HAUFE.
Ausgabe 05/2016

ACCESS

IM UNTERNEHMEN

ACCESS UND LOCALDB

Nutzen Sie den SQL Server light und greifen Sie von Access aus auf die darin gespeicherten Daten zu (ab S. 55).

In diesem Heft:

SQL Server light

Bereits vor einigen Versionen hat Microsoft mit LocalDB eine Version des SQL Servers herausgebracht, die viel leichtgewichtiger ist als der große Bruder. Wer mal eben eine SQL Server-Datenbank zum Testen benötigt oder auch ein lokales Backend für seine Access-Datenbank sucht, die mehr als zwei Gigabyte Speicherplatz bietet, findet hier eine tolle Möglichkeit. Aber: Wo Licht ist, ist auch Schatten!

Im Falle der SQL Server-Variante LocalDB kommt es auf den Anwendungszweck an, ob es sich hier um eine passende Alternative handelt. Die große Einschränkung ist nämlich, dass Sie LocalDB – wie der Name schon sagt – nur lokal einsetzen können, also nicht etwa von einem anderen Rechner über das Netzwerk. Aber seien wir ehrlich: Viele der Leser von Access im Unternehmen dürften nicht nur Datenbankanwendungen für Kunden entwickeln, sondern auch selbst welche betreiben – beispielsweise als Auftrags- oder Kundenverwaltung. Und mal eben eine Beispieldatenbank auf Basis des SQL Servers ausprobieren, ohne gleich die komplette Version installieren zu müssen, ist auch reizvoll. Denn LocalDB ist einfach nur ein Programm und kein Dienst wie der SQL Server und ist ebenso schnell installiert wie in Betrieb genommen.

Nun: Im Beitrag **Access und LocalDB** erfahren Sie, wie Sie LocalDB installieren und wie Sie von Access aus darauf zugreifen können (ab S. 55).

Wenn Sie schon mit LocalDB und SQL Server-Datenbanken experimentieren wollen, kann es nicht schaden, einen Weg zum schnellen Kopieren von Daten von einer Access-Datenbank zum SQL Server zu finden. Im Beitrag **Daten von Access zum SQL Server kopieren** haben wir uns verschiedene Möglichkeiten angesehen (ab S. 41). Vor allem kümmern wir uns dort um das Kopieren der Daten unter Beibehaltung der automatisch vergebenen Primärschlüsselwerte, was nicht trivial ist.

Voraussetzung für das Kopieren der reinen Daten ist jedoch, dass bereits eine SQL Server-Datenbank mit einem entsprechenden Datenmodell vorliegt. Wie Sie dieses unter Zuhilfenahme des SQL Server Management Studio für eine LocalDB-Datenbank erledigen, zeigt der Beitrag **SQL Server-Datenbank erstellen** ab S. 34.

Eine gute Hilfe für den Zugriff auf SQL Server- und auch LocalDB-Datenbanken bietet der Beitrag **RDBMS-Zugriff per VBA: Verbindungen**, der zeigt, wie Sie sich von Access aus per VBA mit einer SQL Server-Datenbank verbinden (ab S. 25).

Aber wir beschäftigen uns in dieser Ausgabe auch noch mit anderen Themen als mit dem SQL Server. Zum Beispiel mit Standardwerten: Wenn ein Benutzer einen neuen Datensatz anlegt, beginnt er in der Regel bei Null. Er muss also die Daten für alle Felder selbst eintragen beziehungsweise auswählen. In manchen Fällen findet man auch Felder, die mit einem Standardwert belegt sind. Wenn Sie es etwas komfortabler haben wollen, können Sie auch dynamische Standardwerte verwenden und beispielweise festlegen, dass Steuerelemente immer mit den zuletzt verwendeten Einträgen vorbelegt werden. Unser Beitrag **Dynamische Standardwerte** zeigt ab S. 2, wie Sie dabei die zuletzt verwendeten Werte auch noch in einer Tabelle speichern, damit diese beim nächsten Starten der Anwendung immer noch zur Verfügung stehen.

In der vorherigen Ausgabe haben wir uns umfassend dem Thema XML-Export gewidmet. Dies ergänzen wir in dieser Ausgabe mit dem Artikel **XML-Export: CDATA** (ab S. 62). Hier erfahren Sie, wie Sie Daten exportieren und Sonderzeichen wie das Kleiner- oder Größer-Zeichen beibehalten.

Der Beitrag **Rechnungsbericht** zeigt, wie Sie einen Bericht zur Darstellung von Rechnungen erstellen (ab S. 11), und ab S. 66 erfahren Sie, wie Sie den **Windows Explorer per VBA steuern** und ihn beispielsweise dafür nutzen, direkt nach dem Exportieren einer Datei das passende Verzeichnis zu öffnen.

Nun aber: Viel Spaß beim Lesen!

Ihr Michael Forster

Dynamische Standardwerte

In vielen Fällen können Sie dem Benutzer bei der Nutzung Ihrer Anwendung durch die Vorgabe von Standardwerten Arbeit abnehmen. Oft kristallisiert sich aber erst später heraus, welche die gewünschten Standardwerte sind oder diese unterscheiden sich je nach Benutzer. Dann wäre es praktisch, wenn die Benutzer selbst die Standardwerte für das Anlegen neuer Datensätze vorgeben könnten. Vielleicht möchten Sie sogar, dass der nächste neue Datensatz die Werte des vorherigen Datensatzes als Standardwerte übernimmt? Wie dies gelingt, zeigt der vorliegende Beitrag.

Als Beispielformular nutzen wir ein einfaches Detailformular namens **frmArtikel**, welches die Daten der Tabelle **tblArtikel** der Beispieldatenbank anzeigt.

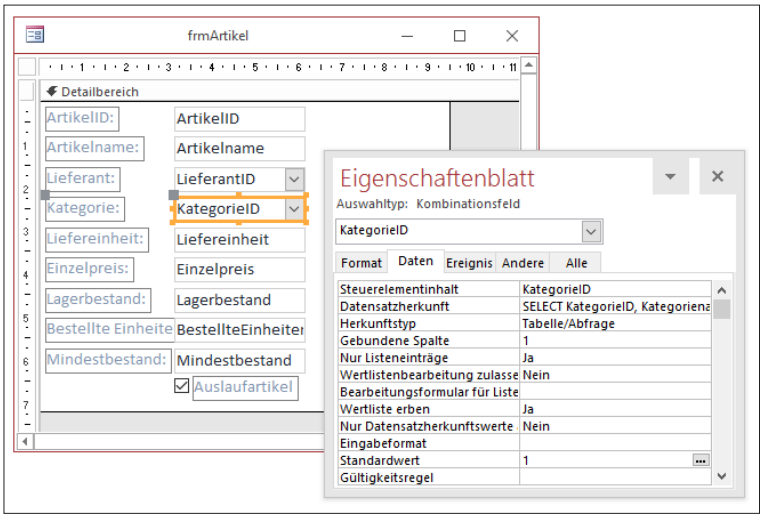

Bild 1: Formular mit der Standardwert-Eigenschaft im Entwurf

Angenommen, die Datenbank enthält überwiegend Artikel einer bestimmten Kategorie, beispielsweise der Kategorie **Getränke** mit dem Primärschlüsselwert **1**, dann könnte es dem Benutzer einige Arbeit sparen, wenn dieser Wert beim Anlegen eines neuen Datensatzes bereits voreingestellt wird. Dies erledigen Sie über die Eigenschaft **Standardwert** des Kombinationsfeldes zur Anzeige der Kategorie im Entwurf des Formulars (s. Bild 1).

Wenn Sie dann in die Formularansicht wechseln und einen neuen Datensatz anzeigen, erscheint im Kombinationsfeld **Kategorie** bereits der voreingestellte Eintrag (s. Bild 2). Dies ändert sich allerdings auch nicht beim Anlegen weiterer Datensätze.

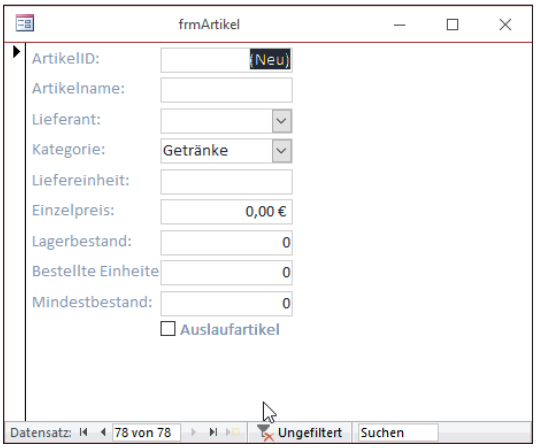

Bild 2: Kategorie **Getränke** als Standardwert

Vorarbeit: Steuerelementname ändern

Da wir die Felder direkt aus der Feldliste in das Formular gezogen haben, hat Access die Namen der Felder als Steuerelementnamen übernommen. Das kann in manchen Situationen zu Problemen führen, wenn Sie per VBA auf das Steuerelement zugreifen wollen – etwa, um die Eigenschaft **DefaultValue** einzustellen. Diese ist nämlich nur für Steuerelemente verfügbar, nicht aber für die Felder der Datenherkunft des Formulars, die aber – wenn Steuerelement und Feld den gleichen Namen tragen – auf die gleiche Weise referenziert werden. Um Missverständnissen vorzubeugen, sollten Sie also das Steuerelement mit einem Präfix versehen,

also etwa **txt** für Textfelder oder **cbo** für Kombinationsfelder. Sie können ganz einfach testen, dass **DefaultValue** für Felder nicht zur Verfügung steht, indem Sie das Kombinationsfeld in **cboKategorieID** umbenennen und dann versuchen, dem Feld **KategorieID** die Eigenschaft **DefaultValue** zuzuweisen. Diese steht dort nicht zur Verfügung, wie Bild 3 zeigt.

Wert des vorherigen Datensatzes übernehmen

Eine Variante lautet nun, den Wert des zuvor eingegebenen Datensatzes als Standardwert für den folgenden Datensatz zu übernehmen. Dazu ist nicht viel Aufwand nötig: Wir fügen dem Klassenmodul des Formulars lediglich eine Ereignisprozedur hinzu, die durch das Ereignis **Nach Aktualisierung** des betroffenen Steuerelements ausgelöst wird. Diese Prozedur sieht wie folgt aus:

```
Private Sub cboKategorieID_AfterUpdate()
    Me!cboKategorieID.DefaultValue = Me!cboKategorieID
End Sub
```

Wenn Sie nun den Wert des Feldes **KategorieID** über das Kombinationsfeld ändern und dann einen neuen Datensatz anzeigen, erscheint der zuletzt gewählte Wert bereits als Standardwert dieses Feldes.

Wenn Sie diese Methode nutzen, benötigen Sie für andere Datentypen leicht abgewandelte Varianten der obigen Methode. Für Felder mit dem Datentyp **Text** etwa müssen Sie den Wert, den Sie der Eigenschaft **DefaultValue** zuweisen wollen, noch in Anführungszeichen einfassen. Für das Textfeld **txtArtikelname** sieht dies wie folgt aus:

```
Private Sub txtArtikelname_AfterUpdate()
    Me!txtArtikelname.DefaultValue = Chr(34) _
        & Me!txtArtikelname & Chr(34)
End Sub
```

Chr(34) entspricht dem Anführungszeichen (**"**). Sie können stattdessen auch zwei in zwei weitere Anführungszeichen eingefasste Anführungszeichen angeben:

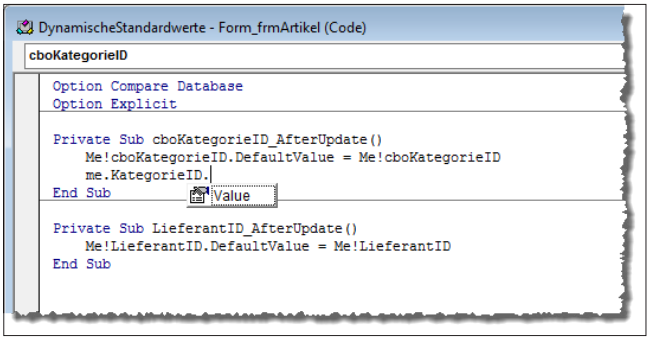

Bild 3: **DefaultValue** steht nur für Steuerelemente zur Verfügung

```
Me!txtArtikelname.DefaultValue = """" _
    & Me!txtArtikelname & """"
```

Die Variante mit **Chr(34)** ist jedoch besser lesbar.

Für andere Felddatentypen wie **Datum**- oder **Ja/Nein**-Felder verwenden Sie auch die zuerst genannte Variante mit der direkten Zuweisung des aktuellen Wertes.

Damit findet der Benutzer nun schon einmal jeweils den zuletzt eingegebenen Wert für verschiedene Steuerelemente als Standardwert für neue Datensätze vor.

Nach dem Schließen und Öffnen

Die Lösung hat allerdings einen kleinen Nachteil: Wenn der Benutzer die Datenbank schließt und wieder öffnet, sind die zusammengestellten Standardwerte wieder weg (genau genommen geschieht dies bereits, wenn das Formular geschlossen wird). Das ist klar, denn wenn wir dynamisch per VBA Eigenschaften zuweisen, werden diese ja nicht gespeichert. Dies wäre nur der Fall, wenn Sie die Eigenschaft **Standardwert** in der Entwurfsansicht festlegen und das Formular dann speichern.

Was tun? Abhilfe schafft eine Tabelle, in welcher wir die Standardwerte für verschiedene Formulare und Steuerelemente speichern können. Die Tabelle heißt **tblStandardwerte** und sieht im Entwurf wie in Bild 4 aus. Damit die Tabelle nicht nur für ein, sondern gleich für mehrere Formulare Standardwerte speichern kann, legen wir zwei Felder namens **Formularename** und **Steuerelementna-**

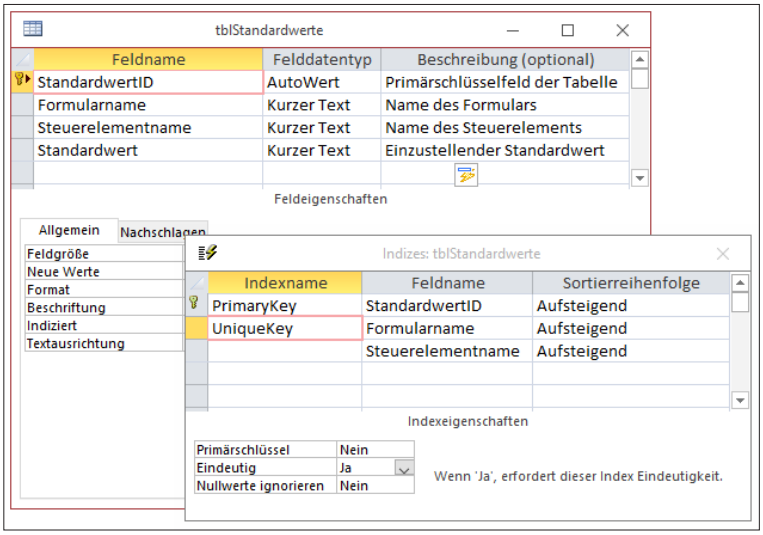

Bild 4: Tabelle zum Speichern von Standardwerten

me an, welche die entsprechenden Daten zum betroffenen Steuerelement speichern. Das Feld **Standardwert** nimmt dann den zu speichernden Standardwert auf. Damit der Benutzer für keine Kombination aus Formularname und Steuerelementname zwei Standardwerte speichern kann, fügen wir gleich noch einen eindeutigen Index namens **UniqueIndex** für die beiden Felder **Formularname** und **Steuerelementname** hinzu.

Um unser Formular mit Funktionen zum Lesen und Schreiben der Standardwerte auszustatten, kopieren wir es zunächst und fügen es unter dem Namen **frmArtikel_Standardwerttabelle** wieder ein. Werfen Sie dann die bereits angelegten Ereignisprozeduren aus der neuen Version des Formulars heraus.

Standardwerte speichern

Das Speichern eines Wertes als Standardwert soll beim Speichern des jeweiligen Datensatzes erfolgen. Dazu bauen wir uns eine Prozedur, die den Namen des Formulars, den Namen des Steuerelements und den neuen Standardwert als Parameter entgegennimmt und nennen diese **StandardwertSpeichern**. Die Prozedur sieht wie in Listing 1 aus. Die Prozedur prüft zunächst, ob der Parameter **varStandardwert** überhaupt

```
Public Sub StandardwertSpeichern(strFormularname As String, strSteuerelementname As String, varStandardwert As Variant)
    Dim db As DAO.Database
    Set db = CurrentDb
    On Error Resume Next
    If Not IsEmpty(varStandardwert) Then
        db.Execute "INSERT INTO tblStandardwerte(Formularname, Steuerelementname, Standardwert) VALUES('" _
            & strFormularname & "', '" & strSteuerelementname & "', '" & varStandardwert & "')", dbFailOnError
        If Err.Number = 3022 Then
            db.Execute "UPDATE tblStandardwerte SET Standardwert = '" & varStandardwert & "' WHERE Formularname = '" _
                & strFormularname & "' AND Steuerelementname = '" & strSteuerelementname & "'", dbFailOnError
        End If
    Else
        db.Execute "INSERT INTO tblStandardwerte(Formularname, Steuerelementname, Standardwert) VALUES('" _
            & strFormularname & "', '" & strSteuerelementname & "', NULL)", dbFailOnError
        If Err.Number = 3022 Then
            db.Execute "UPDATE tblStandardwerte SET Standardwert = NULL WHERE Formularname = '" & strFormularname _
                & "' AND Steuerelementname = '" & strSteuerelementname & "'", dbFailOnError
        End If
    End If
End Sub
```

Listing 1: Prozedur zum Speichern von Standardwerten

einen Wert enthält. Falls das Feld, für das der Standardwert gespeichert werden soll, nämlich den Wert **Null** hat, ist **varStandardwert** leer. Ist varStandardwert leer, geschieht Folgendes:

Die Prozedur versucht, einen neuen Datensatz in der Tabelle **tblStandardwerte** anzulegen. Dabei sollen die mit den drei Parametern **strFormularname**, **strSteuerelementname** und **varStandardwert** übergebenen Werte in die drei Felder **Formularname**, **Steuerelementname** und **Standardwert** der Tabelle **tblStandardwerte** eingetragen werden. Für den dritten Parameter haben wir den Datentyp **Variant** gewählt, damit hier auch der Wert **Null** verarbeitet werden kann.

Da es beim Speichern geschehen kann, dass die Prozedur versucht, einen Datensatz mit einer bereits vorhandenen Kombination aus Formularname und Steuerelementname zu speichern, haben wir zuvor die eingebaute Fehlerbehandlung deaktiviert.

So meldet Access keinen Fehler, wenn das Anlegen des Datensatzes aus dem genannten Grund fehlschlägt. Stattdessen prüft die Prozedur anschließend, ob der Fehler mit der Nummer **3022** aufgetreten ist, was diesem Fehler entspricht. In diesem Fall soll die Prozedur dann den Wert des Feldes **Standardwert** in dem bereits vorhandenen Datensatz überschreiben.

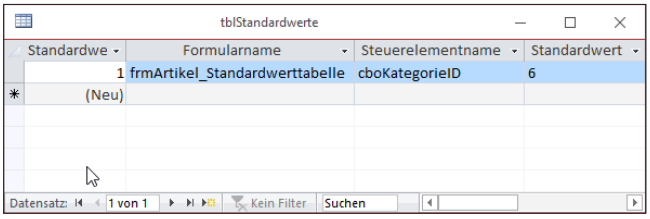

Bild 5: Tabelle zum Speichern von Standardwerten, Datenblattansicht

Sollte **varStandardwert** leer sein, dann legt die Prozedur auch einen Datensatz in der Tabelle **tblStandardwerte** an beziehungsweise ändert den Standardwert für einen eventuell vorhandenen Datensatz für diese Konstellation aus Formularname und Steuerelementname. Allerdings trägt die Prozedur dann den Wert **NULL** für das Feld **Standardwert** ein.

