AUSGABE 01/2026

Access, SQL und Cloud AUTOMATION

MAGAZIN FÜR DIE PROGRAMMIERUNG VON MICROSOFT ACCESS, SQL SERVER UND CLOUD-AUTOMATIONEN MIT VBA UND CO.

IN DIESEM HEFT:

André Minhorst Verlag

Timer ohne Formular mit VBA programmieren

Unter Access kennt man eigentlich nur den Timer, den das Formular-Objekt mitbringt. Damit können wir ein Ereignis definieren, das in einem bestimmten Zeitintervall ausgelöst wird. Was aber, wenn wir einen Timer einmal außerhalb des Kontexts eines Formulars benötigen? Dann erstellen wir uns einfach eine eigene Timer-Klasse. Dazu sind zwar ein paar Tricks und API-Funktionen nötig, aber das soll kein Hindernis sein. Deshalb zeigen wir in diesem Beitrag im Detail, wie das gelingt und wie sich dieser Timer in der Praxis einsetzen lässt. Und im Gegensatz zum Formulartimer haben wir noch einen Vorteil: Wir können nämlich nicht nur einen, sondern beliebig viele Timer einsetzen und laufen lassen.

Klassischer Formular-Timer in Access

Wenn wir in Access einen Timer benötigen, ist das eigentlich nur über den Formular-Timer möglich. Dazu fügt man einem Formular das Ereignis **Bei Zeitgeber** hinzu und legt mit der Eigenschaft **Zeitgeberintervall** fest, in welchen Intervallen dieses Ereignis ausgelöst werden soll (siehe Bild 1).

In der Ereignisprozedur tragen wir den Code ein, der beim Eintreten des Ereignisses **Form_Timer** ausgelöst werden soll. In diesem Fall aktualisieren wir alle 1.000 Millisekunden die Anzeige der Uhrzeit:

```
Private Sub Form_Timer()
    Me.txtUhrzeit = Time()
End Sub
```

Wenn der Timer nicht mehr ausgelöst werden soll, stellen wir die Eigenschaft **Zeitgeberintervall** per VBA auf **0** ein:

```
Me.TimerInterval = 0
```

Wenn mehr als ein Timer benötigt wird ...

Ein Timer ist allerdings manchmal nicht ausreichend. Wenn man nicht mehrere Formulare mit einem Timer ausstatten will, muss man sich also eine andere Lösung überlegen. Außerdem gibt es auch Anwendungsfälle, in denen gerade kein Formular geöffnet ist, das den Timer zur Verfügung stellen kann. Was ist zum Beispiel, wenn man regelmäßig einen bestimmten Status im Ribbon aktualisieren möchte?

Für solche Anwendungsfälle stellen wir nun eine Lösung vor, mit der wir bis zu 100 Timer unabhängig voneinander starten, auslösen und beenden können.

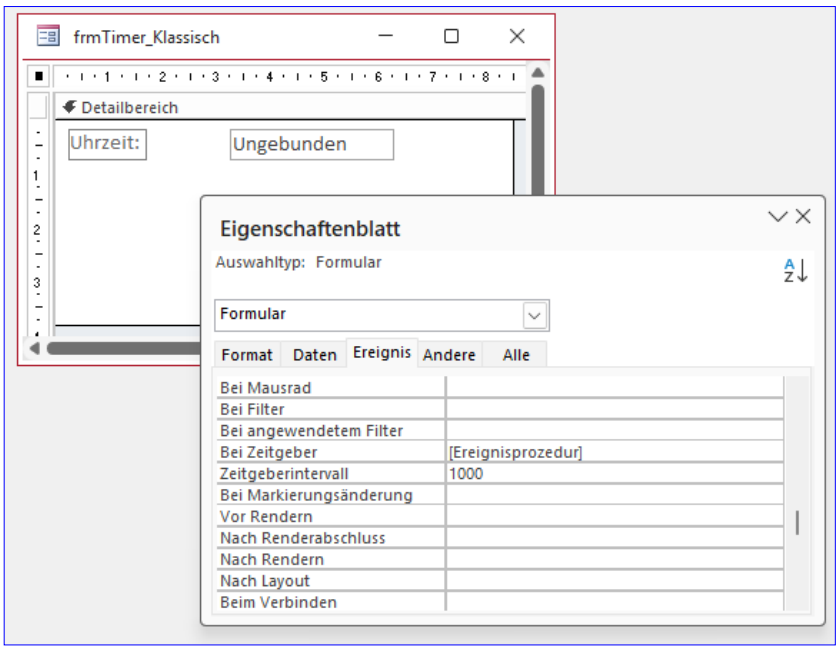


Bild 1: Einfacher Formulartimer

```
Public Event TimerEvent()
Private L_INTERVAL As Long
Private L_ID As LongPtr
Friend Sub ErrRaise(E As Long)
    Dim sText As String
    Dim sSource As String
    If E > 1000 Then
        sSource = "Application" & ".WindowProc"
        Select Case E
            Case eTooManyTimers
                sText = "No more than 1000 timers allowed per class"
            Case eCantCreateTimer
                sText = "Can't create system timer"
        End Select
        Err.Raise E Or vbObjectError, sSource, sText
    Else
        Err.Raise E, sSource
    End If
End Sub
Property Get Interval() As Long
    Interval = L_INTERVAL
End Property
Property Let Interval(lValue As Long)
    Dim b As Boolean
    If lValue > 0 Then
        If L_INTERVAL = lValue Then Exit Property
        If L_INTERVAL Then
            b = TimerDestroy(Me)
            Debug.Assert b
        End If
        L_INTERVAL = lValue
        If TimerCreate(Me) = False Then
            ErrRaise eCantCreateTimer
        End If
    Else
        If (L_INTERVAL > 0) Then
            L_INTERVAL = 0
            b = TimerDestroy(Me)
            Debug.Assert b
        End If
    End If
End Property
Public Sub RaiseInterval()
    RaiseEvent TimerEvent
End Sub
Friend Property Get TimerID() As LongPtr
    TimerID = L_ID
End Property
Friend Property Let TimerID(ID As LongPtr)
    L_ID = ID
End Property
Private Sub Class_Terminate()
    Interval = 0
End Sub
```

Listing 1: Klassenmodul **CLS_AMV_Timer**

Dazu benötigen wir eine Klasse, die den Timer selbst enthält. Zudem benötigen wir ein paar Zeilen Code, um die Timer zu initialisieren und die dadurch ausgelösten Ereignisse zu implementieren.

Die Klasse CLS_AMV_Timer

Die Klasse, welche die Timer bereitstellt, heißt **CLS_AMV_Timer** und ist in Listing 1 zu finden. Sie deklariert zunächst das Ereignis, das durch den Timer ausgelöst wird:

```
Public Event TimerEvent()
```

Die Variablen **L_INTERVAL** und **L_ID** speichern das Intervall eines Timers und die ID:

```
Private L_INTERVAL As Long
Private L_ID As LongPtr
```

Die Prozedur **ErrRaise** wird beim Auftreten eines Fehlers aufgerufen.

Über die **Property Get**-Prozedur Interval können wir das Intervall eines Timers abrufen, mit der entsprechenden **Property Let**-Prozedur setzen wir das Intervall.

Dabei übergeben wir die Anzahl der Millisekunden, nach denen der Timer ausgelöst werden soll. Die Prozedur prüft, ob das Intervall größer als **0** ist, und stellt dieses ein. Wenn das zugewiesene Intervall gleich dem zuvor eingestellten Intervall ist, wird die Prozedur verlassen. Anderenfalls rufen wir die Prozedur **TimerDestroy** auf, die im Standardmodul **MDL_AMV_Timer** enthalten ist (siehe weiter unten).

Die Methode **RaiseInterval** löst das Ereignis **TimerEvent** aus, das wir im implementierenden Klassenmodul definieren. Schließlich gibt es noch zwei **Property**-Prozeduren, mit denen die **TimerID** gelesen und geschrieben werden kann, und die Methode **Class_Terminate**, welche das Intervall auf **0** Millisekunden einstellt.

```
Public Enum eTimerError
    eBaseTimer = 10100
    eTooManyTimers = 10101
    eCantCreateTimer = 10102
End Enum

#If Win64 Then
    Public Declare PtrSafe Function SetTimer Lib "user32" (ByVal hWnd As LongPtr, ByVal nIDEvent As LongPtr, _
        ByVal uElapse As Long, ByVal lpTimerFunc As LongPtr) As LongPtr
    Public Declare PtrSafe Function KillTimer Lib "user32" (ByVal hWnd As LongPtr, ByVal nIDEvent As LongPtr) As Long
#Else
    Public Declare Function SetTimer Lib "user32" (ByVal hWnd As Long, ByVal nIDEvent As Long, _
        ByVal uElapse As Long, ByVal lpTimerFunc As Long) As Long
    Public Declare Function KillTimer Lib "user32" (ByVal hWnd As Long, ByVal nIDEvent As Long) As Long
#End If

Private Const cTimerMax As Long = 100
Private aTimers(1 To cTimerMax) As CLS_AMV_TIMER
Private cTimerCount As Integer
```

Listing 2: Deklarationen im Modul **MDL_AMV_Timer**

ADODB: Datenbankinformationen mit OpenSchema

Wenn Du per VBA auf eine Datenbank zugreifst, stellt sich manchmal die Frage: Welche Tabellen gibt es eigentlich? Welche Spalten hat eine bestimmte Tabelle und welche Datentypen verwenden diese? Gibt es Indizes oder Primärschlüssel? Die OpenSchema-Methode der Connection-Klasse aus der ADODB-Bibliothek liefert genau solche Metadaten – und zwar unabhängig davon, ob Du mit einer Access-Datenbank oder einem SQL Server arbeitest. In diesem Artikel zeigen wir, wie Du mit OpenSchema die Struktur einer Datenbank abfragen kannst. Dabei stellen wir die wichtigsten Schema-Typen vor und zeigen für jeden ein praktisches Beispiel.

Beispieldatenbank

Die Beispiele dieses Artikels findest Du in der Beispieldatenbank **ADODB_OpenSchema.accdb**. Diese enthält ein Modul **mdlOpenSchema** mit allen hier vorgestellten Prozeduren. Die Datenbank enthält außerdem einige Beispieltabellen, auf deren Struktur wir mit **OpenSchema** zugreifen.

Was ist OpenSchema?

OpenSchema ist eine Methode des **Connection**-Objekts von ADODB. Sie liefert ein **Recordset** zurück, das Metadaten über die Struktur der verbundenen Datenbank enthält. Das können Informationen über Tabellen, Spalten, Indizes, Primärschlüssel, gespeicherte Prozeduren und vieles mehr sein.

Der Aufruf sieht grundsätzlich so aus:

```
Dim rst As ADODB.Recordset
Set rst = cnn.OpenSchema(SchemaTyp)
```

Der Parameter **SchemaTyp** ist ein Wert der Enumeration **SchemaEnum**. Dieser bestimmt, welche Art von Metadaten abgerufen wird.

Optional können wir einen zweiten Parameter **Criteria** übergeben, um die Ergebnisse einzuschränken – dazu kommen wir weiter unten.

Voraussetzungen

Für die Verwendung von **OpenSchema** benötigst Du einen Verweis auf die Bibliothek **Microsoft ActiveX Data Objects 6.1 Library**. Diesen fügst Du im VBA-Editor über **Extras|Verweise** hinzu (siehe Bild 1).

