HAUFE.
Ausgabe 01/2023

ACCESS
IM UNTERNEHMEN

WIZARDS IN ACCESS

Die eigene Arbeit vereinfachen – und dann noch mit einem selbst programmierten Assistenten! Wie das geht, zeigen wir in dieser Ausgabe (ab Seite 48).

In diesem Heft:

Wizards mit Access

Microsoft Access kommt mit einigen mehr oder weniger nützlichen Assistenten. Diese finden wir an verschiedenen Stellen – zum Anlegen von Elemente wie Tabellen, Abfragen, Formularen oder Steuerelementen, aber auch zum Einstellen der Werte der Eigenschaften dieser Elemente. Richtig spannend wird es, wenn wir unsere eigenen Assistenten programmieren. Diese können wir für alle möglichen Elemente oder Eigenschaften definieren, um diese mit den von uns gewünschten Einstellungen zu erzeugen. In dieser Ausgabe schauen wir uns an, wie wir Eigenschafts-Assistenten programmieren – und schreiten auch direkt zur Tat und entwickeln einen Assistenten zum Testen und Zuweisen von Formatierungen für Datenfelder.

Den Einstieg in dieses Thema machen wir mit dem Beitrag **Eigenschaftsassistenten oder Property Wizards** ab Seite 24. Hier stellen wir die Grundlagen für die Entwicklung von Eigenschaftsassistenen vor – wie stellen wir sie zusammen, welche Elemente müssen sie unbedingt enthalten, wie können wir die Assistenten installieren? Und vor allem: Wie sorgen wir dafür, das der Assistent an der richtigen Stelle aufgerufen werden kann?

Wie Sie solche Assistenten praktisch einsetzen können, zeigen wir im Beitrag **Formatassistent für Textfelder** ab Seite 57. Hier stellen wir einen Assistenten vor, der immer auftaucht, wenn Sie die **Format**-Eigenschaft eines Elements markieren. Access zeigt dann eine Schaltfläche mit drei Punkten an (...), mit der wir den Assistenten starten können. Dieser bietet dann die Möglichkeit, eine Formatierung wie beispielsweise dd.mm.yyyy für Datumsangaben einzugeben und sich direkt anzuschauen, wie sich die Formatierung auf die vorhandenen Daten auswirkt.

Haben Sie schon einmal Ressourcenprobleme und Fehlermeldungen in Ihrer Datenbank erhalten? Das geschieht meistens, wenn zu viele Recordsets gleichzeitig geöffnet werden. Das kann passieren, wenn zu viele Formulare geöffnet sind, die an Tabellen oder Abfragen gebunden sind, oder auch wenn man in VBA-Prozeduren zu viele Recordsets anspricht. Welche Fehlermeldungen es gibt und welche Ideen wir haben, um diese Fehler zu verhindern, lesen Sie im Beitrag **Ressourcenprobleme mit Recordsets** ab Seite 2.

Zuletzt gab es wieder einige Updates von Microsoft für das Office-Paket (teilweise zum Glück nur im Beta-Kanal), welche wichtige Funktionen von Access-Anwendungen sabotiert haben. In diesem Fall hilft meist nur das Zurücksetzen der jeweiligen Office-Anwendung. Wie das gelingt, zeigen wir unter dem Titel **Fehlerhafte Access/Office-Updates zurücksetzen** ab Seite 9.

Die Datensatznavigation in Formularen ist teilweise recht fummelig, weil die Steuerelemente klein sind und immer am Fuß des Formulars eingeblendet werden. Also stellen wir im Beitrag **Datensatznavigation per Ribbon** ab Seite 13, wie Sie diese Elemente gut sichtbar in das Ribbon verschieben können.

Wollen Sie für jeden Datensatz ein eigenes Bild in einem Endlosformular anzeigen, erfahren Sie unter Bilder im Endlosformular ab Seite 32, wie es gelingt.

Und schließlich setzen wir den Schwerpunkt zum Thema Programmieren von Notion aus dem vorherigen Heft in den Beiträgen **Access und Notion synchronisieren** (ab Seite 36) und **JSON-Daten auslesen** (ab Seite 45) fort.

Jetzt aber viel Spaß beim Lesen und Ausprobieren!

Ihr André Minhorst

Ressourcenprobleme mit Recordsets

In letzter Zeit werde ich wieder häufiger von Entwicklerkollegen angesprochen, die Fehler wie »Nicht genügend Systemressourcen« oder »Nicht genügend Speicherplatz zum Ausführen der Operation« erhalten, wenn sie mehrere Formulare mit Daten öffnen oder Code ausführen, der mit Recordsets arbeitet. Wenn sie dann über den Taskmanager die Ressourcen des Systems betrachten, stellen sie fest, dass es hier keinerlei Engpässe gibt. Wir schauen uns in diesem Beitrag an, wie dieser Fehler entstehen kann und welche Möglichkeiten es gibt, diesen zu beheben.

Fehlermeldungen zu erhalten, die nicht unmittelbar mit dem geschriebenen Code und eventuellen Syntax- oder Kompilierfehlern zusammenhängen, ist selten schön – erst recht nicht, wenn diese Fehler auf dem einen Rechner auftreten, auf dem anderen aber nicht oder zu anderen Zeitpunkten.

Wir schauen uns in diesem Beitrag einmal an, wie wir die beiden eingangs genannten Fehler reproduzieren können. Anschließend kümmern wir uns um mögliche Lösungen für diese Fehler.

Fehler »Nicht genügend Systemressourcen« reproduzieren

Um diesen Fehler zu reproduzieren, erzeugen wir eine ausreichend große Menge von Recordset-Objekten. Um diese ohne viel Aufwand im Speicher zu halten, definieren wir ein Klassenmodul, das lediglich den Verweis auf das geöffnete Recordset-Objekt aufnehmen soll. Der Code dieses Klassenmoduls, das wir **clsRecordsetWrapper** nennen wollen, sieht wie folgt aus:

```
Private m_Recordset As Recordset

Public Property Set Recordset(rst As DAO.Recordset)
    Set m_Recordset = rst
End Property
```

Es enthält also lediglich eine private Variable, die den Verweis auf ein **Recordset**-Objekt speichern soll. Um dieses an die Klasse zu übermitteln, haben wir dieser außerdem die **Property Set**-Prozedur **Recordset** zugewiesen, der wir als Parameter das zu speichernde Recordset übergeben.

Diese Prozedur (siehe Listing 1) deklariert eine **Collection**-Variable zum Speichern der Instanzen der Wrapper-Klasse, Variablen zum Speichern eines Verweises auf das **Database**-Objekt der aktuellen Datenbank und des zu öffnenden Recordsets sowie für eine Instanz der Klasse **clsRecordsetwrapper**. Außerdem instanziiert die Prozedur ein neues **Collection**-Objekt und referenziert es mit der Variablen **col**.

Die folgende **For...Next**-Schleife wird maximal 10.000 Mal durchlaufen. Es ist allerdings davon auszugehen, dass unter der Standardkonfiguration normale Rechner nach ein paar hundert Iterationen aufgeben. Innerhalb der **For...Next**-Schleife erstellen wir ein neues Objekt auf Basis der Klasse **clsRecordsetwrapper** und weisen es der Variablen **objRecordsetWrapper** zu.

Dann weisen wir bei deaktivierter Fehlerbehandlung der Eigenschaft **Recordsets** dieser Klasse ein mit **OpenRecordset** neu geöffnetes Recordset auf Basis einer Beispieltabelle zu. Dies löst ab einer bestimmten Menge erstellter Recordsets den oben genannten Fehler aus. Ist das der Fall (oder tritt ein anderer Fehler auf), wollen wir diesen in einer Meldung ausgeben. Neben der Fehlernummer und der Fehlermeldung gibt die Meldung noch die An-

```
Public Sub Recordsets()
    Dim col As Collection
    Dim i As Long
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim objRecordsetWrapper As clsRecordsetwrapper
    Set db = CurrentDb
    Set col = New Collection
    For i = 1 To 10000
        Set objRecordsetWrapper = New clsRecordsetwrapper
        With objRecordsetWrapper
            On Error Resume Next
            Set .Recordset = db.OpenRecordset("tblVieleKunden", dbOpenDynaset)
            If Not Err.Number = 0 Then
                MsgBox "Fehler: " & Err.Number & vbCrLf & vbCrLf & "Beschreibung: " & Err.Description _
                    & vbCrLf & vbCrLf & "Anzahl Recordsets: " & i
                Exit Sub
            End If
            On Error GoTo 0
        End With
        col.Add objRecordsetWrapper
        Debug.Print i
    Next i
End Sub
```

Listing 1: Prozedur zum Erstellen einiger Recordsets und zum Speichern dieser in einer Wrapper-Klasse inklusive Fehlerbehandlung

zahl der geöffneten Recordsets aus. Nach der Anzeige der Fehlermeldung wird die Prozedur mit **Exit Sub** beendet.

Damit die Recordsets in der Wrapper-Klasse im Speicher verweilen und so erst dafür sorgen, dass der Fehler ausgelöst wird, fügen wir diese mit der **Add**-Methode zu der **Collection** namens **col** hinzu.

Rufen wir diese Prozedur nun auf, erscheint nach einigen bis etlichen Durchläufen der Schleife die Fehlermeldung aus Bild 1 (hier haben wir die Prozedur von einer Schaltfläche aus aufgerufen). Mit einem Klick auf **OK** wird die Prozedur beendet und die durch die **Recordset**-Variablen gebundenen Ressourcen werden wieder freigegeben. Der Fehler ist reproduzierbar und zeigt, dass in Abhängigkeit von der

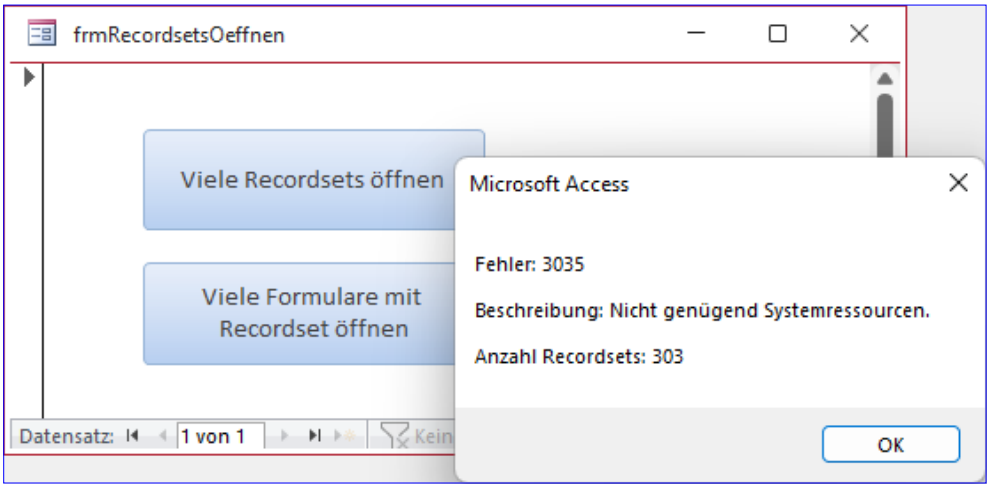


Bild 1: Anzeige der Fehlermeldung nach dem Öffnen von 305 Recordsets

aktuellen Konfiguration des Systems und vermutlich auch abhängig von der aktuellen Auslastung mehr oder weniger **Recordset**-Objekte erstellt werden können.

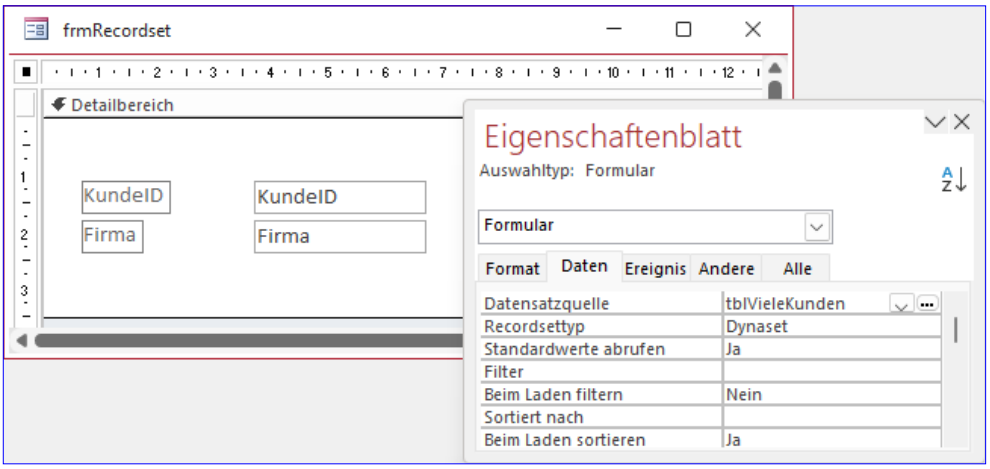


Bild 2: Das an eine Tabelle gebundene Formular

Fehler »Nicht genügend Speicherplatz zum Ausführen der Operation« reproduzieren

Diesen Fehler konnten wir reproduzieren, indem wir einige Formularinstanzen aufgerufen haben. Da wir keine Lust hatten, viele verschiedene Formulare zu erstellen und diese zu öffnen, haben wir zu Beispielzwecken von einer Technik Gebrauch gemacht, die sonst zur Anzeige mehrerer Instanzen von Formularen mit verschiedenen Datensätzen verwendet wird.

Das Formular ist ein einfaches Formular, das wir über die Eigenschaft **Datensatzquelle** an die Tabelle **tblVieleKunden** gebunden haben (siehe Bild 2). Außerdem haben wir die Eigenschaft **Enthält Modul** im Bereich **Andere** des Eigenschaftenblatts auf **Ja** eingestellt. Das ist nötig, damit wir mehrere Instanzen des Formulars erstellen können.

Die Prozedur finden Sie in Listing 2. Hier deklarieren wir eine Variable für ein **Collection**-Objekt und eines zum Erfassen des aktuellen Formulars. Dabei verwenden wir gleich den Typ des Klassenmoduls dieses Formulars, in diesem Fall **Form_frmRecordset**.

Danach instanziieren wir die Collection und starten in eine **For...Next**-Schleife, wie wir wieder bis zu 10.000 Mal durchlaufen. Das sollte reichen, um auch die beste Konfiguration in die Knie zu zwingen. Innerhalb der Schleife deaktivieren wir direkt die eingebaute Fehlerbehandlung. In der folgenden Anweisung weisen wir der Variablen **frm** eine neue Instanz der Formularklasse **Form_frmRecordset** zu.

Ist dabei ein Fehler aufgetreten, wird dieser innerhalb der folgenden **If...Then**-Bedingung mithilfe eines Meldungsfensters ausgegeben. Auch hier geben wir wieder die Fehlernummer, die Fehlerbeschreibung und die Anzahl der bisher geöffneten Formulare aus (siehe Bild 3).

Sollte beim Erstellen der aktuellen Formularinstanz kein Fehler aufgetreten sein, fügen wir die Variable **frm** mit dem Verweis auf diese Instanz der Collection **col** hinzu und laufen mit der **For...Next**-Schleife in den nächsten Durchgang.

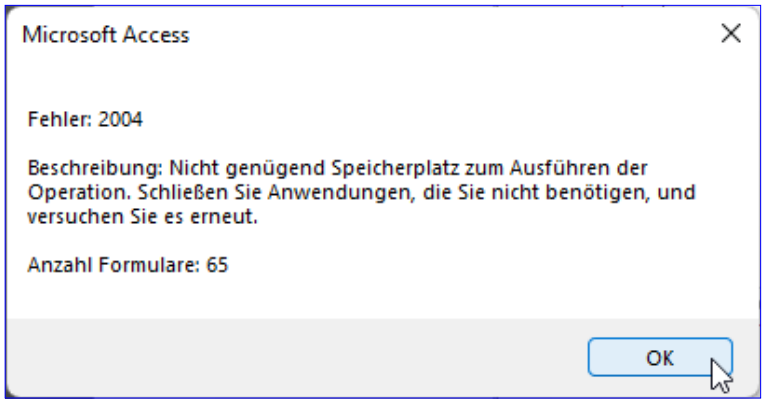


Bild 3: Fehlermeldung bei zu vielen per Formular geöffneten Recordsets

Fehlerhafte Access/Office-Updates zurücksetzen

Im November 2022 war es mal wieder soweit: Microsoft hat ein neues Update (glücklicherweise nur für den Betakanal) herausgegeben, der einen kritischen Fehler auslöste. Beim Bearbeiten oder Anlegen eines Datensatzes über ein Recordset stürzte Access ohne weitere Rückmeldung ab. Die Lösungen für diese Problem waren, solche Bearbeitungen durch alternative Techniken zu ersetzen oder aber das Update rückgängig zu machen. Letztere Alternative beschreiben wir in diesem Beitrag, damit Sie zukünftig eine schnelle Lösung zur Hand haben.

