verlassen wir das Beispielprojekt und erstellen ein neues WPF-Projekt namens **PDFMerger**. Diesem fügen Sie nun zunächst wieder das NuGet-Paket **iTextSharp** hinzu. Dann legen wir die benötigten Steuerelemente im Fenster **MainWindow.xaml** an. Der dazu verwendete Code sieht wie folgt aus. Der **<Windows.Resources>**-Teil legt zwei allgemeine Eigenschaften für die enthaltenen **Button**-Elemente fest:

```
<Window x:Class="PDFMerger.MainWindow" ... Title="MainWindow" Height="350" Width="525">
    <Window.Resources>
        <Style TargetType="{x:Type Button}">
            <Setter Property="Margin" Value="5"></Setter>
            <Setter Property="Height" Value="25"></Setter>
        </Style>
    </Window.Resources>
```

Das Grid besteht aus zwei Zeilen, von denen die obere das **ListView**-Element und das untere ein horizontal orientiertes StackPanel enthält, welches wiederum die fünf Schaltflächen aufnimmt:

```
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition Height="Auto"></RowDefinition>
        </Grid.RowDefinitions>
        <ListView x:Name="lstDateien" Margin="5"></ListView>
```

```
        <StackPanel Orientation="Horizontal" Grid.Row="1">
            <Button x:Name="btnDateiAuswaehlen" Click="btnDateiAuswaehlen_Click">Dateien auswählen</Button>
            <Button x:Name="btnDateienZusammenfuehren" Click="btnDateienZusammenfuehren_Click">Dateien zusammenführen</Button>
            <Button x:Name="btnListeLeeren" Click="btnListeLeeren_Click">Liste leeren</Button>
            <Button x:Name="btnAuf" Click="btnAuf_Click">Auf</Button>
            <Button x:Name="btnAb" Click="btnAb_Click">Ab</Button>
        </StackPanel>
    </Grid>
</Window>
```

Im Entwurf sieht das Fenster nun wie in Bild 2 aus.

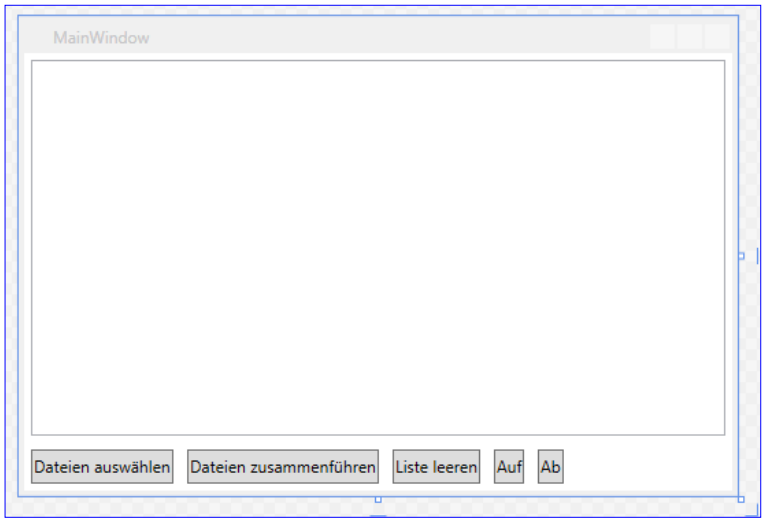



**Bild 2:** Entwurf des Fensters zum Mergen von PDF-Dokumenten

## Fenster programmieren

Die Code behind-Klasse **MainWindow.xaml.cs** verwendet die folgenden Namespaces:

```
using System;
using System.Windows;
using Microsoft.Win32;
using iTextSharp.text;
using iTextSharp.text.pdf;
using System.IO;
using System.Collections.ObjectModel;
```

Die per Dateidialog ermittelten Namen der zusammenzuführenden PDF-Dateien sollen in einer **ObservableCollection** namens **Dateien** aufgenommen werden, die wir als private Variable deklarieren und initialisieren:

```
private ObservableCollection<string> Dateien = new ObservableCollection<string>();
```

Damit die in der **ObservableCollection** enthaltenen Dateinamen im **ListView**-Steuerelement angezeigt werden, weisen wir der Eigenschaft ItemsSource in der Konstruktor-Methode der Klasse einen Verweis auf die Collection zu:

```
public MainWindow() {
    InitializeComponent();
    lstDateien.ItemsSource = this.Dateien;
}
```

