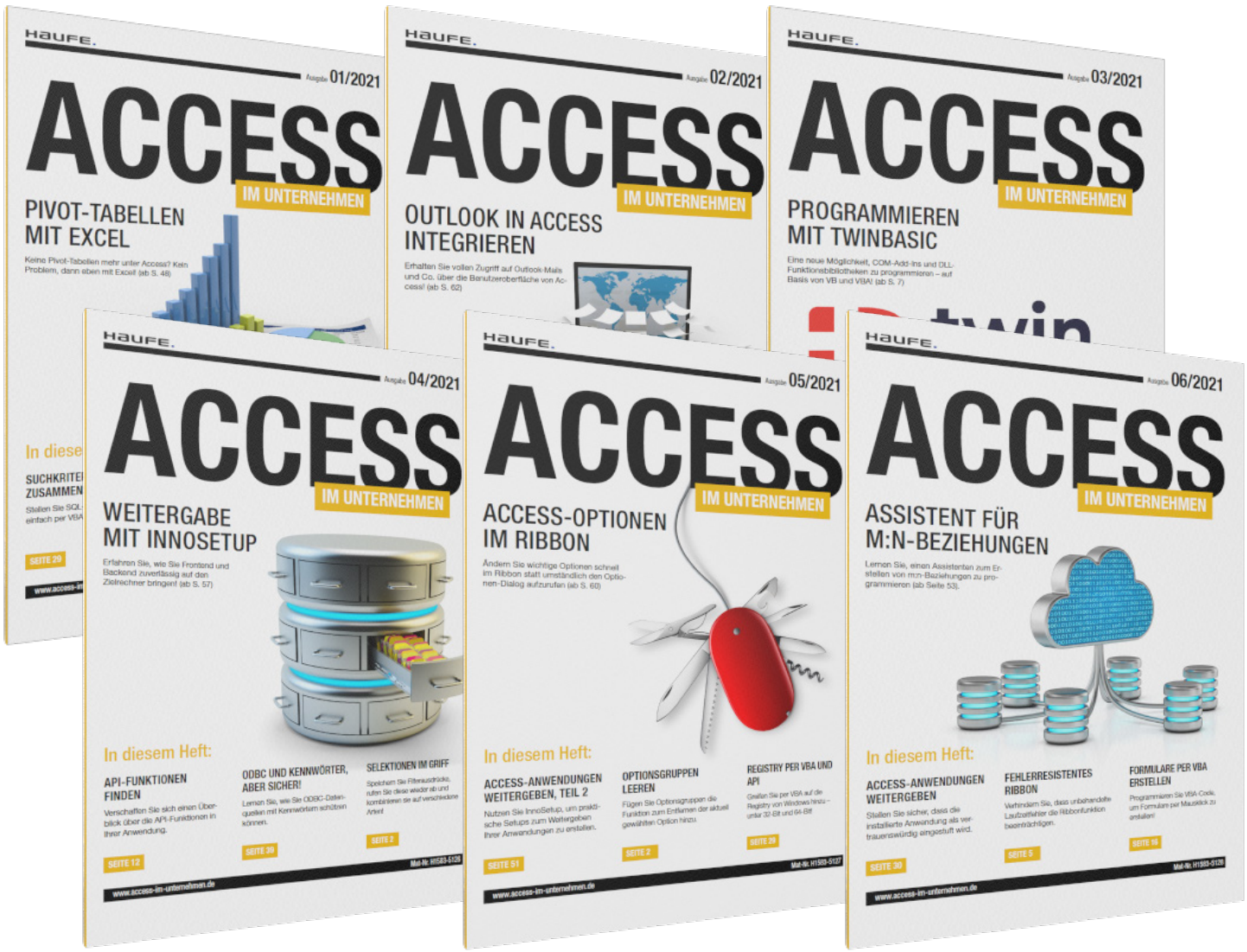
ACCESS IM UNTERNEHMEN

JAHRESARCHIV 2021



Inhalt

Access im Unternehmen Jahresarchiv 2021

Endlich ist es soweit: Es gibt ein Jahresarchiv von Access im Unternehmen mit allen Artikel des Jahres 2021! Einleitend finden Sie hier ein paar Hinweise, wie das Jahresarchiv aufgebaut ist und was es enthält. Sie finden hier die vollständigen Ausgaben eines ganzen Jahres – mit 456 Seiten Access-Know-how in fast 50 Beiträgen mit fast ebenso vielen Beispieldatenbanken zum Ausprobieren, Erweitern und Übernehmen in Ihre eigenen Lösungen.

Bevor Sie in die Lektüre der Beiträge des Jahres 2021 einsteigen, finden Sie hier noch den wichtigsten Hinweis – nämlich den, wo Sie die Beispieldatenbanken zu den einzelnen Artikeln finden. Dazu werfen Sie einen Blick in den unteren Bereich einer beliebigen Seite eines Artikels, der wie in der Abbildung unten aussieht. Hier finden Sie drei Angaben:

- Die Seite in der aktuellen Ausgabe (also die Originalseitenzahl). Diese können Sie nutzen, wenn Sie vom Inhaltsverzeichnis einer jeden Ausgabe aus starten wollen

- Die Seite im Archiv, die auch im Inhaltsverzeichnis des Jahresarchivs verwendet wird (siehe die vorherigen Seiten)

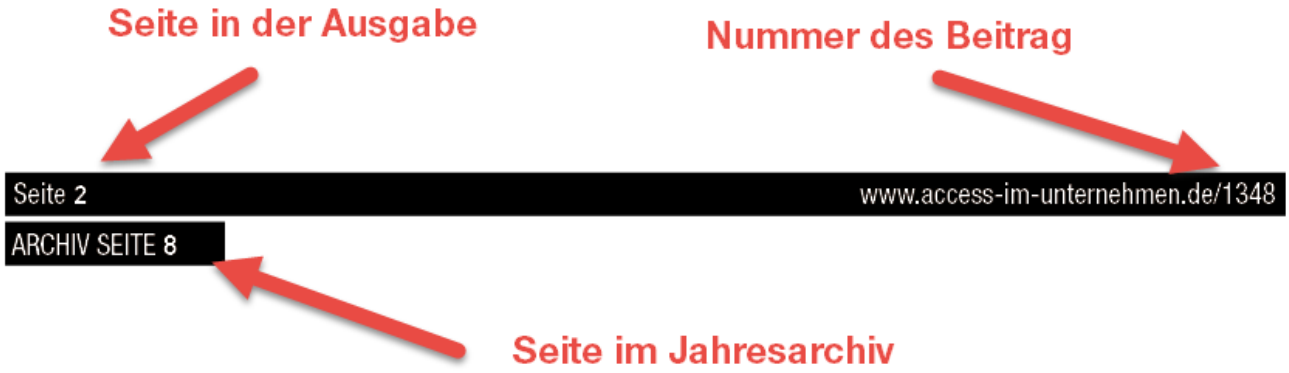
- Die Nummer des Beitrags hinter der URL zur Webseite

.

Diese Nummer ist wichtig für das Auffinden der Beispieldatenbanken. Diese finden Sie im Download im Unterverzeichnis Beispieldateien. In diesem Verzeichnis liegen weitere Unterverzeichnis, die genau diese Nummer als Bezeichnung verwenden. Hier finden Sie also die passenden Beispieldateien.

Damit wünsche ich nun, wie immer, viel Spaß beim Lesen!

Ihr André Minhorst



HAUFE.
Ausgabe 01/2021

ACCESS
IM UNTERNEHMEN

PIVOT-TABELLEN MIT EXCEL

Keine Pivot-Tabellen mehr unter Access? Kein Problem, dann eben mit Excel! (ab S. 48)

In diesem Heft:

SUCHKRITERIEN ZUSAMMENSTELLEN

Stellen Sie SQL-Ausdrucke einfach per VBA zusammen.

SEITE 29

RIBBON-ANZEIGE IM GRIFF

Zeigen Sie das Ribbon in der gewünschten Ansicht an.

SEITE 9

AUTOKORREKTUR FLEXIBEL NUTZEN

Lernen Sie, wie Sie die Autokorrektur per VBA programmieren

SEITE 2

Pivot-Tabellen und -Diagramme

Microsoft hat Access in den letzten Jahren um viele wichtige Funktionen »erleichtert«. Nach dem Sicherheitssystem und der Replikation finden wir seit Access 2013 auch die Pivot-Tabellen und -Diagramme nicht mehr als Ansichten in Access. Immerhin bietet Excel diese für die Aufbereitung und Analyse von Daten wichtigen Tools nach wie vor an. Und da Excel sich von Access aus gut steuern lässt, nutzen wir diese Funktionen in dieser Ausgabe für die Aufbereitung von Daten aus einer Access-Datenbank.

Excel bietet fast die gleichen Möglichkeiten an, die wir noch in Access 2010 zur Erstellung von Pivot-Tabellen und Pivot-Diagrammen gefunden haben. Wir schauen uns in dieser Ausgabe an, welche Möglichkeiten es gibt, um die Daten aus den Tabellen einer Access-Datenbank unter Excel in Pivot-Tabellen oder -Diagrammen darzustellen.

Der Beitrag **Pivot-Tabellen und -Diagramme in Excel** zeigt dabei ab Seite 48, wie Sie eine in einer Access-Datenbank bereitgestellte Abfrage von Access aus exportieren und als Datenquelle für eine Pivot-Tabelle nutzen können. Aufbauend auf der von Access aus erstellten Excel-Datei mit den Daten dieser Abfrage bauen wir in Excel eine Pivot-Tabelle auf. Basierend auf dieser Tabelle gehen wir noch einen Schritt weiter und legen ein Pivot-Diagramm für die Daten an.

Um dem Benutzer einige Schritte auf dem Weg von der Access-Abfrage bis zur Pivot-Tabelle in Excel zu ersparen, verwenden wir im Beitrag **Pivot-Tabellen und -Charts automatisch erstellen** ab Seite 60 die Programmiersprache VBA. Damit exportieren wir die Daten der Access-Abfrage per Mausklick und erzeugen auf die gleiche Weise eine Excel-Datei mit der gewünschten Pivot-Tabelle.

ParamArrays sind eine praktische Einrichtung, wenn Sie eine noch nicht bekannte Anzahl an Parametern an eine Funktion übergeben wollen. Alles rund um ihre Verwendung mit dem Schlüsselwort **ParamArray** lesen Sie im Beitrag **ParamArray-Auflistungen in VBA nutzen** ab Seite 41.

Die Autokorrektur-Funktion kennen viele nur als nervende Einrichtung, die an unerwünschten Stellen Texte verschlimmbessert. Der Beitrag **Autokorrektur-Einträge im Griff** zeigt ab Seite 2, wie die Autokorrektur genau funktioniert und wie Sie diese steuern können. Schließlich gehen wir noch einen Schritt weiter und zeigen, wie Sie eigene Einträge hinzufügen und damit sogar Abkürzungen wir **mfg** automatisch in Texte wie **Mit freundlichen Grüßen** umwandeln.

Das Ribbon macht manchmal, was es will – mal ist es maximiert, mal minimiert. Das ist unpraktisch, wenn Ihre Anwendung wichtige Funktionen im Ribbon anbietet und der Benutzer diese gar nicht standardmäßig zu sehen bekommt. Der Beitrag **Ribbon-Anzeige im Griff** zeigt ab Seite 9, wie Sie das Ribbon in Ihren Anwendungen so anzeigen, wie Sie es wünschen – und welche Fallstricke es noch zu beachten gibt.

Schließlich zeigen wir Ihnen im Beitrag **Add-In-Tools für den Formularentwurf** (ab Seite 34), wie Sie per Add-In schnell Steuerelemente mit durchnummerierten Bezeichnungen wie **txt01**, **txt02** und so weiter versehen.

Viel Spaß beim Lesen!

Ihr André Minhorst

Autokorrektur-Einträge im Griff

Die Autokorrektur ist ein Office-weit verwendetes Tool. Sie funktioniert auch bei der Eingabe von Texten in Access-Steuerelementen. Aber kann man die Einträge der Autokorrektur auch anpassen – und wo werden diese überhaupt gespeichert? Und kann man auch per VBA auf die enthaltenen Daten zugreifen und diese gegebenenfalls automatisiert erweitern? All diese Fragen klären wir im vorliegenden Beitrag.

Wem passiert es nicht einmal, dass er einen Tippfehler beim Eingeben von Daten in das Textfeld eines Access-Formulars macht? Oder auch in Excel, Word und Co.? Wenn das in Access passiert, weil Sie beispielsweise das Wort **wechler** statt **welcher** eingeben, korrigiert die Autokorrektur dies automatisch und zeigt auch an, dass sie aktiv war. Das erkennen Sie wie in Bild 1 an der aufklappbaren Schaltfläche, welche verschiedene Optionen anbietet:

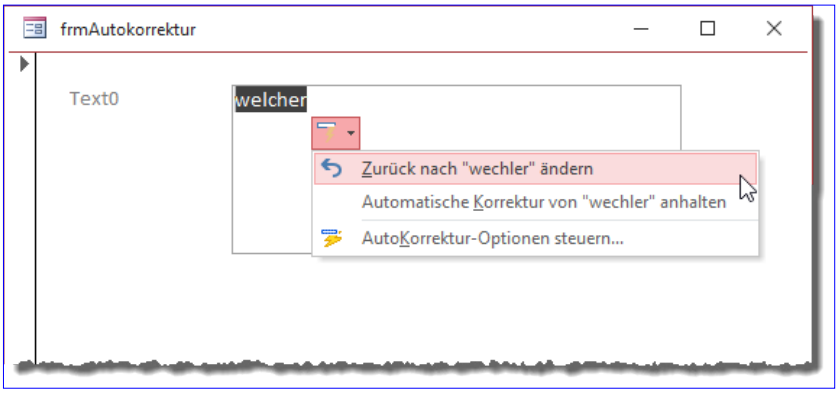


Bild 1: Die Autokorrektur in Aktion

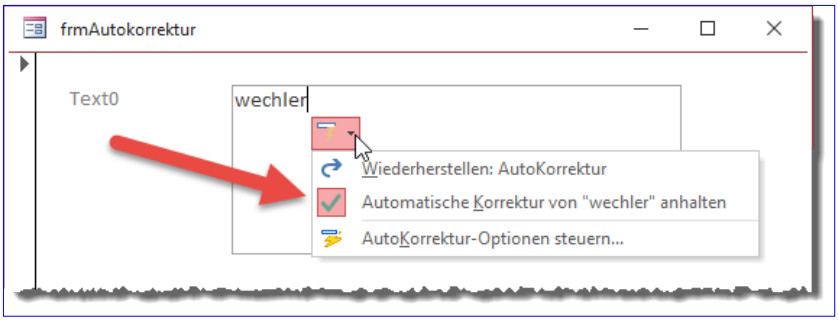


Bild 2: Die Autokorrektur wurde für dieses Wort deaktiviert.

- **Zurück nach "wechler" ändern**: Macht die durch die Autokorrektur vorgenommene Änderung rückgängig.
- **Automatische Korrektur von "wechler" anhalten**: Macht die Änderung ebenfalls wieder rückgängig und sorgt außerdem dafür, dass die Autokorrektur bei diesem Wort nicht mehr aktiv wird.
- **AutoKorrektur-Optionen steuern**: Öffnet einen Dialog, mit dem Sie die Optionen für die Autokorrektur ändern können.

Wenn Sie die Autokorrektur für ein Wort deaktivieren, erscheint vor diesem Eintrag ein Haken-Symbol (siehe Bild 2).

Autokorrektur-Optionen ändern

Wenn Sie die dritte Option wählen, zeigt Access den Dialog aus Bild 3 an. Hier finden Sie zunächst einige allgemeine Optionen, mit denen oft auftretende allgemeine Fehleingaben korrigiert werden können – zum Beispiel die Eingabe von zwei großen Zeichen am Wortanfang, weil man schneller tippt, als man die Umschalttaste wieder losgelassen hat.

Darunter finden Sie eine Liste der voreingestellten durch die Autokorrektur zu ersetzenden Begriffe. Links ist der zu

korrigierende Wert, rechts die Korrektur. Sie können mit diesem Dialog verschiedene Aktionen durchführen:

- Hinzufügen neuer Einträge: Dazu geben Sie die gewünschten Werte in die Felder **Ersetzen** und **Durch** ein und klicken auf die Schaltfläche **Hinzufügen**.

- Löschen vorhandener Einträge: Markieren Sie den zu löschenden Eintrag in der Liste und betätigen Sie die **Löschen**-Schaltfläche.

Aber Vorsicht: Diese Änderungen wirken sich auf die Autokorrektur für alle Office-Anwendungen für den aktuellen Benutzer aus. Wenn Sie also alle Einträge entfernen, wird auch Word keine Autokorrekturen mehr vornehmen.

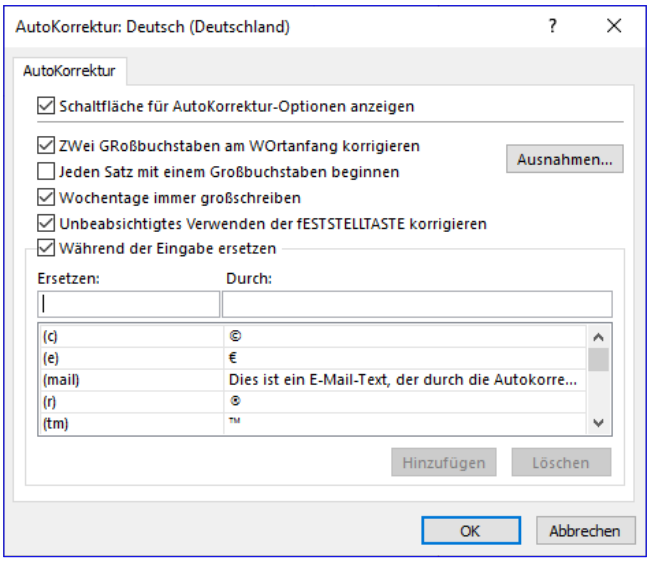


Bild 3: Optionen der Autokorrektur

Autokorrektur per VBA anpassen

Wir sind bekannt dafür, dass wir alles, was mit VBA ferngesteuert werden kann, auch unter die Lupe nehmen.

Dazu schauen wir uns zunächst die Möglichkeiten im Objektkatalog an. Geben Sie hier **Autocorrect** als Suchbegriff ein, erhalten Sie alle interessanten Elemente (siehe Bild 4).

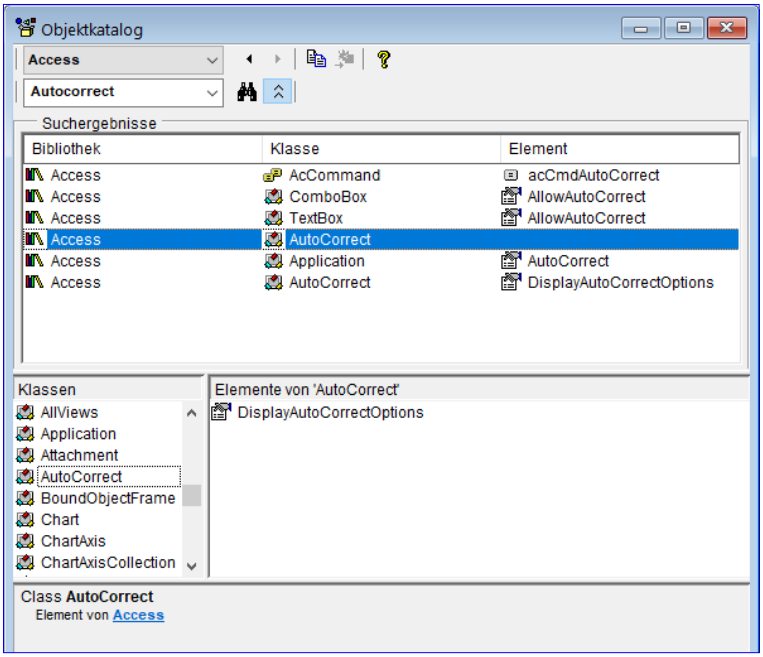


Bild 4: Das Autocorrect-Objekt im Objektkatalog

Das **AutoCorrect**-Objekt enthält selbst nur ein Element, nämlich **DisplayAutoCorrectOptions**. Auf den ersten Blick könnte man meinen, das wäre eine Methode, mit der sich der Optionen-Dialog für die Autokorrektur-Einstellungen öffnen lässt. Tatsächlich handelt es sich um eine Eigenschaft. Sie nimmt die Werte **True** oder **False** entgegen. Stellen wir die Eigenschaft testweise durch Eingabe der folgenden Anweisung im Direktbereich des VBA-Editors auf **False** ein:

```
AutoCorrect.DisplayAutoCorrectOptions = False
```

Dies führt dazu, dass die Autokorrektur zwar noch ausgeführt wird, allerdings erscheint das SmartTag mit den Optionen nicht mehr.

Neben dieser Eigenschaft gibt es noch die RunCommand-Konstante **acCmdAutoCorrect**. Diese öffnet wiederum

den Optionen-Dialog, wenn Sie einen Aufruf wie den folgenden nutzen:

```
RunCommand acCmdAutoCorrect
```

Schließlich gibt es noch die Eigenschaft **AllowAutoCorrect** für die beiden Steuerelementtypen **TextBox** und **ComboBox**. Damit können Sie festlegen, ob die Autokorrektur in einem bestimmten Textfeld aktiviert sein soll. Für das aktuelle Steuerelement deaktivieren wie die Autokorrektur beispielsweise wie folgt im Direktbereich des VBA-Editors:

```
Screen.ActiveControl.AllowAutoCorrect = False
```

Danach arbeitet die Autokorrektur einfach nicht mehr.

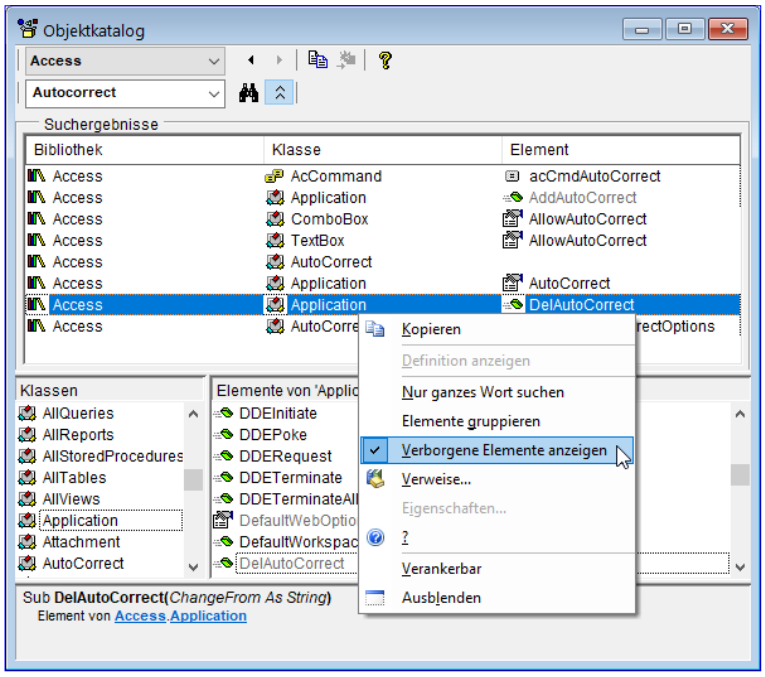


Bild 5: Einblenden verborgener Einträge

Autokorrekturen per VBA hinzufügen und löschen

Wir vermissen bisher noch eine Möglichkeit, per VBA Einträge zur Liste der Autokorrekturen hinzuzufügen. Auf herkömmlichem Wege können wir auch lange suchen, denn die beiden dazu benötigten Methoden sind als verborgen gekennzeichnet. Um diese sichtbar zu machen, klicken wir im Objektkatalog mit der rechten Maustaste in den Bereich **Suchergebnisse**. Im nun erscheinenden Kontextmenü wählen wir den Eintrag **Verborgene Elemente anzeigen** aus (siehe Bild 5).

Daraufhin erscheinen in den Suchergebnissen zwei neue Einträge:

- **AddAutoCorrect**: Fügt einen Autokorrekt-Eintrag hinzu.
- **DelAutoCorrect**: Löscht einen Autokorrekt-Eintrag.

Die beiden Befehle tauchen nun auch unter IntelliSense auf, wenn Sie **Application** gefolgt von einem Punkt im Direktbereich oder im Codefenster eingeben (siehe Bild 6).

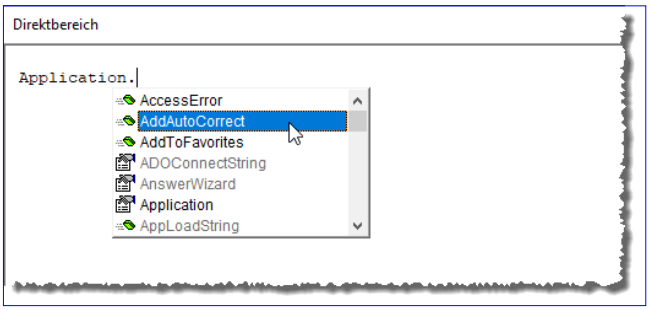


Bild 6: Befehl zum Hinzufügen eines Autokorrektur-Eintrags

Um einen Eintrag hinzuzufügen, geben Sie für den ersten Parameter der **AddAutoCorrect**-Methode den zu ersetzenden Ausdruck und für den zweiten den neuen Ausdruck ein:

```
Application.AddAutoCorrect "daß", "dass"
```

Den neuen Eintrag finden Sie dann nach dem Öffnen der Autokorrektur-Optionen in der Liste vor (siehe Bild 7).

Um diesen Eintrag wieder zu entfernen, verwenden Sie die Methode **DelAutoCorrect**. Diesmal brauchen Sie nur den zu ersetzenden Ausdruck einzugeben:

Ribbon-Anzeige im Griff

Das Ribbon von Microsoft Access macht meist, was es soll – es bietet die im aktuellen Kontext benötigten Befehle zur Auswahl an. Manchmal zickt es aber anscheinend rum: Es verschwindet oder wird nur beim Überfahren der Registerreiter sichtbar. Keine Frage: Meist sitzt das Problem vor dem Rechner und hat dieses Verhalten irgendwie angestoßen. Nur: Wie bekomme ich das Ribbon wieder in den Griff? Dieser Beitrag erläutert, welche verschiedenen Zustände das Ribbon haben kann und wie Sie diese über die Benutzeroberfläche und per VBA einstellen beziehungsweise wiederherstellen können.

Standardansicht des Ribbons

Normalerweise erscheint das Ribbon wie in Bild 1. Sie können die verschiedenen Register beziehungsweise Tabs per Klick anzeigen und diese werden dauerhaft angezeigt.

Es kann jedoch auch sein, dass nur die Leiste mit den Tab-Überschriften erscheint (siehe Bild 2). Diese Anzeige ist recht praktisch, wenn nicht viel Platz auf dem Bildschirm verfügbar ist. Sie klicken dann auf einen der Reiter, um die Befehle des jeweiligen Tabs einzublenden. Nachdem Sie den gewünschten Befehl aufgerufen haben, verschwindet die Leiste mit den Ribbon-Befehlen wieder.

Ribbon anheften

Wenn Sie das Ribbon dauerhaft sehen möchten, finden Sie nach dem Ausklappen auf der rechten Seite ein Pin-Symbol. Überfahren Sie dieses mit der Maus, erscheint die Beschreibung, die besagt, dass sich die Befehlsleiste mit dieser Schaltfläche oder mit der Tastenkombination **Strg + F1** anheften lässt (siehe Bild 3).

Mit der Tastenkombination **Strg + F1** können Sie die Leiste mit den Ribbon-Befehlen auch wieder in den nicht angehefteten Zustand bringen. Dieser Vorgang nennt sich reduzieren – über die Benutzeroberfläche stellen sie

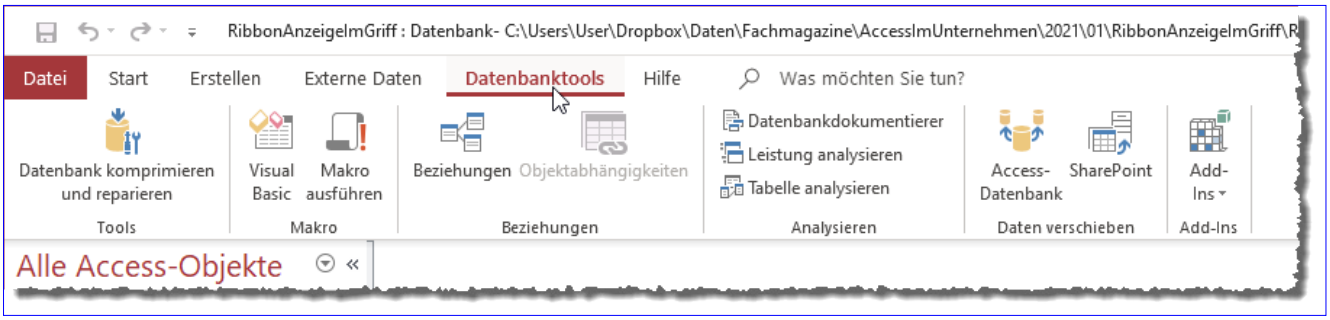


Bild 1: Standardansicht des Ribbons

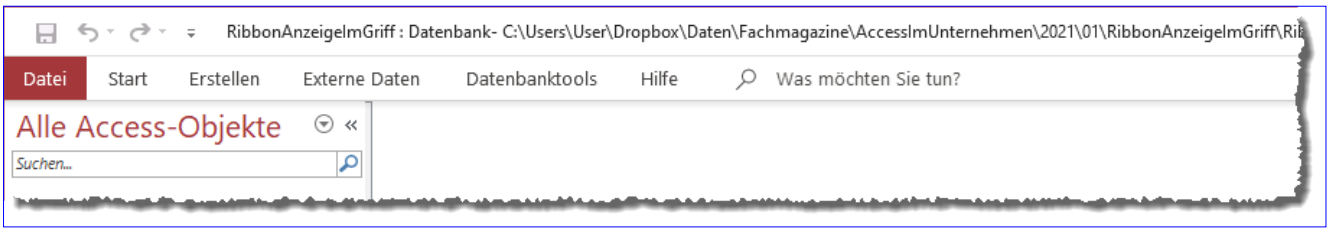


Bild 2: Ribbon minimiert

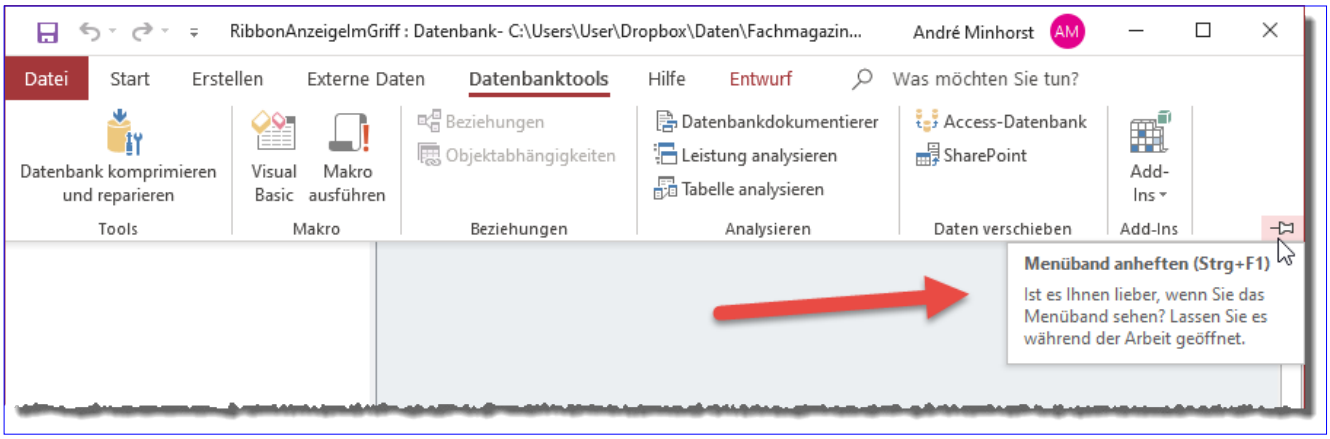


Bild 3: Ausgeklapptes, nicht angeheftetes Ribbon mit der Möglichkeit zum anheften

diesen Zustand her, indem Sie auf den kleinen Pfeil nach oben klicken, der sich im angehefteten Zustand rechts unten im Ribbon befindet.

Ein weiterer Weg, das Ribbon in den angehefteten Zustand zu überführen, ist ein Doppelklick auf einen der Tab-Reiter des Ribbons. Andersherum können Sie ein angeheftetes Ribbon per Doppelklick auf einen der Tab-Reiter auch wieder reduzieren.

Anheften und reduzieren per VBA

Es kann Gründe geben, warum Sie in Ihrer Anwendung den angehefteten oder den reduzierten Zustand herstellen möchten – beispielsweise damit der Benutzer sicher immer alle Ribbon-Befehle sieht und diese nicht verborgen sind. Die Einstellung ist Access-spezifisch, das heißt, dass diese nicht geändert wird, wenn Sie eine Anwendung schließen und die nächste mit Access wieder öffnen.

Wenn der Benutzer also normalerweise aus Platzmangel mit dem reduzierten Ribbon arbeitet und Sie dieses maximieren wollen, um die Funktionalität Ihrer Anwendung sicherzustellen, können Sie das per VBA erledigen. Sie sollten den vorherigen Zustand beim Schließen der Anwendung allerdings wieder herstellen.

Anheften und Reduzieren per Minimieren

Hier taucht eine kleine Ungereimtheit auf. Der folgende Befehl versetzt das Ribbon in den reduzierten Zustand, wenn es angeheftet war:

```
CommandBars.ExecuteMso "MinimizeRibbon"
```

Das ist noch logisch. Sie können den Befehl jedoch auch verwenden, um das Ribbon vom reduzierten in den angehefteten Zustand zu versetzen!

Wenn wir also zuverlässig den angehefteten oder den reduzierten Zustand herstellen wollen, müssen wir vorher abfragen, in welchem Zustand sich das Ribbon gerade befindet. Dafür gibt es wiederum keinen eingebauten Befehl, sondern wir nutzen die Tatsache, dass das Ribbon als **CommandBar**-Objekt referenzierbar ist, und zwar über **CommandBars("Ribbon")**. Machen Sie sich keine Hoffnungen: Das Ribbon ist nun nicht über das **CommandBar**-Objektmodell zu bearbeiten. Das **CommandBar** namens **Ribbon** enthält lediglich ein Control mit dem Wert **Ribbon** in der Eigenschaft **Caption**.

Nützlich ist jedoch, dass die **Height**-Eigenschaft dieses **Control**-Objekts je nach Zustand unterschiedliche Werte zurückliefert. Haben wir das Ribbon zuvor in den angehefteten Zustand versetzt, liefert **Height** unter Access 365 den folgenden Wert:

```
? Commandbars("Ribbon").Controls(1).Height
 161
```

Stellen wir den reduzierten Zustand her, erhalten wir dieses Ergebnis:

```
? Commandbars("Ribbon").Controls(1).Height
 65
```

Diese Werte ändern sich gelegentlich. Unter Access 2010 liefert der reduzierte Zustand beispielsweise den Wert **54** und der angeheftete Zustand den Wert **142**.

Man könnte also annehmen, dass der Wert im reduzierten Zustand unter 100 liegt und im angehefteten Zustand über 100. Allerdings haben wir die Rechnung gemacht, ohne die Möglichkeit zu betrachten, dass man die Schnellzugriffsleiste über und unter dem Ribbon platzieren kann.

Diesen Zustand ändern Sie, indem Sie in der Schnellzugriffsleiste auf das Symbol für das Aufklappmenü klicken und dort den Eintrag **Unter dem Menüband anzeigen** auswählen (siehe Bild 4).

Wenn die Schnellzugriffsleiste sich unter dem Ribbon befindet, liefert **CommandBars("Ribbon").Controls(1).Height** ganz andere Werte, nämlich **100** für den reduzierten Zustand und **196** für den angehefteten Zustand (Access 365)! Unter Access 2010 liegen die Werte bei **78** und **168**.

Wir können uns also bei der Ermittlung des aktuellen Zustandes nicht auf eine einfache Funktion wie die folgende verlassen, die von einem bestimmten Wert ausgeht:

```
Public Function MenuebandAngeheftet() As Boolean
    If CommandBars("Ribbon").Controls(1).Height > 100 Then
        MenuebandAngeheftet = True
    End If
End Function
```

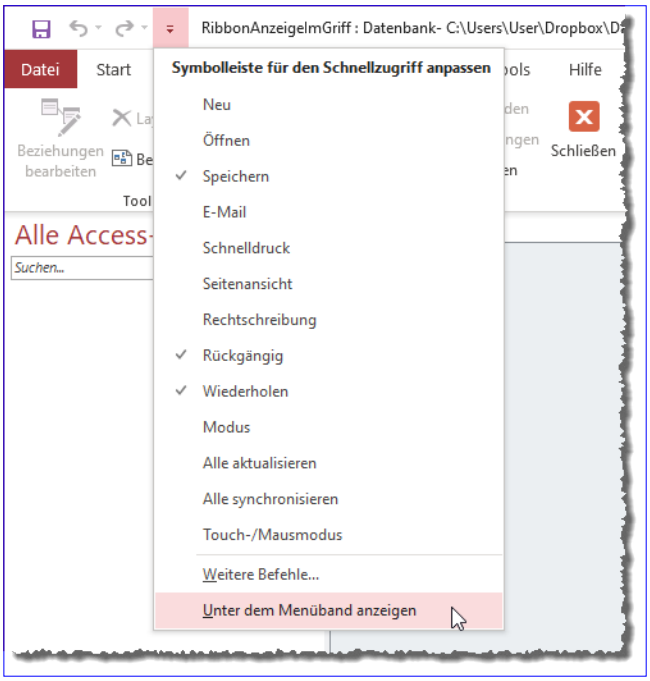


Bild 4: Einstellen, ob die Schnellzugriffsleiste über oder unter dem Ribbon angezeigt werden soll

Wo befindet sich die Schnellzugriffsleiste aktuell?

Stattdessen müssen wir noch ermitteln, ob die Schnellzugriffsleiste gerade über oder unter dem Ribbon angezeigt wird und das in die Funktion einbeziehen.

Aber Google lieferte keine Erkenntnisse darüber, wo wir eine Einstellung finden, mit der wir die Position der Schnellzugriffsleiste, auch Quick Access Toolbar genannt, einstellen können. Schließlich haben wir es auf gut Glück in der Registry gesucht und dort **QuickAccessToolbar** als Suchbegriff eingegeben. Das hat den Eintrag aus Bild 5 geliefert. Der passende Registry-Zweig lautet:

```
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\16.0\Common\
Toolbars\Access
```

Die Abbildung zeigt die Einstellung für die Position über dem Ribbon. Wenn wir die Schnellzugriffsleiste wie oben beschrieben unter das Ribbon verschieben, finden wir in

Metadaten per Zusatztabelle verwalten

Im Laufe der Zeit können sich einen Menge Daten ansammeln, die Sie in Zusammenhang mit Kunden, Produkten oder ähnlichen Entitäten speichern wollen. Dabei möchten Sie vielleicht nicht für jede neue Eigenschaft ein neues Feld anlegen und damit den Tabellenentwurf ändern. Das gilt umso mehr, wenn eine Anwendung bereits von vielen Kunden genutzt wird. Es gibt jedoch eine Alternative: Sie können eine zusätzliche Tabelle hinzufügen, die in jedem Datensatz eine Eigenschaft mit dem jeweiligen Wert für die Entität speichert. Die Frage ist nur: Wie können wir diese Daten genauso nutzen, als wenn diese wie üblich in der gleichen Tabelle wie der Kunde oder das Produkt gespeichert werden?

Nehmen wir an, Sie möchten einer Kundentabelle zwei neue Felder hinzufügen, mit denen Sie festlegen, wann der Kunde sich für den Newsletter angemeldet hat und wann er sich wieder abgemeldet hat. Sie wollen aber nicht die Kundentabelle anpassen, weil Sie das in der Vergangenheit schon öfter gemacht haben und dies immer mit viel Aufwand verbunden ist.

Stattdessen wollen Sie eine Lösung schaffen, mit der Sie langfristig zwar nach Bedarf neue Felder hinzufügen können, aber nicht jedesmal den Entwurf der Kundentabelle ändern müssen.

Lösung per Extra-Tabelle

Die Lösung ist eine zusätzliche Tabelle, welche ausschließlich Metadaten zu den Elementen der Haupttabelle speichert. Wie muss eine solche Tabelle aussehen, wenn wir dort nach Wunsch verschiedene Attribute mit ihren Werten hinzufügen wollen?

Anforderungen an die Metadaten-Tabelle

Die erste Anforderung ist, dass die Daten der Metadaten-Tabelle den Einträgen der Haupttabelle zugeordnet werden können müssen. Die Metadaten-Tabelle, nennen wir Sie hier **tblKundenMeta**, erhält also neben dem Primärschlüsselfeld **KundeMetaID** ein Fremdschlüsselfeld namens **KundeID**. Außerdem wollen wir den Namen des Attributs und den Wert des Attributs in der Tabelle speichern. Dazu legen wir zwei weitere Felder namens **Attributname** und **Attributwert** an. Dem Feld **Attributname** weisen wir logischerweise den Felddatentyp **Kurzer Text** zu.

Aber welchen Datentyp soll das Feld **Attributwert** erhalten – immerhin sollen dort gegebenenfalls Daten mit verschiedenen Datentypen gespeichert werden? In diesem Fall können wir nur den Datentyp **Kurzer Text** wählen, da alle anderen Datentypen Einschränkungen haben – zum Beispiel können Sie Zahlenfelder nicht mit Texten füllen, aber umgekehrt schon.

Jedes Attribut nur einmal pro Datensatz?

Grundsätzlich sollten wir für die Kombination des Fremdschlüsselfeldes (hier **KundeID**) und **Attributname** einen eindeutigen Index festlegen, damit jedes Attribut für jeden Kunden nur einmal angelegt werden kann – so, wie es auch in einem einfachen Feld direkt in der Kundentabelle der Fall ist.

Andererseits stellt sich die Frage, ob es nicht auch Anwendungsfälle gibt, in denen mehrere Werte für ein Attribut benötigt werden, die unter dem gleichen Attributnamen abrufbar sein sollen. Dann wäre ein zusammengesetzter, eindeutiger Index nicht sinnvoll. Und was, wenn es zwar Felder gibt, die nur einmal vorkommen dürfen, andere aber mehrmals? Dies müssten wir in der

Prozedur prüfen und behandeln, mit der wir Daten zu der Meta-Tabelle hinzufügen.

Datenmodell von Haupt- und Metatabelle

Für unser Beispiel sieht das Datenmodell wie in Bild 1 aus. Die Beziehung zwischen den Tabellen wird über das Fremdschlüsselfeld **KundeID** der Tabelle **tblKundenMeta** hergestellt. Für diese Beziehung legen wir referentielle Integrität mit Löschweitergabe fest.

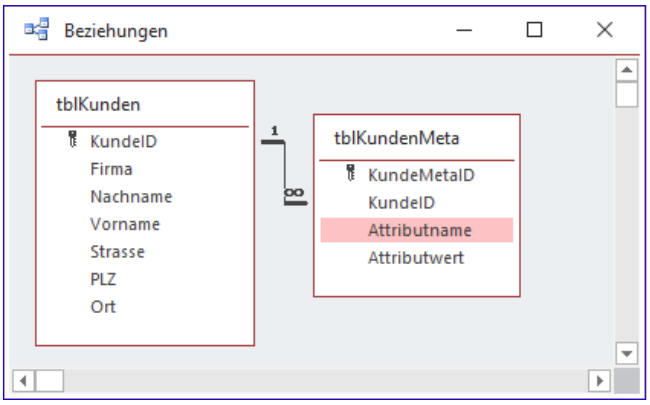


Bild 1: Datenmodell mit Metatabelle

Wenn der Benutzer dann einen Datensatz der Tabelle **tblKunden** löscht, werden die damit verknüpften Datensätze der Tabelle **tblKundenMeta** automatisch mitgelöscht.

Bearbeiten der Metadaten über die Benutzeroberfläche

Die flexibelste Art, die Metadaten in einem Formular anzuzeigen, ist ein Unterformular, dass alle Datensätze der Tabelle **tblKundenMeta** anzeigt, die zu dem im Hauptformular gehörenden Datensatz der Tabelle **tblKunden** gehören. Zum Erstellen des Unterformulars gehen Sie wie folgt vor:

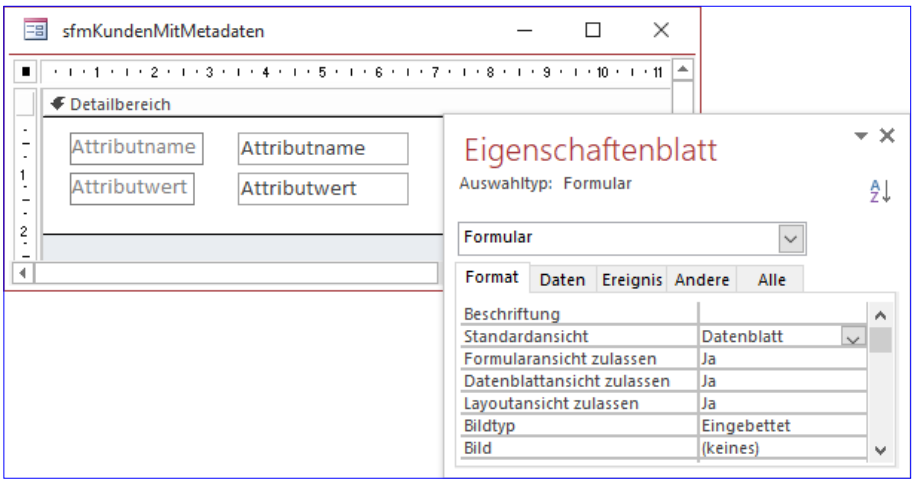


Bild 2: Entwurf des Unterformulars **sfmKundenMitMetadaten**

- Legen Sie ein neues Formular namens **sfmKundenMitMetadaten** an.
- Stellen Sie die Eigenschaft **Datensatzquelle** auf die Tabelle **tblKundenMeta** ein.
- Ziehen Sie die Felder **Attributname** und **Attributwert** in den Detailbereich des Entwurfs.
- Stellen Sie die Eigenschaft **Standardansicht** auf **Datenblatt** ein.
- Schließen und speichern Sie das Unterformular.

Den Entwurf des Unterformulars mit den Eigenschaften finden Sie in Bild 2.

Das Hauptformular legen Sie wie folgt an:

- Legen Sie ein neues Formular namens **frmKundenMitMetadaten** an.
- Stellen Sie die Eigenschaft **Datensatzquelle** auf **tblKunden** ein.
- Ziehen Sie alle Felder der Tabelle in den Entwurf.
- Ziehen Sie das Formular **sfmKundenMitMetadaten** aus dem Navigationsbereich in den Formularentwurf.

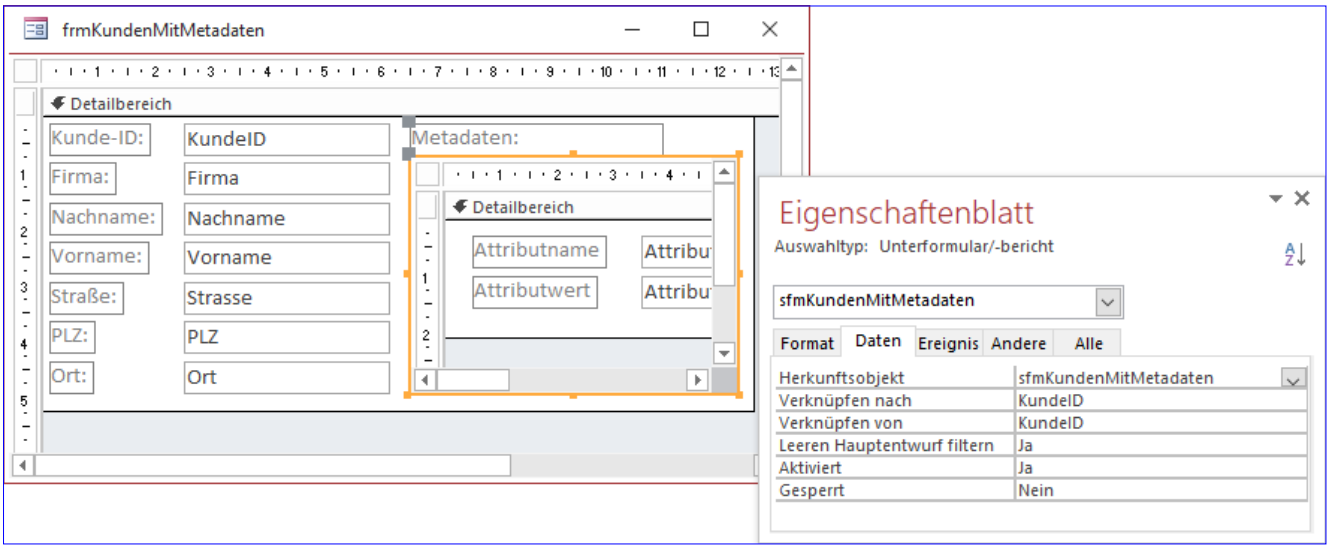


Bild 3: Entwurf des Hauptformulars **frmKundenMitMetadaten**

- Speichern und schließen Sie das Formular.

Wenn die Tabellen korrekt verknüpft sind und Sie nun das Unterformular-Steuerelement markieren, zeigen die Eigenschaften **Verknüpfen von** und **Verknüpfen nach** beide den Wert **KundeID** an (siehe Bild 3).

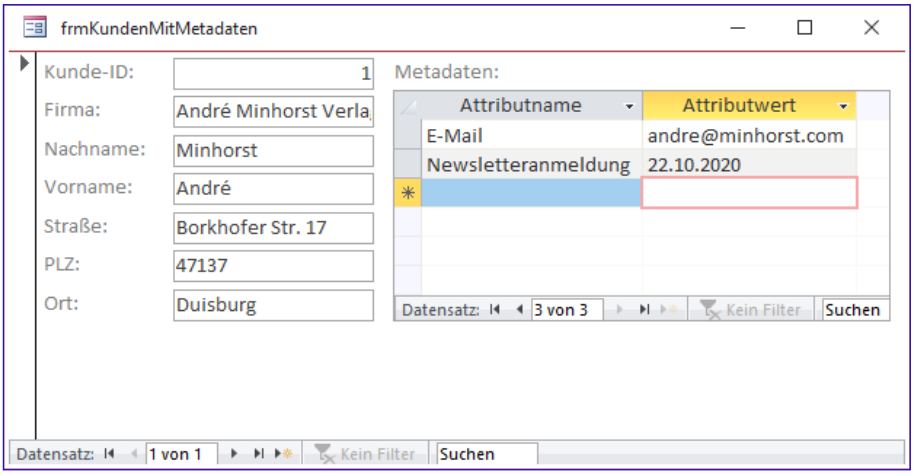


Bild 4: Eingabe von Attributen und Attributwerten

Attribute eingeben

Wechseln Sie in die Formularansicht des Hauptformulars, können Sie erste Daten eintragen. Beginnen Sie mit denen auf der linken Seite. Danach fügen wir rechts zuerst einen Attributnamen und dann den Wert ein. Auf diese Weise legen wir die Eigenschaften **E-Mail** und **Newsletteranmeldung** mit den gewünschten Weren (siehe Bild 4).

Alternative Eingabemöglichkeit

Wenn Sie nun entscheiden, dass Sie für neue Metadaten zwar keine Änderungen am Datenmodell vornehmen möchten, aber sehr wohl an der Benutzeroberfläche, wird es viel aufwendiger.

Aber warum sollte man Änderungen an der Benutzeroberfläche tolerieren, wenn man das Datenmodell nicht antasten möchte? Es kann beispielsweise sein, dass das Backend nicht so einfach angepasst werden kann, weil es beispielsweise ein SQL Server-Backend ist. Beim Frontend hingegen ist das kein Problem – wenn es vernünftig programmiert ist, können Sie dieses einfach ersetzen, ohne dass der Betrieb eingeschränkt wird.

Daher wollen wir nun die beiden neuen Attribute E-Mail und Newsletteranmeldung wie die Felder der Tabelle

tblKunden im Formular anzeigen. Dazu legen wir ein neues Formular namens **tblKundenMitMetafeldern** an.

Wenn wir eine Datensatzquelle für alle Felder erhalten wollen, benötigen wir eine entsprechend formulierte Abfrage. Diese sollte also nicht nur die Felder der Tabelle **tblKunden** enthalten, sondern auch noch zwei Felder namens **EMail** und **Newsletteranmeldung**.

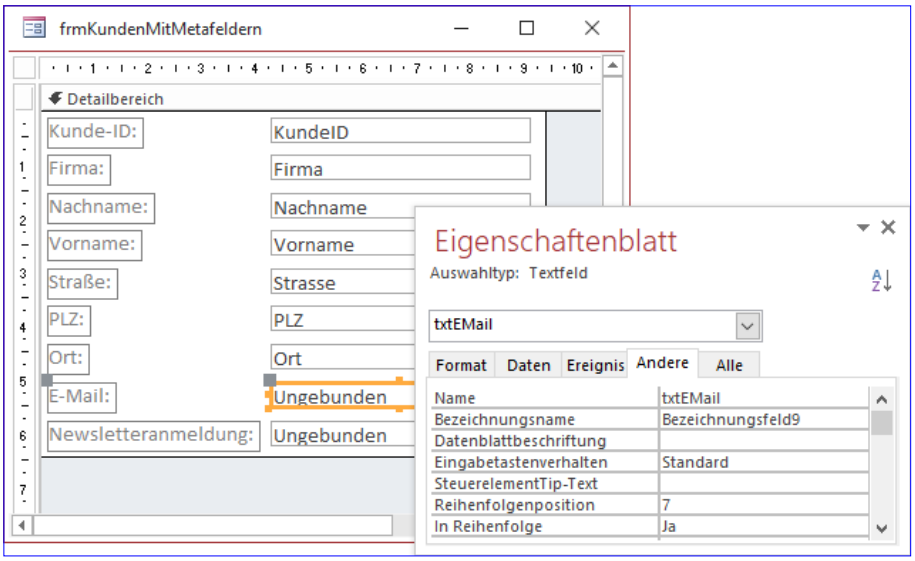


Bild 5: Ungebundene Felder für die Metadaten

Um es kurz zu machen: Eine solche Abfrage, die gleichzeitig auch noch aktualisierbar ist, lässt sich nicht erstellen. Aktualisierbar muss diese aber sein, weil wir auch Daten in das Formular eingeben wollen.

Also legen wir für die beiden Attribute **E-Mail** und **Newsletteranmeldung** ungebundene Textfelder an, die wir beim Anzeigen des Datensatzes mit den Werten der Tabelle **tblKundenMitMetadaten** füllen.

Wenn der Benutzer die Inhalte ändert, tragen wir die Änderungen per Code in diese Tabelle ein.

Die beiden Textfelder nennen wir **txtEMail** und **txtNewsletteranmeldung**. Wir fügen diese wie in Bild 5 in das Formular ein.

Meta-Attributnamen in die Marke-Eigenschaft schreiben

Für später tragen wir für die beiden Textfelder **txtEMail** und **txtNewsletteranmeldung** die jeweiligen Attributnamen in die Eigenschaft **Marke** ein, die Sie auf der Registerseite **Andere** des Eigenschaftenblatts finden.

Diese benötigen wir später für die VBA-Programmierung.

Anzeigen der Daten aus der Metadaten-Tabelle

Damit die Felder beim Anzeigen eines neuen Datensatzes gefüllt werden, hinterlegen wir eine Ereignisprozedur für das Ereignis **Beim Anzeigen**. Diese enthält zwei Anweisungen. Die erste verwendet die **DLookup**-Funktion, um den Wert des Feldes **Attributwert** des Datensatzes der Tabelle **tblKundenMeta** zu ermitteln, für den das Feld

```
Private Sub Form_Current()
    Me!txtEMail = Nz(DLookup("Attributwert", "tblKundenMeta", "KundeID = " & Nz(Me!KundeID, 0) _
        & " AND Attributname = 'E-Mail'"), "")
    Me!txtNewsletteranmeldung = Nz(DLookup("Attributwert", "tblKundenMeta", "KundeID = " & Nz(Me!KundeID, 0) _
        & " AND Attributname = 'Newsletteranmeldung'"), "")
End Sub
```

Listing 1: Einlesen der Metadaten und Schreiben in die Textfelder

KundeID dem aktuellen Kunden und das Feld **Attributname** dem gesuchten Attribut entspricht (siehe Listing 1).

Es kann vorkommen, dass gar nicht alle gesuchten Attribute in der Tabelle **tblKundenMeta** vorhanden sind. Daher fassen wir das Ergebnis der **DLookup**-Funktion noch in die **Nz**-Funktion ein, die eine leere Zeichenkette zurückliefert, wenn kein passender Datensatz gefunden werden konnte.

Schließlich wird dieser Prozedur auch aufgerufen, wenn das Formular einen neuen, leeren Datensatz anzeigt. In diesem Fall ist das Feld **KundeID** noch leer. Also fassen wir auch dieses mit der **Nz**-Funktion ein und liefern den Wert **0** zurück, wenn **KundeID** den Wert **Null** hat. Die zurückgelieferten Zeichenketten landen in jedem Fall in den beiden Feldern **txtEMail** und **txtNewsletterAngemeldet** (siehe Bild 6).

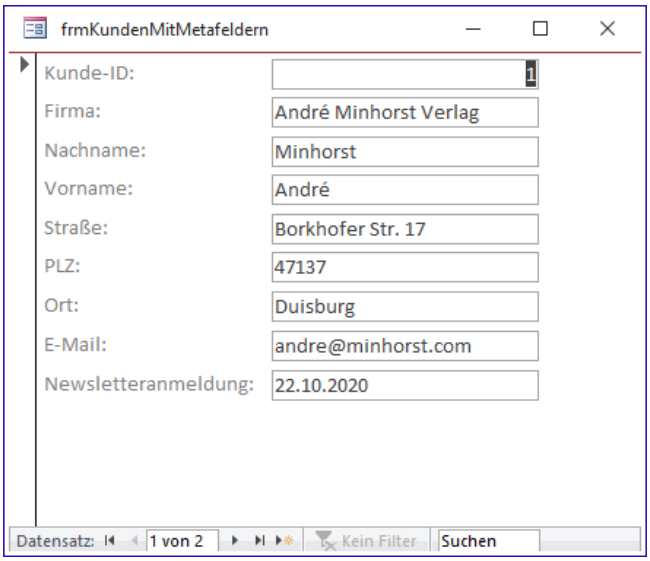


Bild 6: Die unteren beiden Textfelder enthalten Metadaten aus der verknüpften Tabelle **tblKundenMeta**.

```
Private Sub Form_AfterUpdate()
    Dim db As DAO.Database
    Dim lngKundeMetaID As Long
    Set db = CurrentDb
    lngKundeMetaID = Nz(DLookup("KundeMetaID", "tblKundenMeta", "KundeID = " & Me!KundeID _
        & " AND Attributname = 'E-Mail'"), 0)
    If lngKundeMetaID = 0 Then
        db.Execute "INSERT INTO tblKundenMeta(KundeID, Attributname, Attributwert) VALUES(" & Me!KundeID _
            & ", 'E-Mail', '" & Me!txtEMail & "')", dbFailOnError
    Else
        db.Execute "UPDATE tblKundenMeta SET Attributwert = '" & Me!txtEMail & "' WHERE KundeID = " & Me!KundeID _
            & " AND Attributname = 'E-Mail'", dbFailOnError
    End If
    lngKundeMetaID = Nz(DLookup("KundeMetaID", "tblKundenMeta", "KundeID = " & Me!KundeID _
        & " AND Attributname = 'Newsletteranmeldung'"), 0)
    If lngKundeMetaID = 0 Then
        db.Execute "INSERT INTO tblKundenMeta(KundeID, Attributname, Attributwert) VALUES(" & Me!KundeID _
            & ", 'Newsletteranmeldung', '" & Me!txtNewsletteranmeldung & "')", dbFailOnError
    Else
        db.Execute "UPDATE tblKundenMeta SET Attributwert = '" & Me!txtNewsletteranmeldung & "' WHERE KundeID = " _
            & Me!KundeID & " AND Attributname = 'Newsletteranmeldung'", dbFailOnError
    End If
End Sub
```

Listing 2: Schreiben der Metadaten

Suchkriterien einfach zusammenstellen

Wenn Sie per VBA SQL-Suchkriterien zusammenstellen, also den Teil einer SQL-Abfrage, der mit WHERE beginnt, ist das in der Regel eine Menge Schreibarbeit. Eine Menge Schreibarbeit geht oft einher mit Fehleranfälligkeit. Sie benötigen Code, um zu schauen, ob überhaupt ein Vergleichswert eingegeben wurde, müssen dann die verschiedenen Vergleichsausdrücke zusammenstellen, diese mit OR oder AND verknüpfen, dabei unterschiedliche Datentypen beachten und davon abhängig Hochkommata setzen oder auch Datumsangaben konvertieren. All dies können Sie sich sparen, wenn Sie ein paar einfache Hilfsfunktionen einsetzen.

Wenn Sie ein Suchformular mit Steuerelementen zur Eingabe mehrerer Kriterien ausstatten (wie in Bild 1), müssen Sie diese auch auswerten. Das geschieht normalerweise per VBA.

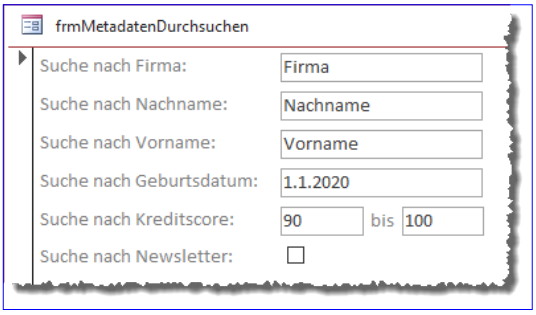


Bild 1: Beispiel-Suchformular

In einer Prozedur, die beispielsweise durch eine Suchen-Schaltfläche ausgelöst wird, stehen dann Anweisungen wie die aus Listing 1.

Hier deklarieren wir zu Beginn eine Variable namens **strWhere**, welche den Inhalt der **WHERE**-Klausel einer **SELECT**-Abfrage aufnehmen soll. Diese werden dann später zusammengesetzt und beispielsweise einem Unterformular oder einem Listenfeld als Datenquelle zugewiesen (über die Eigenschaft **Recordsource** oder **RowSource**).

In unserem Beispiel haben wir einige Felder, die einfache Texte abfragen wie **txtSucheNachFirma**. Andere fragen nach einem Datum wie **txtSucheNachGeburtsdatum** oder nach Zahlenwerten (**txtSucheNachKreditscoreBis**).

Für alle Suchsteuerelemente prüfen wir zunächst, ob dieses überhaupt einen Wert enthält (**Len(Nz(<Suchfeldname>, ""))**). Die **Nz**-Funktion wandelt **Null**-Werte in leere Zeichenfolgen um, die **Len**-Funktion ermittelt die Zeichenkettenlänge. Auf diese Weise prüfen wir in der **If... Then**-Bedingung jeweils, ob sich ein Suchbegriff im Steuerelement befindet. Nur dann führen wir den Teil innerhalb der Bedingung aus und fügen einen Teil zur Variablen **strWhere** hinzu:

```
If Not
Len(Nz(Me!txtSucheNachFirma, ""))
= 0 Then
    strWhere = strWhere & " AND
Firma LIKE '" 7
    & Me!txtSucheNachFirma & "'"
End If
```

Im ersten Fall fügen wir direkt einen Ausdruck hinzu, der mit **AND** beginnt, dann den Namen des zu untersuchenden Feldes (**Firma**), den Vergleichsoperator (**LIKE**) und den Vergleichswert (der Inhalt von **txtSucheNachFirma** in Hochkommata). Frage: Warum beginnen wir gleich den ersten Ausdruck mit **AND**, obwohl wir diesen Teil später, wenn wir diesen an **WHERE** anhängen, sowieso entfernen müssen? Der Grund ist einfach: Wenn wir allen Elementen der **Where**-Klausel ein **AND** voranstellen, können wir auch davon ausgehen, dass bei Vorhandensein mindestens eines Ausdrucks auch ein **AND** ganz vorn steht – und dieses dann einfach abschneiden.

Vergleichsausdrücke mit Datum sehen etwas anders aus. Der Grundaufbau ist zwar gleich, aber wir verwenden hier eine Funktion, um den Inhalt des Datumsfeldes, zum

```
Dim strWhere As String
If Not Len(Nz(Me!txtSucheNachFirma, "")) = 0 Then
    strWhere = strWhere & " AND Firma LIKE '" & Me!txtSucheNachFirma & "'"
End If
If Not Len(Nz(Me!txtSucheNachVorname, "")) = 0 Then
    strWhere = strWhere & " AND Vorname LIKE '" & Me!txtSucheNachVorname & "'"
End If
If Not Len(Nz(Me!txtSucheNachNachname, "")) = 0 Then
    strWhere = strWhere & " AND Nachname LIKE '" & Me!txtSucheNachNachname & "'"
End If
If Not Len(Nz(Me!txtSucheNachGeburtsdatum, "")) = 0 Then
    strWhere = strWhere & " AND Geburtsdatum = " & ISODatum(Me!txtSucheNachGeburtsdatum)
End If
If Not Len(Nz(Me!txtSucheNachKreditscoreVon, "")) = 0 Then
    strWhere = strWhere & " AND Kreditscore >= " & Replace(Me!txtSucheNachKreditscoreVon, ",", ".")
End If
If Not Len(Nz(Me!txtSucheNachKreditscoreBis, "")) = 0 Then
    strWhere = strWhere & " AND Kreditscore <= " & Replace(Me!txtSucheNachKreditscoreBis, ",", ".")
End If
If Not IsNull(Me!chkSucheNachNewsletter) Then
    strWhere = strWhere & " AND Newsletter = " & CInt(Me!chkSucheNachNewsletter)
End If
If Len(strWhere) > 0 Then
    strWhere = Mid(strWhere, 6)
    strWhere = " WHERE " & strWhere
End If
Debug.Print strWhere
```

Listing 1: Zusammensetzen einer Where-Bedingung

Beispiel **31.12.2020**, in einen universell einsetzbaren SQL-Datumsausdruck umzuwandeln, in diesem Fall **#2020/12/31 00:00:00#**. Dazu verwenden wir die Hilfsfunktion **ISODatum**, die Sie im Modul **mdlTools** finden:

```
If Not Len(Nz(Me!txtSucheNachGeburtsdatum, "")) = 0 Then
    strWhere = strWhere & " AND Geburtsdatum = " _
                & ISODatum(Me!txtSucheNachGeburtsdatum)
End If
```

Beim Datum geben wir im Gegensatz zum Auch hier stellen wir dem Teilausdruck ein **AND** voran und fügen diesen dann zum bereits vorhandenen Ausdruck **strWhere** hinzu. Wollen wir mit Zahlenfeldern vergleichen, benötigen wir keine Hochkommata, aber ähnlich wie bem Datum müssen wir berücksichtigen, dass die Zahlenwerte ein Komma enthalten können. Unter SQL wird aber immer der Punkt als Dezimaltrennzeichen verwendet. Das heißt, dass wir die **Replace**-Prozedur nutzen, um ein eventuell vorhandenes Komma durch einen Punkt als Dezimaltrennzeichen ersetzen:

```
If Not Len(Nz(Me!txtSucheNachKreditscoreVon, "")) = 0 Then
    strWhere = strWhere & " AND Kreditscore >= " _
        & Replace(Me!txtSucheNachKreditscoreVon, ",", ".")
End If
```

Schließlich müssen wir noch Kontrollkästchen betrachten. Je nach Anwendungsfall kann ein Kontrollkästchen in der deutschen Version Werte wie Ja/Nein oder Wahr/Falsch

Add-In-Tools für den Formularentwurf

Als ich neulich mal wieder einige Steuerelemente zu einem Formular hinzugefügt habe, die durchnummerierte Namen erhalten sollten wie txt01, txt02 und so weiter, kam die Eingebung: Warum diese langweilige Arbeit immer wieder von Hand erledigen, statt einfach ein Tool dafür zu entwickeln? Gesagt, getan: Es sollte ein kleines Add-In her, mit dem sich diese erste kleine Aufgabe vereinfachen ließ. Wie Sie vermutlich auch, programmiere ich nämlich lieber als dass ich immer wiederkehrende Aufgaben durchführe. Das Durchnummerieren von markierten Steuerelementen nach bestimmten Vorgaben soll die erste Funktion dieses Add-Ins sein. Ihnen und mir fallen sicher noch noch weitere Ideen für den Einsatz dieses Add-Ins ein!

Der Auslöser für die Programmierung des in diesem Beitrag beschriebenen Add-Ins ist das unscheinbare Formulare aus Bild 1. Die drei Kombinationsfelder sollen die Namen **cboSucheNach1**, **cboSucheNach2** und **cboSucheNach3** erhalten und die Textfelder rechts daneben die Namen **txtSucheNach1**, **txtSucheNach2** und **txtSucheNach3**.

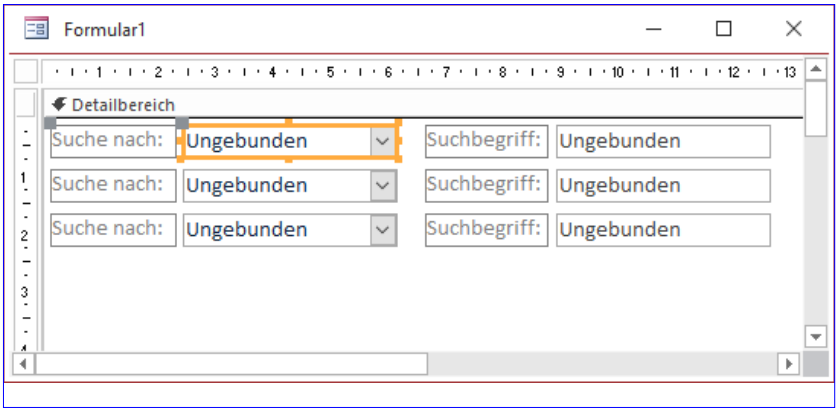


Bild 1: Zu benennende Steuerelemente

Darauf, das von Hand zu erledigen, hatte ich keine Lust. Also habe ich zuerst eine kleine Prozedur geschrieben, die wie in Listing 1 aussieht. Sie erwartet die folgenden Parameter, um die Benennung der Steuerelemente so flexibel wie möglich zu machen:

- **strBezeichnung**: Bezeichnung des Steuerelements inklusive eines Platzhalters, an dessen Stelle die Nummern eingetragen werden.
- **strPlatzhalter**: Platzhalter, der durch die Nummern ersetzt werden soll.
- **intStellen**: Anzahl der Stellen für die Durchnummerierung. Der Wert **2** würde beispielsweise bei den ersten Namen um eine führende **0** ergänzt werden, also beispielsweise **02**.
- **intStart**: Gibt die Nummer für das erste Steuerelement an. Standardwert ist **1**. Geben Sie einen anderen Wert an, wird die Nummerierung mit diesem begonnen.

Ein Aufruf dieser Prozedur sieht beispielsweise wie folgt aus:

```
SteuerelementeBenennen "cboSucheNach[]", "[]", 1, 1
```

Hier wird immer der Basisname **cboSucheNach[]** verwendet, bei dem die Zeichenfolge **[]** durch die aktuelle Zahl ersetzt wird – in diesem Fall mit einer Stelle, also ohne führende **0** bei den ersten Einträgen.

```
Public Sub SteuerelementeBenennen(strBezeichnung As String, strPlatzhalter As String, intStellen As Integer, _
        Optional intStart As Integer = 1)
    Dim frm As Form
    Dim ctl As Control
    Dim intNummer As Integer
    Dim strStellen As String
    intNummer = intStart
    Set frm = Screen.ActiveForm
    strStellen = String(intStellen, "0")
    For Each ctl In frm.Controls
        If ctl.InSelection Then
            ctl.Name = Replace(strBezeichnung, strPlatzhalter, Format(intNummer, strStellen))
            intNummer = intNummer + 1
        End If
    Next ctl
End Sub
```

Listing 1: Prozedur zum Benennen von Steuerelementen

Die Prozedur trägt zuerst den Wert der Startzahl aus dem Parameter **intStart** in die Variable **intNummer** ein. Dann referenziert sie das aktuell geöffnete Formular mit **Screen.ActiveForm**. Diese erste Fassung enthält noch keine Prüfung, ob beispielsweise überhaupt ein Formular geöffnet ist oder ob der Benutzer Steuerelemente zum Umbenennen markiert hat.

Dann verwendet die Prozedur den Wert aus dem Parameter **intStellen**, um in **strStellen** eine Zeichenkette mit so vielen Nullen zu füllen, wie es in **intStellen** angegeben ist. Damit starten wir in eine **For Each**-Schleife über alle Steuerelemente des mit **frm** referenzierten Formulars. In der Schleife prüfen wir zuerst in einer **If...Then**-Bedingung, ob das aktuell durchlaufene Steuerelement markiert ist. Dazu nutzen wir die Eigenschaft **InSelection**. In der **If...Then**-Bedingung stellen wir die Eigenschaft **Name** des mit **ctl** referenzierten aktuellen Steuerelements auf die gewünschte Bezeichnung ein. Diese ermitteln wir, indem wir den Platzhalter **strPlatzhalter** in **strBezeichnung** mit **Replace** ersetzen, und zwar durch den Wert aus der Variablen **intNummer** mit dem Format aus **strStellen**.

Prüfungen

Damit haben wir den größten Teil bereits erledigt. Bevor wir ein Add-In auf Basis dieser Prozedur anlegen, wollen wir jedoch noch zwei Prüfungen einbauen. Die erste soll untersuchen, ob überhaupt ein Formular in der Entwurfsansicht geöffnet ist. Dazu deklarieren wir eine Variable namens **bolEntwurfsansicht**:

```
Dim bolEntwurfsansicht As Boolean
```

Die Abfrage von Screen.ActiveForm fassen wir in **On Error Resume Next/On Error Goto 0** ein, um die Fehlerbehandlung für diese eine Anweisung zu deaktivieren. Der Grund ist, dass der Aufruf ohne geöffnetes Formular einen Fehler auslöst, den wir vermeiden wollen. Dann prüft die Prozedur, ob **frm** leer ist, was der Fall ist, wenn kein Formular geöffnet ist. Ist **frm** nicht leer, prüfen wir mit der Eigenschaft **CurrentView**, ob die aktuelle Ansicht die Entwurfsansicht ist. Falls ja, stellen wir die Variable **bolEntwurfsansicht** auf **True** ein:

```
...
On Error Resume Next
Set frm = Screen.ActiveForm
On Error Goto 0
```

```
If Not frm Is Nothing Then
    If frm.CurrentView = acCurViewDesign Then
        bolEntwurfsansicht = True
    End If
End If
```

Hat **bolEntwurfsansicht** danach noch den Wert **False** hat, ist das aktive Element entweder kein Formular oder dieses ist nicht in der Entwurfsansicht geöffnet. In diesem Fall gibt die Prozedur eine entsprechende Meldung aus und wird abgebrochen:

```
If bolEntwurfsansicht = False Then
    MsgBox "Es ist kein Formular in der Entwurfsansicht ↵
            geöffnet.", vbOKOnly + vbExclamation, "Kein ↵
                                        Formular geöffnet"
    Exit Sub
End If
```

Außerdem kann es sein, dass wir die gleichen Steuerelemente zu Testzwecken mehrere Male umbenennen oder das diese aus anderen Gründen zuvor bereits solche Namen erhalten haben, wie Sie diese mit der Prozedur zuweisen wollen. Dabei kann es vorkommen, dass Sie einem Steuerelement einen Namen geben wollen, der bereits vergeben ist. Dies würde einen Fehler auslösen. Um dies zu verhindern durchlaufen wir eine ähnliche **For Each**-Schleife wie diejenige, welche die gewünschten Namen zuweist, bereits vorher und vergeben Namen wie **xyz1**, **xyz2** und so weiter für die Steuerelemente, bevor diese ihre endgültigen Namen erhalten. Danach müssen wir **intNummer** erneut mit **intStart** initialisieren:

```
For Each ctl In frm.Controls
    If ctl.InSelection Then
        ctl.Name = "xyz" & intNummer
        intNummer = intNummer + 1
    End If
Next ctl
intNummer = intStart
```

Ungünstige Aktivierreihenfolge

Es kann sein, dass die Aktivierreihenfolge der zu benennenden Steuerelemente nicht der Reihenfolge entspricht, in der die Steuerelemente neben- oder untereinander angeordnet sind. Die **For Each**-Schleife über alle **Control**-Elemente des Formulars durchläuft diese jedoch in der Reihenfolge, die der Aktivierreihenfolge entspricht.

Die Aktivierreihenfolge können Sie jedoch zuvor über den Dialog **Reihenfolge** einstellen, wenn die Aktivierreihenfolge nicht stimmt (siehe Bild 2). Diesen Dialog öffnen Sie über den Ribbon-Befehl **Entwurf|Tools|Eigenschaften**.

Parameter per Formular eingeben

Damit der Benutzer die Parameter bequem eingeben kann, wenn er das Add-In aufgerufen hat, stellen wir ihm ein passendes Formular zur Verfügung. Dieses soll die zuletzt verwendeten Werte außerdem speichern. Dazu erstellen wir zuerst eine Tabelle namens **tblOptionen**. Diese enthält die Felder aus Bild 3.

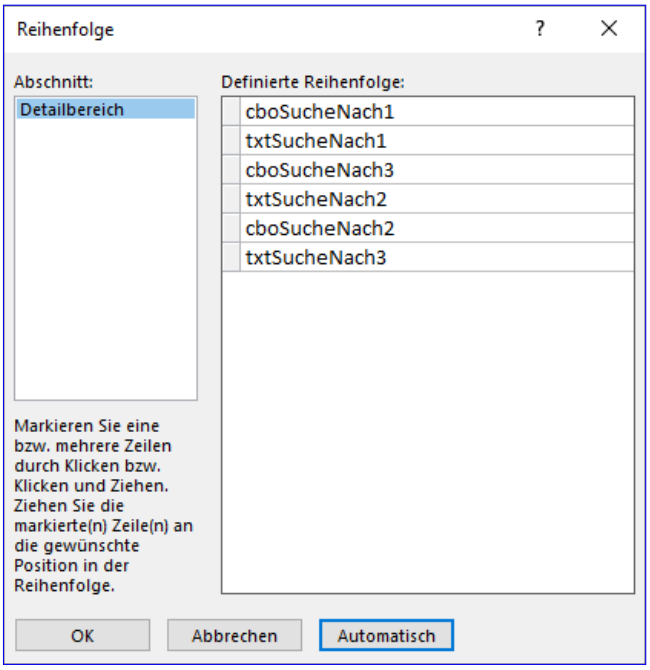


Bild 2: Zu benennende Steuerelemente

ParamArray-Auflistungen in VBA nutzen

Wenn Sie unter VBA eine Prozedur oder eine Funktion definieren, enthält diese immer eine feste Anzahl von Parametern. Diese können Sie auch als optional deklarieren, sodass tatsächlich weniger Werte übergeben werden als Parameter vorhanden sind. Was aber, wenn Sie den Spieß einmal umdrehen und mehr Werte als vorhandene Parameter übergeben wollen – und das auch noch flexibel? Dann kommt die ParamArray-Auflistung ins Spiel. In diesem Beitrag schauen wir uns an, was Sie damit machen können.

Parameter und optionale Parameter

Normalerweise legen Sie für eine Prozedur oder Funktion, nachfolgend als Routine bezeichnet, die Anzahl der Parameter genau fest:

```
Public Sub BeispielParameter(strText As String, _
                             lngZahl As Long)
    Debug.Print strText
    Debug.Print lngZahl
End Sub
```

Der Aufruf erfolgt dann mit der exakten Zahl an Parametern:

```
BeispielParameter "Beispieltext", 123
```

Sie können auch optionale Parameter, die Sie mit dem Schlüsselwort **Optional** versehen und die sich immer am Ende der Parameterliste befinden müssen:

```
Public Sub BeispielOptionaleParameter(_
            strText As String, Optional lngZahl As Long)
    Debug.Print strText
    Debug.Print lngZahl
End Sub
```

Diese Prozedur können Sie so aufrufen:

```
BeispielOptionaleParameter "Beispieltext", 123
```

Sie können den zweiten Parameter aber auch weglassen:

```
BeispielOptionaleParameter "Beispieltext"
```

Die Routine verwendet dann einen Standardwert, den Sie angeben können oder auch nicht. Wenn Sie ihn nicht angeben, wird bei **String**-Variablen eine leere Zeichenkette, bei Zahlen der Wert **0** und bei **Boolean**-Werten **False** angenommen.

Flexible Anzahl an Parametern

Wenn Sie eine flexible Anzahl an Parametern übergeben wollen, wozu brauchen wir dann mehr als optionale Parameter? Davon können wir schließlich so viele definieren, wie wir benötigen:

```
Public Sub VieleParameter(Optional lng1 As Long, _
          Optional lng2 As Long, Optional lng3 As Long, _
          Optional lng4 As Long, Optional lng5 As Long, _
                                 Optional lng6 As Long)
    Debug.Print lng1, lng2, lng3, lng4, lng5, lng6
End Sub
```

Allerdings stimmt das nicht so ganz, denn nach unseren Tests können Sie maximal 60 Parameter für eine Routine definieren – anderenfalls erscheint eine Meldung wie in Bild 1. Und damit kommen wir zum **ParamArray**.

Das Schlüsselwort ParamArray

Sie können in einer Routine genau einen Parameter mit dem Schlüsselwort **ParamArray** ausstatten. Dabei muss es sich zwingend um den letzten Parameter der Routine handeln.

Auflistungszeichen aus Textdateien ersetzen

Wenn Sie die Daten aus Textdateien oder ähnlichen Datenquellen einlesen möchten, benötigen Sie ein Trennzeichen, welches die einzelnen Spalten einer Zeile trennt. Das ist meist durch ein Tabulator-Zeichen, das Semikolon oder das Komma gegeben. Beim Tabulator-Zeichen treten meist keine Probleme auf, aber beim Semikolon oder beim Komma kann es zu folgendem Problem kommen: Die Zeile könnte Spalten enthalten, die ihrerseits das als Trennzeichen verwendete Zeichen enthalten. In diesem Beitrag schauen wir uns an, wie Sie mit solchen Datenquellen unter VBA umgehen können.

Ein ganz einfaches Beispiel für eine solche Zeile ist diese:

```
1; 'Beispielartikel'; 'Dies ist ein Beispielartikel.'
```

Hier können wir einfach nach dem Semikolon trennen und erhalten den Inhalt der drei Spalten. Aber was ist mit der folgenden Zeile?

```
2; 'Noch ein Beispielartikel'; 'Semikola kommen selten vor; manchmal aber schon.'
```

Hier können wir nun nicht mehr einfach nach dem Semikolon trennen. Stattdessen müssten wir untersuchen, ob sich das Semikolon innerhalb eines Paars von einfachen Anführungszeichen befindet – oder auch innerhalb eines Paares von normalen Anführungszeichen.

Wir starten einmal mit einer Prozedur, die nach Schema F vorgeht und so tut, als würde es innerhalb von Spalten keine Delimiter-Zeichen geben. Diese stellt in einem **String**-Array namens **strTestdaten** die beiden oben genannten Zeilen zusammen und durchläuft dieses dann in einer **For...Next**-Schleife über alle Elemente des **String**-Arrays. Darin ruft sie die Prozedur **SpaltenErmitteln** auf, der Sie die jeweilige Zeile übergibt sowie das zu verwendende Delimiter-Zeichen (siehe Listing 1).

Diese Funktion soll ein Array der Spalten der Zeile zurückliefern, die wir dann innerhalb einer weiteren **For...**

```
Public Function Testdaten()
    Dim strTestdaten(2) As String
    Dim i As Integer
    Dim j As Integer
    Dim strSpalten() As String
    strTestdaten(0) = "1; 'Beispielartikel'; 'Dies ist ein Beispielartikel.'"
    strTestdaten(1) = "2; 'Noch ein Beispielartikel'; 'Semikola kommen selten vor; manchmal aber schon.'"
    For i = LBound(strTestdaten) To UBound(strTestdaten)
        strSpalten = SpaltenErmitteln(strTestdaten(i), ";")
        For j = LBound(strSpalten) To UBound(strSpalten)
            Debug.Print j, strSpalten(j)
        Next j
    Next i
End Function
```

Listing 1: Prozedur, die Zeilen an eine Funktion zum Trennen einer Zeile nach dem Trennzeichen übergibt

```
Public Function SpaltenErmitteln(strZeile As String, strDelimiter As String) As Variant
    Dim strSpalten() As String
    strSpalten = Split(strZeile, strDelimiter)
    SpaltenErmitteln = strSpalten
End Function
```

Listing 2: Funktion zum Auftrennen von Zeilen am Delimiter

Next-Schleife, diesmal mit der Zählervariablen **j** versehen, im Direktbereich des VBA-Editors ausgeben.

Die Funktion **SpaltenErmitteln** erwartet die zu untersuchende Zeile sowie das Delimiter-Zeichen. Sie trennt dann die Zeile mit der **Split**-Methode auf, welche die Zeile und den Delimiter erwartet, und ein Array der Elemente zurückliefert (siehe Listing 2).

Das Ergebnis für die obigen beiden Zeilen sieht dann wie folgt aus, wenn wir zusätzlich noch die Spaltennummern ausgeben:

```
0    1
1    'Beispielartikel'
2    'Dies ist ein Beispielartikel.'
0    2
1    'Noch ein Beispielartikel'
2    'Semikola kommen selten vor
3    manchmal aber schon.'
```

Sie sehen hier bereits, dass einmal drei und einmal vier Spalten ermittelt wurden. Das Semikolon innerhalb der Anführungszeichen wurde nämlich als Delimiter erkannt. Das ist keine Basis, um die Daten etwa in eine Tabelle einzutragen.

Also müssen wir einen Weg finden, um nur die Delimiter zu berücksichtigen, die sich nicht innerhalb von Anführungszeichen befinden.

Spalten nur außerhalb von Literalen erkennen

Dazu erweitern wir die Funktion **SpaltenErmitteln** erheblich und gehen innerhalb der Funktion prinzipiell wie folgt vor:

- Wir teilen die Zeile zunächst wieder mit der **Split**-Funktion an allen Stellen, die das Semikolon (oder ein anderes Trennzeichen) enthalten, auf.

- Dann durchlaufen wir alle Elemente des Arrays und schauen, ob die jeweils aktuelle Zeile eine ungerade Anzahl von Anführungszeichen enthält. In diesem Fall setzen wir einen Marker, der beim nächsten Durchlauf dafür sorgt, dass die folgende Zeile an die zuvor untersuchte angehängt wird. Das wiederholen wir solange, bis wir nochmal in einer Zeile eine ungerade Anzahl von Anführungszeichen vorfinden. Dann schließen wir die Zeile ab und ändern den Wert dieses Markers.

Im Detail sieht das wie in Listing 3 aus. Die Funktion **SpaltenErmitteln** nimmt nun noch einen weiteren Parameter entgegen, mit dem Sie festlegen, welches Zeichen als Anführungszeichen verwendet wird – zum Beispiel das Hochkomma oder das normale Anführungszeichen.

Dann deklarieren wir zwei Arrays. Das erste nimmt das Ergebnis der ersten **Split**-Anweisung entgegen, das einfach nur eine Unterteilung der Zeile an allen Vorkommen des Delimiters vornimmt. Die zweite soll dann die tatsächlichen Zeilen aufnehmen:

```
Dim strSpalten() As String
Dim strSpaltenBereinigt() As String
```

i ist die Laufvariable zum Durchlaufen der Elemente des Arrays **strSpalten**, **intAnzahlSpalten** ist die Anzahl der Spalten zum Redimsionieren des im zweiten Schritt zusammengestellten Arrays, **intAnzahlAnfuehrungszeichen** nimmt die Anzahl der Anführungszeichen innerhalb eines Element des ersten Arrays auf und **bolAnfueh-**

Pivot-Tabellen und -Diagramme in Excel

Microsoft hat die Pivot-Tabellen und -Diagramme mit der Version 2010 aus Access herausgenommen. Damit ist Excel das einzige Produkt in der Office-Familie, dass diese praktische Darstellungsweise von Daten beherrscht. Schauen wir uns also an, wie wir diese nutzen können. Im vorliegenden Beitrag widmen wir uns den Grundlagen von Pivot-Tabellen und -Diagrammen in Excel. In weiteren Beiträgen schauen wir uns dann an, wie wir diese von Access aus füllen und nutzen können, sodass der Benutzer möglichst wenige zusätzliche Handgriffe ausführen muss.

Wozu Pivot-Tabellen oder -Diagramme?

Manche Datenmengen sind so groß, dass es unmöglich ist, durch diese Ergebnisse Rückschlüsse zu ziehen. In diesem Fall muss man die Daten irgendwie anders darstellen oder zusammenfassen, um diese sinnvoll auswerten zu können. Unter Access kennen wir dazu Abfragen. Allerdings bedingen Abfragen, dass der Benutzer sich mit dem Entwurf von Abfragen auskennt. Das ist trotz der Entwurfsansicht von Abfragen von Access, welche die dahinterliegende Abfragesprache SQL maskiert, nicht trivial. Einfacher geht es mit sogenannten Pivot-Tabellen. Diese erlauben es, die Daten des gewünschten Bereichs einer Tabelle in verschiedenen Konstellationen darzustellen, diese zu gruppieren und Berechnungen anzustellen.

Beispieldaten

Als Beispieldaten nutzen wir einige Daten der Südsturm-Datenbank, die wir ein einer Abfrage zusammengefasst haben. Diese sieht wie in aus und enthält Daten aus den Tabellen **tblKunden**, **tblBestellungen**, **tblBestelldetails**, **tblArtikel**, **tblKategorien** und **tblLieferanten** (siehe Bild 1). So haben wir ausreichend Spielmaterial für die ersten Gehversuche mit Pivot-Tabellen und -Diagrammen unter Excel.

