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

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.

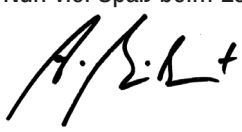
Bevor wir die Formulare zur Verwaltung der Kunden, Produkte und Bestellungen vorstellen, benötigen wir einige Beispieldaten. Wie Sie diese per Mausklick hinzufügen können, zeigen wir im Beitrag **Rechnungsverwaltung: Beispieldaten** ab Seite 25. Hier nutzen wir eine .NET-Bibliothek, um die Tabellen schnell 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.

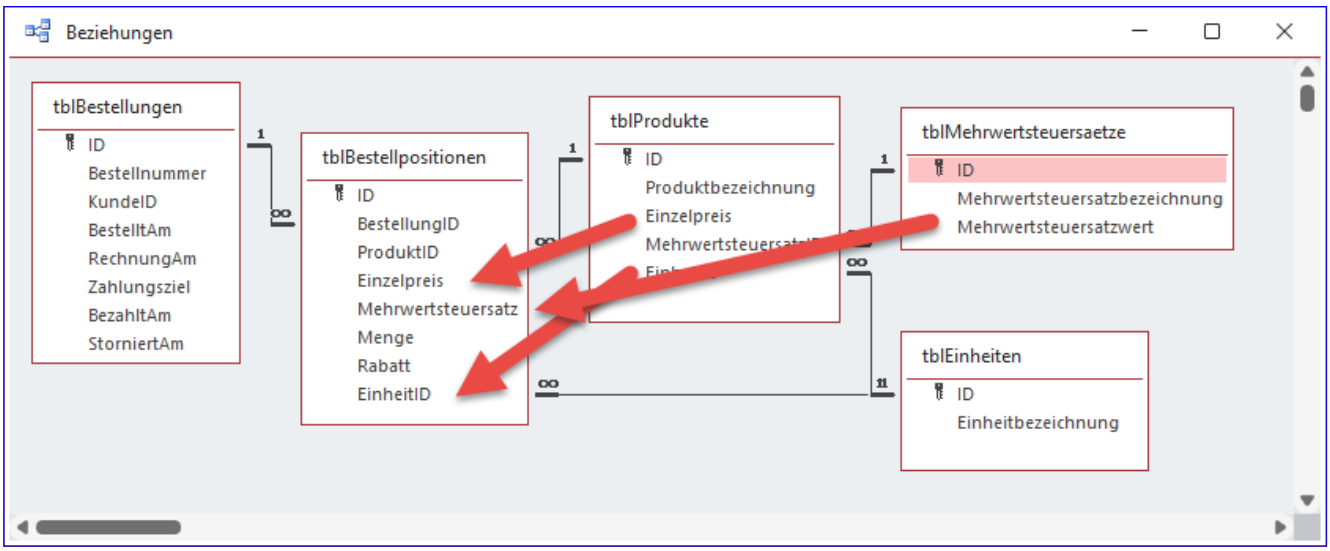


Bild 1: Beteiligte Tabellen dieser Lösung

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.

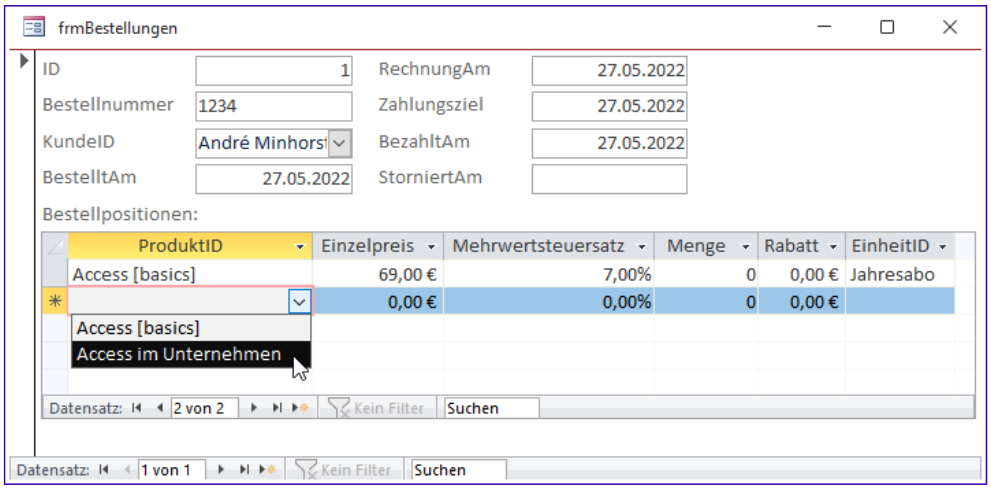


Bild 2: Formular zum Verwalten von Bestellungen und Bestellpositionen

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.

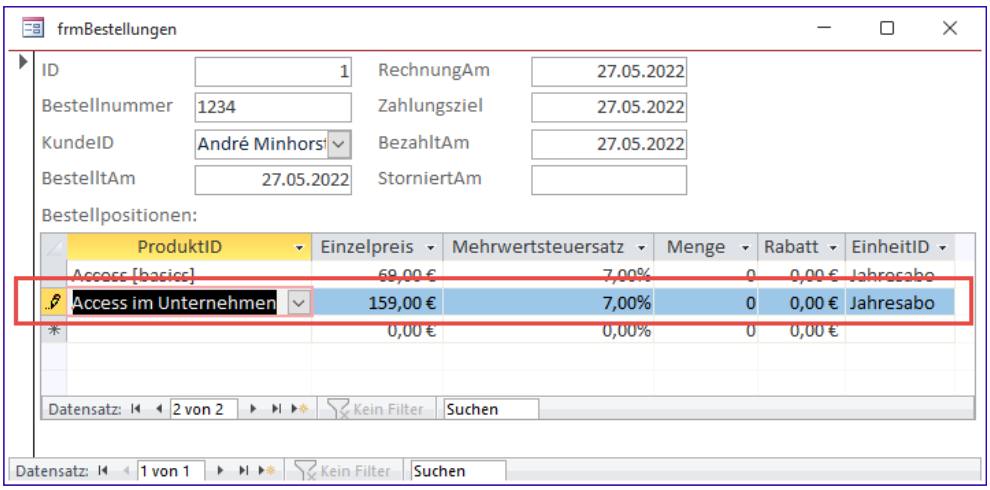


Bild 3: Ergänzte Daten einer Bestellposition

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).

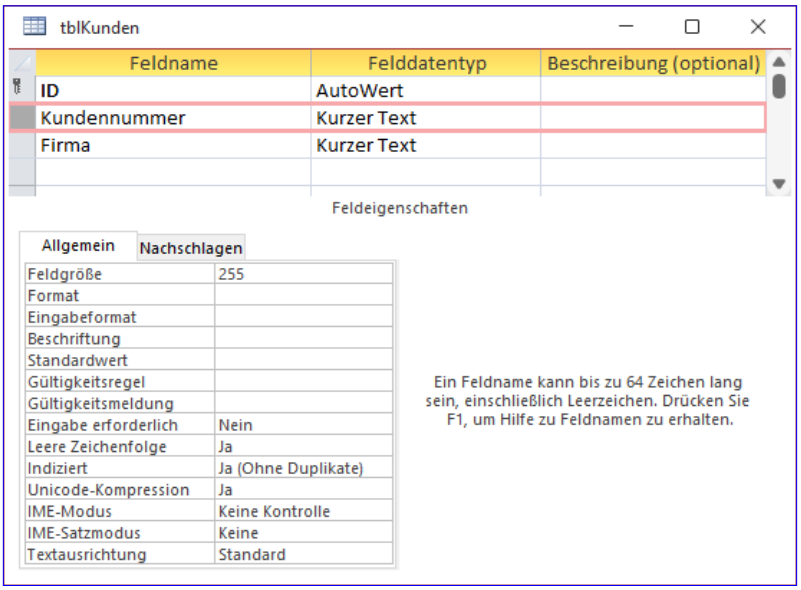


Bild 1: Einrichtung der Kundennummer als Feld mit einem eindeutigen Index

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.