In der **Nach Aktualisierung**-Ereignisprozedur des Formulars tragen wir dann etwa die folgende Zeile ein, um den aktuellen Wert des Feldes **cboKategorieID** in der Tabelle **tblStandardwerte** zu speichern:

```
Private Sub Form_AfterUpdate()
    StandardwertSpeichern Me.Name, "cboKategorieID", _
        Me!cboKategorieID
End Sub
```

Nachdem Sie nun einen Datensatz im Formular **frmArtikel_Standardwerttabelle** bearbeitet haben, speichert

```
Public Sub StandardwerteSetzen(frm As Form)
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM tblStandardwerte WHERE Formularname = '" & frm.Name & "'", dbOpenDynaset)
    On Error Resume Next
    Do While Not rst.EOF
        frm.Controls(rst!Steuerelementname).DefaultValue = rst!Standardwert
        rst.MoveNext
    Loop
    On Error GoTo 0
End Sub
```

Listing 2: Prozedur zum Setzen von Standardwerten

Rechnungsbericht

Die Ausgabe von Rechnungen dürfte einer der beliebtesten Anwendungszwecke für die Erstellung von Berichten sein. Gleichzeitig sind Rechnungsberichte aber auch eine der anspruchsvollsten Aufgaben – zumindest, wenn man sämtlichen Schnickschnack wie Zwischensumme und Übertrag, vernünftige Aufteilung der Positionen bei mehrseitigen Berichten, keine letzte Seite ohne wesentliche Informationen et cetera berücksichtigen will. Und wenn Sie dann noch mehrere Rechnungen in einem Bericht abbilden wollen, haben Sie das Ziel erreicht.

Datenherkunft für den Bericht

Als Grundlage verwenden wir die Bestelldaten aus der Suedsturm-Datenbank. Welche Tabellen Sie für die dem Bericht zugrunde liegende Abfrage benötigen, lässt sich mit einem Blick auf eine herkömmliche Rechnung herausfinden. Für die Anschrift des Kunden benötigen Sie die **tblKunden**-Tabelle, den Rest erledigen die in einer m:n-Beziehung stehenden Tabellen **tblBestellungen**, **tblBestelldetails** und **tblArtikel**. Damit der Kunde weiß, an wen er sich bei Fragen wenden kann, nehmen Sie noch

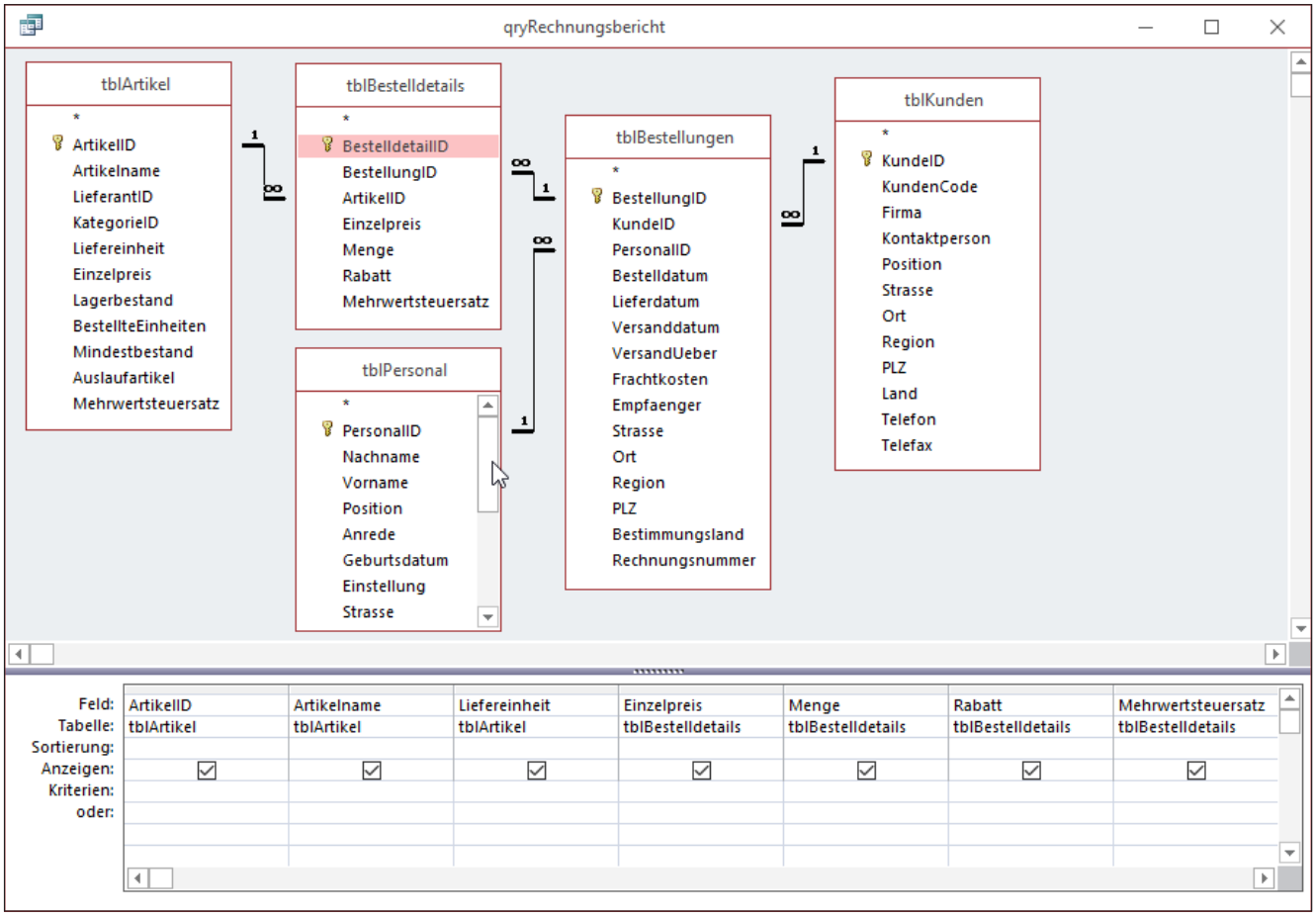

Bild 1: Datensatzquelle für den Rechnungsbericht (**qryRechnungsbericht**)

die Tabelle **tblPersonal** hinzu, die ebenfalls mit der Tabelle **tblBestellungen** verknüpft ist. Die als Datensatzquelle verwendete Abfrage sieht schließlich wie in Bild 1 aus.

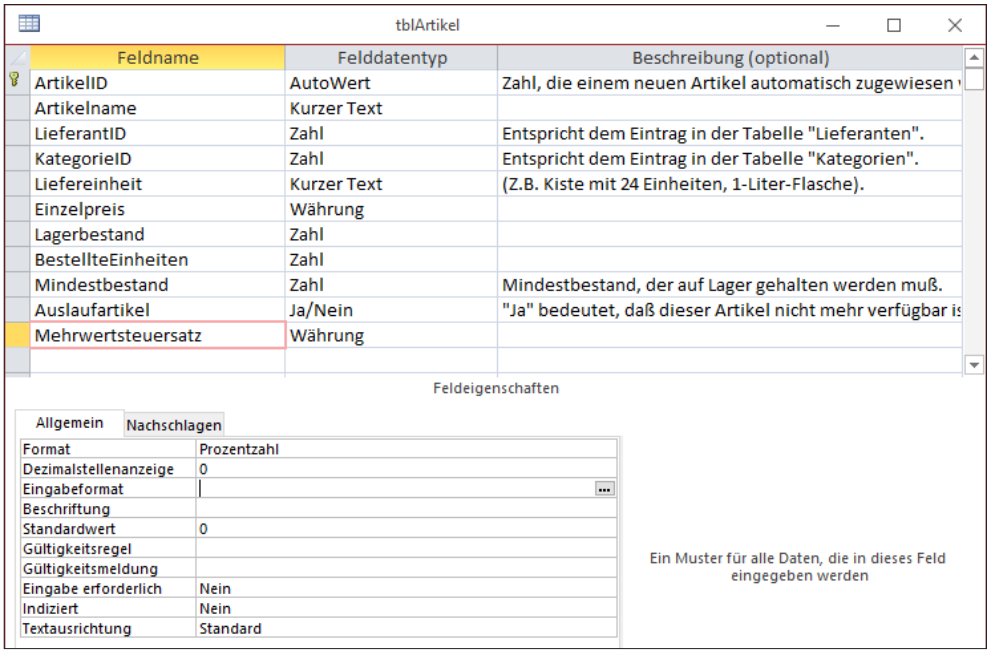

Bild 2: Das neue Feld **Mehrwertsteuersatz** in der Tabelle **tblArtikel**

Den aktuellen Steuergesetzen entsprechend wurde der Tabelle **Bestellung** noch ein Feld namens **Rechnungsnummer** hinzugefügt. Das Feld hat den Datentyp **Text**, um auch Zeichen wie Bindestriche oder Schrägstriche zu ermöglichen. Wie Sie die Rechnungsnummer ermitteln, bleibt Ihnen überlassen. Eine Rechnungsnummer muss aber auf jeden Fall eindeutig sein.

Manch einer verwendet eine bei 1 startende Nummerierung in Zusammenhang mit der Jahreszahl (zum Beispiel **2016-01**); andere lassen die Kundennummer in die Rechnungsnummer einfließen. Wer wichtig wirken möchte, verwendet eine Rechnungsnummer wie **950 752 0642**.

In der Beispieltabelle **tblBestellungen** ist die Rechnungsnummer schlicht eine Kombination aus dem Datum der Rechnungstellung und dem Primärschlüssel der Tabelle (etwa 20161009-11077). So können Sie auch einmal nur einfach in den Ordner mit Rechnungen schauen, wenn ein Kunde (oder der Steuerprüfer) dazu eine Frage hat – vorausgesetzt dort liegen die Rechnungen in chronologischer Reihenfolge vor ...

Mehrwertsteuersätze

Außerdem wurde die Tabelle **Artikel** um ein Feld namens **Mehrwertsteuersatz** erweitert (s. Bild 2), das zu Testzwecken nicht durchgängig den Wert **7%** enthält, wie es für Lebensmittel üblich ist, sondern dem wir hier und da auch mal den Wert 19% hinzugefügt haben. Die Tabelle **tblBestelldetails** enthält das gleiche Feld, in das bei einer Bestellung der gültige Mehrwertsteuersatz übertragen werden soll – auf diese Weise kann man die Mehrwertsteuersätze für die Tabelle **tblArtikel** ändern, ohne dass sich dies auf bereits erstellte Datensätze in der Tabelle **tblBestellpositionen** auswirkt. Wir wollen ja gegebenenfalls auch einmal eine Rechnungskopie drucken können, ohne dass dort wegen geänderter Mehrwertsteuersätze plötzlich neue Rechnungsbeträge herauskommen.

Das Feld **Mehrwertsteuersatz** haben wir mit dem Datentyp **Währung** versehen. Auf diese Weise erhalten wir die gewünschte Genauigkeit. Außerdem haben wir das Format auf **Prozentzahl** eingestellt und die Anzeige der Dezimalstellen auf **0** – sollten einmal Mehrwertsteuersätze wie **19,5%** auftauchen, kann man dies nachträglich ändern.

Auf die gleiche Weise haben wir auch die Felder **Mehrwertsteuersatz** und **Rabatt** in der Tabelle **tblBestelldetails** angepasst.

Berechnete Felder

Die Abfrage selbst wurde um ein berechnetes Feld namens **Bruttopreis** ergänzt, das den Bruttopreis einer jeden Position enthält (siehe Auflistung). Außerdem gibt es noch zwei weitere berechnete Felder namens **Netto** und **MwStBetrag**. Sie verwenden in diesem Bericht folgende Felder:

- Tabelle **tblArtikel**: **ArtikelID**, **Artikelname**, **Lieferein-heit**
- Tabelle **tblBestelldetails**: **Einzelpreis**, **Menge**, **Rabatt**, **Mehrwertsteuer**
- Tabelle **tblBestellungen**: **BestellungID**, **KundeID**, **Bestelldatum**, **Versanddatum**, **VersandUeber**, **Frachtkosten**, **Empfaenger**, **Strasse**, **Ort**, **PLZ**, **Bestimmungsland**, **Rechnungsnummer**
- Tabelle **tblKunden**: **Firma**, **Kontaktperson**, **Strasse**, **Ort**, **PLZ**, **Land**
- Tabelle **tblPersonal**: **Nachname**, **Vorname**, **Anrede**, **DurchwahlBuero**
- Berechnetes Feld **Bruttopreis**: **[tblBestelldetails].[Einzelpreis]*[tblBestelldetails].[Menge]* ([tblBestelldetails].[Rabatt]+1)*([tblArtikel].[Mehrwertsteuer]+1)**
- Berechnetes Feld **Netto**: **[tblBestelldetails.Einzelpreis]*[Menge]*(1+[Rabatt])**
- Berechnetes Feld **MwStBetrag**: **[tblBestelldetails.Einzelpreis]*[Menge]*(1+[Rabatt])*[Mehrwertsteuer]**

Die letzten beiden Felder werden nicht in der Auflistung der Positionen angezeigt, sondern dienen nur der Ermittlung der Summe der Nettopreise und der Mehrwertsteuer am Ende der Rechnung.

Wenn Sie mit früheren Versionen der Suedsturm-Datenbank experimentiert haben, ist Ihnen vielleicht auch aufgefallen, dass wir den Namen des Feldes **Anzahl** in der Tabelle **tblBestelldetails** durch **Menge** ersetzt haben. Der Grund ist einfach: In der deutschen Version von Access ist **Anzahl** ein reserviertes Wort, was hier und da zu Problemen führen könnte. Daher haben wir es durch die Bezeichnung **Menge** ersetzt.

Konzept für die Erstellung des Berichts

Bevor Sie sich an die Erstellung eines Berichts machen, sollten Sie die Daten in Gedanken (oder auch auf dem Papier) kurz auf die einzelnen Bereiche eines Berichts aufteilen und sich überlegen, welche Daten etwa nur auf der ersten Seite angezeigt werden, was passiert, wenn so viele Datensätze vorhanden sind, dass der Bericht über mehrere Seiten geht, und so weiter.

Im vorliegenden Fall haben Sie es mit einem ziemlich großen Berichtskopf zu tun: Dieser enthält den kompletten Briefkopf, die Anschrift des Kunden, Lieferdaten und so weiter. Moment: Ist das wirklich so? Wenn der Berichtskopf bereits kundenspezifische und damit rechnungsspezifische Daten enthält, dann können Sie nur eine Rechnung je Bericht ausgeben. Warum? Weil der Berichtskopf nur einmal je Bericht angezeigt wird. Wenn Sie aber mal einen ganzen Schwung Rechnungen ausdrucken möchten, haben Sie ein Problem: Sie müssen dann schon den gleichen Bericht mehrere Male aufrufen. Natürlich geht das auch, aber in diesem Fall sollen Sie einen Bericht erstellen, der mehrere Rechnungen auf einen Rutsch anfertigen kann.

Also benötigen Sie eine Gruppierung über die einzelnen Bestellungen – als Gruppierungsfeld bietet sich das Feld **BestellungID** an. Im Gruppenkopf bringen Sie nun alles unter, was Sie sonst in den Berichtskopf packen wollten – Briefkopf, Anschrift, Bestelldaten wie Bestellnummer und so weiter.

Die einzelnen Positionen gehören – ganz klar – in den Detailbereich. Darüber benötigen Sie Feldüberschriften: Diese können Sie auf der ersten Seite direkt im Gruppen-

kopf unterbringen, auf den folgenden Seiten im Seitenkopf. Dieser darf dann wiederum nicht auf der ersten Seite angezeigt werden – dazu später mehr. Nach der letzten Rechnungsposition folgt noch die Rechnungssumme. Dafür bietet sich der Berichtsfuß an. Auf der Seite mit dem Berichtsfuß soll wiederum kein Seitenfußbereich angezeigt werden.

Wenn Sie dem Kunden ein wenig mehr Komfort bieten möchten, sorgen Sie auch noch für eine Zwischensumme und einen Übertrag. Die Zwischensumme gehört mit in den Seitenfuß, der Übertrag in den Seitenkopf – dieser erscheint aber nur auf den Seiten, die keinen Gruppierungskopf anzeigen. Sie sehen – eine Rechnung ist nicht gerade der trivialste Bericht, der sich mit Access erstellen lässt, und er enthält noch nicht einmal Gruppierungen.

Briefkopf im Berichtskopf oder Gruppenkopf?

Normalerweise würden wir den Briefkopf im Bereich **Berichtskopf** eines Berichts anlegen. Eigentlich logisch – der erscheint schließlich maximal einmal, und zwar auf der ersten Seite.

Allerdings wollen wir dabei nicht vergessen, dass wir nicht nur eine Rechnung, sondern gegebenenfalls auch mehrere Rechnungen in einem Bericht ausgeben wollen. Das heißt also, dass wir die Daten des Briefkopfs in den Gruppenkopf für die einzelnen Rechnungen packen müssen. So benötigen wir also eine Gruppierung für den Bericht, der die Daten unserer Datenherkunft nach den Datensätzen der Tabelle **tblBestellungen** gruppiert. Diesen richten Sie wie in Bild 3 ein. Hier wählen Sie außerdem die Eigenschaftswerte **mit Kopfzeilenbereich**, **mit Fußzeilenbereich** und **Kopfzeile und ersten Datensatz auf einer Seite zusammenhalten** aus.

Erstellen des Gruppenkopfs

Die Erstellung des Gruppenkopfs einer Rechnung ist im Wesentlichen Design-Arbeit und weniger Denksport. Hier bringen Sie den Briefkopf unter, den Block mit der Empfängeradresse und den Block mit den allgemeinen Rechnungsdaten (s. Bild 4).