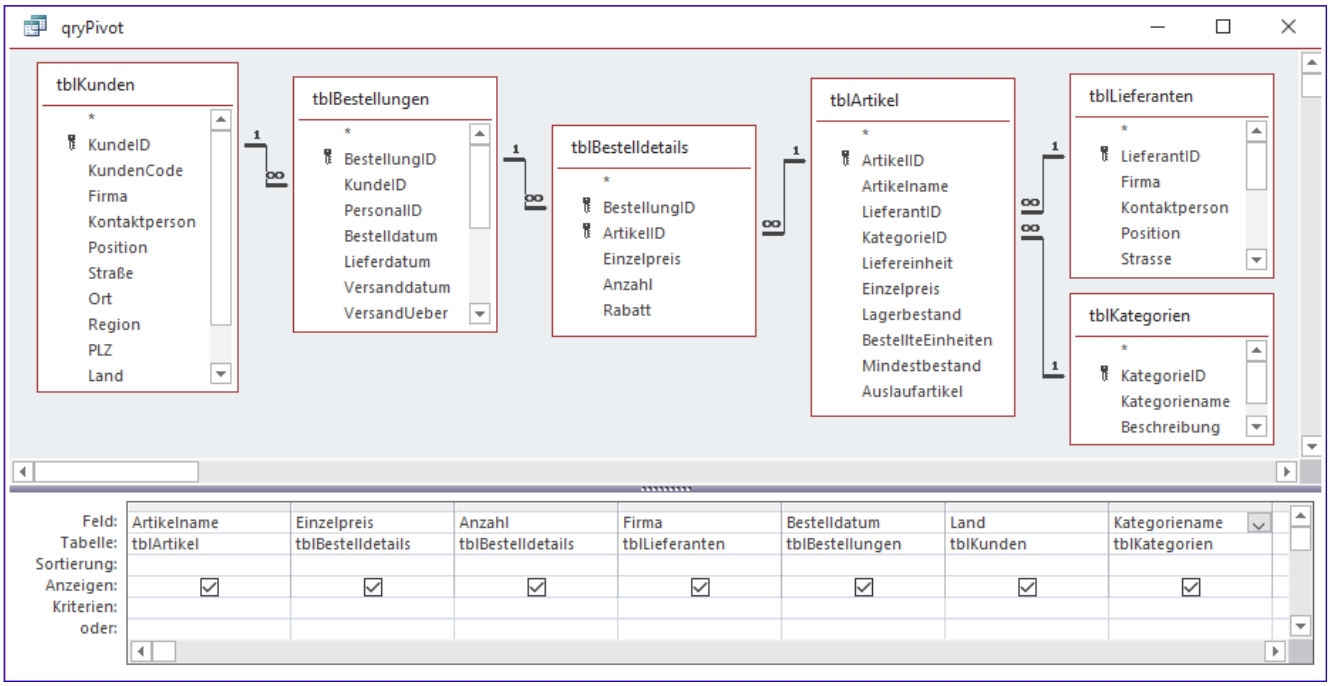


Bild 1: Datenquelle für die Pivot-Auswertungen

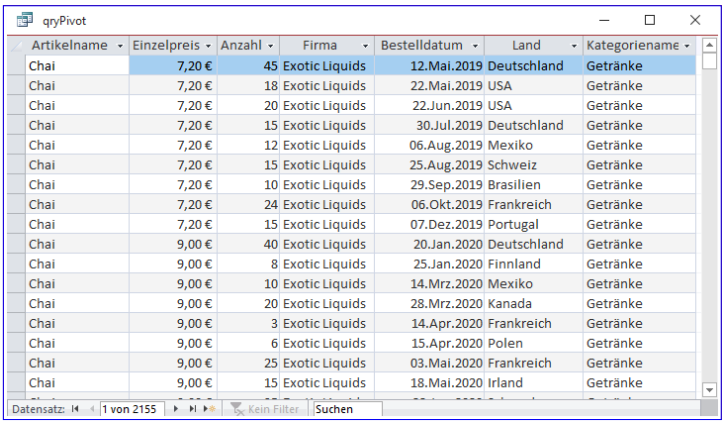


Bild 2: Daten für die Pivot-Auswertungen

Die Daten können wir nach Artikel, Lieferant, Bestelldatum und Zielland gruppieren, und mit über 2.000 Datensätzen sollte auch die Datenmenge ausreichend sein (siehe Bild 2).

Beispieldaten exportieren

Die Daten exportieren wir nun im Excel-Format. Dazu markieren Sie die unter dem Namen **qryPivot** gespeicherte Abfrage im Navigationsbereich und wählen den Kontextmenü-Eintrag **Exportieren|Excel** aus (siehe Bild 3).

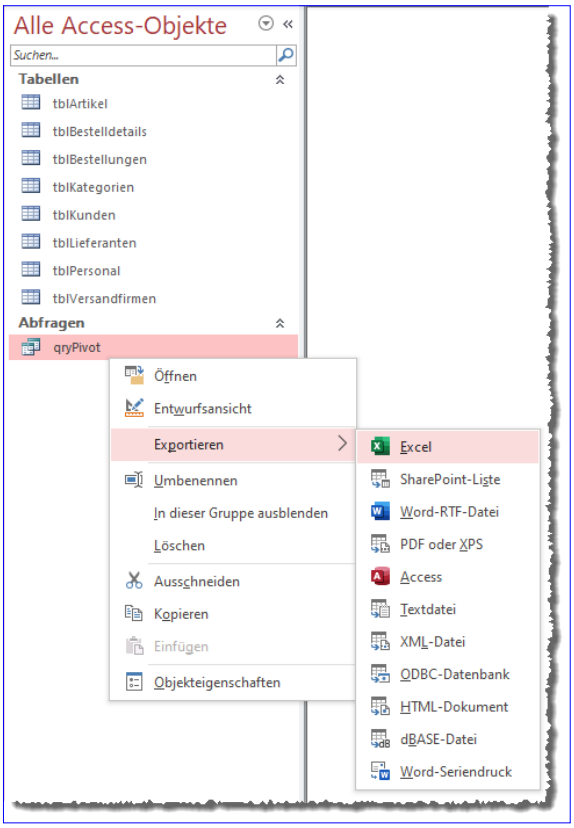


Bild 3: Exportieren der Daten in eine Excel-Datei

Im ersten Schritt des Export-Assistenten geben Sie den Namen der Zieldatei an. Danach klicken Sie einfach auf **OK**, wir benötigen keinen Export mit Layout oder dergleichen. Die neu erstellte Excel-Datei enthält dann ein Tabellenblatt mit den Daten (siehe Bild 4). Damit können wir die Access-Datenbank vorerst schließen.

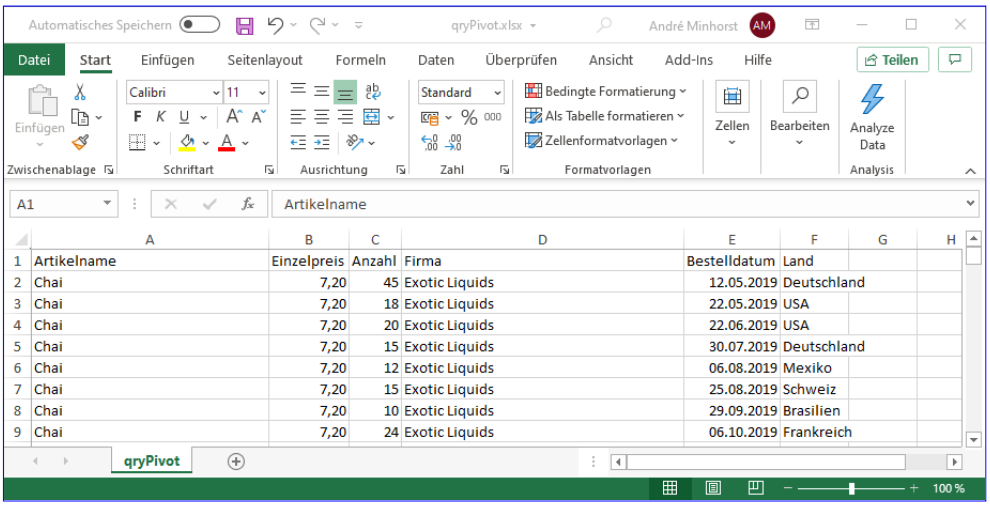


Bild 4: Unser Beispielmaterial für das Erstellen von Pivot-Tabellen

Erste Pivot-Schritte

Nach dem Öffnen der Excel-Datei wollen wir gleich eine erste Pivot-Tabelle erstellen. Unter Excel wechseln Sie dazu im Ribbon zum Bereich **Einfügen**. Hier wählen wir den Eintrag **Tabellen|PivotTable** (siehe Bild 5).

Dies öffnet den Dialog aus Bild 6. Hier wählen Sie im oberen Bereich fest, aus welchem Bereich die Daten stammen, die mit der Pivot-

Tabelle ausgewertet werden sollen. In unserem Fall ist die Auswahl bereits automatisch erfolgt und umfasst alle Zeilen und Spalten inklusive Spaltenüberschriften. Wir können hier auch eine alternative Datenquelle verwenden, also beispielsweise direkt auf eine Access-Abfrage zugreifen. Wie das gelingt, schauen wir uns später an.

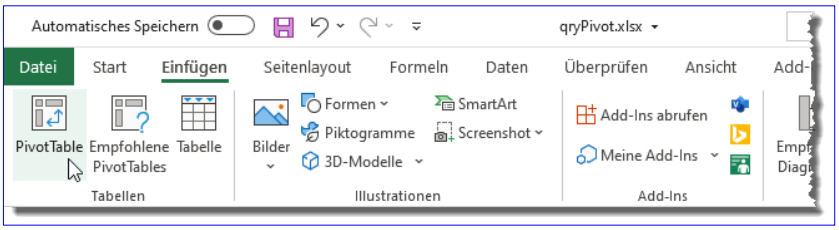


Bild 5: Erstellen einer Pivot-Tabelle per Ribbon-Befehl

Außerdem geben wir in diesem Dialog an, wo die neue Pivot-Tabelle platziert werden soll. Dazu können Sie ein neues Arbeitsblatt nutzen, aber Sie können diese auch direkt auf dem Arbeitsplatt mit den Daten platzieren.

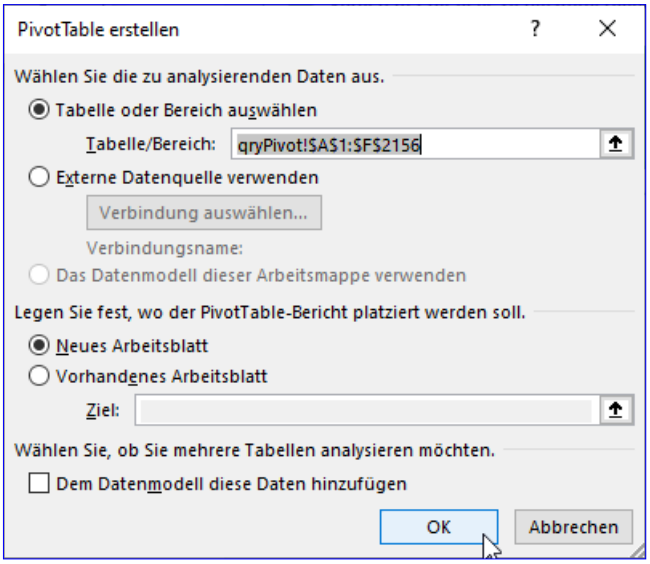


Bild 6: Dialog zum Erstellen einer Pivot-Tabelle

Nach einem Klick auf **OK** erscheint das neue Arbeitsblatt, das zwei wesentliche Bereiche anzeigt – auf der linken Seite den Platzhalter für die zu erzeugende Pivot-Tabelle und auf der rechten die Feldliste, mit der Sie die Felder der Datenquelle den verschiedenen Aufgaben zuweisen können (siehe Bild 7). Die vier Bereiche lauten **Filter**, **Spalten**, **Zeilen** und **Werte**.

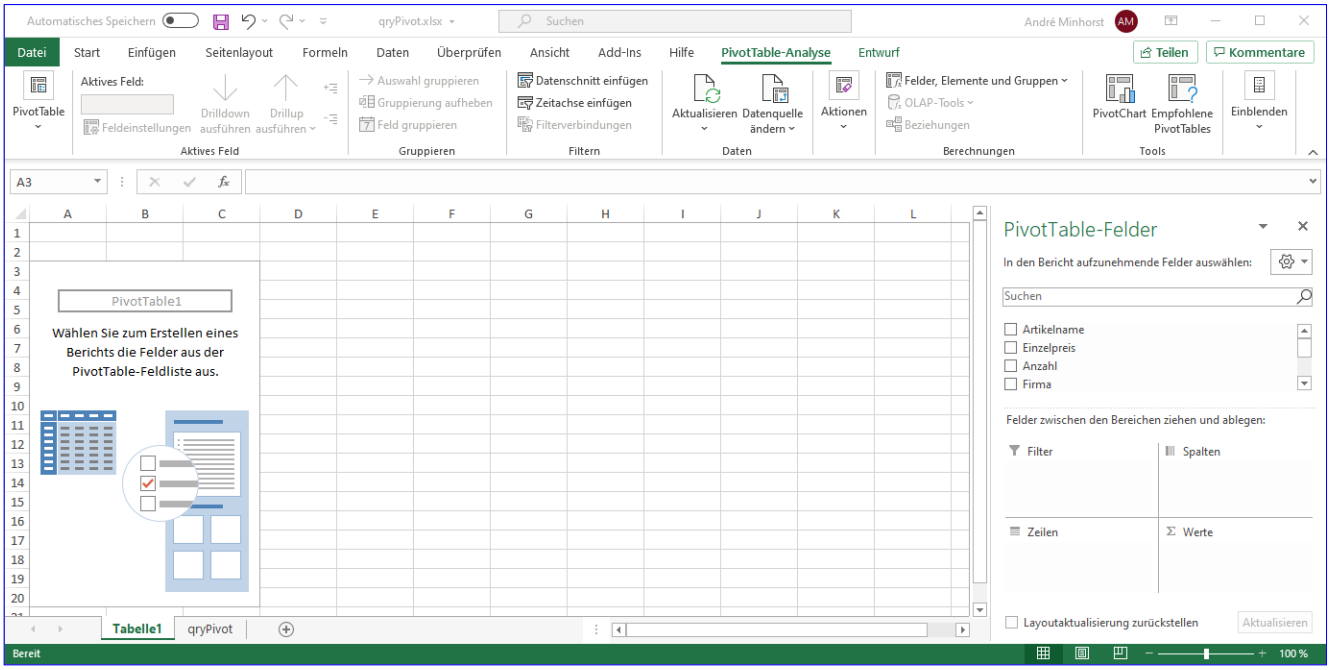


Bild 7: Ausgangspunkt zum Einrichten der Pivot-Tabelle

Zuweisen der Felder zur Pivot-Tabelle

Nun folgt der interessante Teil: Das Zuordnen der Felder aus der Feldliste zu den vier Bereichen. Dies können Sie auf zwei Arten erledigen:

- Durch Setzen von Haken an die hinzuzufügenden Felder aus der Feldliste. Dies fügt die Felder automatisch den Bereichen zu. Damit geben Sie die Entscheidung, welche Felder in welchem Bereich landen, in die Hand von Excel.

- Per Drag and Drop der Felder in jeweils einen der vier Bereiche.

Welches Feld Sie in welchen Bereich ziehen, hängt von Ihren Anforderungen ab. Wir wollen im ersten Schritt einmal ermitteln, welche Artikel in welche Länder verkauft wurden. Dazu ziehen wir das Feld **Land** in den Bereich **Spalten**, das Feld **Artikelname** in den Bereich **Zeilen** und das Feld **Anzahl** in den Bereich **Werte** (siehe Bild 8).

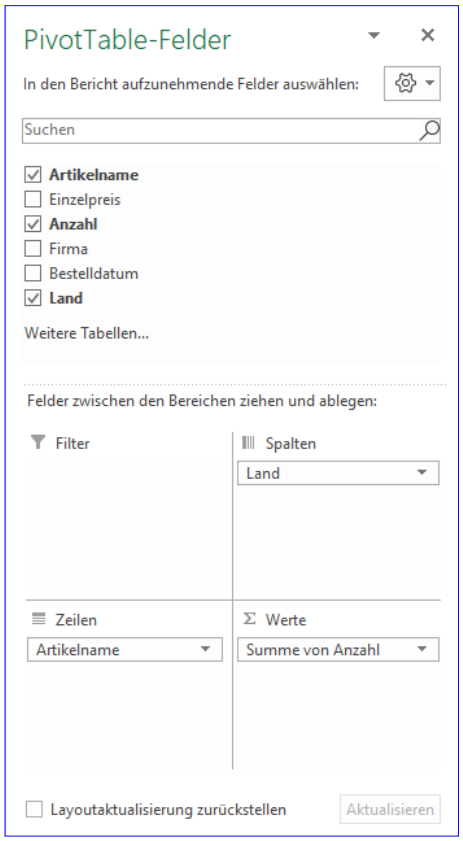
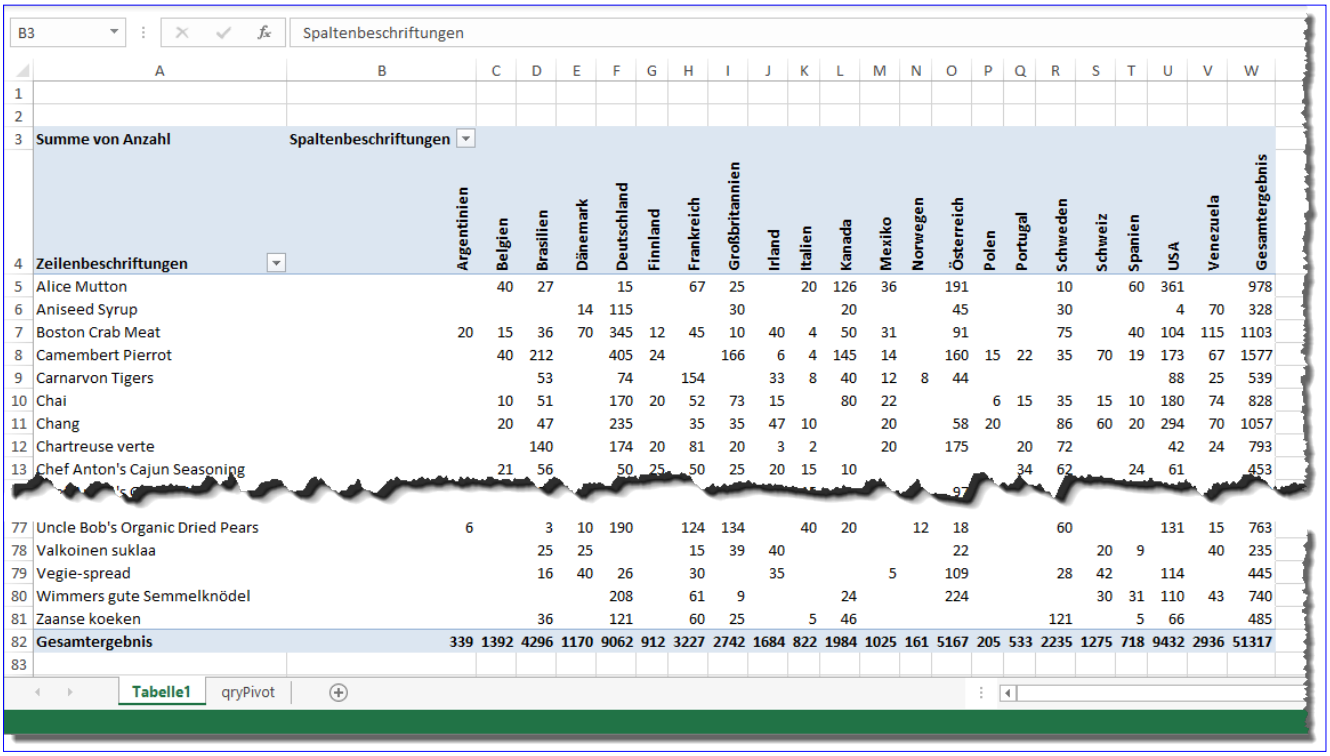


Bild 8: Zuordnen der Felder zu den Bereichen

Damit erhalten wir das Ergebnis aus Bild 9.

Wir haben hier noch eine kleine Änderung vorgenommen: Wir haben die Spaltenüberschriften vertikal angeordnet. Dazu klicken Sie mit der rechten Maustaste auf alle anzupassenden Felder, hier die mit den Spaltenüberschriften, und wählen den Kontextmenü-Eintrag **Zellen formatieren** aus. Im nun erscheinenden Dialog **Zellen formatieren** wechseln Sie zur Registerseite



B3 | Spaltenbeschriftungen

Summe von Anzahl	Spaltenbeschriftungen																						
Zeilenbeschriftungen	Argentinien	Belgien	Brasilien	Dänemark	Deutschland	Finnland	Frankreich	Großbritannien	Irland	Italien	Kanada	Mexiko	Norwegen	Österreich	Polen	Portugal	Schweden	Schweiz	Spanien	USA	Venezuela	Gesamtergebnis
Alice Mutton		40	27		15		67	25		20	126	36		191			10		60	361		978
Aniseed Syrup				14	115			30			20			45			30			4	70	328
Boston Crab Meat	20	15	36	70	345	12	45	10	40	4	50	31		91			75		40	104	115	1103
Camembert Pierrot		40	212		405	24		166	6	4	145	14		160	15	22	35	70	19	173	67	1577
Carnarvon Tigers			53		74		154		33	8	40	12	8	44						88	25	539
Chai		10	51		170	20	52	73	15		80	22			6	15	35	15	10	180	74	828
Chang		20	47		235		35	35	47	10		20		58	20		86	60	20	294	70	1057
Chartreuse verte			140		174	20	81	20	3	2		20		175		20	72			42	24	793
Chef Anton's Cajun Seasoning		21	56		50	25	50	25	20	15	10					34	62		24	61		453
...														9								
Uncle Bob's Organic Dried Pears	6		3	10	190		124	134		40	20		12	18			60			131	15	763
Valkoinen suklaa			25	25			15	39	40					22				20	9		40	235
Vegie-spread			16	40	26		30		35			5		109			28	42		114		445
Wimmers gute Semmelknödel					208		61	9			24			224				30	31	110	43	740
Zaanse koeken			36		121		60	25		5	46						121		5	66		485
Gesamtergebnis	339	1392	4296	1170	9062	912	3227	2742	1684	822	1984	1025	161	5167	205	533	2235	1275	718	9432	2936	51317

Tabelle1 | qryPivot

Bild 9: Ergebnis der ersten Pivot-Tabelle

Ausrichtung und stellen diese auf **90°** ein (siehe Bild 10).

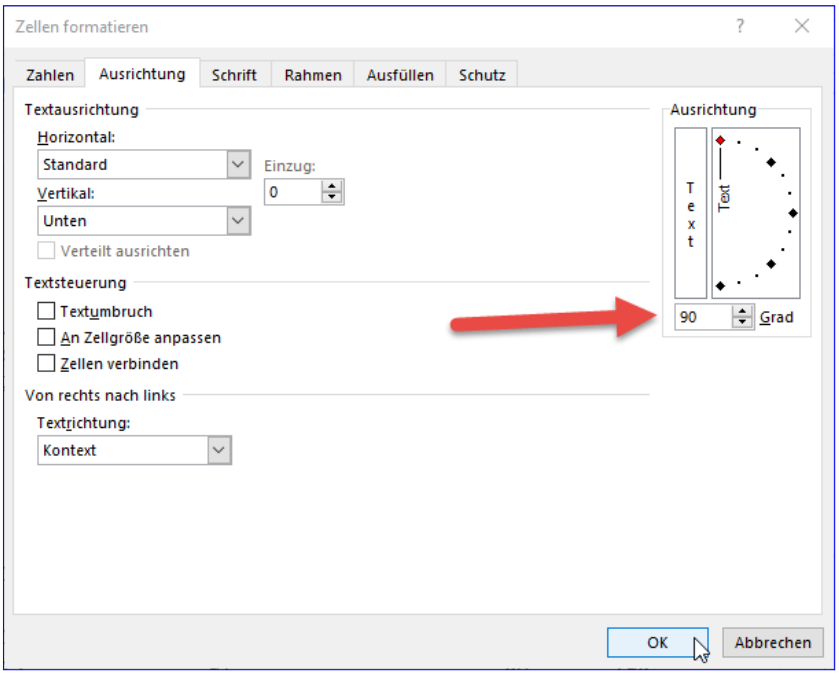


Bild 10: Vertikale Ausrichtung der Spaltenüberschriften

Das Ergebnis ist schon recht interessant: Wir sehen die Menge der verkauften Artikel je Land, die verkaufte Gesamtmenge eines Artikels sowie die Gesamtmenge aller Artikel je Land.

Filtern nach Zeilen oder Spalten

Die nun in den Zeilen und Spalten befindlichen Überschriften können Sie filtern, das heißt, Sie können zum Beispiel nur die Bestellungen aus europäischen Ländern anzeigen lassen. Dazu klicken Sie neben dem Text **Spaltenbeschriftungen** auf die Schaltfläche mit dem nach unten zeigenden Dreieck und selektieren die anzuzeigenden Einträge beziehungsweise wählen die nicht mehr gewünschten Einträge ab (siehe Bild 11). Nach einem Klick auf die Schaltfläche **OK** zeigt die Pivot-Tabelle direkt die resultierenden Werte an. Außerdem ändert sich das Symbol vom nach unten zeigenden Dreieck der Schaltfläche in ein **Filter**-Symbol. Entsprechend der Änderungen wird auch das Gesamtergebnis in der Spalte ganz rechts neu berechnet.

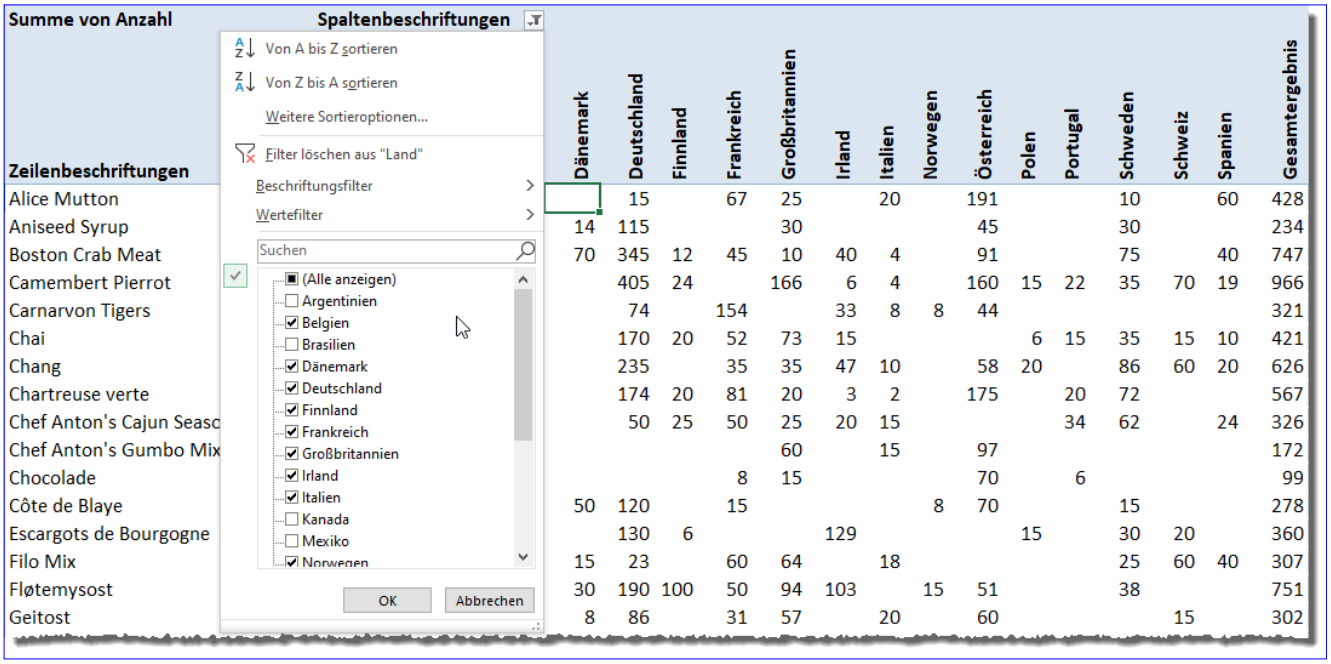


Bild 11: Vertikale Ausrichtung der Spaltenüberschriften

Pivot-Tabellen und -Charts automatisch erstellen

Da wir in Access seit Version 2013 keine eingebaute Anzeige von Pivot-Tabellen oder -Charts haben, müssen wir uns mit Excel behelfen. Die Grundlagen dazu finden Sie im Beitrag »Pivot-Tabellen und -Diagramme in Excel«. Nun schauen wir uns an, wie wir von Access aus per Schaltfläche ein mit den aktuellen Daten einer dafür vorgesehenen Abfrage nach Excel exportieren und dieser Excel-Datei eine Pivot-Tabelle hinzufügen können.

Ziel dieses Beitrags

Im Beitrag **Pivot-Tabellen und -Diagramme in Excel** (**www.access-im-unternehmen.de/****) haben wir uns angesehen, wie Sie die Daten einer Access-Abfrage manuell in eine Excel-Datei exportieren und dieser dann in einem neuen Arbeitsblatt eine Pivot-Tabelle hinzufügen. Diesen Vorgang wollen wir nun automatisieren. Das heißt, dass der Benutzer nur eine Schaltfläche anklicken soll und dann alle anderen Schritte automatisch ablaufen – bis zur Anzeige der Seite der Excel-Datei mit der Pivot-Tabelle. Das ist eine Möglichkeit, Kunden die gewünschte Auswertung anzeigen zu lassen. Weitergehende Anpassungen kann der Kunde dann in der Excel-Datei selbst vornehmen.

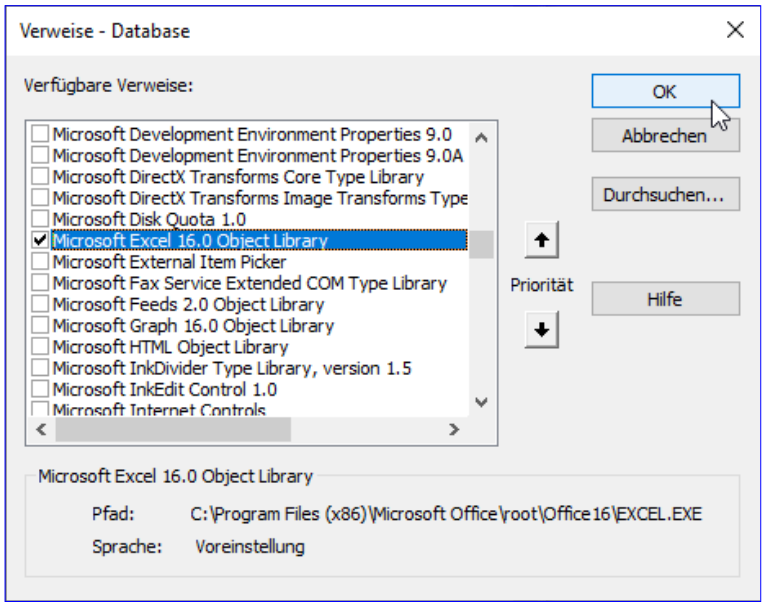


Bild 1: Verweis auf die **Excel**-Objektbibliothek

Excel automatisieren

Die Automatisierung von Excel ist wesentlich einfacher, wenn wir einen Verweis zur Objektbibliothek von Excel herstellen. Wir können dann zum Beispiel im VBA-Editor IntelliSense nutzen. Den Verweis fügen Sie hinzu, indem Sie im VBA-Editor (**Alt + F11**) den Menüpunkt **Extras|Verweise** betätigen und dort den Eintrag **Microsoft Excel 16.0 Object Library** hinzufügen (siehe Bild 1).

Daten exportieren

Das Exportieren der Daten für die gewünschte Tabelle oder Abfrage erledigen wir mit der Funktion **DatenExportieren** (siehe Listing 1). Diese erwartet zwei Parameter:

- **strQuelle**: Name der zu exportierenden Tabelle oder Abfrage
- **strDatei**: Pfad zu der zu erstellenden Datei

Die Funktion ruft bei deaktivierter Fehlerbehandlung die Anweisung **Kill** auf. Diese löscht eine eventuell bereits vorhandene Datei mit dem Namen aus **strDatei**. Die Fehlerbehandlung wird deaktiviert, weil **Kill** einen Fehler auslöst, wenn die angegebene Datei nicht vorhanden ist. Danach verwendet die Funktion die Methode **DoCmd.TransferSpreadsheet**, um die Daten aus der mit **strQuelle** angegebenen Tabelle oder Abfrage in die Datei aus **strDatei** zu exportieren.

```
Private Sub DatenExportieren(strQuelle As String, strDatei As String)
    On Error Resume Next
    Kill strDatei
    On Error GoTo 0
    DoCmd.TransferSpreadsheet acExport, acSpreadsheetTypeExcel12Xml, strQuelle, strDatei, True
End Sub
```

Listing 1: Exportieren der Datenquelle in eine Excel-Datei

Als Format verwenden wir **acSpreadsheetTypeExcel12Xml**. Die Dateiendung muss in diesem Fall **.xlsx** lauten.

Excel starten

Um die Excel-Instanz zu starten, in der wir die Pivot-Tabelle erzeugen wollen, nutzen wir eine weitere kleine Prozedur. Diese heißt **ExcelErzeugen** und sieht wie folgt aus:

```
Private Sub ExcelErzeugen()
    Set objExcel = New Excel.Application
    objExcel.Visible = True
End Sub
```

Für die hier erzeugte Excel-Instanz fügen wir im Klassenmodul des Formulars, aus dem wir die Funktionen aufrufen wollen, eine passende Objektvariable hinzu:

```
Dim objExcel As Excel.Application
```

```
Private Sub cmdPivotTabelleInExcelErstellen_Click()
    Dim objWorkbook As Excel.Workbook
    Dim strDatei As String
    Dim strQuelle As String
    strQuelle = "qryPivot"
    strDatei = CurrentProject.Path & "\qryPivot.xlsx"
    DatenExportieren strQuelle, strDatei
    ExcelErzeugen
    Set objWorkbook = WorkbookOeffnen(strDatei)
    '... weitere Anweisungen
End Sub
```

Listing 2: Exportieren und Öffnen der Daten

Um das soeben erstellte Workbook zu öffnen, verwenden wir diese Funktion:

```
Private Function WorkbookOeffnen(strDatei As String) _
                                As Excel.Workbook
    Set WorkbookOeffnen = objExcel.Workbooks.Open(strDatei)
End Function
```

Dann kommen wir endlich zur Prozedur, die wir durch das Ereignis **Beim Klicken** einer Schaltfläche namens **cmdPivotTabelleInExcelErstellen** auslösen.

Diese deklariert eine Variable für das **WorkBook**-Objekt sowie für den Namen der Quelle und der zu erstellenden Datei und füllt diese noch mit Werten.

Diese Werte können Sie auch über entsprechende Steuerelemente im Formular durch den Benutzer füllen lassen.

Solange wir eine Pivot-Tabelle für eine spezielle Datenquelle erstellen, macht es aber keinen Sinn, eine andere als die dafür vorgesehene Datenquelle anzugeben.

Danach ruft die Prozedur nacheinander die bereits vorgestellten Routinen **DatenExportieren**, **ExcelErzeugen** und **objWorkbook** auf. Das Ergebnis ist eine geöffnete Excel-Instanz, welche die exportierten Daten in einem Worksheet anzeigt (siehe Bild 2).

Nun benötigen wir noch ein zweites Worksheet, in dem wir die Pivot-Tabelle anlegen.

Statische Workflows im Griff

Es gibt immer wieder Standardabläufe im Büroalltag. Bei mir ist es das Schreiben von Artikeln. Für jeden Artikel gibt es einige Aufgaben, die immer in der gleichen Reihenfolge ablaufen. Wer denkt, das geht nach 20 Jahren automatisch, irrt sich – immer wieder bleibt mal eine Teilaufgabe liegen oder wird nicht in der richtigen Reihenfolge erledigt. Um das zu ändern, erstellen wir in diesem Beitrag eine Lösung, um wiederkehrende, nach dem gleichen Schema ablaufende Aufgaben zu verfolgen. Dazu speichern wir die einzelnen Schritte als Felder in einer Tabelle. Über die Benutzeroberfläche stellen wir die Aufgaben und die Teilschritte optisch so dar, dass die nun zu erledigen Teilaufgaben immer im Blickfeld sind!

Beispiel für einen Workflow

Wie sieht ein beispielhafter Workflow aus? Ich wähle gleich den aus meiner täglichen Arbeit – das Erstellen eines Beitrags etwa für das vorliegende Magazin. Hier tauchen die folgenden Aufgaben auf:

- Beitrag in Beitragsdatenbank aufnehmen
- Beitrag schreiben
- Beitrag setzen
- Beitrag zum Lektor schicken
- Beitrag wird durch Lektor geprüft
- Korrekturvorschläge des Lektors einarbeiten
- Beitrag in Beitragsdatenbank einlesen
- Beispieldateien in Beitragsdatenbank aufnehmen
- Beitrag und Beispieldateien online stellen

Zusätzlich gibt es je Magazin noch eine Reihe weiterer Aufgaben:

- PDF erstellen
- PDF zum Verlag schicken
- Verlag macht letzte Korrekturvorschläge
- Korrekturvorschläge des Verlags einarbeiten
- Magazin online stellen
- Downloadseite um neue Ausgabe erweitern
- Neuen Benutzer für die neuen Zugangsdaten anlegen
- Newsletter an Abonnenten schicken

Das sind bereits einige Aufgaben, die im Alltag noch etwas detaillierter ausfallen. Sie sollen jedoch reichen, um Beispielmaterial für die zu erstellende Lösung zu liefern.

In der Praxis gibt es für jeden Beitrag einen neuen Workflow und auch für jede neue Ausgabe. Wir könnten noch einen Schritt weitergehen und die Workflows einander unterordnen, also beispielsweise die Beitrags-Workflows den Magazin-Workflows zuweisen. Dies wollen wir aus Gründen der Übersicht in diesem Beitrag nicht erledigen.

Einfachste Methode

Wenn Sie immer wiederkehrende Unteraufgaben haben, können Sie einfach für jede Aufgabe ein Kontrollkästchen

zur Aufgabe hinzufügen und diese nach Erledigung der Aufgaben abhaken. Damit wissen Sie jederzeit, welchen Stand die Aufgabe hat. Wenn die Unteraufgaben sich nicht verändern, ist das auch die beste und einfachste Methode.

Wenn wir bei meinem Beispiel bleiben, wo sich die Unteraufgaben beim Schreiben eines Beitrags nie ändern, können Sie einfach für jede Unteraufgabe ein **Ja/Nein**-Feld zu der Tabelle hinzufügen, in der Sie auch weitere Daten zu den Beiträgen speichern – wie den Titel, den Inhalt et cetera.

Der Entwurf der Tabelle **tblBeitraege** sieht danach etwa wie in Bild 1 aus.

Danach können Sie die Felder wie in Bild 2 in einem Formular zur Bearbeitung der Beiträge unterbringen. Damit erhalten Sie einen detaillierten Überblick über alle Daten zu einem Beitrag.

Noch praktischer für einen Überblick über alle Aufgaben ist eine Übersicht der Aufgaben mehrerer Beiträge in einem Endlosformular oder in einem Datenblatt. Gegebenenfalls können Sie auch ein Listenfeld nutzen.

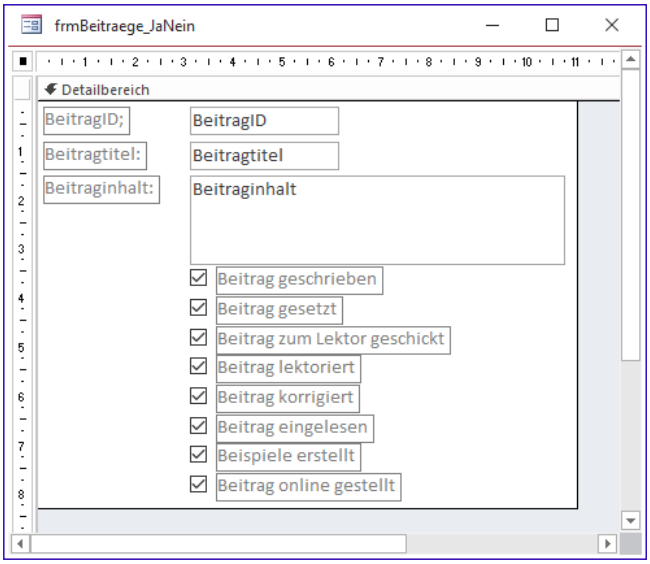


Bild 2: Formularentwurf mit Ja/Nein-Feldern

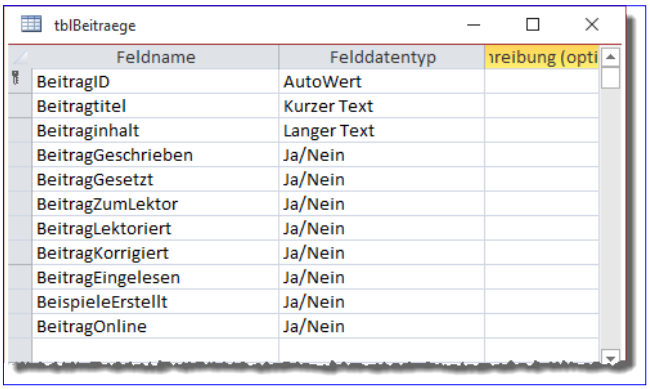


Bild 1: Einfache Variante zum Einbau eines Workflows

Bessere Reproduzierbarkeit mit Datum

Damit sehen wir nun, wie weit jeder Beitrag ist und welche Aufgaben noch zu erledigen sind. Aber was, wenn der Lektor fragt, wo der Beitrag bleibt, sie diesen aber schon als verschickt gekennzeichnet haben? Dann müssen Sie gegebenenfalls den kompletten Ordner der gesendeten Elemente in Outlook durchsuchen.

Etwas einfacher geht es, wenn Sie zumindest wissen, wann Sie den Beitrag an den Lektor verschickt haben. Zu diesem Zweck ändern wir den Datentyp der **Ja/Nein**-Felder der Tabelle **tblBeitraege** in **Datum/Zeit** um (siehe Bild 3).

Allerdings wollen wir diese Information nur abrufen, wenn unbedingt nötig. In einer Übersicht hingegen sollen nur Kontrollkästchen angezeigt werden, die angeben, ob die Unteraufgabe erledigt wurde oder nicht. In der Detailan-

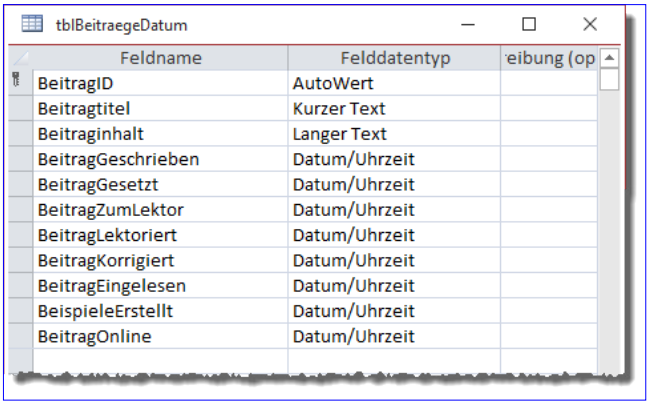


Bild 3: Tabelle mit Datum/Uhrzeit-Feldern

sicht wollen wir auch nur Kontrollkästchen anzeigen, die der Benutzer nach Erledigung einer Teilaufgabe abhaken kann. Nach dem Abhaken soll dann das Datum der Erledigung in das betroffene Feld eingetragen werden.

Zusammenfassend: Die Tabelle soll ein Datumsfeld für das Erledigungsdatum einer jeden Unteraufgabe enthalten. Die Formulare hingegen sollen ein Kontrollkästchen anzeigen. Das Kontrollkästchen soll angehakt sein, wenn das Datumsfeld nicht leer ist, wenn es leer ist, soll das Kontrollkästchen keinen Haken enthalten. Außerdem soll, wenn der Benutzer einen Haken in ein Kontrollkästchen setzt, das Erledigungsdatum in das zugrunde liegende **Datum/Zeit**-Feld eingetragen werden.

Wir haben also ein **Datum/Zeit**-Feld, dessen Status **Null**/**Nicht Null** in Form des Wertes eines Kontrollkästchens erscheinen soll. Dazu benötigen wir eine technische Lösung, die zum Beispiel in einer entsprechend formulierten Abfrage liegen könnte. Diese würde für jedes Unteraufgaben-Feld zwei Felder enthalten – einmal das **Datum/Zeit**-Feld und ein weiteres Feld, das prüft, ob das **Datum/Zeit**-Feld bereits einen Wert enthält oder nicht und abhängig davon den Wert **True** oder **False** ausgibt.

Diese Abfrage formulieren wir wie in Bild 4. Sie enthält das Feld **BeitragGeschrieben** und ein berechnetes Feld namens **BeitragGeschriebenJaNein**, das den Wert **-1** liefert, wenn das Feld **BeitragGeschrieben** gefüllt ist:

```
BeitragGeschriebenJaNein: Nicht
IstNull([BeitragGeschrieben])
```

Bild 5 zeigt, wie das Ergebnis der Abfrage für zwei Datensätze aussieht, von denen einer einen Datumswert im Feld **BeitragGeschrieben** aufweist und der andere den Wert **Null**.

Wir würden nun gern noch das berechnete Feld **BeitragGeschriebenJaNein** als Kontrollkästchen in der Datenblattansicht anzeigen. Dazu verwenden wir normalerweise die Eigenschaft **Steuerelement anzeigen** im Bereich

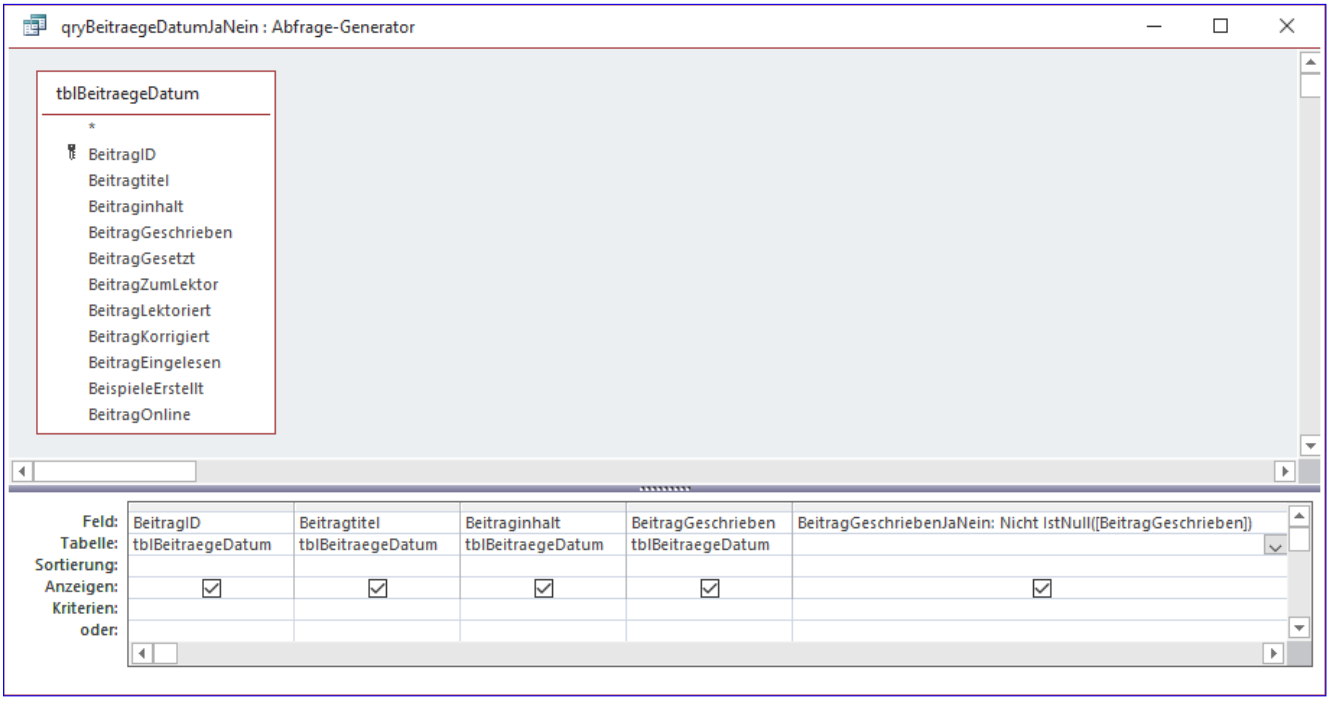


Bild 4: Abfrage, die sowohl das **Datum/Zeit**-Feld als auch ein davon abhängiges **Ja/Nein**-Feld liefert.

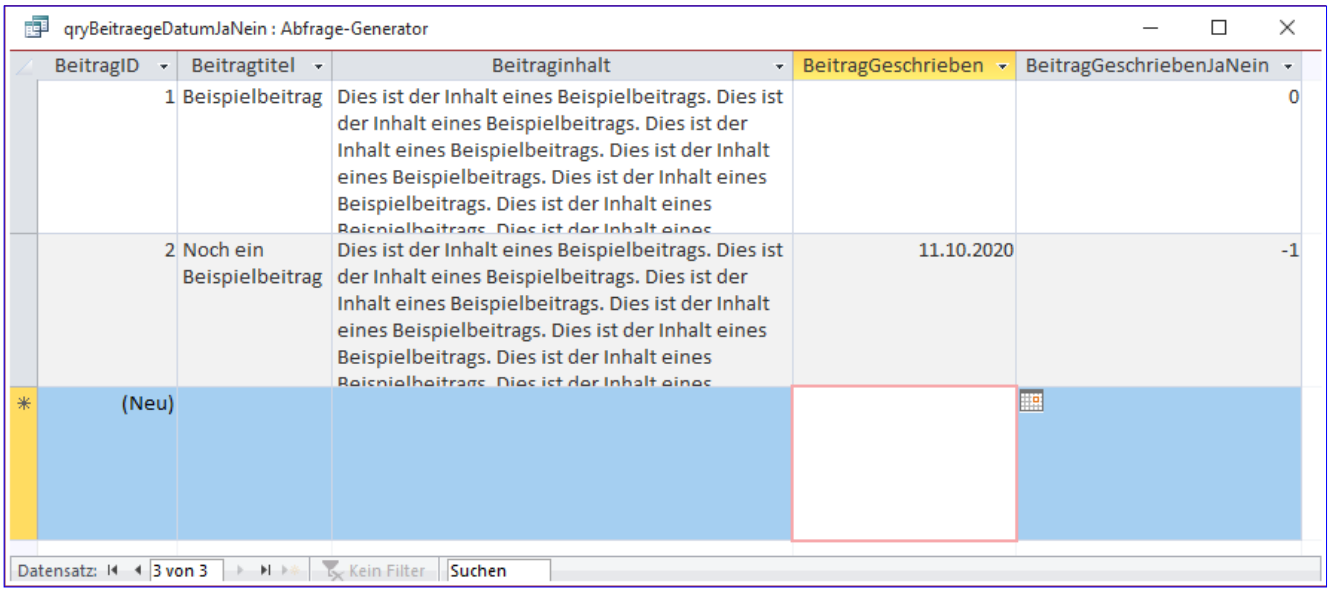


Bild 5: Ergebnis der Abfrage mit je einem gefüllten und einem nicht gefüllten **BeitragGeschrieben**-Feld

Nachschlagen der Eigenschaften einer Spalte der Abfrage. Hier finden wir allerdings nur die drei Möglichkeiten Textfeld, Listenfeld und Kombinationsfeld vor (siehe Bild 6). Anders ist es, wenn wir ein Abfragefeld auf Basis eines **Ja/Nein**-Feldes erstellen – dann stehen Kontrollkästchen, Textfeld und Kombinationsfeld zur Verfügung. Da dies hier nicht der Fall ist, müssen wir in den sauren Apfel beißen und vorerst mit Textfeldern arbeiten.

Bild 6: Die Anzeige als Kontrollkästchen ist nicht möglich.

Der Nachteil davon zeigt sich, wenn wir das nun in einen Formular abbilden wollen – wir hätten dort gern für die berechneten **Ja/Nein**-Felder gern direkt Kontrollkästchen erhalten, aber beim Hinzufügen im Formularentwurf übernimmt Access den in der Tabelle beziehungsweise Abfrage voreingestellten Steuerelementtyp.

Wenn wir die Felder aus der Abfrage, welche die **Ja/Nein**-Ausdrücke für die Datumsfelder berechnen, in das Formular ziehen und versuchen, diese in Kontrollkästchen umzuwandeln, finden wir auch hier diese Möglichkeit nicht vor (siehe Bild 7).

Also müssen wir wohl oder übel jeweils ein Kontrollkästchen für jedes Feld in den Formularentwurf ziehen und dann der Eigenschaft **Steuerelementinhalt** die berechneten **Ja/Nein**-Felder der als Datensatzquelle verwendeten Abfrage zuweisen. Außerdem benennen wir die Kontrollkästchen auch direkt mit dem passenden Präfix, also beispielsweise **chkBeitragGeschrieben**.

Für das berechnete Feld **BeitragGeschriebenJaNein** sieht das Kontrollkästchen wie in Bild 8 aus: Es hat den Namen **chkBeitragGeschrieben** und den Steuerelementinhalt **BeitragGeschriebenJaNein**.

Rechts daneben wollen wir nun noch das Datum abbilden für den Fall, dass die Unteraufgabe bereits erledigt wurde. Dazu fügen wir noch die entsprechenden Textfelder hinzu. Diese markieren wir in der Feldliste und ziehen sie gleich neben die bereits angelegten Kontrollkästchen. Die mit diesen Textfeldern ebenfalls zum Formularentwurf hinzugefügten Bezeichnungsfelder markieren wir und löschen diese direkt wieder. Die Textfelder platzieren wir so, dass sie sich direkt neben den Kontrollkästchen befinden.

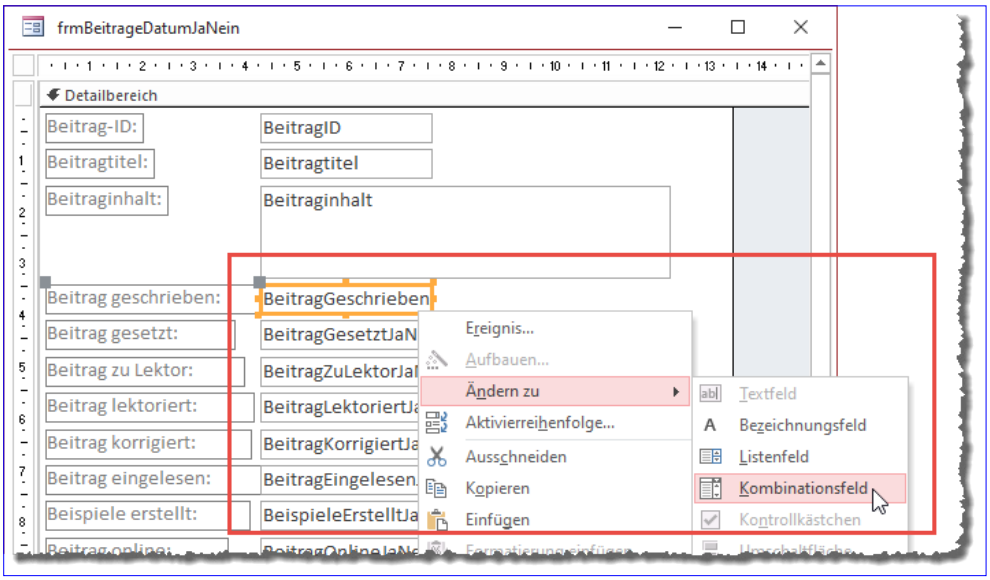


Bild 7: Textfelder können nicht in Kontrollkästchen umgewandelt werden

Die Namen der Textfelder ergänzen wir ebenfalls noch um das entsprechende Präfix, in diesem Fall txt. Das Textfeld für die erste Unteraufgabe heißt so beispielsweise **txtBeitragGeschrieben**.

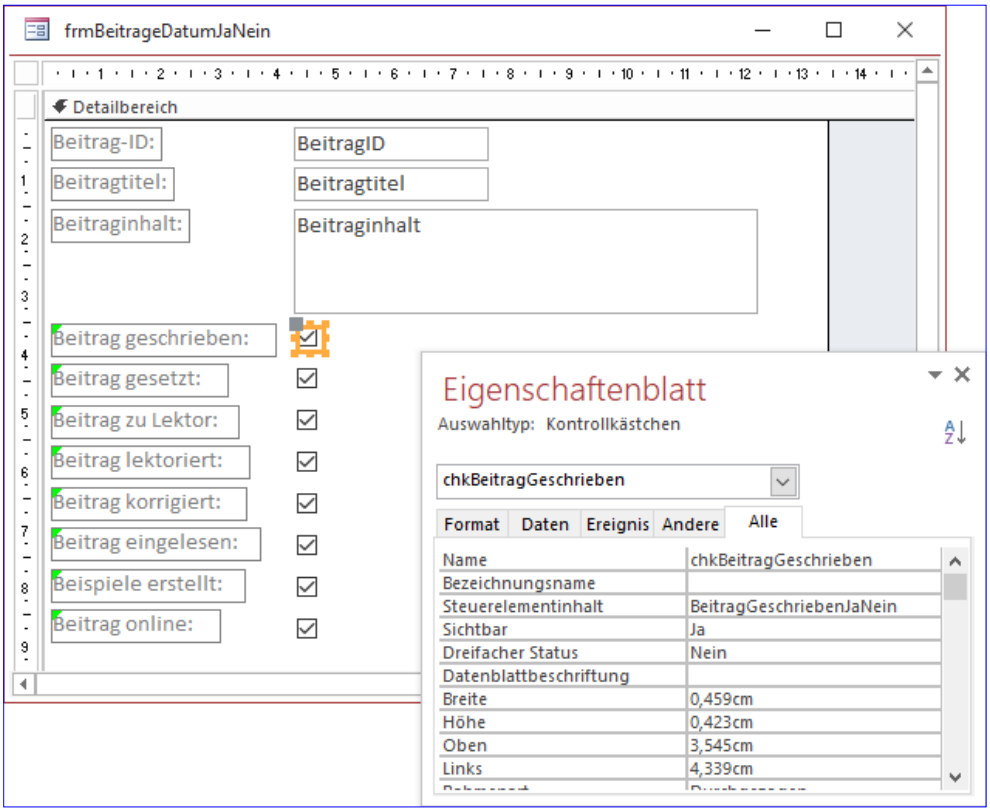


Bild 8: Kontrollkästchen für die Unteraufgaben

Datum für erledigte Aufgaben anzeigen

Nun fehlt noch Code, der dafür sorgt, dass die Textfelder nur erscheinen, wenn die jeweilige Unteraufgabe bereits erledigt wurde. Diesen platzieren wir in der Prozedur, die durch das Ereignis **Beim Anzeigen** des Formulars ausgelöst wird. Diese Prozedur finden Sie in Listing 1.

Hier berechnen wir für jedes Datumsfeld der Datensatzquelle, ob es den Wert Null hat und stellen die Eigenschaft Visible des dazugehörenden Textfeldes zur Anzeige des

HAUFE.
Ausgabe 02/2021

ACCESS
IM UNTERNEHMEN

OUTLOOK IN ACCESS INTEGRIEREN

Erhalten Sie vollen Zugriff auf Outlook-Mails und Co. über die Benutzeroberfläche von Access! (ab S. 62)

In diesem Heft:

Outlook in Access integrieren

Outlook und Access – das sind zwei Office-Anwendungen, die man in der Regel getrennt voneinander verwendet. Outlook kümmert sich um Mails, Termine oder Kontakte und mit Access nutzen Sie Datenbankanwendungen. Gelegentlich tauschen die beiden auch Daten aus. In dieser Ausgabe von Access im Unternehmen schauen wir uns eine bisher recht unbekannte Möglichkeit für die Kooperation von Outlook und Access an: die Integration von Outlook-Elementen in das Formular einer Access-Anwendung.

Dieses Thema haben wir bereits vor einigen Jahren einmal in Angriff genommen. Es gibt seit Langem einige ActiveX-Steuerelemente mit Outlook-Bezug, die sich in Access-Formluare einfügen lassen. Jedoch schien damit kein Zugriff auf Outlook möglich zu sein. Vor kurzer Zeit jedoch wendete sich ein Leser mit der Frage an uns, ob wir nicht einmal etwas zu Steuerelementen wie **Outlook View Control** schreiben können. Und er lieferte gleich noch ein lauffähiges Beispiel mit. Nach weiteren Recherchen im Internet haben wir es dann hinbekommen: Ein Formular, dass in einem **Outlook View Control** die Inhalte verschiedener Outlook-Ordner wie Posteingang, Kalender, Aufgaben et cetera anzeigen konnte.

Diese Ausgabe konzentriert sich auf verschiedene Outlook-Techniken – mit Ausnahme des Beitrags von Bernd Jungbluth mit dem Titel **SQL Server-Security – Teil 5: Rechte für Abteilungen** (ab Seite 45). Hier erfahren Sie, wie Sie für die Mitarbeiter verschiedener Abteilungen eines Unternehmens die entsprechenden Berechtigungen festlegen.

Alle anderen Beiträge drehen sich um Microsoft Outlook. Der erste sogar ausschließlich – hier stellen wir eine Technik vor, die uns im Alltag sehr viel Arbeit spart. Es geht um das Verschiebenen von E-Mails aus dem Posteingang in verschiedene Ordner. Dies realisieren wir einfach durch markieren der E-Mail und anschließendes Betätigen einer Tastenkombination wie **Alt + 1**, **Alt + 2** und so weiter. Den Artikel finden Sie unter dem Titel **Outlook-Mails per Tastenkombination verschieben** ab Seite 2.

Im Beitrag **Kunde zu einer E-Mail öffnen** greifen wir ab Seite 12 diese Technik auf und nutzen diese, um per Tastenkombination den Absender der aktuell markierten E-Mail zu ermitteln und die Kundenverwaltung mit dem Kunden, der diese E-Mail-Adresse besitzt, zu öffnen. Damit springen Sie blitzschnell von einer E-Mail zum Kundendatensatz!

Die übrigen Beiträge drehen sich um das **Outlook View Control**. Der Beitrag **Outlook-Folder in Access anzeigen** zeigt ab Seite 19, wie Sie das Steuerelement überhaupt in einem Formular zum Laufen bringen.

Damit Sie den gewünschten Ordner komfortabel auswählen können, fügen wir mit der Anleitung aus dem Beitrag **TreeView für Outlook-Ordner** ab Seite 35 ein **TreeView**-Steuerelement zur Anzeige der Outlook-Ordner hinzu.

Schließlich bauen wir dieses Beispiel im Beitrag E-Mails verwalten mit dem Outlook View Control (ab Seite 62) zu einem Formular aus, mit dem Sie alle üblichen Techniken rund um das Abfragen, Lesen und Beantworten von E-Mails benötigen.

Viel Spaß beim Ausprobieren!

Ihr André Minhorst

Outlook-Mails per Tastenkombination verschieben

Nicht alle eingehenden E-Mails, die man erhält, müssen beantwortet werden. Beispiele sind Bestellbestätigungen, Rechnungen et cetera. Diese wollen Sie aber vielleicht aus dem Posteingangs-Ordner in einen anderen Ordner verschieben, der beispielsweise nur E-Mails enthält, die mit Bestellungen zu tun haben. Das gelingt per Drag and Drop relativ schnell. Noch besser wäre aber eine Tastenkombination, mit der wir die Mails in die Zielordner verschieben könnten. Dann brauchen Sie beim Durchgehen des Posteingangs die Hände gar nicht mehr von der Tastatur zu nehmen. Die hier vorgestellte Lösung berührt zwar nicht die Datenbankanwendung Microsoft Access, um die es eigentlich in diesem Magazin geht, aber diese Lösung für effizienteres Arbeiten wollen wir Ihnen nicht vorenthalten.

Es gibt sicher Menschen, für die es in Bezug auf die E-Mails im Posteingang von Outlook nur zwei Möglichkeiten gibt: Beantworten und löschen oder direkt löschen. Die meisten müssen aber vielleicht geschäftliche und andere E-Mails aufbewahren oder machen das einfach aus praktischen Gründen. Zum Beispiel ist es sinnvoll, alle E-Mails, die mit Onlinebestellungen zusammenhängen, aufzubewahren.

Dazu bietet es sich unter Outlook an, weitere Ordner unterhalb des Ordners **Posteingang** anzulegen und diesen Bezeichnungen wie beispielsweise **Bestellungen** zu geben (siehe Bild 1). E-Mails, die sich im Posteingang befinden und die mit Bestellungen in Zusammenhang stehen, können Sie dann mit der Maus per Drag and Drop in diesen Ordner verschieben.

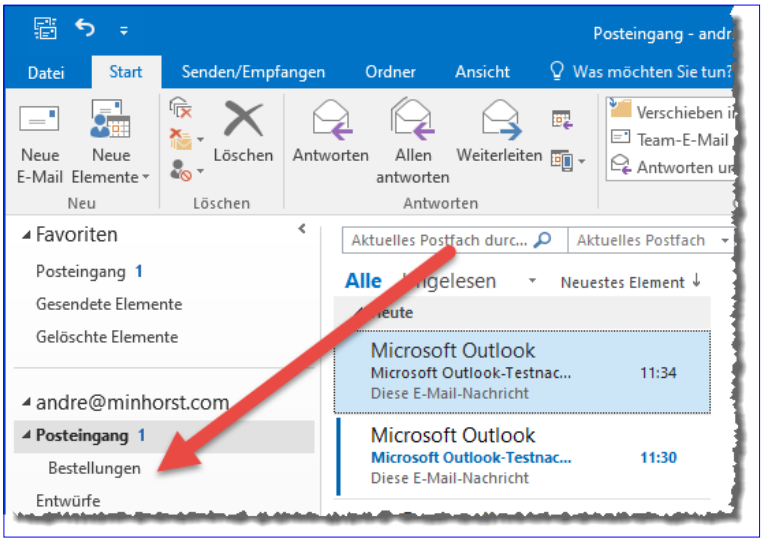


Bild 1: Bestellungen-Ordner in Outlook

Wenn Sie jedoch viele solcher E-Mails bekommen, die Sie schnell einem bestimmten Ordner unterhalb des Posteingang-Ordners zuweisen wollen, wird das ewige Drag and Drop schnell anstrengend.

E-Mails verschieben per Tastenkombination

Also haben wir uns überlegt, wie wir solche Arbeitsschritte vereinfachen können. Die naheliegendste Idee ist der Einsatz einer Tastenkombination, mit der wir eine von uns für diesen Zweck angelegte VBA-Prozedur aufrufen wollen. Die Idee mit der VBA-Prozedur können wir umsetzen, aber es gibt keine direkte Möglichkeit in Outlook, eine solche per Tastenkombination aufzurufen. Daher müssen wir einen kleinen Umweg gehen: Wir können VBA-Prozeduren nämlich mit benutzerdefinierten Schaltflächen im Ribbon aufrufen, und dieses kann mit einigen Einschränkungen per Tastenkombination gesteuert werden.

Wir werden in den nächsten Abschnitten zunächst die benötigten VBA-Prozeduren erstellen und dann die Ribbon-Einträge samt Tastenkombinationen hinzufügen.

Modul im VBA-Projekt von Outlook anlegen

Im Gegensatz zu Access, wo jede Datenbankdatei ein eigenes VBA-Projekt aufweist, gibt es für Outlook ein zentrales VBA-Projekt. Schließlich gibt es in Outlook keine zu öffnenden Dateien beziehungsweise Dokumente wie in den anderen Office-Anwendungen.

Den VBA-Editor mit diesem VBA-Projekt zeigen Sie von Outlook aus mit der Tastenkombination **Alt + F11** an. Hier finden Sie standardmäßig nur den Ordner **Microsoft Outlook Objekte** mit der Klasse **ThisOutlookSession**. Um Prozeduren anzulegen, fügen wir über den Menüeintrag **EinfügenlModul** ein neues Standardmodul hinzu und nennen dieses **mdlMailsAndFolders**. Dieses Modul wir dann im Ordner **Module** angezeigt.

E-Mail per VBA verschieben

Die geplante VBA-Prozedur soll alle zum Zeitpunkt des Aufrufs markierten Einträge des Posteingangs in den mit **strFolder** angegebenen Zielordner verschieben (siehe Listing 1).

Dazu deklarieren wir verschiedene Variablen, die wir anschließend füllen. Als Erstes referenzieren wir mit der Variablen **objExplorer** das mit **ActiveExplorer** ermittelte Fenster zur Ansicht von Ordnerinhalten unter Outlook.

Dann weisen wir den Standardordner des **Explorer**-Objekts der Variablen **objCurrentFolder** zu und prüfen, ob der Name des Ordners **Posteingang** lautet. Wenn Sie eine englische Version von Office nutzen, müssen Sie diese Zeile gegebenenfalls noch anpassen. Hat der Ordner den Namen **Posteingang**, arbeitet die Prozedur die Anweisungen in der **If...Then**-Bedingung ab.

Hier referenzieren wir die ausgewählten Elemente mit der Variablen **objSelection** und den aktuellen MAPI-Namespace mit **objNamespace**. Leider kann man nicht direkt über den Namen auf **Folder**-Objekte zugreifen, die nicht in der ersten Ebene liegen (also unterhalb von **Posteingang**).

Deshalb benötigen wir eine Hilfsfunktion namens **GetFolder**, um den mit **strTargetfolder** angegebenen Ordner zu erhalten. Dieser Funktion, die wir weiter unten beschreiben, übergeben wir die Variablen **objNamespace** und **strTargetfolder** und erhalten ein **Folder**-Objekt mit dem Zielordner zurück.

```
Public Sub MoveToFolder(strTargetfolder As String)
    Dim objMailItem As MailItem
    Dim objExplorer As Explorer
    Dim objSelection As Selection
    Dim objTargetFolder As Folder
    Dim objCurrentFolder As Folder
    Dim objNamespace As NameSpace
    Dim i As Integer
    Set objExplorer = Application.ActiveExplorer
    Set objCurrentFolder = objExplorer.CurrentFolder
    If objCurrentFolder.Name = "Posteingang" Then
        Set objSelection = objExplorer.Selection
        Set objNamespace = Application.GetNamespace("MAPI")
        Set objTargetFolder = GetFolder(objNamespace, strTargetfolder)
        For i = 1 To objSelection.count
            Select Case TypeName(objSelection.Item(i))
                Case "MailItem"
                    Set objMailItem = objSelection.Item(i)
                    objMailItem.Move objTargetFolder
            End Select
        Next i
    End If
End Sub
```

Listing 1: Verschieben der aktuell markierten E-Mails in den mit **strFolder** angegebenen Ordner

Danach durchläuft die Prozedur in einer **For...Next**-Schleife über die Zahlen von 1 bis zur Anzahl der markierten Elemente alle betroffenen Einträge. Nach einer Prüfung, ob es sich bei dem jeweiligen Eintrag um eines mit dem Typ **MailItem** handelt, weisen wir dieses der Variablen **objMailItem** zu. Für dieses rufen wir dann die Methode **Move** auf und geben mit **objTargetfolder** den Zielordner für das Verschieben an.

Die Funktion GetFolder

Die Funktion aus Listing 2 erwartet die Angabe des zu verwendenden **Namespace**-Objekts sowie des Pfades zum zu ermittelten Ordner als Parameter.

Dieser Pfad lautet beispielsweise für den Ordner **Posteingang** wie folgt:

```
\\Outlook\Posteingang
```

Wenn Sie einen Ordner namens **Bestellungen** verwenden wollen, der sich direkt im Ordner **Posteingang** befindet, lautet der Pfad:

```
\\Outlook\Posteingang\Bestellungen
```

Nachfolgend wollen wir die einzelnen Elemente des Pfades in ein Array übertragen und dabei die **Split**-Funktion nutzen, um den Pfad an den Backslash-Zeichen aufzuspalten. Dazu müssen wir zunächst die eventuell vorn angegebenen beiden Backslash-Zeichen entfernen. Ob diese Zeichen angegeben wurden, prüfen wir mit einer ersten **If...Then**-Bedingung und entfernen diese gegebenenfalls.

Um beim obigen Beispiel zu bleiben, erhalten wir nun die folgende Zeichenkette:

```
Outlook\Posteingang\Bestellungen
```

Mit der **Split**-Anweisung fügen wir nun die durch das Backslash-Zeiten getrennten Teilzeichenketten in die Elemente eines Arrays namens **strFolders**.

Nun weisen wir der Variablen **objFolder** den Ordner zu, der den Namen des ersten Elements des Arrays enthält, in diesem Fall **Outlook**. Sofern das Array **strFolders** mehr als ein Element enthält, was der Fall ist, wenn die Differenz aus dem Index des letzten Elements und dem Index des ersten Elements größer als **0** ist, enthält der Pfad einen oder mehrere Ebenen mit Unterordnern.

```
Public Function GetFolder(objNamespace As NameSpace, strSubfolder As String) As Folder
    Dim objFolder As Folder
    Dim strFolders() As String
    Dim i As Integer
    If Left(strSubfolder, 2) = "\\" Then
        strSubfolder = Mid(strSubfolder, 3)
    End If
    strFolders = Split(strSubfolder, "\")
    Set objFolder = objNamespace.Folders(strFolders(0))
    If UBound(strFolders) - LBound(strFolders) > 0 Then
        For i = LBound(strFolders) + 1 To UBound(strFolders)
            Set objFolder = objFolder.Folders(strFolders(i))
        Next i
    End If
    Set GetFolder = objFolder
End Function
```

Listing 2: Ermitteln eines Folder-Objekts anhand des Pfades

Diese wollen wir nun durcharbeiten, bis wir am letzten angegebenen Ordner angelangt sind. Die dazu angelegte **For...Next**-Schleife durchlaufen wir vom zweiten bis zum letzten Element des Arrays.

Darin weisen wir der Variablen **objFolder** jeweils den Ordner zu, der den Namen aus **strFolders(i)** enthält und sich in dem zuvor mit **objFolder** referenzierten **Ordner**-Element befindet.

In unserem Beispiel referenzieren wir im ersten Durchlauf der Schleife das Element **Posteingang** mit der Variablen **objFolder** und im zweiten Durchlauf das Element **Bestellungen**.

Der nach dem Verlassen der Schleife in **objFolder** befindliche Ordner wird schließlich als Funktionsergebnis zurückgeliefert.

Prozedur MoveToFolder aufrufen

Solange wir noch keine Ribbon-Schaltflächen und damit auch noch keine Tastenkombination zu ihrem Aufruf angelegt haben, testen wir die Prozedur vom VBA-Editor aus.

Dazu können Sie, nachdem Sie eine zu verschiebende E-Mail im Posteingang markiert haben, einen Aufruf wie den folgenden im Direktbereich des VBA-Editors absetzen:

```
MoveToFolder "\\Outlook\Posteingang\Bestellungen"
```

Dadurch wird die aktuell markierte E-Mail in den angegebenen Zielordner verschoben. Sie können testweise auch einmal mehrere Elemente gleichzeitig markieren und auf diese Weise verschieben.

Startprozedur hinzufügen

Für den Aufruf über eine Ribbon-Schaltfläche eignen sich nur öffentlich deklarierte Prozeduren ohne Parameter. Also benötigen wir noch eine Wrapper-Prozedur, welche den Parameter für uns übergibt:

```
Public Sub MoveToFolder_Bestellungen()
    MoveToFolder "\\Outlook\Posteingang\Bestellungen"
End Sub
```

Warum bauen wir den Zielordner nicht direkt in die Prozedur **MoveToFolder** ein? Weil Sie gegebenenfalls nicht nur einen Zielordner verwenden wollen, sondern vielleicht auch mehrere.

Dann brauchen Sie für jeden neuen Zielordner nur jeweils eine neue Wrapper-Prozedur hinzuzufügen.

Zielordner ermitteln

Wenn Sie den Pfad zum Zielordner zuverlässig ermitteln wollen, können Sie die folgende Anweisung vom Direktbereich des VBA-Editors aus aufrufen:

```
? Application.GetNamespace("MAPI").PickFolder.FolderPath
```

Dies zeigt den Dialog aus Bild 2 an, mit dem Sie den gewünschten Ordner auswählen können. Die Anweisung gibt den Pfad zum angegebenen Ordner im Direktbereich aus.

Prozedur per Ribbon aufrufen

Auf dem Weg, die Prozedur durch eine Tastenkombination aufrufbar zu machen, benötigen wir eine Ribbon-Schaltfläche. Hier gibt es zwei Möglichkeiten:

- Wir fügen die Schaltfläche dem eigentlichen Ribbon hinzu, als in einer der Gruppen in den **Tab**-Elementen.
- Oder wir legen die Ribbon-Schaltfläche in der Quick Access Toolbar an, das ist die Leiste mit den kleinen Befehlsschaltflächen über dem eigentlichen Ribbon.

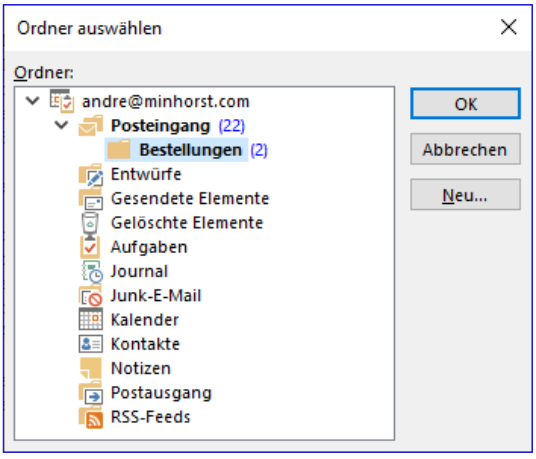


Bild 2: Auswahl eines Ordners

Der Unterschied liegt im Wesentlichen in der Art, wie wir die Befehle per Tastenkombination aufrufen können. Für das eigentlichen Ribbon benötigen wir eine Kombination aus mehreren mit der **Alt**-Taste verwendeten Buchstaben. Wenn Sie die **Alt**-Taste betätigen, werden die direkt verfügbaren Elemente des Ribbons ja mit Buchstaben versehen, über die Sie diese

Kunde zu einer E-Mail öffnen

Viele Access-Benutzer dürften, wenn Sie schon das Office-Paket auf dem Rechner haben, auch Outlook nutzen – zumindest, um E-Mails zum empfangen, zu senden und zu bearbeiten. Vielen fehlt dabei die Möglichkeit einer stärkeren Interaktion zwischen Access und Outlook, beispielsweise bei Verwendung einer Kundenverwaltung. Eine pfiffige Möglichkeit wäre es, beim Anzeigen einer E-Mail von einem bestimmten Kunden direkt den Kundendatensatz in der Kundenverwaltung zu öffnen. Dies sollte dann auch noch auf möglichst einfache Art und Weise geschehen, beispielsweise per Tastenkombination. Wie das gelingt, zeigt der vorliegende Beitrag.

In einem anderen Beitrag mit dem Titel **Outlook-Mails per Tastenkombination verschieben** (**www.access-im-unternehmen.de/1290**) haben wir bereits gezeigt, wie Sie Aktionen für die aktuell markierte E-Mail per Tastenkombination ausführen können.

Im vorliegenden Beitrag wollen wir die Verbindung zwischen der dort beschriebenen reinen Outlook-Technik und Access herstellen. Dazu wollen wir, wenn der Benutzer eine bestimmte Tastenkombination für die aktuell angezeigte E-Mail betätigt, die folgenden Schritte ausführen:

- Die E-Mail-Adresse des Absenders dieser E-Mail ermitteln,
- prüfen, ob die Kundendatenbank bereits geöffnet ist und diese gegebenenfalls öffnen, wobei dann die E-Mail-Adresse übergeben werden soll und
- den Kunden mit der betreffenden E-Mail-Adresse im Kunden-Formular anzeigen.

Ob die Kunden-Datenbank bereits geöffnet ist oder nicht bringt noch eine wichtige Unterscheidung mit sich: Wenn sie noch nicht geöffnet ist, haben wir es etwas einfacher. Wir brauchen die Datenbank dann nämlich einfach nur zu öffnen, was relativ einfach ist, und dabei die entsprechende E-Mail-Adresse zu übergeben. Wenn die Access-Anwendung bereits geöffnet ist, wird es etwas komplizierter, denn dann müssen wir per VBA zunächst die Access-Anwendung referenzieren und diese dann so fernsteuern, dass sie den gewünschten Kunden anzeigt.

Wir wollen jedoch mit dem Aufruf beginnen.

E-Mail auswählen und den Prozess starten

Die Grundlagen zu diesem Schritt haben wir ausführlich in dem eingangs erwähnten Beitrag beschrieben. Dabei haben wir eine VBA-Prozedur erstellt, welche die aktuelle E-Mail ermittelt und dann etwas damit erledigt.

Einige der dort verwendeten Techniken übernehmen wir für die nachfolgend vorgestellte Prozedur namens **ShowCustomerInAccess**, die Sie in Listing 1 finden (die nachfolgend beschriebenen Routinen geben Sie in einem neuen Standardmodul des VBA-Projekts von Outlook ein, das Sie von Outlook aus mit der Tastenkombination **Alt + F11** öffnen). Diese Prozedur schreibt den Pfad zur Kundenverwaltungs-Datenbank in die Variable **strDatabase** und referenziert den aktuell im Outlook-Explorer-Bereich angezeigten Ordner mit der Variablen **objCurrentFolder**.

Heißt dieser Ordner **Posteingang**, wird der Rest der Prozedur ausgeführt. Wenn Sie nicht die deutschsprachige Outlook-Version verwenden, heißt der Ordner möglicherweise anders, in diesem Fall müssen Sie den Code entsprechend anpassen. Die Prozedur erfasst mit der

Variablen **objSelection** die aktuelle Auswahl in diesem Ordner. Wenn die mit **Count** ermittelte Anzahl der markierten Elemente nicht **1** lautet, erscheint eine entsprechende Meldung und die Prozedur wird beendet.

Anderenfalls prüft die Prozedur, ob es sich um ein Element des Typs **MailItem** handelt. Ist das der Fall, weist die Prozedur das markierte Objekt der Variablen **objMailItem** zu. Danach liest es die E-Mail-Adresse des Absenders aus der Eigenschaft **SenderEmailAddress** aus und schreibt sie in die Variable **strMail**.

Anschließend versucht die Prozedur, mit der Funktion **GetDatabase** auf eine laufende Instanz der Datenbank aus **strDatabase** zugreifen. Liefert diese kein Objekt zurück, bleibt **objAccess** leer. Die Prozedur durchläuft dann den **If**-Teil der Bedingung.

```
Public Sub ShowCustomerInAccess()
    Dim objMailItem As MailItem
    Dim objExplorer As Explorer
    Dim objSelection As Selection
    Dim objCurrentFolder As Folder
    Dim objNamespace As NameSpace
    Dim objAccess As Access.Application
    Dim strMail As String
    Dim strDatabase As String
    Dim db As DAO.Database
    strDatabase = "C:\...\Kundenverwaltung.accdb"
    Set objExplorer = Application.ActiveExplorer
    Set objCurrentFolder = objExplorer.CurrentFolder
    If objCurrentFolder.Name = "Posteingang" Then
        Set objSelection = objExplorer.Selection
        Set objNamespace = Application.GetNamespace("MAPI")
        If Not objSelection.count = 1 Then
            MsgBox ("Bitte selektieren Sie genau eine E-Mail.")
            Exit Sub
        End If
        Select Case TypeName(objSelection.Item(1))
            Case "MailItem"
                Set objMailItem = objSelection.Item(1)
                strMail = objMailItem.SenderEmailAddress
                Set objAccess = GetDatabase(strDatabase)
                If objAccess Is Nothing Then
                    Set db = OpenDatabase(strDatabase)
                    If CustomerExists(db, strMail) Then
                        OpenDatabaseInAccess strDatabase, strMail
                    Else
                        MsgBox "Es existiert kein Kunde mit der E-Mail-Adresse '" _
                            & strMail & "'"
                    End If
                    Set db = Nothing
                Else
                    ShowCustomer objAccess, strMail
                End If
        End Select
    End If
End Sub
```

Listing 1: Prozedur zum Anzeigen eines Kunden in Access

Hier erstellt sie mit **OpenDatabase** einen Verweis auf das **Database**-Objekt der Datenbank und referenziert dieses mit der Variablen **db**. Damit ruft sie die Funktion **CustomerExists** auf, die prüft, ob die Datenbank einen Kunden mit der angegebenen E-Mail-Adresse enthält.

Falls ja, ruft die Prozedur eine weitere Routine namens **OpenDatabaseInAccess** auf, welche die Kundenverwaltung öffnen und die E-Mail-Adresse des gesuchten Kunden als Parameter übergeben soll.

Liefert **CustomerExists** den Wert **False** zurück, zeigt die Prozedur eine Meldung an, dass kein Kunde mit der angegebenen E-Mail-Adresse gefunden werden konnte.

```
Public Function GetDatabase(strDatabase As String) As Access.Application
    Dim objAccess As Access.Application
    If IsDatabaseOpen(strDatabase) Then
        Set objAccess = GetObject(strDatabase)
    End If
    Set GetDatabase = objAccess
End Function
```

Listing 2: Funktion zum Holen eines **Access.Application**-Objekts

Es gibt noch den Fall, dass die Funktion **GetDatabase** einen Verweis auf eine Access-Anwendung zurückgeliefert hat. Diese wird im **Else**-Zweig der entsprechenden Bedingung behandelt. Dieser Teil ruft die Prozedur **ShowCustomer** auf und übergibt dieser einen Verweis auf die gefundene Access-Instanz und die E-Mail-Adresse des gesuchten Kunden.

```
Public Function IsDatabaseOpen(strPath As String) As Boolean
    Dim strLACCDB As String
    Dim bolIsDatabaseOpen As Boolean
    strLACCDB = Replace(strPath, ".accdb", ".laccdb")
    If Not Len(Dir(strLACCDB)) = 0 Then
        On Error Resume Next
        Kill strLACCDB
        If Not Err.Number = 0 Then
            bolIsDatabaseOpen = True
        End If
        On Error GoTo 0
    End If
    IsDatabaseOpen = bolIsDatabaseOpen
End Function
```

Listing 3: Funktion zum Prüfen, ob eine Datenbank geöffnet ist

Die hier aufgerufenen Funktionen und Prozeduren beschreiben wir in den folgenden Abschnitten.

Datenbank holen mit GetDatabase

Die erste Funktion, die wir benötigen, heißt **GetDatabase**. Sie erwartet den Pfad der zu holenden Datenbank als Parameter und liefert eine Objektvariable des Typs **Access.Application** zurück (siehe Listing 2).

Die Funktion ruft eine weitere Funktion namens **IsDatabaseOpen** auf, der ebenfalls der Pfad zu der betroffenen Datenbank übergeben wird. Diese Funktion, die wir im Anschluss beschreiben, prüft, ob die mit **strDatabase** angegebene Datenbank geöffnet ist. Ist das der Fall, verwendet die Funktoin **GetDatabase** die VBA-Funktion **GetObject** mit Angabe des Datenbankpfades, um einen Verweis auf das entsprechende **Access.Application**-Objekt zu holen. Der Inhalt der Variablen **objAccess** wird dann als Funktionswert der Funktion **GetDatabase** an die aufrufende Funktion zurückgegeben. Dabei kann **objAccess** an dieser Stelle auch leer sein, also den Wert **Nothing** enthalten.

Auf geöffnete Datenbank prüfen mit IsDatabaseOpen

Die Funktion **IsDatabaseOpen** erwartet mit dem Parameter **strPath** die Angabe der zu untersuchenden Datenbankdatei (siehe Listing 3). Der Ansatz ist, die beim Öffnen einer Datenbankdatei von Access erstellte Datei mit der Endung **.laccdb** zu untersuchen. Diese wird immer im gleichen Verzeichnis wie die **.accdb**-Datei erstellt. Außerdem ist diese Datei, wenn die Datenbank noch geöffnet ist, schreibgeschützt und kann nicht gelöscht werden. Manchmal kommt es vor, dass die Access-Datenbank geschlossen wird und die **.laccdb**-Datei noch vorhanden ist. Dann kann diese aber problemlos gelöscht werden, was wir in der Funktion ausnutzen.

Outlook-Folder in Access anzeigen

Es gibt sehr viele Möglichkeiten für Interaktion zwischen Outlook und Access. Sie können Termine, E-Mails, Kontakte oder Aufgaben zwischen den beiden Anwendungen abfragen, synchronisieren, erstellen oder bearbeiten. Sehr stiefmütterlich wurde bisher allerdings das Thema der Anzeige von Outlook-Elementen in Access behandelt. Zum Glück brachte mich neulich ein Leser auf die Idee, das Thema noch einmal aufzugreifen – und lieferte mir auf mein Zögern hin direkt noch ein Beispiel, wie die Integration funktioniert. Wir haben uns dies einmal genau angesehen. Das Ergebnis und die resultierenden Möglichkeiten finden Sie im vorliegenden Beitrag.

Bereits vor einigen Jahren habe ich aus Neugier einmal versucht, eines der interessant aussehenden Elemente des Dialogs zum Einfügen von ActiveX-Steuerelementen in Formulare zu nutzen (siehe Bild 1).

Allerdings bin ich damals nicht besonders weit gekommen – vermutlich war die Dokumentation dieser Elemente damals noch lückenhafter als heute.

Dank unseres Lesers Oliver Specht habe ich das Thema allerdings nochmal angesehen und dank seiner Vorarbeit lief es nun direkt wesentlich besser.

