HAUFE.

Ausgabe 06/2016

ACCESS

IM UNTERNEHMEN

TOOLS FÜR DEN SQL SERVER

Greifen Sie von Access aus mit unseren Werkzeugen problemlos auf die Datenbanken des SQL Servers zu (ab S. 59).

In diesem Heft:

DATEN AUS EXCEL-SHEETS EINLESEN

Lesen Sie Daten von Excel mit der TransferSpreadsheet-Methode ein.

SEITE 27

ZIPPEN MIT ACCESS

Erfahren Sie, wie Sie per VBA mit Bordmitteln Zip-Dateien erstellen und verwalten und wie Sie diese Funktionen per Formular steuern.

SEITE 14

DYNAMISCHE LISTENFELDSPALTEN

Ändern Sie die Spaltenbreite in Listenfeldern und holen Sie so noch mehr aus diesem Steuerelement heraus.

SEITE 2

ACCESS
IM UNTERNEHMEN

SQL Server-Tools

Die Programmierung von Datenbanklösungen mit Access als Frontend und SQL Server als Backend erfreut sich nach wie vor größter Beliebtheit – eine einfache Frontend-Programmierung auf der einen Seite und ein sicheres und skalierbares Datenbanksystem auf der anderen Seite sind eine gute Kombination. Noch besser wird dieses Duo, wenn Sie schnell und unkompliziert von Access aus auf die SQL Server-Daten zugreifen können. Das gelingt perfekt mit unseren Tools aus dieser Ausgabe!

Im Beitrag **SQL Server-Tools** stellen wir ab S. 59 gleich drei praktische Tools vor, mit denen Sie von Access aus auf den SQL Server zugreifen können. Das erste dient der Zusammenstellung und dem Testen von Verbindungszeichenfolgen, dem A und O der Verbindung vom Frontend zum Backend. Das zweite Tool liefert alle Tabellen einer Datenbank, auf die Sie mit einer zuvor erstellten Verbindungszeichenfolge zugreifen können. Sie können damit aus den Tabellen diejenigen auswählen, für die Sie eine Verknüpfung unter Access anlegen wollen. Damit können Sie dann auf die SQL Server-Daten zugreifen, als ob es sich um lokale Tabellen handelt. Das dritte Tool bietet die Möglichkeit, direkt SQL-Abfragen an den SQL Server zu schicken und die Ergebnisse anzusehen. Damit erhalten Sie also beispielsweise die Datensätze einer **SELECT**-Auswahlabfrage, aber Sie können damit auch Aktionsabfragen wie **INSERT INTO**, **DELETE** oder **UPDATE** durchführen. Sprich: Mit diesen drei Tools vereinfachen Sie die Programmierung von Frontends für den SQL Server.

Passendes Futter zu dieser Programmierung liefert auch der Beitrag **RDBMS per VBA: Daten abfragen**, der ab S. 41 zeigt, wie Sie direkt über eine Tabellenverknüpfung oder aber per auf dem SQL Server gespeicherter Prozedur auf die Daten einer Datenbank zugreifen und diese als Recordset nutzen.

Falls Sie einmal Daten in externe Dateien wie **.txt**-, **.csv**- oder **.xml**-Dateien exportieren und diese zum Weitersenden oder zum Hochladen auf einen Webserver in einem Zip-Archiv zusammenfassen wollen, sind Sie im Beitrag **Zippen mit Access** gut aufgehoben (ab S. 14). Hier erhalten Sie alle Informationen, um Dateien per VBA in Zip-Archive zu schreiben und diese daraus zu entpacken.

Noch einen Schritt weiter geht es unter dem Titel **Zip-Formular** (ab S. 67). Hier erfahren Sie, wie Sie die VBA-Funktionen zum Zippen in einem Formular nutzen und so ein kleines Zip-Programm in Ihre Access-Anwendung integrieren können.

Wer gelegentlich Listenfelder zur Anzeige von Datensätzen nutzt, ärgert sich vielleicht über die fehlende Möglichkeit, die Spaltenbreiten auf einfache Weise anzupassen. Wie Sie dies mit schlichten Mitteln nachrüsten können, erfahren Sie im Beitrag **Dynamische Listenfeldspalte** (ab S. 2).

Weiter mit der Optimierung Ihrer Formulare geht es im Beitrag **Datensatzvorlage per Mausklick**. Hier zeigen wir Ihnen, wie Sie schnell den aktuellen Datensatz als Vorlage für einen neuen Datensatz im gleichen Formular nutzen können (ab S. 10).

Praxis-Know-how zum Thema Excel finden Sie im Beitrag **Daten aus Excel-Sheets importieren**, wo wir die verschiedenen Spielarten des Befehls **DoCmd.TransferSpreadsheet** vorstellen (ab S. 27).

Und nun: Viel Spaß beim Lesen!

M. Forster

Ihr Michael Forster

Dynamische Listenfeldspalten

Listenfelder sind tolle Steuerelemente. Mit ihnen können Sie Daten schnell zur Auswahl bereitstellen. Allerdings haben sie einen Makel: Sie erlauben keine automatische Änderung der Spaltenbreite zur Laufzeit – zumindest nicht mit Bordmitteln. Dieser Beitrag zeigt, wie Sie in Zusammenhang mit den Verankern-Eigenschaften von Formularen zumindest ein klein wenig Flexibilität aus den ansonsten recht statischen Listenfelder herausholen.

Beispiel Kundenliste

Als Beispiel für diesen Beitrag verwenden wir die Tabelle **tblKunden** der Suedsturm-Datenbank. Die Daten dieser Tabelle sollen in einem Listenfeld namens **lstKunden** angezeigt werden. Daher stellen wir die Datensatzherkunft dieses Steuerelements auf eine Abfrage ein, die wie in Bild 1 aussieht.

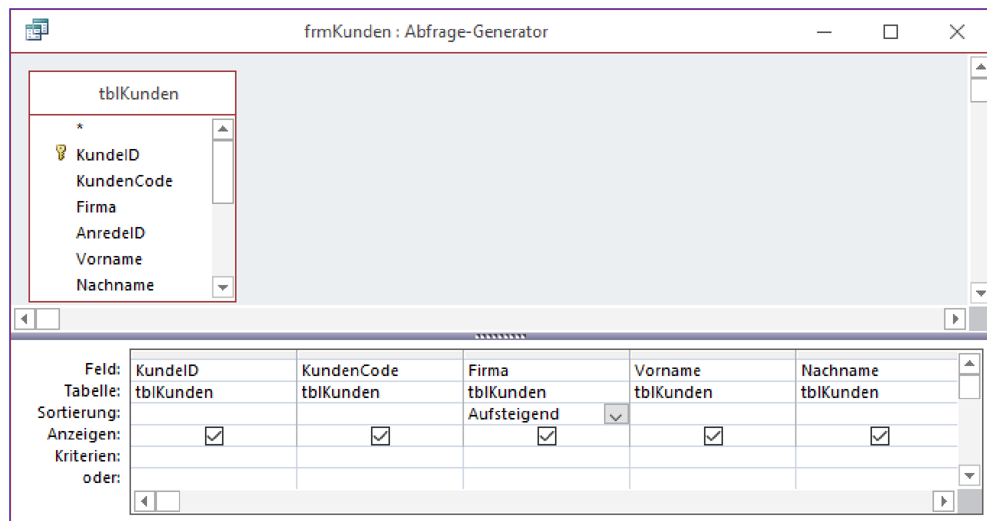

Bild 1: Datensatzherkunft des Listenfeldes **lstKunden**

Diese Abfrage liefert die Felder **KundeID**, **KundenCode**, **Firma**, **Vorname** und **Nachname** der Tabelle **tblKunden** sortiert nach dem Inhalt des Feldes **Firma**. Die Eigenschaften **Datensatzmarkierer**, **Navigationsschaltflächen**, **Trennlinien** und **Bildlaufleisten** dieses Formulars stellen wir auf **Nein** ein. Das Formular sieht in der Entwurfsansicht wie in Bild 2 aus. Es enthält lediglich das Listenfeld **lstKunden** als Steuerelement.

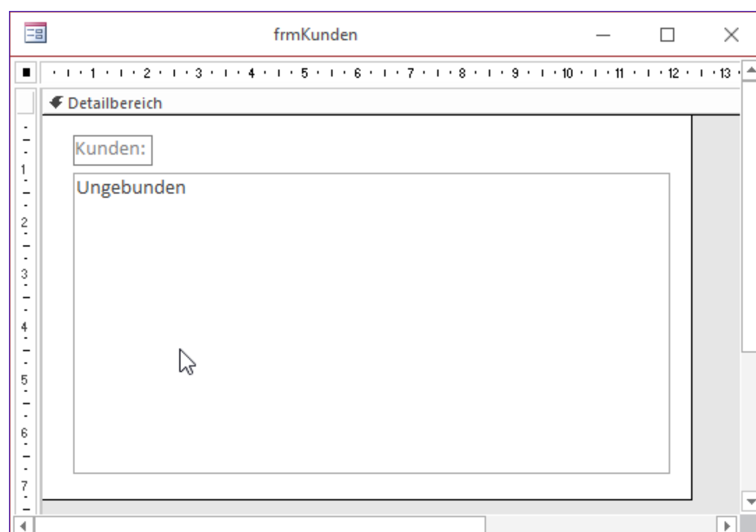

Bild 2: Das Formular **frmKunden** mit dem Listenfeld zur Anzeige der Kunden in der Entwurfsansicht

Das Listenfeld soll alle fünf Felder der als Datensatzherkunft verwendeten Abfrage nutzen, also legen wir für die Eigenschaft **Spaltenanzahl** den Wert **5** fest.

Die Eigenschaft **Spaltenbreiten** stellen wir auf den Wert **0cm;2cm;5cm;2cm;4cm** ein. Damit legen wir fest, dass das erste Feld der Datensatzherkunft, also **KundeID**, nicht im Listenfeld

angezeigt wird. Die übrigen Felder erscheinen in der jeweils angegebenen Breite, wie Bild 3 zeigt.

Dies ist allerdings noch längst nicht die Ansicht, die wir uns wünschen. In der Abbildung haben wir die Größe des Formulars bereits geändert, was sich aber aktuell noch nicht auf die Größe des Listenfeldes auswirkt. Dieses sollte sich eigentlich analog zum Rest des Formulars vergrößern. Damit dies im nächsten Anlauf gelingt, stellen wir die Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** auf den Wert **Beide** ein. Außerdem korrigieren Sie den dabei automatisch geänderten Wert dieser beiden Eigenschaften für das Bezeichnungsfeld des Listenfeldes wieder auf **Links** und **Oben**.

Nun passt es: Das Listenfeld wird automatisch vergrößert, wenn man das Formular anpasst (s. Bild 4). Leider verändert sich innerhalb des Listenfeldes aber nur die Breite der letzten Spalte. Schön wäre es gewesen, wenn sich die Spalte mit der Firma automatisch entsprechend der Breite des Formulars angepasst hätte, damit ihre Inhalte komplett sichtbar sind.

Flexible Spalte – aber nur einmal

Damit kommen wir zum ersten notwendigen Schritt, wenn Sie dafür sorgen wollen, dass genau eine Spalte immer mit dem Formular vergrößert oder verkleinert wird. In diesem stellen wir die Eigenschaft **Spaltenbreiten** auf einen neuen Wert ein, der wie folgt lautet (s. Bild 5):

```
0cm;2cm;;2cm;4cm
```

Was ist wichtig an dieser Stelle? Dass Sie für die Spalte, deren Breite flexibel eingestellt werden soll, keinen numerischen Wert angeben, sondern den Platz zwischen den entsprechenden Semikola (;) einfach freilassen.

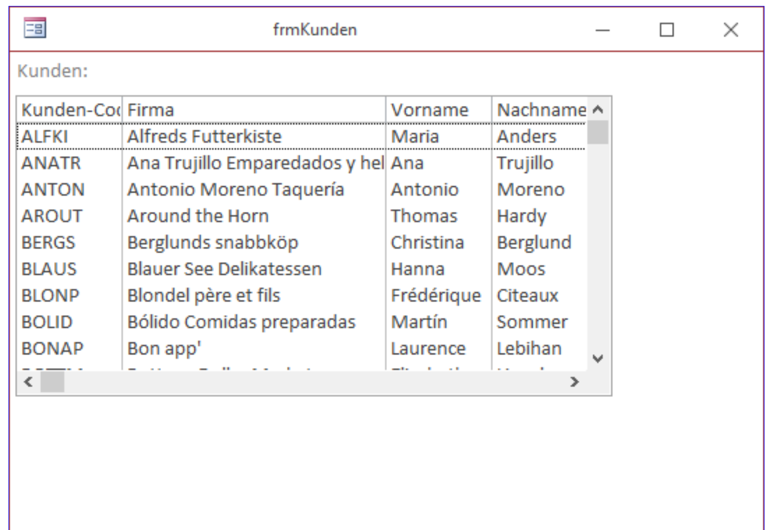

Bild 3: Die Formularansicht entspricht noch nicht unseren Wünschen.

Bild 4: Nun wird zumindest das Listenfeld mit dem Formular vergrößert.

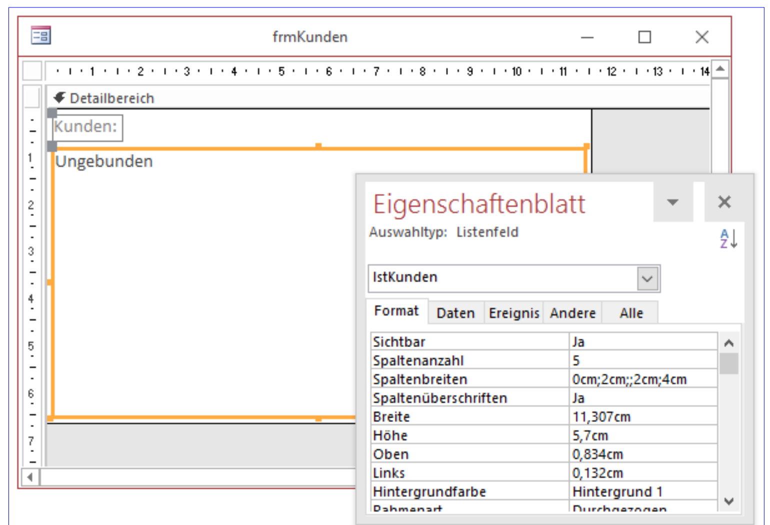

Bild 5: Einstellen der Spaltenbreiten für unseren Anwendungsfall

frmKunden

Kunden:

Kunden-Co	Firma	Vorname	Nachname
ALFKI	Alfreds Futterkiste	Maria	Anders
ANATR	Ana Trujillo Emparedados y helados	Ana	Trujillo
ANTON	Antonio Moreno Taquería	Antonio	Moreno
AROUT	Around the Horn	Thomas	Hardy
BERGS	Berglunds snabbköp	Christina	Berglund
BLAUS	Blauer See Delikatessen	Hanna	Moos
BLONP	Blondel père et fils	Frédérique	Citeaux
BOLID	Bólido Comidas preparadas	Martín	Sommer
BONAP	Bon app'	Laurence	Lebihan
BOTTM	Bottom-Dollar Markets	Elizabeth	Lincoln
BSBEV	B's Beverages	Victoria	Ashworth
CACTU	Cactus Comidas para llevar	Patricio	Simpson
CENTC	Centro comercial Moctezuma	Francisco	Chang
CHOPS	Chop-suey Chinese	Yang	Wang
COMMI	Comércio Mineiro	Pedro	Afonso

Bild 6: Die Spalte mit der Firma wird auf die enthaltenen Daten optimiert.

frmKunden

Kunden:

Kunden-Co	Firma	Vorname	Nachname
ALFKI	Alfreds Futterkiste	Maria	Anders
ANATR	Ana Trujillo Emparec	Ana	Trujillo
ANTON	Antonio Moreno Ta	Antonio	Moreno
AROUT	Around the Horn	Thomas	Hardy
BERGS	Berglunds snabbköp	Christina	Berglund
BLAUS	Blauer See Delikates	Hanna	Moos
BLONP	Blondel père et fils	Frédérique	Citeaux
BOLID	Bólido Comidas prep	Martín	Sommer
BONAP	Bon app'	Laurence	Lebihan
BOTTM	Bottom-Dollar Mark	Elizabeth	Lincoln

Bild 7: Beim erneuten Öffnen bekommt die Spalte **Firma** nun leider noch weniger Platz ab.

Nun erhalten Sie die gewünschte Ansicht (s. Bild 6), aber nur unter einer Voraussetzung: Sie hatten das Formular zuvor geöffnet und bereits einmal auf eine gewisse Breite vergrößert und sind dann wieder zur Entwurfsansicht gewechselt.

Wenn Sie es dann erneut in der Formularansicht öffnen, wird die Spalte **Firma** so breit angezeigt, dass es genau den Rest der Breite des Listenfeldes minus der Spaltenbreite der übrigen Felder einnimmt.

Wenn Sie das Formular nun komplett schließen und wieder öffnen, erhalten Sie nicht nur den alten Zustand – es ist noch übler: Das Feld **Firma** erhält nun noch weniger Platz, nämlich wiederum die Gesamtbreite des Formulars minus der für die übrigen Spalten fix eingestellten Breiten (s. Bild 7).

Dauerhafte flexible Spalte

Nun kommt Schritt zwei der notwendigen Änderungen für eine flexible Spaltenbreite im Listenfeld: Wir fügen dem Formular eine Ereignisprozedur für das Ereignis **Bei Größenänderung** hinzu, welche die folgende Anweisung ausführt:

```
Private Sub Form_Resize()
    Me!lstKunden.ColumnWidths = Me!lstKunden.ColumnWidths
End Sub
```

Das Ergebnis überzeugt: Von nun an ändert sich die Breite der Spalte **Firma** immer, sobald wir am rechten Rand des Formulars ziehen und so die Breite des Listenfeldes ändern (s. Bild 8).