Außerdem benötigst Du ein geöffnetes **Connection**-Objekt. Innerhalb einer Access-Datenbank kannst Du dafür einfach **CurrentProject.Connection** verwenden. Für eine Verbindung zum SQL Server erstellst Du ein eigenes **Connection**-Objekt mit der passenden Verbindungszeichenfolge, wie wir es im Artikel

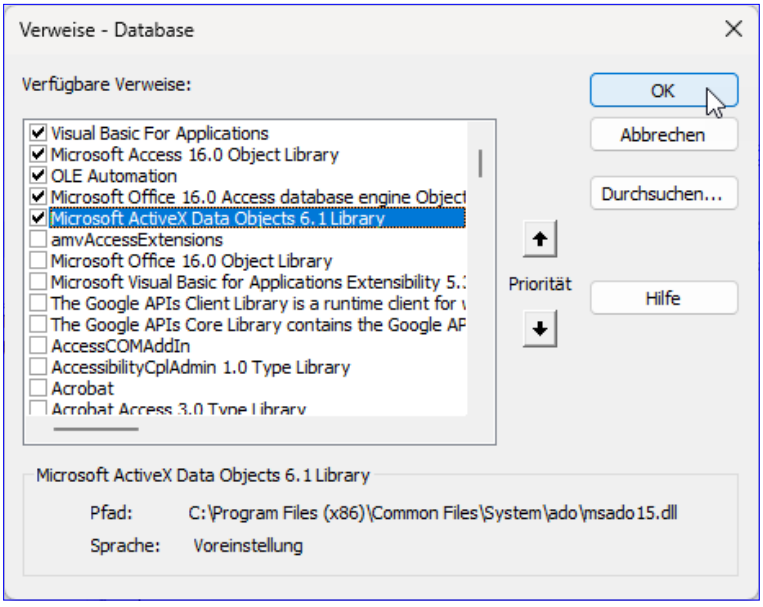


Bild 1: Verweis auf die ADODB-Bibliothek

ADODB: Die Connection-Klasse (**www.vbentwickler.de/446**) beschrieben haben.

Übersicht der wichtigsten Schema-Typen

Die Enumeration **SchemaEnum** enthält eine große Anzahl von Konstanten. Die folgenden sind in der Praxis besonders relevant:

- **adSchemaTables** (**20**): Listet alle Tabellen und Sichten (Views) der Datenbank auf.
- **adSchemaColumns** (**4**): Liefert alle Spalten aller Tabellen mit Datentyp, Länge und weiteren Informationen.
- **adSchemaIndexes** (**12**): Zeigt alle Indizes der Datenbank mit den zugehörigen Spalten und Eigenschaften an.
- **adSchemaPrimaryKeys** (**28**): Liefert die Primärschlüssel aller Tabellen.
- **adSchemaForeignKeys** (**27**): Zeigt alle Fremdschlüsselbeziehungen zwischen den Tabellen.
- **adSchemaProcedures** (**16**): Listet gespeicherte Prozeduren auf. Bei Access sind dies die gespeicherten Abfragen.
- **adSchemaViews** (**23**): Zeigt die Sichten (Views) der Datenbank an.

In den folgenden Abschnitten stellen wir die wichtigsten dieser Schema-Typen mit je einem vollständigen Codebeispiel vor.

Tabellen einer Datenbank auflisten

Der wohl häufigste Anwendungsfall ist das Auflisten aller Tabellen einer Datenbank. Dazu verwenden wir die Konstante **adSchemaTables**. Das zurückgegebene Recordset enthält unter anderem die folgenden Felder:

- **TABLE_CATALOG**: Name der Datenbank
- **TABLE_SCHEMA**: Schema der Tabelle (bei Access meist leer, bei SQL Server zum Beispiel **dbo**)
- **TABLE_NAME**: Name der Tabelle
- **TABLE_TYPE**: Art des Objekts, zum Beispiel **TABLE**, **VIEW**, **SYSTEM TABLE** oder **ACCESS TABLE**

Die Prozedur in Listing 1 gibt alle Tabellen der aktuellen Access-Datenbank im Direktbereich aus.

```
Public Sub TabellenAuflisten()
    Dim cnn As ADODB.Connection
    Dim rst As ADODB.Recordset
    Dim strConnection As String
    Dim strServer As String
    Dim strDatabase As String

    strServer = "amvDesktop2023"
    strDatabase = "Mitarbeiterverwaltung"
    Set cnn = New ADODB.Connection
    strConnection = "Provider=MSOLEDBSQL;Data Source=" _
        & strServer & ";Initial Catalog=" & strDatabase _
        & ";Integrated Security=SSPI;"
    cnn.Open strConnection
    Set rst = cnn.OpenSchema(adSchemaTables)

    Do While Not rst.EOF
        Debug.Print rst!TABLE_NAME, rst!TABLE_TYPE
        rst.MoveNext
    Loop

    rst.Close
    Set rst = Nothing
    cnn.Close
    Set cnn = Nothing
End Sub
```

Listing 1: Alle Tabellen der Datenbank auflisten

Im Direktbereich erscheint dann eine Auflistung aller Tabellen mit dem jeweiligen Typ. Dabei wirst Du feststellen, dass neben den von Dir angelegten Tabellen auch Systemtabellen erscheinen, die Access intern verwendet, und Views.

```
Public Sub NurTabellen()
    ...
    cnn.Open strConnection
    Set rst = cnn.OpenSchema(adSchemaTables, Array(Null, Null, Null, "TABLE"))

    Do While Not rst.EOF
        Debug.Print rst!TABLE_NAME, rst!TABLE_TYPE
        rst.MoveNext
    Loop

    rst.Close
    Set rst = Nothing
End Sub
```

Listing 2: Nur benutzerdefinierte Tabellen anzeigen

Ergebnisse filtern mit Criteria

Oft benötigst Du nicht alle Informationen, die **OpenSchema** liefert. In diesem Fall kannst Du den optionalen Parameter **Criteria** verwenden. Dieser erwartet ein **Array** mit Filterwerten, wobei die Anzahl und Bedeutung der Array-Elemente vom jeweiligen Schema-Typ abhängt.

Für **adSchemaTables** akzeptiert **Criteria** ein Array mit vier Elementen in der folgenden Reihenfolge:

- **TABLE_CATALOG**: Datenbankname (oder **Null** für alle)
- **TABLE_SCHEMA**: Schemaname (oder **Null** für alle)
- **TABLE_NAME**: Tabellenname (oder **Null** für alle)
- **TABLE_TYPE**: Tabellentyp (oder **Null** für alle)

Wenn wir beispielsweise nur die regulären Tabellen anzeigen wollen und keine Systemtabellen oder Sichten, können wir das wie in Listing 2 formulieren.

Hier übergeben wir für die ersten drei Elemente des Arrays den Wert **Null**, da wir keine Einschränkung für Katalog, Schema oder Tabellenname vornehmen möchten. Das vierte Element setzen wir auf den Wert **TABLE**, wodurch nur reguläre Tabellen zurückgegeben werden.

Spalten einer Tabelle ermitteln

Mit **adSchemaColumns** können wir die Spalten einer Tabelle samt Datentyp, Länge und weiteren Eigenschaften abfragen. Das ist besonders nützlich, wenn Du zur Laufzeit die Struktur einer Tabelle analysieren musst – zum Beispiel für einen generischen Export oder eine dynamische Formularerstellung.

Das zurückgegebene Recordset enthält unter anderem diese Felder:

- **COLUMN_NAME**: Name der Spalte
- **DATA_TYPE**: Numerischer Wert für den Datentyp (entspricht den ADODB-Typkonstanten wie **adInteger**, **adVarWChar** und so weiter)
- **CHARACTER_MAXIMUM_LENGTH**: Maximale Zeichenlänge bei Textspalten
- **IS_NULLABLE**: Gibt an, ob die Spalte Null-Werte erlaubt
- **ORDINAL_POSITION**: Position der Spalte in der Tabelle

Der **Criteria**-Parameter für **adSchemaColumns** akzeptiert ein Array mit vier Elementen:

TABLE_CATALOG, **TABLE_SCHEMA**, **TABLE_NAME** und **COLUMN_NAME**.

In Listing 3 filtern wir die Ergebnisse auf eine bestimmte Tabelle.

```
Public Sub SpaltenErmitteln()
    ...
    Set rst = cnn.OpenSchema(adSchemaColumns, Array(Null, Null, "tblKunden", Null))

    Do While Not rst.EOF
        Debug.Print rst!COLUMN_NAME, rst!DATA_TYPE, rst!CHARACTER_MAXIMUM_LENGTH
        rst.MoveNext
    Loop

    rst.Close
    Set rst = Nothing
End Sub
```

Listing 3: Spalten einer Tabelle ermitteln

Die Ausgabe zeigt für jede Spalte den Namen, den numerischen Datentyp und die maximale Zeichenlänge. Bei numerischen Spalten ist die Zeichenlänge **Null** (siehe Bild 2).

Datentyp-Nummern lesbar machen

Die Spalte **DATA_TYPE** liefert numerische Werte, die den Konstanten der Enumeration **DataTypeEnum** entsprechen.

Um die Ausgabe lesbarer zu gestalten, können wir eine Hilfsfunktion erstellen, die den numerischen Wert in einen lesbaren Namen umwandelt (siehe Listing 4).

Diese Funktion kannst Du in der Prozedur **SpaltenErmitteln** verwenden, indem Du die Zeile mit **rst!DATA_TYPE** wie folgt anpasst:

```
Debug.Print rst!COLUMN_NAME,
    DatentypName(rst!DATA_TYPE),
    rst!CHARACTER_MAXIMUM_LENGTH
```

Damit erhalten wir das Ergebnis aus Bild 3.

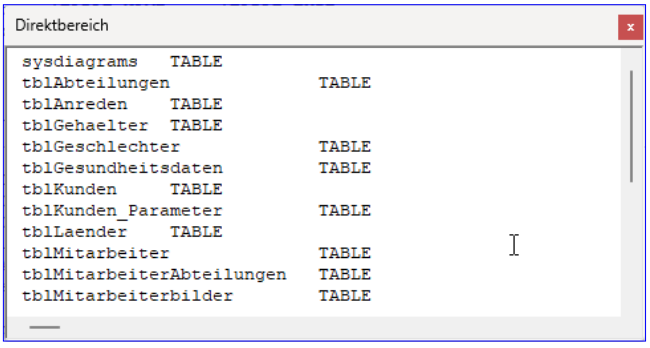


Bild 2: Tabellen einer Datenbank

Indizes einer Tabelle abfragen

Mit **adSchemaIndexes** kannst Du herausfinden, welche Indizes für eine Tabelle definiert sind. Das ist

```
Public Function DatentypName( _
        ByVal lngTyp As Long) As String
    Select Case lngTyp
        Case 2: DatentypName = "SmallInt"
        Case 3: DatentypName = "Integer"
        Case 4: DatentypName = "Single"
        Case 5: DatentypName = "Double"
        Case 6: DatentypName = "Currency"
        Case 7: DatentypName = "Date"
        Case 11: DatentypName = "Boolean"
        Case 17: DatentypName = "Byte"
        Case 72: DatentypName = "GUID"
        Case 130: DatentypName = "Text (WChar)"
        Case 202: DatentypName = "VarWChar"
        Case 203: DatentypName = "LongVarWChar"
        Case 205: DatentypName = "LongVarBinary"
        Case Else
            DatentypName = "Typ " & lngTyp
    End Select
End Function
```

Listing 4: Hilfsfunktion zum Umwandeln der Datentypen

Access und SQL Server-FileTables

Im Artikel » Access und SQL Server-FileTables« (www.vbentwickler.de/489) haben wir gezeigt, wie man einer SQL Server-Datenbank eine sogenannte FileTable-Tabelle hinzufügt, in der man Dateien speichern kann, die gleichzeitig in einem vom SQL Server verwalteten Verzeichnis liegen. Im vorliegenden Artikel kommt nun Microsoft Access als Frontend ins Spiel, mit dem wir nicht nur die Dateien in der FileTable-Tabelle im SQL Server verwalten wollen, sondern wir möchten diese am Beispiel von Bilddateien auch in Access-Formularen anzeigen. Letzteres ist leicht realisierbar, denn wir können dem Bild-Steuerelement einfach den Pfad zu der jeweiligen Datei in dem von SQL Server verwalteten Bereich des Dateisystems zuweisen. Etwas aufwendiger ist es, erst einmal über Access an diese Daten in der FileTable-Tabelle zu gelangen. Wie dies gelingt und wie wir die darin gespeicherten Dateien letztlich verwalten können, zeigen wir auf den folgenden Seiten.

Zugriff auf FileTables von Access über eine per ODBC verknüpfte Tabelle

Erst einmal die schlechte Nachricht vorneweg: Ein Zugriff auf die **FileTable**-Tabelle als eingebundene Tabelle in Access ist nicht möglich.