E-Mails im Access-Formular anzeigen

Die erste Idee war, den Outlook-Explorer mit den E-Mails einmal in einem Access-Formular anzuzeigen.

Dazu gehen Sie wie folgt vor:

- Legen Sie ein neues, leeres Formular an.
- Öffnen Sie mit dem Ribbon-Eintrag **Entwurf|Steuerelemente|ActiveX-Steuerelemente** den Dialog **ActiveX-Steuerelemente einfügen** und scrollen Sie zu den Einträgen, die mit **Outlook** beginnen.

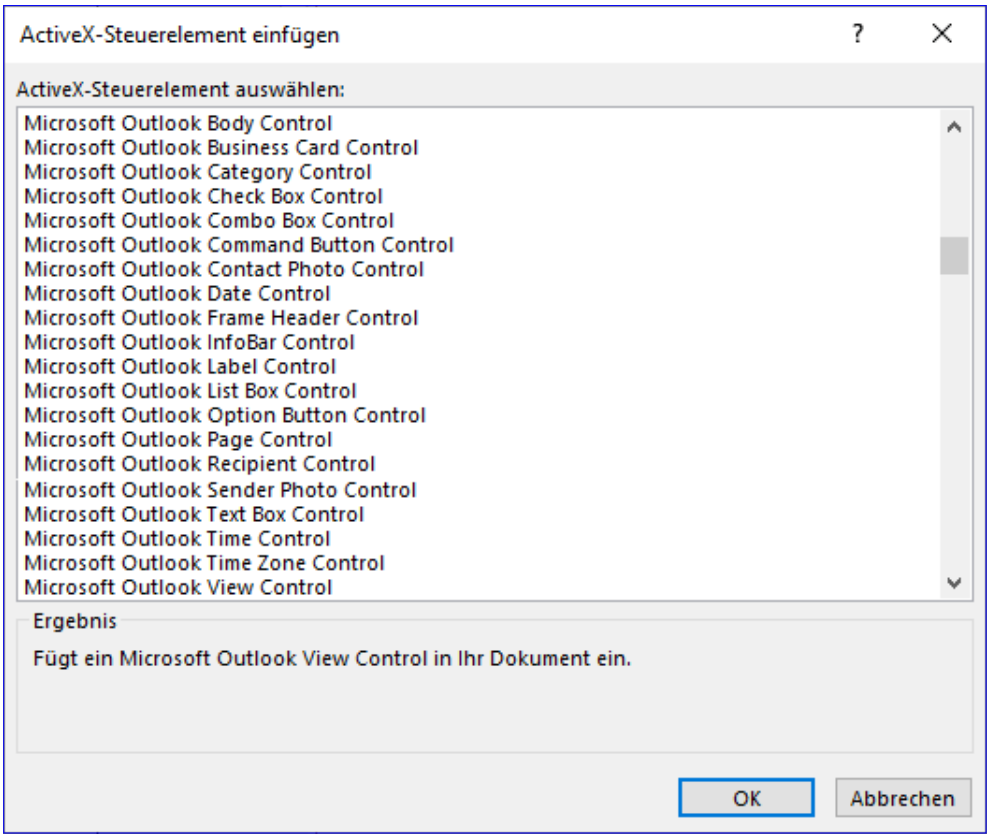


Bild 1: Als ActiveX-Steuerelemente einfügbare Outlook-Elemente

- Fügen Sie das ActiveX-Steuerelement **Microsoft Outlook View Control** in das Formular ein und passen Sie seine Größe nach Wunsch an.

Danach sieht das Formular wie in Bild 2 aus.

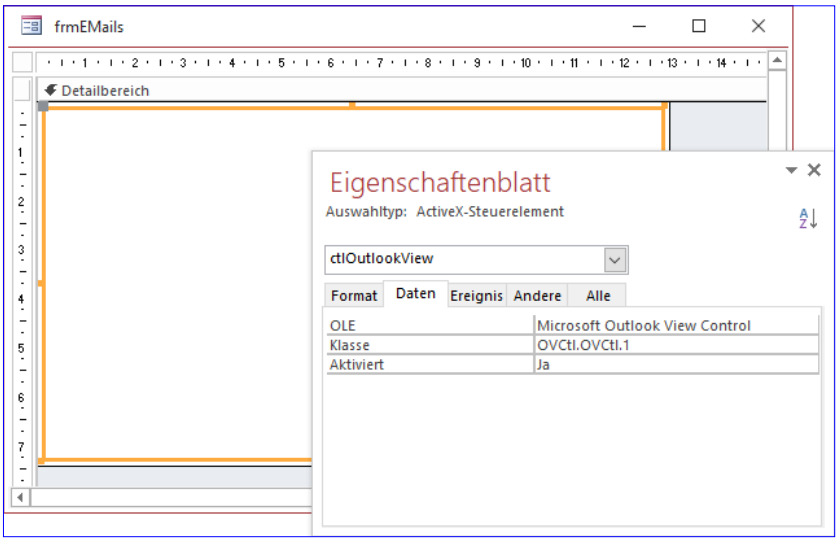


Bild 2: Das **Microsoft Outlook View Control** in einem Access-Formular

Wenn wir nun in die Formularansicht wechseln, erhalten wir die Meldung **In diesem Steuerelement befindet sich kein Objekt**.

Die gleiche Meldung erscheint auch nochmal beim Wechsel zurück zur Entwurfsansicht, danach aber nicht mehr.

Wenn wir das Steuerelement nochmal löschen und erneut einfügen, können Sie einen Test machen: Klicken Sie, bevor Sie in die Formularansicht wechseln, einmal in der Entwurfsansicht doppelt auf das hinzugefügte Steuerelement.

Dieses zeigt dann wie in Bild 3 die Liste der E-Mails an – und zwar die aktuellen, in Outlook angezeigten Einträge. Nach einem Wechsel in die Formularansicht erhalten wir allerdings das gleiche Ergebnis wie zuvor.

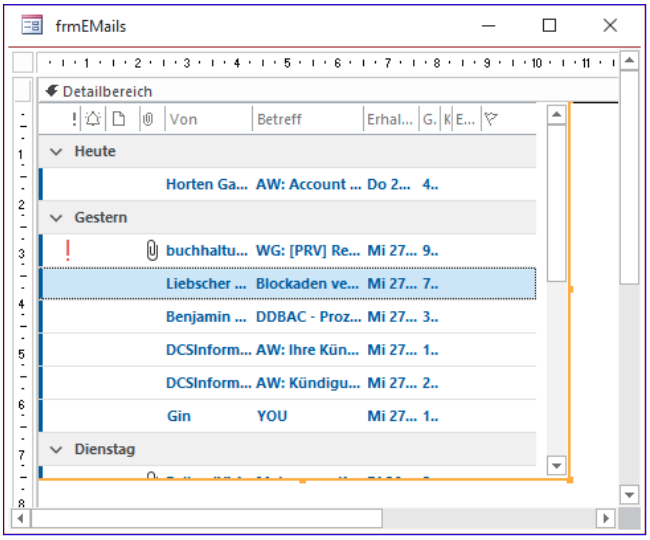


Bild 3: Anzeige von Outlook-Daten im Entwurf

Outlook-View in Frame

Die Lösung ist das Einbetten des **Outlook View Control**-Steuerelements in ein weiteres Steuerelement, nämlich das **Frame**-Steuerelement der **Forms**-Bibliothek.

Der Vorgang ist nicht besonders intuitiv, daher hier die ausführliche Beschreibung:

- Fügen Sie einem neuen, leeren Formular aus dem Dialog **ActiveX-Steuerelemente einfügen** das Element **Microsoft Forms 2.0 Frame** hinzu (siehe Bild 4).

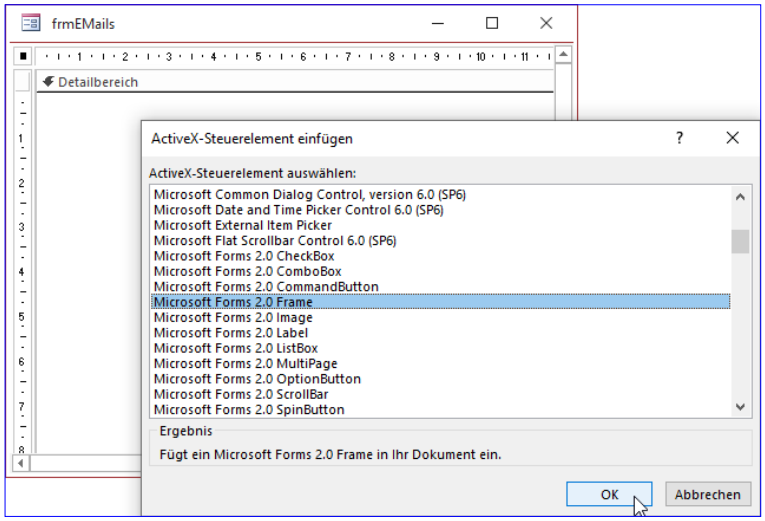


Bild 4: Hinzufügen eines Frames

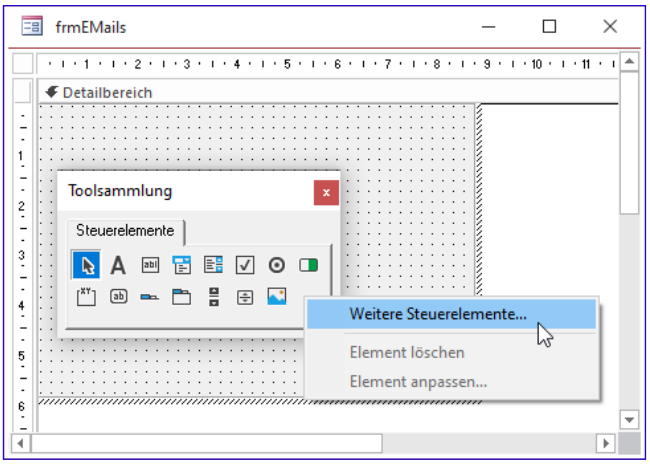


Bild 5: Freischalten weiterer Steuerelemente

- Danach klicken Sie doppelt auf das frisch eingefügte **Frame**-Steuerelement.

- Sollte die Toolbox nicht angezeigt werden, klicken Sie mit der rechten Maustaste in das Steuerelement und aktivieren Sie den Eintrag **Toolsammlung...**.

- Klicken Sie mit der rechten Maustaste in den Bereich mit den Steuerelementen und wählen Sie den Kontextmenü-Eintrag **Weitere Steuerelemente** aus (siehe Bild 5).

- Selektieren Sie im nun erscheinenden Dialog **Weitere Steuerelemente** den Eintrag **Microsoft Outlook View Control** (siehe Bild 6).

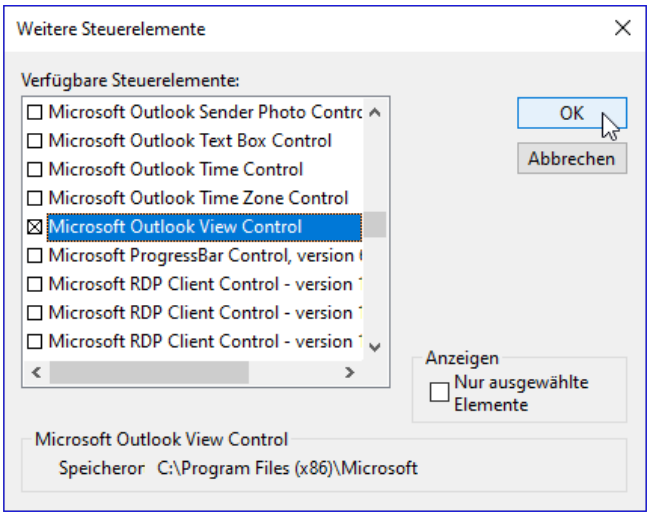


Bild 6: Auswahl des **Outlook View Control**-Steuerelements

- Dieses finden Sie danach in der Toolbox vor. Ziehen Sie es per Drag and Drop in das **Frame**-Objekt (siehe Bild 7).

- Schließlich ziehen Sie es auf die gewünschte Größe – so, dass es die gesamte Fläche des **Frame**-Steuerelements ausfüllt.

Wechseln Sie nun in die Formularansicht, haben wir das erste Zwischenziel erreicht: Das Formular zeigt die Liste der E-Mails an (siehe Bild 8). Sie können die angezeigten

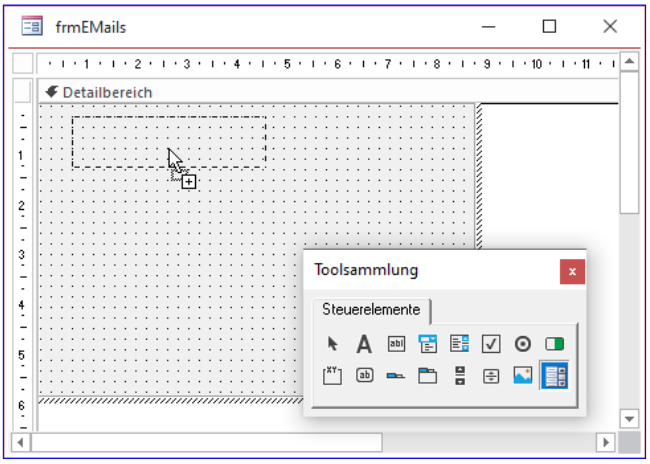


Bild 7: Hinzufügen des **ViewCtl**-Steuerelements per Drag and Drop

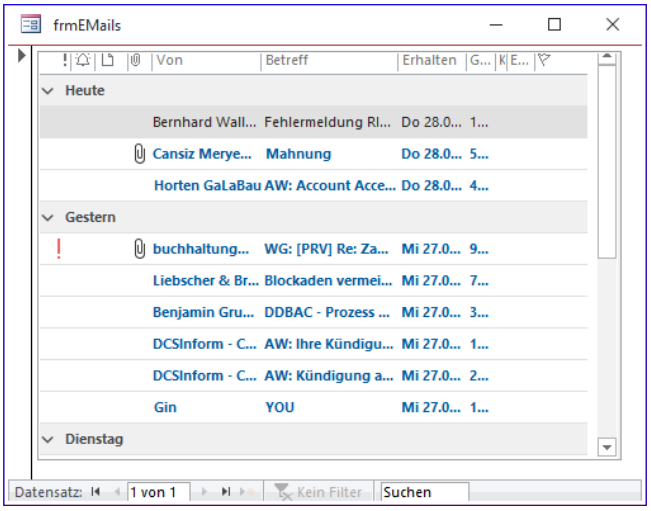


Bild 8: Anzeige der E-Mails im Formular

Elemente genau wie in Outlook nutzen, also die Spalten anpassen, E-Mails löschen oder diese per Doppelklick öffnen. Nach dem Öffnen einer E-Mail wird diese in dem von Outlook bekannten Fenster angezeigt.

View-Steuerelement per VBA referenzieren

Nun wollen wir herausfinden, wie wir per VBA auf das **Microsoft Outlook View Control** zugreifen können. Nun befindet sich das Steuerelement in einem weiteren Steuerelement des Typs **Frame**. Wir stellen einmal nach, wie wir herausfinden, wie das Steuerelement zu referenzieren ist.

Als Erstes wollen wir das **Frame**-Steuerelement per VBA-Variable referenzieren. Dazu müssen wir herausfinden, welchen Typ das **Frame**-Steuerelement hat.

Dazu öffnen wir das Formular, das aktuell nur das **Frame**-Steuerelement mit dem View-Steuerelement enthält, in der Formularansicht. Im Direktbereich können wir nun mithilfe der **TypeName**-Funktion den Typ des **Frame**-Steuerelements herausfinden. Dazu setzen Sie dort den folgenden Befehl ab:

```
Debug.Print TypeName(Screen.ActiveForm.Controls(0))
CustomControl
```

Wir referenzieren mit **Screen.ActiveForm** zunächst das aktuelle Formulare. Dieses könnten wir auch mit **Forms!<Formularname>** referenzieren, aber **Screen.ActiveForm** ist einfacher, weil wir den Formularnamen dazu nicht kennen müssen.

Da das Formular nur ein Steuerelement enthält, können wir dieses mit **Controls(0)** ansprechen. Das Ergebnis lautet **CustomControl**. Das ist nicht das gewünschte Ergebnis.

Der Grund ist: ActiveX-Steuerelemente spricht man immer noch zusätzlich über die **Object**-Eigenschaft an.

Das ergibt dann:

```
Debug.Print TypeName(Screen.ActiveForm.Controls(0).Object)
Frame
```

Damit ist bestätigt, dass wir eine Objektvariable für das **Frame**-Objekt beispielsweise in der Prozedur, die durch das Ereignis **Beim Laden** des Formulars ausgelöst wird, mit dem Datentyp **Frame** deklarieren können:

```
Private Sub Form_Load()
    Dim objFrame As Frame
    Set objFrame = Me!ctlFrame.Object
End Sub
```

Warum wollen wir dieses Steuerelement überhaupt per Objektvariable referenzieren? Weil wir so über IntelliSense auf seine Eigenschaften zugreifen können!

In einer weiteren Anweisung der obigen Prozedur können wir nun den Typ des einzigen Elements der **Controls**-Auflistung des **Frame**-Steuerelements ermitteln:

```
Debug.Print TypeName(objFrame.Controls(0).Object)
```

Dies liefert das Ergebnis **IViewCtl**, was eine Schnittstelle beschreibt. Wir verwenden allerdings **ViewCtl** und weisen das Steuerelement wie folgt zu:

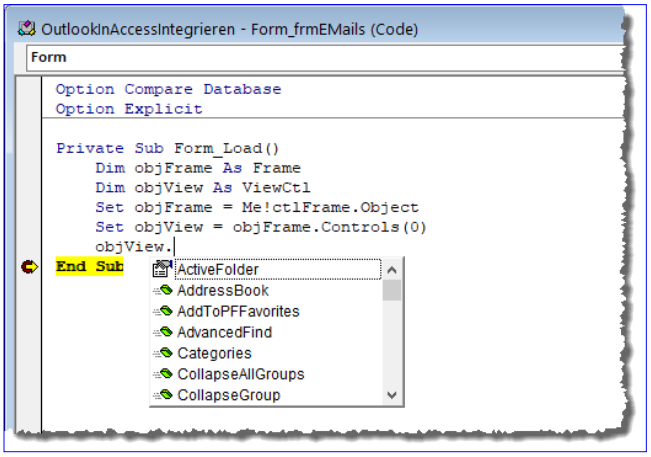


Bild 9: Zugriff per IntelliSense auf die Eigenschaften und Methoden des **View**-Steuerelements

```
Dim objFrame As Frame
Dim objView As ViewCtl
Set objFrame = Me!ctlFrame.Object
Set objView = objFrame.Controls(0)
```

Damit können wir dann per IntelliSense die Eigenschaften und Methoden dieses Steuerelements ansteuern (siehe Bild 9).

Eigenschaften und Methoden des View-Steuerelements

Damit können wir uns nun die Eigenschaften und Methoden ansehen und unter anderem die Ansicht des Steuerelements damit steuern. Einige davon können wir uns aber auch bereits über die Benutzeroberfläche ansehen. Das wollen wir hier noch vorziehen. Dazu zeigen Sie das Formular in der Entwurfsansicht an.

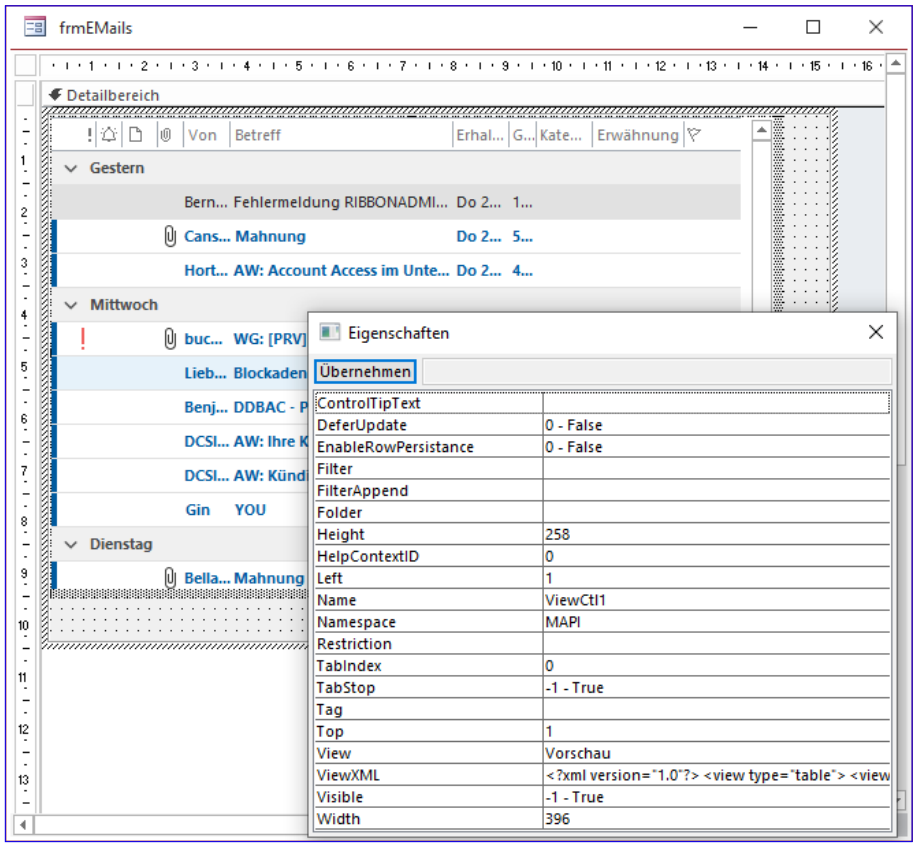


Bild 10: Eigenschaften des **View**-Steuerelements

Klicken Sie doppelt auf das **Frame**-Steuerelement, sodass das **View**-Steuerelement erscheint. Wählen Sie dann aus dem Kontextmenü des **View**-Steuerelements den Eintrag **Eigenschaften** aus. Dies liefert das Eigenschaften-Fenster aus Bild 10.

Eine interessante Eigenschaft lautet **ViewXML**. Diese liefert offensichtlich den Aufbau der Ansicht im XML-Format.

Wollen Sie dieses XML-Dokument betrachten, können Sie es einfach durch Hinzufügen des folgenden Befehls zur Prozedur **Form_Load** im Direktbereich des VBA-Editors ausgeben:

```
Private Sub Form_Load()
    ...
    Debug.Print objView.ViewXML
End Sub
```

In diesem XML-Dokument finden Sie einige Attribute, die das Layout der E-Mail-Liste beschreiben (siehe Listing 1). Am Beispiel der Sortierung können Sie prüfen, dass die XML-Definition jeweils an die aktuellen Einstellungen im **View**-Steuerelement angepasst wird. Wenn Sie beispielsweise eine andere Spalte als Sortierkriterium festlegen, schlägt sich dies direkt im Unterelement **orderby** nieder. Im folgenden Beispiel ist die Sortierreihenfolge auf die Spalte **Erhalten** festgelegt:

```
<orderby>
    <order>
        <heading>Erhalten</heading>
        <prop>urn:schemas:httpmail:datereceived</prop>
        <type>datetime</type>
        <sort>desc</sort>
    </order>
</orderby>
```

Ändern wir die Sortierung auf die Spalte **Von** und fragen die XML-Definition erneut ab, erhalten wir für das Element **orderby** den folgenden Code:

```
<orderby>
  <order>
    <heading>Von</heading>
    <prop>urn:schemas:httpμ
       mail:fromname</prop>
    <type>string</type>
    <sort>asc</sort>
  </order>
</orderby>
```

Funktion für den Zugriff auf das View-Steuerelement per VBA

In den folgenden Abschnitten werden wir einige Dinge zur Steuerung des View-Steuerelements per VBA über den Direktbereich des VBA-Editors ausprobieren. Da es wenig Spaß macht, immer den kompletten Ausdruck zum Referenzieren dieses Steuerelements im aktuell geöffneten Formular einzugeben, bauen wir uns zu diesem Zweck eine kleine Hilfsfunktion in einem Standardmodul.

Diese Funktion heißt **OutlookView** und Sie finden diese in Listing 2. Damit können Sie nun einfach mit **OutlookView** auf das

```
<?xml version="1.0"?>
<view type="table">
    <viewname>Vorschau</viewname>
    <viewstyle>font-family:Segoe UI;...;table-layout:fixed;width:100%</viewstyle>
    <viewtime>0</viewtime>
    <linecolor>8421504</linecolor>
    <linestyle>3</linestyle>
    <previewlines>0</previewlines>
    <previewlineschangenum>2</previewlineschangenum>
    <autopreview>1</autopreview>
    <previewunreadonly>1</previewunreadonly>
    <ensuredcategoriesfield>1</ensuredcategoriesfield>
    <collapsestate/>
    <rowstyle>background-color:White;color:Black</rowstyle>
    <headerstyle>background-color:#D3D3D3</headerstyle>
    <previewstyle/>
    <arrangement>
        <autogroup>1</autogroup>
        <enablexfc>1</enablexfc>
        <collapseclient>01000000</collapseclient>
        <collapseconv/>
        <upgradetoconvchangenum>2</upgradetoconvchangenum>
    </arrangement>
    ...
    <column>
        <heading>Von</heading>
        <prop>urn:schemas:httpmail:fromname</prop>
        <type>string</type>
        <width>38</width>
        <style>text-align:left;padding-left:3px</style>
        <editable>0</editable>
        <displayformat>1</displayformat>
    </column>
    ...
    <orderby>
        <order>
            <heading>Erhalten</heading>
            <prop>urn:schemas:httpmail:datereceived</prop>
            <type>datetime</type>
            <sort>desc</sort>
        </order>
    </orderby>
    <groupbydefault>2</groupbydefault>
    <previewpane>
        <markasread>0</markasread>
    </previewpane>
</view>
```

Listing 1: XML-Definition der aktuellen Ansicht

Steuerelement und seine Eigenschaften und Methoden zugreifen.

```
Public Function OutlookView() As ViewCtl
    Set OutlookView = Forms!frmEMails!ctlFrame.Object.Controls(0)
End Function
```

Listing 2: VBA-Prozedur für den Zugriff auf das **View**-Steuerelement

Verschiedene Ansichten einstellen mit Folder und View

Die XML-Definition liefert eine Menge Informationen, aber wir können dieser nicht entnehmen, welcher Ordner-Inhalt gerade angezeigt wird. Dies lässt sich allerdings mit der **Folder**-Eigenschaft realisieren.

Diese liefert, wenn Sie den aktuellen Wert ausgeben, für die aktuelle Ansicht allerdings keinen Wert – wie wir mit unserer Funktion **OutlookView** im Direktbereich des VBA-Editors belegen können:

```
? OutlookView.Folder
<leere Zeichenkette>
```

Eine zweite wichtige Eigenschaft lautet **View**. Diese liefert direkt nach dem Anzeigen bereits einen Wert:

```
? OutlookView.View
Vorschau
```

Interessanterweise erscheinen die Werte offensichtlich in der jeweiligen Landessprache. Woher aber bekommen wir die möglichen Werte für diese Eigenschaften? Der Objektkatalog liefert jedenfalls kein Element mit dem Text **Vorschau**. Und welche Werte können wir für die Eigenschaft **Folder** verwenden?

Folder-Werte herausfinden

Die **Folder**-Werte entsprechen schlicht den Ordnerbezeichnungen aus Outlook. Um alle möglichen Einträge zu sehen, brauchen Sie nur die Ansicht des Ordnerbereichs von Outlook auf **Ordner** umzustellen.

Das erledigen Sie mit einem Klick auf die Schaltfläche mit den drei Punkten unten im Ordnerbereich und anschließende Auswahl des Eintrags **Ordner** (siehe Bild 11).

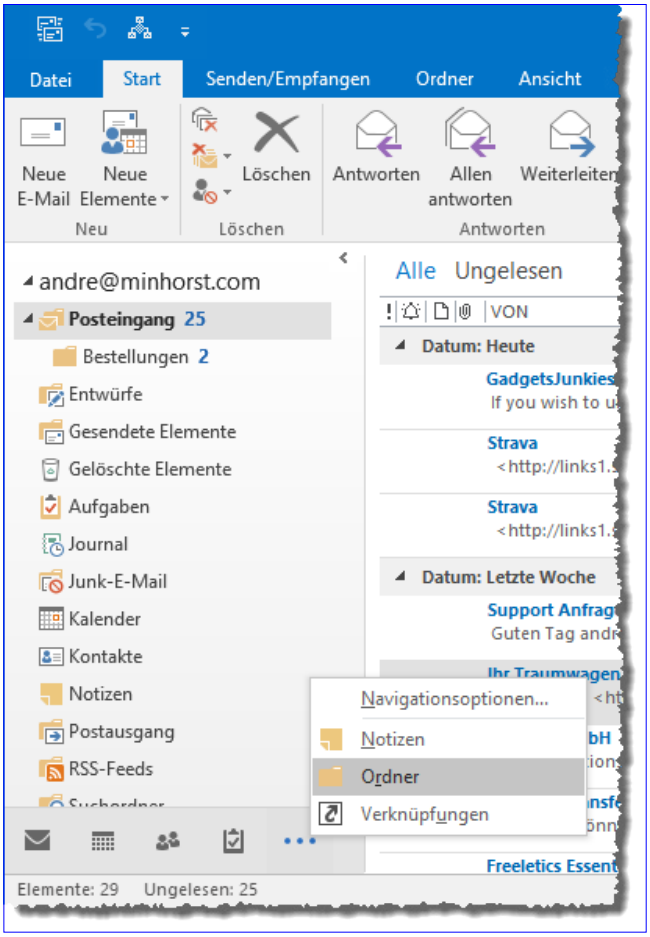


Bild 11: Ordner in Outlook

Danach brauchen Sie die Eigenschaft **Folder** nur noch auf den Namen des Ordners einzustellen, also beispielsweise wie folgt:

```
OutlookView.Folder = "Kalender"
OutlookView.Folder = "Aufgaben"
OutlookView.Folder = "Gesendete Elemente"
```

Wenn Sie dem Posteingang noch weitere Ordner hinzugefügt haben und ihren Inhalt anzeigen wollen, geben Sie

TreeView für Outlook-Ordner

Im Beitrag »Outlook-Folder in Access anzeigen« liefern wir die Grundlagen zur Anzeige von Outlook-Ordnern in Access-Formularen. Dabei haben wir die einzelnen Outlook-Ordner in einem einfachen Listenfeld zur Auswahl angeboten. Im Beitrag »E-Mails verwalten mit dem Outlook View Control« wollen wir dies ein wenig professioneller gestalten und die Outlook-Ordner in einem TreeView-Steuerelement anzeigen. Wie das gelingt, erfahren Sie im vorliegenden Beitrag.

Das **TreeView**-Steuerelement, in dem wir die Outlook-Ordner in der gleichen hierarchischen Anordnung anzeigen wollen wie im entsprechenden Bereich von Outlook, soll die Auswahl eines der Ordner ermöglichen. Der Inhalt dieses Ordners soll in einem **Outlook View Control** erscheinen. Unter Outlook sieht die hierarchische Anzeige wie in Bild 1 aus.

Für den Anfang wollen wir uns dabei bezüglich der Icons damit begnügen, für jeden Ordner das gleiche Icon anzuzeigen, nämlich ein einfaches Ordner-Icon.

Außerdem wollen wir nur die Ordner anzeigen, die einen bestimmten Typ von Elementen enthalten. In diesem Fall sind das die Ordner, die typischerweise E-Mails enthalten. Wie Sie diese Ordner von den anderen unterschieden, erläutern wir weiter unten.

Welche Schritte sind für die Umsetzung unseres Vorhabens nötig? Zunächst benötigen wir einige Steuerelemente:

- ein **TreeView**-Steuerelement und
- ein **ImageList**-Steuerelement zum Speichern der zu verwendenden Icons.

Außerdem benötigen wir VBA-Code, der beim Laden des Formulars ausgelöst wird und der die betreffenden Ordner aus Outlook einliest und entsprechende Einträge im **TreeView**-Steuerelement anlegt.

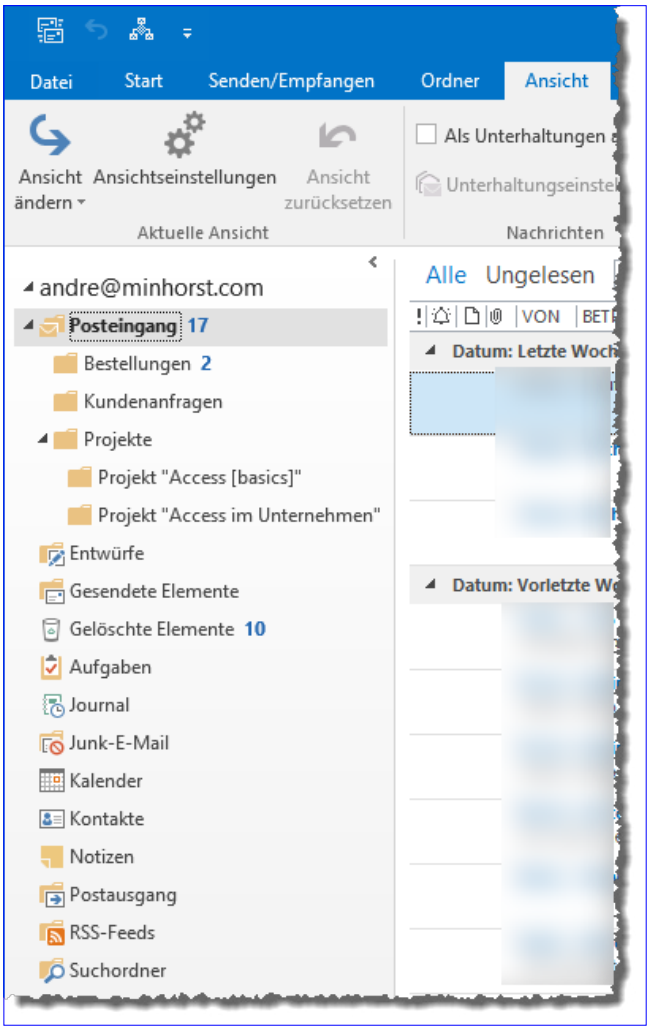


Bild 1: Anzeige der Ordner in Outlook

Steuerelemente hinzufügen

Die beiden benötigten Steuerelemente fügen Sie der Entwurfsansicht eines neuen Formulars namens **frmE-**

MailDetails hinzu, indem Sie über den Ribbon-Eintrag **Entwurfl Steuerelemente|ActiveX-Steuerelemente** den Dialog **ActiveX-Steuerelement einfügen** öffnen. Hier wählen Sie als Erstes das Steuerelement **Microsoft TreeView Control, version 6.0** aus und klicken **OK** (siehe Bild 2).

Danach führen Sie den gleichen Vorgang für das Steuerelement **Microsoft ImageList Control, version 6.0** aus.

Legen Sie für dieses Steuerelement den Namen **ctlImageList** fest und für das **TreeView**-Steuerelement den Namen **ctlTreeView**. Ziehen Sie das **TreeView**-Steuerelement so auf, dass es sich am linken Rand des Formulars befindet (siehe Bild 3).

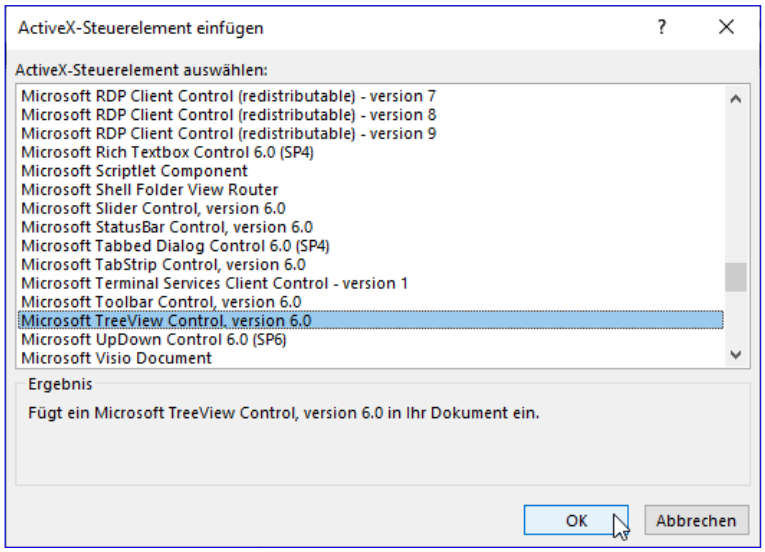


Bild 2: Hinzufügen der ActiveX-Steuerelemente

Icons zu ImageList-Steuerelement hinzufügen

Danach fügen wir das zunächst zu verwendende Icon zum **ImageList**-Steuerelement hinzu. Dazu klicken Sie doppelt auf das Steuerelement und erhalten den Dialog **Eigenschaften von ImageListCtrl**. Hier stellen Sie die Bildgröße auf der Registerseite **General** auf **16 x 16** ein (siehe Bild 4).

Danach wechseln Sie auf die Registerseite **Images** und klicken dort auf die Schaltfläche **Insert Picture...**, was einen **Dateiauswahl**-Dialog öffnet.

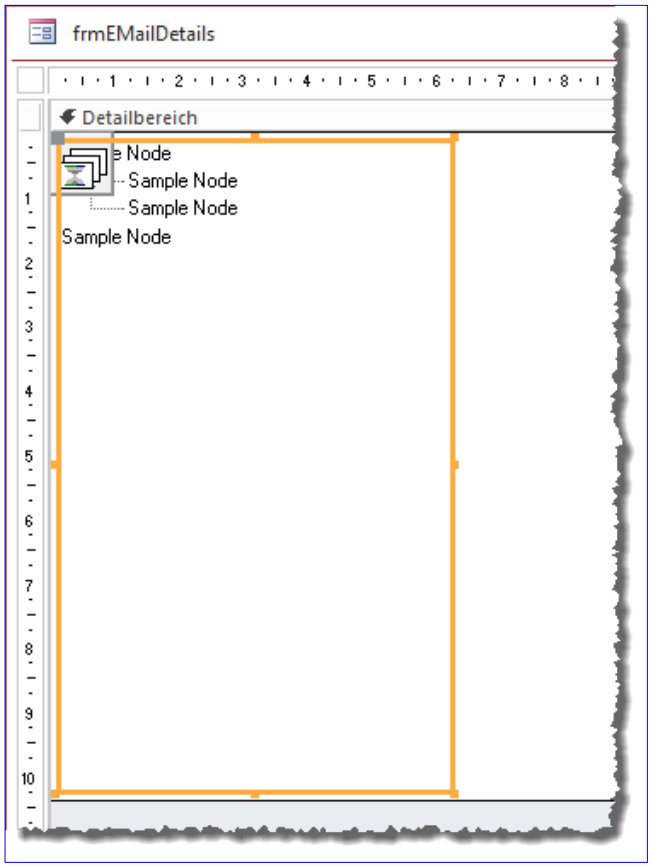


Bild 3: Die beiden neuen Steuerelemente im Formular

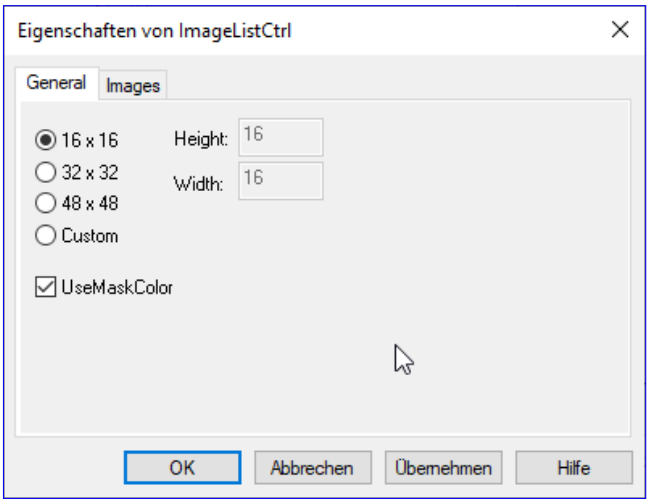


Bild 4: Festlegen der Icon-Größe

Hier fügen Sie nacheinander zwei Icons ein, die einen geschlossenen und einen geöffneten Ordner anzeigen (siehe Bild 5). Für diese beiden Elemente legen Sie im Feld **Key** die Schlüssel **folder_closed** und **folder_open** fest.

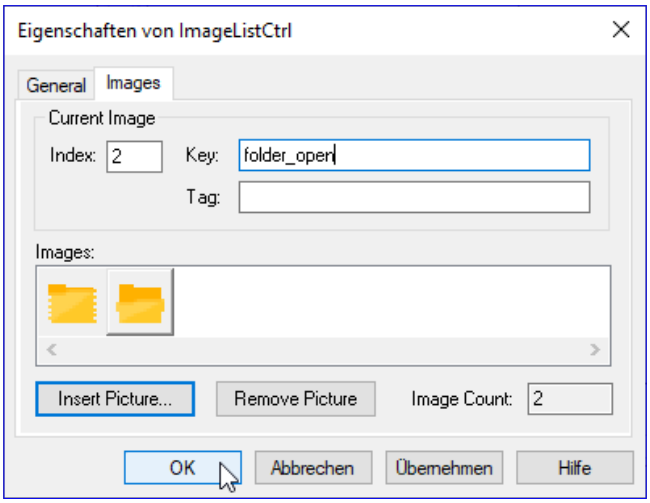


Bild 5: Hinzufügen und benennen der Icons

Einstellungen für das TreeView-Steuerelement

Danach nehmen wir die Einstellungen für das **TreeView**-Steuerelement vor. Das erledigen wir in der Regel per VBA, weil sich so ein Satz von Einstellungen recht leicht von einer Lösung zur nächsten übertragen lässt – das geht schneller, als wenn Sie die Einstellungen jedes Mal erneut im Eigenschaften-Fenster erledigen.

Die Einstellungen nehmen wir in einer eigenen Prozedur vor, die wir **InitializeTreeView** nennen. Diese rufen wir in der Ereignisprozedur auf, die beim Laden des Formulars ausgelöst wird:

```
Private Sub Form_Load()
    InitializeTreeView
    ...
End Sub
```

Die Prozedur **InitializeTreeView** sieht wie folgt aus:

```
Private Sub InitializeTreeView()
    Dim objTreeView As MSComctlLib.TreeView
    Set objTreeView = Me!ctlTreeView.Object
    With objTreeView
        .Appearance = ccFlat
        .BorderStyle = ccNone
        .Font.Name = "Calibri"
        .Font.Size = 9
        .Indentation = 200
        Set .ImageList = Me!ctlImageList.Object
        .LineStyle = tvwRootLines
        .Style = tvwTreelinesPlusMinusPictureText
        .Nodes.Clear
        FillTreeView objTreeView
        ExpandTreeViewNodes objTreeView
    End With
End Sub
```

Sie füllt zunächst die Variable **objTreeView** mit einem Verweis auf das **TreeView**-Steuerelement. Dann stellt sie einige Eigenschaften für das so referenzierte **TreeView**-Steuerelement ein. **Appearance** gibt an, ob es in einer 3D-Ansicht oder flach angezeigt werden soll.

Mit **BorderStyle** legen wir fest, dass kein Rahmen erscheinen soll – wenn, dann definieren wir diesen über die Steuerelementeigenschaften des Containers für das ActiveX-Steuerelement. Mit **Font** referenzieren wir das **Font**-Objekt, mit dessen Eigenschaften **Name** und **Size** wir die Schriftart und die Schriftgröße einstellen.

Den Standardeinzug reduzieren wir mit der Eigenschaft **Indentation** auf den Wert **200** (standardmäßig **567**). Die Eigenschaft **ImageList** legt das zu verwendende **ImageList**-Steuerelement fest, aus dem die Icons für die **TreeView**-Elemente bezogen werden. Hier müssen wir wie beim **TreeView**-Steuerelement mit **Object** auf das im ActiveX-Container enthaltene Steuerelement zugreifen. **LineStyle** gibt mit dem Wert **tvwRootLines** an, dass wir schon für die Root-Elemente Linien und somit auch Plus/Minus-Zeichen anzeigen wollen. Und mit **Style** und dem Wert **tvwTreelinesPlusMinusPictureText** bestimmen wir, dass Linien, Plus/Minus-Zeichen, Icons und Text für die

```
Private Sub FillTreeView(objTreeView As MSComctlLib.TreeView)
    Dim objOutlook As Outlook.Application
    Dim objNamespace As Outlook.Namespace
    Dim objFolder As Outlook.Folder
    Dim objNode As MSComctlLib.Node
    Set objOutlook = New Outlook.Application
    Set objNamespace = objOutlook.GetNamespace("MAPI")
    For Each objFolder In objNamespace.Folders
        If objFolder.DefaultItemType = olMailItem Then
            Set objNode = objTreeView.Nodes.Add(, , objFolder.FolderPath, objFolder.Name, "folder_closed")
            FillNode_Rek objTreeView, objNode, objFolder
        End If
    Next objFolder
End Sub
```

Listing 1: **TreeView**-Steuerelement füllen

einzelnen Elemente angezeigt werden sollen – also das komplette Programm.

Schließlich leeren wir die **Nodes**-Auflistung des Steuerelements, damit eventuell bei einer früheren Anzeige noch verbliebene Elemente verschwinden und rufen die Prozedur **FillTreeView** auf, der wir den Verweis auf das zu füllende **TreeView**-Steuerelement übergeben. Nachdem dies geschehen ist, wollen wir noch den Eingeklappt/Ausgeklappt-Zustand der Elemente wiederherstellen, der vor dem letzten Schließen des Formulars Bestand hatte. Das erledigen wir mit der Prozedur **ExpandTreeViewNodes**, dem wir ebenfalls einen Verweis auf das **TreeView**-Steuerelement übergeben.

Füllen des TreeView-Steuerelements mit den Outlook-Ordner

Die Prozedur **FillTreeView** nimmt mit dem Parameter **objTreeView** einen Verweis auf das zu füllende **TreeView**-Steuerelement entgegen (siehe Listing 1). Sie referenziert eine neue oder die aktive Outlook-Instanz mit der Variablen **objOutlook** und weist der Variablen **objNamespace** dann mit der **GetNamespace**-Methode den MAPI-Namespace zu. Anschließend durchläuft sie alle **Folder**-Objekte, die direkt im MAPI-Namespace enthalten sind. Typischerweise sind das Ordner wie Outlook oder solche, welche nach der E-Mail-Adresse benannt sind, für die eine eigene **.pst**-Datei angelegt wurde. Diese **Folder**-Objekte durchläuft die Prozedur in einer **For Each**-Schleife, wobei sie das jeweilige Objekt mit der Variablen **objFolder** referenziert.

Innerhalb der Schleife prüft die Prozedur den Typ der standardmäßig in diesem Ordner enthaltenen Elemente. Die mit der Eigenschaft **DefaultItemType** ermittelte Einstellung soll in diesem Fall **olMailItem** lauten, da wir nur die Mailordner anzeigen wollen.

Handelt es sich um einen Ordner für E-Mails, legt die Prozedur für diesen ein neues **Node**-Element im **TreeView**-Steuerelement an und referenziert dieses mit der Variablen **objNode**. Das Anlegen erledigen wir mit der **Add**-Methode der **Nodes**-Auflistung des Steuerelements. Dieser übergeben wir für den dritten Parameter den Wert der Eigenschaft **FolderPath** des Ordners, also beispielsweise **//Outlook**. Dieser Parameter nimmt den Wert für die Eigenschaft **Key** auf. Über den Schlüssel legen wir einen eindeutigen Bezeichner für jedes Element im **TreeView**-Steuerelement fest. Daher eignet sich der Wert der Eigenschaft **FolderPath** ausgezeichnet – dieser ist für jeden Ordner in Outlook eindeutig. Der vierte Parameter legt den Wert der Eigenschaft **Name** fest. Hier übergeben wir auch den Namen des jeweiligen Ordners. Der fünfte Parameter schließlich erwartet die Angabe des Schlüssels

```
Private Sub FillNode_Rek(objTreeView As MSComctlLib.TreeView, objParentNode As MSComctlLib.Node, _
        objParentFolder As Outlook.Folder)
    Dim objFolder As Outlook.Folder
    Dim objNode As MSComctlLib.Node
    For Each objFolder In objParentFolder.Folders
        If objFolder.DefaultItemType = olMailItem Then
            Set objNode = objTreeView.Nodes.Add(objParentFolder.FolderPath, tvwChild, objFolder.FolderPath, _
                objFolder.Name, "folder_closed")
            FillNode_Rek objTreeView, objNode, objFolder
        End If
    Next objFolder
End Sub
```

Listing 2: Unterelemente des **TreeView**-Steuerelements füllen

für das für dieses Element anzuzeigende Icon aus dem **ImageList**-Steuerelement. Dieses soll für alle Elemente zunächst **folder_closed** lauten.

Damit füllen wir allerdings zunächst nur die oberste Ebene des **TreeView**-Steuerelements. Um die darunter liegenden Ebenen kümmern wir uns in einer weiteren Prozedur namens **FillNode_Rek**. Diese ist, wie der Name schon andeutet, eine rekursiv definierte Prozedur, die sich selbst aufruft. Der Prozedur übergeben wir Verweise auf das **TreeView**-Steuerelement, auf das soeben angelegte Node-Element sowie auf den Ordner, für den das Node-Element angelegt wurde.

Auf diese Weise durchlaufen wir alle Elemente der ersten Ebene des MAPI-Namespace, die **MailItem**-Elemente enthalten.

Rekursives Füllen für die untergeordneten Ordner

Die Prozedur **FillNode_Rek** ist eine rekursiv definierte Prozedur, die folgende Parameter erwartet:

- **objTreeView**: Verweis auf das zu füllende **TreeView**-Steuerelement
- **objParentNode**: Verweis auf das **Node**-Element, unter dem die neuen Elemente angelegt werden sollen
- **objParentFolder**: Outlook-Folder, dessen Unterordner dem **Node**-Element aus **objParentNode** hinzugefügt werden sollen.

Sie finden die Prozedur in Listing 2.

Die Prozedur definiert noch Elemente der Typen **Outlook.Folder** und **MSComctlLib.Node**, welche zum Durchlaufen der Unterordner des mit **objParentFolder** übergebenen Ordners und zum Anlegen der neuen **Node**-Elemente dienen.

In einer **For Each**-Schleife durchläuft die Prozedur alle **Folder**-Objekte des mit **objParentFolder** gelieferten Ordners. Darin prüft die Prozedur wieder, ob es sich bei dem Unterordner um einen E-Mail-Ordner handelt. Ist das der Fall, legt sie ein neues Element für diesen Ordner unterhalb des mit **objParentNode** übergebenen **Node**-Elements an. Dabei verwendet sie wieder die **Add**-Methode der **Nodes**-Auflistung des **TreeView**-Steuerelements aus **objTreeView**. Diesmal werden auch die ersten beiden Parameter der **Add**-Methode verwendet.

Der erste nimmt den **Key**-Wert des **Node**-Elements entgegen, zu dem das neue Element in einer bestimmten Relation angelegt werden soll (hier mit **objParentFolder.FolderPath** angegeben – wir hätten auch **objParentNode.Key** verwenden können). Diese Relation geben wir mit

SQL Server-Security – Teil 5: Rechte für Abteilungen

Bernd Jungbluth - Horn

Die aktuelle Rechtevergabe der Beispielapplikation erlaubt den Anwendern eine Anmeldung am SQL Server und den Zugang zur Datenbank WaWi_SQL. Innerhalb der Datenbank ist ihnen das Lesen und Schreiben der Daten sowie das Ausführen von Gespeicherten Prozeduren erlaubt. Ein solch pauschales Berechtigungskonzept beinhaltet viel zu viele Rechte. Den Anwendern stehen alle Daten zur Verfügung – sogar die Daten, die sie besser nicht lesen oder ändern sollten. Um das zu verhindern, bedarf es einer detaillierteren Rechtevergabe. Eine mögliche Variante zeigt Ihnen dieser Beitrag.

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!

Aktueller Stand

Das derzeitige Berechtigungskonzept basiert auf der SQL Server-Anmeldung **WaWiMa**. Mit dieser Anmeldung stellt die Access-Applikation **WaWi** die Verbindung zur SQL Server-Datenbank **WaWi_SQL** her. Ob nun eine Mitarbeiterin aus dem Vertrieb oder ein Mitarbeiter aus der Personalabteilung mit der Access-Applikation **WaWi** arbeitet, der Datenzugriff erfolgt immer über die gleiche Anmeldung. Die Zugriffsrechte werden folglich nicht über den Anwender, sondern über die Applikation gesteuert.

Welcher Anwender in der Applikation mit welchen Daten arbeiten darf, wird beim Start der Applikation bestimmt. Im Start-Formular ermittelt die VBA-Funktion **fBerechtigungen** den aktuellen Windows-Benutzer und aktiviert entweder die Schaltflächen für den Vertrieb oder die für die Personalverwaltung.

Auf diese Weise wird verhindert, dass ein Mitarbeiter des Vertriebs die Funktionen der Personalverwaltung nutzen kann. Allerdings schränkt dies nur den Zugriff auf die Funktionen ein, nicht aber den auf die Daten. Die Zugriffssteuerung findet ausschließlich im Start-Formular statt. Umgeht ein Anwender dieses Formular, stehen ihm alle Daten der Datenbank **WaWi_SQL** zur Verfügung. Dazu gehören auch die Daten der Personalverwaltung.

Das gilt es zu vermeiden. Dazu wird das Berechtigungskonzept um eine weitere Anmeldung im SQL Server ergänzt. Das Ziel ist eine Rechtevergabe auf Abteilungsebene.

Pro Abteilung eine Anmeldung

Das neue Berechtigungskonzept enthält zwei SQL Server-Anmeldungen, eine für die Installationen der Access-Applikation **WaWi** im Vertrieb und eine weitere für die Installationen in der Personalabteilung. Zur Umsetzung des neuen Berechtigungskonzepts ist zuerst eine Anmeldung im SQL Server anzulegen und der Datenbank **WaWi_SQL** zuzuordnen.

Der dabei erstellte Benutzer erhält in der Datenbank pauschale Lese- und Schreibrechte sowie das Recht zum Ausführen aller gespeicherten Prozeduren. Auf den Rechnern der Personalabteilung wird dann in den bestehenden ODBC-Datenquellen die neue Anmeldung eingetragen. Abschließend sind in der Access-Applikation die Tabellen neu einzubinden und die Verbindungsei-

genschaften der Pass-Through-Abfragen sowie die der ADO-Zugriffe anzupassen.

Auf den Rechnern des Vertriebs sind keine Änderungen notwendig. Die Verbindung von der Access-Applikation zur SQL Server-Datenbank erfolgt dort weiterhin über die Anmeldung **WaWiMa**. Allerdings wird diese in ihren Rechten eingeschränkt. Die Zugriffe auf die für die Personalverwaltung relevanten Tabellen und gespeicherten Prozeduren werden verweigert. Das Einschränken dieser Rechte findet im SQL Server statt.

Doch der Reihe nach. Als Erstes erstellen Sie die Beispielumgebung mit der SQL Server-Datenbank **WaWi_SQL**, der gleichnamigen ODBC-Datenquelle und der Access-Applikation **WaWi**. Die hierzu notwendigen Schritte sehen Sie in der Installationsanleitung am Ende dieses Beitrags.

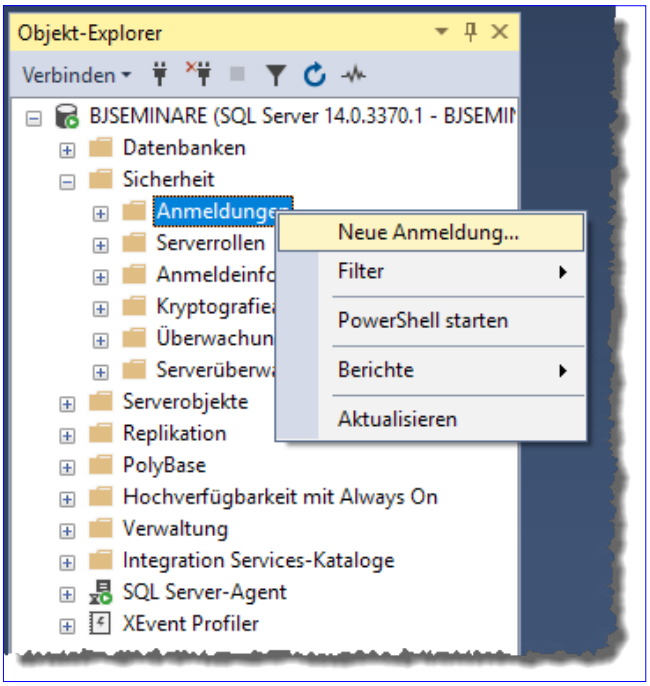


Bild 1: Der Weg zur neuen Anmeldung

Danach legen Sie die neue SQL Server-Anmeldung an. Öffnen Sie dazu das SQL Server Management Studio und verbinden Sie sich mit Ihrem SQL Server. Im Objekt-Explorer erweitern Sie den Ordner **Sicherheit** und wählen im Kontextmenü des Unterordners **Anmeldungen** den Eintrag **Neue Anmeldung** (siehe Bild 1). Im Dialog **Anmeldung - Neu** geben Sie der Anmeldung den Namen **WaWiPersonal** und aktivieren die Option **SQL Server-Authentifizierung**.

Das Kennwort zur Anmeldung tragen Sie in den beiden Eingabefeldern **Kennwort** und **Kennwort bestätigen** ein. Vergeben Sie ein gutes Kennwort. Eines, das nicht leicht zu erraten ist. Immerhin soll es die Daten der Personalverwaltung vor unbefugtem Zugriff schützen. Stellen Sie sich nur einmal die Diskussionen vor, wenn unerlaubterweise die Gehälter der Mitarbeiter veröffentlicht würden!

Die Option **Kennwortrichtline** unterstützt Sie bei der Vergabe des Kennworts. Ist die Option aktiviert, muss das Kennwort den in Ihrem Unternehmen gültigen Richtlinien zur Vergabe von Kennwörtern entsprechen. Sollten Sie keine Kennwortrichtlinien definiert haben, können Sie das Kennwort nach einem Muster erstellen oder Sie lassen sich eines in einem Passwort-Safe generieren. Ein mögliches Muster wie die Vor- und Nachteile beider Vorgehensweisen wurden im dritten Teil dieser Beitragsreihe beschrieben.

Kennwörter sollten Sie regelmäßig ändern. Bei aktivierter Option **Ablauf des Kennworts erzwingen** erinnert Sie der SQL Server gemäß Ihrer Kennwortrichtlinien an diese Empfehlung.

Diese Art der Erinnerung kann sich jedoch schnell kontraproduktiv auswirken. Ist das Kennwort abgelaufen, verlangt der nächste Anmeldeversuch die Vergabe eines neuen Kennworts. Ein neues Kennwort ist schnell vergeben.

Das ist aber nur ein Teil der Aufgabe. Danach sind noch alle Client-Applikationen, die diese Anmeldung verwenden, an das neue Kennwort anzupassen. Je nach Anzahl

der Client-Applikationen ist das schnell ein größerer Aufwand. Am besten deaktivieren Sie die Option **Ablauf des Kennworts erzwingen**. Das spart Ihnen unangenehme Überraschungen. Legen Sie sich stattdessen besser einen Termin an, der Sie an die Vergabe eines neuen Kennworts erinnert.

Das Entfernen des Häkchens in der Option **Ablauf des Kennworts erzwingen** deaktiviert sinnvollerweise ebenso die Option **Benutzer muss das Kennwort bei der nächsten Anmeldung ändern**.

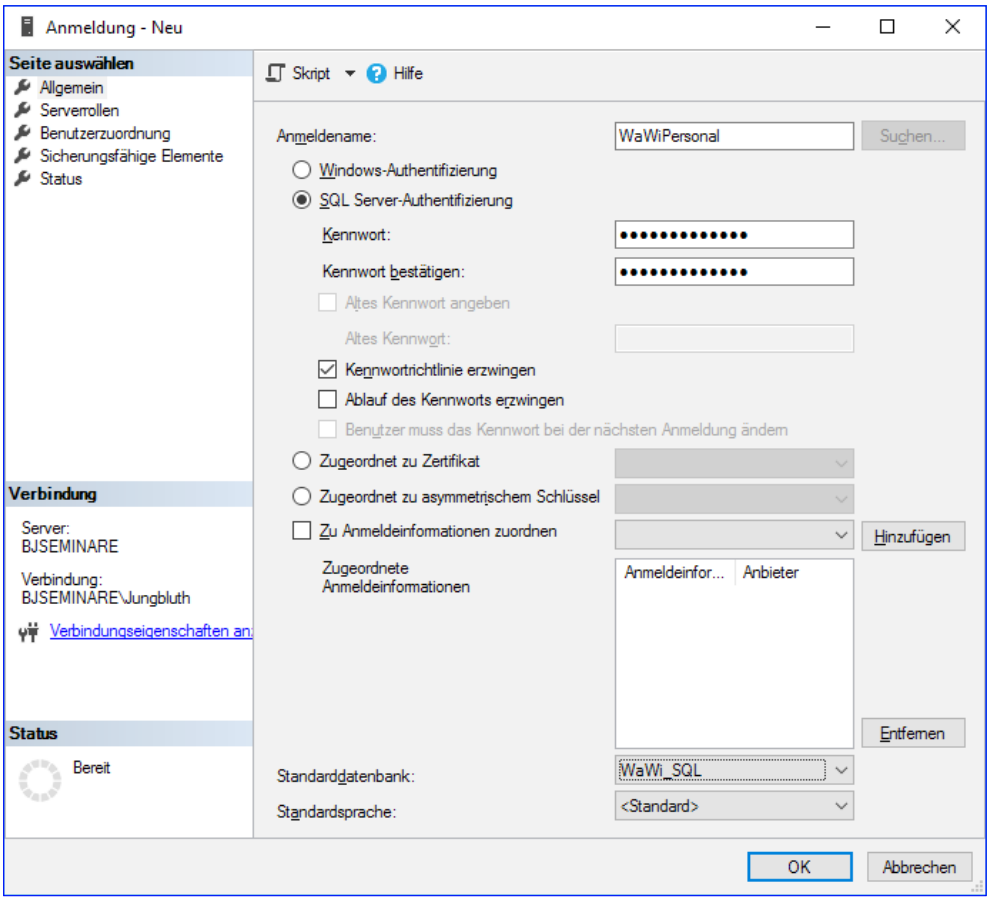


Bild 2: Die Anmeldung WaWiPersonal

Es macht keinen Unterschied, ob das Kennwort beim Ablauf oder bei der nächsten Anmeldung geändert werden muss. Mit der Kennwortänderung alleine ist es nicht getan. Es hat immer eine Anpassung der abhängigen Client-Applikationen zur Folge.

Als Nächstes legen Sie die Standarddatenbank der Anmeldung fest. Die Standarddatenbank wird verwendet, wenn beim Aufbau der Verbindung die Datenbank nicht explizit angegeben ist. Wählen Sie in der Auswahlliste **Standarddatenbank** den Eintrag **WaWi_SQL**.

Die Konfiguration der neuen Anmeldung wäre damit soweit abgeschlossen (siehe Bild 2). Dennoch sollten Sie nicht auf **OK** klicken. In der Seite **Benutzerzuordnung** können Sie jetzt nämlich direkt den zugehörigen Benutzer in der Datenbank **WaWi_SQL** anlegen und ihm die entsprechenden Rechte geben.

Wechseln Sie zur Seite **Benutzerzuordnung** und wählen Sie im oberen Bereich die Datenbank **WaWi_SQL** aus. Durch diese Auswahl wird der Benutzer **WaWiPersonal** in der Datenbank erstellt. Im unteren Bereich geben Sie dem Benutzer die Zugriffsrechte.

Aktivieren Sie hierzu die Systemdatenbankrollen **db_datareader** und **db_datawriter** plus die Benutzerdatenbankrolle **edb_execute** (siehe Bild 3). Diese Datenbankrolle wurde im vorheigen Beitrag dieser Reihe eingeführt (**SQL Server-Security, Teil 4: Schutz mit Datenbankrollen**, **www.access-im-unternehmen.de/1274**).

Sie beinhaltet die Rechte zur Ausführung aller gespeicherten Prozeduren. Um die Datenbankrolle **public** müssen Sie sich vorerst nicht kümmern. Hierbei handelt es sich um eine Standardzuordnung. Die Vorteile und vor

allem die Nachteile dieser Datenbankrolle lernen Sie in einem der nächsten Beiträge kennen.

Ein Klick auf **OK** erstellt zuerst die Anmeldung **WaWiPersonal** und dann in der Datenbank **WaWi_SQL** den zugehörigen Benutzer. Die Anmeldung sehen Sie im Unterordner **Anmeldungen** des Ordners **Sicherheit**. Den Benutzer finden Sie in der Datenbank **WaWi_SQL**. Dort gibt es ebenfalls einen Ordner namens **Sicherheit**. Dieser listet die Benutzer der Datenbank im Ordner **Benutzer** auf.

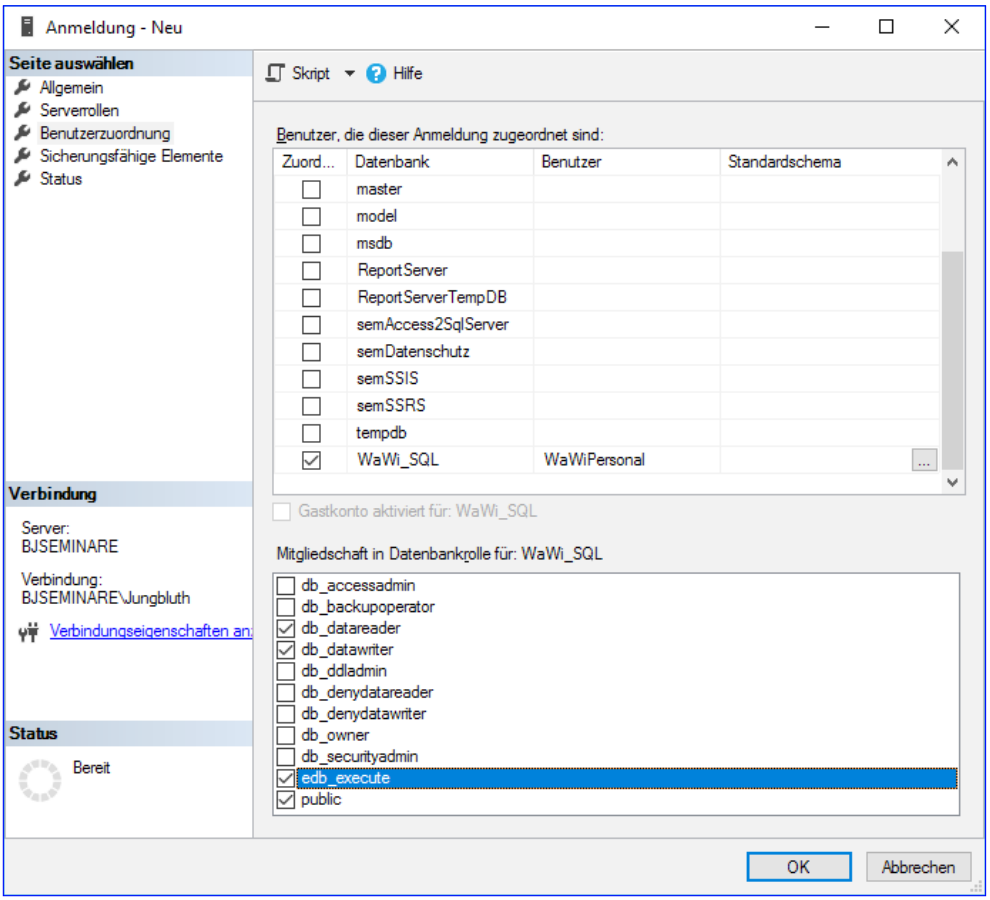


Bild 3: Der Benutzer WaWiPersonal

Das Berechtigungskonzept ist nun um eine Anmeldung reicher. Mit dieser Anmeldung authentifiziert sich der Anwender am SQL Server. Die Anmeldung ist in der Datenbank **WaWi_SQL** fest mit dem gleichnamigen Benutzer verbunden.

Diese Zuordnung autorisiert die Anmeldung und somit den Anwender für den Zugang zur Datenbank. Die Rechte für den Datenzugriff sind am Benutzer definiert. Dieser ist aktuell den Datenbankrollen **db_datareader**, **db_datawriter** und **edb_execute** zugeordnet. Der Anwender darf mit diesen Rechten in der Datenbank jegliche Daten lesen und schreiben sowie alle gespeicherten Prozeduren ausführen.

Soweit in Kurzform die Beschreibung der Anmeldung **WaWiPersonal** innerhalb der mehrstufigen Sicherheitsarchitektur von SQL Server.

Das ist aber noch nicht alles. Schließlich erfolgt der Anmeldevorgang am SQL Server aktuell nicht durch den Anwender, sondern automatisch in der Access-Applikation **WaWi**. Die muss jetzt noch an die neue Anmeldung angepasst werden.

Eine Anmeldung für die Personalabteilung

In der Realität würden Sie nun in die Personalabteilung gehen und dort auf den einzelnen Rechnern die Access-Applikation mitsamt der ODBC-Datenquelle anpassen.

Um dieses Szenario nachzustellen, kopieren Sie die Access-Applikation **WaWi** und geben der Kopie den Namen **WaWi Personal**. Als nächstes ändern Sie die ODBC-Datenquelle **WaWi_SQL**. Den ODBC-Datenquellen-Administrator starten Sie am besten über die Windows-Suche. Dazu drücken Sie die Windows-Taste und tippen **ODBC**.

Das Suchergebnis liefert Ihnen zwei Einträge. 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 **System-DSN**, markieren dort die ODBC-Datenquelle **WaWi_SQL** und klicken auf **Konfigurieren**. Den ersten Schritt des Assistenten überspringen Sie mit der Schaltfläche **Weiter**.

Im zweiten Schritt passen Sie die Anmeldedaten an. Geben Sie im Eingabefeld **Anmelde-ID** den Namen der neuen Anmeldung und im Feld **Kennwort** das zugehörige Kennwort ein (siehe Bild 4). Danach bestätigen Sie diesen sowie den nächsten Schritt des Assistenten mit einem Klick auf **Weiter** und beenden die Konfiguration im letzten Schritt mit der Schaltfläche **Fertig stellen**.

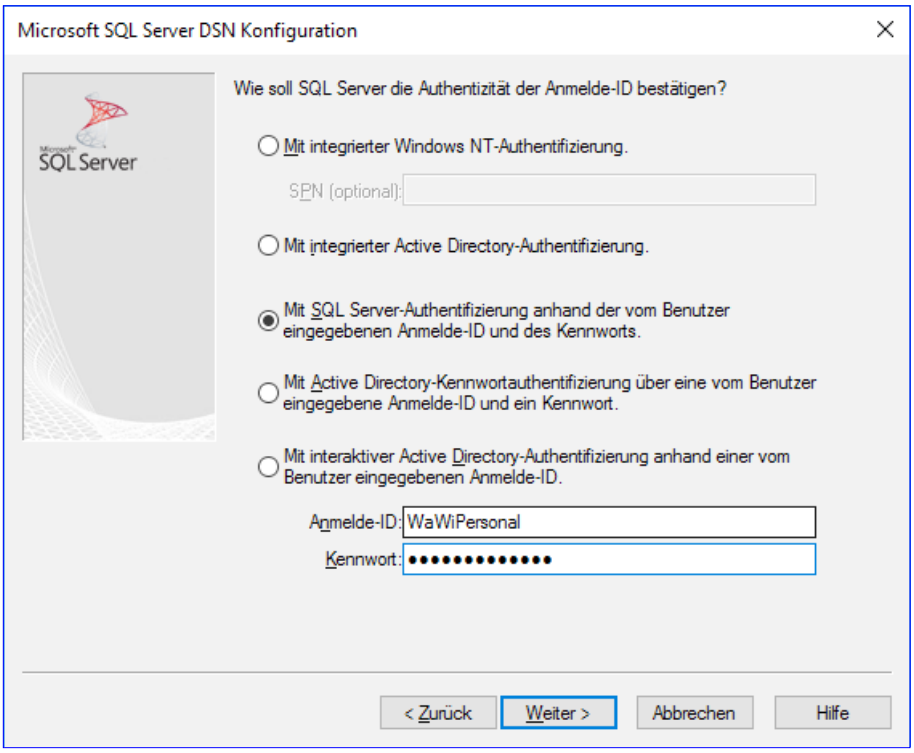


Bild 4: Ändern der ODBC-Datenquelle

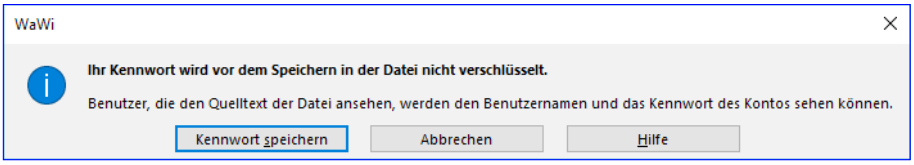


Bild 5: Speichern des Kennworts

Es folgt eine Zusammenfassung der Konfiguration und die Möglichkeit, die neuen Anmeldedaten zu testen. Sollte das Ergebnis nach einem Klick auf **Datenquelle testen** nicht erfolgreich sein, haben Sie sich vielleicht beim Kennwort vertippt. In diesem Fall wiederholen Sie die eben beschriebenen Schritte. Nach einem erfolgreichen Test ist die Änderung der ODBC-Datenquelle abgeschlossen.

Soweit so gut. Die ODBC-Datenquelle verwendet nun die Anmeldung **WaWiPersonal**. Jetzt passen Sie noch die Access-Applikation an die neue Anmeldung an. In der Realität wäre dies auf jedem Rechner der Personalabteilung erforderlich. In der Beispielumgebung öffnen Sie dazu die eben kopierte Access-Applikation **WaWi Personal**. Dort sind die Tabellen noch mit der Anmeldung **WaWiMa** eingebunden.

Aus diesem Grund entfernen Sie alle eingebundenen Tabellen und binden diese neu ein. Starten Sie über **Externe Daten – Neue Datenquelle – Aus Datenbank – Aus SQL Server** den Dialog **Externe Daten - ODBC-Datenbank** und aktivieren Sie dort die zweite Option.

Mit einem Klick auf **OK** kommen Sie zum nächsten Dialog, in dem Sie in der Registerkarte **Computerdatenquelle** die ODBC-Datenquelle **WaWi_SQL** per

Doppelklick auswählen. Nun geben Sie das Kennwort zur Anmeldung **WaWiPersonal** ein und schließen den Vorgang mit **OK** ab.

Als Ergebnis sehen Sie im Dialog **Tabellen verknüpfen** alle Tabellen der SQL Server-Datenbank **WaWi_SQL**. Aktivieren Sie die Option **Kennwort speichern** und markieren Sie die Tabellen mit dem Präfix **dbo**.

Ein Klick auf **OK** übernimmt die Auswahl und liefert Ihnen vor dem Einbinden jeder Tabelle eine Meldung (siehe Bild 5). Bestätigen Sie diese Meldungen mit der Schaltfläche **Kennwort speichern**, um die Anmeldedaten für den Datenzugriff in der Access-Datenbank zu hinterlegen.

Die frisch eingebundenen Tabellen sind nun im Navigationsbereich zu sehen. Allerdings enthalten die Tabellen den Präfix **dbo_**, was nicht den ursprünglich eingebundenen Tabellen entspricht. Dies korrigieren Sie mit der VBA-Funktion **fTabellenUmbenennen**. Öffnen Sie mit der Tastenkombination **Strg + G** den VBA-Direktbereich und führen Sie dort die Funktion aus.

Die zwei Pass Through-Abfragen der Access-Applikation verwenden aktuell noch die Anmeldung **WaWiMa**. Mit der VBA-Funktion **fPassThroughVerbindungen** übertragen Sie die Anmeldedaten der eingebundenen Tabellen in die Verbindungszeichenfolgen der Pass Through-Abfragen. Führen Sie diese VBA-Funktion ebenfalls über den VBA-Direktbereich aus.

Es fehlt noch die Konfiguration für den Datenzugriff per ADO. Die hierzu notwendigen Daten sind in der lokalen Tabelle **ADO** gespeichert. Diese Tabelle ist als Systemtabelle deklariert.

Um die Daten der Tabelle ändern zu können, müssen Sie die Deklaration zunächst entfernen. Dies übernimmt die VBA-Funktion **fTabelleAlsSystemObjekt**, die Sie im VBA-Direktbereich wie folgt ausführen:

```
fTabelleAlsSystemObjekt "ADO", False
```

Der Navigationsbereich enthält jetzt die Tabelle **ADO**. Öffnen Sie die Tabelle und passen Sie die Anmeldedaten in den Feldern **Benutzer** und **Kennwort** an die Anmeldung **WaWiPersonal** an. Um die Tabelle wieder zu verstecken, ändern Sie im VBA-Direktbereich den Parameterwert **False** in **True** und führen Sie die VBA-Funktion erneut aus.

Mit dem Formular **Start** können Sie die neue Anmeldung testen. Ein Klick auf die Schaltfläche **Mitarbeiter** liefert ein Formular mit den Personaldaten. So soll es sein. Jetzt öffnen Sie noch das Formular **Ansprechpartner**. Es zeigt Ihnen wie erwartet die Kontaktinformationen der Ansprechpartner. Der Test ist erfolgreich.

Mit den Rechten der Anmeldung **WaWiPersonal** dürfen Sie in der Datenbank **WaWi_SQL** die Daten aller Tabellen lesen und ändern sowie alle gespeicherten Prozeduren ausführen.

Somit wäre es den Mitarbeitern der Personalabteilung möglich, mit ihrer Version der Access-Applikation **WaWi** auf die Daten des Vertriebs zuzugreifen. Natürlich könnte man dies mit einer weiteren Rechtevergabe verhindern. Warum aber sollte man den Aufwand betreiben? Es ist bestimmt kein Sicherheitsrisiko, wenn ein Mitarbeiter der Personalabteilung die Kontaktinformationen der Ansprechpartner lesen kann.

Diese Argumentation sollten Sie mal mit Ihrem Datenschutzbeauftragten diskutieren. Er wird einige Einwände haben. Als Stichpunkte seien hier Zweckbindung und Privacy By Default genannt. Doch das ist ein Thema für einen späteren Beitrag.

Bis jetzt bringt die Anmeldung **WaWiPersonal** noch keine Vorteile. Schließlich unterscheiden sich die Rechte dieser Anmeldung nicht von denen der Anmeldung **WaWiMa**. Ziel ist es, den Mitarbeitern des Vertriebs den

Zugriff auf die Personaldaten zu verweigern. Dazu ändern Sie die Rechtevergabe zur Anmeldung **WaWiMa**.

Eine Anmeldung für den Vertrieb

Die jetzt anstehenden Änderungen der Rechtevergabe sind überschaubar. Sie müssen weder die installierte Access-Applikation noch die ODBC-Datenquelle anpassen. Alle erforderlichen Schritte finden im SQL Server statt. Öffnen Sie dazu das SQL Server Management Studio.

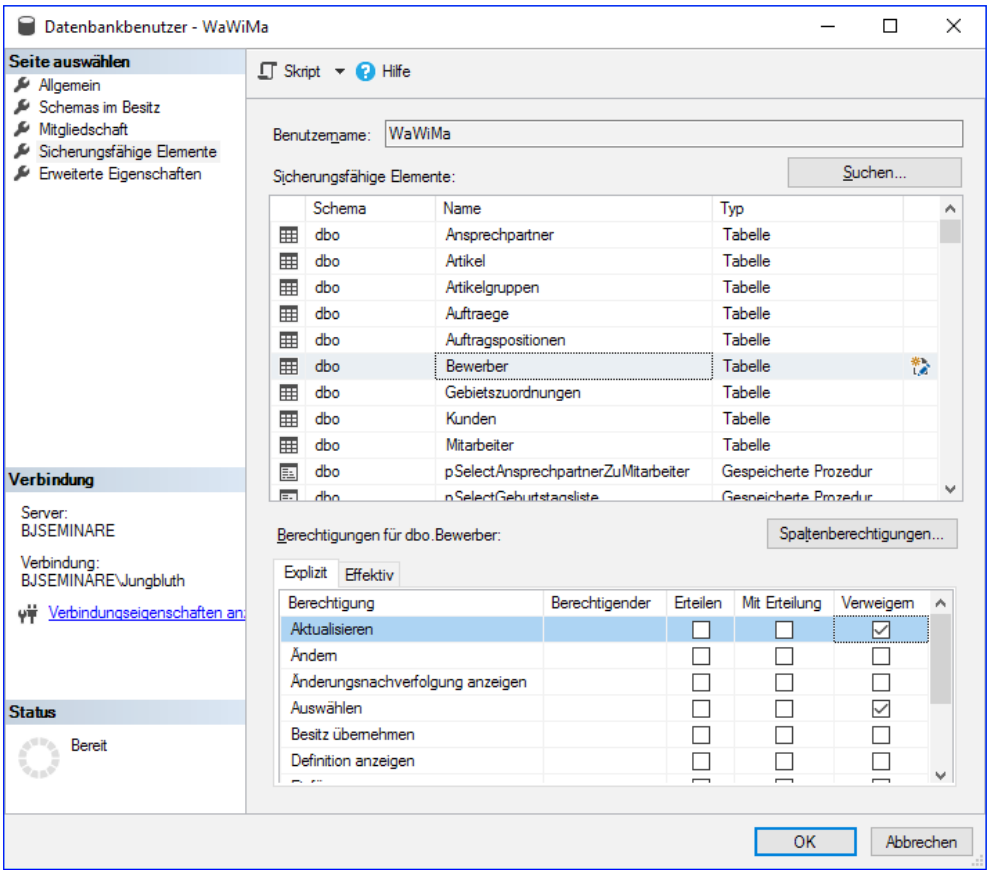


Bild 6: Verweigerter Zugriff auf eine Tabelle

Die Rechte für den Datenzugriff werden nicht an der Anmeldung **WaWiMa**, sondern an dem zugehörigen Benutzer in der Datenbank **WaWi_SQL** definiert. Diesen finden Sie im Objekt-Explorer unter **Datenbanken – WaWi_SQL – Sicherheit – Benutzer**. Mit einem Doppelklick auf den Eintrag **WaWiMa** öffnen Sie den Dialog **Datenbankbenutzer**.

Aktuell ist der Benutzer **WaWiMa** den Datenbankrollen **db_datareader**, **db_datawriter** und **edb_execute** zugeordnet. Dies erlaubt ihm den lesenden und schreibenden Zugriff auf alle Daten der Datenbank sowie das Ausführen aller gespeicherten Prozeduren.

Sie sehen diese Rechtevergabe auf der Seite **Mitgliedschaft**. Um die relevanten Tabellen und gespeicherten Prozeduren für die Personaldaten von diesem pauschalen Zugriff auszuschließen, müssen Sie diesen Datenbankobjekten gesonderte Rechte vergeben.

Die Rechtevergabe an einzelnen Datenbankobjekten findet in der Seite **Sicherungsfähige Elemente** statt. Dort legen Sie mit der Schaltfläche **Suchen** zunächst eine Vorauswahl fest. Ein Klick auf diese Schaltfläche liefert Ihnen den Dialog **Objekte hinzufügen**. Hier aktivieren Sie die Option **Alle Objekte des Typs** und klicken auf **OK**.

Im nächsten Dialog entscheiden Sie sich für die Objekttypen **Tabellen** und **Gespeicherte Prozeduren**. Nach Bestätigen dieser Auswahl sind Sie wieder zurück im Dialog **Datenbankbenutzer** und sehen im oberen Bereich alle Tabellen und gespeicherten Prozeduren der Datenbank.

Markieren Sie die Tabelle **Bewerber** und aktivieren Sie im unteren Bereich die Option **Verweigern** in den Zeilen

E-Mails verwalten mit dem Outlook View Control

Im Beitrag »Outlook-Folder in Access anzeigen« haben wir gezeigt, wie Sie das Outlook View Control in ein Formular integrieren, um damit die Ordner von Outlook anzuzeigen. Im vorliegenden Beitrag bauen wir auf den dort vorgestellten Techniken auf und gehen genauer auf den Umgang mit dem E-Mail-Ordnern ein. Dabei wollen wir Details wie den Betreff, den Inhalt oder den Empfänger oder Absender der aktuell markierten E-Mail in entsprechenden Steuerelementen anzeigen. Außerdem wollen wir die Anzeige der Outlook-Ordner in ein TreeView-Steuerelement verlagern.

Das **Outlook View Control** ist eine tolle Erweiterung, wenn Sie beispielsweise die Outlook-Ordner zur Anzeige von E-Mails wie **Posteingang**, **Postausgang**, **Gesendete** Elemente oder auch benutzerdefinierte Ordner in einer Access-Anwendung anzeigen wollen.

Das **Outlook View Control** können Sie jedoch nicht einfach zu einem Access-Formular hinzufügen, sondern Sie müssen es in ein **Frame**-Steuerelement einbetten. Wie das gelingt, zeigen wir im Beitrag **Outlook-Folder in Access anzeigen** (**www.access-im-unternehmen.de/1292**).

In diesem Beitrag haben wir gezeigt, wie Sie die Outlook-Ordner in ein Listenfeld einlesen und ihre Inhalte durch Anklicken des jeweiligen Listeneintrags im **Outlook View Control** anzeigen.

Da die Anzeige in einem Listenfeld wenig professionell aussieht, haben wir in einem weiteren Beitrag namens **TreeView für Outlook-Ordner** ein **TreeView**-Steuerelement zum Formular hinzugefügt, das alle Ordner mit E-Mails anzeigt (**www.access-im-unternehmen.de/1293**). Der Zwischenstand nach diesem Beitrag sieht wie in Bild 1 aus.

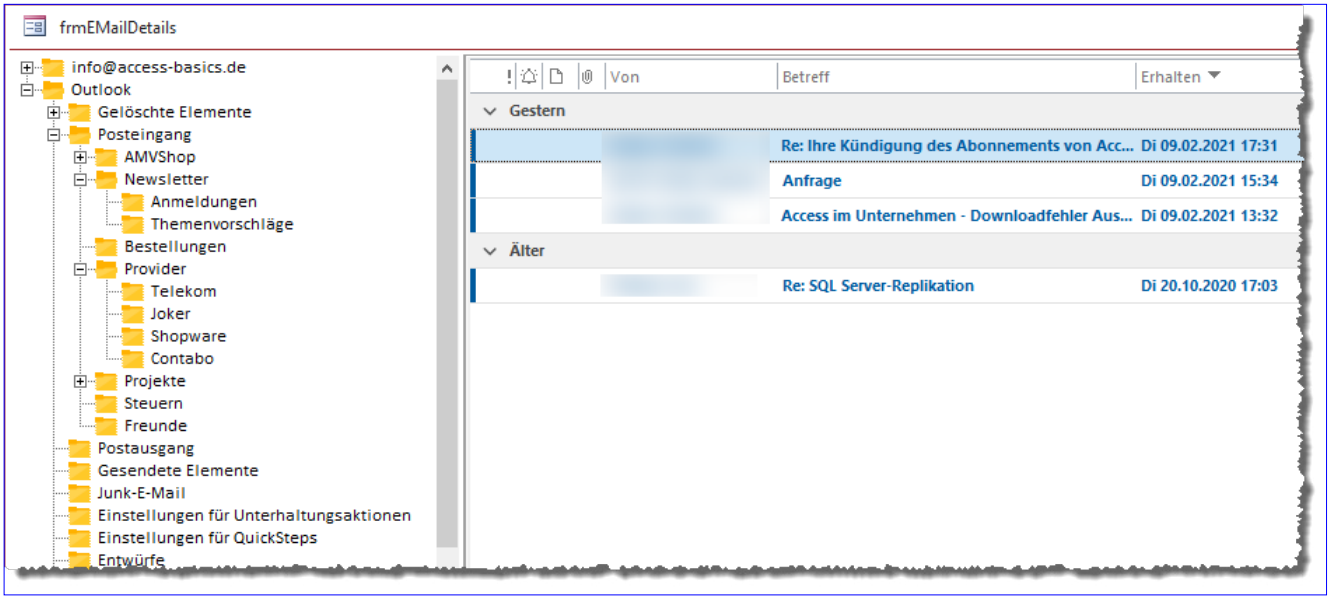


Bild 1: Outlook-Ordner und der Inhalt eines E-Mail-Ordners

Damit kommen wir zum Ziel des vorliegenden Beitrags. Hier wollen wir die übrigen Elemente, die für den Umgang mit E-Mails notwendig sind, hinzufügen.

Dazu gehören die folgenden:

- Gewählten Ordner aus dem **TreeView**-Steuerelement im **Outlook View Control** anzeigen
- Zuletzt angezeigten Ordner speichern und wiederherstellen
- Inhalt der aktuell markierten E-Mail anzeigen (Absender, Empfänger, Betreff, Inhalt, Eingangsdatum)
- Empfangen von E-Mails
- Erstellen einer neuen E-Mail im Outlook-Inspektor
- Öffnen einer E-Mail mit dem dazugehörigen Outlook-Inspektor
- Löschen von E-Mails

Gewählten Ordner im Outlook View Control anzeigen

Das Formluar **frmEMailDetails** enthält das **TreeView**-Steuerelement namens **ctlTreeView** sowie ein **Frame**-Objekt und ein **ViewCtl**-Objekt. Das **ViewCtl**-Objekt ist in das **Frame**-Objekt eingebettet.