Flexible Spalte selbst festlegen

Wir gehen noch einen Schritt weiter: Während bisher die flexible Spalte fest durch den Eintrag in die Eigenschaft **Spaltenbreiten** vorgegeben war, wollen wir diesen nun durch den Benutzer anpassen lassen. Dazu soll dieser einfach die entsprechende Spalte anklicken können und dann die Breite dieser Spalte durch Anpassen der Formu-

frmKunden

Kunden-Co	Firma	Vorname	Nachname
ALFKI	Alfreds Futterkiste	Maria	Anders
ANATR	Ana Trujillo Emparedados y helados	Ana	Trujillo
ANTON	Antonio Moreno Taquería	Antonio	Moreno
AROUT	Around the Horn	Thomas	Hardy
BERGS	Berglunds snabbköp	Christina	Berglund
BLAUS	Blauer See Delikatessen	Hanna	Moos
BLONP	Blondel père et fils	Frédérique	Citeaux
BOLID	Bólido Comidas preparadas	Martín	Sommer
BONAP	Bon app'	Laurence	Lebihan
BOTTM	Bottom-Dollar Markets	Elizabeth	Lincoln

Bild 8: Mit einer kleinen Ereignisprozedur passt sich die Spaltenbreite des Feldes **Firma** immer der Listenfeldbreite minus den übrigen fixen Spaltenbreiten an.

larbreite vergrößern oder verkleinern. Um zu überprüfen, welche Spalte aktuell die flexible Spaltenbreite aufweist, zeigen wir ihren Namen in einem Textfeld namens **txtDynamischeSpalte** an (die neue Version des Formulars samt Textfeld finden Sie in Bild 9).

Was nun folgt, ist das Ergebnis einiger Experimente und der Verwendung des **Maustaste auf**-Ereignisses. Vorab die Idee, die hinter dieser Prozedur steckt: Aktuell ist genau eine Spalte durch Änderung der Breite des Formulars anpassbar.

Dies geschieht erstens dadurch, dass Veränderungen der Formularbreite durch die Einstellung des Wertes **Beide** für das Steuerelement **lstKunden** die Listenfeldbreite verändert. Die oben vorgestellte Ereignisprozedur, die durch das Ereignis **Bei Größenänderung** ausgelöst wird, sorgt dann dafür, dass die Eigenschaft **Spaltenbreiten** des Listenfeldes bei jeder Änderung der Formularbreite aktualisiert wird. So wird immer die Spalte, für die keine feste Breite festgelegt wurde, so angepasst, dass sie der Breite des Listenfeldes minus der Breite der übrigen Spalten entspricht.

Nun wollen wir dafür sorgen, dass der Benutzer durch einen einfachen Mausklick auf eine der Spalten die angeklickte Spalte zur flexibel einstellbaren Spalte machen kann. Das heißt: Zu Beginn ist beispielsweise die Spalte **Firma** flexibel einstellbar. Dann klickt der Benutzer auf die Spalte **Vorname** und ändert dann die Formularbreite. Nun ändert sich nicht mehr automatisch die Breite der Spalte **Firma**, sondern die der Spalte **Vorname**! Dazu sind einige Verrenkungen nötig, aber wenn man länger mit Access arbeitet, weiß man, dass man fast alles erreichen kann, wenn man nur ein wenig Zeit und Hirnschmalz investiert.

Der Plan sieht so aus: Der Benutzer klickt auf die Spalte, die er zur flexibeln Spalte machen möchte. Dadurch löst er das Ereignis **Bei Maustaste auf** aus. Dieses Ereignis liefert mit seinen Parametern beispielsweise den Wert der X-Koordinate der Position, die der Benutzer angeklickt hat:

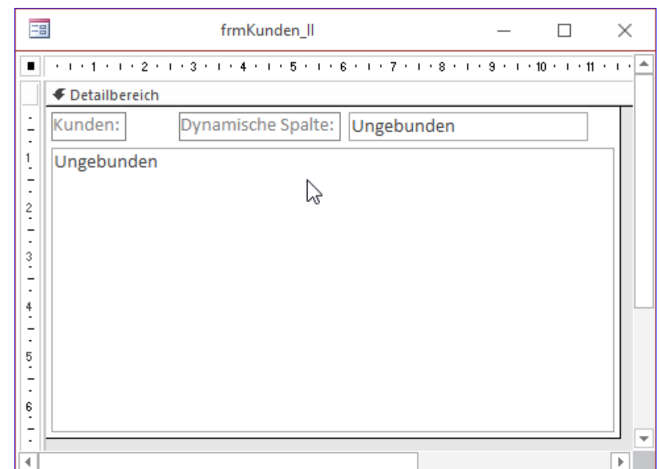

Bild 9: Neues Formular, neues Textfeld zur Anzeige der aktuell veränderbaren Spalte

```
Private Sub lstKunden_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
```

Damit wollen wir nun alle Spalten durchlaufen und deren Start-X-Koordinate und End-X-Koordinate ermitteln. Wenn wir eine Spalte finden, für die der Wert des Parameters **X** der angeklickten Position zwischen Start- und Endkoordinate der Spalte liegt, haben wir bereits die angeklickte Spalte entdeckt.

Nun müssen wir nur noch den Wert der Eigenschaft **Spaltenbreiten** (unter VBA: **ColumnWidths**) so ändern, dass nicht mehr die Spalte **Firma** als flexible Spalte markiert ist (hier etwa mit **0cm;2cm;;2cm;2cm**), sondern nun die Spalte **Vorname**, also etwa mit dem Wert **0cm;2cm;5cm;;2cm** für die Eigenschaft **ColumnWidths**. Dabei müssen wir natürlich zuvor ermitteln, wie breit die bis dahin flexibel anpassbare Spalte **Firma** gewesen ist. Diesen Wert tragen wir dann für **ColumnWidths** dort ein, wo sich zuvor ein Loch zwischen zwei Semikola befand, und entfernen die fixe Spaltenbreite für die nun als flexibel festgelegte Spalte.

Dazu verwenden wir ein paar Hilfsfunktionen, denn ein paar der gesuchten Werte sind nicht allzu einfach zu ermitteln – zum Beispiel können Sie zwar immer mit der **Width**-Eigenschaft die Breite des Listenfeld-Steuerele-

Datensatzvorlage per Mausklick

Im Beitrag »Dynamische Standardwerte« haben wir uns angesehen, wie Sie bestimmte Werte als Standardwerte für neue Datensätze nutzen können. Dabei haben wir immer die zuletzt angegebenen Werte in einer Tabelle gespeichert und diese beim Anlegen eines neuen Datensatzes als Standardwerte vorgegeben. Etwas mehr Flexibilität erhalten Sie noch, wenn Sie selbst festlegen können, welche Daten als Standardwerte für folgende Datensätze genutzt werden sollen. Sprich: Uns fehlt noch die Möglichkeit, einen Wert eines beliebigen Datensatzes als Standardwerts zu nutzen. Außerdem wollen wir noch komplette Datensätze als Vorlage für neue Datensätze nutzen können. Wie dies gelingt, erfahren Sie im vorliegenden Beitrag.

Kompletten Datensatz als Vorlage speichern

Im Beitrag **Dynamische Standardwerte** (**www.access-im-unternehmen.de/1052**) haben wir ein Formular mit einer Funktion ausgestattet, die immer die Daten des zuletzt geänderten oder neu angelegten Datensatzes als Standardwerte zu speichern (siehe auch das Formular **frmArtikel_Standardwerttabelle** in der Beispieldatenbank).

Nun wollen wir in einem weiteren Formular zeigen, wie Sie es dem Benutzer überlassen können, den Datensatz festzulegen, dessen Werte als Standardwerte gespeichert werden sollen. Dazu fügen wir einer Kopie des Formulars **frmArtikel_Standardwerttabelle**, das wir hier unter dem Namen **frmArtikel_Standarddatensatz** gespeichert haben, eine Schaltfläche namens **cmdDatensatzAlsStandard** mit der Beschriftung wie in Bild 1 hinzu.

Dieser Schaltfläche hinterlegen wir für die Ereigniseigenschaft **Beim Klicken** die Ereignisprozedur aus Listing 1. Diese Prozedur ruft für jedes Steuerelement, dessen Standardwert wir speichern möchten (also alle außer dem Primärschlüsselfeld), je einmal die Prozedur **StandardwertSpeichern** auf, die wir im oben genannten Artikel vorgestellt haben. Dabei übergibt sie einen Verweis auf das aufrufende Formular, den Namen des Steuerelements sowie einen Verweis auf den aktuellen Inhalt des Steuerelements an die Prozedur.

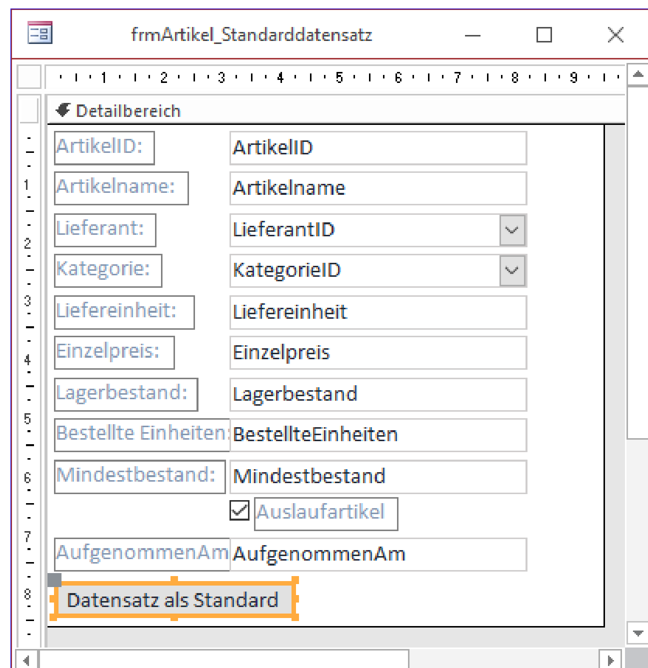

Bild 1: Schaltfläche zum Speichern des Datensatzes als Standard

Wenn Sie nun auf die Schaltfläche klicken, werden die Daten des aktuellen Datensatzes in der Tabelle **tblStandardwerte** gespeichert (s. Bild 2).

Nun benötigen wir noch einen Mechanismus, der diesen Datensatz beim Anzeigen eines neuen Datensatzes automatisch als Standardwert vorgibt. Dazu brauchen wir das Rad nicht neu zu erfinden – die entsprechenden Ereignisprozeduren werden durch die Ereignisse **Beim Anzeigen**

```
Private Sub cmdDatensatzAlsStandard_Click()
    StandardwertSpeichern Me.Name, "txtArtikelname", Nz(Me!txtArtikelname)
    StandardwertSpeichern Me.Name, "cboKategorieID", Nz(Me!cboKategorieID)
    StandardwertSpeichern Me.Name, "cboLieferantID", Nz(Me!cboLieferantID)
    StandardwertSpeichern Me.Name, "txtLiefereinheit", Nz(Me!txtLiefereinheit)
    StandardwertSpeichern Me.Name, "txtEinzelpreis", Nz(Me!txtEinzelpreis)
    StandardwertSpeichern Me.Name, "txtLagerbestand", Nz(Me!txtLagerbestand)
    StandardwertSpeichern Me.Name, "txtBestellteEinheiten", Nz(Me!txtBestellteEinheiten)
    StandardwertSpeichern Me.Name, "txtMindestbestand", Nz(Me!txtMindestbestand)
    StandardwertSpeichern Me.Name, "chkAuslaufartikel", Nz(Me!chkAuslaufartikel)
    StandardwertSpeichern Me.Name, "txtAufgenommenAm", Nz(Me!txtAufgenommenAm)
End Sub
```

Listing 1: Diese Prozedur wird beim Anklicken der Schaltfläche **Datensatz als Standard** ausgelöst.

tblStandardwerte

StandardwertID	Formularname	Steuerelementname	Standardwert
185	frmArtikel_Standarddatensatz	cboKategorieID	2
186	frmArtikel_Standarddatensatz	cboLieferantID	3
192	frmArtikel_Standarddatensatz	chkAuslaufartikel	0
184	frmArtikel_Standarddatensatz	txtArtikelname	Northwoods Cranberry
193	frmArtikel_Standarddatensatz	txtAufgenommenAm	
190	frmArtikel_Standarddatensatz	txtBestellteEinheiten	0
188	frmArtikel_Standarddatensatz	txtEinzelpreis	20
189	frmArtikel_Standarddatensatz	txtLagerbestand	6
187	frmArtikel_Standarddatensatz	txtLiefereinheit	12 x 12-oz-Gläser
191	frmArtikel_Standarddatensatz	txtMindestbestand	0
(Neu)			

Datensatz: 1 von 10 Gefiltert Suchen

Bild 2: Kompletter gespeicherter Datensatz in der Tabelle **tblStandardwerte**

und **Beim Entladen** des Formulars ausgelöst und wurden bereits im oben genannten Beitrag vorgestellt. Die Prozedur **Form_Current** ruft im Falle eines neuen Datensatzes die Prozedur **StandardwerteSetzen** auf und übergibt eine Referenz auf das aktuelle Formular. Die Prozedur **Form_Unload** wird beim Entladen des Formulars ausgelöst und führt ihre Anweisungen ebenfalls nur aus, wenn aktuell ein neuer Datensatz angezeigt wird. Sie fragt dann, ob der neu angelegte Datensatz, der bis dahin noch nicht bearbeitet wurde, gespeichert oder verworfen werden soll. Dies nur zur Sicherheit, falls der Benutzer glaubt, der ausschließlich mit Standardwerten (und somit noch nicht in der Tabelle gespeicherte) gefüllte Datensatz würde automatisch gespeichert.

Standardwerte per Schleife

Bislang ist es noch etwas mühselig, immer alle als Standardwert zu verwendenden Felder wie in den beiden Beispielen des oben genannten Beitrags entweder per Code einzeln zu speichern oder diese zuerst mit einer Marke zu versehen. Zumal man ja vielleicht ohnehin meist eher alle Felder außer dem Primärschlüsselfeld als Standardwerte vorgeben möchte.

Also haben wir noch ein weiteres Formular namens **frmArtikel_StandardwertePerSchleife** angelegt, in dem wir beim Anklicken der Schaltfläche **cmdDatensatzAlsStandard** eine Prozedur aufrufen, die automatisch alle Steuerelemente durchläuft und für alle einen Standardwert in der Tabelle **tblStandardwerte** hinterlegt, die nicht Primärschlüsselwert der zugrunde liegenden Datenherkunft sind.

Hier legen wir für die Ereignisprozedur, die durch das Anklicken der Schaltfläche **cmdDatensatzAlsStandard** ausgelöst wird, die folgende Anweisung an:

```
Private Sub cmdDatensatzAlsStandard_Click()
    StandardwerteSpeichernAusserPK Me
End Sub
```

Die dort aufgerufene Prozedur **StandardwerteSpeichernAusserPK** finden Sie in Listing 2. Die Prozedur erwartet wieder einen Verweis auf das zu untersuchende Formular als Parameter. Die Prozedur speichert einen Verweis auf das in der **Recordset**-Eigenschaft des Formulars enthaltene Objekt in der Variablen **rst**. Dann durchläuft es alle Steuerelemente der **Controls**-Auflistung des Formulars und schreibt das aktuelle Steuerelement jeweils in die Variable **ctl**. Nun fügt sie der Variablen **strControlsource** eine leere Zeichenkette hinzu. Warum das? Weil die

Zippen mit Access

Sie werden immer mal wieder auf die Aufgabe stoßen, automatisiert Zip-Dateien zu erstellen, Daten in Zip-Dateien zu speichern oder Daten aus Zip-Dateien zu extrahieren. Dazu benötigen Sie optimalerweise VBA-Code ohne viel Schnickschnack wie externe Bibliotheken et cetera. Windows liefert glücklicherweise in aktuelleren Versionen Möglichkeiten, Zip-Dateien auch per VBA zu erstellen, zu füllen und zu lesen. Diese sind zwar nicht so einfach zu finden, aber wir haben Ihnen eine Auswahl wichtiger Funktionen zusammengestellt.

Grundlegende Technik

Wenn wir die Windows-Funktionen zum Verwenden von Zip-Dateien nutzen wollen, haben wir nur eine eingeschränkte Menge an Möglichkeiten zur Verfügung. Diese sollten jedoch für den Großteil der denkbaren Einsatzfälle ausreichend sein.

Benötigen Sie spezielle Techniken etwa zum Hinzufügen eines Kennworts zum Schutz des Inhalts der Zip-Datei, müssen Sie auf fertige Bibliotheken zurückgreifen, die möglicherweise kostenpflichtig sind und gegebenenfalls auch erst beim Nutzer installiert werden müssen.