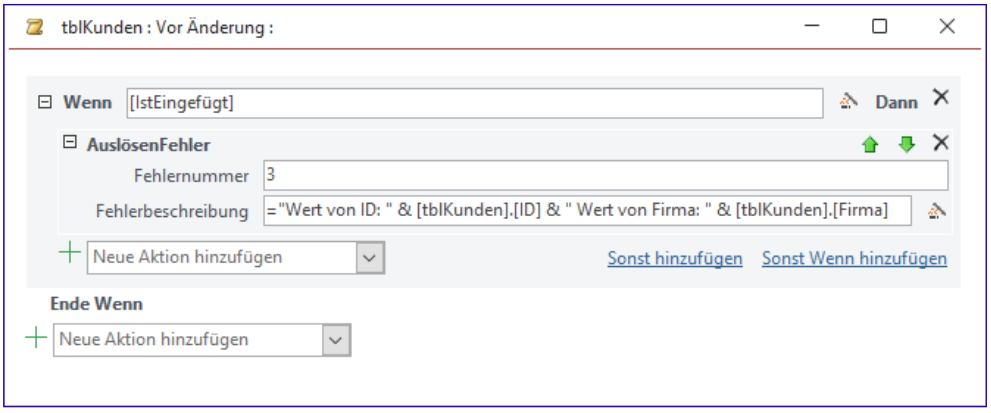


Bild 2: Makro, um herauszufinden, ob das Feld **ID** beim Ereignis **Vor Änderung** bereits gefüllt ist.

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).

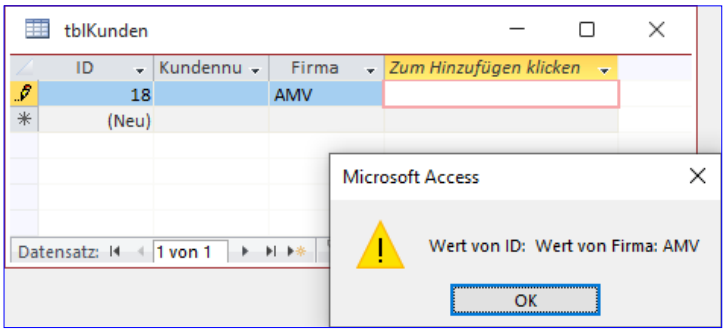


Bild 3: Ergebnis des Makros **Vor Änderung**

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-

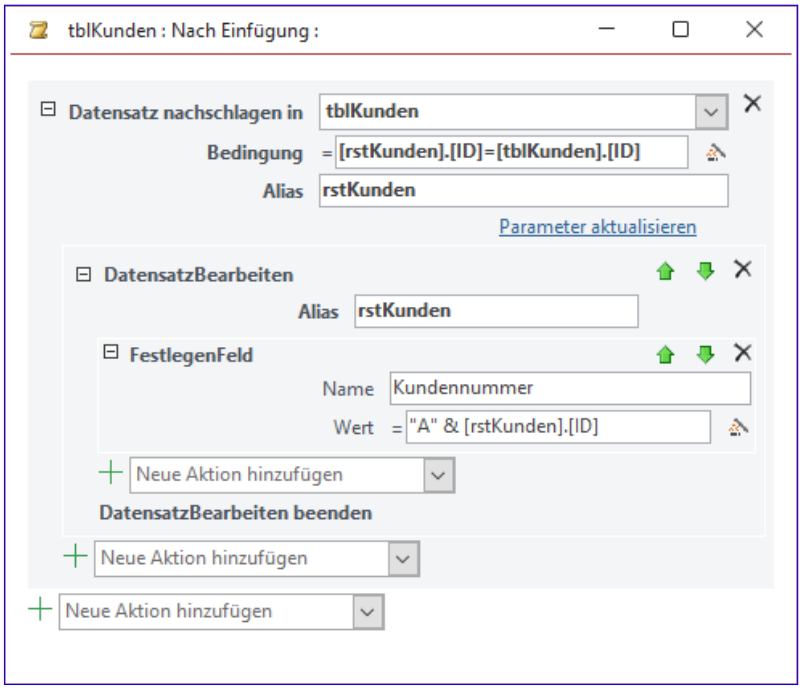


Bild 4: Das Datenmakro für das Ereignis **Nach Einfügung**

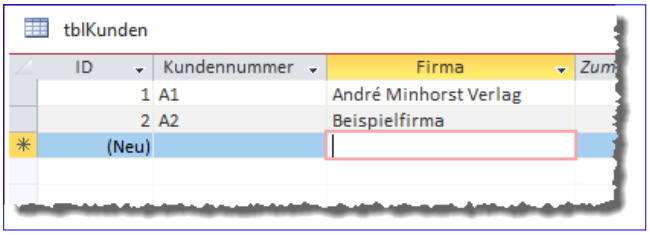


Bild 5: Ergebnis des Datenmakros

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

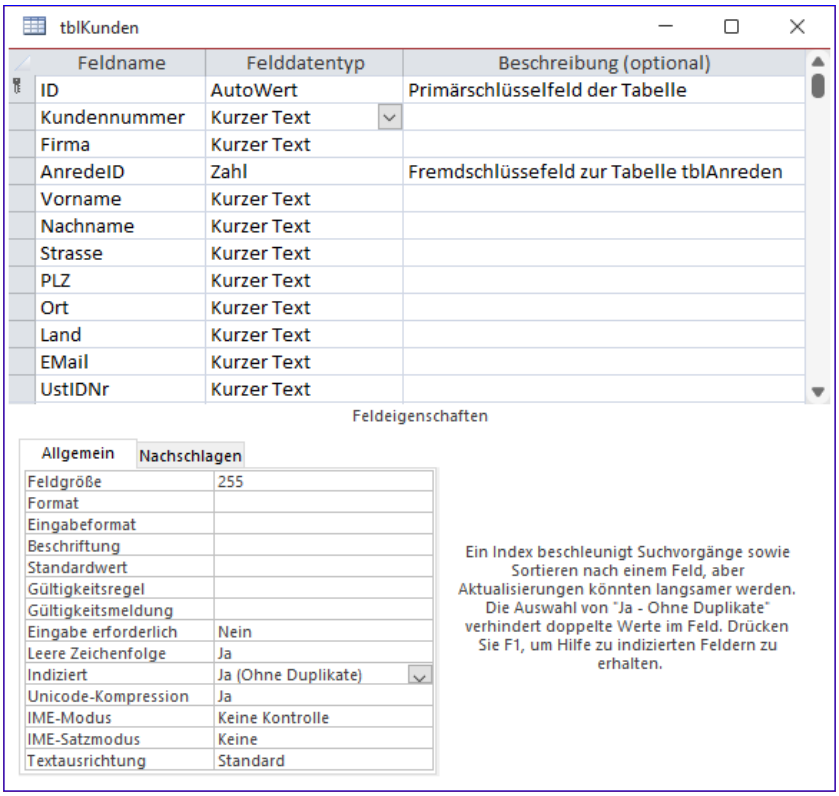


Bild 1: Die Tabelle **tblKunden**

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.