Die Tabelle lässt sich zwar einbinden und wir können uns auch den Entwurf ansehen (siehe Bild 1).

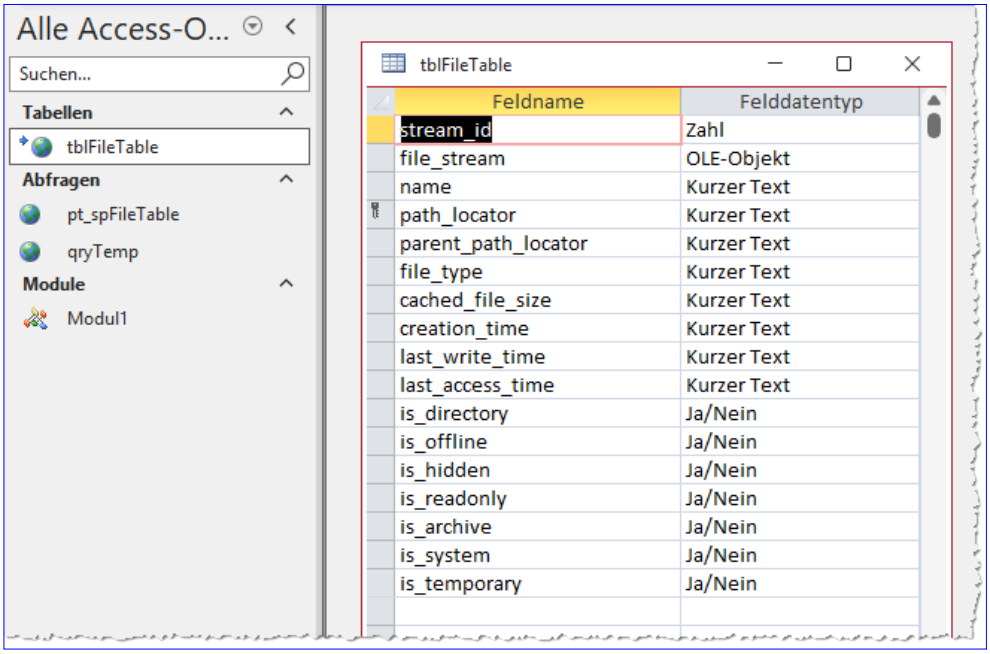


Bild 1: Entwurfsansicht einer per ODBC eingebundenen **FileTable**-Tabelle

Aber wir erhalten keinen Zugriff auf die Daten, sondern die Fehlermeldung **ODBC-Aufruf fehlgeschlagen**, wenn wir versuchen, die Tabelle in der Datenblattansicht anzuzeigen (siehe Bild 2).

Woran liegt dies und wie kann man das ändern? Es sollte einen Weg geben, denn Microsoft propagiert ja nicht umsonst seit Jahren, dass man für den Zugriff auf SQL Server-Datenbanken ODBC und DAO nutzen soll.

Mit ADODB können wir problemlos auf die Daten dieser Tabelle zugreifen.

Aber nun wenden wir uns wieder der Frage zu, woran der Zugriff per ODBC-verknüpfter Tabelle und per DAO scheitert.

Das Problem wird durch die beiden Felder **path_locator** und **parent_path_locator** der

FileTable-Tabelle ausgelöst. Diese haben den Datentyp **hierarchyid**, der interessanterweise eine Methode namens **ToString()** bereitstellt. Und siehe da – mit einer gespeicherten Prozedur gelingt auch der Zugriff per DAO. Diese legen wir wie folgt für die Datenbank mit der **FileTable**-Tabelle an:

```
CREATE PROC [dbo].[spFileTable_name]
AS
SELECT stream_id, file_stream, name,
  path_locator.ToString() AS PathLocator,
  parent_path_locator.ToString() AS ParentPathLocator,
  file_type, is_directory FROM dbo.tblFiletable;
```

Danach legen wir eine neue Abfrage an, die wir in eine PassThrough-Abfrage umwandeln. In dieser legen wir den folgenden SQL-Befehl fest:

```
EXEC spFileTable_name
```

Außerdem hinterlegen wir für die Eigenschaft **ODBC-Verbindung** die Verbindungszeichenfolge zum SQL Server, die beispielsweise wie in Bild 3 aussieht.

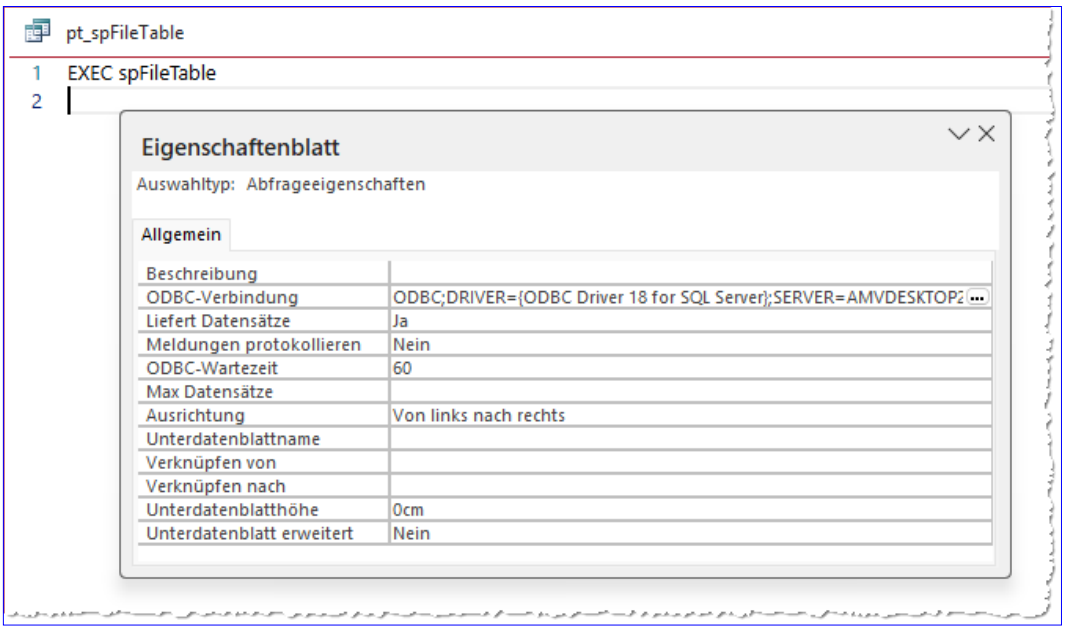


Bild 3: PassThrough-Abfrage für den Zugriff auf eine gespeicherte Prozedur

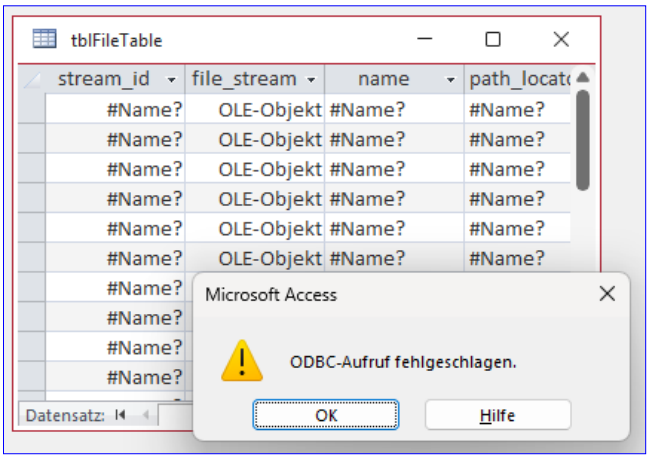


Bild 2: Versuch, die Tabellenverknüpfung mit der **FileTable**-Tabelle in der Datenblattansicht zu öffnen

```
ODBC;DRIVER={ODBC Driver 18 for SQL Server};SERVER=AM-
VDESKTOP2023\SQLEXPRESS;DATABASE=FileTableDB;Trusted_Con-
nection=Yes;TrustServerCertificate=Yes;OPTION=3;LOG_QUE-
RY=1;
```

Und siehe da: Die Pass-Through-Abfrage **pt_spFileTable** liefert alle gewünschten Daten (siehe Bild 4), sogar die aus den Feldern **path_locator** und **parent_path_locator**.

Beispiel Produktbilder

In einer reinen Access-Anwendung würde man, um zu einem Produkt eines oder mehrere Bilder anzuzeigen, den Pfad zur Bilddatei an der entsprechenden Stelle unterbringen.

Wenn es nur ein Produktbild geben soll, könnte man dieses direkt in die

Tabelle **tblProdukte** integrieren, andernfalls würde man eine Tabelle namens **tblProduktbilder** hinzufügen, in der beispielsweise ein Feld namens **Bildbeschreibung** und ein weiteres namens **Bildpfad** gespeichert werden.

Diese Tabelle, nennen wir sie **tblProduktbilder**, würde dann über ein Fremdschlüsselfeld namens **ProduktID** mit der Tabelle **tblProdukte** verknüpft werden.

pt_spFileTable

stream_id	file_stream	name	PathLocator	ParentPathL	file_type	is_directory
005056C00008}	g binary-Daten	pic001.png	/498493578199		png	0
005056C00008}		SQLServerUnd	/199818900764			-1
005056C00008}	g binary-Daten	pic002.png	/199818900764	/199818900764	png	0
005056C00008}	g binary-Daten	pic003.png	/199818900764	/199818900764	png	0
005056C00008}	g binary-Daten	pic007.png	/199818900764	/199818900764	png	0
005056C00008}	g binary-Daten	pic001.png	/199818900764	/199818900764	png	0
005056C00008}	g binary-Daten	pic006.png	/199818900764	/199818900764	png	0
005056C00008}	g binary-Daten	pic008.png	/199818900764	/199818900764	png	0
005056C00008}	g binary-Daten	pic010.png	/199818900764	/199818900764	png	0
005056C00008}	g binary-Daten	pic011.png	/199818900764	/199818900764	png	0
005056C00008}	g binary-Daten	pic013.png	/199818900764	/199818900764	png	0
005056C00008}	g binary-Daten	SQLServerUnd	/199818900764	/199818900764	indd	0
005056C00008}	g binary-Daten	pic004.png	/199818900764	/199818900764	png	0
005056C00008}	g binary-Daten	pic005.png	/199818900764	/199818900764	png	0
005056C00008}	g binary-Daten	pic009.png	/199818900764	/199818900764	png	0
005056C00008}	g binary-Daten	pic012.png	/199818900764	/199818900764	png	0

Datensatz: 2 von 16 | Kein Filter | Suchen

Bild 4: Ergebnis der PassThrough-Abfrage

Gehen wir also zunächst davon aus, dass wir eine sehr einfache Tabelle namens **tblProdukte** im SQL Server angelegt haben, deren Entwurf in Bild 5 zu sehen ist.

Außerdem aktivieren wir für das Feld **ProduktID** den Autowert über die Eigenschaft **Identitätsspezifikation**, die wir unten in den Spalteneigenschaften dieses Feldes finden.

Zwei Beispiele: Ein Bild und mehrere Bilder je Produkt

In den folgenden Abschnitten schauen wir uns gleich beide Konstellationen an:

amvDesktop20....tblProdukte | SQLQuery5.sql ...023\User (94))

Spaltenname	Datentyp	NULL-Werte...
ProduktID	int	☐
Produkt	varchar(255)	☐
Einzelpreis	money	☐
		☐

Bild 5: Entwurf der Tabelle **tblProdukte** im SQL Server

- jedes Produkt soll nur genau ein Produktbild erhalten und
- jedes Produkt soll ein, kein oder mehrere Produktbilder erhalten.

Beispiel: Ein Bild je Produkt

Als Erstes legen wir eine neue **FileTable**-Tabelle an, die **tblProduktbilder** heißen soll.