Hier gibt es folgende Besonderheiten: Das Feld **txtPLZUndOrt** fasst die Felder **PLZ und Ort** der Tabelle

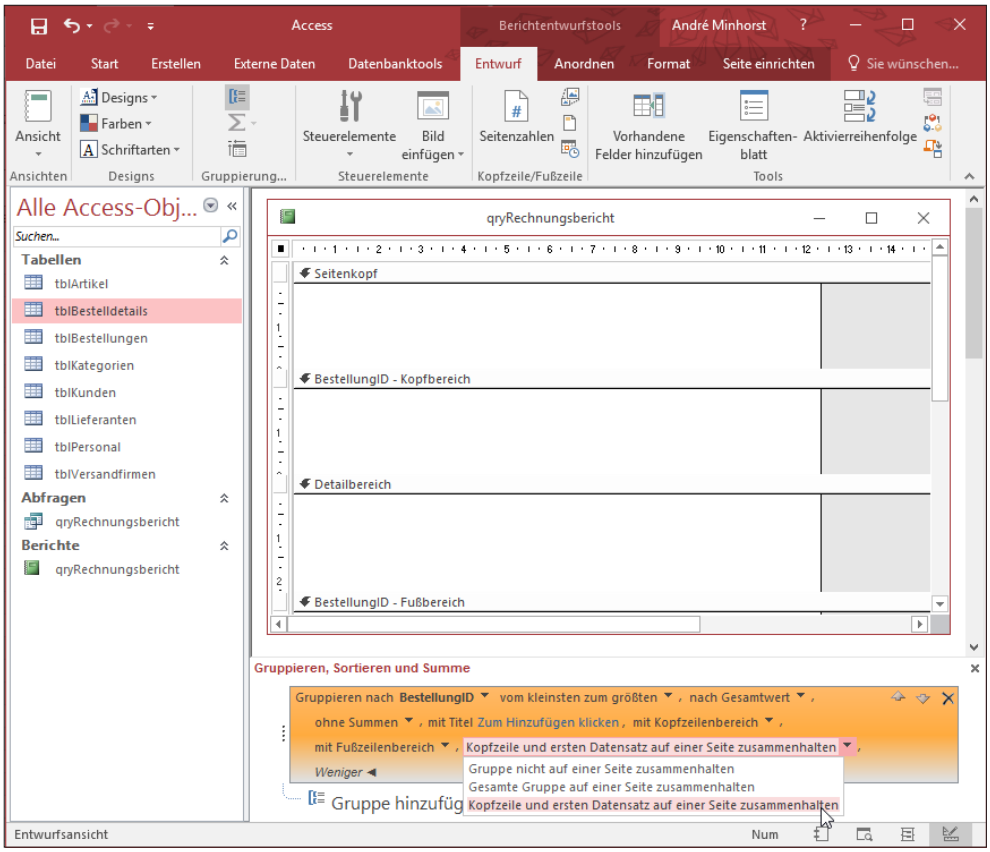

Bild 3: Einrichtung der Gruppierung für den Rechnungsbericht

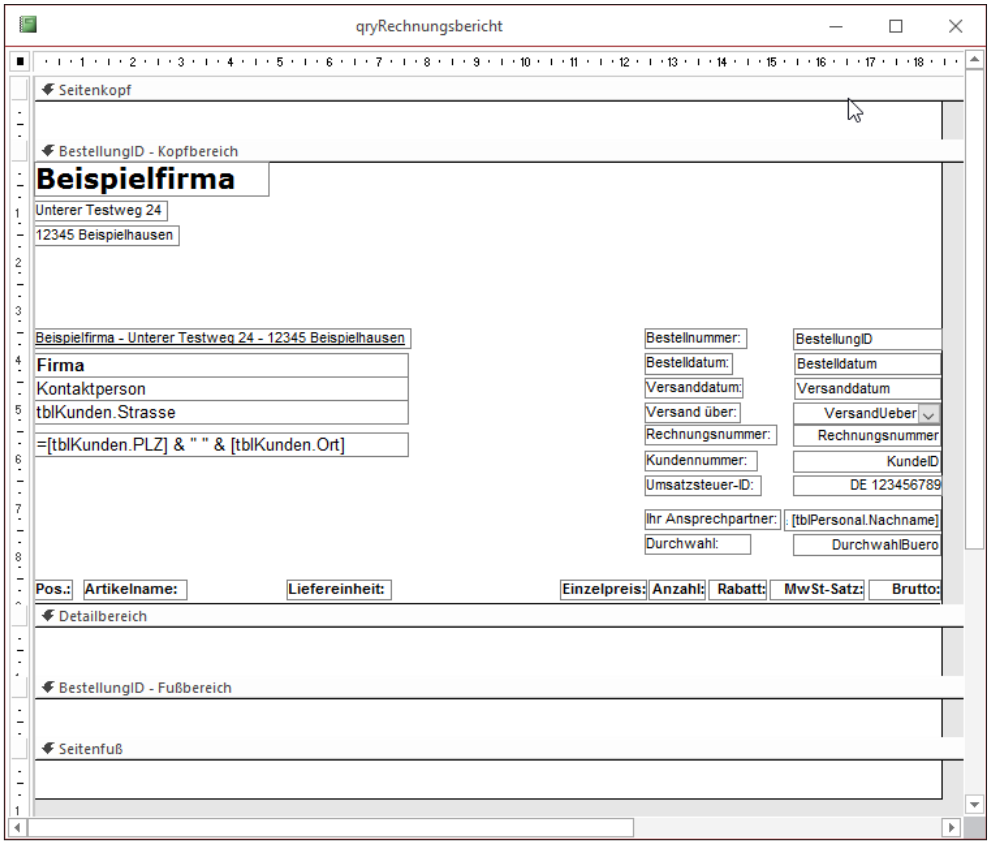

Bild 4: Einrichtung des Gruppenkopfes für die Rechnung

Füllen des Detailbereichs anlegen, in der Abbildung sehen Sie aber bereits ihre Anordnung.

Anlegen des Detailbereichs

Der Detailbereich ist reine Fleißarbeit – abgesehen von einem Feature, das aber erst später hinzukommt, und der Angabe der Rechnungspositionen. Der Detailbereich enthält die Felder **Artikelname**, **Liefereinheit**, **Bruttopreis**, **Einzelpreis, Menge, Rabatt**, **Mehrwertsteuersatz** und **Bruttopreis**, wobei Letzteres das berechnete Feld ist, das Sie der Abfrage weiter oben hinzugefügt haben. Es fehlt noch das ganz linke Feld in der Entwurfsansicht (s. Bild 5). Es heißt **txtPosition** und besitzt als Steuerelementinhalt den Wert **=1**. Damit dieses Feld die Position des Artikels für die **Kunden** zusammen. Wichtig ist hier wie bei anderen Steuerelementen, dass Sie Tabellennamen und Feldnamen von in mehreren Tabellen vorkommenden Feldern durch Punkt getrennt schreiben und in eckige Klammern einfassen. Dies geschieht noch einmal im Gruppenkopf, und zwar im Textfeld **txtAnsprechpartner**. Dessen Inhalt lautet:

```
=[Personal.Anrede] & " " &
[Personal.Vorname] & " " &
[Personal.Nachname]
```

Die Feldüberschriften werden Sie normalerweise erst im folgenden Schritt beim

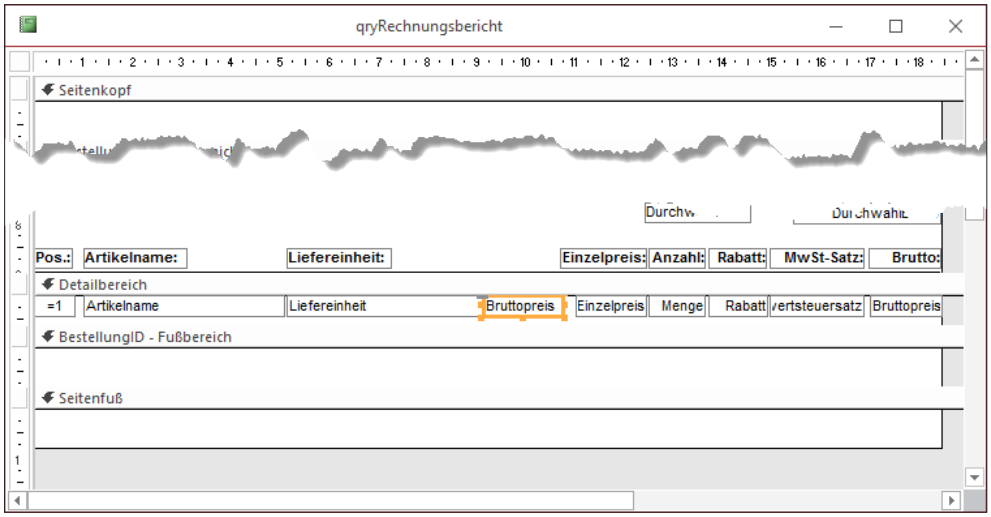

Bild 5: Einrichtung des Detailbereichs

aktuelle Rechnung, also innerhalb der aktuellen Gruppierung, anzeigt, stellen Sie die Eigenschaft **Laufende Summe** dieses Feldes auf **Über Gruppe** ein.

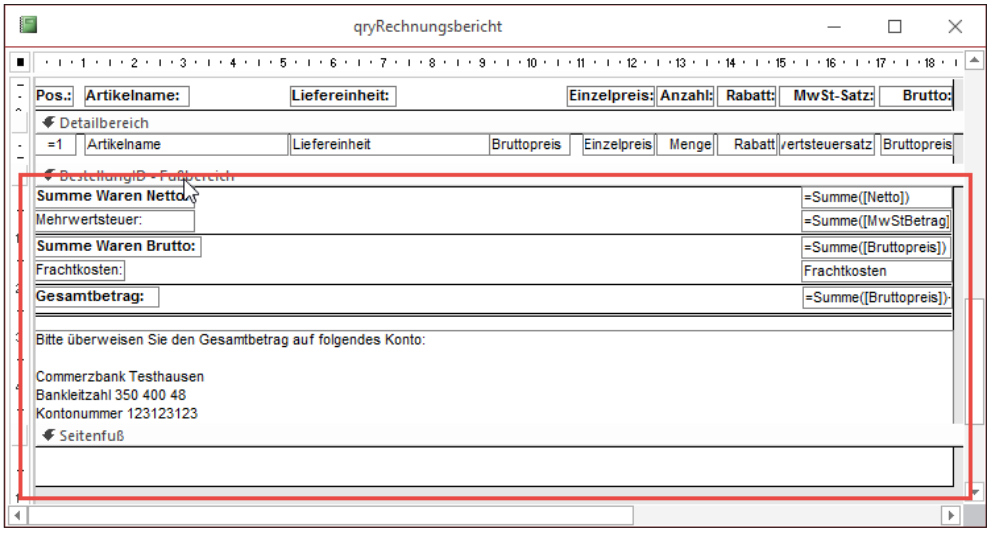

Bild 6: Steuerelemente für den Fußbereich

Außerdem fügen Sie das Feld **Bruttopreis** erneut hinzu (siehe das markierte Feld in der Abbildung). Für dieses stellen Sie die Eigenschaft **Sichtbar** auf **Nein** ein sowie **Laufende Summe** auf **Über Gruppe**. Geben Sie diesem Feld außerdem die Bezeichnung **txtLaufendeSumme**. Wir benötigen es später für die Zwischensumme und den Übertrag.

Berechnungen in Berichten oder Berechnungen in Formularen

Möglicherweise fragen Sie sich, ob man die Berechnung des Bruttopreises und der anderen berechneten Felder in der Abfrage nicht auch innerhalb des Berichts hätte durchführen können. Die Frage ist berechtigt, da das sogar funktionieren würde. Das Problem ist nur, dass Sie keine datensatzübergreifenden Berechnungen über den Bezug auf das Feld mit dem Berechnungsergebnis durchführen könnten. Ein in der Abfrage berechnetes Feld behandelt Access im Bericht aber wie ein normales Tabellenfeld; hier sind Berechnungen wie etwa das Bilden der Summe über alle Datensätze einer Gruppierung möglich.

Streng genommen würde man im Bericht auch ohne das in der Abfrage berechnete Feld auskommen, aber dann müsste man im Feld zur Berechnung der Summe über mehrere Datensätze Bezug auf die Berechnungsformel nehmen und nicht auf das Feld mit dem Berechnungsergebnis.

Summenbildung im Fußbereich der Gruppierung

Im Fußbereich der Gruppierung heißt es: Abrechnen! Hier wird die Summe über die Felder **Netto**, **MwStBetrag** und **Bruttopreis** des Detailbereichs gebildet und schließlich noch der Frachtkostenanteil hinzuaddiert (s. Bild 6).

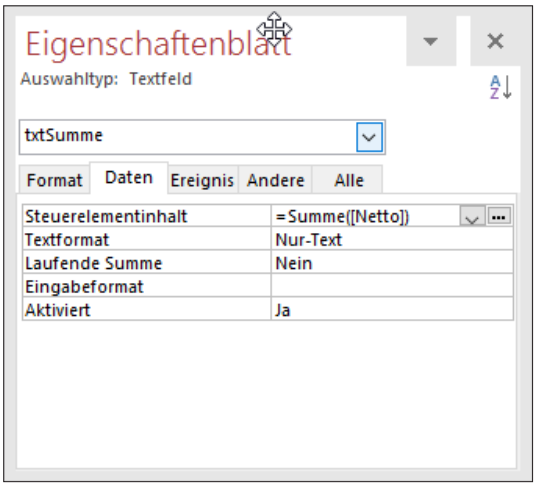

Bild 7: Einstellung der Eigenschaft **LaufendeSumme** für die Summenfelder

Dazu fügen Sie für die Eigenschaft Steuerelement einfach den Feldnamen ein und fügen die **Summe()**-Funktion hinzu, sodass beispielsweise **=Summe([Netto])** daraus wird.

Achtung: Interessanterweise müssen wir für die Eigenschaft **Laufende Summe** für die Summentextfelder nicht etwa den Wert **Über Gruppe** auswählen, sondern den Wert **Nein** beibehalten (s. Bild 7).

RDBMS-Zugriff per VBA: Verbindungen

Wenn Sie von Access aus auf die Daten einer SQL Server-Datenbank (und neuerdings auch auf LocalDB-Datenbanken) zugreifen wollen, müssen Sie mit Bordmitteln arbeiten, die häufig nicht zufriedenstellend sind. Wir haben einen Satz von Tools entwickelt, mit denen Sie eine Reihe von Aufgaben sehr schnell erledigen können: Verbindungen definieren, Tabellen verknüpfen und SQL-Abfragen direkt an den Server schicken. Dieser Beitrag zeigt, wie Sie mithilfe eines Teils dieser Tools per VBA auf SQL Server und Co. zugreifen.

Mal eben eine Auswahlabfrage oder Pass-Through-Abfrage per VBA an den SQL Server absetzen – das wäre doch eine praktische Sache.

Sie werden an verschiedenen Stellen per VBA auf die Tabellen der SQL Server-Datenbank zugreifen müssen – sei es, um eine ODBC-Verknüpfung herzustellen oder zu aktualisieren, das Ergebnis einer gespeicherten Abfrage abzurufen oder eine solche auszuführen oder auch um ein Recordset auf Basis einer gespeicherten Prozedur einem Formular oder einem Steuerelement zuzuweisen. Alles, was Sie wissen müssen, um dies zu erledigen, erfahren Sie in diesem Beitrag.

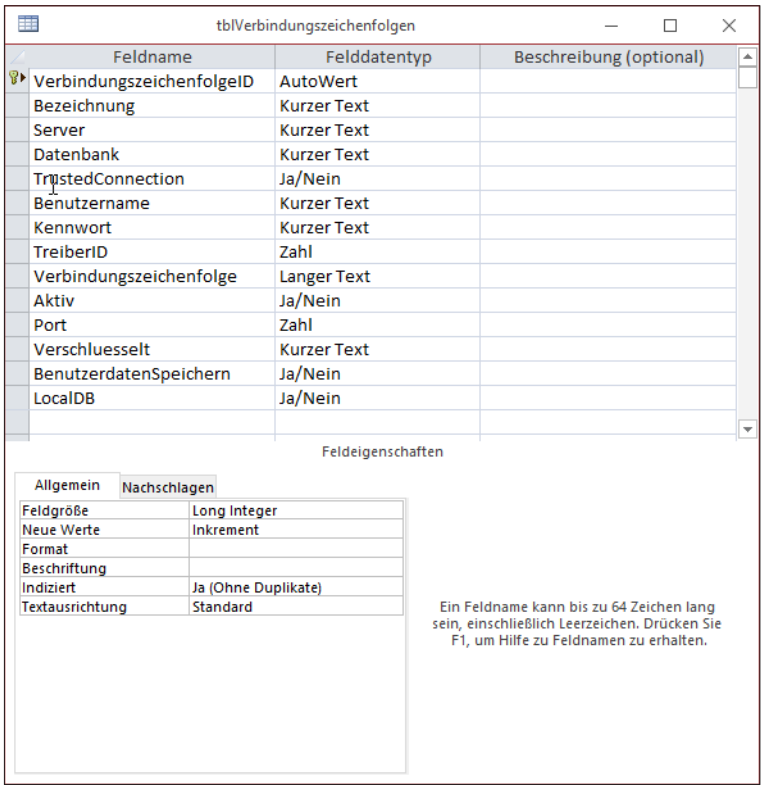

Bild 1: Entwurf der Tabelle zum Speichern der Verbindungszeichenfolgen

Beispieldatenbank

Als Beispieldatenbank verwenden wir die Datenbank **Suedsturm.mdf**, die Sie wie im Beitrag **Access und LocalDB** (**www.access-im-unternehmen.de/1057**) beschrieben nutzen können. Alternativ können Sie diese natürlich auch per SQL Server nutzen. Die Tools, mit denen Sie die in diesem Beitrag verwendeten Verbindungszeichenfolgen zusammenstellen können, stellen wir im Beitrag **SQL Server-Tools** (**www.access-im-unternehmen.de/1061**) in der nächsten Ausgabe vor.

Verbindungszeichenfolgen

Als Erstes benötigen Sie Zugriff auf die SQL Server-Datenbank. Voraussetzung dafür ist eine geeignete Verbindungszeichenfolge. Bei den Verbindungszeichenfolgen gibt es verschiedene Ansätze.

Sie sind relativ flexibel, wenn Sie die notwendigen Informationen in einer lokalen Tabelle im Access-Frontend speichern und von Access aus per VBA darauf zugreifen, um Verbindungszeichenfolgen daraus zu erstellen. Alternativ können Sie Verbindungszeichenfolgen in einer

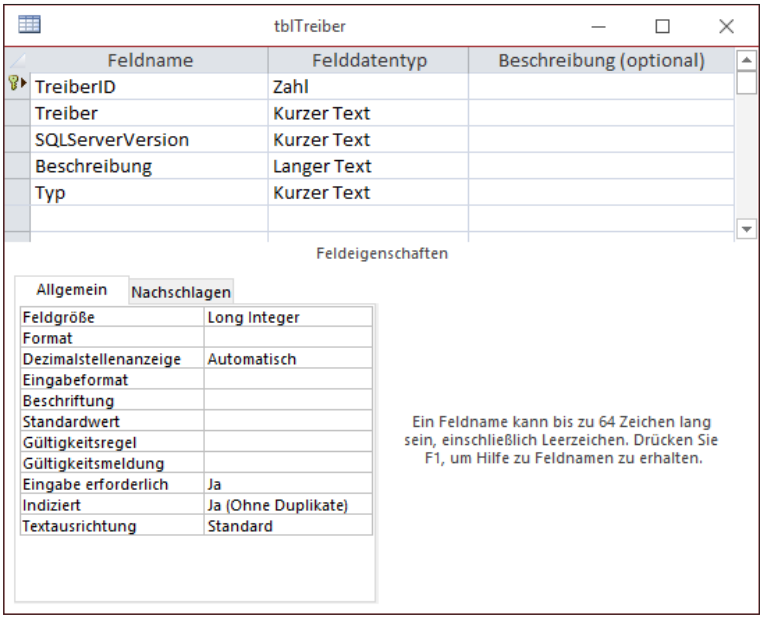

Bild 2: Entwurf der Tabelle zum Speichern der Treiber

DSN-Datei oder als System-DSN in der Registry speichern – wir gehen an dieser Stelle jedoch auf die Variante der tabellenbasierten Verbindungszeichenfolge ein. Die notwendigen Daten speichern wir in den Beispieldatenbanken in einer Tabelle namens **tblVerbindungszeichenfolgen** (s. Bild 1). Diese Tabelle haben Sie bereits im Beitrag **RDBMS-Tools: Verbindungen verwalten** (**www.access-im-unternehmen.de/976**) kennengelernt.