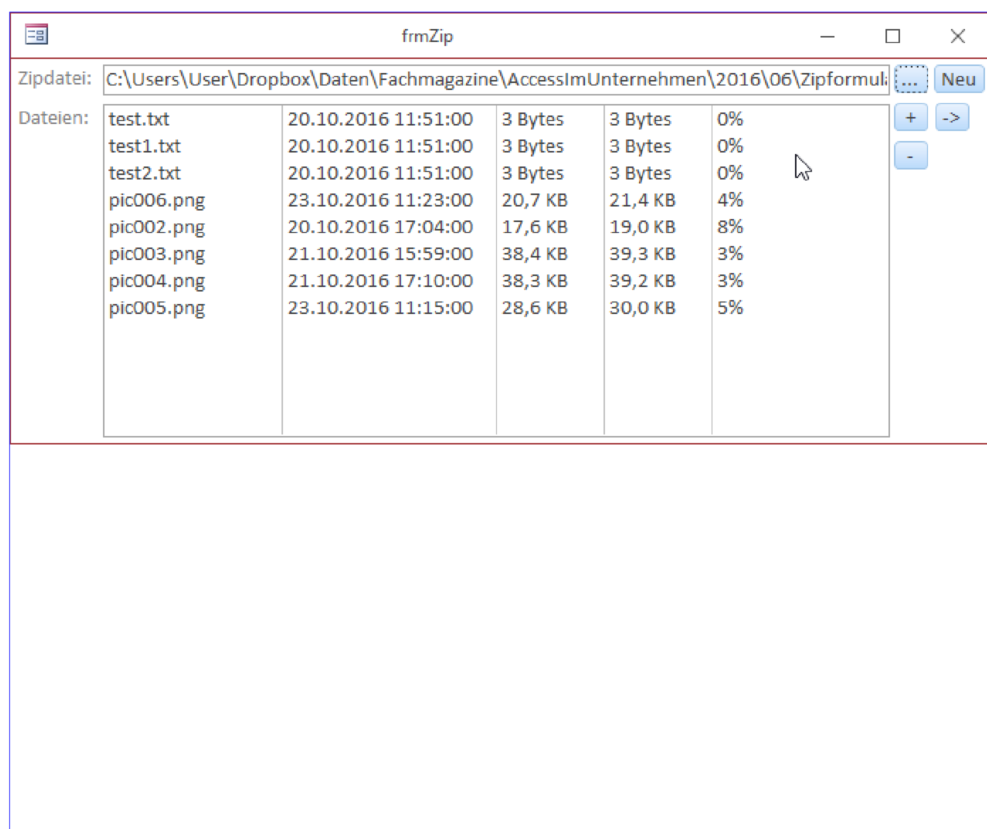

Bild 1: Erstellen einer Zip-Datei mit Windows-Bordmitteln

Die hier vorgestellten Techniken nutzen allein die vom Betriebssystem bereitgestellten Funktionen.

In einem halbwegs frischen System, also ohne zusätzliche Einträge in den Kontextmenüs, sieht die eingebaute Funktion zum Erstellen von Zip-Dateien wie in Bild 1 aus.

Zip-Datei erstellen

Es gibt keine eingebaute Funktion, um eine leere Zip-Datei zu erstellen. Wie können wir dieses Problem umgehen? Dazu gibt es zumindest zwei Möglichkeiten:

- Wir legen eine neue, leere Zip-Datei mit einem geeigneten Programm wie WinZip oder WinRar an, spei-

chern diese in einem Anlage-Feld der Datenbank und exportieren dieses bei Bedarf als leere Zip-Datei in das Zielverzeichnis.

- Wir schauen uns an, wie eine leere Zip-Datei aufgebaut ist, und bauen diese einfach nach.

Ersteres würde zumindest noch eine weitere Tabelle mit dem Anlage-Feld erforderlich machen, was wir an dieser Stelle für übertriebenen Aufwand halten. Also erstellen wir lieber mit WinZip oder WinRar eine leere Zip-Datei und betrachten diese in einem Texteditor.

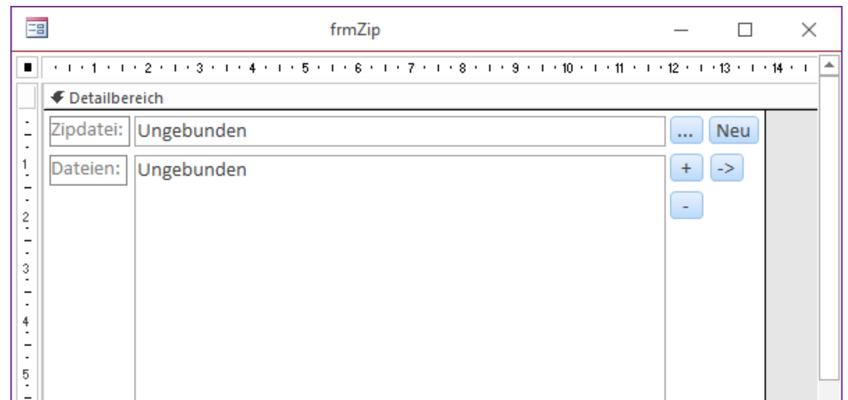

Bild 2: Aussehen einer leeren Zip-Datei

Das Ergebnis sieht dann etwa wie in Bild 2 aus. Was soll schon schieflaufen, wenn wir einfach eine leere Textdatei mit den eingebauten VBA-Befehlen erzeugen und die hier enthaltenen Zeichen einfügen? Also machen wir uns an die Arbeit. Die Prozedur aus Listing 1 erwartet den Namen der zu erstellenden Zip-Datei und legt diese dann an. Dazu erstellt sie eine **String**-Variable namens **strZipinhalt** und füllt sie genau mit dem Inhalt der Datei, die wir oben probehalber erzeugt und im Texteditor angezeigt haben. Die Zeichen in der verwendeten Vorlage werden vom Texteditor im hexadezimalen Format angezeigt. Wir müssen diese, bevor wir die Zeichen mit der Funktion **Chr()** ermitteln, in das Dezimalformat umwandeln. Dies erledigen wir durch Voranstellen der Zeichenkette **&H**. Die achtzehn Nullen fügen wir mit der Funktion **String()** hinzu, der wir als ersten Parameter die Anzahl der zu liefernden Zeichen und als zweiten den Code für das gewünschte Zeichen übergeben.

Die Prozedur ermittelt dann mit der Funktion **FreeFile** eine Dateinummer für den Zugriff auf die neu zu erstellende Datei. Diese öffnen wir mit der **Open**-Methode, der wir den Namen der zu erstellenden Datei folgen lassen (**strZipdatei**). Der Öffnungsmodus ist **For Binary Access Write**, die Dateinummer lautet schließlich **#intDateinummer**. Die **Put**-Methode schreibt den Inhalt der Variablen **strZipinhalt** in die mit **#intDateinummer** geöffnete Datei, die **Close**-Methode schließt diese wieder.

```
Public Sub ZipErstellen(strZipdatei As String)
    Dim intDateinummer As Integer
    Dim strZipinhalt As String
    strZipinhalt = Chr(&H50) & Chr(&H4B) & Chr(&H5) & Chr(&H6) & String(18, 0)
    intDateinummer = FreeFile
    Open strZipdatei For Binary Access Write As #intDateinummer
    Put #intDateinummer, , strZipinhalt
    Close #intDateinummer
End Sub
```

Listing 1: Funktion zum Erstellen einer leeren Zipdatei

Ein beispielhafter Aufruf für diese Prozedur sieht etwa wie folgt aus:

```
Public Sub Test_ZipErstellen()
    On Error Resume Next
    Kill CurrentProject.Path & "\testvba.zip"
    On Error GoTo 0
    ZipErstellen CurrentProject.Path & "\testvba.zip"
End Sub
```

```
Public Function Zippen(strZipdatei As String, strDatei As String) As Boolean
    Dim objZip As Object
    Dim objShell As Object
    Dim intCount As Integer
    Set objShell = CreateObject("Shell.Application")
    Set objZip = objShell.Namespace((strZipdatei))
    If objZip Is Nothing Then
        ZipErstellen strZipdatei
        Set objZip = objShell.Namespace(CVar(strZipdatei))
    End If
    intCount = objZip.items.Count
    'objZip.CopyHere strDatei
    objZip.CopyHere (strDatei) 'Klammer Pflicht, da sonst kein zippen
    Do
        Call Sleep(100)
    Loop While Not ZipdateiGeschlossen(strZipdatei)
    If objZip.items.Count > intCount Then
        Zippen = True
    End If
End Function
```

Listing 2: Funktion zum Hinzufügen einer Datei zu einer Zip-Datei

Die so erzeugte Datei hat genau den gleichen Inhalt wie die oben mit WinZip erzeugte Datei im Texteditor und kann somit auch etwa mit WinZip oder WinRar geöffnet werden. Außerdem, und das ist ja unser erklärtes Ziel, können wir diese neu erstellte Zip-Datei nun mit den von uns gewünschten Dateien füllen.

Zip-Datei füllen

Der zweite Schritt besteht nun darin, eine Datei zu einer Zip-Datei hinzuzufügen. Dazu verwenden wir die Funktion **Zippen** aus Listing 2. Die Datei erwartet zwei Parameter:

- **strZipdatei** ist der Pfad zur Zip-Datei,
- **strDatei** ist der Pfad zu der hinzuzufügenden Datei.

Die Funktion deklariert zwei Objektvariablen namens **objZip** und **objShell** sowie eine **Integer**-Variable namens **intCount**. **objShell** füllen wir mit einem neuen Objekt auf Basis der Klasse **Shell.Application** wie bereits in der Prozedur **ZipErstellen**. **objZip** erhält einen Verweis auf das **Namespace**-Objekt auf Basis der Zip-Datei.

Diesen holen wir über die **Namespace**-Eigenschaft des **Shell.Application**-Objekts. Danach prüft die Funktion, ob **objZip** einen Verweis enthält. Dies ist nicht der Fall, wenn die mit **strZipdatei** übergebene Datei keine Zip-Datei ist. In diesem Fall erstellt die Funktion die Zip-Datei mit der bereits vorgestellten Prozedur **ZipErstellen** neu und erneuert dann den in **objZip** gespeicherten Verweis auf diese Datei.

Nun folgt ein Schritt, der für die Überprüfung des Erfolgs wichtig ist. Mit der Eigenschaft **Count** der **items**-Auflistung des **Namespace**-Objekts mit der Zip-Datei ermitteln wir die Anzahl der in der Zip-Datei enthaltenen Dateien beziehungsweise Elemente (hier werden auch Ordner mitgezählt). Bei einer frisch angelegten Zip-Datei sollte dies den Wert **0** liefern.

Anschließend würden wir normalerweise einfach die **CopyHere**-Methode des **Namespace**-Objekts mit der Zip-Datei aufrufen und als Parameter den Namen der hinzuzufügenden Datei übergeben:

```
objZip.CopyHere strDatei
```

Dies ist auf dem Testsystem jedoch zuverlässig schiefgegangen. Nach einigen Recherchen stellte sich heraus, dass man den Dateinamen in Klammern einfassen muss, damit es gelingt:

```
objZip.CopyHere (strDatei)
```

Die Methode **CopyHere** ist eine asynchrone Methode, das heißt, sie wird ausgeführt, während der Code weiterläuft. Wenn wir also direkt danach mit der folgenden Anweisung prüfen würden, ob die Datei hinzugefügt wurde, kann es sein, dass das Hinzufügen noch nicht abgeschlossen wurde:

```
If objZip.items.Count > intCount Then
```

Also bauen wir eine **Do...Loop**-Schleife ein, die mit einer Hilfsfunktion prüft, ob die Datei bereits zur Zip-Datei hinzugefügt wurde. Mit jedem Durchlauf dieser Schleife rufen wir zunächst die **Sleep**-Funktion mit dem Wert **100** als Parameter auf, was eine Pause von einer Zehntelsekunde hervorruft. Die **Sleep**-Funktion deklarieren wir wie folgt in einem Standardmodul:

```
Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As
Long)
```

Die Abbruchbedingung lautet so:

```
Loop While Not ZipdateiGeschlossen(strZipdatei)
```

Die hier angegebene Funktion **ZipdateiGeschlossen** erwartet den Namen der zu überprüfenden Datei als Parameter (s. Listing 3). Sie ermittelt eine Nummer für den schreibenden Zugriff auf die angegebene Datei und versucht dann, diese mit der **Open**-Anweisung zu öffnen.

Ist die Datei noch geöffnet, löst diese Anweisung einen Fehler aus und die Funktion springt zur Marke **Ende_ZipdateiGeschlossen**. Die Funktion liefert dann den Wert **False** zurück. Anderenfalls schließt die Funktion die Datei mit der **Close**-Anweisung wieder und stellt den Rückgabewert auf **True** ein.

```
Private Function ZipdateiGeschlossen(strZipdatei As String) As Boolean
    Dim intDateinummer As Integer
    intDateinummer = FreeFile
    On Error GoTo Ende_ZipdateiGeschlossen
    Open strZipdatei For Binary Access Read Lock Write As intDateinummer
    Close intDateinummer
    ZipdateiGeschlossen = True
Ende_ZipdateiGeschlossen:
End Function
```

Listing 3: Funktion zur Prüfung, ob eine bestimmte Datei aktuell geschlossen ist

Auf diese Weise wird die **Do...Loop**-Schleife so lange durchlaufen, bis die Zip-Datei nicht mehr durch die **CopyHere**-Anweisung in Beschlag genommen wird und die Datei hinzugefügt wurde. Danach können wir dann in der **If...Then**-Bedingung prüfen, ob sich die Anzahl der enthaltenen Dateien im Anschluss an das Hinzufügen geändert hat. Falls ja, wird der Rückgabewert der Funktion **Zippen** auf den Wert **True** eingestellt.

Die Funktion **Zippen** testen wir etwa mit folgender Routine:

```
Public Sub Test_Zippen_Datei_klein()
    Dim strDatei As String
    Dim strZipdatei As String
    strZipdatei = CurrentProject.Path & "\testvba.zip"
    strDatei = CurrentProject.Path & "\test.txt"
    MsgBox "Zippen erfolgreich? " _
        & Zippen(strZipdatei, strDatei)
End Sub
```

Dies stellt die Pfadangaben zur Zip-Datei und zur hinzuzufügenden Datei zusammen und übergibt diese an die Funktion **Zippen**. Das Ergebnis wird dann in einer Meldung ausgegeben.

Verzeichnisse zippen

Mit der Funktion **Zippen** können Sie auch komplette Verzeichnisse zippen. Dazu übergeben Sie als zweiten Parameter einfach den Pfad zu dem zu zippenden Verzeichnis:

tblDateien

Feldname	Felddatentyp	Beschreibung (optional)
DateiID	AutoWert	Primärschlüsselfeld
Dateiname	Kurzer Text	Name der Datei
ZuletztGeaendert	Datum/Uhrzeit	Letztes Änderungsdatum
GepackteGroesse	Zahl	Größe der gepackten Dateien
UngepackteGroesse	Zahl	Größe der ungepackten Dateien
Ratio	Zahl	Verhältnis gepackt/ungepackt
GepackteGroesseText	Kurzer Text	Größe der gepackten Datei mit Einheit
UngepackteGroesseText	Kurzer Text	Größe der ungepackten Datei mit Einheit

Bild 3: Zip-Datei mit einigen per VBA hinzugefügten Elementen

```
Public Sub Test_Zippen_Folder()
    Dim strDatei As String
    Dim strZipdatei As String
    strZipdatei = CurrentProject.Path & "\testvba.zip"
    strDatei = CurrentProject.Path & "\test\"
    MsgBox "Zippen erfolgreich? " _
        & Zippen(strZipdatei, strDatei)
End Sub
```

```
Public Function Zippen(strZipdatei As String, ByVal strDateiVerzeichnis As String, _
        Optional lngAnzahl As Long, Optional bolOhneVerzeichnis As Boolean = False) As Boolean
    Dim objZip As Object, objShell As Object
    Dim intCount As Integer, strDatei As String, strVerzeichnis As String
    Set objShell = CreateObject("Shell.Application")
    Set objZip = objShell.NameSpace((strZipdatei))
    If objZip Is Nothing Then
        ZipErstellen strZipdatei
        Set objZip = objShell.NameSpace(CVar(strZipdatei))
    End If
    intCount = objZip.items.Count
    If IstVerzeichnis(strDateiVerzeichnis) And bolOhneVerzeichnis Then
        strVerzeichnis = Trim(strDateiVerzeichnis)
        If Not Right(strVerzeichnis, 1) = "\" Then
            strVerzeichnis = strVerzeichnis & "\"
        End If
        strDatei = Dir(strVerzeichnis)
        Do While Len(strDatei) > 0
            objZip.CopyHere (strVerzeichnis & strDatei)
            Do
                Call Sleep(100)
            Loop While Not ZipdateiGeschlossen(strZipdatei)
            strDatei = Dir()
        Loop
    Else
        objZip.CopyHere (strDateiVerzeichnis)
        Do
            Call Sleep(100)
        Loop While Not ZipdateiGeschlossen(strZipdatei)
    End If
    lngAnzahl = objZip.items.Count - intCount
    If lngAnzahl Then
        Zippen = True
    End If
End Function
```

Listing 4: Zippen-Funktion, die auch einzelne Elemente eines Verzeichnisses zippt – mit Rückgabe der Anzahl

Nachdem Sie diese beiden Test-Routinen aufgerufen haben, erhalten Sie eine Zip-Datei, die mit WinRar geöffnet etwa wie in Bild 3 aussieht.

Die bisher beschriebene Version der Funktion **Zippen** haben wir unter dem Namen **Zippen_Version1** im Modul **mdlZippen** abgelegt.

Dateien eines Verzeichnisses einzeln zippen

Nun möchten Sie möglicherweise nicht das komplette Verzeichnis zippen, sondern nur die in diesem Verzeichnis enthaltenen Dateien. Das können Sie erledigen, indem Sie die Funktion **Zippen** entsprechend anpassen.

Die neue Version sieht nun wie in Listing 4 aus. Sie erwartet zwei Parameter mehr als die vorherige Version, nämlich **lngAnzahl**

und **bolOhneVerzeichnis**. Beide sind nur in Zusammenhang mit der Übergabe eines Verzeichnisses mit dem Parameter **strDateiVerzeichnis** interessant, da nur dann mehrere Dateien verarbeitet werden.

Die Funktion verwendet außerdem drei zusätzliche Variablen: **intCount** nimmt die Anzahl der hinzufügten Elemente auf. **strDatei** soll, wenn es sich beim Inhalt des Parameters **strDateiVerzeichnis** um ein Verzeichnis handelt, zum Durchlaufen der darin enthaltenen Dateien dienen. **strVerzeichnis** nimmt entsprechend das Verzeichnis aus **strDateiVerzeichnis** auf – dies nur, um Verwechslungen zu vermeiden. Die Zip-Datei wird wie gehabt referenziert beziehungsweise, sofern noch nicht vorhanden, erstellt. Die Variable **intCount** speichert nach wie vor die Anzahl der Elemente des Zip-Archivs vor dem Hinzufügen der Elemente.

