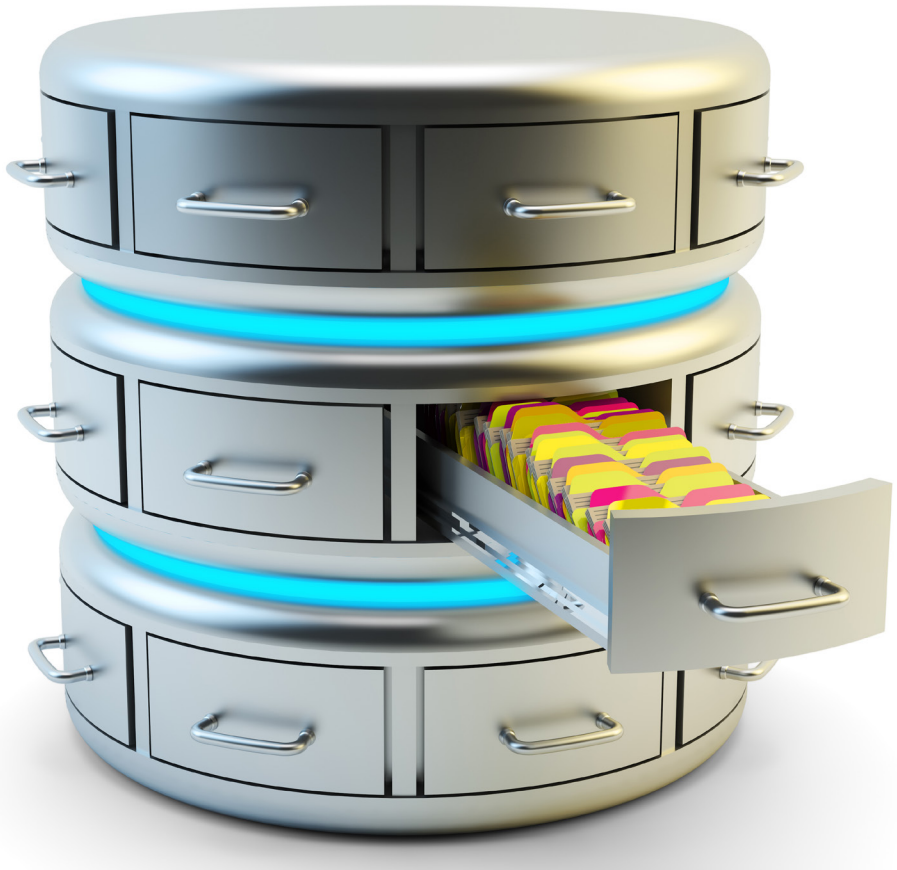
HAUFE.
Ausgabe 04/2021

ACCESS

IM UNTERNEHMEN

WEITERGABE MIT INNOSETUP

Erfahren Sie, wie Sie Frontend und Backend zuverlässig auf den Zielrechner bringen! (ab S. 57)



In diesem Heft:

API-FUNKTIONEN FINDEN

Verschaffen Sie sich einen Überblick über die API-Funktionen in Ihrer Anwendung.

SEITE 12

ODBC UND KENNWÖRTER, ABER SICHER!

Lernen Sie, wie Sie ODBC-Datenquellen mit Kennwörtern schützen können.

SEITE 39

SELEKTIONEN IM GRIFF

Speichern Sie Filterausdrücke, rufen Sie diese wieder ab und kombinieren sie auf verschiedene Arten!

SEITE 2

Weitergabe mit InnoSetup

Ein Thema wird von Microsoft recht stiefmütterlich behandelt: Die Weitergabe von Access-Datenbanken. Wer also seinem Kunden nicht nur die Datenbankdateien mit Frontend und Backend übergeben möchte, auf dass er diese selbst in den richtigen Verzeichnissen speichert, muss Eigeninitiative zeigen. Dabei gibt es seit Langem eine Alternative zu teuren Tools zum Erstellen von Setups, nämlich InnoSetup.

Deshalb schauen wir uns in einer neuen Beitragsreihe genau an, welche Aufgaben InnoSetup für uns übernehmen kann und wo die Grenzen liegen. Für diese Beitragsreihe haben wir einen Spezialisten gewonnen, nämlich Christoph Jüngling. Im ersten Teil der losen Beitragsreihe namens **Setup für Access-Anwendungen** schauen wir uns ab Seite 57 an, welche Aufgabe überhaupt zu erledigen ist und wie InnoSetup hier ins Spiel kommt.

Wenn Sie in einem Formular in der Datenblattansicht eine Auswahl festlegen, indem Sie die eingebauten Filter- und Sortiermöglichkeiten nutzen, erhalten Sie das Ergebnis postwendend. Noch schöner wäre es, wenn man einmal getroffene Selektionen speichern und wiederherstellen könnte. Genau dieses Thema behandelt unser Beitrag **Selektionen speichern** ab Seite 2. Damit legen Sie die Konfiguration per Schaltfläche in einer eigenen Tabelle ab und stellen diese per Auswahl der Konfiguration über ein Kombinationsfeld wieder her.

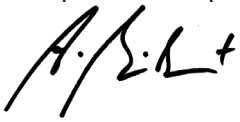
Das Ergebnis dieses Beitrags greifen wir im nächsten Beitrag mit dem Titel **Selektionen kombinieren** wieder auf (ab Seite 8). Hier zeigen wir Ihnen, wie Sie verschiedene Selektionen – beispielsweise alle Artikel der Kategorie Getränke und der Kategorie Süßigkeiten – kombinieren können. Außerdem lassen sich mit den dort vorgestellten Techniken auch die Datensätze einer Selektion von einer anderen Selektion ausschließen.

In nächster Zeit (und vielleicht auch schon jetzt) werden sich viele Entwickler um das Thema API-Funktionen kümmern müssen. Die Deklaration dieser Funktionen kopiert man in der Regel in ein Projekt und nutzt sie dann einfach. Bisher war das problemlos möglich. Jetzt aber wird Office standardmäßig in der 64-Bit-Version installiert und nicht mehr in der 32-Bit-Version. Dabei müssen die Deklarationen der API-Funktionen für die 64-Bit-Version angepasst werden. In den ersten drei Beiträgen zu diesem Thema wollen wir erst einmal herausfinden, welche API-Deklarationen sich überhaupt im VBA-Projekt einer Datenbank befinden.

Der erste Beitrag namens **API-Funktionen finden und speichern** zeigt ab Seite 12, wie Sie die Deklarationen auffinden und in einer Tabelle speichern. Im zweiten Beitrag erledigen wir die Aufgabe für Konstanten und Typen: **API-Typen und -Konstanten finden und speichern** (ab Seite 23). Schließlich untersuchen wir im Beitrag **Verwaiste API-Funktionen finden**, ob die gefundenen API-Funktionen überhaupt alle in Verwendung sind. Alle anderen können Sie prinzipiell auskommentieren und Sie brauchen diese nicht zu migrieren.

Im Beitrag **SQL Server-Security, Teil 6: ODBC-Datenquellen und gespeicherte Kennwörter** zeigt Bernd Jungbluth schließlich noch, was beim Umgang mit dem SQL Server in Bezug auf den Einsatz von ODBC-Datenquellen mit Kennwörtern zu beachten ist (ab Seite 39).

Viel Spaß beim Ausprobieren!



Ihr André Minhorst

Selektionen speichern

Mit den Möglichkeiten der Datenblattansicht können Sie bereits umfassende Filter einsetzen und Sortierungen anwenden. Dazu brauchen Sie nur die Steuerelemente zu nutzen, die bei einem Mausklick auf die Schaltfläche mit dem Pfeil nach unten im Spaltenkopf auftauchen. Aber was, wenn Sie immer wieder die gleichen Selektionen benötigen? Sollen Sie dann etwa immer wieder manuell die gleichen Einstellungen vornehmen? Nein, das wäre nicht die Idee von Access im Unternehmen. Wir zeigen Ihnen im vorliegenden Beitrag, wie Sie die Auswahl in Unterformularen in der Datenblattansichten speichern und einfach wiederherstellen können!

Beispieldatenbank zum Speichern von Selektionen

Ausgangssituation ist ein Hauptformular, das in einem Unterformular die Daten einer Tabelle oder Abfrage in der Datenblattansicht anzeigt. Um dies zu reproduzieren, legen Sie als Erstes ein neues, leeres Formular an und speichern es beispielsweise unter dem Namen **frmSelektionenSpeichern**. Lassen Sie das Formular gleich in der Entwurfsansicht geöffnet und stellen Sie die Eigenschaften **Datensatzmarkierer**, **Navigationssteuerelemente**, **Bildlaufleisten** und **Trennlinien** auf den Wert **Nein** und **Automatisch zentrieren** auf **Ja** ein.

Dann erstellen Sie ein weiteres neues Formular und speichern dieses unter dem Namen **sfmSelektionenSpeichern**. Dieses Formular soll die Daten der gewünschten Tabelle oder Abfrage in der Datenblattansicht anzeigen. Dazu tragen Sie den Namen der Tabelle oder Abfrage als Wert der Eigenschaft **Datensatzquelle** ein. Wir wollen die Tabelle **tblArtikel** unserer altbekannten Beispieldatenbank Südsturm verwenden. Anschließend ziehen Sie alle Felder der Datenquelle aus der Feldliste in den Detailbereich der Entwurfsansicht des Formulars. Eine weitere wichtige Einstellung ist die der Eigenschaft **Standardansicht** auf den Wert **Datenblatt**. Der Entwurf des Unterformulars sieht anschließend wie in Bild 1 aus. Das Unterformular können Sie nun speichern und schließen.

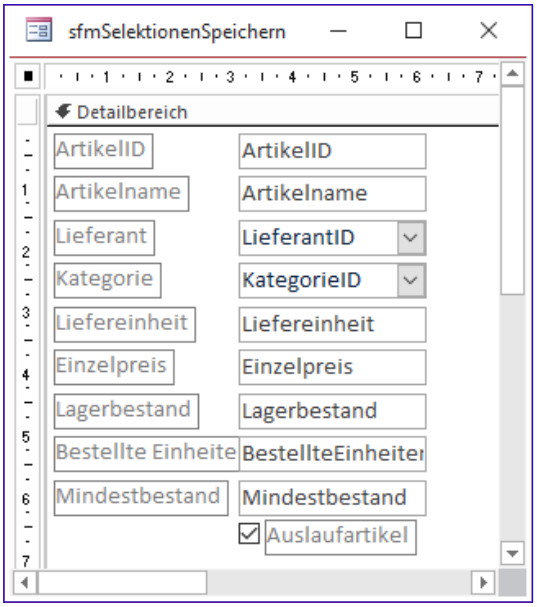


Bild 1: Entwurf des Unterformulars

Unterformular einfügen

Nun ziehen Sie das Unterformular **sfmSelektionenSpeichern** aus dem Navigationsbereich in den Detailbereich des Entwurfs des Hauptformulars und richten es etwa wie in Bild 2 aus.

Genau wie dort dargestellt legen Sie für die beiden Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** die Eigenschaft **Beide** fest.

Selektionen und Sortierungen vornehmen

Wenn Sie nun das Hauptformular in der Formularansicht anzeigen, können Sie mit den Elementen, die nach einem Klick auf die Schaltfläche mit dem nach unten zeigenden Dreieck erscheinen, verschiedene Selektionen und Sortierungen definieren.

Im Beispiel aus Bild 3 zeigen wir, wie Sie einen Textfilter anlegen. Dazu wählen Sie den Eintrag **Textfilter** des Popups mit den verschiedenen Befehlen aus und dann den gewünschten Vergleichsoperator wie beispielsweise **Gleich...** oder **Enthält...**

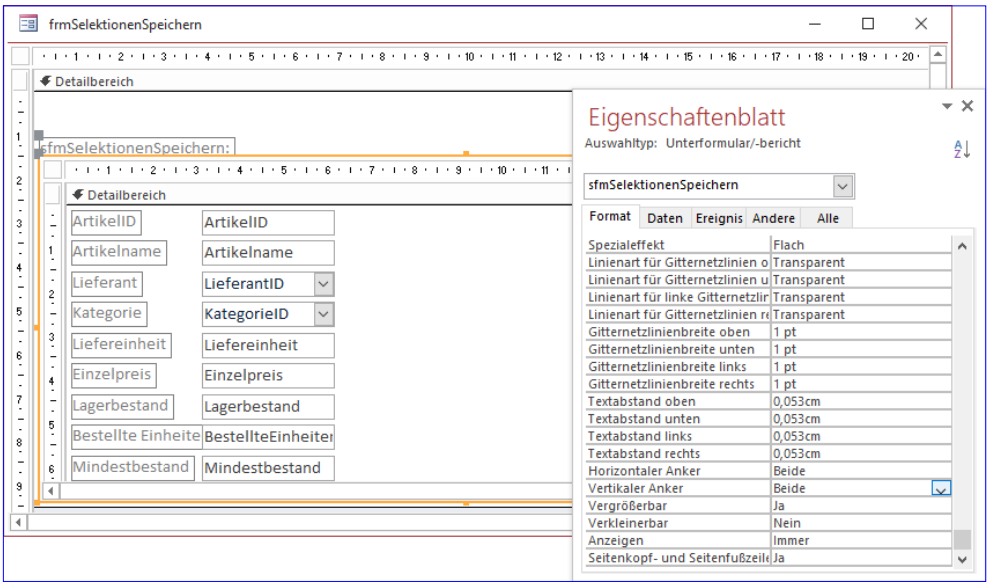


Bild 2: Einfügen des Unterformulars in das Hauptformular

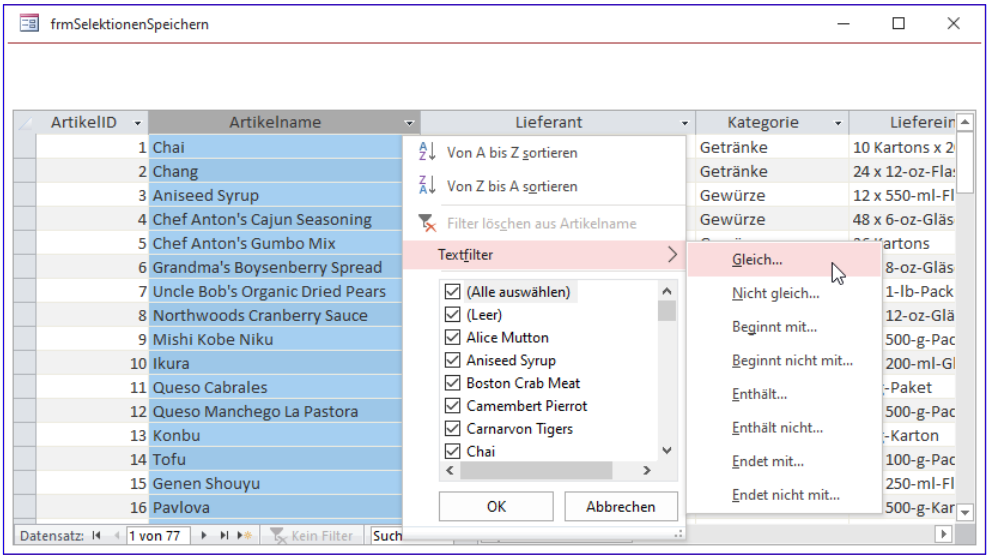


Bild 3: Einstellen von Filter- und Sortierkriterien

Filter- und Sortierausdruck ermitteln

Anschließend zeigt das Unterformular nur noch die den Kriterien entsprechenden Einträge an. Wenn Sie aber nun die gewünschte Sortierung hergestellt haben, wie können Sie diese dann speichern und wieder abrufen?

Um diese zu speichern, müssen wir erst einmal herausfinden, wie wir die Sortierung und den Filter auslesen können. Dazu stellt ein Formular die beiden Eigenschaften **OrderBy** und **Filter** zur Verfügung. Zum Ausprobieren können wir den aktuellen Wert der Eigenschaft **Filter** über den Direktbereich erfragen. Dazu geben Sie dort eine Anweisung wie die folgende ein:

```
? Forms!frmSelektionenSpeichern!sfmSelektionenSpeichern.Form.Filter
```

Im vorliegenden Beispiel haben wir das Unterformular nach allen Datensätzen gefiltert, deren Feld **Artikelname** mit **C** beginnt. Deshalb liefert die Anweisung im Direktbereich die folgende Ausgabe:

```
([tblArtikel].[Artikelname] Like "c*")
```

Wenn Sie noch eine weitere Selektion beispielsweise nach der Kategorie **Getränke** vornehmen, erhalten Sie den folgenden, schon etwas komplizierteren Filterausdruck für das Datenblatt:

```
((([tblArtikel].[Artikelname] Like "c*")))
AND ([Lookup_KategorieID].[Kategoriena-
me]="Getränke")
```

Auch Sortierungen können Sie so ermitteln. Wenn Sie gleichzeitig über die Benutzeroberfläche eine Sortierung nach dem Feld **Artikelname** definieren, finden wir die Sortierung wie folgt heraus:

```
? Forms!frmSelektionenSpeichern!sfmSelektio-
nenSpeichern.Form.OrderBy
```

Das Ergebnis lautet dann:

```
[tblArtikel].[Artikelname]
```

Eine absteigende Sortierung nach dem gleichen Feld liefert folgenden Sortierausdruck:

```
[tblArtikel].[Artikelname] DESC
```

Tabelle zum Speichern der Sortier- und Filterausdrücke

Nun benötigen wir einen Ort, um den aktuell eingestellten Filter- und Sortierausdruck zu speichern. Unter Access bietet sich dazu eine Tabelle an. Bevor wir diese gestalten, klären wir noch, welche Daten wir darin speichern wollen.

Neben jeweils einem Feld für den Filterausdruck und den Sortierausdruck benötigen wir auch noch ein Feld, mit dem wir eine Bezeichnung für diese Konfiguration festlegen. So kann der Benutzer diese später wiederherstellen.