Das **Frame**- und das **ViewCtl**-Objekt deklarieren wir im Kopf des Klassenmoduls des Formulars wie folgt:

```
Private objFrame As Frame
Private WithEvents objView As ViewCtl
```

In der beim Laden des Formulars ausgelösten Prozedur **Form_Load** initialisieren wir das **TreeView**-Steuerelement (siehe Beitrag **TreeView für Outlook-Ordner**) und weisen den beiden Variablen die Elemente zu:

```
Private Sub Form_Load()
    InitializeTreeView
    Set objFrame = Me!ctlFrame.Object
    Set objView = objFrame.Controls(0)
End Sub
```

Nun wollen wir dafür sorgen, dass der vom Benutzer im **TreeView**-Steuerelement angeklickte Ordner im **Outlook View Control** angezeigt wird. Dazu legen wir eine Ereignisprozedur an, die beim Anklicken eines der Elemente des **TreeView**-Steuerelements ausgelöst wird:

```
Private Sub ctlTreeView_NodeClick(ByVal Node As Object)
    Dim objNode As MSComctlLib.Node
    Set objNode = Node
    objView.Folder = objNode.Key
End Sub
```

Das wir das mit dem Parameter **Node** übergebene **Node**-Objekt noch einer Variablen des Typs **MSComctlLib.Node** zuweisen, liegt daran, dass wir so die IntelliSense-Funktion für diese Variable nutzen können. So können wir den in der **Key**-Eigenschaft des **Node**-Elements gespeicherten Outlook-Pfad wie zum Beispiel **\\Outlook\Posteingang** nutzen und diesen der **Folder**-Eigenschaft des **Outlook View Control**-Steuerelements zuweisen.

Zuletzt gewählten Ordner speichern und beim Öffnen wiederherstellen

Nun, da das **Outlook View Control** den Outlook-Ordner anzeigt, den der Benutzer im **TreeView**-Steuerelement ausgewählt hat, wollen wir diesen auch speichern, damit dieser beim nächsten Öffnen auch wieder angezeigt werden kann. Dazu legen wir eine Optionen-Tabelle namens **tblOptions** an. Diese soll lediglich zwei Felder enthalten – ein Primärschlüsselfeld namens **OptionID** und ein Textfeld namens **CurrentMailfolder**. Den Entwurf dieser Tabelle zeigen wir in Bild 2.

Wann speichern wir den zuletzt gewählten Outlook-Ordner in der Optionen-Tabelle? Am sichersten ist es, diesen nach

jeder Auswahl erneut zu speichern. Wir können den Speichervorgang also direkt in die oben vorgestellte Ereignisprozedur integrieren, die beim Anklicken eines der **Node**-Elemente im **TreeView**-Steuerelement ausgelöst wird. Diese erweitern wir um den Aufruf einer Prozedur namens **SaveCurrentMailfolder**, der wir den Pfad zum angeklickten Ordner als Parameter übergeben:

```
Private Sub ctlTreeView_NodeClick(ByVal
Node As Object)
    ...
    SaveCurrentMailFolder objNode.Key
End Sub
```

Die Prozedur **SaveCurrentMailfolder** finden Sie in Listing 1. Sie nimmt mit dem Parameter **strFolder** den zuletzt aufgerufenen Ordner entgegen. Dann führt sie eine **UPDATE**-Abfrage aus, die das Feld **CurrentMailfolder** der Tabelle **tblOptions** auf den übergebenen Wert einstellt.

Es kann sein, dass die Tabelle leer ist, weil Sie diese beispielsweise vor dem Weitergeben an einen Benutzer geleert haben. In diesem Fall liefert die folgende Abfrage der von der **UPDATE**-Abfrage betroffenen Datensätze den Wert **0**. In dem dann ausgelösten Zweig der **If...Then**-Bedingung legen wir diesen Datensatz dann einfach neu an und weisen dem Feld **CurrentMailfolder** den Wert aus **strFolder** zu.

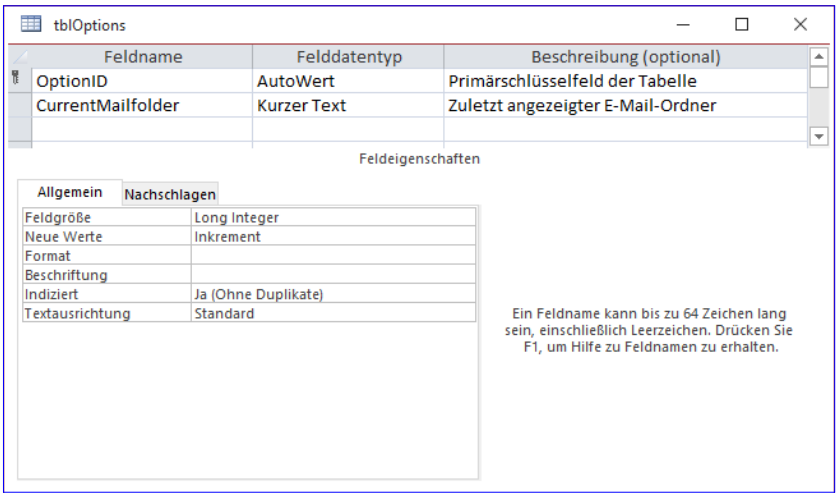


Bild 2: Tabelle zum Speichern von Optionen

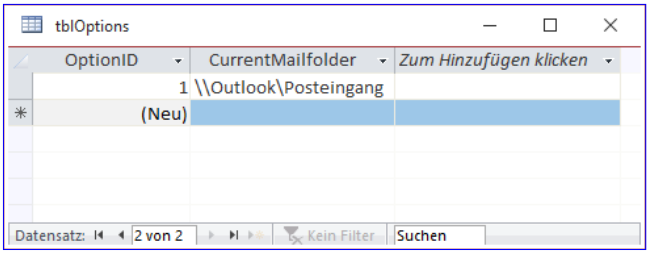


Bild 3: Zuletzt verwendeter Ordner in der Tabelle **tblOptions**

Nach dem Speichern des Wertes sieht die Tabelle **tblOptions** beispielsweise wie in Bild 3 aus.

Zuletzt gewählten Ordner beim Öffnen markieren und anzeigen

Die so gespeicherte Information wollen wir natürlich beim nächsten Öffnen des Formulars nutzen, indem wir den entsprechenden Eintrag im **TreeView**-Steuerelement selektieren und im **Outlook View Control** den passenden

```
Private Sub SaveCurrentMailfolder(strFolder As String)
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "UPDATE tblOptions SET CurrentMailfolder = '" & strFolder & "'", dbFailOnError
    If db.RecordsAffected = 0 Then
        db.Execute "INSERT INTO tblOptions(CurrentMailfolder) VALUES('" & strFolder & "')", dbFailOnError
    End If
End Sub
```

Listing 1: Prozedur zum Speichern des aktuell gewählten Mail-Ordners

Ordner anzeigen. Dazu legen wir eine neue Prozedur namens **SetCurrentMailfolder** an, den wir in der Prozedur **Form_Load** beim Laden des Formulars aufrufen:

```
Private Sub Form_Load()
    ...
    SetCurrentMailfolder
End Sub
```

```
Private Sub SetCurrentMailfolder()
    Dim strFolder As String
    Dim objTreeView As MSComctlLib.TreeView
    strFolder = Nz(DLookup("CurrentMailfolder", "tblOptions"), "")
    If Len(strFolder) = 0 Then
       strFolder = GetDefaultFolder(olFolderInbox)
    End If
    objView.Folder = strFolder
    Set objTreeView = ctlTreeView.Object
    objTreeView.SelectedItem = objTreeView.Nodes(strFolder)
    Me!ctlTreeView.SetFocus
End Sub
```

Listing 2: Prozedur zum Auswählen des zuletzt betrachteten E-Mail-Ordners

Diese Prozedur liefert Listing 2. Die Prozedur ermittelt zunächst den Wert des Feldes **CurrentMailFolder** aus der Tabelle **tblOptions** und schreibt diesen in die Variable **strFolder**. Wenn noch kein Eintrag in dieser Tabelle vorliegt oder das Feld leer ist, sorgt die **Nz**-Funktion dafür, dass **strFolder** mit einer leeren Zeichenkette gefüllt wird.

```
Private Function GetDefaultFolder(intFolder As Outlook.OlDefaultFolders) As String
    Dim objOutlook As Outlook.Application
    Dim objNamespace As Outlook.Namespace
    Dim objFolder As Outlook.Folder
    Set objOutlook = New Outlook.Application
    Set objNamespace = objOutlook.GetNamespace("MAPI")
    Set objFolder = objNamespace.GetDefaultFolder(intFolder)
    GetDefaultFolder = objFolder.FolderPath
End Function
```

Listing 3: Funktion zum Ermitteln eines Standardordners

Die folgende **If...Then**-Bedingung prüft, ob **strFolder** eine leere Zeichenkette enthält und somit kein zuletzt geöffneter E-Mail-Ordner in der Tabelle **tblOptions** vorliegt. In diesem Fall füllt sie **strFolder** mit dem Ergebnis der Funktion **GetDefaultFolder**. Diese finden Sie in Listing 3.

GetDefaultFolder erwartet eine der Konstanten der Auflistung **Outlook.OlDefaultFolders** als Parameter. Es gibt bereits eine Methode namens **GetDefaultFolder**, die zum **Namespace**-Objekt gehört.

Um dieses zu referenzieren, benötigen wir zuvor noch ein **Outlook.Application**-Objekt. Die von uns definierte Funktion **GetDefaultFolder** vereinfacht den Zugriff auf einen der Standardordner von Outlook, indem es das Erstellen einer Instanz von **Outlook.Application** und das Zuweisen des MAPI-Namespace an die Variable **objNamespace** kapselt und das **Folder**-Objekt ermittelt, das dem Parameter **intFolder** entspricht, also beispielsweise **olFolderInbox** für den Posteingang. Die Funktion **GetDefaultFolder** gibt schließlich den Pfad zu dem angegebenen Ordner zurück.

Dadurch, dass wir den Parameter mit dem Typ **Outlook.OlDefaultFolders** deklarieren, können wir beim Aufruf per IntelliSense aus der Liste der verfügbaren Werte auswählen (siehe Bild 4).

Diesen nimmt die Prozedur **SetCurrentMailFolder** entgegen und speichert ihn in der Variablen **strFolder**. Damit stellt sie nun zunächst die **Folder**-Eigenschaft des **Outlook View Controls** ein. Danach referenziert sie das **TreeView**-Steuerelement mit der Variablen **objTreeView**

und stellt die Eigenschaft **SelectedItem** auf das **Node**-Element ein, dass die **Nodes**-Auflistung für den als Parameter verwenden Wert aus **strFolder** (zum Beispiel **\\Outlook\Posteingang**) liefert.

Schließlich wird der Fokus auf das **TreeView**-Steuerelement verschoben, damit der Benutzer direkt den selektierten Ordner erkennen kann.

Müssen wir die übergeordneten Ordner aufklappen, damit der markierte Eintrag sichtbar wird? Es kann ja auch sein, dass der Benutzer einen Ordner auswählt und dann alle Ordner einklappt, sodass der gewählte Ordner nicht mehr im **TreeView**-Steuerelement sichtbar ist.

Hier besteht kein Grund zur Sorge: Das **TreeView**-Steuerelement klappt automatisch alle Elemente bis zum aktuell selektierten Element auf.

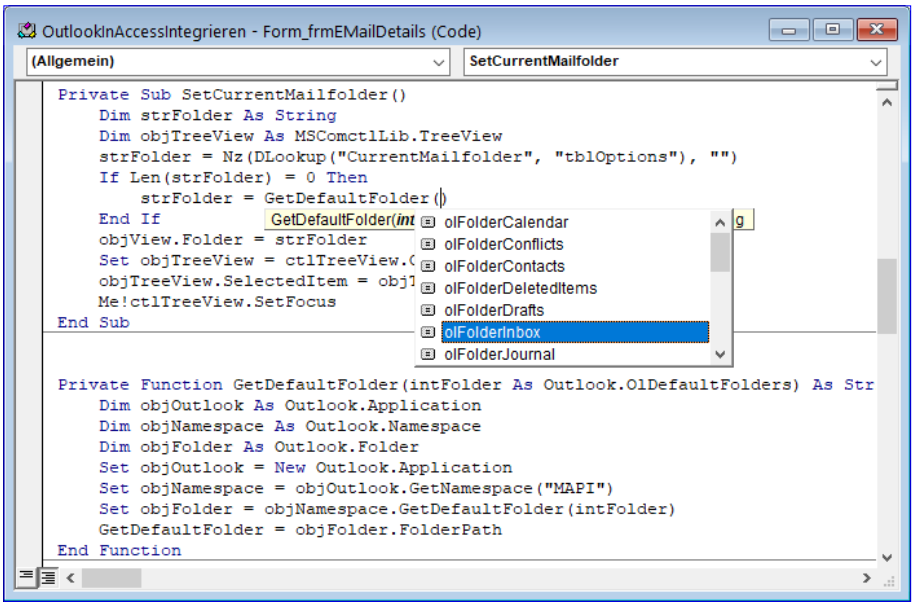


Bild 4: Auswahl des gewünschten Standardordners per IntelliSense

Wir wollen uns dabei auf folgende Elemente beschränken:

- Absender
- Empfänger

Inhalt der aktuell markierten E-Mail anzeigen

Nun wollen wir die Inhalte der aktuell im **Outlook View Control** ausgewählten E-Mail in dafür vorgesehenen Steuerelementen anzeigen.

In Outlook sieht der Bereich wie in Bild 5 aus. Die Ansicht im **Outlook View Control** liefert leider nur den oberen Teil mit der Liste der E-Mails, sodass wir den Rest selbst anlegen müssen.

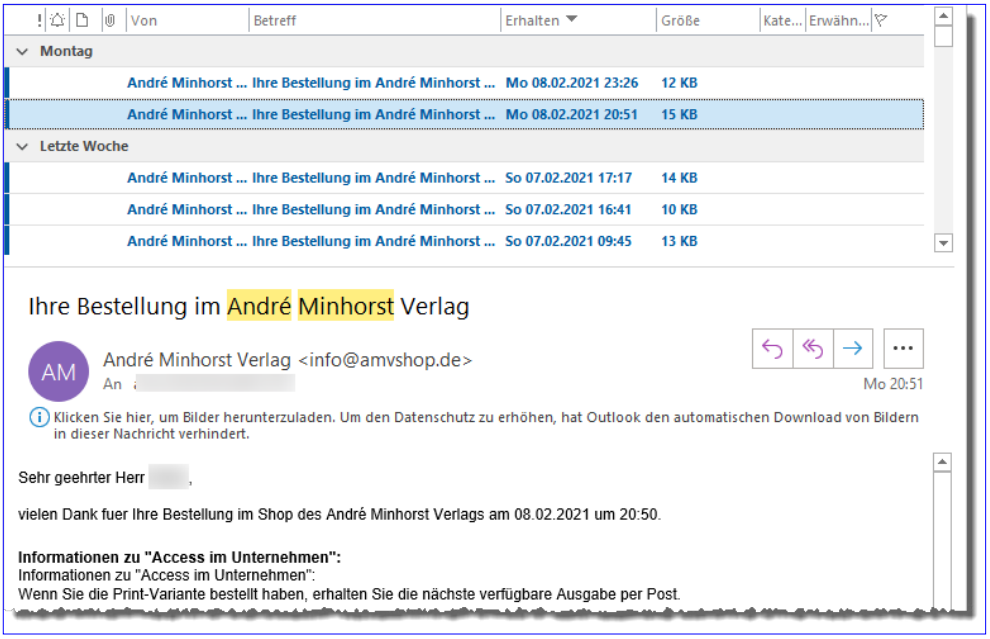


Bild 5: Anzeige einer E-Mail in Outlook

HAUFE.
Ausgabe 03/2021

ACCESS
IM UNTERNEHMEN

PROGRAMMIEREN MIT TWINBASIC

Eine neue Möglichkeit, COM-Add-Ins und DLL-Funktionsbibliotheken zu programmieren – auf Basis von VB und VBA! (ab S. 7)

In diesem Heft:

www.access-im-unternehmen.de
Mat-Nr. H1583-5125

ARCHIV SEITE 157

twinBASIC – Perfekte Ergänzung

Die Entwicklungsumgebung von Microsoft Access wurde seit 2010 nicht mehr aktualisiert, der VBA-Editor noch länger nicht mehr. Gleiches gilt für VB6: Visual Studio 6 wurde zwar durch Visual Studio .NET mit den entsprechenden Programmiersprachen wie Visual Basic.NET oder C# ersetzt, aber wer bestehende Projekte hatte, wurde von Microsoft im Regen stehen gelassen. Dem schafft nun der Programmierer Wayne Philips Abhilfe, indem er mit twinBASIC eine tolle neue Möglichkeit zur Programmierung mit VB schafft.

Gerade in diesen Zeiten, wo immer mehr Office-Pakete in der 64-Bit-Variante installiert werden, macht sich Wehmut breit: Liebgewonnene DLLs, die mit VB6 erstellt wurden, lassen sich damit nicht mehr nutzen. Und auch Elemente der Benutzeroberfläche von Drittherstellern, die von ihren Entwicklern nicht nach 64-Bit migriert wurden, müssen ersetzt werden.

Da kommt es doch gelegen, dass ein in Access-Kreisen schon recht bekannter Entwickler namens Wayne Philips sich aufmacht, eine neue Möglichkeit zum Erstellen von VB-DLLs und -Anwendungen zu schaffen, die ohne die Tools von Microsoft auskommt. Genau genommen hat Wayne eine Erweiterung namens **twinBASIC** programmiert für die Entwicklungsumgebung Visual Studio Code. Das ist wiederum ein kostenloser Code-Editor.

Was Sie schon jetzt, wenige Wochen nach der Erstveröffentlichung mit twinBASIC alles erledigen können, schauen wir uns im Detail in dieser Ausgabe an. Im ersten Beitrag namens **twinBASIC – VB/VBA mit moderner Umgebung** schauen wir uns ab Seite 7 an, wie Sie die Tools installieren und welche Möglichkeiten es bietet.

Der zweite Beitrag zum Thema namens **COM-DLLs mit twinBASIC** zeigt ab Seite 15, wie Sie mit DLLs programmieren können, mit denen Sie beispielsweise wichtige Funktionen kapseln und dann über die DLL in allen Datenbankanwendungen verfügbar machen können.

Da auch beim Programmieren mit twinBASIC Fehler anfallen können, zeigen wir im Beitrag **Debugging in twinBASIC**, welche Debugging-Möglichkeiten twinBASIC in Visual Studio Code anbietet (ab S. 22).

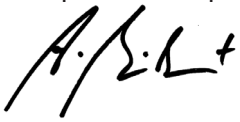
Nach dem Aufwärmen geht es an die richtig interessanten Themen: Wir erstellen mit twinBASIC ein COM-Add-In für Access – und damit können wir tatsächlich sehr komfortabel die Access-Benutzeroberfläche um eigene Funktionen erweitern! Wie das im Detail gelingt, lesen Sie unter dem Titel **twinBASIC: COM-Add-Ins für Access** ab S. 28.

Das Gleiche können Sie praktischerweise mit dem gleichen Tool auch für den VBA-Editor durchführen. Diesen erweitern Sie dann ebenfalls um eigene, neue Funktionen. Die wichtigsten Infos dazu finden Sie ab Seite 39 im Artikel **twinBASIC: COM-Add-Ins für den VBA-Editor**.

Und wo wir schon beim Erweitern sind, stellen wir Ihnen gleich auch noch einen Schaltflächen-Assistenten vor, mit dem Sie leicht benutzerdefinierte Schaltflächen zu einem neuen Formular hinzufügen können. Die Techniken lernen Sie in den beiden Beiträgen **Schaltflächen-Assistent** ab S. 49 und **Schaltflächen per Code anlegen** ab S. 57 kennen.

Und schließlich beschäftigen wir uns im Beitrag **Abhängige Listenfelder** ab Seite 2 noch mit abhängigen Listenfeldern.

Viel Spaß beim Ausprobieren!



Ihr André Minhorst

Abhängige Listenfelder

Die Programmierung abhängiger Kombinationsfelder haben wir bereits mindestens einmal behandelt. Hier soll ein Kombinationsfeld nur Werte abhängig von der Auswahl eines anderen Kombinationsfeldes anzeigen, also zum Beispiel die Artikel zu einer vorher gewählten Kategorie. Mit Listenfeldern geht das auch – dazu sind nur wenige Umstellungen nötig. Zumindest, wenn Sie im ersten Listenfeld nur die Auswahl eines Eintrags gleichzeitig erlauben. In diesem Beitrag wollen wir uns nicht nur das anschauen, sondern auch die Anzeige abhängiger Daten bei einem Listenfeld mit Mehrfachauswahl.

Abhängigkeit bei Kombinationsfeldern

Bei abhängigen Kombinationsfeldern wie denen aus Bild 1 gelingt das Abbilden der Abhängigkeit wie folgt:

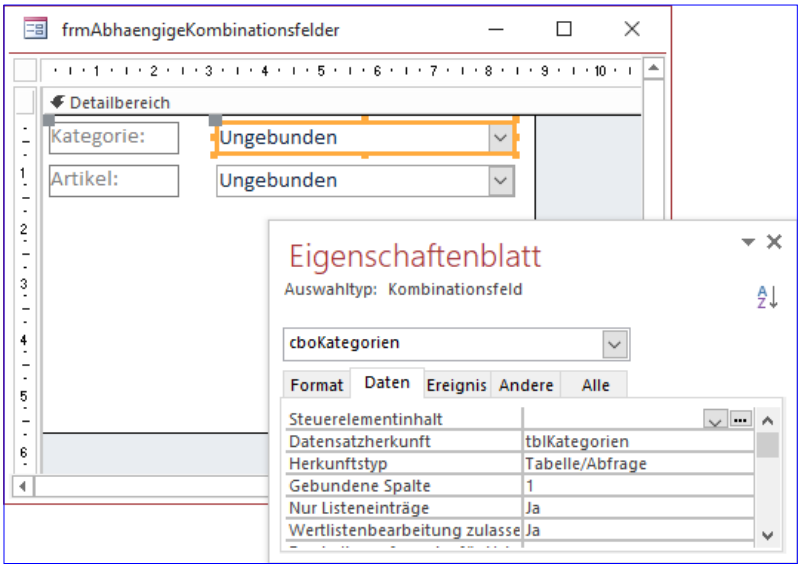


Bild 1: Abhängige Kombinationsfelder in der Entwurfsansicht

Das Kombinationsfeld **cboKategorien** zeigt die Werte der Felder **KategorieID** und **Kategorie** der Tabelle **tblKategorien** an. Wenn der Benutzer einen der Einträge auswählt, soll das zweite Kombinationsfeld **cboArtikel** nur noch die Artikel anzeigen, die zu der im ersten Kombinationsfeld gewählten Kategorie passen. Die Aktualisierung führen wir in der Ereignisprozedur aus, die durch das Wählen eines neuen Eintrags im Listenfeld **cboKategorien** ausgelöst wird, nämlich **Nach Aktualisierung**. Die Ereignisprozedur sieht so aus:

```
Private Sub cboKategorien_AfterUpdate()
    Dim strSQL As String
    strSQL = "SELECT ArtikelID, Artikelname FROM tblArtikel
             WHERE KategorieID = " & Nz(Me!cboKategorien)
    Me!cboArtikel.RowSource = strSQL
End Sub
```

Dadurch weisen wir der Eigenschaft **Datensatzherkunft** einen SQL-Ausdruck zu, der nur noch die Datensätze der Tabelle **tblArtikel** liefert, deren **KategorieID** im ersten Kombinationsfeld ausgewählt wurde. Das Ergebnis ist eine

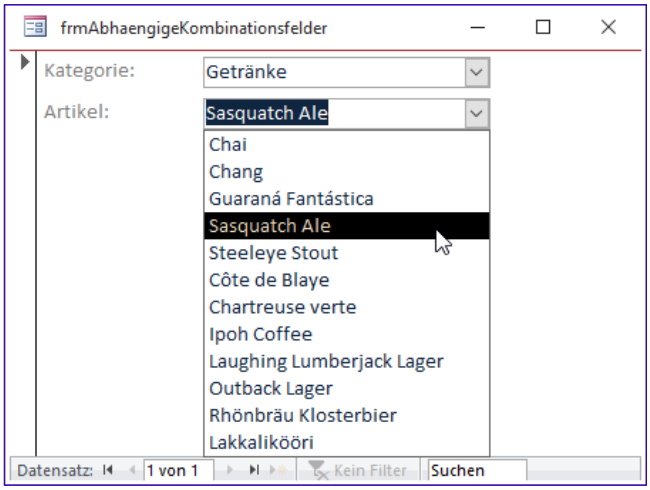


Bild 2: Auswahl abhängiger Datensätze

erheblich reduzierte Anzahl von Datensätzen im zweiten Kombinationsfeld (siehe Bild 2).

Umsetzung für abhängige Listenfelder

Wie unterscheidet sich dies, wenn wir statt Kombinationsfelder einfach Listenfelder verwenden? Dazu kopieren wir dieses Formular **frmAbhaengigeKombinationfselder** in ein neues Formular namens **frmAbhaengigeListenfelder** und wandeln die Kombinationsfelder mit dem Kontextmenü-Eintrag **Ändern zulListenfeld** jeweils in Listenfelder um (siehe Bild 3).

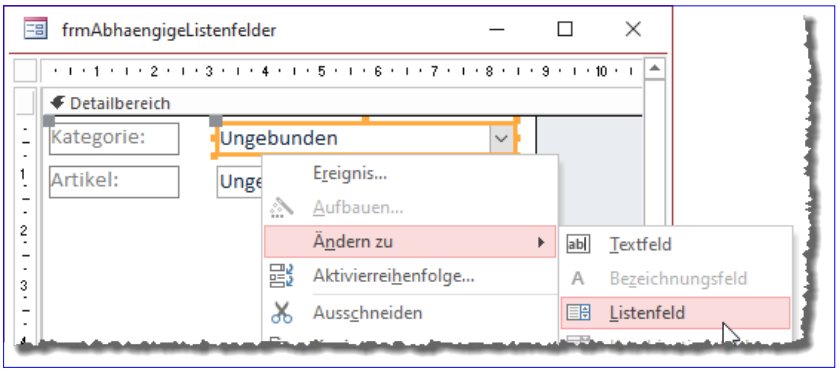


Bild 3: Umwandeln von Kombinationsfeld in Listenfeld

Danach ändern wir noch die Position und Größe der Listenfelder, damit diese ausreichend Datensätze anzeigen können und nebeneinander angezeigt werden (siehe Bild 4).

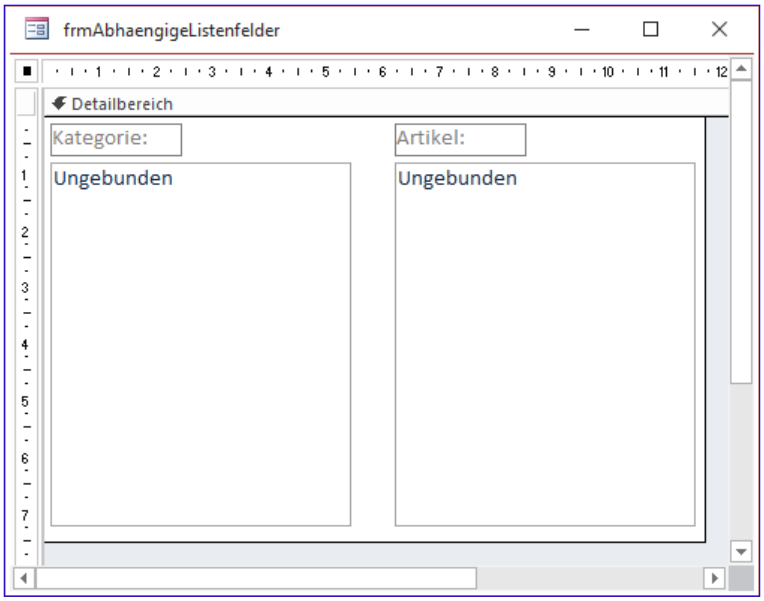


Bild 4: Ausrichten der Listenfelder

Außerdem benennen wir die Listenfelder noch um von **cboKategorien** und **cboArtikel** in **lstKategorien** und **lstArtikel**. Damit brauchen wir nun nur noch das Ereignis **Nach Aktualisierung** neu zu implementieren. Hier ändern sich die Bezeichnungen von Prozedurname und den betroffenen Steuerelementen:

```
Private Sub lstKategorien_AfterUpdate()
    Dim strSQL As String
    strSQL = "SELECT ArtikelID, Artikelname FROM tblArtikel
                WHERE KategorieID = " & Nz(Me!lstKategorien)
    Me!lstArtikel.RowSource = strSQL
End Sub
```

Das Ergebnis sehen Sie dann in Bild 5. Das Filtern gelingt genauso wie mit den Kombinationsfeldern.

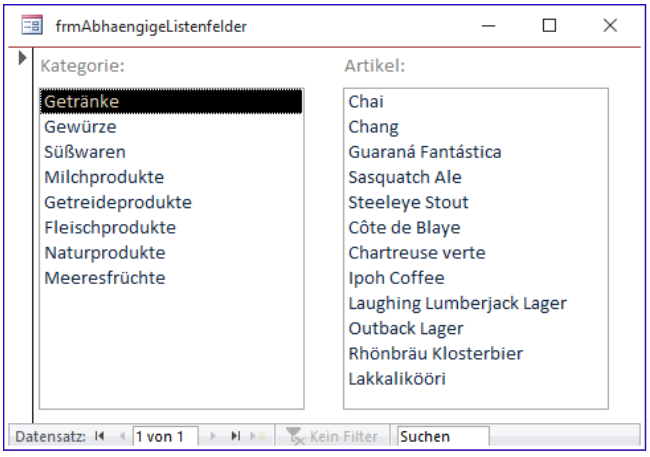


Bild 5: Filtern per Listenfeld

Damit kommen wir zu den anspruchsvolleren Aufgaben – dem Filtern nach mehreren ausgewählten Kategorien.

twinBASIC – VB/VBA mit moderner Umgebung

Wayne Philips ist Access-Entwicklern ein Begriff. Es ist der Erfinder von vbWatchDog, dem Tool für eine professionelle Fehlerbehandlung und er schafft es, .accde-Datenbanken wieder in .accdb-Datenbanken zurückzuverwandeln. Nun kommt sein nächster großer Coup: twinBASIC. Dabei handelt es sich um eine moderne Version der klassischen Programmiersprachen VB6 und VBA. Modern deshalb, weil Sie damit in einer modernen Entwicklungsumgebung arbeiten können statt im angestaubten VBA-Editor, nämlich in VS Code. Und weil es neue Elemente zu diesen Sprachen hinzufügt. Das Ergebnis sind eigenständige Anwendungen, DLLs und mehr. Wir schauen uns in diesem Beitrag an, wie Sie twinBASIC einsatzbereit machen. In weiteren Beiträgen geht es dann in die Details – hier lernen Sie dann, wie Sie beispielsweise eine COM-DLL bauen, die Sie unter Access referenzieren können.

VS Code für twinBASIC

Bevor wir starten: Was ist dieses VS Code eigentlich? Es ist ein recht neuer Quelltext-Editor von Microsoft, der plattformübergreifend verfügbar ist, also für Windows, macOS und Linux. Es bietet einige Funktionen, die im Visual Basic Editor nur teilweise oder gar nicht vorhanden sind, zum Beispiel Syntaxhervorhebung, Code-Faltung, Debugging, Autovervollständigung oder Versionsverwaltung.

Es ist nicht zu verwechseln mit Visual Studio. Visual Studio arbeitet mit Projektdateien, VS Code mit Quelltextdateien und Ordnern. Diese werden in einem Workspace verwaltet und der aktuelle Zustand des Workspaces kann gespei-

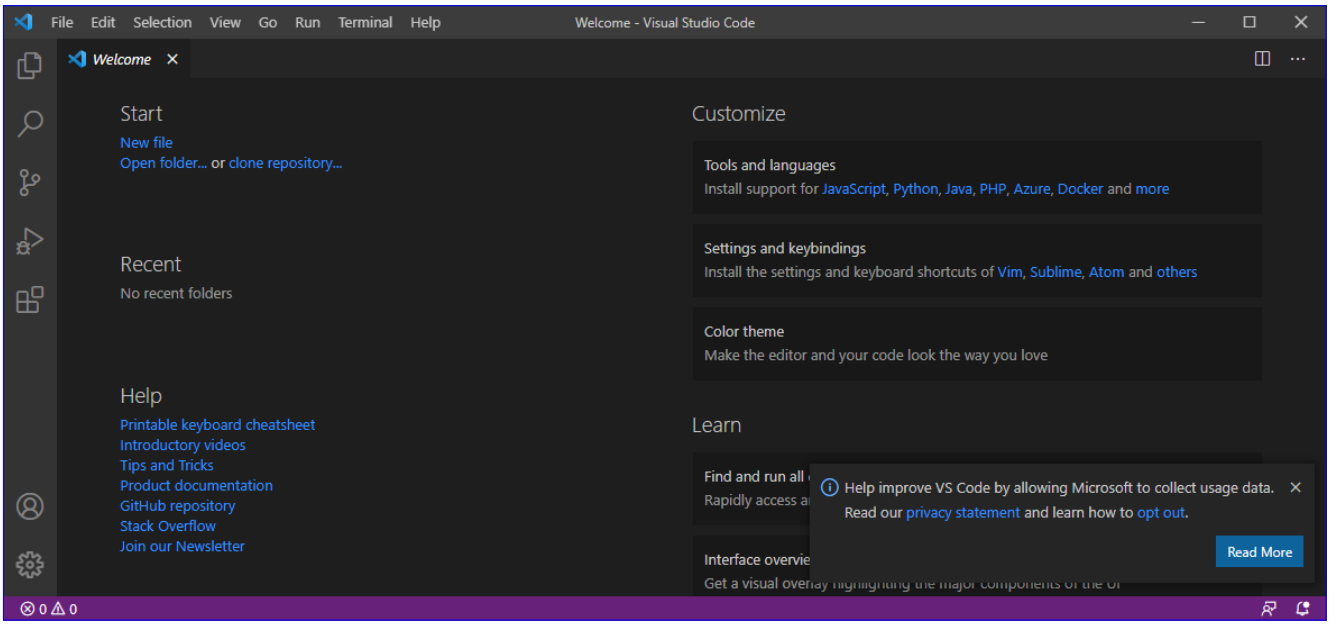


Bild 1: VS Code direkt nach der Installation

chert und wiederhergestellt werden.

Es bietet ein Plug-In-System, das sich auch twinBASIC zunutze macht. Wayne Philips hat ein Plug-In programmiert, mit dem Sie in VS Code in der Sprache twinBASIC programmieren und auch die ausführbaren Dateien kompilieren können.

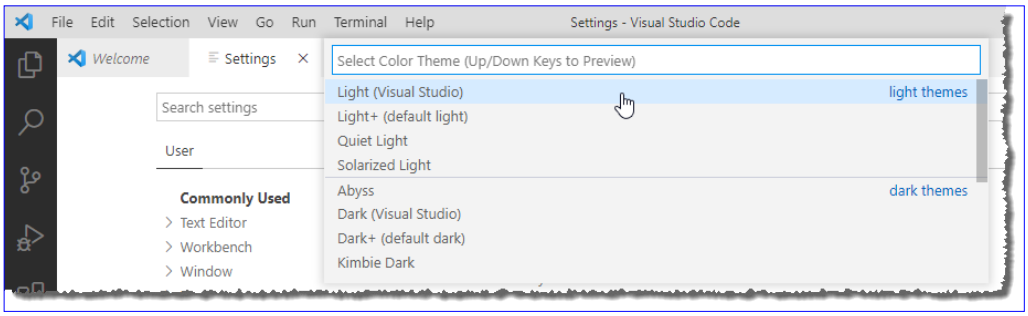


Bild 2: Einstellen des Farbschemas

VS Code ist Open Source und wird von einem Entwicklerteam rund um Erich Gamma entwickelt. Erich Gamma hat bereits das Buch **Design Patterns - Elements of Reusable Object-Oriented Software** geschrieben und die Entwicklung der Entwicklungsumgebung **Eclipse** geleitet.

VS Code herunterladen und installieren

Den Download von VS Code finden Sie unter folgendem Link:

```
https://code.visualstudio.com/download
```

Hier wählen Sie die Windows-Version für Ihr System aus und laden die entsprechende Datei herunter. Anschließend starten Sie die heruntergeladene **.exe**-Datei, um VS Code zu installieren. Im Setup können Sie noch angeben, ob VS Code im Kontextmenü für Dateien untergebracht werden soll oder ob bestimmte Dateitypen direkt damit verknüpft werden sollen.

Danach können wir VS Code direkt starten. Es erscheint dann wie in Bild 1. Auch wenn der schwarze Hintergrund toll aussieht und die meisten sicher der englischen Sprache mächtig sind, nehmen wir hier gleich einmal zwei

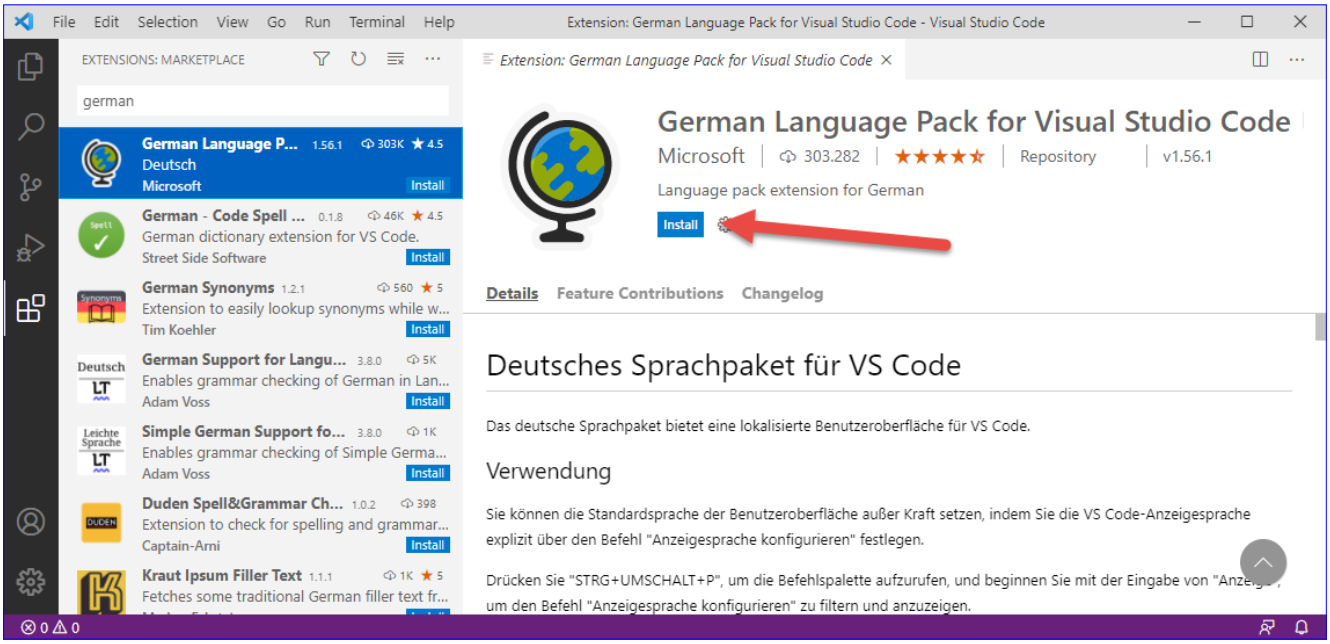


Bild 3: Installieren des deutschen Sprachpakets

Änderungen bezüglich dieser Eigenschaften vor.

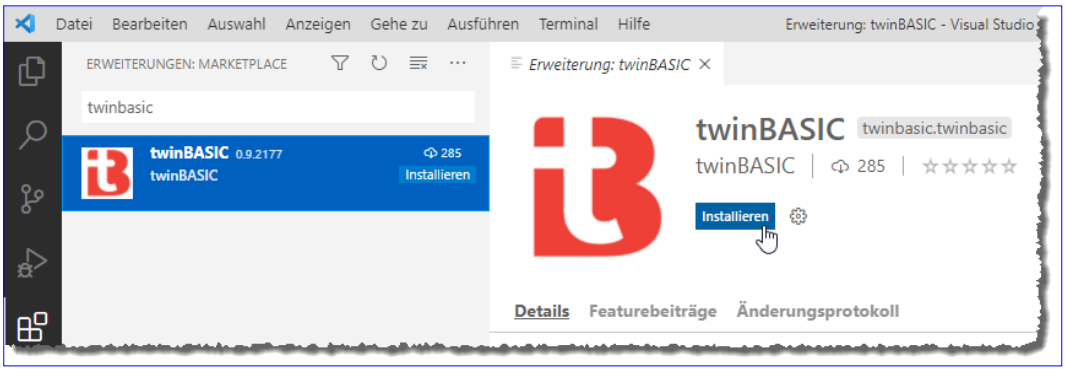


Bild 4: Installieren der twinBASIC-Erweiterung

Farbschema einstellen

Als Erstes stellen wir die Farbe ein. Dazu nutzen wir dem Menübefehl **File|Preferences|Color Theme**. Hier wählen wir das Schema **Light (Visual Studio)** aus (siehe Bild 2).

Sprache einstellen

Wir hätten gern deutsche Befehle und dazu ist die Installation eines Language Packs nötig. Dazu wählen Sie den Menübefehl **File|Preferences|Extensions** aus. Im nun erscheinenden Bereich **Extensions: Marketplace** geben Sie einfach **German** ein und erhalten direkt das **German Language Pack for Visual Studio Code**.

Dazu klicken Sie auf die in Bild 3 sichtbare Schaltfläche **Install**. Die Installation geschieht in etwa einer Sekunde, der anschließende Neustart liefert dann direkt die deutschen Bezeichnungen für die Befehle der Benutzeroberfläche.

twinBASIC-Plugin installieren

Da wir nun schon Übung im Installieren von Extensions haben, fügen wir gleich noch die Erweiterung für twinBASIC hinzu. Also öffnen wir wieder den Bereich von eben, diesmal aber über den deutschsprachigen Menübefehl **Datei|Einstellungen|Erweiterungen**. Im nun erscheinenden Bereich **Erweiterungen** geben wir als Suchbegriff **twinBASIC** ein.

Anschließend klicken Sie auch hier auf die Schaltfläche **Installieren** (siehe Bild 4).

twinBASIC nutzen: Hello World!

Natürlich starten wir mit einem **Hello World!**-Beispiel in die Nutzung von twinBASIC. Wir haben nun VS Code installiert und die twinBASIC-Erweiterung installiert – wie geht es nun weiter? Wenn wir über den Befehl **Datei|Neue Datei** eine neue Datei anlegen, geschieht jedenfalls nichts besonderes – es erscheint eine leere Textdatei, die Sie füllen können. Kein IntelliSense oder ähnliches in Sicht.

Hello World!-Projekt öffnen

twinBASIC verfügt zu diesem Zeitpunkt noch nicht über die Möglichkeit, ein komplett neues Projekt zu erstellen, daher starten wir mit einem vorgefertigten Projekt. Dieses enthält zwei Dateien:

- helloword.code-workspace
- helloworld.twinproj

Nach dem Herunterladen der Dateien wählen Sie in VS Code den Befehl **Datei|Arbeitsbereich öffnen** aus und öffnen über den **Arbeitsbereich öffnen**-Dialog die Workspace-Datei.

Wenn Sie den Explorer anzeigen und den Arbeitsbereich **HelloWorld** aufklappen, finden Sie wie in Bild 5 die Datei **HelloWorld.twin**, die Sie per Doppelklick öffnen und so den Code des Moduls anzeigen.

Klicken Sie nun in den Code der Prozedur **Public Sub Main**, können Sie diese mit der Taste **F5** ausführen. Dies zeigt das erwartete Meldungsfenster an.

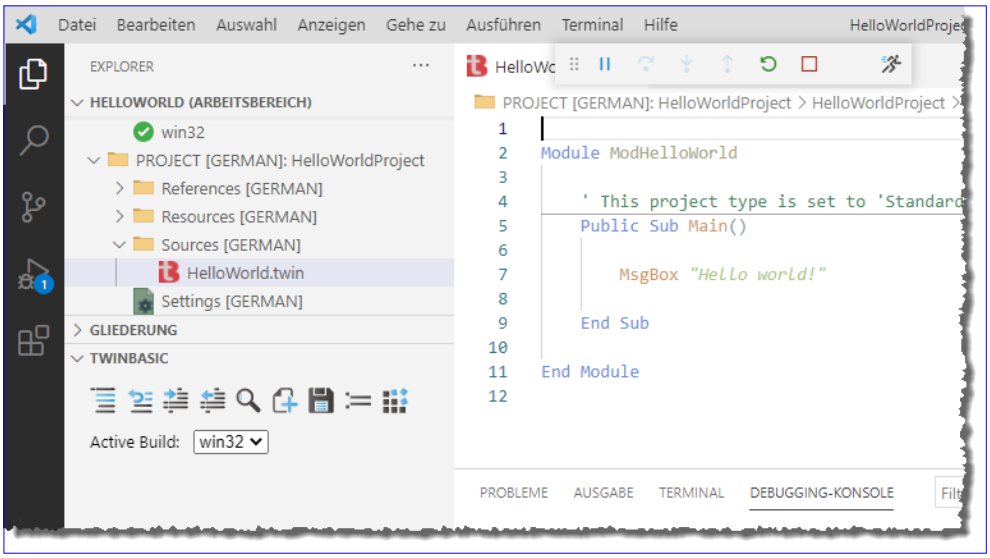


Bild 5: Das Hello World!-Projekt in VS Code

Da es sich hier um die **Main**-Prozedur handelt, die auch beim Starten einer auf Basis dieses Projekts erstellten **.exe**-Datei aufgerufen wird, können Sie auch einfach auf die **Build**-Schaltfläche klicken.

.exe erstellen

Jetzt kommt der Clou: Mit einem Klick auf die **Build**-Schaltfläche erstellen Sie direkt die **.exe**-Datei für das Projekt (siehe Bild 6).

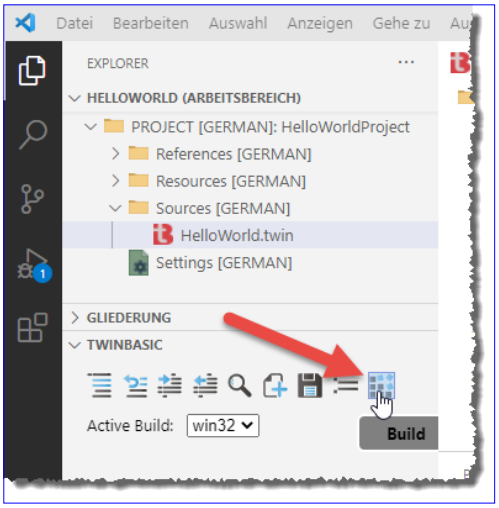


Bild 6: Erstellen des Projekts

Im Ordner **Build** finden wir dann die Datei **HelloWorld-Project_win32.exe** vor, die wir per Doppelklick ausführen können (siehe Bild 7). Dies liefert die gewünschte Meldung. Richtig spannend wird es bei der Dateigröße: Diese beträgt nämlich gerade mal 8 KB! Und wir benötigen neben dieser Datei keine weiteren Dateien.

Dateien des Projekts

Weiter oben haben wir bereits die beiden Dateien des ersten Projekts erwähnt. Die Datei mit der Dateiendung **.code-workspace** ist etwa mit der Solution aus Visual Studio-Projekten vergleichbar.

Die Datei mit der Endung **.twinproj** ist die Projektdatei, die ein virtuelles Dateisystem mit den verschiedenen Elementen des Projekts enthält. Diese beiden Dateien

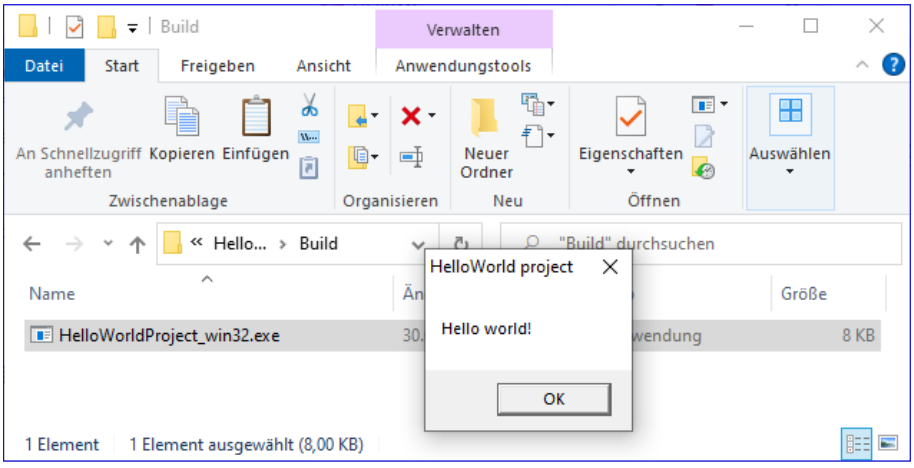


Bild 7: Die .exe-Datei und die dadurch angezeigte Meldung

COM-DLLs mit twinBASIC

Bisher haben wir COM-DLLs entweder mit Visual Studio 6 erstellt oder mit Visual Studio .NET. Mit twinBASIC kommt nun eine weitere Möglichkeit hinzu. twinBASIC ist noch in der Entwicklung, aber Sie können damit durchaus bereits lauffähige COM-DLLs bauen. Dieser Beitrag zeigt, wie Sie das Beispiel-COM-DLL des Entwicklers von twinBASIC, Wayne Philips, verwenden und wie Sie eigene Funktionen hinzufügen.

Vorbereitungen zum Erstellen von COM-DLLs mit twinBASIC

Den Großteil der Vorbereitungen für den Einsatz von twinBASIC haben wir im Beitrag **twinBASIC – VB/VBA mit moderner Umgebung** (**www.access-im-unternehmen.de/1303**) beschrieben. Hier erfahren Sie, wie Sie die Entwicklungsumgebung VS Code installieren und die benötigte Erweiterung twinBASIC hinzufügen. Außerdem haben wir dort ein kleines Beispielprojekt demonstriert.

Aktuelle Versionen

Neben der Version, die wir für die Beschreibungen in diesem Beitrag genutzt haben, wird twinBASIC aktuell sehr schnell weiterentwickelt.

Da es für uns Access-Entwickler sehr ungewohnt ist, dass eine Entwicklungsumgebung weiterentwickelt wird (und das auch noch schnell), weisen wir hier explizit darauf hin, dass es aktuelle Versionen und Informationen an folgenden Orten gibt:

- Die aktuellen Downloads finden Sie unter **https://twinbasic.com**.
- Bugs et cetera können Sie hier melden oder auch einsehen: **https://github.com/WaynePhillipsEA/twinbasic/issues**

Auf **https://twinbasic.com** finden Sie auch einige Beispiele, unter anderem das hier als Grundlage verwendete Beispiel für eine ActiveX-DLL.

Beispiel herunterladen

Die zum Zeitpunkt der Erstellung dieses Beitrags verwendete Version des Beispiels finden Sie wie gewohnt im Download zu diesem Beitrag, aktuellere Versionen gegebenenfalls unter dem oben genannten Link.

Wenn Sie die Zip-Datei heruntergeladen haben, finden Sie darin zwei Dateien – eine mit der Dateiendung **.code-workspace** und eine mit der Dateiendung **.twinproj**. Um das Projekt zu öffnen, klicken Sie doppelt auf die Datei mit der Dateiendung **.code-workspace** (gegebenenfalls müssen Sie die Dateiendung noch mit dieser Anwendung verknüpfen).

Dies öffnet den Arbeitsbereich, der sowohl das Projekt aus der **.twinproj**-Datei anzeigt als auch alle notwendigen Einstellungen enthält. Das Ergebnis sieht nach einem Doppelklick auf den Eintrag **HelloWorld.twin** im Ordner **Sources** wie in Bild 1 aus.

Wir sehen hier eine Klasse namens **MyTestLibrary** mit einer einzigen öffentlich deklarierten Funktion namens **MultipleByTen**.

Projekt erstellen

Wir schauen uns zunächst an, wie wir die COM-DLL zum Laufen bringen. Dazu brauchen Sie lediglich auf die Schaltfläche **Build** im Bereich **TWINBASIC** zu klicken.

Dies erzeugt die **.dll**-Datei im Ordner **Build** des Projektordners und legt gleichzeitig die benötigten Einträge in der Registry an – dazu später mehr.

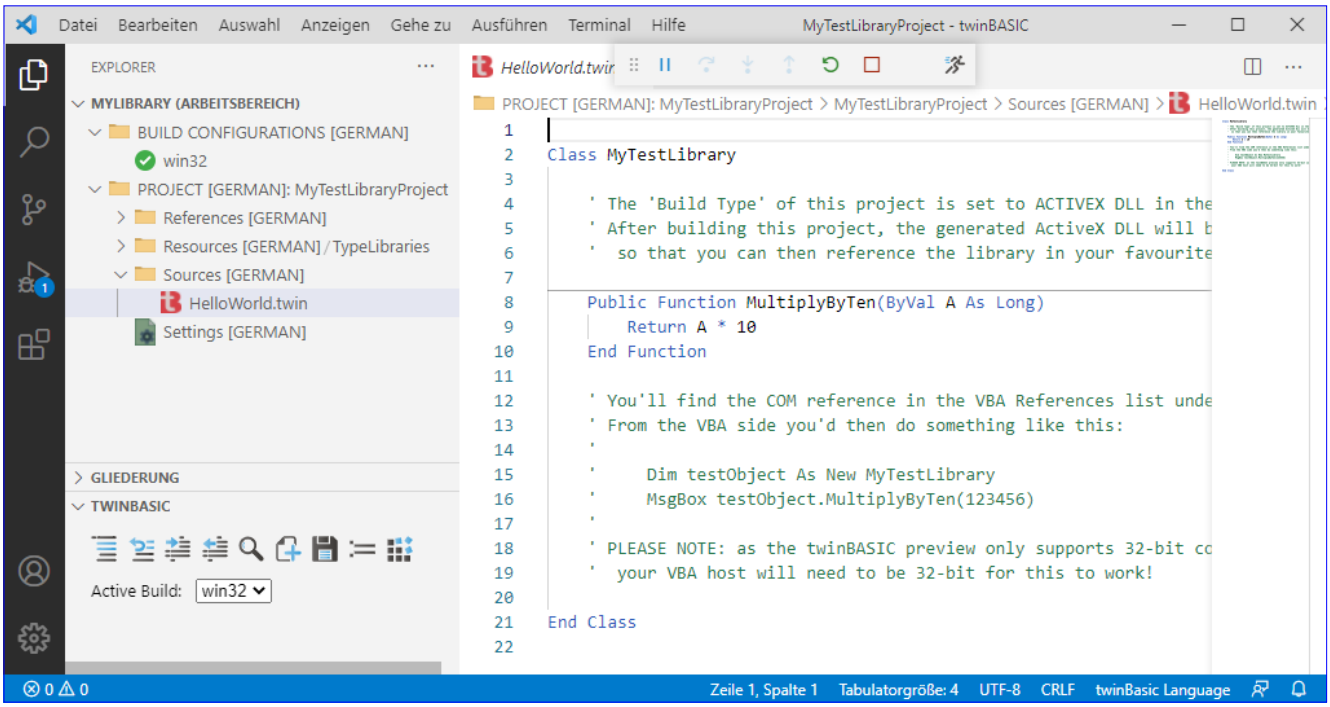


Bild 1: COM-Add-In-Projekt in VS Code

COM-DLL einbinden

Wir öffnen eine Access-Datenbank und starten den VBA-Editor. Hier rufen wir mit dem Menü-Eintrag **Extras|Verweise** den **Verweise**-Dialog auf. Dieser bietet unter dem Namen **MyTestLibrary project** einen neuen Eintrag zum Einbinden der DLL an (siehe Bild 2). Diesen aktivieren wir durch Platzieren eines Hakens im entsprechenden Kontrollkästchen.

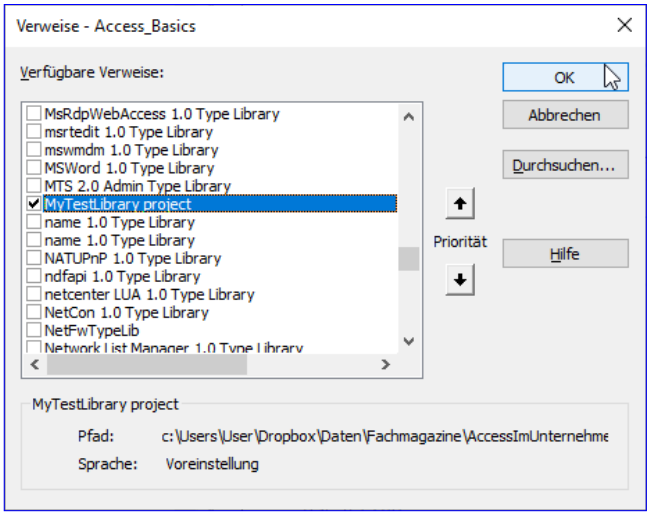


Bild 2: Verweis auf die DLL

Befehle der COM-DLL ausprobieren

Danach können Sie im VBA-Editor folgende Prozedur anlegen:

```
Public Sub TestCOMDLL()
    Dim obj As MyTestLibrary
    Set obj = New MyTestLibrary
    Debug.Print obj.MultiplyByTen(10)
End Sub
```

Rufen Sie die Prozedur auf, liefert diese mit dem Wert **100** das korrekte Ergebnis. Auch IntelliSense funktioniert sehr gut mit der Funktion (siehe Bild 3).

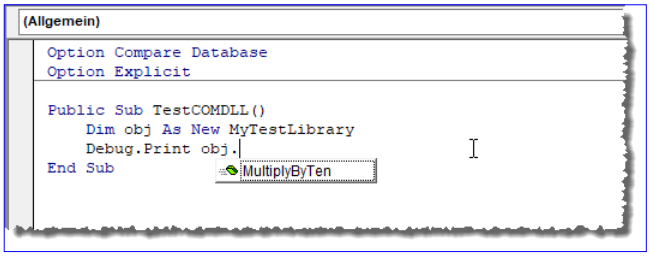


Bild 3: IntelliSense mit der COM-DLL

Late Binding

Das Erstellen funktioniert auch mit Late Binding. Allerdings müssen Sie die Bibliothek zusätzlich zur Klasse angeben. Das sieht dann beispielsweise wie folgt aus:

```
Public Sub TestCOMDLL1()
    Dim obj As Object
    Set obj = CreateObject(_
                    "MyTestLibraryProject.MyTestLibrary")
    Debug.Print obj.MultiplyByTen(10)
End Sub
```

Geben Sie die Bibliothek nicht an, führt dies zu dem Fehler aus Bild 4.

Registrierung der COM-DLL

Die Registrierung der COM-DLL erfolgt beim Build-Vorgang. Diese Information erhalten Sie, wenn Sie sich das Debug-Fenster von VS Code ansehen (siehe Bild 5). Diesen Bereich aktivieren Sie mit dem Menüpunkt **Anzeigen|Debugging-Konsole**.

Sie können dazu nicht die übliche Anweisung **Regexp32.exe <Pfad zur DLL>** verwenden, mit der normalerweise DLLs registriert werden.

Somit steht auch der Befehl **Regexp32.exe <Pfad zur DLL> -u** nicht zur Verfügung, mit dem Sie auf diese Weise registrierte DLLs normalerweise wieder deregistrieren können.

Einstellungen für das Projekt

Wenn Sie doppelt auf den Eintrag **Settings** des Explorers klicken, erscheint der Bereich mit den Einstellungen. Dieser zeigt im oberen Teil den Namen des Projekts sowie die Verweise an (siehe Bild 6).

Hier ist die Beschreibung einiger der Einstellungen:

- **Project: Name**: Name des Projekts. Dieser wird beispielsweise im Dateinamen der zu erstellenden DLL-

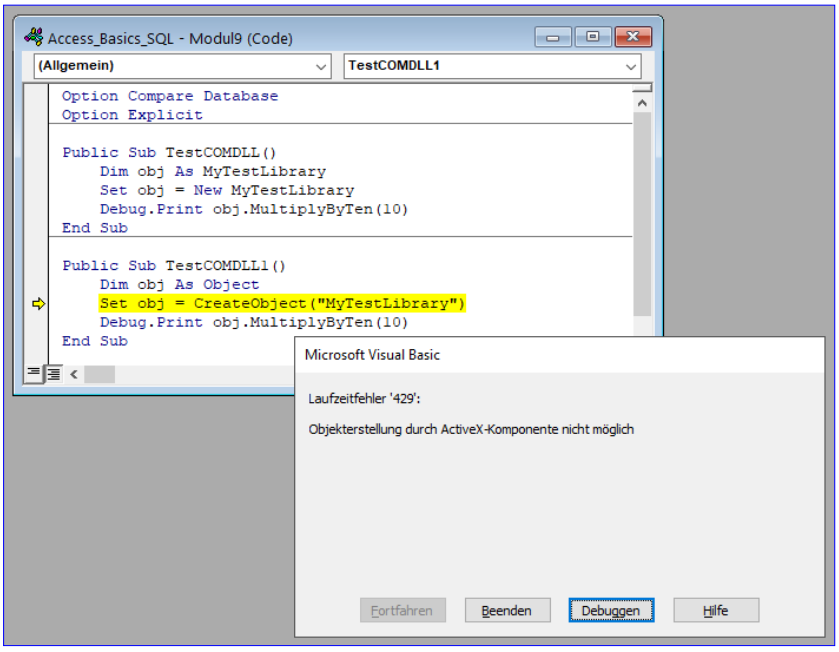


Bild 4: Fehler beim Versuch, die Klasse per Late Binding zu nutzen

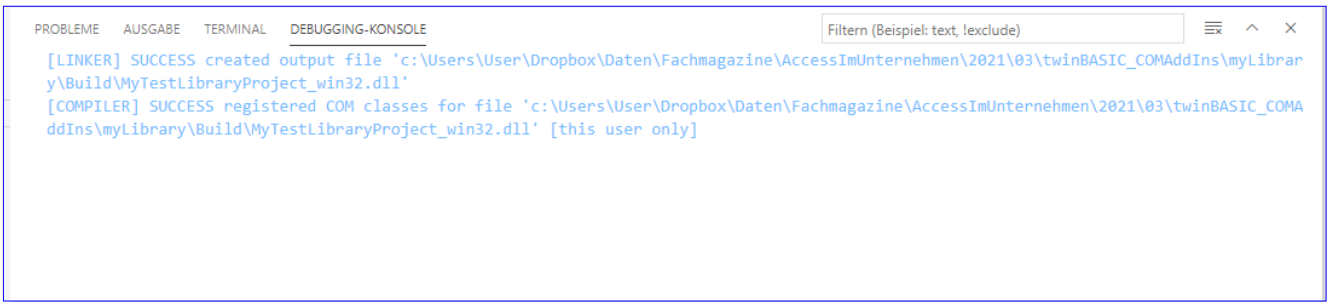


Bild 5: Ausgabe beim Build-Vorgang

Debugging in twinBASIC

Der Entwickler von twinBASIC, Wayne Philips, ist vor allem für sein Fehlerbehandlungstool vbWatchDog bekannt. Kein Wunder, dass er bei der Programmierung der Entwicklungsumgebung darauf geachtet hat, dass ausreichende Debugging-Möglichkeiten vorhanden sind. Wie Sie diese nutzen, zeigt der vorliegende Beitrag. Dabei lernen Sie die verschiedenen Bereiche wie Variablen, Überwachen, Aufrufliste, Haltepunkte und den Debug-Bereich kennen.

Debugging in twinBASIC

Wenn Sie das Beispielprojekt aus dem Download geöffnet und die einzige enthaltene Datei **HelloWorld.twin** angezeigt haben, können Sie mit einem Klick auf die Schaltfläche **Ausführen und Debuggen** am linken Rand die Debugging-Bereiche der twinBASIC-Erweiterung von Visual Studio Code einblenden.

Sie können auch die Tastenkombination **Strg + Umschalt + D** nutzen, um den linken Bereich sichtbar zu machen. Hier finden Sie gleich vier Bereiche vor:

- **Variablen**: Zeigt die Inhalte der sichtbaren Variablen an.
- **Überwachen**: Hier können Sie Ausdrücke zur Überwachung hinzufügen.
- **Aufrufliste**: Hier werden die aktuell aufgerufenen Routinen abgebildet sowie die Routinen, von denen diese aufgerufen wurden.
- **Haltepunkte**: Zeigt die aktuellen Haltepunkte an. Per Klick springen Sie zur jeweiligen Zeile im Code.

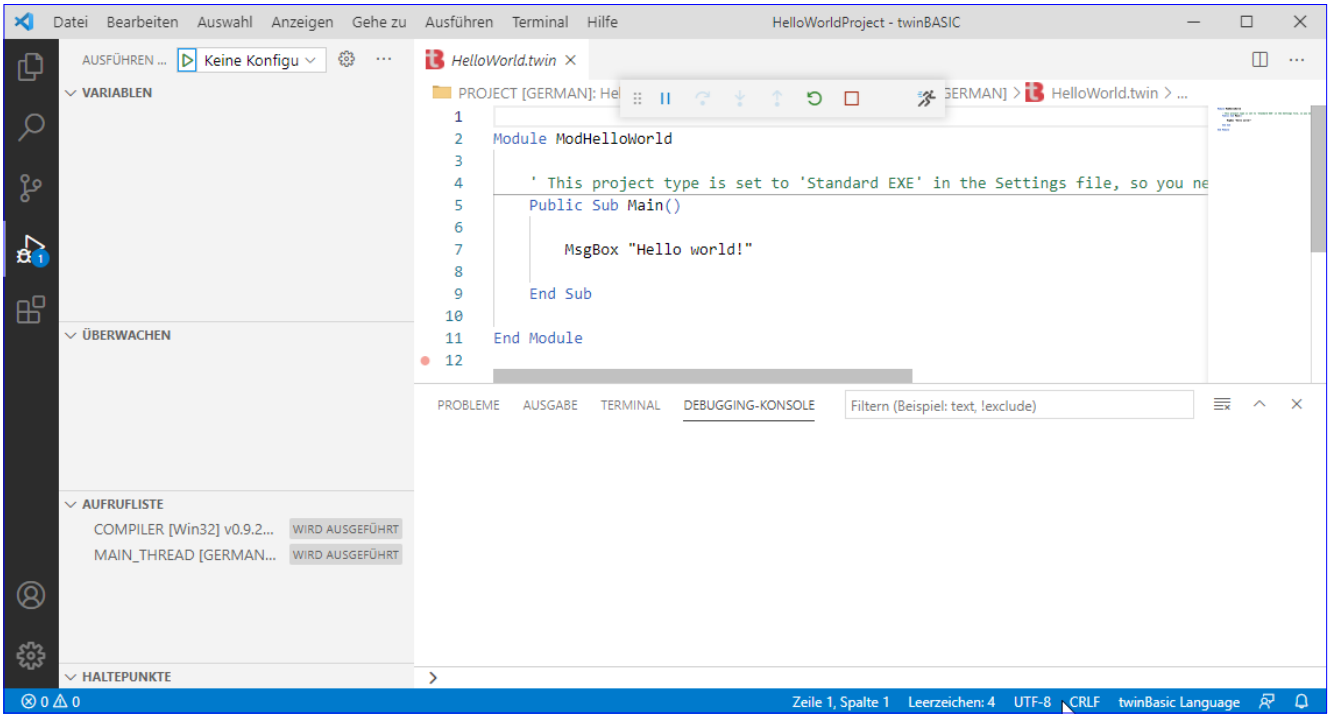


Bild 1: Die verschiedenen Debugging-Bereiche

Außerdem zeigt das Fenster im unteren Bereich noch die Debugging-Konsole an (siehe Bild 1).

Routinen starten und durchlaufen

Derzeit ist twinBASIC noch eine reine Entwicklungsumgebung für EXEs ohne Benutzeroberfläche, DLLs, COM-Add-Ins et cetera. Da findet Debugging natürlich verstärkt in der Entwicklungsumgebung statt. Standardprojekte auf Basis der **HelloWorld**-Vorlage enthalten eine **Main**-Routine, die automatisch beim Start der Anwendung aufgerufen wird.

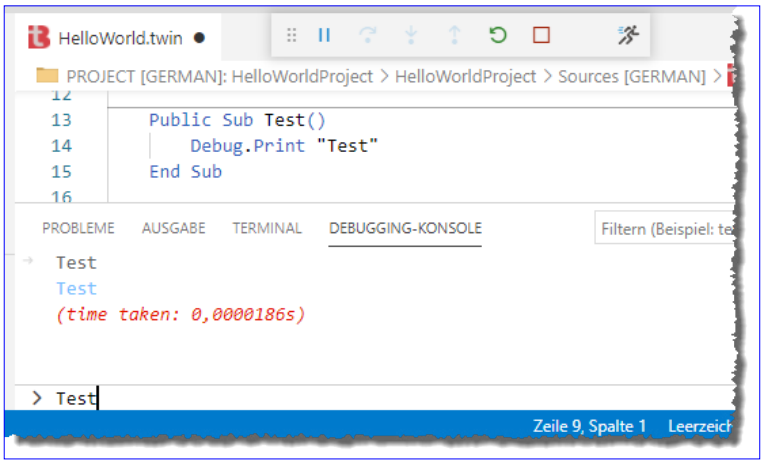


Bild 2: Starten einer Routine über den Debuggen-Bereich

Um diese Routine direkt in der Entwicklungsumgebung zu starten, haben Sie mehrere Möglichkeiten:

- über den Debugging-Bereich
- durch Markieren der Routine und Betätigen der Taste **F5** oder der Schaltfläche zum Starten der aktuell markierten Prozedur
- per Menüeintrag

Routine vom Debugging-Bereich aus aufrufen

Die erste Möglichkeit ist der Aufruf der Routine aus dem Debugging-Bereich heraus. Hier geben Sie den Namen der Routine und gegebenenfalls der Parmaeter an und betätigen die Eingabetaste.

Der Debugging-Bereich sieht etwas anders aus als im VBA-Editor. Sie können hier nicht einfach Befehle in den Bereich eintippen, sondern verwenden dazu den kleinen Bereich mit dem Größer-Zeichen (**>**). In dieser starten wir die Prozedur **Test** durch Eingabe des Routinennamens **Test** und anschließendes Betätigen der Eingabetaste.

Im Debugging-Bereich erfolgt dann die Ausgabe: erstens der Name der Routine, die gerade aufgerufen wurde, dann der Output der Routine mit der **Debug.Print**-Anweisung und schließlich die für den Aufruf benötigte Zeit (siehe Bild 2).

Sie können die **Debug.Print**-Anweisung auch direkt von der **>**-Zeile aus aufrufen. Dann wird die **Debug.Print**-Anweisung ausgeführt und samt Aufruf und Laufzeit im Debugging-Bereich ausgegeben.

Routine mit F5 starten

Wenn Sie keine Lust haben, den Routinennamen erst in den Debugging-Bereich einzutippen, können Sie auch einfach die Einfügemarke in der auszuführenden Routine platzieren und dann die Taste **F5** betätigen – oder die Schaltfläche zum Starten des Debuggings (siehe Bild 3).

Diesen Befehl können Sie auch über den Menüpunkt **Ausführen|Debugging starten** aufrufen (siehe Bild 4). Hier finden wir noch weitere Befehle, die allerdings zum Zeitpunkt der ersten Preview noch nicht funktionierten:

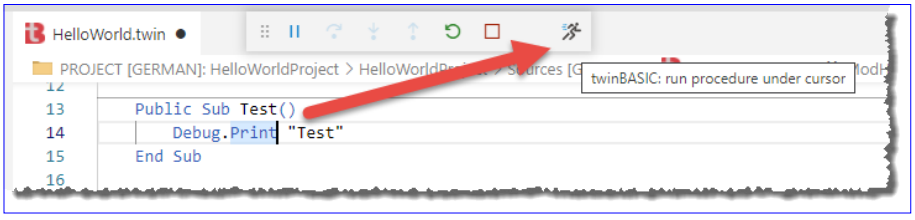


Bild 3: Starten des Debuggens per Schaltfläche

- **Debugging starten**: Startet die aktuell markierte Routine (Taste **F5**).
- **Ohne Debuggen ausführen** (Tastenkombination **Strg + F5**)

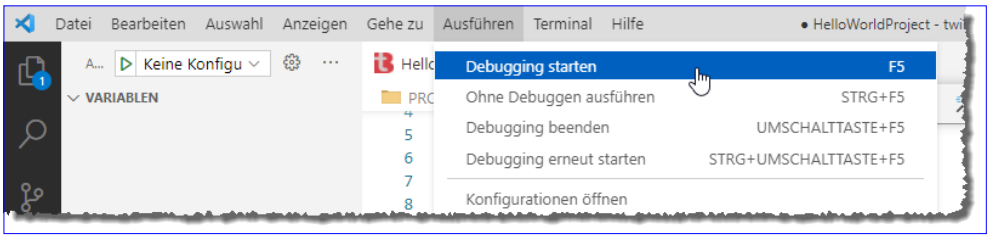


Bild 4: Debugging-Befehle per Menü

- **Debugging beenden**: Beendet das Debuggen (Tastenkombination **Umschalt + F5**)
- **Debugging erneut starten**: Startet das Debuggen erneut (Tastenkombination **Strg + Umschalt + F5**)

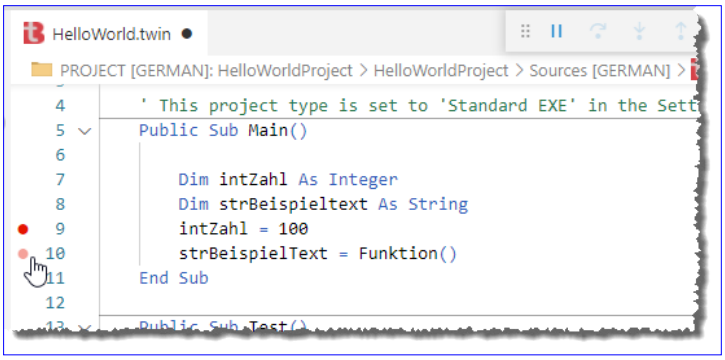


Bild 5: Setzen eines Haltepunktes

Haltepunkte setzen

Genau wie im VBA-Editor können Sie auch in VS Code unter twinBASIC Haltepunkte setzen. Dazu klicken Sie einfach links von der Zeilennummer der Codezeile, in der Sie anhalten möchten. Dort erscheint dann wie in Bild 5 ein roter Punkt.

Wenn Sie die Routine etwa mit der Taste **F5** gestartet haben, bleibt die Abarbeitung in der Zeile mit dem Haltepunkt stehen (siehe Bild 6).

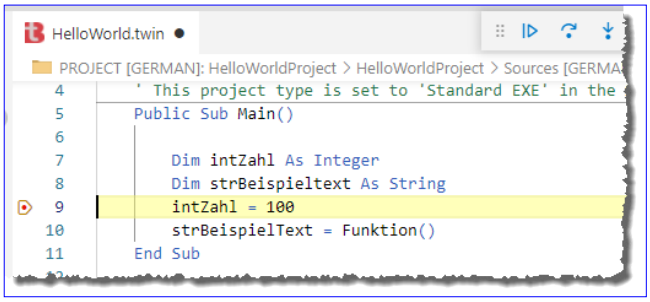


Bild 6: Erreichen eines Haltepunktes

Code fortsetzen

Um die Abarbeitung des Codes bis zum Ende oder bis zum nächsten Haltepunkt fortzusetzen, betätigen Sie einfach nochmal die Taste **F5**.

Code im Einzelschritt-Modus durchlaufen

Wenn Sie jedoch nach Erreichen des Haltepunktes den Code Schritt für Schritt durchlaufen wollen, um diesen zu untersuchen, betätigen Sie die Taste **F11** – also nicht mit **F8** wie im VBA-Editor.

Die Debug.Print-Methode

Das Debugging-Fenster können Sie für die Ausgabe per **Debug.Print** aus dem Code heraus genauso nutzen wie das des VBA-Editors. Sprich: Sie fügen irgendwo im Code eine **Debug.Print**-Anweisung ein, geben den auszugebenden Inhalt als Parameter an und lassen den Code laufen.

Die **Debug.Print**-Methode arbeitet allerdings nicht genau so wie die im VBA-Editor: Es ist (noch) nicht möglich, den auszugebenden Text mit dem Komma- oder Semikolon abzuschließen.

Das führt im VBA-Editor dazu, dass kein Zeilenumbruch an die Ausgabe angehängt wird und die Ausgabe der nächsten **Debug.Print**-Anweisung in der gleichen Zeile erfolgt – beim Komma durch einen Tabulator vom vorherigen Text

twinBASIC: COM-Add-Ins für Access

Neben COM-DLLs können Sie mit twinBASIC auch COM-Add-Ins programmieren, deren Funktionen dann in der Benutzeroberfläche von Access angezeigt und genutzt werden können. Die Möglichkeiten sind unbegrenzt – Sie können damit beispielsweise Ribbon-Einträge hinzufügen, die dauerhaft und unabhängig von der jeweils geöffneten Datenbank verfügbar sind und damit selbst programmierte Funktionen aufrufen. Welche Funktionen sinnvoll sind und sich hier umsetzen lassen, schauen wir uns in weiteren Beiträgen an. Dieser Beitrag beleuchtet zunächst einmal die technischen Grundlagen für die Erstellung von COM-Add-Ins für die Access-Benutzeroberfläche.

COM-Add-Ins und Access-Add-Ins

Bevor wir in die Materie einsteigen, wollen wir den Unterschied zwischen COM-Add-Ins und Access-Add-Ins klären. Access-Add-Ins sind Access-Datenbanken, die eine spezielle Dateiendung enthalten und eine Tabelle namens **USysRegInfo** mit Einträgen, welche Informationen über das Add-In zum Eintragen in die Registry enthalten.

Diese rufen Sie schließlich über den Ribbon-Eintrag **Datenbanktools|Add-Ins|Add-Ins** auf. Bild 1 zeigt einige Add-Ins in der Access-Installation des Autors dieses Beitrags. Ein Klick auf einen dieser Einträge startet diesen und zeigt beispielsweise ein Formular mit den enthaltenen Funktionen an. Es gibt noch weitere Möglichkeiten, Access-Add-Ins zu platzieren – zum Beispiel als Steuerelement-Assistent oder Formular-Assistent.

Der wichtigste Unterschied zwischen Access-Add-Ins und den hier vorgestellten COM-Add-Ins für Access ist, dass Sie für COM-Add-Ins keine speziellen Orte vorfinden, an denen Sie deren Aufruf platzieren können. Stattdessen fügen Sie einfach im gewünschten Ribbon einen Eintrag hinzu oder auch ein eigenes Ribbon, über das Sie das COM-Add-In starten können. Der zweite wichtige Unterschied ist, dass Sie Access-Add-Ins komplett mit Access erstellen. Sie brauchen dann nur die Dateiendung von **.accdb** auf **.accda** zu ändern und die bereits erwähnte Tabelle namens **USysRegInfo** mit Informationen über

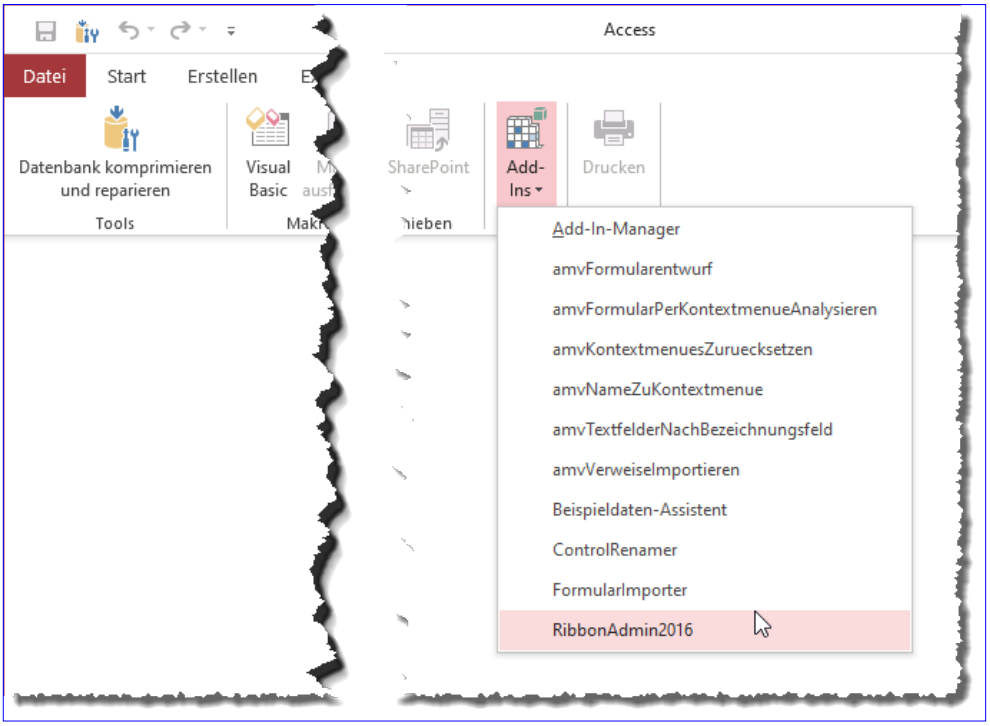


Bild 1: Access-Add-Ins

die Art des Access-Add-Ins, den Namen et cetera sowie die auszuführende Funktion hinzuzufügen.

Ein COM-Add-In hingegen können Sie nicht einfach mit Access erstellen, sondern Sie benötigen dazu eine externe Entwicklungsumgebung. Früher hat man das mit Visual Studio 6 erledigt. Auch der Nachfolger Visual Studio .NET hat im Laufe der Zeit verschiedene Tools zum Erstellen von Office-Add-Ins angeboten.

Für die Erstellung von Access-Add-Ins musste man jedoch unter Visual Studio .NET immer improvisieren: Vorlagen gab es nämlich nur für die populäreren Office-Anwendungen wie Word, Outlook oder Excel.

Kein 64-Bit unter Visual Studio 6

Der Nachteil bei der COM-Add-In-Entwicklung unter Visual Studio 6 zeigt sich aktuell zunehmend darin, dass immer mehr Installationen von Office beziehungsweise Access die 64-Bit-Version verwenden. Und Visual Studio 6 erstellt nur Projekte in der 32-Bit-Version. Insofern sind mit **twinBASIC** erstellte COM-Add-Ins auch zukunftssicherer.

COM-Add-Ins mit twinBASIC

Im April 2021 hat jedoch Wayne Philips, der bereits mit dem Fehler- und Debugging-Tool vbWatchdog von sich hören machte, eine erste Version seiner Entwicklungsumgebung für VB- und VBA-kompatible Projekte veröffentlicht. Diese basiert auf Visual Studio Code und nutzt eine eigene Erweiterung für das Programmieren von VB- und VBA-Projekten.

In Kürze soll man damit auch Anwendungen mit Benutzeroberfläche damit programmieren können, derzeit ist das aber noch nicht möglich. Für uns spielt das keine Rolle – wir schauen uns erst einmal die grundlegenden Möglichkeiten der Erstellung von COM-Add-Ins für die Access-Benutzeroberfläche an.

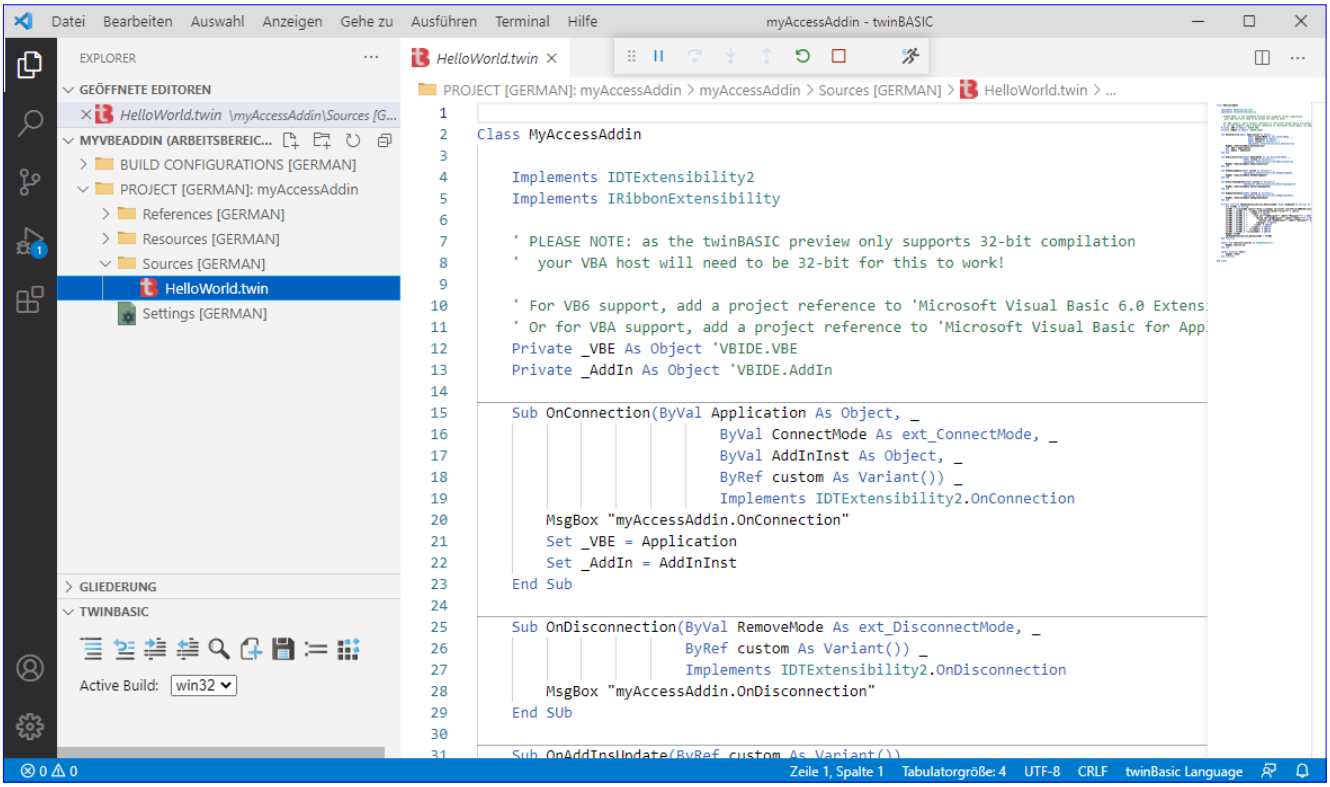


Bild 2: Aufbau eines Projekts für die Erstellung eines COM-Add-Ins

Die Grundlagen zum Einrichten von Visual Studio Code und der für die VB-Programmierung benötigten Erweiterung haben wir bereits im Beitrag **twinBASIC – VB/VBA mit moderner Umgebung** (**www.access-im-unternehmen.de/1303**) beschrieben. Im vorliegenden Beitrag bauen wir auf einem einfachen Template für ein Access-COM-Add-In auf und erläutern die einzelnen Bestandteile.

Anschließend fügen wir dem COM-Add-In einige Beispielfunktionen hinzu.

Wir starten mit einem Beispielprojekt, das Sie per Doppelklick auf die Datei **myAccessAddIn.code-workspace** starten. Dies öffnet das Projekt im twinBASIC-Editor beziehungsweise in Visual Studio Code wie in Bild 2.

```
Class MyAccessAddin
    Implements IDTExtensibility2
    Implements IRibbonExtensibility

    Private _Access As Object 'VBIDE.VBE
    Private _AddIn As Object 'VBIDE.AddIn

    Sub OnConnection(ByVal Application As Object, ByVal ConnectMode As ext_ConnectMode, _
            ByVal AddInInst As Object, ByRef custom As Variant()) _
            Implements IDTExtensibility2.OnConnection
        MsgBox "myAccessAddin.OnConnection"
        Set _Access = Application
        Set _AddIn = AddInInst
    End Sub

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

    Sub OnAddInsUpdate(ByRef custom As Variant()) _
            Implements IDTExtensibility2.OnAddInsUpdate
        MsgBox "myAccessAddin.OnAddInsUpdate"
    End Sub

    Sub OnStartupComplete(ByRef custom As Variant()) _
            Implements IDTExtensibility2.OnStartupComplete
        MsgBox "myAccessAddin.OnStartupComplete"
    End Sub

    Sub OnBeginShutdown(ByRef custom As Variant())_
            Implements IDTExtensibility2.OnBeginShutdown
        MsgBox "myAccessAddin.OnBeginShutdown"
    End Sub

End Class
```

Listing 1: Implementierung der Schnittstelle für COM-Add-Ins

Die hier sichtbaren Prozeduren sind die Implementierung einer Schnittstelle namens **IDExtensibility2**, die bereits seit VB6 zum Implementieren der Funktionen für COM-Add-Ins für die verschiedenen Office-Anwendungen und auch für den VBA-Editor verwendet wird (siehe Listing 1).

Um diese in der Klasse unseres twinBASIC-Projekts zu implementieren, fügen wir dem Modul **HelloWorld.twin** die folgende Zeile hinzu:

```
Implements IDTExtensibility2
```

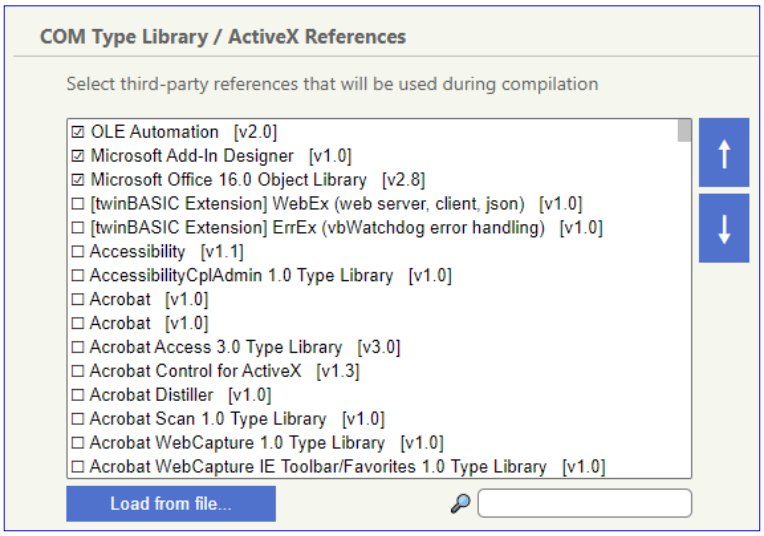


Bild 3: Notwendige Verweise für ein Access-Add-In

Diese Schnittstelle ist in der Bibliothek **Microsoft Add-In Designer** definiert, die genau wie die Implementierung der Schnittstelle bereits im Beispielprojekt enthalten ist. Die Verweise auf Bibliotheken finden Sie übrigens im Bereich **Settings** unter **COM Type Library / ActiveX References** (siehe Bild 3).