Dazu verwenden wir die folgende T-SQL-Anweisung, die zunächst die grundlegende **FileTable**-Tabelle erstellt und für diese das Verzeichnis **Produktbilder** zu dem Verzeichnis für die Dateien der **FileTable**-Tabellen des SQL Servers hinzufügt, das wir wie im Artikel **Access und SQL Server-FileTables** (**www.vbentwickler.de/489**) beschrieben angelegt haben:

```
CREATE TABLE dbo.tblProduktbilder
AS FILETABLE
WITH
(
    FILETABLE_DIRECTORY = N'Produktbilder',
    FILETABLE_COLLATE_FILENAME = database_default
);
```

Leider können wir einer **FileTable**-Tabelle keine eigenen Spalten hinzufügen, sonst hätten wir wie üblich ein Primärschlüsselfeld wie **ProduktID** und gegebenenfalls noch ein Feld wie Bildbeschreibung hinzufügen können.

Stattdessen können wir aber auch das Primärschlüsselfeld der **FileTable**-Tabelle verwenden, um die Tabelle **tblProdukte** über ein geeignetes Fremdschlüsselfeld damit zu verknüpfen.

Dazu fügen wir der Tabelle **tblProdukte** mit der folgenden T-SQL-Anweisung im SQL Server Management Studio das Feld **ProduktbildID** hinzu:

```
ALTER TABLE dbo.tblProdukte
ADD ProduktbildID UNIQUEIDENTIFIER NULL;
```

Danach führen wir die folgende Anweisung aus, um eine Beziehung zwischen den beiden Tabellen herzustellen:

```
ALTER TABLE dbo.tblProdukte
ADD CONSTRAINT FK_tblProdukte_tblProduktbilder
    FOREIGN KEY (ProduktbildID)
    REFERENCES dbo.tblProduktbilder (stream_id);
```

Das Feld **ProduktbildID** der Tabelle **tblProdukte** verweist nun auf das Primärschlüsselfeld **stream_id** der Tabelle **tblProduktbilder**.

Nun erstellen wir eine gespeicherte Prozedur, die uns die Daten der Tabelle **tblProduktbilder** in den nachfolgend beschriebenen Feldern liefert:

- **ProduktbildID**: enthält den Wert des Feldes **stream_id**
- **Bildpfad**: enthält den kompletten Pfad, der mit der SQL Server-Funktion **CONCAT(FileTableRootPath(), file_stream.GetFileNamespacePath())** ermittelt wird.

Die gespeicherte Prozedur erstellen wir mit dieser Anweisung (zum späteren Ändern muss **CREATE** durch **ALTER** ersetzt werden):

```
CREATE PROC dbo.spProduktbilder
AS
SELECT stream_id AS ProduktbildID,
    name As Bildname,
    CONCAT(FileTableRootPath(), file_stream.GetFileName-
spacePath()) As Bildpfad
FROM tblProduktbilder
WHERE file_type = 'png' AND is_directory = 0
```

Sie filtert außerdem nach dem Wert **png** für das Feld **file_type** und liefert nur Dateien zurück (**is_directory = 0**).

Nachdem wir dem Verzeichnis einige Beispielbilder hinzugefügt haben, können wir die gespeicherte Prozedur mit **EXEC spProduktbilder** ausführen und erhalten das Ergebnis aus Bild 6.

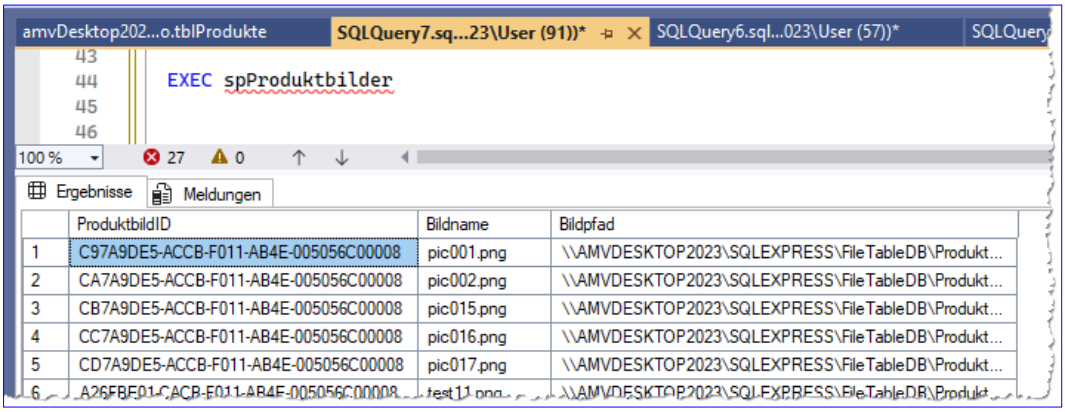
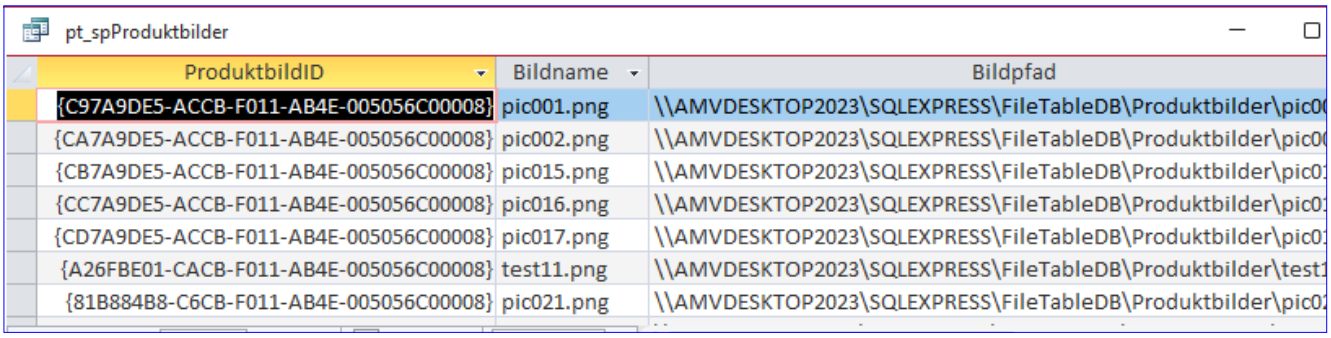


Bild 6: Gespeicherte Prozedur, die alle **.png**-Dateien liefert



pt_spProduktbilder

ProduktbildID	Bildname	Bildpfad
{C97A9DE5-ACCB-F011-AB4E-005056C00008}	pic001.png	\\AMVDESKTOP2023\SQLEXPRESS\FileTableDB\Produktbilder\pic0
{CA7A9DE5-ACCB-F011-AB4E-005056C00008}	pic002.png	\\AMVDESKTOP2023\SQLEXPRESS\FileTableDB\Produktbilder\pic0
{CB7A9DE5-ACCB-F011-AB4E-005056C00008}	pic015.png	\\AMVDESKTOP2023\SQLEXPRESS\FileTableDB\Produktbilder\pic0
{CC7A9DE5-ACCB-F011-AB4E-005056C00008}	pic016.png	\\AMVDESKTOP2023\SQLEXPRESS\FileTableDB\Produktbilder\pic0
{CD7A9DE5-ACCB-F011-AB4E-005056C00008}	pic017.png	\\AMVDESKTOP2023\SQLEXPRESS\FileTableDB\Produktbilder\pic0
{A26FBE01-CACB-F011-AB4E-005056C00008}	test11.png	\\AMVDESKTOP2023\SQLEXPRESS\FileTableDB\Produktbilder\test1
{81B884B8-C6CB-F011-AB4E-005056C00008}	pic021.png	\\AMVDESKTOP2023\SQLEXPRESS\FileTableDB\Produktbilder\pic0

Bild 7: PassThrough-Abfrage, die alle Daten der gespeicherten Prozedur **spProduktbilder** liefert

Gespeicherte Prozedur für Bilder per PassThrough-Abfrage in Access verfügbar machen

Damit wechseln wir zur Access-Anwendung. Dieser fügen wir eine Tabellenverknüpfung zur SQL Server-Tabelle **tblProdukte** hinzu.

Außerdem legen wir eine PassThrough-Abfrage namens **pt_spProduktbilder** an, mit der wir die Daten der gespeicherten Prozedur **spProduktbilder** in Access verfügbar machen wollen.

Öffnen wir diese PassThrough-Abfrage, sehen wir in der Datenblattansicht die gleichen Daten, welche die gespeicherte Prozedur im SQL Server Management Studio geliefert hat (siehe Bild 7).

Produkte und Produktbilder verwalten

Nun stellt sich die Frage, wie wir einem Produkt ein Bild hinzufügen können. Dies geschieht durch folgende Aktionen:

- Auswahl der gewünschten Bilddatei
- Hinzufügen der Bilddatei zur Tabelle **tblProduktbilder**
- Ermitteln des Primärschlüsselwertes für den neuen Datensatz der Tabelle **tblProduktbilder**
- Eintragen dieses Wertes in das Fremdschlüsselfeld **ProduktbildID** der Tabelle **tblProdukte**

Auswahl der gewünschten Bilddatei

Die Bilddatei, die wir der **FileTable**-Tabelle hinzufügen wollen, wollen wir per Dateiauswahl-Dialog ermitteln.

Dazu nutzen wir die folgende Funktion, welche die **FileDialog**-Klasse der **Office**-Bibliothek nutzt. Diese Bibliothek müssen wir zunächst über den **Verweise**-Dialog des VBA-Editors zum Projekt hinzufügen. Die Bibliothek heißt **Microsoft Office 16.0 Object Library**.

Die Funktion zum Anzeigen und Auslesen des Dateidialogs legen wir wie folgt an:

```
Public Function ChooseFolder() As String
    Dim objFileDialog As Office.FileDialog
    Dim strTemp As String
    Set objFileDialog = _
        Application.FileDialog( _
        msoFileDialogFilePicker)
    With objFileDialog
        .Title = "Datei auswählen"
        .ButtonName = "Auswählen"
        .InitialFilename = CurrentProject.Path & "\"
        .Filters.Clear
        .Filters.Add "Bilddateien", "*.png"
        If .Show = True Then
```

```
            strTemp = .SelectedItems(1)
        End If
    End With
    ChooseFolder = strTemp
End Function
```

Hinzufügen der Bilddatei zur Tabelle tblProduktbilder

Zum Hinzufügen eines Bildes zur Tabelle **tblProduktbilder** können wir zunächst die folgende T-SQL-Anweisung nutzen:

```
INSERT INTO dbo.tblProduktbilder (name, file_stream)
OUTPUT inserted.stream_id
SELECT 'pic001.png', BulkColumn
FROM OPENROWSET(
  BULK C:\pic001.png,
  SINGLE_BLOB
) AS FileData;
```

In dieser sind sowohl der Name des Bildes in der Tabelle **tblProduktbilder** als auch der Pfad, aus dem das Bild bezogen werden soll, fest verdrahtet.

In der Praxis müssen wir diese Werte jedoch variabel einfügen können. Wir müssen also eine Möglichkeit finden, wie wir dem SQL Server die notwendigen Daten übergeben.

Die erste Idee dazu ist, eine gespeicherte Prozedur anzulegen, welche die entsprechenden Parameter entgegennimmt und diesen Befehl ausführt.

Das ist aus verschiedenen Gründen problematisch, zum Beispiel weil wir den Pfad der einzufügenden Datei in dieser Anweisung nicht einfach als Parameter einfügen können (das ist eine technische Limitierung dieser speziellen Anweisung).

Wir müssten die Anweisung als SQL-String zusammenstellen und sie dann mit **sp_executesql** ausführen.

Wenn wir die Anweisung mit fest vorgegebenem Pfad in einer gespeicherten Prozedur angeben würden, könnten wir anschließend leicht den Wert des Primärschlüsselfeldes für die Tabelle **tblProduktbilder** ermitteln (das Primärschlüsselfeld in **FileTable**-Tabellen ist übrigens das Feld **path_locator**).

Dazu könnten wir die folgende Abfrage nutzen:

```
SELECT SCOPE_IDENTITY() AS ID
```

Wenn wir die Anweisung allerdings als Parameter von **sp_executesql** ausführen, findet dies in einem anderen Kontext statt als der, auf den wir mit **SELECT SCOPE_IDENTITY** zugreifen können. Wir müssten also anderweitig ermitteln, welcher Datensatz soeben angelegt wurde.

