ACCESS IM UNTERNEHMEN

JAHRESARCHIV 2022

Inhalt

Access im Unternehmen Jahresarchiv 2022

Endlich ist es soweit: Es gibt ein Jahresarchiv von Access im Unternehmen mit allen Artikel des Jahres 2022! Einleitend finden Sie hier ein paar Hinweise, wie das Jahresarchiv aufgebaut ist und was es enthält. Sie finden hier die vollständigen Ausgaben eines ganzen Jahres – mit 456 Seiten Access-Know-how in über 50 Beiträgen mit fast ebenso vielen Beispieldatenbanken zum Ausprobieren, Erweitern und Übernehmen in Ihre eigenen Lösungen.

Bevor Sie in die Lektüre der Beiträge des Jahres 2022 einsteigen, finden Sie hier noch den wichtigsten Hinweis – nämlich den, wo Sie die Beispieldatenbanken zu den einzelnen Artikeln finden. Dazu werfen Sie einen Blick in den unteren Bereich einer beliebigen Seite eines Artikels, der wie in der Abbildung unten aussieht. Hier finden Sie drei Angaben:

- Die Seite in der aktuellen Ausgabe (also die Originalseitenzahl). Diese können Sie nutzen, wenn Sie vom Inhaltsverzeichnis einer jeden Ausgabe aus starten wollen

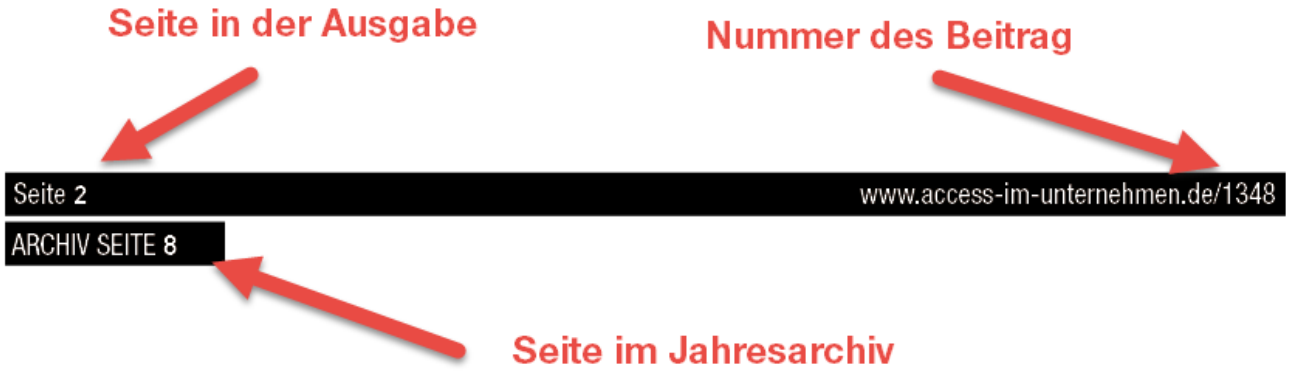
- Die Seite im Archiv, die auch im Inhaltsverzeichnis des Jahresarchivs verwendet wird (siehe die vorherigen Seiten)

- Die Nummer des Beitrags hinter der URL zur Webseite **www.access-im-unternehmen.de**.

Diese Nummer ist wichtig für das Auffinden der Beispieldatenbanken. Diese finden Sie im Download im Unterverzeichnis Beispieldateien. In diesem Verzeichnis liegen weitere Unterverzeichnis, die genau diese Nummer als Bezeichnung verwenden. Hier finden Sie also die passenden Beispieldateien.

Damit wünsche ich nun, wie immer, viel Spaß beim Lesen!

Ihr André Minhorst

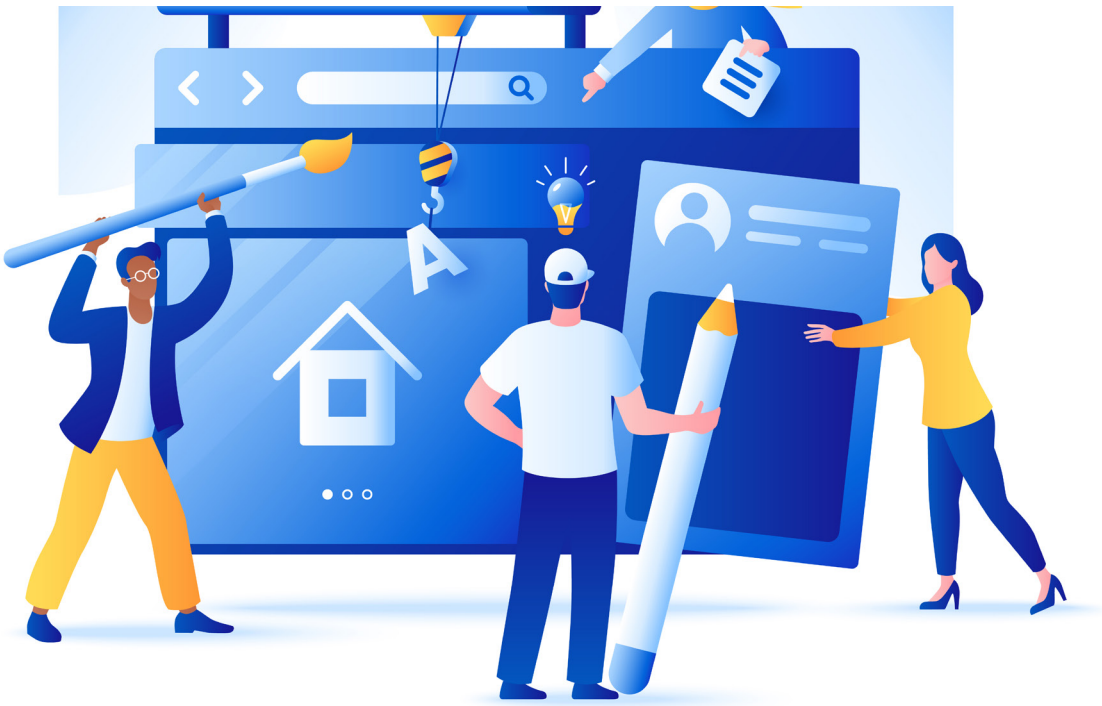


HAUFE.
Ausgabe 01/2022

ACCESS
IM UNTERNEHMEN

VBA-EDITOR PROGRAMMIEREN

Lernen Sie, den VBA-Editor per Code zu steuern und somit Aufgaben zu automatisieren (ab Seite 17).



In diesem Heft:

DATEV-EXPORT

Exportieren Sie Ihre Buchungsdaten in das DATEV-Format und leiten Sie diese direkt an Ihren Steuerberater weiter.

SEITE 60

ETIKETTEN DRUCKEN

Automatisieren Sie die Ausgabe von Adressetiketten direkt auf einen Labeldrucker.

SEITE 2

FEIERTAGE PER VBA

Ermitteln Sie die Feiertage für ein Jahr und ein Bundesland direkt über eine praktische VBA-Funktion.

SEITE 7

www.access-im-unternehmen.de
Mat-Nr. 01583-5129

ARCHIV SEITE 5

VBA-Editor programmieren

Wer nicht nur Access-Anwendungen programmieren möchte, sondern vielleicht auch einmal Tools entwickeln will, mit denen sich bestimmte Schritte bei der Programmierung vereinfachen lassen, kommt nicht um Kenntnisse zur Programmierung der Entwicklungsumgebung selbst herum. Um dieses Thema kümmern wir uns in einem Schwerpunkt in dieser Ausgabe. Dabei schauen wir uns die verschiedenen Bereiche und Techniken an, mit denen Sie die Elemente eines VBA-Projekts per Code selbst erstellen und manipulieren können.

Den Start macht der Beitrag **VBA-Projekt per VBA referenzieren** (ab Seite 17). Hier erfahren Sie, wie Sie das VBA-Projekt der aktuellen Access-Datenbank überhaupt referenzieren, denn das ist die Basis für alle weiteren Schritte beim Programmieren der Entwicklungsumgebung.

Der Beitrag **Zugriff auf den VBA-Editor mit der VBE-Klasse** beschreibt ab Seite 21 die Grundlagen der Klasse, die alle weiteren Elemente, Methoden und Eigenschaften für den Zugriff auf die Elemente des VBA-Editors bereitstellt. Hier lernen Sie auch die grundlegenden Elemente des VBA-Editors aus Sicht dieser Klasse kennen.

Im Beitrag **Zugriff auf VBA-Projekte per VBProject** erfahren Sie ab Seite 25, wie Sie mit der **VBProject**-Klasse auf die weiteren Elemente des VBA-Editors zugreifen können, hier speziell auf die Module, die Sie über die **VBComponents**-Auflistung erreichen. Außerdem können Sie hiermit beispielsweise die Projekteigenschaften einstellen oder ein Kennwort für das VBA-Projekt festlegen. Wie Sie mit den Modulen eines VBA-Projekts arbeiten, erfahren Sie ab Seite 30 im Beitrag **Module und Co. im Griff mit VBComponent**. Die **VBComponent**-Elemente bieten über die **CodeModule**-Eigenschaft Zugriff auf die eigentlichen Module. Sie können diese mit der **VBComponents**-Auflistung durchlaufen und darauf zugreifen sowie neue **VBComponent**-Elemente anlegen oder bestehende löschen.

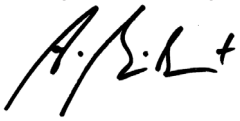
Was Sie genau mit den **CodeModule**-Elementen anfangen können, zeigt dann der Beitrag **VBA-Code manipulieren mit der CodeModule-Klasse** ab Seite 35. Das CodeModule-Element erlaubt den direkten Zugriff auf den Code der Module.

Etwas mehr in Richtung Benutzeroberfläche geht es im Beitrag **Auf VBA-Code zugreifen mit der CodePane-Klasse** ab Seite 47. Hier erfahren Sie beispielsweise, wie Sie den aktuell markierten Code im VBA-Fenster auslesen oder selbst per Code eine Markierung setzen.

Die Ausgabe hat jedoch noch mehr zu bieten: Im Beitrag **Export von Daten in das DATEV-Format** finden Sie ab Seite 60 eine spannende Lösung, mit der Sie die Daten aus Ihrer Buchhaltungsanwendung direkt in das DATEV-Format exportieren können, um es an Ihren Steuerberater weiterzuleiten.

Richtig praktisch wird es ab Seite 2 im Beitrag **Etiketten drucken mit Access**: Hier lernen Sie, wie Sie einen Bericht zum Ausdrucken auf einem Etikettendrucker erstellen und den Ausdruck vorbereiten können. Und unter **Feiertage mit VBA ermitteln** zeigen wir Ihnen ab Seite 7, wie Sie die Feiertage für ein Jahr für ein bestimmtes Bundesland ganz einfach per VBA ermitteln und sogar in eine Tabelle schreiben können.

Viel Spaß beim Ausprobieren!



Ihr André Minhorst

Etiketten drucken mit Access

Ein Kunde fragte neulich, warum ich nicht mal einen Beitrag darüber verfasse, wie man mit einem Thermodrucker beispielsweise Versandetiketten druckt. Also machen wir das einfach! Dieser Artikel zeigt am Beispiel des Brother-Druckers QL-700, wie Sie aus Access heraus Etiketten drucken können.

Adressdaten vorbereiten

Für die Ausgabe von Etiketten benötigen wir einige Daten, zum Beispiel Adressdaten, sowie einen Bericht, der die Ausgabe der Daten formatiert. Die Adressdaten verwalten wir in der Tabelle **tblAdressen**, die mit Daten wie in Bild 1 aussieht.

tblAdressen

AdresseID	Firma	Anrede	Vorname	Nachname	Strasse	PLZ	Ort
1	Krahn GbR	Herr	Adi	Stratmann	Kremser Straße 54	10589	Berlin
2	Göllner AG	Frau	Heidi	Eich	Moosstraße 30	42289	Wuppertal
3	Peukert GmbH & Co. KG	Herr	Wernfried	Birk	Wiener Straße 78	22297	Hamburg
4		Herr	Vitus	Krauße	Burgenlandstraße 77	20355	Hamburg
5	Mader KG	Frau	Jadwiga	Oehme	Peter-Rosegger-Straße	65187	Wiesbaden
6	Fleischhauer GmbH	Herr	Niko	Michel	Kindergartenstraße 42	12051	Berlin
7		Herr	Siegert	Loos	Lenaustraße 80	66115	Saarbrücken
8		Herr	Michl	Schroth	Dr. Karl Renner-Straße	01129	Dresden
9		Herr	Florentius	Wittek	Industriestraße 7	80638	München
10	Krahl KG	Herr	Gernulf	Riegel	Erzherzog-Johann-Stra	81545	München
11	Becker AG	Herr	Tristan	Hübsch	Jahnstraße 78	65193	Wiesbaden
12	Runge GmbH	Herr	Heinfried	Steinert	Kaplanstraße 60	30159	Hannover
13	Albert AG	Frau	Herma	Aigner	Burgstraße 31	22089	Hamburg
14		Frau	Mina	Dörfler	Schloßstraße 66	38118	Braunschweig
15	Quandt GmbH & Co. KG	Frau	Jo	Pickel	Kreuzstraße 29	79114	Freiburg
16		Herr	Heimbert	Rohrer	Lenaustraße 83	10785	Berlin
17	Lührs AG	Herr	Roderich	Reil	Flurstraße 100	40595	Düsseldorf

Datensatz: 23 von 100 | Kein Filter | Suchen

Bild 1: Adressdaten für die Adressetiketten

Die Adressen sind bis auf einige Adressen ohne Firma vollständig, weshalb wir im Bericht zwei verschiedene Formate vorsehen wollen:

- Adressen mit Firma: Firma in der ersten Zeile, Anrede, Vorname und Nachname in der zweiten Zeile, Straße in der dritten und PLZ und Ort in der vierten Zeile.

- Adressen ohne Firma: Anrede in der ersten Zeile, Vorname und Nachname in der zweiten Zeile, Straße in der dritten und PLZ und Ort in der vierten Zeile.

Bericht erstellen

Um den Bericht zu erstellen, der in dem gewünschten Format auf Etiketten ausgedruckt werden soll, legen wir zunächst einen einfachen Bericht in der Entwurfsansicht an. Diesem weisen wir als Datensatzquelle die Tabelle **tblAdressen** zu. Nun könnten wir jedes der Felder einzeln in den Detailbereich des Berichtsentwurfs ziehen, aber wenn wir die Darstellung mit wahlweise Firma oder Anrede in der ersten Zeile realisieren wollen, ist ein einziges Textfeld praktischer.

Nachdem wir dieses hinzugefügt haben, entfernen wir zuerst das Beschriftungsfeld des Textfeldes und stellen dann seine Eigenschaft **Steuerelementinhalt** auf den folgenden Ausdruck ein:

```
=Wenn(IstNull([Firma]);[Anrede];[Firma])
  & Zchn(13) & Zchn(10)
& Wenn(IstNull([Firma]);Null;[Anrede])+" "
  & [Vorname] & " " & [Nachname]
  & Zchn(13) & Zchn(10)
& [Strasse]
  & Zchn(13) & Zchn(10)
& [PLZ] & " " & [Ort]
```

Der Berichtsentwurf sieht danach wie in Bild 2 aus.

Der Ausdruck prüft zu Beginn, ob das Feld **Firma** für den aktuellen Datensatz **Null** ist. Falls ja, wird zuerst das Feld **Anrede** ausgeben, sonst **Firma**. **Zchn(13) & Zchn(10)** ist ein Zeilenumbruch und entspricht der Konstanten **vbCrLf**.

Der Ausdruck für die zweite Zeile prüft wieder, ob **Firma** den Wert **Null** hat. Falls ja, gibt der Wenn-Ausdruck den Wert **Null** aus, falls nein, den Wert des Feldes **Anrede**. Warum geben wir den Wert **Null** aus und nicht einfach eine leere Zeichenkette? Weil wir den Inhalt der **Wenn**-Bedingung per Plus-Zeichen (**+**) mit dem folgenden Teil **" " & Vorname & " " & Nachname** verknüpft haben. Und wenn der **Wenn**-Teil den Wert **Null** liefert, dann wird der komplette Ausdruck, der mit dem Plus-Zeichen verknüpft ist, zum Wert **Null**. Wenn die Anrede also in der zweiten Zeile nicht ausgegeben werden soll, weil sie schon in der ersten Zeile steht, würde dort sonst noch das Leerzeichen zwischen **Anrede** und dem **Vorname** erscheinen. Dies verhindern wir durch das Ausgeben von **Null** in dem Fall, dass die Anrede schon in der ersten Zeile ausgegeben wurde und die Verknüpfung mit dem Ausdruck **" "** durch ein Plus-Zeichen statt durch das Kaufmanns-Und (**&**).

Danach folgt noch eine weitere Zeile mit dem Wert des Feldes **Strasse** sowie in der letzten Zeile **PLZ** und **Ort**. Wechseln wir nun in die **Seitenansicht**, erhalten wir die Adressen wie in Bild 3.

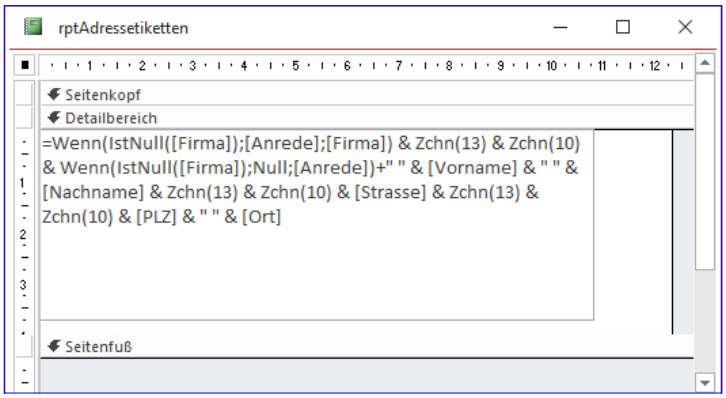


Bild 2: Erster Entwurf des Berichts

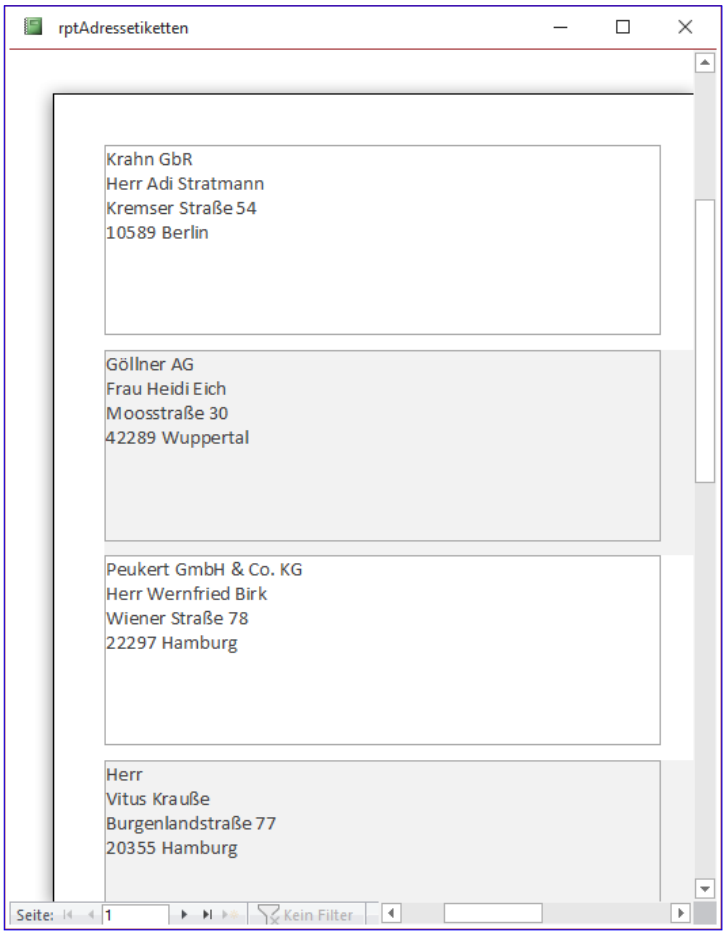


Bild 3: Adressdaten in der Seitenansicht

Bericht an die Adressetiketten anpassen

Damit folgt der interessante Teil der Aufgabe: Wir wollen den Bericht so anpassen, dass jeweils eine Adresse je Seite in der Größe der Etiketten angezeigt wird. Dazu öffnen wir den Bericht in der Entwurfsansicht und wechseln im Ribbon zum Reiter **Seite einrichten**. Hier klicken Sie auf **Seite einrichten**, was den Dialog **Seite einrichten** öffnet. In diesem wechseln Sie zur Registerseite **Seite**. Hier finden Sie die Option **Drucker für [Berichtsname]**. Wählen Sie hier statt **Standarddrucker** die Option **Spezieller Drucker** aus und klicken Sie anschließend auf die Schaltfläche **Drucker...** (siehe Bild 4).

Feiertage per VBA ermitteln

Wenn Sie mit Access Termine verwalten, werden Sie auch Feiertage berücksichtigen wollen. Die entsprechenden Datumsangaben liefert Access leider nicht frei Haus – Sie müssen selbst eine entsprechende Funktion bereitstellen. Der vorliegende Beitrag zeigt, wie Sie die Feiertage ermitteln und wie Sie schnell prüfen, ob ein Feiertag auf ein bestimmtes Datum fällt.

Es gibt verschiedene Gruppen von Feiertagen: Solche, die jedes Jahr auf den gleichen Termin fallen (wie Neujahr, Weihnachten oder Tag der deutschen Einheit), Tage, die vom Datum des Ostersonntags abhängen, das auf komplizierte Art berechnet wird, und Tage, die vom Datum des vierten Advents abhängen. Außerdem müssen Sie bei der Ermittlung von Feiertagen berücksichtigen, dass es einige Feiertage nur in bestimmten Bundesländern gibt.

So ist Heilige Drei Könige am 6. Januar nur in drei Bundesländern ein Feiertag, in den übrigen Bundesländern wird gearbeitet. Außerdem ändern sich die Feiertage der verschiedenen Bundesländer gelegentlich oder es kommen neue Feiertage hinzu – so ist der Internationale Frauentag als Feiertag noch recht neu und wird nur in Berlin begangen und der Weltkindertag findet als Feiertag nur in Thüringen statt.

Tabellen oder reiner Code?

Vor dem Zusammenstellen der Lösung zu diesem Beitrag stand die Frage im Raum, ob man die Basisdaten zu den Feiertagen in Tabellen speichert oder ob man diese komplett per VBA-Code ermittelt. In Anbetracht dessen, dass die Feiertage nur alle Jubeljahre geändert werden, haben wir uns für die reine VBA-Variante entschieden – auch vor dem Hintergrund, dass Sie so nur ein einziges Standardmodul in Ihre Anwendung kopieren müssen, wenn Sie die dynamische Ermittlung der Feiertage verwenden möchten. Der Einsatz einer reinen VBA-Lösung bedeutet zunächst, dass einige grundlegende Informationen in Enumerationen gespeichert werden. Die erste Enumeration heißt **eBundesland** und nimmt alle Bundesländer in Form entsprechender Konstanten und Zahlenwerte auf. Sie sieht wie folgt aus:

```
Public Enum eBundesland
    eBadenWuerttemberg = 1
    eBayern = 2
    eBerlin = 4
    eBrandenburg = 8
    eBremen = 16
    eHamburg = 32
    eHessen = 64
    eMecklenburgVorpommern = 128
    eNiedersachsen = 256
    eNordrheinWestfalen = 512
    eRheinlandPfalz = 1024
    eSaarland = 2048
    eSachsen = 4096
    eSachsenAnhalt = 8192
    eSchleswigHolstein = 16384
    eThueringen = 32768
End Enum
```

Warum nun erhalten die Konstanten für die Bundesländer ausschließlich Zweierpotenzen als Zahlenwerte? Nun: So können wir einfach festlegen und ermitteln, welcher Feiertag in welchem Bundesland gefeiert wird. Dazu nutzen wir eine zweite Enumeration, welche die Feiertage auflistet und deren Zahlenwert Informationen darüber liefert, welche Bundesländer den jeweiligen Feiertag begehen. Diese Enumeration können Sie in Kurz- oder Langform verwenden. Hier ist eine abgekürzte Version der Langform:

```
Public Enum eFeiertageBundeslaender
    eNeujahr = eBadenWuerttemberg + eBayern + eBerlin _
        + eBrandenburg + eBremen + eHamburg + eHessen + _
            eMecklenburgVorpommern + eNiedersachsen + _
```

```
    eNordrheinWestfalen + eRheinlandPfalz + eSaarland + _
    eSachsen + eSachsenAnhalt + eSchleswigHolstein + _
                                        eThueringen
    ...
    eWeltkindertag = eThueringen
    eInternationalerFrauentag = eBerlin
End Enum
```

Wie gut zu erkennen ist, werden zu jedem Feiertag die Konstanten aller Bundesländer summiert, in denen der jeweilige Feiertag stattfindet. Da wir den Konstanten der Bundesländer Zweierpotenzen zugewiesen haben, können wir theoretisch statt der etwas längeren Ausdrücke auch einen Zahlenwert angeben. Für Feiertage, die in allen Bundesländern stattfinden, wäre dies etwa 32.768 + 16.384 + 8.192 + 4.096 + 2.048 + 1.024 + 512 + 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1, also 65.535. Allerdings lassen sich die Informationen doch besser als Konstante prüfen.

Ermittlung des Datums des vierten Advents

Der vierte Advent ist der letzte Sonntag vor dem ersten Weihnachtstag und kann somit auch auf Heiligabend fallen. Der erste bis vierte Advent ist in allen Bundesländern Feiertag. Der erste, zweite und dritte Advent wird jeweils an den Sonntagen vor dem vierten Advent gefeiert, somit sind diese Feiertage vom Datum des vierten Advents abhängig. Auch der Buß- und Bettag, der allerdings nur in Sachsen Feiertag ist, hängt vom Datum des vierten Advents ab. Den vierten Advent berechnen wir so:

```
Function VierterAdvent(intJahr As Integer) As Date
    Dim dat As Date
    dat = CDate("24.12." & intJahr)
    Do While Not Weekday(dat, vbSunday) = vbSunday
        dat = dat - 1
    Loop
    VierterAdvent = dat
End Function
```

Diese Funktion stellt den Wert der Variablen **dat** einfach auf den 24. Dezember des Jahres ein, für welches das Datum des vierten Advents ermittelt werden soll, und wird in einer **Do While**-Schleife so lange um jeweils einen Tag in Richtung Jahresbeginn verschoben, bis es auf einen Sonntag fällt. Dies kann auch gleich beim ersten Durchlauf der Fall sein – dann fällt Heiligabend genau auf den vierten Advent. Die vom vierten Advent abhängigen Feiertage werden dann durch Subtraktion der entsprechenden Anzahl Tage ermittelt.

Ermittlung des Datums des Ostersonntags

Die Ermittlung des Ostersonntags erfolgt durch eine recht komplizierte Berechnung, die durch die folgende Funktion abgebildet wird:

```
Function Ostersonntag(intJahr As Integer) As Date
    Dim a As Integer
    Dim b As Integer
    Dim c As Integer
    Dim d As Integer
    Dim e As Integer
    Dim intTag As Integer
    Dim intMonat As Integer
    a = intJahr Mod 19
    b = intJahr Mod 4
    c = intJahr Mod 7
    d = (19 * a + 24) Mod 30
    e = (2 * b + 4 * c + 6 * d + 5) Mod 7
    intTag = 22 + d + e
    intMonat = 3
    If intTag > 31 Then
        intTag = d + e - 9
        intMonat = 4
    End If
    Ostersonntag = CDate(intTag & "." _
                        & intMonat & "." & intJahr)
End Function
```

Diese Funktion sorgt dafür, dass der Ostersonntag in Abhängigkeit von der Jahreszahl an einem Tag zwischen dem 23. März und dem 26. April liegt. Wenn man sich überlegt, dass Ostern (und die davon abhängigen Tage)

genau wie Weihnachten Jahrestage bestimmter Ereignisse sind, fragt man sich schon, warum hier keine einfachere Regelung gefunden wurde ...

Array der Feiertage zusammenstellen

Die grundlegenden Funktionen und Enumerationen haben wir nun zusammengestellt. Es fehlt noch eine Funktion, die alle Informationen zusammenführt und ein Array zurückliefert, das alle Feiertagsdaten für ein per Parameter angegebenes Jahr enthält. Ein zweiter Parameter gibt an, für welches Bundesland die Feiertage zusammengestellt werden sollen. Dies alles erledigt die Funktion **FeiertageArray** aus Listing 1. Der erste Parameter erwartet die Angabe einer Jahreszahl, also beispielsweise **2022**, der zweite Parameter eine der Konstanten der Enumeration der Bundesländer, also etwa **eNordrheinWestfalen**.

Die Funktion soll ein Array mit den Feiertagen zurückliefern, wobei der erste Wert den Namen des Feiertags enthält und der zweite das Datum des Feiertags. Die Daten der beiden Stichtage (Ostersonntag und der vierte Advent) werden in den Variablen **datOstersonntag** und **datVierterAdvent** gespeichert und später zur Berechnung der davon abhängigen Feiertage herangezogen. Die weiter oben vorgestellten Funktionen **Ostersonntag** und **VierterAdvent** füllen die beiden Datumsvariablen **datOstersonntag** und **datVierterAdvent** gleich zu Beginn der Funktion.

Dann prüft die Prozedur für jeden Feiertag, ob dieser in dem mit dem Parameter **lngBundesland** übergebenen Bundesland begangen wird. Dabei macht sich die Funktion die Enumerationen zunutze: Ein Ausdruck wie **(eNeujahr And lngBundesland) = lngBundesland** liefert so etwa den Wert **True** zurück, wenn der Feiertag im angegebenen Bundesland gefeiert wird. Wie funktioniert das? **eNeujahr** entspricht genau wie **lngBundesland** einem Zahlenwert. **eNeuJahr** hat den Zahlenwert **65.535**, das Bundesland **eNordrheinWestfalen** entspricht beispielsweise dem Wert **1**. Dadurch, dass wir den Bundesländern Zweierpotenzen als Wert zugewiesen haben, können wir durch einen Ausdruck wie **eNeujahr And lngBundesland** ermitteln, ob **eNordrheinWestfalen** in **eNeuJahr** enthalten ist. Diese arithmetische beziehungsweise binäre **Und**-Verknüpfung liefert genau den Wert der gemeinsamen Zweierpotenz, in diesem Fall **1** – dieses Bundesland begeht also diesen Feiertag. Wenn dies der Fall ist, ruft die Funktion eine weitere Funktion namens **FeiertagHinzufuegen** mit einigen Parametern auf:

```
FeiertagHinzufuegen i, arr, "Neujahr", ISODatum("1.1." & intJahr)
```

Der Parameter **i** enthält die laufende Nummer des aktuell abgearbeiteten Feiertags, **arr** ist das Array mit der zu füllenden Liste der Feiertage, der dritte Parameter enthält die Bezeichnung des Feiertags und der vierte das mit der Funktion **ISODatum** im Format **#yyyy/mm/dd#** formatierte Datum. Die Funktion **FeiertagHinzufuegen** sieht so aus:

```
Public Function FeiertagHinzufuegen(i, arr() As String, _
                strFeiertag As String, strDatum As String)
    ReDim Preserve arr(2, i)
    arr(0, i) = strFeiertag
    arr(1, i) = strDatum
    i = i + 1
End Function
```

Die Funktion fügt lediglich den Namen des Feiertags und das Datum zum Array hinzu und erhöht den Zähler **i** um **1**. Dadurch, dass alle Parameter standardmäßig mit **ByRef** übergeben werden, spiegeln sich die Änderungen direkt in den Variablen der aufrufenden Funktion wider.

Verwenden des Arrays

Das resultierende Array können Sie auf verschiedene Arten weiternutzen. Wenn Sie beispielsweise alle Feiertage einfach im Direktfenster ausgeben möchten, verwenden Sie die folgende Prozedur:

```
Public Function FeiertageAusgeben()
    Dim arr As Variant
```

Die Eval-Funktion

Die Eval-Funktion erlaubt das Auswerten von Ausdrücken, die als Parameter an diese Funktion übergeben werden. Damit können Sie sich verschiedene Anwendungszwecke erschließen – zum Beispiel die Eingabe von Berechnungen in einfache Textfelder oder das Ermitteln von Eigenschaften der Benutzeroberflächenelemente ohne Verwendung des VBA-Editors. Dieser Beitrag zeigt die Möglichkeiten der Eval-Funktion auf.

Eval unter VBA

Eine der einfachsten Einsatzmöglichkeiten der **Eval**-Funktion ist das Berechnen eines Ausdrucks, den Sie als Parameter der **Eval**-Funktion übergeben. Damit lässt sich dann beispielsweise die Summe aus **1** und **2** ermitteln:

```
Debug.Print Eval("1+2")
 3
```

Sie sehen schon am ersten Beispiel, dass wir den zu berechnenden Ausdruck in Anführungszeichen einfassen. Das ist erforderlich, weil die **Eval**-Funktion eigentlich nur Parameter des Typs **String** entgegennimmt. Bei als numerisch zu interpretierenden Parameterwerten, zum Beispiel einfache Zahlenwerte wie **12**, interpretiert die Funktion dies korrekt. Auch Berechnungen wie **1+2** werden noch ohne Angabe von Anführungszeichen als numerische Werte erkannt und toleriert.

Wenn Sie jedoch beispielsweise Funktionen wie **Date()** aufrufen, erhalten Sie ohne Anführungszeichen eine Fehlermeldung (Laufzeitfehler **2040**, **Der von Ihnen eingegebene Ausdruck enthält eine ungültige Zahl.**).

Das Berechnen von **1+2** ist nun noch kein Hexenwerk, denn das bekommen Sie auch ohne **Eval** hin:

```
Debug.Print 1+2
 3
```

Aber vielleicht haben Sie eine Funktion, die zwei **Integer**-Zahlen addiert:

```
Public Function Addieren(int1 As Integer, _
                         int2 As Integer) As Integer
    Addieren = int1 + int2
End Function
```

Dann können Sie diese ebenfalls per **Eval** aufrufen:

```
Public Sub EvalAddieren()
    Debug.Print Eval("Addieren(1,2)")
End Sub
```

Berechnung per InputBox

Wenn Sie dem Benutzer sehr einfach eine Berechnungsmöglichkeit bereitstellen wollen, gelingt dies mit der folgenden Prozedur. Diese fragt per **InputBox** den zu berechnenden Ausdruck ab und gibt das Ergebnis aus:

```
Public Sub EvalPerInput()
    Dim strEval As String
    strEval = InputBox("Zu berechnender Ausdruck:")
```

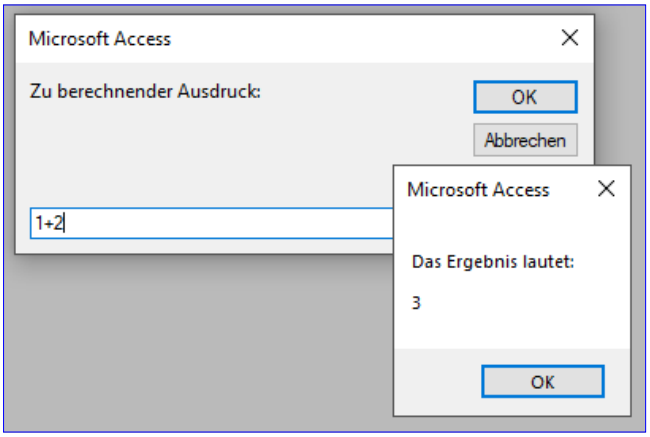


Bild 1: **Eval** per **InputBox**-Funktion

VBA-Projekt per VBA referenzieren

Access bietet nicht nur die Möglichkeit, Tabellen, Abfragen, Formulare und Berichte per VBA-Code zu erstellen. Sie können auch die Elemente, die Sie im VBA-Editor bearbeiten, per VBA erstellen, bearbeiten und wieder löschen. Dieser Beitrag macht den Start in eine Beitragsreihe, die sich mit den Möglichkeiten zur Programmierung des VBA-Editors und von VBA-Code beschäftigen. In diesem Teil schauen wir uns an, wie Sie überhaupt VBA-Projekte mit VBA referenzieren.

Voraussetzung: Extensibility-Verweis

Wenn Sie Routinen erschaffen wollen, mit denen Sie wiederum VBA-Elemente und -Code manipulieren möchten, benötigen Sie einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**.

Diesen fügen Sie im VBA-Editor mit dem **Verweise**-Dialog hinzu, den Sie mit dem Menübefehl **Extras|Verweise** öffnen (siehe Bild 1).

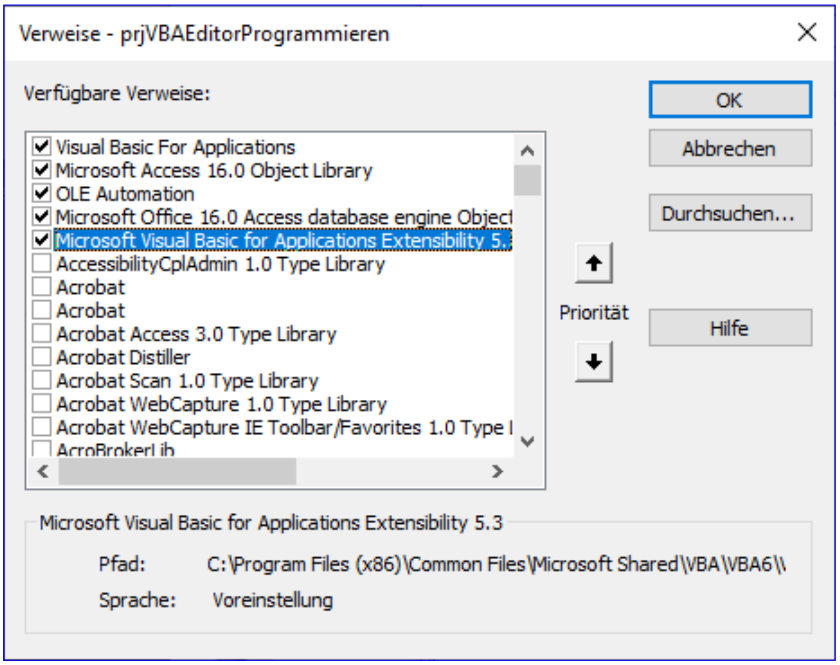


Bild 1: Verweis auf die **Extensibility**-Bibliothek

Access-Datenbank und VBA-Projekt

Eines der praktischen Dinge an Access ist, dass sowohl die Daten als auch die Elemente der Benutzeroberfläche und die Anwendungslogik normalerweise in einer einzigen Datei stecken. Natürlich gibt es Ausnahmen, wo die Tabellen in eine Backend-Access-Datei ausgelagert wurden oder wo die Tabellen Teil einer SQL Server-Datenbank sind. Und es gibt auch noch die Möglichkeit, dass Sie in einer Datenbankanwendung Elemente aus anderen Datenbankdateien nutzen – zum Beispiel, wenn Sie VBA-Code in eine Bibliotheksdatenbank auslagern.

Was jedoch immer garantiert ist, dass eine Access-Datenbankdatei auch ein VBA-Projekt enthält. Das VBA-Projekt speichert dabei die Module und Klassenmodule der Datenbankanwendung, wobei es zwei Typen von Klassenmodulen gibt – alleinstehende oder solche, die zu einem Formular oder einem Bericht gehören.

Derweil muss nicht jedes Formular und jeder Bericht ein Klassenmodul enthalten. Ein Klassenmodul zu einem Formular oder Bericht wird von Access erst durch eine der beiden folgenden Aktionen angelegt:

- wenn Sie die Eigenschaft **Enthält Modul** des Formulars oder Berichts auf **Ja** einstellen oder

- wenn Sie für das Formular oder den Bericht oder ein darin enthaltenes Steuerelement eine Ereignisprozedur definieren.

Aktuelles VBA-Projekt referenzieren

Um das aktuelle VBA-Projekt zu referenzieren, bedarf es keiner besonderen Tricks. Wir nutzen dazu die folgende Technik, nachdem wir den Namen unseres Beispielprojekts auf **prjVBAEditorProgrammieren** eingestellt haben.

Das erledigen Sie über den **Eigenschaften**-Bereich, während Sie im Projektexplorer den Eintrag für das Projekt der aktuellen Datenbank markiert haben (siehe Bild 2).

Danach probieren Sie den folgenden Code aus:

```
Public Sub VBProjectReferenzieren()
    Dim objVBProject As VBProject
    Set objVBProject = VBE.ActiveVBProject
    Debug.Print objVBProject.Name
End Sub
```

Die Prozedur deklariert eine Variable des Typs **VBProject** und füllt diese mit der Eigenschaft **ActiveVBProject** der Klasse **VBE**. Die Klasse **VBE** ist das oberste Element der Bibliothek zur Programmierung des Visual Basic Editors.

Nachdem wir die Variable **objVBProject** gefüllt haben, können wir damit beispielsweise auf den Namen des Projekts zugreifen. Diesen gibt die Prozedur im vorliegenden Fall wie in Bild 3 aus.

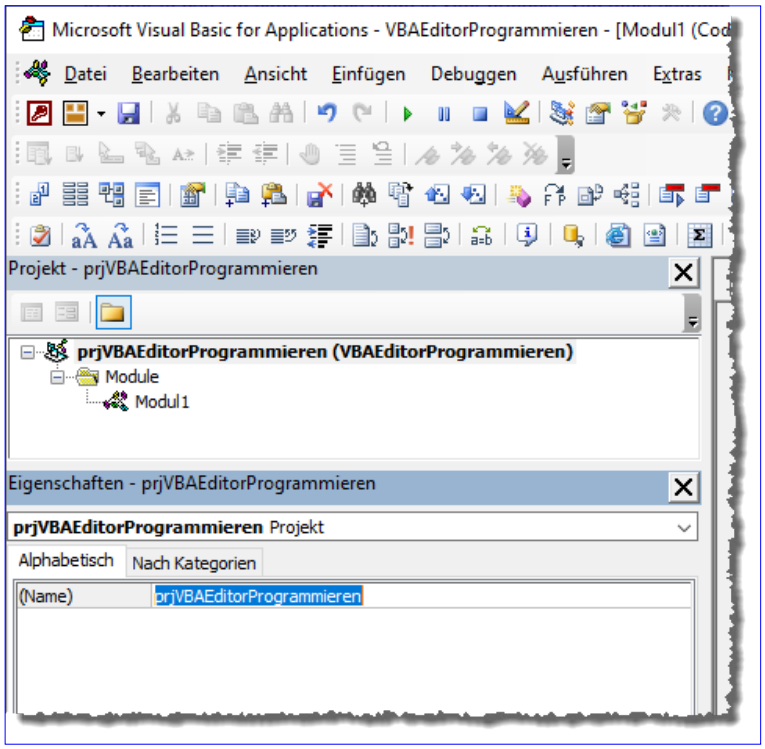


Bild 2: Ändern des Namens des VBA-Projekts

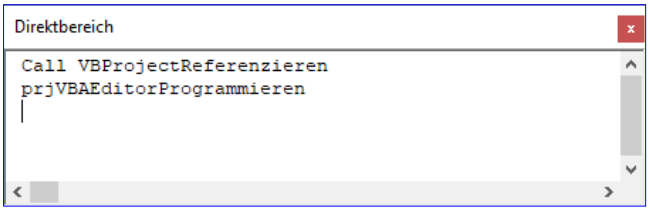


Bild 3: Ausgabe des Namens des VBA-Projekts

Mehrere VBA-Projekte im VBA-Editor?

Aber referenzieren wir mit der Eigenschaft **ActiveVBProject** auch tatsächlich immer das **VBProject**-Objekt der aktuellen Datenbank?

Nein, das ist nicht sichergestellt. Um das zu reproduzieren, wechseln Sie einmal zum Access-Fenster und legen ein neues, leeres Formular in der Entwurfsansicht an. Dann klappen Sie im Ribbon unter **Entwurf|Steuerelemente** die Liste der Steuerelemente auf und aktivieren dort die Option **Steuerelement-Assistent verwenden** (siehe Bild 4).

Danach fügen Sie beispielsweise eine neue Schaltfläche hinzu, was den dazugehörigen Assistenten aktiviert. Das war es schon auf der Access-Seite – Sie können den Assistenten nun abbrechen und wieder zum VBA-Editor wechseln.

Hier finden Sie nun plötzlich zwei VBA-Projekte vor – zusätzlich zu dem der aktuellen Access-Datenbank finden

Zugriff auf den VBA-Editor mit der VBE-Klasse

Die VBE-Klasse ist die Schaltzentrale, wenn es darum geht, die Elemente des VBA-Editors und von VBA-Projekten per VBA zu programmieren. Die Klasse ist Teil einer eigenen Bibliothek namens Microsoft Visual Basic for Applications Extensibility 5.3 Object Library. Diese stellt alle Elemente, Methoden und Eigenschaften zur Verfügung, um die im VBA-Editor bearbeitbaren Elemente zu erstellen, zu bearbeiten oder zu löschen. Dieser Beitrag stellt die Eigenschaften und Auflistungen der VBE-Klasse vor und zeigt, wo Sie weitergehende Informationen zu den einzelnen Elementen finden.

Vorbereitung

Um die Elemente der Klasse **VBE** nutzen zu können, benötigen Sie einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**, den Sie im **Verweise**-Dialog des VBA-Editors hinzufügen können (Menüeintrag **Extras|Verweise**).

Die VBE-Klasse

Um sich einen Überblick über die Elemente einer Klasse zu verschaffen, ist der Objektkatalog (zu öffnen mit **F2**) immer eine gute Anlaufstelle. Hier wählen Sie oben den Eintrag **VBIDE** und selektieren dann links unter **Klassen** den Eintrag **VBE**. Die Elemente der Klasse erscheinen dann im rechten Bereich (siehe Bild 1). Im Einzelnen finden wir dort die folgenden Elemente vor:

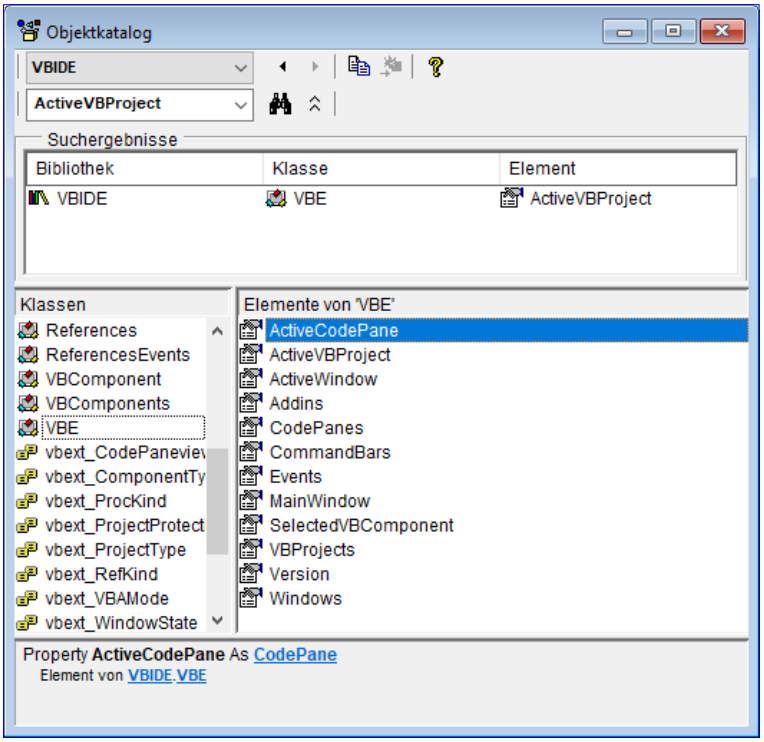


Bild 1: Elemente der VBE-Klasse im Objektkatalog

- **ActiveCodePane**: Verweis auf das **CodePane**-Element, das aktuell den Fokus hat.
- **ActiveVBProject**: Verweis auf das aktive VBA-Projekt, also das Projekt, von dem aus der Aufruf erfolgt.
- **ActiveWindow**: Verweis auf das Window-Objekt, das im VBA-Fenster aktuell den Fokus hat.
- **AddIns**: Auflistung der aktuell verfügbaren Add-Ins im VBA-Editor (nicht zu verwechseln mit den Access-Add-Ins)
- **CodePanes**: Auflistung der **CodePane**-Objekte, die aktuell geöffnet sind
- **CommandBars**: Auflistung der **CommandBar**-Objekte des VBA-Editors
- **Events**: Ermöglicht den Zugriff auf Ereignisse von **CommandBar**- und **Reference**-Elementen
- **MainWindow**: Verweis auf das VBA-Fenster

- **SelectedVBComponent**: Gibt das aktuell im Projektexplorer selektierte Element zurück, falls es sich um ein Element des Typs **VBComponent** handelt – anderenfalls lautet das Ergebnis **Nothing**.

- **VBProjects**: Liefert eine Auflistung der **VBProject**-Elemente, die aktuell im Projektexplorer des VBA-Editors angezeigt werden.

- **Version**: Gibt die Version des VBA-Editors aus.

- **Windows**: Liefert eine Auflistung der **Window**-Elemente des VBA-Editors.

Unterschied zwischen Window, CodePane und VBComponent

In der Auflistung haben wir bereits mehrere Elemente kennengelernt, die per Auflistung und auch jeweils als aktives Element ermittelt werden können. Dabei sind die **VBComponent**-Elemente unter Access die Elemente, die Sie im Projektexplorer in den Bereichen **Microsoft Access Klassenobjekte**, **Module** und **Klassenmodule** finden (siehe Bild 2).

Window-Elemente sind alle Fenster, die innerhalb des VBA-Editors angezeigt werden. Dabei handelt es sich nicht nur um die Fenster, die Code enthalten, sondern auch die **ToolWindow**-Elemente, also die Fenster, die Sie links, rechts, oben oder unten verankern können. **CodePane**-Elemente sind Container in Fenstern, die Code von **VBComponent**-Elementen enthalten. Die Zusammenhänge werden wir in den folgenden Beispielen noch aufschlüsseln.

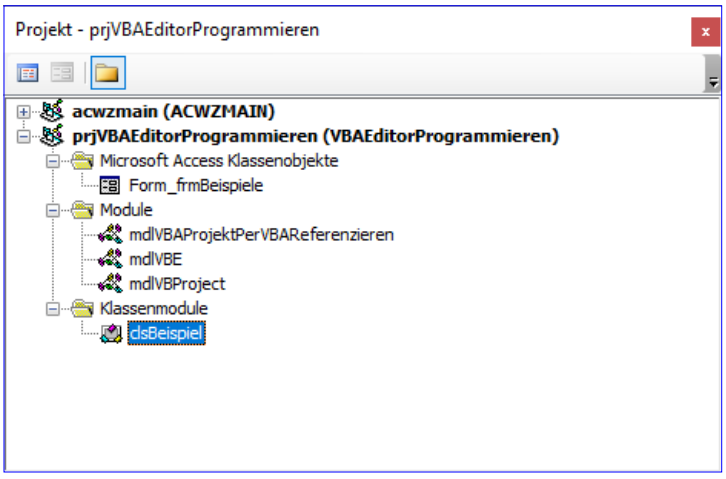


Bild 2: Die verschiedenen **VBComponent**-Elemente

Mit Window-Elementen arbeiten

Mit der **Windows**-Auflistung können wir alle **Window**-Elemente des VBA-Editors durchlaufen. Das erledigen wir in der folgenden Prozedur:

```
Public Sub WindowsAuflisten()
    Dim objWindow As Window
    For Each objWindow In VBE.Windows
        Debug.Print objWindow.Caption, objWindow.Type
    Next objWindow
End Sub
```

Dies erzeugt, wenn einige Codefenster geöffnet sind, die Ausgabe aus Bild 3. Uns interessiert nun, welche Bedeutung die Werte für die Eigenschaft **Type** haben.

Dazu ermitteln wir die Konstanten für diese Eigenschaft, was am schnellsten gelingt, wenn wir im Objektkatalog nach der Klasse **VBIDE** filtern und dort nach Elementen mit dem Teilausdruck **Type** suchen. Damit gelangen wir zur Auflistung **vbext_WindowType**, welche die Werte aus Bild 4 offenbart.

Damit können wir die Prozedur **WindowsAuflisten** wie folgt verfeinern:

```
Public Sub WindowsAuflisten()
    Dim objWindow As Window
    Dim strWindowType As String
    For Each objWindow In VBE.Windows
        Select Case objWindow.Type
            Case 0
                strWindowType = "CodeWindow"
            Case 1
```

Zugriff auf VBA-Projekte per VBProject

Die Klasse VBProject des Objektmodells zum Programmieren des VBA-Editors und der enthaltenen Elemente bietet einige interessante Eigenschaften, Methoden und Auflistungen. Diese schauen wir uns im vorliegenden Beitrag an. Hier wird deutlich, dass die VBProjects im VB-Editor nicht nur für Access-Datenbanken genutzt werden können, sondern auch noch für andere Anwendungen – es gibt nämlich einige Elemente, die unter Access nicht funktionieren.

Vorbereitung

Um die Elemente der Klasse **VBE** nutzen zu können, benötigen Sie einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**, den Sie im **Verweise**-Dialog des VBA-Editors hinzufügen können (Menüeintrag **Extras|Verweise**).

Elemente der VBProject-Klasse

Die **VBProject**-Klasse liefert die folgenden Methoden, Auflistungen und Eigenschaften:

- **BuildFileName**: Gibt den Namen einer DLL zurück. Diese Eigenschaft hat unter Access keine Verwendung, da hier keine DLL erstellt werden kann.

- **Collection**: Liefert einen Verweis auf die Collection, in der sich das **VBProject**-Element befindet. Diese sollte normalerweise nur ein Element enthalten, nämlich das aktuelle VBA-Projekt. Wenn Sie jedoch beispielsweise seit dem Start der aktuellen Access-Session ein Access-Add-In verwendet haben, finden Sie auch dessen **VBProject**-Element in der Auflistung.

- **Description**: Diese Eigenschaft ist standardmäßig leer und kann in den Projekteigenschaften eingestellt werden (siehe weiter unten).

- **FileName**: Liefert den Pfad zu der Datei, in der das aktuelle VBProject-Objekt gespeichert ist – in der Regel also die Datenbankdatei, von der aus Sie den VBA-Editor mit dem aktuellen Projekt geöffnet haben.

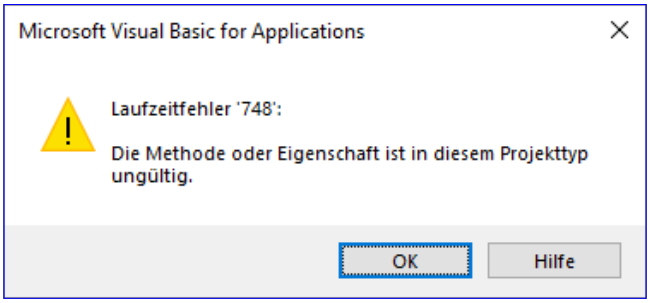


Bild 1: Fehler beim Aufruf der Methode **MakeCompiledFile**

- **MakeCompiledFile**: Führt beim Aufruf aus einem VBA-Projekt zu dem Fehler aus Bild 1 und ist laut Dokumentation zum Erstellen einer DLL für das aktuelle Projekt gedacht – was aber wohl nicht für VBA-Projekte von Access-Datenbanken gilt.

- **Mode**: Gibt den Modus des Projekts an. Kann die drei Werte **vbext_vm_Run** (**0**), **vbext_vm_Break** (**1**) oder **vbext_vm_Design** (**2**) annehmen. Auch diese Eigenschaft lässt sich in VBA-Projekten unter Access nicht sinnvoll nutzen. Wenn Sie einen Haltepunkt setzen, dieser im Code erreicht wird und Sie dann im Direktbereich **Debug.Print VBE.ActiveVBProject.Mode** ausgeben lassen, erhalten Sie dennoch den Wert **0**.

- **Name**: Liefert den Namen des VBA-Projekts, so wie er auch in den Eigenschaften festgelegt ist.

- **Protection**: Kann die Werte **vbext_pp_locked** (**1**) oder **vbext_pp_none** (**0**) annehmen. Standardmäßig ist der Wert **0** eingestellt. Diesen Wert ändern Sie durch Vergeben eines Kennworts für das VBA-Projekt.

- **References**: Liefert eine Auflistung der Verweise des aktuellen VBA-Projekts.
- **SaveAs**: Nicht für Access-VBA-Projekte verfügbar.
- **Saved**: Gibt an, ob es noch ungespeicherte Änderungen im VBA-Projekt gibt. **True** bedeutet, dass alle Änderungen gespeichert sind, **False**, dass noch nicht gespeicherte Änderungen vorliegen.
- **Type**: Gibt den Typ des VBA-Projekts zurück. Es gibt die beiden Werte **vbext_pt_HostProject** (**100**) und **vbext_pt_Standalone** (**101**).
- **VBComponents**: Liefert eine Auflistung aller **VBComponent**-Objekte.
- **VBE**: Verweis auf das übergeordnete VBE-Objekt

Projekteigenschaften einstellen

Die weiter oben erwähnte Eigenschaft **Description** ist standardmäßig leer. Sie können diese füllen, indem Sie den Dialog **[Projektname] - Projekteigenschaften** öffnen, was Sie mit dem Menüeintrag **Extras|Eigenschaften von [Projektname]...** erledigen. Hier finden Sie auf der ersten Seite gleich die Eigenschaft **Projektbeschreibung**, die Sie mit einem beliebigen Text füllen können (siehe Bild 2).

Anschließend rufen Sie diesen Text mit der folgenden Anweisung beispielsweise im Direktbereich ab:

```
? VBE.ActiveVBProject.Description
Dies ist eine Projektbeschreibung.
```

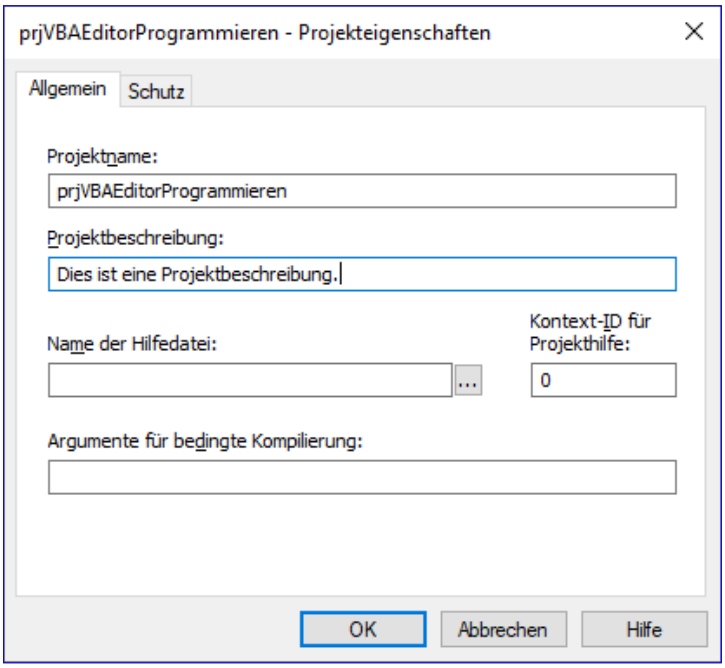


Bild 2: Einstellen der Beschreibung eines VBA-Projekts

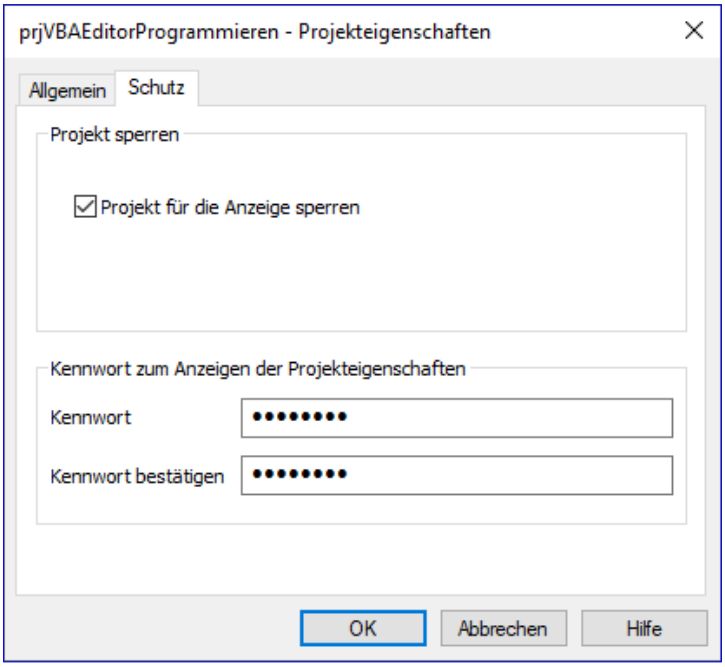


Bild 3: Festlegen des Kennwortschutzes

Kennwortschutz aktivieren und abfragen

Den Kennwortschutz aktivieren Sie im gleichen Dialog, in dem Sie auch die Projektbeschreibung eingeben – allerdings auf der zweiten Registerseite unter **Schutz**. Um den Schutz zu aktivieren, setzen Sie einen Haken für die Option **Projekt für die Anzeige** sperren (siehe Bild 3). Danach können Sie unten das Kennwort eingeben und nochmals bestätigen.

Module und Co. im Griff mit VBComponent

Bei der Programmierung des VBA-Editors per VBA ist eine der Kernkomponenten das Element VBComponent. Wir können diese mit der Auflistung VBComponents durchlaufen oder direkt über den Namen der Komponente oder den Index darauf zugreifen. Danach ergeben sich verschiedene Möglichkeiten, die erst mit dem Zugriff auf das CodeModule des VBComponent-Elements interessant werden. Bis dahin schauen wir uns aber noch an, welche Möglichkeiten das VBComponent-Element bietet.

Vorbereitung

Um die Elemente der Klasse **VBE** nutzen zu können, benötigen Sie einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**, den Sie im **Verweise**-Dialog des VBA-Editors hinzufügen können (Menüeintrag **Extras|Verweise**).

Elemente der VBComponent-Klasse

Die **VBComponent**-Klasse und ihre Eigenschaften, Methoden und Auflistungen können Sie im Objektkatalog im Überblick ansehen, wenn Sie dort nach **VBComponent** suchen (siehe Bild 1). Die Klasse bietet folgende Elemente:

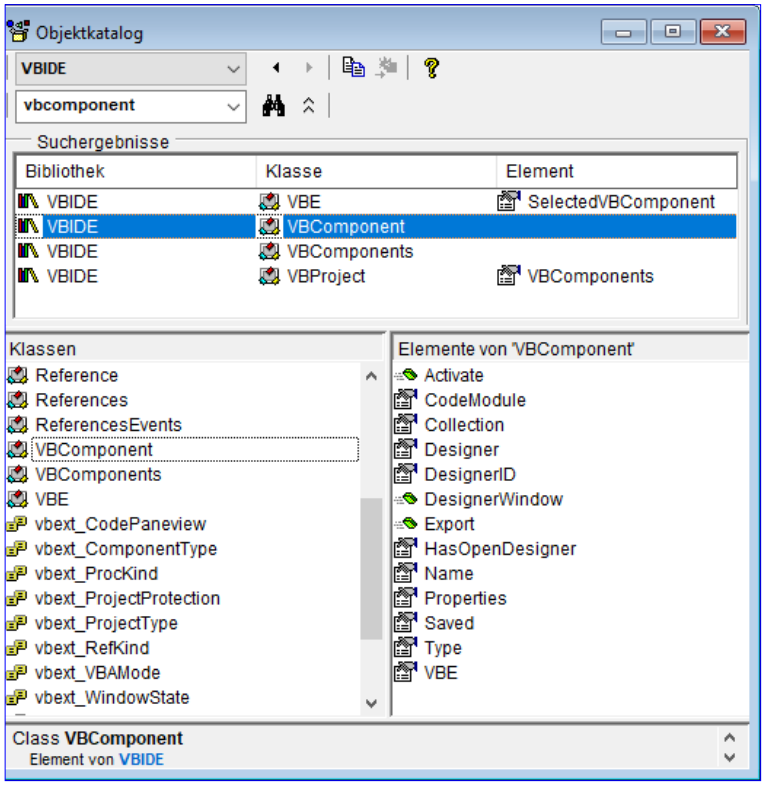


Bild 1: Das **VBComponent**-Element im Objektkatalog

- **Activate**: Aktiviert das **VBComponent**-Objekt und zeigt es in seinem Fenster an.
- **CodeModule**: Liefert einen Verweis auf das **CodeModule**-Element des **VBComponent**-Objekts.
- **Collection**: Liefert Zugriff auf die Collection mit diesem **VBComponent**-Objekt und allen anderen, die in der übergeordneten **VBComponents**-Auflistung enthalten sind.
- **Designer**: Liefert einen Verweis auf den Designer des **VBComponent**-Objekts. Dieser liefert bei den üblicherweise unter Access verwendeten **VBComponent**-Elementen immer **Nothing**. Sie können ihn nutzen, wenn Sie beispielsweise **UserForm**-Objekte programmieren wollen, was nicht Thema dieses Beitrags ist und normalerweise unter Access nicht geschieht.
- **DesignerID**: Siehe Eigenschaft **Designer**.
- **DesignerWindow**: Siehe Eigenschaft **Designer**.
- **Export**: Exportiert das Objekt in eine Textdatei.

- **HasOpenDesigner**: Siehe **Designer**. Liefert daher für die standardmäßig verwendeten Access-Module immer den Wert **False**.
- **Name**: Liefert den Namen des **VBComponent**-Objekts.
- **Properties**: Liefert eine Auflistung der **Property**-Eigenschaften des **VBComponent**-Elements.
- **Saved**: Gibt an, ob es ungespeicherte Änderungen an dem **VBComponent**-Objekt gibt.
- **Type**: Gibt den Typ des **VBComponent**-Objekts zurück.
- **VBE**: Verweis auf das VBA-Editor-Objekt des **VBComponent**-Objekts

VBComponent-Objekte durchlaufen

Die folgende Prozedur referenziert mit der Variablen **objVBProject** das aktuelle VBA-Projekt und durchläuft dann in einer **For Each**-Schleife die Elemente der Auflistung **VBComponents**. Dabei speichert sie den Verweis auf das jeweils aktuelle Objekt in der Variablen **objVBComponent**. Damit gibt die Prozedur den Namen des **VBComponent**-Objekts aus:

```
Public Sub VBComponentsDurchlaufen()
    Dim objVBProject As VBProject
    Dim objVBComponent As VBComponent
    Set objVBProject = VBE.ActiveVBProject
    For Each objVBComponent In objVBProject.VBComponents
        Debug.Print objVBComponent.Name
    Next objVBComponent
End Sub
```

Das Ergebnis entspricht den Elementen, die Sie in den drei Ordnern **Microsoft Access Klassenobjekte**, **Module** und **Klassenmodule** des Projektexplorers sehen, zum Beispiel:

```
mdlVBAProjektPerVBAReferenzieren
mdlVBProject
mdlVBE
Form_frmBeispiele
clsBeispiel
mdlVBComponent
```

VBComponent-Objekte gezielt referenzieren

Wenn Sie wissen, auf welches **VBComponent**-Objekt Sie zugreifen wollen, beispielsweise um sein **CodeModule**-Objekt zu bearbeiten, können Sie dies auch über den Index oder den Namen erledigen. Der Index ist 1-basiert:

```
? VBE.ActiveVBProject.VBComponents(1).Name
mdlVBAProjektPerVBAReferenzieren
```

Das Referenzieren per Name geht so – hier mit Ausgabe des Wertes der Eigenschaft **Saved**:

```
? VBE.ActiveVBProject.VBComponents("mdlVBE").Saved
Wahr
```

Neues VBComponent-Objekt hinzufügen

Um ein neues **VBComponent**-Objekt hinzuzufügen und somit ein Klassenmodul eines Formulars oder Berichts, ein Standardmodul oder ein alleinstehendes Klassenmodul, benötigen Sie die **Add**-Methode der **VBComponents**-Auflistung. Dieser übergeben Sie lediglich den Typ der zu erstellenden Komponente (siehe Bild 2).

Das Ergebnis referenzieren wir beispielsweise mit der Variablen **objVBComponent** und bearbeiten dieses anschließend weiter. Dabei geben wir im folgenden Beispiel zunächst nur den Namen der Komponente an:

```
Public Sub VBComponentNeu()
    Dim objVBProject As VBProject
    Dim objVBComponent As VBComponent
    Set objVBProject = VBE.ActiveVBProject
    Set objVBComponent = objVBProject.VBComponents.
                                Add(vbext_ct_StdModule)
    With objVBComponent
        .Name = "mdlNeu"
```

VBA-Code manipulieren mit der CodeModule-Klasse

Wenn Sie sich mit den anderen Beiträgen dieser Reihe bis zum VBComponent-Objekt eines Moduls vorgearbeitet haben, ist es nur noch ein Katzensprung bis zur CodeModule-Klasse. Damit können Sie dann die Inhalte eines VBA-Moduls auslesen und bearbeiten. Dieser Beitrag zeigt, welche Methoden die CodeModule-Klasse bietet und wie Sie diese für die verschiedenen Anwendungszwecke einsetzen können.

Vorbereitung

Um die Elemente der Klasse **VBE** nutzen zu können, benötigen Sie einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**, den Sie im **Verweise**-Dialog des VBA-Editors hinzufügen können (Menüeintrag **Extras|Verweise**).

Vorbereitende Beiträge

Wenn Sie erfahren wollen, wie Sie überhaupt bis zu der hier beschriebenen Klasse gelangen, helfen die folgenden Beiträge weiter:

- **VBA-Projekt per VBA referenzieren** (**www.access-im-unternehmen.de/1337**)
- **Zugriff auf den VBA-Editor mit der VBE-Klasse** (**www.access-im-unternehmen.de/1350**)
- **Zugriff auf VBA-Projekte per VBProject** (**www.access-im-unternehmen.de/1351**)
- **Module und Co. im Griff mit VBComponent** (**www.access-im-unternehmen.de/1352**)

Elemente der VBComponent-Klasse

Die **CodeModule**-Klasse und ihre Eigenschaften, Methoden und Auflistungen können Sie im Objektkatalog im Überblick ansehen, wenn Sie dort nach **CodeModule** suchen (siehe Bild 1). Hier sehen wir neben den Elementen dieser Klasse auch, dass diese sowohl ein Unterelement von **VBComponent** als auch von **CodePane** ist. Als Element von **VBComponent** können Sie es anlegen, und **CodePane** ist die Schnittstelle zwischen dem anzeigenden **Window**-Element im VBA-Editor und dem **CodeModule**-Objekt, das einige Möglichkeiten für den Zugriff auf den Code über die Benutzeroberfläche ermöglicht.

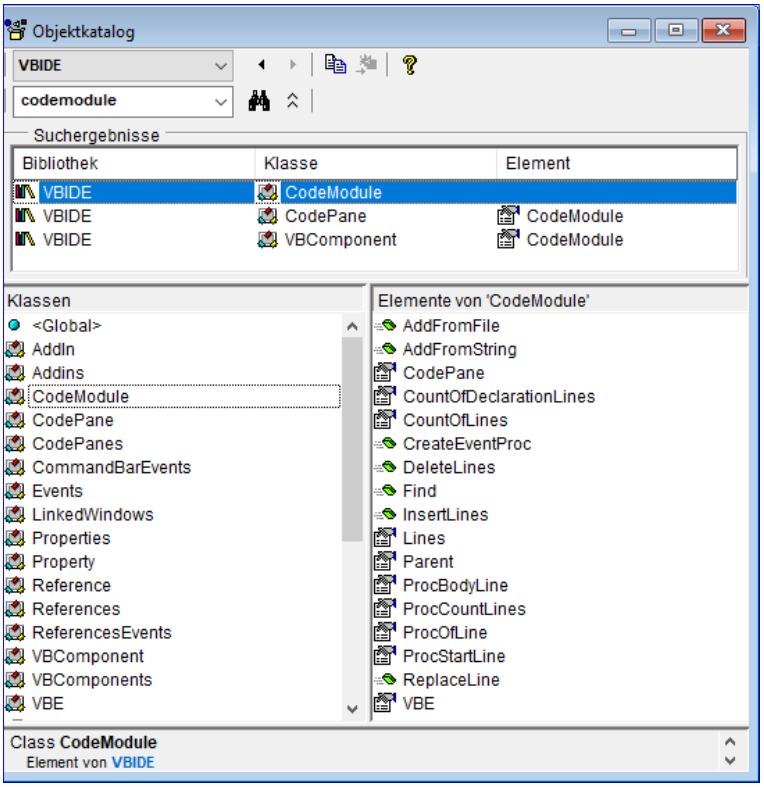


Bild 1: Die **CodeModule**-Klasse im Objektkatalog

Die Klasse **CodeModule** bietet folgende Elemente:

- **AddFromFile**: Fügt VBA-Code aus der per Parameter angegebenen Textdatei hinzu.

- **AddFromString**: Fügt VBA-Code aus der per Parameter übergebenen Zeichenkette hinzu.
- **CodePane**: Liefert einen Verweis auf das **CodePane**-Element, mit dem Sie speziell Zugriffsmöglichkeiten auf die angezeigte Version des **CodeModule**-Objekts haben – beispielsweise um Markierungen abzufragen oder zu setzen.
- **CountOfDeclarationLines**: Liefert die Anzahl der Zeilen im Deklarationsbereich des Moduls.
- **CountOfLines**: Liefert die gesamte Anzahl der Zeilen im Modul.
- **CreateEventProc**: Erstellt eine Ereignisprozedur für das mit den beiden Parametern angegebene Ereignis und Objekt.
- **DeleteLines**: Löscht die mit dem zweiten Parameter angegebene Anzahl von Zeilen ab der mit dem ersten Parameter angegebenen Zeile.
- **Find**: Sucht nach dem mit dem ersten Parameter angegebenen Ausdruck und liefert mit den folgenden Parametern die Position des Fundorts zurück.
- **InsertLines**: Fügt ab der mit dem ersten Parameter angegebenen Zeile den mit dem zweiten Parameter angegebenen Text im Modul ein.
- **Lines**: Liefert den Inhalt einer oder mehrerer Zeilen, wobei der erste Parameter die Nummer der ersten Zeile und der zweite die Anzahl der Zeilen angibt.
- **Parent**: Referenziert das übergeordnete Objekt, in diesem Fall ein Objekt des Typs **VBComponent**.
- **ProcBodyLine**: Liefert die Nummer der Zeile der angegebenen Prozedur mit dem **Sub|Function|Property**-Schlüsselwort.
- **ProcCountLines**: Liefert die Anzahl der Zeilen einer Prozedur.
- **ProcOfLine**: Gibt die Prozedur und den Typ der Prozedur für eine bestimmte Zeile zurück.
- **ProcStartLine**: Liefert die Nummer der ersten Zeile nach der letzten Zeile der vorherigen Prozedur oder des allgemeinen Deklarationsteils.
- **ReplaceLine**: Ersetzt die Zeile mit der im ersten Parameter angegebenen Nummer durch den mit dem zweiten Parameter angegebenen Text.
- **VBE**: Referenziert das VBA-Editor-Objekt des **CodeModule**-Objekts.

Das CodeModule-Objekt referenzieren

Um das **CodeModule**-Objekt eines Moduls zu referenzieren, benötigen Sie zuerst Zugriff auf das entsprechende **VBComponent**-Element. Meist wollen Sie direkt auf ein bestimmtes Objekt zugreifen, dessen Namen Sie kennen. Dann können Sie dieses über die **VBComponents**-Auflistung des **VBProject**-Elements referenzieren.

Das **VBComponent**-Element bietet dann über die **CodeModule**-Eigenschaft die Möglichkeit des Zugriffs auf das **CodeModule**-Objekt an. Im folgenden Beispiel referenzieren wir dieses und geben dann die Anzahl der Codezeilen in diesem **CodeModule**-Objekt aus:

```
Public Sub CodeModuleReferenzieren()
    Dim objVBProject As VBProject
    Dim objVBComponent As VBComponent
    Dim objCodeModule As CodeModule
    Set objVBProject = VBE.ActiveVBProject
    Set objVBComponent = 7
                        objVBProject.VBComponents("mdlVBE")
    Set objCodeModule = objVBComponent.CodeModule
    Debug.Print objCodeModule.CountOfLines
End Sub
```

Code aus einer Textdatei hinzufügen

Mit der **Export**-Methode des **VBComponent**-Objekts können Sie eine Textdatei mit dessen Inhalt exportieren. Diese können Sie beispielsweise mit der **AddFromFile**-Methode der **CodeModule**-Klasse wieder einlesen.

Wie das gelingt, zeigen wir in folgendem Beispiel. Hier legen wir ein neues, leeres **VBComponent**-Objekt an und nutzen dann die **AddFromFile**-Methode seines **CodeModule**-Objekts, um den Inhalt des exportierten Moduls in das neue Modul einzulesen:

```
Public Sub CodeModuleAddFromFile()
    Dim objVBProject As VBProject
    Dim objVBComponent As VBComponent
    Dim objCodeModule As CodeModule
    Set objVBProject = VBE.ActiveVBProject
    Set objVBComponent = _
        objVBProject.VBComponents.Add(vbext_ct_StdModule)
    Set objCodeModule = objVBComponent.CodeModule
    objCodeModule.AddFromFile CurrentProject.Path _
                                        & "\Neu.txt"
End Sub
```

Zu beachten ist hier, dass bei exportierten Modulen auch die Attribute mit exportiert werden, also zum Beispiel der Modulname.

Dieser wird beim Importieren in ein vorhandenes Modul dann für das übergeordnete **VBComponent**-Objekt verwendet.

Sie können mit **AddFromFile** jedoch auch Textdateien mit dem reinen Inhalt des Moduls laden.

Code per Zeichenkette hinzufügen

Gegebenenfalls stellen Sie den Code für ein Modul in einer Textvariablen zusammen oder lesen diesen von anderer Stelle ein, beispielsweise aus einem Feld einer Datenbanktabelle. Wenn Sie den Inhalt der Variablen schnell in ein neues, leeres Modul einfügen möchten, bietet sich die Methode **AddFromString** an. Diese schreibt den Code direkt in das **CodeModule**-Objekt.

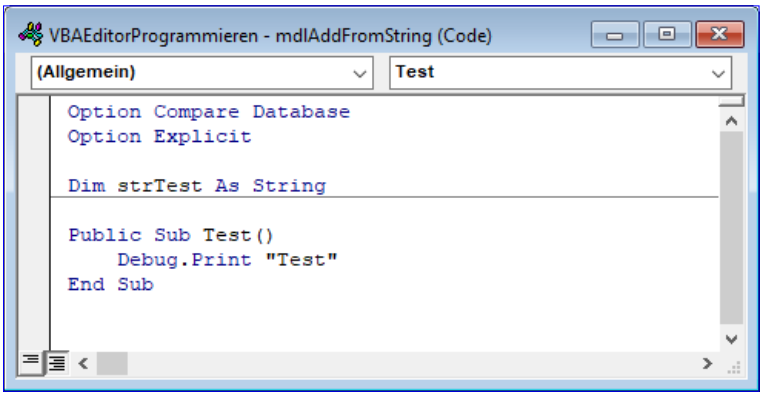


Bild 2: Per **AddFromString** initial hinzugefügter Code

Im folgenden Beispiel stellen wir den einzufügenden Code zuvor in der Variablen **strCode** zusammen und weisen diesen dann mit **AddFromString** dem **CodeModule**-Objekt zu:

```
Public Sub CodeModuleAddFromString()
    Dim objVBProject As VBProject
    Dim objVBComponent As VBComponent
    Dim objCodeModule As CodeModule
    Dim strCode As String
    Set objVBProject = VBE.ActiveVBProject
    Set objVBComponent = objVBProject.VBComponents._
                                Add(vbext_ct_StdModule)
    objVBComponent.Name = "mdlAddFromString"
    Set objCodeModule = objVBComponent.CodeModule
    strCode = "Dim strTest As String" & vbCrLf & vbCrLf
    strCode = strCode & "Public Sub Test()" & vbCrLf
    strCode = strCode & "    Debug.Print ""Test""" _
                                        & vbCrLf
    strCode = strCode & "End Sub"
    objCodeModule.AddFromString strCode
End Sub
```

Das Ergebnis finden Sie in Bild 2.

Sie können **AddFromString** auch nutzen, um Code in ein Modul einzufügen, das bereits Code enthält. Der neu einzufügende Code landet dann genau hinter der letzten Deklarationszeile im Modul.

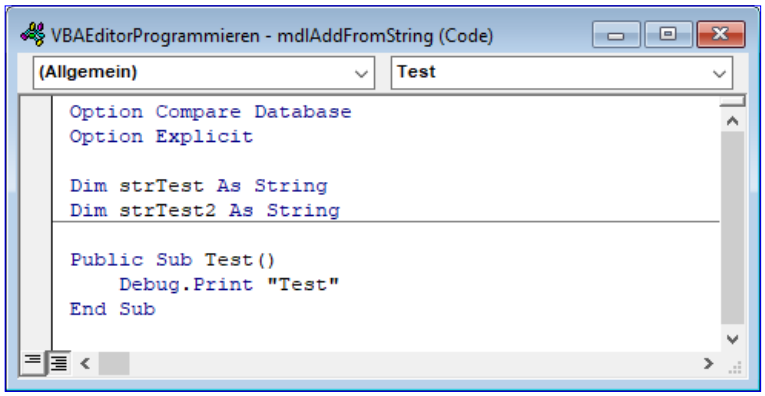


Bild 3: Per **AddFromString** nachträglich hinzugefügter Code

Das macht Sinn, denn wenn der einzufügende Code auch im oberen Bereich Deklarationszeilen und im unteren Routinen enthält, dann gibt es weiterhin eine Trennung zwischen den Deklarationen im oberen Bereich und der Programmlogik im unteren Bereich. Die folgende Prozedur fügt eine Deklarationszeile zum bestehenden Code aus dem vorherigen Beispiel hinzu:

```
Public Sub CodeModuleAddMoreFromString()
    Dim objVBProject As VBProject
    Dim objCodeModule As CodeModule
    Dim strCode As String
    Set objVBProject = VBE.ActiveVBProject
    Set objCodeModule = objVBProject.VBComponents(_
                            "mdlAddFromString").CodeModule
    strCode = "Dim strTest2 As String"
    objCodeModule.AddFromString strCode
End Sub
```

Dieser wird dann im Modul wie in Bild 3 eingefügt.

Das CodePane-Objekt eines Moduls referenzieren

Im Beitrag **Zugriff auf den VBA-Editor mit der VBE-Klasse** (**www.access-im-unternehmen.de/1350**) haben wir den Unterschied und die Zusammenhänge zwischen **VBComponent**, **Window** und **CodePane** erläutert. Das **CodePane** ist das Element, in dem das **CodeModule**-Objekt im **Window**-Objekt angezeigt wird.

Deshalb können wir vom **CodeModule**-Element über die Eigenschaft **CodePane** auch auf das betroffene **CodePane**-Element zugreifen. Das **CodePane**-Element referenzieren wir dabei wie folgt:

```
Public Sub CodePaneReferenzieren()
    Dim objVBProject As VBProject
    Dim objCodeModule As CodeModule
    Dim objCodePane As CodePane
    Set objVBProject = VBE.ActiveVBProject
    Set objCodeModule = objVBProject.VBComponents(_
                                "mdlCodeModule").CodeModule
    Set objCodePane = objCodeModule.CodePane
    '... Dinge mit dem CodePane erledigen
End Sub
```

Welche Möglichkeiten die **CodePane**-Klasse bietet, lesen Sie im Beitrag **Auf VBA-Code zugreifen per CodePane** (**www.access-im-unternehmen.de/1354**).

Schneller Zugriff auf das CodeModule per CodePane

Das **CodePane**-Objekt hat eine Eigenart, die wir uns in den folgenden Beispielen zunutze machen wollen: Sie können das aktive **CodePane**-Objekt, also den Container im aktuell aktiven VBA-Fenster mit dem **CodeModule**-Objekt, auch direkt mit der Eigenschaft **ActiveCodePane** der übergeordneten **VBE**-Klasse referenzieren.

Zeilen zählen im Modul

Es gibt einige Eigenschaften, mit denen Sie verschiedene Zeilenanzahlen ermitteln können.

Diese schauen wir uns in den nächsten Abschnitten an.

Anzahl aller Zeilen im Modul

Am einfachsten ist das Zählen aller Zeilen. Dies erledigen wir mit der Eigenschaft **CountOfLines**. Diese liefert die Anzahl der Zeilen von der ersten bis zur letzten Zeile des Moduls.

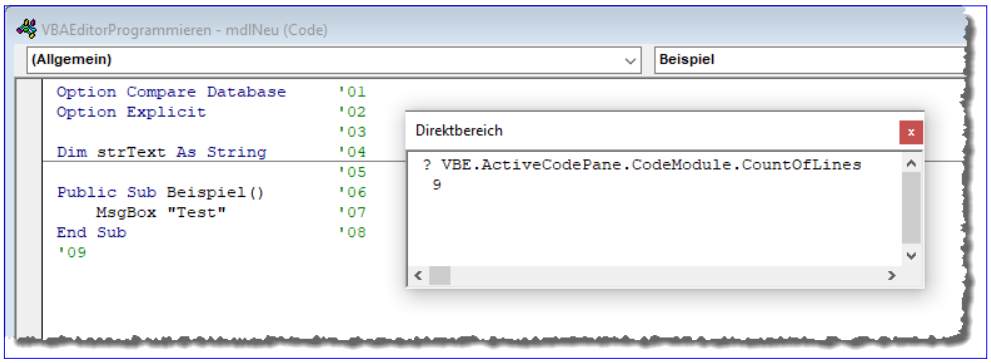


Bild 4: Abfragen von **CodeModule**-Eigenschaften per Direktbereich

Wir gehen davon aus, dass das zu untersuchende Modul geöffnet ist und dass das entsprechende VBA-Fenster den Fokus hat.

Dann können wir nämlich ganz einfach wie folgt über den Direktbereich auf zu die untersuchenden Eigenschaften zugreifen – hier auf die Anzahl der Codezeilen:

```
? VBE.ActiveCodePane.CodeModule.CountOfLines
 9
```

Die Eigenschaft **CountOfLines** liefert die Anzahl der Zeilen, wobei auch solche Zeilen ohne Inhalt mitgezählt werden. In Bild 4 ist die Zeile mit dem Kommentar **'09** die letzte Zeile. Wenn wir hinter **'09** noch einmal die Eingabetaste betätigen und somit einen Zeilenumbruch hinzufügen, liefert **CountOfLines** folglich den Wert **10**.

Beispielmodul für die folgenden Beispiele

Damit Sie Beispielmaterial haben, anhand dessen Sie die folgenden Beispiele nachvollziehen können, haben wir das Modul **mdlBeispielcode** aus Bild 5 zum Projekt hinzugefügt. Mit der **CountOfLines**-Eigenschaft erhalten Sie für dieses Modul den Wert **28**.

Anzahl der Deklarationszeilen

Die Eigenschaft **CountOfDeclarationLines** liefert die Anzahl der Zeilen bis zur letzten Deklarationsanweisung im Modul. Da Sie keine Deklarationszeilen hinter der ersten Routine mehr verwenden dürfen, können Sie beide Bereiche somit sehr gut voneinander unterscheiden.

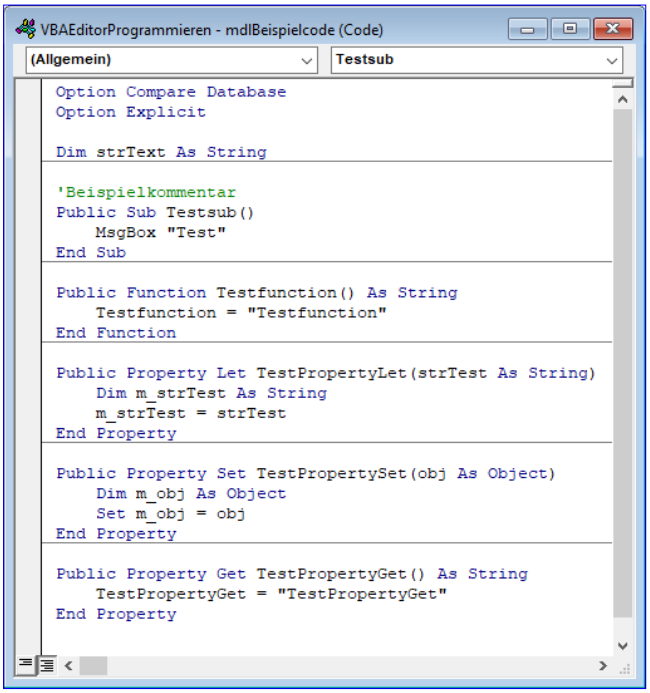


Bild 5: Beispielmodul

Im Gegensatz zu **CountOfLines** kümmert sich **CountOfDeclarationLines** situationsbedingt nicht um nachfolgende Leerzeilen:

- Wenn nach der letzten Deklarationszeile mindestens eine **Sub**-, **Function**- oder **Property**-Prozedur folgt, dann gibt **CountOfDeclarationLines** die Anzahl der Zeilen bis zur letzten Deklarationszeile aus.

- Wenn nach der letzten Deklarationszeile keine **Sub**-, **Function**- oder **Property**-Prozedur folgt, wenn das

Auf VBA-Code zugreifen mit der CodePane-Klasse

In den vorherigen Beiträgen dieser Beitragsreihe haben wir uns bereits angesehen, wie Sie auf die Module im VBA-Editor zugreifen, neue Module erstellen und den enthaltenen Code bearbeiten. Es fehlt allerdings noch eine wichtige Schnittstelle zwischen Benutzer und der automatisierten Bearbeitung von VBA-Code: Die Klasse CodePane, die unter anderem die Möglichkeit bietet, vom Benutzer gesetzte Markierungen im Code auszulesen und solche zu setzen. Letzteres können Sie beispielsweise nutzen, um per VBA gesuchte Stellen im Code zu markieren, damit der Benutzer diese erkennen kann. Dieser Beitrag stellt die CodePane-Klasse und ihre Möglichkeiten vor.

Vorbereitung

Um die Elemente der Klasse **VBE** nutzen zu können, benötigen Sie einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library**, den Sie im **Verweise**-Dialog des VBA-Editors hinzufügen können (Menüeintrag **Extras|Verweise**).

Vorbereitende Beiträge

Wenn Sie erfahren wollen, wie Sie überhaupt bis zu der hier beschriebenen Klasse gelangen, helfen die folgenden Beiträge weiter:

- **VBA-Projekt per VBA referenzieren** (**www.access-im-unternehmen.de/1337**)
- **Zugriff auf den VBA-Editor mit der VBE-Klasse** (**www.access-im-unternehmen.de/1350**)
- **Zugriff auf VBA-Projekte per VBProject** (**www.access-im-unternehmen.de/1351**)
- **Module und Co. im Griff mit VBComponent** (**www.access-im-unternehmen.de/1352**)
- **VBA-Code manipulieren mit der CodeModule-Klasse** (**www.access-im-unternehmen.de/1353**)

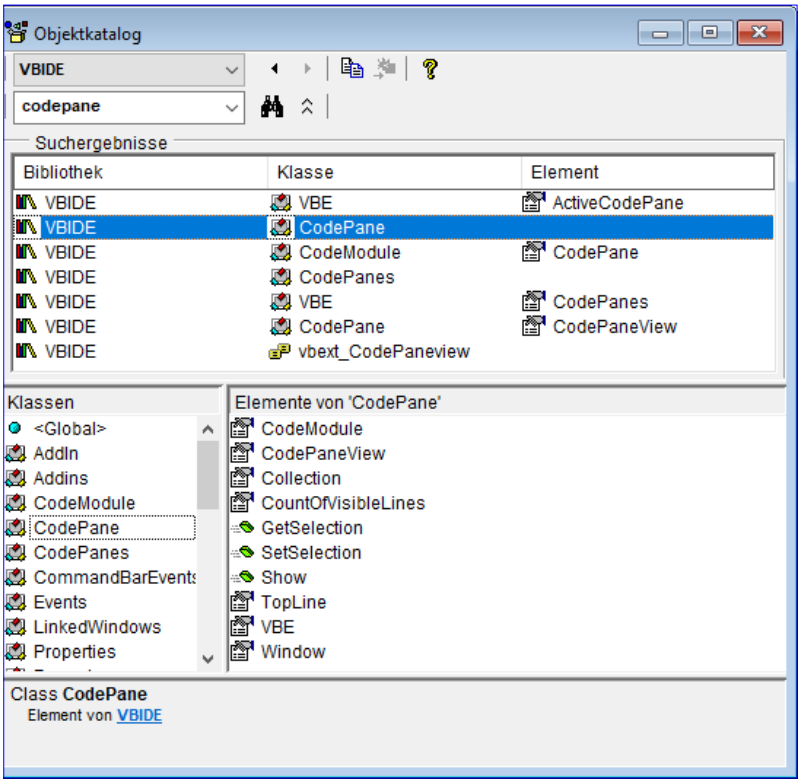


Bild 1: Die **CodePane**-Klasse im Objektkatalog

Elemente der CodePane-Klasse

Die **CodePane**-Klasse und ihre Eigenschaften, Methoden und Auflistungen können Sie im Objektkatalog (zu öffnen mit der Taste **F2**) im Überblick ansehen, wenn Sie dort nach **CodePane** suchen (siehe Bild 1). Die **CodePane**-Klasse hat folgende Eigenschaften, Methoden und Auflistungen:

- **CodeModule**: Verweis auf das **CodeModule**-Objekt, das im **CodePane**-Objekt enthalten ist
- **CodePaneView**: Gibt die aktuelle Darstellung im CodePane wieder – entweder vollständiges Modul (**1**) oder nur einzelne Prozeduren (**0**).
- **Collection**: Erlaubt den Zugriff auf alle geöffneten **CodePane**-Objekte.
- **CountOfVisibleLines**: Liefert die Anzahl der sichtbaren Zeilen im Fenster.
- **GetSelection**: Liefert den aktuell markierten Bereich mit den vier Rückgabeparametern.
- **SetSelection**: Setzt eine Markierung anhand von vier Parametern.
- **Show**: Versieht das **CodePane**-Objekt, für das diese Methode aufgerufen wurde, mit dem Fokus.
- **TopLine**: Gibt die Zeilennummer der oben im Fenster angezeigten Zeile aus und erlaubt auch das Einstellen dieser Zeile.
- **VBE**: Verweis auf die **VBE**-Klasse
- **Window**: Verweis auf das übergeordnete **Window**-Objekt

Auf das CodePane-Objekt zugreifen

Um auf ein **CodePane**-Objekt zuzugreifen, gibt es mehrere Möglichkeiten. Der Zugriff erfolgt entweder von der Auflistung **CodePanes** der **VBE**-Klasse, vom untergeordneten **CodeModule**-Element oder über die Eigenschaft **ActiveCodePane** der **VBE**-Klasse.

Wenn Sie den Namen eines Moduls kennen und auf sein **CodePane**-Objekt zugreifen wollen, nutzen Sie den Weg über **CodeModule**. Dazu referenzieren Sie zuerst das **CodeModule**-Objekt, das sie über die gleichnamige Eigenschaft des entsprechenden Elements der **VBComponents**-Auflistung des aktuellen VB-Projekts erhalten. Dieses bietet über die **CodePane**-Eigenschaft Zugriff auf das **CodePane**-Objekt.

Damit wir sehen, ob wir das richtige **CodePane**-Objekt referenzieren, zeigen wir sein Fenster mit der **Show**-Methode an:

```
Public Sub CodePanePerCodeModule()
    Dim objCodeModule As CodeModule
    Dim objCodePane As CodePane
    Set objCodeModule = VBE.ActiveVBProject.
                VBComponents("mdlBeispielcode").CodeModule
    Set objCodePane = objCodeModule.CodePane
    objCodePane.Show
End Sub
```

Damit können Sie auf alle **CodePane**-Elemente zugreifen, auch die von aktuell noch geschlossenen Modulen. Nur die aktuell in **Window**-Elementen angezeigten **CodePane**-Elemente können Sie über die Auflistung **CodePanes** der **VBE**-Klasse ansprechen.

Die folgende Prozedur durchläuft in einer **For Each**-Schleife alle Elemente der **CodePanes**-Auflistung und referenziert das jeweils aktuelle Element mit der Variablen **objCodePane**:

```
Public Sub CodePanePerWindow()
    Dim objCodePane As CodePane
    For Each objCodePane In VBE.CodePanes
        Debug.Print objCodePane.CodeModule.Parent.Name
    Next objCodePane
End Sub
```

Da das **CodePane**-Objekt selbst keine **Name**-Eigenschaft enthält, greifen wir über **objCodePane.CodeModule.Parent** auf das **VBComponent**-Objekt dieses **CodePane**-Objekts zu und geben seinen Namen aus. Das liefert nur die Namen der Module, die aktuell geöffnet sind.

Setup für Access-Applikationen, Restarbeiten

Autor: Christoph Jüngling, https://www.juengling-edv.de

In diesem Teil widmen wir uns einigen Restarbeiten für das Erstellen eines Setups für Access-Applikationen. Diese Arbeiten sind zwar keineswegs unbedingt notwendig, runden aber unser Setup ab und sorgen daher beim Anwender oder Administrator für ein »gutes Gefühl«. Wir wollen zunächst sicherstellen, dass unsere Applikation nicht läuft, wenn wir sie updaten wollen. Dann unterscheiden wir bei der Installation zwischen Beta- und finaler Version. Und letztlich wollen wir unser Setup dann noch digital signieren.

Läuft die Applikation?

Wie können wir sicherstellen, dass die Access-Applikation nicht läuft, wenn wir sie updaten wollen?

Prinzip eines "Mutex"

Sicher haben Sie schon bei Setups bemerkt, dass eine Meldung wie diese kam:

```
Das Setup hat festgestellt, dass die Applikation Xyz noch
läuft. Bitte beenden Sie diese und starten Sie dann das
Setup erneut.
```

Dem Anwender ist das natürlich egal, aber der Entwickler fragt sich unweigerlich: »Woher weiß das Setup das eigentlich?« Die Frage ist berechtigt, denn einfach mal »nachschauen« ist halt für eine Software nicht ganz so einfach wie für uns.

Das Geheimnis heißt »Mutex«. Das ist die Kurzform von »mutual exclusion«, auf Deutsch »gegenseitiger Ausschluss«, eine beeindruckend gute Bezeichnung für diesen Mechanismus.

Denn die Setupdurchführung und das laufende Programm schließen sich gegenseitig aus. Während die ACCDB aktiv ist, sollte man sie nicht per Setup austauschen.

Um dies zu nutzen, benötigen wir also sowohl in der Applikation als auch im Setup jeweils einen Mechanismus, der den Mutex setzen, löschen und überprüfen kann. Unsere Access-Applikation erzeugt den Mutex beim Start und löscht ihn kurz vor dem Ende.

Das Setup wiederum überprüft nur, ob es ihn gibt. Falls ja, kommt eine Meldung wie oben gezeigt, falls nein, läuft das Setup einfach weiter.

Das lässt sich in InnoSetup sogar soweit automatisieren, dass ein von der Accdb angestoßenes Update automatisch abläuft.

Mutex-Klasse (VBA)

Mit Hilfe eines kurzen Codesegments schaffen wir es, den Mutex beim Starten unserer Access-Applikation zu erzeugen. Das übernimmt in meinen Projekten immer eine Klasse, die ich eigens dafür geschrieben habe.

Es müsste natürlich keine Klasse sein, eine Sub-Prozedur täte es auch, aber mit der Klasse haben wir den Vorteil, dass wir mit einer Einheit beide Aktionen erledigen können: Erzeugen und Löschen des Mutex wird innerhalb der Klasse durchgeführt – und zwar über den Konstruktor **Initialize** und den Destruktor **Terminate** der Klasse.

Zur Integration in die Applikation müssen neben der Klasse selbst nur noch eine Deklaration und zwei Zeilen Code eingefügt werden.

Schauen wir uns zunächst die Mutex-Klasse an. Besonders aufwändig ist sie nicht (siehe Listing 1).

Die Erzeugung passiert in **Class_Initialize**, die Zerstörung in **Class_Terminate**. Der einzige Bezug in unsere

```
''
' Manage an Application Mutex for the current application
' @remarks  Mutex name = APP_MUTEX
' @author   Christoph Juengling <christoph@juengling-edv.de>
' @link   https://gitlab.com/juengling/vb-and-vba-code-library
Option Explicit
#If VBA7 Then
Private Declare PtrSafe Function CreateMutex Lib "kernel32" Alias "CreateMutexA" (lpMutexAttributes As Any, _
     ByVal bInitialOwner As Long, ByVal lpName As String) As LongPtr
Private Declare PtrSafe Function CloseHandle Lib "kernel32" (ByVal hObject As LongPtr) As Long
Private Declare PtrSafe Function ReleaseMutex Lib "kernel32" (ByVal hMutex As LongPtr) As Long
#Else
Private Declare Function CreateMutex Lib "kernel32" Alias "CreateMutexA" (lpMutexAttributes As Any, _
    ByVal bInitialOwner As Long, ByVal lpName As String) As Long
Private Declare Function CloseHandle Lib "kernel32" (ByVal hObject As Long) As Long
Private Declare Function ReleaseMutex Lib "kernel32" (ByVal hMutex As Long) As Long
#End If
Private m_lMutexHandle As Long
Private Sub Class_Initialize()
    m_lMutexHandle = 0
    CreateMyMutex
End Sub

Private Sub Class_Terminate()
    ReleaseMyMutex
End Sub

' CreateMutex the Mutex
Public Sub CreateMyMutex()
    m_lMutexHandle = CreateMutex(ByVal CLng(0), CLng(1), APP_MUTEX)
End Sub

''
' ReleaseMutex the Mutex and close handle
'
Public Sub ReleaseMyMutex()
    If m_lMutexHandle > 0 Then
        ReleaseMutex m_lMutexHandle
        CloseHandle m_lMutexHandle
        m_lMutexHandle = 0
    End If
End Sub
```

Listing 1: Klasse zum Nutzen eines Mutex

Applikation ist hier die Nutzung der globalen Konstanten **APP_MUTEX**. Diese deklariert den Namen des Mutex:

```
Public Const APP_MUTEX = "Mein Name ist Hase"
```

Die Bezeichnung sollte sinnvollerweise dem Namen der Applikation entsprechen, so dass man natürlich auch **APP_NAME** verwenden kann.

Wichtig ist die Einzigartigkeit im Hinblick auf andere Applikationen! Eine solche Deklaration muss natürlich namensgleich später auch im InnoSetup-Skript enthalten sein, damit Setup und Applikation über dieselbe Bezeichnung verfügen.

Nun müssen wir nur noch dafür sorgen, dass

- eine Modul-Variable für die Instanz der Klasse existiert,
- diese beim Start instanziiert wird und
- vor dem Beenden der Applikation zerstört wird.

Das ist ebenfalls trivial. In einem Modul, in dem vielleicht noch weitere globale Deklarationen stehen, fügen wir diese Zeile ein:

```
Public mutex As clsMutex
```

In dem Code für die Initialisierung:

```
Set mutex = New clsMutex
```

Und für das Beenden:

```
Set mutex = Nothing
```

Das war doch einfach, oder? Streng genommen ist die letzte Aktion nicht nötig, da der Mutex mit dem Beenden der Applikation automatisch gelöscht wird.

Den Mutex in InnoSetup eintragen

In InnoSetup ist die Sache sogar noch einfacher, denn der ganze Mechanismus zum Überprüfen und Reagieren ist dort bereits enthalten, Code müssen wir dafür nicht schreiben.

Wir müssen nur dafür sorgen, dass InnoSetup den Namen des Mutex erfährt. Wie schon erwähnt, muss diese Deklaration natürlich identisch mit der Konstanten aus unserer Applikation sein, und das betrifft auch die Groß-/Kleinschreibung!

Im Deklarationsbereich zu Beginn des Skriptes schreiben wir also:

```
#define MyAppMutex "Mein Name ist Hase"
```

Und in der Gruppe **[Setup]**:

```
AppMutex={#MyAppMutex}
```

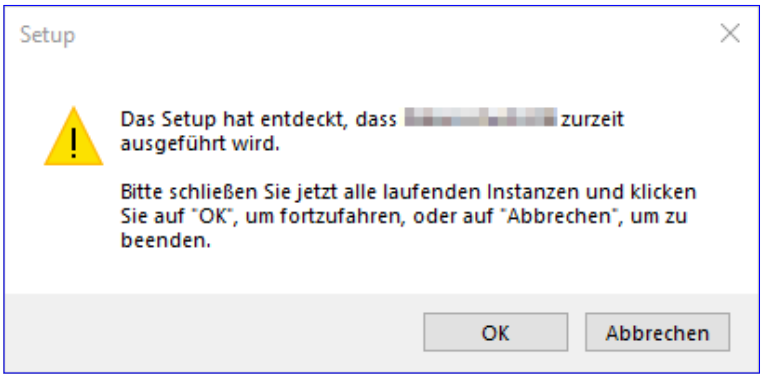


Bild 1: Mutex-Meldung

Mehr ist nicht nötig. Und wie funktioniert das nun?

Zum einen genau wie oben beschrieben. Wenn das Setup bei der Ausführung entdeckt, dass der Mutex bereits existiert, zeigt es die Meldung aus Bild 1 an.

Nun kann der Anwender entsprechend darauf reagieren. Besonders elegant arbeitet InnoSe-

tup jedoch, wenn es für die Installation mit dem Kommandozeilen-Argument **silent** aufgerufen wurde.

Dann nämlich verzichtet es auf alle Rückfragen mit Benutzerinteraktion. Für den Mutex-Fall bedeutet das, dass das Setup solange abwartet, bis der Mutex verschwunden ist. Dann wird das Setup mit Standardeinstellungen fortgeführt.

Dieser Mechanismus hat den besonderen Reiz, dass die Applikation selbst per **Shell**-Execute die Ausführung des Setups anstarten und sich dann in aller Ruhe beenden kann. Erst wenn der Mutex gelöscht ist, läuft das Setup durch.