Dann prüft die Hilfsfunktion **IstVerzeichnis**, ob es sich bei dem mit **strDateiVerzeichnis** übergebenen Pfad um ein Verzeichnis handelt. Ist dies der Fall und gleichzeitig der Parameter **bolOhneVerzeichnis** auf **True** eingestellt, wird der erste Teil der **If...Then**-Bedingung abgearbeitet.

Innerhalb dieses Abschnitts trägt die Funktion den Wert von **strDateiVerzeichnis** in die Variable **strVerzeichnis** ein. Dann prüft sie, ob **strVerzeichnis** ein abschließendes Backslash-Zeichen (\) enthält, und fügt dieses gegebenenfalls noch hinzu. Schließlich liest die Prozedur mit der **Dir**-Funktion den Namen des ersten Elements des in **strVerzeichnis** angegebenen Verzeichnisses in die Variable **strDatei** ein.

Damit steigt die Funktion in eine **Do While**-Schleife ein, die so lange läuft, wie die Länge der in **strDatei** gespeicherten Zeichenkette größer als **0** ist, die Variable also einen Wert enthält. Innerhalb der Schleife fügt die **CopyHere**-Methode nun die Datei mit dem aus **strVerzeichnis** und **strDatei** zusammengesetzten Pfad zu der mit **objZip** referenzierten Zip-Datei hinzu. Die **Do...Loop**-Schleife mit dem Ausdruck **Not ZipdateiGeschlossen(strZipdatei)** hält die weitere Ausführung der Funktion wieder so lange an, bis die Zip-Datei durch den asynchronen Befehl **CopyHere** fertig beschrieben wurde.

Danach ermittelt die Funktion über den einfachen Aufruf der **Dir**-Funktion die nächste Datei in dem angegebenen Verzeichnis. Warum kein Parameter? Weil die **Dir**-Funktion so strukturiert ist, dass sie mit Verzeichnisangabe das erste Element dieses Verzeichnisses zurückliefert und ohne Parameter immer das jeweils folgende Element. Die Schleife beginnt nun von vorn, wo sie wieder prüft, ob **strDatei** noch mit einem Dateinamen gefüllt ist. Falls ja, werden auch die folgenden Dateien zur Zip-Datei hinzugefügt, anderenfalls ist die Schleife abgearbeitet.

Der Else-Teil der **If...Then**-Bedingung fügt wie bereits in der vorherigen Version der **Zippen**-Funktion die mit **strDateiVerzeichnis** angegebene Datei oder das Verzeichnis zur Zip-Datei hinzu.

Ob während des Aufrufs der **Zippen**-Funktion ein oder mehrere Dateien zur Zip-Datei hinzugefügt wurden, ermitteln wir wieder mit der Differenz aus **objZip.items.Count** und dem Wert aus der Variablen **intCount**. Diesmal landet das Ergebnis allerdings gleich im Rückgabeparameter **lngAnzahl**. Deren Inhalt überprüft dann auch die **If...Then**-Bedingung, welche den Rückgabewert der Funktion **Zippen** im Falle eines Wertes größer **0** auf **True** einstellt.

Die Hilfsfunktion **IstVerzeichnis**, mit der wir prüfen, ob es sich bei dem mit **strDateiVerzeichnis** übergebenen

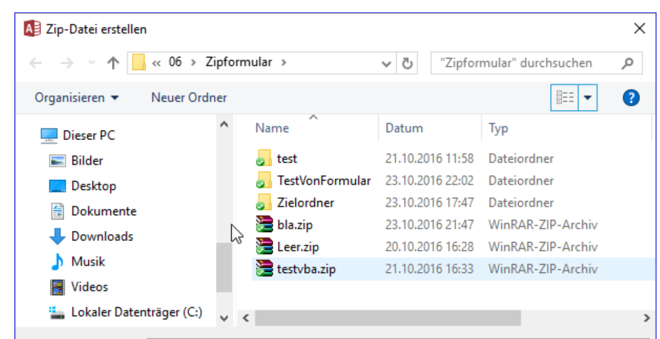

Bild 4: Zip-Datei mit dem Inhalt eines Verzeichnis als einzelne Dateien

Daten aus Excel-Sheets importieren

Gelegentlich werden Sie als Access-Entwickler vor der Aufgabe stehen, den Import aus Excel-Dateien zu implementieren. Manchmal gibt es dabei Sonderfälle, in denen etwa die Daten aus mehreren Sheets eingelesen oder bereitgestellt werden sollen. Wir schauen uns an, wie das mit mehreren Sheets gelingt.

Verschiedene Excel-Sheets importieren

Wenn Sie einmal verschiedene Seiten eines Excel-Dokuments importieren möchten, müssen Sie dafür bei der **TransferSpreadsheet**-Methode des **DoCmd**-Objekts einen bestimmten Ausdruck für den Parameter **Range** angeben.

Angenommen, Sie verwenden eine Excel-Datei, die zwei Sheets namens **Tabelle1** und **Tabelle2** wie in Bild 1 enthält.

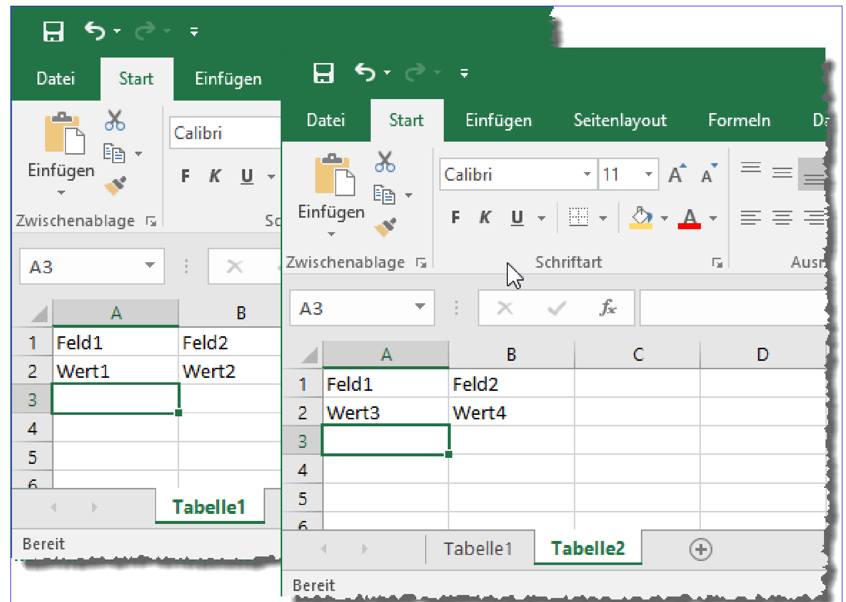

Bild 1: Excel-Datei mit zwei Sheets

Im ersten Beispiel wollen wir beide Sheets in jeweils eine eigene Tabelle einfügen – die Feldüberschriften sind ja bereits auf geeignete Weise ausgelegt.

Der dazu verwendete Befehl lautet **DoCmd.Transfer-Spreadsheet**. Normalerweise würden Sie diesen Befehl wie folgt einsetzen, um die Daten von Excel in eine Access-Datenbank zu importieren:

```
DoCmd.TransferSpreadsheet acImport, acSpreadsheetType-
Excel12, "tblTest1", CurrentProject.Path & "\Test.xlsx",
True
```

Der erste Parameter legt dabei fest, dass es sich um einen Import und nicht etwa um eine Verknüpfung handelt. Der zweite legt die Art der Quelle fest. Der dritte Parameter bestimmt den Namen der Tabelle, in welche die Daten importiert werden sollen, der vierte die Quelldatei.

Schließlich legen Sie mit dem Wert **True** für den fünften Parameter fest, dass die Werte der ersten Zeile als Spaltenüberschriften interpretiert werden sollen.

Um diese Anweisung testweise wiederholt aufrufen zu können, haben wir diese in die Prozedur aus Listing 1 eingefügt. Die ersten Zeilen dieser Anweisung versuchen, bei deaktivierter eingebauter Fehlerbehandlung eine eventuell vorhandene Tabelle namens **tblTest1** zu löschen. Dies kann zum Beispiel fehlschlagen, weil die Tabelle noch gar nicht vorhanden ist – dieser Fehler wird ignoriert.

Es kann jedoch auch sein, dass die Tabelle gerade geöffnet ist. In diesem Fall erscheint eine entsprechende Meldung und die Prozedur wird abgebrochen. Alternativ könnten Sie hier auch eine Anweisung voranstellen, welche die Tabelle zuvor schließt.

```
Public Sub ImportExcelEinfach()
    On Error Resume Next
    DoCmd.DeleteObject acTable, "tblTest1"
    If Err.Number = 2008 Then
        MsgBox "Die Tabelle kann nicht gelöscht und neu erstellt werden, da diese geöffnet ist."
        Exit Sub
    End If
    On Error GoTo 0
    DoCmd.TransferSpreadsheet acImport, acSpreadsheetTypeExcel12, "tblTest1", CurrentProject.Path & "\Test.xlsx", True
End Sub
```

Listing 1: Import eines Excel-Sheets

Das Ergebnis sieht wie in Bild 2 aus. Es werden natürlich nur die Daten der ersten gefundenen Tabelle der Excel-Datei importiert.

Zwei Sheets in zwei Tabellen

Im nächsten Schritt wollen wir die Daten der beiden Sheets in jeweils eine eigene Tabelle importieren. Hier wird es interessant: Spätestens in der zweiten **DoCmd.TransferSpreadsheet**-Anweisung müssen wir irgendwie festlegen, dass nicht das erste, sondern das zweite Sheet der Excel-Datei importiert werden soll.

Wenn man den Trick einmal kennt, ist es jedoch ganz einfach: Sie geben einfach für den sechsten Parameter namens **Range** den Namen des Sheets an, wie er unten im Reiter der Excel-Tabelle angegeben ist – in unserem Fall also etwa **Tabelle1** oder **Tabelle2**. Das ist allerdings

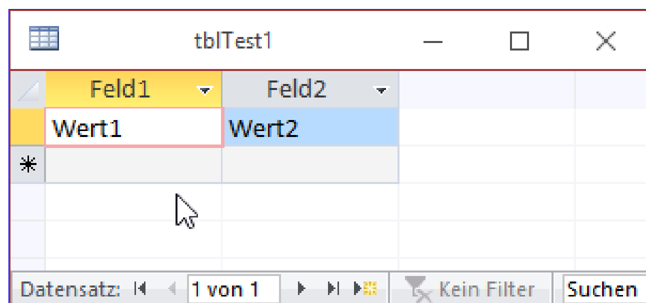

Bild 2: Import ohne Angabe des Quellsheets

noch nicht der ganze Clou: Sie müssen diesen Bezeichnungen nämlich noch ein Ausrufezeichen anhängen, also beispielsweise **Tabelle1!** oder **Tabelle2!**. Wie dies aussieht, sehen Sie in Listing 2.

Das Ergebnis finden Sie schließlich in Bild 3. Die Tabellen könnten auch unterschiedlich aufgebaut sein, da der Import einzeln erfolgt.

```
Public Sub ImportExcelNachTabelle()
    On Error Resume Next
    DoCmd.Close acTable, "tblTest1"
    DoCmd.Close acTable, "tblTest2"
    DoCmd.DeleteObject acTable, "tblTest1"
    DoCmd.DeleteObject acTable, "tblTest2"
    On Error GoTo 0
    DoCmd.TransferSpreadsheet acImport, acSpreadsheetTypeExcel12, "tblTest1", CurrentProject.Path & "\Test.xlsx", _
        True, "Tabelle1!"
    DoCmd.TransferSpreadsheet acImport, acSpreadsheetTypeExcel12, "tblTest2", CurrentProject.Path & "\Test.xlsx", _
        True, "Tabelle2!"
End Sub
```

Listing 2: Import zweier Excel-Sheets

Steuerelemente zur Laufzeit debuggen

Wenn im Formular oder in den enthaltenen Steuerelementen zur Laufzeit merkwürdige Werte angezeigt werden oder die Elemente nicht wie gewünscht reagieren, kann das verschiedene Ursachen haben. Viele davon lassen sich am einfachsten aufdecken, wenn Sie die Werte und Eigenschaften auslesen, während das Formular in der Formularansicht geöffnet ist. Doch das ist nicht ganz einfach, denn es erfordert eine Menge Tipparbeit – und oft die genaue Kenntnis der Formular-, Unterformular- oder Steuerelementnamen. Wir zeigen eine Methode, die das Abfragen der Eigenschaftswerte erheblich vereinfacht.

Die Standardmethode

Sind wir mal ehrlich: Wenn Sie den Wert der Eigenschaft eines Steuerelements eines geöffneten Formulars ermitteln wollen, geht man üblicherweise über das Direktfenster und ermittelt den gesuchten Wert mit der **Debug.Print**-Anweisung.

Schauen Sie sich beispielsweise das Formular aus Bild 1 an. Es enthält einige Kombinationsfelder, die natürlich – wie es in einer professionel gestalteten Anwendung der Fall sein sollte – nicht die Primärschlüsselwerte der gebundenen Datenherkunft, sondern die für den Benutzer wichtigen Werte wie den Kundennamen, den Namen des Sachbearbeiters oder auch im Unterformular den Namen des Artikels anzeigen. Nun kommt es vor, dass Sie sich fragen, welches Formular Sie dort gerade überhaupt geöffnet haben. Wenn es das einzige Formular ist, können Sie dies mit dem ersten Eintrag der **Forms**-Auflistung herausfinden:

```
? Forms(0).Name
frmBestellungen
```

Sind mehrere Formular geöffnet, hilft dies nicht weiter, denn Sie kennen ja den Index des zu untersuchenden Formulars nicht. In diesem Fall greifen Sie einfach über die Eigenschaft **ActiveForm** des **Screen**-Objekts auf das Formular zu und lassen sich dessen Namen ausgeben:

```
? Screen.ActiveForm.Name
frmBestellungen
```

Auf diese Weise haben Sie schon einmal den Namen des Formulars herausgefunden. Damit können Sie nun gezielt auf dieses Formular und seine Eigenschaften, wie zum Beispiel **Name**, zugreifen (was zugegebenermaßen wenig

Bild 1: Formular und Unterformular zur Verwaltung von Bestellungen

Sinn macht, da wir den Namen ja schon kennen – aber es ist ja nur ein Beispiel!):

```
? Forms!frmBestellungen.Name
frmBestellungen
```

Sie können damit natürlich auch etwa die Datenherkunft des Formulars ausgeben lassen:

```
? Forms!frmBestellungen.Recordsource
tblBestellungen
```

Auf die gleiche Weise greifen Sie auch auf die übrigen Eigenschaften des Formulars zu.

Steuerelemente ansprechen

Nun gehen wir einen Schritt weiter und kümmern uns um die Steuerelemente. Wenn Sie nicht wissen, wie ein Steuerelement heißt, können Sie sich in einer Schleife die Steuerelementnamen ausgeben lassen:

```
Public Sub Steuerelementnamen()
    Dim ctl As Control
    For Each ctl In Forms!frmBestellungen.Controls
        Debug.Print ctl.Name
    Next ctl
End Sub
```

Dies gibt die Namen aller Steuerelemente im Direktfenster aus. Wenn Sie das gewünschte Steuerelement gefunden haben, können Sie wie folgt darauf zugreifen – in diesem Beispiel auf das Steuerelement **BestellungID**:

```
? Forms!frmBestellungen.Controls("BestellungID").Value
 10248
```

Das entspricht der folgenden Syntax mit dem Ausrufezeichen:

```
? Forms!frmBestellungen!BestellungID
10248
```

Aktivierbare Steuerelemente ansprechen

Bei den Steuerelementen, die aktivierbar sind, also denen Sie den Fokus etwa per Mausklick zuweisen können, gibt es noch eine einfachere Methode, beispielsweise den Namen zu ermitteln. Dazu verwenden Sie eine weitere Eigenschaft des **Screen**-Objekts, nämlich **ActiveControl**. Wenn Sie einmal das Steuerelement zur Anzeige des Feldes **BestellungID** per Mausklick aktivieren und dann im Direktfenster die folgende Anweisung absetzen, erhalten Sie ebenfalls den Namen dieses Steuerelements:

```
? Screen.ActiveControl.Name
BestellungID
```

Auf den enthaltenen Wert oder andere Eigenschaften greifen Sie auf diesem Wege zu:

```
? Screen.ActiveControl.Value
 10248
```

Nicht sichtbare Werte ermitteln

Aber warum sollte man überhaupt auf den Wert eines Steuerelements zugreifen, wenn man diesen sowieso im Formular selbst ablesen kann? Nun: Es gibt ja auch Steuerelemente, die ihren eigentlichen Inhalt nicht offenbaren – zum Beispiel die Kombinationsfelder dieses Beispielformulars, die allesamt nicht den Wert der gebundenen Spalte anzeigen (also den Feldwert des Feldes der Datenherkunft des Formulars, an das sie gebunden sind), sondern den Wert einer anderen Spalte der Datensatzherkunft des Kombinationsfeldes. Um die **KundeID** des Kunden aus dem Kombinationsfeld **KundeID** zu ermitteln, geben Sie folglich ein:

```
? Forms!frmBestellungen!KundeID
 90
```

Diese Information können Sie dem Kombinationsfeld sonst nicht entlocken. Sie können natürlich auch auf den Inhalt der übrigen Spalten der Datensatzherkunft des Kombinationsfeldes zugreifen:

```
? Forms!frmBestellungen!KundeID.Column(1)
Wilman Kala
```

Noch sinnvoller ist dies natürlich, wenn die Datensatzherkunft des Kombinationsfeldes noch weitere Felder enthält, die nicht eingeblendet sind. Mit der Eigenschaft **Column** stehe ich übrigens auf Kriegsfuß – ich gebe immer Columns ein, da ich denke, dass diese Eigenschaft, der man ja einen Index übergibt, eine Auflistung sein und dementsprechend im Plural angegeben werden müsste. Deshalb wäre es praktisch, wenn man hier mit IntelliSense arbeiten könnte – aber der VBA-Editor erkennt an dieser Stelle nicht den Typ des Objekts **KundeID** und blendet somit auch nicht dessen Eigenschaften und Methoden ein. Später schauen wir uns allerdings eine Möglichkeit an, auch über das Direktfenster auf solche Informationen per IntelliSense zuzugreifen.