Wenn wir noch weiterdenken, könnten wir auch noch ein Feld festlegen, in dem wir die aktuelle Datensatzquelle speichern. So könnten wir die Tabelle und den geplanten Mechanismus nicht nur in einem Formular einsetzen, sondern gleich in mehreren. Also entwerfen wir die Tabelle mit den folgenden Feldern:

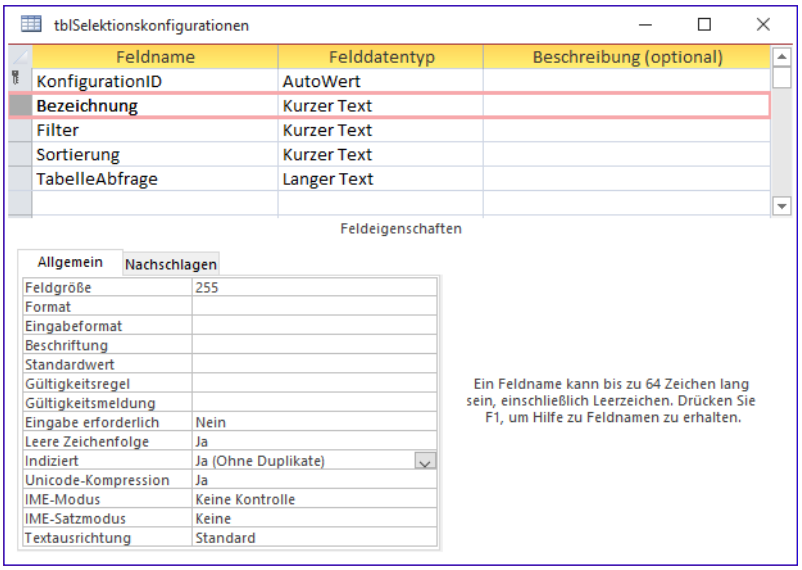


Bild 4: Einfügen des Unterformulars in das Hauptformular

- **KonfigurationID**: Primärschlüsselfeld der Tabelle
- **Bezeichnung**: Eindeutige Bezeichnung der Konfiguration
- **Filter**: Filterausdruck
- **Sortierung**: Sortierausdruck
- **TabelleAbfrage**: Name der Tabelle oder Abfrage beziehungsweise SQL-Ausdruck

Die Tabelle sieht in der Entwurfsansicht wie in Bild 4 aus. Hier erkennen Sie auch, dass wir über die Eigenschaft **Indiziert** mit dem Wert **Ja (Ohne Duplikate)** einen eindeutigen Index für das Feld **Bezeichnung** festgelegt haben.

Filter- und Sortierausdrücke speichern

Das Speichern der Filter- und Sortierausdrücke in der Tabelle **tblSelektionskonfigurationen** erfolgt mit der Prozedur aus Listing 1, die durch die Schaltfläche **cmdSelektionSpeichern** ausgelöst wird.

Die Prozedur ermittelt zunächst mit der **InputBox**-Funktion die Bezeichnung für die zu speichernde Konfiguration

Selektionen kombinieren

Bei einem Newsletter-Dienstleister habe ich neulich eine tolle Möglichkeit gesehen, um Daten zusammenzuführen. Dabei konnte man aus unterschiedlichen Verteilerlisten eine neue Verteilerliste zusammenstellen. Es war nicht nur möglich, Verteilerlisten hinzuzufügen, sondern auch das Ausschließen war eine Option. Man konnte also alle Adressen auswählen, die in Verteiler A und Verteiler B waren, aber nicht in Verteiler C. Das wollen wir mit Access auch nachbilden! Als Voraussetzung haben wir im Beitrag »Selektionen speichern« bereits die Möglichkeit zum Definieren verschiedener Filter und Sortierungen für die gleiche Tabelle geschaffen. Diese wollen wir nun als Basis zum Zusammenstellen darauf aufbauender Listen verwenden.

Das A und O der Lösung dieses Beitrags ist, Sie ahnen es bereits, eine geeignete Benutzeroberfläche. Wir können natürlich Einträge in einem Listenfeld als markiert darstellen. Die nicht markierten Einträge sollen dann nicht berücksichtigt werden. Aber wie selektieren wir dann noch diejenigen Listen, die ausgeschlossen werden sollen?

Um zu veranschaulichen, was das Ziel ist, schauen wir uns die Darstellung an, die der Newsletter-Provider uns zum Auswählen und Abwählen von Listen anbietet (siehe Bild 1). Das Auswählen ist allein über das Kästchen möglich, das Abwählen nur über den Link **Die Liste ausschließen** ganz rechts in den Einträgen.

Am praktischsten wäre ein Kontrollkästchen mit den hier verwendeten Icons, dessen drei Status man durch wiederholtes Anklicken einstellen kann – also erster Klick zum Einschließen, zweiter Klick zum Ausschließen und dritter Klick, um die Liste nicht zu berücksichtigen.

Da die meisten Nutzer das so nicht kennen, ist das Anzeigen eines Links wie hier rechts in der Auflistung wohl hilfreich.

Wenn wir allerdings in der Datenblattansicht bleiben wollen, damit der Benutzer auch die Möglichkeit hat, die Listen zu sortieren oder zu filtern, gelingt das nicht. Hier können wir nur die drei Steuerelemente Textfeld, Kontrollkästchen und Kombinationsfeld nutzen.

Wie können wir dennoch einzuschließende und auszuschließende Listen berücksichtigen? Dazu gibt es verschiedene Möglichkeiten. Die beste ist wohl eine, die optisch genau anzeigt, ob die Einträge der Liste einge-

3 ausgewählte Listen and 1 ausgeschlossene Liste

ID	Name der Liste	Ordner	Kontaktanzahl	Alles ausschließen
#35	NL_12501_13000	Access Basics	80	Die Liste ausschließen
#34	NL_12001_12500	Access Basics	500	Die Liste ausschließen
#33	NL_11501_12000	Access Basics	500	Die Liste ausschließen
~~#32~~	~~NL_11001_11501~~	~~Access Basics~~	~~500~~	Ausgeschlossene Liste(s)
#31	NL_10501_11000	Access Basics	500	Die Liste ausschließen
#30	NL_10001_10500	Access Basics	500	Die Liste ausschließen
#29	NL_9501_10000	Access Basics	500	Die Liste ausschließen

Bild 1: Aus- und Abwählen von Listen im Newslettertool

schlossen oder ausgeschlossen werden sollen (oder ob wir sie ignorieren). Warum also nicht beispielsweise die bedingte Formatierung nutzen? Allerdings benötigen wir dann noch ein weiteres Feld in der Tabelle **tblSelektionskonfigurationen**, mit dem wir den Zustand definieren. Dieses nennen wir schlicht und einfach **Selektion** und legen den Datentyp **Zahl** fest. Die Tabelle sieht im Entwurf danach wie in Bild 2 aus. Den Standardwert legen wir auf **1** fest. Dies soll der neutrale Zustand sein. **-1** soll die Liste einschließen, **0** die Liste ausschließen. Für bereits vorhandene Datensätze ohne einen Wert im Feld **Selektion** fügen Sie den Wert **1** hinzu.

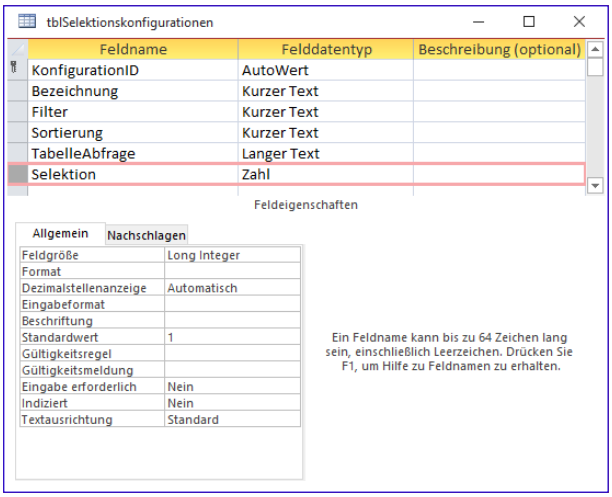


Bild 2: Erweitern der Tabelle **tblSelektionskonfigurationen**

Formular zur Auswahl der Listen

Das Formular **frmSelektionenKombinieren** enthält ein Unterformular namens **sfmSelektionenKombinieren**, das die zu selektierenden Selektionen anzeigt (siehe Bild 3).

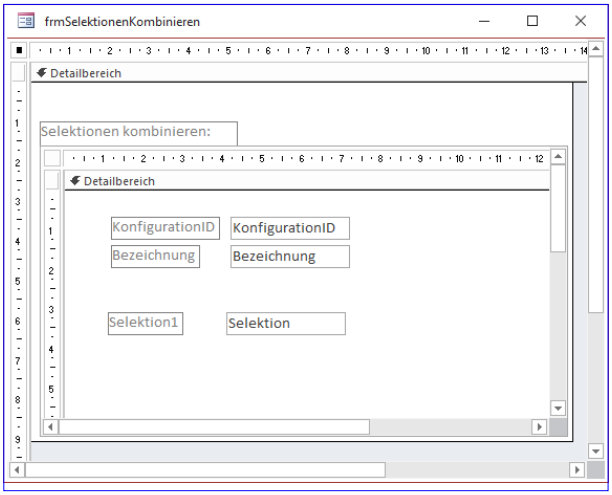


Bild 3: Entwurf des Formulars zur Anzeige der zu wählenden Selektionen

Als Datensatzquelle des Unterformulars dient die Abfrage aus Bild 4. Sie enthält alle Felder der Tabelle **tblSelektionskonfiguration** und sortiert diese nach dem Wert des Feldes **Selektion**.

Bedingte Formatierung für farbige Markierung

Damit die Datensätze je nach dem Wert des Feldes Selektion entweder grün, rot oder gar nicht markiert werden, fügen wir für jedes der Steuerelemente zwei bedingte Formatierungen hinzu. Die erste soll den Hintergrund des Feldes grün färben, wenn das Feld **Selektion** den Wert **-1** hat, der zweite färbt rot, wenn der Wert **0** ist. Die bedingten Formatierungen fügen wir der Einfachheit halber schnell mit VBA hinzu (siehe Listing 1).

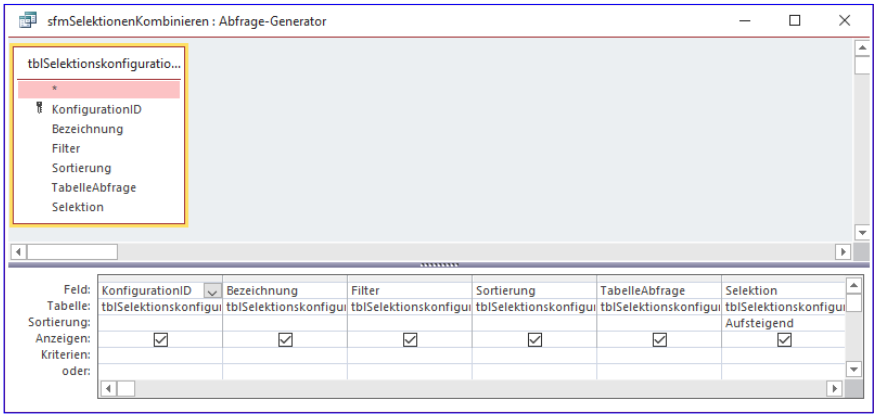


Bild 4: Abfrage für das Unterformular

Aktualisieren des Felds Selektion beim Anklicken

Wenn der Benutzer auf eines der Felder **KonfigurationID** oder Bezeichnung klickt, soll der Wert des Feldes **Selektion** den Wert wechseln – und zwar von **0** zu **-1**, von **-1** zu **1** und **1** wieder zu **0**. Da wir dazu die glei-

API-Funktionen finden und speichern

Eine umfangreiche Access-Anwendung kann in ihrem VBA-Projekt einige API-Deklarationen enthaltenen. Je länger diese Anwendung bereits entwickelt wird, desto mehr solcher Deklarationen haben sich im Laufe der Zeit in vielen verschiedenen Modulen angesammelt. Und umso mehr dieser APIs werden vielleicht gar nicht mehr verwendet, weil man sie grundsätzlich nicht mehr braucht oder sie durch andere Funktionen oder DLLs ersetzt hat. Daher ist es grundsätzlich interessant, nicht mehr verwendete Deklarationen von API-Funktionen aus der Anwendung zu entfernen. Noch interessanter wird dies, wenn die Migration einer für 32-Bit-Access ausgelegten Anwendung zu einer Anwendung ansteht, die auch unter 64-Bit-Access ihren Dienst tun soll. Je weniger API-Funktionen dann deklariert sind, umso weniger Anpassungen sind notwendig. Im vorliegenden Beitrag schauen wir uns zunächst an, wie Sie die API-Deklarationen und die gegebenenfalls benötigten Konstanten und Typen überhaupt finden.

Spätestens, wenn man Funktionen nutzen möchte, die nicht typischerweise von den eingebundenen Bibliotheken oder anderen, über den **Verweise**-Dialog hinzugefügten Bibliotheken bereitgestellt werden, benötigt man API-Funktionen.

Gängige Sätze solcher Funktionen dienen beispielsweise für folgende und viele weitere Anwendungszwecke:

- Anzeige von Dialogen zum Auswählen von Dateien und Ordnern
- Arbeiten mit Bildern
- Einsatz von FTP
- Verwenden von Verschlüsselungstechniken
- Erzeugen von GUIDs

Für manche Szenarien fügt man dazu umfangreiche Module mit vielen Deklarationen von API-Funktionen zu einem VBA-Projekt hinzu. In vielen Fällen verwendet die Anwendung jedoch nur einen Bruchteil der in den Modulen enthaltenen Befehle – zum Beispiel bei Modulen mit Funktionen für den Umgang mit Bildern per GDI.

Überblick über die API-Deklarationen verschaffen

Bevor man sich daran machen kann, die API-Funktionen in der mit 32-Bit-Access kompatiblen Fassung nach 64-Bit zu konvertieren, verschafft man sich erstmal einen Überblick über alle in einem Projekt enthaltenen APIs. Nun meinen wir damit nicht, dass Sie alle enthaltenen Module öffnen und diese manuell nach API-Deklarationen suchen. Und wir wollen auch nicht die Suchfunktion nutzen, sondern uns selbst eine Routine schreiben, mit der wir alle Deklarationen von API-Funktionen finden.

Dazu nutzen wir die Klassen, Eigenschaften und Methoden der Bibliothek **Microsoft Visual Basic for Applications 5.3 Extensibility**, die wir über den **Verweise**-Dialog (Menüpunkt **Extras|Verweise** im VBA-Editor) zum Projekt hinzufügen (siehe Bild 1).

Aktuelles VBA-Projekt identifizieren

Als Erstes benötigen wir einen Verweis auf das aktuelle VBA-Projekt. Es kann nämlich sein, dass der VBA-Editor

mehr als ein Projekt gleichzeitig im Projekt-Explorer anzeigt – zum Beispiel, wenn gerade ein Access-Add-In geöffnet ist oder wenn der Benutzer eine Bibliotheksdatenbank als Verweis eingebunden hat.

Um dieses VBA-Projekt zu ermitteln, verwenden wir die Hilfsfunktion **GetCurrentVBProject**. Es durchläuft alle Elemente der **VBProjects**-Auflistung des VBA-Editors, den wir mit der Klasse **VBE** referenzieren, in einer **For Each**-Schleife.

Dabei vergleicht die Funktion den Dateinamen des **VBProject**-Objekts mit dem der aktuell geöffneten Access-Datenbank.

Stimmen diese überein, enthält **objVBProject** einen Verweis auf das VBA-Projekt der aktuellen Datenbank-Datei und die Funktion liefert diesen als Funktionsergebnis zurück:

```
Public Function GetCurrentVBProject() As VBProject
    Dim objVBProject As VBProject
    Dim objVBComponent As VBComponent
    For Each objVBProject In VBE.VBProjects
        If (objVBProject.FileName = CurrentDb.Name) Then
            Set GetCurrentVBProject = objVBProject
            Exit Function
        End If
    Next objVBProject
End Function
```

Suche nach den API-Deklarationen

Dann beginnt der spannende Teil: die Suche nach den Deklarationen. Wie finden wir API-Deklarationen? Müssen wir dazu jede einzelne Codezeile durchsuchen – und wonach genau suchen wir dabei?

Der erste Hinweis ist: Jede Deklaration einer API-Funktion enthält das Schlüsselwort **Declare**. Dieses finden wir in keiner anderen Prozedur oder Funktion.

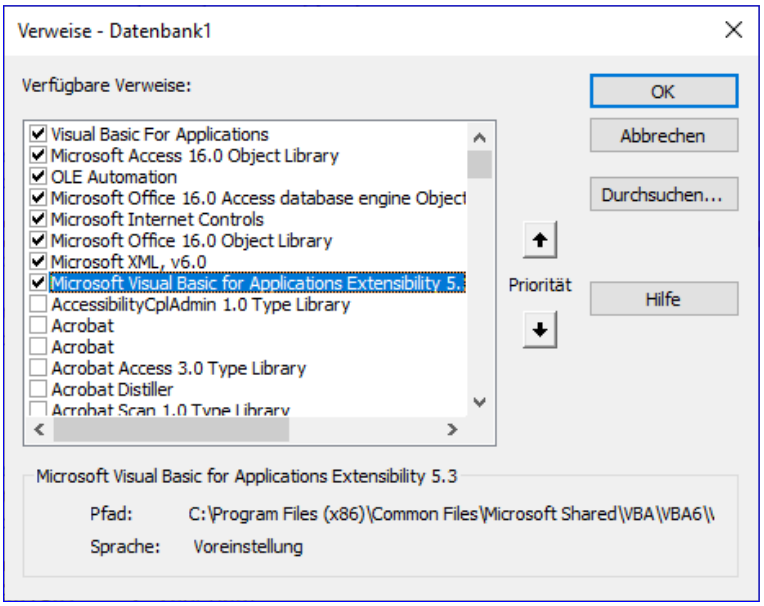


Bild 1: Verweis auf die Bibliothek **Microsoft Visual Basic for Applications 5.3 Extensibility**

Der zweite Hinweis lautet: Jedes Modul ist unterteilt in den oberen Bereich mit den Deklarationen und den unteren Bereich mit den Funktionen und Prozeduren. Vielleicht haben Sie schon einmal festgestellt, dass Sie die Deklaration einer Variable oder auch einer API-Funktion in einem VBA-Modul nicht unterhalb der ersten Routine platzieren können.

Probieren Sie dies dennoch, erhalten Sie beim Kompilieren des Projekts die Fehlermeldung aus Bild 2. Genau genommen ist der Text der Meldung nicht ganz korrekt, denn hinter **End Sub** und so weiter können natürlich auch noch weitere **Sub**-, **Function**- oder **Property**-Funktionen stehen.