Beta oder Final?

Nicht immer ist ein Programm sofort fertig, auch wenn der Entwickler noch so überzeugt von seiner Arbeit ist. Daher kann es sinnvoll sein, ausgewählte User in einen Beta-Test einzubeziehen.

Dabei sollte jedoch immer auf eine strenge Trennung nicht nur der Programminstallation, sondern auch bezüglich der Daten geachtet werden. Eine Möglichkeit dabei ist eine Kombination aus dem Setup-Skript und dem Code, der für die Tabelleneinbindung sorgt. Worauf müssen wir achten, und wie machen wir uns die Arbeit möglichst einfach?

Beginnen wir mit der Frage, bezüglich welcher Einstellungen sich Beta- und Final-Version unterscheiden:

- Installationspfad
- Beta-Hinweis oder Lizenzvereinbarung
- Pfad zum Backend

Installationspfad

Nehmen wir (wie schon im ersten Teil dargelegt) an, dass wir unser »normales« Programm unter **C:\Users\USERNAME\AppData\Local\Programs** installieren wollen. Dort wird sicherlich für jedes Programm ein eigenes Unterverzeichnis verwendet werden.

Es bietet sich also an, dies für unser eigenes Programm ebenfalls zu tun, und zwar getrennt für Beta- und Final-Version.

Dazu müssen wir also die Setup-Einstellung **DefaultDirName** angemessen verändern. Bisher steht dort **{userpf}\{#MyAppName}**. Es spricht also nichts dagegen, zum Beispiel **{userpf}\{#MyAppName}**-Beta zu verwenden.

Doch mir widerstrebt es, jedesmal mitten im Skript eine Änderung vorzunehmen. Da kann man sich leicht vertun, und es wird auch nicht die einzige Änderung bleiben. Eine Lösung für diesen Fall sind wieder einmal die Konstanten, die wir ja bereits kennengelernt haben. Ich füge also zunächst eine weitere hinzu:

```
#define MyAppStatus "Beta"
```

oder

```
#define MyAppStatus "Final"
```

Damit haben wir wieder eine Einstellung, die wir nur am Anfang des Skriptes verändern müssen, wodurch hoffentlich alle anderen Settings entsprechend angepasst werden.

Damit das funktioniert, nutzen wir eine weitere Möglichkeit von InnoSetup, die uns als Entwickler natürlich vertraut ist: Die **If-Then-Else**-Anweisung. Sie funktioniert im Prinzip wie von VBA her bekannt, nur sieht die Syntax ein wenig anders aus.

Im folgenden Beispiel erweitere ich bei einer Betaversion den Standard-Installationspfad und die Startmenü-Gruppe zum Beispiel durch den Zusatz **-Beta**:

Export von Daten in das DATEV-Format

Um Daten Ihrer eigenen Software so zu exportieren, dass Ihr Steuerberater diese in das DATEV-System einlesen kann, benötigen Sie gar nicht mal so viel Know-how. Die wesentlichen Informationen finden wir auf der Webseite von DATEV. Dieser Beitrag zeigt, wie Sie Daten im DATEV-Format exportieren und was dabei zu beachten ist. Die so entstandene Datei kann der Steuerberater dann per Importfunktion in das DATEV Rechnungswesen einlesen.

Die grundlegenden Informationen finden Sie in der öffentlich verfügbaren Webseite unter folgender Adresse:

```
https://developer.datev.de/portal/de/dtvf
```

Dieser Beitrag beschränkt sich auf die Zusammenstellung einer Datei mit den **Header**-Daten und Daten im Format **Buchungsstapel**.

Grundlegender Aufbau einer DATEV-Datei

Die DATEV-Datei enthält immer eine Zeile mit Daten im Format **Header**. Danach folgt eine Zeile mit den Spaltenüberschriften für die Buchungsdaten und schließlich eine oder mehrere Zeilen mit den eigentlichen Buchungsdaten.

Verschiedene Formate

Das DATEV-Format besteht aus verschiedenen Formatbeschreibungen. Verschiedene Formate deshalb, weil es unterschiedliche Informationen gibt, die Sie damit beschreiben können. Wir wollen uns auf die wesentlichen Elemente beschränken – die übrigen können Sie mit dem Know-how aus diesem Beitrag und der Dokumentation dann leicht selbst realisieren.

Es gibt die folgenden Formatsätze:

- **Header**: Erste Zeile der CSV-Datei mit den Informationen zur Verarbeitung der Datei
- **Buchungsstapel**: Enthält die eigentlichen Buchungen, in der Regel für einen bestimmten Zeitraum je Datei, zum Beispiel monatlich.
- Weitere Formatsätze, die dieser Beitrag nicht behandelt: **Wiederkehrende Buchungen**, **Debitoren/Kreditoren**, **Sachkontenbeschriftungen**, **Zahlungsbedingungen** und **Diverse Adressen**.

Zusammenstellen der Header-Zeile

Die Beschreibung des Satzaufbaus für die Header-Zeile finden Sie hier:

```
https://developer.datev.de/portal/de/dtvf/formate/header
```

Die Tabelle auf dieser Seite enthält die Überschriften und die Beschreibung der möglichen Werte als regulärer Ausdruck.

Eine Header-Zeile sieht beispielsweise wie folgt aus:

```
"EXTF";700;21;"Buchungsstapel";12;20211007000000000;4;"Buchungen_Konto1";1;"EUR"
```

Dabei entsprechen die einzelnen Elemente diesen Feldern:

- **Kennzeichen**: **EXTF** oder **DTVF** (hier **EXTF** für Export aus einer 3rd-Party-Anwendung)
- **Versionsnummer**: aktuell **700** (dient der Sicherstellung von Abwärtskompatibilität)
- **Formatkategorie**: Gibt an, in welchem Format die folgenden Daten sind (im Falle des Buchungsstapels, mit dem wir uns hier beschäftigen, nutzen wir den Wert **21**).

- **Formatname**: Name des Formats, hier **Buchungsstapel**

- **Formatversion**: Auch hier kommt ein Zahlenwert für Buchungsstapel zum Einsatz, in diesem Fall **12**.

- **Erzeugt am**: Datum, an dem die Datei erzeugt wurde, im Format **YYYYMMDDHHMMSSFFF**.

Es gibt noch einige weitere Felder, die wir weiter unten berücksichtigen.

Tabelle für Headerdaten

Wie speichern wir die Headerdaten am einfachsten? Wenn Sie aus ihrer selbstprogrammierten Buchhaltungsdatenbank die Buchungsdaten für den Import in die DATEV-Software Ihres Steuerberaters exportieren wollen, werden Sie das regelmäßig erledigen – beispielsweise monatlich oder einmal je Quartal.

Die Felder des Headers enthalten Daten, die wir über eine Tabelle namens **tblHeader** erfassen (Entwurf siehe Bild 1). Diese enthält nicht genau die Felder, die in der Header-Datei gespeichert werden sollen. Die Felder **ID** und **Referenzbezeichnung** sind das Primärschüsselfeld sowie ein Feld für eine interne Bezeichnung, zum Beispiel **Export 12/2021**. Die übrigen Felder nehmen meist direkt die benötigten Daten auf, einige jedoch sind Fremdschlüsselfelder, welche zur Auswahl von Daten aus Lookup-Tabellen dienen.

tblHeader

Feldname	Felddatentyp	Beschreibung (optional)
ID	AutoWert	ID des Headersatzes
Dateiname	Kurzer Text	Name der zu erstellenden Datei ohne Präxix und Suffix)
Kennzeichen	Kurzer Text	Kennzeichen, entweder EXTF (Export aus 3rd Party App) oder DTVF (Export aus Datev App)
Versionsnummer	Kurzer Text	Versionsnummer des Headers
FormatkategorieID	Zahl	siehe Tabelle tblFormatkategorien
Formatversion	Zahl	Formatversion
ErzeugtAm	Datum/Uhrzeit	Erzeugungsdatum
7_Reserviert	Kurzer Text	Leerfeld
8_Reserviert	Kurzer Text	Leerfeld
9_Reserviert	Kurzer Text	Leerfeld
10_Reserviert	Kurzer Text	Leerfeld
Beraternummer	Kurzer Text	Beraternummer
Mandantennummer	Kurzer Text	Mandantennummer
Wirtschaftsjahresbeginn	Datum/Uhrzeit	Beginn des Wirtschaftsjahres
Sachkontenlaenge	Zahl	Nummernlänge der Sachkonten
DatumVon	Datum/Uhrzeit	Startdatum
DatumBis	Datum/Uhrzeit	Enddatum
Bezeichnung	Kurzer Text	Bezeichnung des Stapels
Diktatkuerzel	Kurzer Text	Kürzel des Bearbeiters
Buchungstyp	Zahl	siehe Tabelle tblBuchungstypen
Rechnungslegungszweck	Zahl	siehe Tabelle tblRechnungslegungszwecke
Festschreibung	Zahl	siehe Tabelle tblFestschreibungen
Waehrungskennzeichen	Kurzer Text	ISO-Code der verwendeten Währung
23_Reserviert	Kurzer Text	Leerfeld
Derivatskennzeichen	Kurzer Text	Leerfeld
25_Reserviert	Kurzer Text	Leerfeld
26_Reserviert	Kurzer Text	Leerfeld
Sachkontenrahmen	Kurzer Text	Sachkontenrahmen, der verwendet wurde
IDDerBranchenLoesung	Kurzer Text	Falls eine spezielle Branchenlösung verwendet wurde
29_Reserviert	Kurzer Text	Leerfeld
30_Reserviert	Kurzer Text	Leerfeld
Anwendungsinformationen	Kurzer Text	Verarbeitungskennzeichen der abgebenden Anwendung

Bild 1: Entwurf der Tabelle **tblHeader**

Diese Lookup-Tabellen stellen wir in Bild 2 übersichtlich dar.

Die Beziehungen zwischen der Tabelle **tblHeader** und den übrigen Tabellen haben wir jeweils als Nachschlagefeld definiert.

Formular zur Eingabe der Headerdaten

Damit der Benutzer die Headerdaten einfach eingeben kann, haben wir dazu ein eigenes Formular bereitgestellt. Dieses heißt **frmHeader** und verwendet die Tabelle **tblHeader** als Datensatzquelle. Wir haben alle Felder der Tabelle aus der Feldliste in den Formularentwurf gezogen, die in der Dokumentation nicht als Leerfeld gekennzeichnet sind (siehe Bild 3).

Hier sehen Sie bereits, dass die Felder, für die wir Lookup-Tabellen angelegt haben, auch im Formularentwurf als Nachschlagefelder angelegt werden.

Nach dem Wechsel in die Formularansicht können Sie die Daten direkt in das Formular eingeben beziehungsweise mit den Nachschlagefeldern auswählen (siehe Bild 4).

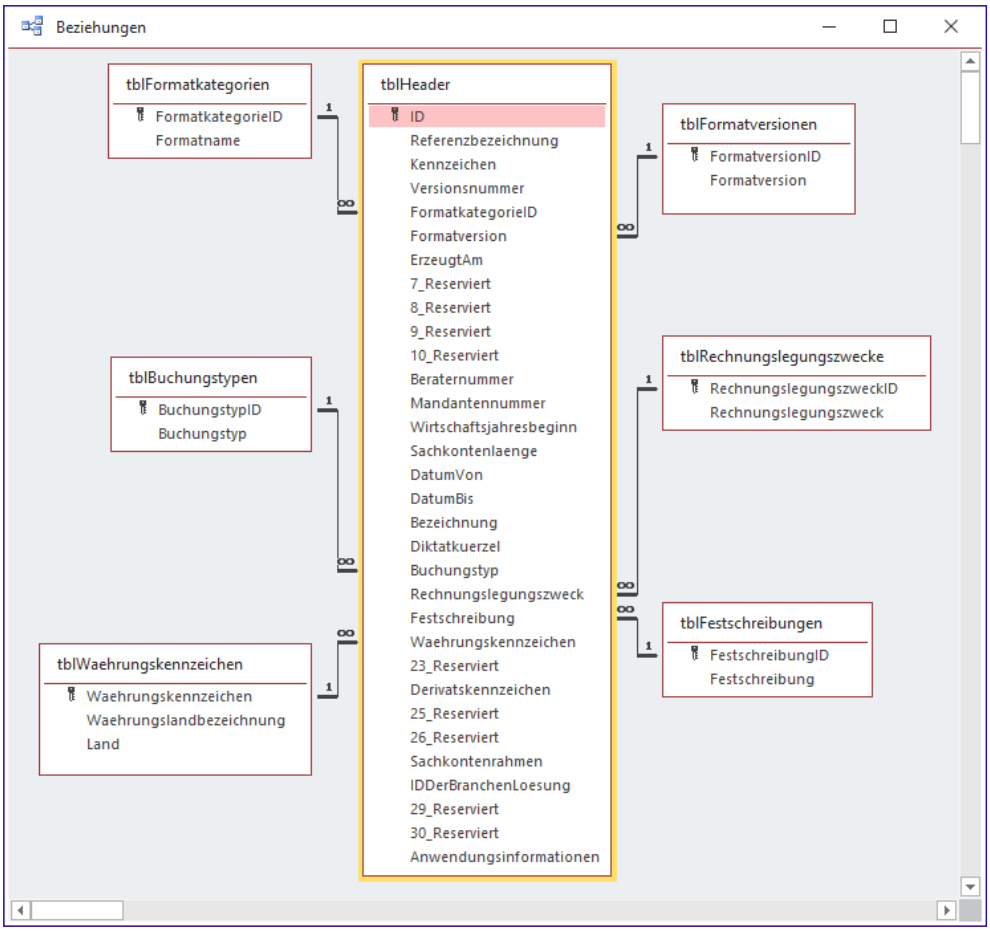


Bild 2: Übersicht des Datenmodells

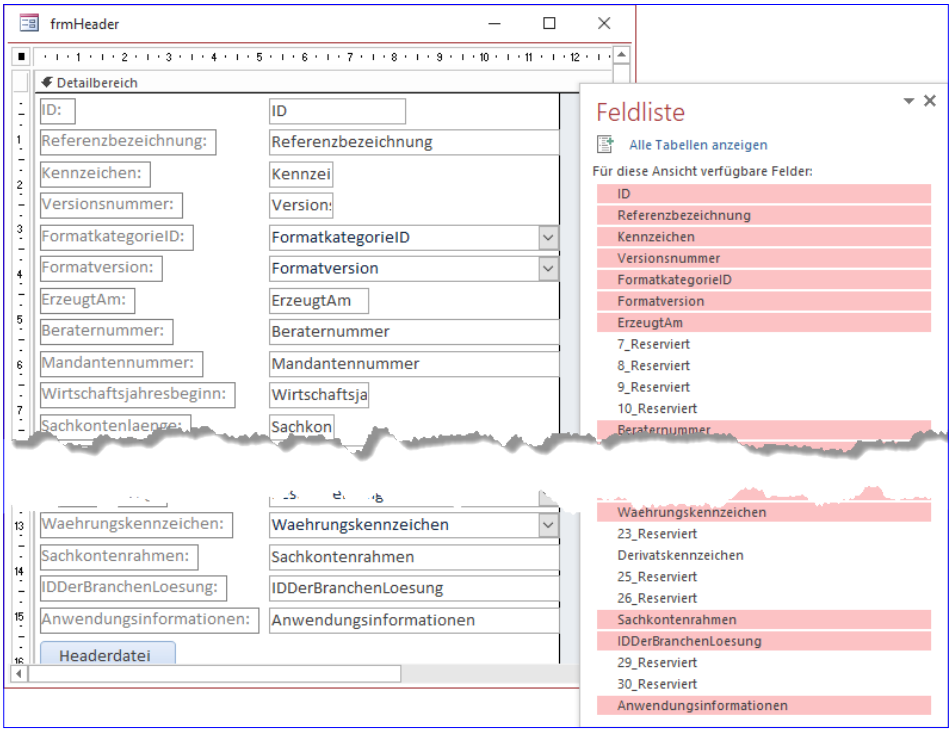


Bild 3: Entwurf des Formulars zum Eingeben der Headerdaten

Nachdem wir diese Eingabemöglichkeit geschaffen haben, wollen wir daraus noch die Headerdatei exportieren. Dazu sehen wir eine Schaltfläche namens **cmdHeaderdateiErzeugen** vor, die wir unten im Formular platzieren.

Diese Schaltfläche ruft die Prozedur aus Listing 1 auf.

Die Prozedur deklariert zwei Variablen, von denen eine die zu speichernde Headerzeile aufnimmt und die andere den Dateinamen.

Den Dateinamen setzt die Prozedur aus dem Pfad der aktuellen Datenbankdatei, dem Präfix **DTVF_**, dem Inhalt des Feldes **Dateiname** und der Dateiendung **.csv** zusammen.

Bevor wir richtig loslegen, löscht die Prozedur eine eventuell bereits vorhandene Datei mit dem Namen aus **strDateiname**. Dabei deaktivieren wir die eingebaute Fehlerbehandlung, da die **Kill**-Anweisung beim Fehlen der angegebenen Datei einen Fehler auslöst.

Die folgenden Zeilen setzen den Inhalt der Headerdatei zusammen.

Dabei sind folgende Besonderheiten zu beachten (in allen anderen Fällen landet einfach der eingegebene Wert zwischen zwei Semikola):

- Das Kennzeichen wird in Anführungszeichen eingefasst.
- Den Wert des Feldes **Formatname** lesen wir aus der Lookup-Tabelle **tblFormatversionen** ein.
- Den Wert für das Feld **Erzeugt am** formatieren wir im Format **YYYYMMDDHHNNSS000**.
- Den Wert für das Feld **Wirtschaftsjahresbeginn** formatieren wir mit dem Format **YYYYMMDD**.
- Die Werte für die Felder **Datum von** und **Datum bis** formatieren wir ebenfalls mit **YYYYMMDD**.

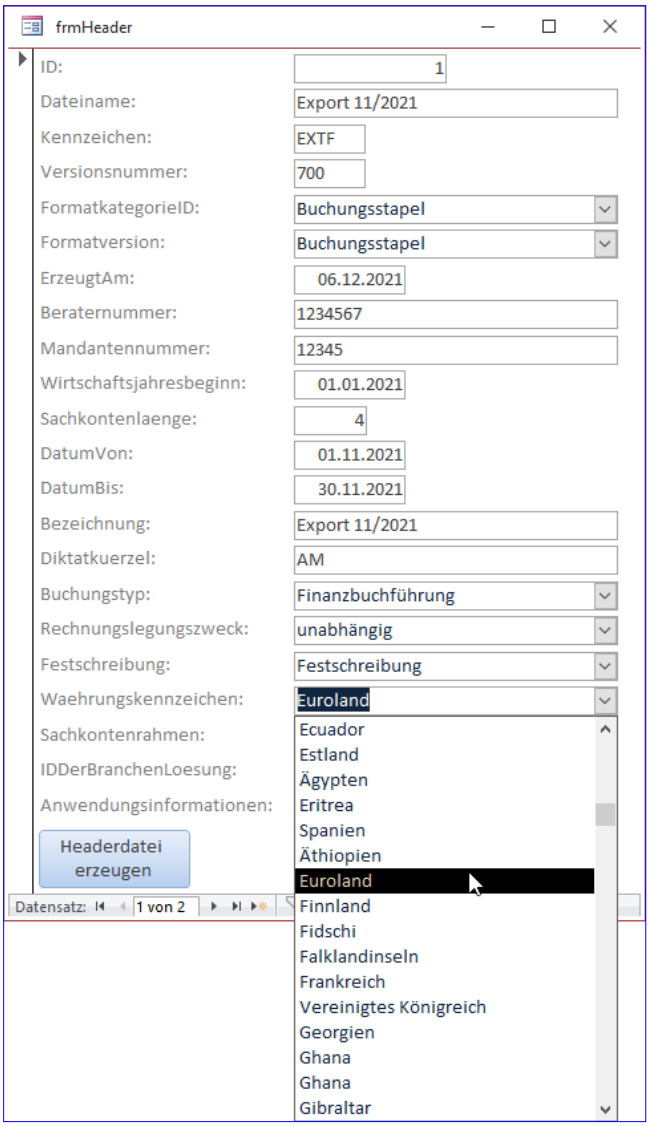


Bild 4: Eingabe der Headerdaten

Anschließend öffnen wir mit **Open** eine Datei mit dem Namen aus **strDateiname** und schreiben mit der **Print**-Anweisung den Inhalt der Variablen **strHeader** in die Datei, bevor wir diese mit der **Close**-Anweisung wieder schließen.

Das Ergebnis sieht für unser Beispiel wie folgt aus:

```
"EXTF";700;21;Buchungsstapel;12;20211206000000000;;;
;;1234567;12345;20210101;4;20211101;20211130;Export
11/2021;AM;1;0;1;EUR;;;;;03;;;;
```

Damit können wir nun einen Schritt weitergehen – und das Ergebnis mit einem speziell für diesen Zweck vorgesehenen Tool prüfen. Wie das gelingt, erfahren Sie im nächsten Schritt.

```
Private Sub cmdHeaderdateiErzeugen_Click()
    Dim strHeader As String
    Dim strDateiname As String
    strDateiname = CurrentProject.Path & "\DTVF_" & Me!Dateiname & ".csv"
    On Error Resume Next
    Kill strDateiname
    On Error GoTo 0
    strHeader = strHeader & """ & Me!Kennzeichen & """;"
    strHeader = strHeader & Me!Versionsnummer & ";"
    strHeader = strHeader & Me!FormatkategorieID & ";"
    strHeader = strHeader & DLookup("Formatversion", "tblFormatversionen", "FormatversionID = " & Me!Formatversion) & ";"
    strHeader = strHeader & Me!Formatversion & ";"
    strHeader = strHeader & Format(Me!ErzeugtAm, "YYYYMMDDHHNNSS000") & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & Me!Beraternummer & ";"
    strHeader = strHeader & Me!Mandantennummer & ";"
    strHeader = strHeader & Format(Me!Wirtschaftsjahresbeginn, "YYYYMMDD") & ";"
    strHeader = strHeader & Me!Sachkontenlaenge & ";"
    strHeader = strHeader & Format(Me!DatumVon, "YYYYMMDD") & ";"
    strHeader = strHeader & Format(Me!DatumBis, "YYYYMMDD") & ";"
    strHeader = strHeader & Me!Bezeichnung & ";"
    strHeader = strHeader & Me!Diktatkuerzel & ";"
    strHeader = strHeader & Me!Buchungstyp & ";"
    strHeader = strHeader & Me!Rechnungslegungszweck & ";"
    strHeader = strHeader & Me!Festschreibung & ";"
    strHeader = strHeader & Me!Waehrungskennzeichen & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & Me!Sachkontenrahmen & ";"
    strHeader = strHeader & Me!IDDerBranchenLoesung & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & ";"
    strHeader = strHeader & Me!Anwendungsinformationen
    Open strDateiname For Append As #1
    Print #1, strHeader
    Close #1
End Sub
```

Listing 1: Prozedur zum Erzeugen der Headerdatei

Datei im DATEV-Format prüfen

Unter dem folgenden Link finden Sie ein Tool zum Prüfen der Daten im DATEV-Format:

`https://developer.datev.de/portal/de/dtvf/tools`

Mit diesem können Sie die erstellte Datei öffnen und prüfen. Das Tool zeigt dann an, welche Elemente fehlen oder das falsche Format haben. Es ist daher sehr hilfreich, den Export initial auf Basis von Daten aus der Datenbank zu programmieren und gleich zu testen.

Um die soeben erstellte Headerdatei zu prüfen, wählen Sie im Prüfprogramm den Menübefehl **Datei|Öffnen** aus.

Im nun erscheinenden Dialog **Datev Format Datei öffnen** wählen Sie die soeben erstellte Datei aus und bestätigen die Auswahl mit der Schaltfläche **Öffnen**.

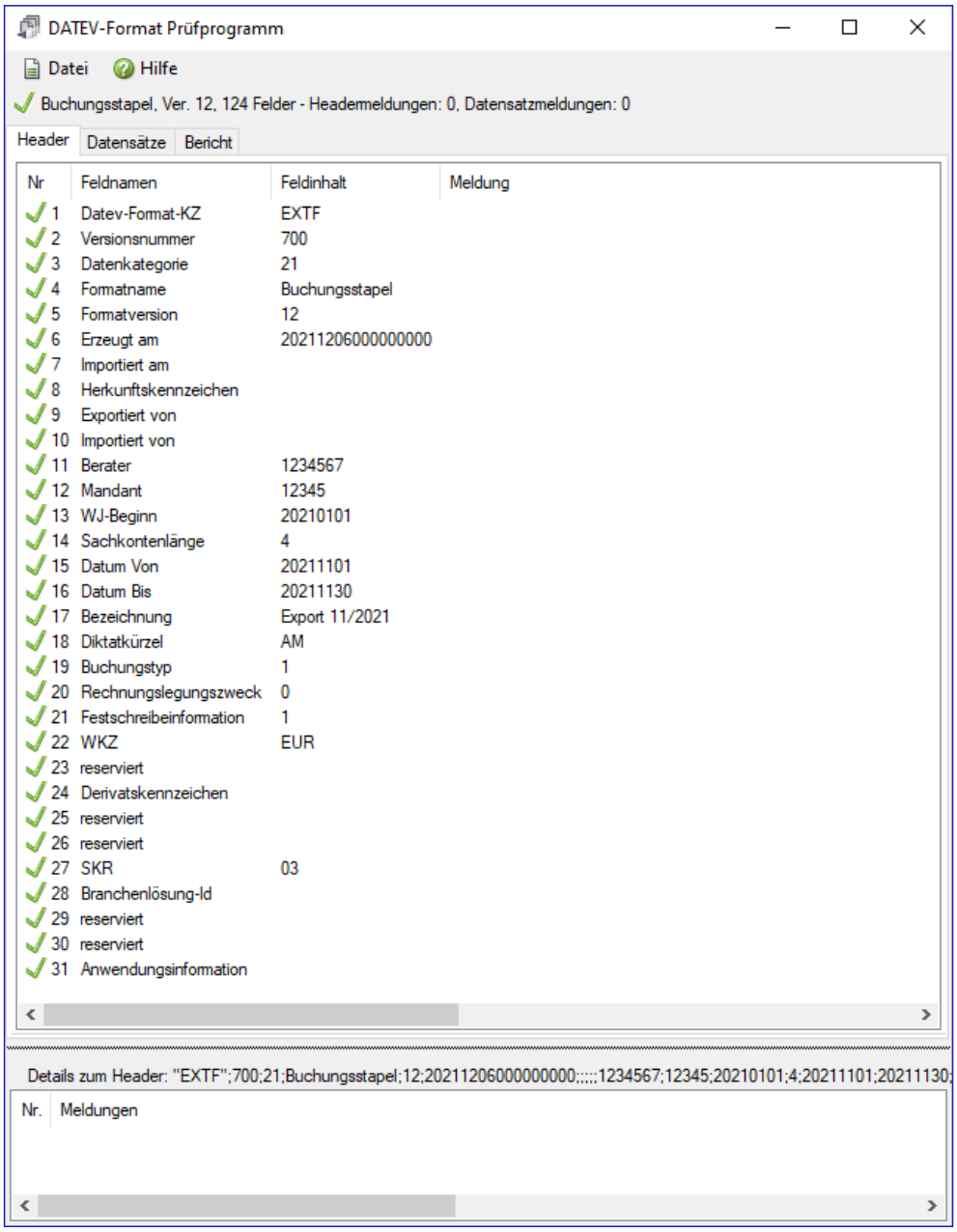


Bild 5: Prüfung des Headers

Bild 5 zeigt das Ergebnis für die Eingaben aus dem Beispiel von oben. Hier sehen Sie, dass alle Eingaben korrekt sind. Wäre dies nicht der Fall, würden Sie in der Spalte **Meldung** hilfreiche Informationen finden, mit denen Sie den Export schnell anpassen und funktionstüchtig machen könnten.

Export der Buchungsdaten programmieren

Der Export der Buchungsdaten ist grundsätzlich aufwändiger, weil eine Zeile viel mehr Informationen enthalten kann als die Headerzeile, aber wir reduzieren die auszugebenden Daten auf einige wenige Pflichtdaten.

Als Erstes legen wir wieder eine Tabelle an, welche die zu exportierenden Daten enthält. Diese nennen wir **tblBu-**

HAUFE.
Ausgabe 02/2022

ACCESS
IM UNTERNEHMEN

BEISPIELDATEN GENERIEREN

Nutzen Sie .NET-Techniken, um per Mausklick Beispieldaten für Ihre Entwicklungsprojekte zu generieren (ab Seite 55).

In diesem Heft:

E-MAIL-ANLAGEN

Fügen Sie einer Mail unter Outlook flexibel eine oder mehrere Anlagen hinzu und versenden Sie diese.

SEITE 69

DATEIAUSWAHL PER FILEDIALOG

Wählen Sie Dateien zum Öffnen oder Speichern aus oder selektieren Sie Verzeichnisse mit der FileDialog-Klasse.

SEITE 39

SQL SERVER-SICHERHEIT, TEIL 7

Realisieren Sie Sicherheitsmechanismen über die Windows-Authentifizierung.

SEITE 2

Beispieldaten mit .NET

Wenn Sie schon einmal eine Datenbankanwendung von Grund auf erstellt haben, für die es auch noch keine Vorgängeranwendung gab, stehen Sie vor einem Problem: Woher nehmen Sie die Daten, um die Tabellen, Abfragen, Formulare, Berichte und den VBA-Code Ihrer Anwendung zu testen? Natürlich können Sie immer wieder von Hand neue Kundendatensätze eingeben, die Sie sich zuvor ausgedacht haben, aber das ist bestenfalls für das Testen eines dafür vorgesehenen Eingabeformulars sinnvoll.

Wer mehr testen möchte als nur die Eingabe von Daten in ein dafür vorgesehenes Formular, benötigt dazu gegebenenfalls eine ganze Menge von Daten – und zwar solche Daten, die immer wieder schnell rekonstruiert werden können.

Für Access beziehungsweise VBA gibt es keine fertige Lösung, um Tabellen schnell mit Daten zu füllen, die dem jeweiligen Datentyp entsprechen. Aber wozu gibt es .NET? Durch die gute Erweiterbarkeit von .NET-Anwendungen gibt es zahllose Erweiterungen, darunter auch solche, mit denen sich Beispieldaten generieren lassen. Eine davon heißt Bogus und diese stellen wir im Beitrag **Beispieldaten generieren mit .NET und Bogus** ab Seite 55 vor. Die Bibliothek stellt Klassen und Funktionen bereit, mit denen Sie verschiedenartige Daten generieren können, und zwar in beliebigen Mengen. So lassen sich etwas Adressdatensätze zusammenstellen und Bestellungen, und Sie finden auch noch Möglichkeiten, um die so generierten Bestellungen den Adressen zuzuordnen.

Das funktioniert leider nicht von Access aus, sondern wir benötigen ein Tool, mit dem wir auf einfache Weise .NET-Code programmieren und laufen lassen können. Dazu nutzen wir das Tool LINQPad. Es erlaubt das einfache Schreiben und Testen von Code – und das eben auch für die zuvor beschreibene Lösung zum Erstellen von Beispielcode. Alles über LINQPad und wie Sie damit auf die Tabellen von Access-Datenbanken zugreifen können, lernen Sie im Beitrag **Datenzugriff mit .NET, LINQPad und LINQ to DB** ab Seite 49. Hier nutzen wir zusätzlich die Bibliothek LINQ to DB für den Datenzugriff.

Auch in unserer Reihe zum Thema Sicherheit und SQL Server geht es weiter. Unter dem Titel **SQL Server-Security – Teil 7: Windows-Authentifizierung** zeigt unser Autor Bernd Jungbluth ab Seite 3, wie Sie die Windows-Authentifizierung für die Steuerung des Zugriffs auf SQL Server-Datenbanken nutzen können und was es dabei zu beachten gibt.

Bei der Entwicklung von Anwendungen benötigen Sie früher oder später Dateiauswahldialoge. Diese stellt Office als Bordmittel bereit. Dennoch kommen Sie um ein wenig VBA-Programmierung nicht herum. Damit alles wie gewünscht gelingt, wenn Sie Dateien zum Öffnen oder Speichern auswählen wollen oder wenn es darum geht, einen Ordner zu selektieren, hilft Ihnen der Beitrag **Dateien und Verzeichnisse auswählen mit FileDialog** ab Seite 39 weiter.

Vielleicht möchten Sie die dort gewählten Dateien ja als Anlage zu einer neuen E-Mail hinzufügen und diese so verschicken? Dann finden Sie im Beitrag **E-Mails mit Anlagen mit Outlook versenden** ab Seite 69 noch die richtigen Programmierungen. Fügen Sie ganz einfach eine oder mehrere Dateien aus einem oder mehreren verschiedenen Verzeichnissen zur Mail hinzu und senden Sie diese per Mail an den gewünschten Empfänger!

Nun viel Spaß beim Lesen!

Ihr André Minhorst

SQL Server-Security – Teil 7: Windows-Authentifizierung

Bernd Jungbluth, Horn (info@berndjungbluth.de)

Kennwörter. Sie sind ein mehr oder weniger notwendiges Übel, stellen Sie doch den Zugang zu Systemen sicher. Dennoch sind sie nicht selten eine Schwachstelle im Sicherheitskonzept, wie im aktuellen Szenario dieser Beitragsreihe. Für eine automatische Anmeldung am SQL Server wird der Anmeldename und das Kennwort in der Access-Applikation gespeichert. Dort aber lassen sich diese Informationen nicht ausreichend schützen. Dabei kann es so einfach sein. Mit der Windows-Authentifizierung sind die Anmeldedaten für den Zugang zum SQL Server nicht mehr erforderlich. In diesem Teil der Beitragsreihe lernen Sie die Vorteile der Windows-Authentifizierung kennen.

Warnung

Die beschriebenen Aktionen haben Auswirkungen auf Ihre SQL Server-Installation. Führen Sie die Aktionen nur in einer Testumgebung aus. Verwenden Sie unter keinen Umständen Ihren produktiven SQL Server!

Das aktuelle Berechtigungskonzept der Beispielumgebung basiert auf der SQL Server-Authentifizierung. Der Zugang zum SQL Server erfolgt über die Anmeldungen **WaWiMa** und **WaWiPersonal**. Ein Anwender braucht lediglich das Kennwort zu einer der beiden Anmeldungen und schon kann er mit den entsprechenden Rechten auf die Daten des SQL Servers zugreifen. Einen direkten Bezug zum tatsächlichen Anwender gibt es bei dieser Art der Authentifizierung nicht.

Das ist bei der Windows-Authentifizierung anders. Hier müssen Sie keinen Anmeldenamen mitsamt Kennwort anlegen, sondern sie verweisen lediglich auf das Windows-Benutzerkonto des Anwenders. Diese so erstellte Anmeldung ordnen Sie dann wie gewohnt der Datenbank zu. Dem dadurch in der Datenbank erstellten Benutzer erteilen Sie dort die Rechte über die Datenbankrollen. Es ändert sich also lediglich die Authentifizierung am SQL Server, die Autorisierung an der Datenbank und die darauf folgende Rechtevergabe bleiben gleich.

Nur haben Sie weniger Aufwand bei der Verwaltung der Anmeldungen und durch den Wegfall der Kennwörter erhalten Sie mehr Sicherheit. Mehr noch, denn durch den Verweis zum Windows-Benutzerkonto erfolgt der Datenzugriff immer mit direktem Bezug zum Anwender. Ein anonymer Zugriff ist nicht mehr möglich. Sie sehen, die Windows-Authentifizierung ist einfach und effektiv.

Sie kennen die Windows-Authentifizierung bereits von Ihrem Windows-Benutzerkonto. Das haben Sie bei der Installation des SQL Servers als Anmeldung zugeordnet. Sie finden es im SQL Server Management Studio unter **Sicherheit|Anmeldungen**. Diese Anmeldung ist Mitglied der Serverrolle **sysadmin** und bietet Ihnen somit alle Rechte innerhalb Ihres SQL Servers, weshalb sie sich nicht für einen adäquaten Test der Rechtevergabe eignet. Dazu benötigen Sie ein weiteres Windows-Benutzerkonto.

Ein Windows-Benutzerkonto

Möglicherweise arbeiten Sie mit Ihrem Rechner in einer Windows-Domäne und Ihr Windows-Benutzerkonto wird über ein Active Directory verwaltet. Vielleicht aber testen Sie die Beispiele dieser Beitragsreihe in einer Entwicklungsumgebung mit einem lokalen Windows-Benutzerkonto. Da das Anlegen eines Windows-Benutzerkontos im Active Directory an fehlenden Rechten oder mangelnder

Bereitschaft Ihres IT-Systemadministrators scheitern könnte, legen Sie für die weitere Vorgehensweise ein lokales Windows-Benutzerkonto an. Dazu klicken Sie mit der rechten Maustaste auf das Windowslogo in der Taskleiste und wählen den Eintrag **Computerverwaltung** (siehe Bild 1). In der Computerverwaltung erweitern Sie auf der linken Seite den Ordner **Lokale Benutzer und Gruppen**.

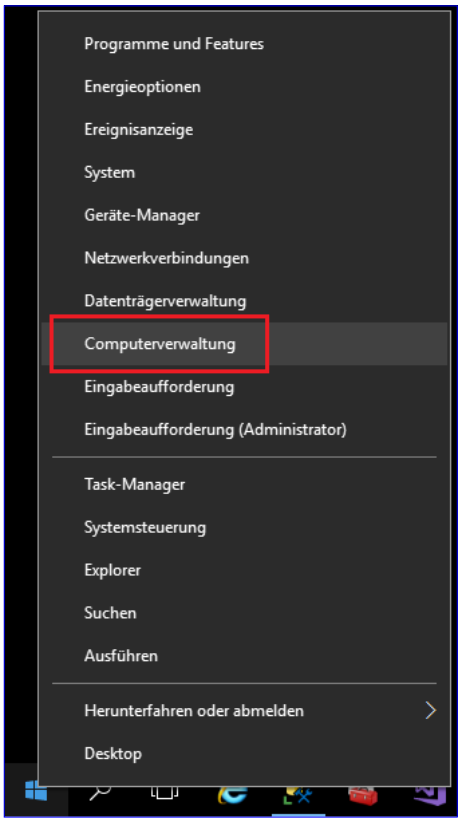


Bild 1: Start der Computerverwaltung

Nach einem Rechtsklick auf den Unterordner **Benutzer** öffnen Sie mit dem Eintrag **Neuer Benutzer** den gleichnamigen Dialog. Hier geben Sie als Benutzernamen **Bienlein** ein. Anschließend sichern Sie das Benutzerkonto mit einem Kennwort in den Eingabefeldern **Kennwort** und **Kennwort** bestätigen. Eine Neuvergabe des Kennworts bei der ersten Verwendung des Windows-Benutzerkontos ist nicht erforderlich. Es handelt sich schließlich um ein Testkonto, das Sie selbst verwenden. Daher können Sie die Option **Benutzer muss Kennwort bei der nächsten Anmeldung ändern** deaktivieren. Mit einem Klick auf **Erstellen** wird das neue Windows-Benutzerkonto angelegt (siehe Bild 2).

Glückwunsch. Nun haben Sie auf Ihrem Rechner einen virtuellen Anwender namens **Bienlein**. Beenden Sie den Dialog mit der Schaltfläche **Schließen**.

Die Anmeldung Bienlein

Im nächsten Schritt erstellen Sie für das Windows-Benutzerkonto **Bienlein** eine Anmeldung in Ihrem SQL Server. Starten Sie das SQL Server Management Studio und melden Sie sich an der SQL Server-Instanz mit Ihrem Windows-Benutzerkonto oder einer anderen Anmeldung mit administrativen Rechten an.

Im SQL Server Management Studio navigieren Sie im Objekt-Explorer zum Ordner **Sicherheit** und erweitern dort den Unterordner **Anmeldungen**. Wählen Sie in dessen Kontextmenü den Eintrag **Neue Anmeldung** (siehe Bild 3) und klicken Sie in dem darauffolgenden Dialog auf die Schaltfläche **Suchen**.

In dem Auswahlfenster **Benutzer oder Gruppe auswählen** legen Sie den Verweis zu dem Windows-Benutzerkonto **Bienlein** fest. Der schnellste Weg ist die direkte Eingabe des Windows-Benutzerkontos und einem anschließenden Klick auf **Namen überprüfen**. Ist die Eingabe korrekt, wird diese durch Ihren Rechnernamen ergänzt (siehe Bild 4). Mit einem Klick auf **OK** übernehmen Sie die Auswahl.

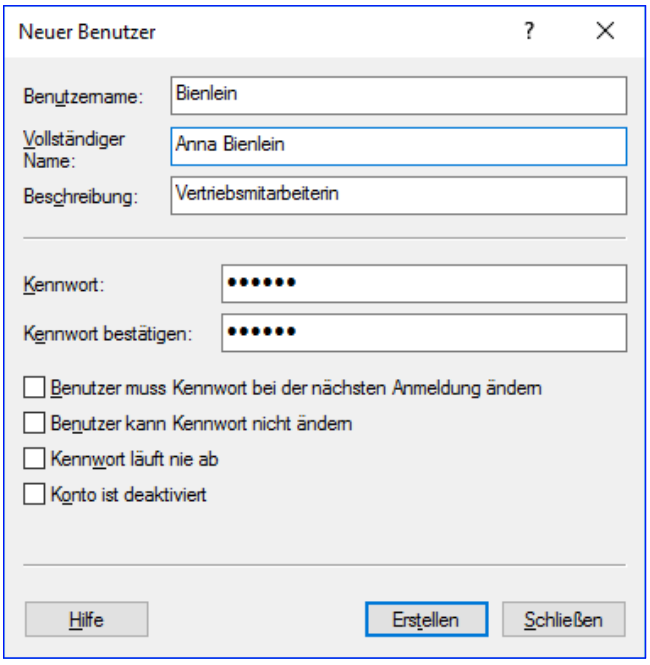


Bild 2: Windows-Benutzer Bienlein

Die Windows-Benutzerkonten Ihrer Domäne werden Sie über diesen Weg nicht finden. Hierfür müssen Sie in dem Auswahlfenster **Benutzer oder Gruppe auswählen** zunächst über die Schaltfläche **Pfade** den Suchpfad zu Ihrer Domäne festlegen. Anschließend können Sie wie oben beschrieben das Benutzerkonto aus Ihrem Active Directory auswählen.

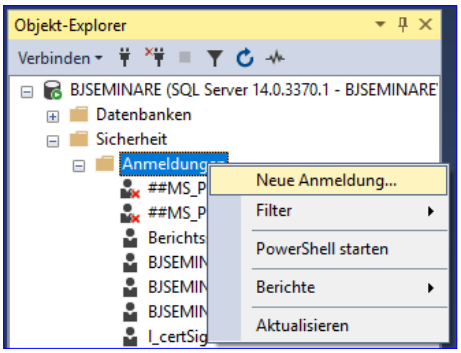


Bild 3: Neue Anmeldung im SQL Server

Sie haben es bestimmt bemerkt. Die Option **Windows-Authentifizierung** ist im Dialog standardmäßig aktiviert. Aus gutem Grund. Schließlich empfiehlt Microsoft diese Authentifizierungsmethode. Die SQL Server-Authentifizierung hingegen sollte nur in Ausnahmefällen verwendet werden. Zum Beispiel, wenn für den Betrieb der Client-Applikation keine Windows-Benutzerkonten verfügbar sind.

Die weiteren Schritte sind Ihnen von den beiden Anmeldungen **WaWiMa** und **WaWiPersonal** bekannt. Als Erstes weisen Sie der neuen Anmeldung in der Auswahlliste **Standarddatenbank** die Datenbank **WaWi_SQL** zu. Sollte mit einer Client-Applikation eine Verbindung zum SQL Server ohne die explizite Angabe einer Datenbank erfolgen, wird bei dieser Anmeldung immer die Datenbank **WaWi_SQL** verwendet.

In der Seite **Benutzerzuordnung** autorisieren Sie die Anmeldung für den Zugriff auf die Datenbanken. Hierzu wählen Sie im oberen Bereich die Datenbank **WaWi_SQL** aus. Durch diese Auswahl wird in der Datenbank ein Benutzer erstellt. Den Namen dieses Benutzers sehen Sie neben der gewählten Datenbank in der Spalte **Benutzer**.

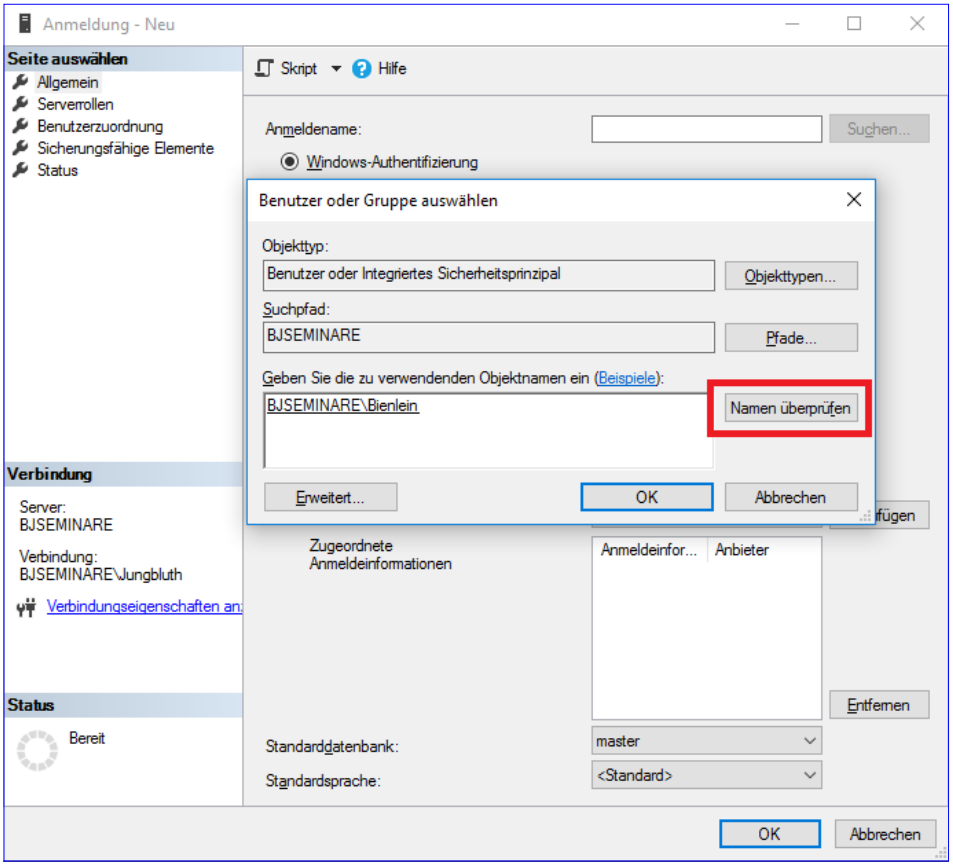


Bild 4: Anmeldung mit Windows-Authentifizierung

Zwar können Sie den Namen ändern, es ist jedoch nicht unbedingt empfehlenswert. Eines der obersten Gebote der Security ist die Einfachheit. Gestalten Sie die Vergabe und Pflege von Zugriffsberechtigungen so einfach wie möglich. Mit einem von der Anmeldung abweichenden Benutzernamen erhöhen Sie nur den Aufwand. Bei einer Änderung der Zugriffsrechte dieses Benutzers müssten Sie immer erst einmal nachschauen, zu welcher Windows-Authentifizierung er letztendlich gehört.

Ein Standardschema brauchen Sie nicht anzugeben. Ohne Angabe wird das Schema **dbo** verwendet. Die Bedeutung dieses Schemas sowie die Vorteile eigener Schemata ist das Thema einer der nächsten Beiträge.

Die Zugriffsrechte des neuen Benutzers legen Sie im unteren Bereich des Dialogs fest. Aktivieren Sie die Datenbankrollen **db_datareader**, **db_datawriter** und **edb_execute** (siehe Bild 5). Mit dieser Auswahl darf der Benutzer die Daten aller Tabellen lesen und ändern sowie alle gespeicherten Prozeduren ausführen. Die Zuordnung zur Datenbankrolle **public** können Sie nicht ändern. Jeder Benutzer ist hier standardmäßig Mitglied. Die Möglichkeiten und vor allem die Gefahren dieser Datenbankrolle lernen Sie im nächsten Teil dieser Beitragsreihe kennen.

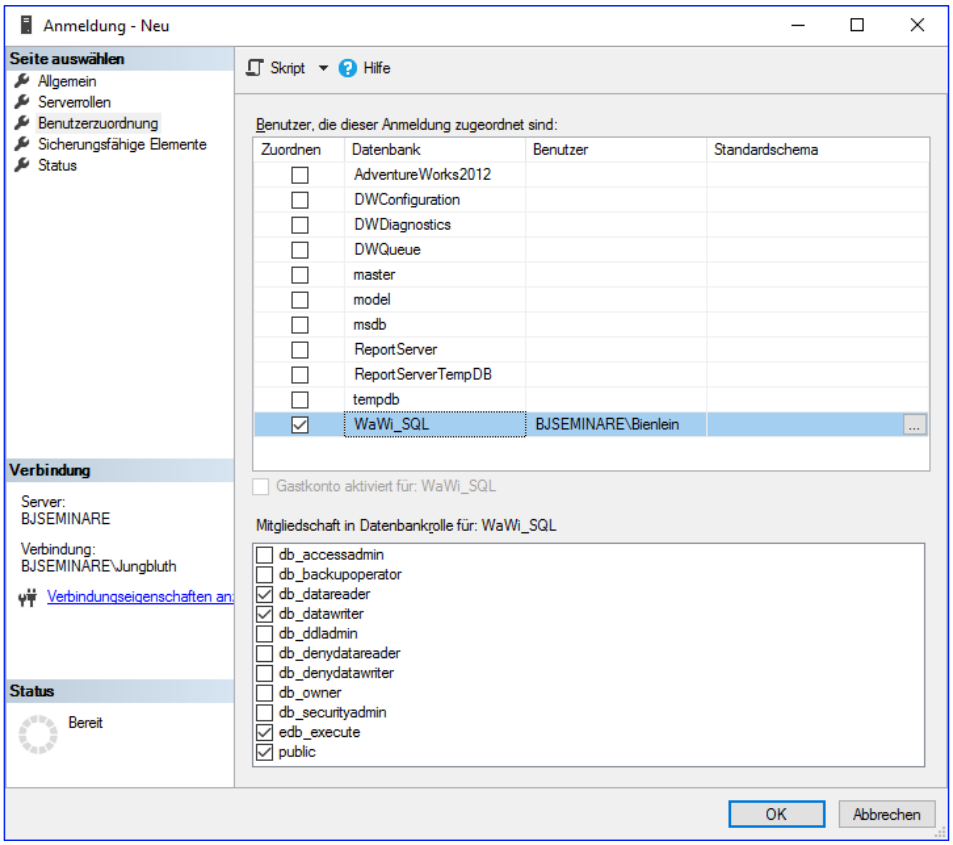


Bild 5: Benutzerzuordnung einer Anmeldung

Soweit so gut. Mit einem Klick auf **OK** legen Sie die neue Anmeldung an. Im Ordner **Anmeldungen** unter **Sicherheit** sehen Sie den neuen Eintrag **Bienlein**, wobei dem Namen **Bienlein** die Bezeichnung Ihres Rechners vorangestellt ist. Der Einfachheit halber wird im weiteren Text nur der Name des Windows-Benutzerkontos verwendet und auf den Hinweis des Rechnernamens verzichtet.

Mit der Anmeldung wurde der zugehörige Benutzer in der Datenbank **WaWi_SQL** erstellt. Wechseln Sie zur Datenbank **WaWi_SQL** und erweitern Sie dort den Ordner **Sicherheit**. Der Unterordner **Benutzer** listet nun neben den bekannten Einträgen **WaWiMa** und **WaWiPersonal** den neuen Benutzer **Bienlein** auf (siehe Bild 6).

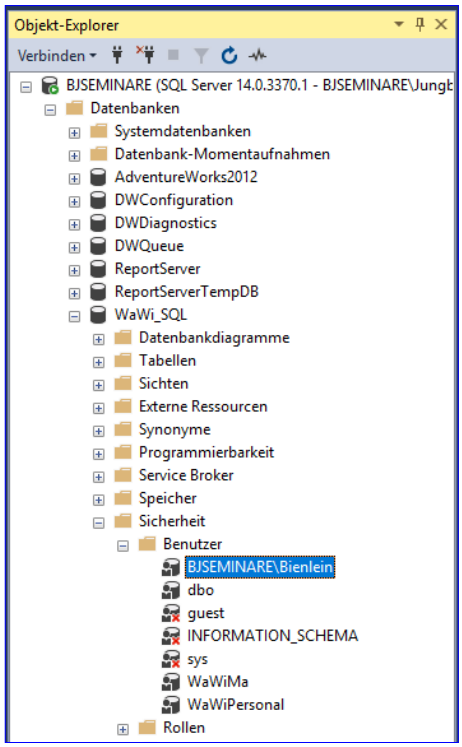


Bild 6: Datenbankbenutzer Bienlein

Frau Bienlein arbeitet im Vertrieb. Daher soll sie dieselben Rechte erhalten wie der Benutzer **WaWiMa**. Diesem wird aktuell der Zugriff auf die Tabellen und gespeicherten Prozeduren der Personalabteilung verweigert. Um diese Einschränkungen festzulegen, öffnen Sie mit einem Doppelklick auf den Eintrag **Bienlein** den Dialog **Datenbankbenut-**

zer. Auf der Seite **Sicherungsfähige Elemente** legen Sie die Rechte für einzelne Tabellen und gespeicherte Prozeduren fest. Zur Auswahl dieser Objekte gibt es mehrere Wege. Ein Weg erfolgt über das Schema. Klicken Sie auf **Suchen** und aktivieren Sie im Auswahlfenster **Objekte hinzufügen** die Option **Alle Objekte, die dem Schema angehören** (siehe Bild 7). Unter **Schemaname** wählen Sie das Schema **dbo** und bestätigen dies mit einem Klick auf **OK**.

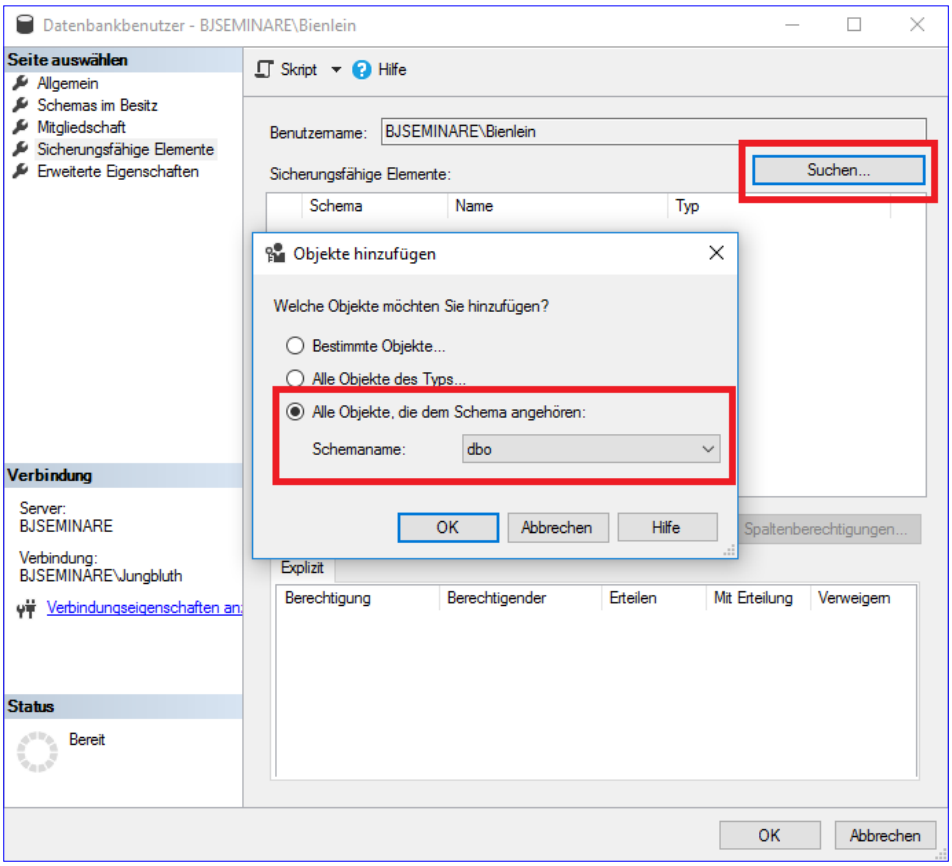


Bild 7: Auswahl der Objekte nach Schema

Der Dialog **Datenbankbenutzer** listet nun unter **Sicherungsfähige Elemente** alle Objekte der Datenbank vom Schema **dbo** auf. Dort markieren Sie die Tabelle **Bewerber** und aktivieren im unteren Bereich in der Zeile **Aktualisieren** die Option **Verweigern** (siehe Bild 8). Mit den Rechten zur gespeicherten Prozedur **pSelectGeburtstagsliste** verfahren Sie ähnlich. Sie wählen die gespeicherte Prozedur im oberen Bereich aus und aktivieren dann im unteren Bereich in der Zeile **Ausführen** die Option **Verweigern**.

Das war noch nicht alles. Sie müssen noch weitere Zugriffsrechte verweigern. Um Ihnen die umständlichen Klicks zu ersparen und gleichzeitig eine Dokumentation zu dieser Rechtevergabe zu erhalten, wählen Sie im oberen Teil des Dialogs unter **Skript** den Eintrag **Skript für Aktion in Fenster „Neue Abfrage“ schreiben**. Das SQL Server Management Studio zeigt Ihnen danach die T-SQL-Befehle zur Rechtevergabe in einer neuen Registerkarte. Genau hier werden Sie die Rechtevergabe vervollständigen. Doch vorher beenden Sie den Dialog **Datenbankbenutzer** mit einem Klick auf **Abbrechen**.

In der Registerkarte ergänzen Sie diese Zeile mit den Befehlen **SELECT**, **INSERT** und **DELETE**:

```
DENY UPDATE ON [dbo].[Bewerber] TO [Ihr Rechnername\Bienlein]
```

Danach sieht die Zeile wie folgt aus:

```
DENY SELECT, INSERT, UPDATE, DELETE ON [dbo].[Bewerber] TO [Ihr Rechnername\Bienlein]
```

Danach kopieren Sie die Zeile und fügen sie zweimal in das Skript ein. Ersetzen Sie in der ersten eingefügten Zeile den Namen der Tabelle mit **Mitarbeiter** und in der zweiten mit **Stellen**:

```
DENY SELECT, INSERT, UPDATE, DELETE ON [dbo].[Mitarbeiter] TO [Ihr Rechnername \Bienlein]
```

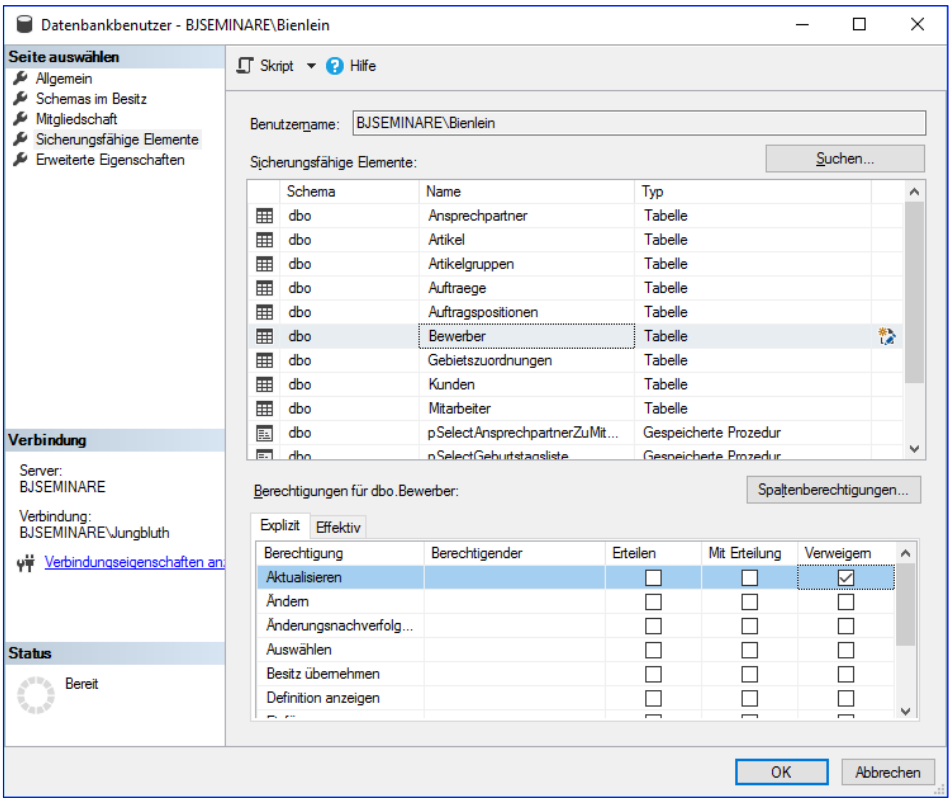


Bild 8: Rechtevergabe eines Benutzers

bankrollen **db_datareader**, **db_datawriter** und **edb_execute** Lese- und Schreibrechte an allen Tabellen der Datenbank sowie das Recht zum Ausführen aller gespeicherten Prozeduren. Der Zugriff auf die Tabellen **Bewerber**, **Mitarbeiter**, **Stellen** und der gespeicherten Prozedur **pSelectGeburtstagsliste** wird ihm jedoch durch das Recht **DENY** explizit verweigert – und dieses Recht steht über allen anderen.

Zur Kontrolle öffnen Sie erneut den Dialog **Datenbankbenutzer** mit einem Doppelklick auf den Eintrag **Bienlein**. Dieser zeigt Ihnen nun die erteilten Rechte in der Seite **Sicherungsfähige Elemente** (siehe Bild 10). Die zugeordneten Datenbankrollen sehen Sie in der Seite **Mitgliedschaft**.

```
DENY SELECT, INSERT, UPDATE, DELETE ON [dbo].[Stellen] TO
[Ihr Rechnername \Bienlein]
```

Die nächsten drei Zeilen mit **GO**, **USE WaWi_SQL** und einem weiteren **GO** sind für das Skript nicht erforderlich. Entfernen Sie diese Zeilen und ergänzen Sie anschließend das Skript mit entsprechenden Hinweisen und Kommentaren. Heraus kommt ein T-SQL-Skript wie in Bild 9, das Sie in Ihrer Dokumentation zur Rechtevergabe speichern. Mit einem Klick auf die Schaltfläche **Ausführen** erweitern Sie die Zugriffsrechte des Benutzers **Bienlein**.

Wobei es sich beim Erweitern hier eher um ein Einschränken handelt. Grundsätzlich besitzt der Benutzer **Bienlein** durch die Zuordnung zu den Datenbankrollen **db_datareader**, **db_datawriter** und **edb_execute** Lese- und Schreibrechte an allen Tabellen der Datenbank sowie das Recht zum Ausführen aller gespeicherten Prozeduren. Der Zugriff auf die Tabellen **Bewerber**, **Mitarbeiter**, **Stellen** und der gespeicherten Prozedur **pSelectGeburtstagsliste** wird ihm jedoch durch das Recht **DENY** explizit verweigert – und dieses Recht steht über allen anderen.

Mit der Rechtevergabe des Benutzers ist das Erstellen der Anmeldung vollständig. Die neue Anmeldung basiert auf einer Windows-Authentifizierung, in diesem Fall auf dem Windows-Benutzerkonto **Bienlein**. Jetzt ist noch die Beispielapplikation an die neue Authentifizierungsmethode anzupassen.

```
-- Ergänzende Rechtevergabe in Datenbank WaWi_SQL
-- Erteilt am 20211008 von Datenbankadministrator

-- Verweigern des Zugriffs auf Daten der Personalverwaltung durch die Anmeldung BJSEMINARE\Bienlein

USE WaWi_SQL;
GO
-- Gespeicherte Prozeduren der Personalverwaltung
DENY EXECUTE ON dbo.pSelectGeburtstagsliste TO [BJSEMINARE\Bienlein];

-- Tabellen der Personalverwaltung
DENY SELECT, INSERT, UPDATE, DELETE ON dbo.Bewerber     TO [BJSEMINARE\Bienlein];
DENY SELECT, INSERT, UPDATE, DELETE ON dbo.Mitarbeiter  TO [BJSEMINARE\Bienlein];
DENY SELECT, INSERT, UPDATE, DELETE ON dbo.Stellen      TO [BJSEMINARE\Bienlein];
GO
```

Bild 9: Rechtevergabe per T-SQL

Access und die Windows-Authentifizierung

Die Authentifizierung am SQL Server findet in der Beispielapplikation mit der Funktion **fTabellenEinbinden** statt. Diese ermittelt über die Access-Abfrage **Anmeldedaten** die zum aktuellen Anwender passende SQL Server-Anmeldung. Für die Mitarbeiter im Vertrieb ist das die Anmeldung **WaWiMa**, für die Kollegen in der Personalabteilung die Anmeldung **WaWiPersonal**. Die Zuordnungen wie die Anmeldedaten sind in den Tabellen **Benutzer** und **Datenquellen** hinterlegt. Zum Schutz dieser Daten und insbesondere der dort gespeicherten Kennwörter sind beide Tabellen als Systemobjekte definiert und somit nicht ohne weiteres sichtbar.

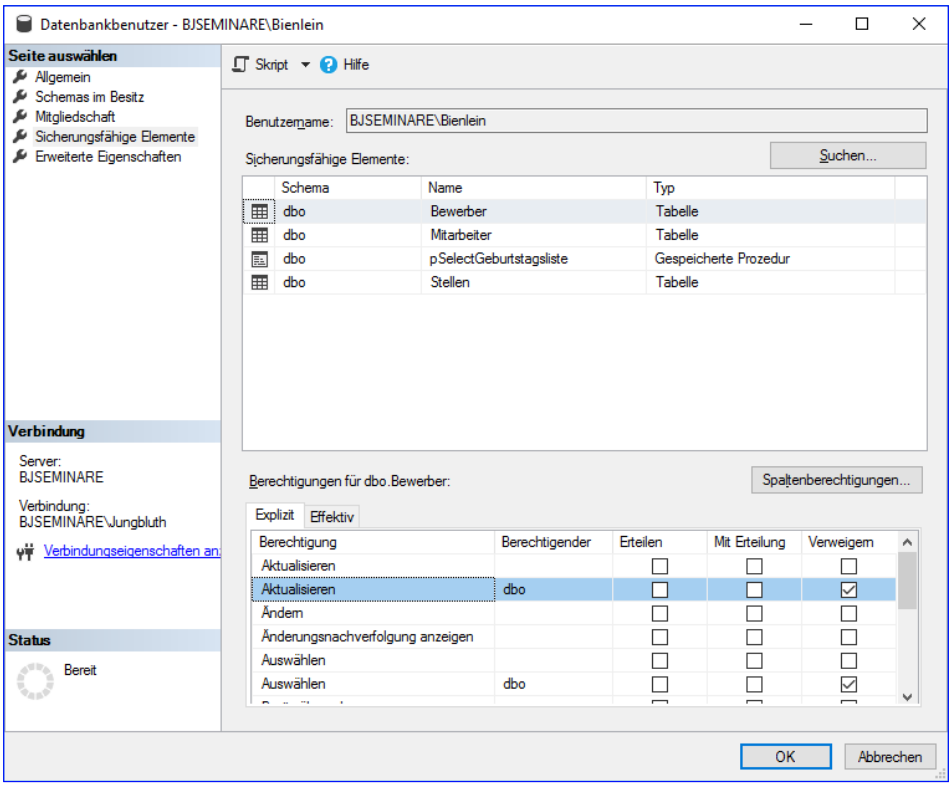


Bild 10: Verweigerter Zugriff für Bienlein

Der ganze Aufwand ist nur zum Schutz der Kennwörter zu den Anmeldungen **WaWiMa** und **WaWiPersonal** erforderlich. Diese dürfen lediglich den Entwicklern der Applikation beziehungsweise den Datenbankadministratoren bekannt sein. Den Anwendern und vor allem möglichen Angreifern soll das unerlaubte Ermitteln und Einsetzen dieser Kennwörter so schwer wie möglich gemacht werden.

Wobei letztere wohl eher mit einem Texteditor direkt in die ACCDB schauen. Was auf diesem Weg dort alles zu sehen ist, haben Sie im Teil 7 der Beitragsreihe erfahren.

Mit der Windows-Authentifizierung fällt der komplette Aufwand weg. Das Speichern von Kennwörtern ist nicht mehr erforderlich. Die Zugriffsrechte ergeben sich durch das aktuell verwendete Windows-Benutzerkonto des Anwenders. Frau Bienlein greift nun mit der zu ihrem Windows-Benutzerkonto eingerichteten Anmeldung auf die Datenbank **WaWi_SQL** zu. Damit diese Art der Anmeldung funktioniert, muss die Verbindungszeichenfolge in der Beispielapplikation angepasst werden.

Öffnen Sie die Beispielapplikation **WaWi v2** und wechseln Sie zur Funktion **fOdbcPerTabelle** im Modul **OBDC**. Hier wird die Verbindungszeichenfolge zusammengesetzt. Die nun anstehenden Änderungen werden die Funktion um einiges verkürzen. Es beginnt mit der Quelle der einzelnen Bestandteile einer Verbindungszeichenfolge, der Abfrage **Anmeldedaten**. Diese liefert die Bezeichnung des SQL Servers, den Namen der Datenbank sowie die zum aktuellen Anwender passende Anmeldung inklusive des Kennworts. Da die Anmeldedaten nicht mehr benötigt werden, ändern Sie in der folgenden Zeile die Quelle des Recordsets von Anmeldedaten in Datenquellen.

```
Set rsVerbindung = CurrentDb.OpenRecordset("Datenquellen",
dbOpenSnapshot)
```

Durch den Wegfall der Anmeldedaten sind die Variablen **strBenutzer** und **strKennwort** ebenfalls nicht mehr notwendig. Daher löschen Sie die Deklaration der Variablen sowie diese beiden Zeilen:

```
If IsNull(rsVerbindung("Benutzer")) Then boKomplett =
False
If IsNull(rsVerbindung("Kennwort")) Then boKomplett =
False
```

Aus dem gleichen Grund trennen Sie sich auch von diesen Zeilen:

```
";UID=" & rsVerbindung("Benutzer") & _
";PWD=" & fKennwortLesen(rsVerbindung("Kennwort"))
```

Jetzt ändern Sie den Eintrag **Trusted_Connection=No** in **Trusted_Connection=Yes**. Durch diesen Parameter wird bei der Verbindung zum SQL Server die aktuelle Windows-Anmeldung des Anwenders übergeben. Das Erstellen der Verbindungszeichenfolge sieht nun so aus:

```
'Datenquelle ermitteln
Set rsVerbindung = CurrentDb.OpenRecordset(_
                        "Datenquellen", dbOpenSnapshot)
If Not rsVerbindung.EOF Then
    'Inhalte prüfen
    If IsNull(rsVerbindung("Server")) Then _
                                    boKomplett = False
    If IsNull(rsVerbindung("Datenbank")) Then _
                                    boKomplett = False
    'Anmeldedaten vollständig?
    If boKomplett = True Then
        'Connection definieren
        fOdbcPerTabelle = _
            "ODBC;DRIVER=ODBC Driver 17 for SQL Server" _
            & ";Trusted_Connection=Yes" & _
            ";Server=" & rsVerbindung("Server") & _
            ";Database=" & rsVerbindung("Datenbank")
    End If
Else
    'Keine Anmeldedaten gefunden
    boKomplett = False
End If
```

Natürlich könnten der Name des SQL Servers und der Name der Datenbank direkt in der Funktion eingetragen werden. Dadurch wäre die lokale Tabelle **Datenquellen** ebenso hinfällig. Aber warum sollten Sie auf diesen Komfort verzichten wollen? Über die beiden Werte lässt sich die Access-Applikation recht einfach auf einen anderen SQL Server oder eine andere Datenbank umschalten. So sind Sie zum Beispiel in der Lage, schnell zwischen Entwicklungsumgebung und Produktivumgebung zu wechseln. Dabei kann die Datenbank zur Entwicklungsumgebung unter einem anderen Namen im gleichen SQL Server liegen oder unter dem Originalnamen in einer anderen SQL Server-Instanz.

Fehlt der Name des SQL Servers beziehungsweise der Datenbank, erhalten Sie durch die Messagebox am Ende der Funktion eine Meldung. Diese können Sie noch etwas anpassen, denn sie prüft nicht mehr die Anmeldedaten, sondern lediglich die Verbindungsdaten:

```
'Verbindungsdaten unvollständig
If boKomplett = False Then
    MsgBox "Die Verbindungsdaten sind nicht _
        vollständig erfasst." & vbCrLf & "Bitte ergänzen _
          Sie die fehlenden Informationen.", vbCritical, _
                              CurrentDb.Properties!AppTitle
End If
```

Soweit zum Datenzugriff per ODBC der für die eingebundenen Tabellen und die Pass Through-Abfragen relevant ist. Die Beispielapplikation nutzt zusätzlich die Datenzugriffsmethode ADO und diese wiederum verwendet OLE DB für den Zugriff auf die Daten. Es ist also eine weitere Anpassung erforderlich.

Dazu wechseln Sie zur Funktion **fAdoPerTabelle** im Modul **ADO**. Die nun anstehenden Änderungen sind ähnlich der

eben durchgeführten. Als erstes ändern Sie hier ebenfalls die Quelle des Recordsets in Datenquellen.

```
Set rsVerbindung = CurrentDb.OpenRecordset(_
                        "Datenquellen", dbOpenSnapshot)
```

Danach entfernen Sie die Deklarationen der Variablen **strBenutzer** und **strKennwort** sowie die folgenden Zeilen:

```
";User ID=" & rsVerbindung("Benutzer") & _
";Password=" & fKennwortLesen(rsVerbindung("Kennwort"))
```

Anschließend ergänzen Sie den Aufbau der Verbindungszeichenfolge mit dem Parameter **Integrated Security**. Mit diesem Parameter findet bei OLE DB die Übergabe der aktuellen Windows-Anmeldung statt. Der Parameter akzeptiert als Wert sowohl **SSPI** wie auch **Yes**. Sie sollten an dieser Stelle **SSPI** verwenden, denn der Wert **Yes** wird nicht von allen OLE DB-Treibern unterstützt. **SSPI** steht für **Security Support Provider Interface**. Diese Schnittstelle ermöglicht die Verwendung von Funktionen verfügbarer Sicherheitsmodelle, unter anderem das Prüfen einer Authentifizierung. Mit dem Eintrag **Integrated Security** haben die Zeilen zu den Parametern **User ID** und **Password** keine Bedeutung mehr. Weshalb Sie die beiden Zeilen nun löschen. Danach sollte der Aufbau der Verbindungszeichenfolge so aussehen:

```
fAdoPerTabelle = "Provider=MSOLEDBSQL;Integrated _
       Security=SSPI;Server=" & rsVerbindung("Server") & _
                  ";Database=" & rsVerbindung("Datenbank")
```

Zum Abschluss passen Sie noch die Meldung am Ende der Funktion an. Am besten nutzen Sie denselben Text wie in der Funktion **fOdbcPerTabelle**.

Nach diesen Änderungen wird weder der Anmeldename noch das Kennwort zum Erstellen der Verbindungszeichenfolgen verwendet. Daher werden Sie die Quelle dieser Informationen nun ebenfalls löschen. Beide sind in der Tabelle **Datenquellen** hinterlegt. Bevor Sie die Tabelle ändern können, müssen Sie ihr zunächst den Status der Systemtabelle entziehen. Führen Sie dazu im VBA-Direktbereich die folgende Anweisung aus:

```
fTabelleAlsSystemObjekt "Datenquellen", False
```

Wiederholen Sie die Anweisung für die Tabelle **Benutzer**. Hier gibt es gleich eine weitere Änderung. Danach wechseln Sie zu Access in den Navigationsbereich und öffnen die Tabelle **Datenquellen** im Entwurfsmodus. Entfernen Sie die nicht mehr benötigten Spalten **Benutzer** und **Kennwort** und speichern Sie die neue Struktur der Tabelle. Anschließend öffnen Sie die Datenblattansicht. Ohne die Information der Anmeldedaten sehen Sie jetzt zwei Datensätze mit gleichem Inhalt. Beide zeigen die Bezeichnung zu Ihrem SQL Server und zur Datenbank **WaWi_SQL**. Löschen Sie einen der Einträge. Und wenn Sie schon beim Löschen sind, entfernen Sie gleich noch die Tabelle **Benutzer** und die Access-Abfrage **Anmeldedaten**. Jegliche Information dieser Tabelle sowie der Abfrage ist für die Windows-Authentifizierung irrelevant.

Zugegeben, das waren jetzt viele Änderungen. Im Umkehrschluss zeigt dies aber sehr deutlich, von wieviel Ballast Sie sich bei einer Windows-Authentifizierung trennen. Zum Test des neuen Anmeldeverfahrens ist ein Neustart der Beispielapplikation erforderlich, denn es besteht bereits eine Verbindung zum SQL Server. Sind Sie der Installationsanleitung gefolgt, handelt es sich hierbei um die Anmeldung **WaWiPersonal** und somit um eine Anmeldung basierend auf einer SQL Server-Authentifizierung. Wie Sie im letzten Beitrag erfahren haben, werden Sie solche Verbindungen nur durch einen Neustart der Access-Applikation wieder los.

Ein Neustart öffnet das unsichtbare Formular **Verbindung**, das die Funktion **fTabellenEinbinden** und dort wiederum die eben angepasste Funktion **fOdbcPerTabelle** ausführt. Nachdem die Funktion **fTabellenEinbinden** alle Tabellen neu verknüpft und die Verbindungszeichenfolgen der Pass Through-Abfragen angepasst hat, wird das Formular

Steuerelemente per VBA erstellen

Im Beitrag »Formulare per VBA erstellen« (www.access-im-unternehmen.de/1332) haben wir gezeigt, wie Sie per VBA ein neues, leeres Formular erstellen und seine Eigenschaften einstellen. Darauf wollen wir in diesem Beitrag aufbauen und zeigen, wie Sie dem Formular per VBA die gewünschten Steuerelemente hinzufügen können. Und auch Steuerelemente haben eine Menge Eigenschaften, die wir nach dem Anlegen festlegen müssen – Position, Aussehen und auch wieder Ereigniseigenschaften. Nach der Lektüre des vorliegenden Beitrags haben Sie alle Werkzeuge, die Sie brauchen, um beispielsweise Access-Add-Ins zu nutzen, um einer Anwendung neue Formulare und Steuerelemente hinzuzufügen.

Vorbereitung: Formular anlegen

Als Vorbereitung wollen wir mit den Techniken, die wir im oben genannten Beitrag zum Erstellen von Formularen gelernt haben, zunächst ein neues Formular erzeugen. Dazu verwenden wir eine Funktion, die weitere Funktionen aus dem oben genannten Beitrag nutzt.

Die folgende Funktion erwartet den Namen des zu erstellenden Formulars und übergibt diesen an eine weitere Funktion namens **CreateNewForm**. Liefert diese den Wert **True** zurück, wurde das Formular erfolgreich erstellt.

Dann öffnen wir dieses Formular in der Formularansicht und stellen noch die Anzeige von Kopf- und Fußbereich ein – diese Bereiche werden wir weiter unten nämlich auch mit Steuerelementen versehen. Schließlich gibt die Funktion einen Verweis auf das noch in der Entwurfsansicht geöffnete **Form**-Objekt zurück:

```
Public Function GetNewForm(strForm As String) As Form
    Dim frm As Form
    If CreateNewForm(strForm) = True Then
        DoCmd.OpenForm strForm, acDesign
        Set frm = Forms(strForm)
        With frm
            RunCommand acCmdFormHdrFtr
            .Section(acHeader).Visible = True
            .Section(acFooter).Visible = True
        End With
    End If
    Set GetNewForm = frm
End Function
```

Steuerelemente anlegen per VBA

Diese Funktion rufen wir von der folgenden Prozedur aus auf und referenzieren das erstellte und geöffnete Formular mit der Variablen **frm**:

```
Public Sub SteuerelementeAnlegen()
    Dim frm As Form
    Set frm = GetNewForm("frmMitSteuerelementen")
    'Steuerelemente anlegen
End Sub
```

Danach können wir bereits loslegen, indem wir die gewünschten Anweisungen anstelle des Kommentars **'Steuerelemente anlegen** einfügen. Die minimale Version der Methode **CreateControl** hat nur zwei Parameter – den Namen des Formulars und den Typ des Steuerelements, hier **acTextBox** für ein Textfeld:

```
CreateControl frm.Name, acTextBox
```

Dieser Befehl legt bereits ein einfaches Steuerelement an, wobei die Position links oben im Detailbereich des Formulars ist.

Wichtig ist, dass Sie genau wie über die Benutzeroberfläche nur Steuerelemente anlegen können, wenn das Formular in der Entwurfsansicht geöffnet ist.

Die Methode CreateControl

Die Methode **CreateControl**, eigentlich ein Teil des **Application**-Objekts, aber auch ohne dessen Angabe aufrufbar, hat noch weitere Parameter. Nur die ersten beiden Parameter **FormName** und **ControlType** sind Pflichtparameter:

- **FormName**: Name des Formulars, in dem das Steuerelement angelegt werden soll.
- **ControlType**: Typ des Steuerelements, der durch eine Konstante angegeben wird – zum Beispiel **acTextBox**, **acLabel** oder **acCommandButton**.
- **Section**: Bereich, in dem das Steuerelement angelegt werden soll. Sinnvolle Werte für das Anlegen von Steuerelementen in Formularen sind **acDetail**, **acHeader** und **acFooter**. Die übrigen Möglichkeiten, die per IntelliSense angezeigt werden, sind für Berichte gedacht.
- **Parent**: Verweis auf das übergeordnete Steuerelement. Hiermit können Sie beispielsweise festlegen, zu welchem Textfeld ein Bezeichnungsfeld gehört.
- **ColumnName**: Angabe eines Feldes der Datensatzquelle des Formulars, an welches das Steuerelement gebunden werden soll. Füllt die Eigenschaft **Steuerelementinhalt** und mehr – siehe weiter unten.
- **Left**: Position vom linken Rand des Formulars
- **Top**: Position vom oberen Rand des Formulars
- **Width**: Breite des Steuerelements
- **Height**: Höhe des Steuerelements

Textfeld mit Bezeichnungsfeld anlegen

Um ein Textfeld mit einem Bezeichnungsfeld zu erstellen, benötigen Sie zwei Aufrufe der **CreateControl**-Methode. Mit dem ersten erstellen Sie das Textfeld, mit dem zweiten das Bezeichnungsfeld.

Für das Bezeichnungsfeld stellen wir den Parameter **Parent** auf das Textfeld ein. Auf diese Weise werden die beiden Steuerelemente so verknüpft, wie es auch beim Anlegen von Textfeldern über den Entwurf geschieht oder wenn Sie Felder aus der Feldliste in den Entwurf ziehen.

Für ein einfaches Textfeld mit einem Bezeichnungsfeld benötigen Sie die folgenden Anweisungen. Die Variablen **txt** und **lbl** dienen zum Referenzieren der neu erstellten Steuerelemente. Das Textfeld erstellen wir im Formular mit dem Namen, den wir mit **frm.Name** für das mit **frm** wie oben referenzierte Formular abfragen, als **acTextBox** im Bereich **acDetail**. Außerdem legen wir die Position mit 1400 Twips vom linken und 100 Twips vom oberen Rand und die Größe mit einer Breite von 2000 Twips und einer Höhe von 300 Twips fest:

```
Dim txt As TextBox
Dim lbl As Label
Set txt = CreateControl(frm.Name, acTextBox, _
                        acDetail, , , 1400, 100, 2000, 300)
```

Danach erstellen wir das Bezeichnungsfeld ebenfalls in **frm.Name** als **acLabel** im Bereich **acDetail**. Es soll dem zuvor erstellten Textfeld untergeordnet werden, also geben wir für den Parameter **Parent** den Namen des Textfeldes an (mit **txt.Name**). **txt.Name** funktioniert immer, auch wenn wir den Namen des Textfeldes nicht explizit festgelegt haben und liefert für das erste Textfeld im Formular beispielsweise den Wert **Text0**.

Die Position und die Abmessungen definieren wir so, dass das Bezeichnungsfeld links vom Textfeld erscheint. Anschließend legen wir noch die Beschriftung fest, indem wir die Eigenschaft **Caption** für das zuvor definierte und mit

der Variablen **lbl** referenzierte Bezeichnungsfeld auf den Wert **Textfeld:** festlegen:

```
Set lbl = CreateControl(frm.Name, acLabel, _
                acDetail, txt.Name, , 100, 100, 1200, 300)
lbl.Caption = "Textfeld:"
```

Indem Sie das Bezeichnungsfeld mit dem **Parent**-Parameter an das Textfeld binden, sorgen Sie gleichzeitig dafür, dass die Eigenschaft **Bezeichnungsname** des Textfeldes auf den Namen des Bezeichnungsfeldes eingestellt wird.

Das Ergebnis finden Sie in Bild 1 (siehe Prozedur **FormularMitTextfeldUndBezeichnungsfeld**).

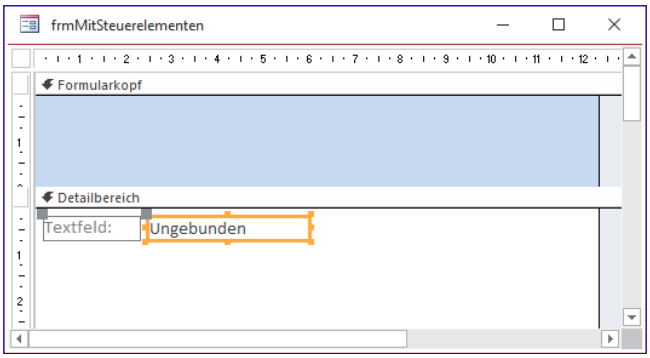


Bild 1: Textfeld mit Bezeichnungsfeld

Gebundene Textfelder anlegen

Nun gehen wir einen Schritt weiter und wollen zwei gebundenen Textfelder anlegen. Dazu stellen wir als Erstes die Eigenschaft **Recordsource** (**Datensatzquelle**) des Formulars auf die Tabelle **tblArtikel** ein:

```
frm.RecordSource = "tblArtikel"
```

Danach erstellen wir nacheinander zwei Textfelder mit Bezeichnungsfeldern, wobei wir im Gegensatz zum vorherigen Beispiel nun den Parameter **ColumnName** nutzen, um das Feld anzugeben, an welches das jeweilige Textfeld gebunden werden soll:

```
Dim txt As TextBox
Dim lbl As Label
Set txt = CreateControl(frm.Name, acTextBox, _
            acDetail, , "ArtikelID", 1500, 100, 2000, 300)
Set lbl = CreateControl(frm.Name, acLabel, _
                acDetail, txt.Name, , 100, 100, 1300, 300)
lbl.Caption = "ArtikelID:"
Set txt = CreateControl(frm.Name, acTextBox, _
          acDetail, , "Artikelname", 1500, 500, 2000, 300)
Set lbl = CreateControl(frm.Name, acLabel, _
                acDetail, txt.Name, , 100, 500, 1300, 300)
lbl.Caption = "Artikelname:"
```

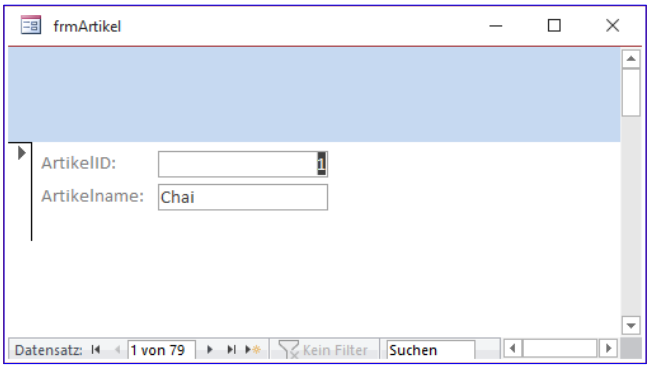


Bild 2: Formular mit gebundenen Textfeldern

Das so erstellte Formular sieht in der Formularansicht wie in Bild 2 aus (siehe Prozedur **GebundeneTextfelder**).

Gebundene Kombinationsfelder anlegen

Wenn wir die Tabelle **tblArtikel** betrachten, finden wir als Nächstes zwei Nachschlagefelder.

Diese wollen wir nun in Form von Kombinationsfeldern zum Formular hinzufügen. Für das erste Nachschlagefeld namens **KategorieID** sieht der dazu nötige Code wie folgt aus:

```
Dim cbo As ComboBox
...
Set cbo = CreateControl(frm.Name, acComboBox, _
          acDetail, , "KategorieID", 1500, 900, 2000, 300)
Set lbl = CreateControl(frm.Name, acLabel, _
                acDetail, cbo.Name, , 100, 900, 1300, 300)
lbl.Caption = "KategorieID:"
```

Wie wir in Bild 3 sehen, brauchen wir nur das zugrunde liegende Nachschlagefeld der Tabelle mit dem Parameter **ColumnName** anzugeben, damit ein neues Kombinationsfeld mit allen in der Tabelle für dieses Feld festgelegten Eigenschaften erstellt wird (siehe Prozedur **GebundeneKombinationsfelder**).

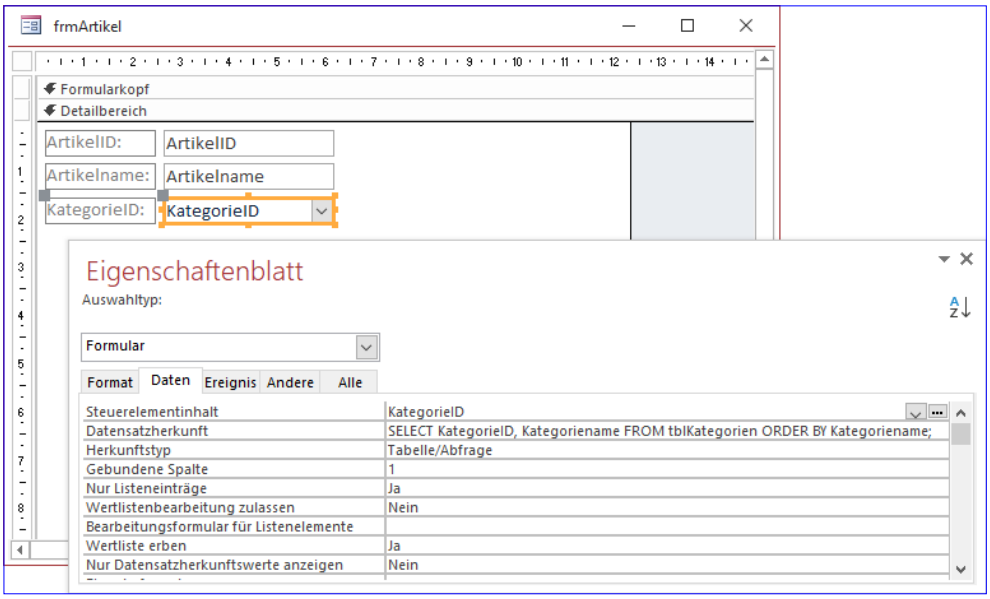


Bild 3: Formular mit gebundenem Kombinationsfeld

Übernahme von Eigenschaften bei gebundenen Steuerelementen

Wenn Sie mit dem Parameter **ColumnName** ein Feld der zugrunde liegenden Datensatzquelle angeben, stellt dies nicht nur die Eigenschaft **Steuerelementinhalt** auf den Namen des zu bindenden Feldes ein. Dazu gehören Eigenschaften wie **Textformat**, **Eingabeformat**, **Standardwert**, **Gültigkeitsregel** oder **Gültigkeitsmeldung**. Bei Kombinationsfeldern auf Basis von Nachschlagefeldern werden, wir oben gezeigt, sogar die Eigenschaften des Nachschlagefeldes übernommen.

Detailformular erstellen

Ein oft erledigter Schritt ist das Erstellen eines Detailformulars, also ein Formular, das die Daten einer Tabelle oder Abfrage anzeigen soll. Dazu sind normalerweise einige Schritte nötig wie das Erstellen des Formulars, das Hinzufügen der Datensatzquelle, das Hineinziehen der Felder in den Formularentwurf und gegebenenfalls noch das Anordnen der Felder, weil Access beim Hineinziehen von Feldern aus der Feldliste beispielsweise Kontrollkästchen immer links vom Bezeichnungsfeld anordnet und die übrigen Steuerelemente immer rechts vom Bezeichnungsfeld.

Deshalb zeigen wir im Folgenden eine praktische Anwendung der Techniken, die Sie weiter oben gelernt haben.

Die Prozedur **DetailformularErstellen** erwartet drei Parameter:

- **strForm**: Name des zu erstellenden Formulars
- **strRecordsource**: Name der Datensatzquelle
- **lngCaptionWidth**: Breite der Bezeichnungsfelder mit einem Standardwert von 1500 Twips

Die Prozedur, deren Code Sie in Listing 1 finden, schließt zunächst eine gegebenenfalls noch geöffnete Instanz des Formulars mit dem in **strForm** übergebenen Namen. Dann erstellt sie dieses mit der Funktion **GetNewForm** unter dem angegebenen Namen neu und referenziert das neu erstellte und in der Entwurfsansicht geöffnete Formular mit der Variablen **frm**.

Die einzige Formulareigenschaft, welche die Prozedur einstellt, ist **RecordSource**. Sie erhält den Wert aus **strRecordsource**.

Danach referenziert die Prozedur mit **db** das aktuelle **Database**-Objekt und mit **rst** ein Recordset auf Basis der mit

```
Public Sub DetailformularErstellen(strForm As String, strRecordsource As String, Optional lngCaptionWidth As Long = 1500)
    Dim frm As Form
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim fld As DAO.Field
    Dim strDisplayControl As String
    Dim strCaption As String
    Dim txt As TextBox
    Dim lbl As Label
    Dim cbo As ComboBox
    Dim chk As CheckBox
    Dim lngTop As Long
    DoCmd.Close acForm, strForm
    Set frm = GetNewForm(strForm)
    frm.RecordSource = strRecordsource
    Set db = CurrentDb
    Set rst = db.OpenRecordset(strRecordsource)
    For Each fld In rst.Fields
        strDisplayControl = 0
        strCaption = ""
        On Error Resume Next
        intDisplayControl = fld.Properties("DisplayControl")
        strCaption = fld.Properties("Caption")
        On Error GoTo 0
        If Len(strCaption) = 0 Then
            strCaption = fld.Name
        End If
        Select Case intDisplayControl
            Case acCheckBox
                Set chk = CreateControl(frm.Name, acCheckBox, acDetail, , fld.Name, lngCaptionWidth + 200, lngTop + 100)
                chk.Name = "chk" & fld.Name
                Set lbl = CreateControl(frm.Name, acLabel, acDetail, chk.Name, , 100, lngTop + 100, lngCaptionWidth, 300)
            Case acComboBox
                Set cbo = CreateControl(frm.Name, acComboBox, acDetail, , fld.Name, lngCaptionWidth + 200, _
                    lngTop + 100, 2000, 300)
                cbo.Name = "cbo" & fld.Name
                Set lbl = CreateControl(frm.Name, acLabel, acDetail, cbo.Name, , 100, lngTop + 100, lngCaptionWidth, 300)
            Case Else
                Set txt = CreateControl(frm.Name, acTextBox, acDetail, , fld.Name, lngCaptionWidth + 200, _
                    lngTop + 100, 2000, 300)
                txt.Name = "txt" & fld.Name
                Set lbl = CreateControl(frm.Name, acLabel, acDetail, txt.Name, , 100, lngTop + 100, lngCaptionWidth, 300)
        End Select
        lbl.Caption = strCaption
        lngTop = lngTop + 400
    Next fld
End Sub
```

Listing 1: Prozedur zum Erstellen eines Detailformulars

strRecordsource übergebenen Datensatzquelle, bei der es sich um eine Tabelle, Abfrage oder auch einen SQL-Ausdruck handeln kann. Anschließend durchläuft sie in einer **For Each**-Schleife alle Felder der **Fields**-Auflistung des Recordsets.

In dieser Schleife stellt die Prozedur zunächst die beiden Variablen **intDisplayControl** und **strCaption** auf die Werte **0** beziehungsweise eine leere Zeichenkette ein. **intDisplayControl** soll später den Steuerelementtyp aufnehmen, der für die Anzeige des Feldes in der Datenblattansicht definiert wurde. Dieser Steuerelementtyp wird beim Entwerfen einer Tabelle automatisch von Access vorgegeben – bei Feldern des Datentyps **Kurzer Text** zum Beispiel Textfeld, bei Nachschlagefeldern Kombinationsfeld oder bei **Ja/Nein**-Feldern Kontrollkästchen.

Bei reinen Zahlenfeldern wird kein Steuerelement voreingestellt. Bild 4 zeigt, wo Sie die Eigenschaft **DisplayControl** beziehungsweise **Steuerelement anzeigen** im Entwurf der Tabelle einsehen können. **DisplayControl** liefert Werte einer Enumeration wie **acTextBox**, **acComboBox** oder **acCheckBox**.

Die Variable **strCaption** soll den Wert der Eigenschaft **Beschriftung** aufnehmen oder, wenn keine Beschriftung angelegt wurde, den Feldnamen. Die Eigenschaft **Beschriftung** heißt unter VBA **Caption**. Sie können diese auf der Registerseite **Allgemein** des oben referenzierten Screenshots festlegen, diese Beschriftung wird dann als Spaltenüberschrift oder auch als Label-Beschriftung beim Ziehen von Feldern aus der Feldliste verwendet.

Warum setzen wir die beiden Variablen **intDisplayControl** und **strCaption** immer wieder auf **0** und **""**? Weil die Properties **DisplayControl** und **Caption**, mit denen wir diese füllen wollen, nicht immer vorhanden sind, und wir versuchen, diese unter Deaktivierung der Fehlerbehandlung aus den Properties abzufragen. Da wir dies innerhalb einer Schleife machen, würden die Variablen, wenn sie nicht aus den Properties gefüllt werden können, gegebenenfalls

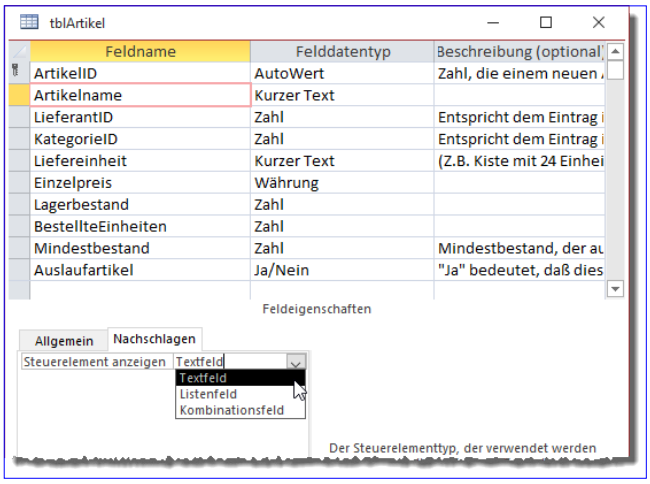


Bild 4: Wert der Eigenschaft **DisplayControl**

noch den Wert aus dem vorherigen Durchgang enthalten, wenn wir sie nicht leeren.

Die Werte der Eigenschaften **DisplayControl** und **Caption** fragen wir also bei deaktivierter Fehlerbehandlung über die Auflistung **Properties** ab und speichern diese in **intDisplayControl** und **strCaption**. War die Property **Caption** nicht vorhanden, ist **strCaption** danach leer und wir füllen es ersatzweise mit dem Namen des Feldes.

intDisplayControl nimmt einen der Werte **acCheckbox**, **acComboBox**, **acTextBox** oder **0** an. Diesen prüfen wir in der folgenden **Select Case**-Bedingung. Im Falle von **acCheckBox** erstellt die Prozedur mit der **CreateControl**-Methode ein neues Steuerelement des Typs **acCheckbox** im Detailbereich des Formulars. Das Feld wird an den mit **fld.Name** gelieferten Feldnamen gebunden.

Die Breite ermitteln wir aus dem Parameter **lngCaptionWidth** plus **200**, damit wir 200 Twips Abstand zwischen Label und Steuerelement erhalten. Den Abstand von oben stellen wir über den Wert der Variablen **lngTop** plus **100** ein. **lngTop** erhöhen wir nach jedem Steuerelement um **400**, damit das nächste Steuerelement um 400 Twips weiter unten angeordnet wird. Im Falle von **acCheckBox** brauchen wir Höhe und Breite nicht anzugeben, da die **CheckBox** immer gleich groß ist.

Anschließend weisen wir dem mit **chk** referenzierten neuen **CheckBox**-Steuerelement über die Eigenschaft **Name** den Namen des Feldes plus Präfix **chk** zu. Das folgende **Label**-Steuerelement legen wir links vom **CheckBox**-Steuerelement an. Weiter unten, außerhalb der **Select Case**-Bedingung, stellen wir noch seine Eigenschaft **Caption** auf den Wert aus **strCaption** ein.

Auf ähnliche Weise verfahren wir mit den Feldern, die für die Eigenschaft **DisplayControl** die Werte **acComboBox** oder **acTextBox** hinterlegt haben. Der Unterschied ist, dass wir diese Variablen des jeweiligen Typs zuweisen (**ComboBox** und **TextBox**) und dass wir für diese noch die Breite und die Höhe auf die Werte **2000** und **300** einstellen.

Schließlich weisen wir auch diesen Steuerelementen als Name ein Präfix wie **cbo** oder **txt** plus dem Feldnamen zu und erstellen die entsprechenden **Label**-Steuerelemente links neben dem Kombinations- oder Textfeld.

Die letzte Bedingung prüft übrigens nicht auf den Wert **acTextBox**, sondern verarbeitet alle übrigen Werte. Der Hintergrund ist, dass wir es ja nicht unbedingt mit **acTextBox** zu tun haben, sondern dass bei dem Steuerelement die Eigenschaft **DisplayControl** nicht definiert ist – dann soll dieses auch als **TextBox**-Steuerelement realisiert werden.

Nach dem Verlassen der **Select Case**-Bedingung stellen wir noch die Eigenschaft **Caption** des **Label**-Steuerelements aus **lbl** ein und erhöhen den Wert von **lngTop** um **400**, damit das folgende Steuerelement unter dem aktuellen Steuerelement angelegt wird.

Den Aufruf gestalten wir beispielsweise wie folgt:

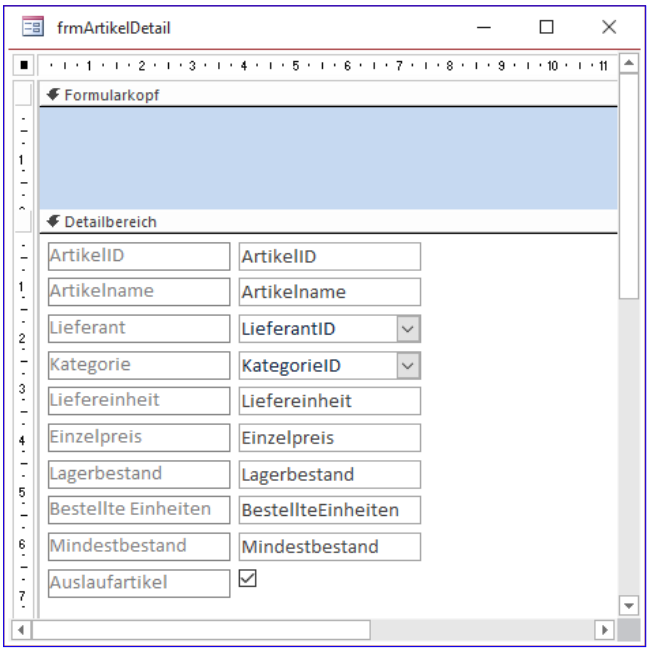


Bild 5: Entwurf des per Code erstellten Formulars zur Anzeige von Artikeln in der Detailansicht

```
DetailformularErstellen "frmArtikelDetail", _
                        "tblArtikel", 2000
```

Das Ergebnis finden Sie in Bild 5.

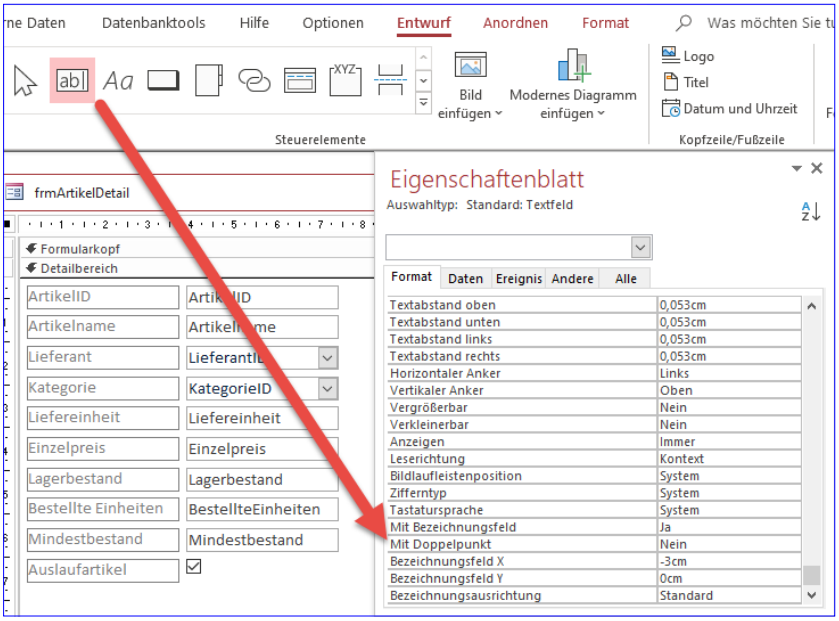


Bild 6: Einstellen der Eigenschaften für das Standardsteuerelement, hier für ein Textfeld.