Unterformular-Steuerelement referenzieren

Nun stürzen wir uns allerdings erstmal auf das Unterformular. Hier entsteht oft das Missverständnis, dass das Unterformular-Steuerelement, das ja quasi nur ein Rahmen für das eigentliche, als Unterformular angegebene Formular ist, als Formular angesprochen wird.

Das Unterformular-Steuerelement hat jedoch ganz andere Eigenschaften als das darin enthaltene Formular, und Sie können auch nicht direkt über das Unterformular-Steuerelement auf die enthaltenen Steuerelemente des Unterformulars zugreifen. Zur Verdeutlichung haben wir das Unterformular-Steuerelement, das beim Hineinziehen des Unterformulars in den Entwurf des Hauptformulars unpraktischerweise den gleichen Namen erhält wie das Unterformular selbst (hier **sfmBestellungen**), umbenannt, und zwar in **sfmBestellungen_UFD** (für Unterformular-Control, s. Bild 2).

Nun können wir zunächst wie folgt auf das Unterformular-Steuerelement zugreifen und uns zum Beispiel die beiden Felder ausgeben lassen, über welche die Synchronisation zwischen den Datensätzen aus Haupt- und Unterformular hergestellt wird:

```
? Forms!frmBestellungen!sfmBestellungen_UFC.LinkChild-
Fields
BestellungID
? Forms!frmBestellungen!sf
mBestellungen_UFC.LinkMas-
terFields
BestellungID
```

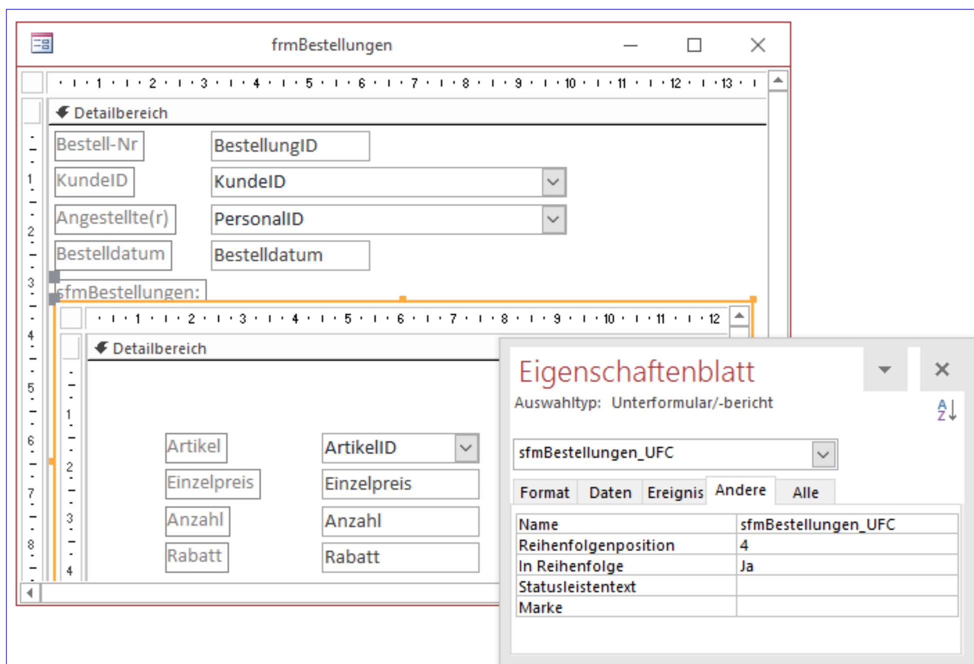

Bild 2: Umbenennen des Unterformulare-Steuerelements

Den Namen dieser beiden Eigenschaften müsste man zu diesem Zweck auch erst mal kennen beziehungsweise ermitteln. Das Unterformular-Steuerelement ist übrigens eines der Steuerelemente, dem Sie nicht den Fokus zuweisen können. Sie können also auch nicht mit **Screen.ActiveControl** darauf zugreifen.

RDBMS-Zugriff per VBA: Daten abfragen

Im Beitrag »RDBMS-Zugriff per VBA: Verbindungen« haben wir die Grundlage für den Zugriff auf SQL Server-Datenbanken geschaffen. Nun gehen wir einen Schritt weiter: Wir wollen mit den dort beschriebenen Methoden etwa zum Zusammenstellen einer Verbindungszeichenfolge auf die Daten einer SQL Server-Datenbank zugreifen. Dabei lernen Sie eine Reihe interessanter Funktionen kennen, die den Zugriff deutlich vereinfachen und die auch noch überaus performant sind.

Der eingangs erwähnte Beitrag **RDBMS-Zugriff per VBA: Verbindungen** (**www.access-im-unternehmen.de/1054**) hat die Werkzeuge dafür geliefert, dass Sie Verbindungszeichenfolgen aus einer Tabelle zusammenstellen und diese zum Aufbau einer Verbindung nutzen können. Im vorliegenden Beitrag nutzen wir vor allem die mit der Funktion **Standardverbindungszeichenfolge** ermittelte Verbindungszeichenfolge, um auf die Tabellen der SQL Server-Datenbank zuzugreifen. Sie können aber natürlich auch eine manuell als String zusammengestellte Verbindungszeichenfolge nutzen.

Bild 1: Die in diesem Beitrag verwendete Verbindungszeichenfolge im Formular

Beispieldatenbank

Wir verwenden weiterhin die im ersten Teil der Beitragsreihe vorgestellte Datenbank **Suedsturm** auf Basis des LocalDB-Datenbanksystems. Sie können aber natürlich auch eine Datenbank über den SQL Server nutzen (s. Bild 1).

Die Verknüpfung zu den Tabellen der SQL Server-Datenbank erstellen Sie am einfachsten mit dem Formular **frmTabellenVerknuepfen**, das wir im Beitrag **SQL Server-Tools** (**www.access-im-unternehmen.de/1061**) vorstellen.

Recordsets auf Basis von SQL Server-Daten

Recordsets sind unter Access ein häufig genutztes Mittel für den Zugriff auf die Daten einer Tabelle oder Abfrage. Natürlich werden Sie diese auch füllen wollen, wenn die Daten Ihrer Datenbank längst in einer SQL Server-Datenbank gelandet sind. Daher schauen wir uns nun zunächst an, wie Sie Recordsets auf der Basis verschiedener Zugriffsarten auf die Daten der SQL Server-Datenbank erstellen und füllen können.

Dabei greifen wir auf einfachem Wege auf die Daten zu, nämlich über eine verknüpfte Tabelle, aber auch auf subtilere Weise – etwa über eine Pass-Through-Abfrage, die sich einer gespeicherten Prozedur als Datenquelle bedient.

Recordset auf Basis einer verknüpften Tabelle

Die einfachste Möglichkeit, per VBA auf die Daten einer verknüpften Tabelle zuzugreifen, ist das DAO-Recordset. Dieses

erhalten Sie mit der **OpenRecordset**-Methode, der Sie als ersten Parameter den Namen der verknüpften Tabelle, als zweiten den Wert **dbOpenSnapshot** (Daten ändern wollen wir nicht per verknüpfter Tabelle, da zu unperformant – ein rein lesender Zugriff sorgt außerdem für wesentlich weniger Sperren auf dem Server) und als dritten Parameter den Wert **dbSeeChanges**, um Fehler in Zusammenhang mit Autowerten in der zugrunde liegenden Tabelle zu vermeiden.

Die folgende Beispielprozedur erstellt ein Recordset auf Basis der Tabelle **tblArtikel** und gibt die Werte der Felder **ArtikelID** und **Artikelname** aller Datensätze dieser Tabelle im Direktfenster aus:

```
Public Sub VerknuepfteTabellePerRecordset()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM tblArtikel", _
        dbOpenSnapshot, dbSeeChanges)
    Do While Not rst.EOF
        Debug.Print rst!ArtikelID, rst!Artikelname
        rst.MoveNext
    Loop
    rst.Close
    Set rst = Nothing
    Set db = Nothing
End Sub
```

Wenn Sie die Daten aus der SQL Server-Tabelle nicht nur lesen, sondern auch ändern wollen, verwenden Sie **dbOpenDynaset** statt **dbOpenSnapshot**.

Vereinfachungen für den Zugriff auf gespeicherte Prozeduren

Gespeicherte Prozeduren sind prinzipiell Abfragen, die Auswahl- oder Aktionsabfragen enthalten und direkt auf dem SQL Server gespeichert sind und dort ausgeführt werden. Sie liefern nur die benötigten Daten zurück und sind daher meist viel schneller, als wenn Sie etwa mit verknüpften Tabellen arbeiten.

Hinweis: Die ab jetzt vorgestellten Techniken erlauben lediglich den lesenden Zugriff auf die ermittelten Daten.

Wenn Sie ein Recordset auf Basis des Ergebnisses einer gespeicherten Prozedur verwenden möchten, benötigen Sie folgende Dinge:

- die gespeicherte Prozedur in der SQL Server-Datenbank,
- eine Pass-Through-Abfrage in der Access-Datenbank und
- VBA-Code, der auf die Pass-Through-Abfrage zugreift.

Im einfachsten Fall handelt es sich um eine gespeicherte Prozedur, die keine Parameter erwartet. Warum ist dies so unkompliziert? Dies liegt in der Natur des Aufrufs einer gespeicherten Prozedur über eine Pass-Through-Abfrage begründet.

Die gespeicherte Prozedur liegt auf dem SQL Server und wird über eine Pass-Through-Abfrage mit der **EXEC**-Anweisung aufgerufen:

```
EXEC cbo.spBeispielprozedur
```

Kein Problem – die Pass-Through-Abfrage kann wie gesehen verwendet werden. Wenn Sie jedoch einen Parameter übergeben müssen, gehört dieser in den SQL-Text der Pass-Through-Abfrage – also beispielsweise so:

```
EXEC dbo.spBeispielProzedurMitParameter 'Beispielparameter'
```

Beispielparameter ist aber nicht bei jedem Aufruf gleich, sonst brauchten Sie ja keinen Parameter. Es wird also immer ein anderer Wert übergeben – die **ID** eines zu löschenden Datensatzes, das Vergleichskriterium für eine **SELECT**-Abfrage et cetera.

Das bedeutet, dass Sie den SQL-Text der Abfrage mit jedem Aufruf neu erstellen müssen.

Als weiteres Kriterium kommt hinzu, dass sich die Verbindungszeichenfolge bei der Arbeit mit einer Kombination aus Access-Frontend und SQL Server-Backend ändern kann – sei es, weil sich der Name des Servers, der Name der Datenbank, die Authentifizierungsmethode oder der zu verwendende Treiber ändert. Die Verbindungszeichenfolge wird an vielen Stellen verwendet, vor allem aber als Eigenschaft der Pass-Through-Abfragen, die sich im Laufe der Entwicklung einer Access-Anwendung mit SQL Server-Backend ansammeln werden.

Aber brauchen Sie all diese Pass-Through-Abfragen überhaupt? Letztlich müssen die meisten ohnehin jeweils mit einem neuen SQL-Ausdruck gefüllt werden, da diese etwa verschiedene Parameter verwenden. Warum also den Navigationsbereich mit hunderten von Pass-Through-Abfragen füllen, wenn man diese auch jeweils temporär erzeugen kann? Die Antwort ist: Irgendwo müssen wir ja den Code für den Zugriff auf die gespeicherten Abfragen speichern. Dazu bietet sich jeweils eine Pass-Through-Abfrage je gespeicherter Abfrage durchaus an.

Wir verwenden aber später dennoch einen Satz von VBA-Funktionen und -Prozeduren, welche die jeweiligen Pass-Through-Abfragen untersuchen, den enthaltenen T-SQL-Code zum Aufruf der gespeicherten Prozedur entnehmen und die Pass-Through-Abfrage auf Basis der zu übergebenden Parameterwerte und der Verbindungszeichenfolge neu erzeugen und das gewünschte Objekt zurückliefern. Denn: Für den Zugriff auf eine gespeicherte Prozedur benötigen wir gar keine spezielle gespeicherte Pass-Through-Abfrage, sondern lediglich eine zur Laufzeit erstellte Abfrage, die wir mit dem Schlüsselwort **EXEC**, dem Namen der gespeicherten Prozedur sowie den gegebenenfalls benötigten Parameterwerten füllen.

In den folgenden Abschnitten sehen wir uns an, wie Sie den Zugriff auf gespeicherte Prozeduren per VBA vereinfachen können. Insgesamt stellen wir dort die folgenden Prozeduren vor, die sich allesamt im Modul **mdlToolsSQLServer** befinden:

- Ausführen einer gespeicherten Prozedur ohne Parameter und Rückgabe des Ergebnisses als Recordset
- Ausführen einer gespeicherten Prozedur mit Parameter und Rückgabe des Ergebnisses als Recordset
- Erstellen einer Pass-Through-Abfrage und Rückgabe des Namens der Abfrage, etwa als Wert der Eigenschaft **RecordSource** (Formulare, Berichte) oder **RowSource** (Kombinationsfeld, Listenfeld), ohne Parameter
- Erstellen derselben Pass-Through-Abfrage, diesmal mit der Übergabe von Parametern

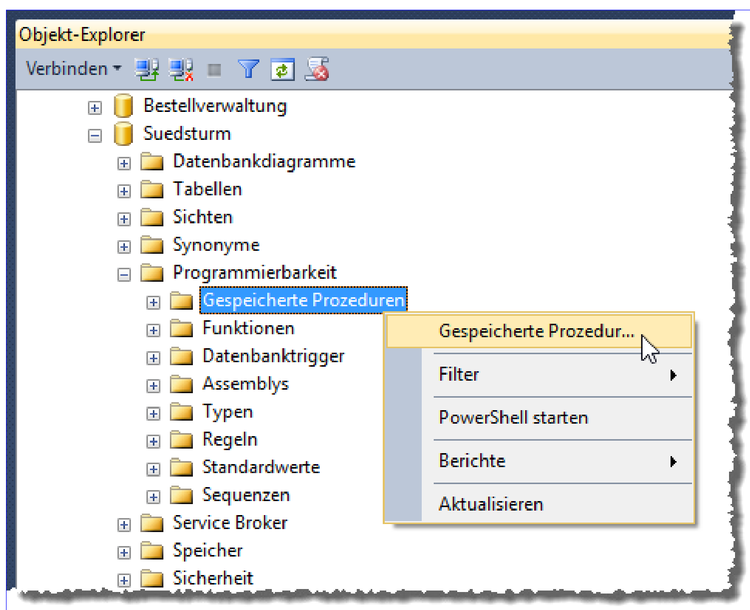

Bild 2: Anlegen einer gespeicherten Prozedur

Recordset aus gespeicherter Prozedur

Die nächste Variante, per Recordset auf die Daten einer SQL Server-Tabelle zuzugreifen, ist der Zugriff auf eine gespeicherte Prozedur (**Stored Procedure**) über eine Pass-Through-Abfrage.

Eine solche Pass-Through-Abfrage müssen Sie zunächst einmal erstellen. Da wir davon

SQLQuery2.sql - (l...OP-PC-I5\User (54))*

```
EXEC pAlleArtikel;
```

	ArtikelID	Artikelname	AufgenommenAm	Auslaufartikel	BestellteEinheiten	Ei
1	1	Chai	2016-01-01	1	1	9.
2	2	Chang	NULL	0	40	9.
3	3	Aniseed Syrup	NULL	1	70	5.
4	4	Chef Anton's...	NULL	1	0	1
5	5	Chef Anton's...	NULL	1	0	1
6	6	Grandma's B...	NULL	0	0	1
7	7	Uncle Bob's ...	NULL	0	0	1
8	8	Northwoods	NULL	0	0	2

ocaldb)\MSSQLLocalDB (13.... | DESKTOP-PC-I5\User (54) | Suedsturm | 00:00:00 | 77 Zeilen

Bild 3: Testen einer gespeicherten Prozedur

ausgehen, dass Sie neben LocalDB und/oder SQL Server auch das SQL Server Managment Studio installiert haben, erstellen wir die neue gespeicherte Prozedur einfach von dort aus.

Dazu starten Sie das SQL Server Management Studio, verbinden sich mit der entsprechenden Instanz (in unserem Fall mit **(localdb)\MSSQLLocalDB**) und wechseln zur Datenbank **Suedsturm**.

Gespeicherte Prozedur erstellen

Hier öffnen Sie das Datenbank-Element **Suedsturm** und darunter die Elemente **Programmierbarkeit** und **Gespeicherte Prozeduren**. Letzteres Element bietet im Kontextmenü den Eintrag **Gespeicherte Prozedur** an, mit der Sie eine neue Prozedur erstellen können (s. Bild 2).