Dazu können wir zum Beispiel eine Abfrage verwenden, die den Datensatz mit einem Kriterium ermittelt, das den Inhalt des Feldes **name** des neuen Datensatzes enthält.

Und auch das ist nur eindeutig, wenn wir die **FileTable**-Tabelle nur zum einfachen Ablegen von Dateien nutzen, ohne Unterverzeichnisse zu verwenden – denn dann könnten wiederum mehrere Datensätze den gleichen Wert im Feld **name** enthalten, bei unterschiedlichem Wert im Feld **parent_path_locator**.

Wert des Feldes stream_id für den neuen Datensatz ermitteln

Vorausgesetzt, dass wir die Dateien nur im Hauptverzeichnis der **FileTable**-Tabelle ablegen, was in den meisten Fällen kein Problem ist, können wir mit diesem Wissen auch gleich eine VBA-Funktion anlegen, welche die T-SQL-Anweisung zum Hinzufügen der Datei zur **FileTable**-Tabelle hinzufügt und aufgrund des Wertes im Feld name beispielsweise den Wert des Feldes **stream_id** des neu hinzugefügten Datensatzes ermittelt.

Diesen können wir dann nutzen, um die Tabelle tblProdukte mit dem entsprechenden Datensatz der Tabelle tblProduktbilder zu verknüpfen.

Also legen wir eine Funktion an, die eine entsprechende SQL-Anweisung zusammenstellt, diese in eine temporäre PassThrough-Abfrage schreibt, ein Recordset auf Basis dieser Abfrage erstellt und damit die ID des angelegten Datensatzes ermittelt.

Diese Funktion finden wir in Listing 1. Die Funktion erwartet die beiden folgenden Parameter:

- **strBildpfad**: Pfad zu der einzufügenden Datei
- **strBildname**: Name, unter dem das Bild in der Tabelle **tblProduktbilder** gespeichert werden soll

Die Prozedur stellt zunächst in der Variablen **strConnect** die Verbindungszeichenfolge für die SQL Server-Datenbank zusammen. Diese musst Du gegebenenfalls an Deine Konstellation anpassen.

Danach stellt sie in der Variablen **strSQL** die auszuführenden Anweisungen zusammen. Wenn wir vom Wert **c:\pic001.png** für **strBildpfad** und **pic001.png** für **strBildname** ausgehen, würde der Inhalt von strSQL wie folgt aussehen:

```
INSERT INTO dbo.tblProduktbilder (name, file_stream)
OUTPUT inserted.stream_id
SELECT pic001.png, BulkColumn
FROM OPENROWSET(
  BULK C:\pic001.png,
  SINGLE_BLOB
) AS FileData;
```

Danach holt die Funktion einen Verweis auf das aktuelle **Database**-Objekt in die Variable **db** und legt mit dessen Methode **CreateQueryDef** eine neue, temporäre Abfrage in der Variablen **qdf** an.

Für dieses **QueryDef**-Objekt stellt sie dann die Eigenschaft **Connect** auf die Verbindungszeichenfolge aus **strConnect** ein. Die Einstellung **True** für die Eigenschaft **ReturnsRecords** legt fest, dass die enthaltenen Anweisungen ein Ergebnis zurückliefern sollen.

Für die Eigenschaft **SQL** übergeben wir die Anweisungen aus **strSQL**. Schließlich führen wir die Abfrage mit der **OpenRecordset**-Methode aus und referenzieren das resultierende Recordset mit der Variablen **rst**.

Wenn das Recordset vorhanden und nicht leer ist, lesen wir daraus den Wert des ersten Feldes aus, den wir im Falle des Wertes **NULL** noch mit der **Nz**-Funktion durch eine leere Zeichenkette ersetzen. Das Ergebnis speichern wir in der Variablen **strStreamID**.

Danach schließen wir das Recordset und leeren die verwendeten Objektvariablen. Die Funktion gibt den Wert aus **strStreamID** als Ergebnis zurück.

Der testweise Aufruf dieser Funktion erfolgt so:

```
Public Sub Test_ProduktbildEinfuegen_PT()
    Dim strBildpfad As String
    Dim strBildname As String
    Dim strStreamID As String

    strBildname = "pic001.png"
    strBildpfad = "C:\...\pic001.png"

    strStreamID = _
        ProduktbildEinfuegen_PT(strBildpfad, strBildname)

    Debug.Print "Neue Stream_ID: " & strStreamID
End Sub
```

Die Prozedur füllt die Parameter **strBildpfad** und **strBildname** mit den Testwerten und ruft damit die Funktion auf. Das Ergebnis wird direkt der Variablen **strStreamID** zugewiesen, deren Inhalt wir am Ende

SQL Server-Migration: Anpassen des VBA-Codes

Wenn Du Deine Datenbank zum SQL Server migriert hast, also alle Tabellen in einer SQL Server-Datenbank liegen und mit dem Frontend verknüpft sind, kannst Du ohne Probleme über die Tabellenverknüpfungen auf die Tabellen im SQL Server zugreifen. Wenn Du jedoch per VBA, speziell mit den Methoden OpenRecordset und Execute, auf diese Tabellen zugreifen möchtest, kann es zu Fehlermeldungen kommen. In diesem Artikel erläutern wir, welche Fehler dies sind, warum sie auftreten und wie Du Deine Datenzugriffe auf die Tabellen der SQL Server-Datenbank so anpasst, dass sie fehlerfrei laufen. Außerdem schauen wir uns eine Besonderheit an, die beim Anlegen neuer Datensätze mit der AddNew-Methode des Recordset-Objekts auftritt – und schließlich gibt es noch eine Besonderheit bei Formularen, die wir ebenfalls berücksichtigen.

Wenn Du die Migration zum SQL Server zum Beispiel mit dem SQL Server Migration Assistant durchgeführt hast, steht als nächster Schritt in der Regel der Test an, wie sich das Access-Frontend mit den Tabellen aus dem neuen SQL Server-Backend verträgt.

Dabei schauen wir uns in diesem Artikel primär die VBA-Seite an. Hier treten verschiedene Fehler auf, die wir uns zunächst einmal ansehen.

Fehler beim Öffnen eines Recordsets

Der erste Fehler, der auftreten kann und vermutlich auch auftreten wird, wenn Du die **OpenRecordset**-Methode des **Database**-Objekts verwendest, ist der aus Bild 1.

Die Fehlermeldung ist aussagekräftig genug, um den Fehler selbst zu beheben: Wir benötigen einen weiteren Parameter namens **dbSeeChanges**.

Diesen fügen wir wie folgt zur **OpenRecordset**-Methode hinzu:

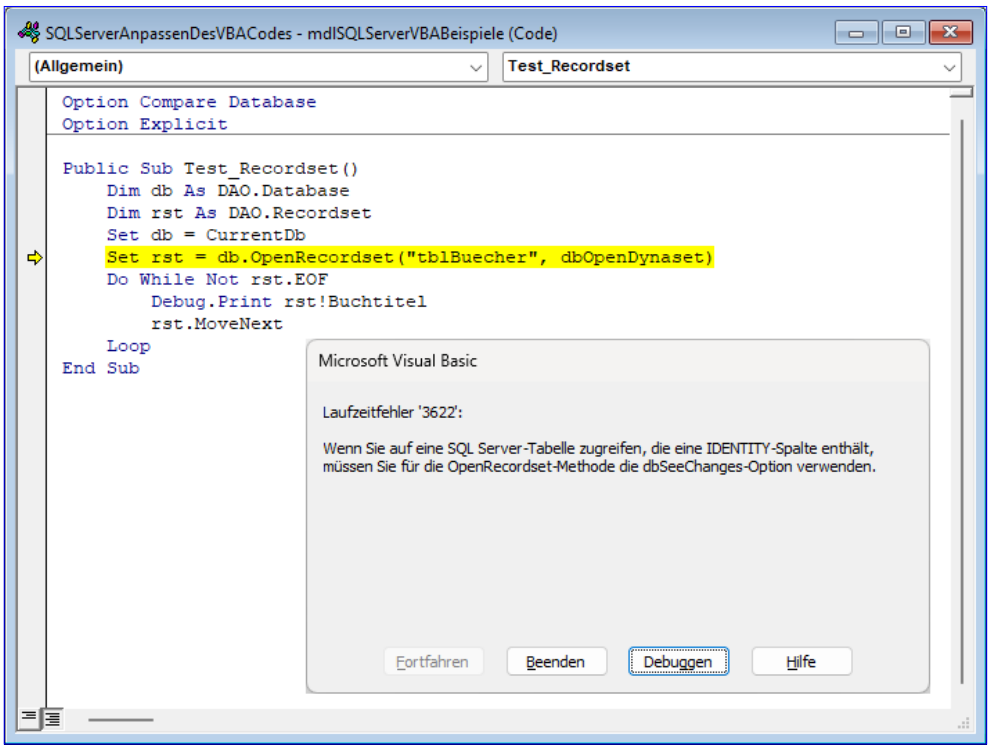


Bild 1: Fehler in der **OpenRecordset**-Methode

```
Set rst = db.OpenRecordset("tblBuecher", _
    dbOpenDynaset, dbSeeChanges)
```

Diese Konstanten müssen wir immer als dritten Parameter angeben, wenn wir mit **OpenRecordset** eine Tabelle oder Abfrage öffnen wollen.

Wenn wir bisher wie in folgendem Beispiel überhaupt keinen Parameter angegeben haben, erscheint die gleiche Fehlermeldung:

```
Set rst = db.OpenRecordset("tblBuecher")
```

Wir können dann aber nicht einfach **dbSeeChanges** wie hier als dritten Parameter setzen und den zweiten Parameter leer lassen:

```
Set rst = db.OpenRecordset("tblBuecher", , dbSeeChanges)
```

Dies würde bedeuten, dass beim **OpenRecordset** der Standardwert für den zweiten Parameter verwendet wird, der **dbOpenTable** lautet.

Und **dbOpenTable** funktioniert mit ODBC-Tabellenverknüpfungen gar nicht. Wir müssen also einen der übrigen Werte für den zweiten Parameter angeben.

Wenn wir hingegen **dbOpenSnapshot** oder **dbOpenForwardOnly** nutzen, brauchen wir **dbSeeChanges** nicht anzugeben. Diese beiden Methoden öffnen das Recordset nämlich schreibgeschützt.

Dies liefert bereits einen wichtigen Hinweis auf den Grund für die Fehlermeldung. Wenn wir diese nur erhalten, wenn wir das Recordset in dem Modus öffnen, in dem wir es bearbeiten können, muss es etwas mit dem Bearbeiten zu tun haben.

Außerdem tritt dieser Fehler nur auf, wenn die zum SQL Server migrierte Tabelle ein Autowert-Feld als Primärschlüsselfeld enthält. Dieses wird im SQL Server bei der Migration als **IDENTITY**-Spalte abgebildet, was im Grunde nichts anderes ist als ein Autowert.

Der entscheidende Unterschied ist jedoch, dass die Identitätsspalte nun vom SQL Server gefüllt und verwaltet wird und nicht mehr direkt von der Access-Datenbank aus.

Nun stellen wir uns vor, dass wir im Recordset mit **AddNew** einen neuen Datensatz anlegen. Dies sorgt dafür, dass beim Speichern zwar ein neuer Identitätswert in der SQL Server-Tabelle angelegt wird, aber dieser wird nicht automatisch an Access zurückgegeben.

Um genau dies zu gewährleisten, muss man den Parameter **dbSeeChanges** setzen. Er sorgt dafür, dass bei Anlegen neuer Datensätze das Recordset direkt aktualisiert wird und SQL Server die ID des neuen Datensatzes an Access zurückmeldet.

Zusammengefasst: **dbSeeChanges** zwingt DAO/ODBC dazu, nach einem **Insert**/**Update** die serverseitig erzeugten beziehungsweise geänderten Werte (zum Beispiel **IDENTITY**) wieder einzulesen. Ohne dieses Flag bricht DAO bei schreibenden Zugriffen auf verknüpfte SQL-Server-Tabellen mit Fehler **3622** ab.