Dateien und Verzeichnisse auswählen mit FileDialog

Bereits seit einiger Zeit bietet die Office-Bibliothek das FileDialog-Objekt an. In früheren Versionen gab es dort einige Einschränkungen, weshalb Programmierer gern auf Alternativen zurückgegriffen haben wie etwa die entsprechenden Funktionen der Windows-API oder auch die der nicht dokumentierten WizHook-Klasse. Irgendwann hat Microsoft jedoch auch für Access alle Funktionen der FileDialog-Klasse freigeschaltet, unter anderem auch das Auswählen von zu speichernden Dateien. Daher schauen wir uns in diesem Beitrag einmal an, welche Möglichkeiten die FileDialog-Klasse nun bietet und ob wir diese für unsere Zwecke nutzen können.

Die FileDialog-Klasse verfügbar machen

Um auf die **FileDialog**-Klasse und ihre Methoden und Eigenschaften zugreifen zu können, benötigen Sie entweder einen Verweis auf die Bibliothek **Microsoft Office x.0 Object Library** oder Sie referenzieren diese per Late Binding. Da wir die Vorzüge von IntelliSense zu schätzen wissen, nutzen wir hier die Bibliothek, die Sie im VBA-Editor einbinden können. Dazu wählen Sie den Menüeintrag **Extras|Verweise** aus und selektieren im nun erscheinenden Dialog den entsprechenden Eintrag (siehe Bild 1).

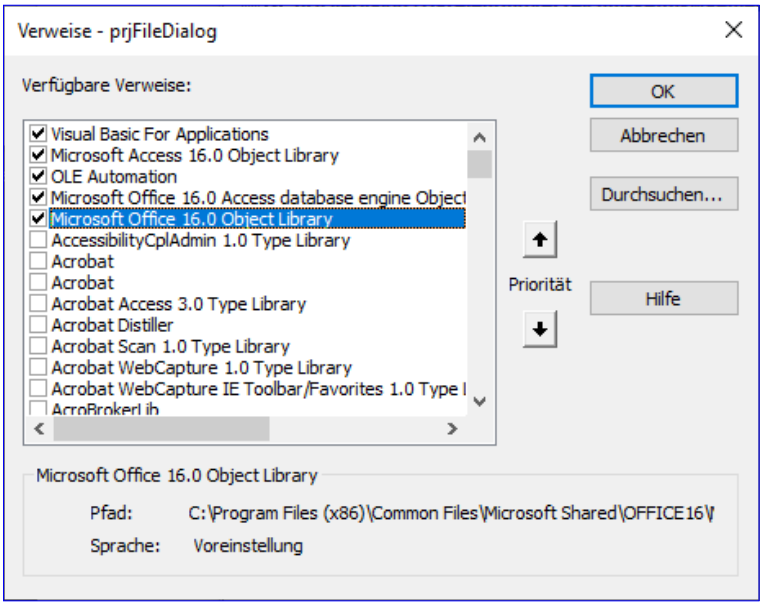


Bild 1: Verweis auf die Office-Bibliothek

Einen FileDialog anzeigen

Grundsätzlich zeigen Sie einen **FileDialog** wie folgt an:

```
Dim objFiledialog As Office.FileDialog
Set objFiledialog = FileDialog(msoFileDialogFilePicker)
objFiledialog.Show
```

Sie benötigen also eine Objektvariable, die den Typ **Office.FileDialog** aufweist und füllen diese mit einem Verweis auf die **FileDialog**-Klasse unter Angabe des Typs, den Sie wünschen – in diesem Fall **msoFileDialogPicker**.

Dann zeigen Sie den Dialog mit der Methode **Show** an. Das Ergebnis dieses einfachen Aufrufs finden Sie in Bild 2.

Auswahl im FileDialog auswerten

Nun können Sie nach der Auswahl wie im oben angegebenen Beispiel mit dem Ergebnis noch nichts anfangen. Das erledigen wir, indem wir das Ergebnis des Aufrufs prüfen und dann die selektierten Daten auswerten.

Die ersten beiden Zeilen bleiben gleich:

```
Dim objFiledialog As Office.FileDialog
Set objFiledialog = FileDialog(msoFileDialogFilePicker)
```

Danach rufen wir die **Show**-Methode jedoch als Teil einer **If...Then**-Bedingung auf und prüfen in dieser das Ergebnis.

Dieses kann **True** oder **False** lauten. Es lautet **True**, wenn der Benutzer die Eingabe mit der **Öffnen**-Schaltfläche abgeschlossen hat, und **False**, wenn er die **Abbrechen**-Schaltfläche wählt.

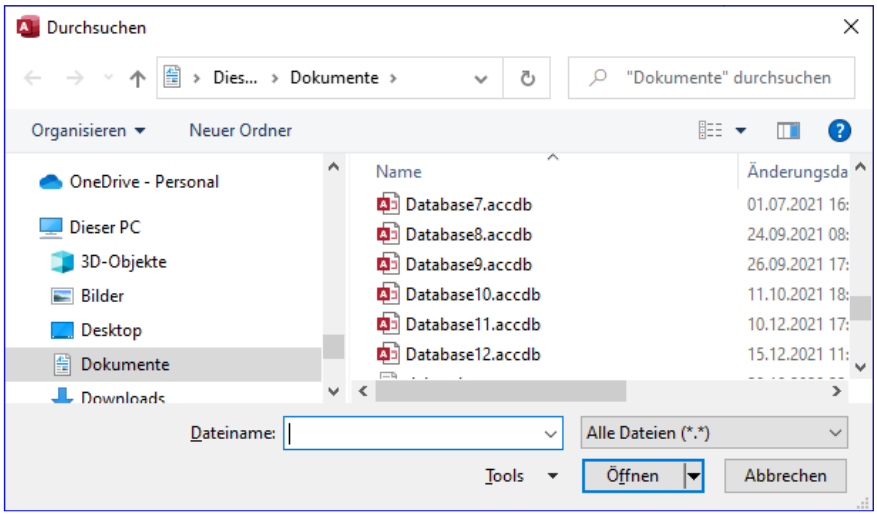


Bild 2: Ein einfacher **FileDialog**

Im ersten Fall greifen wir mit **objFileDialog.SelectedItems(1)** auf den ersten ausgewählten Eintrag zu und geben diesen im Direktbereich des VBA-Editors aus, im zweiten zeigen wir den Text **Keine Datei ausgewählt** an:

```
If objFiledialog.Show = True Then
    Debug.Print objFiledialog.SelectedItems(1)
Else
    Debug.Print "Keine Datei ausgewählt"
End If
```

Verschiedene Dialogtypen

Sie können verschiedene Dialogtypen nutzen. Dazu geben Sie unterschiedliche Werte beim Zuweisen des **FileDialog**-Objekts an.

Diese lauten:

- **msoFileDialogFilePicker**: Zeigt einen Dialog zum Auswählen einer oder mehrerer Dateien an.
- **msoFileDialogFolderPicker**: Zeigt einen Dialog zum Auswählen eines Verzeichnisses an (siehe Bild 3).
- **msoFileDialogOpen**: Zeigt den gleichen Dialog an wie **msoFileDialogFilePicker**, jedoch mit dem Titel **Öffnen**. Ist für andere Office-Anwendungen wie Word oder Excel geeignet, wo die Dokumente so direkt geöffnet werden können.
- **msoFileDialogSaveAs**: Zeigt einen Dialog zum Angeben einer zu speichernden Datei an. Es kann eine vorhandene Datei ausgewählt werden oder auch ein neuer Dateiname angegeben werden (siehe Bild 4). Die Schaltfläche zum Bestätigen der Eingabe ist hier mit **Speichern** beschriftet.

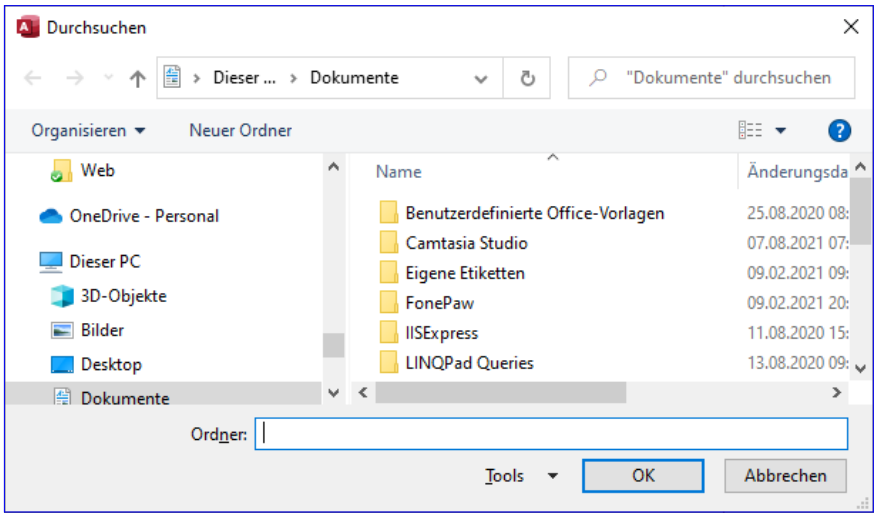


Bild 3: Der **FileDialog** mit **msoFileDialogFolderPicker** zeigt nur Ordner an, keine Dateien.

Eigenschaften werden beibehalten

Wenn Sie die nachfolgend beschrieben Eigenschaften nutzen, werden die Einstellungen beim erneuten Öffnen eines **FileDialog**-Fensters aus der gleichen Access-Session heraus beibehalten, wenn Sie nicht explizit neue Werte für die Eigenschaften angeben.

Mehrfachauswahl erlauben

Mit der Eigenschaft **AllowMultiSelect** können Sie festlegen, ob der Benutzer nur einen Eintrag (**False**) oder mehrere Einträge auswählen kann (**True**).

Diese Eigenschaft stellen Sie nach dem Zuweisen der **FileDialog**-Klasse an die Variable **objFileDialog** ein:

```
objFiledialog.AllowMultiSelect = True
```

Wenn Sie so einen **msoFileDialogFilePicker**-Dialog öffnen, können Sie beispielsweise bei gedrückter **Strg**-Taste mehrere Dateien wie in Bild 5 auswählen.

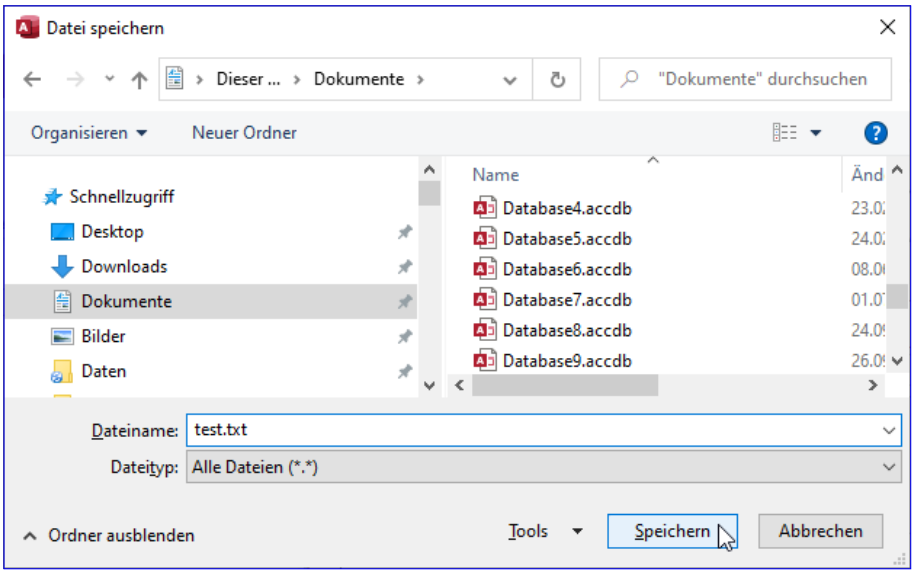


Bild 4: Der **FileDialog** mit **msoFileDialogSaveAs** erlaubt auch die Eingabe noch nicht vorhandener Dateinamen.

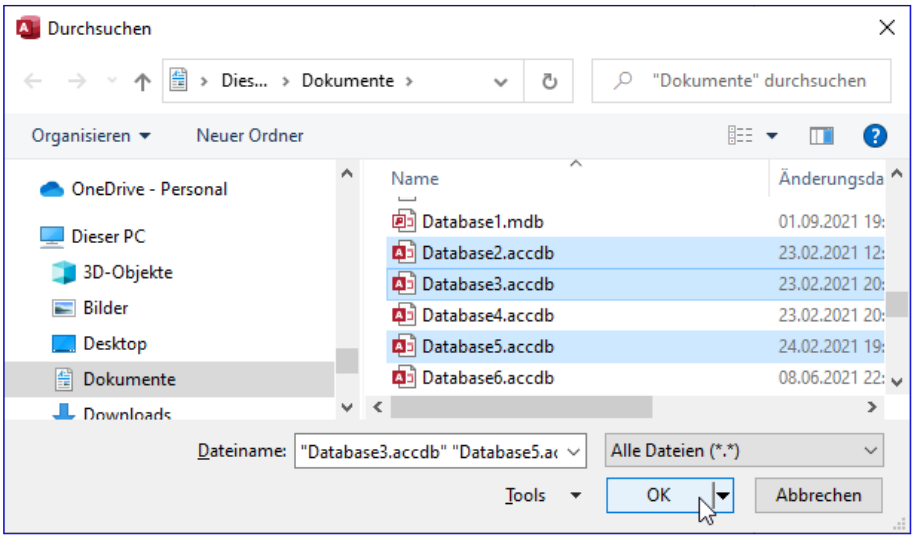


Bild 5: Mehrfachauswahl mit dem **FileDialog**-Objekt

Danach ist allerdings eine andere Auswertung erforderlich als bei der einfachen Auswahl, denn wir erhalten ja nicht nur einen, sondern gegebenenfalls auch mehrere Einträge zurück. Im Beispiel aus Listing 1 ermitteln wir zunächst die Anzahl der selektierten Dateien und geben diese im Direktbereich aus. Anschließend durchlaufen wir in einer Schleife alle Einträge der Auflistung **SelectedItems** und geben diese ebenfalls im Direktbereich aus.

Beschriftung der Schaltfläche zum Auswählen ändern

Mit der Eigenschaft **ButtonName** können Sie die sichtbare Bezeichnung der Schaltfläche zum Auswählen der gewählten Dateien oder Verzeichnisse ändern. Das gelingt allerdings nicht bei allen **FileDialog**-Typen auf Anhieb, sondern bei manchen erst nach der Auswahl eines Eintrags. Bei den Dialogen der Typen **msoFileDialogFolderPicker** und **msoFileDialogSaveAs** erscheint der

```
Private Sub FileDialog_FilePicker_Mehrfach()
    Dim objFiledialog As Office.FileDialog
    Dim l As Long
    Set objFiledialog = FileDialog(msoFileDialogFilePicker)
    objFiledialog.AllowMultiSelect = True
    If objFiledialog.Show = True Then
        Debug.Print "Es wurden " & objFiledialog.SelectedItems.Count & " Dateien ausgewählt:"
        For l = 1 To objFiledialog.SelectedItems.Count
            Debug.Print objFiledialog.SelectedItems(l)
        Next l
    Else
        Debug.Print "Keine Datei ausgewählt"
    End If
End Sub
```

Listing 1: Ausgabe einer Mehrfachauswahl im Direktbereich

gewünschte Text sofort. Das sieht beispielsweise wie im folgenden Codeausschnitt aus:

```
Dim objFiledialog As Office.FileDialog
Set objFiledialog = FileDialog(msoFileDialogFolderPicker)
objFiledialog.ButtonName = "FolderPicker"
If objFiledialog.Show = True Then
    ...
```

Das Ergebnis sehen Sie in Bild 6.

Bei den anderen Einstellungen wird der gewünschte Text erst angezeigt, wenn der Benutzer mindestens eine Datei ausgewählt hat (siehe Bild 7).

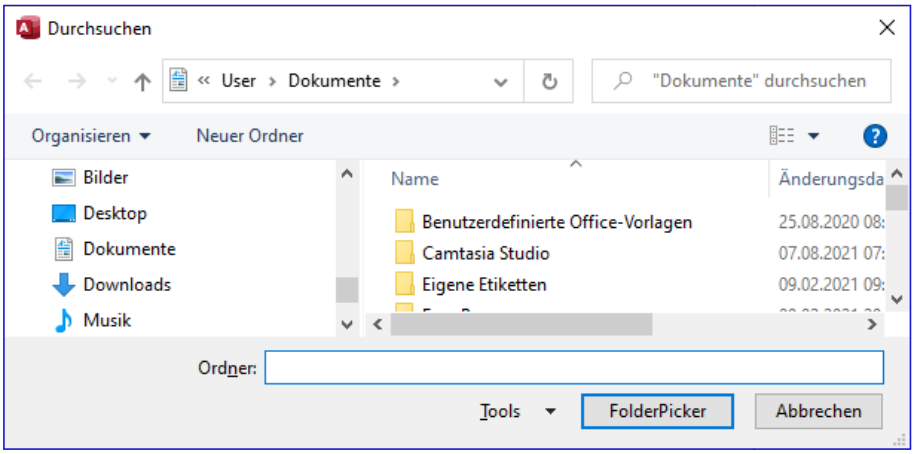


Bild 6: Ändern der Beschriftung der linken Schaltfläche

Überschrift des Dialogs einstellen

Mit der Eigenschaft **Title** können Sie den Text in der Titelzeile festlegen:

```
objFiledialog.Title = "Datei auswählen"
```

Beim Öffnen anzuzeigenden Ordner einstellen

Wenn Sie direkt beim Öffnen des Dialogs einen bestimmten Ordner anzeigen wollen, übergeben Sie diesen für die Eigenschaft **InitialFileName**.

Den Ordnernamen müssen Sie immer mit einem Backslash (\) abschließen. Wenn Sie beispielsweise das Verzeichnis **c:** anzeigen wollen, geben Sie **c:** an:

```
objFiledialog.InitialFileName =
"c:\"
```

Wollen Sie beispielsweise im Verzeichnis der aktuellen Datenbank starten, verwenden Sie die Funktion **CurrentProject.Path** und schließen dies mit dem Backslash-Zeichen ab:

```
objFiledialog.InitialFileName =
CurrentProject.Path & "\"
```

Datenzugriff mit .NET, LINQPad und LINQ to DB

.NET bietet eine ganze Menge Möglichkeiten, die uns unter Access/VBA nicht zur Verfügung stehen. Und diese Möglichkeiten wachsen ständig weiter, denn Entwickler stellen ihre eigenen Erweiterungen auf der NuGet-Plattform zur Verfügung. Was haben wir als Access-Entwickler nun davon? Mittlerweile gibt es Bibliotheken wie LINQ to DB, mit denen Sie leicht von einer .NET-Anwendung auf Access-Datenbanken zugreifen können. Und es gibt mit LINQPad eine Benutzeroberfläche, mit der Sie einfache Prozeduren mit Visual Basic programmieren können, ohne Visual Studio zu benötigen. Wir wollen diese beiden Tools als Vorbereitung zu einem weiteren Beitrag vorstellen. Dort werden wir diese nutzen, um die Tabellen einer Access-Datenbank mit zufälligen Beispieldaten zu füllen.

Wir werden in diesem Beitrag gleich mehrere Tools einführen, die Sie als Access-Entwickler noch nicht kennen – aber keine Sorge: Wir beschreiben alles ganz genau, sodass Sie die Techniken ohne Probleme für eigene Projekte anpassen können. Es handelt sich um die folgenden Tools:

- **LINQPad**: Dies ist ein Editor, mit dem Sie C#- und VB-Code eingeben und ausführen können, ohne dass Sie Visual Studio benötigen.

- **Linq To DB**: Dies ist eine Schnittstelle, mit der Sie beispielsweise von LINQPad direkt auf die Daten einer Access-Datenbank zugreifen können.

Damit haben wir ein perfektes Team: **LINQPad** ist unsere Entwicklungsumgebung zum Programmieren von Code, der unserer Datenbank Beispieldaten hinzufügt. **LINQ To DB** schafft die Verbindung zur Access-Datenbank.

Und in einem weiteren Beitrag namens **Beispieldaten generieren mit .NET und Bogus** (**www.access-im-unternehmen.de/1359**) zeigen wir, wie Sie mit einer weiteren Bibliothek namens **Bogus** die Tabellen einer Access-Datenbank mit zufällig generierten Beispieldaten füllen.

LINQPad herunterladen und installieren

Die »Spielwiese für .NET-Programmierer« können Sie unter dem folgenden Link herunterladen:

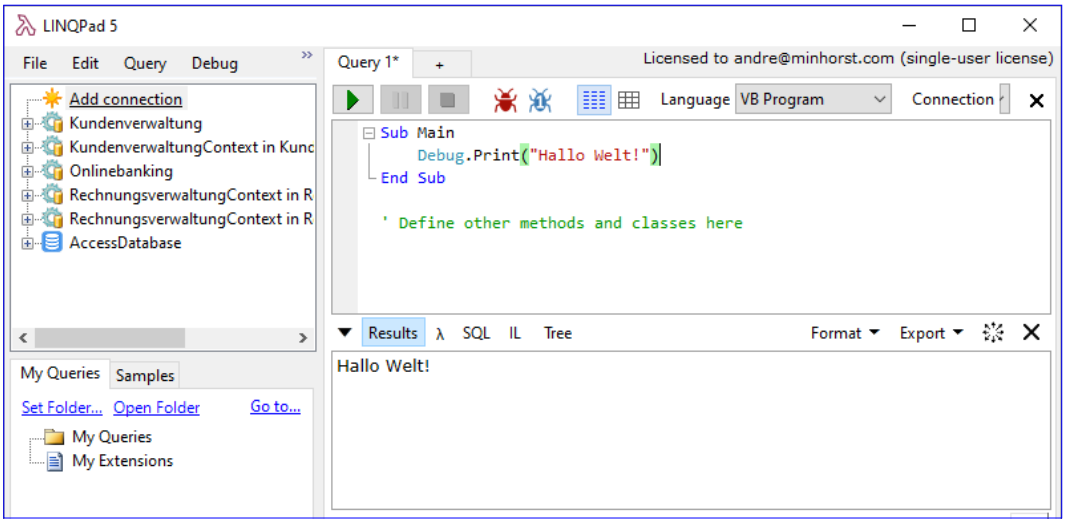


Bild 1: Ein erstes Beispiel in LINQPad

https://www.linqpad.net/

Danach installieren Sie das Tool und starten es.

Programme mit LINQPad schreiben

Da wir den Umgang mit VBA gewohnt sind, liegt es nahe, dass wir unter LINQPad **Visual Basic** nutzen und nicht C#. Um lauffähige Programme mit Prozeduren zu entwickeln, nutzen wir dort die Vorlage **VB Program**, die Sie für den einzigen bisher angezeigten Registerreiter unter **Language** auswählen.

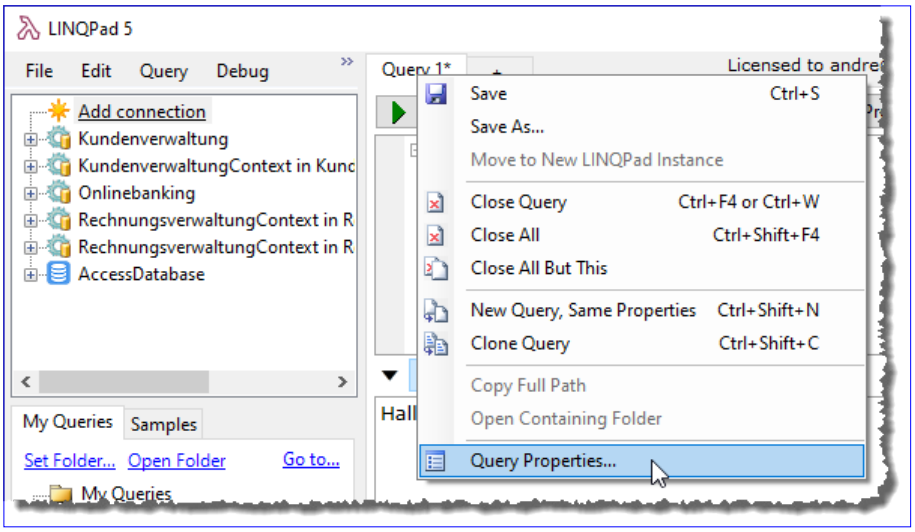


Bild 2: Anzeigen der Query Properties

Hier ergänzen Sie nun die bereits vorliegende Prozedur Main um eine **Debug.Print**-Anweisung und führen die Prozedur durch Betätigen der Taste **F5** aus (siehe Bild 1).

F5 löst immer die **Main**-Prozedur aus. Dieser können Sie nun den kompletten benötigten Code zuweisen, Sie können aber auch **Function**- oder **Sub**-Prozeduren definieren und aufrufen oder auch Klassen deklarieren und initialisieren.

An dieser Stelle direkt der Hinweis, dass Sie Parameter unter VB immer in Klammern einfassen müssen. Außerdem erfolgen Zuweisungen auch an Objektvariablen ohne das **Set**-Schlüsselwort.

Linq To DB installieren

Damit wir auf unsere Beispieldatenbank zugreifen können, benötigen wir eine Erweiterung namens **Linq To DB**. Um diese hinzuzufügen, öffnen Sie mit dem Kontextmenü-Eintrag **Query Properties...** für den aktuellen Registerreiter die Einstellung für das aktuelle Fenster (siehe Bild 2).

Dies öffnet den Dialog **Query Properties**, der unten die Schaltfläche **Add NuGet...** bereithält (siehe Bild 3). Über **NuGet**-Pakete können Sie verschiedene Pakete mit Tools zu einem Projekt hinzufügen.

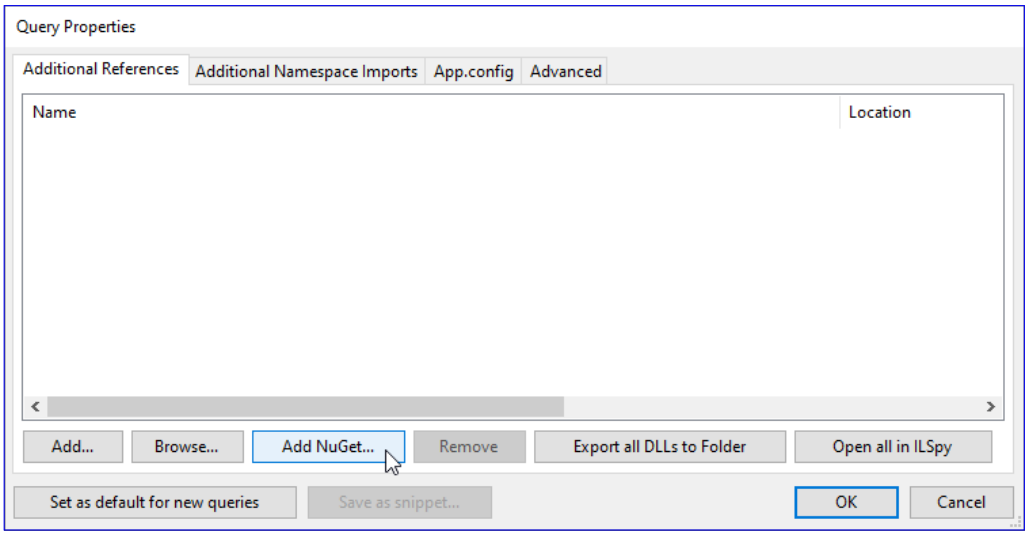


Bild 3: Hinzufügen von NuGet-Paketen

Klicken Sie diese an, erscheint der Dialog **LINQPad NuGetManager**.

Hier geben Sie unter **Search online** den Suchbegriff **Linq To DB** ein und finden schnell einen passenden Eintrag, den Sie mit einem Klick auf die Schaltfläche **Add to Query** zum aktuellen Query-Fenster hinzufügen (siehe Bild 4).

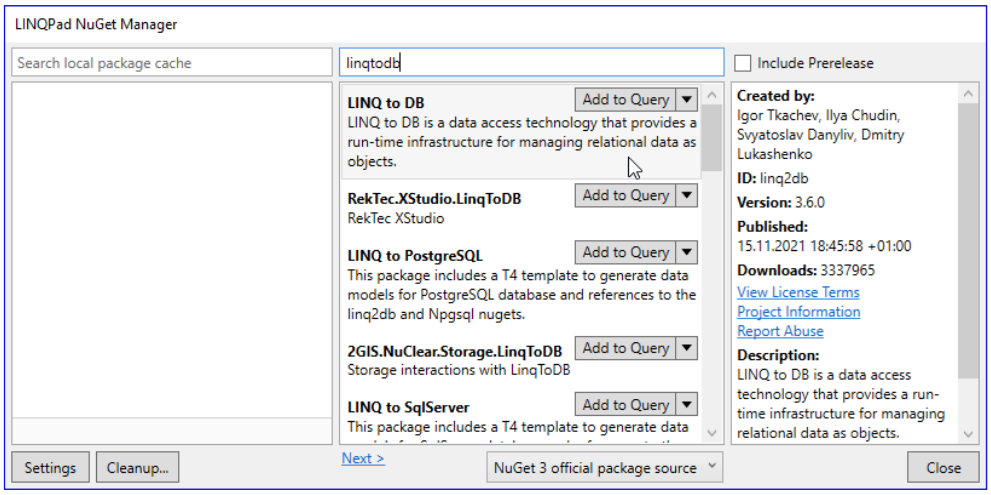


Bild 4: **LINQ to DB** hinzufügen

Tabellen der Datenbank verfügbar machen

Danach können Sie bereits eine Verbindung zur gewünschten Access-Datenbank hinzufügen. Dazu klicken Sie im linken Bereich des Fensters auf den Eintrag **Add connection**.

Es erscheint der Dialog **Choose Data Context**. Hier selektieren Sie im oberen Bereich den Eintrag **LINQ to DB** und klicken anschließend auf die Schaltfläche **Next** (siehe Bild 5).

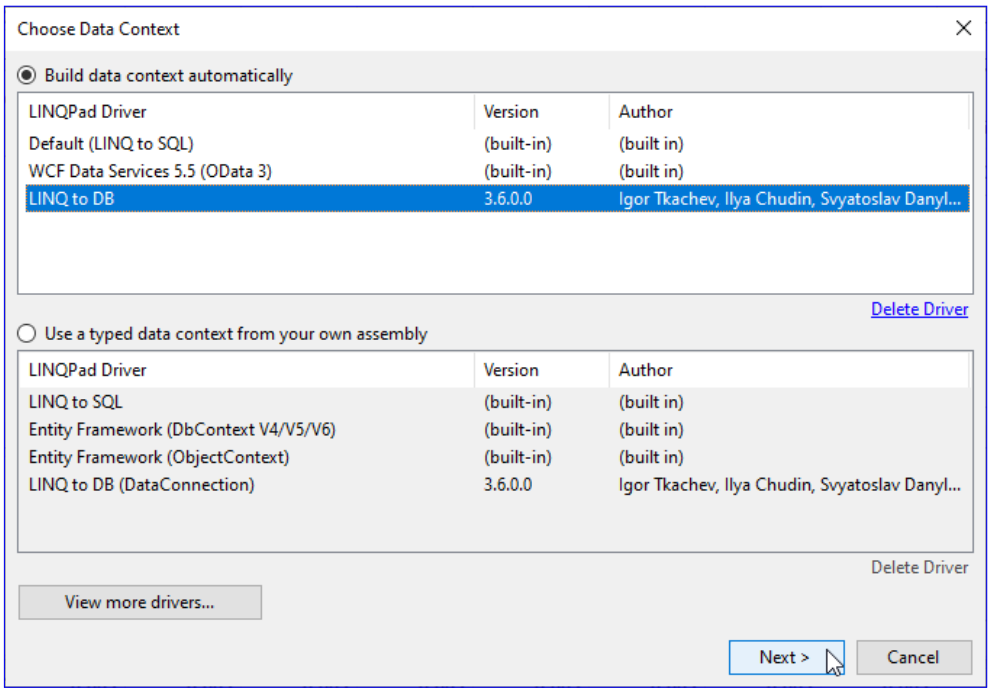


Bild 5: Herstellen einer Verbindung zur Datenbank

Der folgende Schritt zeigt den Dialog **LINQ to DB Connection** an. Hier wählen Sie oben zunächst den **Data Provider** aus, in diesem Fall **Microsoft Access (OleDb)**. Dann geben Sie die Verbindungszeichenfolge ein.

Bei der folgenden Verbindungszeichenfolge brauchen Sie nur den Pfad zur Access-Datenbank anzupassen:

```
Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\
Users\...\LINQPad.accdb;Persist Security Info=False
```

Damit testen Sie schließlich noch die Verbindung (siehe Bild 6).

War dies erfolgreich, können Sie den Dialog mit **OK** beenden.

Beispieldaten generieren mit .NET und Bogus

Das Produzieren von Beispieldaten ist immer wieder eine mühselige Aufgabe. Beispieldaten benötigen Sie, um beim Entwickeln neuer Anwendungen die Funktionen zu testen, die mit der Anzeige, dem Bearbeiten oder Löschen von Daten zusammenhängen. Und auch zum Testen des Hinzufügens von Daten benötigen Sie gegebenenfalls schon Daten in verknüpften Tabellen zur Auswahl. Unter .NET gibt es verschiedene Bibliotheken, die das Generieren von Beispieldaten erleichtern. Leider sind diese nicht so ohne Weiteres unter Access verfügbar. Zum Glück gibt es Tools, mit denen Sie diese Bibliotheken dennoch für Ihre Zwecke einsetzen können. In diesem Beitrag nutzen wir den Editor LINQPad, um Beispieldaten mit der Bogus-Bibliothek zu erzeugen und diese dann mit der Bibliothek LINQ to DB den Tabellen einer Beispieldatenbank hinzuzufügen.

Vorbereitungen

Für die beschriebenen Techniken benötigen Sie die Entwicklungsumgebung LINQPad sowie die Datenzugriffsbibliothek LINQ to DB. Wie Sie diese beiden herunterladen, installieren und verwenden, beschreiben wir im Beitrag **Datenzugriff mit .NET, LINQPad und LINQ to DB** (**www.access-im-unternehmen.de/1358**).

Wie Sie diese beiden nutzen, um mit einer weiteren Bibliothek namens Bogus zufällige Beispieldaten für eine Datenbank zu erzeugen, zeigen wir im vorliegenden Beitrag.

Bogus zum Projekt hinzufügen

Zum Ermitteln der Beispieldaten, die wir zu den Tabellen der Anwendung hinzufügen wollen, nutzen wir die bereits eingangs erwähnte Bibliothek **Bogus**.

Diese müssen wir genau wie **LINQ to DB** zunächst verfügbar machen. Dazu klicken Sie unter **LINQPad** mit der rechten Maustaste auf den Registerreiter des **Query**-Bereichs und wählen den Eintrag **Query Properties...** aus (siehe Bild 1).

Im nun erscheinenden Dialog **Query Properties** klicken Sie auf die Schaltfläche **Add NuGet...** und geben im folgenden Dialog unter **Search online** den Suchbegriff **Bogus** ein. Den nun angezeigten Eintrag aus Bild 2 fügen Sie dann mit einem Klick auf die Schaltfläche **Add to Query** hinzu.

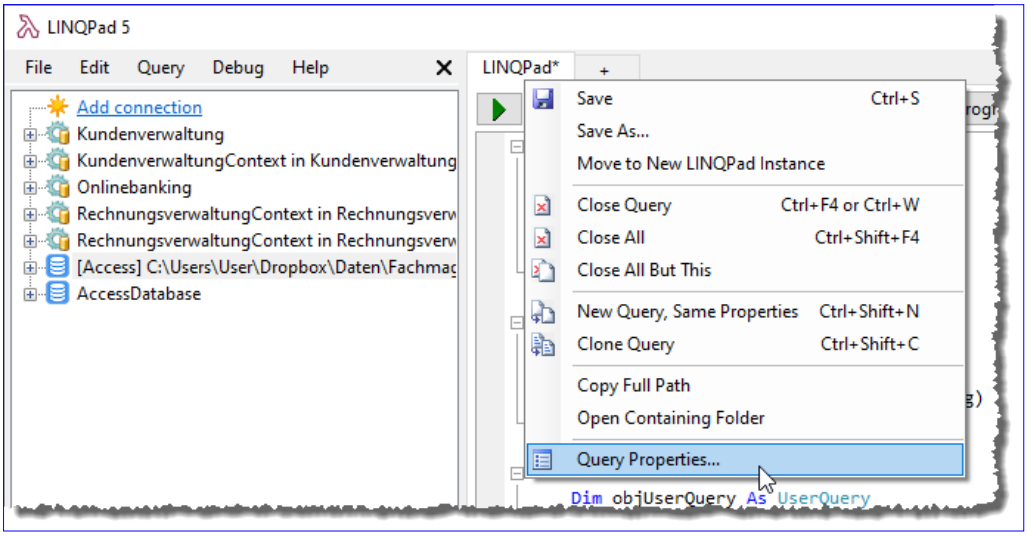


Bild 1: Aufrufen des Dialogs **Query Properties** per Kontextmenü

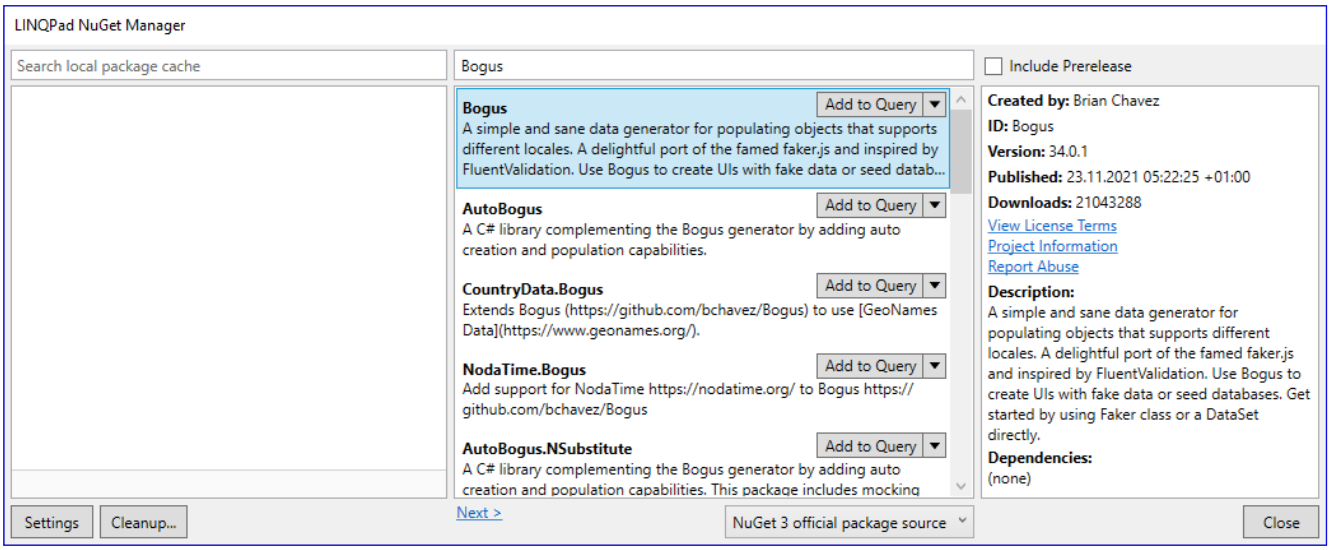


Bild 2: Hinzufügen des **Bogus**-Pakets

Es erscheint nun ein neuer Eintrag im linken Bereich des Dialogs **LINQPad NuGet Manager**. Hier klicken Sie noch auf **Add namespaces** (siehe Bild 3).

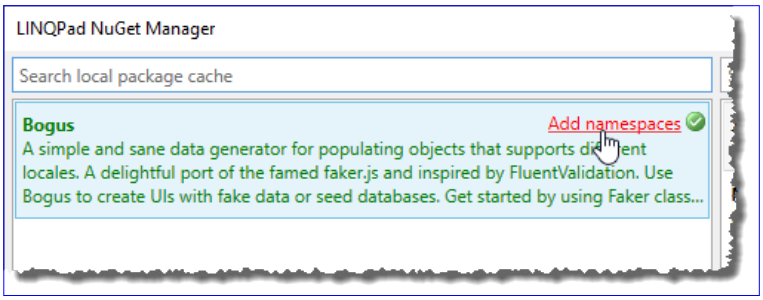


Bild 3: Hinzufügen des **Bogus**-Namespaces

Dies öffnet einen weiteren Dialog namens **Add Namespaces From NuGet Assemblies**. Hier wählen Sie den Eintrag **Bogus** aus und klicken auf **OK** (siehe Bild 4).

Nachdem Sie diesen Dialog und auch den Dialog **LINQPad NuGet Manager** geschlossen haben, finden Sie im Dialog **Query Properties** unter **Additional References** den Eintrag **Bogus** vor (siehe Bild 5).

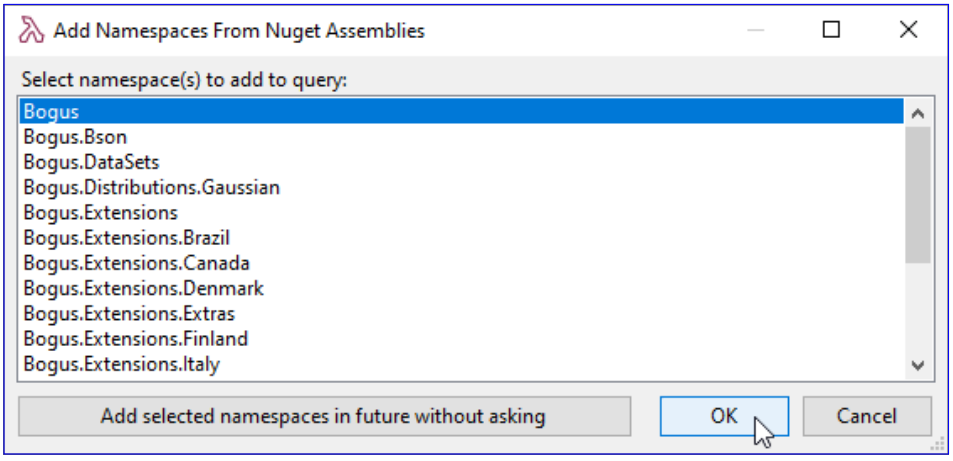


Bild 4: Hinzufügen eines Verweises auf den **Bogus**-Namespace

Damit sind die Vorbereitungen abgeschlossen und wir können uns dem Einsatz von Bogus zum Generieren von Beispieldaten zuwenden.

Vorweg: Anreden anlegen

Einen Schritt erledigen wir allerdings noch vorneweg – das Anlegen der Datensätze in der Tabelle **tblAnreden**. Für diese benötigen wir keinen Zufallsgenerator und somit auch nicht

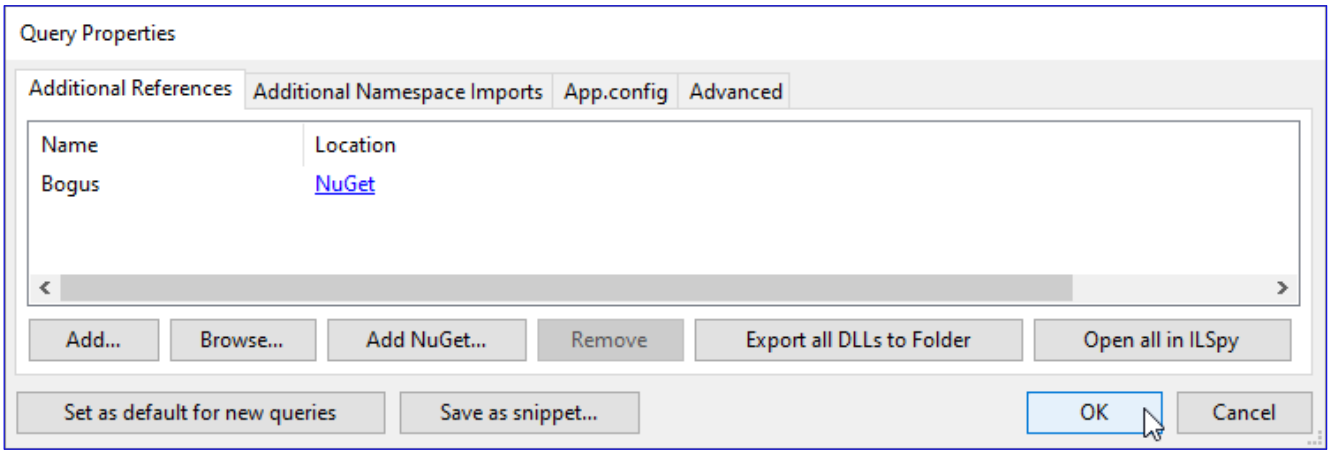


Bild 5: Der Verweis auf den **Bogus**-Namespace

die Bibliothek **Bogus**. Wir fügen die Anreden einfach mit der folgenden Prozedur hinzu:

```
Public Sub AnredenAnlegen
    Dim objUserQuery As UserQuery
    Dim objAnrede As tblAnreden
    objUserQuery = tblAnreden.DataContext
    objAnrede = New tblAnreden
    With objAnrede
        .ID = 1
        .Anredebezeichnung = "Herr"
    End With
    objUserQuery.Insert(objAnrede)
    objAnrede = New tblAnreden()
    With objAnrede
        .ID = 2
        .Anredebezeichnung = "Frau"
    End With
    objUserQuery.Insert(objAnrede)
End Sub
```

Den Code können wir mit Visual Basic übrigens auch wie folgt schreiben und sparen dabei nicht nur eine Variable, sondern auch einige Zeilen Code:

```
Public Sub AnredenAnlegen
    Dim objUserQuery As UserQuery
    objUserQuery = tblAnreden.DataContext
    objUserQuery.Insert(New tblAnreden With {.ID = 1, _
                        .Anredebezeichnung = "Herr"})
    objUserQuery.Insert(New tblAnreden With {.ID = 2, _
                        .Anredebezeichnung = "Frau"})
End Sub
```

Damit wir immer mit einer frisch aufgesetzten Tabelle **tblAnreden** starten, können wir zuvor die vorhandenen Anreden löschen:

```
Public Sub AnredenLoeschen
    Dim objUserQuery As UserQuery
    objUserQuery = tblAnreden.DataContext
    objUserQuery.Execute("DELETE FROM tblAnreden")
End Sub
```

Und schließlich geben wir die frisch erzeugten Anreden nachher einmal im Direktbereich von LINQPad aus:

```
Public Sub AnredenAusgeben
    Dim objAnrede As tblAnreden
    For Each objAnrede In tblAnreden
        Debug.Print(objAnrede.ID & " " _
                    & objAnrede.Anredebezeichnung)
    Next objAnrede
End Sub
```

Zufallsdaten mit Bogus ermitteln

Bogus bietet verschiedene Klassen, die Funktionen zum Erstellen von Zufallsdaten thematisch zusammenfassen. Diese werden unter folgendem Link ausführlicher beschrieben:

```
https://github.com/bchavez/Bogus#bogus-api-support
```

Es gibt die folgenden Klassen:

- **Address**: Liefert verschiedene Eigenschaften von Adressen wie Straße, PLZ, Ort und Land, aber auch beispielsweise Höhen- und Breitengrade oder Angaben wie Himmelsrichtungen.
- **Commerce**: Interessant für alles, was mit dem Handel zu tun hat. Bietet Daten wie Produktnamen, Farben, Adjektive oder Materialien, Preise, Werte für Barcodes oder Kategorien und Abteilungen.
- **Company**: Liefert Firmennamen, Gesellschaftsformen und Phrasen zu Unternehmen
- **Database**: Liefert Feldnamen, Datentypen und weitere Datenbank-relevante Informationen
- **Date**: Liefert Datumsangaben in verschiedenen Bereichen sowie Monate oder Wochentage
- **Finance**: Gibt beispielsweise IBANs aus, wobei Sie festlegen können, aus welchem Land diese stammen und ob diese formatiert sein sollen. Außerdem liefert diese Klasse Geldbeträge, Transaktionstypen, Währungen, Kreditkartennummern, Prüfziffern und vieles mehr.
- **Hacker**: Liefert zufällige Abkürzungen, Adjektive, Nomen, Verben oder komplette Phrasen, allerdings alles nur auf Englisch.
- **Images**: Hiermit können Sie beispielsweise URLs zum Download von Bildern abfragen, welche Bilder in den angegebenen Dimensionen liefern.
- **Internet**: Erstellt E-Mail-Adressen, Benutzernamen, Domainnamen, Domainendungen, Portnummern, IP-Adressen, Kennwörter und vieles mehr.
- **Lorem**: Bietet einige Zufallsdaten rund um die Erstellung von Texten. Hierbei können Sie einzelne Wörter, Absätze, Buchstaben, Sätze et cetera generieren lassen – allerdings nur auf Lateinisch!
- **Name**: Liefert Vornamen, Nachnamen, komplette Namen, aber auch Berufsbezeichnungen
- **Phone**: Liefert Telefonnummern
- **Rant**: Liefert Reviews, allerdings nur auf Englisch
- **System**: Liefert Dateinamen, Verzeichnisse (nur im Unix-Format), Dateiendungen, Versionsnummern, Texte von Ausnahmen, Fahrzeugnummern
- **Vehicle**: Liefert Fahrzeughersteller, -modelle und -typen, VINs (Fahrzeugnummern) und Kraftstoffarten
- **Random**: Liefert Zufallswerte für die verschiedenen Datentypen und in den angegebenen Bereichen

Zufallsdaten mit der Faker-Klasse

Mit den in den Klassen enthaltenen Funktionen können Sie die verschiedensten Daten ermitteln. Um dies auszuprobieren, erstellen Sie einfach eine neues Objekt auf Basis der Klasse **Bogus.Faker** und greifen dann direkt auf die Klassen aus der obigen Auflistung und die darin enthaltenen Funktionen zu.

Im folgenden Beispiel schreiben wir den Code direkt in die **Main**-Klasse der LINQPad-Datei:

```
Sub Main
    Dim objFaker As New Bogus.Faker
    Debug.Print(objFaker.Name.FirstName)
End Sub
```

Dies gibt einfach einen zufällig gewählten Nachnamen im Direktbereich aus. Auf die gleiche Weise probieren Sie auch die anderen Funktionen dieser und anderer Klassen aus.

Adressdaten für Deutschland

Beim Ausprobieren von Funktionen wie **Address.StreetAddress**, **Address.ZipCode** oder **Address.City** werden Sie feststellen, dass die Daten sich auf englische Adressen beziehen.

Sie können allerdings leicht auf deutsche Anschriften umschwenken, indem Sie beim Erstellen der **Faker**-Klasse den Wert **"de"** als Parameter übergeben. Die folgenden Zeilen liefern beispielsweise direkt deutsche Namen und Adressdaten:

```
Dim objFaker As New Bogus.Faker("de")
Debug.Print(objFaker.Name.FirstName)
Debug.Print(objFaker.Address.StreetAddress)
Debug.Print(objFaker.Address.ZipCode)
Debug.Print(objFaker.Address.City)
```

Kunden anlegen

Nachdem Sie aus dem Beitrag **Datenzugriff mit .NET, LINQPad und LINQ to DB** (**www.access-im-unternehmen.de/1358**) wissen, wie Sie neue Datensätze mit den gewünschten Feldwerten zu einer Tabelle wie **tblKunden** hinzufügen, haben Sie nun prinzipiell alle Techniken zum Erstellen von auf Zufallsdaten basierenden Datenbankinhalten zur Hand.

Sie brauchen die Anweisungen zum Füllen der Felder nur noch mit den oben vorgestellten Funktionen der verschiedenen Klassen des **Bogus.Faker**-Objekts zu füllen.

Dann müssten wir allerdings für jeden Kunden ein neues **Bogus.Faker**-Objekt erstellen. Das brauchen wir jedoch nur ein einziges Mal zu tun, denn die **Bogus.Faker**-Klasse bietet eine Methode namens **Generate** an. Mit der können Sie, wenn Sie für den Parameter einen Zahlenwert angeben, eigentlich sogar gleich mehrere Objekte gleichzeitig erzeugen, aber diese können wir im aktuellen Kontext nicht verarbeiten. Also nutzen wir die **Generate**-Methode nur zum Erstellen eines Kunden gleichzeitig.

Bevor wir mit dem eigentlichen Code zum Erstellen einsteigen, deklarieren wir noch eine **Enum**-Auflistung für das Geschlecht – wozu wir diese benötigen, erläutern wir anschließend:

```
Public Enum Gender
    Male = 0
    Female = 1
End Enum
```

Die Prozedur **KundenAnlegen** erwartet die Anzahl der anzulegenden Kunden als Parameter. Dann deklariert sie die benötigten Variablen:

```
Public Sub KundenAnlegen(intMenge As Int32)
    Dim Beispielkunde As New tblKunden
    Dim objUserQuery As UserQuery
    Dim i As Int32
    Dim intGender As int32
```

Sie erstellt dann ein **UserQuery**-Objekt auf Basis der Tabelle **tblKunden** und ein neues **Bogus.Faker**-Objekt, dem Sie die Klasse **tblKunden** als Typ zuweisen.

Außerdem fügen wir noch den Wert **"de"** als Parameter an, damit Bogus deutsche Daten liefert:

```
    objUserQuery = tblKunden.DataContext
    Dim objFaker As New Bogus.Faker(Of tblKunden)("de")
```

Dann folgt ein Teil, von dem nur die inneren Zeilen interessant sind – und mit dem wir die Regeln festlegen, nach denen wir neue Objekte des Typs **tblKunden** füllen.

Wichtig ist nur, dass Sie wissen, dass der Buchstabe **k** für das **tblKunden**-Objekt steht und **f** für das **Bogus.Faker**-

E-Mails mit Anlagen mit Outlook versenden

Das Versenden von E-Mails haben wir bereits ausführlich in Access im Unternehmen beschrieben. Dort kam auch gelegentlich das Thema auf, wie Sie Dateien an solche E-Mails anhängen. Allerdings gibt es bei genauerem Hinsehen Anforderungen, die wir noch nicht behandelt haben – zum Beispiel das Anhängen vieler Dateien auf einen Rutsch oder auch das Hinzufügen von Dateien, die nicht aus einem Verzeichnis stammen. Hierzu sind dann mehrere Aufrufe des jeweiligen Dialogs zum Auswählen der Dateien erforderlich. Auf diese Spezialfälle gehen wir im vorliegenden Beitrag ein.

Anforderungen an die Beispiellösung

Dieser Beitrag soll eine Beispiellösung mit folgenden Funktionen beschreiben:

- Versenden einer E-Mail unter Angabe von Empfänger, Betreff und Inhalt
- Hinzufügen von Anlagen über einen Dateiauswahldialog. Dabei sollen mehrere Anlagen aus einem Verzeichnis gleichzeitig selektiert werden können und beim Hinzufügen weiterer Anlagen aus einem weiteren Verzeichnis sollen die bereits hinzugefügten Anlagen erhalten bleiben.
- Bereits hinzugefügte Anlagen sollen markiert und aus der E-Mail gelöscht werden können.

Vorbereitungen

Wir benötigen Verweise auf zwei VBA-Bibliotheken. Diese fügen Sie dem VBA-Projekt über den **Verweise**-Dialog hinzu, den Sie mit dem Menübefehl **Extras|Verweise** des VBA-Editors öffnen:

- **Microsoft Office x.0 Object LIbrary**: Liefert das **FileDialog**-Objekt, mit dem wir den Dialog zum Auswählen der Anlagen anzeigen.
- **Microsoft Outlook x.0 Object Library**: Stellt das **MailItem**-Objekt zum Versenden der E-Mail bereit.

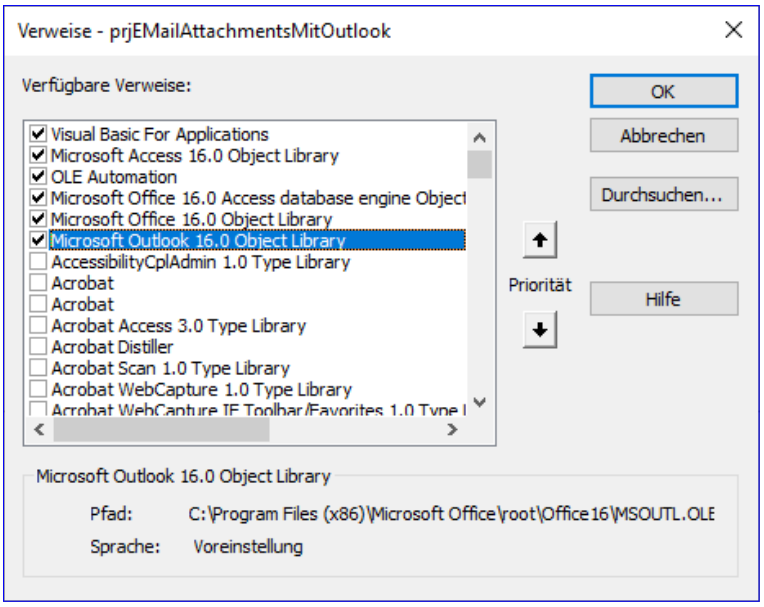


Bild 1: Verweise des VBA-Projekts

Mit den beiden hinzugefügten Verweisen sieht der Verweise-Dialog wie in Bild 1 aus.

Formular zum Erstellen der E-Mail mit Anlage

Das Formular, in dem wir das Versenden von E-Mails mit Anlagen steuern wollen, heißt **frmMailsMitAttachment** und sieht in der Entwurfsansicht wie in Bild 2 aus.

Das Formular verwendet die folgenden Steuerelemente:

- **txtAn**: Dient der Erfassung des Empfängers der E-Mail.
- **txtBetreff**: Nimmt den Betreff der E-Mail auf.

- **txtInhalt**: Hier können Sie den Inhalt der E-Mail eingeben.

- **lstAnlagen**: Listenfeld zur Anzeige der bereits hinzugefügten Anlagen. Das Listenfeld soll die Mehrfachauswahl erlauben, damit auch mehrere Anlagen gleichzeitig aus der Liste entfernt werden können. Dazu legen wir für die Eigenschaft **Mehrfachauswahl** den Wert **Erweitert** fest. Außerdem stellen wir für dieses Listenfeld die Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** auf **Beide** ein, damit das Listenfeld mit dem Formular vergrößert werden kann. Damit das dazugehörige Bezeichnungsfeld an Ort und Stelle bleibt, erhalten **Horizontaler Anker** und **Vertikaler Anker** hier die Werte **Links** und **Oben**. Damit wir die ausgewählten Dateien einzeln hinzufügen können, stellen wir außerdem die Eigenschaft **Herkunftstyp** auf **Wertliste** ein.

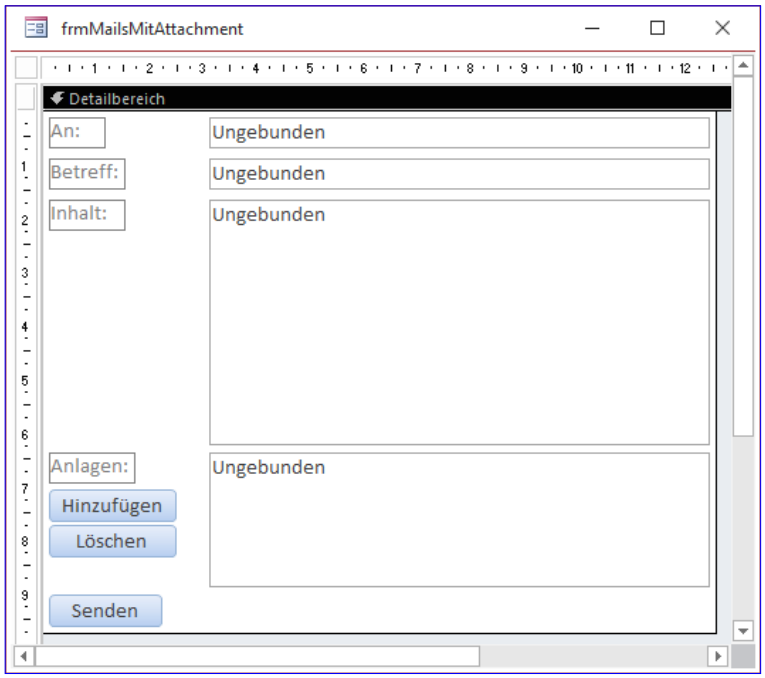


Bild 2: Formular in der Entwurfsansicht

- **cmdHinzufuegen**: Öffnet einen Dateiauswahldialog zum Auswählen der hinzuzufügenden Anlagen und fügt diese nach der Auswahl zum Listenfeld **lstAnlagen** hinzu.

- **cmdLoeschen**: Löscht die aktuell markierten Einträge des Listenfeldes. Die Schaltfläche soll nur markiert sein, wenn im Listenfeld **lstAnlagen** mindestens ein Eintrag markiert ist.

- **cmdSenden**: Sendet die E-Mail mit den Angaben der Felder **txtAn**, **txtBetreff** und **txtInhalt** und mit den im Listenfeld **lstAnlagen** angegebenen Dateien. Für diese Schaltfläche stellen wir die Eigenschaft **Vertikaler Anker** auf **Unten** ein, damit es beim Vergrößern des Formulars unten verankert ist.

Die Eigenschaften **Datensatzmarkierer**, **Navigationsschaltflächen**, **Bildlaufleisten** und **Trennlinien** sollen den Wert **Nein** und die Eigenschaft **Automatisch zentrieren** den Wert **Ja** haben.

Anlagen zur Liste hinzufügen

Die Schaltfläche **cmdHinzufuegen** soll einen Dateiauswahldialog öffnen, mit dem der Benutzer eine oder mehrere Dateien aus einem Verzeichnis auswählen und zur Liste hinzufügen kann. Die Liste zeigt die Dateien dann mit dem vollständigen Pfad an. Dabei wollen wir das mehrfache Hinzufügen von Dateien unterbinden. Dies wäre möglich, da wir erlauben wollen, dass der Benutzer durch mehrfaches Anklicken der **Hinzufügen**-Schaltfläche auch mehrfach Dateien aus dem gleichen Verzeichnis hinzufügen könnte.

Also prüfen wir vor dem Hinzufügen einer Datei zum Listenfeld **lstAnlagen**, ob diese bereits in der Liste enthalten ist.

Die durch die Schaltfläche **cmdHinzufuegen** ausgelöste Prozedur finden Sie in Listing 1. Hier deklarieren wir zwei Laufvariablen sowie das **FileDialog**-Objekt und

eine **Boolean**-Variable. Die Variable **objFileDialog** füllen wir mit einem Objekt der Klasse **FileDialog** mit dem Typ **msoFileDialogPicker** – mehr Informationen über diese Klasse finden Sie im Beitrag **Dateien und Verzeichnisse auswählen mit FileDialog** (**www.access-im-unternehmen.de/1345**).

Wir stellen mit der Eigenschaft **AllowMultiSelect** ein, dass der Benutzer mehrere Dateien gleichzeitig auswählen kann. Mit **Title** legen wir den Titel fest, dann leeren wir mit **Filters.Clear** die vorhandenen Filter und fügen mit **Add** den Filter für alle Dateitypen hinzu. Die Bezeichnung der Schaltfläche zum Übernehmen der Auswahl stellen wir auf **ButtonName** ein.

Danach folgt der Aufruf des Dialogs mit der **Show**-Methode, was wie in Bild 3 aussieht. Nach der Auswahl mit der Schaltfläche **Hinzufügen** wird der **If**-Teil der **If...Then**-Ab-

```
Private Sub cmdHinzufuegen_Click()
    Dim objFileDialog As FileDialog
    Dim l As Long
    Dim m As Long
    Dim bolVorhanden As Boolean
    Set objFileDialog = FileDialog(msoFileDialogFilePicker)
    With objFileDialog
        .AllowMultiSelect = True
        .Title = "Anlagen auswählen"
        .Filters.Clear
        .Filters.Add "Alle Dateien", "*.*"
        .ButtonName = "Hinzufügen"
        If .Show = True Then
            For l = 1 To .SelectedItems.Count
                If Not Len(Me!lstAnlagen.RowSource + .SelectedItems(l)) > 32750 Then
                    bolVorhanden = False
                    For m = 1 To Me!lstAnlagen.ListCount
                        If Me!lstAnlagen.ItemData(m) = .SelectedItems(l) Then
                            bolVorhanden = True
                            Exit For
                        End If
                    Next m
                    If Not bolVorhanden Then
                        Me!lstAnlagen.AddItem .SelectedItems(l)
                    End If
                Else
                    MsgBox "Es können keine weiteren Dateien zur Liste hinzugefügt werden."
                    Exit Sub
                End If

            Next l
        End If
    End With
End Sub
```

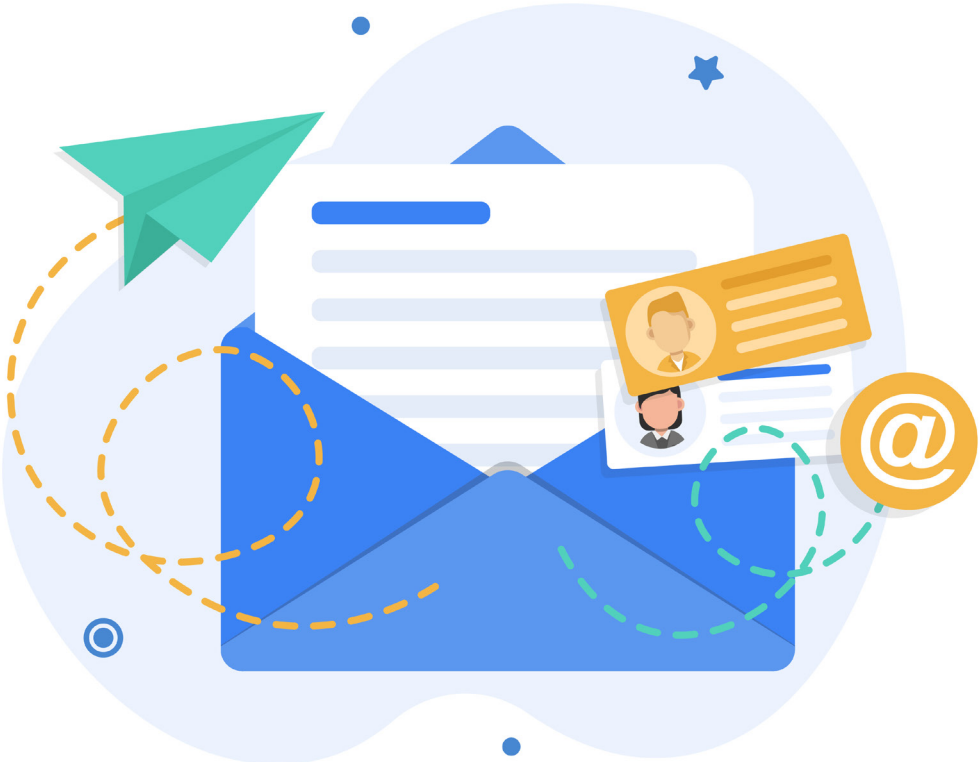
Listing 1: Prozedur zum Auswählen von Dateien

HAUFE.
Ausgabe 03/2022

ACCESS
IM UNTERNEHMEN

E-MAILS MIT CDO VERSENDEN

Nutzen Sie eine alternative Technik abseits von Outlook, um Mails direkt aus der Datenbank zu senden (ab Seite 43).



In diesem Heft:

LÖSCHEN FÜR PROFIS

Lernen Sie die Abläufe beim Löschen von Daten in Formularen genau kennen und nutzen Sie die Ereignisse für Ihre Zwecke.

SEITE 36

CODE BEIM ÖFFNEN UND SCHLIESSEN AUSFÜHREN

Lernen Sie diverse Methoden zum automatischen Ausführen von Code beim Öffnen oder Schließen der Datenbank.

SEITE 5

ZULETZT GENUTZTE DATENSÄTZE ANZEIGEN

Nutzen Sie ein Listenfeld oder ein Ribbonmenü, um die zuletzt verwendeten Datensätze schnell verfügbar zu machen.

SEITE 14

E-Mails mit CDO verschicken

Wenn Sie aus der Access-Anwendung heraus E-Mails verschicken, nutzen Sie üblicherweise die Automation von Outlook. Allerdings ist Outlook nicht immer verfügbar. Gibt es denn keine andere Möglichkeit – außer ein paar veralteten SMTP-DLLs, die unter Visual Studio 6 erstellt wurden und mittlerweile nicht mehr die Sicherheitsstandards erfüllen und schon gar nicht unter der 64-Bit-Version von Access arbeiten? Doch: Es gibt die gute, alte CDO-Bibliothek. Was das ist, erfahren Sie in dieser Ausgabe.

Die CDO-Bibliothek ist standardmäßig in Windows 10 und auch im neuen Windows 11 enthalten und funktioniert in beiden Systemen reibungslos. Wir widmen dem Thema gleich drei Beiträge.

Der erste Beitrag heißt **E-Mails versenden mit CDO** (ab Seite 43) und zeigt, wie Sie einfache E-Mails mit der CDO-Bibliothek verschicken können. Hier lernen Sie auch die Grundlagen dieser Bibliothek und erfahren, wie Sie moderne Verschlüsselungstechniken für den Mailversand unterstützen können.

Im zweiten Beitrag zum Thema namens **Serienmails versenden mit CDO** erfahren Sie ab Seite 50, wie Sie SMTP-Konfigurationen für unterschiedliche Absenderkonten einrichten können und damit Serienmails an Empfänger beispielsweise aus einer Kundentabelle verschicken können. Dabei schauen wir uns auch an, wie Sie die E-Mail-Adressen der Empfänger aus der jeweiligen Tabelle auslesen.

In den bisherigen Beiträgen zum Thema CDO haben wir nur mit VBA-Code gearbeitet, im dritten mit dem Namen **Benutzeroberfläche für CDO-Serienmails** erfahren Sie, wie Sie eine praktische Benutzerüberfläche für die bisher erlernten Techniken hinzufügen können. Damit verwalten Sie die Konfigurationen, die Mailings und auch noch die einzelnen Adressaten der Mailings per Formular – dies alles ab Seite 59.

Wenn Sie schon immer einmal wissen wollten, wie Sie beim Start einer Access-Datenbank nicht nur ein Formular anzeigen, sondern sogar noch VBA-Code ausführen können, haben Sie ebenfalls gleich drei Beiträge für Sie:

- Der Beitrag **Code beim Öffnen der Anwendung: AutoExec** zeigt, wie Sie VBA-Code per Autostart-Makro aufrufen können (ab Seite 5).
- Unter **Code beim Öffnen der Anwendung: Formular** lernen Sie, wie Sie das Gleiche durch ein automatisch geöffnetes Formular erreichen können (ab Seite 7).
- Und **Code beim Öffnen der Anwendung: Ribbon** zeigt eine sicher für die meisten Entwickler neue Möglichkeit, gleich beim Öffnen Code auszuführen - mehr dazu ab Seite 9.

Und falls Sie auch noch wissen möchten, wie Sie Code beim Schließen der Anwendung ausführen – alles dazu lesen Sie unter **Code beim Schließen der Anwendung ausführen** ab Seite 11.

Spannend sind auch die beiden Beiträge zum Thema **Anzeige zuletzt verwendeter Datensätze**. Diese lassen sich etwa im Ribbon auflisten oder auch direkt in einem Listenfeld – und bieten so die Möglichkeit, die Datensätze schnell wieder anzuzeigen. Mehr dazu lesen Sie unter **Zuletzt verwendete Datensätze im Ribbon** (ab Seite 14) und **Zuletzt verwendete Datensätze per Listenfeld** (ab Seite 23).

Und nun wie immer: Viel Spaß beim Ausprobieren!

Ihr André Minhorst

Emulation im Webbrowser-Steuerelement einstellen

Wenn Sie das Webbrowser-Steuerelement nutzen, wird üblicherweise der Internet Explorer emuliert. Für viele Anwendungen ist das jedoch wenig hilfreich, denn sie funktionieren nur mit neueren Versionen des Internet Explorers oder auch nur noch mit Microsoft Edge. Dieser Beitrag zeigt, wie Sie einstellen, welche Version des Microsoft-Browsers im Webbrowser-Steuerelement verwendet wird – und Sie erfahren auch, wie Sie herausfinden, welchen Browser das Steuerelement gerade anzeigt.

Vorbereitung

Bevor wir uns anschauen, wie Sie das **Webbrowser**-Steuerelement zum Emulieren einer der Versionen des Internet Explorers oder von Microsoft Edge bewegen, legen wir als Erstes in einer Beispieldatenbank ein neues Formular namens **frmWebbrowser** an und fügen diesem ein **Webbrowser**-Steuerelement namens **ctlWebbrowser** hinzu. Um den Aufbau so praktisch wie möglich zu gestalten und unsere Versuche schnell durchführen zu können, wollen wir dafür sorgen, dass das **Webbrowser**-Steuerelement immer die folgende Webseite anzeigt:

```
https://www.whatsmybrowser.org/
```

Diese Seite liefert uns einen Hinweis darauf, welcher Browser im Webbrowser gerade emuliert wird. Damit diese direkt beim Öffnen des Formulars mit dem Webbrowser erscheint, fügen wir dem Formular außerdem ein Textfeld namens **txtURL** hinzu.

Dieses erhält die folgende Ereignisprozedur für die Ereigniseigenschaft **AfterUpdate**:

```
Private Sub txtURL_AfterUpdate()
    Me!ctlWebbrowser.ControlSource = "=""" & Me!txtURL _
        & """"
End Sub
```

Diese stellt die Eigenschaft **Steuerelementinhalt** auf die im Textfeld **txtURL** angegebene Internetadresse ein.

Nun wollen wir noch beim Laden die Adresse **https://www.whatsmybrowser.org/** einstellen und öffnen. Dazu fügen wir für die Ereigniseigenschaft **Beim Laden** noch die folgende Prozedur hinzu, welche die Adresse in das Textfeld **txtURL** schreibt und die Prozedur **txtURL_AfterUpdate** aufruft:

```
Private Sub Form_Load()
    Me!txtURL = "https://www.whatsmybrowser.org/"
    txtURL_AfterUpdate
End Sub
```

Damit das Formular direkt beim Öffnen der Datenbankdatei angezeigt wird, stellen wir in den Access-Optionen noch die Option **Formular anzeigen** im Bereich **Aktuelle Datenbank** unter **Anwendungsoptionen** auf den Namen des Formulars ein, hier **frmWebbrowser**.

Öffnen Sie die Anwendung nun, erscheint direkt das Formular und zeigt die oben genannte Webseite an.

Anzeige des verwendeten Browsers

Dies sorgt bei den Standardeinstellungen von Windows üblicherweise für die Anzeige aus Bild 1. Das ist natürlich nicht das, was wir uns wünschen – wir hätten lieber aktuellere Funktionen wie etwa von Microsoft Edge.

Anderen Browser emulieren

Wie also erreichen wir die Emulation eines anderen Browsers? Dazu sind folgende Schritte nötig:

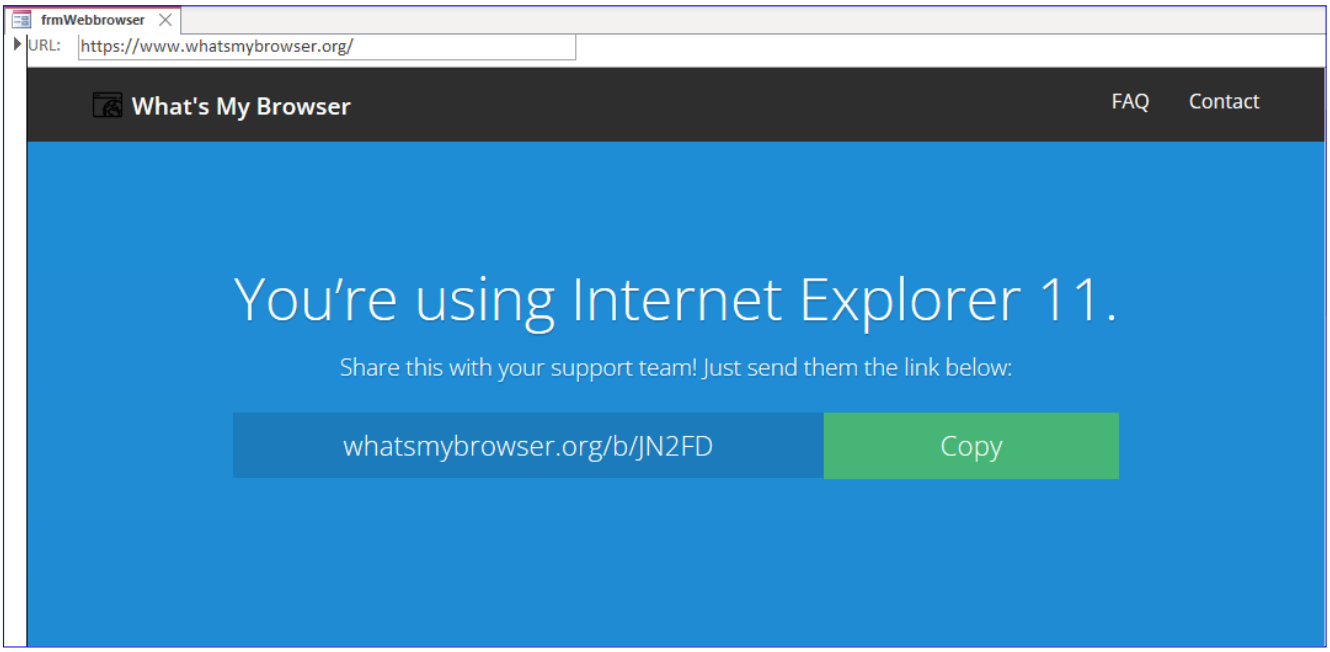


Bild 1: Anzeige des verwendeten beziehungsweise emulierten Browsers im **Webbrowser**-Steuerelement

- Alle (!) Instanzen von **MsAccess.exe** schließen. Also auch die, die gegebenenfalls unsichtbar im Hintergrund laufen. Ob das der Fall ist, erfahren Sie im Task Manager von Windows.

- In der Windows Registry einen bestimmten Wert einstellen.

- Die Beispieldatenbank erneut öffnen und schauen, ob sich der für die Emulation verwendete Browser geändert hat.

Im Detail sieht es so aus, dass Sie herausfinden müssen, welches Registry-Element den notwendigen Wert enthält. Warum das? Weil es unter Umständen mehrere gibt, welche diesen möglicherweise enthalten – mehr dazu weiter unten.

Richtigen Registry-Eintrag finden

Der Registry-Eintrag, der für die gewählte Emulation im **Webbrowser**-Steuerelement verantwortlich ist, befindet sich in einem neuen Windows 11-System an der folgenden Stelle:

```
Computer\HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\Main\FeatureControl\FEATURE_BROWSER_EMULATION
```

Um diesen anzupassen, öffnen Sie zunächst die Registry. Dazu geben Sie in die Windows-Suche den Suchbegriff **RegEdit** ein und betätigen die Eingabetaste. Es erscheint der Dialog **Registrierungs-Editor**.

Hier können Sie direkt den obigen Pfad in das Textfeld im oberen Bereich eingeben und diesen mit der Eingabetaste übernehmen. Es erscheint dann das Element mit den entsprechenden Registry-Einträgen (siehe Bild 2).

Hier befindet sich ein Registry-Eintrag namens **msaccess.exe**, welcher den dezimalen Wert **11.001** enthält. Um diesen Wert zu ändern, klicken Sie doppelt auf den Eintrag **MsAccess.exe**. Es erscheint ein neuer Dialog, in dem Sie zunächst die Anzeige des Wertes von **Hexadezimal** auf **Dezimal** ändern. Dann geben Sie dort einmal den Wert **99.999** ein (siehe Bild 3).

Wenn Sie das Eingabefenster für den Wert nun schließen, können Sie die Access-Anwendung erneut öffnen. Wir

Code beim Öffnen der Anwendung: AutoExec

Wenn Sie Code beim Öffnen einer Access-Datenbank ausführen wollen, gibt es zwar keine direkte Möglichkeit wie etwa ein Ereignis beim Öffnen eines Formulars. Es gibt allerdings verschiedene Techniken, mit denen Sie dennoch den Zeitpunkt des Öffnens der Anwendung abfangen und dabei VBA-Code ausführen können. Bisher nutzte man hier vornehmlich die Möglichkeiten über das Makro namens AutoExec und das Startformular, aber es gibt noch eine weitere Möglichkeiten: nämlich über ein benutzerdefiniertes Ribbon. Im vorliegenden Beitrag schauen wir uns das AutoExec-Makro an und zeigen, wie Sie damit VBA-Code aufrufen können.

Microsoft Access bietet zwei Möglichkeiten an, um Abläufe zu programmieren und auf Ereignisse zu reagieren. Die erste und wohl eher von Einsteigern genutzte Technik sind die sogenannten Makros. Diese sind ein eigener Objekttyp und nicht mit den Makros unter Word und Excel zu verwechseln: Unter Word und Excel heißen nämlich VBA-Prozeduren, die das Wiederholen von bestimmten Abläufen ermöglichen und die auch aufgezeichnet werden können, ebenfalls Makros.

Die zweite Möglichkeit, unter Microsoft Access Abläufe zu programmieren, ist VBA. Diese Programmiersprache ist zwar komplizierter, aber wesentlich flexibler als die Makros. Wenn Sie beim Start einer Access-Anwendung jedoch VBA-Code ausführen wollen, gibt es keinen eigens dafür vorgesehenen Weg. Stattdessen ist einer der Wege eben eines der zuvor beschriebenen Makros.

Genau genommen benötigen Sie sogar ein spezielles Makro dafür, das sich durch seinen Namen von anderen Makros unterscheidet: Es muss auf jeden Fall **AutoExec** heißen. Access prüft nämlich beim Öffnen einer Datenbank, ob diese ein Makro namens **AutoExec** enthält. Falls ja, wird dieses gleich beim Starten ausgeführt. Nun haben wir ja schon beschrieben, dass die Programmierung mit Makros an sich weniger Möglichkeiten bietet und so möchten Sie vielleicht auch schon beim Start der Anwendung Prozeduren aufrufen, die in der Programmiersprache VBA geschrieben sind.

Das ist kein Problem, denn Makros bieten eine ganze Reihe verschiedener Befehle an, unter anderem auch einen, mit dem Sie eine VBA-Funktion aufrufen können. Sie finden hier also die perfekte Symbiose aus einem Makro und einer VBA-Routine.

Beim Start auszuführende Funktion definieren

Bevor wir das Makro anlegen, das beim Öffnen der Anwendung gestartet werden und unsere VBA-Funktion ausführen soll, legen wir erst einmal die VBA-Funktion an. In unserem Fall wollen wir einfach eine Meldung ausgeben, die angibt, dass sie von einer VBA-Funktion aufgerufen wurde.

In einer neuen, leeren Access-Datenbank nutzen Sie den Ribbonbefehl **Erstellen|Makros und Code|Modul**, um ein neues Standardmodul anzulegen und im VBA-Editor zu öffnen. Hier fügen wir nun die folgende Funktion ein:

```
Public Function Startup()
    MsgBox "Meldung per VBA"
End Function
```

Diese Funktion können Sie durch Platzieren der Einfügemarke innerhalb des Funktionscodes und Betätigen der Taste **F5** starten und sich von der Funktion überzeugen. Sollte dies nicht klappen, ist die Anwendung vielleicht noch nicht als vertrauenswürdig eingestuft – das müssten Sie dann prüfen und korrigieren.

Code beim Öffnen der Anwendung: Formular

Wenn Sie Code beim Öffnen einer Access-Datenbank ausführen wollen, gibt es zwar keine direkte Möglichkeit wie etwa ein Ereignis beim Öffnen eines Formulars. Es gibt allerdings verschiedene Techniken, mit denen Sie dennoch den Zeitpunkt des Öffnens der Anwendung abfangen und dabei VBA-Code ausführen können. Bisher nutzte man hier vornehmlich die Möglichkeiten über das Makro namens AutoExec und das Startformular, aber es gibt noch eine weitere Option: nämlich über ein benutzerdefiniertes Ribbon. Im vorliegenden Beitrag zeigen wir zunächst, wie Sie über das als Startformular definierte Formular VBA-Code ausführen können.

Neben dem **AutoExec**-Makro, dessen Einsatz wir im Beitrag **Code beim Öffnen der Anwendung: AutoExec** (**www.access-im-unternehmen.de/1367**) beschreiben, gibt es mit dem Startformular noch eine weitere Option.

Dabei gehen wir in dem oben angegebenen Beitrag davon aus, dass wie eine Funktion wie die folgende beim Start der Anwendung aufrufen wollen:

```
Public Function Startup()
    MsgBox "Meldung per VBA"
End Function
```

Statt der **MsgBox**-Anweisung, die hier angegeben ist, können Sie die beim Start Ihrer Anwendung notwendigen Anweisungen einfügen.

Startformular erstellen

Wie aber können wir diese VBA-Funktion nun mithilfe eines Formulars aufrufen? Dazu erstellen wir als Erstes einmal das benötigte Formular und speichern es unter dem Namen **frmStartup**.

Dann legen Sie für das Ereignis **Bei Laden** des Formulars den Wert **[Ereignisprozedur]** fest und klicken auf die Schaltfläche mit den drei Punkten (**...**), um die Ereignisprozedur im Klassenmodul des Formulars anzulegen (siehe Bild 1).

Im VBA-Editor erscheint die Prozedur, die wir wie folgt ergänzen:

```
Private Sub Form_Load()
    Call Startup
End Sub
```

Diese Prozedur ruft nun beim Laden des Formulars unsere Startfunktion auf. Das testen wir, indem Sie das Formular in der Formularansicht anzeigen. Dies sollte zuerst die

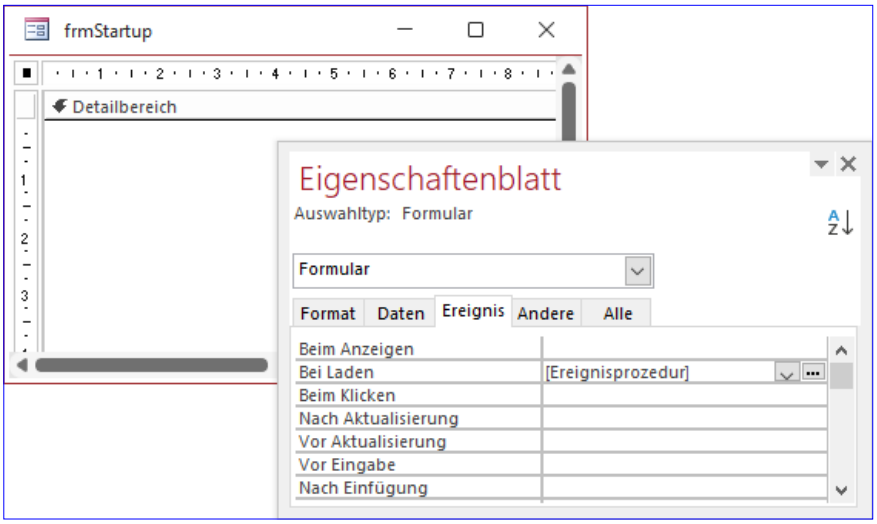


Bild 1: Das Startformular

Code beim Öffnen der Anwendung: Ribbon

Wenn Sie Code beim Öffnen einer Access-Datenbank ausführen wollen, gibt es zwar keine direkte Möglichkeit wie etwa ein Ereignis beim Öffnen eines Formulars. Es gibt allerdings verschiedene Techniken, mit denen Sie dennoch den Zeitpunkt des Öffnens der Anwendung abfangen und dabei VBA-Code ausführen können. Bisher nutzte man hier vornehmlich die Möglichkeiten über das Makro namens AutoExec und das Startformular, aber es gibt noch eine weitere Option: nämlich über ein benutzerdefiniertes Ribbon. Im vorliegenden Beitrag zeigen wir, wie Sie eine VBA-Funktion beim Starten unter Verwendung des Ribbons aufrufen können.

Neben dem **AutoExec**-Makro, dessen Einsatz wir im Beitrag **Code beim Öffnen der Anwendung: AutoExec** (**www.access-im-unternehmen.de/1367**) beschreiben, gibt es mit dem Startformular noch eine weitere Option – siehe **Code beim Öffnen der Anwendung: Formular** (**www.access-im-unternehmen.de/1368**).

Es gibt seit der Einführung des Ribbons jedoch noch eine weitere Möglichkeit, gleich beim Öffnen der Datenbank VBA-Code auszuführen. Diese hier ist zwar etwas aufwendiger, aber wenn Sie ohnehin ein Ribbon beim Start der Datenbankanwendung anzeigen, ist nur eine einzige zusätzliche Zeile zum Aufrufen der Startfunktion nötig. Außerdem sparen Sie so ein **AutoExec**-Makro beziehungsweise ein beim Starten der Anwendung zu öffnendes Formular ein.

Startfunktion

Wir gehen in den beiden oben angegebenen Beiträgen davon aus, dass wir eine Funktion wie die folgende beim Start der Anwendung aufrufen wollen:

```
Public Function Startup()
    MsgBox "Meldung per VBA"
End Function
```

Um diese automatisch beim Start aufzurufen und dabei ein Ribbon statt des **AutoExec**-Makros oder eines Startformulars zu nutzen, benötigen Sie zunächst eine minimale Ribbondefinition. Diese sieht wie folgt aus:

```
?xml version="1.0"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" onLoad="OnLoad_Startup"/>
```

Diese Definition fügen Sie einer Tabelle namens **USysRibbons** hinzu, deren Entwurf wie in Bild 1 aussieht. Für das Feld **RibbonName** stellen wir einen eindeutigen Index ein. Wenn Sie diese Tabelle so angelegt und unter dem Namen **USysRibbons** gespeichert haben, wird diese möglicherweise nicht im Navigationsbereich angezeigt – das liegt daran, dass die Tabelle mit diesem Namen als Systemtabelle erkannt und ausgeblendet wird. Sie können diese dann entweder mit dem folgenden Befehl öffnen oder die Anzeige der Systemobjekte in den Optionen aktivieren:

```
DoCmd.OpenTable "USysRibbons"
```

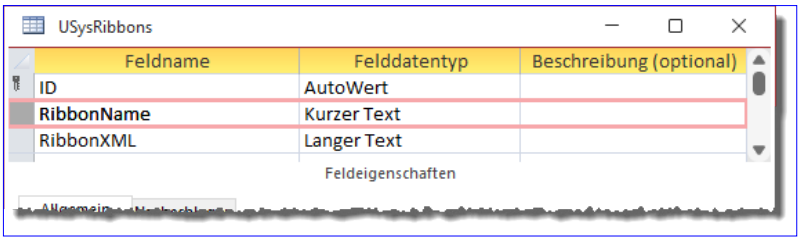


Bild 1: Entwurf der Ribbontabelle **USysRibbons**

Code beim Schließen der Anwendung ausführen

Beim Öffnen einer Access-Anwendung haben Sie verschiedene Möglichkeiten, Code auszuführen – Sie können ein Formular anzeigen, das beim Öffnen Code ausführt, das AutoExec-Makro nutzen, um eine Prozedur aufzurufen oder sogar das Ribbon dafür instrumentalisieren. Wenn es jedoch um Code geht, der beim Beenden der Anwendung ausgeführt werden soll, finden wir keine offiziell dafür vorgesehene Technik. Allerdings gibt es einen Trick, um doch noch VBA-Code auszuführen, wenn die Access-Anwendung durch den Benutzer geschlossen wird.

Code beim Starten der Anwendung ausführen

Wie Sie Code ausführen, wenn der Benutzer die Datenbank öffnet, haben wir bereits in drei Beiträgen gezeigt. Das gelingt auf folgende Arten:

- Mit dem Autostart-Makro: **Code beim Öffnen der Anwendung: AutoExec** (**www.access-im-unternehmen.de/1367**)
- Mit dem Startformular: **Code beim Öffnen der Anwendung: Formular** (**www.access-im-unternehmen.de/1368**)
- Und es gelingt sogar von einem Ribbon aus, das beim Start angezeigt wird: **Code beim Öffnen der Anwendung: Ribbon** (**www.access-im-unternehmen.de/1369**)

Code beim Schließen der Anwendung ausführen

Wenn es um das Ausführen von Code beim Schließen einer Anwendung geht, finden wir allerdings wesentlich weniger Möglichkeiten.

Auch hier können wir schon vorher sagen: Es gibt kein spezielles Ereignis etwa für die Datenbank, das Sie nutzen können, um Code beim Schließen der Datenbank auszuführen.

Stattdessen behelfen wir uns mit einem Trick. Dieser sieht kurzgefasst folgendes vor:

- Wir öffnen beim Starten ein Formular, das wir aber gleich wieder ausblenden.
- Diesem Formular fügen wir eine Ereignisprozedur für das Ereignis **Beim Entladen** hinzu.
- Wenn der Benutzer nun die Anwendung schließen will, muss zuvor das noch offene, aber ausgeblendete Formular geschlossen werden.
- Das Schließen des Formulars erfolgt automatisch, wenn die Datenbank geschlossen wird, was nach sich zieht, dass auch die **Beim Entladen**-Prozedur des Formulars noch ausgeführt wird.

Beim Schließen-Code Schritt für Schritt

Schauen wir uns im Detail an, was nötig ist, um beim Schließen der Datenbank VBA-Code auszuführen.

Die erste Aktion ist das Erstellen eines Formulars, das direkt beim Start der Anwendung geöffnet und ausgeblendet wird. Dieses Formular nennen wir **frmStart** und es enthält keinerlei Elemente – das ist auch nicht nötig, da der Benutzer dieses nie zu Gesicht bekommt.

Die erste Frage ist: Wie sorgen wir dafür, dass das Formular direkt nach dem Start ausgeblendet wird? Im Gegensatz zu Steuerelementen finden wir in den Formulareigenschaften keine Eigenschaft namens **Sichtbar**, also müssen wir das Formular per VBA ausblenden. Das

erledigen wir so früh wie möglich, damit das Formular noch nicht einmal aufblitzt. Welche ist die erste Ereignisprozedur, wo wir diesen Schritt durchführen können?

Um das herauszufinden, haben wir für die in Frage kommenden Ereigniseigenschaften jeweils eine leere Ereignisprozedur angelegt und diese mit einem Haltepunkt versehen. Das Ergebnis ist, dass das Ereignis **Beim Öffnen** als erstes ausgelöst wird (siehe Bild 1).

Also fügen wir hier die Anweisung zum Ausblenden des Formulars ein, was wie folgt aussieht:

```
Private Sub Form_Open(Cancel As Integer)
    Me.Visible = False
End Sub
```

Allerdings gelingt das nicht auf diese Weise, und auch nicht durch Einfügen der Prozedur in einer der übrigen im Bild zu findenden Ereignisprozeduren. Der Grund ist einfach und wir bestätigen diesen, indem wir uns jeweils den Wert der Eigenschaft **Visible** ausgeben lassen – zum Beispiel so:

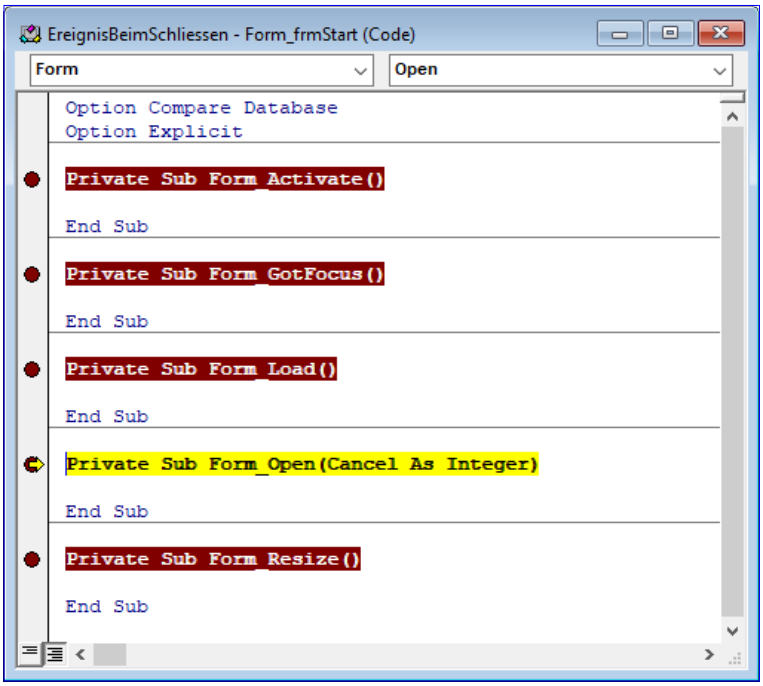


Bild 1: Ermitteln der ersten Prozedur beim Öffnen eines Formulars

```
Private Sub Form_Load()
    Debug.Print "Load: " & Me.Visible
End Sub
```

Hiermit stellen wir schnell fest, dass die **Visible**-Eigenschaft hier ohnehin noch den Wert **False** aufweist, das Formular also noch gar nicht sichtbar ist. Eine Idee wäre noch, das **Timer**-Ereignis zu nutzen, aber bis dieses erstmalig aufgerufen wird, wurde auch das Formular bereits einmal eingeblendet.

Wir können das Formular also für diesen Zweck nicht für die Option **Startformular** in den Optionen der Datenbank festlegen.

Öffnen des Formulars im ausgeblendeten Modus

Wenn wir das Formular schon nicht direkt beim Öffnen unsichtbar machen können, dann müssen wir einen anderen Weg finden, das Formular direkt unsichtbar zu machen. Die einfachste Variante ist, die **DoCmd.OpenForm**-Methode mit einem speziellen Parameter zu nutzen. Dieser heißt **WindowMode**.

Normalerweise nutzen wir diesen mit dem Wert **acDialog**, um das Formular als modalen Dialog zu öffnen. Nun aber wollen wir das Formular ausgeblendet öffnen und dazu nutzen wir den Wert **acHidden**. Der Aufruf sieht wie folgt aus:

```
DoCmd.OpenForm "frmStart", WindowMode:=acHidden
```

Wenn Sie diesen Aufruf im Direktbereich absetzen, haben Sie ein Problem: Wie finden Sie nun heraus, ob das Formular geöffnet wurde? Das fragen wir über die Auflistung **Forms** ab. Diese sollte für die Eigenschaft **Count** den Wert **1** liefern, sofern keine anderen Formulare geöffnet sind. Und das ist hier der Fall:

```
? Forms.Count
 1
```

Zuletzt verwendete Datensätze im Ribbon

Im Beitrag "Zuletzt verwendete Datensätze per Listenfeld" zeige wir, wie Sie die zuletzt in einem Formular angezeigten Kundendatensätze in einem Listenfeld aufführen können, um diese schnell wieder zu öffnen. Vielleicht haben Sie im Formular zur Kundenverwaltung aber keinen Platz für diese Liste oder Sie möchten diese einfach immer verfügbar haben. Dann bietet sich das Ribbon als Ort für diese Liste an. Im vorliegenden Beitrag zeigen wir, wie Sie das Ribbon um ein Steuerelement zur Anzeige und Auswahl der zuletzt verwendeten Datensätze erweitern.

Als Formular zur Anzeige der Kundendaten verwenden wir das gleiche Formular, das wir im Artikel **Zuletzt verwendete Datensätze per Listenfeld** (**www.access-im-unternehmen.de/1365**) erstellt haben. Auf diese Weise können wir gleich abgleichen, ob das Ribbonsteuerelement die richtigen Datensätze anzeigt. Das Formular sieht wie in Bild 1 aus.

Welches Ribbonsteuerelement für die zuletzt verwendeten Datensätze?

Als Erstes stellt sich die Frage, welches der Ribbonsteuerelemente am besten für unsere Aufgabe geeignet ist. Eigentlich sind das alle Steuerelemente, die Listen von Einträgen anzeigen können, und dafür kommen die folgenden in Frage:

- **comboBox**
- **dropDown**
- **menu**
- **dynamicMenu**
- **gallery**
- **splitButton**

Und Sie können sogar, wenn Sie eine begrenzte Menge von zuletzt verwendeten Datensätzen anzeigen wollen, für jeden eine Schaltfläche anzeigen und so alle interessanten Datensätze gleichzeitig bereitstellen.

Wichtig ist nur, dass wir die Anzeige mit jedem neu im Formular verwendeten Datensatz aktualisieren, und das ist mit all diesen Elementen möglich. Also schauen wir uns anhand verschiedener Beispielen an, wie das gelingt.

comboBox oder dropDown?

Die beiden Steuerelemente **comboBox** und **dropDown** haben verschiedene Eigenschaften, die sie für verschiedene Aufgaben prädestinieren. Wenn es darum geht, nicht nur den im jeweiligen Element ausgewählten Text zu ermitteln, sondern auch noch einen weiteren Wert wie beispielsweise einen Index oder eine Id für diesen Wert, dann ist das **dropDown**-Element die erste Wahl.

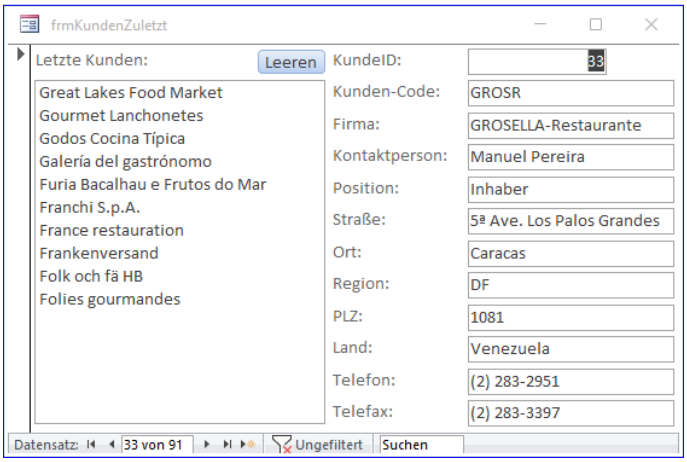


Bild 1: Die in diesem Formular zuletzt angezeigten Datensätze sollen auch per Ribbon angezeigt werden.

Also verwenden wir dieses als Erstes. Bevor wir es programmieren, benötigen wir jedoch noch eine Datensatzherkunft für die anzuzeigenden Elemente.

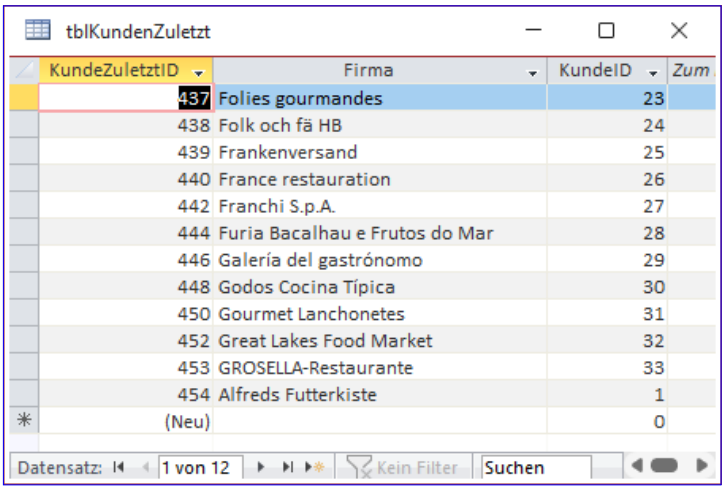


Bild 2: Tabelle mit den zuletzt angezeigten Datensätzen

Abfrage für die zuletzt angezeigten Datensätze

In der oben beschriebenen Lösung haben wir alle Datensätze der Tabelle **tblKunden**, die im Formular **frmKundenZuletzt** angezeigt wurden, in die Tabelle **tblKundenZuletzt** eingetragen. Diese sah dann nach dem Durchlaufen einiger Datensätze wie in Bild 2 aus.

Das Listenfeld, das die zehn zuletzt verwendeten Datensätze angezeigt hat, nutzt eine Abfrage, um diese in der richtigen Reihenfolge aus dieser Tabelle auszulesen, also in umgekehrter Reihenfolge zum Anlegezeitpunkt.

Außerdem hatte diese Abfrage ein Kriterium, das dafür gesorgt hat, dass der aktuell im Formular angezeigte Datensatz nicht in der Liste erscheint, denn dieser Datensatz ist ja schon sichtbar.

Diese Abfrage, die direkt für das Listenfeld hinterlegt war, kopieren wir nun in eine neue Abfrage namens **qryKundeZuletzt** (siehe Bild 3). Danach entfernen wir das dort markierte Kriterium.

Warum das? Weil es sein kann, dass das Ribbon auch angezeigt wird, obwohl das Formular gar nicht sichtbar ist.

Dann ist es natürlich wichtig, dass auch der zuletzt angezeigte Eintrag zur Auswahl steht.