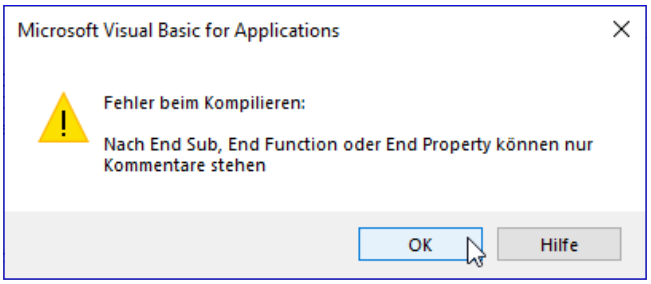


Bild 2: Fehler beim Deklarieren einer Variablen hinter einer Routine

Das ist jedoch hier irrelevant – es geht nur darum, dass jegliche Deklaration von Variablen, Kontanten und API-Funktionen immer vor der ersten **Sub**-, **Function**- oder **Property**-Prozedur stehen muss.

Alle Module durchlaufen

Wir müssen auf jeden Fall alle Module durchlaufen, um alle Deklarationen von API-Funktionen zu finden.

Das erledigen wir in einer einfachen **For Each**-Schleife über alle Elemente des Typs **VBComponent** der Auflistung **VBComponents** des aktuellen **VBProject**-Elements:

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

In diesem Fall geben wir die Namen aller Moduls im Direktbereich des VBA-Editors aus.

Das Durchsuchen der Codezeilen eines **VBComponent**-Elements umfasst einige Anweisungen, daher erstellen wir dafür direkt eine eigene Routine, die wir innerhalb der Schleife aufrufen:

```
...
For Each objVBComponent In objVBProject.VBComponents
    FindAPIDeclaresInModule objVBComponent
Next objVBComponent
...
```

Alle Deklarationszeilen untersuchen

Die Prozedur **FindAPIDeclaresInModule** nimmt den Verweis auf das **VBComponent**-Objekt entgegen und weist das enthaltene **CodeModule**-Objekt der Variablen **objCodeModule** hinzu:

```
Public Sub FindAPIDeclaresInModule(7
                        objVBComponent As VBComponent)
    Dim objCodeModule As CodeModule
    Dim lngLine As Long
    Set objCodeModule = objVBComponent.CodeModule
    For lngLine = 1 To objCodeModule.CountOfDeclarationLines
        Debug.Print objCodeModule.Lines(lngLine, 1)
    Next lngLine
End Sub
```

Für die danach folgende **For...Next**-Schleife muss man wissen, dass das **CodeModule**-Objekt eine eigene Eigenschaft bereitstellt, welche das Ende des Deklarationsbereichs des jeweiligen Moduls in Form der Zeilennummer liefert. Diese heißt **CountOfDeclarationLines**.

Wir brauchen also nur alle Zeilen von der ersten bis zu der mit **CountOfDeclarationLines** angegebenen Zeile zu durchlaufen, um alle möglichen Zeilen mit API-Deklarationen zu erwischen.

Aufgabe: Zeilenumbrüche

API-Deklarationen findet man oft in der mehrzeiligen Darstellung. Das ist übersichtlicher, wenn man beispielsweise jeden Parameter in einer neuen Zeile anzeigt:

```
Declare Function GetDiskFreeSpace Lib "kernel32" Alias
"GetDiskFreeSpaceA" ( _
    ByVal lpRootPathName As String, _
    lpSectorsPerCluster As Long, _
    lpBytesPerSector As Long, _
    lpNumberOfFreeClusters As Long, _
    lpTotalNumberOfClusters As Long) _
As Long
```

Die Untersuchung wird dadurch jedoch erschwert. Um jegliche interessante Idee von Entwicklern auszuschließen, wollen wir vor der Untersuchung einer Zeile auf eine API-Funktion zunächst die mit dem Unterstrich-Zeichen gesplitteten Zeilen in einer Variablen zusammenführen. Das gelingt in der Prozedur wie in Listing 1.

Hier durchlaufen wir nach wie vor alle Zeilen des Deklarationsbereichs, also bis zur Zeile aus **CountOfDeclarationLines**. Dabei schreiben wir zuerst den Inhalte der aktuellen Zeile in die Variable **strLine**.

In der folgenden **Do While**-Schleife prüfen wir, ob die Anweisung der aktuellen Zeile mit einem Unterstrich endet, was bedeutet, dass diese in der folgenden Zeile fortgesetzt wird. In diesem Fall nehmen wir einige Änderungen vor:

```
...
For lngLine = 1 To objCodeModule.CountOfDeclarationLines
    strLine = objCodeModule.Lines(lngLine, 1)
    Do While Not InStr(1, objCodeModule.Lines(lngLine, 1), " _") = 0
        strLine = VBA.Replace(strLine, " _", " ")
        strLine = VBA.Replace(strLine, "  ", " ")
        strLine = VBA.Replace(strLine, "  ", " ")
        strLine = VBA.Replace(strLine, "  ", " ")
        strLine = VBA.Replace(strLine, "( ", "(")
        strLine = VBA.Replace(strLine, " )", ")")
        lngLine = lngLine + 1
        strLine = strLine & objCodeModule.Lines(lngLine, 1)
    Loop
    If Not InStr(1, strLine, "'") = 0 Then
        strLine = Mid(strLine, InStr(1, strLine, "'"))
    End If
    Debug.Print strLine
Next lngLine
...
```

Listing 1: Einlesen von mehrzeiligen Anweisungen als einzeilige Anweisungen

- Wir ersetzen Unterstriche, die auf Leerzeichen folgen, durch Leerzeichen.
- Wir ersetzen doppelte Leerzeichen durch einfache Leerzeichen, um die Einrückungen folgender Zeilen zu entfernen.
- Wir ersetzen öffnende Klammern mit folgendem Leerzeichen durch öffnende Klammern und schließende Klammern mir vorangestelltem Leerzeichen durch schließende Klammern.

Schließlich erhöhen wir innerhalb des **Do While**-Schleifendurchlaufs den Wert der Zählervariablen **IngZeile** um **1**, damit die gleiche Zeile nicht beim nächsten Durchlauf der **For...Next**-Schleife erneut untersucht wird.

Außerdem fügen wir den Inhalt der aktuellen Zeile an den Inhalt von **strLine** an.

Danach prüfen wir noch, ob die Zeile noch weitere Hochkommata enthält, was auf angehängte Kommentare hinweist. Hier schneiden wir den Kommentar ab dem Hochkomma ab:

```
If Not InStr(1, strLine, "'") = 0 Then
    strLine = Mid(strLine, InStr(1, strLine, "'"))
End If
```

Das Schlüsselwort Declare finden

Als Ergebnis landen mehrzeilige Anweisungen als einzeilige Anweisung in der Variablen **strLine**, wo wir diese dann weiter untersuchen können. In diesem Fall wollen wir wissen, ob die Zeile das Schlüsselwort **Declare** enthält. Dieses Schlüsselwort kann sich allerdings an verschiedenen Positionen befinden. Die erste ist der Anfang der Zeile:

```
Declare Function CloseClipboard Lib "user32" () As Long
```

Oder es folgt auf eines der Schlüsselwörter **Private** oder **Public**:

```
Private Declare Function CloseClipboard Lib "user32" () As Long
```

```
Public Function IsDeclare(strLine As String) As Boolean
    strLine = Trim(strLine)
    If Not Left(strLine, 1) = "'" Then
        If InStr(1, strLine, "Declare") = 1 Or Not InStr(1, strLine, " Declare ") = 0 Then
            IsDeclare = True
        End If
    End If
End Function
```

Listing 2: Ermitteln, ob eine Zeile eine **Declare**-Anweisung enthält

Es könnte sich auch in einem Routinen- oder Parameternamen verstecken:

```
Public Sub FindDeclares()
```

Und zu guter Letzt kann eine **Declare**-Zeile auch auskommentiert sein:

```
'Declare Function CloseClipboard Lib "user32" () As Long
```

Wir machen also nichts verkehrt, wenn wir eine kleine Funktion programmieren, die prüft, ob es sich tatsächlich um eine **Declare**-Zeile für eine API-Funktion handelt und die wir jeweils nach dem Einlesen einer vollständigen, gegebenenfalls auch aus mehreren Zeilen bestehenden Anweisung aufrufen. Hier zunächst der Aufruf der Funktion aus der Prozedur **FindAPIDeclaresInModule** heraus:

```
For lngLine = 1 To objCodeModule.CountOfDeclarationLines
    ...
    If Not InStr(1, strLine, "Declare ") = 0 Then
        If IsDeclare(strLine) Then
            Debug.Print strLine
        End If
    End If
Next lngLine
```

Die Funktion **IsDeclare** finden Sie in Listing 2. Die Funktion entfernt mit der **Trim**-Funktion alle führenden und folgenden Leerzeichen der Zeile aus **strLine**. Dann prüft sie, ob das erste Zeichen ein Kommentarzeichen ist (**'**). Falls nicht, untersucht sie, ob die Zeile die Zeichenfolge **Declare** direkt zu Beginn aufweist oder ob die Zeichenfolge **Declare** mit führendem und folgendem Leerzeichen weiter hinten folgt.

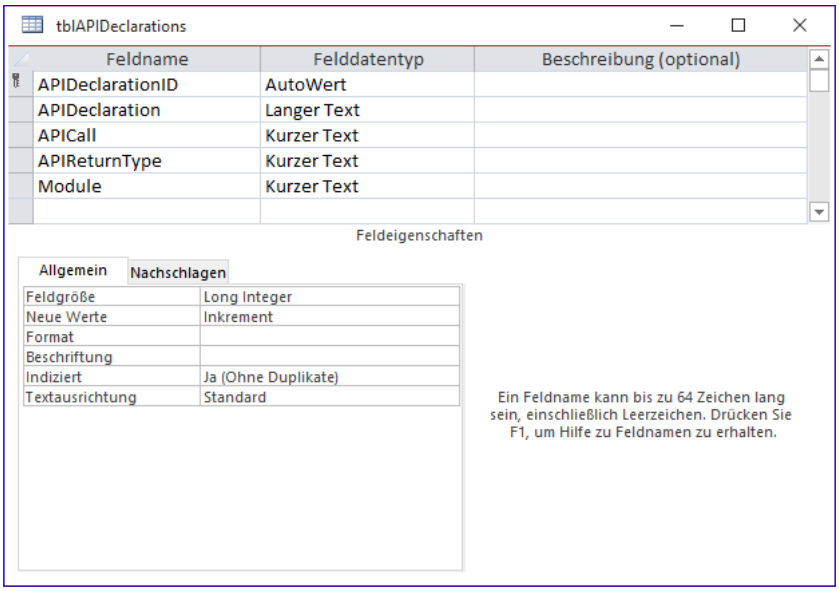


Bild 3: Entwurf der Tabelle zum Speichern der API-Deklarationen

Damit können wir innerhalb der **If IsDeclare(strLine) Then**-Bedingung per Debug.Print alle tatsächlichen API-Deklarationen ausgeben.

API-Funktionsdeklarationen in Tabelle speichern

Damit ist die Arbeit allerdings noch nicht zu Ende. Wir wollen die Deklarationen

API-Typen und -Konstanten finden und speichern

Im Beitrag »API-Deklarationen finden und speichern« haben wir gezeigt, wie Sie alle API-Deklarationen eines VBA-Projekts finden und sowohl die Daten der API-Funktion als auch die der Parameter in zwei Tabellen speichern. Zu API-Deklarationen gehören jedoch auch einige Konstanten und Typen, die beim Aufruf der API-Funktionen verwendet werden oder als Parameter der Funktionen dienen. Im Sinne des Schaffens einer Möglichkeit zum Migrieren von API-Deklarationen von 32-Bit zu 64-Bit wollen wir auch diese Elemente zunächst in entsprechenden Tabellen speichern, um diese dann per Code anzupassen und in der 64-Bit-Version auszugeben.

Vorarbeiten

Im oben genannten Beitrag **API-Deklarationen finden und speichern** (**www.access-im-unternehmen.de/****) haben wir bereits eine Prozedur angelegt, die alle Module einer Datenbank durchläuft und die enthaltenen API-Deklarationen ausliest. Dieser fügen wir einen weiteren Aufruf hinzu, und zwar für eine Prozedur namens **FindAPITypes**:

```
Public Sub FindAPIDeclares()
    Dim objVBProject As VBProject
    Dim objVBComponent As VBComponent
    Set objVBProject = GetCurrentVBProject
    For Each objVBComponent In objVBProject.VBComponents
        FindAPIDeclaresInModule objVBComponent
        FindAPITypes objVBComponent
    Next objVBComponent
End Sub
```

Die Prozedur **FindAPITypes** und die von dieser Prozedur aufgerufenen Routinen sehen wir uns im vorliegenden Beitrag an.

Wie sind Type-Konstrukte aufgebaut?

Bevor wir Code programmieren können, mit denen wir **Type**-Elemente einlesen, müssen wir uns erst einmal ansehen, wie diese Elemente aufgebaut sind. Ein **Type**-Element besteht immer mindetens aus dem **Type**-Schlüsselwort und dem Namen des Types in der ersten Zeile, einem Element, das aus dem Elementnamen, dem Schlüsselwort **As** und dem Variablentyp besteht, und dem Abschluss mit der Zeile **End Type**:

```
Type bla
    blub As String
End Type
```

Das **Type**-Element kann in der ersten Zeile auch noch ein Schlüsselwort enthalten, das die Gültigkeit bezeichnet, also **Private** oder **Public**.

Eines der im Type-Konstrukt enthaltenen Elemente mit dem Datentyp **String** kann auch noch die Angabe einer festen Zeichenanzahl enthalten, zum Beispiel mit der Zeichenanzahl **50**:

```
blub As String * 50
```

Ein praktisches Beispiel ist das folgende:

```
Private Type shellBrowseInfo
    hwndOwner      As Long
    pIDLRoot       As Long
    pszDisplayName As Long
    lpszTitle      As String
    ulFlags        As Long
    lpfnCallback   As Long
    lParam         As Long
```

```
Public Sub FindAPITypes(objVBComponent As VBComponent)
    Dim objCodeModule As CodeModule
    Dim lngLine As Long
    Dim strLine As String
    Dim strType As String
    Set objCodeModule = objVBComponent.CodeModule
    For lngLine = 1 To objCodeModule.CountOfDeclarationLines
        strLine = Trim(objCodeModule.Lines(lngLine, 1))
        strLine = ReplaceMultipleSpaces(strLine)
        If Left(strLine, 5) = "Type " Or Left(strLine, 13) = "Private Type " Or Left(strLine, 12) = "Public Type " Then
            strType = strLine & vbCrLf
            Do While Not Left(strLine, 8) = "End Type"
                lngLine = lngLine + 1
                strLine = Trim(objCodeModule.Lines(lngLine, 1))
                strLine = ReplaceMultipleSpaces(strLine)
                strType = strType & strLine & vbCrLf
            Loop
            SaveAPIType strType, objVBComponent.Name
        End If
    Next lngLine
End Sub
```

Listing 1: Prozedur zum Finden der API-Type-Strukturen

```
    iImage          As Long
End Type
```

Typen in VBA-Modulen finden

In der oben vorgestellten Prozedur namens **FindAPIDeclares** durchlaufen wir alle **VBComponent**-Objekte, also alle Module des aktuellen VBA-Projekts. Für jedes Modul rufen wir einmal die Prozedur **FindAPITypes** auf und übergeben dieser einen Verweis auf das jeweilige Modul:

```
FindAPITypes objVBComponent
```

Die Prozedur **FindAPITypes** finden Sie in Listing 1. Die Prozedur nimmt das **VBComponent**-Objekt mit dem Parameter **objVBComponent** entgegen und füllt die Variable **objCodeModule** mit dem Objekt aus der Eigenschaft **CodeModule** der VBA-Komponente aus **objVBComponent**.

Dann durchlaufen wir alle Zeilen des Deklarationsbereichs des jeweils aktuellen Moduls aus **objCodeModule** in einer **For Next**-Schleife von der Zeile **1** bis zu der mit **CountOfDeclaration** ermittelten letzten Zeile des Deklarationsbereichs. Hier übertragen wir die aktuelle Zeile, von der wir mit der **Trim**-Funktion überschüssige Leerzeichen vorn und hinten entfernen, in die Variable **strLine**. Diese schicken wir dann durch die Funktion **ReplaceMultipleSpaces**. Damit wollen wir überschüssige Leerzeichen aus der aktuellen Zeile entfernen, sodass aus der Zeile

```
Private Type       Beispiel
```

die folgende Zeile wird:

```
Private Type Beispiel
```

Die dazu verwendete Funktion **ReplaceMultipleSpaces** sieht wie folgt aus und durchläuft solange eine **Do While**-Schleife, bis keine doppelten Leerzeichen mehr in der mit dem Parameter **strLine** übergebenen Zeichenkette enthalten sind. Innerhalb der **Do While**-Schleife ersetzt

die Funktion jeweils ein doppeltes Leerzeichen durch ein einfaches Leerzeichen. Anschließend gibt sie die Zeichenkette aus **strLine** als Rückgabewert der Funktion zurück:

```
Public Function ReplaceMultipleSpaces(_
                        ByVal strLine As String) As String
    Do While Not InStr(1, strLine, "  ") = 0
        strLine = VBA.Replace(strLine, "  ", " ")
    Loop
    ReplaceMultipleSpaces = strLine
End Function
```

Die so behandelte Zeile landet in der aufrufenden Prozedur wieder in der Variablen **strLine**. Nachdem diese nun keine führenden oder folgenden Leerzeilen mehr aufweist, können wir die aktuelle Zeile dahingehend überprüfen, ob diese die erste Zeile eines **Type**-Konstrukts ist.

Dafür testen wir, ob die ersten fünf Zeichen der Zeichenkette **"Type "**, die ersten dreizehn Zeichen der Zeichenkette **"Private Type "** oder die ersten zwölf Zeichen der Zeichenkette **"Public Sub "** lauten. Ist das der Fall, fügen wir der Variablen **strType** die erste Zeile des **Type**-Konstrukts inklusive eines Zeilenumbruchs hinzu.

Danach durchlaufen wir solange eine **Do While**-Schleife, bis wir auf eine Zeile stoßen, die mit der Zeichenfolge **End Type** beginnt – also bis zur letzten Zeile des **Type**-Konstrukts. Dabei erhöhen wir die Zählervariable **lngLine** jeweils um **1** und lesen die jeweilige Zeile aus **objCodeModule.Lines(lngLine, 1)** in die Variable **strLine** ein.

Dieser entfernen wir wieder mit der Funktion **ReplaceMultipleSpaces** überflüssige Leerzeichen. Den Grund dafür erfahren Sie übrigens weiter unten. Danach fügen wir die bereinigte aktuelle Zeile als neue Zeile an die Zeichenkette **strType** an und hängen noch ein **vbCrLf** hinten an.