Außerdem benötigen wir eine Tabelle namens **tblTreiber**, welche die Daten der gängigen Treiber enthält (s. Bild 2). Die Tabelle **tblVerbindungszeichenfolgen** ist über das Fremdschlüsselfeld **TreiberID** mit der Tabelle **tblTreiber** verknüpft.

```
Public Function VerbindungszeichenfolgeNachID(lngVerbindungszeichenfolgeID As Long, _
        Optional bolZugangsdatenAusVariablen As Boolean) As String
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strTemp As String, strTreiber As String, strServer As String, strDatenbank As String
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM tblVerbindungszeichenfolgen WHERE VerbindungszeichenfolgeID = " _
        & lngVerbindungszeichenfolgeID)
    strTreiber = DLookup("Treiber", "tblTreiber", "TreiberID = " & rst!TreiberID)
    strServer = rst!Server
    strDatenbank = rst!Datenbank
    strTemp = "ODBC;DRIVER={" & strTreiber & "};" & "SERVER=" & strServer & ";" _
        & "DATABASE=" & strDatenbank & ";"
    If rst!TrustedConnection = True Then
        strTemp = strTemp & "Trusted_Connection=Yes"
    Else
        If Len(Nz(rst!Benutzername, "")) > 0 And bolZugangsdatenAusVariablen = False Then
            strBenutzername = rst!Benutzername
        End If
        If Len(Nz(rst!Kennwort, "")) > 0 And bolZugangsdatenAusVariablen = False Then
            strKennwort = rst!Kennwort
        End If
        strTemp = strTemp & "UID=" & strBenutzername & ";"
        strTemp = strTemp & "PWD=" & strKennwort
    End If
    VerbindungszeichenfolgeNachID = strTemp
End Function
```

Listing 1: Ermitteln einer Verbindungszeichenfolge aus der Tabelle **tblVerbindungszeichenfolgen**

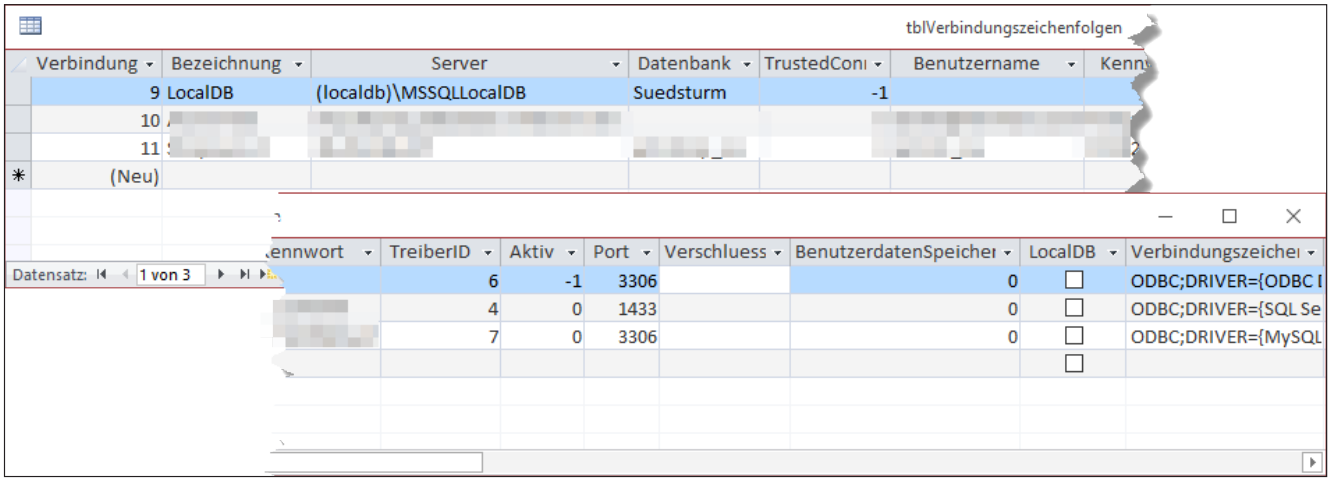

Bild 3: Tabelle mit den Daten einer Verbindungszeichenfolge

In der Beispielanwendung verwenden wir die Prozedur aus Listing 1, um eine Verbindungszeichenfolge aus den Daten der Tabelle **tblVerbindungszeichenfolgen** zu ermitteln.

Die Funktion arbeitet direkt mit der Tabelle **tblVerbindungszeichenfolgen**, in der die Verbindungsprarameter gespeichert sind. Dabei liest sie den Datensatz der Tabelle **tblVerbindungszeichenfolgen** ein, dessen Primärschlüsselwert dem mit dem Parameter **lngVerbindungszeichenfolgeID** übergebenen Wert entspricht. Der zweite Parameter der Funktion heißt **bolZugangsdatenAusVariablen** und legt fest, ob auf jeden Fall die Zugangsdaten aus den beiden Variablen **strBenutzername** und **strKennwort** verwendet werden sollen. Dies benötigen wir für einen Aufruf aus der später erläuterten Funktion **VerbindungTesten**, die gegebenenfalls die Zugangsdaten vorher per Dialog abfragt und diese aus Sicherheitsgründen nur in den beiden Variablen **strBenutzername** und **strKennwort** speichert, aber nicht in der Tabelle.

Es gibt zwei Variablen namens **strBenutzername** und **strKennwort**, die speziell für den Einsatz mit der SQL Server-Authentifizierung vorgesehen sind (oder auch für die Anmeldung an einem anderen SQL-Server-System wie MySQL mit der Anforderung von Benutzerdaten). In diesem Fall muss die Access-Anwendung die Benutzerdaten bereitstellen, um eine Verbindung herzustellen. Diese sollen aber nicht in der Datenbank gespeichert werden, da die Daten sonst für jedermann zugänglich wären. Die Variablen sollen zuvor per Formular vom Benutzer einmalig pro Sitzung abgefragt und in entsprechenden Variablen gespeichert werden, die wie folgt deklariert werden:

```
Public strBenutzername As String
Public strKennwort As String
```

Die Funktion **VerbindungszeichenfolgeNachID** prüft dann, ob die Tabelle eigene Werte für Benutzername und Kennwort enthält, und trägt diese gegebenenfalls in die Variablen **strBenutzername** und **strKennwort** ein. Danach setzt die Funktion die einzelnen Elemente dann zu einer Verbindungszeichenfolge zusammen. Abhängig davon, ob die Windows-Authentifizierung oder SQL Server-Authentifizierung gewählt wurde, erhält die Verbindungszeichenfolge das Name-Wert-Paar **Trusted_Connection=Yes**, anderenfalls die Benutzerdaten in der Form **UID=<Benutzername>;PWD=<Kennwort>**. Woher die Werte der beiden Variablen **strBenutzername** beziehungsweise **strKennwort** kommen, haben wir ja weiter oben bereits erläutert.

Aus dem obersten Eintrag der Tabelle **tblVerbindungszeichenfolgen** aus Bild 3 würde die Funktion mit dem Wert **9** als Parameter etwa folgendes Ergebnis liefern:

```
Public Function Standardverbindungszeichenfolge() As String
    Dim lngAktivID As Long
    If Not lngAktivID = 0 Then
        Standardverbindungszeichenfolge = VerbindungszeichenfolgeNachID(lngAktivID)
    Else
        MsgBox "Achtung: Es ist keine Verbindungszeichenfolge als aktiv gekennzeichnet."
    End If
End Function
```

Listing 2: Ermitteln der Standardverbindungszeichenfolge aus der Tabelle **tblVerbindungszeichenfolgen**

```
? VerbindungszeichenfolgeNachID(9)
ODBC;DRIVER={ODBC Driver 11 for SQL
Server};SERVER=(localdb)\MSSQLLocalDB;DATABASE=Suedsturm;T
rusted_Connection=Yes
```

Standardverbindungszeichenfolge

In der Tabelle **tblVerbindungszeichenfolgen** finden Sie auch ein **Boolean**-Feld namens **Aktiv**. Dieses legt fest, welche Verbindungszeichenfolge für die aktuelle Datenbank standardmäßig verwendet werden soll.

Der generelle Vorteil des Speicherns der Daten für die Verbindungszeichenfolge in einer Tabelle ist, dass Sie diese bei Bedarf einfach ändern können. Der zweite Vorteil ist: Sie können auch mehrere Verbindungszeichenfolgen angeben und zwischen diesen Zeichenfolgen wechseln. Genau dies ermöglicht das Feld **Aktiv**.

Während Sie nun mit der Funktion **VerbindungszeichenfolgeNachID** immer die ID der aktuell benötigten Verbindungszeichenfolge angeben müssen, ermöglicht die Funktion **Standardverbindungszeichenfolge**, direkt die als **Aktiv** markierte Verbindungszeichenfolge zu ermitteln (s. Listing 2).

Die Funktion ermittelt per **DLookup**-Funktion den ersten Eintrag der Tabelle **tblVerbindungszeichenfolgen**, dessen Feld **Aktiv** den Wert **True** aufweist. Dieser wird dann der Funktion **VerbindungszeichenfolgeNachID** übergeben, um die entsprechende Verbindungszeichenfolge zu ermitteln. Sollte keine Verbindungszeichenfolge als aktiv markiert sein, erscheint eine entsprechende Meldung.

Wenn hingegen mehr als eine Verbindungszeichenfolge den Wert **True** im Feld **Aktiv** enthält, liefert die Funktion die Verbindungszeichenfolge für den ersten markierten Eintrag zurück.

ID der aktiven Verbindungszeichenfolge ermitteln

Andere Routinen erwarten gegebenenfalls die ID der zu verwendenden Verbindungszeichenfolge. Wenn Sie nicht explizit einen Wert des Feldes **VerbindungszeichenfolgeID** der Tabelle **tblVerbindungszeichenfolgen** überge-

```
Public Function AktiveVerbindungszeichenfolgeID() As Long
    Dim lngAktivID As Long
    lngAktivID = Nz(DLookup("VerbindungszeichenfolgeID", "tblVerbindungszeichenfolgen", "Aktiv = True"), 0)
    If lngAktivID = 0 Then
        MsgBox "Achtung: Es ist keine Verbindungszeichenfolge als aktiv gekennzeichnet."
    End If
    AktiveVerbindungszeichenfolgeID = lngAktivID
End Function
```

Listing 3: Ermitteln der ID der Standardverbindungszeichenfolge aus der Tabelle **tblVerbindungszeichenfolgen**

SQL Server-Datenbank erstellen

Das SQL Server Management Studio gibt es mittlerweile als kostenlosen Download, den Sie separat zu einer bestehenden SQL Server-Datenbank installieren können. Da es mittlerweile mit LocalDB auch eine sehr schlanke SQL Server-Version gibt, wollen wir uns einmal anschauen, wie Sie mit diesen beiden Softwarekomponenten eine SQL Server-Datenbank aufsetzen können. Dabei wollen wir unsere oft genutzte Beispieldatenbank »Suedsturm.mdb« nachbilden.

Wenn Sie »SQL Server Management Studio Download« als Suchbegriff bei Google eingeben, landen Sie schnell bei dem folgenden Link:

```
https://msdn.microsoft.com/de-de/library/mt238290.aspx
```

Dieser führt Sie zu der Seite mit dem Titel **Herunterladen von SQL Server Management Studio (SSMS)**. Die Suchbegriffe, mit denen ich auf den Download gestoßen bin, gebe ich extra an, weil ja mit neuen Versionen neue Links entstehen – und Sie sollen diesen Beitrag ja auch mit Folgeversionen noch nutzen können.

Ein wichtiger Hinweis ist, dass Sie SQL Server Management Studio auch in verschiedenen Versionen nebeneinander installieren können. Ich habe zum Beispiel die Versionen 2014 und 2016 auf meinem Rechner installiert. Wenn Sie im Startmenü nach SQL Server Management Studio suchen, erscheinen dann zwei Einträge – **SQL Server Management Studio** und **SQL Server Management Studio 2014**. Der Eintrag ohne zusätzliche Versionsangabe ist dann zumindest in diesem Fall die neuere, aktuelle Version.

Voraussetzungen

In diesem Beitrag wollen wir mit der LocalDB-Variante von SQL Server arbeiten. Diese sollten Sie daher zuvor installieren. Weitere Informationen dazu finden Sie im Beitrag **Access und LocalDB** (**www.access-im-unternehmen.de/1057**). Ein Starten wie beim SQL Server-Dienst ist nicht nötig, da es sich bei LocalDB nicht um einen Dienst, sondern um eine Anwendung handelt. Sie können also einfach so auf eine Instanz von LocalDB zugreifen.

SQL Server Management Studio starten

Nach dem ersten Start von SQL Server Management Studio habe ich mir dieses Programm an die Taskleiste angeheftet, damit ich nicht immer über das Startmenü danach suchen muss. Nach dem Start zeigt SQL Server Management Studio immer zuerst den Dialog zur Herstellung einer Verbindung mit einem Server an.

Normalerweise bauen Sie die Verbindung mit folgendem Server auf:

```
(localdb)\MSSQLLocalDB
```

Wenn Sie mehrere Versionen auf Ihrem Rechner installiert haben und eine andere als die aktuellste Version verbinden wollen, geben Sie die Versionsnummer etwa wie folgt an:

```
(localdb)\v11.0
```

Vielleicht möchten Sie auch mit einer eigenen, benannten Instanz arbeiten, die Sie zuvor über die Kommandozeile etwa mit folgendem Befehl gestartet haben:

```
(localdb)\Beispielinstanz
```

Für unsere Zwecke sollte jedoch die Standardinstanz (**MSSQLLocalDB**) ausreichen (s. Bild 1).

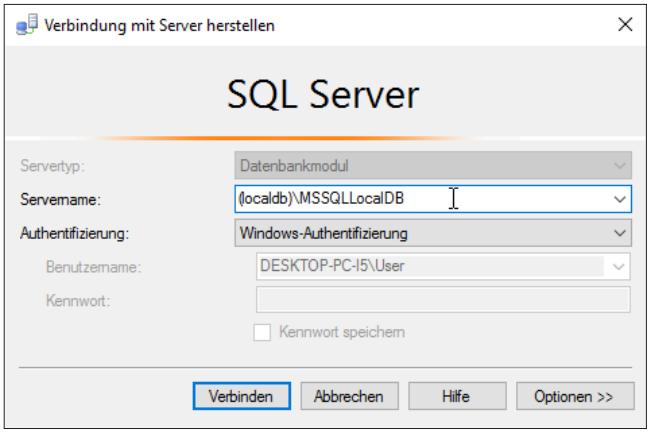

Bild 1: Aufbau einer Verbindung

Neue Datenbank anlegen

Danach zeigt Microsoft SQL Server Management Studio bereits den Objekt-Explorer mit der Verbindung und eventuell vorhandenen Datenbanken an. Sie können dort nun über den Kontextmenü-Eintrag **Neue Datenbank...** eine neue Datenbank einfügen (s. Bild 2).

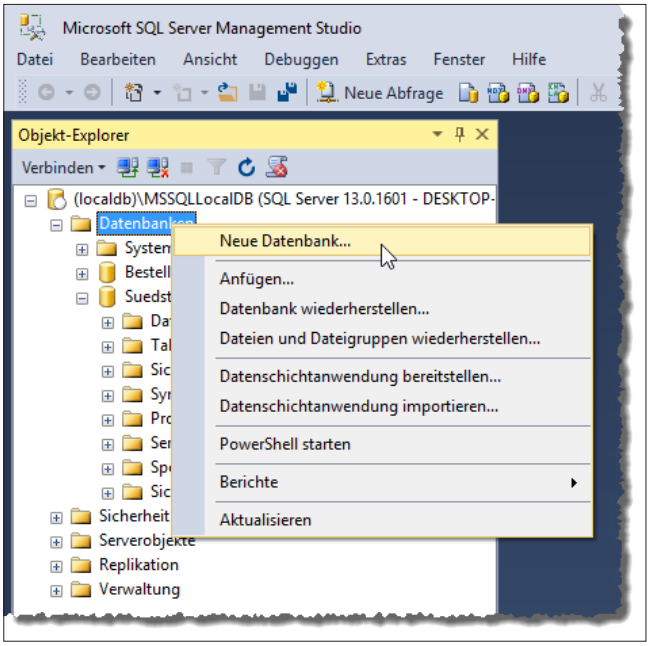

Bild 2: Anlegen einer neuen Datenbank

Im folgenden Dialog **Neue Datenbank** geben Sie den Namen der Datenbank an (**Suedsturm**). Außerdem sollten Sie noch den Speicherort für die entstehenden Dateien **Suedsturm.mdf** und **Suedsturm_log.ldf** nach Ihren Wünschen anpassen (s. Bild 3).

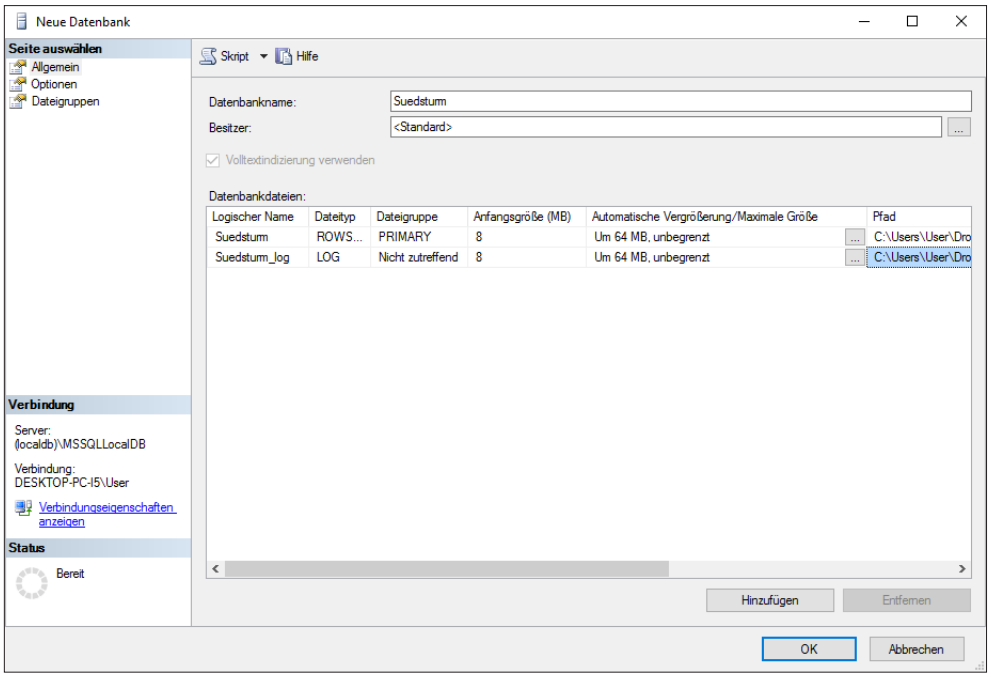

Bild 3: Festlegen von Eigenschaften wie dem Datenbanknamen und dem Speicherort

Klicken Sie dann auf **OK**, um die Datenbank zu erstellen. Die neue Datenbank erscheint nun im Objekt-Explorer. Klappen Sie den Eintrag auf, finden Sie unter anderem den Punkt **Tabellen**. Diesen klicken Sie wiederum mit der rechten Maustaste an, um aus dem Kontextmenü den Befehl **Tabelle** auszuwählen (s. Bild 4).

Es ist etwas ungewöhnlich, dass dort nicht beispielsweise **Neue Tabelle** oder **Tabelle anlegen** steht, aber damit müssen wir leben.