Anlegen des Ribbons zur Anzeige des dropDowns

Um das Ribbon anzulegen, sind einige Schritte notwendig:

- Hinzufügen einer Tabelle namens **USysRibbons**
- Eintragen der Ribbondefinition in diese Tabelle
- Hinzufügen der Callbackprozeduren für die Funktionalität des Ribbons
- Festlegen der Ribbondefinition als Startribbon

Ribbontabelle mit Inhalt anlegen

Die Ribbontabelle ist eine Tabelle namens **USysRibbons**. Sie enthält die drei Felder **RibbonID** (Primärschlüsselfeld

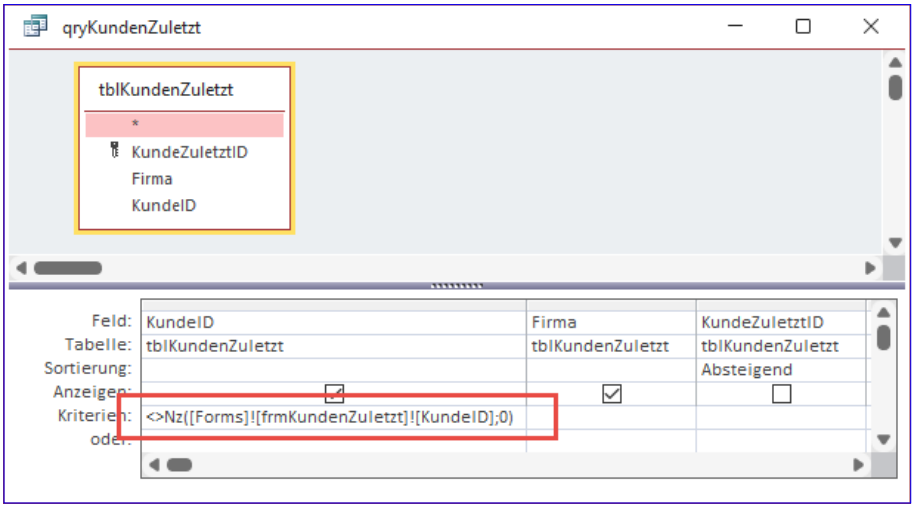


Bild 3: Abfrage zum Ermitteln der Datensätze in der richtigen Reihenfolge

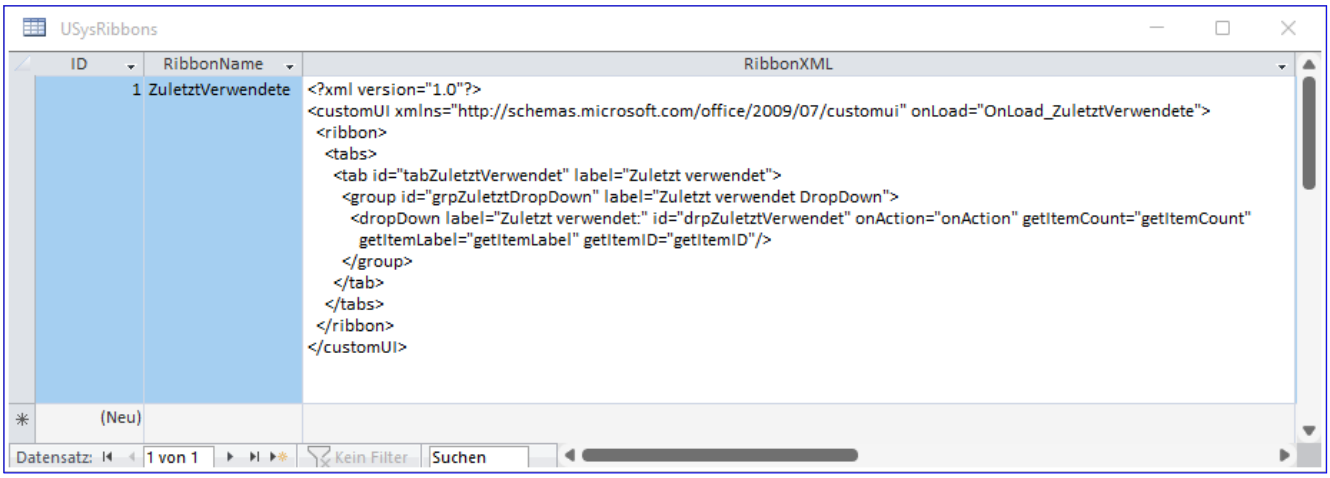


Bild 4: Die Tabelle **USysRibbons** mit der Ribbondefinition für die Anzeige der zuletzt angezeigten Datensätze im **dropDown**-Element

mit Autowert), **RibbonName** (Datentyp **Kurzer Text**) und **RibbonXML** (Datentyp **Langer Text**).

Ribbondefinition für das dropDown-Element erstellen

Die Ribbondefinition fügen wir in einen neuen Datensatz dieser Tabelle ein, für den wir außerdem den Wert **ZuletztVerwendete** im Feld **RibbonName** festlegen. Die Tabelle sieht dann wie in Bild 4 aus.

Die Ribbondefinition sehen Sie im Detail in Listing 1. Die meisten Elemente dienen der Herstellung der Struktur aus **tab**- und **group**-Element. Interessant ist das **dropDown**-Element mit den angegebenen Ereignisattributen. Hier legen wir die Prozeduren fest, die zum Füllen des **dropDown**-Elements nötig sind.

Ribbon-Verweis beim Laden speichern

Beim erstmaligen Anzeigen des Ribbons können wir das Ereignisattribut **onLoad** nutzen, um eine Prozedur anzugeben, die beim Laden des Ribbons ausgelöst wird. Hier geben wir den Wert **onLoad_ZuletztVerwendete** an.

Den Verweis auf die Ribbondefinition wollen wir in einer Variablen namens **objRibbon_ZuletztVerwendete** mit dem Datentyp **IRibbonUI** speichern. Dieser Datentyp ist

```
<?xml version="1.0"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" onLoad="OnLoad_ZuletztVerwendete">
  <ribbon>
    <tabs>
      <tab id="tabZuletztVerwendet" label="Zuletzt verwendet">
        <group id="grpZuletztDropDown" label="Zuletzt verwendet DropDown">
          <dropDown label="Zuletzt verwendet:" id="drpZuletztVerwendet" onAction="onAction" getItemCount="getItemCount"
            getItemLabel="getItemLabel" getItemID="getItemID"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Listing 1: Die Definition des **dropDown**-Elements zur Anzeige der zuletzt verwendeten Datensätze

in der Bibliothek **Microsoft Office x.0 Object Library** enthalten, auf die wir daher einen Verweis zum VBA-Projekt hinzufügen. Dazu wählen Sie im VBA-Editor den Menübefehl **Extras|Verweise** aus. Hier fügen Sie einen Verweis auf die genannte Bibliothek hinzu, sodass der Dialog anschließend wie in Bild 5 aussieht.

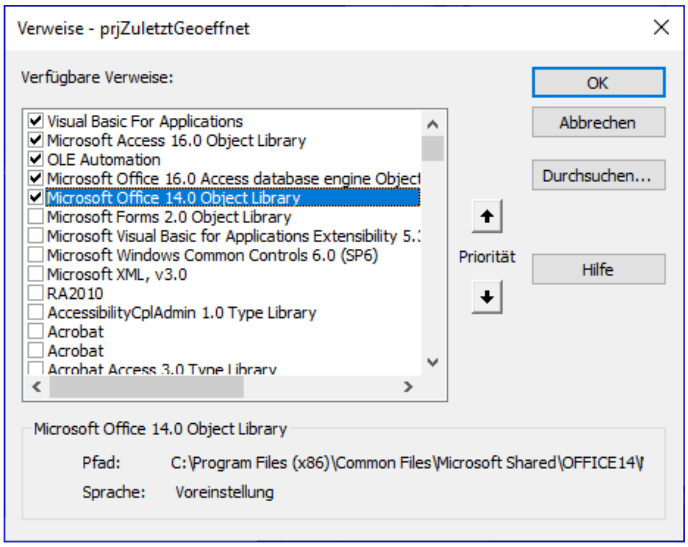


Bild 5: Hinzufügen eines Verweises auf die **Office**-Bibliothek

Nun können wir die Variable zum Referenzieren der Ribbondefinition zu einem neuen, leeren Modul namens **mdlRibbons** hinzufügen:

```
Public objRibbon_ZuletztVerwendete As IRibbonUI
```

Die beim Laden des Ribbons ausgeführte und in der Ribbondefinition für das Attribut **onLoad** des Elements **customUI** angegebene Prozedur definieren wir wie folgt:

```
Sub onLoad_ZuletztVerwendete(ribbon As IRibbonUI)
    Set objRibbon_ZuletztVerwendete = ribbon
End Sub
```

Die Prozedur liefert mit dem Parameter **ribbon** einen Verweis auf die Ribbondefinition, welche die Prozedur in **objRibbon_ZuletztVerwendete** speichert.

Hinzufügen der Callbackprozeduren für die Funktionalität des Ribbons

Nun wird es interessant, denn wir wollen das **dropDown**-Element mit den Daten aus der Abfrage **qryKundenZuletzt** füllen. Dazu haben wir im **dropDown**-Element drei Ereignisattribute mit entsprechenden VBA-Prozeduren definiert. In der ersten zählen wir die einzulesenden Elemente und tragen diese in ein Array ein, in der zweiten und dritten, die jeweils einmal für die ermittelte Anzahl aufgerufen werden, lesen wir die Daten aus dem Array aus und fügen die IDs und die Texte zum **dropDown**-Element hinzu. Als Erstes benötigen wir eine Variable, in der wir die anzuzeigenden Informationen speichern:

```
Private strZuletzt() As String
```

Danach folgt die Callbackprozedur, welche die Elemente zählt und im Array speichert. Diese füllt eine Recordset-Variable mit den Daten aus der Abfrage **qryKundenZuletzt** und durchläuft die einzelnen Datensätze in einer **Do While**-Schleife.

Dabei zählt sie die Elemente in der Variablen **i** und trägt gleichzeitig die Inhalte der Felder **KundeID** und **Firma** in das Array **strZuletzt** ein:

```
Sub getItemCount(control As IRibbonControl, ByRef count)
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim i As Integer
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM ➥
                        qryKundenZuletzt", dbOpenDynaset)
    Do While Not rst.EOF
        ReDim Preserve strZuletzt(1, i) As String
        strZuletzt(0, i) = rst!KundeID
        strZuletzt(1, i) = rst!Firma
        i = i + 1
        rst.MoveNext
    Loop
    count = i
End Sub
```

Die beiden folgenden Prozeduren werden danach für die mit **count** zurückgegebene Anzahl jeweils einmal aufgerufen. Die erste Prozedur **getItemID** liefert mit **index** den Index des mit **id** zurückzugebenden ID-Wertes, den wir aus der ersten Spalte des Arrays auslesen:

```
Sub getItemID(control As IRibbonControl, _
                              index As Integer, ByRef id)
    id = strZuletzt(0, index)
End Sub
```

Die zweite Prozedur holt aus der zweiten Spalte des Arrays den anzuzeigenden Text für das Element:

```
Sub getItemLabel(control As IRibbonControl, _
                              index As Integer, ByRef label)
    label = strZuletzt(1, index)
End Sub
```

Festlegen der Ribbondefinition als Anwendungsribbon

Damit wird es Zeit für einen ersten Test. Also wählen wir das soeben definierte Ribbon als Anwendungsribbon aus. Dazu müssen wir die Datenbank zunächst schließen und erneut starten. Danach steht die Ribbondefinition unter dem Namen **ZuletztVerwendete** in der Option **Name des Menübands** in den Access-Optionen zur Verfügung, die Sie mit dem Ribboneintrag **DateiOptionen** öffnen.

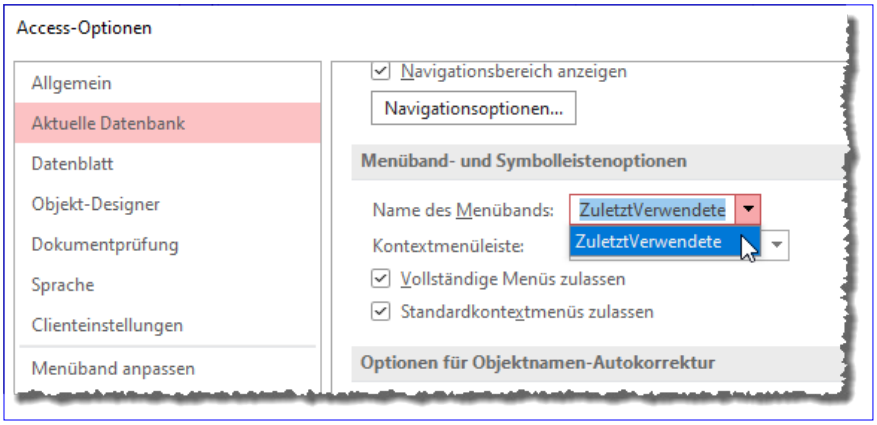


Bild 6: Einstellen des Ribbons in den Access-Optionen

Hier wechseln Sie zum Bereich **Aktuelle Datenbank** und finden unterhalb von **Menüband- und Symbolleistenoptionen** die passende Option vor (siehe Bild 6).

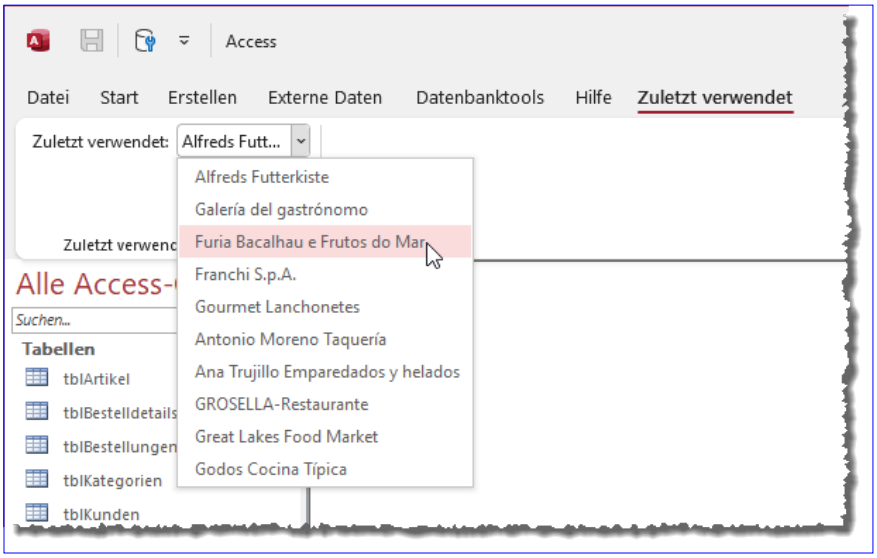


Bild 7: Das **dropDown**-Element mit den zuletzt angezeigten Kunden

Wenn Sie die Anwendung nun erneut schließen und wieder öffnen, wird das von uns definierte Ribbontab im Ribbon sichtbar. Klicken Sie dieses an und klappen dann das **dropDown**-Element aus, sieht dieses wie in Bild 7 aus.

Nun fehlen noch zwei Funktionen, und zwar zum Anzeigen des jeweiligen Kunden nach der Auswahl im **dropDown**-Element und zum Aktualisieren der Einträge im **dropDown**-Element, nachdem der Benutzer einen neuen Kunden im Formular **frmKundenZuletzt** ausgewählt hat.

Diese beiden Funktionen schauen wir uns nun an.

Zuletzt verwendete Datensätze per Listenfeld

Eine sehr praktische Funktion findet sich in den Office-Anwendungen und auch in vielen anderen Produkten. Wo auch immer Dateien verwendet werden, finden Sie beim Öffnen der jeweiligen Anwendung eine Liste der zuletzt verwendeten Dateien vor. Warum sollte man dies nicht auch in einer Access-Datenbank nutzen, um die zuletzt angezeigten Datensätze in Formularen zur Auswahl anzubieten? Wenn Sie zum Beispiel eine Bestellverwaltung nutzen, könnte es sehr sinnvoll sein, die zuletzt angezeigten oder bearbeiteten Kunden in einer Schnellauswahl zum erneuten Aufrufen vorzufinden. Wie das gelingt und welche Erweiterungen dazu notwendig sind, zeigt der vorliegende Beitrag.

Feature wie unter Office

Wenn Sie mit Anwendungen wie Word, Excel oder Access arbeiten, kennen Sie die Anzeige der zuletzt verwendeten Dateien, die beim Starten der Anwendung über die jeweilige Verknüpfung erscheint.

Hier wählen Sie immer aus den zuletzt verwendeten Dateien aus oder alternativ aus einer Liste von angehefteten Dateien, die unabhängig davon, wann sie zuletzt benutzt wurden, angezeigt werden sollen.

Vorbereitungen

Wir wollen zunächst nur den ersten Teil auch für die Datensätze einer Tabelle realisieren, in diesem Fall für die Kunden einer Bestellverwaltung.

Wir nutzen dazu wieder die Tabellen der Beispieldatenbank **Südsturm**, wobei für uns die Tabelle **tblKunden** interessant ist.

Als Basis dient das Formular **frmKundenZuletzt**, für dessen Eigenschaft **Datensatzquelle** wir die Tabelle **tblKunden** angeben.

Aus der Feldliste ziehen wir alle Felder der Tabelle in den Detailbereich des Entwurfs des Formulars (siehe Bild 1).

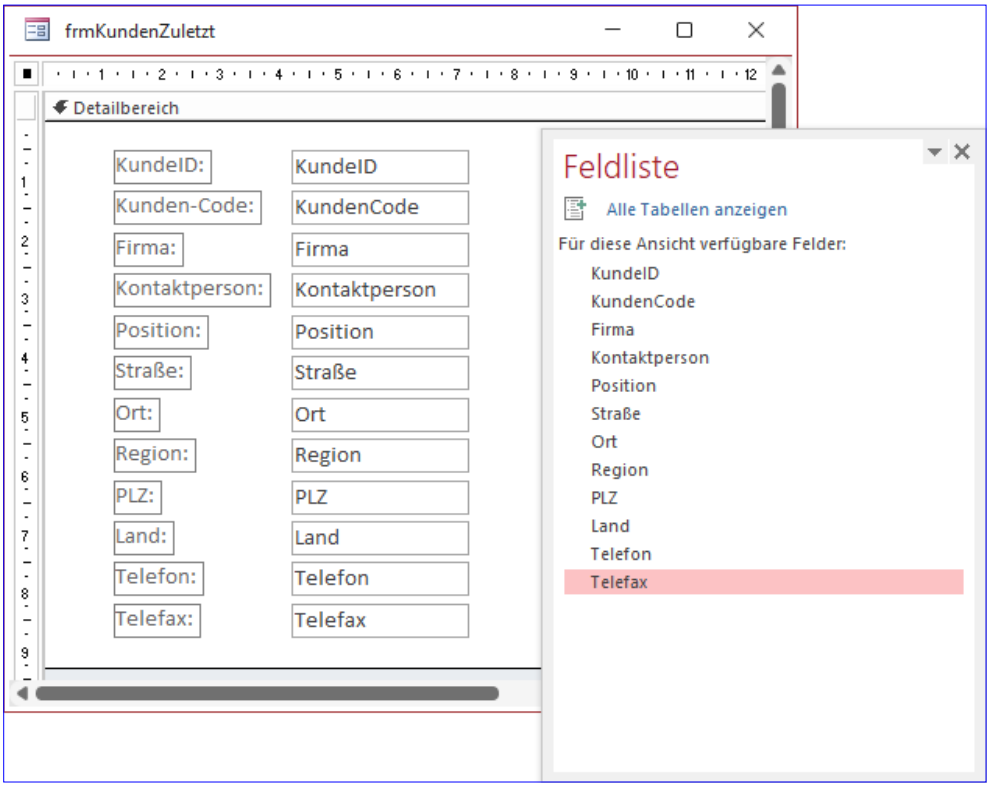


Bild 1: Dieses Formular wollen wir mit einer Liste der zuletzt verwendeten Kunden ausstatten.

Konzept für die Liste zuletzt verwendeter Datensätze

Bevor wir loslegen und dem Formular ein Listenfeld zur Anzeige der zuletzt verwendeten Datensätze hinzufügen, stellen wir ein paar grundlegende Überlegungen an.

Welche Datensätze speichern wir in der Liste der zuletzt verwendeten Datensätze?

Die erste Frage, die sich uns stellt, ist folgende: Zu welchen Datensätzen beziehungsweise wann soll ein Datensatz überhaupt gespeichert und in der Liste der zuletzt verwendeten Datensätze angezeigt werden?

Es gibt verschiedene Möglichkeiten:

- Jeder angezeigte Datensatz wird in der Liste gespeichert.
- Jeder bearbeitete Datensatz wird gespeichert.
- Jeder gezielt angezeigte Datensatz, beispielsweise durch das Anzeigen von einem Übersichtsformular aus, wird gespeichert.
- Nur Datensätze, die explizit gespeichert werden sollen – beispielsweise über einen Klick auf eine entsprechende Schaltfläche – werden in die Liste übernommen.

Wir wollen im ersten Entwurf den ersten Ansatz umsetzen und alle Datensätze in der Liste der zuletzt angezeigten Datensätze speichern, die überhaupt im Formular **frmKundenZuletzt** angezeigt wurden.

Wo sollen die Informationen über die zuletzt verwendeten Datensätze gespeichert werden?

Die zweite Frage lautet: Wo wollen wir überhaupt die Information speichern, welche Datensätze der Tabelle **tblKunden** zuletzt verwendet wurden?

Hier gibt es wiederum mindestens zwei Möglichkeiten:

- In einer eigenen Tabelle, die wir beispielsweise **tblKundenZuletzt** nennen. Hier speichern wir Informationen wie den Namen und/oder den Primärschlüsselwert des jeweiligen Datensatzes. Die Liste zur Anzeige der zuletzt verwendeten Datensätze kann ihre Daten direkt aus dieser Tabelle beziehen.
- Direkt in der Tabelle **tblKunden**. Wir könnten ein eigenes Feld mit dem Datentyp **Datum** beispielsweise namens **ZuletztAngezeigt** anlegen, das wir immer aktualisieren, wenn der Datensatz angezeigt wurde. Die Liste soll dann die X zuletzt geöffneten Datensätze anhand dieses Feldes ermitteln.

Hier wählen wir ebenfalls den ersten Ansatz. Dies ist praktikabler, weil wir dann nicht den Entwurf der Tabelle **tblKunden** anpassen müssen. Gegebenenfalls handelt es sich bei dieser Tabelle um eine per ODBC eingebundene SQL Server-Tabelle und es ist gar nicht möglich, diese Tabelle anzupassen. Außerdem spricht für die Speicherung in einer eigenen Tabelle, dass diese im Frontend gespeichert werden kann und somit die zuletzt verwendeten Datensätze für den jeweiligen Benutzer sichert.

Anlegen der Tabelle zum Speichern der zuletzt angezeigten Datensätze

Die Tabelle wollen wir **tblKundenZuletzt** nennen. Wir könnten auch den Namen **tblZuletztGeoeffnet** oder ähnlich verwenden, aber vielleicht wollen wir die gleiche Funktion auch noch für die Artikel hinzufügen – daher ist es sinnvoller, den Namen der betroffenen Elemente direkt im Tabellennamen anzugeben.

Welche Felder soll die Tabelle für die zuletzt angezeigten Datensätze enthalten?

Nun wird es interessanter. Wenn es nur um die Anzeige der zuletzt verwendeten Einträge der Tabelle **tblKunden** geht, könnte man neben dem Primärschlüsselfeld nur noch ein Feld mit der Bezeichnung des jeweiligen Elements, in diesem Falle der Kunden, anlegen. Allerdings wollen Sie den jeweiligen Kunden ja auch über das Listen-

feld öffnen, und dazu müssen wir genau wissen, um welchen Datensatz es sich handelt. Also benötigen wir auch noch den Primärschlüsselwert aus der Tabelle **tblKunden**.

Wenn wir aber den Primärschlüsselwert und auch den Namen des Kunden aus der Tabelle **tblKunden** in der Tabelle **tblKundenZuletzt** speichern, laufen wir Gefahr, inkonsistente Daten anzuzeigen. Es kann nämlich sein, dass wir im Formular **frmKundenLetzte** den Namen eines Kunden ändern, nachdem wir diesen angezeigt haben.

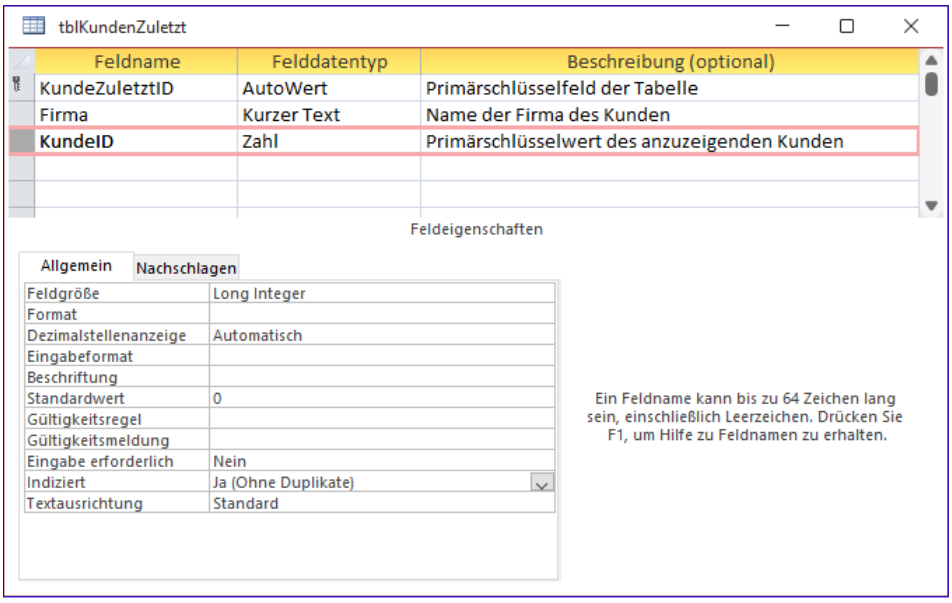


Bild 2: Entwurf der Tabelle **tblKundenZuletzt**

Dem können wir allerdings vorbeugen, indem wir im Formular **frmKundenLetzte** nicht nur eine Ereignisprozedur anlegen, die beim Anzeigen eines Kunden ausgelöst wird und dessen Daten in der Tabelle **tblKundenZuletzt** speichert, sondern auch noch eine, die beim Ändern eines Datensatzes ausgelöst wird und den aktuellen Kunden dann in der Tabelle **tblKundenZuletzt** überschreibt.

Und bei der Gelegenheit sollten wir auch im Hinterkopf behalten, dass der Benutzer ja auch einmal einen Kundendatensatz löschen kann. Dieser soll dann selbstverständlich auch aus der Tabelle **tblKundenZuletzt** entfernt werden.

Entwurf der Tabelle tblKundenZuletzt

Also gestalten wir die Tabelle **tblKundenZuletzt** wie in Bild 2. Neben dem Primärschlüsselfeld legen wir ein Feld an, das den gleichen Namen erhält wie das Feld, aus dem die in der Liste der zuletzt geöffneten Einträge anzuzeigenden Werte stammen. Für das Feld **KundeID**, welches den Primärschlüsselwert der anzuzeigenden Einträge aus der Tabelle **tblKunden** aufnehmen soll, legen wir einen eindeutigen Index fest. Dieser sorgt dafür, dass jeder Datensatz der Tabelle **tblKunden** nur einmal in der Tabelle **tblKundenZuletzt** referenziert werden kann. Später nutzen wir das Auslösen eines Fehlers beim Hinzufügen eines Datensatzes mit einem bereits vorhandenem Wert in diesem Feld für unsere Zwecke – mehr dazu weiter unten.

Benötigen wir kein Feld in tblKundenZuletzt, das die Reihenfolge angibt?

Wir könnten die Funktion mit noch mehr Features ausstatten und beispielsweise die Zeit anzeigen, wann der Datensatz zuletzt aufgerufen wurde. Fürs erste reicht es uns jedoch, einfach nur die zuletzt verwendeten Kunden so anzuzeigen, dass der zuletzt verwendete Kunde ganz oben erscheint. Benötigen wir dazu ein spezielles Feld? Nein, wir können dazu das Primärschlüsselfeld **KundeZuletztID** nutzen und einfach in absteigender Reihenfolge nach dem Inhalt dieses Feldes sortieren.

Prozedur erstellen, die den aktuellen Kunden zur Tabelle hinzufügt

Nachdem wir diese Tabelle erstellt haben, wollen wir dem Formular **frmKundeZuletzt** zuerst einmal eine Ereignisprozedur hinzufügen, die bei Anzeigen eines Kunden ausgelöst wird und welche die Daten für diesen Kunden in der

```
Private Sub Form_Current()
    Dim db As DAO.Database
    Dim strINSERT As String
    Dim strDELETE As String
    If Not IsNull(Me!KundeID) Then
        Set db = CurrentDb
        On Error Resume Next
        strINSERT = "INSERT INTO tblKundenZuletzt(Firma, KundeID) VALUES('" & Replace(Me!Firma, "'", "''") & "', " _
            & Me!KundeID & ")"
        db.Execute strINSERT, dbFailOnError
        Select Case Err.Number
            Case Is = 3022
                strDELETE = "DELETE FROM tblKundenZuletzt WHERE KundeID = " & Me!KundeID
                db.Execute strDELETE, dbFailOnError
                db.Execute strINSERT, dbFailOnError
            Case 0
            Case Else
                MsgBox "Fehler " & Err.Number & vbCrLf & vbCrLf & Err.Description
        End Select
    End If
End Sub
```

Listing 1: Prozedur zum Speichern der Daten für den aktuellen Datensatz in der Tabelle **tblKundenZuletzt**

Tabelle **tblKundenZuletzt** speichert. Erst danach fügen wir das Listenfeld mit den Daten der Tabelle **tblKunden-Zuletzt** hinzu.

Das passende Ereignis für unsere Zwecke heißt **Beim Anzeigen**. Es wird sowohl beim Anzeigen des ersten Datensatzes beim Öffnen des Formulars als auch beim Wechseln zu einem anderen Datensatz ausgelöst.

Diese Prozedur füllen wir mit dem Code aus Listing 1. Die Prozedur prüft zuerst, ob **KundeID** nicht den Wert **Null** hat, was der Fall ist, wenn der Datensatzzeiger auf einen neuen, leeren Datensatz verschoben wird. Diesen Fall müssten wir später berücksichtigen, wenn wir uns um das Speichern beim Ändern eines Datensatzes kümmern.

Danach deaktivieren wir die eingebaute Fehlerbehandlung mit **On Error Resume Next**. Das ist nötig, damit wir versuchen können, den Datensatz mit dem entsprechenden Wert im Feld **KundeID** zur Tabelle **tblKundenZuletzt** hinzuzufügen. Ist bereits ein Datensatz mit diesem Wert für das Feld **KundeID** vorhanden, wird nämlich der Fehler mit der Nummer **3022** ausgelöst. In diesem Fall wollen wir den vorhandenen Datensatz mit dem Wert aus **KundeID** aus der Tabelle **tblKundenZuletzt** löschen und diesen erneut anlegen. Warum belassen wir diesen dann nicht einfach in der Tabelle? Weil der Datensatz nun der zuletzt verwendeten Datensatz ist und da wir absteigend nach dem Inhalt des Feldes **KundeZuletztID** sortieren, müssen wir den Datensatz löschen und wieder anlegen, um dieses entsprechend zu aktualisieren.

In dem Fall, dass der Kunde noch nicht in der Liste enthalten ist, fügen wir diesen einfach mit der Anweisung aus **strINSERT** zur Tabelle **tblKundenZuletzt** hinzu. Für einen Beispieldatensatz sieht diese Anweisung etwa wie folgt aus:

```
INSERT INTO tblKundenZuletzt(Firma, KundeID) VALUES('Centro comercial Moctezuma', 13)
```

Falls tatsächlich der Fehler **3022** auftritt, den wir in einer **Select Case**-Bedingung über den Wert von **Err.Number** prüfen, stellen wir eine **DELETE**-Anweisung zusammen, um den betroffenen Datensatz zu löschen und diesen anschließend mit der Anweisung aus **strINSERT** erneut anzulegen. Die **DELETE**-Anweisung sieht für diese Fall wie folgt aus:

```
DELETE FROM tblKundenZuletzt WHERE KundeID = 13
```

Die **Select Case**-Bedingung enthält noch zwei weitere Zweige. Der mit dem Wert **0** ist für einen fehlerfreien Aufruf der **INSERT INTO**-Anweisung vorgesehen und erledigt nichts. Der **Else**-Zweig zeigt für alle anderen Fehler eine entsprechende Meldung an. Nachdem wir in die Formularansicht gewechselt sind und einige Datensätze durchlaufen haben – inklusive eines Sprungs zum Datensatz mit der Nummer **6** am Ende – sieht die Tabelle **tblKundenZuletzt** wie in Bild 3 aus.

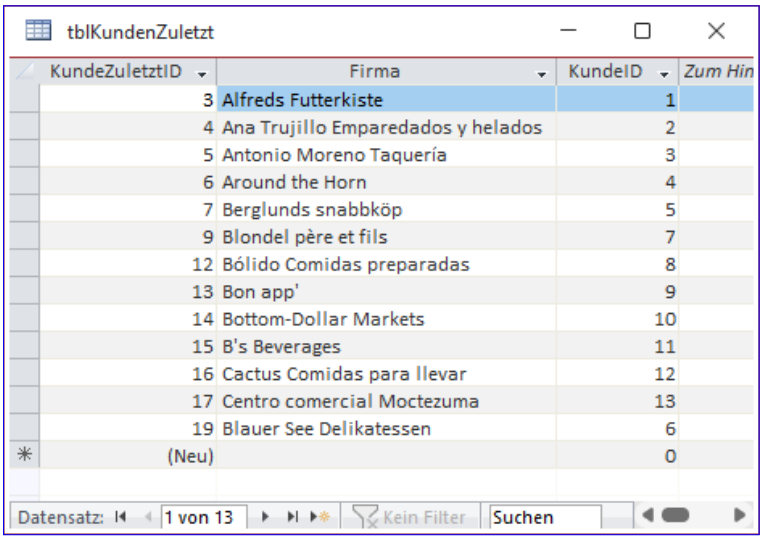


Bild 3: Einige zur Tabelle **tblKundenZuletzt** hinzugefügte Datensätze

Zuletzt verwendete Datensätze in Listenfeld anzeigen

Nun fügen wir dem Formular **frmKundenZuletzt** auf der linken Seite ein Listenfeld namens **lstKundenZuletzt** hinzu. Zuvor verschieben wir die Textfelder des Formulars weiter nach rechts, damit genug Platz für das Listenfeld vorhanden ist. Das Ergebnis sehen Sie in Bild 4.

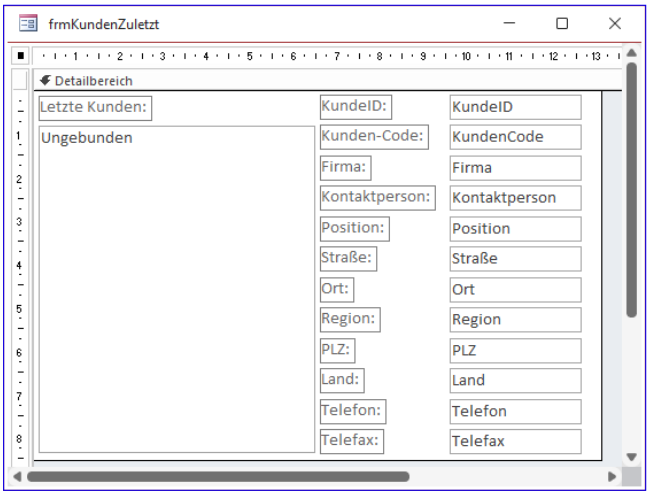


Bild 4: Entwurf des Formulars **frmKundenZuletzt** mit Listenfeld

Das Listenfeld füllen wir über die Eigenschaft **Datensatzherkunft** mit einer Abfrage, die auf der Tabelle **tblKundenZuletzt** basiert. Warum benötigen wir dazu eine Abfrage? Weil wir die Datensätze absteigend nach dem Primärschlüsselfeld **KundeZuletztID** sortieren wollen. Die Abfrage sieht in der Entwurfsansicht wie in Bild 5 aus.

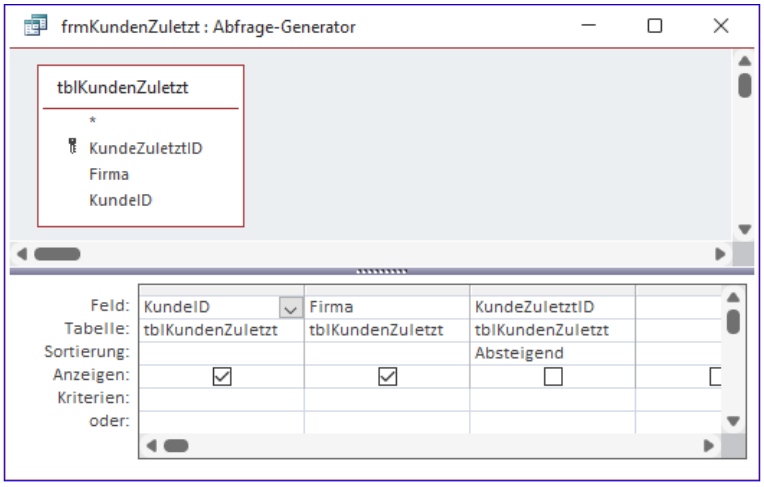


Bild 5: Abfrage mit den Daten für das Listenfeld **lstKundenZuletzt**

Sie enthält alle Felder der Tabelle **tblKundenZuletzt** und sortiert die Datensätze absteigend

Löschereignisse und -optionen im Zusammenspiel

Formulare haben gleich drei Ereignisse, die sich rund um das Thema Löschen drehen. Welche davon ausgelöst werden, hängt auch von der Einstellung einer Access-Option ab. Um sicherzugehen, dass Aktionen, die nach dem Löschen eines Datensatzes über das Formular ausgeführt werden sollen, tatsächlich stattfinden, müssen Sie einige Dinge beachten. Dieser Beitrag stellt die drei betroffenen Ereignisprozeduren vor, erläutert die Access-Option, die sich auf die Ausführung dieser Ereignisprozeduren auswirkt und zeigt, wie Sie das alles so zusammenbringen, dass die gewünschten Folgeaktionen zuverlässig ausgeführt werden.

Formularereignisse

Formulare bieten so einige Ereignisse, die durch verschiedene Aktionen des Benutzers ausgelöst werden. Dazu gehören das Öffnen und Laden des Formulars sowie das Schließen und Entladen, außerdem gibt es Ereignisse, die beim Anzeigen, vor der Aktualisierung oder nach der Aktualisierung eines Datensatzes feuern.

Löschen-Ereignisse

In manchen Fällen kann es auch wichtig sein, die Ereignisse beim Löschen eines Datensatzes abzufangen. Und derer gibt es gleich drei.

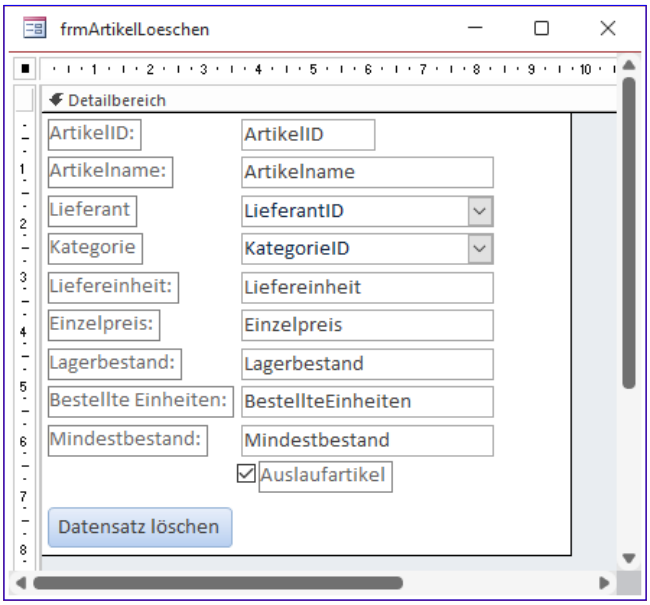


Bild 1: Beispielformular mit Löschen-Schaltfläche

Diese heißen folgendermaßen:

- **Beim Löschen**
- **Vor Löschbestätigung**
- **Nach Löschbestätigung**

Beispielformular erstellen

Um mit diesen Ereignissen experimentieren zu können, haben wir ein Formular namens **frmArtikelLoeschen** erstellt, das die Tabelle **tblArtikel** als Datensatzquelle verwendet und der wir eine Schaltfläche namens **cmdDatensatzLoeschen** hinzugefügt haben (siehe Bild 1).

Die Schaltfläche löst die folgende Ereignisprozedur aus:

```
Private Sub cmdDatensatzLoeschen_Click()
    RunCommand acCmdDeleteRecord
End Sub
```

Der Befehl **RunCommand acCmdDeleteRecord** entspricht dem Löschen eines Datensatzes durch Markieren des Datensatzes über den Datensatzmarkierer auf der linken Seite und anschließendes Betätigen der **Löschen**- oder der **Entf**-Taste. Alternativ können Sie auch die Tastenkombination **Strg + -** nutzen. Später werden wir sehen, dass hier noch eine Fehlerbehandlung notwendig ist.

Löschen-Ereignisse implementieren

Als Nächstes wollen wir die drei Ereignisprozeduren für die Ereignisse **Beim Löschen**, **Vor Löschbestätigung** und **Nach Löschbestätigung** implementieren.

Das erledigen wir wie üblich über das Eigenschaftenblatt des Formulars im Bereich **Ereignis**. Hier wählen Sie für die drei Ereignisse jeweils den Wert **[Ereignisprozedur]** aus und klicken auf die Schaltfläche mit den drei Punkten (**...**) – siehe Bild 2.

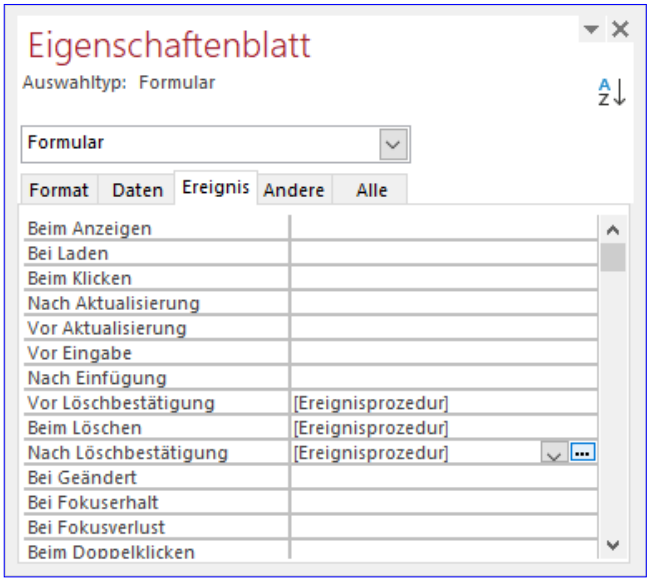


Bild 2: Implementieren der Löschen-Ereignisse

Damit legen Sie im Klassenmodul **Form_frmArtikel-Loeschen** drei Ereignisprozeduren an. Um diese analysieren zu können, fügen wir nun jeweils einen Haltepunkt hinzu und/oder eine **Debug.Print**-Anweisung, die den Namen der jeweiligen Prozedur im Direktbereich ausgibt – je nachdem, was für Sie praktischer erscheint (siehe Bild 3).

Reihenfolge der Ereignisprozeduren

Danach wechseln wir zur Formularansicht des Formulars **frmArtikelLoeschen** und löschen einen der Datensätze. Wie wir nun sehen, wird als Erstes die Ereignisprozedur **Form_Delete** (**Beim Löschen**) ausgelöst.

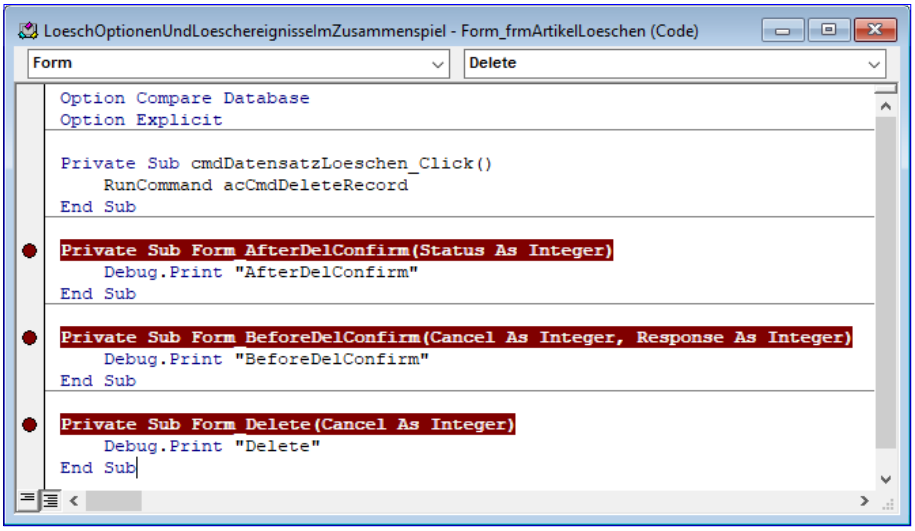


Bild 3: Die Löschen-Ereignisse im VBA-Editor

Zu diesem Zeitpunkt ist der Datensatz bereits aus dem Formular verschwunden. Dann wird das Ereignis **Form_BeforeDelConfirm** (**Vor Löschbestätigung**) ausgelöst. Danach gibt es einen Unterbrechung, in der die Meldung aus Bild 4 erscheint.

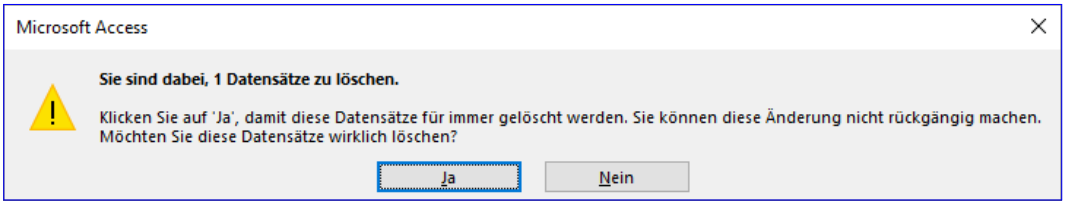


Bild 4: Meldung nach dem Ereignis **Vor Löschbestätigung**

Unabhängig davon, ob Sie die Schaltfläche **Ja** oder **Nein** betätigen, wird im Anschluss noch die Prozedur **Form_AfterDelConfirm** (**Nach Löschbestätigung**) aufgerufen. Allerdings wird im Falle von **Ja** der

aktuell angezeigte Datensatz gelöscht und im Falle von **Nein** wird dieser beibehalten und auch im Formular wieder angezeigt. Und auch beim Aufruf des Ereignisses **Nach Löschbestätigung** gibt es einen Unterschied, wie wir gleich sehen werden.

Parameter der Ereignisprozeduren

Hier wird es interessant, denn die Parameter von Ereignisprozeduren liefern entweder wichtige Informationen oder Sie können diese nutzen, um bestimmte Informationen zu übergeben.

Beginnen wir mit der Prozedur **Form_Delete**:

```
Private Sub Form_Delete(Cancel As Integer)
    Debug.Print "Delete", Cancel
End Sub
```

Diese enthält einen Parameter namens **Cancel**, der standardmäßig den Wert **0** enthält, was **False** entspricht. Das bedeutet: Standardmäßig hat **Cancel** den Wert **False**, was dazu führt, dass der Löschvorgang nicht abgebrochen wird.

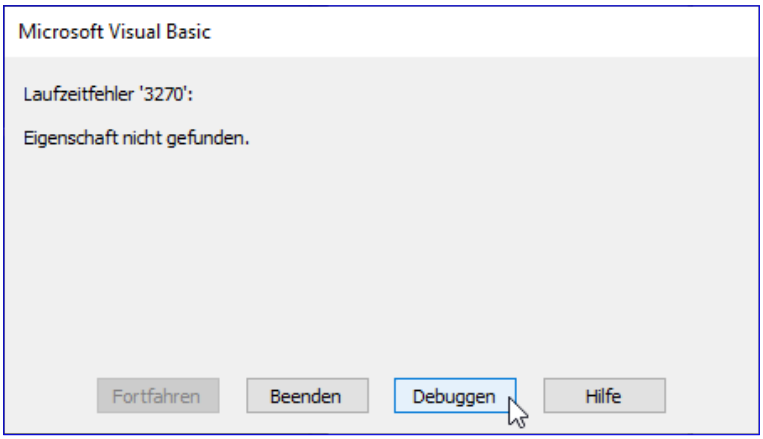


Bild 5: Fehler beim Abbruch von **RunCommand acCmdDeleteRecord**

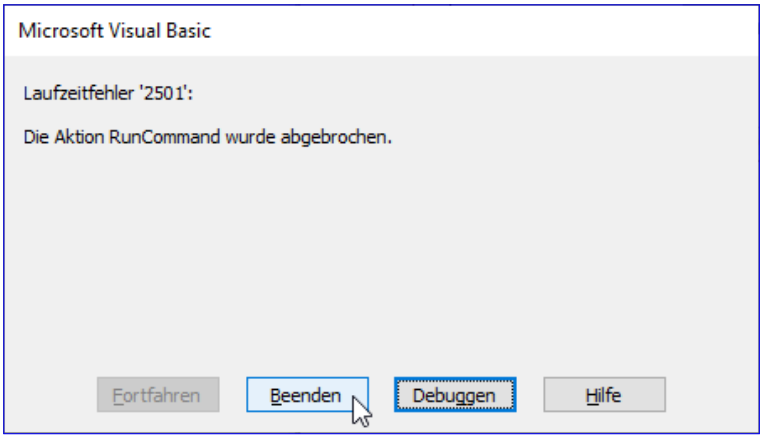


Bild 6: Fehler beim Abbruch von **RunCommand acCmdDeleteRecord** über die Standardrückfrage.

Individuelle Rückfrage zum Löschvorgang

Um das auszuprobieren, erweitern wir die Prozedur wie folgt:

```
Private Sub Form_Delete(Cancel As Integer)
    Debug.Print "Delete", Cancel
    If MsgBox("Löschen abbrechen?", vbYesNo) = vbYes Then
        Cancel = True
    End If
End Sub
```

Dies führt nun dazu, dass beim Löschen eine Meldung mit dem Text **Löschen abbrechen?** angezeigt wird. Klicken Sie hier auf **Ja**, stellt die Prozedur den Parameter **Cancel** auf den Wert **True** ein. Das funktioniert ohne Probleme, wenn Sie den Datensatz mit der Tastenkombination **Strg + -** löschen oder durch Anklicken des Datensatzmarkierers und anschließendes Betätigen der **Löschen**- oder der **Entf**-Taste.

Wenn Sie allerdings die Schaltfläche **cmdDatensatz-Loeschen** verwendet haben, löst dies einen Fehler aus, dessen Meldung wir uns in diesem Zusammenhang nicht erklären können (siehe Bild 5).

Bei der Gelegenheit zeigen wir auch direkt die Fehlermeldung, die erscheint, wenn Sie das Löschen mit **RunCommand acCmdDeleteRecord** initiieren und dann in der standardmäßigen Rückfrage den Löschvorgang abbrechen (siehe Bild 6). Das ist noch nachvollziehbar und dieser

E-Mails versenden mit CDO

Für das Versenden von E-Mails von einer Access-Anwendung aus gibt es verschiedene Möglichkeiten. Die naheliegendste ist der Versand unter Verwendung von Outlook, da dieses üblicherweise auf Rechnern mit Microsoft Access installiert ist. Es gibt jedoch auch Fälle, bei denen kein Office-Paket vorliegt und daher eine alternative Lösung gefragt ist. Früher gab es die Bibliothek vbSendmail, die auch heute noch eingeschränkt funktioniert. Eingeschränkt deshalb, weil beispielsweise SSL nicht unterstützt wird. Also haben wir nach einer Alternative gesucht, die auch moderne, sichere Versender unterstützt und sind dabei auf eine eher betagte Lösung gestoßen: die Bibliothek CDO, die auf jedem Windows-System installiert ist.

CDO-Bibliothek einbinden

Um die Elemente der CDO-Bibliothek zu nutzen, haben Sie wie üblich zwei Möglichkeiten – Late Binding und Early Binding. Da wir immer gern mit IntelliSense arbeiten, nutzen wir auch in diesem Fall wieder Early Binding. Das zieht nach sich, dass wir einen Verweis auf die Bibliothek zum VBA-Projekt der Zielanwendung hinzufügen müssen.

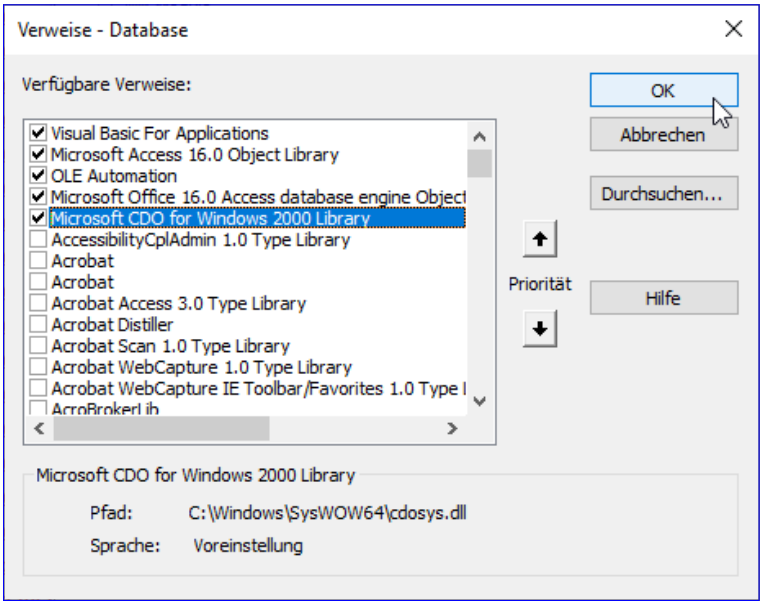


Bild 1: Verweis auf die CDO-Bibliothek

Dazu öffnen Sie den VBA-Editor mit der Tastenkombination **Alt + F11** und betätigen dann den Menübefehl **Extras|Verweise**. Im nun erscheinenden Verweise-Dialog für das aktuelle VBA-Projekt suchen Sie nach einem Eintrag namens **Microsoft CDO for Windows 2000 Library** und markieren diesen (siehe Bild 1). Dem Namen können Sie bereits entnehmen, warum wir die Bibliothek in der Einführung als »betagt« bezeichnet haben.

Nach dem Hinzufügen der Bibliothek schauen wir zuerst einmal in den Objektkatalog und wählen dort ganz oben den Eintrag **CDO** aus.

Hier finden wir schnell eine interessante Klasse namens **Message**, die alle notwendigen Eigenschaften und Methoden zu bieten scheint – inklusive **To**, **From**, **Subject**, **TextBody** und **Send** (siehe Bild 2).

Allerdings vermissen wir hier Eigenschaften, mit denen wir festlegen können, über welchen SMTP-Server wir die E-Mails versenden und wie wir uns dort authentifizieren. Die Bibliothek bietet neben einigen Interface-Klassen nur noch zwei weitere Klassen – nämlich **Configuration** und **DropDirectory**.

Darüber hinaus bietet die Klasse **Message** auch noch eine Eigenschaft namens **Configuration**, sodass wir wohl die notwendigen Eigenschaften über die **Configuration**-Klasse vornehmen werden.

Schauen wir allerdings im Objektkatalog in die Elemente der Klasse **Configuration**, finden wir dort nicht viel, was uns direkt weiterbringt – hier sehen wir nämlich nur die drei Elemente **Fields**, **GetInterface** und **Load**.

Auf der Suche nach dem Schlüsselwort **SMTP** landen wir jedoch schnell bei der Klasse **CdoConfiguration**, die einige Elemente bereitstellt, die hilfreich aussehen. Diese sehen Sie in Bild 3.

Eine weitere Recherche im Internet lieferte den Hinweis, dass wir Eigenschaften wie etwa den SMTP-Server über die **Fields**-Auflistung des **Configuration**-Objekts festlegen und dieses dann über die Eigenschaft **Configuration** dem **Message**-Objekt zuweisen.

Grundlegende Anwendung

Wie sich zeigen wird, benötigen wir für verschiedene Provider unterschiedliche Konfigurationen. Wir schauen uns erst einmal die einfachste Variante an, bei der wir von einem E-Mail-Provider mit einem einfachen SMTP-Server ausgehen, der keine zusätzlichen Funktionen wie SSL oder dergleichen bietet.

Um die zu verwendenden Parameter herauszufinden, starten wir erst einmal ohne Parameter mit der folgenden Prozedur.

Hier deklarieren wir zuerst einmal ein **Message**- und ein **Configuration**-Objekt:

```
Public Sub MailSenden()
    Dim objMessage As CDO.Message
    Dim objConfiguration As CDO.Configuration
```

Dann erstellen wir das **Configuration**-Objekt, mit dem wir erst einmal nichts konfigurieren, sowie ein **Message**-Objekt:

```
    Set objConfiguration = New CDO.Configuration
    Set objMessage = New CDO.Message
```

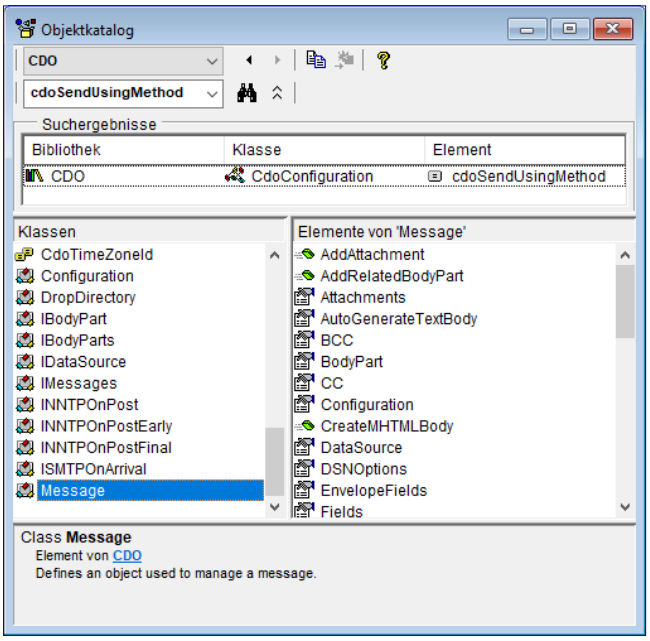


Bild 2: Eigenschaften und Methoden der **Message**-Klasse

Danach weisen wir dem **Message**-Objekt das leere **Configuration**-Objekt zu und legen einige andere, offensicht-

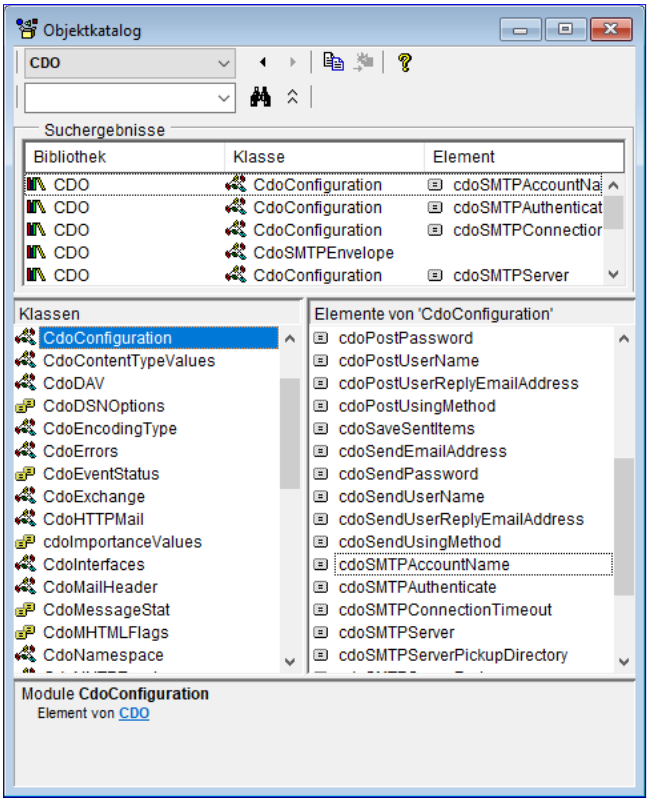


Bild 3: Konstanten für die **Fields**-Eigenschaft

lich benötigte Informationen fest – nämlich den Adressat, den Absender, den Betreff und den Inhalt der E-Mail:

```
    With objMessage
        Set .Configuration = objConfiguration
        .To = "andre@minhorst.com"
        .From = "info@amvshop.de"
        .Subject = "Beispielbetreff"
        .TextBody = "Dies ist ein Beispieltext."
```

Danach rufen wir bei deaktivierter eingebauter Fehlerbehandlung die **Send**-Methode auf und lassen uns im Falle eines Fehlers die Fehlernummer und den Fehlertext ausgeben:

```
        On Error Resume Next
        .Send
        If Not Err.Number = 0 Then
            Debug.Print Err.Number, Err.Description
        End If
    End With
    Set objConfiguration = Nothing
    Set objMessage = Nothing
End Sub
```

Erster Versuch des Mailversands

Rufen wir die Prozedur so auf, erhalten wir den folgenden Fehler:

```
-2147220960
Der "SendUsing"-Konfigurationswert ist ungültig.
```

Mit **SendUsing** landen wir im Objektkatalog schnell bei der Klasse **CdoSendUsing** mit der Eigenschaften **SendUsingPort**.

Senden mit Port

Also fügen wir dem **Configuration**-Objekt ein **Fields**-Element mit dieser Eigenschaft hinzu:

```
With objConfiguration
    .Fields(cdoSendUsingMethod).Value = cdoSendUsingPort
    .Fields.Update
End With
```

Dies liefert den folgenden Fehler:

```
-2147220982
Der erforderliche Name des SMTP-Servers wurde in der Konfigurationsquelle nicht gefunden.
```

Also geben wir auch noch einen SMTP-Server an:

```
...
.Fields(cdoSMTPServer).Value = "smtp.minhorst.com"
...
```

Damit versenden wir tatsächlich schon die erste E-Mail mit einem einfachen SMTP-Server ohne besondere Absicherung. Das funktionierte in unserem Fall aber auch nur, weil die Absender- und die Empfängeradresse beide mit dem angegebenen SMTP-Server arbeiten.

E-Mail-Server mit Verschlüsselung nutzen

Je nachdem, welchen Server Sie verwenden, können die notwendigen Angaben variieren. In den folgenden Abschnitten schauen wir uns die notwendigen Einstellungen für das **Configuration**-Objekt an.

Dazu haben wir eine neue, parametrisierte Version der obigen Prozedur namens **SendMail** geschaffen, mit der Sie alle notwendigen Einstellungen inklusive der Informationen für die E-Mails übergeben können (siehe Listing 1).

Die Prozedur erwartet die folgenden Parameter:

- **strServername**: Name des Servers, zum Beispiel **smtp.mailserver.de**
- **strUsername**: Benutzername für das Mailkonto, in der Regel die E-Mail-Adresse

Serienmails versenden mit CDO

Zum Verwenden von Serienmails nutzt man meist Outlook und schreibt eine Mail an sich selbst, während man die Empfänger dann möglichst dem Feld BCC hinzufügt. Auf diese Weise bleibt der Datenschutz gewahrt, denn Sie wollen ja nicht jedem Empfänger die E-Mail-Adressen aller anderen Empfänger der Serienmail mitteilen. Der Nachteil ist, dass Sie so noch nicht einmal einfache Individualisierungen realisieren können wie etwa eine persönliche Anrede. Wenn Sie das erledigen wollen, versenden Sie per Automation aus einer Datenbank Nachrichten über Outlook. Falls das nicht in Frage kommt, weil beispielsweise Outlook nicht auf dem Rechner installiert ist, können Sie noch eine Alternative nutzen, nämlich die Bibliothek CDO. Im Beitrag »E-Mails versenden mit CDO« haben wir bereits die grundlegenden Techniken zum E-Mail-Versand mit dieser Bibliothek vorgestellt. Nun gehen wir einen Schritt weiter und zeigen, wie das auch noch für Serienmails gelingt.

Vorbereitung für den Einsatz der CDO-Bibliothek

Damit Sie die CDO-Bibliothek mit den Objekten und Methoden zum Versenden von E-Mails nutzen können, benötigen Sie einen Verweis auf diese Bibliothek. Diesen fügen Sie über den **Verweise**-Dialog hinzu, den Sie vom VBA-Editor aus öffnen. Hier wählen Sie den Menüeintrag **Extras|Verweise** und wählen dann im erscheinenden Dialog **Verweise** den Eintrag **Microsoft CDO for Windows 2000 Library** aus (siehe Bild 1).

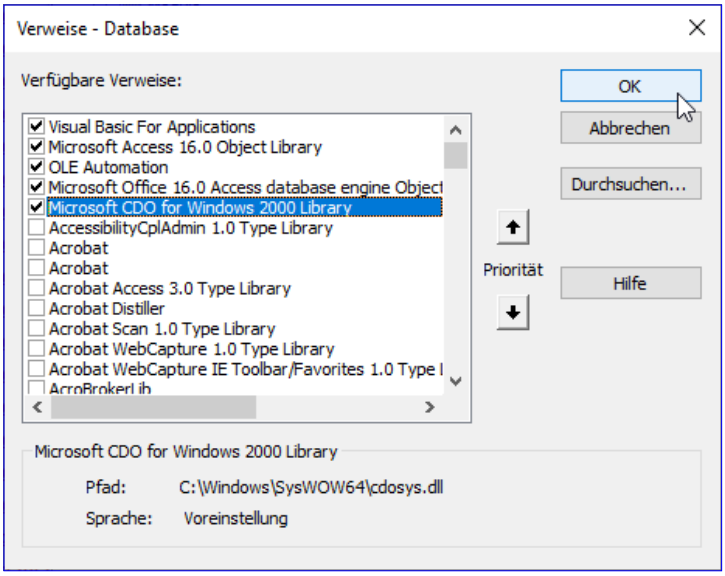


Bild 1: Verweis auf die CDO-Bibliothek

Vorbereiten der Datenbank

Um die hier vorgestellten Techniken nutzen zu können, benötigen Sie zumindest eine Tabelle mit den Daten der E-Mail-Empfänger. Diese sollte im besten Fall die Felder **Anrede**, **Vorname**, **Nachname** und **E-Mail-Adresse** enthalten.

Wir haben eine solche Tabelle wie in Bild 2 definiert. Das dortige Feld **AnredeID** ist ein Fremdschlüsselfeld, mit dem Sie die Datensätze der Tabelle **tblAnreden** auswählen können. Diese enthält neben dem Primärschlüsselfeld **AnredeID** noch das Feld **Anrede** mit der

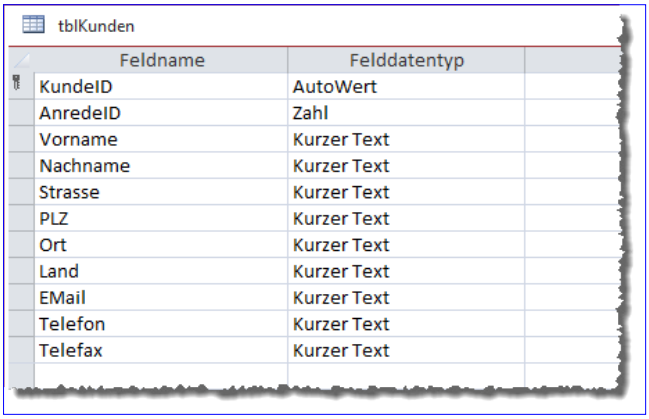


Bild 2: Beispieltabelle für Kundendaten im Entwurf

Bezeichnung der Anrede sowie ein weiteres Feld namens Mailanrede, das wir mit einer Anrede speziell für die E-Mails gefüllt haben.

Diese Tabelle sieht mit den Daten für die Anreden **Herr** und **Frau** wie in Bild 3 aus.

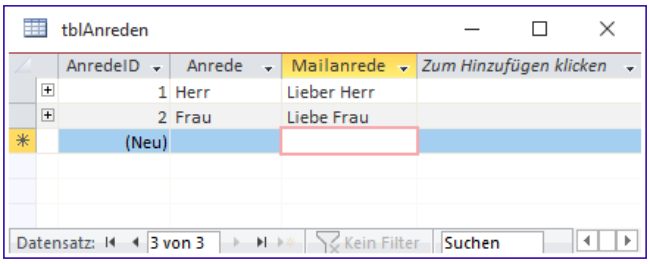


Bild 3: Tabelle mit den Anreden

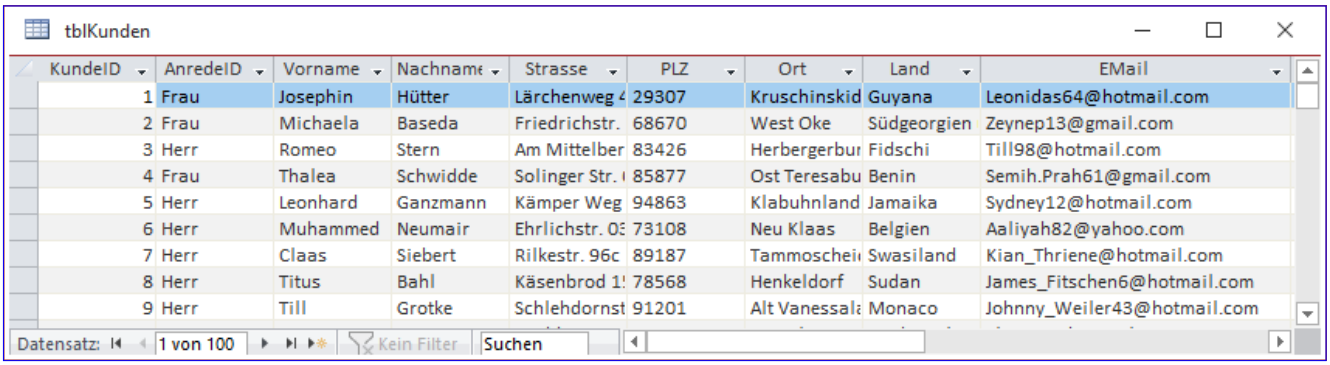


Bild 4: Tabelle mit den aufgefüllten Datensätzen

Die Tabelle **tblKunden** haben wir mit den Techniken aus dem Beitrag **Beispieldaten generieren mit .NET und Bogus** (**www.access-im-unternehmen.de/1359**) gefüllt. Die Tabelle sieht anschließend wie in Bild 4 aus. Wir werden später nur die ersten zwei, drei Datensätze, um testweise E-Mails zu versenden. Diesen fügen wir daher tatsächlich vorhandene E-Mail-Adressen hinzu, bei denen wir auch prüfen können, ob diese angekommen sind.

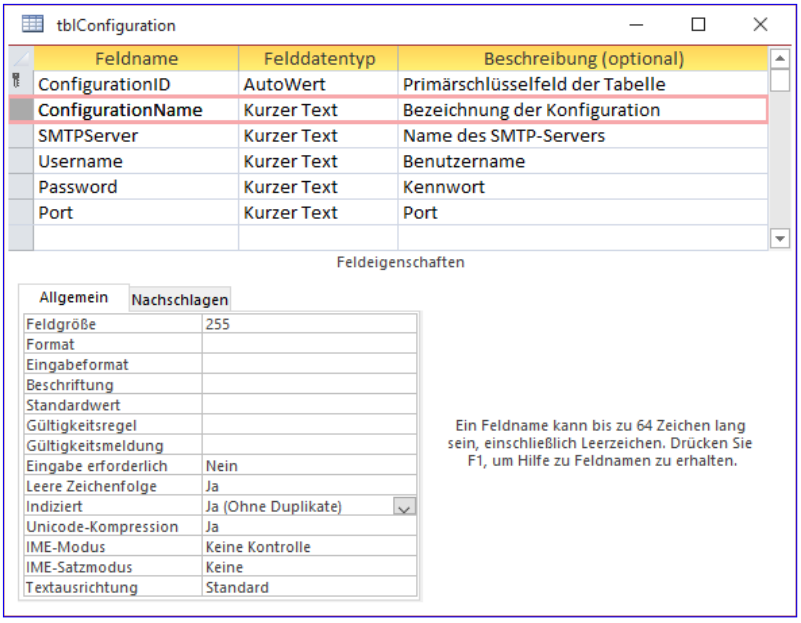


Bild 5: Tabelle zum Speichern der SMTP-Konfiguration

Speichern der SMTP-Konfiguration

Im Beitrag **E-Mails versenden mit CDO** (**www.access-im-unternehmen.de/1363**) haben wir die Daten für den Zugriff auf den SMTP-Server, der für den Mailversand verwendet werden soll, direkt in den Code geschrieben.

Das wollen wir in diesem Beitrag direkt ein wenig professioneller gestalten und die Daten in einer Tabelle unterbringen, deren Inhalte wir dann über ein Formular anpassen können.

Die Tabelle finden Sie in der Entwurfsansicht in Bild 5. Sie nimmt neben dem Primärschlüsselfeld zunächst ein Feld zum Speichern der Bezeichnung eines Satzes von SMTP-Server-Daten auf. Danach folgen die

Felder für die Adresse des SMTP-Servers, für den Benutzernamen und das Kennwort, unter dem die Anmeldung erfolgen soll, sowie gegebenenfalls der Port.

Auslesen der Konfiguration und Erstellen eines Configuration-Objekts

Zusätzlich programmieren wir eine Funktion, der wir nur noch den Primärschlüsselwert der zu liefernden Konfiguration übergeben. Diese soll dann ein fertiges **Configuration**-Objekt auf Basis der Daten aus dem entsprechenden Datensatz der Tabelle zurückliefern.

Diese Funktion sieht wie in Listing 1 aus. Sie erwartet den Parameter **lngConfigurationID** und liefert ein Objekt des Typs **Configuration** zurück. Sie erzeugt ein **Database**-Objekt auf Basis der aktuellen Datenbank und ein Recordset mit dem Datensatz der Tabelle **tblConfigurations**, dessen Wert im Feld **ConfigurationID** dem mit **lngConfigurationID** übergebenen Parameter entspricht.

Aus diesem Recordset liest die Funktion die einzelnen Feldwerte aus und schreibt diese in die Variablen **strSMTPServer**, **strPort**, **strUsername** und **strPassword**.

```
Public Function GetConfiguration(lngConfigurationID As Long) As CDO.Configuration
    Dim objConfiguration As CDO.Configuration
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strSMTPServer As String
    Dim strPort As String
    Dim strUsername As String
    Dim strPassword As String
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM tblConfigurations WHERE ConfigurationID = " & lngConfigurationID, _
        dbOpenDynaset)
    strSMTPServer = rst!SMTPServer
    strPort = Nz(rst!Port, "")
    strUsername = rst!UserName
    strPassword = rst!Password
    Set objConfiguration = New CDO.Configuration
    With objConfiguration
        .Fields(cdoSendUsingMethod).Value = cdoSendUsingPort
        .Fields(cdoSMTPServer).Value = strSMTPServer
        Debug.Print .Fields(cdoSMTPServerPort)
        If Not Len(strPort) = 0 Then
            .Fields(cdoSMTPServerPort).Value = strPort
        End If
        .Fields(cdoSMTPUseSSL).Value = "true"
        .Fields(cdoSMTPAuthenticate).Value = cdoBasic
        .Fields(cdoSendUserName).Value = strUsername
        .Fields(cdoSendPassword).Value = strPassword
        .Fields.Update
    End With
    Set GetConfiguration = objConfiguration
End Function
```

Listing 1: Funktion zum Ermitteln eines **Configuration**-Objekts

Dann erstellt sie ein neues Objekt des Typs **CDO.Configuration** und weist den Eigenschaften dieses Objekts die Werte aus den Textfeldern zu – neben einigen Werten für Eigenschaften, die immer gleich sind wie etwa für **cboSendUsingMethode**, **cdoSMTPUserSSL** oder **cdoSMTPAuthenticate**. Nachdem alle relevanten Eigenschaften gefüllt sind, gibt die Funktion das mit **objConfiguration** referenzierte Objekt als Ergebnis zurück.

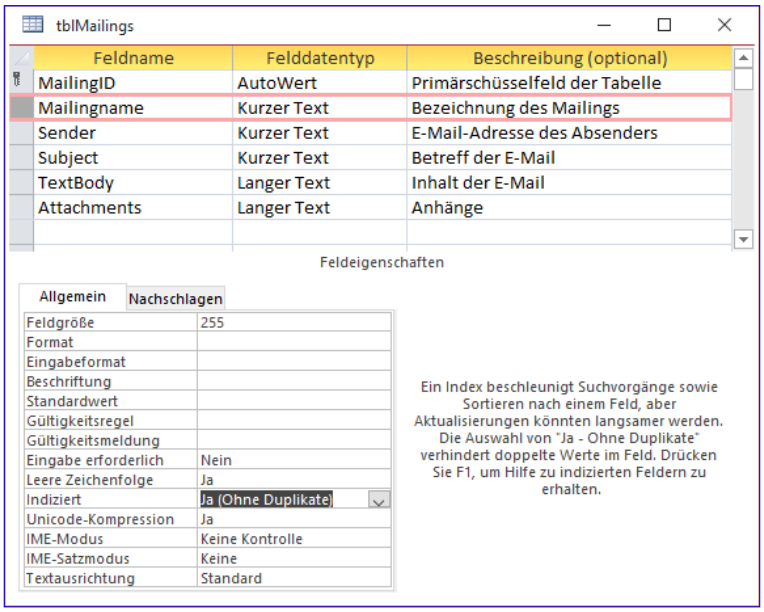


Bild 6: Tabelle zum Speichern der Daten eines Mailings

Serien-E-Mails verschicken

In der Lösung aus dem Beitrag **E-Mails senden mit CDO** (**www.access-im-unternehmen.de/1363**) haben wir das Zusammenstellen des **Configuration**-Objekts und des **Message**-Objekts, das zum Versenden einer E-Mail dient, in einer Prozedur erstellt. Das ist beim Versand einer Serienmail nicht sinnvoll – wenn immer der gleiche SMTP-Server mit den gleichen Anmeldedaten zum Einsatz kommt, brauchen wir nicht für jede E-Mail erneut ein **Configuration**-Objekt zu erstellen. Wenn wir also gleich eine Prozedur programmieren, mit der wir die Empfänger der E-Mail durchlaufen, wird dies berücksichtigen werden.

Für das Versenden einer Serienmail benötigen wir noch einige Daten, die wir am Besten auch direkt in einer eigenen Tabelle speichern. Dabei handelt es sich um die Daten, die für jede Mail eines Mailings gleich sind, also:

- E-Mail-Adresse des Absenders
- Betreff der E-Mail
- Inhalt der E-Mail, gegebenenfalls mit Platzhaltern
- Anhänge für die E-Mail, gegebenenfalls mit Platzhaltern

Diese Daten speichern wir in der Tabelle **tblMailings** aus Bild 6. Neben den genannten Daten nimmt diese Tabelle nur noch ein Primärschlüsselfeld auf sowie ein Feld für die Bezeichnung des Mailings. Für das Feld **Mailingname**

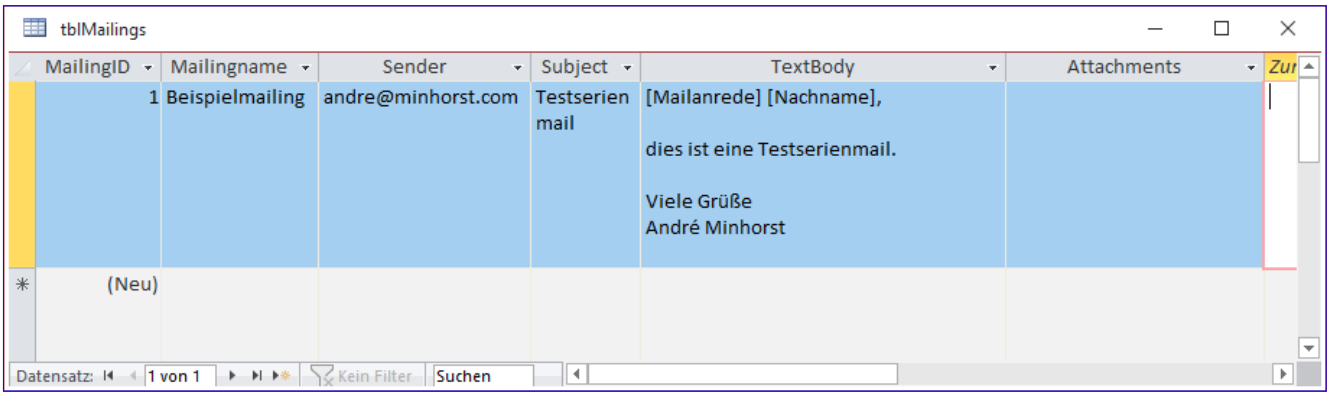


Bild 7: Tabelle zum Speichern der Daten eines Mailings mit Beispieldaten

Benutzeroberfläche für CDO-Serienmails

Im Beitrag »Serienmails versenden mit CDO« haben wir einige Prozeduren und Funktionen vorgestellt, mit denen Sie Serien-E-Mails über die CDO-Bibliothek von Windows versenden können. Das macht natürlich nur halb soviel Spaß, wenn nur die nackten Routinen vorliegen. Also zeigen wir im vorliegenden Beitrag auch noch, wie Sie eine praktische Benutzeroberfläche zum Verwalten der für den Versand einer Serienmail benötigten Daten programmieren.

Datenmodell der Anwendung

Die Anwendung zum Versenden von Serienmails enthält einige Tabellen, die wie im Folgenden vorstellen.

Im **Beziehungen**-Fenster aus Bild 1 sehen Sie die benötigten Tabellen. Die ersten beiden heißen **tblKunden** sowie **tblAnreden**. Die Tabelle **tblConfigurations** enthält die Daten für die Konfiguration des Mailservers und die Tabelle **tblMailings** die Daten zum Mailing selbst, also eine Bezeichnung, die Absenderadresse, den Betreff, den Inhalt und die Anlagen. Der Tabelle **tblMailings** haben wir ein Fremdschlüsselfeld namens **ConfigurationID** hinzugefügt, mit dem wir die für das jeweilige Mailing verwendete Konfiguration speichern können. Außerdem enthält die Tabelle **tblMailings** zwei Felder, mit denen die Adressaten des Mailings definiert werden. **SQLSource** nimmt den SQL-Ausdruck auf, der die Empfängeradressen liefert und **Mailfield** den Namen des Feldes aus diesem Ausdruck, der festlegt, welches Feld die Mailadresse enthält.

Zu erstellende Formulare

Wir wollen der Lösung einige Formulare hinzufügen, mit der Sie diese komfortabel steuern können. Dazu gehören die folgenden:

- Formular zum Verwalten der Konfigurationen
- Formular zum Verwalten der Mailings
- Formular zum Auswählen der Empfänger des Mailings

Formular zum Verwalten der Konfigurationen

Wir beginnen mit dem Formular **frmConfigurations**. Das Formular verwendet die Tabelle **tblConfigurations** als Datensatzquelle. Ziehen Sie alle Felder aus der Feldliste in das Formular und ordnen Sie diese so an wie in Bild 2. Dann fügen Sie noch einige weitere Steuerelemente hinzu:

- Kombinationsfeld **cboSchnellauswahl** (in den Formularkopf)

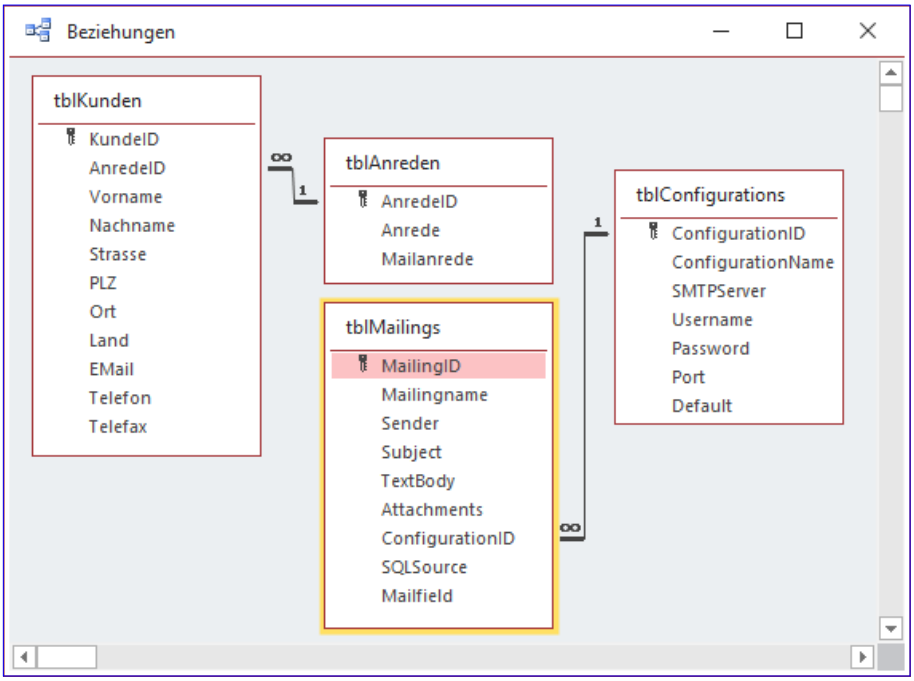


Bild 1: Datenmodell der Anwendung

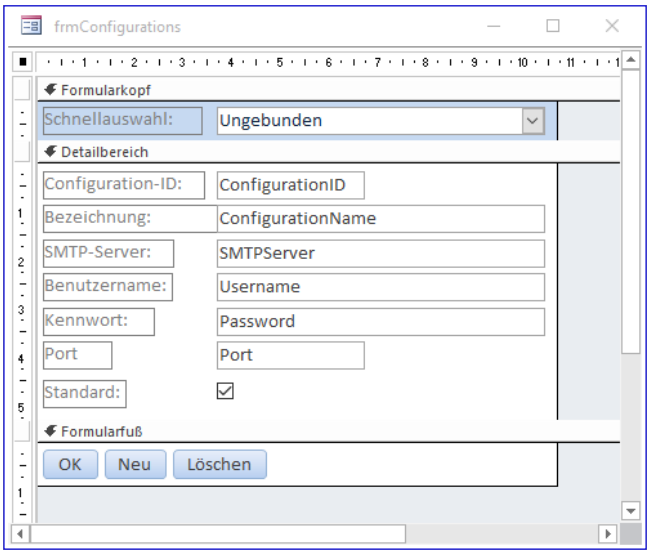


Bild 2: Entwurf des Formulars **frmConfigurations**

- Schaltfläche **cmdOK** (mit den anderen Schaltflächen in den Formularfuß)
- Schaltfläche **cmdNeu**
- Schaltfläche **cmdLoeschen**

Außerdem passen wir den Namen des Kontrollkästchens für das Feld **Standard** auf **chkDefault** an.

Schnellauswahl für die Konfiguration

Das Kombinationsfeld **cboSchnellauswahl** verwendet eine Abfrage auf Basis der Tabelle **tblConfigurations** als Datensatzherkunft. Dabei verwenden wir nur die ersten beiden Felder **ConfigurationID** und **ConfigurationName**, wobei wir diese nach dem Feld **ConfigurationName** sortieren wollen:

```
SELECT ConfigurationID, ConfigurationName
FROM tblConfigurations
ORDER BY ConfigurationName;
```

Damit das Kombinationsfeld nur die Bezeichnung der Konfiguration anzeigt, aber nicht den Inhalt des Autowertfeldes, stellen wir die Eigenschaft **Spaltenanzahl** auf **2** und **Spaltenbreiten** auf **0cm** ein. Damit das Kombinationsfeld nach dem Wechsel zu einem anderen Datensatz im Formular den Namen der aktuellen Konfiguration anzeigt, fügen wir die folgende Prozedur hinzu, die durch das Ereignis **Beim Anzeigen** ausgelöst wird:

```
Private Sub Form_Current()
    Me!cboSchnellauswahl = Me!ConfigurationID
End Sub
```

Gegebenenfalls ändert der Benutzer die Bezeichnung der Konfiguration. Damit sich dies direkt im Kombinationsfeld zur Schnellauswahl wiederspiegelt, aktualisieren wir dieses, wenn der Benutzer Daten im Formular aktualisiert und gespeichert hat:

```
Private Sub Form_AfterUpdate()
    Me!cboSchnellauswahl.Requery
End Sub
```

Aktionen beim Laden des Formulars

Beim Öffnen des Formulars soll dieses den ersten Datensatz anzeigen, der im Feld **Default** den Wert **True** enthält – der also als Standardkonfiguration definiert ist. Damit dies geschieht, suchen wir direkt in der Prozedur, die durch das Ereignis **Beim Laden** ausgelöst wird, mit der **FindFirst**-Methode des Recordsets des Formulars nach diesem Datensatz. Unabhängig davon, ob dies einen Datensatz findet, soll das Kombinationsfeld **cboSchnellauswahl** auf den gleichen Datensatz eingestellt werden, den auch das Formular anzeigt. Dafür stellen wir den Wert von **Me!cboSchnellauswahl** anschließend auf **Me!ConfigurationID** ein:

```
Private Sub Form_Load()
    Me.Recordset.FindFirst "Default = True"
    Me!cboSchnellauswahl = Me!ConfigurationID
End Sub
```

Auswahl einer anderen Konfiguration per Kombinationsfeld

Damit das Formular nach der Auswahl eines anderen Datensatzes über das Kombinationsfeld **cboSchnellaus-**

wahl den gewünschte Datensatz anzeigt, fügen wir dem Kombinationsfeld eine Prozedur für das Ereignis **Nach Aktualisierung** hinzu. Diese sieht wie folgt aus und prüft zunächst, ob im Kombinationsfeld überhaupt ein Datensatz ausgewählt ist. Falls ja, sucht die Prozedur mit der **FindFirst**-Methode des Recordsets des Formulars nach dem betroffenen Datensatz und stellt diesen ein:

```
Private Sub cboSchnellauswahl_AfterUpdate()
    If Not Nz(Me!cboSchnellauswahl, 0) = 0 Then
        Me.Recordset.FindFirst "ConfigurationID = " _
                                & Me!cboSchnellauswahl
    End If
End Sub
```

Einstellen der Standardkonfiguration

Das Feld **Default** der Tabelle **tblConfigurations** legt fest, welche Konfiguration standardmäßig verwendet werden soll. Das bedeutet schlicht und einfach, dass diese Konfiguration beim Anlegen neuer Mailings für das Fremdschlüsselfeld **ConfigurationID** der Tabelle **tblMailings** festgelegt wird. Außerdem soll diese Konfiguration beim Öffnen des Formulars **frmConfigurations** standardmäßig angezeigt werden, was wir weiter oben bereits realisiert haben.

Mit dem Kontrollkästchen wollen wir dem Benutzer die Möglichkeit bieten, eine andere Konfiguration als Standardkonfiguration festzulegen. Dabei nutzen wir als Erstes die Prozedur, die durch das Ereignis **Vor Aktualisierung** des Kontrollkästchens **chkDefault** ausgelöst wird. Diese prüft, ob der Benutzer soeben den Haken aus dem Kontrollkästchen entfernt hat. Falls ja, soll eine Meldung erscheinen, die den Benutzer darauf hinweist, dass er zum Ändern der Standardkonfiguration erst zu der gewünschten Standardkonfiguration wechseln muss und diese dann mit einem Klick auf das Kontrollkästchen mit dem Text **Standard** festlegen kann. Das Ereignis **Vor Aktualisierung** nutzen wir deshalb, weil wir hier mit dem **Cancel**-Parameter einstellen können, dass die Änderung rückgängig gemacht wird, wenn der Benutzer das Kontrollkästchen deaktiviert hat:

```
Private Sub chkDefault_BeforeUpdate(Cancel As Integer)
    If Not Me!chkDefault Then
        MsgBox "Um eine andere Konfiguration zur _
         Standardkonfiguration zu machen, aktivieren Sie _
           diese Option für die gewünschte Konfiguration."
        Cancel = True
    End If
End Sub
```

Wenn der Benutzer hingegen das Kontrollkästchen **chkDefault** aktiviert hat, soll die aktuell angezeigte Konfiguration als Standardkonfiguration eingestellt werden. In diesem Fall feuert auch noch das Ereignis **Nach Aktualisierung** des Kontrollkästchens, was die folgende Prozedur auslöst:

```
Private Sub chkDefault_AfterUpdate()
    Dim db As DAO.Database
    If Me!chkDefault Then
        Set db = CurrentDb
        db.Execute "UPDATE tblConfigurations _
           SET Default = 0 WHERE NOT ConfigurationID = " _
                    & Me!ConfigurationID, dbFailOnError
    End If
End Sub
```

Diese Prozedur prüft, ob **chkDefault** den Wert **True** hat, also ob der Benutzer das Kontrollkästchen für diesen Datensatz aktiviert hat. Falls ja, führt die Prozedur mit der **Execute**-Methode des **Database**-Objekts eine Abfrage aus, die den Wert des Feldes **Default** für alle anderen Datensätze der Tabelle **tblConfigurations** auf **0** einstellt. Damit ist sichergestellt, dass das Feld **Default** nur für den aktuell angezeigten Datensatz den Wert **True** aufweist.

Löschen einer Konfiguration

Mit einem Klick auf die Schaltfläche **cmdLoeschen** kann der Benutzer die aktuelle Konfiguration löschen. Dazu löst die Schaltfläche die folgende Prozedur aus. Diese löscht den aktuellen Datensatz und versucht danach, den Datensatzzeiger auf den ersten Datensatz einzustellen, dessen Feld **Default** den Wert **True** aufweist:

```
Private Sub cmdLoeschen_Click()
    RunCommand acCmdDeleteRecord
    Me.Recordset.FindFirst "Default = True"
End Sub
```

Anlegen einer neuen Konfiguration

Um eine neue Konfiguration anzulegen, muss der Benutzer nur auf die Schaltfläche **cmdNeu** klicken. Diese springt dann zu einem neuen, leeren Datensatz:

```
Private Sub cmdNeu_Click()
    DoCmd.GoToRecord Record:=acNewRec
End Sub
```

Schließen des Formulars

Ein Klick auf die Schaltfläche **cmdOK** ruft die **DoCmd.Close**-Methode auf und schließt das aktuelle Formular:

```
Private Sub cmdOK_Click()
    DoCmd.Close acForm, Me.Name
End Sub
```

Wenn Sie in die Formularansicht wechseln und einen Datensatz eingegeben haben, sieht dieses wie in Bild 3 aus.

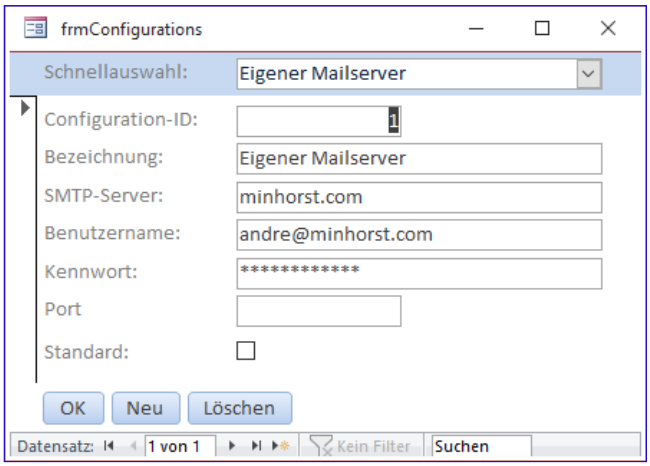


Bild 3: Das Formular **frmConfigurations** in der Formularansicht

Kennwort verbergen

Damit das Textfeld **Password** das Kennwort nicht direkt anzeigt, sondern Sternchen als Platzhalter verwendet, stellen wir die Eigenschaft **Eingabeformat** auf **Kennwort** ein.

Formular zum Verwalten der Mailings

Das Formular **frmMailings** dient zum Verwalten der Mailings. Es verwendet die Tabelle **tblMailings** als Datensatzquelle. Aus dieser Tabelle haben wir alle Felder in den Detailbereich der Entwurfsansicht gezogen.

Für die in Bild 4 markierten Felder stellen wir die Eigenschaft **Horizontaler Anker** auf den Wert **Beide** ein. So können wir die Felder vergrößern, indem wir die Breite des Formulars vergrößern. Anschließend müssen Sie die Bezeichnungsfelder dieser Steuerelemente markieren und für diese die Eigenschaft **Horizontaler Anker** auf **Links** einstellen, da diese durch die vorherige Einstellung auf **Rechts** festgelegt wurde.

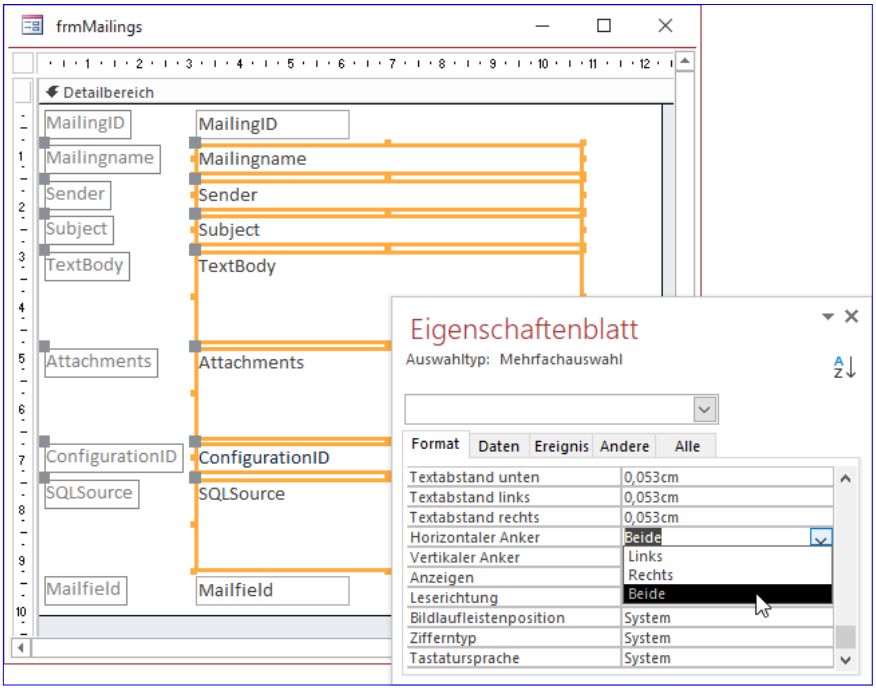


Bild 4: Das Formulars **frmMailings** in der Entwurfsansicht

Eines der Felder können wir außerdem in der Höhe anpassbar machen. Das ist für das Feld **TextBody** am sinnvollsten. Daher stellen wir seine Eigenschaft **Vertikaler Anker** auf **Beide** ein. Auch hier müssen wir die Eigenschaft **Vertikaler Anker** für das entsprechende Bezeichnungsfeld wieder auf **Oben** festlegen. Außerdem müssen wir die Eigenschaft **Vertikaler Anker** für alle Felder, die sich unterhalb des Textfeldes **TextBody** befinden, auf **Unten** einstellen, da diese sonst beim Vergrößern der Höhe des Formulars vom Textfeld **TextBody** überlappt werden.

Die Beschriftungen der Bezeichnungsfelder wurden bisher von den Feldnamen übernommen, diese passen wir jedoch auch noch an (noch nicht im Screenshot sichtbar).

Listenfeld für die Anhänge

Sie sehen die geänderten Bezeichnungen in Bild 5. Hier sehen Sie auch, wie Sie das Textfeld für die Anzeige der Daten des Feldes **Attachments** in ein Listenfeld umwandeln können – und zwar über das Kontextmenü. Anschließend ändern Sie noch den Namen des Listenfeldes in **lstAttachments**. Leider können wir die Bindung des Listenfeldes an das Feld **Attachments** danach nicht mehr nutzen, denn ein Listenfeld bindet man in der Regel an eine Datensatzherkunft wie eine Tabelle oder Abfrage. Deshalb bauen wir das Listenfeld ein wenig um. Als Erstes entfernen Sie den Inhalt der Eigenschaft **Steuerelementinhalt**. Dann stellen Sie die Eigenschaft **Herkunftstyp** auf **Wertliste** ein. Schließlich sorgen wir dafür, dass beim Wechseln des Datensatzes im Formular der Inhalt des Feldes **Attachments** als Wert des Listenfeldes eingestellt wird.

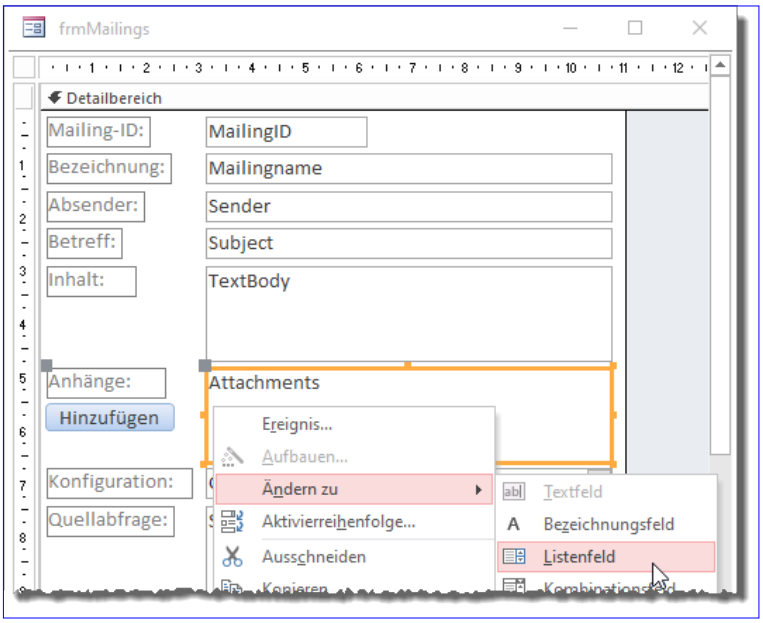


Bild 5: Ändern des Textfeldes für die Anhänge in ein Listenfeld

Dazu legen wir die Prozedur **Form_Current** mit der folgenden Anweisung an:

```
Private Sub Form_Current()
    Me!lstAttachments.RowSource = Me!Attachments
End Sub
```

Damit zeigt das Listenfeld bei jedem Datensatzwechsel automatisch die enthaltenen Attachments an.

Attachments hinzufügen

Dem Listenfeld **lstAttachments** stellen wir eine Schaltfläche namens **cmdHinzufuegen** zur Seite, welche die Ereignisprozedur aus Listing 1 auslöst. Diese stellen wir in ähnlicher Form im Beitrag **E-Mails mit Anlagen mit Outlook versenden** (**www.access-im-unternehmen.de/1357**) vor. Wir mussten allerdings einige Anpassungen vornehmen, da unser Listenfeld wie oben beschrieben nicht die Daten einer eigenen Tabelle oder Abfrage anzeigt, sondern den Inhalt des Feldes **Attachment** als Wertliste. Deshalb arbeiten wir beim Hinzufügen nicht mit der **AddItem**-Methode des Listenfeldes, sondern fügen die im Dateiauswahl-Dialog selektierten Daten dem Text aus dem Feld **Attachments** der Tabelle **tblMailings** hinzu, sofern die jeweilige Datei dort noch nicht enthalten ist. Für die Nutzung des Dateiauswahl-Dialogs fügen Sie dem VBA-Projekt noch einen Verweis auf die Bibliothek **Microsoft Office x.0 Object Library** hinzu.

Attachments leeren

Im Gegensatz zu dem Listenfeld für Anlagen aus dem soeben genannten Beitrag wollen wir hier nur eine Schaltfläche zum Leeren des Listenfeldes hinzufügen und nicht das Selektieren einzelner zu entfernender Einträge erlauben. Dazu hinterlegen wir für die Schaltfläche **cmdLeeren** die folgende Ereignisprozedur:

```
Private Sub cmdLeeren_Click()
    Me!Attachments = Null
    Me!lstAttachments.RowSource = ""
End Sub
```

Diese leert sowohl das Feld **Attachments** der Datensatzquelle als auch das Listenfeld **lstAttachments**.

Konfiguration automatisch auswählen

Damit bei einem neuen Mailing direkt die als Standard markierte Konfiguration im Feld **cboConfigurationID** ausgewählt wird, legen wir die folgende Prozedur für das Ereignis **Beim Anzeigen** des Formulars fest:

```
Private Sub Form_Current()
    Dim lngConfigurationID As Long
    If Me.NewRecord Then
        lngConfigurationID = Nz(DLookup("ConfigurationID", "tblConfigurations", "Default = True"), 0)
        If Not lngConfigurationID = 0 Then
            Me!cboConfigurationID.DefaultValue = lngConfigurationID
        End If
```