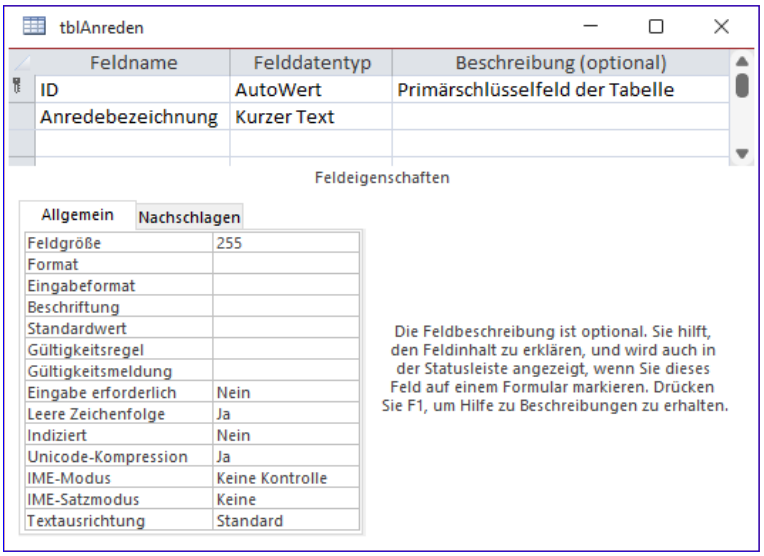


Bild 2: Die Tabelle **tblAnreden**

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).

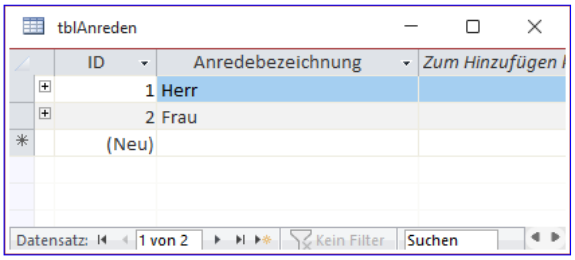


Bild 3: Die Tabelle **tblAnreden** in der Datenblattansicht

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.

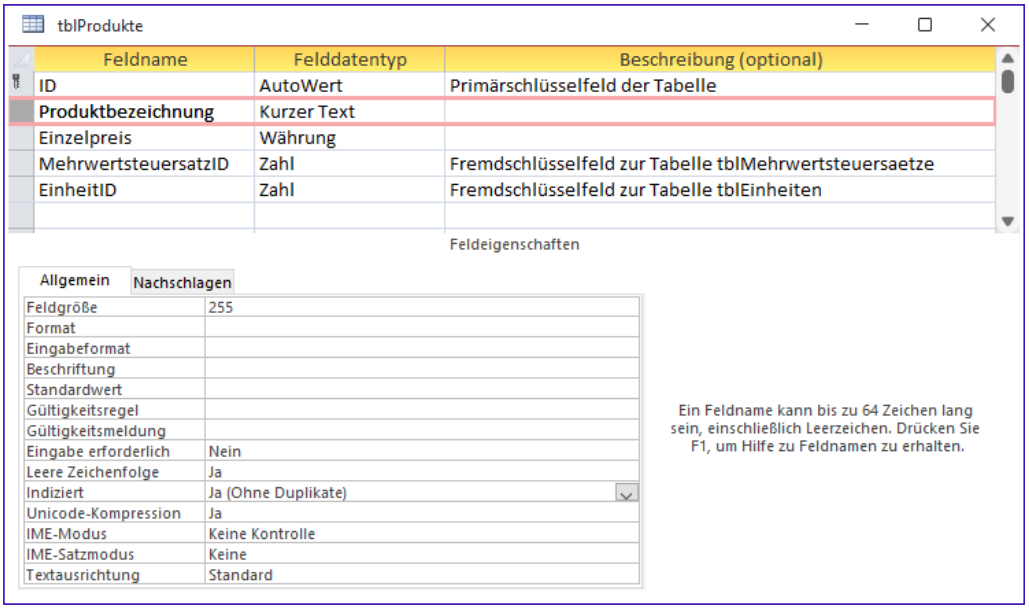


Bild 4: Die Tabelle **tblProdukte**

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.

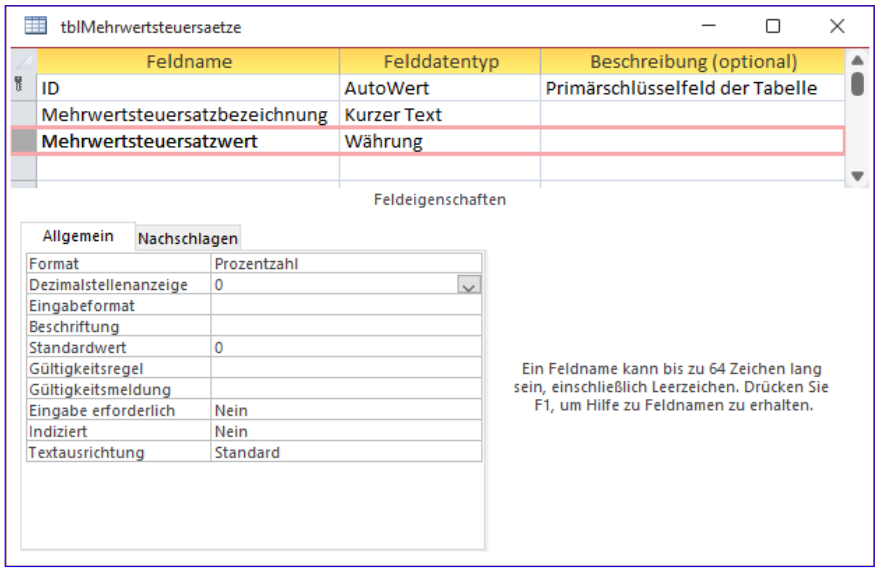


Bild 5: Die Tabelle **tblMehrwertsteuersaetze** in der Entwurfsansicht

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**.

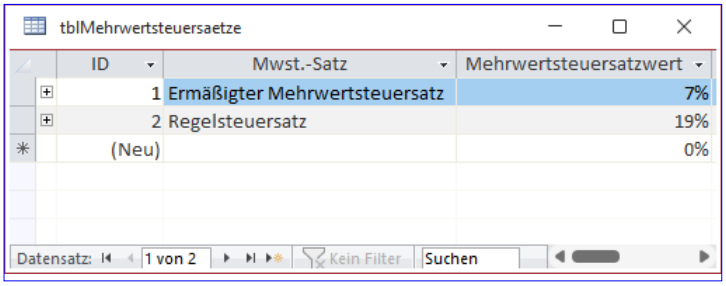


Bild 6: Die Tabelle **tblMehrwertsteuersaetze** in der Datenblattansicht

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

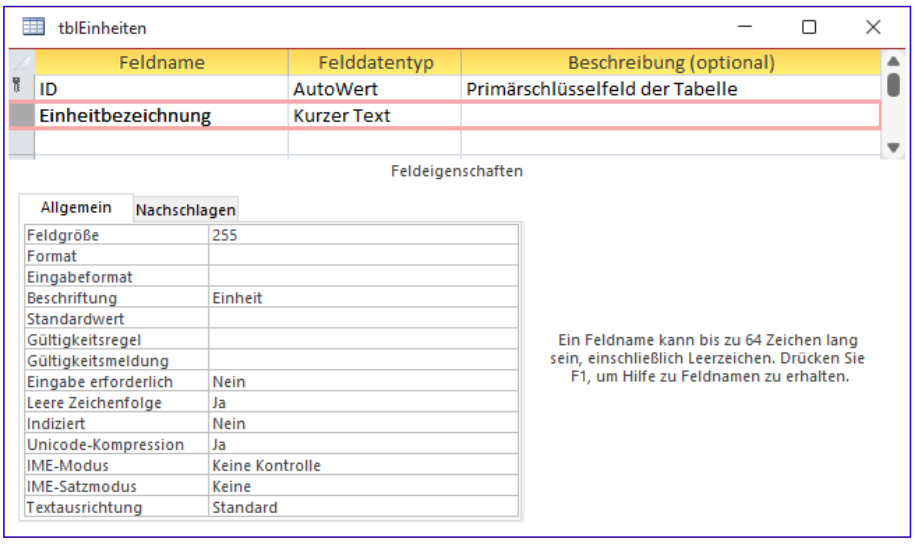


Bild 7: Die Tabelle **tblEinheiten** in der Entwurfsansicht

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.