Sie können dies allerdings auch direkt über eine neue Abfrage erledigen, die Sie über den Kontextmenü-Eintrag **Neue Abfrage** des **Suedsturm**-Elements öffnen. Hier tragen Sie die T-SQL-Anweisung ein, mit der Sie die gespeicherte Abfrage erstellen und die wie folgt aussieht:

```
CREATE PROC dbo.pAlleArtikel
AS
SELECT dbo.tblArtikel.ArtikelID,
    dbo.tblArtikel.Artikelname,
    dbo.tblArtikel.AufgenommenAm,
    dbo.tblArtikel.Auslaufartikel,
    dbo.tblArtikel.BestellteEinheiten,
    dbo.tblArtikel.Einzelpreis,
  dbo.tblArtikel.KategorieID,
  dbo.tblArtikel.Lagerbestand,
  dbo.tblArtikel.LieferantID,
  dbo.tblArtikel.Liefereinheit,
  dbo.tblArtikel.Mindestbestand
FROM dbo.tblArtikel;
```

Danach führen Sie die Abfrage mit der Taste **F5** aus (s. Bild 3). Sie finden nun nach einer Aktualisierung einen

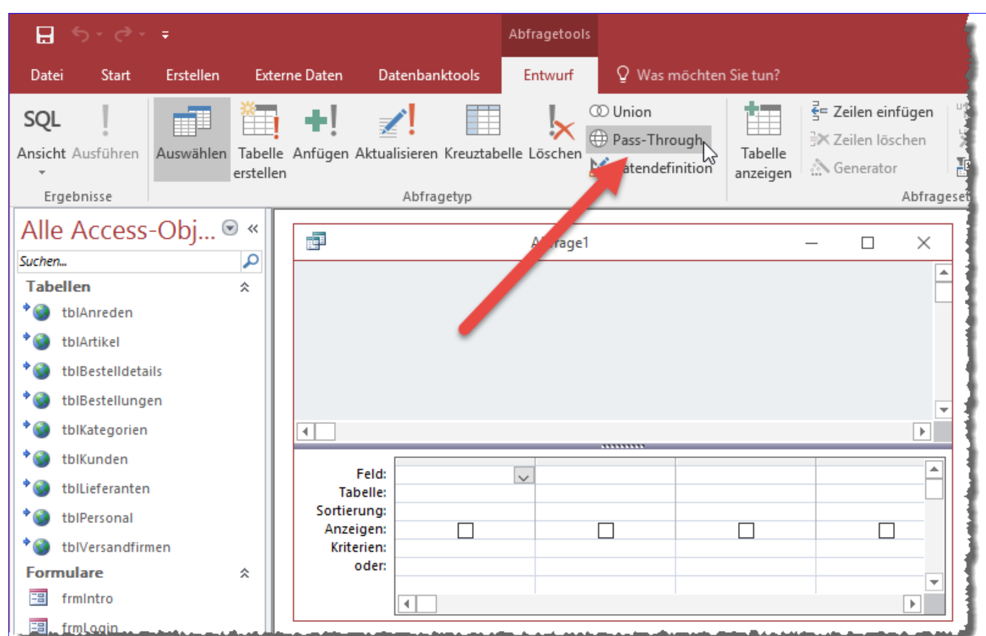

Bild 4: Umwandeln einer Abfrage in eine Pass-Through-Abfrage

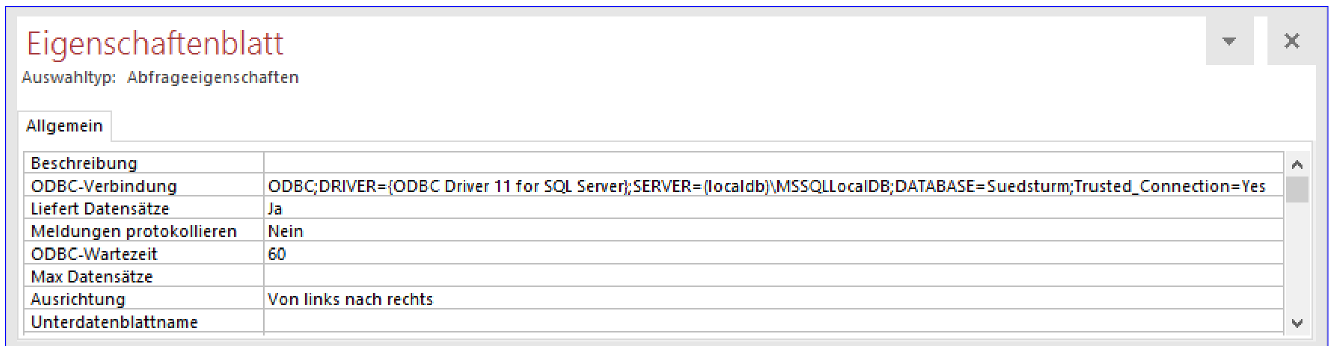

Bild 5: Eigenschaften der Pass-Through-Abfrage

neuen Eintrag namens **pAlleArtikel** unter dem Element **Suedsturm|Programmierbarkeit|Gespeicherte Prozeduren**.

Sie können diese testen, indem Sie in einem Abfragefenster die Anweisung **EXEC pAlleArtikel;** eingeben und diese mit **F5** ausführen.

Pass-Through-Abfrage erstellen

Nun erstellen wir in der Access-Datenbank eine Pass-Through-Abfrage, mit der wir über die gespeicherte Prozedur auf die Daten der Tabelle **tblArtikel** zugreifen wollen.

Legen Sie dazu eine neue, leere Abfrage in der Entwurfsansicht an und schließen Sie den automatisch erscheinenden Dialog **Tabelle anzeigen**. Nun klicken Sie auf den Ribbon-Eintrag **Entwurf|Abfragetyp|Pass-Through** (s. Bild 4).

Dies ändert vor allem die Ansicht der Abfrage, nämlich in die SQL-Ansicht. Hier sind nun zwei Schritte zu erledigen: Als Erstes geben Sie die Anweisung ein, mit der Sie zuvor schon die gespeicherte Prozedur im SQL Server Management Studio getestet haben, also **EXEC dbo.pAlleArtikel**. Außerdem müssen Sie der Abfrage noch mitteilen, woher sie ihre Daten beziehen soll. Dies erledigen wir per Hand über die Eigenschaft ODBC-Verbindung, der wir in unserem Beispiel die folgende Zeichenkette übergeben:

```
ODBC;DRIVER={ODBC Driver 11 for SQL
Server};SERVER=(localdb)\MSSQLLocalDB;DATABASE=Suedsturm;T
rusted_Connection=Yes
```

Schließlich muss noch die Eigenschaft **Liefert Datensätze** auf den Wert **Ja** eingestellt sein (s. Bild 5).

Anschließend können Sie die Datenblattansicht dieser Abfrage aktivieren und erhalten das gewünschte Ergebnis (s. Bild 6).

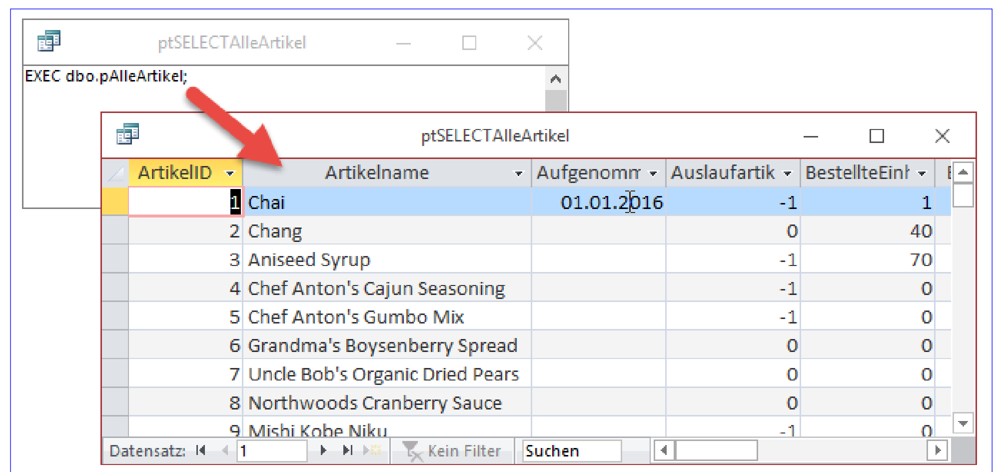

Bild 6: Ergebnis der Pass-Through-Abfrage

Per Recordset auf gespeicherte Prozedur zugreifen

Nun wollen wir auch per VBA über die Pass-Through-Abfrage auf die von der gespeicherten Prozedur gelieferten Daten zugreifen.

Auf diese Pass-Through-Abfrage greifen Sie wie folgt zu und durchlaufen dann ihre Datensätze über das zuvor gefüllte **Recordset**-Objekt:

```
Public Sub GespeicherteProzedurPerRecordset()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set db = CurrentDb
    Set rst = db.OpenRecordset("ptSELECTAlleArtikel")
    Do While Not rst.EOF
        Debug.Print rst!ArtikelID, rst!Artikelname
        rst.MoveNext
    Loop
    rst.Close
    Set rst = Nothing
    Set db = Nothing
End Sub
```

Hier geschieht rein oberflächlich nicht viel anderes, als wenn Sie wie weiter oben auf eine verknüpfte Tabelle zugreifen – mit dem Unterschied, dass wir hier den Namen der Pass-Through-Abfrage der **OpenRecordset**-Methode verwenden.

Vorbereitend auf weitere Varianten, in denen wir beispielsweise Parameter verwenden wollen, die in den Entwurf der Abfragen einfließen müssen, schauen wir uns nun eine Alternative an, bei der wir zunächst ein **QueryDef**-Objekt auf Basis der Pass-Through-Abfrage **ptSELECTAlleArtikel** erstellen und erst dann mit der **OpenRecordset**-Methode auf diesem Objekt ein Recordset erstellen:

```
Public Sub GespeicherteProzedurPerQueryDef()
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Dim rst As DAO.Recordset
    Set db = CurrentDb
    Set qdf = db.QueryDefs("ptSELECTAlleArtikel")
    Set rst = qdf.OpenRecordset()
    Do While Not rst.EOF
        Debug.Print rst!ArtikelID, rst!Artikelname
        rst.MoveNext
    Loop
    rst.Close
    Set rst = Nothing
    Set qdf = Nothing
    Set db = Nothing
End Sub
```

Was geschieht bei geänderter Verbindungszeichenfolge?

Wenn Sie die Anwendung auf Ihrem Rechner entwickeln und diese dann beispielsweise an den Kunden weitergeben, werden Sie voraussichtlich eine andere Verbindungszeichenfolge nutzen müssen.

In diesem Fall haben Sie ein Problem: Wir haben ja weiter oben beschrieben, wie Sie einer Pass-Through-Abfrage die Verbindungszeichenfolge mitteilen – und zwar durch direktes Eintragen dieser Zeichenfolge in die Eigenschaft **ODBC-Verbindung**. Wenn Sie nun eine ganze Reihe dieser Pass-Through-Abfragen nutzen, müssten Sie diese vor jeder Weitergabe an andere Anwender ändern. Diese Arbeit wollen Sie nicht wirklich erledigen, also automatisieren wir dies. Dazu erstellen wir eine Prozedur namens **PTAbfrageAktualisieren**, die wie in Listing 1 aussieht.

Die Funktion erwartet den Namen der Pass-Through-Abfrage und die Verbindungszeichenfolge als Parameter. Da diese den weiter oben vorgestellten Techniken zufolge mit der Funktion **VerbindungszeichenfolgeNachID** oder **Standardverbindungszeichenfolge** zusammengestellt wird, enthält sie bei Benutzung der SQL Server-Authentifizierung gegebenenfalls das Kennwort des aktuellen Benutzers.

```
Public Sub PTAbfrageAktualisieren(strPTAbfrage As String, strVerbindungszeichenfolge As String)
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Dim intStart As Integer
    Dim intEnde As Integer
    Dim strVerbindungOhnePWD As String
    intStart = InStr(1, strVerbindungszeichenfolge, ";PWD")
    If intStart > 0 Then
        strVerbindungOhnePWD = Left(strVerbindungszeichenfolge, intStart)
        intEnde = InStr(intStart + 1, strVerbindungszeichenfolge, ";")
        If Not intEnde = 0 Then
            strVerbindungOhnePWD = strVerbindungOhnePWD & Mid(strVerbindungszeichenfolge, intEnde + 1)
        End If
    Else
        strVerbindungOhnePWD = strVerbindungszeichenfolge
    End If
    Set db = CurrentDb
    Set qdf = db.QueryDefs(strPTAbfrage)
    qdf.Connect = strVerbindungOhnePWD
End Sub
```

Listing 1: Prozedur zum Aktualisieren von Pass-Through-Abfragen

Daher enthält die Funktion einige Zeilen Code, welche die Verbindungszeichenfolge auf das Auftreten des Ausdrucks **;PWD=** untersuchen. Ist dieser vorhanden, wird eine neue Verbindungszeichenfolge namens **strVerbindungOhnePWD** zusammengestellt, die alle Zeichen bis zum Auftreten von **;PWD=** enthält und gegebenenfalls alle Zeichen hinter dem nachfolgenden Semikolon.

Dazu ermittelt die Prozedur zunächst mit der **InStr**-Funktion die Position des ersten Auftretens der Zeichenfolge **;PWD**. Ist diese ungleich **0**, enthält die Zeichenfolge dieses Element. Eine neue **String**-Variable namens **strVerbindungOhnePWD** nimmt dann zunächst den Inhalt der Verbindungszeichenfolge bis zum Beginn von **;PWD** auf. Dann ermittelt sie die Position des ersten Semikolons hinter der Zeichenfolge **;PWD**, also das Ende des entsprechenden Name-Wert-Paares.

Hat das Ergebnis des entsprechenden Aufrufs der **InStr**-Funktion den Wert **0**, ist die Angabe von **PWD** das letzte Element der Verbindungszeichenfolge und es sind keine weiteren Schritte nötig – die Verbindungszeichenfolge exklusive **PWD**-Element befindet sich nun in der Variablen **strVerbindungOhnePWD**. Anderenfalls gibt es noch mindestens ein weiteres Element hinter dem Name-Wert-Paar mit dem Kennwort – also wird der komplette Rest hinter der ermittelten Position an den bisherigen Inhalt von **strVerbindungOhnePWD** angehängt.

Nachdem das PWD-Element aus der Verbindungszeichenfolge entfernt und in der Variablen **strVerbindungOhnePWD** gespeichert wurde, referenziert die Prozedur **PTAbfrageAktualisieren** die zugrunde liegende Pass-Through-Abfrage und stellt die Verbindungszeichenfolge mit der **Connect**-Eigenschaft auf den neuen Wert ein.

Die Beispieldatenbank enthält beispielsweise in der Tabelle **tblVerbindungszeichenfolgen** zwei Beispielverbindungen mit den ID-Werten **9** und **10**. Wenn Sie die erste davon verwenden wollen, rufen Sie die Prozedur wie folgt auf:

```
PTAbfrageAktualisieren "PTSELECTAlleArtikel", 
    VerbindungszeichenfolgeNachID(9)
```

```
Public Sub AlleAbfragenAktualisieren()
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Set db = CurrentDb
    For Each qdf In db.QueryDefs
        If InStr(Nz(qdf.Connect,""), "ODBC;") > 0 Then
            PTAbfrageAktualisieren qdf.Name, Standardverbindungszeichenfolge
        End If
    Next qdf
End Sub
```

Listing 2: Prozedur zum Aktualisieren aller Pass-Through-Abfragen mit der aktuellen Standardverbindungszeichenfolge

Wollen Sie dann wechseln, nutzen Sie die andere ID:

```
PTAbfrageAktualisieren "PTSELECTAlleArtikel", 
                         VerbindungszeichenfolgeNachID(10)
```

Sie können natürlich auch direkt die als Standardverbindungszeichenfolge deklarierte Verbindungszeichenfolge als Parameter angeben:

```
PTAbfrageAktualisieren "PTSELECTAlleArtikel", 
                           Standardverbindungszeichenfolge
```

Sie können sich zwischendurch den Entwurf der Pass-Through-Abfrage ansehen – die Eigenschaft **ODBC-Verbindung** enthält jeweils die entsprechende Verbindungszeichenfolge.

Alle Pass-Through-Abfragen aktualisieren

Wenn Sie mit dieser Prozedur alle Pass-Through-Abfragen aktualisieren wollen, durchlaufen Sie in einer weiteren Prozedur alle Einträge der Tabelle **MSysObjects**, deren Feld **Connect** einen Wert enthält, der mit **ODBC=** beginnt, und rufen von dort für jeden Eintrag die Prozedur **PTAbfrageAktualisieren** auf (s. Listing 2).

Wann aktualisieren?

Es stellt sich die Frage, wann man etwa die Verbindungszeichenfolge von Pass-Through-Abfragen aktualisiert. Die performanteste Lösung wäre wohl, wenn man dies nur dann durchführt, wenn sich die Standardverbindungszeichenfolge der Anwendung ändert. Wenn Sie also etwa die Instanz der Anwendung auf Ihrem Rechner auf die Weitergabe zum Kunden vorbereiten, könnten Sie einfach die Standardverbindungszeichenfolge ändern (etwa mithilfe des Formulars **frmVerbindungszeichenfolgen**, das wir im Beitrag **SQL Server-Tools**, **www.access-im-unternehmen.de/1061**, vorstellen) und danach die Prozedur **AlleAbfragenAktualisieren** aufrufen.

Gegebenenfalls fügen Sie den Aufruf dieser Prozedur sogar zu der Prozedur hinzu, die im Formular die aktuelle Verbindung als Standardverbindung einstellt.

Sie könnten die jeweilige Pass-Through-Abfrage auch immer dann aktualisieren, wenn Sie überhaupt ein Recordset auf Basis einer solchen Abfrage erstellen. Wenn Sie gespeicherte Prozeduren mit Parametern verwenden, müssen Sie die Abfrage ohnehin aktualisieren – dann können Sie auch gleich die Verbindungszeichenfolge mit hinzu nehmen.

Recordset-Funktionen

Da Sie gegebenenfalls regelmäßig Recordsets für die verschiedenen Anforderungen benötigen, stellen wir dazu geeignete Funktionen zusammen. Diese können Sie dann beispielsweise einsetzen, wenn Sie Daten per VBA durchlaufen wollen, aber Sie können diese auch der Recordset-Eigenschaft von Formularen, Berichten, Kombinationsfeldern oder Listenfeldern zuweisen. Wir stellen zwei verschiedene Funktionen vor:

```
Public Function SPRecordset(strStoredProcedure As String, strVerbindungszeichenfolge As String) As DAO.Recordset
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Set db = CurrentDb
    Set qdf = db.CreateQueryDef("")
    With qdf
        .Connect = strVerbindungszeichenfolge
        .SQL = "EXEC " & strStoredProcedure
        Set SPRecordset = .OpenRecordset
    End With
    Set db = Nothing
End Function
```

Listing 3: Diese Funktion liefert ein Recordset basierend auf einer gespeicherten Prozedur und einer entsprechenden Verbindungszeichenfolge.