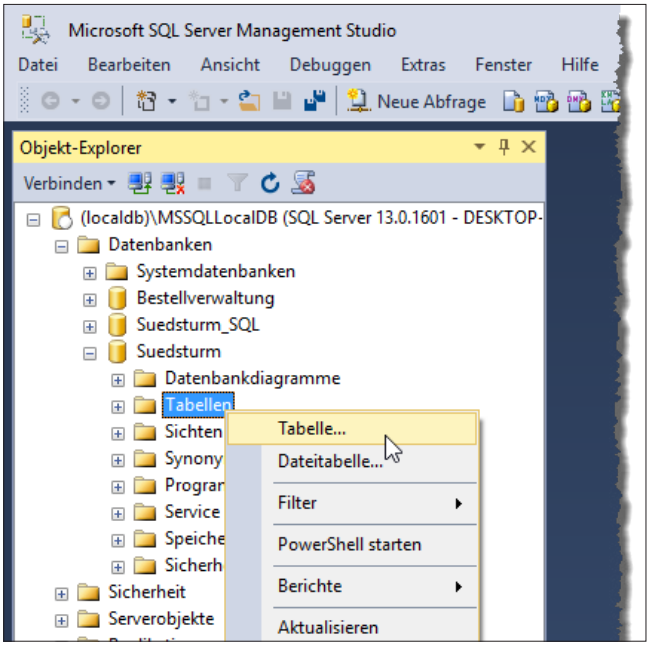

Bild 4: Anlegen einer neuen Tabelle

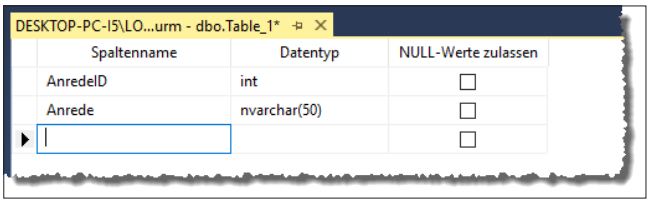

Bild 5: Entwurf der neuen Tabelle **tblAnreden**

Tabelle tblAnreden erstellen

Wir wollen die Tabellen in der Reihenfolge erstellen, in der wir sie für die Erstellung von Fremdschlüsselfeldern benötigen – die Tabellen, die nicht über Fremdschlüsselfelder mit anderen Tabellen verknüpft sind, kommen also zuerst. Den Start macht die Tabelle **tblAnreden**.

Fügen Sie im nun erscheinenden Entwurf die beiden Felder **AnredeID** und **Anrede** hinzu. Legen Sie für **AnredeID** den Felddatentyp **int** fest und für **Anrede** den Datentyp **nvarchar(255)**. Entfernen Sie für beide Felder den Haken in der Spalte **NULL-Werte zulassen**. Der Zwischenstand sieht nun wie in Bild 5 aus. Speichern Sie die Tabelle nun unter dem Namen **tblAnreden**. Nun müssen wir noch das Primärschlüsselfeld definieren und dieses als Autowertfeld definieren. Den Primärschlüssel fügen Sie einfach über den Kontextmenü-Eintrag **Primärschlüssel festlegen** des Feldes **AnredeID** hinzu (s. Bild 6).

Dann fehlt noch eine Eigenschaft, die dafür sorgt, dass neue Datensätze automatisch mit einem eindeutigen Primärschlüsselwert gefüllt werden. Unter Access heißt das **Autowert**, unter SQL Server **Identitätsspezifikation**. Diese Eigenschaft finden Sie unter dem Tabellenentwurf im Bereich **Spalteneigenschaften**.

Unterhalb des Eintrags Tabellendesigner finden Sie den Punkt **Identitätsspezifikation**, den Sie noch aufklappen können. Darunter tauchen dann die drei Eigenschaften **(Ist Identity)**, **ID-Ausgangswert** und **ID-Inkrement** auf. Stellen Sie **(Ist Identity)** auf **Ja** ein, aktiviert dies die beiden übrigen Eigenschaften, mit denen Sie nach Wunsch den Startwert und die Schrittweite für die Werte festlegen können (s. Bild 7).

Damit haben wir es geschafft – die erste Tabelle ist fertiggestellt. Wenn Sie nun erneut speichern wollen, erscheint allerdings die Meldung aus Bild 8. Manche Eigenschaften einer Tabelle erlauben es nicht, die Tabelle einfach zu speichern – diese muss dazu gelöscht und neu erstellt werden. In diesem Fall handelt es sich um das Hinzufügen der Identitätsspezifikation. Damit Sie diese Tabelle

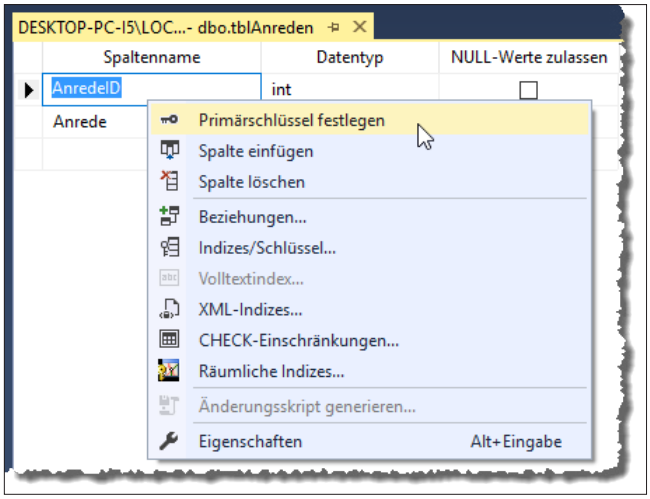

Bild 6: Definieren des Primärschlüsselfeldes

Daten von Access zum SQL Server kopieren

Manchmal benötigt man eine 1:1-Kopie der Daten aus einer Access-Tabelle in einer anderen Tabelle – vielleicht in der gleichen, vielleicht aber auch in einer anderen Datenbank. Das ist mit entsprechenden Einfügeabfragen kein Problem, solange kein anderes Datenbanksystem wie etwa der SQL Server als Ziel infrage kommt – und Sie gleichzeitig die Daten nicht nur einfach einfügen, sondern auch noch die Inhalte der Autowertfelder beibehalten wollen. Dieser Beitrag zeigt praktische Erfahrungen und Lösungsweisen für spezielle Migrationsvorhaben auf.

Warum von Tabelle zu Tabelle kopieren?

Bei mir ist die oben erwähnte Aufgabe aufgetreten, als ich die Daten einer lokalen Suedsturm-Datenbank in das neu erstellte Datenmodell einer LocalDB-SQL Server-Datenbank kopieren wollte. Ich habe die Tabellen der LocalDB-Datenbank dann in die aktuelle Access-Datenbank eingebunden, sodass der komplette Satz der Tabellen jeweils zweimal vorlag – einmal unter den richtigen Namen als lokale Tabellen und einmal mit dem Präfix **1** als eingebundene Tabellen (also etwa **tblArtikel** und **tblArtikel1**).

Kopieren per Drag and Drop

Der erste Ansatz, die Tabelleninhalte von A nach B zu kopieren, war der per Drag and Drop. Also die Quelltabelle öffnen, mit **Strg + A** alle Datensätze markieren und mit **Strg + C** kopieren und somit in die Zwischenablage einfügen. Dann die Zieltabelle öffnen und versuchen, die Daten einzufügen. Der erste Versuch: einfach wild ein Feld markiert und **Strg + V** zum Einfügen betätigt. Das gelang natürlich nicht, weil die Struktur des markierten Bereichs nicht mit dem einzufügenden Inhalt übereinstimmt. Wenn Sie dann den kompletten neuen, leeren Datensatz in der Zieltabelle markieren und erneut **Strg + V** betätigen, klappt es – die Daten landen in der Zieltabelle (s. Bild 1).

Schon fertig?

Es klappt also allein mit Bordmitteln – das wird ja ein kurzer Beitrag! Doch leider war das nicht alles, denn wir haben hier den Idealfall erwischt. Wenn Sie nun feststellen, dass irgendetwas beim Kopieren nicht wie gewünscht

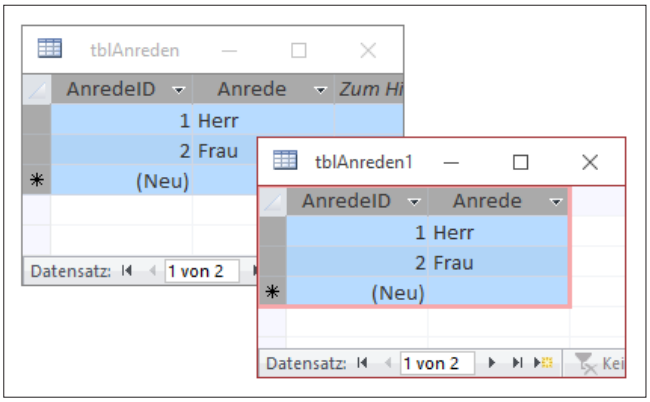

Bild 1: Kopieren der Inhalte einer Tabelle in eine baugleiche Tabelle

funktioniert hat, etwa weil eines der Zielfelder den falschen Datentyp hat oder weil Sie vielleicht noch mal neu beginnen wollen, weil die Daten der per Fremdschlüsselfeld verknüpften Daten noch gar nicht kopiert wurden, treten die ersten Probleme auf.

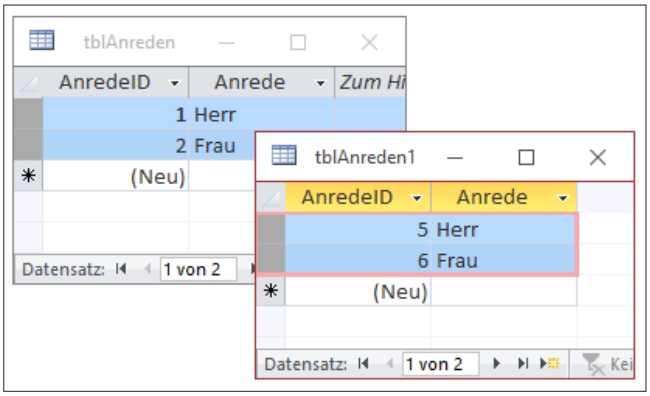

Bild 2: Erneutes Kopieren liefert andere Primärschlüsselwerte.

Wenn Sie beispielhaft die verknüpfte Tabelle **tblAnreden** leeren und die Daten erneut kopieren, erhalten Sie beispielsweise das Ergebnis aus Bild 2. Das sieht nun gar nicht mehr so gut aus, denn die Werte des Primärschlüsselfeldes **AnredeID** entsprechen nicht mehr denen aus der Originaltabelle. Dies wird zu Problemen führen, wenn Sie die Datensätze der Tabelle **tblKunden** kopieren, deren Fremdschlüsselfeld **AnredeID** ja die Werte der ursprünglichen Tabelle enthält. Hier wird dann also mit den Datensätzen der Tabelle **tblAnreden** verknüpft, welche die Werte **1** und **2** enthalten, aber nicht die mit den Werten **5** und **6**. Wir müssten also im schlimmsten Fall auch noch die Fremdschlüsselfelder anpassen und auf die neuen Werte mappen.

Sie sehen also: Wir haben nur zufällig genau die gleichen Daten wie in der Ausgangstabelle erhalten, weil die Autowert-Funktion der Zieltabelle gerade auf den gleichen Startwert wie die einzufügenden Datensätze eingestellt war.

Die Autowert-Funktion der Zieltabelle wird uns noch weitere Probleme bescheren: Sie hat ja üblicherweise die Aufgabe, vom zuletzt eingefügten Wert ausgehend die nächste Ganzzahl als neuen Wert für das Primärschlüsselfeld zu ermitteln. In der Regel werden die Datensätze also von 1 bis n durchnummeriert. Sollten wir also irgendwann einmal einen Datensatz gelöscht haben, folgt das nächste Dilemma: Die Datensätze werden nun ab diesem Loch wiederum anders nummeriert als in der Originaltabelle.

Die Lösung: INSERT INTO

Es gibt leider keine Möglichkeit, Access mitzuteilen, dass es die Werte eines mit Autowert-Funktion ausgestatteten Primärschlüsselfeldes beim Kopieren über die Benutzeroberfläche beibehalten soll. Das Einfügen von Datensätzen auf diese Art und Weise wird Access immer als manuelles Einfügen einzelner Datensätze interpretieren, und somit werden die Autowerte vom System vergeben (in diesem Fall übrigens von der LocalDB-Instanz – siehe Beitrag **Access und LocalDB**, **www.access-im-unternehmen.de/1057**). Zum Glück gibt es aber ja auch noch codegesteuerte Möglichkeiten, Datensätze zu kopieren. Also probieren wir es aus!

Also leeren wir die Tabelle **tblAnreden1** wieder, um es per VBA zu probieren. Diesmal wollen wir es mit einer **INSERT INTO**-Abfrage probieren, die einfach nur alle Felder der Quelltabelle in die entsprechenden Felder der Zieltabelle kopiert. Dazu brauchen wir noch nicht einmal alle Felder anzugeben, sondern können das Sternchen (*) als Platzhalter für alle Felder verwenden. Die Abfrage zum Kopieren der Inhalte der Tabelle **tblAnreden** sieht wie folgt aus:

```
INSERT INTO tblAnreden1 SELECT * FROM tblAnreden
```

Der hintere Teil legt fest, welche Daten eingefügt werden sollen, der vordere Teil gibt das Ziel an. Wir erstellen noch eine kleine VBA-Prozedur, welche uns das Ausführen dieser SQL-Anweisung abnimmt und zusätzlich eine Erfolgsmeldung liefert:

```
Public Sub AnredenKopieren()
    Dim db As DAO.Database
    Dim strSQL As String
    Dim lngAnzahl As Long
    Set db = CurrentDb
    strSQL = "INSERT INTO tblAnreden1 "
                                SELECT * FROM tblAnreden"
    db.Execute strSQL, dbFailOnError
    lngAnzahl = db.RecordsAffected
    MsgBox "Hinzugefügte Datensätze: " & lngAnzahl
End Sub
```

Die Abfrage speichern wir in der Variablen **strSQL**. Diese wiederum übergeben wir als ersten Parameter für die Methode **Execute** des **Database**-Objekts für die aktuelle Datenbank. Damit Fehler an die Benutzeroberfläche gemeldet werden, geben wir als zweiten Parameter den Wert **dbFailOnError** mit. Die Eigenschaft **RecordsAffected** liefert immer die Anzahl der von der letzten Aktionsabfrage betroffenen Datensätze.

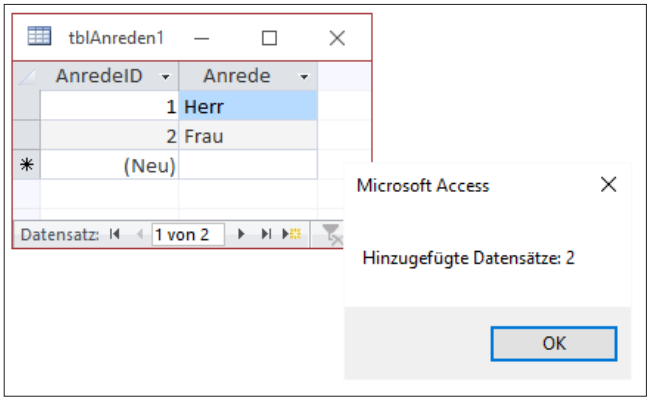

Bild 3: Mit der **INSERT INTO**-Anweisung gelingt der Kopiervorgang.

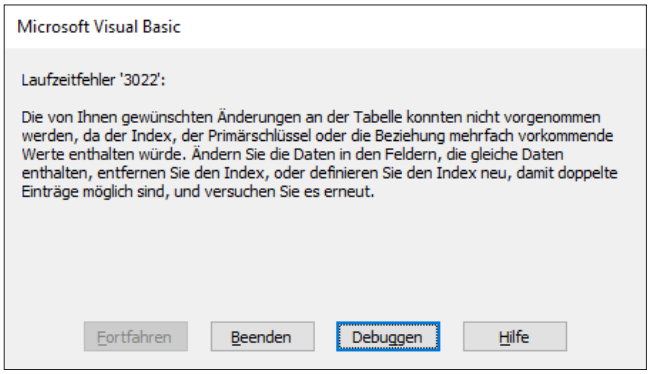

Bild 4: Fehler beim Einfügen eines Datensatzes mit einem bereits vorhandenen Primärschlüsselwert in eine lokale Tabelle

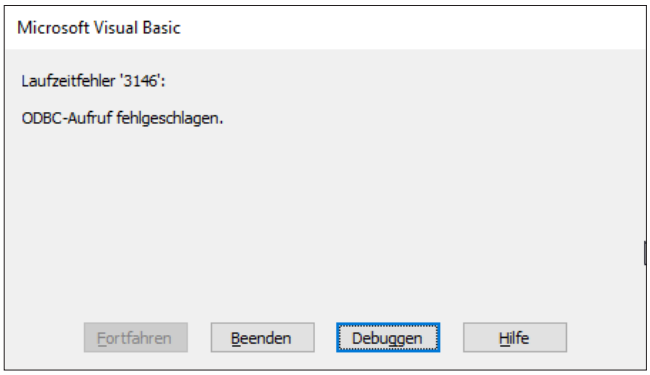

Bild 5: Fehler beim Einfügen eines Datensatzes mit einem bereits vorhandenen Primärschlüsselwert in eine per ODBC verknüpfte Tabelle

Und es gelingt – wir erhalten eine Erfolgsmeldung über zwei angefügte Datensätze und die Tabelle **tblAnreden1** zeigt die neuen Datensätze mit den Original-Primärschlüsselwerten an (s. Bild 3).

Nun bekommen wir nur noch ein Problem, wenn bereits Datensätze mit den Primärschlüsselwerten der einzufügenden Datensätze vorhanden sind.

Sollten wir also nun beispielsweise erneut die beiden Anreden kopieren, sind ja bereits zwei Datensätze mit den Primärschlüsselwerten **1** und **2** vorhanden – und da das Primärschlüsselfeld immer nur eindeutige Werte enthalten darf, sollte beim Einfügen ein Fehler ausgelöst werden.

Wenn wir dies mit einer lokalen Tabelle, also einer Access-Tabelle, als Ziel durchführen (wir haben diese **tblAnreden2** genannt), löst dies den Fehler aus Bild 4 aus.

Handelt es sich bei der Zieltabelle um eine per ODBC verknüpfte Tabelle, wie es bei der Tabelle **tblAnreden1** der Fall ist, erscheint die Meldung aus Bild 5. Während wir mit der Meldung für die lokale Tabelle noch etwas anfangen können, liefert diese Fehlermeldung nur wenig wirkliche Informationen. Allerdings können wir noch weitere Hinweise erhalten, wenn wir die VBA-Prozedur, die den Fehler ausgelöst hat, erweitern.

Fehlermeldungen aufbohren

Wenn ein Fehler durch einen Zugriff per ODBC auf eine verknüpfte Tabelle entsteht, liefert Access nämlich immer nur die Standardfehlermeldung mit der Nummer **3146**. Weitere Informationen liefert dann die **Errors**-Auflistung des **DBEngine**-Objekts. Diese enthält auf jeden Fall den auch schon von VBA gemeldeten Fehler **3146**, gegebenenfalls aber auch noch weitere Fehler. Im Falle des Anlegens von Daten mit bereits vorhandenem Primärschlüsselwert finden wir beispielsweise mit folgender Anweisung während des Debuggens des Fehlers die Anzahl **3**:

```
Debug.Print DBEngine.Errors.Count
```

Wir können dann über die **Errors**-Auflistung direkt auf die Fehlermeldungen der übrigen Fehler zugreifen.

```
Debug.Print DBEngine.Errors(1).Description
```

liefert dann beispielsweise einen weiteren Fehler mit dem folgenden Text:

```
[Microsoft][ODBC Driver 11 for SQL Server][SQL Server]The statement has been terminated.
```

Und der Fehler mit dem Index **2** lautet so:

```
[Microsoft][ODBC Driver 11 for SQL Server][SQL Server] Violation of PRIMARY KEY constraint 'PK_tblAnreden'. Cannot insert duplicate key in object 'dbo.tblAnreden'. The duplicate key value is (1).
```

Dies nutzen wir aus, indem wir die Prozedur zum Kopieren um eine ausführlichere Fehlerbehandlung ergänzen (s. Listing 1).

```
Public Sub AnredenKopieren()
    Dim db As DAO.Database
    Dim strSQL As String
    Dim lngAnzahl As Long
    Dim strFehler As String
    Dim i As Integer
    Set db = CurrentDb
    strSQL = "INSERT INTO tblAnreden1 SELECT * FROM tblAnreden"
    On Error Resume Next
    db.Execute strSQL, dbFailOnError
    Select Case Err.Number
        Case 3022
            MsgBox Err.Description
        Case 3146
            For i = DBEngine.Errors.Count - 1 To 0 Step -1
                strFehler = strFehler & DBEngine.Errors(i).Description & vbCrLf & vbCrLf
            Next i
            MsgBox strFehler
        Case 0
            lngAnzahl = db.RecordsAffected
            MsgBox "Hinzugefügte Datensätze: " & lngAnzahl
        Case Else
            MsgBox Err.Number & " " & Err.Description
    End Select
    On Error GoTo 0
End Sub
```

Listing 1: Kopieren von Tabellen mit ausführlicher Fehlerbehandlung

Hier benötigen wir zunächst zwei weitere Variablen namens **strFehler** und **i** als Laufvariable.