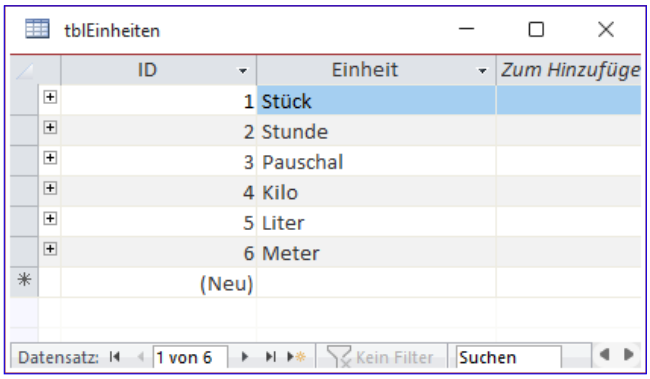


Bild 8: Die Tabelle **tblEinheiten** mit einigen Beispieldatensätzen

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-

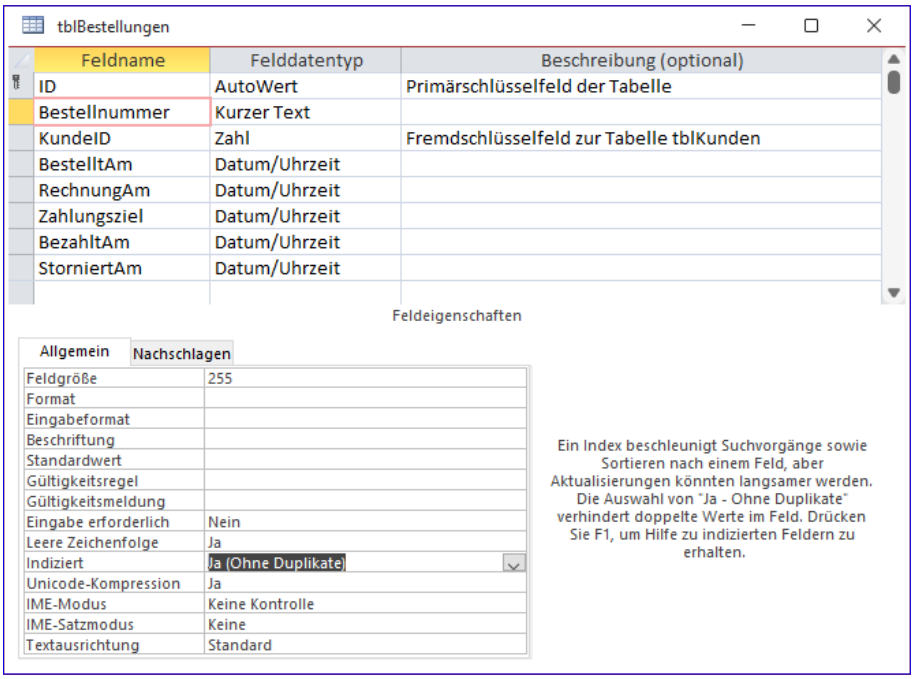


Bild 9: Die Tabelle **tblBestellungen** in der Entwurfsansicht

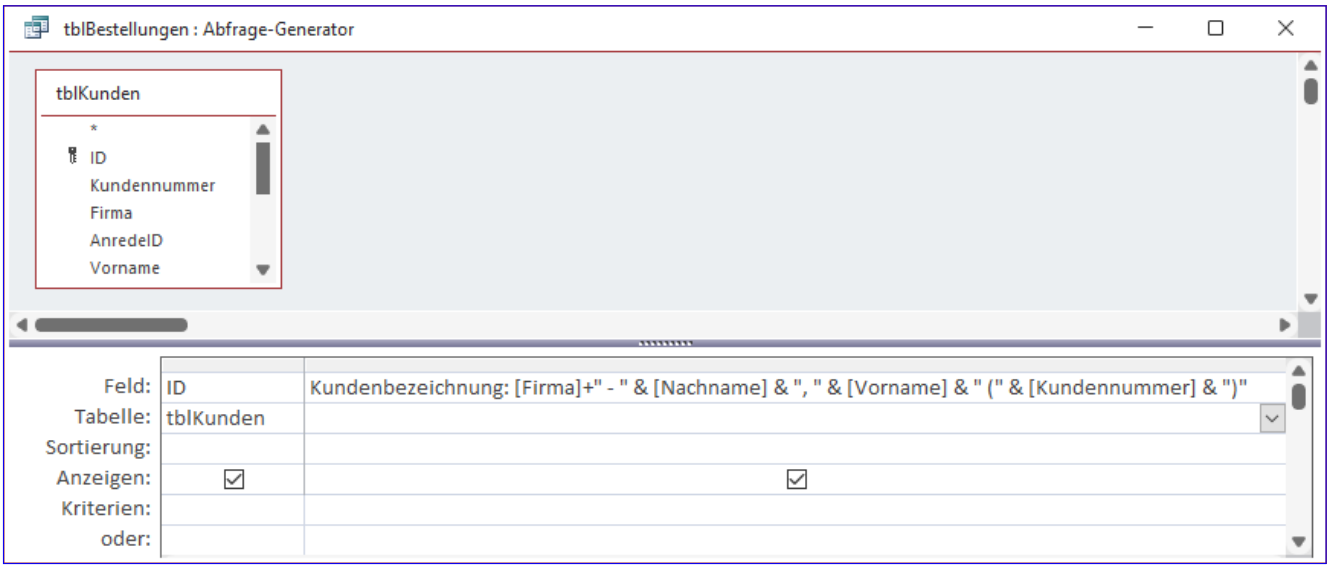


Bild 10: Datensatzherkunft für das Nachschlagefeld zur Auswahl des Kunden

gen Index fest. Auf diese Weise stellen wir sicher, dass der Benutzer jede Bestellnummer nur einmal verwenden kann.

Nachschlagefeld für den Kunden zu einer Bestellung anpassen

Den Kunden ordnen wir dabei wieder über ein Nachschlagefeld zu, mit dem wir den Kunden auswählen. Mit dem Nachschlage-Assistenten konnten wir dabei neben dem Primärschlüsselfeld der Tabelle **tblKunden** nur ein oder mehrere Felder der Tabelle auswählen, die beim Ausklappen des Nachschlagefeldes erscheinen.

Angezeigt wird aber immer nur das erste dort angegebene Feld (neben der gebundenen Spalte). Also passen wir die Abfrage, die wir mit dem Nachschlageassistenten erstellt haben, noch an, sodass diese in einem Feld gleich mehrere Informationen liefert – in diesem Fall die **Firma**, **Vorname** und **Nachname** sowie die Kundennummer.

Im ersten Anlauf haben wir damit für die Eigenschaft **Datensatzherkunft** im Bereich **Nachschlagen** der Feldeigenschaften die folgende Abfrage erstellt:

```
SELECT [tblKunden].[ID], [tblKunden].[Firma] FROM tblKunden;
```

Dieses ändern wir nun, indem wir auf die Schaltfläche mit den drei Punkten rechts von der Eigenschaft klicken. Im nun erscheinenden Abfrageentwurf bearbeiten wir das Feld der zweiten Spalte wie in Bild 10.

Was macht dieser Ausdruck genau und warum verwenden wir den **&**-Operator und den **+**-Operator gemischt? Wir wollen einen Ausdruck erzeugen, der für einen Kunden mit Firma wie folgt aussieht:

```
André Minhorst Verlag - Minhorst, André (123)
```

Für einen Kunden ohne Angabe einer Firma soll die zweite Spalte der Abfrage den folgenden Ausdruck liefern:

```
Müller, Klaus (234)
```

Wir wollen also dafür sorgen, dass wenn das Feld **Firma** den Wert **Null** enthält, auch das Minus-Zeichen zwischen **Firma** und **Nachname** wegfällt.