```
Private Sub cmdHinzufuegen_Click()
    Dim objFileDialog As FileDialog
    Dim l As Long, Dim m As Long
    Dim bolVorhanden As Boolean
    Dim strAttachments As String
    Dim varAttachment As Variant
    strAttachments = Me!lstAttachments.RowSource
    Set objFileDialog = FileDialog(msoFileDialogFilePicker)
    With objFileDialog
        .AllowMultiSelect = True
        .Title = "Anlagen auswählen"
        .Filters.Clear
        .Filters.Add "Alle Dateien", "*.*"
        .ButtonName = "Hinzufügen"
        If .Show = True Then
            For l = 1 To .SelectedItems.Count
                If Not Len(Me!lstAttachments.RowSource + .SelectedItems(l)) > 32750 Then
                    bolVorhanden = False
                    For Each varAttachment In Split(strAttachments, ";")
                        If varAttachment = .SelectedItems(l) Then
                            bolVorhanden = True
                            Exit For
                        End If
                    Next varAttachment
                    If Not bolVorhanden Then
                        strAttachments = strAttachments & ";" & .SelectedItems(l)
                    End If
                    If Left(strAttachments, 1) = ";" Then
                        strAttachments = Mid(strAttachments, 2)
                    End If
                Else
                    MsgBox "Es können keine weiteren Dateien hinzugefügt werden."
                    Exit Sub
                End If
            Next l
        End If
    End With
    Me!lstAttachments.RowSource = strAttachments
    Me!Attachments = strAttachments
End Sub
```

Listing 1: Prozedur zum Hinzufügen von Anlagen

HAUFE.
Ausgabe 04/2022

ACCESS
IM UNTERNEHMEN

RECHNUNGEN VERWALTEN

Programmieren Sie eine komplette Rechnungsverwaltung mit unserer neuen Beitragsreihe (ab Seite 15).

In diesem Heft:

NUMMERN GENERIEREN

Erstellen Sie individuelle Nummern für Kunden, Bestellungen oder Produkte nach Ihren eigenen Vorgaben.

SEITE 9

EINGABE IN TEXTFELDER EINSCHRÄNKEN

Verhindern Sie Eingaben beispielsweise von Buchstaben in Felder, die nur Zahlen aufnehmen sollen.

SEITE 46

BESCHRIFTUNGSFELDER IM GRIFF

Lernen Sie die Tücken von Beschriftungsfeldern kennen und umgehen Sie diese.

SEITE 38

www.access-im-unternehmen.de
Mat-Nr. 01583-5132

ARCHIV SEITE 233

Bestellungen und Rechnungen verwalten

In dieser Ausgabe von Access im Unternehmen starten wir eine neue Beitragsreihe, in der wir eine Beispieldatenbank zur Erfassung von Bestellungen und zum Erstellen der Rechnungen programmieren – von der Erstellung des Datenmodells, über das Generieren von Beispieldaten und die Entwicklung von Beispielformularen, bis hin zu Berichten zur Ausgabe der Rechnungen. Und natürlich dürfen auch der Versand der Rechnungen und die Erfassung der Rechnungseingänge nicht fehlen.

Den Start zu dieser Beitragsreihe machen wir mit dem Beitrag **Rechnungsverwaltung: Datenmodell** ab Seite 15. Hier stellen wir die Tabellen vor, die zur Erfassung von Kunden, Produkten und Bestellungen samt Bestellpositionen benötigt werden.

Einige dieser Tabellen könnten das Vorhandensein eines zusätzlichen Feldes neben dem Primärschlüsselfeld zur Erfassung von Nummern wie Kundennummer, Produktnummer oder Bestellnummer erfordern. Während das Primärschlüsselfeld vorrangig dazu dient, die Datensätze eindeutig zu identifizieren und eine Möglichkeit zu bieten, die Datensätze der verschiedenen Tabellen einander über Beziehung zuzuordnen, haben Kundennummer, Produktnummer oder Bestellnummer andere Aufgaben – zum Beispiel das schnelle Auffinden eines Kundendatensatzes, wenn sich ein Kunde telefonisch oder per E-Mail mit einer Frage zu einer Bestellung meldet. Daher zeigen wir im Beitrag **Nummern für Bestellungen generieren** ab Seite 9, wie Sie solche Nummern auch mit speziellen Formaten wie K99000001 einfach generieren lassen können.

Beim Anlegen einer Bestellposition, die ein Produkt einer Bestellung zuordnet, wollen wir meist Daten wie Einzelpreis, Mehrwertsteuersatz und andere Werte individuell mit der Bestellposition abspeichern. Damit wir diese Daten nicht beim Anlegen einer jeden Position manuell übertragen müssen, automatisieren wir dies. Eine Möglichkeit dazu finden Sie im Beitrag **Bestellposition per Datenmakro ergänzen** ab Seite 2 vor. Hier zeigen wir, wie Sie diese Daten über ein Datenmakro hinzufügen, das automatisch beim Speichern des Datensatzes ausgelöst wird.

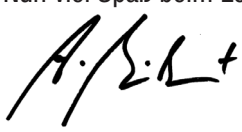
Bevor wir die Formulare zur Verwaltung der Kunden, Produkte und Bestellungen vorstellen, benötigen wir einige Beispieldaten. Wie Sie diese per Mausklick hinzufügen können, zeigen wir im Beitrag **Rechnungsverwaltung: Beispieldaten** ab Seite 25. Hier nutzen wir eine .NET-Bibliothek, um die Tabellen schnell mit Daten zu füllen.

Diese Daten nutzen wir dann bereits in der Bestellübersicht, die wir im Beitrag **Rechnungsverwaltung: Bestellübersicht** ab Seite 55 vorstellen. Hier zeigen wir die Programmierung eines Formulars zur Anzeige aller Bestellungen – mit umfangreichen Filterfunktionen!

Bei der Eingabe von Daten wie E-Mail-Adressen kann es sinnvoll sein, diese direkt zu validieren. So lassen sich allzu offensichtliche Fehler schnell aufdecken. Alle technischen Informationen zu diesem Thema finden Sie im Beitrag **E-Mail-Adressen validieren per VBA** ab Seite 69.

Schließlich zeigen wir im Beitrag **Bezeichnungsfelder im Griff** noch, welche Tücken Bezeichnungsfelder mit sich bringen (ab Seite 38) und wie Sie dafür sorgen können, dass der Benutzer beispielsweise in einem Feld für Postleitzahlen nur Zahlen eingeben kann – siehe **Textfeld nur mit bestimmten Zeichen füllen** ab Seite 46.

Nun viel Spaß beim Lesen!



Ihr André Minhorst

Bestellposition per Datenmakro ergänzen

Bestellpositionen speichern wir in einer eigenen Tabelle beispielsweise namens tblBestellpositionen, die als m:n-Verknüpfungstabelle zwischen Tabellen wie tblBestellungen und tblProdukte dient. Diese Tabelle nimmt dann jeweils noch Felder auf wie Einzelpreis, Mehrwertsteuersatz und Einheit, die wir aus der Produkte-Tabelle in die Bestellpositionen-Tabelle kopieren. Damit das automatisch beim Anlegen einer Bestellposition geschieht, fügen wir normalerweise ein Ereignis zum Eingabeformular für die Bestellpositionen hinzu, das diese Daten ausliest und in die Bestellposition einträgt. Es gibt jedoch noch eine Alternative: Dabei verwenden wir ein Datenmakro, das durch das Ereignis »Vor Änderung« des Datensatzes ausgelöst wird und verlegen die Logik damit in die Tabelle selbst. Wie das gelingt, zeigt der vorliegende Beitrag.

Beteiligte Tabellen

Die hier vorgestellte Lösung soll dazu dienen, die Daten aus mehreren Tabellen zum Zwecke der Archivierung direkt in die Tabelle **tblBestellpositionen** zu schreiben. Der Hintergrund ist, dass sich die Daten von Produkten wie Einzelpreis und Mehrwertsteuersatz sowie die Einheiten immer mal ändern können. Wenn wir diese Informationen dann nicht mit einer Bestellposition gespeichert haben und zu einem späteren Zeitpunkt beispielsweise die Umsätze für einen Zeitraum in der Vergangenheit untersuchen wollen, müssen wir die Daten aus den Tabellen **tblBestellpositionen**, **tblProdukte** und **tblMehrwertsteuersaetze** zusammensuchen. Das Problem dabei ist, dass die Daten in diesen Tabellen sich mittlerweile geändert haben könnten und wir nicht die in diesem Zeitraum tatsächlich gültigen Preise und Mehrwertsteuersätze berücksichtigen können.

Die an dieser Situation beteiligten Tabellen sehen Sie in Bild 1.

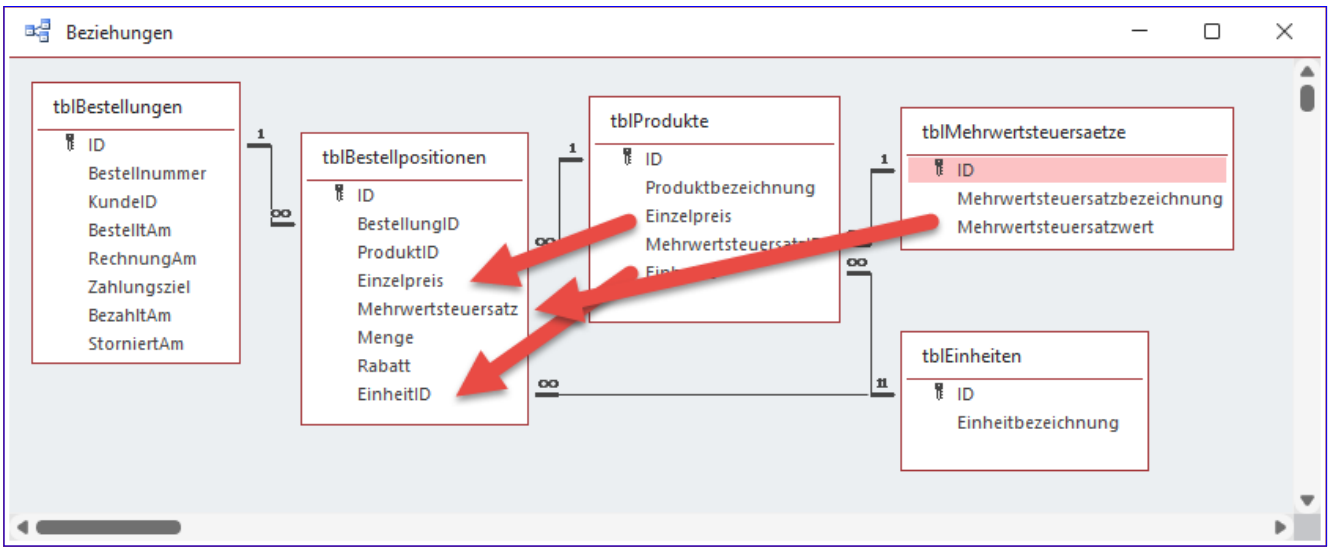


Bild 1: Beteiligte Tabellen dieser Lösung

Damit das geschilderte Problem nicht auftritt und wir zu jedem Zeitpunkt die tatsächlichen Umsätze für die jeweiligen Rechnungspositionen zusammentragen können, müssen wir die zu diesem Zeitpunkt gültigen Preise und Mehrwertsteuersätze irgendwo speichern. Dazu nutzen wir die Verknüpfungstabelle **tblBestellpositionen** und fügen dieser drei Felder namens **Einzelpreis**, **Mehrwertsteuersatz** und **EinheitID** hinzu, die wir beim Hinzufügen einer Bestellposition aus den verknüpften Tabellen füllen. Zusätzlich finden Sie hier noch das Feld **Rabatt**, das auch für jede Bestellposition individuell festgelegt werden kann.

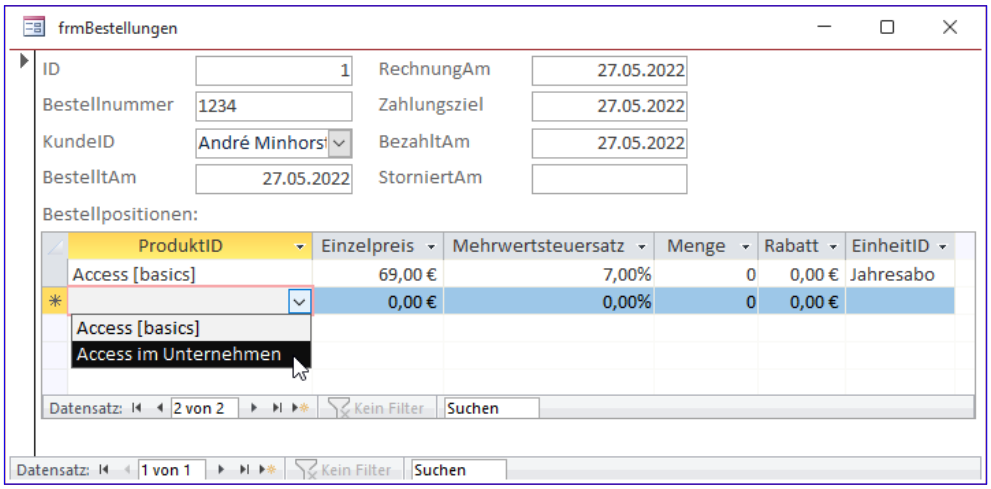


Bild 2: Formular zum Verwalten von Bestellungen und Bestellpositionen

Übliche Vorgehensweise

Normalerweise würden Sie für das Hinzufügen der genannten Daten zu einer Bestellposition ein Ereignis nutzen, das nach dem Auswählen des entsprechenden Produkts ausgelöst wird.

Dies geschieht in einem Formular etwa namens **frmBestellungen**, das an die Daten der Tabelle **tblBestellungen** gebunden ist. Dieses enthält wiederum ein Unterformular namens **sfmBestellungen**, das die Daten der Tabelle **tblBestellpositionen** anzeigt, und zwar in der Datenblattansicht (siehe Bild 2).

Dieses Unterformular zeigt alle Felder der Tabelle **tblBestellpositionen** mit Ausnahme des Primärschlüsselfeldes **ID** und des Fremdschlüsselfeldes **BestellungID** an. Die Daten des Unterformulars werden über dieses Fremdschlüsselfeld mit dem Feld **ID** des Datensatzes im übergeordneten Formular **frmBestellungen** verknüpft, sodass das Unterformular immer nur die zur aktuellen Bestellung gehörenden Bestellpositionen anzeigt. Durch diese Verknüpfung erhalten außerdem neu angelegte Bestellpositionen automatisch den Wert des Feldes **ID** der Bestellung im Hauptformular als Wert für das Fremdschlüsselfeld **BestellungID** der Tabelle **tblBestellpositionen**.

Damit nach der Auswahl eines neuen Produkts aus dem Nachschlagefeld **ProduktID** im Unterformular direkt die Daten aus den verknüpften Tabellen in den aktuellen Datensatz geschrieben werden, hinterlegen wir eine Ereignisprozedur für das Ereignis **Vor Aktualisierung** des Nachschlagefeldes. Dieses haben wir zu diesem Zweck in **cboProduktID** umbenannt.

Die Prozedur finden Sie in Listing 1. Sie deklariert eine **Database**-Objektvariable, die wir mit einem Verweis auf das aktuelle **Database**-Objekt füllen sowie eine Variable für ein **Recordset**-Objekt, das auf die Daten der Tabelle **tblProdukte** verweisen soll – hier genau auf den Datensatz, den der Benutzer mit dem Kombinationsfeld **cboProduktID** ausgewählt hat.

Nach dem Ermitteln dieses Datensatzes schreibt die Prozedur die Werte der Felder **Einzelpreis** und **EinheitID** dieses Datensatzes in die entsprechenden Felder des aktuell im Unterformular angezeigten Datensatzes der Tabelle **tblBestellpositionen**. Schließlich ermittelt sie noch mit einer **DLookup**-Funktion den **Mehrwertsteuer-**

```
Private Sub cboProduktID_BeforeUpdate(Cancel As Integer)
    Dim db As DAO.Database
    Dim rstProdukte As DAO.Recordset
    Dim curMehrwertsteuersatz As Currency
    Set db = CurrentDb
    Set rstProdukte = db.OpenRecordset("SELECT * FROM tblProdukte WHERE ID = " & Me!ProduktID, dbOpenDynaset)
    Me!Einzelpreis = rstProdukte!Einzelpreis
    Me!EinheitID = rstProdukte!EinheitID
    curMehrwertsteuersatz = DLookup("Mehrwertsteuersatzwert", "tblMehrwertsteuersaetze", "ID = " _
        & rstProdukte!MehrwertsteuersatzID)
    Me!Mehrwertsteuersatz = curMehrwertsteuersatz
End Sub
```

Listing 1: Ereignisprozedur beim Auswählen eines Produkts für eine Bestellposition

satzwert aus der Tabelle **tblMehrwertsteuersaetze** für die **MehrwertsteuersatzID** des gewählten Produkts.

Das Ergebnis dieser Abfrage landet in der Variablen **curMehrwertsteuersatzwert** und von dort im Feld **Mehrwertsteuersatz** der Bestellposition.

Wählt der Benutzer nun eines der Produkte aus, werden die Felder automatisch wie in Bild 3 mit den gewünschten Daten gefüllt.

Bevor wir nun die Alternative ausprobieren, sollten Sie die Ereignisprozedur **cboProduktID_BeforeUpdate** wieder entfernen oder zumindest auskommentieren – andernfalls bearbeiten wir die gleichen Felder durch zwei unterschiedliche Mechanismen und können die Ergebnisse nicht mehr sauber interpretieren.

Bestellpositionen erweitern per Datenmakro

Das hier beschriebene Verhalten wollen wir nun mit einem Datenmakro für die Tabelle **tblBestellpositionen** abbilden.

Die Datenmakros von Access arbeiten so ähnlich wie die Trigger, die Sie vielleicht vom SQL Server kennen. Dort gibt es die Möglichkeit, für bestimmte Aktionen wie das Anlegen, Ändern oder Löschen von Daten Ereignisse zu definieren, die wiederum Änderungen an Daten vornehmen oder andere Aktionen auslösen können.

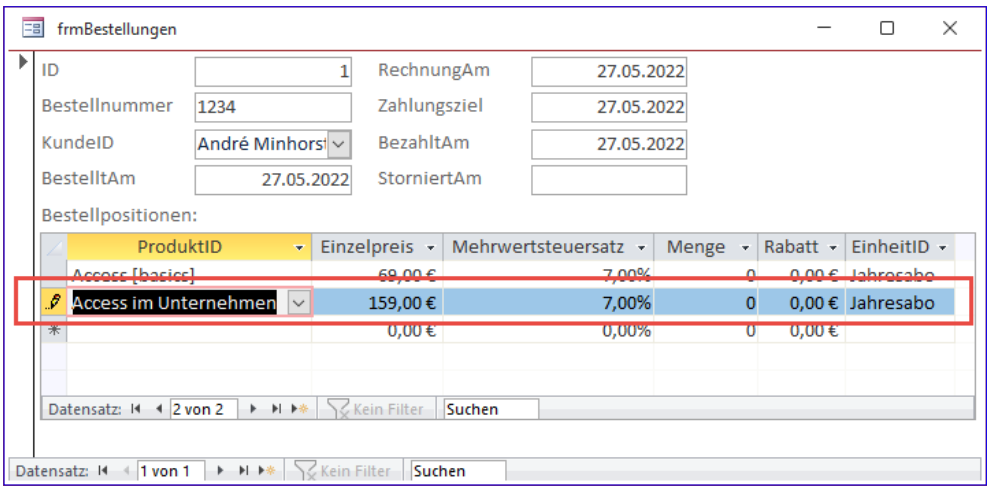


Bild 3: Ergänzte Daten einer Bestellposition

Access bietet gleich fünf solcher Ereignisse an, die wir hier nicht im Detail auflisten wollen – denn wir benötigen nur eines davon. Dieses heißt **Vor Änderung**.

Es gibt zwei Möglichkeiten, durch Ereignisse ausgelöste Datenmakros anzulegen. Die erste finden Sie in der Entwurfsansicht

Nummern für Bestellungen generieren

In vielen Beispieltabellen, die wir in diesem Magazin vorstellen, verwenden wir einfach das Primärschlüsselfeld als Kundennummer, Bestellnummer und so weiter. In manchen Fällen ist das nicht praktikabel, weil diese Nummern nach bestimmten anderen Regeln erstellt werden müssen. Dann bietet es sich an, dennoch ein Primärschlüsselfeld mit Autowertfunktion zu nutzen, um die Datensätze eindeutig zu identifizieren und dieses auch für das Herstellen von Beziehungen zu nutzen. Die Kundennummern oder Bestellnummern möchte man aber dennoch nicht von Hand eingeben, sondern die Datenbank soll das erledigen. Wie Sie das realisieren können, zeigt der vorliegende Beitrag.

Individuelle Nummern

Die Datenbankwelt wäre einfach, wenn man immer ein Primärschlüsselfeld mit Autowert als einziges eindeutiges Merkmal von Datensätzen nutzen könnte. Aber die Realität sieht anders aus. Manchmal gibt schon die Historie einer Anwendung vor, dass Kundennummern, Bestellnummern, Produktnummern und so weiter nach einem bestimmten Format generiert werden müssen, das nicht selten nicht nur Zahlen, sondern auch Buchstaben enthält.

Damit ist die Autowert-Funktion, die sich für die Ermittlung der Werte für Primärschlüsselfelder anbietet, überfordert – sie liefert nur Werte der Typen **Long Integer** oder **Replikations-ID** (sprich: GUID), und das im Falle der Zahlenwerte entweder aufsteigend oder zufällig.

Oft enthalten die verschiedenen Nummernkreise ein bestimmtes, aus Buchstaben bestehendes Kürzel, damit man direkt erkennen kann, ob es sich um eine Bestellnummer, eine Kundennummer oder andere Informationen handelt.

Wir sollten also in der Lage sein, mithilfe einer Funktion oder anderen technischen Möglichkeiten Werte für die gewünschten Anforderungen zu ermitteln.

Diese Anforderung könnte beispielsweise lauten, dass wir Werte benötigen, die mit dem Buchstaben **A** beginnen und danach aus acht Ziffern bestehen. Welche Herausforderungen können wir daraus ableiten?

Wir gehen davon aus, dass die Werte möglichst aufsteigend sein sollen, das heißt, dass der numerische Anteil so ermittelt wird, dass wir den bisher größten Wert herausfinden und diesem **1** hinzuaddieren. Auf **A00000001** soll also **A00000002** folgen. Das wäre alles kein Problem, wenn wir es mit einem Zahlenfeld zu tun hätten – wir würden dann einfach mit **DMax** den größten bisher vergebenen Wert ermitteln und diesen als Grundlage nutzen.

Ein möglicher Weg, dies zu umgehen, ist die Ableitung vom Primärschlüsselwert. Wie gesagt, wollen wir ein Primärschlüsselfeld mit Autowert als erstes eindeutiges Merkmal eines jeden Datensatzes nutzen, die zusätzliche Nummer (Kundennummer, Bestellnummer, ...) ist nur eine weitere eindeutige Information. Wir können unsere Nummer also auch zusammensetzen aus dem Buchstaben **A**, dem Primärschlüsselwert, und dazwischen eine Reihe Nullen, sodass wir das gewünschte Format von einer Länge von neun Zeichen erhalten. Beim Primärschlüsselwert **123** also beispielsweise **A** plus **00000** plus **123** gleich **A00000123**.

Wir schauen uns im Folgenden diesen Weg an und gehen von einer Kundennummer im oben genannten Format aus.

Kundennummer auf Basis des Primärschlüsselwertes

Im Beispiel verwenden wir eine Tabelle namens **tblKunden**, deren Primärschlüsselfeld namens **ID** als **Autowert**-

Feld definiert ist. Das Feld **Kundennummer** haben wir als eindeutiges Feld definiert (siehe Bild 1).

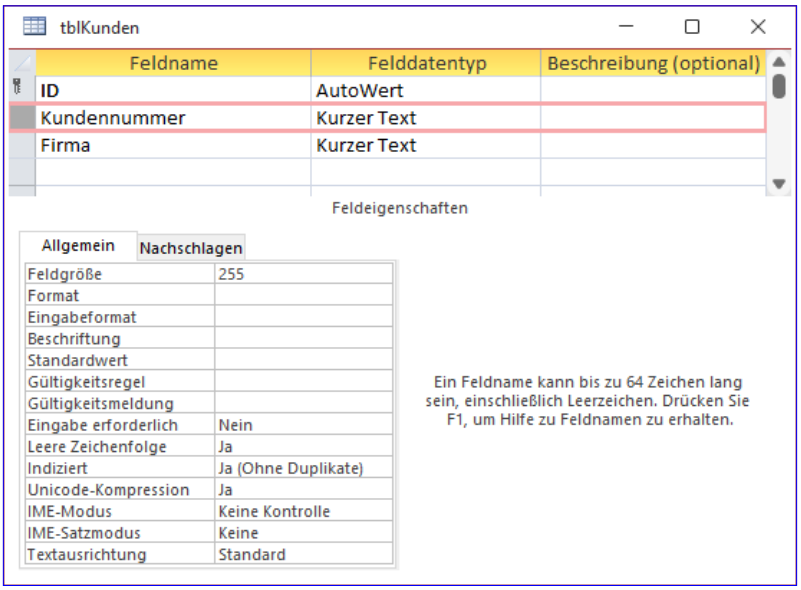


Bild 1: Einrichtung der Kundennummer als Feld mit einem eindeutigen Index

Nun wollen wir dafür sorgen, dass möglichst automatisch beim Anlegen eines neuen Datensatzes auch das Feld **Kundennummer** mit einem Wert gefüllt wird, der das oben beschriebene Format enthält.

Unsere erste Idee war, die **Format**-Funktion als Standardwert für das Feld **Kundennummer** zu nutzen. Dort wollten wir einen Eintrag wie den folgenden hinterlegen:

```
=Format([ID];"A00000000")
```

Wie wir schnell herausgefunden haben, funktioniert das nicht, weil man für das Zuweisen von Standardwerten einfach nicht auf die anderen Feldwerte zugreifen kann.

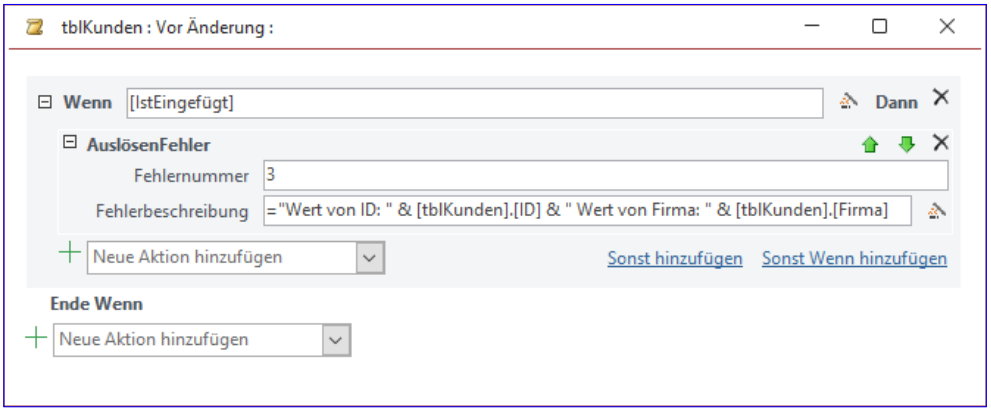


Bild 2: Makro, um herauszufinden, ob das Feld **ID** beim Ereignis **Vor Änderung** bereits gefüllt ist.

Kundennummer per Datenmakro

Also nutzen wir eines der Tabellenereignisse der Tabelle, in diesem Fall zunächst mit dem Ereignis **Vor Änderung**. Hier wollten wir prüfen, ob der Datensatz soeben angelegt wurde und dann aus dem Wert des Feldes **ID** die gewünschte Kundennummer zusammenstellen. Dies zeigte sich als falscher Ansatz, da das Feld **ID** zu diesem Zeitpunkt noch nicht gefüllt ist.

Um dies herauszufinden, haben wir für das Ereignis **Vor Änderung** das Datenmakro aus Bild 2 angelegt. Hier prüfen wir in einem ersten Schritt, ob es sich bei der Änderung um das Anlegen eines neuen Datensatzes handelt. In diesem Fall liefert **[IstEingefuegt]** den Wert **True** und die Aktionen innerhalb des **Wenn**-Konstrukts werden ausgeführt.

Hier haben wir, um eine Meldung zu generieren, die Aktion **AuslösenFehler** missbraucht. Diese soll einen Text ausgeben, der sowohl den Wert des Feldes **ID** als auch den des Feldes **Firma** enthält.

Wie das Ergebnis aus Bild 3 zeigt, wird zwar der neue Datensatz referenziert, aber das Feld **ID** ist zu diesem Zeitpunkt noch leer. Mit diesem Datenmakro können wir die Aufgabe also nicht lösen.

Da das Makro auch das Speichern des Datensatzes unterbindet, entfernen wir seinen Inhalt wieder.

Anschließend haben wir es mit dem Datenmakro für das Ereignis **Nach Einfügung** probiert. Hier haben wir zuerst die Makroaktion **NachschlagenDatensatz** verwendet, um den neuen Datensatz zu referenzieren. Diesen haben wir dann mit der Methode **DatensatzBearbeiten** bearbeitet, indem wir für das Feld **Kundennummer** einen Wert eingestellt haben, der aus dem Buchstaben **A** und dem Wert des Feldes **ID** des neuen Datensatzes besteht (siehe Bild 4).

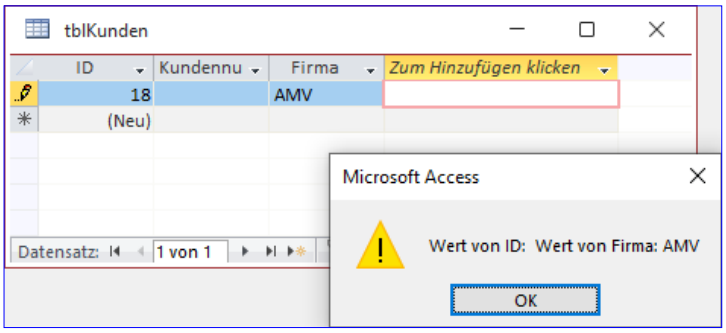


Bild 3: Ergebnis des Makros **Vor Änderung**

Das Ergebnis entspricht noch nicht ganz unseren Anforderungen, denn wir stellen im Feld **Kundennummer** ja zunächst nur den Buchstaben **A** mit dem neuen Wert für das Feld **ID** zusammen, also zum Beispiel **A1**, **A2** und so weiter (siehe Bild 5).

Wir wollen aber eigentlich die Kundennummer beispielsweise wie **A00000001** erhalten. Warum haben also nicht einfach die **Format**-Funktion verwendet? Ganz einfach: Die **Format**-Funktion steht in Datenmakros nicht zur Verfügung. Hier finden wir nur Einträge wie **FormatDatumZeit**, **FormatProzent**, **FormatWährung** und **FormatZahl**.

Also gehen wir einen kleinen Umweg und verwenden für den Parameter **Wert** in der Makroaktion **FestlegenFeld** den folgenden Ausdruck:

```
"A" & Rechts("0000000" & [rstKunden].[ID];8)
```

Dies fügt zunächst eine Zeichenfolge aus sieben Nullen (**0000000**) mit der **ID** zusammen. Wenn die **ID** noch einstellig ist, passt das Ergebnis gemäß unseren Anforderungen. Sobald sie aber zweistellig ist, erhalten wir eine Zeichenkette aus sieben Nullen und der **ID**, also beispielsweise **000000012**, was in diesem Fall neun Stellen entspricht. Hier schneiden wir die überflüssigen Nullen ab, indem wir mit der **Rechts**-Funktion nur die rechten acht Zeichen ermitteln. Stellen wir diesem noch den Buch-

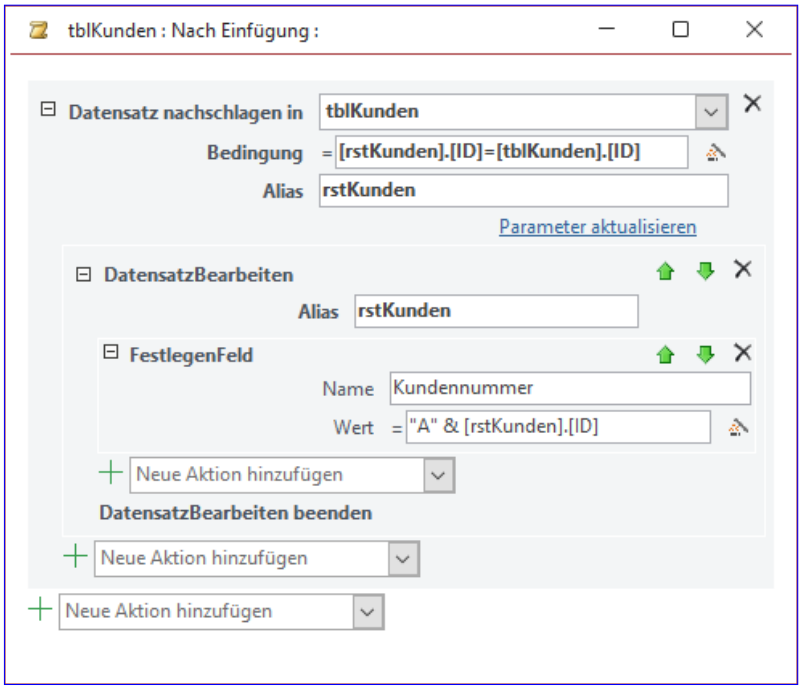


Bild 4: Das Datenmakro für das Ereignis **Nach Einfügung**

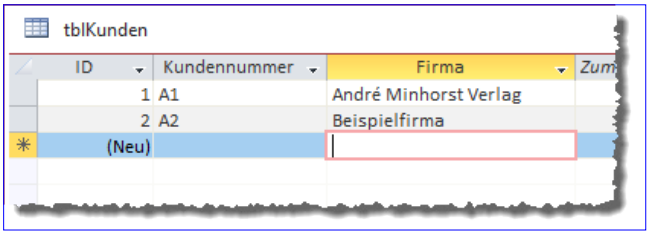


Bild 5: Ergebnis des Datenmakros

Rechnungsverwaltung: Datenmodell

In einer Beitragsreihe namens »Rechnungsverwaltung« wollen wir eine kleine Rechnungsverwaltung programmieren. Im ersten Teil kümmern wir uns um das Datenmodell der Rechnungsverwaltung und zeigen an einem Praxisbeispiel in einem weiteren Teil, wie Sie die im Beitrag »Beispieldaten generieren mit .NET und Bogus« (www.access-im-unternehmen.de/1359) vorgestellte Technik zum Erstellen von Beispieldaten einsetzen können. Das resultierende Datenmodell mit seinen Daten ist die Grundlage für weitere Beitragsteile, in denen wir Formulare zur Verwaltung der Rechnungen vorstellen sowie einen Rechnungsbericht erstellen, der gleich noch einen EPC-QR-Code zum schnellen Überweisen per Smartphone enthält. Außerdem schauen wir uns noch an, wie Sie mithilfe von Kontoumsätzen schnell abgleichen können, welche Rechnungen bezahlt sind.

Tabellen der Rechnungsverwaltung

In diesem ersten Teil der Beitragsreihe schauen wir uns die Tabellen an, die zum Speichern von Kunden, Produkten, Bestellungen, Bestelldetails und weiteren Informationen benötigt werden. Außerdem werfen wir einen Blick auf die Beziehungen zwischen diesen Tabellen.

Schließlich nutzen wir ein Tool, um die Tabellen mit Beispieldaten zu füllen, damit die anschließende Programmierung von Formularen und Berichten mit realistischen Daten erfolgen kann und Sie diese nicht von Hand eingeben müssen.

Tabellen zum Speichern der Kundendaten

Die Kundendaten an sich speichern wir in einer Tabelle namens **tblKunden**. Es gibt eine weitere Tabelle namens **tblAnreden**, die wir als Nachschlagetabelle für die Anrede des Kunden nutzen wollen.

Der Entwurf der Kundentabelle sieht wie in Bild 1 aus. Das Primärschlüsselfeld zum Speichern eines eindeutigen Index heißt **ID** und wird mit der Autowert-Funktion befüllt. Für die Kundennummer haben wir ein eigenes Feld vorgesehen, da es sein kann, dass Sie Kundennummern aus einem Onlineshop oder anderen Quellen übernehmen müssen. Für das Feld **Kundennummer** legen wir einen eindeutigen Index fest, damit auch dieses

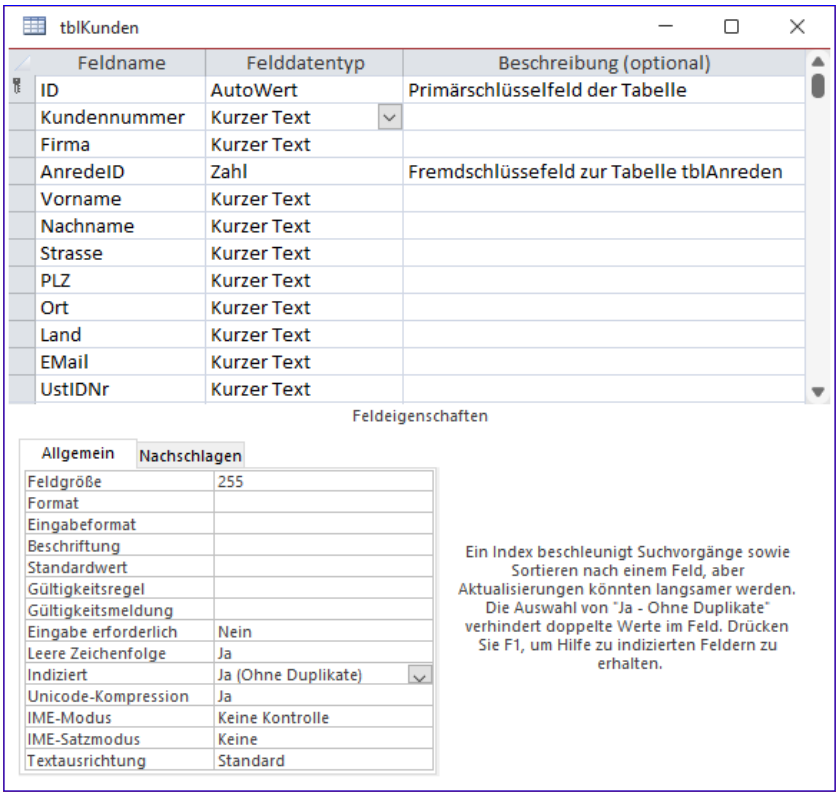


Bild 1: Die Tabelle **tblKunden**

eindeutig ist. Außerdem stellen wir den Felddatentyp auf **Kurzer Text** ein. Das Feld **AnredeID** ist ein Fremdschlüsselfeld mit Nachschlagefunktion für das Auswählen eines der Datensätze der Tabelle **tblAnreden**.

Für das Feld **PLZ** haben wir eine Feldgröße von **5** eingestellt. Damit kann der Benutzer nur fünf Zeichen für die PLZ eingeben.

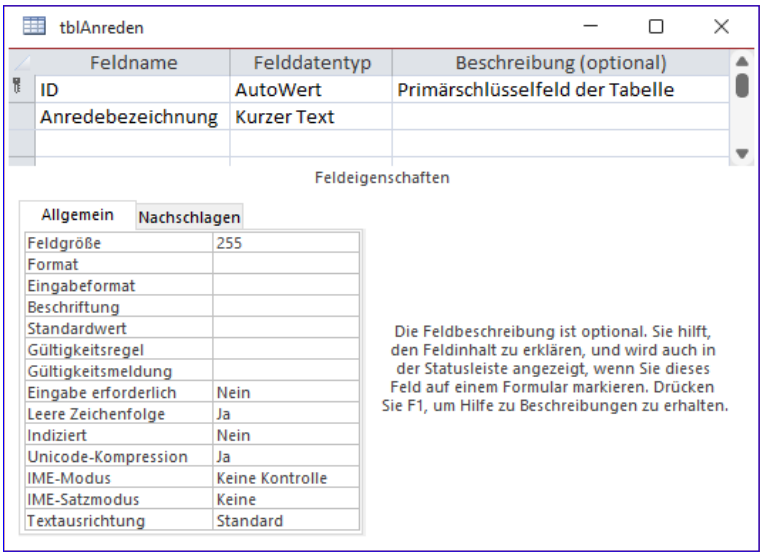


Bild 2: Die Tabelle **tblAnreden**

Tabelle zum Speichern der Anreden

Die Tabelle **tblAnreden** enthält, wie es meistens der Fall ist, nur zwei Felder – eines namens **ID** mit dem Primärschlüssel und eines für die Bezeichnung der Anrede (siehe Bild 2).

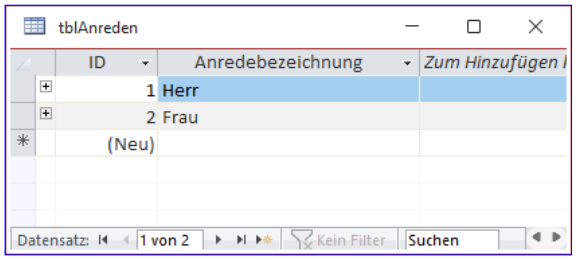


Bild 3: Die Tabelle **tblAnreden** in der Datenblattansicht

Die Beziehung zwischen den beiden Tabellen sowie die für die Auswahl der jeweiligen Anrede notwendigen Nachschlagefeld-Eigenschaften haben wir komfortabel mit dem Nachschlage-Assistent festgelegt, den Sie über die Auswahl des Datentyps **Nachschlage-Assistent** starten. Dort haben wir im letzten Schritt eingestellt, dass wir für die Beziehung die Datenintegrität aktivieren – das ist die neue Bezeichnung im Nachschlagefeld-Assistenten für referenzielle Integrität.

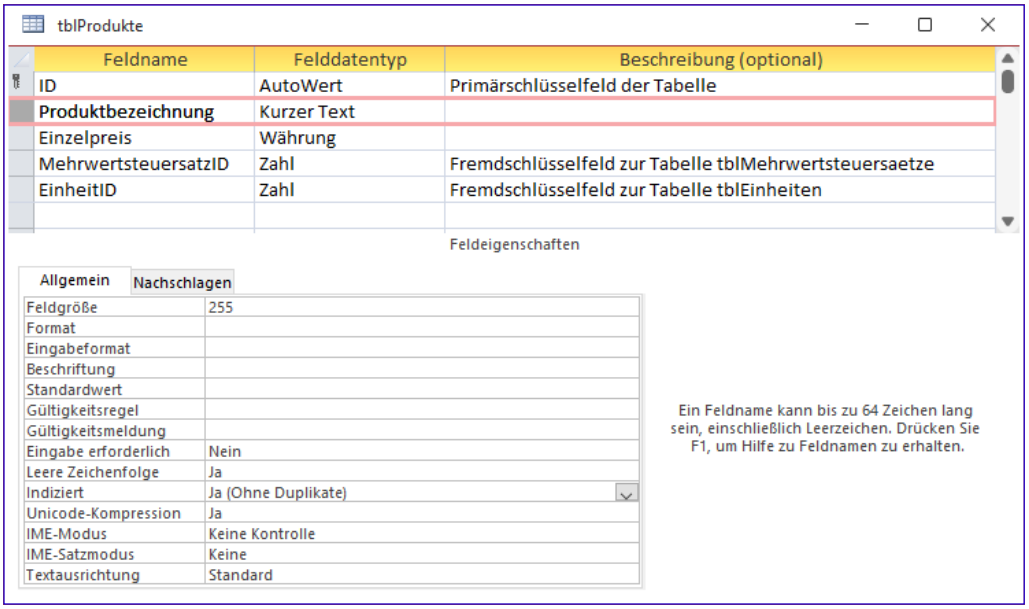


Bild 4: Die Tabelle **tblProdukte**

Die beiden Werte für die Tabelle **tblAnreden** tragen wir gleich ein, sodass wir diese nicht mehr mit dem Beispieldatengenerator füllen müssen. Das Ergebnis sieht schließlich wie in Bild 3 aus.

Tabelle zum Speichern der Produkte

Die Tabelle **tblProdukte** sieht in der Entwurfsansicht wie in Bild 4 aus. Sie enthält ein Primärschlüsselfeld namens **ID** sowie ein Feld namens **Produktbezeichnung**, das wir mit einem eindeutigen Index versehen. Auf diese Weise kann nicht versehentlich ein Produkt gleichen Namens zwei Mal angelegt werden.

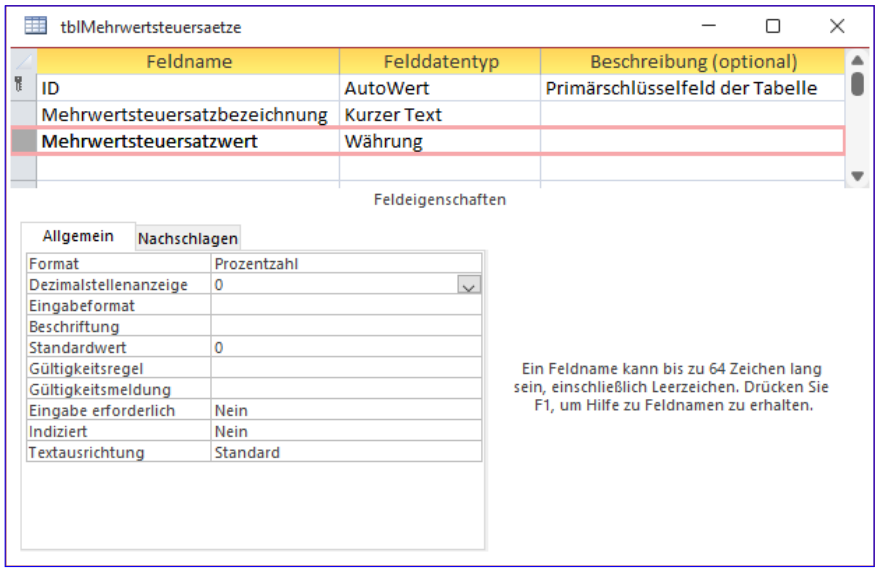


Bild 5: Die Tabelle **tblMehrwertsteuersaetze** in der Entwurfsansicht

Außerdem enthält die Tabelle noch ein Feld namens **Einzelpreis** und zwei Fremdschlüsselfelder. Das erste heißt **MehrwertsteuersatzID** und dient als Nachschlagefeld für die Datensätze der Tabelle **tblMehrwertsteuersaetze** und das zweite namens **EinheitID** erlaubt die Auswahl eines der Datensätze der Tabelle **tblEinheiten**.

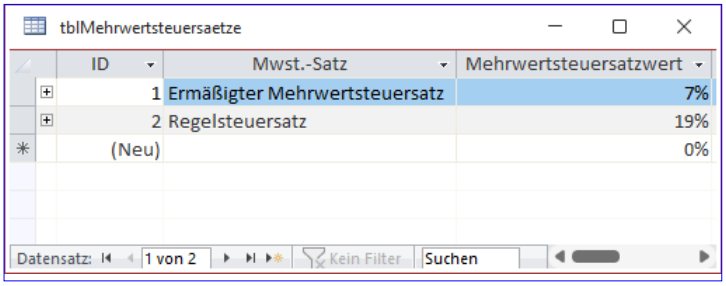


Bild 6: Die Tabelle **tblMehrwertsteuersaetze** in der Datenblattansicht

Die Tabelle **tblMehrwertsteuersaetze** enthält die Felder aus der Entwurfsansicht aus Bild 5. Sie enthält nicht nur ein Primärschlüsselfeld und ein Feld für den jeweiligen Mehrwertsteuersatz, sondern auch noch ein Feld mit der Bezeichnung des Mehrwertsteuersatzes. Für das Feld **Mehrwertsteuersatzwert** haben wir den Datentyp **Währung** festgelegt, weil dies die einzige Möglichkeit ist, die Genauigkeit des dahinter liegenden Datentyps **Decimal** abzubilden. Da das Feld Prozentsätze abbilden soll, haben wir allerdings die Eigenschaft **Format** auf **Prozentzahl** eingestellt. Da wir es in der Regel mit Prozentsätzen mit ganzen Prozentpunkten zu tun haben, also beispielsweise **7%** oder **19%**, stellen wir

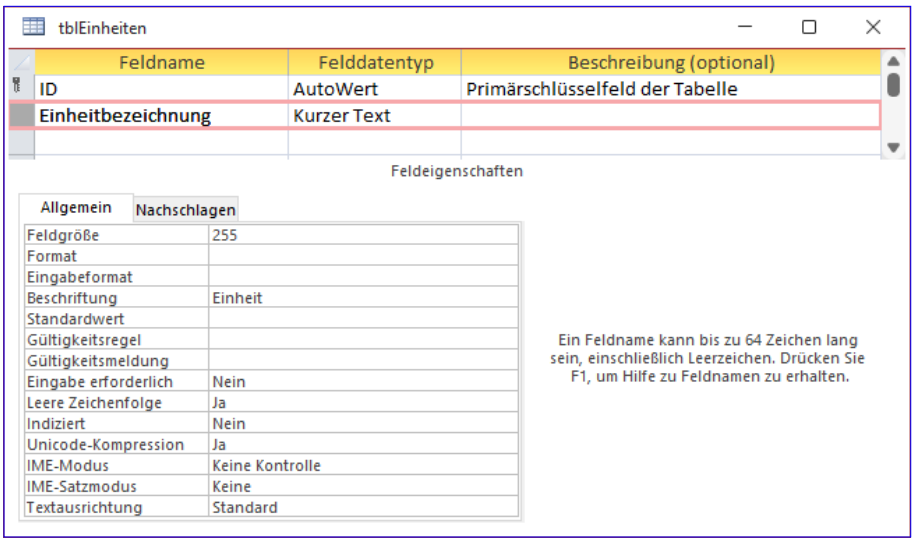


Bild 7: Die Tabelle **tblEinheiten** in der Entwurfsansicht

außerdem die Eigenschaft **Dezimalstellenanzeige** auf den Wert **0** ein.

Die Daten sehen nach der Eingabe wie in Bild 6 aus. Im Nachschlagefeld für das Feld **MehrwertsteuersatzID** in der Tabelle **tblProdukte** haben wir die beiden Felder **ID** und **Mehrwertsteuersatzwert** der Tabelle **tblMehrwertsteuersaetze** als Datensatzherkunft ausgewählt.

Tabelle zum Speichern der Einheiten

Die Tabelle **tblEinheiten** enthält nur die beiden Felder **ID** und **Einheitbezeichnung** (siehe Bild 7). Warum verwenden wir hier und in den vorhergehenden Tabellen gelegentlich den scheinbar unnötigen Zusatz **Bezeichnung** wie in **Einheitbezeichnung**? Vielleicht verwenden Sie die Tabellen einmal als Datenquelle für eine .NET-Anwendung unter Anbindung über das Entity Framework.

Dort würden Sie für die Tabelle **tblEinheiten** eine Klasse namens **Einheit** erstellen sowie eine Auflistung namens **Einheiten**. Wenn das Feld **Einheitbezeichnung** nun **Einheit** hieße, hätte es die gleiche Bezeichnung wie die Klasse, was jedoch nicht zulässig ist.

Wenn Sie sicher nicht vorhaben, jemals eine solche Anwendung auf dem Datenmodell aufzusetzen, dann können Sie auch mit Bezeichnungen wie **Einheit** oder **Mehrwertsteuersatz** arbeiten. Und falls es dann doch geschieht, können Sie das Problem immer noch durch ein geeignetes Mapping umgehen. Die Tabelle **tblEinheiten** füllen wir auch gleich mit ein paar Einträgen – siehe Bild 8.

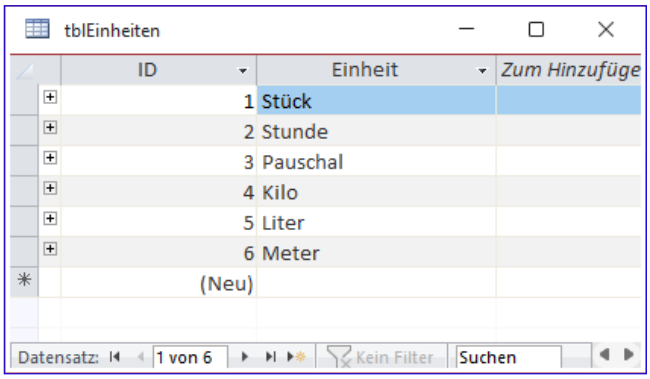


Bild 8: Die Tabelle **tblEinheiten** mit einigen Beispieldatensätzen

Tabelle zum Speichern der Bestellungen

Wir wollen jede Bestellung einem Kunden zuordnen, und außerdem soll jede Bestellung alle bestellten Produkte definieren. Das gelingt nicht in einer einzigen Tabelle. Wir erstellen also erst einmal eine Tabelle namens **tblBestellungen** und legen dort Informationen fest wie Bestellnummer, Datumsangaben (zum Beispiel Bestelldatum, Rechnungsdatum et cetera) und den Kunden, für den diese Bestellung erfasst wurde (siehe Bild 9).

Für das Feld **Bestellnummer**, das wir zusätzlich zum Primärschlüsselfeld **ID** nutzen, um individuelle Bestellnummern vergeben zu können, legen wir einen eindeuti-

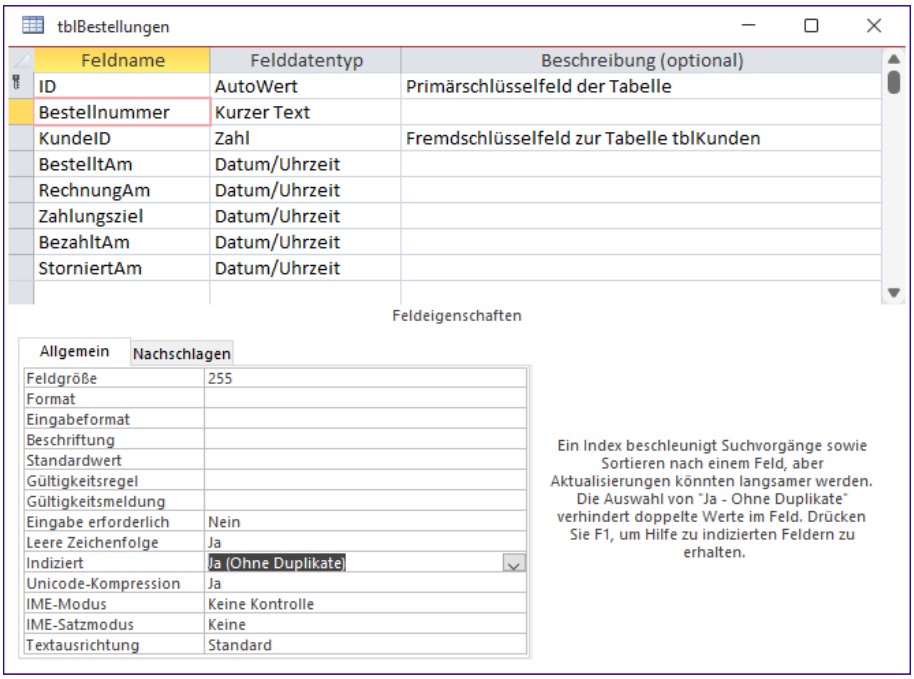


Bild 9: Die Tabelle **tblBestellungen** in der Entwurfsansicht

Rechnungsverwaltung: Beispieldaten

Nachdem wir im Beitrag »Rechnungsverwaltung: Datenmodell« das Datenmodell für die Rechnungsverwaltung definiert haben, könnten wir eigentlich zur Programmierung der für die Dateneingabe benötigten Formulare übergehen. Allerdings macht die Programmierung von Formularen deutlich mehr Spaß, wenn bereits einige Beispieldaten vorliegen und man direkt damit ausprobieren kann, ob die Formulare funktionieren. Als Hilfsmittel zum Erstellen der Beispieldaten verwenden wir das im Beitrag »Beispieldaten generieren mit .NET und Bogus« vorgestellte Werkzeug.

Vorbereitungen

Im Beitrag **Datenzugriff mit .NET, LINQPad und LINQ to DB** (**www.access-im-unternehmen.de/1358**) finden Sie alles, was Sie benötigen, um die Datenbank, die Sie mit Beispieldaten füllen wollen, von LINQPad aus zu referenzieren.

Im Beitrag **Beispieldaten generieren mit .NET und Bogus** (**www.access-im-unternehmen.de/1359**) finden Sie zusätzlich ausführliche Hinweise darauf, wie Sie die für die Verwendung des Tools **Bogus** notwendigen Schritte durchführen.

Deshalb hier nur noch einmal in aller Kürze die notwendigen Schritte:

- Installieren Sie das Tool **LINQPad**, mit dem Sie die vielen Bibliotheken nutzen können, die nur über .NET, aber nicht über VBA verfügbar sind.
- Installieren Sie **LINQ to DB** in **LINQPad**.
- Fügen Sie eine Verbindung zu der Datenbank hinzu, die Sie mit Beispieldaten füllen wollen. Sie sollten die Verbindungseigenschaften beispielsweise wie in Bild 1 ausfüllen.

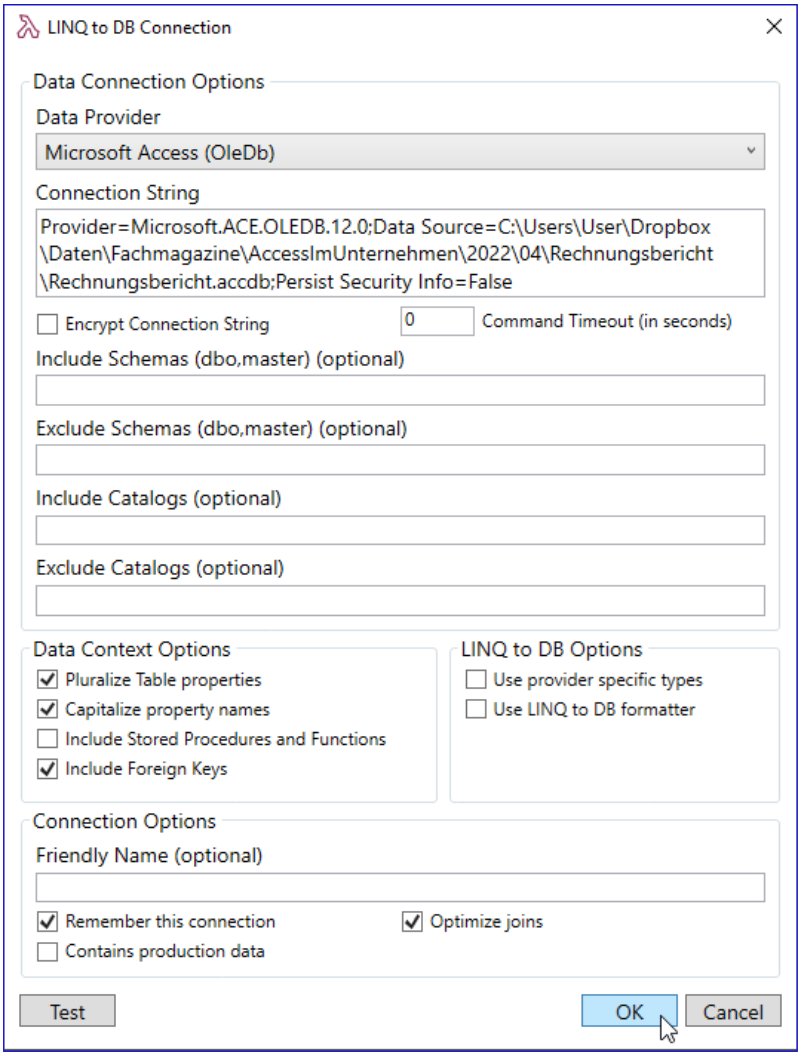


Bild 1: Verbindungseigenschaften für die Datenbank

- Danach sollte LINQPad die Tabellen wie in Bild 2 anzeigen.

- Damit können Sie eine neue Query zu LINQPad hinzufügen, was Sie über den Registerreiter mit dem **+**-Zeichen erledigen. Speichern Sie die Query unter dem Namen **Rechnungsbericht_Beispieldaten**.

- Wählen Sie unter **Connection** die soeben hinzugefügte Verbindung aus.

- Legen Sie unter **Language** den Wert **VB Program** fest.

- Fügen Sie wie im Beitrag **Beispieldaten generieren mit .NET und Bogus** (**www.access-im-unternehmen.de/1359**) beschrieben die Elemente hinzu, die für den Einsatz von Bogus notwendig sind.

Vorhandene Daten löschen

Wir wollen immer, wenn wir die Anwendung ausprobieren, einen frischen Satz von Daten bereitstellen. Eventuell bei einem vorherigen Test geänderte, hinzugefügte oder gelöschte Daten wollen wir wieder in den Urzustand versetzen – damit ermöglichen wir auch das automatisierte Testen von Teilen der Anwendung. Zum Löschen der Daten verwenden wir die Prozedur aus Listing 1. Diese rufen wir von der automatisch beim Hinzufügen der Query bereitgestellten Methode **Main** aus auf:

```
Sub Main
    DatenLoeschen
End Sub
```

Die Prozedur **DatenLoeschen** definiert das **UserQuery**-Objekt, über das wir auf die einzelnen Tabellen der Datenbank zugreifen können. Hier rufen wir für jede Tabelle einmal die **Execute**-Methode auf und übergeben jeweils eine **DELETE**-Abfrage, mit der wir die Daten der jeweiligen Tabelle löschen. Dies erledigen wir gleich für alle Tabellen der Datenbank, und zwar in einer Reihenfolge, in der zuerst die Daten aus den Tabellen gelöscht werden, die über Fremdschlüsselfelder mit anderen Tabellen verknüpft sind. Die Tabelle **tblBestellpositionen** ist gleich über zwei Fremdschlüsselfelder mit den Tabellen **tblBestellungen** und **tblProdukte** verknüpft, sodass wir diese zuerst leeren.

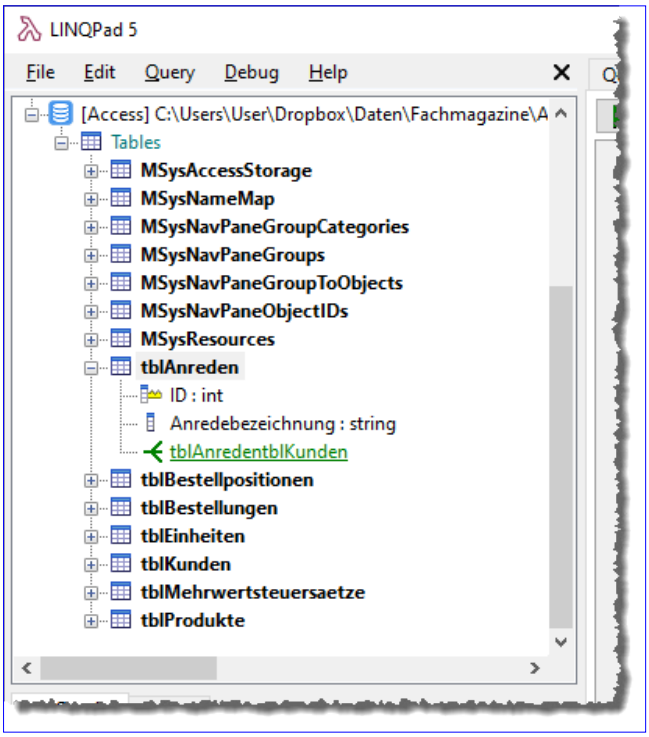


Bild 2: Die verfügbaren Tabellen

```
Public Sub DatenLoeschen
    Dim objUserQuery As UserQuery
    objUserQuery = tblAnreden.DataContext
    objUserQuery.Execute("DELETE FROM tblBestellpositionen")
    objUserQuery.Execute("DELETE FROM tblBestellungen")
    objUserQuery.Execute("DELETE FROM tblKunden")
    objUserQuery.Execute("DELETE FROM tblProdukte")
    objUserQuery.Execute("DELETE FROM tblAnreden")
    objUserQuery.Execute("DELETE FROM tblEinheiten")
    objUserQuery.Execute("DELETE FROM tblMehrwertsteuersaetze")
End Sub
```

Listing 1: Löschen der Daten

Danach folgt die Tabelle **tblBestellungen**, die mit der Tabelle **tblKunden** verknüpft ist. Die Tabelle **tblKunden** ist wiederum mit den Datensätzen der Tabelle **tblAnreden** verknüpft und wird als nächstes gelöscht. Die Tabelle **tblProdukte** greift auf Daten aus den Tabellen **tblEinheiten** und **tblMehrwertsteuersaetze** zu und muss folglich vor diesen Tabellen geleert werden. Wenn wir für die Beziehungen die Löschweitergabe definiert hätten, könnten wir die Löschvorgänge andersherum steuern und würden auch weniger **DELETE**-Anweisungen brauchen, weil ja die Inhalte verknüpfter Tabellen direkt mit gelöscht werden.

```
Public Sub AnredenAnlegen
    Dim objUserQuery As UserQuery
    objUserQuery = tblAnreden.DataContext
    objUserQuery.Insert(New tblAnreden With {.ID = 1, .Anredebezeichnung = "Herr"})
    objUserQuery.Insert(New tblAnreden With {.ID = 2, .Anredebezeichnung = "Frau"})
End Sub
```

Listing 2: Anlegen der Anreden

Anschließend beginnen wir damit, einige der Tabellen erneut zu füllen. Am einfachsten geht das mit den Tabellen, für die wir keine Zufallsdaten generieren müssen, sondern die wir direkt mit den gewünschten Daten füllen.

Da wäre als Erstes die Tabelle **tblAnreden**, die wir mit den bekannten Anreden füllen. Die dazu notwendige Prozedur finden Sie in Listing 2.

Auf ähnliche Weise füllen wir die Tabelle **tblEinheiten**, und zwar mit der Prozedur **EinheitenAnlegen** (siehe Listing 3).

Und auch die Tabelle **tblMehrwertsteuersaetze** füllen wir nach diesem Schema (siehe Listing 4).

```
Public Sub EinheitenAnlegen
    Dim objUserQuery As UserQuery
    objUserQuery = tblEinheiten.DataContext
    objUserQuery.Insert(New tblEinheiten With {.ID = 1, .Einheitbezeichnung = "Stück"})
    objUserQuery.Insert(New tblEinheiten With {.ID = 2, .Einheitbezeichnung = "Stunde"})
    objUserQuery.Insert(New tblEinheiten With {.ID = 3, .Einheitbezeichnung = "Pauschal"})
    objUserQuery.Insert(New tblEinheiten With {.ID = 4, .Einheitbezeichnung = "Kilo"})
    objUserQuery.Insert(New tblEinheiten With {.ID = 5, .Einheitbezeichnung = "Liter"})
    objUserQuery.Insert(New tblEinheiten With {.ID = 6, .Einheitbezeichnung = "Meter"})
    objUserQuery.Insert(New tblEinheiten With {.ID = 7, .Einheitbezeichnung = "Abonnement"})
End Sub
```

Listing 3: Anlegen der Einheiten

```
Public Sub MehrwertsteuersaetzeAnlegen
    Dim objUserQuery As UserQuery
    objUserQuery = tblMehrwertsteuersaetze.DataContext
    objUserQuery.Insert(New tblMehrwertsteuersaetze With {.ID = 1, .Mehrwertsteuersatzbezeichnung = _
        "Ermäßigter Mehrwertsteuersatz", .Mehrwertsteuersatzwert = 0,07})
    objUserQuery.Insert(New tblMehrwertsteuersaetze With {.ID = 2, .Mehrwertsteuersatzbezeichnung = _
        "Regelsteuersatz", .Mehrwertsteuersatzwert = 0,19})
End Sub
```

Listing 4: Anlegen der Mehrwertsteuersätze

Damit geht es nun an die wirklich interessanten Daten – nämlich die, welche wir mit dem Zufallsgenerator erstellen lassen. In welcher Reihenfolge gehen wir mit den übrigen Tabellen nun vor? Wir beginnen mit den Tabellen, die nur mit den bisher gefüllten Tabellen verknüpft sind. Hier bieten sich zunächst die Tabellen **tblProdukte** und **tblKunden** an.

Produkte per Zufallsgenerator füllen

Als Nächstes wollen wir also die Tabelle **tblProdukte** mit einigen Datensätzen füllen. Wir hätten gern 100 verschiedene Produkte, also fügen wir schon einmal den Aufruf der noch zu definierenden Prozedur **ProdukteAnlegen** mit dem Wert **100** als Parameter zur Methode **Main** hinzu:

```
Sub Main
    DatenLoeschen
    AnredenAnlegen
    MehrwertsteuersaetzeAnlegen
    EinheitenAnlegen
    ProdukteAnlegen(100)
End Sub
```

Die dadurch aufgerufene Prozedur finden Sie in Listing 5. Hier finden wir als Erstes den Parameter **intMenge** vor, dem wir die Anzahl der zu erzeugenden Produkte übergeben.

Die Prozedur greift über die Tabelle **tblProdukte** auf den **DataContext** der Datenbankverbindung zu. Mit einer Lambda-Funktion (eine Erläuterung würde den Rahmen sprengen) definieren wir die Regeln für das Anlegen der Beispieldaten. Als Produktbezeichnung wählen wir die Eigenschaft **ProduktName** der Klasse **Commerce**. Für den Preis nutzen wir die Eigenschaft **Price** der gleichen Klasse und geben hier den kleinsten und den größten möglichen Preis an sowie die Anzahl der Nachkommastellen. Wir erhalten hiermit Preise zwischen 10 und 100 EUR mit zwei Nachkommastellen.

Dann folgen die interessanten Elemente, nämlich die zufällige Auswahl von Einträgen der Tabelle **tblEinheiten**. Dies erreichen wir mit der Funktion **PickRandom**, der wir den Typ der Klasse mit den gewünschten Daten übergeben und für die wir anschließend einen Verweis auf die

```
Public Sub ProdukteAnlegen(intMenge As Int32)
    Dim Beispielprodukt As New tblProdukte
    Dim objUserQuery As UserQuery
    Dim i As Int32
    objUserQuery = tblProdukte.DataContext
    Dim objFaker As New Bogus.Faker(Of tblProdukte)("de")
    objFaker.Rules(Function(f, p)
        p.Produktbezeichnung = f.Commerce.ProductName
        p.Einzelpreis = f.Commerce.Price(10, 100, 2)
        p.EinheitID = f.PickRandom(Of tblEinheiten)(tblEinheiten).ID
        p.MehrwertsteuersatzID = f.PickRandom(Of tblMehrwertsteuersaetze)(tblMehrwertsteuersaetze).ID
    End Function)
    For i = 1 To intMenge
        Beispielprodukt = objFaker.Generate
        Beispielprodukt.ID = i
        Debug.Print("ID: " & i & " " & Beispielprodukt.ID.ToString & " " & Beispielprodukt.Produktbezeichnung)
        objUserQuery.Insert(Beispielprodukt)
    Next
End Sub
```

Listing 5: Zufallsgesteuertes Anlegen von Produkten

zu verwendende Auflistung **tblEinheiten** übergeben. Von dem jeweils gewählten Element möchten wir dem Feld **Einheit** des neuen Produkts die Eigenschaft **ID** der Tabelle **tblEinheiten** zuweisen.

Anschließend durchläuft die Prozedur eine **For...Next**-Schleife über die mit dem Parameter **intMenge** angegebene Anzahl von Elementen. Darin erstellen wir mit der **Generate**-Methode des **Faker**-Objekts ein neues Element und weisen seiner Eigenschaft **ID** den Wert der Laufvariablen zu. Anschließend geben wir zur Prüfung noch ein paar Eigenschaften aus, bevor wir das Element mit der **Insert**-Methode zur Tabelle **tblProdukte** hinzufügen.

Fehlermeldung beim Hinzufügen von Produkten

Nun wollen wir den Code ausprobieren, um die Produkte hinzuzufügen. Dazu ist es wie immer wichtig, dass die Datenbank geschlossen ist. Anderenfalls kann LINQPad nicht schreibend auf die enthaltenen Tabellen zugreifen.

Beim Aufrufen der Methode **Main** beispielsweise mit der Taste **F5** erhalten wir jedoch den Fehler aus Bild 3. Allerdings tritt dieser Fehler nicht immer auf – es passiert meist nach einer scheinbar willkürlich angelegten Anzahl von Produktdatensätzen.

Die Fehlermeldung weist darauf hin, dass ein eindeutiger Schlüssel verletzt wurde, also dass wir versuchen, einen bereits vorhandenen Wert nochmals in ein Feld mit einem eindeutigen Index einzufügen. Leider erhalten wir keinen Hinweis, um welches Feld es sich handelt. Also können wir nur selbst nachsehen und müssten dazu eigentlich die Datenbank öffnen – was nach sich ziehen würde, dass wir auch den Zugriff auf die Datenbank via LINQPad unterbre-

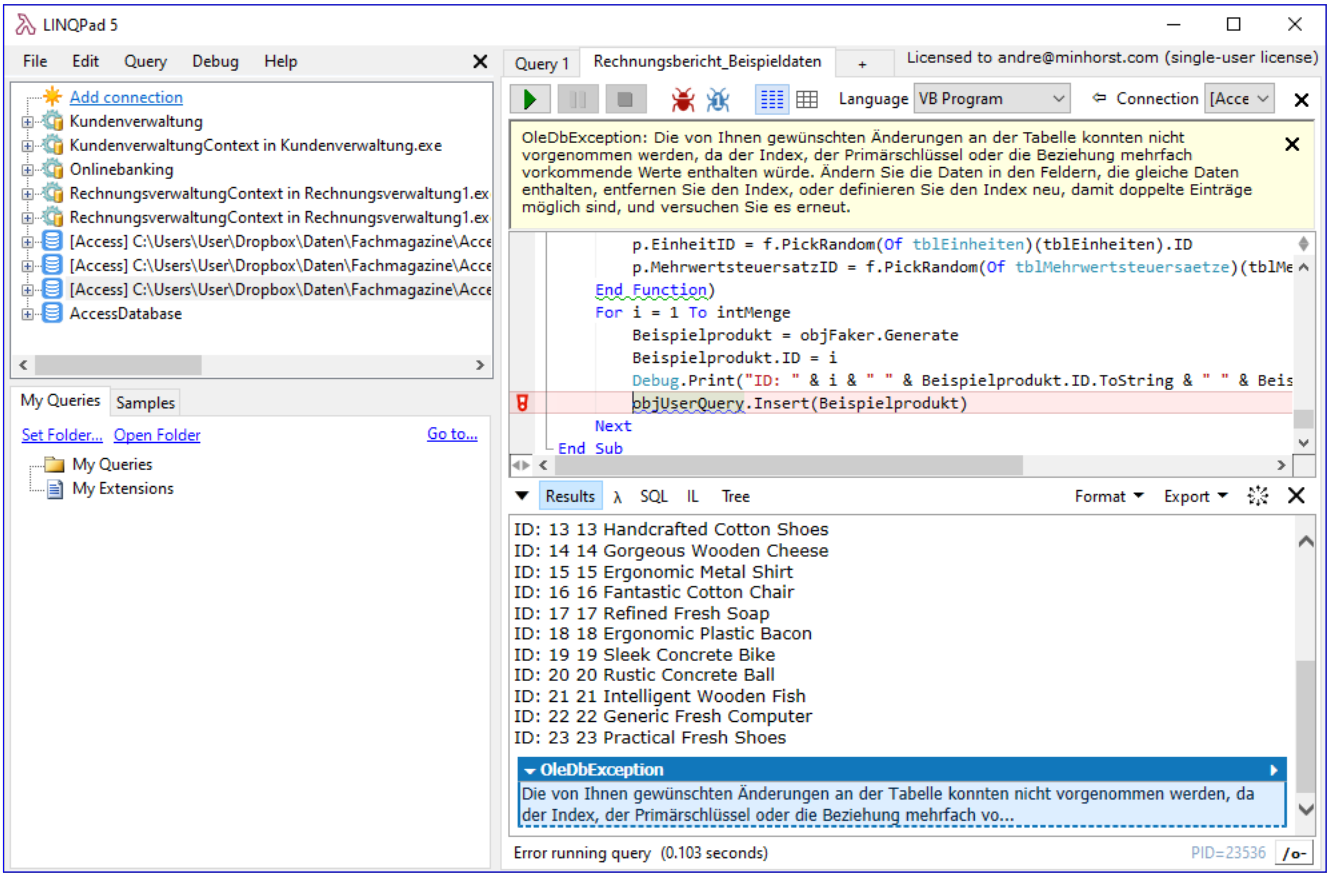


Bild 3: Fehlermeldung beim Hinzufügen von Produkten

```
For i = 1 To intMenge
    Beispielprodukt = objFaker.Generate
    Beispielprodukt.ID = i
    Debug.Print("ID: " & i & " " & Beispielprodukt.ID.ToString & " " & Beispielprodukt.Produktbezeichnung)
    Try
        objUserQuery.Insert(Beispielprodukt)
    Catch e As Exception
        Debug.Print("Fehler: " & e.Message)
        i = i - 1
    End Try
Next
```

Listing 6: Erweiterung um eine Fehlerbehandlung

chen müssen, was am zuverlässigsten geschieht, indem wir LINQPad schließen.

Allerdings haben wir bereits eine **Debug.Print**-Anweisung hinterlegt, welche die ID und den Produktnamen des anzulegenden Datensatzes ausgibt. Hier finden wir schnell heraus, dass es nicht an der ID liegt, sondern am Produktnamen – Bogus liefert schlicht keine eindeutigen Produktnamen und so kommt es früher oder später (oder auch mal gar nicht) dazu, dass ein Produktname doppelt angelegt wird.

Fehlerbehandlung mit Try...Catch

Die performanteste Möglichkeit, dies zu umgehen, ist das Ignorieren des Fehlers und das erneute Schreiben eines Produktdatensatzes im Fehlerfall. Dazu fügen wir eine **Try...Catch**-Anweisung zur Schleife hinzu, die wie in Listing 6 aussieht. Wir fügen die **Insert**-Methode in den **Try**-Block ein. Tritt hier ein Fehler auf, wird der **Catch**-Block ausgeführt.

In diesem Block geben wir, nur zur Information, die Fehlermeldung im Direktbereich von LINQPad aus. Außerdem setzen wir **i** um **1** zurück, damit die Prozedur einen neuen Anlauf starten kann, einen eindeutigen Produktnamen zu finden.

Wir könnten an dieser Stelle auch jeweils prüfen, ob der neu ermittelte Produktname bereits vorhanden ist, aber dies würde mit wachsender Anzahl vorhandener Produkte immer mehr Zeit in Anspruch nehmen. Die Methode mit **Try...Catch** scheint die performantere Möglichkeit zu sein.

Ergebnis in der Tabelle tblProdukte prüfen

Schließen wir nun LINQPad und öffnen wir die Tabelle **tblProdukte** der Datenbank, finden wir die gewünschten

tblProdukte

ID	Produktbezeichnung	Einzelprei	Mehrwert	EinheitID
1	Intelligent Cotton Fish	41,09 €	19,00%	Meter
2	Awesome Frozen Computer	14,54 €	19,00%	Stück
3	Rustic Wooden Chicken	96,01 €	19,00%	Liter
4	Fantastic Plastic Shoes	81,66 €	7,00%	Pauschal
5	Handmade Cotton Pizza	46,73 €	19,00%	Liter
6	Refined Steel Bacon	95,26 €	7,00%	Abonnement
7	Generic Wooden Car	97,48 €	19,00%	Abonnement
8	Intelligent Fresh Bike	76,14 €	7,00%	Meter
9	Tasty Frozen Soap	61,48 €	19,00%	Stunde
10	Ergonomic Cotton Computer	39,92 €	7,00%	Liter
11	Ergonomic Wooden Gloves	25,26 €	7,00%	Stunde
12	Intelligent Concrete Bike	58,95 €	19,00%	Pauschal
13	Handmade Cotton Shirt	87,22 €	7,00%	Kilo
14	Intelligent Frozen Soap	73,00 €	19,00%	Kilo
15	Incredible Concrete Sausages	89,66 €	19,00%	Meter
16	Ergonomic Plastic Hat	64,36 €	19,00%	Meter
17	Incredible Steel Soap	25,47 €	19,00%	Kilo
18	Unbranded Cotton Soap	19,12 €	19,00%	Meter
19	Sleek Concrete Bacon	66,61 €	19,00%	Stück
20	Refined Cotton Chicken	60,69 €	19,00%	Meter
21	Ergonomic Plastic Pants	73,61 €	7,00%	Kilo
22	Tasty Wooden Hat	60,64 €	19,00%	Kilo
23	Refined Steel Table	95,86 €	19,00%	Meter
24	Ergonomic Concrete Table	26,87 €	19,00%	Pauschal

Datensatz: 1 von 100 Kein Filter Suchen

Bild 4: Die Tabelle **tblProdukte** mit Beispieldaten

Bezeichnungsfelder im Griff

Bezeichnungsfelder oder auch Beschriftungsfelder sind ein wichtiger Bestandteil von Formularen und Berichten, denn sie geben in der Regel an, welche Daten der Benutzer in Steuerelemente eingeben kann oder dienen als Überschriften in Datenblättern oder Berichten in der Tabellenansicht. Beschriftungen für gebundene Felder lassen sich bereits im Tabellenentwurf festlegen, sodass das Anlegen dieser Felder in Formularen und Berichten ein Kinderspiel werden könnte. Wenn Sie allerdings noch wünschen, dass Beschriftungsfelder wie in Vorname: mit einem Doppelpunkt ausgestattet werden, müssen Sie eigentlich doch wieder jedes Beschriftungsfeld von Hand ändern. Außer natürlich, Sie lesen diesen Beitrag. Hier erfahren Sie nämlich alle Tricks rund um Beschriftungsfelder.

Frisch hinzugefügte Steuerelemente mit Bezeichnungsfeld

Wenn Sie ein Textfeld oder andere Steuerelemente wie Kombinationsfelder, Listenfelder, Kontrollkästchen et cetera anlegen, fügt Access automatisch ein Bezeichnungsfeld hinzu. Beim Einfügen eines solchen Steuerelements aus dem Ribbonbereich **EntwurflSteuerelemente** landet ein Bezeichnungsfeld links neben dem Steuerelement, welches eine Beschriftung wie **Text0** enthält (siehe Bild 1).

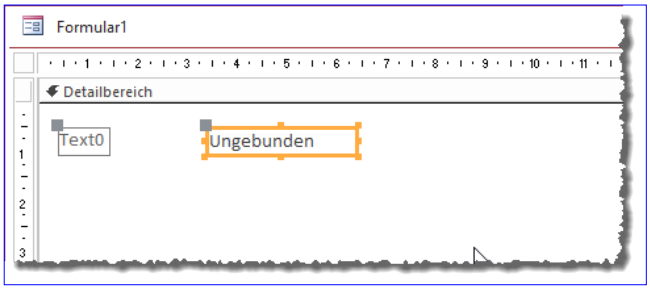


Bild 1: Ein einfaches Beschriftungsfeld

Bezeichnungsfelder für gebundene Steuerelemente

Wenn Sie hingegen in einem an eine Tabelle gebundenen Formular (wie hier durch Einstellen der Eigenschaft **Datensatzquelle** auf die Tabelle **tblKunden**) Felder aus der Feldliste in den Detailbereich der Entwurfsansicht ziehen, erhalten Sie immerhin schon mal die Feldnamen, die in der Tabelle definiert sind, als Texte in den Bezeichnungsfeldern (siehe Bild 2).

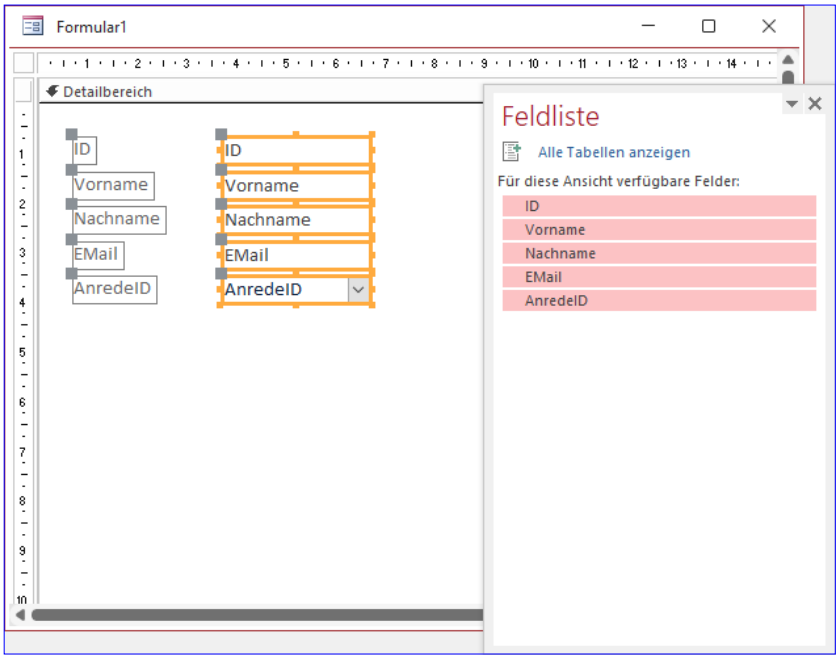


Bild 2: Gebundene Textfelder mit Beschriftungsfeldern

Daran gefallen uns noch zwei Dinge nicht: Wir hätten gern einen Doppelpunkt am Ende der Beschriftungen und außer-

dem sollen statt der Feldnamen **EMail** und **AnredeID** Texte wie **E-Mail** und **Anrede** verwendet werden. Der Benutzer sollte also nicht direkt erkennen, dass die Texte aus der Tabellendefinition stammen.

Nun können wir die Doppelpunkte manuell hinzufügen und auch die Beschriftungen können wir natürlich selbst an Ort und Stelle anpassen. Bei einer größeren Menge von Bezeichnungsfeldern und wenn die Felder gleich in mehreren Formularen und Berichten zum Einsatz kommen, erhalten Sie jedoch einiges an zusätzlicher Arbeit – Arbeit, die wir uns stark vereinfachen können.

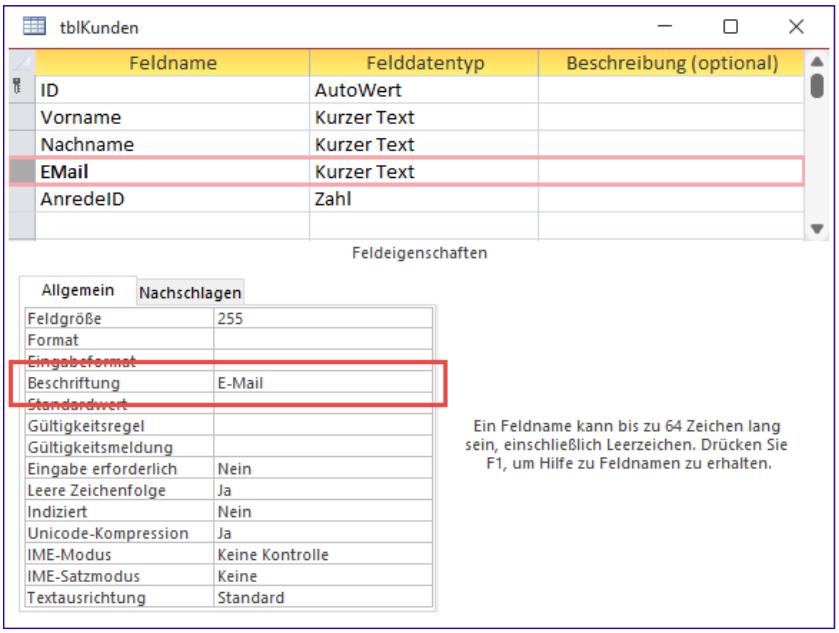


Bild 3: Zentrale Änderung der Beschriftung

Beschriftungen zentral anpassen

Der erste Trick ist, die Beschriftung für die Bezeichnungsfelder von gebundenen Steuerelementen nicht überall einzeln anzupassen, sondern diese nur an einer Stelle zu optimieren. Diese Stelle finden wir im Entwurf der jeweiligen Tabelle, in diesem Fall der Tabelle **tblKunden**.

Zeigen wir diese im Entwurf an und klicken auf ein Feld mit einer zu ändernden Beschriftung, wie hier **EMail**, dann sehen wir in den Eigenschaften den Eintrag **Beschriftung**. Hier tragen wir die Beschriftung ein, die wir in Formularen und Berichten sehen wollen, wenn wir dieses Feld dort hinzufügen, in diesem Fall **E-Mail** (siehe Bild 3).

Die gleiche Änderung nehmen wir auch noch für das Feld **AnredeID** vor, wo wir die Eigenschaft **Beschriftung** auf **Anrede** einstellen. Warum wollen wir nicht gleich noch den Doppelpunkt hinzufügen? Weil dieser gegebenenfalls nicht in jeder Beschriftung erscheinen soll. In einer Tabelle mit Daten in einem Bericht möchten Sie vielleicht nur die Beschriftungen in den Spaltenköpfen sehen, nicht aber einen Doppelpunkt. Für den Doppelpunkt gibt es eine andere Lösung.

Nach der Änderung der Eigenschaft **Beschriftung** für die betroffenen Felder speichern und schließen wir die Tabelle und fügen ihre Felder erneut zum Entwurf eines Formulars hinzu. Wie wir sehen, zeigt zwar die Feldliste noch die eigentlichen Feldnamen an, aber wenn wir diese in den Detailbereich ziehen, erscheinen die soeben eingestellten Beschriftungen für die Bezeichnungsfelder (siehe Bild 4).

Bild 4: Die neue Beschriftung wird in die Bezeichnungsfelder übernommen.

Doppelpunkte automatisch hinzufügen

Nun fehlen noch die Doppelpunkte. Hier können Sie in einem ersten Schritt dafür sorgen, dass Access die Bezeichnungsfelder für verschiedene Steuerelemente automatisch mit Doppelpunkten am Ende der Beschriftung ausstattet.

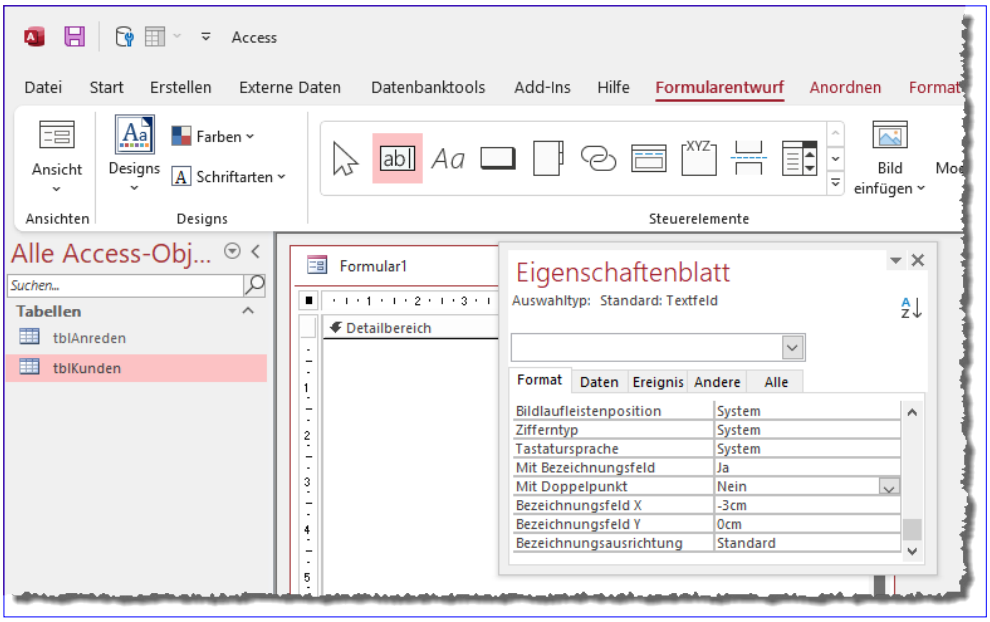


Bild 5: Einstellen der Eigenschaft **Mit Doppelpunkt**

Dazu entfernen wir die Steuerelemente aus dem vorherigen Beispiel nochmals. Nun wollen wir eine Einstellung anpassen, die nur zu einem bestimmten Zeitpunkt für ein Steuerelement zur Verfügung steht – nämlich nach dem Anklicken des hinzuzufügenden Steuerelements im Ribbonbereich **Formularentwurf\|Steuerelemente** und vor dem tatsächlichen Einfügen des Steuerelements im Formular oder Bericht. Sie klicken also beispielsweise die Schaltfläche für das Textfeld im Ribbon an und scrollen dann auf der Seite **Format im** Eigenschaftenblatt nach unten. Dort finden Sie die Eigenschaft **Mit Doppelpunkt** vor (siehe Bild 5).

Stellen Sie diese einmal auf **Ja** ein und klicken Sie in das Formular, um das Textfeld hinzuzufügen. Wie Sie sehen, enthält das Bezeichnungsfeld nun einen abschließenden Doppelpunkt.

Das Textfeld können wir nun wieder löschen und ziehen anschließend erneut die Felder aus der Feldliste in den Detailbereich des Formularentwurfs. Und siehe da: Die Bezeichnungsfelder enthalten nun alle automatisch einen Doppelpunkt. Eines allerdings wurde nicht berücksichtigt – das Bezeichnungsfeld für das Feld **AnredeID** (siehe Bild 6). Mehr dazu weiter unten.

Besonderes Eigenschaftenblatt

Vielleicht fragen Sie sich, was es mit dieser speziellen Art der Anzeige des Eigenschaftenblatts auf sich hat. Normalerweise zeigt das Eigenschaftenblatt immer den Namen des Bereichs oder Steuerelements, auf welches sich die Eigenschaften beziehen, in dem Kombinationsfeld über den Eigenschaften an. Dieses ist allerdings in unserem speziellen Fall leer (siehe Bild 7). Außerdem steht direkt

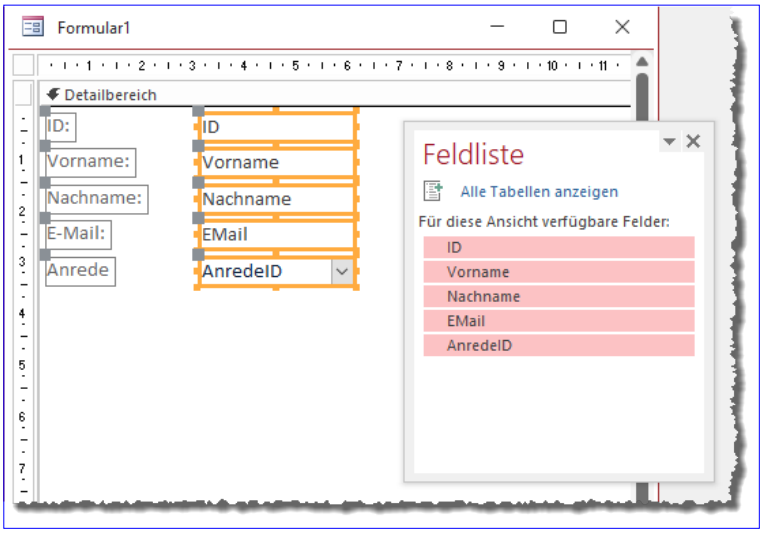


Bild 6: Alle Beschriftungsfelder außer das für das Kombinationsfeld erhalten nun einen Doppelpunkt.

unter der Überschrift **Eigenschaftenblatt** kein Text wie **Auswahltyp: Textfeld**, sondern **Auswahltyp: Standard: Textfeld**. Das heißt also, dass wir hier Standardeigenschaften für diesen Steuerelementtyp bearbeiten.

Diese Eigenschaften werden übrigens mit dem Formular gespeichert. Wenn Sie das Formular also anschließend nochmals in der Entwurfsansicht öffnen, um beispielsweise neue Felder hinzuzufügen, dann wirken diese Einstellungen immer noch.

Doppelpunkte für alle betroffenen Steuerelemente

Der Grund dafür, dass nicht die Bezeichnungsfelder aller Steuerelemente einen abschließenden Doppelpunkt erhalten, ist, dass dieses Feld nicht als Textfeld, sondern als Kombinationsfeld realisiert wurde. Wir haben die Eigenschaft **Mit Doppelpunkt** aber nur für Textfelder voreinstellt.

Also wiederholen wir den Vorgang zum Einstellen dieser Eigenschaft für das Kombinationsfeld. Beim anschließenden erneuten Einfügen aus der Feldliste erhält auch das Feld **AnredeID** einen Doppelpunkt im Bezeichnungsfeld.

Wenn das Formular nur Text- und Kombinationsfelder enthält, reicht es aus, die Eigenschaft **Mit Doppelpunkt** nur für diese beiden Steuerelemente zu aktivieren. Anderenfalls erledigen Sie dies auch für die übrigen verwendeten Steuerelemente, die ein Bezeichnungsfeld verwenden. Dies ist bei folgenden Steuerelementen der Fall:

- Textfeld
- Registersteuerelement
- Kombinationsfeld
- Listenfeld
- Kontrollkästchen

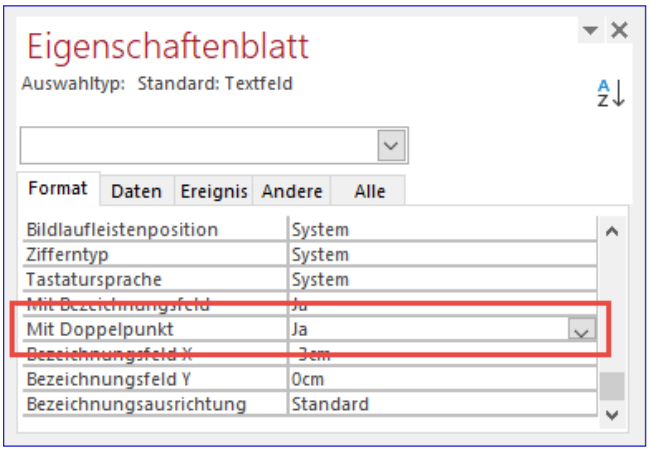


Bild 7: Ein besonderes Eigenschaftenblatt

- Anlagefeld
- Optionsfeld
- Unterformular/-bericht
- Gebundenes Objektfeld

Die folgenden beiden Steuerelemente weisen ebenfalls die Eigenschaft **Mit Doppelpunkt** auf, aber hier gilt diese für die Beschriftung des Elements selbst – es gibt kein separates Bezeichnungsfeld:

- Schaltfläche
- Umschaltfläche

Weitere Standardeigenschaften

Wenn wir schon bei den Eigenschaften von Steuerelementen sind, die nur über das **Standard**-Eigenschaftenblatt einstellbar sind, schauen wir uns auch gleich noch die übrigen Eigenschaften an.

Dabei handelt es sich um die folgenden:

- **Mit Bezeichnungsfeld**: Gibt an, ob beim Erstellen des Steuerelements überhaupt automatisch ein Bezeichnungsfeld hinzugefügt werden soll.

Textfeld nur mit bestimmten Zeichen füllen

Manche Felder in Tabellen dürfen nur bestimmte Zeichen aufnehmen. So soll beispielsweise eine Postleitzahl nur aus Zahlen bestehen, Namen sollen keine Zahlen enthalten, Telefonnummern nur Zahlen und bestimmte Zeichen wie Plus, Minus und Klammern. Um dies durchzusetzen, gibt es verschiedene Möglichkeiten. Die einfachste ist, nach der Eingabe zu prüfen, ob das Feld nur die zulässigen Zeichen enthält und den Benutzer darauf hinzuweisen. Man könnte aber auch direkt bei der Eingabe nur die zulässigen Zeichen akzeptieren. Dabei gibt es jedoch einige Fallstricke. In diesem Beitrag schauen wir uns die verschiedenen Möglichkeiten an.

Prüfung nach der Eingabe

Die einfachste Variante ist wohl die, direkt nach der Eingabe zu prüfen, ob der Text nicht erlaubte Zeichen enthält. Dazu müssen wir zunächst einmal definieren, welche Zeichen erlaubt sind und welche nicht. Ein einfaches Beispiel ist die Postleitzahl. Sie darf nur Zahlen enthalten, was leicht zu erreichen wäre, wenn man das Feld, an das man das Textfeld zur Eingabe bindet, direkt als Zahlenfeld definieren würde.

Postleitzahlen enthalten aber auch mal führende Nullen, und die werden eben nur bei Verwendung eines Felddatentyps wie **Kurzer Text** gespeichert. Es geht zwar auch mit einem Zahlenfeld und bestimmten Formatierungen, aber spätestens beim Export in eine Textdatei, etwa zur Übergabe von Adressen, werden standardmäßig nur die tatsächlich gespeicherten Werte weitergereicht.

Wir haben eine kleine Tabelle namens **tblBeispielfelder** erstellt, die drei Felder namens **PLZ**, **Telefon** und **EMail** für uns als Spielmaterial enthält. Diese binden wir über die Eigenschaft **Datensatzquelle** an ein neues Formular.

Wir definieren als Erstes eine Konstante, die alle zulässigen Zeichen enthält:

```
Const cStrZahlen As String = "0123456789"
```

Dann hinterlegen wir für das Ereignis **Vor Aktualisierung** des Textfeldes **txtPLZ**, das an das Feld **PLZ** gebunden ist, die Ereignisprozedur aus Listing 1. Diese liest den Inhalt des Textfeldes **txtPLZ** in die Variable **strPLZ** ein und ersetzt einen eventuellen Nullwert durch eine leere Zeichenkette. Dann untersucht sie in einer **For... Next**-Schleife alle Buchstaben der Zeichenkette **strPLZ**.

```
Private Sub txtPLZ_BeforeUpdate(Cancel As Integer)
    Dim i As Integer
    Dim strPLZ As String
    strPLZ = Nz(Me!PLZ, "")
    For i = 1 To Len(strPLZ)
        If InStr(1, cStrZahlen, Mid(strPLZ, i, 1)) = 0 Then
            MsgBox "Die PLZ darf nur aus Zahlen bestehen.", vbOKOnly + vbExclamation, "Ungültige PLZ"
            Cancel = True
        End If
    Next i
End Sub
```

Listing 1: Prüfen der PLZ nach der Eingabe

Dabei ermittelt sie den aktuellen Buchstaben mit **Mid(strPLZ, i, 1)** und prüft dann mit der **InStr**-Funktion, ob dieser in der Zeichenkette aus **cStrZahlen** enthalten ist. Ist das nicht der Fall, liefert **InStr** die Position, die gleich **0** ist. Dann erscheint eine entsprechende Meldung und die Eingabe wird mit **Cancel = True** abgebrochen – der Benutzer kann das Feld erst verlassen, wenn die Prüfung erfolgreich ist (siehe Bild 1). Dazu kann das Feld auch vollständig geleert werden.

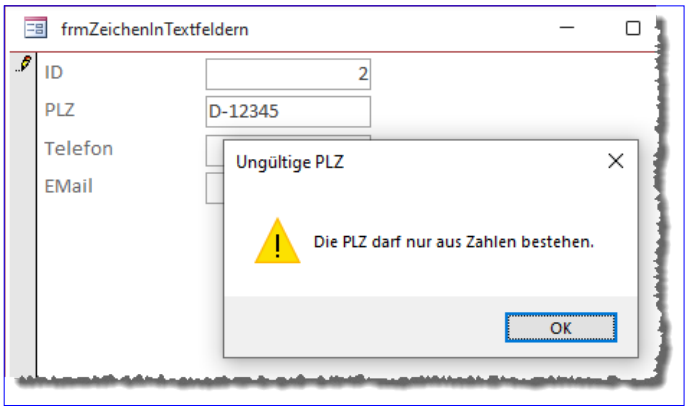


Bild 1: Meldung nach dem **Vor Aktualisierung**-Ereignis

Prüfung während der Eingabe

Die alternative Variante ist, die Zeichen direkt bei der Eingabe zu prüfen. Dazu benötigen wir eine andere Ereignisprozedur, nämlich **Bei Taste ab**. Vorab ein Hinweis: Damit decken wir nicht alle Fälle ab. Dazu jedoch später mehr.

Da wir auch erst einmal die Eingabe der Postleitzahl beschreiben wollen, kopieren wir das Feld **txtPLZ** in ein neues Textfeld namens **txtPLZ2**. Für dieses hinterlegen wir dann die Ereignisprozedur **txtPLZ2_KeyDown**. Was können wir mit dem Ereignis **Bei Taste ab** erledigen?

Die dadurch ausgelöste Prozedur **txtPLZ2_KeyDown** ist so aufgebaut:

```
Private Sub txtPLZ2_KeyDown(KeyCode As Integer, _
                            Shift As Integer)

End Sub
```

Hier sehen wir zwei Parameter:

- **KeyCode**: Liefert einen Zahlenwert, der die betätigte Taste repräsentiert.
- **Shift**: Gibt an, ob eine der Tasten **Umschalt**, **Alt** oder **Strg** gedrückt ist.

KeyCode liefert aber nicht nur einen Zahlenwert für die aktuell angeklickte Taste. Sie können durch Einstellen von **KeyCode** auf den Wert **0** auch dafür sorgen, dass der Tastenanschlag nicht an das aktuelle Steuerelement weitergeleitet wird. Und Sie können sogar festlegen, dass statt des gedrückten Zeichens ein anderes Zeichen weitergegeben wird. Für uns ist allerdings nur interessant, die Eingabe eines nicht gewünschten Zeichens zu unterbinden, indem wir **KeyCode** auf **0** einstellen.

Wie finden wir heraus, welcher **KeyCode** welchem Zeichen entspricht, um entsprechend darauf reagieren zu können? Ganz einfach: Wir geben den **KeyCode** einfach im Direktbereich aus.