Dies alles geschieht, wie oben bereits beschrieben, bis wir in strLine den Ausdruck **End Type** finden. Damit ist das **Type**-Konstrukt komplett in die Variable **strType** eingelesen. Dieses übergeben wir dann nebst dem Namen der VBA-Komponente an eine weitere Prozedur namens **SaveAPIType**.

Tabelle zum Speichern der Type-Elemente anlegen

Um die **Type**-Konstrukte beziehungsweise deren Informationen zu speichern, verwenden wir zwei Tabellen. Die erste heißt **tblAPITypes** und sieht in der Entwurfsansicht wie in Bild 1 aus. Die Felder speichern die folgenden Informationen:

- **APITypeID**: Primärschlüsselfeld der Tabelle
- **APITypeName**: Name des Type-Konstrukts, also der in der ersten Zeile angegebene Bezeichner
- **Module**: Modul, in dem sich die Definition des Type-Elements befindet
- **Visibility**: Sichtbarkeit, als entweder **Private** oder **Public**

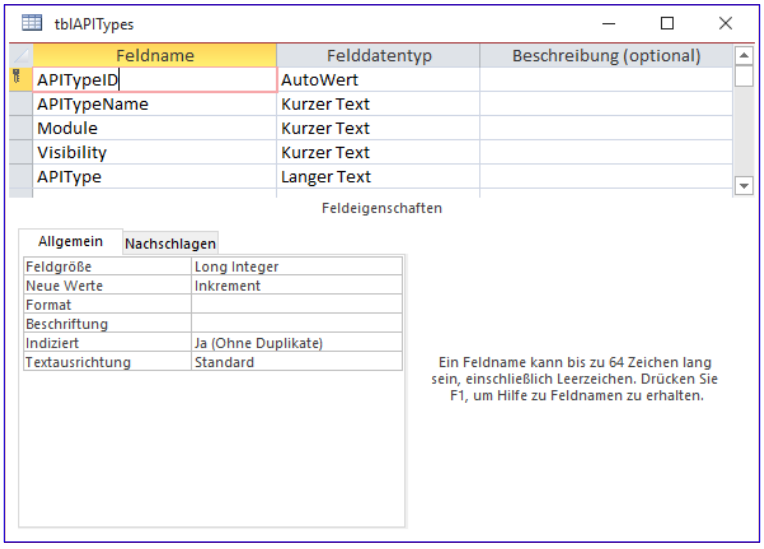


Bild 1: Entwurf der Tabelle **tblAPITypes**

```
Public Sub SaveAPIType(ByVal strAPIType As String, ByVal strModule As String)
    Dim strLines() As String
    Dim strLine As String
    Dim db As dao.Database
    Dim rstTypes As dao.Recordset
    Dim strVisibility As String
    Dim strTypeElements() As String
    Dim lngTypeID As Long
    Dim strAPITypeName As String
    Set db = CurrentDb
    Set rstTypes = db.OpenRecordset("SELECT * FROM tblAPITypes", dbOpenDynaset)
    If Right(strAPIType, 2) = vbCrLf Then
        strAPIType = Left(strAPIType, Len(strAPIType) - 2)
    End If
    strLines = Split(strAPIType, vbCrLf)
    strLine = strLines(0)
    strTypeElements = Split(strLine, " ")
    Select Case UBound(strTypeElements) - LBound(strTypeElements) + 1
        Case 2
            strVisibility = "Public"
            strAPITypeName = Split(strLine, " ")(1)
        Case 3
            strVisibility = Split(strLine, " ")(0)
            strAPITypeName = Split(strLine, " ")(2)
    End Select
    With rstTypes
        .AddNew
        !APITypeName = strAPITypeName
        !Module = strModule
        !Visibility = strVisibility
        !APIType = strAPIType
        lngTypeID = !APITypeID
        .Update
    End With
    SaveAPITypeElements db, strLines, lngTypeID
End Sub
```

Listing 2: Prozedur zum Speichern der API-Typen in der Tabelle **tblAPITypes**

- **APIType**: Text des kompletten Typs inklusive der darin definierten Elemente

Speichern des Type-Konstrukts

Das Speichern erledigt die Prozedur **SaveAPIType**. Sie erwartet mit dem ersten Parameter den kompletten Text des **Type**-Konstrukts und mit dem zweiten den Namen des Moduls, in dem sich der Type befindet (siehe Listing 2).

Gleich zu Beginn referenziert die Prozedur mit der Variablen **db** das **Database**-Objekt der aktuellen Datenbank. Damit rufen wir die Methode **OpenRecordset** auf, um eine Abfrage auf Basis der Tabelle **tblAPITypes** als Recordset zu öffnen und mit **rstTypes** zu referenzieren.

Sollte das mit **strAPIType** übergebene Type-Konstrukt noch einen Zeilenumbruch am Ende enthalten, wird dieser

zunächst entfernt. Danach füllt die Prozedur das Array **strLines** mit den Zeilen aus **strAPIType**. Dazu verwendet es die **Split**-Funktion und spaltet den Inhalt aus **strAPIType** jeweils an den Zeilenumbrüchen in einzelne Array-Elemente auf.

Die erste Zeile erfassen wir nun mit **strLines(0)** und schreiben diese in die Variable **strLine**. Nun folgt der Grund, warum wir weiter oben so darauf bedacht waren, doppelte Leerzeichen sowie führende und folgende Leerzeichen aus den Zeilen zu verbannen: wir füllen ein weiteres Array mit den einzelnen Elementen der ersten Zeile. Diese sieht vorher beispielsweise wie folgt aus:

```
Public Type Beispiel
```

Die drei Elemente landen un in einem Array namens **strTypeElements**, sodass dieses die folgenden Elemente enthält:

```
strTypeElements(0): Public
strTypeElements(1): Type
strTypeElements(2): Beispiel
```

Es könnte auch sein, dass die erste Zeile so lautet:

```
Type Beispiel
```

Dann sieht das Array so aus:

```
strTypeElements(0): Type
strTypeElements(1): Beispiel
```

Die Elemente, die wir gleich in die Tabelle **tblAPITypes** schreiben wollen, befinden sich nun also je nach **Type**-Definition an verschiedenen Positionen im Array.

Deshalb prüfen wir in der folgenden **Select Case**-Bedingung die Anzahl der Elemente des Arrays **strTypeElements**, indem wir vom Wert des größten Index (zum Beispiel **2**) den Wert des kleinsten Index (zum Beispiel **0**) subtrahieren und **1** addieren, was für **Public Type Beispiel** die Anzahl **3** ergibt.

Im Falle von **2** gehen wir davon aus, dass nur das Schlüsselwort **Type** und der Name vorhanden sind, sodass der Benutzer nicht explizit angegebenen hat, ob der Type privat oder öffentlich sein soll.

In diesem Fall wird implizit ein öffentlicher Type verwendet. Also schreiben wir in die Variable **Visibility** den Wert **Public**. In die Variable **strAPITypeName** tragen wir dann das zweite Element des Arrays ein, beispielsweise den Namen **Beispiel**.

Wenn die **Select Case**-Bedingung den Zweig **Case 3** ansteuert, wir es also mit einer ersten Zeile wie **Public Type Beispiel** zu tun haben, verwenden wir den ersten Wert des Arrays für die Variable **strVisibility** und den dritten als **strAPITypeName**.

Damit ausgestattet fügen wir schließlich einen neuen Datensatz zum Recordset **rstTypes** für die Tabelle **tblAPITypes** hinzu und legen die Werte der Felder **APITypeName** (aus **strAPITypeName**), Module (aus dem Parameter **strModule**), **Visibility** (aus **strVisibility**) und **APIType** (aus dem Parameter **strAPIType**) fest. Bevor wir den Datensatz mit der **Update**-Methode speichern, lesen wir noch den Primärschlüsselwert des neuen Datensatzes in die Variable **lngTypeID** ein.

Diesen übergeben wir dann samt der Variablen **db** und den in **strLines** gespeicherten einzelnen Zeilen des Types an die Prozedur **SaveAPITypeElements** weiter, welche die Informationen über die einzelnen Typen in der Tabelle **tblAPITypesElements** speichern soll.

Tabelle zum Speichern der Elemente des Type-Konstrukts

Die Tabelle zum Speichern der Informationen der einzelnen Elemente des **Type**-Konstrukts heißt **tblAPITypesElements** und sieht in der Entwurfsansicht wie in Bild 2 aus.

Verwaiste API-Funktionen finden

Wenn Sie eine größere Anwendung pflegen müssen, kann es hin und wieder sinnvoll sein, nicht mehr verwendete Routinen rauszuwerfen oder zumindest auszukommentieren. Das ist gerade praktisch, wenn Sie die API-Funktionen von 32-Bit auf 64-Bit umstellen wollen: Um hier Arbeit zu sparen, können Sie erst einmal alle API-Funktionen auskommentieren, die nicht mehr benötigt werden. Um verwaisten Routinen zu finden, gibt es verschiedene Möglichkeiten: Sie können dies durch Auskommentieren und Testen von Hand erledigen, ein Tool wie MZ-Tools einsetzen oder auch ein selbst programmiertes Tool. Letzteres könnte der entsprechenden Funktion von MZ-Tools noch einen draufsetzen und auch die Aufrufe aus den Eigenschaften von Formularsteuerelementen oder Abfragen ermitteln.

Hintergrund: 32-Bit- vs. 64-Bit-Access

Mit Office 2019 hat Microsoft erstmalig die 64-Bit-Variante des Office-Pakets als Standard bei der Installation festgelegt. Zuvor war das noch die 32-Bit-Version. Wer also nicht gezielt die 64-Bit-Version installieren wollte, erhielt dort noch die 32-Bit-Version.

Das führte dazu, dass hier und da das Problem auftauchte, dass die für 32-Bit-Anwendungen ausgelegten API-Funktionen unter VBA nicht mehr funktionierten. Sie mussten an einigen Stellen angepasst werden, damit sie mit der 64-Bit-Version kompatibel waren.

Zu dieser Zeit konnte man Kunden in einigen Fällen noch zur Neuinstallation von Access in der 32-Bit-Version bewegen – auch vor dem Hintergrund, dass auch wichtige Steuerelemente wie das **TreeView**-Steuerelement und andere Elemente der Bibliothek **MSCOMCTL.ocx** nur unter 32-Bit zur Verfügung standen. Nunmehr ist auch das kein Grund mehr, nicht die 64-Bit-Version zu verwenden, denn die **MSCOMCTL.ocx**-Bibliothek steht nun auch dort zur Verfügung.

Also steht bei vielen Entwicklern nun auch die Migration von 32-Bit zu 64-Bit auf dem Programm. In den beiden Beiträgen **API-Funktionen finden und speichern** (**www.access-im-unternehmen.de/****) und **API-Typen und -Konstanten finden und speichern** (**www.access-im-unternehmen.de/****) haben wir bereits einige Vorbereitungen getroffen und Routinen programmiert, die alle API-Funktionen eines VBA-Projekts auslesen und in Tabellen schreiben – das Gleiche für die Konstanten und Deklarationen eines VBA-Projekts.

Nun benötigen wir noch ein Werkzeug, mit dem wir entscheiden können, welche dieser Elemente überhaupt in der Anwendung benötigt werden – und welche wir gegebenenfalls einfach auskommentieren können und nicht mehr anzupassen brauchen. Genau das erledigen wir in diesem Beitrag.

Woher kommen überzählige API-Funktionen?

Eine Frage, die sich hier stellt, ist folgende: Woher kommen denn eigentlich überzählige API-Funktionen? Sollte man nicht annehmen, dass der Programmierer nur die unbedingt benötigten API-Funktionen zu seinem VBA-Projekt hinzufügt? Und das überzählige API-Funktionen nur dann anfallen, wenn man sich im jeweiligen Fall für eine alternative Technik entscheidet oder die Anforderung einfach wegfällt und man dann vergisst, die API-Funktion zu entfernen?

In der Tat gibt es meist komplette Module mit den Deklarationen von Typen, Konstanten und Variablen samt API-

Funktionen, die einen zusammenhängenden Bereich abdecken – beispielsweise die Verschlüsselung oder Grafikfunktionen. Wenn man eine benötigte API-Funktion in einem solchen Modul vorfindet, fügt man es in der Regel auch komplett zum VBA-Projekt hinzu – einfach, weil es zu aufwendig erscheint, nur die benötigten Elemente zu verwenden. Wenn Sie jedoch ein komplettes Modul mit GDI-Funktoinen zum Projekt hinzufügen und nur eine Funktion daraus nutzen, sollten Sie spätestens bei einer drohenden Migration von 32-Bit nach 64-Bit überlegen, den Umfang auf die benötigten Elemente zu reduzieren.

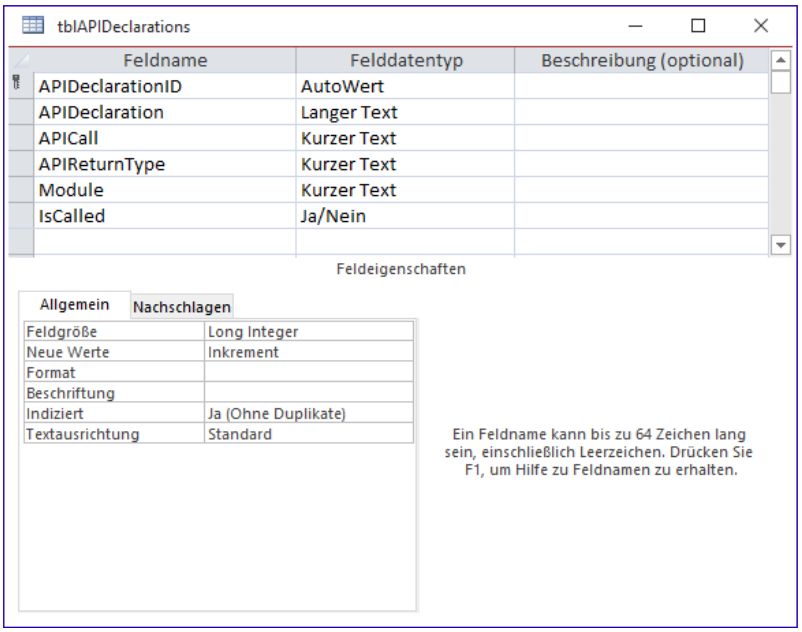


Bild 1: Tabelle zum Speichern der API-Deklarationen eines VBA-Projekts

Basis: Funktionsname

Wir wollen in diesem Beitrag davon ausgehen, dass eine mehr oder weniger große Menge von API-Funktionen vorliegt, die wir entweder aus der Tabelle entnehmen, die wir im Beitrag **API-Funktionen finden und speichern** erstellt und gefüllt haben oder die wir einfach so als Parameter an die Funktion zum Untersuchen von Aufrufen dieser Funktion übergeben.

Damit ist auch schon das Grundgerüst definiert: Wir benötigen eine VBA-Funktion, der wir den Namen einer API-Funktion übergeben und die das komplette VBA-Projekt nach Aufrufen dieser Funktion durchsucht.

Wie aber finden wir solche Aufrufe? Dazu suchen wir nach Zeilen, die folgende Bedingungen erfüllen:

- Die Zeile enthält den Namen der API-Funktion, gefolgt von einer öffnenden Klammer – also zum Beispiel **Sleep(** –, gefolgt von einem Leerzeichen oder gefolgt von einem Zeilenumbruch.

- Die Zeile enthält nicht die Deklaration der API-Funktion selbst. Diese ist leicht zu definieren, nämlich durch das Schlüsselwort **Declare**.

- Die Zeile ist keine Kommentarzeile.

Ein API-Aufruf kann nämlich beispielsweise wie folgt aussehen:

```
Sleep 30 'mit Leerzeichen
lngDriveType = GetDriveType("c:") 'mit öffnender Klammer
EmptyClipboard 'mit folgendem Zeilenumbruch
```

Speichern der Ergebnisse in einer Tabelle

Im Beitrag **API-Funktionen finden und speichern** haben wir bereits eine Tabelle namens **tblAPIDeclarations** vorgestellt, in die wir mir einer ebenfalls in diesem Beitrag vorgestellten Routine alle Deklarationen von API-Funktionen eintragen.

Diese Tabellen wollen wir für den vorliegenden Beitrag um ein Feld erweitern. Dieses soll den Namen **IsCalled** erhalten.

Sobald wir den Aufruf einer der in der Tabelle gespeicherten API-Funktionen gefunden haben, stellen wir den Wert dieses Feldes auf den Wert **True** ein (siehe Bild 1).

```
Public Sub FindAPICalls(objVBproject As VBProject)
    Dim objVBComponent As VBComponent, objCodeModule As CodeModule
    Dim db As DAO.Database, rst As DAO.Recordset
    Dim lngStartLine As Long, lngEndLine As Long, lngStartColumn As Long, lngEndColumn As Long
    Dim strLine As String
    Dim bolApiCall As Boolean, bolFound As Boolean
    Set db = CurrentDb
    db.Execute "UPDATE tblAPIDeclarations SET IsCalled = False", dbFailOnError
    Set rst = db.OpenRecordset("SELECT * FROM tblAPIDeclarations", dbOpenDynaset)
    Do While Not rst.EOF
        Debug.Print rst!ApiCall
        For Each objVBComponent In objVBproject.VBComponents
            Set objCodeModule = objVBComponent.CodeModule
            lngStartLine = 0
            lngStartColumn = 0
            lngEndLine = 0
            lngEndColumn = 0
            bolFound = objCodeModule.Find(rst!ApiCall & "(", lngStartLine, lngStartColumn, lngEndLine, lngEndColumn) _
                Or objCodeModule.Find(rst!ApiCall & " ", lngStartLine, lngStartColumn, lngEndLine, lngEndColumn) _
                Or objCodeModule.Find(rst!ApiCall & vbCrLf, lngStartLine, lngStartColumn, lngEndLine, lngEndColumn)
            Do While bolFound = True
                bolApiCall = True
                strLine = objCodeModule.Lines(lngStartLine, 1)
                strLine = Trim(strLine)
                If Not InStr(1, strLine, "'") = 0 Then
                    If InStr(1, strLine, "'") < InStr(1, strLine, rst!ApiCall) Then
                        bolApiCall = False
                    End If
                End If
                If Not InStr(1, strLine, "Declare ") = 0 Then
                    bolApiCall = False
                End If
                If bolApiCall = True Then
                    rst.Edit
                    rst!IsCalled = True
                    rst.Update
                    Exit For
                End If
                lngStartLine = lngStartLine + 1
                lngEndLine = 0
                bolFound = objCodeModule.Find(rst!ApiCall & "(", lngStartLine, lngStartColumn, lngEndLine, lngEndColumn) _
                    Or objCodeModule.Find(rst!ApiCall & " ", lngStartLine, lngStartColumn, lngEndLine, lngEndColumn) _
                    Or objCodeModule.Find(rst!ApiCall & vbCrLf, lngStartLine, lngStartColumn, lngEndLine, lngEndColumn)
            Loop
        Next objVBComponent
        rst.MoveNext
    Loop
End Sub
```

Listing 1: Prozedur zum Ermitteln, ob eine API-Funktion verwendet wird

SQL Server-Security, Teil 6: ODBC-Datenquellen und gespeicherte Kennwörter

Bernd Jungbluth, Horn – www.berndjungbluth.de

Die Verbindung von Access zum SQL Server erfolgt in der Regel über ODBC. Hierzu wird vorab eine ODBC-Datenquelle erstellt und unter einem Data Source Name – kurz DSN – gespeichert. Für den Datenzugriff liefert die ODBC-Datenquelle die Bezeichnung des SQL Servers und den Namen der Datenbank. Der Anmeldename und das Kennwort hingegen kommen direkt aus der Access-Applikation. Dabei sind die in Access gespeicherten Anmeldedaten ein nicht zu unterschätzendes Sicherheitsrisiko. Dieser Beitrag beschreibt die Risiken und zeigt Ihnen Mittel und Wege, wie Sie diese vermeiden können.