Wenn man feststellt, dass nach einem Update etwas nicht mehr so läuft wie gewünscht, ist die Wahrscheinlichkeit hoch, dass es etwas mit diesem Update zu tun hat. In diesem Fall provozierte folgender Code einen Absturz von Access, und zwar in der vorletzten Zeile:

```
Dim db As DAO.Database
Dim rst As DAO.Recordset
Set db = CurrentDb
Set rst = db.OpenRecordset("SELECT * FROM Tabelle1", _
    dbOpenDynaset)
rst.AddNew
rst!Feld1 = "Test"
rst.Update
```

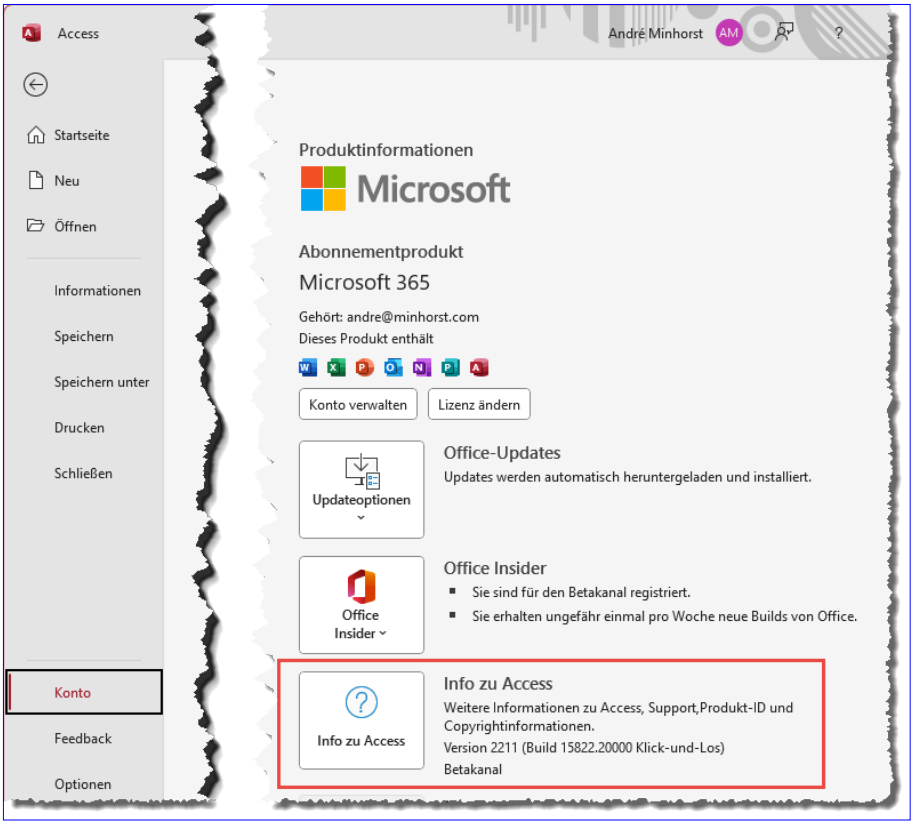


Bild 1: Ermitteln der aktuellen Version

Update rückgängig machen

Wenn man eine Ahnung hat, dass dieses Verhalten durch ein Update ausgelöst wurde, gibt es die Möglichkeit, dieses zurückzusetzen. Dazu haben wir zwei Wege:

Der erste führt über die Wiederherstellungs-Funktion der Systemsteuerung. Diese hat bei mir in diesem Fall funktioniert. Allerdings muss man beachten, dass hier auch andere Änderungen am System rückgängig gemacht werden, die seit dem angestrebten Wiederherstellungspunkt durchgeführt wurden.

Diese Variante würde ich also nur wählen, wenn die zweite, nachfolgend vorgestellte Variante nicht funktioniert. Anleitungen zur

Wiederherstellung über die Systemsteuerung finden sich im Internet.

Die zweite Variante ist das gezielte Zurücksetzen der Office-Version. Dazu muss man herausfinden, welche Version aktuell vorliegt und den Fehler verursacht und dann ermitteln, welche Version die vorherige ist. Wie das gelingt, zeigen wir in den folgenden Abschnitten.

Wiederherzustellende Office-Version ermitteln

Um die Office-Version rückgängig zu machen und die dazu benötigte Versionsnummer zu ermitteln, öffnen wir beispielsweise unter Access zuerst den Bereich **DateilKonto**.

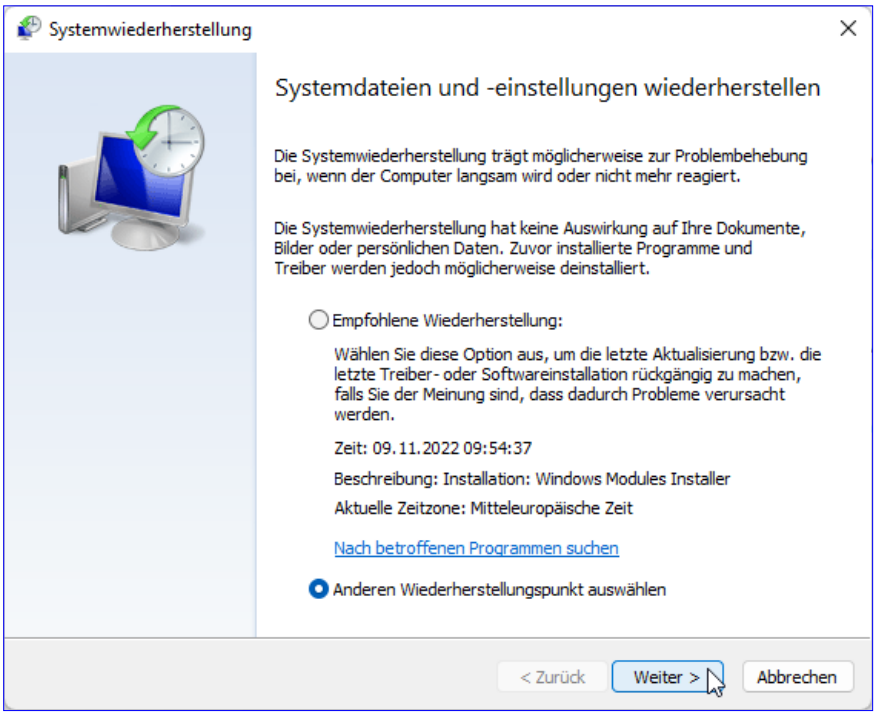


Bild 2: Anzeige anderer Wiederherstellungspunkte

Hier sehen wir rechts unten den Bereich **Info zu Access**, wo wir die Version ablesen können. Die problembehaftete Version lautete in diesem Fall **Version 2211 (Build 16.0.15831.20002 Klick-und-Los)**.

In Bild 1 sehen wir bereits die Versionsnummer nach dem Zurücksetzen. Wichtig ist auch die Information, dass es sich hier um den **Betakanal** handelt.

Um zu einer vorherigen Version von Office zurückzukehren, benötigen wir die genaue Angabe der wiederherzustellenden Version, also beispielsweise **16.0.15822.20000**. Wie finden wir die gesuchte Version heraus? Hier greifen wir nun doch auf die Systemwiederherstellung zu, denn diese enthält die gesuchten Informationen.

Um die Systemwiederherstellung zu starten, geben wir im **Suchen**-Feld von Windows den Begriff **Wiederherstellung** ein und wählen den nun erscheinenden Eintrag **Wiederherstellung** aus. Dies öffnet das Hauptmenü für die Wiederherstellung. Hier klicken Sie auf den Link

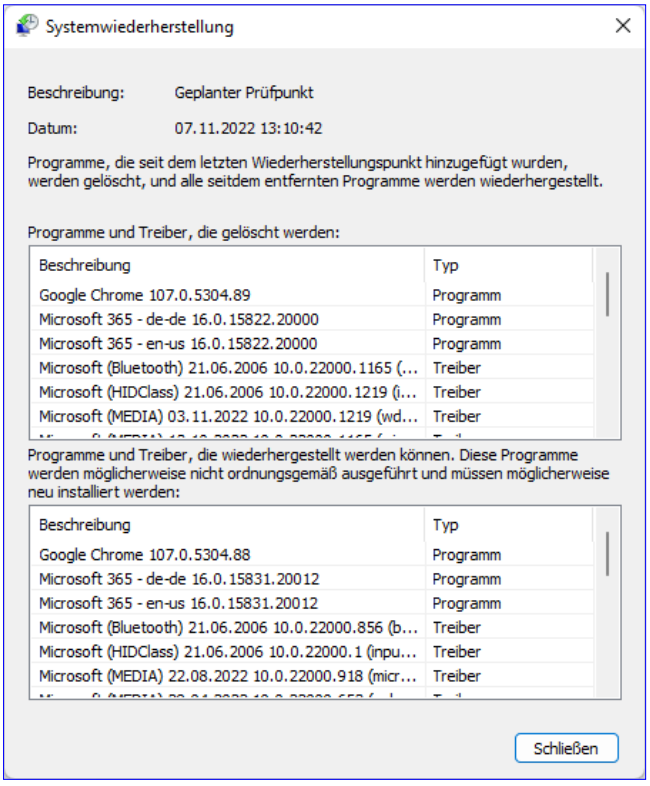


Bild 3: Versionen, die durch diese Wiederherstellung wiederhergestellt werden

Datensatznavigation per Ribbon

Die Navigationszeile am unteren Rand von Formularen oder Unterformularen in der Datenblattansicht ist recht fummelig und fällt nicht auf den ersten Blick auf, wenn man nicht gewohnt ist, damit zu arbeiten. Es gibt eine Menge Lösungen, bei denen diese Steuerelemente in Form von Formular-Schaltflächen bereitgestellt werden. Uns ist aber noch keine Lösung über den Weg gelaufen, bei der die Navigationssteuerelemente im Ribbon abgebildet wurden. Zwar gibt es einen eigenen Bereich, der Werkzeuge für den Umgang mit der Datenblattansicht bereitstellt, aber dieser bietet keine Steuerelemente zum Navigieren in den Datensätzen. Aber kein Problem: Wir liefern das nach!

Wenn wir eine Tabelle in einem Formular in der Datenblattansicht anzeigen wie in Bild 1, sehen wir am unteren Rand des Formulars die Navigationssteuerelemente – also Schaltflächen und ein Textfeld mit den folgenden Funktionen:

- Ersten Datensatz anzeigen/markieren
- Vorherigen Datensatz anzeigen/markieren
- Ausgabe des aktuellen Datensatzes und der gesamten Anzahl samt Eingabe des Datensatzindex, zu dem navigiert werden soll
- Nächsten Datensatz anzeigen/markieren
- Letzten Datensatz anzeigen/markieren
- Neuen Datensatz anzeigen/markieren

Es wäre doch praktisch, wenn diese wichtigen Steuerelemente nicht klein und ganz unten angezeigt würden, sondern oben prominent im Ribbon – sodass der Benutzer diese nicht übersehen kann!

Also schauen wir uns in diesem Beitrag an, wie wir dies realisieren können.

Verhalten der Navigationssteuerelemente untersuchen

Da wir das Verhalten der oben genannten Steuerelemente möglichst genau abbilden wollen, müssen wir es zunächst studieren. In der Regel öffnet man ein Formular mit Daten und erhält den ersten Datensatz der Datensatzquelle. In diesem Zustand sind alle Steuerelemente mit Ausnahme der Schaltfläche zum Anzeigen des vorherigen Datensatzes aktiviert. Das ergibt Sinn, denn man kann nicht zu

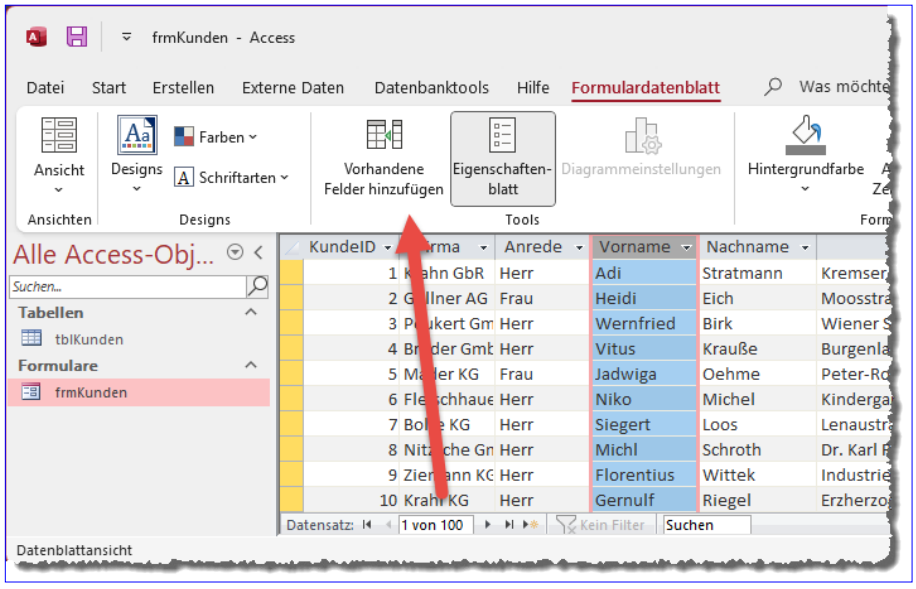


Bild 1: Daten in der Datenblattansicht

einem früheren Datensatz springen, wenn schon der erste Datensatz angezeigt wird. Aber kann man zum ersten Datensatz springen, wenn schon der erste angezeigt wird? Man könnte sich streiten, ob die Schaltfläche zum Ansteuern des ersten Datensatzes aktiviert sein muss, wenn der erste Datensatz angezeigt wird, aber Tatsache ist: Unter Access ist es so.

Die nächste zu untersuchende Situation ist die, wenn sich der Datensatzzeiger irgendwo zwischen dem ersten und dem letzten Datensatz befindet. Hier sind immer alle Schaltflächen aktiviert.

Das ist auch bei der Anzeige des letzten Datensatzes der Fall! Warum aber ist hier die Schaltfläche zum Anzeigen des nächsten Datensatzes aktiviert, obwohl wir ja eigentlich nicht mehr zu einem weiteren Datensatz springen können?

Ganz einfach: Unter Access können wir in dieser Situation noch zu einem neuen, leeren Datensatz springen, wenn wir diese Schaltfläche anklicken.

Sonderfall nicht aktualisierbare Datensatzquelle

Eine Situation führt übrigens dazu, dass auch die Schaltfläche zum Anzeigen eines neuen Datensatzes einmal deaktiviert ist: dann nämlich, wenn das Formular Daten anzeigt, die nicht aktualisierbar sind.

Und dann wird auch die Schaltfläche zum Anzeigen des nächsten Datensatzes deaktiviert, wenn sich der Datensatzzeiger auf dem letzten Datensatz befindet.

Dieses Verhalten können wir für ein Formular übrigens relativ einfach abbilden – wir müssen dazu nur die Eigenschaft **Recordsettyp** auf **Snapshot** einstellen oder eine nicht aktualisierbare Abfrage zusammenstellen.

Die hier definierten Regeln wollen wir beim Programmieren der Ribbonanpassung zum Steuern der Datensatzanzeige berücksichtigen.

Aussehen des Ribbons definieren

Das geplante Ribbon soll wie in Bild 2 aussehen. Es enthält also fünf Schaltflächen sowie ein **EditBox**-Element, das die aktuelle Position des Datensatzzeigers anzeigt und auch die Eingabe einer neuen Position erlaubt.

Was wir hier nicht beschreiben, aber dennoch benötigt wird und in der Beispieldatenbank enthalten ist:

- das Modul **mdlRibbonImages**, das Funktionen enthält, mit denen wir die Bilder aus der Tabelle **MSysResources** auslesen und in geeigneter Form für die Anzeige im Ribbon bereitstellen

- die Funktion **LoadImages** im Modul **mdlRibbons**, welches wir für das Attribut **loadImages** des **customUI**-Elements des Ribbons hinterlegen und die dafür sorgt, dass für jedes Element, welches das **image**-Attribut enthält, das entsprechende Bild aus der Tabelle **MSysResources** geladen wird.

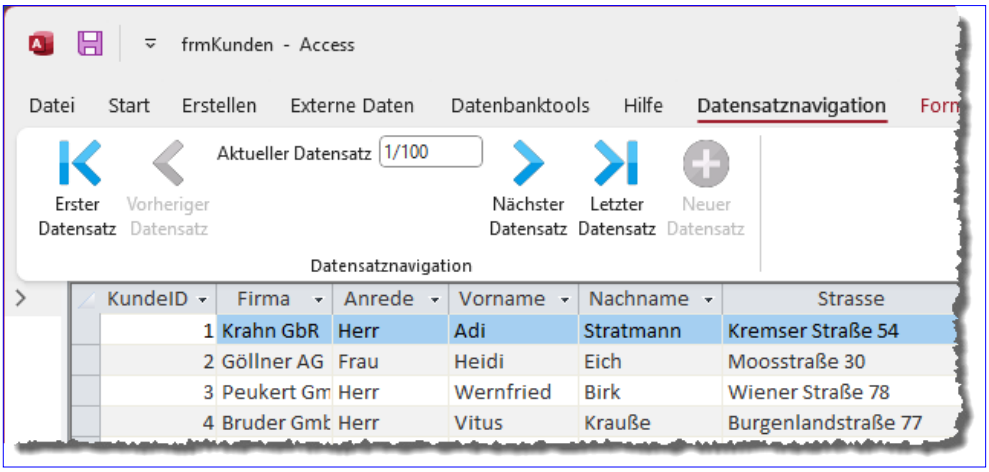


Bild 2: So soll das Ribbon zur Datensatznavigation aussehen.

Davon ab nutzen wir eine Tabelle namens **USysRibbons**, in der wir die Ribbondefinition im XML-Format speichern.

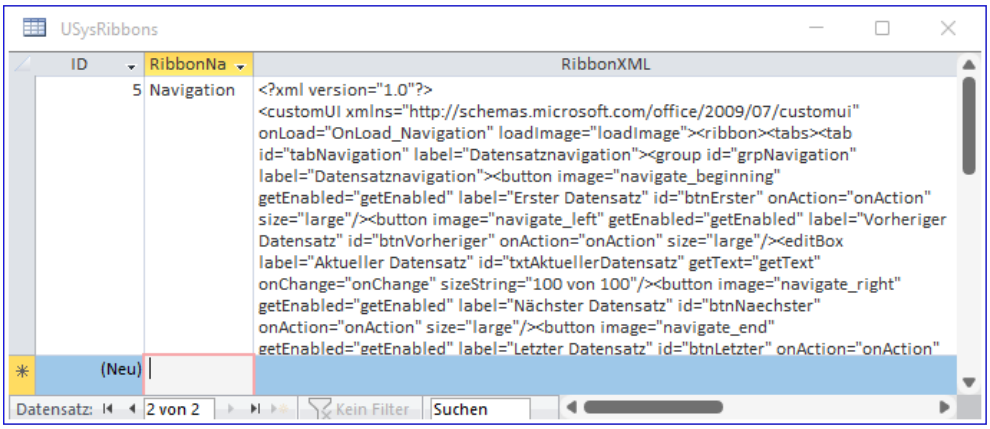


Bild 3: Tabelle zum Speichern der Ribbondefinition

Diese enthält drei Felder, nämlich **RibbonID** (Primärschlüsselfeld mit Autowert), **RibbonName** (Datentyp **Kurzer Text**) und **RibbonXML** (Datentyp **Langer Text**). Darin speichern wir die nötigen Informationen wie in Bild 3.