```
Private Sub txtPLZ2_KeyDown(KeyCode As Integer, _
                            Shift As Integer)
    Debug.Print KeyCode
End Sub
```

Damit erfahren wir durch Eingeben von Zeichen in das Feld **txtPLZ2** schnell, dass die Zahlen von **0** bis **9** den Werten **48** bis **57** für **KeyCode** entsprechen. Wir können nun in einem ersten Ansatz dafür sorgen, dass der Benutzer nur noch diese Zeichen in das Textfeld eingeben kann. Dazu prüfen wir, ob **KeyCode** einen Wert von **48** bis **57** enthält und falls nicht, geben wir mit dem Parameter **KeyCode** den Wert **0** zurück:

```
Private Sub txtPLZ2_KeyDown(KeyCode As Integer, _
                            Shift As Integer)
```

```
    Select Case KeyCode
        Case 48 To 57
        Case Else
            KeyCode = 0
    End Select
End Sub
```

Das funktioniert genau wie angenommen: Nur die Tasten **0** bis **9** werden weitergeleitet und im Textfeld ausgegeben.

Wir haben allerdings ein kleines Problem: Auch die Tasten zum Bewegen der Einfügemarke, die Eingabetaste, die Tabulatortaste et cetera werden innerhalb des Textfeldes blockiert. Also geben wir auch solche Tasten wieder frei.

Wie Sie herausfinden, welche KeyCodes hinter den Tasten stecken, haben wir ja bereits beschrieben. Der Übersicht halber tragen wir die zusätzlichen Elemente in eine weitere **Case**-Bedingung ein:

- **8**: Back
- **9**: Tabulator
- **13**: Eingabetaste
- **27**: Escape
- **37**: Cursor nach links
- **39**: Cursor nach rechts
- **46**: Entfernen
- **113**: F2 (zum Aufheben von Markierungen)

Nummernblock berücksichtigen

Gegebenenfalls verwenden Benutzer auch den Nummernblock der Tastatur, um Zahlen einzugeben. Das müssen wir berücksichtigen, denn die **KeyCode**-Werte entsprechen nicht denen der normalen Zahlentasten, sondern den Zahlen von **96** bis **105**. Wir fügen diese dem **Case**-Zweig für die normalen Zahlen hinzu:

```
        Case 48 To 57, 96 To 105
```

Die Eingabetaste im Nummernblock hat allerdings auch den **KeyCode**-Wert **13**, sodass hier keine Erweiterung notwendig ist.

Die vollständige Prozedur sieht nun wie folgt aus:

```
Private Sub txtPLZ2_KeyDown(KeyCode As Integer, 7
                                    Shift As Integer)
    Select Case KeyCode
        Case 48 To 57, 96 To 105
        Case 8, 9, 13, 27, 37, 39, 46, 113
        Case Else
            KeyCode = 0
    End Select
End Sub
```

Für bessere Lesbarkeit können wir auch die Konstanten der Enumeration **KeyCodeConstants** nutzen.

Bei der Gelegenheit fällt uns noch eine Ergänzung ein: Aktuell lassen wir alle Betätigungen der Tasten von **0** bis **9** zu – und wir prüfen noch nicht, ob der Benutzer dabei eine der Tasten **Umschalt**, **Strg** oder **Alt** drückt. Das heißt, dass der Benutzer die Zahlen auch bei gedrückter Umschalt-Taste eingeben kann, was die Eingabe von Ausrufezeichen, Anführungszeichen und so weiter erlaubt. Wir müssen also im **Case**-Zweig für die Zahlen noch prüfen, ob der Parameter **Shift** den Wert **0** enthält, und falls nicht, das eingegebene Zeichen abfangen.

Dann sieht die Prozedur wie in Listing 2 aus.

Einfügen über die Zwischenablage

Der Fall, den wir mit dieser Methode nicht verarbeiten können, ist das Einfügen von Texten über die Zwischenablage. Das heißt, der Benutzer hat beispielsweise den

```
Private Sub txtPLZ2_KeyDown(KeyCode As Integer, Shift As Integer)
    Select Case KeyCode
        Case vbKey0 To vbKey9, vbKeyNumpad0 To vbKeyNumpad9
            If Not Shift = 0 Then
                KeyCode = 0
            End If
        Case vbKeyBack, vbKeyTab, vbKeyReturn, vbKeyEscape, vbKeyLeft, vbKeyRight, vbKeyDelete, vbKeyF2
        Case Else
            KeyCode = 0
    End Select
End Sub
```

Listing 2: Prüfen der PLZ nach der Eingabe mit **KeyCode**-Konstanten

Text **D-12345** in die Zwischenablage kopiert, markiert dann das Textfeld **txtPLZ2** und betätigt entweder die Tastenkombination **Strg + V** oder den Kontextmenübefehl **Einfügen**.

Als Erstes fällt auf: Wir haben die Tastenkombination **Strg + V** noch gar nicht freigeschaltet. Das ist etwas aufwendiger, da wir auch noch prüfen müssen, ob die **Strg**-Taste gleichzeitig gedrückt wird. Ob die **Strg**-Taste gedrückt wird, ermitteln wir wie bei der **Umschalt**-Taste und der **Alt**-Taste über den Parameter **Shift**. **Shift** kann die folgenden Werte annehmen, die auch kombiniert werden können:

- **0**: Keine Taste gedrückt.
- **1**: **Umschalt**-Taste
- **2**: **Strg**-Taste
- **3**: **Umschalt**- und **Strg**-Taste (**1** plus **2**)
- **4**: **Alt**-Taste
- **5**: **Umschalt**- und **Alt**-Taste (**1** plus **4**)
- **6**: **Strg**- und **Alt**-Taste (**2** plus **4**)
- **7**: **Umschalt**-, **Strg**- und **Alt**-Taste (**1** plus **2** plus **4**)

Auch hier gibt es entsprechende Konstanten. Allerdings verwenden wir nicht die der Enumeration **KeyCodeConstants**, sondern entsprechende Access-Konstanten:

- **Umschalt**-Taste: **acShiftMask**
- **Strg**-Taste: **acCtrlMask**
- **Alt**-Taste: **acAltMask**

Die Betätigung der Tastenkombination lassen wir also mit dem folgenden **Case**-Zweig in der **Select Case**-Bedingung zu:

```
Case vbKeyC
    If Shift = acCtrlMask Then
    Else
        KeyCode = 0
    End If
```

Beim Betätigen des Buchstaben **c** prüfen wir also, ob der Benutzer dabei die **Strg**-Taste (**acCtrlMask**) betätigt hat. Falls dies nicht in Kombination mit der **Strg**-Taste geschehen ist, wird dies mit **KeyCode = 0** unterbunden.

Aber was tun wir, wenn der Benutzer einen Text mit **Strg + V** einfügt? Wir könnten natürlich an dieser Stelle den Inhalt der Zwischenablage abgreifen und untersuchen. Aber wie machen es uns ein wenig einfacher. Die

Rechnungsverwaltung: Bestellübersicht

Im Beitrag »Rechnungsverwaltung: Bestellformular« (www.access-im-unternehmen.de/1382) der kommenden Ausgabe 5/2022 stellen wir ein Formular zur Eingabe neuer Bestellungen inklusive Bestellpositionen vor. Damit der Benutzer komfortabel auf bereits angelegte Bestellungen zugreifen und neue Bestellungen anlegen kann, stellen wir ihm ein Übersichtsformular für die Bestellungen zur Seite. Wie Sie dieses erstellen, zeigen wir im vorliegenden Beitrag. Dabei wollen wir nicht nur die Bestellungen in der Übersicht anzeigen, sondern auch Möglichkeiten zum Durchsuchen der Rechnungen sowie für die Anzeige der zuletzt verwendeten Rechnungen anbieten.

Listenfeld oder Unterformular in der Datenblattansicht?

Wenn man eine Übersicht wie die hier geplante darstellen möchte, stellt sich immer die Frage, ob man ein Listenfeld oder ein Datenblatt zur Auflistung der Einträge verwenden soll. Wenn der Benutzer in der Lage sein soll, auf einfache Weise selbst die Datensätze zu sortieren oder nach verschiedenen Kriterien zu filtern, bietet sich ein Unterformular in der Datenblattansicht an.

Im Gegensatz zum Listenfeld kann der Benutzer hier jedoch standardmäßig die Daten bearbeiten, was Sie gegebenenfalls unterbinden müssen. Wie das gelingt, zeigen wir ebenfalls in diesem Beitrag. Die Datenblattansicht bietet noch weitere Vorteile: Der Benutzer kann die Anordnung der Spalten variieren und auch Spalten ein- und ausblenden.

Unterformular erstellen

Im Gegensatz zu üblichen Gepflogenheiten starten wir in diesem Beitrag einmal nicht mit dem Hauptformular, sondern legen direkt das Unterformular zur Anzeige der Bestellungen an. Dieses soll **sfmBestellungenUebersicht** heißen und die Tabelle **tblBestellungen** als Datensatzquelle verwenden. Wir ziehen alle Felder aus dieser Tabelle mit Ausnahme des Feldes **ID** in den Detailbereich der Entwurfsansicht. Außerdem stellen wir die Eigenschaft **Standardansicht** auf **Datenblatt** ein und schließen das Formular (siehe Bild 1).

Hauptformular erstellen

Das Hauptformular für die Bestellübersicht soll **frmBestellungenUebersicht** heißen. Wir fügen ihm als Erstes das Unterformular **sfmBestellungenUebersicht** hinzu, indem wir dieses aus dem Navigationsbereich in den Detailbereich des Formularentwurfs ziehen.

Damit erhalten wir die Anzeige der Bestellungen in der Datenblattansicht und können gleichzeitig im Hauptformular noch weitere Steuerelemente hinzufügen, was in

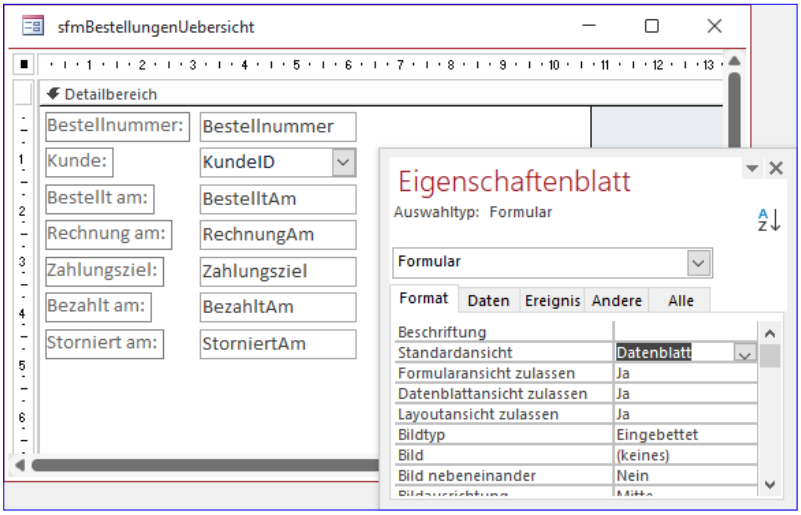


Bild 1: Das Unterformular **sfmBestellungenUebersicht**

einem Formular in der Datenblattansicht allein nicht möglich gewesen wäre.

Für das Unterformular-Steuerelement stellen wir die Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** jeweils auf den Wert **Beide** ein. So wird das Unterformular gemeinsam mit dem Hauptformular vergrößert. Außerdem löschen wir das Bezeichnungsfeld des Unterformulars.

Und wir wissen bereits, dass wir im Hauptformular selbst keine Daten anzeigen wollen. Daher können wir die Eigenschaften **Datensatzmarkierer**, **Navigationsschaltflächen**, **Bildlaufleisten** und **Trennlinien** hier direkt auf den Wert **Nein** einstellen.

Außerdem soll das Formular, wenn es nicht ohnehin maximiert erscheint, zumindest mittig im Access-Fenster landen. Daher stellen wir **Automatisch zentrieren** auf **Ja** ein.

Verwalten von Bestellungen

Ausgehend von hier können wir uns nun überlegen, welche Funktionen wir dem Formular hinzufügen wollen. Hier sind die Ideen, die wir in den folgenden Abschnitten umsetzen:

- Schaltfläche zum Hinzufügen einer neuen Bestellung über das Formular **frmBestellungDetails** mit anschließender Anzeige dieser Bestellung im Unterformular **sfmBestellungenUebersicht**
- Anzeigen der Details zu einer Bestellung im Formular **frmBestellungDetails**, entweder durch Auswahl und anschließendes Betätigen einer Schaltfläche oder per Doppelklick
- Löschen der aktuell markierten Bestellung
- Verhindern von Änderungen der Daten im Unterformular
- Verschiedene Suchkriterien, zum Beispiel Schnellsuche nach Bestellnummer oder Kunde und nach den verschiedenen Datumsangaben

Hinzufügen einer neuen Bestellung

Das Hinzufügen einer neuen Bestellung erfolgt über eine Schaltfläche namens **cmdNeueBestellung**. Diese soll das Formular **frmBestellungDetails** zum Eingeben eines neuen Datensatzes öffnen. Nach dem Hinzufügen soll der neue Datensatz direkt mit den übrigen Bestellungen im Unterformular angezeigt und auch markiert werden. Der notwendige Code lautet wie in Listing 1.

Wir öffnen das Formular **frmBestellungDetails** als modalen Dialog und im Modus zum Hinzufügen eines Datensatzes (siehe Bild 2). In diesem Formular gibt der Benutzer nun einen neuen Datensatz ein. Mit einem Klick auf die **OK**-Schaltfläche schließt er die Eingabe ab und macht das Formular unsichtbar, was den aufrufenden Code fortsetzt.

Hier prüfen wir mit der Hilfsfunktion **IstFormularGeoeffnet**, ob das Formular **frmBestellungDetails** noch geöffnet ist. Ist das der Fall, liest die Prozedur den Wert des dortigen Feldes **ID** in die Variable **lngNeueBestellungID** ein und schließt das Formular endgültig. Danach aktualisiert die Prozedur den Inhalt des Unterformulars **sfmBestellun-**

```
Private Sub cmdNeueBestellung_Click()
    Dim lngNeueBestellungID As Long
    DoCmd.OpenForm "frmBestellungDetails", WindowMode:=acDialog, DataMode:=acFormAdd
    If IstFormularGeoeffnet("frmBestellungDetails") Then
        lngNeueBestellungID = Forms!frmBestellungDetails!ID
        DoCmd.Close acForm, "frmBestellungDetails"
        Me!sfmBestellungenUebersicht.Form.Requery
        Me!sfmBestellungenUebersicht.Form.Recordset.FindFirst "ID = " & lngNeueBestellungID
    End If
End Sub
```

Listing 1: Code zum Öffnen einer neuen Bestellung

genUebersicht und stellt dieses auf den neu angelegten Datensatz ein.

Hier kommt es noch zu einem Problem, wenn die durch den Benutzer eingegebenen Daten nicht erfolgreich validiert werden können. Bisher blendet die Schaltfläche **cmdOK** im Formular **frmBestellungDetails** das Formular nur aus, aber die Validierung erfolgt ja nur, wenn der Datensatz gespeichert wird – und das geschieht erst, wenn die aufrufende Prozedur das Formular mit **DoCmd.Close acForm, "frmBestellungDetails"** schließt. Damit die Validierung auch beim Ausblenden mit der Schaltfläche **cmdOK** durchgeführt wird, rufen wir die Prozedur **Form_BeforeUpdate** explizit auf und werten den darin gesetzten Rückgabeparameter **Cancel** wie folgt aus:

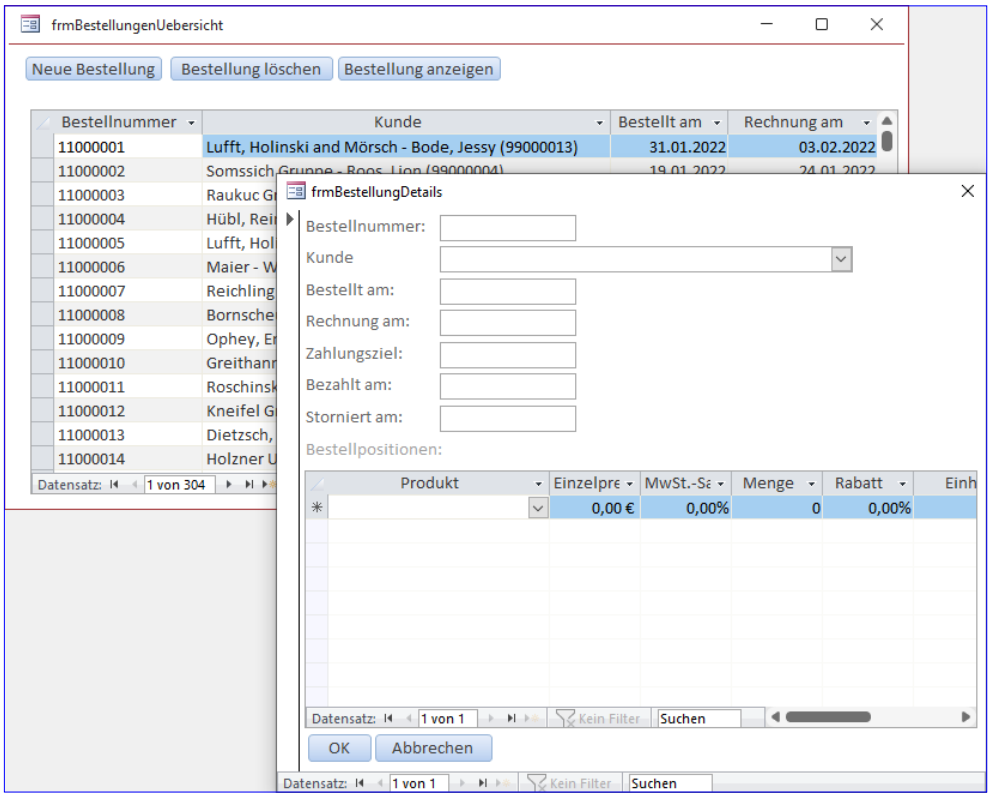


Bild 2: Anlegen einer neuen Bestellung vom Formular **frmBestellungenUebersicht** aus

```
Private Sub cmdOK_Click()
    Dim intCancel As Integer
    If Me.Dirty = False Then
        DoCmd.Close acForm, Me.Name
        Exit Sub
    End If
    Form_BeforeUpdate intCancel
    If intCancel = 0 Then
        Me.Visible = False
    End If
End Sub
```

Im ersten Teil prüfen wir außerdem noch, ob der Benutzer den angezeigten Datensatz überhaupt geändert hat und ob somit überhaupt eine Validierung nötig ist. Ist das nicht der Fall und die Eigenschaft **Me.Dirty** liefert den Wert **False** zurück, schließt **cmdOK_Click** das Formular direkt und beendet die Prozedur.

Anzeigen der Details einer Bestellung

Wenn der Benutzer einen der Datensätze im Unterformular markiert hat und auf die Schaltfläche **cmdBestellungAnzeigen** klickt, soll das Formular **frmBestellungDetails** geöffnet werden und den aktuell markierten Datensatz anzeigen.

Das realisieren wir mit der Prozedur **cmdBestellungAnzeigen_Click** aus Listing 2. Hier rufen wir das Formular **frmBestellungDetails** wieder mit der **DoCmd.OpenForm**-Methode auf, allerdings übergeben wir für den Parameter **DataMode** diesmal den Wert **acFormEdit**. Außerdem legen wir mit **WhereCondition** fest, welchen Datensatz das Formular anzeigen soll.

```
Private Sub cmdBestellungAnzeigen_Click()
    Dim lngAktuelleBestellungID As Long
    lngAktuelleBestellungID = Me!sfmBestellungenUebersicht.Form!ID
    DoCmd.OpenForm "frmBestellungDetails", WindowMode:=acDialog, DataMode:=acFormEdit, WhereCondition:="ID = " _
        & lngAktuelleBestellungID
    If IstFormularGeoeffnet("frmBestellungDetails") Then
        DoCmd.Close acForm, "frmBestellungDetails"
        Me!sfmBestellungenUebersicht.Form.Refresh
    End If
End Sub
```

Listing 2: Code zum Anzeigen einer vorhandenen Bestellung

Die Vorgehensweise, wenn der Benutzer das Formular mit der **OK**-Schaltfläche unsichtbar macht, sieht etwas anders aus. Wir prüfen nur noch, ob es bereits geschlossen ist und holen dies gegebenenfalls nach. Außerdem erneuern wir in diesem Fall die angezeigten Daten im Unterformular-Steuerelement mit der **Refresh**-Methode.

Refresh reicht in diesem Fall aus, da kein neuer Datensatz hinzugefügt wurde, sondern maximal eine Änderung an einem Datensatz durchgeführt wurde, die auch direkt in der Übersicht angezeigt werden soll.

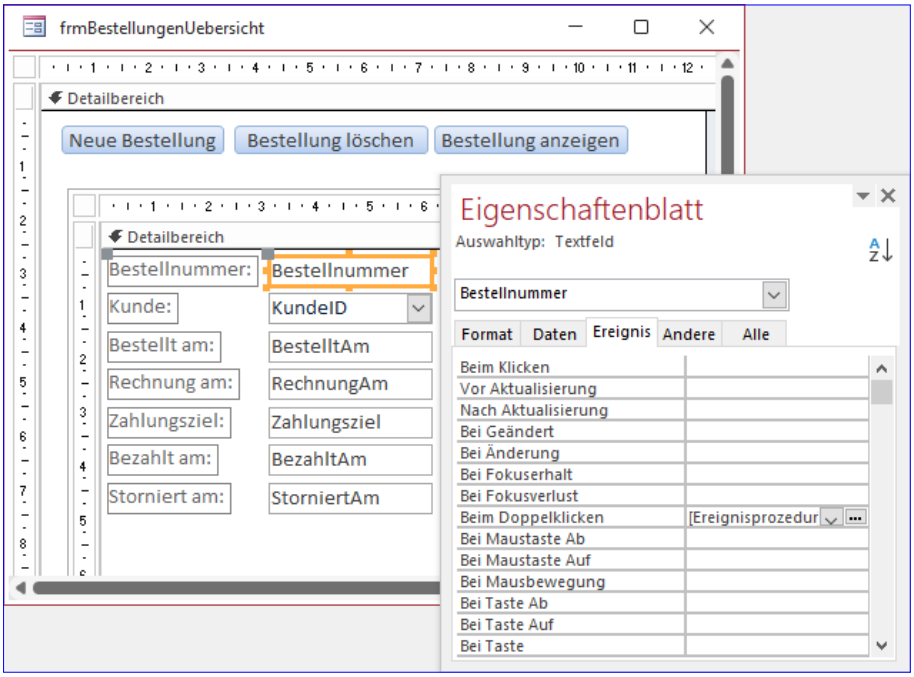


Bild 3: Anlegen einer Ereignisprozedur für den Doppelklick auf die Bestellnummer

Anzeigen der Details einer Bestellung per Doppelklick

Praktisch wäre es auch, wenn der Benutzer eine Bestellung direkt per Doppelklick anzeigen könnte. Wir wollen nur die Spalte mit der Bestellnummer für diese Funktion vorsehen.

Deshalb markieren wir in der Entwurfsansicht des Formulars das Feld **Bestellnummer** und legen für dessen Ereignis **Beim Doppelklicken** eine Ereignisprozedur an (siehe Bild 3).

Die dadurch ausgelöste Ereignisprozedur finden Sie in Listing 3. Sie arbeitet wie die zuvor beschriebene Ereignisprozedur, aber da sie für das Unterformular definiert wurde und nicht aus der Perspektive des Hauptformulars, kann sie leichter auf die ID des doppelt angeklickten Datensatzes zugreifen – und auch der Aufruf der Refresh-Methode erfolgt wesentlich einfacher.

Anzeigen der Details eines Kunden

Natürlich könnte man diese Funktion auch in einem Formular mit einer Kundenübersicht unterbringen, aber

```
Private Sub Bestellnummer_DblClick(Cancel As Integer)
    Dim lngAktuelleBestellungID As Long
    lngAktuelleBestellungID = Me!ID
    DoCmd.OpenForm "frmBestellungDetails", WindowMode:=acDialog, DataMode:=acFormEdit, WhereCondition:="ID = " _
        & lngAktuelleBestellungID
    If IstFormularGeoeffnet("frmBestellungDetails") Then
        DoCmd.Close acForm, "frmBestellungDetails"
        Me.Refresh
    End If
End Sub
```

Listing 3: Code zum Anzeigen einer vorhandenen Bestellung per Doppelklick auf die Bestellnummer im Unterformular

warum nicht auch direkt in der Übersicht der Bestellungen? Vielleicht möchten Sie nicht nur eine Bestellung betrachten, sondern gleich den Kundendatensatz mit allen Bestellungen, die der Kunde bis dato aufgegeben hat.

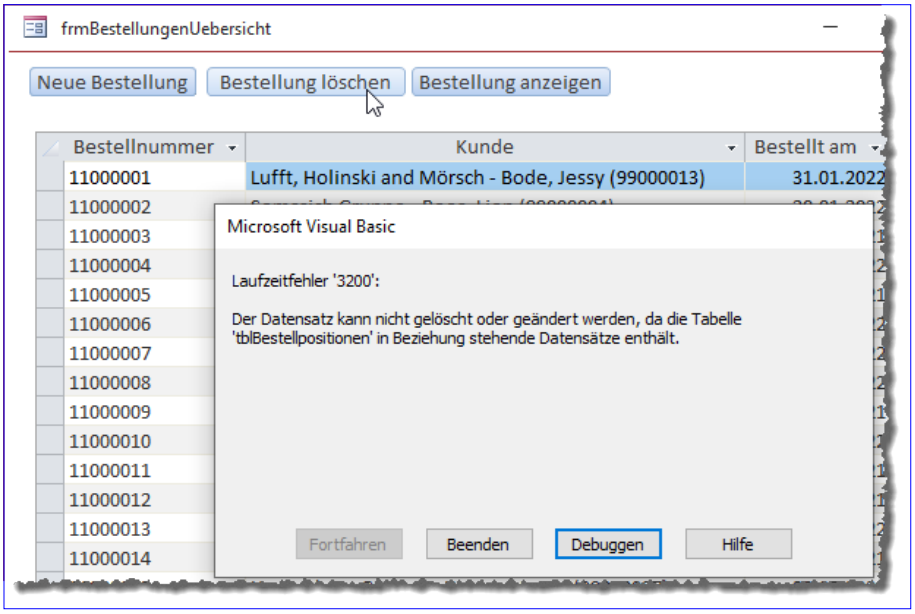


Bild 4: Fehler beim Versuch, eine Bestellung zu löschen, die bereits Bestellpositionen enthält

Also fügen wir dem Formular noch zwei Möglichkeiten hinzu, den Kundendatensatz zur aktuell markierten Bestellung zu öffnen. Die erste ist eine Schaltfläche, die wir diesmal **cmdKundeAnzeigen** nennen. Diese Schaltfläche soll ein Formular namens **frmKunde-Details** öffnen, das wir in einem weiteren Beitrag mit dem Titel **Rechnungsverwaltung: Kundenübersicht** (**www.access-im-unternehmen.de/1387**) in der nächsten Ausgabe beschreiben.

Die Schaltfläche **cmdKundeAnzeigen** löst die Prozedur aus Listing 4 aus. Die Prozedur liest die **ID** des Kunden aus dem Fremdschlüsselfeld **KundeID** der markierten Bestellung ein und verwendet diese als Vergleichswert des Parameters **WhereCondition** beim Öffnen des Formulars **frmKundeDetails**. Dort kann der Benutzer die gewünschten Änderungen durchführen und die Bearbeitung mit der Schaltfläche **OK** abschließen. Eventuelle Änderungen werden dann mit der **Refresh**-Methode in das Feld **cboKundeID** des Unterformulars **sfmBestellungenUebersicht** übernommen.

Die gleiche Funktion möchten wir für einen Doppelklick auf das Steuerelement **cboKundeID** im Unterformular **sfmBestellungenUebersicht** abbilden. Dazu hinterlegen wir die Prozedur aus Listing 5 für das Ereignis **Beim Doppelklicken** dieses Steuerelements.

Löschen einer Bestellung

Die Schaltfläche **cmdBestellungLoeschen** soll das Löschen einer Bestellung ermöglichen. Hier wird es spannend: Wir haben im Datenmodell angegeben, dass nur

```
Private Sub cmdKundeAnzeigen_Click()
    Dim lngAktuellerKundeID As Long
    lngAktuellerKundeID = Me!sfmBestellungenUebersicht.Form!cboKundeID
    DoCmd.OpenForm "frmKundeDetails", WindowMode:=acDialog, DataMode:=acFormEdit, WhereCondition:="ID = " _
        & lngAktuellerKundeID
    If IstFormularGeoeffnet("frmKundeDetails") Then
        DoCmd.Close acForm, "frmKundeDetails"
        Me!sfmBestellungenUebersicht.Form.Refresh
    End If
End Sub
```

Listing 4: Öffnen des Kunden der aktuell ausgewählten Bestellung per Klick auf **cmdKundeAnzeigen**

```
Private Sub cboKundeID_DblClick(Cancel As Integer)
    Dim lngAktuellerKundeID As Long
    lngAktuellerKundeID = Me!cboKundeID
    DoCmd.OpenForm "frmKundeDetails", WindowMode:=acDialog, DataMode:=acFormEdit, WhereCondition:="ID = " _
        & lngAktuellerKundeID
    If IstFormularGeoeffnet("frmKundeDetails") Then
        DoCmd.Close acForm, "frmKundeDetails"
        Me.Refresh
    End If
End Sub
```

Listing 5: Öffnen des Kunden der aktuell ausgewählten Bestellung per Doppelklick auf den Kundennamen im Unterformular

Bestellungen gelöscht werden können sollen, für die noch keine Datensätze in der verknüpften Tabelle **tblBestellpositionen** angelegt wurden.

Ein einfaches Löschen mit einer **DELETE**-Aktionsabfrage würde daher zu der Fehlermeldung aus Bild 4 führen.

Daher führen wir die Prozedur, die durch die Schaltfläche **cmdBestellungLoeschen** ausgeführt wird, wie in Listing 6 aus. Diese ermittelt zunächst die **ID** der zu löschenden Bestellung und verwendet diese als Vergleichswert der nachfolgend ausgeführten **DELETE**-Aktionsabfrage. Vorher deaktivieren wir jedoch die eingebaute Fehlerbehandlung, um auf den oben erwähnten Fehler mit einer eigenen Meldung reagieren zu können.

Diese fordert den Benutzer auf, die Bestellpositionen zu sichten und gegebenenfalls manuell zu entfernen, bevor er einen neuen Anlauf zum Löschen des Bestelldatensatzes startet. Dazu öffnen wir direkt das Formular **frmBestellungDetails** und zeigen den zu löschenden Datensatz an. Nachdem der Benutzer dieses Formular mit **OK** ausgeblendet hat, schließt die Prozedur das Formular und aktualisiert die Bestellübersicht. Dann markiert sie den zu löschenden Datensatz erneut.

Falls beim Löschen des Bestelldatensatzes kein Fehler aufgetreten ist, aktualisiert die Prozedur einfach noch die Bestellübersicht im Unterformular.

Änderungen im Unterformular verhindern

Wir wollen außerdem sicherstellen, dass der Benutzer direkt im Unterformular mit den Bestellungen keine Datensätze ändern kann. Das soll dem Formular **frmBestellungDetails** vorbehalten sein. Dazu ist nicht viel zu tun: Wir müssen einfach nur in der Entwurfsansicht das Unterformular **sfmBestellungenUebersicht** markieren und für dieses die Eigenschaften **Anfügen zulassen**, **Löschen**

E-Mail-Adressen validieren per VBA

Immer mehr Vorgänge werden per E-Mail verarbeitet. Dazu gehören auch Bestellungen, Rechnungen et cetera. Früher wurden beispielsweise Rechnungen an die Postadresse geschickt, was einigermaßen fehlertolerant war. Spätestens der Briefträger hat die falsche Hausnummer erkannt und die Sendung dank regelmäßigem Zustellungsgebiet beim richtigen Adressaten abgeliefert. Bei E-Mails verhält sich dies völlig anders: Hier führt jede Ungenauigkeit zur Unzustellbarkeit, toleriert werden allenfalls noch Abweichungen bei der Groß-/Kleinschreibung. Daher lohnt es sich, die E-Mail-Adressen von Kunden und anderen Adressaten zumindest oberflächlich zu prüfen.

Aufbau einer E-Mail-Adresse

Als Erstes unterteilen wir eine E-Mail-Adresse grob in den Teil vor dem @-Zeichen und dahinter. Davor finden wir den sogenannten Lokalteil, dahinter die Domain des E-Mail-Providers.

Schauen wir uns zuerst die Domain an. Diese besteht wiederum aus drei Teilen: dem Hostnamen, einem Punkt und der Top-Level-Domain. Da es immer mehr Top-Level-Domains gibt, wird es schwierig, diese konkret auf Gültigkeit zu prüfen. Auch die Anzahl der Zeichen wurde im Laufe der Zeit immer mehr ausgeweitet. Früher gab es nur Top-Level-Domains mit zwei oder drei Buchstaben (wie **de** oder **com**), mittlerweile finden wir auch solche mit mehr Buchstaben.

Der Teil vor dem @-Zeichen heißt Lokalteil und kann je Domain nur einmal vergeben werden. Er kann die folgenden Zeichen verwenden:

```
A-Za-z0-9.!#$%&'*+-/=?^_`{|}~
```

Allerdings finden wir in der Regel nur Buchstaben, Zahlen und den Punkt vor. Dennoch können wir die möglichen Zeichen in die Prüfung mit einbeziehen.

Möglichkeiten für die Prüfung

Wir können verschiedene Techniken für die Prüfung von E-Mail-Adressen einsetzen. Wenn wir nur eine grundlegende Prüfung vornehmen wollen, kommen wir mit den Zeichenkettenfunktionen von VBA aus. Für eine weitergehende Prüfung ist auch die Nutzung von regulären Ausdrücken möglich. Dazu benötigen wir jedoch bereits eine zusätzliche Bibliothek. Diese beiden Methoden erlauben jedoch nur die Prüfung der Syntax der E-Mail-Adresse.

Wenn Sie sicherstellen wollen, dass die E-Mail auch beim Empfänger ankommt, können Sie einen der zahlreichen Onlinedienste beanspruchen. Diese sind jedoch meist kostenpflichtig oder bieten keinen Webservice an, mit dem wir diese nutzen können.

Grundlegende Prüfung

Die grundlegendste Prüfung kontrolliert schlicht, ob die E-Mail-Adresse bestimmte Elemente und keine unerlaubten Sonderzeichen enthält.

Diese wollen wir nun per VBA programmieren. Die dazu notwendige Funktion soll **CheckEMailSyntax** heißen und die zu prüfende E-Mail-Adresse als Parameter entgegennehmen. Das Ergebnis soll ein **Boolean**-Wert sein, der im Falle einer gültigen E-Mail-Adresse den Wert **True** enthält. Diese definieren wir zunächst wie folgt:

```
Public Function CheckEMailSyntax(strEMail As String) As
Boolean

End Function
```

Nun wollen wir ein paar Testfälle mit vorgegebenen Ergebnissen definieren, mit denen wir die Funktion direkt während der Entwicklung testen können. Dazu verwenden wir zwei verschiedene Prozeduren.

```
Public Sub Test_CheckEMailSyntax(strEMail As String, bolResult As Boolean)
    If CheckEMailSyntax(strEMail) = bolResult Then
        Debug.Print "Test mit '" & strEMail & "' erfolgreich"
    Else
        Debug.Print "Test mit '" & strEMail & "' fehlgeschlagen"
    End If
End Sub
```

Listing 1: Eigentliche Testprozedur

Die erste ruft die eigentliche Testprozedur auf und übergibt dieser jeweils die zu untersuchende E-Mail-Adresse und die Angabe, ob diese gültig ist oder nicht. Hier sind die Aufrufe mit einigen Test-Adressen:

```
Public Sub Test()
    Test_CheckEMailSyntax "andre@minhorst.com", True
    Test_CheckEMailSyntax "andre@minhorst,com", False
    Test_CheckEMailSyntax "andreQminhorst.com", False
    Test_CheckEMailSyntax "andre@minhorst.de", True
    Test_CheckEMailSyntax "@minhorst.de", False
    Test_CheckEMailSyntax "andre@.de", False
    Test_CheckEMailSyntax "andre@minhorst.", False
    Test_CheckEMailSyntax "andre@minhorst@de", False
End Sub
```

Die zweite Prozedur führt den Testaufruf aus und gleicht das Ergebnis mit dem erwarteten Ergebnis ab. Liefert der Test das erwartete Ergebnis, erscheint eine Zeile im Direktbereich, die auf den erfolgreichen Test verweist.

Anderenfalls wird der fehlgeschlagene Test gemeldet (siehe Listing 1).

Mit den obigen Tests sieht das Ergebnis wie folgt aus:

```
Test mit 'andre@minhorst.com' erfolgreich
Test mit 'andre@minhorst,com' erfolgreich
Test mit 'andreQminhorst.com' erfolgreich
Test mit 'andre@minhorst.de' erfolgreich
Test mit '@minhorst.de' fehlgeschlagen
Test mit 'andre@.de' fehlgeschlagen
Test mit 'andre@minhorst.' fehlgeschlagen
Test mit 'andre@minhorst@de' erfolgreich
```

Zufälligerweise liefert die Funktion mit dem Wert **False** in einigen Fällen das richtige Ergebnis, was sich aber gleich relativieren wird.

Als Erstes wollen wir prüfen, ob ein **@**-Zeichen in der E-Mail enthalten ist. Dazu erweitern wir die Funktion wie folgt:

```
Public Function CheckEMailSyntax(strEMail As String) _
                                                As Boolean
    If InStr(1, strEMail, "@") = 0 Then
        CheckEMailSyntax = False
        Exit Function
    End If
    '...
    'weitere Prüfungen
    '...
    CheckEMailSyntax = True
End Function
```

Hier prüfen wir mit der **InStr**-Funktion, ob in **strEMail** überhaupt ein **@**-Zeichen enthalten ist. Rufen wir die **Test**-Prozedur erneut auf, liefern nun andere Tests ein positives Ergebnis. Wenn die Bedingung falsch ist, also die Funktion nicht über **Exit Function** verlassen wird, gibt die Funktion den Wert **True** zurück.

Aber die Prüfung ist nicht ganz korrekt, denn es soll nicht nur überhaupt ein **@**-Zeichen enthalten sein, sondern genau eines. Wie können wir prüfen, ob genau ein **@**-Zei-

chen vorhanden ist? Wir ersetzen dieses durch eine leere Zeichenkette und vergleichen dann die Länge mit der Länge des Originals. Der durch die folgende Bedingung ermittelte Unterschied muss 1 sein:

```
If Not Len(strEMail) - Len(Replace(strEMail, "@", "")) _
                                                = 1 Then
    CheckEMailSyntax = False
    Exit Function
End If
```

Als Nächstes fügen wir eine Bedingung hinzu, die auf das Vorhandensein des Punkts prüft. Im Gegensatz zum @-Zeichen kann die E-Mail-Adresse auch mehrere Punkte enthalten, wie zum Beispiel in **andre.minhorst@t-online.de**. Allerdings darf nur ein Punkt hinter dem **@**-Zeichen auftauchen. Wir schauen uns also nur den Teil hinter dem **@**-Zeichen an.

Dazu deklarieren wir eine Variable namens **strDomain** und fügen den Teil hinter dem **@**-Zeichen in diese Variable ein:

```
Dim strDomain As String
...
strDomain = Mid(strEMail, InStr(1, strEMail, "@") + 1)
```

Danach führen wir für **strDomain** und den Punkt die gleiche Prüfung wie oben für **strEMail** und das **@**-Zeichen durch und prüfen so, ob genau ein Punkt in **strDomain** enthalten ist:

```
If Not Len(strDomain) - Len(Replace(strDomain, ".", "")) _
                                                = 1 Then
    CheckEMailSyntax = False
    Exit Function
End If
```

Damit gelingen schon einmal die ersten vier Tests, also schauen wir uns an, was wir noch nicht berücksichtigt haben, dass der fünfte Test fehlschlägt. Hier testen wir auf **@minhorst.com**, was das Ergebnis **False** liefern sollte, aber die Prozedur liefert das Ergebnis **True**.

Wir müssen also vor dem **@**-Zeichen noch mindestens ein weiteres Zeichen vorfinden. Dazu lesen wir den Teil vor dem **@**-Zeichen in die Variable **strLocal** ein:

```
Dim strLocal As String
...
strLocal = Left(strEMail, InStr(1, strEMail, "@") - 1)
```

Danach können wir zunächst einmal prüfen, ob **strLocal** mindestens die Länge **1** aufweist:

```
If Len(strLocal) = 0 Then
    CheckEMailSyntax = False
    Exit Function
End If
```

Damit kommen wir zum nächsten Test, der fehlschlägt: **andre@.de** liefert einen Fehler. Logisch: Hier fehlt der Hostname, also der Teil vor dem Punkt. Also nutzen wir zwei weitere Variablen, in denen wir den Hostnamen und die Top Level Domain speichern:

```
Dim strHostname As String
Dim strTLD As String
```

Diese ermitteln wir aus **strDomain** – für den Hostnamen bis zum ersten Punkt von hinten, für die Top Level Domain vom ersten Punkt von hinten an:

```
strHostname = Left(strDomain, InStrRev(strDomain, ".") - 1)
strTLD = Mid(strDomain, InStrRev(strDomain, ".") + 1)
```

Die Prüfung, ob ein Hostname enthalten ist, sieht so aus:

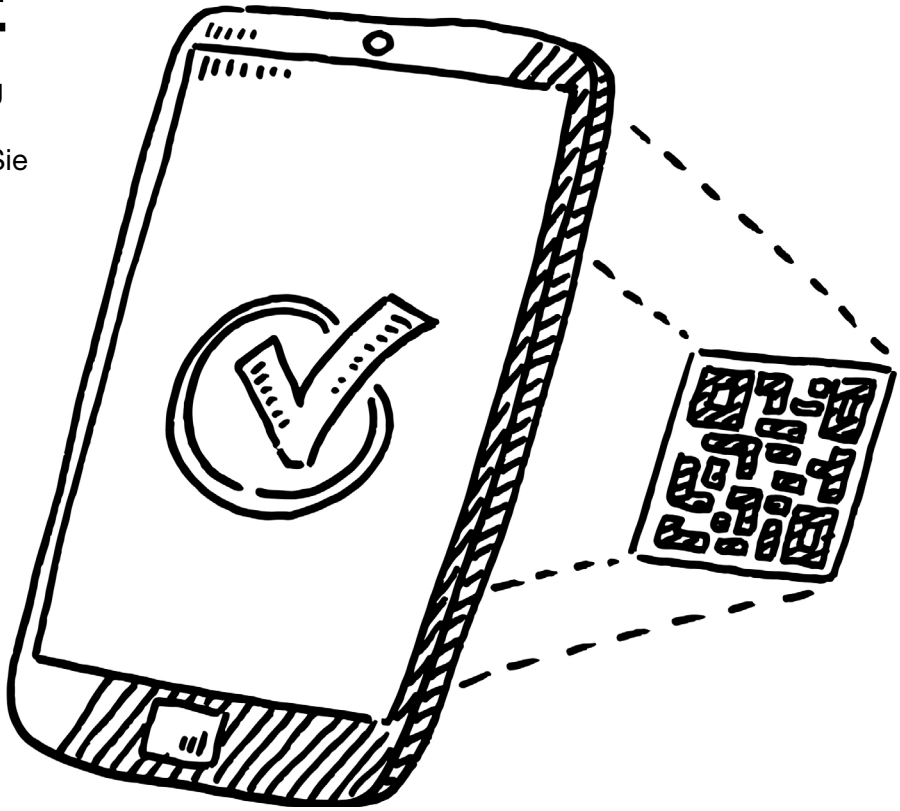
```
If Len(strHostname) = 0 Then
    CheckEMailSyntax = False
    Exit Function
End If
```

HAUFE.
Ausgabe 05/2022

ACCESS
IM UNTERNEHMEN

RECHNUNGEN BEZAHLEN PER QR-CODE

Statten Sie Ihre Rechnungserstellung mit Funktionen zum Hinzufügen von QR-Bezahlcodes aus und ersparen Sie Ihren Kunden viel Zeit (ab S. 65).



In diesem Heft:

FORMULARPOSITION MERKEN

Sorgen Sie dafür, dass sich Access die Positionen von Objekten beim Schließen merkt und beim Öffnen wiederherstellt.

SEITE 2

SETUP MIT DER ACCESS-RUNTIME

Statten Sie Benutzer ohne Access-Vollversion mit der kostenlosen Runtime aus – ganz einfach per Anwendungssetup.

SEITE 43

RECHNUNGEN VERWALTEN

Unsere Musterlösung zur Rechnungsverwaltung wächst weiter – diesmal um ein Bestellformular und ein Kundendetailformular.

SEITE 14

www.access-im-unternehmen.de
Mat-Nr. 01583-5133

ARCHIV SEITE 309

Rechnungen einfacher bezahlen

In letzter Zeit wird die Vereinfachung der Bezahlung und auch Verarbeitung von Rechnungen forciert. Erst war es die XRechnung, über die wir bereits in Ausgabe 6/2020 berichtet haben. Die XRechnung ist eine Version einer Rechnung im XML-Format, die leicht beispielsweise in eine Datenbank eingelesen werden kann. Nun folgt mit dem EPC-QR-Code eine weitere Vereinfachung auch für Privatpersonen. Dieser Code wird auf üblichen Rechnungen aufgebracht, damit der Empfänger die Überweisungsdaten mit seiner Banking-App einlesen und die Überweisung schnell ausführen kann.

Beides greifen wir in der aktuellen Ausgabe auf. Während wir in Ausgabe 6/2020 berichtet haben, wie Sie XRechnungen aus den Daten Ihrer Datenbank erstellen können, beschreiben wir im Beitrag **XRechnung, Teil 2: Rechnungen einlesen** ab Seite 52, wie Sie die Daten einer XRechnung in die Tabellen Ihrer Datenbank einlesen können.

Die zweite Lösung dieser Ausgabe dreht sich um das Vereinfachen des Bezahlvorgangs nach dem Erhalt einer Rechnung. Viele Onlinebanking-Apps bieten bereits Schaltfläche mit Texten wie **Fotoüberweisung** oder **QR-Code scannen**. Nach dem Anklicken bieten diese die Möglichkeit, einen auf der Rechnung abgebildeten QR-Code zu scannen, der alle für die Überweisung wichtigen Daten wie Verwendungszweck, Empfängerdaten und den zu überweisenden Betrag enthält. Und damit beginnt unser Teil der Aufgabe: Das Bereitstellen eines solchen QR-Codes auf den mit Access erstellten Rechnungsberichten. Die notwendigen Schritte zum Erstellen eines solchen QR-Codes beschreiben wir ab Seite 65 im Beitrag **EPC-QR-Code per COM-DLL erstellen**.

Wer viele Formulare und andere Elemente in seiner Access-Datenbank so nebeneinander anordnet, dass er optimal damit arbeiten kann, ärgert sich vielleicht, wenn er nach dem Schließen und erneuten Öffnen seine Wunschkonfiguration immer wieder neu herstellen muss. Wenn das bei Ihnen so ist, haben wir perfekte Lösung! Im Beitrag **Objektpositionen speichern und wiederherstellen** beschreiben wir ab Seite 2, wie Sie Ihrer Anwendung eine Funktion hinzufügen, die Ihr Problem dauerhaft löst. Diese Lösung enthält eine Tabelle sowie zwei Prozeduren. Die eine wird beim Schließen der Datenbank ausgelöst und schreibt die aktuellen Positionen der geöffneten Objekte in die Tabelle. Die andere wird beim Öffnen der Anwendung gestartet und öffnet die in der Tabelle gespeicherten Objekte wieder – samt Wiederherstellung von Position und Größe.

In unserer Beitragsreihe zur Rechnungsverwaltung, die später darin gipfeln wird, dass wir Rechnungen mit dem oben erwähnten QR-Code erstellen werden, finden Sie zwei neue Teile. Im ersten Teil namens **Rechnungsverwaltung: Bestellformular** erläutern wir ab Seite 14, wie wir per Formular neue Bestellungen und die entsprechenden Bestellpositionen erfassen können.

Der zweite neue Teil mit dem Titel **Rechnungsverwaltung: Kundendetails** beschreibt ab Seite 31, wie wir die Kundendetails inklusive der bereits für diesen Kunden erfassten Bestellungen verwalten können.

Und schließlich zeigt unser Autor Chris Jüngling ab Seite 43 im Beitrag **Access-Applikation mit Runtime installieren**, wie Sie für Benutzer, die kein Access auf dem Rechner installiert haben, noch die kostenlose Runtime-Version gemeinsam mit der zu installierenden Datenbankanwendung in einem Setup verpacken.

Nun viel Spaß beim Lesen!

Ihr André Minhorst

Objektpositionen speichern und wiederherstellen

Neulich fragte ein Leser, ob und wie man die Position von Objekten im Access-Fenster speichern und wiederherstellen könne. Der Hintergrund ist, dass er immer wieder mühsam Tabellen, Abfragen und andere Objekte zu einem Arbeitsbereich zusammengestellt hat und wenn er die Anwendung schließt, ist die ganze Arbeit dahin – und am nächsten Tag muss er die Objekte erneut anordnen. Ich fühlte mich ein wenig an Zeiten erinnert, wo man zwar einen Homecomputer zum Programmieren, aber kein Gerät zum Speichern der eingetippten Spiele aus den Computermagazinen hatte ... Da sich die Zeiten zum Glück geändert haben, zeige ich in diesem Beitrag, wie Sie die Position und Größe der beim Schließen einer Datenbank geöffneten Objekte abspeichern und beim nächsten Öffnen wieder herstellen können.

Aufgabenstellung

Aus den Anforderungen des Lesers ergeben sich die folgenden Aufgabenstellungen:

- Herausfinden, wie wir die Position und Größe aller zu einem bestimmten Zeitpunkt geöffneten Objekte ermitteln können
- Definieren einer Tabelle zum Speichern der geöffneten Objekte mit Name und Objekttyp und ihrer Position und Größe sowie der aktuellen Ansicht
- Prozedur zum Speichern dieser Informationen in einer geeigneten Tabelle in der jeweilige Datenbankanwendung
- Herausfinden, wie wir die abgespeicherten Objekte wieder öffnen und die Position und die Größe zu einem bestimmten Zeitpunkt wiederherstellen können
- Festlegen eines Automatismus, die Position und Größe der Objekte beim Schließen der Datenbank zu speichern.
- Festlegen eines weiteren Automatismus, um die Position und Größe bei Öffnen der Datenbankanwendung wiederherzustellen

Größe und Position der geöffneten Objekte ermitteln

Die erste Aufgabe ist bereits die anspruchsvollste: Wie können wir alle geöffneten Objekte ermitteln und wie finden wir die aktuelle Position und Größe dieser Objekte heraus?

Für diese Aufgabe gibt es keine Lösung, mit der wir alle Objekte gleichermaßen behandeln können. So gibt es zwar Auflistungen namens **Forms** und **Reports**, mit denen direkt auf die aktuell geöffneten Formulare und Berichte zugegriffen werden kann.

Entsprechende Auflistungen für Tabellen und Abfragen beispielsweise namens **Tables** oder **Queries** suchen wir jedoch vergeblich.

Wenn wir jedoch nach diesen Schlüsselwörtern im Objektkatalog des VBA-Editors suchen, finden wir schnell passende Einträge, nämlich **AllTables** und **AllQueries** (siehe Bild 1).

Damit haben wir zumindest schon einmal Auflistungen für alle Objekttypen gefunden – nun schauen wir uns an, wie wir an die jeweils gewünschten Informationen wie Name, aktuelle Ansicht, Position vom linken und oberen Rand, Höhe und Breite gelangen.

Formulare und Berichte analysieren

Bei Formularen und Berichten macht Access es uns ein wenig leichter als bei Tabellen und Abfragen, die wir uns im Anschluss anschauen. Mit der **Forms**- und der **Reports**-Auflistung können wir direkt die aktuell geöffneten Objekte referenzieren. Mit der folgenden Prozedur durchlaufen wir beispielsweise alle Formulare, die aktuell geöffnet sind:

```
Public Sub Formulare()
    Dim frm As Form
    For Each frm In Forms
        Debug.Print frm.Name, _
          frm.WindowLeft, frm.WindowTop, _
          frm.WindowWidth, frm.WindowHeight
    Next frm
End Sub
```

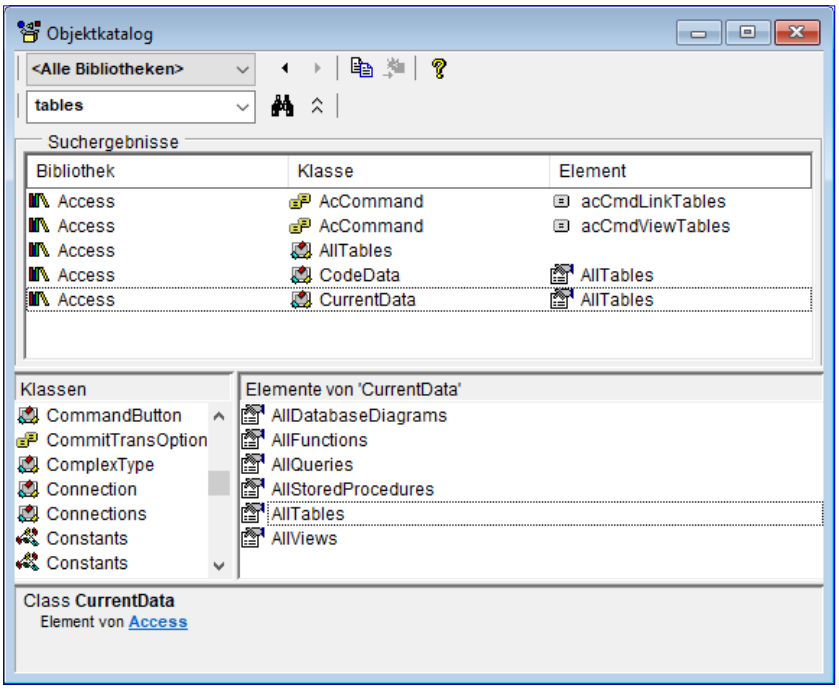


Bild 1: Finden von geeigneten Auflistungen für den Zugriff auf Tabellen und Abfragen

Die Werte der Eigenschaften **WindowLeft**, **WindowTop**, **WindowWidth** und **WindowHeight** liefert Access in der Einheit **Twips**. Vorneweg: Diese Eigenschaften sind schreibgeschützt und wir können diese später nicht nutzen, um die Größe und die Position der Formulare und Berichte wiederherzustellen. Allerdings arbeitet auch die Methode, die wir dazu später nutzen werden, mit Angaben in dieser Einheit. Daher verzichten wir an dieser Stelle auf eine Beschreibung, was Twips genau sind.

Den Namen des jeweiligen Formulars oder Berichts lesen wir einfach aus der Eigenschaft **Name** aus. Spannend ist nun noch, die aktuelle Ansicht zu ermitteln.

Formulare können in den Ansichten **Einzelnes Formular**, **Endlosformular**, **Datenblatt** oder **Geteiltes Formular** angezeigt werden (siehe Bild 2). Bei Berichten kommen noch die Ansichten **Berichtssicht** und **Seitenansicht** hinzu.

Diese Eigenschaft können wir per VBA einerseits über **CurrentView** ermitteln. **CurrentView** liefert die folgenden Werte:

- **0** (**acCurViewDesign**): Entwurfsansicht

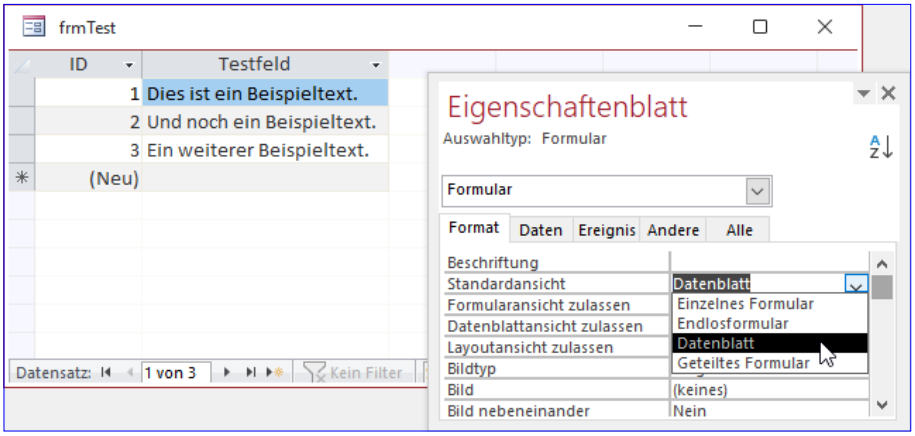


Bild 2: Mögliche Ansichten für ein Formular

- **1** (**acCurViewFormBrowse**): Formularansicht
- **2** (**acCurViewDatasheet**): Datenblattansicht
- **5** (**acCurViewPreview**): Seitenansicht (bei Berichten)
- **6** (**acCurViewReportBrowse**): Berichtssicht (bei Berichten)
- **7** (**acCurViewLayout**): Layoutansicht

Aber Moment – das sind ja gar nicht die Werte, die wir für die Eigenschaft **Standardansicht** aus dem Eigenschaftenblatt auswählen können. Diese können wir der VBA-Eigenschaft **DefaultView** eines **Form**-Objekts entnehmen und die Werte stimmen nicht mit den Zahlenwerten für die Eigenschaft **CurrentView** überein.

Im Gegenteil – es gibt sogar für Formulare und Berichte teilweise gleiche Werte mit unterschiedlicher Bedeutung:

- **0**: Einzelnes Formular
- **1**: Endlosformular
- **2**: Datenblatt
- **3**: PivotTable
- **4**: PivotChart
- **5**: Geteiltes Formular

Für Berichte gibt es die folgenden beiden Werte:

- **0**: Seitenansicht
- **1**: Berichtsansicht

Wie aber kommen wir an die aktuelle Ansicht? Immerhin kann der Benutzer diese ja, wenn die möglichen Ansichten

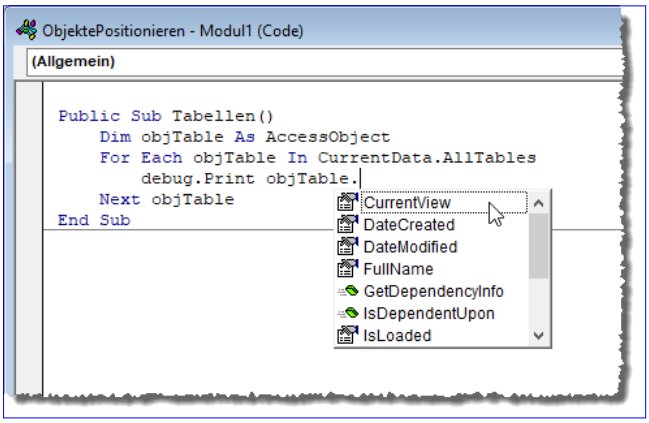


Bild 3: Erkunden der Eigenschaften von **AccessObject**-Elementen

nicht auf eine Ansicht eingeschränkt ist, wechseln. Daher verwenden wir die Eigenschaft **CurrentView**, um die aktuelle Ansicht zu ermitteln.

Tabellen und Abfragen analysieren

Etwas komplizierter wird die Analyse bei Tabellen und Abfragen. Wie bereits erwähnt, gibt es keine Auflistung, die alle derzeit geöffneten Tabellen oder Abfragen auflistet.

Also schauen wir, was wir mit den beiden Auflistungen **AllTables** und **AllQueries**, die wir weiter oben im Objektkatalog gefunden haben, für unsere Zwecke nutzen können.

Um diese Auflistungen zu durchlaufen, wollen wir zunächst herausfinden, welchen Datentyp die damit referenzierten Elemente überhaupt aufweisen. Dazu greifen wir einfach auf das erste Element einer solchen Auflistung zu und lassen uns den Objekttyp mit der Funktion **Typename** im Direktbereich ausgeben:

```
? Typename(CurrentData.AllTables(0))
AccessObject
```

Jetzt, da wir den Objekttypen kennen, können wir die Elemente direkt in einer **For Each**-Schleife durchlaufen und auch per IntelliSense auf die Eigenschaften der Elemente zugreifen (siehe Bild 3).

Um herauszufinden, welche Objekte geöffnet sind und in welcher Ansicht sie dann angezeigt werden, verwenden wir die folgende Prozedur:

```
Public Sub GeoeffneteTabellenDurchlaufen()
    Dim objTable As AccessObject
    For Each objTable In CurrentData.AllTables
        Debug.Print objTable.Name;
        If objTable.IsLoaded Then
            Debug.Print objTable.CurrentView
        Else
            Debug.Print
        End If
    Next objTable
End Sub
```

Damit erhalten wir beispielsweise folgendes Ergebnis, wenn die beiden Tabellen **tblObjecttypes** und **tblTest** in der Datenblattansicht und die Tabelle **tblObjects** in der Entwurfsansicht geöffnet sind:

```
MSysAccessStorage
MSysACEs
... weitere nicht geöffnete Systemtabellen
MSysRelationships
MSysResources
tblObjects 0
tblObjecttypes 2
tblTest 2
```

Interessant sind für Tabellen die folgenden Ansichten:

- **0**: Entwurfsansicht
- **2**: Datenblattansicht

Abfragen durchlaufen wir auf ähnliche Weise – wir verwenden hier lediglich die Auflistung **AllQueries**:

```
Public Sub GeoeffneteAbfragenDurchlaufen()
    Dim objQuery As AccessObject
    For Each objQuery In CurrentData.AllQueries
        Debug.Print objQuery.Name;
        If objQuery.IsLoaded Then
            Debug.Print objQuery.CurrentView
        Else
            Debug.Print
        End If
    Next objQuery
End Sub
```

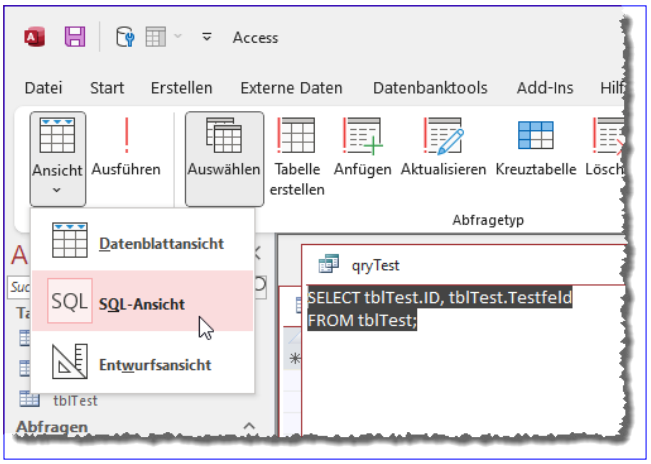


Bild 4: Die SQL-Ansicht einer Abfrage

Interessant ist hier, dass es eigentlich noch eine weitere Ansicht gibt, nämlich die SQL-Ansicht (siehe Bild 4). Wenn wir eine Abfrage in dieser Ansicht öffnen und die Prozedur **GeoeffneteAbfrageDurchlaufen** starten, gibt diese ebenfalls den Wert **0** für die Eigenschaft **CurrentView** zurück.

Davon ausgehend, dass der Benutzer nicht den Zustand von in der SQL-Ansicht geöffneten Abfragen speichern möchte, stellt das aber auch im Rahmen dieses Beitrags kein Problem dar.

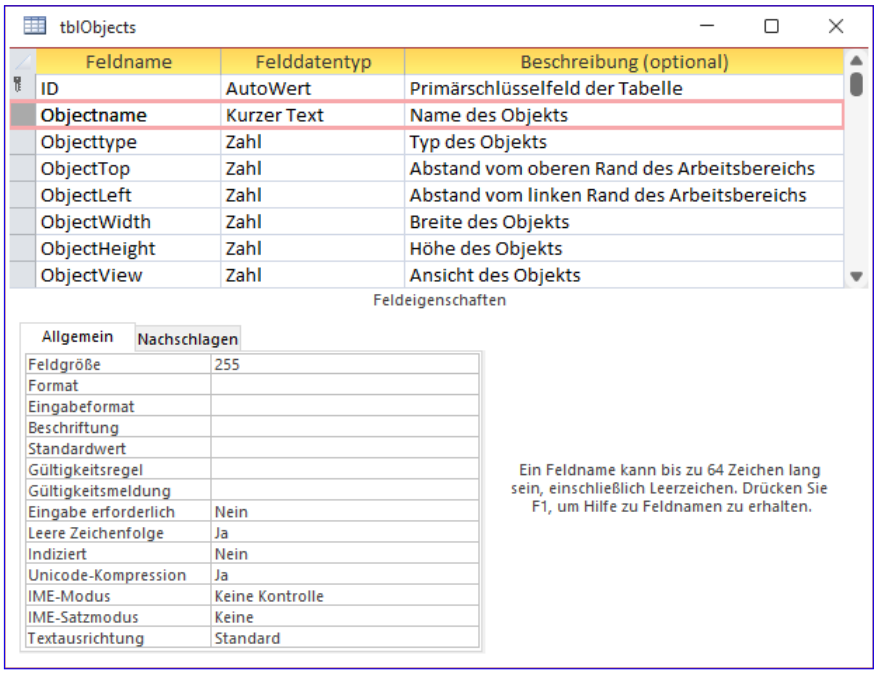
Position und Größe von Tabellen und Abfragen ermitteln

Nachdem wir die Größe und Position von Formularen und Berichten direkt über die Eigenschaften der **Form**- beziehungsweise **Report**-Objekte ermitteln können, schauen wir uns nun die Tabellen und Abfragen an. Da das **AccessObject**-Element keine diesbezüglichen

Eigenschaften offeriert, müssen wir einen kleinen Umweg gehen, um die Informationen zu beschaffen.

Ausgehend davon, dass wir ohnehin nur geöffnete Elemente verwenden wollen, können wir dann das **Screen**-Objekt nutzen, welches uns Informationen über die jeweils aktiven Objekte liefert, also die Objekte, die aktuell den Fokus besitzen. Mit dem **Screen**-Objekt können wir über die folgenden Eigenschaften auf die jeweiligen Objekte zugreifen:

- **Screen.ActiveDatasheet**: Aktive Tabellen und Abfragen in der Datenblattansicht
- **Screen.ActiveForm**: Aktives Formular
- **Screen.ActiveReport**: Aktiver Bericht


tblObjects

Feldname	Felddatentyp	Beschreibung (optional)
ID	AutoWert	Primärschlüsselfeld der Tabelle
Objectname	Kurzer Text	Name des Objekts
Objecttype	Zahl	Typ des Objekts
ObjectTop	Zahl	Abstand vom oberen Rand des Arbeitsbereichs
ObjectLeft	Zahl	Abstand vom linken Rand des Arbeitsbereichs
ObjectWidth	Zahl	Breite des Objekts
ObjectHeight	Zahl	Höhe des Objekts
ObjectView	Zahl	Ansicht des Objekts

Feldeigenschaften

Allgemein | Nachschlagen

Feldgröße	255
Format	
Eingabeformat	
Beschriftung	
Standardwert	
Gültigkeitsregel	
Gültigkeitsmeldung	
Eingabe erforderlich	Nein
Leere Zeichenfolge	Ja
Indiziert	Nein
Unicode-Kompression	Ja
IME-Modus	Keine Kontrolle
IME-Satzmodus	Keine
Textausrichtung	Standard

Ein Feldname kann bis zu 64 Zeichen lang sein, einschließlich Leerzeichen. Drücken Sie F1, um Hilfe zu Feldnamen zu erhalten.

Bild 5: Tabelle zum Speichern der Objekteigenschaften

Hier sehen wir also eine weitere Einschränkung bezüglich der Ansichten, die wir beim Speichern von Position und Größe berücksichtigen können: Die Entwurfsansicht von Tabellen oder Abfragen fällt weg, da wir diese nicht mit den **Active...**-Eigenschaften referenzieren können.

Die Eigenschaft **ActiveDatasheet** liefert tatsächlich nur einen Objektverweis zurück, wenn aktuell eine Tabelle oder Abfrage in der Datenblattansicht geöffnet ist. Aber auch das ist kein Problem, zumindest nicht gemessen an dem Wunsch des Lesers, die Position und Größe von in der Datenblattansicht geöffneten Elementen zu ermitteln und zu speichern.

Allerdings hilft uns **Screen.ActiveDatasheet** noch nicht weiter, wenn das Objekt, das wir untersuchen wollen, nicht den Fokus hat. Deshalb müssen wir, bevor wir auf eine geöffnete Tabelle in der Datenblattansicht zugreifen wollen, zunächst den Fokus auf diese Tabelle verschieben. Dass sie überhaupt geöffnet ist, können wir voraussetzen, denn wir wollen ja nur die Position und Größe von Tabellen ermitteln, die derzeit im Arbeitsbereich sichtbar sind.

Den Namen der zu untersuchenden Tabelle kennen wir über die **Name**-Eigenschaft auch. Also können wir die **DoCmd.SelectObject**-Methode nutzen, um die zu untersuchende Tabelle oder Abfrage in den Fokus zu setzen – beispielsweise so:

```
DoCmd.SelectObject acTable, objTable.Name
```

Tabelle zum Speichern der Objekteigenschaften

Als Nächstes erstellen wir die Tabelle, in der wir die Größe, Position und Ansicht der aktuell geöffneten Objekte speichern wollen.

Welche Informationen wir speichern wollen, haben wir bereits ausführlich besprochen, daher hier nur der Verweis auf den Entwurf der Tabelle **tblObjects** in Bild 5.

Rechnungsverwaltung: Bestellformular

Nachdem wir das Datenmodell für unsere Rechnungsverwaltung angelegt sowie die Tabellen mit Beispieldaten gefüllt haben, kommt als Nächstes die Benutzeroberfläche zum Verwalten der Kunden-, Produkt- und Bestelldaten an die Reihe. Die dazu notwendigen Formulare stellen wir in mehreren Teilen dieser Beitragsreihe vor. Die Basis ist das Formular zum Anzeigen der Bestellungen, mit dem wir den Kunden auswählen, die Bestelldaten eingeben und die Bestellpositionen hinzufügen können. Die Programmierung dieses Formulars zeigen wir im vorliegenden Beitrag – inklusive Validierung und mehr.

Leichter programmieren mit Testdaten

Das Schöne ist, dass wir im Beitrag **Rechnungsverwaltung: Beispieldaten** (**www.access-im-unternehmen.de/1381**) bereits einige Beispieldatensätze angelegt haben, sodass wir beim Programmieren der Formulare nicht immer noch mühselig von Hand Testdaten eingeben müssen. Dabei sollten wir aber nicht vergessen, dass der Kunde die Anwendung gegebenenfalls ohne Daten erhält. In diesem Fall müssen wir die Formulare auch noch mit leeren Tabellen testen, um zu prüfen, ob das initiale Anlegen von Daten ebenfalls funktioniert und ob die Formulare komplett ohne Daten genauso gut funktionieren.

Reihenfolge beim Anlegen der Formulare

Wenn Sie keine Testdaten zur Verfügung hätten, würden Sie die Formulare logischerweise in einer Reihenfolge erstellen, in der auch die Daten eingegeben werden. Wir würden also zuerst ein Formular zur Eingabe von Anreden, Einheiten und Mehrwertsteuersätzen benötigen, dann für Kunden und Produkte und schließlich für Bestellungen und Bestelldetails (wobei letztere in einem Formular plus Unterformular untergebracht werden).

Mit Testdaten können wir das Erstellen der Formulare allerdings in beliebiger Reihenfolge gestalten. Also beginnen wir doch gleich mal mit dem aufwendigsten Formular – dem zur Eingabe der Bestellungen.

Formulare zur Eingabe von Bestellungen

Zur Eingabe von Bestellungen benötigen wir eigentlich nur ein einfaches Formular, aber zu Bestellungen gehören ja auch noch Bestellpositionen. Und da diese über eine 1:n-

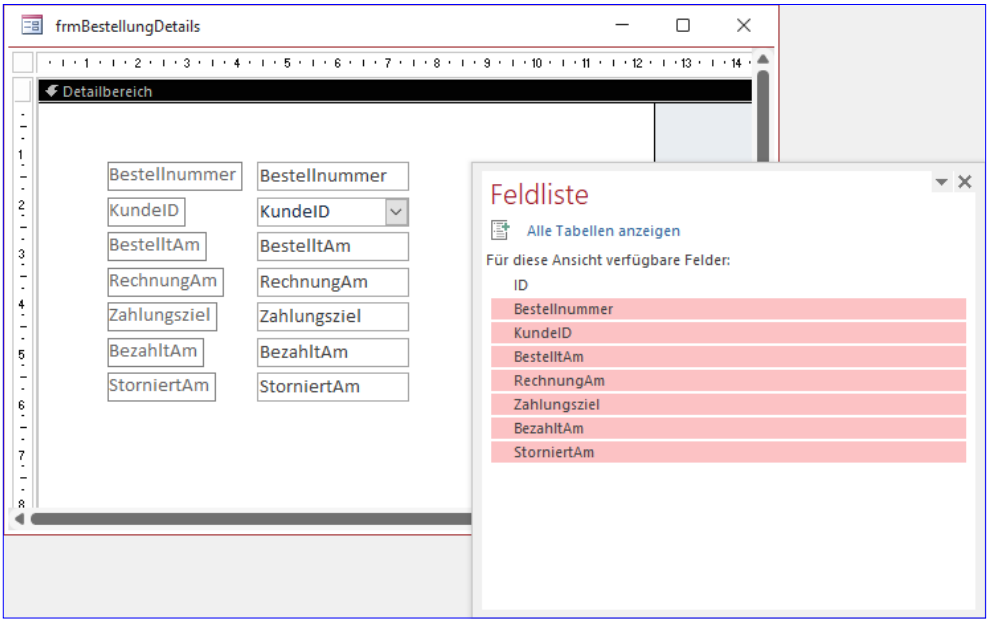


Bild 1: Das Formular **frmBestellungDetails** in der Entwurfsansicht

Beziehung mit den Bestellungen verknüpft sind, bietet sich die Verwendung eines Unterformulars an.

Wir erstellen zuerst das Hauptformular und öffnen es in der Entwurfsansicht. Dieses nennen wir **frmBestellungDetails** und weisen ihm für die Eigenschaft **Datensatzquelle** die Tabelle **tblBestellungen** zu. Danach wechseln wir zur Feldliste und ziehen alle Felder außer **ID** in den Detailbereich des Formularentwurfs (siehe Bild 1). Warum nicht das Feld **ID**? Weil dieses ein rein für die Herstellung von Beziehungen verwendetes Feld ist und der Benutzer dieses ohnehin nicht ändern kann und soll. Die Beschriftungen müssen wir noch ein wenig anpassen, sodass beispielsweise aus **KundeID** die Beschriftung **Kunde** wird oder aus **BestelltAm** die Beschriftung **Bestellt am**. Außerdem fehlen überall noch Doppelpunkte.

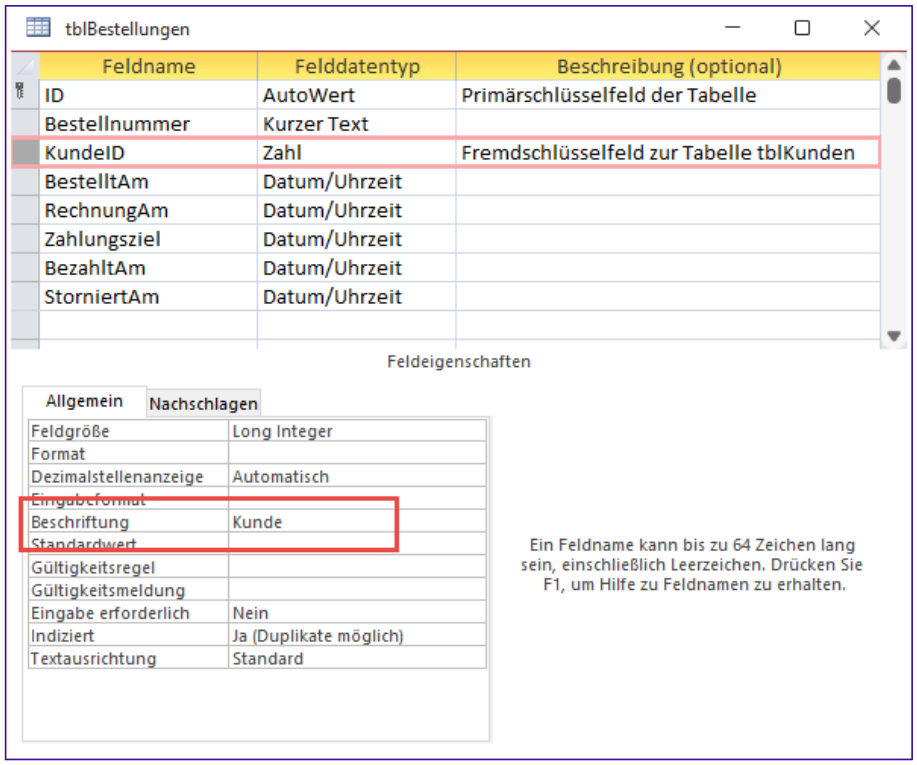


Bild 2: Voreinstellung für die Beschriftung von Feldern, auch in Formularen oder Berichten

Diese Änderungen hätten wir auch schon zu einem früheren Zeitpunkt vorbereiten können, nämlich im Tabellenentwurf. Dort hätten wir diese Bezeichnungen für die Eigenschaft **Beschriftung** der jeweiligen Felder eintragen können. Da wir nicht wissen, ob wir noch weitere Formulare oder Berichte auf Basis dieser Felder erstellen, nehmen wir diese Änderungen noch schnell vor. Dazu öffnen wir die Tabelle **tblBestellungen** nochmals in der Entwurfsansicht und stellen dort für die verschiedenen Felder die gewünschten Beschriftungen in der gleichnamigen Eigenschaft ein (siehe Bild 2).

Doppelpunkte zu Beschriftungen hinzufügen

Und auch den Doppelpunkt hinter der Beschriftung müssen wir nicht von Hand anlegen. Wir können dies für das aktuelle Formular auch so einstellen, dass jedes Feld automatisch einen Doppelpunkt erhält. Dazu müssen Sie jedoch bereits vor dem Hinzufügen der Felder der Datensatzquelle eine bestimmte Eigenschaft einstellen.

Klicken Sie dazu in der Entwurfsansicht des Formulars im Ribbon auf **Formularentwurf|Steuerelemente|Textfeld**, aber fügen Sie kein Textfeld zum Formular hinzu. Das Eigenschaftsfeld zeigt nun einige Eigenschaften an, die nach dem Hinzufügen von Textfeldern nicht mehr erscheinen – zum Beispiel **Mit Doppelpunkt**. Diese Eigenschaft stellen Sie, falls dies noch nicht der Fall ist, auf **Ja** ein (siehe Bild 3).

Anschließend betätigen Sie die **Esc**-Taste, um die Auswahl des Textfeldes abzubrechen. Das Feld **KundeID** haben wir in der Tabelle als Nachschlagefeld ausgelegt. Das heißt, dieses Feld wird beim Ziehen aus der Feldliste in den Formularentwurf als Kombinationsfeld erstellt. Da wir die Änderung zum Anzeigen des Doppelpunkts soeben nur für Textfelder eingestellt haben, müssen wir dies auch noch

für Kombinationsfelder erledigen.

Anschließend ziehen wir erneut die gewünschten Felder aus der Feldliste in den Detailbereich des Formularentwurfs. Das Ergebnis sieht schon viel besser aus – die Beschriftungen aus dem Tabellenentwurf wurden übernommen und auch die Doppelpunkte wurden hinzugefügt (siehe Bild 4).

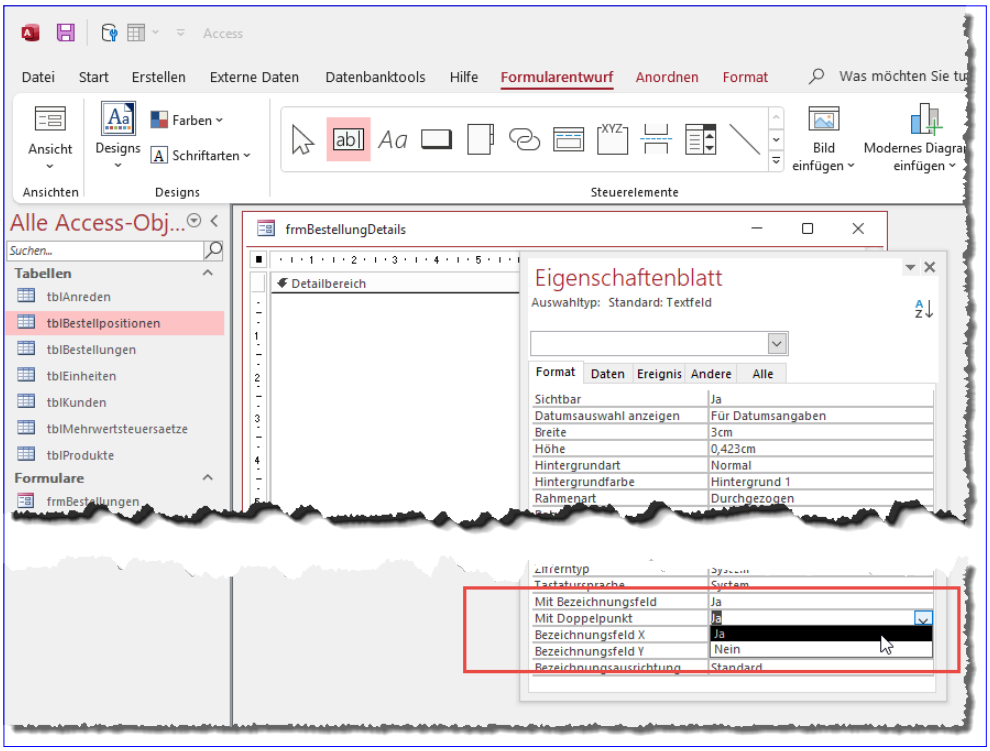


Bild 3: Aktivieren des Doppelpunkts für Beschriftungsfelder

Damit sind die Arbeiten am Hauptformular vorerst erledigt. Sie können nun bereits in die Datenblattansicht wechseln und sehen dort die Testdaten zum Durchblättern. Hier erkennen Sie auch, dass wir die Breite des Kombinationsfeldes **KundeID** noch vergrößert haben (siehe Bild 5).

Unterformular für Bestellpositionen

Damit kommen wir zum Unterformular, das die Bestellpositionen zur jeweiligen Bestellung anzeigen soll. Dieses legen wir unter dem Namen **sfmBestellungDetails** an. Als Datensatzquelle fügen wir die Tabelle **tblBestellpositionen** zu. Diese passen wir zuvor ähnlich wie die Tabelle **tblBestellungen** noch an, indem wir passende Beschriftungen für die Felder **ProduktID** (**Produkt**), **Mehrwertsteuersatz** (**MwSt.-Satz**) und **EinheitID** (**Einheit**) einstellen.

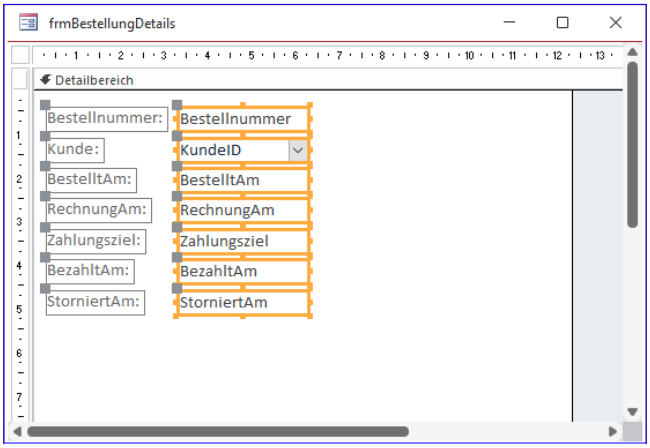


Bild 4: Bezeichnungsfelder mit Doppelpunkten

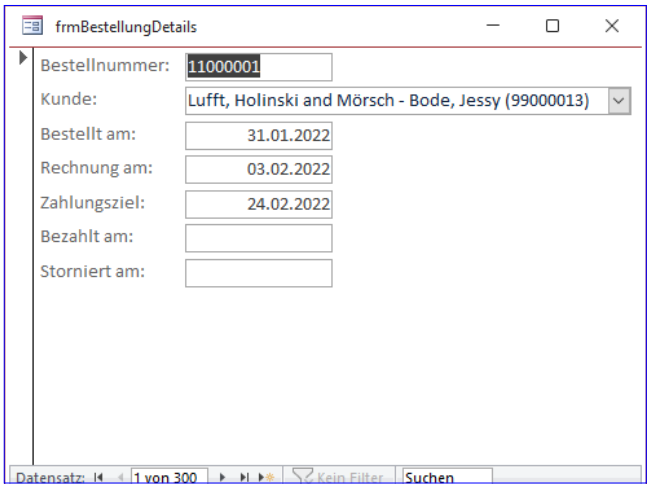


Bild 5: Steuerelemente mit angepasster Breite

Danach ziehen wir alle Felder außer **ID** und **BestellungID** in den Detailbereich des Formulars. ID benötigen wir nicht, weil es das Primärschlüsselfeld der Tabelle ist, und **BestellungID** dient nur dem Herstellen einer Beziehung zum jeweils im Hauptformular angezeigten Datensatz.

Außerdem stellen wir die Eigenschaft **Standardansicht** des Formulars auf **Datenblatt** ein. Die Bestellpositionen sollen im Unterformular tabellarisch dargestellt werden. Danach schließen wir das als Unterformular zu verwendende Formular.

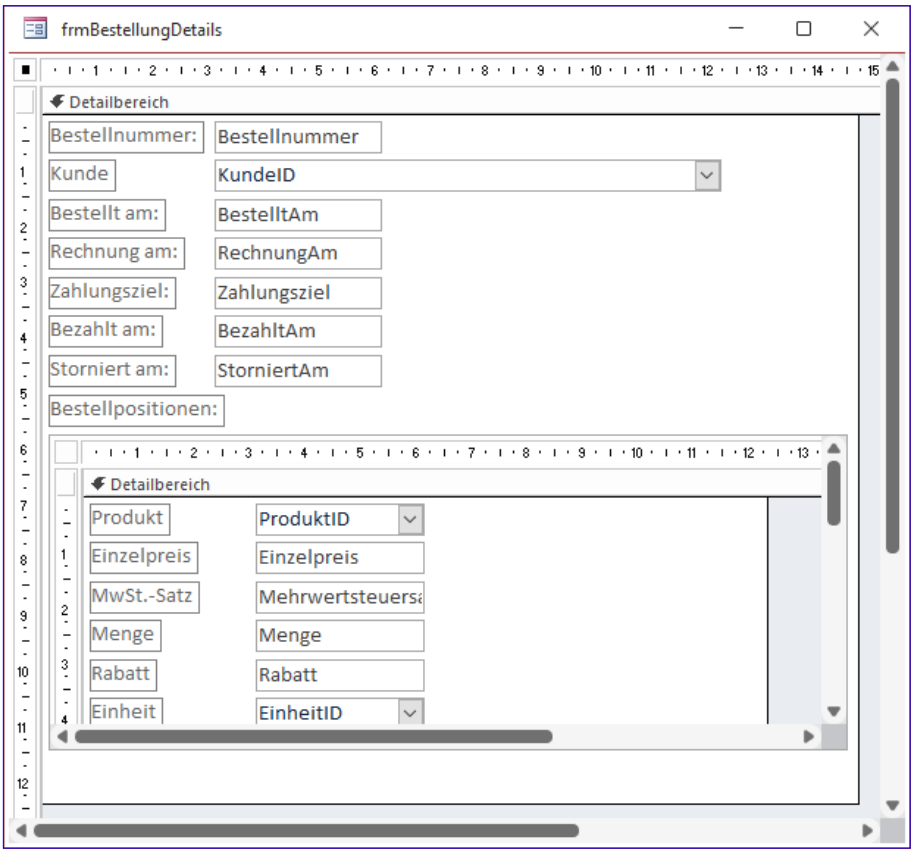


Bild 6: Haupt- und Unterformular

Unterformular zum Hauptformular hinzufügen

Nun öffnen wir wieder das Formular **frmBestellungDetails** in der Entwurfsansicht und fügen das Unterformular **sfmBestellungDetails** zum Hauptformular hinzu, indem wir es aus dem Navigationsbereich in den Detailbereich des Hauptformulars ziehen – und es unter den dort bereits befindlichen Steuerelementen fallenlassen.

Dort platzieren wir es wie in Bild 6 und ändern seine Beschriftung auf **Bestellpositionen**. Für die bessere Bedienbarkeit nehmen wir noch weitere Änderungen vor. So stellen wir die Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** des Unterformular-Steuerelements jeweils auf **Beide** ein.

Dies ändert automatisch die entsprechenden Eigenschaften des Bezeichnungsfeldes der Unterformular-Steuerelemente, die wir wieder auf **Oben** beziehungsweise **Links** zurückstellen.

Wie soll das Unterformular nun wissen, dass es nicht alle Bestellpositionen anzeigen soll, sondern nur die Bestellpositionen, die zum aktuell im Hauptformular angezeigten Datensatz gehören? Die notwendige Einstellung haben wir indirekt bereits getroffen, indem wir eine Beziehung zwi-

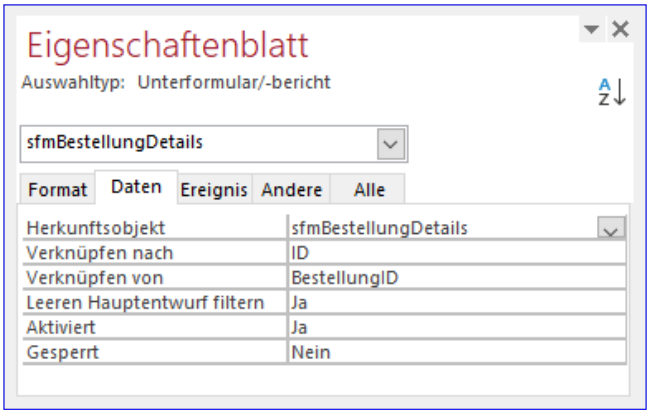


Bild 7: Herstellen der Beziehung der Daten aus Haupt- und Unterformular

schen den Feldern **BestellungID** der Tabelle **tblBestellpositionen** und **ID** der Tabelle **tblBestellungen** definiert haben. Access erkennt diese Beziehung beim Hinzufügen eines Unterformulars und trägt die relevanten Daten direkt in die beiden Eigenschaften **Verknüpfen von** und **Verknüpfen nach** des Unterformular-Steuerelements ein. Das Ergebnis sieht wie in Bild 7 aus.

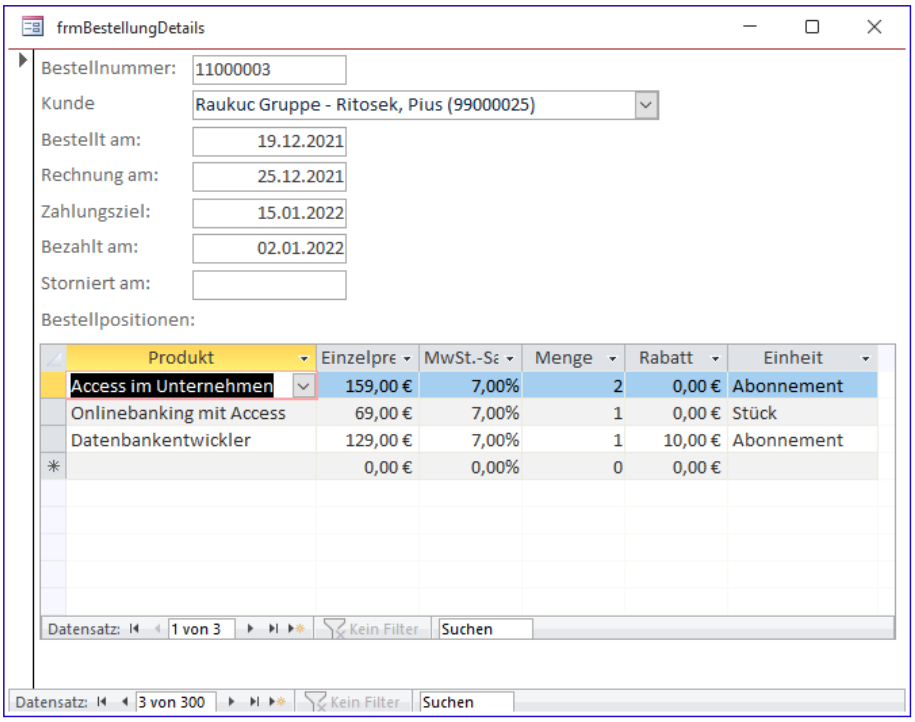


Bild 8: Haupt- und Unterformular in der Formularansicht

Ausprobieren des Bestellungen-Formulars

Nun wechseln wir in die Formularansicht des Formulars **frmBestellungDetails** und finden das Ergebnis aus Bild 8 vor.

Wenn wir durch die Datensätze des Hauptformulars navigieren, zeigt das Unterformular auch jeweils die passenden Datensätze an.

Damit können wir nun auch einen der weiter oben erwähnten Tests durchführen, der das Anlegen einer neuen Bestellung betrifft. Dazu wechseln wir im Hauptformular zu einem neuen, leeren Datensatz. Dieser zeigt sowohl im Hauptformular als auch im Unterformular noch keine Daten an. Wenn wir nun den herkömmlichen Weg gehen und zuerst die Daten der Bestellung im Hauptformular anlegen und dann über das Unterformular Bestellpositionen hinzufügen, läuft alles wie erwartet.

Aber es kann ja auch sein, dass der Kunde anruft und direkt sagt, er möchte Produkt X und Produkt Y erhalten und der neue Mitarbeiter trägt erst einmal die entsprechenden Bestellpositionen ein, ohne die übrigen Bestelldaten hinzuzufügen.

Dann geschieht Folgendes: Da im Hauptformular noch kein Datensatz angelegt wurde, trägt Access in die neuen Datensätze im Unterformular mit den Bestellpositionen den Wert in das Fremdschlüsselfeld **BestellungID** ein, der auch gerade im damit verknüpften Feld **ID** im Hauptformular enthalten ist – und dieser lautet **Null** (siehe Bild 9).

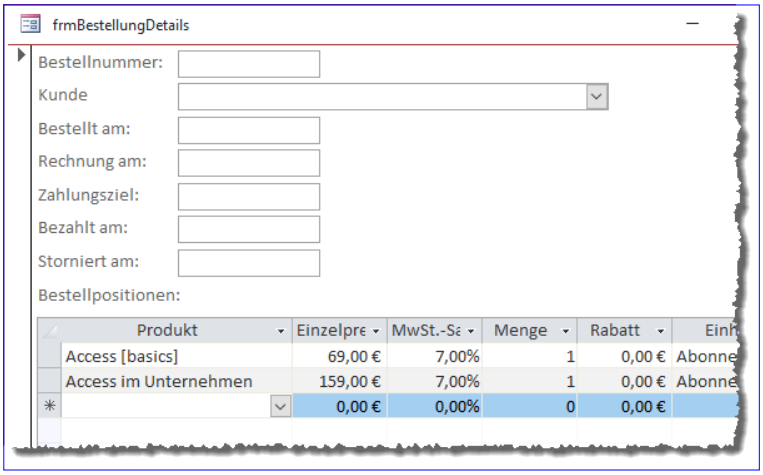


Bild 9: Anlegen von Daten im Unterformular ohne Datensatz im Hauptformular

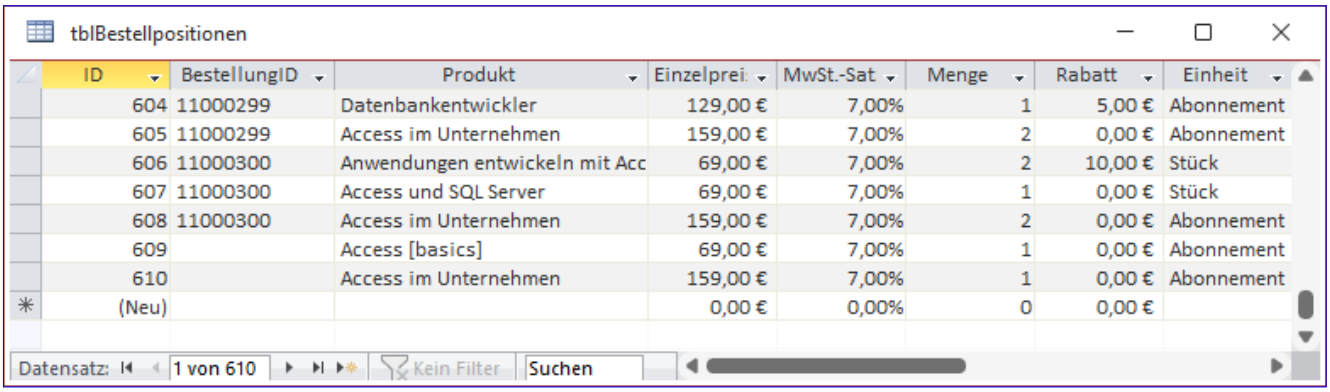


Bild 10: Bestellpositionen ohne **BestellungID**

Wir haben also ein paar Bestellpositionen ohne Zuweisung zu einer Bestellung (siehe Bild 10).

Noch schlimmer wird es, wenn der Mitarbeiter nun nach der Eingabe der Bestellpositionen die übrigen Daten der Bestellung nachträgt. Das sieht zu Beginn noch so aus, als würde es funktionieren (siehe Bild 11).

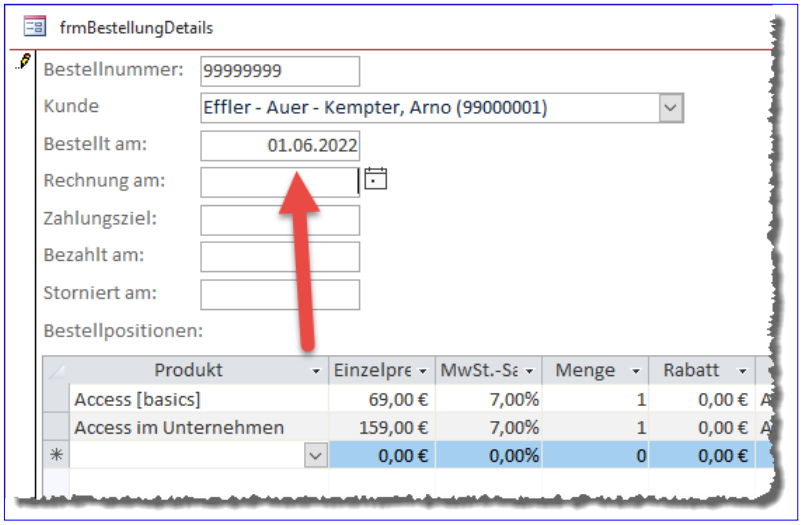


Bild 11: Eingabe der Bestelldaten nach den Bestellpositionen ...

Wenn der Benutzer dann allerdings zu einem anderen Datensatz wechselt und dann zum vorherigen Datensatz zurückkehrt, ist das Unterformular mit den Bestellpositionen plötzlich leer (siehe Bild 12).

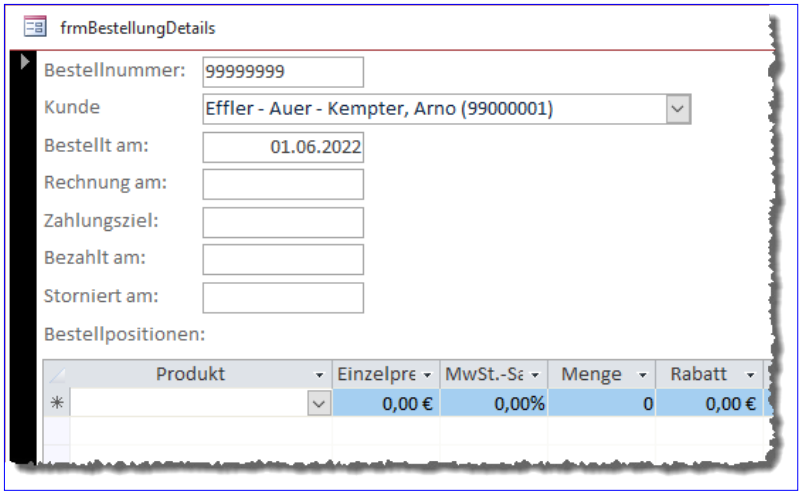


Bild 12: ... führt nach dem Datensatzwechsel zum Verschwinden der vermeintlich verknüpften Daten im Unterformular

Was tun? Wir müssen irgendwie dafür sorgen, dass der Benutzer nur Daten in das Unterformular eingeben kann, wenn der Datensatz im Hauptformular bereits angelegt wurde.

Daten erst ins Hauptformular eingeben

Um sicherzustellen, dass zuerst Daten ins Hauptformular und dann erst ins Unterformular eingegeben werden, wollen wir den Benutzer auf irgendeine Weise davon abhalten, die Daten umgekehrt einzugeben.

Rechnungsverwaltung: Kundendetails

Eine Rechnungsverwaltung, mit der Rechnungen an verschiedene Kunden geschickt werden sollen, benötigt eine Tabelle zum Speichern dieser Kunden. Logisch, dass wir dieser Tabelle auch ein Formular zum komfortablen Bearbeiten der Kunden an die Seite stellen. Dieses enthält allerdings nicht nur die reinen Kundendaten, sondern wir wollen damit auch noch die Bestellungen des jeweiligen Kunden in einem Unterformular anzeigen – und darüber die Anzeige der Bestelldetails ermöglichen.

Unterformular sfmKundeDetails für die Bestellungen

Wir beginnen direkt mit dem Entwurf des Unterformulars zur Anzeige der Bestellungen des Kunden. Dieses wollen wir **sfmKundeDetails** nennen. Diesem fügen wir über die Eigenschaft **Datensatzquelle** gleich die Tabelle **tblBestellungen** hinzu.

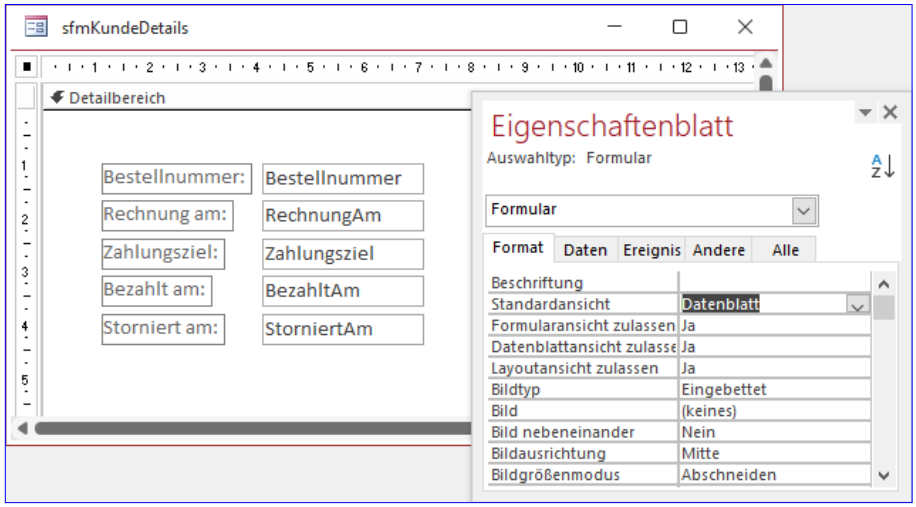


Bild 1: Entwurf des Unterformulars **sfmKundeDetails**

Im Gegensatz zum Unterformular aus dem Beitrag **Rechnungsverwaltung: Bestellübersicht** (**www.access-im-unternehmen.de/1384**), wo wir alle Bestellungen inklusive der Angabe des jeweiligen Kunden in einem Unterformular darstellen, benötigen wir hier nicht mehr die Anzeige des Kunden – dieser wird ja schon im Hauptformular angezeigt, das wir gleich noch erstellen werden. Also fügen wir nun die Felder **Bestellnummer**, **RechnungAm**, **Zahlungsziel**, **BezahltAm** und **StorniertAm** zum Detailbereich des Formularentwurfs hinzu.

Dieser sieht anschließend wie in Bild 1 aus. Damit die Daten in der Datenblattansicht angezeigt werden, legen wir die Eigenschaft **Standardansicht** dort auf den Wert **Datenblatt** fest. Außerdem wollen wir, dass der Kunde die Daten in diesem Unterformular nicht direkt bearbeiten kann. Daher legen wir die Eigenschaften **Anfügen zulassen**, **Löschen zulassen** und **Bearbeitungen zulassen** jeweils auf den Wert **Nein** ein. Damit können wir die Arbeiten an diesem Formular vorerst beenden und dieses schließen.

Hauptformular frmKundeDetails anlegen

Danach legen wir ein weiteres Formular namens **frmKundeDetails** an. Bevor wir diesem das Unterformular **sfmKundeDetails** hinzufügen, müssen wir die Datensatzquelle für das Hauptformular festlegen. So kann Access direkt erkennen, dass es zwischen den Datensatzquellen von Haupt- und Unterformular eine Beziehung gibt und dies entsprechend in den Eigenschaften **Verknüpfen von** und **Verknüpfen nach** des Unterformular-Steuerelements vermerken.

Wenn wir schon die Datensatzquelle definiert haben, können wir auch direkt die gewünschten Felder aus der

Feldliste in den Detailbereich des Formulars ziehen. Dabei berücksichtigen wir alle Felder mit Ausnahme des Feldes **ID**, das nur zu Verknüpfungszwecken gepflegt wird und für den Benutzer unsichtbar bleiben soll (siehe Bild 2).

Falls Sie sich wundern, dass in unserem Formular beispielsweise für das Feld **AnredeID** ein Beschriftungsfeld mit dem Text **Anrede** angelegt wurde: Wir haben direkt im Tabellenentwurf die für die Bezeichnungsfelder gewünschten Texte für die Eigenschaft **Beschriftung** der jeweiligen Felder hinterlegt. Mehr dazu erfahren Sie im Beitrag **Bezeichnungsfelder im Griff** (**www.access-im-unternehmen.de/1380**).

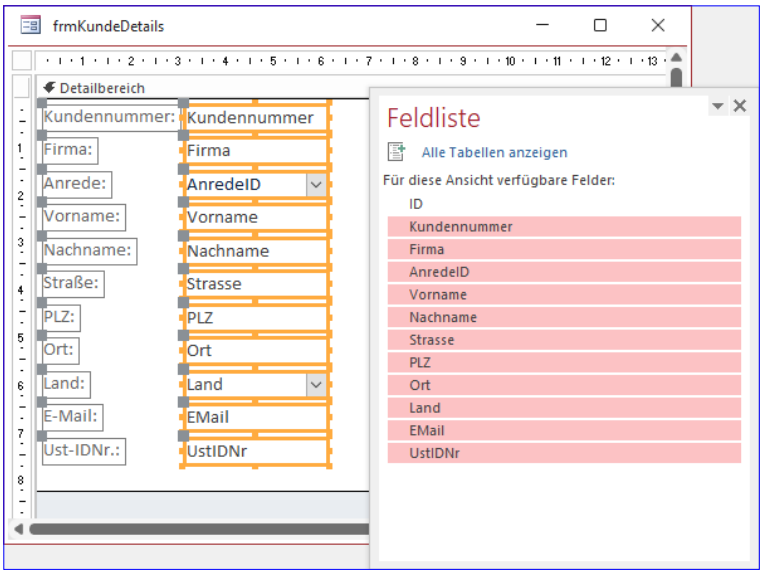


Bild 2: Entwurf des Hauptformulars **frmKundeDetails**

Unterformular zum Hauptformular hinzufügen

Danach teilen wir die Felder auf zwei Spalten auf, sodass wir unten das Unterformular **sfmKundeDetails** platzieren können. Dieses ziehen wir aus dem Navigationsbereich in den Formularentwurf und erhalten nach wenigen Anpassungen das Ergebnis aus Bild 3. Zu diesen Anpassungen gehört neben der Ausrichtung und der Einstellung der Größe das Festlegen der Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** jeweils auf den Wert **Beide**. Damit wird das Unterformular mit dem Hauptformular vergrößert.

Gegebenenfalls können Sie sich nun noch davon überzeugen, dass Access automatisch das Primärschlüsselfeld **ID** der Tabelle **tblKunden** für die Eigenschaft **Verknüpfen nach** und das Fremdschlüsselfeld **KundeID** der Tabelle **tblBestellungen** für die Eigenschaft **Verknüpfen von** eingetragen hat. Dies stellt sicher, dass das Unterformular immer nur die Datensätze der Tabelle **tblBestellungen** anzeigt, die mit dem Datensatz der Tabelle **tblKunden** aus dem Hauptformlar verknüpft sind.

Da wir bereits einige Beispieldatensätze angelegt haben, wie im Beitrag **Rechnungsverwaltung: Beispieldaten** (**www.access-im-unternehmen.de/1381**) beschrieben,

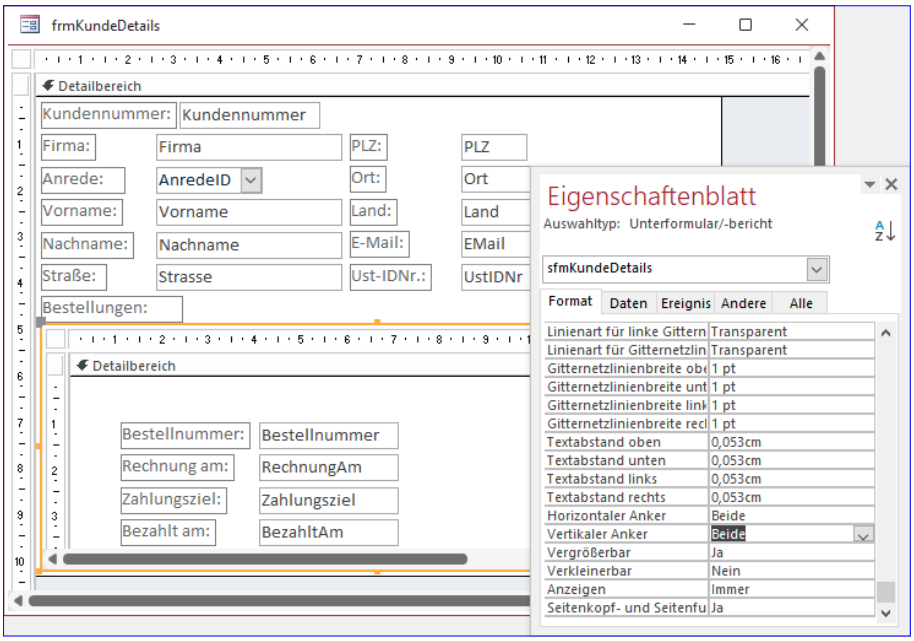


Bild 3: Das Hauptformular **frmKundeDetails** mit dem Unterformular

finden wir beim Wechsel in die Formularansicht bereits einige Beispieldaten vor (siehe Bild 4).

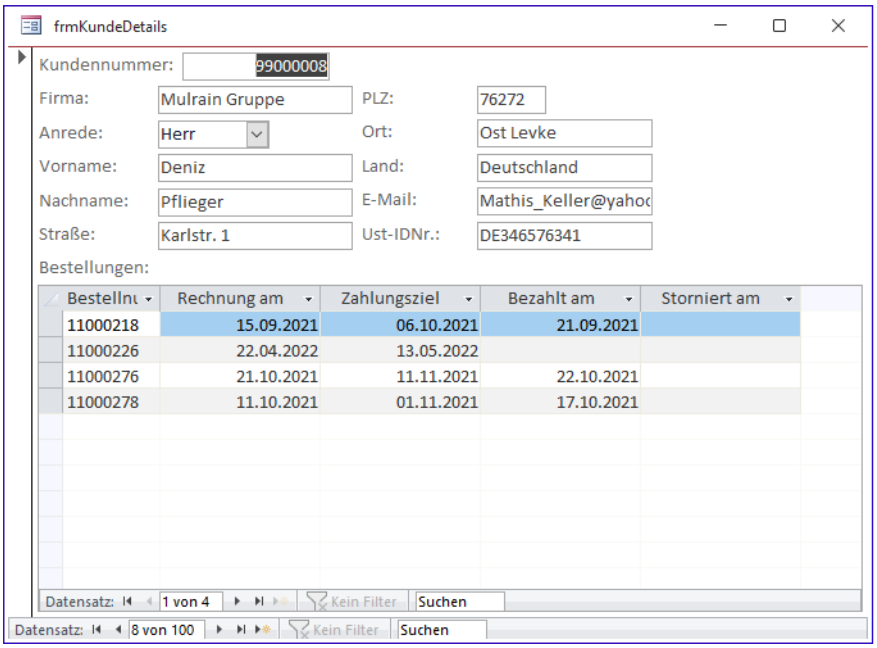


Bild 4: Ein Kunde und seine Bestellungen

Validierung im Hauptformular

Damit können wir uns nun um die Validierung der Eingabefelder im Hauptformular kümmern.

Hier sind Einschränkungen bei folgenden Feldern nötig:

- **Anrede**: Pflichtfeld
- **Vorname**, **Nachname**, **Straße**, **Ort**, **Land**: Pflichtfelder
- **PLZ**: Ausgehend von der vereinfachenden Annahme, wir hätten es mit Adressen aus dem Bereich Deutschland, Österreich und der Schweiz zu tun, muss diese fünf Stellen (für Deutschland) oder vier Stellen (für Österreich und Schweiz) aufweisen.
- **E-Mail**: Diese soll grob validiert werden, also auf ein enthaltenes @-Zeichen und einen Punkt.
- **Ust-IDNr.**: Soll für Deutschland und Österreich geprüft werden auf **DE** plus neun Ziffern beziehungsweise auf **ATU** plus acht Ziffern, für Schweiz muss das Feld leer sein.

Da Pflichtfelder nur beim Speichern des kompletten Datensatzes geprüft werden können und dies für die abhängigen Felder ohnehin der Fall ist, können wir uns hier auf das Ereignis **Vor Aktualisierung** des Formulars konzentrieren. Für dieses hinterlegen wir die Prozedur aus Listing 1. Das Ereignis wird nur beim Versuch ausgelöst, den Datensatz nach vorherigen Änderungen zu speichern – also etwa beim Wechsel zu einem anderen Datensatz oder beim Speichern mit der Tastenkombination **Strg + S**.

Vorab die Information, dass wir alle gebundenen Steuerelemente mit Präfixen versehen haben, in diesem Fall die Textfelder mit **txt** und die Kombinationsfelder mit **cbo**. Auf diese Weise kann man sauber zwischen den Feldnamen der Datensatzquelle und den daran gebundenen Steuerelementen unterscheiden.

Diese Prozedur enthält einige **If...Then**-Bedingungen, die jeweils eine Überprüfung für ein Feld vornehmen. Die erste untersucht beispielsweise, ob das Feld **txtFirma** leer ist. Ist das der Fall, erscheint eine entsprechende Meldung, der Fokus wird auf das Textfeld eingestellt und der Rückgabeparameter **Cancel** auf den Wert **True**.

Außerdem verlassen wir an dieser Stelle mit **Exit Sub** die Prozedur. Das Einstellen des Parameters **Cancel** auf **True** sorgt dafür, dass der **Speichern**-Vorgang, der das Ereignis **Vor Aktualisierung** ausgelöst hat, abgebrochen wird.

Validieren der E-Mail-Adresse

Die E-Mail-Adresse können wir direkt nach der Eingabe validieren und den Benutzer darauf hinweisen, falls die E-Mail-Adresse nicht gültig ist. Deshalb reicht es auch aus,