Hier haben wir auch gleich noch die Bibliothek **Microsoft Office 16.0 Object Library** hinzugefügt, denn wir wollen später auch noch ein Ribbon zum Aufrufen der Befehle des COM-Add-Ins zum Projekt hinzufügen. Diesen Verweis finden Sie schnell in der Liste, wenn Sie im Suchen-Feld Teile des Verweisnamens eingeben, beispielsweise **Office**.

Diese Schnittstelle enthält einige Ereignisprozeduren, die zu verschiedenen Gelegenheiten ausgelöst werden. Das Besondere an einer Schnittstelle ist, dass Sie alle dafür definierten Ereignisse implementieren müssen – auch wenn diese gar keine Befehle enthalten und somit nichts tun.

Die wichtigste Ereignisprozedur für diese Implementierung für uns ist in der Regel die Methode **OnConnection**. Diese wird aufgerufen, wenn Access startet und nachdem es im entsprechenden Registry-Zweig nachgesehen hat, ob irgendwelche COM-Add-Ins geladen werden müssen – dazu später mehr.

Diese Ereignisprozedur stellt zwei wichtige Parameter zur Verfügung:

- **Application**: Liefert einen Verweis auf das **Application**-Objekt der Anwendung, die das COM-Add-In aufruft.
- **AddInInst**: Liefert einen Verweis auf das **AddIn**-Objekt.

Mit dem Verweis auf das **Application**-Objekt der Anwendung können Sie dann vom Code des COM-Add-Ins aus auf die Anwendung selbst zugreifen. Um diesen Verweis während der Verwendung des COM-Add-Ins ständig im Zugriff zu haben, speichern wir es in einer Variablen, die wir wir folgt deklarieren:

```
Private _Access As Access.Application
```

Sie ahnen es bereits: Wenn wir Elemente der Access-Bibliothek verwenden, benötigen wir auch noch einen Verweis auf die Bibliothek **Microsoft Access 16.0 Object Library**. Diesen fügen Sie noch über den oben bereits erwähnten Bereich **COM Type Library / ActiveX References** hinzu.

Wichtig: Damit die enthaltenen Elemente auch im Code zur Verfügung stehen, müssen Sie die geänderten Einstellungen einmal speichern.

twinBASIC: COM-Add-Ins für den VBA-Editor

Neben COM-Add-Ins für Access selbst (und natürlich auch für die übrigen Office-Anwendungen) können Sie mit twinBASIC auch COM-Add-Ins für den VBA-Editor programmieren. Dieser Beitrag beleuchtet detailliert, welche Schritte dazu notwendig sind. Dabei beginnen wir mit einer COM-Add-In-Vorlage, die wir für unseren Anwendungszweck anpassen. Anders als bei COM-Add-Ins für Access können Sie die Funktionen von COM-Add-Ins für den VBA-Editor nicht per Ribbon-Eintrag aufrufen, sondern benötigen einen Menüleisteneintrag dafür. Wir erläutern, wie dieser erzeugt wurd und wie Sie dann auf die Elemente des VBA-Editors zugreifen können.

twinBASIC

Im Beitrag **twinBASIC – VB/VBA mit moderner Umgebung** (**www.access-im-unternehmen.de/1303**) zeigen wir, was twinBASIC ist und wie Sie es installieren. twinBASIC ist eine Programmierumgebung auf Basis von Visual Studio Code mit einer Erweiterung von Wayne Philips, den manche vielleicht schon als Entwickler des Fehlerbehandlungstools **vbWatchDog** kennen. Sie können damit aktuell EXE-Dateien ohne Benutzeroberfläche erstellen oder DLLs, die Sie in Access-VBA-Projekte einbinden – siehe **COM-DLLs mit twinBASIC** (**www.access-im-unternehmen.de/1304**). Außerdem können Sie damit COM-Add-Ins erstellen. Im Beitrag **twinBASIC – COM-Add-Ins für Access** (**www.access-im-unternehmen.de/1306**) erfahren Sie, wie Sie COM-Add-Ins für Access-Anwendungen programmieren.

Wozu COM-Add-Ins für den VBA-Editor?

Wer immer nur mit Access programmiert und keine anderen Entwicklungsumgebungen als den VBA-Editor nutzt, hat sich vielleicht schon so sehr an die recht betagte Entwicklungsumgebung gewöhnt, dass er nichts vermisst. Wer jedoch gelegentlich über den Tellerrand schaut und sieht, welche Möglichkeiten andere Entwicklungsumgebungen bieten, würde sich vielleicht doch über die eine oder andere Weiterentwicklung freuen. Nun ist die Benutzeroberfläche von Access an sich seit der Version 2010 nicht mehr weiterentwickelt worden – bis auf einige unbedeutende Änderungen. Der Funktionsumfang wurde hingegen eher eingeschränkt – man denke an die Replikation und das Sicherheitssystem. Ein ähnliches Bild liefert der VBA-Editor. Genau genommen liegen hier die letzten Änderungen noch länger zurück.

Gelegentlich hat ein Entwickler Ideen, wie er den VBA-Editor verbessern könnte, was theoretisch auch mit .NET-COM-Add-Ins möglich wäre. Es gibt einige Vorlagen, mit denen man Word-, Excel- oder Outlook-Erweiterungen programmieren kann. Für Access oder auch für den VBA-Editor gibt es jedoch keine Projektvorlagen, also müsste man schon komplett selbst herausfinden, an welchen Stellen in den Vorlagen für die anderen Office-Anwendungen man schrauben müsste.

Da ist es doch erfrischend, dass Wayne Philips eine Möglichkeit bietet, COM-Add-Ins in einer neuen, alternativen Entwicklungsumgebung zum VBA-Editor und zu Visual Studio (.NET) zu entwickeln – von Visual Studio 6 reden wir erst gar nicht mehr, da damit nur die Entwicklung von Programmen auf Basis von 32-Bit möglich ist.

Mit twinBASIC können Sie jedoch, wie auf den folgenden Seiten beschrieben, auch COM-Add-Ins für den VBA-Editor entwickeln und es ist sogar geplant, dass dies auch für die 64-Bit-Version von Office möglich sein soll (aktuell ist twinBASIC noch in einem frühen Entwicklungsstadium, Features wie die Programmierung von Userinterfaces beispielsweise folgen erst später).

Projektvorlage umwandeln

Auf der Webseite von Wayne Philips finden Sie unter dem Link **https://twinbasic.com/preview.html** einige Beispiele, darunter auch das Beispiel eines COM-Add-Ins. Dieses verwenden wir als Basis für unser erstes eigenes COM-Add-In für den VBA-Editor.

Der Ordner in der Zip-Datei enthält vier Dateien:

- **myVBEAddin.code-workspace**: Datei mit den Workspace-Einstellungen
- **myVBEAddin.twinproj**: Datei mit den eigentlichen Projektdaten
- **RegisterAddin32.reg**: Datei mit Informationen zum Registrieren des COM-Add-Ins
- **UnregisterAddin32.reg**: Datei mit Informationen zum Entfernen der Registrierung des COM-Add-Ins

Die beiden ersten Dateien müssen immer den gleichen Namen aufweisen. Sie öffnen ein Projekt, indem Sie die Datei mit der Dateiendung **code-workspace** öffnen (gegebenenfalls müssen Sie die Dateiendung noch mit Visual Studio Code verknüpfen).

Dies liefert im nun erscheinenden Visual Studio Code die Ansicht aus Bild 1. Um weitere Dateien müssen Sie sich noch keine Gedanken machen, aktuell reicht eine Projektdatei mit dem Code völlig aus.

Einstellungen anpassen

Außerdem finden Sie noch Einstellungen, die wir uns nun ansehen und gegebenenfalls erweitern. Klicken Sie auf den Bereich **Settings**, finden Sie einige interessante Eigenschaften vor, die wir an dieser Stelle kurz vorstellen wollen (siehe Bild 2):

- **Project: Name**: Name des Projekts. Erscheint jedoch im Gegensatz zu COM-DLLs nirgends.

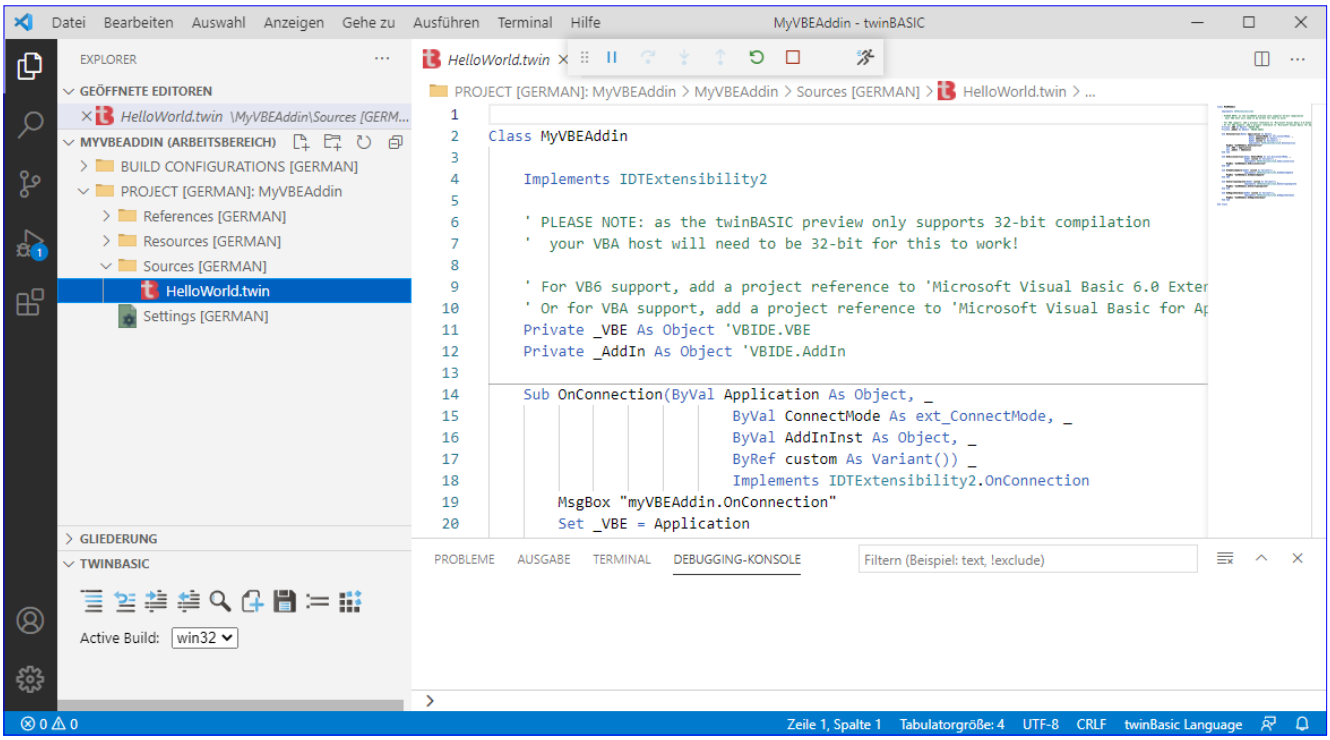


Bild 1: Entwicklungsumgebung mit dem Code des COM-Add-Ins für den VBA-Editor

- **COM Type Library / ActiveX References**: Liste der Verweise des Projekts, die Sie erweitern können.
- **Project: GUID**: GUID des Projekts. Diese sollten Sie anpassen, wenn Sie ein neues Projekt auf Basis dieser Vorlage erstellen.
- **Project: Build Output Path**: Pfad mit Platzhaltern, der angibt, wo die zu erstellende Datei gespeichert werden soll.
- **Project: Description**: Beschreibung des Projekts. Erscheint jedoch im Gegensatz zu COM-DLLs nirgends.

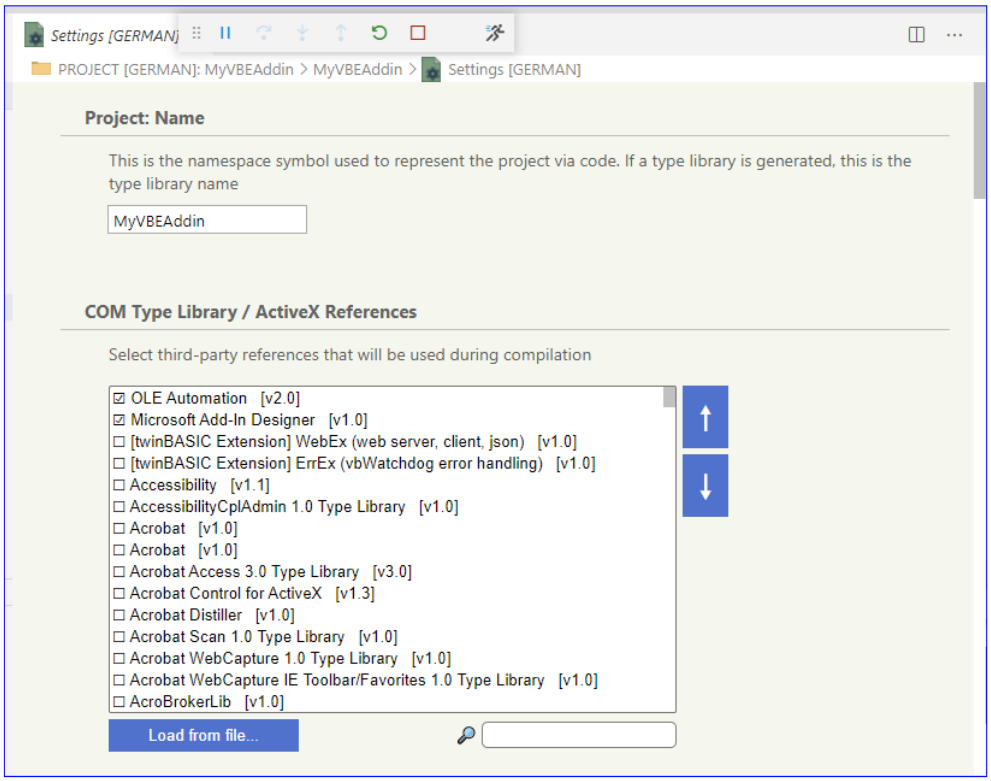


Bild 2: Einstellungen für das Projekt

Sie haben nun schon an zwei Stellen gelesen, dass Eigenschaften nirgends erscheinen. Das ist ein Unterschied zu COM-DLLs, bei denen die hier angegebenen Informationen zum Beispiel im Verweise-Dialog erscheinen. Wo und wie Sie den Namen für das COM-Add-In für den VBA-Editor einstellen und wo es dann erscheint, lesen Sie weiter unten.

Code des COM-Add-Ins für den VBA-Editor

Wenn Sie das Modul **HelloWorld.twin** ansehen, finden Sie eine Klasse vor, welche die Schnittstelle **IDTExtensibility2** implementiert. Das bedeutet, dass die **Class**-Anweisung den Zusatz **Implements IDEXTENSIBILITY2** enthält und dass die Klasse alle Elemente dieser Schnittstelle implementieren muss, was bedeutet, dass die in der Klasse enthaltenen Methoden vorliegen müssen (siehe Listing 1).

In den Methoden finden Sie zunächst nur jeweils eine **MsgBox**-Anweisung, die nur dazu dient, die grundlegende Funktion des COM-Add-Ins für den VBA-Editor zu demonstrieren, solange Sie noch keine echte Funktionalität eingebaut haben.

Außerdem enthält die Klasse die Deklaration zweier Variablen:

```
Private _VBE As Object 'VBIDE.VBE
Private _AddIn As Object 'VBIDE.AddIn
```

Diese sind noch mit dem Datentyp **Object** ausgestattet, damit das Add-In flexibel mit verschiedenen Hosts eingesetzt werden kann. Wir ersetzen dies gleich, benötigen aber zuvor zwei weitere Verweise. Diese fügen wir im Bereich Settings unter **COM Type Library / ActiveX References** zwei neue Verweise hinzu, nämlich **Microsoft Visual Basic For Applications Extensibility 5.3** und **Microsoft Office 16.0 Object Library** (siehe Bild 3). An-

```
Class MyVBEAddin
    Implements IDTExtensibility2
    Private _VBE As Object 'VBIDE.VBE
    Private _AddIn As Object 'VBIDE.AddIn

    Sub OnConnection(ByVal Application As Object, ByVal ConnectMode As ext_ConnectMode, ByVal AddInInst As Object, _
            ByRef custom As Variant()) Implements IDTExtensibility2.OnConnection
        MsgBox "myVBEAddin.OnConnection"
        Set _VBE = Application
        Set _AddIn = AddInInst
    End Sub

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

    Sub OnAddInsUpdate(ByRef custom As Variant()) Implements IDTExtensibility2.OnAddInsUpdate
        MsgBox "myVBEAddin.OnAddInsUpdate"
    End SUb

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

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

Listing 1: Aufbau der Klasse des COM-Add-Ins für den VBA-Editor

schließend müssen Sie die Änderungen an den Einstellungen speichern, was Sie beispielsweise mit der Tastenkombination **Strg + S** erledigen können.

Danach können Sie die Deklaration der beiden Variablen wie folgt ändern:

```
Private _VBE As VBIDE.VBE
Private _AddIn As VBIDE.AddIn
```

Diese werden in der Prozedur **OnConnection** gefüllt:

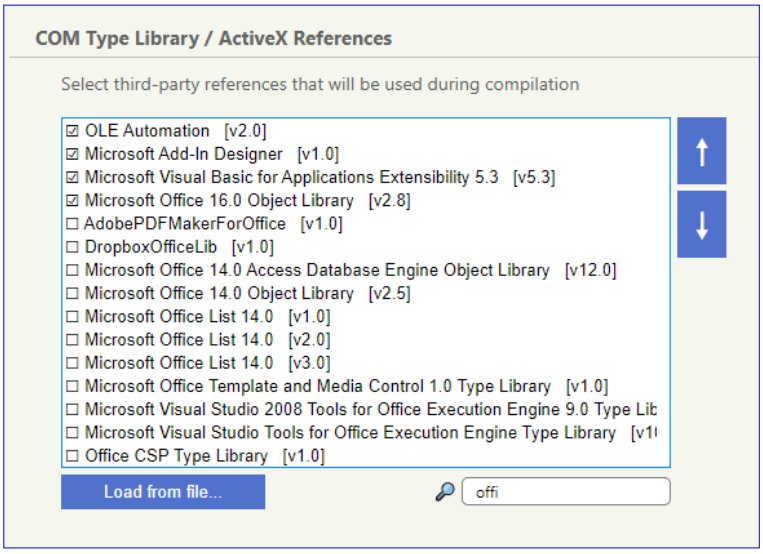


Bild 3: Hinzufügen einiger Verweise auf Projektbibliotheken

Schaltflächen-Assistent

Access bietet für Einsteiger bereits einen Befehlsschaltflächen-Assistent an, der grundlegende Operationen unterstützt. Die Funktionen sind allerdings stark begrenzt und das Ergebnis ist nicht an die aktuellen Möglichkeiten von Access angepasst. Wir zeigen in diesem Beitrag, wie Sie selbst einen praktischen Assistenten zum schnellen Erstellen von Schaltflächen programmieren können. Vorher schauen wir uns an, was der eingebaute Assistent kann und was wir verbessern und ergänzen wollen.

Steuerelement-Assistenten unter Access

Neben den Assistenten, die Sie über den Ribbon-Eintrag **Datenbanktools\|Add-Ins\|Add-Ins** starten können, gibt es auch einige Assistenten, die Sie beim Erstellen von Steuerelementen unterstützen sollen. Diese bietet Access standardmäßig beim Erstellen von Steuerelementen an.

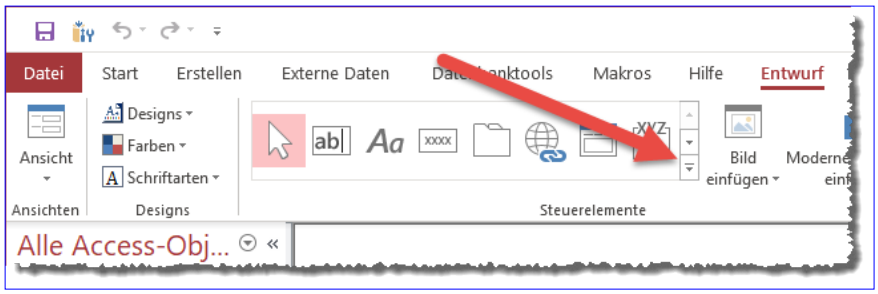


Bild 1: Ausklappen weiterer Steuerelemente und Optionen

Falls dies bei Ihnen nicht der Fall ist, wurde gegebenenfalls die entsprechende Option deaktiviert. Um die Steuerelement-Assistenten wieder zu aktivieren, öffnen Sie zunächst ein Formular in der Entwurfsansicht. Dann klicken Sie im Ribbon auf den Tab **Entwurf** und dann im Bereich **Steuerelemente** auf das Symbol zum Aufklappen weiterer Steuerelemente (siehe Bild 1).

Wenn die weiteren Optionen ausgeklappt sind, finden Sie unter der Liste aller eingebauten Steuerelemente den Eintrag **Steuerelement-Assistenten** verwenden (siehe Bild 2). Ist das Wizard-Symbol farbig hinterlegt, ist die Option aktiviert, sonst nicht. In diesem Fall aktivieren Sie die Option per Mausklick.

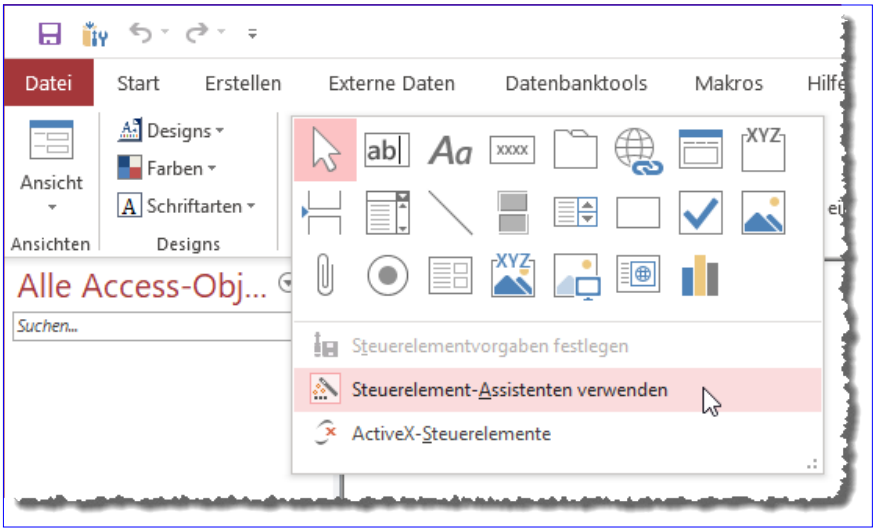


Bild 2: Aktivieren der Anzeige der Steuerelement-Assistenten

Der Befehlsschaltflächen-Assistent

Wenn Sie nun auf den Eintrag zum Anlegen einer neuen Schaltfläche klicken und dann auf die Stelle im Formularentwurf, an der die neue Schaltfläche erscheinen soll, zeigt Access den ersten Schritt des Befehlsschaltflächen-Assistenten an (siehe Bild 3). Hier können Sie im linken Listenfeld eine Kategorie auswählen

wie beispielsweise **Datensatznavigation** und im rechten Listenfeld eine der Aktionen für diese Kategorie, zum Beispiel **Datensatz suchen**.

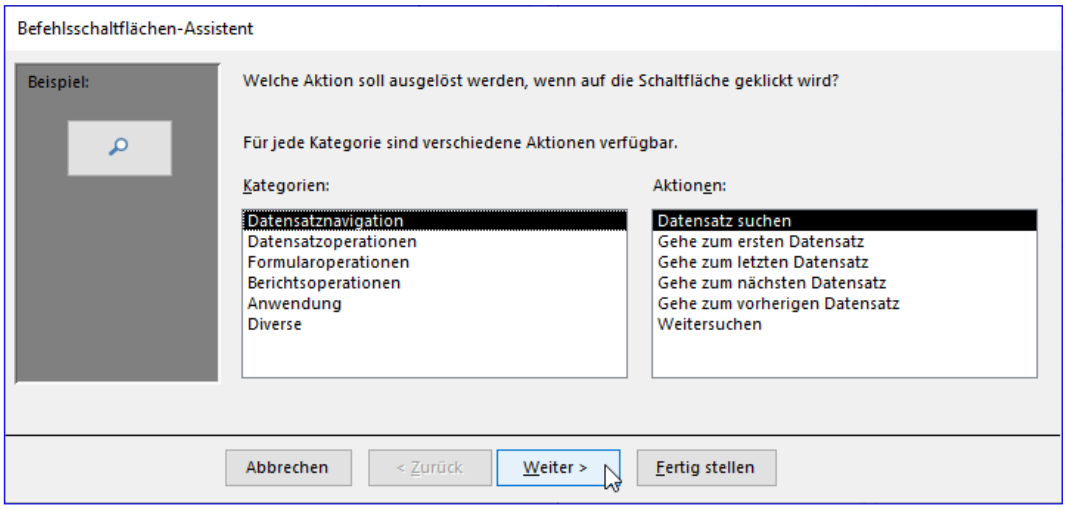


Bild 3: Erster Schritt des Befehlsschaltflächen-Assistenten

Mit einem Klick auf die Schaltfläche **Weiter** landen wir beim zweiten Schritt des Assistenten (siehe Bild 4).

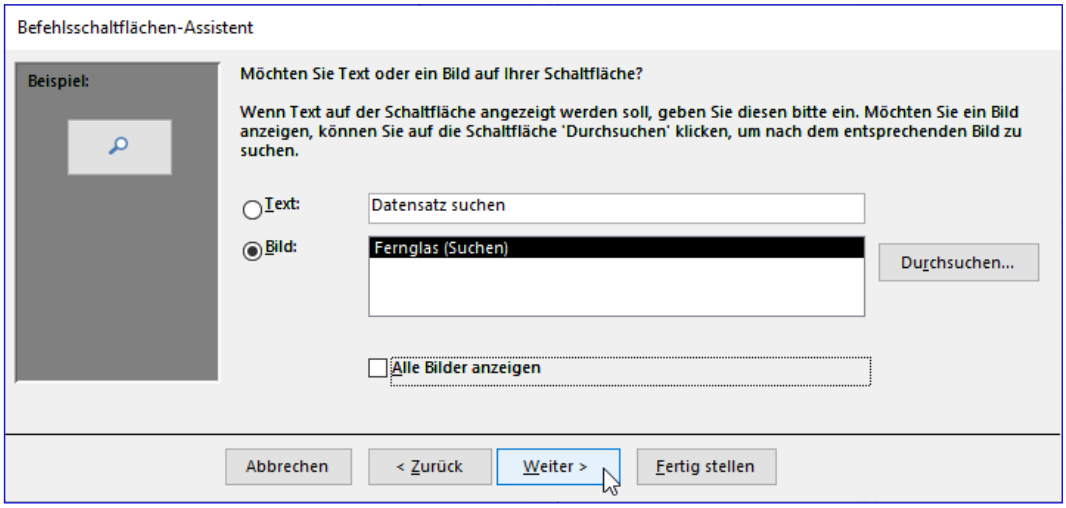


Bild 4: Auswahl von Text und Bild für die Schaltfläche

Hier finden wir zunächst die Möglichkeit, zwischen Text und Bild als Inhalt der Schaltfläche zu entscheiden. Den Text können Sie direkt neben der passenden Option in ein Textfeld eingeben beziehungsweise anpassen. Für die Auswahl eines Bildes gibt es verschiedene Möglichkeiten. Die erste ist, das angebotene Bild für diese Aktion beizubehalten. Die zweite, die Option **Alle Bilder anzeigen** zu aktivieren und im Listenfeld neben der Option **Bild** eines von mehreren Bildern zu selektieren. Wählen Sie ein anderes Bild aus, zeigt der Assistent dieses direkt in der Vorschau links an.

Sie können auch auf **Durchsuchen** klicken. Das öffnet einen **Bild auswählen**-Dialog, mit dem Sie aus dem Dateisytem **.bmp**- oder **.ico**-Dateien auswählen können. Eine so ausgewählte Datei erscheint anschließend ebenfalls in der Vorschau.

Einen Klick weiter geben Sie noch den Namen für die Befehlsschaltfläche ein. Hier schlägt der Assistent die Bezeichnung **Befehl1** vor. Wir ändern diese in **cmdDatensatzSuchen**.

Klicken Sie dann auf **Fertigstellen**, legt der Assistent die soeben konfigurierte Schaltfläche an – siehe Bild 5.

Schauen wir uns zunächst das Ergebnis des Assistenten an. Das Bild wurde direkt eingebettet. Die Schaltfläche hat den angegebenen Namen erhalten. Die Beschriftung, die ja nicht angezeigt wird, wurde entsprechend nicht geändert.

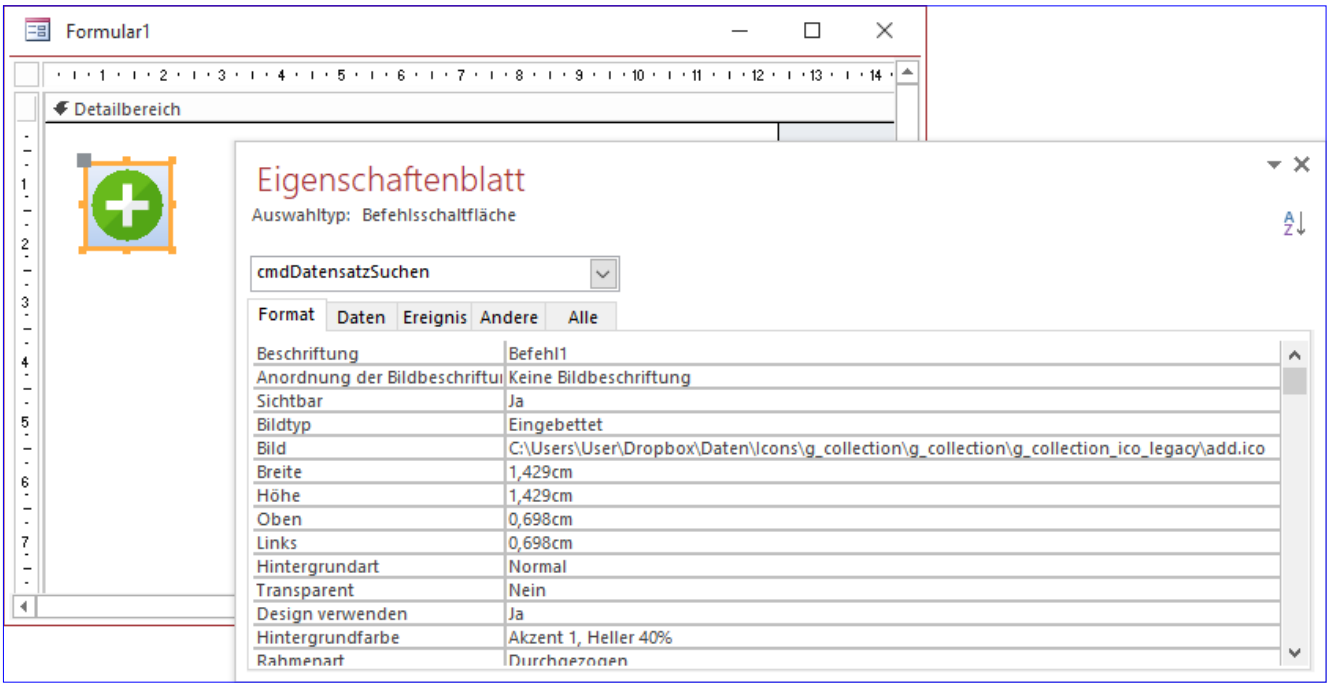


Bild 5: Eine per Befehlsschaltflächen-Assistent angelegte Schaltfläche

Ein Blick in den Bereich **Ereignis** des Eigenschaftenblatts zeigt, dass der Assistent ein Makro angelegt hat und keine VBA-Prozedur (siehe Bild 6). Wir werfen noch einen Blick auf das erstellte Makro.

Dazu klicken wir auf die Schaltfläche mit den drei Punkten neben der Eigenschaft **Beim Klicken**. Dies öffnet den Entwurf des Makros aus Bild 7, der die Makro-Variante des Suchen-Befehls definiert.

Verbesserungsideen

Beim Durchlaufen des eingebauten Befehlsschaltflächen-Assistenten sind uns die folgenden Verbesserungsvorschläge eingefallen:

- Warum sollte eine Schaltfläche nur Text oder Icon anzeigen? Wir wollen die Möglichkeit schaffen, beides anzuzeigen.

- Die Bild-Datei wird direkt in die Schaltfläche eingebettet. Access bietet aber die Möglichkeit, Bilder etwa für Schaltflächen und andere Anwendungszwecke in der Tabelle **MSysResources** zu speichern. Die dort gespeicherten Bilder können wiederverwendet werden. Das sollte unser Assistent realisieren.

- Der Benutzer hat kaum Gestaltungsmöglichkeiten bezüglich des Designs der Schaltfläche. Hier wollen wir die Vorgabe von verschiedenen Eigenschaften erlauben.

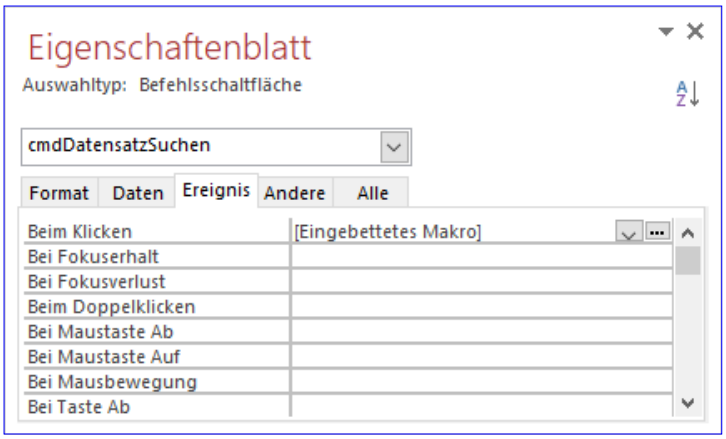


Bild 6: Der Assistent hat ein Makro für die Schaltfläche hinterlegt.

- Makros haben zwar den Vorteil, dass Einsteiger diese leichter programmieren können. Allerdings sind diese wesentlich unflexibler als VBA-Ereignisprozeduren. Daher wollen wir keine Makros verwenden.

- Der Benutzer soll verschiedene Sätze von Anweisungen anlegen können, die er beim Anlegen einer neuen Schaltfläche auswählt – oder er entscheidet sich, eine leere Ereignisprozedur zu definieren.

- Wenn der Benutzer die Beschriftung eingibt, soll nach bestimmten Kriterien direkt der Name des Steuerelements ermittelt werden.

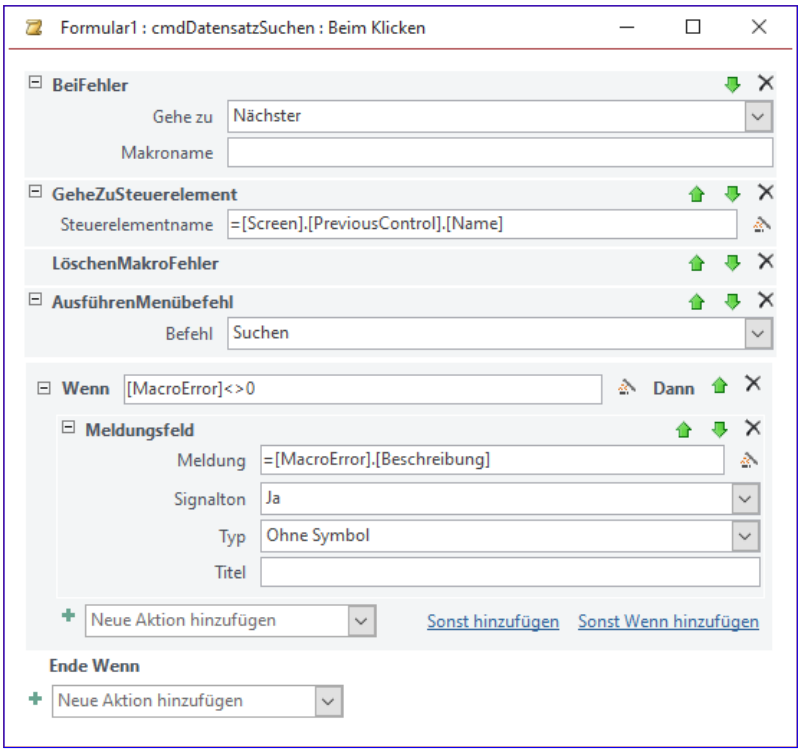


Bild 7: Das vom Assistenten angelegte Makro

Erstellen eines Steuerelement-Add-Ins

Damit machen wir uns an die Arbeit. In den ersten Schritten wollen wir den Assistenten erstellen, damit dieser beim Anlegen einer neuen Schaltfläche als Formular angezeigt wird. Dazu legen Sie zunächst eine neue Access-Datenbankdatei an und ändern ihre Dateiendung in **.accda**. Dann fügen Sie eine neue Tabelle namens **USysRegInfo** hinzu.

Diese füllen Sie wie in Bild 8. Die Daten, die in dieser Tabelle gespeichert sind, werden beim Installieren des Add-Ins über den Add-In-Manager von Access abgefragt und in die Registry eingetragen.

Die Einträge der Tabelle **USysRegInfo** haben die folgende Bedeutung. Das Feld **Subkey** enthält den Schlüssel, in

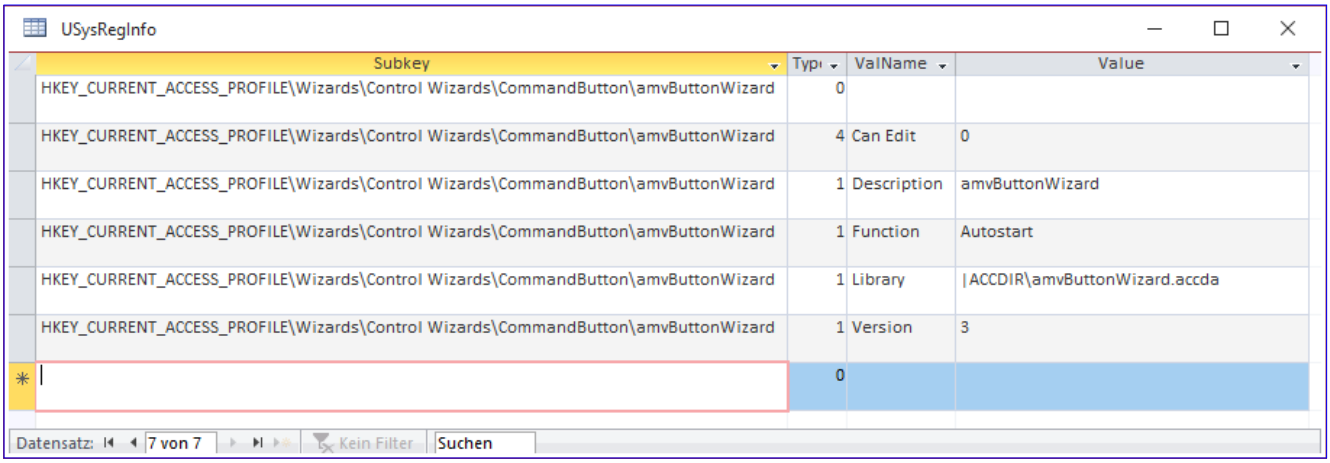


Bild 8: Die Tabelle **USysRegInfo** mit den Daten für die Registry

Schaltflächen per Code anlegen

Im Beitrag Schaltflächen-Assistent (www.access-im-unternehmen.de/1308) zeigen wir, wie Sie das Grundgerüst eines Schaltflächen-Assistenten definieren. Was Sie mit dem Schaltflächen-Assistenten anfangen können, zeigen wir Ihnen im vorliegenden Beitrag. Wir wollen zunächst das Anlegen bestimmter Standardschaltflächen erlauben. Die erste sind einfache OK- und Abbrechen-Schaltflächen. Diese Aufgabe kostet in jedem Formular, das sie neu erstellen, ein paar Minuten. Zeit, die Sie sich sparen können – indem Sie einmalig Zeit in die Entwicklung eines passenden Steuerelement-Assistenten investieren.

Ziel des Assistenten

Das Ziel des Assistenten ist das Anlegen einer neuen **OK**-Schaltfläche. Dies soll folgende Eigenschaften umfassen:

- Schaltfläche mit einem Icon (zum Beispiel grüner Haken) und dem Text **OK** als Beschriftung
- Einstellen einiger Eigenschaften wie Rahmenart, Hintergrundfarbe, Größe
- Hinzufügen einer Ereignisprozedur, die das Formular schließt oder alternativ andere Aktionen ausführt

Grundgerüst des Steuerelement-Assistenten

Das Grundgerüst des Assistenten haben wir bereits im oben genannten Beitrag beschrieben. Dieses nutzen wir als Basis für die im vorliegenden Beitrag beschriebenen Techniken.

Beim Starten des Steuerelement-Assistenten soll als Erstes ein Formular angezeigt werden, mit dem der Benutzer einige Einstellungen vornehmen kann, bevor er das Anlegen der Schaltfläche bestätigt. Dieses Formular soll auch eine Vorschau der zu erstellenden Schaltfläche anzeigen.

Formular zum Anpassen der zu erstellenden Schaltfläche

Für dieses Formular, das **frmButtonWizardOK** heißen soll, stellen wir zunächst einige Eigenschaften ein. Die Eigenschaften **Datensatzmarkierer**, **Navigationsschaltflächen**, **Trennlinien** und **Bildlaufleisten** sollen den Wert **Nein** erhalten, die Eigenschaft **Automatisch zentrieren** den Wert **Ja**.

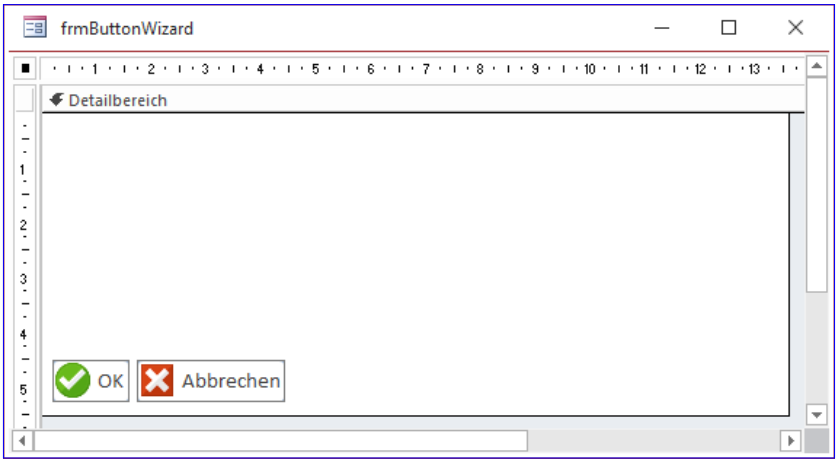


Bild 1: Erster Entwurf des Formulars zum Einstellen von Schaltflächen-Eigenschaften

Außerdem soll das Formular selbst eine **OK**-Schaltfläche sowie eine **Abbrechen**-Schaltfläche enthalten, damit der Benutzer die vorgenommenen Änderungen direkt anwenden oder auch verwerfen kann. Danach sieht das Formular wie in Bild 1 aus.

Beispiel-Schaltfläche

Nun wollen wir ja mit dem Formular festlegen, wie eine zu erstellende **OK**-Schaltfläche aussehen soll. Dazu fügen wir

Steuerelemente hinzu, mit denen wir die wichtigsten Einstellungen vornehmen. Diese haben wir in Bild 2 bereits hinzugefügt.

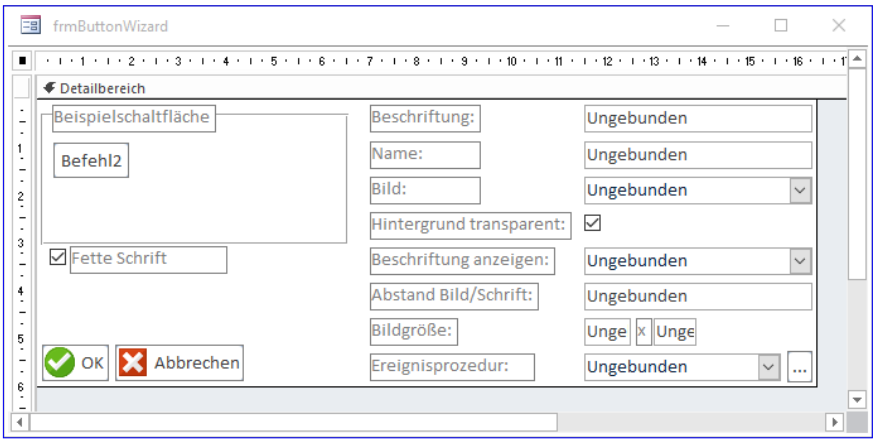


Bild 2: Formular mit Steuerelementen zum Einstellen der Schaltflächeneigenschaften

Dies sind die Steuerelemente:

- **txtBeschriftung**: Nimmt die Beschriftung für das Steuerelement entgegen.
- **txtName**: Enthält den Namen für das Steuerelement.
- **cboBilder**: Zeigt die Namen der im Add-In verfügbaren Bilder an. Diese Bilder werden in der Tabelle **MSysResources** gespeichert.
- **chkTransparent**: Gibt an, ob der Hintergrund transparent angezeigt werden soll.
- **cboBeschriftungAnzeigen**: Erlaubt die Auswahl der verschiedenen Anordnungen von Bild und Beschriftung oder auch nur die Anzeige eines Bildes.
- **txtAbstandBildSchrift**: Erlaubt das Einfügen von ein bis drei Leerzeichen zwischen Bild und Beschriftung.
- **txtBildbreite**: Breite des Bildes in Pixel
- **txtBildhoehe**: Höhe des Bildes in Pixel
- **cboEreignisprozeduren**: Dient der Auswahl der Anweisungen der Ereignisprozedur für die Schaltfläche.
- **cmdEreignisprozedurenBearbeiten**: Öffnet das Formular zum Bearbeiten der Ereignisprozeduren.

Natürlich hätten wir noch einige weitere Eigenschaften aufnehmen können, aber das würde den Rahmen dieses Beitrags sprengen.

Wichtig: Option Compare Binary statt Option Compare Database

In einer Prozedur dieses Formulars wollen wir Umlaute in Vokale umwandeln. Die Replace-Funktion ändert bei Verwendung von **Option Compare Database** im Kopf des Moduls **ä** in **ae**, aber auch **Ä** in **ae**. Damit **Ä** in **Ae** geändert wird, stellen wir im Klassenmodul des Formulars **frmButtonWizard** die Anweisung **Option Compare Binary** ein:

```
Option Compare Binary
Option Explicit
```

Hinzufügen einer Beschriftung

Das Eintragen einer Beschriftung ist der erste Schritt. Hierbei sollen direkt einige Aktionen erfolgen. Zum einen wollen wir dem Benutzer die Arbeit abnehmen, den Namen der Schaltfläche einzutragen.

Daher soll dieser dynamisch basierend auf der Beschriftung generiert werden – mehr dazu weiter unten. Außerdem soll die Größe der Vorschau immer direkt an die enthaltene Beschriftung, gegebenenfalls mit Icon, angepasst werden.

Da wir die Beschriftung auch noch in anderen Prozeduren benötigen, legen wir für diese eine modulweit deklarierte Variable namens **strBeschriftung** an:

```
Private Sub txtBeschriftung_Change()
    Dim strText As String
    Dim lngPosLeer As Long
    strText = Me!txtBeschriftung.Text
    strText = Trim(strText)
    strBeschriftung = strText
    Me!cmdBeispiel.Caption = strText
    lngPosLeer = InStr(1, strText, " ")
    Do While Not lngPosLeer = 0
        strText = Left(strText, lngPosLeer - 1) & UCase(Mid(strText, lngPosLeer + 1, 1)) & Mid(strText, lngPosLeer + 2)
        lngPosLeer = InStr(lngPosLeer + 1, strText, " ")
    Loop
    strText = UmlauteErsetzen(strText)
    Me!txtName = "cmd" & strText
    GroesseOptimieren
End Sub
```

Listing 1: Ereignisprozedur, die beim Ändern der Beschriftung ausgelöst wird

```
Dim strBeschriftung As String
```

Danach können wir in die Ereignisprozedur einsteigen, die aufgerufen wird, wenn der Benutzer den Text im Textfeld **txtBeschriftung** anpasst. Das löst mit der Eingabe eines jeden Zeichens die Prozedur aus Listing 1 aus.

Die Prozedur erfasst zunächst den aktuellen Text im Textfeld, und zwar über die Eigenschaft **Text**. Diese unterscheidet sich vom üblicherweise abgefragten Wert des Textfeldes, weil der Wert oder die Eigenschaft Value immer den zuletzt bestätigten Wert liefert. **Text** hingegen liefert den aktuell angezeigten Text. Der Text landet in der Variablen **strText**, den wir danach mit der **Trim**-Funktion von führenden und folgenden Leerzeichen befreien. Dann tragen wir den aktuellen Wert von **strText** in die Variable **strBeschriftung** ein, die wir später in weiteren Prozeduren abfragen. Außerdem landet die Beschriftung aus **strText** in der Eigenschaft **Caption** der Beispielschaltfläche **cmdBeispiel**, damit der Text direkt angezeigt wird.

CamelCase für den Schaltflächennamen

Danach folgen die Schritte, in denen wir den Namen für die Schaltfläche aus dem aktuellen Text für die Beschreibung ableiten. Der Schaltflächenname soll aus dem Präfix **cmd** plus der Beschriftung bestehen, wobei wir aus der Beschriftung die Leerzeichen entfernen, neue Wörter in der CamelCase-Notation schreiben und Umlaute in die entsprechenden Vokale umwandeln. Aus **Jetzt löschen** wird also der Schaltflächenname **cmdJetztLoeschen**.

Dazu ermitteln wir zuerst die Position des ersten Leerzeichens und speichern diese in der Variablen **lngPosLeer**. Ist diese nicht **0**, kommt also mindestens ein Leerzeichen in der Beschriftung vor, steigen wir in eine **Do While**-Schleife ein.

Diese stellt den Wert in **strText** neu zusammen, und zwar aus den Zeichen bis zum ersten Leerzeichen (**Left(strText, lngPosLeer - 1)**), dem ersten Buchstaben des Wortes nach dem Leerzeichen als Großbuchstaben (**UCase(Mid(strText, lngPosLeer + 1, 1))**) sowie dem Rest des Textes nach diesem Buchstaben (**Mid(strText, lngPosLeer + 2)**).

Aus **Jetzt alles schließen** wird dann **JetztAlles schließen**. Danach prüfen wir innerhalb der **Do While**-Schleife, ob es noch ein Leerzeichen im Ausdruck gibt. Das ist bei unserem Beispiel der Fall, also ist **lngPosLeer** noch nicht

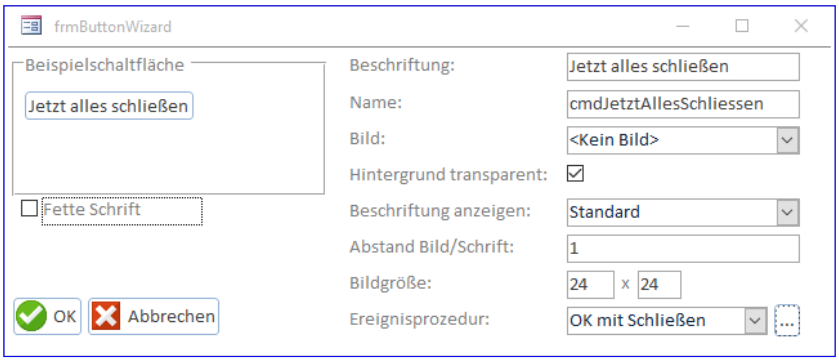


Bild 3: Zwischenergebnis nach der Eingabe der Beschriftung

0 und wir durchlaufen die **Do While**-Schleife ein zweites Mal. Danach hat **strText** den Inhalt **JetztAllesSchließen**.

Damit verlässt die Prozedur die Schleife und ruft die Funktion **UmlauteErsetzen** auf, die wie folgt aussieht:

```
Public Function UmlauteErsetzen(ByVal strText As String) _
                                                As String
    strText = Replace(strText, "ä", "ae")
    strText = Replace(strText, "ö", "oe")
    strText = Replace(strText, "ü", "ue")
    strText = Replace(strText, "Ä", "Ae")
    strText = Replace(strText, "Ö", "Oe")
    strText = Replace(strText, "Ü", "Ue")
    strText = Replace(strText, "ß", "ss")
    UmlauteErsetzen = strText
End Function
```

Die Funktion ersetzt alle Umlaute durch Vokale und **ß** durch **ss**, was in unserem Beispiel zum Ergebnis **JetztAllesSchliessen** führt. Dem stellt die Ereignisprozedur noch das Präfix **cmd** voran: **cmdJetztAllesSchliessen**. Schließlich wird die Prozedur **GroesseOptimieren** aufgerufen, welche die Größe der Beispielschaltfläche anpasst.

Die Prozedur **GroesseOptimieren** schauen wir uns weiter unten an, da diese noch weitere Steuerelemente berücksichtigt, die wir noch nicht erstellt haben. In Bild 3 sehen Sie einen Zwischenstand.

Namen der Schaltfläche ändern

Wenn Sie den nun automatisch aus der Beschriftung der Schaltfläche abgeleiteten Namen ändern möchten, können Sie dies problemlos tun. Beachten Sie nur, dass dieser bei erneuter Änderung der Beschriftung wieder von der Beschriftung abgeleitet wird. Sie können bei Bedarf eine Option einbauen, welche die nachträgliche Änderung des Namens verhindert, wenn dieser einmal festgelegt wurde. Wir wollen es an dieser Stelle allerdings nicht zu kompliziert machen.

Bild auswählen

Bei der Bereitstellung der Bilder wollen wir eine eingebaute Funktion von Access nutzen. Dabei können Sie in der Entwurfsansicht eines Formulars den Ribbon-Befehl **Entwurf|Steuerelemente|Bild einfügen|Durchsuchen...** nutzen, um Bilder zur Datenbank hinzuzufügen (siehe Bild 4).

Diese können wir beim Programmieren der **.accda**-Datenbank zunächst nutzen, um ein paar als Icon zu verwendende Bilddateien in der Datenbank zu speichern. Die Bilder landen dann in einem Anlagefeld in der Tabelle **MSysResources**.

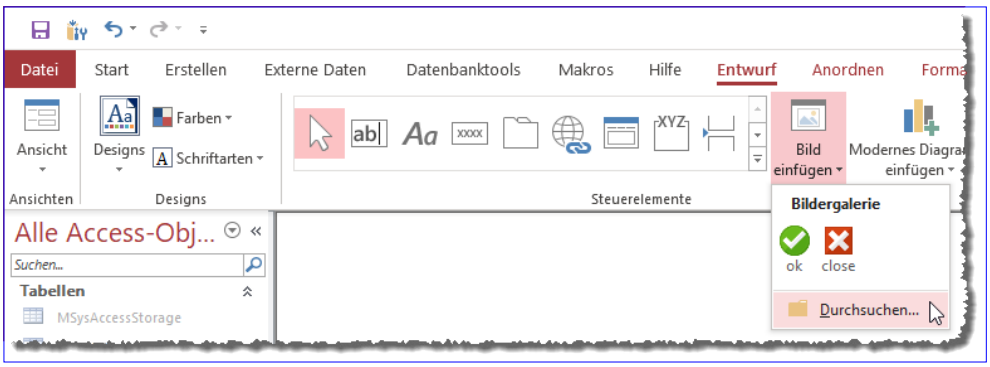


Bild 4: Auswählen von Bildern für die aktuelle Datenbank

Diese Tabelle verwenden wir in der Abfrage für die Datensatzherkunft des Kombinationsfeldes **cboBilder** als Datenquelle. Und wir setzen noch einen drauf, denn wir wollen auch einen Eintrag mit dem Text **<Kein Bild>** anzeigen. Daher benötigen wir eine **UNION**-Abfrage, deren SQL-Anweisung wie folgt aussieht:

```
SELECT 0 AS ID, "<Kein Bild>" AS Name, "" AS Type
FROM MSysResources
UNION
SELECT Id, Name, Type
FROM MSysResources WHERE Type="img"
```

Dies liefert ein Ergebnis wie in Bild 5.

Für das Kombinationsfeld **cboBilder** stellen wir die Eigenschaften **Spaltenanzahl** auf **2** und **Spaltenbreiten** auf **0cm** ein. Danach können Sie mit dem Kombinationsfeld die Bilder wie in Bild 6 auswählen. Das funktioniert für den Moment, später werden wir hier noch nachjustieren müssen.

Das Aktualisieren des Kombinationsfeldes löst das Ereignis **Nach Aktualisierung** mit der folgenden Ereignisprozedur aus. Diese ruft zwei Prozeduren auf: **BildEinstellen**, um das neue Bild festzulegen, und die bereits oben einmal aufgerufene Prozedur **GroesseOptimieren**. Diese soll die Größe an den geänderten Inhalt anpassen:

```
Private Sub cboBilder_AfterUpdate()
    BildEinstellen
    GroesseOptimieren
End Sub
```

Die Prozedur **BildEinstellen** prüft, ob das Kombinationsfeld **cboBilder** den Wert **0** hat, was dem Text **<Kein Bild>** entspricht. In diesem Fall stellt es die Eigenschaft **Picture** der Beispielschaltfläche auf eine leere Zeichenkette ein. In allen anderen Fällen weist die Prozedur der

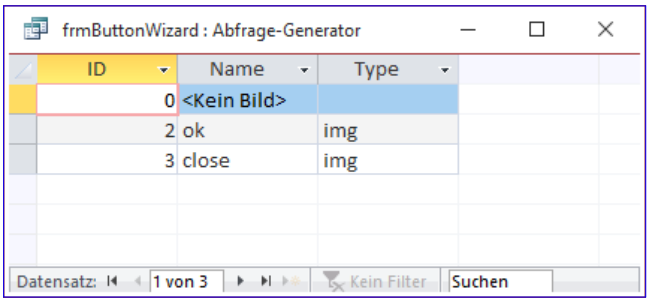


Bild 5: Datensatzherkunft für das Kombinationsfeld **cboBilder**

Eigenschaft **Picture** den Namen des Bildes aus der Tabelle **MSysResources** zu, also beispielsweise **ok** oder **close**:

```
Private Sub BildEinstellen()
    If Me!cboBilder = 0 Then
        Me!cmdBeispiel.Picture = ""
    Else
        Me!cmdBeispiel.Picture = Me!cboBilder.Column(1)
    End If
End Sub
```

Hintergrund transparent schalten

Das Kontrollkästchen **chkTransparent** soll die Eigenschaft **BackStyle** entweder auf den Wert **0** einstellen (transparent) oder auf den Wert **1** (ausgefüllt).

Dies wird direkt nach dem Wählen eines neuen Wertes auf die Beispielschaltfläche angewendet:

```
Private Sub chkTransparent_AfterUpdate()
    If Me!chkTransparent Then
        Me!cmdBeispiel.BackStyle = 0
    Else
```

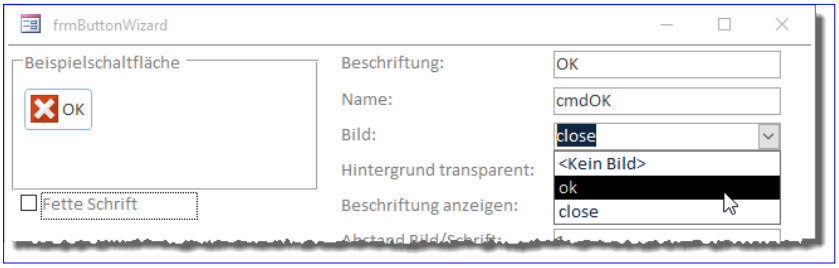


Bild 6: Auswahl von Bildern mit dem Kombinationsfeld **cboBilder**

```
        Me!cmdBeispiel.BackStyle = 1
    End If
End Sub
```

Beschriftungsanzeige einstellen

Das Kombinationsfeld **cboBeschriftungAnzeigen** erlaubt es, einen der sechs vorgesehenen Werte für die Eigenschaft **Anordnung der Bildbeschriftung** vorzunehmen. Die sechs Werte tragen wir direkt in die Eigenschaft **Datensatzherkunft** des Kombinationsfeldes ein:

```
"0";"Keine Beschriftung";"1";"Standard";2;"Beschriftung
oben";3;"Beschriftung unten";4;"Beschriftung links";5;"Be-
schriftung rechts"
```

Damit nur die Texte und nicht die Zahlen angezeigt werden und die Liste überhaupt korrekt verarbeitet wird, legen Sie die Eigenschaft **Herkunftstyp** auf **Wertliste**, **Spaltenanzahl** auf **2** und **Spaltenbreiten** auf **0cm** fest.

Nach der Auswahl eines der Werte wird der passende Zahlenwert der VBA-Eigenschaft **PictureCaptionArrangement** zugewiesen. Anschließend erfolgt der obligatorische Aufruf der Prozedur **GroesseOptimieren**:

```
Private Sub cboBeschriftungAnzeigen_AfterUpdate()
    Me!cmdBeispiel.PictureCaptionArrangement = _
                                Me!cboBeschriftungAnzeigen
    GroesseOptimieren
End Sub
```

Abstand zwischen Bild und Schrift einstellen

Wenn Sie einer Schaltfläche ein Icon und einen Text zuweisen, kann es sein, dass der Platz zwischen beiden Elementen etwas zu klein ist. Dann bietet es sich an, dem Text ein oder mehrere Leerzeichen voranzustellen. Das erledigen wir mit dem Kombinationsfeld **cboAbstandBildSchrift**, das die Werte **0;1;2;3** als Wertliste anzeigt. Nach der Auswahl eines Wertes mit diesem Feld feuert das Ereignis **Nach Aktualisierung** und löst die folgende Ereignisprozedur aus:

```
Private Sub cboAbstandBildSchrift_AfterUpdate()
    Dim strLeerzeichen As String
    strLeerzeichen = Space(Me!cboAbstandBildSchrift)
    Me!cmdBeispiel.Caption = _
                        strLeerzeichen & strBeschriftung
    GroesseOptimieren
End Sub
```

Die Prozedur speichert eine Zeichenkette mit der dem Zahlenwert entsprechen Menge Leerzeichen in der Variablen **strLeerzeichen**. Dann fügt sie diese vor der Beschriftung aus **strBeschriftung** ein und weist das Ergebnis der Eigenschaft **Caption** der Beispielschaltfläche zu. Schließlich ruft auch diese Prozedur die Routine **GroesseOptimieren** auf.

Bildgröße eingeben

Wir könnten theoretisch die zu verwendenden Bilder analysieren und die Größe per Code ermitteln. Das ist uns aber an dieser Stelle zu aufwendig, weshalb wir dem Benutzer selbst die Möglichkeit geben, die Bildabmessungen in die beiden Textfelder **txtBildbreite** und **txtBildhoehe** einzugeben.

Die Eingabe löst die folgenden Prozeduren aus, die jeweils die Routine **GroesseOptimieren** aufrufen:

```
Private Sub txtBildbreite_AfterUpdate()
    GroesseOptimieren
End Sub

Private Sub txtBildhoehe_AfterUpdate()
    GroesseOptimieren
End Sub
```

Fett dargestellte Beschriftung

Das Kontrollkästchen **chkFetteSchrift** gibt an, ob die Beschriftung der Schaltfläche fett angezeigt werden soll. Die Änderung des Wertes des Kontrollkästchens löst diese Prozedur aus, die das Format der Beschriftung der Beispielschaltfläche anpasst:

```
Private Sub chkFetteSchrift_AfterUpdate()
    Me!cmdBeispiel.FontBold = Me!chkFet-
teSchrift
    GroesseOptimieren
End Sub
```

Ereignisprozedur auswählen

Schließlich wollen wir dem Benutzer noch die Möglichkeit geben, den Inhalt der Ereignisprozedur, die durch das Ereignis **Beim Klicken** ausgelöst wird, festzulegen. Dazu haben wir eine Tabelle vorgesehen, um verschiedene Anweisungen zu speichern.

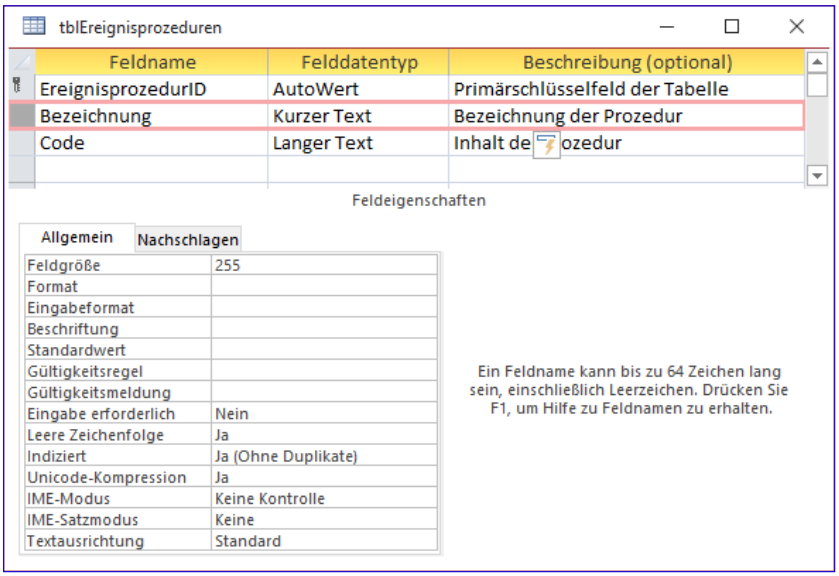


Bild 7: Entwurf der Tabelle zum Speichern der Prozedurinhalte

Diese Tabelle enthält die Felder **EreignisprozedurID**, **Bezeichnung** und **Code** und sieht im Entwurf wie in Bild 7 aus. Für das Feld **Bezeichnung** haben wir einen eindeutigen Index festgelegt, da jede Bezeichnung nur einmal vorkommen darf.

Für das Kombinationsfeld **cboEreignisprozeduren** legen wir die folgende Abfrage auf Basis der Tabelle **tblEreignisprozeduren** als Datensatzherkunft fest:

```
SELECT 0 AS EreignisprozedurID,
'<Keine Ereignisprozedur>' AS Bezeichnung
FROM tblEreignisprozeduren
UNION
SELECT EreignisprozedurID, Bezeichnung
FROM tblEreignisprozeduren
ORDER BY Bezeichnung;
```

```
Private Sub cmdEreignisprozedurenBearbeiten_Click()
    Dim lngEreignisprozedurID As Long
    If Nz(Me!cboEreignisprozeduren, 0) = 0 Then
        DoCmd.OpenForm "frmEreignisprozeduren", WindowMode:=acDialog, datamode:=acFormAdd
    Else
        DoCmd.OpenForm "frmEreignisprozeduren", WindowMode:=acDialog, WhereCondition:="EreignisprozedurID = " _
            & Me!cboEreignisprozeduren, datamode:=acFormEdit
    End If
    If IstFormularGeoeffnet("frmEreignisprozeduren") Then
        lngEreignisprozedurID = Nz(Forms!frmEreignisprozeduren!EreignisprozedurID, 0)
        If Not lngEreignisprozedurID = 0 Then
            Me!cboEreignisprozeduren = lngEreignisprozedurID
        End If
        DoCmd.Close acForm, "frmEreignisprozeduren"
    End If
End Sub
```

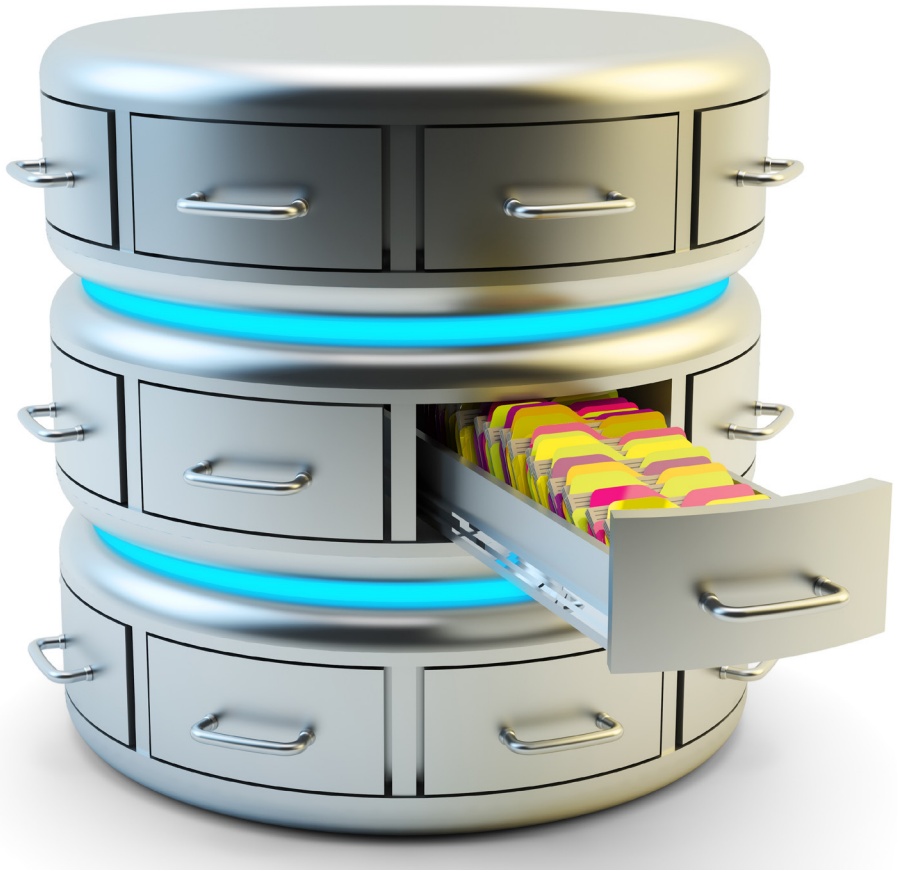
Listing 2: Ereignisprozedur zum Anzeigen des Dialogs für die Verwaltung von Ereignisprozeduren

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

www.access-im-unternehmen.de
Mat-Nr. H1583-5126

ARCHIV SEITE 233

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.

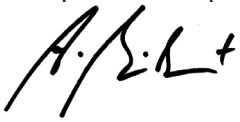
Das Ergebnis dieses Beitrags greifen wir im nächsten Beitrag mit dem Titel **Selektionen kombinieren** wieder auf (ab Seite 8). Hier zeigen wir Ihnen, wie Sie verschiedene Selektionen – beispielsweise alle Artikel der Kategorie Getränke und der Kategorie Süßigkeiten – kombinieren können. Außerdem lassen 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.

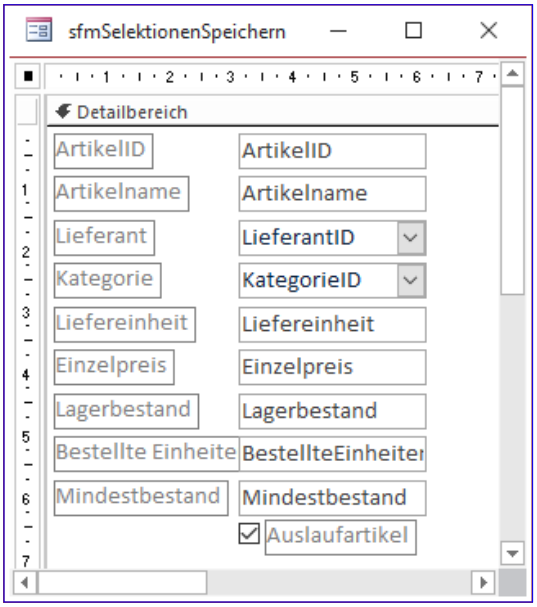


Bild 1: Entwurf des Unterformulars

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...**

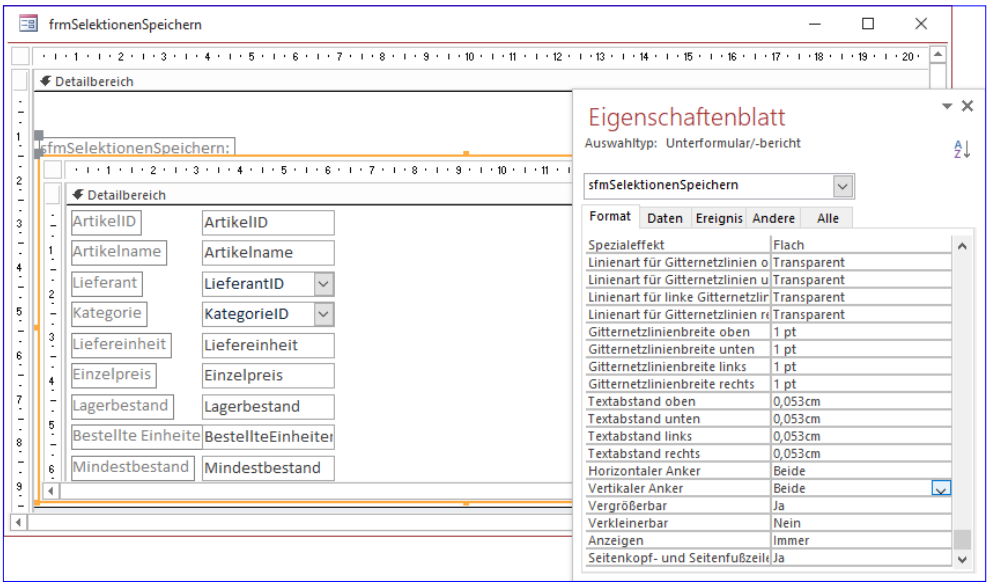


Bild 2: Einfügen des Unterformulars in das Hauptformular

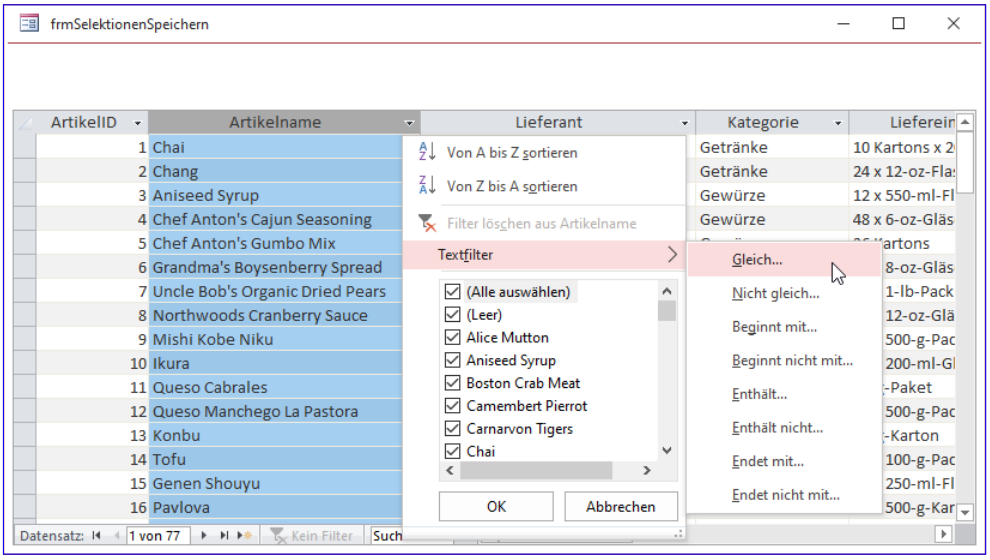


Bild 3: Einstellen von Filter- und Sortierkriterien

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:

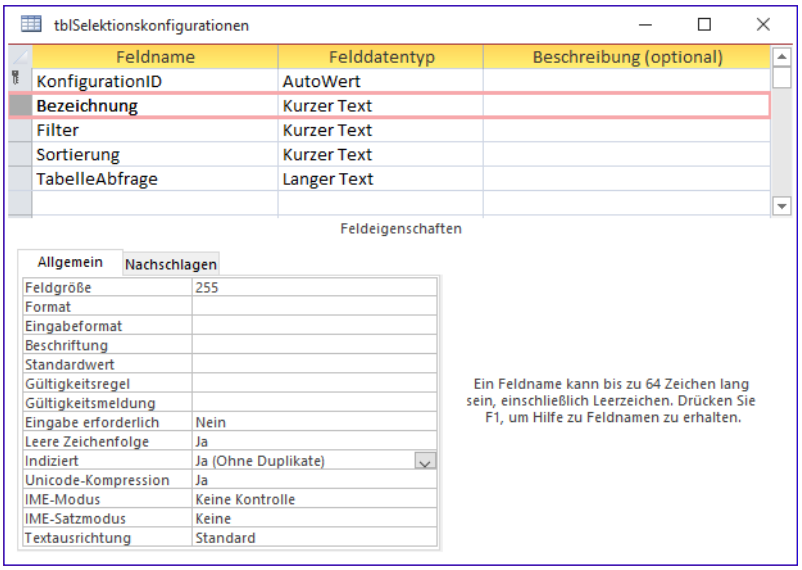


Bild 4: Einfügen des Unterformulars in das Hauptformular

- **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.

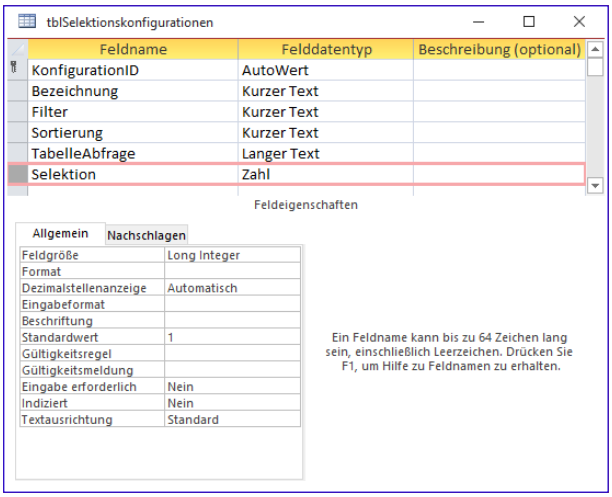


Bild 2: Erweitern der Tabelle **tblSelektionskonfigurationen**

Formular zur Auswahl der Listen

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

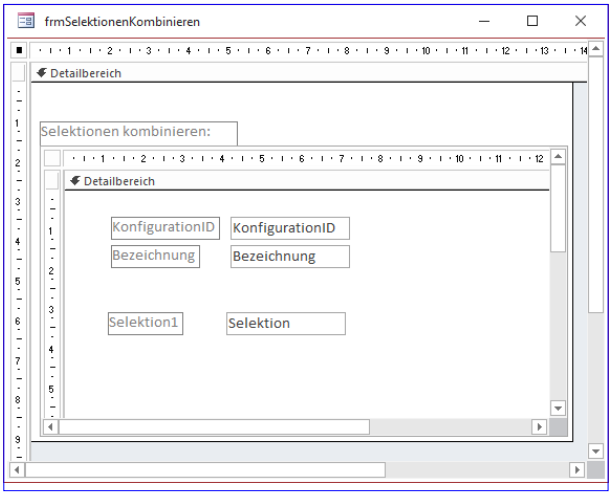


Bild 3: Entwurf des Formulars zur Anzeige der zu wählenden Selektionen

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).

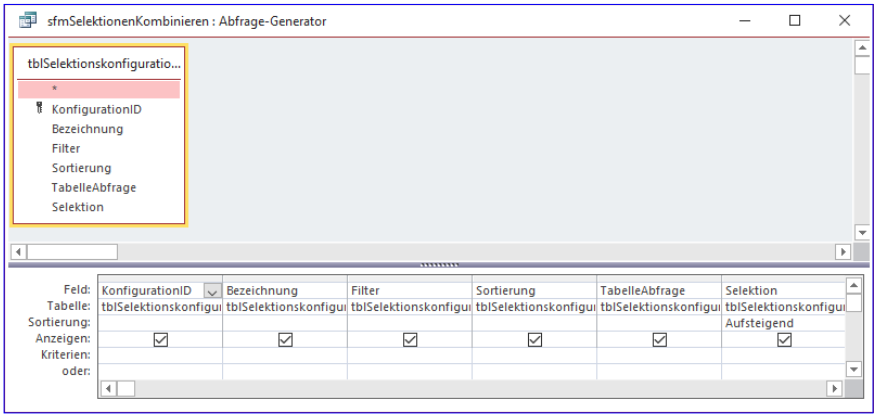


Bild 4: Abfrage für das Unterformular

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:

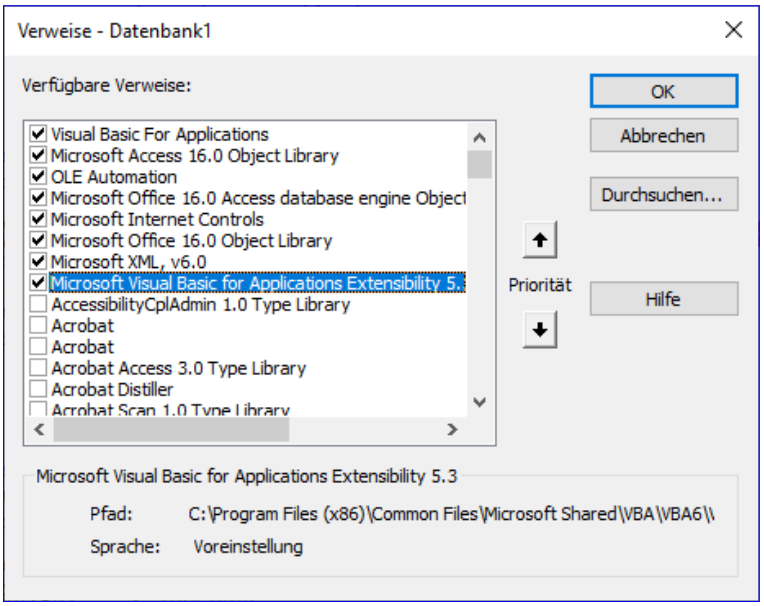


Bild 1: Verweis auf die Bibliothek **Microsoft Visual Basic for Applications 5.3 Extensibility**

```
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.

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.

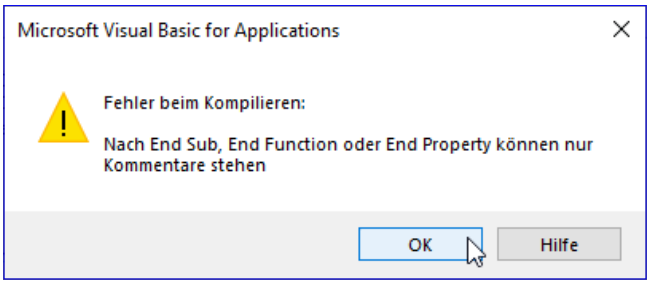


Bild 2: Fehler beim Deklarieren einer Variablen hinter einer Routine

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(
                            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.