- **SPRecordset**: Liefert ein einfaches Recordset ohne Einsatz von Parametern

- **SPRecordsetMItParameter**: Liefert ein Recordset auf Basis einer gespeicherten Prozedur mit Parametern.

Recordset aus gespeicherter Prozedur ohne Parameter

Die Funktion **SPRecordset** soll ein Recordset basierend auf der per Funktionsparameter angegebenen gespeicherten Prozedur für eine ebenfalls per Funktionsparameter übergebene Verbindung zurückliefern.

Um das Ergebnis an ein **Recordset**-Objekt zu übergeben, ruft die Prozedur die **OpenRecordset**-Methode für die temporäre Pass-Through-Abfrage mit der gespeicherten Prozedur auf. Diese Variante arbeitet mit gespeicherten Prozeduren, die keine Parameter erwarten (s. Listing 3).

Wenn Sie die Funktion ausprobieren und das zurückgelieferte **Recordset**-Objekt weiterverwenden möchten, können Sie dies mit einer Prozedur wie der aus Listing 4 erledigen. Diese deklariert ihrerseits ein **Recordset**-Objekt und weist diesem das Ergebnis der Funktion **SPRecordset** zu. Anschließend durchläuft die Prozedur noch die Datensätze des zurückgelieferten Recordsets.

Sie können die Funktion auch innerhalb eines Formulars testen. Dazu versehen Sie ein leeres Formular zunächst mit der verknüpften Tabelle als Datenherkunft. Damit können Sie dann die gebundenen Felder ganz einfach aus der Feldliste in das Formular ziehen (s. Bild 7). Das Formular würde nun beim Wechseln in die Formularansicht die Daten der Tabelle **tblArtikel** anzeigen – allerdings aus der verknüpften Tabelle. Wir wollen aber die Daten aus der gespeicherten Prozedur liefern. Dazu fügen wir dem Formular eine Ereignisprozedur hinzu, die durch das Ereignis **Beim Laden** ausgelöst wird:

```
Public Sub TestSPRecordset()
    Dim rst As DAO.Recordset
    Set rst = SPRecordset("dbo.spSELECTAlleWarengruppen", Standardverbindungszeichenfolge)
    Do While Not rst.EOF
        Debug.Print rst!WarengruppeID
        rst.MoveNext
    Loop
End Sub
```

Listing 4: Test der Funktion zum Holen eines Recordsets auf Basis einer gespeicherten Prozedur

SQL Server-Tools

Wenn Sie mit Access als Frontend arbeiten und mit dem SQL Server als Backend, kommen Sie nicht umhin, sich mit Verbindungszeichenfolgen, dem Verknüpfen von Tabellen oder dem Ausprobieren und Absetzen von SQL-Abfragen direkt an den SQL Server auseinanderzusetzen. Dieser Beitrag stellt drei Tools vor, die Sie direkt in Ihre Access-Anwendung importieren können und mit denen Sie komfortabel Verbindungen definieren, Tabellen verknüpfen und SQL-Abfragen an den SQL Server schicken können.

Verbindungszeichenfolgen verwalten

Optimalerweise gibt es immer nur eine Verbindungszeichenfolge innerhalb einer Datenbank. Wenn Sie jedoch eine Anwendung entwickeln, von der Sie immer wieder Zwischenstände an die Benutzer verteilen, sei es zum Testen oder für den Produktivbetrieb, werden Sie mehrere verschiedene Verbindungszeichenfolgen nutzen müssen – also etwa eine für das Entwicklungssystem und eine für die Produktivsysteme.

Üblicherweise macht das Zusammenstellen von Verbindungszeichenfolgen immer wieder Mühe, denn wer kann sich schon die verschiedenen Namen von SQL Server-Instanzen, die Parameter für die Windows- und die SQL Server-Authentifizierung oder die in der Verbindungszeichenfolge anzugebene Bezeichnung für den Treiber merken?

Hier setzt das Formular **frmVerbindungszeichenfolgen** aus Bild 1 an, das Sie ganz einfach in die gewünschte Datenbank ziehen können – nebst einigen weiteren Objekten, die wir weiter unten noch aufführen.

In diesem Formular können Sie mit den Schaltflächen im unteren Bereich die aktuelle Verbindungszeichenfolge testen, die Verbindungszeichenfolge als Standard setzen, eine neue Verbindungszeichenfolge erstellen, die aktuelle Verbindungszeichenfolge als neue Verbindungszeichenfolge kopieren oder den aktuellen Eintrag einfach löschen.

Bild 1: Das Formular **frmVerbindungszeichenfolgen**

Bild 2: Auswahl einer Verbindungszeichenfolge über die Bezeichnung

Darüber finden Sie beispielsweise das Kombinationsfeld **Schnellauswahl**, mit dem Sie direkt eines der bereits angelegten Elemente auswählen und seine Daten im Formular anzeigen können (s. Bild 2).

Mit dem Feld **Bezeichnung** legen Sie die Bezeichnung fest, über die Sie die einzelnen Einträge per Schnellauswahl selektieren können. Das Feld **SQL Server** erwartet den Namen des Servers, im Falle der **LocalDB**-Standardinstanz also beispielsweise **(localdb)\MSSQLLocalDB**.

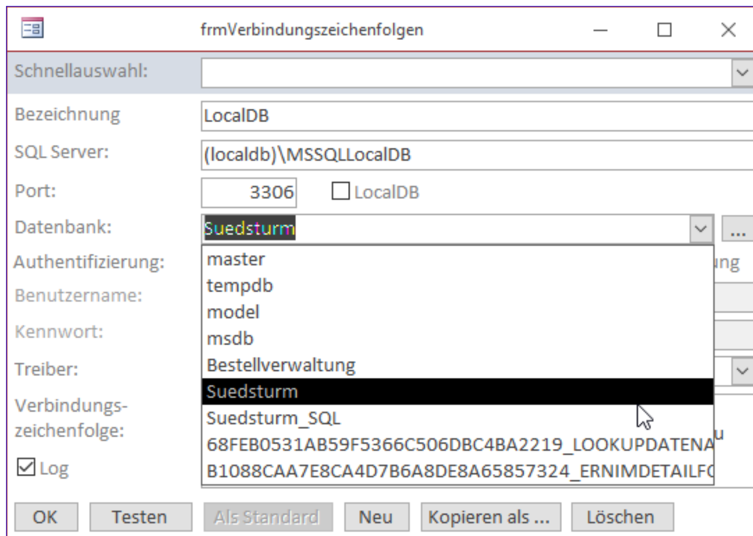

Bild 3: Auswahl einer Datenbank

Die Option **Port** erwartet die Angabe des Ports, über den auf den SQL Server zugegriffen wird. Dies ist standardmäßig der Wert **3306**. Das Kontrollkästchen **LocalDB** können Sie aktivieren, wenn Sie über eine **LocalDB**-Instanz statt über eine SQL Server-Instanz auf die Datenbank zugreifen wollen.

Das darunter liegende Kombinatonsfeld **Datenbank** erlaubt die Auswahl der Datenbanken für den aktuellen SQL Server. Um dieses Kombinationsfeld zu füllen, klicken Sie auf die Schaltfläche rechts vom Kombinationsfeld. Anschließend können Sie unter den verfügbaren Datenbanken wählen (s. Bild 3).

Die folgenden Eingestellungen betreffen die Sicherheit. Hier wählen Sie zunächst, ob Sie die Windows-Authentifizierung oder die SQL Server-Authentifizierung nutzen wollen. Zur Erläuterung nochmal der Unterschied: Wenn Sie die Windows-Authentifizierung nutzen, werden die Windows-Anmeldedaten als Benutzerdaten verwendet. Dies ist praktischer, da Sie sich auf diese nicht noch einmal separat mit weiteren Daten am SQL Server anmelden müssen. Die SQL Server-Authentifizierung verlangt genau dies: In diesem Fall werden im SQL Server für jeden Benutzer, der auf die Daten zugreifen soll, nochmals neue Benutzerdaten inklusive Kennwort angelegt. Diese müssen dann entweder beim Zugriff angegeben werden oder Sie speichern diese irgendwo in der Frontend-Datenbank, um sie für den Zugriff in die Verbindungszeichenfolge zu integrieren.

Wenn Sie die SQL Server-Authentifizierung wählen, können Sie die Zugangsdaten, also den **Benutzernamen** und das **Kennwort**, in die beiden gleichnamigen Felder im Formular **frmVerbindungszeichenfolgen** eintragen.

Mit dem Kombinationsfeld **Treiber** wählen Sie den Treiber für den Zugriff auf die Datenbank aus. Hier können Sie nicht nur SQL Server-Treiber angeben, sondern auch Treiber etwa für MySQL (s. Bild 4). Die Treiber werden übrigens in der Tabelle **tblTreiber** gespeichert, die Sie manuell pflegen müssen.

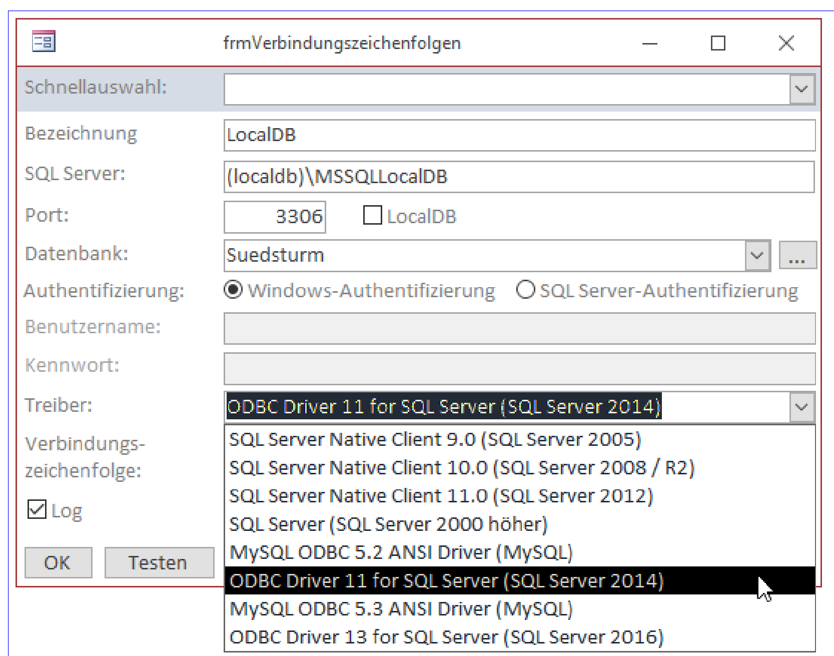

Bild 4: Auswahl des gewünschten Treibers

Mit dem Kontrollkästchen **Log** geben Sie an, ob der Parameter, der festlegt, ob die Interaktion mit dem Server geloggt werden soll, hinzugefügt werden soll. Dieser Teil lautet dann so:

```
LOG_QUERY=1
```

Diese Option ist nur für MySQL vorgesehen, nicht für den SQL Server.

Das Textfeld **Verbindungszeichenfolge** zeigt dann jeweils den aktuellen, den Angaben entsprechenden String für die Verbindungszeichenfolge.

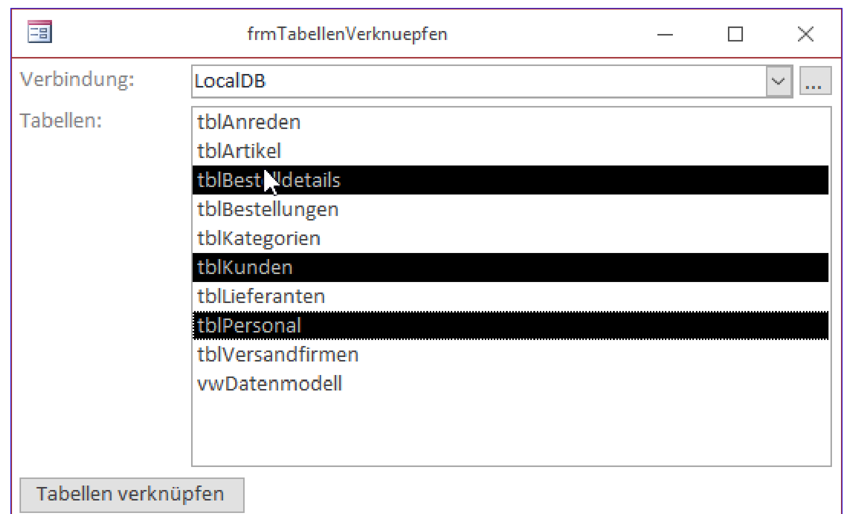

Bild 5: Verknüpfen von Tabellen

Mit einem Klick auf die Schaltfläche **Testen** prüfen Sie, ob die Verbindungszeichenfolge funktioniert. Ist dies der Fall, haben Sie die Grundlage geschaffen, mit den übrigen beiden Tools entweder Verknüpfungen zu Tabellen herzustellen oder direkt SQL-Befehle an die festgelegte Datenbank zu schicken.

Außerdem können Sie die in den Beiträgen **RDBMS-Zugriff per VBA: Verbindungen** (**www.access-im-unternehmen.de/1054**) und **RDBMS-Zugriff per VBA: Daten abfragen** (**www.access-im-unternehmen.de/1062**) vorgestellten Techniken nutzen, die auf die Funktionen **Standardverbindungszeichenfolge** oder **VerbindungszeichenfolgeNachID** zugreifen. Diese nutzen nämlich die Tabelle **tblVerbindungszeichenfolgen**, die auch vom Formular **frmVerbindungszeichenfolgen** gefüllt wird.

Tabellen verknüpfen

Das zweite sehr praktische Formular der Tool-Sammlung ist das Formular **frmTabellenVerknuepfen** aus Bild 5. Es erlaubt im oberen Bereich die Auswahl einer der in der Tabelle **tblVerbindungszeichenfolgen** gespeicherten Verbindungen. Nach der Auswahl liest sie automatisch alle Tabellen der dort festgelegten Datenbank ein und zeigt diese im Listenfeld mit der Bezeichnung **Tabellen** an. Rechts neben dem Kombinationsfeld finden Sie noch eine kleine Schaltflächen mit drei Punkten (**...**) als Beschriftung. Wenn Sie darauf klicken, öffnen Sie automatisch das bereits erwähnte Formular **frmVerbindungszeichenfolgen**.

Dieses erscheint dann neben dem aufrufenden Formular und zeigt die Daten der aktuell im Formular **frmTabellenVerknuepfen** ausgewählten Verbindungszeichenfolge an. Sie können die Daten nun bearbeiten, aber auch über die Schnellauswahl eine andere Verbindungszeichenfolge im Formular **frmVerbindungszeichenfolgen** auswählen. Wenn Sie das Formular **frmVerbindungszeichenfolgen** dann schließen, wird die gewählte Verbindungszeichenfolge im entsprechenden Kombinationsfeld des Formulars **frmTabellenVerknuepfen** angezeigt (s. Bild 6).

Wenn die Datenbank noch keine verknüpften Tabellen enthält, können Sie diese nun hinzufügen. Dazu markieren Sie einfach die zu verknüpfenden Tabellen im Listenfeld. Nach Abschluss der Auswahl klicken Sie auf die Schaltfläche **Tabellen verknüpfen**, um die Verknüpfungen hinzuzufügen. Das Listenfeld hat ein paar interessante Features. Das erste ist, dass Sie damit mehrere Tabellen gleichzeitig auswählen können, und zwar wie bei der Auswahl im Windows Explorer. Um mehrere zusammen-

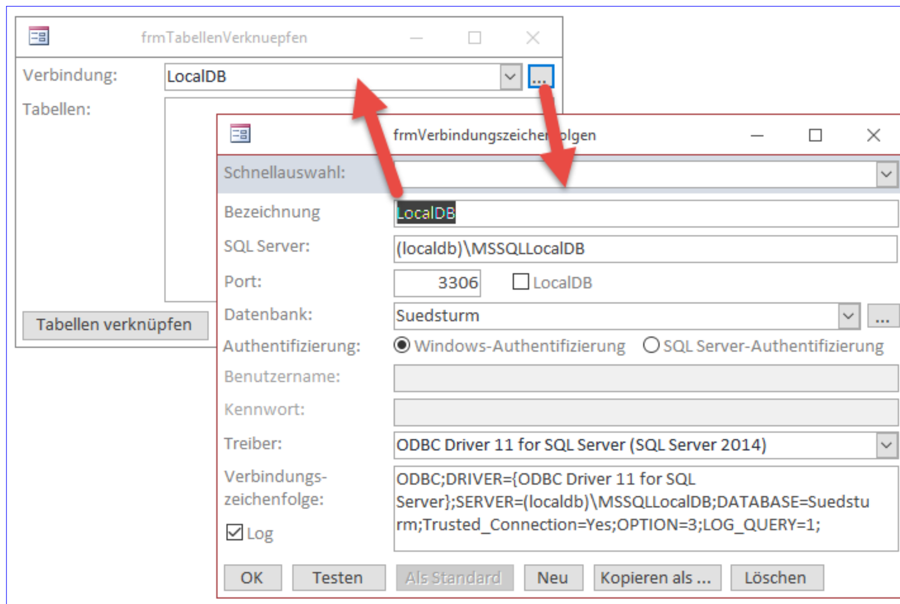

Bild 6: Öffnen des Formulars **frmVerbindungszeichenfolgen** zur Ansicht und Auswahl einer Verbindungszeichenfolge

hängende Einträge zu selektieren, klicken Sie den ersten Eintrag an und dann bei gedrückter Umschalttaste den letzten Eintrag. Wenn Sie nicht zusammenhängende Einträge auswählen möchten, halten Sie einfach die **Strg**-Taste gedrückt. Dann können Sie durch Anklicken einzelne Einträge auswählen, aber auch wieder aus der Auswahl entfernen.