```
Private Sub Form_BeforeUpdate(Cancel As Integer)
    If Len(Nz(Me!txtKundennummer, "")) = 0 Then
        MsgBox "Bitte geben Sie eine Kundennummer ein.", vbOKOnly + vbExclamation, "Kundennummer fehlt"
        Me!txtKundennummer.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Nz(Me!cboAnredeID, 0) = 0 Then
        MsgBox "Bitte wählen Sie eine Anrede aus.", vbOKOnly + vbExclamation, "Anredefehlt"
        Me!cboAnredeID.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Len(Nz(Me!txtVorname, "")) = 0 Then
        MsgBox "Bitte geben Sie einen Vornamen ein.", vbOKOnly + vbExclamation, "Vorname fehlt"
        Me!txtVorname.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Len(Nz(Me!txtStrasse, "")) = 0 Then
        MsgBox "Bitte geben Sie eine Straße ein.", vbOKOnly + vbExclamation, "Straße fehlt"
        Me!txtStrasse.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Len(Nz(Me!txtPLZ, "")) = 0 Then
        MsgBox "Bitte geben Sie eine PLZ ein.", vbOKOnly + vbExclamation, "PLZ fehlt"
        Me!txtPLZ.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Len(Nz(Me!txtOrt, "")) = 0 Then
        MsgBox "Bitte geben Sie einen Ort ein.", vbOKOnly + vbExclamation, "Ort fehlt"
        Me!txtOrt.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Len(Nz(Me!cboLand, 0)) = 0 Then
        MsgBox "WählenSie ein Land aus.", vbOKOnly + vbExclamation, "Land fehlt"
        Me!cboLand.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Len(Nz(Me!txtEMail, "")) = 0 Then
        MsgBox "Bitte geben Sie eine E-Mail-Adresse ein.", vbOKOnly + vbExclamation, "E-Mail-Adresse fehlt"
        Me!txtEMail.SetFocus
        Cancel = True
        Exit Sub
    End If
End Sub
```

Listing 1: Validieren beim Speichern des Datensatzes

Access-Applikation mit Runtime installieren

Christoph Jüngling, https://www.juengling-edv.de

Office-Dokumente wie Word- oder Excel-Dateien lassen sich mittlerweile auf fast allen Geräten lesen. Wenn das nicht möglich ist, kann man diese oft in die jeweils vorhandene Textverarbeitung oder Tabellenkalkulation importieren. Bei Datenbankanwendungen ist das anders: Dass der Entwickler eine Vollversion von Microsoft Access auf dem Rechner hat, ist Voraussetzung. Aber was ist, wenn wir eine Datenbankanwendung in einem Unternehmen an viele Arbeitsplätze verteilen oder diese online an Kunden verkaufen wollen? Muss in dem Fall für alle User ebenfalls eine Vollversion von Access beschafft werden? Glücklicherweise lautet die Antwort nein. Es gibt eine kostenlose Runtime-Version von Access, die das Nötigste für den Betrieb von Access-Anwendungen mit sich bringt. Der vorliegende Artikel zeigt, welche Vorbereitungen dafür in unserer Applikation erforderlich sind und wie man die Runtime in ein eigenes Setup integriert.

Microsoft Access ist ein tolles Produkt, aber es hat – global gesehen – leider einen entscheidenden Nachteil: Es kostet Geld. Na gut, das kann man verstehen, wir alle müssen ja Miete zahlen, und das geht nicht vom guten Willen (will heißen „Spenden“) allein. Doch glücklicherweise gibt es für einige von uns eine Möglichkeit, wenigstens diese Kosten zu vermeiden – und dabei rede ich natürlich nicht von „Hacks“, nein, das geht ganz legal!

Historisches

Wegen der anfallenden Lizenzkosten kann man nicht grundsätzlich davon ausgehen, dass Access beim User schon installiert ist. Je nach aktueller Lizenzpolitik seitens Microsoft gibt es vielleicht eine Sparversion von Office, die Access außen vor lässt und dafür günstiger ist. Oder die IT-Administration hat entschieden, dass ohnehin niemand Access braucht, weil Excel ja genauso gut ist und auch „Datenbank kann“.

Was auch immer nun der Grund dafür ist, Access nicht zu installieren, für uns Entwickler stellt sich die Frage, wie wir damit umgehen. Immerhin stellt Microsoft schon seit längerem eine „Access-Runtime-Version“ bereit. In der Vergangenheit musste der Entwickler diese gelegentlich bezahlen, damit er sie dann kostenfrei an seine Anwender weitergeben konnte. Seit längerem jedoch ist die Runtime kostenlos herunterladbar. Dies verursacht dem Entwickler also keine Zusatzkosten mehr, und vor allem dürfen wir diese Runtime auch überall kostenfrei installieren.

Auch in Hinsicht dessen, was man als Entwickler darf, scheinen sich die Ansichten bei Microsoft etwas gelockert zu haben. Ich erinnere mich noch an meine früheren Recherchen darüber, nach denen ich die Runtime meinen Kunden nicht hätte bereitstellen dürfen; sie mussten sie sich damals selbst herunterladen (einschließlich der Registrierung). Dazu habe ich aktuell nichts mehr auf der Microsoft-Website gefunden. Selbstverständlich soll dies keine Rechtsberatung darstellen.

Das Problem: Die Kosten

Wie wir alle wissen, ist Microsoft Access eine lizenzpflichtige Software. Das bedeutet, dass jeder, der sie nutzen will, dafür einen gewissen Betrag investieren muss. Das schließt auch alle Anwender mit ein, und das kann für ein kleines Unternehmen ganz schön ins Geld gehen! Natürlich gibt es verschiedene Möglichkeiten, diese Kosten in Grenzen zu halten, zum Beispiel das Action Pack (**https://**

partner.microsoft.com/de-de/membership/action-pack) oder Volumenlizenzen, aber darum soll es hier nicht gehen.

Das mit den Kosten ist übrigens auch dann der Fall, wenn die Anwender eine individuell entwickelte Access-Applikation benutzen, die ja bereits mit einer lizenzierten Access-Version erstellt wurde! Klingt vielleicht unfair, ist aber so, daran können wir nichts ändern. Oder doch?

Die Lösung: Runtime-Version verwenden

Die Erklärung ist einfach: Diese als **.accdb**, **.accde** oder **.accdr** bei den Anwendern installierte (oder einfach aufgespielte) Applikation ist ohne weitere Hilfe leider nicht lauffähig. Wir können mit Access keine eigenständigen Programme schreiben, denn es gibt keinen Compiler/Linker wie beispielsweise bei den Programmiersprachen C++ oder VB.Net.

Es entsteht also keine **.exe**-Datei. Stattdessen muss Access immer mit im Boot sein, es interpretiert die Datenbankdatei und führt die darin beschriebenen Aktionen aus.

Wenn wir allerdings die Runtime-Version von Access nutzen, wird kein Access mehr benötigt, um die Applikation laufen zu lassen. Allerdings gibt es dabei noch ein paar Einschränkungen, die wir uns genauer anschauen müssen.

Unterschiede zur Vollversion

Natürlich gibt es Unterschiede zwischen Runtime und Vollversion, sonst würde ja niemand mehr die Lizenz von Access kaufen, sondern immer gleich zur kostenlosen Runtime-Version greifen. Der wichtigste Unterschied ist der, dass man mit der Runtime eine Datenbank zwar benutzen, aber nicht entwickeln kann. Entwickeln soll hier stellvertretend für alles stehen, was ein Entwickler so macht, unter anderem also Formulare, Berichte und andere Datenbankobjekte bearbeiten und den Navigationsbereich anzeigen und nutzen. Das alles geht in der Runtime nicht mehr, dafür braucht man die Vollversion von Access.

Ein echtes Problem ist das sicher nicht, denn genau dies alles wird der reine Anwender in aller Regel sowieso nicht machen. Und mal ehrlich: Als Entwickler würden wir es sogar begrüßen, wenn der Anwender das alles auch gar nicht kann.

Nur der Vollständigkeit halber sei noch gesagt, dass die Bearbeitung der Daten auch mit der Runtime natürlich ohne Probleme funktioniert!

Vorüberlegungen

Bevor man eine Access-Applikation für die Runtime-Version freigibt, sollten also einige Überlegungen angestellt werden. Wir benötigen sinnvollerweise:

- Frontend-/Backend-Trennung,
- ein eigenes Ribbon,
- einen Mechanismus zum automatischen Start aller benötigten Formulare etc.,
- 32-Bit oder 64-Bit,
- ein Setup
- und wir sollten unser Setup testen.

Frontend-/Backend-Trennung

Die Frontend-/Backend-Trennung und alle damit verbundenen Aktionen (zum Beispiel automatische Tabelleneinbindung) ist generell sinnvoll. Der wichtigste Grund ist, dass damit die Applikation einfach durch einen Austausch der Datei aktualisiert werden kann (mit oder ohne Setup), ohne die bereits eingegebenen Daten zu verlieren.

Eine mögliche Vorgehensweise dazu ist denkbar einfach:

- Im ersten Schritt duplizieren wir mit Hilfe des Windows-Explorers die Datenbankdatei (Copy/Paste).

- In einer der beiden Dateien löschen wir nun alle Tabellen. Diese Datei nennen wir **Frontend**.

- In der anderen Datei löschen wir alles außer den Tabellen. Diese Datei nennen wir **Backend**.

- Nun verknüpfen wir die Tabellen aus dem Backend in das Frontend (**Externe Daten|Neue Datenquelle|Aus Datenbank|Access**).

Weitergehende Informationen finden Sie im Beitrag **Setup für Access-Anwendungen** (**www.access-im-unternehmen.de/1316**) unter **Exkurs: Frontend-/Backend-Trennung**.

Ribbon

Da das Access-eigene Ribbon in der Runtime-Umgebung nicht angezeigt wird, müssen wir als Entwickler ein eigenes vorbereiten. Hierzu gibt es bereits zahlreiche Artikel zu Ribbons. Die Alternative besteht darin, dass wir gar nicht mit Ribbons arbeiten, sondern alle Funktionen der Anwendung aus einem beim Anwendungsstart geöffneten Formular (nicht datengebunden) heraus starten.

Dieses Ribbon sollte mindestens die Icons beinhalten, die der Anwender während der üblichen Arbeiten benötigt, zum Beispiel die Gruppen **Zwischenablage**, **Sortieren und Filtern**, **Datensätze** und **Suchen**. Der Vorteil bei Verwendung der eingebauten Bestandteile ist, dass sie auch ihre Automatik (aktiv/inaktiv je nach Kontext) mitbringen.

In Bild 1 sehen Sie ein Beispiel dafür. Natürlich sind Sie in Ihrer eigenen Applikation frei in der Gestaltung dieses Ribbons. Wenn Sie jedoch keines vorbereiten, wird in der Runtime auch keines enthalten sein.

Automatischer Start

Der automatische Start aller benötigten Objekte (zum Beispiel Ribbon, Formulare et cetera) muss berücksichtigt werden, da der Navigationsbereich nicht zugänglich ist, wenn die Applikation in der Runtime-Umgebung läuft. Wir brauchen also entweder ein **Autoexec**-Makro, ein Startformular mit Code für das **Form_Open**-Ereignis, oder wir nutzen den in dem Artikel **Code beim Öffnen der Anwendung: Ribbon** (**www.access-im-unternehmen.de/1369**) beschriebenen Trick.

Der Start des Ribbons erfolgt durch Access automatisch, wenn wir dieses in der Datenbank korrekt eingetragen haben. Dazu muss die Tabelle **USysRibbons** mit der vorgegebenen Struktur angelegt und mit der Beschreibung der Ribbon-Definition gefüllt sein. Außerdem muss im Menüpunkt **Datei|Optionen|Aktuelle Datenbank|Menüband- und Symbolleistenoptionen|Name des Menübands** der Name des initial zu ladenden Ribbons eingetragen sein. Wie das genau geht, ist in den im vorigen Abschnitt verlinkten Artikeln bereits beschrieben.

Wenn Sie ein Programm wie zum Beispiel den **Ribbon-Admin 2016** (**https://shop.minhorst.com/access-tools/309/ribbon-admin-2016?c=78**) zur Erstellung eines Ribbons verwenden, wird alles Benötigte bereits von diesem erledigt.

Der Start der weiteren Objekte unterscheidet sich in der Runtime-Version nicht von dem, was Sie wahrscheinlich auch in der **.accdb**/**.accde**-Datei längst tun.

Bitbreite

Mit „Bitbreite“ ist selbstverständlich nicht im wörtlichen Sinne die Breite eines Bits gemeint. Näheres dazu gibt es in der Wikipedia. Es geht konkret um die Frage 32-Bit oder

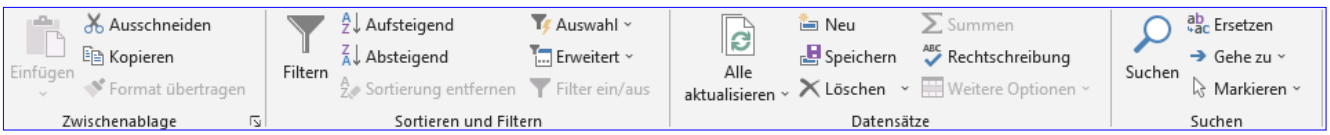


Bild 1: Wichtige Befehle des Ribbons

64-Bit, und zwar im Hinblick auf Access, nicht Windows. Wir müssen im Code einiges vorbereiten, wenn unsere Access-Applikation in beiden Welten lauffähig sein soll. Dieses Thema ist in dem Artikel **VBA unter Access mit 64 Bit** (**www.access-im-unternehmen.de/961**) ausführlich besprochen worden.

Allerdings gibt es im Hinblick auf unsere Installation einen Fallstrick: Die **.accdb**-Datei mag nach sorgfältigen Vorbereitungen unter beiden Bitbreiten laufen, jedoch stimmt das für die kompilierte **.accde**-Datei nicht: Diese muss für 32-Bit und 64-Bit in der jeweiligen Access-Variante erzeugt werden, was unseren Buildprozess ein wenig komplizierter macht. Das ist allerdings nicht neu, sondern schon seit einem Vierteljahrhundert so.

Setup

Damit sind wir gedanklich beim Setup angekommen. Es erleichtert dem Anwender, die Applikation zu installieren, andernfalls müsste er sich um das Aufspielen an die richtige Stelle, die Verknüpfung mit Startmenü und Desktop-Icon und vielleicht noch einiges mehr selbst kümmern. Bei der späteren Deinstallation wiederum müsste er ebenfalls an all dies denken. Wenn der Benutzer das nicht aus eigener Kraft schafft, müsste jedes Mal ein Administrator aktiv werden.

Durch ein Setup bekommen auch unerfahrene Anwender schnell einen lauffähigen Zustand. Sinnvollerweise schließen wir die Installation der Runtime-Version von Access gleich mit ein, sofern diese benötigt wird.

Es gibt noch eine Reihe anderer Vorteile und Möglichkeiten, die wir in den Beiträgen **Setup für Access-Anwendungen** (**www.access-im-unternehmen.de/1316**), **Setup für Access: Umsetzung mit InnoSetup** (**www.access-im-unternehmen.de/1326**), **Setup für Access: Vertrauenswürdige Speicherorte** (**www.access-im-unternehmen.de/1333**) und **Setup für Access-Applikationen, Restarbeiten** (**www.access-im-unternehmen.de/1355**) ausführlich besprochen haben. Im Rahmen dieses Artikels soll nur darauf eingegangen werden, wie man die Runtime-Version in unser Setup integriert.

Test

Für den Test unseres Setups ist es erforderlich, dass wir einen zweiten Rechner haben, egal ob nun physisch oder als virtuelle Maschine. Denn eine Parallelinstallation von Vollversion und Runtime auf einem einzigen Rechner ist problematisch.

Eine Windows-Installation in einer virtuellen Maschine (zum Beispiel Microsoft Virtual PC, VMWare Workstation oder Oracle VirtualBox) bietet sich hierbei an, da mittels eines Snapshots der Grundzustand sehr einfach gesichert und danach jederzeit wiederhergestellt werden kann.

Wenn wir dann auch noch 32-Bit und 64-Bit unterstützen wollen, brauchen wir mehrere virtuelle Maschinen. Beachten Sie dabei, dass Sie die erforderliche Anzahl Lizenzen besitzen. Ich denke aber, das oben verlinkte Actionpack ist schon eine gute Grundlage dafür.

Zu testen wäre in erster Linie, ob

- die Runtime immer installiert wird, wenn sie gebraucht wird.
- die Runtime nicht installiert wird, wenn diese oder die Vollversion bereits installiert sind.
- die Applikation mit der richtigen Bitbreite installiert wird (falls wir diese Unterscheidung machen).

Das bedeutet, dass die folgenden VMs vorhanden sein sollten:

- Mit Access, 32 Bit
- Mit Access-Runtime, 32 Bit
- Mit Access, 64 Bit

XRechnung, Teil 2: Rechnungen einlesen

Nachdem wir im ersten Teil dieser Beitragsreihe gezeigt haben, wie Sie aus Daten wie Kundeninformationen, Rechnungsdatum und Rechnungspositionen ein XML-Dokument im XRechnung-Format erstellen, wollen wir in diesem Beitrag den umgekehrten Weg gehen: Wir wollen die Daten aus einer so generierten Rechnung auslesen und zurück in das Datenmodell schreiben. Dazu sind vor allem Fähigkeiten im Auslesen von XML-Dokumenten erforderlich – und der Umgang mit Namespace-Deklarationen in diesen Dokumenten. Nach der Lektüre dieses Beitrags sind Sie in der Lage, die Daten aus einer XRechnung automatisiert in ein entsprechendes Datenmodell einzulesen.

Ausgangssituation

Für bestimmte Empfänger müssen Rechnungen mittlerweile in einem automatisiert lesbaren Format vorliegen, zum Beispiel im Format **XRechnung**. Dieses Format hat gegenüber Rechnungen im PDF-Format den Vorteil, dass alle Informationen an der entsprechenden in der vorgegebenen XML-Struktur zu finden sind und diese somit maschinell verarbeitet werden können. Im Beitrag **XRechnung, Teil 1: Rechnungen generieren** (**www.access-im-unternehmen.de/1277**) haben wir die Daten aus einer Rechnungsverwaltung per Knopfdruck in ein solches Dokument geschrieben.

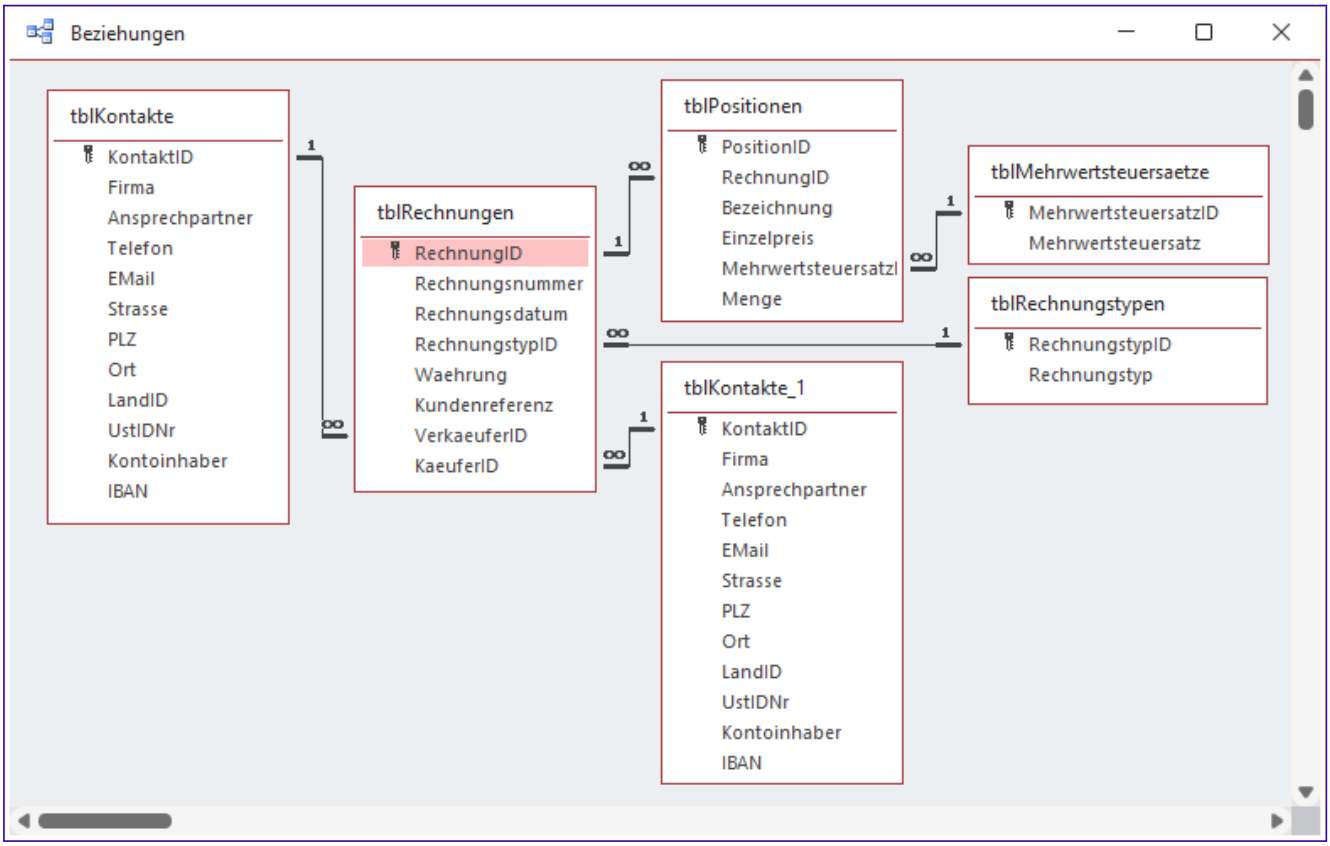


Bild 1: Datenmodell für die Daten aus einer XRechnung

Nun ist es zu erwarten und auch wünschenswert, dass sich solche Formate allgemein durchsetzen, damit niemand mehr Rechnungen in gedruckter Form oder als PDF entgegennehmen und diese händisch verarbeiten muss. Deshalb werden früher oder später alle Unternehmen in der Lage sein müssen, solche Rechnungen zu verarbeiten.

Im vorliegenden Beitrag wollen wir die Daten aus einem solchen XRechnung-Dokument in ein vorgegebenes Datenmodell einlesen können. In einem weiteren Beitrag schauen wir uns dann an, wie wir die eingelesenen Daten in einem Bericht als herkömmliche Rechnung, lesbar für das menschliche Auge, präsentieren können.

Änderung im Datenmodell

Im Vergleich zum Datenmodell des ersten Teils der Beitragsreihe haben wir die Tabelle **tblRechnungen** leicht angepasst. Wir haben ein Feld namens **Rechnungsnummer** zum Speichern der jeweiligen Rechnungsnummer hinzugefügt und das Feld **RechnungID** mit dem Datentyp **Autowert** versehen. Zuvor mussten wir die Beziehung zwischen dem Feld **RechnungID** der Tabelle **tblPositionen** und der Tabelle **tblRechnungen** löschen und diese anschließend erneut anlegen.

Außerdem haben wir der Tabelle **tblPositionen** noch ein Feld namens **Position** hinzugefügt, mit dem die in der XRechnung angegebenen Positionsnummern gespeichert werden können, sowie ein Feld namens **Einheit**.

Beispielhafter Import

Der Standard der XRechnung umfasst natürlich alle denkbaren Informationen. Diese können wir im Rahmen dieses Beitrags nicht alle erfassen. Es geht hier allein darum, die grundlegenden Techniken für die Erfassung der gängigsten Informationen zu präsentieren.

Wenn Sie oder Ihre Kunden es mit Daten in XRechnungen zu tun bekommen, deren Übertragung hier nicht berücksichtigt wurde, so lernen Sie dennoch die Techniken, die nötig sind, die notwendigen Ergänzungen vorzunehmen.

Datenmodell für die Daten aus der XRechnung

Das Datenmodell finden Sie in Bild 1. Jede Rechnung wird grundsätzlich in der Tabelle **tblRechnungen** erfasst und enthält dort einige grundlegende Daten wie das Rechnungsdatum, den Rechnungstyp, die Währung, eine Kundenreferenz sowie Verweise auf den Käufer und den Verkäufer. Käufer und Verkäufer werden in je einem Datensatz der Tabelle **tblKontakte** gespeichert und über Fremdschlüsselfelder der Tabelle **tblRechnungen** zugewiesen. Der Rechnungstyp wird aus einer Tabelle namens **tblRechnungstypen** ausgewählt.

Die einzelnen Rechnungspositionen landen schließlich in der Tabelle **tblPositionen**, die über das Fremdschlüsselfeld **RechnungID** dem jeweiligen Datensatz aus der Tabelle **tblRechnungen** zugewiesen werden.

Jede Position enthält Bezeichnung, Einzelpreis, Mehrwertsteuersatz und Menge, wobei der Mehrwertsteuersatz wiederum über ein Fremdschlüsselfeld aus der Tabelle **tblMehrwertsteuersaetze** ausgewählt wird.

Einlesen der Basisdaten

Der Anfang des XRechnung-Dokuments sieht wie in Listing 1 aus. Hier sehen wir zunächst das Root-Element **ubl**, das einige Namespaces aufweist, um die wir uns später explizit kümmern werden.

Als erste untergeordnete Elemente folgen nun bereits die grundlegenden Rechnungsdaten. Das Element **cbc:CustomizationID** liefert das Format, in dem die Rechnung verfasst wurde, hier in der Version 1.2.

Das Element **cbc:ID** liefert die Rechnungsnummer, **cbc:IssueDate** das Rechnungsdatum. Mit dem Wert **380** im Feld **cbc:InvoiceTypeCode** erhalten wir einen Hinweis auf die Art der Rechnung. **380** steht für **Commercial Invoice**.

Das Element **cbc:DocumentCurrencyCode** liefert das Kürzel für die Währung der Rechnung. In der Regel

finden wir hier die Einstellung **EUR** für Euro vor. Das Feld **cbc:BuyerReference** nimmt die sogenannte Leitweg-ID auf.

Außerdem gibt es noch einige weitere Elemente für verschiedene Referenzen wie **OrderReference**, **ContractDocumentReference**, **ProjectReference** et cetera, die wir an dieser Stelle jedoch nicht verarbeiten wollen.

Wir wollen den Einlesevorgang zunächst VBA-gesteuert ausführen, daher erstellen wir zuerst eine Prozedur, mit der wir die einzulesende Datei per Dateiauswahl-Dialog ermitteln und dann die eigentliche Prozedur zum Einlesen aufrufen. Die Prozedur zum Initialisieren des Einlesevorgangs sieht wie folgt aus:

```
Public Sub Test_XRechnungEinlesen()
    Dim strPfad As String
    strPfad = OpenFileName(CurrentProject.Path, "XRechnung auswählen", "XRechnung (*.xml)")
    XRechnungEinlesen strPfad
End Sub
```

Die Funktion **OpenFileName** finden Sie im Modul **mdlFileDialog**.

Die Funktion XRechnungEinlesen

Danach starten wir mit der Funktion **XRechnungEinlesen** den eigentlichen Einlesevorgang. Dazu prüfen wir in einer **If...Then**-Bedingung zunächst, ob ein Dateipfad mit **strPfad** übergeben wurde und ob die Datei überhaupt vorhanden ist (siehe Listing 2).

Ist das der Fall, erstellt die Funktion ein neues Objekt des Typs **MSXML2.DOMDocument60**. Um diese verwenden zu können, benötigen wir einen Verweis auf die Bibliothek

```
<ubl:Invoice xmlns:ubl="urn:oasis:names:specification:ubl:schema:xsd:Invoice-2"
    xmlns:cac="urn:oasis:names:specification:ubl:schema:xsd:CommonAggregateComponents-2"
    xmlns:cbc="urn:oasis:names:specification:ubl:schema:xsd:CommonBasicComponents-2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:oasis:names:specification:ubl:schema:xsd:Invoice-2
    http://docs.oasis-open.org/ubl/os-UBL-2.1/xsd/maindoc/UBL-Invoice-2.1.xsd">
    <cbc:CustomizationID>urn:cen.eu:en16931:2017#compliant#urn:xoev-de:kosit:standard:xrechnung_1.2</cbc:CustomizationID>
    <cbc:ID>1234</cbc:ID>
    <cbc:IssueDate>2020-09-24</cbc:IssueDate>
    <cbc:InvoiceTypeCode>380</cbc:InvoiceTypeCode>
    <cbc:DocumentCurrencyCode>EUR</cbc:DocumentCurrencyCode>
    <cbc:BuyerReference>1234</cbc:BuyerReference>
    <cac:OrderReference>
        <cbc:ID/>
    </cac:OrderReference>
    <cac:ContractDocumentReference>
        <cbc:ID/>
    </cac:ContractDocumentReference>
    <cac:ProjectReference>
        <cbc:ID/>
    </cac:ProjectReference>
    ...
</ubl>
```

Listing 1: Basisdaten der Rechnung im XML-Dokument

```
Public Function XRechnungEinlesen(strPfad As String) As Long
    Dim objXML As MSXML2.DOMDocument60
    Dim objInvoice As MSXML2.IXMLDOMNode
    Dim db As DAO.Database
    Set db = CurrentDb
    If Not Len(Dir(strPfad)) = 0 Then
        Set objXML = New MSXML2.DOMDocument60
        objXML.SetProperty "SelectionNamespaces", "xmlns:ubl='urn:oasis:names:specification:ubl:schema:xsd:Invoice-2'" _
            & "xmlns:cac='urn:oasis:names:specification:ubl:schema:xsd:CommonAggregateComponents-2' " _
            & "xmlns:cbc='urn:oasis:names:specification:ubl:schema:xsd:CommonBasicComponents-2' " _
            & "xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'"
        objXML.Load strPfad
        Set objInvoice = objXML.childNodes.Item(0)
        XRechnungEinlesen = RechnungsdatenEinlesen(objInvoice, db)
    End If
End Function
```

Listing 2: Startprozedur zum Einlesen der XRechnung

Microsoft XML, v6.0, die wir im **Verweise**-Fenster, das wir mit dem Menübefehl **Extras|Verweise** des VBA-Editors öffnen, markieren müssen (siehe Bild 2).

Nun kommt ein entscheidender Schritt, ohne den wir dieses mit einigen Namespace-Präfixen gespickte XML-Dokument nicht einlesen können: Wir stellen die Eigenschaft **SelectionNamespaces** mit der Methode **SetProperty** auf einen Wert ein, der die Definition der im Element **ubl:Invoice** des XML-Dokuments angegebenen Namespaces enthält.

Danach können wir die **Load**-Methode des Objekts **objXML** nutzen, um die unter **strPfad** angegebene XRechnung zu laden. Dann referenzieren wir das Root-Element, das wir mit **objXML.childNodes.Item(0)** ermitteln, mit der Variablen **objInvoice** und übergeben diese an die erste Unterfunktion **RechnungsdatenEinlesen**. Dieser übergeben wir auch den zuvor mit **CurrentDb** ermittelten Verweis auf das **Database**-Objekt der aktuellen Datenbankdatei.

Die Funktion **RechnungsdatenEinlesen** soll nach erfolgreichem Einlesen in die Tabelle **tblRechnungen** den Primärschüsselwert des neu erstellten Datensatzes zurückliefern, den wir wiederum als Rückgabewert der aufrufenden Funktion **XRechnungEinlesen** festlegen.

Funktion zum Einlesen der Rechnungsdaten

Die Funktion **RechnungsdatenEinlesen** erwartet das Root-Element sowie einen Verweis auf das aktuelle **Database**-Objekt als Parameter (siehe Listing 3).

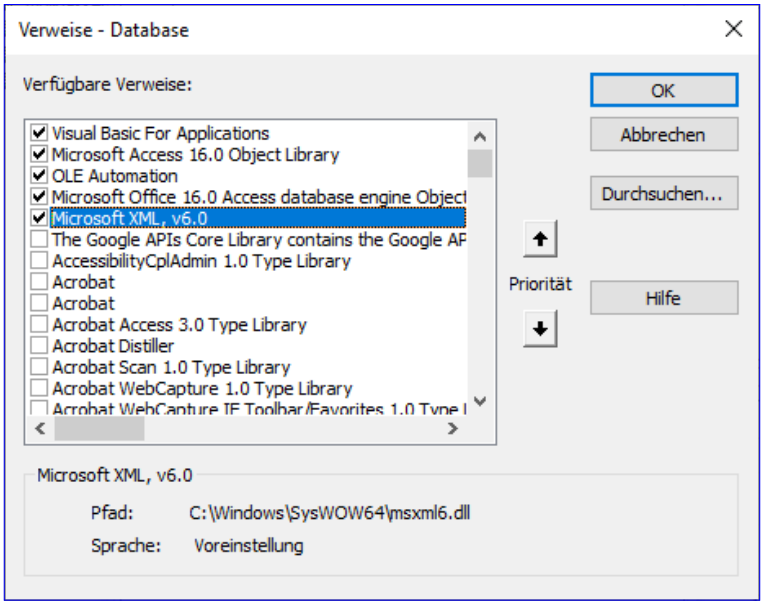


Bild 2: Hinzufügen eines Verweises auf die XML-Bibliothek

Sie deklariert einige Variablen des Typs **IXMLDOMNode**, um Unterelemente von **objInvoice** aufzunehmen, sowie ein Element des Typs **IXMLDOMNodeList** für die Rechnungspositionen in der XRechnung. Außerdem benötigen

```
Private Sub RechnungsdatenEinlesen(objInvoice As MSXML2.IXMLDOMElement, db As DAO.Database) As Long
    Dim rst As DAO.Recordset
    Dim strID As String
    Dim datIssueDate As Date
    Dim lngInvoiceTypeCode As Long
    Dim strDocumentCurrencyCode As String
    Dim strBuyerReference As String
    Dim objKaeufer As MSXML2.IXMLDOMNode
    Dim objVerkaeufer As MSXML2.IXMLDOMNode
    Dim objPaymentMeans As MSXML2.IXMLDOMNode
    Dim objInvoiceLines As MSXML2.IXMLDOMNodeList
    Dim lngVerkaeuferID As Long
    Dim lngKaeuferID As Long
    Dim lngRechnungID As Long
    strID = TextEinlesen(objInvoice, "cbc:ID")
    datIssueDate = TextEinlesen(objInvoice, "cbc:IssueDate")
    lngInvoiceTypeCode = TextEinlesen(objInvoice, "cbc:InvoiceTypeCode")
    strDocumentCurrencyCode = TextEinlesen(objInvoice, "cbc:DocumentCurrencyCode")
    strBuyerReference = TextEinlesen(objInvoice, "cbc:BuyerReference")
    Set objKaeufer = ElementEinlesen(objInvoice, "//cac:AccountingCustomerParty/cac:Party")
    lngKaeuferID = AdresseEinlesen(objKaeufer, db)
    Set objVerkaeufer = ElementEinlesen(objInvoice, "cac:AccountingSupplierParty/cac:Party")
    lngVerkaeuferID = AdresseEinlesen(objVerkaeufer, db)
    Set objPaymentMeans = ElementEinlesen(objInvoice, "cac:PaymentMeans")
    ZahlungsinformationenSchreiben objPaymentMeans, db, lngVerkaeuferID
    Set rst = db.OpenRecordset("SELECT * FROM tblRechnungen WHERE 1=2", dbOpenDynaset)
    With rst
        .AddNew
        !Rechnungsnummer = strID
        !Rechnungsdatum = datIssueDate
        !RechnungstypID = lngInvoiceTypeCode
        !Waehrung = strDocumentCurrencyCode
        !Kundenreferenz = strBuyerReference
        !VerkaeuferID = lngVerkaeuferID
        !KaeuferID = lngKaeuferID
        lngRechnungID = !RechnungID
        .Update
    End With
    Set objInvoiceLines = ElementeEinlesen(objInvoice, "cac:InvoiceLine")
    PositionenEinlesen objInvoiceLines, db, lngRechnungID
    RechnungsdatenEinlesen = lngRechnungID
End Sub
```

Listing 3: Hauptprozedur zum Einlesen der Rechnungsdaten

wir eine **Recordset**-Variable namens **rst**, der wir die Tabelle **tblRechnungen** zuweisen und über die wir den neuen Rechnungsdatensatz anlegen. Daneben dienen einige weitere Variablen wie **strID**, **datIssueDate** et cetera dazu, die Werte aus der XRechnung nach dem Auslesen und vor dem Eintragen in den neuen Datensatz temporär aufzunehmen.

```
<cac:AccountingCustomerParty>
    <cac:Party>
        <cac:PostalAddress>
            <cbc:StreetName>Teststr. 1</cbc:StreetName>
            <cbc:AdditionalStreetName/>
            <cbc:CityName>Berlin</cbc:CityName>
            <cbc:PostalZone>12121</cbc:PostalZone>
            <cac:Country>
                <cbc:IdentificationCode>DE</cbc:IdentificationCode>
            </cac:Country>
        </cac:PostalAddress>
        <cac:PartyTaxScheme>
            <cbc:CompanyID>DE232323232</cbc:CompanyID>
            <cac:TaxScheme>
                <cbc:ID>VAT</cbc:ID>
            </cac:TaxScheme>
        </cac:PartyTaxScheme>
        <cac:PartyLegalEntity>
            <cbc:RegistrationName>Müller AG</cbc:RegistrationName>
        </cac:PartyLegalEntity>
        <cac:Contact>
            <cbc:Name>Klaus Müller</cbc:Name>
            <cbc:Telephone>0123-2121212</cbc:Telephone>
            <cbc:ElectronicMail>klaus@mueller.de</cbc:ElectronicMail>
        </cac:Contact>
    </cac:Party>
</cac:AccountingCustomerParty>
```

Listing 4: Element mit den Daten des Rechnungsempfängers

Basisdaten der Rechnung aus der XRechnung einlesen

Die ersten Anweisungen lesen die Werte der entsprechenden Elemente aus der XRechnung ein. Dazu nutzen diese verschiedene Hilfsfunktionen, die wir am Ende des Beitrags beschreiben. Die erste heißt **TextEinlesen** und sucht aus einem **IXMLDomNode**-Element den Text des mit dem zweiten Parameter übergebenen Wertes aus – in der ersten Anweisung beispielsweise das Element **cbc:ID**. Dieser Wert landet in der Variablen **strID**. Das Gleiche geschieht mit den Elementen **cbc:IssueDate**, **cbc:InvoiceTypeCode**, **cbc:DocumentCurrencyCode** und **cbc:BuyerReference**. Damit erhalten wir die Basisdaten der Rechnung.

Danach nutzen wir eine weitere Hilfsfunktion namens **ElementEinlesen**, um das Unterelement mit dem Namen **cac:AccountingCustomerParty/cac:Party** mit der Variablen **objKaeufer** zu referenzieren. Dieser Teilbereich der XRechnung sieht beispielsweise wie in Listing 4 aus.

Nachdem wir dieses Element aufgerufen haben, rufen wir die Funktion **AdresseEinlesen** auf und übergeben dieser das **IXMLDOMNode**-Element aus **objKaeufer** sowie einen Verweis auf das **Database**-Objekt. Der Übersicht halber beschreiben wir diese Funktion erst später. An dieser Stelle interessiert uns nur das Ergebnis dieser Funktion. Diese schreibt nämlich die Käuferdaten in die Tabelle **tblKontakte** und liefert den Wert des Primärschlüssels des neu hinzugefügten Datensatzes zurück. Diesen können wir dann in der Variablen **lngKaeuferID** zwischenspeichern und beim Erstellen eines neuen Rechnungsdatensatzes direkt in die Tabelle **tblRechnungen** schreiben.

Auf die gleiche Weise lesen wir auch noch die Daten des Verkäufers ein. Diese finden wir im Element **cac:AccountingSupplierParty/cac:Party**. Auch für dieses Element rufen wir die Funktion **AdresseEinlesen** auf, was dazu führt, dass auch der Verkäufer als neuer Datensatz in der

EPC-QR-Code per COM-DLL erstellen

Spätestens seit sich das Onlinebanking immer mehr auf das Smartphone verschiebt, wird das Eingeben von Rechnungsdaten wie langen IBANs oder Verwendungszwecken zu einer undankbaren Aufgabe. Und auch wenn die Papierrechnungen weniger werden und sich die Daten von PDF-Rechnungen leicht per Copy und Paste übertragen lassen, so ist doch der EPC-QR-Code eine tolle Erleichterung: Dieser QR-Code enthält alle für eine Überweisung benötigten Daten und viele Onlinebanking-Apps bieten mittlerweile die Möglichkeit, solche Codes mithilfe der Smartphone-Kamera einzulesen. Um dieses Feature in Access-Berichten bereitzustellen, benötigen wir erst einmal eines: Ein Tool, mit dem wir solche QR-Codes erstellen können. Dieser Beitrag zeigt, wie wir eine .NET-DLL programmieren, die uns diese Aufgabe abnimmt.

Für viele Anwendungsbereiche bietet .NET Bibliotheken und Tools, die dem VBA-Entwickler auf direktem Wege nicht zugänglich sind. Zum Glück wird es immer einfacher, diese Helferlein in Form beispielsweise von COM-DLLs auf der Basis von .NET zu programmieren und diese mit einer Schnittstelle auszustatten, auf die wir auch von Access aus leicht zugreifen können.

Für die Aufgabe, die wir uns in diesem Beitrag vornehmen, benötigen wir die folgenden Dinge:

- eine Definition, wie die Informationen zusammengestellt werden, die für das Einlesen in eine Überweisung nötig sind,
- eine Bibliothek, die das Erstellen von QR-Codes erlaubt und
- ein VB.NET-Projekt, das die ersten beiden zusammenführt und in Form einer COM-DLL für den Zugriff von VBA aus verfügbar macht.

Schließlich benötigen wir noch eine Datenbank, welche die damit zu erstellenden Bilddateien mit QR-Codes nutzt und in Rechnungsberichte einbettet, sodass diese vom Rechnungsempfänger per Smartphone-Kamera erfasst und für das schnelle Eintragen der Rechnungsdaten in das Überweisungsformular genutzt werden kann. Diese beschreiben wir in einem weiteren Beitrag namens **Rechnungsbericht mit EPC-QR-Code** (**www.access-im-unternehmen.de/1400**) in der nächsten Ausgabe.

Wie sieht der Inhalt des EPC-QR-Codes aus?

Als Erstes schauen wir uns an, welchen Aufbau der Text hat, der als Grundlage für das Erstellen des EPC-QR-Codes genutzt wird. EPC steht für European Payment Code.

Ein Beispiel für den Inhalt finden wir auf Wikipedia, die praktischerweise direkt die Bezahldaten für eine Spende an Wikipedia selbst im Beispielcode untergebracht haben. Der Inhalt sieht wie folgt aus:

```
BCD
001
1
SCT
BFSWDE33BER
Wikimedia Foerdergesellschaft
DE33100205000001194700
EUR123.45


Spende fuer Wikipedia
```

In diesem Beispiel sehen Sie einige Leerzeilen, und auch die letzte Zeile könnte noch mit Inhalt gefüllt sein. Daher hier die Zeilen mit der Beschreibung der Inhalte:

- Zeile 1 (Beispiel: **BCD**): Service Tag (fester Wert)
- Zeile 2 (Beispielwert: **002**): Version (**001** oder **002**)
- Zeile 3 (Beispielwert: **2**): Zeichencodierung (1=UTF-8, 2=ISO 8859-1, 3=ISO 8859-2, 4=ISO 8859-4, 5=ISO 8859-5, 6=ISO 8859-7, 7=ISO 8859-10, 8=ISO 8859-15)
- Zeile 4 (Beispielwert: **SCT**): Identifikation, dreistelliger Code – derzeit nur **SCT** (**SEPA Credit Transfer**)
- Zeile 5 (Beispielwert: **BFSWDE33BER**): BIC der Empfängerbank (in Version **001** erforderlich; in Version **002** innerhalb des EWR optional)
- Zeile 6 (Beispielwert: **Wikimedia Foerdergesellschaft**): Name des Zahlungsempfängers (maximal 70 Zeichen Text)
- Zeile 7 (Beispielwert: **DE33100205000001194700**): Internationale Bankkontonummer (IBAN) des Zahlungsempfängers
- Zeile 8 (Beispielwert: **EUR123.45**): Zahlungsbetrag (Format **EUR#.##**, zwischen 0.01 und 999999999.99, optional)
- Zeile 9 (Beispielwert: **CHAR**): Zweck (max. vierstelliger Code analog dem Textschlüssel nach DTA-Verfahren, optional)
- Zeile 10 (Beispielwert **RF18 5390 0754 7034**): Referenz (strukturierter 35-Zeichen-Code gem. ISO 11649 RF Creditor Reference, optional)
- Zeile 11 (Beispielwert: **Spende fuer Wikipedia**): Verwendungszweck (unstrukturierter maximal 140 Zeichen langer Text, optional)
- Zeile 12 (Beispielwert: ./.): Hinweis an den Nutzer (maximal 70 Zeichen, optional)

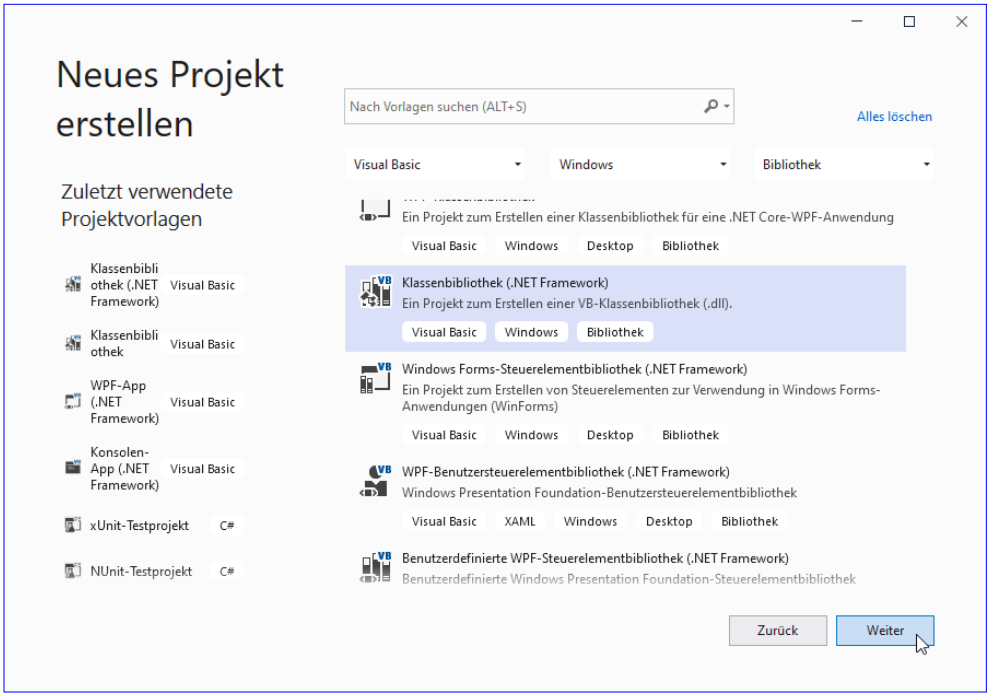


Bild 1: Auswahl des geeigneten Projekttyps

Wir wollen in diesem Beispielprojekt lediglich die wichtigsten Daten unterbringen. Wie Sie oben gesehen haben, legt man in der zweiten Zeile die Version des EPC-QR-Codes fest, der eigentlich nur einen Unterschied bewirkt: Die erste Version verlangt nach einem BIC, die zweite tut das nicht. Um das Beispiel einfach zu halten, nutzen wir hier die Version 2 und geben den BIC nicht an. Die ersten vier Zeilen enthalten Standardwerte, einige Zeilen bleiben leer,

so müssen wir nur die folgenden Daten mit der zu erstellenden COM-DLL entgegennehmen:

- Zahlungsempfänger
- IBAN
- Zahlungsbetrag
- Zweck

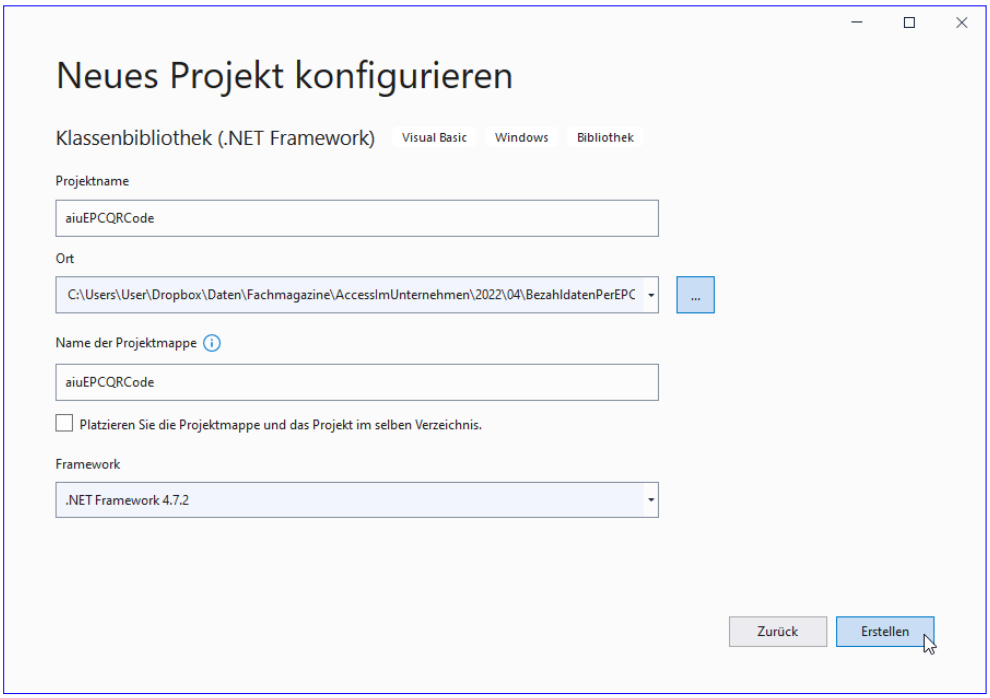


Bild 2: Eingabe von Zielordner und Projektname

.NET-Projekt für die COM-DLL erstellen

Damit können wir direkt mit der Erstellung der COM-DLL in Visual Studio beginnen. Visual Studio ist in der Community-Edition kostenlos. Scheuen Sie sich also nicht, es auszuprobieren – es macht Spaß, mal mit einer alternativen Entwicklungsumgebung zu arbeiten!

Nachdem Visual Studio gestartet ist, hier in der Version 2019, legen Sie ein neues Projekt an. Dazu verwenden wir die Vorlage **Klassenbibliothek (.NET-Framework)**, die Sie wie in Bild 1 auswählen und mit einem Klick auf die **Weiter**-Schaltfläche bestätigen.

Mit dem folgenden Dialog ist das Erstellen des Projekts bereits abgeschlossen: Hier geben Sie noch den Namen für das Projekt ein und legen den Zielordner fest.

Sie müssen keinen neuen Ordner anlegen, Visual Studio legt einen Ordner mit dem zuvor festgelegten Namen des Projekts an (siehe Bild 2).

Klassennamen ändern

Wenn Sie den Projektmappen-Explorer einsehen, finden Sie hier eine einzige Klasse namens **Class1.vb**. Den Namen dieser Klasse ändern wir auf **EPCQRCodeGenerator.vb** (siehe Bild 3).

Die anschließende Meldung, ob auch alle Verweise darauf geändert werden sollen, akzeptieren wir mit **Ja**. Damit ändert sich auch gleich der Name der Klassenbezeichnung im Modul selbst auf **EPCQRCodeGenerator**.

NuGet-Paket mit QR-Code-Bibliothek importieren

Bevor wir uns an das Programmieren begeben, fügen wir die für das eigentliche Erstellen des QR-Codes notwendige

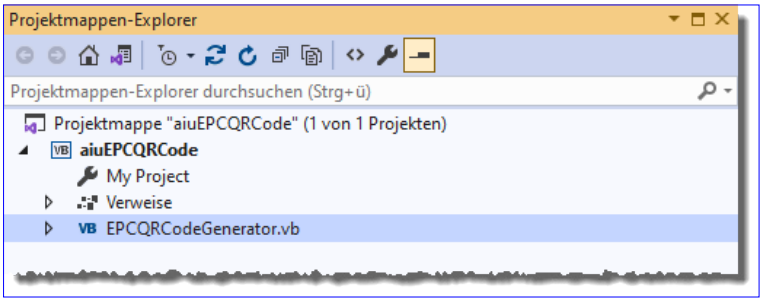


Bild 3: Ändern des Namens der einzigen Klasse

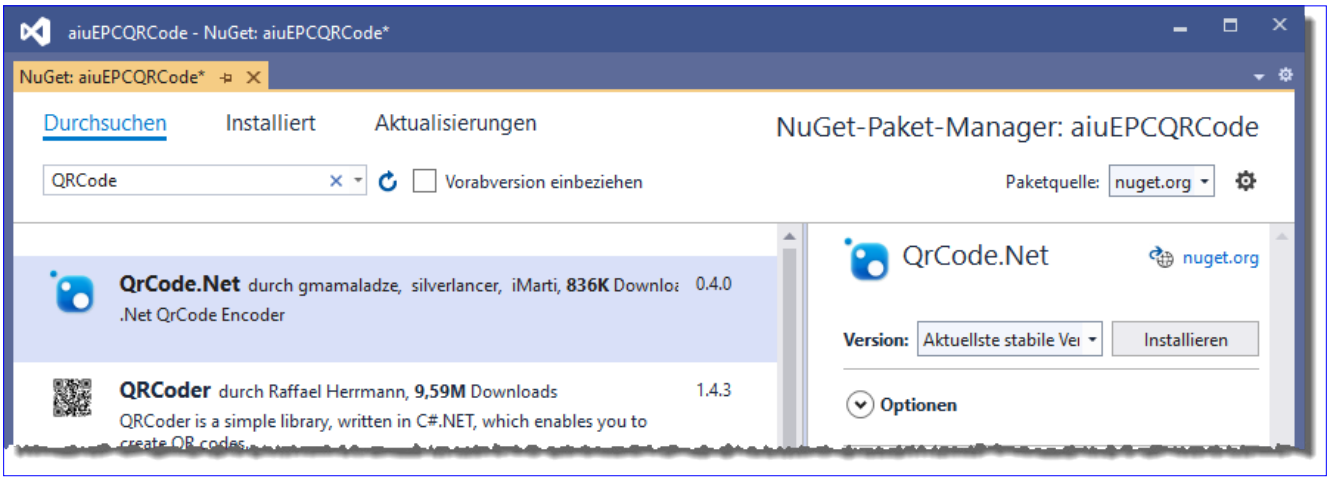


Bild 4: Hinzufügen des NuGet-Pakets **QrCode.Net**

Bibliothek zum Projekt hinzu. Dazu rufen Sie mit dem Menübefehl **Projekt|NuGet-Pakete verwalten ...** den NuGet-Manager auf. Hier klicken Sie auf den Bereich **Durchsuchen** und geben im Suchen-Feld den Text **QRCode** ein.

Daraufhin erscheint das Paket **QrCode.Net**, das wir anklicken und mit einem Klick auf die nun erscheinende Schaltfläche **Installieren** zum Projekt hinzufügen (siehe Bild 4). Dies legt ein paar Elemente im Projekt an, um die wir uns aber nicht weiter kümmern müssen.

Namespaces importieren

Um in der Datei **EPCQRCodeGenerator.vb** auf die Elemente des hinzugefügten NuGet-Pakets und auf einige andere Elemente zugreifen zu können, machen wir diese mit einigen **Imports**-Anweisungen verfügbar.

Diese landen direkt ganz oben in der Datei, also noch über der **Public Class**-Anweisung, und sehen wir folgt aus:

```
Imports System.Runtime.InteropServices
Imports Gma.QrCodeNet.Encoding
Imports Gma.QrCodeNet.Encoding.Windows.Render
Imports System.Drawing
```

Für den Import des Namespaces **System.Drawing** müssen wir überdies noch einen entsprechenden Verweis auf die gleichnamige Bibliothek hinzufügen. Dazu öffnen Sie mit dem Menübefehl **Projekt|Verweis hinzufügen...** den Dialog **Verweis-Manager** und suchen dort unter Assemblys nach dem Suchbegriff **System.Drawing**. Den gefundenen Eintrag fügen Sie durch Setzen eines Hakens

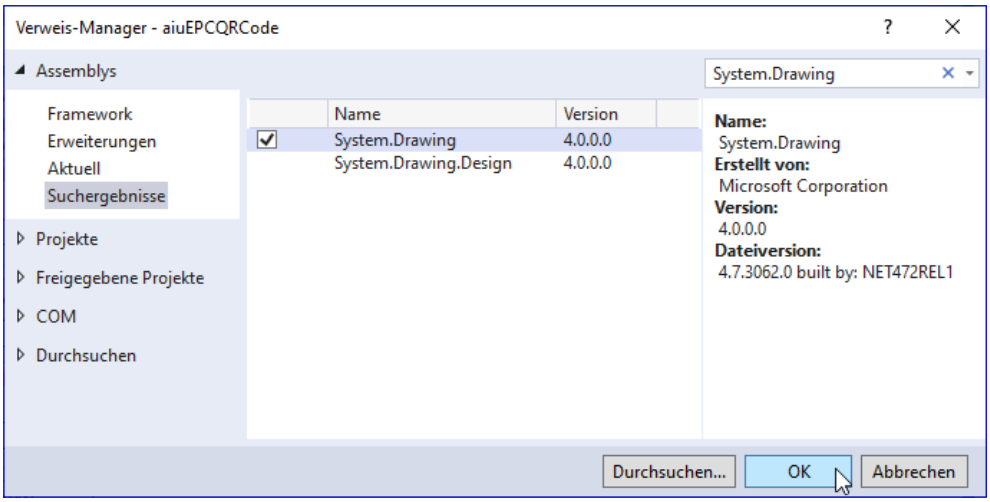


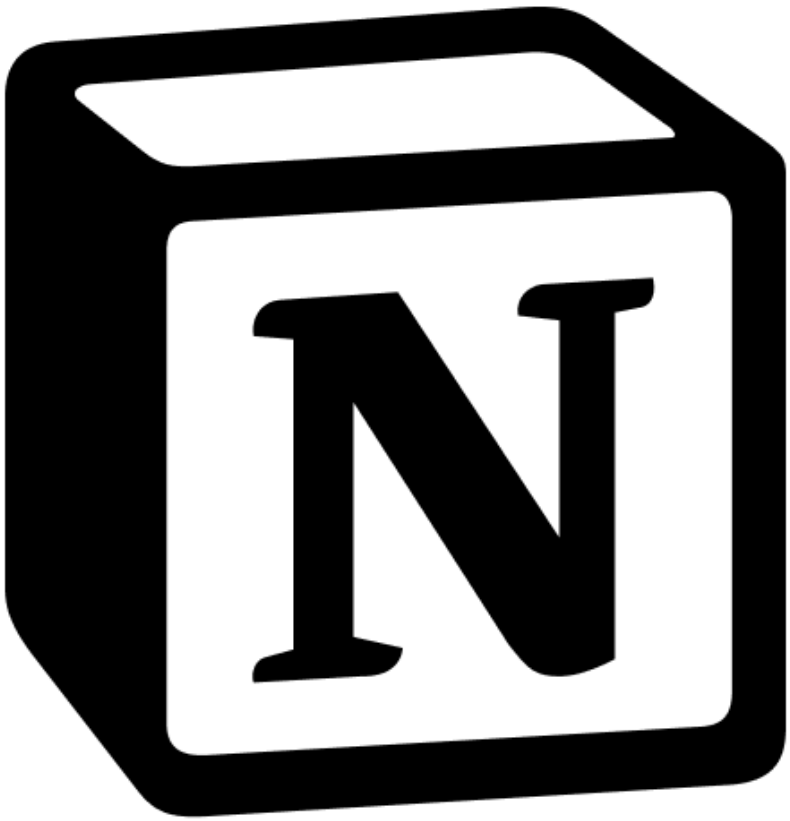
Bild 5: Hinzufügen von Verweisen zum Projekt

HAUFE.
Ausgabe 06/2022

ACCESS IM UNTERNEHMEN

PRODUKTIVITÄT MIT NOTION

Die neue Produktivitäts-App Notion soll viele Aufgaben vereinfachen. Noch besser klappt das, wenn wir von Access aus per VBA darauf zugreifen können (ab Seite 48).



In diesem Heft:

www.access-im-unternehmen.de
Mat-Nr. 01583-5134

Produktivität mit Notion

Es gibt mittlerweile unzählige Blogs, YouTube-Kanäle, Bücher, Tools et cetera rund um das Thema Produktivität. Das Ziel ist, die täglichen Abläufe zu optimieren, weniger Arbeit gleichzeitig auf dem Schreibtisch zu haben, Aufgaben nach Themen zu sortieren und abzuarbeiten und die Zeitfenster für diese Aufgaben vorzudefinieren. Gerade für Access-Entwickler gibt es da viele nützliche Ansätze. Ein praktisches Tool ist Notion. Dabei handelt es sich um ein webbasiertes Tool, mit dem man Listen, Datenbanken oder Kalender verwalten kann. Aber was hilft schon ein Tool, auf das wir nicht mit Access zugreifen können? Genau das erledigen wir in einer Beitragsreihe, die in dieser Ausgabe startet.

Notion ist in einer einfachen Version kostenlos erhältlich und bietet sich deshalb zum Ausprobieren an. Hat man einmal begonnen, seine Projekte, Aufgaben und andere Daten in die Notion-Datenbanken zu schreiben und die verschiedenen Ansichten zu ihrer Darstellung zu nutzen, zum Beispiel als Liste mit Filter- und Sortiermöglichkeiten, in einem Kanban-Board oder, wenn die Daten eine Datumsangabe enthalten, auch in einem Kalender, will man nicht mehr davon weg. Einen kleinen Einblick in die Möglichkeiten bietet der Beitrag **Produktivität mit Notion steigern** ab Seite 48.

Doch wie eingangs erwähnt, wollen wir als Access- und VBA-Entwickler wissen, wie wir programmgesteuert auf die in Notion hinterlegten Daten zugreifen und diese gegebenenfalls ändern oder sogar neue Daten anlegen können. Damit können wir dann nicht nur die Daten aus Access-Tabellen, sondern gegebenenfalls auch Termine, Kontakte et cetera aus Outlook mit Notion synchronisieren. Wie wir per VBA aus einer Access-Datenbank auf Notion zugreifen, stellen wir im Beitrag **Mit Access auf Notion zugreifen** ab Seite 61 vor.

In zwei weiteren Beiträgen schauen wir uns das Ribbon von Access genauer an. Der erste heißt **Kontextabhängige tab-Elemente im Ribbon** (ab Seite 2). Hier untersuchen wir die sogenannten kontextabhängigen Tabs. Diese werden im Kontext mit bestimmten Objekttypen eingeblendet. So erscheint beispielsweise das Tab mit der Beschriftung **Tabelle Felder** erst, wenn der Benutzer eine Tabelle in der Datenblattansicht öffnet. Wir schauen uns an, welche kontextabhängigen Tabs es gibt und wie wir diese einblenden, ausblenden oder erweitern können.

Das gleiche Thema greifen wir nochmals im folgenden Beitrag **Ribbontab beim Öffnen eines Formulars anzeigen** ab Seite 9 auf. Hier stellen wir die verschiedenen Möglichkeiten vor, um Ribbonbefehle nur dann einzublenden, wenn der Benutzer ein bestimmtes Formular geöffnet hat.

Der Beitrag **Dynamische Bereichshöhe im Endlosformular** zeigt ab Seite 17, wie wir die Höhe mehrerer gleichzeitig angezeigter Bereiche im Endlosformular an die Höhe des Formulars anpassen können – so, dass eine bestimmte Anzahl Datensätze erscheinen.

Unsere Beitragsreihe zur Rechnungserstellung setzen wir mit den beiden Beiträgen **Rechnungsverwaltung: Kundenübersicht mit Suche** (ab Seite 21) und **Kunden nach bestellten Produkten filtern** (ab Seite 31) fort.

Schließlich stellen wir im Beitrag **Prüfen, ob Datenbank geöffnet ist** ab Seite 43 eine Funktion vor, mit der wir sichergehen können, dass wir exklusiv auf eine Datenbankdatei zugreifen können.

Und unter **Dateien per VBA öffnen** zeigen wir ab Seite 45, wie Sie Dateien mit der dafür geeigneten Anwendung starten können.

Jetzt aber viel Spaß beim Lesen und Ausprobieren!

Ihr André Minhorst

Kontextabhängige tab-Elemente im Ribbon

Wenn Sie schon einmal benutzerdefinierte Ribbondefinitionen in einer Ihrer Anwendungen eingesetzt haben, kennen Sie vielleicht auch schon die kontextabhängigen Tabs, die man mit ein paar Extra-Elementen definiert und die gemeinsam mit dem jeweils zugeteilten Formular angezeigt werden. Die Besonderheit ist, dass diese kontextabhängigen tab-Elemente, auf Englisch Contextual Tabs, optisch etwas anders angezeigt werden und zusätzlich zu den aktuell angezeigten Ribbon-Tabs erscheinen. Es gibt jedoch nicht nur kontextabhängige Tabs für Formulare und Berichte, sondern auch noch weitere, die beispielsweise in der Entwurfsansicht verschiedener Elemente erscheinen oder in der Datenblattansicht. In diesem Beitrag schauen wir uns an, welche es gibt und wie wir diese Tabs selbst erweitern oder anpassen können.

Im Beitrag **Ribbontab beim Öffnen eines Formulars anzeigen** (**www.access-im-unternehmen.de/1391**) schauen wir uns an, wie wir bei Anzeige eines Formulars ein benutzerdefiniertes, kontextabhängiges Ribbon-Tab einblenden können.

Dort nutzen wir unter anderem die Möglichkeit, ein neues Tab im Ribbon als kontextabhängiges **tab**-Element zu platzieren. Dazu fügen wir unter dem **ribbon**-Element nicht direkt das **tabs**-Element und darin die **tab**-Elemente ein, sondern schalten noch ein **contextualTabs**-Element und ein **tabSet**-Element dazwischen.

Dieses **tabSet**-Element können wir nicht mit einem individuellen Wert für das Attribut **id** versehen, sondern es gibt nur das Attribut **idMso**. Für dieses hinterlegen wir dort den Wert **TabSetFormReportExtensibility**.

Erst darunter können wir dann die **tab**-Elemente und die untergeordneten Elemente anlegen, die dann in einem optisch hervorgehobenen **tab**-Element angezeigt werden – allerdings nur, wenn die Ribbondefinition aktiv ist und ein Formular oder Bericht in der Formular- oder Berichtsansicht angezeigt wird.

Kontextabhängige tab-Elemente nur für Formulare?

Dieses kontextabhängige Element fügen wir normalerweise einer benutzerdefinierten Ribbondefinition hinzu, die über die Eigenschaft **Name des Menübands** einem Formular zugewiesen wird.

Dann erscheint das kontextabhängige **tab**-Element auch nur, wenn das Formular angezeigt wird.

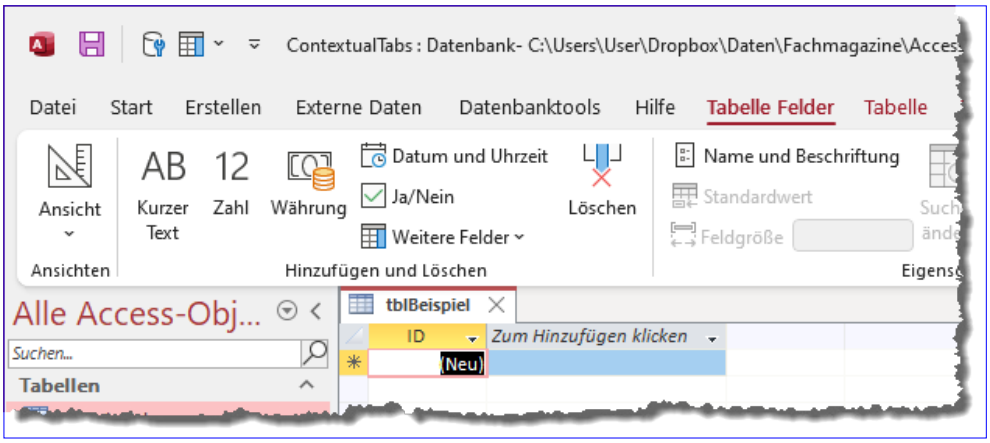


Bild 1: Kontextabhängige Tabs bei Anzeige einer Tabelle in der Datenblattansicht

Eingebaute tabSet-Elemente

Neben den kontextabhängigen **tabSet**-Elementen gibt es auch einige eingebaute **tabSet**-Elemente, die zum Beispiel beim Anzeigen verschiedener Datenbankobjekte rechts neben den Standard-Tabs erscheinen.

Ein Beispiel ist die Anzeige einer Tabelle in der Datenblattansicht (siehe Bild 1).

Benutzerdefinierte tabSet-Elemente

Wenn man an benutzerdefinierte kontextabhängige **tab**-Elemente denkt, fällt einem zuerst das für Formulare und Berichte ein. Wir können aber auch für alle anderen Objekte in verschiedenen Ansichten kontextabhängige Erweiterungen hinzufügen oder auch die vorhandenen Erweiterungen anpassen. Wir schauen uns zuerst einmal an, wie wir den eingebauten **tabSet**-Elementen eigene **tab**-Elemente hinzufügen können.

Eingebaute tabSet-Elemente erweitern

Bevor wir die eingebauten **tabSet**-Elemente um eigene **tab**-Elemente mit entsprechenden **group**-Elementen und darin enthaltenen Steuerelementen ergänzen können, müssen wir zuerst einmal die Werte kennen, die wir für das Attribut **idMso** angeben müssen, um die **tabSet**-Elemente zu referenzieren.

Deshalb schauen wir uns diese zuerst an. Dabei fangen wir mit den aktuell, also mit den Access-Versionen 2016 und 2019 kompatiblen **idMso**-Werten an:

- **TabSetTableToolsDatasheet**: Wird mit der Datenblattansicht einer Tabelle eingeblendet und enthält die beiden **tab**-Elemente **Tabelle Felder** und **Tabelle**.
- **TabSetTableToolsDesign**: Wird mit der Entwurfsansicht einer Tabelle eingeblendet und bietet das **tab**-Element **Tabellenentwurf** an.
- **TabSetQueryTools**: Wird mit der Entwurfsansicht und der SQL-Ansicht von Abfragen angezeigt und enthält das **tabSet**-Element namens **Abfrageentwurf**.
- **TabSetFormToolsLayout**: Wird mit der Layoutansicht von Formularen eingeblendet und zeigt die **tabSet**-Elemente **Entwurf des Formularlayouts**, **Anordnen** und **Format** an.
- **TabSetFormTools**: Erscheint mit der Entwurfsansicht eines Formulars und enthält die **tabSet**-Elemente **Formularentwurf**, **Anordnen** und **Format**.
- **TabSetFormDatasheet**: Wird mit der Datenblattansicht von Formularen angezeigt und liefert das **tabSet**-Element **Formulardatenblatt**.

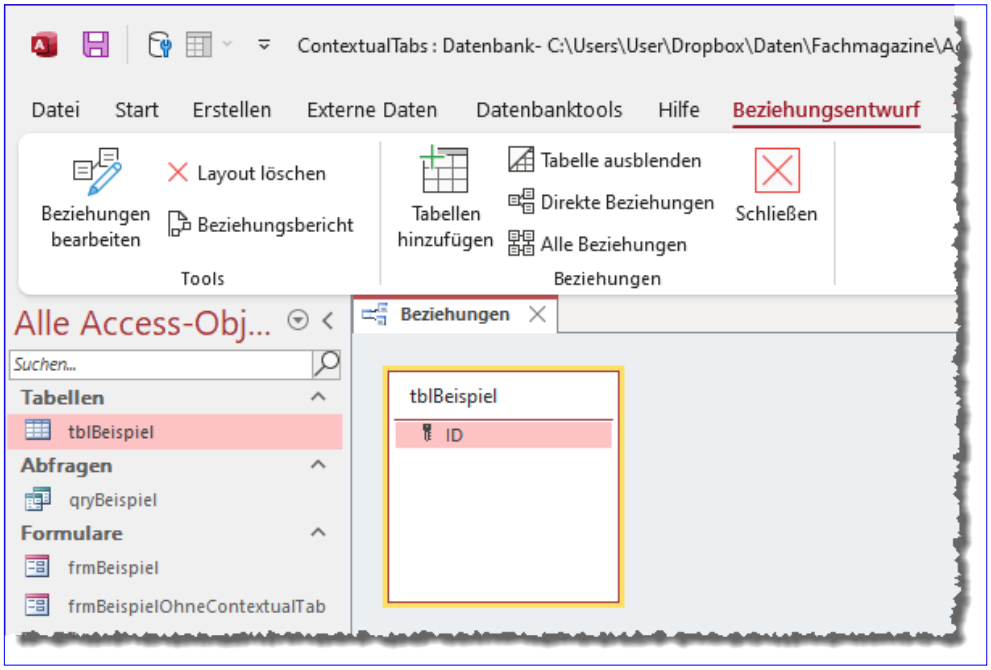


Bild 2: Contextual Tab für das Beziehungsfenster

- **TabSetReportToolsLayout**: Wird mit der Layoutansicht von Berichten aktiviert und enthält die **tabSet**-Elemente **Berichtslayoutentwurf**, **Anordnen**, **Format** und **Seite einrichten**.

- **TabSetReportTools**: Erscheint mit der Entwurfsansicht von Berichten und zeigt die **tabSet**-Elemente **Berichtsentwurf**, **Anordnen**, **Format** und **Seite einrichten** an.

- **TabSetMacroTools**: Wird mit der Entwurfsansicht eines Makros angezeigt und zeigt das **tabSet**-Element **Makroentwurf** an.

- **TabSetRelationshipTools**: Wird mit dem Beziehungsfenster eingeblendet und stellt das **tabSet**-Element Beziehungsentwurf bereit (siehe Bild 2).

Das **tabSet**-Element **TabSetFormReportExtensibility** ist ein Sonderfall. Eigentlich dachten wir, es würde nur angezeigt werden, wenn es in einer Ribbondefinition definiert ist, die einem Formular oder einem Bericht zugeordnet ist. Natürlich haben wir ausprobiert, dieses in einer Ribbondefinition anzulegen, die wir der Eigenschaft **Name des Menübands** der Anwendung selbst in den Access-Optionen zugewiesen haben. Haben wir dann ein Formular in der Formularansicht geöffnet, ist dieses jedoch nicht erschienen.

Wenn wir allerdings ein **tabSet** des Typs **TabSetFormReportExtensibility** für das Anwendungsribbon definieren und auch für ein Formular und das Formular dann anzeigen, dann erscheinen beide benutzerdefinierte **tabSet**-Elemente.

Es gibt noch einige weitere **idMso**-Werte für **tabSet**-Elemente, die für ältere Access-Versionen noch funktionieren und die vielleicht für den einen oder anderen Leser noch interessant sind. Zunächst die Elemente für die Anzeige von Tabellen in den Pivot-Ansichten, die schon länger nicht mehr verfügbar sind:

- **TabSetPivotTableAccess**

- **TabSetPivotChartAccess**

Noch länger fehlt die Möglichkeit, Access-Projekte (**.adp**) zu verwenden. Dennoch der Vollständigkeit halber hier die **idMso**-Werte für die verschiedenen Ansichten der Elemente, die es nur in **.adp**-Datenbanken gibt:

- **TabSetAdpFunctionAndViewTools**

- **TabSetAdpStoredProcedure**

- **TabSetAdpSqlStatement**

- **TabSetAdpDiagram**

Erweitern um benutzerdefinierte tab-Elemente

Wenn wir die eingebauten **tabSet**-Elemente um eigene tab-Elemente erweitern wollen, um den verschiedenen Ansichten von Tabellen, Abfragen, Formularen, Berichten, Makros und dem Beziehungsfenster neue Steuerelemente hinzuzufügen, benötigen wir also jeweils das **contextualTabs**-Element, darin das entsprechende **tabSet**-Element mit einer der oben angegebenen **idMso**-Werte und darin die gewünschten **tab**-, **group**- und Steuerelemente.

In Listing 1 finden Sie Beispiele für alle **tabSet**-Elemente, denen wir jeweils ein **tab**-Element hinzugefügt haben. Wenn Sie diese Ribbondefinition mit dem Wert **Main** im Feld **RibbonName** in der Tabelle **USysRibbons** speichern, die Anwendung neu starten, den Wert **Main** für die Eigenschaft **Name des Menübands** in den Access-Optionen einstellen und die Anwendung nochmals neu starten, wird diese Definition angewendet.

Es erscheint dann beispielsweise bei der Anzeige einer Tabelle in der Entwurfsansicht ein zusätzliches **tab**-Element wie in Bild 3. Das ist nicht nur hilfreich bei der Identifikation der einzelnen **tabSet**-Elemente, sondern kann auch

Ribbontab beim Öffnen eines Formulars anzeigen

Sie kennen sicher die Ribbontabs, die erscheinen, wenn Sie bestimmte Objekte in Access öffnen. Wenn Sie eine neue Tabelle anlegen, erscheint beispielsweise ein Tab namens »Tabellenentwurf«. Wechseln Sie zur Datenblattansicht der Tabelle, erscheinen die Tabs »Tabelle Felder« und »Tabelle«. Die Gemeinsamkeiten dieser Elemente sind, dass diese sich optisch ein wenig von den links davon befindlichen Tabs unterscheiden. Wie genau, hängt von der jeweils verwendeten Access-Version ab. In diesem Beitrag schauen wir uns an, wie wir überhaupt Ribbons mit einem Formular einblenden und dieses aktivieren und wie wir kontextabhängige Ribbons programmieren können.

Verschiedene Möglichkeiten

Wir stellen in diesem Beitrag die folgenden Varianten für das Anzeigen eines Ribbontabs beim Öffnen eines Formulars vor:

- Einfaches zusätzliches Tab, das mit dem Öffnen des Formulars erscheint, aber nicht aktiviert wird
- Tab, das alle anderen Elemente ausblendet und deshalb den Fokus erhält
- Tab, das als kontextabhängiges Tab ausgelegt ist und direkt mit dem Anzeigen des Formulars erscheint und aktiviert wird – aber nur beim ersten Aufruf des Formulars
- Zusätzliches Tab, das mit dem Formular erscheint und auch direkt aktiviert wird, und zwar bei jedem Öffnen des Formulars erneut.
- Die dort verwendete Technik wenden wir dann auch noch auf ein kontextabhängiges Tab an, damit dieses nicht nur beim ersten Anzeigen aktiviert wird, sondern bei jedem Öffnen des Formulars.

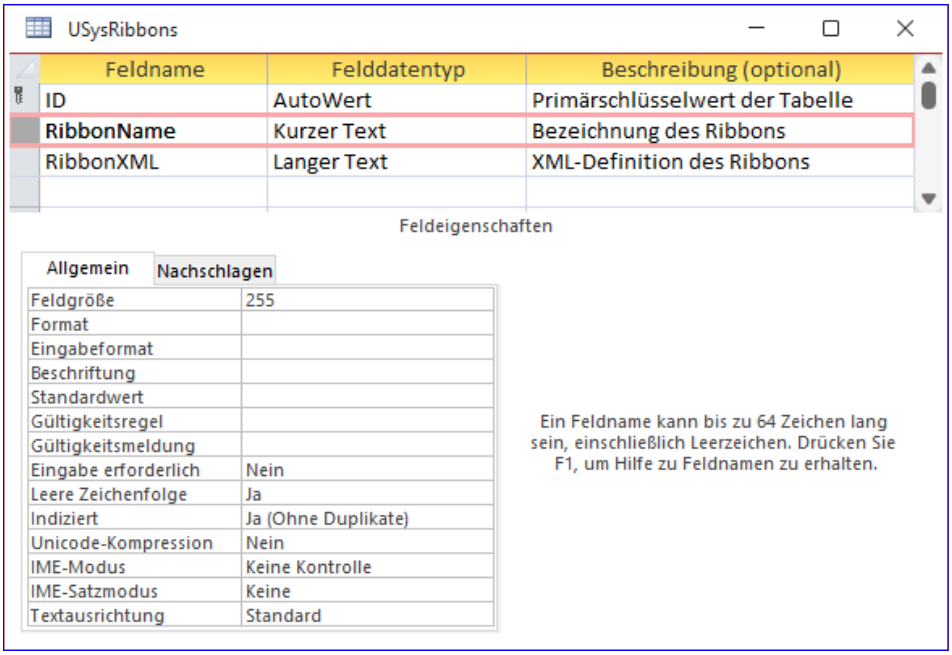


Bild 1: Tabelle zum Speichern der Ribbondefinitionen

Vorbereitung für den Einsatz von Ribbons

Bevor wir uns die Eigenarten der verschiedenen Ribbondefinitionen ansehen, schaffen wir die Voraussetzungen für die Anzeige von Ribbons. Dazu benötigen wir als Erstes eine Tabelle namens **USysRibbons**, welche die Ribbondefinitionen samt Bezeichnung speichert. Diese enthält die Felder **ID**, **RibbonName** (mit eindeutigem Index, damit jede Bezeichnung nur einmal vergeben wird) und **RibbonXML** und sieht im Entwurf wie in Bild 1 aus. Nach dem Speichern verschwindet diese Tabelle direkt, da die Bezeichnung **USys...** wie **MSys...** Systemobjekten vorbe-

halten ist, die nicht standardmäßig im Navigationsbereich angezeigt werden.

Außerdem benötigen wir ein Modul namens **mdlRibbons**, in das wir den Code der durch das Ribbon ausgelösten Ereignisse schreiben sowie ein Modul namens **mdlRibbonImages**. Dieses können Sie aus der Beispieldatenbank herauskopieren – sie ist bereits mit einigen Routinen gefüllt, welche die Anzeige von Bildern im Ribbon ermöglichen. Um Fehler in den Ribbondefinitionen zu erkennen, aktivieren wir die Option **Fehler von Benutzeroberflächen-Add-Ins anzeigen** in den Access-Optionen unter **Clienteinstellungen**.

Definition des Anwendungsribbons, das die Formulare öffnet

Als Erstes erstellen wir eine Ribbondefinition, die beim Öffnen der Anwendung erscheint und Schaltflächen enthält, mit denen wir die Formulare der nachfolgend beschriebenen Beispiele öffnen können. Diesen weisen wir dann jeweils eine Ribbondefinition zu, die mit dem Öffnen des Formulars angewendet wird.

Diese Definition sieht wie in Listing 1 aus und damit sie für die Anwendung verfügbar ist, fügen wir einen neuen Datensatz zur Tabelle **USysRibbons** hinzu, der im Feld **RibbonName** den Wert **Main** enthält und im Feld **RibbonXML** das XML-Dokument.

Im Ribbon-Dokument definieren wir ein **Ribbon**-Element, das durch den Wert **True** für das Attribut **startFromScratch** die eingebauten Elemente ausblendet – so ist es übersichtlicher. Außerdem enthält es fünf **button**-Elemente, die alle jeweils mit dem Attribut **onAction** ausgestattet sind.

Ribbon als Anwendungsribbon festlegen

Damit die Anwendung dieses Ribbon beim Öffnen anzeigt, müssen wir dieses als Anwendungsribbon festlegen. Damit dies überhaupt möglich ist, muss Access dieses

```
<?xml version="1.0"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" loadImage="loadImage">
  <ribbon startFromScratch="true">
    <tabs>
      <tab id="tabAnwendung" label="Anwendung">
        <group id="grpFormulare" label="Formulare">
          <button image="table_selection_row" label="Kundendetails" id="btnKundendetails" onAction="onAction"
            size="large"/>
          <button image="form" label="Kundendetails StartFromScratch"
            id="btnKundendetailsStartFromScratch" onAction="onAction" size="large"/>
          <button image="table_selection_all" label="Kundendetails Contextual" id="btnKundendetailsContextual"
            onAction="onAction" size="large"/>
          <button image="tables" label="Kundendetails mit Fokus" id="btnKundendetailsFokus" onAction="onAction"
            size="large"/>
          <button image="table_selection_cell" label="Kundendetails Contextual Fokus"
            id="btnKundendetailsContextualFocus" onAction="onAction" size="large"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Listing 1: Definition des Anwendungsribbons

Ribbon erst einmal als vorhanden erkennen. Das einfache Hinzufügen eines Datensatzes zur Tabelle **USysRibbons** reicht dazu nicht aus, da die Inhalte nur beim Öffnen der Access-Datenbank eingelesen werden. Daher schließen wir die Datenbankanwendung und öffnen diese erneut. Anschließend können wir die Ribbondefinition in den Access-Optionen als Anwendungsribbon einstellen. Dazu wählen wir dort im Bereich **Aktuelle Datenbank** unter **Menüband- und Symbolleistenoptionen** für die Eigenschaft **Name des Menübands** den Wert **Main** aus (siehe Bild 2).

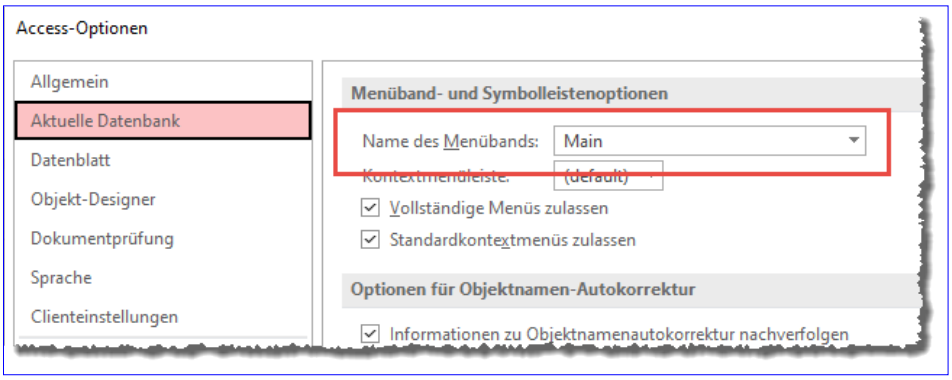


Bild 2: Einstellen des Anwendungsribbons

Damit die Anwendung dieses Ribbon anzeigt, müssen wir sie nun nochmals schließen und erneut öffnen. Erst dann prüft Access beim Öffnen, ob die Eigenschaft **Name des Menübands** einen Wert enthält und wendet dann die in der Tabelle **USysRibbons** gespeicherte Ribbondefinition an. Das Ergebnis sieht dann schließlich wie in Bild 3 aus.

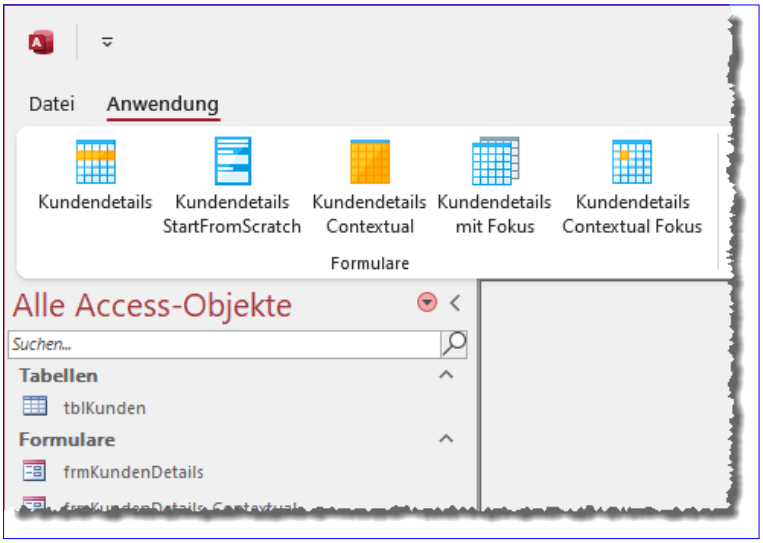


Bild 3: Das von uns definierte Anwendungsribbon

Die onAction-Prozedur

Beim Anklicken einer der Ribbon-Schaltflächen, die mit dem **onAction**-Attribut versehen sind, rufen diese die Prozedur **onAction** auf, die wir im Modul **mdlRibbons** wie folgt hinterlegt haben:

```
Sub onAction(control As IRibbonControl)
    Select Case control.ID
        Case "btnKundenDetails"
            DoCmd.OpenForm "frmKundenDetails"
        Case "btnKundenDetailsStartFromScratch"
            DoCmd.OpenForm _
                "frmKundenDetails_StartFromScratch"
        Case "btnKundendetailsContextual"
            DoCmd.OpenForm "frmKundendetails_Contextual"
        Case "btnKundendetailsFokus"
            DoCmd.OpenForm "frmKundendetails_Fokus"
        Case "btnKundendetailsContextualFocus"
            DoCmd.OpenForm _
                "frmKundendetails_Contextual_Fokus"
        Case Else
            Debug.Print control.ID
    End Select
End Sub
```

Die Prozedur prüft jeweils, welches **button**-Element diese aufgerufen hat und öffnet im jeweiligen **Case**-Zweig der **Select Case**-Anweisung das entsprechende Formular.

Dynamische Bereichshöhe im Endlosformular

Ein Leser stellte mir neulich die Frage, ob und wie man die Höhe der einzelnen Bereiche im Endlosformular dynamisch so einstellen kann, dass beispielsweise immer drei Datensätze angezeigt werden – auch, wenn der Benutzer die Höhe des Endlosformulars während der Anzeige ändert. Dieser Beitrag zeigt, wie das möglich ist. Dabei behelfen wir uns mit einer Ereignisprozedur, die immer beim Ändern der Größe eines Formulars ausgelöst wird – und eines kleinen Tricks, der wegen der enthaltenen Steuerelemente nötig wurde.

Aufgabenstellung

Zum besseren Verständnis hilft der Screenshot aus Bild 1. Wenn der Benutzer die Höhe des Formulars ändert, dann sollen die drei angezeigten Detailbereiche ihre Höhe automatisch so ändern, dass sie immer jeweils 1/3 des verfügbaren Platzes einnehmen.

1/3 deshalb, weil wir uns für das Beispiel in diesem Beitrag dazu entschieden haben, dass das Endlosformular immer drei Datensätze anzeigen soll (außer natürlich, es enthält weniger als drei Datensätze oder der Benutzer scrollt ganz nach unten).

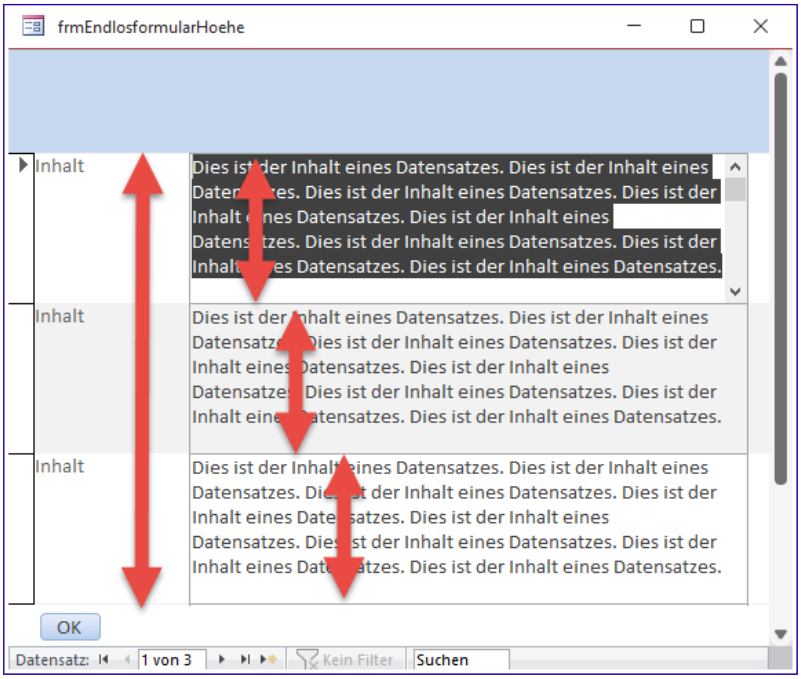


Bild 1: So soll die Höhe der Detailbereiche angepasst werden, wenn der Benutzer die Höhe des Formulars ändert.

Beispielformular

Als Beispiel verwenden wir das Formular **frmEndlosformularHoehe**. Es verwendet die Tabelle **tblInhalte** als Datensatzquelle. Diese Tabelle enthält nur die beiden Felder **ID** und **Inhalt** (Felddatentyp **Langer Text**).

Wir ziehen nun das Feld **Inhalt** aus der Feldliste in den Detailbereich und passen Höhe von Detailbereich und Textfeld einander an. Außerdem stellen wir die Eigenschaft **Standardansicht** auf **Endlosformular** ein und aktivieren die in Endlosformularen häufig sichtbaren Bereiche **Formularkopf** und **Formularfuß** (siehe Bild 2).

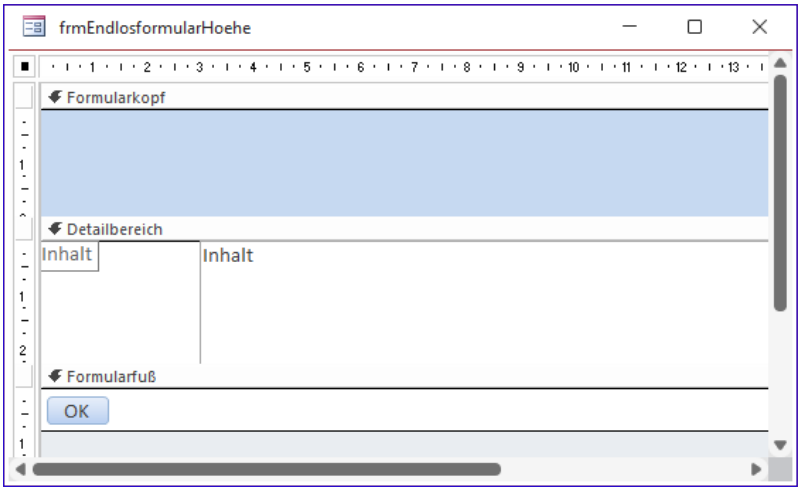


Bild 2: Entwurf des Formulars

Umsetzung mit dem Ereignis »Bei Größenänderung«

Wer schon ein wenig mit Formularen programmiert hat, der weiß: Wir werden vermutlich mit dem Ereignis **Bei Größenänderung** arbeiten, um das gewünschte Ergebnis zu erreichen.

Und wir benötigen einige Kenntnisse über die Eigenschaften, mit denen wir die Höhen verschiedener Bereiche des Formulars und des Formulars selbst ermitteln können.

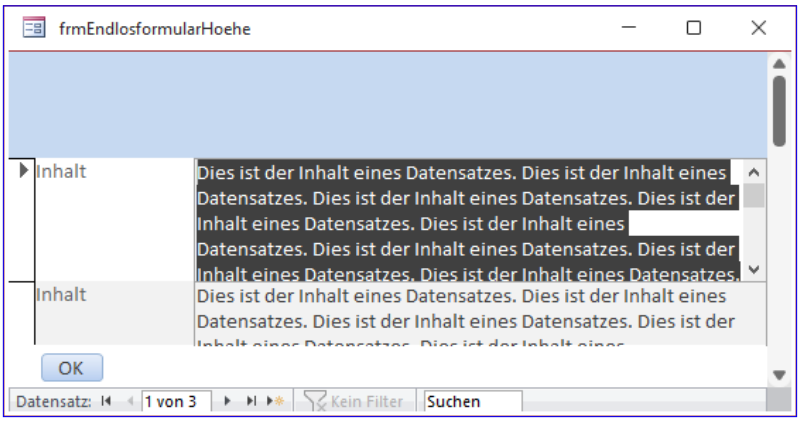


Bild 3: Die Detailbereiche werden nur bis zur Entwurfshöhe verkleinert.

Höhe der verschiedenen Bereiche

Man mag vermuten, dass man die Höhe eines Formulars mit einer Eigenschaft wie **Me.Height** ermitteln kann. Weit gefehlt: Diese Eigenschaft bietet das Formular gar nicht an. Die **Height**-Eigenschaft finden wir allerdings für die verschiedenen Bereiche des Formulars vor – für den Detailbereich, den Formularkopf und den Formularfuß beispielsweise. Wenn wir allerdings ermitteln wollen, wie hoch der Detailbereich sein darf, damit genau drei Datensätze im Bereich zwischen Formularkopf und Formularfuß sichtbar sind, benötigen wir eine Information über die gesamte Höhe des Formulars.

Ein genauerer Blick in das Objektmodell liefert schnell die Eigenschaft **InsideHeight**. **InsideHeight** liefert genau die Höhe des sichtbaren Bereichs innerhalb des Formularrahmens. Damit können wir die Höhe des Detailbereichs für drei Datensätze nach der Formel **(Gesamthöhe – Formularkopf – Formularfuß) / 3** berechnen.

In einem ersten, naiven Ansatz könnten wir also die folgende Anweisung in die Ereignisprozedur schreiben, die durch das Ereignis **Bei Größenänderung** ausgelöst wird:

```
Private Sub Form_Resize()
    Me.Detailbereich.Height = (Me.InsideHeight - _
        Me.Formularkopf.Height - Me.Formularfuß.Height) / 3
End Sub
```

Damit kommen wir schon erstaunlich weit. Gleich beim Öffnen zeigt das Formular drei Detailbereiche an, die genau den Platz zwischen Formularkopf und Formularfuß einnehmen. Wenn wir die Höhe des Formulars vergrößern, passt sich die Höhe der Detailbereiche automatisch an. Wenn wir die Höhe des Formulars allerdings so weit verkleinern, dass die im Entwurf eingestellte Höhe der Detailbereiche unterschritten wird, verkleinert die Prozedur die Höhe der Detailbereiche nicht mehr (siehe Bild 3).

Warum ist das der Fall? Der Grund ist schnell gefunden: Die enthaltenen Textfelder ändern ihre Höhe nicht analog zu der Höhe der Detailbereiche. Auch die Einstellung der Eigenschaft **Horizontaler Anker** auf den Wert **Beide** bewirkt keine Änderung. Die erste alternative Idee hierzu ist, die Höhe des Textfeldes **Inhalt** einfach auf die gleiche Höhe einzustellen wie den Detailbereich:

```
Private Sub Form_Resize()
    Me.Detailbereich.Height = (Me.InsideHeight - _
        Me.Formularkopf.Height - Me.Formularfuß.Height) / 3
    Me!Inhalt.Height = Me.Detailbereich.Height
End Sub
```

Das klappt auch, solange man die Höhe des Formulars nur vergrößert. Sobald wir die Höhe verkleinern, tut sich nichts mehr – weder die Höhe des Detailbereichs noch die des Textfeldes ändert sich. Auch dafür gibt es einen guten Grund. Wir versuchen hier, zuerst die Höhe des Detailbe-

Rechnungsverwaltung: Kundenübersicht mit Suche

Wenn ein Kunde anruft, möchten Sie schnell den entsprechenden Kundendatensatz auf dem Bildschirm haben. Dazu stellen wir im vorliegenden Beitrag ein Formular samt Unterformular zusammen, mit denen die gewünschten Daten schnell ermittelt werden können. Im Hauptformular bieten wir einige Suchfunktionen an, im Unterformular liefern wir die den Suchkriterien entsprechenden Daten in der Datenblattansicht. Außerdem soll das Formular die Möglichkeit bieten, den gefundenen Kundendatensatz im Detailformular zu öffnen, damit wir auch noch die Bestellungen des Kunden einsehen können.