Wenn wir nun wissen, dass das Kaufmanns-Und (**&**) immer alle Teile einer Zeichenkverkettung zurückliefert, auch wenn einer der Teile **Null** ist, und das Plus-Zeichen (**+**) immer den Wert **Null** zurückgibt, wenn nur eines der beiden

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.

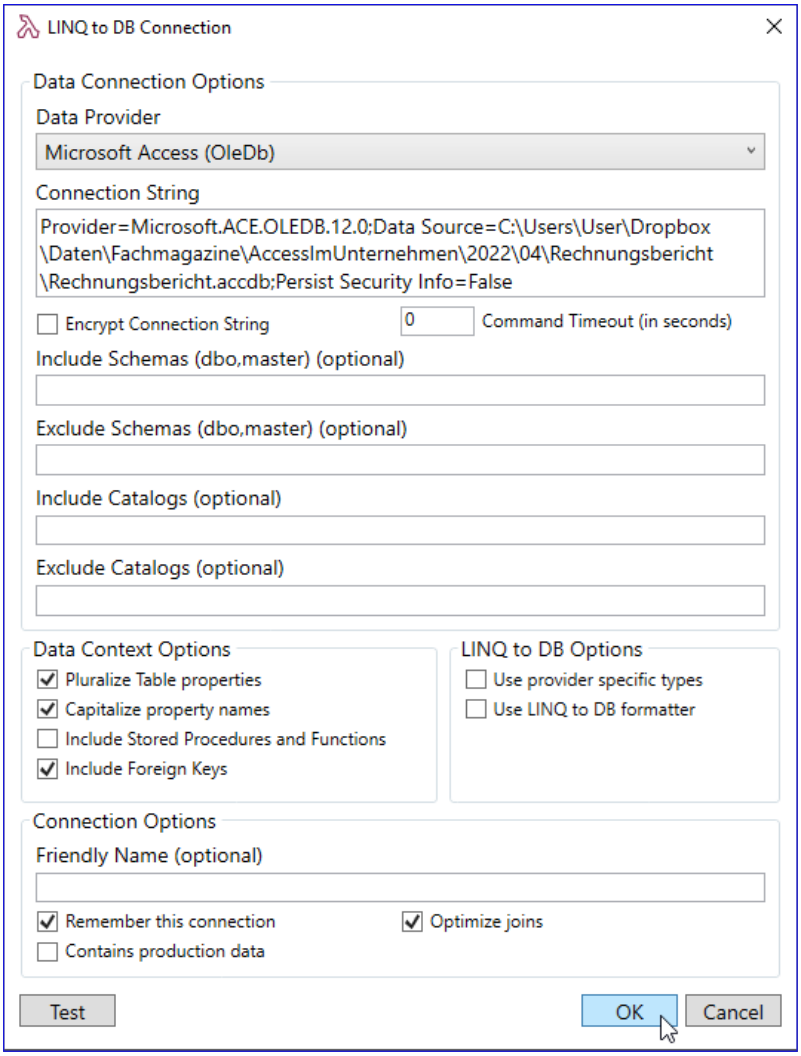


Bild 1: Verbindungseigenschaften für die Datenbank

- 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.

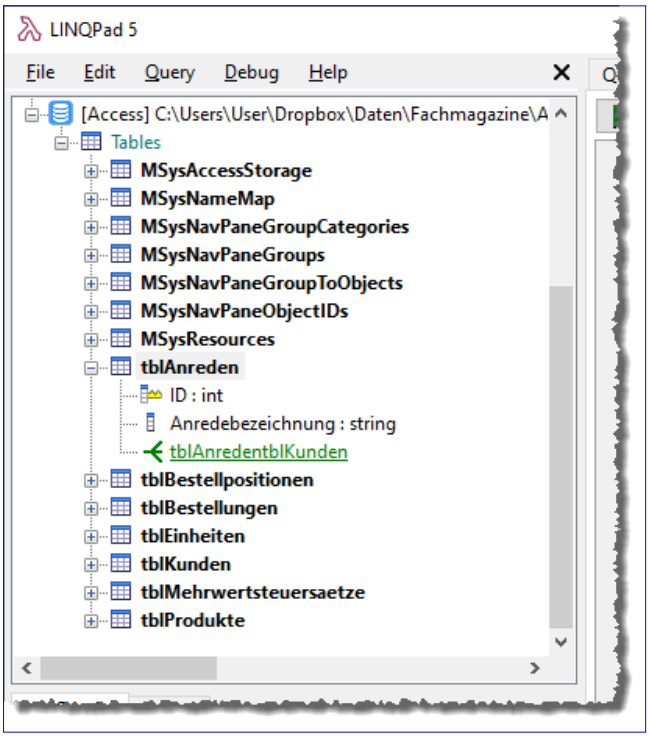


Bild 2: Die verfügbaren Tabellen

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.

```
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-

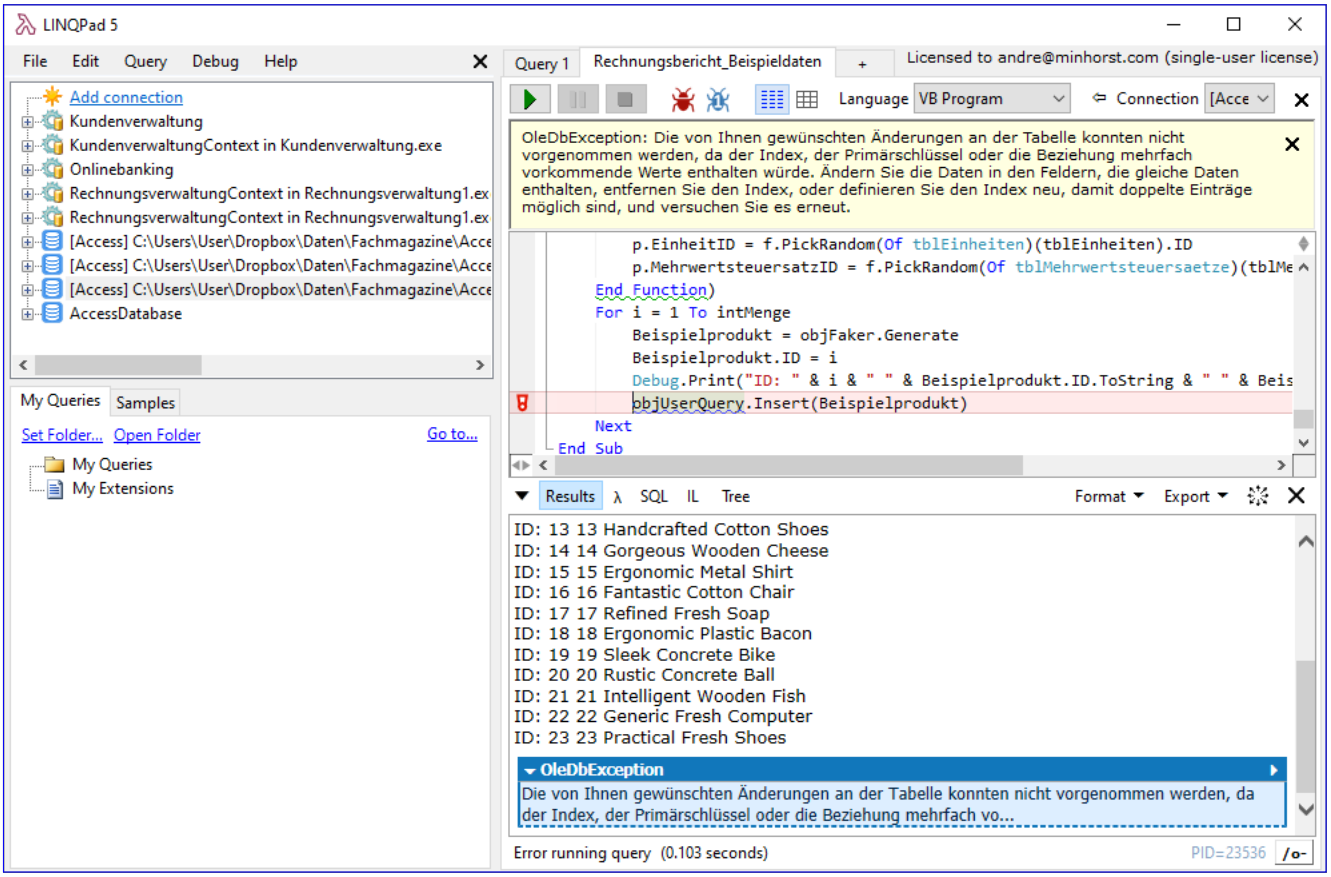


Bild 3: Fehlermeldung beim Hinzufügen von Produkten

```
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).