Warnung

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

ODBC-Datenquellen

Die Definition einer ODBC-Datenquelle für den Zugriff auf eine SQL Server-Datenbank besteht im Wesentlichen aus den Angaben zum SQL Server, zur Datenbank und den Anmeldedaten. Bei Bedarf lassen sich noch weitere Optionen ergänzen, zum Beispiel zur Verschlüsselung der Datenübertragung. Der ODBC-Datenquelle geben Sie im ODBC-Datenquellen-Editor einen Namen, den sogenannten Data Source Name oder kurz DSN. Mit diesem DSN binden Sie die Tabellen der SQL Server-Datenbank in die Access-Applikation ein und definieren die ODBC-Verbindung der Pass Through-Abfragen.

In der Beispielumgebung verwendet die ODBC-Datenquelle den DSN **WaWi_SQL**. Sind Sie der Installationsanleitung zur Beispielumgebung gefolgt, enthält der DSN die Anmeldung **WaWiPersonal**. Sollte dies nicht der Fall sein, passen Sie die Anmeldedaten in der ODBC-Datenquelle an. Nur mit dieser Anmeldung haben Sie Zugriff auf alle Tabellen und gespeicherten Prozeduren. Die Installationsanleitung zur Beispielumgebung finden Sie am Ende des Beitrags.

Der Data Source Name

Die ODBC-Datenquelle mit dem DSN **WaWi_SQL** ermöglicht Ihnen den Zugriff auf die Daten der gleichnamigen SQL Server-Datenbank. Dabei ist es Ihnen überlassen, mit welcher Applikation Sie auf die Daten zugreifen. So können Sie die Daten unter anderem in Excel auswerten oder in einer eigenen Access-Datenbank verarbeiten. Der Vorgang für den Datenzugriff ist immer der gleiche. Sie wählen den DSN der ODBC-Datenquelle und geben das Kennwort zur Anmeldung ein. Beides ist in der Access-Applikation **WaWi** nicht mehr notwendig. Hier haben Sie bereits beim Einbinden der Tabellen den DSN **WaWi_SQL** gewählt sowie das Kennwort eingegeben und es in Access gespeichert. Dadurch gehört der DSN mitsamt dem Kennwort zu den Eigenschaften einer eingebundenen Tabelle. Diese können Sie sich in der Entwurfsansicht anschauen.

Starten Sie die Access-Applikation **WaWi** und öffnen Sie die Tabelle **Artikel** in der Entwurfsansicht. Nach Bestätigen der Meldung mit einem Klick auf **Ja** blenden Sie mit der Taste **F4** das Eigenschaftenblatt ein. In der Eigenschaft

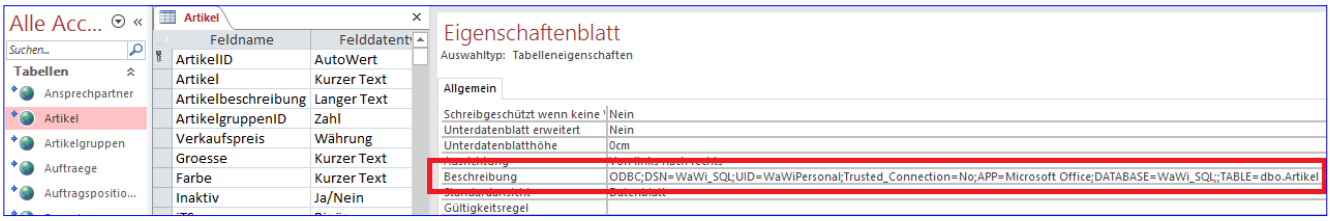


Bild 1: Beschreibung zur Datenquelle

Beschreibung sehen Sie die Informationen zur Datenquelle der eingebundenen Tabelle (siehe Bild 1). Neben dem Verweis zum DSN zeigt die **Eigenschaft** noch weitere Angaben:

```
ODBC; DSN=WaWi_SQL; UID=WaWiPersonal; Trusted_Connection=No; APP=Microsoft Office; DATABASE=WaWi_SQL; ; TABLE=dbo.Artikel
```

Mit dem Namen der Datenbank im Parameter **DATABASE**, dem Anmeldenamen unter **UID** sowie der Bezeichnung der Originaltabelle im Parameter **TABLE** enthält die Eigenschaft drei wesentliche Informationen zur externen Datenquelle. Der Eintrag **Trusted_Connection=No** zeigt, dass für die Verbindung eine Anmeldung per SQL Server-Authentifizierung verwendet wird. Das Kennwort zur Anmeldung ist an dieser Stelle aus Sicherheitsgründen nicht zu sehen. Als ergänzende Information ist im Parameter **APP** noch der Name der Applikation aufgeführt.

Neben den eingebundenen Tabellen bieten Ihnen zwei Pass Through-Abfragen den Zugriff auf die Daten der SQL Server-Datenbank. Diese enthalten in ihren Eigenschaften ebenso die Informationen zum DSN der ODBC-Datenquelle. Öffnen Sie die Pass Through-Abfrage **ptSelectGeburtstagsliste** in der Entwurfsansicht und aktivieren Sie dort das Eigenschaftenblatt.

Hier ist es die Eigenschaft **ODBC-Verbindung**, die Ihnen die Informationen zur Datenquelle zeigt:

```
ODBC; DSN=WaWi_SQL; UID=WaWiPersonal; PWD=SicherIn2020!; Trusted_Connection=No; APP=Microsoft Office; DATABASE=WaWi_SQL;
```

Der Eintrag unterscheidet sich zu dem einer eingebundenen Tabelle in zwei Parametern. Zum einen fehlt logischerweise der Verweis zur Originaltabelle und zum anderen ist hier das Kennwort klar und deutlich zu sehen. Im Gegensatz zur Tabelle sind diese Informationen weitaus mehr als eine Beschreibung, sie gehören vielmehr zur Definition der Pass Through-Abfrage. Über die Schaltfläche am Ende der Eigenschaft **ODBC-Verbindung** können Sie diese Definition ändern. Klicken Sie auf die Schaltfläche und wählen Sie im folgenden Dialog in der Registerkarte **Computerdatenquelle** die ODBC-Datenquelle über den DSN **WaWi_SQL** aus. Beim Anmeldedialog verwenden Sie die Anmelde-ID **WaWiMa** mit dem zugehörigen Kennwort **SicherIn2020!**. Es folgt die Frage, ob Sie das Kennwort speichern möchten. Mit einem Klick auf **Ja** übernehmen Sie das Kennwort sowie den Verweis zum DSN in die Eigenschaft **ODBC-Verbindung**. Diese zeigt nun folgenden Inhalt:

```
ODBC; DSN=WaWi_SQL; UID=WaWiMa; PWD=SicherIn2020!; Trusted_Connection=No; APP=Microsoft Office; DATABASE=WaWi_SQL;
```

Sie sehen zum einen den DSN **WaWi_SQL** und zum anderen unter **UID** und **PWD** die Anmeldedaten zur Anmeldung **WaWiMa**. Indirekt stehen nun zwei Anmeldungen zur Verfügung, die Anmeldung **WaWiPersonal** im DSN und der direkte Eintrag zur Anmeldung **WaWiMa** in den Eigenschaften der Pass Through-Abfrage.

Welche Anmeldung wird denn nun beim Datenzugriff verwendet? Ein Klick auf Ausführen beantwortet diese Frage. Anstelle der Geburtstage der Mitarbeiter sehen Sie eine Fehlermeldung. Das ist korrekt und es entspricht der

aktuell gültigen Rechtevergabe der Beispielumgebung. Der Anmeldung **WaWiMa** wird das Ausführen der gespeicherten Prozedur **pSelectGeburtstagsliste** verweigert. Für den Datenzugriff sind also die Anmeldedaten der Pass Through-Abfrage maßgebend. Die im DSN gespeicherten Anmeldedaten hingegen spielen dabei keine Rolle. Eine Abweichung der Anmeldedaten ist auch bei eingebundenen Tabellen möglich. Dazu geben Sie beim Einbinden der Tabellen nach der Auswahl der ODBC-Datenquelle einfach andere Anmeldedaten an.

Gerade solche abweichenden Konfigurationen führen schnell zu einer falschen Bewertung des Zugriffsschutzes. So könnte ein Blick in den DSN eine Anmeldung mit geringen Zugriffsrechten zeigen, in Access jedoch wird eine andere Anmeldung mit weitaus höheren Rechten verwendet. Also gerade umgekehrt zum aktuellen Beispiel, dafür aber mit schlimmeren Folgen. Der Anwender agiert mit mehr Rechten als der Administrator dies nach einem Blick in den DSN vermutet. Wenn das kein gutes Argument ist, in Zukunft auf den DSN zu verzichten!

Ohne Data Source Name – DSN-less

Aktuell stellt der DSN **WaWi_SQL** die Bezeichnungen zum verwendeten Treiber, zu Ihrem SQL Server und zur Datenbank bereit. Für einen Datenzugriff werden diese Informationen mit den in Access gespeicherten Anmeldedaten ergänzt und in einer Verbindungszeichenfolge zusammengefasst. Diese ebenso als Connection-String bekannte Verbindungszeichenfolge ist die Basis für die Verbindung zur externen Datenquelle. Sie beinhaltet im Grunde genommen die Wegbeschreibung, in diesem Fall die Route zur Ihrem SQL Server und dort zur Datenbank **WaWi_SQL**. Dabei enthält sie eigentlich nur Informationen, die Sie ebenso gut direkt in Access speichern können. Sie kennen den verwendeten ODBC-Treiber, Ihren SQL Server und die Datenbank. Es fehlt Ihnen nur noch die Syntax der Verbindungszeichenfolge.

Die Syntax lässt sich recht einfach ermitteln. Dazu erstellen Sie eine weitere ODBC-Datenquelle, dieses Mal jedoch als Datei-DSN. Öffnen Sie den ODBC-Datenquellen-Editor in der bewährten Art und Weise über die Windows-Taste und der Eingabe **ODBC**. Das Suchergebnis zeigt zwei Einträge. Hier wählen Sie je nach installierter Access-Version den Eintrag **ODBC-Datenquellen (32-Bit)** oder **ODBC-Datenquellen (64-Bit)**.

Im ODBC-Datenquellen-Administrator wechseln Sie zur Registerkarte **Datei-DSN**. Dort klicken Sie auf **Hinzufügen** und wählen im folgenden Dialog den Treiber **ODBC Driver 17 for SQL Server** aus. Informationen zu diesem Treiber finden Sie in der Installationsanleitung der Beispielumgebung. Nach einem Klick auf **Weiter** legen Sie den Speicherort und Dateinamen fest. Vielleicht nennen Sie die Datei **WaWi_SQL** und speichern sie im Ordner der Beispieldateien. Bestätigen Sie die bisherigen Angaben mit einem Klick auf **Weiter** und anschließend mit der Schaltfläche **Fertig stellen**.

Jetzt beginnt die eigentliche Definition der ODBC-Datenquelle. Dabei geben Sie im ersten Schritt im Eingabefeld **Server** den Namen Ihres SQL Servers ein. In Schritt zwei legen Sie mit der dritten Option die Anmeldung per SQL Server-Authentifizierung fest und ergänzen diese Auswahl mit dem Anmeldename **WaWiPersonal** im Eingabefeld **Anmelde-ID** und dem zugehörigen Kennwort im Eingabefeld **Kennwort**. Die Datenbank **WaWi_SQL** wählen Sie in Schritt drei aus, nachdem Sie dort die Option **Die Standarddatenbank ändern auf** aktiviert haben. Mit einem Klick auf **Weiter** und anschließend auf **Fertig stellen** schließen Sie die Definition der ODBC-Datenquelle ab. Es folgt der Dialog zum Test der Verbindung. Nach einem letzten Klick auf **OK** beenden Sie das Erstellen der Datei-DSN.

Nun öffnen Sie die Datei-DSN mit einem Texteditor. Sie sehen die einzelnen Parameter zur Verbindungszeichenfolge (siehe Bild 2). Bevor Sie diese Informationen in Access nutzen können, sind noch ein paar Anpassungen notwendig. Als erstes löschen Sie die eckigen Klammern rund um den Begriff **ODBC**. Danach entfernen Sie die Einträge der

Parameter **TrustServerCertificate**, **WSID** und **APP**. Jetzt fügen Sie an jedes Zeilenende ein Semikolon ein und löschen den Zeilenumbruch. Als Ergebnis sehen Sie die Verbindungszeichenfolge in einer einzelnen Zeile. Diese ergänzen Sie mit dem Parameter **PWD** und dem Kennwort **SicherIn2020!**. Mit Ausnahme des Werts im Parameter **Server** müsste Ihre Verbindungszeichenfolge nun der in Bild 3 entsprechen.

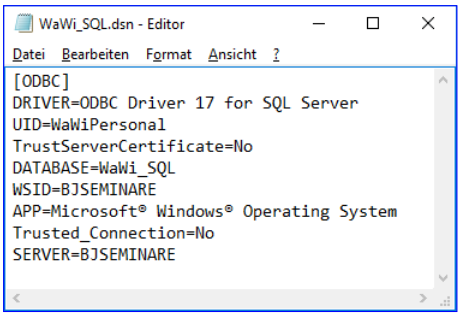


Bild 2: Inhalt einer Datei-DSN

Kopieren Sie die Verbindungszeichenfolge und gehen Sie zurück zur Access-Applikation **WaWi**. Dort öffnen Sie die Pass Through-Abfrage **ptSelectGeburtstagsliste** in der Entwurfsansicht und überschreiben den Inhalt der Eigenschaft **ODBC-Verbindung** mit der kopierten Verbindungszeichenfolge. Speichern Sie die Änderungen und klicken Sie auf Ausführen. Als Ergebnis sehen Sie die Geburtstage der Mitarbeiter – und das ohne DSN.

Wenn Sie schon dabei sind, ändern Sie doch direkt noch die Pass Through-Abfrage **ptSelectAnsprechpartnerZuMitarbeiter**. Überschreiben Sie hier ebenfalls die Eigenschaft **ODBC-Verbindung** mit dem Inhalt Ihrer Datei-DSN. Glückwunsch! Ab jetzt erfolgt der Datenzugriff der Pass Through-Abfragen ohne Verweis zu einem DSN. Beide arbeiten nun DSN-less. Alle für den Datenzugriff notwendigen Informationen sind direkt in den Pass Through-Abfragen gespeichert. Leider enthält die Eigenschaft **ODBC-Verbindung** immer noch das Kennwort im Klartext. Doch dazu später mehr.

Vorher soll das Prinzip **DSN-less** noch bei den eingebundenen Tabellen angewendet werden. Nur ist das hier nicht so einfach wie bei den Pass Through-Abfragen. Zwar enthält eine eingebundene Tabelle ebenfalls den Verweis zum DSN, dieser lässt sich dort aber nicht ändern.

Die Entwurfsansicht eingebundener Tabellen ist schreibgeschützt. Zudem zeigt die Eigenschaft **Beschreibung** lediglich eine Dokumentation zur Datenquelle. Der eigentliche Speicherort der Verbindungszeichenfolge ist die Spalte **Connect** in der Access-Systemtabelle **MSysObjects**. Dort gibt es zu jeder eingebundenen Tabelle den folgenden Eintrag – und an dieser Stelle sogar mit lesbarem Kennwort.

```
DSN=WaWi_SQL; UID=WaWiPersonal; PWD=SicherIn2020!; Trusted_Connection=No; APP=Microsoft Office; DATABASE=WaWi_SQL;
```

Die Daten sind hier ebenfalls schreibgeschützt. Die Verbindungszeichenfolge können Sie nur ändern, indem Sie die Tabellen neu einbinden. Doch da gibt es gleich die nächste Hürde. Die Dialoge über den Menüpunkt **Externe Daten** bieten keine Möglichkeit, die Tabellen ohne einen DSN einzubinden. In den Access-Versionen 2019 und 365 hat der Tabellenverknüpfungs-Manager jedoch eine Lösung für Sie.

DSN-less per Tabellenverknüpfungs-Manager

Mit dem neuen Tabellenverknüpfungs-Manager lässt sich der DSN recht einfach mit einer eigenen Verbindungszeichenfolge ersetzen. Starten Sie den Tabellenverknüpfungs-Manager über das Ribbon **Externe Daten**, aktivieren Sie dort den Eintrag **ODBC** (siehe Bild 4) und klicken Sie auf **Bearbeiten**.

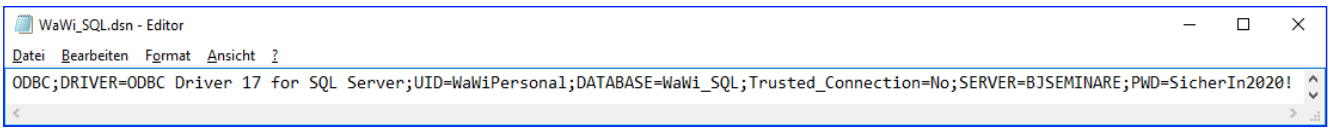


Bild 3: Die Verbindungszeichenfolge

Im folgenden Dialog überschreiben Sie das Eingabefeld **Verbindungszeichenfolge** mit dem Inhalt Ihrer Datei-DSN (siehe Bild 5) und bestätigen dies mit einem Klick auf **Speichern**. Zurück im Tabellenverknüpfungs-Manager klicken Sie auf **Aktualisieren**. Das war es schon. Sie können den Tabellenverknüpfungs-Manager wieder schließen.