Ein Klick auf die Schaltfläche **btnDateiAuswaehlen** ruft die folgende Ereignismethode auf. Die Methode erstellt ein neues Objekt des Typs **OpenFileDialog** und stellt einige Eigenschaften für die entsprechende Objektvariable ein. Dieses erlaubt so die Auswahl mehrerer Dateien gleichzeitig (**Multiselect = true**), zeigt nur **.pdf**-Dokumente an (**Filter = "PDF-Dokumente (*.pdf)|*.pdf"**) und startet mit dem aktuelle Anwendungsverzeichnis:

# EDM: Bestellungen und Bestellpositionen

**Auch in dieser Ausgabe geht es mit unserer Beispiellösung Bestellverwaltung weiter. Nachdem wir bereits Kunden, Produkte und Kategorien verwalten können, nehmen wir uns nun die Bestellungen und Bestellpositionen vor. Dabei brauchen wir ein paar neue Seiten, um die Details einer Bestellung mit den Bestellpositionen und die Details einer neuen oder zu bearbeitenden Bestellposition anzuzeigen. Damit werden wir erstmal die Bearbeitung von Daten einer m:n-Beziehung ermöglichen.**

## Neue Funktionen

Die neue Version der Bestellverwaltung enthält wieder eine neue Gruppe im Ribbon, mit der zwei neue Seiten geöffnet werden können. Die erste davon ist eine Übersicht der Bestellungen in absteigender Reihenfolge nach dem Bestelldatum (siehe Bild 1).

Hier kann der Benutzer die aktuell markierte Bestellung per Mausklick löschen oder auch eine neue Bestellung anlegen. Per Doppelklick lässt sich die angeklickte Bestellung öffnen (siehe Bild 2).

Ein Klick auf den Ribbon-Eintrag **Neue Bestellung** soll den Dialog mit den Bestelldetails für eine neue Bestellung anzeigen, wobei direkt das Kombinationsfeld zur Auswahl des Kunden für diese Bestellung ausgeklappt wird (siehe Bild 3).

Auch wenn der Benutzer mit einem Klick auf die Schaltfläche **Neue Bestellposition** klickt, soll sich direkt das Kombinationsfeld zur Auswahl des betroffenen Produkts öffnen und die schnelle Auswahl erleichtern.