Vor die **Execute**-Methode setzen wir die Zeile **On Error Resume Next**, damit eventuell auftretende Fehler nicht über die eingebaute Fehlerbehandlung abgehandelt werden. Danach fragen wir dann in einer **Select Case**-Bedingung den Wert der Eigenschaft **Err.Number** ab. Lautet dieser **3146**, handelt es sich um einen ODBC-Fehler und wir wollen weitere Informationen ermitteln. Dazu hinterlegen wir eine **For...Next**-Schleife mit der Laufvariablen **i**, die alle Elemente der **Errors**-Auflistung von **DBEngine** durchläuft und diese nacheinander an die **String**-Variable **strFehler** anhängt. Dies liefert dann für den Fehler durch die bereits vorhandenen Primärschlüsselwerte den Fehler aus Bild 6.

An dieser Stelle muss man sich dann darüber im Klaren sein, welche Datensätze man überschreibt – ob es sich um bereits vorhandene Datensätze handelt, die geschützt werden müssen oder ob die Datensätze überschrieben werden können.

Für den Fall, dass Sie die Daten aus den Tabellen zweier Datenbanken zusammenführen müssen, wäre eine andere Strategie angezeigt als die in diesem Beitrag beschriebene – hier wollen wir nur die kompletten Daten einer Tabelle in eine andere, möglichst leere Tabelle übertragen. Für das Zusammenführen von Daten müssten Sie das

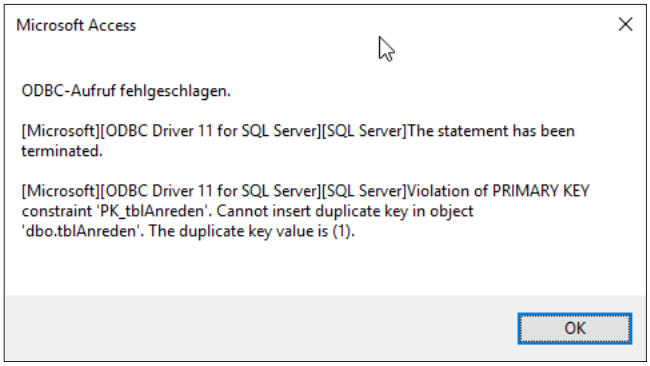

Bild 6: ODBC-Fehler mit Hintergrundinformationen

Zielsystem die jeweiligen Autowerte bestimmen lassen und die Fremdschlüsselwerte der damit verknüpften Tabellen entsprechend aktualisieren. Dies soll aber nicht Thema dieses Beitrags sein.

Reihenfolge beachten!

Wenn Sie mehrere Tabellen auf die oben beschriebene Weise kopieren möchten, können Sie das nicht in beliebiger Reihenfolge erledigen. Wenn Sie beispielsweise die Daten der Tabellen **tblArtikel** und der Tabelle **tblKategorien** und **tblLieferanten** kopieren (siehe Prozedur **ArtikelKopieren** im Modul **mdlBeispiele_DatenKopieren**), und zwar in dieser Reihenfolge, erhalten Sie den Fehler aus Bild 7.

Die Originaltabelle **tblArtikel** enthält im ersten Datensatz beispielsweise den Wert **1** im Fremdschlüsselfeld **LieferantenID**. In der verknüpften Tabelle ist aber noch gar kein Datensatz vorhanden, und somit auch keiner mit dem Wert **1** im Primärschlüsselfeld. In der Zieldatenbank ist für dieses Fremdschlüsselfeld jedoch referenzielle Integrität definiert, was bedeutet, dass das Feld **LieferantID** der Tabelle **tblArtikel** nur Werte aufnehmen darf, die bereits in der Tabelle **tblLieferanten** vorhanden sind.

Was tun wir in diesem Fall? Ganz einfach: Wir kopieren natürlich zuerst die Tabellen, welche nur einen Primärschlüssel, aber keinen Fremdschlüssel für die Verknüpfung mit anderen Tabellen enthalten. In diesem Fall kopieren wir also zuerst die beiden Tabellen **tblLieferanten** und **tblKategorien**, bevor wir die Tabelle **tblArtikel** kopieren.

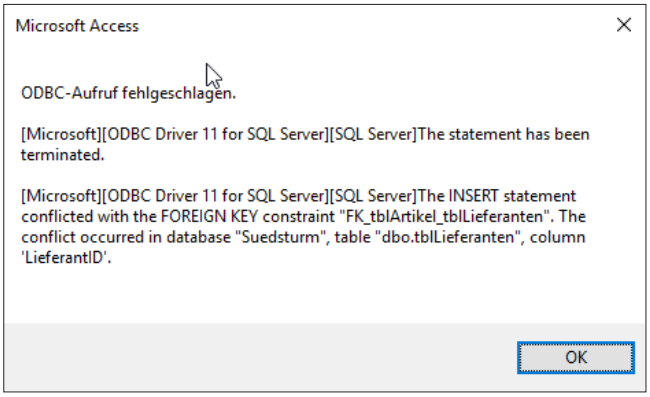

Bild 7: Fehler durch das Füllen von Fremdschlüsselfeldern ohne Vorhandensein der Datensätze mit den entsprechenden Primärschlüsselfeldern

Parametrisiertes Kopieren

Und bevor wir nun für jede Tabelle eine neue Kopie der Prozedur **AnredenKopieren** erstellen und dort nur die Zeile mit der Zuweisung der SQL-Anweisung an die Variable **strSQL** ändern, übergeben wir die variablen Daten doch lieber per Parameter. Wir verwenden also die beiden Parameter **strQuelle** und **strZiel** zur Angabe der jeweiligen Quell- und Zieltabelle. In der Variablen **strSQL** setzen wir dann die benötigte SQL-Anweisung zusammen. Der aktuelle Stand sieht nun wie in Listing 2 aus.

Zum Kopieren etwa der drei Tabellen **tblArtikel**, **tblLieferanten** und **tblKategorien** würden wir folgende Anweisungen aufrufen:

```
Public Sub Beispiel_TabellenKopieren()
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "DELETE FROM tblArtikel", dbFailOnError
    db.Execute "DELETE FROM tblKategorien", dbFailOnError
    db.Execute "DELETE FROM tblLieferanten", dbFailOnError
    TabelleKopieren "tblKategorien", "tblKategorien1"
    TabelleKopieren "tblLieferanten", "tblLieferanten1"
    TabelleKopieren "tblArtikel", "tblArtikel1"
End Sub
```

Da wir diese Prozedur sicher des öfteren testweise aufrufen, ist es sinnvoll, das Löschen der vorhandenen

Datensätze in der Zieltabelle zu Beginn durchzuführen. Danach rufen wir die Methode **TabelleKopieren** für die drei Tabellen auf.

```
Public Sub TabelleKopieren(strQuelle As String, strZiel As String)
    Dim db As DAO.Database
    Dim lngAnzahl As Long
    Dim strFehler As String
    Dim i As Integer
    Dim strSQL As String
    Set db = CurrentDb
    On Error Resume Next
    strSQL = "INSERT INTO " & strZiel & " SELECT * FROM " & strQuelle
    db.Execute strSQL, dbFailOnError
    Select Case Err.Number
        Case 3022
            MsgBox Err.Description
        Case 3146
            For i = DBEngine.Errors.Count - 1 To 0 Step -1
                strFehler = strFehler & DBEngine.Errors(i).Description & vbCrLf & vbCrLf
            Next i
            MsgBox strFehler
        Case 0
            lngAnzahl = db.RecordsAffected
            MsgBox "Hinzugefügte Datensätze: " & lngAnzahl
        Case Else
            MsgBox Err.Number & " " & Err.Description
    End Select
    On Error GoTo 0
End Sub
```

Listing 2: Kopieren von Tabellen mit Parametern für die SQL-Anweisung

Das sieht dann wie in Listing 3 aus – die zu füllenden Tabellen leeren wir vorab noch, damit der Fehler wegen bereits vorhandener Identitätswerte nicht auftritt.

Allerdings erhalten wir nach dem Start für den dritten Aufruf der Prozedur **TabelleKopieren** den Fehler aus Bild 8. Was heißt das nun wieder?

```
Cannot insert explicit value for identity column in table 'tblArtikel' when IDENTITY_INSERT is set to OFF.
```

```
Public Sub Beispiel_TabellenKopieren()
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "DELETE FROM tblArtikel1", dbFailOnError
    db.Execute "DELETE FROM tblKategorien1", dbFailOnError + dbSeeChanges
    db.Execute "DELETE FROM tblLieferanten1", dbFailOnError
    TabelleKopieren "tblKategorien", "tblKategorien1"
    TabelleKopieren "tblLieferanten", "tblLieferanten1"
    TabelleKopieren "tblArtikel", "tblArtikel1"
End Sub
```

Listing 3: Aufruf der Prozedur zum Kopieren von Tabellen

Eine kleine Recherche nach dem Schlüsselwort **IDENTITY_INSERT** ergibt, dass es normalerweise nicht vorgesehen ist, dass man die Identitätswerte eines Datensatzes selbst vergibt. Diese SQL Server-Eigenschaft hat normalerweise für alle Tabellen den Wert **OFF**. Sie kann nur für eine Tabelle gleichzeitig den Wert **ON** annehmen.

Wenn sie den Wert **ON** besitzt, muss man den Wert der Identitätsspalte mit angeben. Das hört sich logisch an – man kann so per T-SQL temporär dafür sorgen, dass man individuelle Identitätswerte eintragen kann, wenn diese sonst automatisch vom System vergeben werden. Allerdings sind unsere Tests nicht so eindeutig. Wenn Sie beispielsweise die Prozedur **Beispiel_TabelleKopieren**

ausführen, werden die ersten beiden Tabellen mit den Daten gefüllt, ohne dass wir an irgendeiner Stelle **IDENTITY_INSERT** auf **ON** stellen. Erst bei der dritten Tabelle, hier **tblArtikel**, tritt der Fehler auf.

Noch verwirrender wird es, wenn wir noch einen Kopiervorgang für die Tabelle **tblAnreden** voranstellen:

```
TabelleKopieren "tblAnreden", "tblAnreden1"
TabelleKopieren "tblKategorien", "tblKategorien1"
TabelleKopieren "tblLieferanten", "tblLieferanten1"
TabelleKopieren "tblArtikel", "tblArtikel1"
```

Dann gelingt das Einfügen in die Tabelle **tblAnreden1**, bei **tblKategorien1** erscheint der obige Fehler, das Einfügen in **tblLieferanten1** gelingt wieder und bei **tblArtikel1** gibt es wieder den Fehler.

Wenn wir die Datenbank schließen und wieder öffnen, sieht das Ergebnis mitunter noch anders aus. Wie können wir also sicherstellen, dass wir die Daten inklusive Identitätswerten in die SQL Server-Tabellen übertragen?

IDENTITY_INSERT im SQL Server Management Studio

Schauen wir uns an, ob es an Access liegt, und testen ein paar SQL-Anweisungen direkt im SQL Server Management Studio. Wenn wir die folgende Anweisung absetzen, erhalten wir den gleichen Fehler wie unter Access:

```
INSERT INTO tblAnreden(AnredeID, Anrede) VALUES(3,'bla');
```

Wir müssen zunächst angeben, dass für diese Tabelle Identitätswerte individuell geschrieben werden sollen, und zwar mit folgender Anweisung:

```
SET IDENTITY_INSERT dbo.tblAnreden ON;
```

Danach können wir dann mit der zuvor verwendeten Anweisung einen neuen Datensatz zur Tabelle **tblAnreden** hinzufügen.

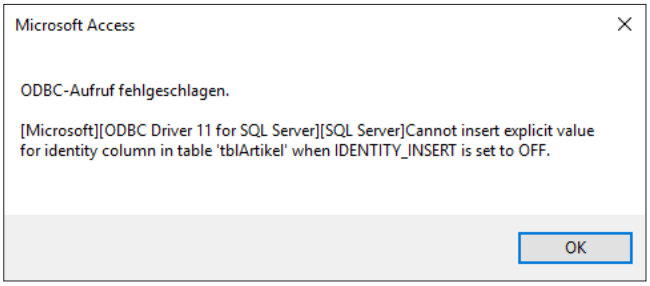

Bild 8: Fehler durch die Eigenschaft **IDENTITY_INSERT**

Hier ist das Bild also etwas eindeutiger: Wenn wir nicht zuvor **IDENTITY_INSERT** für die betroffene Tabelle auf **ON** einstellen, können wir keine individuellen Identitätswerte festlegen. Unter Access scheint dies also in manchen Fällen etwas aufgeweicht zu werden, wobei wir nicht wissen, in welchen Fällen.

Auch beeinflusst das Setzen von **IDENTITY_INSERT** vom SQL Server Management Studio aus nicht das Verhalten beim Schreiben von Daten unter Access – es scheint also sitzungsbezogen zu sein.

Wenn wir also im SQL Server Management Studio **IDENTITY_INSERT** für die Tabelle **tblKategorien** auf **ON** setzen, können wir noch lange keine Datensätze mit individuellen Identitätswerten per ODBC von Access aus hinzufügen.

Also versuchen wir, einen Weg zu finden, von Access aus zuvor anzugeben, dass die jeweilige Tabelle mit individuellen Identitätswerten gefüllt werden soll. Leider gelingt dies nicht wie gewünscht: Wir können zwar eine Pass-Through-Abfrage der folgenden Art absetzen, was erfolgreich verläuft:

```
SET IDENTITY_INSERT tblAnreden ON
INSERT INTO tblAnreden(AnredeID, Anrede) VALUES(101,
'Test')
SET IDENTITY_INSERT tblAnreden OFF
```

Allerdings legen wir damit nur einen Datensatz an, dessen Werte wir direkt in der Abfrage explizit festlegen. Wir wollen aber die Daten der lokalen Tabelle in die LocalDB-

Access und LocalDB

Seit SQL Server 2012 gibt es eine kleine, aber feine Instanz des SQL Servers namens LocalDB. Dabei handelt es sich um eine SQL Server-Engine, die zwar einige Unterschiede zur Vollversion des SQL Servers aufweist, aber auch einige Features bietet, die gerade für Access-Entwickler interessant sind. Eines vorweg: Sie können damit nur lokale Datenbanken betreiben, also solche, die auf dem gleichen Rechner wie das Frontend liegen. Ansonsten aber bietet diese Version fast alles, was das Entwicklerherz begehrt.

Der Grund, weshalb ich mich vor einiger Zeit über LocalDB informiert habe, war der, dass ich eine einfache Möglichkeit suchte, eine SQL Server-Instanz zu installieren und zu starten und ohne Umschweife damit arbeiten zu können. Ach ja – und kostenlos sollte sie auch noch sein. Wer schon einmal eine SQL Server-Installation durchgeführt hat, weiß, dass dies eine ziemlich aufwendige und zeitraubende Sache sein kann. Das ist auf jeden Fall nichts, was Sie mal eben durchführen wollen, um kurz etwas zu testen. Die schnelle Installation ist aber auch nur der wichtigste Punkt, weitere folgen weiter unten.

Download von LocalDB

Mit LocalDB verhält es sich anders: Mit der Google-Suche nach **localdb download** landen Sie schnell auf einer Seite, die den Download von **Microsoft SQL Server 2016 Express** anbietet. Diesen können Sie beziehen und die Installation starten, um dann im Auswahlmenü LocalDB zu installieren. Alternativ verwenden Sie den direkten Download der Datei **SqlLocalDB.msi**, die Sie zur Drucklegung dieses Beitrags etwa unter folgendem Link finden:

```
https://download.microsoft.com/download/E/1/2/
E12B3655-D817-49BA-B934-CEB9DAC0BAF3/SqlLocalDB.msi
```

Installation von LocalDB

Die komplette Installationsdatei ist sage und schreibe nur 45 MB groß! Die Installation brauchen wir an dieser Stelle nicht zu beschreiben, denn diese enthält keinerlei Optionen, an denen Sie Entscheidungen treffen müssten. Dafür gibt es nach der Installation aber auch keinen einzigen neuen Eintrag etwa im Startmenü von Windows. Wozu auch? Das Programm wird ohnehin nur gestartet, wenn Sie von einem Client aus auf die Instanz zugreifen.

Stille Installation

Wenn Sie möchten, können Sie die Installation auch komplett ohne Benutzeroberfläche durchführen. Dazu rufen Sie über die Kommandozeile die folgende Anweisung auf (Sie müssen sich dazu im gleichen Verzeichnis wie die Datei **SqlLocalDB.msi** befinden):

```
msiexec /i SqlLocalDB.msi /qn IACCEPTSQLLOCALDBLICENSETER
MS=YES
```

Beachten Sie, dass Sie die Eingabeaufforderung dazu als Administrator ausführen müssen. Dazu suchen Sie den Befehl **Eingabeaufforderung**, beispielsweise über das Suchfeld von Windows, und klicken dann mit der rechten Maustaste auf den gefundenen Eintrag. Dort finden Sie dann den Kontextmenü-Eintrag **Als Administrator ausführen**.

Zugriff per Eingabeaufforderung

Nun können Sie bereits über die Eingabeaufforderung auf LocalDB zugreifen. Geben Sie beispielsweise den Befehl **sqllocaldb -?** ein, um Informationen über die verfügbaren Befehle zu erhalten. Ein Befehl lautet etwa wie folgt:

```
C:\Users\User>sqllocaldb info
MSSQLLocalDB
v11.0
```

Dies gibt die Namen aller Instanzen aus. Auf diesem Rechner sind zwei verschiedene Versionen installiert – 2016 ist die aktuelle Instanz, die mit dem Namen **MSSQLLocalDB** angegeben wird. Die Instanz der Version von 2012 wird mit **v11.0** angegeben. Dies sind auch gleich die Namen, unter denen Sie in den später vorgestellten Verbindungszeichenfolgen auf die Instanzen zugreifen können.

Wenn Sie genau wissen möchten, welche Versionen installiert sind, setzen Sie folgende Anweisung ab:

```
C:\Users\User>sqllocaldb versions
Microsoft SQL Server 2012 (11.0.2100.60)
Microsoft SQL Server 2016 (13.0.1601.5)
```

Hier erhalten Sie also die vollständigen Versionshinweise.