OpenRecordset für ein QueryDef-Objekt

Das Gleiche gilt, wenn wir **OpenRecordset** als Methode eines **QueryDef**-Objekts ausführen. Im folgenden Beispiel erstellen wir ein **QueryDef**-Objekt auf Basis einer Abfrage.

Wenn wir ein Recordset für dieses **QueryDef**-Objekt öffnen wollen, müssen wir ebenfalls dbSeeChanges übergeben – dieses Mal allerdings mit dem zweiten Parameter:

```
Public Sub Test_Recordset_1()
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
```

```
    Dim rst As DAO.Recordset
    Set db = CurrentDb
    Set qdf = db.CreateQueryDef("", _
        "SELECT * FROM tblBuecher")
    Set rst = qdf.OpenRecordset(dbOpenDynaset, _
        dbSeeChanges)
    Do While Not rst.EOF
        Debug.Print rst!Buchtitel
        rst.MoveNext
    Loop
End Sub
```

Fehler beim Ausführen von Aktionsabfragen

Aktionsabfragen wie **INSERT INTO**, **UPDATE** oder **DELETE** können wir auf zwei Arten ausführen:

- **Docmd.RunSQL**
- **Execute**-Methode des **Database**-Objekts

Aktionsabfragen mit DoCmd.RunSQL

Wenn wir die **DoCmd.RunSQL**-Methode verwenden, können jede der folgenden Aktionsabfragen ohne Fehler ausführen:

```
DoCmd.RunSQL "INSERT INTO tblBuecher(BuchID, Buchtitel)
VALUES(38, 'Buch')"

DoCmd.RunSQL "UPDATE tblBuecher SET Buchtitel = 'Buch 1'
WHERE BuchID = 38"

DoCmd.RunSQL "DELETE FROM tblBuecher WHERE BuchID = 38"
```

Aktionsabfragen mit Execute

Bei Verwendung der **Execute**-Methode sieht dies anders aus. Allein das Anlegen eines Datensatzes wie folgt gelingt ohne Fehlermeldung:

```
Public Sub Test_DbExecute_INSERTINTO()
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "INSERT INTO tblBuecher(BuchID, Buchtitel) " _
        & "VALUES(37, 'Buch')", dbFailOnError
End Sub
```

Wenn wir jedoch versuchen, diesen Datensatz zu aktualisieren, erhalten wir wieder den Fehler mit der Nummer **3622**:

```
Public Sub Test_DbExecute_UPDATE()
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "UPDATE tblBuecher SET Buchtitel = 'Buch' " _
        & "WHERE BuchID = 36", dbFailOnError
End Sub
```

Und auch beim Löschen eines Datensatzes tritt dieser Fehler auf:

```
Public Sub Test_DbExecute_DELETE()
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "DELETE FROM tblBuecher WHERE BuchID = 36", _
        dbFailOnError
End Sub
```

Diesen können wir ähnlich beheben wie bei der OpenRecordset-Methode. Dieses Mal müssen wir **dbSeeChanges** jedoch nicht als weiteren Parameter anhängen, sondern wir müssen diesen mit dem vorhandenen (und immer dringend empfohlenen) **dbFailOnError** in einem Parameter vereinen. Das gelingt über den Plus-Operator (+) etwa wie folgt:

```
db.Execute "UPDATE tblBuecher SET Buchtitel = 'Buch' " _
    & "WHERE BuchID = 36", dbFailOnError + dbSeeChanges
```

Warum aber funktioniert **INSERT INTO** ohne die Angabe von **dbSeeChanges**, **UPDATE** und **DELETE** nicht? Weil **INSERT INTO** in diesem Fall direkt an den SQL Server durchgereicht wird. Für die anderen Abfra-

gen sind jedoch Änderungen an bestehenden Datensätzen nötig – und dafür müssen die Primärschlüssel wieder zwischen Access und dem SQL Server synchron gehalten werden.

Probleme mit Ja/Nein-Feldern

Ein weiteres Problem, das dieses Mal eher inhaltliche Fehler produziert statt Laufzeitfehler, sind **Ja/Nein**-Felder. Wenn wir eine Tabelle mit **Ja/Nein**-Feldern zum SQL Server übertragen, wird daraus standardmäßig der Datentyp **bit** erzeugt.

Hier gibt es zunächst einen wichtigen Unterschied zu berücksichtigen: Während in Access der Wert für **Ja/True** intern als **-1** und der Wert für **Nein/False** als **0** gespeichert wird, kann das **bit**-Feld lediglich die Werte **1** und **0** speichern.

Im SQL Server haben die Werte **1** und **0** auch keine tiefergehende Bedeutung – sie entsprechen lediglich den Zahlenwerten.

Dass Access die Werte dieser Felder korrekt als **True** oder **False** interpretiert, stellt der ODBC-Treiber sicher. Mit Tabellen, die per ODBC verknüpft sind, ändert sich auch erst einmal nichts: **1** entspricht **True** und **0** entspricht **False**.

Wenn wir also zum Beispiel in der Beispieldatenbank alle gelesenen Bücher abfragen wollen, funktioniert die folgende Abfrage auf der entsprechenden Tabellenverknüpfung korrekt:

```
SELECT * FROM tblBuecher WHERE Gelesen = True
```

Auch **False** funktioniert hier einwandfrei.

Gerade wenn wir solche Abfragen mit dem Abfrageentwurf zusammenstellen, ist sichergestellt, dass diese funktionieren, da dieser für **Ja/Nein**-Felder auch die Werte **-1** und **0** immer in **Wahr** und **Falsch** übersetzt.

Nun schauen wir uns aber ein Recordset an, das wir per VBA öffnen. Hier verwenden wir nicht den Wert **True** als Vergleichswert, sondern den Wert **-1**:

```
Public Sub Test_TrueFalse()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM tblBuecher " _
        & "WHERE Gelesen = -1", dbOpenDynaset, dbSeeChanges)
    Do While Not rst.EOF
        Debug.Print rst!BuchID, rst!Buchtitel
        rst.MoveNext
    Loop
End Sub
```

Führen wir diese Abfrage aus, erhalten wir überraschenderweise keine Daten im Direktbereich!

Auch eine **DCount**-Funktion, mit der wir die Anzahl der gelesenen Bücher prüfen wollen, liefert den Wert **0** im Direktbereich:

```
Public Sub Test_TrueFalseDLookup()
    Debug.Print DCount("BuchID", "tblBuecher", "Gelesen = -1")
End Sub
```

Damit scheint festzustehen: In Abfragen, in denen nicht durch den Abfrageentwurf der Wert **-1** durch den Wert **Wahr/True** ersetzt wird, erhalten wir nicht die korrekten Ergebnisse, weil hier tatsächlich der Access-Wert **-1** mit dem im SQL Server gespeicherten Wert **1** verglichen wird – und das liefert keine Ergebnisse.

Korrekt wäre hier also für das Recordset:

```
Set rst = db.OpenRecordset("SELECT * FROM tblBuecher " _
    & "WHERE Gelesen = 1", dbOpenDynaset, dbSeeChanges)
```

Und in der **DCount**-Anweisung müssten wir Folgendes schreiben, um das korrekte Ergebnis zu erhalten:

Codeschnipsel für SSMS per Access erstellen

Im Artikel Code-Snippets im »SQL Server Management Studio« (www.vbentwickler.de/476) haben wir die Codeschnipsel-Funktion für das SQL Server Management Studio vorgestellt und gezeigt, wie man eigene Codeschnipsel erstellen kann. Wir wollen nun eine Access-Datenbank programmieren, mit der wir die Codeschnipsel einfach verwalten und anlegen können. Dabei wollen wir das umständliche manuelle Eintragen der verschiedenen Attribute, das Definieren von Platzhaltern und das Einfügen von Platzhaltern im Codeschnipsel vereinfachen. Die Anwendung soll Codeschnipsel einfach zum Bearbeiten einlesen und diese auch wieder in das Verzeichnis der benutzerdefinierten Codeschnipsel schreiben.

Die Codeschnipsel des SQL Server Management Studios werden in einem speziellen Verzeichnis im Dateisystem gespeichert und von dort beim Start von SQL Server Management Studio geladen. Die nun vorgestellte Access-Lösung soll die einfache Verwaltung dieser Codeschnipsel ermöglichen.

Laden vorhandener Ordner und Codeschnipsel

Die erste Aufgabe ist, die vorhandenen Ordner und Codeschnipsel zu laden und in einem TreeView-Steuerelement anzuzeigen, damit wir diese anklicken und bearbeiten können.

TreeView und ImageList für die Anzeige der vorhandenen Codeschnipsel

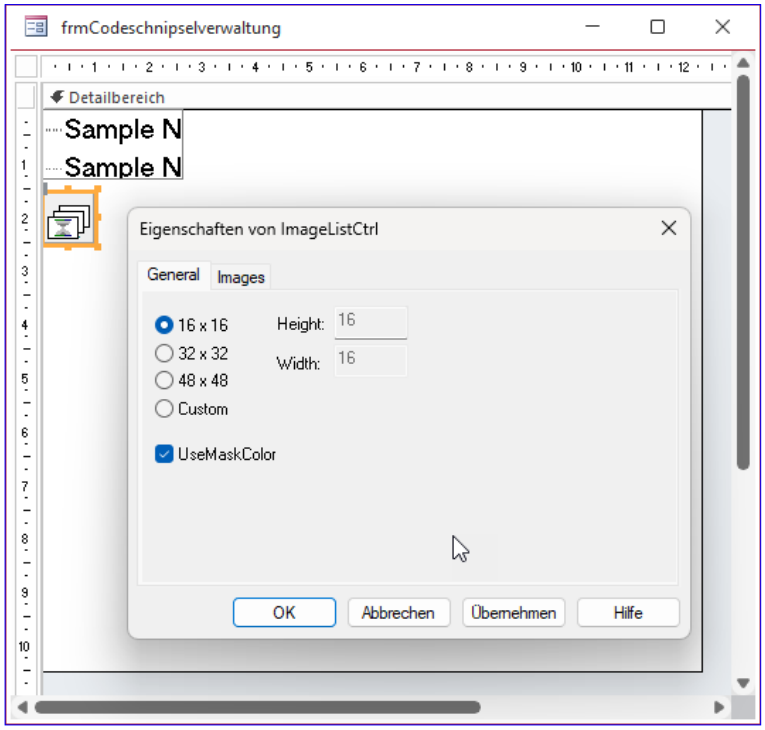


Bild 1: Einrichten des **ImageList**-Steuerelements

Dazu fügen wir ein **TreeView**-Steuerelement hinzu und benennen es in **ctlTreeView** um. Da wir im TreeView für die Ordner und die Codeschnipsel unterschiedliche Icons verwenden wollen, benötigen wir auch noch ein **ImageList**-Steuerelement, das wir **ctlImageList** nennen. Dieses klicken wir doppelt an und stellen die Größe der Icons auf 16 x 16 ein (siehe Bild 1).

Dann wechseln wir zur Registerseite **Images** und fügen zwei Icons für Ordner und Codeschnipsel hinzu.