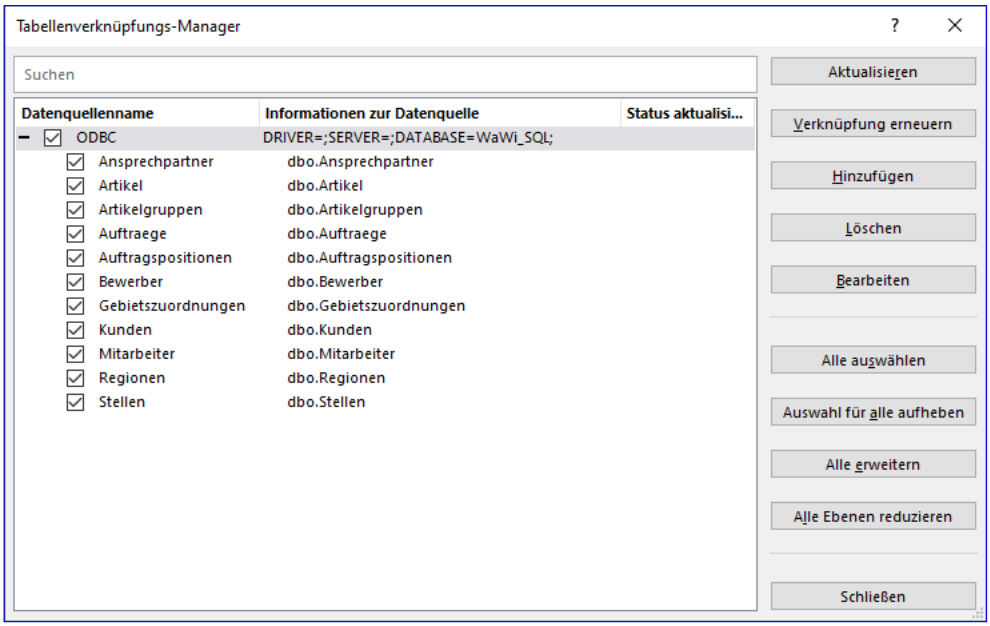


Bild 4: Der Tabellenverknüpfungs-Manager

Der Datenzugriff über eine eingebundene Tabelle erfolgt jetzt ebenfalls ohne DSN. Testen Sie es einmal. Öffnen Sie die Tabelle **Kunden** in der Entwurfsansicht und schauen Sie in die Eigenschaft **Beschreibung**. Dort sehen Sie Ihre Verbindungszeichenfolge. Das Kennwort wird aus Sicherheitsgründen weiterhin nicht angezeigt.

Der neue Tabellenverknüpfungs-Manager macht vieles einfacher. Sollten Sie noch eine ältere Version von Access verwenden, bleibt Ihnen diese Möglichkeit verwehrt. Alternativ können Sie die Tabellen per VBA einbinden.

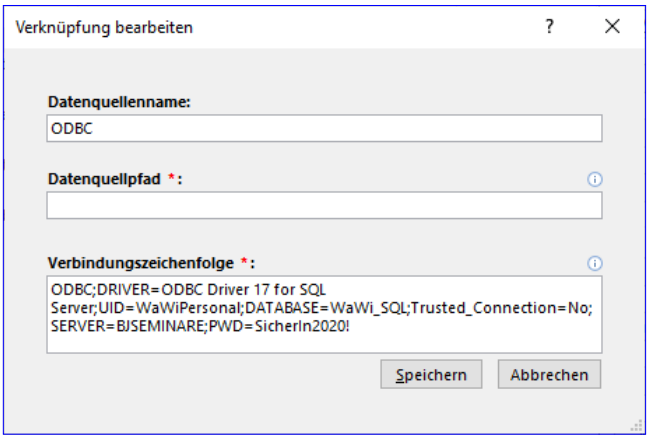


Bild 5: DSN-less per Tabellenverknüpfungs-Manager

DSN-less per VBA

Zum Einbinden der Tabellen mit einer eigenen Verbindungszeichenfolge bietet Ihnen die neue Version der Access-Applikation **WaWi** die Funktion **fTabellenEinbinden**. Schauen Sie sich einmal die Funktion an. Sie finden sie im Modul **Tabellen**.

Die Funktion legt als erstes die Verbindungszeichenfolge fest. Dies erfolgt mit einer weiteren Funktion namens **fOdbcPerFunction**.

Mit einem Rechtsklick auf den Funktionsnamen und anschließender Auswahl des Eintrags **Definition** wechseln Sie direkt zur Funktion im Modul **ODBC**. Hier überschreiben Sie den Eintrag Ihre Verbindungszeichenfolge mit dem Inhalt Ihrer Datei-DSN (siehe Bild 6). Ein erneuter Rechtsklick und der Auswahl des Eintrags **Letzte Position** bringt Sie wieder zurück zur Funktion **fTabellenEinbinden**.

Dort wird Ihre Verbindungszeichenfolge in der Variablen **strODBC** gespeichert. Anschließend löschen die folgenden Zeilen alle eingebundenen Tabellen und aktualisieren danach den neuen Stand der Tabellendefinitionen:

```
For Each tdf In CurrentDb.TableDefs
    If Left(tdf.Connect, 5) = "ODBC;" Then
        CurrentDb.TableDefs.Delete tdf.Name
    End If
Next
CurrentDb.TableDefs.Refresh
```

Es folgt das Neueinbinden der Tabellen. Die Bezeichnungen der Tabellen liefert eine temporäre Pass Through-Abfrage.

```
Set ptqry = CurrentDb.CreateQueryDef("")
With ptqry
    .Connect = strODBC
    .SQL = " SELECT [name] As Tabelle, SCHEMA_NAME(
          [schema_id]) + '.' + [name] As Originaltabelle
                    FROM sys.objects WHERE [type] = 'U';"
    .ReturnsRecords = True
End With
```

Die Pass Through-Abfrage verwendet als Verbindungszeichenfolge den Inhalt der Variablen **strODBC** und führt somit auf Ihrem SQL Server in der Datenbank **WaWi_SQL** diese SQL-Anweisung aus:

```
SELECT [name] As Tabelle, SCHEMA_NAME([schema_id]) + '.'
                        + [name] As Originaltabelle
                FROM sys.objects WHERE [type] = 'U';
```

Die **SELECT**-Anweisung ermittelt in der Tabelle **sys.objects** alle Systemobjekte vom Typ **U**. **U** steht an dieser Stelle für **User**-Table und kennzeichnet die vom Datenbankentwickler erstellten Tabellen. Das Ergebnis umfasst zwei Spalten, den Namen der Tabelle ohne Schema und den Namen der Tabelle mit Schema.

Sie möchten sich diese Daten einmal anschauen? Dann öffnen Sie das SQL Server Management Studio und melden Sie sich dort mit der Anmeldung **WaWiPersonal** an. Eventuell müssen Sie hierzu den Anmeldetyp in **SQL Server-Authentifizierung** ändern (siehe Bild 7).

```
Public Function fOdbcPerFunction() As String
'20210412 - Bernd Jungbluth
'Ermitteln der Verbindungszeichenfolge
On Error GoTo ErrHandler

    fOdbcPerFunction = "Ihre Verbindungszeichenfolge"

ExitHere:
    Exit Function
ErrHandler:
    fFehler "Tabellen", "fODBC"
    Resume ExitHere
End Function
```

Bild 6: Die Funktion **fOdbcPerFunction**

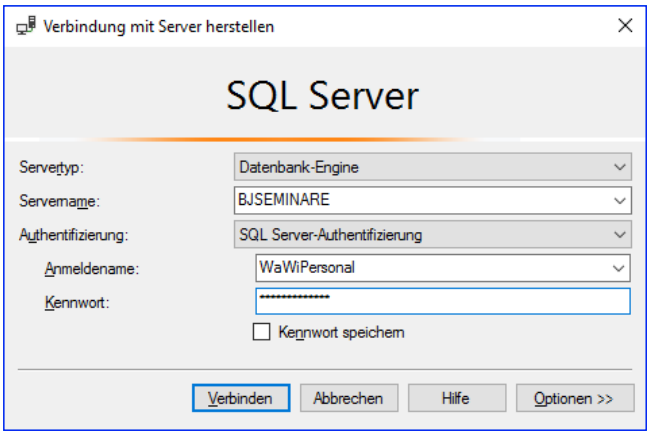


Bild 7: Die Anmeldung am SSMS

Nach der Anmeldung erweitern Sie den Objekt-Explorer, markieren die Datenbank **WaWi_SQL** mit der rechten Maustaste und wählen aus dem Kontextmenü den Eintrag **Neue Abfrage**. Sie erhalten eine leere Registerkarte, in der Sie die **SELECT**-Anweisung eingeben und dann mit der Taste **F5** ausführen. Das Ergebnis listet alle Tabellen auf, zu denen die Anmeldung **WaWiPersonal** entsprechende Zugriffsrechte besitzt (siehe Bild 8). Dabei enthält die Spalte **Originaltabelle** neben dem Tabellennamen auch den Namen des Schemas. Das ist in der aktuellen Beispieldatenbank das Standardschema **dbo**. Welche Vorteile Sie bei der Verwendung verschiedener Schemata haben, erfahren Sie in einem der nächsten Beiträge.

Die Definition der temporären Pass Through-Abfrage endet mit der Zuweisung **ReturnsRecords = True**. Diese sorgt für die Rückgabe der ermittelten Daten zur Weiterverarbeitung in VBA. Die nächste Anweisung erstellt das Recordset **rsObjekte** und füllt es mit den Daten der Pass Through-Abfrage:

```
SELECT [name] As Tabelle, SCHEMA_NAME([schema_id]) + '.' + [name] As Originaltabelle
FROM sys.objects WHERE [type] = 'U';
```

150 % Ergebnisse Meldungen

	Tabelle	Originaltabelle
1	Ansprechpartner	dbo.Ansprechpartner
2	Artikel	dbo.Artikel
3	Artikelgruppen	dbo.Artikelgruppen
4	Auftraege	dbo.Auftraege
5	Auftragspositionen	dbo.Auftragspositionen
6	Bewerber	dbo.Bewerber
7	Gebietszuordnungen	dbo.Gebietszuordnungen
8	Kunden	dbo.Kunden
9	Mitarbeiter	dbo.Mitarbeiter
10	Regionen	dbo.Regionen
11	Stellen	dbo.Stellen

Bild 8: Die Tabellen der Anmeldung **WaWiPersonal**

```
Set rsObjekte = ptqry.OpenRecordset
```

Zu jedem Eintrag des Recordsets wird dann eine neue Tabellendefinitionen erstellt.

```
Do While Not rsObjekte.EOF
    'Tabelle einbinden
    On Error GoTo 0
    Set tdf = CurrentDb.CreateTableDef(rsObjekte!Tabelle, _
            dbAttachSavePWD, rsObjekte!Originaltabelle, _
                                                strODBC)
    CurrentDb.TableDefs.Append tdf
    CurrentDb.TableDefs(rsObjekte!Tabelle).RefreshLink
    rsObjekte.MoveNext
Loop
```

Die Parameter der Anweisung **CurrentDb.CreateTableDef** beinhalten alle wichtigen Informationen für den späteren Datenzugriff. Dabei übergibt der Parameter **rsObjekte!Tabelle** den Namen, unter dem die eingebundene Tabelle später in Access zu sehen ist, während der Parameter **dbAttachSavePWD** das Kennwort der Anmeldedaten in Access speichert. Die Bezeichnung der Originaltabelle liefert der Parameter **rsObjekte!Originaltabelle** und den Weg dorthin beschreibt die Verbindungszeichenfolge im Parameter **strODBC**.

Mit den nächsten beiden Anweisungen wird die Tabellendefinition der Access-Applikation hinzugefügt und die Informationen zur neu eingebundenen Tabelle aktualisiert:

```
CurrentDb.TableDefs.Append tdf
CurrentDb.TableDefs(rsObjekte!Name).RefreshLink
```

Dieser Vorgang wiederholt sich nun für alle Tabellen, die mit der Pass Through-Abfrage ermittelt wurden. Dabei spielt die Verbindungszeichenfolge der Pass Through-Abfrage eine wichtige Rolle. Mit den Anmeldedaten der Anmeldung **WaWiPersonal** werden alle Tabellen der Datenbank eingebunden. Bei der Anmeldung **WaWiMa** hingegen sind es nur acht Tabellen. Im aktuellen Berechtigungskonzept ist dieser Anmeldung der Zugriff auf die Tabellen **Bewerber**, **Mitarbeiter** und **Stellen** verweigert – und ohne Rechte kann die Pass Through-Abfrage diese Tabellen nicht ermitteln. Die Funktion bindet also nur die Tabellen ein, zu denen die dabei verwendete Anmeldung entsprechende Zugriffsrechte besitzt. Ein nicht zu unterschätzender Sicherheitsaspekt.

Nachdem alle Tabellen der Datenbank hinzugefügt sind, wird der Navigationsbereich aktualisiert und die offenen Objekte geschlossen:

```
Application.RefreshDatabaseWindow
rsObjekte.Close
Set rsObjekte = Nothing
ptqry.Close
Set ptqry = Nothing
```

Soweit die Theorie, es folgt die Praxis. Dazu aktivieren Sie per Tastenkombination **Strg + G** den VBA-Direktbereich und führen dort die Funktion **fTabellenEinbinden** aus. Anschließend wechseln Sie zum Navigationsbereich in Access und öffnen die Tabelle **Kunden** in der Entwurfsansicht. Die Eigenschaft **Beschreibung** enthält jetzt Ihre Verbindungszeichenfolge. Ein Klick auf die Datenblatt-

ansicht liefert Ihnen die Daten der Kunden. Sie sehen, der Datenzugriff über eingebundene Tabellen funktioniert auch ohne DSN. Wodurch die ODBC-Datenquelle **WaWi_SQL** für die Beispielapplikation entbehrlich ist. Sie kann gelöscht werden. Natürlich nur, wenn Sie diese nicht in anderen Applikationen wie Excel oder weiteren Access-Datenbanken verwenden.

Für eine Installation der Access-Applikation **WaWi** auf einem neuen Rechner ist die ODBC-Datenquelle nicht mehr erforderlich. In Zukunft müssen Sie dort lediglich den ODBC-Treiber und die Access-Applikation installieren. Das reduziert nicht nur den Arbeits- und Pflegeaufwand, es schließt gleich noch eine Sicherheitslücke. Denn ohne die ODBC-Datenquelle sind auch keine unerwünschten Datenzugriffe möglich. Apropos Sicherheitslücke: Ihre Datei-DSN enthält das Kennwort im Klartext und da Sie die Datei nicht mehr benötigen, sollten Sie diese umgehend löschen.

Das Kennwort zur Anmeldung

Kennwort ist ein gutes Stichwort. Aktuell ist das Kennwort zur Anmeldung in der Funktion **fObdcPerFunction** hinterlegt und somit für jeden sichtbar, der in den Quellcode schauen darf. Das muss nicht sein, denn das Kennwort lässt sich recht einfach verstecken.

Die Access-Applikation **WaWi** arbeitet bereits mit einem versteckten Kennwort. Im Formular **Artikel** erfolgt die Datenermittlung per ADO. Die hierfür notwendige Verbindungszeichenfolge setzt die Funktion **fAdoPerTabelle** zusammen. Diese ermittelt in der lokalen Tabelle **ADO** die Bezeichnung des SQL Servers, den Namen der Datenbank und die Anmeldedaten.

Die Tabelle ist als Systemobjekt definiert. Dadurch ist sie zum einen schreibgeschützt und zum anderen im Navigationsbereich nicht ohne weiteres zu sehen. Zusätzlich ist das Kennwort in der Tabelle mit dem Eingabeformat **Kennwort** maskiert und daher in der Datenblattansicht nicht lesbar.

```
Public Function fTabellenEinbinden()
'20210412 - Bernd Jungbluth
'Tabellen einbinden
On Error GoTo ErrHandler

    Dim rsObjekte As DAO.Recordset
    Dim tdf As DAO.TableDef
    Dim strODBC As String
    Dim ptQry As DAO.QueryDef

    'Verbindungszeichenfolge ermitteln
    strODBC = fOdbcPerTabelle

    'Eingebundene Tabellen entfernen
    For Each tdf In CurrentDb.TableDefs
        If Left(tdf.Connect, 5) = "ODBC;" Then
```

Bild 9: Einbinden mit **fOdbcPerTabelle**

Ein perfektes Szenario, das sich ebenso gut zum Einbinden der Tabellen per ODBC verwenden lässt. Dann ist allerdings der Name **ADO** nicht mehr so treffend. Aus diesem Grund wurde die Tabelle in **Datenquellen** umbenannt.

Um das Einbinden der Tabellen an die neue Vorgehensweise anzupassen, müssen Sie lediglich die Funktion zum Erstellen der Verbindungszeichenfolge austauschen. So wird in der Funktion **fTabellenEinbinden** aus der Zeile **strODBC = fOdbcPerFunction** die Zeile **strODBC = fOdbcPerTabelle** (siehe Bild 9). Die Funktion **fOdbcPerTabelle** ermittelt in der Tabelle Datenquellen die Informationen zur Verbindung und erstellt die Verbindungszeichenfolge. Alles weitere bleibt unverändert.

Die Bezeichnung Ihres SQL Servers haben Sie bereits bei der Installation der Beispielumgebung in die Tabelle Datenquellen eingetragen. Sie können die Änderung also direkt testen. Dazu führen Sie im VBA-Direktbereich die Funktion **fTabellenEinbinden** erneut aus. Anschließend klicken Sie auf die einzelnen Schaltflächen im Formular Start. Da Sie die Anmeldung **WaWiPersonal** verwenden, liefert Ihnen jede Schaltfläche die entsprechenden Daten.

Das Kennwort liegt jetzt geschützt in einer versteckten Tabelle. So richtig vollständig ist der Kennwortschutz aber noch nicht. Dazu müssen Sie erst noch die Funktion **fOdbcPerFunction** löschen. Denn dort ist das Kennwort immer noch sehen.