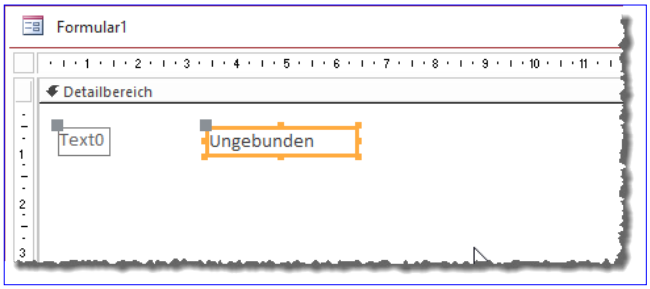


Bild 1: Ein einfaches Beschriftungsfeld

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).

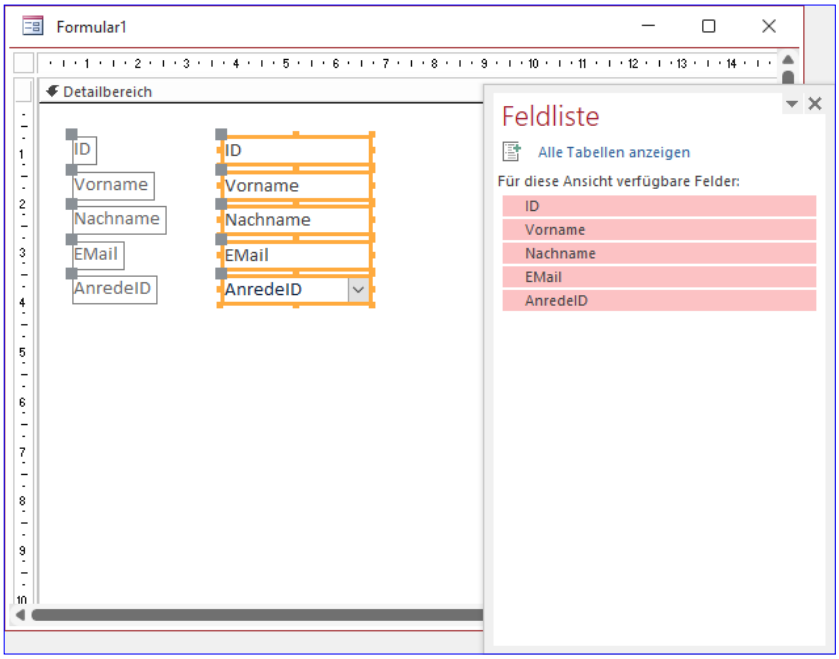


Bild 2: Gebundene Textfelder mit Beschriftungsfeldern

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.

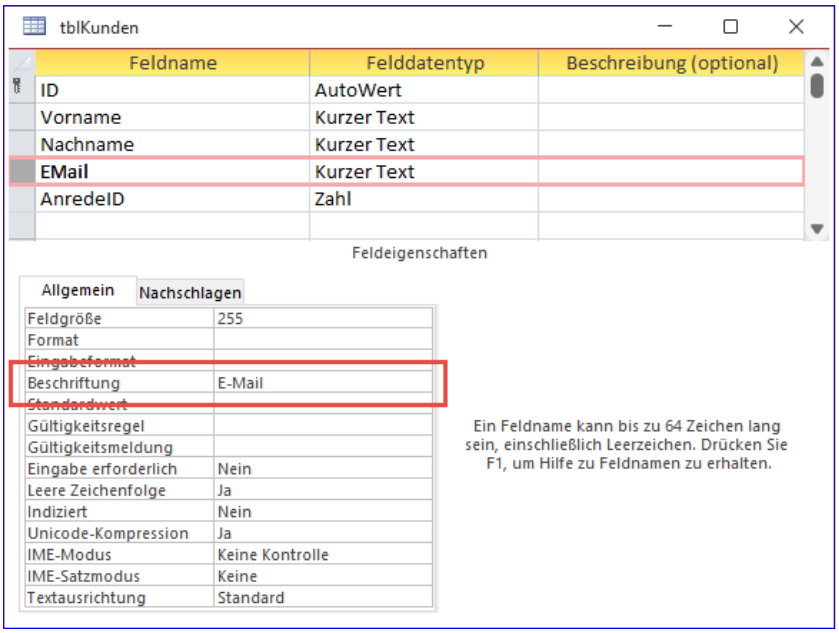


Bild 3: Zentrale Änderung der Beschriftung

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.

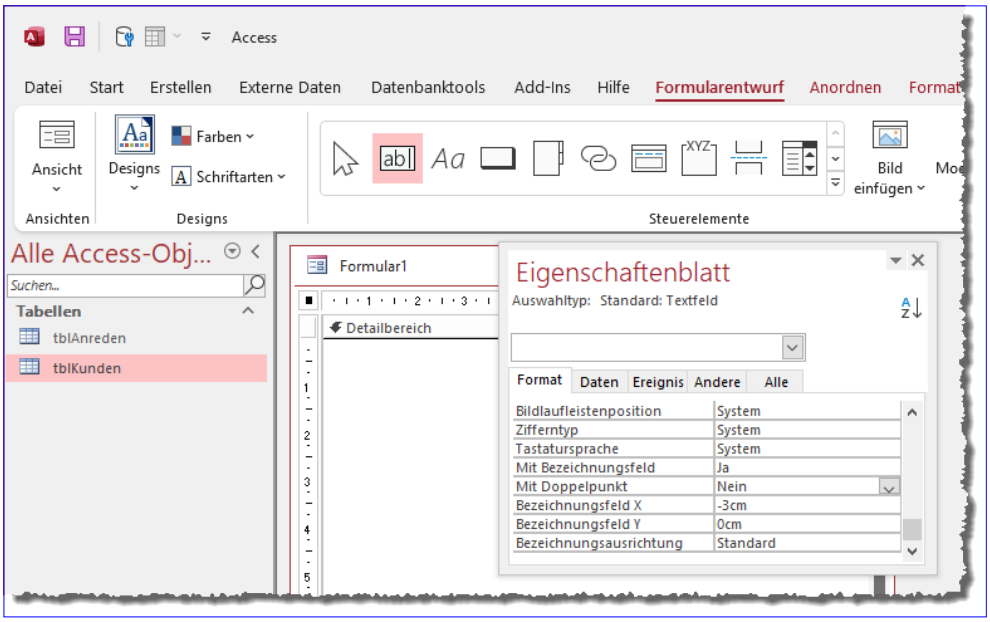


Bild 5: Einstellen der Eigenschaft **Mit Doppelpunkt**

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

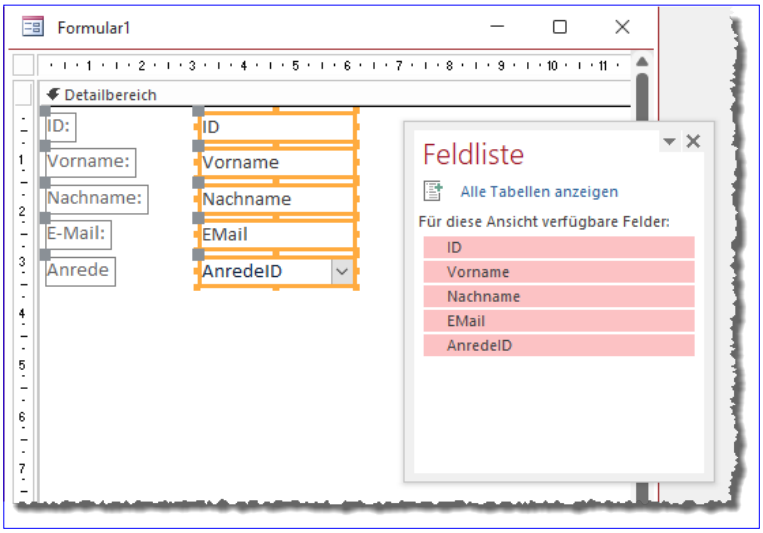


Bild 6: Alle Beschriftungsfelder außer das für das Kombinationsfeld erhalten nun einen Doppelpunkt.