Nun schauen wir uns die darin enthaltene Ribbondefinition an. Diese haben wir in Listing 1 abgebildet. Das Element **customUI** enthält die beiden Attribute **onLoad** und **loadImage**, welche die beim Laden des Ribbons und für die Anzeige von Bildern benötigten VBA-Prozeduren angeben. Die für **onLoad** hinterlegte Prozedur **OnLoad_Navigation** wird direkt beim Laden ausgelöst, die für **loadImage** hinterlegte für jedes Element, welches das Attribut **image** aufweist und somit ein Bild anzeigen soll.

```
<?xml version="1.0"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" onLoad="OnLoad_Navigation" loadImage="loadImage">
    <ribbon>
        <tabs>
            <tab id="tabNavigation" getLabel="getLabel">
                <group id="grpNavigation" label="Datensatznavigation">
                    <button image="navigate_beginning" getEnabled="getEnabled" label="Erster Datensatz"
                        id="btnErster" onAction="onAction" size="large"/>
                    <button image="navigate_left" getEnabled="getEnabled" label="Vorheriger Datensatz"
                        id="btnVorheriger" onAction="onAction" size="large"/>
                    <editBox label="Aktueller Datensatz" id="txtAktuellerDatensatz" getText="getText"
                        onChange="onChange" sizeString="100 von 100" getEnabled="getEnabled"/>
                    <button image="navigate_right" getEnabled="getEnabled" label="Nächster Datensatz"
                        id="btnNaechster" onAction="onAction" size="large"/>
                    <button image="navigate_end" getEnabled="getEnabled" label="Letzter Datensatz"
                        id="btnLetzter" onAction="onAction" size="large"/>
                    <button image="add" getEnabled="getEnabled" label="Neuer Datensatz" id="btnNeu"
                        onAction="onAction" size="large"/>
                </group>
            </tab>
        </tabs>
    </ribbon>
</customUI>
```

Listing 1: Definition des Ribbons für die Datensatznavigation

Vorbereitung zum Aktualisieren der Ribbonelemente

Wenn wir zur Laufzeit die Eigenschaften von Ribbonelementen aktualisieren wollen, was hier absehbar ist, weil Schaltflächen aktiviert und deaktiviert werden sollen und auch das **editBox**-Element die jeweilige Position des Datensatzzeigers anzeigen soll, müssen wir eine Variable des Typs **IRibbonUI** deklarieren. Dieses bietet die Methode **Invalidate** an, mit der wir dafür sorgen können, dass die Attribute wie **getEnabled**, **getText** et cetera ihre Werte per Callbackprozedur erneut abrufen. Diese Variable deklarieren wir wie folgt:

```
Public objRibbon_Navigation As IRibbonUI
```

Außerdem implementieren wir die für das Attribut **onLoad** hinterlegte Prozedur wie folgt, damit diese mit der Variablen **objRibbon_Navigation** gleich beim Laden einen Verweis auf die angewendete Ribbondefinition hinterlegt:

```
Sub onLoad_Navigation(ribbon As IRibbonUI)
    Set objRibbon_Navigation = ribbon
End Sub
```

Unterhalb des **customUI**-Elements befinden sich die obligatorischen **ribbon**-, **tabs**- und **tab**-Elemente. Das **tab**-Element mit der Bezeichnung **tabNavigation** enthält das Callback-Attribut **getLabel**, welches die Beschriftung des **tab**-Elements ermitteln soll. Diese Beschriftung soll entweder den Wert der Eigenschaft **Beschriftung** des Formulars enthalten oder, wenn diese nicht angegeben wurde, den Namen des Formulars als **tab**-Beschriftung anzeigen.

tab-Beschriftung ermitteln

Die für das Attribut **getLabel** hinterlegte Prozedur wird direkt beim Anzeigen des Ribbons aufgerufen und sieht wie folgt aus:

```
Sub getLabel(control As IRibbonControl, ByRef label)
    Dim frm As Form
    Select Case control.ID
        Case "tabNavigation"
            Set frm = Screen.ActiveForm
            If Len(frm.Caption) = 0 Then
                label = frm.Name
            Else
                label = frm.Caption
            End If
    End Select
End Sub
```

Sie prüft per **Select Case**, ob sie für das Element **tabNavigation** aufgerufen wurde und referenziert dann das aktuelle Formular. Wenn wir, wie geplant, diese Ribbondefinition als Ribbon für ein Formular angeben, ist auf jeden Fall ein Formular geöffnet, wenn das Ribbon erscheint. Für dieses prüfen wir dann, ob die Eigenschaft **Caption** leer ist. Ist das der Fall, gibt die Prozedur den Namen des Formulars zurück, anderenfalls die Beschriftung aus der Eigenschaft **Caption**.

Gruppe mit Steuerelementen

Unter dem **tab**-Element folgt das **group**-Element, das die eigentlichen Steuerelemente enthält. Die **button**-Elemente erhalten jeweils den Namen des anzuzeigenden Icons für das **image**-Attribut, zum Beispiel **navigate_beginning**. Außerdem zeigen sie eine Beschriftung an, die wir für das Attribut **label** angeben und ein eindeutiger Bezeichner, der im Attribut **id** landet, darf auch nicht fehlen.

Schließlich enthält jedes **button**-Element noch zwei Attribute, für die wir VBA-Prozeduren hinterlegen:

- Für das Attribut **onAction** hinterlegen wir die Prozedur, die beim Anklicken ausgelöst werden soll. Diese heißt **OnAction**.
- Für das Attribut **getEnabled** hinterlegen wir eine Prozedur, die anhand der aktuellen Position des Datensatzzeigers und weiterer Parameter prüfen soll, ob die Schaltfläche derzeit aktiviert sein darf.

Anzeige der Datensatzzeigerposition

In einem Textfeld wollen wir einen Wert wie **1/100** anzeigen und somit die Position des Datensatzzeigers sowie die Anzahl der Datensätze darstellen. Das erledigen wir mit einem **editBox**-Element. Für dieses hinterlegen wir auch drei Callback-Attribute:

- Für **getText** hinterlegen wir die Prozedur, welche die aktuelle Position des Datensatzzeigers und die Anzahl der Datensätze liefert.
- Für **onChange** geben wir die Prozedur an, die ausgelöst wird, wenn der Benutzer einen Zahlenwert eingibt, um den Datensatzzeiger zu der angegebenen Position zu bewegen.
- Für **getEnabled** hinterlegen wir eine Prozedur, die ermittelt, ob das **editBox**-Element aktiviert sein soll. Das ist eigentlich nur nicht der Fall, wenn das Recordset leer ist und nicht bearbeitet werden kann.

Bilder bereitstellen

Die fünf Bilddateien, welche die Schaltflächen als Icon anzeigen sollen, speichern wir in der Tabelle **MSysResources**. Um diese dort zu hinterlegen, gibt es einen einfachen Trick: Wir zeigen ein Formular in der Entwurfsansicht an, klicken auf den Ribbonbefehl zum Hinzufügen eines Bild-Steuerelements, wählen das Bild mit dem dann erscheinenden Dateiauswahl-Dialog aus und brechen dann das Hinzufügen des Bild-Steuerelements ab.

Damit legen wir zwar kein Bild-Steuerelement an, aber das soeben ausgewählte Bild wird in der Tabelle **MSysResources** gespeichert. Diesen Schritt wiederholen wir einfach für alle anzuzeigenden Bilder, bis die Tabelle **MSysResources** wie in Bild 4 aussieht. Diese Tabelle wird im Navigationsbereich übrigens nur erscheinen, wenn die Anzeige von Systemobjekten und ausgeblendeten Objekten aktiviert ist. Man kann sie aber auch mit dem folgenden Befehl öffnen (das gilt auch für die Tabelle **USysRibbons**):

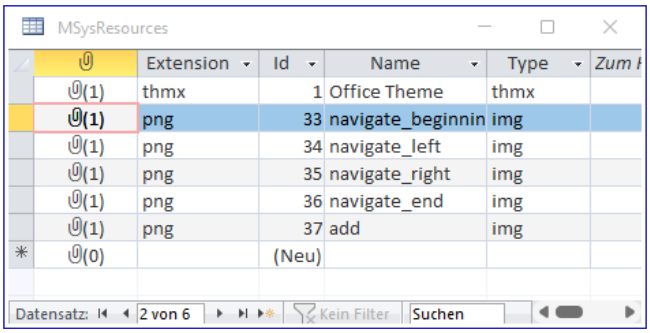


Bild 4: Tabelle zum Speichern der Bilder, die als Icons der Schaltflächen im Ribbon erscheinen sollen

```
DoCmd.OpenTable "MSysResources"
```

Für die Tabelle **USysRibbons** analog:

```
DoCmd.OpenTable "USysRibbons"
```

Das die Bilder letztlich angezeigt werden, erreichen wir durch die Kombination der Angabe des Wertes aus dem Feld **Name** für das Attribut **image** der **button**-Elemente sowie der Verwendung der für das Attribut **loadImage** des **customUI**-Elements hinterlegten Callbackprozedur.

Text für die Anzeige der Position des Datensatzzeigers anzeigen

Damit das **editBox**-Element einen Text wie **1/100** anzeigt und diese Anzeige zu gegebenen Zeitpunkten aktualisiert, müssen wir eine entsprechende Callbackprozedur hinterlegen. Diese sieht wie folgt aus:

```
Sub getText(control As IRibbonControl, ByRef text)
    Dim frm As Form
    Dim rst As DAO.Recordset
    Set frm = Screen.ActiveForm
    Set rst = frm.Recordset
    If Not rst.Updatable And rst.RecordCount = 0 Then
        text = ""
    Else
        If Not frm.NewRecord Then
            text = rst.AbsolutePosition + 1 & "/" _
                & rst.RecordCount
```

Eigenschaftsassistenten oder Property Wizards

Vielleicht kennen Sie die kleinen Helferlein, die beim Anklicken einer Eigenschaft eines Formulars oder Steuerelements in Form einer Schaltfläche mit drei Punkten (...) auftauchen und beim Anklicken Unterstützung für das Füllen der jeweiligen Eigenschaft präsentieren. Wenn wir schon Access-Add-Ins, Tabellen-, Abfrage-, Formular und Berichtsassistenten sowie Steuerelement-Assistenten programmieren können, warum dann nicht auch noch Eigenschaftsassistenten? In diesem Beitrag zeigen wir die Grundlagen für das Programmieren eines Assistenten, der beim Anklicken der genannten Schaltfläche erscheint und seine Hilfe anbietet.

Eingebaute Eigenschaftsassistenten

Beispiele für eingebaute Assistenten findet man schnell. Die **Bild**-Eigenschaft von Formularen oder Steuerelementen bietet die Schaltfläche mit den drei Punkten an, um die anzuzeigende Bilddatei auszuwählen.

Ein anderes Beispiel, das Sie in Bild 1 sehen, wird beim Anklicken der Eigenschaft **Steuerelementinhalt** aktiviert.

Dieses öffnet den Ausdrucks-Generator aus Bild 2, mit dem wir den Ausdruck für die entsprechende Eigenschaft aus verschiedenen Listen und Optionen zusammenklicken können.

Eigenschaftsassistent im Eigenbau

Wie aber bauen wir selbst einen solchen Assistenten? Dazu sind die folgenden Schritte nötig:

- Erstellen einer Access-Datenbank mit der Dateiendung **.accda**
- Hinzufügen einer Tabelle namens **USysRegInfo**, welche die Informationen über das Add-In zum Einfügen in die Registry enthält
- Hinzufügen einer VBA-Funktion, die beim Aufrufen des Add-Ins gestartet wird und welche beispielsweise die Benutzeroberfläche des Add-Ins anzeigt
- Programmieren der eigentlichen Funktion des Add-Ins inklusive Benutzeroberfläche
- Installieren des Add-Ins mit dem Add-In-Manager von Access

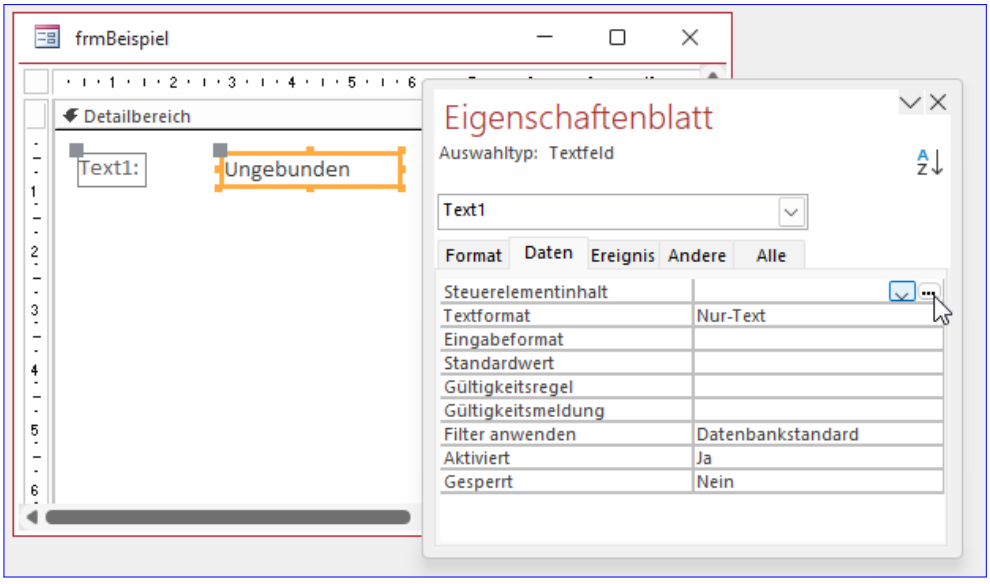


Bild 1: Starten eines Eigenschaftsassistenten

Aufgabe unseres Beispiel-Eigenschaftsassistenten

Damit der Beitrag nicht den Rahmen sprengt, wollen wir in diesem erst einmal einen rudimentären Eigenschaftsassistenten erstellen. Dieser soll ein einfaches Formular anzeigen, welches den Wert für das Add-In per Textfeld abfragt.

In einem weiteren Beitrag schauen wir uns ein konkretes und in der Praxis nutzbares Beispiel für einen Eigenschaftsassistenten an (siehe Ausblick am Ende des Beitrags).

Aufbau der Startfunktion des Assistenten

Die VBA-Funktion, die wir oben erwähnt haben, muss ein bestimmtes Format aufweisen.

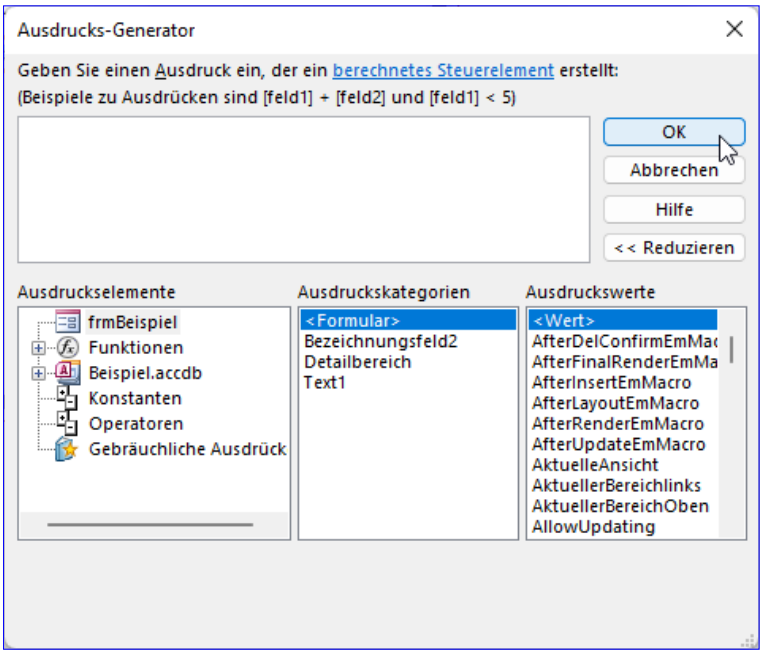


Bild 2: Der Ausdrucks-Generator

Der Name der Funktion und der drei Parameter spielt keine Rolle, aber wir müssen drei **String**-Parameter deklarieren und die Funktion muss das Ergebnis des Assistenten als String zurückgeben.

Ein Beispiel für diese Funktion lautet:

```
Public Function Autostart(strObject As String, _
    strControl As String, strCurVal As String) As String

End Function
```

Die drei Parameter werden beim Aufrufen des Assistenten mit drei Werten gefüllt:

- **strObject**: Name des Objekts, also Formulars oder Berichts, für welches der Assistent aufgerufen wurde
- **strControl**: Name des Steuerelements, zu dem die Eigenschaft mit dem zu ermittelnden Wert gehört
- **strCurVal**: Aktueller Wert der Eigenschaft für das mit **strObject** und **strControl** angegebene Element

Aufbau des Formulars für die Benutzeroberfläche des Assistenten

Als Benutzeroberfläche werden Sie typischerweise ein Formular verwenden. Die wichtigste Information hierzu ist, dass Sie es mit der oben genannten Funktion aufrufen können und beim Schließen des Formulars den Wert für die Eigenschaft daraus auslesen müssen.