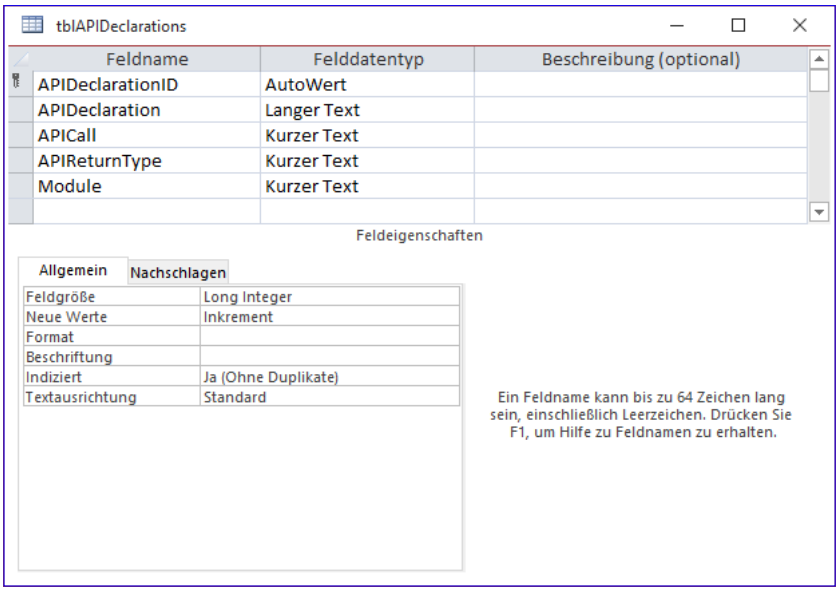


Bild 3: Entwurf der Tabelle zum Speichern der API-Deklarationen

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**

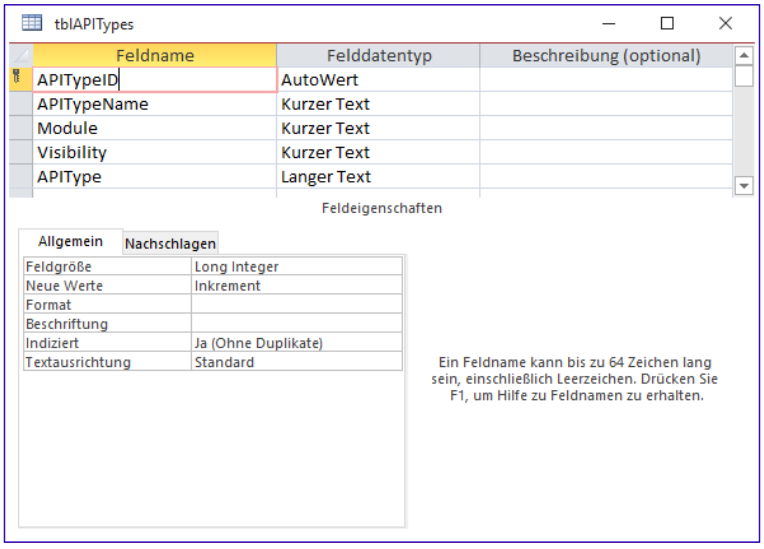


Bild 1: Entwurf der Tabelle **tblAPITypes**

```
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

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.

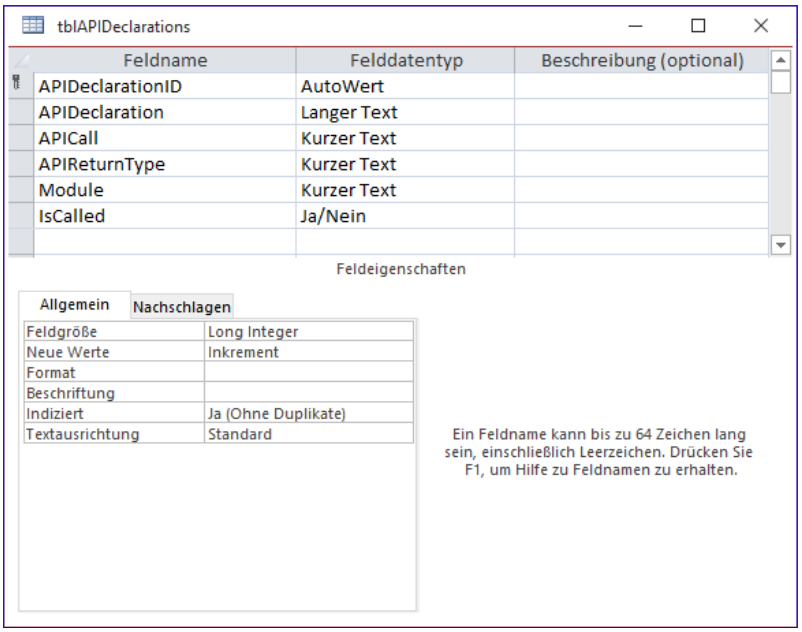


Bild 1: Tabelle zum Speichern der API-Deklarationen eines VBA-Projekts

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).

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

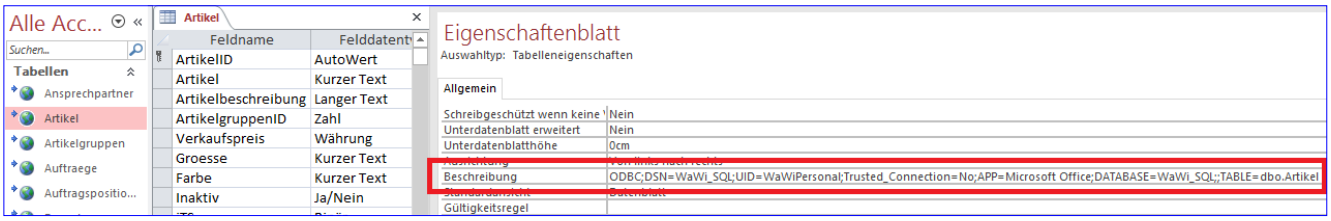


Bild 1: Beschreibung zur Datenquelle

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.

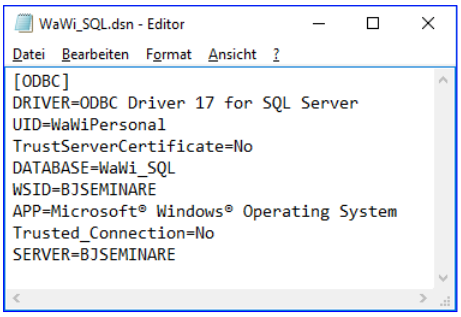


Bild 2: Inhalt einer Datei-DSN

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**.

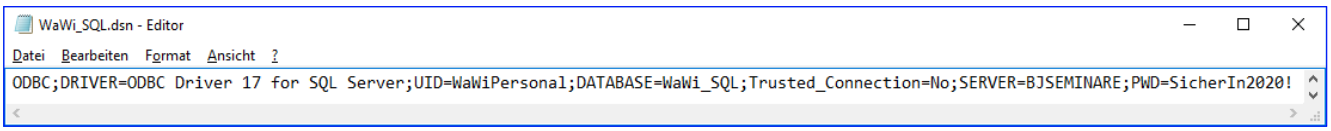


Bild 3: Die Verbindungszeichenfolge

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.

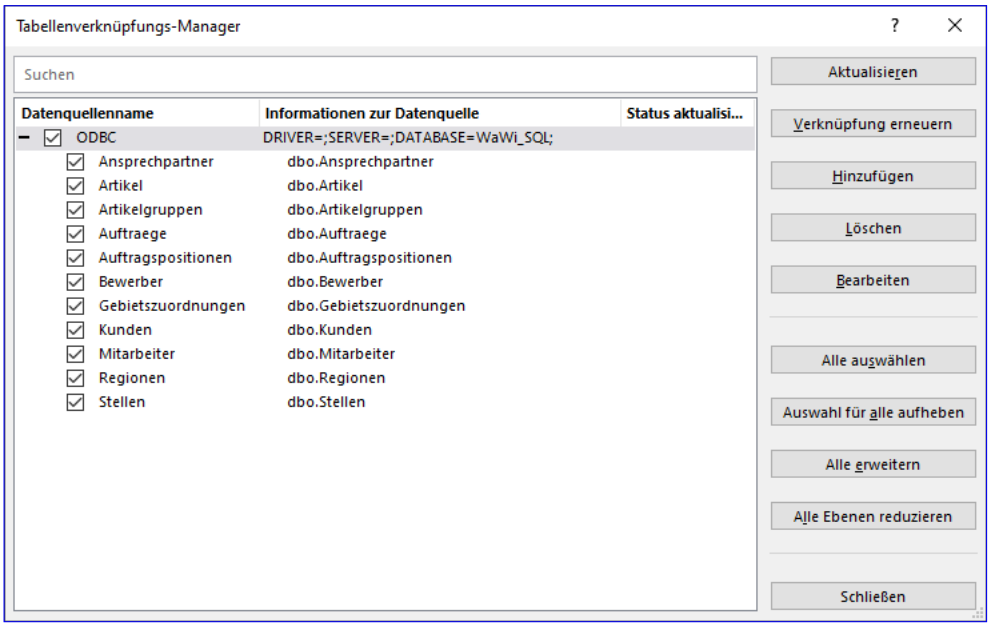


Bild 4: Der Tabellenverknüpfungs-Manager

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.

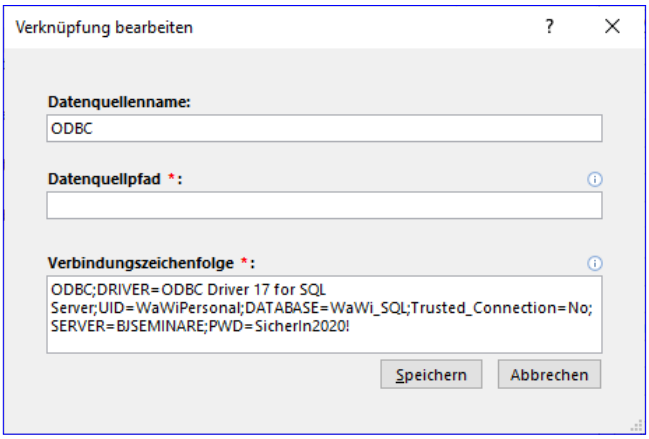


Bild 5: DSN-less per Tabellenverknüpfungs-Manager

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**

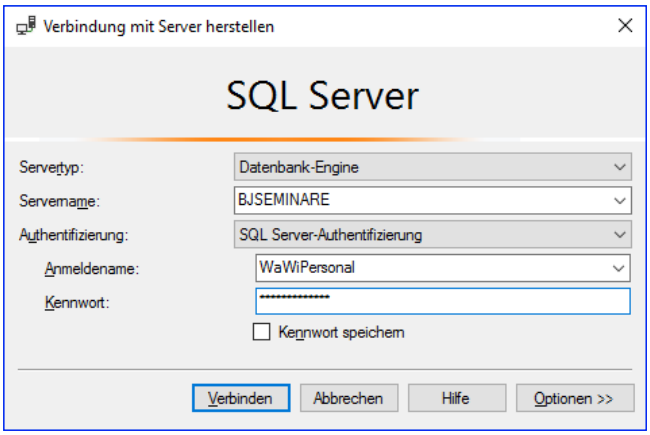


Bild 7: Die Anmeldung am SSMS

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-

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.

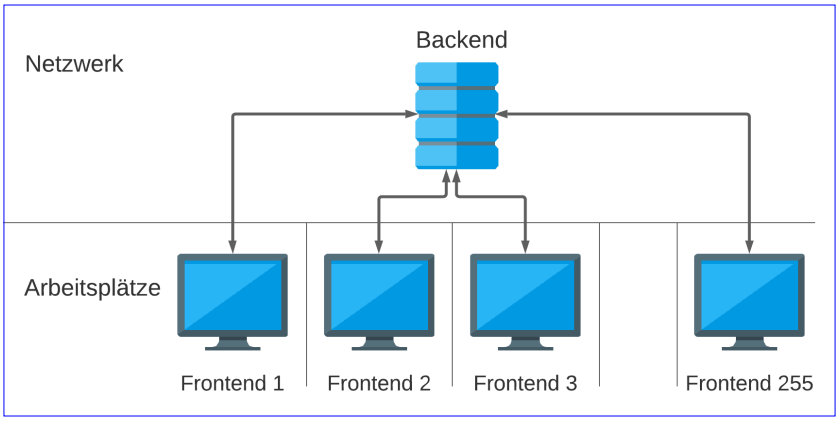


Bild 1: Frontend und Backend im Netzwerk

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

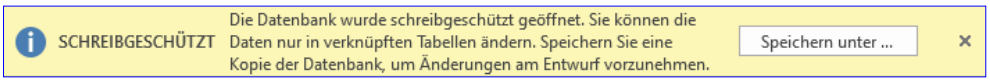


Bild 2: Warnmeldung beim Öffnen

- 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

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).

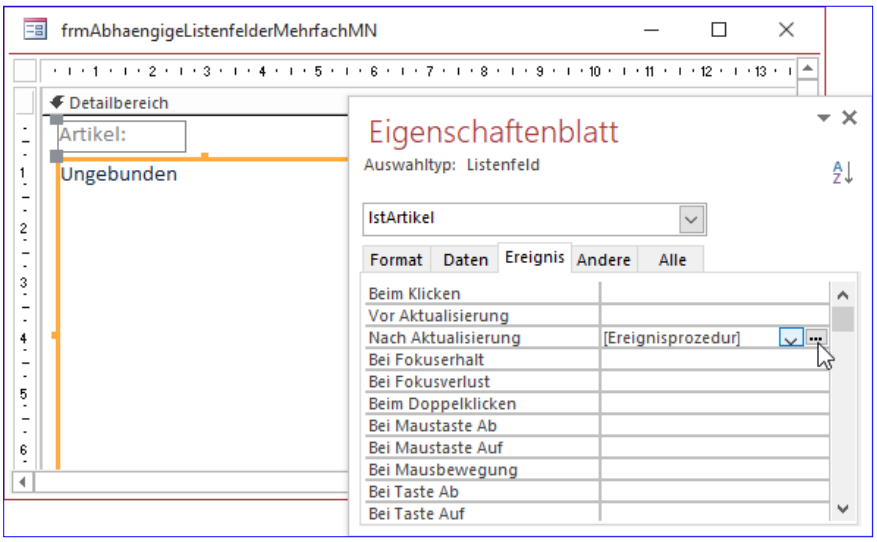


Bild 1: Üblicher Weg zum Anzeigen der Ereignisprozedur eines Steuerelements

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).

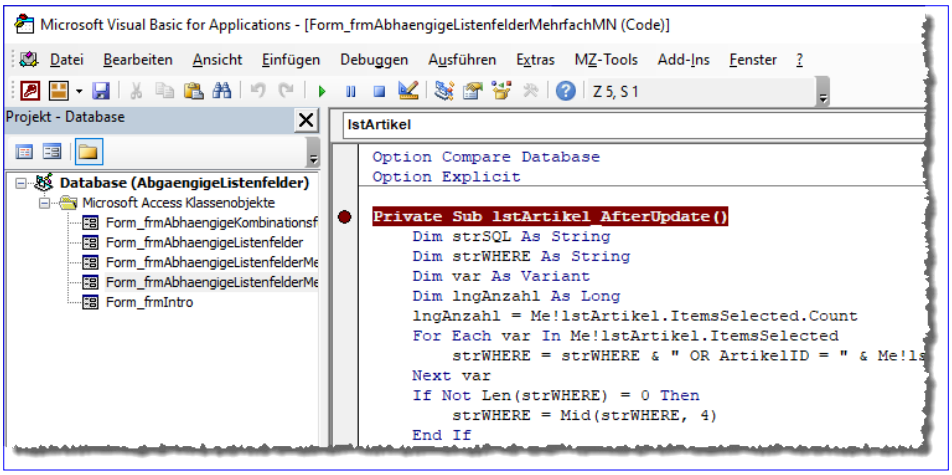


Bild 2: Setzen eines Haltepunktes in der ersten Zeile einer Ereignisprozedur

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.

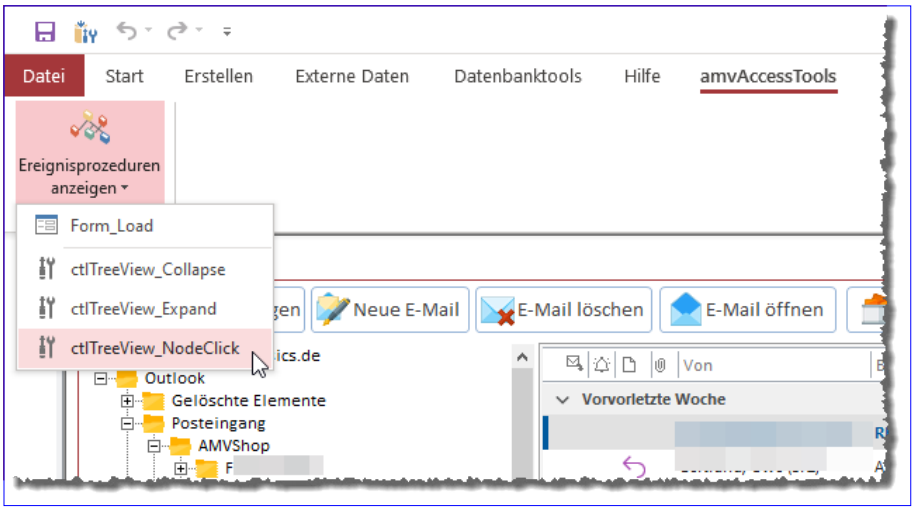


Bild 3: Anzeige der Ereignisprozedur per Ribbon-Befehl

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.

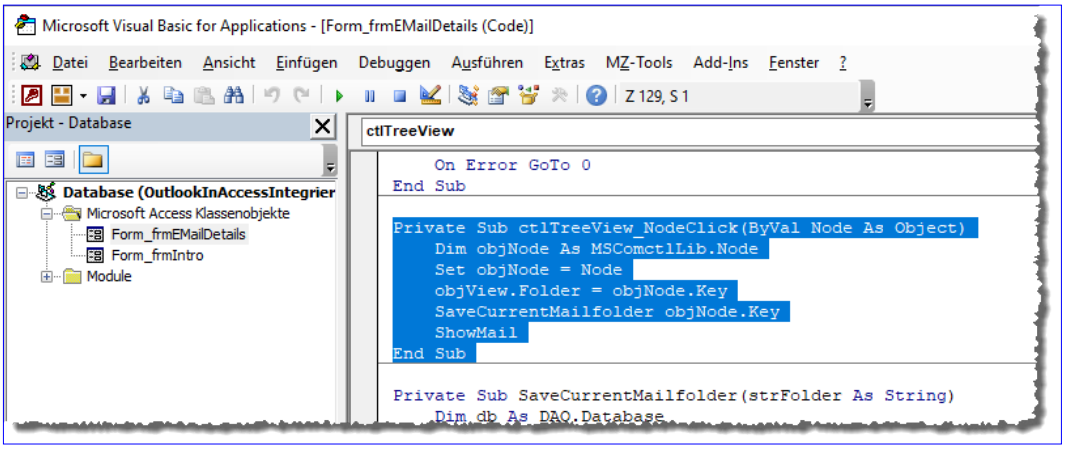


Bild 4: Anzeige und Markierung der Ereignisprozedur

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**

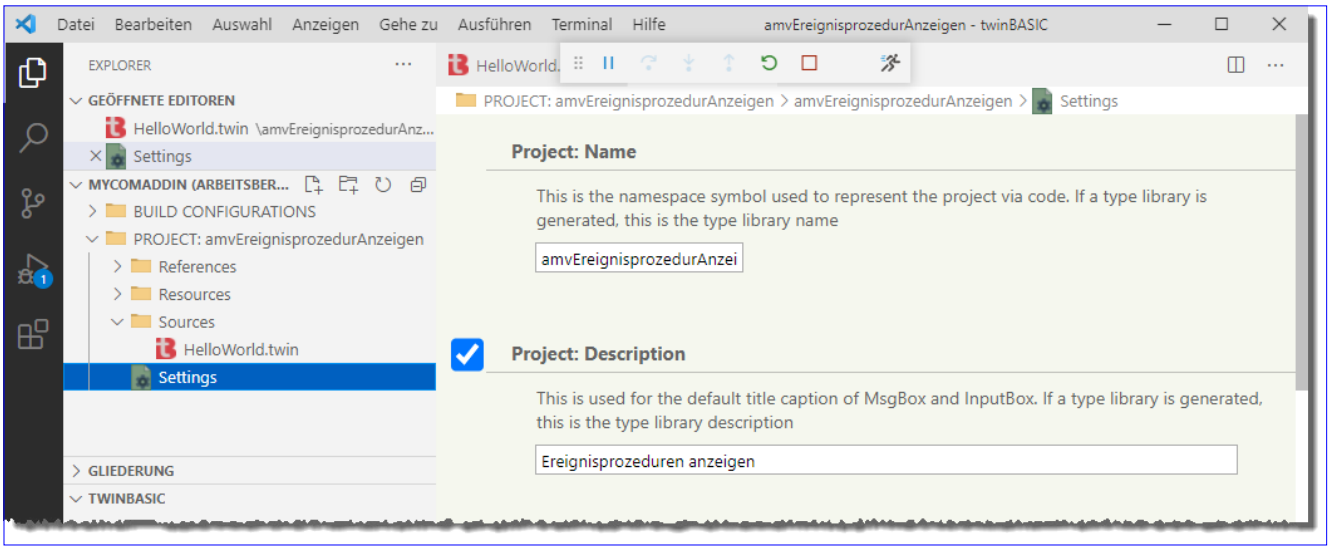


Bild 5: Anpassen der Benennung von Elementen

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.

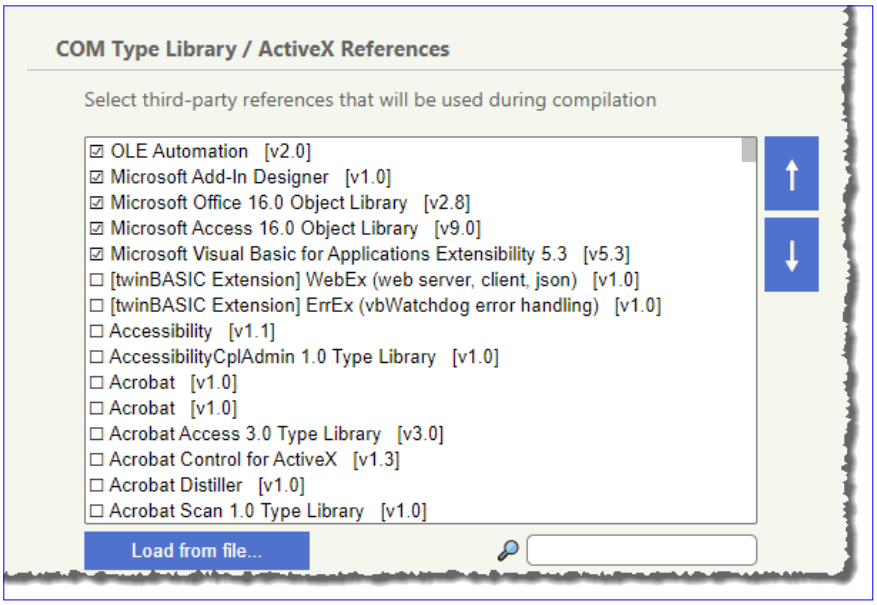


Bild 6: Hinzufügen von Verweisen

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 = z
```

HAUFE.
Ausgabe 05/2021

ACCESS

IM UNTERNEHMEN

ACCESS-OPTIONEN IM RIBBON

Ändern Sie wichtige Optionen schnell im Ribbon statt umständlich den Optionen-Dialog aufzurufen (ab S. 60)

In diesem Heft:

ACCESS-ANWENDUNGEN WEITERGEBEN, TEIL 2

Nutzen Sie InnoSetup, um praktische Setups zum Weitergeben Ihrer Anwendungen zu erstellen.

SEITE 51

OPTIONSGRUPPEN LEEREN

Fügen Sie Optionsgruppen die Funktion zum Entfernen der aktuell gewählten Option hinzu.

SEITE 2

REGISTRY PER VBA UND API

Greifen Sie per VBA auf die Registry von Windows hinzu – unter 32-Bit und 64-Bit!

SEITE 29

www.access-im-unternehmen.de
Mat-Nr. H1583-5127

ARCHIV SEITE 309

Optionen per Ribbon

Nervt Sie das auch, wenn Sie immer wieder umständlich den Optionen-Dialog von Access öffnen müssen, um oft verwendete Optionen einzustellen? Datei-Menü aufrufen, auf Optionen klicken, dann überlegen, in welchem Bereich sich die Option überhaupt befindet und so weiter. Wie schön wäre es doch, wenn oft verwendete Optionen einfach im Ribbon angezeigt würden, wo man sie direkt per Mausklick einstellen kann. Genau dafür finden Sie eine Lösung in der aktuellen Ausgabe: Ein COM-Add-In, welches das Ribbon um genau diese Funktion erweitert!

Im Beitrag **Access-Optionen per Ribbon ändern** finden Sie ab Seite 60 eine genaue Beschreibung, wie Sie mit dem neuen twinBASIC ein COM-Add-In programmieren, das einige wichtige Access-Optionen direkt in einem eigenen Reiter im Ribbon anzeigt. Das heißt, Sie bekommen nicht nur eine sofort einsetzbare Lösung, sondern auch noch die Anleitung, wie Sie diese Lösung programmieren und an Ihre Bedürfnisse anpassen können! Immerhin wird nicht jeder von uns die gleichen Optionen regelmäßig nutzen.

Im Rahmen dieser Lösung haben wir uns ein wenig Arbeit gemacht und einmal geschaut, wie Sie die im Optionen-Dialog von Access enthaltenen Optionen überhaupt per VBA ändern können. Der Beitrag **Optionen per VBA für Access 2019** zeigt ab Seite 8, wie Sie die einzelnen Optionen anpassen können. Dabei gibt es grundsätzlich zwei Möglichkeiten: für Optionen die aktuelle Datenbank betreffend über Properties der Datenbank und für allgemeine Access-Optionen die Registry. Anhand dieses Beitrags können Sie für alle Optionen entnehmen, wie Sie diese per VBA ändern können.

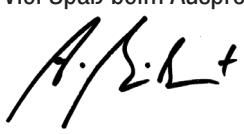
Einige der allgemeinen Access-Optionen befinden sich in einem speziellen Bereich der Registry, den Sie mit speziellen VBA-Befehlen lesen und beschreiben können. Andere Optionen, etwa diejenigen, die sich auf alle Office-Anwendungen beziehen, finden Sie an anderen Stellen in der Registry. Diese können Sie nur über API-Funktionen manipulieren. Deshalb haben wir uns auch angesehen, wie dies gelingt – das Ergebnis lesen Sie ab Seite 29 unter dem Titel **Registry per VBA, 32- und 64-Bit**.

Da wir in unserer Lösung Ribbon-Steuerelemente zum Einstellen von Optionen hinzufügen, nutzen wir auch einige Callbackfunktionen. Diese lauten für COM-Add-Ins allerdings anders als für VBA. Deshalb haben wir den Aufbau dieser Funktionen samt Parametern angeschaut. Das Resultat liefert der Beitrag **Ribbon: Callback-Signaturen für VBA und VB6** ab Seite 40.

Das in der vorherigen Ausgabe begonnene Thema Weitergabe von Access-Anwendungen führen wir ab Seite 51 mit dem Beitrag **Setup für Access: Umsetzung mit InnoSetup** fort.Das Steuerelement **Optionsgruppe** bietet eine einfache Möglichkeit zur Auswahl eines Wertes aus mehreren Optionen. Allerdings können Sie diese nach einmal erfolgter Auswahl nicht mehr leeren. Dafür haben wir eine Lösung entwickelt, die Sie unter **Optionsgruppe leeren mit Klasse** ab Seite 2 lesen.

Schließlich schauen wir uns noch an, wie der Datei speichern-Dialog 64-Bit-kompatibel realisiert wird (**API-Funktion GetSaveFileDialog (32-Bit und 64-Bit)**, ab Seite 23), was eigentlich mit 32-Bit- und 64-Bit-Versionen gemeint ist (**32-Bit, 64-Bit, VBA-Version und Co.** ab Seite 28) und Sie **Benutzerdefinierte Bilder in twinBASIC anzeigen** (ab Seite 48).

Viel Spaß beim Ausprobieren!



Ihr André Minhorst

Optionsgruppe leeren mit Klasse

Optionsgruppen sind praktische Steuerelemente für die Auswahl einiger weniger, vorab fest definierter Optionen. Leider bietet dieses Steuerelement nach einmaliger Auswahl nicht mehr die Möglichkeit, dieses wieder zu leeren. Im vorliegenden Beitrag schauen wir uns an, wie das grundsätzlich zu erledigen ist. Außerdem erstellen wir eine Klasse, mit der Sie den dazu benötigten Code für die Nutzung in mehreren Optionsgruppen wiederverwenden können, statt ihn jedes Mal zu reproduzieren.

Vorbereitung

Für die Beispiele in diesem Beitrag legen wir ein Formular mit einer Optiongruppe namens **ogrOptionen** an. Dieser fügen wir drei Optionen hinzu, deren Wert für die Eigenschaft **Optionswert** wir auf **1**, **2** und **3** festlegen und die wir **opt1**, **opt2** und **opt3** benennen. Der Entwurf dieses Formulars sieht wie in Bild 1 aus.

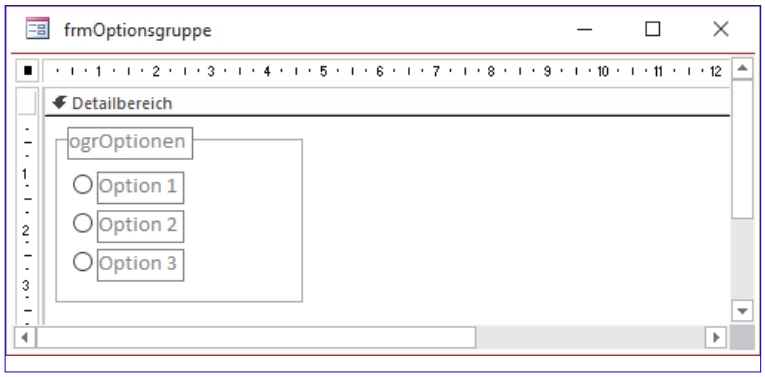


Bild 1: Entwurf des Formulars mit der Optionsgruppe

Dieses Formular kopieren wir direkt einmal, und zwar unter dem Namen **frmOptionenMitKlasse**. Wir schauen uns also zuerst an, wie wir das Leeren der Optionsgruppe direkt im Klassenmodul des Formulars erledigen.

Danach programmieren wir Klassen, mit denen wir das Verhalten abbilden und wiederverwendbar machen, und wenden diese in der Kopie des Formulars an.

Gewünschtes Verhalten

Wir wollen erreichen, dass der Benutzer sowohl mit der Maus als auch mit der Tastatur eine einmal getätigte Auswahl in der Optionsgruppe wieder rückgängig machen kann.

Dazu untersuchen wir zwei Ereignisse: das Loslassen der linken Maustaste sowie das Betätigen der Leertaste. Hier prüfen wir jeweils, ob die aktuelle Option gerade aktiviert ist und nur in diesem Fall soll die Optionsgruppe geleert werden.

Ereignisse zum Abbilden des Verhaltens

Dazu nutzen wir zwei Ereignisse der jeweiligen Optionsfelder, nämlich **Bei Maustaste auf** und **Bei Taste auf**. Diese legen wir für alle vorhandenen Optionsfelder an. Für das erste Optionsfeld sieht das im ersten Schritt dann wie folgt aus:

```
Private Sub opt1_MouseUp(Button As Integer, _
                Shift As Integer, X As Single, Y As Single)

End Sub

Private Sub opt1_KeyUp(KeyCode As Integer, _
                                    Shift As Integer)

End Sub
```

Hier brauchen wir nun Action. Für das Loslassen des Mauszeigers sieht das wie folgt aus:

- Prüfen, ob die linke Maustaste gedrückt wurde
- Prüfen, ob die Maus vor dem Loslassen die aktuell selektierte Option angeklickt hat
- Dann die Optionsgruppe leeren

Für das Ereignis **Bei Taste auf** stehen folgende Aufgaben an:

- Prüfen, ob die Leertaste gedrückt wurde
- Prüfen, ob beim Betätigen der Leertaste der Fokus auf der aktuell selektierten Option liegt
- Dann die Optionsgruppe leeren

Individueller Code für diese Optionsgruppe

Die Lösung für die eingangs beschriebene Optionsgruppe finden Sie in Listing 1. Wir kümmern uns zunächst um die Mausereignisse. Dazu schauen wir uns den Code für das Ereignis **opt1_MouseUp** an. Die Prozedur vergleicht zunächst den Wert des Parameters **Button** mit dem Wert der Konstanten

```
Private Sub opt1_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
    If Button = acLeftButton Then
        UpdateOptiongroup Me!ogrOptionen, Me!opt1
    End If
End Sub

Private Sub opt2_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
    If Button = acLeftButton Then
        UpdateOptiongroup Me!ogrOptionen, Me!opt2
    End If
End Sub

Private Sub opt3_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
    If Button = acLeftButton Then
        UpdateOptiongroup Me!ogrOptionen, Me!opt3
    End If
End Sub

Private Sub opt1_KeyUp(KeyCode As Integer, Shift As Integer)
     Select Case KeyCode
         Case vbKeySpace
             UpdateOptiongroup Me!ogrOptionen, Me!opt1
     End Select
End Sub

Private Sub opt2_KeyUp(KeyCode As Integer, Shift As Integer)
     Select Case KeyCode
         Case vbKeySpace
             UpdateOptiongroup Me!ogrOptionen, Me!opt2
     End Select
End Sub

Private Sub opt3_KeyUp(KeyCode As Integer, Shift As Integer)
     Select Case KeyCode
         Case vbKeySpace
             UpdateOptiongroup Me!ogrOptionen, Me!opt2
     End Select
End Sub

Private Sub UpdateOptiongroup(ogr As OptionGroup, opt As OptionButton)
     If ogr = opt.OptionValue Then
         ogr = Null
     End If
End Sub
```

Listing 1: Prozeduren, um die aktuelle Option wieder abzuwählen

acLeftButton und prüft damit, ob der Benutzer überhaupt die linke Maustaste gedrückt hat.

Ist das der Fall, ruft die Ereignisprozedur eine weitere Prozedur namens **UpdateOptiongroup** auf und übergibt dieser als Parameter einen Verweis auf die Optionsgruppe sowie auf das angeklickte Optionsfeld. Diese Prozedur prüft, ob der Wert der Optionsgruppe (aus dem Parameter **ogr**) mit dem Wert des Optionsfeldes (aus dem Parameter **opt**) übereinstimmt. In diesem Fall stellt sie den Wert der Optionsgruppe aus dem Parameter **ogr** auf **Null** ein.

Auf die gleiche Weise programmieren wir die Prozeduren für das Ereignis **Bei Maustaste auf** der anderen Optionsfelder. Die Prozeduren **opt1_KeyUp**, **opt2_KeyUp** und **opt3_KeyUp** werden beim Betätigen von Tasten ausgelöst. Hier prüfen wir, welche Taste der Benutzer gedrückt hat, denn das Ereignis wird beim Betätigen jeder Taste ausgelöst. Also vergleichen wir den Wert des Parameters **KeyCode** mit der Konstanten für die Leertaste, **vbKeySpace**. Sind die beiden gleich, rufen wir wieder die bereits beschriebene Prozedur **UpdateOptiongroup** auf und übergeben die entsprechenden Parameter.

Optionsgruppe leeren wiederverwendbar machen

Nun haben sie bereits gesehen, dass es relativ viel Aufwand ist, den Code für eine Optionsgruppe zu schreiben. Wollen Sie den Code für eine weitere Optionsgruppe nutzen, reicht einfaches Kopieren nicht aus, denn Sie müssen diesen auch noch an die jeweiligen Steuerelementbezeichnungen und Optionswerte anpassen.

Deshalb wollen wir zwei Klassen programmieren, welche dies automatisch erledigen – und die Sie für jede betroffene Optionsgruppe einfach nur noch zu deklarieren, initialisieren und zuzuweisen brauchen.

Wir brauchen zwei Klassen, weil wir erstens die Optionsgruppen damit erfassen wollen und dort in einer Auflistung in einer zweiten Klasse die Optionsfelder. In dieser zweiten Klasse definieren wir die Ereignisse, die durch **Bei Maustaste auf** und **Bei Taste auf** ausgelöst werden.

Klasse zum Kapseln der Optionsfelder

Die Klasse, mit der wir jeweils ein Optionsfeld erfassen, heißt **clsOptionbutton**. Diese Klasse finden Sie in der Übersicht in Listing 2.

Sie deklariert eine Variable namens **m_OptionButton** für das **OptionButton**-Element, das sie aufnehmen soll, und legt für dieses das Schlüsselwort **WithEvents** fest. Dadurch können wir in dieser Klasse Ereignisprozeduren für das **OptionButton**-Element definieren. Außerdem definiert sie eine weitere Variable namens **m_OptionGroup**. Diese soll einen Verweis auf das **OptionGroup**-Steuerelement erhalten, den wir später in den Ereignisprozeduren benötigen.

Die erste **Property Set**-Methode bietet die Möglichkeit, der Klasse einen Verweis auf das zu verwendende **OptionButton**-Steuerelement zu übergeben, und zwar mit dem Parameter **opt**. Den Inhalt dieses Parameters weist die Prozedur der Variablen **m_OptionButton** zu. Anschließend legt sie für die beiden Eigenschaften **OnMouseUp** und **OnKeyUp** den Wert **[Event Procedure]** fest.

Das ist der Schritt neben dem Schlüsselwort **WithEvents**, um nicht nur das Anlegen, sondern auch das Auslösen von Ereignisprozeduren für dieses Steuerelement in der aktuellen Klasse zu ermöglichen.

Die zweite **Property Set**-Prozedur erlaubt das Zuweisen eines Verweises auf das betroffene **OptionGroup**-Steuerelement zu der Variablen **m_OptionGroup**.

Nun fehlen noch die Ereignisprozeduren. Diese legen Sie an, indem Sie im Kopf des Klassenmoduls im linken Kombinationsfeld den Eintrag **m_OptionButton** auswählen und im rechten einmal **OnMouseUp** und einmal **OnKeyUp**. Beim ersten Auswählen von **m_OptionButton** wird automatisch die Ereignisprozedur für das **Click**-Ereignis

Optionen per VBA für Access 2019

In einem früheren Beitrag namens »Access-Optionen gestern und heute« haben wir uns einmal angesehen, welche Access-Optionen es gibt und wie Sie diese per VBA einstellen können – unter anderem mit den Methoden SetOption oder über die Eigenschaften des Database-Objekts der aktuell geöffneten Datenbank. Damals ging es noch um den Optionen-Dialog von Access 2003, der sich mittlerweile stark verändert hat. Um diese aktualisierte Version soll es in diesem Beitrag gehen. Grundlage ist dabei die Access 365-Version von Mitte 2021.

Wo findet man die Optionen per VBA?

Bevor wir in die Referenz der Optionen und ihrer VBA-Pendants einsteigen, schauen wir uns an, wie Sie überhaupt Optionen per VBA lesen und schreiben können. Dazu brauchen wir zunächst eine wichtige Unterscheidung: Es gibt Optionen für die Access-Anwendung und für die jeweils geöffnete Datenbank.

Optionen in der Registry

Die Optionen der Access-Anwendung werden in der Registry im Bereich **HKEY_CURRENT_USER** gespeichert und wirken sich somit nach der Änderung ausschließlich auf die vom aktuellen Benutzer geöffneten Access-Instanzen aus. Diese können wir mit den beiden VBA-Befehlen **GetOption** und **SetOption** lesen und schreiben.

Genaugenommen finden Sie die Optionen in der Registry unter **HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\16.0\Access\Settings**. Die meisten dieser Optionen ändern Sie in der Benutzeroberfläche über die verschiedenen Elemente des Dialogs **Access-Optionen**, ein paar auch über den Dialog **Navigationsoptionen**.

Optionen in den Eigenschaften des Database-Objekts

Die Optionen der Access-Datenbank wiederum beziehen sich nur auf die aktuell geöffnete Access-Datenbank. Die meisten davon befinden sich im Dialog **Access-Optionen** im Bereich **Aktuelle Datenbank**. Tatsächlich werden die Werte für diese Optionen in der **Properties**-Auflistung des **Database**-Objekts der aktuellen Datenbank gespeichert. Sie können diese mit folgender Anweisung einstellen:

```
CurrentDb.Properties("<Eigenschaftsname>") = <Eigenschaftswert>
```

Eine nicht vorhandene Eigenschaft müssen Sie zuvor noch anlegen. Mit folgender Anweisung fragen Sie den Wert im Direktbereich von Access ab:

```
Debug.Print CurrentDb.Properties("<Eigenschaftsname>")
```

Optionen in weiteren Bereichen der Registry

Einige Optionen, etwa die für alle Office-Anwendungen gültigen Optionen, befinden sich in anderen Bereichen der Registry. Für diese gibt es keine einfache VBA-Anweisung wie **SetOption** oder **GetOption**. Um diese zu lesen oder zu schreiben, benötigen Sie spezielle Befehle, die wir in einem weiteren Beitrag namens **Registry per VBA, 32- und 64-Bit** (**www.access-im-unternehmen.de/1323**) untersuchen.

Access-Optionen und ihre VBA-Pendants

Damit steigen wir gleich in die Optionen ein, die Sie über die Benutzeroberfläche anpassen können, und schauen uns an, wie Sie diese unter VBA einstellen können.

Wir haben die Eigenschaften jeweils in den Screenshots der Dialoge nummeriert. Sie finden zu den Nummern die deutsche Bezeichnung (in Klammern), die englische

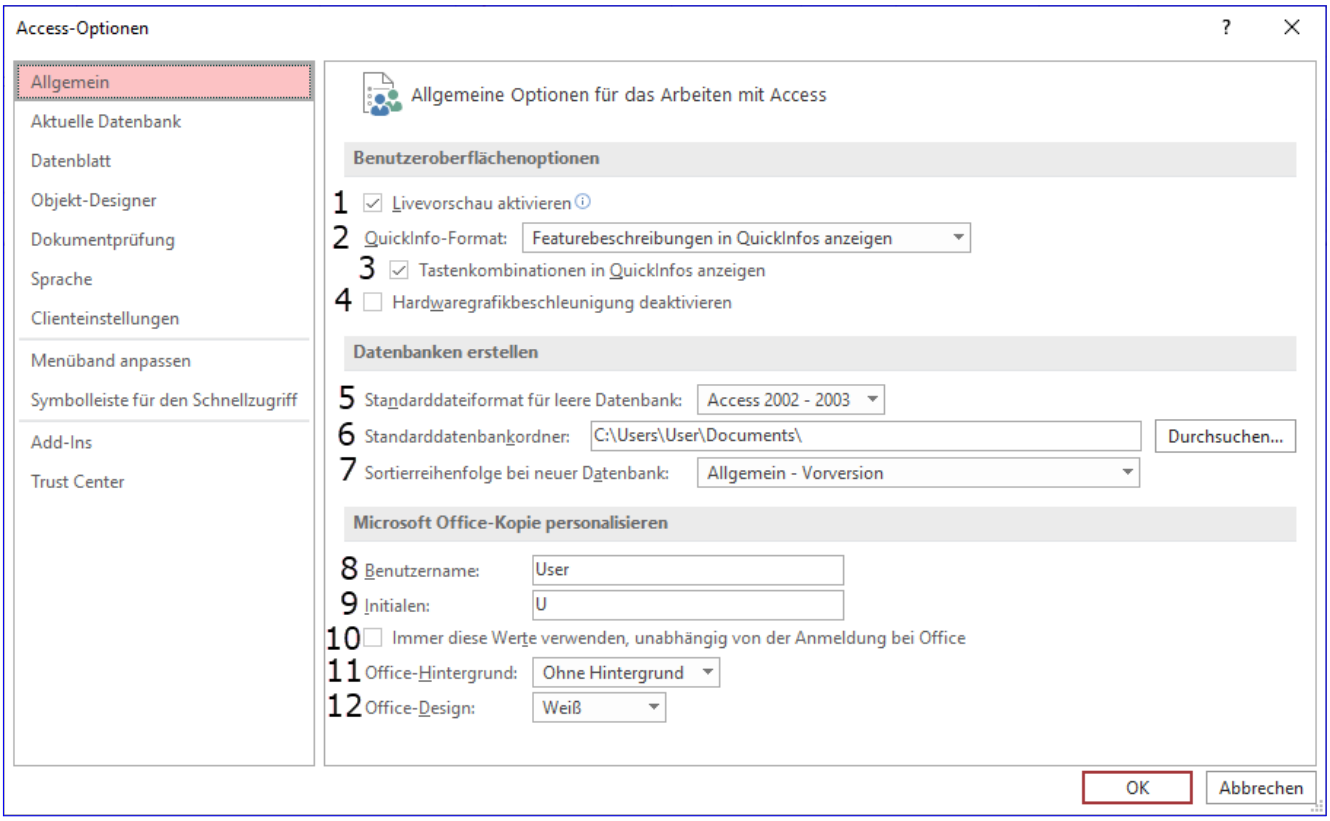


Bild 1: Optionen des Bereichs **Allgemein**

Bezeichnung, den Speicherort der Eigenschaft (Registry/Database oder im Falle von anderen Speicherorten in der Registry denentsprechenden Zweig) und die möglichen Werte.

Optionen des Bereichs Allgemein

Gleich im ersten Bereich des Optionen-Dialogs von Access finden wir einige Optionen, die sich tatsächlich nicht von Access aus per VBA steuern lassen (siehe Bild 1).

- **1** (**Live-Vorschau aktivieren**): **LivePreview**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung angelegt
- **2** (**QuickInfo-Format**): nicht gefunden
- **3** (**Tastenkombinationen in QuickInfos anzeigen**): nicht gefunden
- **4** (**Hardwaregrafikbeschleunigung deaktivieren**):
- **5** (**Standarddateiformat für leere Datenbank**): **Default File Format**, Registry, **Access 2000**: **9**, **Access 2002 - 2003**: **10**, **Access 2007 - 2019, Access 365**: **12**
- **6** (**Standarddatenbankordner**): **Default Database Directory**, Registry, Verzeichnis im String-Format
- **7** (**Sortierreihenfolge bei neuer Datenbank**): **New Database Sort Order**, Registry, verschiedene Werte, zum Beispiel **2052** (**Allgemein - Vorversion**) oder **1033** (**Deutsches Telefonbuch**)
- **8** (**Benutzername**): Office-Einstellung
- **9** (**Initialen**): Office-Einstellung
- **10** (**Immer diese Werte verwenden, unabhängig von der Anmeldung bei Office**): Office-Einstellung

- **11** (**Office-Hintergrund**): Office-Einstellung
- **12** (**Office-Design**): Office-Einstellung

Optionen des Bereichs Aktuelle Datenbank

Die Elemente im Bereich **Aktuelle Datenbank** des Optionen-Dialogs von Access sehen Sie in Bild 2.

Hier sind die Schrauben, an denen Sie mit VBA drehen können:

- **1** (**Anwendungstitel**): **AppTitle**, Database, Anwendungstitel im String-Format, wird bei erster Änderung erstellt
- **2** (**Anwendungssymbol**): **AppIcon**, Database, Pfad zum Icon im String-Format, wird bei erster Änderung erstellt
- **3** (**Als Formular- und Berichtssymbol verwenden**): **UseAppIconForFrmRpt**, Database, Boolean, Neustart erforderlich
- **4** (**Formular anzeigen**): **StartupForm**, Database, Formularname im **String**-Format, wird bei erster Änderung erstellt, Neustart erforderlich
- **5** (**Webanzeigeformular**): im Desktop-Modus nicht verfügbar
- **6** (**Statusleiste anzeigen**): **StartUpShowStatusBar**, Database, Boolean, Neustart erforderlich
- **7** (**Dokumentfensteroptionen**): **UseMdiMode**, Database, **0**: **Dokumente im Registerkartenformat**, **1**: **Überlappende Fenster**, Neustart erforderlich

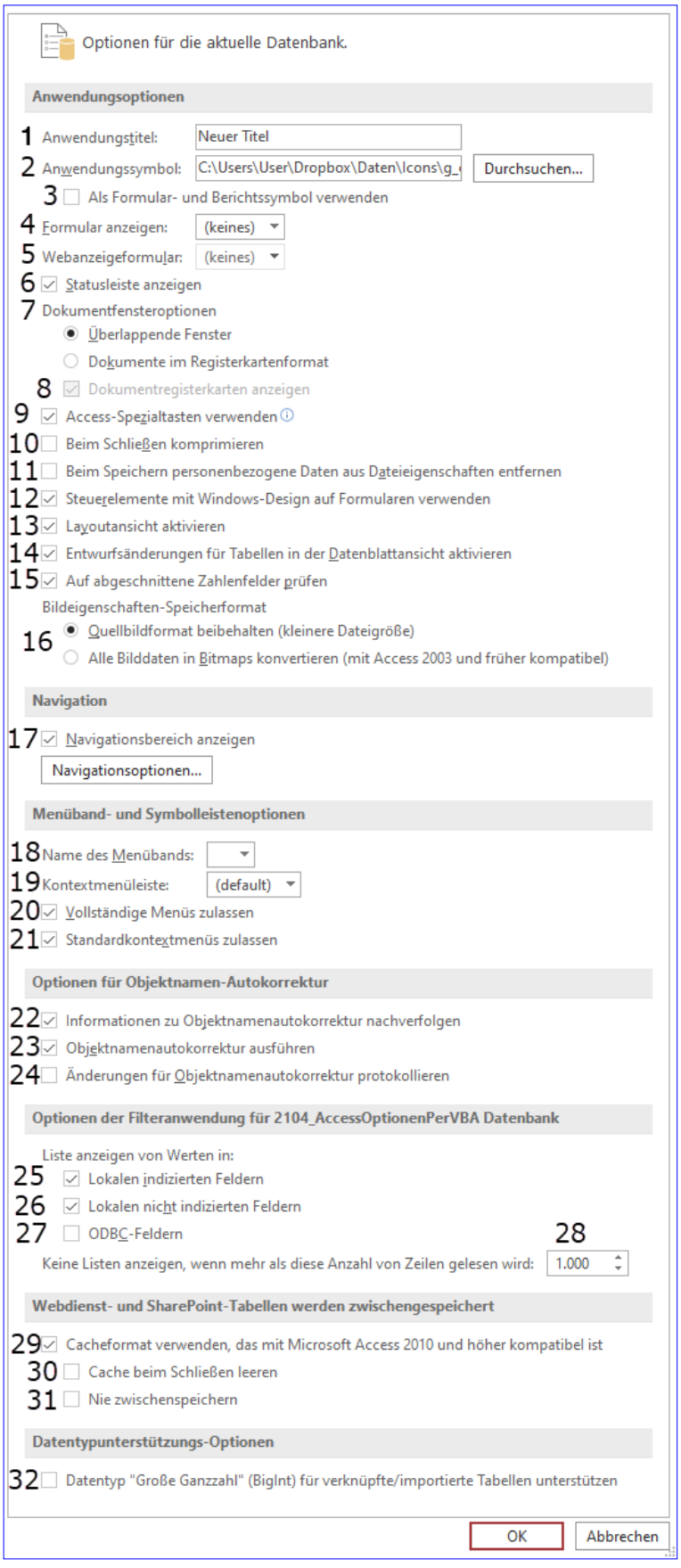


Bild 2: Optionen des Bereichs **Aktuelle Datenbank**

- **8** (**Dokumentregisterkarten anzeigen**): **ShowDocumentTabs**, Database, Boolean-Format, Neustart erforderlich
- **9** (**Access-Spezialtasten verwenden**): **AllowSpecialKeys**, Database, Boolean, Neustart erforderlich
- **10** (**Beim Schließen komprimieren**): **Auto Compact**, Database, **1**: Ja, **0**: Nein, Neustart erforderlich
- **11** (**Beim Speichern personenbezogene Daten aus Dateieigenschaften entfernen**): **Remove Personal Information**, Database, **1**: Ja, **0**: Nein, wird bei erster Änderung erstellt, Neustart erforderlich
- **12** (**Steuerelemente mit Windows-Design auf Formularen verwenden**): **Themed Form Controls**, Database, **1**: **Ja**, **2**: **Nein**
- **13** (**Layoutansicht aktivieren**): **DesignWithData**, Database, Boolean
- **14** (**Entwurfsänderungen für Tabellen in der Datenblattansicht aktivieren**): **AllowDatasheetSchema**, Database, Boolean
- **15** (**Auf abgeschnittene Zahlenfelder prüfen**): **CheckTruncatedNumFields**, Database, Boolean
- **16** (**Bildeigenschaften-Speicherformat**): **Picture Property Storage Format**, Database, **0**: **Quellbildformat beibehalten (kleinere Dateigröße)**, **1**: **Alle Bilddaten in Bitmaps konvertieren (mit Access 2003 und früher kompatibel)**
- **17** (**Navigationsbereich anzeigen**): **StartUpShowDBWindow**, Database, Boolean
- **18** (**Name des Menübands**): **CustomRibbonID**, Database, Name des Ribbons als String, wird bei erster Änderung erstellt
- **19** (**Kontextmenüleiste**): **StartupShortcutMenuBar**, Database, Name der Kontextmenüleiste, wird bei erster Änderung erstellt
- **20** (**Vollständige Menüs zulassen**): **AllowFullMenus**, Database, Boolean-Wert
- **21** (**Standardkontextmenüs zulassen**): **AllowShortcutMenus**, Database, Boolean-Wert
- **22** (**Informationen zu Objektnamenautokorrektur nachverfolgen**): **Track Name AutoCorrect Info**, Database, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung erstellt
- **23** (**Objektnamenautokorrektur ausführen**): **Perform Name AutoCorrect**, Database, **0**: **Nein**, **1**: **Ja**
- **24** (**Änderungen für Objektnamenautokorrektur protokollieren**): **Log Name AutoCorrect Changes**, Database, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung erstellt
- **25** (**Liste anzeigen von Werten in ... Lokalen indizierten Feldern**): **Show Values in Indexed**, Database, **0**: **Nein**, **1**: **Ja**
- **26** (**Liste anzeigen von Werten in ... Lokalen nicht indizierten Feldern**): **Show Values in Non-Indexed**, Database, **0**: **Nein**, **1**: **Ja**
- **27** (**Liste anzeigen von Werten in ... ODBC-Feldern**): **Show Values in Remote**, Database , **0**: **Nein**, **1**: **Ja**
- **28** (**Keine Listen anzeigen, wenn mehr als diese Anzahl von Zeilen gelesen wird**): **Show Values Limit**, Database, Zahl
- **29** (**Cacheformat verwenden, das mit Microsoft Access 2010 und höher kompatibel ist**): **Use Microsoft 2007 compatible cache**, Database, **0**: **Nein**, **1**: **Ja**

- **30** (**Cache beim Schließen leeren**): **Clear Cache on Close**, Database, **0**: **Nein**, **1**: **Ja**
- **31** (**Nie zwischenspeichern**): **Never Cache**, Database, **0**: **Nein**, **1**: **Ja**
- **32** (**Datentyp "Große Ganzzahl" (BigInt) für verknüpfte/importierte Tabellen unterstützen**): **Use BigInt for linking and importing data**, Database,, **0**: **Nein**, **1**: **Ja**

Optionen des Dialogs Navigationsoptionen

Diesen Dialog öffnen Sie entweder über die Schaltfläche **Navigationsoptionen...** im Bereich **Aktuelle Datenbank** der Access-Optionen oder über das Kontextmenü des Titels des Navigationsbereichs (siehe Bild 3).

Hier finden Sie die folgenden Optionen:

- **1** (**Ausblendete Objekte anzeigen**): **Show Hidden Objects**: Registry, **0**: **Nein**, **1**: **Ja**
- **2** (**Systemobjekte anzeigen**): **Show System Objects**: Registry, **0**: **Nein**, **1**: **Ja**

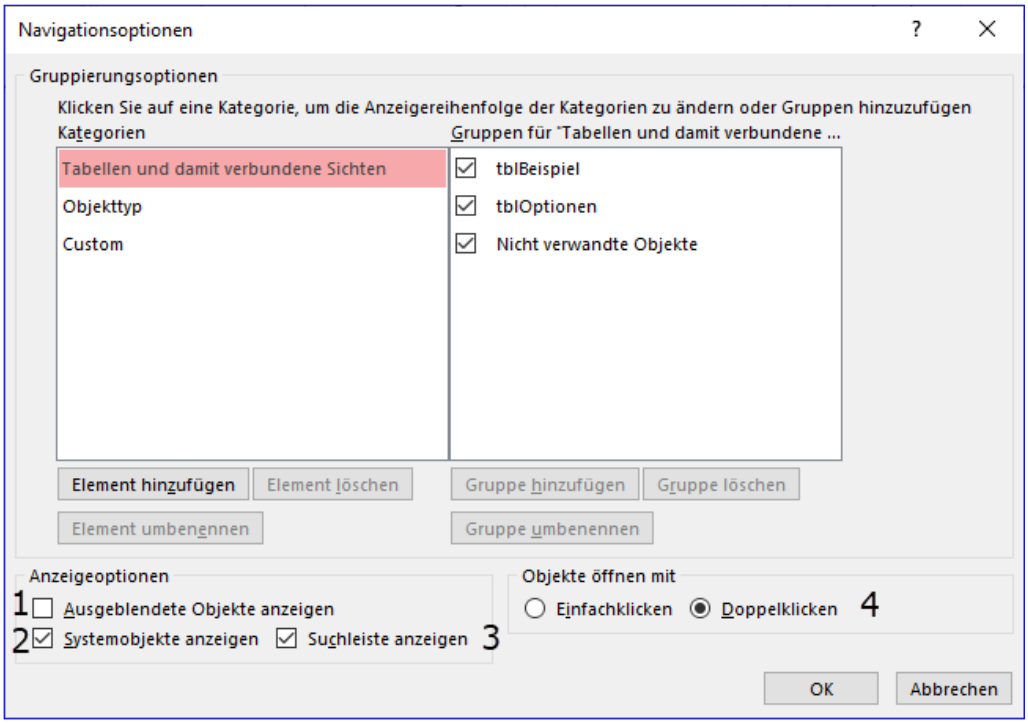


Bild 3: Optionen des Bereichs **Navigationsoptionen**

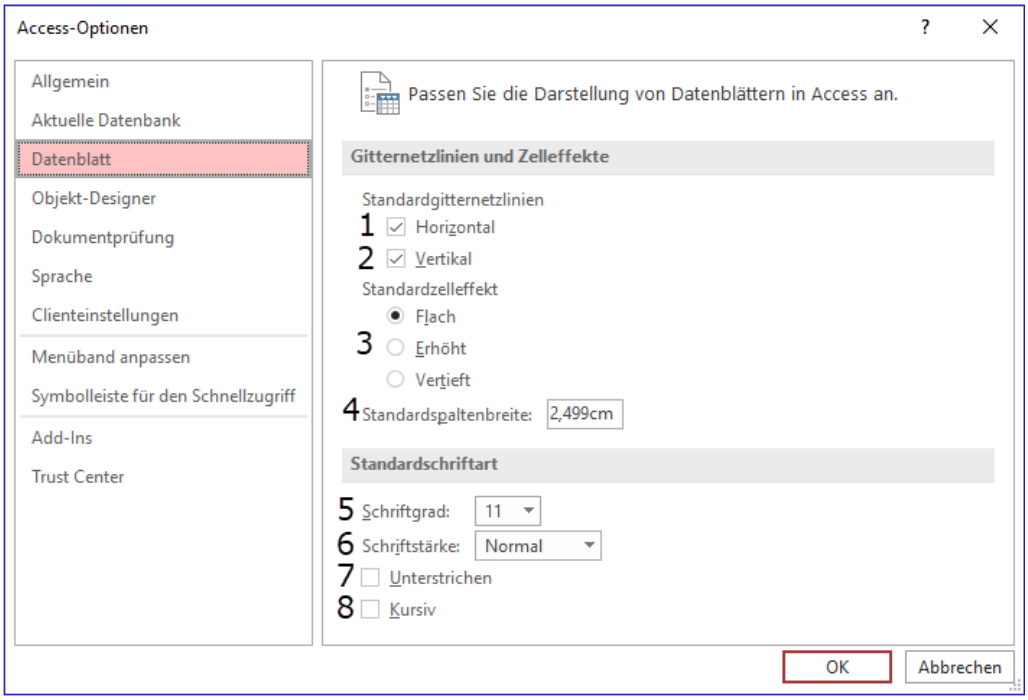


Bild 4: Optionen des Bereichs **Datenblatt**

- **3** (**Suchleiste anzeigen**): **Show Navigation Pane Search Bar**, Database, Boolean

- **4** (**Objekte öffnen mit**): **Database Explorer Click Behavior**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung erstellt

Optionen des Bereichs Datenblatt

Der Bereich **Datenblatt** hält die Optionen aus Bild 4 bereit. Diese bieten die folgenden Möglichkeiten zur Anpassung per VBA:

- **1** (**Standardgitternetzlinien – Horizontal**): **Default Gridlines Horizontal**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung erstellt

- **2** (**Standardgitternetzlinien – Vertikal**): **Default Gridlines Vertical**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Änderung erstellt

- **3** (**Standardzelleffekt**): **Default Cell Effect**, Registry, **0**: **Flach**, **1**: **Erhöht**, **2**: **Vertieft**, wird bei erster Änderung erstellt

- **4** (**Standardspaltenbreite**): **Default Column Width**, Registry, Breite in Twips, wird bei erster Änderung erstellt

- **5** (**Schriftgrad**): **Default Font Size**, Registry, Zahl, wird bei erster Änderung erstellt

- **6** (**Schriftstärke**): **Default Font Weight**, Registry, **0**: **Extra dünn**, **1**: **Sehr Dünn**, **2**: **Dünn**, 3: Normal, **4**: **Mittel**, **5**: **Halb Fett**, **6**: **Fett**, **7**: **Sehr Fett**, **8**: **Extra Fett** wird bei erster Verwendung erstellt

- **7** (**Unterstrichen**): **Default Font Underline**, Registry, **0**: **Nein**, **1**: **Ja**, wird bei erster Verwendung erstellt

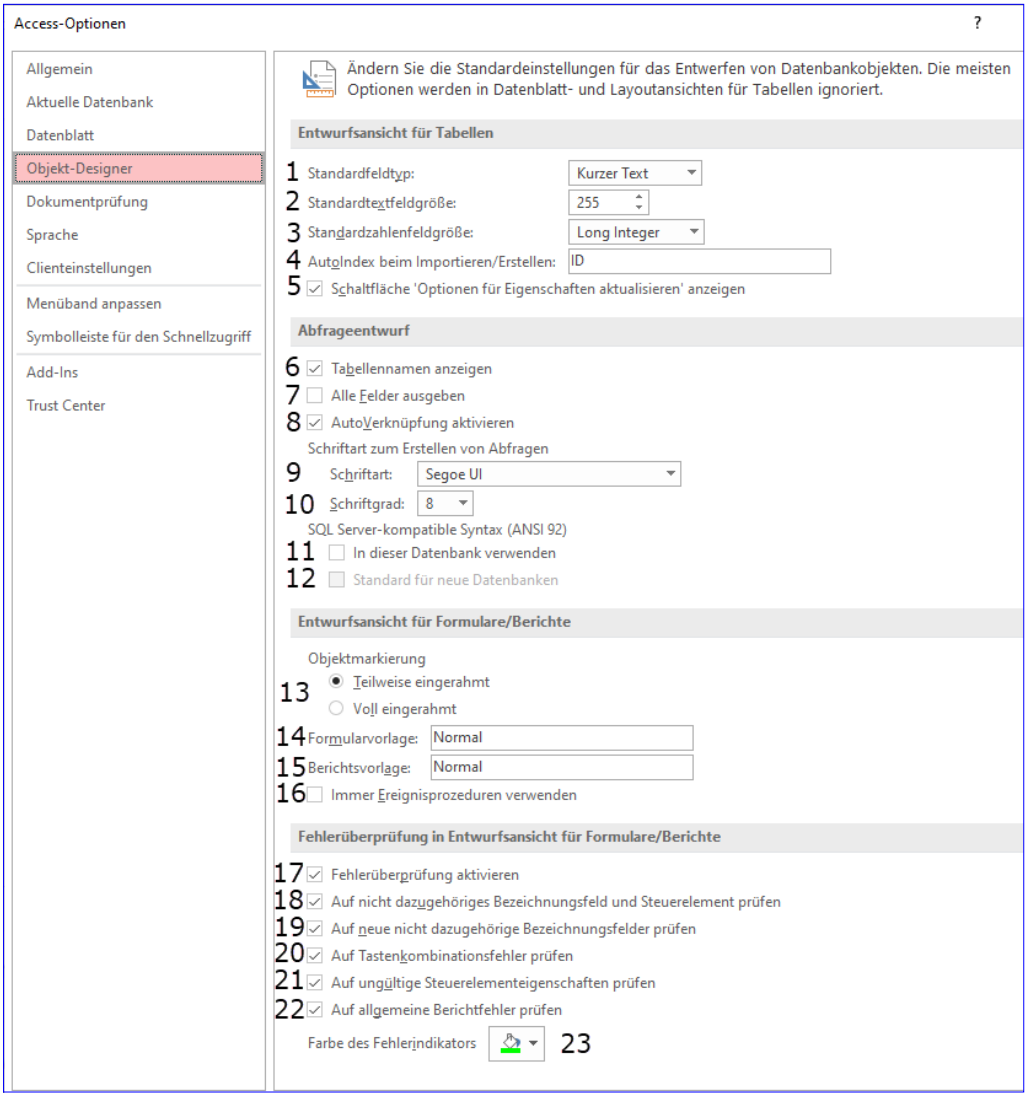


Bild 5: Optionen des Bereichs **Objekt-Designer**

API-Funktion GetSaveFileDialog (32-Bit und 64-Bit)

Das Öffnen eines Dialogs zum Auswählen des Namens einer zu speichernden Datei erledigen Sie beispielsweise mit der API-Funktion »GetSaveFileDialog«. Diese stellen wir im vorliegenden Beitrag für 32-Bit- und 64-Bit-Office vor. Dabei stellen wir auch die Änderungen heraus, die für das Update einer eventuell bereits bestehenden 32-Bit-Version auf die 64-Bit-Version notwendig sind.

Die API-Funktion **GetSaveFileDialog** ist eine Funktion der Bibliothek **comdlg32.dll**. Sie erwartet alle Parameter, die das Aussehen des Speichern-Dialogs beeinflussen, in Form eines Typs namens **OPENFILENAME**. Er heißt **OPENFILENAME**, weil er in gleicher Form auch für die API-Funktion zum Anzeigen eines **Datei öffnen**-Dialogs zum Einsatz kommt. Sie liefert den Pfad der zu speichernden Datei zurück oder eine leere Zeichenkette, falls kein Name ausgewählt wurde – zum Beispiel, weil der Benutzer den Dialog mit der **Abbrechen**-Schaltfläche geschlossen hat.

Die Wrapper-Funktion GetSaveFile

Vor dem Aufruf der API-Funktion **GetSaveFileDialog** füllt man also den Typ **OPENFILENAME** mit den gewünschten Einstellungen.

Da Sie das nicht jedes Mal erledigen sollen, wenn Sie diese Funktion zu einer Ihrer Anwendungen hinzufügen wollen, liefern wir eine Wrapperfunktion namens **GetSaveFile**. Diese sieht wie in Listing 1 aus.

```
Public Function GetSaveFile(Optional strStartDir As String, _
        Optional strDefFileName As String, _
        Optional strFilter As String = "Alle Dateien (*.*)", _
        Optional strTitle As String) As String
    Dim udtOpenFileName As OPENFILENAME
    Dim strExt As String
    On Error GoTo Fehler
    If Len(strStartDir) = 0 Then
        strStartDir = CurrentProject.Path
    End If
    With udtOpenFileName
        .nStructSize = LenB(udtOpenFileName)
        .hwndOwner = Application.hWndAccessApp
        strFilter = strFilter & vbNullChar & vbNullChar
        .sFilter = strFilter
        .nFilterIndex = 1
        .sInitDir = strStartDir & vbNullChar
        .sDlgTitle = strTitle
        .sFile = Space$(256) & vbNullChar
        .nFileSize = Len(.sFile)
        If Len(strDefFileName) <> 0 Then
            Mid(.sFile, 1) = strDefFileName
        End If
        .sFileTitle = Space$(256) & vbNullChar
        .nTitleSize = Len(.sFileTitle)
        If GetSaveFileName(udtOpenFileName) Then
            GetSaveFile = Left(udtOpenFileName.sFile, InStr(.sFile, vbNullChar) - 1)
        Else
            GetSaveFile = ""
        End If
    End With
Ende:
    Exit Function
Fehler:
    MsgBox Err.Description, vbCritical, "GetSaveFileName"
    Resume Ende
End Function
```

Listing 1: Wrapperfunktion für die API-Funktion **GetSaveFileName**

Dies sind die Parameter:

- **strStartDir**: Hier geben Sie das Verzeichnis an, das beim Öffnen des Dialogs vorausgewählt werden soll.
- **strDefFileName**: Vordefinierter Dateiname, der beim Öffnen des Dialogs im Feld **Dateiname** eingetragen werden soll.
- **strFilter**: Ein Ausdruck, der angibt, welche Dateitypen als Speichername verwendet werden dürfen.
- **strTitle**: Titel des Dialogs

Ablauf von GetSaveFile

Der Wrapperfunktion für die API-Funktion **GetSaveFileName** haben wir einen etwas verkürzten Namen gegeben – eben **GetSaveFile**. Sie definiert als Erstes eine Variable namens **udtOpenFileName** des Typs **OPENFILENAME**. Wie dieser genau aussieht und deklariert wird, zeigen wir weiter unten.

Dieser Typ arbeitet wie eine Klasse und hat verschiedene Eigenschaften, die wir mit der Funktion **GetSaveFile** füllen. **nStructSize** nimmt beispielsweise die Größe des Typs **udtOpenFileName** entgegen, die wir mit der Funktion **LenB** ermitteln. Das ist übrigens auch eine der Änderungen der 64-Bit-Version gegenüber der 32-Bit-Version – früher wurde hier die Funktion **Len** verwendet. **Len** ermittelt die Anzahl der Zeichen einer Zeichenkette, **LenB** die Anzahl der Bytes dieser Zeichenkette. **hwndOwnder** füllen wir mit dem Handle des aktuellen Fensters. Den Filter aus dem Parameter **strFilter** erweitern wir noch um zwei **vbNullChar**, bevor wir ihn der Eigenschaft **sFilter** zuweisen. Gültige Werte dazu schauen wir uns gleich im Anschluss an. Welcher der Filter voreingestellt wird, legt die Funktion für die Eigenschaft **nFilterIndex** fest, in diesem Fall mit dem Wert **1**.

Das mit dem Parameter **strStartDir** übergebene Startverzeichnis ergänzt die Funktion ebenfalls um **vbNullChar** und schreibt sie dann in die Eigenschaft **sInitDir**. Der Titel für den Dialog landet unbehandelt in **sDlgTitle**.

In **sFile** trägt die Funktion eine leere Zeichenkette mit einer Länge von 256 Zeichen und einem abschließenden **vbNullChar** ein. **nFileSize** erhält die Länge von **sFile**.

Wenn der Aufruf einen Wert für den Parameter **strDefFileName** enthält, also den voreinzustellenden Speichernamen, wird dieser vorn in der Eigenschaft **sFile** eingesetzt. Damit die Länge der darin enthaltenen Zeichenkette nicht verändert wird, verwenden wir dazu die **Mid**-Funktion in einer eher unbekannten Art. Der Aufruf **Mid(.sFile, 1) = strDefFileName** sorgt dafür, dass der Inhalt von **strDefFileName** die entsprechenden Zeichen von **.sFile** ab der ersten Position überschreibt.

sFileTitle soll den ermittelten Namen ohne Verzeichnis aufnehmen und wird mit 256 Leerzeichen plus abschließendem **vbNullChar** gefüllt. Auch die Länge diese Zeichenkette landet in **udtOpenFileName**, und zwar in der Eigenschaft **nTitleSize**.

Nachdem alle relevanten Eigenschaften von **udtOpenFileName** gefüllt sind, ruft die Funktion die API-Funktion **GetSaveFileName** mit **udtOpenFileName** als Parameter auf. Liefert diese Funktion den Wert **True** zurück, was der Fall ist, wenn der Benutzer die Schaltfläche **OK** betätigt, schreibt die Funktion den Inhalt von **sFile** aus **udtOpenFileName** in den Rückgabewert der Funktion. Anderenfalls liefert die Funktion eine leere Zeichenkette zurück.

Deklaration der API-Funktion GetSaveFileName

Wichtig für die Einsatzbereitschaft der Funktion **GetSaveFileName** unter 32-Bit- und 64-Bit-Office ist, dass Sie für beide Varianten die richtige Deklaration bereitstellen. Genau genommen ist es sogar so, dass Sie nur prüfen müssen, ob Sie VBA 6.x oder VBA 7.x verwenden. VBA 6.0 kam mit älteren Office-Versionen bis 2007. VBA 7 wurde eingeführt, um die Kompatibilität mit 64-Bit zu sichern. Grundsätzlich müssen wir also prüfen, welche Access-

32-Bit, 64-Bit, VBA-Version und Co.

Mitunter kommt es zu Missverständnissen, wenn es darum geht, die Kompatibilität von VBA-Code für verschiedene Zielversionen sicherzustellen. Dieser Beitrag erläutert in Kürze die wichtigsten Grundlagen.

Von VBA 6.x zu VBA 7.x

Mit Office 2010 wurde eine neue VBA-Version namens VBA 7.0 eingeführt (mittlerweile VBA 7.1). Diese Version stellt unter anderem die Kompatibilität mit den 64-Bit-Versionen von Office und Windows sicher.

Wir hören und lesen oft, dass Benutzer ihre Anwendung nicht mehr nutzen können, weil diese nicht mit der 64-Bit-Version von Office kompatibel ist. Das liegt daran, dass die Deklarationen von API-Funktionen und den davon verwendeten Typen nicht mit der 64-Bit-Version von Office kompatibel sind.

In vielen Beispielen findet man dann eine Unterscheidung, die durch das Prüfen einer Kompilerkonstanten erfolgt. Diese sieht so aus:

```
#If VBA7 Then
    'mit 64-Bit kompatibler Code
#Else
    'nicht mit 64-Bit kompatibler Code
#End If
```

Im ersten Teil finden wir dann schon oft besprochene Änderungen für die Kompatibilität von APIs für 64-Bit wie das Schlüsselwort **PtrSafe** oder den Datentyp **LongPtr**.

Die Unterscheidung könnte zu dem Missverständnis führen, dass **VBA7** synonym mit 64-Bit ist. Das ist aber nicht der Fall. Genau genommen macht diese Unterscheidung Folgendes: Sie hat im ersten Teil 64-Bit-kompatiblen Code, der aber nicht nur unter 64-Bit läuft, sondern auch unter 32-Bit. Wichtig ist, dass es VBA 7.x ist.

Der **Else**-Teil des Codes enthält Code, der auf jeden Fall mit VBA 6.x kompatibel ist, was sich vor allem dadurch darstellt, dass eben noch nicht das Schlüsselwort **PtrSave** oder der Datentyp **LongPtr** in APIs verwendet werden.

Sie brauchen diese umständliche Schreibweise mit der Kompilerkonstanten **VBA7** also nur, wenn Sie noch Code erzeugen, der auch in Anwendungen verwendet werden soll, die in Access 2007 und älter laufen.

Nur eine Version ab Access 2010 nötig

Wenn Sie nur noch Code für Anwendungen schreiben, die unter Access 2010 und neuer laufen, dann brauchen Sie nur den Teil des Codes aus dem ersten Teil der Bedingung **#If VBA 7 Then** abzubilden und können die Bedingung weglassen.

Denn: Die für die Nutzung mit 64-Bit eingeführten Elemente sind unter VBA 7.x auch mit der 32-Bit-Version kompatibel.

Wozu die Kompilerkonstante Win64?

Hier unterscheiden wir zwischen Office- beziehungsweise VBA-Version und der Windows-Version. Windows in der 64-Bit-Version unterstützt die von der 32-Bit-Version bekannten APIs weiter, aber es wurden auch ein paar neue Funktionen eingeführt. Wenn Sie für die 32-Bit- und die 64-Bit-Version von Windows programmieren und unter Windows 64-Bit die neuen Funktionen nutzen wollen, müssen Sie eine entsprechende Unterscheidung mithilfe der Kompilerkonstanten **Win64** treffen:

```
#If Win64 Then
    'neue, nur unter Win64 vorhandene Funktionen
#Else
    'Pendant der Funktion unter Win32
#End If
```

Registry per VBA, 32- und 64-Bit

Die Registry von Windows ist für den einen oder anderen ein Buch mit sieben Siegeln. Tatsache ist: Dort landen manche wichtigen Informationen, die Sie gegebenenfalls einmal mit VBA auslesen wollen, oder Sie wollen dafür sorgen, dass per VBA bestimmte Elemente in der Registry angelegt werden. Wir stellen einige Routinen vor, die Ihnen die Arbeit mit der Registry erleichtern. Gleichzeitig liefern wir den Code in 64-Bit-kompatibler Form.

Per VBA mit der Registry arbeiten

Wann und wo Sie auf eine Anforderung stoßen, die mit dem Lesen oder Schreiben von Registry-Werten per VBA zu tun hat, lassen wir einmal dahingestellt. Wichtig ist allein, dass Sie nach der Lektüre dieses Beitrags auf den Ernstfall vorbereitet sind!

Daher schauen wir uns in diesem Beitrag nicht nur einen Satz von Funktionen an, welche wiederum die API von Windows für den lesenden und schreibenden Zugriff auf die Registry nutzen, sondern liefern auch noch eine Reihe von Beispielen für die Anwendung dieser Funktionen.

Unser konkreter Anlass, uns mit dem Thema VBA-Zugriff auf die Registry auseinanderzusetzen, war der Beitrag **Optionen per VBA für Access 2019** (**www.access-im-unternehmen.de/1320**). Hier haben wir untersucht, welche Einstellungen des Dialogs Access-Optionen in der Registry landen. Um schnell prüfen zu können, ob sich nach dem Ändern einer Option unter Access einer der Ein-

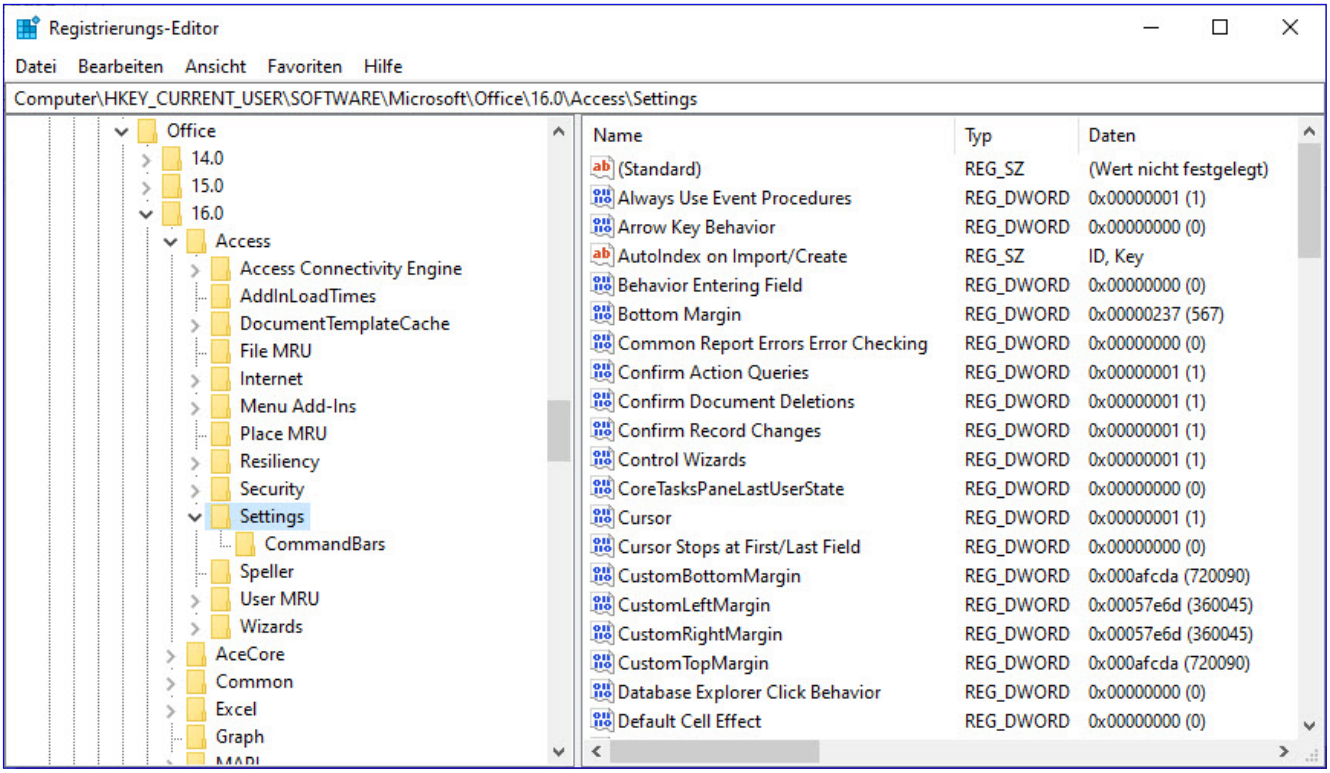


Bild 1: Dieser Bereich der Registry speichert die meisten der Access-Optionen.

träge in der Registry geändert hat, haben wir eine Tabelle erstellt, die alle bereits in der Registry vorhandenen Werte enthält. Diese Werte haben wir initial einmal aus dem entsprechenden Bereich der Registry ausgelesen.

Anschließend haben wir nach jeder Änderung eine Prozedur durchlaufen lassen, welche alle Werte der Registry mit den bestehenden Werten in der Tabelle abgeglichen und eventuelle Unterschiede oder gar neue Werte im Direktbereich ausgegeben hat.

Access-Einstellungen in der Registry

Den Bereich der Registry, der die Access-Optionen enthält, findet sich unter **HKEY_CURRENT_USER** im Ordner **SOFTWARE\Microsoft\Office\16.0\Access\Settings**. Der Screenshot aus Bild 1 zeigt einige der Einstellungen.

Auf diesen Ordner wollen wir uns in diesem Beitrag konzentrieren – wie lesen die Informationen aus, legen neue Werte an, erstellen Unterordner und so weiter. Vorher schauen wir uns allerdings noch die dazu notwendigen Funktionen im Modul **mdlRegistry** an.

Das Modul mdlRegistry und die Registry-Funktionen

Da das Modul die Deklaration von API-Funktion enthält sowie dafür vorgesehene Wrapperfunktionen, benötigen wir auch einige **Enum**- und **Const**-Elemente. Diese sehen wie folgt aus:

```
Public Enum Key
    HKEY_CLASSES_ROOT = &H80000000
    HKEY_CURRENT_USER = &H80000001
    HKEY_LOCAL_MACHINE = &H80000002
    HKEY_USERS = &H80000003
End Enum

Public Enum DataType
    dtString = 1
    dtNumber = 4
End Enum

Global Const ERROR_NONE = 0
Global Const ERROR_BADDB = 1
Global Const ERROR_BADKEY = 2
Global Const ERROR_CANTOPEN = 3
Global Const ERROR_CANTREAD = 4
Global Const ERROR_CANTWRITE = 5
Global Const ERROR_OUTOFMEMORY = 6
Global Const ERROR_INVALID_PARAMETER = 7
Global Const ERROR_ACCESS_DENIED = 8
Global Const ERROR_INVALID_PARAMETERS = 87
Global Const ERROR_NO_MORE_ITEMS = 259
Global Const KEY_ALL_ACCESS = &H3F
Global Const REG_OPTION_NON_VOLATILE = 0
```

Neben diesen Elementen benötigen wir die Deklarationen der eigentlichen API-Funktionen. Dabei handelt es sich um 13 Funktionen, deren Deklaration Sie in Listing 1 finden.

Wrapperfunktion zum Auslesen von Werten

Die erste Wrapperfunktion soll das Auslesen eines Wertes für einen Eintrag eines Schlüssels/Unterschlüssels ermöglich. Diese Funktion erwartet drei Parameter:

- **lngKey**: Einen der Werte der **Enum**-Auflistung **Key**, zum Beispiel **HKEY_CURRENT_USER**
- **strKeyName**: Den Unterschlüssel, zum Beispiel **SOFTWARE\Microsoft\Office\16.0\Access\Settings**
- **strValueName**: Den Namen des zu ermittelnden Elements, zum Beispiel **Form Template**

Die Funktion nutzt die API-Funktion **RegOpenKeyEx**, um den angegebenen Schlüssel zu öffnen. Dann liest sie mit **RegQueryValueEx** das gewünschte Element aus. Der Parameter **varValue** nimmt den gesuchten Wert auf.

Dieser wird in den folgenden Schritten noch untersucht und von nicht erwünschten Zeichen befreit. Schließlich wird dieser Wert zurückgegeben und der Registry-Schlüs-

```
Declare PtrSafe Function RegCloseKey Lib "advapi32.dll" (ByVal hKey As Long) As Long
Declare PtrSafe Function RegCreateKeyEx Lib "advapi32.dll" Alias "RegCreateKeyExA" (ByVal hKey As Long, _
    ByVal lpSubKey As String, ByVal Reserved As Long, ByVal lpClass As String, ByVal dwOptions As Long, _
    ByVal samDesired As Long, ByVal lpSecurityAttributes As Long, phkResult As Long, lpdwDisposition As Long) As Long
Declare PtrSafe Function RegOpenKeyEx Lib "advapi32.dll" Alias "RegOpenKeyExA" (ByVal hKey As Long, _
    ByVal lpSubKey As String, ByVal ulOptions As Long, ByVal samDesired As Long, phkResult As Long) As Long
Declare PtrSafe Function RegQueryValueExString Lib "advapi32.dll" Alias "RegQueryValueExA" (ByVal hKey As Long, _
    ByVal lpValueName As String, ByVal lpReserved As Long, lpType As Long, ByVal lpData As String, lpcbData As Long) As Long
Declare PtrSafe Function RegQueryValueExLong Lib "advapi32.dll" Alias "RegQueryValueExA" (ByVal hKey As Long, _
    ByVal lpValueName As String, ByVal lpReserved As Long, lpType As Long, lpData As Long, lpcbData As Long) As Long
Declare PtrSafe Function RegQueryValueExNULL Lib "advapi32.dll" Alias "RegQueryValueExA" (ByVal hKey As Long, _
    ByVal lpValueName As String, ByVal lpReserved As Long, lpType As Long, ByVal lpData As Long, lpcbData As Long) As Long
Declare PtrSafe Function RegSetValueExString Lib "advapi32.dll" Alias "RegSetValueExA" (ByVal hKey As Long, _
    ByVal lpValueName As String, ByVal Reserved As Long, ByVal dwType As Long, ByVal lpValue As String, _
    ByVal cbData As Long) As Long
Declare PtrSafe Function RegSetValueExLong Lib "advapi32.dll" Alias "RegSetValueExA" (ByVal hKey As Long, _
    ByVal lpValueName As String, ByVal Reserved As Long, ByVal dwType As Long, lpValue As Long, ByVal cbData As Long) _
    As Long
Declare PtrSafe Function RegDeleteKey& Lib "advapi32.dll" Alias "RegDeleteKeyA" (ByVal hKey As Long, _
    ByVal lpSubKey As String)
Declare PtrSafe Function RegDeleteValue& Lib "advapi32.dll" Alias "RegDeleteValueA" (ByVal hKey As Long, _
    ByVal lpValueName As String)
Declare PtrSafe Function RegOpenKey Lib "advapi32.dll" Alias "RegOpenKeyA" (ByVal hKey As Long, _
    ByVal lpSubKey As String, phkResult As Long) As Long
Declare PtrSafe Function RegEnumKeyEx Lib "advapi32.dll" Alias "RegEnumKeyExA" (ByVal hKey As Long, _
    ByVal dwIndex As Long, ByVal lpName As String, lpcbName As Long, ByVal lpReserved As Long, _
    ByVal lpClass As String, lpcbClass As Long, lpftLastWriteTime As Any) As Long
Declare PtrSafe Function RegEnumValue Lib "advapi32.dll" Alias "RegEnumValueA" (ByVal hKey As Long, _
    ByVal dwIndex As Long, ByVal lpValueName As String, lpcbValueName As Long, ByVal lpReserved As Long, _
    lpType As Long, lpData As Byte, lpcbData As Long) As Long
```

Listing 1: Die benötigten API-Funktionen

sel mit der API-Funktion **RegCloseKey** wieder geschlossen (siehe Listing 2)

Wrapperfunktion zum Auslesen aller Einträge eines Schlüssels

Die Wrapperfunktion **EnumKeyValues** ermittelt alle Einträge des mit den Parametern übergebenen Schlüssel/Unterschlüssel-Kombination.

Die Funktion erwartet diese beiden Parameter:

- **lngKey**: Einen der Werte der **Enum**-Auflistung **Key**, zum Beispiel **HKEY_CURRENT_USER**
- **strKeyName**: Den Unterschlüssel, zum Beispiel **SOFTWARE\Microsoft\Office\16.0\Access\Settings**

Die Funktion öffnet wieder mit **RegOpenKey** den Schlüssel. Dann startet sie eine **Do...Loop**-Schleife, die dann endet, wenn die Funktion **RegEnumValue** den Wert **0** zurückgibt. Diese erhält in der Schleife wiederum einen Index als zweiten Parameter, der mit jedem Schleifendurchlauf erhöht wird, sowie als dritten Parameter eine mit 255 Leerzeichen vorbelegte Zeichenkette. Diese wird, sofern der aktuelle Index aus lngIndex ein gültiges Ergebnis hergibt, mit dem Namen des Eintrags gefüllt. Der Inhalt von **strSave** wird dann mit der Funktion **StripTerminator**

```
Public Function QueryValue(lngKey As Key, strKeyName As String, strValueName As String)
    Dim lngRetVal As Long
    Dim hKey As Long
    Dim varValue As Variant
    lngRetVal = RegOpenKeyEx(lngKey, strKeyName, 0, KEY_ALL_ACCESS, hKey)
    lngRetVal = QueryValueEx(hKey, strValueName, varValue)
    varValue = Trim(varValue)
    If varValue <> "" Then
        If Asc(Right(varValue, 1)) > 127 Then
            varValue = Left(varValue, Len(varValue) - 1)
        End If
    End If
    If varValue <> "" Then
        If Asc(Right(varValue, 1)) < 21 Then
            varValue = Left(varValue, Len(varValue) - 1)
        End If
    End If
    QueryValue = varValue
    RegCloseKey hKey
End Function
```

Listing 2: Die benötigten API-Funktionen

von **vbNullchar**-Zeichen befreit und an das Array **strKeys** angehängt. Dieses gibt die Funktion schließlich als Ergebnis zurück:

```
Public Function EnumKeyValues(lngKey As Key, _
        strSubkey As String) As String()
    Dim hKey As Long
    Dim lngIndex As Long
    Dim strSave As String
    Dim strKeys() As String
    RegOpenKey lngKey, strSubkey, hKey
    Do
        strSave = String(255, 0)
        If RegEnumValue(hKey, lngIndex, strSave, 255, _
                0, ByVal 0&, ByVal 0&, ByVal 0&) <> 0 Then
            Exit Do
        Else
            ReDim Preserve strKeys(lngIndex)
            strKeys(lngIndex) = StripTerminator(strSave)
            lngIndex = lngIndex + 1
        End If
    Loop
    RegCloseKey hKey
    EnumKeyValues = strKeys
End Function
```

Die Hilfsfunktion StripTerminator

Die in der vorher vorgestellten Funktion verwendete Hilfsfunktion **StripTerminator** befreit die übergebene Zeichenkette von Vorkommen des Zeichens **vbNullChar**. Die API-Funktion **RegEnumValue** ersetzt in der **String**-Variablen **strSave** die 255 Leerzeichen durch das Ergebnis und füllt die übrigen Zeichen mit dem Zeichen **vbNullChar** auf. Diese wollen wir entfernen und verwenden dazu die Funktion **StripTerminator**. Die Funktion erwartet die zu untersuchende Zeichenkette als Parameter und gibt die überarbeitete Zeichenkette zurück. Sie sucht im ersten Schritt nach dem ersten Vorkommen von **vbNullChar** und geht davon aus, dass hier das eigentliche Ergebnis beendet ist. Die Suche erledigt sie mit der Funktion **InStr**, welche die Position des ersten Vorkommens zurückliefert. Ist das Ergebnis größer als **0**, wurde ein **vbNullChar**-

Ribbon: Callback-Signaturen für VBA und VB6

Wer das Ribbon um benutzerdefinierte Erweiterungen ergänzen möchte, kommt früher oder später nicht um die Programmierung von Callbackfunktionen herum. Das sind Funktionen, die für Callback-Attribute von Ribbon-Elementen angegeben werden und die für verschiedene Aktionen aufgerufen werden – beispielsweise beim Anklicken einer Schaltfläche, beim Ändern des Inhalts eines Textfeldes oder schlicht, um vor dem Anzeigen dynamisch Einstellungen für Attribute des Ribbons einzulesen. Diese Callbackfunktionen haben eine bestimmte Signatur (sprich Definition der ersten Zeile). Da diese für den Einsatz von VBA und VB6 unterschiedlich aussehen und aktuell twinBASIC als Ersatz für VB6 heranreift, wollen wir eine Referenz der Callback-Signaturen für diese beiden Programmiersprachen anbieten.

Warum verschiedene Callback-Signaturen?

Als ich neulich angefangen habe, mit der neuen Programmiersprache twinBASIC COM-Add-Ins für die Benutzeroberfläche von Access zu programmieren, wollte ich natürlich auch Callbackfunktionen einsetzen.

Also habe ich einfach die gleichen Callbackfunktionen verwendet, die ich sonst unter Access und VBA einsetze. Der Aufruf lieferte allerdings immer wieder den gleichen Fehler – siehe Bild 1.

Ich bin fast verzweifelt, bis ich mich dann daran erinnerte, schon einmal mit dem alten Visual Studio unter VB6 ein COM-Add-In mit Ribbon programmiert zu haben – und dass dort die Signatur der Callbackfunktionen anders ausgesehen hat.

Unter VB6 sind Callbackfunktionen nämlich tatsächlich Funktionen – und nicht wie unter VBA Callbackprozeduren, die einen weiteren Parameter enthalten, der als Rückgabewert verwendet wird.

Schauen wir uns also die unterschiedlichen Signaturen für die einzelnen Steuerelemente des Ribbons für VBA und VB6 an!

Wichtige Informationen

Die wichtigste Information: Verwenden Sie alle Callback-Signaturen genau so, wie Sie hier abgebildet sind. Manchmal kann schon das Hinzufügen eines Datentyps zu einem Parameter, der hier keinen Datentyp aufweist, zu Problemen führen. Wichtig ist auch, auf die korrekte Verwendung von **Function/Sub** zu achten. Wenn Sie twinBASIC verwenden, können Sie das Ergebnis zum Zurückgeben entweder einer Variablen mit dem gleichen Namen wie die Funktion zuweisen, aber auch die **Return**-Anweisung nutzen:

```
Function GetText(control As IRibbonControl, ByRef text) _
                                                As String
    ...
    Return strText
End Function
```

Bei den folgenden Definitionen stellen wir zunächst die je nach Steuerelement individuellen Callbackfunktionen vor. Am Ende finden Sie eine Auflistung all jener Callbackfunk-

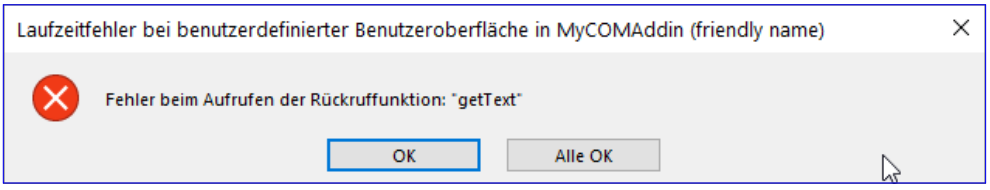


Bild 1: Fehler durch falsche Callback-Signatur

tionen, die von einer großen Anzahl an Steuerelementen verwendet wird.