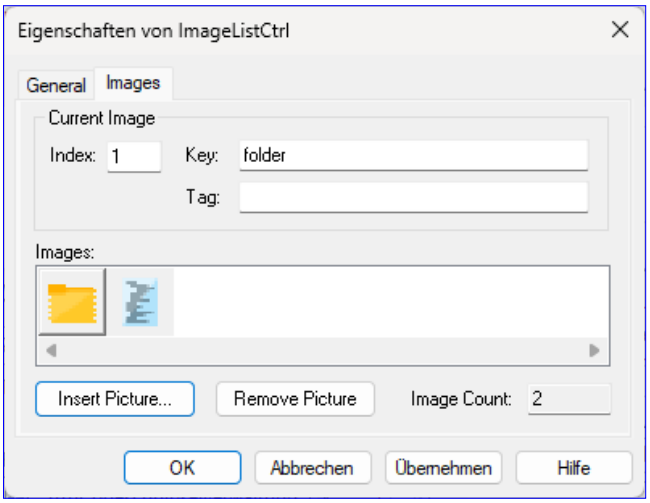


Bild 2: Hinzufügen von Icons zum **ImageList**-Steuerelement

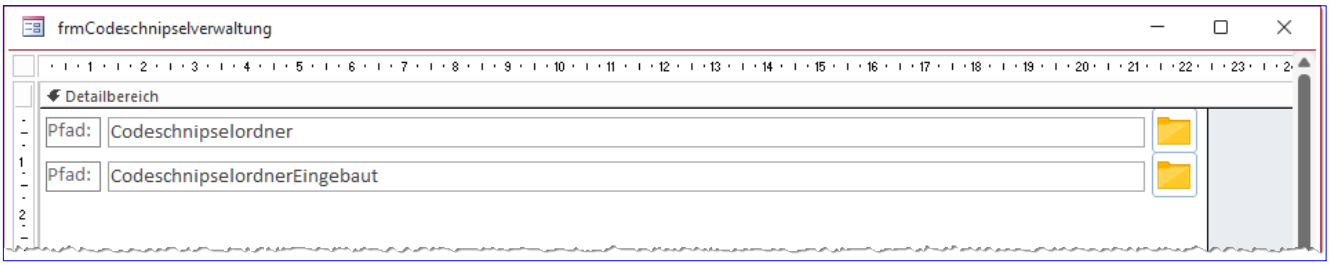


Bild 3: Aktueller Stand des Formulars mit den Elementen zum Auswählen eines Verzeichnisses

Für diese stellen wir die Eigenschaft **Key** jeweils auf **folder** und **code** ein (siehe Bild 2). Danach können wir die Eigenschaften wieder schließen.

Damit sich das **TreeView**-Steuerelement beim Vergrößern des Formulars ebenfalls vergrößert, stellen wir seine Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** auf **Beide** ein. Da das Formular keine Elemente zur Datensatznavigation anzeigen soll, stellen wir **Datensatzmarkierer** und **Navigationsschaltflächen** auf **Nein** ein. Außerdem legen wir **Bildlaufleisten** auf **Nein** fest und **Automatisch zentrieren** auf **Ja**.

Auswählen der Codeschnipsel-Verzeichnisse

Außerdem benötigen wir zwei Textfelder namens **txtOrdner** und **txtOrdnerEingebaut**, in die wir den Pfad zu den benutzerdefinierten und den eingebauten Codeschnipseln eintragen können, und Schaltflächen zum Anzeigen eines Ordnerauswahl-Dialogs. Die erste heißt **cmdOrdnerAuswaehlen** und ruft die Funktion **ChooseFolder** auf, der sie den zu verwendenden Standardordner übergibt. Nur wenn der Rückgabewert nicht leer ist, wird dieser als neuer Wert in **txtOrdner** eingetragen:

```
Private Sub cmdOrdnerAuswaehlen_Click()
    Dim strOrdner As String
    strOrdner = ChooseFolder(Nz(Me.txtOrdner))
    If Not Len(strOrdner) = 0 Then
        Me.txtOrdner = strOrdner
    End If
End Sub
```

Die zweite Schaltfläche funktioniert analog zur ersten und erlaubt die Auswahl des Ordners für die eingebauten Codeschnipsel:

```
Private Sub cmdOrdnerEingebautAuswaehlen_Click()
    Dim strOrdner As String
    strOrdner = ChooseFolder(Nz(Me.txtOrdnerEingebaut, ""))
    If Not Len(strOrdner) = 0 Then
        Me.txtOrdnerEingebaut = strOrdner
    End If
End Sub
```

Die Steuerelemente fügen wir oberhalb des **TreeView**-Steuerelements ein. Das Formular sieht nun wie in Bild 3 aus.

Funktion zum Auswählen eines Verzeichnisses

Die von den beiden Schaltflächen aufgerufene Funktion **ChooseFolder** nimmt den aktuellen Wert des jeweiligen Textfeldes mit dem Ordner entgegen und verwendet diesen als voreingestellten Ordner. Falls kein Ordner übergeben wurde, wird der aktuelle Datenbankordner als Startverzeichnis verwendet. Die Funktion hinterlegen wir im Modul **mdlFileDialog**:

```
Public Function ChooseFolder( _
        Optional strDefault As String = "") As String
    Dim objFileDialog As Office.FileDialog
    Dim strTemp As String
    Set objFileDialog = _
        Application.FileDialog(msoFileDialogFolderPicker)
    With objFileDialog
```

```
        .Title = " auswählen"
        .ButtonName = "Auswählen"
        If Len(strDefault) = 0 Then
            .InitialFileName = CurrentProject.Path & "\"
        Else
            .InitialFileName = strDefault
        End If
        If .Show = True Then
            strTemp = .SelectedItems(1)
        End If
    End With
    ChooseFolder = strTemp
End Function
```

Sie öffnet einen Dialog zur Auswahl eines Verzeichnisses. Für die hier verwendete Klasse **Office.FileDialog** benötigen wir noch einen Verweis auf die Bibliothek **Microsoft Office 16.0 Object Library**, den wir über den **Verweise**-Dialog hinzufügen.

Speichern des Codeschnipsel-Ordners

Damit wir die Codeschnipsel-Ordner nicht bei jedem Start neu auswählen müssen, speichern wir diese in einer Tabelle namens **tblOptionen**. Dieser fügen wir lediglich das Feld **Codeschnipselordner** mit dem Datentyp **Kurzer Text** hinzu (siehe Bild 4).

Anschließend stellen wir die Eigenschaft **Datensatzquelle** des Formulars auf diese Tabelle ein und setzen die Eigenschaft **Steuerelementinhalt** für das Textfeld **txtOrdner** auf **Codeschnipselordner** sowie für das Textfeld **txtOrdnerEingebaut** aus **Codeschnipselordner-Eingebaut**.

Öffnen wir das Formular nun erneut in der Formularansicht, ist das Textfeld zunächst leer. Wählen wir nun einen Ordner aus, wird dieser automatisch in der Tabelle gespeichert. Beim nächsten Öffnen des Formulars ist der Ordner nun direkt vorhanden.

TreeView-Steuerelement mit Ordnern und Elementen füllen

Nun wollen wir dafür sorgen, dass das **TreeView**-Steuerelement beim Laden des Formulars direkt mit den Elementen aus dem gewählten Verzeichnis mit den Codeschnipseln gefüllt wird.

Außerdem wollen wir einige Einstellungen vornehmen, damit die richtigen Icons für die Einträge angezeigt werden. Damit wir direkt ein paar Beispieldatensätze haben, legen wir ein paar Dummydaten im Codeschnipselordner **My Code Snippets** an.

Dieser befindet sich zum Beispiel für SQL Server Management Studio 21 in diesem Ordner:

```
C:\Users\User\Documents\SQL Server Management Studio 21\
Code Snippets\SQL\My Code Snippets
```

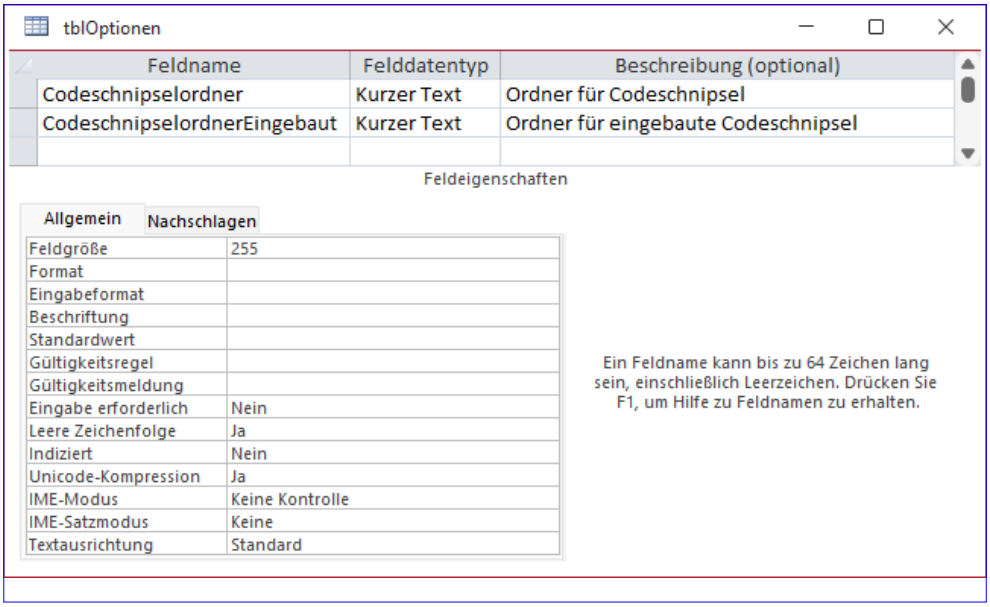


Bild 4: Tabelle zum Speichern des Codeschnipselordners

```
Public Sub CodeschnipselEinlesen(strOrdner As String, strOrdnerEingebaut As String)
    Dim objTreeView As MSComctlLib.TreeView
    Dim objRoot As MSComctlLib.node
    Dim objImageList As MSComctlLib.ImageList
    Dim objFSO As Object

    Set objTreeView = Me.ctlTreeView.Object
    Set objImageList = Me.ctlImageList.Object
    Set objTreeView.ImageList = objImageList

    objTreeView.Nodes.Clear

    Set objFSO = CreateObject("Scripting.FileSystemObject")

    If objFSO.FolderExists(strOrdner) Then
        Set objRoot = objTreeView.Nodes.Add(, , strOrdner, objFSO.GetFolder(strOrdner).Name, "folder")
        CodeschnipselEinlesen_Rek objFSO, objTreeView, objRoot, objFSO.GetFolder(strOrdner)
        objRoot.Expanded = True
        objRoot.Tag = "Rootfolder"
    Else
        MsgBox "Ordner nicht gefunden: " & strOrdner, vbExclamation
    End If

    If objFSO.FolderExists(strOrdnerEingebaut) Then
        Set objRoot = objTreeView.Nodes.Add(, , strOrdnerEingebaut, objFSO.GetFolder(strOrdnerEingebaut).Name, "folder")
        CodeschnipselEinlesen_Rek objFSO, objTreeView, objRoot, objFSO.GetFolder(strOrdnerEingebaut)
        objRoot.Expanded = True
        objRoot.Tag = "Subfolder"
    Else
        MsgBox "Ordner nicht gefunden: " & strOrdner, vbExclamation
    End If
End Sub
```

Listing 1: Funktion zum Einlesen der ersten Ebene von Ordnern und Codeschnipseln

Nun fügen wir für das Ereignis **Beim Laden** des Formulars die folgende Ereignisprozedur hinzu:

```
Private Sub Form_Load()
    Set objTreeView = Me.ctlTreeView.Object
    Call KontextmenuesErzeugen
    Call CodeschnipselEinlesen(Nz(Me.txtOrdner, ""), _
        Nz(Me.txtOrdnerEingebaut, ""))
End Sub
```

Die Objektvariable **objTreeView** deklarieren wir im allgemeinen Teil des Moduls:

```
Dim WithEvents objTreeView As MSComctlLib.TreeView
```

Danach erzeugen wir mit der Prozedur **KontextmenuesErzeugen** die Kontextmenüs für das **TreeView**-Steuerelement – mehr dazu weiter unten.

Anschließend rufen wir die Prozedur **CodeschnipselEinlesen** aus Listing 1 auf. Die Prozedur erwartet den Hauptordner als Parameter und referenziert das **ImageList**-Steuerelement mit der Variablen **objImageList**. Dann fügt es das **ImageList**-Steuerelement der Eigenschaft **ImageList** von **objTreeView** hinzu.

```
Private Sub CodeschnipselEinlesen_Rek(objFSO As Object, ctlTree As MSComctlLib.TreeView, _
        objParent As MSComctlLib.node, objFolder As Object)
    Dim objSubfolder As Object
    Dim objFile As Object
    Dim objChild As MSComctlLib.node

    For Each objSubfolder In objFolder.SubFolders
        Set objChild = ctlTree.Nodes.Add(objParent, tvwChild, objSubfolder.Path, objSubfolder.Name, "folder")
        objChild.Expanded = True
        objChild.Tag = "Subfolder"
        CodeschnipselEinlesen_Rek objFSO, ctlTree, objChild, objSubfolder
    Next objSubfolder

    For Each objFile In objFolder.Files
        If objFSO.getextensionname(objFile) = "snippet" Then
            Set objChild = ctlTree.Nodes.Add(objParent, tvwChild, objFile.Path, objFile.Name, "code")
            With objChild
                objChild.Tag = "Codesnippet"
            End With
        End If
    Next objFile
End Sub
```

Listing 2: Rekursive Funktion zum Einlesen der untergeordneten Ebenes von Ordnern und Codeschnipseln

Es leert das **TreeView**-Steuerelement und erstellt ein neues Objekt des Typs **FileSystemObject**. Dann prüft es, ob der übergebene Ordner existiert, und erstellt ein Root-Element im **TreeView**-Steuerelement, das den Namen des Ordners anzeigt, hier **My Code Snippets**.