Um eine Instanz zu starten, verwenden Sie den Start-Parameter und geben den Namen der zu startenden Instanz an:

```
C:\Users\User>sqllocaldb start "MSSQLLocalDB"
Die LocalDB-Instanz 'mssqllocaldb' wurde gestartet.
```

Mit **Stop** können Sie die Instanz anhalten:

```
C:\Users\User>sqllocaldb stop "MSSQLLocalDB"
Die LocalDB-Instanz 'mssqllocaldb' wurde beendet.
```

Hier haben wir nun die Standardinstanz gestartet. Sie können auch weitere Instanzen erzeugen. Um etwa eine Instanz namens **Beispielinstanz** zu starten, verwenden Sie den folgenden Befehl:

```
C:\Users\User>sqllocaldb create "Beispielinstanz"
Die LocalDB-Instanz 'Beispielinstanz' wurde mit Version
11.0 erstellt.
```

Eine solche Instanz können Sie natürlich auch wieder löschen:

```
C:\Users\User>sqllocaldb delete "Beispielinstanz"
Die LocalDB-Instanz 'Beispielinstanz' wurde gelöscht.
```

Wenn Sie mehrere Versionen installiert haben und eine Instanz für eine spezielle Version erstellen möchten, geben Sie die Version wie folgt an:

```
C:\Users\User>sqllocaldb
create "Beispielinstanz"
13.0
Die LocalDB-Instanz 'Bei-
spielinstanz' wurde mit
Version 13.0 erstellt.
```

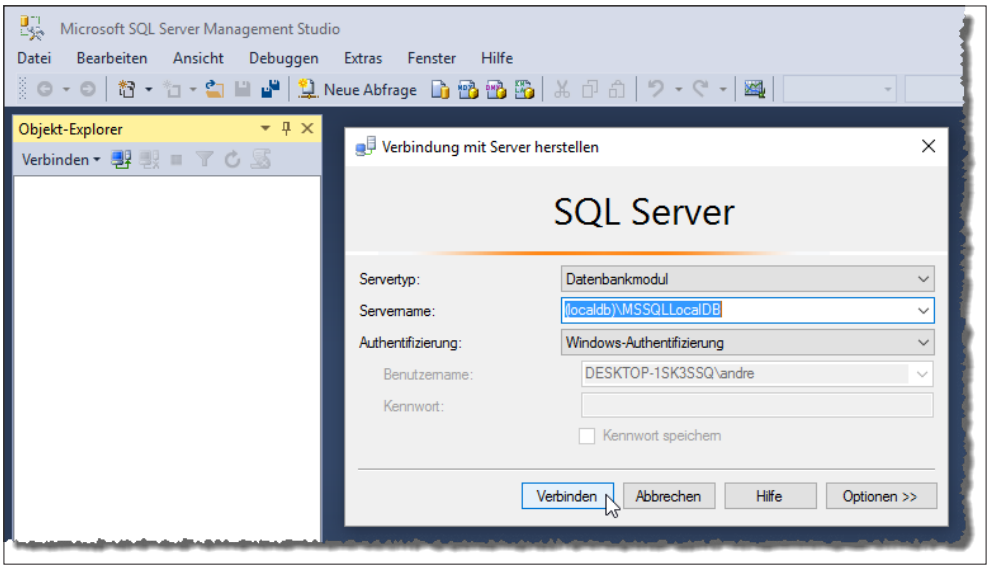

Bild 1: Das SQL Server Management Studio nach dem Start

Zugriff per SQL Server Management Studio

Für uns ist aber viel interessanter: Wie kann ich nun auf eine solche Instanz zugreifen, um eine Datenbank mit Tabellen und weiteren Objekten zu erstellen? Und wie greife ich dann von Access aus auf diese Da-

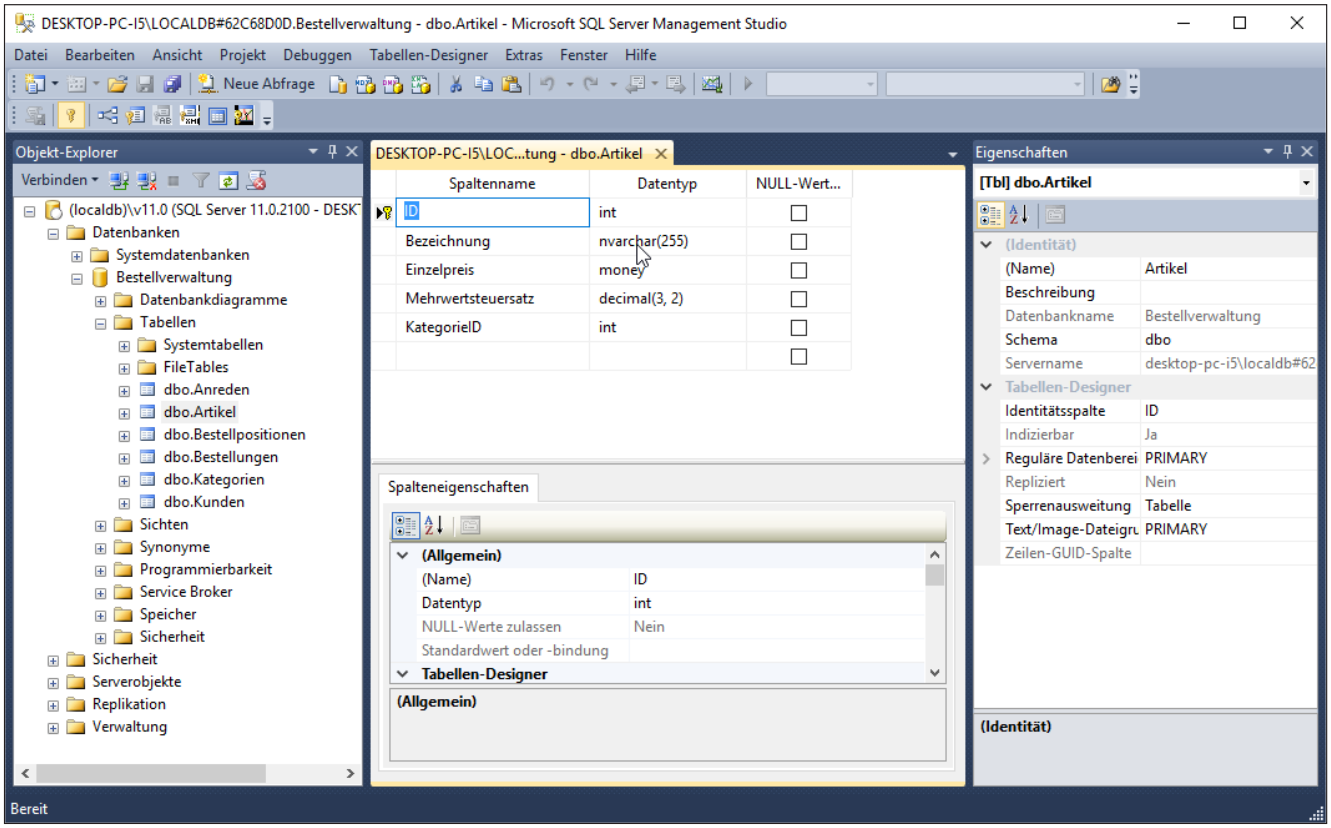

Bild 2: Bearbeiten einer Tabelle in einer LocalDB-Datenbank

tenbank zu? Auch das ist kein Problem. Unter folgendem Link finden Sie eine kostenlose Version des SQL Server Management Studio in der neuesten Version von 2016:

```
https://msdn.microsoft.com/de-de/library/mt238290.aspx
```

Nach dem Download und der Installation können Sie SQL Server Management Studio auch direkt starten. Es erscheint nun das Hauptfenster und der Dialog zum Herstellen einer Verbindung mit dem gewünschten Server (s. Bild 1). Hier tragen Sie, wenn Sie nur eine Version von **LocalDB** installiert haben, den folgenden Server ein:

```
(localdb)\MSSQLLocalDB
```

Oder, wenn Sie eine ältere, parallel installierte Instanz verwenden wollen, wie etwa auf meinem Rechner, den folgenden Server für die Version 2012:

```
(localdb)\v11.0
```

Sie können auch auf die selbst erstellte Instanz zugreifen:

```
(localdb)\Beispielinstanz
```

Grundsätzlich gilt hier die Regel, dass Sie mit dem SQL Server Management Studio nicht auf LocalDB-Versionen zugreifen können, die neuer sind als die installierte SQL Server Management Studio-Version. Die Nutzung von LocalDB in der Version von 2016 ist also mit der hier installierten Version 2014 des SQL Server Management Studios nicht möglich.

Datenbanken bearbeiten

Sie können dann nach erfolgreichem Verbindungsaufbau direkt wie mit der Vollversion des SQL Servers neue Datenbanken anlegen, Tabellen hinzufügen, Felder bearbeiten und mehr. In Bild 2 sehen Sie eine Beispielda-

tenbank, die mit dem SQL Server Management Studio auf Basis von LocalDB erstellt wurde. Wie Sie das Datenmodell einer solchen Beispieldatenbank erstellen, erfahren Sie im Beitrag **SQL Server-Datenbank erstellen** (**www.access-im-unternehmen.de/1055**).

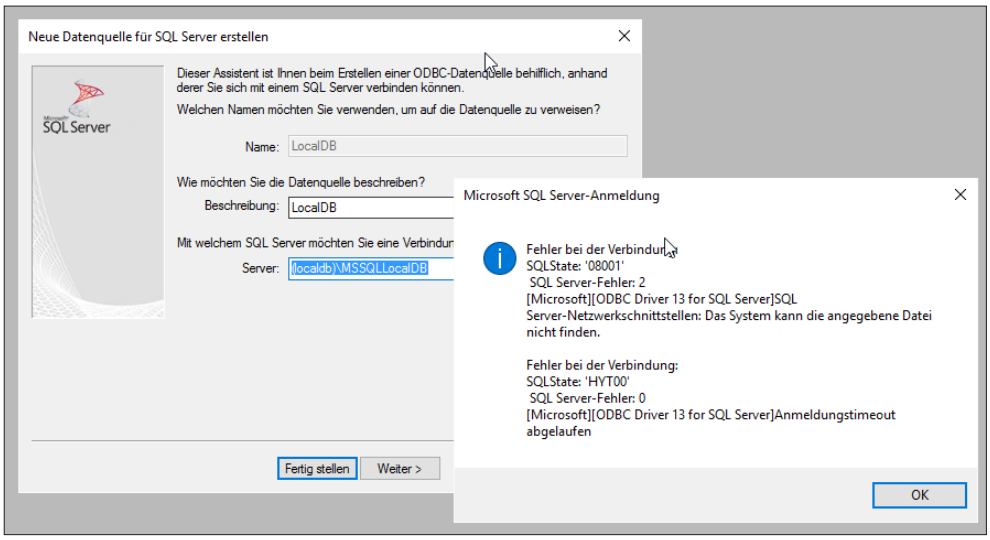

Bild 3: Fehler beim Versuch, per Datenquelle auf LocalDB zuzugreifen

Zugriff von Access auf LocalDB

Nun kommt der interessanteste Teil: Wir wollen von Microsoft Access aus eine LocalDB-Datenbank als Backend verwenden. Das gelingt, nach aktueller Empfehlung von Microsoft, am besten mit einer ODBC-Verbindung. Um eine solche herzustellen, könnten Sie mit Bordmitteln arbeiten. Wir haben das ausprobiert und den neuesten ODBC-Treiber für den SQL Server verwendet, aber der Zugriff gelingt nicht, wie Bild 3 zeigt. Das Problem liegt darin, dass wir nicht über den Parameter **SERVER**, sondern **DATA SOURCE** auf LocalDB zugreifen müssen.

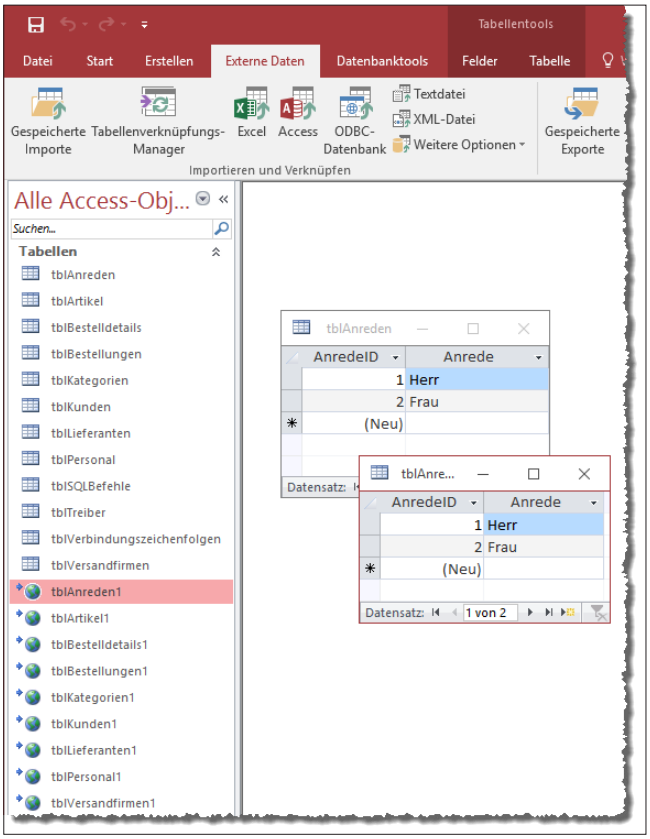

Bild 4: Lokale und verknüpfte Tabellen

Zum Glück haben wir aber ein paar Tools entwickelt, mit denen das noch schneller und besser geht. Den aktuellen Stand dieser Tools beschreiben wir im Beitrag **SQL Server-Tools** (**www.access-im-unternehmen.de/1061**) in der nächsten Ausgabe – dort sind beispielsweise die Änderungen enthalten, die für den Zugriff auf LocalDB-Datenbanken nötig sind. Zumindest erhalten Sie damit eine gangbare Verbindungszeichenfolge, die wie folgt aussieht:

```
ODBC;DRIVER={ODBC Driver 11 for SQL Server};DATA SOURCE=(localdb)\MSSQLLocalDB;DATABASE=Suedsturm;Trusted_Connection=Yes;OPTION=3;
```

Wichtig ist, dass Sie einen halbwegs aktuellen Datenbanktreiber verwenden. Das heißt also entweder mindestens den **SQL Server Native Client 11.0** (SQL Server 2012) oder den entsprechenden **ODBC Driver for SQL**

XML-Export: CDATA

In der vorherigen Ausgabe haben wir uns ausführlich mit dem Export von Tabellendaten in XML-Dokumente beschäftigt. Dabei haben wir zum Formen der Ausgabe auch Gebrauch von .xslt-Dateien gemacht, die Anweisungen zur Aufbereitung der Inhalte enthalten. In den Beiträgen haben wir uns noch keine Gedanken um den Export von Inhalten aus Text- oder Memofeldern gemacht, die als CDATA markiert werden sollen. Was CDATA überhaupt ist und wie Sie Ihre Daten als solche markieren, zeigt dieser Beitrag.

Die beiden Beiträge, in denen wir uns um den Export von Daten aus Access-Tabellen in XML-Dateien gekümmert haben, heißen **XML-Export ohne VBA** (**www.access-im-unternehmen.de/1045**) und **XML-Export mit VBA** (**www.access-im-unternehmen.de/1046**). Hier haben wir beispielsweise die Tabelle **tblKategorien**, deren Werte aus den Feldern **Beschreibung** (**Memo**) und **Abbildung** (**OLE-Objekt**) gegebenenfalls einfach in herkömmliche XML-Elemente geschrieben wurden – also etwa so:

```
<?xml version="1.0" encoding="UTF-8"?>
<dataroot xmlns:od="urn:schemas-microsoft-com:officedata"
                          generated="2016-09-04T17:33:45">
  <tblKategorien>
    <KategorieID>1</KategorieID>
    <Kategoriename>Getränke</Kategoriename>
    <Beschreibung>Alkoholfreie Getränke, Kaffee, Tee,
                                      Bier</Beschreibung>
  </tblKategorien>
  ...
</dataroot>
```

Unser Ziel ist aber nun, den Inhalt des Feldes **Beschreibung** wie folgt im XML-Dokument zu speichern:

```
<Beschreibung><![CDATA[Alkoholfreie Getränke, Kaffee, Tee,
Bier]]></Beschreibung>
```

Innerhalb des Beschreibung-Elements sollen also noch ein öffnendes **<![CDATA[** und ein schließendes **]]>** um den Inhalt herum eingefügt werden.

Was aber ist **CDATA** überhaupt und was ist unser Nutzen, wenn wir den Inhalt des Elements damit einfassen? Gelegentlich kommt es vor, dass die in der Datenbank enthaltenen Zeichenketten auch Zeichen wie das Kleiner(<)- oder Größer-Zeichen(>) enthalten. Solange sich dieses in einem Feld einer Access-Tabelle befindet, machen diese keinen Ärger. Wenn Sie einen solchen Inhalt aber in ein XML-Dokument exportieren, dann würde dies als Element-Markup interpretiert werden. Wie Sie wissen, werden das Größer- und das Kleiner-Zeichen ja zum Markieren des Beginns und des Endes eines XML-Elements verwendet (**<Beschreibung>...</Beschreibung>**).

Taucht dann innerhalb dieser Markierungen im Inhalt eines dieser Zeichen auf, wird es nicht als Zeichen, sondern als neues Elementmarkup ausgewertet. Im folgenden Beispiel war jemand faul und hat statt des Wortes **kleiner** das Kleiner-Zeichen verwendet:

```
Die Anzahl der Artikel dieser Kategorie ist < als die der
übrigen Kategorien.
```

Soll dies nun in ein XML-Dokument exportiert werden, wird Folgendes daraus:

```
<Beschreibung>Die Anzahl der Artikel dieser Kategorie ist
&lt; als die der übrigen Kategorien.</Beschreibung>
```

Das Kleiner-Zeichen wird also automatisch durch **<** ersetzt, eine sogenannte HTML-Entität. Wenn das nicht geschehen würde, weil wir das XML-Dokument bei-

spielsweise nicht über den eingebauten XML-Export erstellen, sondern per VBA, erhielten wir die folgende Zeile:

```
<Beschreibung>Die Anzahl der Artikel dieser Kategorie ist < als die der übrigen Kategorien.</Beschreibung>
```

Würden Sie diese Datei in einem XML-Editor öffnen, würde diese beispielsweise die Fehlermeldungen aus Bild 1 liefern. Das heißt also, dass wir entweder mit der Darstellung mit den Entitäten wie eben **<** für das Kleiner-Zeichen leben müssen oder aber den Inhalt als **CDATA**-Block ausgeben.

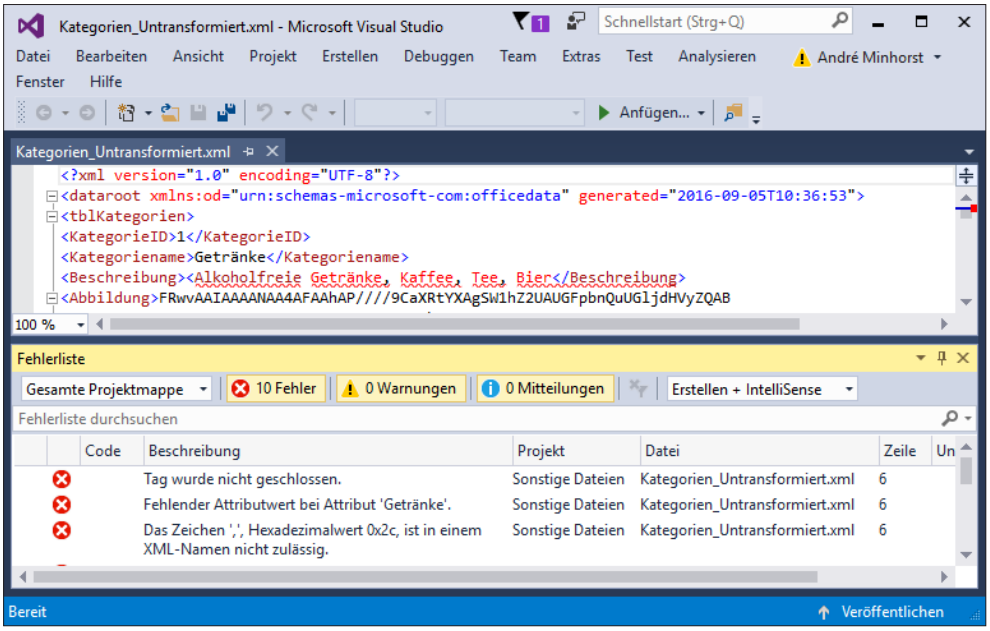

Bild 1: Fehlermeldungen beim Anzeigen einer XML-Datei mit unzulässigen Markupzeichen

Anwendungsfälle für CDATA

Während es oft durch Zufall geschieht, dass ein Zeichen in einer Tabelle landet, das für die Ausgabe in ein XML-Dokument speziell behandelt werden muss, gibt es auch Anwendungsfälle, bei denen dies regelmäßig vorkommt. Wenn Sie etwa in einem Tabellenfeld den Inhalt eines XML-Dokuments selbst speichern und dieses als Teil eines XML-Dokuments exportieren wollen, sollten Sie dieses der Einfachheit halber einfach in einen **CDATA**-Block einfassen.