Setup für Access-Anwendungen

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

Auch eine Access-Applikation muss irgendwann den User erreichen, und je einfacher wir es diesem machen, umso besser für uns. Zu diesem Zweck verwenden Entwickler seit vielen Jahren sogenannte Installations- oder Setup-Programme. Das kostenlose InnoSetup ist ein solches, das über den in Access bereits enthaltenen "Verpackungs- und Weitergabeassistenten" weit hinaus geht. In diesem Artikel erfahren Sie die Grundlagen, weitere Artikel befassen sich mit der Umsetzung und möglichen Erweiterungen.

Nach der Entwicklung ist vor der Auslieferung

Die »Installation« einer Access-Applikation ist eigentlich schnell gemacht. Einfach die **.accdb**- oder **.accde**-Datei irgendwohin kopieren (oder auspacken), und fertig. Dann sorgt ein Doppelklick auf diese Datei für den Start durch das hoffentlich bereits vorhandene Access. Was will man mehr?

Ja, die Frage ist erst gemeint, was kann man mehr wollen? Nun, das Übliche halt:

- ein Icon auf dem Desktop
- ein Eintrag im Startmenü zum Starten der Applikation
- ein Eintrag im Startmenü zum Öffnen der Online-Hilfe (falls vorhanden)
- eine saubere Deinstallation über die Systemsteuerung
- sicherstellen, dass die Applikation beim Update nicht bereits läuft

Das alles wäre mit einer reinen Kopieroperation sicher nicht so leicht möglich. Ein paar dieser Anforderungen könnten mit etwas Aufwand auch von der Access-Applikation selbst erledigt werden, aber spätestens bei den letzten beiden Punkten müsste die Kontrolle von außerhalb stattfinden.

Exkurs: Frontend-/Backend-Trennung

Es gibt sicher viele Möglichkeiten, eine Access-Applikation umzusetzen. Ein wichtiger Punkt dabei ist jedoch die Frontend-/Backend-Trennung. Ein Grund hierfür ist, dass alle Anwender zwar auf dieselben Daten zugreifen sollen, deren Applikation jedoch ohne Datenverlust einfach aktualisierbar sein soll.

Würden wir die Applikation mit den Datentabellen durch eine neue Version ersetzen, dann wäre alle bisherige Arbeit umsonst gewesen, die bereits eingegebenen Daten verloren. Deshalb ist es sehr empfehlenswert, die Daten von der Applikation zu trennen.

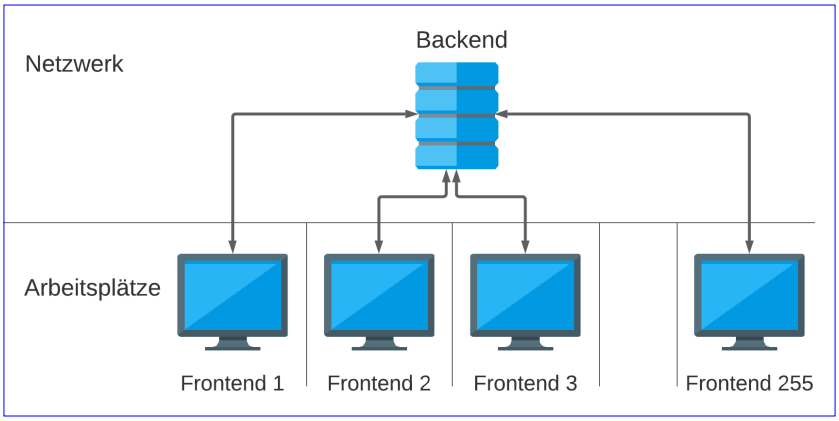


Bild 1: Frontend und Backend im Netzwerk

Kurz gesagt bedeutet das, dass unser Projekt zwei verschiedene Dateien beinhaltet (siehe Bild 1):

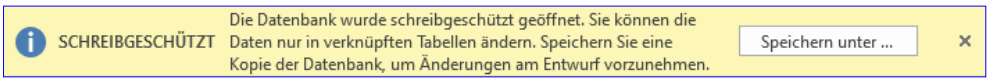


Bild 2: Warnmeldung beim Öffnen

- Das Frontend (»vorderes Ende«) steht für die Applikation selbst, also alles, was aktiv vom User gesehen und bedient wird. In Access betrifft das hauptsächlich Formulare und Berichte, aber natürlich auch Abfragen, den Programmcode in Modulen sowie eventuelle Makros.

- Das Backend (»hinteres Ende«) wiederum enthält ausschließlich die Tabellen, also die gespeicherten Daten.

Damit das Frontend auf die Tabellen zugreifen kann, müssen dort Tabellenverknüpfungen enthalten sein. Jeder dieser Einträge ist also ein Verweis auf die in einer anderen Datenbank gespeicherte Tabelle.

Access ist eigentlich genügsam, was den Speicherort angeht. Lokal oder im Netz, in **C:\Programme** oder im Benutzer-Space, auf den ersten Blick scheint das alles egal zu sein. Doch es gibt gute Orte und schlechte Orte.

Wie bereits gesagt, sollte das Backend im Netzwerk liegen. Einige Leute sagen, auch das Frontend kann einfach in's Netz, denn da kommt jeder ran. Das klingt zunächst logisch, schließlich kann Access ja auch damit umgehen, wenn mehrere Anwender gleichzeitig die Datenbankanwendung benutzen.

Ein paar Punkte gibt es dabei aber zu beachten:

- Temporäre Tabellen in der Datenbank erfordern Schreibrechte. Dabei kann es auch Konflikte geben, wenn mehrere User dieselbe Datenbank als Frontend nutzen.

- Wenn bei einem Benutzer ein Crash erfolgt, kann das die Datenbank nachhaltig beschädigen. Das hätte dann Auswirkungen auch auf alle anderen User, die bis zur erfolgreichen Reparatur nicht mehr damit arbeiten können.

- Wenn ein User in einem Formular Veränderungen vornimmt, weil ihm dies so besser gefällt, dann könnte dies die anderen Anwender sehr verwirren.

- Oder es sorgt ebenfalls für Probleme, falls der User dabei einen Fehler gemacht hat.

- Wenn wir das Installationsverzeichnis gegen Überschreiben schützen, erhalten wir eine Warnmeldung (siehe Bild 2).

- Gleiches passiert, wenn wir **C:\Programme** als Ziel einsetzen. Diese Warnmeldung sollte also verhindert werden.

Wohin mit dem Frontend?

Und nun können wir gedanklich wieder zu unserem Setup zurückkehren. Denn das Frontend ist das, welches wir auf den Benutzerrechnern per Setup installieren (lassen). Aufgrund der vorherigen Überlegungen entscheiden wir uns für den »User Program Folder«, der in Windows genau für diesen Zweck vorgesehen ist. Dieser liegt im Pfad **C:\Users\USERNAME\AppData\Local\Programs** (ersetzen Sie **USERNAME** durch Ihren Anmeldenamen, um Ihr Verzeichnis herauszufinden) und damit unter der Kontrolle dieses Users. Als Nebeneffekt ergibt sich daraus, dass für die Installation unserer Access-Applikation keine Adminrechte erforderlich sind. Da das Setup-Programm überdies sehr einfach zu bedienen ist, können wir diese Arbeit dem User guten Gewissens überlassen.

Das Backend wiederum wird wahrscheinlich vom Administrator irgendwo im Netzwerk platziert. Wo genau ist im Grunde egal, wichtig ist jedoch, dass alle User

auf diese Stelle zugreifen können. Davon müssen sie allerdings nichts erfahren, denn das macht das Frontend unserer Applikation ganz allein.

Jetzt geht es nur noch darum, dass das Frontend nach der Installation erfährt, wo genau die Backend-Datenbank liegt. Darauf kommen wir später nochmal zurück.

Um es kurz zu machen:

- Das Access-Frontend sollte auf dem Rechner jedes Benutzers liegen.
- Aus Wartungsgründen wählt man dabei optimalerweise bei allen Usern das gleiche Verzeichnis.
- Es soll durch ein Icon auf dem Desktop und/oder im Startmenü gestartet werden können.
- Es soll sich in der Systemsteuerung (**Programme hinzufügen oder entfernen**) ordentlich registrieren.
- Es soll über die Systemsteuerung vollständig deinstalliert werden können.
- Es soll sich automatisch mit dem im Netzwerk liegenden Backend verbinden.

Idealerweise könnten wir noch dafür sorgen, dass die Applikation durch die Softwareverteilung des Unternehmens »still installiert« werden kann.

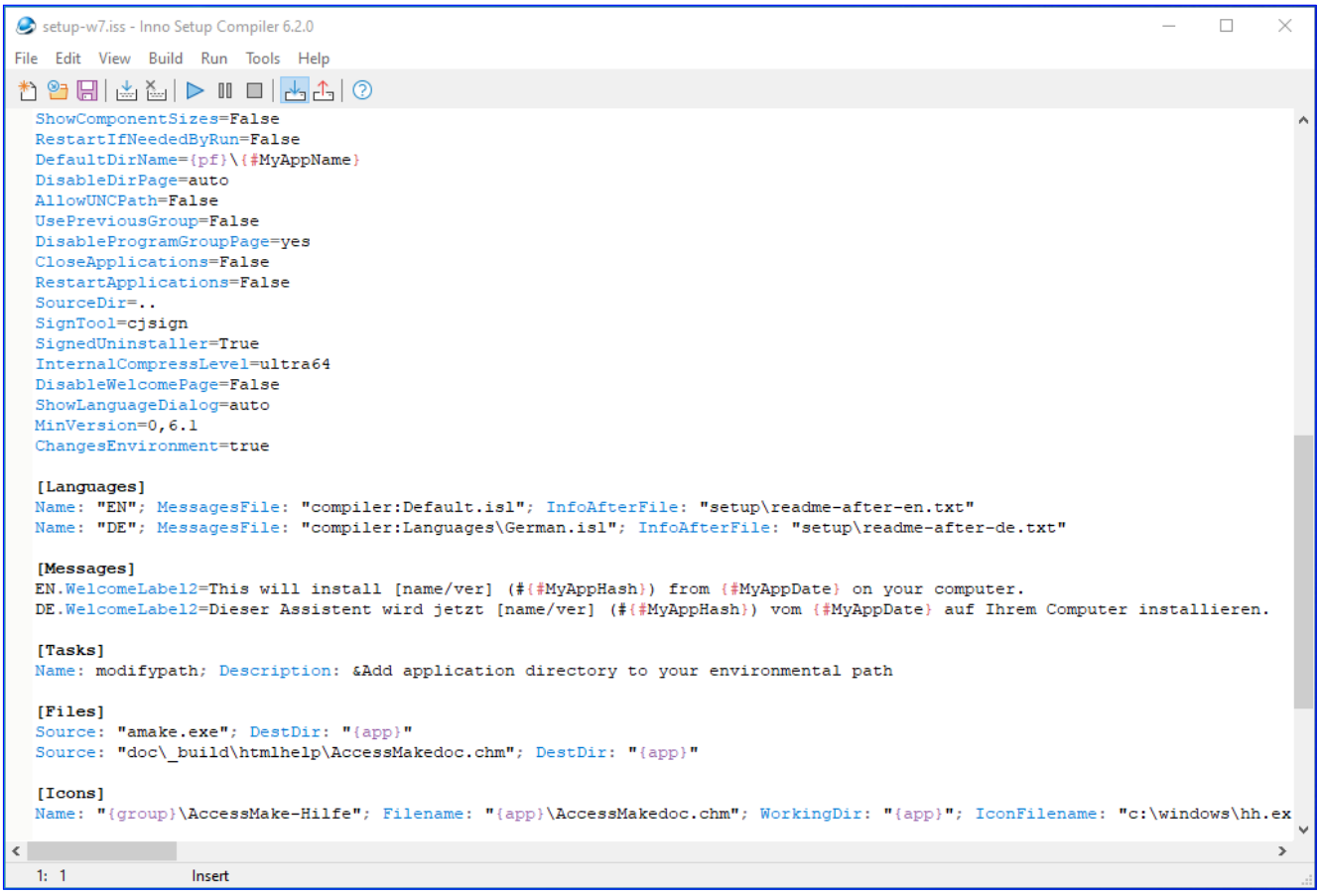


Bild 3: InnoSetup-Compiler-Fenster

COM-Add-In: Ereignisprozedur zur Laufzeit anzeigen

Bei unserer Arbeit mit Access passiert es immer wieder, dass wir schnell prüfen wollen, was eine durch ein bestimmtes Ereignis ausgelöste Prozedur überhaupt erledigt. Dann muss man in den Entwurf wechseln, einen Haltepunkt setzen, wieder den Formularentwurf aktivieren und dann das Ereignis auslösen. Wir wäre es mit einem Add-In, mit dem Sie die Ereignisse des aktuell markierten Steuerelements direkt anzeigen könnten? Ein solches COM-Add-In wollen wir in diesem Beitrag entwickeln und vorstellen. Das ist ein perfekter Anwendungszweck für die neue Entwicklungsumgebung twinBASIC, die wir in Ausgabe 3/2021 im Detail vorgestellt haben.

Debuggen schwer gemacht

Kennen Sie das auch? Sie geben testweise während der Entwicklung einer Anwendung Daten ein oder betätigen eine Schaltfläche, und es geschieht einfach nicht das, was Sie gerade erwarten. Also geht es ans Debuggen. Wechseln in die Entwurfsansicht des Formulars, dort das betreffende Steuerelement anklicken, damit seine Eigenschaften im Eigenschaftenblatt erscheinen, die passende Ereignisprozedur auswählen und auf die Schaltfläche mit den drei Punkten klicken (siehe Bild 1).

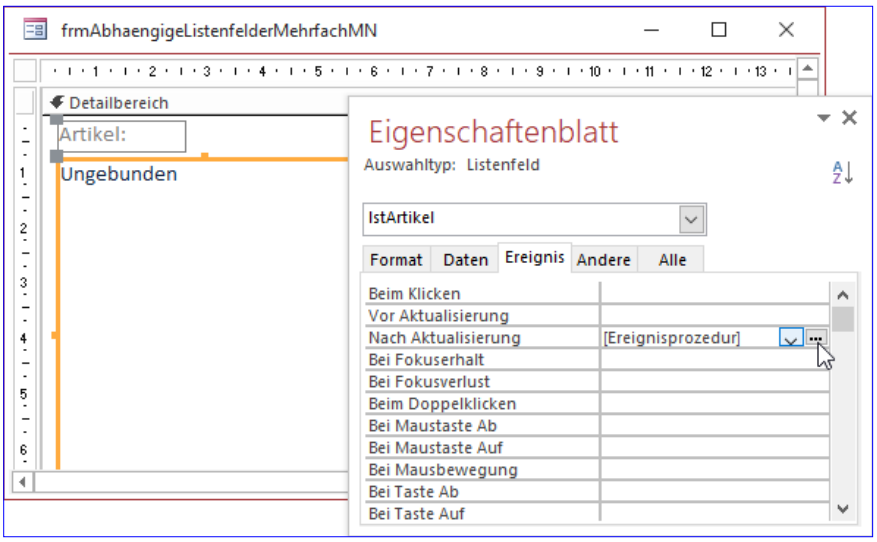


Bild 1: Üblicher Weg zum Anzeigen der Ereignisprozedur eines Steuerelements

Der VBA-Editor erscheint und zeigt die durch das Ereignis ausgelöste Prozedur an. Hier setzen wir dann einen Haltepunkt, um den Prozedurablauf gleich im Detail ansehen zu können (siehe Bild 2).

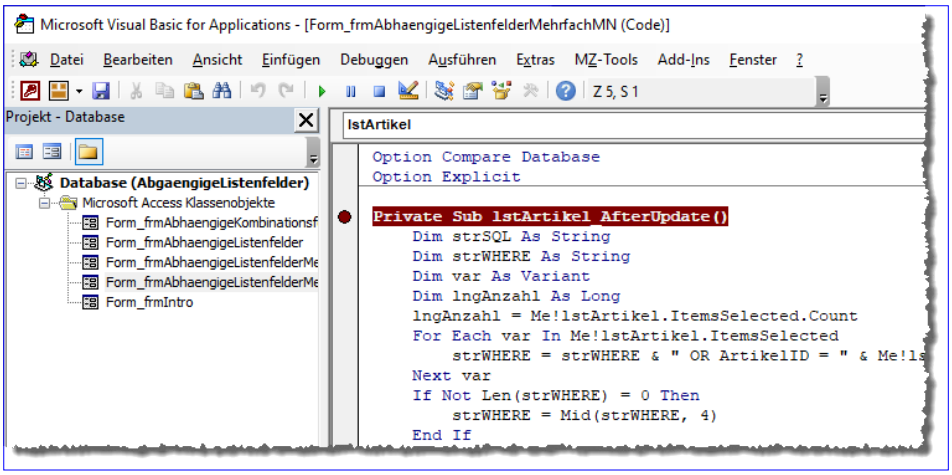


Bild 2: Setzen eines Haltepunktes in der ersten Zeile einer Ereignisprozedur

Damit sind die Arbeiten im VBA-Editor vorerst erledigt und wir können zum Access-

Fenster zurückkehren. Dort wechseln wir im aktuellen Formular wieder in die Formularansicht und führen den Vorgang erneut aus.

Gegebenenfalls ist es hierzu sogar noch notwendig, das Formular zu schließen und es anschließend über das Ribbon oder von einem anderen Formular aus erneut zu öffnen, damit die benötigte Datenkonstellation wieder vorhanden ist.

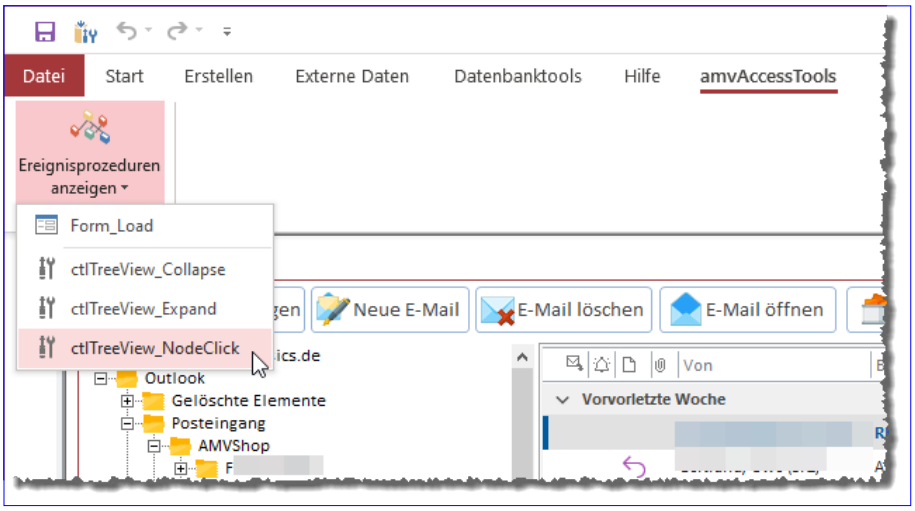


Bild 3: Anzeige der Ereignisprozedur per Ribbon-Befehl

Wenn Sie das kennen, wissen Sie: Das ist anstrengend und wirkt irgendwie aufwendiger, als es sein muss. Das haben wir uns auch gedacht und ein COM-Add-In entwickelt, dass es beim Entwickeln einer Anwendung Folgendes erlaubt: Sie markieren einfach das zu untersuchende Steuerelement und wählen per Ribbon-Befehl aus allen Ereignissen der aktuellen Situation das zu untersuchende aus, welches dann automatisch im VBA-Editor angezeigt wird.