Außerdem ruft sie für diesen Ordner die rekursiv definierte Funktion **CodeschnipselEinlesen_Rek** auf und übergibt ihr die notwendigen Informationen zum Untersuchen der im aktuellen Ordner enthaltenen Elemente (siehe Listing 2). Schließlich klappt sie den Root-Ordner auf.

Auf die gleiche Weise lesen wir die Elemente aus dem Verzeichnis für die eingebauten Codeschnipsel ein und fügen diese zum **TreeView**-Steuerelement hinzu.

Die Funktion **CodeschnipselEinlesen_Rek** erwartet einen Verweis auf das **FileSystemObject**, das **TreeView**-Steuerelement, das übergeordnete **Node**-Element und den **FileSystemObject**-Ordner als Parameter.

Sie durchläuft nun alle Unterordner über die **Subfolders**-Auflistung und legt für alle Unterordner je ein neues **Node**-Element mit dem Namen des jeweiligen Unterordners an. Auch diese Unterordner-Elemente werden durch Setzen von **Expanded** auf den Wert **True**

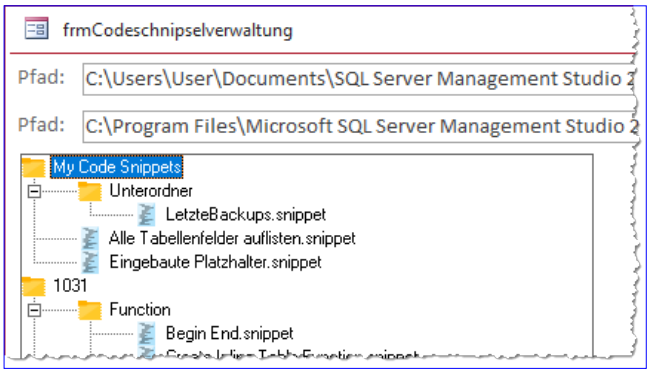


Bild 5: Die Ordner und Codeschnipsel im **TreeView**-Steuerelement

ausgeklappt. Für jeden Unterordner ruft die Funktion sich selbst erneut auf.

Nachdem alle Unterordner durchlaufen sind, verarbeitet die Prozedur in einer weiteren Schleife alle im aktuellen übergeordneten Ordner enthaltenen Dateien. Diese ermittelt sie mit der **Files**-Auflistung des **Folder**-Objekts. Hier prüfen wir noch, ob die Dateiendung der aktuellen Datei **snippet** lautet und fügen in diesem Fall ein neues Element zum übergeordneten Element hinzu.

Das Ergebnis sieht schließlich wie in Bild 5 aus.

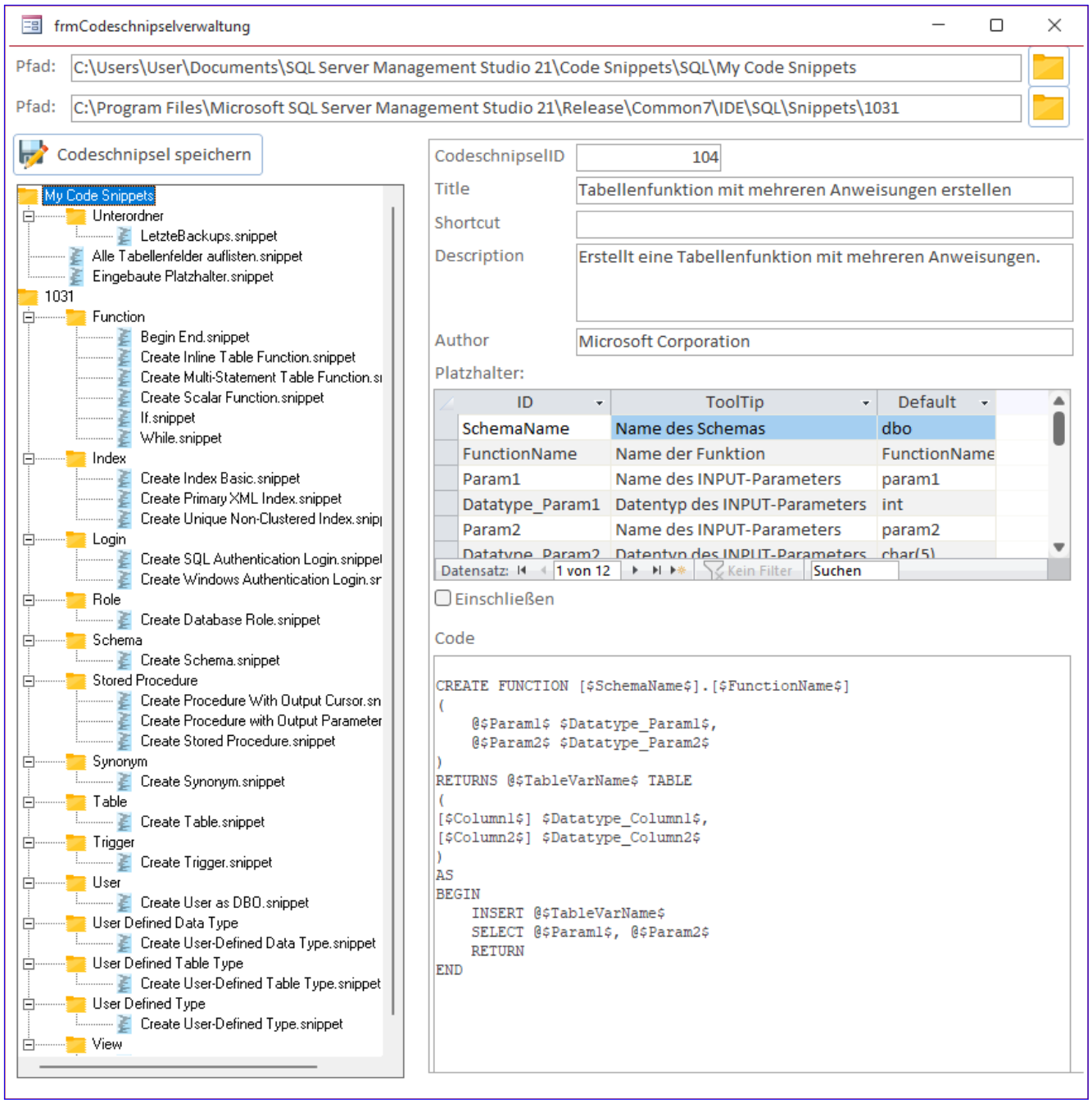


Bild 6: Details eines Codeschnipsels im Formular

Codeschnipsel markieren und Eigenschaften anzeigen

Nun fügen wir eine Prozedur hinzu, die beim Anklicken eines der Elemente des **TreeView**-Steuerelements ausgelöst wird. Diese soll prüfen, ob es sich beim angeklickten Element um ein Codeschnipsel-Element handelt, und dann aus der Eigenschaft **Key** den Pfad zur jeweiligen **.snippet**-Datei in **strPfad** schreiben.

Dann ruft sie die Prozedur **CodeschnipselAnalysieren** auf, der sie den Pfad aus **strPfad** übergibt:

```
Private Sub objTreeView_Click()
    Dim objNode As MSComctlLib.Node
    Dim strPfad As String
    Set objNode = objTreeView.SelectedItem
    If objNode.Image = "Code" Then
        strPfad = objNode.Key
        Call CodeschnipselAnalysieren(strPfad)
    End If
End Sub
```

Codeschnipsel einlesen

Die **.snippet**-Dateien sind im XML-Format geschrieben. Um diese auszulesen, benötigen wir einen Verweis auf die Bibliothek **Microsoft XML, v6.0**.

Wenn wir auf einen der Codeschnipsel geklickt haben, wollen wir seinen T-SQL-Code und die Eigenschaften in einem Unterformular anzeigen. Das Ergebnis soll wie in Bild 6 aussehen.

Die Daten des Codeschnipsels speichern wir dazu temporär in einer Tabelle namens **tblCodeschnipsel**, die wir in Bild 7 sehen.

Jeder Codeschnipsel kann außerdem Platzhalter enthalten. Diese speichern wir in der Tabelle **tblPlatzhalter**, deren Entwurf wir in Bild 8 finden.

Die Funktion **CodeschnipselAnalysieren** nimmt den Pfad zu der zu untersuchenden Codeschnipsel-Datei entgegen (siehe Listing 3). Sie referenziert mit der Variablen **db** das aktuelle **Database**-Objekt und löscht zunächst eventuell vorhandene Einträge in der Tabelle **tblCodeschnipsel**.

Da die Beziehung von der Tabelle **tblPlatzhalter** über das Fremdschlüsselfeld **CodeschnipselID** zu dieser Tabelle mit Löschweitergabe definiert ist, werden so auch alle dazugehörigen Einträge aus der Tabelle **tblPlatzhalter** gelöscht.

Danach öffnet die Funktion ein neues Recordset auf Basis der Tabelle **tblCodeschnipsel** und erstellt ein neues Objekt des Typs **DOMDocument60**. In dieses liest sie mit der **Load**-Funktion die XML-Datei mit

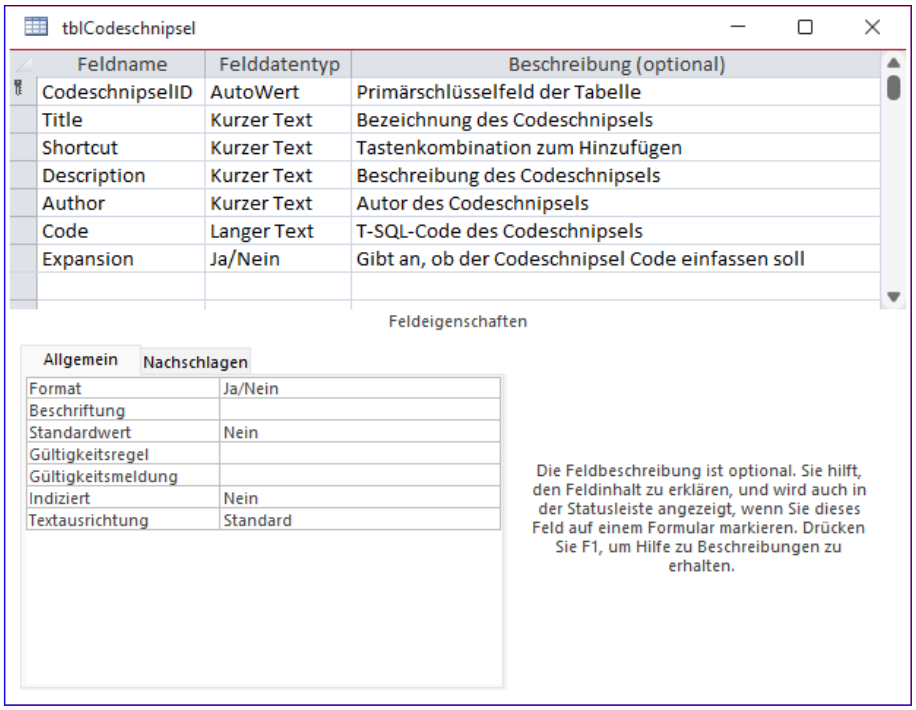


Bild 7: Tabelle zum Speichern der Daten eines Codeschnipsels