Die Callbackfunktionen des button-Elements

Das **button**-Element bietet die folgenden Callbackfunktionen:

getShowImage: Fragt ab, ob ein Image angezeigt werden soll. Das geschieht unter VBA mit dem Rückgabe-Parameter **showImage**, unter VB6/twinBASIC mit dem Rückgabewert.

```
VBA: Sub GetShowImage (control As IRibbonControl, ByRef showImage)
VB6/twinBASIC: Function GetShowImage (control As IRibbonControl) As Boolean
```

getShowLabel: Fragt ab, ob eine Bezeichnung angezeigt werden soll. Der Wert **True** oder **False** wird mit dem Parameter **showLabel** oder dem Rückgabewert übergeben.

```
VBA: Sub GetShowLabel (control As IRibbonControl, ByRef showLabel)
VB6/twinBASIC: Function GetShowLabel (control As IRibbonControl) As Boolean
```

onAction: Wird beim Anklicken ausgelöst. Diese Signatur gilt für benutzerdefinierte **button**-Elemente.

```
VBA: Sub OnAction(control As IRibbonControl)
VB6/twinBASIC: Sub OnAction(control As IRibbonControl)
```

onAction, Alternative: Wird beim Anklicken eines eingebauten Elements ausgelöst, für das Sie ein **command**-Element definiert haben.

Wie Sie diese Variante von **onAction** einsetzen, lernen Sie im Beitrag **Funktion von eingebauten Ribbon-Steuerelementen überschreiben** (**www.access-im-unternehmen.de/1328**).

```
VBA: Sub OnAction(control As IRibbonControl, byRef CancelDefault)
VB6/twinBASIC: Sub OnAction(control As IRibbonControl, byRef CancelDefault)
```

Die Callbackfunktionen des checkBox-Elements

Das **checkBox**-Element bietet die folgenden Callbackfunktionen:

getPressed: Stellt den Zustand des **checkBox**-Steuerelements ein, also auf **True** oder **False**. Der Rückgabewert heißt unter VBA **returnValue**.

```
VBA: Sub GetPressed(control As IRibbonControl, ByRef returnValue)
VB6/twinBASIC: Function GetPressed(control As IRibbonControl) As Boolean
```

onAction: Wird ausgelöst, wenn der Benutzer auf ein **checkBox**-Steuerelement klickt und seinen Wert damit ändert. Liefert den neuen Wert mit dem **Boolean**-Parameter **pressed**.

```
VBA: Sub OnAction(control As IRibbonControl, pressed As Boolean)
VB6/twinBASIC: Sub OnAction(control As IRibbonControl, pressed As Boolean)
```

Die Callbackfunktionen des comboBox-Elements

Das **comboBox**-Element bietet die folgenden Callbackfunktionen:

getItemCount: Callbackfunktion zum Abfragen der Anzahl der im **comboBox**-Steuerelement anzuzeigenden Einträge. Zu liefern mit dem Parameter **count** oder dem Rückgabewert der Callbackfunktion.

```
VBA: Sub GetItemCount(control As IRibbonControl, ByRef count)
VB6/twinBASIC: Function GetItemCount(control As IRibbonControl) As Integer
```

getItemID: Ruft den Wert für die **ItemID** eines der Einträge des **comboBox**-Steuerelements ab, dessen Index unter VBA mit dem Parameter **index** abgefragt wird – oder mit dem Rückgabewert unter Visual Basic 6/ twinBASIC.

```
VBA: Sub GetItemID(control As IRibbonControl, index As Integer, ByRef id)
VB6/twinBASIC: Function GetItemID(control As IRibbonControl, index As Integer) As String
```

getItemImage: Ruft den Namen des Bildes eines der Einträge des **comboBox**-Steuerelements ab, dessen Index unter VBA mit dem Parameter **index** oder unter VB6/ twinBASIC mit dem Rückgabewert der Callbackfunktion abgefragt wird.

```
VBA: Sub GetItemImage(control As IRibbonControl, index As Integer, ByRef image)
VB6/twinBASIC: Function GetItemImage(control As IRibbonControl, index As Integer) As IPictureDisp
```

getItemLabel: Ruft die Beschriftung eines der Einträge des **comboBox**-Steuerelements ab, dessen Index mit dem Parameter **index** oder dem Rückgabewert übergeben wird.

```
VBA: Sub GetItemLabel(control As IRibbonControl, index As Integer, ByRef label)
VB6/twinBASIC: Function GetItemLabel(control As IRibbonControl, index As Integer) As String
```

getItemScreenTip: Ruft den **screenTip** eines der Einträge des **comboBox**-Steuerelements ab, dessen Index mit dem Parameter **index** oder dem Rückgabewert übergeben wird.

```
VBA: Sub GetItemScreenTip(control As IRibbonControl, index As Integer, ByRef screentip)
VB6/twinBASIC: Function GetItemScreentip(control As IRibbonControl, index As Integer) As String
```

getItemSuperTip: Ruft den **superTip** eines der Einträge des **comboBox**-Steuerelements ab, dessen Index mit dem Parameter **index** oder dem Rückgabewert übergeben wird.

```
VBA: Sub GetItemSuperTip (control As IRibbonControl, index As Integer, ByRef supertip)
VB6/twinBASIC: Function GetItemSuperTip (control As IRibbonControl, index As Integer) As String
```

getText: Ruft den aktuell im **comboBox**-Element anzuzeigenden Text ab. Dies geschieht über den Parameter **text** oder dem Rückgabewert.

```
VBA: Sub GetText(control As IRibbonControl, ByRef text)
VB6/twinBASIC: Function GetText(control As IRibbonControl) As String
```

onChange: Wird ausgelöst, wenn der Benutzer einen anderen Eintrag im **comboBox**-Steuerelement auswählt. Liefert den neuen Wert mit dem Parameter **text**.

```
VBA: Sub OnChange(control As IRibbonControl, text As String)
VB6/twinBASIC: Sub OnChange(control As IRibbonControl, text As String)
```

Die Callbackfunktionen des customUI-Elements

Das **customUI**-Element bietet die folgenden Callbackfunktionen:

loadImage: Wird beim Anzeigen eines Ribbons für jedes Steuerelement im Ribbon einmal ausgelöst, für das im Attribut **image** ein Bildname hinterlegt ist.

```
VBA: Sub LoadImage(imageId As string, ByRef image)
VB6/twinBASIC: Function LoadImage(imageId As String) As IPictureDisp
```

onLoad: Wird beim erstmaligen Laden des Ribbons ausgelöst und erlaubt das Referenzieren der mit **ribbon** gelieferten Instanz der Ribbon-Definition.

```
VBA: Sub OnLoad(ribbon As IRibbonUI)
VB6/twinBASIC: Function OnLoad(ribbon As IRibbonUI)
```

Die Callbackfunktionen des dropDown-Elements

Das **dropDown**-Element bietet die folgenden Callbackfunktionen:

getItemCount: Callbackfunktion zum Abfragen der Anzahl der im **dropDown**-Steuerelement anzuzeigenden Einträge, zu liefern mit dem Parameter **count** oder dem Rückgabewert als Integer.

```
VBA: Sub GetItemCount(control As IRibbonControl, ByRef
count)
VB6/twinBASIC: Function GetItemCount(control As IRibbon-
Control) As Integer
```

getItemID: Ruft den Wert für die **ItemID** eines der Einträge des **dropDown**-Steuerelements ab, dessen Index mit dem Parameter **index** übergeben wird. Die **ItemID** wird mit dem Parameter **id** oder dem Rückgabewert im Format **String** erwartet.

```
VBA: Sub GetItemID(control As IRibbonControl, index As
Integer, ByRef id)
VB6/twinBASIC: Function GetItemID(control As IRibbonCon-
trol, index As Integer) As String
```

getItemImage: Ruft den Namen des Bildes eines der Einträge des **dropBox**-Steuerelements ab, dessen Index mit dem Parameter **index** geliefert wird. Der Bildname wird mit dem Parameter image oder mit dem Rückgabewert des Datentyps **IPictureDisp** übergeben.

```
VBA: Sub GetItemImage(control As IRibbonControl, index As
Integer, ByRef image)
VB6/twinBASIC: Function GetItemImage(control As IRibbon-
Control, index As Integer) As IPictureDisp
```

getItemLabel: Ruft die Beschriftung eines der Einträge des **dropDown**-Steuerelements ab, dessen Index mit dem Parameter **index** übergeben wird und die Beschriftung mit dem Parameter **label** beziehungsweise mit dem Rückgabewert als **String** erwartet.

```
VBA: Sub GetItemLabel(control As IRibbonControl, index As
Integer, ByRef label)
VB6/twinBASIC: Function GetItemLabel(control As IRibbon-
Control, index As Integer) As String
```

getItemScreenTip: Ruft den **screenTip** eines der Einträge des **getItemScreenTip**-Steuerelements ab, dessen Index mit dem Parameter **index** geliefert und dessen Wert mit dem Parameter **screenTip** oder dem Rückgabewert übergeben wird.

```
VBA: Sub GetItemScreenTip(control As IRibbonControl, index
As Integer, ByRef screenTip)
VB6/twinBASIC: Function GetItemScreentip(control As IRib-
bonControl, index As Integer) As String
```

getItemSuperTip: Ruft den **superTip** eines der Einträge des **dropDown**-Steuerelements ab, dessen Index mit dem Parameter **index** übergeben wird und dessen Wert mit dem Parameter **superTip** oder mit dem **String**-Rückgabewert zurückgegeben wird.

```
VBA: Sub GetItemSuperTip (control As IRibbonControl, index
As Integer, ByRef superTip)
VB6/twinBASIC: Function GetItemSuperTip (control As IRib-
bonControl, index As Integer) As String
```

getSelectedItemID: Fragt den Wert des Attributs **itemID** für das aktuell zu selektierende Element für ein **dropDown**-Steuerelement ab – entweder über den Parameter **index** oder den Rückgabewert der Callbackfunktion im **Integer**-Format.

```
VBA: Sub GetSelectedItemID(control As IRibbonControl,
ByRef index)
VB6/twinBASIC: Function GetSelectedItemID(control As IRib-
bonControl) As Integer
```

Benutzerdefinierte Bilder in twinBASIC

Die Programmiersprache/Entwicklungsumgebung twinBASIC entwickelt sich aktuell stetig weiter. Der Entwickler Wayne Philips fügt ständig neue Elemente hinzu. In den ersten Beiträgen mussten wir das Ribbon eines COM-Add-Ins noch mit den eingebauten Icons ausstatten, da es nicht möglich war, Bilddateien als Resourcen in die DLL zu integrieren. Das hat sich nun geändert: Sie können Bilddateien zum Projekt hinzufügen und diese sehr einfach im Ribbon nutzen. Dieser Beitrag zeigt, wie das gelingt.

Voraussetzungen

Wie Sie ein COM-Add-In erstellen, erfahren Sie beispielsweise in den beiden Beiträgen **twinBASIC – COM-Add-Ins für Access** (**www.access-im-unternehmen.de/1306**) und **Access-Optionen per Ribbon ändern** (**www.access-im-unternehmen.de/1327**).

Icons nutzen

Icons verwenden wir bei COM-Add-Ins aktuell für Schaltflächen und anderen Elemente im Ribbon. Um Icons zu nutzen, sind zwei Schritte nötig:

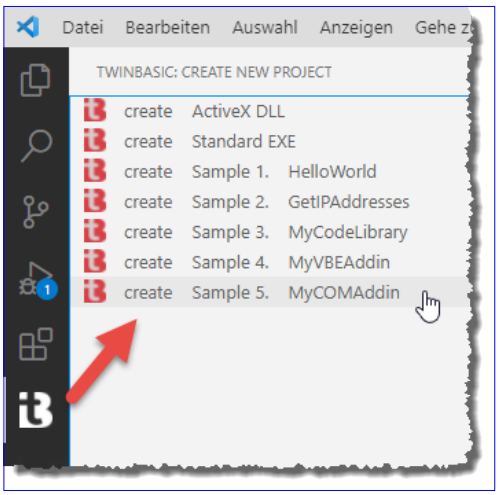


Bild 1: Erstellen eines COM-Add-In-Projekts

- Hinzufügen zum Projekt
- Laden, wenn das Ribbon angezeigt wird.

Beides schauen wir uns nun an.

Icons zum Projekt hinzufügen

Das Beispielprojekt für ein COM-Add-In, dass Sie mit dem Befehl aus Bild 1 erstellen, enthält bereits zwei Ordner, die Sie für das Hinzufügen von Bildern benötigt werden.

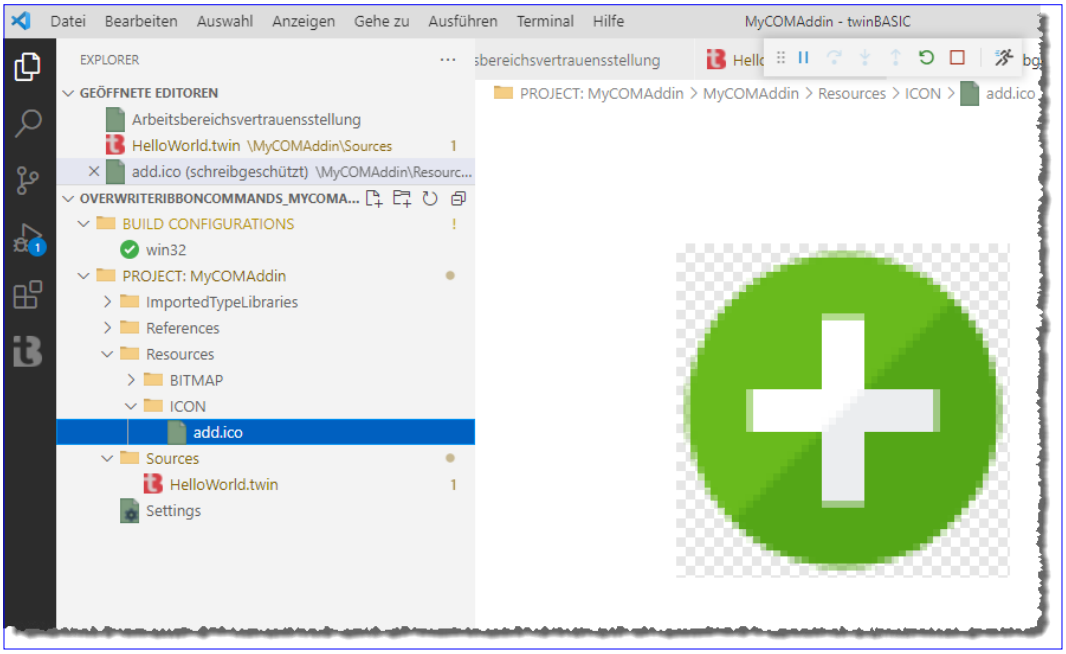


Bild 2: Hinzufügen eines Icons zum Projekt

Ziehen Sie **.ico**-Dateien in den **ICON**-Ordner, sodass diese dort wie in Bild 2 erscheinen. Achtung: Wenn Drag and Drop nicht funktioniert, kann es sein, dass

```
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"">" & vbCrLf
        strXML &= "  <ribbon startFromScratch=""false"">" & vbCrLf
        strXML &= "    <tabs>" & vbCrLf
        strXML &= "      <tab id=""tab"" label=""Tab mit Bild"">" & vbCrLf
        strXML &= "        <group id=""grp"" label=""Gruppe mit Bild"">" & vbCrLf
        strXML &= "          <button id=""btnIcon"" size=""large"" label=""Icon"" getIimage=""getImage""" _
            & " onAction=""onAction""/>" & vbCrLf
        strXML &= "        </group>" & vbCrLf
        strXML &= "      </tab>" & vbCrLf
        strXML &= "    </tabs>" & vbCrLf
        strXML &= "  </ribbon>" & vbCrLf
        strXML &= "</customUI>" & vbCrLf
        Return strXML
End Fun ction
```

Listing 1: Icon zu einer Schaltfläche hinzufügen per **getImage**

Sie Visual Studio Code als Administrator gestartet haben. Wegen der höheren Rechte können Sie dann keine Dateien mehr aus dem normal geöffneten Explorer hineinziehen.

Ribbon-Anpassung

In der Ribbon-Definition haben Sie zwei Möglichkeiten, das Icon einzulesen. Die erste ist die Callback-Funktion **getImage**, der Sie einfach den Namen der aufzurufenden Callback-Funktion übergeben (siehe Listing 1).

Diese prüft den Namen des aufrufenden Steuerelements und weist dem Rückgabewert das mit der Funktion **LoadResPicture** ermittelte Bild zu. Zu beachten ist hier, dass Sie die Größe mit den letzten beiden Parametern angeben – hier 32x32 (für Icons in Schaltflächen mit dem Wert **normal** für den Parameter **size** verwenden Sie 16x16):

```
Public Function getImage(Control As IRibbonControl) 7
                                                As IPictureDisp
    Dim objPicture As Object
    Select Case Control.Id
        Case "btnICON"
            Set objPicture = LoadResPicture("add.ico", 7
                                vbResBitmapFromIcon, 32, 32)
    End Select
    Return objPicture
End Function
```

Bild 3: Ribbon mit Icon

Das Ergebnis sehen sie in Bild 3.

Wenn viele Steuerelemente mit Icons ausgestattet werden sollen, verwenden Sie das Attribut **LoadImages** des Elements **customUI** und geben das Image für das entsprechende

Setup für Access: Umsetzung mit InnoSetup

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

Im ersten Teil haben wir uns mit den Grundlagen eines Setups beschäftigt, nun geht es »in medias res«. Dieser Teil beinhaltet die konkrete Umsetzung der Gedanken. Wir schauen uns das Setup-Script im Detail an und lernen, was die einzelnen Bestandteile bedeuten. Wir werden neben der Access-Datenbank auch Startmenü-Einträge und Desktop-Icons anlegen. Die Setup-Sprache wird variabel gemacht, es darf eine Lizenzvereinbarung geben, und es wird für eine ordnungsgemäße Deinstallation gesorgt.

Organisatorisches

Vor der Umsetzung sollten wir uns kurz überlegen, was wir erreichen wollen, denn »wer nicht weiß, wo er hin will, darf sich nicht wundern, wenn er ganz woanders ankommt«.

Gehen wir zunächst von einem Minimalsetup aus. Was wäre das Minimum, das ein Access-Entwickler von »seinem« Setup erwartet?

- Einspielen der **.accdb**/**.accde**-Datei
- Einrichten von Startmenü- und Desktop-Icons
- Vorbereitung der Deinstallation

Als kleine Erweiterung behalten wir uns dann noch folgendes vor:

- Zu Beginn Darstellung von Betahinweis oder der Lizenzbedingungen, Bestätigung anfordern
- Zu Beginn Darstellung im Stile “Was ist neu in diesem Release?”
- Abschließend eine Readme-Datei anzeigen

Das ist zunächst sicher nicht viel, und wer den ersten Teil dieser kleinen Artikelserie gelesen hat, hätte sicher noch einiges mehr erwartet. Aber machen wir es uns zu Beginn nicht zu schwer, Erweiterungen sind ja jederzeit möglich.

Ich erlaube mir, an dieser Stelle noch ein anderes hilfreiches Werkzeug des Softwareentwicklers zu erwähnen. Wahrscheinlich werden Sie dies ohnehin nutzen, dann ist es nur eine Erinnerung, dass das auch für unser Setup-Script funktioniert.

Es geht um Versionsverwaltung, auch Quellcodeverwaltung genannt. Unser Setup-Script ist im »Quellcode« schließlich auch nur ein wenig »plain text«, sodass dieses Script genauso mit Git (oder oder einer anderen Quellcodeverwaltung) verwaltet werden kann – und sollte.

Die **.exe**-Datei würde ich dabei von der Verwaltung ausnehmen (Git: ***.exe** in die **.gitignore**-Datei). Einerseits wird sie recht groß sein, andererseits ist sie redundant, denn alles, was zu ihrer Erstellung benötigt wird, ist ja ohnehin komplett versioniert.

Grundprinzip des Setup-Scripts

Das Setup-Script ähnelt in weiten Bereichen streng genommen mehr einer **.ini**-Datei als einem Programm. Ich will es dennoch weiterhin »Script« nennen, da der Begriff durchaus als gängig gelten kann. Die Festlegungen in diesem Script werden vom InnoSetup-Compiler inter-

pretiert, der letztlich dann das eigentliche Setup (eine **.exe**-Datei) zusammenbaut. Ausgeliefert wird dann nur diese **.exe**-Datei.

Wie bei einer **.ini**-Datei üblich haben wir Sektionen mit Überschriften in eckigen Klammern wie zum Beispiel **[Setup]** gleich zu Beginn. Innerhalb dieser Sektionen stehen dann zahlreiche Key-Value-Einträge im Format **Key = Value**.

Beginnt eine Zeile mit einem Semikolon, wird diese komplett als Kommentarzeile interpretiert, also nicht für die Setup-Erstellung berücksichtigt. Dadurch können wir bestimmte Einträge vorbereiten und bei Bedarf aktivieren oder deaktivieren. Nur ein erneuter Compilerlauf ist dann erforderlich.

Wer ein neues Setup erstellen will, kann dies sowohl über InnoSetup selbst als auch über eine der im ersten Teil bereits angesprochenen GUIs tun. Dort ist in der Regel ein Assistent enthalten, mit dessen Hilfe das Grundgerüst schnell eingerichtet werden kann.

Da die Bedienung solcher Assistenten zumeist sehr intuitiv ist, will ich diese hier nicht näher besprechen, sondern lieber auf die Eintragungen im Script selbst eingehen. Ob Sie das Script mittels Assistenten oder von Hand erstellen oder einfach das am Ende dieses Artikels bereitgestellte fertige Script verwenden, bleibt dabei Ihnen überlassen.

Zu Beginn definieren wir in der Sektion **[Setup]** den Namen des zu installierenden Programms und einige weitere organisatorische Dinge:

```
[Setup]
AppId=TestSetup
AppName=My Awesome AccessApp
AppVersion=1.0.0
AppVerName=AccessApp v1.0.0
AppPublisher=Vorname Name
AppPublisherURL=https://www.meine-homepage.de
AppSupportURL=https://www.meine-homepage.de
AppUpdatesURL=https://www.meine-homepage.de
```

Dadurch werden Name und Version der zu installierenden Software genannt, ebenso Name und Website des Herausgebers und zwei URLs für Support und Updates.

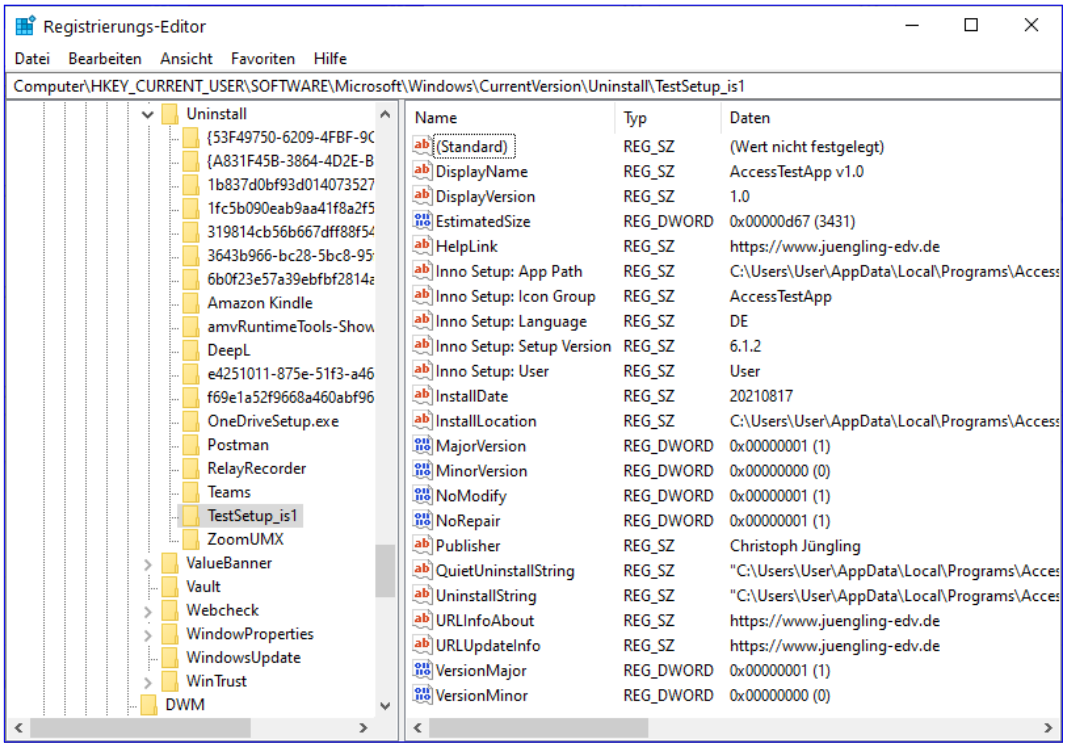


Bild 1: Setup-Eintrag in der Registry

Die **AppId** hat dabei eine Sonderstellung, denn diese Information wird in der Registry als Bezeichnung für den **Uninstall**-Bereich eingetragen (siehe Bild 1). Hier finden Sie detaillierte Informationen über die Deinstallation.

Anhand dieser Bezeichnung erkennt InnoSetup, ob es sich um eine Erstinstallation oder ein Update handelt. Dementsprechend sollte dieser Eintrag im Laufe des Projektes nicht mehr verändert werden!

Anstelle eines sprechenden Namens kann hier natürlich auch eine GUID verwendet werden. Die Compiler-GUI und auch Inno Script Studio besitzen zu deren Erzeugung einen Menübefehl.

Hierbei sehen wir bereits, dass einige Informationen mehrfach eingetragen werden. Zur besseren Übersicht bietet sich daher ein Verfahren an, das wir bereits vom Programmieren her kennen.

Wann immer wir eine immer gleiche Information an mehreren Stellen des Programms verwenden müssen, definieren wir eine globale Konstante und verwenden in der Folge dann nur noch diese anstelle des tatsächlichen Wertes. So ist eine Änderung des Wertes schnell gemacht und die Semantik ist auch sofort klar. Das geht natürlich auch in InnoSetup.

Konstanten im Setup-Script

Die Definition der Konstanten erfolgt dabei zwingend am Anfang des Script vor der **[Setup]**-Sektion nach folgendem Muster:

```
#define MyAppName "My Awesome AccessApp"
#define MyAppVersion "1.0.0"
#define MyAppPublisher "Christoph Jüngling"
#define MyAppURL "https://www.meine-homepage.de"
#define MyAppExeName "Testdatenbank.accdb"
```

Nun können wir das obige Script entsprechend überarbeiten. Dabei werden die Konstanten nach dem Muster **{#NameDerKonstante}** an der Stelle eingefügt, wo deren Wert stehen soll.

```
[Setup]
AppId={#MyAppName}
AppName={#MyAppName}
AppVersion={#MyAppVersion}
AppVerName={#MyAppName} v{#MyAppVersion}
AppPublisher={#MyAppPublisher}
AppPublisherURL={#MyAppURL}
AppSupportURL={#MyAppURL}
AppUpdatesURL={#MyAppURL}
```

Das obige sind natürlich nur Vorschläge. Nach dem nun bekannten Muster lassen sich leicht weitere Konstanten hinzufügen, zum Beispiel um die drei URLs auch wirklich unterschiedlich zu gestalten.

Mindestens eine weitere Information müssen wir noch angeben, und zwar den Ort, an dem die Installation erfolgen soll:

```
DefaultDirName={userpf}\{#MyAppName}
```

Diese heißt **DefaultDirName**, da das Verzeichnis während der Installation durch den User noch geändert werden kann. Die vordefinierte Konstante **{userpf}** wird von InnoSetup erst bei der Installation ausgewertet.

Es handelt sich um das benutzerspezifische Programm-Verzeichnis **C:\Users\USERNAME\AppData\Local\Programs** (siehe auch im Beitrag **Setup für Access-Anwendungen**, **www.access-im-unternehmen.de/1316** unter **Wohin mit dem Frontend?**).

In diesem Zusammenhang wird auch noch eine weitere Frage wichtig: Welche Rechte benötigt der User, der dieses Setup ausführen will?

Bei »normalen« Setups ist es üblich und auch notwendig, dass man Admin-Berechtigungen hat. Wenn wir

aber unsere Applikation wirklich nur im userspezifischen Teil der Festplatte installieren wollen, ist das nicht notwendig, dann genügen die Rechte des Users völlig. Das teilen wir dem Setup-Script natürlich ebenfalls mit:

```
PrivilegesRequired=lowest
```

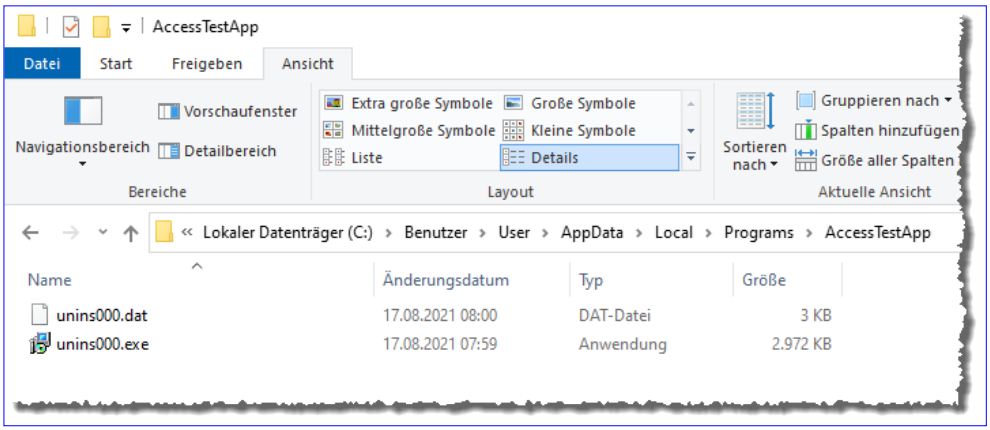


Bild 2: Weitgehend leeres Installationsverzeichnis

Nun können Sie das Setup bereits compilieren, obwohl im Moment natürlich noch gar nichts installiert wird. Zu diesem Zweck drücken Sie **F9** in InnoSetup oder Inno Script Studio, lassen das Setup durchlaufen und schauen dann in das angegebene Verzeichnis. Dort finden wir im Moment nur zwei Dateien (siehe Bild 2).

Es handelt sich um den Uninstaller und die von ihm benötigte Daten-Datei. Zugleich finden wir in der Systemsteuerung unter **Programme hinzufügen oder entfernen** einen passenden Eintrag zu unserer Installation (siehe Bild 3).

Den Button **Deinstallieren** sollten Sie zum Kennenlernen ruhig betätigen, und vergewissern Sie sich bitte anschließend, dass das Installationsverzeichnis wieder vollständig verschwunden ist.

Nun haben wir herausgefunden, dass das Setup grundsätzlich funktioniert. Jetzt wollen wir die Access-Datenbank hinzufügen.

Was und wohin installieren?

Für diese Frage ist die Sektion **[Files]** zuständig. Hier werden alle Dateien aufgeführt, die in das Setup einge-

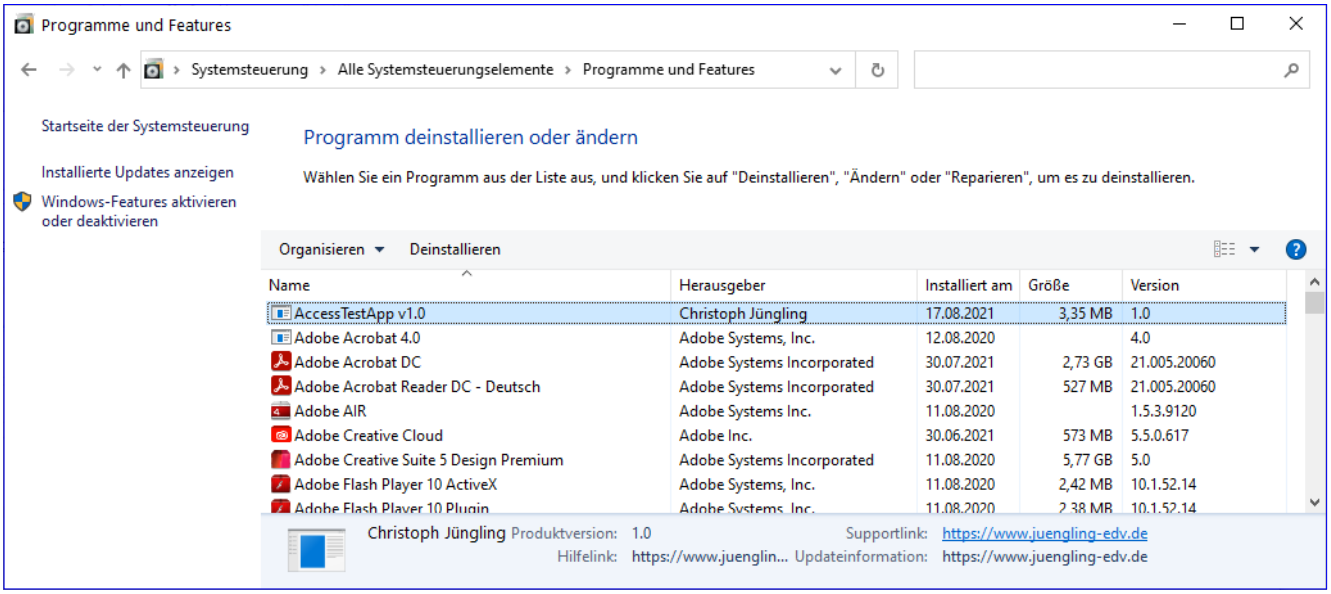


Bild 3: Eintrag in der Systemsteuerung

Access-Optionen per Ribbon ändern

Es gibt einige Access-Optionen, die man immer wieder nutzt. In meinem Fall ist es zum Beispiel die Einstellung, ob Formulare nun als überlappende Fenster oder als Dokumente im Registerkartenformat angezeigt werden sollen. Sicher haben Sie ähnliche Einstellungen, die Sie oft ändern oder die Sie vielleicht einfach nur schnell einsehen können – was bei dem mittlerweile recht umfangreich gewordenen Optionen-Dialog schon einige Sekunden kosten kann. Warum also nicht ein COM-Add-In bauen, dass die Informationen der wichtisten Access-Einstellungen immer direkt im Ribbon anzeigt – anstatt irgendwo versteckt im Optionen-Dialog? Und da wir mit twinBASIC auch noch ein praktisches Tool zum Erstellen von COM-Add-Ins zur Hand haben, können wir direkt loslegen!

Voraussetzung: Visual Studio Code und twinBASIC

Wenn Sie dauerhafte Ergänzungen oder Änderungen am Ribbon vornehmen wollen, benötigen Sie ein COM-Add-In. Eine andere Möglichkeit gibt es nur, wenn Sie mit den eingebauten Funktionen zum Anpassen der Benutzeroberfläche arbeiten wollen – und die ist bei Weitem nicht ausreichend.

Ein COM-Add-In konnten Sie früher mit Visual Studio 6 bauen, heutzutage mit Visual Studio .NET. Das ist allerdings nicht besonders komfortabel und erfordert umfangreichere Softwarevoraussetzungen als Visual Studio 6 gebaute COM-Add-Ins. Seit kurzem gibt es allerdings eine Erweiterung namens twinBASIC für Visual Studio Code, mit der Sie zum Beispiel COM-Add-Ins für Access und den VBA-Editor programmieren können. Wie das geht, haben wir bereits in den Beiträgen **twinBASIC – VB/VBA mit moderner Umgebung** (**www.access-im-unternehmen.de/1303**), **twinBASIC – COM-Add-Ins für Access** (**www.access-im-unternehmen.de/1306**) und anderen erläutert.

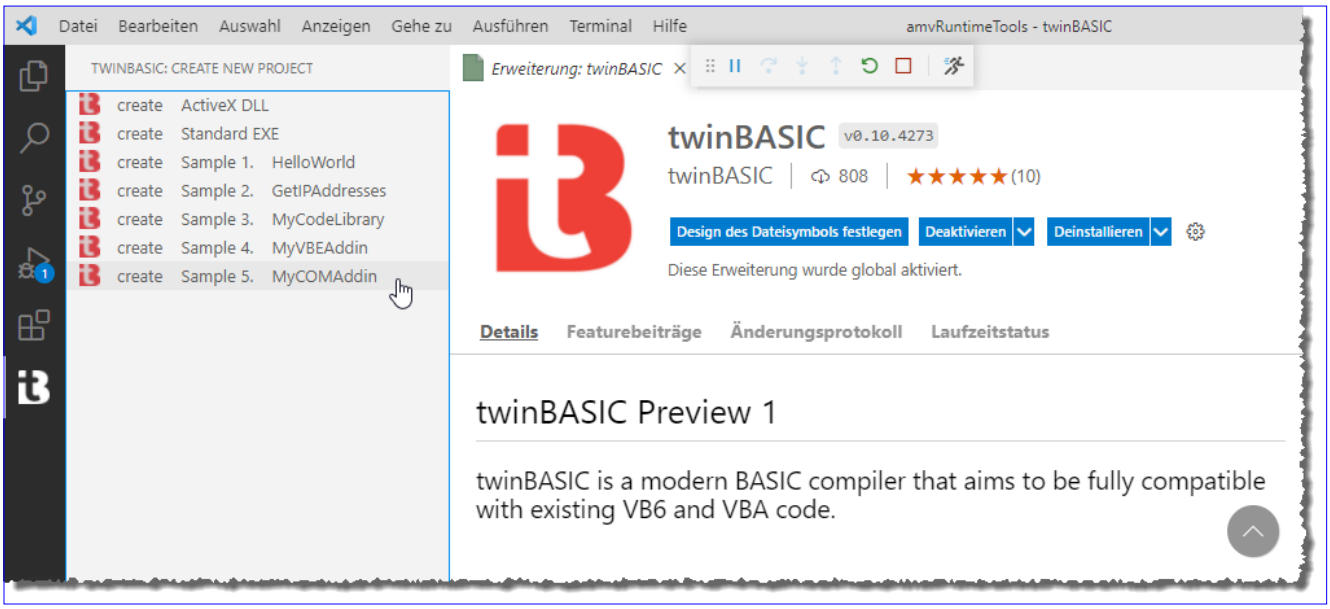


Bild 1: Erstellen eines neuen COM-Add-Ins

Im erstgenannten Beitrag erfahren Sie alles über die Installation, in letzterem die Grundlagen für das Erstellen von COM-Add-Ins. Darauf setzt der genannte Beitrag auf, Details zu diesen Themen finden Sie in den genannten Beiträgen.

Neues COM-Add-In anlegen

Um ein neues COM-Add-In anzulegen, klicken Sie in Visual Studio Code links auf das twinBASIC-Symbol. Es erscheint eine Liste mit Projektvorlagen, wo Sie die Vorlage **Sample 5 MyCOMAddin** auswählen (siehe Bild 1).

Anschließend erscheint ein Dialog, in dem Sie die anzulegende Datei samt Verzeichnis auswählen. Hier geben wir **amvAccessOptionsGoRibbon.twinproj** an. Dies fügt einige Dateien zum gewählten Verzeichnis hinzu, von denen die folgenden für uns interessant sind:

- **amvAccessOptionsGoRibbon_ACCESS_RegisterAddin32.reg**: Datei zum Hinzufügen der Registry-Einträge für das COM-Add-In
- **amvAccessOptionsGoRibbon_ACCESS_UnregisterAddin32.reg**: Datei zum Entfernen der Registry-Einträge für das COM-Add-In
- **amvAccessOptionsGoRibbon_myCOMAddin.code-workspace**: Arbeitsumgebung
- **amvAccessOptionsGoRibbon_myCOMAddin.twinproj**: Projektdatei

Das Projekt wird allerdings auch direkt in Visual Studio Code angezeigt.

Was bietet die Vorlage?

Mit der Vorlage erhalten Sie bereits fast alles, was Sie benötigen:

- Die Implementierung der Schnittstelle **IDTExtensibility2**, die einige Ereignisprozeduren bereitstellt, die zu verschiedenen Zeitpunkten beim Verwenden des COM-Add-Ins ausgelöst werden – zum Beispiel beim Start (**OnConnection**)
- Die Implementierung der Schnittstelle **IRibbonExtensibility** in Form der Ereignisprozedur **GetCustomUI**. Diese wird beim Start des COM-Add-Ins ausgelöst und stellt das Ribbon zusammen, über das die Benutzeroberfläche beziehungsweise die Funktionen des COM-Add-Ins bereitgestellt werden sollen.
- Eine von dieser Ereignisprozedur aufgerufene Callbackfunktion, die zeigt, wie Sie Funktionen über das Ribbon aufrufen können.

COM-Add-In zum Laufen bringen

Dementsprechend brauchen Sie nur zwei Schritte zu erledigen, damit das COM-Add-In läuft:

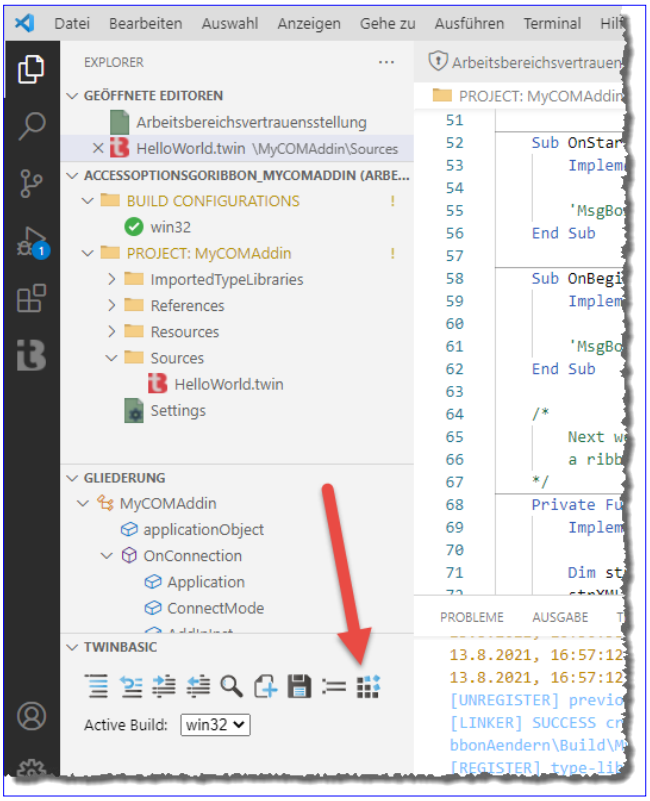


Bild 2: COM-Add-In kompilieren

- Das COM-Add-In kompilieren, indem Sie auf die Schaltfläche Build klicken (siehe Bild 2).
- Die Registrierungsdatei **amvAccessOptionsGoRibbon_ACCESS_RegisterAddin32.reg** starten, indem Sie diese doppelt anklicken und die folgenden Meldungen bestätigen.

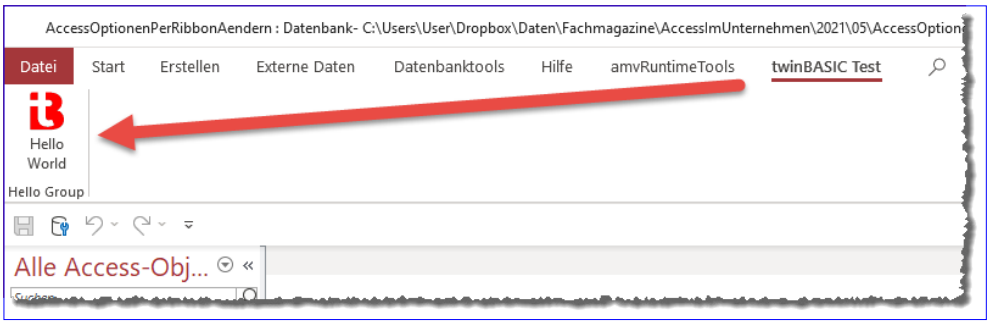


Bild 3: Das COM-Add-In ist nach wenigen Mausklicks einsatzbereit

Wenn Sie nun eine Access-Datenbank öffnen, erscheint ein neuer Eintrag namens **twinBASIC Test** im Ribbon mit einer eigenen Schaltfläche (siehe Bild 3).

Im Gegensatz zu dem COM-Add-In, das wir wie im Beitrag **twinBASIC – COM-Add-Ins für Access** erstellt haben, kann twinBASIC nun auch benutzerdefinierte Icons anzeigen. Wie Sie diese hinzufügen und über entsprechende Callbackfunktionen aufrufen, zeigen wir im Beitrag **Benutzerdefinierte Bilder in twinBASIC** (**www.access-im-unternehmen.de/1325**).

COM-Add-In vorbereiten

Da wir wissen, dass wir das COM-Add-In mit Access einsetzen wollen, können wir eine Objektvariable speziell zum Aufnehmen der aktuellen Access-Instanz deklarieren:

```
Private objAccess As Access.Application
```

Außerdem weisen wir diese in der Prozedur **OnConnection** wie folgt gezielt zu:

```
Private Function GetCustomUI(ByVal RibbonID As String) As String _
    Implements IRibbonExtensibility.GetCustomUI
    Dim strXML As String
    strXML &= "<?xml version="1.0"?>" & vbCrLf
    strXML &= "<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui">" & vbCrLf
    strXML &= "  <ribbon>" & vbCrLf
    strXML &= "    <tabs>" & vbCrLf
    strXML &= "      <tab id="tabOptions" label="Optionen">" & vbCrLf
    strXML &= "        <group id="grpThisDatabase" label="&lt;Aktuelle Datenbank&gt;">" & vbCrLf
    strXML &= "          <editBox label="Datenbanktitel:"id="txtTitle" maxLength="255" " _
           & " sizeString="xxxxxxxxxxxxxxxxxxxxxxxxx" getText="getText" onChange="onChange"/>" & vbCrLf
    strXML &= "        </group>" & vbCrLf
    strXML &= "      </tab>" & vbCrLf
    strXML &= "    </tabs>" & vbCrLf
    strXML &= "  </ribbon>" & vbCrLf
    strXML &= "</customUI>" & vbCrLf
    Return strXML
End Function
```

Listing 1: Einlesen des Ribbons mit dem Textfeld zum Festlegen des Datenbanktitels

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

Hier tragen wir den mit dem Parameter **Application** übergebenen Verweis auf die aktuelle Access-Instanz in die Variable **objAccess** ein.

Optionen mit dem Ribbon einstellen

Um Optionen mit dem Ribbon einzustellen, benötigen wir mehr als nur ein paar Schaltflächen. Immerhin wollen wir ja vor dem Ändern einer Option auch wissen, wie ihr aktueller Wert lautet! Also benötigen wir Steuerelemente wie Kontrollkästchen, Textfelder et cetera.

Da wir diese dynamisch einlesen und auch anpassen wollen, stellen wir ihre Werte zu Beginn mit entsprechenden Callback-Funktionen ein. Später ändern wir diese dann über das Ribbon und sorgen dafür, dass die Änderungen sich auch in den entsprechenden Access-Optionen niederschlagen.

Wir beginnen mit einer ersten Option, um uns den Ablauf anzusehen. Dies soll der Titel der Access-Anwendung sein. Der Vorteil gegenüber einigen anderen Optionen ist, dass wir Änderungen direkt in der Titelleiste angezeigt bekommen – im Gegensatz zu vielen anderen Optionen, die sich erst beim erneuten Öffnen der Datenbankanwendung zeigen.

Also fügen wir der Ribbon-Definition ein Textfeld hinzu. Danach sieht Prozedur **GetCustomUI** zur Definition des Ribbons wie in Listing 1 aus. Hier finden wir statt der bisher vorhandenen Schaltfläche ein **editBox**-Element. Für dieses haben wir die Eigenschaft **label** auf **Datenbanktitel** und **maxLength** auf **255** eingestellt. Letztere sorgt dafür, dass der Titel die maximale Zeichenzahl von **255** nicht überschreitet. Damit die **editBox**-Steuerelemente breite genug dargestellt werden, haben außerdem das Attribut **sizeString** auf den Wert **xxxxxxxxxxxxxxxxxxxxxxxx** eingestellt. Außerdem haben wir gleich zwei Callbackattribute definiert:

- **getText**: Gibt eine Callbackfunktion an, die beim erstmalige Anzeigen oder der Anforderung der Aktualisierung der Anzeige ausgelöst werden soll. Hier wollen wir den aktuellen Titel aus den Access-Optionen einlesen und anschließend im **editBox**-Steuerelement anzeigen.

- **onChange**: Gibt eine Callbackfunktion an, die beim Ändern des Wertes des **editBox**-Steuerelements ausgelöst wird. Hiermit wollen wir nach Änderungen durch den Benutzer den neuen Wert in die Access-Optionen übertragen.

Einlesen des aktuellen Datenbanktitels

Die Callbackfunktion **getText** wird ausgelöst, wenn das Ribbon geladen oder wenn per Code eine der Methoden **Invalidate** oder **InvalidateControl** der in einer Variablen gespeicherten Ribbon-Instanz ausgelöst wird und das Steuerelement danach erstmals sichtbar wird.

Sie erhält als ersten Parameter einen Verweis auf das aufrufende Steuerelement. Der zweite Parameter erwartet die im **editBox**-Steuerelement anzuzeigende Zeichenkette.

Wenn Sie die Prozedur wie nachfolgend in das twinBASIC-Projekt einfügen, werden einige Elemente rot unterstrichen. Das liegt daran, dass wir noch keinen Verweis auf die DAO-Bibliothek hinzugefügt haben:

```
Function GetText(control As IRibbonControl) As String
    Dim db As DAO.Database
    Dim prp As DAO.Property
    Dim strText As String
    On Error Resume Next
    Set db = objAccess.CurrentDb
    Set prp = db.Properties("AppTitle")
```

```
    On Error GoTo 0
    If prp Is Nothing Then
        strText = "<Kein
Titel>"
    Else
        strText = prp.Value
    End If
    On Error GoTo 0
    GetText = strText
End Function
```

Das holen wir nach, indem wir in twinBASIC zum Bereich **Settings** wechseln und dort unter **COM Type Library / Active X References** einen Verweis auf die Bibliothek **Microsoft Office 16.0 Access Database Engine Object Library [v12.0]** hinzufügen (siehe Bild 4). Danach unbedingt mit **Strg + S** speichern, damit die Änderungen wirksam werden!

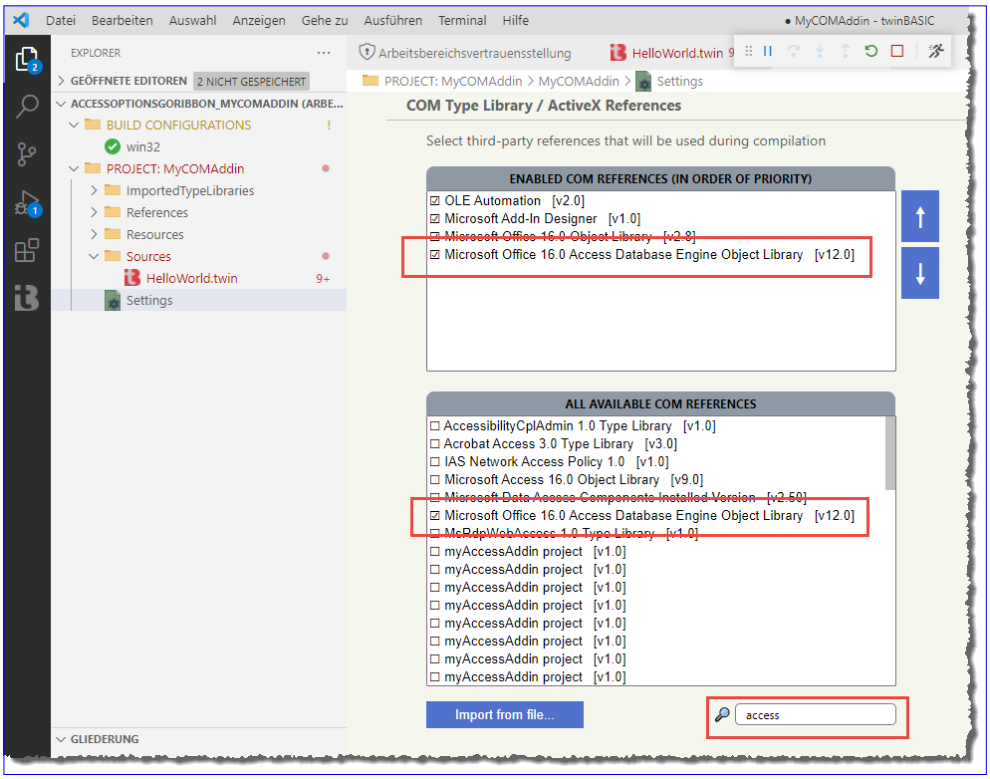


Bild 4: Hinzufügen eines Verweises auf die DAO-Bibliothek

Danach sind fast alle roten Unterstreichungen verschwunden, nur eine nicht – die unter **CurrentDb**. **CurrentDb** ist auch keine Eigenschaft der DAO-Bibliothek, sondern der Access-Bibliothek.

Also fügen wir auch diese noch mit dem Eintrag **Microsoft Access 16.0 Object Library [v9.0]** hinzu. Nach dem erneuten Speichern können wir uns nun den Code der Prozedur **getText** ohne rote Unterstreichungen ansehen.

Die Prozedur deaktiviert zunächst die eingebaute Fehlerbehandlung und ruft den Wert der Eigenschaft **AppTitle** des **Database**-Objekts der aktuellen Datenbank ab. Die Fehlerbehandlung deaktivieren wir dabei, weil ein Fehler ausgelöst wird, wenn diese Eigenschaft noch nicht oder nicht mehr vorhanden ist. Sie ist nur vorhanden, wenn diese bereits durch den Benutzer über den Optionen-Dialog von Access gesetzt wurde (oder per Code, wie wir gleich demonstrieren).

Danach aktivieren wir die Fehlerbehandlung wieder und prüfen per **If...Then**-Bedingung, ob **prp** eine Eigenschaft zugewiesen wurde.

Das ist nur der Fall, wenn die Eigenschaft aktuell vorhanden ist. Ist **prp** leer, übergeben wir die Zeichenkette **<Kein Titel>** als anzuzeigenden Text, enthält **prp** einen Wert, übergeben wir diesen an den Parameter **text**.

Andere Callback-Signatur in twinBASIC-Ribbons

Wie schon im Beitrag **twinBASIC – COM-Add-Ins für Access** festgestellt, sehen wir auch hier wieder eine andere Signatur für die Callbackfunktion. Sie lautet hier:

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

Unter VBA würde diese lauten:

```
Sub getText(control As IRibbonControl,
ByRef text)
```

Die hier verwendete Signatur ist die gleiche, die wir auch in einem VB6-COM-Add-In nutzen würden. Da twinBASIC zum Nachfolger von VB6 avanciert und wir noch weitere Tools damit entwickeln werden, stellen wir in einem weiteren Beitrag namens **Ribbon: Callback-Signaturen für VBA und VB6** (**www.access-im-unternehmen.de/1324**) die Unterschiede vor.

```
Sub OnChange(control As IRibbonControl, text As String)
    Dim db As DAO.Database
    Dim prp As DAO.Property
    Set db = objAccess.CurrentDb
    If Len(text) = 0 Then
        db.Properties.Delete "AppTitle"
    Else
        On Error Resume Next
        Set prp = db.Properties("AppTitle")
        If Not Err.Number = 0 Then
            Set prp = db.CreateProperty("AppTitle", dbText, text)
            db.Properties.Append prp
        Else
            prp.Value = text
        End If
    End If
    objAccess.RefreshTitleBar
End Sub
```

Listing 2: Ändern einer Option nach Änderung im Ribbon

Speichern des geänderten Datenbanktitels

Mit der gleichen Eigenschaft beschäftigt sich die Callbackfunktion **onChange** (siehe Listing 2). Diese hat die gleichen Parameter. Allerdings wird sie ausgelöst, wenn der Benutzer den im **editBox**-Steuerelement enthaltenen Text ändert und die Änderung beispielsweise durch Verlassen des Textfeldes abschließt. Dementsprechend ist der zweite Parameter **text** diesmal auch kein Rückgabeparameter, sondern er liefert den neu vom Benutzer eingegebenen Text.

Den Inhalt von **text** prüft die Prozedur zunächst in einer **If...Then**-Bedingung. Es kann sein, dass der Benutzer das **editBox**-Steuerelement komplett geleert hat, dann soll auch der Titel geleert werden. Das funktioniert allerdings nur in der gewünschten Form, indem wir die Eigenschaft **AppTitle** wieder löschen. Das erledigt die Prozedur im **If**-Teil der Bedingung mit der **Delete**-Methode der **Properties**-Auflistung.

Hat der Benutzer jedoch einen Titel mit mindestens einem Zeichen eingegeben, dann wollen wir die Eigenschaft **AppTitle** auf diesen Wert einstellen. Dazu versucht die Prozedur, nach vorheriger Deaktivierung der Fehlerbehaldnung, die Eigenschaft mit der Variablen **prp** zu referenzieren. Ist dabei ein Fehler aufgetreten, erstellt die Prozedur die Eigenschaft neu und hängt diese an die Auflistung **Properties** des **Database**-Objekts der aktuellen Datenbank an.

Ist die Eigenschaft bereits vorhanden, stellt die Prozedur diese lediglich auf den Wert aus dem Parameter **text** ein.

In jedem Fall soll die Anzeige der Titelleiste anschließend aktualisiert werden, was die Methode **RefreshTitleBar** des **Application**-Objekts der aktuellen Access-Instanz erledigt. Diese haben wir ja zuvor in die Variable **objAccess** eingelesen.

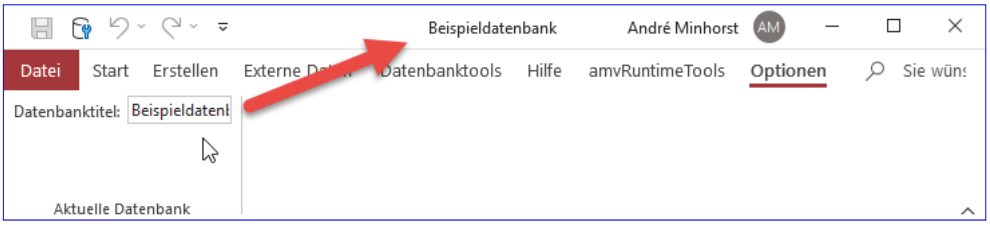


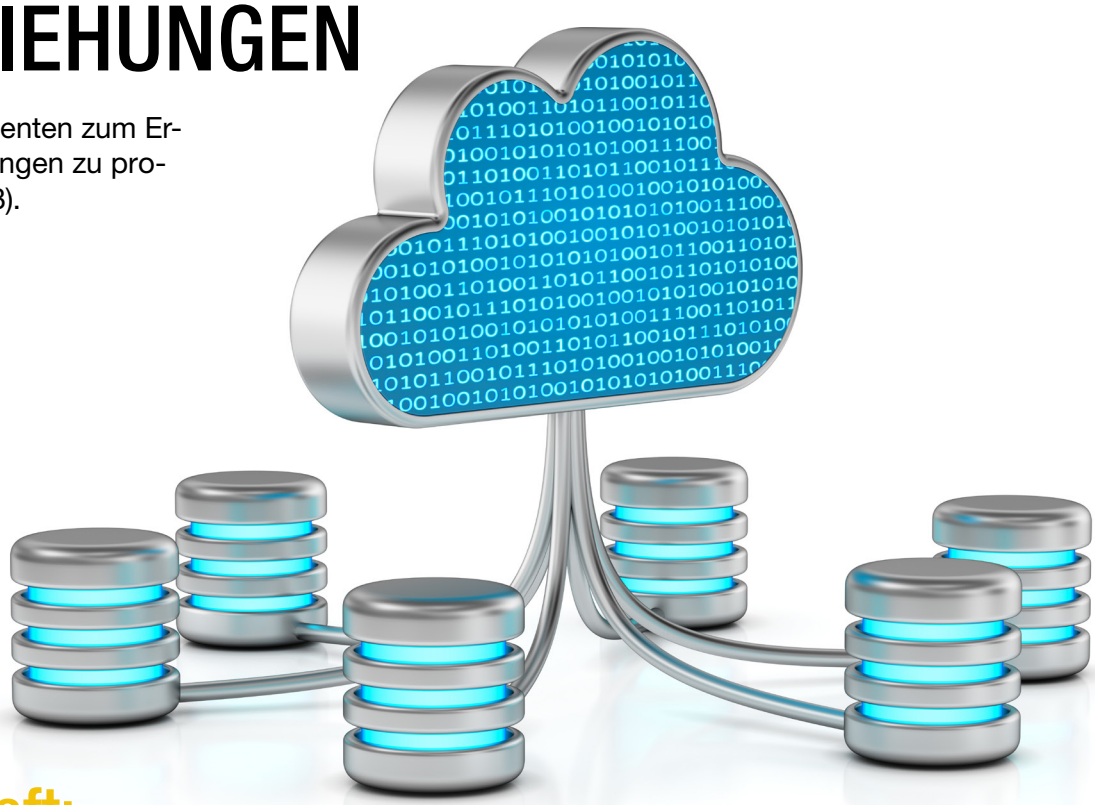
Bild 5: Die erste Access-Option im Access-Ribbon

HAUFE.
Ausgabe 06/2021

ACCESS
IM UNTERNEHMEN

ASSISTENT FÜR M:N-BEZIEHUNGEN

Lernen Sie, einen Assistenten zum Erstellen von m:n-Beziehungen zu programmieren (ab Seite 53).



In diesem Heft:

www.access-im-unternehmen.de
Mat-Nr. H1583-5128

ARCHIV SEITE 385

m:n-Beziehungen leicht gemacht

Wer in die Datenbankentwicklung einsteigt und damit auch in den Entwurf von Datenmodellen, für den kann nach der 1:n-Beziehung die m:n-Beziehung die nächste Herausforderung sein. 1:n-Beziehung baut man unter Access ganz einfach mit dem Nachschlage-Assistenten. Für die Erstellung von m:n-Beziehungen und die dafür benötigte Verknüpfungstabelle sucht man vergeblich nach einem Tool. Kein Problem: Wir haben eines programmiert und zeigen die notwendigen Aktionen in einer ausführlichen Schritt-für-Schritt-Anleitung!

Eigentlich ist eine m:n-Beziehung ja nur eine Kombination zweier 1:n-Beziehungen. Allerdings benötigen wir ausgehend von den beiden zu verknüpfenden Tabellen nicht nur zwei Beziehungen, sondern auch noch eine Verknüpfungstabelle, welche die beiden Beziehungen zusammenführt. Unser Assistent nimmt Ihnen nicht nur die Aufgabe ab, die Beziehungen zu erstellen, sondern definiert auch gleich noch die Verknüpfungstabelle. Sie brauchen nur noch die beiden Tabellen auszuwählen, die verknüpft werden sollen, sowie die Primärschlüsselfelder, die an der Verknüpfung beteiligt sind – den Rest erledigt der Assistent für uns.

Wie Sie diesen programmieren, lernen Sie im Beitrag **Assistent für m:n-Beziehungen** ab Seite 53.

Viele Entwickler wollen, dass der Benutzer die Anwendung genau so nutzt, wie sie es vorgesehen haben – und dass der Benutzer keinen Zugriff auf Elemente wie beispielsweise den Navigationsbereich hat. Solche Elemente kann man ausschalten, sodass sie normalerweise nicht angezeigt werden. Betätigt der Benutzer jedoch beim Öffnen der Datenbank die Umschalttaste, umgeht er diese Einstellungen. Um das zu verhindern, können Sie die Funktion der Umschalttaste beim Öffnen blockieren. Wie das gelingt, lesen Sie unter **Umschalttaste sperren mit AllowByPassKey** ab Seite 2.

Wenn Sie Ihre Anwendung mit einer benutzerdefinierten Ribbondefinition ausgestattet haben, kann es zu Funktionsstörungen kommen, nachdem ein unbehandelter Laufzeitfehler aufgetreten ist. Welche Schritte nötig sind, damit das Ribbon dann immer noch fehlerfrei funktioniert, erfahren Sie im Beitrag **Ribbonvariable fehlerresistent machen** (ab Seite 5).

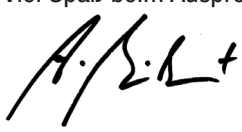
Ein zweiter Beitrag zum Thema Ribbon erläutert, wie Sie beim Auswählen eines der Ribbonreiter durch den Benutzer immer direkt ein dazu passendes Formular anzeigen können (**Bei Tab-Wechsel im Ribbon ein Formular anzeigen**, ab Seite 11).

Das Thema Setup für Access-Anwendungen behandeln wir weiter im Beitrag **Setup für Access: Vertrauenswürdige Speicherorte** (ab Seite 30). Hier schauen wir uns an, wie Sie direkt bei der Installation dafür sorgen können, dass die Access-Anwendung als vertrauenswürdig eingeordnet wird.

Ein wichtiges Objekt bei der Programmierung von Access-Anwendungen ist das Application-Objekt. Der Beitrag **Das Application-Objekt** zeigt ab Seite 36 alle Eigenschaften und Methoden dieses Objekts.

Schließlich liefert der Beitrag **Formulare per VBA erstellen** Informationen darüber, wie Sie programmgesteuert Formulare erstellen können – was beispielsweise für die Programmierung von Assistenten zum Erstellen von Formularen wichtig sein kann (ab Seite 16).

Viel Spaß beim Ausprobieren!



Ihr André Minhorst

Umschalttaste sperren mit AllowByPassKey

Unter Access können Sie einige Einstellungen vornehmen, damit der Benutzer kaum noch etwas davon sieht, dass es sich bei der Anwendung um eine Access-Anwendung handelt. Sie können beispielsweise den Navigationsbereich ausblenden oder die eingebauten Ribbon- und Backstage-Einträge mit einer benutzerdefinierten Ribbon-Definition ausblenden. Pfiffige Anwender finden allerdings schnell heraus, wie sich diese Änderungen beim Öffnen einer Datenbank blockieren lassen: durch einfaches Drücken und Halten der Umschalttaste. Also zeigen wir in diesem Beitrag noch einen Weg, wie Sie auch das noch ein wenig erschweren können. Eine wichtige Rolle spielt dabei eine Eigenschaft namens AllowByPassKey.

Das Öffnen einer Datenbankanwendung bei gedrückter Umschalttaste ist ein gängiges Mittel, um sämtliche vom Entwickler programmierten Aktionen, die beim Starten der Anwendung ausgeführt werden sollen, zu unterbinden. Unter anderem blockieren Sie die folgenden Aktionen:

- das Öffnen des als Startformular angegebenen Formulars
- das Ausführen des **AutoExec**-Makros
- die Anwendung einer benutzerdefinierten Ribbondefinition aus der Tabelle **USysRibbons**, die über die Option **Name des Menübands** ausgewählt wurde
- das Ausblenden des Navigationsbereichs über die Access-Optionen für die aktuelle Datenbank
- Verwenden der Spezialtasten von Access wie beispielsweise **F11** zum Anzeigen des Navigationsbereichs

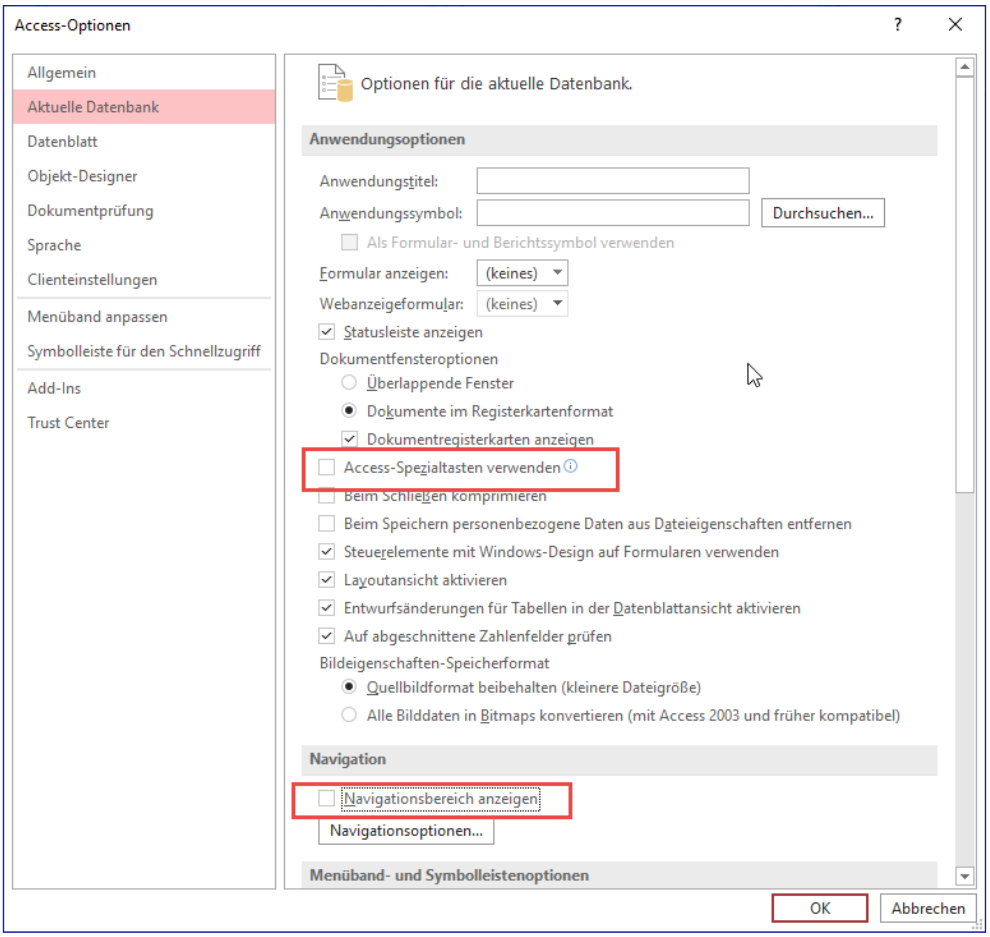


Bild 1: Zwei Beispiele für Optionen, die man per Umschalttaste aushebeln kann

Hält der Benutzer die Umschalttaste beim Öffnen gedrückt, wird die Datenbank so geöffnet, als ob all diese Änderungen nicht vorgenommen worden wären. Also wollen wir die Wirkung des Drückens der Umschalttaste unterbinden.

```
Public Sub UmschalttasteErlauben(Optional bolSperren As Boolean = True)
    Dim db As DAO.Database
    Dim prp As DAO.Property
    Set db = CurrentDb
    On Error Resume Next
    Set prp = db.Properties("AllowBypassKey")
    If Err.Number = 3270 Then
        On Error GoTo 0
        Set prp = db.CreateProperty("AllowBypassKey", dbBoolean, bolSperren)
        db.Properties.Append prp
    Else
        prp.Value = bolSperren
    End If
End Sub
```

Listing 1: Aktivieren oder Deaktivieren der Umschalttaste

Wir haben in den Optionen der Beispieldatenbank die beiden Einstellungen aus Bild 1 vorgenommen. Diese sorgen dafür, dass erstens der Navigationsbereich nicht angezeigt wird und zweitens die Spezialtasten von Access nicht genutzt werden können, um beispielsweise mit **F11** den Navigationsbereich doch noch zu aktivieren.

Die Eigenschaft AllowByPassKey

Es gibt eine Eigenschaft namens **AllowByPassKey**, welche die Funktion der Umschalttaste beim Öffnen einer Access-Anwendung erlaubt – oder auch nicht. Diese Eigenschaft wird als Element der **Properties**-Auflistung der aktuellen Datenbank angelegt. Dies können wir nur per VBA erledigen, weshalb wir die Prozedur aus Listing 1 angelegt haben.

Diese heißt **UmschalttasteErlauben** und erwartet einen **Boolean**-Wert als Parameter:

- **True**: Erlaubt die Funktion der Umschalttaste beim Öffnen der Datenbankanwendung.
- **False**: Sperrt die Funktion der Umschalttaste beim Öffnen der Datenbankanwendung.

Die Prozedur versucht bei deaktivierter Fehlerbehandlung, auf ein Element namens **AllowByPassKey** der Auflistung **Properties** für das mit **CurrentDb** ermittelte **Database**-Objekt der aktuellen Datenbank zuzugreifen. Löst dies einen Fehler aus, ist das Element noch nicht vorhanden und wir legen es mit der **CreateProperty**-Methode an. Diese erhält den Namen der zu erstellenden Property, den Datentyp (**dbBoolean**) und den Wert aus dem Prozedurparameter **bolSperren** als Parameter. Anschließend wird die Property noch mit der **Append**-Methode an die Liste der vorhandenen Properties angehängt.

Ist die Property bereits vorhanden, löst der Zugriff darauf keinen Fehler aus und wir können ihr einfach den gewünschten Wert zuweisen.

Der Aufruf zum Verhindern der Funktion der Umschalttaste, abgesetzt beispielsweise im Direktbereich, lautet:

```
UmschalttasteErlauben False
```

In beiden Fällen können wir den Wert der Eigenschaft anschließend mit der folgenden Anweisung über den Direktbereich des VBA-Editors abfragen:

```
? CurrentDb.Properties("AllowByPassKey")
```

Wenn diese Eigenschaft den Wert **False** liefert, wird die Wirkung der Umschalttaste beim nächsten Öffnen dieser Accessdatenbank unterbunden.

Ribbonvariable fehlerresistent machen

In VBA-Projekten von Access-Datenbanken (und in VBA im Allgemeinen) gibt es das Problem, dass das Auftreten von unbehandelten Laufzeitfehlern dazu führt, dass Objektvariablen geleert werden. Das ist insbesondere dann nachteilig, wenn Sie mit dem Ribbon arbeiten und dieses zwischendurch mit der Invalidate-Methode ungültig machen müssen, damit die Attribute mit get...-Prozeduren erneut eingelesen werden können. Der Aufruf von Invalidate führt dann unweigerlich zu einem Laufzeitfehler. Dieser Beitrag beschreibt das grundlegende Beispiel und liefert eine Lösung, mit der Sie sich keine Sorgen mehr um Objektvariablen machen müssen, die durch Laufzeitfehler geleert werden.

Leeren von Objektvariablen nach Laufzeitfehlern reproduzieren

Als Erstes schauen wir uns an, mit welchem Problem wir es überhaupt zu tun haben. Das Problem betrifft nicht nur die Variablen, die in Zusammenhang mit dem Ribbon erstellt werden. Allerdings haben Objektvariablen üblicherweise eine überschaubare Gültigkeitsdauer – sie werden in der Regel innerhalb einer Prozedur deklariert und verlieren ihre Gültigkeit nach dem Beenden der Prozedur auch wieder.

Grundsätzlich ist es ohnehin nicht empfehlenswert, mit global deklarierten Variablen zu arbeiten, da diese nicht nur von überall gelesen, sondern auch von überall geändert werden können. Dabei treten schnell Probleme auf, wenn man von mehreren Stellen aus auf solche Variablen zugreift und nicht genau weiß, was man da tut.

Spätestens, wenn mal jemand anderer die Anwendung übernimmt und anpasst und sich nicht bewusst ist, das er dort mit global deklarierten Variablen arbeitet, könnte es zu Problemen kommen.

Beispiel für Datenverlust nach Laufzeitfehlern

Schauen wir uns also zunächst einmal den grundsätzlichen Effekt an, um zu sehen, womit wir es zu tun haben. Dazu deklarieren wir in einem Standardmodul die folgende Variable:

```
Public db As DAO.Database
```

db ist nun öffentlich deklariert und kann von überall innerhalb der Anwendung gesetzt und gelesen werden. Wir setzen diese in einer kleinen Prozedur im gleichen Modul:

```
Public Sub DbSetzen()
    Set db = CurrentDb
End Sub
```

Wir können nun im Direktbereich von Access über die Objektvariable **db** auf das referenzierte Objekt zugreifen und so beispielsweise den Pfad der aktuellen Anwendung ermitteln:

```
? db.Name
C:\Users\...\RibbonvariableFehlersicherMachen.accdb
```

Nun provozieren wir einen unbehandelten Laufzeitfehler, indem wir die folgende Prozedur auslösen:

```
Public Sub RaiseError()
    Debug.Print 1 / 0
End Sub
```

Dies ruft die Fehlermeldung aus Bild 1 hervor. Klicken wir hier auf **Beenden**, vervollständigen wir den unbehandelten Laufzeitfehler. Durch einen Klick auf **Debuggen** und gege-

benenfalls Änderungen im Code könnten wir den Fehler behandeln, was wir aber an dieser Stelle nicht wollen. Stattdessen klicken wir schlicht auf **Beenden**.

Dies führt zunächst einmal zu keiner sichtbaren Folge – davon abgesehen, dass die Prozedur an dieser Stelle einfach abgebrochen wird.

Es gibt jedoch noch einen weiteren Nebeneffekt: Die Variable **db** ist nun leer und zeigt nicht mehr auf das soeben referenzierte Objekt des Typs **Database**. Wie können wir das herausfinden? Indem wir einfach nochmal versuchen, den Pfad der aktuellen Anwendung im Direktbereich auszugeben.

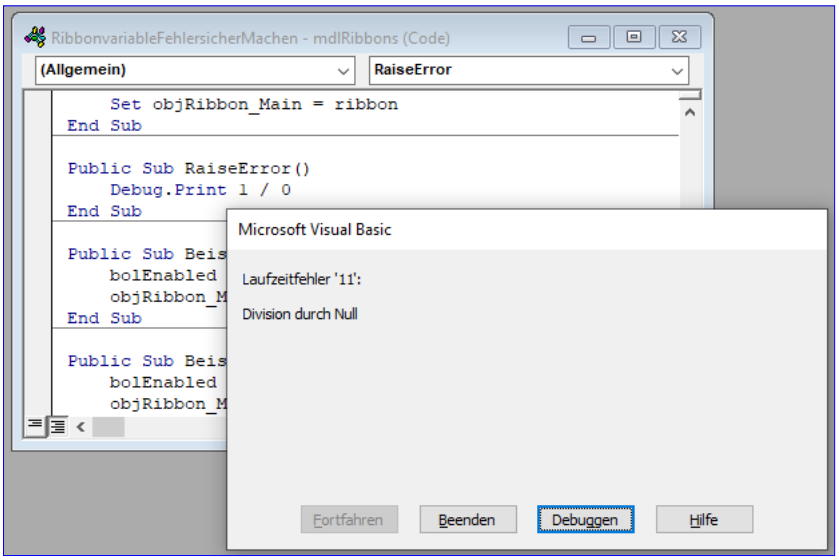


Bild 1: Unbehandelter Laufzeitfehler

Dies führt nun zu dem Fehler aus Bild 2. Die Variable enthält also nicht mehr den Verweis auf das **Database**-Objekt.

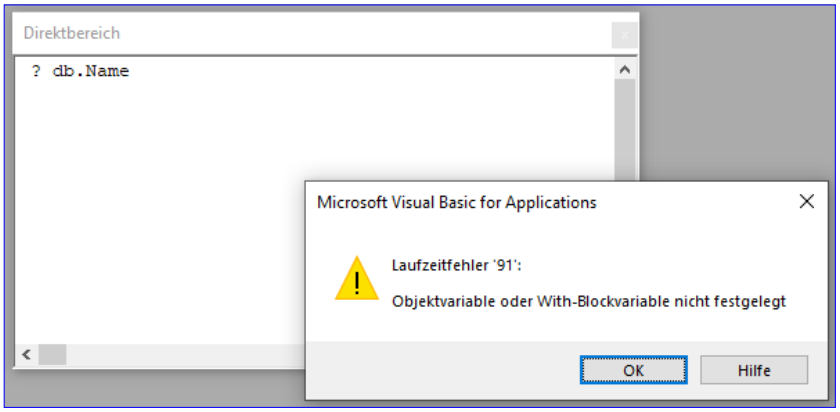


Bild 2: Die Variable **db** ist geleert.

Wozu globale Ribbonvariablen?

Doch zurück zur eigentlichen Problemstellung, die solche Variablen betrifft, die global deklariert sind und Verweise auf Ribbon-Definitionen speichern.

Warum machen wir das überhaupt? Nun: Der Verweis auf eine Ribbon-Definition wird nur einmal beim Anwenden der Ribbon-Definition geliefert, und zwar mit der Prozedur, die Sie für das Ereignisattribut **onLoad** hinterlegen können. Diese enthält einen Parameter namens **ribbon** mit dem Datentyp **IRibbonUI**.

Damit wir später auf diese Variable zugreifen können, müssen wir diese als globale Variable speichern. Dazu verwenden wir die folgende Variable, die wir in einem Standardmodul beispielsweise namens **mdlRibbons** deklarieren:

```
Public objRibbon_Main As IRibbonUI
```

In der Ribbon-Definition legen wir für das Element **CustomUI** ein Attribut namens **onLoad** an, das den Namen der Prozedur enthält, die beim Laden der Ribbondefinition ausgeführt werden soll – außerdem hinterlegen wir für das **button**-Element **btn** noch ein Callbackattribut namens **getEnabled**, mit dem wir gleich prüfen können, ob die **Invalidate**-Methode des Ribbons funktioniert:

```
<?xml version="1.0"?>
<customUI ... onLoad="OnLoad_Main">
  <ribbon>
```

```
    <tabs>
      <tab id="tab" label="Beispieltab">
        <group id="grp" label="Beispielgruppe">
          <button id="btn" getEnabled="getEnabled"
            label="Beispielbutton"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Die Prozedur für das Attribut **onLoad** enthält lediglich eine Anweisung, die den mit dem Parameter **ribbon** übergebenen Verweis in der Variablen **objRibbon_Main** speichert:

```
Sub onLoad_Main(ribbon As IRibbonUI)
    Set objRibbon_Main = ribbon
End Sub
```

Außerdem hinterlegen wir im Modul **mdlRibbons** noch eine Variable für den **Enabled**-Zustand der Schaltfläche **btn**:

```
Public bolEnabled As Boolean
```

Für das Attribut **getEnabled** des **button**-Elements **btn** hinterlegen wir die folgende Prozedur, welche mit dem Parameter **enabled** den Wert der Variablen **bolEnabled** zurückgibt:

```
Sub getEnabled(control As IRibbon-
Control, ByRef enabled)
    enabled = bolEnabled
End Sub
```

Testen, ob die Ribbon-Variable bei Fehler geleert wird

Wenn Sie die Ribbon-Definition wie hier beschrieben und in der Beispielanwendung umgesetzt angelegt haben, sollte nach einem Neustart der Anwendung das Tab mit der Beschriftung **Beispieltab** mit einer Gruppe **Beispielgruppe** und einer Schaltfläche **Beispielbutton** erscheinen (siehe Bild 3).

Die Schaltfläche ist nicht aktiviert, da die Variable **bolEnabled** standardmäßig den Wert **False** hat und noch nicht auf **True** eingestellt wurde.

Also liefert die Prozedur **getEnabled** den Wert **False** für den Parameter **enabled**, der dann an das aufrufende **button**-Element weitergegeben wird.

Nun probieren wir, ob auch das Zuweisen des **IRibbonUI**-Objekts funktioniert hat. Dazu rufen wir die folgende Prozedur auf. Diese stellt zunächst **bolEnabled** auf **True** ein. Beim nächsten Aufruf von **getEnabled** sollte also die Schaltfläche **btn** aktiviert werden. Außerdem ruft die Prozedur die **Invalidate**-Methode von **objRibbon_Main** auf:

```
Public Sub BeispielbuttonAktivieren()
    bolEnabled = True
    objRibbon_Main.Invalidate
End Sub
```

Dies führt dazu, dass die Schaltfläche **btn** nun aktiviert dargestellt wird (siehe Bild 4). Mit der folgenden Prozedur können Sie diese wieder deaktivieren:

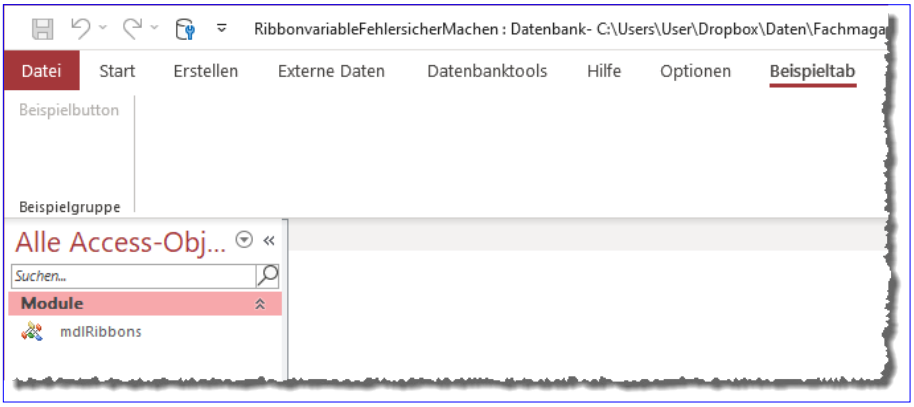


Bild 3: Das Beispielribbon in Aktion

Bei Tab-Wechsel im Ribbon ein Formular anzeigen

Microsoft verwöhnt den Office-Entwickler nicht gerade mit Ereignissen im Ribbon. Es gibt ein bis zwei Ereignisattribute für einige Steuerelemente, eines, das beim Anzeigen einer Ribbondefinition ausgeführt wird – und das war es schon fast. Was aber, wenn Sie andere Ereignisse benötigen? Zum Beispiel, um beim Wechsel von tab-Elementen ein spezielles Formular für das jeweilige tab-Element einzublenden? Hierfür gibt es kein eingebautes Ereignis. Also müssen wir ein wenig improvisieren. Wie das gelingt, zeigen wir in diesem Beitrag!

Ausgangsfrage

Dieser Beitrag ist eine Antwort auf eine Leseranfrage: Der Leser wünschte sich, dass beim Anklicken eines Tabs im Ribbon immer direkt ein passendes Formular geöffnet wird.

Wir haben daraus folgende Beispielanforderung abgeleitet:

Unser Ribbon hat drei Tabs namens **tab1**, **tab2** und **tab3**. Dazu enthält die Lösung drei Formulare namens **frm1**, **frm2** und **frm3**. Direkt beim Öffnen der Anwendung wird das tab-Element **tab1** angezeigt und damit auch das Formular **frm1** (siehe Bild 1).

Wenn der Benutzer auf das **tab**-Element **tab2** klickt, soll sich direkt das Formular **frm2** öffnen (siehe Bild 2), klickt er auf **tab3**, soll das Formular **frm3** erscheinen.

Ist eines der Formulare bereits geöffnet und der Benutzer klickt auf ein **tab**-Element, das gerade nicht aktiviert ist, dann soll das entsprechende Formular aktiviert werden.

Problem: kein passendes Ereignisattribut

Wenn wir uns nun die Attribute des **tab**-Elements ansehen, stellen wir fest, dass dieses kein Attribut anbietet, für das wir beispielsweise beim Anzeigen oder Aktivieren eines **tab**-Elements ein Ereignis auslösen können. Generell bietet das Ribbon quasi keine Ereignisse, die durch

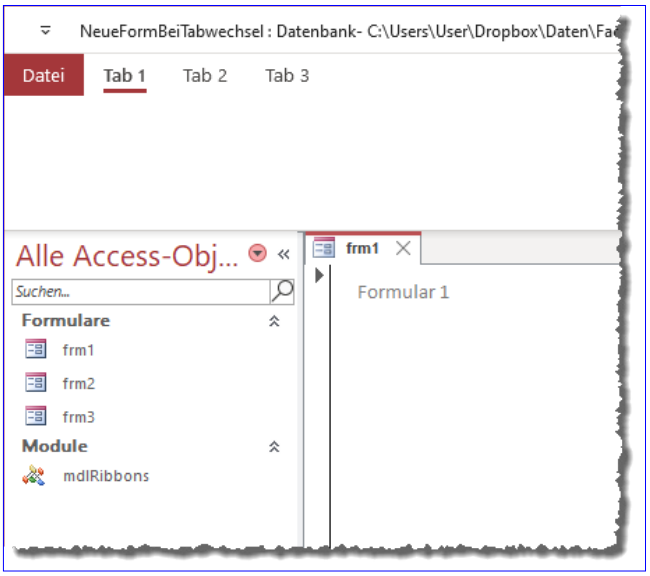


Bild 1: Anzeige des ersten Formulars gleich beim Öffnen

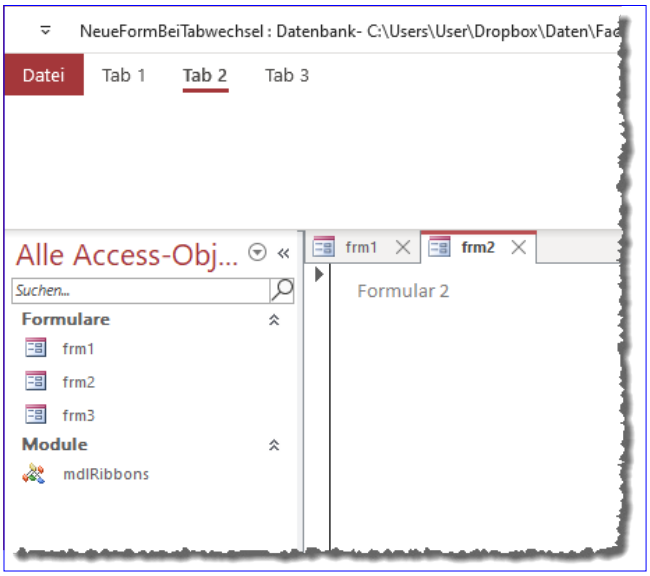


Bild 2: Anzeige des zweiten Formulars beim Klick auf das zweite Tab

```
<?xml version="1.0"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" onLoad="OnLoad_Main">
  <ribbon startFromScratch="true">
    <tabs>
      <tab id="tab1" tag="1" label="Tab 1">
        <group id="grp1" label="Beispielgruppe 1">
          <button label="Beispielschaltfläche 1" getVisible="getVisible" id="btn1" tag="frm1"/>
        </group>
      </tab>
      <tab id="tab2" tag="2" label="Tab 2">
        <group id="grp2" label="Beispielgruppe 2">
          <button label="Beispielschaltfläche 2" getVisible="getVisible" id="btn2" tag="frm2"/>
        </group>
      </tab>
      <tab id="tab3" tag="3" label="Tab 3">
        <group id="grp3" label="Beispielgruppe 3">
          <button label="Beispielschaltfläche 3" getVisible="getVisible" id="btn3" tag="frm3"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Listing 1: Definition des Ribbons zum automatischen Öffnen von Formularen

das Anklicken oder Wechseln eines **tab**-Elements ausgelöst werden. Das **tab**-Element bietet nur die folgenden **get...**-Callbackattribute:

- **getKeytip**
- **getLabel**
- **getVisible**

Diese Attribute erwarten Prozeduren, welche die Werte für die Attribute **Keytip**, **Label** und **Visible** des **tab**-Elements liefern.