Dazu ist es erforderlich, dass das Formular als modaler Dialog geöffnet wird, also beispielsweise wie folgt:

```
DoCmd.OpenForm "frmAssistent", WindowMode:=acDialog
```

Auf diese Weise wird der aufrufende Code angehalten, wenn das Formular geöffnet wird. Nach der Eingabe der gewünschten Informationen wird der Benutzer das Formular dann beispielsweise über eine **OK**-Schaltfläche schließen, was das Formular aber nicht tatsächlich schließen, sondern nur ausblenden sollte.

Auf diese Weise läuft die Funktion, die das Formular geöffnet hat, nun weiter und kann den relevanten Wert einlesen und dann das Formular schließen.

Und die Vorgabe, dass das Formular als modaler Dialog geöffnet wird, hat noch einen anderen Hintergrund: Dadurch, dass der Benutzer so bis zum Schließen des Assistenten nicht auf die übrigen Elemente von Access zugreifen kann, wird sichergestellt, dass bei Schließen des Assistenten die Eigenschaft, die dieser aktualisieren soll, auch noch geöffnet ist.

Speichern der Assistenten-Informationen für die Registry

Access-Add-Ins und Assistenten werden in der Regel über den Add-In-Manager von Access installiert, den Sie über den Ribbonbefehl **Datenbanktools|Add-Ins|Add-Ins-Add-In-Manager** aufrufen können.

Hier wählen Sie dann die **.accda**-Datei des zu installierenden Add-Ins oder Assistenten aus.

Anschließend wertet der Add-In-Manager die Einträge in der Tabelle **USysRegInfo** aus und trägt diese an entsprechender Stelle in die Registry ein. Der Ort, wo diese Eintragungen vorgenommen werden, legt fest, um was für ein Add-In oder was für einen Assistenten es sich handelt, die Einträge selbst geben an, in welcher Datei sich die Funktionen befinden, ob ein Icon angezeigt werden soll, wie die Funktion zum Starten des Add-Ins oder Assistenten heißt und wie die Beschreibung des Tools lautet.

Der Ort für Eigenschaftsassistenten in der Registry heißt **HKEY_CURRENT_USER\Software\MIcrosoft\Office\x.0\Access\Wizards\Property Wizards\[Name des Assistenten]**. Darunter befinden sich Einträge mit den Registry-Werten für unseren Assistenten.

Mögliche Registry-Schlüssel

Für einen Eigenschaftsassistenten können wir die folgenden Schlüssel festlegen:

- **Can Edit** (**DWORD**): Gibt an, ob ein vorhandener Eigenschaftswert geändert werden soll. Wird üblicherweise auf **1** eingestellt.

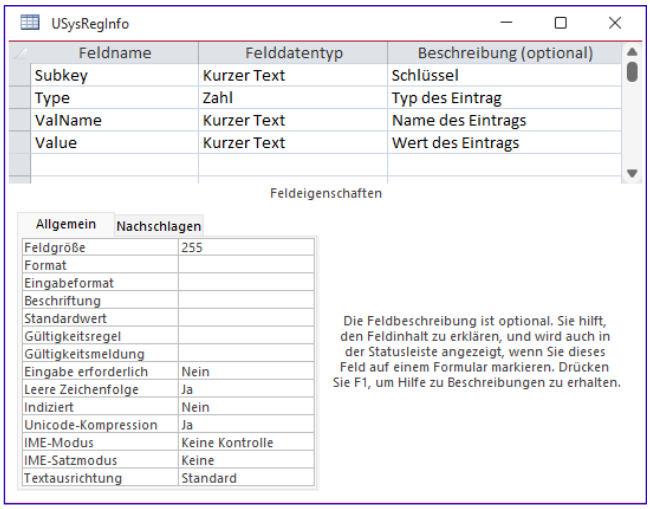


Bild 3: Die Tabelle **USysRegInfo**

- **Description** (**String**): Beschreibung des Assistenten, die nur dann angezeigt wird, wenn mehrere Add-Ins für eine Eigenschaft vorliegen und dazu der Auswahldialog geöffnet wird.

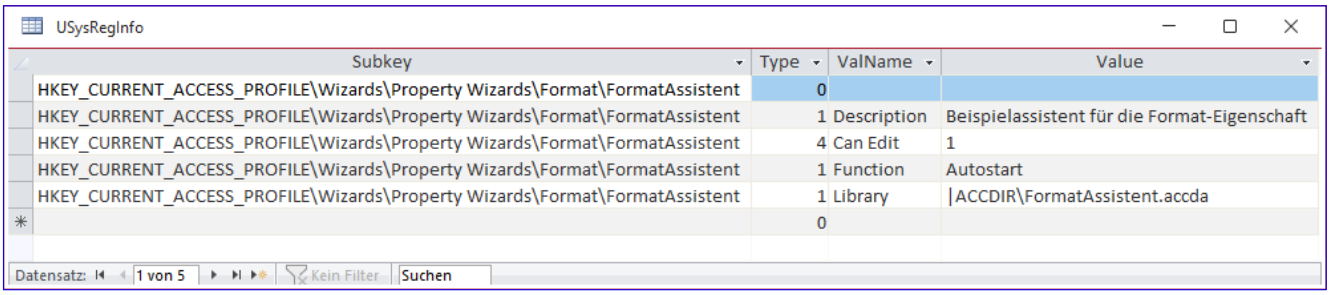
- **Function** (**String**): Name der aufzurufenden VBA-Funktion, in unserem Fall **Autostart**, ohne Gleichheitszeichen oder Klammern.

- **Library** (**String**): Pfad zur **.accda**-Datei mit den Funktionen des Eigenschaftsassistenten.

Aufbau der Tabelle USysRegInfo

Die Tabelle mit den Informationen für die Registry muss den Namen **USysRegInfo** erhalten, damit diese vom Add-In-Manager automatisch erkannt werden kann. Gleichzeitig beginnt der Name dieser Tabelle mit **USys...**, was neben dem Präfix **MSys...** dazu führt, dass diese Tabellen standardmäßig nicht im Navigationsbereich von Access angezeigt werden.

Wenn Sie also eine solche Tabelle mit den Feldern aus Bild 3 angelegt, gespeichert und geschlossen haben, wird diese möglicherweise komplett verschwinden. Um diese wieder einzublenden, klicken Sie mit der rechten Maustaste auf die Titelzeile des Navigationsbereichs und wählen dort den Kontextmenübefehl **Navigationsoptionen...** aus.



USysRegInfo

Subkey	Type	ValName	Value
HKEY_CURRENT_ACCESS_PROFILE\Wizards\Property Wizards\Format\FormatAssistent	0		
HKEY_CURRENT_ACCESS_PROFILE\Wizards\Property Wizards\Format\FormatAssistent	1	Description	Beispielassistent für die Format-Eigenschaft
HKEY_CURRENT_ACCESS_PROFILE\Wizards\Property Wizards\Format\FormatAssistent	4	Can Edit	1
HKEY_CURRENT_ACCESS_PROFILE\Wizards\Property Wizards\Format\FormatAssistent	1	Function	Autostart
HKEY_CURRENT_ACCESS_PROFILE\Wizards\Property Wizards\Format\FormatAssistent	1	Library	|ACCDIR\FormatAssistent.accda
	0		

Datensatz: 1 von 5 | Kein Filter | Suchen

Bild 4: Die Tabelle **USysRegInfo** in der Datenblattansicht

Hier aktivieren Sie die Option Systemobjekte anzeigen. Anschließend wird die Tabelle **USysRegInfo** neben einigen anderen zuvor ausgeblendeten Tabellen eingeblendet.

Füllen der Tabelle USysRegInfo

Nun fügen wir die oben beschriebenen Informationen für die Registry in die Tabelle ein – mit je einem Datensatz pro Information. Das Ergebnis sehen sie in Bild 4. Hier sind zwei Informationen wichtig:

Der Platzhalter **HKEY_CURRENT_ACCESS_PROFILE** wird beim Installieren des Assistenten durch den Registrypfad für die aktuelle Office-Installation ersetzt. In unserem Fall (Windows 64-Bit, Office 32-Bit) bedeutete dies den folgenden Schlüssel:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Office\ClickToRun\
REGISTRY\MACHINE\Software\Wow6432Node\Microsoft\Of-
fice\16.0\Access\Wizards\Property Wizards\Format\FormatAs-
sistent
```

Der Platzhalter **|ACCDIR** wird durch das Add-In-Verzeichnis von Access ersetzt, in aktuellen Kombinationen aus Windows und Office also durch **C:\Users\[Benutzername]\AppData\Roaming\Microsoft\AddIns**.

Funktion im Assistenten anlegen

Nun legen wir eine kleine Beispielfunktion in Form der Funktion **Autostart** an. Diese soll einfach nur testweise die Werte der Parameter **strObject**, **strControl** und **strCurVal** in einer Meldung ausgeben und den neuen Wert für die Eigenschaft per InputBox erfragen:

```
Public Function Autostart(strObject As String, _
        strControl As String, strCurVal As String) As String
    MsgBox strObject & " " & strControl & " " & strCurVal
    Autostart = InputBox("Eigenschaftswert: ")
End Function
```

Installation über den Add-In-Manager

Um den Assistenten zu installieren, sind Administrator-Rechte erforderlich, wir müssen Access also als Administrator öffnen.

Dazu klicken Sie mit der rechten Maustaste auf den Befehl zum Starten von Access und wählen dort den Eintrag **Als Administrator öffnen** aus.

Danach öffnen wir irgendeine Datenbankdatei, da sonst das Ribbon mit dem Befehl zum Starten des Add-**In-Managers** nicht angezeigt wird.

Nun öffnen wir mit **Datenbanktools|Add-Ins|Add-Ins|Add-In-Manager** den **Add-In-Manager**.

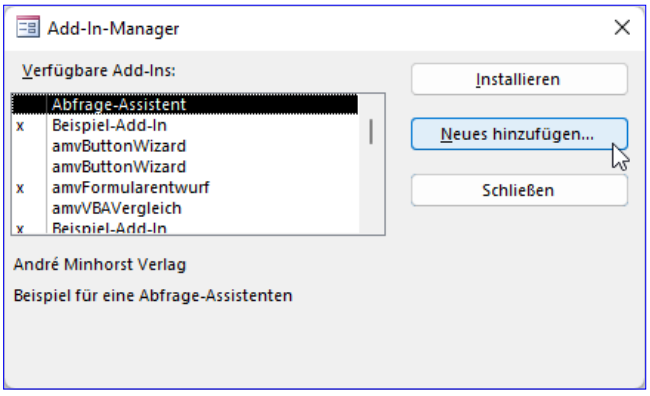


Bild 5: Installieren eines neuen Assistenten

Bilder im Endlosformular

Neulich fragte mich ein Leser, wie er Bilder aus dem Dateisystem, deren Dateiname oder Pfad in einer Tabelle gespeichert sind, in einem Formular in der Endlosansicht darstellen kann. Das ist seit Access 2010 und dem dort aktualisierten Bildsteuerelement gar nicht mal so kompliziert. In diesem Beitrag zeigen wir die notwendigen Schritte, um in einem Formular mehrere Datensätze mit verschiedenen Bildern auf einen Blick anzuzeigen.

Wir schauen uns dabei zwei verschiedene Varianten an:

- Bei der ersten ist für jeden Datensatz der komplette Pfad zu dem anzuzeigenden Bild in einem Feld der Tabelle gespeichert.
- Bei der zweiten finden wir nur noch den Dateinamen in einem Feld der Datenbank. Dabei gehen wir allerdings davon aus, dass sich die Bilder im gleichen Verzeichnis wie die Datenbankdatei befinden oder in einem Verzeichnis unterhalb dieses Verzeichnisses. Dieses Verzeichnis muss jedoch bekannt und relativ zur Datenbankdatei immer gleich sein.

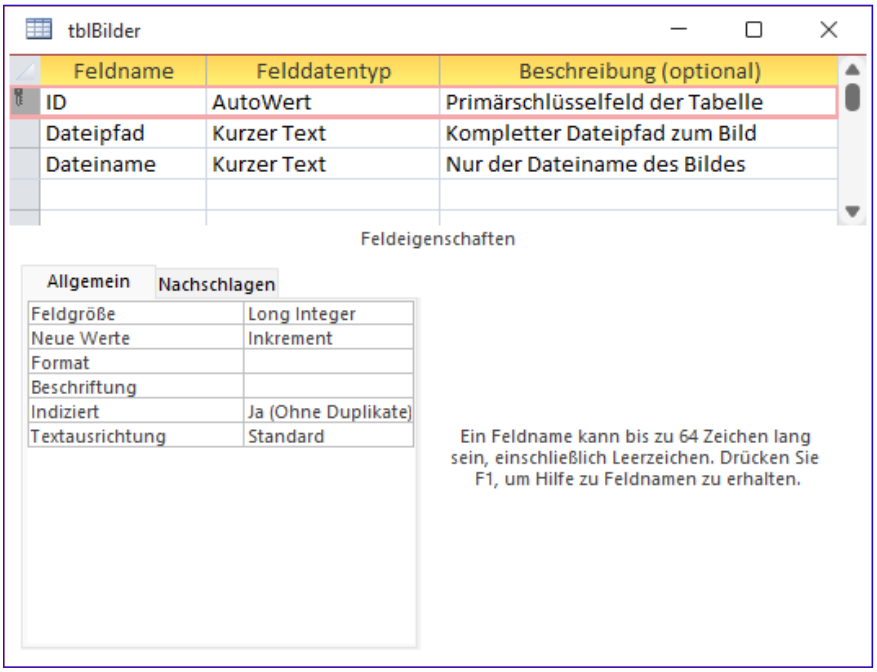


Bild 1: Tabelle mit Bildpfaden und Bilddateinamen in der Entwurfsansicht

Diese beiden verschiedenen Informationen über die anzuzeigenden Bilder speichern wir in den Feldern **Dateipfad** und **Dateiname** der Tabelle **tblBilder**, die in der Entwurfsansicht wie in Bild 1 aussieht.

Wechseln wir dann zur Datenblattansicht und geben einige Daten ein, sieht die Tabelle wie in Bild 2 aus. Der Inhalt des Feldes **Dateiname** entspricht jeweils dem im Feld **Dateipfad** angegebenen Dateinamen.

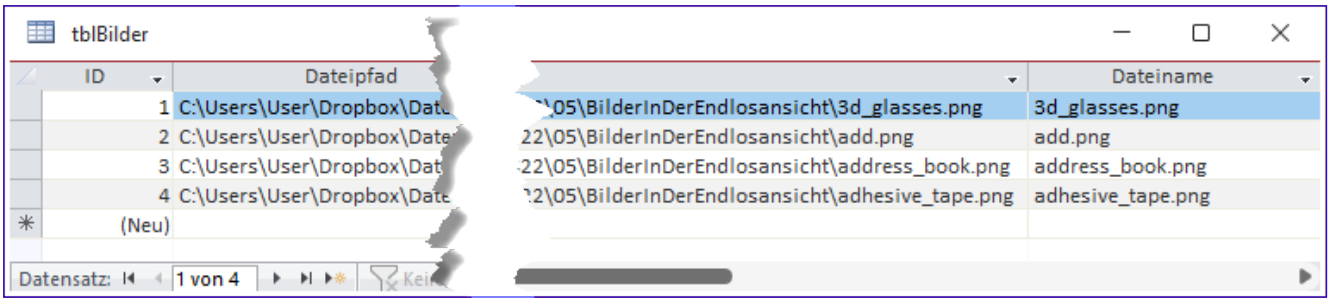


Bild 2: Tabelle mit Bildpfaden und Bilddateinamen in der Datenblattansicht

Die Bilddateien befinden sich in diesem Beispiel im gleichen Verzeichnis wie die Beispieldatenbank. Wenn Sie das Beispiel aus dem Download ausprobieren wollen, müssen Sie also alle Dateien (die **.accdb**-Datenbank und die vier Bilddateien) in das gleiche Verzeichnis entpacken.

Das Feld Dateipfad enthält zu diesem Zeitpunkt natürlich noch die Pfadangaben, die mit der Beispieldatenbank geliefert wurden und die dem Pfad auf dem Rechner entsprechen, mit dem die Beispieldatenbank erstellt wurde.

Die Prozedur, die beim Schließen des **frmIntro**-Formulars der Beispieldatenbank ausgeführt wird, stellt den Inhalt des Feldes **Dateipfad** jedoch auf den Dateipfad der aktuellen Datenbankdatei und dem Bildnamen aus dem Feld **Dateiname** ein:

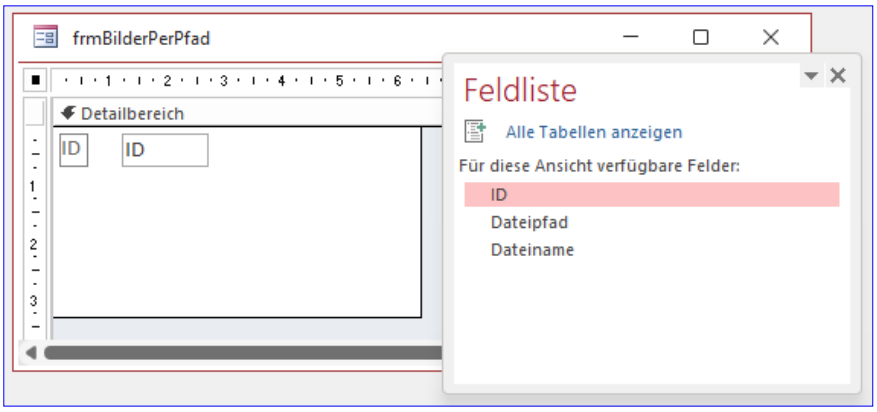


Bild 3: Hinzufügen des Feldes **ID**

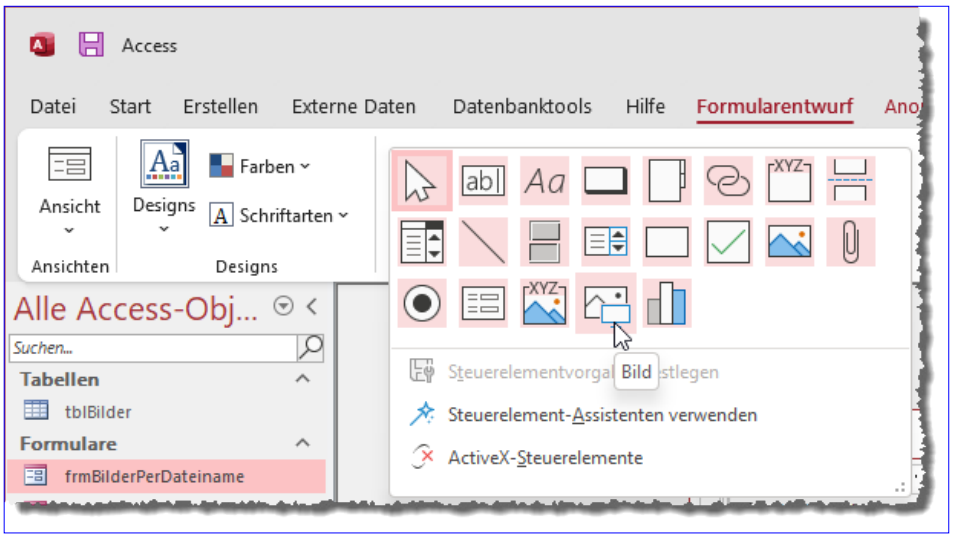


Bild 4: Hinzufügen eines Bildsteuerelements