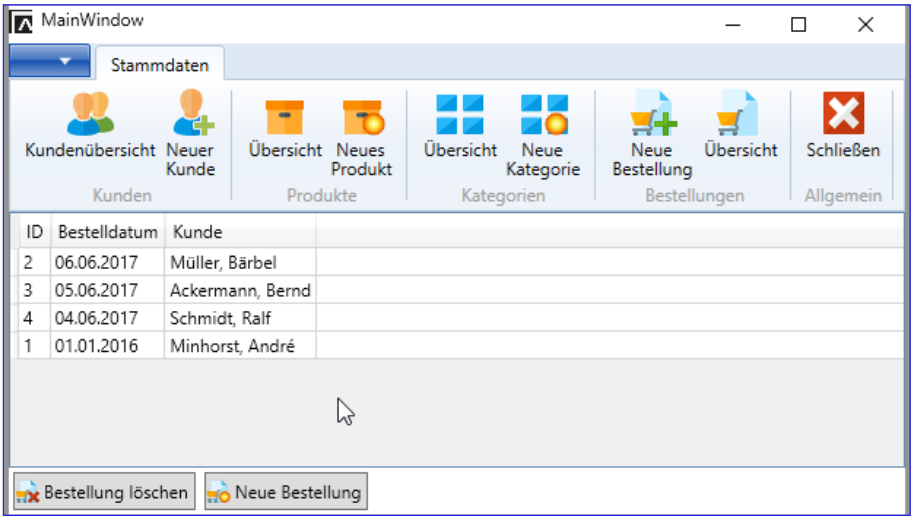



**Bild 1:** Übersicht der Bestellungen

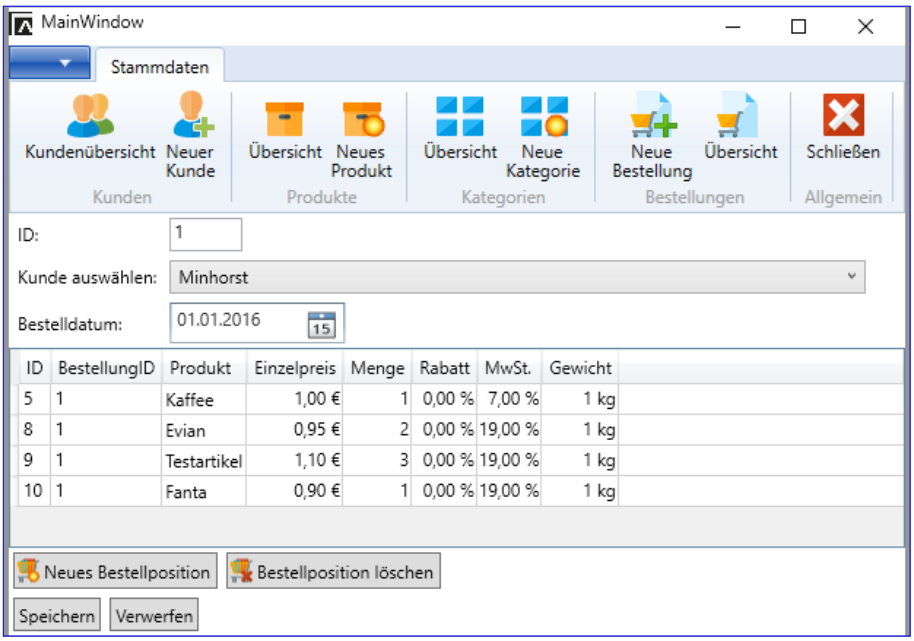



**Bild 2:** Bestellung in der Detailansicht

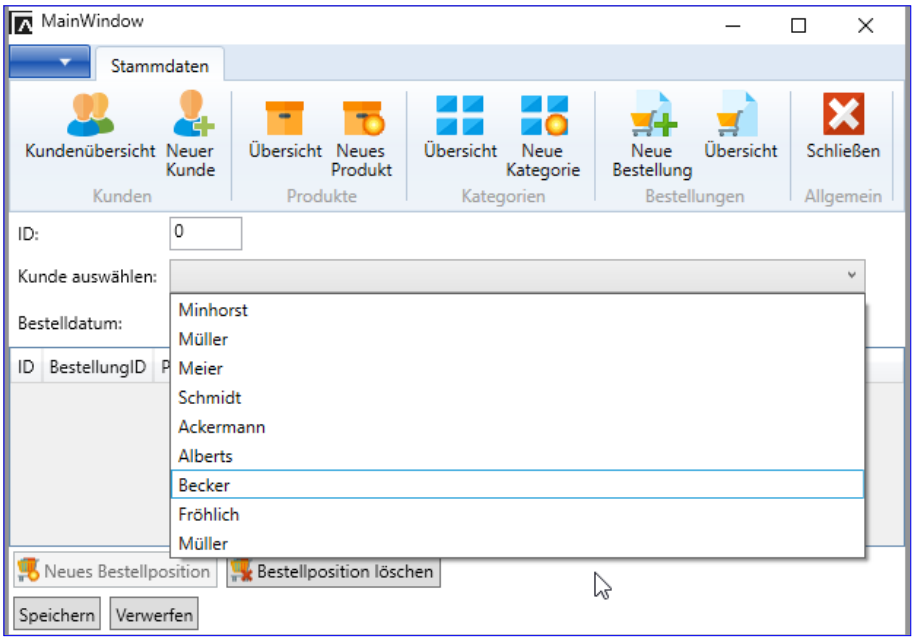



**Bild 3:** Anlegen einer neuen Bestellung

Wenn der Benutzer dann das gewünschte Produkte ausgewählt hat, sollen direkt die Werte der Felder **Einzelpreis**, **Mehrwertsteuersatz** und **Gewicht** aus der Tabelle **Produkte** in den neuen Datensatz der Tabelle **tblBestellpositionen** übernommen werden.

Die Detailseite für eine Bestellposition, wie sie nach einem Doppelklick auf einen der Einträge der Seite mit den Bestelldetails aussieht, finden Sie in Bild 4.