Unterformular für die Kundenliste

Bevor wir beginnen, das Hauptformular zu programmieren, legen wir zunächst das Unterformular an. Dann können wir dieses gleich im Anschluss direkt zum Hauptformular hinzufügen.

Das neue Unterformular soll **sfmKundenuebersicht** heißen und standardmäßig die Daten der Tabelle **tblKunden** anzeigen. Deshalb stellen wir seine Eigenschaft **Datensatzquelle** auf diese Tabelle ein. Anschließend können wir alle Felder der Feldliste in den Detailbereich des Formulars ziehen. Gegebenenfalls lassen wir das Primärschlüsselfeld **ID** weg, da dieses nur zum Herstellen von Beziehungen dient und keine geschäftliche Funktion hat.

Damit das Formular die Daten in der Datenblattansicht anzeigt, legen wir für die Eigenschaft **Standardansicht** noch den Wert **Datenblatt** fest (siehe Bild 1). Damit können wir die Arbeiten am Unterformular bereits beenden und dieses schließen.

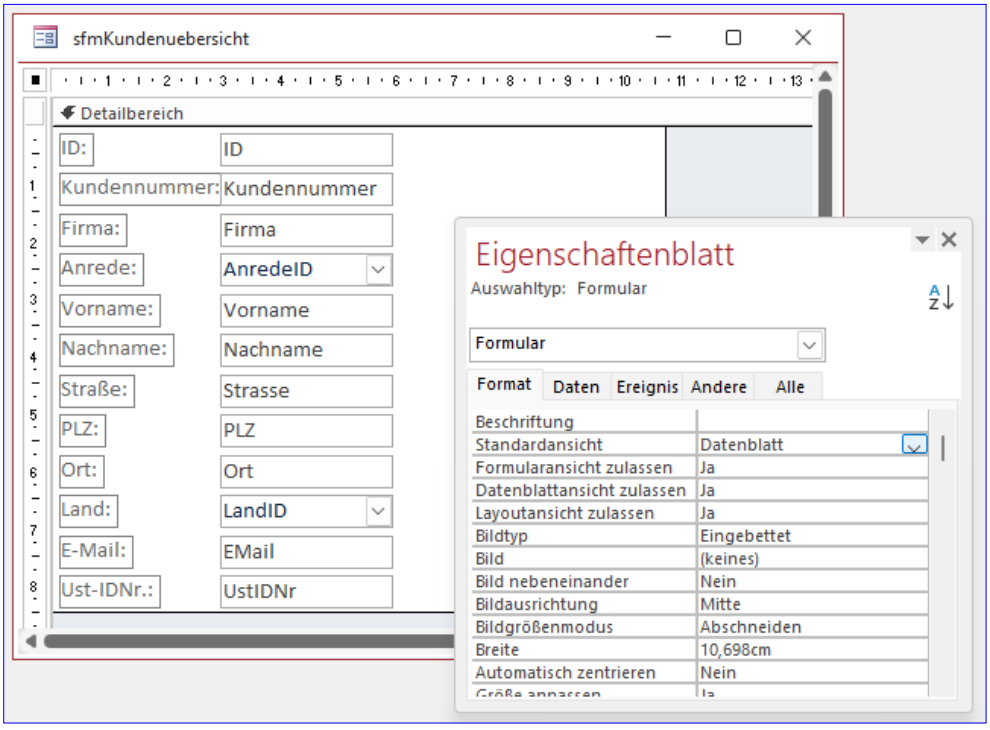


Bild 1: Entwurf des Unterformulars **sfmKundenuebersicht**

Hauptformular erstellen

Anschließend erstellen wir das Hauptformular und speichern dieses direkt unter dem Namen **frmKundenuebersicht**. Diesem weisen wir keine Datensatzquelle zu, weil es selbst keine Daten anzeigen soll – diese liefert allein das Unterformular. Im Hauptformular wollen wir nur Steuerelemente zum Durchsuchen der Kundendaten und zum Öffnen von Detaildatensätzen bereitstellen. Daher benötigen wir im Hauptformular auch nicht die Steuerelemente zum Navigieren in Datensätzen und stellen daher

einige Eigenschaften ein. Die Eigenschaften **Navigationsschaltflächen**, **Datensatzmarkierer**, **Bildlaufleisten** und **Trennlinien** setzen wir auf den Wert **Nein**, die Eigenschaft **Automatisch zentrieren** auf **Ja**.

Dann ziehen wir aus dem Navigationsbereich das Unterformular **sfmKundenuebersicht** in den Detailbereich des Formularentwurfs von **frmKundenuebersicht**. Das Bezeichnungsfeld des Unterformular-Steuerelements entfernen wir, da wir ja wissen, dass dieses Kundendaten anzeigt.

Damit das Unterformular beim Vergrößern des Hauptformulars ebenfalls vergrößert wird, stellen wir seine Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** jeweils auf **Beide** ein.

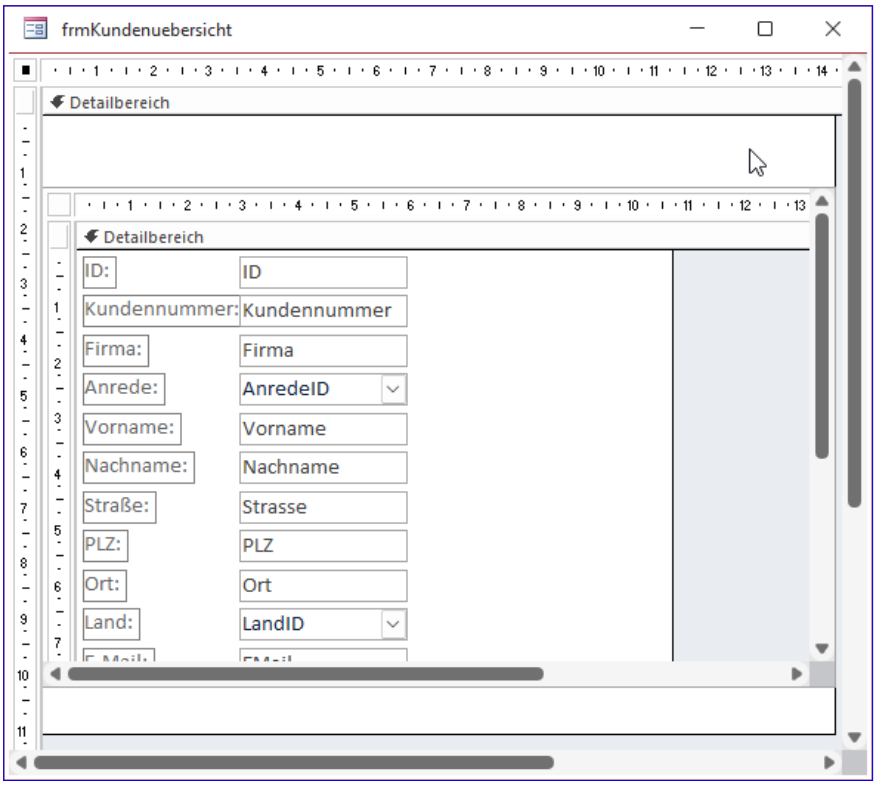


Bild 2: Haupt- und Unterformular im Entwurf

Wenn wir oben und unten ein wenig Platz für die Steuerelemente zum Suchen und zum Aufrufen der Anzeige der Kundendetails lassen, sieht der Entwurf nun wie in Bild 2 aus.

Steuerelemente zum Filtern der Kunden hinzufügen

Damit kommen wir zu den Steuerelementen, mit denen wir die Filterkriterien für die anzuzeigenden Kunden eingeben wollen. Diese platzieren wir wie in Bild 3 im Bereich über dem Unterformular. Die Steuerelemente heißen:

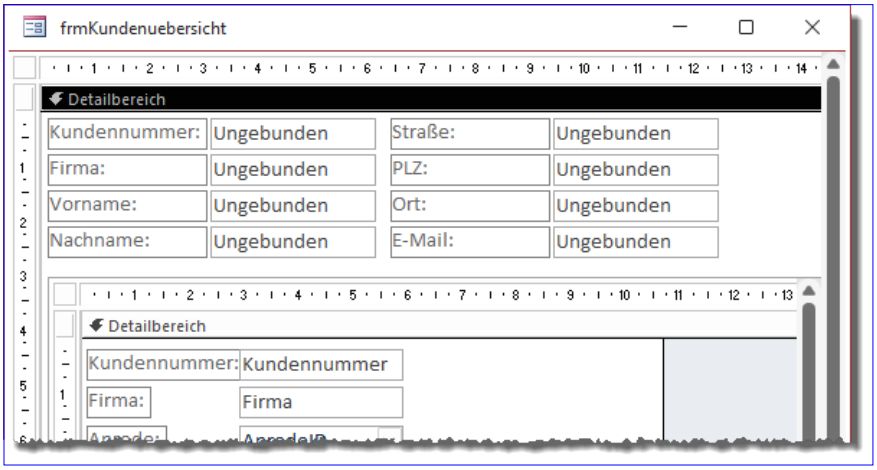


Bild 3: Hinzufügen der Steuerelemente zum Filtern der Kundendatensätze

- **txtFilterKundennummer**
- **txtFilterFirma**
- **txtFilterVorname**
- **txtFilterNachname**
- **txtFilterStrasse**
- **txtFilterPLZ**
- **txtFilterOrt**

- **txtFilterEMail**

Das Filtern wollen wir einfach und schnell realisieren: Jede Eingabe in eines der Filterfelder soll unmittelbar die gefilterten Daten anzeigen. Außerdem soll immer im kompletten zu filternden Feld nach dem im jeweiligen Filterfeld eingegebenen Text gesucht werden. Sprich: Wenn der Benutzer im Feld **txtFilterKundennummer** den Wert **1** eingibt, sollen alle Datensätze angezeigt werden, die an beliebiger Stelle im Feld **Kundennummer** den Wert **1** enthalten.

Das heißt auch, dass wir für jedes der Steuerelemente zur Eingabe der Filterkriterien das Ereignis **Bei Änderung** implementieren müssen. Mit diesem rufen wir dann jeweils eine weitere Prozedur auf, welche die jeweiligen Werte der Filter-Textfelder ausliest und ein entsprechendes SQL-Kriterium zusammenstellt.

Unterschiedliche Eigenschaften zum Ermitteln der Vergleichskriterien

Das Problem dabei ist, dass wir für die Inhalte der Filter-Textfelder auf unterschiedliche Eigenschaften zugreifen müssen. Normalerweise greifen wir einfach auf die **Value**-Eigenschaft zu. Diese können wir allerdings nicht nutzen, wenn wir während der Eingabe auf den Inhalt im aktuellen Textfeld zugreifen wollen. Der Grund ist, dass die **Value**-Eigenschaft erst mit dem aktuell im Textfeld enthaltenen Wert gefüllt wird, wenn der Benutzer die Eingabe bestätigt – beispielsweise, indem er das Textfeld verlässt und den Fokus auf das nächste Textfeld verschiebt. In diesem Fall lässt sich der aktuell im Textfeld dargestellte Text nur mit der Eigenschaft **Text** auslesen.

Wir müssen also zumindest für das aktuelle Textfeld auf die **Text**-Eigenschaft zugreifen und können nicht die **Value**-Eigenschaft nutzen. Aber warum ist das überhaupt ein Problem? Können wir nicht einfach auf den Inhalt aller Felder über die **Text**-Eigenschaft zugreifen? Das wiederum gelingt nicht, weil man nur für das Textfeld auf die **Text**-Eigenschaft zugreifen kann, das aktuell den Fokus hat.

Wir müssen also den Inhalt des aktuell bearbeiteten Textfeldes mit der **Text**- und den der übrigen Textfelder mit der **Value**-Eigenschaft abfragen und prüfen, ob diese Kriteriumsausdrücke enthalten. Wobei die **Value**-Eigenschaft wiederum die Standardeigenschaft der **TextBox**-Klasse ist, weshalb wir diese gar nicht angeben müssen. Statt **Me!txtFilterKundennummer.Value** können wir einfach **Me!txtFilterKundennummer** schreiben.

Prozedur zum Ermitteln des Filterkriteriums für das Unterformular

Wir legen also zunächst für jedes der Filter-Textfelder eine Ereignisprozedur für das Ereignis **Bei Änderung** an. Diese enthält für alle Filter-Textfelder den Aufruf einer weiteren Prozedur namens **KundenFiltern** – wie hier am Beispiel der Prozedur **txtFilterEMail** zu sehen:

```
Private Sub txtFilterEMail_Change()
    KundenFiltern
End Sub
```

Einfache Variante zum Zusammensetzen des Filterkriterums

Die Prozedur zum Zusammenstellen eines Filterausdrucks für das Unterformular können wir geradlinig und einfach aufbauen oder auch auf den ersten Blick etwas komplizierter, dafür aber flexibler. Wir schauen uns beide Varianten an. Die erste finden Sie in Listing 1.

Diese Version verwendet eine Variable namens **strFilter**, um den Filterausdruck darin zusammenzustellen. Sie prüft für jedes der Filter-Textfelder, ob es sich dabei um das aktuelle Steuerelement handelt, also das Steuerelement, das aktuell den Fokus enthält. Das dient der Unterscheidung, ob wir die **Text**- oder die **Value**-Eigenschaft zum Ermitteln des jeweils in dem Steuerelement enthaltenen Textes verwenden müssen.

Um herauszufinden, ob wir es bei dem aktuellen Steuerelement mit dem aktiven Steuerelement zu tun haben, vergleichen wir dieses mit dem Verweis auf das Steuer-

```
Private Sub KundenFiltern()
    Dim strFilter As String
    If Me!txtFilterKundennummer Is Me.ActiveControl Then
        If Not Len(Me!txtFilterKundennummer.Text) = 0 Then
            strFilter = strFilter & " AND Kundennummer LIKE '*" & Me!txtFilterKundennummer.Text & "*'"
        End If
    Else
        If Not Len(Me!txtFilterKundennummer.Value) = 0 Then
            strFilter = strFilter & " AND Kundennummer LIKE '*" & Me!txtFilterKundennummer.Value & "*'"
        End If
    End If
    If Me!txtFilterFirma Is Me.ActiveControl Then
        If Not Len(Me!txtFilterFirma.Text) = 0 Then
            strFilter = strFilter & " AND Firma LIKE '*" & Me!txtFilterFirma.Text & "*'"
        End If
    Else
        If Not Len(Me!txtFilterFirma.Value) = 0 Then
            strFilter = strFilter & " AND txtFilterFirma LIKE '*" & Me!txtFilterFirma.Value & "*'"
        End If
    End If
    '...
    If Not Len(strFilter) = 0 Then
        strFilter = Mid(strFilter, 5)
        With Me!sfmKundenuebersicht.Form
            .Filter = strFilter
            .FilterOn = True
        End With
    Else
        Me!sfmKundenuebersicht.Form.Filter = ""
    End If
End Sub
```

Listing 1: Prozedur zum Einstellen des Filters für das Unterformular, einfache Version

element, das wir mit der Eigenschaft **ActiveControl** des Formulars ermitteln können. Sind beide gleich, handelt es sich um das aktive Steuerelement, sonst nicht. Ist das Steuerelement das aktive Steuerelement und ist dieses nicht leer, lesen wir den Inhalt mit der **Text**-Eigenschaft und stellen damit einen Ausdruck wie den folgenden zusammen:

```
AND Kundennummer LIKE '*Vergleichsausdruck*'
```

Vergleichsausdruck entspricht dabei dem Inhalt des jeweiligen Textfeldes. Falls es sich bei dem aktuell untersuchten Steuerelement nicht um das aktive Steuerelement handelt, verwenden wir für den gleichen Ausdruck den Inhalt der **Value**-Eigenschaft des Textfeldes.

Das Problem ist, dass wir in dieser Version für jedes Textfeld die folgenden Zeilen, hier beispielhaft am Textfeld **txtFilterKundennummer** gezeigt, in der Prozedur anlegen müssen:

```
If Me!txtFilterKundennummer Is Me.ActiveControl Then
    If Not Len(Me!txtFilterKundennummer.Text) = 0 Then
        strFilter = strFilter & " AND Kundennummer
```

Kunden nach bestellten Produkten filtern

Kunden nach bestellten Produkten kann jeder filtern, der sich ein wenig mit dem Abfrageentwurf beschäftigt hat. Etwas aufwendiger ist es schon, ein Formular zu erstellen, das verschiedene Möglichkeiten zum Filtern von Kunden nach den bestellten Produkten bietet. Hier wollen wir beispielsweise ein Produkt auswählen, sodass direkt alle Kunden in einer Liste angezeigt werden, die dieses Produkt bestellt haben. Oder wir gehen noch einen Schritt weiter und wollen Kunden anzeigen, die mindestens eines von mehreren Produkten geordert haben. Um dann vielleicht noch solche Kunden auszuschließen, die bereits ein bestimmtes anderes Produkt besitzen. Also auf ins Abenteuer!

Warum Kunden nach Produkten filtern?

Bevor wir uns an die Arbeit dieses recht aufwendigen Unterfangens machen, wollen wir uns überlegen, wozu wir das Ergebnis überhaupt nutzen können. Mir als Shopbetreiber fällt da direkt ein, Kunden, die ein bestimmtes Produkt erworben haben, über eine neue Version dieses Produkts zu informieren. Oder man möchte dem Kunden mitteilen, dass eine Lizenz ausläuft, damit er diese verlängern kann. Vielleicht wollen wir auch einfach Kunden, die Produkt A bestellt haben, eine Empfehlung zu dem dazu passenden Produkt B geben. Dabei wollen wir dann natürlich nur Kunden anschreiben, die Produkt B noch nicht bestellt haben. Sie sehen: Es lohnt sich, die Kunden nach den bestellten Produkten selektieren zu können.

Voraussetzungen

Als Basis für die Ermittlung von Kunden nach den bestellten Produkten verwenden wir die Beispieldatenbank, die wir bereits in der Beitragsreihe zur Rechnungsverwaltung vorgestellt haben. Den ersten Teil dieser Beitragsreihe finden Sie übrigens unter dem Titel **Rechnungsverwaltung: Datenmodell** (**www.access-im-unternehmen.de/1385**).

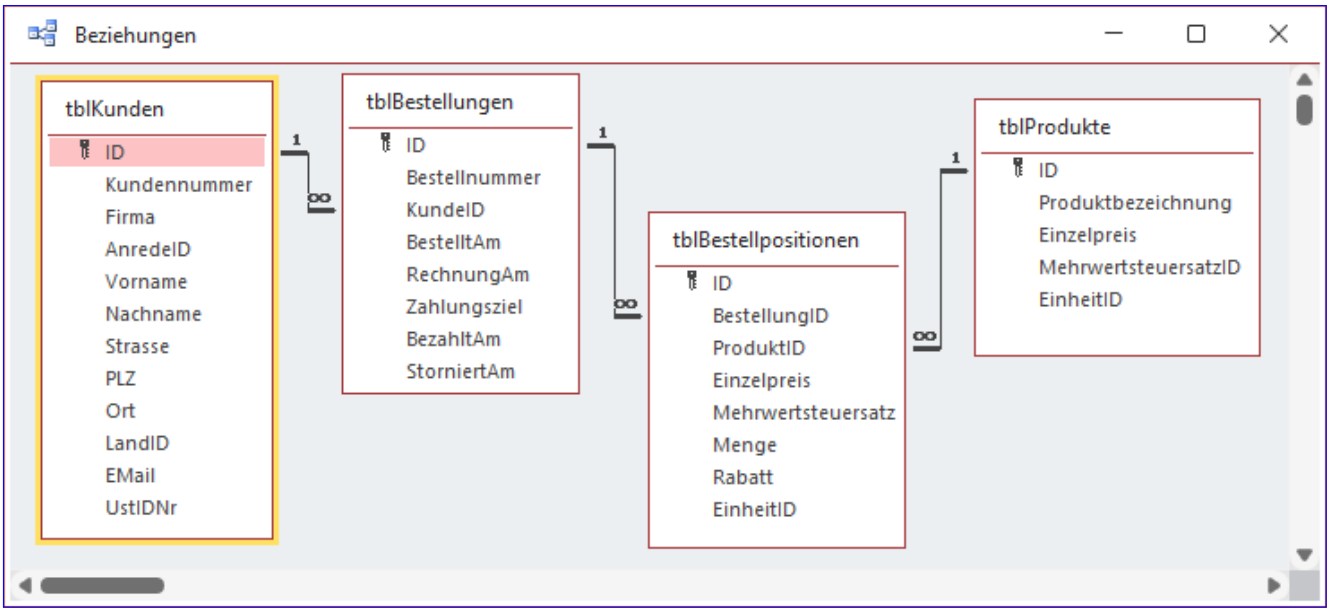


Bild 1: Tabellen, die am Filtern von Kunden nach Produkten beteiligt sind

Der relevante Teil des Datenmodells dieser Datenbank ist in Bild 1 abgebildet.

Geplante Funktionen für die Lösung

Wir wollen mit dem Formular zum Filtern von Kunden nach bestellten Produkten möglichst flexibel arbeiten können.

Also schauen wir uns zunächst an, welche Filtermöglichkeiten wir abbilden wollen:

- Filtern nach Kunden, die ein bestimmtes Produkt bestellt haben
- Filtern nach Kunden, die eines von mehreren Produkten bestellt haben (Oder-Verknüpfung)
- Filtern nach Kunden, die ein bestimmtes Produkt nicht bestellt haben
- Filtern nach Kunden, die bestimmte Produkte nicht bestellt haben (Und-Verknüpfung)

Die ersten beiden Filtermöglichkeiten sollen außerdem per Und-Verknüpfung mit den letzten beiden Möglichkeiten kombiniert werden.

Einfache Auswahl der Produkte, nach denen gefiltert werden soll

Wir wollen zwei Listenfelder im Formular anzeigen, die unterschiedliche Produkte enthalten:

- Das erste Listenfeld soll die Produkte anzeigen, von denen der Kunde mindestens eines bestellt hat.
- Das zweite Listenfeld soll die Produkte anzeigen, die der Kunde nicht bestellt hat.

Um die beiden Listenfelder zu füllen, bieten wir ein drittes Listenfeld an, das alle Produkte enthält. Aus diesem wählen wir dann die Produkte aus, die den beiden zuvor genannten Listenfeldern hinzugefügt werden sollen.

Um aus diesem Listenfeld komfortabel die gewünschten Produkte auswählen zu können, wollen wir über diesem Listenfeld noch ein Textfeld einfügen, mit dem wir die angezeigten Produkte nach dem eingegebenen Text filtern können.

Unter den Listenfeldern zur Auswahl der zu filternden Produkte fügen wir schließlich noch ein Unterformular ein, das die Kunden anzeigt, welche die den Listenfeldern hinzugefügten Produkte entweder bestellt haben – oder auch nicht.

Unterformular zur Anzeige der Kunden

Als Erstes legen wir das Unterformular **sfmKundenNachProduktenFiltern** an. Dieses soll die Kunden anzeigen, die den im Hauptformular festgelegten Kriterien entsprechen. Beim Öffnen des Formulars jedoch soll es zunächst alle Kunden anzeigen. Deshalb stellen wir die Eigenschaft **Datensatzquelle** auf eine Abfrage ein, welche die Tabelle **tblKunden** als Datenherkunft verwendet und davon die Felder **KundeID**, **AnredeID**, **Nachname**, **Vorname** und **EMail** anzeigt, sortiert nach den Feldern **Nachname** und **Vorname** (siehe Bild 2).

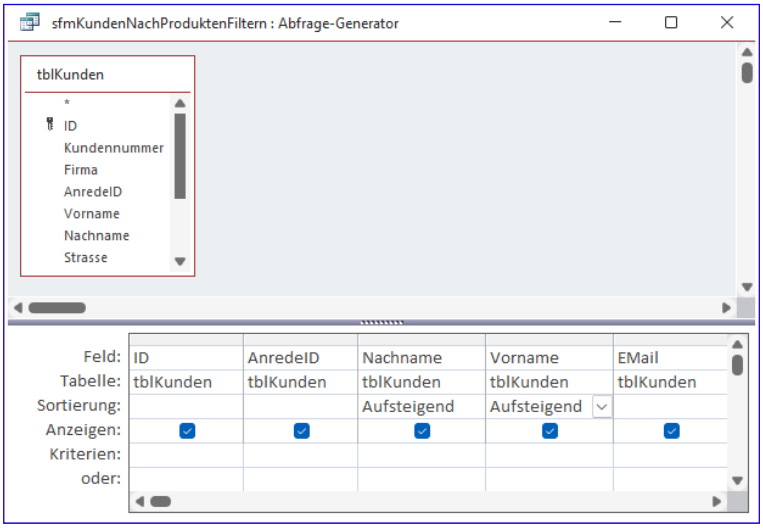


Bild 2: Abfrage für das Unterformular

Anschließend ziehen wir alle Felder dieser Abfrage aus der Feldliste in den Entwurf des Formulars (siehe Bild 3).

Damit dieses Formular seine Daten in der Datenblattansicht anzeigt, stellen wir seine Eigenschaft **Standardansicht** auf den Wert **Datenblatt** ein. Nun schließen und speichern wir das Unterformular.

Hauptformular zum Filtern der Kunden

Dann legen wir ein weiteres neues Formular namens **frmKundenNachProduktenFiltern** an und fügen diesem folgende Steuerelemente hinzu:

- Textfeld **txtProduktfilter**
- Listenfeld **lstAlleProdukte**
- Listenfeld **lstBestellteProdukte**
- Listenfeld **lstNichtBestellteProdukte**

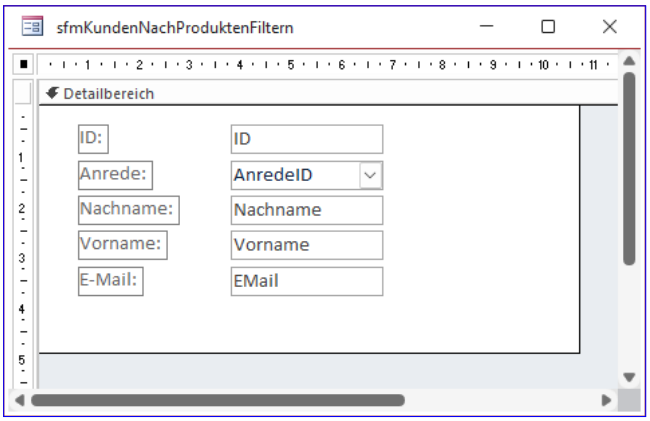


Bild 3: Entwurf des Unterformulars

- Schaltfläche **cmdZuBestelltenProdukten**
- Schaltfläche **cmdZuNichtBestelltenProdukten**

Außerdem ziehen wir aus dem Navigationsbereich das Formular **sfmKundenNachProduktenFiltern** in das Hauptformular. Die Steuerelemente ordnen wir dabei wie in Bild 4 an.

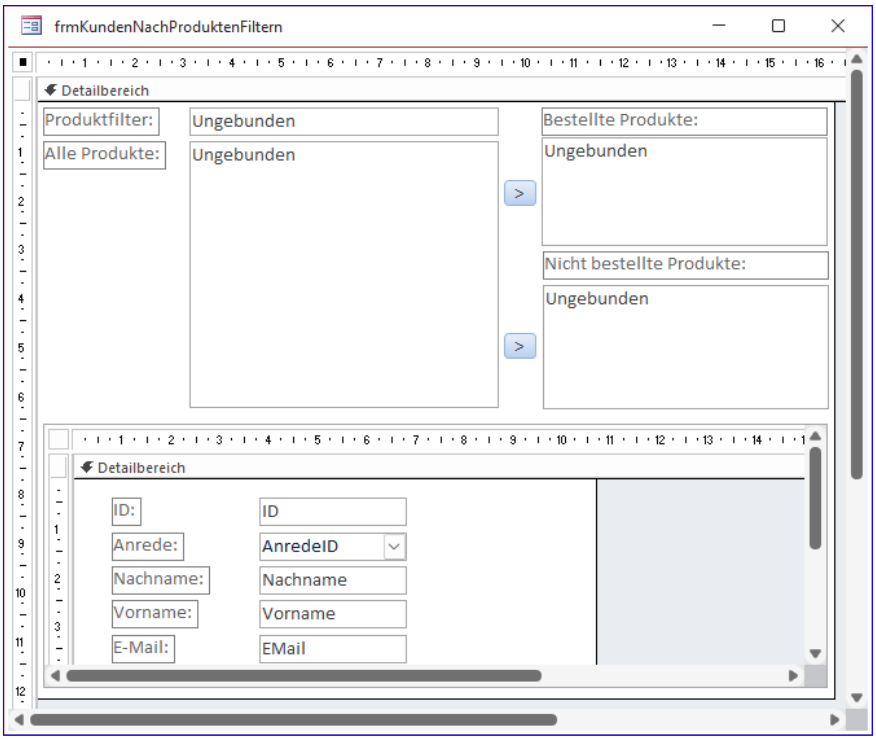


Bild 4: Anordnung der Steuerelemente im Hauptformular

Listenfelder mit Daten füllen

Das Listenfeld **lstAlleProdukte** soll alle Produkte anzeigen mit Ausnahme derer, die bereits zu einem der Listenfelder **lstBestellteProdukte** oder **lstNichtBestellteProdukte** hinzugefügt wurden.

Das Listenfeld **lstBestellteProdukte** soll alle Produkte anzeigen, die nach Markierung im Listenfeld **lstAlleProdukte** mit der Schaltfläche **cmdZuBestelltenProdukten** hinzugefügt wurden.

Das Listenfeld **lstNichtBestellteProdukte** soll analog alle Produkte anzeigen, die nach Markierung im Listenfeld **lstAlleProdukte** mit der Schaltfläche **cmdZuNichtBestelltenProdukten** hinzugefügt wurden.

Nun müssen wir allerdings noch festlegen, wie wir die Produkte markieren, die in einem der beiden Listenfelder **lstBestellteProdukte** oder **lstNichtBestellteProdukte** angezeigt werden.

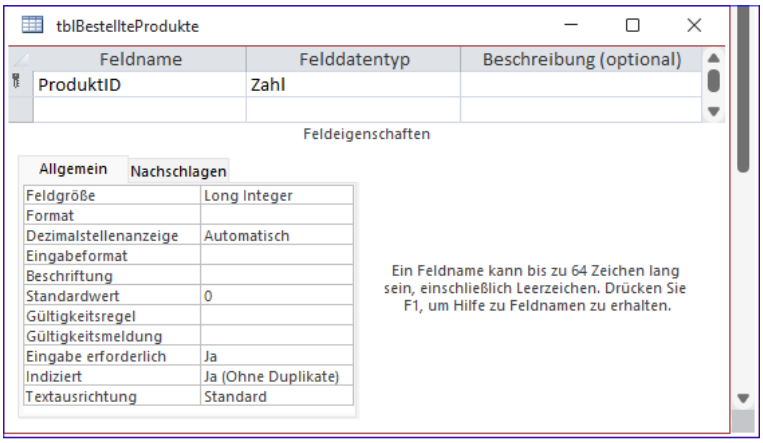


Bild 5: Entwurf der Tabelle **tblBestellteProdukte**

Tabellen zum Speichern der ein- und auszuschließenden Produkte

Wir könnten die beiden Listenfelder einfach mit dem Wert **Wertliste** für die Eigenschaft **Herkunftsart** ausstatten und die Produkte als String-Liste der Eigenschaft **Datensatzherkunft** hinzufügen. Wenn wir die Produkte in diesen beiden Listen allerdings jederzeit nach dem Produktnamen sortiert anzeigen wollen, haben wir mehr Programmieraufwand. Und da wir in Access arbeiten, legen wir schnell zwei Tabellen namens **tblBestellteProdukte** und **tblNichtBestellteProdukte** an. Die erste sieht in der Entwurfsansicht wie in Bild 5 aus und enthält lediglich ein Zahlenfeld zum Speichern der **ProduktID**-Werte, nach denen die Kunden gefiltert werden sollen. Dieses Feld haben wir als Primärschlüsselfeld definiert, jedoch logischerweise nicht mit dem Datentyp **Autowert**. Die zweite unterscheidet sich nur durch den Namen von der ersten Tabelle. Zeitspar-Tipp: Einfach die erste Tabelle kopieren und unter dem Namen **tblNichtBestellteProdukte** speichern.

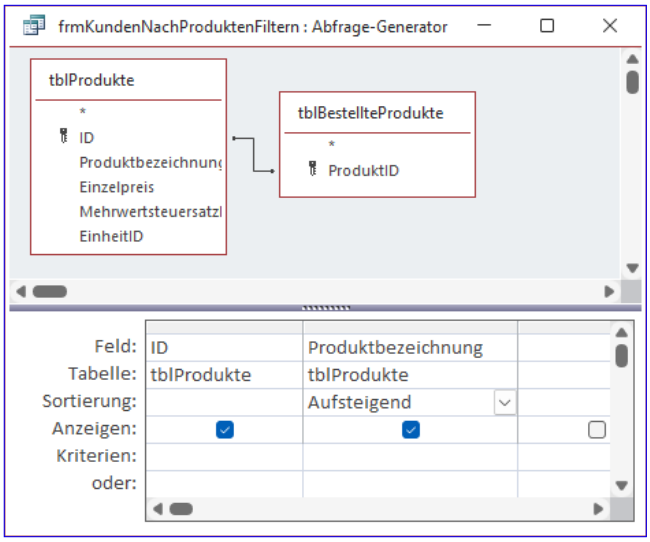


Bild 6: Datensatzherkunft des Listenfeldes **lstBestellteProdukte**

Datensatzherkunft der Listenfelder lstBestellteProdukte und lstNichtBestellteProdukte

Die Datensatzherkunft der beiden rechten Listenfelder können wir nun bereits einstellen. Die für das Listenfeld **lstBestellteProdukte** gestalten wir wie in Bild 6. Sie soll alle Datensätze der Tabelle **tblProdukte** anzeigen, deren Wert im Feld **ProduktID** in der Tabelle **tblBestellteProdukte** gespeichert ist. Nachdem Sie die beiden Tabellen zum Abfrageentwurf hinzugefügt haben, müssen Sie die Beziehung zwischen den Feldern **ID** der Tabelle **tblProdukte** und **ProduktID** der Tabelle **tblBestellteProdukte** manuell hinzufügen. Diese Beziehung wollen wir nicht im Datenbankfenster festlegen.

Die Datensatzherkunft für das Listenfeld **lstNichtBestellteProdukte** erstellen wir analog, diesmal verwenden wir jedoch die Tabelle **tblNichtBestellteProdukte** als zweite Tabelle.

Weitere Einstellungen für die Listenfelder

Für alle drei Listenfelder nehmen wir nun die **Format**-Einstellungen vor. Hier legen wir für die Eigenschaft **Spaltenanzahl** den Wert **2** und für **Spaltenbreiten** den Wert **0cm** fest. So zeigen die Listenfelder immer nur die Produktbezeichnung an, aber nicht den Wert der gebundenen Spalte **ProduktID**.

Daten des Listenfeldes lstAlleProdukte

Der Name dieses Listenfeldes ist eigentlich irreführend, denn es zeigt nur direkt nach dem Öffnen des Formulars alle Produkte an. Sobald der Benutzer über das Textfeld **txtProduktfilter** einen Filter eingegeben hat oder ein Produkt in eines der rechten Listenfelder verschoben hat, zeigt es nicht mehr alle Produkte an.

Es soll dann nur noch die Produkte anzeigen, die nach dem Filtern und Entfernen der bereits in einem der übrigen Listenfeldern enthaltenen Produkte übrig bleiben.

Zuerst aber soll dieses Listenfeld einfach alle Produkte anzeigen, die in der Tabelle **tblProdukte** enthalten sind – und zwar nach dem Alphabet sortiert. Dazu weisen wir der Eigenschaft **Datensatzherkunft** des Listenfeldes die Abfrage aus Bild 7 zu.

Die enthaltenen Daten sollen anschließend nach verschiedenen Aktionen aktualisiert werden. Bei den Aktionen handelt es sich um die folgenden:

- Filtern über das Textfeld **txtProduktfilter** nach dem Auslösen des Ereignisses **Bei Änderung**
- Hinzufügen eines Produkts zu den Listenfeldern **lstBestellteProdukte** und **lstNichtBestellteProdukte mit den Schaltflächen cmdZuBestelltenProdukten** und **cmdZuNichtBestelltenProdukten**
- Entfernen eines Produkts aus den Listenfeldern **lstBestellteProdukte** und **lstNichtBestellteProdukte** per Doppelklick auf einen der Einträge der Listenfelder

Deshalb rufen wir von all den durch die oben beschriebenen Aktionen Prozeduren auf, welche die Datensatzherkunft des Listenfeldes **lstAlleProdukte** aktualisiert (und später auch die Liste der Kunden).

Diese Prozeduren sollen **ProdukteAktualisieren** und **KundenAktualisieren** heißen.

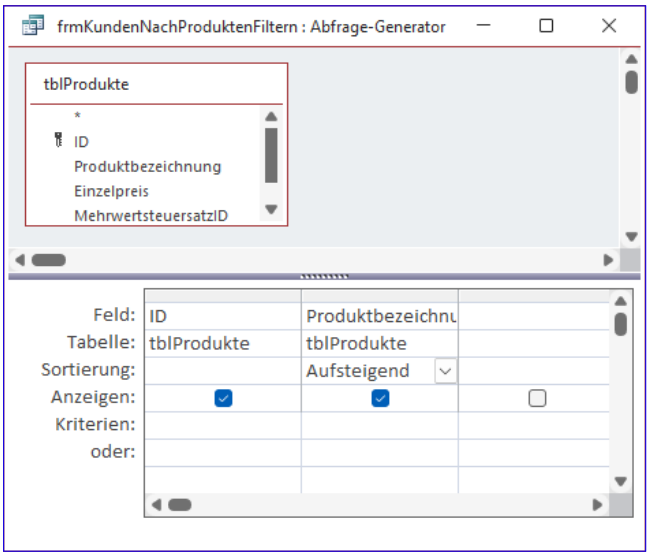


Bild 7: Datensatzherkunft des Listenfeldes **lstAlleProdukte**

Prozeduren zum Aktualisieren von Produkten und Kunden

Die Prozeduren **ProdukteAktualisieren** und **KundenAktualisieren** sollen den im Textfeld **txtProduktfilter** eingegebenen Suchbegriff berücksichtigen und zusätzlich über die in die beiden Tabellen **tblBestellteProdukte** und **tblNichtBestellteProdukte** gespeicherten Einträge die noch verfügbaren Produkte im Listenfeld **lstAlleProdukte** anzeigen. Schließlich soll noch die Anzeige der Kunden passend zu den in den beiden Tabellen gespeicherten Daten aktualisiert werden.

Da diese Prozeduren sowohl durch das Ereignis **Bei Änderung** des Textfeldes **txtProduktfilter** ausgelöst werden, kann als auch durch das Hinzufügen oder Entfernen von Einträgen zu den Tabellen **tblBestellteProdukte** und **tblNichtBestellteProdukte**, haben wir ein kleines Problem: Beim Aufruf über das Textfeld können wir auf den aktuellen Inhalt des Textfeldes nur über die Eigenschaft **Text** des Textfeldes zugreifen. Beim Aufruf über eines der anderen Ereignisse müssen wir die **Value**-Eigenschaft des Textfeldes verwenden.

Wir können das auf verschiedene Arten lösen. Eine ist, den Wert des Textfeldes jeweils in einer Variablen zu

Prüfen, ob Datenbank geöffnet ist

In einem früheren Beitrag haben wir mit einer Funktion geprüft, ob eine Datenbank geöffnet ist. Diese war jedoch nicht in jedem Fall zuverlässig – also liefern wir eine neue Version für eine solche Funktion. In dieser neuen Funktion versuchen wir, die Datenbank exklusiv zu öffnen. Das gelingt nur, wenn diese aktuell nicht geöffnet ist. Mehr dazu im vorliegenden Beitrag!

In der Funktion namens **IsDatabaseOpen**, die wir im Beitrag **Kunde zu einer E-Mail öffnen** (**www.access-im-unternehmen.de/1291**) vorgestellt haben, prüfen wir nur, ob sich im gleichen Verzeichnis der Datenbank auch eine **.laccdb**-Datei befindet. Diese wird normalerweise erstellt, wenn die Datenbank geöffnet ist.

Carsten Gromberg hat uns darauf hingewiesen, dass diese Funktion nicht immer das korrekte Ergebnis liefert und eine verbesserte Version der Funktion beigesteuert.

Dabei verwenden wir einen zuverlässigeren Weg, um zu prüfen, ob eine Datenbank geöffnet ist. Dabei versuchen wir, die Datenbank exklusiv zu öffnen. Das ist nur möglich, wenn die Datenbank bisher gar nicht geöffnet ist.

Die neue Version der Funktion **IsDatabaseOpen** finden Sie in Listing 1. Die Funktion erwartet den Pfad zu der zu untersuchenden Datenbankdatei sowie einen **Boolean**-Parameter, der angibt, ob bei bereits geöffneter Datenbank eine Meldung ausgegeben werden soll.

Die Funktion erstellt ein Objekt auf Basis der verborgenen Klasse **PrivDBEngine**, die im Gegensatz zu **DBEngine** eine neue Session außerhalb der Session der aktuellen Datenbank öffnet und somit unabhängig ist. Warum das wichtig ist, schauen wir uns weiter unten an.

Nach dem Erstellen von **objEngine** deaktiviert die Funktion zunächst die eingebaute Fehlerbehandlung. Somit kann sie die nachfolgende **OpenDatabase**-Methode des

```
Public Function IsDatabaseOpen(ByVal strDBName As String, _
                               Optional ByVal bolShowMessage As Boolean) As Boolean
    Dim objEngine   As DAO.PrivDBEngine
    Set objEngine = New DAO.PrivDBEngine
    On Error Resume Next
    objEngine.OpenDatabase strDBName, True, True
    If Not Err.Number = 0 Then
        IsDatabaseOpen = True
    End If
    If IsDatabaseOpen And bolShowMessage Then
        MsgBox strDBName & vbCrLf _
            & "kann nicht exklusiv geöffnet werden!" & vbCrLf & vbCrLf _
            & "Fehler: " & CStr(Err.Number) & vbCrLf _
            & Err.Description, vbExclamation
    End If
    Set objEngine = Nothing
End Function
```

Listing 1: Funktion zum Prüfen, ob eine Datenbank bereits geöffnet ist

Dateien per VBA öffnen

Es gibt viele Gelegenheiten, zu denen man gern eine Datei per VBA öffnen möchte. Ein gutes Beispiel ist ein soeben auf Basis eines Berichts erstelltes PDF-Dokument. Doch der Standardumfang von VBA liefert keine Möglichkeit, diese Aufgabe zu erledigen. Und tatsächlich ist das Anzeigen einer Datei nicht trivial, zumindest dann nicht, wenn wir vorher noch nicht wissen, welchen Dateityp die Datei hat und mit welcher Anwendung diese geöffnet werden soll. Allerdings weiß Windows ja auch meistens, mit welcher Anwendung eine Datei geöffnet werden soll, wenn wir diese im Windows Explorer doppelt anklicken. Also muss es einen Weg geben, diese Aufgabe per Code zu erledigen. Und die Lösung ist eine API-Funktion namens ShellExecute.

Beispieltabelle mit Dateiinformationen

Zu Beispielzwecken haben wir einer Datenbank eine Tabelle namens **tblDateien** hinzugefügt. Diese sieht wie in Bild 1 aus und enthält neben einer Bezeichnung den Namen von Dateien. Auf Pfadangaben haben wir aufgrund der besseren Nutzbarkeit der Beispieldatenbank verzichtet und gehen davon aus, dass die angegebenen Dateien sich im gleichen Verzeichnis wie die Datenbank befinden. Auf diese Weise können wir in den folgenden Beispielen per Code unter Zuhilfenahme von **CurrentProject.Path** auf die Dateien zugreifen.

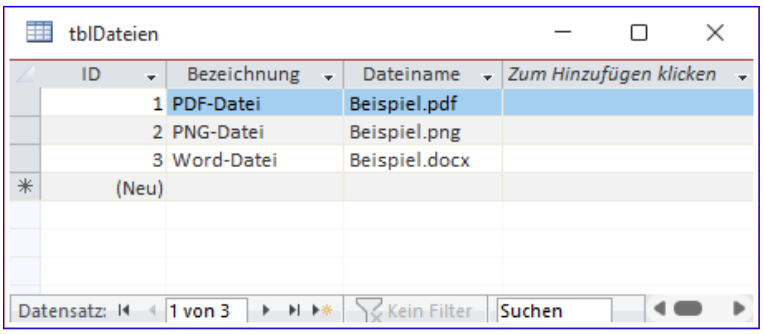


Bild 1: Tabelle mit Dateiangaben

Formular zum Anzeigen der Beispieldateien

Um die Funktion zum Anzeigen der Dateien einfach aufrufen zu können, erstellen wir ein Formular, das die Daten der Tabelle **tblDateien** anzeigt und eine Schaltfläche bereitstellt, mit der wir die Datei anzeigen können (siehe Bild 2). Diese Schaltfläche heißt **cmdDateiOeffnen**. Gleich fügen wir der Schaltfläche den Code zum Öffnen der Datei aus dem Feld **Dateiname** hinzu.

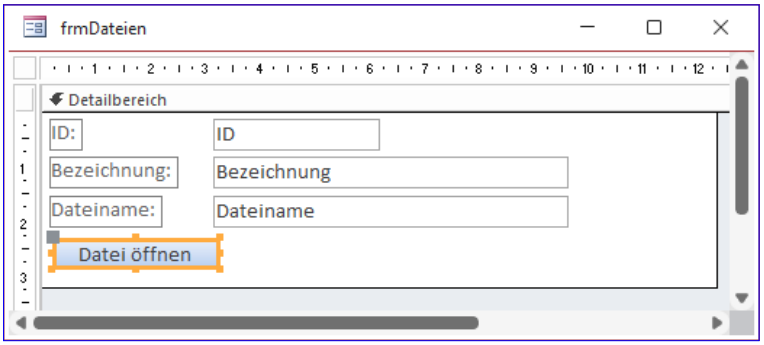


Bild 2: Entwurf eines Formulars zur Anzeige von Dateien in der jeweiligen Anwendung

API-Funktion zum Anzeigen von Dateien

Für das Anzeigen von Dateien benötigen wir die API-Funktion **ShellExecuteA**. Diese deklarieren wir wie in Listing 1 in zweifacher Ausführung, einmal für Access-Versionen, die ein älteres VBA verwenden, und einmal für die aktuellen Fassungen. Die Deklaration platzieren wir oben in einem neuen Standardmodul namens **mdlDateienAnzeigen**. Der Unterschied besteht in der Verwendung des Schlüsselworts **PtrSafe** sowie dem Datentyp **LongPtr** für den Parameter **hWnd** bei der Version für aktuelles VBA.

```
#If VBA7 Then
    Public Declare PtrSafe Function ShellExecuteA Lib "Shell32" (ByVal hWnd As LongPtr, ByVal lpOperation As String, _
      ByVal lpFile As String, ByVal lpParameters As String, ByVal lpDirectory As String, ByVal nCmdShow As Long) As Long
#Else
    Public Declare Function ShellExecuteA Lib "Shell32" (ByVal hWnd As Long, ByVal lpOperation As String, ByVal _
       lpFile As String, ByVal lpParameters As String, ByVal lpDirectory As String, ByVal nCmdShow As Long) As Long
#End If
```

Listing 1: Deklaration der API-Funktion ShellExecute für 32-Bit und 64-Bit-VBA

Die Funktion erwartet einige Parameter, die wir uns hier anschauen:

- **hWnd**: Erwartet ein Handle auf das aufrufende Fenster. Hier können wir den Wert **0** übergeben.
- **lpOperation**: Erwartet eine Zeichenkette mit der auszuführenden Operation. Für unsere Aufgabe, eine Datei anzuzeigen, verwenden wir hier den Wert **Open**.
- **lpFile**: Nimmt den Pfad zu der anzuzeigenden Datei entgegen.
- **lpParameters**: Parameter, den wir hier nicht benötigen und deshalb mit **vbNullString** füllen
- **lpDirectory**: Parameter, den wir hier nicht benötigen und deshalb mit **vbNullString** füllen
- **nCmdShow**: Erwartet eine Konstante für den Anzeigemodus.

Die möglichen Werte für **nCmdShow** lauten:

- **SW_HIDE** (**0**): Öffnet die Datei im versteckten Modus.
- **SW_MAXIMIZE** (**3**): Öffnet das Fenster mit der Datei im maximierten Zustand.
- **SW_MINIMIZE** (**6**): Zeigt das Fenster minimiert an.
- **SW_NORMAL** (**1**): Aktiviert das Fenster beim Öffnen.
- **SW_SHOW** (**5**): Einfache Anzeige.
- **SW_RESTORE** (**9**): Stellt die Fenstergröße wieder her.
- **SW_SHOWMAXIMIZED** (**3**): Zeigt das Fenster an und maximiert es.
- **SW_SHOWMINIMIZED** (**2**): Zeigt das Fenster an und minimiert es.
- **SW_SHOWMINNOACTIVE** (**7**): Minimiert das Fenster und aktiviert es nicht
- **SW_SHOWNA** (**8**): Zeigt das Fenster an, aber aktiviert es nicht.
- **SW_SHOWNOACTIVATE** (**4**): Zeigt das Fenster an, ohne es zu aktivieren.
- **SW_SHOWNORMAL** (**1**): Zeigt das Fenster und aktiviert es.

Damit wir die Konstanten statt der Zahlenwerte angeben können, deklarieren wir diese oberhalb der API-Deklaration im gleichen Modul:

```
Public Const SW_HIDE = 0
Public Const SW_MAXIMIZE = 3
Public Const SW_MINIMIZE = 6
Public Const SW_NORMAL = 1
Public Const SW_SHOW = 5
Public Const SW_RESTORE = 9
Public Const SW_SHOWMAXIMIZED = 3
```

Produktivität mit Notion steigern

Gelegentlich gönnen wir von Access im Unternehmen uns einen Ausflug zu einer anderen Anwendung. In diesem Fall geht es um Notion, einer modernen Produktivitätsapp. Eigentlich ist gar keine große Rechtfertigung notwendig, denn die meisten Leser dieses Magazins arbeiten vermutlich produktiv mit Access und sind auf eine entsprechend strukturierte Arbeitsweise angewiesen. Diese unterstützen Tools wie Notion, denn sie erlauben eine Ablage aller möglichen Informationen in strukturierter Form. Außerdem können Sie damit beispielsweise Aufgaben auflisten und abarbeiten und beliebige andere Daten damit verwalten – in entsprechend kostenpflichtigen Versionen sogar im Team. Und es gibt noch einen wesentlichen Grund, darüber in diesem Magazin zu berichten: Wir können nämlich über die API auf die in Notion abgelegten Daten zugreifen und diese auch von einer Access-Datenbank aus befüllen. Doch dies soll Thema eines anderen Beitrags sein – hier schauen wir uns erst einmal die grundlegenden Funktionen von Notion an.

Notion ist eine Anwendung, die Sie sowohl im Webbrowser als auch über eine eigene App für die verschiedenen mobilen Endgeräte und Betriebssysteme bedienen können. Das macht es interessant, denn während wir zwar nur am Windows-Rechner an unseren Datenbanken programmieren können, so kommen zumindest mir doch immer wieder Ideen zu meinen Projekten, während ich gerade nicht an meinem Arbeitsplatz sitze. Dann kann ich mir jedoch schnell eine entsprechende Notiz in meiner Notion-App auf dem Smartphone machen, die ich dann später am Rechner wieder vorfinden und umsetzen kann.

Start mit Notion

Der Einstieg gelingt sehr schnell über die Webseite **www.notion.so** (siehe Bild 1). Hier findet sich der Button mit der Aufschrift **Try Notion free**, der einen zum Sign up-Bildschirm leitet.

Hier kann man sich aussuchen, ob man mit E-Mail oder einem bereits vorhandenen Konto von Google oder Apple einsteigen möchte.

Nach dem Anlegen eines neuen Accounts gibt man noch ein paar weitere Informationen wie den Namen und das Kennwort an. Außerdem gibt es noch ein paar Fragen zum Einsatzzweck von Notion. Anschließend legt man fest, ob man Notion mit einem Team nutzen möchte oder nur

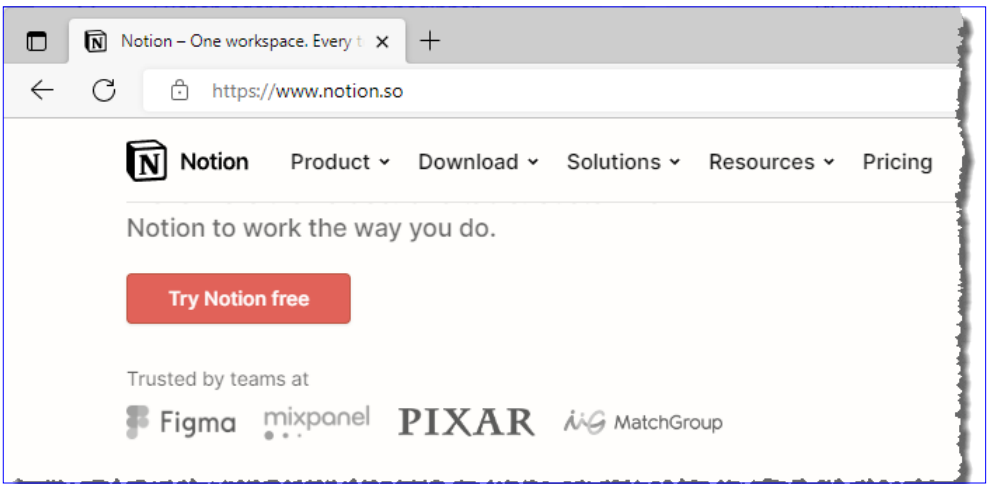


Bild 1: Einstieg in Notion

für sich allein. Wir wählen letztere Variante, weil nur diese ein kostenloses Konto bietet.

Gleich danach landen wir auf der Startseite, wo Notion ein paar Templates zur Orientierung für uns angelegt hat. Wer gern von Null beginnt, kann diese auch ablehnen (siehe Bild 2).

Die **Getting Started-**Seite bietet beispielsweise gleich einmal eine ToDo-Liste an und die Einträge in der Übersicht links bieten die Möglichkeit, andere Formate für Inhalte in Notion anzusehen.

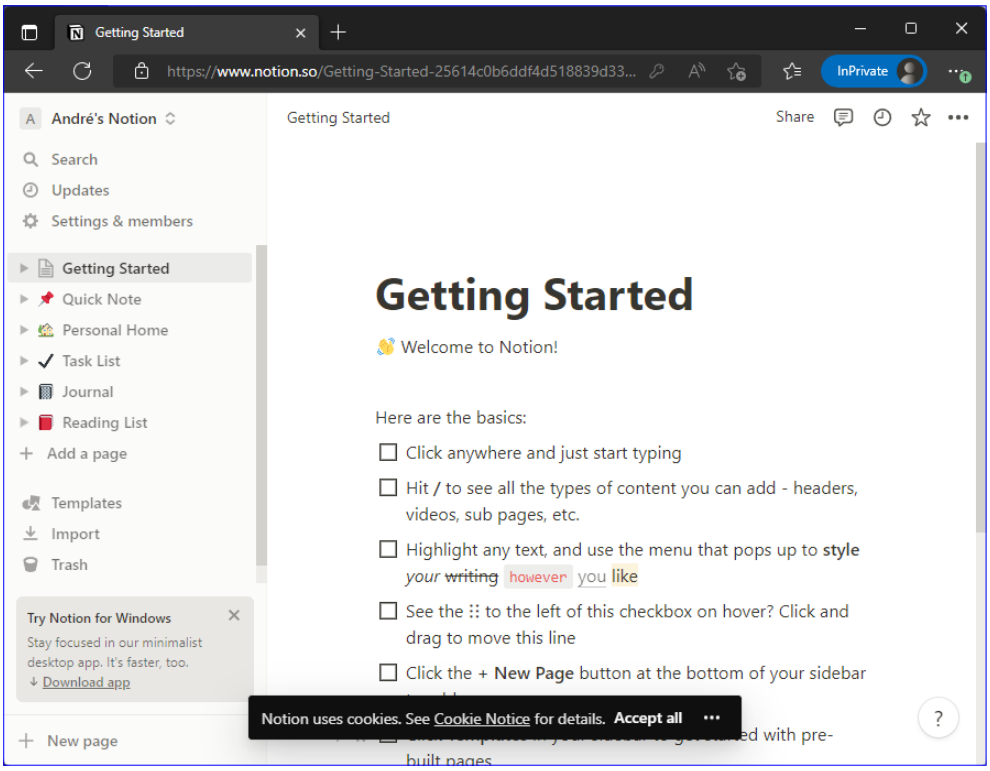


Bild 2: Die Getting-Started-Seite von Notion

Neue Seite anlegen

Wer gleich mit einer neuen, leeren Seite starten möchte, nutzt dazu den Eintrag **Add a page** im linken Menü. Auf der neuen, mit dem Titel **Untitled** versehenen Seite stehen dem User alle Möglichkeiten offen (siehe Bild 3). Als Erstes geben wir hier einen Titel ein, beispielsweise **ToDo-Liste**.

Wer Spaß an optischen Elementen hat, wird begeistert sein: Wir haben die Möglichkeit, ein Icon sowie ein Hintergrundbild als

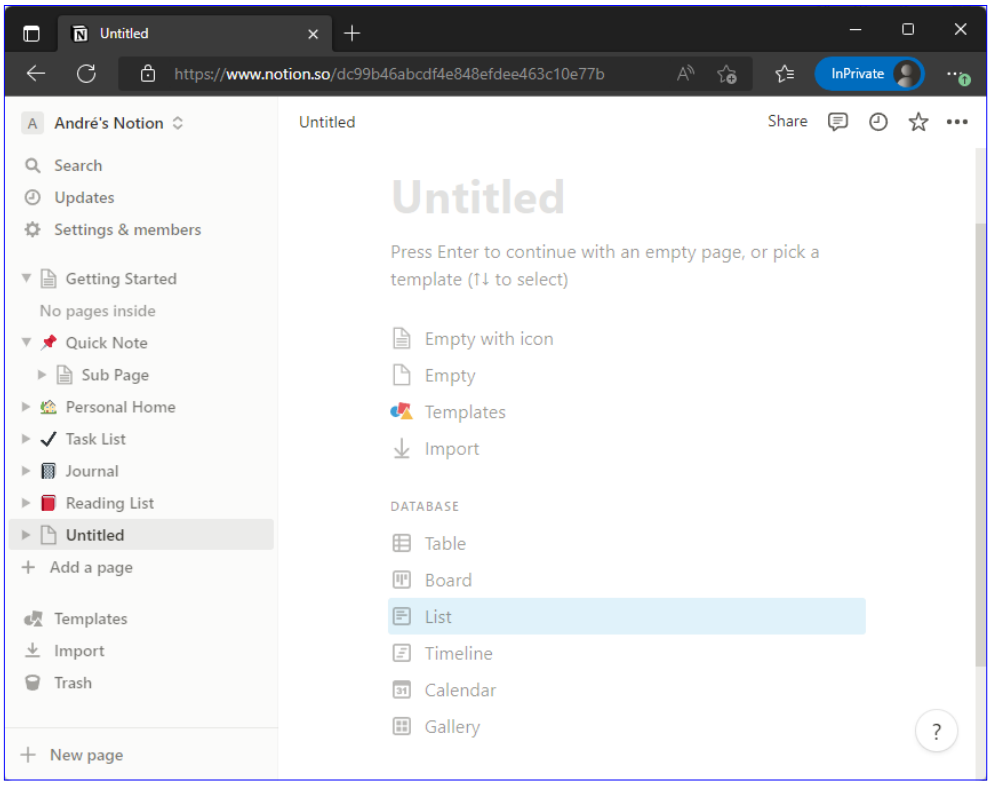


Bild 3: Eine neue, leere Seite

Seitenheader festzulegen. Die dazu notwendigen Befehle tauchen auf, wenn man wie in Bild 4 mit der Maus über den Titel fährt.

Ein paar Mausklicks später haben wir unsere ToDo-Seite mit den von Notion bereitgestellten Elementen optisch aufgewertet (siehe Bild 5).

Elemente zur Page hinzufügen

Um nun tatsächlich Inhalte hinzuzufügen, brauchen wir einfach nur etwas dort einzugeben, wo jetzt noch **Type '/' for commands** steht. Hier können wir nun einfach einen Text eingeben, oder wir fügen eines der vorgefertigten Elemente hinzu. Das gelingt am schnellsten durch die Eingabe des Schrägstrichs (*/*).

Dies zeigt wie in Bild 6 alle möglichen Formate und Elemente an, die wir an dieser Stelle einfügen können. Von normalem Text über ToDo-Listen und verschiedene Überschriftenebenen bis hin zu komplexeren Elementen wie Tabellen, verschiedene Listen, Zitate, Trennstriche, Links, Callouts, Formeln et cetera.

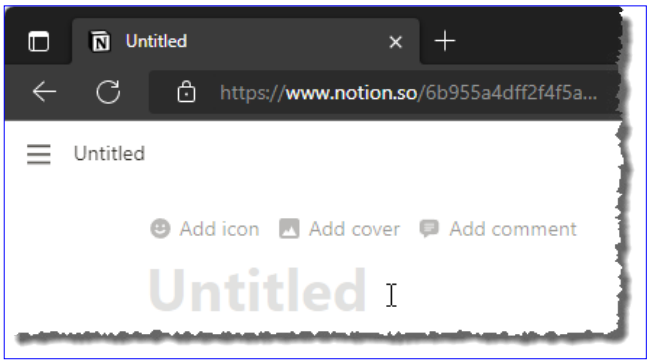


Bild 4: Hinzufügen von Icon, Cover und Comment

Außerdem gibt es noch sogenannte **Databases**. Hier war ich als Access-Entwickler zuerst einigermaßen verwirrt. Es ist aber schnell klar geworden, dass es sich nicht um Datenbanken, sondern um das, was wir unter Access unter einer Tabelle verstehen, handelt.

Vielleicht nennt Notion diese Tabellen **Database** und nicht **Table**, um eine Abgrenzung zu den Tabellen unter Excel zu erreichen. Der Unterschied besteht darin, dass eine Notion-Database ein oder mehrere Spalten enthält, für die feste Datentypen festgelegt werden müssen.

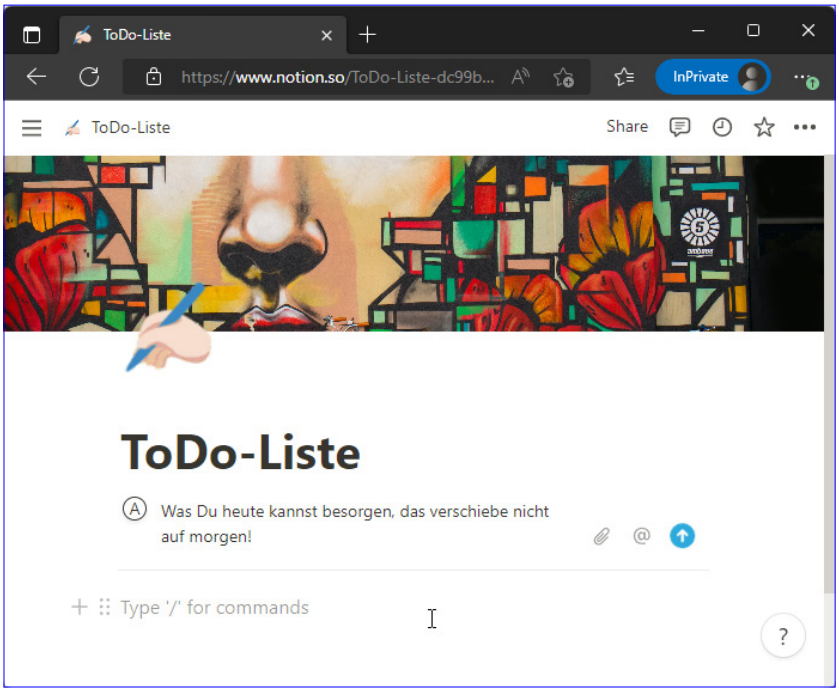


Bild 5: Eine optisch ansprechende ToDo-Liste – allerdings noch ohne ToDos.

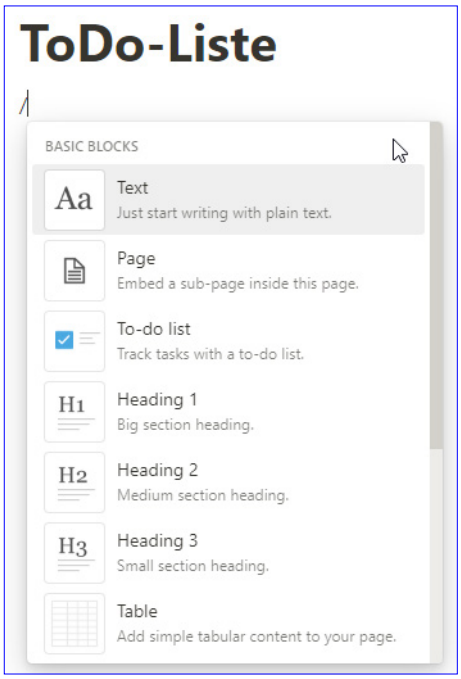


Bild 6: Auswahl der verfügbaren Elemente

Warum sollte man mit einer Notion-Database arbeiten und nicht einfach eine Access-Tabelle verwenden? Während eine Access-Tabelle nur eine Ansicht mit Daten bietet, nämlich die Datenblatt-Ansicht, liefern Notion-Databases gleich eine ganze Reihe von Ansichten, die alle je nach Anwendungszweck sehr praktisch sein können. Dazu kommen wir jedoch später.

Dazu kommen dann noch Bilder, Videos, Audios oder Dateien und man kann auch Code als Code formatiert unterbringen. Die Möglichkeiten sind wirklich umfassend und es ergibt Sinn, hier einfach einmal ein wenig herumzuspielen oder auch nur je nach der aktuellen Anforderung die entsprechenden Elemente zu nutzen.

Das Anlegen einfacher Elemente wie Überschriften, Text, Auflistungen et cetera ist relativ einfach, daher gehen wir direkt mal zum Anlegen und Verlinken von Seiten.

ToDo-Liste anlegen

Wir haben zwar nun eine Seite namens **ToDo-Liste** angelegt, aber darin noch keine ToDo-Liste erstellt. Dazu geben wir nun das Schrägstrich-Zeichen gefolgt von den ersten Buchstaben des Wortes **ToDo** ein, was den Eintrag **To-do list** aktiviert (siehe Bild 7).

Das anschließende Betätigen der Eingabetaste sorgt dafür, dass die ToDo-Liste mit einem ersten Element angelegt wird. Danach können wir schon starten, Einträge hinzuzufügen. Mit der Eingabetaste legen wir den nächsten Eintrag an. Die Tabulator-Taste ordnet den aktuellen Eintrag dem vorherigen unter, mit **Umschalt + Tab** holen wir den Eintrag wieder auf die übergeordnete Ebene (siehe Bild 8).

Weitere Elemente auf der gleichen Seite

Notion legt uns nicht auf ein Element pro Seite fest. Wir können beliebig viele verschiedene Elemente auf einer Seite ablegen. Vor oder hinter der ToDo-Liste passen also Texte, Überschriften und alle anderen verfügbaren Elemente, und natürlich können wir diese auch ohne ToDo-Liste anlegen.

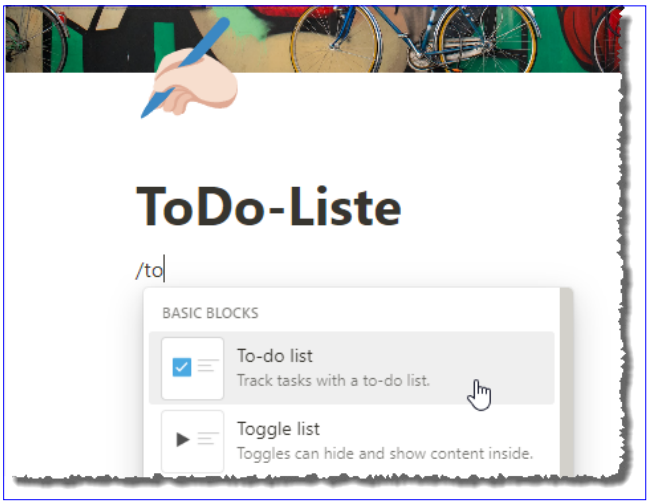


Bild 7: Hinzufügen der eigentlichen ToDo-Liste

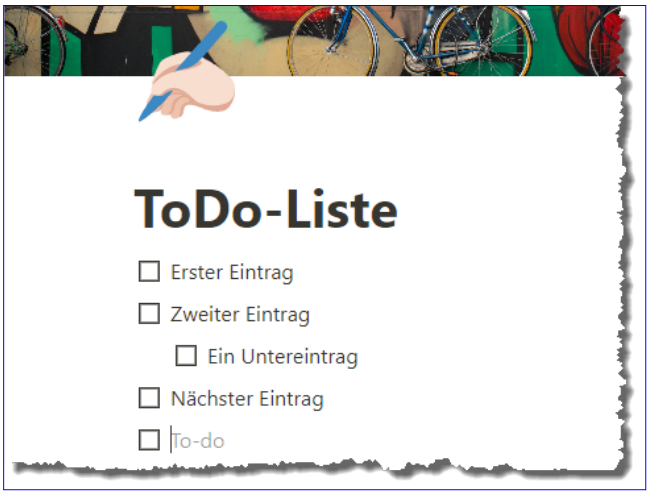


Bild 8: Einträge einer ToDo-Liste

Seiten in anderen Seiten verlinken

Diese ToDo-Liste könnte vielleicht der Mittelpunkt Ihres Notion-Workspaces sein, aber vermutlich wird es weitere Seiten geben. Auch wenn die ToDo-Liste nun in der Navigation im linken Bereich ansteuerbar ist, so möchten Sie vielleicht doch eine Übersichtsseite erstellen, die zu Beginn erscheint und Links zu den wichtigsten Seiten Ihres Workspace in Notion enthält.

Und, was noch wichtiger ist: Mit der nachfolgend vorgestellten Technik können Sie nicht nur in einer Übersichtsseite, sondern von überall auf andere Seiten verweisen. Dazu verwendet man nach dem Eingeben des Schräg-

strichs den Eintrag **Link to page**. Dies öffnet eine Liste aller bereits vorhandenen Seiten, aus denen wir die gewünschte auswählen (siehe Bild 9).

Der neue Link zu einer anderen Seite erscheint wie in Bild 10.

Klicken wir nun auf diesen Link, wechseln wir direkt zu der verlinkten Seite. Diese enthält im oberen Bereich nun die Angabe **1 backlink**, was bedeutet, dass es eine Seite gibt, die auf diese Seite per Link verweist. Klicken wir diesen Eintrag an, erscheint eine Liste der referenzierenden Seiten, auf die wir über diesen Weg zugreifen können (siehe Bild 11).

Mit Notion-Databases arbeiten

Richtig spannend wird es, wenn wir mit den sogenannten Databases arbeiten, was eigentlich Tabellen sind. Es gibt auch das **Table**-Objekt in Notion. Das ist aber nur eine einfache Tabelle ohne jegliche Funktion – es ist also kein Filtern oder Sortieren darin möglich, sondern nur die Darstellung von Werten in Tabellenform.

Um eine neue Database hinzuzufügen, erstellen wir eine neue Seite namens **Artikelübersicht**. Die Database können wir gar nicht explizit erzeugen, sondern wir legen eine der verfügbaren Views für eine Database an. Dazu nutzen wir wieder den Schrägstrich plus die Auswahl eines der Einträge unterhalb von **Database**, in diesem Fall **Table View** (siehe Bild 12).

Als Erstes legen wir die Datenquelle fest. Dazu können wir eine der Beispieldatenquellen verwenden, die mit dem Workspace automatisch angelegt wurden, oder wir klicken auf **New Database**, um eine neue Database hinzuzufügen (siehe Bild 13).

Dies legt eine neue, leere Tabelle an und öffnet diese in der gewählten Ansicht, in diesem Fall **Table** (siehe Bild 14). Statt **Untitled** geben wir hier als Erstes einen Database-Namen ein, zum Beispiel **Artikeldatenbank**.

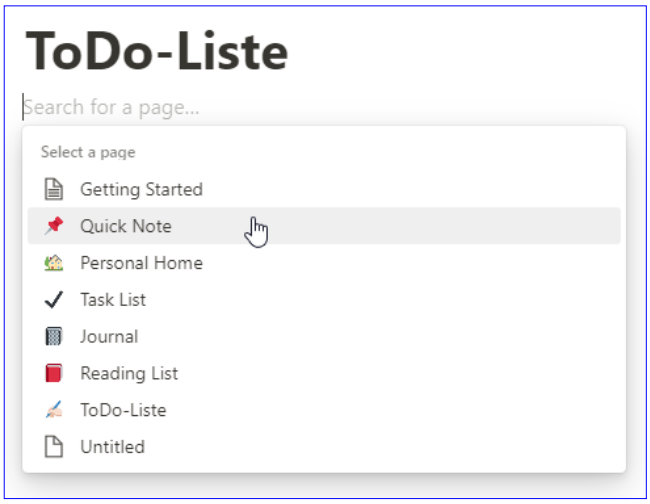


Bild 9: Hinzufügen eines Links zu einer anderen Seite

Bild 10: Link zu einer anderen Seite

Bild 11: Backlink zur verlinkenden Seite

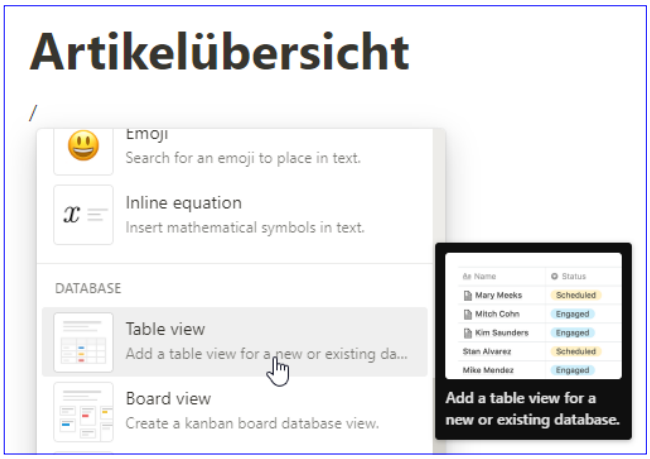


Bild 12: Hinzufügen einer Database in der **Table View**

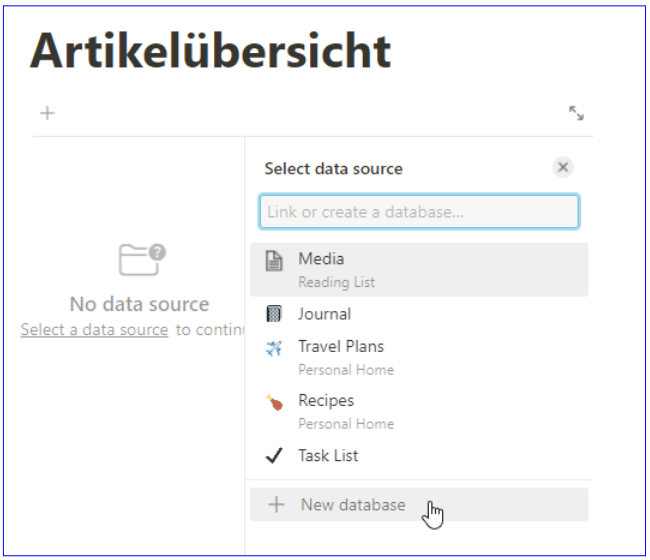


Bild 13: Anlegen einer neuen Database

In der Datenbank finden wir bereits zwei Spalten vor, **Name** und **Tags**, und einige leere Datensätze. Hier können wir nun Daten eingeben oder aber zuerst die Spalten anpassen. Dazu klicken wir auf den Spaltenkopf und erhalten so ein Popup, mit dem wir die wichtigsten Informationen einstellen können. Dazu gehören der Name und der Datentyp und wir können auch gleich eine Sortierung und einen Filter angeben. Letztere werden übrigens nicht mit der Database gespeichert, sondern sind Eigenschaften dieser View.

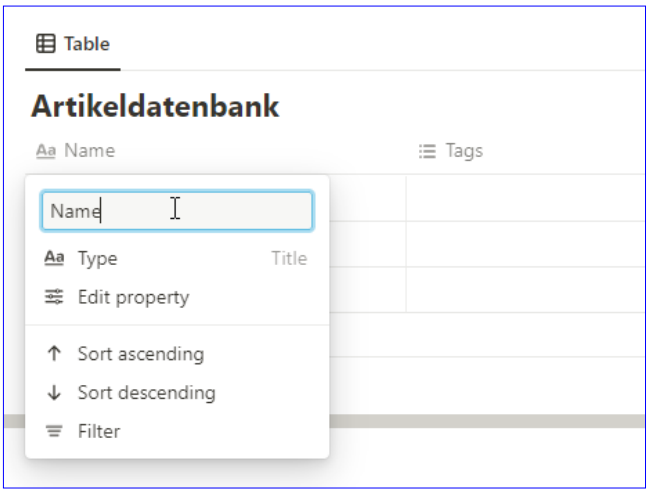


Bild 15: Festlegen eines Feldnamens

Bild 14: Eine neue, leere Datenbank in der **Table**-Ansicht

Gleich bei der ersten Spalte namens **Name** kann man weder den Datentyp ändern – dieser lautet **Title** –, noch kann man dieses Feld entfernen (siehe Bild 15). Aber auch hierfür gibt es einen Grund, den wir uns gleich im Anschluss ansehen. Als Erstes ändern wir den Namen dieses Feldes in **Artikelname**. Dann fügen wir zwei, drei Einträge hinzu.

Klicken wir danach auf einen der neuen Einträge, erscheint im Feld **Artikelname** für diesen Eintrag ein Befehl namens **Open** (siehe Bild 16).

Damit öffnen wir eine Art Detailansicht zum aktuellen Eintrag, wo die einzelnen Felder samt den Werten (die hier noch nicht vorhanden sind) untereinander angezeigt werden. Diese können Sie hier auch bearbeiten und außerdem ist es möglich, einen Kommentar zu diesem Eintrag anzulegen. Dieser wird nicht als Spaltenwert der Datenbank abgespeichert, sondern ist nur in der Detailansicht sichtbar (siehe Bild 17).

Bild 16: Button zum Öffnen der Details

Mit Access auf Notion zugreifen

Notion ist die Produktivitätsapp der Stunde, wenn es um Verwaltung von Listen, Projekten, Teams und vieles mehr geht. Eigentlich sind die Möglichkeiten nur durch die Phantasie begrenzt. Logisch, dass wir uns ansehen wollen, ob man die Daten, die man in Notion angelegt hat, auch von Access aus einlesen kann oder ob man sogar Daten von Access aus nach Notion verschieben kann.

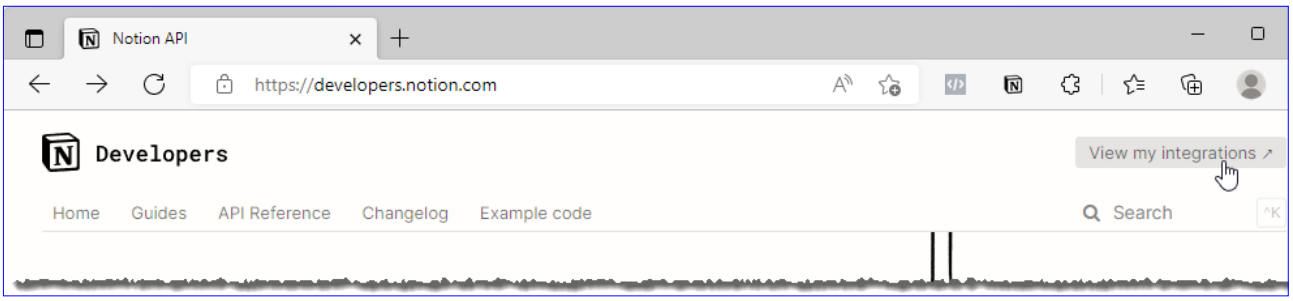


Bild 1: Von der Startseite des Entwicklerbereichs geht es gleich weiter zu den Integrationen.

In einem weiteren Beitrag namens **Produktivität mit Notion steigern** (**www.access-im-unternehmen.de/1402**) zeigen wir die Grundlagen der App **Notion**. Im vorliegenden Beitrag schauen wir uns an, wie wir die dort angelegten Daten von einer Access-Anwendung aus auslesen und sogar schreiben können. Damit erleichtern wir uns einige Arbeiten, zum Beispiel das Anlegen größerer Mengen von Terminen, das Eintragen von Daten aus den Tabellen einer Datenbank et cetera.

Wir gehen davon aus, dass Sie einen Notion-Account angelegt haben. Mit diesem ist auch die Möglichkeit verbunden, sogenannte **Integrations** anzulegen. Dazu wechseln wir zur folgenden Webseite:

```
https://developers.notion.com
```

Dort finden wir oben links einen Link mit dem Text **View my integrations** (siehe Bild 1).

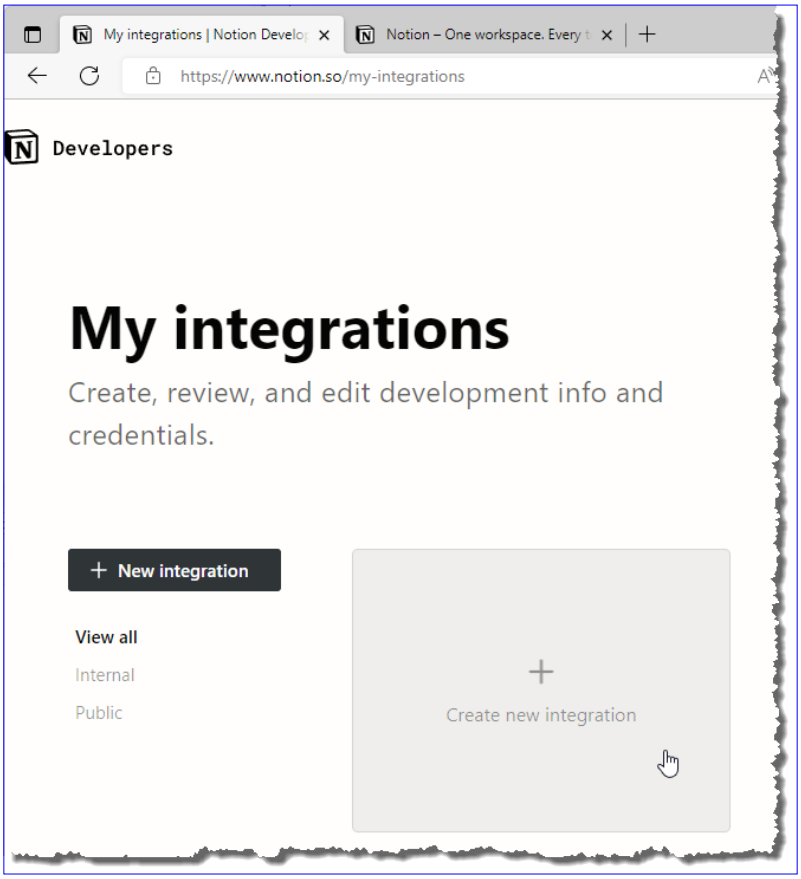


Bild 2: Anlegen einer neuen Integration

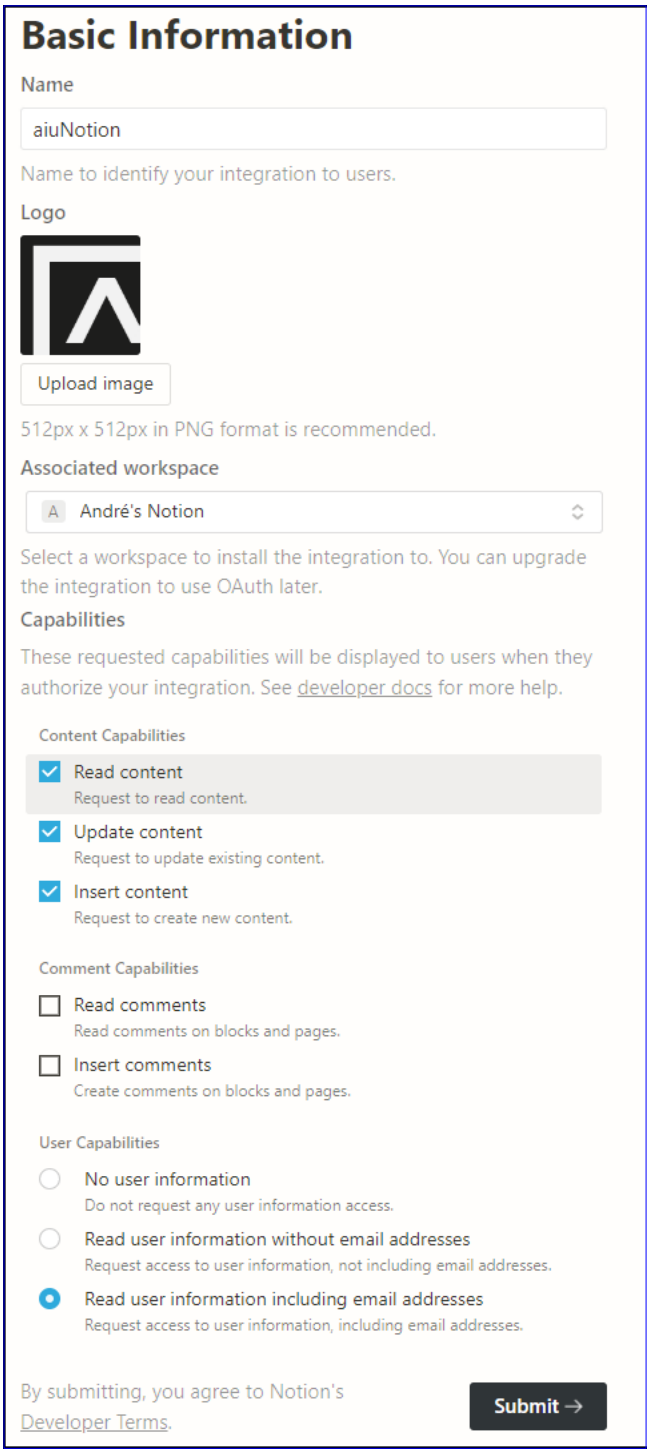


Bild 3: Festlegen der Eigenschaften der Integration

Klicken wir diesen an, landen wir direkt auf der Seite **My integrations**, wo wir mit einem Klick auf die Schaltfläche **Create new integration** eine neue Integration anlegen können (siehe Bild 2).

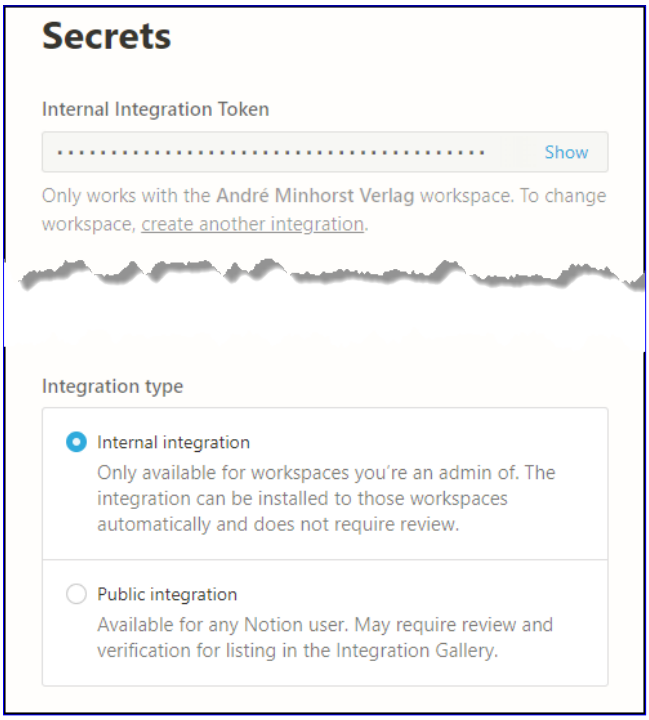


Bild 4: Wichtige Information nach dem Anlegen der Integration

Neue Notion-Integration anlegen

Auf der folgenden Seite fragt Notion einige Informationen über die anzulegende Integration ab. Dazu gehören der Name, ein Logo, ein assoziierter Workspace und die Rechte, die der Integration erteilt werden sollen.

Wir legen die Daten wie in Bild 3 fest. Wenn Sie gerade einen neuen Notion-Account angelegt haben, ist die Zusammenstellung der Daten recht einfach – das Feld **Associated Workspace** bietet dann nur einen einzigen Eintrag an.

Nach dem Klick auf die Schaltfläche **Submit** erscheint eine weitere Seite, auf der wir eine wichtige Information vorfinden: den **Internal Integration Token**. Das ist unsere Eintrittskarte für den VBA-gesteuerten Zugriff auf unsere Notion-Daten (siehe Bild 4). Diesen Token können wir direkt einmal in die Zwischenablage kopieren, wir werden

diesen gleich benötigen. Auf dieser Seite behalten wir die Option **Internal integration** bei, da wir davon ausgehen, dass wir erst einmal nur eine Integration für den Eigengebrauch anlegen wollen. Außerdem fasst die Seite nochmals die zuvor festgelegten Einstellungen zusammen.

Damit sind die Arbeiten auf der Seite von Notion bereits abgeschlossen – wir benötigen hier nur den Token, um diesen als Schlüssel für den Zugriff auf die Daten in unserem Notion-Workspace zu verwenden.

Zur Sicherheit schauen wir noch unter **My integrations** nach und finden dort die neu angelegte Integration vor (siehe Bild 5).

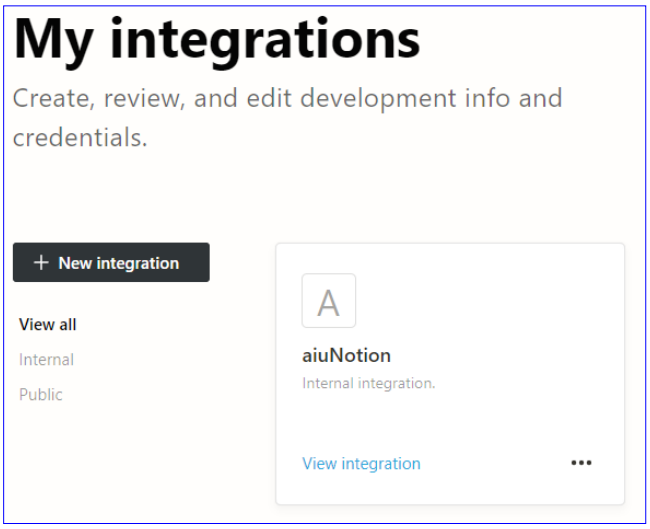


Bild 5: Die neu angelegte Integration

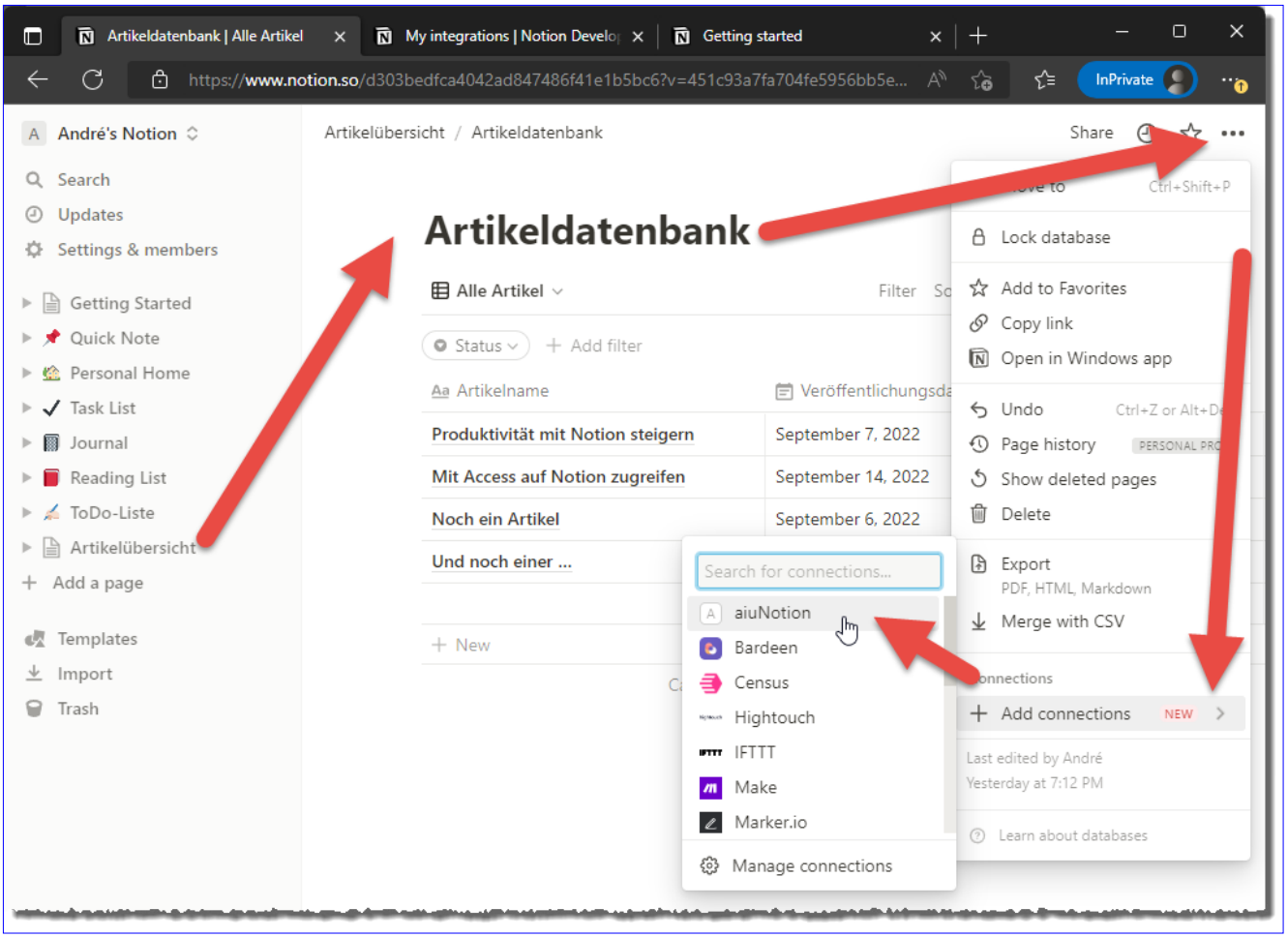


Bild 6: Anlegen einer Verbindung zu einer Notion-Datenbank

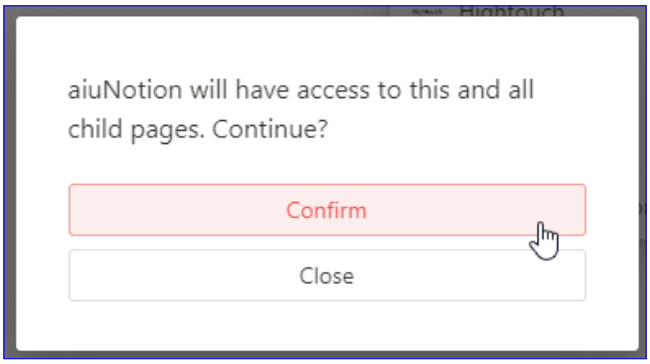


Bild 7: Bestätigen der Freigabe für die Integration

Elemente für die Integration mit VBA freigeben

Bevor wir per VBA auf Datenbanken, Seiten et cetera in Notion zugreifen können, müssen wir diese freigeben beziehungsweise mit unserer Integration teilen. Das gelingt jedoch mit wenigen Schritten.

In meinem Notion-Workspace habe ich eine Datenbank namens **Artikelübersicht** erstellt, mit der ich meine Artikel verwalten möchte – siehe **Produktivität mit Notion steigern** (**www.access-im-unternehmen.de/1402**). Wenn ich diese über den Eintrag in der Seitenleiste aktiviere (entweder in der App oder im Browser), erscheint die entsprechende Datenbank im Hauptbereich.

Nun klicken wir rechts oben auf die drei Punkte, was ein Kontextmenü öffnet. Hier finden wir unten den Eintrag **Connections|Add connections**.

Klicken wir diesen an, sollten wir bereits den Namen unserer Integration in der Liste vorfinden (siehe Bild 6).

Danach erscheint eine Meldung, die wir mit **Confirm** bestätigen (siehe Bild 7).

Ab dann finden wir im soeben geöffneten Kontextmenü unter **Connections** einen neuen Eintrag mit dem Namen unserer Integration. Diese Verbindung können Sie über den einzigen Befehl des Untermenüs dieses Eintrags wieder trennen.

Jede Datenbank einzelnen freigeben

Falls Sie im Laufe der Zeit weitere Datenbanken zu Notion hinzufügen, tauchen diese nicht automatisch auf, wenn Sie, wie nachfolgend beschrieben, die Datenbanken ihres Workspaces auslesen.

Sie müssen jede Datenbank, auf die Ihre Integration zugreifen können soll, einzeln wie oben beschreiben freigeben.

API-Befehle herausfinden

Die Notion-API enthält eine ganze Reihe Befehle, mit denen die meisten der Aktionen durchgeführt werden können, die Sie sonst über die Benutzeroberfläche erledigen würden.

Eine Übersicht der API finden Sie unter dem folgenden Link:

```
https://developers.notion.com/reference/intro
```

Hier gibt es beispielsweise die folgenden API-Befehle:

- **search**: Mit diesem Befehl, angehängt an die URL **https://api.notion.com/v1/**, lesen Sie, wenn ohne Parameter verwendet, alle **database**- oder **page**-Elemente ein, die für die Integration freigegeben wurden. Das enthält also auch die **page**-Elemente, die innerhalb eines **database**-Elements angelegt wurden. Übergeben wir zusätzlich weitere Elemente wie beispielsweise das **query**-Element per Header, können wir auch **database**- oder **page**-Elemente nach dem Titel ermitteln – ein Beispiel finden Sie weiter unten.

- **databases**: Hängen wir den Befehl **databases** an die URL **https://api.notion.com/v1/** an, müssen wir noch die ID der Datenbank folgen lassen. Diese ID erhalten wir beispielsweise über den obigen **search**-Befehl. Außerdem folgt noch **/query**, sodass die URL beispielsweise **https://api.notion.com/v1/databases/d303bedf-ca40-42ad-8474-86f41e1b5bc6/query**

```
Public Function HTTPRequest(strURL As String, strAuthorization As String, Optional strMethod As String = "POST", _
        Optional strContentType As String = "application/json", Optional strVersion As String = "2022-06-28", _
        Optional strData As String, Optional strResponse As String) As Integer
    Dim objHTTP As ServerXMLHTTP60
    Set objHTTP = New MSXML2.ServerXMLHTTP60
    With objHTTP
        .Open strMethod, strURL, False
        .setRequestHeader "Accept", strContentType
        .setRequestHeader "Content-Type", strContentType
        .setRequestHeader "Authorization", strAuthorization
        .setRequestHeader "Notion-Version", strVersion
        .send strData
        strResponse = .responseText
        HTTPRequest = .status
    End With
End Function
```

Listing 1: Prozedur zum Ausführen von HTTP-Anfragen

lautet. Auch hier kann man per Header noch Filterausdrücke übergeben, mit denen nur den Kriterien entsprechende **page**-Elemente ermittelt werden.

- **pages**: Ebenso wie auf **database**-Elemente können wir auch auf **page**-Elemente zugreifen. Dazu müssen wir zuvor die ID der jeweiligen Seite ermittelt haben, was wir beispielsweise mit dem obigen **databases**-Befehl erreichen.

Funktion zum Ausführen einer Anfrage an die Notion-API

Wir definieren als Nächstes eine allgemeine Funktion für das Absenden einer HTTP-Anfrage an die Notion-API (siehe Listing 1). Diese erwartet die folgenden Parameter:

- **strURL**: Aufzurufender API-Endpoint, beispielsweise **https://api.notion.com/v1/search**
- **strMethod**: Optionaler Parameter für die Methode des Aufrufs, also **POST** (Standard), **GET**, **PUT** oder **DELETE**.
- **strAuthorization**: Parameter zum Übergeben des Ausdrucks aus **Bearer** und **Token**
- **strContentType**: Optionaler Parameter zur Übergabe des Inhaltstyps, standardmäßig **application/json**.
- **strVersion**: Optionaler Parameter zur Übergabe der zu verwendenden API-Version, standardmäßig **2022-06-28**.
- **strData**: Optionaler Parameter zur Übergabe zusätzlicher Daten, in diesem Fall im JSON-Format
- **strResponse**: Optionaler Parameter zum Zurückgeben des Ergebnisses des API-Aufrufs

Bevor wir in die Beschreibung der Funktion einsteigen, fügen wir noch einen Verweis auf die Bibliothek **Microsoft XML, v6.0** zum Projekt hinzu. Das erledigen wir im **Verweise**-Dialog des VBA-Editors, den wir mit dem Menübefehl **Extras|Verweise** öffnen (siehe Bild 8).

Die Prozedur erstellt ein neues Objekt des Typs **ServerXMLHTTP60** und ruft dessen **Open**-Methode auf, um die Methode (**POST**) und die URL der API zu übergeben.

Der dritte Parameter gibt mit dem Wert **False** an, dass der Aufruf nicht asynchron gestartet werden soll.

Danach weist die Funktion einige Header hinzu. **Accept** und **Content-Type** erhalten den Wert aus **strContentType**, also standardmäßig **application/json**. Den Header **Authorization** füllen wir mit dem Wert des Parameters **strAuthorization**. Schließlich weisen wir **Notion-Version** den Wert aus **strVersion**, typischerweise **2022-06-28**, zu.

Mit der **send**-Methode schicken wir noch den Inhalt von **strData**. strData nutzen wir, wenn wir zusätzliche Daten zu den mit der URL gesendeten Daten übergeben müssen.

Wenn wir nur Informationen zu verschiedenen Elementen von Notion abrufen wollen wie zu einem **database**- oder **page**-Objekt, brauchen wir nur eine ID per Parameter zu übergeben. Wenn wir jedoch neue **database**- oder **page**-Objekte hinzufügen wollen, dann müssen wir die Eigenschaften dieser Objekte übergeben.

Diese stellen wir wiederum in einem JSON-Dokument zusammen und übergeben dieses mit dem Parameter **strData**. Die **send**-Methode startet auch den eigentlichen Aufruf der API-Funktion von Notion. Das Ergebnis finden wir anschließend in verschiedenen Eigenschaften. **status** liefert einen Zahlenwert wie beispielsweise **200** für einen erfolgreichen Aufruf.

Diesen Wert gibt die Funktion als Ergebnis zurück. Das eigentliche Ergebnis, also beispielsweise Informationen über die Notion-Datenbanken, liefert die Eigenschaft **responseText**, deren Wert wir mit dem Parameter **strResponse** an die aufrufende Instanz zurückschicken.

Konstanten definieren

Im Kopf des Moduls **mdlNotion** speichern wir einige Werte in Konstanten. Diese sehen wie folgt aus:

```
Const cStrURLSearch As String = _
    "https://api.notion.com/v1/search"
Const cStrNotionKey As String = _
    "secret_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
Const cStrContentType As String = "application/json"
Const cStrVersion As String = "2022-06-28"
```

Alle Datenbanken abfragen

Unser erster Aufruf der Funktion **HTTPRequest** soll alle Datenbanken abfragen, die wir in unserem Notion-Workspace angelegt haben. Bevor wir die notwendige Funktion zum Einlesen der Datenbanken per JSON beschreiben, schauen wir uns die Prozedur an, mit der wir einen Test der Funktion starten.

Diese ruft die Funktion **GetJSON** auf und übergibt dieser den zu verwendenden Link (aus **cStrURLSearch**), die Art des Zugriffs (hier **POST**) sowie eine leere **String**-Variable, die mit dem JSON-Dokument gefüllt werden soll. Wenn die Funktion den Wert **True** zurückliefert, enthält die Variable **strResponse** das JSON-Dokument mit den gewünschten Daten. Falls nicht, enthält es die entsprechende Fehlermeldung:

```
Public Sub Test_SEARCH()
    Dim strResponse As String
```

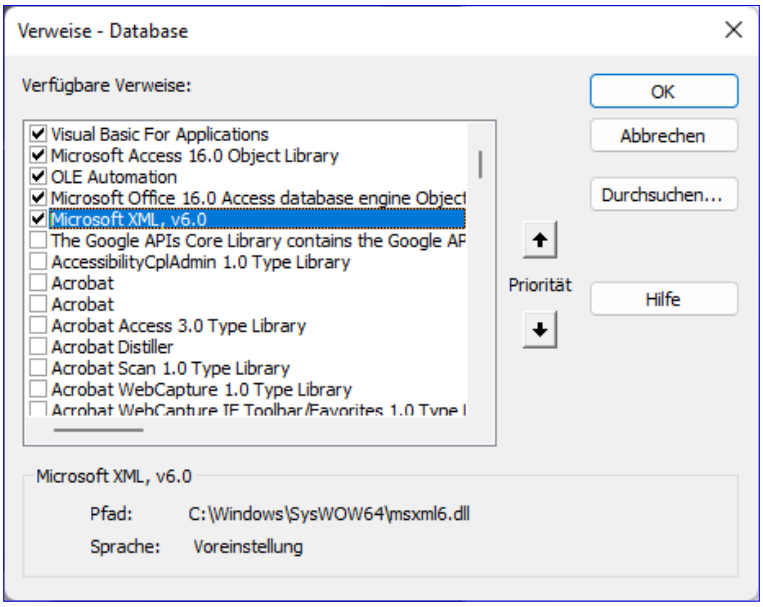


Bild 8: Notwendiger Verweis auf die Bibliothek **Microsoft XML, v6.0**