```
Private Sub cmdGo_Click()
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "UPDATE tblBilder SET Dateipfad = '" _
        & CurrentProject.Path & "\' & Dateiname", _
        dbFailOnError
    ...
End Sub
```

Bilder per Pfad im Endlosformular

Nun legen wir das erste Beispielformular an. Dieses nennen wir **frmBilderPerPfad** und wir weisen ihm als Datensatzquelle die Tabelle **tblBilder** zu.

Danach wechseln wir vom Eigenschaftenblatt zur Feldliste und ziehen das Feld **ID** in das Formular (siehe Bild 3).

Das Bildsteuerelement zum Anzeigen des im Feld **Dateipfad** angegebenen Bildes können wir nicht auf diese Weise zum Formular hinzufügen. Dieses fügen wir in zwei Schritten hinzu:

- Hinzufügen eines Steuerelements des Typs Bild (siehe Bild 4)
- Einstellen der Eigenschaft **Steuerelementinhalt** des Bildsteuerelements auf den Wert **Dateipfad** (siehe Bild 5)

Access und Notion synchronisieren

In weiteren Beiträgen haben wir uns angesehen, wie wir Daten aus Notion abfragen und neu anlegen können. Spannend wird es, wenn wir diese Techniken nutzen, um Daten aus anderen Anwendungen mit Notion zu synchronisieren. Im vorliegenden Beitrag wollen wir uns daher zuerst einmal ansehen, wie wir die Daten aus einer Access-Tabelle in eine Notion-Tabelle übertragen können. Dabei wollen wir die ID, unter welcher die Tabelle in Notion angelegt wurde, auslesen und in der Access-Tabelle speichern, damit wir wissen, welche Datensätze bereits nach Notion übertragen wurden. Gleichzeitig können wir Änderungen an diesen Datensätzen in der Access-Datenbank dann in die Notion-Tabelle übertragen.

Als Beispiel verwenden wir die Tabelle **Artikeldatenbank**, die wir wie in Bild 1 angelegt haben. Wie Sie diese Tabelle anlegen, erfahren Sie im Beitrag **Produktivität mit Notion steigern** (**www.access-im-unternehmen.de/1402**). Hier finden Sie auch wichtige Informationen, auf denen der vorliegende Beitrag aufbaut.

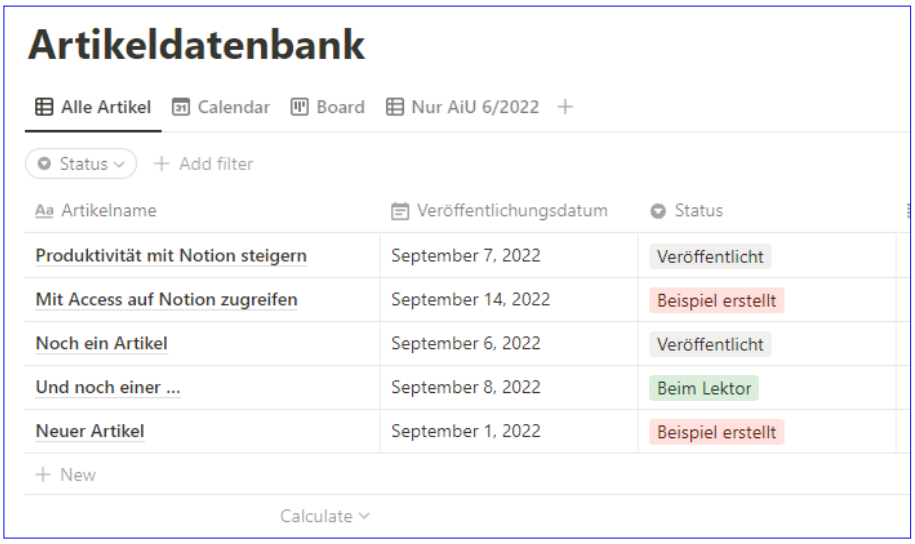


Bild 1: Tabelle in Notion

Datenquelle in der Access-Datenbank

In der Access-Datenbank legen wir zwei Tabellen an. Die erste Tabelle hat den Namen **tblArtikel** und enthält die Felder aus Bild 2. Das Primärschlüsselfeld **ArtikelID** dient der eindeutigen Identifizierung der Datensätze in der Access-Tabelle.

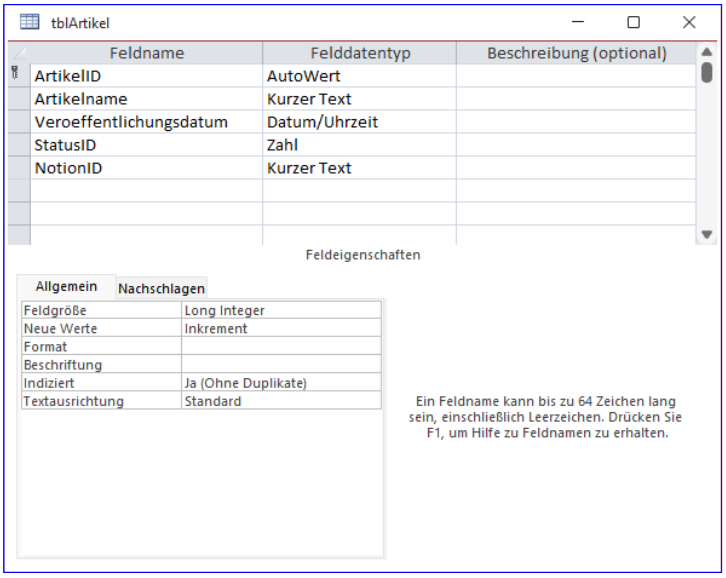


Bild 2: Tabelle mit Artikeldaten in Access

Die Felder **Artikelname** und **Veroeffentlichungsdatum** sollen so in die Notion-Datenbank übernommen werden. Das Feld **StatusID** ist ein Nachschlagefeld zur Auswahl von Datensätzen aus einer weiteren Tabelle namens **tblStatus**.

Diese stellt die Daten aus Bild 3 zur Auswahl im Nachschlagefeld bereit.

Damit wir direkt auf die Daten der Felder **Artikelname**, **Veroeffentlichungsdatum** und **Status** der beiden Tabellen zugreifen können, erstellen wir noch eine Abfrage namens **qryArtikelMitStatus**. Diese enthält die beiden Tabellen **tblArtikel** und **tblStatus** als Datenquelle und zeigt die Felder **ArtikelID**, **Artikelname**, **Veroeffentlichungsdatum**, **Status** und **NotionID** an (siehe Bild 4).

Die Daten dieser Abfrage, die wir in der nachfolgend beschriebenen Prozedur nach Notion exportieren wollen, sehen in der Datenblattansicht der Abfrage wie in Bild 5 aus.

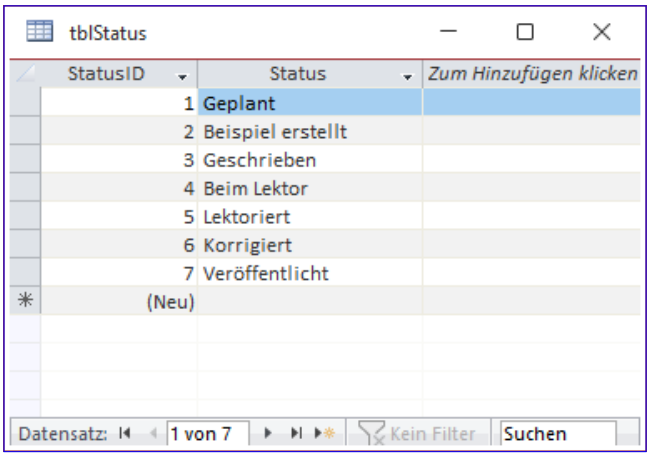


Bild 3: Tabelle mit den verschiedenen Status-Werten

Prozedur zum Übertragen der Daten nach Notion

Damit steigen wir gleich in die Prozedur ein, welche die Daten aus den Tabellen unserer Access-Datenbank in die Notion-Datenbank exportieren soll.

Zur Erinnerung nochmal die Strukturierung in Notion, wo ein Benutzer üblicherweise einen eigenen Workspace verwendet, der eine oder mehrere Datenbanken enthält. Eine Datenbank ist dabei aber eher mit einer

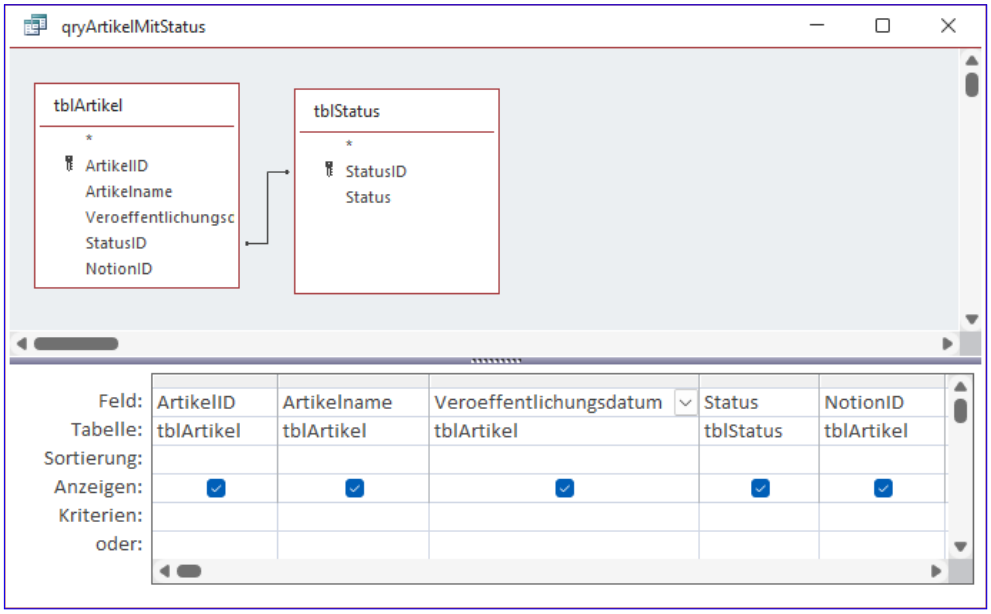


Bild 4: Abfrage mit den Feldern aus den Tabellen **tblArtikel** und **tblStatus**

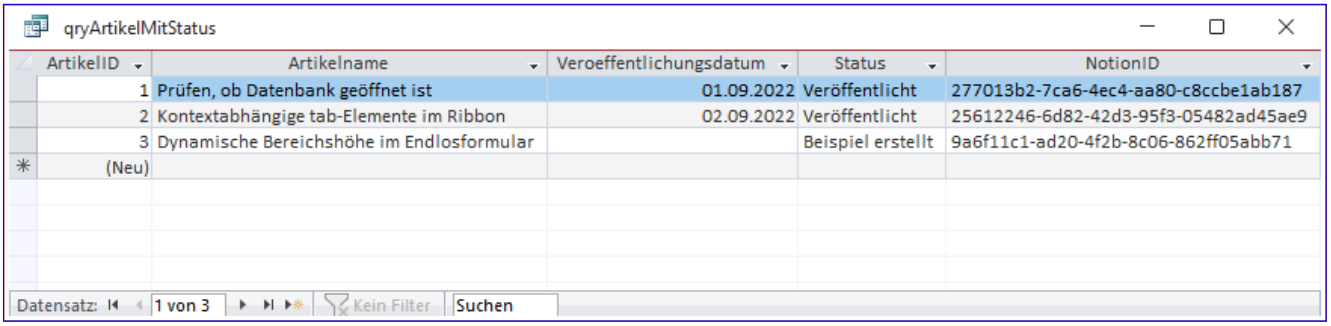


Bild 5: Ergebnis der Abfrage mit den Feldern aus den Tabellen **tblArtikel** und **tblStatus**

Access-Tabelle gleichzusetzen – sie enthält verschiedene Eigenschaften, sogenannte Properties, mit denen die enthaltenen Elemente beschrieben werden. Diese nennen wir Pages. Eine Page in einer Notion-Datenbank entspricht also in etwa einem Datensatz in einer Access-Tabelle.

Bevor wir uns die Prozedur im Detail ansehen, wollen wir noch kurz skizzieren, was darin passieren soll: Wir wollen dort zunächst alle Datensätze der Access-Abfrage **qryArtikelMitStatus** ermitteln, die im Feld **NotionID** den Wert **NULL** enthalten. Der Sinn dahinter ist, dass wir nach dem Hinzufügen der Daten eines Datensatzes der Access-Tabelle in die Notion-Datenbank die ID des in Notion angelegten **Page**-Elements ermitteln und diese für den zugrundeliegenden Datensatz der Tabelle **tblArtikel** in das Feld **NotionID** einfügen wollen. So können wir später immer erkennen, ob dieser Datensatz bereits nach Notion exportiert wurde. Und wenn wir die Daten gegebenenfalls

```
Public Sub ArtikelInNotionAnlegen()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim objJSON As Dictionary
    Dim strResponse As String
    Dim strDatabaseID As String
    Dim strBody As String
    Dim strPageID As String
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM qryArtikelMitStatus WHERE NotionID IS NULL", dbOpenDynaset)
    strDatabaseID = DLookup("DatabaseID", "tblDatabases", "DatabaseTitle = 'Artikeldatenbank'")
    Do While Not rst.EOF
        strBody = "{" & vbCrLf
        strBody = strBody & "  ""object"": ""page""," & vbCrLf
        strBody = strBody & "  ""parent"": {" & vbCrLf
        strBody = strBody & "    ""type"": ""database_id""," & vbCrLf
        strBody = strBody & "    ""database_id"": """ & strDatabaseID & """ & vbCrLf
        strBody = strBody & "  }," & vbCrLf
        strBody = strBody & "  ""properties"": {" & vbCrLf
        strBody = strBody & "    ""Artikelname"": {" & vbCrLf
        strBody = strBody & "      ""title"": [" & vbCrLf
        strBody = strBody & "        {" & vbCrLf
        strBody = strBody & "          ""type"": ""text""," & vbCrLf
        strBody = strBody & "          ""text"": { ""content"": """ & rst!Artikelname & """ }" & vbCrLf
        strBody = strBody & "        }" & vbCrLf
        strBody = strBody & "      ]" & vbCrLf
        strBody = strBody & "    }," & vbCrLf
        strBody = strBody & "    ""Status"": {" & vbCrLf
        strBody = strBody & "      ""id"": ""adPI""," & vbCrLf
        strBody = strBody & "      ""type"": ""select""," & vbCrLf
        strBody = strBody & "      ""select"": {" & vbCrLf
        strBody = strBody & "        ""name"": """ & rst!status & """ & vbCrLf
        strBody = strBody & "        }" & vbCrLf
        ...
```

Listing 1: Artikel nach Notion übertragen, Teil 1

einmal in der Access-Datenbank aktualisieren, können wir diese aktualisierten Datensätze in einer weiteren Prozedur im entsprechenden Datensatz der Notion-Datenbank aktualisieren.

Die Prozedur ArtikelInNotionAnlegen

Diese Prozedur beginnt in Listing 1. Hier deklarieren wir zunächst die benötigten Variablen, zum Beispiel eine namens **db** zum Speichern eines Verweises auf das aktuelle **Database**-Objekt und eine namens **rst** für das Recordset, in dem wir die zu übertragenden Datensätze durchlaufen wollen. Außerdem benötigen wir ein Dictionary, in das wir die Antwort der Notion-API einlesen können, um das Ergebnis auszuwerten. Außerdem benötigen wir diverse **String**-Variablen.

Nach dem Initialisieren der Variablen **db** mit einem Verweis auf das aktuelle **Database**-Objekt referenzieren wir mit der Variablen **rst** alle Datensätze der Abfrage **qryArtikelMitStatus**, bei denen das Feld **NotionID** noch den Wert **Null** enthält. Das zeigt uns, dass die entsprechenden Datensätze noch nicht nach Notion exportiert wurden.