## Datenbankerweiterungen

Für diese neue Version der Bestellverwaltung haben wir ein paar kleine Änderungen am Datenmodell der SQLite-Datenbank **Bestellungen.db** vorgenommen. Dabei sind wie auf zwei Probleme gestoßen, die wir wie folgt gelöst haben:

- Mit dem **DBBrowser for SQLite** konnten wir in manchen Fällen keine Zahlenfelder zu einer bestehenden Tabelle hinzufügen. Dies führt im schlimmsten Fall zum Leeren der kompletten Datenbank. Wir sind dann zu einem anderen Tool namens **SQLiteStudio** gewechselt (**sqlitestudio.pl**), welches die Änderungen klaglos vorgenommen hat.

- Als wir die Änderungen an der Datenbank in das Entity Data Model übernehmen wollten, gab es ebenfalls Probleme. Die Änderungen übernehmen Sie, indem Sie das Entity Data Model (Datei **BestellverwaltungEntities.edmx**) öffnen und dann den Kontextmenüeintrag **Modell aus der Datenbank aktualisieren ...** auswählen). Die nachfolgende Fehlermeldung, deren Text mit **[A]System.Data.SQLite.SQLiteConnection cannot be cast to [B]System.Data.SQLite.SQLiteConnection.** begann, wurde behoben, nachdem wir die aktuelle Version des NuGet-Pakets **System.Data.SQLite** sowie die aktuelle Runtime-Komponente

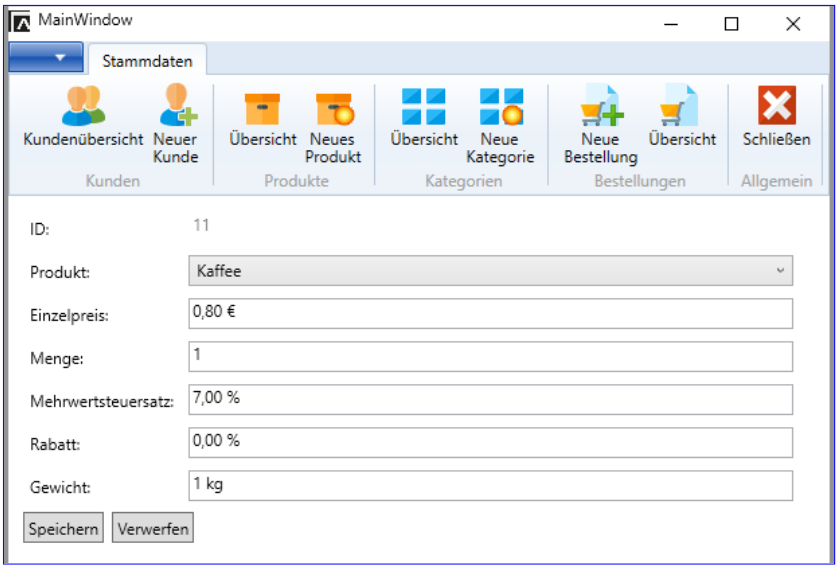



**Bild 4:** Bearbeiten einer Bestellposition

von der Internetseite **https://system.data.sqlite.org/index.html/doc/trunk/www/downloads.wiki** installiert und Visual Studio dann neu gestartet haben.

### Hinzufügen der neuen Ribbon-Elemente

Dieser Teil bringt keine neuen Techniken. Die beiden neu hinzugefügten Ribbon-Schaltflächen heißen **btnNeueBestellung** und **btnBestelluebersicht** und laden die Seiten **BestellungDetails.xaml** beziehungsweise **BestellungUebersicht.xaml** in das **Frame**-Element namens **Workzone**.

Die Schaltfläche **btnBestelluebersicht** löst die folgende Ereignismethode aus:

```
private void btnBestelluebersicht_Click(object sender, RoutedEventArgs e) {
    ShowBestellungenUebersicht();
}
```

Die hier aufgerufene Methode **ShowBestellungenUebersicht** verwendet den optinalen Parameter **bestellungID**, der beim Aufruf nach dem Ändern oder Neuanlegen einer Bestellung übergeben werden kann, damit der geänderte oder neu angelegte Eintrag im DataGrid markiert werden kann.