Das zweite, wesentlich interessantere Feature offenbart sich erst, wenn Sie das erste Mal einige Tabellenverknüpfungen zur Datenbank hinzugefügt haben. Wenn Sie dann das Formular **frmTabellenVerknuepfen** öffnen, hebt es die Tabellennamen der Tabellen, die bereits als Verknüpfung in der Datenbank vorhanden sind, direkt mit einer entsprechenden Markierung hervor (s. Bild 7). Wozu das nützlich ist? Wenn Sie mit einer Frontend-Datenbank auf eine SQL Server- oder auch MySQL-Datenbank zugreifen, die sehr viele Tabellen enthält, brauchen Sie nicht immer Zugriff auf alle Tabellen des Backends.

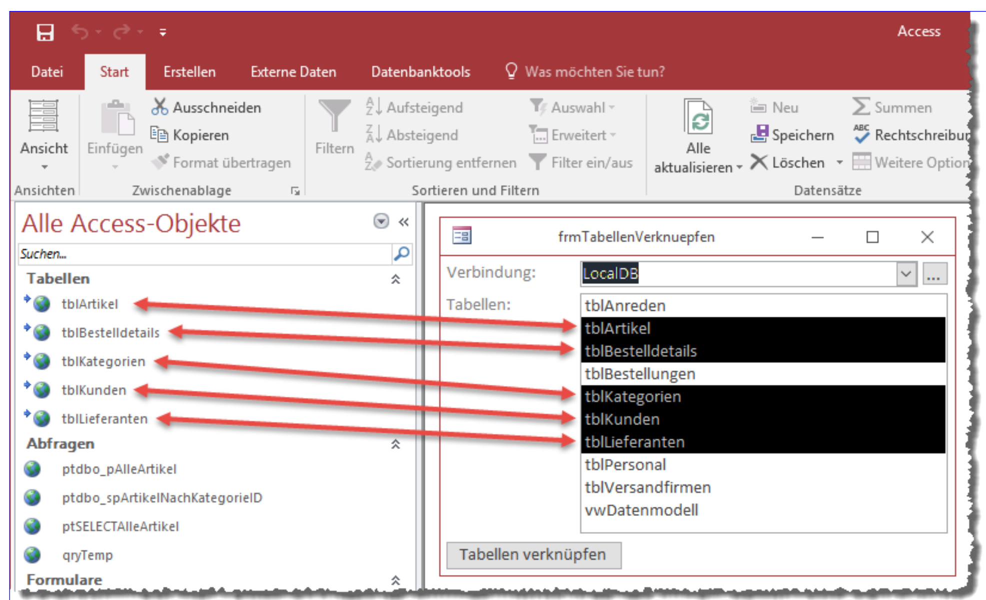

Bild 7: Das Listenfeld zeigt die bereits vorhandenen Verknüpfungen schwarz hinterlegt an.

Bei mir ist das beispielsweise der Fall bei der Kundenverwaltung, die über einige Verknüpfungen zu den Tabellen meines Webshopsystems verfügt – aber ich muss längst nicht auf alle Backendtabellen zugreifen! Wenn ich dann das Formular **frmTabellenverknuepfungen** aufrufe, um die Verknüpfungen beispielsweise nach einer Änderung der Adresse des Internetservers wieder zu aktualisieren, dann brauche ich nicht immer alle Tabellen neu auszuwählen, sondern das Listenfeld selektiert mir die bisher verwendeten Tabellen direkt vor. Auf diese Weise brauche ich nur noch einen Mausklick auf die Schaltfläche **Tabellen verknüpfen**, um die Tabellen neu einzubinden.

Zip-Formular

Wie das Zippen unter Access mit Bordmitteln funktioniert, haben Sie im Artikel »Zippen mit Access« erfahren. Im vorliegenden Beitrag schauen wir uns nun an, wie Sie die VBA-Funktionen zum Packen, Entpacken und Löschen von Elementen aus Zip-Dateien von einem Formular aus nutzen können, sodass auch reine Benutzer einer Anwendung damit arbeiten können.

Formular zum Verwalten einfacher Zip-Funktionen

Im Beitrag **Zippen mit Access** (**www.access-im-unternehmen.de/1064**) haben wir einige Funktionen vorgestellt, mit denen Sie mit Zip-Dateien arbeiten können. Damit lassen sich neue Zip-Dateien erstellen, Dateien hinzufügen, Dateien aus Zip-Dateien extrahieren und die enthaltenen Dateien auflisten.

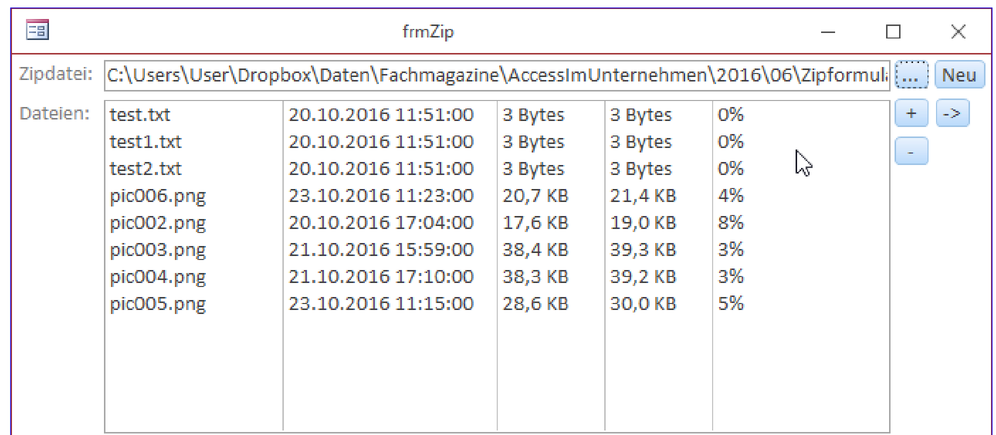

Bild 1: Aussehen des fertigen Zip-Formulars

Schließlich können Sie die enthaltenen Dateien auch aus dem Zip-Archiv löschen. Das Formular soll wie in Bild 1 aussehen. Das Formular bietet ein Textfeld, das den Namen der aktuellen Zip-Datei anzeigt.

Mit der Schaltfläche mit den drei Punkten (...) können Sie einen **Datei öffnen**-Dialog anzeigen, mit dem Sie die zu öffnende Zip-Datei auswählen können. Alternativ klicken Sie auf die **Neu**-Schaltfläche, die einen Dialog zum Angeben des Namens für eine neue Zip-Datei öffnet.

Nach dem Öffnen einer bestehenden Zip-Datei soll das Formular die enthaltenen Dateien im Listenfeld im unteren Bereich anzeigen. Das Listenfeld soll die Auswahl eines oder mehrerer Elemente ermöglichen, die dann wahlweise in einen noch zu spezifizierenden Ordner entpackt (Schaltfläche **->**) oder aus dem Archiv entfernt werden (Schaltfläche **-**).

Die Plus-Schaltfläche (**+**) erlaubt es schließlich, einen weiteren Dateiauswahl-Dialog anzuzeigen, mit dem Sie die zur Zip-Datei hinzuzufügenden Dateien auswählen können. Diese werden dann nach dem Hinzufügen direkt im Listenfeld angezeigt. Nun beginnen wir mit dem Bau dieses Formulars.

Tabellen der Anwendung

Die Anwendung verwendet nur eine einzige Tabelle namens **tblDateien**, die wie in Bild 2 aussieht. Wie diese gefüllt wird, erfahren Sie im Beitrag **Zippen mit Access** (**www.access-im-unternehmen.de/1064**). Gegenüber der dort verwendeten Tabelle haben wir noch zwei Felder hinzugefügt.

Diese heißen **GepackteGroesseText** und **UngepackteGroesseText** und nehmen nicht den per Code ermittelten Zahlenwert für die Dateigröße auf, sondern den kompletten gelieferten Wert inklusive der Einheit, also etwa **Bytes** oder **KB**.

tblDateien

Feldname	Felddatentyp	Beschreibung (optional)
DateiID	AutoWert	Primärschlüsselfeld
Dateiname	Kurzer Text	Name der Datei
ZuletztGeaendert	Datum/Uhrzeit	Letztes Änderungsdatum
GepackteGroesse	Zahl	Größe der gepackten Dateien
UngepackteGroesse	Zahl	Größe der ungepackten Dateien
Ratio	Zahl	Verhältnis gepackt/ungepackt
GepackteGroesseText	Kurzer Text	Größe der gepackten Datei mit Einheit
UngepackteGroesseText	Kurzer Text	Größe der ungepackten Datei mit Einheit

Bild 2: Tabelle zum Speichern der Dateiinformationen der Dateien aus dem Zip-Archiv

Entwurfsansicht

In der Entwurfsansicht sieht das Formular wie in Bild 3 aus. Da das Formular keine Datenherkunft besitzt und somit auch nicht zum Blättern durch Datensätze verwendet wird, stellen wir die Eigenschaften **Navigationsschaltflächen**, **Datensatzmarkierer**, **Trennlinien** und **Bildlaufleisten** auf den Wert **Nein** ein. Die Eigenschaft **Automatisch zentrieren** erhält den Wert **Ja**.

Das Textfeld erhält die Bezeichnung **txtZipdatei**, das Listenfeld soll **lstDateien** heißen. Die Schaltflächen statten Sie mit den Namen **cmdDateiauswahl**, **cmdNeu**, **cmdDateiHinzufuegen**, **cmdDateiEntfernen** und **cmdEntpacken** aus. Für alle Schaltflächen hinterlegen wir entsprechende Ereignisprozeduren.

Steuerelemente verankern

Damit wir mit dem Vergrößern des Formulars auch die entscheidenden Steuerelemente vergrößern können, stellen wir für das Textfeld **txtZipdatei** die Eigenschaft **Horizontaler Anker** auf **Beide** ein.

Das Listenfeld **lstDateien** soll nach rechts und nach unten vergrößerbar sein, also stellen wir nicht nur die Eigenschaft **Horizontaler Anker**, sondern auch die Eigenschaft **Vertikaler Anker** auf **Beide** ein.

Beachten Sie, dass die entsprechenden Eigenschaften der mit den Steuerelementen verknüpften Bezeichnungsfelder automatisch geändert werden. Sie müssen also für das Bezeichnungsfeld des Textfeldes die Eigenschaft **Horizontaler Anker** wieder auf **Links** und für das Bezeichnungsfeld die Eigenschaft **Horizontaler Anker** auf **Links** und **Vertikaler Anker** auf **Oben** zurücksetzen.

Wenn Sie das Formular nun in der Formularansicht nach rechts vergrößern, überdecken das Textfeld und das Listenfeld die Schaltflächen. Diese sollen mit nach rechts verschoben werden. Also stellen Sie die Eigenschaft **Horizontaler Anker** aller Schaltflächen auf den Wert **Rechts** ein.

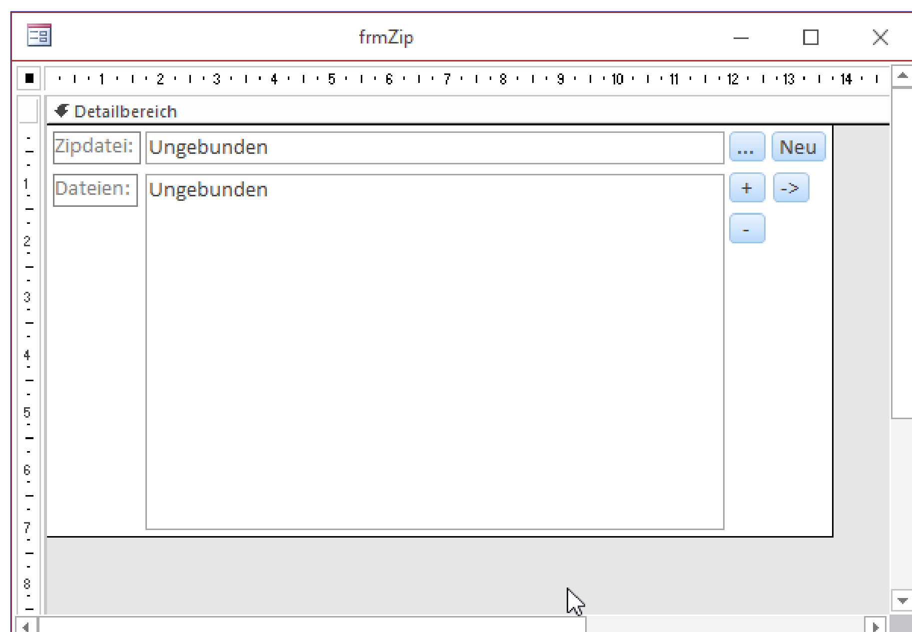

Bild 3: Das Zip-Formular in der Entwurfsansicht

```
Private Sub Form_Load()
    Dim db As DAO.Database
    Set db = CurrentDb
    Me!txtZipdatei = Null
    db.Execute "DELETE FROM tblDateien", dbFailOnError
    Me!lstDateien.Requery
End Sub
```

Listing 1: Prozedur, die durch das Ereignis **Beim Laden** ausgelöst wird

Beim Laden des Formulars

Das Laden des Formulars löst die Ereignisprozedur **Form_Load** aus, die wie in Listing 1 aussieht. Die Prozedur erledigt einige vorbereitende Arbeiten. Gegebenenfalls wurde bereits zuvor eine Zip-Datei eingelesen oder angelegt, daher könnten sich noch Datensätze in der Tabelle **tblDateien** befinden.

Diese sollen mit einer entsprechenden **DELETE**-Anweisung gelöscht werden. Außerdem soll das Textfeld **txtZipdatei** geleert und das Listenfeld aktualisiert werden, sodass es keine Datensätze mehr anzeigt.

Datensatzherkunft des ListenBild 4feldes

Das Listenfeld soll die Daten der Tabelle **tblDateien** anzeigen, allerdings nicht alle Felder davon und außerdem in einer bestimmten Reihenfolge. Bild 4 zeigt, wie die Abfrage aufgebaut ist, um die gewünschten Daten zu liefern.

Die Abfrage verwendet die beiden Felder **GepackteGroesseText** und **UngepackteGroesseText** statt **GepackteGroesse** und **UngepackteGroesse**, da diese den Originalwert aus dem Zip-Archiv enthalten, der gleichzeitig die Einheit wie **Bytes** oder **KB** mitliefert. Die Felder **GepackteGroesse** und **UngepackteGroesse** enthalten die Zahlenwerte ohne Einheit, allerdings wird hieraus nicht deutlich, um welche Einheit es sich jeweils handelt. Das Ergebnis der Abfrage soll nach dem Wert des Feldes **Dateiname** sortiert werden.

Die Abfrage speichern Sie unter dem Namen **qryDateien** (s. Bild 4). Dann weisen Sie diesen Namen der Eigenschaft **Datensatzherkunft** des Listenfeldes zu. Die Eigenschaft **Spaltenanzahl** stellen Sie auf den Wert **6** ein, die Eigenschaft **Spaltenbreiten** auf den Wert **0cm;;4cm;2cm;2cm**.

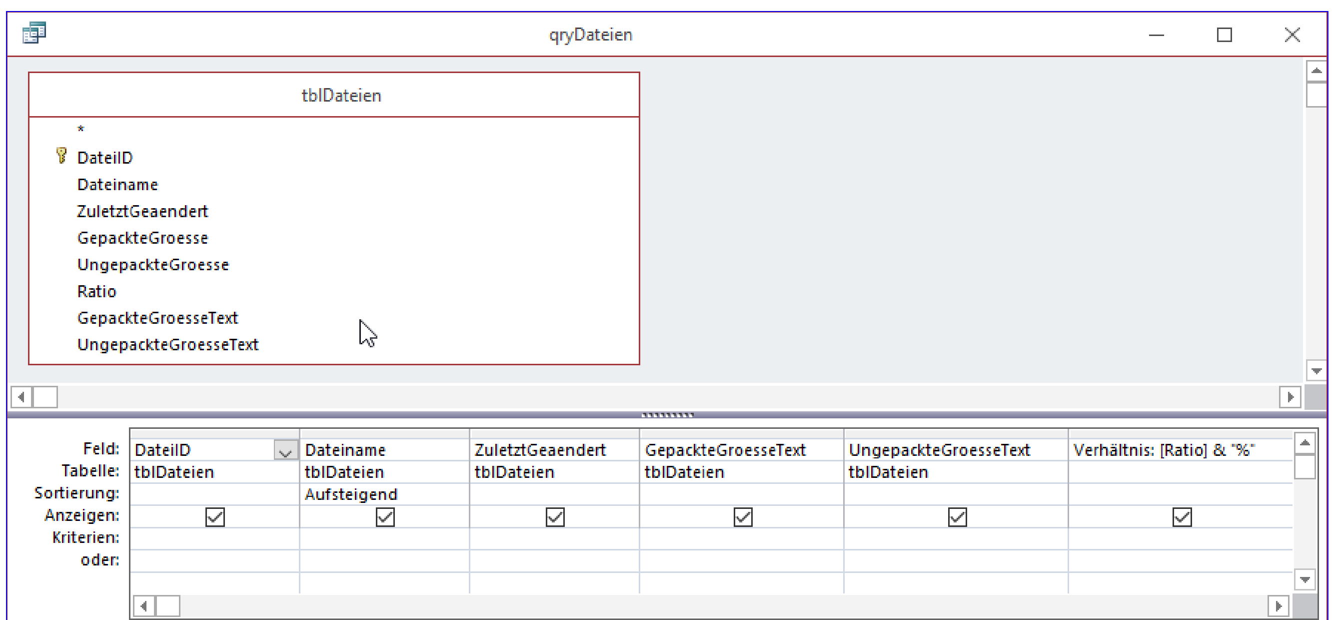

Bild 4: Entwurf der Abfrage **qryDateien**