```
...
        If IsNull(rst!Veroeffentlichungsdatum) Then
            strBody = strBody & "      }" & vbCrLf
        Else
            strBody = strBody & "      }," & vbCrLf
            strBody = strBody & "    ""Veröffentlichungsdatum"": {" & vbCrLf
            strBody = strBody & "      ""id"": ""Eqsc""," & vbCrLf
            strBody = strBody & "      ""type"": ""date""," & vbCrLf
            strBody = strBody & "      ""date"": {" & vbCrLf
            strBody = strBody & "        ""start"": """ & Format(rst!Veroeffentlichungsdatum, "yyyy-mm-dd") & """," _
                & vbCrLf
            strBody = strBody & "        ""end"": null," & vbCrLf
            strBody = strBody & "        ""time_zone"": null" & vbCrLf
            strBody = strBody & "      }" & vbCrLf
            strBody = strBody & "    }" & vbCrLf
        End If
        strBody = strBody & "  }" & vbCrLf
        strBody = strBody & "}" & vbCrLf
        Inzwischenablage strBody
        If GetJSON(cStrCreatePage, "POST", strResponse, strBody) = True Then
            Set objJSON = ParseJson(strResponse)
            strPageID = objJSON.Item("id")
            rst.Edit
            rst!NotionID = strPageID
            rst.Update
        Else
            Debug.Print strResponse
        End If
        rst.MoveNext
    Loop
End Sub
```

Listing 2: Artikel nach Notion übertragen, Teil 2

JSON-Daten auslesen

Für XML gibt es eine eigene Bibliothek zum Lesen und Schreiben der Inhalte eines XML-Dokuments. Bei JSON sieht das anders aus: Microsoft hat bisher keine Bibliothek bereitgestellt, mit der man komfortabel auf ein JSON-Dokument zugreifen kann. Wir wollen diesen Missstand zumindest ein wenig lindern. Es gibt bereits ein Modul, welches den Inhalt einer JSON-Datei in eine Art Objektmodell einliest, das von VBA aus wesentlich einfacher zu lesen ist. Dieses greifen wir in diesem Beitrag auf und zeigen, wie wir die Struktur der enthaltenen Daten erfassen und damit besser und systematisch auswerten können.

Fertiges JSON-Modul

Das in der Einleitung angesprochene JSON-Modul finden Sie unter dem folgenden Link zum Download vor:

`https://github.com/VBA-tools/VBA-JSON`

Es ist aber auch in der Beispieldatenbank zu diesem Beitrag enthalten. Dieses Modul stellt eine für uns interessante Funktion namens **ParseJSON** zur Verfügung. Dieser übergibt man den Inhalt des JSON-Dokuments und erhält als Ergebnis ein Objekt des Typs **Dictionary** zurück. Damit wir dieses unter Verwendung von IntelliSense nutzen können, fügen wir dem VBA-Projekt einen Verweis auf die Bibliothek **Microsoft Scripting Runtime** hinzu (siehe Bild 1).

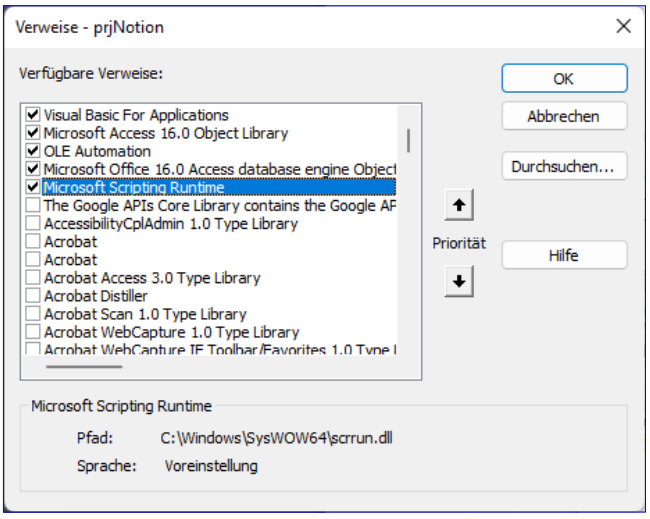


Bild 1: Verweis auf die Bibliothek **Microsoft Scripting Runtime**

Beispieldokument

Als Beispieldokument verwenden wir eines, das wir bei **SelfHTML** unter folgender Adresse gefunden haben:

`https://wiki.selfhtml.org/wiki/JSON`

Dieses Dokument sieht wie in Bild 2 aus. Dieses Beispiel ist übrigens bereits so formatiert, dass es relativ gut lesbar ist und man seine Struktur erkennen kann. Die JSON-Daten, welche die meisten Webservice-APIs zurückliefern, kommen meist ohne Einrückungen und Zeilenumbrüche. Unser Beispiel würde dann etwa wie folgt aussehen:

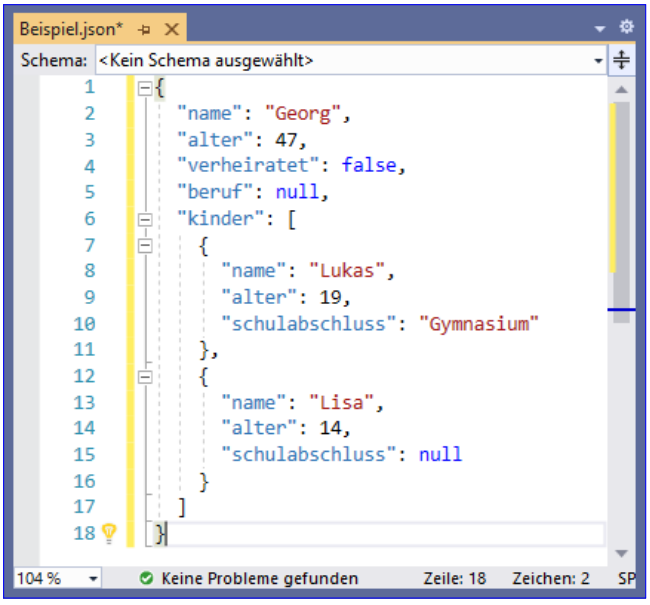


Bild 2: Beispiel für ein JSON-Dokument

```
{"name":"Georg","alter":47,"verheiratet":false,"beruf":null,"kinder":[{"name":"Lukas","alter":19,"schulabschluss":"Gymnasium"},{"name":"Lisa","alter":14,"schulabschluss":null}]}
```

Bereits diese kleine Datenmenge im JSON-Format ist für das menschliche Auge nicht gut lesbar.

JSON in ein Objektmodell überführen

Nun haben wir weiter oben bereits die Funktion **ParseJSON** erwähnt, welche den Inhalt einer JSON-Datei in ein übergeordnetes **Dictionary**-Objekt überführt. Dieses enthält einfache Eigenschaften, also Name-Wert-Paare, weitere **Dictionary**-Objekte sowie **Collection**-Objekte.

Das allein reicht eigentlich aus, um auf die Inhalte zuzugreifen. Das setzt allerdings voraus, dass man die Struktur kennt und wie man den Zugriff auf einzelne Elemente in der Struktur – oder auch auf mehrere Elemente unterhalb einer **Dictionary**- oder **Collection**-Auflistung – realisiert. Auf die einzelnen Werte des JSON-Beispiels können wir nach dem Eintragen in **objJSON** nämlich über die folgenden Ausdrücke zugreifen:

```
objJSON.Item("name")
objJSON.Item("alter")
objJSON.Item("verheiratet")
objJSON.Item("beruf")
objJson.Item("kinder").Item(1).Item("name")
objJson.Item("kinder").Item(1).Item("alter")
objJson.Item("kinder").Item(1).Item("schulabschluss")
objJson.Item("kinder").Item(2).Item("name")
objJson.Item("kinder").Item(2).Item("alter")
objJson.Item("kinder").Item(2).Item("schulabschluss")
```

Wie wir systematisch auf die Elemente eines JSON-Dokuments zugreifen können, schauen wir uns am Ende des Beitrags an.

Zur Erläuterung: Die Elemente in geschweiften Klammern werden von **ParseJSON** in **Dictionary**-Objekte umgewandelt. Diejenigen in eckigen Klammern werden zu **Collection**-Objekten. Und die Name-Wert-Paare wie **"name": "Georg"** werden einfach zu Elementen im **Dictionary**-Objekt, denen wir Name und Wert entnehmen können.

Hier erhalten wir also ein **Dictionary**-Objekt, das einige Unterelemente mit **String**-Werten enthält sowie eines, das als Element ein **Collection**-Element aufnimmt (**Kinder**). Dieses enthält wiederum zwei **Dictionary**-Objekte, eines für jedes Kind.

```
{
  "name": "Georg",
  "alter": 47,
  "verheiratet": false,
  "beruf": null,
  "kinder": [
    {
      "name": "Lukas",
      "alter": 19,
      "schulabschluss": "Gymnasium"
    },
    {
      "name": "Lisa",
      "alter": 14,
      "schulabschluss": null
    }
  ]
}
```

Wenn man die Grundregeln einmal verinnerlicht hat und das JSON-Dokument ordentlich formatiert vorliegen hat, kann man die für den Zugriff notwendigen Ausdrücke gegebenenfalls aus dem Kopf formulieren. Falls nicht, haben wir drei Routinen definiert, welche genau die Ausdrücke ausgeben, die für den Zugriff auf die im JSON-Dokument enthaltenen Elemente nötig sind.

Wir gehen davon aus, dass wir das JSON-Dokument auf irgendeine Weise als Zeichenkette erhalten. In der

folgenden Beispielprozedur lesen wir dazu den Inhalt der Datei **Beispiel.json** aus dem gleichen Verzeichnis wie die Datenbank in die Variable **strJSON** ein.

Danach rufen wir die Prozedur **GetJSONDOM** auf, um die für den Zugriff auf das Document Object Model des JSON-Dokuments nötigen Ausdrücke zu ermitteln:

```
Public Sub JSONBeispiel()
    Dim strJSON As String
    Open CurrentProject.Path & "\Beispiel.json" _
                                        For Input As #1
    strJSON = Input$(LOF(1), #1)
    Close #1
    GetJSONDOM strJSON, True
End Sub
```

JSON-Dictionary auswerten

Die Funktion **GetJSONDOM** erwartet ein JSON-Dokument in Form einer **String**-Variablen als Parameter (siehe Listing 1). Außerdem können wir zwei weitere Parameter übergeben:

- **bolValues**: Gibt an, ob hinter den jeweiligen Zeilen für die Pfade zu den Elementen noch die Werte angezeigt werden sollen.

```
Public Function GetJSONDOM(strJSON As String, Optional bolValues As Boolean, Optional bolDebug As Boolean) As String
    Dim objJSON As Dictionary
    Dim dic As Dictionary
    Dim col As Collection
    Dim strKey As String
    Dim i As Integer
    Dim strJSONDOM As String
    Set objJSON = mdlJSON.ParseJson(strJSON)
    For i = 0 To objJSON.Count - 1
        Select Case TypeName(objJSON.Items(i))
            Case "Dictionary"
                Set dic = objJSON.Items(i)
                JSONDictionary dic, "objJSON", strJSONDOM, bolValues, bolDebug
            Case "Collection"
                Set col = objJSON.Items(i)
                strKey = ".Item(""" & objJSON.Keys(i) & """)"
                JSONCollection col, "objJson" & strKey, strJSONDOM, bolValues, bolDebug
            Case Else
                If Not bolValues Then
                     If bolDebug Then Debug.Print "objJSON.Item(""" & objJSON.Keys(i) & """)"
                    strJSONDOM = strJSONDOM & "objJSON.Item(""" & objJSON.Keys(i) & """)" & vbCrLf
                Else
                    If bolDebug Then Debug.Print "objJSON.Item(""" & objJSON.Keys(i) & """):" & objJSON.Items(i)
                    strJSONDOM = strJSONDOM & "objJSON.Item(""" & objJSON.Keys(i) & """):" & objJSON.Items(i) & vbCrLf
                End If
        End Select
    Next i
    GetJSONDOM = strJSONDOM
End Function
```

Listing 1: Ausgabe des DOM eines JSON-Dokuments

VBA: Punkt oder Ausrufezeichen

Unter VBA gibt es Gelegenheiten, wo man zwischen zwei Elemente entweder einen Punkt oder ein Aufrufezeichen setzen kann, zum Beispiel bei gebundenen Steuerelementen in der Form Me.Vorname oder rst!Vorname. Im Gegensatz dazu stehen beispielsweise die Elemente eines Recordsets – hier können wir nur rst!Vorname verwenden, rst.Vorname führt zu einem Fehler. Ganz kompliziert wird es, wenn wir gebundene Steuerelemente auch noch umbenennen. Dann können wir auf den Wert des Steuerelements sowohl über Me!Vorname, Me.Vorname, Me!txtVorname oder Me.txtVorname zugreifen. Und manchmal führt die Syntax mit dem Punkt zu ernsthaften Verwirrungen, weshalb man in manchen Fällen auf jeden Fall die Ausrufezeichen-Syntax nutzen sollte.

Wir fangen mal vorn an: Von Punkten und Ausrufezeichen bleibt man solange verschont, wie man nicht per VBA auf den Inhalt von Steuerelementen oder Feldern zugreifen möchte. Das Problem daran ist: Ohne das ist man dabei recht eingeschränkt. Das Gute ist: Wenn man einmal an die Stelle gelangt ist, wo man Objektbezüge über Punkte und/oder Ausrufezeichen herstellt, kann man nicht viel falsch machen – und wo es dabei Stolpersteine gibt, erläutern wir in diesem Beitrag.

Punkte und Ausrufezeichen in Recordsets und ähnlichen Elementen

Wir beginnen dort, wo der Einsatz eindeutig geregelt ist. Wenn man beispielsweise eine Schleife programmieren möchte, mit der man die Datensätze eines Recordsets durchläuft, möchte man auf die Felder dieses Recordsets zugreifen. Das Recordset bietet einige Eigenschaften und Methoden an, die man wie bei allen Objekten mit der Punkt-Syntax adressieren kann, also indem man den Namen der Objektvariablen, meist **rst**, eingibt und dann einen Punkt setzt, um per IntelliSense die möglichen Elemente zu sehen. Das Ergebnis ist eine Liste wie in Bild 1.

Darüber findet man beim Recordset alle möglichen Einträge, aber nicht die Felder des Recordsets. Diese können wir auch nicht über die Punkt-Syntax referenzieren. **rst.KategorieID** funktioniert also nicht, sondern wirft den Fehler **Methode oder Datenobjekt nicht gefunden**.

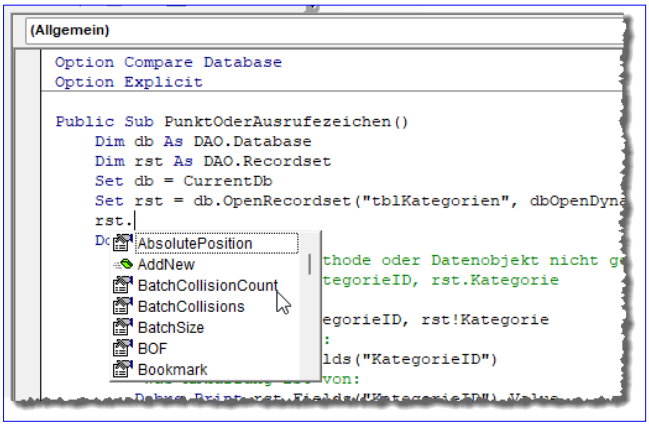


Bild 1: Nutzung des Punktes für die Anzeige von Elementen per IntelliSense

Stattdessen nutzen wir an dieser Stelle in der kürzesten Form das Ausrufezeichen, also beispielsweise **rst!KategorieID**. Wir haben hier keine IntelliSense-Unterstützung, aber immerhin liefert dies den Wert des Feldes für den jeweiligen Datensatz. Im Detail sieht das wie im folgenden Beispiel aus:

```
Public Sub PunktOderAusrufezeichen()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set db = CurrentDb
    Set rst = db.OpenRecordset("tblKategorien", dbOpenDynaset)
    Do While Not rst.EOF
        Debug.Print rst!KategorieID, rst!Kategorie
        rst.MoveNext
```

```
    Loop
End Sub
```

Wie wir hier sehen, werden die Eigenschaften und Methoden, also **EOF** oder **MoveNext**, mit der Punkt-Syntax verwendet. Wir haben in diesem Fall also eine eindeutige Aufteilung: Eigenschaften und Methoden erreichen wir mit einem Punkt, die Felder mit dem Ausrufezeichen.

Warum aber können wir die Felder überhaupt mit dem Ausrufezeichen adressieren? Weil es sich bei den Feldern um Elemente einer Auflistung handelt, genau genommen sogar um die einer Auflistung mit benutzerdefinierten Elementen, und weil das Ausrufezeichen in diesem Fall als Abkürzung für eine ausführlichere Schreibweise verwendet werden kann. **rst!KategorieID** steht nämlich eigentlich für ein Element der **Fields**-Auflistung des **Recordset**-Objekts:

```
Debug.Print rst.Fields("KategorieID")
```

Und auch das ist noch nicht die ganze Wahrheit, denn **rst.Fields("Kategorie")** können wir auch nur verwenden, weil man die Standardeigenschaft eines Elements immer weglassen kann. Eigentlich heißt der Ausdruck nämlich wie folgt, und dabei ist **Value** die Standardeigenschaft des **Field**-Objekts, das wir mit **Fields("KategorieID")** referenzieren:

```
Debug.Print rst.Fields("KategorieID").Value
```

Das Beispiel finden Sie im Modul **mdlPunktUndAusrufezeichenInRecordset**.

Eigenschaften und Felder in Formular- und Berichtsmodulen

Eine andere Situation finden wir vor, wenn wir die Klassenmodule von Formularen oder Berichten betrachten, die an eine Datensatzquelle wie eine Tabelle oder Abfrage gebunden sind – und hier speziell bei den Steuerelementen und den Feldern der Datensatzquelle.

Binden wir ein Formular an eine Tabelle wie **tblKategorien**, welche die beiden Felder **KategorieID** und **Kategorie** enthält, und fügen wir diese beiden Felder zum Detailbereich des Formularentwurfs hinzu, können wir beispielsweise in der VBA-Ereignisprozedur, die durch eine Schaltfläche ausgelöst wird, wie folgt auf die Werte der gebundenen Felder zugreifen:

```
Debug.Print Me.KategorieID, Me.KategorieID
```

Wir nutzen also einfach mit **Me** den Verweis auf das Klassenmodul selbst und können dann nach Eingabe des Punktes nicht nur auf die Eigenschaften und Methoden des Formulars zugreifen, sondern auch auf die Feldnamen der Datensatzquelle des Formulars.

Noch besser: Obwohl es sich hier nicht um eingebaute Eigenschaften der Formularklasse handelt, können wir IntelliSense nutzen, um darauf zuzugreifen – siehe Bild 2! Warum ist das überhaupt möglich? Der Hintergrund ist, dass Access beim Hinzufügen einer Datensatzquelle zu einem Formular automatisch für jedes Feld eine eigene Eigenschaft zum Klassenmodul des Formulars hinzufügt. Deshalb werden diese auch per IntelliSense angezeigt. Wir können hier aber auch, genau wie beim **Recordset**-Objekt, über die Ausrufezeichen-Syntax auf die Felder zugreifen. Das sieht so aus:

```
Debug.Print Me!KategorieID, Me!Kategorie
```

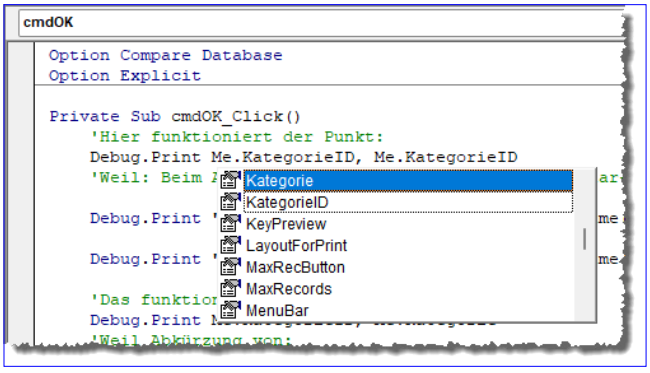


Bild 2: Zugriff per Punkt und IntelliSense auf die Felder der Datensatzherkunft

Formatassistent für Textfelder

Access liefert standardmäßig bereits einige Assistenten mit. Für die Format-Eigenschaft jedoch liefert es nur ein Auswahlfeld mit einigen voreingestellten Optionen wie verschiedneen Zeit-, Zahlen und Boolean-Formaten. Diese dienen durchaus als gute Beispiele, wie man selbst Formatierungen vornehmen kann, aber wenn man sich dann an diese Aufgabe begibt, wird schnell ein ziemliches Experimentieren daraus. Um diesen Teil zu vereinfachen, wollen wir Sie mit einem Assistenten unterstützen. Dieser soll direkt über das Eingabefenster für die Format-Eigenschaft geöffnet werden und die Möglichkeit bieten, verschiedene Werte einzugeben und diese mit dem geünwschten Formatierungsausdruck zu testen. Und natürlich finden Sie in diesem Beitrag die Anleitung, wie Sie den Assistenten selbst erstellen können!

Die Format-Eigenschaft

Die Eigenschaft **Format** kennen Sie von verschiedenen Situationen. Das erste Mal begegnet sie einem beim Erstellen einer neuen Datenbankanwendung bereits im Tabellenentwurf für neue Felder beispielsweise des Datentyps **Kurzer Text** (siehe Bild 1). An dieser Stelle finden wir noch nicht einmal eine Auswahlliste mit verschiedenen Optionen vor, sodass wir zu eigenen Versuchen mit den verschiedenen Platzhaltern und Möglichkeiten der Format-Eigenschaft angehalten sind.

Erstellen wir Abfragen auf Basis einer solchen Tabelle, finden wir hier ebenfalls die Möglichkeit, die **Format**-Eigenschaft zu setzen. Damit würden wir eventuelle Einstellungen aus der zugrunde liegenden Tabelle überschreiben. Auch hier finden wir keine Unterstützung zum Zusammenstellen der Ausdrücke für die **Format**-Eigenschaft.

Damit landen wir bei den Formularen, wo wir Steuerelemente auf Basis der Felder der als Datensatzquelle verwendeten Tabelle oder Abfrage hinzufügen. Auch hier finden wir die Eigenschaft Format vor. Allerdings sehen wir an dieser Stelle bereits ein reichlich gefülltes Auswahlfeld

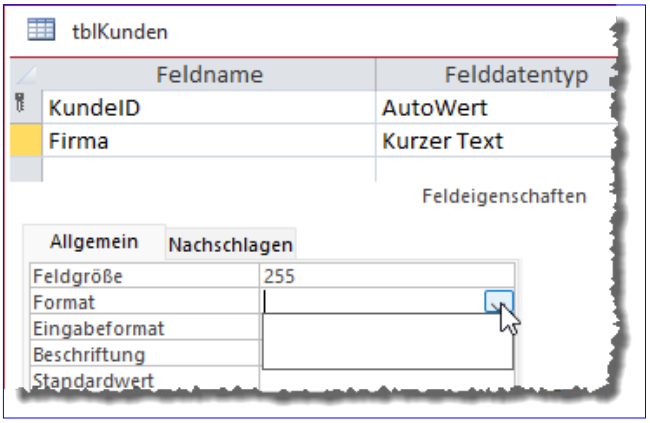


Bild 1: Format-Eigenschaft im Tabellenentwurf

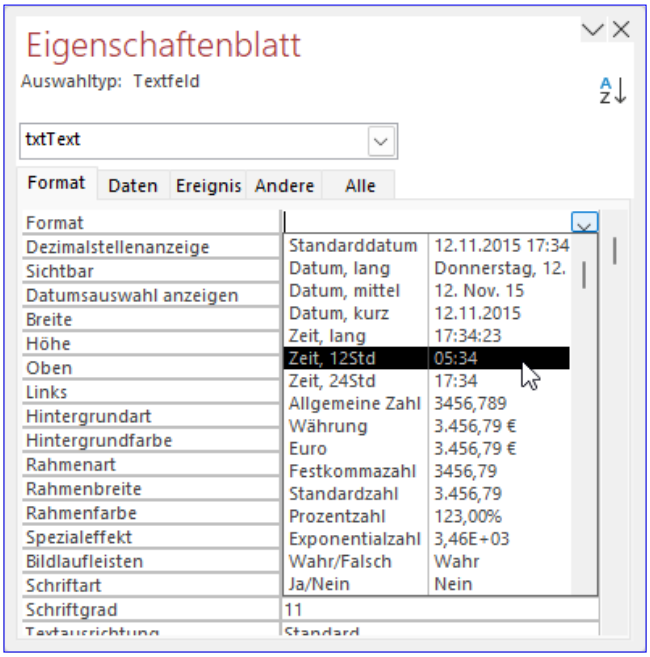


Bild 2: Auswahl verschiedener Formate

für verschiedene Formatierungen (siehe Bild 2).

Wenn wir nun an der einen oder anderen Stelle einen **Format**-Ausdruck festlegen wollen, der die im Feld gespeicherten Daten in der gewünschten Art formatiert, müssen wir in die Entwurfsansicht wechseln, den **Format**-Ausdruck definieren, zurück zur Datenblattansicht wechseln, prüfen, ob der **Format**-Ausdruck die gewünschte Ansicht liefert und falls nicht, wieder zurück zur Entwurfsansicht wechseln, um den Wert der **Format**-Eigenschaft weiter zu optimieren.

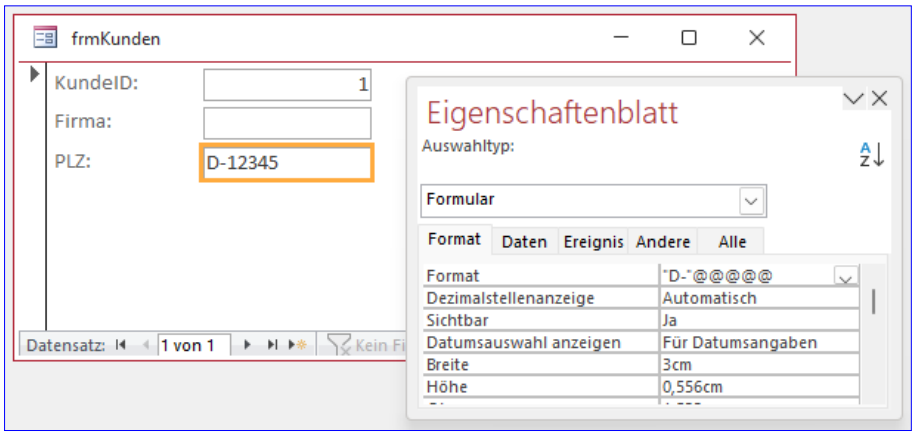


Bild 3: Anpassen von Formaten in der Layoutansicht von Formularen

Es gibt in Formularen eine Möglichkeit, den Formatausdruck für ein Feld ohne ständiges Wechseln zwischen Formular- und Entwurfsansicht zu testen. Dabei handelt es sich um die Layout-Ansicht. Hier können wir das betroffene Feld markieren und die Eigenschaft **Format** anpassen, während die Formatierung direkt auf die aktuell angezeigten Daten angewendet wird (siehe Bild 3).

In Tabellen und Abfragen steht diese Möglichkeit allerdings nicht zur Verfügung, sodass wir mit dem hier vorgestellten Assistenten eine alternative Lösung anbieten wollen. Der Assistent soll außerdem noch einen weiteren Mehrwert bieten: Er soll mehrere Werte gleichzeitig inklusive der formatierten Version anzeigen können.

Grundgerüst des Assistenten

Die grundlegende Vorgehensweise beim Erstellen eines Eigenschafts-Assistenten, wie wir ihn hier beschreiben, finden Sie im Beitrag **Eigenschaftsassistenten oder Property Wizards** (**www.access-im-unternehmen.de/****). Im vorliegenden Beitrag beschreiben wir nur kurz die Anwendung der dort vorgestellten Techniken anhand unserer konkreten Umsetzung.

Als Erstes benötigen wir eine Tabelle namens **USysRegInfo**, welche die Daten für die Registrierung des Assistenten in der Windows-Registry enthält. Diese werden beim Installieren mit dem Add-In-Manager in die Registry eingetragen, damit Access den Assistenten an der gewünschten Stelle anbieten kann.

Die Tabelle enthält die Daten aus Bild 4. Wichtig ist, dass der Name der **.accda**-Datei genau in der Zeile mit dem

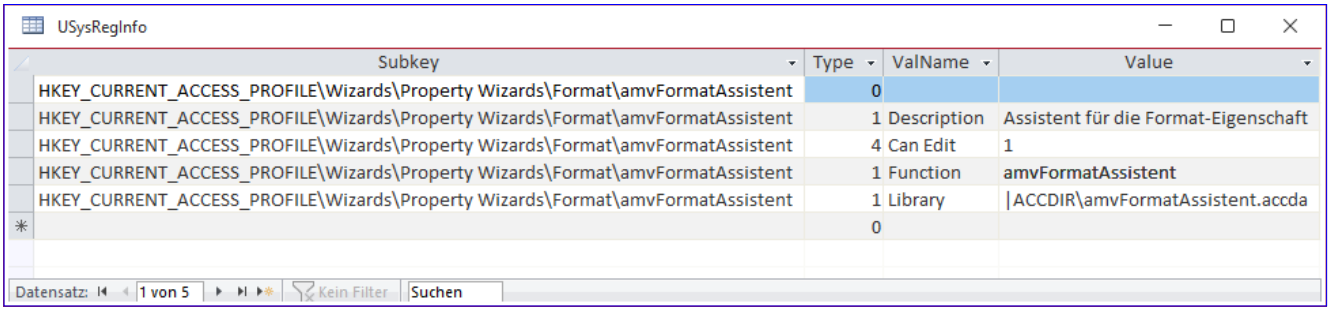


Bild 4: Die Tabelle **USysRegInfo** mit den Daten für die Registrierung des Assistenten

Wert **Library** angegeben wird und das der Name der aufzurufenden Funktion des Assistenten in der Zeile mit dem Wert **Function** erscheint.

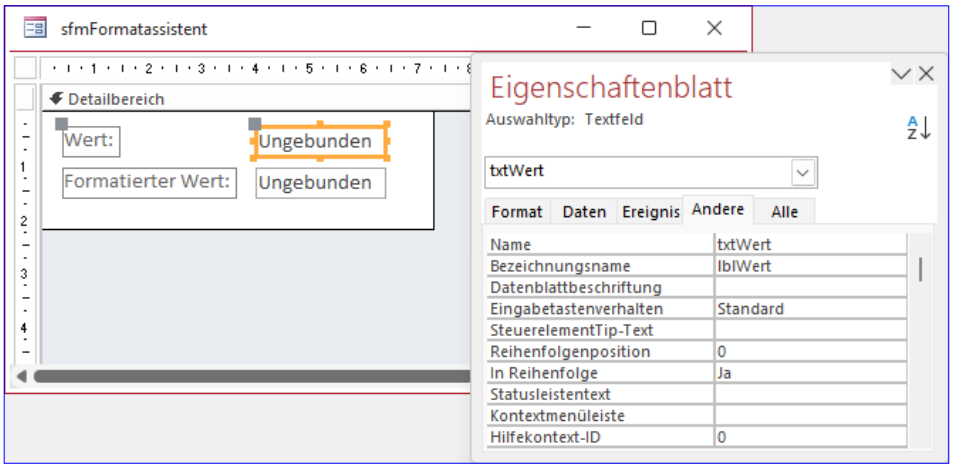


Bild 5: Unterformular des Formulars des Assistenten

Formulare für den Assistenten vorbereiten

Der Assistent soll seine Daten in Haupt- und Unterformular anzeigen. Das Hauptformular soll das Textfeld zur Eingabe des zu testenden Format-Ausdrucks bereitstellen sowie die beiden Schaltflächen zum Übernehmen der Eingabe oder zum Abbrechen. Außerdem soll es das Unterformular anzeigen, das in der Datenblattansicht zwei Spalten anzeigt.

Die erste enthält die Daten, die formatiert werden sollen, in der gespeicherten Version. Die zweite soll die Daten mit dem Formatausdruck formatieren, den der Benutzer im Hauptformular angibt. Die Datensätze des Unterformulars in der Datenblattansicht sollen die Daten aus der eigentlichen Datenquelle anzeigen. Wenn wir also die Eigenschaft **Format** für das Feld **PLZ** einer Tabelle anpassen wollen, dann soll das Unterformular die Werte des Feldes **PLZ** für alle Datensätze der zugrunde liegenden Tabelle anzeigen.

Unterformular anlegen

Daher erstellen wir nun zwei Formulare. Wir beginnen mit dem Unterformular, das wir unter dem Namen **sfmFormatassistent** speichern. Es sieht in der Entwurfsansicht wie in Bild 5 aus. Wir fügen diesem zwei Textfelder samt Bezeichnungsfeldern hinzu. Die Bezeichnungsfelder erhalten die Beschriftungen **Wert** und **Formatierter Wert**. Die Textfelder versehen wir mit den Namen **txtWert** und **txtFormatierterWert**. Damit die Daten des Unterformulars in der Datenblattansicht angezeigt werden, stellen wir die Eigenschaft **Standardansicht** auf den Wert **Datenblatt** ein. Danach speichern und schließen wir das Unterformular.

Hauptformular anlegen

Das Hauptformular legen wir unter dem Namen **frmFormatassistent** an. Solange wir mit dem Formular des Assistenten keine Daten bearbeiten wollen, benötigen wir einige Elemente nicht. Diese blenden wir aus, indem wir die Eigenschaften **Datensatzmarkierer**, **Navigationsschaltflächen**, **Trennlinien** und **Bildlaufleisten** auf **Nein** einstellen. Außerdem soll das Formular immer zentriert im Access-Fenster angezeigt werden, sodass wir die Eigenschaft **Automatisch zentrieren** auf den Wert **Ja** einstellen.

Außerdem stellen wir den Wert der Eigenschaft **Beschriftung** auf einen passenden Titel ein, in diesem Fall **amv-Formatassistent**.

Dann folgen die eigentlichen Steuerelement. Ganz oben platzieren wir ein Textfeld mit dem Namen **txtFormat**. Daneben legen wir ein weiteres Textfeld an, dieses heißt **txtFormatDeutsch**. Der Grund ist, dass die deutschen Formatierungen teilweise anders lauten als die englischen, die wir hier nutzen müssen.

Darunter fügen wir das Unterformular ein, indem wir es aus dem Navigationsbereich in den Formularentwurf ziehen. Unter diesem platzieren wir die beiden Schaltflächen **cmdOK** und **cmdAbbrechen**, sodass das Formular wie in Bild 6 aussieht. Außerdem bringen wir noch eine weitere

Schaltfläche namens **cmdBeispielwerteEinlesen** hinzu.

Für die Schaltflächen hinterlegen wir die folgenden Ereignisprozeduren:

```
Private Sub cmdAbbrechen_Click()
    DoCmd.Close acForm, Me.Name
End Sub

Private Sub cmdOK_Click()
    Me.Visible = False
End Sub
```

Bevor wir mit der Programmierung der Funktion des Formulars fortschreiten, kümmern wir uns als Erstes noch um Beispieldaten für den Assistenten und dann um seinen Aufruf und die Auswertung des Ergebnisses.

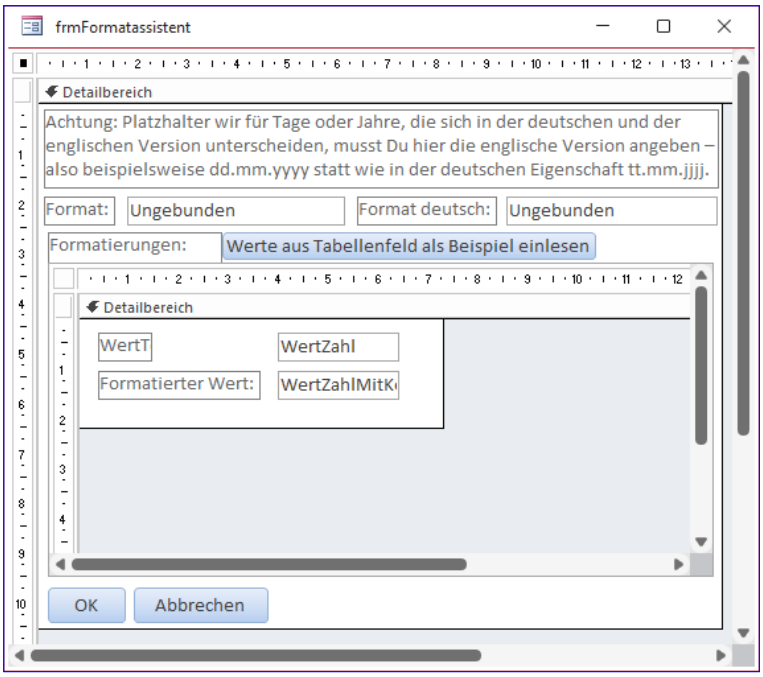


Bild 6: Hauptformular des Assistenten mit Unterformular

Vorbereitung von Beispieldaten für das Unterformular

Der Benutzer soll direkt mit dem Format experimentieren können und deshalb stellen wir ihm im Unterformular zehn Datensätze zur Verfügung. Diese stammen aus der Tabelle **tblWerteFormatierteWerte**, die in der Entwurfsansicht wie in Bild 7 aussieht. Wir haben Felder für die verschiedenen Datentypen festgelegt. Damit diese nicht alle angezeigt werden müssen, der Benutzer aber dennoch mit realistischen Beispielwerten arbeiten kann, weisen wir der Eigenschaft **Steuerelementinhalt** beziehungsweise unter VBA **ControlSource** des Feldes **txtWert** das jeweilige Feld zu.

Wechseln wir in die Datenblattansicht, sehen wir, dass wir bereits einige Beispieldatensätze eingegeben haben – zugegebenermaßen etwas improvisiert, aber der Benutzer kann diese nach seinen eigenen Wünschen anpassen (siehe Bild 8).

Startfunktion für den Assistenten

Um das Formular beim Aufrufen des Assistenten anzuzeigen, müssen wir die Funktion, die wir in der Tabelle **USysRegInfo** hinterlegt haben, zunächst mit den entsprechenden Anweisungen füllen. Diese Funktion soll nicht nur das

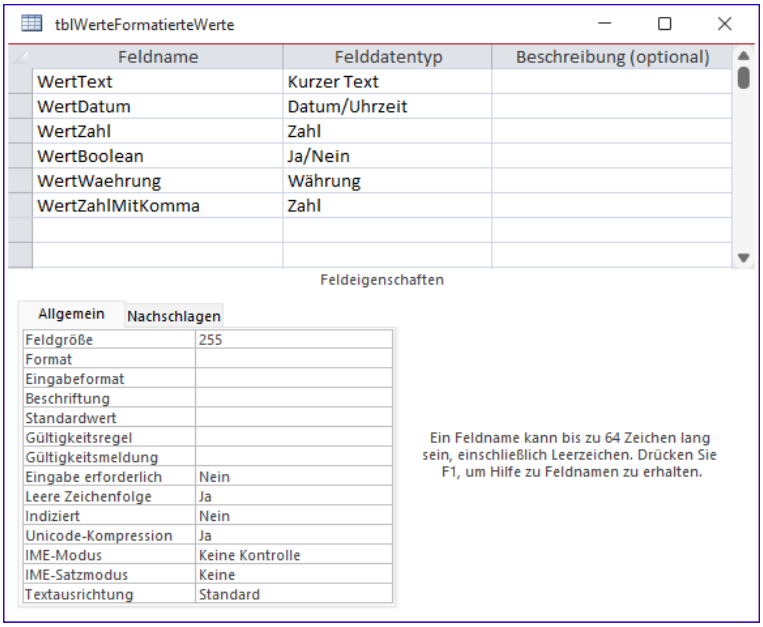


Bild 7: Entwurf der Tabelle für die Beispieldaten

Formular öffnen, sondern auch noch die benötigten Informationen übergeben. Diese erhalten wir in dieser Funktion über die standardmäßig zu definierenden Parameter:

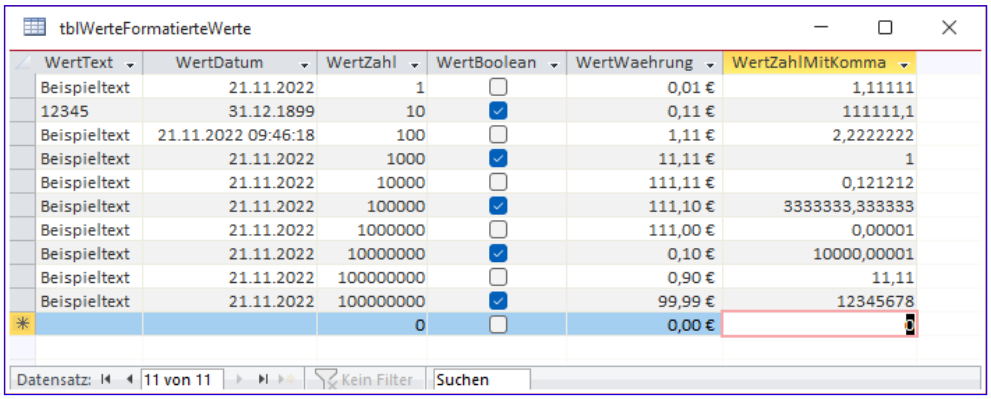


Bild 8: Beispielwerte für die verschiedenen Felddatentypen

- **strObject** liefert den Namen des Objekts, also Tabelle, Abfrage, Formular oder Bericht, von dem aus der Assistent aufgerufen wurde.

- **strControl** liefert den Namen des Tabellen- oder Abfragefeldes beziehungsweise des Formular- oder Berichtssteuerelements, für welches der Assistent gestartet wurde.

- **strCurVal** liefert, sofern vorhanden, den aktuellen Wert der **Format**-Eigenschaft für das auslösende Element.

Die Funktion aus Listing 1 stellt zunächst den Namen des zu öffnenden Formulars, hier **frmFormatassistent**, in der Variablen **strFormular** ein. Dann fügt es die Werte der drei Parameter in einer einzigen Zeichenkette zusammen – getrennt durch das Pipe-Zeichen. Das sieht beispielsweise wie folgt aus:

```
tblKunden|PLZ|"D-"@@@@@
```

Danach ruft die Funktion das Formular mit der **DoCmd.Open**-Methode auf und übergibt den Inhalt von **strOpenArgs** mit dem Parameter **OpenArgs**. Durch den Parameter **WindowMode:=acDialog** wird das Formular außerdem als modaler Dialog geöffnet. Dies sorgt dafür, dass der Code an dieser Stelle angehalten wird.

Hier folgt nun die eigentliche Bearbeitung. Diese schließt der Benutzer entweder mit der **OK**- oder der **Abbrechen**-Schaltfläche ab. Dadurch wird das Formular entweder ausblendet oder geschlossen, was beides dazu führt, dass der aufrufende Code fortgesetzt wird.

Hier kommt die Funktion **IstFormularGeoeffnet** zum Einsatz und überprüft, ob das Formular noch geöffnet ist.

```
Public Function amvFormatAssistent(strObject As String, strControl As String, strCurVal As String) As String
    Dim strOpenArgs As String
    Dim strFormular As String
    strFormular = "frmFormatassistent"
    strOpenArgs = strObject & "|" & strControl & "|" & strCurVal
    DoCmd.OpenForm strFormular, WindowMode:=acDialog, OpenArgs:=strOpenArgs
    If IstFormularGeoeffnet(strFormular) Then
        amvFormatAssistent = Nz(Forms(strFormular)!txtFormatDeutsch, "")
        DoCmd.Close acForm, strFormular
    End If
End Function
```

Listing 1: Startfunktion des Assistenten

```
Private Sub Form_Open(Cancel As Integer)
    Dim strOpenArgs As String
    Dim strCurVal As String
    Dim intDataType As DataTypeEnum
    Dim strSQL As String
    strOpenArgs = Nz(Me.OpenArgs, "")
    If Len(strOpenArgs) = 0 Then
        MsgBox "Der Assistent kann nicht ohne Angabe der Parameter aufgerufen werden."
        Cancel = True
        Exit Sub
    End If
    If Not Len(strOpenArgs) = 0 Then
        strObject = Split(strOpenArgs, "|")(0)
        strControl = Split(strOpenArgs, "|")(1)
        strCurVal = Split(strOpenArgs, "|")(2)
    Else
        MsgBox "Der Assistent kann nicht ohne Angabe der Parameter aufgerufen werden."
        Cancel = True
        Exit Sub
    End If
    Me!txtFormatDeutsch = strCurVal
    intObjectType = GetObjectType(strObject)
    intDataType = GetDataType(strObject, strControl, intObjectType)
    strCurVal = UebersetzeNachEnglisch(strCurVal, intDataType)
    Me!txtFormat = strCurVal
    Select Case intDataType
        Case dbText
            Me!sfmFormatassistent.Form!txtWert.ControlSource = "WertText"
            Me!sfmFormatassistent.Form!txtFormatierterWert.ControlSource = "WertText"
        Case dbCurrency
            Me!sfmFormatassistent.Form!txtWert.ControlSource = "WertWaehrung"
            Me!sfmFormatassistent.Form!txtFormatierterWert.ControlSource = "WertWaehrung"
        Case dbBoolean
            Me!sfmFormatassistent.Form!txtWert.ControlSource = "WertBoolean"
            Me!sfmFormatassistent.Form!txtFormatierterWert.ControlSource = "WertBoolean"
        Case dbDate
            Me!sfmFormatassistent.Form!txtWert.ControlSource = "WertDatum"
            Me!sfmFormatassistent.Form!txtFormatierterWert.ControlSource = "WertDatum"
        Case dbInteger, dbLong
            Me!sfmFormatassistent.Form!txtWert.ControlSource = "WertZahl"
            Me!sfmFormatassistent.Form!txtFormatierterWert.ControlSource = "WertZahl"
        Case dbSingle, dbDouble
            Me!sfmFormatassistent.Form!txtWert.ControlSource = "WertZahl"
            Me!sfmFormatassistent.Form!txtFormatierterWert.ControlSource = "WertZahlMitKomma"
        Case Else
            MsgBox "Nicht definierter Felddatentyp: " & intDataType
    End Select
    Me!sfmFormatassistent.Form!txtFormatierterWert.Format = Me!txtFormat
End Sub
```

Listing 2: Beim Aufrufen des Formulars des Assistenten

Ist das nicht der Fall, hat der Benutzer die **Abbrechen**-Schaltfläche betätigt und es ist nichts weiter zu tun. Ist das Formular jedoch noch geöffnet, muss der Benutzer es durch einen Klick auf die Schaltfläche **cmdOK** unsichtbar gemacht haben und die Anweisungen innerhalb der **If... Then**-Bedingung werden ausgeführt.

Hier liest die Funktion den Wert des Textfeldes **txtFormat-Deutsch** aus dem noch geöffneten Formular als Rückgabewert der Funktion **amvFormatAssistent** ein und schließt das Formular dann. Hier ist zu berücksichtigen, dass das Textfeld **txtFormat** auch leer sein kann, also den Wert **Null** enthält. In diesem Fall ist der Wert **Null** durch eine leere Zeichenkette zu ersetzen, was wir mit der **Nz**-Funktion erledigen.

Beim Öffnen des Formulars

Beim Öffnen des Formulars wird die Prozedur aus Listing 2 aufgerufen. Diese deklariert einige Variablen und liest als erstes den Inhalt der Eigenschaft **OpenArgs** in die Variable **strOpenArgs** ein. Dabei wandelt den eventuell vorhandenen Wert **Null** in eine leere Zeichenkette um, da die Zuweisung des Wertes **Null** zu einer **String**-Variablen einen Fehler auslösen würde.

Dann prüft die Prozedur, ob **strOpenArgs** überhaupt eine Zeichenkette mit einer Länge größer **0** enthält. Falls ja, untersucht sie die Zeichenkette genauer. In diesem Fall zerlegen wir diese mit der **Split**-Funktion in die durch das Pipe-Zeichen getrennten Einzelteile. Über den nachgeschalteten Index greifen wir auf das erste, zweite und dritte Element einer Zeichenkette wie beispielsweise der folgenden zu:

```
tblKunden|PLZ|"D-"@@@@@
```

Das erste Element landet in der Variablen **strObject**, das zweite in **strControl** und das dritte in **strCurVal**. Ist **strOpenArgs** hingegen leer, erscheint eine entsprechende Meldung und das Formular die Prozedur wird mit **Exit Sub** verlassen. Außerdem stellen wir noch den Parameter **Cancel** auf den Wert **True** ein, damit auch das Öffnen des Formulars abgebrochen wird.

Sind alle Werte in die Variablen eingelesen, weisen wir den Wert aus **strCurVal** dem Textfeld **txtFormatDeutsch** zu. Dies ist also die Formatierung, die der Benutzer gegebenenfalls bereits in der entsprechenden Eigenschaft des Feldes oder Steuerelements festgelegt hat. Die englische Version wollen wir im Textfeld **txtFormat** anzeigen. Dazu müssen wir das Format jedoch erst noch übersetzen und dazu benötigen wir noch weitere Informationen.

strObject liefert uns den Namen des Objekts, von dem aus der Benutzer den Assistenten aufgerufen hat. Damit wollen wir den Typ des Objekts ermitteln. Davon ausgehend, dass jeder Objektname nur einmal pro Datenbank angelegt werden kann, können wir damit die Tabelle **MSysObjects** durchsuchen und aus dieser den Typ ermitteln. Das erledigt die Hilfsfunktion **GetObjectType** für uns, die wir weiter unten beschreiben.

Diese wandelt außerdem die kryptischen Zahlenwerte wie zum Beispiel **-32.768** für Formulare in handliche Konstanten wie **acForm** um und wir speichern den ermittelten Wert in der Variablen **intObjectType**.

Den Objekttyp benötigen wir unter anderem für die nachfolgende Ermittlung des Felddatentyps des mit der **Format**-Eigenschaft auszustattenden Feldes. Diese erfolgt mit der Funktion **GetDataType**. Dieser übergeben wir den Namen des Objekts und des Felds beziehungsweise Steuerelements und schreiben das Ergebnis in Form eines der entsprechenden Konstanten wie beispielsweise **acText** für ein Textfeld mit dem Datentyp **Kurzer Text** in die Variable **intDataType**. Diese hat wiederum den Datentyp **DataTypeEnum**.

Den Datentyp nutzen wir nun in der Funktion **UebersetzeNachEnglisch**, der wir den zu übersetzenden Ausdruck übergeben sowie den Datentyp. Das Ergebnis dieses Aufrufs landet schließlich im Textfeld **txtFormat**. Wie und

warum wir das Format übersetzen, erläutern wir weiter unten.

Den Inhalt der Variablen **intDataType** untersuchen wir dann in einer **Select Case**-Anweisung daraufhin, um welchen Datentyp es sich handelt. Abhängig davon nehmen wir nun Einstellungen im Unterformular des Assistenten vor. Dieses zeigt ja bereits die Tabelle der Tabelle **tblWerteFormatierteWerte** an.

Je nachdem, welchen Datentyp das Feld hat, für das der Benutzer den Format-Assistenten aufgerufen hat, stellen wir die Eigenschaft **Steuerelementinhalt** auf eines der Felder **WertText**, **WertWaehrung**, **WertBoolean**, **WertDatum**, **WertZahl** oder **WertZahlMitKomma** ein.

Hat der Benutzer einen anderen Datentyp gewählt, erscheint eine entsprechende Meldung – an dieser Stelle könnte man die Prozedur noch für diesen zusätzlichen Datentyp erweitern. Schließlich stellen wir die Eigenschaft **Format** des Feldes **txtFormatierterWert** auf die im Textfeld **txtFormat** gespeicherte **Formatierung** ein.

Ändern des Formats

Wenn der Benutzer die Formatierung im Textfeld **txtFormat** ändert, soll sich dies jeweils direkt auf die zu formatierenden Beispieltextfelder auswirken. Dazu hinterlegen wir für das Ereignis **Bei Änderung** das Format für die Eigenschaft **Format** des Textfeldes **txtFormatierterWert**. Hier lesen wir allerdings nicht den aktuellen Wert aus, also **Value**, sondern den aktuell angezeigten Inhalt aus der Eigenschaft **Text** (der aktuelle Inhalt wird erst nach dem Speichern des Inhalts zum **Value**):

```
Private Sub txtFormat_Change()
    Me!sfmFormatassistent.Form!txtFormatierterWert.Format _
        = Me!txtFormat.Text
End Sub
```

Modul mdlTools

Die Hilfsfunktion **GetObjectType** erwartet den Namen des zu untersuchenden Objekts als Parameter und soll den Typ in Form einer Konstanten der Auflistung **acObjectType** zurückliefern. Dazu öffnen wir die Systemtabelle **MSysObjects** der Datenbank, von der aus der Assistent geöffnet

```
Public Function GetObjectType(strObject As String) As AcObjectType
    Dim dbc As DAO.Database
    Dim rst As DAO.Recordset
    Set db = CodeDb
    Set rst = db.OpenRecordset("SELECT * FROM MSysObjects WHERE Name = '" & strObject & "'", dbOpenDynaset)
    If Not rst.EOF Then
        Select Case rst!Type
            Case 1 'Tabelle
                GetObjectType = acTable
            Case 5 'Abrfrage
                GetObjectType = acQuery
            Case -32768 'Formulare
                GetObjectType = acForm
            Case -32764 'Bericht
                GetObjectType = acReport
        End Select
    End If
End Function
```

Listing 3: Die Hilfsfunktion **GetObjectType**