Wurde die Seite **BestellungUebersicht** noch nicht erstellt oder gab es eine Änderung bei einer der Bestellungen, wird die Seite neu erstellt und die Variable **BestellungChanged** auf **false** eingestellt. Anschließend abonniert die Klasse **MainWindow.xaml.cs** die beiden Ereignisse **ItemDoubleClicked** und **NewItem** der Klasse **BestellungUebersicht.xaml.cs**. Schließlich wird die Seite, egal ob neu erzeugt oder bereits vorhanden, im **Frame**-Steuerelement **Workzone** angezeigt:

```
private void ShowBestellungenUebersicht(long bestellungID = 0) {
    if (bestellungUebersicht == null | BestellungChanged == true) {
        bestellungUebersicht = new BestellungUebersicht(bestellungID);
        BestellungChanged = false;
    }
    bestellungUebersicht.ItemDoubleClicked += new BestellungUebersicht.EventHandlerItemDoubleClicked(OnBestellungDoubleCl
icked);
    bestellungUebersicht.NewItem += new BestellungUebersicht.EventHandlerNewItem(OnNewBestellung);
    WorkZone.Content = bestellungUebersicht;
}
```

### Übersicht der Bestellungen

Schauen wir uns erst die Seite **BestellungUebersicht.xaml** an. Diese enthält unter anderem ein DataGrid mit drei Spalten, die wie folgt definiert werden. Interessant ist hier wieder die in eine Resource des **DataGrid**-Elements ausgelagerte Ereignisdefinition für das Ereignis **MouseDoubleClick**, welches die Methode **Row_DoubleClick** aufrufen soll.

Außerdem haben wir hier, um die beiden Felder **Nachname** und **Vorname** der Auflistung **Kunden** des **Bestellung**-Objekts in einem **MultiBinding**-Element zusammengefasst:

```
<DataGrid x:Name="dgBestellungen" ItemsSource="{Binding Bestellungen}" AutoGenerateColumns="false" CanUserAddRows="False"
Grid.Row="0">
    <DataGrid.Columns>
        <DataGridTextColumn Binding="{Binding Path=ID}" Header="ID"  />
        <DataGridTextColumn Binding="{Binding Path=Bestelldatum, StringFormat=\{0:dd.MM.yyyy\}}" Header="Bestelldatum" />
        <DataGridTextColumn Header="Kunde Multibinding">
            <DataGridTextColumn.Binding>
                <MultiBinding StringFormat="{}{0}, {1}">
                    <Binding Path="Kunden.Nachname" />
                    <Binding Path="Kunden.Vorname" />
                </MultiBinding>
            </DataGridTextColumn.Binding>
        </DataGridTextColumn>
        <DataGridTextColumn Binding="{Binding Path=Kunde}" Header="Kunde" />
    </DataGrid.Columns>
    <DataGrid.Resources>
        <Style TargetType="DataGridRow">
            <EventSetter Event="MouseDoubleClick" Handler="Row_DoubleClick"/>
        </Style>
    </DataGrid.Resources>
</DataGrid>
```

Das DataGrid wird mit den Bestellungen aus einer **ObservableCollection** namens **bestellungen** mit Elementen des Typs **Bestellung** gefüttert, und zwar über die öffentliche Eigenschaft **Bestellungen**:

```
private ObservableCollection<Bestellung> bestellungen;
public ObservableCollection<Bestellung> Bestellungen {
    get { return bestellungen; }
    set { bestellungen = value; }
}
```

In der Konstruktor-Methode **BestellungUebersicht** erstellen wir das **dbContext**-Element und füllen die Collection **bestellungen**:

```
public BestellungUebersicht(long bestellungID = 0) {
    InitializeComponent();
    dbContext = new BestellverwaltungEntities();
    bestellungen = new ObservableCollection<Bestellung>(dbContext.Bestellungen.OrderByDescending(c => c.Bestelldatum));
    DataContext = this;
    if (bestellungID != 0) {
        Bestellung currentBestellung = dbContext.Bestellungen.Find(bestellungID);
```

```
            dgBestellungen.Focus();
            dgBestellungen.SelectedItem = currentBestellung;
            dgBestellungen.ScrollIntoView(currentBestellung);
        }
}
```