Debuggen, die leichte Version

Nun schauen wir uns erst einmal an, was wir am Ende des Beitrags für eine Lösung erhalten. Wenn Sie nach der Installation des COM-Add-Ins ein Steuerelement markieren, können Sie einfach das Ribbon-Menü **Ereignisprozeduren anzeigen** in der Gruppe **Steuerelemente** des Tabs **amvAccessTools** anzeigen. Hier finden Sie ganz oben die Ereignisprozeduren für das aktuelle Hauptformular, in diesem Fall **Form_Load**. Darunter zeigt die Liste für dieses Beispiel die drei Ereignisprozeduren für das Steuerelement **ctlTreeView** (siehe Bild 3).

Wählen Sie einen dieser Einträge aus, erscheint der VBA-Editor und die ausgewählte Prozedur wird markiert angezeigt (siehe Bild 4). Wenn Sie ein Steuerelement in einem Unterformular selektieren, werden auch noch die Ereignisprozeduren des Unterformulars in der Liste angezeigt. Für weitere Verschachtelungsebenen ist die Lösung in der hier vorgestellten Form jedoch nicht ausgelegt.

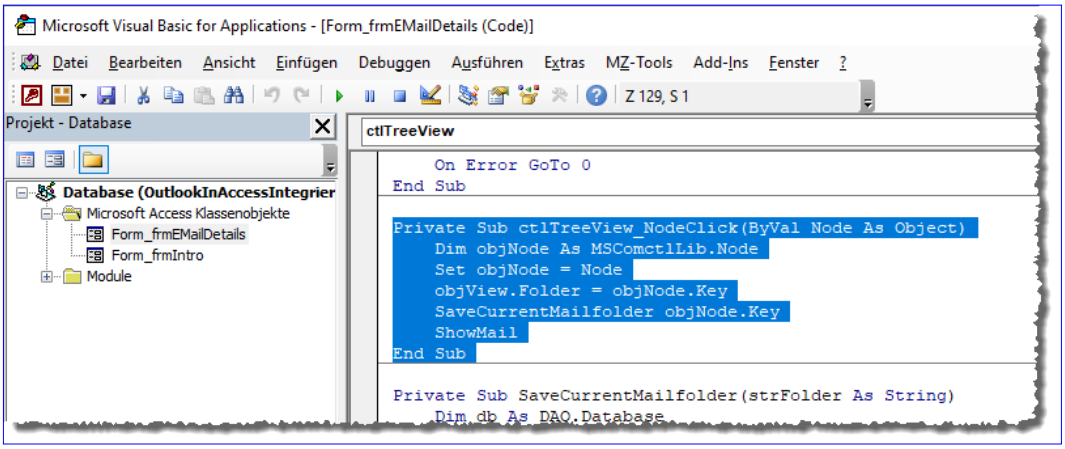


Bild 4: Anzeige und Markierung der Ereignisprozedur

Programmieren des COM-Add-Ins

Das COM-Add-In programmieren wir mit dem neuen Tool twinBASIC, das wir im Beitrag **twinBASIC – VB/VBA mit moderner**

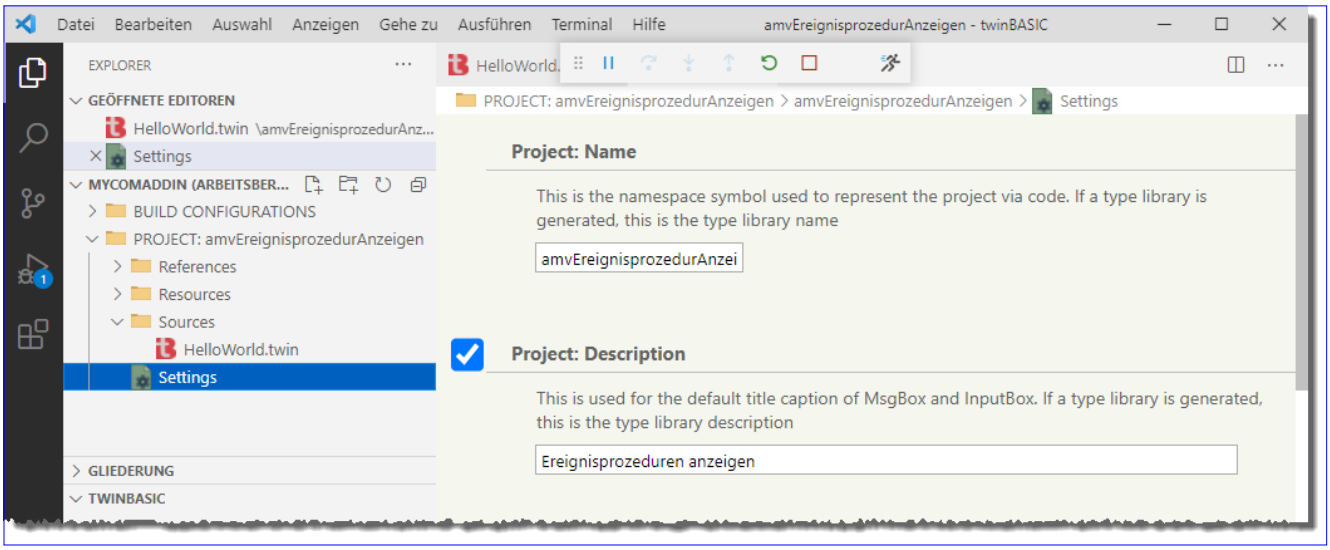


Bild 5: Anpassen der Benennung von Elementen

Umgebung (**www.access-im-unternehmen.de/1303**) im Detail vorgestellt haben. Hier erfahren Sie auch, wie Sie twinBASIC installieren. Wie Sie ein COM-Add-In grundsätzlich programmieren, beschreibt der Beitrag **twinBASIC – COM-Add-Ins für Access** (**www.access-im-unternehmen.de/1306**).

Wir starten mit dem aktuellen Beispielprojekt für ein COM-Add-In für Access, das Sie im Download zu diesem Beitrag in der Zip-Datei **twinBASIC_myCOMAddin.zip** finden.

Projekt öffnen

Nachdem Sie Visual Studio Code und twinBASIC wie im oben genannten Beitrag installiert haben, öffnen Sie das Projekt durch einen Doppelklick auf die Datei **myCOMAddin.code-workspace**.

Projektelemente umbenennen

Als Erstes nehmen wir einige Änderungen bezüglich der Benennung von Projekt, Klasse et cetera vor. Dazu öffnen Sie die durch einen Mausklick auf den Eintrag **Settings** im Projekt-Explorer die Einstellungen des Projekts.

Hier sehen Sie ganz oben die beiden Einträge **Project: Name** und **Project: Description**. Diese passen Sie wie in Bild 5 an. Wichtig: Merken Sie sich den Namen, den Sie für das Projekt vergeben, hier **amvEreignisprozedurAnzeigen**, – diesen benötigen wir gleich noch. Wichtig: Betätigen Sie auf jeden Fall die Tastenkombination **Strg + S**, um die Änderungen zu speichern!

Klassenname ändern

Dann schauen wir uns die Klasse des Projekts an, was wir durch einen Klick auf den Eintrag **HelloWorld.twin** erreichen. Diesen Eintrag ändern wir, indem wir seinen Kontextmenü-Befehl **Umbenennen** aufrufen und dann etwa die Bezeichnung **EreignisprozedurenAnzeigen.twin** eingeben.

Auch im Code der Klasse stellen wir den Namen wie folgt um:

```
Class EreignisprozedurAnzeigen
    Implements IDTExtensibility2
     ...
```

Verweise auf Bibliotheken anlegen

Schließlich benötigen wir für die nachfolgende Programmierung noch Verweise auf ein paar weitere Bibliotheken. Diese legen wir wieder im Bereich **Settings** an. Hier fügen wir im Bereich **COM Type Library / ActiveX References** die beiden folgenden Verweise hinzu:

- **Microsoft Access 16.0 Object Library**
- **Microsoft Visual Basic for Applications Extensibility 5.3**

Der Bereich sollte danach wie in Bild 6 aussehen.

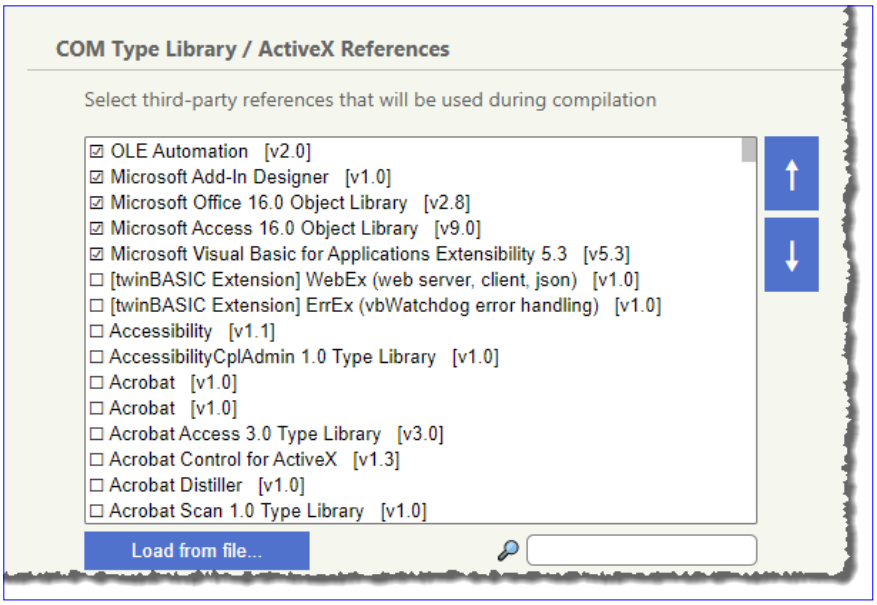


Bild 6: Hinzufügen von Verweisen

COM-Add-In für Access vorbereiten

Die Klasse **EreignisprozedurAnzeigen** enthält nun bereits einige grundlegende Elemente. Zum Beispiel legt die Zeile **Implements IDTExtensibility2** fest, dass die Klasse die angegebene Schnittstelle implementiert. Das bedeutet, dass wir die für die Schnittstelle vorgegebenen Ereignisprozeduren anlegen müssen.

Die wichtigste Ereignisprozedur ist in unserem Fall **OnConnection**. Wenn Sie Access öffnen, startet es die in der Registry angegebenen COM-Add-Ins (mehr zur Registrierung weiter unten). Dabei wird direkt **OnConnection** aufgerufen und der Parameter **Application** liefert einen Verweis auf die aufrufende Anwendung, in diesem Fall Access. Den Inhalt dieses Parameters schreiben wir direkt in die zuvor deklarierte Variable **objApplication**, die mit dem Datentyp **Access.Application** deklariert ist.

Außerdem implementiert die Klasse eine weitere Schnittstelle namens **IRibbonExtensibility**. Diese wird benötigt, um vom COM-Add-In aus Ribbon-Anpassungen vorzunehmen. Hier müssen wir zusätzlich noch die Eigenschaft **WithDispatchForwarding** voranstellen:

```
[WithDispatchForwarding]
Implements IRibbonExtensibility
```

Schließlich benötigen wir noch eine Objektvariable, mit der wir die Ribbon-Erweiterung referenzieren können:

```
Public objRibbon As IRibbonUI
```

Das Grundgerüst der Klasse haben wir in Listing 1 abgebildet.

Aufruf über das Ribbon

Die Funktion des COM-Add-Ins soll, wie eingangs erwähnt, über einen Befehl beziehungsweise ein Menü im Ribbon bereitgestellt werden. Damit dieses erscheint, müssen wir die Ribbon-Erweiterung, die das COM-Add-In bereitstellt, über die ebenfalls beim Start des Add-Ins aufgerufene Funktion **GetCustomUI** zusammenstellen und als Rückgabewert dieser Funktion definieren.

In dieser stellen wir den kompletten XML-Ausdruck für die Ribbon-Erweiterung zusammen (siehe Listing 2).

Dies liefert eine XML-Definition wie die folgende:

```
<customUI xmlns="http://schemas.microsoft.com/
                office/2006/01/customui" onLoad="OnLoad">
    <ribbon startFromScratch="false">
    <tabs>
      <tab id="tabTest" label="amvAccessTools">
```

```
Class EreignisprozedurAnzeigen
    Implements IDTExtensibility2

    [WithDispatchForwarding]
    Implements IRibbonExtensibility

    Private objApplication As Access.Application

    Sub OnConnection(ByVal Application As Object, ByVal ConnectMode As ext_ConnectMode, _
            ByVal AddInInst As Object, ByRef custom As Variant()) Implements IDTExtensibility2.OnConnection
        Set objApplication = Application
    End Sub

    Sub OnDisconnection(ByVal RemoveMode As ext_DisconnectMode, ByRef custom As Variant()) _
        Implements IDTExtensibility2.OnDisconnection
    End Sub

    Sub OnAddInsUpdate(ByRef custom As Variant()) Implements IDTExtensibility2.OnAddInsUpdate
    End Sub

    Sub OnStartupComplete(ByRef custom As Variant()) Implements IDTExtensibility2.OnStartupComplete
    End Sub

    Sub OnBeginShutdown(ByRef custom As Variant()) Implements IDTExtensibility2.OnBeginShutdown
    End Sub
    ...
End Class
```

Listing 1: Grundstruktur des COM-Add-Ins

```
        <group id="grpSteuerelemente" 
                                label="Steuerelemente">
          <dynamicMenu id="dmnEvents" 
                    label="Ereignisprozeduren anzeigen" 
                               getContent="GetContent" 
                   imageMso="CreateModule" size="large"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Das Kernstück ist das **dynamicMenu**-Element, das mit der **getContent**-Callback-Funktion beim Aufklappen dynamisch die Ereignisprozeduren für die aktuell markierten Elemente zusammenstellen soll.

Die dazu notwendige Funktion schauen wir uns weiter unten an.

Ribbon per Variable referenzieren

Wichtig ist auch noch, dass wir für das Attribut **onLoad** des Elements **customUI** eine Callback-Funktion angeben, die beim ersten Laden des Ribbons ausgelöst wird.

Diese schreibt den mit dem Parameter **ribbon** gelieferten Verweis auf das Ribbon in die weiter oben deklarierte Variable **objRibbon**:

```
Sub OnLoad(ribbon As IRibbonUI)
    Set objRibbon = ribbon
End Sub
```

```
Private Function GetCustomUI(ByVal RibbonID As String) As String _
    Implements IRibbonExtensibility.GetCustomUI
    Dim strXML As String
    strXML &= "<customUI xmlns=""http://schemas.microsoft.com/office/2006/01/customui"" onLoad=""OnLoad"">" & vbCrLf
    strXML &= "  <ribbon startFromScratch=""false"">" & vbCrLf
    strXML &= "    <tabs>" & vbCrLf
    strXML &= "      <tab id=""tabTest"" label=""amvAccessTools"">" & vbCrLf
    strXML &= "        <group id=""grpSteuerelemente"" label=""Steuerelemente"">" & vbCrLf
    strXML &= "          <dynamicMenu id=""dmnEvents"" label=""Ereignisprozeduren anzeigen""" _
        & " getContent=""GetContent"" imageMso=""CreateModule"" size=""large""/>" & vbCrLf
    strXML &= "        </group>" & vbCrLf
    strXML &= "      </tab>" & vbCrLf
    strXML &= "    </tabs>" & vbCrLf
    strXML &= "  </ribbon>" & vbCrLf
    strXML &= "</customUI>" & vbCrLf
    Return strXML
End Function
```

Listing 2: Diese Funktion setzt das Ribbon zusammen.

Warum benötigen wir dies? Weil beim Öffnen des **dynamicMenu**-Elements die Callbackfunktion zum Zusammenstellen des Menüs nur einmal aufgerufen wird – außer, wir mach diese »ungültig«. Und dazu müssen wir eine Methode der Objektvariablen für das **IRibbonUI**-Objekt aufrufen.

dynamicMenu beim Aufklappen des Menüs füllen

Nun kommt der spannende Teil. Wie füllen wir das **dynamicMenu**-Element beim Aufklappen des Menüs? Die erste Anmerkung ist für Programmierer, die bereits einmal **dynamicMenu**-Elemente unter Access verwendet haben.

Hier unter twinBASIC nutzen Sie eine etwas andere Syntax als unter VBA. Unter VBA lautet die erste Zeile der Callbackfunktion:

```
Sub GetContent(control As IRibbonControl, ByRef XMLString)
```

Unter twinBASIC verwenden wir die Deklaration wie unter VB6, die etwas anders aussieht:

```
Function GetContent(control As IRibbonControl) As String
```

Danach deklariert die Funktion die notwendigen Variablen, von denen viele aus der Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3** stammen. Das bedeutet: Wir werden direkt auf den Code im VBA-Editor zugreifen, um die Ereignisprozeduren der Elemente zu ermitteln.

VBA-Projekt per Hilfsfunktion holen

Damit steigen wir nun in die Funktion **GetContent** aus Listing 3 ein.

Hier nutze wir als Erstes eine Hilfsfunktion namens **GetCurrentVBProject**, um das aktuelle VBA-Projekt zu referenzieren. Das ist nötig, weil der VBA-Editor auch schon mal mehr als ein VBA-Projekt enthält – beispielsweise, wenn ein Access-Add-In geöffnet ist. Diese Funktion durchläuft alle VBA-Projekte, bis es eines findet, dessen Dateipfad mit dem der aktuellen Access-Anwendung übereinstimmt:

```
Public Function GetCurrentVBProject() As VBProject
    Dim objVBProject As VBProject
    For Each objVBProject In objApplication.VBE.VBProjects
        If (objVBProject.FileName = 7
```