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

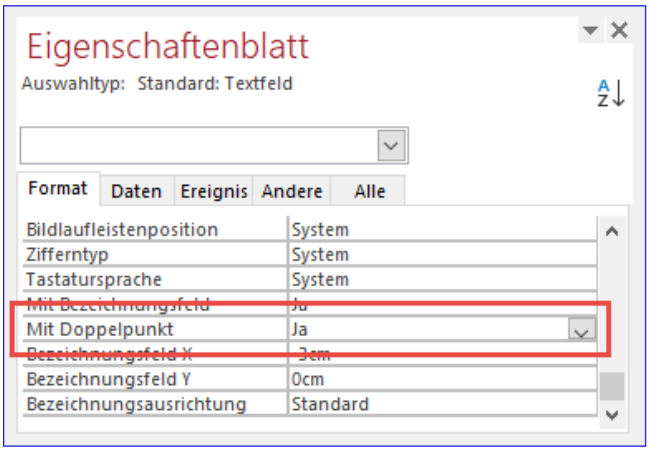


Bild 7: Ein besonderes Eigenschaftenblatt

- 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.

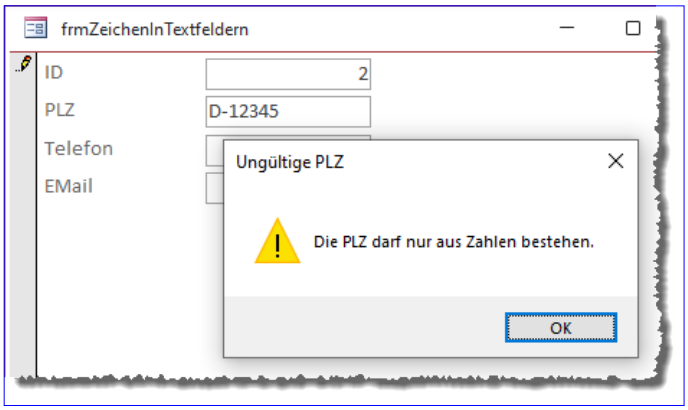


Bild 1: Meldung nach dem **Vor Aktualisierung**-Ereignis

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, _
                                    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

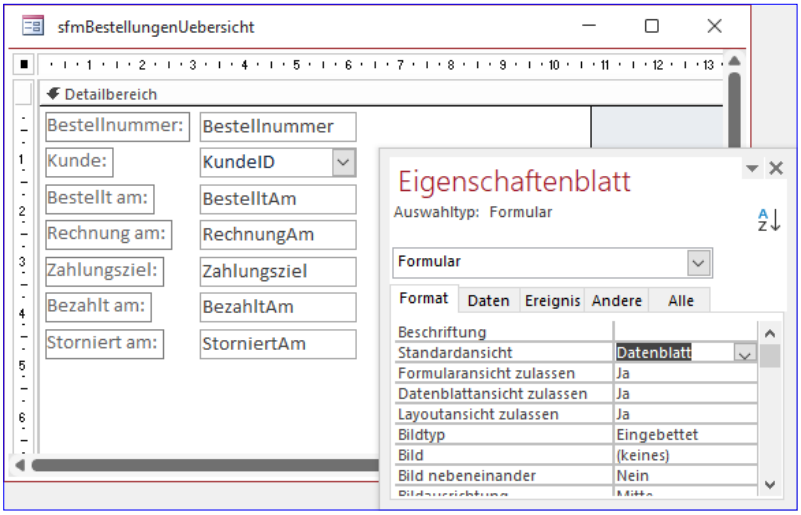


Bild 1: Das Unterformular **sfmBestellungenUebersicht**

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:

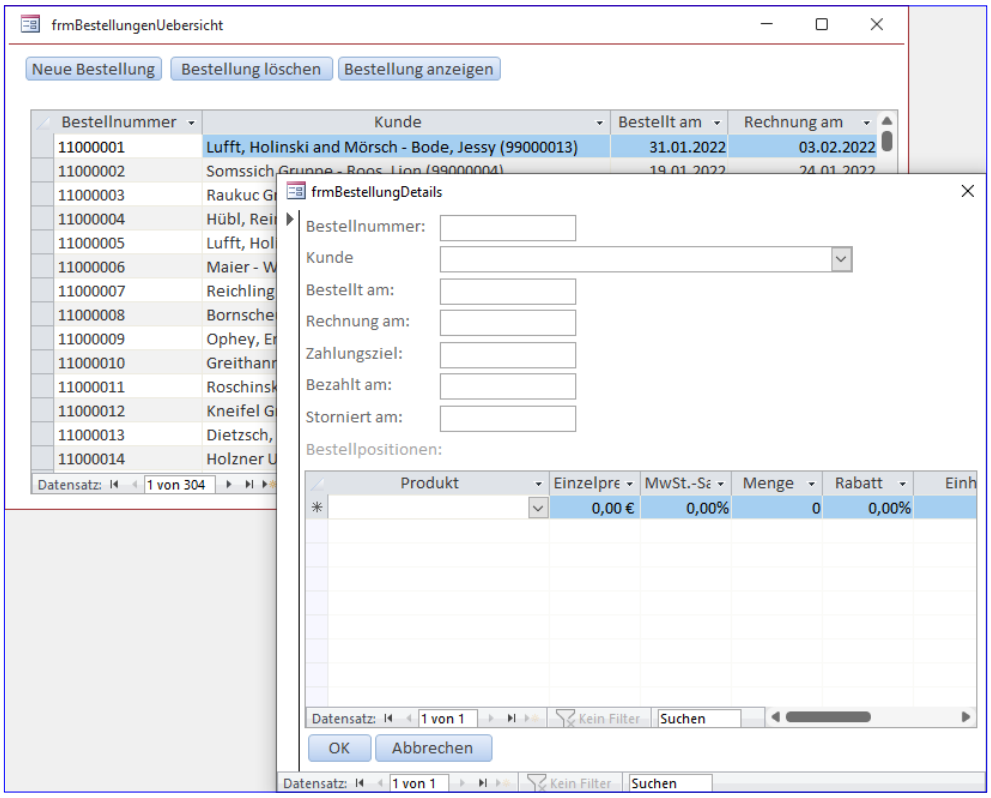


Bild 2: Anlegen einer neuen Bestellung vom Formular **frmBestellungenUebersicht** aus

```
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.

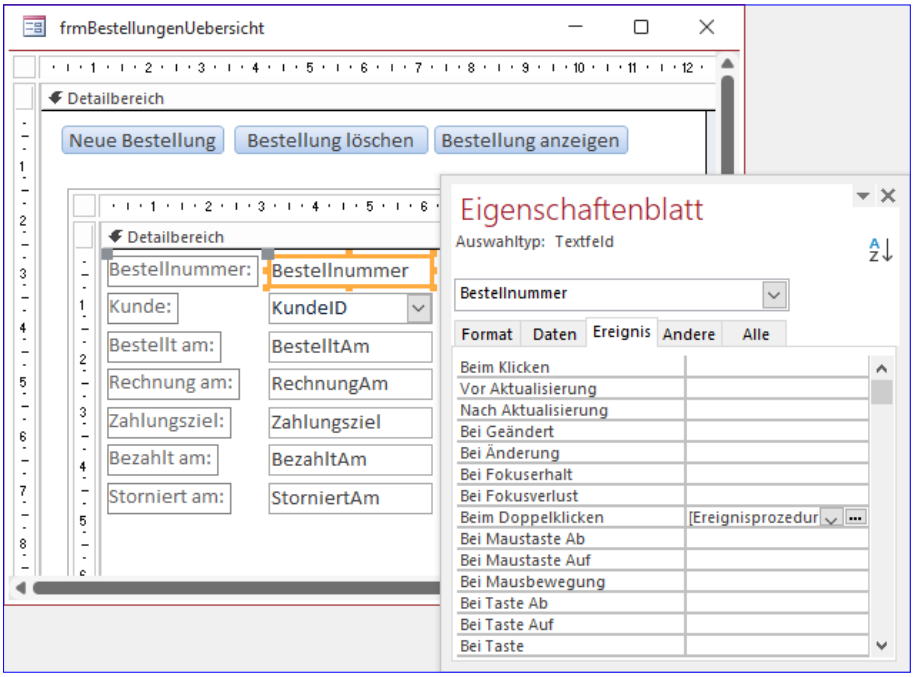


Bild 3: Anlegen einer Ereignisprozedur für den Doppelklick auf die Bestellnummer

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.

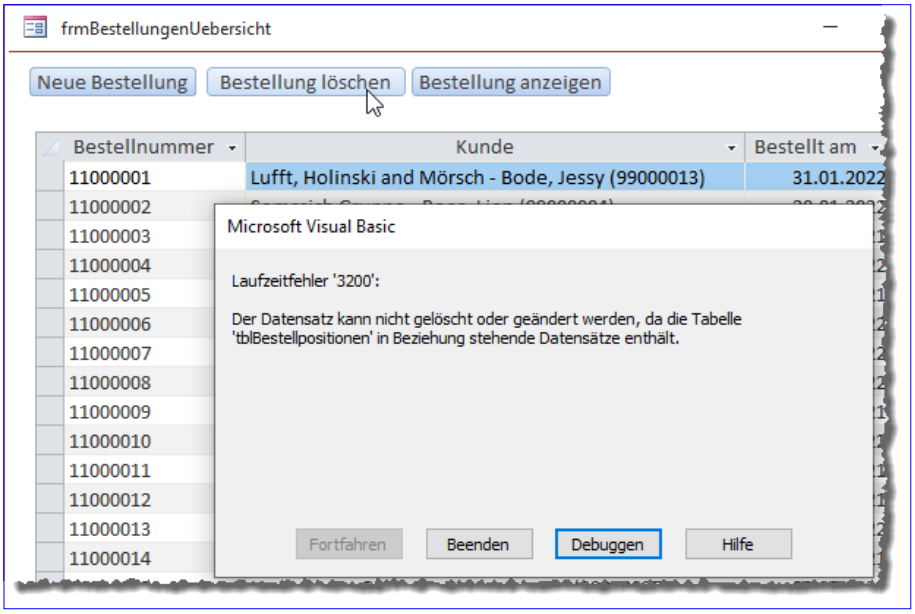


Bild 4: Fehler beim Versuch, eine Bestellung zu löschen, die bereits Bestellpositionen enthält

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**

```
Private Sub cmdBestellungLoeschen_Click()
    Dim db As DAO.Database
    Dim lngZuLoeschendeBestellungID As Long
    Set db = CurrentDb
    lngZuLoeschendeBestellungID = Me!sfmBestellungenUebersicht.Form!ID
    On Error Resume Next
    db.Execute "DELETE FROM tblBestellungen WHERE ID = " & lngZuLoeschendeBestellungID, dbFailOnError
    If Err.Number = 3200 Then
        MsgBox "Die Bestellung enthält bereits Bestellpositionen. Bitte entfernen Sie diese nach Prüfung und " _
            & "löschen Sie erst dann die Bestellung.", vbOKOnly + vbInformation, "Löschen nicht möglich"
        DoCmd.OpenForm "frmBestellungDetails", WindowMode:=acDialog, DataMode:=acFormEdit, WhereCondition:="ID = " _
            & lngZuLoeschendeBestellungID
        If IstFormularGeoeffnet("frmBestellungDetails") Then
            DoCmd.Close acForm, "frmBestellungDetails"
            Me!sfmBestellungenUebersicht.Form.Refresh
            Me!sfmBestellungenUebersicht.Form.Recordset.FindFirst "ID = " & lngZuLoeschendeBestellungID
        End If
    Else
        Me!sfmBestellungenUebersicht.Form.Requery
    End If
End Sub
```

Listing 6: Code zum Löschen einer Bestellung

zulassen und **Bearbeitungen zulassen** auf **Nein** einstellen (siehe Bild 5).

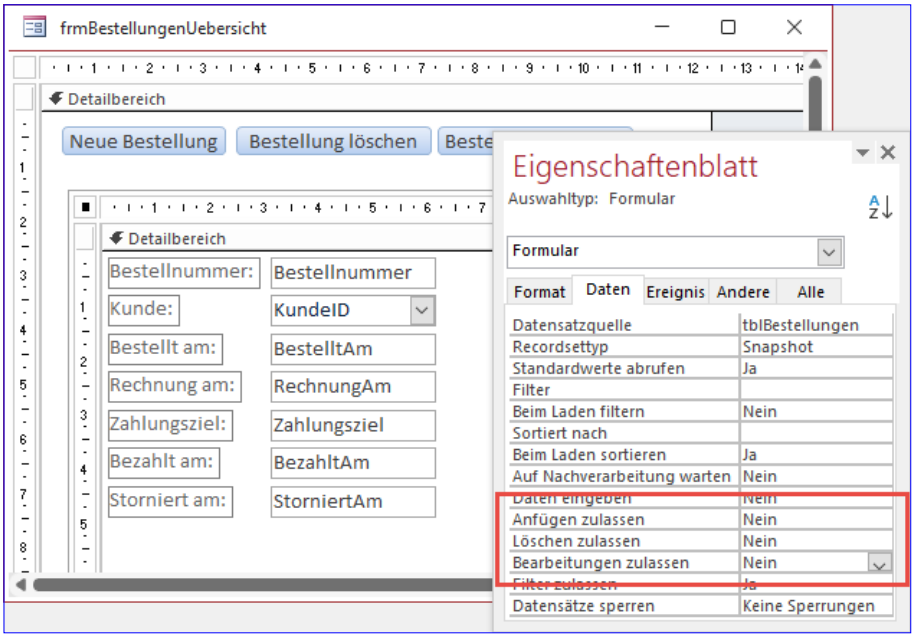


Bild 5: Sperren der Bearbeitung der Daten im Unterformular

Steuerelemente anordnen

Bevor wir eine Suchfunktion für die Bestellungen hinzufügen, machen wir uns Gedanken über die Anordnung der Steuerelemente. Wenn wir die bisher angelegten vier Schaltflächen sowie die noch anzulegenden Steuerelemente zur Eingabe von Suchkriterien über dem Unterformular platzieren, wird es dort etwas unübersichtlich.

Also platzieren wir die Schaltflächen unter dem Unterformular und legen die Steuerelemente für die Suchfunktion am aktuellen Ort der Schaltflächen an. Vergessen Sie nicht, die Eigenschaft **Vertikaler Anker** für die Schaltflächen auf **Unten** einzustellen, damit diese nach unten verschoben werden, wenn der Benutzer die Höhe des Formulars vergrößert.

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
```