### Neue Bestellung anlegen

Ein Klick auf die Schaltfläche **btnNeu** der Seite **BestellungUebersicht.xaml** löst die folgende Methode aus, welche wiederum das Ereignis **NewItem** aufruft:

```
private void btnNeu_Click(object sender, RoutedEventArgs e) {
    NewItem(this, null);
}
```


**Bild 5:** Entwurf der Bestelldetails samt DataGrid für die Bestellpositionen

**NewItem** ist wie folgt in **BestellungUebersicht.xaml.cs** definiert:

```
public delegate void EventHandlerNewItem(object sender, System.EventArgs e);
public event EventHandlerNewItem NewItem;
```

Das Ereignis **NewItem** wird im übergeordneten Fenster-Element **MainWindow.xaml** abonniert, sobald der Benutzer auf die Ribbon-Schaltfläche **btnBestelluebersicht** klickt (mehr siehe in den Methoden **btnBestelluebersicht_Click** und **ShowBestellungenUebersicht** in der Datei **MainWindow.xaml.cs**). Das Ereignis wird in **MainWindow.xaml.cs** durch die folgende Ereignisprozedur implementiert:

```
private void OnNewBestellung(object sender, EventArgs e) {
    BestelldetailsAnzeigen();
}
```

Damit steigen wir in die Methode **BestelldetailsAnzeigen** ein, die wir ebenfalls in der Klasse **MainWindow.xaml.cs** finden. Diese erstellt ein neues Objekt auf Basis der Seite **BestelldetailsAnzeigen** und übergibt den Parameter **bestellungID**, der hier **0** ist. Dann abonniert sie vier Ereignisse, nämlich **ItemChanged**, **Cancelled**, **NewItem** und **ItemDoubleClicked**. Die ersten beiden sind für die beiden Schaltflächen **Speichern** und **Verwerfen**, die hinteren beiden für die Schaltflächen **Neue Bestellposition** und **Bestellposition löschen**, die zum DataGrid zur Anzeige der Bestellpositionen gehören (siehe Bild 5). Nach dem Abonnieren der Ereignisse weist die Methode das neu erstellte **Page**-Objekt auf Basis der Klasse **BestellungDetails** noch der Eigenschaft **Content** des **Frame**-Objekts **Workzone** zu, welches die Seite dann anzeigt:

```
private void BestelldetailsAnzeigen(long bestellungID = 0) {
    bestellungDetails = new BestellungDetails(bestellungID);
    bestellungDetails.ItemChanged += new BestellungDetails.EventHandlerItemChanged(OnBestellungChanged);
    bestellungDetails.Cancelled += new BestellungDetails.EventHandlerCancelled(OnBestellungCancelled);
```

```
    bestellungDetails.NewItem += new BestellungDetails.EventHandlerNewItem(OnNewBestellposition);
    bestellungDetails.ItemDoubleClicked += new BestellungDetails.EventHandlerItemDoubleClicked(OnBestellpositionDoubleClicked);
    WorkZone.Content = bestellungDetails;
}
```