Ansatz mit Bordmitteln

Die beiden oben genannten Beiträge haben die Möglichkeiten des XML-Exports direkt über die Benutzeroberfläche sowie per VBA vorgestellt. Den Weg über die Benutzeroberfläche lassen wir an dieser Stelle zunächst außen vor, da wir davon ausgehen, dass XML-Exporte, wenn überhaupt, auch regelmäßig stattfinden und dass die immer gleiche Abfolge von Schritten über den Export-Assistenten keine Dauerlösung ist. Stattdessen schauen wir uns die kleine Routine aus Listing 1 an, die den Export und die Transformation in zwei Schritten vollzieht. Der erste Schritt verwendet die **ExportXML**-Methode des **Application**-Objekts, um die Tabelle **tblKategorien** in die Datei **Kategorien_Untransformiert.xml** des aktuellen Anwendungsordners zu exportieren.

Dieses lädt die Routine dann in die Variable **objUntransformiert** und erstellt ein weiteres **DOMDocument**-Objekt, in das sie die **.xslt**-Datei **Kategorien.xslt** mit den Informationen für die Transformation füllt. Für das **DOMDocument**-Objekt **objUntransformiert** führt sie dann die Methode **transformNode** aus und übergibt die **.xslt**-Datei als ersten und die Objektvariable **objTransformiert** als zweiten Parameter. Diese soll mit der transformierten Version des Dokuments aus **objUntransformiert** gefüllt und dann unter dem Namen **Kategorien_Transformiert.xml** gespeichert werden.

Dazu schauen wir uns auch noch die **.xslt**-Datei an, die wie in Listing 2 aussieht. Die grundlegende Funktionsweise einer solchen Datei haben wir bereits im Beitrag

Explorer und Shell ansteuern

Hin und wieder kommt es vor, dass Sie aus Access heraus den Windows Explorer öffnen und dabei gleich ein Verzeichnis Ihrer Wahl ansteuern möchten. Gründe dafür gibt es viele. Haben Sie etwa eine Textdatei exportiert, so macht es sich gut, dem Anwender anschließend gleich den passenden Ordner zu präsentieren und dort die Datei zu markieren. Über einige Zeilen VBA-Code rund um ein Shell-Objekt ist das schnell realisiert.

Explorer aufrufen

Der **Explorer** ist eine Anwendung, die die Verwaltung der **Windows Shell** oder zumindest Teile von ihr, erlaubt. Wir erwähnen diesen Zusammenhang nur, weil später noch ausführlicher auf **Shell**-Objekte unter VBA eingegangen wird.

Um ausführbare Dateien zu starten, können Sie die VBA-Anweisung **Shell** verwenden. Sie übergeben ihr als Parameter den Pfad zur ausführbaren Datei und optional noch den Fenstermodus, in dem sie gestartet werden soll:

```
Shell "Explorer.exe", vbNormalFocus
```

Diese Zeile öffnet den Windows Explorer in einem normalen Fenster. Sie können dieses jedoch auch gleich maximieren:

```
Shell "Explorer.exe", vbMaximizedFocus
```

Den Pfad zur **explorer.exe** müssen Sie nicht voll ausschreiben oder kennen, da sie sich im Suchpfad von Windows befindet. War's das schon? Nein, natürlich nicht! Denn erstens gibt es noch diverse Kommandozeilenoptionen für den Explorer, die zu erläutern wären, und zweitens stellen wir noch eine Alternative zur **Shell**-Anweisung vor.

ShellExecute

Vielleicht ist Ihnen diese Funktion schon einmal untergekommen. Es gibt zwar eine **Windows-API**-Funktion gleichen Namens, doch wir verwenden sie im Zusammenhang mit der **Shell**-Bibliothek, wo sie ebenfalls vorkommt und auch identische Funktionalität aufweist.

Dazu benötigen wir einen Verweis auf diese Bibliothek. Öffnen Sie im VBA-Editor unter **Extras|Verweise** den entsprechenden Dialog und aktivieren Sie den Eintrag **Microsoft Shell Controls And Automation**. Diese Bibliothek ist grundsätzlich unter jeder Version von Windows vorhanden und verweist auf die **shell32.dll**. Nun können Sie ein **Shell**-Objekt anlegen:

```
Dim oShell As Shell32.Shell
Set oShell = New Shell32.Shell
```

Wenn Sie im VBA-Objektkatalog oben die Bibliothek **Shell32** auswählen und links zur **Shell**-Klasse navigieren, so finden Sie rechts deren zahlreiche Methoden, und deren Namen sagen bereits einiges über das Thema der Klasse aus.

Uns interessiert im Moment nur die Methode **ShellExecute**, die ähnlich arbeitet, wie die **Shell**-Anweisung von VBA, jedoch mit deutlich mehr Parametern ausgestattet ist, die die Funktionalität erweitern. Mit dieser Zeile öffnen Sie den Explorer, nachdem das **Shell**-Objekt angelegt wurde:

```
oShell.ShellExecute "explorer.exe", _
    , , , "open", 1
```

Mehrere Parameter der Methode sind optional, was durch die Kommas deutlich wird. Zunächst erwartet die Funktion

den Pfad der ausführbaren Datei, wobei dieser auch hier nicht vollständig angegeben werden muss.

```
Public m_Shell As Shell32.Shell

Public Function oShell() As Shell32.Shell
    If m_Shell Is Nothing Then Set m_Shell = New Shell32.Shell
    Set oShell = m_Shell
End Function
```

Listing 1: Funktion zum Zurückgeben des Shell-Objekts

Der zweite Parameter gibt die **Kommandozeilenoptionen** an, die der Anwendung mitgegeben werden sollen. Soll der Explorer das Verzeichnis **c:\windows** anzeigen, so ist hier der richtige Ort für die Pfadangabe:

```
oShell.ShellExecute _
    "explorer.exe", "c:\windows"
```

Damit weicht die Syntax dieser Funktion von der **Shell**-Anweisung ab, bei der beide Angaben in einem String zu kombinieren wären:

```
Shell "explorer.exe c:\windows"
```

Das macht die Sache etwas übersichtlicher. Der dritte optionale Parameter gibt den Ausführungspfad an. Wie bei Dateiverknüpfungen können Sie hier festlegen, welches Standardverzeichnis dem startenden Prozess zugeeignet wird. In der Regel kann dieser Parameter ignoriert werden.

Das Interessante an der **ShellExecute**-Funktion ist nun der vierte Parameter, der das sogenannte Aktions-**Verb** erwartet. Das sind Strings, die die auf den Prozess auszuführende Aktion angeben. Der Standard ist **open** für normales Starten der Anwendung. Lassen Sie den Parameter aus, so setzt Windows funktional automatisch **open** ein. Es gibt aber natürlich noch weitere Verben, auf die später noch die Sprache kommt.

Der letzte Parameter ist identisch mit dem der **Shell**-Anweisung und gibt den Fenstermodus an. Die **1** im Beispiel entspricht dem **vbNormalFocus**. Sie können hier also die gleichen Konstanten angeben, wie bei der **Shell**-Anweisung.

Damit Sie nicht in jeder Routine, die das **Shell**-Objekt benötigt, dieses per **New** neu anlegen müssen, greifen Sie zu einer Hilfsfunktion, wie in Listing 1. Dort ist in einem Modul die Variable **m_Shell** als globales **Shell**-Objekt deklariert.

Die Funktion **oShell** gibt dieses Objekt schlicht zurück, überprüft davor jedoch mit einem Test auf **Nothing**, ob es überhaupt gefüllt ist. Falls nicht, so wird der Variablen ein neues **Shell**-Objekt verabreicht. **oShell** können Sie nun an jeder beliebigen Stelle im VBA-Projekt verwenden, wenn Sie ein **Shell**-Objekt benötigen.

Kommandozeilenoptionen

Anhand der folgenden Prozeduren erläutern wir, wie sich der aufgerufene Explorer über Kommandozeilenoptionen steuern lässt (s. Listing 2).

Die erste Option lernten Sie bereits kennen. Sie öffnet den Explorer mit vorgegebenem Pfad (Prozedur **RunExplorerFolder**). In **RunExplorerList** ist dem Pfad **c:\windows** noch ein **/e,** vorangestellt.

Das soll nach Aussage von Microsoft den Explorer im Modus mit Verzeichnisbaum links und Dateien rechts öffnen. Ohne dieses **/e** würde sich nur der Ordner allein öffnen. Diese Aussage ist für neuere Windows-Versionen nicht mehr gültig. Die geteilte Ansicht tritt in beiden Fällen ein.

Wie daraus aber zu erkennen ist, werden mehrere Optionen durch Kommas getrennt aneinander gefügt. Gibt man als weitere Option **/root** an, so wird der folgende Pfad als Startverzeichnis des Explorers angewählt (Prozedur **Run-**

```
Sub RunExplorerFolder()
    oShell.ShellExecute "explorer.exe", "c:\windows", , "open", 1
End Sub

Sub RunExplorerList()
    oShell.ShellExecute "explorer.exe", "/e,c:\", , "open", 1
End Sub

Sub RunExplorerRoot()
    oShell.ShellExecute "explorer.exe", "/e,/root,c:\windows", , "open", 1
End Sub

Sub RunExplorerSelectFolder()
    oShell.ShellExecute "explorer.exe", _
        "/select,c:\windows\system32", , "open", 1
End Sub

Sub RunExplorerSelectFile()
    oShell.ShellExecute "explorer.exe", _
        "/select,c:\windows\system32\mscomctl.ocx", , "open", 1
End Sub
```

Listing 2: Verschiedene Routinen mit Optionen zum Starten des Explorers

ExplorerRoot). Alle anderen Elemente sind ausgeblendet. Sie können in diesem Explorer also nicht auf darüber liegende Verzeichnisse oder Laufwerke wechseln.

Die entsprechenden Schaltflächen fehlen auch im Menüband.

Schließlich lässt sich über die Option **/Select** auch direkt ein Verzeichnis oder eine Datei im sich öffnenden Explorer markieren.

Dazu fügen Sie deren Pfad unmittelbar hinter die Anweisung an (Prozeduren **RunExplorerSelectFile** und **RunExplorerSelectFolder**).

Leider lässt sich diese Option nicht mit **/root** kombinieren.

Das bedeutet: Wenn Sie Verzeichnisse oder Dateien vormarkieren, so bleibt links leider der ganze Baum bestehen.

Der Versuch, beide Optionen dennoch zu kombinieren, führt ohne Fehlermeldung schlicht zur Anzeige des Desktops im Explorer.

Möchten Sie den Explorer mit der Auswahl **Computer** starten, so greifen Sie wie in Prozedur **RunExplorer-Computer** aus Listing 3 zu einem Trick: Übergeben Sie einfach nur ein Komma als Options-String.

```
Sub RunExplorerComputer()
    oShell.ShellExecute "explorer.exe", ",", , "open", 1
End Sub

Sub OpenFileExplorer(sFile As String)
    oShell.ShellExecute "explorer.exe", "/select," & sFile, , "open", 1
End Sub

Sub RunExplorerSys()
    oShell.ShellExecute "explorer.exe", _
        "::{21EC2020-3AEA-1069-A2DD-08002B30309D}", , "open", 1
End Sub

Sub RunExplorerPrinters()
    oShell.ShellExecute "explorer.exe", "/e,/root, " & _
        "::{21ec2020-3aea-1069-a2dd-08002b30309d}\" & _
        "::{2227a280-3aea-1069-a2de-08002b30309d}", , "open", 1
End Sub
```

Listing 3: Prozeduren zum erweiterten Starten des Explorers

Die Routine **OpenFileExplorer** ist allgemeiner gehalten. Als Parameter übergeben Sie ihr in **sFile** den Pfad einer Datei oder eines Orders. Dieser wird dann im Explorer geöffnet und markiert, indem die **/Select**-Option mit der String-Variablen verkettet wird.

Nun finden Sie im Explorer ja nicht nur Elemente des Dateisystems, wie das noch bis **Windows 98** der Fall war, sondern auch solche, wie **Systemsteuerung** oder **Netzwerke**. Diese Elemente besitzen keinen Verzeichnispfad.

Und dennoch kann man sie ansteuern: Dazu bedarf es anstelle des Pfads einer speziellen **GUID**, die das Element eindeutig identifiziert. Die Routine **RunExplorerSys** etwa öffnet die Systemsteuerung, weil diese grundsätzlich die angegebene **GUID** besitzt.

Die Schreibweise ist dabei immer gleich: vor die eigentliche **GUID** kommen zwei Doppelpunkte. Diese **Shell**-Objekte sind unter dem Namen **SpecialFolders** bekannt, wobei jene mit einer **GUID** eigentlich **SpecialSpecialFolders** heißen müssten, wie wir noch sehen werden.

Da diese Elemente wiederum im Prinzip Ordner darstellen können, ist auch der Bezug auf deren Unterelemente möglich. Die Systemsteuerung enthält etwa die Liste der Drucker.

Auf sie beziehen Sie sich, indem Sie an die **GUID** eine weitere, wie einen Unterordner durch den BackSlash getrennt, anfügen. So kommen Sie in **RunExplorerPrinters** direkt in die Verzeichnisliste der Drucker des Systems.

Sie fragen sich vielleicht, wo die GUIDs herkommen? Das ist leider nicht so einfach zu beantworten. Zwar gibt es generell gültige **Shell**-Elemente mit vorgegebener GUID, wie etwa bei der Systemsteuerung, doch durch installierbare Shell-Erweiterungen sieht die Sache auf jedem System anders aus. Zum Glück können Sie über die **Shell32**-Bibliothek diese **GUIDs** für ihr System selbst ermitteln...

Shell-NameSpaces

Bisher verwendeten wir nur die eine Methode **ShellExecute** des Shell-Objekts. Eine mächtige Funktion ist daneben **NameSpace**, die uns zu neuen Klassen der Bibliothek führt.

NameSpace übergibt man einen Parameter des Typs **Variant**. Dabei kann es sich um einen String oder eine Zahl handeln. Zurück bekommt man dann ein Objekt der Klasse **Folder**.

Die Methode ist also dafür gemacht, um virtuelle Ordner-Objekte anhand von Parametern zurückzugeben. Ein einfaches Beispiel:

```
Dim fld As Shell32.Folder
Set fld = oShell.NameSpace("c:\")
```

Hier bekommen wir ein virtuelles Objekt mit Bezug auf den physischen Ordner **c:**. Virtuell deshalb, weil ein **Folder**-Objekt nicht nur ein Element des Dateisystems darstellen kann, sondern abstrakt zum Beispiel auch die Drucker der Systemsteuerung.

Das **Folder**-Objekt hat verschiedene Eigenschaften, von der die **Title**-Eigenschaft vielleicht die wichtigste ist:

```
Set fld = oShell.NameSpace("c:\")
Debug.Print fld.Title
> "(C:) Windows 7"
```

Leider gibt **Title** die Bezeichnung des Folders zurück, nicht den Pfadnamen allein. Deshalb ist ein kleiner Umstand nötig. Die Eigenschaft **Self** des **Folder**-Objekts stellt ein neues Objekt der Klasse **FolderItem** dar, welche wiederum die Eigenschaft **Path** kennt. Den Pfad eines Folders ermittelt man demnach so:

```
Set fld = oShell.NameSpace("c:\")
Debug.Print fld.Self.Path
> "c:\"
```

Statt des Strings zur Pfadangabe kann – Sie werden es sich denken können – auch eine **SpecialFolder-GUID** als Parameter übergeben werden:

```
Dim sGUID As String
sGUID = "::{21EC2020-3AEA-1069-A2DD-08002B30309D}"
Set fld = oShell.NameSpace(sGUID)
Debug.Print fld.Title
> "Systemsteuerung"
Debug.Print fld.Self.Path
> "::{21EC2020-3AEA-1069-A2DD-08002B30309D}"
```

Und schließlich wurde erwähnt, dass auch eine Zahl als Parameter für **NameSpace** dienen kann. Testen Sie das:

```
Set fld = oShell.NameSpace (0)
Debug.Print fld.Title
> "Desktop"
Debug.Print fld.Self.Path
> "C:\Users\Minhorst\Desktop"
```

Die **0** entspricht also dem Desktop-Ordner Ihres Windows-Kontos. Die Zahlen, die hier übergeben werden können, sind im Bereich von 0 bis 255. Es handelt sich um **IDs** von Windows-Spezialverzeichnissen, eben der **Special Folders**. Dies gibt uns die Möglichkeit, alle Spezialordner von Windows in einer Schleife zu enumerieren:

```
For i = 0 To 255
    Set fld = oShell.NameSpace(i)
    Debug.Print i, fld.Title
    Debug.Print i, fld.Self.Path
Next i
```

Heraus kommt dabei ungefähr das:

```
0  Desktop   C:\Users\Minhorst\Desktop
1  Internet  ::{871C5380-42A0-1069-A2EA-08002B30309D}
2  Programme C:\Users\Minhorst\AppData\Roaming\Microsoft\
Windows\Start Menu\Programs
3  Systemsteuerung ::{21EC2020-3AEA-1069-A2DD-
08002B30309D}
5  Dokumente C:\Users\Sascha\Documents
...
```

Neben der Bezeichnung der virtuellen Ordner, die teilweise lokalisiert und teilweise Englisch ist, erhalten Sie also auch den Pfad jedes Elements, der entweder als Dateisystempfad oder als **GUID** daher kommt. Auf diese Weise ermitteln Sie selbst alle Pfade von Spezialordnern von Windows, und damit kehren wir zurück zum Aufruf des Explorers.

tblShellFolders

SFID	Name	Path
13	Musik	C:\Users\Sascha\Music
14	Videos	C:\Users\Sascha\Videos
16	Desktop	C:\Users\Sascha\Desktop
17	Computer	::{20D04FE0-3AEA-1069-A2D8-08002B30309D}
18	Netzwerk	::{F02C1A0D-BE21-4350-88B0-7367FC96EF3C}
19	Network Shortcuts	C:\Users\Sascha\AppData\Roaming\Microsoft\Windows\Network Shortcuts
20	Fonts	C:\Windows\Fonts
21	Templates	C:\Users\Sascha\AppData\Roaming\Microsoft\Windows\Templates
22	Startmenü	C:\ProgramData\Microsoft\Windows\Start Menu
23	Programme	C:\ProgramData\Microsoft\Windows\Start Menu\Programs
24	Startup	C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup
25	Desktop	C:\Users\Public\Desktop
26	Roaming	C:\Users\Sascha\AppData\Roaming
27	Printer Shortcuts	C:\Users\Sascha\AppData\Roaming\Microsoft\Windows\Printer Shortcuts
28	Local	C:\Users\Sascha\AppData\Local
29	Startup	C:\Users\Sascha\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\
30	Startup	C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup

Datensatz: 1 von 56 | Kein Filter | Suchen

Bild 1: Die Windows-Spezialordner sind in der Tabelle **tblShellFolders** abgespeichert

Um den Explorer mit dem Desktop zu öffnen, können Sie einfach diese Zeile aufrufen:

```
oShell.ShellExecute _
    "explorer.exe", _
    oShell.NameSpace(0), _
    Self.Path
```

Das bedeutet, dass Sie sich um die Ermittlung des