Diese Prozeduren werden dann aufgerufen, wenn die **tab**-Elemente erstmalig angezeigt werden oder nachdem die **Invalidate**-Methode für die Ribbondefinition oder die **InvalidateControl**-Methode für ein einzelnes Ribbonelement aufgerufen wurde.

Das **customUI**-Element bietet ein Ereignisattribut namens **onLoad** an, aber das wird nur einmalig beim Laden der Ribbon-Definition ausgelöst.

Das hilft alles nur wenig weiter – wir benötigen ja schließlich ein Ereignis, das immer beim Aktivieren eines **tab**-Elements ausgelöst wird, damit wir dann das jeweilige Formular anzeigen können.

Kombiniere, kombiniere ...!

Da die **get...**-Callbackattribute immer beim ersten Anzeigen ausgelöst werden (oder beim erneuten Anzeigen, wenn zwischendurch die **Invalidate**- oder die **InvalidateControl**-Methode aufgerufen wurde), könnten wir sie eigentlich auch nutzen! Wir müssen hier nur den Befehl zum Öffnen des jeweiligen Formulars unterbringen und dafür sorgen, dass die **tab**-Elemente immer nach dem Auswählen eines anderen **tab**-Elements wieder »invalidiert« werden.

Formulare per VBA erstellen

Warum sollte man Formulare per VBA erstellen, wenn Microsoft Access doch die gute alte, etwas in die Jahre gekommene Entwurfsansicht dafür bereitstellt? Ganz einfach: Weil es für den effizient arbeitenden Access-Entwickler immer wieder Aufgaben gibt, die er einfach nicht von Hand erledigen möchte. Oder weil der Access-Entwickler immer wiederkehrende Aufgaben in ein Access-Add-In oder ein COM-Add-In auslagern möchte. Und dort gibt es nun einmal keine Entwurfsansicht – dort ist VBA-Code gefragt, um neue Formulare zu erstellen und die gewünschten Steuerelemente auf das Formular zu bringen. Dieser Beitrag liefert alle Techniken, die zum Erstellen von Formularen und zum Ausstatten mit Steuerelementen notwendig sind.

Neues, leeres Formular erstellen

Mit einer ersten Funktion wollen wir ein neues, leeres Formular mit einem vorher festgelegten Namen erstellen. Das erledigt die Funktion **CreateNewForm**. Diese erwartet als Parameter den Namen, den das Formular nach dem Erstellen erhalten soll:

```
Public Sub CreateNewForm(strName As String)
    Dim frm As Form
    Dim strNameTemp As String
    Set frm = CreateForm()
    strNameTemp = frm.Name
    DoCmd.Close acForm, frm.Name, acSaveYes
    DoCmd.Rename strName, acForm, strNameTemp
End Sub
```

Die Prozedur deklariert Variablen für das neue **Form**-Element (**frm**) und für den temporären von Access beim Erstellen vergebenen Namen (**strNameTemp**). Dann erstellt es mit der **CreateForm**-Funktion ein neues Formular. **CreateForm** liefert einen Verweis auf das neu erstellte **Form**-Objekt zurück, welches wir mit der Variablen **frm** referenzieren. Wir würden nun gern den Namen des Formulars einstellen, aber das ist nicht ohne weiteres möglich: Die Eigenschaft **Name** des **Form**-Objekts ist nämlich schreibgeschützt. Wie also dem Formular den gewünschten Namen geben? Klar: Wir könnten es einfach nach dem Speichern und Schließen umbenennen, dafür gibt es die **DoCmd.Rename**-Methode. Allerdings erwartet die auch den vorherigen Namen des Formulars, und den kennen wir nicht. Jedoch können wir diesen zuvor mit **frm.Name** auslesen und speichern ihn in der Variablen **strNameTemp**. Danach schließen wir dann das Formular mit der Methode **DoCmd.Close** und den Parametern **acForm** für den Objekttyp, **frm.Name** für den Namen des zu schließenden Objekts und **acSaveYes**, damit die Änderungen an dem neu erstellten Formular ohne Rückfrage gespeichert werden.

Nach dem Schließen rufen wir dann **DoCmd.Rename** auf und übergeben mit dem ersten Parameter den neuen Namen, mit dem zweiten den Objekttyp (**acForm**) und mit dem dritten den vorherigen Objektnamen.

Rufen wir diese Prozedur nun wie folgt auf, legt dies ein neues, leeres Formular unter dem angegebenen Namen an, das direkt im Navigationsbereich erscheint:

```
CreateNewForm "frmNeuesFormular"
```

Formular bereits vorhanden?

Wenn man diese Prozedur zwei Mal hintereinander mit dem gleichen Formularnamen aufruft, erhält man die Meldung aus Bild 1. Diese wird durch den Versuch ausgelöst, dem neuen Formular den Namen eines bereits vorhandenen Formulars zu übergeben. Dem können wir

bereits zuvor vorbeugen, indem wir abfragen, ob bereits ein Formular dieses Namens vorhanden ist. Ob ein Formular bereits vorhanden ist, können wir auf verschiedene Arten prüfen. Wir könnten zum Beispiel versuchen, es zu öffnen, oder wir durchlaufen eine der Auflistungen der Formulare.

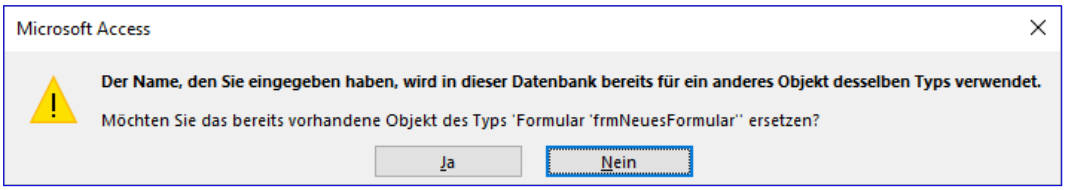


Bild 1: Meldung beim Versuch, einem Formular einen bereits vorhandenen Namen zu geben

Wir wählen letztere Variante. Die Funktion heißt **ExistsForm** und erwartet den Formularnamen als Parameter. Sie liefert das Ergebnis als Booleanwert zurück. Hier deklarieren wir für die Formulare eine Variable des Typs **AccessObject**. Warum nun **AccessObject** und nicht **Form** wie oben in der Prozedur **CreateNewForm**? Weil die dort verwendete Funktion **CreateForm** ein Objekt des Typs **Form** erstellt, die Auflistung **CurrentProject.AllForms**, die wir durchsuchen wollen, jedoch Elemente des Typs **AccessObjects** liefert.

Mit **objForm** durchlaufen wir in einer **For Each**-Schleife alle Elemente der Auflistung **CurrentProject.AllForms** und prüfen, ob das aktuell mit **objForm** referenzierte Formular den mit **strForm** übergebenen Namen hat. Falls ja, stellen wir den Funktionswert auf **True** ein und verlassen die Funktion. Wenn die Funktion alle Elemente aus **CurrentProject.AllForms** durchläuft, ohne dass das passende Formular gefunden wird, liefert sie den Rückgabewert **False** (also den Standardwert für Rückgabewerte des Datentyps **Boolean**):

```
Public Function ExistsForm(strForm As String) As Boolean
    Dim objForm As AccessObject
    For Each objForm In CurrentProject.AllForms
        If objForm.Name = strForm Then
            ExistsForm = True
            Exit Function
        End If
    Next objForm
End Function
```

Formular neu erstellen?

Wenn das Formular bereits existiert, können wir den Benutzer fragen, ob das vorhandene Formular überschrieben werden soll. Ist das der Fall, löschen wir das Formular und erstellen es neu. Dazu erweitern wir die Prozedur **CreateNewForm** wie in Listing 1. Hier fügen wir direkt hinter der Deklaration der Variablen einen Aufruf der Funktion **ExistsForm** ein.

Liefert diese den Wert **True**, dann zeigen wir ein Meldungsfenster an, das den Benutzer fragt, ob er das Formular überschreiben möchte. Falls ja, rufen wir eine Funktion namens **DeleteForm** auf (siehe weiter unten) und übergeben den Namen des zu löschenden Formulars. Diese könnte fehlschlagen, weil beispielsweise das zu löschende Formular gerade geöffnet ist. In diesem Fall soll die Prozedur einfach verlassen werden – die passende Fehlermeldung soll die Funktion **DeleteForm** liefern. Wenn der Benutzer die Frage, ob das vorhandene Formular gelöscht werden soll, mit **Nein** antwortet, soll die Prozedur auch verlassen werden.

Nur wenn das Formular noch nicht existiert oder gelöscht werden konnte, werden die anschließenden Schritte zum Erstellen des neuen Formulars ausgeführt.

Löschen eines vorhandenen Formulars

Wenn das Formular bereits vorhanden ist und der Benutzer es überschreiben möchte, müssen wir es löschen. Dazu haben wir die Funktion **DeleteForm** definiert, die den Namen des zu löschenden Formulars entgegennimmt und einen **Boolean**-Wert zurückliefert.

Die Funktion probiert bei deaktivierter eingebauter Fehlerbehandlung aus, ob das Formular mit **DoCmd.DeleteOb-**

```
Public Function CreateNewForm(strName As String) As Boolean
    Dim frm As Form
    Dim strNameTemp As String
    If ExistsForm(strName) Then
        If MsgBox("Formular '" & strName & "' ist bereits vorhanden. Überschreiben?", vbYesNo + vbExclamation, _
                "Formular überschreiben?") = vbYes Then
            If DeleteForm(strName) = False Then
                Exit Function
            End If
        Else
            Exit Function
        End If
    End If
    Set frm = CreateForm()
    strNameTemp = frm.Name
    frm.Visible = True
    DoCmd.Close acForm, frm.Name, acSaveYes
    DoCmd.Rename strName, acForm, strNameTemp
     CreateNewForm = True
End Function
```

Listing 1: Erstellen eines neuen Formulars

ject gelöscht werden kann. Falls ja, erhält die Funktion den Rückgabewert **True**. Anderenfalls gibt die Funktion die entsprechende Fehlermeldung aus **Err.Description** aus und gibt den Standardwert **False** als Funktionsergebnis an die aufrufende Prozedur zurück:

```
Public Function DeleteForm(strName As String) As Boolean
    On Error Resume Next
    DoCmd.DeleteObject acForm, strName
    If Err.Number = 0 Then
        DeleteForm = True
    Else
        MsgBox Err.Description, vbCritical + vbOKOnly, _
            "Löschen fehlgeschlagen"
    End If
End Function
```

Damit haben wir bereits eine Funktion zum Erstellen eines neuen Formulars, das allerdings noch komplett leer ist und für das wir auch noch keine Eigenschaften eingestellt haben.

Formular weiterverarbeiten

Nun stehen zwei Aufgaben an: Die erste ist das Einstellen verschiedener Eigenschaften für das Formular. Die zweite ist das Erstellen von Steuerelementen im Formular. Letzteres beschreibt der bald erscheinende Beitrag **Steuerelemente erstellen** (**www.access-im-unternehmen.de/1336**). Um die Eigenschaften kümmern wir uns jetzt gleich.

Um Eigenschaften einstellen zu können, benötigen wir einen Verweis auf das Formular in der Entwurfsansicht. Das heißt also, dass wir das Formular auch öffnen müssen. Da stellt sich die Frage: Könnten wir mit der Funktion **CreateNewForm** nicht direkt einen Verweis auf das neu erstellte und in der Formularansicht geöffnete Formular zurückgeben? Das könnten wir tun, wir müssten dazu das Formular allerdings auch in dieser Funktion wieder öffnen.

Das Schließen ist dort unbedingt nötig, da wir es sonst nicht umbenennen können. Da wir nicht wissen, ob und wie das Formular nach dem Erstellen direkt weiterverarbeitet werden soll, geben wir einfach nur einen Boole-

an-Wert zurück, der angibt, ob das Formular erfolgreich erstellt wurde.

Eigenschaften des Formulars einstellen

In der Prozedur, welche die Funktion **CreateNewForm** aufgerufen hat, können wir nun das Formular in der Entwurfsansicht öffnen und die gewünschten Änderungen vornehmen.

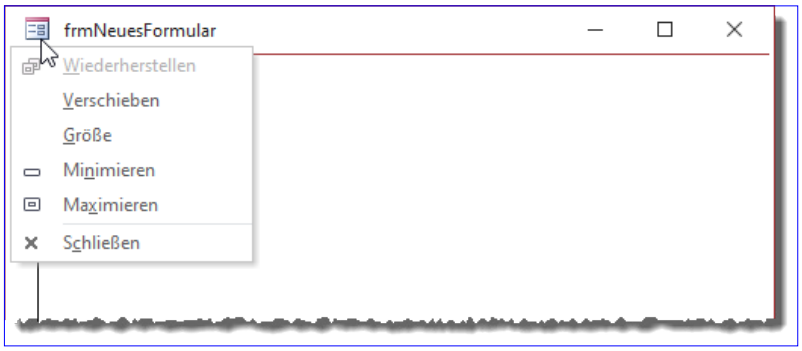


Bild 2: Elemente der Titelleiste eines Formulars

Wie bereits erwähnt, wollen wir uns in diesem Beitrag zunächst um das Einstellen der verschiedenen Eigenschaften des Formulars kümmern und schauen uns diese dabei im Detail an. Das Öffnen gestalten wir wie folgt:

```
Public Sub Test_CreateNewForm()
    Dim frm As Form
    Dim strForm As String
    strForm = "frmNeuesFormular"
    If CreateNewForm(strForm) = True Then
        DoCmd.OpenForm strForm, acDesign
        Set frm = Forms(strForm)
        With frm
            'Eigenschaften einstellen
        End With
    End If
End Sub
```

Hier schreiben wir den Namen des zu erstellenden Formulars direkt in eine Variable, weil wir diesen mehr als einmal benötigen und anschließende Änderungen so nur an einer Stelle erfolgen müssen. Dann erstellen wir das neue Formular mit der Funktion **CreateNewForm**.

Ist dies erfolgreich, öffnen wir das neu erstellte Formular mit **DoCmd.OpenForm** in der Entwurfsansicht. Dazu stellen wir den zweiten Parameter **View** auf **acDesign** ein.

Damit wir danach komfortabel die Eigenschaften des Formulars einstellen können, referenzieren wir das Formular dann über die **Forms**-Auflistung und den Namen des Formulars mit der **Form**-Variablen **frm**. Für **frm** stellen wir dann die nachfolgend beschriebenen Eigenschaften ein.

Einstellungen für die Titelleiste

Die Titelleiste des Formulars (siehe Bild 2) verwendet gleich mehrere Eigenschaften.

- Die Titelleiste des Formulars stellen wir mit der Eigenschaft **Caption** ein.
- Ob das Formular das gleiche Icon anzeigen soll wie die Hauptanwendung, legen Sie übrigens nicht mit einer Formular-Eigenschaft fest, sondern in den Access-Optionen mit der Option **Als Formular- und Berichtssymbol verwenden**. Dies gilt dann für alle Formulare und Berichte.
- Ob die **Schließen**-Schaltfläche aktiviert ist, stellen Sie mit der Eigenschaft **CloseButton** ein. Legen Sie den Wert auf **False** fest, wird die Schaltfläche ausgegraut dargestellt und der Benutzer kann sie nicht betätigen.
- Die Schaltflächen in der Titelleiste zum Minimieren und Maximieren des Fensters können Sie über die Benutzeroberfläche mit der Eigenschaft **MinMaxSchaltflächen** einstellen, über VBA mit **MinMaxButtons**. Die möglichen Werte sind: **0** (**Keine**), **1** (**Min vorhanden**), **2** (**Max vorhanden**) oder **3** (**Beide vorhanden**).

- Die Eigenschaft **Mit Systemfeldmenü** (**ControlBox**) stellt ein, ob überhaupt Steuerelemente in der Titelleiste angezeigt werden sollen. Falls nicht, sieht die Titelleiste wie in Bild 3 aus.

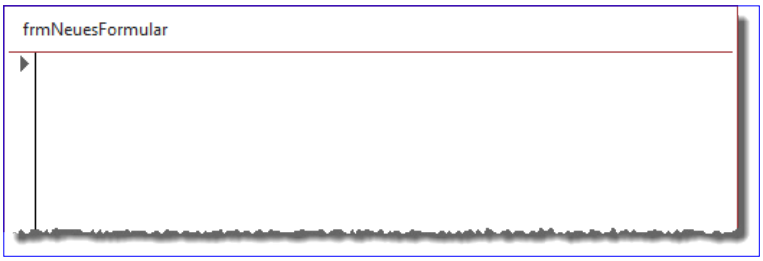


Bild 3: Titelleiste nur mit Überschrift

Abmessungen

Die Abmessungen des Formulars wie die Höhe und die Breite stellen Sie mit Eigenschaften für verschiedene Elemente ein. Die **Breite** ist eine Formular-Eigenschaft, diese entspricht der VBA-Eigenschaft **Width**. Eine Eigenschaft namens **Höhe** beziehungsweise **Height** vermissen wir allerdings im Eigenschaftenblatt. Der Grund ist einfach: Ein Formular kann mehrere Bereiche enthalten, die jeweils eine eigene Höhe haben – zum Beispiel der Detailbereich oder Formularkopf und -Fuß. Die Einstellungen dieser Bereiche besprechen wir weiter unten.

Standardansicht

Die Standardansicht stellt die Eigenschaft **DefaultView** ein. Diese nimmt die Werte der Enumeration **acDefView** entgegen:

- **acDefViewContinuous** (**1**): Endlosformular
- **acDefViewDatasheet** (**2**): Datenblatt
- **acDefViewSingle** (**0**): Einzelnes Formular
- **acDefViewSplitForm** (**5**): Geteiltes Formular

Beispiel zum Einstellen der Eigenschaft **Standardansicht** auf **Datenblatt**:

```
frm.DefaultView = acViewDatasheet
```

Erlaubte Ansichten einstellen

Wenn Sie verhindern wollen, dass die Benutzer zwischen verschiedenen Ansichten wechseln, nutzen Sie die Eigenschaft **ViewsAllowed** dazu.

Mögliche Werte:

- **0**: Formularansicht und Datenblattansicht möglich
- **1**: Kein Wechsel von der Formularansicht zur Datenblattansicht möglich
- **2**: Kein Wechsel von der Datenblattansicht zur Formularansicht möglich

Sichtbarkeit

Mit der Eigenschaft **Sichtbar** (**Visible**) können Sie das Formular ein- und ausblenden.

Eigenschaften für geteilte Formulare

Wenn das Formular über die Eigenschaft **DefaultView** als geteiltes Formular angezeigt wird, werden die folgenden Eigenschaften ausgewertet:

- **Größe des geteilten Formulars** (**SplitFormSize**) gibt je nach der mit **SplitFormOrientation** gewählten Position die Höhe oder Breite des Detailbereichs des geteilten Formulars an. Diese wird im Eigenschaftenblatt in Zentimeter und unter VBA in Twips angegeben.
- **Ausrichtung des geteilten Formulars** (**SplitFormOrientation**): Gibt an, wo das Datenblatt angezeigt werden soll. Es gibt die folgenden Werte: **acDatasheetOnBottom** (**1**), **acDatasheetOnLeft** (**2**), **acDatasheetOnRight** (**3**) und **acDatasheetOnTop** (**0**).
- **Teilerleiste des geteilten Formulars** (**SplitFormSplitterBar**): Gibt an, ob die Leiste zwischen den beiden

Formularbereichen angezeigt werden soll. Wenn der Benutzer die Höhe/Breite der Bereiche nicht verändern soll, stellen Sie diese Eigenschaft auf **False** ein.

- **Datenblatt des geteilten Formulars** (**SplitFormDatasheet**): Hier können Sie festlegen, ob Bearbeitungen im Datenblatt zulässig sein sollen. Es gibt die beiden Werte **Bearbeitungen zulassen** (**acDatasheetAllowEdits**) oder **Schreibgeschützt** (**acDatasheetReadOnly**.).

- **Drucken des geteilten Formulars** (**SplitFormPrinting**): Hiermit legen Sie fest, welcher Bereich des geteilten Formulars im Fall der Fälle gedruckt werden soll. Die Eigenschaft nimmt die Werte **Nur Datenblatt** (**acGridOnly**) oder **Nur Formular** (**acFormOnly**) entgegen.

- **Position der Teilerleiste speichern** (**SaveSplitterBarPosition**): Legt fest, ob die Position der Teilerleiste gespeichert werden soll, wenn diese beim Bearbeiten des geöffneten Formulars mit der Maus verschoben wurde. Beim Schließen wird dann abgefragt, ob die Position so wie geändert erhalten werden soll.

Einstellungen für die Anzeige des Formulars

Die folgenden Einstellungen blenden verschiedene Formularelemente ein oder aus, außerdem enthalten sie Informationen über den Zustand zu modalen und Popup-Fenstern.

Die nachfolgend beschriebenen Elemente finden Sie teilweise im Screenshot aus Bild 4.

- **Automatisch zentrieren** (**AutoCenter**): Legt fest, ob das Formular beim Öffnen automatisch im Access-Fenster zentriert werden soll.

- **Datensatzmarkierer** (**RecordSelectors**): Aktiviert die Anzeige des Datensatzmarkierers (siehe Screenshot unter 1)

- **Navigationsschaltflächen** (**NavigationButtons**): Aktiviert die Anzeige der Navigationsschaltflächen (siehe Screenshot unter 3).

- **Navigationsbeschriftung** (**NavigationCaption**): Stellt die Beschriftung der Navigationsschaltflächen ein. Diese Beschriftung wird dort angezeigt, wo normalerweise **Datensatz:** steht (siehe Screenshot unter 2).

- **Trennlinien** (**DividingLines**): Legt fest, ob im Endlosformular Trennlinien zwischen den Datensätzen angezeigt werden.

- **Bildlaufleisten** (**ScrollBars**): Stellt ein, ob Bildlaufleisten angezeigt werden sollen. Es gibt die folgenden Einstellungen: **0**: **Nein**, **1**: **Nur horizontal**, **2**: **Nur vertikal** und **3**: **In beide Richtungen** (siehe Screenshot unter 4)

- **Rahmenart** (**BorderStyle**): Gibt an, welche Rahmenart für Titelzeile, Formularrahmen et cetera verwendet werden soll. Es gibt die Werte **0** (**Keine**), **1** (**Extra dünn**), **2** (**Veränderbar**) und **3** (**Dialog**).

Außerdem gibt es noch zwei Eigenschaften, welche beide auf **True** eingestellt werden, wenn Sie das Formular mit **DoCmd.OpenForm** mit dem Wert **acDialog** für den Parameter **WindowMode** öffnen:

```
DoCmd.OpenForm "frmBeispiel", WindowMode:=acDialog
```

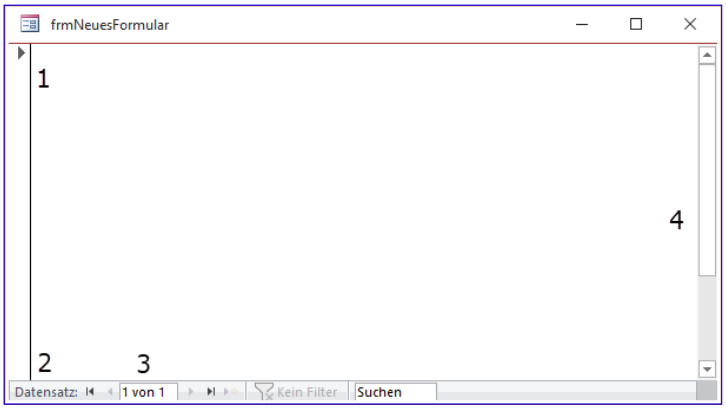


Bild 4: Ein- und ausblendbare Elemente im Formular

Setup für Access: Vertrauenswürdige Speicherorte

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

Das in diesem Artikel beschriebene Konzept hat das Ziel, die Registrierung des Installationsverzeichnisses als vertrauenswürdigen Speicherort zu automatisieren. Dadurch entfällt die Notwendigkeit für den Anwender, dies in Access selbst einzutragen. Vor allem vermeiden wir durch die spezifische Festlegung nur eines Verzeichnisses als vertrauenswürdig das Risiko, dass der Anwender unnötig viele Unterverzeichnisse quasi nebenher als vertrauenswürdig einstuft (was sie vielleicht nicht sein sollten).

Automatisierung des Eintrags für die vertrauenswürdigen Speicherorte

Das Konzept der "vertrauenswürdigen Speicherorte" ist seit längerem für (meines Wissens) alle Office-Programme von Microsoft gängig. Wie schon im vorherigen Artikel erläutert, möchte Microsoft damit verhindern, dass irgendeine Datei mit VBA-Code auf unsere Festplatte gelangt und ohne weitere Kontrolle ausgeführt wird.

Ich habe von Leuten gehört, die der Einfachheit halber **C:** und alle Unterverzeichnisse als vertrauenswürdig einstufen, aber das scheint mir ein scheunentorgroßes Loch in die Sicherheit unseres Rechners zu schlagen. Jedes Verzeichnis, auch das Standardverzeichnis für temporäre Dateien, würde es dann erlauben, dass VBA-Makros ausgeführt werden! Das kann nicht in unserem Sinne sein.

Bei Word und Excel zum Beispiel gibt es einen Trick: die digitale Signatur von VBA-Makros. Diese sichert die VBA-Makros der betreffenden Datei einerseits gegen Veränderungen ab, andererseits macht sie den Autor des Makros überprüfbar.

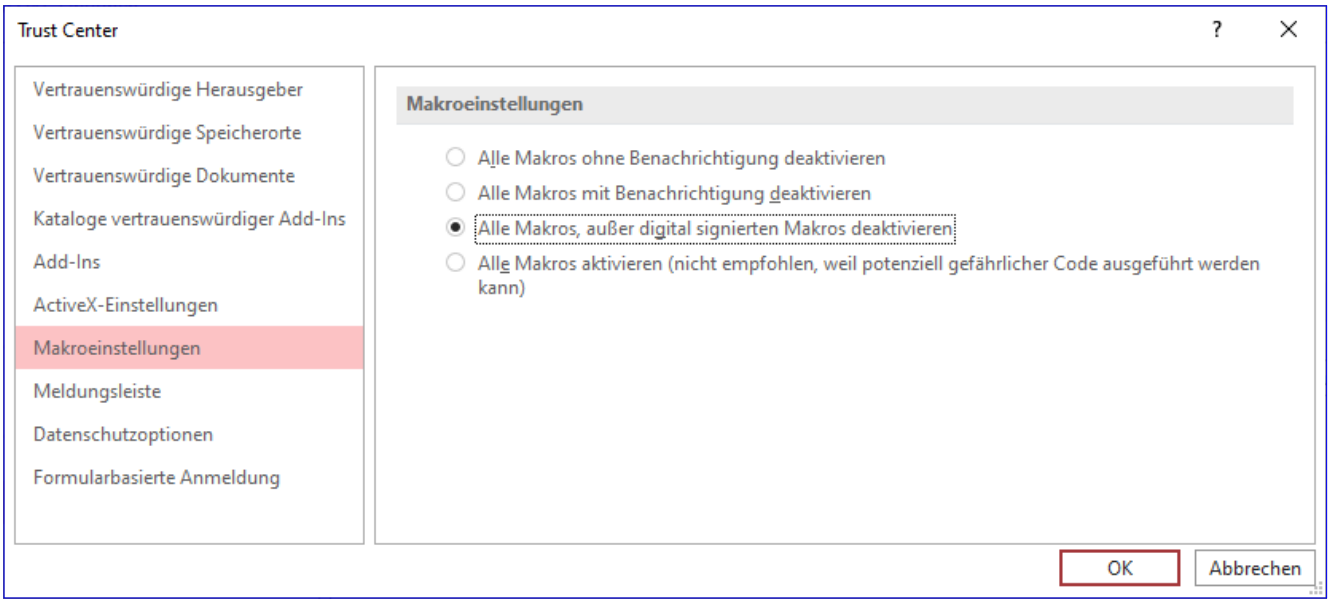


Bild 1: Trustcenter-Einstellung für VBA-Makros

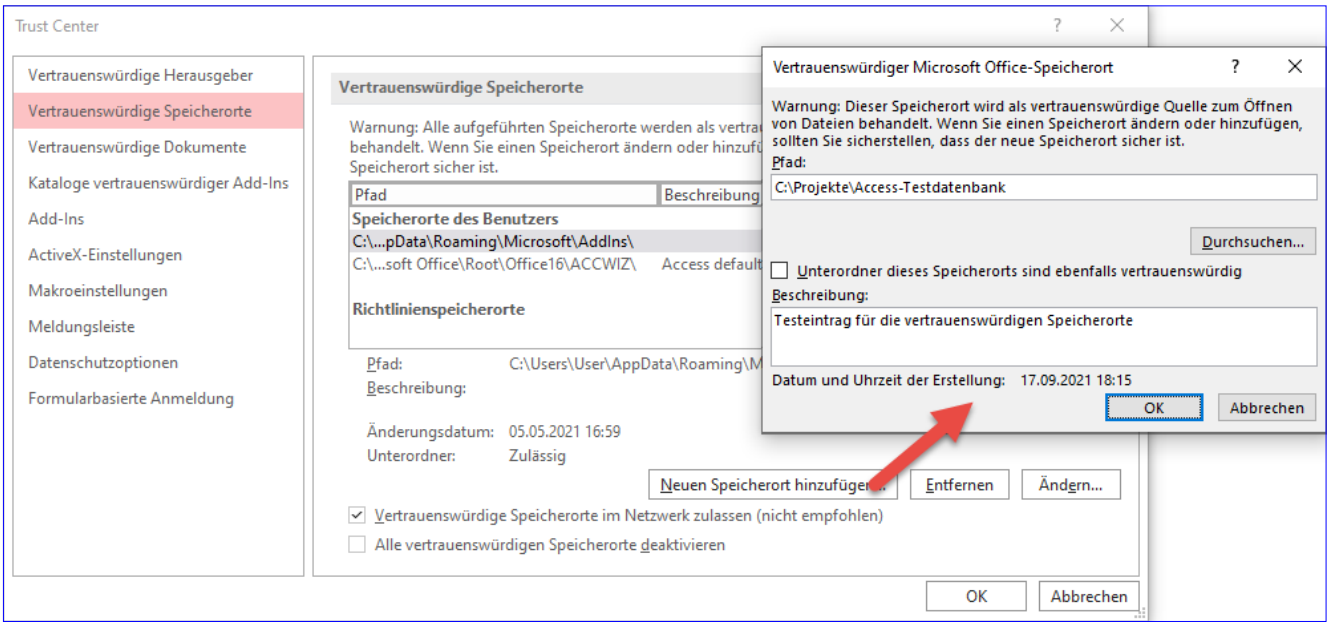


Bild 2: Eingabeformular für einen vertrauenswürdigen Speicherort

In Verbindung mit der Einstellung **Alle Makros außer digital signierten deaktivieren** (siehe Bild 1) erhält man so die Möglichkeit, die Makros aus dem eigenen Haus aktiv zu lassen, alle anderen jedoch zu blockieren.

Leider funktioniert die Verwendung einer digitalen Signatur für die VBA-Makros in Access nicht. Deshalb sind wir auf die korrekte Konfiguration der vertrauenswürdigen Speicherorte angewiesen.

Einen vertrauenswürdigen Speicherort eintragen

Schauen wir uns das zunächst bei den Einstellungen von Access an. Das ist ein wichtiger Aspekt, denn diese Einstellung kann und muss für jedes Programm und jeden Windows-Useraccount individuell vorgenommen werden.

Öffnen Sie in Access **Optionen|Trust Center|Einstellungen** für das Trustcenter. Dort finden Sie den Eintrag **Vertrauenswürdige Speicherorte**. Falls dort bei Ihnen noch nichts steht, tragen Sie ruhig mal ein Verzeichnis ein, wie in Bild 2 gezeigt. Der Haken für die Unterordner muss nicht gesetzt werden, da diese Einstellung nur unsere **.accdb**-Datei betrifft.

Unsere Eingabe wird in der Registry gespeichert, sobald wir die Dialoge mit **OK** bestätigt haben. Dies erfolgt unter dem Pfad **HKEY_CURRENT_USER\SOFTWARE\Microsoft\Office\16.0\Access\Security\Trusted Locations** (siehe Bild 3).

Die einzelnen Einträge erhalten dabei automatisch Namen wie hier im Beispiel **Location1**. Sobald ein solcher Eintrag von uns gelöscht wird, wird er auch aus der Registry entfernt.

Damit haben wir eine Möglichkeit gefunden, den vertrauenswürdigen Speicherort für unsere eigene Applikation automatisiert einzutragen. Jetzt geht es nur noch darum, das richtige Format zu finden.

Wie man oben sieht, enthält der Registry-Pfad unter anderem die interne Access-Versionsnummer. Da ich diesen Artikel mit Access 2019 vorbereitet habe, steht dort **16.0**.

Diese Versionsnummer müssen wir während unserer Setup-Ausführung ermitteln und korrekt verwenden.

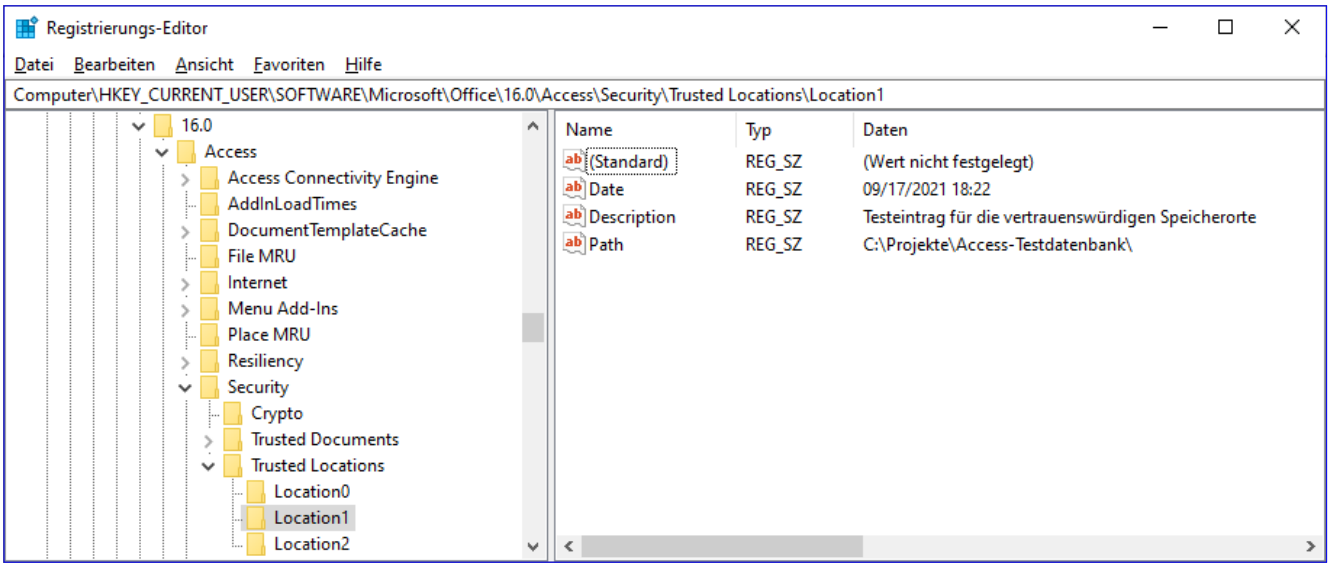


Bild 3: Registry-Eintrag eines vertrauenswürdigen Speicherorts

Die zweite Aufgabe betrifft die erkennbare Durchnummerierung der **Location**-Einträge. Doch hier gibt es eine ganz einfache Lösung: Das muss gar nicht sein. Wenn ich probeweise mal einen Eintrag umbenenne, funktioniert alles weiterhin wie gewohnt. Das heißt, dass wir zum Beispiel unsere Konstante **{#MyAppName}** dafür hernehmen können.

Pascal-Scripting in InnoSetup

Es geht bei InnoSetup schon vieles über die einfachen Einstellungen, aber irgendwann ist halt eine Grenze erreicht. In den Fällen hat InnoSetup ein sehr mächtiges Werkzeug an Bord, auf das wir nun zurückgreifen müssen: Pascal-Scripting.

Was bei Microsoft Office das VBA ist, das ist im Prinzip das Pascal-Scripting für InnoSetup. Es handelt sich im Grunde um ein Konzept, das wir von Access her auch schon kennen.

Denn hier geht es ebenfalls nicht um ein großes Programm, welches das ganze Setup steuert, sondern rein darum, auf bestimmte Ereignisse zu reagieren. Das kann zum Beispiel der Start des Setups sein oder auch der Zugriff auf bestimmte Werte für den Eintrag in die Registry.

Eingeleitet wird das Scripting durch eine Rubrikenüberschrift **[Code]**, danach wird alles anders. Ab hier gelten die Regeln einer Programmiersprache, die rein zeilenorientierte Schreibweise des bisherigen Scripts gilt nicht mehr. Aus diesem Grunde muss die Code-Rubrik auch immer die letzte im gesamten Script sein!

Welche Funktionen benötigen wir? Da ist zunächst, wie gesagt, die aktuelle Access-Version. Diese Funktion nenne ich **CurAccVer**. Des weiteren finden wir in den Daten in der Registry noch das aktuelle Datum in amerikanischer Schreibweise.

Das wäre zwar ebenfalls nicht besonders notwendig, aber es ist erstens durchaus sinnvoll, und zweitens auch nicht schwer zu ermitteln.

Diese Funktion nenne ich einfach **Now**. Schauen wir uns zunächst dieses Script an – später erkläre ich dann, wo und wie man diese Funktionen einbindet (siehe Listing 1).

Pascal und seine Besonderheiten gegenüber VB/VBA

Pascal hat bezüglich seiner Programmiersprache andere Formalien als wir es von VB/VBA her gewohnt sind, ist

Das Application-Objekt

Eines der wichtigsten Objekte bei der Programmierung von Access-Anwendungen ist das Application-Objekt. Es bietet viele Eigenschaften und Methoden, die stiefmütterlich behandelt werden. Dabei lohnt es sich, einmal einen Blick darauf zu werfen – dann weiß man im Fall der Fälle, dass es da irgendwo einen passenden Befehl geben muss. Dieser Beitrag zeigt eine Übersicht der Elemente des Application-Objekts und deren Funktion. In weiteren Beiträgen schauen wir uns die Funktion der einzelnen Eigenschaften und Methoden im Detail an.

Application-Objekt im Objektkatalog

Wenn Sie irgendwann einmal vor einer Aufgabe stehen, von der Sie wissen, dass Sie in diesem Beitrag von einer möglichen Lösung gelesen haben, können Sie schnell im Objektkatalog des VBA-Editors nachsehen – dort finden Sie zumindest eine Auflistung der Methoden und Eigenschaften des **Application**-Objekts. Den Objektkatalog öffnen Sie mit der Taste **F2**. Oben im Suchfeld geben Sie **Application** ein und klicken dann auf den gefundenen Eintrag. Die untere Liste liefert dann alle Member dieser Klasse (siehe Bild 1).

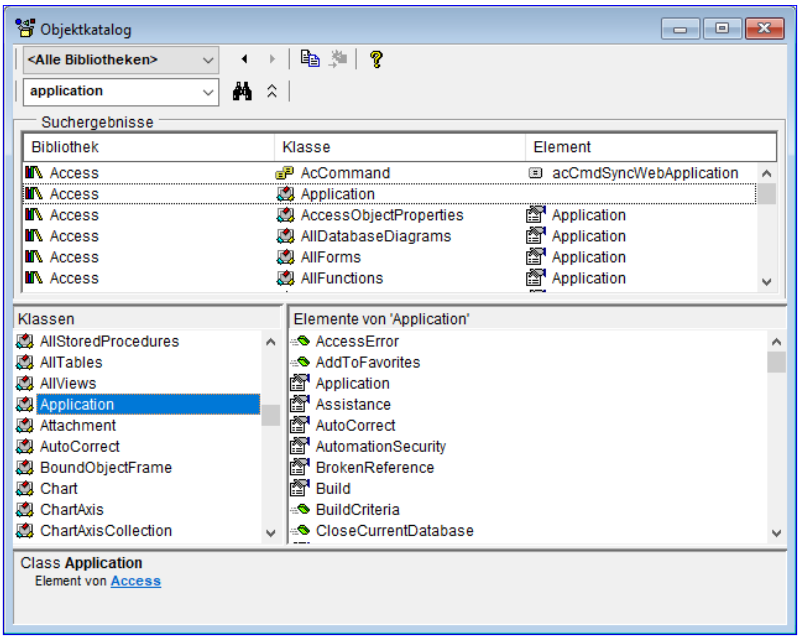


Bild 1: Elemente des **Application**-Objekts

Nachfolgend finden Sie die mehr oder weniger kurzen Beschreibungen der Elemente des **Application**-Objekts. Elemente für Webdatenbanken oder Elemente für ältere Techniken wie DDE haben wir nicht berücksichtigt. Sie können die Befehle auch ohne vorangestelltes **Application** nutzen.

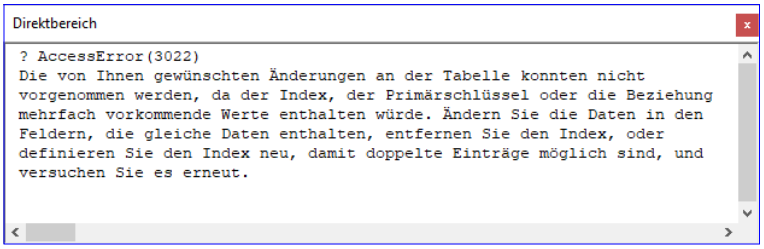


Bild 2: Ermitteln des Textes eines Access-Fehlers

AccessError

Diese Funktion erwartet eine Fehlernummer als Parameter und liefert die Beschreibung des Fehlers in der Anwendungssprache (siehe Bild 2). Praktisch, wenn ein Kunde Ihnen nur die Fehlernummer zu einem Fehler liefert und Sie schnell nachsehen wollen, welche Meldung sich dahinter verbirgt.

AddToFavorites

Hat in früheren Versionen von Access die aktuelle Datenbank zur Liste der Favoriten hinzugefügt.

Assistance

Das mit der Eigenschaft **Assistance** gelieferte Objekt bietet vier Methoden an, die Sie in Bild 3 sehen.

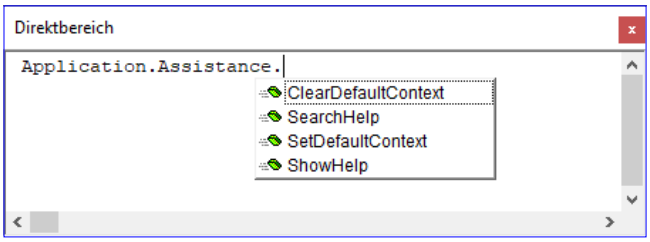


Bild 3: Methoden des **Assistance**-Objekts

SearchHelp scheint noch die nützlichste Methode zu sein, sie öffnet nach Eingabe eines Suchbegriffs als Parameter die Support-Seite von Microsoft mit den passenden Themen:

```
Application.Assistance.SearchHelp "Assistance"
```

Dies funktioniert jedoch nicht gut, denn für dieses Beispiel liefert die Seite keine Suchergebnisse.

AutoCorrect

Das mit **AutoCorrect** gelieferte Objekt bietet nur eine Eigenschaft namens **DisplayAutoCorrectOptions** an. Hiermit können Sie einstellen, ob die entsprechenden Optionen angezeigt werden sollen:

```
Application.AutoCorrect.DisplayAutoCorrectOptions = False
```

Weitere Informationen zur Autokorrektur finden Sie im Beitrag **Autokorrektur-Einträge im Griff** (**www.access-im-unternehmen.de/1279**).

AutomationSecurity

Diese Eigenschaft legt fest, in welchem Sicherheitsmodus Access andere Access-Anwendungen per VBA öffnet (auch einsetzbar von anderen Office-Anwendungen aus). Zum Öffnen wird hier **OpenCurrentDatabase** verwendet (siehe weiter unten). Es gibt drei Einstellungen:

- **msoAutomationSecurityByUI**: Verwendet die Einstellungen in den Access-Optionen (zu finden in dem Dialog, der mit **Datei|Optionen** geöffnet wird, unter **Trust Center|Einstellungen für das Trust Center|Makroeinstellungen**).
- **msoAutomationSecurityForceDisable**: Öffnet die Anwendung ohne Aktivierung von VBA, wenn die Makroeinstellung im oben genannten Dialog auf **Alle Makros ohne Benachrichtigung aktivieren** oder **Alle Makros mit Benachrichtigung aktivieren** steht. Wenn die Einstellung auf **Alle Makros aktivieren** steht, hat **msoAutomationSecurityForceDisable** keine Wirkung.
- **msoAutomationSecurityLow**: Aktiviert VBA, auch wenn die Makroeinstellung im oben genannten Dialog auf **Alle Makros ohne Benachrichtigung aktivieren** oder **Alle Makros mit Benachrichtigung aktivieren** steht.

BrokenReference

Mit dieser Eigenschaft finden Sie heraus, ob das VBA-Projekt kaputte oder nicht vorhandene Verweise enthält:

```
Public Sub Test_BrokenReference()
    Dim objReference As Access.Reference
    If Application.BrokenReference Then
        MsgBox "Es gibt kaputte Verweise."
        For Each objReference In Application.References
            If objReference.IsBroken Then
                Debug.Print objReference.Name
            End If
        Next objReference
    Else
        MsgBox "Alle Verweise in Ordnung."
    End If
End Sub
```

Damit brauchen Sie dann nicht erst die Verweise- beziehungsweise **References**-Auflistung zu durchlaufen. Das ist erst notwendig, wenn **BrokenReference** den Wert

True liefert. In diesem Fall prüfen wir mit der Eigenschaft **IsBroken** des **Reference**-Objekts, ob der aktuell durchlaufene Verweis kaputt ist und geben dann seinen Namen im Direktbereich des VBA-Editors aus.

Build

Gibt die Buildnummer der aktuellen Version von Microsoft Access zurück:

```
? Application.Build
 14228
```

BuildCriteria

Die **BuildCriteria**-Funktion erlaubt es, Vergleichsausdrücke für verschiedene Datentypen zusammenzustellen. Das ist insbesondere praktisch, wenn man Kriterien etwa für die DLookup-Funktion oder andere Domänenfunktionen zusammenstellen möchte. Die Funktion erwartet folgende Parameter:

- **Field**: Name des Feldes, nach dessen Inhalt gesucht werden soll
- **FieldType**: Felddatentyp, zum Beispiel **dbText**, **dbLong**, **dbCurrency** oder **dbDate**
- **Expression**: Vergleichsausdruck, zum Beispiel **André**, **1**, **Date()**

Hier sind einige Beispiele:

```
Debug.Print BuildCriteria("Vorname", dbText, "André")
Vorname="André"
Debug.Print BuildCriteria("Vorname", dbText, "A*")
Vorname Like "A*"
Debug.Print BuildCriteria("Menge", dbLong, "1")
Menge=1
Debug.Print BuildCriteria("Geburtsdatum", dbDate, Date)
Geburtsdatum=#10/30/2021#
Debug.Print BuildCriteria("Startzeit", dbDate, Now)
Startzeit=#10/30/2021 10:13:4#
Debug.Print BuildCriteria("Einzelpreis", dbCurrency,
"10,50")
Einzelpreis=10.5
Debug.Print BuildCriteria("Aktiv", dbBoolean, "Falsch")
Aktiv=False
```

Wir sehen hier, dass die Funktion recht flexibel ist. Beim Vergleich von Textfeldern ohne Platzhalter wie dem Sternchen (*) wird das Gleichheitszeichen als Operator verwendet, bei solchen mit Sternchen **LIKE**. Außerdem werden Zeichenketten automatisch in Anführungszeichen eingefasst. Datumsangaben werden in ein SQL-kompatibles Format umgewandelt und in Zahlen werden Kommata durch das aktuell verwendete Dezimaltrennzeichen ersetzt. Außerdem wandelt die Funktion verschiedene **Boolean**-Werte, zum Beispiel **Wahr**, direkt in eine verwertbare Form um (**True**).

Im bald erscheinenden Beitrag **Suchfunktion mit BuildCritera** (**www.access-im-unternehmen.de/1339**) zeigen wir einen Anwendungszweck für diese Funktion.

CloseCurrentDatabase

Schließt die Datenbank in der mit dem **Application**-Objekt angegebenen Access-Instanz. Sie können diesen Befehl in der aktuellen Access-Instanz wie folgt einsetzen:

```
Application.CloseCurrentDatabase
```

Wenn Sie eine Datenbank wie weiter unten unter dem Befehl **OpenCurrentDatabase** beschrieben in einer eigenen Instanz geöffnet haben, die beispielsweise **objAccess** heißt, verwenden Sie folgende Anweisung, um die Datenbank zu schließen:

```
objAccess.Application.CloseCurrentDatabase
```

CodeContextObject

CodeContextObject liefert den Namen des Objekts, in dem ein Code ausgeführt wird. Es funktioniert nur für Code in den Klassenmodulen von Formular- und Be-

richtsmodulen. Allerdings können Sie es auch in Routinen nutzen, die von Code in Formularen oder Berichten aus aufgerufen werden. So können Sie etwa in Fehlerbehandlungsroutinen abfragen, in welchem Objekt der Fehler ausgelöst wurde, ohne dass Sie einen Verweis auf das Objekt oder seinen Namen als Parameter an die Fehlerbehandlungsroutine übergeben müssen.

Dazu platzieren wir beispielsweise folgende Prozedur hinter der Ereigniseigenschaft einer Schaltfläche in einem Formular:

```
Private Sub cmdFehlerAusloesen_Click()
    On Error Resume Next
    Debug.Print 1 / 0
    If Not Err.Number = 0 Then
        Call ErrorHandler
    End If
End Sub
```

Die von dort aufgerufene Prozedur **ErrorHandler** legen wir in einem Standardmodul an:

```
Public Sub ErrorHandler()
    MsgBox "Fehler in '" _
            & Application.CodeContextObject.Name & "' (" _
           & TypeName(Application.CodeContextObject) & ")"
End Sub
```

Diese liefert dann den Namen des Formulars sowie den Typ (siehe Bild 4).

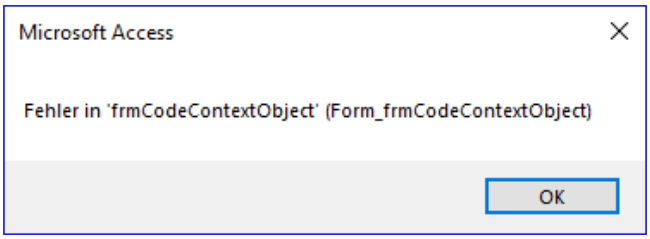


Bild 4: Meldung mit Informationen der Eigenschaft **CodeContextObject**

CodeData

Die Eigenschaft **CodeData** hat eine sehr ähnliche Funktion wie **CurrentData** (für die Beschreibung verweisen wir daher auf die Eigenschaft **CurrentData**, die Sie weiter unten finden). Die Eigenschaft ist für die Nutzung in Access-Add-Ins vorgesehen. Dort können Sie dann mit **CodeData** auf die Elemente der Add-In-Datenbank zugreifen und mit **CurrentData** auf die Elemente der aktuell geladenen Datenbank.

CodeDb

CodeDb entspricht im Prinzip der Eigenschaft **CurrentDb**, daher verweisen wir an dieser Stelle auf die Beschreibung der Eigenschaft **CurrentDb**, die wir weiter unten vorstellen. Genau wie bei **CodeData** und **CurrentData** ist **CodeDb** für den Einsatz in Access-Add-Ins vorgesehen, um auf das **Database**-Objekt der Add-In-Datenbank zuzugreifen. Mit **CurrentDb** hingegen greift man von der Add-In-Datenbank auf das **Database**-Objekt der aktuell in Access angezeigten Datenbank zu.

CodeProject

CodeProject entspricht im Prinzip der Eigenschaft **CurrentProject**, daher verweisen wir an dieser Stelle auf die Beschreibung der Eigenschaft **CurrentProject**, die wir weiter unten vorstellen. Genau wie bei **CodeData** und **CurrentData** ist **CodeProject** für den Einsatz in Access-Add-Ins vorgesehen, um auf das **CurrentProject**-Objekt der Add-In-Datenbank zuzugreifen. Mit **CurrentProject** hingegen greift man von der Add-In-Datenbank auf das **CurrentProject**-Objekt der aktuell in Access angezeigten Datenbank zu.

ColumnHistory

Mit dieser Funktion können Sie den Versionsverlauf der Daten in einem Memofeld abfragen. Dazu erstellen Sie ein Memofeld (Datentyp **Langer Text**) und stellen dessen Eigenschaft **Nur anfügen** auf **Ja** ein (siehe Bild 5). Dann können Sie mit **ColumnHistory** für einen bestimmten Datensatz den Versionsverlauf auslesen. Für unser Beispiel und den Datensatz mit dem Wert **1** im Feld **ID** sieht der Aufruf wie folgt aus:

```
Debug.Print Application.ColumnHistory("tblMemofelder", "Memofeld", "ID = 1")
```

Nachdem wir erst eine, dann eine zweite Zeile eingegeben haben, erhalten wir dieses Ergebnis:

```
[Version: 30.10.2021 11:27:22 ] Erste Zeile.
[Version: 30.10.2021 11:27:43 ] Erste Zeile.
Zweite Zeile.
```

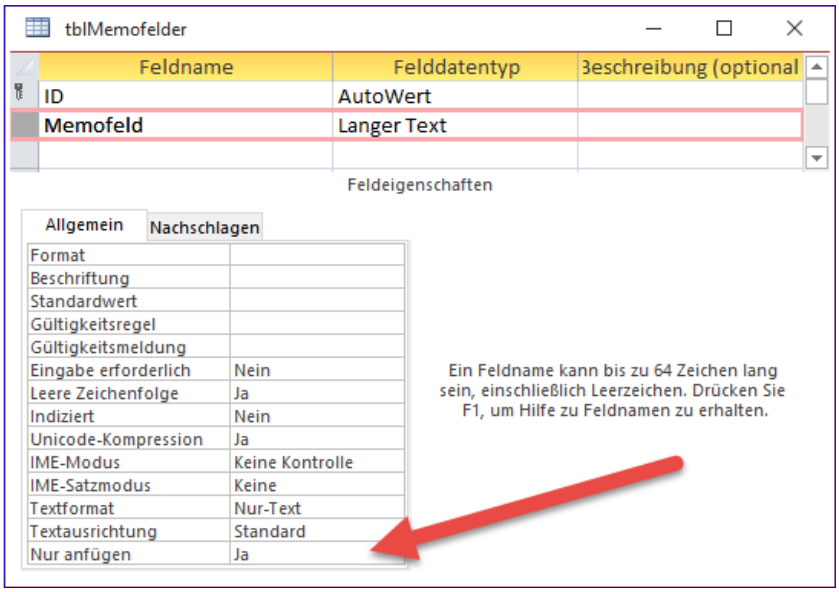


Bild 5: Memofeld zum Anfügen von Texten

COMAddIns

Die Auflistung **COMAddIns** liefert alle COM-Add-Ins, die für die aktuelle Access-Instanz vorliegen. COM-Add-Ins sind Add-Ins, die im Gegensatz zu Access-Add-Ins mit alternativen Entwicklungsumgebungen erstellt werden, zum Beispiel VB6, .NET oder twinBASIC. Die folgende **For Each**-Schleife gibt die Beschreibung und die **ProgID** aller aktuell installierten Elemente aus:

```
Public Sub Test_COMAddIns()
    Dim objCOMAddIn As COMAddIn
    For Each objCOMAddIn In Application.COMAddIns
        Debug.Print objCOMAddIn.Description, _
                                    objCOMAddIn.ProgId
    Next objCOMAddIn
End Sub
```

CommandBars

Mit der **CommandBars**-Auflistung greifen Sie auf alle **CommandBar**-Definitionen der aktuellen Instanz von Access zu. Das mag in Zeiten des Ribbons veraltet zu sein, aber wir wollen die guten, alten Kontextmenüs nicht vergessen: Auch diese lassen sich mit dieser Auflistung ermitteln.

Im folgenden Beispiel durchlaufen wir alle Elemente der **CommandBars**-Auflistung und geben all diejenigen im Direktbereich des VBA-Editors aus, deren Eigenschaft **Type** den Wert **msoBarTypePopup** aufweist:

```
Public Sub Test_CommandBars()
    Dim cbr As CommandBar
    Debug.Print "Anzahl CommandBars:" _
                            & Application.CommandBars.Count
    For Each cbr In Application.CommandBars
        If cbr.Type = msoBarTypePopup Then
            Debug.Print cbr.Name
        End If
    Next cbr
End Sub
```

CompactRepair

Diese Methode komprimiert und repariert die im ersten Parameter angegeben Datenbank und erzeugt eine komprimierte und reparierte Version unter dem mit dem zweiten Parameter angegebenen Dateinamen:

```
Application.CompactRepair _
                CurrentProject.Path & "\Beispiel.accdb", _
    CurrentProject.Path & "\Beispiel_Komprimiert.accdb", _
                                                    True
```

Mit dem dritten Parameter geben Sie an, ob im Falle von bei der Reparatur entdeckten korrupten Elemente eine Log-Datei im Verzeichnis der Datenbank angelegt werden soll.

ConvertAccessProject

Diese Methode konvertiert eine ADP-Datei, also ein Access Data Project, in eine reine Access-Datenbank. Sie erwartet die folgenden Parameter:

- **SourceFilename**: Name der zu konvertierenden ADP-Datei
- **DestinationFilename**: Name der zu erstellenden Datei
- **DestinationFileFormat**: Format der zu erstellenden Datei (zum Beispiel **acFileFormatAccess2007**)

CreateAccessProject

Mit dieser Methode können Sie eine ADP-Datei, also ein Access Data Project, erzeugen. Sie ist allerdings in neueren Versionen von Access nicht mehr verfügbar und der Aufruf löst den Fehler **Das Format ADP (Access Data Project) wird ab dieser Version von Access nicht mehr unterstützt.** aus:

```
Application.CreateAccessProject _
                    CurrentProject.Path & "\Beispiel.adp"
```

Als zweiten Parameter können Sie noch die Verbindungszeichenfolge für die zu verbindende SQL Server-Datenbank angeben.

CreateAdditionalData

Diese Methode dient dazu, bei einem Export im XML-Format der Daten weitere Tabellen zu den Daten der zu exportierenden Tabelle hinzuzufügen.

Weitere Informationen zu dieser Methode finden Sie im Beitrag **XML-Export mit VBA** (**www.access-im-unternehmen.de/1046**).

CreateControl

Die Funktion **CreateControl** erstellt ein neues Steuerelement in einem Formular und liefert einen Verweis auf das neue Steuerelement als Funktionswert zurück. Diese und die Methoden **CreateForm** und **DeleteControl** erläutern wir im Artikel **Formulare per VBA erstellen** (**www.access-im-unternehmen.de/1332**).

CreateForm

Die Methode **CreateForm** erstellt ein neues Formular und liefert einen Verweis auf das neu erstellte Formular als Funktionswert zurück.

CreateGroupLevel

Diese Methode erzeugt in einem Bericht einen neuen Gruppierungs- oder Sortierungsbereich.

Diese und die Methoden **CreateReport**, **CreateReportControl** und **DeleteReportControl** erläutern wir im Artikel **Berichte per VBA erstellen** (**www.access-im-unternehmen.de/1338**).

CreateReport

Diese Funktion erstellt einen neuen Bericht und liefert einen Verweis auf den neu erstellten Bericht als Funktionswert zurück.

CreateReportControl

Diese Funktion erstellt ein Steuerelement in einem Bericht und liefert einen Verweis auf das Steuerelement als Funktionswert zurück.

CurrentData

Das mit der Eigenschaft **CurrentData** gelieferte Objekt liefert Eigenschaften mit verschiedenen Auflistungen:

- **AllQueries**: Liefert eine Auflistung aller Abfragen der aktuellen Datenbank.
- **AllTables**: Liefert eine Auflistung aller Tabellen der aktuellen Datenbank.

Assistent für m:n-Beziehungen

Microsoft Access bietet eine ganze Reihe von praktischen Assistenten. Ich setze beispielsweise sehr oft den Nachschlage-Assistenten ein, der nicht nur eine Beziehung mit den gewünschten Optionen anlegt, sondern das bearbeitete Feld auch noch als Kombinationsfeld auslegt, mit dem die Daten der verknüpften Tabelle leicht ausgewählt werden können. Eine m:n-Beziehung stellen Sie her, indem Sie zwei solcher Nachschlagefelder in der sogenannten Verknüpfungstabelle anlegen. Noch praktischer wäre es, wenn Sie diese Verknüpfungstabelle gar nicht erst anlegen müssten. Stattdessen wären nur die zu verknüpfenden Tabellen auszuwählen und der Assistent erledigt den Rest – das Anlegen der Verknüpfungstabelle mit den notwendigen Feldern sowie das Einrichten der Beziehungen zu den zu verknüpfenden Tabellen. Dieser Beitrag zeigt, wie Sie einen solchen Assistenten programmieren können.

Ausgangsposition

Der Assistent soll zwei Tabellen wie die in Bild 1 miteinander verknüpfen. Dazu benötigen wir eine weitere Tabelle etwa namens **tblProdukteKategorien** mit dem Primärschlüsselfeld **ProduktKategorieID** und den beiden Fremdschlüsselfeldern **ProduktID** und **KategorieID**.

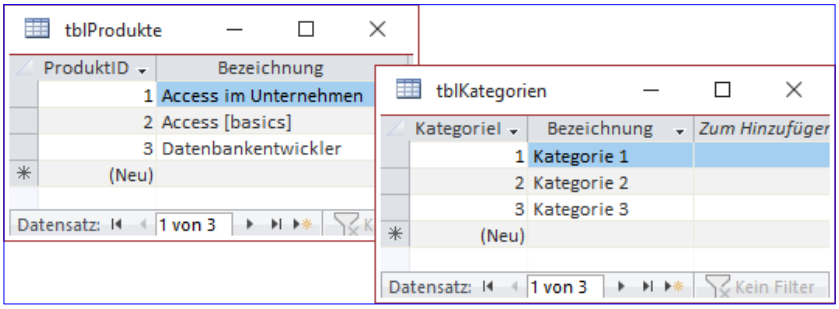


Bild 1: Zu verknüpfende Tabellen

Diese stellen jeweils die Verknüpfung zu den Tabellen **tblProdukte** und **tblKategorien** her, sodass alle Einträge der Tabelle **tblProdukte** mit den Einträgen der Tabelle **tblKategorien** verknüpft werden können. Die Fremdschlüsselfelder legen Sie am schnellsten an, indem Sie im Tabellenentwurf den Eintrag **Nachschlage-Assistent...** wählen und die Verknüpfung darüber vornehmen. Die Verknüpfungstabelle sieht danach wie in Bild 2 aus.

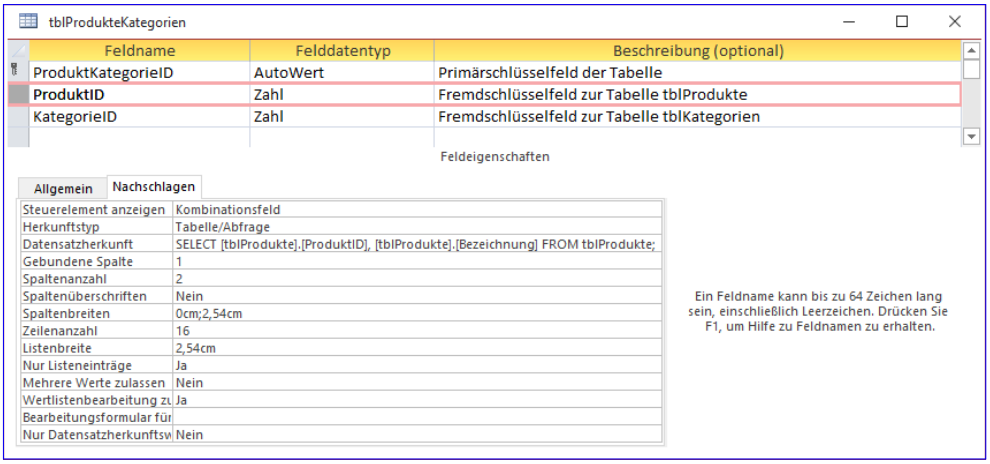


Bild 2: Die Verknüpfungstabelle

Wechseln Sie in die Datenblattansicht, erhalten Sie eine Tabelle, mit deren Fremdschlüsselfeldern Sie

die gewünschten Kombinationen aus Produkten und Kategorien leicht zusammenstellen können (siehe Bild 3).

Die Beziehungen können Sie danach wie in Bild 4 im **Beziehungen**-Fenster einsehen.

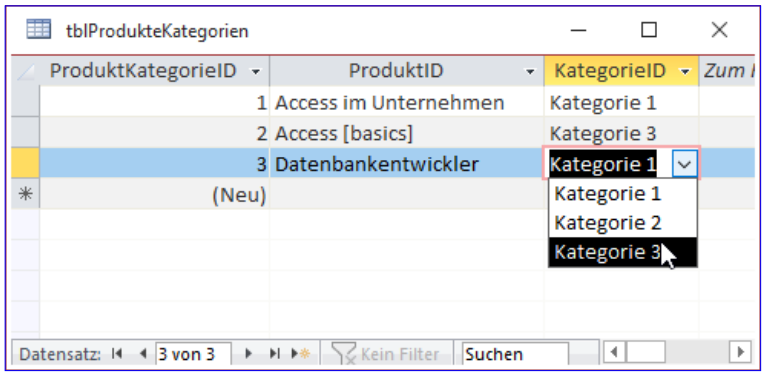


Bild 3: Datenblattansicht der Verknüpfung

Vorher

Um eine m:n-Beziehung zwischen zwei Tabellen herzustellen, sind normalerweise die folgenden Schritte nötig:

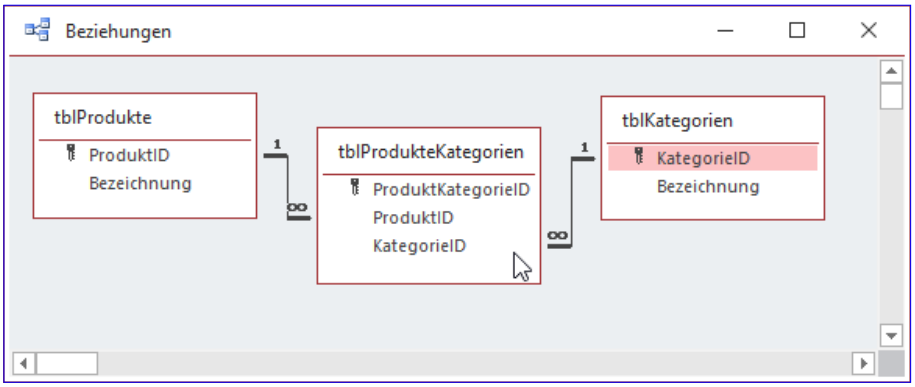


Bild 4: Die m:n-Beziehung im Beziehungen-Fenster

- Erstellen der Verknüpfungstabelle
- Wahl eines Namens für die Verknüpfungstabelle
- Hinzufügen eines Primärschlüsselfeldes
- Hinzufügen eines Fremdschlüsselfeldes zum Verknüpfen der Verknüpfungstabelle mit der m-Tabelle
- Hinzufügen eines Fremdschlüsselfeldes zum Verknüpfen der Verknüpfungstabelle mit der n-Tabelle
- Aufrufen des Nachschlage-Assistenten zum Erstellen der Beziehung zwischen dem ersten Fremdschlüsselfeld und dem Primärschlüsselfeld der m-Tabelle oder, falls kein Nachschlagefeld gewünscht ist, sondern nur die reine Beziehung, manuelles Hinzufügen der Beziehung über das Beziehungen-Fenster. Außerdem Einstellen der Beziehungseigenschaften wie referenzielle Integrität, Löschweitergabe und Aktualisierungsweitergabe.
- Aufrufen des Nachschlage-Assistenten zum Erstellen der Beziehung zwischen dem zweiten Fremdschlüsselfeld und dem Primärschlüsselfeld der n-Tabelle oder, falls kein Nachschlagefeld gewünscht ist, sondern nur die reine Beziehung, manuelles Hinzufügen der Beziehung über das **Beziehungen**-Fenster. Außerdem Einstellen der Beziehungseigenschaften wie referenzielle Integrität, Löschweitergabe und Aktualisierungsweitergabe.
- Gegebenenfalls Hinzufügen weiterer Felder zur m:n-Verknüpfungstabelle wie etwa Einzelpreis oder Menge bei einer Tabelle zum Speichern von Bestellpositionen

Anforderungen

Was genau benötigen wir an Informationen, um automatisiert eine Verknüpfungstabelle für zwei per m:n-Beziehung zu verknüpfenden Tabellen zu erstellen? Beziehungsweise welche Informationen muss der Assistent zur Erstellung einer m:n-Beziehung vom Entwickler abfragen?

- Name der ersten Tabelle

- Name des Primärschlüsselfeldes der ersten Tabelle
- Gegebenenfalls Name des anzuzeigenden Feldes der ersten Tabelle
- Name der zweiten Tabelle
- Name des Primärschlüsselfeldes der zweiten Tabelle
- Gegebenenfalls Name des anzuzeigenden Feldes der zweiten Tabelle
- Name der zu erstellenden Tabelle
- Namen für die Fremdschlüsselfelder zum Herstellen der Beziehung mit der m-Tabelle und der n-Tabelle
- Gegebenenfalls weitere Felder, die zur m:n-Tabelle hinzugefügt werden sollen

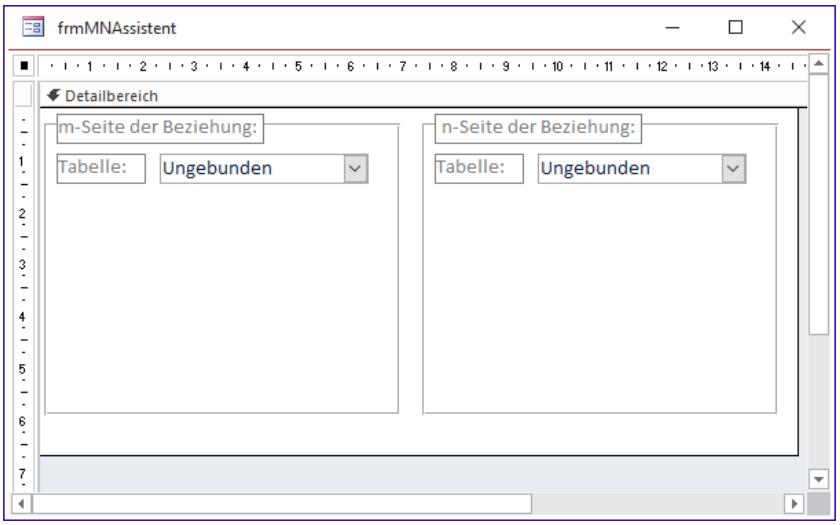


Bild 5: Assistenten-Formular

Programmieren des Add-Ins

Starten wir also direkt mit der Programmierung des Add-Ins. Dieses soll über ein Formular verfügen, mit dem wir alle Einstellungen für die Erstellung der m:n-Beziehung vornehmen können.

Auswahl der beteiligten Tabellen

Als Erstes benötigen wir zwei Kombinationsfelder, mit denen wir die beiden an der Beziehung beteiligten Tabellen ermitteln können.

Diese Kombinationsfelder erhalten nach dem Hinzufügen zu einem neuen Formular namens **frmMNAssistent** die Bezeichnungen **cboMTabelle** und **cboNTabelle**. Diese fügen wir jeweils in einen neuen Rahmen ein, sodass der Aufbau in der Entwurfsansicht zunächst wie in Bild 5 aussieht.

Damit der Benutzer die in der aktuellen Datenbank enthaltenen Tabellen auswählen kann, stellen wir die Eigenschaft **Datensatzherkunft** der beiden Kombinationsfelder auf die Abfrage aus Bild 6 ein.

Damit die hier verwendete Tabelle **MSysObjects** sichtbar ist, müssen Sie zunächst die entsprechende Option aktivieren. Diese finden Sie, wenn Sie mit der rechten Maustaste auf den Navigationsbereich klicken und dann den Eintrag **Navigationsoptionen...** auswählen. Im nun erscheinenden Dialog **Navigationsoptionen** aktivieren Sie die Option **Systemobjekte anzeigen**.

Danach können Sie der Abfrage für die Eigenschaft **Datensatzherkunft** wie in der Abbildung die Tabelle **MSysObjects** hinzufügen und die beiden Felder **Name** und **Type** zum Entwurfsraster der Abfrage hinzufügen. Außerdem stellen Sie als Kriterium für das Feld **Name** den Wert **Nicht Wie "MSys*" Und Nicht Wie "USys*" Und Nicht Wie "f_*"** und für das Feld **Type** den Wert **1** ein. Das sorgt erstens dafür, dass keine System- und temporären Tabellen erscheinen und dass nur lokale Tabellen zur Auswahl angeboten werden.

Nach dem Schließen des Abfrageentwurfs können Sie den Wert der Eigenschaft **Datensatzherkunft** des Kombinationsfeldes **cboMTabelle** in die entsprechende Eigenschaft des Kombinationsfeldes **cboNTabelle** kopieren:

```
SELECT Name, Type
FROM MSysObjects
WHERE (Name Not Like "MSys*"
And Name Not Like "USys*"
And Name Not Like "f_*")
AND (Type=1);
```

Wenn Sie anschließend in die Formularansicht wechseln, können Sie bereits die an der Beziehung beteiligten Tabellen auswählen (siehe Bild 7).

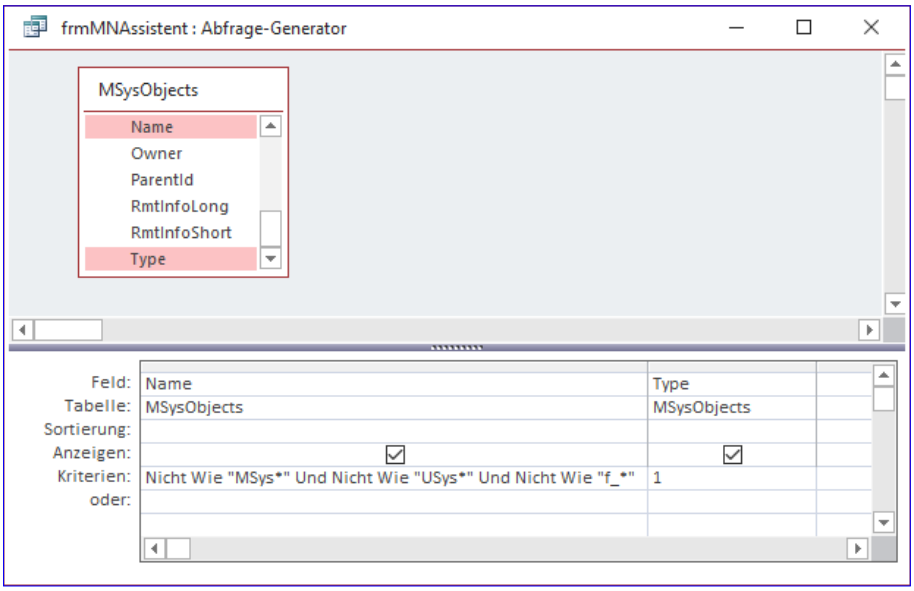


Bild 6: Datensatzherkunft der Kombinationsfelder zur Auswahl der an der Beziehung beteiligten Tabellen

Auswahl der beteiligten Primärschlüsselfelder

Unter den beiden Kombinationsfeldern platzieren wir zwei weitere Kombinationsfelder namens **cboMPrimaerschluessel** und **cboNPrimaerschluessel**.

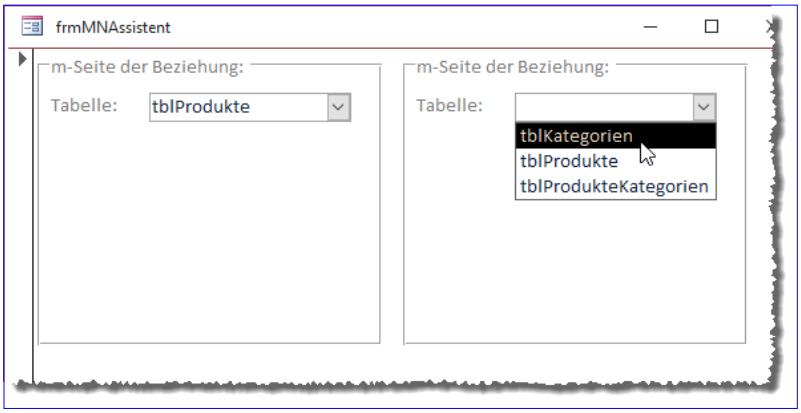


Bild 7: Auswahl der an der Beziehung beteiligten Kombinationsfelder

Diese sollen direkt nach der Auswahl einer Tabelle mit dem darüber befindlichen Kombinationsfeld alle Felder der gewählten Tabelle anzeigen und das Primärschlüsselfeld dieser Tabelle als Vorauswahl einstellen.

Dazu müssen wir zwei Aufgaben erledigen:

- eine Liste aller Felder der jeweiligen Tabelle zusammenstellen und
- das Primärschlüsselfeld aus diesen Feldern ermitteln.

Die Liste der Felder können wir auf zwei Wegen herausfinden. Der erste verwendet ausschließlich VBA und nutzt die **Fields**-Auflistung des jeweiligen **TableDef**-Objekts.

Der zweite verwendet die Herkunftsart **Feldliste** und die im ersten Kombinationsfeld ausgewählte Tabelle als **Datensatzherkunft**. Dabei müssen wir aber dennoch nach der Auswahl VBA bemühen, um die Eigenschaft **Datensatzherkunft** zu füllen. Um nur das zu erledigen, hinterlegen wir die folgenden beiden Ereignisprozeduren jeweils für Kombinationsfelder **cboMTabelle** und **cboNTabelle**:

```
Private Sub cboMTabelle_AfterUpdate()
    Me!cboMPrimaerschluessel.RowSource = Me!cboMTabelle
End Sub

Private Sub cboNTabelle_AfterUpdate()
    Me!cboNPrimaerschluessel.RowSource = Me!cboNTabelle
End Sub
```

Damit bieten die beiden Kombinationsfelder bereits die Felder der jeweiligen Tabelle zur Auswahl an (siehe Bild 8).

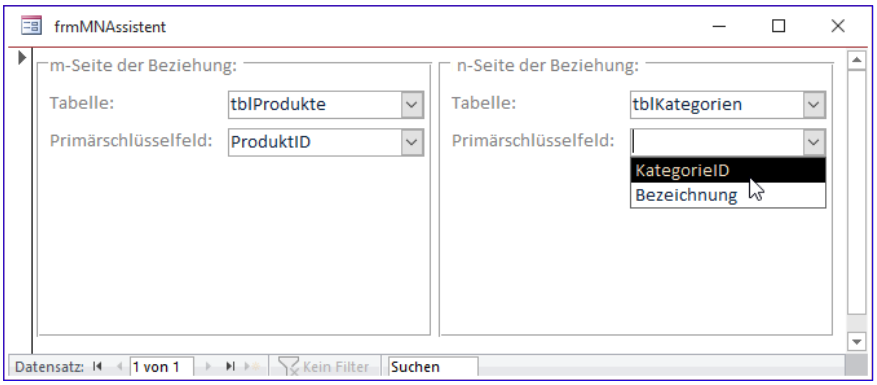


Bild 8: Auswahl des Primärschlüsselfeldes für das Erstellen der m:n-Beziehung

Primärschlüsselfeld auswählen

Nun wollen wir noch dafür sorgen, dass direkt die Primärschlüsselfelder dieser Tabellen angezeigt werden. Schließlich soll der Benutzer so wenig Aufwand wie möglich haben. Dazu fügen wir den beiden Ereignisprozeduren **cboMTabelle_AfterUpdate** und **cboNTabelle_AfterUpdate** jeweils einen Aufruf einer Funktion namens **PrimaerschluesselErmitteln** hinzu:

```
Private Sub cboMTabelle_AfterUpdate()
    Me!cboMPrimaerschluessel.RowSource = Me!cboMTabelle
    Me!cboMPrimaerschluessel = _
            PrimaerschluesselErmitteln(Me!cboMTabelle)
End Sub
```

Diese Funktion erwartet den Namen der zu untersuchenden Tabelle als Parameter. Sie füllt die Variable **db** mit einem Verweis auf das aktuelle **Database**-Objekt und **tdf** mit einem Verweis auf das **TableDef**-Objekt für die Tabelle aus **strTabelle**. Dann durchläuft sie alle Einträge der Auflistung **tdf.Indexes** und speichert das jeweils aktuelle Element in der Variablen **idx**. Für dieses prüft sie den Wert der Eigenschaft **Primary**.

Ist dieser **True**, schreibt sie den Namen des ersten Feldes dieses Indizes in den Rückgabewert der Funktion und beendet diese mit **Exit Function**. Das funktioniert für keinen, einen oder zusammengesetzte Primärschlüssel gleichermaßen sinnvoll: Wenn kein Primärschlüssel für die Tabelle vorhanden ist, liefert die Funktion eine leere Zeichenkette zurück, für einen Primärschlüssel mit einem Feld den Namen des betroffenen Feldes und für einen zusammengesetzten Primärschüssel den Namen des ersten Feldes:

```
Private Function PrimaerschluesselErmitteln(_
                    strTabelle As String) As String
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim idx As DAO.Index
    Set db = CurrentDb
    Set tdf = db.TableDefs(strTabelle)
    For Each idx In tdf.Indexes
        If idx.Primary Then
            PrimaerschluesselErmitteln = _
                                idx.Fields(0).Name
            Exit Function
        End If
    Next idx
End Function
```

Für unser Beispiel mit den Produkten und Kategorien stellen die Prozeduren zuverlässig das jeweilige Primärschüsselfeld ein.

Informationen für das Nachschlagefeld

Zusätzlich wollen wir in diesem Formular die Daten für die Nachschlagefelder abfragen, sofern der Benutzer diese anlegen möchte. Manchmal ist es sinnvoll, zum Beispiel bei Bestellpositionen, wo man direkt aus der m:n-Verknüpfungstabelle das Produkt zu einer Bestellposition auswählen möchte. An anderen Stellen sind die Werte der Verknüpfungstabelle vielleicht gar nicht sichtbar, zum Beispiel wenn diese über zwei Listenfelder angezeigt werden.

Also fügen wir die notwendigsten Felder hinzu plus einem Kontrollkästchen zum Aktivieren oder Deaktivieren dieser Einstellungen.

Die Kontrollkästchen heißen **chkMNachschlagefeld** und **chkNNachschlagefeld**, die Kombinationsfelder zur Auswahl des anzuzeigenden Feldes für das Nachschlagefeld heißen **cboMNachschlagefeld** und **cboNNachschlagefeld** (siehe Bild 9). Damit die beiden Kombinationsfelder genau wie die zur Auswahl der Primärschlüsselfelder auch die Felder der zu verknüpfenden Tabellen anzeigen, fügen wir den Prozeduren **cboMTabelle_AfterUpdate** und **cboNTabelle_AfterUpdate** noch jeweils eine Anweisung hinzu:

```
Private Sub cboMTabelle_AfterUpdate()
    ...
    Me!cboMNachschlagefeld.RowSource = Me!cboMTabelle
End Sub

Private Sub cboNTabelle_AfterUpdate()
    ...
    Me!cboNNachschlagefeld.RowSource = Me!cboNTabelle
End Sub
```

Damit die beiden Kontrollkästchen beim Aktivieren und Deaktivieren auch die beiden Kombinationsfelder **cboMNachschlagefeld** und **cboNNachschlagefeld** aktivieren beziehungsweise deaktivieren, fügen wir diese Ereignisprozeduren für das Ereignis **Nach Aktualisierung** der beiden Kontrollkästchen hinzu:

```
Private Sub chkMNachschlagefeld_AfterUpdate()
    Me!cboMNachschlagefeld.Enabled = Me!chkMNachschlagefeld
End Sub

Private Sub chkNNachschlagefeld_AfterUpdate()
    Me!cboNNachschlagefeld.Enabled = Me!chkNNachschlagefeld
End Sub
```

Name für die zu erstellende Tabelle und ihre Felder ermitteln

Die Bezeichnungen für die zu erstellende Tabelle und die enthaltenen Felder bestimmen wir automatisch auf Basis der gewählten zu verknüpfenden Tabellen und den gewählten Primärschlüsselfeldern. Dies erledigen wir in der Prozedur **VerknuepfungstabelleAktualisieren** aus Listing 1. Sie liest zuerst die gewählten Tabellen in die Variablen **strMTabelle** und **strNTabelle** ein sowie die selektierten Primärschlüsselfelder.

Dann folgt die Prüfung, ob der Benutzer eines der Präfixe **tbl** oder **tbl_** für die beiden zu verknüpfenden Tabellen verwendet. Falls ja, werden diese vorn abgeschnitten, sodass beispielsweise von **tblProdukte** oder **tbl_Produkte** nur noch **Produkte** übrig bleibt.

Daraus leitet die Prozedur den Namen der Verknüpfungstabelle ab, der aus dem Präfix **tbl**, dem Namen der m-Tabelle und dem Namen der n-Tabelle jeweils ohne Präfix besteht.

Das Ergebnis landet im Textfeld **txtVerknuepfungstabelle**.

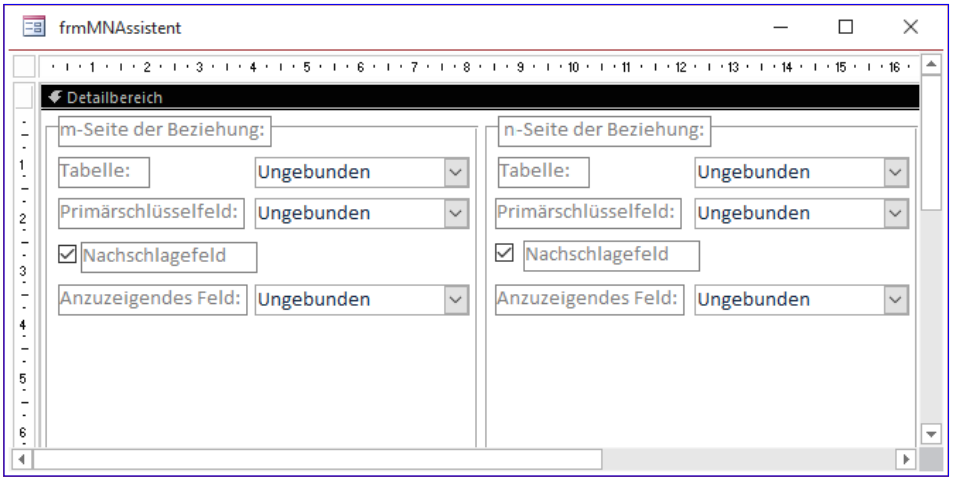


Bild 9: Informationen für die Nachschlagefelder

```
Private Sub VerknuepfungstabelleAktualisieren()
    Dim strMTabelle As String
    Dim strNTabelle As String
    Dim strMPrimaerschluessel As String
    Dim strNPrimaerschluessel As String
    strMTabelle = Nz(Me!cboMTabelle, "")
    strNTabelle = Nz(Me!cboNTabelle, "")
    strMPrimaerschluessel = Nz(Me!cboMPrimaerschluessel, "")
    strNPrimaerschluessel = Nz(Me!cboNPrimaerschluessel, "")
    If Left(strMTabelle, 4) = "tbl_" Then
        strMTabelle = Mid(strMTabelle, 5)
    End If
    If Left(strMTabelle, 3) = "tbl" Then
        strMTabelle = Mid(strMTabelle, 4)
    End If
    If Left(strNTabelle, 4) = "tbl_" Then
        strNTabelle = Mid(strNTabelle, 5)
    End If
    If Left(strNTabelle, 3) = "tbl" Then
        strNTabelle = Mid(strNTabelle, 4)
    End If
    Me!txtVerknuepfungstabelle = "tbl" & strMTabelle & strNTabelle
    If strMPrimaerschluessel = "ID" Then
        strMPrimaerschluessel = strMTabelle & "ID"
    ElseIf Right(strMPrimaerschluessel, 2) = "ID" Then
        strMPrimaerschluessel = Left(strMPrimaerschluessel, Len(strMPrimaerschluessel) - 2)
    End If
    If strNPrimaerschluessel = "ID" Then
        strNPrimaerschluessel = strNTabelle & "ID"
    ElseIf Right(strNPrimaerschluessel, 2) = "ID" Then
        strNPrimaerschluessel = Left(strNPrimaerschluessel, Len(strNPrimaerschluessel) - 2)
     End If
    Me!txtMFremdschluesselfeld = strMPrimaerschluessel & "ID"
    Me!txtNFremdschluesselfeld = strNPrimaerschluessel & "ID"
    Me!txtPrimaerschluesselfeld = strMPrimaerschluessel & strNPrimaerschluessel & "ID"
End Sub
```

Listing 1: Einstellen der Daten der Verknüpfungstabelle

Dann untersucht die Prozedur die gewählten Primärschlüsselfelder der beiden Tabellen. Diese dienen schließlich als Grundlage für die Benennung der Fremdschlüsselfelder der Verknüpfungstabelle.

Wenn der Name eines der Primärschlüsselfelder lediglich **ID** lautet, stellt die Prozedur den oben ermittelten Namen der Tabelle ohne Präfix voran. Das kann in unserem Fall auch der Plural sein, was zu Bezeichnungen wie **ProdukteID** oder **KategorienID** führen kann. Das ist aber kein Problem, denn der Benutzer kann die Bezeichnungen ja noch anpassen.

Sollte die Bezeichnung des Primärschlüsselfeldes jedoch der von uns erwarteten Konvention entsprechen, also aus dem Singular der Bezeichnung der enthaltenen