### Felder in den Bestelldetails definieren

Die Seite **BestellungDetails.xaml** besteht aus drei Bereichen: den oberen Steuerelementen zur Eingabe der Bestelldetails, dem DataGrid zur Anzeige der Bestellpositionen und den Schaltflächen zum Hinzufügen und Löschen von Bestellpositionen sowie den beiden Schaltflächen zum Speichern oder Verwerfen des aktuell angezeigten Objekts aus der Tabelle **Bestellungen**. Schauen wir uns zunächst die drei Steuerelemente an. Die TextBox mit dem Bezeichnungsfeld **ID** ist an das Feld **Bestellung.ID** gebunden. Das heißt, dass wir in der Code behind-Datei ein Objekt namens **Bestellung** bereitstellen müssen – dazu später mehr. Interessant ist die ComboBox. Hier haben wir nicht wie üblich nur ein einziges Feld als anzuzeigenden Inhalt angegeben, sondern einen aus den beiden Feldern **Nachname** und **Vorname** bestehenden Ausdruck, also etwa **Minhorst, André**. Dies erledigen wir, indem wir das Attribut **DisplayMemberPath** weglassen und stattdessen ein Unterelement namens **ComboBox.DataTemplate** einfügen. Dieses soll ein **TextBlock**-Element enthalten, dessen Eigenschaftselement **Text** wir mit einem Multibinding der beiden Felder **Nachname** und **Vorname** in der mit **StringFormat** angegebenen Formatierung ausgeben:

```
<Label Grid.Row="0">ID:</Label>
<TextBox Grid.Column="1" Width="50" HorizontalAlignment="Left"  Text="{Binding bestellung.ID, Mode=TwoWay,
ValidatesOnExceptions=true}" ></TextBox>
<Label Grid.Row="1">Kunde auswählen:</Label>
<ComboBox x:Name="cboKunden" Grid.Column="1" HorizontalAlignment="Stretch" TabIndex="0" Height="23" Grid.Row="1"
            ItemsSource="{Binding Kunden}"
            SelectedItem="{Binding bestellung.Kunden, ValidatesOnDataErrors=True}"
            SelectedValuePath="ID" SelectionChanged="cboKunden_SelectionChanged">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <TextBlock>
                <TextBlock.Text>
                    <MultiBinding StringFormat="{}{0}, {1}">
                        <Binding Path="Nachname" />
                        <Binding Path="Vorname" />
                    </MultiBinding>
                </TextBlock.Text>
            </TextBlock>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
<Label Grid.Row="2">Bestelldatum:</Label>
<DatePicker Grid.Row="2" Grid.Column="1" Width="120" HorizontalAlignment="Left"
    Text="{Binding bestellung.Bestelldatum, Mode=TwoWay, ValidatesOnExceptions=true}" ></DatePicker>
```

Das DataGrid liefert die Felder der Auflistung **Bestellpositionen**, die neben dem Objekt **Bestellung** in der Code behind-Klasse bereitgestellt wird. Hier die gekürzte Fassung aus **BestellungDetails.xaml**. Interessant ist der Inhalt der Spalte mit der Überschrift **Produkt**, deren Wert wir aus der in dem Objekt **Bestellung** enthaltenen Auflistung **Produkte** erhalten. Außerdem definieren wir wieder ein Event, dass beim Doppelklick auf eine Zeile ausgelöst wird:

```
<DataGrid x:Name="dgBestellpositionen" Grid.Column="0" Grid.Row="3" Grid.ColumnSpan="2"
        ItemsSource="{Binding Bestellpositionen}" AutoGenerateColumns="false" CanUserAddRows="False">
    <DataGrid.Columns>
        <DataGridTextColumn Binding="{Binding Path=ID}" Header="ID"  />
        <DataGridTextColumn Binding="{Binding Path=BestellungID}" Header="BestellungID" />
        <DataGridTextColumn Binding="{Binding Path=Produkte.Bezeichnung}" Header="Produkt" />
        ...
    </DataGrid.Columns>
    <DataGrid.Resources>
        <Style TargetType="DataGridRow">
            <EventSetter Event="MouseDoubleClick" Handler="Row_DoubleClick"/>
        </Style>
    </DataGrid.Resources>
</DataGrid>
```

In der Code behind-Klasse **BestellungDetails.xaml.cs** definieren wir dann einige Objekte, die wir im XAML-Code referenzieren. Das Objekt **bestellung** enthält die Daten der angezeigten Bestellung, **Kunden** liefert die Daten für das Kombinationsfeld **cboKunden** und **Bestellpositionen** die Daten für das DataGrid:

```
public Bestellung bestellung { get; set; }
private List<Kunde> kunden;
public List<Kunde> Kunden {
    get { return kunden; }
    set { kunden = value; }
}
private ObservableCollection<Bestellposition> bestellpositionen;
public ObservableCollection<Bestellposition> Bestellpositionen {
    get { return bestellpositionen; }
    set { bestellpositionen = value; }
}
```

### Bestelldetails füllen

Die Konstruktor-Methode **BestellungDetails** nimmt mit dem Parameter **bestellungID** den Primärschlüsselwert einer Bestellung entgegen. Dieser lautet **0**, wenn der Benutzer in **BestellungenUebersicht.xaml** auf die Schaltfläche **btnNeu** geklickt hat. Hat der Benutzer hingegen doppelt auf einen Eintrag im DataGrid von **BestellungenUebersicht.xaml** geklickt, dann wird die **bestellungID** zur angeklickten Bestellung übermittelt, damit diese angezeigt werden kann. **BestellungDetails** prüft den Wert